

The seq_file interface

Copyright 2003 Jonathan Corbet <corbet@lwn.net>

This file is originally from the LWN.net Driver Porting series at <http://lwn.net/Articles/driver-porting/>

There are numerous ways for a device driver (or other kernel component) to provide information to the user or system administrator. One useful technique is the creation of virtual files, in debugfs, /proc or elsewhere. Virtual files can provide human-readable output that is easy to get at without any special utility programs; they can also make life easier for script writers. It is not surprising that the use of virtual files has grown over the years.

Creating those files correctly has always been a bit of a challenge, however. It is not that hard to make a virtual file which returns a string. But life gets trickier if the output is long - anything greater than an application is likely to read in a single operation. Handling multiple reads (and seeks) requires careful attention to the reader's position within the virtual file - that position is, likely as not, in the middle of a line of output. The kernel has traditionally had a number of implementations that got this wrong.

The 2.6 kernel contains a set of functions (implemented by Alexander Viro) which are designed to make it easy for virtual file creators to get it right.

The seq_file interface is available via <linux/seq_file.h>. There are three aspects to seq_file:

- * An iterator interface which lets a virtual file implementation step through the objects it is presenting.
- * Some utility functions for formatting objects for output without needing to worry about things like output buffers.
- * A set of canned file_operations which implement most operations on the virtual file.

We'll look at the seq_file interface via an extremely simple example: a loadable module which creates a file called /proc/sequence. The file, when read, simply produces a set of increasing integer values, one per line. The sequence will continue until the user loses patience and finds something better to do. The file is seekable, in that one can do something like the following:

```
dd if=/proc/sequence of=out1 count=1
dd if=/proc/sequence skip=1 of=out2 count=1
```

Then concatenate the output files out1 and out2 and get the right result. Yes, it is a thoroughly useless module, but the point is to show how the mechanism works without getting lost in other details. (Those wanting to see the full source for this module can find it at <http://lwn.net/Articles/22359/>).

The iterator interface

Modules implementing a virtual file with `seq_file` must implement a simple iterator object that allows stepping through the data of interest. Iterators must be able to move to a specific position – like the file they implement – but the interpretation of that position is up to the iterator itself. A `seq_file` implementation that is formatting firewall rules, for example, could interpret position *N* as the *N*th rule in the chain. Positioning can thus be done in whatever way makes the most sense for the generator of the data, which need not be aware of how a position translates to an offset in the virtual file. The one obvious exception is that a position of zero should indicate the beginning of the file.

The `/proc/sequence` iterator just uses the count of the next number it will output as its position.

Four functions must be implemented to make the iterator work. The first, called `start()` takes a position as an argument and returns an iterator which will start reading at that position. For our simple sequence example, the `start()` function looks like:

```
static void *ct_seq_start(struct seq_file *s, loff_t *pos)
{
    loff_t *spos = kmalloc(sizeof(loff_t), GFP_KERNEL);
    if (! spos)
        return NULL;
    *spos = *pos;
    return spos;
}
```

The entire data structure for this iterator is a single `loff_t` value holding the current position. There is no upper bound for the sequence iterator, but that will not be the case for most other `seq_file` implementations; in most cases the `start()` function should check for a “past end of file” condition and return `NULL` if need be.

For more complicated applications, the private field of the `seq_file` structure can be used. There is also a special value which can be returned by the `start()` function called `SEQ_START_TOKEN`; it can be used if you wish to instruct your `show()` function (described below) to print a header at the top of the output. `SEQ_START_TOKEN` should only be used if the offset is zero, however.

The next function to implement is called, amazingly, `next()`; its job is to move the iterator forward to the next position in the sequence. The example module can simply increment the position by one; more useful modules will do what is needed to step through some data structure. The `next()` function returns a new iterator, or `NULL` if the sequence is complete. Here’s the example version:

```
static void *ct_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
    loff_t *spos = v;
    *pos = ++*spos;
    return spos;
}
```

seq_file.txt

```
}
```

The `stop()` function is called when iteration is complete; its job, of course, is to clean up. If dynamic memory is allocated for the iterator, `stop()` is the place to free it.

```
static void ct_seq_stop(struct seq_file *s, void *v)
{
    kfree(v);
}
```

Finally, the `show()` function should format the object currently pointed to by the iterator for output. The example module's `show()` function is:

```
static int ct_seq_show(struct seq_file *s, void *v)
{
    loff_t *spos = v;
    seq_printf(s, "%lld\n", (long long)*spos);
    return 0;
}
```

If all is well, the `show()` function should return zero. A negative error code in the usual manner indicates that something went wrong; it will be passed back to user space. This function can also return `SEQ_SKIP`, which causes the current item to be skipped; if the `show()` function has already generated output before returning `SEQ_SKIP`, that output will be dropped.

We will look at `seq_printf()` in a moment. But first, the definition of the `seq_file` iterator is finished by creating a `seq_operations` structure with the four functions we have just defined:

```
static const struct seq_operations ct_seq_ops = {
    .start = ct_seq_start,
    .next  = ct_seq_next,
    .stop  = ct_seq_stop,
    .show  = ct_seq_show
};
```

This structure will be needed to tie our iterator to the `/proc` file in a little bit.

It's worth noting that the iterator value returned by `start()` and manipulated by the other functions is considered to be completely opaque by the `seq_file` code. It can thus be anything that is useful in stepping through the data to be output. Counters can be useful, but it could also be a direct pointer into an array or linked list. Anything goes, as long as the programmer is aware that things can happen between calls to the iterator function. However, the `seq_file` code (by design) will not sleep between the calls to `start()` and `stop()`, so holding a lock during that time is a reasonable thing to do. The `seq_file` code will also avoid taking any other locks while the iterator is active.

Formatted output

The `seq_file` code manages positioning within the output created by the

seq_file.txt

iterator and getting it into the user's buffer. But, for that to work, that output must be passed to the seq_file code. Some utility functions have been defined which make this task easy.

Most code will simply use seq_printf(), which works pretty much like printk(), but which requires the seq_file pointer as an argument. It is common to ignore the return value from seq_printf(), but a function producing complicated output may want to check that value and quit if something non-zero is returned; an error return means that the seq_file buffer has been filled and further output will be discarded.

For straight character output, the following functions may be used:

```
int seq_putc(struct seq_file *m, char c);
int seq_puts(struct seq_file *m, const char *s);
int seq_escape(struct seq_file *m, const char *s, const char *esc);
```

The first two output a single character and a string, just like one would expect. seq_escape() is like seq_puts(), except that any character in s which is in the string esc will be represented in octal form in the output.

There is also a pair of functions for printing filenames:

```
int seq_path(struct seq_file *m, struct path *path, char *esc);
int seq_path_root(struct seq_file *m, struct path *path,
                  struct path *root, char *esc)
```

Here, path indicates the file of interest, and esc is a set of characters which should be escaped in the output. A call to seq_path() will output the path relative to the current process's filesystem root. If a different root is desired, it can be used with seq_path_root(). Note that, if it turns out that path cannot be reached from root, the value of root will be changed in seq_file_root() to a root which *does* work.

Making it all work

So far, we have a nice set of functions which can produce output within the seq_file system, but we have not yet turned them into a file that a user can see. Creating a file within the kernel requires, of course, the creation of a set of file_operations which implement the operations on that file. The seq_file interface provides a set of canned operations which do most of the work. The virtual file author still must implement the open() method, however, to hook everything up. The open function is often a single line, as in the example module:

```
static int ct_open(struct inode *inode, struct file *file)
{
    return seq_open(file, &ct_seq_ops);
}
```

Here, the call to seq_open() takes the seq_operations structure we created before, and gets set up to iterate through the virtual file.

On a successful open, seq_open() stores the struct seq_file pointer in file->private_data. If you have an application where the same iterator can

seq_file.txt

be used for more than one file, you can store an arbitrary pointer in the private field of the seq_file structure; that value can then be retrieved by the iterator functions.

The other operations of interest - read(), llseek(), and release() - are all implemented by the seq_file code itself. So a virtual file's file_operations structure will look like:

```
static const struct file_operations ct_file_ops = {
    .owner    = THIS_MODULE,
    .open     = ct_open,
    .read     = seq_read,
    .llseek   = seq_llseek,
    .release  = seq_release
};
```

There is also a seq_release_private() which passes the contents of the seq_file private field to kfree() before releasing the structure.

The final step is the creation of the /proc file itself. In the example code, that is done in the initialization code in the usual way:

```
static int ct_init(void)
{
    struct proc_dir_entry *entry;

    proc_create("sequence", 0, NULL, &ct_file_ops);
    return 0;
}

module_init(ct_init);
```

And that is pretty much it.

seq_list

If your file will be iterating through a linked list, you may find these routines useful:

```
struct list_head *seq_list_start(struct list_head *head,
                                loff_t pos);
struct list_head *seq_list_start_head(struct list_head *head,
                                       loff_t pos);
struct list_head *seq_list_next(void *v, struct list_head *head,
                                loff_t *ppos);
```

These helpers will interpret pos as a position within the list and iterate accordingly. Your start() and next() functions need only invoke the seq_list_* helpers with a pointer to the appropriate list_head structure.

The extra-simple version

For extremely simple virtual files, there is an even easier interface. A module can define only the show() function, which should create all the

seq_file.txt

output that the virtual file will contain. The file's open() method then calls:

```
int single_open(struct file *file,
                int (*show)(struct seq_file *m, void *p),
                void *data);
```

When output time comes, the show() function will be called once. The data value given to single_open() can be found in the private field of the seq_file structure. When using single_open(), the programmer should use single_release() instead of seq_release() in the file_operations structure to avoid a memory leak.