Using Shared Memory in Real-Time Linux

Frederick M. Proctor

Overview of First-In-First-Out Queues (FIFOs) and Shared Memory

Communication between Real-Time Linux (RT-Linux) processes and Linux processes is usually accomplished using first-in-first-out (FIFO) connections, point-to-point queues of serial data analogous to Unix character devices. FIFOs have the following characteristics:

- FIFOs queue data, so no protocol is required to prevent data overwrites.
- Boundaries between successive data writes are not maintained. Applications must detect boundaries between data, particularly if data is of varying size.
- FIFOs support blocking for synchronization. Processes need not poll for the arrival of data.
- FIFOs are a point-to-point communication channel. FIFOs do not support the one writer and several readers model.
- The maximum number of FIFOs is declared in **rt_fifo_new.c** as:

  ```
  #define RTF_NO 64
  ```

  and these appear as devices **/dev/rtf0..63** in the filesystem. This limit can be changed and the **rt_fifo_new.o** module recompiled. There is no limit to the number of FIFOs an application can use, or to the size of data that can be written to a FIFO, other than practical memory limits.

An alternative to FIFOs is shared memory, in which a portion of physical memory is set aside for sharing between Linux and RT processes. Shared memory has the following characteristics:

- Shared memory does not queue data written to it. Applications requiring handshaking must define a protocol to assure data is not overwritten.
- Because data is not queued, individual items in data structures megabytes in size can be quickly updated.
- Shared memory has no point-to-point restriction. Shared memory can be written or read by any number of Linux or RT processes.
- The number of independent shared memory channels is only limited by the size of physical memory.
- Blocking for synchronization is not directly supported. To determine if data is new, the data must contain a count that can be compared against previous reads.
- Mutual exclusion of Linux and RT-Linux processes is not guaranteed. Interrupted reads and writes can be detected, however.

The decision to use FIFOs versus shared memory should be based on the natural communication model of the application. For control applications involving processes that execute cyclically based on the expiration of an interval timer, where data queueing is the exception rather than the rule, shared memory is a good choice for communication.

Setting up the Shared Memory Pool

The shared memory pool is a block of physical memory set aside at boot time so that Linux does not use it for processes. To set up the pool, determine how much physical memory the system has and how much is to be used for shared memory. Subtracting the size of the shared memory desired from the size of physical memory gives the absolute base address of the pool. The result is passed to the Linux boot loader (LILO) at boot time. This is accomplished by editing the **/etc/lilo.conf** and inserting a line with the **append** keyword.

For example, suppose the system has 32 MB of memory and 1 MB is to be used for the shared memory pool. The base address for shared memory is 32 MB - 1 MB, or 31 MB. Assuming the original **/etc/lilo.conf** file contained:

```
image=/boot/zImage
label=rtlinux-0.6
root=/dev/hda2
read-only
```

the file should be modified like this:

```
image=/boot/zImage
label=rtlinux-0.6
root=/dev/hda2
read-only
append="mem=31m"
```

Similarly, suppose the system has 16 MB of memory and 512 KB is to be used for the shared memory pool. The base address for shared memory is 16384 KB - 512 KB = 15872 KB. The **/etc/lilo.conf** file should be modified like this:

```
image=/boot/zImage
label=rtlinux-0.6
root=/dev/hda2
read-only
append="mem=15872k"
```

The size of the shared memory pool must be less than the pagesize declared in /usr/include/asm/param.h. In Intel-Pentium class machines and above, the page size is 4 MB. On earlier machines, the page size is 1 MB.

Addressing the Shared Memory Pool in C Language Programs

The base address of the shared memory pool needs to be declared in C so that both Linux and RT-Linux code can reference it. For example, shared memory based at 31 MB may be accessed using the C statement:

```
#define BASE_ADDRESS (31 * 0x100000)
```

Similarly, shared memory based at 15872 KB may be accessed using the C statement:

```
#define BASE_ADDRESS (15872 * 0x400)
```

This address is used differently in Linux and RT-Linux. Linux processes need to map this physical address into their virtual address space. RT-Linux processes can reference data located at this address as a pointer directly. This is detailed in later sections.

In addition to this declaration, the Linux and RT-Linux C code must agree on the data structures written into shared memory. For this discussion the following example structures will be used:

```
typedef struct
{
  unsigned char inuse;  /* more on this later */
  int command;
  int command_number;
  int arg1;
  int arg2;
} MY_COMMAND;

typedef struct
{
  unsigned char inuse;  /* more on this later */
  int command_echo;
  int command_number_echo;
  int stat1;
  int stat2;
} MY_STATUS;

typedef struct
{
  MY_COMMAND command;
  MY_STATUS status;
} MY_STRUCT;
```

A Linux process sends a command to an RT-Linux process by filling in the **MY_COMMAND** structure and writing it into shared memory. The RT-Linux process reads this structure from shared memory to get the command. An RT-Linux process sends status to a Linux process by filling in the **MY_STATUS** structure and writing it into shared memory. The Linux process reads this structure from shared memory to get the status.

The **MY_STRUCT** structure is a combination of both the command and status structure, and can be used to ensure that the two structures do not overlap and that their fields are aligned on the proper boundaries. It is also possible to define two base addresses, one for each structure, making sure the start of one structure is after the end of the previous one and that all fields are properly aligned. By combining them into a single aggregate structure and letting the compiler allocate storage, the structure will have a valid byte alignment automatically.

In a typical application the **BASE_ADDRESS** declaration and shared structure declarations shown above would be put in a header file shared by both Linux and RT-Linux code.

Accessing the Shared Memory Pool from Non-Realtime Linux

Normal Linux processes are required to map physical memory into their private address space to access it. To do this, the Linux processes calls **open()** on the memory device **/dev/mem**:

```
#include <unistd.h>             /* open() */
#include <fcntl.h>              /* O_RDWR */

int fd;

if ((fd = open("/dev/mem", O_RDWR)) < 0)
{
  /* handle error here */
}
```

Due to security reasons, the default permissions on **/dev/mem** allow only root processes to read or write **/dev/mem**. To access physical memory, the program must be run as root, its permissions must be changed to setuid root, or the permissions on **/dev/mem** must be changed to allow access to users other than root.

After the file descriptor is opened, the Linux process maps the shared memory into its address space using **mmap()**, as shown here:

```
#include <stdlib.h>             /* sizeof() */
#include <sys/mman.h>           /* mmap(), PROT_READ, MAP_FILE */
#define MAP_FAILED ((void *) -1)        /* omitted from Linux mman.h */
#include "myheader.h"

MY_STRUCT *ptr;

ptr = (MY_STRUCT *) mmap(0, sizeof(MY_STRUCT),
                         PROT_READ | PROT_WRITE,
                         MAP_FILE | MAP_SHARED,
                         fd, BASE_ADDRESS);

if (MAP_FAILED == ptr)
{
  /* handle error here */
}

close(fd);                      /* fd no longer needed */
```

BASE_ADDRESS is passed to **mmap()** which returns a pointer to the shared memory as mapped into the Linux process' address space. Once the shared memory is mapped, it may be accessed by dereferencing the pointer, for example:

```
ptr->command.arg1 = 1;
ptr->command.arg2 = 2;
```

When the process terminates, use **munmap()** to unmap the shared memory by passing the pointer and the size of its object:

```
#include <sys/mman.h>
#include "myheader.h"

munmap(ptr, sizeof(MY_STRUCT));
```

Accessing the Shared Memory Pool from RT-Linux

Shared memory access is much easier in RT-Linux since the RT code executes in kernel space and thus is not required to map physical addresses to virtual addresses. In Linux kernels 2.0.XX, the pointer can be set directly, for example:

```
#include "myheader.h"

MY_STRUCT *ptr;

ptr = (MY_STRUCT *) BASE_ADDRESS;

ptr->command.arg1 = 1;
ptr->command.arg2 = 2;
```

In Linux kernels 2.1.XX, the pointer needs to be mapped via a call to the **__**va() macro defined in /usr/include/asm/page.h, for example:

```
ptr = (MY_STRUCT *) __va(BASE_ADDRESS)
```

Detecting New Writes

FIFOs have an advantage over shared memory in that reads and writes follow standard Unix conventions. Non-realtime Linux processes can use **write()** to queue data onto a FIFO, and **read()** returns the number of characters read. Zero characters read from a FIFO means no new data was written since the last read. On the RT-Linux side, a handler is associated with a FIFO which is invoked after a non-realtime Linux process writes to the FIFO. Normally the handler calls **rtf_get()** to dequeue the data from the FIFO.

These functions are not necessary with shared memory. Reads and writes are accomplished by reading and writing directly to pointers. Consequently, the operating system provides no way to detect if the contents of shared memory have been updated. The programmer needs to set up this handshaking explicitly.

One way to do this is to use message identifiers that are incremented for each new message. The receiver then polls the shared memory buffer and compares the current identifier with the previous one. If they are different, a new message has been written. Handshaking to prevent message overrun is implemented by echoing message identifiers in the status structure once they have been received. New messages are not sent until the status echoes the message identifier.

This presumes a time-cyclic polling model on the part of both the Linux and RT-Linux processes. If this is not the natural model for the application, then shared memory may not be a good choice for communication. If shared memory is required for other reasons, and polling is not desirable, other synchronization between Linux and RT-Linux processes can be used. For example, RT-Linux FIFOs can be used only for their synchronization properties. A byte written to a FIFO can be used to wake up a Linux process blocked on a read, or to call the handler in an RT process.

Realizing Mutual Exclusion

It is possible (and therefore a certainty) for a Linux process to be interrupted by an RT process while in the middle of a read or write to memory they are sharing. If the Linux process is interrupted during a read, the Linux process will see stale data at the beginning and fresh data at the end. If the Linux process is interrupted during a write, the RT process will see fresh data at the beginning but stale data at the end. Both problems are fatal in general.

The problem of ensuring data consistency of data shared between two processes is the subject of operating systems research, and general solutions exist [1]. Our problem is simpler because only the Linux process can be interrupted. In no case can a Linux process interrupt an RT process during the execution of its task code.

This simplification means that an "in-use" flag can be used by the Linux process to signal that it is accessing the shared memory. The in-use flag is declared at the beginning of the shared memory structure, and is set by the Linux process when it wants to read or write and cleared when finished. The RT process checks the in-use flag before it accesses shared memory, and if set defers the read or write action until it detects that the flag is cleared.

This may lead to the indefinite postponement of the RT process, if the following conditions are true:

1. the RT process runs at the period of the Linux process, or at multiples of this period;
2. the accesses are synchronized so that the RT process always interrupts the Linux process during the critical section when it has set the in-use flag.

The first condition implies that the RT code is running as slow or slower than the Linux code, and the second implies that the Linux code is running as deterministically as the RT code. Neither is typically true, and both are rarely true at the same time. If these conditions are true for a system, successive deferrals can be detected by the RT code and can trigger actions to keep the system under control.

The following example illustrates the application of the in-use flag for commands written by the Linux process to the RT process, and status written by the RT process and read by the Linux process.

```
typedef struct
{
  unsigned char inuse;
  int command;
  int command_number;
  int arg1;
  int arg2;
} MY_COMMAND;

typedef struct
{
  unsigned char inuse;
  int command_echo;
  int command_number_echo;
  int stat1;
  int stat2;
} MY_STATUS;
```

Assume that **command_ptr** has been set up in a Linux process to point to the shared memory area for commands to the RT process. To write to shared memory, compose the command, set the **inuse** flag in shared memory directly, write the command, and reset the **inuse** flag:

```
MY_COMMAND my_command;
```

```
/* compose command in local structure */
my_command.inuse = 1;  /* will overwrite during copy, so set here too */
my_command.command = 123;
my_command.command_number++;
my_command.arg1 = 2;
my_command.arg2 = 3;

/* set inuse flag */
command_ptr->inuse = 1;

/* copy local structure to shared memory */
memcpy(command_ptr, &my_command, sizeof(MY_COMMAND));

/* clear inuse flag */
command_ptr->inuse = 0;
```

Assuming that the RT code has set **command_ptr** to point to the shared memory for commands, it would read commands like this:

```
if (0 != command_ptr->inuse)
{
  /* ignore it, perhaps incrementing a deferral count */
}
else
{
  /* okay to access shared memory */
}
```

To read status information, the Linux process sets the **inuse** flag before copying out the data. Assuming that **status_ptr** has been set up in a Linux process to point to the shared memory area for status from the RT process, this would look like:

```
MY_STATUS my_status;

/* set inuse flag */
status_ptr->inuse = 1;

/* copy shared memory to local structure */
memcpy(&my_status, status_ptr, sizeof(MY_STATUS));

/* clear inuse flag */
status_ptr->inuse = 0;

/* refer to local struct from now on */
if (my_status.stat1 == 1)
{
  ...
}
```

When writing status, the RT process checks for the in-use flag and defers a status write if it is set. Assuming that the RT code has set **status_ptr** to point to the shared memory for status, this would look like:

```
if (0 != status_ptr->inuse)
{
  /* defer status write, perhaps incrementing deferral count */
}
else
```

```
{
  /* okay to write status */
}
```

Queueing Data in Shared Memory using Ring Buffers

While shared memory is most naturally suited for communications in which data overwrites the previous contents, queueing can be set up using ring buffers. Ring buffers queue 0 or more instances of a data structure, up to a predetermined maximum.

To illustrate the use of ring buffers, consider a system that queues error messages. Errors are declared as strings of a fixed maximum length, and there is a fixed maximum number of errors that can be queued. This is implemented as a two-dimensional array:

```
#define ERROR_NUM 64  /* max number of error strings to be queued */
#define ERROR_LEN 256  /* max string length for an error */
char error[ERROR_NUM][ERROR_LEN];
```

Supplementing the actual list of errors are indices to the start and end of the queue, which wraps around from the end of the shared memory area to the beginning (hence the name "ring buffer"), and a count of the errors queued. As described above, an in-use flag is also declared to signal that a Linux process is accessing the ring buffer to prevent data inconsistencies in the event an RT process interrupts Linux process access. The full shared memory structure declaration is then:

```
#define ERROR_NUM 64  /* max number of error strings to be queued */
#define ERROR_LEN 256  /* max string length for an error */

typedef struct
{
  unsigned char inuse;           /* flag signifying Linux accessing */
  char error[ERROR_NUM][ERROR_LEN]; /* the errors themselves */
  int start;                     /* index of oldest error */
  int end;                       /* index of newest error */
  int num;                       /* number of items */
} MY_ERROR;
```

Both Linux and RT-Linux use the same access functions. However, Linux processes need to set the in-use flag before getting an error off the ring, and RT processes need to check the in-use flag and defer access until the flag is zero. Assuming that **errlog** is a pointer to the shared memory area for both Linux and RT processes, the access functions look like this:

```
/* initialize ring buffer; done once, perhaps in init_module() */
int error_init(MY_ERROR *errlog)
{
  errlog->inuse = 0;
  errlog->start = 0;
  errlog->end = 0;
  errlog->num = 0;

  return 0;
}

/* queue an error at the end */
int error_put(MY_ERROR *errlog, const char *error)
{
```

```
  if (errlog->num == ERROR_NUM)
    {
      /* full */
      return -1;
    }

  strncpy(errlog->error[errlog->end], error, ERROR_LEN);
  errlog->end = (errlog->end + 1) % ERROR_NUM;
  errlog->num++;

  return 0;
}

/* dequeue the error off the front */
int error_get(MY_ERROR *errlog, char *error)
{
  if (errlog->num == 0)
    {
      /* empty */
      return -1;
    }

  strncpy(error, errlog->error[errlog->start], ERROR_LEN);
  errlog->start = (errlog->start + 1) % ERROR_NUM;
  errlog->num--;

  return 0;
}
```

For Linux, getting an error off the ring buffer is accomplished by:

```
char error[ERROR_LEN];  /* place to copy error */

/* set in-use flag in shared memory */
errlog->inuse = 1;

/* copy error out */
if (0 != error_get(errlog, error))
{
  /* empty */
}
else
{
  /* handle it */
}

/* clear in-use flag in shared memory */
errlog->inuse = 0;
```

For an RT process, writing an error to the ring looks like:

```
char error[ERROR_LEN];  /* place to compose error */

/* check for in-use */
if (0 != errlog->inuse)
{
  /* defer writing, perhaps incrementing deferral count */
}
else
{
```

```
    /* compose it */
    strcpy(error, "your error here");
    if (0 != error_put(errlog, error))
    {
      /* full */
    }
  }
```

Sample Code

A sample application illustrating Linux to RT commands, RT to Linux status, and RT to Linux error logging is provided as a gzip'ed tar file, shmex.tgz. Unpack and compile with

```
tar xzvf shmex.tgz
make
```

Note that you have to set up Linux to boot with shared memory set aside, as detailed above.

References

1. Harvey M. Dietel, An Introduction to Operating Systems, Second Edition, Addison-Wesley, 1990, pp. 75-87.

Thanks

Thanks to everyone who contributed to this writeup. You are:

- Rich Bowser, bowser@luz.cs.nmt.edu, for information on the 4 MB memory chunking and the problems with 486 systems.
- Albert D. Cahalan, acahalan@cs.uml.edu, for help with memory page and mmap() information.
- Daniele Lugli, danlugli@tin.it, for review and comments.

Disclaimer