Disk File Monitor Design

Version 0.15.1

October 7, 2009

Steve Dobbelstein

Table of Contents

Catching Physical File I/O

Monitoring physical file I/O—reads and writes to the physical disks—is not as simple as monitoring the system calls for reading and writing files. The operating system has several sophisticated methods, such as caching, that make I/O more efficient for the end user. Using these methods, every read or write request from an application does not necessarily result in I/O to the physical disks. The first read of a file will generate I/O to the disk, while subsequent reads from the file are satisfied from the cache without generating I/O to the physical disks. Several writes to a file can be kept in cache and be flushed at a later time as a single I/O to the physical disks. The goal of this utility is to catch all the physical I/O and map it back to the files that were read or written. The statistics gathered show which files are being used in such a way that they generate more I/O to the physical disks. Knowing which file usages generate the most physical I/O will aid an administrator (or an automated program) to know which files are good candidates for being moved to faster storage, such as solid state drives (SSDs). It is not the amount of data being written to or read from a file that matters. The reads and writes can hit the cache and cause little physical I/O. Moving those files to faster storage won't help performance because so little I/O is satisfied with the physical medium. Rather, it is the files whose usage generates a lot of physical I/O that are good candidates for migration to faster storage.

Kprobes

Monitoring file I/O can be done by simply intercepting the system call for reads and writes. But we are not after file I/O; we want physical I/O. To do that we need to intercept the code paths in the kernel that generate physical I/O. To intercept kernel code paths we use kprobes. We place kprobes on strategic functions in the disk I/O code paths to monitor the disk I/O. Kprobes can only be run in kernel space, so the core of the Disk File Monitor (DFM) code is written as a kernel module.

The actual probes are implemented with jprobes rather than kprobes. A jprobe is a superset of a kprobe. Jprobes have the same function signature as the function on which they are placed and thus give the programmer easy access to the parameters that were passed to the function.

Each of the probes has a check at the beginning to see if the flag that says they should be running is on. Even with that check in place, the probes are not registered until they are started. They are also unregistered when they are stopped. The reason is that even with the check in place, the probes would add code to the disk I/O code paths when they are not running. In an effort to minimize the impact on performance, the probes are unregistered when they are not running.

`submit_bio`

All requests for disk I/O ultimately go through the kernel's `submit_bio` function, so we definitely place a probe on that function. However, the information provided to `submit_bio` does not enable us to map the I/O request back to a file. All that `submit_bio` knows, and in fact needs to know, is the type of I/O (read or write), the targeted device, the offset, and the length. In order to be able to map the I/O back to a file, we will need to intercept the I/O request at one of the parent functions that calls `submit_bio` that has enough information on its function call to map the I/O request back to a file.

`submit_bh`

Almost all of the I/O requests go through the kernel's `submit_bh` function. (More later on I/O that does not go through `submit_bh`.) One of the parameters to `submit_bh` is a pointer to a `buffer_head` structure. Within the `buffer_head` structure is a pointer to an `address_mapping` structure. Within the `address_mapping` structure is a pointer to an `inode` structure. The inode is sufficient for identifying the file for which the disk request is being submitted. We therefore put a probe on `submit_bh`. The `submit_bh` probe grabs a `bh_inode_t` structure (also known in the code as a `bhi`), fills it in, and puts it in the `bhi_hashtable`.

```
typedef struct bh_inode_s {
        struct hlist_node       bhi_hash;       /* Link into the hash table */
        struct task_struct *    bhi_task;       /* Task on which this I/O was issued */
        struct kretprobe *      bhi_krp;        /* kretprobe for the caller */
                                                /* Only this kretprobe is allowed to take */
                                                /* the bh_inode_t off the hash table. */
        struct inode *          bhi_inode;      /* inode for the I/O */
        struct inode *          bhi_aux_inode;  /* Hint for the real inode if bhi_inode */
                                                /* is NULL */
        char *                  bhi_cmd_prefix; /* Prefix to put on the basic I/O command */
                                                /* e.g., "Direct-" */
} bh_inode_t;
```

One of the fields in the `bh_inode_t` structure is a pointer to the current `task_struct`, that is, a field that identifies on which process the disk request is being submitted. The `submit_bio` probe searches the `bhi_hashtable` for a `bh_inode_t` for the current process. If it finds a `bh_inode_t` for the current process, it uses the information in the `bh_inode_t` for logging the statistics for the disk request.

Other Probe Points

There are some I/O code paths to `submit_bio` that do not go through `submit_bh`. Without full knowledge of the disk subsystem, these paths were discovered experimentally by putting code in the `submit_bio` probe to dump the stack when it got called but did not find a `bh_inode_t` for the current process that was saved from the `submit_bh` probe. The following functions were identified using this method.

- `generic_file_aio_read`
- `generic_file_direct_write`
- `blockdev_direct_IO`
- `generic_file_buffered_write`
- `filemap_fdatawrite`
- `__do_page_cache_readahead`
- `journal_do_submit_data`

Additional probes are put on these functions to catch the disk requests going through them. Each probe creates a `bh_inode_t` for the current process and puts it in the hash, just as `submit_bh` does.

Sometimes the code path through these functions ends up going through `submit_bh` anyway. The `submit_bh` probe checks if a `bh_inode_t` for the current process is already in the hash table before it creates one.

Kretprobes

Once the disk request has been submitted through `submit_bio`, the `bh_inode_t` for the current thread can be removed from the hash. In fact, the `bh_inode_t` *should* be removed from the hash so that any subsequent disk requests in the current process are not mistakenly logged against the inode for the previous disk request.

Each probe, with the exception of the one for `submit_bio`, has a corresponding kretprobe that removes the `bh_inode_t` from the hash. The return code path can go through multiple functions that are probed, e.g., `submit_bh` and `generic_file_buffered_write`. Only the function that created the `bh_inode_t` should be the one to remove it from the hash. It is possible for a higher level function to call `submit_bh` several times before it returns. If `submit_bh` were to remove the `bh_inode_t` from the hash, a subsequent disk request from the higher level function would not be logged, since `submit_bio` would not find a `bh_inode_t` for the current thread.

The code for removing a `bh_inode_t` from the hash is the same for all kretprobes. However, since we need to know which kretprobe to use, there are separate `kretprobe` structures to go with each probe. Each `kretprobe` structure points to the common function for removing a `bh_inode_t` from the hash. When the jprobe creates the `bh_inode_t` it stores in it the address of its corresponding `kretprobe` structure. When the common function is called, one of its parameters is the `kretprobe` structure for which it is being called. The common function searches the bhi hash for the `bh_inode_t` for the current process. If the pointer to the `kretprobe` structure stored in the `bh_inode_t` matches the pointer to the `kretprobe` structure it was passed as a parameter, it knows it is being called on the correct kretprobe instance and removes the `bh_inode_t` from the hash. Otherwise, it is not being called for the proper kretprobe instance and leaves the `bh_inode_t` in the hash.

Memory Allocation Optimization

In order to reduce the overhead that DFM adds to the disk I/O code path, the `bh_inode_t` structures are not actually removed from the bhi hash. Instead of allocating and freeing the structures, which adds additional code paths, the `bh_inode_t` structures are left in the hash. The kretprobe code simply clears the significant fields in the `bh_inode_t`. The `submit_bio` probe, when looking for a `bh_inode_t` for the current task, not only checks if one exists but also checks if it is filled in. The other probes that generate `bh_inode_t` structures, such as the one for `submit_bh`, first check if there is a cleared `bh_inode_t` for the current task in the hash before allocating one from memory.

Once a process exits, though, the `bh_inode_t` will be left abandoned in the hash. We don't want to cause a memory leak in the case where processes may be coming and going a lot. We therefore put a probe on `do_exit` and remove from the hash any `bh_inode_t` that may be there for the process that is exiting.

Excluding Devices

A system may have numerous physical disks. A user may be interested in monitoring the files on only a subset of the physical disks on the system. For example, the user may only be interested in migrating files from a specific file system. Monitoring all the disks can end up dumping a lot of data, most of which the user would ignore. DFM provides a way of excluding devices from being monitored.

The user can set a list of devices to be excluded with the `exclude=<device-list>` parameter when the module is loaded or through the `exclude` file in debugfs during run time. (See the `dfm-ctrl` script "exclude" command on page 14 for more details.) The devices are specified as a list of <major>:<minor> pairs separated by spaces. Internally, DFM keeps the list of excluded devices as a zero-terminated array of `dev_t` values. Each of the probes has a check at the beginning to see if the `struct block_device` for the request is on the exclude list. If it is, the probe simply returns without logging any statistics.

Logging

Which messages are logged is determined by the verbose level. Each message has a level associated with it. Messages with a level less than or equal to the verbose level are logged. The verbose level can be set with the `verbose=<level>` parameter when the module is loaded or via the `verbose` file in debugfs during run time. (See the `dfm-ctrl` script "verbose" command on page 14 for more details.) The DFM verbose levels match the kernel's `printk` levels defined in include/linux/kernel.h.

DFM maintains its own internal, in-memory log buffer. In most cases, DFM writes messages to its own log rather than using `printk()`. `printk()` uses more CPU since it activates the syslog mechanism. We want DFM to have the smallest possible impact on the system. `printk()` will also cause additional I/O when the log messages are written to `/var/log/messages`. That I/O is picked up by DFM. If we are not careful with our logging, the tracing of the I/O to `/var/log/messages` can generate more calls to `printk()` which causes more writes to `/var/log/messages` ...

Calls to `printk()` are limited to the initialization code and to messages with the verbose level of CRIT or lower.

Collecting Statistics

The DFM architecture separates the probes from the details of how to collect the statistics. The probes provide the basic information about the I/O request in a call to the `add_IO_stat` function.

```
int add_IO_stat(struct block_device * bdev,
                struct inode * inode,
                char * cmd,
                unsigned int bytes);
```

The `bdev` and `inode` are sufficient to identify the file. In fact, the `inode` itself is sufficient since it has a pointer to the block device. However, there are cases where the `inode` is NULL and `bdev` is needed to at least log the statistics against the block device.

`cmd` is a string that specifies the I/O command, for example, "Read", "Readahead", "Direct-Write", "Metadata-Read".

`bytes` is the number of bytes of I/O.

`add_IO_stat` does the work to log the I/O statistics into the debugfs tree.

DebugFS

The I/O statistics are placed in the debugfs tree for two reasons. One is that if they were saved to a file that would generate disk I/O which would be caught by the probes and result in recursive death unless code was added to detect that the I/O was from the probes and not from the rest of the system. Yuck.

The second reason is that since debugfs is in memory, it is quick. This utility should have the smallest amount of overhead so that it has the least impact on system performance.

Each statistic entry in the debugfs tree is represented by a `dfs_entry_t` structure.

```
extern struct inode * dummy;
typedef typeof(dummy->i_ino) inode_i_ino_t;
typedef typeof(dummy->i_generation) inode_i_gen_t;

typedef struct dfs_entry_s {
      struct hlist_node   dfs_hash;    /* Link into the hash table */
      char                dfs_bdev_name[BDEVNAME_SIZE];
                                       /* Block device of the file */
      inode_i_ino_t       dfs_i_ino;
      inode_i_gen_t       dfs_i_gen;
      lstring_t           dfs_filename;/* Full path of the file name */
      lstring_t           dfs_cmd;     /* I/O command (read, write, etc.) */
      struct dentry *     dfs_dentry;  /* debugfs tree dentry for the file */
                                       /* that shows the stats */
#ifdef ATOMIC64_INIT
      atomic64_t          dfs_count;   /* Number of I/Os */
      atomic64_t          dfs_bytes;   /* Number of bytes of I/O */
#else
      /* No protection on 32-bit architectures */
      i64                 dfs_count;   /* Number of I/Os */
      i64                 dfs_bytes;   /* Number of bytes of I/O */
#endif
      void *              dfs_private; /* Private field for layout handlers */
} dfs_entry_t;
```

Notice that rather than saving the pointers to the structures for the the block device and inode, the block device name, inode number and inode generation are saved instead. The reason is that `block_device` and `inode` structures can come into and go out of existence. There is no guarantee that a given inode pointer will point to the same `inode` (or even an inode at all) after the system has been running for several minutes, hours, days, or however long DFM is left running. The device name, inode number, and inode generation however, will remain constant for a given file.

The `dfs_count` field holds the number of I/O requests of this type (read, write, readahead, etc.) for this file. The `dfs_bytes` field holds the running total of bytes of I/O of this type (read, write, readahead, etc.) for this file. This is where the beauty of using debugfs comes in. The debugfs file entry for this file/type points to the `dfs_entry_t` as the location of the value for the debugfs file. A read from the debugfs file reads the `dfs_count` and `dfs_bytes` fields directly. An update to the file's I/O count and total bytes is simply an update to the `dfs_entry_t` with no other code needed to update the debugfs file. The update is quick. Any conversion to human readable format is only done when the file is read, and all that work is done by debugfs.

Layouts

So exactly how are the statistics laid out in the debugfs tree? However you like! The DFM architecture has pluggable layouts. If you don't like the DFM layouts available, feel free to write your own.

A layout handler exports the following interface.

```
typedef struct layout_handler_s {

      int  (*create) (struct dentry * root, dfs_entry_t * dfs);
      void (*delete) (struct dentry * root, dfs_entry_t * dfs);
      char name[MAX_LAYOUT_NAME_LEN];
} layout_handler_t;
```

A layout handler can optionally give an `init` function if it needs to do some setup before it runs, such as allocating memory. It can also optionally give an `exit` function if it needs to cleanup anything when it is finished.

DFM does not initialize all of the layout handlers when it starts up. It only initializes the the one that is to be active. If the user ever changes the layout, DFM will call the `exit` function of the current layout handler and then the `init` function of the new layout handler. This is done to keep the memory footprint low. Besides, there is no reason why a layout handler that is not in use should have any resources allocated.

A layout handler must provide two functions—`create` and `delete`. Each function is passed the root of the debugfs statistics tree and a pointer to a `dfs_entry_t`. All of the information for the debugfs entry is contained in the `dfs_entry_t` structure. The create function does the work to create the debugfs file for the file/type according to the layout being implemented and saves into the `dfs_dentry` field in the `dfs_entry_t` the dentry for the debugfs file that points to the `dfs_entry_t`. The `delete` function does the work needed to cleanup the debugfs tree when the debugfs file for the file/type is removed.

This version of DFM provides two layouts, *filetree* and *namehash*.

Filetree

The *filetree* layout creates the debugfs entries using the same file name and directory structure of the currently mounted file systems. The name of the file for which the statistics are being collected becomes the name of a directory in the debugfs tree. Within that directory are files with the names of the different types of I/O that have been made to the file. For example, the count of requests and the number of bytes requested for type "Readahead" on file `/usr/src/linux/kernel/sched.c` would be logged in the file `<debugfs-root>/dfm/statistics/usr/src/linux/kernel/sched.c/Readahead`.

Namehash

The filetree layout nicely preserves the file structure of the system. It makes it obvious which files the statistics are for. However, it can take a long time to traverse the directory structure when it comes time to gather the statistics. Gathering the statistics should be fast so that the interruption to collecting the statistics is minimal. (Any program gathering statistics from the DFM debugfs tree should stop the DFM kernel module's probes before gathering the statistics to avoid the danger of trying to traverse a changing directory tree. Stopping the probes for too long can result in a significant loss of statistics collected.)

The *namehash* layout creates a unique hash value for each file and creates a directory that is named with the ASCII string of the hexadecimal value of the hash. As with the filetree layout, the directory contains files with the names of the different types of I/O that have been made to the file. Any program that gathers the statistics needs to traverse only two levels deep to get the statistics. In this layout the name of the file is not apparent. The namehash layout puts a file named `filename` into the directory along with the I/O type files. The `filename` file contains the full path of the file name.

DFM DebugFS Tree Structure

DFM creates a directory named `dfm` in the root of the debugfs tree. All the DFM debugfs activity occurs in the `dfm` directory. The statistics are kept in a `statistics` subdirectory in the `dfm` directory according to the layout that is currently in use.

DFM maintains several control files in the `dfm` directory.

File name	**Function**
`running`	Read this file to see if the probes are running. (The DFM module can be loaded but not running.) 0 = probes stopped 1 = probes running Write a "0" to this file to stop the probes. Write a non-zero value to the file to start the probes.
`reset`	Write a non-zero value to this file to reset the statistics. The probes are temporarily stopped, everything in the `dfm/statistics` directory is deleted, and the probes are restarted.
`layout`	Read this file to see the available layouts and which one is currently being used. The one in use is in square brackets. Write a layout name to this file to set a new layout. Setting a new layout causes a reset of the statistics.
`exclude`	Read this file to see the list of devices, indicated by <major>:<minor>, that are currently excluded from being monitored. Write a list of devices, <major>:<minor> couples separated by spaces, to this file to set the list of excluded devices. Write and empty string to this file to clear the list of excluded devices.
`verbose`	Read this file to see the verbose level. The verbose level determines which messages get logged. Write a number to this file to set the verbose level. See dfm.h for a listing of the verbose levels.
`start_time`	Read this file to see the time that DFM started collecting statistics. If the probes were never started since the module was loaded, this file will be empty.
`stop_time`	Read this file to see the time that DFM stopped collecting statistics. If the probes were never started since the module was loaded, or if DFM is currently collecting statistics, this file will be empty.
`log`	Read this file to see DFM's in-memory log.

Worker Threads

Most of the work for creating a debugfs entry for a statistic is farmed out to a kernel thread. There are two reasons for this. One is that the creation of the debugfs entries can be rather involved and time consuming. In order to reduce the overhead added to the disk I/O path, only the essential code for collecting the statistics is run in the context of a kprobe. The rest of the work is offloaded to another thread.

The second reason is that kprobe code must not block. Bad things happen if a kprobe gets rescheduled on a different CPU. Although none of the DFM code blocks, it does make calls to kernel functions which may block. Those calls are all in the code paths for creating a debugfs entry for a statistic. Thus, that code is removed from the kprobe context and put into the context of a kernel thread, which is allowed to block.

The offloading of the work means that there is a short time where DFM has some statistics for a file but has not yet created a debugfs entry for it. The window should be small, and it will not be noticed by the user because the user won't have any way of knowing that a non-existent file is supposed to exist.

The `dfm` Script

The `dfm` bash script provides a user-friendly interface for gathering the DFM statistics.

The script locates the debugfs tree, usually mounted on `/sys/kernel/debug`. If the debugfs file system is not mounted, the script will mount it, assuming the kernel has support for the debugfs file system.

The script then loads the DFM module, if needed, starts the probes, if needed, and gathers the DFM statistics from the debugfs tree. For each gathering of the statistics the script stops the probes, gathers all the data from the `dfm/statistics/` debugfs tree, prints them out, resets the statistics, and restarts the probes. When the script exits it unloads the DFM module if it loaded it.

Parameters

The script has three optional parameters—*interval, iterations*, and *exclude*. The *interval* parameter is used to set the delay between the gathering of the statistics. The interval can be set in seconds, minutes, hours, or days. The default value is 5 seconds. For example, if the user wants to gather the statistics every 5 minutes, the user could run: `dfm run interval=5m`

The *iterations* parameter is used to set the number of times the script gathers the statistics. The default number of iterations is 24 hours divided by the interval. At the default interval of 5 seconds, the default number of iterations is 60*60*24/5 = 17,280. For example, if the user wants to collect the statistics every ten seconds for five minutes, the user would run:

```
dfm run interval=10 iterations=30
```

The *exclude* parameter is used to set the list of devices that should be excluded from being monitored. The user may be interested in monitoring the files on only a subset of the physical disks on the system. For example, the user may only be interested in monitoring files from a specific file system. The devices may be specified in either `[/dev/]<name>` or `<major>:<minor>` format. If more than one device is specified, the list must be enclosed in quotes so that the list is passed as a single parameter, for example, `exclude="sda sdb 8:96"`. The debugfs interface for the exclude list takes only <major>:<minor> values. The script converts device names to <major>:<minor> values before writing the list to the debugfs interface.

Options

The script gathers the statistics, combining all the statistics for the various read I/O types into one read statistics and combining all the statistics for the various write I/O types into one write statistics. It prints the output in a human readable format.

The script has an option `-r` or `--raw` which tells it to just dump the raw statistics. The option strips the labels and reduces the whitespace for easier parsing. This option is useful if the user wants to process the output with another program.

Signals

The `dfm` script traps several signals to add functionality. The functions provided by SIGUSR2 and SIGUSR1 were added mainly to allow DFM to be used as a profiler in our automated benchmark test framework, but they can be useful outside of the framework as well.

SIGUSR2

The `dfm` script interprets the SIGUSR2 signal to mean "suspend". When it receives this signal it stops the probes and goes into sleep a loop waiting for another signal. It leaves the statistics in the debugfs tree untouched.

SIGUSR1

The `dfm` script interprets the SIGUSR1 signal to mean "resume" or "dump", depending on whether it is suspended or not. If the script was suspended, the signal will kill the sleep command so that the script breaks out of its suspend loop and resumes at the place where it dumps the statistics, resets them, and then loops back to start the probes and wait for the interval, as it normally would.

If the script was not suspended, the signal will kill the sleep command that is waiting for the interval to pass. The script will then proceed to dump the statistics. SIGUSR1 effectively cancels the interval and says, "dump now". This behavior is modeled off the `dd` command which dumps its current progress if you send it SIGUSR1.

Terminating Signals

The script traps signals that would cause it to terminate—SIGINT, SIGQUIT, SIGABRT, SIGSEGV, and SIGTERM. When it gets a termination signal it does one last gathering of the statistics before it exits.

This feature allows the user to collect statistics for an undefined period of time. To monitor disk usage during the run of a program the user could startup DFM with `dfm interval=1day`, run the program, and then press Ctrl-c, or run `killall dfm` to get the disk statistics for the run of the program.

The `dfm-ctrl` Script

The `dfm-ctrl` bash script provides a user-friendly interface to the DFM debugfs files. It aids in controlling the behavior of DFM and in collecting the DFM statistics.

The script locates the debugfs tree, usually mounted on `/sys/kernel/debug`. If the debugfs file system is not mounted, the script will mount it, assuming the kernel has support for the debugfs file system.

The `dfm-ctrl` script provides several commands that make it easier to work with DFM control files and statistics. The structure of the debugfs tree is abstracted away to simple commands.

load

The *load* command attempts to load the DFM module. In an effort to support development of the utility, the script does not require that the module be installed in the `/lib/modules/`uname -r`` directory tree. The script checks to see if the module exists in the same directory from which the script was run. If it finds the module there, it uses `insmod` to load the module. If it does not find the module there, is uses `modprobe` to load the module. Thus, if the script is run from the build directory, it will find the built module in the directory and will load that version of the module. If the script is invoked from its installed directory, `/usr/local/sbin`, it will not find the DFM module there and will run `modprobe`, which will load the installed version of the module, the one in `/lib/modules/`uname -r`/extra/`.

start

The *start* command will load the DFM module if it is not already loaded and then, if the probes are not already running, echoes "1" to `<debugfs_mount>/dfm/running` to start the probes.

stop

The *stop* command checks to see if the DFM probes are running (i.e., `cat <debugfs_mount>/dfm/running` does not return 0) and, if so, echoes "0" to `<debugfs_mount>/dfm/running` to stop the probes.

reset

The *reset* command echoes a "1" to the `<debugfs_mount>/dfm/reset` file which tells the DFM module to stop the probes if they are running, clear the statistics, and restart the probes if they were running.

stats

The *stats* command dumps the statistics in the debugfs tree. The command has a safety check to make sure that the probes are not running before it gathers the statistics. Gathering the statistics involves walking the `<debugfs_mount>/dfm/statistics/` tree. If the probes are running, the contents of the tree can be changing, making a tree walk dangerous. The user can override the safety check with the `--force` option. (The force option can also be specified as `-f` or `force=yes`.)

status

The *status* command displays whether the DFM module is loaded or not, what layouts are available, the current layout being used, the current verbose level, whether the probes are running or not, what time the probes started running, and what time they stopped, if the probes are stopped.

layout

Without any parameters, the *layout* command displays the layout that DFM is currently using. If the *layout* command is given a parameter, it will attempt to set the DFM layout to the parameter given. The *layout* command checks to make sure that the layout given appears in the list of layouts supported in `<debugfs_mount>/dfm/layout`. Changing the layout resets the statistics.

exclude

Without any parameters, the *exclude* command displays the list of devices that are currently excluded from being monitored. The devices are listed as <major>:<minor> pairs. As a help to the user, the script finds the device name associated with the <major>:<minor> and displays the name in parenthesis next to the <major>:<minor>, for example: `8:0 (/dev/sda)` If the *exclude* command is given parameters it will use them to set the list of excluded devices. Devices can be specified in the <major>:<minor> format or by name with the optional `/dev/` prefix, e.g., `sda`.

The *exclude* command has several subcommands to aid in setting the list of devices. The debugfs interface to DFM only supports the setting of the entire list. Users, however, may want to add devices to the list or remove a subset of devices from the list. Also, since the exclude command interprets the lack of parameters to be a query of the excluded devices list, there is no way to set an empty lists, i.e., clear the list. The exclude command supports the following subcommands.

add

The *add* subcommand adds devices to the excluded devices list. As with the native *exclude* command, devices can be specified in the <major>:<minor> format or by name. The *add* subcommand reads the current list of excluded devices from `<debugfs_mount>/dfm/exclude`, appends the new devices, and writes the new list to `<debugfs_mount>/dfm/exclude`.

remove

The *remove* subcommand removes devices from the excluded devices list. As with the native *exclude* command, devices can be specified in the <major>:<minor> format or by name. The *remove* subcommand reads the current list of excluded devices from `<debugfs_mount>/dfm/exclude`. Then for each device given to the command, it searches the list of excluded devices for the given device. If it finds it, it removes it from the list. It then write the new list back to `<debugfs_mount>/dfm/exclude`.

clear

The *clear* subcommand writes an empty list to `<debugfs_mount>/dfm/exclude`, which effectively removes all devices from the exclude list.

verbose

Without any parameters, the *verbose* command displays the current DFM verbose level. It simply does a `cat <debugfs_mount>/dfm/verbose`. If the *verbose* command is given a parameter, it will attempt to set the DFM verbose level to the parameter given by echoing the value into `<debugfs_mount>/dfm/verbose`. The *verbose* command checks to make sure that the verbose level given is a positive integer. It will also accept the keyword names for each level: emerg[ency] (0), alert (1), crit[ical] (2), err[or] (3), warn[ing] (4), notice (5), info (6), and debug (7).

log

The *log* command dumps out the DFM internal log. It simply does a `cat` of the `<debugfs_mount>/dfm/log` file. For convenience, the *log* command checks to see if standard out is a tty and the environment has the PAGER variable set. If so, it pipes the output of the log file into the pager for easy browsing.

unload

The *unload* command unloads the DFM module. It simply does an `rmmod dfm_mod` if the module is loaded.