blktrace User Guide

blktrace: Jens Axboe (jens.axboe@oracle.com)
User Guide: Alan D. Brunelle (Alan.Brunelle@hp.com)

18 February 2007

1 Introduction

blktrace is a block layer IO tracing mechanism which provides detailed information about request queue operations up to user space. There are three major components that are provided:

Kernel patch A patch to the Linux kernel which includes the kernel event logging interfaces, and patches to areas within the block layer to emit event traces. If you run a 2.6.17-rc1 or newer kernel, you don't need to patch blktrace support as it is already included.

blktrace A utility which transfers event traces from the kernel into either long-term on-disk storage, or provides direct formatted output (via blkparse).

blkparse A utility which formats events stored in files, or when run in *live* mode directly outputs data collected by blktrace.

1.1 blktrace Download Area

The blktrace and blkparse utilities and associated kernel patch are provided as part of the following git repository:

git://git.kernel.org/pub/scm/linux/kernel/git/axboe/blktrace.git bt

2 Quick Start Guide

The following sections outline some quick steps towards utilizing blktrace. Some of the specific instructions below may need to be tailored to your environment.

2.1 Retrieving blktrace

As noted above, the kernel patch along with the blktrace and blkparse utilities are stored in a git repository. One simple way to get going would be:

```
% git clone git://git.kernel.org/pub/scm/linux/kernel/git/axboe/blktrace.git bt
% cd bt
% git checkout
```

2.2 Patching and configuring the Linux kernel

A patch for a *specific Linux kernel* is provided in bt/kernel (where *bt* is the name of the directory from the above git sequence). The detailed actual patching instructions for a Linux kernel is outside the scope of this document, but the following may be used as a sample template. Note that you may skip this step, if you kernel is at least 2.6.17-rc1.

As an example, bt/kernel contains blk-trace-2.6.14-rc1-git-G2, download linux-2.6.13.tar.bz2 and patch-2.6.14-rc1.bz2

```
% tar xjf linux-2.6.13.tar.bz2
% mv linux-2.6.13 linux-2.6.14-rc1
% cd linux-2.6.14-rc1
% bunzip2 -c ../patch-2.6.14-rc1.bz2 | patch -p1
```

At this point you may (optionally) remove linux-2.6.13.tar.bz2 and patch-2.6.14-rc1.bz2.

At this point you should configure the Linux kernel for your specific system – again, outside the scope of this document – and then enable *Support for tracing block io actions*. To do this, run

```
% make menuconfig                    or make xconfig, or edit .config, or ...
```

and navigate through *Device Drivers* and *Block devices* and then down to *Support for tracing block io actions* and hit Y.

Install the new kernel (and modules...) and reboot.

2.3 Mounting the debugfs file system

blktrace utilizes files under the debug file system, and thus must have the mount point set up – mounted on the directory /sys/kernel/debug. To do this one may do either of the following:

1. Manually mount after each boot:

```
% mount -t debugfs debugfs /sys/kernel/debug
```

2. Add an entry into /etc/fstab, and have it done automatically at each boot[1]:

```
debug /sys/kernel/debug debugfs default 0 0
```

2.4 Build the tools

To build and install the tools, execute the following sequence (as root):

```
% cd bt
% make && make install
```

2.5 blktrace – live

Now to simply watch what is going on for a specific disk (to stop the trace, hit control-C):

```
% blktrace -d /dev/sda -o - | blkparse -i -
  8,0    3        1     0.000000000   697  G   W 223490 + 8 [kjournald]
  8,0    3        2     0.000001829   697  P   R [kjournald]
  8,0    3        3     0.000002197   697  Q   W 223490 + 8 [kjournald]
  8,0    3        4     0.000005533   697  M   W 223498 + 8 [kjournald]
  8,0    3        5     0.000008607   697  M   W 223506 + 8 [kjournald]
  8,0    3        6     0.000011569   697  M   W 223514 + 8 [kjournald]
  8,0    3        7     0.000014407   697  M   W 223522 + 8 [kjournald]
  8,0    3        8     0.000017367   697  M   W 223530 + 8 [kjournald]
  8,0    3        9     0.000020161   697  M   W 223538 + 8 [kjournald]
  8,0    3       10     0.000024062   697  D   W 223490 + 56 [kjournald]
  8,0    1       11     0.009507758     0  C   W 223490 + 56 [0]
  8,0    1       12     0.009538995   697  G   W 223546 + 8 [kjournald]
  8,0    1       13     0.009540033   697  P   R [kjournald]
  8,0    1       14     0.009540313   697  Q   W 223546 + 8 [kjournald]
  8,0    1       15     0.009542980   697  D   W 223546 + 8 [kjournald]
  8,0    1       16     0.013542170     0  C   W 223546 + 8 [0]
...
^C
...
CPU1 (8,0):
 Reads Queued:           0,        0KiB  Writes Queued:           7,      128KiB
 Read Dispatches:        0,        0KiB  Write Dispatches:        7,      128KiB
 Reads Completed:        0,        0KiB  Writes Completed:       11,      168KiB
 Read Merges:            0               Write Merges:           25
```

[1]Note: after adding the entry to /etc/fstab, you could then mount the directory this time only by doing: % mount debug

```
 IO unplugs:             0               Timer unplugs:          0
...
CPU3 (8,0):
 Reads Queued:           0,        0KiB  Writes Queued:          1,       28KiB
 Read Dispatches:        0,        0KiB  Write Dispatches:       1,       28KiB
 Reads Completed:        0,        0KiB  Writes Completed:       0,        0KiB
 Read Merges:            0               Write Merges:           6
 IO unplugs:             0               Timer unplugs:          0

Total (8,0):
 Reads Queued:           0,        0KiB  Writes Queued:         11,      168KiB
 Read Dispatches:        0,        0KiB  Write Dispatches:      11,      168KiB
 Reads Completed:        0,        0KiB  Writes Completed:      11,      168KiB
 Read Merges:            0               Write Merges:          31
 IO unplugs:             0               Timer unplugs:          3

Events (8,0): 89 entries, 0 skips
```

A *btrace* script is included in the distribution to ease live tracing of devices. The above could also be accomplished by issuing:

```
% btrace /dev/sda
```

By default, *btrace* runs the trace in quiet mode so it will not include statistics when you break the run. Add the *-S* option to get that dumped as well.

2.6 blktrace – SCSI commands

The previous section showed typical file system io actions, but blktrace can also show SCSI commands going in and out of the queue as submitted by applications using the SCSI Generic (*sg*) interface.

```
% btrace /dev/cdrom
[...]
  3,0    0       25     0.004884107 13528  G   R 0 + 0 [inquiry]
  3,0    0       26     0.004890361 13528  I   R 56 (12 00 00 00 38 ..) [inquiry]
  3,0    0       27     0.004891223 13528  P   R [inquiry]
  3,0    0       28     0.004893250 13528  D   R 56 (12 00 00 00 38 ..) [inquiry]
  3,0    0       29     0.005344910     0  C   R (12 00 00 00 38 ..) [0]
```

Here we see a program issuing an INQUIRY command to the CDROM device. The program requested a read of 56 bytes of data, the CDB is included in parenthesis after the data length. The completion event shows shows that the command completed successfully. Tracing SCSI commands can be very useful for debugging problems with programs talking directly to the device. An example of that would be *cdrecord* burning.

2.7 blktrace – post-processing

Another way to run blktrace is to have blktrace save data away for later formatting by blkparse. This would be useful if you want to get measurements while running specific loads.

To do this, one would specify the device (or devices) to be watched. Then go run you test cases. Stop the trace, and at your leisure utilize blkparse to see the results.

In this example, devices /dev/sdaa, /dev/sdc and /dev/sdo are used in an LVM volume called adb3/vol.

```
% blktrace /dev/sdaa /dev/sdc /dev/sdo &
[1] 9713
%
% mkfs -t ext3 /dev/adb3/vol
mke2fs 1.35 (28-Feb-2004)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
16793600 inodes, 33555456 blocks
1677772 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=4294967296
1025 block groups
32768 blocks per group, 32768 fragments per group
16384 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
4096000, 7962624, 11239424, 20480000, 23887872

Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 27 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
%
% kill -15 9713
```

Then you could process the events later:

```
%
% blkparse sdaa sdc sdo > events
% less events
  8,32   1        1     0.000000000  9728  G   R 384 + 32 [mkfs.ext3]
  8,32   1        2     0.000001959  9728  P   R [mkfs.ext3]
```

```
  8,32   1        3     0.000002446  9728  Q   R 384 + 32 [mkfs.ext3]
  8,32   1        4     0.000005110  9728  D   R 384 + 32 [mkfs.ext3]
  8,32   3        5     0.000200570     0  C   R 384 + 32 [0]
  8,224  3        1     0.021658989  9728  G   R 384 + 32 [mkfs.ext3]
...
 65,160  3   163392    41.117070504     0  C   W 87469088 + 1376 [0]
  8,32   3   163374    41.122683668     0  C   W 88168160 + 1376 [0]
 65,160  3   163393    41.129952433     0  C   W 87905984 + 1376 [0]
 65,160  3   163394    41.130049431     0  D   W 89129344 + 1376 [swapper]
 65,160  3   163395    41.130067135     0  D   W 89216704 + 1376 [swapper]
 65,160  3   163396    41.130083785     0  D   W 89304096 + 1376 [swapper]
 65,160  3   163397    41.130099455     0  D   W 89391488 + 1376 [swapper]
 65,160  3   163398    41.130114732     0  D   W 89478848 + 1376 [swapper]
 65,160  3   163399    41.130128885     0  D   W 89481536 + 64 [swapper]
  8,32   3   163375    41.134758196     0  C   W 86333152 + 1376 [0]
 65,160  3   163400    41.142229726     0  C   W 89129344 + 1376 [0]
 65,160  3   163401    41.144952314     0  C   W 89481536 + 64 [0]
  8,32   3   163376    41.147441930     0  C   W 88342912 + 1376 [0]
 65,160  3   163402    41.155869604     0  C   W 89478848 + 1376 [0]
  8,32   3   163377    41.159466082     0  C   W 86245760 + 1376 [0]
 65,160  3   163403    41.166944976     0  C   W 89216704 + 1376 [0]
 65,160  3   163404    41.178968252     0  C   W 89304096 + 1376 [0]
 65,160  3   163405    41.191860173     0  C   W 89391488 + 1376 [0]
...
Events (sdo): 0 entries, 0 skips

CPU0 (65,160):
 Reads Queued:           0,        0KiB  Writes Queued:           9,    5,520KiB
 Read Dispatches:        0,        0KiB  Write Dispatches:        0,        0KiB
 Reads Completed:        0,        0KiB  Writes Completed:        0,        0KiB
 Read Merges:            0              Write Merges:          336
 IO unplugs:             0              Timer unplugs:           0
CPU1 (65,160):
 Reads Queued:       2,411,   38,576KiB  Writes Queued:         769,  425,408KiB
 Read Dispatches:    2,407,   38,512KiB  Write Dispatches:      118,   61,680KiB
 Reads Completed:        0,        0KiB  Writes Completed:        0,        0KiB
 Read Merges:            0              Write Merges:       25,819
 IO unplugs:             0              Timer unplugs:           4
CPU2 (65,160):
 Reads Queued:           2,       32KiB  Writes Queued:          18,   10,528KiB
 Read Dispatches:        2,       32KiB  Write Dispatches:        3,    1,344KiB
 Reads Completed:        0,        0KiB  Writes Completed:        0,        0KiB
 Read Merges:            0              Write Merges:          640
 IO unplugs:             0              Timer unplugs:           0
CPU3 (65,160):
 Reads Queued:      20,572,  329,152KiB  Writes Queued:         594,  279,712KiB
```

```
 Read Dispatches:   20,576,  329,216KiB  Write Dispatches:    1,474,  740,720KiB
 Reads Completed:   22,985,  367,760KiB  Writes Completed:    1,390,  721,168KiB
 Read Merges:            0               Write Merges:       16,888
 IO unplugs:             0               Timer unplugs:           0

Total (65,160):
 Reads Queued:      22,985,  367,760KiB  Writes Queued:       1,390,  721,168KiB
 Read Dispatches:   22,985,  367,760KiB  Write Dispatches:    1,595,  803,744KiB
 Reads Completed:   22,985,  367,760KiB  Writes Completed:    1,390,  721,168KiB
 Read Merges:            0               Write Merges:       43,683
 IO unplugs:             0               Timer unplugs:           4
...
```

3 blktrace User Guide

The *blktrace* utility extracts event traces from the kernel (via the relaying through the debug file system). Some background details concerning the run-time behaviour of blktrace will help to understand some of the more arcane command line options:

- blktrace receives data from the kernel in buffers passed up through the debug file system (relay). Each device being traced has a file created in the mounted directory for the debugfs, which defaults to */sys/kernel/debug* – this can be overridden with the *-r* command line argument.
- blktrace defaults to collecting *all* events that can be traced. To limit the events being captured, you can specify one or more filter masks via the *-a* option.

  Alternatively, one may specify the entire mask utilizing a hexadecimal value that is version-specific. (Requires understanding of the internal representation of the filter mask.)
- As noted above, the events are passed up via a series of buffers stored into debugfs files. The size and number of buffers can be specified via the *-b* and *-n* arguments respectively.
- blktrace stores the extracted data into files stored in the *local* directory. The format of the file names is (by default) *device*.blktrace.*cpu*, where *device* is the base device name (e.g, if we are tracing /dev/sda, the base device name would be *sda*); and *cpu* identifies a CPU for the event stream.

  The *device* portion of the event file name can be changed via the *-o* option.
- blktrace may also be run concurrently with blkparse to produce *live* output – to do this specify *-o* - for blktrace.
- The default behaviour for blktrace is to run forever until explicitly killed by the user (via a control-C, or *kill* utility invocation). There are two ways to modify this:

  1. You may utilize the blktrace utility itself to *kill* a running trace – via the *-k* option.
  2. You can specify a run-time duration for blktrace via the *-w* option – then blktrace will run for the specified number of seconds, and then halt.

3.1 Command line arguments

Short	Long	Description
-A *hex-mask*	–set-mask=*hex-mask*	Set filter mask to *hex-mask*
-a *mask*	–act-mask=*mask*	Add *mask* to current filter (see below for masks)
-b *size*	–buffer-size=*size*	Specifies buffer size for event extraction (scaled by 2^{10})
-d *dev*	–dev=*dev*	Adds *dev* as a device to trace
-k	–kill	Kill on-going trace
-n *num-sub*	–num-sub=*num-sub*	Specifies number of buffers to use
-o *file*	–output=*file*	Prepend *file* to output file name(s)
-r *rel-path*	–relay=*rel-path*	Specifies debugfs mount point
-V	–version	Outputs version
-w *seconds*	–stopwatch=*seconds*	Sets run time to the number of seconds specified
-I *devs file*	–input-devs=*devs file*	Adds devices found in *devs file* to list of devices to trace. (One device per line.)

3.1.1 Filter Masks

The following masks may be passed with the *-a* command line option, multiple filters may be combined via multiple *-a* command line options.

barrier	*barrier* attribute
complete	*completed* by driver
fs	*FS* requests
issue	*issued* to driver
pc	*packet command* events
queue	*queue* operations
read	*read* traces
requeue	*requeue* operations
sync	*synchronous* attribute
write	*write* traces

3.1.2 Request types

blktrace disguingishes between two types of block layer requests, file system and scsi commands. The former are dubbed *fs* requests, the latter *pc* requests. File system requests are normal read/write operations, ie any type of read or write from a specific disk location at a given size. These requests typically originate from a user process, but they may also be initiated by the vm flushing dirty data to disk or the file system syncing a super or journal block to disk. *pc* requests are SCSI commands. blktrace sends the command data block as a payload so that blkparse can decode it.

4 blkparse User Guide

The *blkparse* utility will attempt to combine streams of events for various devices on various CPUs, and produce a formatted output of the event information. As with blktrace, some details concerning blkparse will help in understanding the command line options presented below.

- By default, blkparse expects to run in a post-processing mode – one where the trace events have been saved by a previous run of blktrace, and blkparse is combining event streams and dumping formatted data.

  blkparse *may* be run in a *live* manner concurrently with blktrace by specifying *-i -* to blkparse, and combining it with the live option for blktrace. An example would be:

  ```
  % blktrace -d /dev/sda -o - | blkparse -i -
  ```

- You can set how many blkparse batches event reads via the *-b* option, the default is to handle events in batches of 512.

- If you have saved event traces in blktrace with different output names (via the *-o* option to blktrace), you must specify the same *input* name via the *-i* option.

- The format of the output data can be controlled via the *-f* or *-F* options – see section 4.3 for details.

  By default, blkparse sends formatted data to standard output. This may be changed via the *-o* option, or text output can be disabled via the*-O* option. A merged binary stream can be produced using the *-d* option.

4.1 Command line arguments

Short	Long	Description
-b *batch*	–batch=batch	Standard input read batching
-i *file*	–input=*file*	Specifies base name for input files – default is *device*.blktrace.*cpu*. As noted above, specifying *-i* - runs in *live* mode with blktrace (reading data from standard in).
-F *typ,fmt* -f *fmt*	–format=*typ,fmt* –format-spec=*fmt*	Sets output format (See section 4.3 for details.) The -f form specifies a format for all events The -F form allows one to specify a format for a specific event type. The single-character *typ* field is one of the action specifiers in section 4.3.2
-m	–missing	Print missing entries
-h	–hash-by-name	Hash processes by name, not by PID
-o *file*	–output=*file*	Output file
-O	–no-text-output	Do *not* produce text output, used for binary (-d) only
-d *file*	–dump-binary=*file*	Binary output file
-q	–quiet	Quite mode
-s	–per-program-stats	Displays data sorted by program
-t	–track-ios	Display time deltas per IO
-w *span*	–stopwatch=*span*	Display traces for the *span* specified – where span can be: *end-time* – Display traces from time 0 through *end-time* (in ns) or *start:end-time* – Display traces from time *start* through end-time (in ns).
-v	–verbose	More verbose marginal on marginal errors
-V	–version	Display version

4.2 Trace actions

C – complete A previously issued request has been completed. The output will detail the sector and size of that request, as well as the success or failure of it.

D – issued A request that previously resided on the block layer queue or in the io scheduler has been sent to the driver.

I – inserted A request is being sent to the io scheduler for addition to the internal queue and later service by the driver. The request is fully formed at this time.

Q – queued This notes intent to queue io at the given location. No real requests exists yet.

B – bounced The data pages attached to this *bio* are not reachable by the hardware and must be bounced to a lower memory location. This causes a big slowdown in io performance, since the data must be copied to/from kernel buffers. Usually this can be fixed with using better hardware - either a better io controller, or a platform with an IOMMU.

M – back merge A previously inserted request exists that ends on the boundary of where this io begins, so the io scheduler can merge them together.

F – front merge Same as the back merge, except this io ends where a previously inserted requests starts.

G – get request To send any type of request to a block device, a *struct request* container must be allocated first.

S – sleep No available request structures were available, so the issuer has to wait for one to be freed.

P – plug When io is queued to a previously empty block device queue, Linux will plug the queue in anticipation of future ios being added before this data is needed.

U – unplug Some request data already queued in the device, start sending requests to the driver. This may happen automatically if a timeout period has passed (see next entry) or if a number of requests have been added to the queue.

T – unplug due to timer If nobody requests the io that was queued after plugging the queue, Linux will automatically unplug it after a defined period has passed.

X – split On raid or device mapper setups, an incoming io may straddle a device or internal zone and needs to be chopped up into smaller pieces for service. This may indicate a performance problem due to a bad setup of that raid/dm device, but may also just be part of normal boundary conditions. dm is notably bad at this and will clone lots of io.

A – remap For stacked devices, incoming io is remapped to device below it in the io stack. The remap action details what exactly is being remapped to what.

4.3 Output Description and Formatting

The output from blkparse can be tailored for specific use - in particular, to ease parsing of output, and/or limit output fields to those the user wants to see. The data for fields which can be output include:

Field Specifier	Description
a	Action, a (small) string (1 or 2 characters) – see table below for more details
c	CPU id
C	Command
d	RWBS field, a (small) string (1-3 characters) – see section below for more details
D	7-character string containing the major and minor numbers of the event's device (separated by a comma).
e	Error value
m	Minor number of event's device.
M	Major number of event's device.
n	Number of blocks
N	Number of bytes
p	Process ID
P	Display packet data – series of hexadecimal values
s	Sequence numbers
S	Sector number
t	Time stamp (nanoseconds)
T	Time stamp (seconds)
u	Elapsed value in microseconds (*-t* command line option)
U	Payload unsigned integer

Note that the user can optionally specify field display width, and optionally a left-aligned specifier. These precede field specifiers, with a '%' character, followed by the optional left-alignment specifer (-) followed by the width (a decimal number) and then the field.

Thus, to specify the command in a 12-character field that is left aligned:

```
-f "%-12C"
```

4.3.1 Action Table

The following table shows the various actions which may be output.

Act	Description
A	IO was remapped to a different device
B	IO bounced
C	IO completion
D	IO issued to driver
F	IO front merged with request on queue
G	Get request
I	IO inserted onto request queue
M	IO back merged with request on queue
P	Plug request
Q	IO handled by request queue code
S	Sleep request
T	Unplug due to timeout
U	Unplug request
X	Split

4.3.2 RWBS Description

This is a small string containing at least one character ('R' for read, 'W' for write operation), and optionally either a 'B' (for barrier operations) or 'S' (for synchronous operations).

4.3.3 Default output

The standard *header* (or initial fields displayed) include:

```
"%D %2c %8s %5T.%9t %5p %2a %3d "
```

Breaking this down:

%D Displays the event's device major/minor as: %3d,%-3d.

%2c CPU ID (2-character field).

%8s Sequence number

%5T.%9t 5-charcter field for the seconds portion of the time stamp and a 9-character field for the nanoseconds in the time stamp.

%5p 5-character field for the process ID.

%2a 2-character field for one of the actions.

%3d 3-character field for the RWBS data.

Seeing this in action:

```
8,0    3        1     0.000000000   697  G   W 223490 + 8 [kjournald]
```

The header is the data in this line up to the 223490 (starting block).
The default output for all event types includes this header.

Default output per action

C – complete If a payload is present, this is presented between parenthesis following the header, followed by the error value.

If no payload is present, the sector and number of blocks are presented (with an intervening plus (+) character). If the *-t* option was specified, then the elapsed time is presented. In either case, it is followed by the error value for the completion.

D – issued

I – inserted

Q – queued

B – bounced If a payload is present, the number of payload bytes is output, followed by the payload in hexadecimal between parenthesis.

If no payload is present, the sector and number of blocks are presented (with an intervening plus (+) character). If the *-t* option was specified, then the elapsed time is presented (in parenthesis). In either case, it is followed by the command associated with the event (surrounded by square brackets).

M – back merge

F – front merge

G – get request

S – sleep The starting sector and number of blocks is output (with an intervening plus (+) character), followed by the command associated with the event (surrounded by square brackets).

P – plug The command associated with the event (surrounded by square brackets) is output.

U – unplug

T – unplug due to timer The command associated with the event (surrounded by square brackets) is output, followed by the number of requests outstanding.

X – split The original starting sector followed by the new sector (separated by a slash (/) is output, followed by the command associated with the event (surrounded by square brackets).

A – remap Sector and length is output, along with the original device and sector offset.

Appendix: blktrace Kernel Guide

The blktrace facility provides an efficient event transfer mechanism which supplies block IO layer state transition data via the relay filesystem. This section provides some details as to the interfaces blktrace utilizes in the kernel to effect this. It is good background data to help understand some of the outputs and command-line options above.

4.4 blktrace.h Definitions

Files which include < *linux/blktrace.h* > are supplied with the following definitions:

4.4.1 Trace Action Specifiers

BLK_TA_QUEUE	(RQ) Command queued to request_queue. (BIO) Command queued by elevator.
BLK_TA_BACKMERGE	Back merging elevator operation
BLK_TA_FRONTMERGE	Front merging elevator operation
BLK_TA_GETRQ	Free request retrieved.
BLK_TA_SLEEPRQ	No requests available, device unplugged.
BLK_TA_REQUEUE	Request requeued.
BLK_TA_ISSUE	Command set to driver for request_queue.
BLK_TA_COMPLETE	Command completed by driver.
BLK_TA_PLUG	Device is plugged
BLK_TA_UNPLUG_IO	Unplug device as IO is made available.
BLK_TA_UNPLUG_TIMER	Unplug device after timer expired.
BLK_TA_INSERT	Insert request into queue.
BLK_TA_SPLIT	BIO split into 2 or more requests.
BLK_TA_BOUNCE	BIO was bounced
BLK_TA_REMAP	BIO was remapped

4.5 blktrace.h Routines

Files which include < *linux/blktrace.h* > are supplied with the following kernel routine invocable interfaces:

blk_add_trace_rq(struct request_queue *q, struct request_queue *rq, u32 what)
Adds a trace event describing the state change of the passed in request_queue. The *what* parameter describes the change in the request_queue state, and is one of the request queue action specifiers – BLK_TA_QUEUE, BLK_TA_REQUEUE, BLK_TA_ISSUE, or BLK_TA_COMPLETE.

blk_add_trace_bio(struct request_queue *q, struct bio *bio, u32 what)
Adds a trace event for the BIO passed in. The *what* parameter describes the action being performed on the BIO, and is one of BLK_TA_BACKMERGE, BLK_TA_FRONTMERGE, or BLK_TA_QUEUE.

blk_add_trace_generic(struct request_queue *q, struct bio *bio, int rw, u32 what)
Adds a *generic* trace event – not one of the request queue or BIO traces. The *what* parameter describes the action being performed on the BIO (if bio is non-NULL), and is one of BLK_TA_PLUG, BLK_TA_GETRQ or BLK_TA_SLEEPRQ.

blk_add_trace_pdu_int(struct request_queue *q, u32 what, u32 pdu) Adds a trace with some payload data – in this case, an unsigned 32-bit entity (the *pdu* parameter). The *what* parameter describes the nature of the payload, and is one of BLK_TA_UNPLUG_IO or BLK_TA_UNPLUG_TIMER.

blk_add_trace_remap(struct request_queue *q, struct bio *bio, dev_t dev, sector_t sector)
Adds a trace with a remap event. *dev* and *sector* denote the original device this *bio* was mapped from.