Concurrency And Task Synchronization

Robert Ward
Publisher, The C Users Journal and TECH Specialist

Every non-trivial real-time system monitors or processes more than one simultaneously occurring stream of events. The systems are — by their very nature — concurrent. But many real-time programmers, especially those working in small embedded systems, tend to avoid concurrent programming techniques — often because they believe that solutions to the technical problems created by concurrent programming are too complex and large for an embedded application.

Managing concurrency needn’t be complex if the environment and problem aren’t complex. This paper will describe simple, practical solutions to mutual exclusion and deadlock problems.

MUTUAL EXCLUSION

Whenever two or more concurrent processes each modify a shared resource, the resource may become corrupted unless its use is carefully managed. For example, assume we have two processes, one which must increase a counter (perhaps to keep track of characters it has placed in a buffer) and another which must decrease the same counter (perhaps to reflect that it has removed a character from the same buffer).

Using a generic assembly language (destination operand first), the increment and decrement operations might be coded:

<table>
<thead>
<tr>
<th>Increase Counter</th>
<th>Decrease Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov x, count</td>
<td>mov x, count</td>
</tr>
<tr>
<td>inc x</td>
<td>dec x</td>
</tr>
<tr>
<td>jc overflow</td>
<td>jc underflow</td>
</tr>
<tr>
<td>mov count, x</td>
<td>mov count, x</td>
</tr>
</tbody>
</table>
The code is straightforward, but if it is executed by two separate concurrent tasks, strange results may appear:

<table>
<thead>
<tr>
<th>Process 1 (increase call)</th>
<th>Process 2 (decrease call)</th>
<th>value in count</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov x,count</td>
<td>mov x,count</td>
<td>5</td>
</tr>
<tr>
<td>inc x</td>
<td>dec x</td>
<td>5</td>
</tr>
<tr>
<td>mov x,count</td>
<td>jc underflow</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>mov count, x</td>
<td>4</td>
</tr>
<tr>
<td>mov count, x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Process 2's change has been completely lost.

The increment and decrement code are critical sections (code that must complete without interruption). The variable count is a shared resource. A programmer can avoid this kind of error by enforcing mutual exclusion on the use of count.

The three ways to achieve mutual exclusion are:

1. Make the operation “indivisible” by disabling interrupts during the critical section.

2. Reduce the critical code section to a single indivisible machine instruction.

3. Use a monitor to ensure that only one process at a time is attempting to execute a critical section related to a particular shared resource.

**DISABLE INTERRUPTS**

The simplest general mechanism for protecting a critical section is to disable interrupts during its execution.

<table>
<thead>
<tr>
<th>increase counter</th>
<th>decrease counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov x,count</td>
<td>mov x, count</td>
</tr>
<tr>
<td>inc x</td>
<td>dec x</td>
</tr>
<tr>
<td>mov count, x</td>
<td>mov count, x</td>
</tr>
<tr>
<td>ei</td>
<td>ei</td>
</tr>
<tr>
<td>jc overflow</td>
<td>jc underflow</td>
</tr>
</tbody>
</table>

In this code, overflow and underflow tests have been delayed so an error won’t cause the program to branch around the enable interrupt. Instead, you could include the
branch instruction within the critical section and duplicate the ei instruction in the error handlers. I prefer to keep the bracketing ei/di pairs close together and “structured”. Not, however, that testing for the error outside of the critical section allows a potentially erroneous value to be stored into count — and used by another process before the error handler is invoked.

In general, you must place the di before the first instruction which performs a “significant” read from the shared resource. A read is “significant” only if its result influences the value that will later be stored back to the shared resource. You must place the matching ei after the write that modifies the shared resource.

In simple systems, you may be able to force all critical sections to reside in an interrupt routine. This produces a clean solution because interrupts are normally off during the interrupt routine anyway.

**USE A SINGLE INSTRUCTION**

Some critical operations are so simple that they can be performed in a single machine instruction. For example, if the machine can increment and decrement memory operands, you could rewrite the earlier critical sections as:

```
inc count  dec count
jc overflow jc underflow
```

A non-counting semaphore is the most elemental shared resource. Concurrent processes communicate simple state information (e.g. whether some associated shared resource has been allocated) via semaphores. If the resource is allocated, the semaphore is set; if the resource is free, the semaphore is cleared. When a process needs to use a resource it must:

1. Test the semaphore to see if the resource is free.

2. If so, allocate the resource by setting the semaphore.

Since the semaphore itself is a shared resource, the program must also enforce mutual exclusion on the semaphore. Some architectures support this operation with a single “test and set” instruction, which will test a memory location (setting processor flags based on the result) and set the location if it is “clear”. If you have such an instruction, you can implement semaphores and mutual exclusion without disabling
interrupts. If you don't, have the "test and set" instruction simulate it with a simple macro:

```
; assumes memory to memory operations
; sets zero flag if mem was zero
; else clears zero flag
; changes mem to one if mem was zero
;
TSET MACRO mem
    di
    or mem,mem
    jnz $+2
    inc mem    ; must not affect zero flag
    ei
END
```

USE A MONITOR

Some critical sections are so long that disabling interrupts during the entire computation isn't practical. Using a monitor minimizes the need to disable interrupts while still enforcing mutually exclusive access to a shared resource. A monitor isn't really a separate, identifiable code object; rather it is a technique. Each monitor is associated with a single shared resource and consists of a semaphore and a coding discipline at the beginning and end of each critical section that uses the associated shared resource.

For example, to use a monitor to control access to a printer, first declare a semaphore, say `PRINTER_BUSY`, and then surround each critical section that uses the printer with this code:

```
alloc   TSET PRINTER_BUSY
    jnz alloc  

; if you get here, the printer is yours
;
; insert critical code here
;
; free the printer with an indivisible operation
    clr PRINTER_BUSY
```

This monitor is very unsophisticated, but perfectly appropriate for many embedded systems applications. The busy-wait loop at the top wastes processor time and
creates unnecessary access conflicts for the semaphore PRINTER_BUSY. A better version would avoid the busy-wait by using the operating system's scheduler:

```c
while (tset(&PRINTER_BUSY)) wait(PRINTER_FREE);
/* wait puts a process to sleep until the signal (PRINTER_FREE) is received */
/* insert critical code here */
/* now release the resource */
clear(&PRINTER_BUSY); /* must be indivisible */
signal(PRINTER_FREE);
```

The loop at the top of this monitor isn't a busy-wait; it's a synchronized retry. When the monitor is entered, if the printer is busy the process is put to sleep by the `wait`. Since the monitor coding discipline requires this same structure around every access to the printer, when the printer is freed, a signal will be raised, waking this process. Waking the process, though, doesn't allocate the printer. To allocate the printer the `tset()` must be repeated. It's even possible that the `tset()` could fail on the second try. (Perhaps some other process entered the monitor at the same time that this process received a signal.) Repeating the `tset()` operation ensures the resource is properly allocated.

The `clear()` function merely assigns zero to its argument, but does so as an indivisible operation.

Of course, if your operating system supports `wait()` and `signal()`, it probably also directly supports semaphore and critical sections. If you don't have the support of a sophisticated operating system and still need to eliminate the busy-wait, you can build a simple scheduler and call it at the beginning and end of the monitor.

**DEADLOCK**

When two or more processes need more than one resource, they can get locked in an infinite waiting cycle. For example, if Processes A and B each need exclusive use of both the system's single DMA controller and its single printer, but Process A successfully allocates the DMA controller at the same time that Process B successfully allocates the printer, neither process will be able to continue since each will be waiting for the other to release its resource.

For a system to deadlock, it must meet these four prerequisites [1]:

1. Mutual exclusion. Resources are allocated to each process exclusively.
2. Non-preemptive scheduling. A resource can only be released by the process that acquired it.
3. Partial allocation. Processes can acquire needed resources one at a time.
4. Circular waiting. Processes which have acquired some resources, can enter a state where they wait indefinitely for another process to release a resource.

Avoiding any of these four pre-conditions will prevent deadlock. In my experience, eliminating partial allocation and circular waiting from most embedded systems designs is fairly easy.

AVOIDING PARTIAL ALLOCATION

Partial allocation isn’t an issue unless some process within the system requires multiple resources. But, even when certain processes need multiple resources, you can probably avoid partial allocation by treating certain resource combinations as single entities.

Consider a system with two concurrent I/O processes (each capable of either input or output) and resources consisting of one I/O buffer, one printer and one “reader”. Each I/O process requires exclusive control of the I/O buffer and either the reader or the printer, depending on the operation. If you allow partial allocation this system can deadlock (e.g. the one process planning output acquires the I/O buffer at the same time as the second process acquires the printer). However, if you view the entire set of resources as a single I/O subsystem and require each process to always allocate the entire subsystem, you prevent deadlock.

In this instance (as is often the case), nothing is lost by allocating both the reader and printer to each process (even though they each use just one). Because the system has only one output buffer, it can be performing only one type of I/O at a time anyway.

If the system had two I/O buffers, the simplest solution would be to permanently assign one buffer to the printer and one to the reader. Allocating the printer would implicitly allocate its buffer. This solution avoids deadlock but also reduces speed. If the buffers weren’t permanently assigned, you could improve performance by double buffering one device at a time.

Since certain sets of resources are naturally used together, you can often avoid partial allocation by partitioning the resources into sets that can be allocated as a unit. Consider this more complex system:

<table>
<thead>
<tr>
<th>Processes</th>
<th>Resources Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>A1, A2, B, C</td>
</tr>
<tr>
<td>Process 2</td>
<td>F</td>
</tr>
<tr>
<td>Process 3</td>
<td>E</td>
</tr>
<tr>
<td>Process 4</td>
<td>A, C</td>
</tr>
<tr>
<td>Process 5</td>
<td>D, E</td>
</tr>
<tr>
<td>Process 6</td>
<td>A, B</td>
</tr>
<tr>
<td>Process 7</td>
<td>A, F</td>
</tr>
</tbody>
</table>

(Where A1 and A2 indicate two instances of resource type A.)
If the system had three instances of resource A available, partial allocation could be avoided by dividing the resources into these three classes (or subsystems):

Class 1 A1, A2, B, C
Class 2 A3, F
Class 3 D, E

Since every process can obtain the resources it needs (and occasionally some it doesn't need) with a request to only one class, partial allocation isn't necessary.

Even if the system had only two instances of resource type A, you could still avoid partial allocation, though the required resource grouping is far less "natural":

Class 1 A1, A2, B, C, F
Class 2 D, E

As the groupings become less "natural", the potential for significant inefficiencies and bottlenecks increases.

ORDERED ALLOCATION

Even if you can't always avoid partial allocation, you should always be able to avoid circular waiting — simply force partial allocations to occur in a specific sequence.

For example, to prevent deadlock in a system which has two I/O processes, one I/O buffer, one printer and one reader, simply force all resources to be allocated in this order:

1. readers, if any
2. printers, if any
3. I/O buffers, if any

If all processes acquire resources in this order, it is impossible for a process that owns an I/O buffer to be waiting on a process that owns a printer — no process can acquire an I/O buffer unless it has already acquired any printer it might need!

Any linear ordering will avoid deadlock (strictly speaking, any partial ordering will do); the critical issue is to impose an ordering. Note that these alternate orders also avoid deadlock:

readers printers I/O buffers
I/O buffers I/O buffers readers
printers readers printers

Note also that all of these orderings avoid deadlock even if the system has multiple printers, readers and I/O buffers.
Conceptually, it's simple to force an order upon all allocations. Success in practice, however, can be surprisingly difficult because some independent resources are frighteningly subtle. For example, in some situations you must treat a status flag and the associated data port as separate resources. It is seldom practical to inventory a complex system in such fine-grained detail.

Thus, in all but the simplest systems I apply both methods. At the program's lower levels, I insist that resources be allocated in clumps, (i.e., the status flag and data port become one logical entity). At the higher levels, I allow partial allocation but enforce a strict ordering. This combination of methods usually produces a design that is both manageable and reasonably efficient.

CONCLUSION

Real-time problems are almost always easier to solve when analyzed and modelled as a set of concurrent processes. Properly managing concurrency in an operating system that manages hundreds of diverse resources for hundreds of processes can require complex and bulky schedulers and monitors. On the other hand, managing concurrent access to one or two shared resources for a handful of simple processes is relatively straightforward. The simple techniques shown here address the technical problems of mutual exclusion and deadlock and (for the most part) can be used in even the most primitive systems. □

Robert Ward is the author of *A Programmer’s Introduction to Debugging C* and president of R&D Publications Inc., which publishes *The C Users Journal, Tech Specialist*, and *Unique*. He is president of the C Users Group and holds an M.S. in computer science from the University of Kansas.