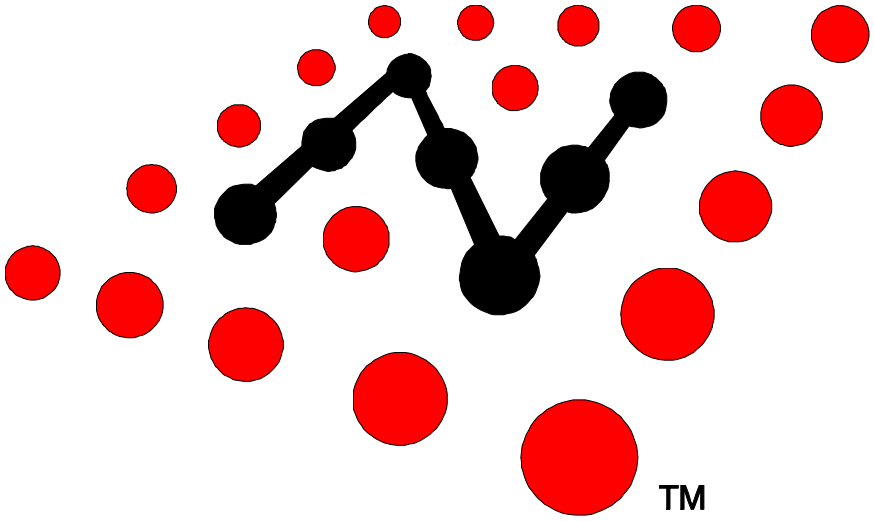

Using NIOS Events Correctly

Version 1.0 Draft 0.1
February 1996

NetWare®

Disclaimer

Novell, Inc. makes no representations or warranties with respect to the contents or use of this document, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Further, Novell, Inc. makes no representations or warranties with respect to any NetWare software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to make changes to any and all parts of NetWare software, at any time, without obligation to notify any person or entity of such changes.

Trademarks

Novell, Inc. has made every effort to supply trademark information about company names, products, and services mentioned in this document. Trademarks were derived from various sources.

NetWare and Novell are registered trademarks of Novell, Inc.

Copyright

This work is an unpublished work and contains confidential, proprietary and trade secret information of Novell, Inc. Access to this work is restricted to (1) Novell employees who have a need to know to perform tasks within the scope of their assignments, and (2) entities other than Novell who have entered into appropriate license agreements. No part of this work may be practiced, performed, copied, distributed, revised, modified, translated, abridged, condensed, expanded, collected, or adapted without the prior written consent of Novell, Inc. Any use or exploitation of this work without authorization could subject the perpetrator to criminal and civil liability.

Novell, Inc.
122 East 1700 South
Provo, Utah 84606 U.S.A.

Draft 0.1
Software version 1.0
February 1996
Novell Part#

Introduction

There are two NIOS routines that are causing deadlock and system performance degradation in the Windows 95 environment when used improperly (these problems are not present in the DOS/WIN NIOS environment, only in Windows 95):

NiosScheduleAESEvent	Schedules an event to fire after a specified amount of time.
NiosScheduleForegroundEvent	Schedules an event that fires in a foreground context.

The problem comes in how Windows 95 handles threads and events. It is important to remember that event time is not the same as process time, and there are a number of things that should not be done during event time. Those things are outlined in this document.

In the next release of NIOS, many of these problems will disappear. This paper describes short-term solutions to keep programmers from inadvertently creating deadlock states during event handling.

From the Microsoft VMM Help (all references from VMM Help are shown in italics):

With the greater degree of multi-tasking available in Windows 95, the opportunity for deadlocking the system grows enormously. Moreover, some operations, while not deadlocking the system, effectively shut off multi-tasking until the operation completes.

Principles for Event Time Execution

There are six general principles to keep in mind when programming in the context of an event.

1. **Save the state of client registers before executing an event**. In the context of an event, if you have to call back into the VM (that is, you're performing nested execution), you must save the state information of the client registers before your event begins, and restore them before returning from your event. This is because events are not synchronized with the virtual machine, and so the contents of client registers are unpredictable. Saving the client registers can be done with either of two NIOS calls:

   To save registers:
   DOSBeginNestExec
   DOSBeginNestExecWithCRS

To restore registers:

DOSEndNestExec
DOSEndNestExecWithCRS

2. **Do not attempt to claim a resource that is owned by the current thread.**

   Since event callbacks can be called while the current thread is blocked on a semaphore or other synchronization object, events should be extremely careful not to create deadlocks by attempting to claim a resource that may already owned by the current thread. For example, consider a thread which takes a resource, then blocks waiting for some other operation to complete, with the intention of releasing the resource after the other operation has completed. While waiting for the semaphore to be signaled, that thread is used to perform an event callback is which attempts to take the same resource. The system is now deadlocked, because the event will wait indefinitely for the resource, which cannot be released until the event returns.

3. **Don't make any call while in the context of an event that will invoke a paging operation if the current thread is already in the middle of its own paging operation.** Because NIOS does not currently ensure that the thread context that an event is running in is not in the middle of a paging operation, it is up to the programmer to ensure that the thread is not in the middle of a paging operation. This can be done as follows:

   A. Check to see if the critical section is owned by the current thread. Before a paging operation is begun, the thread must reserve the critical section. To do this, make the VMM call:

   Get_Crit_Status_Thread

   This routine will tell you whether the critical section is currently owned, and by which thread. If it is not owned, then no paging operations are in progress and the event may perform a paging operation.

   If the critical section is owned but not by this thread, there is still no problem, and the event may perform a paging operation.

   If the critical section is owned by the current thread, then another routine can be called to determine if it is owned because of a paging operation.

B. Determine if the critical section is owned by the current thread because of a paging operation. To do this, the programmer must call in to the PageSwap device. This is done as follows:

```
VxDcall  PageSwap_Test_IO_Valid
jc       Paging_not_permitted
```

If the jc flag is set on return, the event may NOT page, as a paging operation is already in progress, and is owned by this thread.

NOTE: *Allocating memory from the VMM page manager or heap manager may result in paging, so services like* ***_PageAllocate*** *and* ***_HeapFree*** *are also forbidden unless it has already been determined that the thread is not paging. List management functions like* ***List_Allocate*** *are safe to call, provided the list was not created with the LF_HEAP or LF_SWAP bits.*

4. **Do not access the Registry unless you're sure that the thread processing your event callback is not already active in the Registry.** Since NIOS does not allow events to be scheduled with the *PEF_Wait_Not_Nested_Exec* flag, there is no way to restrict the execution of the event, and so no way to ensure that the processing thread is not already active there. So it is best to avoid accessing the Registry at all from events until this can be corrected in NIOS.

   Event callbacks may not access the registry unless it can be ensured that the thread processing the event callback is itself not active in the registry. This can be done by using the PEF_Wait_Not_Nested_Exec restriction on the event.

5. **Don't block at event time.** A common example of how blocking at event time can cause a problem is when the VxD attempts to acquire a resource at event time which the current thread already owns. In effect, the thread is waiting on itself, and deadlock ensues quickly.

   Even if you have ensured that blocking on a semaphore or other synchronization object will not cause a deadlock, bear in mind that blocking at event time seriously impacts the system's ability to multitask smoothly. The thread that got selected to service an event might own resources at ring 3, such as the Win16Mutex, for which other threads are waiting. (Indeed, the fact that the thread is running at all makes it much more likely that it owns such a resource.) While the event is in progress, those resources remain held by the thread even though the thread isn't doing anything with them.

6. **As a rule, don't block while holding a critical section.** Blocking with the critical section can cause deadlock because so many important

system operations require the critical section in order to execute. Imagine the scenario wherein a thread acquires a critical section, schedules an event, and then runs **NIOSPoll** while waiting for the event to complete. While it is currently possible to do this (and it has been widely done), it has the potential to seriously degrade system performance. Further, if programmers rely on this practice, it will prevent NIOS from being corrected in the next release.

General rules to remember are to hold critical sections for only a short time, and to never block while holding a critical section. (A possible solution to the scenario above is to acquire the critical section only after the event has fired.)