The SGI XFS Filesystem

XFS is a high performance journaling filesystem which originated on the SGI IRIX platform. It is completely multi-threaded, can support large files and large filesystems, extended attributes, variable block sizes, is extent based, and makes extensive use of Btrees (directories, extents, free space) to aid both performance and scalability.

Refer to the documentation at https://xfs.wiki.kernel.org/ for further details. This implementation is on-disk compatible with the IRIX version of XFS.

Mount Options

When mounting an XFS filesystem, the following options are accepted.

allocsize=size

> Sets the buffered I/O end-of-file preallocation size when doing delayed allocation writeout (default size is 64KiB). Valid values for this option are page size (typically 4KiB) through to 1GiB, inclusive, in power-of-2 increments.
>
> The default behaviour is for dynamic end-of-file preallocation size, which uses a set of heuristics to optimise the preallocation size based on the current allocation patterns within the file and the access patterns to the file. Specifying a fixed `allocsize` value turns off the dynamic behaviour.

attr2 or noattr2

> The options enable/disable an "opportunistic" improvement to be made in the way inline extended attributes are stored on-disk. When the new form is used for the first time when `attr2` is selected (either when setting or removing extended attributes) the on-disk superblock feature bit field will be updated to reflect this format being in use.
>
> The default behaviour is determined by the on-disk feature bit indicating that `attr2` behaviour is active. If either mount option is set, then that becomes the new default used by the filesystem.
>
> CRC enabled filesystems always use the `attr2` format, and so will reject the `noattr2` mount option if it is set.

discard or nodiscard (default)

> Enable/disable the issuing of commands to let the block device reclaim space freed by the filesystem. This is useful for SSD devices, thinly provisioned LUNs and virtual machine images, but may have a performance impact.
>
> Note: It is currently recommended that you use the `fstrim` application to `discard` unused blocks rather than the `discard` mount option because the performance impact of this option is quite severe.

grpid/bsdgroups or nogrpid/sysvgroups (default)

> These options define what group ID a newly created file gets. When `grpid` is set, it takes the group ID of the directory in which it is created; otherwise it takes the `fsgid` of the current process, unless the directory has the `setgid` bit set, in which case it takes the `gid` from the parent directory, and also gets the `setgid` bit set if it is a directory itself.

filestreams

> Make the data allocator use the filestreams allocation mode across the entire filesystem rather than just on directories configured to use it.

ikeep or noikeep (default)

> When `ikeep` is specified, XFS does not delete empty inode clusters and keeps them around on disk. When `noikeep` is specified, empty inode clusters are returned to the free space pool.

inode32 or inode64 (default)

> When `inode32` is specified, it indicates that XFS limits inode creation to locations which will not result in inode numbers with more than 32 bits of significance.
>
> When `inode64` is specified, it indicates that XFS is allowed to create inodes at any location in the filesystem, including those which will result in inode numbers occupying more than 32 bits of significance.
>
> `inode32` is provided for backwards compatibility with older systems and applications, since 64 bits inode numbers might cause problems for some applications that cannot handle large inode numbers. If applications are in use which do not handle inode numbers bigger than 32 bits, the `inode32` option should be specified.

largeio or nolargeio (default)

> If `nolargeio` is specified, the optimal I/O reported in `st_blksize` by **stat(2)** will be as small as possible to

allow user applications to avoid inefficient read/modify/write I/O. This is typically the page size of the machine, as this is the granularity of the page cache.

If `largeio` is specified, a filesystem that was created with a `swidth` specified will return the `swidth` value (in bytes) in `st_blksize`. If the filesystem does not have a `swidth` specified but does specify an `allocsize` then `allocsize` (in bytes) will be returned instead. Otherwise the behaviour is the same as if `nolargeio` was specified.

logbufs=value

Set the number of in-memory log buffers. Valid numbers range from 2-8 inclusive.

The default value is 8 buffers.

If the memory cost of 8 log buffers is too high on small systems, then it may be reduced at some cost to performance on metadata intensive workloads. The `logbsize` option below controls the size of each buffer and so is also relevant to this case.

logbsize=value

Set the size of each in-memory log buffer. The size may be specified in bytes, or in kilobytes with a "k" suffix. Valid sizes for version 1 and version 2 logs are 16384 (16k) and 32768 (32k). Valid sizes for version 2 logs also include 65536 (64k), 131072 (128k) and 262144 (256k). The logbsize must be an integer multiple of the log stripe unit configured at **mkfs(8)** time.

The default value for version 1 logs is 32768, while the default value for version 2 logs is MAX(32768, log_sunit).

logdev=device and rtdev=device

Use an external log (metadata journal) and/or real-time device. An XFS filesystem has up to three parts: a data section, a log section, and a real-time section. The real-time section is optional, and the log section can be separate from the data section or contained within it.

noalign

Data allocations will not be aligned at stripe unit boundaries. This is only relevant to filesystems created with non-zero data alignment parameters (`sunit`, `swidth`) by **mkfs(8)**.

norecovery

The filesystem will be mounted without running log recovery. If the filesystem was not cleanly unmounted, it is likely to be inconsistent when mounted in `norecovery` mode. Some files or directories may not be accessible because of this. Filesystems mounted `norecovery` must be mounted read-only or the mount will fail.

nouuid

Don't check for double mounted file systems using the file system `uuid`. This is useful to mount LVM snapshot volumes, and often used in combination with `norecovery` for mounting read-only snapshots.

noquota

Forcibly turns off all quota accounting and enforcement within the filesystem.

uquota/usrquota/uqnoenforce/quota

User disk quota accounting enabled, and limits (optionally) enforced. Refer to **xfs_quota(8)** for further details.

gquota/grpquota/gqnoenforce

Group disk quota accounting enabled and limits (optionally) enforced. Refer to **xfs_quota(8)** for further details.

pquota/prjquota/pqnoenforce

Project disk quota accounting enabled and limits (optionally) enforced. Refer to **xfs_quota(8)** for further details.

sunit=value and swidth=value

Used to specify the stripe unit and width for a RAID device or a stripe volume. "value" must be specified in 512-byte block units. These options are only relevant to filesystems that were created with non-zero data alignment parameters.

The `sunit` and `swidth` parameters specified must be compatible with the existing filesystem alignment characteristics. In general, that means the only valid changes to `sunit` are increasing it by a power-of-2 multiple. Valid `swidth` values are any integer multiple of a valid `sunit` value.

Typically the only time these mount options are necessary if after an underlying RAID device has had it's geometry modified, such as adding a new disk to a RAID5 lun and reshaping it.

swalloc

Data allocations will be rounded up to stripe width boundaries when the current end of file is being extended and the file size is larger than the stripe width size.

wsync

When specified, all filesystem namespace operations are executed synchronously. This ensures that when the namespace operation (create, unlink, etc) completes, the change to the namespace is on stable storage. This is useful in HA setups where failover must not result in clients seeing inconsistent namespace presentation during or after a failover event.

Deprecation of V4 Format

The V4 filesystem format lacks certain features that are supported by the V5 format, such as metadata checksumming, strengthened metadata verification, and the ability to store timestamps past the year 2038. Because of this, the V4 format is deprecated. All users should upgrade by backing up their files, reformatting, and restoring from the backup.

Administrators and users can detect a V4 filesystem by running xfs_info against a filesystem mountpoint and checking for a string containing "crc=". If no such string is found, please upgrade xfsprogs to the latest version and try again.

The deprecation will take place in two parts. Support for mounting V4 filesystems can now be disabled at kernel build time via Kconfig option. The option will default to yes until September 2025, at which time it will be changed to default to no. In September 2030, support will be removed from the codebase entirely.

Note: Distributors may choose to withdraw V4 format support earlier than the dates listed above.

Deprecated Mount Options

Name	Removal Schedule
Mounting with V4 filesystem	September 2030
ikeep/noikeep	September 2025
attr2/noattr2	September 2025

Removed Mount Options

Name	Removed
delaylog/nodelaylog	v4.0
ihashsize	v4.0
irixsgid	v4.0
osyncisdsync/osyncisosync	v4.0
barrier	v4.19
nobarrier	v4.19

sysctls

The following sysctls are available for the XFS filesystem:

fs.xfs.stats_clear (Min: 0 Default: 0 Max: 1)

Setting this to "1" clears accumulated XFS statistics in /proc/fs/xfs/stat. It then immediately resets to "0".

fs.xfs.xfssyncd_centisecs (Min: 100 Default: 3000 Max: 720000)

The interval at which the filesystem flushes metadata out to disk and runs internal cache cleanup routines.

fs.xfs.filestream_centisecs (Min: 1 Default: 3000 Max: 360000)

The interval at which the filesystem ages filestreams cache references and returns timed-out AGs back to the free stream pool.

fs.xfs.speculative_prealloc_lifetime

(Units: seconds Min: 1 Default: 300 Max: 86400) The interval at which the background scanning for inodes with unused speculative preallocation runs. The scan removes unused preallocation from clean inodes and releases the unused space back to the free pool.

fs.xfs.speculative_cow_prealloc_lifetime

This is an alias for speculative_prealloc_lifetime.

fs.xfs.error_level (Min: 0 Default: 3 Max: 11)

A volume knob for error reporting when internal errors occur. This will generate detailed messages & backtraces for filesystem shutdowns, for example. Current threshold values are:

XFS_ERRLEVEL_OFF: 0 XFS_ERRLEVEL_LOW: 1 XFS_ERRLEVEL_HIGH: 5

fs.xfs.panic_mask (Min: 0 Default: 0 Max: 256)

Causes certain error conditions to call BUG(). Value is a bitmask; OR together the tags which represent errors which should cause panics:

XFS_NO_PTAG 0 XFS_PTAG_IFLUSH 0x00000001 XFS_PTAG_LOGRES 0x00000002 XFS_PTAG_AILDELETE 0x00000004 XFS_PTAG_ERROR_REPORT 0x00000008 XFS_PTAG_SHUTDOWN_CORRUPT 0x00000010 XFS_PTAG_SHUTDOWN_IOERROR 0x00000020 XFS_PTAG_SHUTDOWN_LOGERROR 0x00000040 XFS_PTAG_FSBLOCK_ZERO 0x00000080 XFS_PTAG_VERIFIER_ERROR 0x00000100

This option is intended for debugging only.

fs.xfs.irix_symlink_mode (Min: 0 Default: 0 Max: 1)

Controls whether symlinks are created with mode 0777 (default) or whether their mode is affected by the umask (irix mode).

fs.xfs.irix_sgid_inherit (Min: 0 Default: 0 Max: 1)

Controls files created in SGID directories. If the group ID of the new file does not match the effective group ID or one of the supplementary group IDs of the parent dir, the ISGID bit is cleared if the irix_sgid_inherit compatibility sysctl is set.

fs.xfs.inherit_sync (Min: 0 Default: 1 Max: 1)

Setting this to "1" will cause the "sync" flag set by the **xfs_io(8)** chattr command on a directory to be inherited by files in that directory.

fs.xfs.inherit_nodump (Min: 0 Default: 1 Max: 1)

Setting this to "1" will cause the "nodump" flag set by the **xfs_io(8)** chattr command on a directory to be inherited by files in that directory.

fs.xfs.inherit_noatime (Min: 0 Default: 1 Max: 1)

Setting this to "1" will cause the "noatime" flag set by the **xfs_io(8)** chattr command on a directory to be inherited by files in that directory.

fs.xfs.inherit_nosymlinks (Min: 0 Default: 1 Max: 1)

Setting this to "1" will cause the "nosymlinks" flag set by the **xfs_io(8)** chattr command on a directory to be inherited by files in that directory.

fs.xfs.inherit_nodefrag (Min: 0 Default: 1 Max: 1)

Setting this to "1" will cause the "nodefrag" flag set by the **xfs_io(8)** chattr command on a directory to be inherited by files in that directory.

fs.xfs.rotorstep (Min: 1 Default: 1 Max: 256)

In "inode32" allocation mode, this option determines how many files the allocator attempts to allocate in the same allocation group before moving to the next allocation group. The intent is to control the rate at which the allocator moves between allocation groups when allocating extents for new files.

Deprecated Sysctls

Name	Removal Schedule
fs.xfs.irix_sgid_inherit	September 2025
fs.xfs.irix_symlink_mode	September 2025
fs.xfs.speculative_cow_prealloc_lifetime	September 2025

Removed Sysctls

Name	Removed
fs.xfs.xfsbufd_centisec	v4.0
fs.xfs.age_buffer_centisecs	v4.0

Error handling

XFS can act differently according to the type of error found during its operation. The implementation introduces the following concepts to the error handler:

-failure speed:

Defines how fast XFS should propagate an error upwards when a specific error is found during the filesystem operation. It can propagate immediately, after a defined number of retries, after a set time period, or simply retry forever.

-error classes:

Specifies the subsystem the error configuration will apply to, such as metadata IO or memory allocation. Different subsystems will have different error handlers for which behaviour can be configured.

-error handlers:

Defines the behavior for a specific error.

The filesystem behavior during an error can be set via `sysfs` files. Each error handler works independently - the first condition met by an error handler for a specific class will cause the error to be propagated rather than reset and retried.

The action taken by the filesystem when the error is propagated is context dependent - it may cause a shut down in the case of an unrecoverable error, it may be reported back to userspace, or it may even be ignored because there's nothing useful we can with the error or anyone we can report it to (e.g. during unmount).

The configuration files are organized into the following hierarchy for each mounted filesystem:

/sys/fs/xfs/<dev>/error/<class>/<error>/

Where:

<dev>

The short device name of the mounted filesystem. This is the same device name that shows up in XFS kernel error messages as "XFS(<dev>): ..."

<class>

The subsystem the error configuration belongs to. As of 4.9, the defined classes are:

- "metadata": applies metadata buffer write IO

<error>

The individual error handler configurations.

Each filesystem has "global" error configuration options defined in their top level directory:

/sys/fs/xfs/<dev>/error/

fail_at_unmount (Min: 0 Default: 1 Max: 1)

Defines the filesystem error behavior at unmount time.

If set to a value of 1, XFS will override all other error configurations during unmount and replace them with "immediate fail" characteristics. i.e. no retries, no retry timeout. This will always allow unmount to succeed when there are persistent errors present.

If set to 0, the configured retry behaviour will continue until all retries and/or timeouts have been exhausted. This will delay unmount completion when there are persistent errors, and it may prevent the filesystem from ever unmounting fully in the case of "retry forever" handler configurations.

Note: there is no guarantee that fail_at_unmount can be set while an unmount is in progress. It is possible that the `sysfs` entries are removed by the unmounting filesystem before a "retry forever" error handler configuration causes unmount to hang, and hence the filesystem must be configured appropriately before unmount begins to prevent unmount hangs.

Each filesystem has specific error class handlers that define the error propagation behaviour for specific errors. There is also a "default" error handler defined, which defines the behaviour for all errors that don't have specific handlers defined. Where multiple retry constraints are configured for a single error, the first retry configuration that expires will cause the error to be propagated. The handler configurations are found in the directory:

/sys/fs/xfs/<dev>/error/<class>/<error>/

max_retries (Min: -1 Default: Varies Max: INTMAX)

Defines the allowed number of retries of a specific error before the filesystem will propagate the error. The retry count for a given error context (e.g. a specific metadata buffer) is reset every time there is a successful completion of the operation.

Setting the value to "-1" will cause XFS to retry forever for this specific error.

Setting the value to "0" will cause XFS to fail immediately when the specific error is reported.

Setting the value to "N" (where 0 < N < Max) will make XFS retry the operation "N" times before propagating the error.

retry_timeout_seconds (Min: -1 Default: Varies Max: 1 day)

Define the amount of time (in seconds) that the filesystem is allowed to retry its operations when the specific error is found.

Setting the value to "-1" will allow XFS to retry forever for this specific error.

Setting the value to "0" will cause XFS to fail immediately when the specific error is reported.

Setting the value to "N" (where 0 < N < Max) will allow XFS to retry the operation for up to "N" seconds before propagating the error.

Note: The default behaviour for a specific error handler is dependent on both the class and error context. For example, the default values for "metadata/ENODEV" are "0" rather than "-1" so that this error handler defaults to "fail immediately" behaviour. This is done because ENODEV is a fatal, unrecoverable error no matter how many times the metadata IO is retried.

Workqueue Concurrency

XFS uses kernel workqueues to parallelize metadata update processes. This enables it to take advantage of storage hardware that can service many IO operations simultaneously. This interface exposes internal implementation details of XFS, and as such is explicitly not part of any userspace API/ABI guarantee the kernel may give userspace. These are undocumented features of the generic workqueue implementation XFS uses for concurrency, and they are provided here purely for diagnostic and tuning purposes and may change at any time in the future.

The control knobs for a filesystem's workqueues are organized by task at hand and the short name of the data device. They all can be found in:

/sys/bus/workqueue/devices/${task}!${device}

Task	Description
xfs_iwalk-$pid	Inode scans of the entire filesystem. Currently limited to mount time quotacheck.
xfs-gc	Background garbage collection of disk space that have been speculatively allocated beyond EOF or for staging copy on write operations.

For example, the knobs for the quotacheck workqueue for /dev/nvme0n1 would be found in /sys/bus/workqueue/devices/xfs_iwalk-1111!nvme0n1/.

The interesting knobs for XFS workqueues are as follows:

Knob	Description
max_active	Maximum number of background threads that can be started to run the work.
cpumask	CPUs upon which the threads are allowed to run.
nice	Relative priority of scheduling the threads. These are the same nice levels that can be applied to userspace processes.