Portable Systems Group

Windows NT Attach Process Design Note

Author: *David N. Cutler*

Original Draft 1.0, February 8, 1989
Revision 1.2, March 30, 1989

This design note discusses a proposal that would allow a thread to attach to the address space of another process, execute code in the attached process's address space, and then detach and resume execution in the original process address space. It is envisioned that this capability will be required to implement the newly proposed system structure.

This capability would not be exported to user mode at all. It is intended for internal use by the executive layer of the system.

The new system structure (i.e. system service servers) requires the ability to perform certain operations on behalf of another process. Typical of these operations is creating and deleting virtual memory. In order to implement these operations, we either have to build the data structures and algorithms such that they can be done outside the recipient process or architect a way to actually execute code within the address space of another process.

A good example of a difficult service to build outside of a process is the deletion of virtual memory. **Mach** stands on its head to implement this capability and, while it is doing such an operation, a global virtual memory lock must be held.

Graham Hamilton (of exDECwest fame) suggested that a way to do this was to have some number of anonymous system threads which could do such an operation. A requesting thread would build a request packet that contained the arguments of the operation to be performed, the function that was to be executed, a pointer to the address map that the thread was to execute in, and an event to synchronize the completion of the operation. The request packet would then be queued to the worker thread, a semaphore signaled, and the requesting thread would wait on the event. A worker thread would be awakened by the signal of the semaphore and would remove an entry from the request queue. The thread would attach to the new address space, perform the operation, set the event, detach from the address space, and then look for more work to do. The requesting thread would then resume execution.

In analyzing Graham's proposal it is clear that there are two extra context switches, a copy of the argument data,

two extra translation buffer and data cache flushes, and the need to attach to an address space. So why not just let the requesting thread directly attach to the target process address space and avoid the worker threads, the argument copy, and the two extra context switches?

When a thread wanted to execute in another process' address space it would execute the following logic:

verify that source process has the rights necessary to perform the desired operation on the destination process
obtain pointers to objects in the source process as necessary
KeAttachProcess(*pPcb*)
perform desired operation in address space of target process
KeDetachProcess()
resume execution is source process

There are several questions and complications that arise from doing this kind of operation. These include:

1. How is the kernel stack of the source thread addressed in the target process?

2. What happens if the source process gets removed from the balance set while an attach operation is in progress and causes the process' thread's kernel stacks to be made pageable?

3. What happens if the target process is not in the balance set?

4. What happens if the source or target processes are terminated?

5. What happens if the source thread is terminated?

6. What happens if a thread tries to do a second attach after having attached to a target process' address space?

7. What object table is visible when a thread is attached to the address space of another process?

8. What working set is manipulated while a thread is attached to the address space of another process?

9. What process gets charged for the time that is consumed while the thread is attached to another process' address space?

10. How is mutex ownership handled between the source and target processes?

11. What happens if user and/or kernel mode are alerted while a process is attached?

12. What happens to **APC**'s that are queued to the thread after it has entered the target process' address space?

13. Can the attached thread receive **APC**'s?

14. What happens if a suspend or resume is performed on the specified thread?

Before attempting to answer these questions it is useful to review the kernel data structures that correspond to process and thread objects. These data structures are described in more detail at the end of this note.

There is a Process Control Block (**PCB**) and a Thread Control Block (**TCB**).

A **PCB** contains a pointer to a process address map (actually the physical address of the Page Directory for the process), a list of all the **TCB**'s that are members of the **PCB**, a count of all the kernel mutexes owned by member **TCB**'s, and a state which is either "included" or "excluded" (corresponds to whether the process is, or is not, in the balance set).

A **TCB** contains a pointer to the **PCB** of which it is a member, an **APC** queue for each of the modes kernel and user, a kernel **APC** in progress flag, a kernel **APC** pending flag, a user **APC** pending flag, a user alert **APC** Control Block (**ACB**), an alerted flag for each of the modes kernel and

user, an alertable wait flag, an owned mutex count, and link pointers for linking the thread into the **PCB**'s **TCB** list.

Actually there are several other fields in the **TCB** and **PCB**, but they are not really pertinent to this discussion.

The kernel data structures that describe the **TCB** and **PCB** are contained within the executive data structures that describe the process and thread objects. The executive must use the linkage structures provided by the kernel and cannot keep a separate set of linkage pointers that tie the data structures together.

The below discussion addresses the questioins raised above and gives an explanation of how **KeAttachProcess** and **KeDetachProcess** work.

How is the kernel stack of the source thread addressed in the target process?

We would like to make kernel stacks addressible in the process part of the address space. However, in order to attach to another process' address space we will need to map kernel stacks in the system part of the address space so we can avoid an argument copy and allocation of a temporary kernel stack. If we do not do this, then we will have to allocate a temporary kernel stack in the system part of the address space, copy necessary argument information to the temporary stack, switch to the temporary stack, attach the target process' address space, execute the necessary logic, switch back to the source address space, switch back to the original stack, and then deallocate the temporary stack.

When a process is in the balance set the kernel stacks of all its threads must be locked in memory (there are several ways we can do this - the reference count on the pages being the most likely candidate). When a process is not in the balance set, the kernel stacks of all its threads are pageable. The locking and unlocking of these pages is performed by the balance set manager when it brings a process into or out of the balance set.

What happens if the source process gets removed from the balance set while an attach operation is in progress and

causes the process' thread's kernel stacks to be made pageable?

If the source process is allowed to leave the balance set while a thread is attached to another process, then the kernel stack on which the thread is running would become pageable. This cannot be allowed to happen since it would cause the system to crash if a page fault occurred on the kernel stack itself. In order to prevent this situation from happening, the **Pcb.MutexCount** in the source **PCB** is incremented by one on attach to ensure that the process is not allowed to leave the balance set. When the corresponding detach is executed the count if decremented by one.

Even though the process is not allowed to leave the balance set any threads that do not own mutexes are prevented from further execution if the process is excluded from the balance set. Threads that do own mutexes are allowed to continue execution until they release all the mutexes they own. Therefore **Tcb.MutexCount** in the **TCB** is incremented by one on attach to ensure that the thread continues to execute. When the corresponding detach is executed the count is decremented by one.

What happens if the target process is not in the balance set?

If the target process is not in the balance set, then the subject **TCB** is inserted in the target **PCB**'s ready queue. When the corresponding process is brought into the balance set, the thread's **TCB** will be inserted in the appropriate dispatcher ready queue. We must ensure that once the target process is brought into the balance set, it is not allowed to leave the balance set until the detach operation is performed. This is required since we have incremented **Tcb.MutexCount** which allows the thread to continue running in the target process' address space even though the process might be removed from the balance set. Therefore **Pcb.MutexCount** is also incremented in the target process' **PCB** during the attach operation. When the detach operation occurs all the mutex counts will be corrected to enable the respective processes to leave the balance set.

What happens if the source or target processes are terminated?

What happens if the source thread is terminated?

The kernel does not allocate or deallocate any data structures that control the execution of threads within the system. It depends on the executive to keep appropriate reference counts, and only when the reference count is zero, can the executive delete data structures. Therefore the executive must ensure that the reference count of the source process, the target process, and the subject thread are such that they cannot be deleted during the execution of a attach/detach sequence.

What happens if a thread tries to do a second attach after having attached to a target process' address space?

The **TCB** of a thread contains the storage necessary to save information for a single execution of an attach/detach sequence. Therefore the rule is that only one level of attach is allowed. If an attempt is made to attach to another address space while an address space is already attached, then a bug check will occur.

What object table is visible when a thread is attached to the address space of another process?

The object table of the attached process is visible to a thread when it is attached to another process' address space. It is doubtful that it will ever be necessary to create an object in another process' object table, but this operation can be performed if necessary.

What working set is manipulated while a thread is attached to the address space of another process?

While a thread is attached to another process' address space it takes page faults and manipulates the working set of that process as if it were really a thread in that process.

What process gets charged for the time that is consumed while the thread is attached to another process' address space?

While a thread is attached to a target process' address space, the target process is charged for the execution time accumulated by the thread. When the detach operation occurs, execution time is again charged to the source process.

How is mutex ownership handled between the source and target processes?

There is simple rule for mutex ownership. When a thread does and attach or detach process it cannot own any mutexes. If an attempt is made to attach/detach while a thread owns a mutex, then a bug check will occur.

What happens if user and/or kernel mode are alerted while a process is attached?

There is no interaction between alert and attach process. Kernel alert applies to whatever context the thread is currently in. The thread can either respond or ignor kernel alert as appropriate. User alert only applies to the source context since user mode cannot be entered when a process is attached.

A user mode alert cannot occur while a thread has a process attached since the thread will never do a wait alertable for user mode. An alert **ACB** may have been queued just prior to attaching the process in which case it will occur when the thread detachs and returns to user mode.

What happens to APC's that are queued to the thread after it has entered the target process' address space?

Can the attached thread receive APC's?

An **ACB** is initialized and directed to a thread running in a specific address space. Therefore **APC**'s directed to a source process context cannot be allowed to occur while the subject thread is attached to the address space of another process. This means that there must be a way to direct an **APC** to the right context and make sure it does not occur at the wrong time.

To accomplish this, each **TCB** will contain an **APC** state index (**Tcb.ApcStateIndex**) which can have a value of zero or one (only one level of attach is allowed). When an **ACB** is initialized the address of the associated **TCB** must be specified. This allows **Tcb.ApcStateIndex** and **Tcb.CurrentApcState.Pcb** to be captured and stored in the **ACB** in addition to the address of the **TCB** itself.

Two sets of **APC** context are stored in the **TCB**; the current **APC** context (**Tcb.CurrentApcState**) and the saved **APC** context (**Tcb.SavedApcState**). Each set of context contains the **APC** state information described for the kernel **TCB** data structure.

An array of pointers is used to address the two sets of **APC** context. When an **ACB** is queued, the appropriate set of **APC** context is selected by using **Acb.ApcStateIndex** to obtain the appropriate array member which contains the address of the corresponding set of **APC** context. A comparison is then made between the **PCB** address stored in the **ACB** and the **PCB** address stored in the selected **APC** context. If a mismatch occurs, then a bug check is executed (i.e. an attach was performed, an **ACB** was initialized (e.g. associated with a timer), a detach was performed, and then the **ACB** was queued). Otherwise the **ACB** is inserted in the selected **APC** queue and appropriate **APC** state bits are updated. If **Tcb.ApcStateIndex** is equal to **Acb.ApcStateIndex**, then the **APC** effects the current context of the subject thread and checks are made to determine if an **APC** should be delivered immediately.

When **Tcb.ApcStateIndex** is zero, the first pointer of the array points to **Tcb.CurrentApcState** and the second pointer points to **Tcb.SavedApcState**. To ensure a **PCB** address mismatch occurs if an attempt is made to queue an **ACB** with an **Acb.ApcStateIndex** value of one, a value of **NIL** is stored in **Tcb.SavedApcState.Pcb**.

When **Tcb.ApcStateIndex** is one, the first pointer of the array points to **Tcb.SavedApcState** and the second pointer of the array points to **Tcb.CurrentApcState**. Both sets of context have a valid **PCB** pointer.

When an attach process is executed, **Tcb.ApcStateIndex** is examined. If the value is one, then a bug check occurs (i.e. an attempt is being made to attach another process while one is already attached). Otherwise **Tcb.ApcStateIndex**

is incremented and the current **APC** context is copied to the saved **APC** context. The two pointers in the array that address the **APC** context blocks are switched and the current **APC** state is initialized.

While a thread is executing in another process' address space, the thread can initialize and receive **APC**'s targeted to that address space.

When a detach process is executed, **Tcb.ApcStateIndex** is examined. If the value is zero, then a bug check occurs (i.e. an attempt is being made to detach an address space when one is not attached). The current **APC** context is also examined to determine if the thread has a "clean" **APC** context. If a kernel **APC** is in progress, the kernel **APC** queue contains an entry, or the user **APC** queue contains as entry, then a bug check occurs. Otherwise **Tcb.ApcStateIndex** is decremented, the saved **APC** context is moved to the current **APC** context, the saved **APC** context **PCB** address is set to **NIL**, and the two entries in the pointer array are switched.

What happens if a suspend or resume is performed on the specified thread?

A thread is suspended by queuing the thread's builtin suspend **ACB**. This **ACB** is initialized such that it's target is the source process' address space and causes a normal kernel **APC**. In an attempt is made to suspend a thread while it attached to another process, then the suspend **ACB** will get queued to the source context and the suspend count will get adjusted. Suspension of the thread will not actually occur until the thread does a detach and reenters the source context. The thread may be suspended and resumed several times while it is attached to another process. This works in the same way as the case where the suspend **APC** cannot be delivered because the thread is either currently in a kernel **APC** or has kernel **APC**'s blocked (**IRQL** raised).

The following pseudo code describes the operation of attach to address space:

```
PROCEDURE KeAttachProcess (
     IN Pcb : POINTER KtPcb;
     );

BEGIN

     Acquire dispatcher database lock;
     Get current TCB address;
     IF Tcb.ApcStateIndex == 1 OR Tcb.MutexCount <> 0 THEN
          Call bugcheck with fatal error;
     ELSE
          Tcb.ApcStateIndex += 1;
          Tcb.SavedApcState = Tcb.CurrentApcState;
          Tcb.CurrentApcState.Pcb = Pcb;
          Tcb.CurrentApcState.KernelApcInProgress = FALSE;
          Tcb.CurrentApcState.KernelApcPending = FALSE;
          Tcb.CurrentApcState.UserApcPending = FALSE;
          Initialize APC queue headers for current state;
               Swap APC context pointers in APC pointer array;
               Tcb.MutexCount += 1;
               Pcb.MutexCount += 1;
               Tcb.SavedApcState.Pcb->Pcb.MutexCount += 1;
               IF Pcb.Active OR Pcb.MutexCount > 1 THEN
                    Flush data cache;
                    Set new page directory pointer;
                    Release dispatcher database lock;
               ELSE
                    Tcb.PcbReadyQueue = TRUE;
                    Insert TCB in PCB's ready queue;
                    Select new thread to run;
                    Call context switch routine;
               END IF;
          END IF;
          RETURN;
     END KeAttachProcess;
```

The following pseudo code describes the operation of detach from address space:

```
PROCEDURE KeDetachProcess (
     );

BEGIN

     Acquire dispatcher database lock;
     Get current TCB address;
     IF Tcb.ApcStateIndex == 0 OR Tcb.MutexCount <> 1 OR
        Tcb.CurrentApcState.KernelApcInProgress OR
        Current kernel APC queue not empty OR
        Current user APC queue not empty THEN
          Call bugcheck with fatal error;
     ELSE
          Tcb.ApcStateIndex -= 1;
               Tcb.CurrentApcState.Pcb->Pcb.MutexCount -= 1;
               IF Tcb.CurrentApcState.Pcb->Pcb.MutexCount == 0
                  AND NOT Tcb.CurrentApcState.Pcb->Pcb.Active
                    THEN
                    Set Tcb.CurrentApcState.Pcb->Pcb.Event;
               END IF;
          Tcb.CurrentApcState = Tcb.SavedApcState;
          Tcb.SavedApcState.Pcb = NIL;
               Swap APC context pointers in APC pointer array;
               Tcb.MutexCount -= 1;
               IF Kernel APC queue not empty THEN
                    Tcb.CurrentApcState.KernelApcPending = TRUE;
                    Set software interrupt at IRQL 1;
               END IF;
               Tcb.CurrentApcState.Pcb->Pcb.MutexCount -= 1;
               IF Tcb.CurrentApcState.Pcb->Pcb.MutexCount == 0
                  AND NOT Tcb.CurrentApcState.Pcb->Pcb.Active
                    THEN
                    Set Tcb.CurrentApcState.Pcb->Pcb.Event;
                    Tcb.PcbReadyQueue = TRUE;
                    Insert TCB in PCB's ready queue;
                    Select new thread to run;
                    Call context switch routine;
               ELSE
                    Flush data cache;
                    Set new page directory pointer;
                    Release dispatcher database lock;
               END IF;
          END IF;
          RETURN;
```

```
END KeDetachProcess;
```

Revision History:

Original Draft 1.0, February 8, 1989

Revision 1.1, February 17, 1989

1. Add text to explain what interactions exist between attach/detach process and suspend/resume, **APC**'s, alerts, and mutexes.

2. Allow APC's to be queued and processed in either the source or target address on attach/detach operations.

Revision 1.2, March 30, 1989

1. Minor edits ot conform to standard format.

[end of attproc]