Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Informatik
Computer Science

Advanced Operating Systems (263-3800-00L)
Microkernels and L4

Timothy Roscoe & Andrew Baumann
Including material courtesy of Gernot Heiser, UNSW

Didn’t we already have a lecture on L4?

- This lecture focuses more on the background of the microkernel concept, and the origins of L4
  - Motivation for microkernel systems
  - Some history: Nucleus, Hydra
  - Mach
  - Problems with microkernel performance
  - L3 & L4: “second generation” microkernels
    - How do you make a microkernel fast?
  - Revisit of L4’s design principles

Outline

Microkernel background
- Motivation
- The Mach microkernel

Problems with microkernel performance
- Chen and Bershad's analysis
- Liedtke's critique

Origins of L4
- L3
- L4
- Design principles revisited

Conclusions

Motivation

- Early operating systems had very little structure
- A strictly layered approach was promoted by Dijkstra
  - THE Operating System
- Later systems mostly followed that approach (e.g., Unix)
- Such systems are known as monolithic kernels

Monolithic OS kernels

Advantages:

- Kernel has access to everything
  - all optimisations possible
  - all techniques/mechanisms/concepts implementable
- Kernel can be extended by adding more code
  - new services
  - support for new hardware

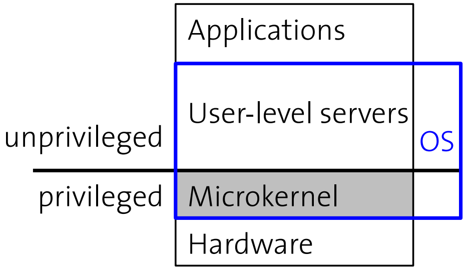
Problems:

- Widening range of services and applications
- OS bigger, more complex, slower, more error prone
- Need to support same OS on different hardware
- Want to support various OS environments

Approaches to tackling complexity

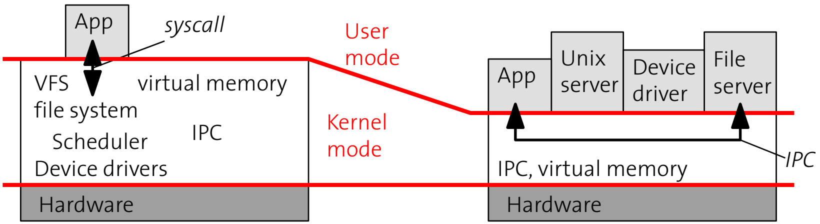
- Classical software engineering approach: modularity
  - Relatively small, self-contained components
  - Well-defined interfaces
  - Enforcement of interfaces
  - Containment of faults
- Doesn't work with monolithic kernels
  - All kernel code executes in privileged mode
  - Faults aren't contained
  - Interfaces cannot be enforced
  - Performance takes priority over structure

Microkernel



Based on ideas of the “Nucleus” [Brinch Hansen, 1970].

Monolithic vs. microkernel OS structure



Monolithic OS:

- lots of privileged code
- services invoked by syscall

Microkernel OS:

- little privileged code
- services invoked by IPC
- “horizontal” structure

Microkernel OS

Kernel:

- Contains code which must run in privileged mode
- Isolates hardware dependence from higher levels
- Small and fast extensible system
- Provides mechanisms

User-level servers:

- Are hardware independent/portable
- Provide the "OS environment/personality"
- May be invoked:
  - From application (via message-passing IPC)
  - From kernel (via upcalls)
- Implement policies

Early example: Hydra

- Separation of mechanism from policy [Levin et al., 1975]
- No hierarchical layering of kernel
- Protection, even within OS
    - Uses capabilities
- Objects, encapsulation, units of protection
- Can be considered the first object-oriented OS

Popular example: Mach

- Developed at CMU by Rashid and others from 1984 [Rashid et al., 1988]

Goals:

- Tailorability: support different OS interfaces
- Portability: almost all code H/W independent
- Real-time capability
- Multiprocessor and distribution support
- Security
- Coined term microkernel

Basic features of Mach kernel

- Task and thread management
- Inter-process communication
  - asynchronous message-passing
- Memory object management
- System call redirection
- Device support
- Multiprocessor support

Mach = µkernel?

- Most OS services implemented at user level
  - Using memory objects and external pagers
  - Provides mechanisms, not policies
- Mostly hardware independent
- Big!
  - 140 system calls (300 in later versions), >100 kLOC
    - Unix 6th edition had 48 system calls, 10kLOC without drivers
  - 200 KiB text size (350 KiB in later versions)
- Poor performance
  - Tendency to move features into kernel

Outline

Critique of microkernel architectures

“Personally, I’m _not_ interested in making device drivers look like user-level. They aren’t, they shouldn’t be, and microkernels are just stupid.”

Linus Torvalds

… Is he right?

Microkernel performance

- First generation microkernel systems ('80s, early '90s)
  - Exhibited poor performance when compared to monolithic UNIX implementations
  - Particularly Mach, the best-known example
- Typical results:
  - Move OS services back into the kernel for performance
  - Move complete OS personalities into kernel
    - Mach Unix server $\rightarrow$ Unix kernel co-located with Mach
    - Chorus Unix
    - Mac OS X (Darwin): complete BSD kernel linked to Mach
    - OSF/1
- Some spectacular failures
  - IBM Workplace OS
  - GNU Hurd

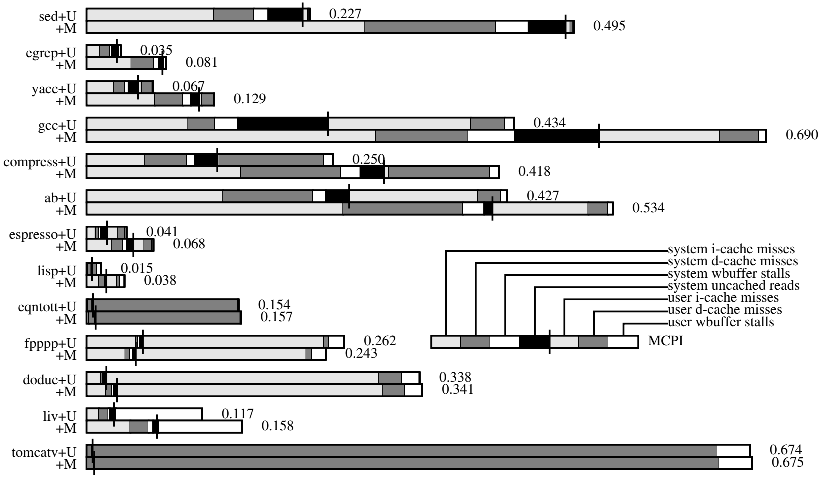
Microkernel performance

- Reasons investigated [Chen and Bershad, 1993]
  - Instrumented user & system code to collect execution traces
  - Run on DECstation 5000/200 (25MHz MIPS R3000)
  - Run under Ultrix and Mach with Unix server
  - Traces fed to memory system simulator
  - Analysed memory cycles per instruction:

$$\text{MCPI} = \frac{\text{stall cycles due to memory system}}{\text{instructions retired}}$$

    - Baseline MCPI (i.e. excluding idle loops)

Ultrix vs. Mach+Unix MCPI



[Chen and Bershad, 1993]

Interpretation

Observations:

- Mach memory penalty higher
  - i.e. cache misses or write stalls
- Mach VM system executes more instructions than Ultrix
  - but is portable and has more functionality

Claim:

- Degraded performance is result of OS structure
- IPC cost is not a major factor:

"…the overhead of Mach's IPC, in terms of instructions executed, is responsible for a small portion of overall system overhead. This suggests that microkernel optimizations focusing exclusively on IPC, without considering other sources of system overhead such as MCPI, will have a limited impact on overall system performance."

Conclusions

- System instruction and data locality is measurably worse than user code
  - Higher cache and TLB miss rates
  - Mach worse than Ultrix
- System execution is more dependent than user on instruction cache behaviour
  - MCPI dominated by system Icache misses
- Competition between user and system code not a problem
  - Few conflicts between user and system cache

  “The impact of Mach’s microkernel structure on competition is not significant.”

Conclusions

- Self-interference, especially on instructions, is a problem for system code
  - Ultrix would benefit more from higher cache associativity (direct-mapped cache was used)
- Block memory operations are responsible for a large component of overall MCPI
  - IO and copying
- Write buffers less effective for system
- Page mapping strategy has significant effect on cache

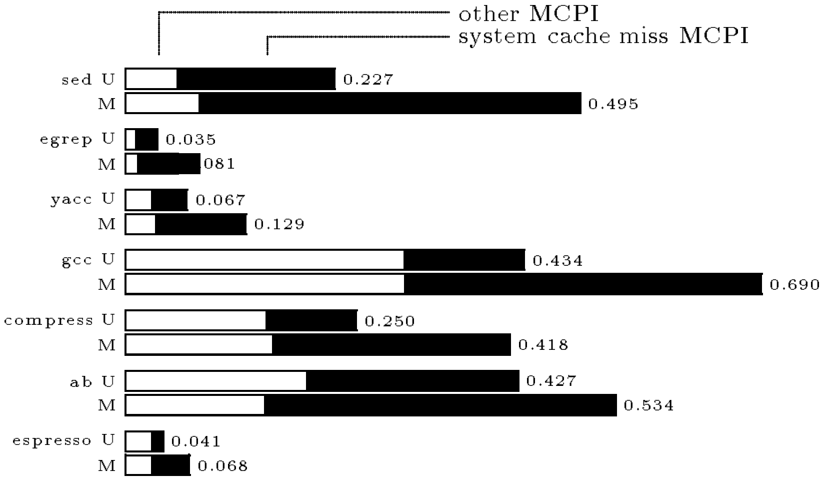
“The locality of system code and data is inherently poor”

Other experience with μkernel performance

- System call costs are high
- Context switching costs are high
  - Getting worse with increasing CPU/memory speed ratios and lengthening pipelines
- IPC (system call + context switch) is therefore expensive
- Microkernels depend heavily on IPC
  - Is the microkernel idea inherently flawed?

A Critique of the critique

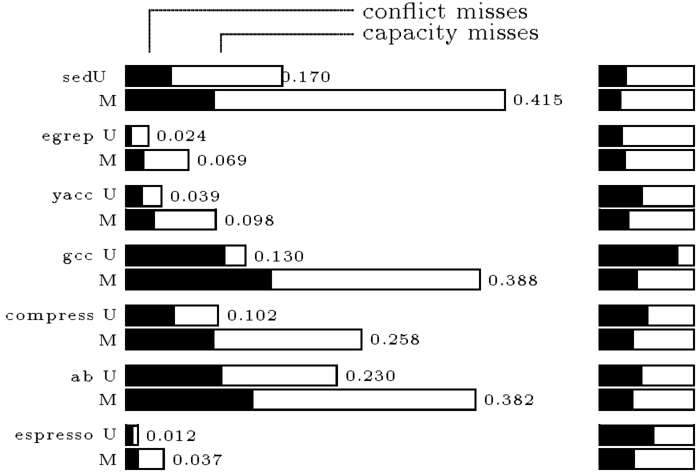
MCPI for Ultrix and Mach



[Liedtke, 1995]

A Critique of the critique

MCPI caused by cache misses



[Liedtke, 1995]

Conclusion

- Mach system is too big
  - Kernel + Unix server + emulation library
- Unix server is essentially the same
- Emulation library irrelevant [Chen and Bershad, 1993]
- Conclusion: Mach kernel working set is too big

Conclusion

- Mach system is too big
  - Kernel + Unix server + emulation library
- Unix server is essentially the same
- Emulation library irrelevant [Chen and Bershad, 1993]
- Conclusion: Mach kernel working set is too big

Can we build microkernels which avoid these problems?

Outline

Improving IPC by kernel design [Liedtke, 1993]

- IPC is the most important operation in a microkernel
- The way to make IPC fast is to design the whole system around it
- Design principle: aim at a concrete performance goal
  - Hardware-dictated costs are 172 cycles (3.5µs) for a 486
  - Aimed at 350 cycles for the implementation
- Applied to the L3 kernel

L3/L4 implementation techniques

- Minimise number of system calls
  - Combined operations: Call, ReplyWait
  - Complex messages
    - Combines multiple messages into one operation
    - As many arguments as possible in registers
- Copy messages only once
  (via direct mapping, not user→kernel→user)
- Fast access to thread control blocks (TCBs)
  - TCBs accessed via VM address determined from thread ID
  - Invalid threads caught via a page fault
  - Separate kernel stack for each thread in TCB
  - Avoids extra TLB misses on fast path

L3/L4 implementation techniques

- Lazy scheduling
  - Don't update scheduling queues until you need to schedule
- Direct process switch to receiver
- Short messages in registers
- Reducing cache and TLB misses
  - Frequently-used TCB data near the beginning (single-byte displacement)
  - Frequently-used TCB data co-located (for cache locality)
  - IPC code and kernel tables in a single page (to reduce TLB pressure and refills)
- Use x86 alias registers (ax = al,ah) to pack arguments
- Avoid jumps and checks on fast path
- and more…

Results (L3)

- A short cross address space IPC (user to user) takes 5.2μs
  - compared to 115μs for Mach
- Code and data together use 592 bytes (7%) of on-chip cache
  - kernel must be small to be fast

On µ-Kernel Construction [Liedtke, 1995]

What primitives should a microkernel implement?

"…a concept is tolerated inside the µ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality."

- Recursively-constructed address spaces
  - Required for protection
- Threads
  - As execution abstraction
- IPC
  - For communication between threads
- Unique identifiers
  - For addressing threads in IPC

What should a microkernel not provide?

- Memory management
- Page-fault handler
- File system
- Device drivers
- ...

Rationale: few features $\rightarrow$ small size $\rightarrow$ low cache use $\rightarrow$ fast

Non-portability

- Liedtke argues that microkernels must be constructed per-processor and are inherently unportable
- Eg. major changes made between 486 and Pentium:
  - Use of segment registers for small address spaces
  - Different TCB layout due to different cache associativity
    - Changes user-visible bit structure of thread identifiers!

Microkernel vs. Exokernel

Do abstractions belong in the kernel?

"Dropping the abstractional approach could only be justified by substantial performance gains. Whether these can be achieved remains open until we have well-engineered exo- and abstractional μ-kernels on the same hardware platform. It might then turn out that the right abstractions are even more efficient than securely multiplexing hardware primitives or, on the other hand, that abstractions are too inflexible."

. . . more on this later in the course!

Liedtke's design principles: what stands?

. . . in modern versions of L4

- Minimality: yes
- Abstractions: yes
  - but no agreement about some of them (e.g. IPC model, threading, security)
  - L4 API continues to evolve and diverge
- Unportable: no
  - Pistachio: C++ with highly tuned assembler IPC fast-path

Outline

Conclusions

- L4 results suggest that efficient microkernels are achievable
  - L4 + L4Linux close to monolithic kernel performance
- Need a real multi-server system to evaluate microkernel idea
  - Attempts: SawMill, DROPS, Mungi, L4/Hurd, MINIX3, . . .
  - Real success only in restricted domains (e.g. embedded)
  - *Is the microkernel approach to blame?*
- OS structure hard to evaluate purely on quantitative grounds

Recent Research

seL4: Formal verification of an OS kernel
Gerwin Klein et al, SOSP 2009

- World's first formally-verified OS kernel
- Machine-checkable proof (Isabelle HOL) of security and correctness properties

NOVA: A microhypervisor-based secure virtualization architecture
Udo Steinberg and Bernhard Kauer, EuroSys 2010

- Combines ideas of microkernels and VMs, reduces VMM TCB

Trust and protection in the Illinois browser operating system
Shuo Tang et al, OSDI 2010

- Specialised OS, for reducing TCB of web browser
- Based on L4Ka::Pistachio

References

Brinch Hansen, P. (1970).
The nucleus of a multiprogramming operating system.
CACM, 13:238–250.

Chen, J. B. and Bershad, B. N. (1993).
The impact of operating system structure on memory system performance.
In *14th SOSP*, pages 120–133.

Levin, R., Cohen, E., Corwin, W., Pollack, F., and Wulf, W. (1975).
Policy/mechanism separation in HYDRA.
In *SOSP*, pages 132–140.

Liedtke, J. (1993).
Improving IPC by kernel design.
In *14th SOSP*, pages 175–188.

Liedtke, J. (1995).
On μ-kernel construction.
In *15th SOSP*, pages 237–250.

Rashid, R., Tevanian, Jr., A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W. J., and Chew, J. (1988).
Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures.
Trans. Computers, C-37:896–908.