by Bob Japenga (USA)

Concurrency in Embedded Systems (Part 5)

Designing Robust Systems with Linux

This is the fifth in a multi-part article series examining concurrency in embedded systems. This article discusses some other ways embedded Linux helps you design robust systems with concurrency

Part 4 of this article series discussed Linux threads and processes. This article will focus on the first forms of inter-process communication (IPC): mutexs, semaphores, and shared memory. As I mentioned last month, remember, we are taking this in thin slices. The goal of this article series is to introduce you to the range of tools available in Linux to assist you in designing robust embedded systems with Linux. In my last article, I mentioned Michael Kerrisk's Michael Kerrisk's *The Linux Programming Interface*. This month, I want to connect you to the POSIX standard webpage to find more about all types of IPC. (See the Resources section for more information about the POSIX standard for shared memory.)

SHARED MEMORY

In my last article, we looked at the memory model Linux uses and how memory is *not* shared between processes by design. Thus, one process cannot share data with another without some system mechanism. Linux provides three mechanisms for designating certain memory as shared and enables you to access that memory: the historical System V UNIX shared-memory model, shared file mapping supported by POSIX, and the POSIX shared-memory model. This article will examine the two POSIX mechanisms. Generally, we don't recommend using the legacy interfaces when a POSIX interface is available.

Functionally, these two mechanisms are very similar. Since POSIX uses the term "object" to refer to a memory region that can be shared between processes, for purposes of this article, I will call files created using shared file mapping "nonvolatile shared-memory objects" and the standard shared-memory objects "volatile shared-memory objects." Let's look at each of these two mechanisms.

VOLATILE VERSUS NONVOLATILE SHARED-MEMORY OBJECTS

In simple terms, both methods enable different processes to directly access shared memory without system calls (i.e., fast). The nonvolatile mechanism enables the memory to be "sticky" across power outages. When power is lost, a volatile shared-memory object's contents are lost. The caveat with nonvolatile objects for real-time embedded systems designers is the question of when the nonvolatile object's data is committed to the underlying file. The kernel will commit the data to file at its leisure. As a designer, POSIX does enable us to commit on demand (using `msync()`), which is a mixed blessing. It frees the designer to access nonvolatile memory across processes but places the burden on the designer to know when to commit the mapped memory to disk.

Unfortunately, the APIs for using these two mechanisms are not identical. In my opinion, there should be a flag that makes a shared-memory object nonvolatile so they can be interchangeably used by the designer. Very often, we design systems where certain data's nonvolatility requirements changes over the product's life cycle. But these differences persist and we must work around them.

CONFIGURING VOLATILE SHARED-MEMORY OBJECTS

Shared-memory objects can be thought of as

volatile files (or files kept on a RAM drive) keeping with the Linux virtual memory paradigm. To configure a volatile shared-memory object, the programmer must perform three steps. The first step is to create a shared object much like you create a file. You select an object name then you open it with the name, flags, and a mode just like you open a file to create it. The object can have read-only or read-write access based on the flags. The POSIX system call `shm_open()` is used to perform this function. It returns a volatile shared-memory object handle. The second step is to use `ftruncate()` to set the object's size. Just like a file, the size can be later expanded or contracted. The final step is use the `mmap()` system call to map all or part of the object into the process' virtual memory. Optionally, the file handle can be closed after the `mmap` call.

Listing 1—This is an example of semaphore code

```
// globals
sem_t semi;
unsigned long long counter; /* shared object */

int main()
{
  pthread_t counter_thread;
  pthread_t reader_thread;

  sem_init(&semi, 0, 1); // semaphore is "named" semi
                         // semi is local
                         //initialize semi to 1

  pthread_create (&counter_thread,
                  NULL,
                  (void *) &incrementer,
                  NULL);
  pthread_create (&reader_thread,
                  NULL,
                  (void *) &reader,
                  NULL);

  while (1 == 1)
  {
    sleep (10);
  }
}
void reader ( void * )
{
  while (1 == 1)
  {
    sem_wait(&semi);        // decrement
    // start of critical section
    printf("Current Counter = %llu\n",counter);
    // end of critical section
    sem_post(&semi);        // increment
    sleep(1);
  }
}
void incrementer ( void * )
{
  while (1 == 1)
  {
    sem_wait(&semi);        // decrement
    // start of critical section
    counter++;
    // end of critical section
    sem_post(&semi);        // increment
    nanosleep(100000); // Sleep for 100 microseconds
  }
}
```

CONFIGURING NONVOLATILE SHARED-MEMORY OBJECTS

Similar steps are performed to create a nonvolatile shared-memory object. In this case, you can use the standard `open()` commands (instead of `shm_open`) to create a standard file. Then, like a volatile object, you use the `mmap()` system call to map all or part of the file's contents to a memory region. The `mmap` call sets the size of the file. As with the volatile object, the file can be closed at this point.

USING SHARED MEMORY

Once the objects are created using one of these two methods, both object types can be identically used by referencing them as memory. The objects look like any other memory object in the process' virtual address space.

PROS & CONS OF USING SHARED-MEMORY OBJECTS

Although shared memory is the fastest means of sharing memory across processes, there is still measurable overhead. As mentioned in my last article, we measured 50 µs per access to a nonvolatile shared-memory object on one 600-mHz ARM9. If you are going to have significant shared memory between concurrent tasks, you would be better off doing this between threads and eliminating the overhead.

Also, remember, just because the OS provides this feature, it does not provide synchronized access to the shared region. You must use something like a semaphore or mutex to guarantee the data's integrity in these shared regions.

SEMAPHORES

As with shared memory, Linux provides both System V semaphores and POSIX semaphores. As before, we will examine only POSIX semaphores.

POSIX defines two types of semaphores: named and unnamed. Named semaphores are handled between processes with a `sem_open()` system call. Unnamed semaphores share a memory address. With processes, this can be accomplished with a volatile shared-memory object. With threads, this can be accomplished with a global variable. The methods of initializing and destroying these two types differ, but everything else is operationally the same.

HOW SEMAPHORES WORK

A semaphore is a simple integer that can be incremented and decremented but can never fall below zero. The `sem_post()` function increments while the `sem_wait()` function decrements the semaphore. If the integer is greater than 0, the `sem_wait` immediately returns. If the integer is 0, the task will block it until the semaphore rises above zero then it will decrement the integer.

USING SEMAPHORES IN CONCURRENT DESIGNS

Let's look at a simple example using unnamed semaphores to illustrate how you would use a semaphore to provide the nec-

essary synchronization with shared-memory objects (in this case shared between threads).

Assume you have a 64-bit counter that is incremented by one thread, read by another, and is stored in nonvolatile shared memory. Assume the underlying hardware only supports 32-bit integer arithmetic. (Note: We are not building this robust, we are not checking the `sem_init` or `pthread_create` returns to keep the example simple.)

The Linux C code segment in Listing 1 shows how this works. The main process creates two threads and a semaphore and then goes to sleep. The counter thread increments a 64-bit counter. Remember, since this is a nonatomic operation, without the semaphore, it could be interrupted midstream and produce incoherent results in the reader thread. The reader thread merely prints the counter value to the standard output device.

With the semaphore in place, the reader thread will always print coherent data. If the reader attempts to print the counter while the counter is being incremented, the semaphore will be set to 0 and the reader thread will block at the `sem_wait()` until the `incrementer` function has updated counter.

Two other nice enhancements for the embedded designer are the `sem_trywait()` and the `sem_timedwait()`, which do exactly what you'd expect them to do. With the `sem_timedwait`, you can protect your thread against an unruly thread and not wait forever for the semaphore. You can use the `sem_trywait()` to control the blocking.

MUTEXES

POSIX supports mutexes (which stands for mutual exclusivity), which is another mechanism for providing synchronization for concurrency. Like semaphores, mutexes can ensure atomic access to any shared resource. Unlike semaphores, which can have an integer number of values, a mutex, by default, is binary in nature and can only be locked or unlocked. Let's see how you create and use mutexes.

CREATING AND USING MUTEXES

A default style mutex is simply made by creating a variable of type `pthread_mutex` and initializing it with `PTHREAD_MUTEX_INITIALIZER`. It can be as simple as:

```
pthread_mutex_t MyMutex = PTHREAD_MUTEX_INITIALIZER;
```

A mutex can then be locked (`pthread_mutex_lock`) and unlocked (`pthread_mutex_unlock`) to achieve the necessary synchronization. Hopefully, you can see that these could be used in the previous code example by replacing the `sem_wait` with the `pthread_mutex_lock` and the `sem_post` with the `pthread_mutex_unlock`.

POSIX SIDEBAR

We haven't talked about portable operating system interface (POSIX) in this column and it is worthy of a column all by itself, but for now, suffice it to say that it defines an operating system (OS) that provides a standard interface for applications written across multiple OSes. The POSIX standard was formalized by the IEEE and is also known as IEEE-STD 1003. Theoretically, you could write a POSIX-compliant application for Linux and then switch to QNX or VxWorks with no change to your code. Let me know if you would like an article or two on POSIX.

Listing 2—When using a mutex, nested layers of critical code sections are unnecessary.

```
    HighLevelFunction()
    {
      // Do some work
     // Enter Critical Section with Semaphore (sem_wait)
or Mutex lock
      AccessSharedResource();
      // Do some work
      LowLevelFunction();
       // Exit Critical Section with Semaphore post
(sem_post) or Mutex unlock
      // Do some work
    }

    LowLevelFunction()
    {
      // Do some work
     // Enter Critical Section with Semaphore (sem_wait)
or Mutex lock
      AccessSharedResource();
       // Exit Critical Section with Semaphore post
(sem_post) or Mutex unlock
      // Do some work
    }
```

WHERE MUTEXES WORK BETTER THAN SEMAPHORES

There are at least two cases where mutexes work better than semaphores. Many times, we create several functions in a concurrent design that access the same shared resources. With a mutex, you need not worry about nested layers of critical sections of code. Listing 2 illustrates how this works.

With a semaphore, you would be locked out forever in the low-level function when called by the high-level function. With the mutex, you have options to easily solve this. Instead of using the default behavior, POSIX has two other types of mutexes. For PTHREAD_MUTEX_ERRORCHECK mutexes, the lock would fail and you would know you need not unlock the mutex. For PTHREAD_MUTEX_RECURSIVE mutexes, the lock maintains a counter so you can enter a critical section a second time with no problems (as in our low-level function). Unlocking merely decrements the counter and doesn't actually unlock the mutex.

The second advantage of a mutex over a semaphore is that the mutex can only be unlocked by the thread with which it was locked.

A way to remember this feature is not to think of lock and unlock but owned and available. If it is owned (i.e., locked), only the owner can make it available (i.e., unlocked). This feature helps create a much more robust design because you are forced to think in a more structured manner.

WHERE SEMAPHORES WORK BETTER THAN

MUTEXES

If you have a resource such as a finite number of open-file handles and you want to gracefully handle times when you run out of file handles (not just abort your program), semaphores do the trick. These file handles are allocated and dynamically used for a time when the file is open and released when closed. In this case, a semaphore works better than a mutex. If you initialize the semaphore to the number of open file handles when it is created, your thread can simply perform a `sem_wait` when it wants to obtain a file handle. If there is at least one available, the code does not block. If all of the open file handles are exhausted, the code will block until one becomes available. This is much more robust than aborting the program under this condition.

ALTERNATIVE OPTIONS

All locks and semaphores incur a resource penalty. However, understanding these options is critical in designing embedded systems with concurrency. If you are fortunate enough to be using a toolchain that supports the latest C standard (C11) there are a number of other concurrency controls built in. But that is for another day, since we only take things in thin slices.

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.

RESOURCES

B. Japenga, "Concurrency in Embedded Systems (Part 4): Introducing Linux and Concurrency," *Circuit Cellar* 269, 2012.

M. Kerisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, No Starch Press, 2010.

The Open Group, "The Open Group Base Specifications Issue 7," 2008, http://pubs.opengroup.org/onlinepubs/9699919799.