Emulating Operating System Synchronization

Ruediger R. Asche
Software Consultant

October 1997

Abstract

This article describes ways to emulate the synchronization mechanisms found in operating systems (that is, interrupt-driven concurrency and appropriate mutual exclusion primitives) using Microsoft Windows synchronization mechanisms. The associated sample application OSPORT implements a very rudimentary operating system that employs the techniques discussed.

Introduction

Most modern operating systems support multithreading and the synchronization schemes related to multithreading. However, the primary concurrency mechanism on the system level of most operating systems is hardware-supported concurrency between user processes and interrupt handlers.

Those of you who write device drivers will be more familiar with interrupt masks and interrupt levels as means of synchronization than with semaphores, events, and threads, which are the domain of low-level application programmers. During a recent project, I had to jump back and forth between being an application and a device driver programmer: My assignment was to port the TCP/IP stack of a public domain-provided operating system to a different platform. Unfortunately, the machine I was to port the operating system to was not yet available when the project started, so I first decided to port the code I needed up to Windows and then back to the final operating system. The main challenge buried in this approach was that the TCP-stack would execute as a user-mode DLL as opposed to a kernel module. Thus, the synchronization mechanisms found in the original code—statements with names such as SetPriorityLevel(NETSOFTINTERRUPT)*—had to be mapped to user-mode synchronization primitives.

This article is sort of a case study of implementing an operating system as a user-mode process. It's up to you to decide whether this is a useful endeavor or not; but regardless of what you decide, you'll hopefully learn quite a lot about synchronization (in particular, new synchronization primitives). The first section of the article explains the inner workings of hardware-supported synchronization, and the second section explains how I successfully implemented these using application-level synchronization primitives. The third section discusses my sample operating system G.R.E.A.T., which I implement as an application.

How Hardware Handles Synchronization

Note

In the following discussion, I will assume "standard" hardware, that is, a CPU that follows the hierarchical interrupt architectures adapted by most of today's processors such as the Intel x86 families, or the Motorola 68xxx families, or the VAX family of microprocessors. I use the generic term toaster processor to depict such a gizmo.

A toaster processor executes only one instruction at any given time, and during normal processing, the sequence of instructions processed is determined by the instructions: Unless the instruction implies a command that branches to somewhere else, the next instruction to be processed is the instruction that immediately succeeds the current instruction.

Concurrency in a toaster processor is implemented via interrupts. Given certain circumstances, the toaster processor can decide to suspend the current stream of execution that we discussed above, and transfer control to a different routine called an "interrupt handler" (hereafter frequently referred to as "ISR" or "interrupt service routine"). When the interrupt handler has finished its processing, control is returned to the instruction stream that was previously interrupted. To the original stream, the execution of the interrupt handler is perfectly transparent; the original stream is not aware of the fact that it was suspended and re-awakened. This idea of concurrent execution is similar to application-level concurrency only in that streams of execution will not be aware of any concurrency.

Interrupt handlers can be nested. That is, if a user-mode application is interrupted, and an interrupt handler gains control over the CPU, that interrupt handler becomes the new current stream of execution, and another interrupt handler can kick in and interrupt the first interrupt handler.

The ordering of interrupt handlers normally follows a priority scheme, that is, interrupt handlers can only be interrupted by handlers that have a higher priority. Priorities are implemented on the hardware level, but can be defined by software. We will work out an example, but before that, let's become a little bit more precise about what constitutes an "interrupt." An interrupt can normally be triggered by one of two conditions:

- A *hardware interrupt* is triggered by some event that is controlled by a hardware device. For example, whenever a character arrives at a serial communications port, the port will generate a hardware interrupt that can be processed by the operating system.
- A *software interrupt* is triggered by a software instruction. Frequently, software interrupts come in two flavors: explicit software interrupts (that is, machine instructions that translate to "invoke the interrupt handler") and implicit software interrupts (for example, when a machine instruction that does a division encounters a 0 in the divisor, a software interrupt handler can be invoked to handle a division by 0 error).

Unfortunately, over different platforms, the terminology concerning interrupt handlers is often unclear; sometimes all software interrupts are labeled "traps," sometimes only implicit software interrupts are called "traps" as opposed to "interrupts," which depict explicit software interrupts on those architectures, sometimes all interrupts are named "traps," and so on. In this discussion, we'll stick to the distinction between hardware interrupts, explicit software interrupts, and implicit software interrupts.

Now let's look at an example. My sample operating system G.R.E.A.T. (this is fictitious, in case you haven't figured that out yet) defines seven interrupt levels in the following order:

There is deliberation behind this ordering, of course. The hardware clock interrupt handler has the highest priority for two reasons: First, whenever this interrupt executes, it can update the software clock (which should be as exact as possible) as well as keep track of such things as timeouts. Second, this interrupt handler will typically execute very briefly so that other time-critical interrupt handlers aren't interrupted for an extensive amount of time. Finally, the data that the hardware clock interrupt handler manipulates are normally not shared by other interrupt handlers and processes in such a way that synchronization is necessary.

Similar reasoning applies to all of the above levels; in general, interrupt handlers that execute less time-critical code than others are placed lower in the hierarchy than those other interrupt handlers. Hardware interrupts from devices that process very fast I/O should be assigned high priority so that the system doesn't run a risk of losing input.

For the sake of this discussion, we can define a user-process as the lowest-level interrupt handler that executes whenever no other interrupt handlers execute. Note that this is not a "correct" definition, but in the discussion to come, I will frequently state things like "a lower-priority interrupt handler is interrupted by . . . ." In order not to have to write "a lower-priority interrupt handler *or a user process* is interrupted by," I imply that a user process behaves a lot like a very low-priority interrupt.

Now, what is a "software clock interrupt," as opposed to a "hardware clock interrupt"? Let's look at typical (pseudo-) code for a hardware interrupt handler:

```
hardwareinterrupt()
{
 read hardware clock and store value in soft clock data structure();
 do other brief time-critical processing();
 submit software clock interrupt();
}
```

This code is executed whenever the hardware clock interrupts the processor (normally at periodic intervals). The important part is the "submit software clock interrupt" instruction, which is an explicit software interrupt: The hardware interrupt handler requests that the code for the software clock interrupt be executed. What is that rubbish?

Well, here's the beauty. Remember that the clock hardware interrupt has a higher priority than the clock software interrupt, so at the point where the "submit software clock interrupt" instruction is submitted, there is no chance for the clock software interrupt handler to execute until later on. The hardware will mark the software interrupt as pending, and as soon as there is no higher-priority interrupt executing, the software clock interrupt handler can do its thing. (Actually, it's not that simple, but we'll get to the details in a second.)

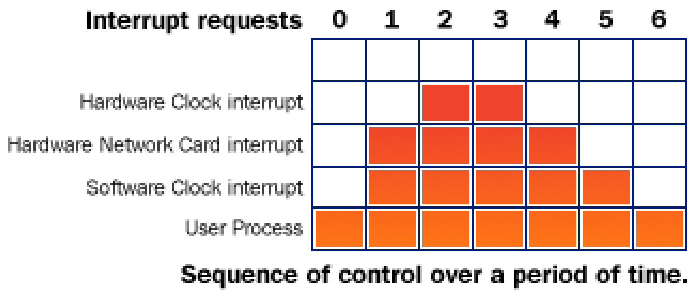

Figure 1. An interrupt scenario

Let's look at Figure 1, which depicts a typical sequence of control over a period of time. The user process is running and gets interrupted by a level 3 network card interrupt (1). That interrupt handler in turn is interrupted by a hardware clock interrupt (2), which, as we saw before, submits a software clock interrupt (3). As we discussed before, the software clock interrupt is marked pending by the hardware. The hardware clock interrupt finishes and returns control to the network card interrupt handler (4).

Remember that the network card interrupt handler still has higher priority than the software clock handler, so the clock software interrupt handler will still not execute. As the level 3 interrupt handler terminates, it would normally return control to the user process, but at this point, there is no interrupt handler at a level higher than the software clock interrupt executing, so the software clock interrupt handler gets to execute (5). Once this interrupt handler finishes what it's doing, the user process continues to execute unless other interrupts have occurred or been marked pending in the meantime (6).

This is most of what you need to know about concurrency on the hardware level of a toaster processor. Have you ever wondered how I/O subsystems work in operating systems? Well, here's the deal in a nutshell. Assume a user process attempts to read 20 bytes from a serial communications port. The process submits a "read" call to the operating system, which in effect means that the operating system asks whether there are 20 characters or more in the serial port's input buffer. If the answer is yes, 20 characters are removed and placed into the user process's buffer and the process continues happily. If there are fewer than 20 characters available, the requesting process is put to sleep.

The serial port's interrupt handler is invoked the next time a character arrives at the port, and the interrupt handler (or possibly some other intermediate interrupt handler which is invoked in turn) will know that a user process is waiting for characters from the port. Thus, the newly arrived character is placed in the process's buffer, and as soon as there are 20 buffers in the user process's input buffer, the process will be resumed. And that's it.

Well, not quite. Here's a glitch. Below is some real-life code, although it is slightly rewritten to protect the innocent. This code is basically adapted from a network protocol suite and implements the control flow that passes network packets on to a protocol.

```
SoftIntHandlerForNetCard()
{
 Place character from input device into buffer();
 look at buffer gathered so far and determine whether the characters
                                       make up a complete packet();
 if we have a packet together, insert packet on protocol's input queue
                             and submit protocol software interrupt();
}

SoftIntHandlerForProtocol()
{
 Remove a packet from input queue();
 look at packet, decode, and forward to whoever needs it();
}
```

Needless to say, the software interrupt handler for the protocol has a lower priority than the handler for the network card. The problem here is the following If you are familiar with concurrent programming, you will have noticed that there is a problem here: Assume the interrupt handler for the net card has scheduled a software protocol interrupt, and eventually, the software interrupt for the protocol executes. As **SoftIntHandlerForProtocol()** attempts to remove a packet from its input queue, the software interrupt handler for the network card interrupts again, and two streams of execution try to party on the same queue concurrently, which is a safe recipe for ugly disaster.

The way to handle this issue is for the lower-priority interrupt to inhibit higher-level interrupts temporarily. For example, a correct way to write the software interrupt handler for the protocol would be as follows:

```
SoftIntHandlerForProtocol()
{
 SuppressAllInterruptsUpToNetCard();
 Remove a packet from input queue();
 RevertBackToPreviousLevel();
 look at packet, decode and forward to whoever needs it();
}
```

Note that this code in practice implements mutual exclusion between the software interrupt handlers for the protocol and the network card, respectively, for the code that parties on the protocol input queue. This mutual exclusion may not be obvious, but remember that by definition, as long as an interrupt handler executes, no lower-priority interrupt handlers have a chance to execute. On the other hand, as soon as the protocol interrupt handler gets to execute **SuppressAllInterruptsUpToNetCard()**, we know that no higher-priority interrupt handler can be executing (otherwise the code wouldn't be executing in the first place), and once this call returns, the higher-priority interrupt handler that competes with the protocol interrupt handler for access to the protocol's input queue will not be allowed to execute until the lower-priority interrupt has finished working on the buffer.

The net effect of this mutual exclusion process is similar to what mutexes can accomplish on the application level, except for two important differences:

- The mutual exclusion is "asymmetric" in that the higher-level interrupt implicitly prevents the lower-level interrupt handler from executing, whereas the lower-level interrupt handler explicitly inhibits ("masks off" is the official terminology for this kind of thing) the higher-level interrupt.
- There are the mutual exclusion process has side effects. For example, when a lower-level interrupt handler masks off interrupts, it masks off all levels in between itself and the highest level that the mask call specified. There is no way for, say, a G.R.E.A.T. level 4 interrupt handler to say, "I don't want level 2 interrupts to go through, but level 3 interrupt are fine." Another side effect is that higher-level interrupt handlers always inhibit all lower-level interrupts, regardless of whether those lower-level interrupt handlers execute critical code or not.

As it turns out, the hardware-controlled synchronization mechanism is no more powerful than a software synchronization scheme that provides adequate primitives. It would be possible for a microprocessor to implement interrupt handlers using multithreading and on-board arbitration code. How to emulate hardware synchronization in software using (in this example) the synchronization primitives is the subject of the rest of this article.

Emulating Hardware

In distinction to the synchronization scheme discussed so far, the Windows API defines threads as the main units of concurrent execution. Thus, we will model each interrupt handler by a separate thread that executes the following pseudo-code,

```
void InterruptThread(int iLevel)
{
 while(1)
 {
   WaitForIntSubmitted(iLevel);
   StartISRProcessing(iLevel);
   intHandler();
   StopISRProcessing(iLevel);
 };
}
```

where intHandler() is the interrupt handler supplied for the appropriate level. **StartISRProcessing()** is a function that ensures two things: (1) the interrupt mask is low enough for this thread to proceed; and (2) all ISRs and user processes running on levels lower than iLevel do not execute when **StartISRProcessing** returns. As a side effect, **StartISRProcessing()** may suspend the current thread until the interrupt mask has been lowered to a level appropriate to the caller.

Any software or hardware that needs to trigger an interrupt calls the function

```
SubmitInt(iLevel);
```

and we also need a function that can mask interrupts. We will use the semantics of Berkeley Software Distribution (BSD) UNIX for this function group. BSD UNIX defines the following function set for interrupt masking:

```
int splXXX(), where XXX is one of a fixed number of string constants,
                                      such as net, tty, imp, or high;
int splYYY(), where YYY is a fixed numeric level from 0 to the level
                            supported by the target operating system;
void splx(int iLevel). The x is NOT a metasymbol here; the function is
                                                 really called splx().
```

The functions are to be used as follows: A section of code that needs to mask interrupts of a given level is wrapped into calls to splXXX() or splYYY() in the beginning and splx() in the end, like so:

```
{
  int s=splnet(); /* Inhibit all interrupts between the current executing IS
  ...     /* Do the processing that relies on net interrupts being masked ou
  splx(s);  /* Restore the previous interrupt level. */
};
```

Each of the functions returns the current interrupt level, and the automatic variable *s* is used to save a previously retrieved level. At the end of the critical code, the interrupt mask is restored to the previous level. Needless to say, even without knowing how these functions work exactly, it is tricky to use them correctly. Those of you who have read a lot of BSD UNIX system code will probably have come across a few occurrences of "sloppy spl-ing," where an interrupt mask level has been set but is never restored to the previous level due to a premature error exit in which the corresponding splx() is not called. Likewise, it is easily possible to deadlock an operating system due to a careless sequence of masking calls.

Anyway, we will use the generic call

```
int SetISRMask(int iLevel);
```

to set and retrieve an interrupt mask.

The API we need to implement to emulate our hardware with software is very small; namely, we have to implement the interrupt levels themselves as well as the functions **WaitForIntSubmitted**, **SubmitInt**, **StartISRProcessing**, **StopISRProcessing**, **SetISRMask**, plus a few functions that set up the entire system.

Easier said than done, of course. As usual with concurrent programming, the hard part is doing it right; it took me about ten attempts to write a functioning set of primitives. In the meantime, I encountered everything that makes the life of a concurrent programmer fun: Unpredictable and irreproducible data corruptions, deadlocks, lockouts, and other surprises. Here's a version that works. It's probably not the only working solution, and by no means the most efficient one, but it works.

A first attempt to tackle the priority hierarchies would try to make use of the built-in priority scheme and map the threads that model the interrupts to respective priority levels. However, it doesn't work like this, for the following reasons:

- There are only five available distinct priority levels per process, which is normally not enough to model a sufficiently complex toaster processor.
- Priorities are not static; that is, priorities can change dynamically over time due to dynamic priority boosting.

Thus, we model the priority preemption scheme through thread suspension and resuming. Let's first define the data structures we need to keep the entire scheme together:

```
typedef struct ISRINFO
{
 int iStatus;
 HANDLE hThread;
 HANDLE hEvent;  /* Determines whether ints can be processed.
```

```
                      Set when the iStatus == STATUS_FREE
                      unset otherwise. */
  ISRProc CriticalInit; /* Contains the critical initialization code. */
  ISRProc ISRFunc;      /* Contains the code to be executed at interrupt
                                                                 time. */
 } ISRINFO, *PISRINFO;

 struct ISRINFO isrlevels[MAXTHREADS];

 int g_CurrentMask;

 HANDLE g_hISRMutex;

 // The interrupt is not masked.

 #define STATUS_FREE 0

 // The interrupt is masked -- suspend ISR.

 #define STATUS_MASKED -1
```

The global array isrlevels has one entry per interrupt handler, where the position of an ISR implies its relative priority (ISRs that come earlier in the array have a higher priority). The iStatus member of the ISRINFO structure can be either STATUS_FREE or STATUS_MASKED. The associated event object hEvent will be used to guard entry to the interrupt handler (we will see how this works in a second). The hThread member contains the identification of the thread that executes the **InterruptThread()** function of the given level. The CriticalInit and ISSRFunc members are pointers to functions that execute initialization code for the given ISR and the ISR code itself, respectively.

Finally, the g_hISRMutex object (which is created at system-initialization time) is a simple mutex object that is used to arbitrate access to the isrlevels table itself.

Let's look at **StartISRProcessing**. I have removed error-handling logic to make the code more readable, but as usual, you’re well advised to do better than me in real life.

```
int StartISRProcessing(int iLevel)
{
 int iReturn;
 TryAgain:
 WaitForSingleObject(g_hISRMutex,INFINITE);
 switch(isrlevels[iLevel].iStatus)
 {
  case STATUS_MASKED:
   SignalObjectAndWait(g_hISRMutex,isrlevels[iLevel].hEvent,
                       INFINITE,FALSE);

   goto TryAgain;
  case STATUS_FREE:
   iReturn=SuspendEveryoneBelowMe(iLevel);
```

```
      ReleaseMutex(g_hISRMutex);
      break;
   };
    return iReturn;
   }
```

The basic idea of the code is very simple: If our level is masked out, wait on the mutex isrlevels[iLevel].hEvent until the interrupt becomes unmasked. If our level is unmasked, suspend all threads that execute on lower priorities. Because each interrupt handler that wishes to execute interrupt code must first call **StartISRProcessing()**, the suspending of threads can be done in a controlled manner: The global mutex object g_hISRMutex ensures that only one interrupt thread can be in masking-related code at any time.

The only gotcha here is that if it weren't for the cool **SignalObjectAndWait** function, this code would be hard to implement, because we can't release g_hISRMutex before waiting on the level-specific event; but on the other hand, we can’t first release the global mutex and then wait on the event object because another ISR may hop in between and change the masking status of our level. Using **SignalObjectAndWait**, we do both the signaling and waiting atomically, which ensures that the status flag and the state of the event are always consistent.

Why is there a goto TryAgain after the **SignalObjectAndWait** call? Doesn't a successful return from the **SignalObjectAndWait** call imply that our level is unmasked, and we can proceed? Theoretically yes, but unfortunately, we don’t own the g_hISRMutex anymore once we return from the **SignalObjectAndWait** call, but we need to own the mutex to prevent other ISRs from trying to enter ISR processing. Worse, we can’t simply claim the mutex here because if we did so, some other ISR may have kicked in after we returned from **SignalObjectAndWait** and before we did enter the mutex, so our level may be masked off again. Thus, we need to jump back until we own the global ISR mutex *and* our level is free. Once those two conditions are met, it's safe to suspend all intermediate ISRs and user threads, like so:

```
int SuspendEveryoneBelowMe(int iLevel)
{
 int iLoop;
 for(iLoop=iLevel+1;iLoop<MAXTHREADS;iLoop++)
  if (isrlevels[iLoop].hThread)
   SuspendThread(isrlevels[iLoop].hThread);
 return 0;
}
```

Note that this scheme is a little weird because it assumes that user threads also have entries in the isrlevels array, although they are strictly speaking not ISRs.

The counterpart to **StartISRProcessing** is **StopISRProcessing()**, which reverses actions taken by **StartISRProcessing**:

```
int StopISRProcessing(int iLevel)
{
 int iReturn;
 WaitForSingleObject(g_hISRMutex,INFINITE);
 iReturn = ResumeEverybodyBelowMe(iLevel);
 ReleaseMutex(g_hISRMutex);
 return iReturn;
}

int ResumeEverybodyBelowMe(int iLevel)
{
 int iLoop;
 for(iLoop=iLevel+1;iLoop<MAXTHREADS;iLoop++)
  if (isrlevels[iLoop].hThread)
   ResumeThread(isrlevels[iLoop].hThread);
 return 0;
}
```

This code is fairly straightforward and doesn't need a lot of annotation; however, here's one bit of trivia that needs to be mentioned. Strictly speaking, it is not necessary to match splXXX() and splx() calls on a toaster-processor–based architecture, because there is no harm done in masking off interrupts that are already masked; the toaster hardware has no "memory" in which masking operations are kept. Normally, there is only one global register that contains masking information, and as soon as an appropriate operation has masked off an interrupt level, that level is masked, regardless of how many times unmasking operations have been submitted.

On the other hand, calls to **SuspendThread** and **ResumeThread** must be matched because an implementation is required to keep track of thread suspension, and a thread that has been suspended *n* times must also be resumed *n* times before resuming execution. Thus, operating system code that uses the Resume/Suspend scheme to implement interrupt masking must match the splXXX with splx() calls.

The remaining function in the synchronization set is **SetISRMask**, which, as we discussed before, is the one-size-fits-all solution for interrupt mask manipulation:

```
int SetISRMask(int iLevel)
{
 int iLoop;
 int iCurrentMask;
 int iReturn;
 WaitForSingleObject(g_hISRMutex,INFINITE);
 iCurrentMask=g_CurrentMask;
 g_CurrentMask = iLevel;
 for (iLoop=0;iLoop<MAXTHREADS;iLoop++)
  switch (isrlevels[iLoop].iStatus)
  {
   case STATUS_FREE:
   if (iLoop>=g_CurrentMask)
```

```
      {
        isrlevels[iLoop].iStatus = STATUS_MASKED;
        ResetEvent(isrlevels[iLoop].hEvent);
      };
      break;
      case STATUS_MASKED:
      if (iLoop<g_CurrentMask)
     {
        SetEvent(isrlevels[iLoop].hEvent);
        isrlevels[iLoop].iStatus=STATUS_FREE;
      };
      break;
   };
   ReleaseMutex(g_hISRMutex);
   return iCurrentMask;
   }
```

Again, this code is rather straightforward. Note that the global variable g_CurrentMask needs to be initialized to the lowest level. **SetISRMask** basically does two things: (1) the function exchanges g_CurrentMask and the level passed in atomically, and (2) all event objects associated with a higher priority than the one requested are set, and the remaining event objects are reset. The event objects are naturally created as manual-reset objects.

A Sample Operating System

The code sample that accompanies this article, OSPORT, implements our little G.R.E.A.T. operating system. OSPORT is our typical compromise between readability of code and usefulness. There are no "user processes" provided with the sample code, only four interrupt handlers that conspire to do something completely silly and impractical: Every few seconds, a dot is displayed on the screen unless the user of OSPORT types in characters. When enough characters have been entered, OSPORT transforms those characters to uppercase and displays them on the screen.

G.R.E.A.T. implements four little interrupt handlers: A hard clock interrupt, a soft clock interrupt, a hard tty interrupt, and a soft tty interrupt. Of course, the term "hard interrupt" is misleading; there is no way for a user-mode application (which OSPORT is, at the end of the day) to access hardware interrupts. The "hard" clock and tty interrupt handlers in OSPORT are fed from background threads that pretend to be hardware devices: The "clock" interrupt thread calls **SubmitInt()** periodically, and the "tty" interrupt thread does so whenever a character has arrived in the input stream (this is actually implemented via **scanf**). The interaction between the respective soft and hard interrupt handlers is much like that in a real operating system: The "hard clock" interrupt synchronizes its own time with the system time and then submits software interrupts for less time-critical code to execute delayed, and the "hard tty" interrupt pre-buffers characters from the input device and then submits a soft tty interrupt which, in a real operating system, would serve user mode processes that wait on input from the respective serviced device.

In order not to complicate the discussion, I have so far omitted the functions that are needed to set up and initialize the data structures and threads. In the code sample, you'll find the functions **InitOS**, **RegisterThreadAsISR**, and **RegisterUserThread** that are needed to set up the ISRs and associated data structures. I do not go into too much detail here because most of the code is very straightforward Windows API programming.

Although the code sample is rather simple and rudimentary, I used the very same API set to port major portions of a full-blown multi-user operating system to proprietary hardware. If you are interested in some of the more subtle issues that arose during the port, please let me know, and I'll be happy to follow up on this article.

© 2015 Microsoft