Introduction

COMP9242 Advanced Operating Systems
2010/S2 Week 1

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
  • growing size
  • limited maintainability

Copyright Notice

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” or “NICTA”
→ The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode
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

<table>
<thead>
<tr>
<th>System</th>
<th>TCB:</th>
</tr>
</thead>
<tbody>
<tr>
<td>traditional embedded</td>
<td>all code</td>
</tr>
<tr>
<td>Linux/Windows</td>
<td>1,000,000's LOC</td>
</tr>
<tr>
<td>Microkernel-based</td>
<td>10,000's LOC</td>
</tr>
</tbody>
</table>

Virtualization

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

Microkernel Promises

- Combat kernel complexity, increase robustness, maintainability
  - dramatic reduction in amount of privileged code
  - modularity with hardware-enforced interfaces
  - normal resource management applicable to OS services
- Flexibility, adaptability, extensibility
  - policies defined 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

100µsec IoT
Introduction: What are microkernels?

Microkernel Performance

L4 History and Future

Basic L4 concepts

First-generation microkernels

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

L4 did better

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

<table>
<thead>
<tr>
<th>Microkernel</th>
<th>CPU@MHz</th>
<th>IPC Cost [cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mach</td>
<td>i486@50</td>
<td>5750</td>
</tr>
<tr>
<td>Amoeba</td>
<td>68020@15</td>
<td>6000</td>
</tr>
<tr>
<td>Spin</td>
<td>21064@133</td>
<td>6783</td>
</tr>
<tr>
<td>L4</td>
<td>i486@50</td>
<td>250</td>
</tr>
</tbody>
</table>

IPC Cost Implications

L4 Performance: Cross Address-Space IPC

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

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

Linux x86: 4.1 Million lines
Mach4 x86: 90,000 lines
OKL4 x86: 15,000 lines

L4 Kernel Size

- Source code (OKL4 Rel 2.5)
  - ~ 9k LOC architecture-independent
  - ~ 0.5–6k LOC architecture/platform-specific
- Memory footprint kernel (not aggressively minimised)

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Version</th>
<th>Text</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>Pistachio</td>
<td>52 KiB</td>
<td>98 KiB</td>
</tr>
<tr>
<td>Itanium</td>
<td>Pistachio</td>
<td>173 KiB</td>
<td>417 KiB</td>
</tr>
<tr>
<td>MIPS-64</td>
<td>Pistachio</td>
<td>61 KiB</td>
<td>100 KiB</td>
</tr>
<tr>
<td>PPC-64</td>
<td>Pistachio</td>
<td>60 KiB</td>
<td>205 KiB</td>
</tr>
<tr>
<td>x86</td>
<td>seL4</td>
<td>74 KiB</td>
<td>98 KiB</td>
</tr>
<tr>
<td>ARM v6</td>
<td>seL4</td>
<td>64 KiB</td>
<td>112 KiB</td>
</tr>
</tbody>
</table>

- Fast IPC path footprint (typical)
  - 10–14 I-cache lines, ~10 D-cache lines
Introduction: What are microkernels?

Microkernel Performance

L4 History and Future

Basic L4 concepts

L4 History: V2 API

Original version by Jochen Liedtke (GMD) ≈ 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 Microkernel (V3)
  - Further evolution of N1 API, NICTA::Pistachio-embedded code base
  - Fully capability-based access control and resource management
  - From-scratch implementation
- OKL4 Microvisor (V4)
  - New, virtualization-oriented API
  - Fully capability-based access control and resource management
  -Shares code with OKL4 Microkernel
- We will be using a somewhat older version of the OKL4 Microkernel
  - Version 2.1, predecessor of V3
  - Transitional version, eg only part of resources cap-controlled

OKL4: Large-Scale Commercial Deployment

- Toshiba W47T
  - 2006
- HTC TyTN II
  - 2007
- HTC Dream (G1)
  - 2008
- Motorola Evoke
  - 2009

More than 750 million OKL4-based devices shipped to date!

L4 Present: seL4

- NICTA seL4 microkernel
  - API suitable for highly secure systems (military, banking etc)
    - Complete control over communication and system resources
  - C-only implementation, performance at par with OKL4
  - Novel kernel resource management
    - Kernel memory controlled by caps just as user memory
- World’s first formally-verified OS kernel!
  - Formal (machine-checked) proof of functional correctness
    - Shows implementation satisfies the spec
  - Formal proof of security properties (on-going)
  - Close to complete proof chain from high-level security requirements to C
  - Basis for on-going NICTA research agenda for trustworthy embedded systems
L4 Future: Trustworthy Embedded Systems

- 4-Year NICTA project, commenced October'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)
  - seL4
  - OKL4 API V2, V3, V4 (Microkernel and Microvisor)
- We will, if in doubt, refer to the OKL4 V2.1 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 V2.1 API is a transitional version
  - Started with 2.0 release (Jan ’08, not public)
  - Stable API 3.0 (Q1’09, not public)
  - 2.1 release is interim step (to ease migration), last open-source release
    - 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
### L4 Abstractions: Address Spaces

- Address space is a unit of protection
  - Initially empty
  - Populated by mapping in frames
- Mapping performed by a privileged `MapControl()` syscall
  - Can only be called from the 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
  - 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: Capabilities

- Capabilities reference threads
  - In future versions all resources
  - Actual cap word (TID) is an 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: Threads

- Thread is a unit of execution
  - Kernel-scheduled
- Thread is an 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: Time

- Used for scheduling time 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 servers
» VM management
  • Performed by user-level pagers
» Device drivers
  • User-level threads registered for interrupt IPC
  • Map device registers