Opal

Robert Grimm
New York University
The Three Questions

- What is the problem?
- What is new or different?
- What are the contributions and limitations?
Technology Change Provides Opportunity

- **Technology**: 64-bit address space architectures
  - DEC Alpha, HP PA-RISC, MIPS R4000
- **Opportunity**: Separate addressing and protection
  - Enhance sharing
  - Simplify integration
  - Improve reliability
- **Approach**: Use a *single* virtual address space
  - Without special-purpose hardware
  - Without loss of protection or performance
  - Without requiring a single type-safe language
Other Operating Systems: Private Address Spaces

- **Advantages**
  - Increase amount of address space for all programs
  - Provide hard memory protection boundaries
  - Permit easy cleanup

- **Disadvantage**
  - Hard to efficiently share, store, or transmit information

- **Two unattractive work-arounds**
  - Independent processes that use pipes, files, messages
  - One process
Better Cooperation Through Shared Memory

- With private address spaces (think Unix `mmap`)
  - Requires a-priori *application* coordination
    - Shared regions must be known in advance
  - Can lead to problems
    - Private pointers in shared regions

- With a single address space
  - Each address has a single meaning, across all protection domains
  - System (rather than applications) coordinates address bindings
Some Notes on Sharing, Trust, and Distribution

- Completely disjoint protection domains are too confining
  - Trust relationships may be asymmetric (read-only access)
  - Protection and trust may be orthogonal
    - Same database program working for different users
    - System protection should not impose modularity!
- Single address space can simplify distribution (when shared across nodes)
  - No pointer swizzling, marshaling, or translation
  - But, this only reduces frequency of these conversions
Opal Abstractions and Mechanisms
Storage and Protection

- Unit of storage allocation is a *segment*
  - Contiguous extent of virtual pages (typically >> 1)
- Unit of execution is a *thread*
- Execution context for a thread is a *protection domain*
  - Provides a *passive* context (instead of an *active* process)
  - Restricts access to a particular set of segments
    - Enforced through page-based hardware mechanisms
    - Does this sound familiar?
- Storage allocation, protection, and reclamation should be *coarse-grained*
  - Fine-grained control best provided by languages / runtime
Access Control

- All kernel resources are named by *capabilities*
  - User-level, *password-style* 256-bit references
    - How does this compare to Mach or Hydra capabilities?
- Name service provides a mapping between symbolic names and capabilities, protected by ACLs
- Segments are *attached* and *detached* by presenting the corresponding capabilities
- Segments can also be attached transparently on address faults
  - Runtime handler retrieves published capability and performs *attach* operation
Inter-domain Communication

- Calls between domains build on *portals*
  - Have we seen a similar concept?
- Portals are identified by a 64-bit ID
  - Entry point is at a fixed, global virtual address
- Password-capabilities build on portals (and RPC)
  - Portal ID, object address, randomized check field
- Language support makes cross-domain calls transparent
  - Client capabilities hidden in *proxy* objects
  - Server *guards* contain check field
Using Protection Domains

- Parents create new domains
  - Attach arbitrary segments
  - Execute arbitrary code
- Protected calls are the only way to transfer control
  - Register a portal
  - Specify a procedure as the entry point
  - Call through that portal
- Rights amplification only possible through privileged server
Linking and Executing Code

- Linking and execution essentially the same
  - Resolve symbols to actual addresses on a *global* level
    - Shared libraries become trivial; they are the default
    - Procedure pointers can be passed directly
    - No possibility of address conflicts

- Private static data requires extra mechanism
  - Private static data must exist at different virtual addresses
  - Linker uses register-relative addressing
  - Base register addresses assigned dynamically
    - Based on instance of module and thus of protection domain
Resource Management and Reclamation

- Storage management based on reference counting
  - Builds on assumption of coarse-grained allocation
  - Does not reflect number of capabilities
  - Rather, indicates general interest in resource (attach/detach)

- What to do about buggy or malicious programs?
  - Hierarchical resource groups
    - Provide *unit of accounting*; implicitly associated with each thread
    - Charges flow up the hierarchy; deletions flow down the hierarchy
  - Reference objects
    - Separate accounting between different references
    - May also imply different access rights
Summary

- Basic abstractions
  - Protection domains, segments, portals, resource groups
- Applications structured as groups of threads in overlapping protection domains
- Resource reclamation based on reference counts
  - Accounting and bulk delete through resource groups
- Address pointers and capabilities are freely shared
(Memory) Protection is decoupled from
- Program execution (RPC)
- Resource naming (capabilities based on portals)
- Resource ownership (resource groups)
- Virtual storage (segments)

Proxies can make RPC transparent to applications
- Runtime facility!
Implementation & Evaluation
Prototype Implementation

- Built on top of Mac 3.0 microkernel
  - Opal server implements basic abstractions
    - Segments, protection domains, portals, resource groups
  - Runtime package provides C++ API to applications
    - User-level threads, capability-based RPC, proxies, heap management
    - Linking utilities assign persistent virtual addresses
- Co-hosted with Unix server
  - Simplifies implementation by leveraging Unix file system
Mapping Opal onto Mach

- Protection domains are Mach tasks
  - Execution context for threads
- Segments are Mach memory objects
  - Virtual memory region backed by user-mode paging server
    - Server uses *inodes*, thus making them accessible through Unix FS
- Domains have Mach ports
  - End-points for receiving messages
  - One port for each domain multiplexes onto all portals
Some Implementation Details

- Opal server maps segments to address ranges
  - But it also contains a mapping from addresses to segments
  - Why?

- Opal server maps domains to Mach port send rights
  - Runtime caches mappings
  - Why?

- Segments are backed by Unix files
  - Address management structures of Opal server also in persistent segment
  - What problem does this raise? How is it solved?
**Applications and Performance**

- Boeing might benefit
  - Humongous aircraft parts database
    - Maintains relationships between parts
    - Consumed by several tools (simulation, analysis)
- A tree-indexing program does benefit
  - Both performance and protection!
- Micro-benchmarks dominated by (sucky) Mach performance
Issues

- How to ensure contiguity of memory?
  - There needs to be a limit on the largest segment size

- How to conserve address space?
  - Heap managers reclaim memory
    - Dangling references only affect programs using heap
  - Opal might reclaim segments
    - What about dangling references?

- How to support a Unix-style *fork*?
  - Create multiple concurrent processes
  - Initialize child’s state
How to avoid data copying?

- Copy-on-reference needs to replace copy-on-write when data is explicitly copied
- But copy-on-write is still possible (!?)
Discussion