First-Class Flexpage-Based Address Spaces

SRL Technical Report SRL-2000-03

Jonathan S. Shapiro, *ed.*
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
shap@eros-os.org

November 27, 2000

Abstract

The L4 mapping managment model – flexpages – is both flexible and attractive. It has two flaws. First, the current design requires the kernel to allocate record-keeping data structures as part of a mapping transfer. Second, address spaces are not first class objects known to the kernel, and therefore cannot be used as long-term storage objects in a persistent design.

This note describes an alternative flexpage-based address space design that we believe resolves both issues.

1 Introduction

L4 provides a mapping transfer mechanism based on flexpages [Lie95]. A *flexpage* is a unit of 2^k pages aligned at a 2^k page boundary. As part of an IPC operation, L4 enables flexpages to be transferred from one process to another. This design has considerable appeal. It is easy to understand. It provides an efficient mechanism for bulk transfer of mappings to be shared among cooperating processes. It is architecture neutral.

Unfortunately, the L4 IPC design requires that the kernel allocate data structures during the mapping transfer operation so that the operation can later be unmapped. Kernel allocation is undesirable both because the kernel may exhaust its resource pool and also because the kernel is in a poor position to prioritize the importance of different resource allocation requests.

A second issue is that we would like address spaces to have a lifetime independent of any particular process or thread. This requires that mapping-related metadata must be accounted for and associated with particular resource pools. L4 currently does not account mapping structures to resource pools. Indeed, L4 has no kernel-level *notion* of a resource pool.

One solution to these objections is to move more mechanism into the L4 microkernel. We believe that this is counterproductive. The alternative is to move the resource allocation *out* of the microkernel. In the course of describing the mapping management issue to my students, a design emerged that moves the resource allocation out of the kernel. This design seems (to me) to preserve the essential character and benefit of the L4 design while eliminating the need for in-kernel resource allocation.

The balance of this technical report sketches this design.

2 Views

Let a "view" be defined as a tuple:

$$view = (\sigma, unitsize, start, count, access)$$

Where:

σ	some address space
$unitsize$	the size of the units mapped, which is defined as 2^{fp} bytes where $fp \geq log_2(pgsize)$. Note that fp specifies a congruency requirement for all mappings of this view.
$start$	mapping base address, which must be an address $2^{k \geq fp}$
$count$	The number of units in the view. The total size of the view is $(count)(2^{fp})$ bytes
access	a subset of $\{\mathbf{r},\mathbf{w}\}$ describing the maximum authorities permitted by this view

As a special case, we define the view v_{phys} to be a view on the physical address space of the underlying machine. This view defines all of the physical address space whether memory exists at a given location or not, and provides $\{\mathbf{r},\mathbf{w}\}$ access to this space. If it is desired to be careful about the presence of discontiguous physical memory regions, an address space (see below) containing only the accessable areas can be defined.

COMMENTARY

Creating a special case address space σ_{phys} would be a fine alternative to creating a special case view. We have chosen to build a special case view on the logic that the physical mapping is constructed by hardware and operating systems create a view on this physical address space.

3 Mappings

Let a "mapping" be defined as

$$mapping = (view, base, count, access)$$

Where:

$view$	a view as defined above
$base$	the base address at which this mapping appears
$count$	the number of page frames covered by this mapping
$access$	the access rights permitted by the mapping, which are a subset of $\{\mathbf{r},\mathbf{w}\}$. Where the access rights in the view describe the rights *granted*, the access rights in the mapping describe the rights *accepted*.

4 Address Spaces and Translation

We define an address space σ as a sequence (note: *not* a set) of mappings:

$$\sigma = m_0 : m_1 : ... : m_n$$

We define the virtual address translation operation $xlate$ as follows:

$$
\begin{array}{l}
\textbf{define } xlate(\sigma, vpage, access) \rightarrow (ppage, access) \textbf{ as} \\
\quad \textbf{for } n \geq i \geq 0 \\
\quad\quad \textbf{if } (vpage >= \sigma[i].base \textbf{ AND } vpage < \sigma[i].count) \\
\quad\quad\quad access := access \cap \sigma[i].access \\
\quad\quad\quad vpage := vpage - \sigma[i].base \\
\quad\quad\quad \textbf{if } \sigma[i].view == v_{phys} \textbf{ then return } (vpage, access) \\
\quad\quad\quad \textbf{let } view := \sigma[i].view \\
\quad\quad\quad\quad \textbf{if } vpage \geq \text{viewlimit}(view) \textbf{ then raise } \text{InvalidAddress} \\
\quad\quad\quad\quad \textbf{return } xlate(view.\sigma, vpage + view.base, access \cap view.access)
\end{array}
$$

The $xlate$ operator implements a "first match" protocol. An address space is defined as a stacking of mappings, and later mappings shadow earlier ones. For example, it is possible to install a large mapping at position i in the sequence and then place a hole in the middle of that mapping by creating a mapping onto a view of size zero at some position $j > i$ in the address space.

4.1 Bounding the Address Space Sequence

The length of the address space sequence can be bounded at any length greater than 2 without loss of generality. Suppose we declare that address spaces can only have 4 slots, and we wish to add a fifth mapping to some existing address space. Then we create:

$$
\begin{array}{lcll}
empty & = & (\sigma_{phys}, 0, 0, \{\}) & \\
\sigma_{orig} & = & [m_0, m_1, m_2, m_3] & \text{(the original space)} \\
\sigma_{new} & = & [(\sigma_{orig}, 0, 2^{maxpage}, \{\mathbf{r}, \mathbf{w}\}), m_4, empty, empty] &
\end{array}
$$

Because this bound is possible, it is feasible to implement address spaces as specified using fixed-size arrays. My thought is that an array containing 16 entries would be appropriate as the "basic unit" for address spaces.

5 Transferring Views and Creating Mappings

In this design, a view can be copied in unit time. Views may be transmitted during an IPC as follows.

The sender must hold a capability to some address space, and the sender offers a (possibly narrowed) "view" onto this address space. This view is constructed by narrowing some existing view already held by the sender. The sender-side specification is therefore of the form

$$(\sigma, index, offset, count, access)$$

which is understood to mean the view v_{new} constructed by extracting the view v from the mapping found at $\sigma[index]$ and defining v_{new} as:

$$(v.\sigma, v.unitsize, v.start+offset, \max(\min(count, v.count-offset), 0), access \cap v.access)$$

The recipient specifies a "template mapping" of the form:

$$(\sigma, index, base, count, access)$$

when the view is transmitted during an IPC operation, a new mapping is created using the $base$, $count$, and $access$ values provided by the recipient and the view provided by the sender. The resulting view is installed in the recipient-specified slot of the recipient-specified address space.

The fact that the recipient controls both the space and the slot into which the view is installed means that the kernel does not have to allocate storage to remember that a mapping has been created.

This copy operation is not quite the same as either the L4 $grant$ operation or the L4 map operation. It differs from the $grant$ operation because the sender's view is not lost as a result of performing this operation. The sender is always free to install an empty view into their own address space at any time, but no atomic mechanism is provided for removing a view as it is transmitted.

Note, however, that the primary purpose of the L4 map/$grant$ distinction is rendered unnecessary by the introduction of first-class address spaces. The fact that I send you a view does not make the view dependent on me. The view is only dependent on the space from which the view was created, and the lifetime of this space is independent of the lifetime of the sending process (see R-**??**).

To simulate the dependency implications of the L4 map operation, I could create a new address space that transparently maps some existing address space, and always send you mappings relative to that new address space.

6 Differences from L4

There is no direct equivalent in this design to the L4 $unmap$ operation. There are two operations that serve similar purposes.

1. A slot in an address space can be overwritten by anyone holding an address space capability for that address space. This means that the holder of an address space capability may drop mappings at any time.
2. An address space can be destroyed, which erases all outstanding mappings to it.

Supporting address space destruction requires some means to efficiently invalidate all of the outstanding capabilities to a space that are embedded in views.

7 Remaining Issues

There is really only one issue in this design, which is that address space objects are a mix of capabilities and data. I find this distasteful, but it is certainly not fatal.

Given a machine whose address size and word size are the same, each mapping can be encoded as one capability plus four words. the $unitsize$ and $start$ fields of the view can be encoded in a single word. The $count$ and both $access$ fields for the view can likewise be encoded in a single word. The $base$ and $count$ values for the mapping take a third word, and the two access rights (for mapping and view) are encoded in the fourth word.

References

[Lie95] Jochen Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Technical Report GMD TR 933, GMD, November 1995.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, December 1999. ACM.