Design Note: Kernel Virtual Map Management[†]

Jonathan Shapiro, Ph.D.   Eric Northup, M.S.
Systems Research Laboratory
Dept. of Computer Science
Johns Hopkins University

December 17, 2005

Abstract

The EROS kernel made many assumptions in its handling of the kernel virtual map. Several of these were problematic, either because they were incompatible with multiprocessor implementations or because they presented problems for systems with large physical memories (e.g. Pentium with 64G physical memory).

This note describes how the kernel virtual map is handled in Coyotos, and the particular differences related to large memories and multiprocessing in the single processor and symmetric multiprocessor kernel. A more radical approach is called for on large-scale NUMA machines.

1 Overview

Any microkernel must deal with several broad classes of mappings:

1. *Mappings for objects that are permanently mapped in the kernel virtual address space, but are not subject to DMA.*

   This category includes kernel code, data, BSS, and the kernel heap.

   Objects in this category have the property that there is no direct correspondence between the underlying physical page addresses and their associated kernel virtual addresses.

2. *Mappings of user data and I/O buffers, which is sometimes subject to DMA.*

   These pages are manipulated by the kernel, but this manipulation is transient in nature. For example, the kernel may zero such a page during allocation.

   These pages have no permanent kernel virtual mapping. The content of these frames may be "pinned" for DMA independent of whether they are currently mapped.

3. *Mapping frames for user mappings.*

   Depending on the hardware mapping design, the available kernel virtual space and the strategy adopted for mapping frame allocation, mapping frames may or may not have permanent kernel virtual mappings.

4. *Per-page overhead structures.*

   These are structures that would have permanent kernel mappings in any sane implementation, but for which permanent mappings may be infeasible in certain large physical memory configurations.

5. *Transient IPC mapping windows.*

   During a fast-path IPC involving a cross-space string copy, a temporary mapping must be established for the target space. This is generally handled as a very special case.

This note attempts to capture how all of this is handled, in an attempt (probably vain) to implement all of this just once. At present, this note considers only the uniprocessor design.

2 The Kernel Virtual Map

The Coyotos kernel virtual map is organized into several regions.

Code, Data, and Heap This region starts at the beginning of the code region and continues upwards. The kernel heap appears at the end of this region and grows upwards (toward higher virtual addresses). This mapping is shared across all CPUs.

Because they are required very early in the bootstrap process, and need to be present before the arrangement of physical memory can be discovered, the master kernel page directory and the CPU0 kernel stack are found within this part of the virtual address space, but are placed there as a special case.

Top-Level Page Tables Because the Coyotos IPC design requires the ability to efficiently copy top-level page table entries, the (non-vestigial) top-level page tables are permanently mapped. This mapping region is shared across all CPUs. Its size is dynamically determined at system startup.

For this purpose, we consider the IA-32 PDPT in PAE mode to be vesitigial, and keep the page directories permanently mapped. We need to be able to quickly copy portions of the map during IPC, and the source mapping tables for these need to be mapped at all times. The entries in the PDPT have too large a span to be able to effectively reserve a kernel virtual window for them.

IPC Mapping Window This region is never smaller than the size of a (possibly misaligned) maximal IPC string. Since the Coyotos string transfer limit is 64 kilobytes, this means that the minimum IPC window is ((64K/pagesize)$+$1). The size of this window is statically known on a per-architecture basis.

On hierarchically mapped machines, there is always some non-vestigial upper level page table.[1] On such systems, the size of the transient IPC mapping window is whatever size is spanned by *two* entries in this structure. The basic idea is that we are going to construct the temporary mapping by copying two top-level page table entries from the top-level receiver page table into the reserved mapping window of the sender's top-level page table.

A key requirement of the IPC mapping window is that any TLB entries derived from it do not survive past the current kernel unit of operation. If necessary, the IPC code must arrange for these entries to be explicitly flushed.

Transient Mapping Window The transient mapping window is a region of virtual page addresses that can be used by the kernel to construct temporary page mappings. The size of this region is some multiple of the leafmost page table size on the hardware. The size of this window is statically known at compile time on a per-architecture basis.

3 Multiprocessing Considerations

Two complications arise in simple multiprocessors: handling of IPC windows and handling of transient mappings.

3.1 IPC Windows

Hardware multiprocessing exposes the difference between simulated and real concurrency. On a uniprocessor, handling of IPC strings is straightforward: the kernel simply constructs a temporary mapping and uses it to perform the string copy. Because there is no concurrent multithreading, the kernel can safely make two assumptions:

1. No other thread in the same sending address space is concurrently using the IPC mapping window.
2. No other thread in the kernel can possibly invalidate any mappings while the current IPC is in progress.

In the uniprocessor case, both invariants can be ensured by invalidating the TLB on the way out of the kernel. On most hardware this is going to happen anyway when the address space is switched, and special provisions are only required for same-space IPC.

Unless special measures are taken, neither of these assumptions holds on a true multiprocessor.

3.1.1 IPC Window Concurrency

In Coyotos, we resolve the two issues in essentially the same way: we make the uppermost non-vestigial page tables (i.e. the ones used for IPC mappings) per-CPU. This decision has several properties:

- It slightly increases pressure on the depend table, but not to any greater degree than we needed to handle in any case.
- It slightly increases the number of page faults associated with migrating a thread to a new CPU for the first time, because a new directory must be constructed. Note that each of these page faults is very high in the mapping tree hierarchy and rebuilds a mapping to an existing page table. Each is therefore amortized well. Further, the per-CPU page directory is reclaimed by an LRU policy, and will probably remain valid as long as the thread continues to get scheduled to the target processor.
- It guarantees that if two threads that share a common address space are running on different CPUs, they do *not* share an IPC mapping window and can perform independent IPC operations without contention or inconsistency.
- It does *not* increase the total number of top level mapping tables required, because we potentially needed one for every running thread in any case when no address space sharing was happening.

[1] For Pentium PAE, the top-most four entry page table is vestigial.

- It gives us a place to stand to build other per-CPU mappings that will prove to be helpful.

3.1.2 Respecting Atomicity During Invalidation

The other issue is avoiding invalidation races. The general form of this problem arises when processor A seeks to invalidate a mapping that is simultaneously in use by processor B. In general, this will occur only when a PATT slot has been cached in a mapping table, referenced on more than one processor, and is subsequently invalidated. There are really two independent problems here:

1. Lower-level mapping tables may be shared, and we would like to avoid unnecessary interprocessor rendezvous in this situation.
2. The transient mappings created by the IPC logic technically violate the dependency tracking invariants, because they are not reflected in the depend table.

The first problem can be partially solved by associating a bitmap with each page table describing which CPUs have loaded entries from it. The second problem is resolved by a combination of locking and coordination.

During the IPC fast path, the sending CPU will copy two top-level entries from the receiving process's uppermost page table. Each of these entries names a lower level page table. In order to ensure notification of invalidation, the sending CPU must first add itself to the ”I have loaded” list on the *containing* (i.e. the uppermost) page table.

When an unrelated CPU goes to invalidate entries in this table, it will notice that the IPC CPU has registered interest. Since this notice of interest exists, it will demand a rendezvous with the IPC CPU to obtain exclusive access over the containing page table. This entails an interprocessor interrupt (IPI).

During the fast path IPC action, the processor makes note that an IPI has been received, but does not respond to the request for coordination until the string transfer has completed. This ensures that there will be no race between string transfer and page table entry invalidation.

3.2 Transient Mappings

Transient mappings occur with high frequency, but rarely involve contention on the objects mapped. We would like to ensure that they also avoid contention on the mapping table entries used to hold them.

The decision to use per-CPU page directories makes this very straightforward, because it allows us to allocate a per-CPU page table that is used for transient mappings on a per-CPU basis. This allows each CPU to perform transient mapping invalidations locally, and in many cases allows them to be completely avoided because the TLB is flushed by IPC operations in any case.

4 Kernel Page Faults

One unfortunate consequence of going to a kernel heap design is that new page tables may need to be added to the kernel mapping table as the heap is extended. These entries may be added *after* top-level user page tables have been fabricated. As a result, the kernel may page fault while trying to access these addresses. This issue is why we have referred to the statically preloaded mapping directory as the *master* mapping directory rather than the CPU0 mapping directory even though it is *used* for CPU0. There are two ways to handle this:

- Invalidate all directories whenever the heap is extended.
- Recognize and deal with this case specially in the machine-specific page fault handler.

Coyotos adopts the latter approach. The rule is that the master mapping directory is always definitive, and the upper defined heap virtual address is always stored in a globally accessable variable. If a page fault occurs within the nominally defined region, the cause is that the active page directory is missing a directory entry update within the kernel region.

5 IA32 Implementation

The IA32 is a 32-bit processor having 4 kilobyte pages. The lowest-level mapping tables map 4 megabyte chunks of virtual space in legacy translation mode, or 2 megabyte chunks in PAE mode. For convenience of implementatation, we allow 4 megabyte regions for the windows even when the second level page table spans only a 2Mbyte region. The kernel virtual map on this processor is organized as follows:

Address	**Description**
0xC100000–??	Start of kernel code
next page boundary	Master mapping directory (r/w)
next page boundary	Transient mapping region (r/w)
next stack boundary	CPU0 stack (dynamic)
next page boundary	Start of heap (dynamic)
...	
(sized at startup)	UPCB mapping region (dynamic)
(sized at startup)	Top-level page tables (dynamic)
0xFF400000 ... top	Transient mapping window (use)
0xFF800000 ... top	IPC Mapping Window

The number of process table entries in this implementation is dynamically selectable. The design allocates enough top-level page tables to have a full upper-level structure for each process. In legacy mode this is NPROC, while in PAE mode it is 4*NPROC.

6 Large Physical Memories

Large physical memories present a significant problem. Our goal is to keep the Coyotos virtual map as small as we possibly can. The problem is that we need a per-frame descriptor structure, and this structure doesn't have to be very big to take up a very large part of the kernel virtual address space.

Consider, as an example, a design in which the per-frame structure is 64 bytes (2^6) and 64 gigabytes of physical memory are populated (2^{24} pages). This design would need tyo use 2^{30} bytes (1 Gigabyte) simply for bookkeeping structures, which is clearly not a viable design. Ideally, we would like to keep the entire kernel virtual region on the IA32 below 0.5 Gigabytes, so we need to do better than this. The overhead structure can probably be shrunk to 32 bytes, but it isn't going to get much smaller than that. Worse, each of these pages needs to be named by at least one in-memory capability in order to be useful.

Our conclusion is that this situation isn't really recoverable. We either need to significantly expand the portion of the virtual address space allocated to the kernel, or we need to arrange to run the kernel in an entirely separate virtual address space from the application. After some back and forth, we concluded that the separate space approach is the right one to use. Alternatively, we could use the area as an exceedingly large cache, but it isn't all that clear what to *do* with that much cache. The current practical limit on configured memory without switching to a separate kernel space appears to be around 16 gigabytes of physical memory. We have no intention of actually testing this configuration any time soon.