



Open Kernel Labs™

Be open. Be safe.

UNSW

Introduction

COMP9242

2009/S2 Week 1

These slides are distributed under the Creative Commons Attribution 3.0 License

- You are free:
 - **to share** — to copy, distribute and transmit the work
 - **to remix** — to adapt the work
- Under the following conditions:
 - **Attribution.** You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:
 - “Courtesy of Gernot Heiser, [Institution]”, where [Institution] is one of
 - “UNSW”, “NICTA”, or “Open Kernel Labs”
- The complete license text can be found at <http://creativecommons.org/licenses/by/3.0/legalcode>

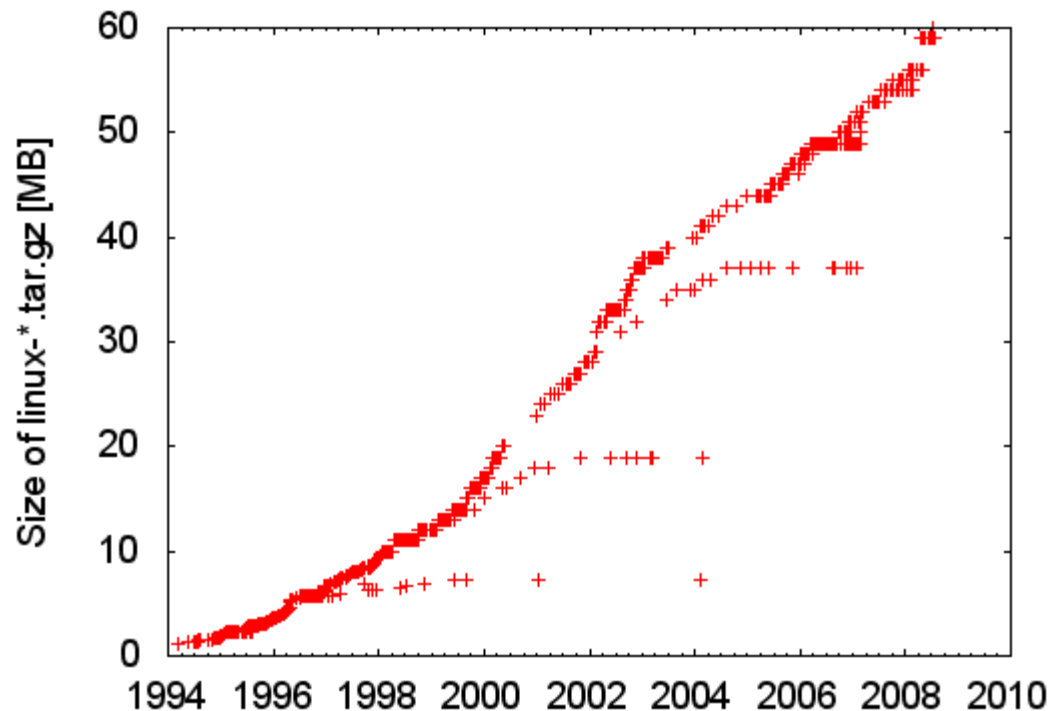
Outline

- *Introduction: What are microkernels?*
- Microkernel Performance
- L4 History and Future
- Basic L4 concepts

Why Microkernels?

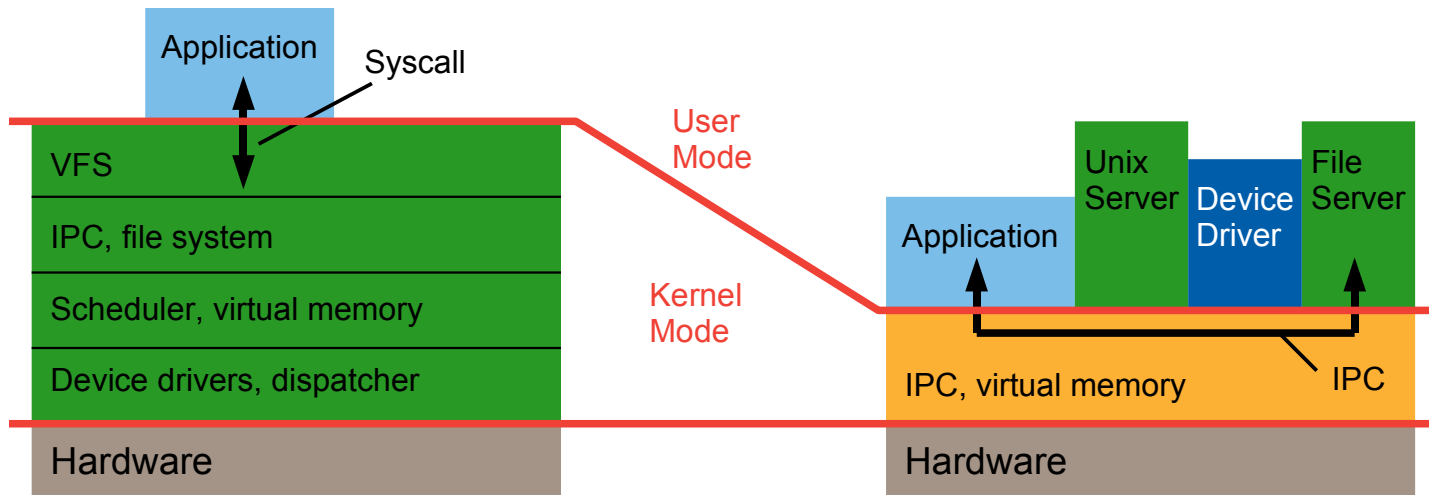
Monolithic Kernel

- Kernel has access to everything
 - all optimizations possible
 - all techniques/mechanisms/concepts implementable
- Can be extended by simply adding code
- Cost: complexity
 - growing size
 - limited maintainability



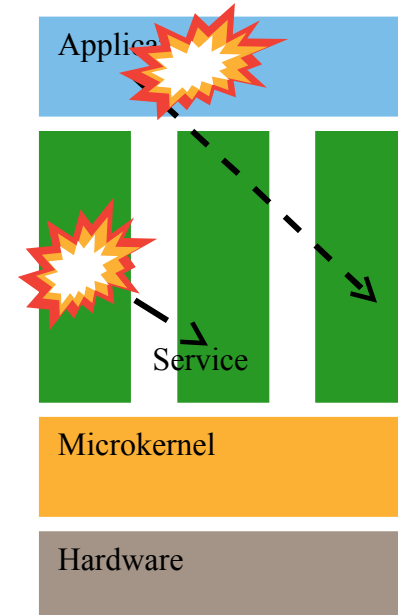
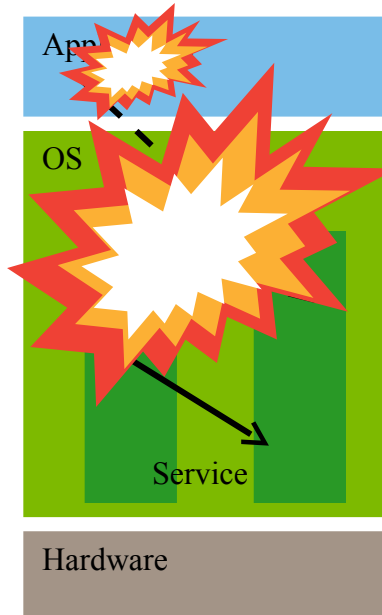
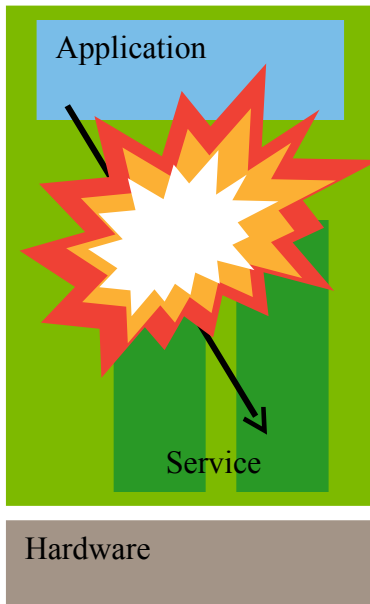
Microkernel: Idea

- Small kernel providing core functionality
 - Only code running in privileged mode
- Most OS services provided by user-level servers
- Applications communicate with servers via message-passing IPC



Trusted Computing Base (TCB)

Definition: The part of the system which can circumvent security



System: traditional embedded

Linux/
Windows

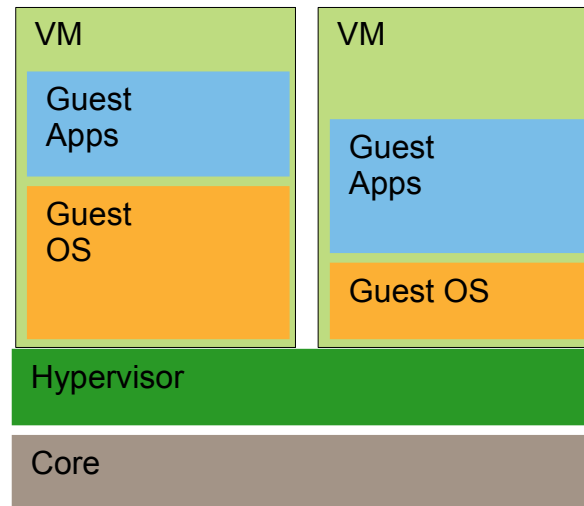
Microkernel-based

TCB: **all** code

100,000's LOC

10,000's LOC

- Partition system into several subsystems
 - Each partition runs its own operating system
 - Hypervisor controls resources
 - Hypervisor is kind-of microkernel



- Typical uses
 - Server consolidation: multiple logical machines on single physical
 - Embedded systems: high-level OS co-hosted with RTOS

- Combat kernel complexity, increase robustness, maintainability
 - dramatic reduction in amount of kernel code
 - modularity with hardware-specific services
 - normal resource management available to OS services
- Flexibility, adaptability
 - policies at user level, subject to change
 - additional services provided by adding servers
- Hardware abstraction
 - hardware-dependent part of system is small, easy to optimise
- Security, safety
 - internal protection boundaries

REALITY CHECK!
slow, inflexible
100usec IPC

Outline

- Introduction: What are microkernels?
- *Microkernel Performance*
- L4 History and Future
- Basic L4 concepts

→ **First-generation microkernels**

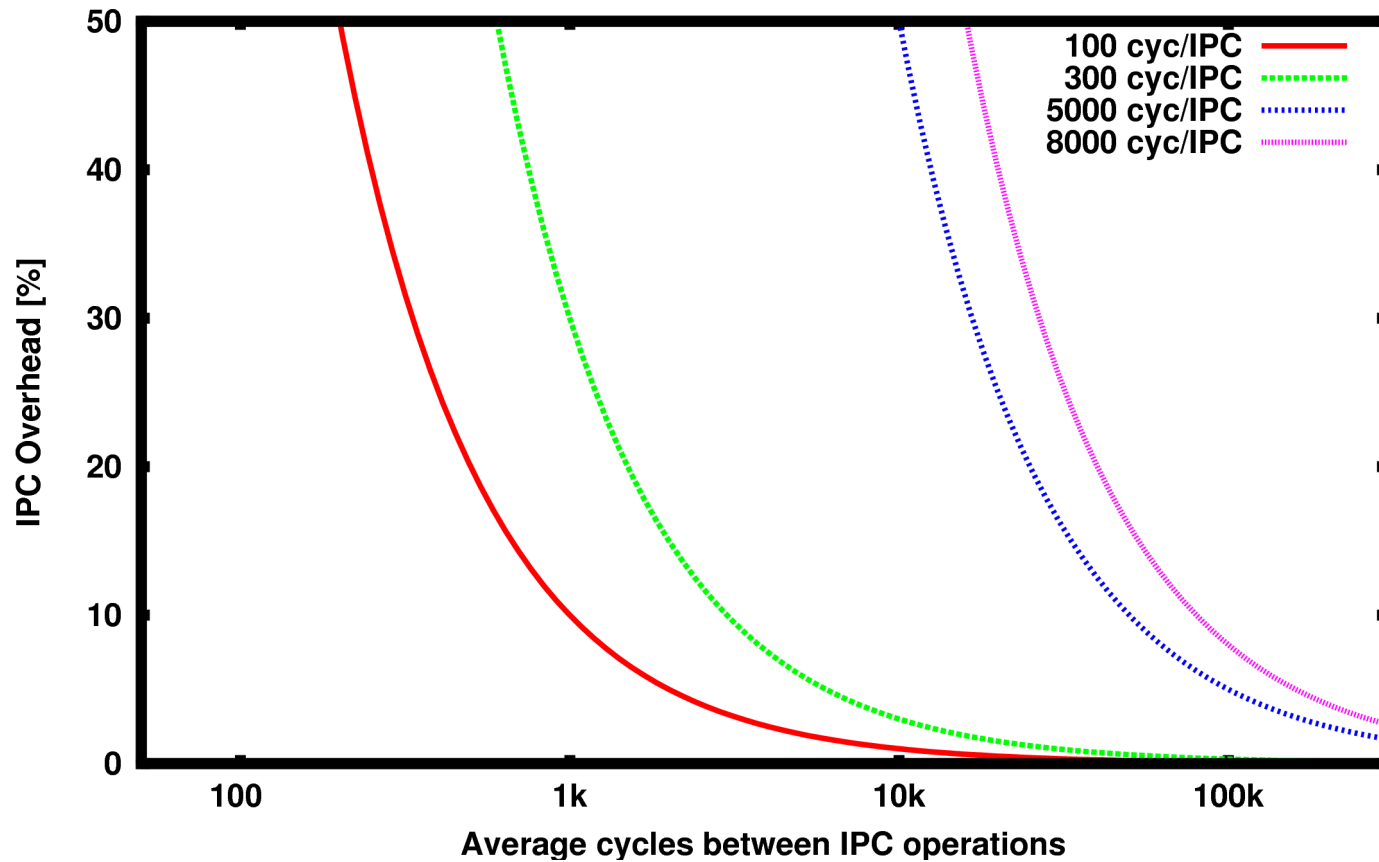
- Mach, Chorus, Amoeba, QNX
... were slow...
- 100 μ s IPC
- almost independent of clock speed!

→ L4 did better

- Close to hardware cost
- 20 times faster than Mach on identical hardware (i486)

Microkernel	IPC cost [Cycles]
Mach	5750
Amoeba	6000
Spin	6783
L4	250

IPC Cost Implications



L4 Performance: Cross Address-Space IPC

Architecture	Intra-core Cycles	Inter-core Cycles
ARM XScale PXA255 400MHz	155	
MIPS-64 100MHz dual core	109	690
Pentium 3	305	
AMD-64	230	
Itanium 2	36	

- IPC overhead generally within 20% of bare hardware cost
- Essentially as fast as it gets

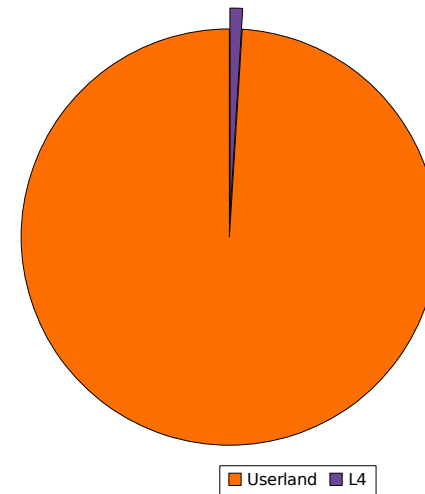
First-generation microkernels were slow

→ Reason: Poor design [Liedtke SOSP'95]

- complex API
- Too many features
- Poor design and implementation
- Large cache footprint \Rightarrow memory-bandwidth limited

→ L4 is fast due to small cache footprint

- 10–14 I-cache lines
- 8 D-cache lines
- Small cache footprint \Rightarrow CPU limited

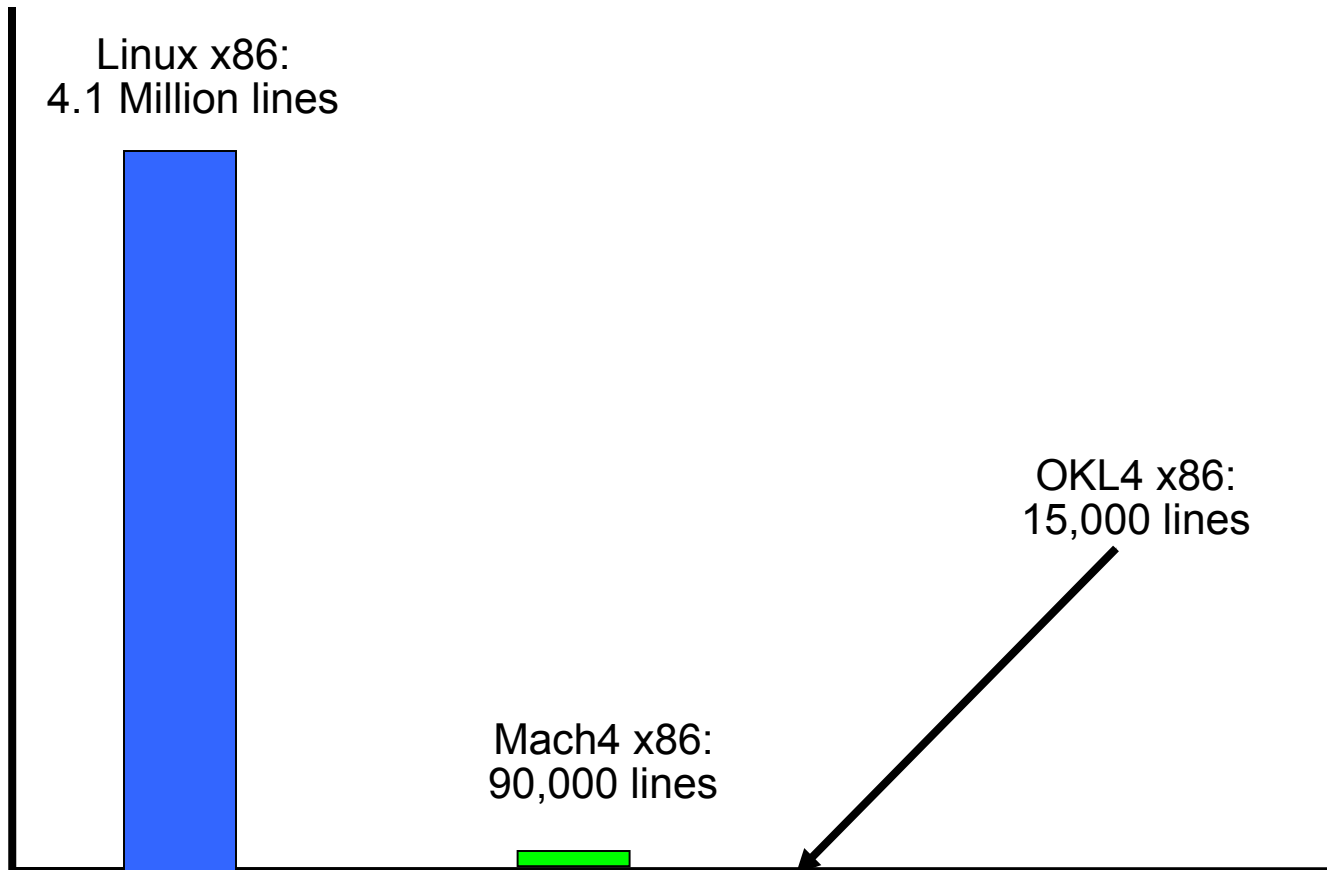


What makes Microkernel Fast?

- Small cache footprint — but how?
 - Minimality: no unnecessary features
 - Orthogonality: complementary features
 - Well-designed, and *well implemented* from scratch!
- Kernel provides *mechanisms*, not *services*
- Microkernel design principle (Minimality):

A feature is only allowed in the kernel if this is required for the implementation of a secure system.
- “*Small is beautiful!*”

Size Comparison



L4 Kernel Size

→ Source code (OKL4 Rel 2.5)

- \approx 9k LOC architecture-independent
- \approx 0.5–6k LOC architecture/platform-specific

→ **Memory** footprint kernel (not aggressively minimised):

- Using gcc (poor code density on RISC/EPIC architectures)

Architecture	Version	Text	Total
x86	L4Ka	52k	98k
Itanium	L4Ka	173k	417k
ARM	OKL4	48k	78k
PPC-32	L4Ka	41k	135k
PPC-64	L4Ka	60k	205k
MIPS-64	NICTA	61k	100k

→ Fast IPC path footprint (typical)

- 10-14 I-cache lines
- \sim 10 D-cache lines

Outline

- Introduction: What are microkernels?
- Microkernel Performance
- *L4 History and Future*
- Basic L4 concepts

L4 History: V2 API

- Original version by Jochen Liedtke (GMD) \approx 93–95
 - “Version 2” API
 - i486 assembler
 - IPC 20 times faster than Mach [SOSP 93, 95]
 - Proprietary code base (GMD)
- Other L4 V2 implementations:
 - L4/MIPS64: assembler + C (UNSW) 95–97
 - Fastest kernel on single-issue CPU (100 cycles on MIPS R4600)
 - Open source (GPL)
 - L4/Alpha: PAL + C (Dresden/UNSW), 95–97
 - First released SMP version (UNSW)
 - Open source (GPL)
 - Fiasco (Pentium): C++ (Dresden), 97–99, ongoing development
 - Open source (GPL)

L4 History: X.1 API

- Experimental “Version X” API
 - Improved hardware abstraction
 - Various experimental features (performance, security, generality)
 - Portability experiments
- Implementations
 - Pentium: assembler, Liedtke (IBM), 97–98
 - Proprietary
 - *Hazelnut* (Pentium+ARM), C, Liedtke et al (Karlsruhe), 98–99
 - Open source (GPL)

L4 History: X.2/V4 API

- “Version 4” (X.2) API, 02
 - Portability, API improvements
- L4Ka::Pistachio, C++ (plus assembler “fast path”)
 - x86, PPC-32, Itanium (Karlsruhe), 02–03
 - Fastest ever kernel (36 cycles on Itanium, NICTA/UNSW)
 - MIPS64, Alpha (NICTA/UNSW), 03
 - Same performance as V2 kernel (100 cycles single issue)
 - ARM, PPC-64 (NICTA/UNSW), x86-64 (Karlsruhe), 03–04
 - Open source (BSD license)
- Portable kernel:
 - ≈ 3 person months porting for core functionality
 - 6–12 person months for full functionality & optimisation

L4 History: N1 API

- NICTA L4-embedded (N1) API, 05–06
 - Transitional API (aiming to support strong isolation/security)
 - De-featured (timeouts, “long” IPC, recursive mappings)
 - Reduced memory footprint for embedded systems
- NICTA::Pistachio-embedded
 - Derived from L4KA::Pistachio
 - ARM7/9, x86, MIPS
 - unreleased (incomplete) ports to PPC 405, SPARC, Blackfin
 - student projects
 - Open source (BSD License)

L4 Present: OKL4

- OKL4 API
 - Further evolution of N1 API, NICTA::Pistachio-embedded code base
 - IPC control (information-flow control)
 - Kernel resource management
- Commercially-developed L4 system by Open Kernel Labs (OK Labs)
 - Commercial-strength code base
 - Used in mobile phone handsets (presently >300M deployed)
 - Professional services for L4 users
 - Commercialisation of present NICTA microkernel research

L4 Future: High-Security API

→ seL4 Project

- Conducted by NICTA in close collaboration with OK
- API suitable for highly secure systems (military, banking etc)
- Complete control over communication and system resources
- Proofs of security properties (Common Criteria)
- Suitable for formal verification of implementation

→ Status:

- Semi-formal specification in Haskell
- “Executable spec”: Haskell implementation plus ISA simulator
- C-only implementation, performance at par with OKL4
- Formal (machine-checked) proof of isolation properties
- Drives on-going OKL4 API evolution
 - OKL4 2.1 release represents significant step towards seL4 security features
 - full capability model since early '09

L4 Future: Formal Verification

→ L4.verified Project

- Conducted by NICTA in close collaboration with OK
- Mathematical proof of implementation correctness (“bug-free kernel”)
- Machine-checked proofs
- Closely linked with seL4 project
- Never done before!

→ Status:

- Proofs of several subsystems
- Extensive proof libraries
- Completed refinement to Haskell level (Dec '07)
 - most formally-verified general-purpose kernel ever
- Completely verified kernel August '09
 - complete proof chain from security properties to C implementation

L4 Future: Component Architecture

→ CAmkES Project

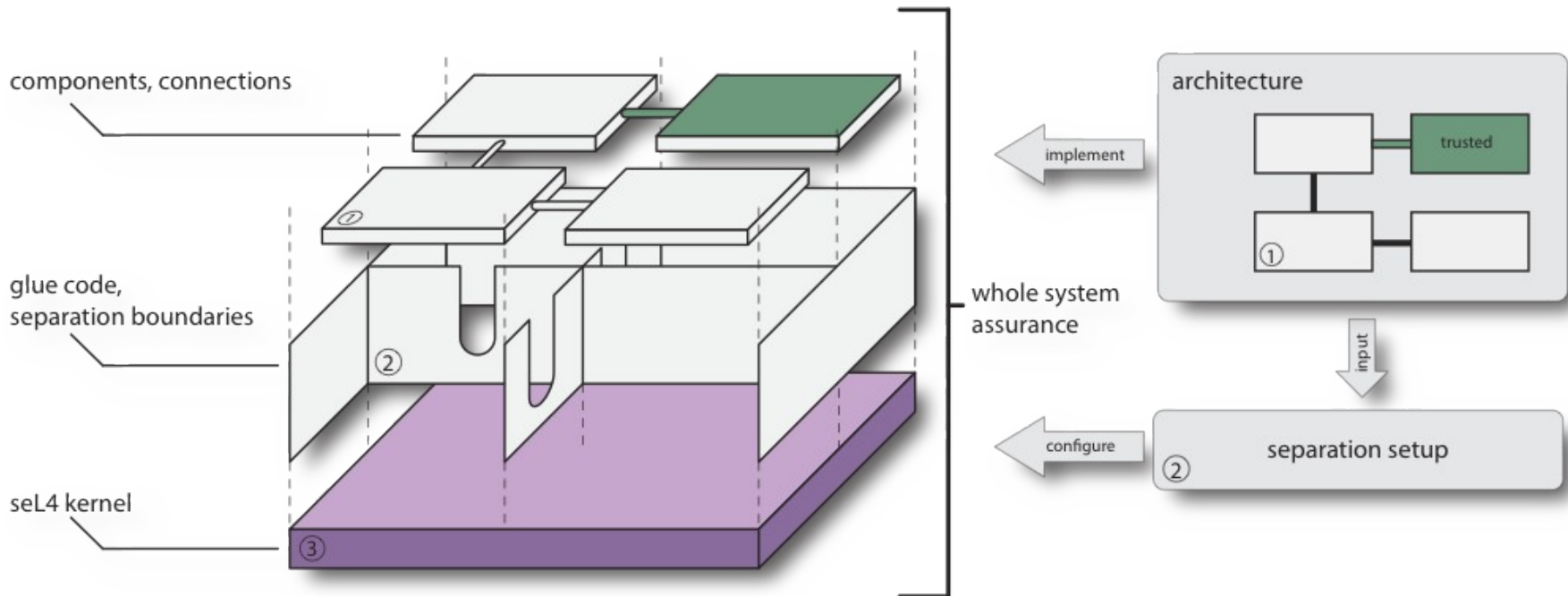
- Conducted by NICTA in close collaboration with OK
- Software-engineering framework for L4-based systems
- Light-weight component system
- Targeted at resource-starved embedded systems
- No overhead for features not used (eg. dynamic components)

→ Status:

- Static version open-sourced Dec'08, supports OKL4
- On-going work on dynamic components
- Working on support for model-driven development
- Working on support for non-functional requirements (real-time, power)

L4 Future: Trustworthy Embedded Systems

- 4-Year NICTA project, commenced July'09
- Aiming at highly secure, safe, reliable systems
- Based on verified kernel as trust base
- Aim: strong guarantees for systems with 1MLOC



Outline

- Introduction: What are microkernels?
- Microkernel Performance
- L4 History and Future
- *Basic L4 concepts*

L4 API Comments

- This overview mostly applies to L4 in general, across all APIs (except seL4)
- However, there are significant differences between different L4 APIs
 - Pistachio's "V4" API
 - Fiasco (supporting V2, V4 and various experimental APIs)
 - NICTA-embedded API (obsolete)
 - OKL4 API
- We will, if in doubt, refer to the OKL4 API
- Some of the differences affect fundamental concepts, especially naming
 - inherent consequence of the move towards a security-oriented API
 - especially capability-based protection
- The OKL4 API is itself undergoing rapid evolution
 - Started with 2.0 release (Jan '08, not public)
 - Stable API with next public release (H2'09)
 - 2.1 release has a transitional API (to ease migration of customers)
 - new concepts (capabilities)
 - some concepts vanished in the next release(s)
 - I'll try to point these out as we go....

L4 Abstractions, Mechanisms, Concepts

Three basic abstractions:

- Address spaces (or virtual MMUs) — for protection
- Threads (or virtual CPUs) — for execution
- Capabilities (for naming and access control) — *from OKL4 2.1*
- Time (for scheduling) — *May vanish in the future*

Two basic mechanisms:

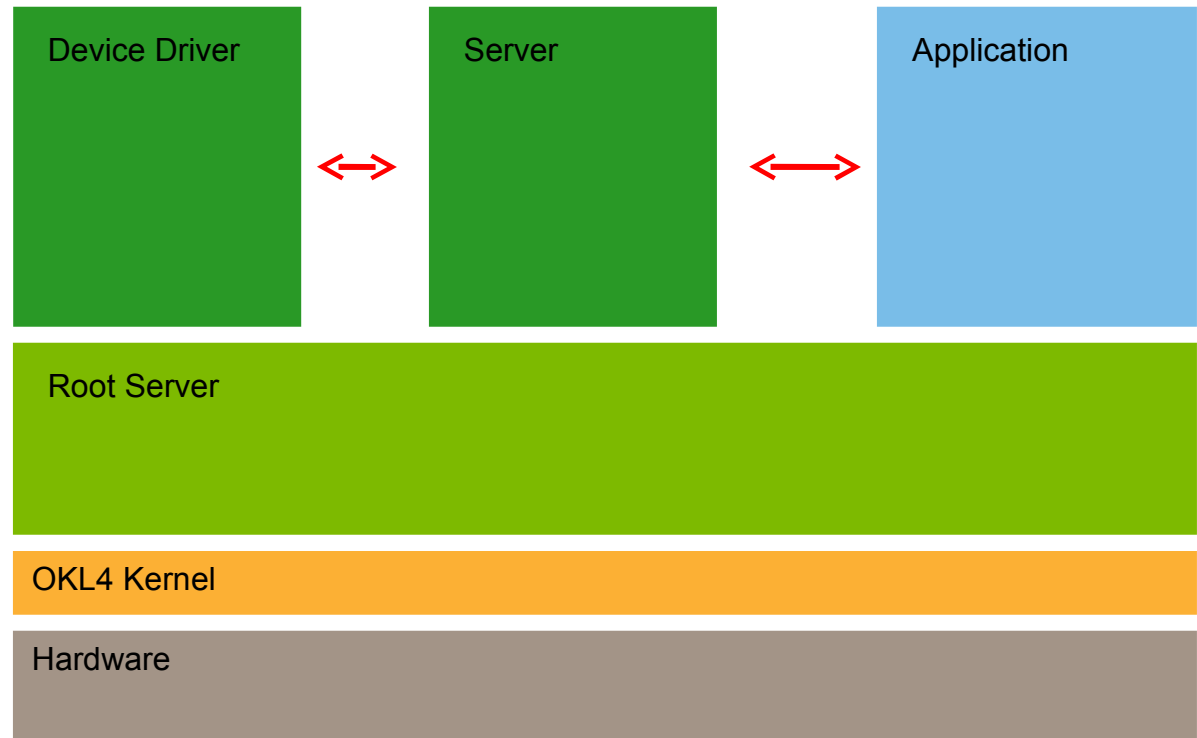
- Message-passing communication (IPC)
- Mapping memory to address spaces

Other core concepts:

- Root task — *Removed in OKL4 2.2*
- Exceptions

An L4-Based System

- OKL4 kernel
- Root server
- Device drivers
- Other servers
- Applications



L4 Abstractions: Address Spaces

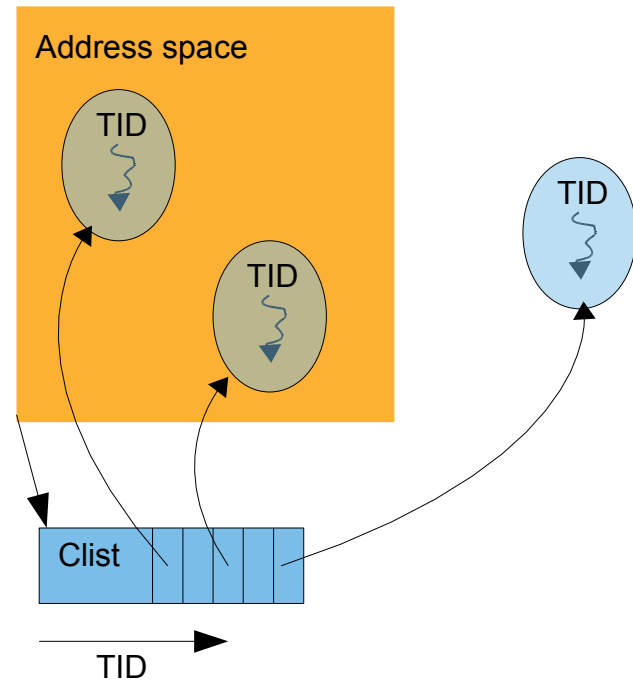
- Address space is unit of protection
 - Initially empty
 - Populated by mapping in frames
- Mapping performed by privileged MapControl() syscall
 - Can only be called from *root task*
 - Also used for revoking mappings (unmap operation)
- Root task
 - Initial address space created at boot time
 - Controls system resources
 - Privileged system calls can only be performed from the root task
 - privileged syscalls identified by names ending in “Control”
 - Privilege is not delegatable
 - this is a shortcoming of the 2.1 API
 - *OKL4 2.2 replaces this with capabilities as access tokens*
 - *removes the concept of a root task*
 - *removes the concept of a privileged system call*

L4 Abstractions: Threads

- Thread is unit of execution
 - Kernel-scheduled
- Thread is addressable unit for IPC
 - *Thread capability* used for addressing and establishing send rights
 - Called *Thread-ID* for backward compatibility
 - *New in OKL4 2.1*, previously Thread IDs were global names
- Threads managed by user-level servers
 - Creation, destruction, association with address space
- Thread attributes:
 - Scheduling parameters (time slice, priority)
 - Unique ID (hidden from userland)
 - referenced via thread capability (local name)
 - Address space
 - Page-fault and exception handler

L4 Abstractions: Capabilities

- Capabilities reference threads
 - in future versions all resources
 - actual cap word (TID) is index into per-address-space capability list (Clist)
- Capability conveys privilege
 - Right to send message to thread
 - May also convey rights to other operations on thread
- Capabilities are local names for global resources



L4 Abstractions: Time

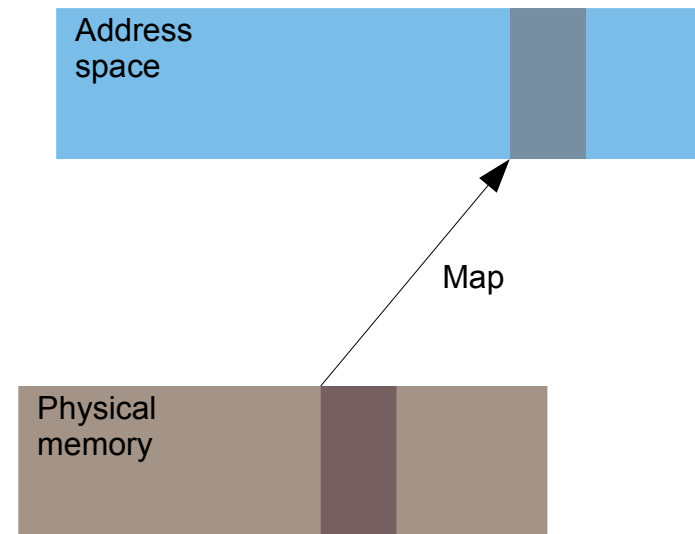
- Used for scheduling times slices
 - Thread has fixed-length time slice for preemption
 - Time slices allocated from (finite or infinite) time quantum
 - Notification when exceeded
- In earlier L4 versions also used for IPC timeouts
 - Removed in OKL4
- Future versions may remove time completely from the kernel
 - If scheduling (incl timer management) is completely exported to user level

L4 Mechanism: IPC

- Synchronous message-passing operation
- Data copied directly from sender to receiver
 - Short messages passed in registers
 - Long messages copied by kernel (semi-)asynchronously — *new in 2.1*
- Can be blocking or polling (fail if partner not ready)
- Asynchronous notification variant
 - No data transfer, only sets notification bit in receiver
 - Receiver can wait (block) or poll
- In earlier L4 versions (removed in OKL4):
 - IPC also used for mapping
 - long synchronous messages

L4 Mechanism: Mapping

- Create a mapping from a physical frame to a page in an address space
 - Privileged syscall MapControl
 - *unprivileged in OKL2.2 (access control via memory caps)*
- Typically done in response to page fault
 - VM server acting as pager
 - can pre-map, of course
- Also used for mapping device registers to drivers
 - VM server acting as pager
 - can pre-map, of course



L4 Exception Handling

→ Interrupts

- Modelled as hardware “thread” sending messages
- Received by registered (user-level) interrupt-handler thread
- Interrupt acknowledged by handler via syscall (optionally waiting for next)
- Timer interrupt handled in-kernel

→ Page Faults

- Kernel fakes IPC message from faulting thread to its pager
- Pager requests root task to set up a mapping
- Pager replies to faulting client, message intercepted by kernel

→ Other Exceptions

- Kernel fakes IPC message from exceptor thread to its exception handler
- Exception handler may reply with message specifying new IP, SP
- Can be signal handler, emulation code, stub for IPCing to server, ...

Features not in Kernel

- System services (file systems, network stacks, ...)
 - Implemented by user-level serves
- VM management
 - Performed by user-level pagers
- Device drivers
 - User-level threads registered for interrupt IPC
 - Map device registers