# fio Documentation

*Release 3.41-49-gde3d*

**a**

**Dec 07, 2025**

# CONTENTS


**1 fio - Flexible I/O tester rev. 3.41** **3**

1.1 Overview and history . . . 3
1.2 Source . . . 3
1.3 Mailing list . . . 3
1.4 Author . . . 4
1.5 Maintainers . . . 4
1.6 Binary packages . . . 4
1.7 Building . . . 5
1.7.1 Windows . . . 5
1.7.2 Documentation . . . 6
1.8 Platforms . . . 6
1.9 Running fio . . . 6
1.10 How fio works . . . 7
1.11 Command line options . . . 7
1.12 Job file format . . . 11
1.12.1 Environment variables . . . 13
1.12.2 Reserved keywords . . . 13
1.13 Job file parameters . . . 13
1.13.1 Parameter types . . . 14
1.13.2 Units . . . 16
1.13.3 Job description . . . 16
1.13.4 Time related parameters . . . 17
1.13.5 Target file/device . . . 18
1.13.6 I/O type . . . 22
1.13.7 Block size . . . 29
1.13.8 Buffers and memory . . . 31
1.13.9 I/O size . . . 33
1.13.10 I/O engine . . . 34
1.13.11 I/O engine specific parameters . . . 38
1.13.12 I/O depth . . . 51
1.13.13 I/O rate . . . 52
1.13.14 I/O latency . . . 53
1.13.15 I/O replay . . . 54
1.13.16 Threads, processes and job synchronization . . . 55
1.13.17 Verification . . . 58
1.13.18 Steady state . . . 61
1.13.19 Measurements and reporting . . . 62
1.13.20 Error handling . . . 65
1.14 Running predefined workloads . . . 66
1.14.1 Act profile options . . . 67

1.14.2 Tiobench profile options . . . 67
1.15 Interpreting the output . . . 67
1.16 Terse output . . . 71
1.17 JSON output . . . 74
1.18 JSON+ output . . . 74
1.19 Trace file format . . . 74
1.19.1 Trace file format v1 . . . 74
1.19.2 Trace file format v2 . . . 74
1.19.3 Trace file format v3 . . . 75
1.20 I/O Replay - Merging Traces . . . 76
1.21 CPU idleness profiling . . . 76
1.22 Verification and triggers . . . 76
1.22.1 Verification trigger example . . . 77
1.22.2 Loading verify state . . . 77
1.23 Log File Formats . . . 77
1.24 Client/Server . . . 78

**2 Examples** **81**
2.1 Poisson request flow . . . 81
2.2 Latency profile . . . 81
2.3 Read 4 files with aio at different depths . . . 82
2.4 Read backwards in a file . . . 82
2.5 Basic verification . . . 83
2.6 Fixed rate submission . . . 83
2.7 Butterfly seek pattern . . . 83

**3 TODO** **85**
3.1 GFIO TODO . . . 85
3.2 Server TODO . . . 86
3.3 Steady State TODO . . . 86

**4 Moral License** **87**

**5 License** **89**

**6 Indices and tables** **97**

**Index** **99**

**Version:** 3.41-49-gde3d

Contents:

CHAPTER

ONE

# FIO - FLEXIBLE I/O TESTER REV. 3.41

## 1.1 Overview and history

Fio was originally written to save me the hassle of writing special test case programs when I wanted to test a specific workload, either for performance reasons or to find/reproduce a bug. The process of writing such a test app can be tiresome, especially if you have to do it often. Hence I needed a tool that would be able to simulate a given I/O workload without resorting to writing a tailored test case again and again.

A test work load is difficult to define, though. There can be any number of processes or threads involved, and they can each be using their own way of generating I/O. You could have someone dirtying large amounts of memory in a memory mapped file, or maybe several threads issuing reads using asynchronous I/O. fio needed to be flexible enough to simulate both of these cases, and many more.

Fio spawns a number of threads or processes doing a particular type of I/O action as specified by the user. fio takes a number of global parameters, each inherited by the thread unless otherwise parameters given to them overriding that setting is given. The typical use of fio is to write a job file matching the I/O load one wants to simulate.

## 1.2 Source

Fio resides in a git repo, the canonical place is:

https://git.kernel.org/pub/scm/linux/kernel/git/axboe/fio

Snapshots are frequently generated and `fio-git-*.tar.gz` include the git meta data as well. Other tarballs are archives of official fio releases. Snapshots can download from:

https://brick.kernel.dk/snaps/

There are also two official mirrors. Both of these are automatically synced with the main repository, when changes are pushed. If the main repo is down for some reason, either one of these is safe to use as a backup:

https://git.kernel.org/pub/scm/linux/kernel/git/axboe/fio.git

https://github.com/axboe/fio.git

## 1.3 Mailing list

The fio project mailing list is meant for anything related to fio including general discussion, bug reporting, questions, and development. For bug reporting, see REPORTING-BUGS.

An automated mail detailing recent commits is automatically sent to the list at most daily. The list address is fio@vger.kernel.org, subscribe by sending an email to fio+subscribe@vger.kernel.org or visit https://subspace.kernel.org/vger.kernel.org.html.

Archives can be found here:

https://www.spinics.net/lists/fio/

or here:

https://lore.kernel.org/fio/

and archives for the old list can be found here:

http://maillist.kernel.dk/fio-devel/

## 1.4 Author

Fio was written by Jens Axboe <axboe@kernel.dk> to enable flexible testing of the Linux I/O subsystem and schedulers. He got tired of writing specific test applications to simulate a given workload, and found that the existing I/O benchmark/test tools out there weren't flexible enough to do what he wanted.

Jens Axboe <axboe@kernel.dk> 20060905

## 1.5 Maintainers

Fio is maintained by Jens Axboe <axboe@kernel.dk and Vincent Fu <vincentfu@gmail.com> - however, for reporting bugs please use the fio reflector or the GitHub page rather than email any of them directly. By using the public resources, others will be able to learn from the responses too. Chances are also good that other members will be able to help with your inquiry as well.

## 1.6 Binary packages

**Debian:**
Starting with Debian "Squeeze", fio packages are part of the official Debian repository. https://packages.debian.org/search?keywords=fio .

**Ubuntu:**
Starting with Ubuntu 10.04 LTS (aka "Lucid Lynx"), fio packages are part of the Ubuntu "universe" repository. https://packages.ubuntu.com/search?keywords=fio .

**Red Hat, Fedora, CentOS & Co:**
Starting with Fedora 9/Extra Packages for Enterprise Linux 4, fio packages are part of the Fedora/EPEL repositories. https://packages.fedoraproject.org/pkgs/fio/ .

**Mandriva:**
Mandriva has integrated fio into their package repository, so installing on that distro should be as easy as typing `urpmi fio`.

**Arch Linux:**
An Arch Linux package is provided under the Community sub-repository: https://www.archlinux.org/packages/?sort=&q=fio

**Solaris:**
Packages for Solaris are available from OpenCSW. Install their pkgutil tool (http://www.opencsw.org/get-it/pkgutil/) and then install fio via `pkgutil -i fio`.

**Windows:**
Beginning with fio 3.31 Windows installers for tagged releases are available on GitHub at https://github.com/axboe/fio/releases. The latest installers for Windows can also be obtained as GitHub Actions artifacts by selecting a build from https://github.com/axboe/fio/actions. These require logging in to a GitHub account.

**BSDs:**
Packages for BSDs may be available from their binary package repositories. Look for a package "fio" using their binary package managers.

## 1.7 Building

Just type:

```
$ ./configure
$ make
$ make install
```

Note that GNU make is required. On BSDs it's available from devel/gmake within ports directory; on Solaris it's in the SUNWgmake package. On platforms where GNU make isn't the default, type `gmake` instead of `make`.

Configure will print the enabled options. Note that on Linux based platforms, the libaio development packages must be installed to use the libaio engine. Depending on the distro, it is usually called libaio-devel or libaio-dev.

For gfio, gtk 2.18 (or newer), associated glib threads, and cairo are required to be installed. gfio isn't built automatically and can be enabled with a `--enable-gfio` option to configure.

To build fio with a cross-compiler:

```
$ make clean
$ make CROSS_COMPILE=/path/to/toolchain/prefix
```

Configure will attempt to determine the target platform automatically.

It's possible to build fio for ESX as well, use the `--esx` switch to configure.

The HTTP engine is enabled depending on if the curl and openssl shared libraries are detected on the system. For Ubuntu, these packages are libcurl4-openssl-dev and libssl-dev.

### 1.7.1 Windows

The minimum versions of Windows for building/running fio are Windows 7/Windows Server 2008 R2. On Windows, Cygwin (https://www.cygwin.com/) is required in order to build fio. To create an MSI installer package install WiX from https://wixtoolset.org and run `dobuild.cmd` from the `os/windows` directory.

How to compile fio on 64-bit Windows:

1. Install Cygwin (https://www.cygwin.com/). Install **make** and all packages starting with **mingw64-x86_64**. Ensure **mingw64-x86_64-zlib** are installed if you wish to enable fio's log compression functionality.
2. Open the Cygwin Terminal.
3. Go to the fio directory (source files).
4. Run `make clean && make -j`.

To build fio for 32-bit Windows, ensure the -i686 versions of the previously mentioned -x86_64 packages are installed and run `./configure --build-32bit-win` before `make`.

It's recommended that once built or installed, fio be run in a Command Prompt or other 'native' console such as console2, since there are known to be display and signal issues when running it under a Cygwin shell (see https://github.com/mintty/mintty/issues/56 and https://github.com/mintty/mintty/wiki/Tips#inputoutput-interaction-with-alien-programs for details).

### 1.7.2 Documentation

Fio uses Sphinx to generate documentation from the reStructuredText files. To build HTML formatted documentation run `make -C doc html` and direct your browser to `./doc/output/html/index.html`. To build manual page run `make -C doc man` and then `man doc/output/man/fio.1`. To see what other output formats are supported run `make -C doc help`.

## 1.8 Platforms

Fio works on (at least) Linux, Solaris, AIX, HP-UX, OSX, NetBSD, OpenBSD, Windows, FreeBSD, and DragonFly. Some features and/or options may only be available on some of the platforms, typically because those features only apply to that platform (like the solarisaio engine, or the splice engine on Linux).

Some features are not available on FreeBSD/Solaris even if they could be implemented, I'd be happy to take patches for that. An example of that is disk utility statistics and (I think) huge page support, support for that does exist in FreeBSD/Solaris.

Fio uses pthread mutexes for signaling and locking and some platforms do not support process shared pthread mutexes. As a result, on such platforms only threads are supported. This could be fixed with sysv ipc locking or other locking alternatives.

Other *BSD platforms are untested, but fio should work there almost out of the box. Since I don't do test runs or even compiles on those platforms, your mileage may vary. Sending me patches for other platforms is greatly appreciated. There's a lot of value in having the same test/benchmark tool available on all platforms.

Note that POSIX aio is not enabled by default on AIX. Messages like these:

```
Symbol resolution failed for /usr/lib/libc.a(posix_aio.o) because:
    Symbol _posix_kaio_rdwr (number 2) is not exported from dependent module /unix.
```

indicate one needs to enable POSIX aio. Run the following commands as root:

```
# lsdev -C -l posix_aio0
    posix_aio0 Defined  Posix Asynchronous I/O
# cfgmgr -l posix_aio0
# lsdev -C -l posix_aio0
    posix_aio0 Available  Posix Asynchronous I/O
```

POSIX aio should work now. To make the change permanent:

```
# chdev -l posix_aio0 -P -a autoconfig='available'
    posix_aio0 changed
```

## 1.9 Running fio

Running fio is normally the easiest part - you just give it the job file (or job files) as parameters:

```
$ fio [options] [jobfile] ...
```

and it will start doing what the *jobfile* tells it to do. You can give more than one job file on the command line, fio will serialize the running of those files. Internally that is the same as using the *stonewall* parameter described in the parameter section.

If the job file contains only one job, you may as well just give the parameters on the command line. The command line parameters are identical to the job parameters, with a few extra that control global parameters. For example, for the job file parameter *iodepth=2*, the mirror command line option would be *--iodepth 2* or *--iodepth=2*. You can also

use the command line for giving more than one job entry. For each `--name` option that fio sees, it will start a new job with that name. Command line entries following a `--name` entry will apply to that job, until there are no more entries or a new `--name` entry is seen. This is similar to the job file options, where each option applies to the current job until a new [] job entry is seen.

fio does not need to run as root, except if the files or devices specified in the job section requires that. Some other options may also be restricted, such as memory locking, I/O scheduler switching, and decreasing the nice value.

If *jobfile* is specified as `-`, the job file will be read from standard input.

## 1.10 How fio works

The first step in getting fio to simulate a desired I/O workload, is writing a job file describing that specific setup. A job file may contain any number of threads and/or files – the typical contents of the job file is a *global* section defining shared parameters, and one or more job sections describing the jobs involved. When run, fio parses this file and sets everything up as described. If we break down a job from top to bottom, it contains the following basic parameters:

*I/O type*

Defines the I/O pattern issued to the file(s). We may only be reading sequentially from this file(s), or we may be writing randomly. Or even mixing reads and writes, sequentially or randomly. Should we be doing buffered I/O, or direct/raw I/O?

*Block size*

In how large chunks are we issuing I/O? This may be a single value, or it may describe a range of block sizes.

*I/O size*

How much data are we going to be reading/writing.

*I/O engine*

How do we issue I/O? We could be memory mapping the file, we could be using regular read/write, we could be using splice, async I/O, or even SG (SCSI generic sg).

*I/O depth*

If the I/O engine is async, how large a queuing depth do we want to maintain?

*Target file/device*

How many files are we spreading the workload over.

*Threads, processes and job synchronization*

How many threads or processes should we spread this workload over.

The above are the basic parameters defined for a workload, in addition there's a multitude of parameters that modify other aspects of how this job behaves.

## 1.11 Command line options

**--debug**=type

Enable verbose tracing *type* of various fio actions. May be `all` for all types or individual types separated by a comma (e.g. `--debug=file,mem` will enable file and memory debugging). Currently, additional logging is available for:

***process***
Dump info related to processes.

***file***
Dump info related to file actions.

***io***
Dump info related to I/O queuing.

***mem***
Dump info related to memory allocations.

***blktrace***
Dump info related to blktrace setup.

***verify***
Dump info related to I/O verification.

***all***
Enable all debug options.

***random***
Dump info related to random offset generation.

***parse***
Dump info related to option matching and parsing.

***diskutil***
Dump info related to disk utilization updates.

***job:x***
Dump info only related to job number x.

***mutex***
Dump info only related to mutex up/down ops.

***profile***
Dump info related to profile extensions.

***time***
Dump info related to internal time keeping.

***net***
Dump info related to networking connections.

***rate***
Dump info related to I/O rate switching.

***compress***
Dump info related to log compress/decompress.

***steadystate***
Dump info related to steadystate detection.

***helperthread***
Dump info related to the helper thread.

***zbd***
Dump info related to support for zoned block devices.

***?*** **or** ***help***
Show available debug options.

`--parse-only`
Parse options only, don't start any I/O.

**--merge-blktrace-only**

Merge blktraces only, don't start any I/O.

**--output**=filename

Write output to file *filename*.

**--output-format**=format

Set the reporting *format* to *normal*, *terse*, *json*, or *json+*. Multiple formats can be selected, separated by a comma. *terse* is a CSV based format. *json+* is like *json*, except it adds a full dump of the latency buckets.

**--bandwidth-log**

Generate aggregate bandwidth logs.

**--minimal**

Print statistics in a terse, semicolon-delimited format.

**--append-terse**

Print statistics in selected mode AND terse, semicolon-delimited format. **Deprecated**, use *--output-format* instead to select multiple formats.

**--terse-version**=version

Set terse *version* output format (default 3, or 2 or 4 or 5).

**--version**

Print version information and exit.

**--help**

Print a summary of the command line options and exit.

**--cpuclock-test**

Perform test and validation of internal CPU clock.

**--crctest**=[test]

Test the speed of the built-in checksumming functions. If no argument is given, all of them are tested. Alternatively, a comma separated list can be passed, in which case the given ones are tested.

**--cmdhelp**=command

Print help information for *command*. May be `all` for all commands.

**--enghelp**=[ioengine[,command]]

List all commands defined by *ioengine*, or print help for *command* defined by *ioengine*. If no *ioengine* is given, list all available ioengines.

**--showcmd**

Convert given job files to a set of command-line options.

**--readonly**

Turn on safety read-only checks, preventing writes and trims. The `--readonly` option is an extra safety guard to prevent users from accidentally starting a write or trim workload when that is not desired. Fio will only modify the device under test if *rw=write/randwrite/rw/randrw/trim/randtrim/trimwrite* is given. This safety net can be used as an extra precaution.

**--eta**=when

Specifies when real-time ETA estimate should be printed. *when* may be *always*, *never* or *auto*. *auto* is the default, it prints ETA when requested if the output is a TTY. *always* disregards the output type, and prints ETA when requested. *never* never prints ETA.

**--eta-interval**=time

By default, fio requests client ETA status roughly every second. With this option, the interval is configurable. Fio imposes a minimum allowed time to avoid flooding the console, less than 250 msec is not supported.

**--eta-newline**=time

Force a new line for every *time* period passed. When the unit is omitted, the value is interpreted in seconds.

**--status-interval**=time

Force a full status dump of cumulative (from job start) values at *time* intervals. This option does *not* provide per-period measurements. So values such as bandwidth are running averages. When the time unit is omitted, *time* is interpreted in seconds. Note that using this option with `--output-format=json` will yield output that technically isn't valid json, since the output will be collated sets of valid json. It will need to be split into valid sets of json after the run.

**--section**=name

Only run specified section *name* in job file. Multiple sections can be specified. The `--section` option allows one to combine related jobs into one file. E.g. one job file could define light, moderate, and heavy sections. Tell fio to run only the "heavy" section by giving `--section=heavy` command line option. One can also specify the "write" operations in one section and "verify" operation in another section. The `--section` option only applies to job sections. The reserved *global* section is always parsed and used.

**--alloc-size**=kb

Allocate additional internal smalloc pools of size *kb* in KiB. The `--alloc-size` option increases shared memory set aside for use by fio. If running large jobs with randommap enabled, fio can run out of memory. Smalloc is an internal allocator for shared structures from a fixed size memory pool and can grow to 16 pools. The pool size defaults to 16MiB.

NOTE: While running `.fio_smalloc.*` backing store files are visible in `/tmp`.

**--warnings-fatal**

All fio parser warnings are fatal, causing fio to exit with an error.

**--max-jobs**=nr

Set the maximum number of threads/processes to support to *nr*. NOTE: On Linux, it may be necessary to increase the shared-memory limit (`/proc/sys/kernel/shmmax`) if fio runs into errors while creating jobs.

**--server**=args

Start a backend server, with *args* specifying what to listen to. See *Client/Server* section.

**--daemonize**=pidfile

Background a fio server, writing the pid to the given *pidfile* file.

**--client**=hostname

Instead of running the jobs locally, send and run them on the given *hostname* or set of *hostname*s. See *Client/Server* section.

**--remote-config**=file

Tell fio server to load this local *file*.

**--idle-prof**=option

Report CPU idleness. *option* is one of the following:

**calibrate**
Run unit work calibration only and exit.

**system**
Show aggregate system idleness and unit work.

**percpu**
As **system** but also show per CPU idleness.

**--inflate-log**=log
Inflate and output compressed *log*.

**--trigger-file**=file
Execute trigger command when *file* exists.

**--trigger-timeout**=time
Execute trigger at this *time*.

**--trigger**=command
Set this *command* as local trigger.

**--trigger-remote**=command
Set this *command* as remote trigger.

**--aux-path**=path
Use the directory specified by *path* for generated state files instead of the current working directory.

Any parameters following the options will be assumed to be job files, unless they match a job file parameter. Multiple job files can be listed and each job file will be regarded as a separate group. Fio will *stonewall* execution between each group.

## 1.12 Job file format

As previously described, fio accepts one or more job files describing what it is supposed to do. The job file format is the classic ini file, where the names enclosed in [] brackets define the job name. You are free to use any ASCII name you want, except *global* which has special meaning. Following the job name is a sequence of zero or more parameters, one per line, that define the behavior of the job. If the first character in a line is a ';' or a '#', the entire line is discarded as a comment.

A *global* section sets defaults for the jobs described in that file. A job may override a *global* section parameter, and a job file may even have several *global* sections if so desired. A job is only affected by a *global* section residing above it.

The *--cmdhelp* option also lists all options. If used with a *command* argument, *--cmdhelp* will detail the given *command*.

See the *examples/* directory for inspiration on how to write job files. Note the copyright and license requirements currently apply to *examples/* files.

So let's look at a really simple job file that defines two processes, each randomly reading from a 128MiB file:

```
; -- start job file --
[global]
rw=randread
size=128m

[job1]

[job2]

; -- end job file --
```

As you can see, the job file sections themselves are empty as all the described parameters are shared. As no *filename* option is given, fio makes up a *filename* for each of the jobs as it sees fit. On the command line, this job would look as follows:

```
$ fio --name=global --rw=randread --size=128m --name=job1 --name=job2
```

Let's look at an example that has a number of processes writing randomly to files:

```
; -- start job file --
[random-writers]
ioengine=libaio
iodepth=4
rw=randwrite
bs=32k
direct=0
size=64m
numjobs=4
; -- end job file --
```

Here we have no *global* section, as we only have one job defined anyway. We want to use async I/O here, with a depth of 4 for each file. We also increased the buffer size used to 32KiB and define numjobs to 4 to fork 4 identical jobs. The result is 4 processes each randomly writing to their own 64MiB file. Instead of using the above job file, you could have given the parameters on the command line. For this case, you would specify:

```
$ fio --name=random-writers --ioengine=libaio --iodepth=4 --rw=randwrite --bs=32k --
↪direct=0 --size=64m --numjobs=4
```

When fio is utilized as a basis of any reasonably large test suite, it might be desirable to share a set of standardized settings across multiple job files. Instead of copy/pasting such settings, any section may pull in an external `filename.fio` file with *include filename* directive, as in the following example:

```
; -- start job file including.fio --
[global]
filename=/tmp/test
filesize=1m
include glob-include.fio

[test]
rw=randread
bs=4k
time_based=1
runtime=10
include test-include.fio
; -- end job file including.fio --
```

```
; -- start job file glob-include.fio --
thread=1
group_reporting=1
; -- end job file glob-include.fio --
```

```
; -- start job file test-include.fio --
ioengine=libaio
iodepth=4
; -- end job file test-include.fio --
```

Settings pulled into a section apply to that section only (except *global* section). Include directives may be nested in that any included file may contain further include directive(s). Include files may not contain [] sections.

### 1.12.1 Environment variables

Fio also supports environment variable expansion in job files. Any sub-string of the form `${VARNAME}` as part of an option value (in other words, on the right of the '='), will be expanded to the value of the environment variable called *VARNAME*. If no such environment variable is defined, or *VARNAME* is the empty string, the empty string will be substituted.

As an example, let's look at a sample fio invocation and job file:

```
$ SIZE=64m NUMJOBS=4 fio jobfile.fio
```

```
; -- start job file --
[random-writers]
rw=randwrite
size=${SIZE}
numjobs=${NUMJOBS}
; -- end job file --
```

This will expand to the following equivalent job file at runtime:

```
; -- start job file --
[random-writers]
rw=randwrite
size=64m
numjobs=4
; -- end job file --
```

Fio ships with a few example job files, you can also look there for inspiration.

### 1.12.2 Reserved keywords

Additionally, fio has a set of reserved keywords that will be replaced internally with the appropriate value. Those keywords are:

**$pagesize**

The architecture page size of the running system.

**$mb_memory**

Megabytes of total memory in the system.

**$ncpus**

Number of online available CPUs.

These can be used on the command line or in the job file, and will be automatically substituted with the current system values when the job is run. Simple math is also supported on these keywords, so you can perform actions like:

```
size=8*$mb_memory
```

and get that properly expanded to 8 times the size of memory in the machine.

## 1.13 Job file parameters

This section describes in details each parameter associated with a job. Some parameters take an option of a given type, such as an integer or a string. Anywhere a numeric value is required, an arithmetic expression may be used, provided it is surrounded by parentheses. Supported operators are:

- addition (+)
- subtraction (-)
- multiplication (*)
- division (/)
- modulus (%)
- exponentiation (^)

For time values in expressions, units are microseconds by default. This is different than for time values not in expressions (not enclosed in parentheses). The following types are used:

### 1.13.1 Parameter types

**str**
: String: A sequence of alphanumeric characters.

**time**
: Integer with possible time suffix. Without a unit value is interpreted as seconds unless otherwise specified. Accepts a suffix of 'd' for days, 'h' for hours, 'm' for minutes, 's' for seconds, 'ms' (or 'msec') for milliseconds and 'us' (or 'usec') for microseconds. For example, use 10m for 10 minutes.

**int**
: Integer. A whole number value, which may contain an integer prefix and an integer suffix:

  [*integer prefix*] **number** [*integer suffix*]

  The optional *integer prefix* specifies the number's base. The default is decimal. *0x* specifies hexadecimal.

  The optional *integer suffix* specifies the number's units, and includes an optional unit prefix and an optional unit. For quantities of data, the default unit is bytes. For quantities of time, the default unit is seconds unless otherwise specified.

  With `kb_base`=1000, fio follows international standards for unit prefixes. To specify power-of-10 decimal values defined in the International System of Units (SI):

  - *K* – means kilo (K) or 1000
  - *M* – means mega (M) or 1000**2
  - *G* – means giga (G) or 1000**3
  - *T* – means tera (T) or 1000**4
  - *P* – means peta (P) or 1000**5

  To specify power-of-2 binary values defined in IEC 80000-13:

  - *Ki* – means kibi (Ki) or 1024
  - *Mi* – means mebi (Mi) or 1024**2
  - *Gi* – means gibi (Gi) or 1024**3
  - *Ti* – means tebi (Ti) or 1024**4
  - *Pi* – means pebi (Pi) or 1024**5

  **For Zone Block Device Mode:**

  - *z* – means Zone

With *kb_base*=1024 (the default), the unit prefixes are opposite from those specified in the SI and IEC 80000-13 standards to provide compatibility with old scripts. For example, 4k means 4096.

For quantities of data, an optional unit of 'B' may be included (e.g., 'kB' is the same as 'k').

The *integer suffix* is not case sensitive (e.g., m/mi mean mebi/mega, not milli). 'b' and 'B' both mean byte, not bit.

Examples with *kb_base*=1000:

- *4 KiB*: 4096, 4096b, 4096B, 4ki, 4kib, 4kiB, 4Ki, 4KiB
- *1 MiB*: 1048576, 1mi, 1024ki
- *1 MB*: 1000000, 1m, 1000k
- *1 TiB*: 1099511627776, 1ti, 1024gi, 1048576mi
- *1 TB*: 1000000000, 1t, 1000m, 1000000k

Examples with *kb_base*=1024 (default):

- *4 KiB*: 4096, 4096b, 4096B, 4k, 4kb, 4kB, 4K, 4KB
- *1 MiB*: 1048576, 1m, 1024k
- *1 MB*: 1000000, 1mi, 1000ki
- *1 TiB*: 1099511627776, 1t, 1024g, 1048576m
- *1 TB*: 1000000000, 1ti, 1000mi, 1000000ki

To specify times (units are not case sensitive):

- *D* – means days
- *H* – means hours
- *M* – means minutes
- *s* – or sec means seconds (default)
- *ms* – or *msec* means milliseconds
- *us* – or *usec* means microseconds

If the option accepts an upper and lower range, use a colon ':' or minus '-' to separate such values. See *irange*. If the lower value specified happens to be larger than the upper value the two values are swapped.

**bool**
Boolean. Usually parsed as an integer, however only defined for true and false (1 and 0).

**irange**
Integer range with suffix. Allows value range to be given, such as 1024-4096. A colon may also be used as the separator, e.g. 1k:4k. If the option allows two sets of ranges, they can be specified with a ',' or '/' delimiter: 1k-4k/8k-32k. Also see *int*.

**float_list**
A list of floating point numbers, separated by a ':' character.

With the above in mind, here follows the complete list of fio job parameters.

### 1.13.2 Units

**kb_base**=int

Select the interpretation of unit prefixes in input parameters.

**1000**
Inputs comply with IEC 80000-13 and the International System of Units (SI). Use:

- power-of-2 values with IEC prefixes (e.g., KiB)
- power-of-10 values with SI prefixes (e.g., kB)

**1024**
Compatibility mode (default). To avoid breaking old scripts:

- power-of-2 values with SI prefixes
- power-of-10 values with IEC prefixes

See *bs* for more details on input parameters.

Outputs always use correct prefixes. Most outputs include both side-by-side, like:

```
bw=2383.3kB/s (2327.4KiB/s)
```

If only one value is reported, then kb_base selects the one to use:

**1000** – SI prefixes

**1024** – IEC prefixes

**unit_base**=int

Base unit for reporting. Allowed values are:

**0**
Use auto-detection (default).

**8**
Byte based.

**1**
Bit based.

### 1.13.3 Job description

**name**=str

ASCII name of the job. This may be used to override the name printed by fio for this job. Otherwise the job name is used. On the command line this parameter has the special purpose of also signaling the start of a new job.

**description**=str

Text description of the job. Doesn't do anything except dump this text description when this job is run. It's not parsed.

**loops**=int

Run the specified number of iterations of this job. Used to repeat the same workload a given number of times. Defaults to 1.

**numjobs**=int

Create the specified number of clones of this job. Each clone of job is spawned as an independent thread or process. May be used to setup a larger number of threads/processes doing the same thing. Each thread is reported

separately; to see statistics for all clones as a whole, use *group_reporting* in conjunction with *new_group*. See *--max-jobs*. Default: 1.

### 1.13.4 Time related parameters

**runtime**=time

Limit runtime. The test will run until it completes the configured I/O workload or until it has run for this specified amount of time, whichever occurs first. It can be quite hard to determine for how long a specified job will run, so this parameter is handy to cap the total runtime to a given time. When the unit is omitted, the value is interpreted in seconds.

**time_based**

If set, fio will run for the duration of the *runtime* specified even if the file(s) are completely read or written. It will simply loop over the same workload as many times as the *runtime* allows.

**startdelay**=irange(time)

Delay the start of job for the specified amount of time. Can be a single value or a range. When given as a range, each thread will choose a value randomly from within the range. Value is in seconds if a unit is omitted.

**ramp_time**=time

If set, fio will run the specified workload for this amount of time before logging any performance numbers. Useful for letting performance settle before logging results, thus minimizing the runtime required for stable results. Note that the ramp_time is considered lead in time for a job, thus it will increase the total runtime if a special timeout or *runtime* is specified. When the unit is omitted, the value is given in seconds.

**clocksource**=str

Use the given clocksource as the base of timing. The supported options are:

**gettimeofday**
*gettimeofday(2)*

**clock_gettime**
*clock_gettime(2)*

**cpu**
Internal CPU clock source

cpu is the preferred clocksource if it is reliable, as it is very fast (and fio is heavy on time calls). Fio will automatically use this clocksource if it's supported and considered reliable on the system it is running on, unless another clocksource is specifically set. For x86/x86-64 CPUs, this means supporting TSC Invariant.

**gtod_reduce**=bool

Enable all of the *gettimeofday(2)* reducing options (*disable_clat*, *disable_slat*, *disable_bw_measurement*) plus reduce precision of the timeout somewhat to really shrink the *gettimeofday(2)* call count. With this option enabled, we only do about 0.4% of the *gettimeofday(2)* calls we would have done if all time keeping was enabled.

**gtod_cpu**=int

Sometimes it's cheaper to dedicate a single thread of execution to just getting the current time. Fio (and databases, for instance) are very intensive on *gettimeofday(2)* calls. With this option, you can set one CPU aside for doing nothing but logging current time to a shared memory location. Then the other threads/processes that run I/O workloads need only copy that segment, instead of entering the kernel with a *gettimeofday(2)* call. The CPU set aside for doing these time calls will be excluded from other uses. Fio will manually clear it from the CPU mask of other jobs.

**job_start_clock_id**=int

The clock_id passed to the call to *clock_gettime* used to record job_start in the *json* output format. Default is 0, or CLOCK_REALTIME.

### 1.13.5 Target file/device

**directory**=str

Prefix filenames with this directory. Used to place files in a different location than `./`. You can specify a number of directories by separating the names with a ':' character. These directories will be assigned equally distributed to job clones created by *numjobs* as long as they are using generated filenames. If specific *filename(s)* are set fio will use the first listed directory, and thereby matching the *filename* semantic (which generates a file for each clone if not specified, but lets all clones use the same file if set).

See the *filename* option for information on how to escape ":" characters within the directory path itself.

Note: To control the directory fio will use for internal state files use *--aux-path*.

**filename**=str

Fio normally makes up a *filename* based on the job name, thread number, and file number (see *filename_format*). If you want to share files between threads in a job or several jobs with fixed file paths, specify a *filename* for each of them to override the default. If the ioengine is file based, you can specify a number of files by separating the names with a ':' colon. So if you wanted a job to open `/dev/sda` and `/dev/sdb` as the two working files, you would use `filename=/dev/sda:/dev/sdb`. This also means that whenever this option is specified, *nrfiles* is ignored. The size of regular files specified by this option will be *size* divided by number of files unless an explicit size is specified by *filesize*.

Each colon in the wanted path must be escaped with a \ character. For instance, if the path is `/dev/dsk/foo@3,0:c` then you would use `filename=/dev/dsk/foo@3,0\:c` and if the path is `F:\filename` then you would use `filename=F\:\filename`.

On Windows, disk devices are accessed as `\.\PhysicalDrive0` for the first device, `\.\PhysicalDrive1` for the second etc. Note: Windows and FreeBSD (refer to geom(4)) prevent write access to areas of the disk containing in-use data (e.g. filesystems).

For HTTP and S3 access, specify a valid URL path or S3 key, respectively. A filename for path-style S3 includes a bucket name (`/bucket/k/e.y`) while a virtual-hosted-style S3 filename `/k/e.y` does not because its bucket name is specified in *http_host*. In both cases, the filename should begin with a `/`. The HTTP engine does not automatically add a leading `/` when constructing URLs from *http_host* and *filename*.

The filename "-" is a reserved name, meaning *stdin* or *stdout*. Which of the two depends on the read/write direction set.

**filename_format**=str

If sharing multiple files between jobs, it is usually necessary to have fio generate the exact names that you want. By default, fio will name a file based on the default file format specification of `jobname.jobnumber.filenumber`. With this option, that can be customized. Fio will recognize and replace the following keywords in this string:

**$jobname**
The name of the worker thread or process.

**$clientuid**
IP of the fio process when using client/server mode.

**$jobnum**
The incremental number of the worker thread or process.

**$filenum**
The incremental number of the file for that worker thread or process.

To have dependent jobs share a set of files, this option can be set to have fio generate filenames that are shared between the two. For instance, if `testfiles.$filenum` is specified, file number 4 for any job will be named `testfiles.4`. The default of `$jobname.$jobnum.$filenum` will be used if no other format specifier is given.

If you specify a path then the directories will be created up to the main directory for the file. So for example if you specify `filename_format=a/b/c/$jobnum` then the directories a/b/c will be created before the file setup part of the job. If you specify *directory* then the path will be relative that directory, otherwise it is treated as the absolute path.

**unique_filename**=bool

To avoid collisions between networked clients, fio defaults to prefixing any generated filenames (with a directory specified) with the source of the client connecting. To disable this behavior, set this option to 0.

**filetype**=str

Assume that all files defined in a job are of this type. By default fio will do *stat(2)* for each file to know its file type. For huge filesets it might be a bottleneck, so the option can be used to skip the huge number of syscalls. The file types are:

**none**
Unset. The default.

**file**
Regular file.

**block**
Block device file.

**char**
Char device file.

**opendir**=str

Recursively open any files below directory *str*. This accepts only a single directory and unlike related options, colons appearing in the path must not be escaped.

**lockfile**=str

Fio defaults to not locking any files before it does I/O to them. If a file or file descriptor is shared, fio can serialize I/O to that file to make the end result consistent. This is usual for emulating real workloads that share files. The lock modes are:

**none**
No locking. The default.

**exclusive**
Only one thread or process may do I/O at a time, excluding all others.

**readwrite**
Read-write locking on the file. Many readers may access the file at the same time, but writes get exclusive access.

**nrfiles**=int

Number of files to use for this job. Defaults to 1. The size of files will be *size* divided by this unless explicit size is specified by *filesize*. Files are created for each thread separately, and each file will have a file number within its name by default, as explained in *filename* section.

**openfiles**=int

Number of files to keep open at the same time. Defaults to the same as *nrfiles*, can be set smaller to limit the number simultaneous opens.

**file_service_type**=str

Defines how fio decides which file from a job to service next. The following types are defined:

**random**
Choose a file at random.

**roundrobin**
Round robin over opened files. This is the default.

**sequential**
Finish one file before moving on to the next. Multiple files can still be open depending on *openfiles*.

**zipf**
Use a *Zipf* distribution to decide what file to access.

**pareto**
Use a *Pareto* distribution to decide what file to access.

**normal**
Use a *Gaussian* (normal) distribution to decide what file to access.

**gauss**
Alias for normal.

For *random*, *roundrobin*, and *sequential*, a postfix can be appended to tell fio how many I/Os to issue before switching to a new file. For example, specifying `file_service_type=random:8` would cause fio to issue 8 I/Os before selecting a new file at random. For the non-uniform distributions, a floating point postfix can be given to influence how the distribution is skewed. See *random_distribution* for a description of how that would work.

**ioscheduler**=str
Attempt to switch the device hosting the file to the specified I/O scheduler before running.

**create_serialize**=bool
If true, serialize the file creation for the jobs. This may be handy to avoid interleaving of data files, which may greatly depend on the filesystem used and even the number of processors in the system. Default: true.

**create_fsync**=bool
*fsync(2)* the data file after creation. This is the default.

**create_on_open**=bool
If true, don't pre-create files but allow the job's open() to create a file when it's time to do I/O. Default: false – pre-create all necessary files when the job starts.

**create_only**=bool
If true, fio will only run the setup phase of the job. If files need to be laid out or updated on disk, only that will be done – the actual job contents are not executed. Default: false.

**allow_file_create**=bool
If true, fio is permitted to create files as part of its workload. If this option is false, then fio will error out if the files it needs to use don't already exist. Default: true.

**allow_mounted_write**=bool
If this isn't set, fio will abort jobs that are destructive (e.g. that write) to what appears to be a mounted device or partition. This should help catch creating inadvertently destructive tests, not realizing that the test will destroy data on the mounted file system. Note that some platforms don't allow writing against a mounted device regardless of this option. Default: false.

**pre_read**=bool
If this is given, files will be pre-read into memory before starting the given I/O operation. This will also clear the *invalidate* flag, since it is pointless to pre-read and then drop the cache. This will only work for I/O engines that are seek-able, since they allow you to read the same data multiple times. Thus it will not work on non-seekable I/O engines (e.g. network, splice). Default: false.

**unlink**=bool

Unlink (delete) the job files when done. Not the default, as repeated runs of that job would then waste time recreating the file set again and again. Default: false.

**unlink_each_loop**=bool

Unlink (delete) job files after each iteration or loop. Default: false.

**zonemode**=str

Accepted values are:

**none**
The *zonerange*, *zonesize*, *zonecapacity* and *zoneskip* parameters are ignored.

**strided**
I/O happens in a single zone until *zonesize* bytes have been transferred. After that number of bytes has been transferred processing of the next zone starts. *zonecapacity* is ignored.

**zbd**
Zoned block device mode. I/O happens sequentially in each zone, even if random I/O has been selected. Random I/O happens across all zones instead of being restricted to a single zone. The *zoneskip* parameter is ignored. *zonerange* and *zonesize* must be identical. Trim is handled using a zone reset operation. Trim only considers non-empty sequential write required and sequential write preferred zones.

**zonerange**=int

Size of a single zone. See also *zonesize* and *zoneskip*.

**zonesize**=int

For *zonemode* =strided, this is the number of bytes to transfer before skipping *zoneskip* bytes. If this parameter is smaller than *zonerange* then only a fraction of each zone with *zonerange* bytes will be accessed. If this parameter is larger than *zonerange* then each zone will be accessed multiple times before skipping to the next zone.

For *zonemode* =zbd, this is the size of a single zone. The *zonerange* parameter is ignored in this mode.

**zonecapacity**=int

For *zonemode* =zbd, this defines the capacity of a single zone, which is the accessible area starting from the zone start address. This parameter only applies when using *zonemode* =zbd in combination with regular block devices. If not specified it defaults to the zone size. If the target device is a zoned block device, the zone capacity is obtained from the device information and this option is ignored.

**zoneskip**=int

For *zonemode* =strided, the number of bytes to skip after *zonesize* bytes of data have been transferred. This parameter must be zero for *zonemode* =zbd.

**read_beyond_wp**=bool

This parameter applies to *zonemode* =zbd only.

Zoned block devices are block devices that consist of multiple zones. Each zone has a type, e.g. conventional or sequential. A conventional zone can be written at any offset that is a multiple of the block size. Sequential zones must be written sequentially. The position at which a write must occur is called the write pointer. A zoned block device can be either drive managed, host managed or host aware. For host managed devices the host must ensure that writes happen sequentially. Fio recognizes host managed devices and serializes writes to sequential zones for these devices.

If a read occurs in a sequential zone beyond the write pointer then the zoned block device will complete the read without reading any data from the storage medium. Since such reads lead to unrealistically high bandwidth and IOPS numbers fio only reads beyond the write pointer if explicitly told to do so. Default: false.

**max_open_zones**=int

When a zone of a zoned block device is partially written (i.e. not all sectors of the zone have been written), the zone is in one of three conditions: 'implicit open', 'explicit open' or 'closed'. Zoned block devices may have a limit called 'max_open_zones' (same name as the parameter) on the total number of zones that can simultaneously be in the 'implicit open' or 'explicit open' conditions. Zoned block devices may have another limit called 'max_active_zones', on the total number of zones that can simultaneously be in the three conditions. The *max_open_zones* parameter limits the number of zones to which write commands are issued by all fio jobs, that is, limits the number of zones that will be in the conditions. When the device has the max_open_zones limit and does not have the max_active_zones limit, the *max_open_zones* parameter limits the number of zones in the two open conditions up to the limit. In this case, fio includes zones in the two open conditions to the write target zones at fio start. When the device has both the max_open_zones and the max_active_zones limits, the *max_open_zones* parameter limits the number of zones in the three conditions up to the limit. In this case, fio includes zones in the three conditions to the write target zones at fio start.

This parameter is relevant only if the *zonemode* =zbd is used. The default value is always equal to the max_open_zones limit of the target zoned block device and a value higher than this limit cannot be specified by users unless the option *ignore_zone_limits* is specified. When *ignore_zone_limits* is specified or the target device does not have the max_open_zones limit, *max_open_zones* can specify 0 to disable any limit on the number of zones that can be simultaneously written to by all jobs.

**job_max_open_zones**=int

In the same manner as *max_open_zones*, limit the number of open zones per fio job, that is, the number of zones that a single job can simultaneously write to. A value of zero indicates no limit. Default: zero.

**ignore_zone_limits**=bool

If this option is used, fio will ignore the maximum number of open zones limit of the zoned block device in use, thus allowing the option *max_open_zones* value to be larger than the device reported limit. Default: false.

**zone_reset_threshold**=float

A number between zero and one that indicates the ratio of written bytes in the zones with write pointers in the IO range to the size of the IO range. When current ratio is above this ratio, zones are reset periodically as *zone_reset_frequency* specifies. If there are multiple jobs when using this option, the IO range for all write jobs has to be the same.

**zone_reset_frequency**=float

A number between zero and one that indicates how often a zone reset should be issued if the zone reset threshold has been exceeded. A zone reset is submitted after each (1 / zone_reset_frequency) write requests. This and the previous parameter can be used to simulate garbage collection activity.

**recover_zbd_write_error**=bool

If this option is specified together with the option *continue_on_error*, check the write pointer positions after the failed writes to sequential write required zones. Then move the write pointers so that the next writes do not fail due to partial writes and unexpected write pointer positions. If *continue_on_error* is not specified, errors out. When the writes are asynchronous, the write pointer move fills blocks with zero then breaks verify data. If an asynchronous IO engine and *verify* workload are specified, errors out. Default: false.

### 1.13.6 I/O type

**direct**=bool

If value is true, use non-buffered I/O. This is usually O_DIRECT. Note that OpenBSD and ZFS on Solaris don't support direct I/O. On Windows the synchronous ioengines don't support direct I/O. Default: false.

**buffered**=bool

If value is true, use buffered I/O. This is the opposite of the *direct* option. Defaults to true.

**readwrite**=str, **rw**=str

Type of I/O pattern. Accepted values are:

**read**
Sequential reads.

**write**
Sequential writes.

**trim**
Sequential trims (Linux block devices and SCSI character devices only).

**randread**
Random reads.

**randwrite**
Random writes.

**randtrim**
Random trims (Linux block devices and SCSI character devices only).

**rw,readwrite**
Sequential mixed reads and writes.

**randrw**
Random mixed reads and writes.

**trimwrite**
Sequential trim+write sequences. Blocks will be trimmed first, then the same blocks will be written to. So if `io_size=64K` is specified, Fio will trim a total of 64K bytes and also write 64K bytes on the same trimmed blocks. This behaviour will be consistent with `number_ios` or other Fio options limiting the total bytes or number of I/O's.

**randtrimwrite**
Like trimwrite, but uses random offsets rather than sequential writes.

Fio defaults to read if the option is not specified. For the mixed I/O types, the default is to split them 50/50. For certain types of I/O the result may still be skewed a bit, since the speed may be different.

It is possible to specify the number of I/Os to do before getting a new offset by appending `:<nr>` to the end of the string given. For a random read, it would look like `rw=randread:8` for passing in an offset modifier with a value of 8. If the suffix is used with a sequential I/O pattern, then the *<nr>* value specified will be **added** to the generated offset for each I/O turning sequential I/O into sequential I/O with holes. For instance, using `rw=write:4k` will skip 4k for every write. Also see the *rw_sequencer* option. If this is used with *verify* then *verify_header_seed* will be disabled, unless its explicitly enabled.

**rw_sequencer**=str

If an offset modifier is given by appending a number to the `rw=<str>` line, then this option controls how that number modifies the I/O offset being generated. Accepted values are:

**sequential**
Generate sequential offset.

**identical**
Generate the same offset.

`sequential` is only useful for random I/O, where fio would normally generate a new random offset for every I/O. If you append e.g. 8 to randread, i.e. `rw=randread:8` you would get a new random offset for every 8 I/Os. The result would be a sequence of 8 sequential offsets with a random starting point. However this behavior may change if a sequential I/O reaches end of the file. As sequential I/O is already sequential, setting `sequential`

for that would not result in any difference. `identical` behaves in a similar fashion, except it sends the same offset 8 number of times before generating a new offset.

Example #1:

```
rw=randread:8
rw_sequencer=sequential
bs=4k
```

The generated sequence of offsets will look like this: 4k, 8k, 12k, 16k, 20k, 24k, 28k, 32k, 92k, 96k, 100k, 104k, 108k, 112k, 116k, 120k, 48k, 52k ...

Example #2:

```
rw=randread:8
rw_sequencer=identical
bs=4k
```

The generated sequence of offsets will look like this: 4k, 4k, 4k, 4k, 4k, 4k, 4k, 4k, 92k, 92k, 92k, 92k, 92k, 92k, 92k, 92k, 48k, 48k, 48k ...

**unified_rw_reporting**=str

Fio normally reports statistics on a per data direction basis, meaning that reads, writes, and trims are accounted and reported separately. This option determines whether fio reports the results normally, summed together, or as both options. Accepted values are:

**none**
: Normal statistics reporting.

**mixed**
: Statistics are summed per data direction and reported together.

**both**
: Statistics are reported normally, followed by the mixed statistics.

**0**
: Backward-compatible alias for **none**.

**1**
: Backward-compatible alias for **mixed**.

**2**
: Alias for **both**.

**randrepeat**=bool

Seed all random number generators in a predictable way so the pattern is repeatable across runs. Default: true.

**allrandrepeat**=bool

Alias for *randrepeat*. Default: true.

**randseed**=int

Seed the random number generators based on this seed value, to be able to control what sequence of output is being generated. If not set, the random sequence depends on the *randrepeat* setting.

**fallocate**=str

Whether pre-allocation is performed when laying down files. Accepted values are:

**none**
: Do not pre-allocate space.

**native**
Use a platform's native pre-allocation call but fall back to **none** behavior if it fails/is not implemented.

**posix**
Pre-allocate via *posix_fallocate(3)*.

**keep**
Pre-allocate via *fallocate(2)* with FALLOC_FL_KEEP_SIZE set.

**truncate**
Extend file to final size via *ftruncate(2)* instead of allocating.

**0**
Backward-compatible alias for **none**.

**1**
Backward-compatible alias for **posix**.

May not be available on all supported platforms. **keep** is only available on Linux. If using ZFS on Solaris this cannot be set to **posix** because ZFS doesn't support pre-allocation. Default: **native** if any pre-allocation methods except **truncate** are available, **none** if not.

Note that using **truncate** on Windows will interact surprisingly with non-sequential write patterns. When writing to a file that has been extended by setting the end-of-file information, Windows will backfill the unwritten portion of the file up to that offset with zeroes before issuing the new write. This means that a single small write to the end of an extended file will stall until the entire file has been filled with zeroes.

`fadvise_hint`=`str`

Use *posix_fadvise(2)* or *posix_fadvise(2)* to advise the kernel on what I/O patterns are likely to be issued. Accepted values are:

**0**
Backwards-compatible hint for "no hint".

**1**
Backwards compatible hint for "advise with fio workload type". This uses **FADV_RANDOM** for a random workload, and **FADV_SEQUENTIAL** for a sequential workload.

**sequential**
Advise using **FADV_SEQUENTIAL**.

**random**
Advise using **FADV_RANDOM**.

**noreuse**
Advise using **FADV_NOREUSE**. This may be a no-op on older Linux kernels. Since Linux 6.3, it provides a hint to the LRU algorithm. See the *posix_fadvise(2)* man page.

`write_hint`=`str`

Use *fcntl(2)* to advise the kernel what life time to expect from a write. Only supported on Linux, as of version 4.13. Accepted values are:

**none**
No particular life time associated with this file.

**short**
Data written to this file has a short life time.

**medium**
Data written to this file has a medium life time.

**long**
Data written to this file has a long life time.

**extreme**
Data written to this file has a very long life time.

The values are all relative to each other, and no absolute meaning should be associated with them.

**offset**=int

Start I/O at the provided offset in the file, given as either a fixed size in bytes, zones or a percentage. If a percentage is given, the generated offset will be aligned to the minimum `blocksize` or to the value of `offset_align` if provided. Data before the given offset will not be touched. This effectively caps the file size at *real_size - offset*. Can be combined with *size* to constrain the start and end range of the I/O workload. A percentage can be specified by a number between 1 and 100 followed by '%', for example, `offset=20%` to specify 20%. In ZBD mode, value can be set as number of zones using 'z'.

**offset_align**=int

If set to non-zero value, the byte offset generated by a percentage `offset` is aligned upwards to this value. Defaults to 0 meaning that a percentage offset is aligned to the minimum block size.

**offset_increment**=int

If this is provided, then the real offset becomes *offset* + *offset_increment* * *thread_number*, where the thread number is a counter that starts at 0 and is incremented for each sub-job (i.e. when *numjobs* option is specified). This option is useful if there are several jobs which are intended to operate on a file in parallel disjoint segments, with even spacing between the starting points. Percentages can be used for this option. If a percentage is given, the generated offset will be aligned to the minimum `blocksize` or to the value of `offset_align` if provided. In ZBD mode, value can also be set as number of zones using 'z'.

**number_ios**=int

Fio will normally perform I/Os until it has exhausted the size of the region set by *size*, or if it exhaust the allocated time (or hits an error condition). With this setting, the range/size can be set independently of the number of I/Os to perform. When fio reaches this number, it will exit normally and report status. Note that this does not extend the amount of I/O that will be done, it will only stop fio if this condition is met before other end-of-job criteria.

**fsync**=int

If writing to a file, issue an *fsync(2)* (or its equivalent) of the dirty data for every number of blocks given. For example, if you give 32 as a parameter, fio will sync the file after every 32 writes issued. If fio is using non-buffered I/O, we may not sync the file. The exception is the sg I/O engine, which synchronizes the disk cache anyway. Defaults to 0, which means fio does not periodically issue and wait for a sync to complete. Also see *end_fsync* and *fsync_on_close*.

**fdatasync**=int

Like *fsync* but uses *fdatasync(2)* to only sync data and not metadata blocks. In Windows, DragonFlyBSD or OSX there is no *fdatasync(2)* so this falls back to using *fsync(2)*. Defaults to 0, which means fio does not periodically issue and wait for a data-only sync to complete.

**write_barrier**=int

Make every *N-th* write a barrier write.

**sync_file_range**=str:int

Use *sync_file_range(2)* for every *int* number of write operations. Fio will track range of writes that have happened since the last *sync_file_range(2)* call. *str* can currently be one or more of:

**wait_before**
SYNC_FILE_RANGE_WAIT_BEFORE

**write**
SYNC_FILE_RANGE_WRITE

**wait_after**
SYNC_FILE_RANGE_WAIT_AFTER

So if you do `sync_file_range=wait_before,write:8`, fio would use `SYNC_FILE_RANGE_WAIT_BEFORE | SYNC_FILE_RANGE_WRITE` for every 8 writes. Also see the *sync_file_range(2)* man page. This option is Linux specific.

**overwrite**=bool

If true, writes to a file will always overwrite existing data. If the file doesn't already exist, it will be created before the write phase begins. If the file exists and is large enough for the specified write phase, nothing will be done. Default: false.

**end_fsync**=bool

If true, *fsync(2)* file contents when a write stage has completed. Default: false.

**fsync_on_close**=bool

If true, fio will *fsync(2)* a dirty file on close. This differs from *end_fsync* in that it will happen on every file close, not just at the end of the job. Default: false.

**rwmixread**=int

Percentage of a mixed workload that should be reads. Default: 50.

**rwmixwrite**=int

Percentage of a mixed workload that should be writes. If both *rwmixread* and *rwmixwrite* is given and the values do not add up to 100%, the latter of the two will be used to override the first. This may interfere with a given rate setting, if fio is asked to limit reads or writes to a certain rate. If that is the case, then the distribution may be skewed. Default: 50.

**random_distribution**=str:float[:float][,str:float][,str:float]

By default, fio will use a completely uniform random distribution when asked to perform random I/O. Sometimes it is useful to skew the distribution in specific ways, ensuring that some parts of the data is more hot than others. fio includes the following distribution models:

**random**
Uniform random distribution

**zipf**
Zipf distribution

**pareto**
Pareto distribution

**normal**
Normal (Gaussian) distribution

**zoned**
Zoned random distribution

**zoned_abs**
Zone absolute random distribution

When using a **zipf** or **pareto** distribution, an input value is also needed to define the access pattern. For **zipf**, this is the *Zipf theta*. For **pareto**, it's the *Pareto power*. Fio includes a test program, **fio-genzipf**, that can be used visualize what the given input values will yield in terms of hit rates. If you wanted to use **zipf** with a *theta* of 1.2, you would use `random_distribution=zipf:1.2` as the option. If a non-uniform model is used, fio will disable use of the random map. For the **normal** distribution, a normal (Gaussian) deviation is supplied as a value between 0 and 100.

The second, optional float is allowed for **pareto**, **zipf** and **normal** distributions. It allows one to set base of distribution in non-default place, giving more control over most probable outcome. This value is in range [0-1] which maps linearly to range of possible random values. Defaults are: random for **pareto** and **zipf**, and 0.5 for **normal**. If you wanted to use **zipf** with a *theta* of 1.2 centered on 1/4 of allowed value range, you would use `random_distribution=zipf:1.2:0.25`.

For a **zoned** distribution, fio supports specifying percentages of I/O access that should fall within what range of the file or device. For example, given a criteria of:

- 60% of accesses should be to the first 10%
- 30% of accesses should be to the next 20%
- 8% of accesses should be to the next 30%
- 2% of accesses should be to the next 40%

we can define that through zoning of the random accesses. For the above example, the user would do:

```
random_distribution=zoned:60/10:30/20:8/30:2/40
```

A **zoned_abs** distribution works exactly like the **zoned**, except that it takes absolute sizes. For example, let's say you wanted to define access according to the following criteria:

- 60% of accesses should be to the first 20G
- 30% of accesses should be to the next 100G
- 10% of accesses should be to the next 500G

we can define an absolute zoning distribution with:

    random_distribution=zoned_abs=60/20G:30/100G:10/500g

For both **zoned** and **zoned_abs**, fio supports defining up to 256 separate zones.

Similarly to how *bssplit* works for setting ranges and percentages of block sizes. Like *bssplit*, it's possible to specify separate zones for reads, writes, and trims. If just one set is given, it'll apply to all of them. This goes for both **zoned zoned_abs** distributions.

**percentage_random**=int[,int][,int]

For a random workload, set how big a percentage should be random. This defaults to 100%, in which case the workload is fully random. It can be set from anywhere from 0 to 100. Setting it to 0 would make the workload fully sequential. Any setting in between will result in a random mix of sequential and random I/O, at the given percentages. Comma-separated values may be specified for reads, writes, and trims as described in *blocksize*.

**norandommap**

Normally fio will cover every block of the file when doing random I/O. If this option is given, fio will just get a new random offset without looking at past I/O history. This means that some blocks may not be read or written, and that some blocks may be read/written more than once. If this option is used with *verify* then *verify_header_seed* will be disabled. If this option is used with *verify* and multiple block-sizes (via *bsrange*), only intact blocks are verified, i.e., partially-overwritten blocks are ignored. With an async I/O engine and an I/O depth > 1, header write sequence number verification will be disabled. See *verify_write_sequence*.

**softrandommap**=bool

See *norandommap*. If fio runs with the random block map enabled and it fails to allocate the map, if this option is set it will continue without a random block map. As coverage will not be as complete as with random maps, this option is disabled by default.

**random_generator**=str

Fio supports the following engines for generating I/O offsets for random I/O:

**tausworthe**
Strong 2^88 cycle random number generator.

**lfsr**
Linear feedback shift register generator.

**tausworthe64**
Strong 64-bit 2^258 cycle random number generator.

**tausworthe** is a strong random number generator, but it requires tracking on the side if we want to ensure that blocks are only read or written once. **lfsr** guarantees that we never generate the same offset twice, and it's also less computationally expensive. It's not a true random generator, however, though for I/O purposes it's typically good enough. **lfsr** only works with single block sizes, not with workloads that use multiple block sizes. If used with such a workload, fio may read or write some blocks multiple times. The default value is **tausworthe**, unless the required space exceeds 2^32 blocks. If it does, then **tausworthe64** is selected automatically.

**sprandom**=bool

SPRandom is a method designed to rapidly precondition SSDs for steady-state random write workloads. It divides the device into equally sized regions and writes the device's entire physical capacity once, selecting offsets so that the regions have a distribution of invalid blocks matching the distribution that occurs at steady state.

Default: false.

It uses **random_generator=lfsr**, which fio will set by default. Selecting any other random generator will result in an error.

**spr_num_regions**=int

See *sprandom*. Specifies the number of regions used for SPRandom. For large devices it is better to use more regions, to increase precision and reduce memory allocation. The allocation is proportional to the region size.

Default=100

**spr_op**=float

See *sprandom*. Over-provisioning ratio in the range (0, 1), as specified by the SSD manufacturer.

Default=0.15

### 1.13.7 Block size

**blocksize**=int[,int][,int], **bs**=int[,int][,int]

The block size in bytes used for I/O units. Default: 4096. A single value applies to reads, writes, and trims. Comma-separated values may be specified for reads, writes, and trims. A value not terminated in a comma applies to subsequent types.

Examples:

**bs=256k**
means 256k for reads, writes and trims.

**bs=8k,32k**
means 8k for reads, 32k for writes and trims.

**bs=8k,32k,**
means 8k for reads, 32k for writes, and default for trims.

**bs=,8k**
means default for reads, 8k for writes and trims.

**bs=,8k,**
means default for reads, 8k for writes, and default for trims.

**blocksize_range**=irange[,irange][,irange], **bsrange**=irange[,irange][,irange]

A range of block sizes in bytes for I/O units. The issued I/O unit will always be a multiple of the minimum size, unless *blocksize_unaligned* is set.

Comma-separated ranges may be specified for reads, writes, and trims as described in *blocksize*.

Example: `bsrange=1k-4k,2k-8k` also the ':' delimiter `bsrange=1k:4k,2k:8k`.

**bssplit**=str[,str][,str]

Sometimes you want even finer grained control of the block sizes issued, not just an even split between them. This option allows you to weight various block sizes, so that you are able to define a specific amount of block sizes issued. The format for this option is:

```
bssplit=blocksize/percentage:blocksize/percentage
```

for as many block sizes as needed. So if you want to define a workload that has 50% 64k blocks, 10% 4k blocks, and 40% 32k blocks, you would write:

```
bssplit=4k/10:64k/50:32k/40
```

Ordering does not matter. If the percentage is left blank, fio will fill in the remaining values evenly. So a bssplit option like this one:

```
bssplit=4k/50:1k/:32k/
```

would have 50% 4k ios, and 25% 1k and 32k ios. The percentages always add up to 100, if bssplit is given a range that adds up to more, it will error out.

Comma-separated values may be specified for reads, writes, and trims as described in *blocksize*.

If you want a workload that has 50% 2k reads and 50% 4k reads, while having 90% 4k writes and 10% 8k writes, you would specify:

```
bssplit=2k/50:4k/50,4k/90:8k/10
```

Fio supports defining up to 64 different weights for each data direction.

**blocksize_unaligned, bs_unaligned**

If set, fio will issue I/O units with any size within *blocksize_range*, not just multiples of the minimum size. This typically won't work with direct I/O, as that normally requires sector alignment.

**bs_is_seq_rand**=bool

If this option is set, fio will use the normal read,write blocksize settings as sequential,random blocksize settings instead. Any random read or write will use the WRITE blocksize settings, and any sequential read or write will use the READ blocksize settings.

**blockalign**=int[,int][,int], **ba**=int[,int][,int]

Boundary to which fio will align random I/O units. Default: *blocksize*. Minimum alignment is typically 512b for using direct I/O, though it usually depends on the hardware block size. This option is mutually exclusive with using a random map for files, so it will turn off that option. Comma-separated values may be specified for reads, writes, and trims as described in *blocksize*.

### 1.13.8 Buffers and memory

**zero_buffers**

Initialize buffers with all zeros. Default: fill buffers with random data.

**refill_buffers**

If this option is given, fio will refill the I/O buffers on every submit. Only makes sense if *zero_buffers* isn't specified, naturally. Defaults to being unset i.e., the buffer is only filled at init time and the data in it is reused when possible but if any of *verify*, *buffer_compress_percentage* or *dedupe_percentage* are enabled then *refill_buffers* is also automatically enabled.

**scramble_buffers**=bool

If *refill_buffers* is too costly and the target is using data deduplication, then setting this option will slightly modify the I/O buffer contents to defeat normal de-dupe attempts. This is not enough to defeat more clever block compression attempts, but it will stop naive dedupe of blocks. Default: true.

**buffer_compress_percentage**=int

If this is set, then fio will attempt to provide I/O buffer content (on WRITEs) that compresses to the specified level. Fio does this by providing a mix of random data followed by fixed pattern data. The fixed pattern is either zeros, or the pattern specified by *buffer_pattern*. If the *buffer_pattern* option is used, it might skew the compression ratio slightly. Setting *buffer_compress_percentage* to a value other than 100 will also enable *refill_buffers* in order to reduce the likelihood that adjacent blocks are so similar that they over compress when seen together. See *buffer_compress_chunk* for how to set a finer or coarser granularity for the random/fixed data region. Defaults to unset i.e., buffer data will not adhere to any compression level.

**buffer_compress_chunk**=int

This setting allows fio to manage how big the random/fixed data region is when using *buffer_compress_percentage*. When *buffer_compress_chunk* is set to some non-zero value smaller than the block size, fio can repeat the random/fixed region throughout the I/O buffer at the specified interval (which particularly useful when bigger block sizes are used for a job). When set to 0, fio will use a chunk size that matches the block size resulting in a single random/fixed region within the I/O buffer. Defaults to 512. When the unit is omitted, the value is interpreted in bytes.

**buffer_pattern**=str

If set, fio will fill the I/O buffers with this pattern or with the contents of a file. If not set, the contents of I/O buffers are defined by the other options related to buffer contents. The setting can be any pattern of bytes, and can be prefixed with 0x for hex values. It may also be a string, where the string must then be wrapped with "". Or it may also be a filename, where the filename must be wrapped with '' in which case the file is opened and read. Note that not all the file contents will be read if that would cause the buffers to overflow. So, for example:

```
buffer_pattern='filename'
```

or:

```
buffer_pattern="abcd"
```

or:

```
buffer_pattern=-12
```

or:

```
buffer_pattern=0xdeadface
```

Also you can combine everything together in any order:

```
buffer_pattern=0xdeadface"abcd"-12'filename'
```

**dedupe_percentage**=int

If set, fio will generate this percentage of identical buffers when writing. These buffers will be naturally dedupable. The contents of the buffers depend on what other buffer compression settings have been set. It's possible to have the individual buffers either fully compressible, or not at all – this option only controls the distribution of unique buffers. Setting this option will also enable *refill_buffers* to prevent every buffer being identical.

**dedupe_mode**=str

If dedupe_percentage=<int> is given, then this option controls how fio generates the dedupe buffers.

**repeat**
Generate dedupe buffers by repeating previous writes

**working_set**
Generate dedupe buffers from working set

repeat is the default option for fio. Dedupe buffers are generated by repeating previous unique write.

working_set is a more realistic workload. With working_set, dedupe_working_set_percentage=<int> should be provided. Given that, fio will use the initial unique write buffers as its working set. Upon deciding to dedupe, fio will randomly choose a buffer from the working set. Note that by using working_set the dedupe percentage will converge to the desired over time while repeat maintains the desired percentage throughout the job.

**dedupe_working_set_percentage**=int

If dedupe_mode=<str> is set to working_set, then this controls the percentage of size of the file or device used as the buffers fio will choose to generate the dedupe buffers from

Note that size needs to be explicitly provided and only 1 file per job is supported

**dedupe_global**=bool

This controls whether the deduplication buffers will be shared amongst all jobs that have this option set. The buffers are spread evenly between participating jobs.

**invalidate**=bool

Invalidate the buffer/page cache parts of the files to be used prior to starting I/O if the platform and file type support it. Defaults to true. This will be ignored if *pre_read* is also specified for the same job.

**sync**=str

Whether, and what type, of synchronous I/O to use for writes. The allowed values are:

**none**
Do not use synchronous IO, the default.

**0**
Same as **none**.

**sync**
Use synchronous file IO. For the majority of I/O engines, this means using O_SYNC.

**1**
Same as **sync**.

**dsync**
Use synchronous data IO. For the majority of I/O engines, this means using O_DSYNC.

**iomem**=str, **mem**=str

Fio can use various types of memory as the I/O unit buffer. The allowed values are:

**malloc**
Use memory from *malloc(3)* as the buffers. Default memory type.

**shm**
Use shared memory as the buffers. Allocated through *shmget(2)*.

**shmhuge**
Same as shm, but use huge pages as backing.

**mmap**
Use *mmap(2)* to allocate buffers. May either be anonymous memory, or can be file backed if a filename is given after the option. The format is *mem=mmap:/path/to/file*.

**mmaphuge**
Use a memory mapped huge file as the buffer backing. Append filename after mmaphuge, ala *mem=mmaphuge:/hugetlbfs/file*.

**mmapshared**
Same as mmap, but use a MMAP_SHARED mapping.

**cudamalloc**
Use GPU memory as the buffers for GPUDirect RDMA benchmark. The *ioengine* must be *rdma*.

The area allocated is a function of the maximum allowed bs size for the job, multiplied by the I/O depth given. Note that for **shmhuge** and **mmaphuge** to work, the system must have free huge pages allocated. This can normally be checked and set by reading/writing `/proc/sys/vm/nr_hugepages` on a Linux system. Fio assumes a huge page is 2 or 4MiB in size depending on the platform. So to calculate the number of huge pages you need for a given job file, add up the I/O depth of all jobs (normally one unless *iodepth* is used) and multiply by the maximum bs set. Then divide that number by the huge page size. You can see the size of the huge pages in `/proc/meminfo`. If no huge pages are allocated by having a non-zero number in *nr_hugepages*, using **mmaphuge** or **shmhuge** will fail. Also see *hugepage-size*.

**mmaphuge** also needs to have hugetlbfs mounted and the file location should point there. So if it's mounted in `/huge`, you would use *mem=mmaphuge:/huge/somefile*.

**iomem_align**=int, **mem_align**=int
This indicates the memory alignment of the I/O memory buffers. Note that the given alignment is applied to the first I/O unit buffer, if using *iodepth* the alignment of the following buffers are given by the *bs* used. In other words, if using a *bs* that is a multiple of the page sized in the system, all buffers will be aligned to this value. If using a *bs* that is not page aligned, the alignment of subsequent I/O memory buffers is the sum of the *iomem_align* and *bs* used.

**hugepage-size**=int
Defines the size of a huge page. Must at least be equal to the system setting, see `/proc/meminfo` and `/sys/kernel/mm/hugepages/`. Defaults to 2 or 4MiB depending on the platform. Should probably always be a multiple of megabytes, so using `hugepage-size=Xm` is the preferred way to set this to avoid setting a non-pow-2 bad value.

**lockmem**=int
Pin the specified amount of memory with *mlock(2)*. Can be used to simulate a smaller amount of memory. The amount specified is per worker.

### 1.13.9 I/O size

**size**=int
The total size of file I/O for each thread of this job. Fio will run until this many bytes has been transferred, unless runtime is altered by other means such as (1) *runtime*, (2) *io_size* (3) *number_ios*, (4) gaps/holes while doing I/O's such as `rw=read:16K`, or (5) sequential I/O reaching end of the file which is possible when

*percentage_random* is less than 100. Fio will divide this size between the available files determined by options such as *nrfiles*, *filename*, unless *filesize* is specified by the job. If the result of division happens to be 0, the size is set to the physical size of the given files or devices if they exist. If this option is not specified, fio will use the full size of the given files or devices. If the files do not exist, size must be given. It is also possible to give size as a percentage between 1 and 100. If `size=20%` is given, fio will use 20% of the full size of the given files or devices. In ZBD mode, value can also be set as number of zones using 'z'. Can be combined with *offset* to constrain the start and end range that I/O will be done within.

**io_size**=int, **io_limit**=int

Normally fio operates within the region set by *size*, which means that the *size* option sets both the region and size of I/O to be performed. Sometimes that is not what you want. With this option, it is possible to define just the amount of I/O that fio should do. For instance, if *size* is set to 20GiB and *io_size* is set to 5GiB, fio will perform I/O within the first 20GiB but exit when 5GiB have been done. The opposite is also possible – if *size* is set to 20GiB, and *io_size* is set to 40GiB, then fio will do 40GiB of I/O within the 0..20GiB region.

**filesize**=irange(int)

Individual file sizes. May be a range, in which case fio will select sizes for files at random within the given range. If not given, each created file is the same size. This option overrides *size* in terms of file size, i.e. if *filesize* is specified then *size* becomes merely the default for *io_size* and has no effect at all if *io_size* is set explicitly.

**file_append**=bool

Perform I/O after the end of the file. Normally fio will operate within the size of a file. If this option is set, then fio will append to the file instead. This has identical behavior to setting *offset* to the size of a file. This option is ignored on non-regular files.

**fill_device**=bool, **fill_fs**=bool

Sets size to something really large and waits for ENOSPC (no space left on device) or EDQUOT (disk quota exceeded) as the terminating condition. Only makes sense with sequential write. For a read workload, the mount point will be filled first then I/O started on the result. This option doesn't make sense if operating on a raw device node, since the size of that is already known by the file system. Additionally, writing beyond end-of-device will not return ENOSPC there.

### 1.13.10 I/O engine

**ioengine**=str

fio supports 2 kinds of performance measurement: I/O and file/directory operation.

I/O engines define how the job issues I/O to the file. The following types are defined:

**sync**
Basic *read(2)* or *write(2)* I/O. *lseek(2)* is used to position the I/O location. See *fsync* and *fdatasync* for syncing write I/Os.

**psync**
Basic *pread(2)* or *pwrite(2)* I/O. Default on all supported operating systems except for Windows.

**vsync**
Basic *readv(2)* or *writev(2)* I/O. Will emulate queuing by coalescing adjacent I/Os into a single submission.

**pvsync**
Basic *preadv(2)* or *pwritev(2)* I/O.

**pvsync2**
Basic *preadv2(2)* or *pwritev2(2)* I/O.

**io_uring**
Fast Linux native asynchronous I/O. Supports async IO for both direct and buffered IO. This engine defines engine specific options.

**io_uring_cmd**
Fast Linux native asynchronous I/O for pass through commands. This engine defines engine specific options.

**libaio**
Linux native asynchronous I/O. Note that Linux may only support queued behavior with non-buffered I/O (set `direct=1` or `buffered=0`). This engine defines engine specific options.

**posixaio**
POSIX asynchronous I/O using *aio_read(3)* and *aio_write(3)*.

**solarisaio**
Solaris native asynchronous I/O.

**windowsaio**
Windows native asynchronous I/O. Default on Windows.

**mmap**
File is memory mapped with *mmap(2)* and data copied to/from using *memcpy(3)*.

**splice**
*splice(2)* is used to transfer the data and *vmsplice(2)* to transfer data from user space to the kernel.

**sg**
SCSI generic sg v3 I/O. May either be synchronous using the SG_IO ioctl, or if the target is an sg character device we use *read(2)* and *write(2)* for asynchronous I/O. Requires *filename* option to specify either block or character devices. This engine supports trim operations. The sg engine includes engine specific options.

**libzbc**
Read, write, trim and ZBC/ZAC operations to a zoned block device using libzbc library. The target can be either an SG character device or a block device file.

**null**
Doesn't transfer any data, just pretends to. This is mainly used to exercise fio itself and for debugging/testing purposes.

**net**
Transfer over the network to given `host:port`. Depending on the *protocol* used, the *hostname*, *port*, *listen* and *filename* options are used to specify what sort of connection to make, while the *protocol* option determines which protocol will be used. This engine defines engine specific options.

**netsplice**
Like **net**, but uses *splice(2)* and *vmsplice(2)* to map data and send/receive. This engine defines engine specific options.

**cpuio**
Doesn't transfer any data, but burns CPU cycles according to the *cpuload*, *cpuchunks* and *cpumode* options. Setting *cpuload*=85 will cause that job to do nothing but burn 85% of the CPU. In case of SMP machines, use *numjobs*=<nr_of_cpu> to get desired CPU usage, as the cpuload only loads a single CPU at the desired rate. A job never finishes unless there is at least one non-cpuio job. Setting *cpumode*=qsort replace the default noop instructions loop by a qsort algorithm to consume more energy.

**rdma**
The RDMA I/O engine supports both RDMA memory semantics (RDMA_WRITE/RDMA_READ) and channel semantics (Send/Recv) for the InfiniBand, RoCE and iWARP protocols. This engine defines engine specific options.

**falloc**
I/O engine that does regular fallocate to simulate data transfer as fio ioengine.

**DDIR_READ**
does fallocate(,mode = FALLOC_FL_KEEP_SIZE,).

**DDIR_WRITE**
does fallocate(,mode = 0).

**DDIR_TRIM**
does fallocate(,mode = FALLOC_FL_KEEP_SIZE|FALLOC_FL_PUNCH_HOLE).

**ftruncate**
I/O engine that sends *ftruncate(2)* operations in response to write (DDIR_WRITE) events. Each ftruncate issued sets the file's size to the current block offset. *blocksize* is ignored.

**e4defrag**
I/O engine that does regular EXT4_IOC_MOVE_EXT ioctls to simulate defragment activity in request to DDIR_WRITE event.

**rados**
I/O engine supporting direct access to Ceph Reliable Autonomic Distributed Object Store (RADOS) via librados. This ioengine defines engine specific options.

**rbd**
I/O engine supporting direct access to Ceph Rados Block Devices (RBD) via librbd without the need to use the kernel rbd driver. This ioengine defines engine specific options.

**http**
I/O engine supporting GET/PUT requests over HTTP(S) with libcurl to a WebDAV or S3 endpoint. This ioengine defines engine specific options.

This engine only supports direct IO of iodepth=1; you need to scale this via numjobs. blocksize defines the size of the objects to be created.

TRIM is translated to object deletion.

**gfapi**
Using GlusterFS libgfapi sync interface to direct access to GlusterFS volumes without having to go through FUSE. This ioengine defines engine specific options.

**gfapi_async**
Using GlusterFS libgfapi async interface to direct access to GlusterFS volumes without having to go through FUSE. This ioengine defines engine specific options.

**libhdfs**
Read and write through Hadoop (HDFS). The *filename* option is used to specify host,port of the hdfs name-node to connect. This engine interprets offsets a little differently. In HDFS, files once created cannot be modified so random writes are not possible. To imitate this the libhdfs engine expects a bunch of small files to be created over HDFS and will randomly pick a file from them based on the offset generated by fio backend (see the example job file to create such files, use `rw=write` option). Please note, it may be necessary to set environment variables to work with HDFS/libhdfs properly. Each job uses its own connection to HDFS.

**mtd**
Read, write and erase an MTD character device (e.g., `/dev/mtd0`). Discards are treated as erases.

Depending on the underlying device type, the I/O may have to go in a certain pattern, e.g., on NAND, writing sequentially to erase blocks and discarding before overwriting. The *trimwrite* mode works well for this constraint.

**dev-dax**
Read and write using device DAX to a persistent memory device (e.g., /dev/dax0.0) through the PMDK libpmem library.

**external**
Prefix to specify loading an external I/O engine object file. Append the engine filename, e.g. `ioengine=external:/tmp/foo.o` to load ioengine `foo.o` in `/tmp`. The path can be either absolute or relative. See `engines/skeleton_external.c` for details of writing an external I/O engine.

**libpmem**
Read and write using mmap I/O to a file on a filesystem mounted with DAX on a persistent memory device through the PMDK libpmem library.

**ime_psync**
Synchronous read and write using DDN's Infinite Memory Engine (IME). This engine is very basic and issues calls to IME whenever an IO is queued.

**ime_psyncv**
Synchronous read and write using DDN's Infinite Memory Engine (IME). This engine uses iovecs and will try to stack as much IOs as possible (if the IOs are "contiguous" and the IO depth is not exceeded) before issuing a call to IME.

**ime_aio**
Asynchronous read and write using DDN's Infinite Memory Engine (IME). This engine will try to stack as much IOs as possible by creating requests for IME. FIO will then decide when to commit these requests.

**libiscsi**
Read and write iscsi lun with libiscsi.

**nbd**
Read and write a Network Block Device (NBD).

**libcufile**
I/O engine supporting libcufile synchronous access to nvidia-fs and a GPUDirect Storage-supported filesystem. This engine performs I/O without transferring buffers between user-space and the kernel, unless *verify* is set or *cuda_io* is *posix*. *iomem* must not be *cudamalloc*. This ioengine defines engine specific options.

**dfs**
I/O engine supporting asynchronous read and write operations to the DAOS File System (DFS) via libdfs.

**nfs**
I/O engine supporting asynchronous read and write operations to NFS filesystems from userspace via libnfs. This is useful for achieving higher concurrency and thus throughput than is possible via kernel NFS.

**exec**
Execute 3rd party tools. Could be used to perform monitoring during jobs runtime.

**xnvme**
I/O engine using the xNVMe C API, for NVMe devices. The xnvme engine provides flexibility to access GNU/Linux Kernel NVMe driver via libaio, IOCTLs, io_uring, the SPDK NVMe driver,

or your own custom NVMe driver. The xnvme engine includes engine specific options. (See https://xnvme.io).

**libblkio**
Use the libblkio library (https://gitlab.com/libblkio/libblkio). The specific *driver* to use must be set using *libblkio_driver*. If *mem*/*iomem* is not specified, memory allocation is delegated to libblkio (and so is guaranteed to work with the selected *driver*). One libblkio instance is used per process, so all jobs setting option *thread* will share a single instance (with one queue per thread) and must specify compatible options. Note that some drivers don't allow several instances to access the same device or file simultaneously, but allow it for threads.

File/directory operation engines define how the job operates file or directory. The following types are defined:

**filecreate**
Simply create the files and do no I/O to them. You still need to set *filesize* so that all the accounting still occurs, but no actual I/O will be done other than creating the file. Example job file: filecreate-ioengine.fio.

**filestat**
Simply do stat() and do no I/O to the file. You need to set 'filesize' and 'nrfiles', so that files will be created. This engine is to measure file lookup and meta data access. Example job file: filestat-ioengine.fio.

**filedelete**
Simply delete the files by unlink() and do no I/O to them. You need to set 'filesize' and 'nrfiles', so that the files will be created. This engine is to measure file delete. Example job file: filedelete-ioengine.fio.

**dircreate**
Simply create the directories and do no I/O to them. You still need to set *filesize* so that all the accounting still occurs, but no actual I/O will be done other than creating the directories. Example job file: dircreate-ioengine.fio.

**dirstat**
Simply do stat() and do no I/O to the directories. You need to set 'filesize' and 'nrfiles', so that directories will be created. This engine is to measure directory lookup and meta data access. Example job file: dirstat-ioengine.fio.

**dirdelete**
Simply delete the directories by rmdir() and do no I/O to them. You need to set 'filesize' and 'nrfiles', so that the directories will be created. This engine is to measure directory delete. Example job file: dirdelete-ioengine.fio.

For file and directory operation engines, there is no I/O throughput, then the statistics data in report have different meanings. The meaningful output indexes are: 'iops' and 'clat'. 'bw' is meaningless. Refer to section: "Interpreting the output" for more details.

### 1.13.11 I/O engine specific parameters

In addition, there are some parameters which are only valid when a specific *ioengine* is in use. These are used identically to normal parameters, with the caveat that when used on the command line, they must come after the *ioengine* that defines them is selected.

**cmdprio_percentage**=int[,int] : [io_uring] [libaio]
Set the percentage of I/O that will be issued with the highest priority. Default: 0. A single value applies to reads and writes. Comma-separated values may be specified for reads and writes. For this option to be effective, NCQ priority must be supported and enabled, and the *direct* option must be set. fio must also be run as the root user. Unlike slat/clat/lat stats, which can be tracked and reported independently, per priority stats only track and report

a single type of latency. By default, completion latency (clat) will be reported, if *lat_percentiles* is set, total latency (lat) will be reported.

**cmdprio_class**=int[,int] : [io_uring] [libaio]

Set the I/O priority class to use for I/Os that must be issued with a priority when *cmdprio_percentage* or *cmdprio_bssplit* is set. If not specified when *cmdprio_percentage* or *cmdprio_bssplit* is set, this defaults to the highest priority class. A single value applies to reads and writes. Comma-separated values may be specified for reads and writes. See *ionice(1)*. See also the *prioclass* option.

**cmdprio_hint**=int[,int] : [io_uring] [libaio]

Set the I/O priority hint to use for I/Os that must be issued with a priority when *cmdprio_percentage* or *cmdprio_bssplit* is set. If not specified when *cmdprio_percentage* or *cmdprio_bssplit* is set, this defaults to 0 (no hint). A single value applies to reads and writes. Comma-separated values may be specified for reads and writes. See also the *priohint* option.

**cmdprio**=int[,int] : [io_uring] [libaio]

Set the I/O priority value to use for I/Os that must be issued with a priority when *cmdprio_percentage* or *cmdprio_bssplit* is set. If not specified when *cmdprio_percentage* or *cmdprio_bssplit* is set, this defaults to 0. Linux limits us to a positive value between 0 and 7, with 0 being the highest. A single value applies to reads and writes. Comma-separated values may be specified for reads and writes. See *ionice(1)*. Refer to an appropriate manpage for other operating systems since meaning of priority may differ. See also the *prio* option.

**cmdprio_bssplit**=str[,str] : [io_uring] [libaio]

To get a finer control over I/O priority, this option allows specifying the percentage of IOs that must have a priority set depending on the block size of the IO. This option is useful only when used together with the *bssplit* option, that is, multiple different block sizes are used for reads and writes.

The first accepted format for this option is the same as the format of the *bssplit* option:

cmdprio_bssplit=blocksize/percentage:blocksize/percentage

In this case, each entry will use the priority class, priority hint and priority level defined by the options *cmdprio_class*, *cmdprio* and *cmdprio_hint* respectively.

The second accepted format for this option is:

cmdprio_bssplit=blocksize/percentage/class/level:blocksize/percentage/class/level

In this case, the priority class and priority level is defined inside each entry. In comparison with the first accepted format, the second accepted format does not restrict all entries to have the same priority class and priority level.

The third accepted format for this option is:

cmdprio_bssplit=blocksize/percentage/class/level/hint:...

This is an extension of the second accepted format that allows one to also specify a priority hint.

For all formats, only the read and write data directions are supported, values for trim IOs are ignored. This option is mutually exclusive with the *cmdprio_percentage* option.

**fixedbufs** : [io_uring] [io_uring_cmd]

If fio is asked to do direct IO, then Linux will map pages for each IO call, and release them when IO is done. If this option is set, the pages are pre-mapped before IO is started. This eliminates the need to map and release for each IO. This is more efficient, and reduces the IO latency as well.

**nonvectored**=int : [io_uring] [io_uring_cmd]

With this option, fio will use non-vectored read/write commands, where address must contain the address directly. Default is -1.

**force_async**=int : [io_uring] [io_uring_cmd]

Normal operation for io_uring is to try and issue an sqe as non-blocking first, and if that fails, execute it in an async manner. With this option set to N, then every N request fio will ask sqe to be issued in an async manner. Default is 0.

**registerfiles** : [io_uring] [io_uring_cmd]

With this option, fio registers the set of files being used with the kernel. This avoids the overhead of managing file counts in the kernel, making the submission and completion part more lightweight. Required for the below *sqthread_poll* option.

**sqthread_poll** : [io_uring] [io_uring_cmd] [xnvme]

Normally fio will submit IO by issuing a system call to notify the kernel of available items in the SQ ring. If this option is set, the act of submitting IO will be done by a polling thread in the kernel. This frees up cycles for fio, at the cost of using more CPU in the system. As submission is just the time it takes to fill in the sqe entries and any syscall required to wake up the idle kernel thread, fio will not report submission latencies.

**sqthread_poll_cpu**=int : [io_uring] [io_uring_cmd]

When *sqthread_poll* is set, this option provides a way to define which CPU should be used for the polling thread.

**cmd_type**=str : [io_uring_cmd]

Specifies the type of uring passthrough command to be used. Supported value is nvme. Default is nvme.

**hipri**

[io_uring] [io_uring_cmd] [xnvme]

If this option is set, fio will attempt to use polled IO completions. Normal IO completions generate interrupts to signal the completion of IO, polled completions do not. Hence they are require active reaping by the application. The benefits are more efficient IO for high IOPS scenarios, and lower latencies for low queue depth IO.

[libblkio]

Use poll queues. This is incompatible with *libblkio_wait_mode=eventfd* and *libblkio_force_enable_completion_eventfd*.

[pvsync2]

Set RWF_HIPRI on I/O, indicating to the kernel that it's of higher priority than normal.

[sg]

If this option is set, fio will attempt to use polled IO completions. This will have a similar effect as (io_uring)hipri. Only SCSI READ and WRITE commands will have the SGV4_FLAG_HIPRI set (not UNMAP (trim) nor VERIFY). Older versions of the Linux sg driver that do not support hipri will simply ignore this flag and do normal IO. The Linux SCSI Low Level Driver (LLD) that "owns" the device also needs to support hipri (also known as iopoll and mq_poll). The MegaRAID driver is an example of a SCSI LLD. Default: clear (0) which does normal (interrupted based) IO.

**userspace_reap** : [libaio]

Normally, with the libaio engine in use, fio will use the *io_getevents(2)* system call to reap newly returned events. With this flag turned on, the AIO ring will be read directly from user-space to reap events. The reaping mode is only enabled when polling for a minimum of 0 events (e.g. when *iodepth_batch_complete=0*).

**hipri_percentage** : [pvsync2]

When hipri is set this determines the probability of a pvsync2 I/O being high priority. The default is 100%.

**nowait**=bool : [pvsync2] [libaio] [io_uring] [io_uring_cmd]

By default if a request cannot be executed immediately (e.g. resource starvation, waiting on locks) it is queued and the initiating process will be blocked until the required resource becomes free.

This option sets the RWF_NOWAIT flag (supported from the 4.14 Linux kernel) and the call will return instantly with EAGAIN or a partial result rather than waiting.

It is useful to also use ignore_error=EAGAIN when using this option.

Note: glibc 2.27, 2.28 have a bug in syscall wrappers preadv2, pwritev2. They return EOPNOTSUP instead of EAGAIN.

For cached I/O, using this option usually means a request operates only with cached data. Currently the RWF_NOWAIT flag does not supported for cached write.

For direct I/O, requests will only succeed if cache invalidation isn't required, file blocks are fully allocated and the disk request could be issued immediately.

**uncached**=int : [pvsync2] [io_uring]

This option will perform buffered IO without retaining data in the page cache after the operation completes.

Reads work like a normal buffered read but pages are evicted immediately after data is copied to userspace. Writes work like buffered writes but a writeback is initiated before the syscall returns. Pages are evicted once the writeback completes.

This option sets the RWF_UNCACHED flag (supported from the 6.14 Linux kernel) on a per-IO basis.

**atomic**=bool : [pvsync2] [libaio] [io_uring]

This option means that writes are issued with torn-write protection, meaning that for a power fail or kernel crash, all or none of the data from the write will be stored, but never a mix of old and new data. Torn-write protection is also known as atomic writes.

This option sets the RWF_ATOMIC flag (supported from the 6.11 Linux kernel) on a per-IO basis.

Writes with RWF_ATOMIC set will be rejected by the kernel when the file does not support torn-write protection. To learn a file's torn-write limits, issue statx with STATX_WRITE_ATOMIC.

**libaio_vectored**=bool : [libaio]

Submit vectored read and write requests.

**fdp**=bool : [io_uring_cmd] [xnvme]

Enable Flexible Data Placement mode for write commands.

**dataplacement**=str : [io_uring_cmd] [xnvme]

Specifies the data placement directive type to use for write commands. The following types are supported:

**none**
Do not use a data placement directive. This is the default.

**fdp**
Use Flexible Data Placement directives for write commands. This is equivalent to specifying *fdp* =1.

**streams**
Use Streams directives for write commands.

**plid_select**=str, **fdp_pli_select**=str : [io_uring_cmd] [xnvme]

Defines how fio decides which placement ID to use next. The following types are defined:

**random**
Choose a placement ID at random (uniform).

**roundrobin**
Round robin over available placement IDs. This is the default.

**scheme**
Choose a placement ID (index) based on the scheme file defined by the option *dp_scheme*.

The available placement ID (indices) are defined by the option *fdp_pli* or *plids* except for the case of **scheme**.

**plids**=str, **fdp_pli**=str : [io_uring_cmd] [xnvme]

Select which Placement ID Indices (FDP) or Placement IDs (streams) this job is allowed to use for writes. This option accepts a comma-separated list of values or ranges (e.g., 1,2-4,5,6-8).

For FDP by default, the job will cycle through all available Placement IDs, so use this option to be selective. The values specified here are array indices for the list of placement IDs returned by the nvme-cli command `nvme fdp status`. If you want fio to use FDP placement identifiers only at indices 0, 2 and 5, set `plids=0,2,5`.

For streams this should be a list of Stream IDs.

**dp_scheme**=str : [io_uring_cmd] [xnvme]

Defines which placement ID (index) to be selected based on offset(LBA) range. The file should contains one or more scheme entries in the following format:

0, 10737418240, 0 10737418240, 21474836480, 1 21474836480, 32212254720, 2 ...

Each line, a scheme entry, contains start offset, end offset, and placement ID (index) separated by comma(,). If the write offset is within the range of a certain scheme entry(start offset  offset < end offset), the corresponding placement ID (index) will be selected. If the write offset belongs to multiple scheme entries, the first matched scheme entry will be applied. If the offset is not within any range of scheme entry, dspec field will be set to 0, default RUH. (Caution: In case of multiple devices in a job, all devices of the job will be affected by the scheme. If this option is specified, the option *plids* or *fdp_pli* will be ignored.)

**md_per_io_size**=int : [io_uring_cmd] [xnvme]

Size in bytes for separate metadata buffer per IO. For io_uring_cmd these buffers are allocated using malloc regardless of what is set for *iomem*. Default: 0.

**pi_act**=int : [io_uring_cmd] [xnvme]

Action to take when nvme namespace is formatted with protection information. If this is set to 1 and namespace is formatted with metadata size equal to protection information size, fio won't use separate metadata buffer or extended logical block. If this is set to 1 and namespace is formatted with metadata size greater than protection information size, fio will not generate or verify the protection information portion of metadata for write or read case respectively. If this is set to 0, fio generates protection information for write case and verifies for read case. Default: 1.

For 16 bit CRC generation fio will use isa-l if available otherwise it will use the default slower generator. (see: https://github.com/intel/isa-l)

**pi_chk**=str[,str][,str] : [io_uring_cmd] [xnvme]

Controls the protection information check. This can take one or more of these values. Default: none.

**GUARD**
Enables protection information checking of guard field.

**REFTAG**
Enables protection information checking of logical block reference tag field.

**APPTAG**
Enables protection information checking of application tag field.

**apptag**=int : [io_uring_cmd] [xnvme]

Specifies logical block application tag value, if namespace is formatted to use end to end protection information. Default: 0x1234.

**apptag_mask**=int : [io_uring_cmd] [xnvme]

Specifies logical block application tag mask value, if namespace is formatted to use end to end protection information. Default: 0xffff.

**num_range**=int : [io_uring_cmd]

For trim command this will be the number of ranges to trim per I/O request. The number of logical blocks per range is determined by the *bs* option which should be a multiple of logical block size. This cannot be used with read or write. Note that setting this option > 1, *log_offset* will not be able to log all the offsets. Default: 1.

**cpuload**=int : [cpuio]

Attempt to use the specified percentage of CPU cycles. This is a mandatory option when using cpuio I/O engine.

**cpuchunks**=int : [cpuio]

Split the load into cycles of the given time. In microseconds.

**cpumode**=str : [cpuio]

Specify how to stress the CPU. It can take these two values:

**noop**
This is the default where the CPU executes noop instructions.

**qsort**
Replace the default noop instructions loop with a qsort algorithm to consume more energy.

**exit_on_io_done**=bool : [cpuio]

Detect when I/O threads are done, then exit.

**namenode**=str : [libhdfs]

The hostname or IP address of a HDFS cluster namenode to contact.

**port**=int

[libhdfs]

The listening port of the HFDS cluster namenode.

[netsplice], [net]

The TCP or UDP port to bind to or connect to. If this is used with *numjobs* to spawn multiple instances of the same job type, then this will be the starting port number since fio will use a range of ports.

[rdma]

The port to use for RDMA-CM communication. This should be the same value on the client and the server side.

**hostname**=str : [netsplice] [net] [rdma]

The hostname or IP address to use for TCP, UDP or RDMA-CM based I/O. If the job is a TCP listener or UDP reader, the hostname is not used and must be omitted unless it is a valid UDP multicast address.

**interface**=str : [netsplice] [net]

The IP address of the network interface used to send or receive UDP multicast.

**ttl**=int : [netsplice] [net]

Time-to-live value for outgoing UDP multicast packets. Default: 1.

**nodelay**=bool : [netsplice] [net]

Set TCP_NODELAY on TCP connections.

**protocol**=str, **proto**=str : [netsplice] [net]

The network protocol to use. Accepted values are:

**tcp**
Transmission control protocol.

**tcpv6**
Transmission control protocol V6.

**udp**
User datagram protocol.

**udpv6**
User datagram protocol V6.

**unix**
UNIX domain socket.

**vsock**
VSOCK protocol.

When the protocol is TCP, UDP or VSOCK, the port must also be given, as well as the hostname if the job is a TCP or VSOCK listener or UDP reader. For unix sockets, the normal *filename* option should be used and the port is invalid. When the protocol is VSOCK, the *hostname* is the CID of the remote VM.

**listen** : [netsplice] [net]

For TCP network connections, tell fio to listen for incoming connections rather than initiating an outgoing connection. The *hostname* must be omitted if this option is used.

**pingpong** : [netsplice] [net]

Normally a network writer will just continue writing data, and a network reader will just consume packages. If pingpong=1 is set, a writer will send its normal payload to the reader, then wait for the reader to send the same payload back. This allows fio to measure network latencies. The submission and completion latencies then measure local time spent sending or receiving, and the completion latency measures how long it took for the other end to receive and send back. For UDP multicast traffic pingpong=1 should only be set for a single reader when multiple readers are listening to the same address.

**window_size** : [netsplice] [net]

Set the desired socket buffer size for the connection.

**mss** : [netsplice] [net]

Set the TCP maximum segment size (TCP_MAXSEG).

**donorname**=str : [e4defrag]

File will be used as a block donor (swap extents between files).

**inplace**=int : [e4defrag]

Configure donor file blocks allocation strategy:

**0**
Default. Preallocate donor's file on init.

**1**
Allocate space immediately inside defragment event, and free right after event.

**clustername**=str : [rbd,rados]

Specifies the name of the Ceph cluster.

**rbdname**=str : [rbd]

Specifies the name of the RBD.

**clientname**=str : [rbd,rados]

Specifies the username (without the 'client.' prefix) used to access the Ceph cluster. If the *clustername* is specified, the *clientname* shall be the full *type.id* string. If no type. prefix is given, fio will add 'client.' by default.

**conf**=str : [rados]

Specifies the configuration path of ceph cluster, so conf file does not have to be /etc/ceph/ceph.conf.

**busy_poll**=bool : [rbd,rados]

Poll store instead of waiting for completion. Usually this provides better throughput at cost of higher(up to 100%) CPU utilization.

**touch_objects**=bool : [rados]

During initialization, touch (create if do not exist) all objects (files). Touching all objects affects ceph caches and likely impacts test results. Enabled by default.

**pool**=str :

[rbd,rados]

Specifies the name of the Ceph pool containing RBD or RADOS data.

[dfs]

Specify the label or UUID of the DAOS pool to connect to.

**cont**=str : [dfs]

Specify the label or UUID of the DAOS container to open.

**chunk_size**=int

[dfs]

Specify a different chunk size (in bytes) for the dfs file. Use DAOS container's chunk size by default.

[libhdfs]

The size of the chunk to use for each file.

**object_class**=str : [dfs]

Specify a different object class for the dfs file. Use DAOS container's object class by default.

**skip_bad**=bool : [mtd]

Skip operations against known bad blocks.

**hdfsdirectory** : [libhdfs]

libhdfs will create chunk in this HDFS directory.

**verb**=str : [rdma]

The RDMA verb to use on this side of the RDMA ioengine connection. Valid values are write, read, send and recv. These correspond to the equivalent RDMA verbs (e.g. write = rdma_write etc.). Note that this only needs to be specified on the client side of the connection. See the examples folder.

**bindname**=str : [rdma]

The name to use to bind the local RDMA-CM connection to a local RDMA device. This could be a hostname or an IPv4 or IPv6 address. On the server side this will be passed into the rdma_bind_addr() function and on the client site it will be used in the rdma_resolve_add() function. This can be useful when multiple paths exist between the client and the server or in certain loopback configurations.

**stat_type**=str : [filestat]

Specify stat system call type to measure lookup/getattr performance. Default is **stat** for *stat(2)*.

**readfua**=bool : [sg] [io_uring_cmd]

With readfua option set to 1, read operations include the force unit access (fua) flag. Default is 0.

**writefua**=bool : [sg] [io_uring_cmd]

With writefua option set to 1, write operations include the force unit access (fua) flag. Default is 0.

**write_mode**=str : [io_uring_cmd]

Specifies the type of write operation. Defaults to 'write'.

**write**
Use Write commands for write operations

**uncor**
Use Write Uncorrectable commands for write operations

**zeroes**
Use Write Zeroes commands for write operations

**verify**
Use Verify commands for write operations

**verify_mode**=str : [io_uring_cmd]

Specifies the type of command to be used in the verification phase. Defaults to 'read'.

**read**
Use Read commands for data verification

**compare**
Use Compare commands for data verification. This option is only valid with specific pattern(s), which means it *must* be given with *verify=pattern* and *verify_pattern=<pattern>*.

**sg_write_mode**=str : [sg]

Specify the type of write commands to issue. This option can take ten values:

**write**
This is the default where write opcodes are issued as usual.

**write_and_verify**
Issue WRITE AND VERIFY commands. The BYTCHK bit is set to 0. This directs the device to carry out a medium verification with no data comparison. The writefua option is ignored with this selection.

**verify**
This option is deprecated. Use write_and_verify instead.

**write_same**
Issue WRITE SAME commands. This transfers a single block to the device and writes this same block of data to a contiguous sequence of LBAs beginning at the specified offset. fio's block size parameter specifies the amount of data written with each command. However, the amount of data actually transferred to the device is equal to the device's block (sector) size. For a device with 512 byte sectors, blocksize=8k will write 16 sectors with each command. fio will still generate 8k of data for each command but only the first 512 bytes will be used and transferred to the device. The writefua option is ignored with this selection.

**same**
This option is deprecated. Use write_same instead.

**write_same_ndob**
Issue WRITE SAME(16) commands as above but with the No Data Output Buffer (NDOB) bit set. No data will be transferred to the device with this bit set. Data written will be a pre-determined pattern such as all zeroes.

**write_stream**
Issue WRITE STREAM(16) commands. Use the **stream_id** option to specify the stream identifier.

**verify_bytchk_00**
Issue VERIFY commands with BYTCHK set to 00. This directs the device to carry out a medium verification with no data comparison.

**verify_bytchk_01**
Issue VERIFY commands with BYTCHK set to 01. This directs the device to compare the data on the device with the data transferred to the device.

**verify_bytchk_11**
Issue VERIFY commands with BYTCHK set to 11. This transfers a single block to the device and compares the contents of this block with the data on the device beginning at the specified offset. fio's block size parameter specifies the total amount of data compared with this command. However, only one block (sector) worth of data is transferred to the device. This is similar to the WRITE SAME command except that data is compared instead of written.

**stream_id**=int : [sg]
Set the stream identifier for WRITE STREAM commands. If this is set to 0 (which is not a valid stream identifier) fio will open a stream and then close it when done. Default is 0.

**http_host**=str : [http]
Hostname to connect to. HTTP port 80 is used automatically when the value of the https parameter is *off*, and HTTPS port 443 if it is *on*. A virtual-hosted-style S3 hostname starts with a bucket name, while a path-style S3 hostname does not. See https://docs.aws.amazon.com/AmazonS3/latest/userguide/VirtualHosting.html for detailed examples. Default is **localhost** (path-style S3 hostname)

**http_user**=str : [http]
Username for HTTP authentication.

**http_pass**=str : [http]
Password for HTTP authentication.

**https**=str : [http]
Enable HTTPS instead of http. *on* enables HTTPS; *insecure* will enable HTTPS, but disable SSL peer verification (use with caution!). Default is **off**

**http_mode**=str : [http]
Which HTTP access mode to use: *webdav*, *swift*, or *s3*. Default is **webdav**

**http_s3_region**=str : [http]
The S3 region/zone string. Default is **us-east-1**

**http_s3_key**=str : [http]
The S3 secret key.

**http_s3_keyid**=str : [http]
The S3 key/access id.

**http_s3_security_token**=str : [http]
The S3 security token.

**http_s3_sse_customer_key**=str : [http]
The encryption customer key in SSE server side.

**http_s3_sse_customer_algorithm**=str : [http]
The encryption customer algorithm in SSE server side. Default is **AES256**

**http_s3_storage_class**=str : [http]

Which storage class to access. User-customizable settings. Default is **STANDARD**

**http_swift_auth_token**=str : [http]

The Swift auth token. See the example configuration file on how to retrieve this.

**http_verbose**=int : [http]

Enable verbose requests from libcurl. Useful for debugging. 1 turns on verbose logging from libcurl, 2 additionally enables HTTP IO tracing. Default is **0**

**http_object_mode**=str : [http]

How to structure objects for HTTP IO: *block* or *range*. Default is **block**.

In *block* mode, one object is created for every block. The HTTP engine treats *blocksize* as the size of the object to read or write, and appends the block start/end offsets to the *filename* to create the target object path. Reads and writes operate on whole objects at a time.

In *range* mode, one object is created for every file. The object path is the filename directly for both read and write I/O. For read requests, the *blocksize* and *offset* will be used to set the "Range" header on read requests to issue partial reads of the object. For write requests, blocksize is used to set the size of the object, the same as in *block* mode.

**uri**=str : [nbd]

Specify the NBD URI of the server to test. The string is a standard NBD URI (see https://github.com/NetworkBlockDevice/nbd/tree/master/doc). Example URIs: nbd://localhost:10809 nbd+unix:///?socket=/tmp/socket nbds://tlshost/exportname

**gpu_dev_ids**=str : [libcufile]

Specify the GPU IDs to use with CUDA. This is a colon-separated list of int. GPUs are assigned to workers roundrobin. Default is 0.

**cuda_io**=str : [libcufile]

Specify the type of I/O to use with CUDA. Default is **cufile**.

**cufile**
: Use libcufile and nvidia-fs. This option performs I/O directly between a GPUDirect Storage filesystem and GPU buffers, avoiding use of a bounce buffer. If *verify* is set, cudaMemcpy is used to copy verificaton data between RAM and GPU. Verification data is copied from RAM to GPU before a write and from GPU to RAM after a read. *direct* must be 1.

**posix**
: Use POSIX to perform I/O with a RAM buffer, and use cudaMemcpy to transfer data between RAM and the GPUs. Data is copied from GPU to RAM before a write and copied from RAM to GPU after a read. *verify* does not affect use of cudaMemcpy.

**nfs_url**=str : [nfs]

URL in libnfs format, eg nfs://<server|ipv4|ipv6>/path[?arg=val[&arg=val]*] Refer to the libnfs README for more details.

**program**=str : [exec]

Specify the program to execute.

**arguments**=str : [exec]

Specify arguments to pass to program. Some special variables can be expanded to pass fio's job details to the program.

**%r**
: Replaced by the duration of the job in seconds.

**%n**
Replaced by the name of the job.

**grace_time**=int : [exec]
Specify the time between the SIGTERM and SIGKILL signals. Default is 1 second.

**std_redirect**=bool : [exec]
If set, stdout and stderr streams are redirected to files named from the job name. Default is true.

**xnvme_async**=str : [xnvme]
Select the xnvme async command interface. This can take these values.

**emu**
This is default and use to emulate asynchronous I/O by using a single thread to create a queue pair on top of a synchronous I/O interface using the NVMe driver IOCTL.

**thrpool**
Emulate an asynchronous I/O interface with a pool of userspace threads on top of a synchronous I/O interface using the NVMe driver IOCTL. By default four threads are used.

**io_uring**
Linux native asynchronous I/O interface which supports both direct and buffered I/O.

**io_uring_cmd**
Fast Linux native asynchronous I/O interface for NVMe pass through commands. This only works with NVMe character device (/dev/ngXnY).

**libaio**
Use Linux aio for Asynchronous I/O.

**posix**
Use the posix asynchronous I/O interface to perform one or more I/O operations asynchronously.

**vfio**
Use the user-space VFIO-based backend, implemented using libvfn instead of SPDK.

**nil**
Do not transfer any data; just pretend to. This is mainly used for introspective performance evaluation.

**xnvme_sync**=str : [xnvme]
Select the xnvme synchronous command interface. This can take these values.

**nvme**
This is default and uses Linux NVMe Driver ioctl() for synchronous I/O.

**psync**
This supports regular as well as vectored pread() and pwrite() commands.

**block**
This is the same as psync except that it also supports zone management commands using Linux block layer IOCTLs.

**xnvme_admin**=str : [xnvme]
Select the xnvme admin command interface. This can take these values.

**nvme**
This is default and uses linux NVMe Driver ioctl() for admin commands.

**block**
Use Linux Block Layer ioctl() and sysfs for admin commands.

**xnvme_dev_nsid**=int : [xnvme]

xnvme namespace identifier for userspace NVMe driver, SPDK or vfio.

**xnvme_dev_subnqn**=str : [xnvme]

Sets the subsystem NQN for fabrics. This is for xNVMe to utilize a fabrics target with multiple systems.

**xnvme_mem**=str : [xnvme]

Select the xnvme memory backend. This can take these values.

**posix**
This is the default posix memory backend for linux NVMe driver.

**hugepage**
Use hugepages, instead of existing posix memory backend. The memory backend uses hugetlbfs. This require users to allocate hugepages, mount hugetlbfs and set an environment variable for XNVME_HUGETLB_PATH.

**spdk**
Uses SPDK's memory allocator.

**vfio**
Uses libvfn's memory allocator. This also specifies the use of libvfn backend instead of SPDK.

**xnvme_iovec**=int : [xnvme]

If this option is set. xnvme will use vectored read/write commands.

**libblkio_driver**=str : [libblkio]

The libblkio *driver* to use. Different drivers access devices through different underlying interfaces. Available drivers depend on the libblkio version in use and are listed at https://libblkio.gitlab.io/libblkio/blkio.html#drivers

**libblkio_path**=str : [libblkio]

Sets the value of the driver-specific "path" property before connecting the libblkio instance, which identifies the target device or file on which to perform I/O. Its exact semantics are driver-dependent and not all drivers may support it; see https://libblkio.gitlab.io/libblkio/blkio.html#drivers

**libblkio_pre_connect_props**=str : [libblkio]

A colon-separated list of additional libblkio properties to be set after creating but before connecting the libblkio instance. Each property must have the format <name>=<value>. Colons can be escaped as \:. These are set after the engine sets any other properties, so those can be overridden. Available properties depend on the libblkio version in use and are listed at https://libblkio.gitlab.io/libblkio/blkio.html#properties

**libblkio_num_entries**=int : [libblkio]

Sets the value of the driver-specific "num-entries" property before starting the libblkio instance. Its exact semantics are driver-dependent and not all drivers may support it; see https://libblkio.gitlab.io/libblkio/blkio.html#drivers

**libblkio_queue_size**=int : [libblkio]

Sets the value of the driver-specific "queue-size" property before starting the libblkio instance. Its exact semantics are driver-dependent and not all drivers may support it; see https://libblkio.gitlab.io/libblkio/blkio.html#drivers

**libblkio_pre_start_props**=str : [libblkio]

A colon-separated list of additional libblkio properties to be set after connecting but before starting the libblkio instance. Each property must have the format <name>=<value>. Colons can be escaped as \:. These are set after the engine sets any other properties, so those can be overridden. Available properties depend on the libblkio version in use and are listed at https://libblkio.gitlab.io/libblkio/blkio.html#properties

**libblkio_vectored** : [libblkio]

Submit vectored read and write requests.

**libblkio_write_zeroes_on_trim** : [libblkio]

Submit trims as "write zeroes" requests instead of discard requests.

**libblkio_wait_mode**=str : [libblkio]

How to wait for completions:

**block (default)**
Use a blocking call to `blkioq_do_io()`.

**eventfd**
Use a blocking call to `read()` on the completion eventfd.

**loop**
Use a busy loop with a non-blocking call to `blkioq_do_io()`.

**libblkio_force_enable_completion_eventfd** : [libblkio]

Enable the queue's completion eventfd even when unused. This may impact performance. The default is to enable it only if *libblkio_wait_mode=eventfd*.

**no_completion_thread** : [windowsaio]

Avoid using a separate thread for completion polling.

### 1.13.12 I/O depth

**iodepth**=int

Number of I/O units to keep in flight against the file. Note that increasing *iodepth* beyond 1 will not affect synchronous ioengines (except for small degrees when *verify_async* is in use). Even async engines may impose OS restrictions causing the desired depth not to be achieved. This may happen on Linux when using libaio and not setting *direct*=1, since buffered I/O is not async on that OS. Keep an eye on the I/O depth distribution in the fio output to verify that the achieved depth is as expected. Default: 1.

**iodepth_batch_submit**=int, **iodepth_batch**=int

This defines how many pieces of I/O to submit at once. It defaults to 1 which means that we submit each I/O as soon as it is available, but can be raised to submit bigger batches of I/O at the time. If it is set to 0 the *iodepth* value will be used.

**iodepth_batch_complete_min**=int, **iodepth_batch_complete**=int

This defines how many pieces of I/O to retrieve at once. It defaults to 1 which means that we'll ask for a minimum of 1 I/O in the retrieval process from the kernel. The I/O retrieval will go on until we hit the limit set by *iodepth_low*. If this variable is set to 0, then fio will always check for completed events before queuing more I/O. This helps reduce I/O latency, at the cost of more retrieval system calls.

**iodepth_batch_complete_max**=int

This defines maximum pieces of I/O to retrieve at once. This variable should be used along with *iodepth_batch_complete_min*=int variable, specifying the range of min and max amount of I/O which should be retrieved. By default it is equal to the *iodepth_batch_complete_min* value.

Example #1:

```
iodepth_batch_complete_min=1
iodepth_batch_complete_max=<iodepth>
```

which means that we will retrieve at least 1 I/O and up to the whole submitted queue depth. If none of I/O has been completed yet, we will wait.

Example #2:

```
iodepth_batch_complete_min=0
iodepth_batch_complete_max=<iodepth>
```

which means that we can retrieve up to the whole submitted queue depth, but if none of I/O has been completed yet, we will NOT wait and immediately exit the system call. In this example we simply do polling.

**iodepth_low**=int

The low water mark indicating when to start filling the queue again. Defaults to the same as *iodepth*, meaning that fio will attempt to keep the queue full at all times. If *iodepth* is set to e.g. 16 and *iodepth_low* is set to 4, then after fio has filled the queue of 16 requests, it will let the depth drain down to 4 before starting to fill it again.

**serialize_overlap**=bool

Serialize in-flight I/Os that might otherwise cause or suffer from data races. When two or more I/Os are submitted simultaneously, there is no guarantee that the I/Os will be processed or completed in the submitted order. Further, if two or more of those I/Os are writes, any overlapping region between them can become indeterminate/undefined on certain storage. These issues can cause verification to fail erratically when at least one of the racing I/Os is changing data and the overlapping region has a non-zero size. Setting `serialize_overlap` tells fio to avoid provoking this behavior by explicitly serializing in-flight I/Os that have a non-zero overlap. Note that setting this option can reduce both performance and the *iodepth* achieved.

This option only applies to I/Os issued for a single job except when it is enabled along with *io_submit_mode*=offload. In offload mode, fio will check for overlap among all I/Os submitted by offload jobs with *serialize_overlap* enabled.

Default: false.

**io_submit_mode**=str

This option controls how fio submits the I/O to the I/O engine. The default is *inline*, which means that the fio job threads submit and reap I/O directly. If set to *offload*, the job threads will offload I/O submission to a dedicated pool of I/O threads. This requires some coordination and thus has a bit of extra overhead, especially for lower queue depth I/O where it can increase latencies. The benefit is that fio can manage submission rates independently of the device completion rates. This avoids skewed latency reporting if I/O gets backed up on the device side (the coordinated omission problem). Note that this option cannot reliably be used with async IO engines.

### 1.13.13 I/O rate

**thinkcycles**=int

Stall the job for the specified number of cycles after an I/O has completed before issuing the next. May be used to simulate processing being done by an application. This is not taken into account for the time to be waited on for *thinktime*. Might not have any effect on some platforms, this can be checked by trying a setting a high enough amount of thinkcycles.

**thinktime**=time

Stall the job for the specified period of time after an I/O has completed before issuing the next. May be used to simulate processing being done by an application. When the unit is omitted, the value is interpreted in microseconds. See *thinktime_blocks*, *thinktime_iotime* and *thinktime_spin*.

**thinktime_spin**=time

Only valid if *thinktime* is set - pretend to spend CPU time doing something with the data received, before falling back to sleeping for the rest of the period specified by *thinktime*. When the unit is omitted, the value is interpreted in microseconds.

**thinktime_blocks**=int

Only valid if *thinktime* is set - control how many blocks to issue, before waiting *thinktime* usecs. If not set, defaults to 1 which will make fio wait *thinktime* usecs after every block. This effectively makes any queue

depth setting redundant, since no more than 1 I/O will be queued before we have to complete it and do our *thinktime*. In other words, this setting effectively caps the queue depth if the latter is larger.

**thinktime_blocks_type**=str

Only valid if *thinktime* is set - control how *thinktime_blocks* triggers. The default is *complete*, which triggers thinktime when fio completes *thinktime_blocks* blocks. If this is set to *issue*, then the trigger happens at the issue side.

**thinktime_iotime**=time

Only valid if *thinktime* is set - control *thinktime* interval by time. The *thinktime* stall is repeated after IOs are executed for *thinktime_iotime*. For example, `--thinktime_iotime=9s --thinktime=1s` repeat 10-second cycle with IOs for 9 seconds and stall for 1 second. When the unit is omitted, *thinktime_iotime* is interpreted as a number of seconds. If this option is used together with *thinktime_blocks*, the *thinktime* stall is repeated after *thinktime_iotime* or after *thinktime_blocks* IOs, whichever happens first.

**rate**=int[,int][,int]

Cap the bandwidth used by this job. The number is in bytes/sec, the normal suffix rules apply. Comma-separated values may be specified for reads, writes, and trims as described in *blocksize*.

For example, using *rate=1m,500k* would limit reads to 1MiB/sec and writes to 500KiB/sec. Capping only reads or writes can be done with *rate=,500k* or *rate=500k,* where the former will only limit writes (to 500KiB/sec) and the latter will only limit reads.

**rate_min**=int[,int][,int]

Tell fio to do whatever it can to maintain at least this bandwidth. Failing to meet this requirement will cause the job to exit. Comma-separated values may be specified for reads, writes, and trims as described in *blocksize*.

**rate_iops**=int[,int][,int]

Cap the bandwidth to this number of IOPS. Basically the same as *rate*, just specified independently of bandwidth. If the job is given a block size range instead of a fixed value, the smallest block size is used as the metric. Comma-separated values may be specified for reads, writes, and trims as described in *blocksize*.

**rate_iops_min**=int[,int][,int]

If fio doesn't meet this rate of I/O, it will cause the job to exit. Comma-separated values may be specified for reads, writes, and trims as described in *blocksize*.

**rate_process**=str

This option controls how fio manages rated I/O submissions. The default is *linear*, which submits I/O in a linear fashion with fixed delays between I/Os that gets adjusted based on I/O completion rates. If this is set to *poisson*, fio will submit I/O based on a more real world random request flow, known as the Poisson process (https://en.wikipedia.org/wiki/Poisson_point_process). The lambda will be 10^6 / IOPS for the given workload.

**rate_ignore_thinktime**=bool

By default, fio will attempt to catch up to the specified rate setting, if any kind of thinktime setting was used. If this option is set, then fio will ignore the thinktime and continue doing IO at the specified rate, instead of entering a catch-up mode after thinktime is done.

**rate_cycle**=int

Average bandwidth for *rate_min* and *rate_iops_min* over this number of milliseconds. Defaults to 1000.

### 1.13.14 I/O latency

**latency_target**=time

If set, fio will attempt to find the max performance point that the given workload will run at while maintaining a latency below this target. When the unit is omitted, the value is interpreted in microseconds. See *latency_window* and *latency_percentile*.

**latency_window**=time

Used with *latency_target* to specify the sample window that the job is run at varying queue depths to test the performance. When the unit is omitted, the value is interpreted in microseconds.

**latency_percentile**=float

The percentage of I/Os that must fall within the criteria specified by *latency_target* and *latency_window*. If not set, this defaults to 100.0, meaning that all I/Os must be equal or below to the value set by *latency_target*.

**latency_run**=bool

Used with *latency_target*. If false (default), fio will find the highest queue depth that meets *latency_target* and exit. If true, fio will continue running and try to meet *latency_target* by adjusting queue depth.

**max_latency**=time[,time][,time]

If set, fio will exit the job with an ETIMEDOUT error if it exceeds this maximum latency. When the unit is omitted, the value is interpreted in microseconds. Comma-separated values may be specified for reads, writes, and trims as described in *blocksize*.

### 1.13.15 I/O replay

**write_iolog**=str

Write the issued I/O patterns to the specified file. See *read_iolog*. Specify a separate file for each job, otherwise the iologs will be interspersed and the file may be corrupt. This file will be opened in append mode.

**read_iolog**=str

Open an iolog with the specified filename and replay the I/O patterns it contains. This can be used to store a workload and replay it sometime later. The iolog given may also be a blktrace binary file, which allows fio to replay a workload captured by **blktrace**. See *blktrace(8)* for how to capture such logging data. For blktrace replay, the file needs to be turned into a blkparse binary data file first (`blkparse <device> -o /dev/null -d file_for_fio.bin`). You can specify a number of files by separating the names with a ':' character. See the *filename* option for information on how to escape ':' characters within the file names. These files will be sequentially assigned to job clones created by *numjobs*. '-' is a reserved name, meaning read from stdin, notably if *filename* is set to '-' which means stdin as well, then this flag can't be set to '-'.

**read_iolog_chunked**=bool

Determines how iolog is read. If false(default) entire *read_iolog* will be read at once. If selected true, input from iolog will be read gradually. Useful when iolog is very large, or it is generated.

**merge_blktrace_file**=str

When specified, rather than replaying the logs passed to *read_iolog*, the logs go through a merge phase which aggregates them into a single blktrace. The resulting file is then passed on as the *read_iolog* parameter. The intention here is to make the order of events consistent. This limits the influence of the scheduler compared to replaying multiple blktraces via concurrent jobs.

**merge_blktrace_scalars**=float_list

This is a percentage based option that is index paired with the list of files passed to *read_iolog*. When merging is performed, scale the time of each event by the corresponding amount. For example, `--merge_blktrace_scalars="50:100"` runs the first trace in halftime and the second trace in realtime. This knob is separately tunable from *replay_time_scale* which scales the trace during runtime and does not change the output of the merge unlike this option.

**merge_blktrace_iters**=float_list

This is a whole number option that is index paired with the list of files passed to *read_iolog*. When merging is performed, run each trace for the specified number of iterations. For example, `--merge_blktrace_iters="2:1"` runs the first trace for two iterations and the second trace for one iteration.

**replay_no_stall**=bool

When replaying I/O with *read_iolog* the default behavior is to attempt to respect the timestamps within the log and replay them with the appropriate delay between IOPS. By setting this variable fio will not respect the timestamps and attempt to replay them as fast as possible while still respecting ordering. The result is the same I/O pattern to a given device, but different timings.

**replay_time_scale**=int

When replaying I/O with *read_iolog*, fio will honor the original timing in the trace. With this option, it's possible to scale the time. It's a percentage option, if set to 50 it means run at 50% the original IO rate in the trace. If set to 200, run at twice the original IO rate. Defaults to 100.

**replay_redirect**=str

While replaying I/O patterns using *read_iolog* the default behavior is to replay the IOPS onto the major/minor device that each IOP was recorded from. This is sometimes undesirable because on a different machine those major/minor numbers can map to a different device. Changing hardware on the same system can also result in a different major/minor mapping. `replay_redirect` causes all I/Os to be replayed onto the single specified device regardless of the device it was recorded from. i.e. *replay_redirect*= `/dev/sdc` would cause all I/O in the blktrace or iolog to be replayed onto `/dev/sdc`. This means multiple devices will be replayed onto a single device, if the trace contains multiple devices. If you want multiple devices to be replayed concurrently to multiple redirected devices you must blkparse your trace into separate traces and replay them with independent fio invocations. Unfortunately this also breaks the strict time ordering between multiple device accesses.

**replay_align**=int

Force alignment of the byte offsets in a trace to this value. The value must be a power of 2.

**replay_scale**=int

Scale byte offsets down by this factor when replaying traces. Should most likely use *replay_align* as well.

**replay_skip**=str

Sometimes it's useful to skip certain IO types in a replay trace. This could be, for instance, eliminating the writes in the trace. Or not replaying the trims/discards, if you are redirecting to a device that doesn't support them. This option takes a comma separated list of read, write, trim, sync.

### 1.13.16 Threads, processes and job synchronization

**thread**

Fio defaults to creating jobs by using fork, however if this option is given, fio will create jobs by using POSIX Threads' function *pthread_create(3)* to create threads instead.

**wait_for**=str

If set, the current job won't be started until all workers of the specified waitee job are done.

`wait_for` operates on the job name basis, so there are a few limitations. First, the waitee must be defined prior to the waiter job (meaning no forward references). Second, if a job is being referenced as a waitee, it must have a unique name (no duplicate waitees).

**nice**=int

Run the job with the given nice value. See man *nice(2)*.

On Windows, values less than -15 set the process class to "High"; -1 through -15 set "Above Normal"; 1 through 15 "Below Normal"; and above 15 "Idle" priority class.

**prio**=int

Set the I/O priority value of this job. Linux limits us to a positive value between 0 and 7, with 0 being the highest. See man *ionice(1)*. Refer to an appropriate manpage for other operating systems since meaning of priority may differ. For per-command priority setting, see I/O engine specific *cmdprio_percentage* and *cmdprio* options.

**prioclass**=int

Set the I/O priority class. See man *ionice(1)*. For per-command priority setting, see I/O engine specific *cmdprio_percentage* and *cmdprio_class* options.

**priohint**=int

Set the I/O priority hint. This is only applicable to platforms that support I/O priority classes and to devices with features controlled through priority hints, e.g. block devices supporting command duration limits, or CDL. CDL is a way to indicate the desired maximum latency of I/Os so that the device can optimize its internal command scheduling according to the latency limits indicated by the user.

For per-I/O priority hint setting, see the I/O engine specific *cmdprio_hint* option.

**cpus_allowed**=str

Controls the same options as *cpumask*, but accepts a textual specification of the permitted CPUs instead and CPUs are indexed from 0. So to use CPUs 0 and 5 you would specify `cpus_allowed=0,5`. This option also allows a range of CPUs to be specified – say you wanted a binding to CPUs 0, 5, and 8 to 15, you would set `cpus_allowed=0,5,8-15`.

On Windows, when `cpus_allowed` is unset only CPUs from fio's current processor group will be used and affinity settings are inherited from the system. An fio build configured to target Windows 7 makes options that set CPUs processor group aware and values will set both the processor group and a CPU from within that group. For example, on a system where processor group 0 has 40 CPUs and processor group 1 has 32 CPUs, `cpus_allowed` values between 0 and 39 will bind CPUs from processor group 0 and `cpus_allowed` values between 40 and 71 will bind CPUs from processor group 1. When using `cpus_allowed_policy=shared` all CPUs specified by a single `cpus_allowed` option must be from the same processor group. For Windows fio builds not built for Windows 7, CPUs will only be selected from (and be relative to) whatever processor group fio happens to be running in and CPUs from other processor groups cannot be used.

**cpus_allowed_policy**=str

Set the policy of how fio distributes the CPUs specified by *cpus_allowed* or *cpumask*. Two policies are supported:

**shared**
All jobs will share the CPU set specified.

**split**
Each job will get a unique CPU from the CPU set.

**shared** is the default behavior, if the option isn't specified. If **split** is specified, then fio will assign one cpu per job. If not enough CPUs are given for the jobs listed, then fio will roundrobin the CPUs in the set.

**cpumask**=int

Set the CPU affinity of this job. The parameter given is a bit mask of allowed CPUs the job may run on. So if you want the allowed CPUs to be 1 and 5, you would pass the decimal value of (1 << 1 | 1 << 5), or 34. See man *sched_setaffinity(2)*. This may not work on all supported operating systems or kernel versions. This option doesn't work well for a higher CPU count than what you can store in an integer mask, so it can only control cpus 1-32. For boxes with larger CPU counts, use *cpus_allowed*.

**numa_cpu_nodes**=str

Set this job running on specified NUMA nodes' CPUs. The arguments allow comma delimited list of cpu numbers, A-B ranges, or *all*. Note, to enable NUMA options support, fio must be built on a system with libnuma-dev(el) installed.

**numa_mem_policy**=str

Set this job's memory policy and corresponding NUMA nodes. Format of the arguments:

```
<mode>[:<nodelist>]
```

`mode` is one of the following memory policies: `default`, `prefer`, `bind`, `interleave` or `local`. For `default` and `local` memory policies, no node needs to be specified. For `prefer`, only one node is allowed. For `bind` and `interleave` the `nodelist` may be as follows: a comma delimited list of numbers, A-B ranges, or *all*.

**cgroup**=str

Add job to this control group. If it doesn't exist, it will be created. The system must have a mounted cgroup blkio mount point for this to work. If your system doesn't have it mounted, you can do so with:

```
# mount -t cgroup -o blkio none /cgroup
```

**cgroup_weight**=int

Set the weight of the cgroup to this value. See the documentation that comes with the kernel, allowed values are in the range of 100..1000.

**cgroup_nodelete**=bool

Normally fio will delete the cgroups it has created after the job completion. To override this behavior and to leave cgroups around after the job completion, set `cgroup_nodelete=1`. This can be useful if one wants to inspect various cgroup files after job completion. Default: false.

**flow_id**=int

The ID of the flow. If not specified, it defaults to being a global flow. See *flow*.

**flow**=int

Weight in token-based flow control. If this value is used, then fio regulates the activity between two or more jobs sharing the same flow_id. Fio attempts to keep each job activity proportional to other jobs' activities in the same flow_id group, with respect to requested weight per job. That is, if one job has *flow=3', another job has `flow=2' and another with `flow=1*, then there will be a roughly 3:2:1 ratio in how much one runs vs the others.

**flow_sleep**=int

The period of time, in microseconds, to wait after the flow counter has exceeded its proportion before retrying operations.

**stonewall, wait_for_previous**

Wait for preceding jobs in the job file to exit, before starting this one. Can be used to insert serialization points in the job file. A stone wall also implies starting a new reporting group, see *group_reporting*.

**exitall**

By default, fio will continue running all other jobs when one job finishes. Sometimes this is not the desired action. Setting `exitall` will instead make fio terminate all jobs in the same group, as soon as one job of that group finishes.

**exit_what**=str

By default, fio will continue running all other jobs when one job finishes. Sometimes this is not the desired action. Setting `exitall` will instead make fio terminate all jobs in the same group. The option `exit_what` allows one to control which jobs get terminated when `exitall` is enabled. The default is `group` and does not change the behaviour of `exitall`. The setting `all` terminates all jobs. The setting `stonewall` terminates all currently running jobs across all groups and continues execution with the next stonewalled group.

**exec_prerun**=str

Before running this job, issue the command specified through *system(3)*. Output is redirected in a file called `jobname.prerun.txt`.

**exec_postrun**=str

After the job completes, issue the command specified though *system(3)*. Output is redirected in a file called `jobname.postrun.txt`.

**uid**=int

Instead of running as the invoking user, set the user ID to this value before the thread/process does any work.

**gid**=int

Set group ID, see *uid*.

### 1.13.17 Verification

**verify_only**

Do not perform specified workload, only verify data still matches previous invocation of this workload. This option allows one to check data multiple times at a later date without overwriting it. This option makes sense only for workloads that write data, and does not support workloads with the *time_based* option set. *verify_write_sequence* and *verify_header_seed* will be disabled in this mode, unless they are explicitly enabled.

**do_verify**=bool

Run the verify phase after a write phase. Only valid if *verify* is set. Default: true.

**verify**=str

If writing to a file, fio can verify the file contents after each iteration of the job. Each verification method also implies verification of special header, which is written to the beginning of each block. This header also includes meta information, like offset of the block, block number, timestamp when block was written, initial seed value used to generate the buffer contents etc. *verify* can be combined with *verify_pattern* option. The allowed values are:

**md5**
Use an md5 sum of the data area and store it in the header of each block.

**crc64**
Use an experimental crc64 sum of the data area and store it in the header of each block.

**crc32c**
Use a crc32c sum of the data area and store it in the header of each block. This will automatically use hardware acceleration (e.g. SSE4.2 on an x86 or CRC crypto extensions on ARM64) but will fall back to software crc32c if none is found. Generally the fastest checksum fio supports when hardware accelerated.

**crc32c-intel**
Synonym for crc32c.

**crc32**
Use a crc32 sum of the data area and store it in the header of each block.

**crc16**
Use a crc16 sum of the data area and store it in the header of each block.

**crc7**
Use a crc7 sum of the data area and store it in the header of each block.

**xxhash**
Use xxhash as the checksum function. Generally the fastest software checksum that fio supports.

**sha512**
Use sha512 as the checksum function.

**sha256**
Use sha256 as the checksum function.

**sha1**
Use optimized sha1 as the checksum function.

**sha3-224**
Use optimized sha3-224 as the checksum function.

**sha3-256**
Use optimized sha3-256 as the checksum function.

**sha3-384**
Use optimized sha3-384 as the checksum function.

**sha3-512**
Use optimized sha3-512 as the checksum function.

**meta**
This option is deprecated, since now meta information is included in generic verification header and meta verification happens by default. For detailed information see the description of the *verify* setting. This option is kept because of compatibility's sake with old configurations. Do not use it.

**pattern**
Verify a strict pattern. Normally fio includes a header with some basic information and checksumming, but if this option is set, only the specific pattern set with *verify_pattern* is verified.

**pattern_hdr**
Verify a pattern in conjunction with a header.

**null**
Only pretend to verify. Useful for testing internals with *ioengine*=null, not for much else.

This option can be used for repeated burn-in tests of a system to make sure that the written data is also correctly read back.

If the data direction given is a read or random read, fio will assume that it should verify a previously written file. In this scenario fio will not verify the block number written in the header. The header seed won't be verified, unless its explicitly requested by setting *verify_header_seed*. Note in this scenario the header seed check will only work if the read invocation exactly matches the original write invocation.

If the data direction includes any form of write, the verify will be of the newly written data.

To avoid false verification errors, do not use the norandommap option when verifying data with async I/O engines and I/O depths > 1. Or use the norandommap and the lfsr random generator together to avoid writing to the same offset with multiple outstanding I/Os.

**verify_offset**=int
Swap the verification header with data somewhere else in the block before writing. It is swapped back before verifying. This should be within the range of *verify_interval*.

**verify_interval**=int
Write the verification header at a finer granularity than the *blocksize*. It will be written for chunks the size of `verify_interval`. *blocksize* should divide this evenly.

**verify_pattern**=str
If set, fio will fill the I/O buffers with this pattern. Fio defaults to filling with totally random bytes, but sometimes it's interesting to fill with a known pattern for I/O verification purposes. Depending on the width of the pattern, fio will fill 1/2/3/4 bytes of the buffer at the time (it can be either a decimal or a hex number). The `verify_pattern` if larger than a 32-bit quantity has to be a hex number that starts with either "0x" or "0X". Use with *verify*. Also, `verify_pattern` supports %o format, which means that for each block offset will be written and then verified back, e.g.:

```
verify_pattern=%o
```

Or use combination of everything:

```
verify_pattern=0xff%o"abcd"-12
```

**verify_pattern_interval**=bool

Recreate an instance of the *verify_pattern* every *verify_pattern_interval* bytes. This is only useful when *verify_pattern* contains the %o format specifier and can be used to speed up the process of writing each block on a device with its offset. Default: 0 (disabled).

**verify_fatal**=bool

Normally fio will keep checking the entire contents before quitting on a block verification failure. If this option is set, fio will exit the job on the first observed failure. Default: false.

**verify_dump**=bool

If set, dump the contents of both the original data block and the data block we read off disk to files. This allows later analysis to inspect just what kind of data corruption occurred. Off by default.

**verify_async**=int

Fio will normally verify I/O inline from the submitting thread. This option takes an integer describing how many async offload threads to create for I/O verification instead, causing fio to offload the duty of verifying I/O contents to one or more separate threads. If using this offload option, even sync I/O engines can benefit from using an *iodepth* setting higher than 1, as it allows them to have I/O in flight while verifies are running. Defaults to 0 async threads, i.e. verification is not asynchronous.

**verify_async_cpus**=str

Tell fio to set the given CPU affinity on the async I/O verification threads. See *cpus_allowed* for the format used.

**verify_backlog**=int

Fio will normally verify the written contents of a job that utilizes verify once that job has completed. In other words, everything is written then everything is read back and verified. You may want to verify continually instead for a variety of reasons. Fio stores the meta data associated with an I/O block in memory, so for large verify workloads, quite a bit of memory would be used up holding this meta data. If this option is enabled, fio will write only N blocks before verifying these blocks.

**verify_backlog_batch**=int

Control how many blocks fio will verify if *verify_backlog* is set. If not set, will default to the value of *verify_backlog* (meaning the entire queue is read back and verified). If verify_backlog_batch is less than *verify_backlog* then not all blocks will be verified, if verify_backlog_batch is larger than *verify_backlog*, some blocks will be verified more than once.

**verify_state_save**=bool

When a job exits during the write phase of a verify workload, save its current state. This allows fio to replay up until that point, if the verify state is loaded for the verify read phase. The format of the filename is, roughly:

```
<type>-<jobname>-<jobindex>-verify.state.
```

<type> is "local" for a local run, "sock" for a client/server socket connection, and "ip" (192.168.0.1, for instance) for a networked client/server connection. Defaults to true.

**verify_state_load**=bool

If a verify termination trigger was used, fio stores the current write state of each thread. This can be used at verification time so that fio knows how far it should verify. Without this information, fio will run a full verification pass, according to the settings in the job file used. Default false.

**experimental_verify**=bool

Enable experimental verification. Standard verify records I/O metadata for later use during the verification phase. Experimental verify instead resets the file after the write phase and then replays I/Os for the verification phase.

**verify_write_sequence**=bool

Verify the header write sequence number. In a scenario with multiple jobs, verification of the write sequence number may fail. Disabling this option will mean that write sequence number checking is skipped. Doing that can be useful for testing atomic writes, as it means that checksum verification can still be attempted. For when *atomic* is enabled, checksum verification is expected to succeed (while write sequence checking can still fail). Defaults to true.

**verify_header_seed**=bool

Verify the header seed value which was used to generate the buffer contents. In certain scenarios with read / verify only workloads, when *norandommap* is enabled, with offset modifiers (refer *readwrite* and *rw_sequencer*) etc verification of header seed may fail. Disabling this option will mean that header seed checking is skipped. Defaults to true.

**trim_percentage**=int

Number of verify blocks to discard/trim.

**trim_verify_zero**=bool

Verify that trim/discarded blocks are returned as zeros.

**trim_backlog**=int

Trim after this number of blocks are written.

**trim_backlog_batch**=int

Trim this number of I/O blocks.

### 1.13.18 Steady state

**steadystate**=str:float, **ss**=str:float

Define the criterion and limit for assessing steady state performance. The first parameter designates the criterion whereas the second parameter sets the threshold. When the criterion falls below the threshold for the specified duration, the job will stop. For example, *iops_slope:0.1%* will direct fio to terminate the job when the least squares regression slope falls below 0.1% of the mean IOPS. If *group_reporting* is enabled this will apply to all jobs in the group. Below is the list of available steady state assessment criteria. All assessments are carried out using only data from the rolling collection window. Threshold limits can be expressed as a fixed value or as a percentage of the mean in the collection window.

When using this feature, most jobs should include the *time_based* and *runtime* options or the *loops* option so that fio does not stop running after it has covered the full size of the specified file(s) or device(s).

**iops**
: Collect IOPS data. Stop the job if all individual IOPS measurements are within the specified limit of the mean IOPS (e.g., `iops:2` means that all individual IOPS values must be within 2 of the mean, whereas `iops:0.2%` means that all individual IOPS values must be within 0.2% of the mean IOPS to terminate the job).

**iops_slope**
: Collect IOPS data and calculate the least squares regression slope. Stop the job if the slope falls below the specified limit.

**bw**
: Collect bandwidth data. Stop the job if all individual bandwidth measurements are within the specified limit of the mean bandwidth.

**bw_slope**
Collect bandwidth data and calculate the least squares regression slope. Stop the job if the slope falls below the specified limit.

**steadystate_duration**=time, **ss_dur**=time

A rolling window of this duration will be used to judge whether steady state has been reached. Data will be collected every *ss_interval*. The default is 0 which disables steady state detection. When the unit is omitted, the value is interpreted in seconds.

**steadystate_ramp_time**=time, **ss_ramp**=time

Allow the job to run for the specified duration before beginning data collection for checking the steady state job termination criterion. The default is 0. When the unit is omitted, the value is interpreted in seconds.

**steadystate_check_interval**=time, **ss_interval**=time

The values during the rolling window will be collected with a period of this value. If *ss_interval* is 30s and *ss_dur* is 300s, 10 measurements will be taken. Default is 1s but that might not converge, especially for slower devices, so set this accordingly. When the unit is omitted, the value is interpreted in seconds.

### 1.13.19 Measurements and reporting

**per_job_logs**=bool

If set to true, fio generates bw/clat/iops logs with per job unique filenames. If set to false, jobs with identical names will share a log filename. Note that when this option is set to false log files will be opened in append mode and if log files already exist the previous contents will not be overwritten. Default: true.

**group_reporting**

It may sometimes be interesting to display statistics for groups of jobs as a whole instead of for each individual job. This is especially true if *numjobs* is used; looking at individual thread/process output quickly becomes unwieldy. To see the final report per-group instead of per-job, use *group_reporting*. Jobs in a file will be part of the same reporting group, unless if separated by a *stonewall*, or by using *new_group*.

NOTE: When *group_reporting* is used along with *json* output, there are certain per-job properties which can be different between jobs but do not have a natural group-level equivalent. Examples include *kb_base*, *unit_base*, *sig_figs*, *thread_number*, *pid*, and *job_start*. For these properties, the values for the first job are recorded for the group.

Also, options like *percentile_list* and *unified_rw_reporting* should be consistent among the jobs in a reporting group. Having options like these vary across the jobs in a reporting group is an unsupported configuration.

**new_group**

Start a new reporting group. See: *group_reporting*. If not given, all jobs in a file will be part of the same reporting group, unless separated by a *stonewall*.

**stats**=bool

By default, fio collects and shows final output results for all jobs that run. If this option is set to 0, then fio will ignore it in the final stat output.

**write_bw_log**=str

If given, write a bandwidth log for this job. Can be used to store data of the bandwidth of the jobs in their lifetime.

If no str argument is given, the default filename of `jobname_type.x.log` is used. Even when the argument is given, fio will still append the type of log. So if one specifies:

```
write_bw_log=foo
```

The actual log name will be `foo_bw.x.log` where *x* is the index of the job (*1..N*, where *N* is the number of jobs). If *per_job_logs* is false, then the filename will not include the *.x* job index.

The included **fio_generate_plots** script uses **gnuplot** to turn these text files into nice graphs. See *Log File Formats* for how data is structured within the file.

**write_lat_log**=str

Same as *write_bw_log*, except this option creates I/O submission (e.g., `name_slat.x.log`), completion (e.g., `name_clat.x.log`), and total (e.g., `name_lat.x.log`) latency files instead. See *write_bw_log* for details about the filename format and *Log File Formats* for how data is structured within the files.

**write_hist_log**=str

Same as *write_bw_log* but writes an I/O completion latency histogram file (e.g., `name_hist.x.log`) instead. Note that this file will be empty unless *log_hist_msec* has also been set. See *write_bw_log* for details about the filename format and *Log File Formats* for how data is structured within the file.

**write_iops_log**=str

Same as *write_bw_log*, but writes an IOPS file (e.g. `name_iops.x.log`) instead. Because fio defaults to individual I/O logging, the value entry in the IOPS log will be 1 unless windowed logging (see *log_avg_msec*) has been enabled. See *write_bw_log* for details about the filename format and *Log File Formats* for how data is structured within the file.

**log_entries**=int

By default, fio will log an entry in the iops, latency, or bw log for every I/O that completes. The initial number of I/O log entries is 1024. When the log entries are all used, new log entries are dynamically allocated. This dynamic log entry allocation may negatively impact time-related statistics such as I/O tail latencies (e.g. 99.9th percentile completion latency). This option allows specifying a larger initial number of log entries to avoid run-time allocations of new log entries, resulting in more precise time-related I/O statistics. Also see *log_avg_msec*. Defaults to 1024.

**log_avg_msec**=int

By default, fio will log an entry in the iops, latency, or bw log for every I/O that completes. When writing to the disk log, that can quickly grow to a very large size. Setting this option directs fio to instead record an average over the specified duration for each log entry, reducing the resolution of the log. When the job completes, fio will flush any accumulated latency log data, so the final log interval may not match the value specified by this option and there may even be duplicate timestamps. See *log_window_value* as well. Defaults to 0, logging entries for each I/O. Also see *Log File Formats*.

**log_hist_msec**=int

Same as *log_avg_msec*, but logs entries for completion latency histograms. Computing latency percentiles from averages of intervals using *log_avg_msec* is inaccurate. Setting this option makes fio log histogram entries over the specified period of time, reducing log sizes for high IOPS devices while retaining percentile accuracy. See *log_hist_coarseness* and *write_hist_log* as well. Defaults to 0, meaning histogram logging is disabled.

**log_hist_coarseness**=int

Integer ranging from 0 to 6, defining the coarseness of the resolution of the histogram logs enabled with *log_hist_msec*. For each increment in coarseness, fio outputs half as many bins. Defaults to 0, for which histogram logs contain 1216 latency bins. See *write_hist_log* and *Log File Formats*.

**log_window_value**=str, **log_max_value**=str

If *log_avg_msec* is set, fio by default logs the average over that window. This option determines whether fio logs the average, maximum or both the values over the window. This only affects the latency logging, as both average and maximum values for iops or bw log will be same. Accepted values are:

**avg**
: Log average value over the window. The default.

**max**
Log maximum value in the window.

**both**
Log both average and maximum value over the window.

**0**
Backward-compatible alias for **avg**.

**1**
Backward-compatible alias for **max**.

**log_offset**=bool

If this is set, the iolog options will include the byte offset for the I/O entry as well as the other data values. Defaults to 0 meaning that offsets are not present in logs. Also see *Log File Formats*.

**log_prio**=bool

If this is set, the *Command priority* field in *Log File Formats* shows the priority value and the IO priority class of the command. Otherwise, the field shows if the command has the highest RT priority class or not. Also see *Log File Formats*.

**log_issue_time**=bool

If this is set, the iolog options will include the command issue time for the I/O entry as well as the other data values. Defaults to 0 meaning that command issue times are not present in logs. Also see *Log File Formats*. This option shall be set together with *write_lat_log* and *log_offset*.

**log_compression**=int

If this is set, fio will compress the I/O logs as it goes, to keep the memory footprint lower. When a log reaches the specified size, that chunk is removed and compressed in the background. Given that I/O logs are fairly highly compressible, this yields a nice memory savings for longer runs. The downside is that the compression will consume some background CPU cycles, so it may impact the run. This, however, is also true if the logging ends up consuming most of the system memory. So pick your poison. The I/O logs are saved normally at the end of a run, by decompressing the chunks and storing them in the specified log file. This feature depends on the availability of zlib.

**log_compression_cpus**=str

Define the set of CPUs that are allowed to handle online log compression for the I/O jobs. This can provide better isolation between performance sensitive jobs, and background compression work. See *cpus_allowed* for the format used.

**log_store_compressed**=bool

If set, fio will store the log files in a compressed format. They can be decompressed with fio, using the *--inflate-log* command line parameter. The files will be stored with a `.fz` suffix.

**log_unix_epoch**=bool

Backwards compatible alias for log_alternate_epoch.

**log_alternate_epoch**=bool

If set, fio will log timestamps based on the epoch used by the clock specified in the log_alternate_epoch_clock_id option, to the log files produced by enabling write_type_log for each log type, instead of the default zero-based timestamps.

**log_alternate_epoch_clock_id**=int

Specifies the clock_id to be used by clock_gettime to obtain the alternate epoch if log_alternate_epoch is true. Otherwise has no effect. Default value is 0, or CLOCK_REALTIME.

**block_error_percentiles**=bool

If set, record errors in trim block-sized units from writes and trims and output a histogram of how many trims it took to get to errors, and what kind of error was encountered.

**bwavgtime**=int

Average the calculated bandwidth over the given time. Value is specified in milliseconds. If the job also does bandwidth logging through *write_bw_log*, then the minimum of this option and *log_avg_msec* will be used. Default: 500ms.

**iopsavgtime**=int

Average the calculated IOPS over the given time. Value is specified in milliseconds. If the job also does IOPS logging through *write_iops_log*, then the minimum of this option and *log_avg_msec* will be used. Default: 500ms.

**disk_util**=bool

Generate disk utilization statistics, if the platform supports it. Default: true.

**disable_lat**=bool

Disable measurements of total latency numbers. Useful only for cutting back the number of calls to *gettimeofday(2)*, as that does impact performance at really high IOPS rates. Note that to really get rid of a large amount of these calls, this option must be used with *disable_slat* and *disable_bw_measurement* as well.

**disable_clat**=bool

Disable measurements of completion latency numbers. See *disable_lat*.

**disable_slat**=bool

Disable measurements of submission latency numbers. See *disable_lat*.

**disable_bw_measurement**=bool, **disable_bw**=bool

Disable measurements of throughput/bandwidth numbers. See *disable_lat*.

**slat_percentiles**=bool

Report submission latency percentiles. Submission latency is not recorded for synchronous ioengines.

**clat_percentiles**=bool

Report completion latency percentiles.

**lat_percentiles**=bool

Report total latency percentiles. Total latency is the sum of submission latency and completion latency.

**percentile_list**=float_list

Overwrite the default list of percentiles for latencies and the block error histogram. Each number is a floating point number in the range (0,100], and the maximum length of the list is 20. Use `:` to separate the numbers. For example, `--percentile_list=99.5:99.9` will cause fio to report the latency durations below which 99.5% and 99.9% of the observed latencies fell, respectively.

**significant_figures**=int

If using *--output-format* of *normal*, set the significant figures to this value. Higher values will yield more precise IOPS and throughput units, while lower values will round. Requires a minimum value of 1 and a maximum value of 10. Defaults to 4.

### 1.13.20 Error handling

**exitall_on_error**

When one job finishes in error, terminate the rest. The default is to wait for each job to finish.

**continue_on_error**=str

Normally fio will exit the job on the first observed failure. If this option is set, fio will continue the job when there is a 'non-fatal error' (EIO or EILSEQ) until the runtime is exceeded or the I/O size specified is completed. If this option is used, there are two more stats that are appended, the total error count and the first error. The error field given in the stats is the first error that was hit during the run.

Note: a write error from the device may go unnoticed by fio when using buffered IO, as the write() (or similar) system call merely dirties the kernel pages, unless *sync* or *direct* is used. Device IO errors occur when the dirty data is actually written out to disk. If fully sync writes aren't desirable, *fsync* or *fdatasync* can be used as well. This is specific to writes, as reads are always synchronous.

The allowed values are:

**none**
Exit on any I/O or verify errors.

**read**
Continue on read errors, exit on all others.

**write**
Continue on write errors, exit on all others.

**io**
Continue on any I/O error, exit on all others.

**verify**
Continue on verify errors, exit on all others.

**all**
Continue on all errors.

**0**
Backward-compatible alias for 'none'.

**1**
Backward-compatible alias for 'all'.

**ignore_error**=str

Sometimes you want to ignore some errors during test in that case you can specify error list for each error type, instead of only being able to ignore the default 'non-fatal error' using *continue_on_error*. `ignore_error=READ_ERR_LIST,WRITE_ERR_LIST,VERIFY_ERR_LIST` errors for given error type is separated with ':'. Error may be symbol ('ENOSPC', 'ENOMEM') or integer. Example:

```
ignore_error=EAGAIN,ENOSPC:122
```

This option will ignore EAGAIN from READ, and ENOSPC and 122(EDQUOT) from WRITE. This option works by overriding *continue_on_error* with the list of errors for each error type if any.

**error_dump**=bool

If set dump every error even if it is non fatal, true by default. If disabled only fatal error will be dumped.

## 1.14 Running predefined workloads

Fio includes predefined profiles that mimic the I/O workloads generated by other tools.

**profile**=str

The predefined workload to run. Current profiles are:

**tiobench**
Threaded I/O bench (tiotest/tiobench) like workload.

**act**
Aerospike Certification Tool (ACT) like workload.

To view a profile's additional options use *--cmdhelp* after specifying the profile. For example:

```
$ fio --profile=act --cmdhelp
```

### 1.14.1 Act profile options

**device-names**=str
Devices to use.

**load**=int
ACT load multiplier. Default: 1.

**test-duration**=time
How long the entire test takes to run. When the unit is omitted, the value is given in seconds. Default: 24h.

**threads-per-queue**=int
Number of read I/O threads per device. Default: 8.

**read-req-num-512-blocks**=int
Number of 512B blocks to read at the time. Default: 3.

**large-block-op-kbytes**=int
Size of large block ops in KiB (writes). Default: 131072.

**prep**
Set to run ACT prep phase.

### 1.14.2 Tiobench profile options

**size**=str
Size in MiB.

**block**=int
Block size in bytes. Default: 4096.

**numruns**=int
Number of runs.

**dir**=str
Test directory.

**threads**=int
Number of threads.

## 1.15 Interpreting the output

Fio spits out a lot of output. While running, fio will display the status of the jobs created. An example of that would be:

```
Jobs: 1 (f=1): [_(1),M(1)][24.8%][r=20.5MiB/s,w=23.5MiB/s][r=82,w=94 IOPS][eta 01m:31s]
```

The characters inside the first set of square brackets denote the current status of each thread. The first character is the first job defined in the job file, and so forth. The possible values (in typical life cycle order) are:

| Idle | Run | |
|---|---|---|
| P | | Thread setup, but not started. |
| C | | Thread created. |
| I | | Thread initialized, waiting or generating necessary data. |
| | p | Thread running pre-reading file(s). |
| | / | Thread is in ramp period. |
| | R | Running, doing sequential reads. |
| | r | Running, doing random reads. |
| | W | Running, doing sequential writes. |
| | w | Running, doing random writes. |
| | M | Running, doing mixed sequential reads/writes. |
| | m | Running, doing mixed random reads/writes. |
| | D | Running, doing sequential trims. |
| | d | Running, doing random trims. |
| | F | Running, currently waiting for *fsync(2)*. |
| | V | Running, doing verification of written data. |
| f | | Thread finishing. |
| E | | Thread exited, not reaped by main thread yet. |
| _ | | Thread reaped. |
| X | | Thread reaped, exited with an error. |
| K | | Thread reaped, exited due to signal. |

Fio will condense the thread string as not to take up more space on the command line than needed. For instance, if you have 10 readers and 10 writers running, the output would look like this:

```
Jobs: 20 (f=20): [R(10),W(10)][4.0%][r=20.5MiB/s,w=23.5MiB/s][r=82,w=94 IOPS][eta␣
↪57m:36s]
```

Note that the status string is displayed in order, so it's possible to tell which of the jobs are currently doing what. In the example above this means that jobs 1–10 are readers and 11–20 are writers.

The other values are fairly self explanatory – number of threads currently running and doing I/O, the number of currently open files (f=), the estimated completion percentage, the rate of I/O since last check (read speed listed first, then write speed and optionally trim speed) in terms of bandwidth and IOPS, and time to completion for the current running group. It's impossible to estimate runtime of the following groups (if any).

When fio is done (or interrupted by `Ctrl-C`), it will show the data for each thread, group of threads, and disks in that order. For each overall thread (or group) the output looks like:

```
Client1: (groupid=0, jobs=1): err= 0: pid=16109: Sat Jun 24 12:07:54 2017
  write: IOPS=88, BW=623KiB/s (638kB/s)(30.4MiB/50032msec)
    slat (nsec): min=500, max=145500, avg=8318.00, stdev=4781.50
    clat (usec): min=170, max=78367, avg=4019.02, stdev=8293.31
     lat (usec): min=174, max=78375, avg=4027.34, stdev=8291.79
    clat percentiles (usec):
     |  1.00th=[  302],  5.00th=[  326], 10.00th=[  343], 20.00th=[  363],
     | 30.00th=[  392], 40.00th=[  404], 50.00th=[  416], 60.00th=[  445],
     | 70.00th=[  816], 80.00th=[ 6718], 90.00th=[12911], 95.00th=[21627],
     | 99.00th=[43779], 99.50th=[51643], 99.90th=[68682], 99.95th=[72877],
     | 99.99th=[78119]
```

(continues on next page)

(continued from previous page)

```
  bw (  KiB/s): min=  532, max=  686, per=0.10%, avg=622.87, stdev=24.82, samples=  100
  iops        : min=   76, max=   98, avg=88.98, stdev= 3.54, samples=  100
 lat (usec)   : 250=0.04%, 500=64.11%, 750=4.81%, 1000=2.79%
 lat (msec)   : 2=4.16%, 4=1.84%, 10=4.90%, 20=11.33%, 50=5.37%
 lat (msec)   : 100=0.65%
 cpu          : usr=0.27%, sys=0.18%, ctx=12072, majf=0, minf=21
 IO depths    : 1=85.0%, 2=13.1%, 4=1.8%, 8=0.1%, 16=0.0%, 32=0.0%, >=64=0.0%
    submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
    complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
    issued rwt: total=0,4450,0, short=0,0,0, dropped=0,0,0
    latency   : target=0, window=0, percentile=100.00%, depth=8
```

The job name (or first job's name when using *group_reporting*) is printed, along with the group id, count of jobs being aggregated, last error id seen (which is 0 when there are no errors), pid/tid of that thread and the time the job/group completed. Below are the I/O statistics for each data direction performed (showing writes in the example above). In the order listed, they denote:

**read/write/trim**
: The string before the colon shows the I/O direction the statistics are for. **IOPS** is the average I/Os performed per second. **BW** is the average bandwidth rate shown as: value in power of 2 format (value in power of 10 format). The last two values show: (**total I/O performed** in power of 2 format **/ runtime** of that thread).

**slat**
: Submission latency (**min** being the minimum, **max** being the maximum, **avg** being the average, **stdev** being the standard deviation). This is the time from when fio initialized the I/O to submission. For synchronous ioengines this includes the time up until just before the ioengine's queue function is called. For asynchronous ioengines this includes the time up through the completion of the ioengine's queue function (and commit function if it is defined). For sync I/O this row is not displayed as the slat is negligible. This value can be in nanoseconds, microseconds or milliseconds — fio will choose the most appropriate base and print that (in the example above nanoseconds was the best scale). Note: in *--minimal* mode latencies are always expressed in microseconds.

**clat**
: Completion latency. Same names as slat, this denotes the time from submission to completion of the I/O pieces. For sync I/O, this represents the time from when the I/O was submitted to the operating system to when it was completed. For asynchronous ioengines this is the time from when the ioengine's queue (and commit if available) functions were completed to when the I/O's completion was reaped by fio.

  For file and directory operation engines, **clat** denotes the time to complete one file or directory operation.

  **filecreate engine**:the time cost to create a new file

  **filestat engine**: the time cost to look up an existing file

  **filedelete engine**:the time cost to delete a file

  **dircreate engine**: the time cost to create a new directory

  **dirstat engine**: the time cost to look up an existing directory

  **dirdelete engine**: the time cost to delete a directory

**lat**
: Total latency. Same names as slat and clat, this denotes the time from when fio created the I/O unit to completion of the I/O operation. It is the sum of submission and completion latency.

**bw**
: Bandwidth statistics based on measurements from discrete intervals. Fio continuously monitors bytes transferred and I/O operations completed. By default fio calculates bandwidth in each half-second interval (see *bwavgtime*) and reports descriptive statistics for the measurements here. Same names as the xlat stats, but also includes the

number of samples taken (**samples**) and an approximate percentage of total aggregate bandwidth this thread received in its group (**per**). This last value is only really useful if the threads in this group are on the same disk, since they are then competing for disk access.

For file and directory operation engines, **bw** is meaningless.

**iops**
IOPS statistics based on measurements from discrete intervals. For details see the description for bw above. See *iopsavgtime* to control the duration of the intervals. Same values reported here as for bw except for percentage.

For file and directory operation engines, **iops** is the most fundamental index to denote the performance. It means how many files or directories can be operated per second.

**filecreate engine**:number of files can be created per second

**filestat engine**: number of files can be looked up per second

**filedelete engine**:number of files can be deleted per second

**dircreate engine**: number of directories can be created per second

**dirstat engine**: number of directories can be looked up per second

**dirdelete engine**: number of directories can be deleted per second

**lat (nsec/usec/msec)**
The distribution of I/O completion latencies. This is the time from when I/O leaves fio and when it gets completed. Unlike the separate read/write/trim sections above, the data here and in the remaining sections apply to all I/Os for the reporting group. 250=0.04% means that 0.04% of the I/Os completed in under 250us. 500=64.11% means that 64.11% of the I/Os required 250 to 499us for completion.

**cpu**
CPU usage. User and system time, along with the number of context switches this thread went through, usage of system and user time, and finally the number of major and minor page faults. The CPU utilization numbers are averages for the jobs in that reporting group, while the context and fault counters are summed.

**IO depths**
The distribution of I/O depths over the job lifetime. The numbers are divided into powers of 2 and each entry covers depths from that value up to those that are lower than the next entry – e.g., 16= covers depths from 16 to 31. Note that the range covered by a depth distribution entry can be different to the range covered by the equivalent submit/complete distribution entry.

**IO submit**
How many pieces of I/O were submitting in a single submit call. Each entry denotes that amount and below, until the previous entry – e.g., 16=100% means that we submitted anywhere between 9 to 16 I/Os per submit call. Note that the range covered by a submit distribution entry can be different to the range covered by the equivalent depth distribution entry.

**IO complete**
Like the above submit number, but for completions instead.

**IO issued rwt**
The number of read/write/trim requests issued, and how many of them were short or dropped.

**IO latency**
These values are for *latency_target* and related options. When these options are engaged, this section describes the I/O depth required to meet the specified latency target.

After each client has been listed, the group statistics are printed. They will look like this:

```
Run status group 0 (all jobs):
   READ: bw=20.9MiB/s (21.9MB/s), 10.4MiB/s-10.8MiB/s (10.9MB/s-11.3MB/s), io=64.0MiB␣
```

(continues on next page)

(continued from previous page)

```
↪(67.1MB), run=2973-3069msec
  WRITE: bw=1231KiB/s (1261kB/s), 616KiB/s-621KiB/s (630kB/s-636kB/s), io=64.0MiB (67.
↪1MB), run=52747-53223msec
```

For each data direction it prints:

**bw**
: Aggregate bandwidth of threads in this group followed by the minimum and maximum bandwidth of all the threads in this group. Values outside of brackets are power-of-2 format and those within are the equivalent value in a power-of-10 format.

**io**
: Aggregate I/O performed of all threads in this group. The format is the same as bw.

**run**
: The smallest and longest runtimes of the threads in this group.

And finally, the disk statistics are printed. This is Linux specific. They will look like this:

```
Disk stats (read/write):
  sda: ios=16398/16511, sectors=32321/65472, merge=30/162, ticks=6853/819634, in_
↪queue=826487, util=100.00%
```

Each value is printed for both reads and writes, with reads first. The numbers denote:

**ios**
: Number of I/Os performed by all groups.

**sectors**
: Amount of data transferred in units of 512 bytes for all groups.

**merge**
: Number of merges performed by the I/O scheduler.

**ticks**
: Number of ticks we kept the disk busy.

**in_queue**
: Total time spent in the disk queue.

**util**
: The disk utilization. A value of 100% means we kept the disk busy constantly, 50% would be a disk idling half of the time.

It is also possible to get fio to dump the current output while it is running, without terminating the job. To do that, send fio the **USR1** signal. You can also get regularly timed dumps by using the *--status-interval* parameter, or by creating a file in `/tmp` named `fio-dump-status`. If fio sees this file, it will unlink it and dump the current output status.

## 1.16 Terse output

For scripted usage where you typically want to generate tables or graphs of the results, fio can output the results in a semicolon separated format. The format is one long line of values, such as:

```
2;card0;0;0;7139336;121836;60004;1;10109;27.932460;116.933948;220;126861;3495.446807;
↪1085.368601;226;126864;3523.635629;1089.012448;24063;99944;50.275485%;59818.274627;
↪5540.657370;7155060;122104;60004;1;8338;29.086342;117.839068;388;128077;5032.488518;
```

(continues on next page)

(continued from previous page)

```
↪1234.785715;391;128085;5061.839412;1236.909129;23436;100928;50.287926%;59964.832030;
↪5644.844189;14.595833%;19.394167%;123706;0;7313;0.1%;0.1%;0.1%;0.1%;0.1%;0.1%;100.0%;0.
↪00%;0.00%;0.00%;0.00%;0.00%;0.00%;0.01%;0.02%;0.05%;0.16%;6.04%;40.40%;52.68%;0.64%;0.
↪01%;0.00%;0.01%;0.00%;0.00%;0.00%;0.00%;0.00%
A description of this job goes here.
```

The job description (if provided) follows on a second line for terse v2. It appears on the same line for other terse versions.

To enable terse output, use the *--minimal* or *--output-format*=terse command line options. The first value is the version of the terse output format. If the output has to be changed for some reason, this number will be incremented by 1 to signify that change.

Split up, the format is as follows (comments in brackets denote when a field was introduced or whether it's specific to some terse version):

```
terse version, fio version [v3], jobname, groupid, error
```

READ status:

```
Total IO (KiB), bandwidth (KiB/sec), IOPS, runtime (msec)
Submission latency: min, max, mean, stdev (usec)
Completion latency: min, max, mean, stdev (usec)
Completion latency percentiles: 20 fields (see below)
Total latency: min, max, mean, stdev (usec)
Bw (KiB/s): min, max, aggregate percentage of total, mean, stdev, number of␣
↪samples [v5]
IOPS [v5]: min, max, mean, stdev, number of samples
```

WRITE status:

```
Total IO (KiB), bandwidth (KiB/sec), IOPS, runtime (msec)
Submission latency: min, max, mean, stdev (usec)
Completion latency: min, max, mean, stdev (usec)
Completion latency percentiles: 20 fields (see below)
Total latency: min, max, mean, stdev (usec)
Bw (KiB/s): min, max, aggregate percentage of total, mean, stdev, number of␣
↪samples [v5]
IOPS [v5]: min, max, mean, stdev, number of samples
```

TRIM status [all but version 3]:

Fields are similar to READ/WRITE status.

CPU usage:

```
user, system, context switches, major faults, minor faults
```

I/O depths:

```
<=1, 2, 4, 8, 16, 32, >=64
```

I/O latencies microseconds:

```
<=2, 4, 10, 20, 50, 100, 250, 500, 750, 1000
```

I/O latencies milliseconds:

```
<=2, 4, 10, 20, 50, 100, 250, 500, 750, 1000, 2000, >=2000
```

Disk utilization [v3]:

```
disk name, read ios, write ios, read merges, write merges, read ticks, write␣
↪ticks,
time spent in queue, disk utilization percentage
```

Additional Info (dependent on continue_on_error, default off):

```
total # errors, first error code
```

Additional Info (dependent on description being set):

```
Text description
```

Completion latency percentiles can be a grouping of up to 20 sets, so for the terse output fio writes all of them. Each field will look like this:

```
1.00%=6112
```

which is the Xth percentile, and the *usec* latency associated with it.

For *Disk utilization*, all disks used by fio are shown. So for each disk there will be a disk utilization section.

Below is a single line containing short names for each of the fields in the minimal output v3, separated by semicolons:

```
terse_version_3;fio_version;jobname;groupid;error;read_kb;read_bandwidth_kb;read_iops;
↪read_runtime_ms;read_slat_min_us;read_slat_max_us;read_slat_mean_us;read_slat_dev_us;
↪read_clat_min_us;read_clat_max_us;read_clat_mean_us;read_clat_dev_us;read_clat_pct01;
↪read_clat_pct02;read_clat_pct03;read_clat_pct04;read_clat_pct05;read_clat_pct06;read_
↪clat_pct07;read_clat_pct08;read_clat_pct09;read_clat_pct10;read_clat_pct11;read_clat_
↪pct12;read_clat_pct13;read_clat_pct14;read_clat_pct15;read_clat_pct16;read_clat_pct17;
↪read_clat_pct18;read_clat_pct19;read_clat_pct20;read_tlat_min_us;read_lat_max_us;read_
↪lat_mean_us;read_lat_dev_us;read_bw_min_kb;read_bw_max_kb;read_bw_agg_pct;read_bw_mean_
↪kb;read_bw_dev_kb;write_kb;write_bandwidth_kb;write_iops;write_runtime_ms;write_slat_
↪min_us;write_slat_max_us;write_slat_mean_us;write_slat_dev_us;write_clat_min_us;write_
↪clat_max_us;write_clat_mean_us;write_clat_dev_us;write_clat_pct01;write_clat_pct02;
↪write_clat_pct03;write_clat_pct04;write_clat_pct05;write_clat_pct06;write_clat_pct07;
↪write_clat_pct08;write_clat_pct09;write_clat_pct10;write_clat_pct11;write_clat_pct12;
↪write_clat_pct13;write_clat_pct14;write_clat_pct15;write_clat_pct16;write_clat_pct17;
↪write_clat_pct18;write_clat_pct19;write_clat_pct20;write_tlat_min_us;write_lat_max_us;
↪write_lat_mean_us;write_lat_dev_us;write_bw_min_kb;write_bw_max_kb;write_bw_agg_pct;
↪write_bw_mean_kb;write_bw_dev_kb;cpu_user;cpu_sys;cpu_csw;cpu_mjf;cpu_minf;iodepth_1;
↪iodepth_2;iodepth_4;iodepth_8;iodepth_16;iodepth_32;iodepth_64;lat_2us;lat_4us;lat_
↪10us;lat_20us;lat_50us;lat_100us;lat_250us;lat_500us;lat_750us;lat_1000us;lat_2ms;lat_
↪4ms;lat_10ms;lat_20ms;lat_50ms;lat_100ms;lat_250ms;lat_500ms;lat_750ms;lat_1000ms;lat_
↪2000ms;lat_over_2000ms;disk_name;disk_read_iops;disk_write_iops;disk_read_merges;disk_
↪write_merges;disk_read_ticks;write_ticks;disk_queue_time;disk_util
```

In client/server mode terse output differs from what appears when jobs are run locally. Disk utilization data is omitted from the standard terse output and for v3 and later appears on its own separate line at the end of each terse reporting cycle.

## 1.17 JSON output

The *json* output format is intended to be both human readable and convenient for automated parsing. For the most part its sections mirror those of the *normal* output. The *runtime* value is reported in msec and the *bw* value is reported in 1024 bytes per second units.

## 1.18 JSON+ output

The *json+* output format is identical to the *json* output format except that it adds a full dump of the completion latency bins. Each *bins* object contains a set of (key, value) pairs where keys are latency durations and values count how many I/Os had completion latencies of the corresponding duration. For example, consider:

> "bins" : { "87552" : 1, "89600" : 1, "94720" : 1, "96768" : 1, "97792" : 1, "99840" : 1, "100864" : 2, "103936" : 6, "104960" : 534, "105984" : 5995, "107008" : 7529, ... }

This data indicates that one I/O required 87,552ns to complete, two I/Os required 100,864ns to complete, and 7529 I/Os required 107,008ns to complete.

Also included with fio is a Python script *fio_jsonplus_clat2csv* that takes json+ output and generates CSV-formatted latency data suitable for plotting.

The latency durations actually represent the midpoints of latency intervals. For details refer to `stat.h`.

## 1.19 Trace file format

There are two trace file format that you can encounter. The older (v1) format is unsupported since version 1.20-rc3 (March 2008). It will still be described below in case that you get an old trace and want to understand it.

In any case the trace is a simple text file with a single action per line.

### 1.19.1 Trace file format v1

Each line represents a single I/O action in the following format:

```
rw, offset, length
```

where *rw=0/1* for read/write, and the *offset* and *length* entries being in bytes.

This format is not supported in fio versions >= 1.20-rc3.

### 1.19.2 Trace file format v2

The second version of the trace file format was added in fio version 1.17. It allows one to access more than one file per trace and has a bigger set of possible file actions.

The first line of the trace file has to be:

```
fio version 2 iolog
```

Following this can be lines in two different formats, which are described below.

The file management format:

```
filename action
```

The *filename* is given as an absolute path. The *action* can be one of these:

**add**
Add the given *filename* to the trace.

**open**
Open the file with the given *filename*. The *filename* has to have been added with the **add** action before.

**close**
Close the file with the given *filename*. The file has to have been opened before.

The file I/O action format:

```
filename action offset length
```

The *filename* is given as an absolute path, and has to have been added and opened before it can be used with this format. The *offset* and *length* are given in bytes. The *action* can be one of these:

**wait**
Wait for *offset* microseconds. Everything below 100 is discarded. The time is relative to the previous *wait* statement. Note that action *wait* is not allowed as of version 3, as the same behavior can be achieved using timestamps.

**read**
Read *length* bytes beginning from *offset*.

**write**
Write *length* bytes beginning from *offset*.

**sync**
*fsync(2)* the file.

**datasync**
*fdatasync(2)* the file.

**trim**
Trim the given file from the given *offset* for *length* bytes.

### 1.19.3 Trace file format v3

The third version of the trace file format was added in fio version 3.31. It forces each action to have a timestamp associated with it.

The first line of the trace file has to be:

```
fio version 3 iolog
```

Following this can be lines in two different formats, which are described below.

The file management format:

```
timestamp filename action
```

The file I/O action format:

```
timestamp filename action offset length
```

The *timestamp* is relative to the beginning of the run (ie starts at 0). The *filename*, *action*, *offset* and *length* are identical to version 2, except that version 3 does not allow the *wait* action.

## 1.20 I/O Replay - Merging Traces

Colocation is a common practice used to get the most out of a machine. Knowing which workloads play nicely with each other and which ones don't is a much harder task. While fio can replay workloads concurrently via multiple jobs, it leaves some variability up to the scheduler making results harder to reproduce. Merging is a way to make the order of events consistent.

Merging is integrated into I/O replay and done when a *merge_blktrace_file* is specified. The list of files passed to *read_iolog* go through the merge process and output a single file stored to the specified file. The output file is passed on as if it were the only file passed to *read_iolog*. An example would look like:

```
$ fio --read_iolog="<file1>:<file2>" --merge_blktrace_file="<output_file>"
```

Creating only the merged file can be done by passing the command line argument *--merge-blktrace-only*.

Scaling traces can be done to see the relative impact of any particular trace being slowed down or sped up. *merge_blktrace_scalars* takes in a colon separated list of percentage scalars. It is index paired with the files passed to *read_iolog*.

With scaling, it may be desirable to match the running time of all traces. This can be done with *merge_blktrace_iters*. It is index paired with *read_iolog* just like *merge_blktrace_scalars*.

In an example, given two traces, A and B, each 60s long. If we want to see the impact of trace A issuing IOs twice as fast and repeat trace A over the runtime of trace B, the following can be done:

```
$ fio --read_iolog="<trace_a>:"<trace_b>" --merge_blktrace_file"<output_file>" --merge_
↪blktrace_scalars="50:100" --merge_blktrace_iters="2:1"
```

This runs trace A at 2x the speed twice for approximately the same runtime as a single run of trace B.

## 1.21 CPU idleness profiling

In some cases, we want to understand CPU overhead in a test. For example, we test patches for the specific goodness of whether they reduce CPU usage. Fio implements a balloon approach to create a thread per CPU that runs at idle priority, meaning that it only runs when nobody else needs the cpu. By measuring the amount of work completed by the thread, idleness of each CPU can be derived accordingly.

An unit work is defined as touching a full page of unsigned characters. Mean and standard deviation of time to complete an unit work is reported in "unit work" section. Options can be chosen to report detailed percpu idleness or overall system idleness by aggregating percpu stats.

## 1.22 Verification and triggers

Fio is usually run in one of two ways, when data verification is done. The first is a normal write job of some sort with verify enabled. When the write phase has completed, fio switches to reads and verifies everything it wrote. The second model is running just the write phase, and then later on running the same job (but with reads instead of writes) to repeat the same I/O patterns and verify the contents. Both of these methods depend on the write phase being completed, as fio otherwise has no idea how much data was written.

With verification triggers, fio supports dumping the current write state to local files. Then a subsequent read verify workload can load this state and know exactly where to stop. This is useful for testing cases where power is cut to a server in a managed fashion, for instance.

A verification trigger consists of two things:

1) Storing the write state of each job.

2) Executing a trigger command.

The write state is relatively small, on the order of hundreds of bytes to single kilobytes. It contains information on the number of completions done, the last X completions, etc.

A trigger is invoked either through creation ('touch') of a specified file in the system, or through a timeout setting. If fio is run with *--trigger-file*= `/tmp/trigger-file`, then it will continually check for the existence of `/tmp/trigger-file`. When it sees this file, it will fire off the trigger (thus saving state, and executing the trigger command).

For client/server runs, there's both a local and remote trigger. If fio is running as a server backend, it will send the job states back to the client for safe storage, then execute the remote trigger, if specified. If a local trigger is specified, the server will still send back the write state, but the client will then execute the trigger.

### 1.22.1 Verification trigger example

Let's say we want to run a powercut test on the remote Linux machine 'server'. Our write workload is in `write-test.fio`. We want to cut power to 'server' at some point during the run, and we'll run this test from the safety or our local machine, 'localbox'. On the server, we'll start the fio backend normally:

```
server# fio --server
```

and on the client, we'll fire off the workload:

```
localbox$ fio --client=server --trigger-file=/tmp/my-trigger --trigger-remote="bash -c \
↪"echo b > /proc/sysrq-triger\""
```

We set `/tmp/my-trigger` as the trigger file, and we tell fio to execute:

```
echo b > /proc/sysrq-trigger
```

on the server once it has received the trigger and sent us the write state. This will work, but it's not **really** cutting power to the server, it's merely abruptly rebooting it. If we have a remote way of cutting power to the server through IPMI or similar, we could do that through a local trigger command instead. Let's assume we have a script that does IPMI reboot of a given hostname, ipmi-reboot. On localbox, we could then have run fio with a local trigger instead:

```
localbox$ fio --client=server --trigger-file=/tmp/my-trigger --trigger="ipmi-reboot␣
↪server"
```

For this case, fio would wait for the server to send us the write state, then execute `ipmi-reboot server` when that happened.

### 1.22.2 Loading verify state

To load stored write state, a read verification job file must contain the *verify_state_load* option. If that is set, fio will load the previously stored state. For a local fio run this is done by loading the files directly, and on a client/server run, the server backend will ask the client to send the files over and load them from there.

## 1.23 Log File Formats

Fio supports a variety of log file formats, for logging latencies, bandwidth, and IOPS. The logs share a common format, which looks like this:

*time* (*msec*), *value*, *data direction*, *block size* (*bytes*), *offset* (*bytes*), *command priority*, *issue time* (*nsec*)

*Time* for the log entry is always in milliseconds. The *value* logged depends on the type of log, it will be one of the following:

**Latency log**
Value is latency in nsecs

**Bandwidth log**
Value is in KiB/sec

**IOPS log**
Value is IOPS

*Data direction* is one of the following:

**0**
I/O is a READ

**1**
I/O is a WRITE

**2**
I/O is a TRIM

The entry's *block size* is always in bytes. The *offset* is the position in bytes from the start of the file for that particular I/O. The logging of the offset can be toggled with *log_offset*.

If *log_prio* is not set, the entry's *Command priority* is 1 for an IO executed with the highest RT priority class (*prioclass* =1 or *cmdprio_class* =1) and 0 otherwise. This is controlled by the *prioclass* option and the io-engine specific *cmdprio_percentage* *cmdprio_class* options. If *log_prio* is set, the entry's *Command priority* is the priority set for the IO, as a 16-bits hexadecimal number with the lowest 13 bits indicating the priority value (*prio* and *cmdprio* options) and the highest 3 bits indicating the IO priority class (*prioclass* and *cmdprio_class* options).

The entry's *issue time* is the command issue time in nanoseconds. The logging of the issue time can be toggled with *log_issue_time*. This field has valid values in completion latency log file (clat), or submit latency log file (slat). The field has value 0 in other logs files.

Fio defaults to logging every individual I/O but when windowed logging is set through *log_avg_msec*, either the average (by default), the maximum (*log_window_value* is set to max) *value* seen over the specified period of time, or both the average *value* and maximum *value1* (*log_window_value* is set to both) is recorded. The log file format when both the values are reported takes this form:

*time* (*msec*), *value*, *value1*, *data direction*, *block size* (*bytes*), *offset* (*bytes*), *command priority*, *issue time* (*nsec*)

Each *data direction* seen within the window period will aggregate its values in a separate row. Further, when using windowed logging the *block size*, *offset* and *issue time* entries will always contain 0.

## 1.24 Client/Server

Normally fio is invoked as a stand-alone application on the machine where the I/O workload should be generated. However, the backend and frontend of fio can be run separately i.e., the fio server can generate an I/O workload on the "Device Under Test" while being controlled by a client on another machine.

Start the server on the machine which has access to the storage DUT:

```
$ fio --server=args
```

where *args* defines what fio listens to. The arguments are of the form `type,hostname` or `IP,port`. *type* is either `ip` (or ip4) for TCP/IP v4, `ip6` for TCP/IP v6, or `sock` for a local unix domain socket. *hostname* is either a hostname or IP address, and *port* is the port to listen to (only valid for TCP/IP, not a local socket). Some examples:

1) `fio --server`

Start a fio server, listening on all interfaces on the default port (8765).

2) `fio --server=ip:hostname,4444`

   Start a fio server, listening on IP belonging to hostname and on port 4444.

3) `fio --server=ip6:::1,4444`

   Start a fio server, listening on IPv6 localhost ::1 and on port 4444.

4) `fio --server=,4444`

   Start a fio server, listening on all interfaces on port 4444.

5) `fio --server=1.2.3.4`

   Start a fio server, listening on IP 1.2.3.4 on the default port.

6) `fio --server=sock:/tmp/fio.sock`

   Start a fio server, listening on the local socket `/tmp/fio.sock`.

Once a server is running, a "client" can connect to the fio server with:

```
fio <local-args> --client=<server> <remote-args> <job file(s)>
```

where *local-args* are arguments for the client where it is running, *server* is the connect string, and *remote-args* and *job file(s)* are sent to the server. The *server* string follows the same format as it does on the server side, to allow IP/hostname/socket and port strings.

Note that all job options must be defined in job files when running fio as a client. Any job options specified in *remote-args* will be ignored.

Fio can connect to multiple servers this way:

```
fio --client=<server1> <job file(s)> --client=<server2> <job file(s)>
```

If the job file is located on the fio server, then you can tell the server to load a local file as well. This is done by using *--remote-config*

```
fio --client=server --remote-config /path/to/file.fio
```

Then fio will open this local (to the server) job file instead of being passed one from the client.

If you have many servers (example: 100 VMs/containers), you can input a pathname of a file containing host IPs/names as the parameter value for the *--client* option. For example, here is an example `host.list` file containing 2 hostnames:

```
host1.your.dns.domain
host2.your.dns.domain
```

The fio command would then be:

```
fio --client=host.list <job file(s)>
```

In this mode, you cannot input server-specific parameters or job files – all servers receive the same job file.

In order to let `fio --client` runs use a shared filesystem from multiple hosts, `fio --client` now prepends the IP address of the server to the filename. For example, if fio is using the directory `/mnt/nfs/fio` and is writing filename `fileio.tmp`, with a *--client hostfile* containing two hostnames `h1` and `h2` with IP addresses 192.168.10.120 and 192.168.10.121, then fio will create two files:

```
/mnt/nfs/fio/192.168.10.120.fileio.tmp
/mnt/nfs/fio/192.168.10.121.fileio.tmp
```

This behavior can be disabled by the *unique_filename* option.

Terse output in client/server mode will differ slightly from what is produced when fio is run in stand-alone mode. See the terse output section for details.

Also, if one fio invocation runs workloads on multiple servers, fio will provide at the end an aggregate summary report for all workloads. This aggregate summary report assumes that options affecting reporting like *unified_rw_reporting* and *percentile_list* are identical across all the jobs summarized. Having different values for these options is an unsupported configuration.

CHAPTER

# TWO

# EXAMPLES

Some job file examples.

## 2.1 Poisson request flow

Download poisson-rate-submission.fio

```
[poisson-rate-submit]
size=128m
rw=randread
ioengine=libaio
iodepth=32
direct=1
# by setting the submit mode to offload, we can guarantee a fixed rate of
# submission regardless of what the device completion rate is.
io_submit_mode=offload
rate_iops=50
# Real world random request flow follows Poisson process. To give better
# insight on latency distribution, we simulate request flow under Poisson
# process.
rate_process=poisson
```

## 2.2 Latency profile

Download latency-profile.fio

```
# Test job that demonstrates how to use the latency target
# profiling. Fio will find the queue depth between 1..128
# that fits within the latency constraints of this 4k random
# read workload.

[global]
bs=4k
rw=randread
random_generator=lfsr
direct=1
ioengine=libaio
iodepth=128
# Set max acceptable latency to 500msec
latency_target=500000
```

(continues on next page)

(continued from previous page)

```
# profile over a 5s window
latency_window=5000000
# 99.9% of IOs must be below the target
latency_percentile=99.9

[device]
filename=/dev/sda
```

## 2.3 Read 4 files with aio at different depths

Download aio-read.fio

```
; Read 4 files with aio at different depths
[global]
ioengine=libaio
buffered=0
rw=randread
bs=128k
size=512m
directory=/data1

[file1]
iodepth=4

[file2]
iodepth=32

[file3]
iodepth=8

[file4]
iodepth=16
```

## 2.4 Read backwards in a file

Download backwards-read.fio

```
# Demonstrates how to read backwards in a file.

[backwards-read]
bs=4k
# seek -8k back for every IO
rw=read:-8k
filename=128m
size=128m
```

## 2.5 Basic verification

Download basic-verify.fio

```
# The most basic form of data verification. Write the device randomly
# in 4K chunks, then read it back and verify the contents.
[write-and-verify]
rw=randwrite
bs=4k
direct=1
ioengine=libaio
iodepth=16
verify=crc32c
# Use /dev/XXX. For running this on a file instead, remove the filename
# option and add a size=32G (or whatever file size you want) instead.
filename=/dev/XXX
```

## 2.6 Fixed rate submission

Download fixed-rate-submission.fio

```
[fixed-rate-submit]
size=128m
rw=read
ioengine=libaio
iodepth=32
direct=1
# by setting the submit mode to offload, we can guarantee a fixed rate of
# submission regardless of what the device completion rate is.
io_submit_mode=offload
rate_iops=1000
```

## 2.7 Butterfly seek pattern

Download butterfly.fio

```
# Perform a butterfly/funnel seek pattern. This won't always alternate ends on
# every I/O but it will get close.

[global]
filename=/tmp/testfile
bs=4k
direct=1

[forward]
rw=read
flow=2
# Uncomment the size= and offset= lines to prevent each direction going past
# the middle of the file
#size=50%
```

(continues on next page)

(continued from previous page)

```
[backward]
rw=read:-8k
flow=2
#offset=50%
```

CHAPTER

THREE

# TODO

## 3.1 GFIO TODO

In no particular order:

- Ability to save job files. Probably in an extended gfio format, so we can include options/settings outside of a fio job file.
- End view improvements:
  - Cleanup the layout
  - Add ability to save the results
  - Add ability to load end-results as well
  - Add ability to request graphs of whatever graphing options the fio job included.
  - Add ability to graph completion latencies, percentiles, etc.
- Add ability to edit job options:
  - We need an options view after sending a job, that allows us to visually see what was parsed, make changes, resubmit.
  - Job options are already converted across the network and are available in gfio_client->o for view/edit. We'll need a FIO_NET_CMD_UPDATE_OPTIONS command to send them back, and backend support for updating an existing set of options.
- Add support for printing end results, graphs, etc.
- Improve the auto-start backend functionality, it's quite buggy.
- Ensure that it works on OSX and Windows. We'll need a bit of porting work there.
- Persistent store of prefences set. This will need a per-OS bit as well, using gfonf on Linux, registry on Windows, ?? on OSX.
- Ensure that local errors go to our log, instead of being displayed on the console.
- Ensure that the whole connect/send/start button logic is sane. Right now it works when you perform the right sequence, but if you connect and disconnect, things can get confused. We'll need to improve how we store and send job files. Right now they are in ge->job_files[] and are always emptied on send. Keep them around?
- Commit rate display is not enabled.
- Group status reporting is not enabled.
- Split gfio.c a bit. Add gfio/ sub directory, and split it into files based on functionality. It's already ~3000 lines long.

- Attempt to ensure that we work with gtk 2.10 and newer. Right now the required version is ~2.18 (not quite known).

## 3.2 Server TODO

- Collate ETA output from multiple connections into 1
- If group_reporting is set, collate final output from multiple connections

## 3.3 Steady State TODO

Known issues/TODO (for steady-state)

- Replace the test script with a better one - Add test cases for the new check_interval option - Parse debug=steadystate output to check calculations
- Instead of calculating *intervals* every time, calculate it once and stash it somewhere
- Add the time unit to the ss_dur and check_interval variable names to reduce possible confusion
- Better documentation for output
- Report read, write, trim IOPS/BW separately
- Semantics for the ring buffer ss->head are confusing. ss->head points to the beginning of the buffer up through the point where the buffer is filled for the first time. afterwards, when a new element is added, ss->head is advanced to point to the second element in the buffer. if steady state is attained upon adding a new element, ss->head is not advanced so it actually does point to the head of the buffer.

CHAPTER

FOUR

# MORAL LICENSE

As specified by the COPYING file, fio is free software published under version 2 of the GPL license. That covers the copying part of the license. When using fio, you are encouraged to uphold the following moral obligations:

- If you publish results that are done using fio, it should be clearly stated that fio was used. The specific version should also be listed.
- If you develop features or bug fixes for fio, they should be sent upstream for inclusion into the main repository. This isn't specific to fio, that is a general rule for any open source project. It's just the Right Thing to do. Plus it means that you don't have to maintain the feature or change internally. In the long run, this is saving you a lot of time.

I would consider the above to fall under "common courtesy", but since people tend to have differing opinions of that, it doesn't hurt to spell out my expectations clearly.

CHAPTER

FIVE

# LICENSE

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.,
 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it.  (Some other Free Software Foundation software is covered by
the GNU Lesser General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
in new free programs; and that you know you can do these things.

  To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have.  You must make sure that they, too, receive or can get the
source code.  And you must show them these terms so they know their
rights.

  We protect your rights with two steps: (1) copyright the software, and
(2) offer you this license which gives you legal permission to copy,
```

(continues on next page)

(continued from previous page)

```
distribute and/or modify the software.

  Also, for each author's protection and ours, we want to make certain
that everyone understands that there is no warranty for this free
software.  If the software is modified by someone else and passed on, we
want its recipients to know that what they have is not the original, so
that any problems introduced by others will not reflect on the original
authors' reputations.

  Finally, any free program is threatened constantly by software
patents.  We wish to avoid the danger that redistributors of a free
program will individually obtain patent licenses, in effect making the
program proprietary.  To prevent this, we have made it clear that any
patent must be licensed for everyone's free use or not licensed at all.

  The precise terms and conditions for copying, distribution and
modification follow.

                    GNU GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License applies to any program or other work which contains
a notice placed by the copyright holder saying it may be distributed
under the terms of this General Public License.  The "Program", below,
refers to any such program or work, and a "work based on the Program"
means either the Program or any derivative work under copyright law:
that is to say, a work containing the Program or a portion of it,
either verbatim or with modifications and/or translated into another
language.  (Hereinafter, translation is included without limitation in
the term "modification".)  Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope.  The act of
running the Program is not restricted, and the output from the Program
is covered only if its contents constitute a work based on the
Program (independent of having been made by running the Program).
Whether that is true depends on what the Program does.

  1. You may copy and distribute verbatim copies of the Program's
source code as you receive it, in any medium, provided that you
conspicuously and appropriately publish on each copy an appropriate
copyright notice and disclaimer of warranty; keep intact all the
notices that refer to this License and to the absence of any warranty;
and give any other recipients of the Program a copy of this License
along with the Program.

You may charge a fee for the physical act of transferring a copy, and
you may at your option offer warranty protection in exchange for a fee.

  2. You may modify your copy or copies of the Program or any portion
of it, thus forming a work based on the Program, and copy and
distribute such modifications or work under the terms of Section 1
```

(continues on next page)

(continued from previous page)

```
above, provided that you also meet all of these conditions:

    a) You must cause the modified files to carry prominent notices
    stating that you changed the files and the date of any change.

    b) You must cause any work that you distribute or publish, that in
    whole or in part contains or is derived from the Program or any
    part thereof, to be licensed as a whole at no charge to all third
    parties under the terms of this License.

    c) If the modified program normally reads commands interactively
    when run, you must cause it, when started running for such
    interactive use in the most ordinary way, to print or display an
    announcement including an appropriate copyright notice and a
    notice that there is no warranty (or else, saying that you provide
    a warranty) and that users may redistribute the program under
    these conditions, and telling the user how to view a copy of this
    License.  (Exception: if the Program itself is interactive but
    does not normally print such an announcement, your work based on
    the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

    a) Accompany it with the complete corresponding machine-readable
    source code, which must be distributed under the terms of Sections
    1 and 2 above on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three
    years, to give any third party, for a charge no more than your
    cost of physically performing source distribution, a complete
```

(continues on next page)

(continued from previous page)

```
    machine-readable copy of the corresponding source code, to be
    distributed under the terms of Sections 1 and 2 above on a medium
    customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer
    to distribute corresponding source code.  (This alternative is
    allowed only for noncommercial distribution and only if you
    received the program in object code or executable form with such
    an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for
making modifications to it.  For an executable work, complete source
code means all the source code for all modules it contains, plus any
associated interface definition files, plus the scripts used to
control compilation and installation of the executable.  However, as a
special exception, the source code distributed need not include
anything that is normally distributed (in either source or binary
form) with the major components (compiler, kernel, and so on) of the
operating system on which the executable runs, unless that component
itself accompanies the executable.

If distribution of executable or object code is made by offering
access to copy from a designated place, then offering equivalent
access to copy the source code from the same place counts as
distribution of the source code, even though third parties are not
compelled to copy the source along with the object code.

  4. You may not copy, modify, sublicense, or distribute the Program
except as expressly provided under this License.  Any attempt
otherwise to copy, modify, sublicense or distribute the Program is
void, and will automatically terminate your rights under this License.
However, parties who have received copies, or rights, from you under
this License will not have their licenses terminated so long as such
parties remain in full compliance.

  5. You are not required to accept this License, since you have not
signed it.  However, nothing else grants you permission to modify or
distribute the Program or its derivative works.  These actions are
prohibited by law if you do not accept this License.  Therefore, by
modifying or distributing the Program (or any work based on the
Program), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Program or works based on it.

  6. Each time you redistribute the Program (or any work based on the
Program), the recipient automatically receives a license from the
original licensor to copy, distribute or modify the Program subject to
these terms and conditions.  You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties to
this License.
```

(continues on next page)

(continued from previous page)

```
  7. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Program at all.  For example, if a patent
license would not permit royalty-free redistribution of the Program by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under
any particular circumstance, the balance of the section is intended to
apply and the section as a whole is intended to apply in other
circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system, which is
implemented by public license practices.  Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

  8. If the distribution and/or use of the Program is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Program under this License
may add an explicit geographical distribution limitation excluding
those countries, so that distribution is permitted only in or among
countries not thus excluded.  In such case, this License incorporates
the limitation as if written in the body of this License.

  9. The Free Software Foundation may publish revised and/or new versions
of the General Public License from time to time.  Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

Each version is given a distinguishing version number.  If the Program
specifies a version number of this License which applies to it and "any
later version", you have the option of following the terms and conditions
either of that version or of any later version published by the Free
Software Foundation.  If the Program does not specify a version number of
this License, you may choose any version ever published by the Free Software
Foundation.
```

(continues on next page)

(continued from previous page)

```
  10. If you wish to incorporate parts of the Program into other free
programs whose distribution conditions are different, write to the author
to ask for permission.  For software which is copyrighted by the Free
Software Foundation, write to the Free Software Foundation; we sometimes
make exceptions for this.  Our decision will be guided by the two goals
of preserving the free status of all derivatives of our free software and
of promoting the sharing and reuse of software generally.

                            NO WARRANTY

  11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY
FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW.  EXCEPT WHEN
OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES
PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE RISK AS
TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU.  SHOULD THE
PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
REPAIR OR CORRECTION.

  12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR
REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING
OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED
TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.

                     END OF TERMS AND CONDITIONS

            How to Apply These Terms to Your New Programs

  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.
```

(continues on next page)

(continued from previous page)

```
    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this
when it starts in an interactive mode:

    Gnomovision version 69, Copyright (C) year name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, the commands you use may
be called something other than `show w' and `show c'; they could even be
mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  `Gnomovision' (which makes passes at compilers) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice

This General Public License does not permit incorporating your program into
proprietary programs.  If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with the
library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.
```

CHAPTER

SIX

# INDICES AND TABLES

- genindex
- search

# INDEX

## Symbols


--alloc-size
    command line option, 10
--append-terse
    command line option, 9
--aux-path
    command line option, 11
--bandwidth-log
    command line option, 9
--client
    command line option, 10
--cmdhelp
    command line option, 9
--cpuclock-test
    command line option, 9
--crctest
    command line option, 9
--daemonize
    command line option, 10
--debug
    command line option, 7
--enghelp
    command line option, 9
--eta
    command line option, 9
--eta-interval
    command line option, 9
--eta-newline
    command line option, 10
--help
    command line option, 9
--idle-prof
    command line option, 10
--inflate-log
    command line option, 11
--max-jobs
    command line option, 10
--merge-blktrace-only
    command line option, 8
--minimal
    command line option, 9
--output
    command line option, 9
--output-format
    command line option, 9
--parse-only
    command line option, 8
--readonly
    command line option, 9
--remote-config
    command line option, 10
--section
    command line option, 10
--server
    command line option, 10
--showcmd
    command line option, 9
--status-interval
    command line option, 10
--terse-version
    command line option, 9
--trigger
    command line option, 11
--trigger-file
    command line option, 11
--trigger-remote
    command line option, 11
--trigger-timeout
    command line option, 11
--version
    command line option, 9
--warnings-fatal
    command line option, 10


## A


allow_file_create
    command line option, 20
allow_mounted_write
    command line option, 20
allrandrepeat
    command line option, 24
apptag
    command line option, 42
apptag_mask

command line option, 42
arguments
command line option, 48
atomic
command line option, 41

## B

ba
command line option, 30
bindname
command line option, 45
block_error_percentiles
command line option, 64
blockalign
command line option, 30
blocksize
command line option, 29
blocksize_range
command line option, 30
blocksize_unaligned
command line option, 30
bs
command line option, 29
bs_is_seq_rand
command line option, 30
bs_unaligned
command line option, 30
bsrange
command line option, 30
bssplit
command line option, 30
buffer_compress_chunk
command line option, 31
buffer_compress_percentage
command line option, 31
buffer_pattern
command line option, 31
buffered
command line option, 22
busy_poll
command line option, 45
bwavgtime
command line option, 65

## C

cgroup
command line option, 57
cgroup_nodelete
command line option, 57
cgroup_weight
command line option, 57
chunk_size
command line option, 45
clat_percentiles
command line option, 65
clientname
command line option, 44
clocksource
command line option, 17
clustername
command line option, 44
cmd_type
command line option, 40
cmdprio
command line option, 39
cmdprio_bssplit
command line option, 39
cmdprio_class
command line option, 39
cmdprio_hint
command line option, 39
cmdprio_percentage
command line option, 38
command line option
--alloc-size, 10
--append-terse, 9
--aux-path, 11
--bandwidth-log, 9
--client, 10
--cmdhelp, 9
--cpuclock-test, 9
--crctest, 9
--daemonize, 10
--debug, 7
--enghelp, 9
--eta, 9
--eta-interval, 9
--eta-newline, 10
--help, 9
--idle-prof, 10
--inflate-log, 11
--max-jobs, 10
--merge-blktrace-only, 8
--minimal, 9
--output, 9
--output-format, 9
--parse-only, 8
--readonly, 9
--remote-config, 10
--section, 10
--server, 10
--showcmd, 9
--status-interval, 10
--terse-version, 9
--trigger, 11
--trigger-file, 11
--trigger-remote, 11
--trigger-timeout, 11

--version, 9
--warnings-fatal, 10
allow_file_create, 20
allow_mounted_write, 20
allrandrepeat, 24
apptag, 42
apptag_mask, 42
arguments, 48
atomic, 41
ba, 30
bindname, 45
block_error_percentiles, 64
blockalign, 30
blocksize, 29
blocksize_range, 30
blocksize_unaligned, 30
bs, 29
bs_is_seq_rand, 30
bs_unaligned, 30
bsrange, 30
bssplit, 30
buffer_compress_chunk, 31
buffer_compress_percentage, 31
buffer_pattern, 31
buffered, 22
busy_poll, 45
bwavgtime, 65
cgroup, 57
cgroup_nodelete, 57
cgroup_weight, 57
chunk_size, 45
clat_percentiles, 65
clientname, 44
clocksource, 17
clustername, 44
cmd_type, 40
cmdprio, 39
cmdprio_bssplit, 39
cmdprio_class, 39
cmdprio_hint, 39
cmdprio_percentage, 38
conf, 45
cont, 45
continue_on_error, 65
cpuchunks, 43
cpuload, 43
cpumask, 56
cpumode, 43
cpus_allowed, 56
cpus_allowed_policy, 56
create_fsync, 20
create_on_open, 20
create_only, 20
create_serialize, 20
cuda_io, 48
dataplacement, 41
dedupe_global, 32
dedupe_mode, 32
dedupe_percentage, 32
dedupe_working_set_percentage, 32
description, 16
direct, 22
directory, 18
disable_bw, 65
disable_bw_measurement, 65
disable_clat, 65
disable_lat, 65
disable_slat, 65
disk_util, 65
do_verify, 58
donorname, 44
dp_scheme, 42
end_fsync, 27
error_dump, 66
exec_postrun, 57
exec_prerun, 57
exit_on_io_done, 43
exit_what, 57
exitall, 57
exitall_on_error, 65
experimental_verify, 60
fadvise_hint, 25
fallocate, 24
fdatasync, 26
fdp, 41
fdp_pli, 42
fdp_pli_select, 41
file_append, 34
file_service_type, 19
filename, 18
filename_format, 18
filesize, 34
filetype, 19
fill_device, 34
fill_fs, 34
fixedbufs, 39
flow, 57
flow_id, 57
flow_sleep, 57
force_async, 39
fsync, 26
fsync_on_close, 27
gid, 58
gpu_dev_ids, 48
grace_time, 49
group_reporting, 62
gtod_cpu, 17
gtod_reduce, 17

hdfsdirectory, 45
hipri, 40
hipri_percentage, 40
hostname, 43
http_host, 47
http_mode, 47
http_object_mode, 48
http_pass, 47
http_s3_key, 47
http_s3_keyid, 47
http_s3_region, 47
http_s3_security_token, 47
http_s3_sse_customer_algorithm, 47
http_s3_sse_customer_key, 47
http_s3_storage_class, 47
http_swift_auth_token, 48
http_user, 47
http_verbose, 48
https, 47
hugepage-size, 33
ignore_error, 66
ignore_zone_limits, 22
inplace, 44
interface, 43
invalidate, 32
io_limit, 34
io_size, 34
io_submit_mode, 52
iodepth, 51
iodepth_batch, 51
iodepth_batch_complete, 51
iodepth_batch_complete_max, 51
iodepth_batch_complete_min, 51
iodepth_batch_submit, 51
iodepth_low, 52
ioengine, 34
iomem, 32
iomem_align, 33
iopsavgtime, 65
ioscheduler, 20
job_max_open_zones, 22
job_start_clock_id, 17
kb_base, 16
lat_percentiles, 65
latency_percentile, 54
latency_run, 54
latency_target, 53
latency_window, 53
libaio_vectored, 41
libblkio_driver, 50
libblkio_force_enable_completion_eventfd, 51
libblkio_num_entries, 50
libblkio_path, 50
libblkio_pre_connect_props, 50
libblkio_pre_start_props, 50
libblkio_queue_size, 50
libblkio_vectored, 50
libblkio_wait_mode, 51
libblkio_write_zeroes_on_trim, 50
listen, 44
lockfile, 19
lockmem, 33
log_alternate_epoch, 64
log_alternate_epoch_clock_id, 64
log_avg_msec, 63
log_compression, 64
log_compression_cpus, 64
log_entries, 63
log_hist_coarseness, 63
log_hist_msec, 63
log_issue_time, 64
log_max_value, 63
log_offset, 64
log_prio, 64
log_store_compressed, 64
log_unix_epoch, 64
log_window_value, 63
loops, 16
max_latency, 54
max_open_zones, 21
md_per_io_size, 42
mem, 32
mem_align, 33
merge_blktrace_file, 54
merge_blktrace_iters, 54
merge_blktrace_scalars, 54
mss, 44
name, 16
namenode, 43
new_group, 62
nfs_url, 48
nice, 55
no_completion_thread, 51
nodelay, 43
nonvectored, 39
norandommap, 28
nowait, 40
nrfiles, 19
num_range, 43
numa_cpu_nodes, 56
numa_mem_policy, 56
number_ios, 26
numjobs, 16
object_class, 45
offset, 26
offset_align, 26
offset_increment, 26

opendir, 19
openfiles, 19
overwrite, 27
per_job_logs, 62
percentage_random, 28
percentile_list, 65
pi_act, 42
pi_chk, 42
pingpong, 44
plid_select, 41
plids, 42
pool, 45
port, 43
pre_read, 20
prio, 55
prioclass, 55
priohint, 56
profile, 66
program, 48
proto, 43
protocol, 43
ramp_time, 17
random_distribution, 27
random_generator, 28
randrepeat, 24
randseed, 24
rate, 53
rate_cycle, 53
rate_ignore_thinktime, 53
rate_iops, 53
rate_iops_min, 53
rate_min, 53
rate_process, 53
rbdname, 44
read_beyond_wp, 21
read_iolog, 54
read_iolog_chunked, 54
readfua, 45
readwrite, 22
recover_zbd_write_error, 22
refill_buffers, 31
registerfiles, 40
replay_align, 55
replay_no_stall, 54
replay_redirect, 55
replay_scale, 55
replay_skip, 55
replay_time_scale, 55
runtime, 17
rw, 22
rw_sequencer, 23
rwmixread, 27
rwmixwrite, 27
scramble_buffers, 31
serialize_overlap, 52
sg_write_mode, 46
significant_figures, 65
size, 33
skip_bad, 45
slat_percentiles, 65
softrandommap, 28
spr_num_regions, 29
spr_op, 29
sprandom, 29
sqthread_poll, 40
sqthread_poll_cpu, 40
ss, 61
ss_dur, 62
ss_interval, 62
ss_ramp, 62
startdelay, 17
stat_type, 45
stats, 62
std_redirect, 49
steadystate, 61
steadystate_check_interval, 62
steadystate_duration, 62
steadystate_ramp_time, 62
stonewall, 57
stream_id, 47
sync, 32
sync_file_range, 26
thinkcycles, 52
thinktime, 52
thinktime_blocks, 52
thinktime_blocks_type, 53
thinktime_iotime, 53
thinktime_spin, 52
thread, 55
time_based, 17
touch_objects, 45
trim_backlog, 61
trim_backlog_batch, 61
trim_percentage, 61
trim_verify_zero, 61
ttl, 43
uid, 57
uncached, 41
unified_rw_reporting, 24
unique_filename, 19
unit_base, 16
unlink, 20
unlink_each_loop, 21
uri, 48
userspace_reap, 40
verb, 45
verify, 58
verify_async, 60

verify_async_cpus, 60
verify_backlog, 60
verify_backlog_batch, 60
verify_dump, 60
verify_fatal, 60
verify_header_seed, 61
verify_interval, 59
verify_mode, 46
verify_offset, 59
verify_only, 58
verify_pattern, 59
verify_pattern_interval, 60
verify_state_load, 60
verify_state_save, 60
verify_write_sequence, 61
wait_for, 55
wait_for_previous, 57
window_size, 44
write_barrier, 26
write_bw_log, 62
write_hint, 25
write_hist_log, 63
write_iolog, 54
write_iops_log, 63
write_lat_log, 63
write_mode, 46
writefua, 46
xnvme_admin, 49
xnvme_async, 49
xnvme_dev_nsid, 49
xnvme_dev_subnqn, 50
xnvme_iovec, 50
xnvme_mem, 50
xnvme_sync, 49
zero_buffers, 31
zone_reset_frequency, 22
zone_reset_threshold, 22
zonecapacity, 21
zonemode, 21
zonerange, 21
zonesize, 21
zoneskip, 21
conf
command line option, 45
cont
command line option, 45
continue_on_error
command line option, 65
cpuchunks
command line option, 43
cpuload
command line option, 43
cpumask
command line option, 56
cpumode
command line option, 43
cpus_allowed
command line option, 56
cpus_allowed_policy
command line option, 56
create_fsync
command line option, 20
create_on_open
command line option, 20
create_only
command line option, 20
create_serialize
command line option, 20
cuda_io
command line option, 48

## D

dataplacement
command line option, 41
dedupe_global
command line option, 32
dedupe_mode
command line option, 32
dedupe_percentage
command line option, 32
dedupe_working_set_percentage
command line option, 32
description
command line option, 16
direct
command line option, 22
directory
command line option, 18
disable_bw
command line option, 65
disable_bw_measurement
command line option, 65
disable_clat
command line option, 65
disable_lat
command line option, 65
disable_slat
command line option, 65
disk_util
command line option, 65
do_verify
command line option, 58
donorname
command line option, 44
dp_scheme
command line option, 42

## E

end_fsync
    command line option, 27
error_dump
    command line option, 66
exec_postrun
    command line option, 57
exec_prerun
    command line option, 57
exit_on_io_done
    command line option, 43
exit_what
    command line option, 57
exitall
    command line option, 57
exitall_on_error
    command line option, 65
experimental_verify
    command line option, 60

## F

fadvise_hint
    command line option, 25
fallocate
    command line option, 24
fdatasync
    command line option, 26
fdp
    command line option, 41
fdp_pli
    command line option, 42
fdp_pli_select
    command line option, 41
file_append
    command line option, 34
file_service_type
    command line option, 19
filename
    command line option, 18
filename_format
    command line option, 18
filesize
    command line option, 34
filetype
    command line option, 19
fill_device
    command line option, 34
fill_fs
    command line option, 34
fixedbufs
    command line option, 39
flow
    command line option, 57
flow_id
    command line option, 57
flow_sleep
    command line option, 57
force_async
    command line option, 39
fsync
    command line option, 26
fsync_on_close
    command line option, 27

## G

gid
    command line option, 58
gpu_dev_ids
    command line option, 48
grace_time
    command line option, 49
group_reporting
    command line option, 62
gtod_cpu
    command line option, 17
gtod_reduce
    command line option, 17

## H

hdfsdirectory
    command line option, 45
hipri
    command line option, 40
hipri_percentage
    command line option, 40
hostname
    command line option, 43
http_host
    command line option, 47
http_mode
    command line option, 47
http_object_mode
    command line option, 48
http_pass
    command line option, 47
http_s3_key
    command line option, 47
http_s3_keyid
    command line option, 47
http_s3_region
    command line option, 47
http_s3_security_token
    command line option, 47
http_s3_sse_customer_algorithm
    command line option, 47
http_s3_sse_customer_key
    command line option, 47
http_s3_storage_class

command line option, 47
http_swift_auth_token
command line option, 48
http_user
command line option, 47
http_verbose
command line option, 48
https
command line option, 47
hugepage-size
command line option, 33

## I

ignore_error
command line option, 66
ignore_zone_limits
command line option, 22
inplace
command line option, 44
interface
command line option, 43
invalidate
command line option, 32
io_limit
command line option, 34
io_size
command line option, 34
io_submit_mode
command line option, 52
iodepth
command line option, 51
iodepth_batch
command line option, 51
iodepth_batch_complete
command line option, 51
iodepth_batch_complete_max
command line option, 51
iodepth_batch_complete_min
command line option, 51
iodepth_batch_submit
command line option, 51
iodepth_low
command line option, 52
ioengine
command line option, 34
iomem
command line option, 32
iomem_align
command line option, 33
iopsavgtime
command line option, 65
ioscheduler
command line option, 20

## J

job_max_open_zones
command line option, 22
job_start_clock_id
command line option, 17

## K

kb_base
command line option, 16

## L

lat_percentiles
command line option, 65
latency_percentile
command line option, 54
latency_run
command line option, 54
latency_target
command line option, 53
latency_window
command line option, 53
libaio_vectored
command line option, 41
libblkio_driver
command line option, 50
libblkio_force_enable_completion_eventfd
command line option, 51
libblkio_num_entries
command line option, 50
libblkio_path
command line option, 50
libblkio_pre_connect_props
command line option, 50
libblkio_pre_start_props
command line option, 50
libblkio_queue_size
command line option, 50
libblkio_vectored
command line option, 50
libblkio_wait_mode
command line option, 51
libblkio_write_zeroes_on_trim
command line option, 50
listen
command line option, 44
lockfile
command line option, 19
lockmem
command line option, 33
log_alternate_epoch
command line option, 64
log_alternate_epoch_clock_id
command line option, 64

log_avg_msec
    command line option, 63
log_compression
    command line option, 64
log_compression_cpus
    command line option, 64
log_entries
    command line option, 63
log_hist_coarseness
    command line option, 63
log_hist_msec
    command line option, 63
log_issue_time
    command line option, 64
log_max_value
    command line option, 63
log_offset
    command line option, 64
log_prio
    command line option, 64
log_store_compressed
    command line option, 64
log_unix_epoch
    command line option, 64
log_window_value
    command line option, 63
loops
    command line option, 16

## M

max_latency
    command line option, 54
max_open_zones
    command line option, 21
md_per_io_size
    command line option, 42
mem
    command line option, 32
mem_align
    command line option, 33
merge_blktrace_file
    command line option, 54
merge_blktrace_iters
    command line option, 54
merge_blktrace_scalars
    command line option, 54
mss
    command line option, 44

## N

name
    command line option, 16
namenode
    command line option, 43
new_group
    command line option, 62
nfs_url
    command line option, 48
nice
    command line option, 55
no_completion_thread
    command line option, 51
nodelay
    command line option, 43
nonvectored
    command line option, 39
norandommap
    command line option, 28
nowait
    command line option, 40
nrfiles
    command line option, 19
num_range
    command line option, 43
numa_cpu_nodes
    command line option, 56
numa_mem_policy
    command line option, 56
number_ios
    command line option, 26
numjobs
    command line option, 16

## O

object_class
    command line option, 45
offset
    command line option, 26
offset_align
    command line option, 26
offset_increment
    command line option, 26
opendir
    command line option, 19
openfiles
    command line option, 19
overwrite
    command line option, 27

## P

per_job_logs
    command line option, 62
percentage_random
    command line option, 28
percentile_list
    command line option, 65
pi_act
    command line option, 42

pi_chk
    command line option, 42
pingpong
    command line option, 44
plid_select
    command line option, 41
plids
    command line option, 42
pool
    command line option, 45
port
    command line option, 43
pre_read
    command line option, 20
prio
    command line option, 55
prioclass
    command line option, 55
priohint
    command line option, 56
profile
    command line option, 66
program
    command line option, 48
proto
    command line option, 43
protocol
    command line option, 43

## R

ramp_time
    command line option, 17
random_distribution
    command line option, 27
random_generator
    command line option, 28
randrepeat
    command line option, 24
randseed
    command line option, 24
rate
    command line option, 53
rate_cycle
    command line option, 53
rate_ignore_thinktime
    command line option, 53
rate_iops
    command line option, 53
rate_iops_min
    command line option, 53
rate_min
    command line option, 53
rate_process
    command line option, 53
rbdname
    command line option, 44
read_beyond_wp
    command line option, 21
read_iolog
    command line option, 54
read_iolog_chunked
    command line option, 54
readfua
    command line option, 45
readwrite
    command line option, 22
recover_zbd_write_error
    command line option, 22
refill_buffers
    command line option, 31
registerfiles
    command line option, 40
replay_align
    command line option, 55
replay_no_stall
    command line option, 54
replay_redirect
    command line option, 55
replay_scale
    command line option, 55
replay_skip
    command line option, 55
replay_time_scale
    command line option, 55
runtime
    command line option, 17
rw
    command line option, 22
rw_sequencer
    command line option, 23
rwmixread
    command line option, 27
rwmixwrite
    command line option, 27

## S

scramble_buffers
    command line option, 31
serialize_overlap
    command line option, 52
sg_write_mode
    command line option, 46
significant_figures
    command line option, 65
size
    command line option, 33
skip_bad
    command line option, 45

slat_percentiles
    command line option, 65
softrandommap
    command line option, 28
spr_num_regions
    command line option, 29
spr_op
    command line option, 29
sprandom
    command line option, 29
sqthread_poll
    command line option, 40
sqthread_poll_cpu
    command line option, 40
ss
    command line option, 61
ss_dur
    command line option, 62
ss_interval
    command line option, 62
ss_ramp
    command line option, 62
startdelay
    command line option, 17
stat_type
    command line option, 45
stats
    command line option, 62
std_redirect
    command line option, 49
steadystate
    command line option, 61
steadystate_check_interval
    command line option, 62
steadystate_duration
    command line option, 62
steadystate_ramp_time
    command line option, 62
stonewall
    command line option, 57
stream_id
    command line option, 47
sync
    command line option, 32
sync_file_range
    command line option, 26

## T

thinkcycles
    command line option, 52
thinktime
    command line option, 52
thinktime_blocks
    command line option, 52
thinktime_blocks_type
    command line option, 53
thinktime_iotime
    command line option, 53
thinktime_spin
    command line option, 52
thread
    command line option, 55
time_based
    command line option, 17
touch_objects
    command line option, 45
trim_backlog
    command line option, 61
trim_backlog_batch
    command line option, 61
trim_percentage
    command line option, 61
trim_verify_zero
    command line option, 61
ttl
    command line option, 43

## U

uid
    command line option, 57
uncached
    command line option, 41
unified_rw_reporting
    command line option, 24
unique_filename
    command line option, 19
unit_base
    command line option, 16
unlink
    command line option, 20
unlink_each_loop
    command line option, 21
uri
    command line option, 48
userspace_reap
    command line option, 40

## V

verb
    command line option, 45
verify
    command line option, 58
verify_async
    command line option, 60
verify_async_cpus
    command line option, 60
verify_backlog
    command line option, 60

verify_backlog_batch
    command line option, 60
verify_dump
    command line option, 60
verify_fatal
    command line option, 60
verify_header_seed
    command line option, 61
verify_interval
    command line option, 59
verify_mode
    command line option, 46
verify_offset
    command line option, 59
verify_only
    command line option, 58
verify_pattern
    command line option, 59
verify_pattern_interval
    command line option, 60
verify_state_load
    command line option, 60
verify_state_save
    command line option, 60
verify_write_sequence
    command line option, 61

## W

wait_for
    command line option, 55
wait_for_previous
    command line option, 57
window_size
    command line option, 44
write_barrier
    command line option, 26
write_bw_log
    command line option, 62
write_hint
    command line option, 25
write_hist_log
    command line option, 63
write_iolog
    command line option, 54
write_iops_log
    command line option, 63
write_lat_log
    command line option, 63
write_mode
    command line option, 46
writefua
    command line option, 46

## X

xnvme_admin
    command line option, 49
xnvme_async
    command line option, 49
xnvme_dev_nsid
    command line option, 49
xnvme_dev_subnqn
    command line option, 50
xnvme_iovec
    command line option, 50
xnvme_mem
    command line option, 50
xnvme_sync
    command line option, 49

## Z

zero_buffers
    command line option, 31
zone_reset_frequency
    command line option, 22
zone_reset_threshold
    command line option, 22
zonecapacity
    command line option, 21
zonemode
    command line option, 21
zonerange
    command line option, 21
zonesize
    command line option, 21
zoneskip
    command line option, 21