Managing Concurrency in Complex Embedded Systems

David Cummings
Kelly Computing, Inc.
4 PUBLICATIONS **28** CITATIONS

Managing Concurrency in Complex Embedded Systems

David M. Cummings
Kelly Technology Group

Background

This paper is based on portions of an invited presentation delivered at a 2010 workshop on Cyber-Physical Systems. The slides for that presentation can be found at: http://www.kellytechnologygroup.com/main/fujitsu-cps-workshop-10jun2010-invited-presentation.pdf.

Introduction

Experienced developers of complex embedded systems know that concurrent programming is hard. Building a system consisting of many processes and/or threads, and ensuring that the system meets its performance objectives and behaves reliably and predictably, sometimes in the face of unpredictable errors, is no small accomplishment. In this paper, I will present some guiding principles that can be very effective in building complex embedded systems that are highly reliable, robust and predictable. Over the course of more than three decades, my colleagues and I have used these principles with success on a wide variety of projects, including:

- A network of 27 Doppler radar processors for the FAA
- A digital receiver for NASA's Deep Space Network
- The flight computer for NASA's Mars Pathfinder Spacecraft
- A TDMA satellite modem card for Windows PCs for broadband IP access over the air
- A terrestrial wireless appliance for broadband IP access over the air
- A clustered backup and disaster recovery appliance

Of course, a single paper cannot capture all of the factors that go into building a successful, complex embedded system. Here I will focus on the following key points:

- The threading model
- The structure within each thread

First, examples of threading models and structures that can cause problems will be presented. I will then present a solution that avoids these problems.

Attributes of Embedded Systems

The typical embedded system considered in this paper includes the following attributes:

- Real-time deadlines
- Extensive interaction with hardware
- Largely or completely autonomous
- Often thought of as hardware devices that "just work"
- Extensible
- Maintainable

These systems may be either uniprocessor systems or multiprocessor/multicore systems. The hardware with which the systems interact sometimes consists of ASICs and/or FPGAs that are co-designed with the software. In addition to extensive interaction with hardware, the systems often include network communication with other devices as well. And, as any experienced developer knows, maintainability and extensibility are critical, because there will be bug fixes and feature enhancements over the lifetime of the system.

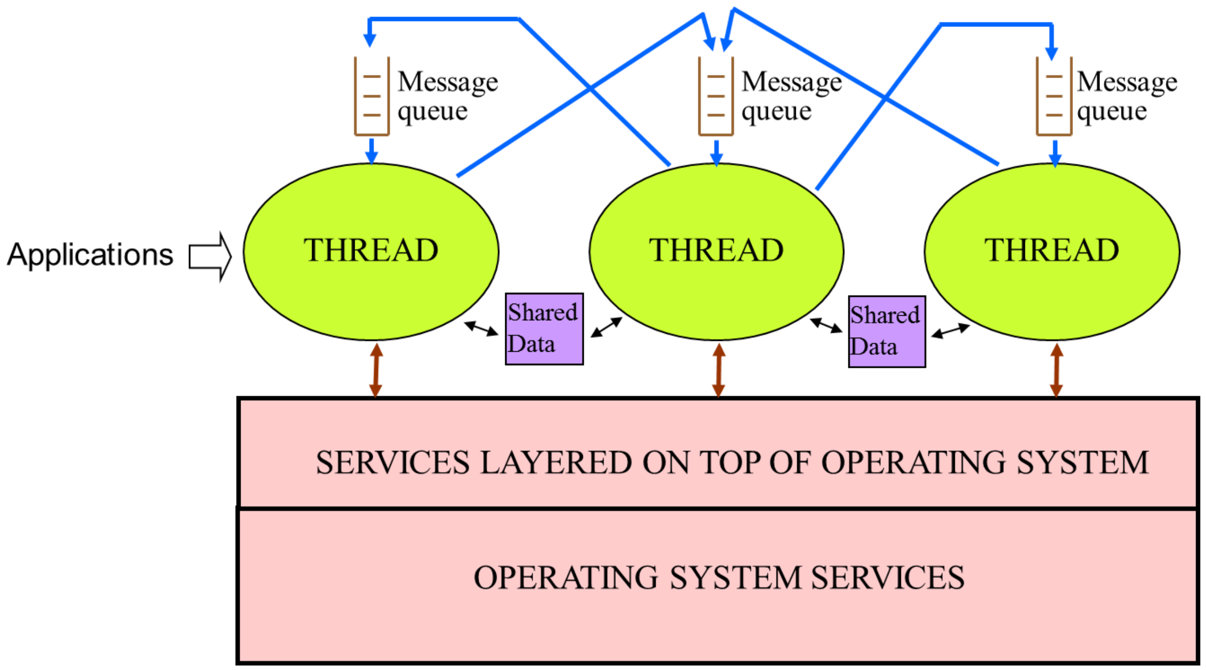
Also, I assume that the system is running at least a minimal real-time kernel with support for:

- Threads
- Preemptive, priority-based scheduling
- Message passing
- Synchronization primitives (e.g., locks/semaphores)

My colleagues and I have put these ideas into practice in both commercial products and government deployments using a number of operating systems, including VxWorks, Linux, and OpenVMS. Most of the systems we have built were written in C or C++. One was written in Ada.

The Threading Model

The following diagram represents a simplified view of a typical threading model for such systems:



The application code is executed within threads, processes, or a combination of the two. For simplicity, in this paper I will assume there are multiple threads sharing a single address space, although the same general considerations apply to multiple processes, each consisting of one or more threads.

The threads communicate with one another by depositing messages into message queues. Usually, this is performed by services in the two underlying layers of the diagram. Sometimes a thread will call a service to explicitly deposit a message into another thread's queue. Alternatively, a thread may call a service to perform some function, which will result, under the covers, in that service depositing a message into another thread's queue.

For large amounts of data that have to be shared between threads, shared memory is often used, in order to limit the sizes of the messages sent via message queues. Often, the data is written into shared memory by the initiating thread, and then a message is sent to the target thread's message queue that informs the target thread, either explicitly or implicitly, that data is available in shared memory. (Alternatively, for timer-based cyclic activities, the initiating thread writes the data into shared memory during one time period, and the target thread reads the data out of shared memory during a second time period.) The underlying services used to manage the data in the shared memory, i.e., the services used by the initiating thread to deposit the data into shared memory, and the services used by the target thread to read the data from shared memory, typically perform any necessary synchronization via locks/semaphores to ensure thread-safe access.

So far, this probably seems straightforward. Indeed, virtually all of the multithreaded embedded systems we have developed over the years, as well as many of the systems that we have seen from others, conform to this general model. However, there are a number of degrees of freedom with respect to the communication disciplines between the threads (e.g., request-response vs. send and forget), as well as the associated mechanisms to implement these communication disciplines. This is where one can get into trouble. This will become apparent in the following sections, where I discuss the internal structure of the threads and then show how that structure can be misused, leading to potential problems.

The Internal Structure of the Threads

The typical structure of a thread within such systems is shown in the following pseudocode:

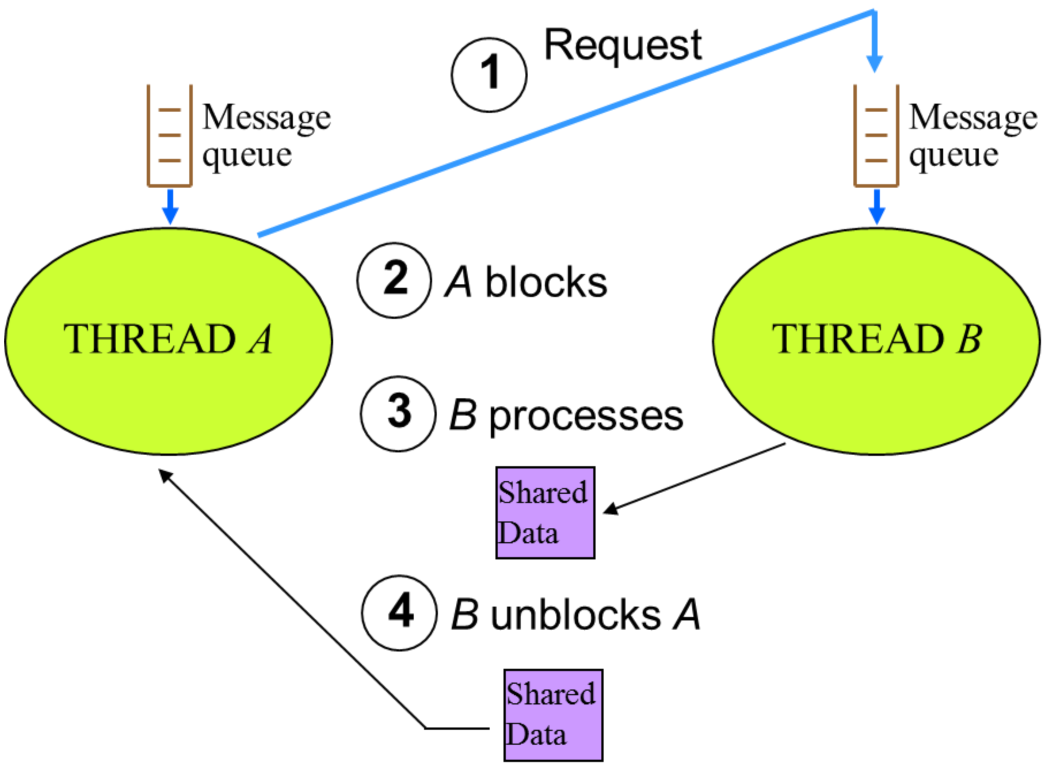
```
INITIALIZE
DO Forever
    Wait for a message
    CASE message ID
        Msg 1: Process Msg 1
               Break
        Msg 2: Process Msg 2
               Break
        Msg 3: Process Msg 3
               Break
          .
          .
          .
        Msg n: Process Msg n
               Break
        Invalid ID: Handle error
    END CASE
END DO
```

Again, this seems straightforward. After performing thread-specific initialization, the thread executes its main event loop. At the top of the loop, the thread waits for the arrival of a message in its message queue. Upon receipt of a message, the thread performs the processing required for that specific message, and then returns to the top of the loop to wait for the next message.

However, this simple structure is often misused. To see how, we'll look at one of the communication disciplines (request-response) that is commonly used between the threads, how it is frequently implemented, and the problems that can result. Often, the implementations described below seem like reasonable solutions to the requirements at hand. Frequently, however, they lead to problems at some point down the road. This is especially true for complex systems that evolve and grow over time.

Request-Response (Client-Server) Communication

It is very common in the embedded systems we are considering for some of the threads to interact in a request-response fashion. One thread (the client) sends a request to another thread (the server). The server subsequently sends a response, at which point the client processes the response. One possible implementation of this (which is quite common based on our experience) is depicted in the following diagram. Thread A is the client and Thread B is the server:



In this example, A sends a message to B that requests a service or an action of some sort (step 1), and A blocks to wait for a response from B (step 2). B receives the request and performs the necessary processing, depositing the response in shared memory (step 3). B then unblocks A so A can read the response from the shared memory and take action based on it (step 4).

So what's the problem? Let's look at how this is actually implemented within Thread A:

THREAD A:

```
INITIALIZE
DO Forever
    Wait for a message
    CASE message ID
        Msg 1: Process message
               Send request to Thread B
               Wait for response
               Process response
               Break
        Msg 2: Process Msg 2
               Break
        Msg 3: Process Msg 3
               Break
          .
          .
          .
        Msg n: Process Msg n
               Break
        Invalid ID: Handle error
    END CASE
END DO
```

In this example, I assume that it is the receipt of Msg 1 that causes A to initiate the request-response communication with B. (It must be the receipt of some message by A that triggers this communication with B.)

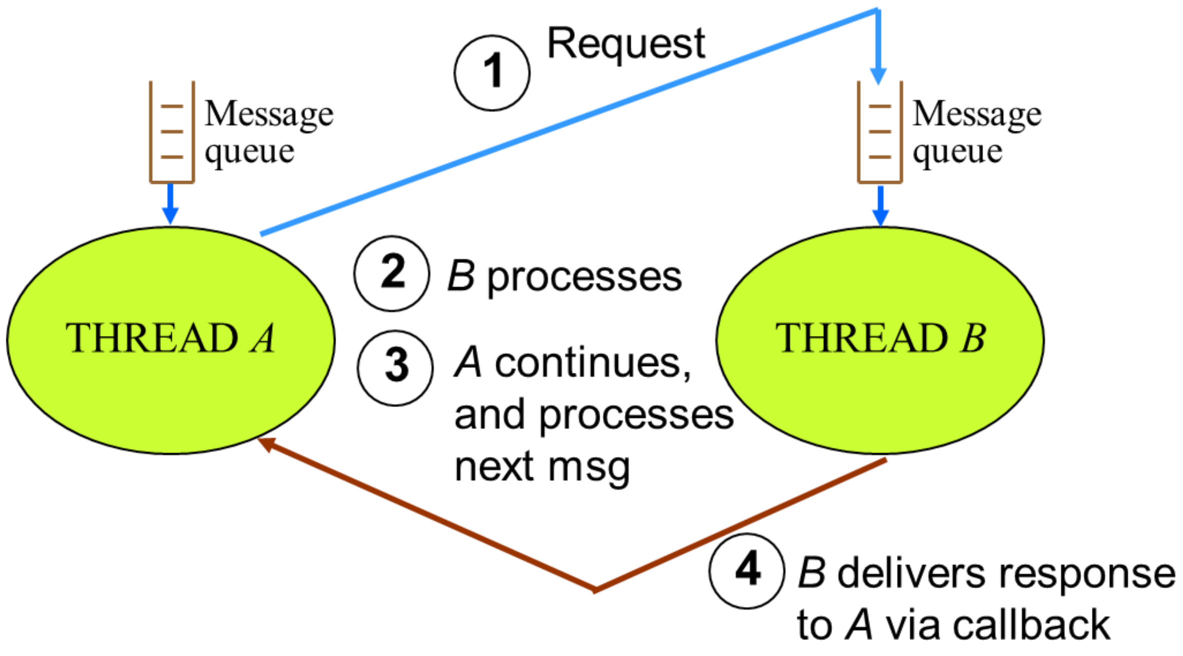
The problem is in the red step. While A waits, it cannot receive and respond to other messages in its message queue. This can have real-time implications if, for example, certain messages sent to A while A is waiting for B require that A respond within a deadline. Also, what if we want to tell A to gracefully shut down while it is waiting for B? In addition, while in this state, A can't respond to a ping from a software watchdog timer, which is a standard reliability feature built into many of these systems. Furthermore, there is a deadlock potential between A and B, if A is holding a resource that B needs in order to complete its processing of the request from A.

The real-time response issue is sometimes addressed by fiddling with priorities to ensure that B's priority is at least as high as A's. However, that might artificially boost B's priority beyond what it should be, in order to accommodate the worst-case deadlines imposed on A for any message A could possibly receive. Furthermore, it relies on B's processing of A's request occurring quickly enough to satisfy the worst-case deadline imposed on A for any message A could receive (introducing dangerous coupling between A and B). In a complex system with many different interacting threads and hardware devices, such a "solution" can end up introducing other problems. Even if you get it right by exhaustively analyzing all possible interactions and timing sequences, which can be quite difficult, a seemingly unrelated change at some point down the road can invalidate such an analysis and break the system.

It might seem that, rather than permanently changing B's priority, a better approach would be to apply one of the well-known techniques for preventing priority inversion, e.g., priority inheritance or Highest Locker functionality. That way, B's priority would only be elevated while A was blocked waiting for the response from B. But this would still result in dangerous coupling between A and B as discussed in the previous paragraph, with the same risks.

Note that there are variants of this request-response approach that exhibit the same basic problems. For example, rather than delivering the response via shared memory, B could deliver the response into a message queue owned by A. In that case, A would wait on that message queue, which would typically be a different message queue than the one used at the top of its event loop, to ensure that A only wakes up upon receipt of the response from B. But as long as A is blocked at this point in the code until the response arrives from B, the problems are essentially the same. It doesn't matter how A waits for B's response, as long as it is waiting here rather than returning to the top of its event loop. (I'll refer to this approach as "in-line blocking.") Yes, we've seen simple embedded applications without hard real-time requirements for which in-line blocking has been made to work. However, for complex, hard real-time, highly multi-threaded applications, it can be a recipe for problems down the road.

Here is another implementation of request-response communication that is also quite common in embedded systems based on our experience. Rather than blocking until the response from B is received, as in the previous example, a callback is used:


1 Request
Message queue
Message queue
THREAD A
THREAD B
2 B processes
3 A continues, and processes next msg
4 B delivers response to A via callback

As in the previous example, A sends a message to B that requests a service or an action of some sort (step 1). B then receives the request and performs the necessary processing (step 2). Unlike the previous example, however, A does not wait for the response from B. Instead, A continues to the top of its event loop to process the next message in its queue (step 3). When B has finished processing and is ready to send the response, it delivers the response using a callback (step 4). This appears to solve the numerous potential problems of the previous example that were caused by A blocking until it receives B's response. In this implementation, A is not blocked waiting for the response from B, so it can respond to other messages. And because it is not blocked waiting for B, the deadlock risk between A and B is mitigated. But, we've traded one set of problems for another. To see this, let's look at the implementation within Thread A:

```
THREAD A:

INITIALIZE
DO Forever
    Wait for a message
    CASE message ID
        Msg 1: Process message
               Send request to Thread B
               Break
        Msg 2: Process Msg 2
               Break
        Msg 3: Process Msg 3
               Break
             .
             .
             .
        Msg n: Process Msg n
               Break
        Invalid ID: Handle error
    END CASE
END DO

Callback (arg1, arg2, …)
{
   Process response
   from Thread B
}
```

As in the previous example, I assume that it is the receipt of Msg 1 that causes A to initiate the request-response communication with B.

The problem is in the red step within the callback. The callback is a function that is logically part of Thread A, since it performs the processing required by Thread A in order to handle the response from Thread B. But the callback is executed within the context of Thread B, not Thread A, because it is Thread B that calls it. One problem with this is that while Thread A processes a subsequent message in its queue (step 3 in the diagram), the callback can get executed asynchronously by Thread B. As a result, there can be concurrency issues if there is any data shared between the callback and the event loop code of Thread A (which is often the case). This has to be handled appropriately. Additional challenges can arise if, for example, Thread A is in the middle of processing a large collection of data when the callback executes, and if the response from B includes an update to a portion of that collection of data. Care would have to be taken to ensure that the appropriate atomicity of the collection of data is enforced.

Furthermore, if the event loop code of Thread A needs to know that the response from Thread B has been received, how is that accomplished? Typically, this is done by polling, that is, by Thread A checking a flag that is set by the callback to indicate that the response from B has arrived. But how often does A check the flag? This could be done whenever A wakes up due to the arrival of a message in A's message queue (unrelated to B's response). But that could result in unpredictable delays in A recognizing and processing B's response. Instead, A could wait for messages in its message queue only up to some maximum timeout value, and if no message is received within that timeout value then it could wake up to check the flag. These solutions require a tradeoff between responsiveness (how quickly A recognizes and processes B's response) and CPU utilization (how often A wakes up to check the flag). If A has tight deadlines, then it might need to wake up very often in order to ensure that it recognizes and processes B's response in a timely fashion. This could waste CPU cycles until B actually responds. As with the previous example (in-line blocking), one can make this work in a simple system (and we have seen people do it), but if this were used routinely in complex, hard real-time, highly multi-threaded applications, it likely would only be a matter of time before it caused a problem.

A Solution

Given the potential pitfalls of the above approaches, what is a better solution? The solution we have used successfully in many systems is to wait for ***all*** possible inputs at one place, that is, at the top of the event loop:

```
INITIALIZE
DO Forever
    Wait for all possible inputs
    CASE input type
        Input A: Process Input A
                 Break
        Input B: Process Input B
                 Break
        Input C: Process Input C
                 Break
            .
            .
            .
        Input n: Process Input n
                 Break
        Invalid input: Handle error
    END CASE
END DO
```

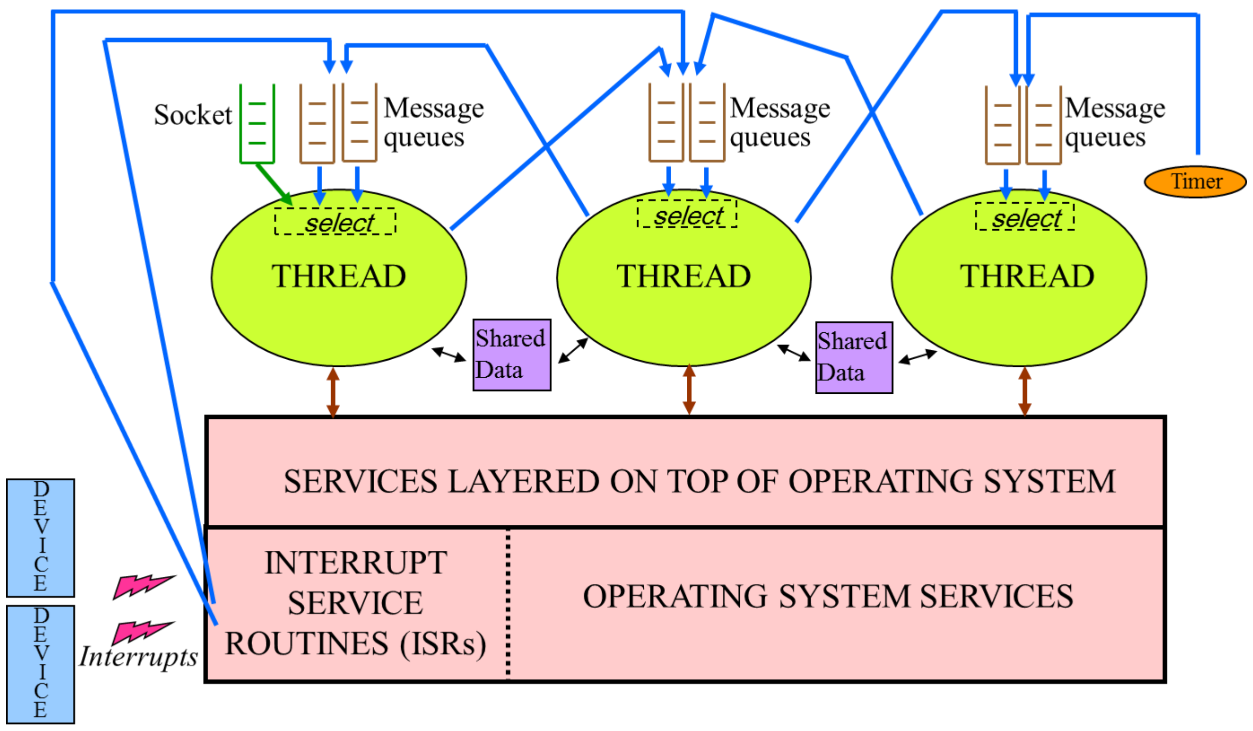
Now, it may seem obvious that this is a straightforward and safe approach. But this approach has at least two challenges, which often result in it not being fully implemented and consistently applied in complex systems. The challenges are:

1. Some operating systems and multithreaded languages may not have a mechanism to receive notification of all possible events of interest via a single call.
2. Even if #1 is solved, strictly adhering to this approach, and not resorting to in-line blocking (the first example above) or callbacks (the second example above), or other shortcuts, can require more complex programming (e.g., implementing state machines). For a variety of reasons, including the intense schedule pressure that is often present during the development of these systems, it can be tempting to resort to in-line blocking, callbacks, or other shortcuts in order to avoid designing and implementing the necessary state machines or similar mechanisms. (I'll say more about this below.)

Regarding item #1, waiting for all inputs at one place can be challenging, because the inputs that may be received by a thread can include:

- Messages from other threads
- Callback notifications (if callbacks are used, then they must be converted to messages delivered in message queues)
- Timer expirations (if the OS delivers timer notifications by callback, then they also must be converted to messages delivered in message queues)
- Network messages (e.g., messages arriving over TCP or UDP sockets)
- Network connection requests (e.g., TCP "listen" events)
- Event notifications or data from hardware devices, including
  - Notifications from interrupt service routines (these also need to be converted to messages delivered in message queues)
  - I/O completion events from device drivers (these also need to be converted to messages delivered in message queues)

Utilizing the "select" system call found in UNIX, Linux, and VxWorks, and translating the appropriate events from the above bulleted list (callbacks, timer expirations, and hardware event notifications) into messages delivered in message queues, one can build a mechanism that allows a thread to wait for all inputs at one place. The resulting threading model is shown in the following diagram:



In this model, each thread receives all of its inputs, and thus blocks waiting for these inputs, at only one place, namely, the top of its event loop. The inputs arrive in a combination of message queues and sockets, all of which are handled by the "select" system call. (Different operating systems refer to "message queues" differently, but all of the operating systems we have used for these systems included some form of message queues together with a "select" call that provided the necessary functionality. In Ada, which includes support for tasks within the language itself, we built our own message queue mechanism because, at least at that time, the only inter-task communication mechanism supported by Ada was the rendezvous.) All asynchronous events, such as interrupt notifications and timer expirations, are converted to synchronous events by inserting messages into message queues.

Regarding item #2 above, we have seen a number of systems that start out with this threading model, but then ultimately get polluted over time by violations of the model. At some point, someone with a nicely structured thread that properly handles a large variety of input message types realizes that, for just one of those input message types, they need request-response semantics, that is, they need to request the services of another thread and receive a response. At that point, the temptation to use in-line blocking, callbacks or some other shortcut, for just that one type of message, can be quite strong, especially if project deadlines are looming. But that's a slippery slope which, for all the reasons discussed above, should be avoided. That one exception is likely to cause problems down the road. And once one exception is made, others may follow, and over time the nicely structured system with nicely structured threads can become something else entirely.

The proper solution, when the need within a thread for request-response semantics first arises, is to bite the bullet and modify the code for that thread to handle the response (which in our earlier examples would be the response from Thread B) as just another message received at the top of the event loop along with all the other types of messages that can be received. In order to do that, the thread needs to retain state. That is, it needs to know that a request to Thread B is outstanding, and therefore a response from Thread B is expected. It may receive a number of other messages in its message queue prior to receiving the response from Thread B, in which case it must handle them appropriately. Complications can arise if the ways in which some of those other messages must be handled depend upon whether or not a request to B is outstanding.

For example: What if a message is received that requires sending another request to B before the response to the first request has been received from B? Is it OK to send a second request to B under these circumstances? If it is not OK, then do we just ignore the need to send the second request, or instead do we queue the second request and send it to B once the response to the first request is received from B? If it is OK to send a second request to B before the response to the first request has been received from B, then what if B responds to the second request before responding to the first request? Also, is there a limit to the allowable number of outstanding requests to B? What if there are potential requests to another thread, C, that should or should not be sent depending upon whether certain requests to B are outstanding? Etc.

Confronting such complications is what often motivates people to take shortcuts. In-line blocking, for example, eliminates the concurrency between A and B that can cause many of these complications. However, dealing with these complications, by creating the appropriate state-based logic and addressing any dependencies between the messages received and the current state of the thread, is the better way to go, based on our experience. Any bugs in the state-based logic are likely to be easier to identify and fix than the timing and concurrency errors that can arise if instead shortcuts such as in-line blocking or asynchronous callbacks are used. And any bugs in the state-based logic are also more likely to be isolated to the functionality of the thread, whereas the timing and concurrency errors associated with the shortcuts described earlier are more likely to have global ramifications. Experience has shown that by carefully analyzing the various possible states and the associated complications, by imposing constraints on the number of allowable states based on this analysis, and by logging at run-time any deviations from the allowable states that may occur (i.e., any bugs), the state-based approach is eminently manageable and is safer and more reliable than the shortcuts.

In summary, in a complex embedded system that relies heavily on concurrency, giving in to the temptation to take shortcuts such as in-line blocking or callbacks for request-response semantics, or otherwise deviating from the principle of waiting at the top of the event loop for all inputs, should be resisted. It can lead to subtle concurrency and timing bugs, which can be very difficult to track down and fix. Although waiting at the top of the event loop for all inputs without exception can result in seemingly more complex code due to the need to retain state within the thread, that is the preferred approach. It will result in a more stable, robust, maintainable and extensible system, which will also exhibit more predictable performance.

Epilogue: A Bug Involving the "select" System Call - Mars Pathfinder Priority Inversion

I'll close with a brief description of one of the many challenging bugs encountered over the years involving multithreaded systems. I have selected this particular bug because:

- It occurred even though, to the best of my recollection, we attempted to follow the general threading model guidelines described in this paper. It exposes a related, but different, vulnerability: a threading vulnerability due to the use of off-the-shelf software.
- The bug is related to the "select" system call, which is a key element of the approach suggested in this paper.
- This was a high-profile bug. It occurred on the Mars Pathfinder spacecraft, and it received a lot of publicity and interest within the embedded systems community at the time (1997).
- During my presentation, on which this paper is based, this bug generated a fair amount of interest and discussion among the attendees.

(For a more detailed description of this bug, please see the authoritative account written by Glenn Reeves, who led the Pathfinder flight software team and has continued to accomplish great things at JPL: http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html.)

After landing on the surface of Mars, the Pathfinder spacecraft started experiencing unexpected reboots of the flight computer because one of the VxWorks tasks (threads) was detecting that a real-time deadline had been missed. The software was designed to treat this missed deadline as a fatal error and force a reboot. Back at JPL, the team set out to reproduce the problem, and within 18 hours of testing they succeeded. (By then, I was working elsewhere, so I was not directly involved in the debugging effort, although I was in touch with members of the team.) Because of the instrumentation we had built into the software, once the problem was reproduced, it was straightforward to determine its cause. To our surprise, it was a case of priority inversion.

This was quite unexpected to those of us who had developed the software. We were well aware of the risks of priority inversion, and we had gone to great lengths to ensure that it could not happen in our software (or so we thought). We selected VxWorks as the RTOS for the spacecraft partly because of its support for priority inheritance, to prevent priority inversion. We tried to be meticulous in using semaphores that had the priority inheritance option enabled wherever there was a risk of priority inversion. We wrote a tool so that we could see VxWorks' priority inheritance in action, to make sure that it was working correctly. Indeed, when we ran the tool, we could see, over and over again, situations in which priority inversion would have occurred were it not for our use of semaphores with priority inheritance.

So what went wrong? As it turns out, there was a semaphore within VxWorks itself that was not utilizing priority inheritance. The semaphore was used by the VxWorks “select” system call, on which we relied. We had been unaware of that particular semaphore because it was hidden behind the “select” call and thus was not exposed to the user. Once this was discovered, the team contacted Wind River (the VxWorks vendor) and worked with them to make the necessary modification so that the semaphore within “select” utilized priority inheritance. The modification was made, the modified software was uploaded to the spacecraft, and the unexpected reboots on the spacecraft disappeared. (I’ve glossed over some details here regarding the number of semaphores potentially affected by the fix – see Glenn’s account for those details.)

One of the lessons learned from this experience is that if a system relies on off-the-shelf software, then making unverified assumptions about how that software behaves can be dangerous. In this particular example, we assumed that there were no priority inversion risks within the off-the-shelf software. That assumption proved to be false.

Another lesson reinforced by this experience is the importance of the liberal use of error checking (assertions) and instrumentation **in the production code**. Without both of these, the outcome in this particular case might have been very different. That is another topic from my presentation that is the possible subject of a future paper.