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

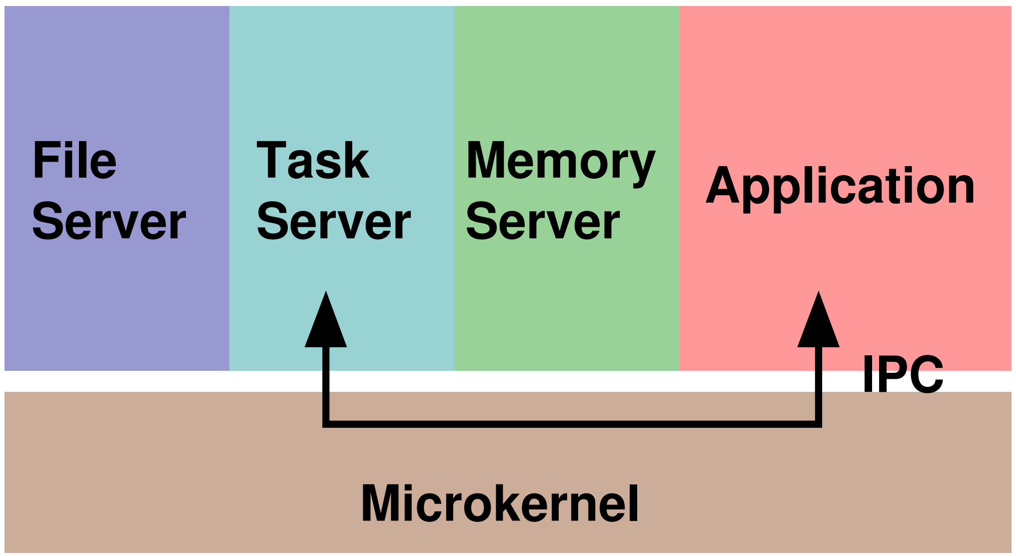
WHY MICROKERNELS?

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

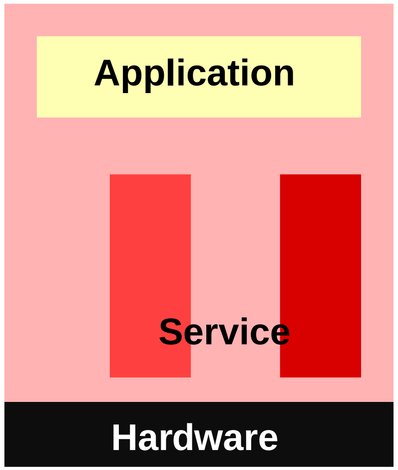


Trusted Computing Base

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

TRUSTED COMPUTING BASE

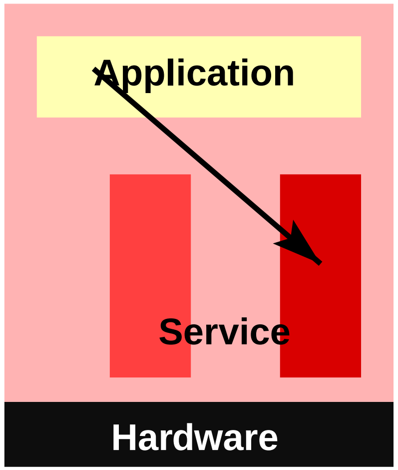
The part of the system which must be trusted to operate correctly



System: traditional
embedded

TRUSTED COMPUTING BASE

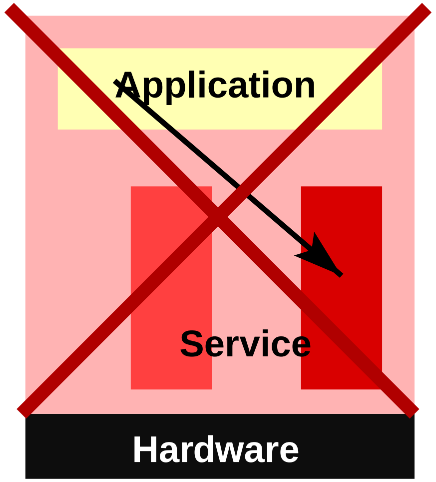
The part of the system which must be trusted to operate correctly



System: traditional
embedded

Trusted Computing Base

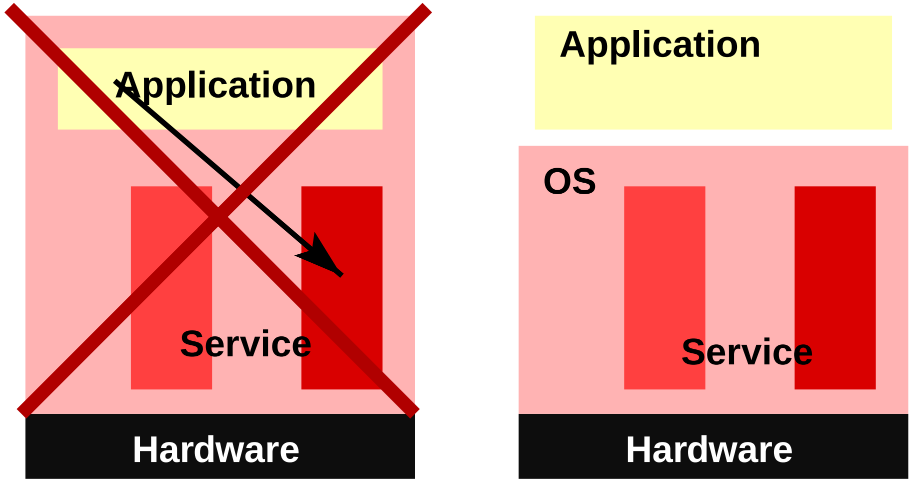
The part of the system which must be trusted to operate correctly



System: traditional
embedded

Trusted Computing Base

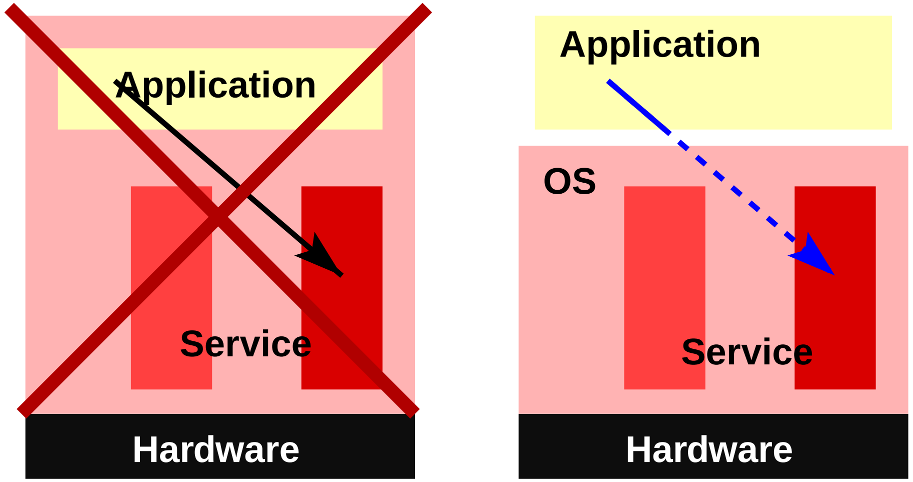
The part of the system which must be trusted to operate correctly



System: traditional embedded Linux/ Windows

TRUSTED COMPUTING BASE

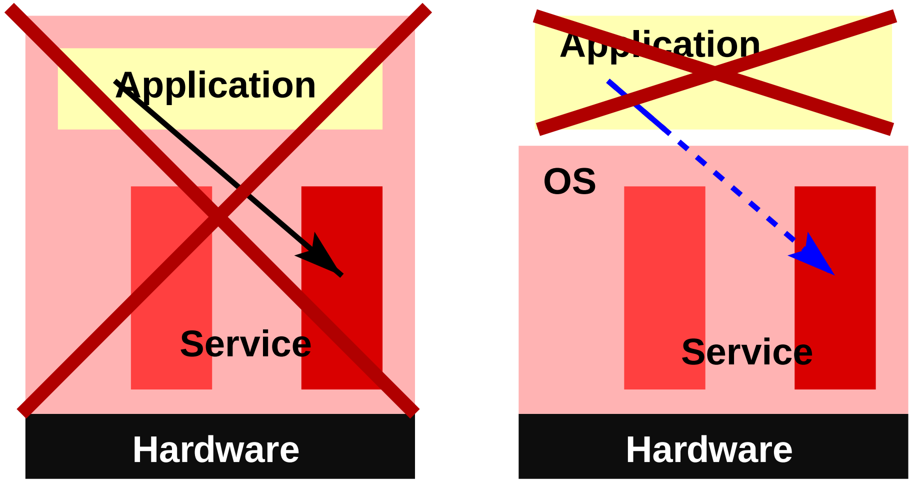
The part of the system which must be trusted to operate correctly



System: traditional embedded Linux/ Windows

TRUSTED COMPUTING BASE

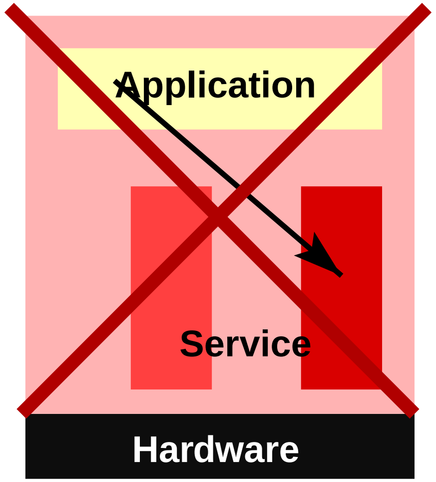
The part of the system which must be trusted to operate correctly

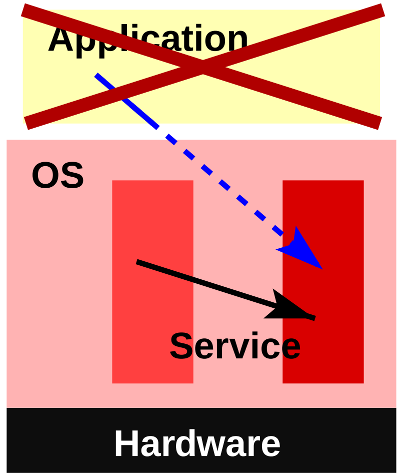


System: traditional embedded Linux/ Windows

Trusted Computing Base

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

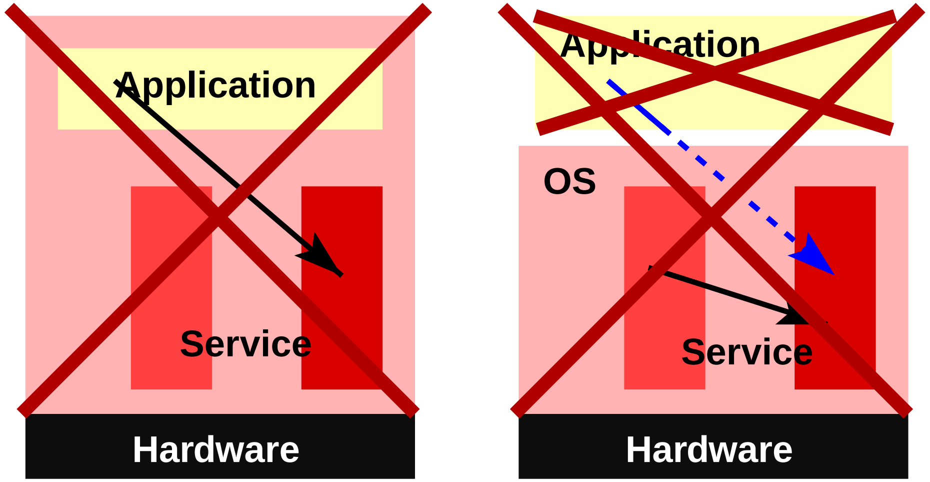




System: traditional embedded Linux/ Windows

Trusted Computing Base

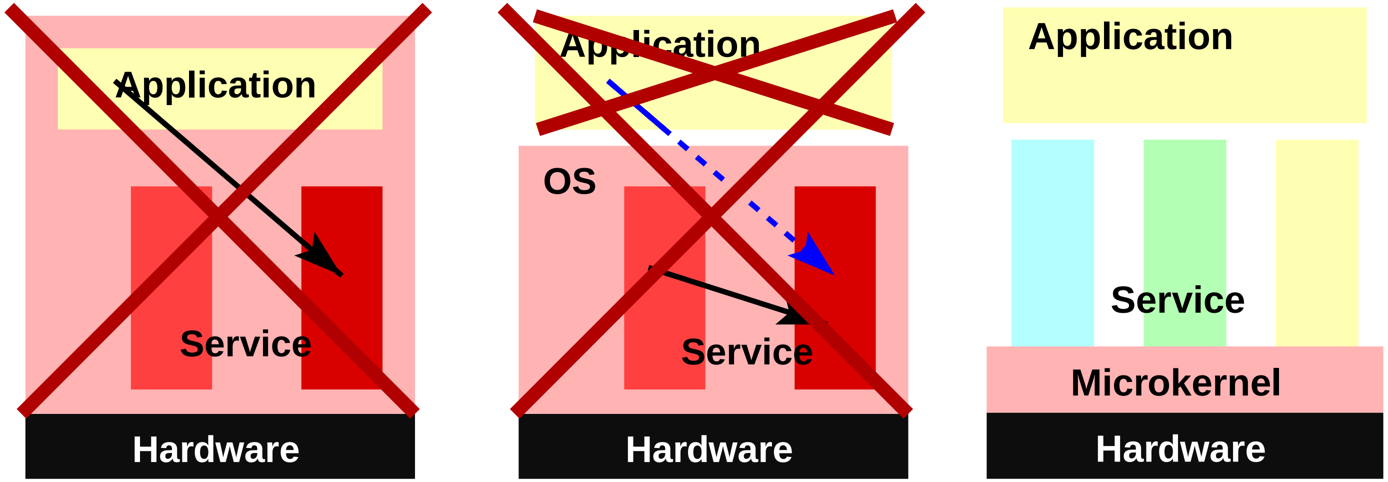
The part of the system which must be trusted to operate correctly



System: traditional embedded Linux/ Windows

Trusted Computing Base

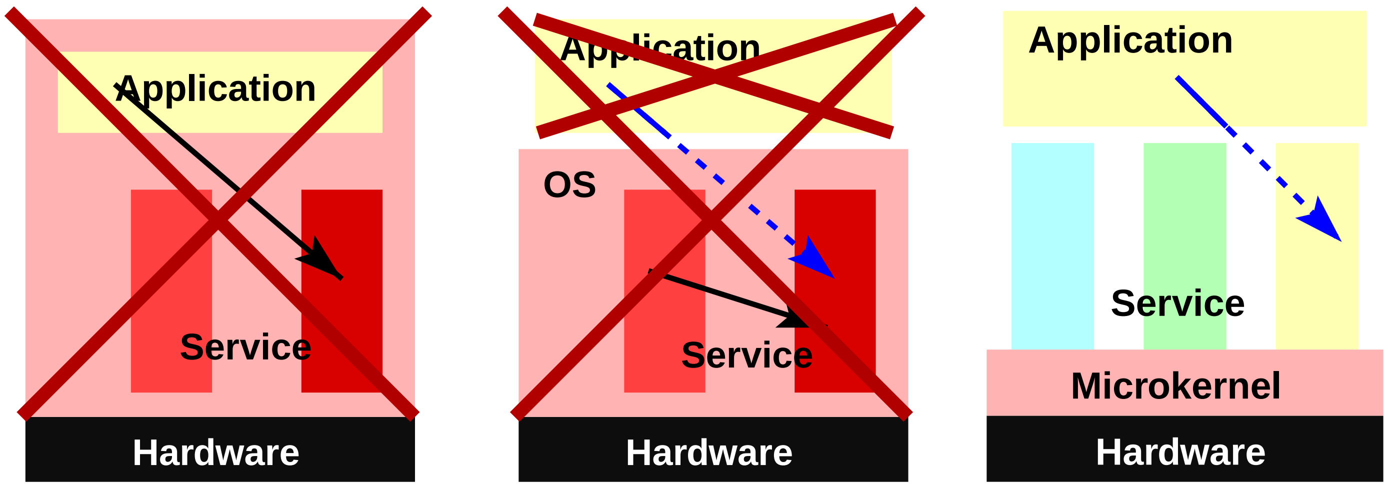
The part of the system which must be trusted to operate correctly



System:	traditional embedded	Linux/ Windows	Microkernel-based

Trusted Computing Base

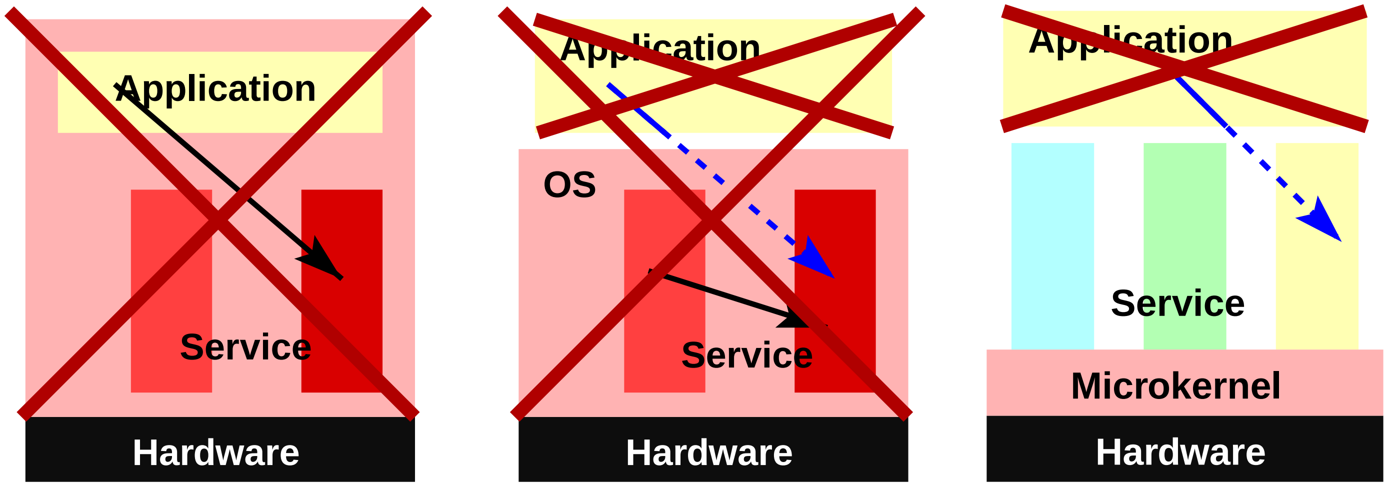
The part of the system which must be trusted to operate correctly



System: traditional embedded | Linux/ Windows | Microkernel-based

TRUSTED COMPUTING BASE

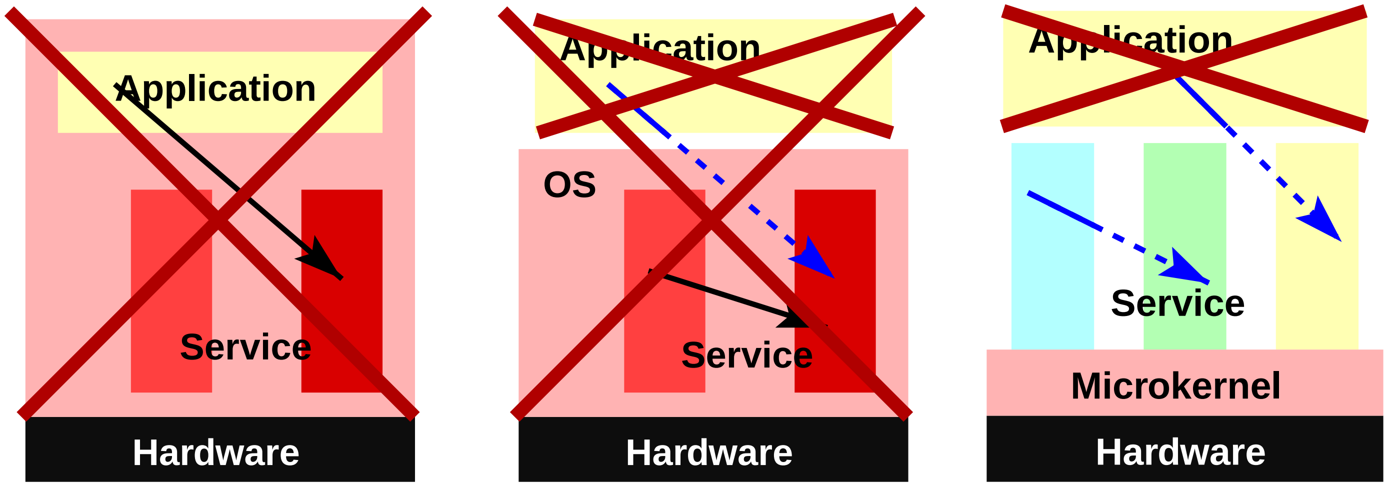
The part of the system which must be trusted to operate correctly



System: traditional embedded Linux/ Windows Microkernel-based

TRUSTED COMPUTING BASE

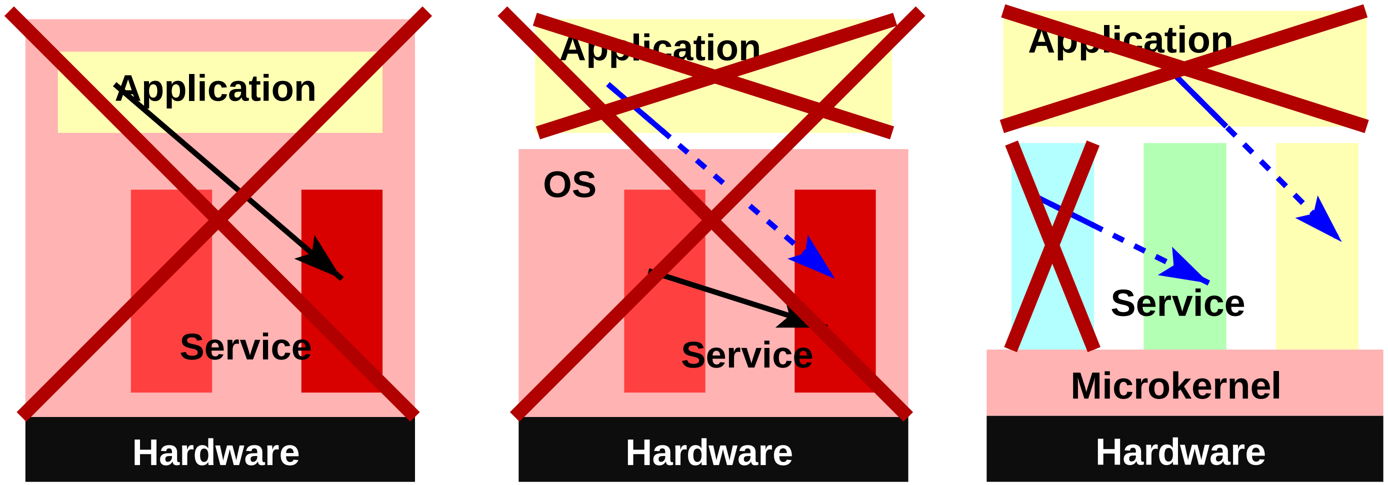
The part of the system which must be trusted to operate correctly



System: traditional embedded | Linux/ Windows | Microkernel-based

TRUSTED COMPUTING BASE

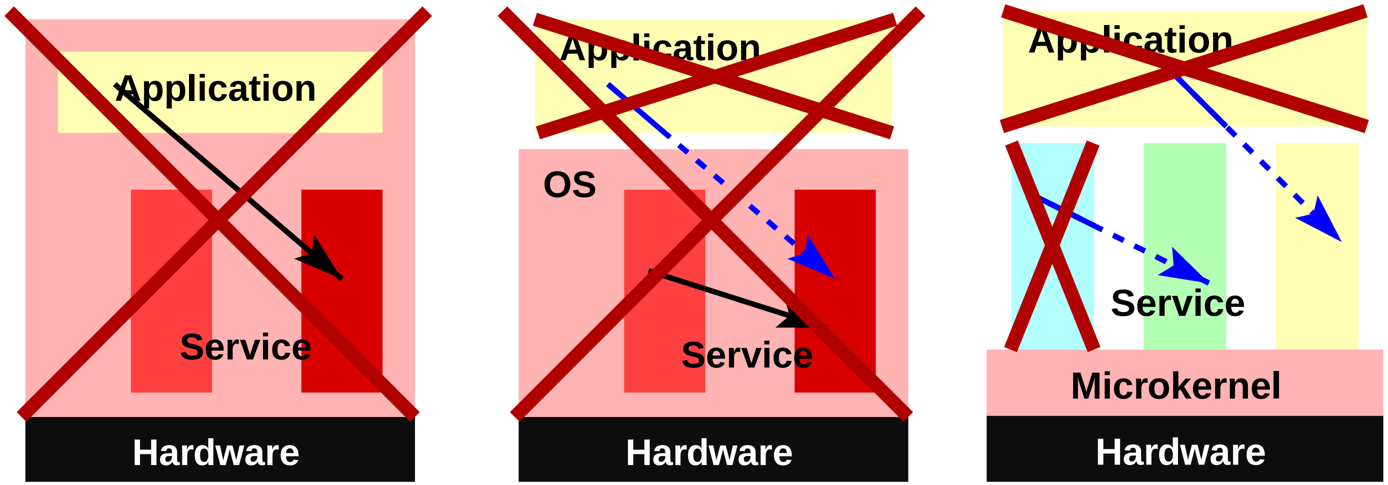
The part of the system which must be trusted to operate correctly



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
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

REALITY CHECK!

- 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...

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

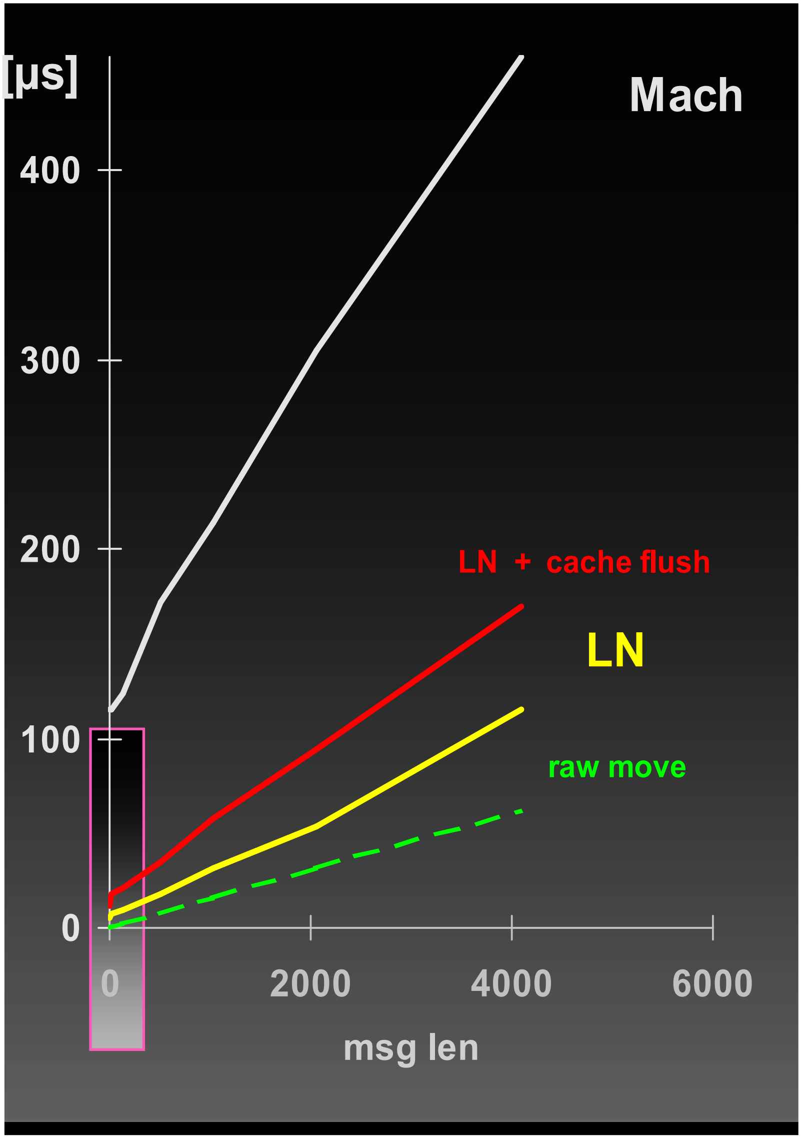
IPC Costs

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

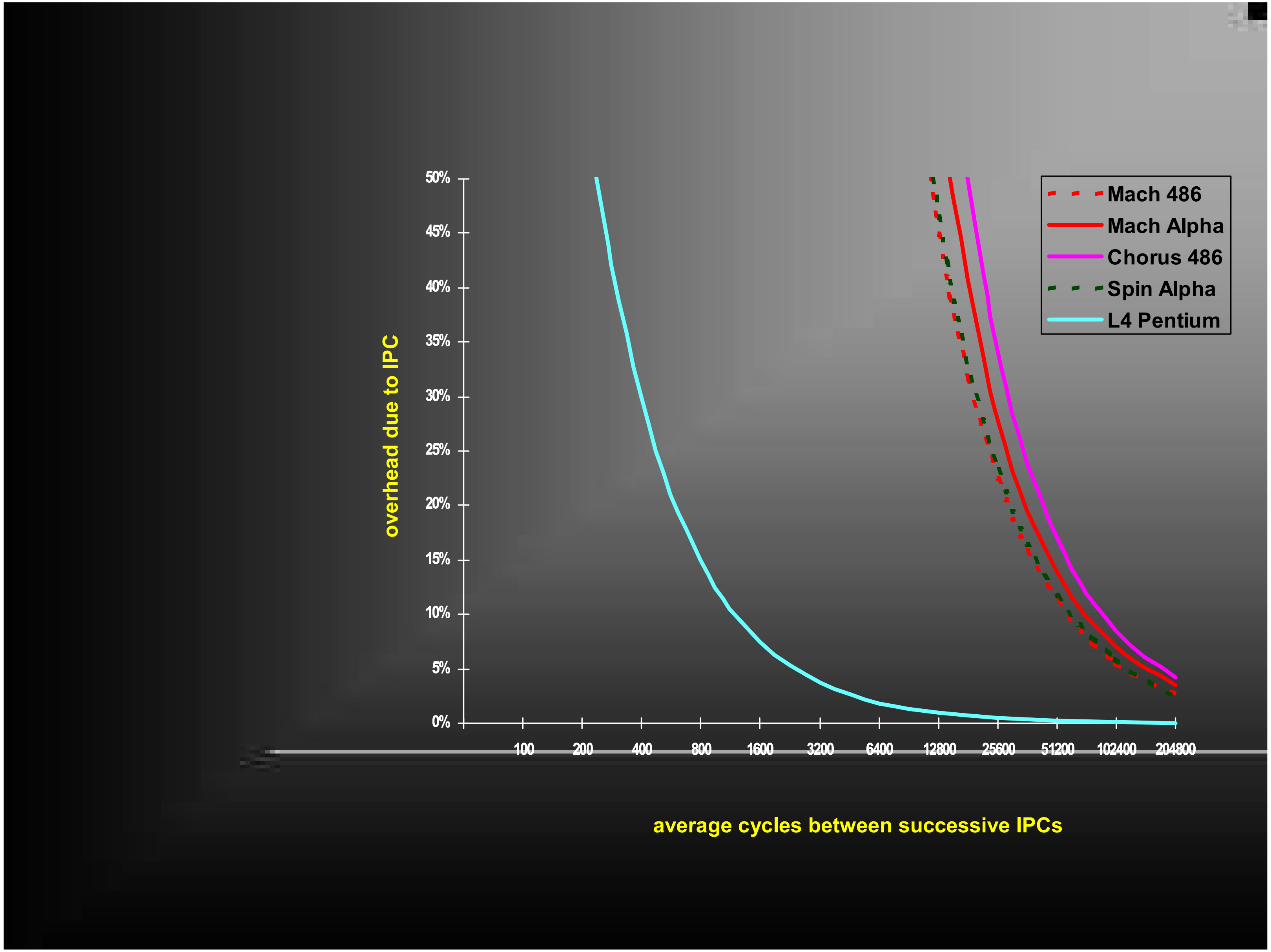
  ... were slow...

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

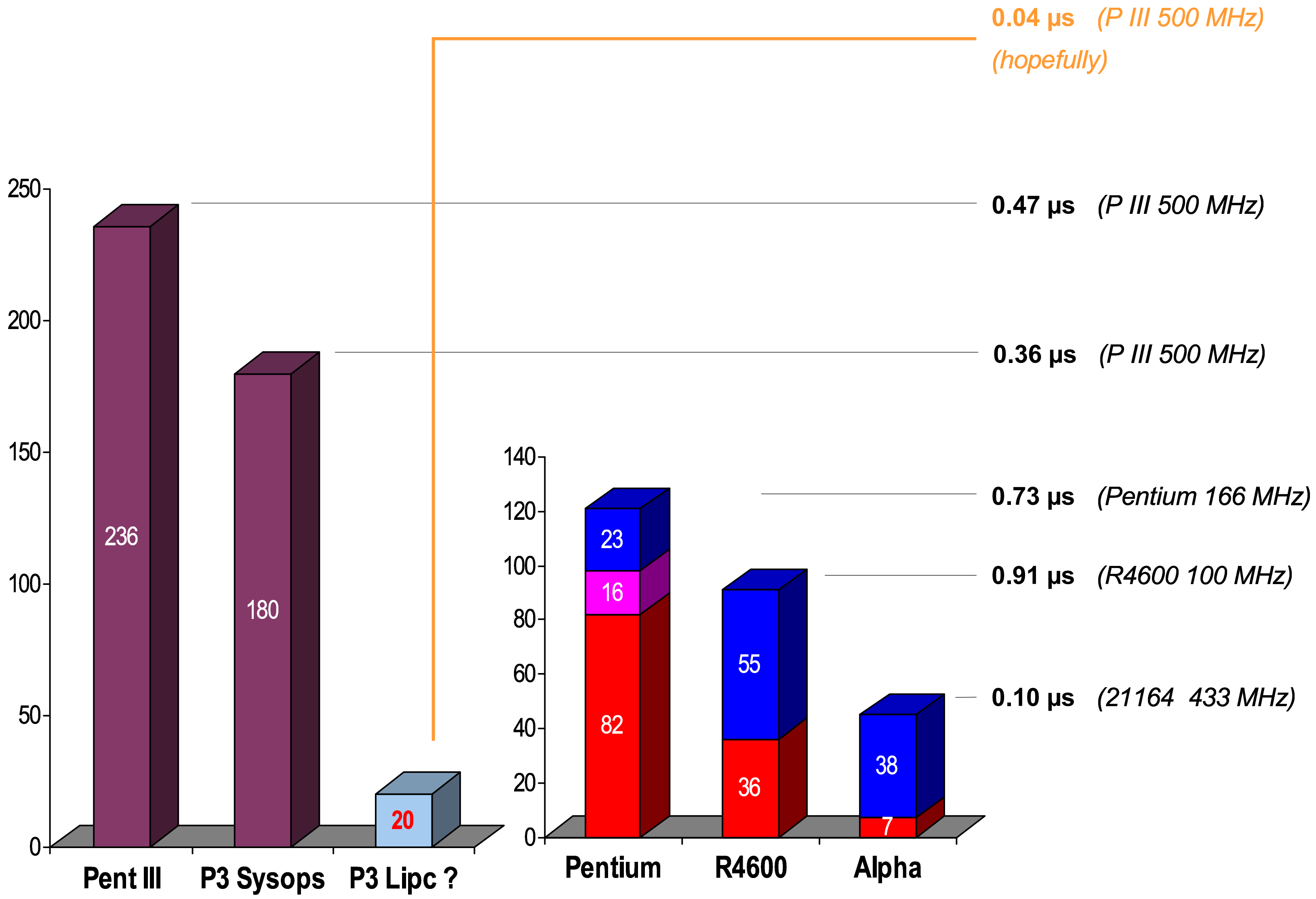
- L4 does better
  - ➜ close to hardware cost
  - ➜ 20 times faster than Mach on identical hardware



IPC Cost Implications



L4 IPC


0.04 µs (P III 500 MHz)
(hopefully)
0.47 µs (P III 500 MHz)
0.36 µs (P III 500 MHz)
0.73 µs (Pentium 166 MHz)
0.91 µs (R4600 100 MHz)
0.10 µs (21164 433 MHz)
250
200
150
100
50
0
236
180
20
Pent III
P3 Sysops
P3 Lipc ?
140
120
100
80
60
40
20
0
23
16
82
55
36
38
7
Pentium
R4600
Alpha

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

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:
  - ➜ $\approx$ 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

L4Ka::Pisatchio: Size

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

L4Ka::Pisatchio: Size

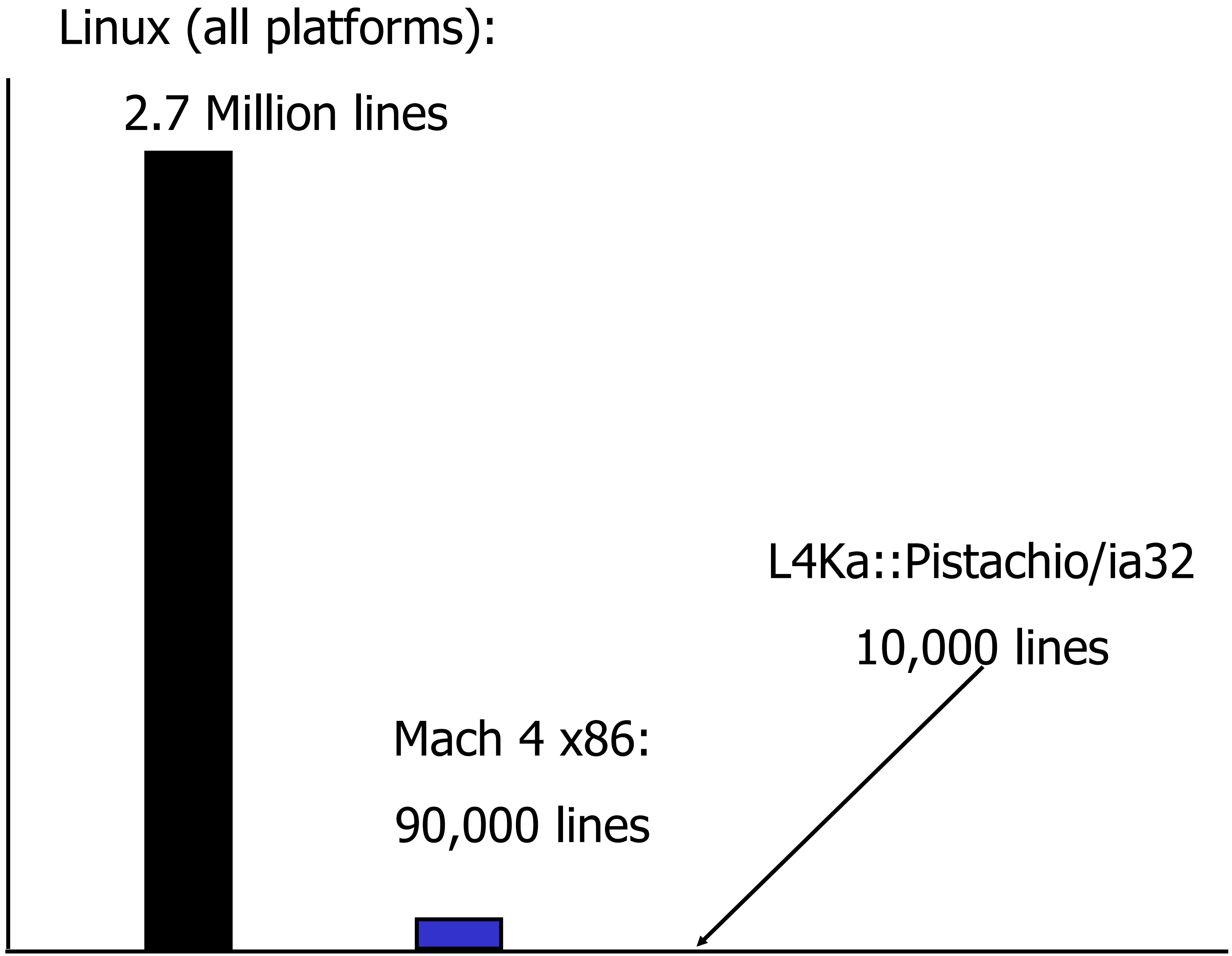
- Source code:
  - ➜ ≈ 10k loc architecture independent
  - ➜ ≈ 0.5–2k loc architecture specific

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

Architecture	Text	Total
x86	52k	98k
Itanium	173k	417k
ARM	68k	180k
PPC-32	41k	135k
PPC-64	60k	205k
MIPS-64	61k	100k

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

SIZE COMPARISON



PISTACHIO PERFORMANCE: IPC

Architecture	port/ optimisation	C++ intra AS	C++ inter AS	optimised intra AS	optimised inter AS
Pentium-3	UKa	180	367	113	305
Small Spaces	UKa				213
Pentium-4	UKa	385	983	196	416
Itanium 2	UKa/NICTA	508	508	36	36
cross CPU	UKa	7419	7410	N/A	N/A
MIPS64	NICTA/UNSW	276	276	109	109
cross CPU	NICTA/UNSW	3238	3238	690	690
PowerPC-64	NICTA/UNSW	330	518	200‡	200‡
Alpha 21264	NICTA/UNSW	440	642	≈70†	≈70†
ARM/XScale	NICTA/UNSW	250	11,400	120–140‡	10,000‡
FASS	NICTA/UNSW	340	340	120–140‡	120–140‡
UltraSPARC	NICTA/UNSW			100‡	100‡

† "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

L4 Abstractions: Address Spaces

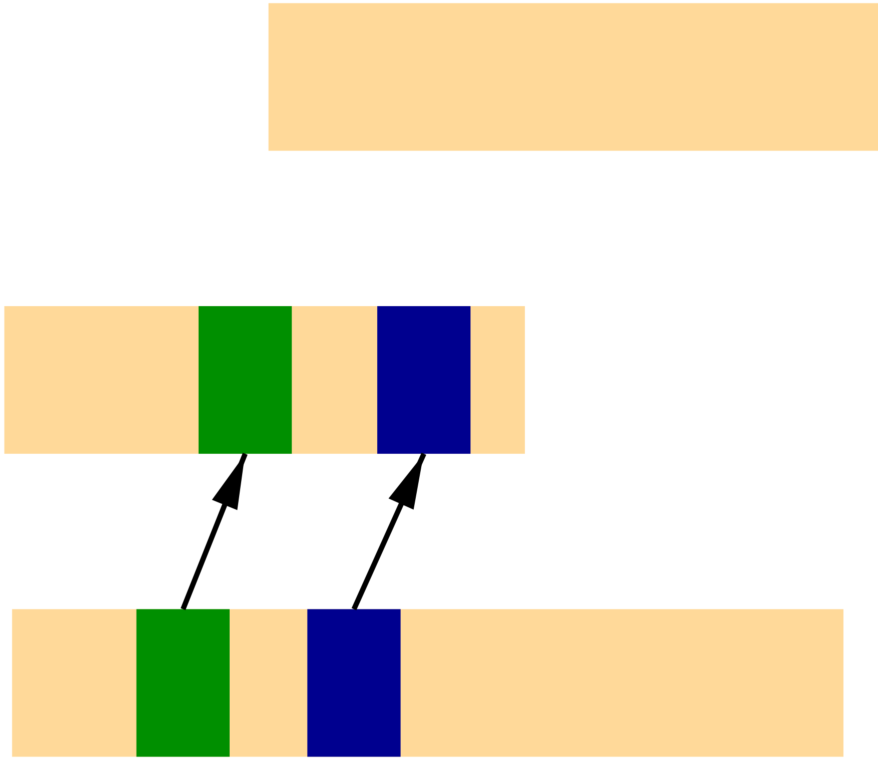
- Address space is unit of protection
  - ➜ initially empty
  - ➜ constructed recursively by mapping pages between address spaces

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* σ_0
    - ➜ σ_0 is an identical map of physical memory

L4 ABSTRACTIONS: ADDRESS SPACES

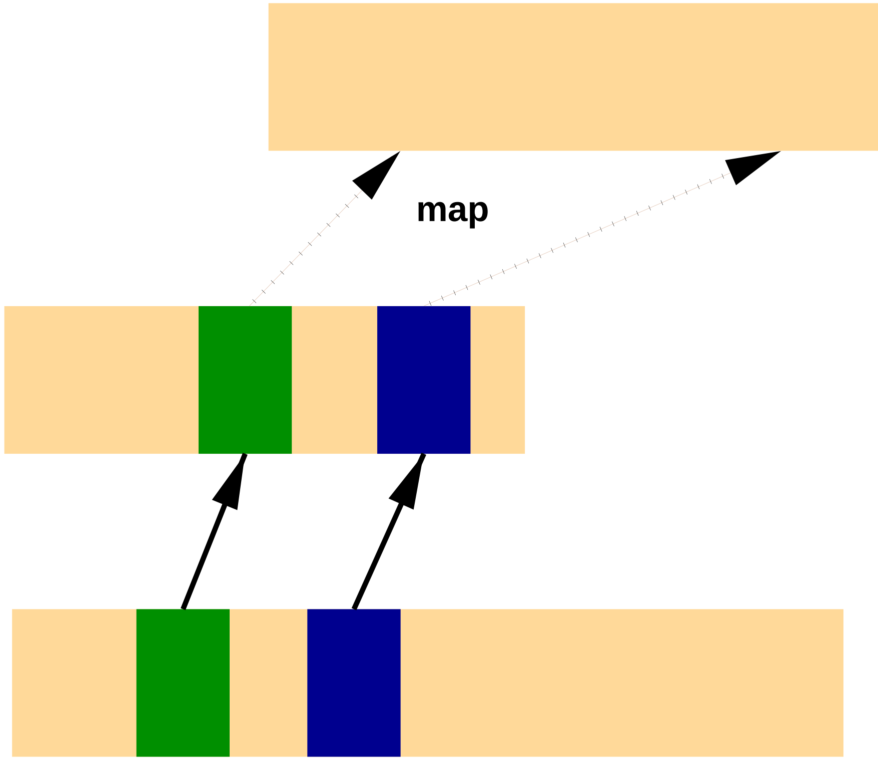
- Address space is unit of protection
  - ➜ initially empty
  - ➜ constructed recursively by mapping pages between address spaces
  - ➜ ... staring from *initial address space* σ_0
    - ➜ σ_0 is an identical map of physical memory
- Mechanism: 3 basic operations:



L4 Abstractions: Address Spaces

- Address space is unit of protection
  - ➜ initially empty
  - ➜ constructed recursively by mapping pages between address spaces
  - ➜ ... staring from *initial address space* σ_0
    - ➜ σ_0 is an identical map of physical memory

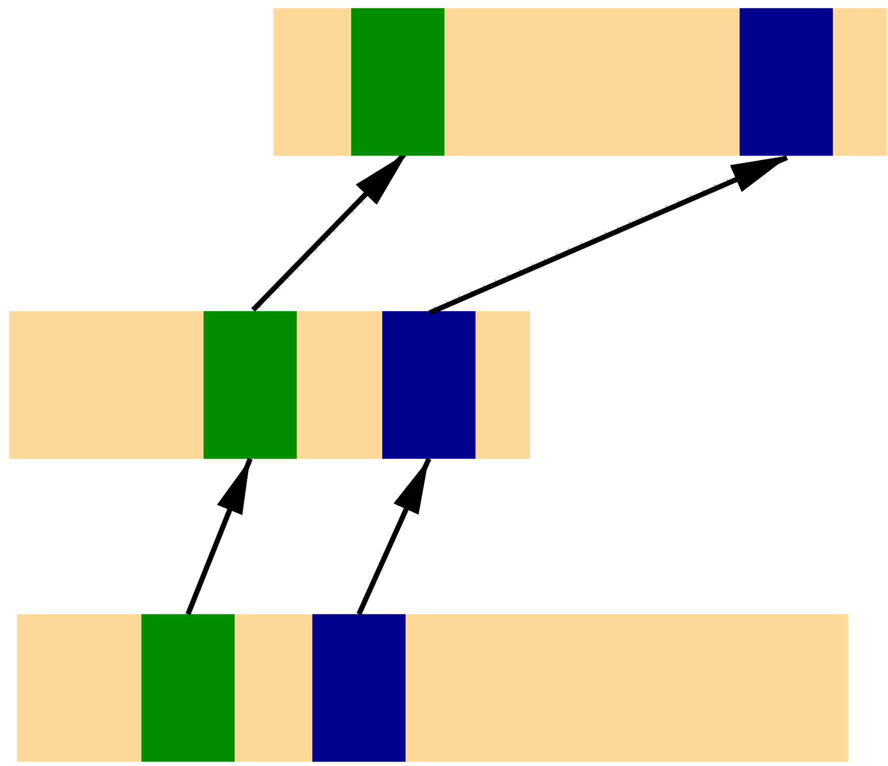
- Mechanism: 3 basic operations:
  - ➜ map



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* σ_0
    - ➜ σ_0 is an identical map of physical memory

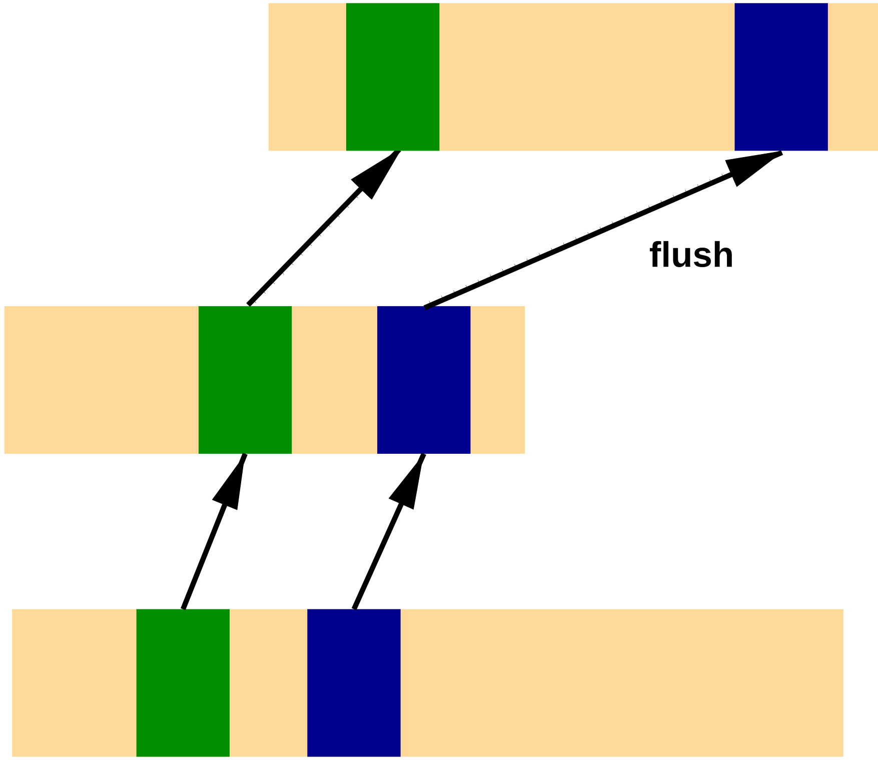
- Mechanism: 3 basic operations:
  - ➜ map



L4 Abstractions: Address Spaces

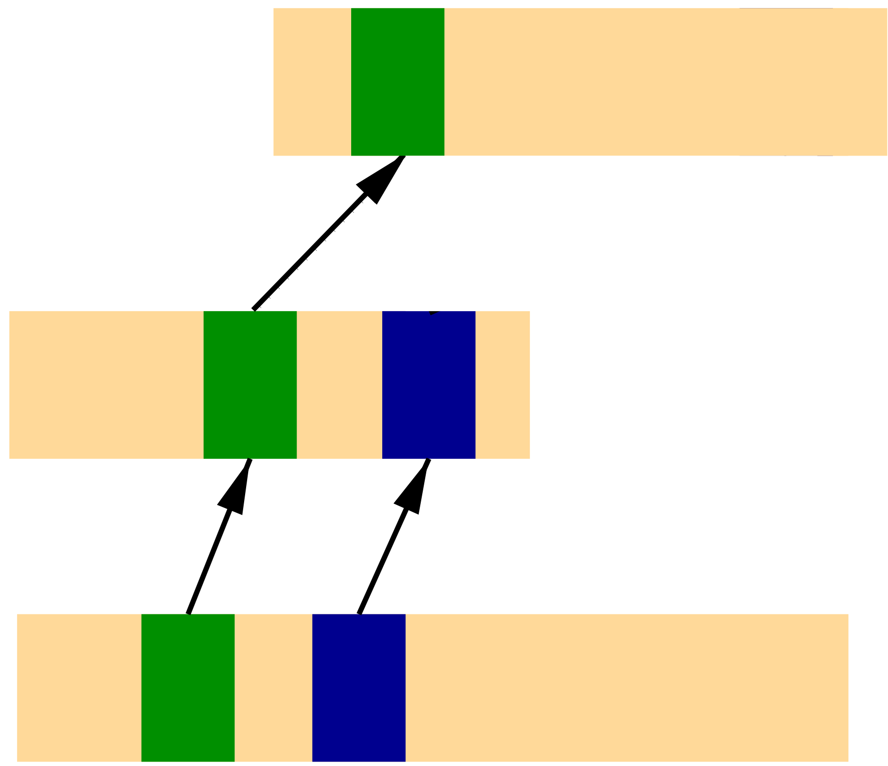
- Address space is unit of protection
  - ➜ initially empty
  - ➜ constructed recursively by mapping pages between address spaces
  - ➜ ... staring from *initial address space* σ_0
    - ➜ σ_0 is an identical map of physical memory

- Mechanism: 3 basic operations:
  - ➜ map
  - ➜ flush (unmap)



L4 Abstractions: Address Spaces

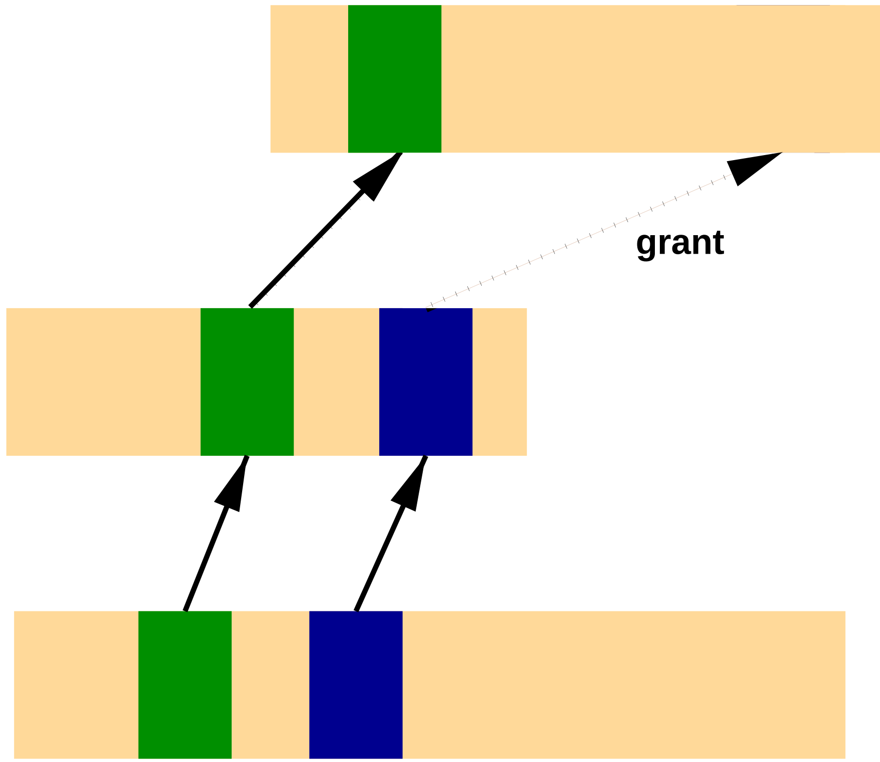
- Address space is unit of protection
  - ➜ initially empty
  - ➜ constructed recursively by mapping pages between address spaces
  - ➜ ... staring from *initial address space* σ_0
    - ➜ σ_0 is an identical map of physical memory
- Mechanism: 3 basic operations:
  - ➜ map
  - ➜ flush (unmap)



L4 Abstractions: Address Spaces

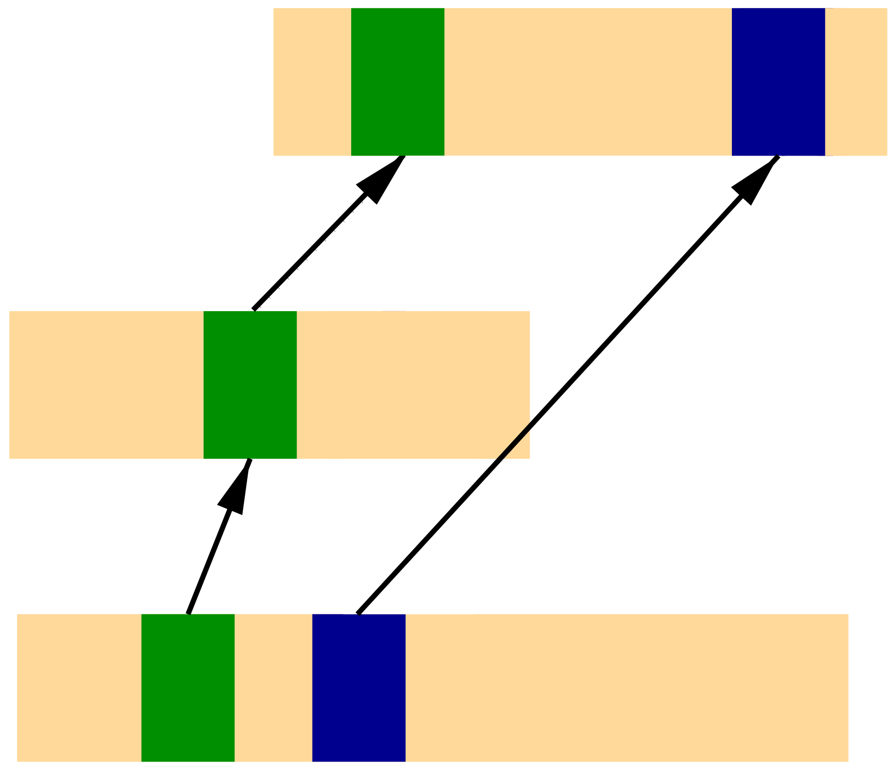
- Address space is unit of protection
  - ➜ initially empty
  - ➜ constructed recursively by mapping pages between address spaces
  - ➜ ... staring from *initial address space* σ_0
    - ➜ σ_0 is an identical map of physical memory

- Mechanism: 3 basic operations:
  - ➜ map
  - ➜ flush (unmap)
  - ➜ grant



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* σ_0
    - ➜ σ_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

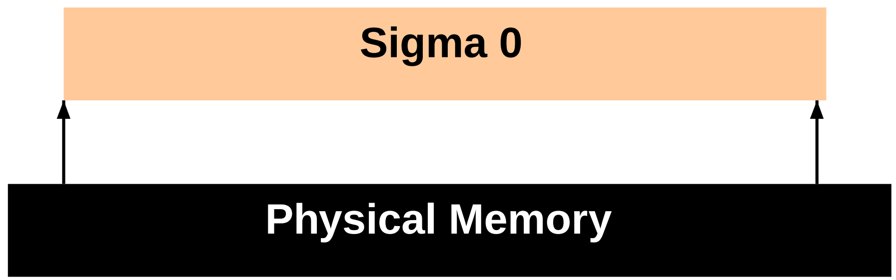
- First task started at boot time
- Can perform *privileged system calls*

L4 Concepts: Root Task

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

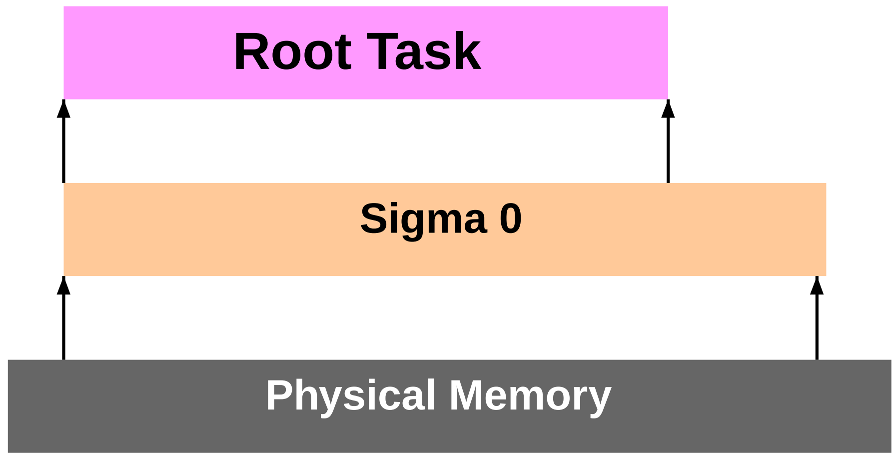
L4 Concepts: Root Task

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



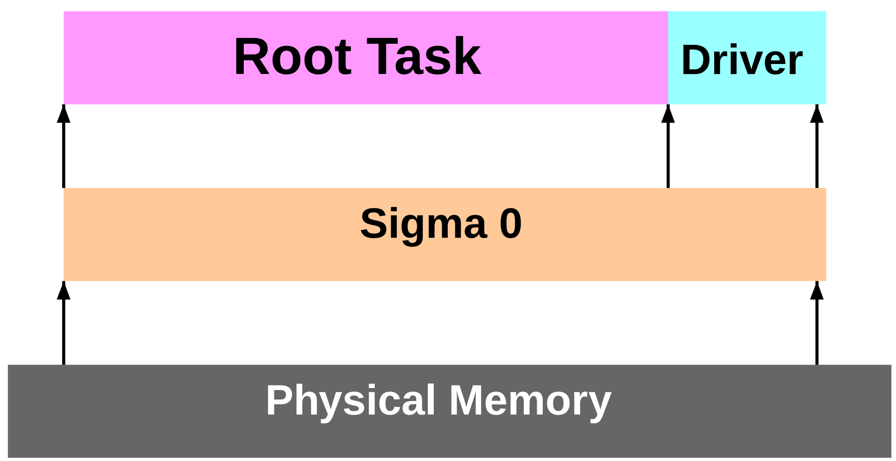
L4 Concepts: Root Task

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



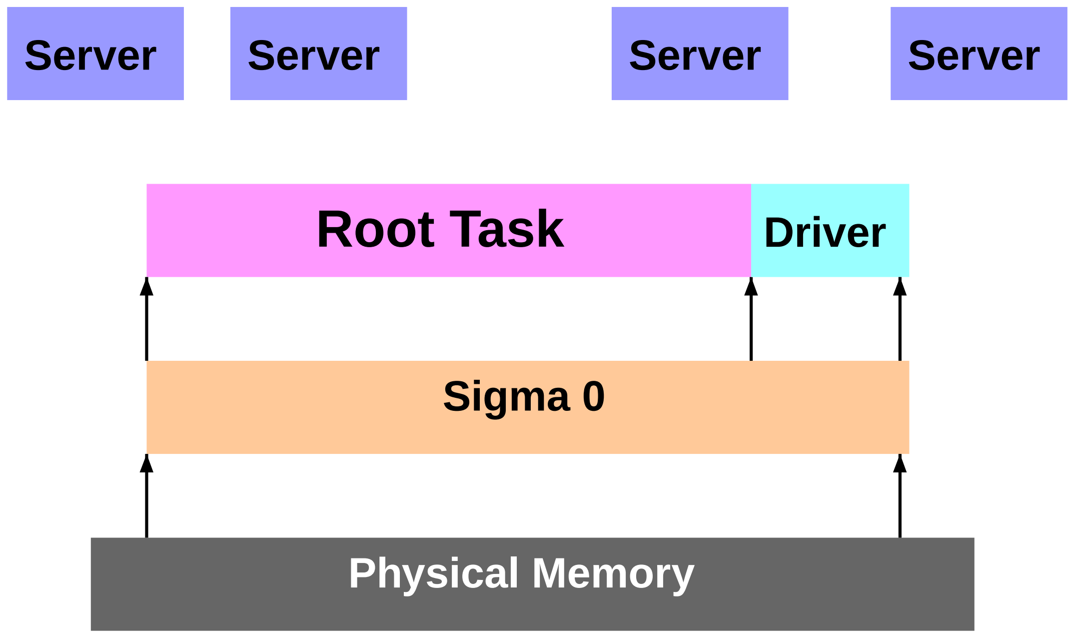
L4 Concepts: Root Task

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



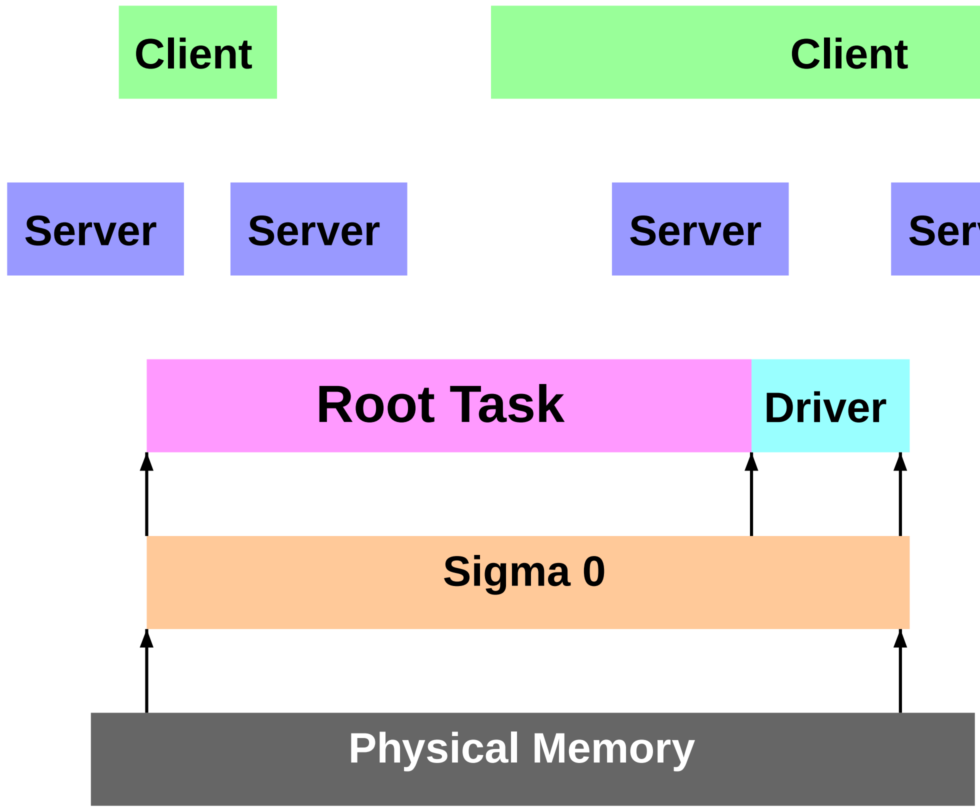
L4 Concepts: Root Task

- 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



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 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

References