Some Issues for Single Address Space Systems

Jeff Chase, Mike Feeley, and Hank Levy
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195
{chase,feeley,levy}@cs.washington.edu

At the last WWOS we described Opal, an OS environment that has a single virtual address space common to all protection domains, rather than the usual private virtual address space per protection domain (e.g., a Unix process). All threads on an Opal node see the same mapping of virtual to physical addresses; any thread can reference any virtual address, but access to the data is determined by the thread's protection domain. The single address space approach is made feasible by the large virtual address spaces of the newest RISC processors. It allows protection domains to pass and share memory segments by mutual consent, while preserving the meaning of pointers.

At the time Opal was a paper design, with only a few pieces of implementation. Since that time the key features of Opal have been prototyped [Chase et al. 93]. While the current prototype runs on 32-bit MIPS systems, it demonstrates that the model is both workable and useful. Yet the single address space idea remains a difficult adjustment, both conceptually and practically, for those of us long accustomed to per-process address spaces. Having spent much of the past year arguing the benefits of the model, our purpose now is to outline some of its limitations and complications. The goal is to separate the real problems from the false ones, and to focus the debate on the areas that we think are most important, based on our experience.

1 How and Why

Most aspects of a single address space operating system are unchanged from traditional systems. Opal as prototyped is just another OS personality hosted on Mach, coexisting with Unix. Its primary action is to assign a unique global address range to each virtual memory segment when the segment is created, in order to prevent address conflicts if it is later shared. Segments add structure to the global virtual memory, and the system tracks which segments are accessed from each protection domain. This means that segment-grained access control, symbolic naming of segments, and reclamation of inactive transient segments are much the same as they are in a conventional system. Also, debuggers can easily determine which memory segments are relevant to a given target application.

This work is supported in part by the National Science Foundation (Grants No. CCR-8907666, CDA-9123308, and CCR-9200832), the Washington Technology Center, Digital Equipment Corporation, Boeing Computer Services, Intel Corporation, Hewlett-Packard Corporation, and Apple Computer.

From an application perspective, the main benefit of a single address space is that it broadens the choices for protection structuring of complex software systems. For example, CAD systems are composed of independently developed tools (editors, constraint checkers, simulators) whose inputs and outputs are pointer-based data structures, often passed from one tool to another or read by multiple tools simultaneously. Under Opal, these interacting tools can execute in overlapping protection domains that share access to some but not all of their memory segments. This supports isolation among tools accessing a shared mapped design database, and allows untrusted tools to be run in restricted environments. The global address space ensures that the static structure of the programs and their data is independent of which tools are executing in which domains, and which data segments those domains can access – thus the protection structure is easily changed. We are collaborating with a group at the Boeing Company to use Opal as a substrate for an aerospace CAD system with these properties.

2 Size and Scope of the Address Space

Perhaps the most obvious question raised by single address space systems is: are there enough address bits? Opal views virtual address space as a global resource controlled by the operating system, like disk space or physical memory. This requires accounting and quotas, and the system must reject any request to create a segment that exceeds the available address space (e.g., **malloc(2**64)** fails). Some people find the very idea of system-controlled address space unacceptable, like "charging money for air", but it is not a problem in practice if the address space is large enough to meet legitimate needs. Whether or not the address space is "large enough" depends not just on the address width, but also on (1) the system's ability to recycle addresses, and (2) the scope of the address space. We discuss each in turn.

Opal allocates address space at two levels, and both levels may recycle address ranges assigned to data that is no longer active. At a fine grain, user code fully controls address allocation within segments, i.e., heap block reuse is always permitted. In this case, application code or language-level support must avoid dangling references in the traditional manner, and if it fails it can only harm itself. At a coarse grain, the system may recycle address ranges allocated to reclaimed segments. Here, however, the system must rely on the segment access control mechanism to trap dangling or erroneous references. For this reason, Opal delays address recycling as long as possible.

The single address model can accommodate address spaces of different scopes. That is, a system that defines a single address space for active data on a host can in principle extend that address space to include persistent data, and/or coordinate address usage across a network. A distributed or persistent address space reduces the need for pointer conversion when data moves through the system, at the cost of consuming address space faster. The resulting performance improvement, if any, is determined by the relative speeds of the disk, network, and CPU. For example, memory-to-memory transfers of pointer structures through a fast network would benefit, but not reads from a disk whose latency of access dominates the CPU cost of the pointer conversion. Even if performance is improved, the *mechanisms* for conversion must remain in place to move data across a wide area network or to accommodate heterogeneous machines. In addition, the system must still rely on conventional methods for managing distributed coherency and stable updates to persistent storage.

The current Opal prototype defines one virtual address space per host. This supports physical sharing across programs on the same workstation, and it should fit comfortably in the terabyte+ virtual address spaces supported by the Alpha and others today. We also preserve address bindings for persistent segments (i.e., mapped files). This is convenient but not strictly necessary: [Wilson 91] has shown that pointer-based disk data can be converted (*swizzled*) at low marginal cost, given the high latency of disk access with current technology. (But note that programs using the same data from the persistent store will want to share a cached copy, which must be mapped at common addresses – thus even swizzling systems benefit from a shared *transient* address space.)

3 Global vs. Overloaded Mappings

Opal is a pure single address space system with no address overloading; all threads use the same virtual-to-physical translations. This has three potential advantages: (1) virtual address are unambiguous, (2) private data can easily be converted to shared, and (3) cache aliasing is eliminated. However, these benefits have a cost: since address overloading is prohibited there is no way to copy data without relocating it. In particular, this precludes copy-on-write *for data with internal pointers*, because pointers into the original must be converted if they are to point into the copy.

The lack of support for address overloading causes several specific problems, illustrated below with examples of how they have affected the handling of executable code in our Opal prototype:

(1) *It complicates the representation of private references in shared data.* Code segments in the prototype are statically linked into the global address space, and can therefore be shared and dynamically loaded. However, shared code modules may use writable static data (e.g., global variables) that is *private*, meaning that each domain loading the module sees a separate instance of the data. Shared library implementations for Unix often map the shared module's private data at the same reserved virtual address range in each process; private data references can then be either PC-relative or absolute virtual addresses. This is not possible in Opal because the different instances of the static data must reside at different addresses.

We could have solved this problem with compiler changes similar to [Gingell et al. 87], but for our MIPS-based prototype we decided instead to represent all static references as offsets from the *global pointer* (GP) register. This has certain advantages: it is efficient, the static data can be addressed from multiple domains, and one domain can even maintain multiple instances of the data, e.g., to execute multiple disjoint "processes" in the same hardware protection domain [Yarvin et al. 93]. However, each domain in our prototype can attach only one application module that defines private static data, because of the difficulty of coordinating the GP offsets.

(2) *It can increase the cost of making a private copy of shared data.* In our prototype, initial values for a code segment's private static data must be copied immediately when the segment is attached to any protection domain. We cannot easily use a lazy copy because a static data item may be initialized to the address of another piece of static data, and these pointers must be converted as part of the load step.

(3) *It can force early decisions about when a private copy of shared data might be needed.* We cannot set breakpoints in shared modules in our prototype: the programmer must know at build time that the code will undergo debugging, and link to a private copy of the procedures installed in a different part of the address space from the shared copy. This is because of the difficulty of relinking if code pages are copied on-the-fly.

These examples could be seen as artifacts of the way we chose to link code modules in our prototype, but they are important for two reasons. First, each example merely illustrates a general problem that could be reflected in other situations, such as copying segments for backups or versioning. Second, they illustrate that a single address space can have subtle effects on seemingly isolated aspects of the computing environment. For example, the problems with static data turned out to be significant only because some compilers on the MIPS treat constants and other readonly items as initialized writable static data. Unix does not penalize this choice as much, because it uses copy-on-write to defer the copying overhead – or avoid it altogether.

Some of these problems can be avoided by allowing some address ranges to have different translations in each protection domain. For example, [Garrett et al. 93] statically partition the address space into regions that are *private* (with per-process translations) or *shared* (with global translations). Like Opal, this hybrid scheme supports shared pointer structures, and unlike Opal it is backward compatible with traditional processes created with Unix-style **fork**. But the same problems with copying continue to afflict the shared regions, and the programmer must make early decisions about whether data is private or shared.

4 Address Translation

An alternative solution to the problems of the previous section might be to allow any process to make a private shadow copy of any shared region. This appears to complicate the VM system unacceptably. The technology for managing a large shared address space – hashing TLB misses into a global resident page table, backed by per-segment linear page tables (e.g., inodes) for handling page faults – is well-understood and in widespread production use [Huck & Hays 93]. The best approach to managing very large private address spaces is less clear, and it is not at all clear how to handle arbitrary conversions between private and shared regions.

Our Opal prototype is capable of mixing private and shared regions in this way, but only because it works by coordinating address usage across private address spaces supported by the native Mach VM system. Thus the mappings in the global resident page table are cached, often replicated, in per-domain linear page tables that feed the TLB handler. This is fine for an experimental prototype, but it increases the cost of shared data, and it will not scale to 64-bit address spaces. A native Opal system using a global translation table would not support private regions as cheaply.

5 Conclusion

A single address space system with protection domains makes a different set of tradeoffs from traditional systems based on private address spaces. But note that old programs that do not attempt to use the new features are not affected by any of the complications we have outlined. Old programs are affected in two ways: they cannot use the entire address space supported by the hardware, and they see a sparse and noncontiguous view of virtual memory. These restrictions should not affect ordinary programs written in high-level languages.

References

[Chase et al. 93] Chase, J. S., Levy, H. M., Feeley, M. J., and Lazowska, E. D. Sharing and protection in a single address space operating system. Technical Report 93-04-02, University of Washington, Department of Computer Science and Engineering, April 1993.

[Garrett et al. 93] Garrett, W., Scott, M., Bianchini, R., Kontothanassis, L., McCallum, R., Thomas, J., Wisniewski, R., and Luk, S. Linking shared segments. In *Proceedings of the Winter 1993 Usenix*, January 1993.

[Gingell et al. 87] Gingell, R. A., Lee, M., Dang, X. T., and Weeks, M. S. Shared libraries in SunOS. In *Proceedings of the Summer 1987 Usenix*, June 1987.

[Huck & Hays 93] Huck, J. and Hays, J. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[Wilson 91] Wilson, P. R. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *ACM SIGARCH Computer Architecture News*, 19(4), June 1991. University of Illinois at Chicago Technical Report UIC-EECS-90-6, December 1990.

[Yarvin et al. 93] Yarvin, C., Bukowski, R., and Anderson, T. Anonymous RPC: Low latency protection in a 64-bit address space. In *Proceedings of the Summer USENIX Conference*, June 1993.