Distribution in a Single Address Space Operating System

Jeff Chase, Valérie Issarny, and Hank Levy

Department of Computer Science and Engineering FR-35
University of Washington
Seattle, WA 98195
USA

Abstract

The recent appearance of architectures with flat 64-bit virtual addressing opens an opportunity to reconsider the way our operating systems use virtual address spaces. We are building an operating system called Opal for these wide-address architectures. The key feature of Opal is a single global virtual address space that extends to data on long-term storage and across the network. Hardware-enforced memory protection exists within this single address space.

This paper outlines our ideas for extending Opal to a distributed environment, focusing on the naming and binding of data and services to allow uniform treatment across the network. Our central point is that although the *meaning* of names (i.e., the entities denoted by those names) should be uniform throughout the network, at a lower level the *binding* of names to physical data or servers may vary with the node uttering the name, in order to accommodate caching, replication, and migration. This principle affects Opal's handling of both data names (virtual addresses) and resource names (capabilities).

1 Introduction

The Opal project is an investigation into the effect of wide-address architectures on the structure of operating systems and applications. Our premise is that the next generation of workstations and servers will use processors with 64-bit data paths, and sparse, flat, 64-bit virtual addressing. The MIPS R4000 [MIP 91] and Digital's Alpha family [Dig 92] are recent examples of the trend to wider addresses. The move to 64-bit addressing is a qualitative shift that is far more significant than the move to 32-bit architectures in the 1970s. 64-bit architectures remove basic addressing limitations that have driven operating system design for three decades: a full 64-bit address space consumed at the rate of 100 megabytes per second will last for 5000 years. On 32-bit architectures, virtual addresses are a scarce resource that must be multiply allocated in order to supply executing programs with sufficient name space. On 64-bit architectures it is no longer necessary to reuse virtual addresses in this way; the system may instead assign globally unique virtual addresses to data.

We are building a 64-bit operating system called Opal with a *single virtual address space* that maps all primary and secondary storage across a network [Chase et al. 92a, Chase et al. 92b]. The principal advantage of a single address space is that virtual addresses have a globally unique interpretation – a given piece of data appears at the same virtual address regardless of where it is stored or which programs access it. This context-independence simplifies sharing of code and data, and it supports a uniform single-level virtual store, allowing data structures to be saved on secondary storage without the overhead of converting pointers to a separate format. We believe that these features are important for integrated software environments.

The next section outlines Opal's abstractions for data storage, protection, and communication. Section 3 describes our plans for extending this scheme to accommodate distribution, focusing on the naming, binding, and protection of data and resources.

This work was supported in part by the National Science Foundation under Grants No. CCR-8619663 and CCR-8907666; by the Washington Technology Center; and by Digital Equipment Corporation through the Systems Research Center, DECwest Engineering, and the External Research Program. Valérie Issarny is supported by a grant from INRIA.

2 Virtual Storage and Protection in Opal

Opal's global virtual address space is partitioned into *virtual segments* of varying size. Segments are *logical* groupings of pages treated as a unit for access control or storage management. In particular, segment boundaries are unknown to the hardware, and addressing is independent of segments; all memory references use a fully qualified flat virtual address. Each segment occupies a fixed range of virtual addresses, assigned when the segment is created, and disjoint from the address ranges occupied by all other segments in the system. Segments are potentially distributed and persistent, but the storage management policies are not dictated by the kernel. Instead, consistency and recoverability are managed by external paging servers similar to Mach [Young et al. 87] and Chorus [Abrossimov et al. 89].

Memory protection is independent of Opal's global virtual address space. Though an executing program can attempt to address any piece of data in the system, programs execute within *protection domains* that limit their access to global virtual storage. An Opal protection domain is in some ways the analog of a Unix process. For example, there is typically one domain per executing application, containing some private data, threads, and RPC connections to other domains. The difference is that an Opal domain is a private set of access privileges for globally addressable segments, rather than a private virtual naming environment. Opal domains can grow, shrink, or overlap arbitrarily, by sharing or exchanging segment permissions with other domains. The uniform address space encourages sharing because linked data structures in shared segments are always meaningful to any domain; pointers can be passed in shared or persistent storage without name conflicts or pointer translation. One goal of our research is to understand the uses and effects of this sharing.

It is sometimes unsafe to allow a protection domain to access a given piece of data directly. We use the term *resource* to mean a piece of data (an *object*) managed by a server domain through a protected RPC interface. For example, a resource might be an instance of a system service, such as a file or an entry in a name server. Since a server may serve many resources, client domains need some way to name a resource and demonstrate permission to operate on it. Resource objects are named by capabilities based on *sparse capabilities* in Amoeba [Mullender & Tanenbaum 86]. Any domain that knows the value of a sparse capability can gain access to the named resource, but it is safe to assume that the value cannot be guessed. We chose sparse capabilities over kernel-controlled capabilities (as in Mach) so that capabilities can be passed through shared or persistent memory, just like ordinary pointers.

An Opal capability is a 256-bit value. The first half of a capability is resolved by the system to locate the server; it contains a 64-bit *portal* number identifying an RPC endpoint registered for some server domain.[1] The second half of a capability identifies an object managed by the server, and is chosen and resolved by the server; it contains a 64-bit object identifier and a 64-bit check field used to validate the capability. The system does not dictate a format for these fields. However, we expect that in the common case, the object identifier will simply be the virtual address of the object, thus we often refer to capabilities as *protected pointers*.

Virtual segments are examples of protected resources named by capabilities. A segment capability confers permission to the holder to *attach* the segment and access its contents directly with **load** and **store** instructions. The kernel resolves hard and soft page faults with upcalls to the segment's paging server through the portal named by the segment capability, which is passed to the kernel as an argument to the *attach* call.

[1] Portals are analogous to *ports* in Chorus or Amoeba. They are called "portals" in Opal because they are not message queues, but rather are entry points for switching control to a new protection domain. Any thread that knows the name of a portal can make a domain switch through the portal; conceptually, the thread begins executing in the server domain at a virtual address chosen by the server and specified when the portal is registered. The portal model was chosen as a basis for implementing local cross-domain calls using the *lightweight RPC* (LRPC) techniques described in [Bershad et al. 90].

3 Distribution in Opal

Opal's single virtual address space is intended to span a network of independent nodes. Each node contains physical memory and one or more processors attached to that memory. We assume that the processors have a homogeneous addressing model (e.g., all processors have wide virtual addresses). We are initially restricting our focus to local area networks, but we believe that a 64-bit address space is large enough to accommodate a network of about 10,000 nodes: this leaves 1600 terabytes of virtual address space per node.

Our basic approach to managing distribution is to extend the global virtual address space and portal name space across the network by placing servers on each node that cooperate to partition the name spaces. There are two logically separate servers on each node:

(1) The *address space service* coarsely partitions the virtual address space, assigning large ranges to segment servers which subdivide them into segments.

(2) The *portal service* assigns unique portal names to services, and can locate a server given a portal name.

We do not discuss the manner in which these servers cooperate with their peers on other nodes – this is an important issue, but the problems are similar to other systems for distributed name management. Instead, we focus on the key design question for the management of distribution in Opal, that is, how data and resource names are assigned and resolved, and the differences between name resolution within a single node and across the network.

3.1 Resolving Data Names

Within each node, Opal maintains a single mapping of virtual to physical pages. Enforcing such a constraint across multiple nodes would be too restrictive, because caching of multiple copies of a page is crucial for performance. Furthermore, the system should tolerate *inconsistency* of cached pages in order to allow different coherency models, such as weak coherency (e.g., [Gharachorloo et al. 90]). As another example, object-based distributed virtual memory systems[2] rely on memory that is physically noncoherent across nodes, but use language-level knowledge to make objects appear coherent to the application.

We wish to allow maximum flexibility for implementing a wide range of coherency protocols, rather than dictating a single model. For this reason the "single virtual address space" simply means that segment servers are assigned unique virtual address ranges to carve into segments. The Opal kernel will not allow segment servers to assign multiple physical pages to the same virtual page within a node. However, the meaning of references to a segment across nodes is defined by the manner in which the service handles page faults and controls page protections on the various nodes accessing the segment; no single virtual-to-physical mapping is enforced by the system. The coherency guarantees for a particular segment are simply a "contract" between the clients who attach to the segment and the service that backs the segment.

3.2 Resolving Resource Names

We believe that resource names should also be resolved according to the node where they are referenced, for similar reasons. It is frequently useful to structure a service as a group of cooperating servers. For example, consider a segment with page-based distributed coherency. Experiments in Ivy [Li & Hudak 86]

[2] By this we mean a distributed virtual address space with coherency maintained at the object grain, notably Amber [Chase et al. 89], as opposed to distributed object systems that use surrogate names for objects, such as Emerald [Jul et al. 88] and Orca [Bal & Tanenbaum 88].

and Mach [Forin et al. 89] have shown that the best performance is achieved by placing a paging server on each node, and allowing the server to cooperate with peer servers on other nodes, rather than centralizing the paging server. Each server acts as an *agent* through which local clients access the distributed service. We believe that such a structuring of services as a group of cooperating servers will be a common design choice. The agents may keep replicated copies of the service's data for performance, reliability, or availability. As a noticeable example, the basic Opal services that assign virtual address and portal names are themselves distributed services configured in this way.

The structuring of distributed services has implications for how capabilities are resolved. For instance, let us consider that a domain on node **B** attempts to attach a segment created by the server on node **A** of a distributed segment service. **A** passes **B** a capability for the segment, containing a portal name and an object identifier meaningful to the service. The capability was minted on **A**, but when the segment is attached on **B** the capability should resolve to the closest server, e.g., the agent on **B**. In Mach, this requires that the client explicitly translate the capability by calling its local agent before it can attach the segment. This means that the client must have knowledge of where the capability was created and which local server to call. We believe that this should be handled transparently by the system.

For this reason, Opal portal names are unique within a node, but may have multiple bindings across the network. That is, there is a unique binding of a portal to a *service*, but a service may be structured as a group of agents, each of which may register the same portal name. Of course, this approach assumes that the object identifier embedded in the capability is resolved to the same resource by each agent. Resource location and caching can be handled automatically if the agents cooperate using distributed shared memory or a distributed data object system (e.g., Emerald, Orca, or Amber). This raises questions about the suitability of existing programming systems for implementing distributed services.

Node-dependent portal bindings are related to the notion of *port groups* in Chorus and Amoeba. In these systems, a port group name embedded in a capability identifies a set of eligible servers to handle a request. The request may then be forwarded to the closest server among those eligible or even broadcasted to all the servers, as determined by the client. However, our approach is different in that the distributed nature of a service is encapsulated within the service. Our "agents" are also similar to *proxies* [Shapiro 86]. We use a new term to emphasize that the proxy is isolated from the clients by a memory protection boundary.

4 Summary

This paper has described the high-level concepts of Opal and its support for distribution, concentrating primarily on name resolution for data and resources. Opal is intended to execute on a network of 64-bit address space computers. The key concepts of Opal are:

(1) the use of a unique, global, virtual address space, encompassing all data and long-term storage for all executing threads, and

(2) the separation of protection and addressing through the use of protection domains that limit a thread's access to the global address space.

The principal objective of this approach is to enhance sharing of complex data through the use of a single virtual address space, which guarantees that pointers have the same *meaning* independent of the context in which those pointers are used.

Managing distribution, however, requires attention to the performance implications of the network. Gaining performance in the face of distribution may require caching and a weakening of the coherency guarantees provided on a single node. Different applications may wish to receive different guarantees. For this reason, Opal does not dictate a single model for managing distribution; instead, it provides a framework within which multiple models can be implemented. In general, for each model, an Opal service running within the network will support distributed segments (or objects) with identical distribution semantics. The service is a *logical* entity, realized through distributed servers and agents located on Opal nodes.

Thus, while addresses when issued by multiple entities on a single node will always access the same physical memory location, we relax this guarantee within the network. While the same address always refers to the same *logical* data, multiple copies of that data may exist across nodes; the coherency of those copies is a contract between the cooperating applications and the service supporting distribution of the shared segment.

Higher-level resources on Opal are managed through services and are addressed by capabilities; a capability refers to a unique portal attached to the service. As with virtual memory addresses, a portal number is unique when used within a node, but may refer to different copies of the logical service when used by different entities within the network. This distinction between logical service and physical instance, supported by the naming system, allows the application substantial flexibility in deciding the degree of centralization or distribution to use, as well as permitting for dynamic changing of that decision.

Opal is currently being prototyped on a network of MIPS-based workstations at the University of Washington.

References

[Abrossimov et al. 89] Abrossimov, V., Rozier, M., and Shapiro, M. Generic virtual memory management for operating system kernels. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 123–136, December 1989.

[Bal & Tanenbaum 88] Bal, H. E. and Tanenbaum, A. S. Distributed programming with shared data. In *Proceedings of the International Conference on Computer Languages*, pages 82–91, October 1988.

[Bershad et al. 90] Bershad, B., Anderson, T., Lazowska, E., and Levy, H. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1), February 1990.

[Chase et al. 89] Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M., and Littlefield, R. J. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 147–158, December 1989.

[Chase et al. 92a] Chase, J. S., Levy, H. M., Baker-Harvey, M., and Lazowska, E. D. How to use a 64-bit virtual address space. Technical Report 92-03-02, University of Washington, Department of Computer Science and Engineering, February 1992.

[Chase et al. 92b] Chase, J. S., Levy, H. M., Baker-Harvey, M., and Lazowska, E. D. Lightweight shared objects in a 64-bit operating system. Technical Report 92-03-09, University of Washington, Department of Computer Science and Engineering, March 1992.

[Dig 92] Digital Equipment Corporation, Maynard, MA. *Alpha Architecture Handbook*, preliminary edition, 1992.

[Forin et al. 89] Forin, A., Barrera, J., and Sanzi, R. The shared memory server. In *Proceedings of the Usenix Conference*, pages 229–242, Winter 1989.

[Gharachorloo et al. 90] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Annual Symposium on Computer Architecture, Computer Architecture News*, pages 15–26. ACM, June 1990.

[Jul et al. 88] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Li & Hudak 86] Li, K. and Hudak, P. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[MIP 91] MIPS Computer Systems, Inc., Sunnyvale, CA. *MIPS R4000 Microprocessor User's Manual*, first edition, 1991.

[Mullender & Tanenbaum 86] Mullender, S. and Tanenbaum, A. The design of a capability-based operating system. *The Computer Journal*, 29(4):289–299, 1986.

[Shapiro 86] Shapiro, M. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, May 1986.

[Young et al. 87] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.