Microkernels: From Mach to seL4 (Lecture 18, cs262a)

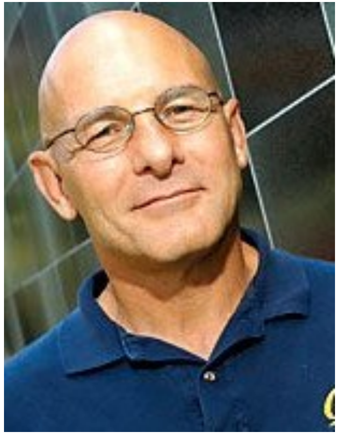
Ali Godsi & Ion Stoica,
UC Berkeley
March 21, 2018

Papers

"Microkernel Operating System Architecure and Mach", D. Black, D. Golub, D. Julin, R. Rashid, R. Draves, R. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman (https://amplab.github.io/cs262a-fall2016/notes/Mach.pdf)

"seL4: Formal Verification of an OS Kernel", Gerwin Klein , Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, (https://www.sigops.org/sosp/sosp09/papers/klein-sosp09.pdf)

David Patterson – Turing Award winner (2017)



For RISC (Reduce Instruction Set Computer) project

- SPARC processor from Sun (now Oracle)
- ARM (Acorn Risc Machine)
- Every processor today has a RISC architecture at its core

Lots of other impactful work: RAID, Recovery Oriented Computing, etc.

Key Observation (~1985)

Mondern OSes at that time (e.g., Unix, OS/2) primarily distinguished by the programming environment they provide and not by the way they manage resources

Opportunity:

- Factor out the common part
- Make it easier to build new OSes

Microkernels separates OS in two parts

Part of OS that control basic hardware resources (i.e.. microkernel)

Part of OS that determine unique characteristics of application environment (e.g., file system)

What problem do they try to solve?

Portability:

- Environment mostly independent on the instruction set architecture

Extensibility & customization:

- Can easily add new versions of environments
- Enable environments to evolve faster (decouples them from microkernel)
- Can simultaneously provide environments emulating interfaces

Sounds familiar?

- Microkernel as a narrow waist (anchor point) of OSes
- Provide hardware independence, similar to data independence in relational data models

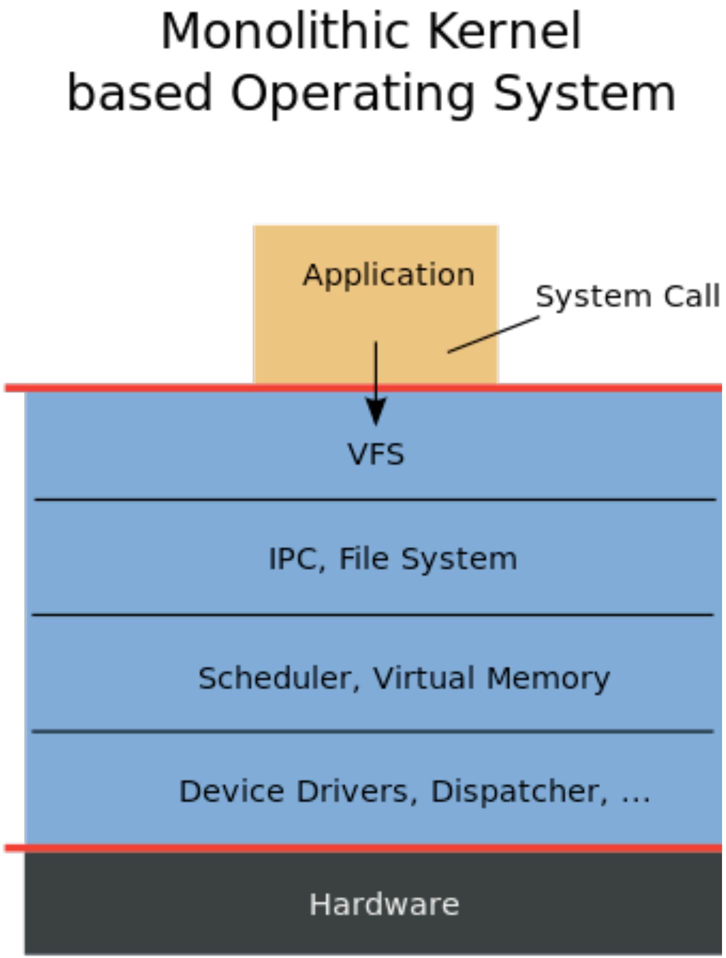
What problem do they try to solve?

Easier to provide better functionality and performance for kernel:

- Real-time: no need to maintain lock for extended periods of time; environments are preemptable
- Multiprocessor support: simpler functionality → easier to parallelize
- Multicomputer support: simpler functionality → easier to distribute
- Security: simpler functionality → easier to secure

Flexibility (network accessibility):

- System environment can run remotely



(https://en.wikipedia.org/wiki/Microkernel)

Mach

Goal: show that microkernels can be as efficient as monolithic operating systems:

- "… achieving the levels of functionality and performance expected and required of commercial products"

Sounds familiar?

- Similar goals as System R and Ingress: Show that a conceptually superior solution (i.e., relational model) admit efficient implementations that can match the performance of existing solutions (i.e., network and hierarchical models)

Mach

Developed at CMU

Led by Rick Rashid

- Founded Microsoft Research

Initial release: 1985

Big impact (as we will see)

Rick Rashid

What does a microkernel (Mach) do?

Task and thread management:

- Task (process) unit of allocation
- Thread, unit of execution
- Implements CPU scheduling: exposed to apps
  - Applications/environments can implement their own scheduling policies

Inter-process communication (IPC)

- Between threads via ports
- Secured by capabilities

What does a microkernel (Mach) do?

Memory object management:

- Essentially virtual memory
- Persistent store accessed via IPC

System call redirection:

- Enable to trap system calls and transfer control to user mode
- Essentially enable applications to modify/extend the behavior and functionality of system calls, e.g.,
  - Enable binary emulation of environments, tracing, debugging

What else does a microkernel (Mach) do?

Device support:

- Implemented using IPC (devices are contacted via ports)
- Support both synchronous and asynchronous devices

User multiprocessing:

- Essentially a user level thread package, with wait()/signal() primitives
- One or more user threads can map to same kernel thread

Multicomputer support:

- Can map transparently tasks/resources on different nodes in a cluster

Mach 2.5

Contains BSD code compatibility code, e.g., one-to-one mapping between tasks and processes

Some commercial success:

- NeXT
  - Steve Jobs' company after he left Apple
  - Used by Tim Berners-Lee to develop WWW
- Encore, OSF (Open Software Foundation), …

Mach 3

Eliminate BSD code

Rewrite IPC to improve performance

- RPC on (then) contemporary workstations: 95 usec

Expose device interface

Provide more control to user applications via continuation:

- Address of an user function to be called when thread is rescheduled plus some data: essentially a callback
- Enable application to save restore state, so that the microkernel doesn't need to do it, e.g., saving and restoring register state

OSes and Application Programs

Mach allows application to implement:

- Paging
- Control data cached by virtual memory
- …

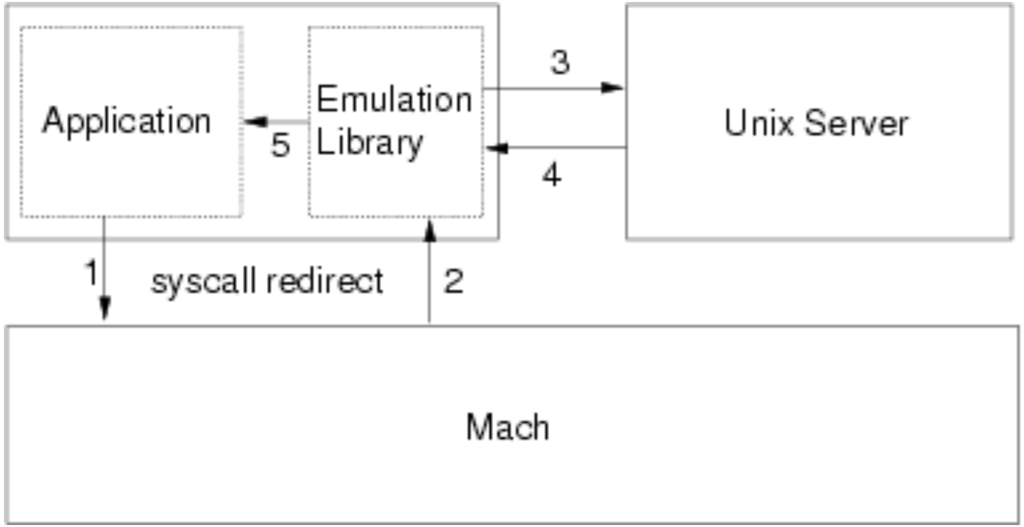
Redirection allows call traps to link directly to executable binaries without modifying he kernel!

- Just need an emulation library

Emulation Libraries

Translator for system services and a cache for their results

- Converts app calls to Mach calls
- Invoke functionality of the environment (e.g., OS) and reply to app
- Typically linked to app to avoid another context switching



OSes Environment Architectures

Fully implemented in the emulation library

- Simple, single user systems (e.g., MS-DoS)

As a server (see previous slide)

Native OSes: use the code of the original systems

- Used to implement both MacOS, and DOS
- Emulation library also virtualizes the physical resources

Performance: Mach 2.5 vs 3.0

Virtually the same as Mach 2.5, and commercial Unix systems of that time

- SunOS 4.1 and Ultrix 4.1

Why?

- I/O dominated tasks (read, write, compile)

Microbenchmarks would have been nice, e.g.:

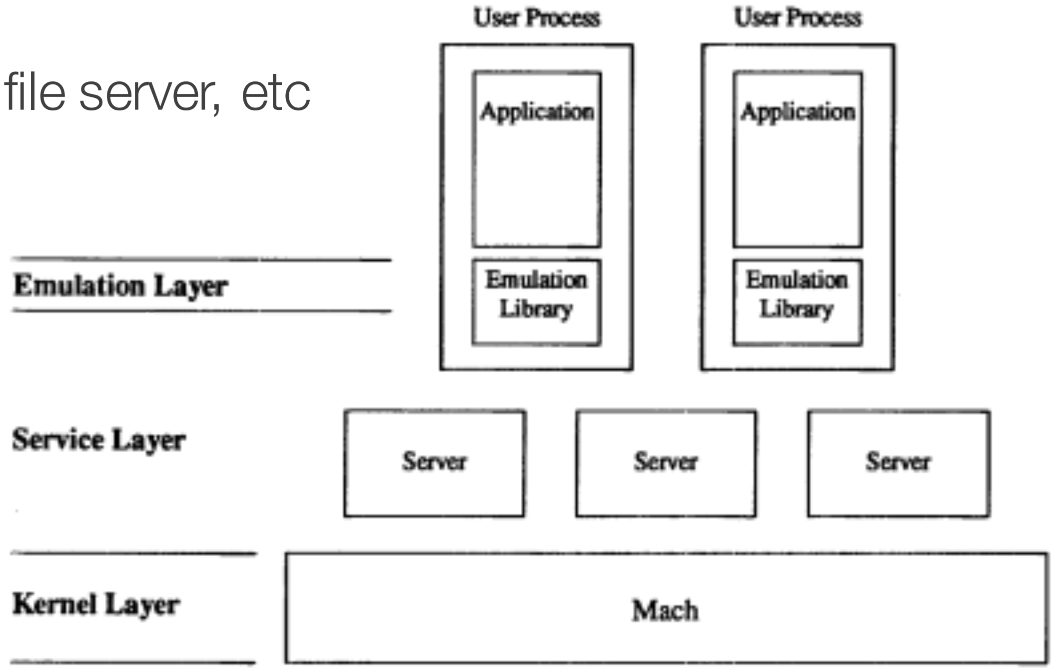
- IPC
- Cost of a page fault

OSF/1 Unix Server

Even more modularity: different OS functionalities implemented as different servers, e.g.,

  - IPC, process management, file server, etc

Server proxies on client side for optimization



L3 → seL4

How it started? (1993)

Microkernels (e.g., Mach) still too slow

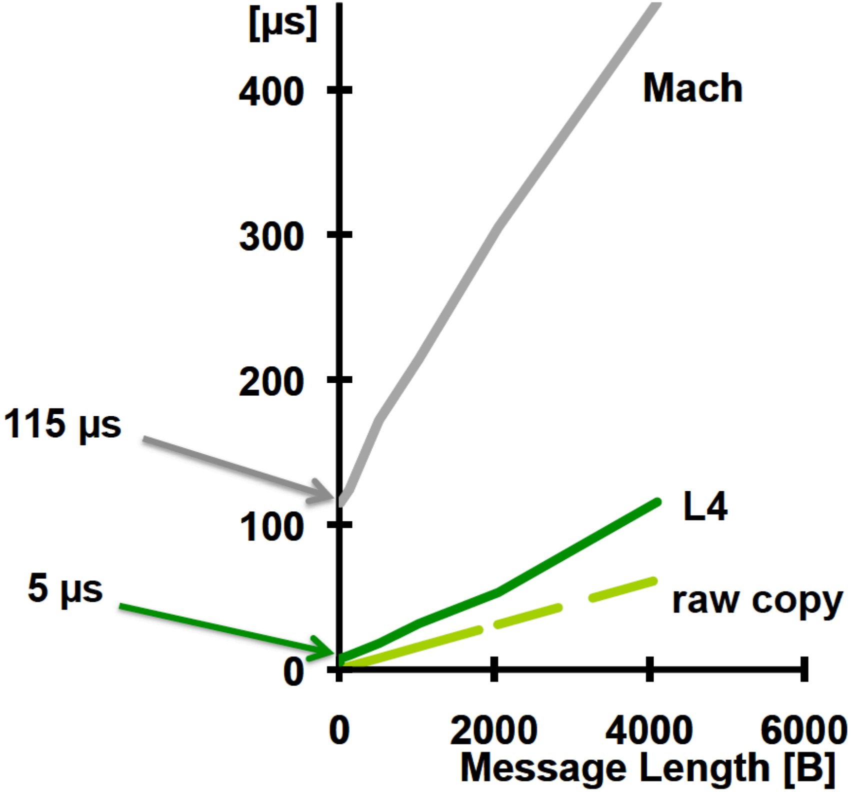
- Mostly because IPCs

Tide was turning towards monolithic kernels

Jochen Liedtke (GMD – Society for Mathematics and Information technology) aimed to show that IPC can be supper-fast

Jochen Liedtke

How fast?



115 µs

5 µs

**Culprit:
Cache
footprint
[SOSP’95]**

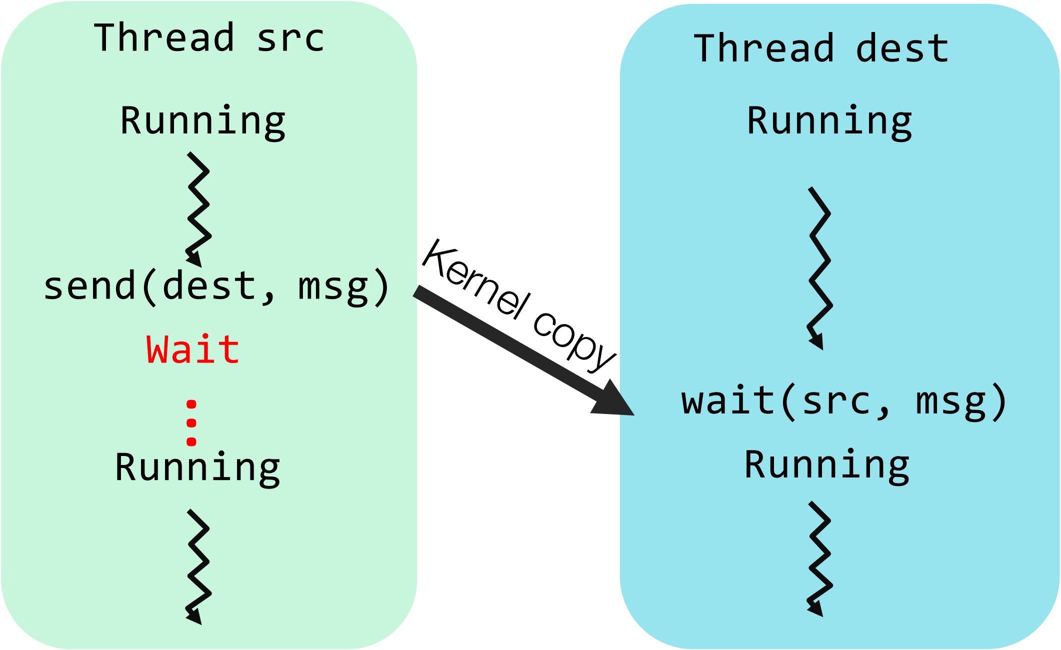
**i486 @
50 MHz**

How did he do it?

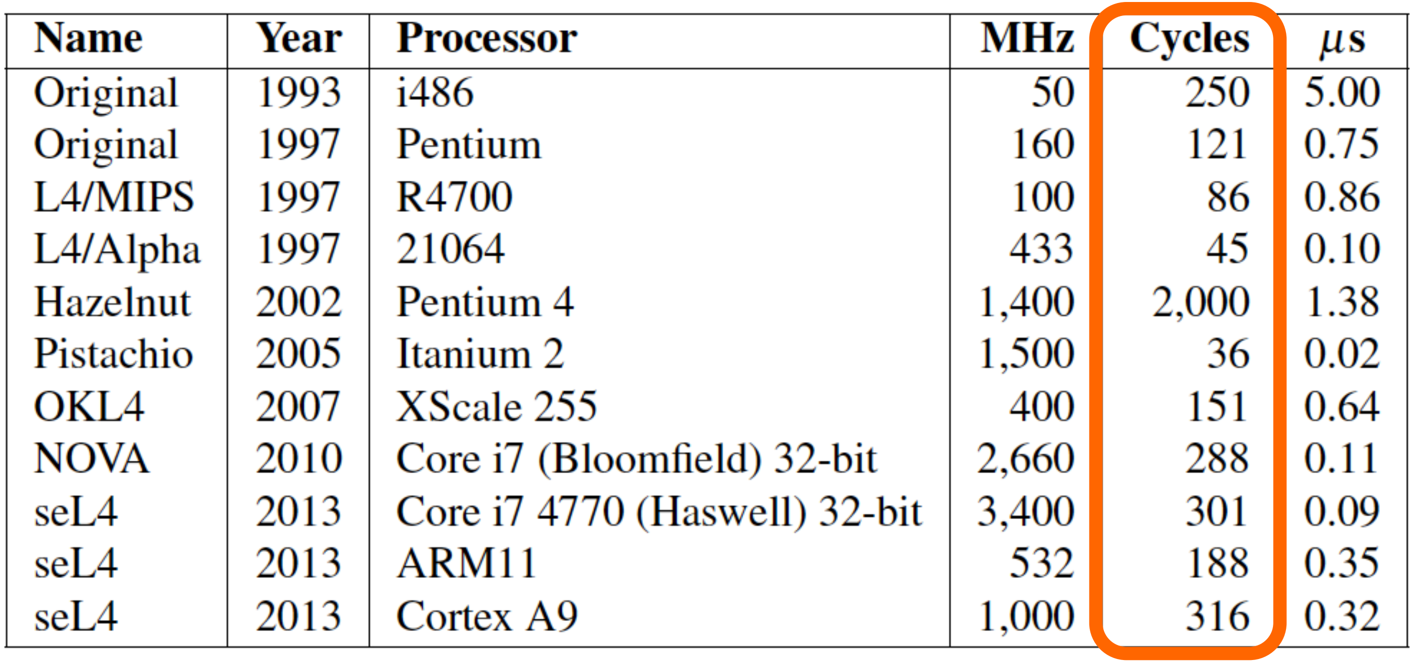
Synchronous IPC → Rendezvous model

Kernel executes in sender's context

- copies memory data directly to receiver (**single-copy**)
- leaves message registers unchanged during context switch (**zero copy**)



One-way IPC cost over years



Name	Year	Processor	MHz	Cycles	μs
Original	1993	i486	50	250	5.00
Original	1997	Pentium	160	121	0.75
L4/MIPS	1997	R4700	100	86	0.86
L4/Alpha	1997	21064	433	45	0.10
Hazelnut	2002	Pentium 4	1,400	2,000	1.38
Pistachio	2005	Itanium 2	1,500	36	0.02
OKL4	2007	XScale 255	400	151	0.64
NOVA	2010	Core i7 (Bloomfield) 32-bit	2,660	288	0.11
seL4	2013	Core i7 4770 (Haswell) 32-bit	3,400	301	0.09
seL4	2013	ARM11	532	188	0.35
seL4	2013	Cortex A9	1,000	316	0.32

Minimalist design

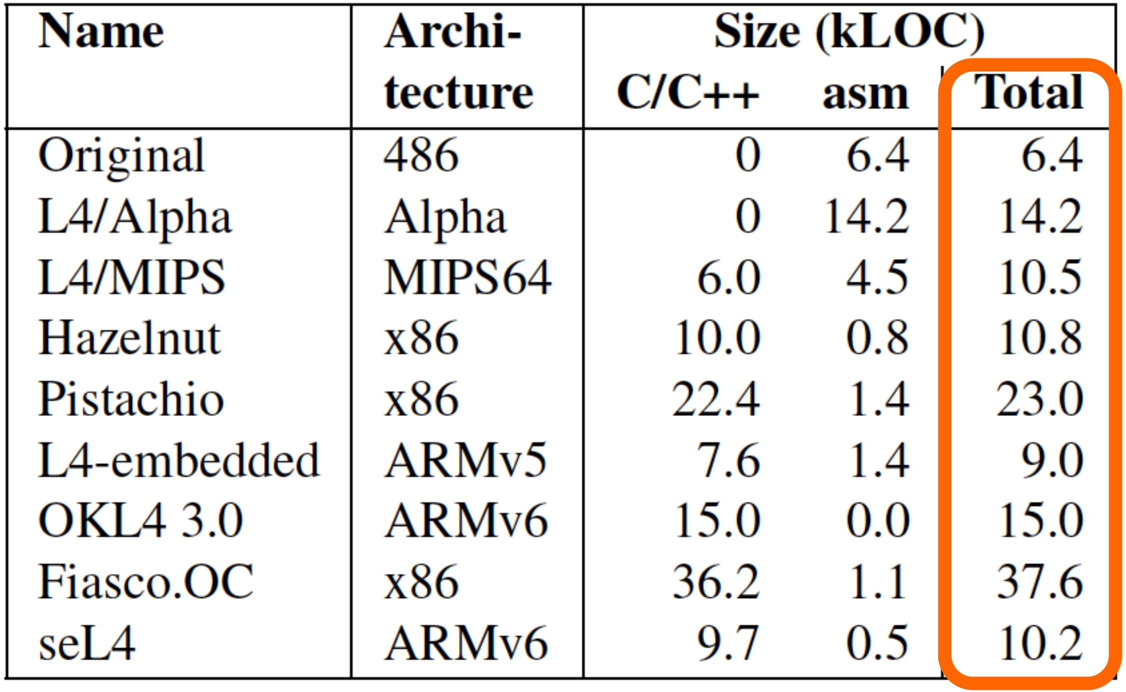
“A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system’s required functionality”

Sounds familiar?

“Don’t implement anything in the network that can be implemented correctly by the hosts”

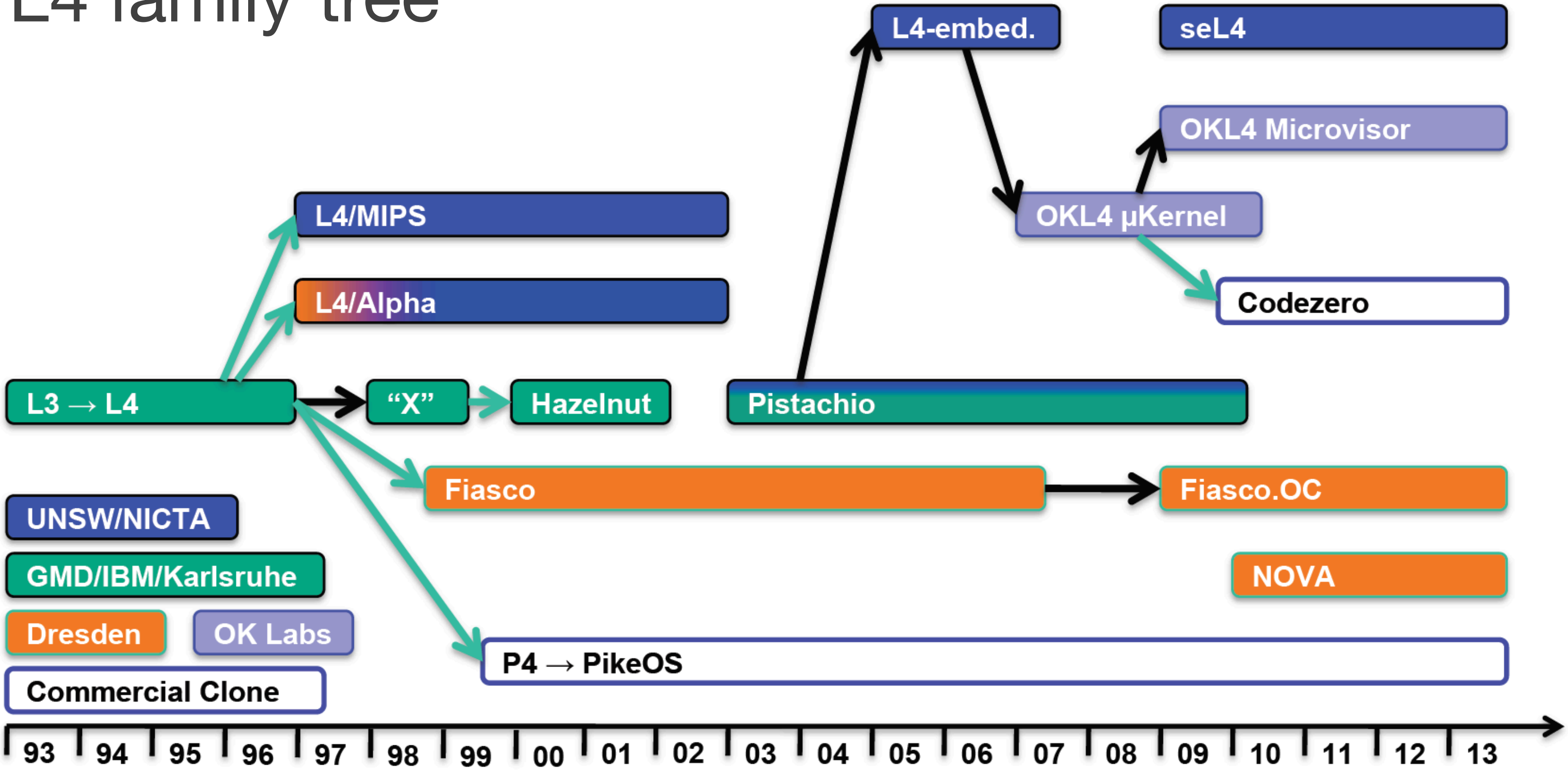
-- radical interpretation of the e2e argument!

Source Lines of Code

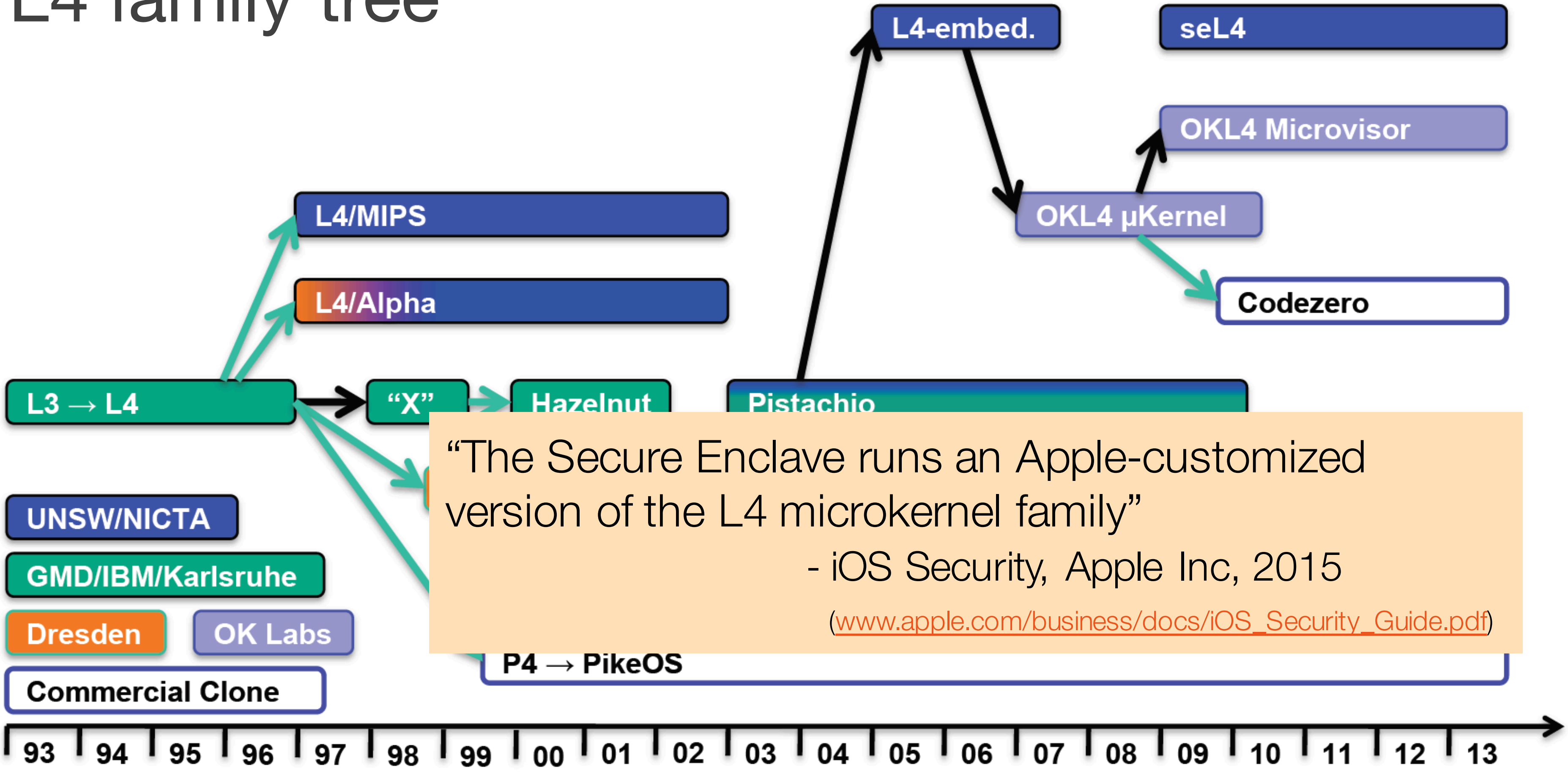


Name	**Architecture**	**Size (kLOC)**		
		C/C++	**asm**	**Total**
Original	486	0	6.4	6.4
L4/Alpha	Alpha	0	14.2	14.2
L4/MIPS	MIPS64	6.0	4.5	10.5
Hazelnut	x86	10.0	0.8	10.8
Pistachio	x86	22.4	1.4	23.0
L4-embedded	ARMv5	7.6	1.4	9.0
OKL4 3.0	ARMv6	15.0	0.0	15.0
Fiasco.OC	x86	36.2	1.1	37.6
seL4	ARMv6	9.7	0.5	10.2

L4 family tree



L4 family tree



> "The Secure Enclave runs an Apple-customized version of the L4 microkernel family"
>
> \- iOS Security, Apple Inc, 2015
>
> (www.apple.com/business/docs/iOS_Security_Guide.pdf)

Secure L4 (seL4) – Design Goals

Create a formal model of a microkernel

Implement the microkernel

Prove that it always behaves according to the specification

Assumptions

Hardware works correctly

Compiler produces machine code that fits their formalization

Some unchecked assembly code is correct

Boot loader is correct

How to design kernel + spec?

Bottom-Up-Approach: Concentrate on low-level details to maximize performance

Problem: Produces complex design, hard to verify

Reminder

Not all equivalent programs are equally amenable to verification

```
void swap(ptr A, ptr B)
{
        ptr C := A;
        A := B;
        B := C;
}
```

vs.

```
void swap(ptr A, ptr B)
{
        A := A xor B;
        B := A xor B;
        A := A xor B;
}
```

Postcondition: $A_{post} = B_{pre} \wedge B_{post} = A_{pre}$

How to design kernel + spec?

Top-Down-Approach: Create formal model of kernel and generate code from it

Problem: High level of abstraction from hardware

How to design kernel + spec?

Compromise: build prototype in high-level language (Haskell)

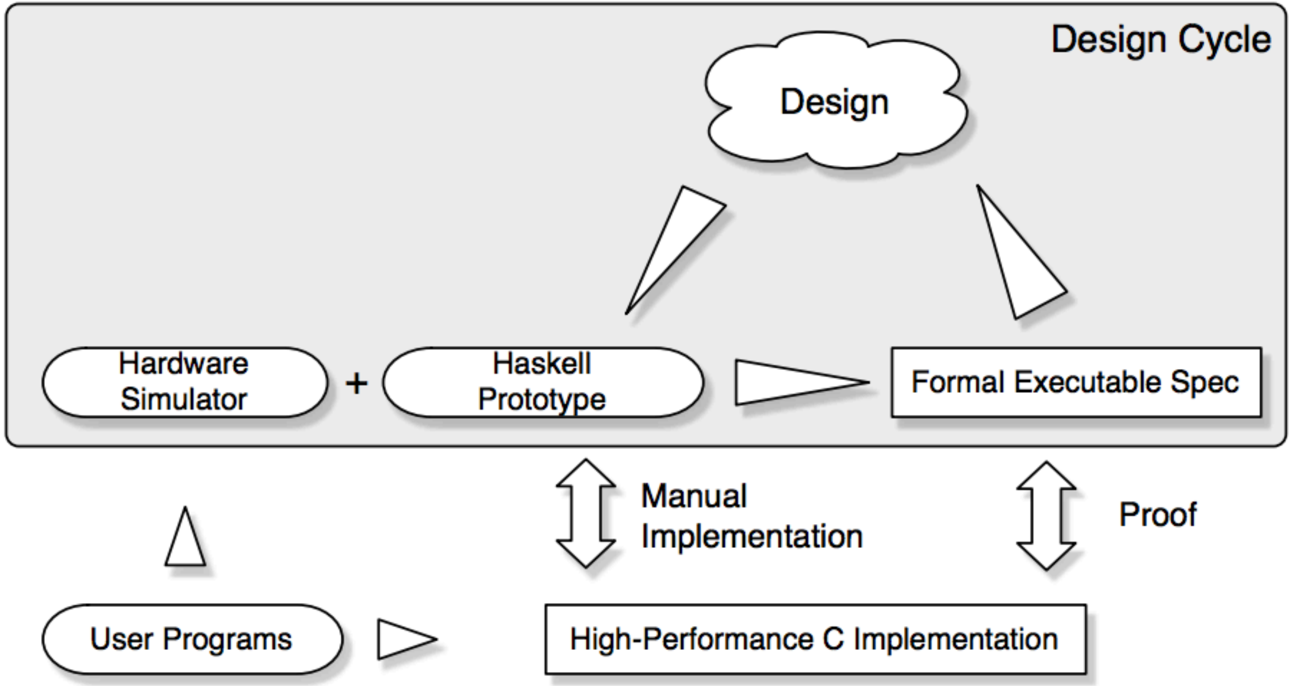
Generate “executable specification” from prototype

Re-implement executable specification in C

Prove refinements:

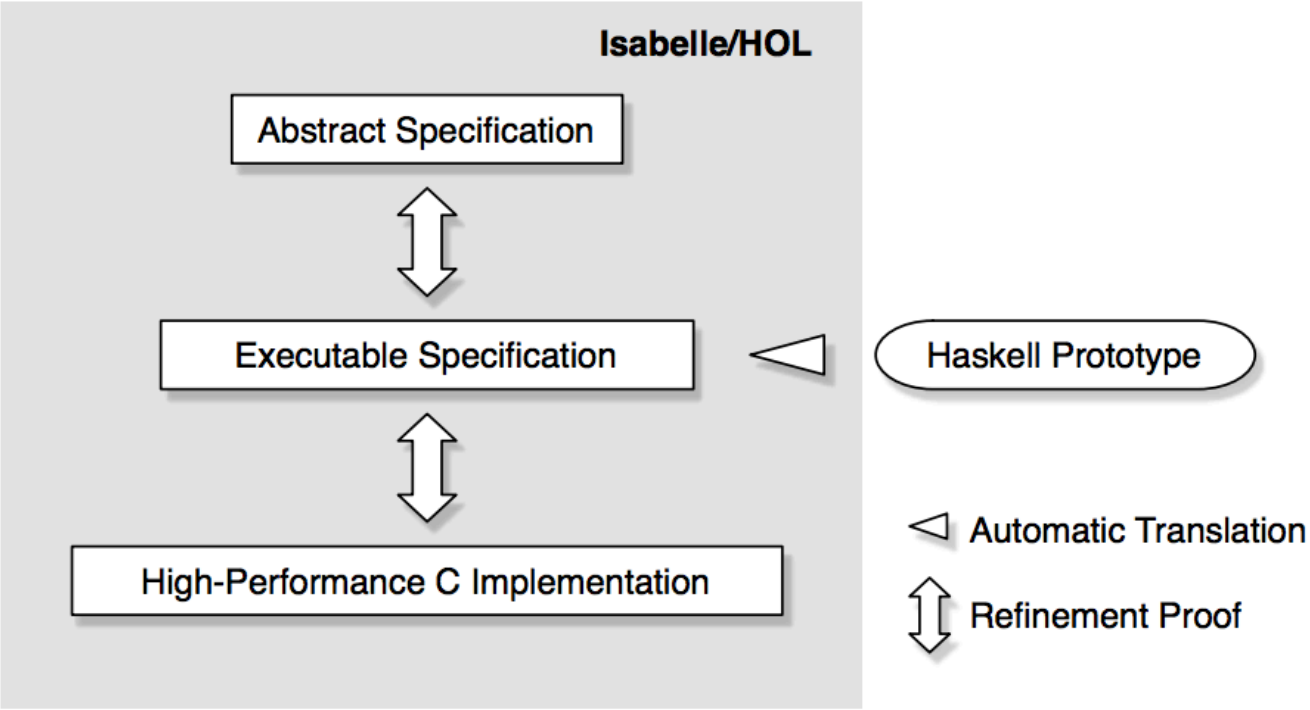
- C ⇔ executable specification
- Executable specification ⇔ Abstract specification (more high-level)

seL4 design process



Source: seL4, Klein et al.

seL4 verification



Source: seL4, Klein et al.

Concurrency is a problem

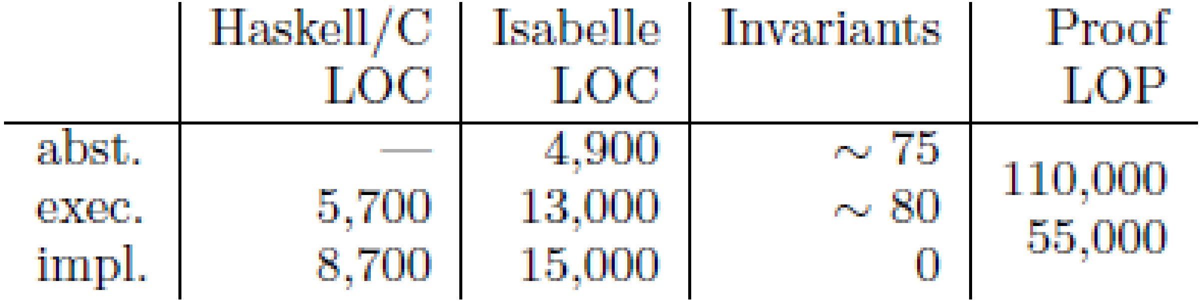
Multiprocessors not included in the model

- seL4 can only run on a single processor

Interrupts are still there

- Yield points need to establish all system invariants

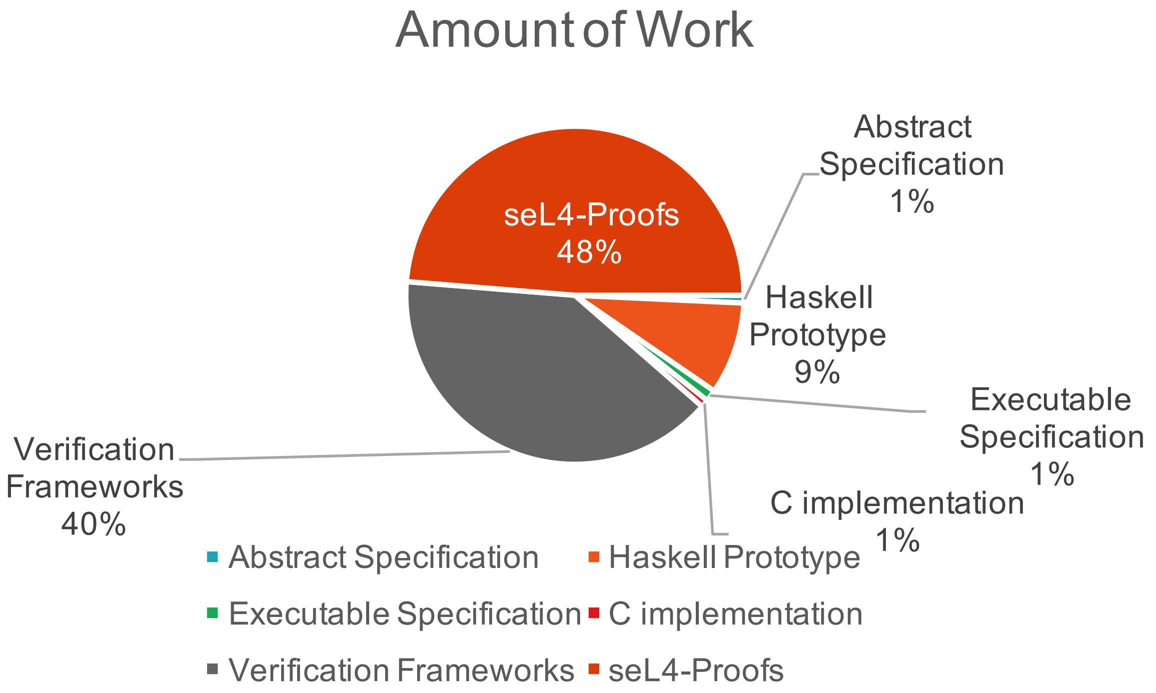
Cost of Verification



	Haskell/C LOC	Isabelle LOC	Invariants	Proof LOP
abst.	—	4,900	~ 75	110,000
exec.	5,700	13,000	~ 80	55,000
impl.	8,700	15,000	0	

Source: seL4, Klein et al.

Cost of Verification



Source of Data: seL4, Klein et al.

Takeaway

Functional verification of microkernels is possible

Performance of verified kernels can be OK

However:

- Verification is a huge effort
- Still needs to assume compiler correctness (➔ huge trusted base)

Is proving functional correctness worth the effort?

What drove L4’s evolution?

Application domain: embedded devices (natural fit!)

- Small footprint
- Devices ran few applications, didn’t need all OS services (e.g., file system)

Embedded devices required:

- Security and resilience → special attention to DoS attacks, formal verification
- Real-time guarantees → non-preemptable kernel

Did microkernels take over the world?

Pretty much…

- MacOS, based on NeXT, based on Mach
- iOS has both bits of Mach and L4
- Windows: hybrid (similar design goals to Mach)

With one notable exception, **Linux!**

So why didn't take over *entire* world!

Hardware standardization:

- Intel and ARM dominating
- Less need for portability, one of main goals of Mach

Software standardization:

- Windows, MacOS/iOS, Linux/Android
- Less need to factor out common functionality

Maybe just a fluke?

- Linux could have been very well adopted the microkernel approach
- Philosophical debate between Linus and Andy Tanembaum
  - One of Linus main arguments: there is only i386 I need to write code for! (http://www.oreilly.com/openbook/opensources/book/appa.html)

What drove L4’s evolution? (cont’d)

User experience, e.g.,

- New features, e.g., async IPC
- Remove features not useful: timeouts, clans & chiefs

Software evolution:

- E.g., Linux raise and POSIX decline obviate the need for long IPCs

Hardware advances

- Bigger caches, bigger TLBs, better context switching support → obviate the need for some optimizations (e.g., virtual TLBs. Thread IDs as destination IDs)
- Multicores → push for some optimizations (async wait)

Long IPCs: Transferring large messages

What happens during page faults?

IPC page faults are nested exceptions

- L4 executes with interrupts disabled for performance, no concurrency
- Must invoke untrusted user mode page-fault handlers
  - Potential for DOSing other thread (i.e., page fault handler hangs)
- Can use timeouts to avoid DOS attacks
  - Complex, goes against minimalist design

Why long IPCs?

POSIX-style APIs

- Use message passing between apps and OS, e.g., write(fd, buf, nbytes)

Linux became de-facto standard

- Communicate via shared memory

Long IPC abandoned

Supporting POSIX not as critical

- Message passing can be emulated anyway via shared memory

IPC destinations

Initially use thread identifier (why?)

- Wanted to avoid cache an TLB pollution

But

- Poor information hiding (e.g., multi-threaded server has to expose the structure to the clients)
- Large caches and TLBs reduced pollution

Thread IDs *replaced* by port-like endpoints

Timeouts

Synchronous IPC may lead to thread being blocked indefinitely

- E.g., a thread which waits for another thread that hangs

Solution: timeouts

- No reliable way to pick a timeout; application specific

Timeouts abandoned

Ended up just using two values: 0 and infinity

- Client sends and receives with infinite timeouts
- Servers requests with an infinite timeout but replies with a zero timeout

Asynchronous IPCs

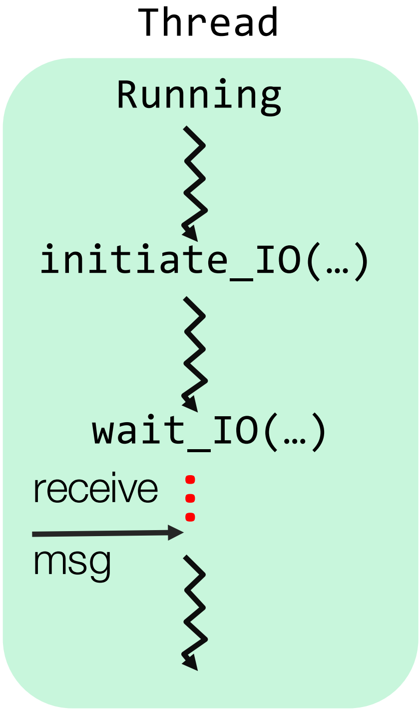
Insufficient (Why?)

Disadvantages of synchronous IPCs

- Have to block on IO operations
- Forces apps to use multithreading
- Poor choice for multicores (no need to block if IO executes on another core!)

Want async IPCs

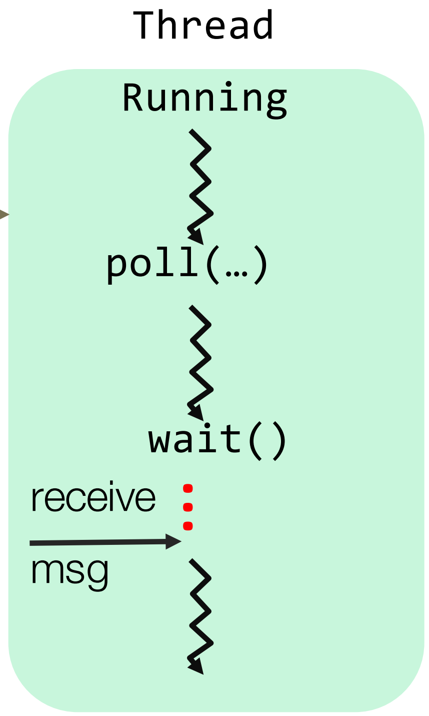
- Want something like select()/poll()/epoll() in Unix



Async notifications

Sending is non-blocking and asynchronous

Receiver, who can block or poll for message →



seL4: Asynchronous Endpoints (AEP)

- Single-word notification field
- Send sets a bit in notification field
- Bits in notification field are ORed → notification
- wait(), effectively select() across notification fields

Added async notifications to complement syn IPCs

Lazy scheduling

What is the problem?

- Lot’s of queue manipulations: threads frequently switch between ready and wait queues due to the rendezvous IPC model

Lazy scheduling:

- When a thread blocks on an IPC operation, leave it in ready queue
- Dequeue all blocked threads at next scheduling event

Why does it work?

- Move work from a high-frequency IPC operation to the less frequently scheduler calls

Benno scheduling

Lazy scheduling drawback

- Bad when many threads → worst-case proportional with # of threads

Benno scheduling

- Ready queue contains all runnable threads, except current running one
- When a thread is unblocked by an IPC operation, run it without placing in ready queue (as it may block again very soon)
- If running thread is preempted, place it in ready queue
- Still need to maintain wait queues but typically they are in hit cache

Replace lazy scheduling by Benno scheduling

Summary

Original design decision	Maintained/Abandoned	Notes
Synchronous IPC	✚	Added async notifications
In-register msg transfer	✖	Replaced physical with virtual registers
Long IPC	✖	
IPC timeouts	✖	
Clans and chiefs	✖	
User level drivers	✔	
Process hierarchy	✖	
Recursive page mapping	−	Some retained it some didn’t
Kernel memory control	✚	Added user-level control

Summary (cont'd)

Original design decision	Maintained/Abandoned	Notes
Scheduling policies	?	Unresolved: no policy agnostic solution
Multicores	?	Unresolved: cannot be verified
Virtual TCP addressing	✖	
Lazy scheduling	✖	Replaced with Benno scheduling
Non-preemptable kernel	✔	Mostly maintained
Non-portability	✖	Mostly portable
Non-standard calling	✖	Replaced by C standard calling convention
Language	✖	Assembly/C++ mostly replaced by C

Discussions: L4 tenets

Minimalist design: strict interpretation of e2e argument

- Only functionality that cannot be implemented completely in the app
- No policies in the microkernel

Obsessive optimization of IPC

Unlike Mach, didn't care about portability (at least initially)

So what got in besides IPC?

- Scheduling, including scheduling policies
- Some device drivers: timer, interrupt controller
- Minimal memory management

What drove L4’s evolution?

Application domain: embedded devices (natural fit!)

- Small footprint
- Devices ran few applications, didn’t need all OS services (e.g., file system)

Embedded devices required:

- Security and resilience → special attention to DoS attacks, formal verification
- Real-time guarantees → non-preemptable kernel

What drove L4's evolution? (cont'd)

User experience, e.g.,

- New features, e.g., async IPC
- Remove features not useful: timeouts, clans & chiefs

Software evolution:

- E.g., Linux raise and POSIX decline obviate the need for long IPCs

Hardware advances

- Bigger caches, bigger TLBs, better context switching support → obviate the need for some optimizations (e.g., virtual TLBs. Thread IDs as destination IDs)
- Multicores → push for some optimizations (async wait)

Did microkernels take over the world?

Pretty much…

- MacOS, based on NeXT, based on Mach
- iOS has both bits of Mach and L4
- Windows: hybrid (similar design goals to Mach)

With one notable exception, **Linux!**

So why didn’t take over *entire* world!

Hardware standardization:

- Intel and ARM dominating
- Less need for portability, one of main goals of Mach

Software standardization:

- Windows, MacOS/iOS, Linux/Android
- Less need to factor out common functionality

Maybe just a fluke?

- Linux could have been very well adopted the microkernel approach
- Philosophical debate between Linus and Andy Tanembaum
  - One of Linus main arguments: there is only i386 I need to write code for! (http://www.oreilly.com/openbook/opensources/book/appa.html)

seL4 – Takeaway Goal

Functional verification of microkernels is possible

Performance of verified kernels can be OK

BUT:

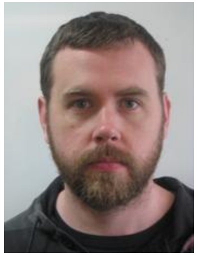
Verification is a colossal effort

Still needs to assume compiler correctness (➔ huge trusted base)

seL4 - Authors

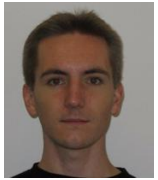






Gerwin Klein Kevin Elphinstone Gernot Heiser June Andronick David Cock





Philip Derrin Kai Engelhardt Michael Norrish Harvey Tuch

Dhammika Elkaduwe Rafal Kolanski Thomas Sewell Simon Winwood

seL4 – Project Leaders

Gerwin Klein

- TU Munich (PhD)
- University of New South Wales
- Does not put a CV on his webpage

Gernot Heiser

- ETH Zurich (PhD, 1991)
- University of New South Wales
- Created Startup “Open Kernel Labs” to sell L4 technology
- Collaborated with Jochen Liedtke (L4)

Kevin Elphinstone

- University of New South Wales
- Does not put a CV on his webpage
- Collaborated with Jochen Liedtke (L4)

Secure L4 – Design Goal

Create a formal model of a microkernel

Implement the microkernel

Prove that it always behaves according to the specification

Assumptions

Hardware works correctly

Compiler produces machine code that fits their formalization

Some unchecked assembly code is correct

Boot loader is correct

How to design kernel + spec?

Bottom-Up-Approach:

Concentrate on low-level details to maximize performance

Problem:

Produces complex design, hard to verify

Reminder

Not all equivalent programs are equally amenable to verification

```
void swap(ptr A, ptr B)
{
      ptr C := A;
      A := B;
      B := C;
}
```

vs.

```
void swap(ptr A, ptr B)
{
      A := A xor B;
      B := A xor B;
      A := A xor B;
}
```

Postcondition: $A_{post} = B_{pre} \wedge B_{post} = A_{pre}$

How to design kernel + spec?

Top-Down-Approach:

Create formal model of kernel

- Generate code from that

Problem:

High level of abstraction from hardware

How to design kernel + spec?

Compromise:

Build prototype in high-level language (Haskell)

- Generate “executable specification” from prototype
- Re-implement executable specification in C
- Prove refinements:
  - C ⇔ executable specification
  - Executable specification ⇔ Abstract specification (more high-level)

Concurrency is a problem

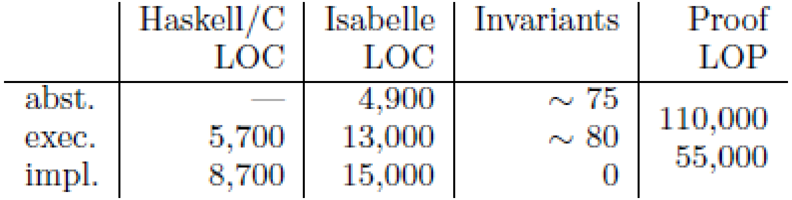
Multiprocessors not included in the model

- seL4 can only run on a single processor

Interrupts are still there

- Yield points need to establish all system invariants

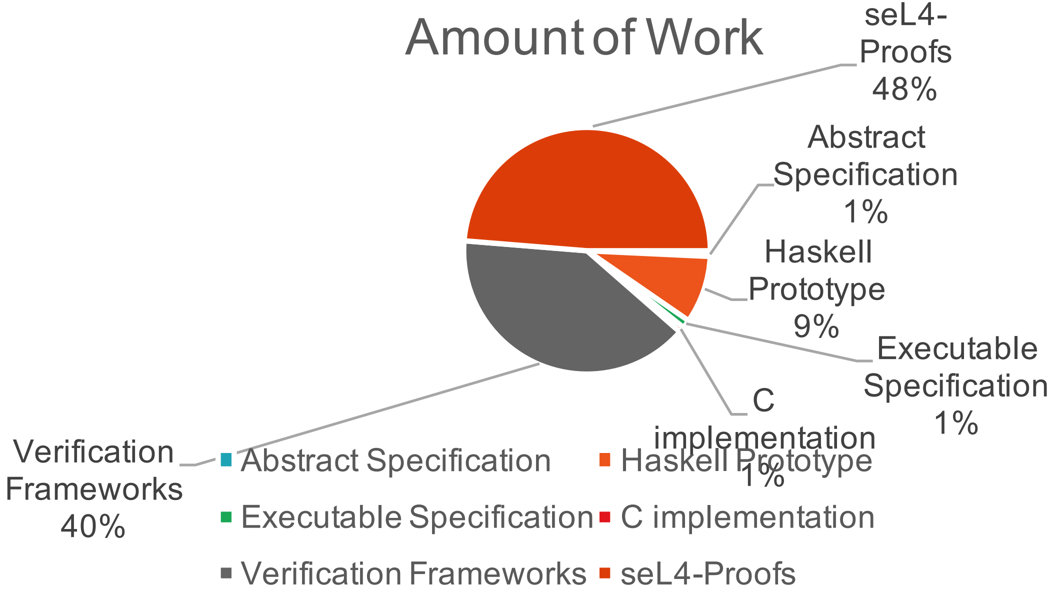
Cost of Verification



	Haskell/C LOC	Isabelle LOC	Invariants	Proof LOP
abst.	—	4,900	~ 75	110,000
exec.	5,700	13,000	~ 80	55,000
impl.	8,700	15,000	0	

Source: seL4, Klein
et al.

Cost of Verification



Source of Data: seL4, Klein et al.

Takeaway

Functional verification of microkernels is possible
Performance of verified kernels can be OK

BUT:
Verification is a colossal effort
Still needs to assume compiler correctness (➔ huge trusted base)

Discussion
Is proving functional correctness worth the effort?

Singularity vs. seL4

Goal

Singularity	seL4
A verifiably safe system. Kernel should fail “safely” when an error occurs.	A verifiably correct system. There just should not be any errors.

Ease of Verification

Singularity	seL4
Most guarantees come for free Annotations and contracts can give more guarantees	Several person-years just for proving about 80 invariants.

Perspective

Lots of room between Singularity and seL4

- I.e.: more parts of Singularity can be verified for functional correctness

Both are verified microkernels

- Good Isolation → additional components can be verified independently