Semaphores afford significant flexibility for providing interprocessor synchronization and mutual exclusion.

Semaphores Aid Multiprocessor Designs

By Ted Raineault

Multiprocessing designs that share memory among DSPs and other processors beg for some form of mutual exclusion or interprocessor synchronization. Such designs are becoming pervasive: DSPs are widely used in dense multiprocessor arrangements at the network edge, and systems-on-chip often include DSP cores to accelerate math-intensive computation. Although the DSP/BIOS operating system kernel provides a standard, efficient, robust API for uniprocessor applications, designers sometimes encounter situations in which interprocessor synchronization mechanisms would be very useful. One method implements interprocessor semaphores by using DSP/BIOS.

Shared-memory semaphores are basic tools for interprocessor synchronization. Although self-imposed design constraints can often reduce synchronization requirements, semaphores offer multiprocessor system designers significant flexibility. In addition, a multitasking OS, like DSP/BIOS, is virtually invaluable for shared-memory multiprocessing systems.

Assume that two or more processors share a physical pool of memory, in the sense that each processor sees the memory as directly addressable. Indeed, many multiprocessor DSP systems are designed in this manner.

SHARED-MEMORY ARCHITECTURES

One common architecture uses a large region of single-port RAM shared by all devices, often including a host. Although arbitration issues complicate the hardware design for this architecture, software engineers appreciate a large shared-memory systems-on-chip pool visible to multiple processors. When you can reduce bus contention and arbitration inefficiencies with algorithm and data transport strategies, software-related conveniences make this architecture an attractive option.

A second architecture uses dual-port RAM between processors. The downside here is the relatively high cost and small storage capacity of these devices; large banks of expensive dual-port RAM are seldom practical. However, in applications that use segmented data transport or small data sets for which small amounts of dual-port RAM are sufficient, this type of memory is very useful. Dual-port RAM is relatively fast; it's easy to design into a system; and, unlike FIFOs, it can store shared data structures used for interprocessor communication.

A word of caution about using shared memory. When processors have on-chip cache or systems use write posting, you must pay attention to shared-variable coherence. To prevent loss of coherence, you can, depending on the processor, disable the cache, use cache bypass, or flush the cache to ensure that a shared location is in a proper state. The cache control API in Texas Instruments' comprehensive Chip Support Library, for example, provides an excellent tool for managing cache subsystems. Solutions to write-post delay problems are system-specific.

Assume that two processors use a

Listing 1: Peterson's algorithm

```
initialization

   P1_wants_entry = P2_wants_entry = FALSE
   turn = P1

task P1

   P1_wants_entry = TRUE                    /* set lock */
   turn = P2                                /* grant turn to other task */
   loop while (P2_wants_entry and turn == P2) /* buzz waiting for lock */

   critical section                         /* execute critical section */

   P1_wants_entry = FALSE                   /* release lock */

task P2                                     /* same logic as P1 */

   P2_wants_entry = TRUE
   turn = P1
   loop while (P1_wants_entry and turn == P1)

   critical section

   P2_wants_entry = FALSE
```

common shared-memory buffer to pass data or to operate cooperatively on a data set. In either case, one or more tasks on the processors might need to know the state of the buffer before accessing it, and possibly to block while the buffer is in use. As in the case of single-processor multitasking, a mutual exclusion mechanism to prevent inappropriate concurrent operations on the shared resource is needed. A quick review of mutual exclusion will help clarify multiprocessor issues.

Shared-resource management is a fundamental challenge of multitasking. A task (or thread, or process) needs the ability to execute sequences of instructions without interference so that it can manipulate shared data atomically. These sequences, known as *critical sections,* are bracketed by entry and exit protocols that satisfy four properties: mutual exclusion, absence of deadlock, absence of unnecessary delay, and eventual entry (no starvation). The focus here is mutual exclusion; the remaining properties are detailed in any number of textbooks and will be satisfied by the multiprocessor semaphore discussed below.

Relative to a shared resource, mutual exclusion requires that only one task at a time execute in a critical section. Critical section entry and exit protocols use such mechanisms as polled flags (often called simple locks or spin locks) or more abstract entities, like blocking semaphores. Simple locks can be used to build protection mechanisms of greater complexity.

Introduced by Edgar Dijkstra in the mid-1960s, the semaphore is a system-level abstraction used for interprocess synchronization. The semaphore provides two atomic operations, wait (P) and signal (V), which are invoked to manipulate a nonnegative integer in the semaphore data structure. The wait operation checks the value of the integer and either decrements it if it's positive or blocks the calling task. The signal operation, in turn, checks for tasks blocked on the semaphore and either unblocks a task waiting for the semaphore or increments the semaphore if no tasks are waiting. A binary semaphore, which has counter values limited to 0 and 1, can be used effectively by an application to guard critical sections.

You can implement a multiprocessor semaphore by placing its data structure in shared memory and using RTOS services on each processor to handle blocking. Before outlining an implementation, let's look at two aspects of semaphores that cause complications in a multiprocessor environment. One is low-level mutual exclusion to protect shared data within a semaphore, and the other is wake-up notification when a semaphore is released.

LOW-LEVEL MUTUAL EXCLUSION

At its core, a semaphore has a count variable and possibly other data elements that must be manipulated atomically. System calls use simple mutual exclusion mechanisms to guard very short critical sections where the semaphore structure is accessed. This arrangement prevents incorrect results caused by concurrent modification of shared data within the semaphore.

In a uniprocessor environment, interrupt masking is a popular technique used to ensure that sequential operations occur without interference. With this technique, interrupts are disabled on entrance to a critical section and re-enabled on exit. In a multiprocessor situation, however, interrupt masking isn't an option. Even if one processor could disable the interrupts of another

(rarely the case), the second processor would still execute an active thread and might inadvertently violate mutual exclusion requirements.

A second technique uses an atomic test-and-set (or similar) instruction to manipulate a variable. This variable might be the semaphore count itself or a simple lock used to guard critical sections where semaphore data is accessed. Either way, a specialized instruction guarantees atomic read-modify-write in a multitasking environment.

Although this solution looks straightforward, test-and-set instructions have disadvantages in both uniprocessor and multiprocessor scenarios. One drawback is dependence on machine instructions, which vary across processors, provide only a small number of atomic operations, and are sometimes unavailable.

A second problem is bus locking: If multiple processors share a common bus that doesn't support locking during test-and-set, these processors might interleave accesses to a shared variable at the bus level while executing seemingly atomic test-and-set instructions.

A third problem concerns test-and-set behavior in multiport RAM systems: Even if all buses can be locked, simultaneous test-and-set sequences at different ports might produce overlapped accesses

Now consider two approaches that are very useful in shared-memory scenarios. One relies on simple atomic hardware locks; the other is a general-purpose software solution known as *Peterson's algorithm.*

In shared-memory systems, hardware-assisted mutual exclusion can be implemented with special bit flags found in multiport RAMs. Dual-port RAM logic prevents overlap of concurrent operations on the hardware flags, forcing them to maintain the correct state during simultaneous accesses. Also, because processors use standard read and write instructions to manipulate the flags, specialized atomic test instructions aren't required. However, this solution is still limited, as shared-memory systems often lack dedicated hardware flags.

Let's take one more step to arrive at a general-purpose hardware-independent method.

PETERSON'S ALGORITHM

Peterson's algorithm, published in 1981, provides an elegant software solution to the *n*-process critical section problem and has two key advantages over test-and-set spin locks. One is that atomic test-and-set isn't required: the algorithm eliminates the need for special instructions and bus locking. The other is eventual entry: A task waiting for entry to a critical section won't starve in a typical scheduling environment. Although Peterson's algorithm looks deceptively simple, it's a culmination of many attempts by researchers to solve the critical section problem.

Peterson's algorithm provides an elegant software solution to the *n*-process critical section problem and has two key advantages over test-and-set spin locks.

The pseudocode in Listing 1 shows the entry and exit protocols used to enforce two-process mutual exclusion. Note that Peterson adds a secondary `turn` variable. This variable prevents incorrect results caused by race conditions and also ensures that each waiting task will eventually enter the critical section.

We can easily imagine situations in which more than two processes try to enter their critical sections concurrently. Peterson's algorithm can, as noted, be generalized to *n* processes and used to enforce mutual exclusion for more than two tasks, and other *n*-process solutions, such as the bakery algorithm, are readily available in computer science textbooks. For reasons of clarity and brevity, the discussion here is limited to the two-process case. Pseudocode for the *n*-process Peterson's algorithm is available at www.electricsand.com.

Now that we have a low-level mutual exclusion tool with which to safely manipulate shared data within a semaphore, consider the other key ingredient of semaphores: blocking. Assuming that each processor runs DSP/BIOS or another multitasking OS, we'll develop our wait operation using services that are already available on each individual processor. DSP/BIOS provides a flexible semaphore module (SEM) that we'll use in the implementation.

When the owner of a uniprocessor semaphore releases it with a signal system call, the local scheduler has immediate knowledge of the signal event and can unblock a task waiting for the semaphore. In contrast, a multiprocessor semaphore implies that the owner and the requestor can reside on different processors. Because a remote kernel has no implicit knowledge of

Listing 2: Semaphore implementation pseudocode

```
MBS_wait () {

   success = FALSE                        /* not part of semaphore */
   while (success == FALSE) {             /* repeat acquisition attempt */

         TSK_disable ()                   /* prevent DSP/BIOS task switch */
         Peterson entry                   /* Peterson's entry protocol */
                                          /* critical section begins */
         if (sem_value == 0) {
               sem_value = sem_value - 1
               success = TRUE
         }
         else {
               notification_request = TRUE /* flag within semaphore */
         }

         Peterson exit                    /* end critical section */
         TSK_enable ()                    /* re-enable DSP/BIOS scheduler */

         if (success == FALSE) {          /* local variable shows result */
               SEM_pend ()                /* sleep using DSP/BIOS semaphore */
         }
   }
}

MBS_interrrupt {

   SEM_post ()                            /* local wake-up signal */
}

MBS_signal () {                           /* release the semaphore */

   TSK_disable ()                         /* prevent task switch */
   Peterson entry                         /* entry protocol */
                                          /* critical section begins */

   sem_value = sem_value +1;              /* increment the semaphore */

   if (notification_request == TRUE) {    /* notify a remote task? */
         notification_request = FALSE     /* clear flag in semaphore */
         send notification interrupt      /* yes - send an interrupt */
   }

   Peterson exit                          /* end critical section */
   TSK_enable ()                          /* re-enable DSP/BIOS scheduler */
}
```

signal calls to a local kernel, the remote kernel needs to be notified of local signal events in a timely manner. Our solution uses interprocessor interrupts to notify other processors of local activity involving a shared semaphore.

This implementation of a multiprocessor binary semaphore (MBS) assumes that the hardware supports interprocessor interrupts and that a task won't try to acquire a semaphore while a task on the same processor owns it. The latter restriction simplifies the example and can be easily removed with some additional design work.

USING SEMAPHORES

MBS_wait is invoked to acquire a shared-memory semaphore. If the semaphore is available, MBS_wait decrements it and continues. If the semaphore is already owned, the requesting task blocks within MBS_wait until a release notification interrupt makes the task ready to run. Once the interrupt occurs and higher-priority tasks have relinquished the CPU, the task waiting for the semaphore wakes up within MBS_wait and loops to retest it. Note that the task doesn't assume ownership immediately when unblocked. Because a remote task might reacquire the semaphore by the time the requestor wakes up, MBS_wait loops to compete for the semaphore again.

When MBS_wait determines that a semaphore is unavailable, it sets a notification request flag in the shared-semaphore data structure to indicate that the processor should be interrupted when the semaphore is released elsewhere in the system. To avoid a race condition known as the lost wake-up problem, MBS_wait tests the semaphore atomically and sets the notification request flag if the semaphore is unavailable.

Code for the wait operation is divided into two distinct parts: MBS_wait, which contains the blocking code and is called by an application, and MBS_interrupt, which runs in response to the notification interrupt and posts a local signal to the task waiting on the semaphore. This arrangement is very similar to that for a device driver: The upper part of a driver suspends a task pending I/O service and the interrupt-driven

lower part wakes up the task.

MBS_signal releases a semaphore by incrementing its value and posting an interrupt to the processor that requested release notification. These actions cause MBS_interrupt to execute on the remote processor where a task is blocked waiting for the semaphore. Note that this sequence of events varies slightly from those of the uniprocessor signal operation described earlier in which the semaphore is incremented only if no tasks are blocked.

PSEUDOCODE

Now that we have a notion of shared-semaphore architecture, let's look at the pseudocode describing the wait and signal operations, shown in Listing 2. General solutions for multiple tasks per processor and for a greater number of processors can be implemented with modified MBS operations using the *n*-process Peterson's algorithm.

Note that the critical sections, enforced by Peterson's algorithm (Peterson entry and Peterson exit), are very short instruction sequences used to manipulate the semaphore data structure. The details of Peterson's algorithm aren't shown; they're implicit in the Peterson entry and exit operations. The lock and turn variables used in Peterson's algorithm are distinct from the semaphore data elements accessed in the critical sections.

The critical sections are preceded with DSP/BIOS TSK_disable calls to prevent task switching. A task switch during a critical section could cause another processor trying to enter the same critical section to spin indefinitely in Peterson entry if it tried to acquire the same semaphore. The critical sections should be executed as quickly as possible.

Also note that the example omits error checking, return values, and timeouts. The pseudocode is meant to highlight discussion topics rather than provide a detailed implementation template. ◆

Ted Raineault is the cofounder and technical director of Electric Sand Inc. in Poway, Calif. He previously worked as a sales executive for a DSP board company and has 12 years' experience as an embedded software engineer.