Microkernels and L4

Introduction

COMP9242 2006/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
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
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
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
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
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
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 embedded  Linux/ Windows  Microkernel-based
The part of the system which must be trusted to operate correctly

System: traditional
        embedded

TCB: all code

Linux/
Windows

100,000’s loc

Microkernel-based

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

REALITY CHECK!
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
100 µsec IPC
IPC Costs

- First-generation microkernels
  - Mach, Chorus, Amoeba
First-generation microkernels

- Mach, Chorus, Amoeba

... were slow...

- 100µs IPC
- almost independent of clock speed!
IPC Costs

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

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

average cycles between successive IPCs

overhead due to IPC
L4 IPC

- Pentium: 236
- P3 Sysops: 180
- P3 Lipc?: 20

Pentium: 23
- Alpha: 33
- R4600: 55

- Pentium 166 MHz: 0.73 µs
- R4600 100 MHz: 0.91 µs
- (21164 433 MHz): 0.10 µs

P III 500 MHz
- 0.47 µs
- (hopefully)
- 0.04 µs
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 \(\Rightarrow\) memory bandwidth limited
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

● 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) \( \approx 93–95 \)
  - “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
L4 History

- “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 History**

- NICTA L4-embedded ($N_x$) API, 05–
  - transitional API (pre-seL4)
  - de-featured (timeouts, “long” IPC, recursive mappings)
  - reduced memory footprint for embedded systems
L4 History

- NICTA L4-embedded ($N_x$) API, 05–
  - transitional API (pre-seL4)
  - de-featured (timeouts, “long” IPC, recursive mappings)
  - reduced memory footprint for embedded systems

- NICTA::Pistachio-embedded, derived from L4KA::Pistachio
  - ARM9/ARM11, x86, MIPS
  - PPC 405, Blackfin under development
L4 History

- NICTA L4-embedded ($N_x$) API, 05–
  - transitional API (pre-seL4)
  - de-featured (timeouts, “long” IPC, recursive mappings)
  - reduced memory footprint for embedded systems

- NICTA::Pistachio-embedded, derived from L4KA::Pistachio
  - ARM9/ARM11, x86, MIPS
  - PPC 405, Blackfin under development

- You’ll be using the (unreleased) N2 API implementation
L4 Present

- NICTA L4-embedded commercially deployed
  - adopted by Qualcomm for CDMA chipsets
  - under evaluation/development for other products at a number of multinationals
  - about to establish strong presence in wireless and CE markets
L4 Present

• NICTA L4-embedded commercially deployed
  ➔ adopted by Qualcomm for CDMA chipsets
  ➔ under evaluation/development for other products at a number of multinationals
  ➔ about to establish strong presence in wireless and CE markets

• NICTA spinning out Open Kernel Labs
  ➔ further development of L4-embedded
  ➔ professional services for L4 users
  ➔ commercialisation of present NICTA microkernel research
L4 Future

- Security API: NICTA seL4
  - draft published March 06
  - semi-formal specification in Haskell
  - “executable spec”: Haskell implementation plus ISA simulator
  - used for exercising and porting apps
  - stable API August 06
  - C implementation end of 06
  - similar project at TU Dresden: L4sec (draft API Oct 05)
L4 Future

- Security API: NICTA seL4
  - draft published March 06
  - semi-formal specification in Haskell
  - "executable spec": Haskell implementation plus ISA simulator
  - used for exercising and porting apps
  - stable API August 06
  - C implementation end of 06
  - similar project at TU Dresden: L4sec (draft API Oct 05)

- Features:
  - user-level management of kernel resources (esp. memory)
  - low-overhead information-flow control mechanisms
  - suitable for formal verification
L4 Future

- Security API: NICTA seL4
  - draft published March 06
  - semi-formal specification in Haskell
  - “executable spec”: Haskell implementation plus ISA simulator
  - used for exercising and porting apps
  - stable API August 06
  - C implementation end of 06
  - similar project at TU Dresden: L4sec (draft API Oct 05)

- Features:
  - user-level management of kernel resources (esp. memory)
  - low-overhead information-flow control mechanisms
  - suitable for formal verification

- Formal verification of L4 implementation: L4.verified project
  - mathematical proof that implementation matches spec
Pistachio: Size

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

- **Source code:**
  - \( \approx 10k \) loc architecture independent
  - \( \approx 0.5–2k \) 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>Version</th>
<th>Text</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>L4Ka</td>
<td>52k</td>
<td>98k</td>
</tr>
<tr>
<td>Itanium</td>
<td>L4Ka</td>
<td>173k</td>
<td>417k</td>
</tr>
<tr>
<td>ARM</td>
<td>NICTA</td>
<td>55k</td>
<td>117k</td>
</tr>
<tr>
<td>PPC-32</td>
<td>L4Ka</td>
<td>41k</td>
<td>135k</td>
</tr>
<tr>
<td>PPC-64</td>
<td>L4Ka</td>
<td>60k</td>
<td>205k</td>
</tr>
<tr>
<td>MIPS-64</td>
<td>L4Ka</td>
<td>61k</td>
<td>100k</td>
</tr>
</tbody>
</table>
**Pistachio: Size**

- **Source code:**
  - $\approx 10k$ loc architecture independent
  - $\approx 0.5$–$2k$ 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>Version</th>
<th>Text</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>L4Ka</td>
<td>52k</td>
<td>98k</td>
</tr>
<tr>
<td>Itanium</td>
<td>L4Ka</td>
<td>173k</td>
<td>417k</td>
</tr>
<tr>
<td>ARM</td>
<td>NICTA</td>
<td>55k</td>
<td>117k</td>
</tr>
<tr>
<td>PPC-32</td>
<td>L4Ka</td>
<td>41k</td>
<td>135k</td>
</tr>
<tr>
<td>PPC-64</td>
<td>L4Ka</td>
<td>60k</td>
<td>205k</td>
</tr>
<tr>
<td>MIPS-64</td>
<td>L4Ka</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 Small Spaces</td>
<td>UKa</td>
<td>180</td>
<td>367</td>
<td>113</td>
<td>305</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 cross CPU</td>
<td>UKa/NICTA</td>
<td>508</td>
<td>508</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>MIPS64 cross CPU</td>
<td>NICTA/UNSW</td>
<td>276</td>
<td>276</td>
<td>109</td>
<td>109</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>340</td>
<td>340</td>
<td>151</td>
<td>151</td>
</tr>
</tbody>
</table>

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

Three Basic Abstractions:

• Address spaces
• Threads
• Time (second-class abstraction in N2 API, to vanish completely)

Two Basic Mechanisms:

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

- Address space is unit of protection
L4 Abstractions: Address Spaces

- Address space is unit of protection
  - initially empty
  - populated by mapping in frames
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)
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
  - Non-delegatable privilege (shortcoming of N2 API)
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 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 N2
L4 MECHANISM: IPC

- Synchronous message-passing operation
L4 MECHANISM: IPC

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

- Synchronous message-passing operation

- Data copied directly from sender to receiver
  ➔ short messages passed in registers

- Can be blocking or polling (fail if partner not ready)
L4 MECHANISM: IPC

- Synchronous message-passing operation
- Data copied directly from sender to receiver
  ➔ short messages passed in registers
- 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
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

Server

Driver

Root Task

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:
- **Root Task**
- **Physical Memory**
- **Client**
- **Server**
- **Driver**
L4 Exception Handling

- Interrupts

- 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**

- **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 requests root task to set up a mapping
  - pager replies to faulting client, message 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 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 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