

On m-Kernel Construction

Liedtke 1995

What kind of paper is this?

- Refute conventional wisdom.
- Big idea (μ -kernels are fast enough).

The Story

- μ -Kernels have gotten a bad rap
- They are considered inefficient and inflexible
- The problem is not the μ -Kernel idea but the implementation
- μ -Kernels should **not** be portable
- If you build a μ -Kernel correctly, then you get good performance AND flexibility
- So, what should you do?
 - Reason about m-Kernel concepts

The m-Kernel Concepts

- Only put in the kernel that which, if moved outside the kernel, would prohibit functionality
- Assumptions
 - Require security
 - Require page-based VM
- Requirements
 - Subsystems must be able to be isolated from one another
 - Must be able to communicate between 2 subsystems without interference from a 3rd
- Components
 - Address spaces
 - Threads and IPC

- Unique IDs (not much more to say here, but you need to be able to identify address spaces, threads, and message).

Address Spaces

- All through inheritance
- One master address space (physical memory)
- All others are selections from this space
- Interface
 - Grant: move a page from your address space to a new one.
 - Map: share a page with another address space
 - Flush: remove a page from someone's address space

Threads

- Threads execute in an address space.
- Included in μ -kernel because a thread is associated with a particular address space (although the association may change over time); changes to address state must be managed by kernel.
- Thread state includes registers (minimally defined) and address space.
- Communication among threads (IPC) must also be a μ -kernel feature.
- IPC represents an agreement: sender sends and receiver agrees to receive (e.g., in a grant, the receiver of the page must agree to take the page).
- Interrupts are IPC messages with no payload; only purpose is to supply the sender ID so that the interrupt can be associated with a particular hardware device (each of which has a unique id that is used as the sender).
- Kernel must turn real hardware interrupts into thread events, but the rest of the kernel need never be aware of interrupts as they are truly manifested.

Implementing user-level services on a μ -kernel

- Memory manager/Pager
 - Grant, map, flush are the basic mechanisms.
 - Policies for how and when you issue these calls can be made in a user-level manager.
 - Each address space can have its own manager.
 - Can have application-specific managers (e.g., multimedia resource allocator) that must, itself, coordinate with other managers to make appropriate guarantees.
- Device Drivers
 - Can live outside the kernel, because they don't access hardware directly; they send/receive messages from the thread that represents the hardware.
- Cache and TLB handling
 - User pagers to implement whatever policies you like.
 - In practice, 1st level TLB handling still needs to be in the kernel for performance.
- Unix server
 - Simply a user-level server that creates and manages its own Unix address spaces.

Performance

- Context switches (e.g., system calls) shouldn't be that expensive.
 - Conventional systems pay almost 90% of their context switch time in "overhead."
 - L3 does not.
 - Why?
- Similarly, switching between address spaces shouldn't be so expensive.
 - Include thread and address space switching in the discussion (because that's what people measure).
 - If caches are physical, these don't affect context switch time.
 - If TLBs are untagged, an address space switch requires a flush of the TLB.
 - Use PowerPC hacks to get rid of TLB reload problem, by use of segment registers.
 - Tailor context switch code to the HW and figure out how to get it fast.
- Thread switches and IPC
 - Empirically show that L3 has fast IPC.

- Memory Effects
 - Chen and Bershad "showed" that microkernels had significantly worse memory behavior.
 - Liedtke shows that the difference is entirely in the cache miss behavior.
 - What kinds of misses are these (capacity or conflict)?
 - What would each imply?
 - conflict: might imply structure
 - capacity: code is just too big
 - Chen and Bershad show that self-interference causes more misses than user/kernel.
 - Ratio of conflict to capacity is much lower in Mach.
 - Liedtke concludes that the problem is simply too much code, and that most of that code is in Mach.

Portability

- μ -kernels should not be portable; they *are* the hardware compatibility layer.
- Example: implementation decision between 486 and Pentium is different if you are going for high performance. This difference suggests significant rewriting to port.
- Think of μ -kernel as microcode.