BTRFS

unknown

Jun 21, 2022

OVERVIEW

CHAPTER

ONE

INTRODUCTION

BTRFS is a modern copy on write (COW) filesystem for Linux aimed at implementing advanced features while also focusing on fault tolerance, repair and easy administration. Its main features and benefits are:

- Snapshots which do not make the full copy of files
- Built-in volume management, support for software-based RAID 0, RAID 1, RAID 10 and others
- Self-healing - checksums for data and metadata, automatic detection of silent data corruptions

Feature overview:

- Extent based file storage
- 2^{64} byte == 16 EiB maximum file size (practical limit is 8 EiB due to Linux VFS)
- Space-efficient packing of small files
- Space-efficient indexed directories
- Dynamic inode allocation
- Writable snapshots, read-only snapshots
- Subvolumes (separate internal filesystem roots)
- Checksums on data and metadata (crc32c, xxhash, sha256, blake2b)
- Compression (ZLIB, LZO, ZSTD), heuristics
- **Integrated multiple device support**
  - File Striping
  - File Mirroring
  - File Striping+Mirroring
  - Single and Dual Parity implementations (experimental, not production-ready)
- SSD (flash storage) awareness (TRIM/Discard for reporting free blocks for reuse) and optimizations (e.g. avoiding unnecessary seek optimizations, sending writes in clusters, even if they are from unrelated files. This results in larger write operations and faster write throughput)
- Efficient incremental backup
- Background scrub process for finding and repairing errors of files with redundant copies
- Online filesystem defragmentation
- Offline filesystem check
- In-place conversion of existing ext2/3/4 and reiserfs file systems

- Seed devices. Create a (readonly) filesystem that acts as a template to seed other Btrfs filesystems. The original filesystem and devices are included as a readonly starting point for the new filesystem. Using copy on write, all modifications are stored on different devices; the original is unchanged.
- Subvolume-aware quota support
- **Send/receive of subvolume changes**
  - Efficient incremental filesystem mirroring
- Batch, or out-of-band deduplication (happens after writes, not during)
- Swapfile support
- Tree-checker, post-read and pre-write metadata verification
- Zoned mode support (SMR/ZBC/ZNS friendly allocation)

CHAPTER

TWO

MANUAL PAGES

2.1 btrfs(8)

2.1.1 SYNOPSIS

btrfs <command> [<args>]

2.1.2 DESCRIPTION

The **btrfs** utility is a toolbox for managing btrfs filesystems. There are command groups to work with subvolumes, devices, for whole filesystem or other specific actions. See section *COMMANDS*.

There are also standalone tools for some tasks like **btrfs-convert** or **btrfstune** that were separate historically and/or haven't been merged to the main utility. See section *STANDALONE TOOLS* for more details.

For other topics (mount options, etc) please refer to the separate manual page `btrfs(5)`.

2.1.3 COMMAND SYNTAX

Any command name can be shortened so long as the shortened form is unambiguous, however, it is recommended to use full command names in scripts. All command groups have their manual page named **btrfs-<group>**.

For example: it is possible to run **btrfs sub snaps** instead of **btrfs subvolume snapshot**. But **btrfs file s** is not allowed, because **file s** may be interpreted both as **filesystem show** and as **filesystem sync**.

If the command name is ambiguous, the list of conflicting options is printed.

For an overview of a given command use **btrfs command --help** or **btrfs [command...] --help --full** to print all available options.

2.1.4 COMMANDS

balance
: Balance btrfs filesystem chunks across single or several devices. See `btrfs-balance(8)` for details.

check
: Do off-line check on a btrfs filesystem. See `btrfs-check(8)` for details.

device
: Manage devices managed by btrfs, including add/delete/scan and so on. See `btrfs-device(8)` for details.

filesystem
Manage a btrfs filesystem, including label setting/sync and so on. See `btrfs-filesystem(8)` for details.

inspect-internal
Debug tools for developers/hackers. See `btrfs-inspect-internal(8)` for details.

property
Get/set a property from/to a btrfs object. See `btrfs-property(8)` for details.

qgroup
Manage quota group(qgroup) for btrfs filesystem. See `btrfs-qgroup(8)` for details.

quota
Manage quota on btrfs filesystem like enabling/rescan and etc. See `btrfs-quota(8)` and `btrfs-qgroup(8)` for details.

receive
Receive subvolume data from stdin/file for restore and etc. See `btrfs-receive(8)` for details.

replace
Replace btrfs devices. See `btrfs-replace(8)` for details.

rescue
Try to rescue damaged btrfs filesystem. See `btrfs-rescue(8)` for details.

restore
Try to restore files from a damaged btrfs filesystem. See `btrfs-restore(8)` for details.

scrub
Scrub a btrfs filesystem. See `btrfs-scrub(8)` for details.

send
Send subvolume data to stdout/file for backup and etc. See `btrfs-send(8)` for details.

subvolume
Create/delete/list/manage btrfs subvolume. See `btrfs-subvolume(8)` for details.

2.1.5 STANDALONE TOOLS

New functionality could be provided using a standalone tool. If the functionality proves to be useful, then the standalone tool is declared obsolete and its functionality is copied to the main tool. Obsolete tools are removed after a long (years) depreciation period.

Tools that are still in active use without an equivalent in **btrfs**:

btrfs-convert
in-place conversion from ext2/3/4 filesystems to btrfs

btrfstune
tweak some filesystem properties on a unmounted filesystem

btrfs-select-super
rescue tool to overwrite primary superblock from a spare copy

btrfs-find-root
rescue helper to find tree roots in a filesystem

Deprecated and obsolete tools:

btrfs-debug-tree
moved to **btrfs inspect-internal dump-tree**. Removed from source distribution.

btrfs-show-super
: moved to **btrfs inspect-internal dump-super**, standalone removed.

btrfs-zero-log
: moved to **btrfs rescue zero-log**, standalone removed.

For space-constrained environments, it's possible to build a single binary with functionality of several standalone tools. This is following the concept of busybox where the file name selects the functionality. This works for symlinks or hardlinks. The full list can be obtained by **btrfs help --box**.

2.1.6 EXIT STATUS

btrfs returns a zero exit status if it succeeds. Non zero is returned in case of failure.

2.1.7 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.1.8 SEE ALSO

`btrfs(5)`, `btrfs-balance(8)`, `btrfs-check(8)`, `btrfs-convert(8)`, `btrfs-device(8)`, `btrfs-filesystem(8)`, `btrfs-inspect-internal(8)`, `btrfs-property(8)`, `btrfs-qgroup(8)`, `btrfs-quota(8)`, `btrfs-receive(8)`, `btrfs-replace(8)`, `btrfs-rescue(8)`, `btrfs-restore(8)`, `btrfs-scrub(8)`, `btrfs-send(8)`, `btrfs-subvolume(8)`, `btrfstune(8)`, `mkfs.btrfs(8)`

2.2 btrfs-man5(5)

2.2.1 DESCRIPTION

This document describes topics related to BTRFS that are not specific to the tools. Currently covers:

1. mount options
2. filesystem features
3. checksum algorithms
4. compression
5. filesystem exclusive operations
6. filesystem limits
7. bootloader support
8. file attributes
9. zoned mode
10. control device
11. filesystems with multiple block group profiles
12. seeding device
13. raid56 status and recommended practices

14. storage model, hardware considerations

2.2.2 MOUNT OPTIONS

This section describes mount options specific to BTRFS. For the generic mount options please refer to `mount(8)` manpage. The options are sorted alphabetically (discarding the *no* prefix).

Note: Most mount options apply to the whole filesystem and only options in the first mounted subvolume will take effect. This is due to lack of implementation and may change in the future. This means that (for example) you can't set per-subvolume *nodatacow*, *nodatasum*, or *compress* using mount options. This should eventually be fixed, but it has proved to be difficult to implement correctly within the Linux VFS framework.

Mount options are processed in order, only the last occurrence of an option takes effect and may disable other options due to constraints (see eg. *nodatacow* and *compress*). The output of **mount** command shows which options have been applied.

acl, noacl
(default: on)

Enable/disable support for Posix Access Control Lists (ACLs). See the `acl(5)` manual page for more information about ACLs.

The support for ACL is build-time configurable (BTRFS_FS_POSIX_ACL) and mount fails if *acl* is requested but the feature is not compiled in.

autodefrag, noautodefrag
(since: 3.0, default: off)

Enable automatic file defragmentation. When enabled, small random writes into files (in a range of tens of kilobytes, currently it's 64KiB) are detected and queued up for the defragmentation process. Not well suited for large database workloads.

The read latency may increase due to reading the adjacent blocks that make up the range for defragmentation, successive write will merge the blocks in the new location.

> **Warning:** Defragmenting with Linux kernel versions < 3.9 or 3.14-rc2 as well as with Linux stable kernel versions 3.10.31, 3.12.12 or 3.13.4 will break up the reflinks of COW data (for example files copied with **cp --reflink**, snapshots or de-duplicated data). This may cause considerable increase of space usage depending on the broken up reflinks.

barrier, nobarrier
(default: on)

Ensure that all IO write operations make it through the device cache and are stored permanently when the filesystem is at its consistency checkpoint. This typically means that a flush command is sent to the device that will synchronize all pending data and ordinary metadata blocks, then writes the superblock and issues another flush.

The write flushes incur a slight hit and also prevent the IO block scheduler to reorder requests in a more effective way. Disabling barriers gets rid of that penalty but will most certainly lead to a corrupted filesystem in case of a crash or power loss. The ordinary metadata blocks could be yet unwritten at the time the new superblock is stored permanently, expecting that the block pointers to metadata were stored permanently before.

On a device with a volatile battery-backed write-back cache, the *nobarrier* option will not lead to filesystem corruption as the pending blocks are supposed to make it to the permanent storage.

check_int, check_int_data, check_int_print_mask=<value>
(since: 3.0, default: off)

These debugging options control the behavior of the integrity checking module (the BTRFS_FS_CHECK_INTEGRITY config option required). The main goal is to verify that all blocks from a given transaction period are properly linked.

check_int enables the integrity checker module, which examines all block write requests to ensure on-disk consistency, at a large memory and CPU cost.

check_int_data includes extent data in the integrity checks, and implies the *check_int* option.

check_int_print_mask takes a bitmask of BTRFSIC_PRINT_MASK_* values as defined in *fs/btrfs/check-integrity.c*, to control the integrity checker module behavior.

See comments at the top of *fs/btrfs/check-integrity.c* for more information.

clear_cache
Force clearing and rebuilding of the disk space cache if something has gone wrong. See also: *space_cache*.

commit=<seconds>
(since: 3.12, default: 30)

Set the interval of periodic transaction commit when data are synchronized to permanent storage. Higher interval values lead to larger amount of unwritten data, which has obvious consequences when the system crashes. The upper bound is not forced, but a warning is printed if it's more than 300 seconds (5 minutes). Use with care.

compress, compress=<type[:level]>, compress-force, compress-force=<type[:level]>
(default: off, level support since: 5.1)

Control BTRFS file data compression. Type may be specified as *zlib*, *lzo*, *zstd* or *no* (for no compression, used for remounting). If no type is specified, *zlib* is used. If *compress-force* is specified, then compression will always be attempted, but the data may end up uncompressed if the compression would make them larger.

Both *zlib* and *zstd* (since version 5.1) expose the compression level as a tunable knob with higher levels trading speed and memory (*zstd*) for higher compression ratios. This can be set by appending a colon and the desired level. Zlib accepts the range [1, 9] and zstd accepts [1, 15]. If no level is set, both currently use a default level of 3. The value 0 is an alias for the default level.

Otherwise some simple heuristics are applied to detect an incompressible file. If the first blocks written to a file are not compressible, the whole file is permanently marked to skip compression. As this is too simple, the *compress-force* is a workaround that will compress most of the files at the cost of some wasted CPU cycles on failed attempts. Since kernel 4.15, a set of heuristic algorithms have been improved by using frequency sampling, repeated pattern detection and Shannon entropy calculation to avoid that.

Note: If compression is enabled, *nodatacow* and *nodatasum* are disabled.

---

datacow, nodatacow
(default: on)

Enable data copy-on-write for newly created files. *Nodatacow* implies *nodatasum*, and disables *compression*. All files created under *nodatacow* are also set the NOCOW file attribute (see `chattr(1)`).

Note: If *nodatacow* or *nodatasum* are enabled, compression is disabled.

Updates in-place improve performance for workloads that do frequent overwrites, at the cost of potential partial writes, in case the write is interrupted (system crash, device failure).

datasum, nodatasum
(default: on)

Enable data checksumming for newly created files. *Datasum* implies *datacow*, ie. the normal mode of operation. All files created under *nodatasum* inherit the "no checksums" property, however there's no corresponding file attribute (see `chattr(1)`).

Note: If *nodatacow* or *nodatasum* are enabled, compression is disabled.

There is a slight performance gain when checksums are turned off, the corresponding metadata blocks holding the checksums do not need to updated. The cost of checksumming of the blocks in memory is much lower than the IO, modern CPUs feature hardware support of the checksumming algorithm.

degraded
(default: off)

Allow mounts with less devices than the RAID profile constraints require. A read-write mount (or remount) may fail when there are too many devices missing, for example if a stripe member is completely missing from RAID0.

Since 4.14, the constraint checks have been improved and are verified on the chunk level, not an the device level. This allows degraded mounts of filesystems with mixed RAID profiles for data and metadata, even if the device number constraints would not be satisfied for some of the profiles.

Example: metadata -- raid1, data -- single, devices -- /dev/sda, /dev/sdb

Suppose the data are completely stored on *sda*, then missing *sdb* will not prevent the mount, even if 1 missing device would normally prevent (any) *single* profile to mount. In case some of the data chunks are stored on *sdb*, then the constraint of single/data is not satisfied and the filesystem cannot be mounted.

device=<devicepath>
Specify a path to a device that will be scanned for BTRFS filesystem during mount. This is usually done automatically by a device manager (like udev) or using the **btrfs device scan** command (eg. run from the initial ramdisk). In cases where this is not possible the *device* mount option can help.

Note: Booting eg. a RAID1 system may fail even if all filesystem's *device* paths are provided as the actual device nodes may not be discovered by the system at that point.

---

discard, discard=sync, discard=async, nodiscard
(default: off, async support since: 5.6)

Enable discarding of freed file blocks. This is useful for SSD devices, thinly provisioned LUNs, or virtual machine images; however, every storage layer must support discard for it to work.

In the synchronous mode (*sync* or without option value), lack of asynchronous queued TRIM on the backing device TRIM can severely degrade performance, because a synchronous TRIM operation will be attempted instead. Queued TRIM requires newer than SATA revision 3.1 chipsets and devices.

The asynchronous mode (*async*) gathers extents in larger chunks before sending them to the devices for TRIM. The overhead and performance impact should be negligible compared to the previous mode and it's supposed to be the preferred mode if needed.

If it is not necessary to immediately discard freed blocks, then the `fstrim` tool can be used to discard all free blocks in a batch. Scheduling a TRIM during a period of low system activity will prevent latent interference with the performance of other operations. Also, a device may ignore the TRIM command if the range is too small, so running a batch discard has a greater probability of actually discarding the blocks.

enospc_debug, noenospc_debug
(default: off)

Enable verbose output for some ENOSPC conditions. It's safe to use but can be noisy if the system reaches near-full state.

fatal_errors=<action>
(since: 3.4, default: bug)

Action to take when encountering a fatal error.

bug
BUG() on a fatal error, the system will stay in the crashed state and may be still partially usable, but reboot is required for full operation

panic
panic() on a fatal error, depending on other system configuration, this may be followed by a reboot. Please refer to the documentation of kernel boot parameters, eg. *panic*, *oops* or *crashkernel*.

flushoncommit, noflushoncommit
(default: off)

This option forces any data dirtied by a write in a prior transaction to commit as part of the current commit, effectively a full filesystem sync.

This makes the committed state a fully consistent view of the file system from the application's perspective (i.e. it includes all completed file system operations). This was previously the behavior only when a snapshot was created.

When off, the filesystem is consistent but buffered writes may last more than one transaction commit.

fragment=<type>
(depends on compile-time option BTRFS_DEBUG, since: 4.4, default: off)

A debugging helper to intentionally fragment given *type* of block groups. The type can be *data*, *metadata* or *all*. This mount option should not be used outside of debugging environments and is not recognized if the kernel config option *BTRFS_DEBUG* is not enabled.

nologreplay
(default: off, even read-only)

The tree-log contains pending updates to the filesystem until the full commit. The log is replayed on next mount, this can be disabled by this option. See also *treelog*. Note that *nologreplay* is the same as *norecovery*.

> **Warning:** Currently, the tree log is replayed even with a read-only mount! To disable that behaviour, mount also with *nologreplay*.

max_inline=<bytes>
(default: min(2048, page size))

Specify the maximum amount of space, that can be inlined in a metadata b-tree leaf. The value is specified in bytes, optionally with a K suffix (case insensitive). In practice, this value is limited by the filesystem block size (named *sectorsize* at mkfs time), and memory page size of the system. In case of sectorsize limit, there's some space unavailable due to leaf headers. For example, a 4KiB sectorsize, maximum size of inline data is about 3900 bytes.

Inlining can be completely turned off by specifying 0. This will increase data block slack if file sizes are much smaller than block size but will reduce metadata consumption in return.

Note: The default value has changed to 2048 in kernel 4.6.

---

metadata_ratio=<value>
(default: 0, internal logic)

Specifies that 1 metadata chunk should be allocated after every *value* data chunks. Default behaviour depends on internal logic, some percent of unused metadata space is attempted to be maintained but is not always possible if there's not enough space left for chunk allocation. The option could be useful to override the internal logic in favor of the metadata allocation if the expected workload is supposed to be metadata intense (snapshots, reflinks, xattrs, inlined files).

norecovery
(since: 4.5, default: off)

Do not attempt any data recovery at mount time. This will disable *logreplay* and avoids other write operations. Note that this option is the same as *nologreplay*.

Note: The opposite option *recovery* used to have different meaning but was changed for consistency with other filesystems, where *norecovery* is used for skipping log replay. BTRFS does the same and in general will try to avoid any write operations.

---

rescan_uuid_tree
(since: 3.12, default: off)

Force check and rebuild procedure of the UUID tree. This should not normally be needed.

rescue
(since: 5.9)

Modes allowing mount with damaged filesystem structures.

- *usebackuproot* (since: 5.9, replaces standalone option *usebackuproot*)
- *nologreplay* (since: 5.9, replaces standalone option *nologreplay*)
- *ignorebadroots*, *ibadroots* (since: 5.11)
- *ignoredatacsums*, *idatacsums* (since: 5.11)
- *all* (since: 5.9)

skip_balance
(since: 3.3, default: off)

Skip automatic resume of an interrupted balance operation. The operation can later be resumed with **btrfs balance resume**, or the paused state can be removed with **btrfs balance cancel**. The default behaviour is to resume an interrupted balance immediately after a volume is mounted.

space_cache, space_cache=<version>, nospace_cache
(*nospace_cache* since: 3.2, *space_cache=v1* and *space_cache=v2* since 4.5, default: *space_cache=v1*)

Options to control the free space cache. The free space cache greatly improves performance when reading block group free space into memory. However, managing the space cache consumes some resources, including a small amount of disk space.

There are two implementations of the free space cache. The original one, referred to as *v1*, is the safe default. The *v1* space cache can be disabled at mount time with *nospace_cache* without clearing.

On very large filesystems (many terabytes) and certain workloads, the performance of the *v1* space cache may degrade drastically. The *v2* implementation, which adds a new b-tree called the free space tree, addresses this issue. Once enabled, the *v2* space cache will always be used and cannot be disabled unless it is cleared. Use *clear_cache,space_cache=v1* or *clear_cache,nospace_cache* to do so. If *v2* is enabled, kernels without *v2* support will only be able to mount the filesystem in read-only mode.

The `btrfs-check(8)` and `mkfs.btrfs(8)` commands have full *v2* free space cache support since v4.19.

If a version is not explicitly specified, the default implementation will be chosen, which is *v1*.

ssd, ssd_spread, nossd, nossd_spread
(default: SSD autodetected)

Options to control SSD allocation schemes. By default, BTRFS will enable or disable SSD optimizations depending on status of a device with respect to rotational or non-rotational type. This is determined by the contents of */sys/block/DEV/queue/rotational*). If it is 0, the *ssd* option is turned on. The option *nossd* will disable the autodetection.

The optimizations make use of the absence of the seek penalty that's inherent for the rotational devices. The blocks can be typically written faster and are not offloaded to separate threads.

Note: Since 4.14, the block layout optimizations have been dropped. This used to help with first generations of SSD devices. Their FTL (flash translation layer) was not effective and the optimization was supposed to improve the wear by better aligning blocks. This is no longer true with modern SSD devices and the optimization had no real benefit. Furthermore it caused increased fragmentation. The layout tuning has been kept intact for the option *ssd_spread*.

The *ssd_spread* mount option attempts to allocate into bigger and aligned chunks of unused space, and may perform better on low-end SSDs. *ssd_spread* implies *ssd*, enabling all other SSD heuristics as well. The option *nossd* will disable all SSD options while *nossd_spread* only disables *ssd_spread*.

subvol=<path>
Mount subvolume from *path* rather than the toplevel subvolume. The *path* is always treated as relative to the toplevel subvolume. This mount option overrides the default subvolume set for the given filesystem.

subvolid=<subvolid>
Mount subvolume specified by a *subvolid* number rather than the toplevel subvolume. You can use **btrfs subvolume list** of **btrfs subvolume show** to see subvolume ID numbers. This mount option overrides the default subvolume set for the given filesystem.

Note: If both *subvolid* and *subvol* are specified, they must point at the same subvolume, otherwise the mount will fail.

---

thread_pool=<number>
(default: min(NRCPUS + 2, 8))

The number of worker threads to start. NRCPUS is number of on-line CPUs detected at the time of mount. Small number leads to less parallelism in processing data and metadata, higher numbers could lead to a performance hit due to increased locking contention, process scheduling, cache-line bouncing or costly data transfers between local CPU memories.

treelog, notreelog
(default: on)

Enable the tree logging used for *fsync* and *O_SYNC* writes. The tree log stores changes without the need of a full filesystem sync. The log operations are flushed at sync and transaction commit. If the system crashes between two such syncs, the pending tree log operations are replayed during mount.

Warning: Currently, the tree log is replayed even with a read-only mount! To disable that behaviour, also mount with *nologreplay*.

The tree log could contain new files/directories, these would not exist on a mounted filesystem if the log is not replayed.

usebackuproot
(since: 4.6, default: off)

Enable autorecovery attempts if a bad tree root is found at mount time. Currently this scans a backup list of several previous tree roots and tries to use the first readable. This can be used with read-only mounts as well.

Note: This option has replaced *recovery*.

---

user_subvol_rm_allowed
(default: off)

Allow subvolumes to be deleted by their respective owner. Otherwise, only the root user can do that.

Note: Historically, any user could create a snapshot even if he was not owner of the source subvolume, the subvolume deletion has been restricted for that reason. The subvolume creation has been restricted but this mount option is still required. This is a usability issue. Since 4.18, the `rmdir(2)` syscall can delete an empty subvolume just like an ordinary directory. Whether this is possible can be detected at runtime, see *rmdir_subvol* feature in *FILESYSTEM FEATURES*.

DEPRECATED MOUNT OPTIONS

List of mount options that have been removed, kept for backward compatibility.

recovery
(since: 3.2, default: off, deprecated since: 4.5)

Note: This option has been replaced by *usebackuproot* and should not be used but will work on 4.5+ kernels.

---

inode_cache, noinode_cache
(removed in: 5.11, since: 3.0, default: off)

Note: The functionality has been removed in 5.11, any stale data created by previous use of the *inode_cache* option can be removed by **btrfs check --clear-ino-cache**.

NOTES ON GENERIC MOUNT OPTIONS

Some of the general mount options from `mount(8)` that affect BTRFS and are worth mentioning.

noatime
under read intensive work-loads, specifying *noatime* significantly improves performance because no new access time information needs to be written. Without this option, the default is *relatime*, which only reduces the number of inode atime updates in comparison to the traditional *strictatime*. The worst case for atime updates under 'relatime' occurs when many files are read whose atime is older than 24 h and which are freshly snapshotted. In that case the atime is updated and COW happens - for each file - in bulk. See also https://lwn.net/Articles/499293/ - *Atime and btrfs: a bad combination? (LWN, 2012-05-31)*.

Note that *noatime* may break applications that rely on atime uptimes like the venerable Mutt (unless you use maildir mailboxes).

2.2.3 FILESYSTEM FEATURES

The basic set of filesystem features gets extended over time. The backward compatibility is maintained and the features are optional, need to be explicitly asked for so accidental use will not create incompatibilities.

There are several classes and the respective tools to manage the features:

at mkfs time only
This is namely for core structures, like the b-tree nodesize or checksum algorithm, see `mkfs.btrfs(8)` for more details.

after mkfs, on an unmounted filesystem::
Features that may optimize internal structures or add new structures to support new functionality, see `btrfstune(8)`. The command **btrfs inspect-internal dump-super /dev/sdx** will dump a superblock, you can map the value of *incompat_flags* to the features listed below

after mkfs, on a mounted filesystem
The features of a filesystem (with a given UUID) are listed in */sys/fs/btrfs/UUID/features/*, one file per feature. The status is stored inside the file. The value *1* is for enabled and active, while *0* means the feature was enabled at mount time but turned off afterwards.

Whether a particular feature can be turned on a mounted filesystem can be found in the directory */sys/fs/btrfs/features/*, one file per feature. The value *1* means the feature can be enabled.

List of features (see also `mkfs.btrfs(8)` section *FILESYSTEM FEATURES*):

big_metadata
(since: 3.4)

the filesystem uses *nodesize* for metadata blocks, this can be bigger than the page size

compress_lzo
(since: 2.6.38)

the *lzo* compression has been used on the filesystem, either as a mount option or via **btrfs filesystem defrag**.

compress_zstd
(since: 4.14)

the *zstd* compression has been used on the filesystem, either as a mount option or via **btrfs filesystem defrag**.

default_subvol
(since: 2.6.34)

the default subvolume has been set on the filesystem

extended_iref
(since: 3.7)

increased hardlink limit per file in a directory to 65536, older kernels supported a varying number of hardlinks depending on the sum of all file name sizes that can be stored into one metadata block

free_space_tree
(since: 4.5)

free space representation using a dedicated b-tree, successor of v1 space cache

metadata_uuid
(since: 5.0)

the main filesystem UUID is the metadata_uuid, which stores the new UUID only in the superblock while all metadata blocks still have the UUID set at mkfs time, see `btrfstune(8)` for more

mixed_backref
(since: 2.6.31)

the last major disk format change, improved backreferences, now default

mixed_groups
(since: 2.6.37)

mixed data and metadata block groups, ie. the data and metadata are not separated and occupy the same block groups, this mode is suitable for small volumes as there are no constraints how the remaining space should be used (compared to the split mode, where empty metadata space cannot be used for data and vice versa)

on the other hand, the final layout is quite unpredictable and possibly highly fragmented, which means worse performance

no_holes
(since: 3.14)

improved representation of file extents where holes are not explicitly stored as an extent, saves a few percent of metadata if sparse files are used

raid1c34
(since: 5.5)

extended RAID1 mode with copies on 3 or 4 devices respectively

raid56
(since: 3.9)

the filesystem contains or contained a raid56 profile of block groups

rmdir_subvol
(since: 4.18)

indicate that `rmdir(2)` syscall can delete an empty subvolume just like an ordinary directory. Note that this feature only depends on the kernel version.

skinny_metadata
(since: 3.10)

reduced-size metadata for extent references, saves a few percent of metadata

send_stream_version
(since: 5.10)

number of the highest supported send stream version

supported_checksums
(since: 5.5)

list of checksum algorithms supported by the kernel module, the respective modules or built-in implementing the algorithms need to be present to mount the filesystem, see *CHECKSUM ALGORITHMS*

supported_sectorsizes
(since: 5.13)

list of values that are accepted as sector sizes (**mkfs.btrfs --sectorsize**) by the running kernel

supported_rescue_options
(since: 5.11)

list of values for the mount option *rescue* that are supported by the running kernel, see `btrfs(5)`

zoned
(since: 5.12)

zoned mode is allocation/write friendly to host-managed zoned devices, allocation space is partitioned into fixed-size zones that must be updated sequentially, see *ZONED MODE*

2.2.4 SWAPFILE SUPPORT

A swapfile is file-backed memory that the system uses to temporarily offload the RAM. It is supported since kernel 5.0. Use `swapon(8)` to activate the swapfile. There are some limitations of the implementation in BTRFS and linux swap subsystem:

- filesystem - must be only single device
- filesystem - must have only *single* data profile
- swapfile - the containing subvolume cannot be snapshotted
- swapfile - must be preallocated
- swapfile - must be nodatacow (ie. also nodatasum)
- swapfile - must not be compressed

The limitations come namely from the COW-based design and mapping layer of blocks that allows the advanced features like relocation and multi-device filesystems. However, the swap subsystem expects simpler mapping and no background changes of the file blocks once they've been attached to swap.

With active swapfiles, the following whole-filesystem operations will skip swapfile extents or may fail:

- balance - block groups with swapfile extents are skipped and reported, the rest will be processed normally
- resize grow - unaffected
- resize shrink - works as long as the extents are outside of the shrunk range
- device add - a new device does not interfere with existing swapfile and this operation will work, though no new swapfile can be activated afterwards
- device delete - if the device has been added as above, it can be also deleted
- device replace - ditto

When there are no active swapfiles and a whole-filesystem exclusive operation is running (eg. balance, device delete, shrink), the swapfiles cannot be temporarily activated. The operation must finish first.

To create and activate a swapfile run the following commands:

```
# truncate -s 0 swapfile
# chattr +C swapfile
# fallocate -l 2G swapfile
# chmod 0600 swapfile
# mkswap swapfile
# swapon swapfile
```

Please note that the UUID returned by the *mkswap* utility identifies the swap "filesystem" and because it's stored in a file, it's not generally visible and usable as an identifier unlike if it was on a block device.

The file will appear in */proc/swaps*:

```
# cat /proc/swaps
Filename          Type          Size          Used      Priority
/path/swapfile    file          2097152       0         -2
--------------------
```

The swapfile can be created as one-time operation or, once properly created, activated on each boot by the **swapon -a** command (usually started by the service manager). Add the following entry to */etc/fstab*, assuming the filesystem that provides the */path* has been already mounted at this point. Additional mount options relevant for the swapfile can be set too (like priority, not the BTRFS mount options).

```
/path/swapfile        none        swap        defaults      0 0
```

2.2.5 CHECKSUM ALGORITHMS

Data and metadata are checksummed by default, the checksum is calculated before write and verifed after reading the blocks from devices. The whole metadata block has a checksum stored inline in the b-tree node header, each data block has a detached checksum stored in the checksum tree.

There are several checksum algorithms supported. The default and backward compatible is *crc32c*. Since kernel 5.5 there are three more with different characteristics and trade-offs regarding speed and strength. The following list may help you to decide which one to select.

CRC32C (32bit digest)
: default, best backward compatibility, very fast, modern CPUs have instruction-level support, not collision-resistant but still good error detection capabilities

XXHASH (64bit digest)
: can be used as CRC32C successor, very fast, optimized for modern CPUs utilizing instruction pipelining, good collision resistance and error detection

SHA256 (256bit digest)::
: a cryptographic-strength hash, relatively slow but with possible CPU instruction acceleration or specialized hardware cards, FIPS certified and in wide use

BLAKE2b (256bit digest)
: a cryptographic-strength hash, relatively fast with possible CPU acceleration using SIMD extensions, not standardized but based on BLAKE which was a SHA3 finalist, in wide use, the algorithm used is BLAKE2b-256 that's optimized for 64bit platforms

The *digest size* affects overall size of data block checksums stored in the filesystem. The metadata blocks have a fixed area up to 256 bits (32 bytes), so there's no increase. Each data block has a separate checksum stored, with additional overhead of the b-tree leaves.

Approximate relative performance of the algorithms, measured against CRC32C using reference software implementations on a 3.5GHz intel CPU:

Digest	Cycles/4KiB	Ratio	Implementation
CRC32C	1700	1.00	CPU instruction
XXHASH	2500	1.44	reference impl.
SHA256	105000	61	reference impl.
SHA256	36000	21	libgcrypt/AVX2
SHA256	63000	37	libsodium/AVX2
BLAKE2b	22000	13	reference impl.
BLAKE2b	19000	11	libgcrypt/AVX2
BLAKE2b	19000	11	libsodium/AVX2

Many kernels are configured with SHA256 as built-in and not as a module. The accelerated versions are however provided by the modules and must be loaded explicitly (**modprobe sha256**) before mounting the filesystem to make use of them. You can check in */sys/fs/btrfs/FSID/checksum* which one is used. If you see *sha256-generic*, then you may want to unmount and mount the filesystem again, changing that on a mounted filesystem is not possible. Check the file */proc/crypto*, when the implementation is built-in, you'd find

```
name         : sha256
driver       : sha256-generic
module       : kernel
priority     : 100
...
```

while accelerated implementation is e.g.

```
name         : sha256
driver       : sha256-avx2
module       : sha256_ssse3
priority     : 170
...
```

2.2.6 COMPRESSION

Btrfs supports transparent file compression. There are three algorithms available: ZLIB, LZO and ZSTD (since v4.14), with various levels. The compression happens on the level of file extents and the algorithm is selected by file property, mount option or by a defrag command. You can have a single btrfs mount point that has some files that are uncompressed, some that are compressed with LZO, some with ZLIB, for instance (though you may not want it that way, it is supported).

Once the compression is set, all newly written data will be compressed, ie. existing data are untouched. Data are split into smaller chunks (128KiB) before compression to make random rewrites possible without a high performance hit. Due to the increased number of extents the metadata consumption is higher. The chunks are compressed in parallel.

The algorithms can be characterized as follows regarding the speed/ratio trade-offs:

ZLIB

- slower, higher compression ratio
- levels: 1 to 9, mapped directly, default level is 3
- good backward compatibility

LZO

- faster compression and decompression than zlib, worse compression ratio, designed to be fast
- no levels
- good backward compatibility

ZSTD

- compression comparable to zlib with higher compression/decompression speeds and different ratio
- levels: 1 to 15, mapped directly (higher levels are not available)
- since 4.14, levels since 5.1

The differences depend on the actual data set and cannot be expressed by a single number or recommendation. Higher levels consume more CPU time and may not bring a significant improvement, lower levels are close to real time.

2.2.7 How to enable compression

Typically the compression can be enabled on the whole filesystem, specified for the mount point. Note that the compression mount options are shared among all mounts of the same filesystem, either bind mounts or subvolume mounts. Please refer to section *MOUNT OPTIONS*.

```
$ mount -o compress=zstd /dev/sdx /mnt
```

This will enable the `zstd` algorithm on the default level (which is 3). The level can be specified manually too like `zstd:3`. Higher levels compress better at the cost of time. This in turn may cause increased write latency, low levels are suitable for real-time compression and on reasonably fast CPU don't cause noticeable performance drops.

```
$ btrfs filesystem defrag -czstd file
```

The command above will start defragmentation of the whole *file* and apply the compression, regardless of the mount option. (Note: specifying level is not yet implemented). The compression algorithm is not persistent and applies only to the defragmentation command, for any other writes other compression settings apply.

Persistent settings on a per-file basis can be set in two ways:

```
$ chattr +c file
$ btrfs property set file compression zstd
```

The first command is using legacy interface of file attributes inherited from ext2 filesystem and is not flexible, so by default the *zlib* compression is set. The other command sets a property on the file with the given algorithm. (Note: setting level that way is not yet implemented.)

2.2.8 Compression levels

The level support of ZLIB has been added in v4.14, LZO does not support levels (the kernel implementation provides only one), ZSTD level support has been added in v5.1.

There are 9 levels of ZLIB supported (1 to 9), mapping 1:1 from the mount option to the algorithm defined level. The default is level 3, which provides the reasonably good compression ratio and is still reasonably fast. The difference in compression gain of levels 7, 8 and 9 is comparable but the higher levels take longer.

The ZSTD support includes levels 1 to 15, a subset of full range of what ZSTD provides. Levels 1-3 are real-time, 4-8 slower with improved compression and 9-15 try even harder though the resulting size may not be significantly improved.

Level 0 always maps to the default. The compression level does not affect compatibility.

2.2.9 Incompressible data

Files with already compressed data or with data that won't compress well with the CPU and memory constraints of the kernel implementations are using a simple decision logic. If the first portion of data being compressed is not smaller than the original, the compression of the file is disabled -- unless the filesystem is mounted with *compress-force*. In that case compression will always be attempted on the file only to be later discarded. This is not optimal and subject to optimizations and further development.

If a file is identified as incompressible, a flag is set (*NOCOMPRESS*) and it's sticky. On that file compression won't be performed unless forced. The flag can be also set by **chattr +m** (since e2fsprogs 1.46.2) or by properties with value *no* or *none*. Empty value will reset it to the default that's currently applicable on the mounted filesystem.

There are two ways to detect incompressible data:

- actual compression attempt - data are compressed, if the result is not smaller, it's discarded, so this depends on the algorithm and level
- pre-compression heuristics - a quick statistical evaluation on the data is performed and based on the result either compression is performed or skipped, the NOCOMPRESS bit is not set just by the heuristic, only if the compression algorithm does not make an improvement

```
$ lsattr file
----------------------m file
```

Using the forcing compression is not recommended, the heuristics are supposed to decide that and compression algorithms internally detect incompressible data too.

2.2.10 Pre-compression heuristics

The heuristics aim to do a few quick statistical tests on the compressed data in order to avoid probably costly compression that would turn out to be inefficient. Compression algorithms could have internal detection of incompressible data too but this leads to more overhead as the compression is done in another thread and has to write the data anyway. The heuristic is read-only and can utilize cached memory.

The tests performed based on the following: data sampling, long repeated pattern detection, byte frequency, Shannon entropy.

2.2.11 Compatibility

Compression is done using the COW mechanism so it's incompatible with *nodatacow*. Direct IO works on compressed files but will fall back to buffered writes and leads to recompression. Currently *nodatasum* and compression don't work together.

The compression algorithms have been added over time so the version compatibility should be also considered, together with other tools that may access the compressed data like bootloaders.

2.2.12 FILESYSTEM EXCLUSIVE OPERATIONS

There are several operations that affect the whole filesystem and cannot be run in parallel. Attempt to start one while another is running will fail (see exceptions below).

Since kernel 5.10 the currently running operation can be obtained from */sys/fs/UUID/exclusive_operation* with following values and operations:

- balance
- balance paused (since 5.17)
- device add
- device delete
- device replace
- resize
- swapfile activate
- none

Enqueuing is supported for several btrfs subcommands so they can be started at once and then serialized.

There's an exception when a paused balance allows to start a device add operation as they don't really collide and this can be used to add more space for the balance to finish.

2.2.13 FILESYSTEM LIMITS

maximum file name length
255

maximum symlink target length
depends on the *nodesize* value, for 4KiB it's 3949 bytes, for larger nodesize it's 4095 due to the system limit PATH_MAX

The symlink target may not be a valid path, ie. the path name components can exceed the limits (NAME_MAX), there's no content validation at `symlink(3)` creation.

maximum number of inodes
2^{64} but depends on the available metadata space as the inodes are created dynamically

inode numbers
minimum number: 256 (for subvolumes), regular files and directories: 257

maximum file length
inherent limit of btrfs is 2^{64} (16 EiB) but the linux VFS limit is 2^{63} (8 EiB)

maximum number of subvolumes
the subvolume ids can go up to 2^{64} but the number of actual subvolumes depends on the available metadata space, the space consumed by all subvolume metadata includes bookkeeping of shared extents can be large (MiB, GiB)

maximum number of hardlinks of a file in a directory
65536 when the *extref* feature is turned on during mkfs (default), roughly 100 otherwise

minimum filesystem size
the minimal size of each device depends on the *mixed-bg* feature, without that (the default) it's about 109MiB, with mixed-bg it's is 16MiB

2.2.14 BOOTLOADER SUPPORT

GRUB2 (https://www.gnu.org/software/grub) has the most advanced support of booting from BTRFS with respect to features.

U-boot (https://www.denx.de/wiki/U-Boot/) has decent support for booting but not all BTRFS features are implemented, check the documentation.

EXTLINUX (from the https://syslinux.org project) can boot but does not support all features. Please check the upstream documentation before you use it.

The first 1MiB on each device is unused with the exception of primary superblock that is on the offset 64KiB and spans 4KiB.

2.2.15 FILE ATTRIBUTES

The btrfs filesystem supports setting file attributes or flags. Note there are old and new interfaces, with confusing names. The following list should clarify that:

- *attributes*: `chattr(1)` or `lsattr(1)` utilities (the ioctls are FS_IOC_GETFLAGS and FS_IOC_SETFLAGS), due to the ioctl names the attributes are also called flags
- *xflags*: to distinguish from the previous, it's extended flags, with tunable bits similar to the attributes but extensible and new bits will be added in the future (the ioctls are FS_IOC_FSGETXATTR and FS_IOC_FSSETXATTR but they are not related to extended attributes that are also called xattrs), there's no standard tool to change the bits, there's support in `xfs_io(8)` as command **xfs_io -c chattr**

Attributes

a

append only, new writes are always written at the end of the file

A

no atime updates

c

compress data, all data written after this attribute is set will be compressed. Please note that compression is also affected by the mount options or the parent directory attributes.

When set on a directory, all newly created files will inherit this attribute. This attribute cannot be set with 'm' at the same time.

C

no copy-on-write, file data modifications are done in-place

When set on a directory, all newly created files will inherit this attribute.

Note: Due to implementation limitations, this flag can be set/unset only on empty files.

---

d

no dump, makes sense with 3rd party tools like `dump(8)`, on BTRFS the attribute can be set/unset but no other special handling is done

D

synchronous directory updates, for more details search `open(2)` for *O_SYNC* and *O_DSYNC*

i

immutable, no file data and metadata changes allowed even to the root user as long as this attribute is set (obviously the exception is unsetting the attribute)

m

no compression, permanently turn off compression on the given file. Any compression mount options will not affect this file. (`chattr` support added in 1.46.2)

When set on a directory, all newly created files will inherit this attribute. This attribute cannot be set with *c* at the same time.

S

synchronous updates, for more details search `open(2)` for *O_SYNC* and *O_DSYNC*

No other attributes are supported. For the complete list please refer to the `chattr(1)` manual page.

XFLAGS

There's overlap of letters assigned to the bits with the attributes, this list refers to what `xfs_io(8)` provides:

i
: *immutable*, same as the attribute

a
: *append only*, same as the attribute

s
: *synchronous updates*, same as the attribute *S*

A
: *no atime updates*, same as the attribute

d
: *no dump*, same as the attribute

2.2.16 ZONED MODE

Since version 5.12 btrfs supports so called *zoned mode*. This is a special on-disk format and allocation/write strategy that's friendly to zoned devices. In short, a device is partitioned into fixed-size zones and each zone can be updated by append-only manner, or reset. As btrfs has no fixed data structures, except the super blocks, the zoned mode only requires block placement that follows the device constraints. You can learn about the whole architecture at https://zonedstorage.io .

The devices are also called SMR/ZBC/ZNS, in *host-managed* mode. Note that there are devices that appear as non-zoned but actually are, this is *drive-managed* and using zoned mode won't help.

The zone size depends on the device, typical sizes are 256MiB or 1GiB. In general it must be a power of two. Emulated zoned devices like *null_blk* allow to set various zone sizes.

Requirements, limitations

- all devices must have the same zone size
- maximum zone size is 8GiB
- minimum zone size is 4MiB
- mixing zoned and non-zoned devices is possible, the zone writes are emulated, but this is namely for testing
- **the super block is handled in a special way and is at different locations than on a non-zoned filesystem:**
  - primary: 0B (and the next two zones)
  - secondary: 512GiB (and the next two zones)
  - tertiary: 4TiB (4096GiB, and the next two zones)

Incompatible features

The main constraint of the zoned devices is lack of in-place update of the data. This is inherently incompatibile with some features:

- nodatacow - overwrite in-place, cannot create such files
- fallocate - preallocating space for in-place first write
- mixed-bg - unordered writes to data and metadata, fixing that means using separate data and metadata block groups
- booting - the zone at offset 0 contains superblock, resetting the zone would destroy the bootloader data

Initial support lacks some features but they're planned:

- only single profile is supported
- fstrim - due to dependency on free space cache v1

Super block

As said above, super block is handled in a special way. In order to be crash safe, at least one zone in a known location must contain a valid superblock. This is implemented as a ring buffer in two consecutive zones, starting from known offsets 0B, 512GiB and 4TiB.

The values are different than on non-zoned devices. Each new super block is appended to the end of the zone, once it's filled, the zone is reset and writes continue to the next one. Looking up the latest super block needs to read offsets of both zones and determine the last written version.

The amount of space reserved for super block depends on the zone size. The secondary and tertiary copies are at distant offsets as the capacity of the devices is expected to be large, tens of terabytes. Maximum zone size supported is 8GiB, which would mean that eg. offset 0-16GiB would be reserved just for the super block on a hypothetical device of that zone size. This is wasteful but required to guarantee crash safety.

2.2.17 CONTROL DEVICE

There's a character special device */dev/btrfs-control* with major and minor numbers 10 and 234 (the device can be found under the 'misc' category).

```
$ ls -l /dev/btrfs-control
crw------- 1 root root 10, 234 Jan  1 12:00 /dev/btrfs-control
```

The device accepts some ioctl calls that can perform following actions on the filesystem module:

- scan devices for btrfs filesystem (ie. to let multi-device filesystems mount automatically) and register them with the kernel module
- similar to scan, but also wait until the device scanning process is finished for a given filesystem
- get the supported features (can be also found under */sys/fs/btrfs/features*)

The device is created when btrfs is initialized, either as a module or a built-in functionality and makes sense only in connection with that. Running eg. mkfs without the module loaded will not register the device and will probably warn about that.

In rare cases when the module is loaded but the device is not present (most likely accidentally deleted), it's possible to recreate it by

```
# mknod --mode=600 /dev/btrfs-control c 10 234
```

or (since 5.11) by a convenience command

```
# btrfs rescue create-control-device
```

The control device is not strictly required but the device scanning will not work and a workaround would need to be used to mount a multi-device filesystem. The mount option *device* can trigger the device scanning during mount, see also **btrfs device scan**.

2.2.18 FILESYSTEM WITH MULTIPLE PROFILES

It is possible that a btrfs filesystem contains multiple block group profiles of the same type. This could happen when a profile conversion using balance filters is interrupted (see `btrfs-balance(8)`). Some **btrfs** commands perform a test to detect this kind of condition and print a warning like this:

```
WARNING: Multiple block group profiles detected, see 'man btrfs(5)'.
WARNING:   Data: single, raid1
WARNING:   Metadata: single, raid1
```

The corresponding output of **btrfs filesystem df** might look like:

```
WARNING: Multiple block group profiles detected, see 'man btrfs(5)'.
WARNING:   Data: single, raid1
WARNING:   Metadata: single, raid1
Data, RAID1: total=832.00MiB, used=0.00B
Data, single: total=1.63GiB, used=0.00B
System, single: total=4.00MiB, used=16.00KiB
Metadata, single: total=8.00MiB, used=112.00KiB
Metadata, RAID1: total=64.00MiB, used=32.00KiB
GlobalReserve, single: total=16.25MiB, used=0.00B
```

There's more than one line for type *Data* and *Metadata*, while the profiles are *single* and *RAID1*.

This state of the filesystem OK but most likely needs the user/administrator to take an action and finish the interrupted tasks. This cannot be easily done automatically, also the user knows the expected final profiles.

In the example above, the filesystem started as a single device and *single* block group profile. Then another device was added, followed by balance with *convert=raid1* but for some reason hasn't finished. Restarting the balance with *convert=raid1* will continue and end up with filesystem with all block group profiles *RAID1*.

Note: If you're familiar with balance filters, you can use *convert=raid1,profiles=single,soft*, which will take only the unconverted *single* profiles and convert them to *raid1*. This may speed up the conversion as it would not try to rewrite the already convert *raid1* profiles.

Having just one profile is desired as this also clearly defines the profile of newly allocated block groups, otherwise this depends on internal allocation policy. When there are multiple profiles present, the order of selection is RAID6, RAID5, RAID10, RAID1, RAID0 as long as the device number constraints are satisfied.

Commands that print the warning were chosen so they're brought to user attention when the filesystem state is being changed in that regard. This is: **device add**, **device delete**, **balance cancel**, **balance pause**. Commands that report space usage: **filesystem df**, **device usage**. The command **filesystem usage** provides a line in the overall summary:

```
Multiple profiles:                yes (data, metadata)
```

2.2.19 SEEDING DEVICE

The COW mechanism and multiple devices under one hood enable an interesting concept, called a seeding device: extending a read-only filesystem on a device with another device that captures all writes. For example imagine an immutable golden image of an operating system enhanced with another device that allows to use the data from the golden image and normal operation. This idea originated on CD-ROMs with base OS and allowing to use them for live systems, but this became obsolete. There are technologies providing similar functionality, like *unionmount*, *overlayfs* or *qcow2* image snapshot.

The seeding device starts as a normal filesystem, once the contents is ready, **btrfstune -S 1** is used to flag it as a seeding device. Mounting such device will not allow any writes, except adding a new device by **btrfs device add**. Then the filesystem can be remounted as read-write.

Given that the filesystem on the seeding device is always recognized as read-only, it can be used to seed multiple filesystems from one device at the same time. The UUID that is normally attached to a device is automatically changed to a random UUID on each mount.

Once the seeding device is mounted, it needs the writable device. After adding it, something like **remount -o remount,rw /path** makes the filesystem at */path* ready for use. The simplest use case is to throw away all changes by unmounting the filesystem when convenient.

Alternatively, deleting the seeding device from the filesystem can turn it into a normal filesystem, provided that the writable device can also contain all the data from the seeding device.

The seeding device flag can be cleared again by **btrfstune -f -S 0**, eg. allowing to update with newer data but please note that this will invalidate all existing filesystems that use this particular seeding device. This works for some use cases, not for others, and the forcing flag to the command is mandatory to avoid accidental mistakes.

Example how to create and use one seeding device:

```
# mkfs.btrfs /dev/sda
# mount /dev/sda /mnt/mnt1
... fill mnt1 with data
# umount /mnt/mnt1

# btrfstune -S 1 /dev/sda

# mount /dev/sda /mnt/mnt1
# btrfs device add /dev/sdb /mnt/mnt1
# mount -o remount,rw /mnt/mnt1
... /mnt/mnt1 is now writable
```

Now */mnt/mnt1* can be used normally. The device */dev/sda* can be mounted again with a another writable device:

```
# mount /dev/sda /mnt/mnt2
# btrfs device add /dev/sdc /mnt/mnt2
# mount -o remount,rw /mnt/mnt2
... /mnt/mnt2 is now writable
```

The writable device (*/dev/sdb*) can be decoupled from the seeding device and used independently:

```
# btrfs device delete /dev/sda /mnt/mnt1
```

As the contents originated in the seeding device, it's possible to turn */dev/sdb* to a seeding device again and repeat the whole process.

A few things to note:

- it's recommended to use only single device for the seeding device, it works for multiple devices but the *single* profile must be used in order to make the seeding device deletion work
- block group profiles *single* and *dup* support the use cases above
- the label is copied from the seeding device and can be changed by **btrfs filesystem label**
- each new mount of the seeding device gets a new random UUID

Chained seeding devices

Though it's not recommended and is rather an obscure and untested use case, chaining seeding devices is possible. In the first example, the writable device */dev/sdb* can be turned onto another seeding device again, depending on the unchanged seeding device */dev/sda*. Then using */dev/sdb* as the primary seeding device it can be extended with another writable device, say */dev/sdd*, and it continues as before as a simple tree structure on devices.

```
# mkfs.btrfs /dev/sda
# mount /dev/sda /mnt/mnt1
... fill mnt1 with data
# umount /mnt/mnt1

# btrfstune -S 1 /dev/sda

# mount /dev/sda /mnt/mnt1
# btrfs device add /dev/sdb /mnt/mnt1
# mount -o remount,rw /mnt/mnt1
... /mnt/mnt1 is now writable
# umount /mnt/mnt1

# btrfstune -S 1 /dev/sdb

# mount /dev/sdb /mnt/mnt1
# btrfs device add /dev/sdc /mnt
# mount -o remount,rw /mnt/mnt1
... /mnt/mnt1 is now writable
# umount /mnt/mnt1
```

As a result we have:

- *sda* is a single seeding device, with its initial contents
- *sdb* is a seeding device but requires *sda*, the contents are from the time when *sdb* is made seeding, ie. contents of *sda* with any later changes
- *sdc* last writable, can be made a seeding one the same way as was *sdb*, preserving its contents and depending on *sda* and *sdb*

As long as the seeding devices are unmodified and available, they can be used to start another branch.

2.2.20 RAID56 STATUS AND RECOMMENDED PRACTICES

The RAID56 feature provides striping and parity over several devices, same as the traditional RAID5/6. There are some implementation and design deficiencies that make it unreliable for some corner cases and the feature **should not be used in production, only for evaluation or testing**. The power failure safety for metadata with RAID56 is not 100%.

Metadata

Do not use *raid5* nor *raid6* for metadata. Use *raid1* or *raid1c3* respectively.

The substitute profiles provide the same guarantees against loss of 1 or 2 devices, and in some respect can be an improvement. Recovering from one missing device will only need to access the remaining 1st or 2nd copy, that in general may be stored on some other devices due to the way RAID1 works on btrfs, unlike on a striped profile (similar to *raid0*) that would need all devices all the time.

The space allocation pattern and consumption is different (eg. on N devices): for *raid5* as an example, a 1GiB chunk is reserved on each device, while with *raid1* there's each 1GiB chunk stored on 2 devices. The consumption of each 1GiB of used metadata is then *N * 1GiB* for vs *2 * 1GiB*. Using *raid1* is also more convenient for balancing/converting to other profile due to lower requirement on the available chunk space.

Missing/incomplete support

When RAID56 is on the same filesystem with different raid profiles, the space reporting is inaccurate, eg. **df**, **btrfs filesystem df** or **btrfs filesystem usage**. When there's only a one profile per block group type (eg. raid5 for data) the reporting is accurate.

When scrub is started on a RAID56 filesystem, it's started on all devices that degrade the performance. The workaround is to start it on each device separately. Due to that the device stats may not match the actual state and some errors might get reported multiple times.

The *write hole* problem. An unclean shutdown could leave a partially written stripe in a state where the some stripe ranges and the parity are from the old writes and some are new. The information which is which is not tracked. Write journal is not implemented. Alternatively a full read-modify-write would make sure that a full stripe is always written, avoiding the write hole completely, but performance in that case turned out to be too bad for use.

The striping happens on all available devices (at the time the chunks were allocated), so in case a new device is added it may not be utilized immediately and would require a rebalance. A fixed configured stripe width is not implemented.

2.2.21 STORAGE MODEL, HARDWARE CONSIDERATIONS

Storage model

A storage model is a model that captures key physical aspects of data structure in a data store. A filesystem is the logical structure organizing data on top of the storage device.

The filesystem assumes several features or limitations of the storage device and utilizes them or applies measures to guarantee reliability. BTRFS in particular is based on a COW (copy on write) mode of writing, ie. not updating data in place but rather writing a new copy to a different location and then atomically switching the pointers.

In an ideal world, the device does what it promises. The filesystem assumes that this may not be true so additional mechanisms are applied to either detect misbehaving hardware or get valid data by other means. The devices may (and do) apply their own detection and repair mechanisms but we won't assume any.

The following assumptions about storage devices are considered (sorted by importance, numbers are for further reference):

1. atomicity of reads and writes of blocks/sectors (the smallest unit of data the device presents to the upper layers)
2. there's a flush command that instructs the device to forcibly order writes before and after the command; alternatively there's a barrier command that facilitates the ordering but may not flush the data
3. data sent to write to a given device offset will be written without further changes to the data and to the offset
4. writes can be reordered by the device, unless explicitly serialized by the flush command
5. reads and writes can be freely reordered and interleaved

The consistency model of BTRFS builds on these assumptions. The logical data updates are grouped, into a generation, written on the device, serialized by the flush command and then the super block is written ending the generation. All logical links among metadata comprising a consistent view of the data may not cross the generation boundary.

When things go wrong

No or partial atomicity of block reads/writes (1)

- *Problem*: a partial block contents is written (*torn write*), eg. due to a power glitch or other electronics failure during the read/write
- *Detection*: checksum mismatch on read
- *Repair*: use another copy or rebuild from multiple blocks using some encoding scheme

The flush command does not flush (2)

This is perhaps the most serious problem and impossible to mitigate by filesystem without limitations and design restrictions. What could happen in the worst case is that writes from one generation bleed to another one, while still letting the filesystem consider the generations isolated. Crash at any point would leave data on the device in an inconsistent state without any hint what exactly got written, what is missing and leading to stale metadata link information.

Devices usually honor the flush command, but for performance reasons may do internal caching, where the flushed data are not yet persistently stored. A power failure could lead to a similar scenario as above, although it's less likely that later writes would be written before the cached ones. This is beyond what a filesystem can take into account. Devices or controllers are usually equipped with batteries or capacitors to write the cache contents even after power is cut. (*Battery backed write cache*)

Data get silently changed on write (3)

Such thing should not happen frequently, but still can happen spuriously due the complex internal workings of devices or physical effects of the storage media itself.

- *Problem*: while the data are written atomically, the contents get changed
- *Detection*: checksum mismatch on read
- 'Repair*: use another copy or rebuild from multiple blocks using some encoding scheme

Data get silently written to another offset (3)

This would be another serious problem as the filesystem has no information when it happens. For that reason the measures have to be done ahead of time. This problem is also commonly called 'ghost write'.

The metadata blocks have the checksum embedded in the blocks, so a correct atomic write would not corrupt the checksum. It's likely that after reading such block the data inside would not be consistent with the rest. To rule that out there's embedded block number in the metadata block. It's the logical block number because this is what the logical structure expects and verifies.

The following is based on information publicly available, user feedback, community discussions or bug report analyses. It's not complete and further research is encouraged when in doubt.

Main memory

The data structures and raw data blocks are temporarily stored in computer memory before they get written to the device. It is critical that memory is reliable because even simple bit flips can have vast consequences and lead to damaged structures, not only in the filesystem but in the whole operating system.

Based on experience in the community, memory bit flips are more common than one would think. When it happens, it's reported by the tree-checker or by a checksum mismatch after reading blocks. There are some very obvious instances of bit flips that happen, e.g. in an ordered sequence of keys in metadata blocks. We can easily infer from the other data what values get damaged and how. However, fixing that is not straightforward and would require cross-referencing data from the entire filesystem to see the scope.

If available, ECC memory should lower the chances of bit flips, but this type of memory is not available in all cases. A memory test should be performed in case there's a visible bit flip pattern, though this may not detect a faulty memory module because the actual load of the system could be the factor making the problems appear. In recent years attacks on how the memory modules operate have been demonstrated ('rowhammer') achieving specific bits to be flipped. While these were targeted, this shows that a series of reads or writes can affect unrelated parts of memory.

Further reading:

- https://en.wikipedia.org/wiki/Row_hammer

What to do:

- run *memtest*, note that sometimes memory errors happen only when the system is under heavy load that the default memtest cannot trigger
- memory errors may appear as filesystem going read-only due to "pre write" check, that verify meta data before they get written but fail some basic consistency checks

Direct memory access (DMA)

Another class of errors is related to DMA (direct memory access) performed by device drivers. While this could be considered a software error, the data transfers that happen without CPU assistance may accidentally corrupt other pages. Storage devices utilize DMA for performance reasons, the filesystem structures and data pages are passed back and forth, making errors possible in case page life time is not properly tracked.

There are lots of quirks (device-specific workarounds) in Linux kernel drivers (regarding not only DMA) that are added when found. The quirks may avoid specific errors or disable some features to avoid worse problems.

What to do:

- use up-to-date kernel (recent releases or maintained long term support versions)
- as this may be caused by faulty drivers, keep the systems up-to-date

Rotational disks (HDD)

Rotational HDDs typically fail at the level of individual sectors or small clusters. Read failures are caught on the levels below the filesystem and are returned to the user as *EIO - Input/output error*. Reading the blocks repeatedly may return the data eventually, but this is better done by specialized tools and filesystem takes the result of the lower layers. Rewriting the sectors may trigger internal remapping but this inevitably leads to data loss.

Disk firmware is technically software but from the filesystem perspective is part of the hardware. IO requests are processed, and caching or various other optimizations are performed, which may lead to bugs under high load or unexpected physical conditions or unsupported use cases.

Disks are connected by cables with two ends, both of which can cause problems when not attached properly. Data transfers are protected by checksums and the lower layers try hard to transfer the data correctly or not at all. The errors

from badly-connecting cables may manifest as large amount of failed read or write requests, or as short error bursts depending on physical conditions.

What to do:

- check **smartctl** for potential issues

Solid state drives (SSD)

The mechanism of information storage is different from HDDs and this affects the failure mode as well. The data are stored in cells grouped in large blocks with limited number of resets and other write constraints. The firmware tries to avoid unnecessary resets and performs optimizations to maximize the storage media lifetime. The known techniques are deduplication (blocks with same fingerprint/hash are mapped to same physical block), compression or internal remapping and garbage collection of used memory cells. Due to the additional processing there are measures to verity the data e.g. by ECC codes.

The observations of failing SSDs show that the whole electronic fails at once or affects a lot of data (eg. stored on one chip). Recovering such data may need specialized equipment and reading data repeatedly does not help as it's possible with HDDs.

There are several technologies of the memory cells with different characteristics and price. The lifetime is directly affected by the type and frequency of data written. Writing "too much" distinct data (e.g. encrypted) may render the internal deduplication ineffective and lead to a lot of rewrites and increased wear of the memory cells.

There are several technologies and manufacturers so it's hard to describe them but there are some that exhibit similar behaviour:

- expensive SSD will use more durable memory cells and is optimized for reliability and high load
- cheap SSD is projected for a lower load ("desktop user") and is optimized for cost, it may employ the optimizations and/or extended error reporting partially or not at all

It's not possible to reliably determine the expected lifetime of an SSD due to lack of information about how it works or due to lack of reliable stats provided by the device.

Metadata writes tend to be the biggest component of lifetime writes to a SSD, so there is some value in reducing them. Depending on the device class (high end/low end) the features like DUP block group profiles may affect the reliability in both ways:

- *high end* are typically more reliable and using 'single' for data and metadata could be suitable to reduce device wear
- *low end* could lack ability to identify errors so an additional redundancy at the filesystem level (checksums, *DUP*) could help

Only users who consume 50 to 100% of the SSD's actual lifetime writes need to be concerned by the write amplification of btrfs DUP metadata. Most users will be far below 50% of the actual lifetime, or will write the drive to death and discover how many writes 100% of the actual lifetime was. SSD firmware often adds its own write multipliers that can be arbitrary and unpredictable and dependent on application behavior, and these will typically have far greater effect on SSD lifespan than DUP metadata. It's more or less impossible to predict when a SSD will run out of lifetime writes to within a factor of two, so it's hard to justify wear reduction as a benefit.

Further reading:

- https://www.snia.org/educational-library/ssd-and-deduplication-end-spinning-disk-2012
- https://www.snia.org/educational-library/realities-solid-state-storage-2013-2013
- https://www.snia.org/educational-library/ssd-performance-primer-2013
- https://www.snia.org/educational-library/how-controllers-maximize-ssd-life-2013

What to do:

- run **smartctl** or self-tests to look for potential issues
- keep the firmware up-to-date

NVM express, non-volatile memory (NVMe)

NVMe is a type of persistent memory usually connected over a system bus (PCIe) or similar interface and the speeds are an order of magnitude faster than SSD. It is also a non-rotating type of storage, and is not typically connected by a cable. It's not a SCSI type device either but rather a complete specification for logical device interface.

In a way the errors could be compared to a combination of SSD class and regular memory. Errors may exhibit as random bit flips or IO failures. There are tools to access the internal log (**nvme log** and **nvme-cli**) for a more detailed analysis.

There are separate error detection and correction steps performed e.g. on the bus level and in most cases never making in to the filesystem level. Once this happens it could mean there's some systematic error like overheating or bad physical connection of the device. You may want to run self-tests (using **smartctl**).

- https://en.wikipedia.org/wiki/NVM_Express
- https://www.smartmontools.org/wiki/NVMe_Support

Drive firmware

Firmware is technically still software but embedded into the hardware. As all software has bugs, so does firmware. Storage devices can update the firmware and fix known bugs. In some cases the it's possible to avoid certain bugs by quirks (device-specific workarounds) in Linux kernel.

A faulty firmware can cause wide range of corruptions from small and localized to large affecting lots of data. Self-repair capabilities may not be sufficient.

What to do:

- check for firmware updates in case there are known problems, note that updating firmware can be risky on itself
- use up-to-date kernel (recent releases or maintained long term support versions)

SD flash cards

There are a lot of devices with low power consumption and thus using storage media based on low power consumption too, typically flash memory stored on a chip enclosed in a detachable card package. An improperly inserted card may be damaged by electrical spikes when the device is turned on or off. The chips storing data in turn may be damaged permanently. All types of flash memory have a limited number of rewrites, so the data are internally translated by FTL (flash translation layer). This is implemented in firmware (technically a software) and prone to bugs that manifest as hardware errors.

Adding redundancy like using DUP profiles for both data and metadata can help in some cases but a full backup might be the best option once problems appear and replacing the card could be required as well.

Hardware as the main source of filesystem corruptions

If you use unreliable hardware and don't know about that, don't blame the filesystem when it tells you.

2.2.22 SEE ALSO

`acl(5)`, `btrfs(8)`, `chattr(1)`, `fstrim(8)`, `ioctl(2)`, `mkfs.btrfs(8)`, `mount(8)`, `swapon(8)`

2.3 btrfs-balance(8)

2.3.1 SYNOPSIS

btrfs balance <subcommand> <args>

2.3.2 DESCRIPTION

The primary purpose of the balance feature is to spread block groups across all devices so they match constraints defined by the respective profiles. See `mkfs.btrfs(8)` section *PROFILES* for more details. The scope of the balancing process can be further tuned by use of filters that can select the block groups to process. Balance works only on a mounted filesystem. Extent sharing is preserved and reflinks are not broken. Files are not defragmented nor recompressed, file extents are preserved but the physical location on devices will change.

The balance operation is cancellable by the user. The on-disk state of the filesystem is always consistent so an unexpected interruption (eg. system crash, reboot) does not corrupt the filesystem. The progress of the balance operation is temporarily stored as an internal state and will be resumed upon mount, unless the mount option *skip_balance* is specified.

> **Warning:** Running balance without filters will take a lot of time as it basically move data/metadata from the whole filesystem and needs to update all block pointers.

The filters can be used to perform following actions:

- convert block group profiles (filter *convert*)
- make block group usage more compact (filter *usage*)
- perform actions only on a given device (filters *devid*, *drange*)

The filters can be applied to a combination of block group types (data, metadata, system). Note that changing only the *system* type needs the force option. Otherwise *system* gets automatically converted whenever *metadata* profile is converted.

When metadata redundancy is reduced (eg. from RAID1 to single) the force option is also required and it is noted in system log.

Note: The balance operation needs enough work space, ie. space that is completely unused in the filesystem, otherwise this may lead to ENOSPC reports. See the section *ENOSPC* for more details.

2.3.3 Compatibility

Note: The balance subcommand also exists under the **btrfs filesystem** namespace. This still works for backward compatibility but is deprecated and should not be used any more.

Note: A short syntax **btrfs balance <path>** works due to backward compatibility but is deprecated and should not be used any more. Use **btrfs balance start** command instead.

2.3.4 Performance implications

Balancing operations are very IO intensive and can also be quite CPU intensive, impacting other ongoing filesystem operations. Typically large amounts of data are copied from one location to another, with corresponding metadata updates.

Depending upon the block group layout, it can also be seek heavy. Performance on rotational devices is noticeably worse compared to SSDs or fast arrays.

2.3.5 SUBCOMMAND

cancel <path>
cancels a running or paused balance, the command will block and wait until the current blockgroup being processed completes

Since kernel 5.7 the response time of the cancellation is significantly improved, on older kernels it might take a long time until currently processed chunk is completely finished.

pause <path>
pause running balance operation, this will store the state of the balance progress and used filters to the filesystem

resume <path>
resume interrupted balance, the balance status must be stored on the filesystem from previous run, eg. after it was paused or forcibly interrupted and mounted again with *skip_balance*

start [options] <path>
start the balance operation according to the specified filters, without any filters the data and metadata from the whole filesystem are moved. The process runs in the foreground.

Note: The balance command without filters will basically move everything in the filesystem to a new physical location on devices (ie. it does not affect the logical properties of file extents like offsets within files and extent sharing). The run time is potentially very long, depending on the filesystem size. To prevent starting a full balance by accident, the user is warned and has a few seconds to cancel the operation before it starts. The warning and delay can be skipped with *--full-bauance* option.

Please note that the filters must be written together with the *-d*, *-m* and *-s* options, because they're optional and bare *-d* and *-m* also work and mean no filters.

Note: When the target profile for conversion filter is *raid5* or *raid6*, there's a safety timeout of 10 seconds to warn users about the status of the feature

Options

-d[<filters>]
act on data block groups, see *FILTERS* section for details about *filters*

-m[<filters>]
act on metadata chunks, see *FILTERS* section for details about *filters*

-s[<filters>]
act on system chunks (requires *-f*), see *FILTERS* section for details about *filters*.

-f force a reduction of metadata integrity, eg. when going from *raid1* to *single*, or skip safety timeout when the target conversion profile is *raid5* or *raid6*

--background|--bg
run the balance operation asynchronously in the background, uses `fork(2)` to start the process that calls the kernel ioctl

--enqueue wait if there's another exclusive operation running, otherwise continue

-v (deprecated) alias for global '-v' option

status [-v] <path>
Show status of running or paused balance.

Options

-v (deprecated) alias for global *-v* option

2.3.6 FILTERS

From kernel 3.3 onwards, btrfs balance can limit its action to a subset of the whole filesystem, and can be used to change the replication configuration (e.g. moving data from single to RAID1). This functionality is accessed through the *-d*, *-m* or *-s* options to btrfs balance start, which filter on data, metadata and system blocks respectively.

A filter has the following structure: *type[=params][,type=...]*

The available types are:

profiles=<profiles>
Balances only block groups with the given profiles. Parameters are a list of profile names separated by "|" (pipe).

usage=<percent>, usage=<range>
Balances only block groups with usage under the given percentage. The value of 0 is allowed and will clean up completely unused block groups, this should not require any new work space allocated. You may want to use *usage=0* in case balance is returning ENOSPC and your filesystem is not too full.

The argument may be a single value or a range. The single value *N* means *at most N percent used*, equivalent to *..N* range syntax. Kernels prior to 4.4 accept only the single value format. The minimum range boundary is inclusive, maximum is exclusive.

devid=<id>
Balances only block groups which have at least one chunk on the given device. To list devices with ids use **btrfs filesystem show**.

drange=<range>
Balance only block groups which overlap with the given byte range on any device. Use in conjunction with *devid* to filter on a specific device. The parameter is a range specified as *start..end*.

vrange=<range>
Balance only block groups which overlap with the given byte range in the filesystem's internal virtual address

space. This is the address space that most reports from btrfs in the kernel log use. The parameter is a range specified as *start..end*.

convert=<profile>
Convert each selected block group to the given profile name identified by parameters.

Note: Starting with kernel 4.5, the *data* chunks can be converted to/from the *DUP* profile on a single device.

Note: Starting with kernel 4.6, all profiles can be converted to/from *DUP* on multi-device filesystems.

---

limit=<number>, limit=<range>
Process only given number of chunks, after all filters are applied. This can be used to specifically target a chunk in connection with other filters (*drange*, *vrange*) or just simply limit the amount of work done by a single balance run.

The argument may be a single value or a range. The single value *N* means *at most N chunks*, equivalent to *..N* range syntax. Kernels prior to 4.4 accept only the single value format. The range minimum and maximum are inclusive.

stripes=<range>
Balance only block groups which have the given number of stripes. The parameter is a range specified as *start..end*. Makes sense for block group profiles that utilize striping, ie. RAID0/10/5/6. The range minimum and maximum are inclusive.

soft
Takes no parameters. Only has meaning when converting between profiles. When doing convert from one profile to another and soft mode is on, chunks that already have the target profile are left untouched. This is useful e.g. when half of the filesystem was converted earlier but got cancelled.

The soft mode switch is (like every other filter) per-type. For example, this means that we can convert metadata chunks the "hard" way while converting data chunks selectively with soft switch.

Profile names, used in *profiles* and *convert* are one of: *raid0*, *raid1*, *raid1c3*, *raid1c4*, *raid10*, *raid5*, *raid6*, *dup*, *single*. The mixed data/metadata profiles can be converted in the same way, but it's conversion between mixed and non-mixed is not implemented. For the constraints of the profiles please refer to `mkfs.btrfs(8)`, section *PROFILES*.

2.3.7 ENOSPC

The way balance operates, it usually needs to temporarily create a new block group and move the old data there, before the old block group can be removed. For that it needs the work space, otherwise it fails for ENOSPC reasons. This is not the same ENOSPC as if the free space is exhausted. This refers to the space on the level of block groups, which are bigger parts of the filesystem that contain many file extents.

The free work space can be calculated from the output of the **btrfs filesystem show** command:

```
Label: 'BTRFS'  uuid: 8a9d72cd-ead3-469d-b371-9c7203276265
        Total devices 2 FS bytes used 77.03GiB
        devid    1 size 53.90GiB used 51.90GiB path /dev/sdc2
        devid    2 size 53.90GiB used 51.90GiB path /dev/sde1
```

size - used = free work space

53.90GiB - 51.90GiB = 2.00GiB

An example of a filter that does not require workspace is *usage=0*. This will scan through all unused block groups of a given type and will reclaim the space. After that it might be possible to run other filters.

CONVERSIONS ON MULTIPLE DEVICES

Conversion to profiles based on striping (RAID0, RAID5/6) require the work space on each device. An interrupted balance may leave partially filled block groups that consume the work space.

2.3.8 EXAMPLES

A more comprehensive example when going from one to multiple devices, and back, can be found in section *TYPICAL USECASES* of `btrfs-device(8)`.

MAKING BLOCK GROUP LAYOUT MORE COMPACT

The layout of block groups is not normally visible; most tools report only summarized numbers of free or used space, but there are still some hints provided.

Let's use the following real life example and start with the output:

```
$ btrfs filesystem df /path
Data, single: total=75.81GiB, used=64.44GiB
System, RAID1: total=32.00MiB, used=20.00KiB
Metadata, RAID1: total=15.87GiB, used=8.84GiB
GlobalReserve, single: total=512.00MiB, used=0.00B
```

Roughly calculating for data, *75G - 64G = 11G*, the used/total ratio is about *85%*. How can we can interpret that:

- chunks are filled by 85% on average, ie. the *usage* filter with anything smaller than 85 will likely not affect anything
- in a more realistic scenario, the space is distributed unevenly, we can assume there are completely used chunks and the remaining are partially filled

Compacting the layout could be used on both. In the former case it would spread data of a given chunk to the others and removing it. Here we can estimate that roughly 850 MiB of data have to be moved (85% of a 1 GiB chunk).

In the latter case, targeting the partially used chunks will have to move less data and thus will be faster. A typical filter command would look like:

```
# btrfs balance start -dusage=50 /path
Done, had to relocate 2 out of 97 chunks

$ btrfs filesystem df /path
Data, single: total=74.03GiB, used=64.43GiB
System, RAID1: total=32.00MiB, used=20.00KiB
Metadata, RAID1: total=15.87GiB, used=8.84GiB
GlobalReserve, single: total=512.00MiB, used=0.00B
```

As you can see, the *total* amount of data is decreased by just 1 GiB, which is an expected result. Let's see what will happen when we increase the estimated usage filter.

```
# btrfs balance start -dusage=85 /path
Done, had to relocate 13 out of 95 chunks

$ btrfs filesystem df /path
```

(continues on next page)

(continued from previous page)

```
Data, single: total=68.03GiB, used=64.43GiB
System, RAID1: total=32.00MiB, used=20.00KiB
Metadata, RAID1: total=15.87GiB, used=8.85GiB
GlobalReserve, single: total=512.00MiB, used=0.00B
```

Now the used/total ratio is about 94% and we moved about *74G - 68G = 6G* of data to the remaining blockgroups, ie. the 6GiB are now free of filesystem structures, and can be reused for new data or metadata block groups.

We can do a similar exercise with the metadata block groups, but this should not typically be necessary, unless the used/total ratio is really off. Here the ratio is roughly 50% but the difference as an absolute number is "a few gigabytes", which can be considered normal for a workload with snapshots or reflinks updated frequently.

```
# btrfs balance start -musage=50 /path
Done, had to relocate 4 out of 89 chunks

$ btrfs filesystem df /path
Data, single: total=68.03GiB, used=64.43GiB
System, RAID1: total=32.00MiB, used=20.00KiB
Metadata, RAID1: total=14.87GiB, used=8.85GiB
GlobalReserve, single: total=512.00MiB, used=0.00B
```

Just 1 GiB decrease, which possibly means there are block groups with good utilization. Making the metadata layout more compact would in turn require updating more metadata structures, ie. lots of IO. As running out of metadata space is a more severe problem, it's not necessary to keep the utilization ratio too high. For the purpose of this example, let's see the effects of further compaction:

```
# btrfs balance start -musage=70 /path
Done, had to relocate 13 out of 88 chunks

$ btrfs filesystem df .
Data, single: total=68.03GiB, used=64.43GiB
System, RAID1: total=32.00MiB, used=20.00KiB
Metadata, RAID1: total=11.97GiB, used=8.83GiB
GlobalReserve, single: total=512.00MiB, used=0.00B
```

GETTING RID OF COMPLETELY UNUSED BLOCK GROUPS

Normally the balance operation needs a work space, to temporarily move the data before the old block groups gets removed. If there's no work space, it ends with *no space left*.

There's a special case when the block groups are completely unused, possibly left after removing lots of files or deleting snapshots. Removing empty block groups is automatic since 3.18. The same can be achieved manually with a notable exception that this operation does not require the work space. Thus it can be used to reclaim unused block groups to make it available.

```
# btrfs balance start -dusage=0 /path
```

This should lead to decrease in the *total* numbers in the **btrfs filesystem df** output.

2.3.9 EXIT STATUS

Unless indicated otherwise below, all **btrfs balance** subcommands return a zero exit status if they succeed, and non zero in case of failure.

The **pause**, **cancel**, and **resume** subcommands exit with a status of **2** if they fail because a balance operation was not running.

The **status** subcommand exits with a status of **0** if a balance operation is not running, **1** if the command-line usage is incorrect or a balance operation is still running, and **2** on other errors.

2.3.10 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.3.11 SEE ALSO

`mkfs.btrfs(8)`, `btrfs-device(8)`

2.4 btrfs-check(8)

2.4.1 SYNOPSIS

btrfs check [options] <device>

2.4.2 DESCRIPTION

The filesystem checker is used to verify structural integrity of a filesystem and attempt to repair it if requested. It is recommended to unmount the filesystem prior to running the check, but it is possible to start checking a mounted filesystem (see *--force*).

By default, **btrfs check** will not modify the device but you can reaffirm that by the option *--readonly*.

btrfsck is an alias of **btrfs check** command and is now deprecated.

> **Warning:** Do not use *--repair* unless you are advised to do so by a developer or an experienced user, and then only after having accepted that no *fsck* successfully repair all types of filesystem corruption. Eg. some other software or hardware bugs can fatally damage a volume.

The structural integrity check verifies if internal filesystem objects or data structures satisfy the constraints, point to the right objects or are correctly connected together.

There are several cross checks that can detect wrong reference counts of shared extents, backreferences, missing extents of inodes, directory and inode connectivity etc.

The amount of memory required can be high, depending on the size of the filesystem, similarly the run time. Check the modes that can also affect that.

2.4.3 SAFE OR ADVISORY OPTIONS

-b|--backup
use the first valid set of backup roots stored in the superblock

This can be combined with *--super* if some of the superblocks are damaged.

--check-data-csum verify checksums of data blocks

This expects that the filesystem is otherwise OK, and is basically an offline *scrub* that does not repair data from spare copies.

--chunk-root <bytenr> use the given offset *bytenr* for the chunk tree root

-E|--subvol-extents <subvolid>
show extent state for the given subvolume

-p|--progress
indicate progress at various checking phases

-Q|--qgroup-report
verify qgroup accounting and compare against filesystem accounting

-r|--tree-root <bytenr>
use the given offset 'bytenr' for the tree root

--readonly (default) run in read-only mode, this option exists to calm potential panic when users are going to run the checker

-s|--super <N>
use Nth superblock copy, valid values are 0, 1 or 2 if the respective superblock offset is within the device size

This can be used to use a different starting point if some of the primary superblock is damaged.

--clear-space-cache v1|v2
completely wipe all free space cache of given type

For free space cache *v1*, the *clear_cache* kernel mount option only rebuilds the free space cache for block groups that are modified while the filesystem is mounted with that option. Thus, using this option with *v1* makes it possible to actually clear the entire free space cache.

For free space cache *v2*, the *clear_cache* kernel mount option destroys the entire free space cache. This option, with *v2* provides an alternative method of clearing the free space cache that doesn't require mounting the filesystem.

--clear-ino-cache remove leftover items pertaining to the deprecated inode map feature

2.4.4 DANGEROUS OPTIONS

--repair enable the repair mode and attempt to fix problems where possible

---

Note: There's a warning and 10 second delay when this option is run without *--force* to give users a chance to think twice before running repair, the warnings in documentation have shown to be insufficient

---

--init-csum-tree create a new checksum tree and recalculate checksums in all files

> **Warning:** Do not blindly use this option to fix checksum mismatch problems.

--init-extent-tree build the extent tree from scratch

> **Warning:** Do not use unless you know what you're doing.

--mode <MODE> select mode of operation regarding memory and IO

The *MODE* can be one of:

original
: The metadata are read into memory and verified, thus the requirements are high on large filesystems and can even lead to out-of-memory conditions. The possible workaround is to export the block device over network to a machine with enough memory.

lowmem
: This mode is supposed to address the high memory consumption at the cost of increased IO when it needs to re-read blocks. This may increase run time.

Note: *lowmem* mode does not work with *--repair* yet, and is still considered experimental.

--force allow work on a mounted filesystem. Note that this should work fine on a quiescent or read-only mounted filesystem but may crash if the device is changed externally, eg. by the kernel module. Repair without mount checks is not supported right now.

This option also skips the delay and warning in the repair mode (see *--repair*).

2.4.5 EXIT STATUS

btrfs check returns a zero exit status if it succeeds. Non zero is returned in case of failure.

2.4.6 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.4.7 SEE ALSO

`mkfs.btrfs(8)`, `btrfs-scrub(8)`, `btrfs-rescue(8)`

2.5 btrfs-convert(8)

2.5.1 SYNOPSIS

btrfs-convert [options] <device>

2.5.2 DESCRIPTION

The **btrfs-convert** tool can be used to convert existing source filesystem image to a btrfs filesystem in-place. The original filesystem image is accessible in subvolume named like *ext2_saved* as file *image*.

Supported filesystems:

- ext2, ext3, ext4 -- original feature, always built in
- reiserfs -- since version 4.13, optionally built, requires libreiserfscore 3.6.27
- ntfs -- external tool https://github.com/maharmstone/ntfs2btrfs

The list of supported source filesystem by a given binary is listed at the end of help (option *--help*).

> **Warning:** If you are going to perform rollback to the original filesystem, you should not execute **btrfs balance** command on the converted filesystem. This will change the extent layout and make **btrfs-convert** unable to rollback.

The conversion utilizes free space of the original filesystem. The exact estimate of the required space cannot be foretold. The final btrfs metadata might occupy several gigabytes on a hundreds-gigabyte filesystem.

If the ability to rollback is no longer important, the it is recommended to perform a few more steps to transition the btrfs filesystem to a more compact layout. This is because the conversion inherits the original data blocks' fragmentation, and also because the metadata blocks are bound to the original free space layout.

Due to different constraints, it is only possible to convert filesystems that have a supported data block size (ie. the same that would be valid for **mkfs.btrfs**). This is typically the system page size (4KiB on x86_64 machines).

BEFORE YOU START

The source filesystem must be clean, eg. no journal to replay or no repairs needed. The respective **fsck** utility must be run on the source filessytem prior to conversion. Please refer to the manual pages in case you encounter problems.

For ext2/3/4:

```
# e2fsck -fvy /dev/sdx
```

For reiserfs:

```
# reiserfsck -fy /dev/sdx
```

Skipping that step could lead to incorrect results on the target filesystem, but it may work.

REMOVE THE ORIGINAL FILESYSTEM METADATA

By removing the subvolume named like *ext2_saved* or *reiserfs_saved*, all metadata of the original filesystem will be removed:

```
# btrfs subvolume delete /mnt/ext2_saved
```

At this point it is not possible to do a rollback. The filesystem is usable but may be impacted by the fragmentation inherited from the original filesystem.

MAKE FILE DATA MORE CONTIGUOUS

An optional but recommended step is to run defragmentation on the entire filesystem. This will attempt to make file extents more contiguous.

```
# btrfs filesystem defrag -v -r -f -t 32M /mnt/btrfs
```

Verbose recursive defragmentation (*-v*, *-r*), flush data per-file (*-f*) with target extent size 32MiB (*-t*).

ATTEMPT TO MAKE BTRFS METADATA MORE COMPACT

Optional but recommended step.

The metadata block groups after conversion may be smaller than the default size (256MiB or 1GiB). Running a balance will attempt to merge the block groups. This depends on the free space layout (and fragmentation) and may fail due to lack of enough work space. This is a soft error leaving the filesystem usable but the block group layout may remain unchanged.

Note that balance operation takes a lot of time, please see also `btrfs-balance(8)`.

```
# btrfs balance start -m /mnt/btrfs
```

2.5.3 OPTIONS

--csum <type>, --checksum <type> Specify the checksum algorithm. Default is *crc32c*. Valid values are *crc32c*, *xxhash*, *sha256* or *blake2*. To mount such filesystem kernel must support the checksums as well.

-d|--no-datasum
disable data checksum calculations and set the NODATASUM file flag, this can speed up the conversion

-i|--no-xattr
ignore xattrs and ACLs of files

-n|--no-inline
disable inlining of small files to metadata blocks, this will decrease the metadata consumption and may help to convert a filesystem with low free space

-N|--nodesize <SIZE>
set filesystem nodesize, the tree block size in which btrfs stores its metadata. The default value is 16KiB (16384) or the page size, whichever is bigger. Must be a multiple of the sectorsize, but not larger than 65536. See `mkfs.btrfs(8)` for more details.

-r|--rollback
rollback to the original ext2/3/4 filesystem if possible

-l|--label <LABEL>
set filesystem label during conversion

-L|--copy-label
use label from the converted filesystem

-O|--features <feature1>[,<feature2>...]
A list of filesystem features enabled the at time of conversion. Not all features are supported by old kernels. To disable a feature, prefix it with ^. Description of the features is in section *FILESYSTEM FEATURES* of `mkfs.btrfs(8)`.

To see all available features that btrfs-convert supports run:

```
btrfs-convert -O list-all+
```

-p|--progress
show progress of conversion (a heartbeat indicator and number of inodes processed), on by default

--no-progress disable progress and show only the main phases of conversion

--uuid <SPEC> set the FSID of the new filesystem based on 'SPEC':

- *new* - (default) generate UUID for the FSID of btrfs
- *copy* - copy UUID from the source filesystem
- *UUID* - a conforming UUID value, the 36 byte string representation

2.5.4 EXIT STATUS

btrfs-convert will return 0 if no error happened. If any problems happened, 1 will be returned.

2.5.5 SEE ALSO

`mkfs.btrfs(8)`

2.6 btrfs-device(8)

2.6.1 SYNOPSIS

btrfs device <subcommand> <args>

2.6.2 DESCRIPTION

The **btrfs device** command group is used to manage devices of the btrfs filesystems.

2.6.3 DEVICE MANAGEMENT

2.6.4 SUBCOMMAND

add [-Kf] <device> [<device>...] <path>
Add device(s) to the filesystem identified by *path*.

If applicable, a whole device discard (TRIM) operation is performed prior to adding the device. A device with existing filesystem detected by `blkid(8)` will prevent device addition and has to be forced. Alternatively the filesystem can be wiped from the device using eg. the `wipefs(8)` tool.

The operation is instant and does not affect existing data. The operation merely adds the device to the filesystem structures and creates some block groups headers.

`Options`

-K|--nodiscard
do not perform discard (TRIM) by default

-f|--force
force overwrite of existing filesystem on the given disk(s)

--enqueue wait if there's another exclusive operation running, otherwise continue

remove [options] <device>|<devid> [<device>|<devid>...] <path>
Remove device(s) from a filesystem identified by <path>

Device removal must satisfy the profile constraints, otherwise the command fails. The filesystem must be converted to profile(s) that would allow the removal. This can typically happen when going down from 2 devices to 1 and using the RAID1 profile. See the section *TYPICAL USECASES*.

The operation can take long as it needs to move all data from the device.

It is possible to delete the device that was used to mount the filesystem. The device entry in the mount table will be replaced by another device name with the lowest device id.

If the filesystem is mounted in degraded mode (*-o degraded*), special term *missing* can be used for *device*. In that case, the first device that is described by the filesystem metadata, but not present at the mount time will be removed.

Note: In most cases, there is only one missing device in degraded mode, otherwise mount fails. If there are two or more devices missing (e.g. possible in RAID6), you need specify *missing* as many times as the number of missing devices to remove all of them.

`Options`

--enqueue wait if there's another exclusive operation running, otherwise continue

delete <device>|<devid> [<device>|<devid>...] <path>
Alias of remove kept for backward compatibility

ready <device>
Wait until all devices of a multiple-device filesystem are scanned and registered within the kernel module. This is to provide a way for automatic filesystem mounting tools to wait before the mount can start. The device scan is only one of the preconditions and the mount can fail for other reasons. Normal users usually do not need this command and may safely ignore it.

scan [options] [<device> [<device>...]]
Scan devices for a btrfs filesystem and register them with the kernel module. This allows mounting multiple-device filesystem by specifying just one from the whole group.

If no devices are passed, all block devices that blkid reports to contain btrfs are scanned.

The options *--all-devices* or *-d* can be used as a fallback in case blkid is not available. If used, behavior is the same as if no devices are passed.

The command can be run repeatedly. Devices that have been already registered remain as such. Reloading the kernel module will drop this information. There's an alternative way of mounting multiple-device filesystem without the need for prior scanning. See the mount option *device*.

`Options`

-d|--all-devices
Enumerate and register all devices, use as a fallback in case blkid is not available.

-u|--forget
Unregister a given device or all stale devices if no path is given, the device must be unmounted otherwise it's an error.

stats [options] <path>|<device>
Read and print the device IO error statistics for all devices of the given filesystem identified by *path* or for a single *device*>. *The filesystem must be mounted. See section *DEVICE STATS* for more information about the reported statistics and the meaning.

Options

-z|--reset
Print the stats and reset the values to zero afterwards.

-c|--check
Check if the stats are all zeros and return 0 if it is so. Set bit 6 of the return code if any of the statistics is no-zero. The error values is 65 if reading stats from at least one device failed, otherwise it's 64.

usage [options] <path> [<path>...]::
Show detailed information about internal allocations on devices.

The level of detail can differ if the command is run under a regular or the root user (due to use of restricted ioctls). The first example below is for normal user (warning included) and the next one with root on the same filesystem:

```
WARNING: cannot read detailed chunk info, per-device usage will not be shown, run␣
↪as root
/dev/sdc1, ID: 1
   Device size:           931.51GiB
   Device slack:              0.00B
   Unallocated:           931.51GiB

/dev/sdc1, ID: 1
   Device size:           931.51GiB
   Device slack:              0.00B
   Data,single:           641.00GiB
   Data,RAID0/3:            1.00GiB
   Metadata,single:        19.00GiB
   System,single:          32.00MiB
   Unallocated:           271.48GiB
```

- *Device size* -- size of the device as seen by the filesystem (may be different than actual device size)
- *Device slack* -- portion of device not used by the filesystem but still available in the physical space provided by the device, eg. after a device shrink
- *Data,single*, *Metadata,single*, *System,single* -- in general, list of block group type (Data, Metadata, System) and profile (single, RAID1, ...) allocated on the device
- *Data,RAID0/3* -- in particular, striped profiles RAID0/RAID10/RAID5/RAID6 with the number of devices on which the stripes are allocated, multiple occurrences of the same profile can appear in case a new device has been added and all new available stripes have been used for writes
- *Unallocated* -- remaining space that the filesystem can still use for new block groups

Options

-b|--raw
raw numbers in bytes, without the *B* suffix

-h|--human-readable
print human friendly numbers, base 1024, this is the default

-H	print human friendly numbers, base 1000
--iec	select the 1024 base for the following options, according to the IEC standard

--si select the 1000 base for the following options, according to the SI standard

-k|--kbytes
show sizes in KiB, or kB with --si

-m|--mbytes
show sizes in MiB, or MB with --si

-g|--gbytes
show sizes in GiB, or GB with --si

-t|--tbytes
show sizes in TiB, or TB with --si

If conflicting options are passed, the last one takes precedence.

2.6.5 DEVICE STATS

The device stats keep persistent record of several error classes related to doing IO. The current values are printed at mount time and updated during filesystem lifetime or from a scrub run.

```
$ btrfs device stats /dev/sda3
[/dev/sda3].write_io_errs   0
[/dev/sda3].read_io_errs    0
[/dev/sda3].flush_io_errs   0
[/dev/sda3].corruption_errs 0
[/dev/sda3].generation_errs 0
```

write_io_errs
Failed writes to the block devices, means that the layers beneath the filesystem were not able to satisfy the write request.

read_io_errors
Read request analogy to write_io_errs.

flush_io_errs
Number of failed writes with the *FLUSH* flag set. The flushing is a method of forcing a particular order between write requests and is crucial for implementing crash consistency. In case of btrfs, all the metadata blocks must be permanently stored on the block device before the superblock is written.

corruption_errs
A block checksum mismatched or a corrupted metadata header was found.

generation_errs
The block generation does not match the expected value (eg. stored in the parent node).

Since kernel 5.14 the device stats are also available in textual form in */sys/fs/btrfs/FSID/devinfo/DEVID/error_stats*.

2.6.6 EXIT STATUS

btrfs device returns a zero exit status if it succeeds. Non zero is returned in case of failure.

If the *-c* option is used, *btrfs device stats* will add 64 to the exit status if any of the error counters is non-zero.

2.6.7 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.6.8 SEE ALSO

`mkfs.btrfs(8)`, `btrfs-replace(8)`, `btrfs-balance(8)`

2.7 btrfs-filesystem(8)

2.7.1 SYNOPSIS

btrfs filesystem <subcommand> <args>

2.7.2 DESCRIPTION

btrfs filesystem is used to perform several whole filesystem level tasks, including all the regular filesystem operations like resizing, space stats, label setting/getting, and defragmentation. There are other whole filesystem tasks like scrub or balance that are grouped in separate commands.

2.7.3 SUBCOMMAND

df [options] <path>

Show a terse summary information about allocation of block group types of a given mount point. The original purpose of this command was a debugging helper. The output needs to be further interpreted and is not suitable for quick overview.

An example with description:

- device size: *1.9TiB*, one device, no RAID
- filesystem size: *1.9TiB*
- created with: **mkfs.btrfs -d single -m single**

```
$ btrfs filesystem df /path
Data, single: total=1.15TiB, used=1.13TiB
System, single: total=32.00MiB, used=144.00KiB
Metadata, single: total=12.00GiB, used=6.45GiB
GlobalReserve, single: total=512.00MiB, used=0.00B
```

- *Data*, *System* and *Metadata* are separate block group types. *GlobalReserve* is an artificial and internal emergency space, see below.
- *single* -- the allocation profile, defined at mkfs time

- *total* -- sum of space reserved for all allocation profiles of the given type, ie. all Data/single. Note that it's not total size of filesystem.
- *used* -- sum of used space of the above, ie. file extents, metadata blocks

GlobalReserve is an artificial and internal emergency space. It is used eg. when the filesystem is full. Its *total* size is dynamic based on the filesystem size, usually not larger than 512MiB, *used* may fluctuate.

The GlobalReserve is a portion of Metadata. In case the filesystem metadata is exhausted, *GlobalReserve/total* + *Metadata/used* = *Metadata/total*. Otherwise there appears to be some unused space of Metadata.

`Options`

-b|--raw
raw numbers in bytes, without the *B* suffix

-h|--human-readable
print human friendly numbers, base 1024, this is the default

-H	print human friendly numbers, base 1000
--iec	select the 1024 base for the following options, according to the IEC standard
--si	select the 1000 base for the following options, according to the SI standard

-k|--kbytes
show sizes in KiB, or kB with --si

-m|--mbytes
show sizes in MiB, or MB with --si

-g|--gbytes
show sizes in GiB, or GB with --si

-t|--tbytes
show sizes in TiB, or TB with --si

If conflicting options are passed, the last one takes precedence.

defragment [options] <file>|<dir> [<file>|<dir>...]
Defragment file data on a mounted filesystem. Requires kernel 2.6.33 and newer.

If *-r* is passed, files in dir will be defragmented recursively (not descending to subvolumes, mount points and directory symlinks). The start position and the number of bytes to defragment can be specified by start and length using *-s* and *-l* options below. Extents bigger than value given by *-t* will be skipped, otherwise this value is used as a target extent size, but is only advisory and may not be reached if the free space is too fragmented. Use 0 to take the kernel default, which is 256KiB but may change in the future. You can also turn on compression in defragment operations.

Warning: Defragmenting with Linux kernel versions < 3.9 or 3.14-rc2 as well as with Linux stable kernel versions 3.10.31, 3.12.12 or 3.13.4 will break up the reflinks of COW data (for example files copied with **cp --reflink**, snapshots or de-duplicated data). This may cause considerable increase of space usage depending on the broken up reflinks.

Note: Directory arguments without *-r* do not defragment files recursively but will defragment certain internal trees (extent tree and the subvolume tree). This has been confusing and could be removed in the future.

For *start*, *len*, *size* it is possible to append units designator: *K*, *M*, *G*, *T*, *P*, or *E*, which represent KiB, MiB, GiB, TiB, PiB, or EiB, respectively (case does not matter).

Options

-c[<algo>]
compress file contents while defragmenting. Optional argument selects the compression algorithm, *zlib* (default), *lzo* or *zstd*. Currently it's not possible to select no compression. See also section *EXAMPLES*.

-r defragment files recursively in given directories, does not descend to subvolumes or mount points

-f flush data for each file before going to the next file.

This will limit the amount of dirty data to current file, otherwise the amount accumulates from several files and will increase system load. This can also lead to ENOSPC if there's too much dirty data to write and it's not possible to make the reservations for the new data (ie. how the COW design works).

-s <start>[kKmMgGtTpPeE]
defragmentation will start from the given offset, default is beginning of a file

-l <len>[kKmMgGtTpPeE]
defragment only up to *len* bytes, default is the file size

-t <size>[kKmMgGtTpPeE]
target extent size, do not touch extents bigger than *size*, default: 32MiB

The value is only advisory and the final size of the extents may differ, depending on the state of the free space and fragmentation or other internal logic. Reasonable values are from tens to hundreds of megabytes.

-v (deprecated) alias for global *-v* option

du [options] <path> [<path>..]
Calculate disk usage of the target files using FIEMAP. For individual files, it will report a count of total bytes, and exclusive (not shared) bytes. We also calculate a 'set shared' value which is described below.

Each argument to **btrfs filesystem du** will have a *set shared* value calculated for it. We define each *set* as those files found by a recursive search of an argument (recursion descends to subvolumes but not mount points). The *set shared* value then is a sum of all shared space referenced by the set.

set shared takes into account overlapping shared extents, hence it isn't as simple as adding up shared extents.

Options

-s|--summarize
display only a total for each argument

--raw raw numbers in bytes, without the *B* suffix.

--human-readable print human friendly numbers, base 1024, this is the default

--iec select the 1024 base for the following options, according to the IEC standard.

--si select the 1000 base for the following options, according to the SI standard.

--kbytes show sizes in KiB, or kB with --si.

--mbytes show sizes in MiB, or MB with --si.

--gbytes show sizes in GiB, or GB with --si.

--tbytes show sizes in TiB, or TB with --si.

label [<device>|<mountpoint>] [<newlabel>]
Show or update the label of a filesystem. This works on a mounted filesystem or a filesystem image.

The *newlabel* argument is optional. Current label is printed if the argument is omitted.

Note: The maximum allowable length shall be less than 256 chars and must not contain a newline. The trailing newline is stripped automatically.

resize [options] [<devid>:][+/-]<size>[kKmMgGtTpPeE]|[<devid>:]max <path>
Resize a mounted filesystem identified by *path*. A particular device can be resized by specifying a *devid*.

Warning: If *path* is a file containing a BTRFS image then resize does not work as expected and does not resize the image. This would resize the underlying filesystem instead.

The *devid* can be found in the output of **btrfs filesystem show** and defaults to 1 if not specified. The *size* parameter specifies the new size of the filesystem. If the prefix + or - is present the size is increased or decreased by the quantity *size*. If no units are specified, bytes are assumed for *size*. Optionally, the size parameter may be suffixed by one of the following unit designators: *K*, *M*, *G*, *T*, *P*, or *E*, which represent KiB, MiB, GiB, TiB, PiB, or EiB, respectively (case does not matter).

If *max* is passed, the filesystem will occupy all available space on the device respecting *devid* (remember, devid 1 by default).

The resize command does not manipulate the size of underlying partition. If you wish to enlarge/reduce a filesystem, you must make sure you can expand the partition before enlarging the filesystem and shrink the partition after reducing the size of the filesystem. This can done using `fdisk(8)` or `parted(8)` to delete the existing partition and recreate it with the new desired size. When recreating the partition make sure to use the same starting partition offset as before.

Growing is usually instant as it only updates the size. However, shrinking could take a long time if there are data in the device area that's beyond the new end. Relocation of the data takes time.

See also section *EXAMPLES*.

`Options`

--enqueue wait if there's another exclusive operation running, otherwise continue

show [options] [<path>|<uuid>|<device>|<label>]
Show the btrfs filesystem with some additional info about devices and space allocation.

If no option none of *path*/*uuid*/*device*/*label* is passed, information about all the BTRFS filesystems is shown, both mounted and unmounted.

`Options`

-m|--mounted
probe kernel for mounted BTRFS filesystems

-d|--all-devices
scan all devices under */dev*, otherwise the devices list is extracted from the */proc/partitions* file. This is a fallback option if there's no device node manager (like udev) available in the system.

--raw raw numbers in bytes, without the *B* suffix

--human-readable print human friendly numbers, base 1024, this is the default

--iec select the 1024 base for the following options, according to the IEC standard

--si select the 1000 base for the following options, according to the SI standard

--kbytes show sizes in KiB, or kB with --si

--mbytes show sizes in MiB, or MB with --si

--gbytes show sizes in GiB, or GB with --si

--tbytes show sizes in TiB, or TB with --si

sync <path>

Force a sync of the filesystem at *path*, similar to the `sync(1)` command. In addition, it starts cleaning of deleted subvolumes. To wait for the subvolume deletion to complete use the **btrfs subvolume sync** command.

usage [options] <path> [<path>...]

Show detailed information about internal filesystem usage. This is supposed to replace the **btrfs filesystem df** command in the long run.

The level of detail can differ if the command is run under a regular or the root user (due to use of restricted ioctl). For both there's a summary section with information about space usage:

```
$ btrfs filesystem usage /path
WARNING: cannot read detailed chunk info, RAID5/6 numbers will be incorrect, run as␣
↪root
Overall:
    Device size:                   1.82TiB
    Device allocated:              1.17TiB
    Device unallocated:          669.99GiB
    Device missing:                0.00B
    Used:                          1.14TiB
    Free (estimated):            692.57GiB      (min: 692.57GiB)
    Free (statfs, df)            692.57GiB
    Data ratio:                       1.00
    Metadata ratio:                   1.00
    Global reserve:              512.00MiB      (used: 0.00B)
    Multiple profiles:                  no
```

- *Device size* -- sum of raw device capacity available to the filesystem
- *Device allocated* -- sum of total space allocated for data/metadata/system profiles, this also accounts space reserved but not yet used for extents
- *Device unallocated* -- the remaining unallocated space for future allocations (difference of the above two numbers)
- *Device missing* -- sum of capacity of all missing devices
- *Used* -- sum of the used space of data/metadata/system profiles, not including the reserved space
- *Free (estimated)* -- approximate size of the remaining free space usable for data, including currently allocated space and estimating the usage of the unallocated space based on the block group profiles, the *min* is the lower bound of the estimate in case multiple profiles are present
- *Free (statfs, df)* -- the amount of space available for data as reported by the **statfs** syscall, also returned as *Avail* in the output of **df**. The value is calculated in a different way and may not match the estimate in some cases (eg. multiple profiles).
- *Data ratio* -- ratio of total space for data including redundancy or parity to the effectively usable data space, eg. single is 1.0, RAID1 is 2.0 and for RAID5/6 it depends on the number of devices
- *Metadata ratio* -- dtto, for metadata
- *Global reserve* -- portion of metadata currently used for global block reserve, used for emergency purposes (like deletion on a full filesystem)
- *Multiple profiles* -- what block group types (data, metadata) have more than one profile (single, raid1, ...), see `btrfs(5)` section *FILESYSTEMS WITH MULTIPLE BLOCK GROUP PROFILES*.

And on a zoned filesystem there are two more lines in the *Device* section:

```
Device zone unusable:          5.13GiB
Device zone size:            256.00MiB
```

- *Device zone unusable* -- sum of of space that's been used in the past but now is not due to COW and not referenced anymory, the chunks have to be reclaimed and zones reset to make it usable again
- *Device zone size* -- the reported zone size of the host-managed device, same for all devices

The root user will also see stats broken down by block group types:

```
Data,single: Size:1.15TiB, Used:1.13TiB (98.26%)
   /dev/sdb        1.15TiB

Metadata,single: Size:12.00GiB, Used:6.45GiB (53.75%)
   /dev/sdb       12.00GiB

System,single: Size:32.00MiB, Used:144.00KiB (0.44%)
   /dev/sdb       32.00MiB

Unallocated:
   /dev/sdb      669.99GiB
```

Data is block group type, *single* is block group profile, *Size* is total size occupied by this type, *Used* is the actually used space, the percent is ratio of *Used/Size*. The *Unallocated* is remaining space.

`Options`

-b|--raw
raw numbers in bytes, without the *B* suffix

-h|--human-readable
print human friendly numbers, base 1024, this is the default

-H print human friendly numbers, base 1000

--iec select the 1024 base for the following options, according to the IEC standard

--si select the 1000 base for the following options, according to the SI standard

-k|--kbytes
show sizes in KiB, or kB with --si

-m|--mbytes
show sizes in MiB, or MB with --si

-g|--gbytes
show sizes in GiB, or GB with --si

-t|--tbytes
show sizes in TiB, or TB with --si

-T show data in tabular format

If conflicting options are passed, the last one takes precedence.

2.7.4 EXAMPLES

$ btrfs filesystem defrag -v -r dir/

Recursively defragment files under *dir/*, print files as they are processed. The file names will be printed in batches, similarly the amount of data triggered by defragmentation will be proportional to last N printed files. The system dirty memory throttling will slow down the defragmentation but there can still be a lot of IO load and the system may stall for a moment.

$ btrfs filesystem defrag -v -r -f dir/

Recursively defragment files under *dir/*, be verbose and wait until all blocks are flushed before processing next file. You can note slower progress of the output and lower IO load (proportional to currently defragmented file).

$ btrfs filesystem defrag -v -r -f -clzo dir/

Recursively defragment files under *dir/*, be verbose, wait until all blocks are flushed and force file compression.

$ btrfs filesystem defrag -v -r -t 64M dir/

Recursively defragment files under *dir/*, be verbose and try to merge extents to be about 64MiB. As stated above, the success rate depends on actual free space fragmentation and the final result is not guaranteed to meet the target even if run repeatedly.

$ btrfs filesystem resize -1G /path

$ btrfs filesystem resize 1:-1G /path

Shrink size of the filesystem's device id 1 by 1GiB. The first syntax expects a device with id 1 to exist, otherwise fails. The second is equivalent and more explicit. For a single-device filesystem it's typically not necessary to specify the devid though.

$ btrfs filesystem resize max /path

$ btrfs filesystem resize 1:max /path

Let's assume that devid 1 exists and the filesystem does not occupy the whole block device, eg. it has been enlarged and we want to grow the filesystem. By simply using *max* as size we will achieve that.

Note: There are two ways to minimize the filesystem on a given device. The **btrfs inspect-internal min-dev-size** command, or iteratively shrink in steps.

2.7.5 EXIT STATUS

btrfs filesystem returns a zero exit status if it succeeds. Non zero is returned in case of failure.

2.7.6 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.7.7 SEE ALSO

`btrfs-subvolume(8)`, `mkfs.btrfs(8)`

2.8 btrfs-find-root(8)

2.8.1 SYNOPSIS

btrfs-find-root [options] <device>

2.8.2 DESCRIPTION

btrfs-find-root is used to find the satisfied root, you can filter by root tree's objectid, generation, level.

2.8.3 OPTIONS

-a	Search through all metadata extents, even the root has been already found.
-g <generation>	Filter root tree by it's original transaction id, tree root's generation in default.
-o <objectid>	Filter root tree by it's objectid,tree root's objectid in default.
-l <level>	Filter root tree by b-tree's level, level 0 in default.

2.8.4 EXIT STATUS

btrfs-find-root will return 0 if no error happened. If any problems happened, 1 will be returned.

2.8.5 SEE ALSO

`mkfs.btrfs(8)`

2.9 btrfs-image(8)

2.9.1 SYNOPSIS

btrfs-image [options] <source> <target>

2.9.2 DESCRIPTION

btrfs-image is used to create an image of a btrfs filesystem. All data will be zeroed, but metadata and the like is preserved. Mainly used for debugging purposes.

In the dump mode, source is the btrfs device/file and target is the output file (use - for stdout).

In the restore mode (option *-r*), source is the dumped image and target is the btrfs device/file.

2.9.3 OPTIONS

-r Restore metadump image. By default, this fixes super's chunk tree, by using 1 stripe pointing to primary device, so that file system can be restored by running tree log reply if possible. To restore without changing number of stripes in chunk tree check *-o* option.

-c <value> Compression level (0 ~ 9).

-t <value> Number of threads (1 ~ 32) to be used to process the image dump or restore.

-o Use the old restore method, this does not fixup the chunk tree so the restored file system will not be able to be mounted.

-s Sanitize the file names when generating the image. One -s means just generate random garbage, which means that the directory indexes won't match up since the hashes won't match with the garbage filenames. Using -ss will calculate a collision for the filename so that the hashes match, and if it can't calculate a collision then it will just generate garbage. The collision calculator is very time and CPU intensive so only use it if you are having problems with your file system tree and need to have it mostly working.

-w Walk all the trees manually and copy any blocks that are referenced. Use this option if your extent tree is corrupted to make sure that all of the metadata is captured.

-m Restore for multiple devices, more than 1 device should be provided.

2.9.4 EXIT STATUS

btrfs-image will return 0 if no error happened. If any problems happened, 1 will be returned.

2.9.5 SEE ALSO

`mkfs.btrfs(8)`

2.10 btrfs-inspect-internal(8)

2.10.1 SYNOPSIS

btrfs inspect-internal <subcommand> <args>

2.10.2 DESCRIPTION

This command group provides an interface to query internal information. The functionality ranges from a simple UI to an ioctl or a more complex query that assembles the result from several internal structures. The latter usually requires calls to privileged ioctls.

2.10.3 SUBCOMMAND

dump-super [options] <device> [device...]
(replaces the standalone tool **btrfs-show-super**)

Show btrfs superblock information stored on given devices in textual form. By default the first superblock is printed, more details about all copies or additional backup data can be printed.

Besides verification of the filesystem signature, there are no other sanity checks. The superblock checksum status is reported, the device item and filesystem UUIDs are checked and reported.

Note: The meaning of option *-s* has changed in version 4.8 to be consistent with other tools to specify superblock copy rather the offset. The old way still works, but prints a warning. Please update your scripts to use *--bytenr* instead. The option *-i* has been deprecated.

`Options`

-f|--full
print full superblock information, including the system chunk array and backup roots

-a|--all
print information about all present superblock copies (cannot be used together with *-s* option)

-i <super> (deprecated since 4.8, same behaviour as *--super*)

--bytenr <bytenr> specify offset to a superblock in a non-standard location at *bytenr*, useful for debugging (disables the *-f* option)

If there are multiple options specified, only the last one applies.

-F|--force
attempt to print the superblock even if a valid BTRFS signature is not found; the result may be completely wrong if the data does not resemble a superblock

-s|--super <bytenr>
(see compatibility note above)

specify which mirror to print, valid values are 0, 1 and 2 and the superblock must be present on the device with a valid signature, can be used together with *--force*

dump-tree [options] <device> [device...]
(replaces the standalone tool **btrfs-debug-tree**)

Dump tree structures from a given device in textual form, expand keys to human readable equivalents where possible. This is useful for analyzing filesystem state or inconsistencies and has a positive educational effect on understanding the internal filesystem structure.

Note: Contains file names, consider that if you're asked to send the dump for analysis. Does not contain file data.

`Options`

-e|--extents
print only extent-related information: extent and device trees

-d|--device
print only device-related information: tree root, chunk and device trees

-r|--roots
print only short root node information, ie. the root tree keys

-R|--backups
same as *--roots* plus print backup root info, ie. the backup root keys and the respective tree root block offset

-u|--uuid
print only the uuid tree information, empty output if the tree does not exist

-b <block_num> print info of the specified block only, can be specified multiple times

--follow use with *-b*, print all children tree blocks of <*block_num*>

--dfs (default up to 5.2)

use depth-first search to print trees, the nodes and leaves are intermixed in the output

--bfs (default since 5.3)

use breadth-first search to print trees, the nodes are printed before all leaves

--hide-names print a placeholder *HIDDEN* instead of various names, useful for developers to inspect the dump while keeping potentially sensitive information hidden

This is:

- directory entries (files, directories, subvolumes)
- default subvolume
- extended attributes (name, value)
- hardlink names (if stored inside another item or as extended references in standalone items)

Note: Lengths are not hidden because they can be calculated from the item size anyway.

--csum-headers print b-tree node checksums stored in headers (metadata)

--csum-items print checksums stored in checksum items (data)

--noscan do not automatically scan the system for other devices from the same filesystem, only use the devices provided as the arguments

-t <tree_id> print only the tree with the specified ID, where the ID can be numerical or common name in a flexible human readable form

The tree id name recognition rules:

- case does not matter
- the C source definition, eg. BTRFS_ROOT_TREE_OBJECTID
- short forms without BTRFS_ prefix, without _TREE and _OBJECTID suffix, eg. ROOT_TREE, ROOT
- convenience aliases, eg. DEVICE for the DEV tree, CHECKSUM for CSUM
- unrecognized ID is an error

inode-resolve [-v] <ino> <path>
(needs root privileges)

resolve paths to all files with given inode number *ino* in a given subvolume at *path*, ie. all hardlinks

`Options`

-v (deprecated) alias for global *-v* option

logical-resolve [-Pvo] [-s <bufsize>] <logical> <path>
(needs root privileges)

resolve paths to all files at given *logical* address in the linear filesystem space

`Options`

-P skip the path resolving and print the inodes instead

-o ignore offsets, find all references to an extent instead of a single block. Requires kernel support for the V2 ioctl (added in 4.15). The results might need further processing to filter out unwanted extents by the offset that is supposed to be obtained by other means.

-s <bufsize> set internal buffer for storing the file names to *bufsize*, default is 64KiB, maximum 16MiB. Buffer sizes over 64Kib require kernel support for the V2 ioctl (added in 4.15).

-v (deprecated) alias for global *-v* option

min-dev-size [options] <path>
(needs root privileges)

return the minimum size the device can be shrunk to, without performing any resize operation, this may be useful before executing the actual resize operation

`Options`

--id <id> specify the device *id* to query, default is 1 if this option is not used

rootid <path>
for a given file or directory, return the containing tree root id, but for a subvolume itself return its own tree id (ie. subvol id)

Note: The result is undefined for the so-called empty subvolumes (identified by inode number 2), but such a subvolume does not contain any files anyway

---

subvolid-resolve <subvolid> <path>
(needs root privileges)

resolve the absolute path of the subvolume id *subvolid*

tree-stats [options] <device>
(needs root privileges)

Print sizes and statistics of trees.

`Options`

-b Print raw numbers in bytes.

2.10.4 EXIT STATUS

btrfs inspect-internal returns a zero exit status if it succeeds. Non zero is returned in case of failure.

2.10.5 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.10.6 SEE ALSO

`mkfs.btrfs(8)`

2.11 btrfs-map-logical(8)

2.11.1 SYNOPSIS

btrfs-map-logical <options> <device>

2.11.2 DESCRIPTION

btrfs-map-logical can be used to find out what the physical offsets are on the mirrors, the result is dumped to stdout by default.

Mainly used for debug purpose.

2.11.3 OPTIONS

-l|--logical <logical_num>
Logical extent to map.

-c|--copy <copy>
Copy of the extent to read(usually 1 or 2).

-o|--output <filename>
Output file to hold the extent.

-b|--bytes <bytes>
Number of bytes to read.

2.11.4 EXIT STATUS

btrfs-map-logical will return 0 if no error happened. If any problems happened, 1 will be returned.

2.11.5 SEE ALSO

`mkfs.btrfs(8)`

2.12 btrfs-property(8)

2.12.1 SYNOPSIS

btrfs property <subcommand> <args>

2.12.2 DESCRIPTION

btrfs property is used to get/set/list property for given filesystem object. The object can be an inode (file or directory), subvolume or the whole filesystem.

btrfs property provides an unified and user-friendly method to tune different btrfs properties instead of using the traditional method like `chattr(1)` or `lsattr(1)`.

Object types

A property might apply to several object types so in some cases it's necessary to specify that explicity, however it's not needed in the most common case of files and directories.

The subcommands take parameter *-t*, use first letter as a shortcut (*f/s/d/i*) of the type:

- filesystem
- subvolume
- device
- inode (file or directory)

Inode properties

compression
: compression algorithm set for an inode (it's not possible to set the compression level this way), possible values:

  - *lzo*
  - *zlib*
  - *zstd*
  - *no* or *none* - disable compresssion (equivalent to `chattr +m`)
  - *""* (empty string) - set the default value

  Note: This has changed in version 5.18 of btrfs-progs and requires kernel 5.14 or newer to work.

Subvolume properties

ro
read-only flag of subvolume: true or false. Please also see section *SUBVOLUME FLAGS* in `btrfs-subvolume(8)` for possible implications regarding incremental send.

Filesystem properties

label
label of the filesystem. For an unmounted filesystem, provide a path to a block device as object. For a mounted filesystem, specify a mount point.

2.12.3 SUBCOMMAND

get [-t <type>] <object> [<name>]
Read value of a property *name* of btrfs *object* of given *type*, empty *name* will read all of them

list [-t <type>] <object>
List available properties with their descriptions for the given object.

set [-f] [-t <type>] <object> <name> <value>
Set *value* of property *name* on a given btrfs object.

`Options`

-f Force the change. Changing some properties may involve safety checks or additional changes that depend on the properties semantics.

2.12.4 EXAMPLES

Set compression on a file:

```
$ touch file1
$ btrfs prop get file1
[ empty output ]
$ btrfs prop set file1 compression zstd
$ btrfs prop get file1
compression=zstd
```

Make a writeable subvolume read-only:

```
$ btrfs subvol create subvol1
[ fill subvol1 with data ]
$ btrfs prop get subvol1
ro=false
$ btrfs prop set subvol1 ro true
ro=true
```

2.12.5 EXIT STATUS

btrfs property returns a zero exit status if it succeeds. Non zero is returned in case of failure.

2.12.6 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.12.7 SEE ALSO

`mkfs.btrfs(8)`, `lsattr(1)`, `chattr(1)`

2.13 btrfs-qgroup(8)

2.13.1 SYNOPSIS

btrfs qgroup <subcommand> <args>

2.13.2 DESCRIPTION

btrfs qgroup is used to control quota group (qgroup) of a btrfs filesystem.

Note: To use qgroup you need to enable quota first using **btrfs quota enable** command.

Warning: Qgroup is not stable yet and will impact performance in current mainline kernel (v4.14).

2.13.3 QGROUP

Quota groups or qgroup in btrfs make a tree hierarchy, the leaf qgroups are attached to subvolumes. The size limits are set per qgroup and apply when any limit is reached in tree that contains a given subvolume.

The limits are separated between shared and exclusive and reflect the extent ownership. For example a fresh snapshot shares almost all the blocks with the original subvolume, new writes to either subvolume will raise towards the exclusive limit.

The qgroup identifiers conform to *level/id* where level 0 is reserved to the qgroups associated with subvolumes. Such qgroups are created automatically.

The qgroup hierarchy is built by commands **create** and **assign**.

Note: If the qgroup of a subvolume is destroyed, quota about the subvolume will not be functional until qgroup *0/<subvolume id>* is created again.

2.13.4 SUBCOMMAND

assign [options] <src> <dst> <path>
Assign qgroup *src* as the child qgroup of *dst* in the btrfs filesystem identified by *path*.

`Options`

--rescan (default since: 4.19) Automatically schedule quota rescan if the new qgroup assignment would lead to quota inconsistency. See *QUOTA RESCAN* for more information.

--no-rescan Explicitly ask not to do a rescan, even if the assignment will make the quotas inconsistent. This may be useful for repeated calls where the rescan would add unnecessary overhead.

create <qgroupid> <path>
Create a subvolume quota group.

For the *0/<subvolume id>* qgroup, a qgroup can be created even before the subvolume is created.

destroy <qgroupid> <path>
Destroy a qgroup.

If a qgroup is not isolated, meaning it is a parent or child qgroup, then it can only be destroyed after the relationship is removed.

limit [options] <size>|none [<qgroupid>] <path>
Limit the size of a qgroup to *size* or no limit in the btrfs filesystem identified by *path*.

If *qgroupid* is not given, qgroup of the subvolume identified by *path* is used if possible.

`Options`

-c limit amount of data after compression. This is the default, it is currently not possible to turn off this option.

-e limit space exclusively assigned to this qgroup.

remove <src> <dst> <path>
Remove the relationship between child qgroup *src* and parent qgroup *dst* in the btrfs filesystem identified by *path*.

`Options`

--rescan (default since: 4.19) Automatically schedule quota rescan if the removed qgroup relation would lead to quota inconsistency. See *QUOTA RESCAN* for more information.

--no-rescan Explicitly ask not to do a rescan, even if the removal will make the quotas inconsistent. This may be useful for repeated calls where the rescan would add unnecessary overhead.

show [options] <path>
Show all qgroups in the btrfs filesystem identified by <path>.

`Options`

-p print parent qgroup id.

-c print child qgroup id.

-r print limit of referenced size of qgroup.

-e print limit of exclusive size of qgroup.

-F list all qgroups which impact the given path(include ancestral qgroups)

-f list all qgroups which impact the given path(exclude ancestral qgroups)

--raw raw numbers in bytes, without the *B* suffix.

--human-readable print human friendly numbers, base 1024, this is the default

--iec select the 1024 base for the following options, according to the IEC standard.

--si select the 1000 base for the following options, according to the SI standard.

--kbytes show sizes in KiB, or kB with --si.

--mbytes show sizes in MiB, or MB with --si.

--gbytes show sizes in GiB, or GB with --si.

--tbytes show sizes in TiB, or TB with --si.

--sort=[+/-]<attr>[,[+/-]<attr>]...
list qgroups in order of <attr>.

<attr> can be one or more of qgroupid,rfer,excl,max_rfer,max_excl.

Prefix + means ascending order and - means descending order of *attr*. If no prefix is given, use ascending order by default.

If multiple *attr* values are given, use comma to separate.

--sync To retrieve information after updating the state of qgroups, force sync of the filesystem identified by *path* before getting information.

2.13.5 QUOTA RESCAN

The rescan reads all extent sharing metadata and updates the respective qgoups accordingly.

The information consists of bytes owned exclusively (*excl*) or shared/referred to (*rfer*). There's no explicit information about which extents are shared or owned exclusively. This means when qgroup relationship changes, extent owners change and qgroup numbers are no longer consistent unless we do a full rescan.

However there are cases where we can avoid a full rescan, if a subvolume whose *rfer* number equals its *excl* number, which means all bytes are exclusively owned, then assigning/removing this subvolume only needs to add/subtract *rfer* number from its parent qgroup. This can speed up the rescan.

2.13.6 EXAMPLES

Make a parent group that has two quota group children

Given the following filesystem mounted at */mnt/my-vault*

```
Label: none  uuid: 60d2ab3b-941a-4f22-8d1a-315f329797b2
       Total devices 1 FS bytes used 128.00KiB
       devid    1 size 5.00GiB used 536.00MiB path /dev/vdb
```

Enable quota and create subvolumes. Check subvolume ids.

```
$ cd /mnt/my-vault
$ btrfs quota enable .
$ btrfs subvolume create a
```

(continues on next page)

(continued from previous page)

```
$ btrfs subvolume create b
$ btrfs subvolume list .

ID 261 gen 61 top level 5 path a
ID 262 gen 62 top level 5 path b
```

Create qgroup and set limit to 10MiB.

```
$ btrfs qgroup create 1/100 .
$ btrfs qgroup limit 10M 1/100 .
$ btrfs qgroup assign 0/261 1/100 .
$ btrfs qgroup assign 0/262 1/100 .
```

And check qgroups.

```
$ btrfs qgroup show .

qgroupid         rfer         excl
--------         ----         ----
0/5          16.00KiB     16.00KiB
0/261        16.00KiB     16.00KiB
0/262        16.00KiB     16.00KiB
1/100        32.00KiB     32.00KiB
```

2.13.7 EXIT STATUS

btrfs qgroup returns a zero exit status if it succeeds. Non zero is returned in case of failure.

2.13.8 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.13.9 SEE ALSO

`mkfs.btrfs(8)`, `btrfs-subvolume(8)`, `btrfs-quota(8)`,

2.14 btrfs-quota(8)

2.14.1 SYNOPSIS

btrfs quota <subcommand> <args>

2.14.2 DESCRIPTION

The commands under **btrfs quota** are used to affect the global status of quotas of a btrfs filesystem. The quota groups (qgroups) are managed by the subcommand `btrfs-qgroup(8)`.

Note: Qgroups are different than the traditional user quotas and designed to track shared and exclusive data per-subvolume. Please refer to the section *HIERARCHICAL QUOTA GROUP CONCEPTS* for a detailed description.

PERFORMANCE IMPLICATIONS

When quotas are activated, they affect all extent processing, which takes a performance hit. Activation of qgroups is not recommended unless the user intends to actually use them.

STABILITY STATUS

The qgroup implementation has turned out to be quite difficult as it affects the core of the filesystem operation. Qgroup users have hit various corner cases over time, such as incorrect accounting or system instability. The situation is gradually improving and issues found and fixed.

2.14.3 HIERARCHICAL QUOTA GROUP CONCEPTS

The concept of quota has a long-standing tradition in the Unix world. Ever since computers allow multiple users to work simultaneously in one filesystem, there is the need to prevent one user from using up the entire space. Every user should get his fair share of the available resources.

In case of files, the solution is quite straightforward. Each file has an *owner* recorded along with it, and it has a size. Traditional quota just restricts the total size of all files that are owned by a user. The concept is quite flexible: if a user hits his quota limit, the administrator can raise it on the fly.

On the other hand, the traditional approach has only a poor solution to restrict directories. At installation time, the harddisk can be partitioned so that every directory (eg. /usr, /var/, ...) that needs a limit gets its own partition. The obvious problem is that those limits cannot be changed without a reinstallation. The btrfs subvolume feature builds a bridge. Subvolumes correspond in many ways to partitions, as every subvolume looks like its own filesystem. With subvolume quota, it is now possible to restrict each subvolume like a partition, but keep the flexibility of quota. The space for each subvolume can be expanded or restricted on the fly.

As subvolumes are the basis for snapshots, interesting questions arise as to how to account used space in the presence of snapshots. If you have a file shared between a subvolume and a snapshot, whom to account the file to? The creator? Both? What if the file gets modified in the snapshot, should only these changes be accounted to it? But wait, both the snapshot and the subvolume belong to the same user home. I just want to limit the total space used by both! But somebody else might not want to charge the snapshots to the users.

Btrfs subvolume quota solves these problems by introducing groups of subvolumes and let the user put limits on them. It is even possible to have groups of groups. In the following, we refer to them as *qgroups*.

Each qgroup primarily tracks two numbers, the amount of total referenced space and the amount of exclusively referenced space.

referenced
: space is the amount of data that can be reached from any of the subvolumes contained in the qgroup, while

exclusive
: is the amount of data where all references to this data can be reached from within this qgroup.

Subvolume quota groups

The basic notion of the Subvolume Quota feature is the quota group, short qgroup. Qgroups are notated as *level/id*, eg. the qgroup 3/2 is a qgroup of level 3. For level 0, the leading '0/' can be omitted. Qgroups of level 0 get created automatically when a subvolume/snapshot gets created. The ID of the qgroup corresponds to the ID of the subvolume, so 0/5 is the qgroup for the root subvolume. For the `btrfs qgroup` command, the path to the subvolume can also be used instead of *0/ID*. For all higher levels, the ID can be chosen freely.

Each qgroup can contain a set of lower level qgroups, thus creating a hierarchy of qgroups. Figure 1 shows an example qgroup tree.

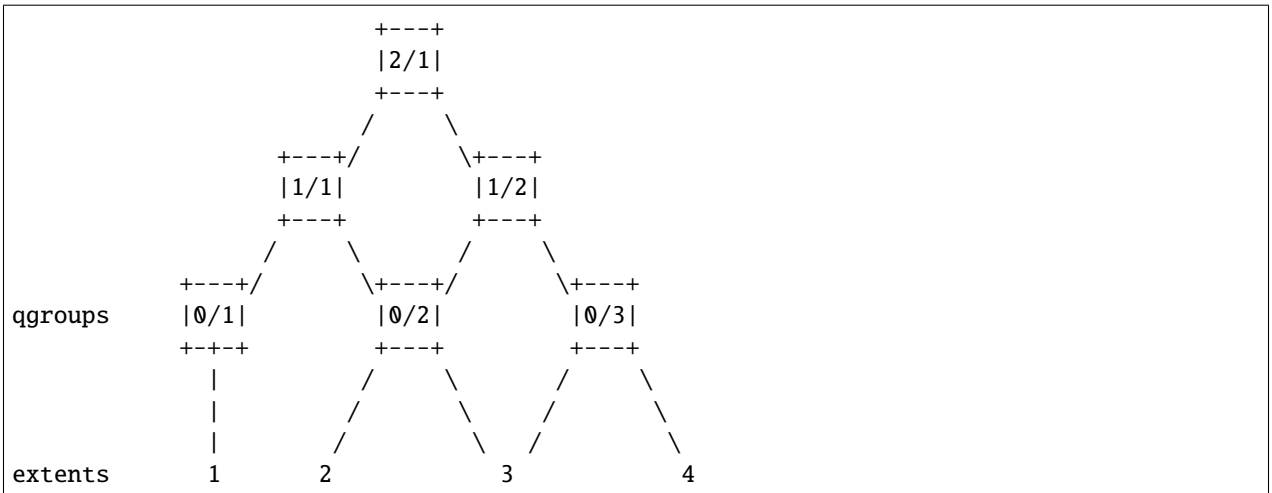

Figure 1: Sample qgroup hierarchy

At the bottom, some extents are depicted showing which qgroups reference which extents. It is important to understand the notion of *referenced* vs *exclusive*. In the example, qgroup 0/2 references extents 2 and 3, while 1/2 references extents 2-4, 2/1 references all extents.

On the other hand, extent 1 is exclusive to 0/1, extent 2 is exclusive to 0/2, while extent 3 is neither exclusive to 0/2 nor to 0/3. But because both references can be reached from 1/2, extent 3 is exclusive to 1/2. All extents are exclusive to 2/1.

So exclusive does not mean there is no other way to reach the extent, but it does mean that if you delete all subvolumes contained in a qgroup, the extent will get deleted.

Exclusive of a qgroup conveys the useful information how much space will be freed in case all subvolumes of the qgroup get deleted.

All data extents are accounted this way. Metadata that belongs to a specific subvolume (i.e. its filesystem tree) is also accounted. Checksums and extent allocation information are not accounted.

In turn, the referenced count of a qgroup can be limited. All writes beyond this limit will lead to a 'Quota Exceeded' error.

Inheritance

Things get a bit more complicated when new subvolumes or snapshots are created. The case of (empty) subvolumes is still quite easy. If a subvolume should be part of a qgroup, it has to be added to the qgroup at creation time. To add it at a later time, it would be necessary to at least rescan the full subvolume for a proper accounting.

Creation of a snapshot is the hard case. Obviously, the snapshot will reference the exact amount of space as its source, and both source and destination now have an exclusive count of 0 (the filesystem nodesize to be precise, as the roots of the trees are not shared). But what about qgroups of higher levels? If the qgroup contains both the source and the destination, nothing changes. If the qgroup contains only the source, it might lose some exclusive.

But how much? The tempting answer is, subtract all exclusive of the source from the qgroup, but that is wrong, or at least not enough. There could have been an extent that is referenced from the source and another subvolume from that qgroup. This extent would have been exclusive to the qgroup, but not to the source subvolume. With the creation of the snapshot, the qgroup would also lose this extent from its exclusive set.

So how can this problem be solved? In the instant the snapshot gets created, we already have to know the correct exclusive count. We need to have a second qgroup that contains all the subvolumes as the first qgroup, except the subvolume we want to snapshot. The moment we create the snapshot, the exclusive count from the second qgroup needs to be copied to the first qgroup, as it represents the correct value. The second qgroup is called a tracking qgroup. It is only there in case a snapshot is needed.

Use cases

Below are some use cases that do not mean to be extensive. You can find your own way how to integrate qgroups.

Single-user machine

`Replacement for partitions`

The simplest use case is to use qgroups as simple replacement for partitions. Btrfs takes the disk as a whole, and /, /usr, /var, etc. are created as subvolumes. As each subvolume gets it own qgroup automatically, they can simply be restricted. No hierarchy is needed for that.

`Track usage of snapshots`

When a snapshot is taken, a qgroup for it will automatically be created with the correct values. 'Referenced' will show how much is in it, possibly shared with other subvolumes. 'Exclusive' will be the amount of space that gets freed when the subvolume is deleted.

Multi-user machine

`Restricting homes`

When you have several users on a machine, with home directories probably under /home, you might want to restrict /home as a whole, while restricting every user to an individual limit as well. This is easily accomplished by creating a qgroup for /home , eg. 1/1, and assigning all user subvolumes to it. Restricting this qgroup will limit /home, while every user subvolume can get its own (lower) limit.

`Accounting snapshots to the user`

Let's say the user is allowed to create snapshots via some mechanism. It would only be fair to account space used by the snapshots to the user. This does not mean the user doubles his usage as soon as he takes a snapshot. Of course, files that are present in his home and the snapshot should only be accounted once. This can be accomplished by creating a qgroup for each user, say '1/UID'. The user home and all snapshots are assigned to this qgroup. Limiting it will extend

the limit to all snapshots, counting files only once. To limit /home as a whole, a higher level group 2/1 replacing 1/1 from the previous example is needed, with all user qgroups assigned to it.

`Do not account snapshots`

On the other hand, when the snapshots get created automatically, the user has no chance to control them, so the space used by them should not be accounted to him. This is already the case when creating snapshots in the example from the previous section.

`Snapshots for backup purposes`

This scenario is a mixture of the previous two. The user can create snapshots, but some snapshots for backup purposes are being created by the system. The user's snapshots should be accounted to the user, not the system. The solution is similar to the one from section 'Accounting snapshots to the user', but do not assign system snapshots to user's qgroup.

2.14.4 SUBCOMMAND

disable <path>
Disable subvolume quota support for a filesystem.

enable <path>
Enable subvolume quota support for a filesystem.

rescan [-s] <path>
Trash all qgroup numbers and scan the metadata again with the current config.

`Options`

-s show status of a running rescan operation.

-w wait for rescan operation to finish(can be already in progress).

2.14.5 EXIT STATUS

btrfs quota returns a zero exit status if it succeeds. Non zero is returned in case of failure.

2.14.6 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.14.7 SEE ALSO

`mkfs.btrfs(8)`, `btrfs-subvolume(8)`, `btrfs-qgroup(8)`

2.15 btrfs-receive(8)

2.15.1 SYNOPSIS

btrfs receive [options] <path>

or

btrfs receive --dump [options]

2.15.2 DESCRIPTION

Receive a stream of changes and replicate one or more subvolumes that were previously generated by **btrfs send**. The received subvolumes are stored to *path*, unless *--dump* option is given.

If *--dump* option is specified, **btrfs receive** will only do the validation of the stream, and print the stream metadata, one operation per line.

btrfs receive will fail in the following cases:

1. receiving subvolume already exists
2. previously received subvolume has been changed after it was received
3. default subvolume has changed or you didn't mount the filesystem at the toplevel subvolume

A subvolume is made read-only after the receiving process finishes successfully (see BUGS below).

`Options`

-f <FILE> read the stream from *FILE* instead of stdin,

-C|--chroot
confine the process to *path* using `chroot(1)`

-e terminate after receiving an *end cmd* marker in the stream.

Without this option the receiver side terminates only in case of an error on end of file.

-E|--max-errors <NERR>
terminate as soon as NERR errors occur while stream processing commands from the stream

Default value is 1. A value of 0 means no limit.

-m <ROOTMOUNT> the root mount point of the destination filesystem

By default the mountpoint is searched in */proc/self/mounts*. If */proc* is not accessible, eg. in a chroot environment, use this option to tell us where this filesystem is mounted.

--force-decompress if the stream contains compressed data (see *--compressed-data* in `btrfs-send(8)`), always decompress it instead of writing it with encoded I/O

--dump dump the stream metadata, one line per operation

Does not require the *path* parameter. The filesystem remains unchanged.

-q|--quiet
(deprecated) alias for global *-q* option

-v (deprecated) alias for global *-v* option

`Global options`

-v|--verbose
increase verbosity about performed actions, print details about each operation

-q|--quiet
suppress all messages except errors

2.15.3 BUGS

btrfs receive sets the subvolume read-only after it completes successfully. However, while the receive is in progress, users who have write access to files or directories in the receiving *path* can add, remove, or modify files, in which case the resulting read-only subvolume will not be an exact copy of the sent subvolume.

If the intention is to create an exact copy, the receiving *path* should be protected from access by users until the receive operation has completed and the subvolume is set to read-only.

Additionally, receive does not currently do a very good job of validating that an incremental send stream actually makes sense, and it is thus possible for a specially crafted send stream to create a subvolume with reflinks to arbitrary files in the same filesystem. Because of this, users are advised to not use *btrfs receive* on send streams from untrusted sources, and to protect trusted streams when sending them across untrusted networks.

2.15.4 EXIT STATUS

btrfs receive returns a zero exit status if it succeeds. Non zero is returned in case of failure.

2.15.5 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.15.6 SEE ALSO

`mkfs.btrfs(8)`, `btrfs-send(8)`

2.16 btrfs-replace(8)

2.16.1 SYNOPSIS

btrfs replace <subcommand> <args>

2.16.2 DESCRIPTION

btrfs replace is used to replace btrfs managed devices with other device.

2.16.3 SUBCOMMAND

cancel <mount_point>
: Cancel a running device replace operation.

start [options] <srcdev>|<devid> <targetdev> <path>
: Replace device of a btrfs filesystem.

  On a live filesystem, duplicate the data to the target device which is currently stored on the source device. If the source device is not available anymore, or if the -r option is set, the data is built only using the RAID redundancy mechanisms. After completion of the operation, the source device is removed from the filesystem. If the *srcdev* is a numerical value, it is assumed to be the device id of the filesystem which is mounted at *path*, otherwise it is the path to the source device. If the source device is disconnected, from the system, you have to use the devid parameter format. The *targetdev* needs to be same size or larger than the *srcdev*.

Note: The filesystem has to be resized to fully take advantage of a larger target device; this can be achieved with `btrfs filesystem resize <devid>:max /path`

`Options`

-r only read from *srcdev* if no other zero-defect mirror exists. (enable this if your drive has lots of read errors, the access would be very slow)

-f force using and overwriting *targetdev* even if it looks like it contains a valid btrfs filesystem.

A valid filesystem is assumed if a btrfs superblock is found which contains a correct checksum. Devices that are currently mounted are never allowed to be used as the *targetdev*.

-B no background replace.

--enqueue wait if there's another exclusive operation running, otherwise continue

-K|--nodiscard
Do not perform whole device TRIM operation on devices that are capable of that. This does not affect discard/trim operation when the filesystem is mounted. Please see the mount option *discard* for that in `btrfs(5)`.

status [-1] <mount_point>
Print status and progress information of a running device replace operation.

`Options`

-1 print once instead of print continuously until the replace operation finishes (or is cancelled)

2.16.4 EXAMPLES

Replacing an online drive with a bigger one

Given the following filesystem mounted at */mnt/my-vault*

```
Label: 'MyVault'  uuid: ae20903e-b72d-49ba-b944-901fc6d888a1
        Total devices 2 FS bytes used 1TiB
        devid    1 size 1TiB used 500.00GiB path /dev/sda
        devid    2 size 1TiB used 500.00GiB path /dev/sdb
```

In order to replace */dev/sda* (*devid 1*) with a bigger drive located at */dev/sdc* you would run the following:

```
btrfs replace start 1 /dev/sdc /mnt/my-vault/
```

You can monitor progress via:

```
btrfs replace status /mnt/my-vault/
```

After the replacement is complete, as per the docs at `btrfs-filesystem(8)` in order to use the entire storage space of the new drive you need to run:

```
btrfs filesystem resize 1:max /mnt/my-vault/
```

2.16.5 EXIT STATUS

btrfs replace returns a zero exit status if it succeeds. Non zero is returned in case of failure.

2.16.6 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.16.7 SEE ALSO

`mkfs.btrfs(8)`, `btrfs-device(8)`, `btrfs-filesystem(8)`,

2.17 btrfs-rescue(8)

2.17.1 SYNOPSIS

btrfs rescue <subcommand> <args>

2.17.2 DESCRIPTION

btrfs rescue is used to try to recover a damaged btrfs filesystem.

2.17.3 SUBCOMMAND

chunk-recover [options] <device>
Recover the chunk tree by scanning the devices

`Options`

-y assume an answer of *yes* to all questions.

-h help.

-v (deprecated) alias for global *-v* option

Note: Since **chunk-recover** will scan the whole device, it will be very slow especially executed on a large device.

---

fix-device-size <device>
fix device size and super block total bytes values that are do not match

Kernel 4.11 starts to check the device size more strictly and this might mismatch the stored value of total bytes. See the exact error message below. Newer kernel will refuse to mount the filesystem where the values do not match. This error is not fatal and can be fixed. This command will fix the device size values if possible.

```
BTRFS error (device sdb): super_total_bytes 92017859088384 mismatch with fs_devices␣
↪total_rw_bytes 92017859094528
```

The mismatch may also exhibit as a kernel warning:

```
WARNING: CPU: 3 PID: 439 at fs/btrfs/ctree.h:1559 btrfs_update_device+0x1c5/0x1d0␣
↪[btrfs]
```

clear-uuid-tree <device>
Clear uuid tree, so that kernel can re-generate it at next read-write mount.

Since kernel v4.16 there are more sanity check performed, and sometimes non-critical trees like uuid tree can cause problems and reject the mount. In such case, clearing uuid tree may make the filesystem to be mountable again without much risk as it's built from other trees.

super-recover [options] <device>
Recover bad superblocks from good copies.

Options

-y assume an answer of *yes* to all questions.

-v (deprecated) alias for global *-v* option

zero-log <device>
clear the filesystem log tree

This command will clear the filesystem log tree. This may fix a specific set of problem when the filesystem mount fails due to the log replay. See below for sample stacktraces that may show up in system log.

The common case where this happens was fixed a long time ago, so it is unlikely that you will see this particular problem, but the command is kept around.

Note: Clearing the log may lead to loss of changes that were made since the last transaction commit. This may be up to 30 seconds (default commit period) or less if the commit was implied by other filesystem activity.

One can determine whether **zero-log** is needed according to the kernel backtrace:

```
? replay_one_dir_item+0xb5/0xb5 [btrfs]
? walk_log_tree+0x9c/0x19d [btrfs]
? btrfs_read_fs_root_no_radix+0x169/0x1a1 [btrfs]
? btrfs_recover_log_trees+0x195/0x29c [btrfs]
? replay_one_dir_item+0xb5/0xb5 [btrfs]
? btree_read_extent_buffer_pages+0x76/0xbc [btrfs]
? open_ctree+0xff6/0x132c [btrfs]
```

If the errors are like above, then **zero-log** should be used to clear the log and the filesystem may be mounted normally again. The keywords to look for are 'open_ctree' which says that it's during mount and function names that contain *replay*, *recover* or *log_tree*.

2.17.4 EXIT STATUS

btrfs rescue returns a zero exit status if it succeeds. Non zero is returned in case of failure.

2.17.5 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.17.6 SEE ALSO

`mkfs.btrfs(8)`, `btrfs-scrub(8)`, `btrfs-check(8)`

2.18 btrfs-restore(8)

2.18.1 SYNOPSIS

btrfs restore [options] <device> <path> | -l <device>

2.18.2 DESCRIPTION

btrfs restore is used to try to salvage files from a damaged filesystem and restore them into *path* or just list the subvolume tree roots. The filesystem image is not modified.

If the filesystem is damaged and cannot be repaired by the other tools (`btrfs-check(8)` or `btrfs-rescue(8)`), **btrfs restore** could be used to retrieve file data, as far as the metadata are readable. The checks done by restore are less strict and the process is usually able to get far enough to retrieve data from the whole filesystem. This comes at a cost that some data might be incomplete or from older versions if they're available.

There are several options to attempt restoration of various file metadata type. You can try a dry run first to see how well the process goes and use further options to extend the set of restored metadata.

For images with damaged tree structures, there are several options to point the process to some spare copy.

Note: It is recommended to read the following btrfs wiki page if your data is not salvaged with default option:

https://btrfs.wiki.kernel.org/index.php/Restore

2.18.3 OPTIONS

-s|--snapshots
: get also snapshots that are skipped by default

-x|--xattr
: get extended attributes

-m|--metadata
: restore owner, mode and times for files and directories

-S|--symlinks
: restore symbolic links as well as normal files

-i|--ignore-errors
: ignore errors during restoration and continue

-o|--overwrite
: overwrite directories/files in *path*, eg. for repeated runs

-t <bytenr> use *bytenr* to read the root tree

-f <bytenr> only restore files that are under specified subvolume root pointed by *bytenr*

-u|--super <mirror>
use given superblock mirror identified by <mirror>, it can be 0,1 or 2

-r|--root <rootid>
only restore files that are under a specified subvolume whose objectid is *rootid*

-d find directory

-l|--list-roots
list subvolume tree roots, can be used as argument for *-r*

-D|--dry-run
dry run (only list files that would be recovered)

--path-regex <regex> restore only filenames matching a regular expression (`regex(7)`) with a mandatory format

```
^/(|home(|/username(|/Desktop(|/.*))))$
```

The format is not very comfortable and restores all files in the directories in the whole path, so this is not useful for restoring single file in a deep hierarchy.

-c ignore case (*--path-regex* only)

-v|--verbose
(deprecated) alias for global *-v* option

`Global options`

-v|--verbose
be verbose and print what is being restored

2.18.4 EXIT STATUS

btrfs restore returns a zero exit status if it succeeds. Non zero is returned in case of failure.

2.18.5 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.18.6 SEE ALSO

`mkfs.btrfs(8)`, `btrfs-rescue(8)`, `btrfs-check(8)`

2.19 btrfs-scrub(8)

2.19.1 SYNOPSIS

btrfs scrub <subcommand> <args>

2.19.2 DESCRIPTION

Scrub is a pass over all filesystem data and metadata and verifying the checksums. If a valid copy is available (replicated block group profiles) then the damaged one is repaired. All copies of the replicated profiles are validated.

Note: Scrub is not a filesystem checker (fsck) and does not verify nor repair structural damage in the filesystem. It really only checks checksums of data and tree blocks, it doesn't ensure the content of tree blocks is valid and consistent. There's some validation performed when metadata blocks are read from disk but it's not extensive and cannot substitute full *btrfs check* run.

The user is supposed to run it manually or via a periodic system service. The recommended period is a month but could be less. The estimated device bandwidth utilization is about 80% on an idle filesystem. The IO priority class is by default *idle* so background scrub should not significantly interfere with normal filesystem operation. The IO scheduler set for the device(s) might not support the priority classes though.

The scrubbing status is recorded in */var/lib/btrfs/* in textual files named *scrub.status.UUID* for a filesystem identified by the given UUID. (Progress state is communicated through a named pipe in file *scrub.progress.UUID* in the same directory.) The status file is updated every 5 seconds. A resumed scrub will continue from the last saved position.

Scrub can be started only on a mounted filesystem, though it's possible to scrub only a selected device. See **btrfs scrub start** for more.

2.19.3 SUBCOMMAND

cancel <path>|<device>
: If a scrub is running on the filesystem identified by *path* or *device*, cancel it.

  If a *device* is specified, the corresponding filesystem is found and **btrfs scrub cancel** behaves as if it was called on that filesystem. The progress is saved in the status file so **btrfs scrub resume** can continue from the last position.

resume [-BdqrR] [-c <ioprio_class> -n <ioprio_classdata>] <path>|<device>
: Resume a cancelled or interrupted scrub on the filesystem identified by *path* or on a given *device*. The starting point is read from the status file if it exists.

  This does not start a new scrub if the last scrub finished successfully.

  `Options`

  see **scrub start**.

start [-BdqrRf] [-c <ioprio_class> -n <ioprio_classdata>] <path>|<device>
: Start a scrub on all devices of the mounted filesystem identified by *path* or on a single *device*. If a scrub is already running, the new one will not start. A device of an unmounted filesystem cannot be scrubbed this way.

  Without options, scrub is started as a background process. The automatic repairs of damaged copies is performed by default for block group profiles with redundancy.

The default IO priority of scrub is the idle class. The priority can be configured similar to the `ionice(1)` syntax using *-c* and *-n* options. Note that not all IO schedulers honor the ionice settings.

`Options`

-B do not background and print scrub statistics when finished

-d print separate statistics for each device of the filesystem (*-B* only) at the end

-r run in read-only mode, do not attempt to correct anything, can be run on a read-only filesystem

-R raw print mode, print full data instead of summary

-c <ioprio_class> set IO priority class (see `ionice(1)` manpage)

-n <ioprio_classdata> set IO priority classdata (see `ionice(1)` manpage)

-f force starting new scrub even if a scrub is already running, this can useful when scrub status file is damaged and reports a running scrub although it is not, but should not normally be necessary

-q (deprecated) alias for global *-q* option

status [options] <path>|<device>

Show status of a running scrub for the filesystem identified by *path* or for the specified *device*.

If no scrub is running, show statistics of the last finished or cancelled scrub for that filesystem or device.

`Options`

-d print separate statistics for each device of the filesystem

-R print all raw statistics without postprocessing as returned by the status ioctl

--raw print all numbers raw values in bytes without the *B* suffix

--human-readable print human friendly numbers, base 1024, this is the default

--iec select the 1024 base for the following options, according to the IEC standard

--si select the 1000 base for the following options, according to the SI standard

--kbytes show sizes in KiB, or kB with --si

--mbytes show sizes in MiB, or MB with --si

--gbytes show sizes in GiB, or GB with --si

--tbytes show sizes in TiB, or TB with --si

2.19.4 EXIT STATUS

btrfs scrub returns a zero exit status if it succeeds. Non zero is returned in case of failure:

1

scrub couldn't be performed

2

there is nothing to resume

3

scrub found uncorrectable errors

2.19.5 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.19.6 SEE ALSO

`mkfs.btrfs(8)`, `ionice(1)`

2.20 btrfs-select-super(8)

2.20.1 SYNOPSIS

btrfs-select-super -s number <device>

2.20.2 DESCRIPTION

Destructively overwrite all copies of the superblock with a specified copy. This helps in certain cases, for example when write barriers were disabled during a power failure and not all superblocks were written, or if the primary superblock is damaged, eg. accidentally overwritten.

The filesystem specified by *device* must not be mounted.

Note: Prior to overwriting the primary superblock, please make sure that the backup copies are valid!

To dump a superblock use the **btrfs inspect-internal dump-super** command.

Then run the check (in the non-repair mode) using the command **btrfs check -s** where *-s* specifies the superblock copy to use.

Superblock copies exist in the following offsets on the device:

- primary: 64KiB (65536)
- 1st copy: 64MiB (67108864)
- 2nd copy: 256GiB (274877906944)

A superblock size is 4KiB (4096).

2.20.3 OPTIONS

-s|--super <N>
use Nth superblock copy, valid values are 0 1 or 2 if the respective superblock offset is within the device size

2.20.4 SEE ALSO

btrfs(8)

2.21 btrfs-send(8)

2.21.1 SYNOPSIS

btrfs send [-ve] [-p <parent>] [-c <clone-src>] [-f <outfile>] <subvol> [<subvol>...]

2.21.2 DESCRIPTION

This command will generate a stream of instructions that describe changes between two subvolume snapshots. The stream can be consumed by the **btrfs receive** command to replicate the sent snapshot on a different filesystem. The command operates in two modes: full and incremental.

All snapshots involved in one send command must be read-only, and this status cannot be changed as long as there's a running send operation that uses the snapshot.

In the full mode, the entire snapshot data and metadata will end up in the stream.

In the incremental mode (options *-p* and *-c*), previously sent snapshots that are available on both the sending and receiving side can be used to reduce the amount of information that has to be sent to reconstruct the sent snapshot on a different filesystem.

The *-p* <*parent*> option can be omitted when *-c* <*clone-src*> options are given, in which case **btrfs send** will determine a suitable parent from among the clone sources.

You must not specify clone sources unless you guarantee that these snapshots are exactly in the same state on both sides--both for the sender and the receiver. For implications of changed read-write status of a received snapshot please see section *SUBVOLUME FLAGS* in `btrfs-subvolume(8)`.

`Options`

-e if sending multiple subvolumes at once, use the new format and omit the 'end cmd' marker in the stream separating the subvolumes

-p <parent> send an incremental stream from *parent* to *subvol*

-c <clone-src> use this snapshot as a clone source for an incremental send (multiple allowed)

-f <outfile> output is normally written to standard output so it can be, for example, piped to btrfs receive. Use this option to write it to a file instead.

--no-data::
send in *NO_FILE_DATA* mode

The output stream does not contain any file data and thus cannot be used to transfer changes. This mode is faster and is useful to show the differences in metadata.

--proto <N> use send protocol version N

The default is 1, which was the original protocol version. Version 2 encodes file data slightly more efficiently; it is also required for sending compressed data directly (see *--compressed-data*). Version 2 requires at least btrfs-progs 5.18 on both the sender and receiver and at least Linux 5.18 on the sender. Passing 0 means to use the highest version supported by the running kernel.

--compressed-data send data that is compressed on the filesystem directly without decompressing it

If the receiver supports the *BTRFS_IOC_ENCODED_WRITE* ioctl (added in Linux 5.18), it can also write it directly without decompressing it. Otherwise, the receiver will fall back to decompressing it and writing it normally.

This requires protocol version 2 or higher. If *--proto* was not used, then *--compressed-data* implies *--proto 2*.

-q|--quiet
(deprecated) alias for global *-q* option

-v|--verbose
(deprecated) alias for global *-v* option

`Global options`

-q|--quiet
suppress all messages except errors

-v|--verbose
increase output verbosity, print generated commands in a readable form

2.21.3 EXIT STATUS

btrfs send returns a zero exit status if it succeeds. Non zero is returned in case of failure.

2.21.4 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.21.5 SEE ALSO

`mkfs.btrfs(8)`, `btrfs-receive(8)`, `btrfs-subvolume(8)`

2.22 btrfs-subvolume(8)

2.22.1 SYNOPSIS

btrfs subvolume <subcommand> [<args>]

2.22.2 DESCRIPTION

btrfs subvolume is used to create/delete/list/show btrfs subvolumes and snapshots.

A BTRFS subvolume is a part of filesystem with its own independent file/directory hierarchy and inode number namespace. Subvolumes can share file extents. A snapshot is also subvolume, but with a given initial content of the original subvolume. A subvolume has always inode number 256.

Note: A subvolume in BTRFS is not like an LVM logical volume, which is block-level snapshot while BTRFS subvolumes are file extent-based.

A subvolume looks like a normal directory, with some additional operations described below. Subvolumes can be renamed or moved, nesting subvolumes is not restricted but has some implications regarding snapshotting. The numeric id (called *subvolid* or *rootid*) of the subvolume is persistent and cannot be changed.

A subvolume in BTRFS can be accessed in two ways:

- like any other directory that is accessible to the user
- like a separately mounted filesystem (options *subvol* or *subvolid*)

In the latter case the parent directory is not visible and accessible. This is similar to a bind mount, and in fact the subvolume mount does exactly that.

A freshly created filesystem is also a subvolume, called *top-level*, internally has an id 5. This subvolume cannot be removed or replaced by another subvolume. This is also the subvolume that will be mounted by default, unless the default subvolume has been changed (see `btrfs subvolume set-default`).

A snapshot is a subvolume like any other, with given initial content. By default, snapshots are created read-write. File modifications in a snapshot do not affect the files in the original subvolume.

Subvolumes can be given capacity limits, through the qgroups/quota facility, but otherwise share the single storage pool of the whole btrfs filesystem. They may even share data between themselves (through deduplication or snapshotting).

Note: A snapshot is not a backup: snapshots work by use of BTRFS' copy-on-write behaviour. A snapshot and the original it was taken from initially share all of the same data blocks. If that data is damaged in some way (cosmic rays, bad disk sector, accident with dd to the disk), then the snapshot and the original will both be damaged. Snapshots are useful to have local online "copies" of the filesystem that can be referred back to, or to implement a form of deduplication, or to fix the state of a filesystem for making a full backup without anything changing underneath it. They do not in themselves make your data any safer.

2.22.3 Subvolume flags

The subvolume flag currently implemented is the *ro* property (read-only status). Read-write subvolumes have that set to *false*, snapshots as *true*. In addition to that, a plain snapshot will also have last change generation and creation generation equal.

Read-only snapshots are building blocks of incremental send (see `btrfs-send(8)`) and the whole use case relies on unmodified snapshots where the relative changes are generated from. Thus, changing the subvolume flags from read-only to read-write will break the assumptions and may lead to unexpected changes in the resulting incremental stream.

A snapshot that was created by send/receive will be read-only, with different last change generation, read-only and with set *received_uuid* which identifies the subvolume on the filesystem that produced the stream. The use case relies on matching data on both sides. Changing the subvolume to read-write after it has been received requires to reset the *received_uuid*. As this is a notable change and could potentially break the incremental send use case, performing it by **btrfs property set** requires force if that is really desired by user.

Note: The safety checks have been implemented in 5.14.2, any subvolumes previously received (with a valid *received_uuid*) and read-write status may exist and could still lead to problems with send/receive. You can use **btrfs subvolume show** to identify them. Flipping the flags to read-only and back to read-write will reset the *received_uuid* manually. There may exist a convenience tool in the future.

2.22.4 Nested subvolumes

There are no restrictions for subvolume creation, so it's up to the user how to organize them, whether to have a flat layout (all subvolumes are direct descendants of the toplevel one), or nested.

What should be mentioned early is that a snapshotting is not recursive, so a subvolume or a snapshot is effectively a barrier and no files in the nested appear in the snapshot. Instead there's a stub subvolume (also sometimes **empty subvolume** with the same name as original subvolume, with inode number 2). This can be used intentionally but could be confusing in case of nested layouts.

Case study: system root layouts

There are two ways how the system root directory and subvolume layout could be organized. The interesting usecase for root is to allow rollbacks to previous version, as one atomic step. If the entire filesystem hierarchy starting in "/" is in one subvolume, taking snapshot will encompass all files. This is easy for the snapshotting part but has undesirable consequences for rollback. For example, log files would get rolled back too, or any data that are stored on the root filesystem but are not meant to be rolled back either (database files, VM images, ...).

Here we could utilize the snapshotting barrier mentioned above, each directory that stores data to be preserved accross rollbacks is it's own subvolume. This could be eg. `/var`. Further more-fine grained partitioning could be done, eg. adding separate subvolumes for `/var/log`, `/var/cache` etc.

That there are separate subvolumes requrires separate actions to take the snapshots (here it gets disconnected from the system root snapshots). This needs to be taken care of by system tools, installers together with selection of which directories are highly recommended to be separate subvolumes.

2.22.5 Mount options

Mount options are of two kinds, generic (that are handled by VFS layer) and specific, handled by the filesystem. The following list shows which are applicable to individual subvolume mounts, while there are more options that always affect the whole filesystem:

- generic: noatime/relatime/..., nodev, nosuid, ro, rw, dirsync
- fs-specific: compress, autodefrag, nodatacow, nodatasum

An example of whole filesystem options is eg. *space_cache*, *rescue*, *device*, *skip_balance*, etc. The exceptional options are *subvol* and *subvolid* that are actually used for mounting a given subvolume and can be specified only once for the mount.

Subvolumes belong to a single filesystem and as implemented now all share the same specific mount options, changes done by remount have immediate effect. This may change in the future.

Mounting a read-write snapshot as read-only is possible and will not chnange the *ro* property and flag of the subvolume.

The name of the mounted subvolume is stored in file `/proc/self/mounts` in the 4th column:

```
27 21 0:19 /subv1 /mnt rw,relatime - btrfs /dev/sda rw,space_cache
           ^^^^^^
```

2.22.6 Inode numbers

A proper subvolume has always inode number 256. If a subvolume is nested and then a snapshot is taken, then the cloned directory entry representing the subvolume becomes empty and the inode has number 2. All other files and directories in the target snapshot preserve their original inode numbers.

Note: Inode number is not a filesystem-wide unique identifier, some applications assume that. Please user pair *subvolumeid:inodenumber* for that purpose.

2.22.7 Performance

Subvolume creation needs to flush dirty data that belong to the subvolume, this step may take some time, otherwise once there's nothing else to do, the snapshot is instant and in the metadata it only creates a new tree root copy.

Snapshot deletion has two phases: first its directory is deleted and the subvolume is added to a list, then the list is processed one by one and the data related to the subvolume get deleted. This is usually called *cleaning* and can take some time depending on the amount of shared blocks (can be a lot of metadata updates), and the number of currently queued deleted subvolumes.

2.22.8 SUBVOLUME AND SNAPSHOT

A subvolume is a part of filesystem with its own independent file/directory hierarchy. Subvolumes can share file extents. A snapshot is also subvolume, but with a given initial content of the original subvolume.

Note: A subvolume in btrfs is not like an LVM logical volume, which is block-level snapshot while btrfs subvolumes are file extent-based.

A subvolume looks like a normal directory, with some additional operations described below. Subvolumes can be renamed or moved, nesting subvolumes is not restricted but has some implications regarding snapshotting.

A subvolume in btrfs can be accessed in two ways:

- like any other directory that is accessible to the user
- like a separately mounted filesystem (options *subvol* or *subvolid*)

In the latter case the parent directory is not visible and accessible. This is similar to a bind mount, and in fact the subvolume mount does exactly that.

A freshly created filesystem is also a subvolume, called *top-level*, internally has an id 5. This subvolume cannot be removed or replaced by another subvolume. This is also the subvolume that will be mounted by default, unless the default subvolume has been changed (see subcommand *set-default*).

A snapshot is a subvolume like any other, with given initial content. By default, snapshots are created read-write. File modifications in a snapshot do not affect the files in the original subvolume.

2.22.9 SUBCOMMAND

create [-i <qgroupid>] [<dest>/]<name>
Create a subvolume *name* in *dest*.

If *dest* is not given, subvolume *name* will be created in the current directory.

`Options`

-i <qgroupid> Add the newly created subvolume to a qgroup. This option can be given multiple times.

delete [options] [<subvolume> [<subvolume>...]], delete -i|--subvolid <subvolid> <path>
Delete the subvolume(s) from the filesystem.

If *subvolume* is not a subvolume, btrfs returns an error but continues if there are more arguments to process.

If *--subvolid* is used, *path* must point to a btrfs filesystem. See `btrfs subvolume list` or `btrfs inspect-internal rootid` how to get the subvolume id.

The corresponding directory is removed instantly but the data blocks are removed later in the background. The command returns immediately. See `btrfs subvolume sync` how to wait until the subvolume gets completely removed.

The deletion does not involve full transaction commit by default due to performance reasons. As a consequence, the subvolume may appear again after a crash. Use one of the *--commit* options to wait until the operation is safely stored on the device.

The default subvolume (see `btrfs subvolume set-default`) cannot be deleted and returns error (EPERM) and this is logged to the system log. A subvolume that's currently involved in send (see `btrfs send`) also cannot be deleted until the send is finished. This is also logged in the system log.

`Options`

-c|--commit-after
wait for transaction commit at the end of the operation.

-C|--commit-each
wait for transaction commit after deleting each subvolume.

-i|--subvolid <subvolid>
subvolume id to be removed instead of the <path> that should point to the filesystem with the subvolume

-v|--verbose
(deprecated) alias for global *-v* option

find-new <subvolume> <last_gen>
List the recently modified files in a subvolume, after *last_gen* generation.

get-default <path>
Get the default subvolume of the filesystem *path*.

The output format is similar to **subvolume list** command.

list [options] [-G [+|-]<value>] [-C [+|-]<value>] [--sort=rootid,gen,ogen,path] <path>
List the subvolumes present in the filesystem *path*.

For every subvolume the following information is shown by default:

ID *ID* gen *generation* top level *parent_ID* path *path*

where *ID* is subvolume's (root)id, *generation* is an internal counter which is updated every transaction, *parent_ID* is the same as the parent subvolume's id, and *path* is the relative path of the subvolume to the top level subvolume.

The subvolume's ID may be used by the subvolume set-default command, or at mount time via the *subvolid=* option.

`Options`

Path filtering:

-o print only subvolumes below specified <path>.

-a print all the subvolumes in the filesystem and distinguish between absolute and relative path with respect to the given *path*.

Field selection:

-p print the parent ID (*parent* here means the subvolume which contains this subvolume).

-c print the ogeneration of the subvolume, aliases: ogen or origin generation.

-g print the generation of the subvolume (default).

-u print the UUID of the subvolume.

-q print the parent UUID of the subvolume (*parent* here means subvolume of which this subvolume is a snapshot).

-R print the UUID of the sent subvolume, where the subvolume is the result of a receive operation.

Type filtering:

-s only snapshot subvolumes in the filesystem will be listed.

-r only readonly subvolumes in the filesystem will be listed.

-d list deleted subvolumes that are not yet cleaned.

Other:

-t print the result as a table.

Sorting:

By default the subvolumes will be sorted by subvolume ID ascending.

-G [+|-]<value>
list subvolumes in the filesystem that its generation is >=, <= or = value. '+' means >= value, '-' means <= value, If there is neither '+' nor '-', it means = value.

-C [+|-]<value>
list subvolumes in the filesystem that its ogeneration is >=, <= or = value. The usage is the same to *-G* option.

--sort=rootid,gen,ogen,path
list subvolumes in order by specified items. you can add + or - in front of each items, + means ascending, - means descending. The default is ascending.

for *--sort* you can combine some items together by ,, just like *--sort=+ogen,-gen,path,rootid*.

set-default [<subvolume>|<id> <path>]
Set the default subvolume for the (mounted) filesystem.

Set the default subvolume for the (mounted) filesystem at *path*. This will hide the top-level subvolume (i.e. the one mounted with *subvol=/* or *subvolid=5*). Takes action on next mount.

There are two ways how to specify the subvolume, by *id* or by the *subvolume* path. The id can be obtained from **btrfs subvolume list**, **btrfs subvolume show** or **btrfs inspect-internal rootid**.

show [options] <path>
Show more information about a subvolume (UUIDs, generations, times, flags, related snapshots).

```
/mnt/btrfs/subvolume
        Name:                   subvolume
        UUID:                   5e076a14-4e42-254d-ac8e-55bebea982d1
        Parent UUID:            -
        Received UUID:          -
        Creation time:          2018-01-01 12:34:56 +0000
        Subvolume ID:           79
        Generation:             2844
        Gen at creation:        2844
        Parent ID:              5
        Top level ID:           5
        Flags:                  -
        Snapshot(s):
```

Options

-r|--rootid <ID>
show details about subvolume with root *ID*, looked up in *path*

-u|--uuid UUID
show details about subvolume with the given *UUID*, looked up in *path*

snapshot [-r] [-i <qgroupid>] <source> <dest>|[<dest>/]<name>
Create a snapshot of the subvolume *source* with the name *name* in the *dest* directory.

If only *dest* is given, the subvolume will be named the basename of *source*. If *source* is not a subvolume, btrfs returns an error.

Options

-r	Make the new snapshot read only.
-i <qgroupid>	Add the newly created subvolume to a qgroup. This option can be given multiple times.

sync <path> [subvolid...]
Wait until given subvolume(s) are completely removed from the filesystem after deletion. If no subvolume id is given, wait until all current deletion requests are completed, but do not wait for subvolumes deleted in the meantime.

Options

-s <N>	sleep N seconds between checks (default: 1)

2.22.10 EXAMPLES

Deleting a subvolume

If we want to delete a subvolume called *foo* from a btrfs volume mounted at */mnt/bar* we could run the following:

```
btrfs subvolume delete /mnt/bar/foo
```

2.22.11 EXIT STATUS

btrfs subvolume returns a zero exit status if it succeeds. A non-zero value is returned in case of failure.

2.22.12 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.22.13 SEE ALSO

`mkfs.btrfs(8)`, `mount(8)`, `btrfs-quota(8)`, `btrfs-qgroup(8)`, `btrfs-send(8)`

2.23 btrfstune(8)

2.23.1 SYNOPSIS

btrfstune [options] <device> [<device>...]

2.23.2 DESCRIPTION

btrfstune can be used to enable, disable, or set various filesystem parameters. The filesystem must be unmounted.

The common usecase is to enable features that were not enabled at mkfs time. Please make sure that you have kernel support for the features. You can find a complete list of features and kernel version of their introduction at https://btrfs.wiki.kernel.org/index.php/Changelog#By_feature . Also, the manual page `mkfs.btrfs(8)` contains more details about the features.

Some of the features could be also enabled on a mounted filesystem by other means. Please refer to the *FILESYSTEM FEATURES* in `btrfs(5)`.

2.23.3 OPTIONS

-f
: Allow dangerous changes, e.g. clear the seeding flag or change fsid. Make sure that you are aware of the dangers.

-m
: (since kernel: 5.0)

  change fsid stored as 'metadata_uuid' to a randomly generated UUID, see also '-U'

-M <UUID>
: (since kernel: 5.0)

  change fsid stored as *metadata_uuid* to a given UUID, see also *-U*

  The metadata_uuid is stored only in the superblock and is a backward incompatible change. The fsid in metadata blocks remains unchanged and is not overwritten, thus the whole operation is significantly faster than *-U*.

  The new metadata_uuid can be used for mount by UUID and is also used to identify devices of a multi-device filesystem.

-n (since kernel: 3.14)

Enable no-holes feature (more efficient representation of file holes), enabled by mkfs feature *no-holes*.

-r (since kernel: 3.7)

Enable extended inode refs (hardlink limit per file in a directory is 65536), enabled by mkfs feature *extref*.

-S <0|1> Enable seeding on a given device. Value 1 will enable seeding, 0 will disable it. A seeding filesystem is forced to be mounted read-only. A new device can be added to the filesystem and will capture all writes keeping the seeding device intact. See also section *SEEDING DEVICE* in `btrfs(5)`.

> **Warning:** Clearing the seeding flag on a device may be dangerous. If a previously-seeding device is changed, all filesystems that used that device will become unmountable. Setting the seeding flag back will not fix that.
>
> A valid usecase is 'seeding device as a base image'. Clear the seeding flag, update the filesystem and make it seeding again, provided that it's OK to throw away all filesystems built on top of the previous base.

-u Change fsid to a randomly generated UUID or continue previous fsid change operation in case it was interrupted.

-U <UUID> Change fsid to 'UUID' in all metadata blocks.

The *UUID* should be a 36 bytes string in `printf(3)` format *"%08x-%04x-%04x-%04x-%012x"*. If there is a previous unfinished fsid change, it will continue only if the *UUID* matches the unfinished one or if you use the option *-u*.

All metadata blocks are rewritten, this may take some time, but the final filesystem compatibility is unaffected, unlike *-M*.

> **Warning:** Cancelling or interrupting a UUID change operation will make the filesystem temporarily unmountable. To fix it, rerun *btrfstune -u* and let it complete.

-x (since kernel: 3.10)

Enable skinny metadata extent refs (more efficient representation of extents), enabled by mkfs feature *skinny-metadata*.

All newly created extents will use the new representation. To completely switch the entire filesystem, run a full balance of the metadata. Please refer to `btrfs-balance(8)`.

2.23.4 EXIT STATUS

btrfstune returns 0 if no error happened, 1 otherwise.

2.23.5 COMPATIBILITY NOTE

This deprecated tool exists for historical reasons but is still in use today. Its functionality will be merged to the main tool, at which time **btrfstune** will be declared obsolete and scheduled for removal.

2.23.6 SEE ALSO

`btrfs(5)`, `btrfs-balance(8)`, `mkfs.btrfs(8)`

2.24 fsck.btrfs(8)

2.24.1 SYNOPSIS

fsck.btrfs [-aApy] [<device>...]

2.24.2 DESCRIPTION

fsck.btrfs is a type of utility that should exist for any filesystem and is called during system setup when the corresponding `/etc/fstab` entries contain non-zero value for *fs_passno*, see `fstab(5)` for more.

Traditional filesystems need to run their respective fsck utility in case the filesystem was not unmounted cleanly and the log needs to be replayed before mount. This is not needed for BTRFS. You should set fs_passno to 0.

If you wish to check the consistency of a BTRFS filesystem or repair a damaged filesystem, see `btrfs-check(8)`. By default filesystem consistency is checked, the repair mode is enabled via the *--repair* option (use with care!).

2.24.3 OPTIONS

The options are all the same and detect if **fsck.btrfs** is executed in non-interactive mode and exits with success, otherwise prints a message about btrfs check.

2.24.4 EXIT STATUS

There are two possible exit codes returned:

0
: No error

8
: Operational error, eg. device does not exist

2.24.5 FILES

`/etc/fstab`

2.24.6 SEE ALSO

`btrfs(8)`, `fsck(8)`, `fstab(5)`

2.25 mkfs.btrfs(8)

2.25.1 SYNOPSIS

mkfs.btrfs [options] <device> [<device>...]

2.25.2 DESCRIPTION

mkfs.btrfs is used to create the btrfs filesystem on a single or multiple devices. The *device* is typically a block device but can be a file-backed image as well. Multiple devices are grouped by UUID of the filesystem.

Before mounting such filesystem, the kernel module must know all the devices either via preceding execution of **btrfs device scan** or using the *device* mount option. See section *MULTIPLE DEVICES* for more details.

The default block group profiles for data and metadata depend on number of devices and possibly other factors. It's recommended to use specific profiles but the defaults should be OK and allowing future conversions to other profiles. Please see options *-d* and *-m* for further details and `btrfs-balance(8)` for the profile conversion post mkfs.

2.25.3 OPTIONS

-b|--byte-count <size>
Specify the size of the filesystem. If this option is not used, then mkfs.btrfs uses the entire device space for the filesystem.

--csum <type>, --checksum <type> Specify the checksum algorithm. Default is *crc32c*. Valid values are *crc32c*, *xxhash*, *sha256* or *blake2*. To mount such filesystem kernel must support the checksums as well. See *CHECKSUM ALGORITHMS* in `btrfs(5)`.

-d|--data <profile>
Specify the profile for the data block groups. Valid values are *raid0*, *raid1*, *raid1c3*, *raid1c4*, *raid5*, *raid6*, *raid10* or *single* or *dup* (case does not matter).

See *DUP PROFILES ON A SINGLE DEVICE* for more details.

On multiple devices, the default was *raid0* until version 5.7, while it is *single* since version 5.8. You can still select *raid0* manually, but it was not suitable as default.

-m|--metadata <profile>
Specify the profile for the metadata block groups. Valid values are *raid0*, *raid1*, *raid1c3*, *raid1c4*, *raid5*, *raid6*, *raid10*, *single* or *dup* (case does not matter).

Default on a single device filesystem is *DUP* and is recommended for metadata in general. The duplication might not be necessary in some use cases and it's up to the user to changed that at mkfs time or later. This depends on hardware that could potentially deduplicate the blocks again but this cannot be detected at mkfs time.

Note: Up to version 5.14 there was a detection of a SSD device (more precisely if it's a rotational device, determined by the contents of file */sys/block/DEV/queue/rotational*) that used to select *single*. This has changed in version 5.15 to be always *dup*.

Note that the rotational status can be arbitrarily set by the underlying block device driver and may not reflect the true status (network block device, memory-backed SCSI devices, real block device behind some additional device mapper layer, etc). It's recommended to always set the options *--data/--metadata* to avoid confusion and unexpected results.

See *DUP PROFILES ON A SINGLE DEVICE* for more details.

On multiple devices the default is *raid1*.

-M|--mixed
Normally the data and metadata block groups are isolated. The *mixed* mode will remove the isolation and store both types in the same block group type. This helps to utilize the free space regardless of the purpose and is suitable for small devices. The separate allocation of block groups leads to a situation where the space is reserved for the other block group type, is not available for allocation and can lead to ENOSPC state.

The recommended size for the mixed mode is for filesystems less than 1GiB. The soft recommendation is to use it for filesystems smaller than 5GiB. The mixed mode may lead to degraded performance on larger filesystems, but is otherwise usable, even on multiple devices.

The *nodesize* and *sectorsize* must be equal, and the block group types must match.

Note: Versions up to 4.2.x forced the mixed mode for devices smaller than 1GiB. This has been removed in 4.3+ as it caused some usability issues.

Mixed profile cannot be used together with other profiles. It can only be set at creation time. Conversion to or from mixed profile is not implemented.

-l|--leafsize <size>
Alias for *--nodesize*. Deprecated.

-n|--nodesize <size>
Specify the nodesize, the tree block size in which btrfs stores metadata. The default value is 16KiB (16384) or the page size, whichever is bigger. Must be a multiple of the sectorsize and a power of 2, but not larger than 64KiB (65536). Leafsize always equals nodesize and the options are aliases.

Smaller node size increases fragmentation but leads to taller b-trees which in turn leads to lower locking contention. Higher node sizes give better packing and less fragmentation at the cost of more expensive memory operations while updating the metadata blocks.

Note: Versions up to 3.11 set the nodesize to 4KiB.

-s|--sectorsize <size>
Specify the sectorsize, the minimum data block allocation unit.

The default value is the page size and is autodetected. If the sectorsize differs from the page size, the created filesystem may not be mountable by the running kernel. Therefore it is not recommended to use this option unless you are going to mount it on a system with the appropriate page size.

-L|--label <string>
Specify a label for the filesystem. The *string* should be less than 256 bytes and must not contain newline characters.

-K|--nodiscard
Do not perform whole device TRIM operation on devices that are capable of that. This does not affect discard/trim operation when the filesystem is mounted. Please see the mount option *discard* for that in `btrfs(5)`.

-r|--rootdir <rootdir>
Populate the toplevel subvolume with files from *rootdir*. This does not require root permissions to write the new files or to mount the filesystem.

Note: This option may enlarge the image or file to ensure it's big enough to contain the files from *rootdir*. Since version 4.14.1 the filesystem size is not minimized. Please see option *--shrink* if you need that functionality.

--shrink Shrink the filesystem to its minimal size, only works with *--rootdir* option.

If the destination block device is a regular file, this option will also truncate the file to the minimal size. Otherwise it will reduce the filesystem available space. Extra space will not be usable unless the filesystem is mounted and resized using **btrfs filesystem resize**.

Note: Prior to version 4.14.1, the shrinking was done automatically.

-O|--features <feature1>[,<feature2>...]
A list of filesystem features turned on at mkfs time. Not all features are supported by old kernels. To disable a feature, prefix it with ^.

See section *FILESYSTEM FEATURES* for more details. To see all available features that **mkfs.btrfs** supports run:

```
$ mkfs.btrfs -O list-all
```

-R|--runtime-features <feature1>[,<feature2>...]
A list of features that be can enabled at mkfs time, otherwise would have to be turned on on a mounted filesystem. To disable a feature, prefix it with ^.

See section *RUNTIME FEATURES* for more details. To see all available runtime features that **mkfs.btrfs** supports run:

```
$ mkfs.btrfs -R list-all
```

-f|--force
Forcibly overwrite the block devices when an existing filesystem is detected. By default, **mkfs.btrfs** will utilize *libblkid* to check for any known filesystem on the devices. Alternatively you can use the **wipefs** utility to clear the devices.

-q|--quiet
Print only error or warning messages. Options *--features* or *--help* are unaffected. Resets any previous effects of *--verbose*.

-U|--uuid <UUID>
Create the filesystem with the given *UUID*. The UUID must not exist on any filesystem currently present.

-v|--verbose
Increase verbosity level, default is 1.

-V|--version
Print the **mkfs.btrfs** version and exit.

--help Print help.

2.25.4 SIZE UNITS

The default unit is *byte*. All size parameters accept suffixes in the 1024 base. The recognized suffixes are: *k*, *m*, *g*, *t*, *p*, *e*, both uppercase and lowercase.

2.25.5 MULTIPLE DEVICES

Before mounting a multiple device filesystem, the kernel module must know the association of the block devices that are attached to the filesystem UUID.

There is typically no action needed from the user. On a system that utilizes a udev-like daemon, any new block device is automatically registered. The rules call **btrfs device scan**.

The same command can be used to trigger the device scanning if the btrfs kernel module is reloaded (naturally all previous information about the device registration is lost).

Another possibility is to use the mount options *device* to specify the list of devices to scan at the time of mount.

```
# mount -o device=/dev/sdb,device=/dev/sdc /dev/sda /mnt
```

Note: This means only scanning, if the devices do not exist in the system, mount will fail anyway. This can happen on systems without initramfs/initrd and root partition created with RAID1/10/5/6 profiles. The mount action can happen before all block devices are discovered. The waiting is usually done on the initramfs/initrd systems.

Warning: RAID5/6 has known problems and should not be used in production.

2.25.6 FILESYSTEM FEATURES

Features that can be enabled during creation time. See also `btrfs(5)` section *FILESYSTEM FEATURES*.

mixed-bg
(kernel support since 2.6.37)

mixed data and metadata block groups, also set by option *--mixed*

extref
(default since btrfs-progs 3.12, kernel support since 3.7)

increased hardlink limit per file in a directory to 65536, older kernels supported a varying number of hardlinks depending on the sum of all file name sizes that can be stored into one metadata block

raid56
(kernel support since 3.9)

extended format for RAID5/6, also enabled if raid5 or raid6 block groups are selected

skinny-metadata
(default since btrfs-progs 3.18, kernel support since 3.10)

reduced-size metadata for extent references, saves a few percent of metadata

no-holes
(default since btrfs-progs 5.15, kernel support since 3.14)

improved representation of file extents where holes are not explicitly stored as an extent, saves a few percent of metadata if sparse files are used

zoned
(kernel support since 5.12)

zoned mode, data allocation and write friendly to zoned/SMR/ZBC/ZNS devices, see *ZONED MODE* in `btrfs(5)`, the mode is automatically selected when a zoned device is detected

2.25.7 RUNTIME FEATURES

Features that are typically enabled on a mounted filesystem, eg. by a mount option or by an ioctl. Some of them can be enabled early, at mkfs time. This applies to features that need to be enabled once and then the status is permanent, this does not replace mount options.

quota
(kernel support since 3.4)

Enable quota support (qgroups). The qgroup accounting will be consistent, can be used together with *--rootdir*. See also `btrfs-quota(8)`.

free-space-tree
(default since btrfs-progs 5.15, kernel support since 4.5)

Enable the free space tree (mount option *space_cache=v2*) for persisting the free space cache.

2.25.8 BLOCK GROUPS, CHUNKS, RAID

The highlevel organizational units of a filesystem are block groups of three types: data, metadata and system.

DATA
store data blocks and nothing else

METADATA
store internal metadata in b-trees, can store file data if they fit into the inline limit

SYSTEM
store structures that describe the mapping between the physical devices and the linear logical space representing the filesystem

Other terms commonly used:

block group, chunk
a logical range of space of a given profile, stores data, metadata or both; sometimes the terms are used interchangeably

A typical size of metadata block group is 256MiB (filesystem smaller than 50GiB) and 1GiB (larger than 50GiB), for data it's 1GiB. The system block group size is a few megabytes.

RAID
a block group profile type that utilizes RAID-like features on multiple devices: striping, mirroring, parity

profile
when used in connection with block groups refers to the allocation strategy and constraints, see the section *PROFILES* for more details

2.25.9 PROFILES

There are the following block group types available:

Profiles	Redundancy Copies	Redundancy Parity	Redundancy Striping	Space utilization	Min/max devices
single	1			100%	1/any
DUP	2 / 1 device			50%	1/any (see note 1)
RAID0	1		1 to N	100%	1/any (see note 5)
RAID1	2			50%	2/any
RAID1C3	3			33%	3/any
RAID1C4	4			25%	4/any
RAID10	2		1 to N	50%	2/any (see note 5)
RAID5	1	1	2 to N-1	(N-1)/N	2/any (see note 2)
RAID6	1	2	3 to N-2	(N-2)/N	3/any (see note 3)

Warning: It's not recommended to create filesystems with RAID0/1/10/5/6 profiles on partitions from the same device. Neither redundancy nor performance will be improved.

Note 1: DUP may exist on more than 1 device if it starts on a single device and another one is added. Since version 4.5.1, **mkfs.btrfs** will let you create DUP on multiple devices without restrictions.

Note 2: It's not recommended to use 2 devices with RAID5. In that case, parity stripe will contain the same data as the data stripe, making RAID5 degraded to RAID1 with more overhead.

Note 3: It's also not recommended to use 3 devices with RAID6, unless you want to get effectively 3 copies in a RAID1-like manner (but not exactly that).

Note 4: Since kernel 5.5 it's possible to use RAID1C3 as replacement for RAID6, higher space cost but reliable.

Note 5: Since kernel 5.15 it's possible to use (mount, convert profiles) RAID0 on one device and RAID10 on two devices.

PROFILE LAYOUT

For the following examples, assume devices numbered by 1, 2, 3 and 4, data or metadata blocks A, B, C, D, with possible stripes eg. A1, A2 that would be logically A, etc. For parity profiles PA and QA are parity and syndrom, associated with the given stripe. The simple layouts single or DUP are left out. Actual physical block placement on devices depends on current state of the free/allocated space and may appear random. All devices are assumed to be present at the time of the blocks would have been written.

RAID1

device 1	device 2	device 3	device 4
A	D		
B			C
C			
D	A	B	

RAID1C3

device 1	device 2	device 3	device 4
A	A	D	
B		B	
C		A	C
D	D	C	B

RAID0

device 1	device 2	device 3	device 4
A2	C3	A3	C2
B1	A1	D2	B3
C1	D3	B4	D1
D4	B2	C4	A4

RAID5

device 1	device 2	device 3	device 4
A2	C3	A3	C2
B1	A1	D2	B3
C1	D3	PB	D1
PD	B2	PC	PA

RAID6

device 1	device 2	device 3	device 4
A2	QC	QA	C2
B1	A1	D2	QB
C1	QD	PB	D1
PD	B2	PC	PA

2.25.10 DUP PROFILES ON A SINGLE DEVICE

The mkfs utility will let the user create a filesystem with profiles that write the logical blocks to 2 physical locations. Whether there are really 2 physical copies highly depends on the underlying device type.

For example, a SSD drive can remap the blocks internally to a single copy--thus deduplicating them. This negates the purpose of increased redundancy and just wastes filesystem space without providing the expected level of redundancy.

The duplicated data/metadata may still be useful to statistically improve the chances on a device that might perform some internal optimizations. The actual details are not usually disclosed by vendors. For example we could expect that not all blocks get deduplicated. This will provide a non-zero probability of recovery compared to a zero chance if the single profile is used. The user should make the tradeoff decision. The deduplication in SSDs is thought to be widely available so the reason behind the mkfs default is to not give a false sense of redundancy.

As another example, the widely used USB flash or SD cards use a translation layer between the logical and physical view of the device. The data lifetime may be affected by frequent plugging. The memory cells could get damaged, hopefully not destroying both copies of particular data in case of DUP.

The wear levelling techniques can also lead to reduced redundancy, even if the device does not do any deduplication. The controllers may put data written in a short timespan into the same physical storage unit (cell, block etc). In case this unit dies, both copies are lost. BTRFS does not add any artificial delay between metadata writes.

The traditional rotational hard drives usually fail at the sector level.

In any case, a device that starts to misbehave and repairs from the DUP copy should be replaced! **DUP is not backup**.

2.25.11 KNOWN ISSUES

SMALL FILESYSTEMS AND LARGE NODESIZE

The combination of small filesystem size and large nodesize is not recommended in general and can lead to various ENOSPC-related issues during mount time or runtime.

Since mixed block group creation is optional, we allow small filesystem instances with differing values for *sectorsize* and *nodesize* to be created and could end up in the following situation:

```
# mkfs.btrfs -f -n 65536 /dev/loop0
btrfs-progs v3.19-rc2-405-g976307c
See http://btrfs.wiki.kernel.org for more information.

Performing full device TRIM (512.00MiB) ...
Label:              (null)
UUID:               49fab72e-0c8b-466b-a3ca-d1bfe56475f0
Node size:          65536
Sector size:        4096
Filesystem size:    512.00MiB
Block group profiles:
  Data:             single            8.00MiB
  Metadata:         DUP              40.00MiB
  System:           DUP              12.00MiB
SSD detected:       no
Incompat features:  extref, skinny-metadata
Number of devices:  1
Devices:
  ID        SIZE  PATH
   1   512.00MiB  /dev/loop0
```

(continues on next page)

(continued from previous page)

```
# mount /dev/loop0 /mnt/
mount: mount /dev/loop0 on /mnt failed: No space left on device
```

The ENOSPC occurs during the creation of the UUID tree. This is caused by large metadata blocks and space reservation strategy that allocates more than can fit into the filesystem.

2.25.12 AVAILABILITY

mkfs.btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

2.25.13 SEE ALSO

`btrfs(5)`, `btrfs(8)`, `btrfs-balance(8)`, `wipefs(8)`

CHAPTER

THREE

ADMINISTRATION

The main administration tool for BTRFS filesystems is `btrfs`. Please refer to the manual pages of the subcommands for further documentation.

3.1 Mount options

This section describes mount options specific to BTRFS. For the generic mount options please refer to `mount(8)` manpage. The options are sorted alphabetically (discarding the *no* prefix).

Note: Most mount options apply to the whole filesystem and only options in the first mounted subvolume will take effect. This is due to lack of implementation and may change in the future. This means that (for example) you can't set per-subvolume *nodatacow*, *nodatasum*, or *compress* using mount options. This should eventually be fixed, but it has proved to be difficult to implement correctly within the Linux VFS framework.

Mount options are processed in order, only the last occurrence of an option takes effect and may disable other options due to constraints (see eg. *nodatacow* and *compress*). The output of **mount** command shows which options have been applied.

acl, noacl
(default: on)

Enable/disable support for Posix Access Control Lists (ACLs). See the `acl(5)` manual page for more information about ACLs.

The support for ACL is build-time configurable (BTRFS_FS_POSIX_ACL) and mount fails if *acl* is requested but the feature is not compiled in.

autodefrag, noautodefrag
(since: 3.0, default: off)

Enable automatic file defragmentation. When enabled, small random writes into files (in a range of tens of kilobytes, currently it's 64KiB) are detected and queued up for the defragmentation process. Not well suited for large database workloads.

The read latency may increase due to reading the adjacent blocks that make up the range for defragmentation, successive write will merge the blocks in the new location.

Warning: Defragmenting with Linux kernel versions < 3.9 or 3.14-rc2 as well as with Linux stable kernel versions 3.10.31, 3.12.12 or 3.13.4 will break up the reflinks of COW data (for example files copied with **cp --reflink**, snapshots or de-duplicated data). This may cause considerable increase of space usage depending on the broken up reflinks.

barrier, nobarrier
(default: on)

Ensure that all IO write operations make it through the device cache and are stored permanently when the filesystem is at its consistency checkpoint. This typically means that a flush command is sent to the device that will synchronize all pending data and ordinary metadata blocks, then writes the superblock and issues another flush.

The write flushes incur a slight hit and also prevent the IO block scheduler to reorder requests in a more effective way. Disabling barriers gets rid of that penalty but will most certainly lead to a corrupted filesystem in case of a crash or power loss. The ordinary metadata blocks could be yet unwritten at the time the new superblock is stored permanently, expecting that the block pointers to metadata were stored permanently before.

On a device with a volatile battery-backed write-back cache, the *nobarrier* option will not lead to filesystem corruption as the pending blocks are supposed to make it to the permanent storage.

check_int, check_int_data, check_int_print_mask=<value>
(since: 3.0, default: off)

These debugging options control the behavior of the integrity checking module (the BTRFS_FS_CHECK_INTEGRITY config option required). The main goal is to verify that all blocks from a given transaction period are properly linked.

check_int enables the integrity checker module, which examines all block write requests to ensure on-disk consistency, at a large memory and CPU cost.

check_int_data includes extent data in the integrity checks, and implies the *check_int* option.

check_int_print_mask takes a bitmask of BTRFSIC_PRINT_MASK_* values as defined in *fs/btrfs/check-integrity.c*, to control the integrity checker module behavior.

See comments at the top of *fs/btrfs/check-integrity.c* for more information.

clear_cache
Force clearing and rebuilding of the disk space cache if something has gone wrong. See also: *space_cache*.

commit=<seconds>
(since: 3.12, default: 30)

Set the interval of periodic transaction commit when data are synchronized to permanent storage. Higher interval values lead to larger amount of unwritten data, which has obvious consequences when the system crashes. The upper bound is not forced, but a warning is printed if it's more than 300 seconds (5 minutes). Use with care.

compress, compress=<type[:level]>, compress-force, compress-force=<type[:level]>
(default: off, level support since: 5.1)

Control BTRFS file data compression. Type may be specified as *zlib*, *lzo*, *zstd* or *no* (for no compression, used for remounting). If no type is specified, *zlib* is used. If *compress-force* is specified, then compression will always be attempted, but the data may end up uncompressed if the compression would make them larger.

Both *zlib* and *zstd* (since version 5.1) expose the compression level as a tunable knob with higher levels trading speed and memory (*zstd*) for higher compression ratios. This can be set by appending a colon and the desired level. Zlib accepts the range [1, 9] and zstd accepts [1, 15]. If no level is set, both currently use a default level of 3. The value 0 is an alias for the default level.

Otherwise some simple heuristics are applied to detect an incompressible file. If the first blocks written to a file are not compressible, the whole file is permanently marked to skip compression. As this is too simple, the *compress-force* is a workaround that will compress most of the files at the cost of some wasted CPU cycles on failed attempts. Since kernel 4.15, a set of heuristic algorithms have been improved by using frequency sampling, repeated pattern detection and Shannon entropy calculation to avoid that.

Note: If compression is enabled, *nodatacow* and *nodatasum* are disabled.

datacow, nodatacow
(default: on)

Enable data copy-on-write for newly created files. *Nodatacow* implies *nodatasum*, and disables *compression*. All files created under *nodatacow* are also set the NOCOW file attribute (see `chattr(1)`).

Note: If *nodatacow* or *nodatasum* are enabled, compression is disabled.

Updates in-place improve performance for workloads that do frequent overwrites, at the cost of potential partial writes, in case the write is interrupted (system crash, device failure).

datasum, nodatasum
(default: on)

Enable data checksumming for newly created files. *Datasum* implies *datacow*, ie. the normal mode of operation. All files created under *nodatasum* inherit the "no checksums" property, however there's no corresponding file attribute (see `chattr(1)`).

Note: If *nodatacow* or *nodatasum* are enabled, compression is disabled.

There is a slight performance gain when checksums are turned off, the corresponding metadata blocks holding the checksums do not need to updated. The cost of checksumming of the blocks in memory is much lower than the IO, modern CPUs feature hardware support of the checksumming algorithm.

degraded
(default: off)

Allow mounts with less devices than the RAID profile constraints require. A read-write mount (or remount) may fail when there are too many devices missing, for example if a stripe member is completely missing from RAID0.

Since 4.14, the constraint checks have been improved and are verified on the chunk level, not an the device level. This allows degraded mounts of filesystems with mixed RAID profiles for data and metadata, even if the device number constraints would not be satisfied for some of the profiles.

Example: metadata -- raid1, data -- single, devices -- /dev/sda, /dev/sdb

Suppose the data are completely stored on *sda*, then missing *sdb* will not prevent the mount, even if 1 missing device would normally prevent (any) *single* profile to mount. In case some of the data chunks are stored on *sdb*, then the constraint of single/data is not satisfied and the filesystem cannot be mounted.

device=<devicepath>
Specify a path to a device that will be scanned for BTRFS filesystem during mount. This is usually done automatically by a device manager (like udev) or using the **btrfs device scan** command (eg. run from the initial ramdisk). In cases where this is not possible the *device* mount option can help.

Note: Booting eg. a RAID1 system may fail even if all filesystem's *device* paths are provided as the actual device nodes may not be discovered by the system at that point.

discard, discard=sync, discard=async, nodiscard
(default: off, async support since: 5.6)

Enable discarding of freed file blocks. This is useful for SSD devices, thinly provisioned LUNs, or virtual machine images; however, every storage layer must support discard for it to work.

In the synchronous mode (*sync* or without option value), lack of asynchronous queued TRIM on the backing device TRIM can severely degrade performance, because a synchronous TRIM operation will be attempted instead. Queued TRIM requires newer than SATA revision 3.1 chipsets and devices.

The asynchronous mode (*async*) gathers extents in larger chunks before sending them to the devices for TRIM. The overhead and performance impact should be negligible compared to the previous mode and it's supposed to be the preferred mode if needed.

If it is not necessary to immediately discard freed blocks, then the `fstrim` tool can be used to discard all free blocks in a batch. Scheduling a TRIM during a period of low system activity will prevent latent interference with the performance of other operations. Also, a device may ignore the TRIM command if the range is too small, so running a batch discard has a greater probability of actually discarding the blocks.

enospc_debug, noenospc_debug
(default: off)

Enable verbose output for some ENOSPC conditions. It's safe to use but can be noisy if the system reaches near-full state.

fatal_errors=<action>
(since: 3.4, default: bug)

Action to take when encountering a fatal error.

bug
BUG() on a fatal error, the system will stay in the crashed state and may be still partially usable, but reboot is required for full operation

panic
panic() on a fatal error, depending on other system configuration, this may be followed by a reboot. Please refer to the documentation of kernel boot parameters, eg. *panic*, *oops* or *crashkernel*.

flushoncommit, noflushoncommit
(default: off)

This option forces any data dirtied by a write in a prior transaction to commit as part of the current commit, effectively a full filesystem sync.

This makes the committed state a fully consistent view of the file system from the application's perspective (i.e. it includes all completed file system operations). This was previously the behavior only when a snapshot was created.

When off, the filesystem is consistent but buffered writes may last more than one transaction commit.

fragment=<type>
(depends on compile-time option BTRFS_DEBUG, since: 4.4, default: off)

A debugging helper to intentionally fragment given *type* of block groups. The type can be *data*, *metadata* or *all*. This mount option should not be used outside of debugging environments and is not recognized if the kernel config option *BTRFS_DEBUG* is not enabled.

nologreplay
(default: off, even read-only)

The tree-log contains pending updates to the filesystem until the full commit. The log is replayed on next mount, this can be disabled by this option. See also *treelog*. Note that *nologreplay* is the same as *norecovery*.

Warning: Currently, the tree log is replayed even with a read-only mount! To disable that behaviour, mount also with *nologreplay*.

max_inline=<bytes>
(default: min(2048, page size))

Specify the maximum amount of space, that can be inlined in a metadata b-tree leaf. The value is specified in bytes, optionally with a K suffix (case insensitive). In practice, this value is limited by the filesystem block size (named *sectorsize* at mkfs time), and memory page size of the system. In case of sectorsize limit, there's some space unavailable due to leaf headers. For example, a 4KiB sectorsize, maximum size of inline data is about 3900 bytes.

Inlining can be completely turned off by specifying 0. This will increase data block slack if file sizes are much smaller than block size but will reduce metadata consumption in return.

Note: The default value has changed to 2048 in kernel 4.6.

---

metadata_ratio=<value>
(default: 0, internal logic)

Specifies that 1 metadata chunk should be allocated after every *value* data chunks. Default behaviour depends on internal logic, some percent of unused metadata space is attempted to be maintained but is not always possible if there's not enough space left for chunk allocation. The option could be useful to override the internal logic in favor of the metadata allocation if the expected workload is supposed to be metadata intense (snapshots, reflinks, xattrs, inlined files).

norecovery
(since: 4.5, default: off)

Do not attempt any data recovery at mount time. This will disable *logreplay* and avoids other write operations. Note that this option is the same as *nologreplay*.

Note: The opposite option *recovery* used to have different meaning but was changed for consistency with other filesystems, where *norecovery* is used for skipping log replay. BTRFS does the same and in general will try to avoid any write operations.

---

rescan_uuid_tree
(since: 3.12, default: off)

Force check and rebuild procedure of the UUID tree. This should not normally be needed.

rescue
(since: 5.9)

Modes allowing mount with damaged filesystem structures.

- *usebackuproot* (since: 5.9, replaces standalone option *usebackuproot*)
- *nologreplay* (since: 5.9, replaces standalone option *nologreplay*)
- *ignorebadroots*, *ibadroots* (since: 5.11)
- *ignoredatacsums*, *idatacsums* (since: 5.11)
- *all* (since: 5.9)

skip_balance
(since: 3.3, default: off)

Skip automatic resume of an interrupted balance operation. The operation can later be resumed with **btrfs balance resume**, or the paused state can be removed with **btrfs balance cancel**. The default behaviour is to resume an interrupted balance immediately after a volume is mounted.

space_cache, space_cache=<version>, nospace_cache
(*nospace_cache* since: 3.2, *space_cache=v1* and *space_cache=v2* since 4.5, default: *space_cache=v1*)

Options to control the free space cache. The free space cache greatly improves performance when reading block group free space into memory. However, managing the space cache consumes some resources, including a small amount of disk space.

There are two implementations of the free space cache. The original one, referred to as *v1*, is the safe default. The *v1* space cache can be disabled at mount time with *nospace_cache* without clearing.

On very large filesystems (many terabytes) and certain workloads, the performance of the *v1* space cache may degrade drastically. The *v2* implementation, which adds a new b-tree called the free space tree, addresses this issue. Once enabled, the *v2* space cache will always be used and cannot be disabled unless it is cleared. Use *clear_cache,space_cache=v1* or *clear_cache,nospace_cache* to do so. If *v2* is enabled, kernels without *v2* support will only be able to mount the filesystem in read-only mode.

The `btrfs-check(8)` and `` `mkfs.btrfs(8)`` commands have full *v2* free space cache support since v4.19.

If a version is not explicitly specified, the default implementation will be chosen, which is *v1*.

ssd, ssd_spread, nossd, nossd_spread
(default: SSD autodetected)

Options to control SSD allocation schemes. By default, BTRFS will enable or disable SSD optimizations depending on status of a device with respect to rotational or non-rotational type. This is determined by the contents of */sys/block/DEV/queue/rotational*). If it is 0, the *ssd* option is turned on. The option *nossd* will disable the autodetection.

The optimizations make use of the absence of the seek penalty that's inherent for the rotational devices. The blocks can be typically written faster and are not offloaded to separate threads.

Note: Since 4.14, the block layout optimizations have been dropped. This used to help with first generations of SSD devices. Their FTL (flash translation layer) was not effective and the optimization was supposed to improve the wear by better aligning blocks. This is no longer true with modern SSD devices and the optimization had no real benefit. Furthermore it caused increased fragmentation. The layout tuning has been kept intact for the option *ssd_spread*.

The *ssd_spread* mount option attempts to allocate into bigger and aligned chunks of unused space, and may perform better on low-end SSDs. *ssd_spread* implies *ssd*, enabling all other SSD heuristics as well. The option *nossd* will disable all SSD options while *nossd_spread* only disables *ssd_spread*.

subvol=<path>
Mount subvolume from *path* rather than the toplevel subvolume. The *path* is always treated as relative to the toplevel subvolume. This mount option overrides the default subvolume set for the given filesystem.

subvolid=<subvolid>
Mount subvolume specified by a *subvolid* number rather than the toplevel subvolume. You can use **btrfs subvolume list** of **btrfs subvolume show** to see subvolume ID numbers. This mount option overrides the default subvolume set for the given filesystem.

Note: If both *subvolid* and *subvol* are specified, they must point at the same subvolume, otherwise the mount will fail.

thread_pool=<number>
(default: min(NRCPUS + 2, 8))

The number of worker threads to start. NRCPUS is number of on-line CPUs detected at the time of mount. Small number leads to less parallelism in processing data and metadata, higher numbers could lead to a performance hit due to increased locking contention, process scheduling, cache-line bouncing or costly data transfers between local CPU memories.

treelog, notreelog
(default: on)

Enable the tree logging used for *fsync* and *O_SYNC* writes. The tree log stores changes without the need of a full filesystem sync. The log operations are flushed at sync and transaction commit. If the system crashes between two such syncs, the pending tree log operations are replayed during mount.

Warning: Currently, the tree log is replayed even with a read-only mount! To disable that behaviour, also mount with *nologreplay*.

The tree log could contain new files/directories, these would not exist on a mounted filesystem if the log is not replayed.

usebackuproot
(since: 4.6, default: off)

Enable autorecovery attempts if a bad tree root is found at mount time. Currently this scans a backup list of several previous tree roots and tries to use the first readable. This can be used with read-only mounts as well.

Note: This option has replaced *recovery*.

user_subvol_rm_allowed
(default: off)

Allow subvolumes to be deleted by their respective owner. Otherwise, only the root user can do that.

Note: Historically, any user could create a snapshot even if he was not owner of the source subvolume, the subvolume deletion has been restricted for that reason. The subvolume creation has been restricted but this mount option is still required. This is a usability issue. Since 4.18, the `rmdir(2)` syscall can delete an empty subvolume just like an ordinary directory. Whether this is possible can be detected at runtime, see *rmdir_subvol* feature in *FILESYSTEM FEATURES*.

3.1.1 DEPRECATED MOUNT OPTIONS

List of mount options that have been removed, kept for backward compatibility.

recovery
(since: 3.2, default: off, deprecated since: 4.5)

Note: This option has been replaced by *usebackuproot* and should not be used but will work on 4.5+ kernels.

---

inode_cache, noinode_cache
(removed in: 5.11, since: 3.0, default: off)

Note: The functionality has been removed in 5.11, any stale data created by previous use of the *inode_cache* option can be removed by **btrfs check --clear-ino-cache**.

3.1.2 NOTES ON GENERIC MOUNT OPTIONS

Some of the general mount options from `mount(8)` that affect BTRFS and are worth mentioning.

noatime
under read intensive work-loads, specifying *noatime* significantly improves performance because no new access time information needs to be written. Without this option, the default is *relatime*, which only reduces the number of inode atime updates in comparison to the traditional *strictatime*. The worst case for atime updates under 'relatime' occurs when many files are read whose atime is older than 24 h and which are freshly snapshotted. In that case the atime is updated and COW happens - for each file - in bulk. See also https://lwn.net/Articles/499293/ - *Atime and btrfs: a bad combination? (LWN, 2012-05-31)*.

Note that *noatime* may break applications that rely on atime uptimes like the venerable Mutt (unless you use maildir mailboxes).

3.2 Bootloaders

GRUB2 (https://www.gnu.org/software/grub) has the most advanced support of booting from BTRFS with respect to features.

U-boot (https://www.denx.de/wiki/U-Boot/) has decent support for booting but not all BTRFS features are implemented, check the documentation.

EXTLINUX (from the https://syslinux.org project) can boot but does not support all features. Please check the upstream documentation before you use it.

The first 1MiB on each device is unused with the exception of primary superblock that is on the offset 64KiB and spans 4KiB.

3.3 Filesystem limits

maximum file name length
255

maximum symlink target length
depends on the *nodesize* value, for 4KiB it's 3949 bytes, for larger nodesize it's 4095 due to the system limit PATH_MAX

The symlink target may not be a valid path, ie. the path name components can exceed the limits (NAME_MAX), there's no content validation at `symlink(3)` creation.

maximum number of inodes
2^{64} but depends on the available metadata space as the inodes are created dynamically

inode numbers
minimum number: 256 (for subvolumes), regular files and directories: 257

maximum file length
inherent limit of btrfs is 2^{64} (16 EiB) but the linux VFS limit is 2^{63} (8 EiB)

maximum number of subvolumes
the subvolume ids can go up to 2^{64} but the number of actual subvolumes depends on the available metadata space, the space consumed by all subvolume metadata includes bookkeeping of shared extents can be large (MiB, GiB)

maximum number of hardlinks of a file in a directory
65536 when the *extref* feature is turned on during mkfs (default), roughly 100 otherwise

minimum filesystem size
the minimal size of each device depends on the *mixed-bg* feature, without that (the default) it's about 109MiB, with mixed-bg it's is 16MiB

3.4 Flexibility

The underlying design of BTRFS data structures allows a lot of flexibility and making changes after filesystem creation, like resizing, adding/removing space or enabling some features on-the-fly.

- **dynamic inode creation** -- there's no fixed space or tables for tracking inodes so the number of inodes that can be created is bounded by the metadata space and it's utilization
- **block group profile change on-the-fly** -- the block group profiles can be changed on a mounted filesystem by running the balance operation and specifying the conversion filters
- **resize** -- the space occupied by the filesystem on each device can be resized up (grow) or down (shrink) as long as the amount of data can be still contained on the device

CHAPTER

FOUR

HARDWARE CONSIDERATIONS

4.1 Storage model

A storage model is a model that captures key physical aspects of data structure in a data store. A filesystem is the logical structure organizing data on top of the storage device.

The filesystem assumes several features or limitations of the storage device and utilizes them or applies measures to guarantee reliability. BTRFS in particular is based on a COW (copy on write) mode of writing, ie. not updating data in place but rather writing a new copy to a different location and then atomically switching the pointers.

In an ideal world, the device does what it promises. The filesystem assumes that this may not be true so additional mechanisms are applied to either detect misbehaving hardware or get valid data by other means. The devices may (and do) apply their own detection and repair mechanisms but we won't assume any.

The following assumptions about storage devices are considered (sorted by importance, numbers are for further reference):

1. atomicity of reads and writes of blocks/sectors (the smallest unit of data the device presents to the upper layers)
2. there's a flush command that instructs the device to forcibly order writes before and after the command; alternatively there's a barrier command that facilitates the ordering but may not flush the data
3. data sent to write to a given device offset will be written without further changes to the data and to the offset
4. writes can be reordered by the device, unless explicitly serialized by the flush command
5. reads and writes can be freely reordered and interleaved

The consistency model of BTRFS builds on these assumptions. The logical data updates are grouped, into a generation, written on the device, serialized by the flush command and then the super block is written ending the generation. All logical links among metadata comprising a consistent view of the data may not cross the generation boundary.

4.2 When things go wrong

No or partial atomicity of block reads/writes (1)

- *Problem*: a partial block contents is written (*torn write*), eg. due to a power glitch or other electronics failure during the read/write
- *Detection*: checksum mismatch on read
- *Repair*: use another copy or rebuild from multiple blocks using some encoding scheme

The flush command does not flush (2)

This is perhaps the most serious problem and impossible to mitigate by filesystem without limitations and design restrictions. What could happen in the worst case is that writes from one generation bleed to another one, while still letting the filesystem consider the generations isolated. Crash at any point would leave data on the device in an inconsistent state without any hint what exactly got written, what is missing and leading to stale metadata link information.

Devices usually honor the flush command, but for performance reasons may do internal caching, where the flushed data are not yet persistently stored. A power failure could lead to a similar scenario as above, although it's less likely that later writes would be written before the cached ones. This is beyond what a filesystem can take into account. Devices or controllers are usually equipped with batteries or capacitors to write the cache contents even after power is cut. (*Battery backed write cache*)

Data get silently changed on write (3)

Such thing should not happen frequently, but still can happen spuriously due the complex internal workings of devices or physical effects of the storage media itself.

- *Problem*: while the data are written atomically, the contents get changed
- *Detection*: checksum mismatch on read
- 'Repair*: use another copy or rebuild from multiple blocks using some encoding scheme

Data get silently written to another offset (3)

This would be another serious problem as the filesystem has no information when it happens. For that reason the measures have to be done ahead of time. This problem is also commonly called 'ghost write'.

The metadata blocks have the checksum embedded in the blocks, so a correct atomic write would not corrupt the checksum. It's likely that after reading such block the data inside would not be consistent with the rest. To rule that out there's embedded block number in the metadata block. It's the logical block number because this is what the logical structure expects and verifies.

The following is based on information publicly available, user feedback, community discussions or bug report analyses. It's not complete and further research is encouraged when in doubt.

4.3 Main memory

The data structures and raw data blocks are temporarily stored in computer memory before they get written to the device. It is critical that memory is reliable because even simple bit flips can have vast consequences and lead to damaged structures, not only in the filesystem but in the whole operating system.

Based on experience in the community, memory bit flips are more common than one would think. When it happens, it's reported by the tree-checker or by a checksum mismatch after reading blocks. There are some very obvious instances of bit flips that happen, e.g. in an ordered sequence of keys in metadata blocks. We can easily infer from the other data what values get damaged and how. However, fixing that is not straightforward and would require cross-referencing data from the entire filesystem to see the scope.

If available, ECC memory should lower the chances of bit flips, but this type of memory is not available in all cases. A memory test should be performed in case there's a visible bit flip pattern, though this may not detect a faulty memory module because the actual load of the system could be the factor making the problems appear. In recent years attacks on how the memory modules operate have been demonstrated ('rowhammer') achieving specific bits to be flipped. While these were targeted, this shows that a series of reads or writes can affect unrelated parts of memory.

Further reading:

- https://en.wikipedia.org/wiki/Row_hammer

What to do:

- run *memtest*, note that sometimes memory errors happen only when the system is under heavy load that the default memtest cannot trigger
- memory errors may appear as filesystem going read-only due to "pre write" check, that verify meta data before they get written but fail some basic consistency checks

4.4 Direct memory access (DMA)

Another class of errors is related to DMA (direct memory access) performed by device drivers. While this could be considered a software error, the data transfers that happen without CPU assistance may accidentally corrupt other pages. Storage devices utilize DMA for performance reasons, the filesystem structures and data pages are passed back and forth, making errors possible in case page life time is not properly tracked.

There are lots of quirks (device-specific workarounds) in Linux kernel drivers (regarding not only DMA) that are added when found. The quirks may avoid specific errors or disable some features to avoid worse problems.

What to do:

- use up-to-date kernel (recent releases or maintained long term support versions)
- as this may be caused by faulty drivers, keep the systems up-to-date

4.5 Rotational disks (HDD)

Rotational HDDs typically fail at the level of individual sectors or small clusters. Read failures are caught on the levels below the filesystem and are returned to the user as *EIO - Input/output error*. Reading the blocks repeatedly may return the data eventually, but this is better done by specialized tools and filesystem takes the result of the lower layers. Rewriting the sectors may trigger internal remapping but this inevitably leads to data loss.

Disk firmware is technically software but from the filesystem perspective is part of the hardware. IO requests are processed, and caching or various other optimizations are performed, which may lead to bugs under high load or unexpected physical conditions or unsupported use cases.

Disks are connected by cables with two ends, both of which can cause problems when not attached properly. Data transfers are protected by checksums and the lower layers try hard to transfer the data correctly or not at all. The errors from badly-connecting cables may manifest as large amount of failed read or write requests, or as short error bursts depending on physical conditions.

What to do:

- check **smartctl** for potential issues

4.6 Solid state drives (SSD)

The mechanism of information storage is different from HDDs and this affects the failure mode as well. The data are stored in cells grouped in large blocks with limited number of resets and other write constraints. The firmware tries to avoid unnecessary resets and performs optimizations to maximize the storage media lifetime. The known techniques are deduplication (blocks with same fingerprint/hash are mapped to same physical block), compression or internal remapping and garbage collection of used memory cells. Due to the additional processing there are measures to verity the data e.g. by ECC codes.

The observations of failing SSDs show that the whole electronic fails at once or affects a lot of data (eg. stored on one chip). Recovering such data may need specialized equipment and reading data repeatedly does not help as it's possible with HDDs.

There are several technologies of the memory cells with different characteristics and price. The lifetime is directly affected by the type and frequency of data written. Writing "too much" distinct data (e.g. encrypted) may render the internal deduplication ineffective and lead to a lot of rewrites and increased wear of the memory cells.

There are several technologies and manufacturers so it's hard to describe them but there are some that exhibit similar behaviour:

- expensive SSD will use more durable memory cells and is optimized for reliability and high load
- cheap SSD is projected for a lower load ("desktop user") and is optimized for cost, it may employ the optimizations and/or extended error reporting partially or not at all

It's not possible to reliably determine the expected lifetime of an SSD due to lack of information about how it works or due to lack of reliable stats provided by the device.

Metadata writes tend to be the biggest component of lifetime writes to a SSD, so there is some value in reducing them. Depending on the device class (high end/low end) the features like DUP block group profiles may affect the reliability in both ways:

- *high end* are typically more reliable and using 'single' for data and metadata could be suitable to reduce device wear
- *low end* could lack ability to identify errors so an additional redundancy at the filesystem level (checksums, *DUP*) could help

Only users who consume 50 to 100% of the SSD's actual lifetime writes need to be concerned by the write amplification of btrfs DUP metadata. Most users will be far below 50% of the actual lifetime, or will write the drive to death and discover how many writes 100% of the actual lifetime was. SSD firmware often adds its own write multipliers that can be arbitrary and unpredictable and dependent on application behavior, and these will typically have far greater effect on SSD lifespan than DUP metadata. It's more or less impossible to predict when a SSD will run out of lifetime writes to within a factor of two, so it's hard to justify wear reduction as a benefit.

Further reading:

- https://www.snia.org/educational-library/ssd-and-deduplication-end-spinning-disk-2012
- https://www.snia.org/educational-library/realities-solid-state-storage-2013-2013
- https://www.snia.org/educational-library/ssd-performance-primer-2013
- https://www.snia.org/educational-library/how-controllers-maximize-ssd-life-2013

What to do:

- run **smartctl** or self-tests to look for potential issues
- keep the firmware up-to-date

4.7 NVM express, non-volatile memory (NVMe)

NVMe is a type of persistent memory usually connected over a system bus (PCIe) or similar interface and the speeds are an order of magnitude faster than SSD. It is also a non-rotating type of storage, and is not typically connected by a cable. It's not a SCSI type device either but rather a complete specification for logical device interface.

In a way the errors could be compared to a combination of SSD class and regular memory. Errors may exhibit as random bit flips or IO failures. There are tools to access the internal log (**nvme log** and **nvme-cli**) for a more detailed analysis.

There are separate error detection and correction steps performed e.g. on the bus level and in most cases never making in to the filesystem level. Once this happens it could mean there's some systematic error like overheating or bad physical connection of the device. You may want to run self-tests (using **smartctl**).

- https://en.wikipedia.org/wiki/NVM_Express
- https://www.smartmontools.org/wiki/NVMe_Support

4.8 Drive firmware

Firmware is technically still software but embedded into the hardware. As all software has bugs, so does firmware. Storage devices can update the firmware and fix known bugs. In some cases the it's possible to avoid certain bugs by quirks (device-specific workarounds) in Linux kernel.

A faulty firmware can cause wide range of corruptions from small and localized to large affecting lots of data. Self-repair capabilities may not be sufficient.

What to do:

- check for firmware updates in case there are known problems, note that updating firmware can be risky on itself
- use up-to-date kernel (recent releases or maintained long term support versions)

4.9 SD flash cards

There are a lot of devices with low power consumption and thus using storage media based on low power consumption too, typically flash memory stored on a chip enclosed in a detachable card package. An improperly inserted card may be damaged by electrical spikes when the device is turned on or off. The chips storing data in turn may be damaged permanently. All types of flash memory have a limited number of rewrites, so the data are internally translated by FTL (flash translation layer). This is implemented in firmware (technically a software) and prone to bugs that manifest as hardware errors.

Adding redundancy like using DUP profiles for both data and metadata can help in some cases but a full backup might be the best option once problems appear and replacing the card could be required as well.

4.10 Hardware as the main source of filesystem corruptions

If you use unreliable hardware and don't know about that, don't blame the filesystem when it tells you.

CHAPTER

FIVE

CHANGES (BTRFS-PROGS)

Signed release tarballs can be found at https://www.kernel.org/pub/linux/kernel/people/kdave/btrfs-progs/

5.1 btrfs-progs-5.18 (2022-05-25)

- **fixes:**
  - dump-tree: don't print traling zeros in checksums
  - recognize paused balance as exclusive operation state, allow to start device add
  - convert: properly initialize target filesystem label
  - mkfs: don't create free space bitmaps for empty filesystem
- restore: make lzo support build-time configurable, print supported compression in help text
- update kernel-lib sources
- **other:**
  - documentation updates, finish conversion to RST, CHANGES and INSTALL could be included into RST
  - fix build detection of experimental mode
  - new tests

5.1.1 btrfs-progs-5.18.1 (2022-06-06)

- **fixes:**
  - convert: fix self reference of toplevel directory
  - build: make kernel lib headers compatible with C++
- zoned mode: verify minimum zone size 4MiB
- libbtrfs: cleanups, merge headers and remove declarations of unexported symbols
- other: documentation updates

5.2 btrfs-progs-5.17 (2022-04-26)

- **check:**
  - repair wrong num_devices in superblock
  - recognize overly long xattr names
  - fix wrong total bytes check for seed device
- auto-repair on read on RAID56
- property set: unify handling of empty value to mean default, changed meaning for property 'compression' to allow reset to default and to set NOCOMPRESS, since kernel 5.14
- **fixes:**
  - dump-tree: print fs-verity items
  - fix location of system chunk on zoned filesystem
  - do not allow setting seeding flag on a filesystem with dirty log
  - mkfs and subpage support: use sectorsize as nodesize fallback for mixed profiles
- preparatory work for extent tree v2, global roots
- **experimental feature (unstable interface, not built by default,**
  do not use for production) * btrfstune: option --csum to switch checksum algorithm
- **other:**
  - cleanups, refactoring
  - update documentation build, remove asciidocs leftovers
  - update fssum to consider xattrs
  - add fsstress

5.3 btrfs-progs-5.16 (2022-01-12)

- rescue: new subcommand clear-uuid-tree to fix failed mount due to bad uuid subvolume keys, caught by tree-checker
- fi du: skip inaccessible files
- prop: properly resolve to symlink targets
- send, receive: fix crash after parent subvolume lookup errors
- **build:**
  - fix build on 5.12+ kernels due to changes in linux/kernel.h
  - fix build on musl with old kernel headers
- **other:**
  - error handling fixes, cleanups, refactoring
  - extent tree v2 preparatory work
  - lots of RST documentation updates (last release with asciidoc sources), https://btrfs.readthedocs.io

5.3.1 btrfs-progs-5.16.1 (2022-02-04)

- mkfs: support DUP on metadata on zoned devices
- subvol delete: drop warning for root when search ioctl fails
- **check:**
  - fix --init-csum-tree to not create checksums for extents that are not supposed to have them
  - add check for metadata item levels
- add udev rule for zoned devices as they require mq-deadline
- build: fix redefinition of ALIGN on mixed old/new kernel/userspace (5.11)
- **other:**
  - typo fixes
  - new tests
  - CI targets updated

5.3.2 btrfs-progs-5.16.2 (2022-02-16)

- mkfs: fix detection of profile type for zoned mode when creating DUP
- **build:**
  - add missing stub for zoned mode helper when zoned mode not enabled
  - fix 64bit types on MIPS and PowerPC
  - improved zoned mode support autodetection, for systems with existing blkzone.h header but missing support for zone capacity
- **other:**
  - doc updates
  - test updates

5.4 btrfs-progs-5.15 (2021-11-05)

- **mkfs: new defaults!**
  - no-holes
  - free-space-tree
  - DUP for metadata unconditionally
- **libbtrfsutil**
  - add missing profile defines
- **libbtrfs**
  - minimize its impact on the other code, refactor and separate implementation where needed, cleanup afterwards, reduced header exports
- **documentation**

  - introduce sphinx build and RST versions of manual pages, will become the new format and replace asciidoc

- **fixes:**
  - fix warning regarding v1 space cache when only v2 (free space tree) is enabled
- **other**
  - lots of cleanups and refactoring
  - zoned mode uses direct io for file backed images
  - new and updated tests

5.4.1 btrfs-progs-5.15.1 (2021-11-22)

- **fixes:**
  - fi usage: fix wrongly reported space of used or unallocated space
  - fix detection of block device discard capability
- check: add more sanity checks for checksum items
- build: make sphinx optional backend for documentation

5.5 btrfs-progs-5.14 (2021-09-10)

- **convert:**
  - new option --uuid to copy, generate or set a given uuid
  - improve output
- **mkfs:**
  - allow to create degenerate raid0 (on 1 device) and raid10 (on 2 devices)
- **image:**
  - improved error messages
  - fix some alignment of restored image
- subvol delete: allow to delete by id when path is not resolvable
- **check:**
  - require alignment of nodesize for 64k page systems
  - detect and fix invalid block groups
- **libbtrfs (deprecated):**
  - remove most exported symbols, leave only a few that are used by snapper
  - no version change (still 0.1)
  - remove btrfs-list.h, btrfsck.h
- **fixes:**
  - reset generation of space v1 if v2 is used

  - fi us: don't wrongly report missing device size when partition is not readable
- **other:**
  - build: experimental features
  - build: better detection of 64bit timestamp support for ext4
  - corrupt-block: block group items
  - new and updated tests
  - refactoring
- **experimental features:**
  - new image dump format, with data

5.5.1 btrfs-progs-5.14.1 (2021-09-20)

- **fixes:**
  - defrag: fix parsing of compression (option -c)
  - add workaround for old kernels when reading zone sizes
  - let only check and restore open the fs with transid failures, namely preventing btrfstune to do so
  - convert: --uuid copy does not fail on duplicate uuids

5.5.2 btrfs-progs-5.14.2 (2021-10-08)

- **fixes**
  - **zoned mode**
    - properly detect non-zoned devices in emulation mode
    - properly create quota tree
    - raid1c3/4 also excluded from unsupported profiles
  - use sysfs-based detection of device discard capability, fix mkfs-time trim for non-standard devices
  - mkfs: fix creation of populated filesystem with free space tree
  - detect multipath devices (needs libudev)
- replace start: add option -K/--nodiscard, similar to what mkfs or device add has
- dump-tree: print complete root_item
- mkfs: add option --verbose
- sb-mod: better help, no checksum calculation on read-only actions
- **subvol show:**
  - print more information (regarding send and receive)
  - print warning if read-write subvolume has received_uuid set
- **property set:**
  - add parameter -f to force changes

  - changing ro->rw switch now needs -f if subvolume has received_uuid set, (see documentation)
- **build**
  - optional libudev (on by default)
- **other**
  - remove deprecated support for CREATE_ASYNC bit for subvolume ioctl
  - CI updates
  - new and updated tests

5.6 btrfs-progs-5.13 (2021-07-13)

- restore: remove loop checks for extent count and directory scan
- inspect dump-tree: new options to print node (--csum-headers) and data checksums (--csum-items)
- **fi usage:**
  - print stripe count for striped profiles
  - print zoned information: size, total unusable
- mkfs: print note about sha256 accelerated module loading issue
- check: ability to reset dev_item::bytes_used
- **fixes**
  - detect zoned kernel support at run time too
  - exclusive op running check return value
- fi resize: support cancel (kernel 5.14)
- device remove: support cancel (kernel 5.14)
- **documentation about general topics**
  - compression
  - zoned mode
  - storage model
  - hardware considerations
- **other**
  - libbtrfsutil API overview
  - help text fixes and updates
  - hash speedtest measure time, cycles using perf and print throughput

5.6.1 btrfs-progs-5.13.1 (2021-07-30)

- build: fix build on musl libc due to missing definition of NAME_MAX
- **check:**
  - batch more work into one transaction when clearing v1 free space inodes
  - detect directories with wrong number of links
- libbtrfsutil: fix race between subvolume iterator and deletion
- mkfs: be more specific about supported profiles for zoned device
- **other:**
  - documentation updates

5.7 btrfs-progs-5.12 (2021-05-10)

- libbtrfsutil: relicensed to LGPL v2.1+
- mkfs: zoned mode support (kernel 5.12+)
- fi df: show zone_unusable per profile type in zoned mode
- fi usage: show total amount of zone_unusable
- fi resize: fix message for exact size
- image: fix warning and enlarge output file if necessary
- **core**
  - refactor chunk allocator for more modes
  - implement zoned mode support: allocation and writes, sb log
  - crypto/hash refactoring and cleanups
  - refactoring and cleanups
- **other**
  - test updates
  - CI updates
  - travis-ci integration disabled
  - docker images updated, more coverage
  - incomplete build support for Android removed
- **doc updates**
  - chattr mode m for 'NOCOMPRESS"
  - swapfile used from fstab
  - how to add a new export to libbtrfsutil
  - update status of mount options since 5.9

5.7.1 btrfs-progs-5.12.1 (2021-05-13)

- build: fix missing symbols in libbtrfs
- mkfs: check for minimal number of zones
- check: fix warning about cache generation when free space tree is enabled
- fix superblock write in zoned mode on 16K pages

5.8 btrfs-progs-5.11 (2021-03-05)

- fix device path canonicalization for device mapper devices
- receive: remove workaround for setting capabilities, all stable kernels have been patched
- receive: fix duplicate mount path detection
- rescue: new subcommand create-control-device
- device stats: minor fix for plain text format output
- build: detect if e2fsprogs support 64bit timestamps
- build: drop libmount, required functionality has been reimplemented
- mkfs: warn when raid56 is used
- balance convert: warn when raid56 is used
- **other**
  - new and updated tests
  - **documentation updates**
    - seeding device
    - raid56 status
  - **CI updates**
    - docker images for various distros

5.8.1 btrfs-progs-5.11.1 (2021-03-24)

- properly format checksums when a mismatch is reported
- check: fix false alert on tree block crossing 64K page boundary
- **convert:**
  - refuse to convert filesystem with 'needs_recovery'
  - update documentation to require fsck before conversion
- balance convert: fix raid56 warning when converting other profiles
- fi resize: improved summary
- **other**
  - build: fix checks and autoconf defines
  - fix symlink paths for CI support scripts

  - updated tests

5.9 btrfs-progs-5.10 (2021-01-18)

- **scrub status:**
  - print percentage of progress
  - add size unit options
- fi usage: also print free space from statfs
- convert: copy full 64 bit timestamp from ext4 if available
- **check:**
  - add ability to repair extent item generation
  - new option to remove leftovers from inode number cache (-o inode_cache)
- check for already running exclusive operation (balance, device add/...) when starting one
- preliminary json output support for 'device stats'
- **fixes:**
  - subvolume set-default: id 0 correctly falls back to toplevel
  - receive: align internal buffer to allow fast CRC calculation
  - logical-resolve: distinguish -o subvol and bind mounts
- build: new dependency libmount
- **other**
  - doc fixes and updates
  - new tests
  - ci on gitlab temporarily disabled
  - debugging output enhancements

5.9.1 btrfs-progs-5.10.1 (2021-02-05)

- static build works again
- **other:**
  - add a way to test static binaries with the testsuite
  - clarify scrub docs
  - update dependencies, minimum version for libmount is 2.24, this may change in the future

5.10 btrfs-progs-5.9 (2020-10-23)

- **mkfs:**
  - switch default to single profile for multi-device filesystem, up to now it was raid0 that may not be simple to convert to some other profile as raid0 needs a workspace on all device for that
  - new option -R for run-time options (eg. mount time enabled), now understands free-space-tree
- **subvolume delete:**
  - refuse to delete the default subvolume (kernel will not allow that but the error reason is not obvious)
  - warn on EPERM, eg. if send is on progress on the subvolume
- **convert:**
  - fix 32bit overflows on large filesystems
  - improved error handling and error messages
  - check free space taking fragmentation into account
- **check:**
  - detect and repair wrong inode generation
  - minor improvement in error reporting on roots
- **libbtrfsutils: follow main package versioning (5.9)**
  - add pkg-config file definitions
- python-btrfsutil: follow main package versioning (5.9)
- inspect tree-stats: print node counts for each level, fanout
- **other:**
  - **docs:**
    - remove obsolete mount options (alloc_start, subvolrootid)
    - deleting default subvolume is not permitted
  - updated or fixed tests
  - .editorconfig updates
  - move files to kernel-shared/
  - **CI:**
    - updated to use zstd 1.4.5
    - fix reiserfs build
    - more builds with asan, ubsan
  - sb-mod updates
  - **build:**
    - print .so versions of libraries in configure summary

5.11 btrfs-progs-5.8 (skipped)

5.12 btrfs-progs-5.7 (2020-07-02)

- **mkfs:**
  - new option to enable features otherwise enabled at runtime, now implemented for quotas, 'mkfs.btrfs -R quota'
  - fix space accounting for small image, DUP and --rootdir
  - option -A removed
- check: detect ranges with overlapping csum items
- fi usage: report correct numbers when plain RAID56 profiles are used
- convert: ensure the data chunks size never exceed device size
- libbtrfsutil: update documentation regarding subvolume deletion
- build: support libkcapi as implementation backend for cryptographic primitives
- core: global options for verbosity (-v, -q), subcommands -v or -q are aliases and will continue to work but are considered deprecated, current command output is preserved to keep scripts working
- **other:**
  - block group code cleanups
  - build warning fixes
  - more files moved to kernel-shared
  - btrfs-debugfs ported to python 3
  - documentation updates
  - new tests

5.13 btrfs-progs-5.6 (2020-04-05)

- inspect logical-resolve: support LOGICAL_INO_V2 as new option '-o', helps advanced dedupe tools
- inspect: user larger buffer (64K) for results
- subvol delete: support deletion by id (requires kernel 5.7+)
- **libbtrfsutil:**
  - support subvolume deletion by id
  - bump version to 1.2
  - library symbols are now versioned
- dump-tree: new option --hide-names, replace any names (file, directory, subvolume, xattr) in the output with stubs
- convert: warn if the filesystem is not mountable on the machine
- **fixes:**
  - restore: proper mirror iteration on decompression error

  - restore: make symlink messages less noisy
  - check: handle holes at the beginning or end of file
  - fix xxhash output on big endian machines
  - receive: fix lookup of subvolume by uuid in case it was already received before
- **other:**
  - new and updated tests
  - add missing binaries in exported testsuite
  - docs updates
  - remove obsolete files
  - move files to more appropriate directories
  - fixes reported by valgrind
  - many typos fixed

5.13.1 btrfs-progs-5.6.1 (2020-05-07)

- print warning when multiple block group profiles exist, update 'fi usage' summary, add docs to maual page explaining the situation
- build: optional support for libgcrypt or libsodium, providing hash implementations
- **other:**
  - fixed, updated and new tests
  - cleanups
  - updated docs

5.14 btrfs-progs-5.5 (skipped)

5.15 btrfs-progs-5.4 (2019-12-03)

- **support new hash algorithms (kernel 5.5):**
  - mkfs.btrfs and btrfs-convert with --csum, crc32c, xxhash, sha256, blake2
- mkfs: support new raid1c3 and raid1c4 block group profiles (kernel 5.5)
- **check:**
  - --repair delays start with a warning, can be skipped using --force
  - enhanced detection of inode types from partial data, more options for repair
- receive: fix quiet option
- image: speed up chunk loading
- **fi usage:**
  - sort devices by id

    - print ratio of used/total per block group type
- rescue zero-log: reset the log pointers directly, avoid reading some other potentially damaged structures
- new make target install-static to install only static binaries/libraries
- **other**
    - docs updates
    - new tests
    - cleanups and refactoring

5.15.1 btrfs-progs-5.4.1 (2020-01-09)

- build: fix docbook5 build
- check: do extra verification of extent items, inode items and chunks
- qgroup: return ENOTCONN if quotas not running (needs updated kernel)
- other: various test fixups

5.16 btrfs-progs-5.3 (2019-10-21)

- **mkfs:**
    - new option to specify checksum algorithm (only crc32c)
    - fix xattr enumeration
- dump-tree: BFS (breadth-first) traversal now default
- libbtrfsutil: remove stale BTRFS_DEV_REPLACE_ITEM_STATE_x defines
- ci: add support for gitlab
- **other:**
    - preparatory work for more checksum algorithms
    - docs update
    - switch to docbook5 backend for asciidoc
    - fix build on uClibc due to missing backtrace()
    - lots of printf format fixups

5.16.1 btrfs-progs-5.3.1 (2019-10-25)

- libbtrfs: fix link breakage due to missing symbols

5.17 btrfs-progs-5.2 (2019-07-05)

- subvol show: print qgroup information when available
- **scrub:**
  - status: show ETA, revamp the whole output
  - fix reading/writing of last position on resume/cancel, potentially skipping part of the filesystem on next resume
- dump-tree: add new option --noscan to use only devices given on the commandline
- all-in-one binary (busybox style) with mkfs.btrfs, btrfs-image, btrfs-convert, btrfstune
- image: fix hang when there are more than 32 cpus online and compression is requested
- convert: fix some false ENOSPC errors when --rootdir is used
- build: fix gcc9 warnings
- **core changes**
  - command handling cleanups
  - dead code removal
  - cmds-* files moved to cmds/
  - other shared userspace files moved to common/
  - utils.c split into more files
  - preparatory work for more output formats
  - libbtrfsutil: fix unaligned access
- **other**
  - new and updated tests
  - fix tests so CI passes again
  - sb-mod can modify more superblock items

5.17.1 btrfs-progs-5.2.1 (2019-07-26)

- scrub status: fix ETA calculation after resume
- check: fix crash when using -Q
- restore: fix symlink owner restoration
- mkfs: fix regression with mixed block groups
- core: fix commit to process all delayed refs
- **other:**
  - minor cleanups
  - test updates

5.17.2 btrfs-progs-5.2.2 (2019-09-05)

- **check:**
  - fix false report of wrong byte count for orphan inodes
  - option -E was not handled correctly
  - new check and repair for root item generation
- balance: check for full-balance before background fork
- mkfs: check that total device size does not overflow 16EiB
- dump-tree: print DEV_STATS key type
- **other:**
  - new and updated tests
  - doc fixups and updates

5.18 btrfs-progs-5.1 (2019-05-17)

- (version 5.0 skipped)
- **check:**
  - repair: flush/FUA support to avoid breaking metadata COW
  - file extents repair no longer relies on data in extent tree
  - lowmem: fix false error reports about gaps between extents
  - add inode mode check and repair for various objects
  - add check for invalid combination of nocow/compressed extents
- device scan: new option to forget scanned devices
- mkfs: use same chunk size as kernel for initial creation
- dev-replace: better report when other exclusive operation is running
- help: for syntax errors on command line, print only the relevant messages, not the whole help text
- **receive:**
  - new option for quiet mode
  - on -vv print information about written ranges
  - fix endless loop with --dump on an invalid stream
- defrag: able open files in RO mode (needs kernel support to work)
- dump-tree: --block can be specified multiple times
- libbtrfsutil: fix: don't close fd on error in btrfs_util_subvolume_id_fd
- **core:**
  - add sync before superblock write
  - better error handling on the transaction commit path
  - try to find best copy when reading tree blocks

  - update backup roots on commit transaction
- **other:**
  - fuzz tests pass and are enabled in CI
  - cleanups
  - new tests

5.18.1 btrfs-progs-5.1.1 (2019-06-11)

- convert and mkfs will try to use optimized crc32c
- fi show: accept a file-backed image
- fi show: fix possible crash when device is deleted in parallel
- **build:**
  - support extra flags for python bindings
  - separate LDFLAGS for libbtrfsutil
- **other:**
  - space reservation fixes or debugging improvements
  - V0 extent code removed
  - more tests and cleanups

5.19 btrfs-progs-4.x (2019-02-25)

5.19.1 btrfs-progs-4.20 (2019-01-19)

- **new feature: metadata uuid**
  - lightweight change of UUID without rewriting all metadata (incompatible change)
  - done by btrfstune -m/-M
  - needs kernel support, 5.0+
- **image:**
  - fix block groups when restoring from multi-device image
  - only enlarge result image if it's a regular file
- **check**
  - more device extent checks and fixes
  - can repair dir item with mismatched hash
- mkfs: uuid tree created with proper contents
- fix mount point detection due to partial prefix match
- **other:**
  - new tests

  - libbtrfsutil: fix tests if kernel lacks support for new subvolume ioctls
  - build fixes
  - doc fixes

btrfs-progs-4.20.1 (2019-01-23)

- libbtrfs: fix build of external tools due to missing symbols
- ci: enable library test

btrfs-progs-4.20.2 (2019-02-25)

- ci: use newer distro for builds
- dump-super: minor output fixup
- revert fix for prefix detection of receive path, this is temporary and unbreaks existing user setups

5.19.2 btrfs-progs-4.19 (2018-11-03)

- check: support repair of fs with free-space-tree feature
- **core:**
  - port delayed ref infrastructure from kernel
  - support write to free space tree
- dump-tree: new options for BFS and DFS enumeration of b-trees
- quota: rescan is now done automatically after 'assign'
- btrfstune: incomplete fix to uuid change
- subvol: fix 255 char limit checks
- completion: complete block devices and now regular files too
- **docs:**
  - ship uncompressed manual pages
  - btrfsck uses a manual page link instead of symlink
- **other**
  - improved error handling
  - docs
  - new tests

btrfs-progs-4.19.1 (2018-12-05)

- **build fixes**
  - big-endian builds fail due to bswap helper clashes
  - 'swap' macro is too generic, renamed to prevent build failures
- **libbtrfs**
  - minor version update to 1.1.0
  - fix default search to top=0 as documented
  - rename 'async' to avoid future python binding problems

  - add support for unprivileged subvolume listing ioctls
  - added tests, API docs
- **other**
  - lot of typos fixed
  - warning cleanups
  - doc formatting updates
  - CI tests against zstd 1.3.7

5.19.3 btrfs-progs-4.18 (skipped)

5.19.4 btrfs-progs-4.17 (2018-06-14)

- **check**
  - many lowmem mode improvements
  - properly report qgroup mismatch errors
  - check symlinks with append/immutable flags
- **fi usage**
  - correctly calculate allocated/unallocated for raid10
  - minor output updates
- **mkfs**
  - detect ENOSPC on thinly provisioned devices
  - fix spurious EEXIST during directory traversal
- restore: fix relative path for restore target
- dump-tree: print symbolic tree names for backrefs
- send: fix regression preventing send -p with subvolumes mounted on "/"
- corrupt-tree: refactoring and command line updates
- **build**
  - make it work with e2fsprogs < 1.42 again
  - restore support for autoconf 2.63
  - detect if -std=gnu90 is supported
- **other**
  - new tests
  - cleanups

btrfs-progs-4.17.1 (2018-08-06)

- **check:**
  - add ability to fix wrong ram_bytes for compressed inline files
  - beautify progress output

- btrfstune: allow to continue uuid change after unclean interruption
- **several fuzz fixes:**
  - detect overlapping chunks
  - chunk loading error handling
  - don't crash with unexpected root refs to extents
- relax option parsing again to allow mixing options and non-options arguments
- fix qgroup rescan status reporting
- **build:**
  - drop obsolete dir-test
  - new configure option to disable building of tools
  - add compatibility options --disable-static and --disable-shared
- **other:**
  - cleanups and preparatory work
  - new test images

5.19.5 btrfs-progs-4.16 (2018-04-06)

- **libbtrfsutil - new LGPL library to wrap userspace functionality**
  - **several 'btrfs' commands converted to use it:**
    - properties
    - filesystem sync
    - subvolume set-default/get-default/delete/show/sync
  - python bindings, tests
- **build**
  - use configured pkg-config path
  - CI: add python, musl/clang, built dependencies caching
  - convert: build fix for e2fsprogs 1.44+
  - don't install library links with wrong permissions
- **fixes**
  - prevent incorrect use of subvol_strip_mountpoint
  - dump-super: don't verify csum for unknown type
  - convert: fix inline extent creation condition
- **check:**
  - lowmem: fix false alert for 'data extent backref lost for snapshot'
  - lowmem: fix false alert for orphan inode
  - lowmem: fix false alert for shared prealloc extents
- **mkfs:**

    - add UUID and otime to root of FS_TREE - with the uuid, snapshots will be now linked to the toplevel subvol by the parent UUID
    - don't follow symlinks when calculating size
    - pre-create the UUID tree
    - fix --rootdir with selinux enabled
- dump-tree: add option to print only children nodes of a given block
- image: handle missing device for RAID1
- **other**
    - new tests
    - test script cleanups (quoting, helpers)
    - tool to edit superblocks
    - updated docs

btrfs-progs-4.16.1 (2018-04-24)

- remove obsolete tools: btrfs-debug-tree, btrfs-zero-log, btrfs-show-super, btrfs-calc-size
- sb-mod: new debugging tool to edit superblock items
- mkfs: detect if thin-provisioned device does not have enough space
- check: don't try to verify checksums on metadata dump images
- build: fail documentation build if xmlto is not found
- build: fix build of btrfs.static

5.19.6 btrfs-progs-4.15 (2018-02-01)

- mkfs --rootdir reworked, does not minimize the final image but can be still done using a new option --shrink
- fix allocation of system chunk, don't allocate from the reserved area
- **other**
    - new and updated tests
    - cleanups, refactoring
    - doc updates

btrfs-progs-4.15.1 (2018-02-16)

- **build**
    - fix build on musl
    - support asciidoctor for doc generation
- **cleanups**
    - sync some code with kernel
    - check: move code to own directory, split to more files
- **tests**
    - more build tests in travis

  - tests now pass with asan and ubsan
  - testsuite can be exported and used separately

5.19.7 btrfs-progs-4.14 (2017-11-16)

- build: libzstd now required by default
- check: more lowmem mode repair enhancements
- subvol set-default: also accept path
- prop set: compression accepts no/none, same as “”
- filesystem usage: enable for filesystem on top of a seed device
- rescue: new command fix-device-size
- **other**
  - new tests
  - cleanups and refactoring
  - doc updates

btrfs-progs-4.14.1 (2018-01-05)

- dump-tree: print times of root items
- check: fix several lowmem mode bugs
- convert: fix rollback after balance
- **other**
  - new and updated tests, enabled lowmem mode in CI
  - docs updates
  - fix travis CI build
  - build fixes
  - cleanups

5.19.8 btrfs-progs-4.13 (2017-09-08)

- convert: reiserfs support
- check: new option --force to allow check of a mounted filesystem (no repair)
- mkfs: --rootdir will now copy special files
- dump-tree: minor output changes
- inspect rootid: accept file as argument
- dev usage: don’t calculate slack space for missing devices
- fi du: don’t print error on EMPTY_SUBVOL (inode number 2)
- **build:**
  - fixed support for sanitization features on gcc (tsan, asan, ubsan)
  - fix PIE build

- **other:**
  - misc cleanups and stability fixes
  - travis CI enhancements
  - new tests, fuzzed images
  - testsuite cleanups

btrfs-progs-4.13.1 (2017-09-25)

- image: speed up generating the sanitized names, do not generate unprintable chars
- completion: add missing commands, better mount point detection
- restore: add zstd support; libzstd detected automatically, will be requested by default in the future, or can be configured out
- **other:**
  - misc fixes found by sparse
  - doc enhancements, ioctl manual page started
  - updated and new tests
  - build fixes

btrfs-progs-4.13.2 (2017-10-06)

- **subvol list:**
  - don't list toplevel subvolume among deleted (broken since 4.8.3)
  - minor adjustments of uuid print format
- **subvol delete:**
  - fix swapped behaviour of --commit-each and --commit-after
  - fix potentially lost sync if subvolumes are from different filesystems
- check: add cache for metadata blocks, should improve performance
- **other:**
  - new tests, testsuite updates
  - doc updates
  - cleanups

btrfs-progs-4.13.3 (2017-10-16)

- check: fix --force, wrong check for a mounted block device
- build: fix --with-convert parsing
- subvol list: don't list TOPLEVEL
- other: update tests

5.19.9 btrfs-progs-4.12 (2017-07-26)

- subvol show: new options --rootid, --uuid to show subvol by the given spec
- convert: progress report fixes, found by tsan
- image: progress report fixes, found by tsan
- fix infinite looping in find-root, or when looking for free extents
- **other:**
  - code refactoring
  - docs updates
  - build: ThreadSanitizer support
  - tests: stricter checks for mounted filesystem

btrfs-progs-4.12.1 (2017-08-25)

- **build:**
  - fix cross-compilation
  - use gnu90 explicitly
- dump-tree: more relaxed checks so -b can print block on a damaged fs
- convert: fix the 1MB range exclusion
- check: more dir_item hash checks
- **other**
  - added missing getopt spec for some options
  - doc fixes
  - cleanups
  - test updates

5.19.10 btrfs-progs-4.11 (2017-05-18)

- receive: fix handling empty stream with -e (multi-stream)
- send dump: fix printing long file names
- stability fixes for: dump-super, print-tree, check
- option parser updates: global options are parsed before the subcommand name (old xfstests will fail)
- new and updated tests
- documentation updates

btrfs-progs-4.11.1 (2017-06-30)

- image: restoring from multiple devices
- dev stats: make --check option work
- check: fix false alert with extent hole on a NO_HOLE filesystem
- check: lowmem mode, fix false alert in case of mixed inline and compressed extent

- convert: work with large filesystems (many TB)
- convert: fix overwriting of eb header flags
- convert: do not clear NODATASUM flag in inodes when run with --no-datasum
- docs updates
- build: sync Android.mk with Makefile
- **tests:**
  - new tests
  - fix 008 and 009, shell quotation mistake

5.19.11 btrfs-progs-4.10 (2017-03-08)

- send: dump output fixes: missing newlies
- check: several fixes for the lowmem mode, improved error reporting
- **build**
  - removed some library deps for binaries that not use them
  - ctags, cscope
  - split Makefile to the autotool generated part and the rest, not needed to autogen.sh after adding a file
- shared code: sync easy parts with kernel sources
- **other**
  - lots of cleanups
  - source file reorganization: convert, mkfs, utils
  - lots of spelling fixes in docs, other updates
  - more tests

btrfs-progs-4.10.1 (2017-03-17)

- receive: handle subvolume in path clone
- convert: rollback fixed (rewrite was needed to address previous design issues)
- build: fix build of 3rd party tools, missing <linux/sizes.h>
- dump-tree: print log trees
- **other**
  - new and updated tests

btrfs-progs-4.10.2 (2017-03-31)

- check: lowmem mode fix for false alert about lost backrefs
- convert: minor bugfix
- library: fix build, missing symbols, added tests

5.19.12 btrfs-progs-4.9 (2016-12-23)

- check: many lowmem mode updates
- send: use splice syscall to copy buffer from kernel
- receive: new option to dump the stream in textual form
- **convert:**
  - move sources to own directory
  - prevent accounting of blocks beyond end of the device
  - make it work with 64k sectorsize
- mkfs: move sources to own directory
- defrag: warns if directory used without -r
- **dev stats:**
  - new option to check stats for non-zero values
  - add long option for -z
- library: version bump to 0.1.2, added subvol_uuid_search2
- **other:**
  - cleanups
  - docs updates

btrfs-progs-4.9.1 (2017-01-27)

- **check:**
  - use correct inode number for lost+found files
  - lowmem mode: fix false alert on dropped leaf
- size reports: negative numbers might appear in size reports during device deletes (previously in EiB units)
- mkfs: print device being trimmed
- defrag: v1 ioctl support dropped
- quota: print message before starting to wait for rescan
- qgroup show: new option to sync before printing the stats
- **other:**
  - corrupt-block enhancements
  - backtrace and co. cleanups
  - doc fixes

5.19.13 btrfs-progs-4.8 (2016-10-05)

- error handling improvements all over the place
- new fuzzed images, test updates
- doc fixups
- minor cleanups and improvements
- kernel library helpers moved to own directory
- qgroup: fix regression leading to incorrect status after check, introduced in 4.7

btrfs-progs-4.8.1 (2016-10-12)

- 32bit builds fixed
- build without backtrace support fixed

btrfs-progs-4.8.2 (2016-10-26)

- convert: also convert file attributes
- convert: fix wrong tree block alignment for unaligned block group
- check: quota verify fixes, handle reloc tree
- build: add stub for FIEMAP_EXTENT_SHARED, compiles on ancient kernels
- build: add stub for BUILD_ASSERT when ioctl.h is included
- dump-tree: don't crash on unrecognized tree id for -t
- **tests:**
  - add more ioctl tests
  - convert: more symlink tests, attribute tests
  - quota verify for reloc tree
- other cleanups

btrfs-progs-4.8.3 (2016-11-11)

- **check:**
  - support for clearing space cache (v1)
  - size reduction of inode backref structure
- **send:**
  - fix handling of multiple snapshots (-p and -c options)
  - transfer buffer increased (should reduce number of context switches)
  - reuse existing file for output (-f), eg. when root cannot create files (NFS)
- **dump-tree:**
  - print missing items for various structures
  - new: dev stats, balance status item
  - sync key names with kernel (the persistent items)
- subvol show: now able to print the toplevel subvolume -- the creation time might be wrong though
- **mkfs:**

    - store the creation time of toplevel root inode
    - print UUID in the summary
- build: travis CI for devel
- **other:**
    - lots of cleanups and refactoring
    - switched to on-stack path structure
    - fixes from coverity, asan, ubsan
    - new tests
    - updates in testing infrastructure
    - fixed convert test 005

btrfs-progs-4.8.4 (2016-11-25)

- check: support for clearing space cache v2 (free-space-tree)
- **send:**
    - more sanity checks (with tests), cleanups
    - fix for fstests/btrfs/038 and btrfs/117 failures
- **build:**
    - fix compilation of standalone ioctl.h, pull NULL definition
    - fix library link errors introduced in 4.8.3
- **tests:**
    - add more fuzzed images from bugzilla
    - add bogus send stream checks
    - fixups and enhancements for CI environment builds
    - misc refinements and updates of testing framework
- **other:**
    - move sources for btrfs-image to own directory
    - deprecated and not build by default: btrfs-calc-size, btrfs-show-super
    - docs updates

btrfs-progs-4.8.5 (2016-11-30)

- receive: fix detection of end of stream (error reported even for valid streams)
- **other:**
    - added test for the receive bug
    - fix linking of library-test

5.19.14 btrfs-progs-4.7 (2016-07-29)

- convert: fix creating discontig extents
- check: speed up traversing heavily reflinked extents within a file
- check: verify qgroups of higher levels
- check: repair can now fix wrong qgroup numbers
- balance: new option to run in the background
- defrag: default extent target size changed to 32MiB
- du: silently skip non-btrfs dirs/files
- documentation updates: btrfs(5), btrfs(8), balance, subvolume, scrub, filesystem, convert
- **bugfixes:**
  - unaligned access (reported for sparc64) in raid56 parity calculations
  - use /bin/bash
  - other stability fixes and cleanups
- more tests

btrfs-progs-4.7.1 (2016-08-25)

- **check:**
  - new optional mode: optimized for low memory usage (memory/io tradeoff)
  - --mode=lowmem, not default, still considered experimental
  - does not work with --repair yet
- convert: regression fix, ext2_subvol/image rw permissions
- **mkfs/convert:**
  - two-staged creation, partially created filesystem will not be recognized
  - improved error handling (fewer BUG_ONs)
- convert: preparation for more filesystems to convert from
- documentation updates: quota, qgroup
- **other**
  - message updates
  - more tests
  - more build options, enhanced debugging

btrfs-progs-4.7.2 (2016-09-05)

- **check:**
  - urgent fix: false report of backref mismatches; do not --repair last unaffected version 4.6.1 (code reverted to that state)
- **fuzzing and fixes**
  - added more sanity checks for various structures
  - testing images added

- build: udev compatibility: do not install .rules on version < 190
- **other:**
  - dump-super: do not crash on garbage value in csum_type
  - minor improvements in messages and help strings
- **documentation:**
  - filesystem features

btrfs-progs-4.7.3 (2016-09-21)

- fixed free space tree compat status
- check: low-mem mode: handle partially dropped snapshots
- dump-super: consolidate options for superblock copy
- tree-stats: check mount status
- subvol delete: handle verbosity option
- defrag: print correct error string
- mkfs: fix reading rotational status
- **other:**
  - UBSAN build option
  - documentation updates
  - enhanced tests: convert, fuzzed images, more tools to run on fuzzed images

5.19.15 btrfs-progs-4.6 (2016-06-10)

- **convert - major rewrite:**
  - fix a long-standing bug that led to mixing data blocks into metadata block groups
  - the workaround was to do full balance after conversion, which was recommended practice anyway
  - explicitly set the lowest supported version of e2fstools to 1.41
- provide and install udev rules file that addresses problems with device mapper devices, renames after removal
- send: new option: quiet
- dev usage: report slack space (device size minus filesystem area on the dev)
- image: support DUP
- build: short options to enable debugging builds
- **other:**
  - code cleanups
  - build fixes
  - more tests and other enhancements

btrfs-progs-4.6.1 (2016-06-24)

- filesystem resize: negative resize argument accepted again
- qgroup rescan: fix skipping when rescan is in progress

- mkfs: initialize stripesize to correct value
- testsuite updates, mostly convert tests
- **documentation updates**
  - btrfs-device, btrfs-restore manual pages enhanced
  - misc fixups

5.19.16 btrfs-progs-4.5 (2016-03-20)

New/moved commands:

- btrfs-show-super -> btrfs inspect-internal dump-super
- btrfs-debug-tree -> btrfs inspect-internal dump-tree

New commands:

- btrfs filesystem du - calculate disk usage, including shared extents

Enhancements:

- device delete - delete by id (needs kernel support, not merged to 4.6)
- check - new option to specify chunk root
- debug-tree/dump-tree - option -t understands human readable name of the tree (along numerical ids)
- btrfs-debugfs - can dump block group information

Bugfixes:

- all commands should accept the option separator "--"
- several fixes in device scan
- restore works on filesystems with sectorsize > 4k
- debug-tree/dump-tree - print compression type as string
- subvol sync: fix crash, memory corruption
- argument parsing fixes: subvol get-default, qgroup create/destroy/ assign, inspect subvolid-resolve
- check for block device or regular file in several commands

Other:

- documentation updates
- manual pages for the moved tools now point to btrfs-filesystem
- testsuite updates

btrfs-progs-4.5.1 (2016-03-31)

- mkfs: allow DUP on multi-device filesystems
- bugfixes: build fixes, assorted other fixes

btrfs-progs-4.5.2 (2016-05-02)

- new/moved command: btrfs-calc-stats -> btrfs inspect tree-stats

- check: fix false alert for metadata blocks crossing stripe boundary
- check: catch when qgroup numbers mismatch
- check: detect running quota rescan and report mismatches
- balance start: add safety delay before doing a full balance
- filesystem sync: is now silent
- filesystem show: don't miss filesystems with partially matching uuids
- dev ready: accept only one argument for device
- dev stats: print "devid:N" for a missing device instead of "(null)"
- **other:**
  - lowest supported version of e2fsprogs is 1.41
  - minor cleanups, test updates

btrfs-progs-4.5.3 (2016-05-11)

- ioctl: fix unaligned access in buffer from TREE_SEARCH; might cause SIGBUS on architectures that do not support unaligned access and do not performa any fixups
- improved validation checks of superblock and chunk-related structures
- subvolume sync: fix handling of -s option
- balance: adjust timing of safety delay countdown with --full-balance
- rescue super-recover: fix reversed condition check
- check: fix bytes_used accounting
- documentation updates: mount options, scrub, send, receive, select-super, check, mkfs
- testing: new fuzzed images, for superblock and chunks

5.19.17 btrfs-progs-4.4 (2016-01-18)

User visible changes:

- mkfs.btrfs --data dup

People asked about duplicating data on a single device for a long time. There are no technical obstacles preventing that, so it got enabled with a warning about potential dangers when the device will not do the duplicated copies. See mkfs.btrfs section DUP PROFILES ON A SINGLE DEVICE.

- **support balance filters added/enhanced in linux 4.4**
  - usage=min..max -- enhanced to take range
  - stripes=min..max -- new, filter by stripes for raid0/10/5/6
  - limit=min..max -- enhanced to take range

Note: due to backward compatibility, the range maximum for 'usage' is not inclusive as for the others, to keep the same behaviour as usage=N .

- manual pages enhanced (btrfs, mkfs, mount, filesystem, balance)
- error messages updates, rewordings -- some fstests may break due to that

- added support for free-space-tree implementation of space cache -- this requires kernel 4.5 and is not recommended for non-developers yet
- btrfs filesystem usage works with mixed blockgroups

Other:

- installation to /usr/local -- this has unintentionally changed during conversion to autotools in 3.19
- check: fix a false alert where extent record has wrong metadata flag
- improved stability on fuzzed/crafted images when reading sys array in superblock
- build: the 'ar' tool is properly detected during cross-compilation
- debug-tree: option -t understands ids for tree root and chunk tree
- preparatory work for btrfs-convert rewrite
- sparse, gcc warning fixes
- more memory allocation failure handling
- cleanups
- more tests

Bugfixes:

- chunk recovery: fix floating point exception
- chunk recovery: endianity bugfix during rebuild
- mkfs with 64K pages and nodesize reported superblock checksum mismatch
- check: properly reset nlink of multi-linked file

btrfs-progs-4.4.1 (2016-02-26)

- find-root: don't skip the first chunk
- free-space-tree compat bits fix
- build: target symlinks
- documentation updates
- test updates

5.19.18 btrfs-progs-4.3 (2015-11-06)

- **mkfs**
  - mixed mode is not forced for filesystems smaller than 1GiB
  - mixed mode broken with mismatching sectorsize and nodesize, fixed
  - print version info earlier
  - print devices sorted by id
  - do not truncate target image with --rootsize
- **filesystem usage:**
  - don't print global block reserve

    - print device id
    - minor output tuning
    - other cleanups
- **calc-size:**
    - div-by-zero fix on an empty filesystem
    - fix crash
- **bugfixes:**
    - more superblock sanity checks
    - consistently round size of all devices down to sectorsize
    - misc leak fixes
    - convert: don't try to rollback with a half-deleted ext2_saved subvolume
- **other:**
    - check: add progress indicator
    - scrub: enhanced error message
    - show-super: read superblock from a given offset
    - add README
    - docs: update manual page for mkfs.btrfs, btrfstune, balance, convert and inspect-internal
    - build: optional build with more warnings (W=...)
    - build: better support for static checkers
    - build: html output of documentation
    - pretty-print: last_snapshot for root_item
    - pretty-print: stripe dev uuid
    - error reporting wrappers, introduced and example use
    - refactor open_file_or_dir
    - other docs and help updates
- **testing:**
    - test for nodes crossing stripes
    - test for broken 'subvolume sync'
    - basic tests for mkfs, raid option combinations
    - basic tests for fuzzed images (check)
    - command instrumentation (eg valgrind)
    - print commands if requested
    - add README for tests

btrfs-progs-4.3.1 (2015-11-16)

- **fixes**
    - device delete: recognize 'missing' again

  - mkfs: long names are not trimmed when doing ssd check
  - support partitioned loop devices
- **other**
  - replace several mallocs with on-stack variables
  - more memory allocation failure handling
  - add tests for bugs fixed
  - cmd-device: switch to new message printing helpers
  - minor code cleanups

5.19.19 btrfs-progs-4.2 (2015-09-03)

- **enhancements:**
  - mkfs: do not create extra single chunks on multiple devices
  - resize: try to guess the minimal size, 'inspect min-dev-size'
  - qgroup assign: add option to schedule rescan
  - chunk-recover: be more verbose about the scanning process
- **fixes:**
  - **check:**
    * find stripes crossing stripe boundary -- created by convert
    * print correct range for file hole when there are no extents and learn how to fix it
  - replace: more sanity checks
  - convert: concurrency fixes related to reporting progress
  - find-root: option -a will not skip the current root anymore
  - subvol list: fix occasional crash
  - do not create stripes crossing stripe boundary
- **build:**
  - fixes for musl libc
  - preliminary support for android (not working yet, more code changes needed)
  - new EXTRA_CFLAGS and EXTRA_LDFLAGS
- **other:**
  - lots of cleanups
  - tests: lots of updates, new tests, framework improvements
  - documentation updates
  - debugging: print-tree shows stripe length

btrfs-progs-4.2.1 (2015-09-20)

- **fix an off-by-one error in cross-stripe boundary check**

  - if nodesize was 64k, any metadata block was reported as crossing, this leads to mkfs failure for example due to "no free blocks found"
  - for other nodesizes, if the end of the metadata block was 64k aligned, it was incorrectly reported by fsck
- convert: don't write uninitialized data to image
- **image:**
  - don't loop with option -t0
  - don't create threads if compression is not requested
- other: minor cleanups

btrfs-progs-4.2.2 (2015-10-05)

- filesystem label: use fallback if the label ioctl is not available
- convert: check nodesize constraints against commandline features (-O)
- scrub: report status 'running' until all devices are finished
- device scanning might crash in some scenarios
- filesystem usage: print summary for non-root users

btrfs-progs-4.2.3 (2015-10-19)

- subvol sync: make it actually work again: it's been broken since 4.1.2, due to a reversed condition it returned immediately instead of waiting
- scanning: do not scan already discovered filesystems (minor optimization)
- convert: better error message in case the filesystem is not finalized
- restore: off-by-one symlink path check fix

5.19.20 btrfs-progs-4.1 (2015-06-22)

Bugfixes:

- fsck.btrfs: no bash-isms
- bugzilla 97171: invalid memory access (with tests)
- **receive:**
  - cloning works with --chroot
  - capabilities not lost
- mkfs: do not try to register bare file images
- option --help accepted by the standalone utilities

Enhancements:

- corrupt block: ability to remove csums
- **mkfs:**
  - warn if metadata redundancy is lower than for data
  - options to make the output quiet (only errors)
  - mixed case names of raid profiles accepted

    - rework the output:
    - more comprehensive, 'key: value' format
- **subvol:**
    - **show:**
        * print received uuid
        * update the output
        * new options to specify size units
    - sync: grab all deleted ids and print them as they're removed, previous implementation only checked if there are any to be deleted - change in command semantics
- scrub: print timestamps in days HMS format
- **receive:**
    - can specify mount point, do not rely on /proc
    - can work inside subvolumes
- send: new option to send stream without data (NO_FILE_DATA)
- convert: specify incompat features on the new fs
- **qgroup:**
    - show: distinguish no limits and 0 limit value
    - limit: ability to clear the limit
- help for 'btrfs' is shorter, 1st level command overview
- debug tree: print key names according to their C name

New:

- rescue zero-log
- **btrfsune:**
    - rewrite uuid on a filesystem image
    - new option to turn on NO_HOLES incompat feature

Deprecated:

- standalone btrfs-zero-log

Other:

- **testing framework updates**
    - uuid rewrite test
    - btrfstune feature setting test
    - zero-log tests
    - more testing image formats
- manual page updates
- ioctl.h synced with current kernel uapi version
- convert: preparatory works for more filesystems (reiserfs pending)

- use static buffers for path handling where possible
- add new helpers for send uilts that check memory allocations, switch all users, deprecate old helpers
- Makefile: fix build dependency generation
- map-logical: make it work again

btrfs-progs-4.1.1 (2015-07-10) -- Do not use this version!

Bugfixes:

- defrag: threshold overflow fix
- **fsck:**
  - check if items fit into the leaf space
  - fix wrong nbytes
- **mkfs:**
  - create only desired block groups for single device
  - preparatory work for fix on multiple devices

Enhancements:

- new alias for 'device delete': 'device remove'

Other:

- fix compilation on old gcc (4.3)
- documentation updates
- debug-tree: print nbytes
- test: image for corrupted nbytes
- corrupt-block: let it kill nbytes

btrfs-progs-4.1.2 (2015-07-14)

- urgent bugfix: mkfs creates invalid filesystem, must be recreated

5.19.21 btrfs-progs-4.0 (2015-04-29)

- **resize:**
  - don't accept file as an argument (it's confusing)
  - print better error message in case of an error
- restore: optionally restore metadata (time, mode, uid/gid)
- receive: optionally enforce chroot
- new rescue subcommand 'zero-log', same as btrfs-zero-log, but now also part of the main utility
- **check:**
  - free space checks match kernel, fixes incorrect reports
- convert: fix setting of checksum bit if --no-datasum is used
- fsck.btrfs: don't print messages

- fix quota rescan on PPC64 (mangled ioctl number)
- test updates
- documentation: files renamed to .asciidoc, misc fixups

btrfs-progs-4.0.1 (2015-05-20)

- **restore:**
  - can restore symlinks, new option --symlinks
  - long option variants added
- convert: dropped dependency on acl.h header and libacl is not required for build
- fix for 'check' crash
- device remove error message fix
- preparatory works for fsid change

5.20 btrfs-progs-3.x (2015-03-25)

5.20.1 btrfs-progs-3.19 (2015-03-11)

- build converted to autotools
- **btrfs-image**
  - restore can now run in parallel threads
  - fixed restore of multiple image from multiple devices onto a single dev
  - introduced metadump v2
- check: make --init-csum-tree and --init-extent-tree work together
- find-new: option to search through all metadata even if a root was already found
- convert: show progress by default, can be turned off
- corrupt-block: option to work on a specific root
- bash completion script for all subcommands

btrfs-progs-3.19.1 (2015-03-25)

- **convert:**
  - new option to specify metadata block size
  - --no-progress actually works
- restore: properly handle the page boundary corner case
- **build fixes:**
  - missing macro from public header, BTRFS_BUILD_VERSION
  - wrong handling of --enable-convert
- filesystem usage: reports correct space for degraded mounts
- **other:**
  - mkfs: help string updates

  - completion: added 'usage' subcommands
  - cleanups in qgroup code, preparatory work

5.20.2 btrfs-progs-3.18 (2014-12-30)

- mkfs - skinny-metadata feature is now on by default, first introduced in kernel 3.10
- filesystem usage - give an overview of fs usage in a way that's more comprehensible than existing 'filesystem df'
- device usage - more detailed information about per-device allocations
- **check**
  - option to set a different tree root byte number
  - ability to link lost files to lost+found, caused by a recent kernel bug
  - repair of severely corrupted fs (use with care)
- convert - option to show progress
- subvol create - print the commit mode inline, print the global mode only if --verbose
- other updates: musl-libc support, coverity bugfixes, new test images, documentation

btrfs-progs-3.18.1 (2015-01-09)

- minor fixes
- documentation updates

btrfs-progs-3.18.2 (2015-01-27)

- qgroup show: print human readable sizes, options to say otherwise
- check: new option to explicitly say no to writes
- mkfs: message about trimming is not printed to stderr
- filesystem show: fixed return value
- tests: new infrastructure
- btrfstune: force flag can be used together with seeding option
- backtrace support is back
- getopt cleanups
- doc and help updates

5.20.3 btrfs-progs-3.17 (2014-10-17)

- check: --init-csum-tree actually does something useful, rebuilds the whole csum tree
- /dev scanning for btrfs devices is gone
- /proc/partitions scanning is gone, blkid is used exclusively
- new subcommand subvolume sync
- filesystem df: new options to set unit format
- convert: allow to copy label from the origin, or specify a new one

btrfs-progs-3.17.1 (2014-11-04)

- filesystem df: argument handling
- fix linking with libbtrfs
- replace: better error reporting
- filesystem show: fixed stall if run concurrently with balance
- check: fixed argument parsing for --subvol-extents
- filesystem df: SI prefixes corrected

btrfs-progs-3.17.2 (2014-11-19)

- **check improvements**
  - add ability to replace missing dir item/dir indexes
  - fix missing inode items
  - create missing root dirid
- corrupt block: enhancements for testing fsck
- zero-log: able to reset a fs with bogus log tree pointer (bug_72151)

btrfs-progs-3.17.3 (2014-12-04)

- convert: fix conversion of sparse ext* filesystems
- show: resolve to the correct path
- fsck: more verbose error for root dir problems

5.20.4 btrfs-progs-3.16 (2014-08-26)

- mkfs: new option to specify UUID, drop experimental notice
- check: new option to verify quotas, reduced memory requirements, new option to print extent sharing
- restore: check length before decompression, more error handling, option to loop during restoring
- balance: new filter 'limit'
- recover: allow to read all sb copies
- btrfstune: new option to force dangerous changes
- receive: new option to limit number of errors
- show-super: skip unrecognized sb, add option to force
- debug-tree: print tree by id
- documentation updates

btrfs-progs-3.16.1 (2014-09-15)

- print GlobalReserve in filesystem df output
- new option -R in subvol list
- library version defines
- static build is fixed
- build without documentation is possible

btrfs-progs-3.16.2 (2014-10-01)

- a few fixes in fsck and image tools

5.20.5 btrfs-progs-3.15 (skipped)

5.20.6 btrfs-progs-3.14 (2014-04-06)

- fsck: fixes and enhancements to --init-extent-tree mode
- fsck: chunk-recover updates
- scrub: add force option -f
- send: check if subvolumes are read-only
- subvol delete: add options to affect commit behaviour
- btrfs: add property command group
- restore: add dry-run option
- restore: fix restoring of compressed files
- mkfs: support for no-holes feature
- mkfs: option -r deals with hardlinks and relative paths
- mkfs: discard phase is interruptible
- documentation updates

btrfs-progs-3.14.1 (2014-04-18)

- properties: fix handling of option -t
- restore: fix reading of compressed extents
- minor code and doc updates

btrfs-progs-3.14.2 (2014-05-29)

- documentation is now written in asciidoc and there are manpages for each subcommand
- misc bugfixes

5.20.7 btrfs-progs-3.13 (skipped)

5.20.8 btrfs-progs-3.12 (2013-11-25)

- announcement, tarballs
- first release after 0.19 (2009/06) with a lot of changes

CHAPTER

SIX

CHANGES (FEATURE/VERSION)

Major features or significant feature enhancements by kernel version. For more information look below.

The version states at which version a feature has been merged into the mainline kernel. It does not tell anything about at which kernel version it is considered mature enough for production use. For an estimation on stability of features see [[Status]] page.

6.1 3.x

3.0 - scrub
Read all data and verify checksums, repair if possible.

3.2 - auto raid repair
Automatic repair of broken data from a good copy

3.2 - root backups
Save a few previous versions of the most imporant tree roots at commit time, used by *-o recovery*

3.3 - integrity checker
Optional infrastructure to verify integrity of written metadata blocks

3.3 - backref walking
Groundwork to allow tracking owner of blocks, used via *inspect-internal*

3.3 - restriper
RAID profiles can be changed on-line, balance filters

3.4 - big metadata blocks
Support for metadata blocks larger than page size

> **Note:** Default nodesize is 16k since btrfs-progs 3.12

3.4 - error handling
Generic infrastructure for graceful error handling (EIO)

3.5 - device statistics
Persistent statistics about device errors

3.5 - fsync speedup
Noticeable improvements in fsync() implementation

3.6 - qgroups
Subvolume-aware quotas

3.6 - send/receive
Ability to transfer one filesystem via a data stream (full or incremental) and apply the changes on a remote filesystem.

3.7 - extrefs
Hardlink count limit is lifted to 64k

Note: Default since btrfs-progs 3.12

3.7 - hole puching
Implement the FALLOC_FL_PUNCH_HOLE mode of *fallocate*

3.8 - device replace
Efficient replacement of existing device (add/remove in one go)

3.9 - raid 5/6 *(incomplete)*
Basic support for RAD5/6 profiles, no crash resiliency, replace and scrub support

3.9 - snapshot-aware defrag
Defrag does not break links between shared extents (snapshots, reflinked files)

Note: Disabled since 3.14 (and backported to some stable kernel versions) due to problems. Will be enabled in the future.

3.9 - lightweight send
A mode of *send* that does not add the actual file data to the stream

3.9 - on-line label set/get
Label editable on mounted filesystems

3.10 - skinny metadata

Reduced metadata size (format change) of extents

Note: Default since btrfs-progs 3.18

3.10 - qgroup rescan
Sync qgroups with existing filesystem data

3.12 - uuid tree
A map of subvolume/UUID that vastly speeds up send/receive

3.12 - out-of-bound dedup
Support for deduplicating extents on a given set of files.

3.14 - no-holes
No extent representation for file holes (format change), may reduce overall metadata consumption

3.14 - feature bits in sysfs
/sys/fs/btrfs exports various bits about filesystem capabilities and feature support

3.16 - O_TMPFILE
Mode of open() to safely create a temporary file

3.16 - search ioctl v2
The extended SEARCH_TREE ioctl able to get more than a 4k data

3.18 - auto blockgroup reclaim
Automatically remove blockgroups (aka. chunks) that become completely empty.

3.19 - raid56: scrub, replace
Scrub and device replace works on RAID56 filesystems.

6.2 4.x

4.0 - store otime
Save creation time (otime) for all new files and directories. For future use, current tool cannot read it directly.

4.2 - rootid ioctl accessible
The INO_LOOKUP will return root id (id of the containing subvolume), unrestricted and to all users if the *treeid* is 0.

4.2 - dedupe possible on the same inode
The EXTENT_SAME ioctl will accept the same inode as source and destination (ranges must not overlap).

4.3 - trim all free space
Trim will be performed also on the space that's not allocated by the chunks, not only free space within the allocated chunks.

4.4 - balance filter updates
Enhanced syntax and new balance filters: * limit=min..max * usage=min..max * stripes=min..max

4.5 - free space tree
Improved implementation of free space cache (aka v2), using b-trees.

Note: Default since btrfs-progs 5.15, Kernel 4.9 fixes endianity bugs on big-endian machines, x86* is ok

---

4.5 - balance filter updates
Conversion to data/DUP profile possible through balance filters -- on single-device filesytem.

Note: mkfs.btrfs allows creating DUP on single device in the non-mixed mode since 4.4

4.6 - max_inline default
The default value of max_inline changed to 2048.

4.6 - read features from control device
The existing ioctl GET_SUPPORTED_FEATURES can be now used on the control device (/dev/btrfs-control) and returns the supported features without any mounted filesystem.

4.7 - delete device by id
Add new ioctl RM_DEV_V2, pass device to be deleted by its ID.

4.7 - more renameat2 modes
Add support for RENAME_EXCHANGE and RENAME_WHITEOUT to *renameat2* syscall. This also means that *overlayfs* is now supported on top of btrfs.

4.7 - balance filter updates
Conversion to data/DUP profile possible through balance filters -- on multiple-device filesystems.

Note: mkfs.btrfs allows creating DUP on multiple devices since 4.5.1

4.12 - raid56: auto repair
Scrub will attempt auto-repair (similar to raid1/raid10)

4.13 - statx
Support for the enhanced statx syscall; file creation timestamp

4.13 - sysfs qgroups override
qgroups: new sysfs control file to allow temporary quota override with CAP_SYS_RESOURCE

4.13 - *deprecated mount option alloc_start*
That was a debugging helper, not used and not supposed to be used nowadays.

4.14 - ZSTD compression
New compression algorithm ZSTD, supposedly better ratio/speed performance.

4.14 - improved degraded mount
Allow degraded mount based on the chunk constraints, not device number constraints. Eg. when one device is missing but the remaining one holds all *single* chunks.

4.14 - *deprecated user transaction ioctl*
BTRFS_IOC_TRANS_START and BTRFS_IOC_TRANS_END, no known users, tricky to use; scheduled to be removed in 4.17

4.14 - refine SSD optimizations
The mount option *ssd* does not make any assumptions about block layout or management by the device anymore, leaving only the speedups based on low seek cost active. This could avoid some corner cases leading to excessive fragmentation. https://git.kernel.org/linus/583b723151794e2ff1691f1510b4e43710293875 The story so far.

4.15 - overlayfs
Overlayfs can now use btrfs as the lower filesystem.

4.15 - *ref-verify*
Debugging functionality to verify extent references. New mount option ref-verify, must be built with CONFIG_BTRFS_FS_REF_VERIFY.

4.15 - zlib level
Allow to set the zlib compression level via mount option, eg. like *compress=zlib:9*. The levels match the default zlib compression levels. The default is 3.

4.15 - v2 of LOGICAL_INO ioctl
An enhanced version of ioctl that can translate logical extent offset to inode numbers, "who owns this block". For certain usecases the V1 performs bad and this is addressed by V2. [https://git.kernel.org/linus/d24a67b2d997c860a42516076f3315c2ad2d2884 Read more.]

4.15 - compression heuristics
Apply a few heuristics to the data before they're compressed to decide if it's likely to gain any space savings. The methods: frequency sampling, repeated pattern detection, Shannon entropy calculation.

4.16 - fallocate: zero range
Mode of the [http://man7.org/linux/man-pages/man2/fallocate.2.html *fallocate*] syscall to zero file range.

4.17 - *removed user transaction ioctl*
deprecated in 4.14, see above

4.17 - *rmdir* on subvolumes
Allow rmdir to delete an empty subvolume.

4.18 - XFLAGS ioctl
Add support for ioctl FS_IOC_FSSETXATTR/FS_IOC_FSGETXATTR, successor of FS_IOC_SETFLAGS/FS_IOC_GETFLAGS ioctl. Currently supports: APPEND, IMMUTABLE, NOATIME, NODUMP, SYNC. Note that the naming is very confusing, though it's named *xattr*, it does not mean the extended attributes. It should be referenced as extended inode flags or *xflags*.

4.18 - EXTENT_SAME ioctl / 16MiB chunks
The range for out-of-band deduplication implemented by the EXTENT_SAME ioctl will split the range into 16MiB chunks. Up to now this was the overall limit and effectively only the first 16MiB was deduplicated.

4.18 - GET_SUBVOL_INFO ioctl
New ioctl to read subvolume information (id, directory name, generation, flags, UUIDs, time). This does not require root permissions, only the regular access to to the subvolume.

4.18 - GET_SUBVOL_ROOTREF ioctl
New ioctl to enumerate subvolume references of a given subvolume. This does not require root permissions, only the regular access to to the subvolume.

4.18 - INO_LOOKUP_USER ioctl
New ioctl to lookup path by inode number. This does not require root permissions, only the regular access to to the subvolume, unlike the INO_LOOKUP ioctl.

4.19 - defrag ro/rw
Allow to run defrag on files that are normally accesible for read-write, but are currently opened in read-only mode.

6.3 5.x

5.0 - swapfile
With some limitations where COW design does not work well with the swap implementation (nodatacow file, no compression, cannot be snapshotted, not possible on multiple devices, ...), as this is the most restricted but working setup, we'll try to improve that in the future

5.0 - metadata uuid
An optional incompat feature to assign a new filesystem UUID without overwriting all metadata blocks, stored only in superblock, unlike what `btrfstune -u`

5.1 - FORGET_DEV ioctl
Unregister devices previously added by the scan ioctl, same effect as if the kernel module is reloaded.

5.1 - zstd level
Allow to set the zstd compression level via mount option, eg. like *compress=zstd:9*. The levels match the default zstd compression levels. The default is 3, maximum is 15.

5.2 - pre-write checks
Verify metadata blocks before submitting them to the devices. This can catch consistency problems or bitflips.

5.5 - more checksums
New checksum algorithms: xxhash (64b), SHA256 (256b), BLAKE2b (256b).

5.5 - RAID1C34
RAID1 with 3- and 4- copies (over all devices).

5.6 - async discard
Mode of discard (*mount -o discard=async*) that merges freed extents to larger chunks and submits them for discard in a less intrusive way

5.6 - device info in sysfs
More information about device state can be found in per-filesystem sysfs directory.

5.7 - reflink/clone works on inline files
Inline files can be reflinked to the tail extent of other files

5.7 - faster balance cancel
More cancellation points in balance that will shorten the time to stop processing once <tt>btrfs balance cancel</tt> is called.

5.7 - *removed flag BTRFS_SUBVOL_CREATE_ASYNC*
Remove support of flag BTRFS_SUBVOL_CREATE_ASYNC from subvolume creation ioctl.

5.7 - v2 of snapshot deletion ioctl
New ioctl BTRFS_IOC_SNAP_DESTROY_V2, deletion by subvolume id is now possible.

5.9 - mount option *rescue*
Unified mount option for actions that may help to access a damaged filesystem. Now supports: nologreplay, usebackuproot

5.9 - qgroups in sysfs
The information about qgroup status and relations is exported in */sys/fs/UUID/qgroups*

5.9 - FS_INFO ioctl
Export more information: checksum type, checksum size, generation, metadata_uuid

5.10 - exclusive ops in sysfs
Export which filesystem exclusive operation is running (balance, resize, device add/delete/relpace, ...)

5.11 - remove *inode_cache*
Remove inode number caching feature (mount -o inode_cache)

5.11 - more rescue=
Additional modes for mount option *rescue*=: ignorebadroots/ibadroots, ignoredatacsums/idatacsums. All are exported in sysfs.

5.12 - zoned mode
Support for zoned devices with special allocation/write mode to fixed-size zones. See [[Zoned]].

5.13 - supported_sectorsizes in sysfs
List supported sector sizes in sysfs file /sys/fs/btrfs/features/supported_sectorsizes

5.14 - sysfs scrub bw limit
Tunable bandwidth limit (/sys/fs/btrfs/FSID/devinfo/DEVID/scrub_speed_max) for scrub (and device replace) for a given device.

5.14 - sysfs device stats
The device stats can be also found in /sys/fs/btrfs/FSID/devinfo/DEVID/error_stats.

5.14 - cancellable resize, device delete
The filesystem resize and device delete operations can be cancelled by specifying *cancel* as the device name.

5.14 - property value reset
Change how empty value is interpreted. New behaviour will delete the value and reset it to default. This affects *btrfs.compression* where value *no* sets NOCOMPRESS bit while empty value resets all compression settings (either compression or NOCOMPRESS bit).

5.15 - fsverity
The fs-verity is a support layer that filesystems can hook into to support transparent integrity and authenticity protection of read-only files. https://www.kernel.org/doc/html/latest/filesystems/fsverity.html

5.15 - idmapped mount
Support mount with uid/gid mapped according to another namespace. https://lwn.net/Articles/837566/

5.16 - ZNS in zoned
Zoned namespaces. https://zonedstorage.io/docs/introduction/zns , https://lwn.net/Articles/865988/

5.17 - send and relocation
Send and relocation (balance, device remove, shrink, block group reclaim) can now work in parallel

5.17 - device add vs balance
It is possible to add a device with paused balance

Note: Since kernel 5.17.7 and btrfs-progs 5.17.1

5.17 - *no warning with flushoncommit*
Mounting with *-o flushoncommit* does not triggher the (harmless) warning at each transaction commit

Note: Also backported to 5.15.27 and 5.16.13

5.18 - zoned and DUP metadata
DUP metadata works with zoned mode

5.18 - encoded data ioctl
New ioctls to read and write pre-encoded data (ie. no transformation and directly written as extents), now works for compressed data

5.18 - *removed balance ioctl v1*
The support for ioctl BTRFS_IOC_BALANCE has been removed, superseded by BTRFS_IOC_BALANCE_V2m long time ago

CHAPTER

SEVEN

GLOSSARY

Terms in *italics* also appear in this glossary.

allocator
: Usually *allocator* means the *block* allocator, ie. the logic inside filesystem which decides where to place newly allocated blocks in order to maintain several constraints (like data locality, low fragmentation).

  In btrfs, allocator may also refer to *chunk* allocator, ie. the logic behind placing chunks on devices.

balance
: An operation that can be done to a btrfs filesystem, for example through `btrfs fi balance /path`. A balance passes all data in the filesystem through the *allocator* again. It is primarily intended to rebalance the data in the filesystem across the *devices* when a device is added or removed. A balance will regenerate missing copies for the redundant *RAID* levels, if a device has failed. As of linux kernel 3.3, a balance operation can be made selective about which parts of the filesystem are rewritten.

barrier
: An instruction to the disk hardware to ensure that everything before the barrier is physically written to permanent storage before anything after it. Used in btrfs's *copy on write* approach to ensure filesystem consistency.

block
: A single physically and logically contiguous piece of storage on a device, of size eg. 4K.

block group
: The unit of allocation of space in btrfs. A block group is laid out on the disk by the btrfs *allocator*, and will consist of one or more *chunks*, each stored on a different *device*. The number of chunks used in a block group will depend on its *RAID* level.

B-tree
: The fundamental storage data structure used in btrfs. Except for the *superblocks*, all of btrfs *metadata* is stored in one of several B-trees on disk. B-trees store key/item pairs. While the same code is used to implement all of the B-trees, there are a few different categories of B-tree. The name *btrfs* refers to its use of B-trees.

btrfsck
: Tool in *btrfs-progs* that checks a filesystem *offline* (ie. unmounted), and reports on any errors in the filesystem structures it finds. By default the tool runs in read-only mode as fixing errors is potentially dangerous. See also *scrub*.

btrfs-progs
: User mode tools to manage btrfs-specific features. Maintained at http://github.com/kdave/btrfs-progs.git . The main frontend to btrfs features is the standalone tool *btrfs*, although other tools such as *mkfs.btrfs* and *btrfstune* are also part of btrfs-progs.

chunk
: A part of a *block group*. Chunks are either 1 GiB in size (for data) or 256 MiB (for *metadata*).

chunk tree
A layer that keeps information about mapping between physical and logical block addresses. It's stored within the *system* group.

cleaner
Usually referred to in context of deleted subvolumes. It's a background process that removes the actual data once a subvolume has been deleted. Cleaning can involve lots of IO and CPU activity depending on the fragmentation and amount of shared data with other subvolumes.

copy-on-write
Also known as *COW*. The method that btrfs uses for modifying data. Instead of directly overwriting data in place, btrfs takes a copy of the data, alters it, and then writes the modified data back to a different (free) location on the disk. It then updates the *metadata* to reflect the new location of the data. In order to update the metadata, the affected metadata blocks are also treated in the same way. In COW filesystems, files tend to fragment as they are modified. Copy-on-write is also used in the implementation of *snapshots* and *reflink copies*. A copy-on-write filesystem is, in theory, 'always' consistent, provided the underlying hardware supports *barriers*.

COW
See *copy-on-write*.

default subvolume
The *subvolume* in a btrfs filesystem which is mounted when mounting the filesystem without using the `subvol=` mount option.

device
A Linux block device, e.g. a whole disk, partition, LVM logical volume, loopback device, or network block device. A btrfs filesystem can reside on one or more devices.

df
A standard Unix tool for reporting the amount of space used and free in a filesystem. The standard tool does not give accurate results, but the *btrfs* command from *btrfs-progs* has an implementation of *df* which shows space available in more detail. See the [[FAQ#Why_does_df_show_incorrect_free_space_for_my_RAID_volume.3F|FAQ]] for a more detailed explanation of btrfs free space accounting.

DUP
A form of "*RAID*" which stores two copies of each piece of data on the same *device*. This is similar to *RAID-1*, and protects against *block*-level errors on the device, but does not provide any guarantees if the entire device fails. By default, btrfs uses *DUP* profile for metadata on filesystems with one rotational device, *single* profile on filesystems with one non-rotational device, and *RAID1* profile on filesystems with more than one device.

ENOSPC
Error code returned by the OS to a user program when the filesystem cannot allocate enough data to fulfill the user requested. In most filesystems, it indicates there is no free space available in the filesystem. Due to the additional space requirements from btrfs's *COW* behaviour, btrfs can sometimes return ENOSPC when there is apparently (in terms of *df*) a large amount of space free. This is effectively a bug in btrfs, and (if it is repeatable), using the mount option `enospc_debug` may give a report that will help the btrfs developers. See the [[FAQ#if_your_device_is_large_.28.3E16GiB.29|FAQ entry]] on free space.

extent
Contiguous sequence of bytes on disk that holds file data.

A file stored on disk with 3 extents means that it consists of three fragments of contiguous bytes. See *filefrag*. A file in one extent would mean it is not fragmented.

Extent buffer
An abstraction to allow access to *B-tree* blocks larger than a page size.

fallocate
Command line tool in util-linux, and a syscall, that reserves space in the filesystem for a file, without actually

writing any file data to the filesystem. First data write will turn the preallocated extents into regular ones. See <code>man 1 fallocate</code> and <code>man 2 fallocate</code> for more details.

filefrag
A tool to show the number of extents in a file, and hence the amount of fragmentation in the file. It is usually part of the e2fsprogs package on most Linux distributions. While initially developed for the ext2 filesystem, it works on Btrfs as well. It uses the *FIEMAP* ioctl.

free space cache
Btrfs doesn't track free space, it only tracks allocated space. Free space is by definition any holes in the allocated space, but finding these holes is actually fairly I/O intensive. The free space cache stores a condensed representation of what is free. It is updated on every *transaction* commit.

fsync
On Unix and Unix-like operating systems (of which Linux is the latter), the `lfsync()` system call causes all buffered file descriptor related data changes to be flushed to the underlying block device. When a file is modified on a modern operating system the changes are generally not written to the disk immediately but rather those changes are buffered in memory for reasons of performance, calling `fsync()` causes any in-memory changes to be written to disk.

generation
An internal counter which updates for each *transaction*. When a *metadata* block is written (using *copy on write*), current generation is stored in the block, so that blocks which are too new (and hence possibly inconsistent) can be identified.

key
A fixed sized tuple used to identify and sort items in a *B-tree*. The key is broken up into 3 parts: *objectid*, *type*, and *offset*. The *type* field indicates how each of the other two fields should be used, and what to expect to find in the item.

item
A variable sized structure stored in B-tree leaves. Items hold different types of data depending on key type.

log tree
A b-tree that temporarily tracks ongoing metadata updates until a full transaction commit is done. It's a performance optimization of `fsync`. The log tracked in the tree are replayed if the filesystem is not unmounted cleanly.

metadata
Data about data. In btrfs, this includes all of the internal data structures of the filesystem, including directory structures, filenames, file permissions, checksums, and the location of each file's *extents*. All btrfs metadata is stored in *B-trees*.

mkfs.btrfs
The tool (from *btrfs-progs*) to create a btrfs filesystem.

offline
A filesystem which is not mounted is offline. Some tools (e.g. *btrfsck*) will only work on offline filesystems. Compare *online*.

online
A filesystem which is mounted is online. Most btrfs tools will only work on online filesystems. Compare *offline*.

orphan
A file that's still in use (opened by a running process) but all directory entries of that file have been removed.

RAID
A class of different methods for writing some additional redundant data across multiple *devices* so that if one device fails, the missing data can be reconstructed from the remaining ones. See *RAID-0*, *RAID-1*, *RAID-5*,

RAID-6, *RAID-10*, *DUP* and *single*. Traditional RAID methods operate across multiple devices of equal size, whereas btrfs's RAID implementation works inside *block groups*.

RAID-0
A form of *RAID* which provides no form of error recovery, but stripes a single copy of data across multiple devices for performance purposes. The stripe size is fixed to 64KB for now.

RAID-1
A form of *RAID* which stores two complete copies of each piece of data. Each copy is stored on a different *device*. btrfs requires a minimum of two devices to use RAID-1. This is the default for btrfs's *metadata* on more than one device.

RAID-5
A form of *RAID* which stripes a single copy of data across multiple *devices*, including one device's worth of additional parity data. Can be used to recover from a single device failure.

RAID-6
A form of *RAID* which stripes a single copy of data across multiple *devices*, including two device's worth of additional parity data. Can be used to recover from the failure of two devices.

RAID-10
A form of *RAID* which stores two complete copies of each piece of data, and also stripes each copy across multiple devices for performance.

reflink
Parameter to `cp`, allowing it to take advantage of the capabilities of *COW*-capable filesystems. Allows for files to be copied and modified, with only the modifications taking up additional storage space. May be considered as *snapshots* on a single file rather than a *subvolume*. Example: `cp --reflink file1 file2`

relocation
The process of moving block groups within the filesystem while maintaining full filesystem integrity and consistency. This functionality is underlying *balance* and *device* removing features.

scrub
An *online* filesystem checking tool. Reads all the data and metadata on the filesystem, and uses *checksums* and the duplicate copies from *RAID* storage to identify and repair any corrupt data.

seed device
A readonly device can be used as a filesystem seed or template (e.g. a CD-ROM containing an OS image). Read/write devices can be added to store modifications (using *copy on write*), changes to the writable devices are persistent across reboots. The original device remains unchanged and can be removed at any time (after Btrfs has been instructed to copy over all missing blocks). Multiple read/write file systems can be built from the same seed.

single
A "*RAID*" level in btrfs, storing a single copy of each piece of data. The default for data (as opposed to *metadata*) in btrfs. Single is also default metadata profile for non-rotational (SSD, flash) devices.

snapshot
A *subvolume* which is a *copy on write* copy of another subvolume. The two subvolumes share all of their common (unmodified) data, which means that snapshots can be used to keep the historical state of a filesystem very cheaply. After the snapshot is made, the original subvolume and the snapshot are of equal status: the original does not "own" the snapshot, and either one can be deleted without affecting the other one.

subvolume
A tree of files and directories inside a btrfs that can be mounted as if it were an independent filesystem. A subvolume is created by taking a reference on the root of another subvolume. Each btrfs filesystem has at least one subvolume, the *top-level subvolume*, which contains everything else in the filesystem. Additional subvolumes can be created and deleted with the <*code*>*btrfs*</*code*> tool. All subvolumes share the same pool of free space in the filesystem. See also *default subvolume*.

superblock
: The *block* on the disk, at a fixed known location and of fixed size, which contains pointers to the disk blocks containing all the other filesystem *metadata* structures. btrfs stores multiple copies of the superblock on each *device* in the filesystem at offsets 64 KiB, 64 MiB, 256 GiB, 1 TiB and PiB.

system array
: Cryptic name of *superblock* metadata describing how to assemble a filesystem from multiple device. Prior to mount, the command *btrfs dev scan* has to be called, or all the devices have to be specified via mount option *device=/dev/ice*.

top-level subvolume
: The *subvolume* at the very top of the filesystem. This is the only subvolume present in a newly-created btrfs filesystem, and internally has ID 5, otherwise could be referenced as 0 (eg. within the *set-default* subcommand of *btrfs*).

transaction
: A consistent set of changes. To avoid generating very large amounts of disk activity, btrfs caches changes in RAM for up to 30 seconds (sometimes more often if the filesystem is running short on space or doing a lot of *fsync*s), and then writes (commits) these changes out to disk in one go (using *copy on write* behaviour). This period of caching is called a transaction. Only one transaction is active on the filesystem at any one time.

transid
: An alternative term for *generation*.

writeback
: *Writeback* in the context of the Linux kernel can be defined as the process of writing "dirty" memory from the page cache to the disk, when certain conditions are met (timeout, number of dirty pages over a ratio).

CHAPTER

EIGHT

INSTALLATION INSTRUCTIONS

The Btrfs utility programs require the following libraries/tools to build:

- libuuid - provided by util-linux, e2fsprogs/e2fslibs or libuuid
- libblkid - block device id library
- liblzo2 - LZO data compression library
- zlib - ZLIB data compression library
- libzstd - ZSTD data compression library version >= 1.0.0

For the btrfs-convert utility:

- e2fsprogs - ext2/ext3/ext4 file system libraries, or called e2fslibs
- libreiserfscore - reiserfs file system library version >= 3.6.27

Optionally, the checksums based on cryptographic hashes can be implemented by external libraries. Builtin implementations are provided in case the library dependencies are not desired.

- libgcrypt
- libsodium
- libkcapi

Optionally, multipath device detection requires libudev and running udev daemon, as it's the only source of the path information. Static build has a fallback and does not need static version of libudev.

- libudev

For zoned device support, the system headers installed in /usr/include/linux must be 5.10 or newer.

Generating documentation:

- sphinx

Please note that the package names may differ according to the distribution. See https://btrfs.wiki.kernel.org/index.php/Btrfs_source_repositories#Dependencies .

8.1 Building from sources

To build from git sources you need to generate the configure script using the autotools:

$ ./autogen.sh

To build from the released tarballs:

$ ./configure $ make $ make install

To install the libbtrfsutil Python bindings:

$ make install_python

You may disable building some parts like documentation, btrfs-convert or backtrace support. See ./configure --help for more.

Specific CFLAGS or LDFLAGS should be set like

$ CFLAGS=… LDFLAGS=… ./configure --prefix=/usr

and not as arguments to make. You can specify additional flags to build via variables EXTRA_CFLAGS and EXTRA_LDFLAGS that get appended to the predefined values of the respective variables. There are further build tuning options documented in the Makefile.

$ make EXTRA_CFLAGS=-ggdb3

The build utilizes autotools, dependencies for generating the configure scripts are:

- autoconf, autoheader
- automake, aclocal
- pkg-config

8.2 Statically built binaries

The makefiles are ready to let you build static binaries of the utilities. This may be handy in rescue environments. Your system has to provide static version of the libraries.

$ make static $ make btrfs.static $ make btrfs-convert.static

The resulting binaries have the '.static' suffix, the intermediate object files do not conflict with the normal (dynamic) build.

8.3 All-in-one binary (busybox style)

Since version 5.2 it's possible to build a single binary that can act as other standalone tools, based on the file name:

$ make btrfs.box $ mv btrfs.box btrfs $ ln -s btrfs mkfs.btrfs

The list of built-ins can be obtained by

$ btrfs help --box

The basic set will always contain: mkfs.btrfs, btrfs-image, btrfs-convert.

8.4 32bit build on 64bit host

The combination of 32bit build on 64bit host could work but depends on the libraries that must provide the 32bit versions, or even 32bit static versions. This is fairly uncommon on contemporary distributions and building 32bit versions on a 32bit host is recommended.

References: * https://btrfs.wiki.kernel.org

CHAPTER

NINE

COMMON LINUX FEATURES

The Linux operating system implements a POSIX standard interfaces and API with additional interfaces. Many of them have become common in other filesystems. The ones listed below have been added relatively recently and are considered interesting for users:

birth/origin inode time
: a timestamp associated with an inode of when it was created, cannot be changed and requires the *statx* syscall to be read

statx
: an extended version of the *stat* syscall that provides extensible interface to read more information that are not available in original *stat*

fallocate modes
: the *fallocate* syscall allows to manipulate file extents like punching holes, preallocation or zeroing a range

FIEMAP
: an ioctl that enumerates file extents, related tool is `filefrag`

filesystem label
: another filesystem identification, could be used for mount or for better recognition, can be set or read by an ioctl or by command `btrfs filesystem label`

O_TMPFILE
: mode of open() syscall that creates a file with no associated directory entry, which makes it impossible to be seen by other processes and is thus safe to be used as a temporary file (https://lwn.net/Articles/619146/)

xattr, acl
: extended attributes (xattr) is a list of *key*=*value* pairs associated with a file, usually storing additional metadata related to security, access control list in particular (ACL) or properties (`btrfs property`)

cross-rename
: mode of *renameat2* syscall that can atomically swap 2 directory entries (files/directories/subvolumes)

9.1 File attributes, XFLAGS

The btrfs filesystem supports setting file attributes or flags. Note there are old and new interfaces, with confusing names. The following list should clarify that:

- *attributes*: `chattr(1)` or `lsattr(1)` utilities (the ioctls are FS_IOC_GETFLAGS and FS_IOC_SETFLAGS), due to the ioctl names the attributes are also called flags
- *xflags*: to distinguish from the previous, it's extended flags, with tunable bits similar to the attributes but extensible and new bits will be added in the future (the ioctls are FS_IOC_FSGETXATTR and FS_IOC_FSSETXATTR

but they are not related to extended attributes that are also called xattrs), there's no standard tool to change the bits, there's support in `xfs_io(8)` as command **xfs_io -c chattr**

9.1.1 Attributes

a

append only, new writes are always written at the end of the file

A

no atime updates

c

compress data, all data written after this attribute is set will be compressed. Please note that compression is also affected by the mount options or the parent directory attributes.

When set on a directory, all newly created files will inherit this attribute. This attribute cannot be set with 'm' at the same time.

C

no copy-on-write, file data modifications are done in-place

When set on a directory, all newly created files will inherit this attribute.

Note: Due to implementation limitations, this flag can be set/unset only on empty files.

---

d

no dump, makes sense with 3rd party tools like `dump(8)`, on BTRFS the attribute can be set/unset but no other special handling is done

D

synchronous directory updates, for more details search `open(2)` for *O_SYNC* and *O_DSYNC*

i

immutable, no file data and metadata changes allowed even to the root user as long as this attribute is set (obviously the exception is unsetting the attribute)

m

no compression, permanently turn off compression on the given file. Any compression mount options will not affect this file. (`chattr` support added in 1.46.2)

When set on a directory, all newly created files will inherit this attribute. This attribute cannot be set with *c* at the same time.

S

synchronous updates, for more details search `open(2)` for *O_SYNC* and *O_DSYNC*

No other attributes are supported. For the complete list please refer to the `chattr(1)` manual page.

9.1.2 XFLAGS

There's overlap of letters assigned to the bits with the attributes, this list refers to what `xfs_io(8)` provides:

i
: *immutable*, same as the attribute

a
: *append only*, same as the attribute

s
: *synchronous updates*, same as the attribute *S*

A
: *no atime updates*, same as the attribute

d
: *no dump*, same as the attribute

CHAPTER

TEN

CUSTOM IOCTLS

Filesystems are usually extended by custom ioctls beyond the standard system call interface to let user applications access the advanced features. They're low level and the following list gives only an overview of the capabilities or a command if available:

- reverse lookup, from file offset to inode, `btrfs inspect-internal logical-resolve`
- resolve inode number to list of name, `btrfs inspect-internal inode-resolve`
- tree search, given a key range and tree id, lookup and return all b-tree items found in that range, basically all metadata at your hand but you need to know what to do with them
- informative, about devices, space allocation or the whole filesystem, many of which is also exported in `/sys/fs/btrfs`
- query/set a subset of features on a mounted filesystem

CHAPTER

ELEVEN

AUTO-REPAIR ON READ

Data or metadata that are found to be damaged (eg. because the checksum does not match) at the time they're read from the device can be salvaged in case the filesystem has another valid copy when using block group profile with redundancy (DUP, RAID1, RAID5/6). The correct data are returned to the user application and the damaged copy is replaced by it.

CHAPTER

TWELVE

BALANCE

The primary purpose of the balance feature is to spread block groups across all devices so they match constraints defined by the respective profiles. See `mkfs.btrfs(8)` section *PROFILES* for more details. The scope of the balancing process can be further tuned by use of filters that can select the block groups to process. Balance works only on a mounted filesystem. Extent sharing is preserved and reflinks are not broken. Files are not defragmented nor recompressed, file extents are preserved but the physical location on devices will change.

The balance operation is cancellable by the user. The on-disk state of the filesystem is always consistent so an unexpected interruption (eg. system crash, reboot) does not corrupt the filesystem. The progress of the balance operation is temporarily stored as an internal state and will be resumed upon mount, unless the mount option *skip_balance* is specified.

Warning: Running balance without filters will take a lot of time as it basically move data/metadata from the whole filesystem and needs to update all block pointers.

The filters can be used to perform following actions:

- convert block group profiles (filter *convert*)
- make block group usage more compact (filter *usage*)
- perform actions only on a given device (filters *devid*, *drange*)

The filters can be applied to a combination of block group types (data, metadata, system). Note that changing only the *system* type needs the force option. Otherwise *system* gets automatically converted whenever *metadata* profile is converted.

When metadata redundancy is reduced (eg. from RAID1 to single) the force option is also required and it is noted in system log.

Note: The balance operation needs enough work space, ie. space that is completely unused in the filesystem, otherwise this may lead to ENOSPC reports. See the section *ENOSPC* for more details.

12.1 Compatibility

Note: The balance subcommand also exists under the **btrfs filesystem** namespace. This still works for backward compatibility but is deprecated and should not be used any more.

Note: A short syntax **btrfs balance <path>** works due to backward compatibility but is deprecated and should not be used any more. Use **btrfs balance start** command instead.

12.2 Performance implications

Balancing operations are very IO intensive and can also be quite CPU intensive, impacting other ongoing filesystem operations. Typically large amounts of data are copied from one location to another, with corresponding metadata updates.

Depending upon the block group layout, it can also be seek heavy. Performance on rotational devices is noticeably worse compared to SSDs or fast arrays.

12.3 Filters

From kernel 3.3 onwards, btrfs balance can limit its action to a subset of the whole filesystem, and can be used to change the replication configuration (e.g. moving data from single to RAID1). This functionality is accessed through the *-d*, *-m* or *-s* options to btrfs balance start, which filter on data, metadata and system blocks respectively.

A filter has the following structure: *type[=params][,type=...]*

The available types are:

profiles=<profiles>
: Balances only block groups with the given profiles. Parameters are a list of profile names separated by "|" (pipe).

usage=<percent>, usage=<range>
: Balances only block groups with usage under the given percentage. The value of 0 is allowed and will clean up completely unused block groups, this should not require any new work space allocated. You may want to use *usage=0* in case balance is returning ENOSPC and your filesystem is not too full.

  The argument may be a single value or a range. The single value *N* means *at most N percent used*, equivalent to *..N* range syntax. Kernels prior to 4.4 accept only the single value format. The minimum range boundary is inclusive, maximum is exclusive.

devid=<id>
: Balances only block groups which have at least one chunk on the given device. To list devices with ids use **btrfs filesystem show**.

drange=<range>
: Balance only block groups which overlap with the given byte range on any device. Use in conjunction with *devid* to filter on a specific device. The parameter is a range specified as *start..end*.

vrange=<range>
: Balance only block groups which overlap with the given byte range in the filesystem's internal virtual address space. This is the address space that most reports from btrfs in the kernel log use. The parameter is a range specified as *start..end*.

convert=<profile>
Convert each selected block group to the given profile name identified by parameters.

Note: Starting with kernel 4.5, the *data* chunks can be converted to/from the *DUP* profile on a single device.

Note: Starting with kernel 4.6, all profiles can be converted to/from *DUP* on multi-device filesystems.

limit=<number>, limit=<range>
Process only given number of chunks, after all filters are applied. This can be used to specifically target a chunk in connection with other filters (*drange*, *vrange*) or just simply limit the amount of work done by a single balance run.

The argument may be a single value or a range. The single value *N* means *at most N chunks*, equivalent to *..N* range syntax. Kernels prior to 4.4 accept only the single value format. The range minimum and maximum are inclusive.

stripes=<range>
Balance only block groups which have the given number of stripes. The parameter is a range specified as *start..end*. Makes sense for block group profiles that utilize striping, ie. RAID0/10/5/6. The range minimum and maximum are inclusive.

soft
Takes no parameters. Only has meaning when converting between profiles. When doing convert from one profile to another and soft mode is on, chunks that already have the target profile are left untouched. This is useful e.g. when half of the filesystem was converted earlier but got cancelled.

The soft mode switch is (like every other filter) per-type. For example, this means that we can convert metadata chunks the "hard" way while converting data chunks selectively with soft switch.

Profile names, used in *profiles* and *convert* are one of: *raid0*, *raid1*, *raid1c3*, *raid1c4*, *raid10*, *raid5*, *raid6*, *dup*, *single*. The mixed data/metadata profiles can be converted in the same way, but it's conversion between mixed and non-mixed is not implemented. For the constraints of the profiles please refer to `mkfs.btrfs(8)`, section *PROFILES*.

CHAPTER

THIRTEEN

COMPRESSION

Btrfs supports transparent file compression. There are three algorithms available: ZLIB, LZO and ZSTD (since v4.14), with various levels. The compression happens on the level of file extents and the algorithm is selected by file property, mount option or by a defrag command. You can have a single btrfs mount point that has some files that are uncompressed, some that are compressed with LZO, some with ZLIB, for instance (though you may not want it that way, it is supported).

Once the compression is set, all newly written data will be compressed, ie. existing data are untouched. Data are split into smaller chunks (128KiB) before compression to make random rewrites possible without a high performance hit. Due to the increased number of extents the metadata consumption is higher. The chunks are compressed in parallel.

The algorithms can be characterized as follows regarding the speed/ratio trade-offs:

ZLIB

- slower, higher compression ratio
- levels: 1 to 9, mapped directly, default level is 3
- good backward compatibility

LZO

- faster compression and decompression than zlib, worse compression ratio, designed to be fast
- no levels
- good backward compatibility

ZSTD

- compression comparable to zlib with higher compression/decompression speeds and different ratio
- levels: 1 to 15, mapped directly (higher levels are not available)
- since 4.14, levels since 5.1

The differences depend on the actual data set and cannot be expressed by a single number or recommendation. Higher levels consume more CPU time and may not bring a significant improvement, lower levels are close to real time.

13.1 How to enable compression

Typically the compression can be enabled on the whole filesystem, specified for the mount point. Note that the compression mount options are shared among all mounts of the same filesystem, either bind mounts or subvolume mounts. Please refer to section *MOUNT OPTIONS*.

```
$ mount -o compress=zstd /dev/sdx /mnt
```

This will enable the `zstd` algorithm on the default level (which is 3). The level can be specified manually too like `zstd:3`. Higher levels compress better at the cost of time. This in turn may cause increased write latency, low levels are suitable for real-time compression and on reasonably fast CPU don't cause noticeable performance drops.

```
$ btrfs filesystem defrag -czstd file
```

The command above will start defragmentation of the whole *file* and apply the compression, regardless of the mount option. (Note: specifying level is not yet implemented). The compression algorithm is not persistent and applies only to the defragmentation command, for any other writes other compression settings apply.

Persistent settings on a per-file basis can be set in two ways:

```
$ chattr +c file
$ btrfs property set file compression zstd
```

The first command is using legacy interface of file attributes inherited from ext2 filesystem and is not flexible, so by default the *zlib* compression is set. The other command sets a property on the file with the given algorithm. (Note: setting level that way is not yet implemented.)

13.2 Compression levels

The level support of ZLIB has been added in v4.14, LZO does not support levels (the kernel implementation provides only one), ZSTD level support has been added in v5.1.

There are 9 levels of ZLIB supported (1 to 9), mapping 1:1 from the mount option to the algorithm defined level. The default is level 3, which provides the reasonably good compression ratio and is still reasonably fast. The difference in compression gain of levels 7, 8 and 9 is comparable but the higher levels take longer.

The ZSTD support includes levels 1 to 15, a subset of full range of what ZSTD provides. Levels 1-3 are real-time, 4-8 slower with improved compression and 9-15 try even harder though the resulting size may not be significantly improved.

Level 0 always maps to the default. The compression level does not affect compatibility.

13.3 Incompressible data

Files with already compressed data or with data that won't compress well with the CPU and memory constraints of the kernel implementations are using a simple decision logic. If the first portion of data being compressed is not smaller than the original, the compression of the file is disabled -- unless the filesystem is mounted with *compress-force*. In that case compression will always be attempted on the file only to be later discarded. This is not optimal and subject to optimizations and further development.

If a file is identified as incompressible, a flag is set (*NOCOMPRESS*) and it's sticky. On that file compression won't be performed unless forced. The flag can be also set by **chattr +m** (since e2fsprogs 1.46.2) or by properties with value *no* or *none*. Empty value will reset it to the default that's currently applicable on the mounted filesystem.

There are two ways to detect incompressible data:

- actual compression attempt - data are compressed, if the result is not smaller, it's discarded, so this depends on the algorithm and level
- pre-compression heuristics - a quick statistical evaluation on the data is performed and based on the result either compression is performed or skipped, the NOCOMPRESS bit is not set just by the heuristic, only if the compression algorithm does not make an improvement

```
$ lsattr file
---------------------m file
```

Using the forcing compression is not recommended, the heuristics are supposed to decide that and compression algorithms internally detect incompressible data too.

13.4 Pre-compression heuristics

The heuristics aim to do a few quick statistical tests on the compressed data in order to avoid probably costly compression that would turn out to be inefficient. Compression algorithms could have internal detection of incompressible data too but this leads to more overhead as the compression is done in another thread and has to write the data anyway. The heuristic is read-only and can utilize cached memory.

The tests performed based on the following: data sampling, long repeated pattern detection, byte frequency, Shannon entropy.

13.5 Compatibility

Compression is done using the COW mechanism so it's incompatible with *nodatacow*. Direct IO works on compressed files but will fall back to buffered writes and leads to recompression. Currently *nodatasum* and compression don't work together.

The compression algorithms have been added over time so the version compatibility should be also considered, together with other tools that may access the compressed data like bootloaders.

CHAPTER

FOURTEEN

CHECKSUMMING

Data and metadata are checksummed by default, the checksum is calculated before write and verifed after reading the blocks from devices. The whole metadata block has a checksum stored inline in the b-tree node header, each data block has a detached checksum stored in the checksum tree.

There are several checksum algorithms supported. The default and backward compatible is *crc32c*. Since kernel 5.5 there are three more with different characteristics and trade-offs regarding speed and strength. The following list may help you to decide which one to select.

CRC32C (32bit digest)
: default, best backward compatibility, very fast, modern CPUs have instruction-level support, not collision-resistant but still good error detection capabilities

XXHASH (64bit digest)
: can be used as CRC32C successor, very fast, optimized for modern CPUs utilizing instruction pipelining, good collision resistance and error detection

SHA256 (256bit digest)::
: a cryptographic-strength hash, relatively slow but with possible CPU instruction acceleration or specialized hardware cards, FIPS certified and in wide use

BLAKE2b (256bit digest)
: a cryptographic-strength hash, relatively fast with possible CPU acceleration using SIMD extensions, not standardized but based on BLAKE which was a SHA3 finalist, in wide use, the algorithm used is BLAKE2b-256 that's optimized for 64bit platforms

The *digest size* affects overall size of data block checksums stored in the filesystem. The metadata blocks have a fixed area up to 256 bits (32 bytes), so there's no increase. Each data block has a separate checksum stored, with additional overhead of the b-tree leaves.

Approximate relative performance of the algorithms, measured against CRC32C using reference software implementations on a 3.5GHz intel CPU:

Digest	Cycles/4KiB	Ratio	Implementation
CRC32C	1700	1.00	CPU instruction
XXHASH	2500	1.44	reference impl.
SHA256	105000	61	reference impl.
SHA256	36000	21	libgcrypt/AVX2
SHA256	63000	37	libsodium/AVX2
BLAKE2b	22000	13	reference impl.
BLAKE2b	19000	11	libgcrypt/AVX2
BLAKE2b	19000	11	libsodium/AVX2

Many kernels are configured with SHA256 as built-in and not as a module. The accelerated versions are however provided by the modules and must be loaded explicitly (**modprobe sha256**) before mounting the filesystem to make

use of them. You can check in */sys/fs/btrfs/FSID/checksum* which one is used. If you see *sha256-generic*, then you may want to unmount and mount the filesystem again, changing that on a mounted filesystem is not possible. Check the file */proc/crypto*, when the implementation is built-in, you'd find

```
name         : sha256
driver       : sha256-generic
module       : kernel
priority     : 100
...
```

while accelerated implementation is e.g.

```
name         : sha256
driver       : sha256-avx2
module       : sha256_ssse3
priority     : 170
...
```

CHAPTER

FIFTEEN

CONVERT

The **btrfs-convert** tool can be used to convert existing source filesystem image to a btrfs filesystem in-place. The original filesystem image is accessible in subvolume named like *ext2_saved* as file *image*.

Supported filesystems:

- ext2, ext3, ext4 -- original feature, always built in
- reiserfs -- since version 4.13, optionally built, requires libreiserfscore 3.6.27
- ntfs -- external tool https://github.com/maharmstone/ntfs2btrfs

The list of supported source filesystem by a given binary is listed at the end of help (option *--help*).

Warning: If you are going to perform rollback to the original filesystem, you should not execute **btrfs balance** command on the converted filesystem. This will change the extent layout and make **btrfs-convert** unable to rollback.

The conversion utilizes free space of the original filesystem. The exact estimate of the required space cannot be foretold. The final btrfs metadata might occupy several gigabytes on a hundreds-gigabyte filesystem.

If the ability to rollback is no longer important, the it is recommended to perform a few more steps to transition the btrfs filesystem to a more compact layout. This is because the conversion inherits the original data blocks' fragmentation, and also because the metadata blocks are bound to the original free space layout.

Due to different constraints, it is only possible to convert filesystems that have a supported data block size (ie. the same that would be valid for **mkfs.btrfs**). This is typically the system page size (4KiB on x86_64 machines).

BEFORE YOU START

The source filesystem must be clean, eg. no journal to replay or no repairs needed. The respective **fsck** utility must be run on the source filessytem prior to conversion. Please refer to the manual pages in case you encounter problems.

For ext2/3/4:

```
# e2fsck -fvy /dev/sdx
```

For reiserfs:

```
# reiserfsck -fy /dev/sdx
```

Skipping that step could lead to incorrect results on the target filesystem, but it may work.

REMOVE THE ORIGINAL FILESYSTEM METADATA

By removing the subvolume named like *ext2_saved* or *reiserfs_saved*, all metadata of the original filesystem will be removed:

```
# btrfs subvolume delete /mnt/ext2_saved
```

At this point it is not possible to do a rollback. The filesystem is usable but may be impacted by the fragmentation inherited from the original filesystem.

MAKE FILE DATA MORE CONTIGUOUS

An optional but recommended step is to run defragmentation on the entire filesystem. This will attempt to make file extents more contiguous.

```
# btrfs filesystem defrag -v -r -f -t 32M /mnt/btrfs
```

Verbose recursive defragmentation (*-v*, *-r*), flush data per-file (*-f*) with target extent size 32MiB (*-t*).

ATTEMPT TO MAKE BTRFS METADATA MORE COMPACT

Optional but recommended step.

The metadata block groups after conversion may be smaller than the default size (256MiB or 1GiB). Running a balance will attempt to merge the block groups. This depends on the free space layout (and fragmentation) and may fail due to lack of enough work space. This is a soft error leaving the filesystem usable but the block group layout may remain unchanged.

Note that balance operation takes a lot of time, please see also `btrfs-balance(8)`.

```
# btrfs balance start -m /mnt/btrfs
```

CHAPTER

SIXTEEN

DEDUPLICATION

Going by the definition in the context of filesystems, it's a process of looking up identical data blocks tracked separately and creating a shared logical link while removing one of the copies of the data blocks. This leads to data space savings while it increases metadata consumption.

There are two main deduplication types:

- **in-band** *(sometimes also called on-line)* -- all newly written data are considered for deduplication before writing
- **out-of-band** *(sometimes also called offline)* -- data for deduplication have to be actively looked for and deduplicated by the user application

Both have their pros and cons. BTRFS implements **only out-of-band** type.

BTRFS provides the basic building blocks for deduplication allowing other tools to choose the strategy and scope of the deduplication. There are multiple tools that take different approaches to deduplication, offer additional features or make trade-offs. The following table lists tools that are known to be up-to-date, maintained and widely used.

Name	File based	Block based	Incremental
BEES	No	Yes	Yes
duperemove	Yes	No	Yes

16.1 File based deduplication

The tool takes a list of files and tries to find duplicates among data only from that files. This is suitable eg. for files that originated from the same base image, source of a reflinked file. Optionally the tools could track a database of hashes and allow to deduplicate blocks from more files, or use that for repeated runs and update the database incrementally.

16.2 Block based deduplication

The tool typically scans the filesystem and builds a database of file block hashes, then finds candidate files and deduplicates the ranges. The hash database is kept as an ordinary file and can be scaled according to the needs.

As the file changes, the hash database may get out of sync and the scan has to be done repeatedly.

16.3 Safety of block comparison

The deduplication inside the filesystem is implemented as an `ioctl` that takes a source file, destination file and the range. The blocks from both files are compared for exact match before merging to the same range (ie. there's no hash based comparison). Pages representing the extents in memory are locked prior to deduplication and prevent concurrent modification by buffered writes or mmaped writes.

16.4 Limitations, compatibility

Files that are subject do deduplication must have the same status regarding COW, ie. both regular COW files with checksums, or both NOCOW, or files that are COW but don't have checksums (NODATASUM attribute is set).

If the deduplication is in progress on any file in the filesystem, the *send* operation cannot be started as it relies on the extent layout being unchanged.

CHAPTER

SEVENTEEN

DEFRAGMENTATION

Defragmentation of files is supposed to make the layout of the file extents to be more linear or at least coalesce the file extents into larger ones that can be stored on the device more efficiently. The reason there's a need for defragmentation stems from the COW design that BTRFS is built on and is inherent. The fragmentation is caused by rewrites of the same file data in-place, that has to be handled by creating a new copy that may lie on a distant location on the physical device. Fragmentation is the worst problem on rotational hard disks due to the delay caused by moving the drive heads to the distant location. With the modern seek-less devices it's not a problem though it may still make sense because of reduced size of the metadata that's needed to track the scattered extents.

File data that are in use can be safely defragmented because the whole process happens inside the page cache, that is the central point caching the file data and takes care of synchronization. Once a filesystem sync or flush is started (either manually or automatically) all the dirty data get written to the devices. This however reduces the chances to find optimal layout as the writes happen together with other data and the result depends on the remaining free space layout and fragmentation.

Warning: Defragmentation does not preserve extent sharing, eg. files created by **cp --reflink** or existing on multiple snapshots. Due to that the data space consumption may increase.

Defragmentation can be started together with compression on the given range, and takes precedence over per-file compression property or mount options.

CHAPTER

EIGHTEEN

INLINE FILES

Files up to some size can be stored in the metadata section ("inline" in the b-tree nodes), ie. no separate blocks for the extents. The default limit is 2048 bytes and can be configured by mount option `max_inline`. The data of inlined files can be also compressed as long as they fit into the b-tree nodes.

If the filesystem has been created with different data and metadata profiles, namely with different level of integrity, this also affects the inlined files. It can be completely disabled by mounting with `max_inline=0`. The upper limit is either the size of b-tree node or the page size of the host.

An inline file can be identified by enumerating the extents, eg. by the tool `filefrag`:

```
$ filefrag -v inlinefile
Filesystem type is: 9123683e
File size of inlinefile is 463 (1 block of 4096 bytes)
 ext:     logical_offset:        physical_offset: length:   expected: flags:
   0:        0..    4095:          0..      4095:   4096:             last,not_aligned,
↪inline,eof
```

In the above example, the file is not compressed, otherwise it would have the *encoded* flag. The inline files have no limitations and behave like regular files with respect to copying, renaming, reflink, truncate etc.

CHAPTER

NINETEEN

QUOTA GROUPS

The concept of quota has a long-standing tradition in the Unix world. Ever since computers allow multiple users to work simultaneously in one filesystem, there is the need to prevent one user from using up the entire space. Every user should get his fair share of the available resources.

In case of files, the solution is quite straightforward. Each file has an *owner* recorded along with it, and it has a size. Traditional quota just restricts the total size of all files that are owned by a user. The concept is quite flexible: if a user hits his quota limit, the administrator can raise it on the fly.

On the other hand, the traditional approach has only a poor solution to restrict directories. At installation time, the harddisk can be partitioned so that every directory (eg. /usr, /var/, ...) that needs a limit gets its own partition. The obvious problem is that those limits cannot be changed without a reinstallation. The btrfs subvolume feature builds a bridge. Subvolumes correspond in many ways to partitions, as every subvolume looks like its own filesystem. With subvolume quota, it is now possible to restrict each subvolume like a partition, but keep the flexibility of quota. The space for each subvolume can be expanded or restricted on the fly.

As subvolumes are the basis for snapshots, interesting questions arise as to how to account used space in the presence of snapshots. If you have a file shared between a subvolume and a snapshot, whom to account the file to? The creator? Both? What if the file gets modified in the snapshot, should only these changes be accounted to it? But wait, both the snapshot and the subvolume belong to the same user home. I just want to limit the total space used by both! But somebody else might not want to charge the snapshots to the users.

Btrfs subvolume quota solves these problems by introducing groups of subvolumes and let the user put limits on them. It is even possible to have groups of groups. In the following, we refer to them as *qgroups*.

Each qgroup primarily tracks two numbers, the amount of total referenced space and the amount of exclusively referenced space.

referenced
: space is the amount of data that can be reached from any of the subvolumes contained in the qgroup, while

exclusive
: is the amount of data where all references to this data can be reached from within this qgroup.

19.1 Subvolume quota groups

The basic notion of the Subvolume Quota feature is the quota group, short qgroup. Qgroups are notated as *level/id*, eg. the qgroup 3/2 is a qgroup of level 3. For level 0, the leading '0/' can be omitted. Qgroups of level 0 get created automatically when a subvolume/snapshot gets created. The ID of the qgroup corresponds to the ID of the subvolume, so 0/5 is the qgroup for the root subvolume. For the `btrfs qgroup` command, the path to the subvolume can also be used instead of *0/ID*. For all higher levels, the ID can be chosen freely.

Each qgroup can contain a set of lower level qgroups, thus creating a hierarchy of qgroups. Figure 1 shows an example qgroup tree.

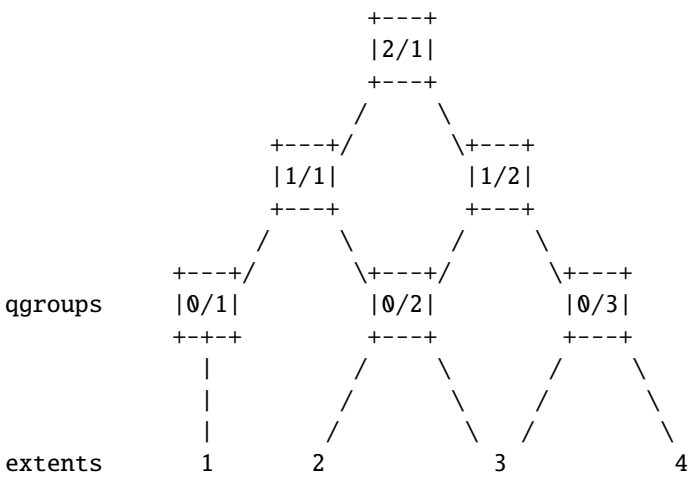

Figure 1: Sample qgroup hierarchy

At the bottom, some extents are depicted showing which qgroups reference which extents. It is important to understand the notion of *referenced* vs *exclusive*. In the example, qgroup 0/2 references extents 2 and 3, while 1/2 references extents 2-4, 2/1 references all extents.

On the other hand, extent 1 is exclusive to 0/1, extent 2 is exclusive to 0/2, while extent 3 is neither exclusive to 0/2 nor to 0/3. But because both references can be reached from 1/2, extent 3 is exclusive to 1/2. All extents are exclusive to 2/1.

So exclusive does not mean there is no other way to reach the extent, but it does mean that if you delete all subvolumes contained in a qgroup, the extent will get deleted.

Exclusive of a qgroup conveys the useful information how much space will be freed in case all subvolumes of the qgroup get deleted.

All data extents are accounted this way. Metadata that belongs to a specific subvolume (i.e. its filesystem tree) is also accounted. Checksums and extent allocation information are not accounted.

In turn, the referenced count of a qgroup can be limited. All writes beyond this limit will lead to a 'Quota Exceeded' error.

19.2 Inheritance

Things get a bit more complicated when new subvolumes or snapshots are created. The case of (empty) subvolumes is still quite easy. If a subvolume should be part of a qgroup, it has to be added to the qgroup at creation time. To add it at a later time, it would be necessary to at least rescan the full subvolume for a proper accounting.

Creation of a snapshot is the hard case. Obviously, the snapshot will reference the exact amount of space as its source, and both source and destination now have an exclusive count of 0 (the filesystem nodesize to be precise, as the roots of the trees are not shared). But what about qgroups of higher levels? If the qgroup contains both the source and the destination, nothing changes. If the qgroup contains only the source, it might lose some exclusive.

But how much? The tempting answer is, subtract all exclusive of the source from the qgroup, but that is wrong, or at least not enough. There could have been an extent that is referenced from the source and another subvolume from that qgroup. This extent would have been exclusive to the qgroup, but not to the source subvolume. With the creation of the snapshot, the qgroup would also lose this extent from its exclusive set.

So how can this problem be solved? In the instant the snapshot gets created, we already have to know the correct exclusive count. We need to have a second qgroup that contains all the subvolumes as the first qgroup, except the

subvolume we want to snapshot. The moment we create the snapshot, the exclusive count from the second qgroup needs to be copied to the first qgroup, as it represents the correct value. The second qgroup is called a tracking qgroup. It is only there in case a snapshot is needed.

19.3 Use cases

Below are some use cases that do not mean to be extensive. You can find your own way how to integrate qgroups.

19.3.1 Single-user machine

`Replacement for partitions`

The simplest use case is to use qgroups as simple replacement for partitions. Btrfs takes the disk as a whole, and /, /usr, /var, etc. are created as subvolumes. As each subvolume gets it own qgroup automatically, they can simply be restricted. No hierarchy is needed for that.

`Track usage of snapshots`

When a snapshot is taken, a qgroup for it will automatically be created with the correct values. 'Referenced' will show how much is in it, possibly shared with other subvolumes. 'Exclusive' will be the amount of space that gets freed when the subvolume is deleted.

19.3.2 Multi-user machine

`Restricting homes`

When you have several users on a machine, with home directories probably under /home, you might want to restrict /home as a whole, while restricting every user to an individual limit as well. This is easily accomplished by creating a qgroup for /home , eg. 1/1, and assigning all user subvolumes to it. Restricting this qgroup will limit /home, while every user subvolume can get its own (lower) limit.

`Accounting snapshots to the user`

Let's say the user is allowed to create snapshots via some mechanism. It would only be fair to account space used by the snapshots to the user. This does not mean the user doubles his usage as soon as he takes a snapshot. Of course, files that are present in his home and the snapshot should only be accounted once. This can be accomplished by creating a qgroup for each user, say '1/UID'. The user home and all snapshots are assigned to this qgroup. Limiting it will extend the limit to all snapshots, counting files only once. To limit /home as a whole, a higher level group 2/1 replacing 1/1 from the previous example is needed, with all user qgroups assigned to it.

`Do not account snapshots`

On the other hand, when the snapshots get created automatically, the user has no chance to control them, so the space used by them should not be accounted to him. This is already the case when creating snapshots in the example from the previous section.

`Snapshots for backup purposes`

This scenario is a mixture of the previous two. The user can create snapshots, but some snapshots for backup purposes are being created by the system. The user's snapshots should be accounted to the user, not the system. The solution is similar to the one from section 'Accounting snapshots to the user', but do not assign system snapshots to user's qgroup.

CHAPTER

TWENTY

REFLINK

Reflink is a type of shallow copy of file data that shares the blocks but otherwise the files are independent and any change to the file will not affect the other. This builds on the underlying COW mechanism. A reflink will effectively create only a separate metadata pointing to the shared blocks which is typically much faster than a deep copy of all blocks.

The reflink is typically meant for whole files but a partial file range can be also copied, though there are no ready-made tools for that.

```
cp --reflink=always source target
```

There are some constraints:

- cross-filesystem reflink is not possible, there's nothing in common between so the block sharing can't work
- reflink crossing two mount points of the same filesystem does not work due to an artificial limitation in VFS (this may change in the future)
- reflink requires source and target file that have the same status regarding NOCOW and checksums, for example if the source file is NOCOW (once created with the chattr +C attribute) then the above command won't work unless the target file is pre-created with the +C attribute as well, or the NOCOW attribute is inherited from the parent directory (chattr +C on the directory) or if the whole filesystem is mounted with *-o nodatacow* that would create the NOCOW files by default

CHAPTER

TWENTYONE

RESIZE

A BTRFS mounted filesystem can be resized after creation, grown or shrunk. On a multi device filesystem the space occupied on each device can be resized independently. Data that reside in the area that would be out of the new size are relocated to the remaining space below the limit, so this constrains the minimum size to which a filesystem can be shrunk.

Growing a filesystem is quick as it only needs to take note of the available space, while shrinking a filesystem needs to relocate potentially lots of data and this is IO intense. It is possible to shrink a filesystem in smaller steps.

CHAPTER

TWENTYTWO

SCRUB

Scrub is a pass over all filesystem data and metadata and verifying the checksums. If a valid copy is available (replicated block group profiles) then the damaged one is repaired. All copies of the replicated profiles are validated.

Note: Scrub is not a filesystem checker (fsck) and does not verify nor repair structural damage in the filesystem. It really only checks checksums of data and tree blocks, it doesn't ensure the content of tree blocks is valid and consistent. There's some validation performed when metadata blocks are read from disk but it's not extensive and cannot substitute full *btrfs check* run.

The user is supposed to run it manually or via a periodic system service. The recommended period is a month but could be less. The estimated device bandwidth utilization is about 80% on an idle filesystem. The IO priority class is by default *idle* so background scrub should not significantly interfere with normal filesystem operation. The IO scheduler set for the device(s) might not support the priority classes though.

The scrubbing status is recorded in */var/lib/btrfs/* in textual files named *scrub.status.UUID* for a filesystem identified by the given UUID. (Progress state is communicated through a named pipe in file *scrub.progress.UUID* in the same directory.) The status file is updated every 5 seconds. A resumed scrub will continue from the last saved position.

Scrub can be started only on a mounted filesystem, though it's possible to scrub only a selected device. See **btrfs scrub start** for more.

CHAPTER

TWENTYTHREE

SEEDING DEVICE

The COW mechanism and multiple devices under one hood enable an interesting concept, called a seeding device: extending a read-only filesystem on a device with another device that captures all writes. For example imagine an immutable golden image of an operating system enhanced with another device that allows to use the data from the golden image and normal operation. This idea originated on CD-ROMs with base OS and allowing to use them for live systems, but this became obsolete. There are technologies providing similar functionality, like *unionmount*, *overlayfs* or *qcow2* image snapshot.

The seeding device starts as a normal filesystem, once the contents is ready, **btrfstune -S 1** is used to flag it as a seeding device. Mounting such device will not allow any writes, except adding a new device by **btrfs device add**. Then the filesystem can be remounted as read-write.

Given that the filesystem on the seeding device is always recognized as read-only, it can be used to seed multiple filesystems from one device at the same time. The UUID that is normally attached to a device is automatically changed to a random UUID on each mount.

Once the seeding device is mounted, it needs the writable device. After adding it, something like **remount -o remount,rw /path** makes the filesystem at */path* ready for use. The simplest use case is to throw away all changes by unmounting the filesystem when convenient.

Alternatively, deleting the seeding device from the filesystem can turn it into a normal filesystem, provided that the writable device can also contain all the data from the seeding device.

The seeding device flag can be cleared again by **btrfstune -f -S 0**, eg. allowing to update with newer data but please note that this will invalidate all existing filesystems that use this particular seeding device. This works for some use cases, not for others, and the forcing flag to the command is mandatory to avoid accidental mistakes.

Example how to create and use one seeding device:

```
# mkfs.btrfs /dev/sda
# mount /dev/sda /mnt/mnt1
... fill mnt1 with data
# umount /mnt/mnt1

# btrfstune -S 1 /dev/sda

# mount /dev/sda /mnt/mnt1
# btrfs device add /dev/sdb /mnt/mnt1
# mount -o remount,rw /mnt/mnt1
... /mnt/mnt1 is now writable
```

Now */mnt/mnt1* can be used normally. The device */dev/sda* can be mounted again with a another writable device:

```
# mount /dev/sda /mnt/mnt2
# btrfs device add /dev/sdc /mnt/mnt2
```

(continues on next page)

(continued from previous page)

```
# mount -o remount,rw /mnt/mnt2
... /mnt/mnt2 is now writable
```

The writable device (*/dev/sdb*) can be decoupled from the seeding device and used independently:

```
# btrfs device delete /dev/sda /mnt/mnt1
```

As the contents originated in the seeding device, it's possible to turn */dev/sdb* to a seeding device again and repeat the whole process.

A few things to note:

- it's recommended to use only single device for the seeding device, it works for multiple devices but the *single* profile must be used in order to make the seeding device deletion work
- block group profiles *single* and *dup* support the use cases above
- the label is copied from the seeding device and can be changed by **btrfs filesystem label**
- each new mount of the seeding device gets a new random UUID

23.1 Chained seeding devices

Though it's not recommended and is rather an obscure and untested use case, chaining seeding devices is possible. In the first example, the writable device */dev/sdb* can be turned onto another seeding device again, depending on the unchanged seeding device */dev/sda*. Then using */dev/sdb* as the primary seeding device it can be extended with another writable device, say */dev/sdd*, and it continues as before as a simple tree structure on devices.

```
# mkfs.btrfs /dev/sda
# mount /dev/sda /mnt/mnt1
... fill mnt1 with data
# umount /mnt/mnt1

# btrfstune -S 1 /dev/sda

# mount /dev/sda /mnt/mnt1
# btrfs device add /dev/sdb /mnt/mnt1
# mount -o remount,rw /mnt/mnt1
... /mnt/mnt1 is now writable
# umount /mnt/mnt1

# btrfstune -S 1 /dev/sdb

# mount /dev/sdb /mnt/mnt1
# btrfs device add /dev/sdc /mnt
# mount -o remount,rw /mnt/mnt1
... /mnt/mnt1 is now writable
# umount /mnt/mnt1
```

As a result we have:

- *sda* is a single seeding device, with its initial contents
- *sdb* is a seeding device but requires *sda*, the contents are from the time when *sdb* is made seeding, ie. contents of *sda* with any later changes

- *sdc* last writable, can be made a seeding one the same way as was *sdb*, preserving its contents and depending on *sda* and *sdb*

As long as the seeding devices are unmodified and available, they can be used to start another branch.

CHAPTER

TWENTYFOUR

SEND/RECEIVE

Send and receive are complementary features that allow to transfer data from one filesystem to another in a streamable format. The send part traverses a given read-only subvolume and either creates a full stream representation of its data and metadata (*full mode*), or given a set of subvolumes for reference it generates a difference relative to that set (*incremental mode*).

Receive on the other hand takes the stream and reconstructs a subvolume with files and directories equivalent to the filesystem that was used to produce the stream. The result is not exactly 1:1, eg. inode numbers can be different and other unique identifiers can be different (like the subvolume UUIDs). The full mode starts with an empty subvolume, creates all the files and then turns the subvolume to read-only. At this point it could be used as a starting point for a future incremental send stream, provided it would be generated from the same source subvolume on the other filesystem.

The stream is a sequence of encoded commands that change eg. file metadata (owner, permissions, extended attributes), data extents (create, clone, truncate), whole file operations (rename, delete). The stream can be sent over network, piped directly to the receive command or saved to a file. Each command in the stream is protected by a CRC32C checksum.

CHAPTER

TWENTYFIVE

SUBPAGE SUPPORT

Subpage block size support, or just *subpage* for short, is a feature to allow using a filesystem that has different size of data block size (*sectorsize*) and the host CPU page size. For easier implementation the support was limited to the exactly same size of the block and page. On x86_64 this is typically 4KiB, but there are other architectures commonly used that make use of larger pages, like 64KiB on 64bit ARM or PowerPC. This means filesystems created with 64KiB sector size cannot be mounted on a system with 4KiB page size.

While with subpage support, systems with 64KiB page size can create (still needs "-s 4k" option for mkfs.btrfs) and mount filesystems with 4KiB sectorsize, allowing us to push 4KiB sectorsize as default sectorsize for all platforms in the near future.

25.1 Requirements, limitations

The initial subpage support has been added in v5.15, although it's still considered as experimental at the time of writing (v5.18), most features are already working without problems.

End users can mount filesystems with 4KiB sectorsize and do their usual workload, while should not notice any obvious change, as long as the initial mount succeeded (there are cases a mount will be rejected though).

The following features has some limitations for subpage:

- RAID56 support This support is already queued for v5.19 cycle. Any fs with RAID56 chunks will be rejected at mount time for now.
- Support for page size other than 64KiB The support for other page sizes (16KiB, 32KiB and more) are already queued for v5.19 cycle. Initially the subpage support is only for 64KiB support, but the design makes it pretty easy to enable support for other page sizes.
- No inline extent creation This is an artificial limit, to prevent mixed inline and regular extents.

  It's possible to create mixed inline and regular extents even with non-subpage mount for certain corner cases, it's way easier to create such mixed extents for subpage.

  Thus max_inline mount option will be sliently ignored for subpage mounts, and it always acts as "max_inline=0".
- Compression write is limited to page aligned ranges Compression write for subpage is introduced in v5.16, with the limitation that only page aligned range can be compressed. This limitation is due to how btrfs handles delayed allocation.
- No support for v1 space cache V1 space cache is considered deprecated, and we're defaulting to v2 cache in btrfs-progs already. The old v1 cache has quite some hard coded page size usage, and consider it is already deprecated, we force v2 cache for subpage.
- Slightly higher memory usage for scrub This is due to how we allocate pages for scrub, and will be fixed in the coming releases soon.

CHAPTER

TWENTYSIX

SUBVOLUMES

A BTRFS subvolume is a part of filesystem with its own independent file/directory hierarchy and inode number namespace. Subvolumes can share file extents. A snapshot is also subvolume, but with a given initial content of the original subvolume. A subvolume has always inode number 256.

Note: A subvolume in BTRFS is not like an LVM logical volume, which is block-level snapshot while BTRFS subvolumes are file extent-based.

A subvolume looks like a normal directory, with some additional operations described below. Subvolumes can be renamed or moved, nesting subvolumes is not restricted but has some implications regarding snapshotting. The numeric id (called *subvolid* or *rootid*) of the subvolume is persistent and cannot be changed.

A subvolume in BTRFS can be accessed in two ways:

- like any other directory that is accessible to the user
- like a separately mounted filesystem (options *subvol* or *subvolid*)

In the latter case the parent directory is not visible and accessible. This is similar to a bind mount, and in fact the subvolume mount does exactly that.

A freshly created filesystem is also a subvolume, called *top-level*, internally has an id 5. This subvolume cannot be removed or replaced by another subvolume. This is also the subvolume that will be mounted by default, unless the default subvolume has been changed (see `btrfs subvolume set-default`).

A snapshot is a subvolume like any other, with given initial content. By default, snapshots are created read-write. File modifications in a snapshot do not affect the files in the original subvolume.

Subvolumes can be given capacity limits, through the qgroups/quota facility, but otherwise share the single storage pool of the whole btrfs filesystem. They may even share data between themselves (through deduplication or snapshotting).

Note: A snapshot is not a backup: snapshots work by use of BTRFS' copy-on-write behaviour. A snapshot and the original it was taken from initially share all of the same data blocks. If that data is damaged in some way (cosmic rays, bad disk sector, accident with dd to the disk), then the snapshot and the original will both be damaged. Snapshots are useful to have local online "copies" of the filesystem that can be referred back to, or to implement a form of deduplication, or to fix the state of a filesystem for making a full backup without anything changing underneath it. They do not in themselves make your data any safer.

26.1 Subvolume flags

The subvolume flag currently implemented is the *ro* property (read-only status). Read-write subvolumes have that set to *false*, snapshots as *true*. In addition to that, a plain snapshot will also have last change generation and creation generation equal.

Read-only snapshots are building blocks of incremental send (see `btrfs-send(8)`) and the whole use case relies on unmodified snapshots where the relative changes are generated from. Thus, changing the subvolume flags from read-only to read-write will break the assumptions and may lead to unexpected changes in the resulting incremental stream.

A snapshot that was created by send/receive will be read-only, with different last change generation, read-only and with set *received_uuid* which identifies the subvolume on the filesystem that produced the stream. The use case relies on matching data on both sides. Changing the subvolume to read-write after it has been received requires to reset the *received_uuid*. As this is a notable change and could potentially break the incremental send use case, performing it by **btrfs property set** requires force if that is really desired by user.

Note: The safety checks have been implemented in 5.14.2, any subvolumes previously received (with a valid *received_uuid*) and read-write status may exist and could still lead to problems with send/receive. You can use **btrfs subvolume show** to identify them. Flipping the flags to read-only and back to read-write will reset the *received_uuid* manually. There may exist a convenience tool in the future.

26.2 Nested subvolumes

There are no restrictions for subvolume creation, so it's up to the user how to organize them, whether to have a flat layout (all subvolumes are direct descendants of the toplevel one), or nested.

What should be mentioned early is that a snapshotting is not recursive, so a subvolume or a snapshot is effectively a barrier and no files in the nested appear in the snapshot. Instead there's a stub subvolume (also sometimes **empty subvolume** with the same name as original subvolume, with inode number 2). This can be used intentionally but could be confusing in case of nested layouts.

26.2.1 Case study: system root layouts

There are two ways how the system root directory and subvolume layout could be organized. The interesting usecase for root is to allow rollbacks to previous version, as one atomic step. If the entire filesystem hierarchy starting in "/" is in one subvolume, taking snapshot will encompass all files. This is easy for the snapshotting part but has undesirable consequences for rollback. For example, log files would get rolled back too, or any data that are stored on the root filesystem but are not meant to be rolled back either (database files, VM images, ...).

Here we could utilize the snapshotting barrier mentioned above, each directory that stores data to be preserved accross rollbacks is it's own subvolume. This could be eg. `/var`. Further more-fine grained partitioning could be done, eg. adding separate subvolumes for `/var/log`, `/var/cache` etc.

That there are separate subvolumes requrires separate actions to take the snapshots (here it gets disconnected from the system root snapshots). This needs to be taken care of by system tools, installers together with selection of which directories are highly recommended to be separate subvolumes.

26.3 Mount options

Mount options are of two kinds, generic (that are handled by VFS layer) and specific, handled by the filesystem. The following list shows which are applicable to individual subvolume mounts, while there are more options that always affect the whole filesystem:

- generic: noatime/relatime/..., nodev, nosuid, ro, rw, dirsync
- fs-specific: compress, autodefrag, nodatacow, nodatasum

An example of whole filesystem options is eg. *space_cache*, *rescue*, *device*, *skip_balance*, etc. The exceptional options are *subvol* and *subvolid* that are actually used for mounting a given subvolume and can be specified only once for the mount.

Subvolumes belong to a single filesystem and as implemented now all share the same specific mount options, changes done by remount have immediate effect. This may change in the future.

Mounting a read-write snapshot as read-only is possible and will not chnange the *ro* property and flag of the subvolume.

The name of the mounted subvolume is stored in file `/proc/self/mounts` in the 4th column:

```
27 21 0:19 /subv1 /mnt rw,relatime - btrfs /dev/sda rw,space_cache
           ^^^^^^
```

26.4 Inode numbers

A proper subvolume has always inode number 256. If a subvolume is nested and then a snapshot is taken, then the cloned directory entry representing the subvolume becomes empty and the inode has number 2. All other files and directories in the target snapshot preserve their original inode numbers.

Note: Inode number is not a filesystem-wide unique identifier, some applications assume that. Please user pair *subvolumeid:inodenumber* for that purpose.

26.5 Performance

Subvolume creation needs to flush dirty data that belong to the subvolume, this step may take some time, otherwise once there's nothing else to do, the snapshot is instant and in the metadata it only creates a new tree root copy.

Snapshot deletion has two phases: first its directory is deleted and the subvolume is added to a list, then the list is processed one by one and the data related to the subvolume get deleted. This is usually called *cleaning* and can take some time depending on the amount of shared blocks (can be a lot of metadata updates), and the number of currently queued deleted subvolumes.

CHAPTER

TWENTYSEVEN

SWAPFILE

A swapfile is file-backed memory that the system uses to temporarily offload the RAM. It is supported since kernel 5.0. Use `swapon(8)` to activate the swapfile. There are some limitations of the implementation in BTRFS and linux swap subsystem:

- filesystem - must be only single device
- filesystem - must have only *single* data profile
- swapfile - the containing subvolume cannot be snapshotted
- swapfile - must be preallocated
- swapfile - must be nodatacow (ie. also nodatasum)
- swapfile - must not be compressed

The limitations come namely from the COW-based design and mapping layer of blocks that allows the advanced features like relocation and multi-device filesystems. However, the swap subsystem expects simpler mapping and no background changes of the file blocks once they've been attached to swap.

With active swapfiles, the following whole-filesystem operations will skip swapfile extents or may fail:

- balance - block groups with swapfile extents are skipped and reported, the rest will be processed normally
- resize grow - unaffected
- resize shrink - works as long as the extents are outside of the shrunk range
- device add - a new device does not interfere with existing swapfile and this operation will work, though no new swapfile can be activated afterwards
- device delete - if the device has been added as above, it can be also deleted
- device replace - ditto

When there are no active swapfiles and a whole-filesystem exclusive operation is running (eg. balance, device delete, shrink), the swapfiles cannot be temporarily activated. The operation must finish first.

To create and activate a swapfile run the following commands:

```
# truncate -s 0 swapfile
# chattr +C swapfile
# fallocate -l 2G swapfile
# chmod 0600 swapfile
# mkswap swapfile
# swapon swapfile
```

Please note that the UUID returned by the *mkswap* utility identifies the swap "filesystem" and because it's stored in a file, it's not generally visible and usable as an identifier unlike if it was on a block device.

The file will appear in */proc/swaps*:

```
# cat /proc/swaps
Filename          Type          Size          Used      Priority
/path/swapfile    file          2097152       0         -2
--------------------
```

The swapfile can be created as one-time operation or, once properly created, activated on each boot by the **swapon -a** command (usually started by the service manager). Add the following entry to */etc/fstab*, assuming the filesystem that provides the */path* has been already mounted at this point. Additional mount options relevant for the swapfile can be set too (like priority, not the BTRFS mount options).

```
/path/swapfile        none        swap        defaults      0 0
```

CHAPTER

TWENTYEIGHT

TREE CHECKER

Tree checker is a feature that verifies metadata blocks before write or after read from the devices. The b-tree nodes contain several items describing the filesystem structures and to some degree can be verified for consistency or validity. This is an additional check to the checksums that only verify the overall block status while the tree checker tries to validate and cross reference the logical structure. This takes a slight performance hit but is comparable to calculating the checksum and has no noticeable impact while it does catch all sorts of errors.

There are two occasions when the checks are done:

28.1 Pre-write checks

When metadata blocks are in memory and about to be written to the permanent storage, the checks are performed, before the checksums are calculated. This can catch random corruptions of the blocks (or pages) either caused by bugs or by other parts of the system or hardware errors (namely faulty RAM).

Once a block does not pass the checks, the filesystem refuses to write more data and turns itself to read-only mode to prevent further damage. At this point some the recent metadata updates are held *only* in memory so it's best to not panic and try to remember what files could be affected and copy them elsewhere. Once the filesystem gets unmounted, the most recent changes are unfortunately lost. The filesystem that is stored on the device is still consistent and should mount fine.

A message may look like:

```
[ 1716.823895] BTRFS critical (device vdb): corrupt leaf: root=18446744073709551607␣
↪block=38092800 slot=0, invalid key objectid: has 1 expect 6 or [256,␣
↪18446744073709551360] or 18446744073709551604
[ 1716.829499] BTRFS info (device vdb): leaf 38092800 gen 19 total ptrs 4 free space␣
↪15851 owner 18446744073709551607
[ 1716.832891] BTRFS info (device vdb): refs 3 lock (w:0 r:0 bw:0 br:0 sw:0 sr:0) lock_
↪owner 0 current 1506
[ 1716.836054]  item 0 key (1 1 0) itemoff 16123 itemsize 160
[ 1716.837993]          inode generation 1 size 0 mode 100600
[ 1716.839760]  item 1 key (256 1 0) itemoff 15963 itemsize 160
[ 1716.841742]          inode generation 4 size 0 mode 40755
[ 1716.843393]  item 2 key (256 12 256) itemoff 15951 itemsize 12
[ 1716.845320]  item 3 key (18446744073709551611 48 1) itemoff 15951 itemsize 0
[ 1716.847505] BTRFS error (device vdb): block=38092800 write time tree block corruption␣
↪detected
```

The line(s) before the *write time tree block corruption detected* message is specific to the found error.

28.2 Post-read checks

Metadata blocks get verified right after they're read from devices and the checksum is found to be valid. This protects against changes to the metadata that could possibly also update the checksum, less likely to happen accidentally but rather due to intentional corruption or fuzzing.

```
[ 4823.612832] BTRFS critical (device vdb): corrupt leaf: root=7 block=30474240 slot=0,␣
↪invalid nritems, have 0 should not be 0 for non-root leaf
[ 4823.616798] BTRFS error (device vdb): block=30474240 read time tree block corruption␣
↪detected
```

28.3 The checks

As implemented right now, the metadata consistency is limited to one b-tree node and what items are stored there, ie. there's no extensive or broad check done eg. against other data structures in other b-tree nodes. This still provides enough opportunities to verify consistency of individual items, besides verifying general validity of the items like the length or offset. The b-tree items are also coupled with a key so proper key ordering is also part of the check and can reveal random bitflips in the sequence (this has been the most successful detector of faulty RAM).

The capabilities of tree checker have been improved over time and it's possible that a filesystem created on an older kernel may trigger warnings or fail some checks on a new one.

28.4 Reporting problems

In many cases the bug is caused by hardware and cannot be automatically fixed by *btrfs check --repair*, so do not try that without being adviced to. Even if the error is unfixable it's useful to report it, either to validate the cause but also to give more ideas how to improve the tree checker. Please consider reporting it to the mailing list *linux-btrfs@vger.kernel.org*.

CHAPTER

TWENTYNINE

TRIM/DISCARD

Trim or discard is an operation on a storage device based on flash technology (SSD, NVMe or similar), a thin-provisioned device or could be emulated on top of other block device types. On real hardware, there's a different lifetime span of the memory cells and the driver firmware usually tries to optimize for that. The trim operation issued by user provides hints about what data are unused and allow to reclaim the memory cells. On thin-provisioned or emulated this is could simply free the space.

There are three main uses of trim that BTRFS supports:

synchronous
: enabled by mounting filesystem with `-o discard` or `-o discard=sync`, the trim is done right after the file extents get freed, this however could have severe performance hit and is not recommended as the ranges to be trimmed could be too fragmented

asynchronous
: enabled by mounting filesystem with `-o discard=async`, which is an improved version of the synchronous trim where the freed file extents are first tracked in memory and after a period or enough ranges accumulate the trim is started, expecting the ranges to be much larger and allowing to throttle the number of IO requests which does not interfere with the rest of the filesystem activity

manually by fstrim
: the tool `fstrim` starts a trim operation on the whole filesystem, no mount options need to be specified, so it's up to the filesystem to traverse the free space and start the trim, this is suitable for running it as periodic service

The trim is considered only a hint to the device, it could ignore it completely, start it only on ranges meeting some criteria, or decide not to do it because of other factors affecting the memory cells. The device itself could internally relocate the data, however this leads to unexpected performance drop. Running trim periodically could prevent that too.

When a filesystem is created by `mkfs.btrfs` and is capable of trim, then it's by default performed on all devices.

CHAPTER

THIRTY

VOLUME MANAGEMENT

BTRFS filesystem can be created on top of single or multiple block devices. Devices can be then added, removed or replaced on demand. Data and metadata are organized in allocation profiles with various redundancy policies. There's some similarity with traditional RAID levels, but this could be confusing to users familiar with the traditional meaning. Due to the similarity, the RAID terminology is widely used in the documentation. See `mkfs.btrfs(8)` for more details and the exact profile capabilities and constraints.

The device management works on a mounted filesystem. Devices can be added, removed or replaced, by commands provided by `btrfs device` and `btrfs replace`.

The profiles can be also changed, provided there's enough workspace to do the conversion, using the `btrfs balance` command and namely the filter *convert*.

Type
: The block group profile type is the main distinction of the information stored on the block device. User data are called *Data*, the internal data structures managed by filesystem are *Metadata* and *System*.

Profile
: A profile describes an allocation policy based on the redundancy/replication constraints in connection with the number of devices. The profile applies to data and metadata block groups separately. Eg. *single*, *RAID1*.

RAID level
: Where applicable, the level refers to a profile that matches constraints of the standard RAID levels. At the moment the supported ones are: RAID0, RAID1, RAID10, RAID5 and RAID6.

30.1 Typical use cases

30.1.1 Starting with a single-device filesystem

Assume we've created a filesystem on a block device */dev/sda* with profile *single/single* (data/metadata), the device size is 50GiB and we've used the whole device for the filesystem. The mount point is */mnt*.

The amount of data stored is 16GiB, metadata have allocated 2GiB.

Add new device

We want to increase the total size of the filesystem and keep the profiles. The size of the new device */dev/sdb* is 100GiB.

```
$ btrfs device add /dev/sdb /mnt
```

The amount of free data space increases by less than 100GiB, some space is allocated for metadata.

Convert to RAID1

Now we want to increase the redundancy level of both data and metadata, but we'll do that in steps. Note, that the device sizes are not equal and we'll use that to show the capabilities of split data/metadata and independent profiles.

The constraint for RAID1 gives us at most 50GiB of usable space and exactly 2 copies will be stored on the devices.

First we'll convert the metadata. As the metadata occupy less than 50GiB and there's enough workspace for the conversion process, we can do:

```
$ btrfs balance start -mconvert=raid1 /mnt
```

This operation can take a while, because all metadata have to be moved and all block pointers updated. Depending on the physical locations of the old and new blocks, the disk seeking is the key factor affecting performance.

You'll note that the system block group has been also converted to RAID1, this normally happens as the system block group also holds metadata (the physical to logical mappings).

What changed:

- available data space decreased by 3GiB, usable roughly (50 - 3) + (100 - 3) = 144 GiB
- metadata redundancy increased

IOW, the unequal device sizes allow for combined space for data yet improved redundancy for metadata. If we decide to increase redundancy of data as well, we're going to lose 50GiB of the second device for obvious reasons.

```
$ btrfs balance start -dconvert=raid1 /mnt
```

The balance process needs some workspace (ie. a free device space without any data or metadata block groups) so the command could fail if there's too much data or the block groups occupy the whole first device.

The device size of */dev/sdb* as seen by the filesystem remains unchanged, but the logical space from 50-100GiB will be unused.

Remove device

Device removal must satisfy the profile constraints, otherwise the command fails. For example:

```
$ btrfs device remove /dev/sda /mnt
ERROR: error removing device '/dev/sda': unable to go below two devices on raid1
```

In order to remove a device, you need to convert the profile in this case:

```
$ btrfs balance start -mconvert=dup -dconvert=single /mnt
$ btrfs device remove /dev/sda /mnt
```

CHAPTER

THIRTYONE

ZONED MODE

Since version 5.12 btrfs supports so called *zoned mode*. This is a special on-disk format and allocation/write strategy that's friendly to zoned devices. In short, a device is partitioned into fixed-size zones and each zone can be updated by append-only manner, or reset. As btrfs has no fixed data structures, except the super blocks, the zoned mode only requires block placement that follows the device constraints. You can learn about the whole architecture at https://zonedstorage.io .

The devices are also called SMR/ZBC/ZNS, in *host-managed* mode. Note that there are devices that appear as non-zoned but actually are, this is *drive-managed* and using zoned mode won't help.

The zone size depends on the device, typical sizes are 256MiB or 1GiB. In general it must be a power of two. Emulated zoned devices like *null_blk* allow to set various zone sizes.

31.1 Requirements, limitations

- all devices must have the same zone size
- maximum zone size is 8GiB
- minimum zone size is 4MiB
- mixing zoned and non-zoned devices is possible, the zone writes are emulated, but this is namely for testing
- **the super block is handled in a special way and is at different locations than on a non-zoned filesystem:**
  - primary: 0B (and the next two zones)
  - secondary: 512GiB (and the next two zones)
  - tertiary: 4TiB (4096GiB, and the next two zones)

31.2 Incompatible features

The main constraint of the zoned devices is lack of in-place update of the data. This is inherently incompatibile with some features:

- nodatacow - overwrite in-place, cannot create such files
- fallocate - preallocating space for in-place first write
- mixed-bg - unordered writes to data and metadata, fixing that means using separate data and metadata block groups
- booting - the zone at offset 0 contains superblock, resetting the zone would destroy the bootloader data

Initial support lacks some features but they're planned:

- only single profile is supported
- fstrim - due to dependency on free space cache v1

31.3 Super block

As said above, super block is handled in a special way. In order to be crash safe, at least one zone in a known location must contain a valid superblock. This is implemented as a ring buffer in two consecutive zones, starting from known offsets 0B, 512GiB and 4TiB.

The values are different than on non-zoned devices. Each new super block is appended to the end of the zone, once it's filled, the zone is reset and writes continue to the next one. Looking up the latest super block needs to read offsets of both zones and determine the last written version.

The amount of space reserved for super block depends on the zone size. The secondary and tertiary copies are at distant offsets as the capacity of the devices is expected to be large, tens of terabytes. Maximum zone size supported is 8GiB, which would mean that eg. offset 0-16GiB would be reserved just for the super block on a hypothetical device of that zone size. This is wasteful but required to guarantee crash safety.

CHAPTER

THIRTYTWO

SOURCE REPOSITORIES

Since 2.6.29-rc1, Btrfs has been included in the mainline kernel.

32.1 Kernel module

The kernel.org git repository is not used for development, only for pull requests that go to Linus and for linux-next integration:

- https://git.kernel.org/pub/scm/linux/kernel/git/kdave/linux.git -- pull requests, branch *for-next* gets pulled to the linux-next tree

The following git repositories are used for development and are updated with patches from the mailinglist:

- https://github.com/kdave/btrfs-devel
- https://gitlab.com/kdave/btrfs-devel

Branches are usually pushed to both repositories, either can be used.

There are:

- main queue with patches for next development cycle (branch name *misc-next*)
- queue with patches for current release cycle (the name has the version, eg *for-4.15* or *misc-4.15*).
- topic branches, eg. from a patchset picked from mailinglist
- snapshots of *for-next*, that contain all of the above (eg. for-next-20200512)

Note that the branches get rebased. The base point for patches depend on the development phase. See [[Developer%27s_FAQ#Development_schedule]]. Independent changes can be based on the *linus/master* branch, changes that could depend on patches that have been added to one of the queues should use that as a base.

32.2 btrfs-progs git repository

32.2.1 Official repositories

The sources of the userspace utilities can be obtained from these repositories:

- git://git.kernel.org/pub/scm/linux/kernel/git/kdave/btrfs-progs.git (http://git.kernel.org/?p=linux/kernel/git/kdave/btrfs-progs.git;a=summary) -- release repository, not for development

The **master** branch contains the latest released version and is never rebased.

Development git repositories:

- git://github.com/kdave/btrfs-progs.git (https://github.com/kdave/btrfs-progs)
- git://gitlab.com/kdave/btrfs-progs.git (https://gitlab.com/kdave/btrfs-progs)

For build dependencies and installation instructions please see https://github.com/kdave/btrfs-progs/blob/master/INSTALL

32.2.2 Development branches

The latest **development branch** is called **devel**. Contains patches that are reviewed or tested and on the way to the next release. When a patch is added to the branch, a mail notification is sent as a reply to the patch.

The git repositories on *kernel.org* are not used for development or integration branches.

32.2.3 Note to GitHub users

The pull requests will not be accepted directly, the preferred way is to send patches to the mailinglist instead. You can link to a branch in any git repository if the mails do not make it to the mailinglist or for convenience.

The development model of btrfs-progs shares a lot with the kernel model. The github way is different in some ways. We, the upstream community, expect that the patches meet some criteria (often lacking in github contributions):

- proper **subject line**: eg. prefix with *btrfs-progs: subpart, ...* , descriptive yet not too long
- proper **changelog**: the changelogs are often missing or lacking explanation *why* the change was made, or *how* is something broken, *what* are user-visible effects of the bug or the fix, *how* does an improvement help or the intended *usecase*
- the **Signed-off-by** line: this document who authored the change, you can read more about the *The Developer's Certificate of Origin* here (chapter 11)]
- **one logical change** per patch: eg. not mixing bugfixes, cleanups, features etc., sometimes it's not clear and will be usually pointed out during reviews

32.3 Administration and support tools

There is a separate repository of useful scripts for common administrative tasks on btrfs. This is at:

https://github.com/kdave/btrfsmaintenance/

32.4 Patches sent to mailinglist

A convenient interface to get an overview of patches and the related mail discussions can be found at https://patchwork.kernel.org/project/linux-btrfs/list/ .

It is possible to directly apply a patch by pasting the *mbox* link from the patch page to the command:

```
$ wget -O - '<nowiki>https://patchwork.kernel.org/patch/123456/mbox</nowiki>' | git am -
```

You may want to add *--reject*, or decide otherwise what to do with the patch.

CHAPTER

THIRTYTHREE

CONTRIBUTORS

The following companies contribute to Btrfs code, not counting the treewide and other subsystem changes. Infrequent contributions are not reflected in this list, please have a look to the git history for complete list.

Sorted by amount of contributions:

- SUSE
- Facebook
- Western Digital
- Oracle

The following contributed in the past (sorted alphabetically):

- Fujitsu
- Fusion-IO
- Intel
- Linux Foundation
- Red Hat
- STRATO AG

33.1 Statistics for 5.x series

Version	Contributors	SLOC	Raw lines	Patches	Diffstat
5.0	15	98298	134159	144	+3173 -2297
5.1	18	98992	135308	116	+2208 -1059
5.2	22	99888	136521	255	+3524 -2311
5.3	20	100254	137224	124	+4106 -3400
5.4	18	100660	137889	166	+10752 -10087
5.5	17	100638	138212	168	+3055 -2729
5.6	18	101482	139742	114	+3370 -1840
5.7	22	101661	140694	221	+4484 -3532
5.8	21	101562	140930	158	+3176 -2940
5.9	18	101973	141748	188	+2218 -1400
5.10	22	102378	142760	187	+3148 -2135
5.11	15	102418	143124	229	+4872 -4507
5.12	20	105026	147099	195	+5310 -1316
5.13	19	105820	148503	145	+3334 -1930
5.14	19	106324	149550	121	+2823 -1774
5.15	17	106895	151006	133	+2879 -1422
5.16	24	107854	152760	173	+4770 -3016
5.17	17	107910	153407	154	+4024 -3378
5.18	30	109159	155372	143	+3489 -1523

Legend:

- *Files:* fs/btrfs/*.[ch], fs/btrfs/tests/*.[ch], include/uapi/linux/btrfs.h, include/uapi/linux/btrfs_tree.h, include/linux/btrfs.h, include/trace/events/btrfs.h
- *Version:* mainline version
- *Contributors:* number of people that sent patches that modified ‘’Files”, direct btrfs development or originating from other tree-wide changes
- *SLOC:* lines of code, http://dwheeler.com/sloccount/ (generated by version 2.26)
- *Raw lines:* counted by ‘’wc -l” over ‘’Files”
- *Patches:* number of patches from ‘’Contributors”, merge commits excluded
- *Diffstat:* lines added and deleted in ‘’Files”

33.2 Statistics for 4.x series

Version	Contributors	SLOC	Raw lines	Patches	Diffstat
4.0	22	85849	115716	97	+1622 -937
4.1	25	87596	118253	120	+2415 -1062
4.2	19	87935	118790	119	+2392 -1855
4.3	23	88384	119576	74	+1516 -730
4.4	26	89543	121456	138	+3184 -1304
4.5	26	91708	124363	127	+4370 -1462
4.6	29	92134	125045	100	+1890 -1208
4.7	33	92922	126264	161	+3721 -2502
4.8	22	93769	127392	114	+2732 -1604
4.9	25	94237	128040	64	+1959 -1311
4.10	24	94303	128156	105	+4874 -4758
4.11	24	94365	128340	210	+2084 -1900
4.12	21	94931	129230	85	+1803 -913
4.13	29	93892	127970	135	+2017 -1607
4.14	28	94296	124346	132	+2114 -1520
4.15	30	97091	132221	128	+3761 -1795
4.16	25	97637	133305	188	+2562 -1481
4.17	21	98027	133003	194	+2723 -3024
4.18	26	98387	133667	200	+3643 -2979
4.19	25	97547	132655	193	+2058 -3070
4.20	22	97830	133283	128	+1560 -932

33.3 Statistics for 3.x series

Version	Contributors	SLOC	Raw lines	Patches	Diffstat
3.0	25	48665	65192	126	+7508 -5175
3.1	24	48647	65248	106	+1762 -1586
3.2	30	51574	69552	184	+6344 -2040
3.3	27	56216	75485	129	+7151 -1218
3.4	25	57865	77671	118	+4597 -2411
3.5	21	59683	79983	108	+3570 -1258
3.6	25	65894	88123	104	+9145 -1005
3.7	30	67348	90171	151	+3802 -1754
3.8	25	70289	93916	161	+5599 -1854
3.9	29	73414	98602	160	+6430 -1242
3.10	24	74449	99980	133	+3529 -2151
3.11	21	74875	100657	100	+2538 -1857
3.12	32	76265	102497	158	+4373 -2533
3.13	24	77532	104108	123	+2741 -1123
3.14	28	79879	107069	171	+5290 -2329
3.15	27	80308	107544	152	+2389 -1914
3.16	29	82292	110331	137	+4361 -1574
3.17	19	82625	110841	44	+1060 -550
3.18	25	83910	112906	149	+3696 -1631
3.19	18	85420	115031	82	+2802 -677

33.4 Statistics for 2.6.x series

Version	Contributors	SLOC	Raw lines	Patches	Diffstat
2.6.30	22	33838	45377	70	+4403 -2632
2.6.31	19	38825	51693	68	+9207 -2862
2.6.32	15	40211	53515	95	+4291 -2469
2.6.33	17	40408	53806	43	+1332 -1041
2.6.34	18	41100	54715	54	+1374 -465
2.6.35	14	43014	57082	50	+5230 -2863
2.6.36	4	43016	57088	6	+39 -33
2.6.37	17	44781	59491	83	+3104 -701
2.6.38	23	46573	61980	90	+3472 -983
2.6.39	28	47206	62859	102	+2115 -1236

CHAPTER

THIRTYFOUR

QUICK START

...

CHAPTER

THIRTYFIVE

INTEROPERABILITY

CHAPTER

THIRTYSIX

TROUBLESHOOTING PAGES

System messages printed to the log (dmesg, syslog, journal) have limited space for description and may need further explanation what needs to be done.

36.1 Error: parent transid verify error

Reason: result of a failed internal consistency check of the filesystem's metadata. Type: permanent

```
[ 4007.489730] BTRFS error (device vdb): parent transid verify failed on 30736384 wanted␣
↪10 found 8
```

The b-tree nodes are linked together, a block pointer in the parent node contains target block offset and generation that last changed this block. The block it points to then upon read verifies that the block address and the generation matches. This check is done on all tree levels.

The number in **faled on 30736384** is the logical block number, **wanted 10** is the expected generation number in the parent node, **found 8** is the one found in the target block. The number difference between the generation can give a hint when the problem could have happened, in terms of transaction commits.

Once the mismatched generations are stored on the device, it's permanent and cannot be easily recovered, because of information loss. The recovery tool `btrfs restore` is able to ignore the errors and attempt to restore the data but due to the inconsistency in the metadata the data need to be verified by the user.

The root cause of the error cannot be easily determined, possible reasons are:

- logical bug: filesystem structures haven't been properly updated and stored
- misdirected write: the underlying storage does not store the data to the exact address as expected and overwrites some other block
- storage device (hardware or emulated) does not properly flush and persist data between transactions so they get mixed up
- lost write without proper error handling: writing the block worked as viewed on the filesystem layer, but there was a problem on the lower layers not propagated upwards

36.2 Error: No space left on device (ENOSPC)

Type: transient

Space handling on a COW filesystem is tricky, namely when it's in combination with delayed allocation, dynamic chunk allocation and parallel data updates. There are several reasons why the ENOSPC might get reported and there's not just a single cause and solution. The space reservation algorithms try to fairly assign the space, fall back to heuristics or block writes until enough data are persisted and possibly making old copies available.

The most obvious way how to exhaust space is to create a file until the data chunks are full:

```
$ df -h .
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda        4.0G  3.6M  2.0G   1% /mnt/

$ cat /dev/zero > file
cat: write error: No space left on device

$ df -h .
Filesystem      Size  Used Avail Use% Mounted on
/dev/sdc        4.0G  2.0G     0 100% /mnt/data250

$ btrfs fi df .
Data, single: total=1.98GiB, used=1.98GiB
System, DUP: total=8.00MiB, used=16.00KiB
Metadata, DUP: total=1.00GiB, used=2.22MiB
GlobalReserve, single: total=3.25MiB, used=0.00B
```

The data chunks have been exhausted, so there's really no space left where to write. The metadata chunks have space but that can't be used for that purpose.

36.2.1 Metadata space got exhausted

Cannot track new data extents, no inline files, no reflinks, no xattrs. Deletion still works.

36.2.2 Balance does not have enough workspace

Relocation of block groups requires a temporary work space, ie. area on the device that's available for the filesystem but without any other existing block groups. Before balance starts a check is performed to verify the requested action is possible. If not, ENOSPC is returned.

36.3 Error: unable to start balance with target metadata profile

```
unable to start balance with target metadata profile 32
```

This means that a conversion has been attempted from profile *RAID1* to *dup* with btrfs-progs earlier than version 4.7. Update and you'll be able to do the conversion.

36.4 Error: balance will reduce metadata integrity

The full message in system log

```
balance will reduce metadata integrity, use force if you want this
```

This means that conversion will remove a degree of metadata redundancy, for example when going from profile *RAID1* or *dup* to *single*. The force parameter to `btrfs balance start -f` is needed.

36.5 How to clean old super block

The preferred way is to use the `wipefs` utility that is part of the *util-linux* package. Running the command with the device will not destroy the data, just list the detected filesystems:

```
# wipefs /dev/sda
offset               type
----------------------------------------------------------------
0x10040              btrfs   [filesystem]
                     UUID:  7760469b-1704-487e-9b96-7d7a57d218a5
```

Remove the filesystem signature at a given offset or wipe all recognized signatures on the device:

```
# wipefs -o 0x10040 /dev/sda
8 bytes [5f 42 48 52 66 53 5f 4d] erased at offset 0x10040 (btrfs)

# wipefs -a /dev/sda
8 bytes [5f 42 48 52 66 53 5f 4d] erased at offset 0x10040 (btrfs)
```

Note: The process is reversible, if the 8 bytes are written back, the device is recognized again. See below.

Note: *wipefs* clears only the first super block. If available, the second and third copies can be used to resurrect the filesystem.

36.5.1 Stale signature on device

Related problem regarding partitioned and unpartitioned device: *Long time ago I created btrfs on /dev/sda. After some changes btrfs moved to /dev/sda1.*

Use `wipefs -o 0x10040` (ie. with the offset of the btrfs signature), it won't touch the parition table.

36.5.2 Manual deletion of super block signature

There are three superblocks: the first one is located at 64KiB, the second one at 64MiB, the third one at 256GiB. The following lines reset the signature on all the three copies:

```
# dd if=/dev/zero bs=1 count=8 of=/dev/sda seek=$((64*1024+64))
# dd if=/dev/zero bs=1 count=8 of=/dev/sda seek=$((64*1024*1024+64))
# dd if=/dev/zero bs=1 count=8 of=/dev/sda seek=$((256*1024*1024*1024+64))
```

If you want to restore the super block signatures:

```
# echo "_BHRfS_M" | dd bs=1 count=8 of=/dev/sda seek=$((64*1024+64))
# echo "_BHRfS_M" | dd bs=1 count=8 of=/dev/sda seek=$((64*1024*1024+64))
# echo "_BHRfS_M" | dd bs=1 count=8 of=/dev/sda seek=$((256*1024*1024*1024+64))
```

36.6 Generic errors, errno

Note there's a established text message for the errors, though they are used in a broader sense (eg. error mentions a file but it can be relevant for another structure). The title of each section uses the nonstandard meaning that is perhaps more suitable for a filesystem.

36.6.1 ENOENT (No such entry)

Common error "no such entry", in general it may mean that some structure hasn't been found, eg. an entry in some in-memory tree. This becomes a critical problem when the entry is expected to exist because of consistency of the structures.

36.6.2 ENOMEM (Not enough memory)

Memory allocation error. In many cases the error is recoverable and the operation restartable after it's reported to userspace. In critical contexts, like when a transaction needs to be committed, the error is not recoverable and leads to flipping the filesystem to read-only. Such cases are rare under normal conditions. Memory can be artificially limited eg. by cgroups, which may trigger the condition, which is useful for testing but any real workload should have resources scaled accordingly.

36.6.3 EINVAL (Invalid argument)

This is typically returned from ioctl when a parameter is invalid, ie. unexpected range, a bit flag not recognized, or a combination of input parameters that does not make sense. Errors are typically recoverable.

36.6.4 EUCLEAN (Filesystem corrupted)

The text of the message is confusing "Structure needs cleaning", in reality this is used to describe a severe corruption condition. The reason of the corruption is unknown at this point, but some constraint or condition has been violated and the filesystem driver can't do much. In practice such errors can be observed on fuzzed images, faulty hardware or misinteraction with other parts of the operating system.

36.6.5 EIO (Input/output error)

"Input output error", typically returned as an error from a device that was unable to read data, or finish a write. Checksum errors also lead to EIO, there isn't an established error for checksum validation errors, although some filesystems use EBADMSG for that.

36.6.6 EEXIST (Object already exists)

36.6.7 ENOSPC (No space left)

36.6.8 EOPNOTSUPP (Operation not supported)

36.7 TODO

Transient

- enospc
- operation cannot be done

Possibly both

- checksum errors from changes on the medium under hands
- transient because of direct io
- stored from faulty data in memory

CHAPTER

THIRTYSEVEN

EXPERIMENTAL FEATURES

Experimental or unstable features may be enabled by

```
./configure --enable-experimental
```

but as it says, the interface, command names, output formatting should be considered unstable and not for production use. However testing is welcome and feedback or bugs filed as issues.

In the code use it like:

```
if (EXPERIMENTAL) {
    ...
}
```

in case it does not interfere with other code or does not depend on an *#if* where it would break default build.

Or:

```
#if EXPERIMENTAL
...
#endif
```

for larger code blocks.

Note: Do not use *#ifdef* as the macro is always defined so this would not work as expected.

Each feature should be tracked in an issue with label **experimental** (list of active issues https://github.com/kdave/btrfs-progs/labels/experimental), with a description and a todo list items. Individual tasks can be tracked in other issues if needed.

CHAPTER

THIRTYEIGHT

BTRFS-IOCTL(3)

38.1 NAME

btrfs-ioctl - documentation for the ioctl interface to btrfs

38.2 DESCRIPTION

The ioctl() system call is a way how to request custom actions performed on a filesystem beyond the standard interfaces (like syscalls). An ioctl is specified by a number and an associated data structure that implement a feature, usually not available in other filesystems. The number of ioctls grows over time and in some cases get promoted to a VFS-level ioctl once other filesystems adopt the functionality. Backward compatibility is maintained and a formerly private ioctl number could become available on the VFS level.

38.3 DATA STRUCTURES AND DEFINITIONS

```
struct btrfs_ioctl_vol_args {
        __s64 fd;
        char name[BTRFS_PATH_NAME_MAX + 1];
};
```

```
struct btrfs_ioctl_vol_args_v2 {
        __s64 fd;
        __u64 transid;
        __u64 flags;
        union {
                struct {
                        __u64 size;
                        struct btrfs_qgroup_inherit __user *qgroup_inherit;
                };
                __u64 unused[4];
        };
        char name[BTRFS_SUBVOL_NAME_MAX + 1];
};
```

```
BTRFS_SUBVOL_NAME_MAX = 4039
BTRFS_PATH_NAME_MAX = 4087
```

38.4 OVERVIEW

The ioctls are defined by a number and associated with a data structure that contains further information. All ioctls use file descriptor (fd) as a reference point, it could be the filesystem or a directory inside the filesystem.

An ioctl can be used in the following schematic way:

```
struct btrfs_ioctl_args args;

memset(&args, 0, sizeof(args));
args.key = value;
ret = ioctl(fd, BTRFS_IOC_NUMBER, &args);
```

The 'fd' is the entry point to the filesystem and for most ioctls it does not matter which file or directory is that. Where it matters it's explicitly mentioned. The 'args' is the associated data structure for the request. It's strongly recommended to initialize the whole structure to zeros as this is future-proof when the ioctl gets further extensions. Not doing that could lead to mismatch of old userspace and new kernel versions, or vice versa. The 'BTRFS_IOC_NUMBER' is says which operation should be done on the given arguments. Some ioctls take a specific data structure, some of them share a common one, no argument structure ioctls exist too.

The library 'libbtrfsutil' wraps a few ioctls for convenience. Using raw ioctls is not discouraged but may be cumbersome though it does not need additional library dependency. Backward compatibility is guaranteed and incompatible changes usually lead to a new version of the ioctl. Enhancements of existing ioctls can happen and depend on additional flags to be set. Zeroed unused space is commonly understood as a mechanism to communicate the compatibility between kernel and userspace and thus zeroing is really important. In exceptional cases this is not enough and further flags need to be passed to distinguish between zero as implicit unused initialization and a valid zero value. Such cases are documented.

38.5 LIST OF IOCTLS

- BTRFS_IOC_SUBVOL_CREATE -- (obsolete) create a subvolume
- BTRFS_IOC_SNAP_CREATE
- BTRFS_IOC_DEFRAG
- BTRFS_IOC_RESIZE
- BTRFS_IOC_SCAN_DEV
- BTRFS_IOC_SYNC
- BTRFS_IOC_CLONE
- BTRFS_IOC_ADD_DEV
- BTRFS_IOC_RM_DEV
- BTRFS_IOC_BALANCE
- BTRFS_IOC_CLONE_RANGE
- BTRFS_IOC_SUBVOL_CREATE
- BTRFS_IOC_SNAP_DESTROY
- BTRFS_IOC_DEFRAG_RANGE
- BTRFS_IOC_TREE_SEARCH

- BTRFS_IOC_TREE_SEARCH_V2
- BTRFS_IOC_INO_LOOKUP
- BTRFS_IOC_DEFAULT_SUBVOL
- BTRFS_IOC_SPACE_INFO
- BTRFS_IOC_START_SYNC
- BTRFS_IOC_WAIT_SYNC
- BTRFS_IOC_SNAP_CREATE_V2
- BTRFS_IOC_SUBVOL_CREATE_V2 -- create a subvolume
- BTRFS_IOC_SUBVOL_GETFLAGS
- BTRFS_IOC_SUBVOL_SETFLAGS
- BTRFS_IOC_SCRUB
- BTRFS_IOC_SCRUB_CANCEL
- BTRFS_IOC_SCRUB_PROGRESS
- BTRFS_IOC_DEV_INFO
- BTRFS_IOC_FS_INFO
- BTRFS_IOC_BALANCE_V2
- BTRFS_IOC_BALANCE_CTL
- BTRFS_IOC_BALANCE_PROGRESS
- BTRFS_IOC_INO_PATHS
- BTRFS_IOC_LOGICAL_INO
- BTRFS_IOC_SET_RECEIVED_SUBVOL
- BTRFS_IOC_SEND
- BTRFS_IOC_DEVICES_READY
- BTRFS_IOC_QUOTA_CTL
- BTRFS_IOC_QGROUP_ASSIGN
- BTRFS_IOC_QGROUP_CREATE
- BTRFS_IOC_QGROUP_LIMIT
- BTRFS_IOC_QUOTA_RESCAN
- BTRFS_IOC_QUOTA_RESCAN_STATUS
- BTRFS_IOC_QUOTA_RESCAN_WAIT
- BTRFS_IOC_GET_FSLABEL
- BTRFS_IOC_SET_FSLABEL
- BTRFS_IOC_GET_DEV_STATS
- BTRFS_IOC_DEV_REPLACE
- BTRFS_IOC_FILE_EXTENT_SAME
- BTRFS_IOC_GET_FEATURES

- BTRFS_IOC_SET_FEATURES
- BTRFS_IOC_GET_SUPPORTED_FEATURES

38.6 DETAILED DESCRIPTION

38.6.1 BTRFS_IOC_SUBVOL_CREATE

Note: obsoleted by BTRFS_IOC_SUBVOL_CREATE_V2

(since: 3.0, obsoleted: 4.0) Create a subvolume.

ioctl fd
file descriptor of the parent directory of the new subvolume

argument type
struct btrfs_ioctl_vol_args

fd
ignored

name
name of the subvolume, although the buffer can be almost 4k, the file size is limited by linux VFS to 255 characters and must not contain a slash ('/')

38.6.2 BTRFS_IOC_SUBVOL_CREATE_V2

Note: obsoletes BTRFS_IOC_SUBVOL_CREATE

(since: 3.6) Create a subvolume, qgroup inheritance can be specified.

ioctl fd
file descriptor of the parent directory of the new subvolume

argument type
struct btrfs_ioctl_vol_args_v2

fd
ignored

transid
ignored

flags
ignored

size
...

qgroup_inherit
...

name
name of the subvolume, although the buffer can be almost 4k, the file size is limited by linux VFS to 255 characters and must not contain a slash ('/')

devid

...

38.7 AVAILABILITY

btrfs is part of btrfs-progs. Please refer to the btrfs wiki http://btrfs.wiki.kernel.org for further details.

38.8 SEE ALSO

ioctl(2)

CHAPTER

THIRTYNINE

CONVENTIONS AND STYLE FOR DOCUMENTATION

Manual pages structure:

- add references to all external commands mentioned anywhere in the text to the *SEE ALSO* section - also add related, not explicitly mentioned
- the heading levels are validated - mandatory, manual page `===` - mandatory, sections `---` - optional, sub-sections `^^^`
- command-specific examples are mostly free of restrictions but should be readable in all output formats (manual page, html)
- subcommands are in alphabetical order
- long command output or shell examples: verbatim output - use code-block:: directive

Quotation in subcommands:

- exact syntax: monotype `usage=0`
- reference to arguments etc: *italics*
- command reference: bold `btrfs filesystem show` - subcommand names should be spelled in full, ie. *filesystem* instead of *fi*
- section references: italics *EXAMPLES*
- argument name in option description: caps in angle brackets <NAME> - reference in help text: caps NAME - also possible: caps italics *NAME*
- command short description: - command name: bold **command** - optional unspecified: brackets [options] - mandatory argument: angle brackets <path> - optional parameter with argument: [-p <path>]

Other:

- for notes use note:: directive, is rendered as a separate paragraph and should be used only for important information
- warning:: directive is rendered as a separate paragraph and most likely more visible than NOTE, use for critical information that may cause harm, irreversible state or performance problems - should point reader to other part of documentation to seek more details

References: - RST and Sphinx Cheatsheet https://spl.hevs.io/spl-docs/writing/rst/cheatsheet.html - RST Cheat Sheet https://sphinx-tutorial.readthedocs.io/cheatsheet/