Microkernels and L4

Introduction

COMP9242 2005/S2 Week 1
WHY MICROKERNELS?

MONOLITHIC KERNEL
Why Microkernels?

Monolithic Kernel:

- Kernel has access to everything
  - all optimisations possible
  - all techniques/mechanisms/concepts implementable
Why Microkernels?

Monolithic Kernel:

- Kernel has access to everything
  - all optimisations possible
  - all techniques/mechanisms/concepts implementable
- Can be extended by simply adding code
WHY MICROKERNELS?

MONOLITHIC KERNEL:

• Kernel has access to everything
  ➔ all optimisations 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

The part of the system which must be trusted to operate correctly
The part of the system which must be trusted to operate correctly

System: traditional embedded
The part of the system which must be trusted to operate correctly

System: traditional
embedded
TRUSTED COMPUTING BASE

The part of the system which must be trusted to operate correctly

System: traditional embedded
The part of the system which must be trusted to operate correctly.

System: traditional embedded

Linux/
Windows
The part of the system which must be trusted to operate correctly

- System: traditional
  - embedded
- Linux/
  - Windows

**TRUSTED COMPUTING BASE**
The part of the system which must be trusted to operate correctly

System: traditional embedded Linux/ Windows

TRUSTED COMPUTING BASE
The part of the system which must be trusted to operate correctly

System: traditional embedded
System: Linux/Windows
The part of the system which must be trusted to operate correctly

System: traditional embedded Linux/ Windows
TRUSTED COMPUTING BASE

The part of the system which must be trusted to operate correctly

System:  traditional  Linux/  Microkernel-
        embedded        Windows  based

Hardware  Service  Hardware  Service  Hardware

Application  OS  Application

Microkernel
The part of the system which must be trusted to operate correctly

System:  traditional embedded  Linux/ Windows  Microkernel-based
**TRUSTED COMPUTING BASE**

The part of the system which must be trusted to operate correctly.

System: traditional embedded

Linux/Windows

Microkernel-based
The part of the system which must be trusted to operate correctly

System: traditional embedded  Linux/Windows  Microkernel-based
The part of the system which must be trusted to operate correctly

System:  traditional  |  Linux/Windows  |  Microkernel-based
         embedded
The part of the system which must be trusted to operate correctly

System: traditional embedded Linux/Windows Microkernel-based

TCB: all code 100,000’s loc 10,000’s loc
MICROKERNEL PROMISES

- Combat kernel complexity, increase robustness, maintainability
  - dramatic reduction of amount of privileged code
  - modularisation with hardware-enforced interfaces
  - normal resource management applicable to system services

- Flexibility, adaptability, extensibility
  - policies defined at user level, easy 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
MICROKERNEL PROMISES

• Combat kernel complexity, increase robustness, maintainability
  ➔ dramatic reduction of amount of privileged code
  ➔ modularisation with hardware-enforced interfaces
  ➔ normal resource management applicable to system services

• Flexibility, adaptability, extensibility
  ➔ policies defined at user level, easy 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
MICROKERNEL PROMISES

• Combat kernel complexity, increase robustness, maintainability
  ➔ dramatic reduction of amount of privileged code
  ➔ modularisation with hardware-enforced interfaces
  ➔ normal resource management applicable to system services

• Flexibility, adaptability, extensibility
  ➔ policies defined at user level, easy 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
MICROKERNEL PROMISES

- Combat kernel complexity, increase robustness, maintainability
  - dramatic reduction of amount of privileged code
  - modularisation with hardware-enforced interfaces
  - normal resource management applicable to system services

- Flexibility, adaptability, extensibility
  - policies defined at user level, easy 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 IPC
IPC Costs

- First-generation microkernels
  → Mach, Chorus, Amoeba
IPC Costs

- First-generation microkernels
  - Mach, Chorus, Amoeba
  
  ... were slow...

- $100\mu s$ IPC
- almost independent of clock speed!
IPC Costs

- First-generation microkernels
  - Mach, Chorus, Amoeba
  - ... were slow...
  - 100\(\mu\)s IPC
  - almost independent of clock speed!

- L4 does better
  - close to hardware cost
  - 20 times faster than Mach on identical hardware
IPC COST IMPLICATIONS
MICROKERNEL PERFORMANCE

FIRST-GENERATION MICROKERNELS WERE SLOW

• Reasons: Poor design [Liedtke SOSP 95]
  ➔ complex API
  ➔ too many features
  ➔ poor design and implementation
MICROKERNEL PERFORMANCE

FIRST-GENERATION MICROKERNELS WERE SLOW

- Reasons: Poor design [Liedtke SOSP 95]
  - complex API
  - too many features
  - poor design and implementation
  - large cache footprint: memory bandwidth limited
First-generation microkernels were slow

- Reasons: 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
WHAT MAKES A MICROKERNEL FAST?

- Small cache footprint, but how?
WHAT MAKES A MICROKERNEL FAST?

- Small cache footprint, but how?
  ➔ minimality: no unnecessary features
  ➔ orthogonality: complementary features
  ➔ well-designed, and *well implemented* from scratch!
WHAT MAKES A 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*
WHAT MAKES A 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

• Design principle (minimality):

  A feature is only allowed in the kernel if this is required for the implementation of a secure system.
L4 History
L4 History

- Original version by Jochen Liedtke (GMD) ≈ 93–95
  - “Version 2” API
  - i486 assembler
  - IPC 20 times faster than Mach [SOSP 93, 95]
L4 History

- Original version by Jochen Liedtke (GMD) \(\approx 93–95\)
  - “Version 2” API
  - i486 assembler
  - IPC 20 times faster than Mach [SOSP 93, 95]

- Other L4 V2 implementations:
  - L4/MIPS64: assembler + C (UNSW) 95–97
    - fastest kernel on single-issue CPU (100 cycles)
  - L4/Alpha: PAL + C (Dresden/UNSW), 95–97
    - first released SMP version
  - Fiasco (Pentium): C++ (Dresden), 97–99
L4 History

- Experimental “Version X” API
  - improved hardware abstraction
  - various experimental features (performance, security, generality)
  - portability experiments
L4 History

- Experimental “Version X” API
  - improved hardware abstraction
  - various experimental features (performance, security, generality)
  - portability experiments

- Implementations
  - Pentium: assembler, Liedtke (IBM), 97-98
  - Hazelnut (Pentium+ARM), C, Liedtke et al (Karlsruhe), 98–99
“Version 4” (X.2) API, 02

⇒ portability, API improvements
L4 History

- “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, 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
  ➔ UltraSPARC (NICTA/UNSW), 04–??
L4 History

- “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, 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
  - UltraSPARC (NICTA/UNSW), 04–??

- Portable kernel:
  - ≈ 3 person months for core functionality
  - 6–12 person months for full functionality & optimisation
L4 Future

- Security API
  - in discussion/design stage (NICTA, Dresden)
  - to satisfy secure system requirements
  - kernel resource management
  - improved real-time features
L4 Future

● Security API
  ➔ in discussion/design stage (NICTA, Dresden)
  ➔ to satisfy secure system requirements
  ➔ kernel resource management
  ➔ improved real-time features

● Time line:
  ➔ initial NICTA draft “seL4” expected for September 05
  ➔ stable by February 06
  ➔ possibly “executable spec” (Haskell) February 06
  ➔ C version August 06
Source code:

- ≈ 10k loc architecture independent
- ≈ 0.5–2k loc architecture specific
L4Ka::Pisatchio: Size

- **Source code:**
  - ≈ 10k loc architecture independent
  - ≈ 0.5–2k loc architecture specific

- **Memory footprint kernel (no attempt to minimise yet):**
  - using gcc (poor code density on RISC/EPIC architectures)

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Text</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>52k</td>
<td>98k</td>
</tr>
<tr>
<td>Itanium</td>
<td>173k</td>
<td>417k</td>
</tr>
<tr>
<td>ARM</td>
<td>68k</td>
<td>180k</td>
</tr>
<tr>
<td>PPC-32</td>
<td>41k</td>
<td>135k</td>
</tr>
<tr>
<td>PPC-64</td>
<td>60k</td>
<td>205k</td>
</tr>
<tr>
<td>MIPS-64</td>
<td>61k</td>
<td>100k</td>
</tr>
</tbody>
</table>

- **Fast IPC cache footprint (typical):**
  - 10–14 I-cache lines
  - 8 D-cache lines
SIZE COMPARISON

Linux (all platforms):

2.7 Million lines

Mach 4 x86:

90,000 lines

L4Ka::Pistachio/ia32

10,000 lines
# Pistachio Performance: IPC

<table>
<thead>
<tr>
<th>Architecture</th>
<th>port/optimisation</th>
<th>C++ intra AS</th>
<th>C++ inter AS</th>
<th>optimised intra AS</th>
<th>optimised inter AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium-3</td>
<td>UKa</td>
<td>180</td>
<td>367</td>
<td>113</td>
<td>305</td>
</tr>
<tr>
<td>Small Spaces</td>
<td>UKa</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium-4</td>
<td>UKa</td>
<td>385</td>
<td>983</td>
<td>196</td>
<td>416</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>UKa/NICTA</td>
<td>508</td>
<td>508</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>cross CPU</td>
<td>UKa</td>
<td>7419</td>
<td>7410</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MIPS64</td>
<td>NICTA/UNSW</td>
<td>276</td>
<td>276</td>
<td>109</td>
<td>109</td>
</tr>
<tr>
<td>cross CPU</td>
<td>NICTA/UNSW</td>
<td>3238</td>
<td>3238</td>
<td>690</td>
<td>690</td>
</tr>
<tr>
<td>PowerPC-64</td>
<td>NICTA/UNSW</td>
<td>330</td>
<td>518</td>
<td>200‡</td>
<td>200‡</td>
</tr>
<tr>
<td>Alpha 21264</td>
<td>NICTA/UNSW</td>
<td>440</td>
<td>642</td>
<td>≈70†</td>
<td>≈70†</td>
</tr>
<tr>
<td>ARM/XScale</td>
<td>NICTA/UNSW</td>
<td>250</td>
<td>11,400</td>
<td>120–140‡</td>
<td>10,000‡</td>
</tr>
<tr>
<td>FASS</td>
<td>NICTA/UNSW</td>
<td>340</td>
<td>340</td>
<td>120–140‡</td>
<td>120–140‡</td>
</tr>
<tr>
<td>UltraSPARC</td>
<td>NICTA/UNSW</td>
<td></td>
<td></td>
<td>100‡</td>
<td>100‡</td>
</tr>
</tbody>
</table>

† “Version 2” assembler kernel
‡ Guestimate!
L4 Abstractions and Mechanisms

Three basic abstractions:

- Address spaces
- Threads
- Time

Two basic mechanisms:

- Inter-process communication (IPC)
- Mapping
L4 Abstractions: Address Spaces

- Address space is unit of protection
Address space is unit of protection

- initially empty
- constructed recursively by mapping pages between address spaces
L4 Abstractions: Address Spaces

- Address space is unit of protection
  - initially empty
  - constructed recursively by mapping pages between address spaces
  - ... staring from initial address space $\sigma_0$
    - $\sigma_0$ is an identical map of physical memory
L4 Abstractions: Address Spaces

- Address space is unit of protection
  - initially empty
  - constructed recursively by mapping pages between address spaces
  - ... staring from *initial address space* $\sigma_0$
    - $\sigma_0$ is an identical map of physical memory

- Mechanism: 3 basic operations:
L4 Abstractions: Address Spaces

- Address space is unit of protection
  - initially empty
  - constructed recursively by mapping pages between address spaces
  - ... starting from \textit{initial address space} $\sigma_0$
    - $\sigma_0$ is an identical map of physical memory

- Mechanism: 3 basic operations:
  - map
L4 Abstractions: Address Spaces

- Address space is unit of protection
  - initially empty
  - constructed recursively by mapping pages between address spaces
  - ... staring from initial address space $\sigma_0$
    - $\sigma_0$ is an identical map of physical memory

- Mechanism: 3 basic operations:
  - map
Address space is unit of protection

- initially empty
- constructed recursively by mapping pages between address spaces
- ... staring from initial address space $\sigma_0$
  - $\sigma_0$ is an identical map of physical memory

Mechanism: 3 basic operations:

- map
- flush (unmap)
L4 Abstractions: Address Spaces

- Address space is unit of protection
  - initially empty
  - constructed recursively by mapping pages between address spaces
  - ... staring from initial address space $\sigma_0$
    - $\sigma_0$ is an identical map of physical memory

- Mechanism: 3 basic operations:
  - map
  - flush (unmap)
L4 Abstractions: Address Spaces

- Address space is unit of protection
  - initially empty
  - constructed recursively by mapping pages between address spaces
  - ... staring from *initial address space* $\sigma_0$
    - $\sigma_0$ is an identical map of physical memory

- Mechanism: 3 basic operations:
  - map
  - flush (unmap)
  - grant
Address space is unit of protection

- Initially empty
- Constructed recursively by mapping pages between address spaces
- ... staring from *initial address space* $\sigma_0$
  - $\sigma_0$ is an identical map of physical memory

Mechanism: 3 basic operations:

- Map
- Flush (unmap)
- Grant
L4 Abstractions: Threads

- Thread is unit of execution
  - kernel-scheduled
L4 Abstractions: Threads

- Thread is unit of execution
  ➞ kernel-scheduled

- Thread is addressable unit for IPC
  ➞ thread-ID is unique identifier
L4 Abstractions: Threads

- Thread is unit of execution
  - kernel-scheduled

- Thread is addressable unit for IPC
  - thread-ID is unique identifier

- Threads managed by user-level servers
  - creation, destruction, association with address space
L4 Abstractions: Threads

- Thread is unit of execution
  - kernel-scheduled

- Thread is addressable unit for IPC
  - thread-ID is unique identifier

- Threads managed by user-level servers
  - creation, destruction, association with address space

- Thread attributes:
  - scheduling parameters (time slice, priority)
  - unique ID
  - address space
  - page-fault and exception handler
L4 Abstractions: Time

- Used for IPC timeouts
  - relative timeouts
  - absolute timeouts
L4 Abstractions: Time

- Used for IPC timeouts
  - relative timeouts
  - absolute timeouts

- 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
L4 Mechanism: IPC

- Synchronous (blocking) message-passing operation
L4 Mechanism: IPC

- Synchronous (blocking) message-passing operation
- Data copied directly from sender to receiver
  ➔ short messages passed in registers
L4 Mechanism: IPC

- Synchronous (blocking) message-passing operation
- Data copied directly from sender to receiver
  ➔ short messages passed in registers
- Optionally map or grant pages
L4 MECHANISM: IPC

- Synchronous (blocking) message-passing operation
- Data copied directly from sender to receiver
  - short messages passed in registers
- Optionally map or grant pages
- Timeouts to prevent indefinite blocking
  - receive from *nil thread* used for timed sleep
L4 Concepts: Root Task

- First task started at boot time
- Can perform *privileged system calls*
L4 Concepts: Root Task

- First task started at boot time
- Can perform *privileged system calls*
- Controls access to resources
  - threads
  - address spaces
  - physical memory
L4 CONCEPTS: ROOT TASK

- First task started at boot time
- Can perform *privileged system calls*
- Controls access to resources
  - threads
  - address spaces
  - physical memory
L4 Concepts: Root Task

- First task started at boot time
- Can perform *privileged system calls*
- Controls access to resources
  - threads
  - address spaces
  - physical memory
L4 Concepts: Root Task

- First task started at boot time
- Can perform *privileged system calls*
- Controls access to resources
  - threads
  - address spaces
  - physical memory
L4 Concepts: Root Task

- First task started at boot time
- Can perform *privileged system calls*
- Controls access to resources
  - threads
  - address spaces
  - physical memory
L4 Concepts: Root Task

- First task started at boot time
- Can perform *privileged system calls*
- Controls access to resources
  - threads
  - address spaces
  - physical memory

![Diagram showing root task and associated components](image-url)
L4 Exception Handling

- Interrupts
- Page faults
- Other exceptions
Interrupts

- modelled as hardware “thread” sending messages
- received by registered (user-level) interrupt-handler thread
- interrupt acknowledged when handler blocks on receive
- timer interrupt handled in-kernel

Page faults

Other exceptions
L4 Exception Handling

- **Interrupts**
  - modelled as hardware “thread” sending messages
  - received by registered (user-level) interrupt-handler thread
  - interrupt acknowledged when handler blocks on receive
  - timer interrupt handled in-kernel

- **Page faults**
  - kernel fakes IPC message from faulting thread to its pager
  - pager may reply with page mapping, intercepted by kernel

- **Other exceptions**
L4 Exception Handling

- Interrupts
  - modelled as hardware “thread” sending messages
  - received by registered (user-level) interrupt-handler thread
  - interrupt acknowledged when handler blocks on receive
  - timer interrupt handled in-kernel

- Page faults
  - kernel fakes IPC message from faulting thread to its pager
  - pager may reply with page mapping, 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
Features Not In Kernel

- System services (file system, network stack, ...)
  ➝ implemented by user-level servers

- VM management
  ➝ performed by (hierarchy) of user-level pagers
FEATURES NOT IN KERNEL

- System services (file system, network stack, ...)
  ➔ implemented by user-level servers

- VM management
  ➔ performed by (hierarchy) of user-level pagers

- Device drivers
  ➔ user-level threads registered for interrupt IPC
  ➔ map device registers