Technotes

Download  Acrobat file (K)	Download  AppleWorks file (50K)

Traditional Device Drivers: Sync or Swim

or, "I've Got the vSyncWait Blues"

Technote 1067 **September 1996**

As with any software system, the Mac OS works fine just as long as everybody plays by the rules. For general Toolbox operations, these rules are fairly easy to understand. However, traditional Mac OS device drivers (DRVRs), by their very nature, are complicated programs, and the rules that govern their behavior are harder to understand.

This Technote discusses one of the dangers of writing a traditional Mac OS device driver, namely, the possibility of **deadlock** when calling another device driver synchronously from a device driver. It demonstrates several ways this deadlock can occur, and then goes on to describe how to avoid the possibility of deadlock.

The Note is intended for developers writing traditional Mac OS device drivers, especially drivers which call other device drivers, and for those with a morbid curiousity about the internals of the traditional Mac OS.

Two Rules That Govern Device Drivers

There are two important rules about writing traditional Macintosh device drivers.

Rule #1

If your device driver can be called asynchronously and you call another device driver, you must call it asynchronously.

If you don't follow **Rule #1**, you run the risk of deadlocking the system, with catastrophic results for the user.

Note:

> According to Andrew S. Tanenbaum, in *Modern Operating Systems*, "A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause." For example, if I can't finish this Technote until I have a reliable version of your disk driver, and you can't make your disk driver reliable until you have read the finished Technote, we are deadlocked. Humans are good at avoiding and recovering from deadlocks, but computers are not.

Rule #2

If your device driver can be called asynchronously, always operate as if you are being called asynchronously .

In other words, you should not test to see whether this operation is synchronous or asynchronous, and do

different things in each case.

If you don't follow **Rule #2**, the system might trick you into doing illegal things at interrupt time, such as breaking **Rule #1**.

Proof by Example

This section describes two examples of how breaking the rules will cause the system to deadlock. Both examples rely on the following scenario:

You have written a block device driver that fetches disk blocks over the network using the "classic" AppleTalk device drivers. When you use the Finder to copy files to the disk image, the system will sometimes deadlock in a routine called vSyncWait*. This is much more common when running Open Transport than when running classic networking.*

I chose this example because recently I've helped a number of Macintosh developers who are writing such drivers. These people broke the rules and everything seemed to work fine. Unfortunately, Open Transport significantly changed the way AppleTalk is implemented, and the authors suddenly discovered the penalty of breaking the rules: unpredictable system deadlocks.

When the Finder copies a file, it makes a sequence of chained ioCompletion routines (see "Asynchronous Routines on the Macintosh," *develop* 13) that reads chunks of data from the source volume and writes them to the destination volume. The Finder actually makes File Manager calls but, in cases like this, the File Manager effectively passes these calls straight through to the appropriate block device driver.

The next two sections show how your block device driver can deadlock the system by not following these rules.

An Example of Breaking Rule #1

Imagine that your device driver breaks **Rule #1**, i.e., it calls another device driver synchronously. The following is a step-by-step description of how this leads to system deadlock:

1. The Finder calls PBReadAsync to read a chunk of the file from the local hard disk. This invokes the File Manager, which invokes the device driver, which invokes the SCSI Manager, which invokes the SCSI hardware.

2. The SCSI operation completes. The SCSI hardware interrupts the CPU to inform it of this. The interrupt service routine calls the ioCompletion for the SCSI device driver, which calls the ioCompletion routine for the File Manager, which calls the ioCompletion for the Finder. Because of the nature of the interrupt service routine, interrupts are still disabled at this point.

3. The Finder's ioCompletion routine calls PBWriteAsync to write the data to the volume mounted on you block device driver. This invokes the File Manager, which invokes your device driver.

4. Your device driver calls the AppleTalk device driver synchronously. This is Request A. Unfortunately, the AppleTalk device driver is busy fielding another request (Request B) from a completely independent process, and queues your request waiting for Request B to complete. Your code sits inside the Device Manager's vSyncWait loop, waiting for the Request B to complete.

5. The prior networking operation (Request B) completes. The networking hardware attempts to interrupt the CPU to let it know that Request B has completed. Unfortunately the CPU is sitting inside vSyncWait with interrupts disabled -- they were disabled by the SCSI hardware when it interrupted the CPU in step 2 -- and doesn't notice the interrupt from the networking hardware. The system is deadlocked.

The basic problem here is that traditional Mac OS device drivers are single threaded, i.e., they can only handle a single request at a time. If you make a second request and the driver is busy and you don't give the driver the ability to complete the request, you deadlock.

An Example of Breaking Rule #2

Imagine that your device driver breaks **Rule #2**, i.e., it tests whether it's being called synchronously or asynchronously, and behaves differently in each case. The following is a step-by-step description of how this leads to system deadlock:

1. The Finder calls PBWriteAsync to write a chunk of data to your block device driver (Request A). You recognise this request is asynchronous (by testing ioTrap in the ParamBlock), send it to AppleTalk asynchronously, and then return to the Finder.

2. Another process calls PBWriteSync to write some data to your driver (Request B). Because your driver is currently busy, this request gets queued in your driver's queue.

3. The AppleTalk driver calls your ioCompletion routine to signify that your request is done. In response to this you call jIODone. jIODone completes Request A (and calls the Finder's ioCompletion routine) and then checks your driver's queue for any more pending requests. It notices that Request B is pending, and it calls your device driver's Prime to start that Request B. Remember that you are still running at interrupt time.

4. Your device driver is called to start Request B. It tests ioTrap, notices that the Request B is synchronous, and so calls AppleTalk synchronously. At this point your driver is calling a device driver synchronously at interrupt time, and you can deadlock the system as described in the previous section.

The basic problem here is that the ioTrap word in a ParamBlock only denotes whether the request was made synchronously, not whether the request is being executed at non-interrupt time.

Avoiding Deadlock

The only good way to avoid the problems described in the previous section is to issue any device driver calls you make inside your device driver *asynchronously* .

A traditional Mac OS device driver (**DRVR A**) that calls another driver (**DRVR B**) should operate in the following fashion:

1. Accept a request (**Request A**).

2. Ignore whether the request is synchronous or asynchronous.

3. Asynchronously issue the sub-request (**Request B1**) to DRVR B.

4. Return to the caller.

The sub-request should have an ioCompletion routine. The code for that ioCompletion routine should do the following:

1. If this is the last operation we need to do to fulfil **Request A**, jump to jIODone.

2. Otherwise, asynchronously issue another sub-request (**Request Bn**) to DRVR B with the same ioCompletion routine, and then return to the caller.

Structuring your device driver in this fashion turns your device driver into a state machine. If, for example, you have to issue two sub-requests for each request you receive, your device driver ends up with two states: Request B1, and Request B2. Your completion routine would go:

1. Copy the data from the completed sub-request into the main request's result buffer.

2. If the state is Request B2, complete the initial request by jumping to jIODone.

3. Otherwise, increment the state, issue the next request asynchronously, and return to the caller.

This structure works regardless of whether your driver (DRVR A) is called synchronously or asynchronously, and regardless of whether the driver you are calling (DRVR B) is synchronous or asynchronous.

Figures 1, 2, 3 and 4 illustrate these four cases.

Figure 1 A driver called synchronously, calling a synchronous driver.

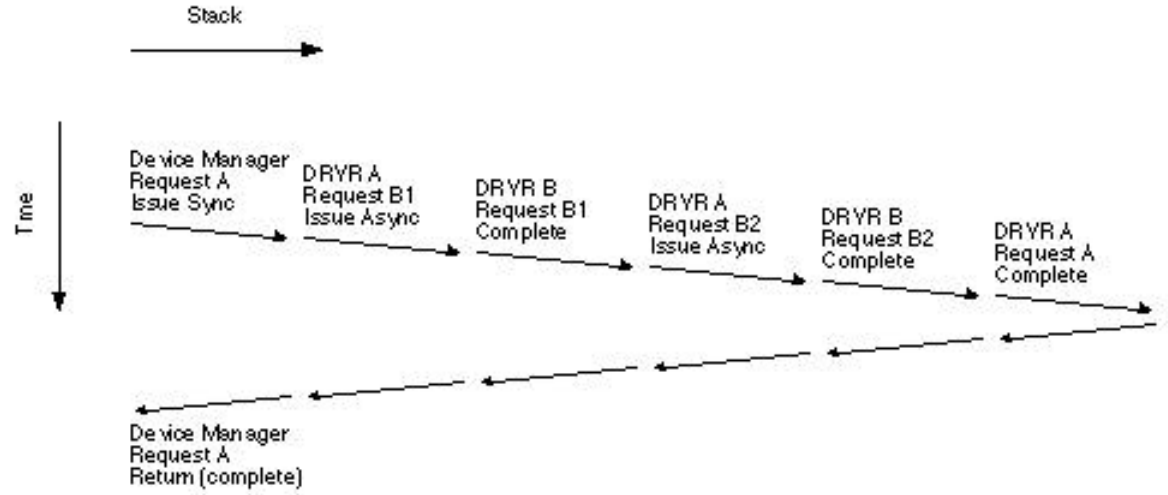


Figure 2 A driver called asynchronously, calling a synchronous driver.

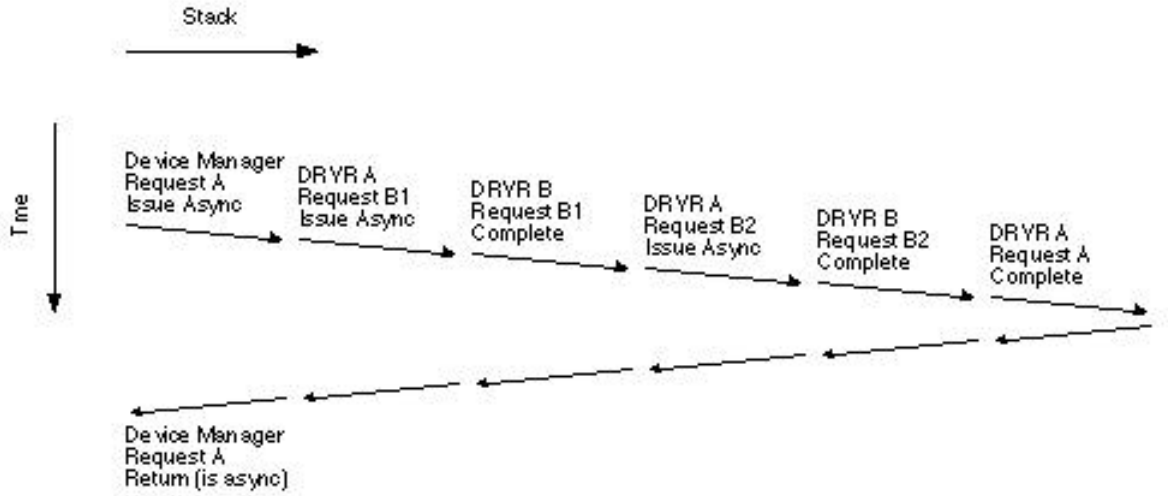


Figure 3 A driver called synchronously, calling an asynchronous driver.

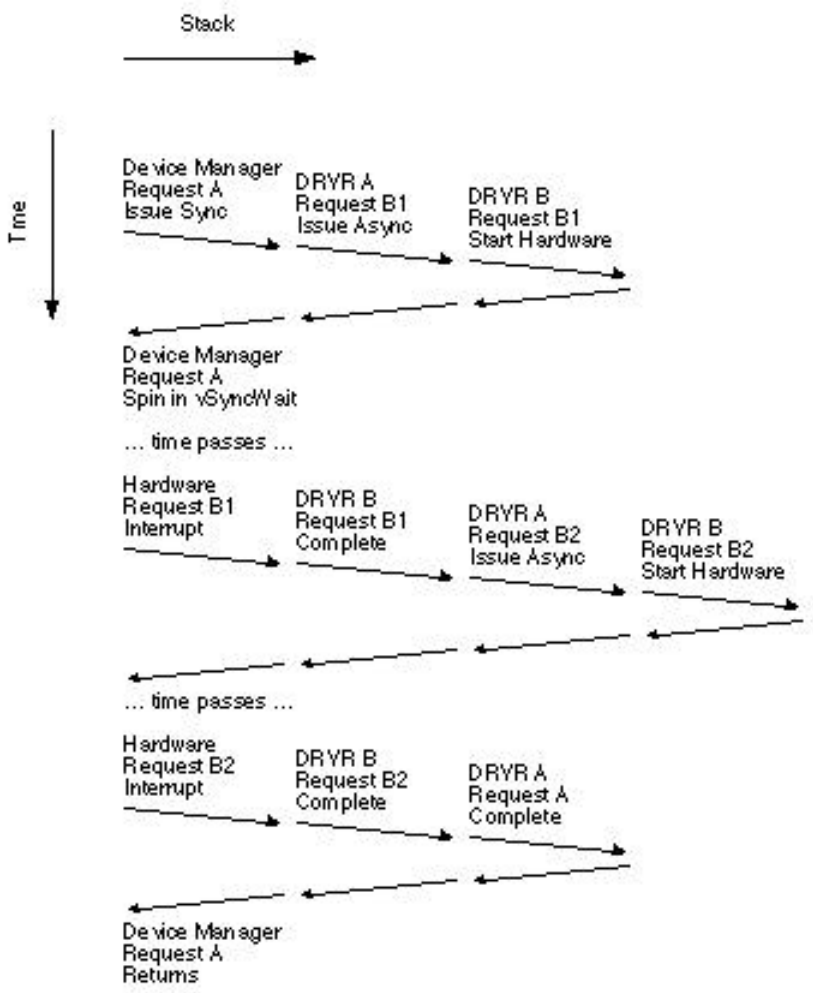
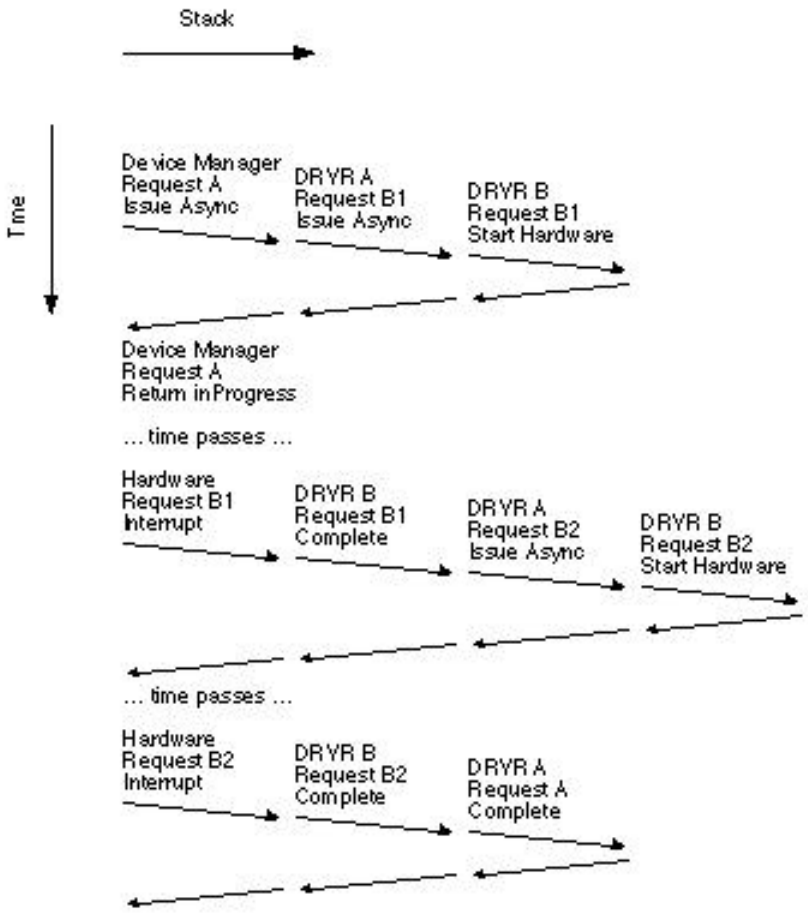


Figure 4 A driver called asynchronously, calling an asynchronous driver



You might think to simplify your life by detecting whether you are being called synchronously or asynchronously, and changing your operation in each case. While it sounds like a neat idea in theory, it is a very bad idea in practice, for two reasons:

1. You have to handle the asynchronous case properly anyway, and if you handle the asynchronous case properly, the synchronous case just works. There's no point in writing twice as much code when half as much would do.

2. You will break **Rule #2** and leave yourself open to deadlocks.

Exceptions That Prove the Rule

The expression "The Exception Proves the Rule" actually means that the exception *tests* the rule. For **Rules #1** and **#2**, there are two significant exceptions that we must explore in order to understand this topic fully.

Drivers That Aren't Called Asynchronously: Exception #1

Many people use traditional Mac OS device drivers for things that don't fit the model. The obvious example here are desk accessories, but older programs often use device drivers as a shared library mechanism. Drivers used in this way are not really part of the I/O system, and generally aren't called asynchronously. This is the reason for the "If you can be called asynchronously" clause in the rules.

Note that using device drivers as a shared library mechanism is now strongly discouraged. The Macintosh has a number of real shared library mechanisms that should meet your needs.

Note:

If you use a DRVR as a pseudo shared library, you should have your client issue immediate calls to your driver (e.g., PBControlImmed). These calls are sent through to your driver directly, and are not queued in any way.

Classic SCSI Manager: Exception #2

One case that seems to contradict **Rule #1** is the relationship between the File Manager and "classic" SCSI device drivers. The "classic" epithet refers to SCSI device drivers that were written prior to SCSI Manager 4.3. These drivers were necessarily synchronous because the SCSI Manager did not support asynchronous operations.

So how is it possible to call the File Manager, which in turn calls the device driver, which calls the SCSI Manager, at interrupt time, even though the SCSI Manager is necessarily synchronous?

The answer is simple: the File Manager contains specific code to support this. When you issue an asynchronous request, the File Manager checks whether the SCSI hardware is already in use. If it is, the File Manager defers the operation of the command until the SCSI hardware is free again. This gets around the potential deadlock, but it contradicts the spirit of **Rule #1**.

Note that the File Manager is only able to defer the request because the request is asynchronous. If you called to File Manager synchronously, you would deadlock.

Summary

Most Macintosh programmers want to play by the rules. Unfortunately, in the case of traditional Mac OS device drivers, the rules are poorly documented and sometimes hard to understand. This Technote explains two important rules for traditional Mac OS device drivers. If you follow them, you will make the system as a whole more reliable.

Further Reference

- *develop 13* , Asynchronous Routines on the Macintosh, by Jim Luther
- *Modern Operating Systems* , by Andrew S. Tanenbaum, Prentice-Hall, 1992, ISBN 0-13-588187-0
