CSE221 Lecture 9

Aronya Baksy

October 2024

1 Microkernel Design

- **Goal**: move as much OS functionality to user level as possible
- Monolithic kernels: e.g. Linux
- Mach, with microkernel exposing basic IPC primitives running in privileged level, and virtmem, FS, apps all running at user level
- L4, later versions of Mach: OS and apps runs in user land, with very minimal microkernel at the kernel space

2 The performance of microkernel based systems

2.1 Goals

- show that microkernel based systems are usable in practice with good performance.
- Enhance and optimize existing OS services, and show that it is extensible
- Ask if co-location is required - if OS can run either in user mode (for easy debugging and dev) or kernel mode
- Focus on the problem of porting an existing monolithic operating system (Linux) to a microkernel instead of building from the ground-up

2.2 L4 Microkernel

- Core abstractions: **threads**, **address spaces**, **IPC** (messages)
- construct recursive address spaces created by user-level applications via grant, map, unmap operations
- dynamically associate pages with process or threads
- IO treated as parts of address space to be mapped and unmapped
- Interrupts and traps are synchronous to raising thread, mirrored by kernel in user space
- small address spaces can be used up to 512MB for hardware support. (only on Pentium). after 512MB switch back to "normal" 3GiB mode
- Works on Pentium originally, ported to Alpha and Mips processors: same APIs, but different implementations

2.3 Linux on L4

- Modify the architecture dependent parts of Linux and try to maintain the independent parts
- L4 hides physical page tables so Linux kernel running on top of L4 has to maintain its own logical tables
- L4 maps hardware interrupts to messages which are then received by kernel threads: top half interrupt executes, then bottom halves
- Each Linux user process is an L4 task, i.e. an address space together with a set of threads executing in this space.

- Linux kernel threads operate in a tiny address space simulating a tagged TLB
- Each Linux user-level process has an additional signal-handling thread which listens from signals from the Linux kernel and shifts the user process into the kernel upon receiving a signal
- L4 uses physical copy in/out to exchange data between kernel and user process
- L4 handles linux scheduling, so not much to do when syscall completes and reschedule bit is not set, put process back onto the CPU
- TLBs are critical to keeping performance for memory hits: need smaller and more compact address space layouts to avoid conflicts to prevent TLB misses.

2.3.1 System Calls

- Modified version of libc sends a message to L4 which routes it to the message queue for Linux and wake it up so that it can read from the message queue and handle the syscall
- Linux then sends it back to L4 via another message

2.3.2 Page Fault

- Look up TLB, upon a miss trap to the L4
- L4 routes page fault to Linux, Linux allocates free page, updates its own page table and sends a message to L4
- Linux is modified to make a syscall to update the L4 page table (i.e. shadow page table) each time it updates its own virtual table
- L4 has its own page table, Linux has its own virtual page tables,

2.3.3 Memory Organization

- Hierarchical, L4 grants memory to Linux, and Linux can delegate it to its children (similar to Nucleus paper)
- Each pager has its own address space

2.4 Evaluation

- 5-10% slower than monolithic kernel, acceptable on personal computing devices
- Performs better than co-located Mach system on lmbench and hbench benchmarks
- Extensibility: cache partitioning, optimized pipe implementation wchich removes copies to kernel, user-level paging

3 Exokernel: An OS Architecture for Application-level resource management

3.1 Goals

- Provide app-level mgmt of physical resources instead of a fixed interface & implementation
- Push "traditional" OS functionality to the OS level
- Main principle: Lower level primitives are more flexible and easier to optimize, separate protection from management (i.e. separate policy and mechanism)

3.1.1 Exporting resources

- Tracking **ownership** of resources using table structures
- **Secure Bindings** - Enable applications to securely interact with resources through efficient, application-defined protection checks.
- **Visible Resource Revocation** - Allows applications to actively manage resource deallocation, improving efficiency and reducing kernel intervention.
- **Abort Protocol** - Ensures stability by reclaiming resources from unresponsive applications.

3.2 Problems of High-Level Abstractions

- Cost of abstraction:
  - Hurt perf. because there is no single way that resources can be abstracted which is best for all apps
  - tradeoffs taken for generality's sake can hurt apps which work best with specific patterns
  - e.g. RDBs and GCs have specific mem. access patterns and suffer when LRU is used
- "End to end argument":
  - Apps should have control over their own resource mgmt
  - Each app is statically linked with a library OS that runs at the user level
  - Library OSs connect to secure bindings provided by the exokernel interface, and applications run on Library OSs.
  - Library OSs provide application specific policies and implementations
  - Applications can be portable if Library OSs are compatible with standards like POSIX

3.3 Benefits vs L4

- Less message passing overhead, particularly for system calls
- More specialization and extensibility than L4

3.4 Design Principles

- **Securely Expose Hardware**: Exokernel should provide low-level primitives that allow applications to access hardware resources as directly as possible, safely exporting all privileged operations and resources (minimizes the kernel's role in managing resources beyond essential protection)
- **Expose Allocation**: Library OSs should be involved in each allocation decision and have the ability to request specific resources.
- **Expose Physical Names**: Exokernel should use physical resource names (such as physical page numbers) instead of abstract ones, eliminating extra translation layers and giving applications insight into resources' inherent properties (e.g., cache behavior).
- **Visible Resource Revocation**: Revocation should be visible to applications, enabling them to manage and release resources efficiently. This cooperative approach allows applications to maintain better control and adapt to resource availability.
- **Decentralized Policy Decisions**: The Exokernel delegates resource management policy to applications, retaining only the minimal necessary control over resource allocation and revocation. This flexibility enables application-specific optimizations while supporting traditional policies as needed.

3.4.1 Memory Management

- Each lib OS maintains its own page table
- Exokernel tracks list of allowed pages per process
- Illegal memory access traps to exokernel, exokernel makes an upcall to the LibOS asking for the PTE, once libOS responds then exokernel checks ownership and updates TLBs with the correct translation
- Extra overhead in updating PTEs, upcalls etc.

3.5 Secure Bindings

- Protection mechanism that decouples authn from the resource
- Protection checks are fast and simple
- Mgmt decoupled from protection because protection checks only at bind time
- Hardware based protection e.g. file server buffering data in memory pages provides capabilites to authorized apps, and exokernel need not be aware of this mechanism.
- Software TLB caches frequently used bindings
- When a library OS allocates a physical memory page, the exokernel creates a secure binding for that page by recording the owner and the r/w capabilities specified by the library OS.
- Every access to a physical memory page requires that library OS present the capbility
- All privileged h/w operations (TLB/DMA manipulation) must be guarded by the exokernel
- The kernel implements a programmable **packet filter**, which executes programs in a bytecode language designed for easy security-checking by the kernel.
- If a library OS does not respond to revocation requests within some timeout, the exokernel can simply break the secure bindings for that resource (referred to as the *abort protocol*).

3.6 Aegis and ExOS

- Aegis is an exokernel implementation, ExOS is the library OS on top of it
- Aegis exports the processor, physical memory, TLB, exceptions, and interrupts.
- ExOS implements processes, virtual memory, user-level exceptions, various interprocess abstractions, and network protocols (IP, UDP, NFS)
- The kernel represents the processor resources as a timeline from which programs can allocate intervals of time.
- A program can yield the rest of its time slice to another designated program (**synchronous control transfer** also adds all future time slices to this, in contrast to **async control transfer**)
- The kernel notifies programs of processor events, such as interrupts, hardware exceptions, and the beginning or end of a time slice.
- If a program takes a long time to handle an event, the kernel will penalize it on subsequent time slice allocations; in extreme cases the kernel can abort the program
- small and lean kernel improves performance: it keeps its data structures in physical memory.
- by caching secure bindings in a software TLB, most hardware TLB misses can be handled efficiently.

3.7 Extensibility Benchmarks

- Series of benchmarks that show the benefits of extensibility that ExOS and Aegis provides
- Different RPC mechanisms with optimizations that trust RPC server to save callee's registers improve performance 2x (tlrpc vs lrpc)
- Use of inverted page tables works better for apps with sparse addresses spaces, while apps with dense address spaces benefit from linear page tables
- Apps define custom scheduling using the Aegis `yield` primitive, testing *stride scheduling* allows throughput improvements

3.8 Conclusions

- **Efficient Implementation**: The Exokernel's simple, low-level primitives enable efficient implementation and keep the kernel small
- **Effective Multiplexing**: Secure, low-level resource multiplexing can be achieved without significant overhead.
- **Application-Level Abstractions**: Fundamental abstractions, such as virtual memory and interprocess communication, are implemented more efficiently at the application level than in traditional OS models.
- **Customizability for Applications**: Applications can create custom implementations of operating system abstractions by modifying library OSs rather than the kernel itself