Singularity Design Note

29

Hardware Process Isolation in Singularity

In which we describe how Singularity can optionally make use of hardware protection mechanisms to isolate processes

1. Motivation

Support for hardware protection mechanisms were added to Singularity in order to allow traditional hardware-based process-isolation techniques to be directly compared to the software techniques that had previously been used exclusively in Singularity.

The "Deconstructing Process Isolation" Technical Report (MSR-TR-2006-43) was the first product of this work, and presents a set of experiments that compare the costs and benefits of hardware-based isolation to software-based isolation.

This document aims to document the design changes that were necessary to accommodate hardware protection. It also documents the current state of the implementation as of **June 20th, 2006**.

```
Notes on low-level implementation details, or on limitations of the
current implementation, are formatted like this.
```

whereas

Definitions or general notes are formatted like this.

From the previous version of this document, which acted as a project proposal, we have preserved this elaboration of one of the primary motivations for exploring alternatives to software protection in the first place:

1.1. Threats to the integrity of software isolation

To date, Singularity has relied exclusively on software techniques to prevent processes from interfering with each other. Singularity processes are referred to as "SIP"s (Software Isolated Processes) to underscore this design.

It is only possible for SIPs to operate without the use of hardware protection because SIPs are believed to have a certain set of properties that ensure their correct operation without hardware oversight:

- SIPs do not contain privileged instructions that manipulate processor state inappropriately
- SIPs do not reference other process' heap data
- SIPs manipulate data structures shared with other processes (such as the communication heap and channel endpoint structures), but preserve their integrity while doing so
- SIPs access certain kernel structures (such as processor and thread state blocks) but preserve their integrity while doing so

These properties are established and verified by a considerable body of code (not all actually in existence today), including an MSIL verifier, an MSIL-to-machine-code compiler (Bartok), and the trusted runtime component of every SIP. In general, software isolation rests on the correct operation of the entire Trusted Computing Base, which encompasses all these vital components. Reducing the size of the Trusted Computing Base while preserving the software-only isolation approach is an ongoing subject of research.

Obviously, flaws may exist in the TCB:

- Singularity applications may mistakenly be converted into machine code that contains, or can arrange to execute, privileged instructions
- Singularity processes may improperly reference memory, either because verification failed to reject impermissible application code or because the conversion to machine code was incorrect
- The trusted runtime code, which accesses shared system data structures, may be buggy and inadvertently corrupt those structures

In addition to potential flaws in the TCB, if unreliable *hardware* should corrupt pointer values or instructions in memory, and if hardware protections are not being used, nothing would prevent a corrupted process from affecting data structures of another process, or compromising the integrity of the system by executing privileged instructions that affect the processor state. Using hardware protection mechanisms may provide a partial defense against this type of corruption, if the errors occur during the execution of non-kernel code.

The use of hardware protections, then, can be motivated by a distrust of the mechanisms involved in ensuring the integrity of software isolation.

2. Design Overview

This section presents the design of hardware protection features in Singularity in broad terms.

2.1. Protection Domains

A thorough comparison of hardware protection techniques to software protection techniques demanded the ability to measure the performance of the system with various permutations of protections, with the ability to control:

- Whether hardware is used to protect the kernel's data structures from specific processes
- Whether hardware is used to protect specific processes from each other

In order to enable the widest set of possible configurations, **protection domains** were introduced.

A Protection Domain is a collection of processes. Hardware is used to isolate each protection domain from all others.

The kernel, of course, is a special case. In the usual case, hardware mechanisms are also used to isolate the processes within a protection domain from the kernel. However, a protection domain can also be configured so that hardware is *not* used to isolate its processes from the kernel. Such a protection domain is referred to as a **kernel domain**, and there can be more than one kernel domain.

A Kernel Protection Domain, or "kernel domain", is a protection domain such that hardware protections are *not* used to isolate the processes within the domain from the kernel.

Protection domains can be used to group together processes that should be able to communicate as efficiently as possible, at the cost of forgoing hardware protection between them. When it is desirable to use hardware protection techniques to isolate two processes from each other, they should be placed in separate protection domains. In the extreme, every process can be situated in its own protection domain.

The full power of SIP isolation continues to apply to processes situated in the same protection domain. This is quite different from traditional operating systems, in which code executing within the same hardware-protection boundary (i.e., process) used weak isolation mechanisms, if any.

Similarly, processes that can be "trusted" sufficiently to not warrant hardware protection from the kernel can be placed in kernel domains, decreasing the cost of their communication with the kernel. Processes that are not "trusted" to this extent can be placed in "normal" protection domains, with a corresponding increase in the cost of their communications with the kernel.

The configuration that most resembles a traditional non-microkernel operating system such as Windows is one wherein all applications run in their own protection domains (one domain per process), but where driver processes are loaded into a kernel domain. A configuration that could be considered closest to a traditional micro-kernel architecture would be one in which every process, including driver processes, executed in its own protection domain.

2.2. Implementation Requirements

The implementation aims to satisfy the following constraints:

- To the greatest possible extent, processes co-located within a protection domain should be able to communicate as efficiently as they would on a Singularity system that does not use hardware protection at all. That is, hardware-protection costs should be paid only at hardware-protection boundaries.
- To the greatest possible extent, enabling hardware protection should not imply any global costs, for example a performance penalty for processes not isolated from each other by a hardware boundary.
- Hardware protection must be as absolute as possible, that is, it should be possible to run a completely untrusted process within a protection domain without any risk of compromising the rest of the system. This precludes relying on a "trusted runtime" within hardware-isolated processes.

Note, with respect to the last constraint, that "completely untrusted" processes are a new construct in Singularity and indeed, we still have no real-world examples of such processes. The experiments conducted for "Deconstructing Process Isolation" did not involve "untrusted" processes per se, although in an attempt to quantify the overhead associated with ensuring type and memory safety at runtime, one set of experiments did involve eliding the usual run-time checks associated with those guarantees.

In principle, though, the introduction of hardware isolation should permit experimentation with processes that do not have the usual type and memory-safety properties that we have traditionally relied on in Singularity. Whether this is actually desirable or not is left to the judgment of the reader.

3. Design

This section presents the specific changes that have either already been made to Singularity, or are intended, to support hardware protection.

In order to support the "Deconstructing Hardware Protection" experiments, a core set of changes have already been carried out to implement hardware isolation. However, these changes carry significant limitations, which are called out.

There are two main mechanisms available on full-featured modern processors that we leverage to provide hardware isolation:

- **Memory virtualization** (via the organization of memory into pages, and a Page Translation Table), with associated per-page access protections, and
- **Execution privilege levels** to restrict the use of potentially dangerous machine instructions

Swapping, or the practice of selectively writing memory pages to a backing store, such as a hard disk, in order to up RAM for other use, is not directly related to the issue of hardware *protection*, and is, in any event, not part of this project.

3.1. Memory Virtualization

As in traditional systems, the processor's ability to employ Page Translation Tables, which map the logical addresses requested by running code to physical memory locations, is used to create separate memory "address spaces", that serve to isolate processes from each other by making it impossible for one process to address the memory used by another. Switching between address spaces involves flushing and reloading the processor's cache of Page Translation-Table entries (the Page Translation Lookaside Buffer), which potentially costs many memory cycles, as the new page table entries must be loaded from main memory.

In Singularity, each protection domain has an associated address space.

3.1.1. Organization of virtual memory

The memory that is accessible to processes when the processor's page-translation feature has been activated is referred to as "virtual memory", because its mapping to physical memory is flexible and under the control of the operating system. Note that our usage of this term has nothing to do with "swapping".

Singularity organizes the 32-bit addressable memory space into a *kernel range*, in low memory, and an *application range*, in high memory. The location of the boundary between the kernel range and the application range is controlled by code constants.

```
Microsoft.Singularity.BootInfo.KERNEL_BOUNDARY determines the location
of the kernel / application memory boundary; currently it must be set
to a multiple of 1GB to facilitate the organization of the page-table
structure.
```

As in traditional systems, the explicit kernel range ensures that the kernel's data structures are always accessible. The "application range" can accommodate the address space of one Protection Domain at a time. Switching from one Protection Domain to another involves reloading the page-table register (CR3 on an x86 processor), which results in the processor flushing its internal cache of page-table entries.


```
Singularity takes advantage of the "Global" flag on page-table
entries. This flag indicates, on x86 architectures, that flagged
entries should not be flushed across CR3 reloads. Because the kernel
range is mapped in the same way in every address space, this flag is
used to prevents mappings for pages in the kernel range from being
invalidated on address-space switches.

N.B: The organization of virtual memory on a 64-bit architecture is
likely to be significantly different. In particular, virtual address
space is no longer a scarce resource, so it becomes possible to keep
all address spaces mapped at all times, and employ different
techniques to ensure their isolation from each other.
```

3.1.2. Memory tiers

In the original Singularity implementation, there was an identity mapping between virtual addresses and physical addresses. As an implementation detail, memory paging was activated during the OS boot sequence, but only for the convenience of being able to mark the first page of memory as unmapped in order to trap references to the null address at runtime.

A single OS facility, encapsulated in Microsoft.Singularity.Memory.Pages, administered physical memory and satisfied memory-allocation requests from processes. This page-manager implementation maintained free lists of memory ranges, organized by size. Because of the identity mapping between virtual addresses and physical addresses, it was possible for inconvenient memory-usage patterns to fragment physical memory, and indeed this phenomenon was repeatedly observed to degrade performance and sometimes cause memory allocation requests for more than one page to fail.

```
On non-PAGING-enabled Singularity builds, memory is never yielded back
to the operating system by process runtimes, in order to avoid
fragmentation of physical memory. Obviously, this is an interim
"solution".
```

With the introduction of a proper virtual-memory management system, the issue of physical-memory fragmentation can be prevented, at the cost of additional complexity. Memory in Singularity is now organized in the following tiered fashion:

- **Physical Pages** are administered by Microsoft.Singularity.Memory.PhysicalPages. Physical memory pages are tracked with a physical-pages table, which is constructed during the boot sequence before processor paging is enabled. The physical-pages table resides in an identity-mapped area low in the kernel memory range, so it is always directly accessible after processor paging has been enabled. The page table is implemented such that free pages can be identified rapidly. Page allocations are only supported one-at-a-time, as memory fragmentation is meant to be prevented in higher tiers.
- **I/O Memory** is a fixed range of *physically contiguous* memory that is allocated during the boot sequence, before processor paging is enabled, in order to support drivers that require more than one physically contiguous page of memory for DMA transfers. Note that since this range of memory is *always* backed by physical RAM, it is a precious commodity. This range of memory is also identity-mapped after processor paging is enabled, and is administered separately by the Memory.PhysicalHeap object. The PhysicalHeap manager is able to satisfy requests for more than one contiguous page of memory, although it is possible for the I/O memory range to become fragmented, much as the original Singularity memory pool did. Memory is administered by maintaining a page table at the beginning of the I/O memory range that includes a threaded free-block list, organized by size, similarly to the original Singularity memory manager.

- **Virtual Memory Ranges** are administered by the Memory.VirtualMemoryRange class. A virtual memory range spans a subset of the addressable virtual memory space, and within that range, an instance of the Memory.VirtualMemoryRange object is able to satisfy requests to allocate *virtually contiguous* ranges of memory. Unlike physical memory and I/O memory, a virtual memory range is not initially backed by physical memory pages. To satisfy an allocation request, the VirtualMemoryRange implementation first identifies an available range of *virtual addresses*, and then requests physical pages (which need not be contiguous) to back that address range. VirtualMemoryRange makes use of the Memory.VMManager machinery to map and unmap memory pages.

```
The allocation scheme employed by VirtualMemoryRange is currently very
simplistic: a high-water-mark for memory allocation is maintained,
which is lowered when the topmost allocated memory is freed. New
allocation requests are satisfied from above the high-water mark.
Fragmentation may cause allocations to eventually fail. In the long
run, improved interaction between the virtual memory manager and
process GCs will need to occur to avoid fragmentation.
```

- **Process Heap Memory** is administered by each process' runtime library. Every protection domain's memory area resides on the substrate of a VirtualMemoryRange manager object, so processes within a protection domain may interfere with each other by causing fragmentation of their shared memory space. This is permitted by the design.

```
There is currently an unhealthy coupling between the Singularity
process runtime and the operating system's memory management: the
process runtime assumes that there is a page table accessible with a
full word available to describe each addressable page of memory. The
runtime further assumes that it can determine the availability and
nature of memory pages by examining this table directly, and that GC-
specific state information can be stored into certain bits of this
table's per-page data words.

In order to satisfy this assumption, VirtualMemoryRange currently
allocates such a page table for each memory range. However, the space
consumption of this table is considerable, and encapsulation of the
memory tiers is compromised by its existence.

As part of the hardware-protection work, the runtime memory code was
altered to not assume, among other things, that its addressable range
began at the zero address and extended for all of memory! Further
improvements are necessary to eliminate the need for this large,
linear memory table.
```

3.1.3. Multiplicity of Exchange Heaps

The current Singularity design includes a single, system-controlled "exchange heap" through which all IPC is conducted. Because no part of a hardware-isolated Singularity process may be trusted, this approach must be revisited when hardware protection is to be enabled.

Rather than use a single exchange heap for the entire system, a Singularity system now:

- Always has an exchange heap for the use of the kernel
- Has an additional exchange heap per non-kernel protection domain

Processes running in kernel domains use the kernel's exchange heap.

The multiplicity of exchange heaps is necessary to ensure that even without trusting any portion of any process running in a non-kernel domain, we can be assured that a process running in one protection domain cannot corrupt the exchange heap used by a process running in a different protection domain.

Microsoft.Singularity.Memory.SharedHeap has been generalized so it can be multiply instantiated. Previously, it was a singleton class.

3.1.3.1. Integrity of exchange heap records

In the software-only Singularity design, the integrity of the system-global exchange heap is safeguarded by the trusted runtime component of every process. With hardware protection, we no longer wish to rely on the runtime component of applications to ensure the integrity of hardware-protection boundaries.

Because it is impossible for the kernel to marshal data into a heap structure whose integrity cannot be guaranteed, it becomes essential to defend the integrity of the exchange heaps with hardware techniques.

This is accomplished by ensuring that exchange heap allocation records are situated on physically distinct pages from the actual data of each exchange heap block. The allocation-record pages can be written only by the kernel, ensuring their integrity. Allocations and deallocations must be accomplished via ABI calls, which was already the case in the previous Singularity design.

N.B: Allocation-record pages must remain readable to client processes because pointers to exchange heap blocks are actually pointers to the block's allocation record, and the client process performs a double-indirection to access the block's data.

The exchange heap implementation (Memory.SharedHeap) was already arranged such that allocation records are on physically distinct pages, with one exception: block refcounts are situated inline with the block data, on data pages. The refcount feature, however, is used only by the implementation of channel endpoints, but never directly by client applications. With the advent of hardware protections, the refcounting facility in the exchange heap is deprecated, to be removed once conversion to the new, hardware-protection-friendly channel implementation is complete.

3.2. Execution privilege

Virtual memory is used to isolate the processes running in different protection domains from each other; since it becomes impossible for a process in one protection domain to form an address that refers to the memory accessible to the processes of another protection domain, the domains cannot interfere with each others' data.

Kernel memory, however, is always mapped, so it is always possible to form an address that refers to kernel memory. Processor privilege modes are used to ensure that processes running in non-kernel domains are unable to access memory belonging to the kernel. All memory pages within the kernel range are marked (in the page-mapping table) as being accessible only when the processor is running in "supervisor" mode (as opposed to "user" mode). On an x86-architecture processor, supervisor mode corresponds to "ring 0", and user mode corresponds to "ring 3".

```
Currently, kernel memory is not marked inaccessible to user-mode
processes, since user / supervisor separation of shared structures
such as the ThreadContext structure has not been completed.
```

3.2.1. ABI (process-to-kernel) transitions

In the original Singularity design, Application Binary Interface crossings from processes to the kernel are implemented simply as function calls (modulo housekeeping such as indicating the transition boundary via stack markers). When processes are run within non-kernel protection domains, this is no longer possible, as code running at lowered processor privilege cannot directly call code that must run at high processor privilege (and is located on memory accessible only to high-privilege code!)

A multi-step process enables processes to invoke the kernel via ABI routines. In broad terms, a processor privilege-transition mechanism must be used. On the x86 architecture, Singularity employs the syscall /sysreturn pair of instructions for transitioning from ring 3 to ring 0 and back again.

The syscall mechanism allows the operating system (or other privileged code) to arrange for unprivileged code to be able to safely invoke privileged code. This is accomplished by loading certain processor registers with the address of a master dispatcher routine to be invoked for any unprivileged-to-privileged code transition, along with certain pieces of state information to be loaded into the processor at transition time. Unprivileged code is permitted to execute the sysenter instruction, and the effect is to jump to the master dispatcher routine and elevate processor privilege. A calling convention must be defined between unprivileged and privileged code; this convention typically supports requesting more than one distinct operation by use of a selector code.

```
Such a calling convention is defined for Singularity. On calling
sysenter, x86 register ECX contains the ABI-routine selector code, and
EDX contains a pointer to the first function argument, situated on the
unprivileged code's stack. EDX+4 contains the second argument. EDX+8
is ignored, and further arguments are found at EDX+12. On ABI
completion, execution is returned to EDX-4. [Review!]
```

3.2.1.1. Build Time

A "normal" Singularity process is linked against a stand-in for the kernel assembly, such that the final executable indicates that each ABI entry point that the process uses must be fixed up at load time by the OS loader. The Singularity loader matches each imported symbol to the appropriate kernel address at process-load time, and by the time the process code executes, it has all necessary addresses for jumping directly to the kernel when invoking an ABI.

Currently, a process that is to be run at lowered privilege on Singularity must be prepared differently at build time.

This requirement should be lifted.

The process, rather than being linked such that it has imported symbols to be resolved by the loader, instead becomes fully self-contained, by ensuring that each internal call to an ABI is instead resolved to an assembly stub that sets up correctly and then employs the sysenter instruction to jump into the kernel.

Currently, a build tool named syscallbuilder is employed to automatically generate process-side assembly stubs for each ABI entry point used by a process.

Implementation limitations:

syscallbuilder is driven off dumpbin output, because we require the use of fully-decorated names in order to be able to calculate the size of function arguments. This should instead be driven off an improved version of the human-maintained ABI-symbols list used to drive linking against the kernel assembly.

Also, syscallbuilder uses hard-coded knowledge of the size of various primitive types used in ABI functions in order to generate assembly stubs. This information should be extracted from Bartok somehow instead, to eliminate fragility.

On the kernel side, the syscallbuilder tool also generates an assembly stub that converts from the ABI-invocation convention to the argument-passing convention expected by each ABI implementation function (currently, fastcall). These stubs are invoked by the master dispatch routine to properly invoke the implementation for the chosen ABI operation.

3.2.1.2. Runtime ABI invocation

ABI invocation in a hardware-protection world is a delicate matter, and not only because a special mechanism must be used to transition across processor-privilege levels. Because we do not wish to trust any portion of the runtime system for hardware-protected processes, the kernel must take pains to avoid assuming that any process-supplied data, including implicit data such as its stack pointer, is valid. It is also essential that the kernel not allow any of *its* state, in particular the kernel's stack, from being accessible to unprivileged process code, lest the process accidentally or maliciously corrupt it.

Implementation limitation:

Every single ABI entry point should carefully validate its parameters. None do in a way that is adequate to defend against completely untrusted processes.

The sequence of events on ABI invocation is roughly as follows:

- Application runtime code pushes all ABI arguments onto the process stack and prepares the processor state as per the ABI invocation conventions.
- The application executes sysenter.
- Execution is transferred to the kernel's master ABI dispatching routine, at elevated privilege. Interrupts are disabled as a side effect.
- The kernel immediately switches the stack pointer to a per-processor stack segment for use in early-ABI entry. This ensures that no housekeeping computation is performed using the untrustworthy process stack pointer.
- Shortly thereafter, the kernel switches to a fresh, kernel-memory-range stack chunk. When this is complete, interrupts can be re-enabled and the processor-global ABI-entry stack chunk is available for reuse.

- The master-dispatch routine invokes the appropriate automatically-generated stub to convert process arguments to the calling convention expected by the ABI-implementation function, and calls the appropriate implementation function.
- After completion of the ABI operation, sysreturn is used to lower processor privilege before jumping back to the process' indicated resumption point.

```
Implementation limitations:

Stack marking is almost certainly not correct at the moment; need to
look into this

The early-stage dispatcher stack chunk should be per-processor, but is
currently system-global

We should allocate and cache an initial kernel stack chunk per thread
to optimize away as many of the stack-chunk allocations on ABI
invocations as possible.

Dispatch handling appears incorrect in places but does not break
because we have not yet enabled memory protection!

The entire ABI implementation path should be resistant to bad data
being passed in by the process. It may be impossible to accomplish
this without properly processing page faults (converting them to
exceptions that can be caught)
```


3.3. Cross-protection domain channel communication

In the original Singularity design, each channel endpoint contains a pointer to its peer endpoint, and data transmission is accomplished by the sending process by writing directly into its peer-endpoint structure.

In the hardware-protected world, channel communication can occur across protection-domain boundaries, in which case channel endpoints are situated in different address spaces. This prevents the implementation strategy of writing directly into the receiving endpoint's data structure. As well, of course, that implementation raises trust issues: processes should not be able to arbitrarily corrupt each others' endpoint structures when they are situated in different protection domains.

In order to preserve the integrity of the channel-endpoint structures, they are cleaved in two: the *core endpoint* structure resides in the kernel range of memory, and is not directly accessible to the owning process. The core structure contains the waithandle used to signal the arrival of data, which could be arbitrarily corrupted were it accessible to the process, as well as the peer-core pointer, which could be similarly corrupted. The core structure also contains a buffer area for marshalling data, as described below. The application-space endpoint structure contains the usual buffer area for receiving data transmitted from the peer.

3.3.1. Data marshalling

The approach selected for marshalling data across protection domain boundaries is currently **double-buffering**. That is, transmitted data is first copied into kernel memory, then, at a later time, when the target address space is loaded, it is copied into the application-range of memory for the receiving process. Obviously, this approach has the disadvantage of inefficiency. We were unable to produce a satisfactory direct-copy design in the constrained time ahead of the submission deadline that drove the original implementation work.

In the original Singularity design, marshalling of structured data (data that contains pointers to other data blocks) was accomplished by generated code in the transmitting process. The generated code embodied the compiler's knowledge of the layout of the data to be transmitted, albeit not in a form that was directly usable by other code at runtime. In the hardware-protected world, we have shifted to a different design. Via an extension to the SystemType mechanism, processes must register a structural description of the types they intend to exchange via channels. The description includes the location and types of any pointers within the structure. The description is sufficient to allow the kernel to fully marshal data structures once the types involved have been registered.

> Implementation limitation:
>
> Currently, the extended SystemType mechanism has not been implemented; although there is a type-registration method, it does not include a description of pointer offsets and types. As a consequence, the current data-marshalling implementation is limited to "flat" data.

3.3.1.1. Channel data transmission

Transmission of channel data is relatively straightforward. In all cases, there is a peer-endpoint structure to the transmitting endpoint allocated in the transmitting process' memory space. In the case that the endpoint peer is truly co-located with the transmitting process, this structure corresponds directly to the receiving process' endpoint structure. In the case that the receiving process is situated in another protection domain, this structure is a dummy used for purposes of marshalling data.

In either case, the transmitting process first writes the data to be transmitted directly into the apparent peer-endpoint structure. On completion, it performs an ABI call, NotifyPeer(), to signal its peer that data is waiting. This ABI call has been extended to now potentially signal to the caller that its peer is, in fact, situated in a remote protection domain. In the event that it receives this indication, the transmitter is required to make further calls to cause its data to be marshaled to the recipient.

Today, the additional operations involve multiple calls to **MarshallPointer** and **MarshallMessage**, but this is a consequence of the incomplete implementation. Eventually, a single call to **MarshallData**() will suffice, since the kernel will be aware of the structure of the data being transmitted, and a single call to **MarshallMessage**() will complete the operation.

In previous implementations, a single ABI call was required to signal the peer endpoint's waithandle. Today, in the case that the sending and receiving endpoints are situated in the same protection domain, a single ABI call to NotifyPeer() is required. The runtime overhead, obviously, is comparable.

There is also a new ABI, **GetPeer**(), to retrieve the peer pointer. This operation did not previously involve an ABI call. However, we believe that the peer pointer can be cached in the process' endpoint structure, eliminating this cost.

3.3.1.2. Channel data reception

Receiving channel data is somewhat more involved. There are two overall cases: it is possible that, at the moment that data reception is attempted, data is already waiting in the recipient's endpoint structure. This receive mode is currently (and remains) extremely fast. The second case occurs if data is not already waiting. In this case, the recipient must sleep against his endpoint's data-ready signal event, taking care to address possible race conditions involving the sender.

In the case that data is already waiting in the recipient's endpoint, runtime performance remains as before, as no ABI crossings are involved.

In the case that data is arriving from across a protection-domain boundary, it will initially appear to the recipient that no data has been received. As part of the recipient's preparations to suspend itself pending data reception, it will call the PeerClosed() ABI to determine whether its peer has closed the channel. The kernel seizes on this opportunity to marshal data into the recipient process' exchange heap, and signal the reception of data in the recipient's endpoint structure, such that it appears to the recipient that data arrived concurrently with its call to the PeerClosed() ABI.

The PeerClosed() operation must remain an ABI call, whereas determining whether one's peer had closed the channel did not previously involve an ABI call. **This may represent an additional cost to be borne by channel communications even when not spanning protection domain boundaries**.

Currently, GetWaitHandle() is also a new ABI call, although, similarly to the peer pointer, we believe that an endpoint's data-ready waithandle can be cached in the owning process.

3.3.1.3. Further limitations

Currently, the implementation of channel operations on PAGING-enabled builds is completely separate from the implementation on non-PAGING builds. There remains work to unify the implementation such that substantially the same codepaths are used on both builds.

4. Open Issues

- The ThreadContext structure needs to be cleaved into a kernel-protected portion and a process-accessible portion. We have a design for this but it has not been carried out yet
- We are liable to uncover unexpected complications when we eventually enable memory protection
- It is still an open question whether the final unified design for channel communication can be made every bit as fast as the current implementation when not crossing a protection boundary

The overarching open question:

- **Some of the remaining implementation work may take considerable time to complete properly. What criteria should be applied to determine what level of additional time investment is appropriate?**

