The Performance of μ-Kernel-Based Systems

Simon Bertron

September 17, 2024

Outline

- Microkernels
- Mach
- L4
- The Paper!

Outline

Microkernels

Mach

L4

The Paper!

Some Context

- 1981
- Michael Stonebraker publishes Operating System Support for Database Management
  - File system, scheduler, concurrency control all suboptimal for databases
  - Perhaps different applications require different OS primitives

Some Context

- 1989
- Unix alternatives are proliferating
  - Unix flavors endorsed by different standards bodies
  - But also non-Unix alternatives like Mac OS or MS-DOS

	Intel	ARM	PowerPC	...
BSD				...
Mac OS				...
MS-DOS				...
⋮	⋮	⋮	⋮	⋱

- Introducing a new OS or architecture required adding a full row or column to this matrix

What is a Microkernel?

A microkernel is an operating system which attempts to push traditional OS operations out of the "core" OS kernel. It ends up smaller than a traditional *monolithic* kernel as a result. Hence, it is "micro".

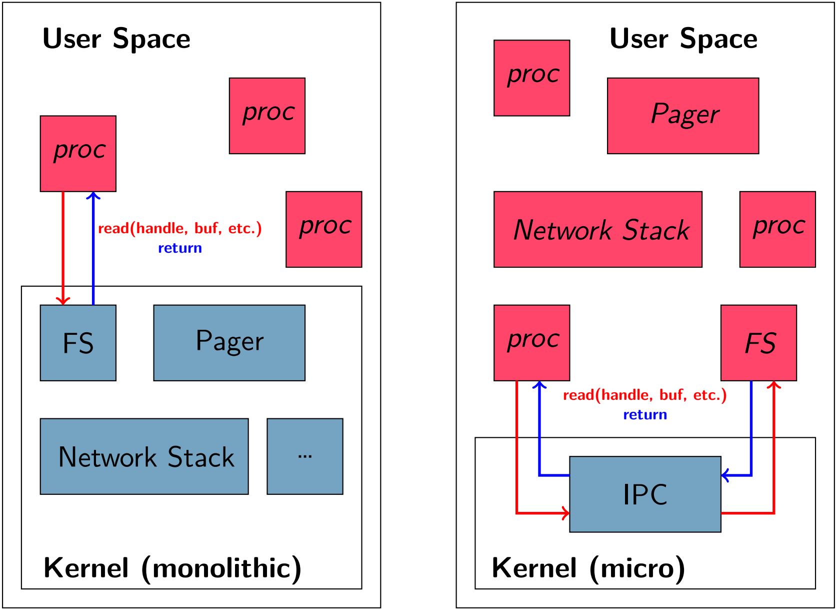
Why Microkernels?

1. Hardware Abstraction
2. Modularity
3. Security
4. Performance

Approaches to micro-ing the kernel

- Exokernel
  - Having any architecture-independent software abstraction layer at all is folly
- Spin
  - Make kernel modules much safer
  - Then everyone can load and unload modules to run the monolithic kernel that they need/want
- Mach/L4
  - Build an OS out of “servers” (basically kernel modules) in user space

Monolithic Kernels vs. Mach/L4 style microkernels



Discussion Questions

1. Thinking back to the end-to-end argument for system design, is the kernel the right place to put OS functionality? Is user space?
2. Do microkernels do anything for OS design that good software engineering practices wouldn't?
3. What security/isolation benefits do we achieve by putting kernel functionality into userspace processes? What threats remain?

Outline

What is a Mach?

- Mach provided threads, scheduling, architecture-independent virtual memory management, and IPC
- plus hooks for user processes to act as syscall handlers for other user processes
- and hooks for user processes act as pagers for other user processes

Mach design goals

1. **Hardware Abstraction**
   - ability to build/run other OSes on top of Mach, even simultaneously!
   - in some ways a proto-virtualization layer
2. **Modularity**
   - force more modularity in future OS designs by decoupling subsystems into separate user space "servers"
3. ~~Security~~
4. ~~Performance~~
5. Servicing page faults and syscalls over the network, for some reason?

How did Mach do?

- Mach seemingly ended up as mostly a substrate for existing monolithic kernels to run on top of
- Due to performance issues, Mach re-incorporated large chunks of OS functionality back into the kernel
- The Mach paper reports that BSD-on-Mach outperforms SunOS handsomely

Discussion Questions

1. We live in the future and we know that even after de-microing the kernel Mach performance was pretty abysmal. Why do you think it was so slow?

Outline

Microkernels

Mach

L4

The Paper!

Mach was terrible...

- Mach (and other gen-1 microkernels) performed so terribly that no one took the idea seriously
- High overheads prevented adoption as a substrate for monolithic kernels
- High overheads would negate any benefits from creative new systems built directly on Mach

...but L4 could be better

- IPC needs to be 1-2 orders of magnitude faster
- Address space switches need to be less costly

L4 design goals

1. Hardware Abstraction
2. Modularity
3. **Security**
4. **Performance**
   - L4 really wanted to prove that microkernels were *better* than monolithic kernels
5. Servicing page faults and syscalls over the network, again?

IPC

- Really was an order of magnitude faster than Mach in the first generation (and it pretty much only got faster)
- Which enabled using IPC to handle hardware interrupts and syscalls

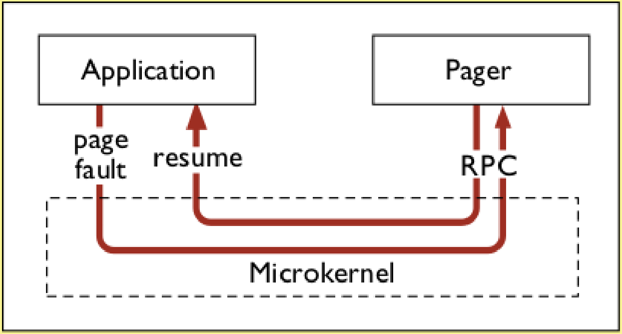


Figure 1. Page fault processing

Address Spaces

- L4 allowed recursive address space construction
- Any process could *grant* a page in its own address space to another consenting process
  - (this removed the page from the granter's address space)
- Any process could *map* a page in its own address space to another consenting process
  - (this kept the page in the granter's address space, creating a shared page)
- Processes could *unmap* any page in their own address space from all process who had inherited the page directly or indirectly from the unmapper
- In this way user space servers could perform almost all memory management functions

Address Spaces

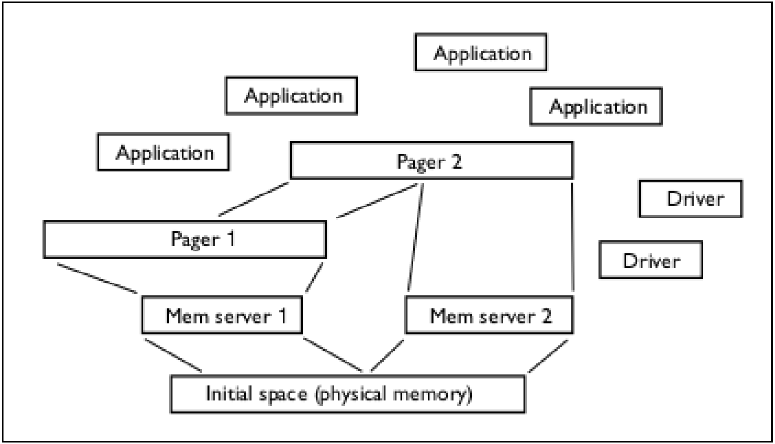


Figure 6. A maps page by IPC to B

Address Spaces

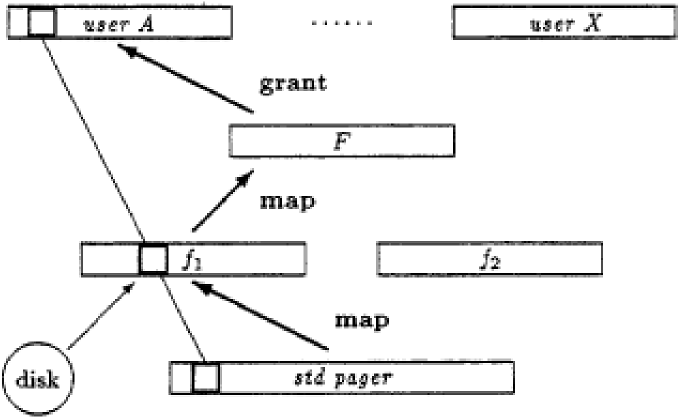


Figure 1: *A Granting Example.*

Discussion Questions

1. Do these abstractions seem flexible enough to support *any* operating system on top of them performantly?
   - Flexible enough to implement any system someone might care to actually build?

Outline

Microkernels

Mach

L4

The Paper!

Authors

Jochen Liedtke (1953-2001) invented the L4 microkernel and worked on several generations of microkernels, from L3 (a contemporary of Mach) to Hazelnut/Pistachio (a successor to L4).

Overview

- This paper is sort of like a giant evaluation section for the ongoing L4 project
- Is L4 performant enough to run Linux in userspace with low overhead?
- Is L4 expressive enough to build better-than-Linux OS structures?

L4 Linux Overhead

- The overhead is decent and massively improved over Mach

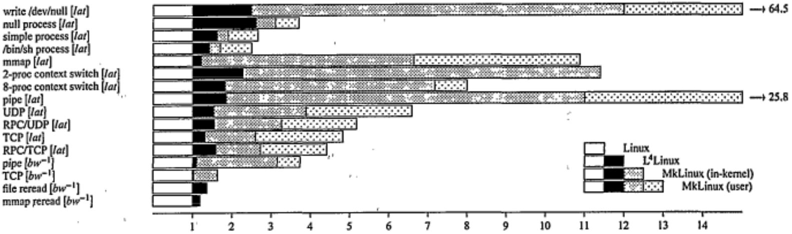


Figure 6: *lmbench results, normalized to native Linux.* These are presented as slowdowns: a shorter bar is a better result. [*lat*] is a latency measurement, [bw^{-1}] the inverse of a bandwidth one. Hardware is a 133 MHz Pentium.

L4 Linux Overhead

- The macrobenchmarks look better than the microbenchmarks

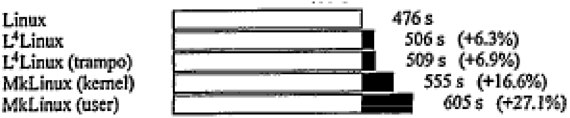


Figure 7: *Real time for compiling the Linux Server.* (133 MHz Pentium)

L4 Linux Overhead

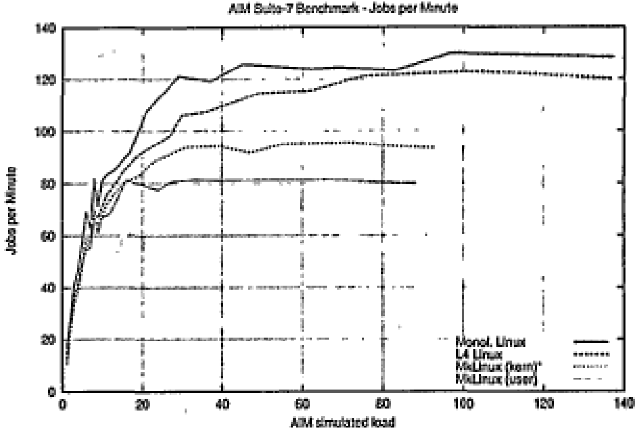


Figure 9: *AIM Multiuser Benchmark Suite VII.* Jobs completed per minute depending on AIM load units. (133 MHz Pentium)

Beating Linux

- Chose a simple test: build a good inter-process pipe
- L4 did great, but pipes are exactly what L4 is built to be really good at

System	Latency	Bandwidth
(1) Linux pipe	29 μs	41 MB/s
(1a) L^4Linux pipe	46 μs	40 MB/s
(1b) L^4Linux (trampoline) pipe	56 μs	38 MB/s
(1c) MkLinux (user) pipe	722 μs	10 MB/s
(1d) MkLinux (in-kernel) pipe	316 μs	13 MB/s
(2) L4 pipe	22 μs	48–70 MB/s
(3) synchronous L4 RPC	5 μs	65–105 MB/s
(4) synchronous mapping RPC	12 μs	2470–2900 MB/s

Table 4: *Pipe and RPC performance.* (133 MHz Pentium.) Only communication costs are measured, not the costs to generate or consume data.

Discussion Questions

1. What are some optimizations that could be built on top of L4 by specializing OS functionality for specific applications?
2. What is an example of a system that would benefit by co-locating a real time operating system with a timesharing operating system?