Portable Systems Group

Windows NT Alerts Design Note

Author: *David N. Cutler*

Original Draft 1.0, February 9, 1989
Revision 1.2, March 30, 1989

This design note discusses a proposal to implement alerts in both kernel and user mode. The alert capability can be used to interrupt thread execution in either processor mode at well defined points. A companion design note on **APC**'s contains information and algorithms that are pertinent to this design.

There are three alert specific kernel services; *TestAlertThread*, *AlertThread*, and *AlertResumeThread*. In addition, the kernel *Wait* functions take a mode and an alertable flag as arguments.

Each thread has an alerted flag for each of the processor modes user and kernel. These flags are set by calling the AlertThread function and specifying the thread and the mode which are to be alerted.

If AlertThread is called and the target thread is in a wait state, then several additional tests are performed to determine the correct action to take.

If the mode of the wait is user, the alertable flag is set, and the alert mode is user, then a thread specific **APC** is queued to user mode which will raise the condition "alerted", the user **APC** pending flag is set, and the thread is unwaited with a completion status of "alerted".

If the mode of the wait is kernel or user, the alertable flag is set, and the alert mode is kernel, then the thread is unwaited with a status of "alerted". There is no **APC** queued for kernel mode.

The following pseudo code describes the logic of AlertThread:

```
PROCEDURE AlertThread (
     IN Mode : KtProcessorMode;
     IN Tcb : POINTER KtThread;
     );

BEGIN

     Acquire dispatcher database lock;
     IF Tcb.State == Waiting THEN
          IF Tcb.WaitMode >= Mode AND Tcb.Alertable THEN
               IF Mode == User THEN
```

```
                    Queue Tcb.AlertAcb;
                    Tcb.UserApcPending = True;
               END IF;
               Unwait thread with a status of Alerted;
          ELSE
               Tcb.Alerted[Mode] = True;
          END IF;
     ELSE
          Tcb.Alerted[Mode] = True;
     END IF;
     Release dispatcher database lock;
END AlertThread;
```

When the user mode alerted flag gets set, it remains set until either a TestAlert or a Wait alertable is performed which clears the flag.

The kernel mode alerted flag is treated somewhat differently in that it is cleared on each system service entry to the system. The reasoning behind this is that a kernel mode alert should only persist for the duration of time that execution continues in kernel mode. As soon as execution leaves kernel mode, the alerted flag is no longer significant. This is a very important feature which allows the conditional aborting of native system services by protected subsystems which provide system services for other operating system **API's**. This subject is discussed in more detail at the end of this document.

The kernel service *AlertResumeThread* allows a thread to be alerted and then resumed in a single operation. This operation is really a kernel mode AlertThread followed by a ResumeThread, but is provided as a kernel service so that is can be executed without any race conditions.

The following pseudo code describes the logic of AlertResumeThread:

```
PROCEDURE AlertResumeThread (
     IN Tcb : POINTER KtThread;
     ) RETURNS integer;

VARIABLE

     OldCount : integer;
```

```
BEGIN

     Acquire dispatcher database lock;
     IF Tcb.State == Waiting THEN
          IF Tcb.Alertable THEN
               Unwait thread with a status of Alerted;
          ELSE
               Tcb.Alerted[Kernel] = True;
          END IF;
     ELSE
          Tcb.Alerted[kernel] = True;
     END IF;
     OldCount = Tcb.SuspendCount;
     IF Tcb.SuspendCount <> 0 THEN
          Tcb.SuspendCount = Tcb.SuspendCount - 1;
          IF Tcb.SuspendCount == 0 THEN
               Release Tcb.SuspendSemaphore;
          END IF;
     END IF;
     Release dispatcher database lock;
     RETURN OldCount;
END AlertResumeThread;
```

TestAlertThread tests the alerted flag for a specified processor mode and returns a status value of "alerted" if the flag was set and "normal" if the flag was clear. If the alerted flag was set, then it is cleared, and if the specified mode is user, then an alert **APC** is queued to user mode and user **APC** pending is set in the calling thread's **TCB.**

In addition, TestAlertThread also tests whether a user **APC** should be delivered. If the specified mode is user and the user **APC** queue contains an entry, then **APC** pending is set in the calling thread's **TCB.**

The following pseudo code describes the logic of TestAlert:

```
PROCEDURE TestAlertThread (
     IN Mode : KtProcessorMode;
     ) RETURNS KtStatus;

BEGIN
```

```
     Acquire dispatcher database lock;
     Get current TCB address;
     IF Tcb.Alerted[Mode] THEN
          Tcb.Alerted[Mode] = False;
          IF Mode == User THEN
               Queue Tcb.AlertAcb;
               Tcb.UserApcPending = True;
          END IF;
          Release dispatcher database lock;
          RETURN Alerted;
     ELSE
          IF Mode == User AND Tcb.ApcQueue[User] <> NIL
THEN
               Tcb.UserApcPending = True;
          END IF;
          Release dispatcher database lock;
          RETURN Normal;
     END IF;
END TestAlertThread;
```

Wait tests the alerted flags for the specified and all more privileged processor modes if the alertable argument value is true. If an alerted flag is set, then a status value of "alerted" is returned.

In addition, Wait also tests whether a user **APC** should be delivered if the alertable argument value is true and the specified mode is user. For this case, if the user **APC** queue contains an entry, then **APC** pending is set in the calling thread's **TCB** and a status value of "UserApc" is returned.

The following pseudo code describes the logic of Wait:

```
PROCEDURE Wait (
     IN Mode : KtProcessorMode;
     IN Alertable : boolean;
     IN WaitObject : POINTER KtDispatcherObject;
     IN Timeout : POINTER integer;
     ) RETURNS KTStatus;

BEGIN

Repeat:
     Acquire dispatcher database lock;
     Get current TCB address;
     IF Alertable THEN
          IF Tcb.Alerted[Mode] THEN
               Tcb.Alerted[Mode] = False;
               IF Mode == User THEN
                    Queue Tcb.AlertAcb;
                    Tcb.UserApcPending = True;
               END IF;
               Release dispatcher database lock;
               RETURN Alerted;
          ELSEIF Mode == User THEN
               IF Tcb.UserApcQueue <> NIL THEN
                    Tcb.UserApcPending = True;
                    Release dispatcher database lock;
                    RETURN UserApc;
               ELSEIF Tcb.Alerted[Kernel] THEN
                    Tcb.Alerted[Kernel] = False;
                    Release dispatcher database lock;
                    RETURN Alerted;
               END IF;
          END IF;
     END IF;
     IF WaitObject.Signal THEN
          Satisfy wait for WaitObject;
          Release dispatcher database lock;
          RETURN Tcb.WaitStatus;
     ELSE
          Tcb.Alertable = Alertable;
          Construct wait control block for WaitObject;
          Initialize Tcb.Timer with time out value;
          Insert wait control block in wait queue;
          Insert Tcb.Timer in timer queue;
          Select new thread to run;
          Swap context to new thread;
          IF Tcb.WaitStatus == KernelApc THEN
               Goto Repeat;
          ELSE
               RETURN Tcb.WaitStatus;
          END IF;
     END IF;
END Wait;
```

It is the responsibility of the executive to test for the "alerted" return status from TestAlert and Wait and perform the correct operation (e.g. cleaning up data structure, unwinding, etc).

Wait and AlertThread both allow a thread that is waiting user mode alertable to be awakened by a kernel mode alert. If this were not done, then it would not be possible to abort the Wait system service.

The interesting combinations of initial conditions and the resultant action when a Wait system service is executed are given below.

Case 1

```
     Wait Mode = Kernel
     Tcb.Alerted[User] = True
     Tcb.Alerted[Kernel] = False
     Alertable = True

     Action - Put thread in wait state
```

Case 2

```
     Wait Mode = Kernel
     Tcb.Alerted[User] = x
     Tcb.Alerted[Kernel] = True
     Alertable = True

     Action - Clear Tcb.Alerted[Kernel] and return Alerted
```

Case

```
     Wait Mode = User
     Tcb.Alerted[User] = True
     Tcb.Alerted[Kernel] = x
     Alertable = True
     Action - Clear Tcb.Alerted[User], queue Tcb.AlertAcb,
          and set Tcb.UserApcPending

Case 4

     Wait Mode = User
     Tcb.Alerted[User] = False
     Tcb.Alerted[Kernel] = True
     Alertable = True

     Action - Clear Tcb.Alerted[Kernel] and return Alerted

Case 5

     Wait Mode = User
     Tcb.Alerted[User] = False
     Tcb.Alerted[Kernel] = False
     Alertable = True

     Action - Put thread in wait state
```

Kernel mode alerts can be used to implement the semantics necessary to abort native system services. The following discussion describes how this can be implemented in **Windows NT**.

In **Mach** the operations necessary to abort a native system service are suspend, abort service, and resume. This capability is used to get a thread out of a possible wait state in the system and deliver a signal, terminate execution, etc.

A similar set of primitives can be provided in **Windows NT** using the kernel alert capability.

Windows NT suspends a thread by sending it a normal kernel **APC** that causes the thread to wait on an semaphore that is built into the thread object. The resume operation simply releases the builtin semaphore which continues thread execution.

The suspend wait operation is nonalertable to ensure that the alert and resume operation functions properly; see below.

If a thread is in a wait state when it is suspended, then the wait completion status is set to "kernel **APC**". This is done so the wait can be repeated when the **APC** returns.

Implementing the primitives to abort native system services does not quite solve the whole problem. Each native service that can result in a long wait must be written such that it is responsive to kernel alerts. This means that a native service should wait alertable in kernel mode when it does a wait that could take a long time. Also, if very long algorithms are being performed, then TestAlert should also be called at appropriate points.

It is preferable that a native service either complete successfully or be entirely aborted. For those cases where there are really two parts to the service such as an operation followed by a wait, the service should be broken into two parts. Each part should be executed separately from the calling mode.

A protected subsystem that is a system service server can stop, alter, and a resume a thread by performing the sequence of operations suspend, get state, set state, and alert and resume.

If a native service is active when the suspend operation takes place, then the kernel alerted flag will remain set for the duration of the service after the thread is resumed. The alerted flag can be tested by the service using the TestAlert function.

A more interesting case is when the native service is waiting kernel mode alertable. The suspend service causes a normal kernel **APC** to be sent to the target thread which completes its wait with a status of "kernel **APC**". The target thread then waits nonalertable on its builtin suspend semaphore.

When the subsystem executes the alert and resume service, the kernel alerted flag is set in the target thread and the target thread's suspend semaphore is released. This causes the target thread to be unwaited with a status that is the key value of the semaphore.

Unwaiting the thread causes it to continue execution in the suspend **APC** routine which simply returns to the kernel **APC** delivery code. The kernel **APC** delivery code restores the state of the thread and resumes execution at the point of interruption which is in the wait code. The wait code tests the wait completion status and determines that the wait was satisfied to deliver a kernel **APC**. The wait is repeated and finds that the kernel alerted flag is set and that the wait is alertable. Thus it returns immediately with a wait completion status of "alerted".

Note that the kernel **APC** delivery code must save and restore the wait completion status in the **TCB** so that the subsequent suspend wait does not destroy it.

Revision History:

Original Draft 1.0, February 9, 1989

Revision 1.1, February 10, 1989

1. Include tests for nonempty user **APC** queue in TestAlert and Wait algorithm descriptions.

Revision 1.2, March 30, 1989

1. Minor edits to conform to standard format.

[end of alerts]