RTLinux POSIX API for IO on Real-time FIFOs and Shared Memory

Cort Dougan Matt Sherer

Finite State Machine Labs
Socorro, NM 87801
{cort,sherer}@fsmlabs.com
`http://www2.fsmlabs.com/˜cort/`
`http://www2.fsmlabs.com/˜sherer/`

Abstract

The primary means of communication between RTLinux threads and Linux processes are RTLinux FIFOs. These have been limited by the fact that there was a static number of them and their names were restricted to `/dev/rtf0` through `/dev/rtf...`. Starting with RTLinux/Pro 1.2 and soon Open RTLinux, we have added a means for creating RTLinux FIFOs and shared memory with arbitrary names and locations that can be operated on with ordinary UNIX semantics.

1 Overview

In this paper we describe a new RTLinux POSIX-compliant API for creating arbitrarily named real-time FIFOs, performing asynchronous and synchronous IO on them and removing them. This new and completely POSIX compliant method for creating and operating on real-time FIFOs deprecates the old `/dev/rtf0` though `/dev/rtf...` method. The new method is simpler and more flexible. However, programs that rely on the older mechanism will continue to be supported and the `/dev/rtf*` entries will remain. Additionally, support for shared memory has been done in a POSIX-compliant method that also allows for arbitrary names and locations.

2 Creating a RTLinux FIFO

Creating and using a real-time FIFO is no different than doing the same under a UNIX system. Fig. 1 shows code that will create, open and then close a real-time FIFO. The same code compiles and runs properly under Linux as well as RTLinux. It is now possible to write, test and

```
int fifo(void)
{
    int fd;

    if ( (fd = mkfifo("/dir/myfifo", 0)) )
        return −1;

    if ( (fd = open("/dir/myfifo",
                O_RDONLY|O_NONBLOCK)) >= 0 )
        return −1;

    close(fd);
    return 0;
}
```

Figure 1: Creating a FIFO in RTLinux

debug code under Linux and then compile and run for RTLinux.

Fig. 1 shows code that will create a FIFO (of the default size) that is visible only to other RTLinux applications but not visible to Linux applications. These calls follow the expected semantics for the POSIX definition of them and those listed in common UNIX programming books[Stevens].

2.1 When is it safe?

In most cases `mkfifo()` is only safe to call from a Linux context. This would be in `init_module()` or `cleanup_module()`. Calling `mkfifo()` within a RTLinux thread is not safe unless buffer space has been pre-allocated for the FIFO. The `mkfifo()` call must allocate space for the FIFO and other record-keeping structures which cannot be done in a real-time context. So, Fig. 2 is not allowed and would be unsafe without pre-allocated space.

```
pthread_t thread;

void *thread_code(void *t)
{
        /* create a FIFO unsafely */
        fd = mkfifo("/home/filetest", 0);

        return NULL;
}

int init_module(void)
{
        pthread_create( &thread, NULL, thread_code, 0 );

        return 0;
}
```

Figure 2: Unsafe Creating a FIFO in RTLinux

Fig. 2 could be made safe by preallocating space for the FIFO. This can be turned on with the "*Preallocated fifo buffers*" option when configuring RTLinux. This will allow you to select the default size of the FIFOs and how many should be pre-allocated. Turning this option on does allocate space at run-time that may be unused so it is best to limit the number of allocated FIFOs to what you know you will use or avoid the need for pre-allocated FIFOs entirely. Most RTLinux applications know at initialization time what FIFOs will be needed and should create them then.

3 Deleting a FIFO

A well behaved program should remove any FIFOs that it has created. Since real-time FIFOs

```
int fifo(void)
{
    int ret;

    if ( (ret = mkfifo("/dir/myfifo", 0)) )
        return -1;

    if ( (ret = unlink("/dir/myfifo")) )
        return -1;

    return 0;
}
```

Figure 3: Removing a New Style FIFO

are not persistent on the RTLinux side they will not remain across reboots. However, they will remain between successive loads of a module. For example, if one loads and runs the program in Fig. 1 it will succeed the first time but fail every time after that. This is because the FIFO that was created with the `mkfifo()` call on the first run still exist so following `mkfifo()` calls will fail.

It's possible to remove the entry by hand, but there is a better way. Just as with any UNIX system removing the file is done with `unlink()`. Fig. 3 gives an example.

If a FIFO is removed with a call to `unlink()` while there are still open file descriptors on it (through calls to `open()`) the FIFO will not be removed until the last `close()` call completes. It is important that file descriptors are closed explicitly when they are not being used anymore since unloading a module does not cause an explicit `close()` call.

```
/* will succeed */
open("/dev/rtf0", O_CREAT|O_NONBLOCK);

/* will fail */
open("/dev/rtf0", O_NONBLOCK);
```

Figure 4: Creating an Old Style FIFO

4 Differences in open() and close()

Calls to `open()` and `close()` act differently when used on files created with `mkfifo()` than they do when called on the old FIFOs (`/dev/rtf*`). Fig. 4 shows both the correct and incorrect way to open an old style FIFO. The second version of the `open()` call will fail because it lacks `O_CREAT`. The `O_CREAT` flag in the call to `open()` will cause `open()` to create, allocate space for and open the FIFO.

Both calls in Fig. 5 will fail because no FIFO has been created. The `open()` call does not implicitly create a FIFO. This more closely matches POSIX and UNIX semantics. Instead, Fig. 6 will correctly create a FIFO and open it. The lack of `O_CREAT` in the `open()` call is important. If `O_CREAT` is used the `open()` call will fail, since the FIFO exists due to the call to `mkfifo()`.

Destroying a FIFO is similar. A call to `close()` on an old-style FIFO will cause the FIFO to be closed, destroyed and its resources released. A `close()` on a new-style FIFO will only close the FIFO but will not destroy it. Other threads can still operate on the FIFO or the same thread can re-open the FIFO and use it without another `mkfifo()` call. The only way

```
/* fails */
open("/file", O_CREAT|O_NONBLOCK);

/* also fails */
open("/file", O_NONBLOCK);
```

Figure 5: Fails to Create a New Style FIFO

```
mkfifo("/file", 0);

open("/file", O_NONBLOCK);
```

Figure 6: Create/Open a New Style FIFO

to destroy a new FIFO is through `unlink()`.

4.1 Calling Context

Since an `open()` of an old-style FIFO allocates space for and creates the FIFO it is only safe to call from a Linux context. Calls to `open()` on new-style FIFOs do not allocate any space and are free of that restriction. `open()` can be called from Linux and RTLinux contexts on new-style FIFOs. The same is true of calls to `close()`.

5 Making FIFOs visible to Linux

The call to `mkfifo()` in Fig. 6 uses 0 as the mode of the FIFO to be created. This indicates that the FIFO should not be visible to Linux processes. If the call is made with a non-zero argument as in Fig. 7 then a corresponding FIFO is created on the Linux side with the given permission bits and the same name.

```
/* create a FIFO that is visible to Linux */
mkfifo("/myfifo", 0755);
```

Figure 7: Creates a FIFO that is visible to Linux

There is no error indication if a file exists with the same name on the Linux side already. Instead, that file is destroyed and then recreated as a real-time FIFO device. **Be very careful when choosing names of RTLinux FIFOs and then advertising them to Linux since they can destroy files on the Linux side!**

When calling `mkfifo()` from a Linux context the create on the Linux side file is synchronous. That means when the `mkfifo()` call returns successfully you can be certain that the Linux side file has been created and is now visible to Linux user processes. Likewise, when `unlink()` returns successfully in a Linux context it is guaranteed that the file is no longer visible on the Linux side.

Calling `mkfifo()` from a RTLinux context with non-zero permission is not supported. Future versions of RTLinux may allow `mkfifo()` calls from RTLinux threads to create Linux-side FIFOs, though.

6 Changing RTLinux Filesystem Options

There are hard limits on the maximum number of open files, the maximum number of filesystem entries (existing files), maximum number of FIFOs and many other things. All of these are configurable. Understanding these configuration choices allows developers to make sure that no extra resources are consumed by the system and that applications will not fail due to too few resources being available at run-time.

The configuration option "`Number of Entries Created with Legacy rtl_register_rtldev call`" creates the variable `RTL_MAX_LEGACY_DEV`. This is the number of devices that are created by calls to `rtl_register_rtldev()`. The old FIFOs are created with a call to this function so this variable changes the number of old style FIFOs available as `/dev/rtf0` through `/dev/rtf...`. Other older drivers may also use this variable so does not only affect the number of old-style FIFOs available. Change it with care.

The maximum number of FIFOs available on the system is listed in the RTLinux configuration as "*Max number of fifos*" and represented by the value of `CONFIG_RTL_NFIFOS`. This is the maximum number of available FIFOs on the system. This value must be at least as great as `RTL_MAX_LEGACY_DEV`. The number of new-style FIFOs available is:

$$\texttt{CONFIG\_RTL\_NFIFOS} - \texttt{RTL\_MAX\_LEGACY\_DEV}$$

Changing the "*Max Number of Open Files*" configuration option will change how many files can be open at one time in RTLinux. This is the maximum number of file descriptors available to RTLinux threads. "*Max Number of Filesystem Entries*" is the maximum number of files that can be created. This value includes FIFOs created by calls to `rtl_register_rtldev()`, FIFOs created with calls to `mkfifo()` and any other special files created by drivers in RTLinux.

7 Asynchronous IO on FIFOs

The old non-POSIX method for installing FIFO read/write handlers is `rtf_create_handler()`. This method is supported for both old and new-style FIFOs but there is a better way that fits in with POSIX.

A common method for waiting for a FIFO to have data ready for read or write under UNIX is using `select()`. Calls to `select()` are blocking, so applications have to create a new threads for each FIFO handler. In addition to the extra resources being consumed by more threads performance is compromised since each `select()` event requires a reschedule. `rtf_create_handler()` installs handlers that are called directly rather than through scheduler redirection and this high-performance behavior is maintained for POSIX applications through `rtl_sigaction()`.

Applications can register handlers for FIFO read and write events through `rtl_sigaction()`. Fig. 8 lists code that creates a FIFO, opens it and then installs a handler to be called when reads and writes occur on it. The field `sa_fd` of the `rtl_sigaction` structure specifies which file descriptor the handler should be installed for. Even though an application may register many handlers for the `SIGPOLL` signal they can all be different if the `sa_fd` field is different. `SIGPOLL` signal handlers are defined by both the signal number `SIGPOLL` and the value of `sa_fd` when making calls `rtl_sigaction()`.

```
void fifo_handler(int sig, siginfo_t *sig, void *v)
{
        char msg[64];

        /* if there was a write at the other end... */
        if ( sig->si_code == POLL_OUT )
                write( sig->si_fd, &msg, 64 );
        /* if there was a read at the other end... */
        else if ( sig->si_code == POLL_IN )
                read( sig->si_fd, &msg, 64 );
}

void fifo(void)
{
        int fd;
        struct rtl_sigaction act;

        /* create a FIFO */
        mkfifo("/myfifo", 0755);

        /* open the FIFO for read/write */
        fd = open("/myfifo",
                   O_RDWR | O_NONBLOCK);

        /* register a SIGPOLL handler */
        rtl_sigaction.sa_sigaction = fifo_handler;
        /* the file that we want the signal for */
        rtl_sigaction.sa_fd = fd;
        /* we want read and write notification */
        rtl_sigaction.sa_flags = RTL_SA_RDWR |
                RTL_SA_SIGINFO;

        /* install the handler */
        rtl_sigaction( SIGPOLL, &act, NULL );
}
```

Figure 8: Registering a FIFO read/write handler

In order for a `rtl_sigaction()` call to succeed with `sa_flags` set to RTL_SA_RDWR the file descriptor that `sa_fd` is set to must be open for read and write (`open()` call with flags `O_RDWR`). The same is true for read-only (`RTL_SA_RDONLY`) and write-only (`RTL_SA_WRONLY`). A thread cannot register a `SIGPOLL` handler for an operation that would not be permitted on that file descriptor by normal IO operations such as `read()` or `write()`.

`RTL_SA_RDONLY`, `RTL_SA_WRONLY` and `RTL_SA_RDWR` are actually bitmasks and not distinct values as is UNIX tradition for flags to `open()`[Stevens] (`O_RDWR`, `O_RDONLY` and `O_WRONLY`). So, users should be careful of this when setting these values. Fig. 9 shows how to safely clear and set these values in the `sa_flags` member.

Note that in Fig. 8 writes to a FIFO that cause `fifo_handler()` to be invoked set the `si_fd` field to `RTL_POLL_OUT`. Conversely, reads from a FIFO that cause `fifo_handler()` to be called set `si_fd` to `RTL_POLL_IN`. The handler is called directly in response to the corresponding `read()` or `write()` operation and is called in the same context of those operations (including the active stack). `rtf_put()` and `rtf_get()` calls will cause any registered signal handlers for the corresponding FIFO to be called as well.

Fig. 10 shows a typical use of `rtl_sigaction()` to install handler for Linux-side writes to a FIFO. This example is an example of a RTLinux-side consumer of data that writes nothing back to Linux.

```
struct rtl_sigaction sig;

/* unsafe since sa_flags value is unknown */
sig.sa_flags |= RTL_SA_RDONLY;

/* set for read AND write */
sig.sa_flags |= RTL_SA_WRONLY;

/* safe way to set write-only */

/* turn off the RTL_SA_RDONLY and RTL_SA_WRONLY flags */
sig.sa_flags &= ~(RTL_SA_RDONLY|RTL_SA_WRONLY);

/* or... */
sig.sa_flags &= ~(RTL_SA_RDWR);

/* set write-only */
sig.sa_flags |= RTL_SA_WRONLY;
```

Figure 9: Operating on `sa_flags`

```
void fifo_handler(int sig, rtl_siginfo_t *sig, void *v)
{
        char msg[64];

        read( sig->si_fd, &msg, 64 );
}

void fifo(void)
{
        int fd;
        struct rtl_sigaction act;

        /* create a FIFO */
        mkfifo("/myfifo", 0755);

        /* open the FIFO for read */
        fd = open("/myfifo", O_RDONLY|O_NONBLOCK);

        /* register a SIGPOLL handler for the FIFO */
        sigaction.sa_sigaction = fifo_handler;
        /* the file that we want the signal for */
        sigaction.sa_fd = fd;
        /* we want write event notification */
        sigaction.sa_flags = RTL_SA_RDONLY | RTL_SA_SIGINFO;

        /* install the handler */
        rtl_sigaction( SIGPOLL, &act, NULL );
}
```

Figure 10: Registering a FIFO read handler

```
fd = shm_open("/shared_area", O_CREAT, 0);
```

Figure 11: Creating a shared area

```
ftruncate(fd, DESIRED_SIZE);
```

Figure 12: Allocating space for a shared area

7.1 Restrictions

There may be only one `SIGPOLL` handler installed for a given real-time FIFO at one time. There may be several file descriptors that refer to a single FIFO but they all share a single `SIGPOLL` handler. Installing a `SIGPOLL` handler for a given file descriptor may over-write one installed on the same FIFO through a different file descriptor.

8 Shared Memory

Support for shared memory has been handled in the GPL version of RTLinux using Tomasz Motylewski's mbuff driver for some time. RTLinux/Pro, however, uses common POSIX shared memory calls for manipulating these regions. Part of the POSIX realtime extensions call for `shm_open()` and `shm_unlink()`. We will demonstrate how these calls are used in RTLinux.

8.1 Creating a Shared Area

First, let's look at the process of creating a shared area with `shm_open()`. Fig. 11 demonstrates this call. Note that `shm_open()` creates an area and returns a file descriptor in one step, as opposed to the two involved in `mkfifo()` and `open()`.

The parameters for `shm_open()` are fairly straightforward: A name to be used for the device, open flags and mode bits. The open flags operate as expected - if `O_CREAT` is specified, it will attempt to create the area, otherwise it will return a descriptor to an already created area. Specifying `O_EXCL` with `O_CREAT` will cause a failure if the device has already been created.

The mode parameter operates exactly as with `mkfifo()` described earlier. Specifying 0 for the mode will create the device within the scope of RTLinux threads but will not make this object available to Linux. Specifying a non-zero value will cause RTLinux to create the device in the Linux filesystem and allow Linux file operations to use the device.

8.1.1 When is it safe?

`shm_open()` can be used from any context except when specifying that a Linux-side device is to be created. In this case, it must be done from within the `init_module()` context (exactly the same as with `mkfifo()` calls). However, we still need to allocate space for the region, as we will see in the next section. So generally speaking, the initial `shm_open()` occurs along with the allocating call during intialization, while further calls may happen at any time.

8.2 Sizing the shared region

As you have probably noticed, we have a file descriptor to a device that has been registered but

```
unsigned char *addr;

addr = mmap(0,DESIRED_SIZE,
        PROT_READ|PROT_WRITE,
        MAP_SHARED,fd,0);
```

Figure 13: Accessing a shared area

we haven't specified how much memory is to be used. As of this point, no memory has been allocated. Sizing of the region must be done with a call to `ftruncate()` as shown in Fig. 12. This operates on the file descriptor and takes a size parameter as normal with any `ftruncate()` call.

As this call causes memory allocation, it must be done in `init_module()` context. This is for initial creation of an area and also for resizing, as each case causes memory to be allocated or released.

8.2.1 Accessing the area

Now we've covered how to create an area and size it appropriately but how do we access it? This is done with the standard `mmap()` call on the file descriptor we received from `shm_open()`. Additionally, another thread can do a normal `open()` on the device created with `shm_open()`, and use `mmap()` on that file descriptor. Fig. 13 demonstrates common usage.

As you can see, this behaves the same way as a normal `mmap()` call. The protection flags are not used at this time between real-time threads, although specifying `PROT_READ|PROT_WRITE` is recommended. The address returned from this call can be used to reference the area as needed.

```
shm_unlink("/shared_area");
```

Figure 14: Destroying a shared area

Once the user is done with this area `close()` should be called on the file descriptor. This will leave the shared area valid, though - there is one more call to make it disappear from the filesystem.

8.3 Destroying a Shared Area

Closing all of the open file descriptors on a shared area will still leave the area present in memory. This means that a new thread can open the same area and access what was placed there previously. If this isn't necessary, destroying the area is preferred. Destroying shared memory is handled through `shm_unlink()`, as demonstrated in Fig. 14. There are no surprises here, as this simply frees the area. This can be done in any context.

8.4 Userspace Access to Shared Area

If a non-zero mode was passed to `shm_open()` the device will be visible to Linux. Accessing the shared area from Linux is very simple, as we can see in Fig. 15. Essentially, the operation is symmetric, as the user can just `open()` and `mmap()` as in RTLinux context.

```
int fd;
unsigned char *addr;

fd = open("/shared_area", O_RDWR);
addr = mmap(0, DESIRED_SIZE,
          PROT_READ|PROT_WRITE,
          MAP_SHARED, fd, 0);
/* work with the area */
close(fd);
```

Figure 15: Userspace access to shared memory

9 Reference Counting

As well as the POSIX additions detailed, all RTLinux devices are now reference counted. This simplifies operations so users don't have to worry about the state of the RTLinux thread when accessing a segment of shared memory from userspace. If all of the RTLinux threads have closed and exited, and a userspace process is the last holder of a device, the device remains usable until that last user releases it. This fits more closely with POSIX file operations.

As an example, consider the case where 2 RTLinux threads have initialized and are using a device for shared memory. If the userspace management system starts and attaches to that device, there are now three users. If the user decides that it is time to shut down, the RTLinux threads are directed to exit which causes them to `close()` and call `shm_unlink()` when the module is unloaded.

Now the device has been explicitly unlinked with `shm_unlink()`, but there is still a user. In our example, it is a userspace user, but it could just as easily be a RTLinux thread. Our application may still be analyzing the data held in that shared area. Once it has completed and exits, the usage count finally drops to 0, and the area is then destroyed.

This reference counting is used for all RTLinux devices, so this approach works the same for users of shared memory as it does for real-time FIFOs and for any other device driver written on top of RTLinux/Pro.

References

[Stevens] W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley Publishing Company, 1992.