

Microkernels: L3

- based on the paper “On micro-kernel Construction” by Jochen Liedtke.

Micro-kernels

- Micro-kernels represent a well-studied OS structuring idea:
 - very small kernel (implementing little more than address spaces and threads)
 - most of the functionality outside the kernel
 - very fast communication between OS components (to compensate for distribution)

Advantages (software technology)

- more modular structure (but modules are common in other OSs—e.g., Linux)
- fault isolation: faults don't bring down other services, one service cannot access another's variables
- the system is more flexible: interfaces are not ad hoc. This encourages components that are drop-in replacements. Possibly different implementations can coexist

Micro-kernel concepts

- What should the kernel implement?
 - implies limited functionality guaranteed, also need for protection domain crossings
- The answer should depend on functionality— not performance
 - many of the traditional performance bottlenecks can be eliminated (e.g., Liedtke shows that user-to-user IPC can be done 22 times faster than in Mach—which is also a micro-kernel)
- Also – can cross-protection domain interactions be efficiently supported

- Objective: System must support interactive and not completely trustworthy applications
 - protection
 - principle of independence – S can make guarantees independent of S'
 - principle of integrity – S1 and S2 must be able to establish a channel that can be neither corrupted nor eavesdropped by S'
- The kernel should implement only three things:
 - address spaces
 - threads and IPC
 - unique identifiers

Address spaces

- Only a few concepts are fundamental:
 - an initial address space that represents the physical memory
 - operations *grant*, *map*, and *flush*
- **Grant**: the owner of address space A gives the page to another address space, B. The page is removed from A
 - the recipient has to agree
 - only used in special cases: when a server is an intermediary for others
- **Map**: like *grant*, but the owner of A shares the page with B
- **Flush**: when the owner of A flushes a page, the page is removed from all address spaces that received it from A through a (direct or indirect) grant or map
- I/O devices are supported by associating an address space
 - works for memory-mapped I/O and I/O ports

Examples of Address Space Use

- A regular memory manager is just a “process” (*server*) managing the initial address space
 - entirely outside the kernel!
- Memory managers can be stacked: one can grant parts of the physical memory to the next
- A pager can be a totally separate server, communicating through IPC
 - of course it could also be integrated in a memory manager
 - advantage: easy to implement specific policies (e.g., real time, flow-based, etc.)

Threads and IPC

- The kernel needs to have a concept of activities and their state of execution (i.e., registers, instruction pointer, stack pointer)
- Threads need to communicate but modifying a shared address space can lead to corruption
- The kernel can add more structure with kernel controlled IPC
 - at least some rudimentary IPC mechanism must exist and more complex ones can be built on top
 - IPC is safe because both parties agree (e.g., the server receives information and interprets it voluntarily)
 - no chance of corruption of unsuspecting servers
- Interrupts are just IPC
 - the HW is a thread sending IPC messages
 - the micro-kernel does the transformation

Unique Identifiers

- Either threads or communication channels need to be uniquely identified
- Alternatively viewed: the kernel implements communication—it also needs to implement naming

Examples of microkernel flexibility

- Device drivers won't be in the kernel. They are just servers that access I/O ports mapped to their address space
- RPC on top of a basic IPC mechanism can be implemented as an external server
- Unix system calls can be implemented as an external server
- everything but processor architecture, first-level TLB and cache

- Micro-kernel based OSs can offer extreme flexibility but they are criticized for bad performance (due to switching and memory effects)
- Paper argues that problem is not in the concept but in the implementation!

Switching Overhead

- Kernel-user switch (measurements on a 50MHz x86 machine)
 - cost of machine instructions for entering supervisor mode and back: 71 + 36 cycles
 - switch between kernel, user stacks
 - push/pop flag register, instruction pointer
 - but measured cost to enter the Mach kernel on this machine is 900 cycles!
 - Liedtke's L3 kernel does the same in 15 extra cycles if code is not in cache, it incurs 3 TLB misses, 10 cache misses, and argues that if better hardware support existed total cost can go down to ~20cycles

Address space switches

- if no flush is needed (tagged architectures or architectures with segment registers) the cost is smaller
- crucial idea: let's keep small, commonly used server address spaces shared with regular “large” address spaces
- for this to happen, address space assignment to a linear space must be dynamic and dependent on size

- IPC and thread switches
 - this can be made fairly fast for most purposes.
 - system call, argument copy, stack and address space switch...
 - for L3 expensive – 100 cycle trap overhead

Memory effects

- Liedtke acknowledges that some measurements have shown that Mach (a microkernel) together with a Unix server has much worse memory performance than a monolithic Unix kernel
- But then deconstructs the argument. Bottom line: the cache misses are capacity misses because Mach (esp. its IPC?) has a large cache working set
 - Many routines being used a lot.
 - A few routines using a lot of cache.
 - Neither should be the reason...

Non-portability

- Many micro-kernels have relied on a hardware abstraction layer but were themselves hardware-independent
- Liedtke's argument is that the micro-kernel should be processor-dependent
 - not just the coding, but even the algorithms inside a micro-kernel should be tuned to the architecture
 - a good example of the changes needed to port a micro-kernel from the 486 to a Pentium (“compatible” processors) is shown
- Understanding raw number differences
 - RPC comparison exokernel and L3