Opal

Robert Grimm
New York University

The Three Questions

- What is the problem?
- What is new or different?
- What are the contributions and limitations?

The Three Questions

- What is the problem?
  - Applications share large data-set with links between items
- What is new or different?
  - Separation of protection domain and (memory) addressing
    - Leverages technology opportunity: 64-bit address spaces
- What are the contributions and limitations?
  - Identification of problem, hardware trend
  - Exploration of single, virtual address space
  - Register-relative addressing for private data "hack-ish"

Technology Change as Opportunity

- Technology: 64-bit address space architectures
  - Then: DEC Alpha, HP PA-RISC, MIPS R4000
- Opportunity: Separate addressing and protection
  - Enhance sharing
  - Simplify integration
  - Improve reliability and performance
- 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 OS's: Private Address Spaces

- Advantages
  - Increase amount of address space for each program
  - Provide hard memory protection boundaries
  - Permit easy cleanup
- Disadvantage
  - Hard to efficiently store, share, 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
  - Provides only limited and somewhat brittle sharing
    - Private pointers in shared regions
- With a single address space
  - Each address has a single, global meaning across all protection domains
  - System (rather than applications) coordinates address bindings

Some Notes on Sharing, Trust, and Distribution

- Completely disjoint protection domains 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, marshalling, or translation
    - Actually, conversions become less frequent

Opal Abstractions and Operations

Memory and Protection

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

Access Control

- All kernel resources are named by *capabilities*
  - User-level, *password-style* 256-bit tuples
    - How does this compare to Hydra or Mach?
- Name service provides mapping between 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
  - Portal ID, object address, randomized check field
- Language support makes cross-domain calls mostly transparent through RPC
  - Client capabilities hidden in *proxy* objects
  - Server *guards* contain check field

Using Protection Domains

- Parent creates new domain
  - Attaches arbitrary segments
  - Executes arbitrary code
- Protected calls are the only way to transfer control
  - Register a portal with procedure as entry point
  - Hand out capability so that clients can call through portal
- Rights amplification only possible through more privileged server

Linking and Executing Code

- Linking and execution become easier
  - Symbols resolve to virtual addresses on a *global* level
    - Shared libraries are trivial; they are the default
    - Procedure pointers can be passed directly
    - No possibility of address conflicts
- Linking and execution become harder
  - 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

- Segment management based on reference counting
  - Builds on assumption of coarse-grained allocation
  - Does not reflect number of capabilities
  - Rather, indicates "general interest" in resource (at-/de-tach)
- What to do about buggy/malicious code?
  - Hierarchical resource groups
    - Provide *unit of accounting*, associated with each thread
    - Charges flow up hierarchy; deletions flow down hierarchy
  - Reference objects
    - Separate accounting between different references
    - May also imply different access rights

Summary

- Basic abstractions
  - Segments, threads, protection domains, portals, capabilities, resource groups
- Applications structured as groups of threads running in (overlapping) protection domains
- Addresses and capabilities freely shared

Summary (cont.)

- Process abstraction broken down into
  - Program execution (threads and RPC)
  - Resource protection (domains, portals, and capabilities)
  - Resource ownership (resource groups)
  - Virtual storage (segments)
- Proxies can make RPC transparent to applications
  - Runtime facility!

Opal Implementation and Evaluation

Prototype Implementation

- Built on top of Mach 3.0 microkernel
  - Opal server provides most abstractions
    - Protection domains, segments, portals, resource groups
  - Runtime package provides simple 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 boot-strapping and development
  - Simplifies management 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 back by user-level paging server
    - Server uses inodes, which are accessible through Unix FS
- Domains have Mach ports, with Mach thread listening
  - 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)
- Two cooperating tree-indexing programs do benefit
  - Both in terms of 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 Unix-style *fork*?
  - Use multiple threads or initialize child's state before-hand

Issues (cont.)

- 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