Introduction

COMP9242
2008/S2 Week 1

Outline

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

My 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

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

Microkernel: Idea

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
Virtualization

- 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

Microkernel Promises

- Combat kernel complexity, increase robustness, maintainability
  - dramatic reduction in amount of complex code
  - modularity with hardware and software
  - normal resource management service for OS services
- Flexibility, adaptability
  - policies, services, over-clock to change
  - adapt to changes provided by physical servers
- Hardware independence
  - hardware dependent service system is small, easy to optimise
- Security, safety
  - internal protection boundaries

Outline

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

IPC Costs

- 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 (486)

<table>
<thead>
<tr>
<th>Microkernel</th>
<th>IPC cost (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mach</td>
<td>5750</td>
</tr>
<tr>
<td>Amoeba</td>
<td>6000</td>
</tr>
<tr>
<td>Spur</td>
<td>6783</td>
</tr>
<tr>
<td>L4</td>
<td>250</td>
</tr>
</tbody>
</table>

IPC Cost Implications

L4 Performance: Cross Address-Space IPC

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Intra-core Cycles</th>
<th>Inter-core Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM XScale PXA255 400MHz</td>
<td>155</td>
<td></td>
</tr>
<tr>
<td>MIPS-64 150MHz dual core</td>
<td>109</td>
<td>690</td>
</tr>
<tr>
<td>Pentium 3</td>
<td>305</td>
<td></td>
</tr>
<tr>
<td>AMD-64</td>
<td>230</td>
<td></td>
</tr>
<tr>
<td>Itanium 2</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

- 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 ⇒ memory-bandwidth limited
- L4 is fast due to small cache footprint
  - 10–14 I-cache lines
  - 8 D-cache lines
  - Small cache footprint ⇒ CPU limited

Microkernel Performance

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!”

L4 Kernel Size
- Source code (OKL4)
  - ≈ 9k LOC architecture-independent
  - ≈ 0.5–6k LOC architecture/platform-specific
- Memory footprint kernel (not aggressively minimised):
  - Using gcc (poor code density on RISC/EPIC architectures)
  - Fast IPC path footprint (typical)
    - 10-14 I-cache lines
    - 8 D-cache lines

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/MPLSoC: assembler + C (UNSW) 96–97
  - Freescale kernel on 12-issue CPU (200 cycles on MIPS R4400)
  - Open source (GPL)
- L4/Alpha: PAX + C (Dresden/UNSW), 95–97
  - First released SMP version (UNSW)
  - Open source (GPL)
- Fiasco (Pentium): C++ (Dresden), 97–99, ongoing development
  - Open source (GPL)

Size Comparison

Outline
- Introduction: What are microkernels?
- Microkernel Performance
- L4 History and Future
- Basic L4 concepts
### 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
  - Hazelwood (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 >100M deployed)
  - Forthcoming developments in CE devices (set-top boxes)
  - 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 kernel prototype, 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 seL4-like model scheduled for 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
  - Complete verified kernel by December '08
  - complete proof chain from security properties to C/asm 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 prototype available, undergoing performance tuning
  - On-going work on dynamic components
  - Working on support for model-driven development
  - Working on support for non-functional requirements (real-time, power)

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
- 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
- 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 expected in H1 ’09
  - Present 2.1 release has a transitional API (to ease migration of customers)
  - new concepts (capabilities)
  - some concepts will vanish in the next release(s)
- I’ll try to point these out as we go...

L4 Abstractions, Mechanisms, Concepts

- Three basic abstractions:
  - Address spaces (for protection)
  - Threads (for execution)
  - Capabilities (for naming and access control) — New in 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)
  - Reference 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 slivers 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 (and time 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)
- 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 OKL4.2 (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
  - Modeled 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 takes IPC message from faulting thread to its pager
  - Pager requests root bsk to set up a mapping
  - Pager replies to faulting client, message intercepted by kernel
- Other Exceptions
  - Kernel takes IPC message from exception 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