NAME

btrfs-man5 − topics about the BTRFS filesystem (mount options, supported file attributes and other)

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. filesystem limits
5. bootloader support
6. file attributes
7. control device

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.

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 64K) 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 defaul 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, **nodiscard**

(default: off)

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. if the backing device does not support asynchronous queued TRIM, then this operation 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.

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.

inode_cache, **noinode_cache**

(since: 3.0, default: off)

Enable free inode number caching. Not recommended to use unless files on your filesystem get assigned inode numbers that are approaching 264. Normally, new files in each subvolume get assigned incrementally (plus one from the last time) and are not reused. The mount option turns on caching of the existing inode numbers and reuse of inode numbers of deleted files.

This option may slow down your system at first run, or after mounting without the option.

Note

Defaults to off due to a potential overflow problem when the free space checksums don't fit inside a single page.

Don't use this option unless you really need it. The inode number limit on 64bit system is 264, which is practically enough for the whole filesystem lifetime. Due to implementation of linux VFS layer, the inode numbers on 32bit systems are only 32 bits wide. This lowers the limit significantly and makes it possible to reach it. In such case, this mount option will help. Alternatively, files with high inode numbers can be copied to a new subvolume which will effectively start the inode numbers from the beginning again.

logreplay, **nologreplay**

(default: on, even read−only)

Enable/disable log replay at mount time. 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 4k 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.

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**(8) command currently only has read−only support for *v2*. A read−write command may be run on a *v2* filesystem by clearing the cache, running the command, and then remounting with *space_cache=v2*.

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, **nousebackuproot**

(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.

alloc_start=*bytes*

(default: 1M, minimum: 1M, deprecated since: 4.13)

Debugging option to force all block allocations above a certain byte threshold on each block device. The value is specified in bytes, optionally with a K, M, or G suffix (case insensitive).

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.

subvolrootid=*objectid*

(irrelevant since: 3.2, formally deprecated since: 3.10)

A workaround option from times (pre 3.2) when it was not possible to mount a subvolume that did not reside directly under the toplevel subvolume.

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).

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 device** 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

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

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

SWAPFILE SUPPORT

The swapfile 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 subystem:

- filesystem − must be only single device
- 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 (ie. balance, device delete, shrink), the swapfiles cannot be temporarily activated. The operation must finish first.

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

CHECKSUM ALGORITHMS

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 256bits (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:

[cols="^,>,>",width="50%"]

Digest	**Cycles/4KiB**	**Ratio**
CRC32C	1700	1.00
XXHASH	2500	1.44
SHA256	105000	61
BLAKE2b	22000	13

FILESYSTEM LIMITS

maximum file name length

255

maximum symlink target length

depends on the *nodesize* value, for 4k 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

264 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 264 (16 EiB) but the linux VFS limit is 263 (8 EiB)

maximum number of subvolumes

the subvolume ids can go up to 264 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

BOOTLOADER SUPPORT

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

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

FILE ATTRIBUTES

The btrfs filesystem supports setting the following file attributes using the **chattr**(1) utility:

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.

C

no copy−on−write, file 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)

S

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

X

no compression, permanently turn off compression on the given file. Any compression mount options will not affect this file.

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

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

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 usually created by a system device node manager (eg. udev), but can be created manually:

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

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

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