relay interface (formerly relayfs)
==================================

The relay interface provides a means for kernel applications to efficiently log and transfer large quantities of data from the kernel to userspace via user-defined 'relay channels'.

A 'relay channel' is a kernel->user data relay mechanism implemented as a set of per-cpu kernel buffers ('channel buffers'), each represented as a regular file ('relay file') in user space. Kernel clients write into the channel buffers using efficient write functions; these automatically log into the current cpu's channel buffer. User space applications mmap() or read() from the relay files and retrieve the data as it becomes available. The relay files themselves are files created in a host filesystem, e.g. debugfs, and are associated with the channel buffers using the API described below.

The format of the data logged into the channel buffers is completely up to the kernel client; the relay interface does however provide hooks which allow kernel clients to impose some structure on the buffer data. The relay interface doesn't implement any form of data filtering - this also is left to the kernel client. The purpose is to keep things as simple as possible.

This document provides an overview of the relay interface API. The details of the function parameters are documented along with the functions in the relay interface code - please see that for details.

Semantics
=========

Each relay channel has one buffer per CPU, each buffer has one or more sub-buffers. Messages are written to the first sub-buffer until it is too full to contain a new message, in which case it it is written to the next (if available). Messages are never split across sub-buffers. At this point, userspace can be notified so it empties the first sub-buffer, while the kernel continues writing to the next.

When notified that a sub-buffer is full, the kernel knows how many bytes of it are padding i.e. unused space occurring because a complete message couldn't fit into a sub-buffer. Userspace can use this knowledge to copy only valid data.

After copying it, userspace can notify the kernel that a sub-buffer has been consumed.

A relay channel can operate in a mode where it will overwrite data not yet collected by userspace, and not wait for it to be consumed.

The relay channel itself does not provide for communication of such data between userspace and kernel, allowing the kernel side to remain simple and not impose a single interface on userspace. It does provide a set of examples and a separate helper though, described below.

The read() interface both removes padding and internally consumes the read sub-buffers; thus in cases where read(2) is being used to drain the channel buffers, special-purpose communication between kernel and user isn't necessary for basic operation.

One of the major goals of the relay interface is to provide a low overhead mechanism for conveying kernel data to userspace. While the read() interface is easy to use, it's not as efficient as the mmap() approach; the example code attempts to make the tradeoff between the two approaches as small as possible.

klog and relay-apps example code

The relay interface itself is ready to use, but to make things easier, a couple simple utility functions and a set of examples are provided.

The relay-apps example tarball, available on the relay sourceforge site, contains a set of self-contained examples, each consisting of a pair of .c files containing boilerplate code for each of the user and kernel sides of a relay application. When combined these two sets of boilerplate code provide glue to easily stream data to disk, without having to bother with mundane housekeeping chores.

The 'klog debugging functions' patch (klog.patch in the relay-apps tarball) provides a couple of high-level logging functions to the kernel which allow writing formatted text or raw data to a channel, regardless of whether a channel to write into exists or not, or even whether the relay interface is compiled into the kernel or not. These functions allow you to put unconditional 'trace' statements anywhere in the kernel or kernel modules; only when there is a 'klog handler' registered will data actually be logged (see the klog and kleak examples for details).

It is of course possible to use the relay interface from scratch, i.e. without using any of the relay-apps example code or klog, but you'll have to implement communication between userspace and kernel, allowing both to convey the state of buffers (full, empty, amount of padding). The read() interface both removes padding and internally consumes the read sub-buffers; thus in cases where read(2) is being used to drain the channel buffers, special-purpose communication between kernel and user isn't necessary for basic operation. Things such as buffer-full conditions would still need to be communicated via some channel though.

klog and the relay-apps examples can be found in the relay-apps tarball on http://relayfs.sourceforge.net

The relay interface user space API

The relay interface implements basic file operations for user space access to relay channel buffer data. Here are the file operations that are available and some comments regarding their behavior:

```
open()      enables user to open an _existing_ channel buffer.
```

```
mmap()       results in channel buffer being mapped into the caller's
             memory space. Note that you can't do a partial mmap - you
             must map the entire file, which is NRBUF * SUBBUFSIZE.

read()       read the contents of a channel buffer.  The bytes read are
             'consumed' by the reader, i.e. they won't be available
             again to subsequent reads.  If the channel is being used
             in no-overwrite mode (the default), it can be read at any
             time even if there's an active kernel writer.  If the
             channel is being used in overwrite mode and there are
             active channel writers, results may be unpredictable -
             users should make sure that all logging to the channel has
             ended before using read() with overwrite mode.  Sub-buffer
             padding is automatically removed and will not be seen by
             the reader.

sendfile()   transfer data from a channel buffer to an output file
             descriptor. Sub-buffer padding is automatically removed
             and will not be seen by the reader.

poll()       POLLIN/POLLRDNORM/POLLERR supported.  User applications are
             notified when sub-buffer boundaries are crossed.

close()      decrements the channel buffer's refcount.  When the refcount
             reaches 0, i.e. when no process or kernel client has the
             buffer open, the channel buffer is freed.
```

In order for a user application to make use of relay files, the host filesystem must be mounted. For example,

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

```
NOTE:   the host filesystem doesn't need to be mounted for kernel
        clients to create or use channels - it only needs to be
        mounted when user space applications need access to the buffer
        data.
```

The relay interface kernel API

Here's a summary of the API the relay interface provides to in-kernel clients:

```
TBD(curr. line MT:/API/)
  channel management functions:

    relay_open(base_filename, parent, subbuf_size, n_subbufs,
               callbacks, private_data)
    relay_close(chan)
    relay_flush(chan)
    relay_reset(chan)

  channel management typically called on instigation of userspace:

    relay_subbufs_consumed(chan, cpu, subbufs_consumed)
```

```
  write functions:

    relay_write(chan, data, length)
    __relay_write(chan, data, length)
    relay_reserve(chan, length)

  callbacks:

    subbuf_start(buf, subbuf, prev_subbuf, prev_padding)
    buf_mapped(buf, filp)
    buf_unmapped(buf, filp)
    create_buf_file(filename, parent, mode, buf, is_global)
    remove_buf_file(dentry)

  helper functions:

    relay_buf_full(buf)
    subbuf_start_reserve(buf, length)
```

Creating a channel
----

relay_open() is used to create a channel, along with its per-cpu channel buffers. Each channel buffer will have an associated file created for it in the host filesystem, which can be and mmapped or read from in user space. The files are named basename0...basenameN-1 where N is the number of online cpus, and by default will be created in the root of the filesystem (if the parent param is NULL). If you want a directory structure to contain your relay files, you should create it using the host filesystem's directory creation function, e.g. debugfs_create_dir(), and pass the parent directory to relay_open(). Users are responsible for cleaning up any directory structure they create, when the channel is closed - again the host filesystem's directory removal functions should be used for that, e.g. debugfs_remove().

In order for a channel to be created and the host filesystem's files associated with its channel buffers, the user must provide definitions for two callback functions, create_buf_file() and remove_buf_file(). create_buf_file() is called once for each per-cpu buffer from relay_open() and allows the user to create the file which will be used to represent the corresponding channel buffer. The callback should return the dentry of the file created to represent the channel buffer. remove_buf_file() must also be defined; it's responsible for deleting the file(s) created in create_buf_file() and is called during relay_close().

Here are some typical definitions for these callbacks, in this case using debugfs:

```
/*
 * create_buf_file() callback.  Creates relay file in debugfs.
 */
static struct dentry *create_buf_file_handler(const char *filename,
                                              struct dentry *parent,
```

```
                                        int mode,
                                        struct rchan_buf *buf,
                                        int *is_global)
{
        return debugfs_create_file(filename, mode, parent, buf,
                                   &relay_file_operations);
}

/*
 * remove_buf_file() callback.  Removes relay file from debugfs.
 */
static int remove_buf_file_handler(struct dentry *dentry)
{
        debugfs_remove(dentry);

        return 0;
}

/*
 * relay interface callbacks
 */
static struct rchan_callbacks relay_callbacks =
{
        .create_buf_file = create_buf_file_handler,
        .remove_buf_file = remove_buf_file_handler,
};
```

And an example relay_open() invocation using them:

```
  chan = relay_open("cpu", NULL, SUBBUF_SIZE, N_SUBBUFS, &relay_callbacks,
NULL);
```

If the create_buf_file() callback fails, or isn't defined, channel creation and thus relay_open() will fail.

The total size of each per-cpu buffer is calculated by multiplying the number of sub-buffers by the sub-buffer size passed into relay_open(). The idea behind sub-buffers is that they're basically an extension of double-buffering to N buffers, and they also allow applications to easily implement random-access-on-buffer-boundary schemes, which can be important for some high-volume applications. The number and size of sub-buffers is completely dependent on the application and even for the same application, different conditions will warrant different values for these parameters at different times. Typically, the right values to use are best decided after some experimentation; in general, though, it's safe to assume that having only 1 sub-buffer is a bad idea - you're guaranteed to either overwrite data or lose events depending on the channel mode being used.

The create_buf_file() implementation can also be defined in such a way as to allow the creation of a single 'global' buffer instead of the default per-cpu set. This can be useful for applications interested mainly in seeing the relative ordering of system-wide events without the need to bother with saving explicit timestamps for the purpose of merging/sorting per-cpu files in a postprocessing step.

To have relay_open() create a global buffer, the create_buf_file() implementation should set the value of the is_global outparam to a non-zero value in addition to creating the file that will be used to represent the single buffer. In the case of a global buffer, create_buf_file() and remove_buf_file() will be called only once. The normal channel-writing functions, e.g. relay_write(), can still be used - writes from any cpu will transparently end up in the global buffer - but since it is a global buffer, callers should make sure they use the proper locking for such a buffer, either by wrapping writes in a spinlock, or by copying a write function from relay.h and creating a local version that internally does the proper locking.

The private_data passed into relay_open() allows clients to associate user-defined data with a channel, and is immediately available (including in create_buf_file()) via chan->private_data or buf->chan->private_data.

Buffer-only channels

These channels have no files associated and can be created with relay_open(NULL, NULL, ...). Such channels are useful in scenarios such as when doing early tracing in the kernel, before the VFS is up. In these cases, one may open a buffer-only channel and then call relay_late_setup_files() when the kernel is ready to handle files, to expose the buffered data to the userspace.

Channel 'modes'
----

relay channels can be used in either of two modes - 'overwrite' or 'no-overwrite'. The mode is entirely determined by the implementation of the subbuf_start() callback, as described below. The default if no subbuf_start() callback is defined is 'no-overwrite' mode. If the default mode suits your needs, and you plan to use the read() interface to retrieve channel data, you can ignore the details of this section, as it pertains mainly to mmap() implementations.

In 'overwrite' mode, also known as 'flight recorder' mode, writes continuously cycle around the buffer and will never fail, but will unconditionally overwrite old data regardless of whether it's actually been consumed. In no-overwrite mode, writes will fail, i.e. data will be lost, if the number of unconsumed sub-buffers equals the total number of sub-buffers in the channel. It should be clear that if there is no consumer or if the consumer can't consume sub-buffers fast enough, data will be lost in either case; the only difference is whether data is lost from the beginning or the end of a buffer.

As explained above, a relay channel is made of up one or more per-cpu channel buffers, each implemented as a circular buffer subdivided into one or more sub-buffers. Messages are written into the current sub-buffer of the channel's current per-cpu buffer via the write functions described below. Whenever a message can't fit into the current sub-buffer, because there's no room left for it, the client is notified via the subbuf_start() callback that a switch to a new sub-buffer is about to occur. The client uses this callback to 1) initialize the next sub-buffer if appropriate 2) finalize the previous sub-buffer if appropriate and 3) return a boolean value indicating whether or not to actually move on to the next sub-buffer.

To implement 'no-overwrite' mode, the userspace client would provide an implementation of the subbuf_start() callback something like the following:

```
static int subbuf_start(struct rchan_buf *buf,
                        void *subbuf,
                        void *prev_subbuf,
                        unsigned int prev_padding)
{
        if (prev_subbuf)
                *((unsigned *)prev_subbuf) = prev_padding;

        if (relay_buf_full(buf))
                return 0;

        subbuf_start_reserve(buf, sizeof(unsigned int));

        return 1;
}
```

If the current buffer is full, i.e. all sub-buffers remain unconsumed, the callback returns 0 to indicate that the buffer switch should not occur yet, i.e. until the consumer has had a chance to read the current set of ready sub-buffers. For the relay_buf_full() function to make sense, the consumer is responsible for notifying the relay interface when sub-buffers have been consumed via relay_subbufs_consumed(). Any subsequent attempts to write into the buffer will again invoke the subbuf_start() callback with the same parameters; only when the consumer has consumed one or more of the ready sub-buffers will relay_buf_full() return 0, in which case the buffer switch can continue.

The implementation of the subbuf_start() callback for 'overwrite' mode would be very similar:

```
static int subbuf_start(struct rchan_buf *buf,
                        void *subbuf,
                        void *prev_subbuf,
                        unsigned int prev_padding)
{
        if (prev_subbuf)
                *((unsigned *)prev_subbuf) = prev_padding;

        subbuf_start_reserve(buf, sizeof(unsigned int));

        return 1;
}
```

In this case, the relay_buf_full() check is meaningless and the callback always returns 1, causing the buffer switch to occur unconditionally. It's also meaningless for the client to use the relay_subbufs_consumed() function in this mode, as it's never consulted.

The default subbuf_start() implementation, used if the client doesn't define any callbacks, or doesn't define the subbuf_start() callback, implements the simplest possible 'no-overwrite' mode, i.e. it does nothing but return 0.

Header information can be reserved at the beginning of each sub-buffer by calling the subbuf_start_reserve() helper function from within the subbuf_start() callback. This reserved area can be used to store whatever information the client wants. In the example above, room is reserved in each sub-buffer to store the padding count for that sub-buffer. This is filled in for the previous sub-buffer in the subbuf_start() implementation; the padding value for the previous sub-buffer is passed into the subbuf_start() callback along with a pointer to the previous sub-buffer, since the padding value isn't known until a sub-buffer is filled. The subbuf_start() callback is also called for the first sub-buffer when the channel is opened, to give the client a chance to reserve space in it. In this case the previous sub-buffer pointer passed into the callback will be NULL, so the client should check the value of the prev_subbuf pointer before writing into the previous sub-buffer.

Writing to a channel

Kernel clients write data into the current cpu's channel buffer using relay_write() or __relay_write(). relay_write() is the main logging function - it uses local_irqsave() to protect the buffer and should be used if you might be logging from interrupt context. If you know you'll never be logging from interrupt context, you can use __relay_write(), which only disables preemption. These functions don't return a value, so you can't determine whether or not they failed - the assumption is that you wouldn't want to check a return value in the fast logging path anyway, and that they'll always succeed unless the buffer is full and no-overwrite mode is being used, in which case you can detect a failed write in the subbuf_start() callback by calling the relay_buf_full() helper function.

relay_reserve() is used to reserve a slot in a channel buffer which can be written to later. This would typically be used in applications that need to write directly into a channel buffer without having to stage data in a temporary buffer beforehand. Because the actual write may not happen immediately after the slot is reserved, applications using relay_reserve() can keep a count of the number of bytes actually written, either in space reserved in the sub-buffers themselves or as a separate array. See the 'reserve' example in the relay-apps tarball at http://relayfs.sourceforge.net for an example of how this can be done. Because the write is under control of the client and is separated from the reserve, relay_reserve() doesn't protect the buffer at all - it's up to the client to provide the appropriate synchronization when using relay_reserve().

Closing a channel

The client calls relay_close() when it's finished using the channel. The channel and its associated buffers are destroyed when there are no longer any references to any of the channel buffers. relay_flush() forces a sub-buffer switch on all the channel buffers, and can be used to finalize and process the last sub-buffers before the channel is closed.

Misc

Some applications may want to keep a channel around and re-use it rather than open and close a new channel for each use. relay_reset() can be used for this purpose - it resets a channel to its initial state without reallocating channel buffer memory or destroying existing mappings. It should however only be called when it's safe to do so, i.e. when the channel isn't currently being written to.

Finally, there are a couple of utility callbacks that can be used for different purposes. buf_mapped() is called whenever a channel buffer is mmapped from user space and buf_unmapped() is called when it's unmapped. The client can use this notification to trigger actions within the kernel application, such as enabling/disabling logging to the channel.

Resources

For news, example code, mailing list, etc. see the relay interface homepage:

    http://relayfs.sourceforge.net

Credits

The ideas and specs for the relay interface came about as a result of discussions on tracing involving the following:

```
Michel Dagenais		<michel.dagenais@polymtl.ca>
Richard Moore		<richardj_moore@uk.ibm.com>
Bob Wisniewski		<bob@watson.ibm.com>
Karim Yaghmour		<karim@opersys.com>
Tom Zanussi		<zanussi@us.ibm.com>
```

Also thanks to Hubertus Franke for a lot of useful suggestions and bug reports.