Embedded Operating Systems - Part 3: Intertask communication

April 16, 2013

Editor's Note: Embedded Systems Architecture, 2nd Edition, is a practical and technical guide to understanding the components that make up an embedded system’s architecture. Offering detailed explanations and numerous code examples, the book provides a comprehensive get-up-and-running reference for those new to the field and those updating their skills. This excerpt offers an introduction and review of embedded operating systems. In Part 1, the author defined the unique characteristics of an embedded OS and describes how an embedded OS works with processes. In Part 2, the author reviewed the different types of process schedulers found in embedded OSs. In this installment, the author discusses how embedded OSs provide mechanisms that let tasks intercommunicate and synchronize their behavior.

Adapted from "Embedded Systems Architecture, 2nd Edition" by Tammy Noergaard (Newnes)

9.2.3 Intertask Communication and Synchronization

Different tasks in an embedded system typically must share the same hardware and software resources or may rely on each other in order to function correctly. For these reasons, embedded OSs provide different mechanisms that allow for tasks in a multitasking system to intercommunicate and synchronize their behavior so as to coordinate their functions, avoid problems, and allow tasks to run simultaneously in harmony.

Embedded OSs with multiple intercommunicating processes commonly implement interprocess communication (IPC) and synchronization algorithms based upon one or some combination of memory sharing, message passing, and signaling mechanisms.

With the shared data model shown in Figure 9-28, processes communicate via access to shared areas of memory in which variables modified by one process are accessible to all processes.

![Figure 9-28. Memory sharing.](image-url)
While accessing shared data as a means to communicate is a simple approach, the major issue of race conditions can arise. A race condition occurs when a process that is accessing shared variables is pre-empted before completing a modification access, thus affecting the integrity of shared variables. To counter this issue, portions of processes that access shared data, called critical sections, can be earmarked for mutual exclusion (or Mutex for short). Mutex mechanisms allow shared memory to be locked up by the process accessing it, giving that process exclusive access to shared data. Various mutual exclusion mechanisms can be implemented not only for coordinating access to shared memory, but for coordinating access to other shared system resources as well. Mutual exclusion techniques for synchronizing tasks that wish to concurrently access shared data can include:

Processor-assisted locks for tasks accessing shared data that are scheduled such that no other tasks can pre-empt them; the only other mechanisms that could force a context switch are interrupts. Disabling interrupts while executing code in the critical section would avoid a race condition scenario if the interrupt handlers access the same data. Figure 9-29 demonstrates this processor-assisted lock of disabling interrupts as implemented in VxWorks. VxWorks provides an interrupt locking and unlocking function for users to implement in tasks. Another possible processor-assisted lock is the “test-and-set-instruction” mechanism (also referred to as the condition variable scheme). Under this mechanism, the setting and testing of a register flag (condition) is an atomic function, a process that cannot be interrupted, and this flag is tested by any process that wants to access a critical section. In short, both the interrupt disabling and the condition variable type of locking schemes guarantee a process exclusive access to memory, where nothing can pre-empt the access to shared data and the system cannot respond to any other event for the duration of the access.

![Figure 9-29. VxWorks processor-assisted locks.][10]

Semaphores, which can be used to lock access to shared memory (mutual exclusion) and also can be used to coordinate running processes with outside events (synchronization). The semaphore functions are atomic functions, and are usually invoked through system calls by the process. Example 10 demonstrates semaphores provided by VxWorks.

Example 10: VxWorks Semaphores

VxWorks defines three types of semaphores:

Binary semaphores are binary (0 or 1) flags that can be set to be available or unavailable. Only the associated resource is affected by the mutual exclusion when a binary semaphore is used as a mutual exclusion mechanism (whereas processor assisted locks, for instance, can affect other unrelated resources within the system). A binary semaphore is initially set = 1 (full) to show the resource is available. Tasks check the binary semaphore of a resource when wanting access and, if available, then take the associated semaphore when accessing a resource (setting the binary
Semaphore = 0), and then give it back when finishing with a resource (setting the binary semaphore = 1). When a binary semaphore is used for task synchronization, it is initially set equal to 0 (empty), because it acts as an event other tasks are waiting for. Other tasks that need to run in a particular sequence then wait (block) for the binary semaphore to be equal to 1 (until the event occurs) to take the semaphore from the original task and set it back to 0. The VxWorks pseudocode example below demonstrates how binary semaphores can be used in VxWorks for task synchronization.

```
#include "VxWorks.h"
#include "sem.lib.h"
#include "arch/arch/vkernel.h" /* replace arch with architecture type */

SEM_ID syncSem; /* ID of sync semaphore */

init (int someIntNum)
{
    /* connect interrupt service routine */
    intConnect (INUM_TO_IVEC (someIntNum), eventInterruptSvcRout, 0);

    /* create semaphore */
    syncSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY);

    /* spawn task used for synchronization. */
    taskSpawn ("sample", 100.0, 20000, task1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
}

task1 (void)
{
    semTake (syncSem, WAIT_FOREVER); /* wait for event to occur */
    printf ("task 1 got the semaphore!");
    /* process event */
}

eventInterruptSvcRout (void)
{
    semGive (syncSem); /* let task 1 process event */
}
```

Mutual exclusion semaphores are binary semaphores that can only be used for mutual exclusion issues that can arise within the VxWorks scheduling model, such as priority inversion, deletion safety (ensuring that tasks that are accessing a critical section and blocking other tasks aren't unexpectedly deleted), and recursive access to resources. Below is a pseudocode example of a mutual exclusion semaphore used recursively by a task's subroutines.
Counting semaphores are positive integer counters with two related functions: incrementing and decrementing. Counting semaphores are typically used to manage multiple copies of resources. Tasks that need access to resources decrement the value of the semaphore; when tasks relinquish a resource, the value of the semaphore is incremented. When the semaphore reaches a value of “0,” any task waiting for the related access is blocked until another task gives back the semaphore.

On a final note, with mutual exclusion algorithms, only one process can have access to shared memory at any one time, basically having a lock on the memory accesses. If more than one process blocks waiting for their turn to access shared memory, and relying on data from each other, a deadlock can occur (such as priority inversion in priority based scheduling). Thus, embedded OSs have to be able to provide deadlock-avoidance mechanisms as well as deadlock-recovery mechanisms.
mechanisms. As shown in the examples above, in VxWorks, semaphores are used to avoid and prevent deadlocks.

Intertask communication via message passing is an algorithm in which messages (made up of data bits) are sent via message queues between processes. The OS defines the protocols for process addressing and authentication to ensure that messages are delivered to processes reliably, as well as the number of messages that can go into a queue and the message sizes. As shown in Figure 9-30, under this scheme, OS tasks send messages to a message queue, or receive messages from a queue to communicate.

![Figure 9-30. Message queues.](image)

The wind kernel supports two types of signal interface: UNIX BSD-style and POSIX-compatible signals.

Microkernel-based OSs typically use the message passing scheme as their main synchronization mechanism. Example 11 demonstrates message passing in more detail, as implemented in VxWorks.

Example 11: Message Passing in VxWorks

VxWorks allows for intertask communication via message passing queues to store data transmitted between different tasks or an interrupt service routine (ISR). VxWorks provides the programmer four system calls to allow for the development of this scheme:

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgQCreate()</td>
<td>Allocates and initializes a message queue</td>
</tr>
<tr>
<td>msgQDelete()</td>
<td>Terminates and frees a message queue</td>
</tr>
<tr>
<td>msgQSend()</td>
<td>Sends a message to a message queue</td>
</tr>
<tr>
<td>msgQReceive()</td>
<td>Receives a message from a message queue</td>
</tr>
</tbody>
</table>

These routines can then be used in an embedded application, as shown in the source code example below, to allow for tasks to intercommunicate:
Signals and Interrupt Handling (Management) at the Kernel Level

Signals are indicators to a task that an asynchronous event has been generated by some external event (other processes, hardware on the board, timers, etc.) or some internal event (problems with the instructions being executed, etc.). When a task receives a signal, it suspends executing the current instruction stream and context switches to a signal handler (another set of instructions). The signal handler is typically executed within the task’s context (stack) and runs in the place of the signaled task when it is the signaled task’s turn to be scheduled to execute.

The wind kernal supports two types of signal interface: UNIX BSD-style and POSIX-compatible signals (see Figure 9-31).
Signals are typically used for interrupt handling in an OS, because of their asynchronous nature. When a signal is raised, a resource’s availability is unpredictable. However, signals can be used for general intertask communication, but are implemented so that the possibility of a signal handler blocking or a deadlock occurring is avoided. The other intertask communication mechanisms (shared memory, message queues, etc.), along with signals, can be used for ISR-to-Task level communication, as well.

When signals are used as the OS abstraction for interrupts and the signal handling routine becomes analogous to an ISR, the OS manages the interrupt table, which contains the interrupt and information about its corresponding ISR, as well as provides a system call (subroutine) with parameters that that can be used by the programmer. At the same time, the OS protects the integrity of the interrupt table and ISRs, because this code is executed in kernel/supervisor mode. The general process that occurs when a process receives a signal generated by an interrupt and an interrupt handler is called is shown in Figure 9-32.

As mentioned in previous chapters, the architecture determines the interrupt model of an embedded system (that is, the number of interrupts and interrupt types). The interrupt device drivers initialize and provide access to interrupts for higher layer of software. The OS then provides the signal IPC mechanism to allow for its processes to work with interrupts, as well as being able to provide various interrupt subroutines that abstracts out the device driver.

While all OSs have some sort of interrupt scheme, this will vary depending on the architecture they are running on, since architectures differ in their own interrupt schemes. Other variables include interrupt latency/response, the time between the actual initiation of an interrupt and the execution
Example 12 shows an interrupt scheme of a real-world embedded RTOS.

Example 12: Interrupt Handling in VxWorks

Except for architectures that do not allow for a separate interrupt stack (and thus the stack of the interrupted task is used), ISRs use the same interrupt stack, which is initialized and configured at system start-up, outside the context of the interrupting task. Table 9-1 summarizes the interrupt routines provided in VxWorks, along with a pseudocode example of using one of these routines.

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intConnect()</td>
<td>Connects a C routine to an interrupt vector</td>
</tr>
<tr>
<td>intContext()</td>
<td>Returns TRUE if called from interrupt level</td>
</tr>
<tr>
<td>intCount()</td>
<td>Gets the current interrupt nesting depth</td>
</tr>
<tr>
<td>intLevelSet()</td>
<td>Sets the processor interrupt mask level</td>
</tr>
<tr>
<td>intLock()</td>
<td>Disables interrupts</td>
</tr>
<tr>
<td>intUnlock()</td>
<td>Re-enables interrupts</td>
</tr>
<tr>
<td>intVecBaseSet()</td>
<td>Sets the vector base address</td>
</tr>
<tr>
<td>intVecBaseGet()</td>
<td>Gets the vector base address</td>
</tr>
<tr>
<td>intVecSet()</td>
<td>Sets an exception vector</td>
</tr>
<tr>
<td>intVecGet()</td>
<td>Gets an exception vector</td>
</tr>
</tbody>
</table>

/* This routine initializes the serial driver, sets up interrupt vectors, * and performs hardware initialization of the serial ports. */
void InitSerialPort (void)
{
    intConnect(INUM_TO_IVEC (INT_NUM_SCC), serialInt, 0);
    _
    InitSerialPort();
}