

INTRODUCTION

COMP9242 2007/S2 Week 1

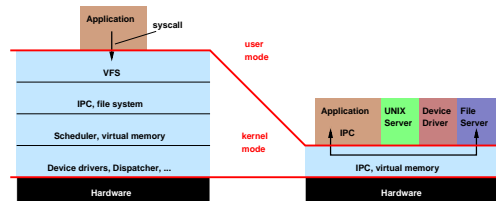
WHY MICROKERNELS?

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

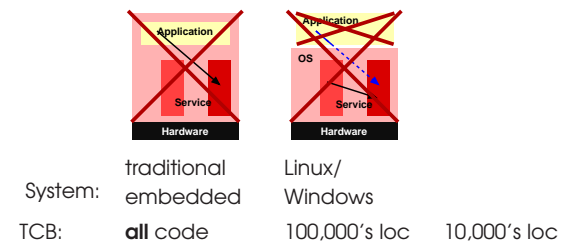
MICROKERNEL: IDEA

- 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



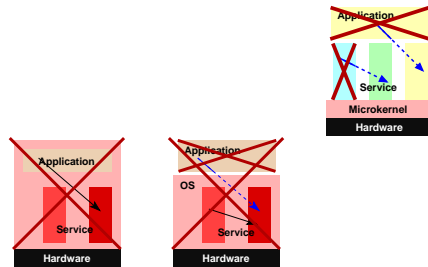
TRUSTED COMPUTING BASE

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	Linux/Windows	Microkernel-based
TCB:	all code	100,000's loc	10,000's loc

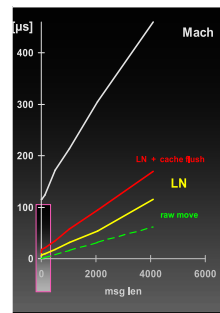
MICROKERNEL PROMISES

- Combat kernel complexity, increase of robustness, maintainability
 - dramatic reduction of amount of privileged code
 - modularisation with hardware-enforced interfaces
 - normal resource management for applications, system services
- Flexibility, adaptability, extensibility
 - policy 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

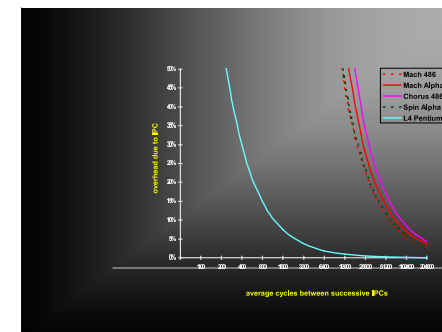
DEATH CHECK!
slow, inflexible
100/sec IPC

IPC COSTS

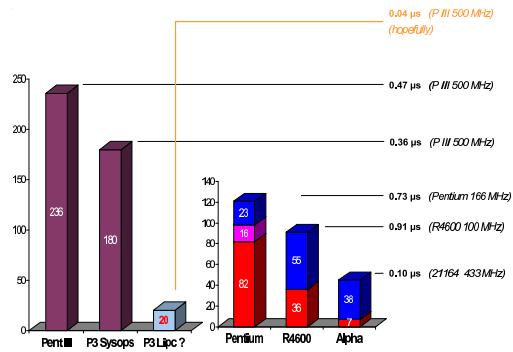
- First-generation microkernels
 - Mach, Chorus, Amoeba
 - ... were slow...
 - 100 μ s IPC
 - almost independent of clock speed!
- L4 does better
 - close to hardware cost
 - 20 times faster than Mach on identical hardware



IPC COST IMPLICATIONS



L4 IPC



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?
 - 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

- Original version by Jochen Liedtke (GMD) ≈ 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
- 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
- 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, 05-06
 - 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
 - You’ll be using the (unreleased) N2 API implementation
- Open Kernel Labs OKL4 API, 06-
 - based on N2
 - active development
 - taking over from NICTA, replacing L4-embedded

L4 PRESENT

- Open Kernel Labs spun out from NICTA
 - on-going commercial L4 development
 - professional services and training for L4 users
 - commercialisation of present NICTA microkernel research
- OKL4 commercially deployed
 - adopted by Qualcomm for CDMA chipsets
 - OKL4-based Toshiba phone on market since ‘06
 - under evaluation/development for other products at a number of multinationals
 - about to establish strong presence in wireless and CE markets

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

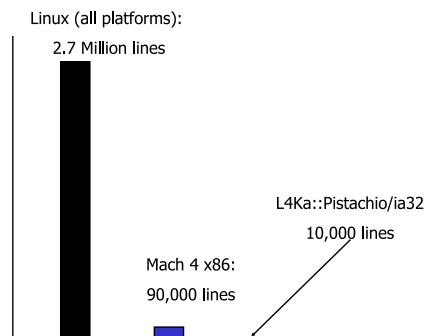
PISTACHIO: SIZE

- Source code:
 - $\approx 10k$ loc architecture independent
 - $\approx 0.5-2k$ loc architecture specific
- Memory footprint kernel (no attempt to minimise yet):

Architecture	Version	Text	Total
x86	L4Ka	52k	98k
Itanium	L4Ka	173k	417k
ARM	NICTA	55k	117k
PPC-32	L4Ka	41k	135k
PPC-64	L4Ka	60k	205k
MIPS-64	L4Ka	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++		optimised	
		intra AS	inter AS	intra AS	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	$\approx 70^\dagger$	$\approx 70^\dagger$
ARM/XScale	NICTA/UNSW	340	340	151	151

[†] "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
 - 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
- 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

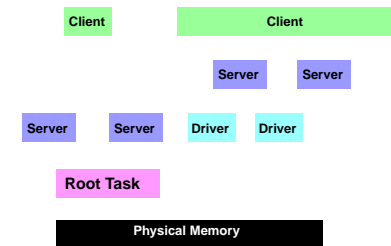
- Used for scheduling time slices
 - thread has fixed-length time slice for preemption
 - time slices allocated from (finite or infinite) time quantum
 - notification when exceeded
- In earlier L4 versions also used for IPC timeouts
 - removed in N2

L4 MECHANISM: IPC

- Synchronous (blocking) 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*
- Controls access to resources
 - threads
 - address spaces
 - physical memory



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