by Bob Japenga (USA)

Concurrency in Embedded Systems (Part 6)

Introducing Linux and Concurrency

This is the sixth article in a multi-part series focused on concurrency in embedded systems. This article discusses two additional mechanisms Linux provides for creating robust embedded systems.

Over the past few months, we have been looking at some mechanisms embedded systems designers can utilize when using embedded Linux. Specifically, we have examined the mechanisms that aid us in creating systems with concurrency. Part 4 of this article series discussed the various multi-tasking options (i.e., threads and processes) that can be used to create concurrency. Part 5 provided details on how shared memory, semaphores, and mutexes help us communicate between concurrent processes. If you have been following this series, you'll know that we prefer using the POSIX standard mechanisms to implement our systems so we can easily port our design to another operating system (OS) should the need arise. Thus, all the mechanisms we have examined i this series have been the POSIX-compliant versions.

Recently at work, this cherished value of using standards-based mechanisms was severely challenged. We were creating some Java native interfaces (JNIs) for a new hardware design that would run both Linux and Android. As prime candidates for the "Android for Dummies" books, we proceeded with our POSIX-compliant design "unencumbered by the thought process." In particular, our design used POSIX message queues. Everything came up fine under Linux and things were proceeding on schedule. Then we started testing under Android. Oh, did I forget to mention Android does not support POSIX message queues? But isn't Android built on Linux? Aren't message queues implemented in the kernel? Yes, but…

After much pain and grief, we switched our interface from POSIX message queues to POSIX first in, first outs (FIFOs), which are supported under Android. So, it seems appropriate for me to address these two mechanisms available for inter-process communication (IPC) under Linux in this month's article.

POSIX FIFOS

A FIFO provides a unidirectional inter-process communication (IPC) that has a read end and a write end. Both hardware and software designers should be familiar with FIFOs. Basically, if you put the letters A, B, and C into a FIFO, they will come out in the order they were put in (A, B, C). UARTs are the most common hardware devices that can contain FIFOs. If 16 characters come into the FIFO over the serial port, they come out in the order they went in.

Under Linux, you can create a FIFO and use it just like a hardware FIFO. If you put 16 bytes into the FIFO, you pull the bytes out in the order they were put in. Once you have pulled the data out of the FIFO, it no longer exists in the FIFO.

After you create the FIFO (using the `mkfifo` function), you can open it exactly like a file. All the standard attributes of files now apply to your FIFO (e.g., permissions, read only, etc). Under conventional operation, one process will open the FIFO for writing and one will open it for reading. The default operation forces both the reader and the writer to block until the other task opens its end of the FIFO. This can be overridden by using the `O_NONBLOCK` flag with slightly different results for the reader and the writer. If the reader opens the FIFO with and the FIFO has not been opened for writing, the open will not block. If the writer uses `O_NONBLOCK` to open the FIFO and the FIFO has not been opened for reading, the open will fail. Table 1 provides a

FIFOs	Regular Files
Once the data is read out of the FIFO, the data is no longer available	Data can be read anywhere in the file using lseek
Under default operation, both the writer and reader of the FIFO will block until the other end is opened (as described above, the O_NONBLOCK can alter that behavior).	Opens never block
The FIFO data is volatile (deleted at power up)	File data volatility depends on the underlying file system
Even though the FIFO has a pathname in the file system, the data put into a FIFO does not affect the underlying file system. This means that you don't need to place the FIFO on your RAM drive rather than your flash file system to minimize flash usage. Even if you place the FIFO in the path on the FLASH file system, the name and the data is stored in volatile RAM. Another way to say this is that all reads and writes to the FIFO are passed internally in the kernel and do not affect the file system.	Performance of read/writes is dependent upon the underlying file system. Flash file systems have limited write cycles.
All data disappears if both reader and writer close the file	Data does not go away when the file is closed.
The amount of data that a FIFO can contain is limited by a kernel parameter (FIFO_SIZE) and is set to 65536 for Linux 2.6.11 and greater.	Data size is limited only by the size of the file system.
All writes of less than or equal to PIPE_BUF (4096 for most Linux systems but the actual size varies on different kernels) are atomic. POSIX requires PIPE_BUF to be at least 512 so that is what we use in our designs.	Data writes are not atomic across processes.

Table 1—There are many differences between FIFOs and regular files.

summary of the differences between POSIX FIFOs and regular files.

DEBUGGING FIFOS

Since FIFOs look like files on the file system, the data being put in and taken out can be viewed from the command line (e.g., using `hexdump` or `cat`). This could be useful during debugging or even remotely should a released piece of code contain problems (which of course, your code never does!).

POSIX MESSAGE QUEUES

Linux provides both System V message queues and POSIX message queues. As we have previously discussed, our thin slice will limit my discussion to POSIX message queues.

When we discovered that Android did not support POSIX message queues, we chose to port this API with a POSIX FIFO. We determined that FIFOs were the closest IPC available under Android to the message queue for our purposes. One major thing we lost was that message queues allow for variable-length packetized data. In other words, the writer function would write a variable-length message into the queue and the reader would read that variable-length message out as a complete packet. FIFOs are byte streams and do not make any distinction concerning message boundaries. Thus, on the reader side, we did not know how many bytes to read out of the FIFO. We needed to use a terminator to determine the message size. And that is a big reason to use message queues instead of FIFOs in your designs.

POSIX message queues were not available under Linux prior to 2.6.6. Unlike FIFOs, which use standard `open`, `write`, `read`, and `unlink`, POSIX message queues are opened with `mq_open`, written to with `mq_send`, read with `mq_receive`, and unlinked with `mq_unlink`.

MESSAGE QUEUE ATTRIBUTES

POSIX message queues have the following settable attributes: the maximum number of messages you would allow and the maximum message size (in bytes). These values are set when the queue is created and cannot be changed after that. In addition, the programmer can obtain the number of messages currently in the queue and know whether or not the queue was created for blocking or nonblocking I/O (which can be changed after queue is created).

MESSAGE NOTIFICATION

POSIX message queues enable a reader process to be asynchronously notified when a previously empty queue is no longer empty. Thus, your process does not need to block waiting on a `mq_receive` but can perform other functions while waiting for the message. Your process can be notified by the standard signal mechanism in Linux (a topic for another day) when a message comes in. It does this through what is called registration. Message notification is another major advantage of queues versus FIFOs.

There are some things to keep in mind about message notification. A process is only notified if a message comes in after it registers to be notified. This means that if there are messages in the queue when a process registers, it will not be notified of that existing message.

Only one process can be notified upon the arrival of a message in the message queue. You should use blocking if your design requires two listeners. To enable multiple listeners, each process must re-register after each notification.

MESSAGE QUEUES LIMITATIONS

Just as all OSes provide some limit to the number of open file handles, Linux limits the number of message queues. This number is 256 by default but can be dynamically adjusted up to a maximum of `INT_MAX`. This number is architecture dependent but never less than 4 billion. If you are creating a system with more than 4 billion message queues, you are designing systems far more complex than I can handle.

The maximum message size is also defined for Linux as 8,192 by default, but this can be dynamically adjusted up to a maximum of 1,048,576. Linux also provides the programmer a way to limit the number of bytes that all queues for a particular user can use. This is a helpful check to protect your system against some runaway process that is putting data into the queues but never taking them out.

DEBUGGING MESSAGE QUEUES

Although the message queues are internal to the kernel, many message queue parameters are accessible from the command line. We have had systems that have exhibited aberrant behavior (must have been hardware problems!) and we were able to use this feature to remotely determine what was going on in the message queues. To access the queue, you need to first create a mount point for the queue:

```
mkdir /dev/my_queue
```

then mount the queue with the command:

```
mount -t mqueue none /dev/my_queue
```

At that point, you can find out the queue size, the signals used to notify processes, and you can even peek into the data. In addition, you can tell the message queue to send a notification to the process waiting for a signal from a particular message.

WHEN TO USE FIFOS OR MESSAGE QUEUES

I would say (unless you are developing an Android-native interface) that message queues are the preferred method since they are more flexible and are not significantly more complex to use than FIFOs. If you need the ability to handle variable-length data and require notification rather than blocking, message queues win hands down. We have never noticed any appreciable difference in the resource utilization (real time and memory) between the two. In other words, we have not noticed that FIFOs are significantly faster than message queues, nor have we seen significant differences in memory usage (although we suspect that FIFOs are slightly faster and use less memory). So the only time I would recommend using FIFOs instead of message queues for IPC would be if the slight overhead advantage makes a difference.

THE COMPLEXITY OF EMBEDDED SYSTEMS

Creating embedded systems is a complex task and is getting more complex every day. The embedded design community is creating some pretty amazing products these days because of tools like Linux. POSIX FIFOs and message queues are two tools in your arsenal when you choose Linux. In my next article, we will continue looking at using Linux to create your own amazing products.

Bob Japenga has been designing embedded systems since 1973. In 1988, along with his best friend, he started MicroTools, which specializes in creating a variety of real-time embedded systems. With a combined embedded systems experience base of more than 200 years, they love to tackle impossible problems together. Bob has been awarded 11 patents in many areas of embedded systems and motion control. You can reach him at rjapenga@microtoolsinc.com.