Portable Systems Group

Windows NT APC Design Note

Author: *David N. Cutler*

Original Draft 1.0, February 6, 1989
Revision 1.2, March 30, 1989

The following design note describes a proposal for the handling of **APC**'s in **Windows NT**. The companion design notes on alerts and attach process contain information and algorithms that are pertinent to this design.

The nice thing about **APC**'s is that they interrupt thread execution at any point and cause a procedure to be executed in the context of a specified thread. This capability can be used to reduce the number of threads required to perform a particular function and can alleviate the need for polling.

The new model for implementing **OS/2** and **POSIX** compatibility with protected subsystems would suggest that **APC**'s could be used to substantially reduce the overhead and implementation complexity of these subsystems. For instance **OS/2** timers could be implemented by **NT** timers that queue an **APC** when they expire. The **APC** would be fielded by the **OS/2** subsystem which would clear the appropriate semaphore and delete or repeat the timer as appropriate.

As good as this all sounds it is not without flaw. The very thing that makes **APC**'s so useful is also the same thing that makes them so bad. This is the fact that they interrupt a thread at arbitrary points. To get past this liability, the capability to "disable" **APC**'s over short regions of code is needed. But this then has the problem of not being very modular and also requires a lot of thought on the part of the user. Writing code that is "**APC**" safe is VERY difficult.

SRC never recognized the need for **APC**'s but did recognize that it was useful to be able to send a thread an alert signal. This signal typically means quit what you are doing and reset to some canonical state. **SRC's** system provides a function to send an alert to a thread (*AlertThread*), a function to test if a thread had been alerted (*TestAlert*), and a form of wait that allows a thread to be alerted while it is waiting (*WaitAlertable*).

When TestAlert or WaitAlertable is called and the subject thread has been alerted, then the condition "alert" is raised. In addition, if AlertThread is called while a thread is waiting as the result of a call to WaitAlertable, then the thread is unwaited and the "alert" condition is raised.

The nice thing about the **SRC** alert design is that the alert condition occurs at well defined points in the execution of a program. These points are exactly the points where the program says it is alertable. Writing code that is "alert" safe is easy.

We do not want to drop the flexibility of **APC**'s, but at the same time we do not want to interrupt the execution of a thread at arbitrary points. Therefore why not combine the notion of alertable with the functionality of **APC**'s? To do this we simply do not deliver an **APC** unless the thread is alertable or calls TestAlert.

We only need to do this for user mode, and it fact, do not want to do this for kernel mode as we need to break into the kernel mode execution of a thread at an arbitrary point. As system designers this does not (or more succinctly better not!) present us with the same level of difficulty that it does the run of the mill user.

Thus in user mode, **APC**'s are only delivered at points where the program is alertable. In kernel mode **APC**'s are delivered when the appropriate enabling conditions are present.

The following is an explantion of how **APC**'s would work using the concepts described above.

There are three types of **APC**'s:

1. special kernel

2. normal kernel

3. normal user

A special kernel **APC** is deliverable whenever the Interrupt Request Level (**IRQL**) of the corresponding thread is equal to zero, and executes in kernel mode at **IRQL** 1. This type of **APC** is used to break into a thread's execution and perform some short operation such as posting **I/O** status. Code that runs as the result of a special kernel **APC** is not allowed to acquire any mutexes that can also be acquired at **IRQL** 0. Special kernel **APC** code is allowed to take page faults, and thus memory management code must ensure that it runs at **IRQL** 1 when it owns a mutex that could also be acquired during a special kernel **APC**.

A normal kernel **APC** is deliverable whenever the **IRQL** of the corresponding thread is equal to zero, a normal kernel **APC** is not already in progress, and the thread does not own any kernel level mutexes. Normal kernel **APC** code executes at **IRQL** 0 and is allowed to execute any code including all system services.

A normal user **APC** is deliverable at any time the target thread is user mode alertable. Normal user **APC** code executes at **IRQL** 0 and is allowed to execute any code including all system services.

Both normal kernel and normal user **APC's** can also specify a routine that is to be executed in kernel mode at **IRQL** 1 just prior to executing the normal **APC** routine.

Each thread has a machine state which includes **IRQL**, an **APC** pending flag for each of the modes kernel and user, an **APC** in progress flag for kernel mode, and the number of mutexes that are owned in kernel mode. This state is used to determine when an **APC** should be delivered to a thread.

Unlike **VAX** or **PRISM**, there is no hardware support for **APC**'s. Thus at each exit from kernel mode (i.e. on each **REI** type of operation), appropriate tests must be made to determine whether an **APC** should be delivered or not.

The following pseudo code describes the logic of system exit:

```
ExitFromSystem:

     disable interrupts;
     IF Previous IRQL == 0 THEN
          Get current TCB address;
          IF Previous mode == Kernel THEN
               IF Tcb.KernelApcPending THEN
                    IRQL = 1;
                    Call kernel APC delivery code;
               END IF;
          ELSEIF Tcb.UserApcPending THEN
               IRQL = 1;
               Call user APC delivery code;
          END IF;
     END IF;
     Restore state and continue execution;
```

The user **APC** delivery code is only called when an **APC** can actually be delivered to user mode. Calling the kernel **APC** delivery code, however, does not guarantee that a kernel **APC** can really be delivered. Further checks must be performed to ensure that proper enabling conditions are present. These tests include whether the thread currently owns any mutexes and whether a normal kernel **APC** is already in progress.

A thread in **Windows NT** can be in one of six states:

1. initialized - the thread has been initialized but has not been readied for execution.
2. running - the thread is currently in execution on some processor.

3. ready - the thread is either in a processor ready queue (i.e. ready to execute) or in a process ready queue (i.e. process is not in balance set).

4. standby - the thread has been selected to run on a processor but has not actually started its execution.

5. terminated - the thread has terminated but has not yet been rundown (e.g. all resources have not been returned).

6. waiting - the thread is waiting on one or more dispatcher objects to attain a state of signaled.

When an **APC** is queued, certain tests must be performed to determine what action if any should be taken.

The following pseudo code describes the logic of queuing an **APC**:

```
PROCEDURE QueueApc (
     IN Acb : POINTER KtApc;
     IN Tcb : POINTER KtThread;
     );

BEGIN

     IF Acb.Mode == Kernel THEN
          IF Acb.Type == Special THEN
               Insert APC at front of thread kernel APC
                    queue selected by Acb.ApcIndex;
          ELSE
               Insert APC at end of thread kernel APC queue
                    selected by Acb.ApcIndex;
          END IF;
          IF Tcb.State == Running AND
             Acb.ApcIndex == Tcb.ApcIndex THEN
               IF Tcb.NextProcessor == CurrentProcessor THEN
                    Set software interrupt at IRQL 1;
               ELSE
                    Set APC delivery request for target
                         processor;
                    Set interrupt request for target
                         processor;
               END IF;
          ELSEIF (Tcb.State == Waiting AND
                  Acb.ApcIndex == Tcb.ApcIndex AND
                  Tcb.WaitIrql == 0) AND
                  (Acb.Type == Special OR
                  (Tcb.MutexCount == 0 AND
                       NOT Tcb.KernelApcInProgress)) THEN
                  Unwait thread with status of KernelApc;
          END IF;
          Tcb.KernelApcPending = True;
     ELSE
          Insert APC at the end of thread user APC queue
```

```
            selected by Acb.ApcIndex;
         IF Tcb.State == Waiting AND
            Acb.ApcIndex == Tcb.ApcIndex AND
            Tcb.WaitMode == User AND
            Tcb.Alertable THEN
              Tcb.UserApcPending = True;
              Unwait thread with status of Alerted;
         END IF;
      END IF;
END QueueApc;
```

A thread may be unwaited to execute a special kernel, normal kernel, or normal user **APC**.

If the **APC** executes in kernel mode then the **APC** will have already been executed by the time that execution continues in the wait code. For this case the wait function is merely repeated.

If the **APC** executes in user mode, then execution continues in the wait code without having deliverd the user **APC**. For this case, the wait code simply returns the status "alerted" to the executive level Wait routine. The executive level Wait routine must return a status of "RepeatService" to the system service dispatch. The system service dispatcher backs up the **PC** so that the wait service will be repeated, restores state as necessary, and then executes the "**REI**" which will cause a user **APC** to occur.

Revision History

Original Draft 1.0, February 6, 1989

Revision 1.1, February 10, 1989

1. Move alert algorithms to alert design note.

2. Add test for attached process in QueueApc procedure.

3. Add software interrupt request when **APC** is queued to the current processor in kernel mode.

4. Correct algorithm for delivery of user **APC**.

Revision 1.2, March 30, 1989

1. Minor edits to confrom to standard format.

2. Add capability to receive **APC**'s while attached to another address space.

[end of apc.doc]