Microkernels in a Bit More Depth

COMP9242
2009/S2 Week 3
Copyright Notice

These slides are distributed under the Creative Commons Attribution 3.0 License

→ You are free:
  • to share — to copy, distribute and transmit the work
  • to remix — to adapt the work

→ Under the following conditions:
  • Attribution. You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:
    • “Courtesy of Gernot Heiser, [Institution]”, where [Institution] is one of “UNSW”, “NICTA”, or “Open Kernel Labs”

→ The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode
Motivation

→ Early operating systems had very little structure
→ A strictly layered approach was promoted by Dijkstra
  • THE Operating System [Dij68]
→ Later OS (more or less) followed that approach (e.g., Unix).
→ Such systems are known as *monolithic kernels*
Issues of Monolithic Kernels

Advantages:
- Kernel has access to everything:
  - all optimisations possible
  - all techniques/mechanisms/concepts implementable
- Kernel can be extended by adding more code, e.g. for:
  - new services
  - support for new hardware

Problems:
- Widening range of services and applications
- OS bigger, more complex, slower, more error prone.
- Need to support same OS on different hardware.
- Like to support various OS environments.
- Distribution
  - Impossible to provide all services from same (local) kernel
Evolution of the Linux Kernel
Approaches to Tackling Complexity

→ Classical software-engineering approach: modularity
  • (Relatively) small, mostly self-contained components
  • Well-defined interfaces between them
  • Enforcement of interfaces
  • Containment of faults to few modules

→ Doesn’t work with monolithic kernels:
  • All kernel code executes in privileged mode
  • Faults aren't contained
  • Interfaces cannot be enforced
  • Performance takes priority over structure
Evolution of the Linux Kernel — Part 2

Software-engineering study of Linux kernel [SJW+02]:

- Looked at size and interdependencies of kernel "modules"
  - "common coupling": interdependency via global variables
- Analysed development over time (linearised version number)
- Result 1: Module size grows linearly with version number
Cross-Module Dependencies ("Spaghettiness")
Evolution of the Linux Kernel — Part 2

Software-engineering study of Linux kernel [SJW+02]:

→ Looked at size and interdependencies of kernel "modules“
  • "common coupling": interdependency via global variables

→ Analyzed development over time (linearised version number)

→ Result 1: Module size grows linearly with version number

→ Result 2: Interdependency grows \textit{exponentially} with version!

→ \textit{The present Linux model is doomed!}

→ There is no reason to believe that others are different
  • e.g. Windows, MacOS, ...

→ Need better software engineering in operating systems!
Monolithic vs. Microkernel OS Structure

Based on the ideas of Brinch Hansen's “Nucleus” [BH70]
Monolithic vs. Microkernel OS Structure

Monolithic OS
- lots of privileged code
- vertical structure
- invoked by system call

Microkernel OS
- little privileged code
- horizontal structure
- invoked by IPC
Microkernel OS

Kernel:
• Contains code which must run in supervisor mode
• Isolates hardware dependence from higher levels
• Is small and fast extensible system
• Provides mechanisms.

User-level servers:
• Are hardware independent/portable
• Provide "OS environment"/"OS personality" (maybe several)
• May be invoked:
  - From application (via message-passing IPC)
  - From kernel (upcalls)
    • Implement policies [BH70].
Downcall vs. Upcall

**Downcall:**
- Unprivileged code enters kernel mode
- Implemented via trap

**Upcall:**
- Privileged code enters user mode
- Implemented via signal/IPC
Microkernel-Based Systems
Hybrid system
- Linux for legacy support or high-level API requirements
- RTOS for legacy support for real-time apps
- Highly componentised system for robustness

Provides migration path from legacy to componentised
Early Example: Hydra

→ Separation of mechanism from policy
  • e.g. protection vs. security
→ No hierarchical layering of kernel
→ Protection, even within OS
  • Uses (segregated) capabilities
→ Objects, encapsulation, units of protection.
→ Unique object name, no concept of object ownership.
→ Object persistence based on reference counting [WCC+74]
Hydra...

→ Can be considered the first *object-oriented OS*
→ Has been called the first microkernel OS
   • by people who ignored Brinch Hansen
→ Has had enormous influence on later OS research
→ Was never widely used even at CMU because of
   • poor performance
   • lack of a complete environment
Popular Example: Mach

- Developed at CMU by Rashid and others [RTY+88] from 1984
- Successor of Accent [FR86] and RIG [Ras88]

Goals:
- Tailorability: support different OS interfaces
- Portability: almost all code H/W independent
- Real-time capability
- Multiprocessor and distribution support
- Security
- Coined term microkernel
Basic Features of Mach Kernel

- Task and thread management
- Interprocess communication
  - asynchronous message-passing
- Memory object management
- System call redirection
  - for virtualization (although they didn't call it that)
- Device support
- Multiprocessor support
Mach Tasks and Threads

→ Thread
  • active entity (basic unit of CPU utilisation)
  • own stack, kernel scheduled
  • may run in parallel on multiprocessor

→ Task
  • consists of one or more threads
  • provides address space and other environment
  • created from "blueprint"
    - Empty or inherited address space
    - Similar approach adopted by Linux clone
  • Activated by creating a thread in it

→ “Privileged user-state program" may control scheduling
Mach IPC: Ports

- Addressing based on ports:
  - port is a mailbox, allocated/destroyed via a system call
  - has a fixed-size message queue associated with it
  - is protected by (segregated) capabilities
  - as exactly one receiver, but possibly many senders
  - can have "send-once" capability to a port
    - for RPC replies (server invocation)

- Can pass the receive capability for a port to another process
  - give up read access to the port

- Kernel detects (and cleans up) ports without senders or receiver

- Processes may have many ports (UNIX server has 2000!)
  - can be grouped into port sets
  - supports listening to many (similar to Unix select)

- Send blocks if queue is full
  - blocking limited by timeout

- Indirection via ports supports transparent distribution
  - Local proxy port forwards message to receiver on remote node
Mach IPC: Messages

→ Segregated capabilities:
  • Threads refer to them via local indices
  • Kernel marshals capabilities in messages
  • Message format must identify caps

→ Message contents
  • Send capability to destination port (mandatory)
    − Used by kernel to validate operation
  • Optional send capability to reply port
    − For use by receiver to send reply
  • Possibly other capabilities
  • “in-line” (by-value) data
  • “out-of-line” (by reference) data, using copy-on-write,
    − May contain whole address spaces
Mach IPC

<table>
<thead>
<tr>
<th>Message Header</th>
<th>Port rights (Capabilities)</th>
<th>Out-of-line data</th>
<th>In-line data</th>
</tr>
</thead>
</table>

Virtual address space
Task 1

Mapping before IPC

IPC

Virtual address space
Task 2

Mapping after IPC

Physical Memory

Message Header | Port rights (Capabilities) | Out-of-line data | In-line data |
|----------------|---------------------------|------------------|-------------|

Virtual address space
Task 1

Mapping before IPC

IPC

Virtual address space
Task 2

Mapping after IPC

Physical Memory
Mach Virtual Memory Management

Address space constructed from memory regions

- Initially empty
- Populated by:
  - explicit allocation
  - explicitly mapping a memory object
  - inheriting from parent
    - by-region inheritance: none, copy, shared
  - allocated automatically by kernel during IPC
    - when passing by-reference parameters
    - kernel determines mapping location
- Leads to sparse virtual memory use (unlike UNIX)
  - uses complex address-map datastructure to limit impact
- Extensive use of copy-on-write for efficiency
  - imposes alignment restrictions
  - not necessarily a win for single pages
Mach Memory Objects

Kernel doesn't support file system

Memory objects are an abstraction of secondary storage:
- can be mapped into virtual memory
- are cached by the kernel in physical memory
- pager invoked if unmapped page is touched (or R/O page written to)
  - invoke file system server to provide data

Support data sharing
- by mapping objects into several address spaces

Mach views virtual memory only as a cache for memory objects
User-Level Page Fault Handlers

→ All actual I/O performed by *pager* — can be
  • default pager (provided by kernel), or
  • *external* pager, running at user level

→ Intrinsic page fault cost: 2 IPCs

(1) Check protection & locate memory object
  • uses address map

(2) Check cache, invoke pager if cache miss
  • uses a hashed page table

(3) Check copy-on-write
  • perform physical copy if write fault

(4) Enter new mapping into H/W page tables
Mach Unix Virtualization

- Emulation library in user address space handles IPC
- Invoked by system call redirection (trampoline mechanism)
  - Supports binary compatibility
  - Example of what's now called para-virtualization
Mach = Microkernel?

→ Most OS services implemented at user level
  • Using memory objects and external pagers
  • Provides mechanisms, not policies
→ Mostly hardware independent
→ Big!
  • 140 system calls (300 in later versions), >100 kLOC
    - Compare: Unix 6th edition had 48 syscalls (10 kLOC without drivers)
  • 200 KiB text size (350 KiB in later versions)
→ Performance poor
  • Tendency to move features into kernel
    - OSF/1
    - Darwin (base of MacOS X): complete BSD kernel inside Mach
→ Further information on Mach: [YTR+87, CDK94, Sin97]
Other Client-Server Systems

→ Lots! Most notable systems:

   **Amoeba**: FU Amsterdam, early 1980's [TM81, TM84, MT86]
   - followed by Minix ('87), Minix 3 ('05)

   **Chorus**: INRIA (France), early 1980's [DA92, RAA+90, RAA+92]
   - Commercialised by Chorus Systèmes in 1988
   - Targeted embedded systems (esp. network infrastructure)
   - Bought by Sun in 1997, closed down in 2002
   - Chorus team spun out to create Jaluna (renamed VirtualLogix in '06)
   - Now market embedded virtualization technology

   **QNX**: “first commercial microkernel” (early '80s)
   - highly successful in automotive and other transport systems

   **Green Hills Integrity**
   - '97 for military, commercial release '02
   - market leader in aerospace, military

   **Windows NT**: Microsoft (early 1990's) [Cus93]
   - Early versions (NT 3) were microkernel-ish
   - Now run main servers and most drivers in kernel mode
Critique of Microkernel Architectures

I'm not interested in making devices look like user-level.
They aren't, they shouldn't, and microkernels are just stupid.

Linus Torvalds

Is Linus right?
Microkernel Performance

First generation microkernel systems ('80s, early '90s)
- Exhibited poor performance when
  - Compared to monolithic UNIX implementations
- Particularly Mach, the best-known example
  - But others weren't better

Typical result: re-kernelise systems
- Move OS services back into the kernel for performance
- Move complete OS personalities into kernel
  - Mach Unix “server” → Unix kernel co-located with Mach
  - Chorus Unix
  - Mac OS X
  - OSF/1....

Some spectacular failures
- most notorious: IBM Workplace OS [Phelan et al. 93]
- also the GNU Hurd
- many others...
IBM Workplace OS (1991–96)

→ Unify IBM's operating systems (and produce cost savings)
  • DOS, OS/2, Posix, AIX, OS/400, Windows (binary compatible)
  • all on same underlying platform, available concurrently
  • apps can use services from multiple OSes
  • “Grand Unification Theory of Operating Systems” (GUTS)

→ Scale across a wide range of environments
  • PDAs (ARM)
  • desktops (x86, PowerPC)
  • massively-parallel machines (Power, ...)

→ Decided to base on Mach
  • “Workplace OS microkernel” derived from Mach 3.0
  • for providing concurrent OS personalities
  • share personality neutral services (PNSs)
IBM Workplace OS

- Significant modifications to Mach to address its problems
  - synchronous IPC, single-copy message-passing
  - direct support for RPC
    - send+receive-reply without user-level capability manipulation
  - migrating threads model
    - thread moves with message during IPC
  - improvements in memory management
    - eg. use mappings for message transfers
  - security tokens that reduce number of rights checks
  - generally simplified and optimised code base
  - more than doubled overall code size
  - improved IPC performance \( \approx 3 \) times (still \( \approx 8 \) times slower than L4)

- Plagued by problems
  - Schedule overruns
  - Budget overruns
  - On-going technical problems
IBM Workplace OS History

→ One of the biggest OS projects ever: US$2G
  • 400 microkernel, 1500 OS/2 programmers
→ Jan '91: Project start
→ Fall '92: Demoed OS/2, DOS and Unix on Mach
→ Fall '93: Announced that Workplace would not replace AIX
→ Jan '95: completely abandoned AIX personality
→ Oct '95: GA release of microkernel for PowerPC
→ Oct '95: Workplace project cancelled, Personal Power Div closed
→ Early '96: shipped last version (2.0) for x86, PowerPC, ARM
→ Considered a prime example of vapourware
  • much marketing before technology was created
IBM Workplace OS Lessons

Analysis by Fleisch, Allan [1998]

➔ Difficulty to map personality services to shared PNSs
  • required extensive restructuring of existing code
  • difficult to get PNS APIs right

➔ Featurism

➔ Focussed on microkernel, too late on personalities

➔ Too much focus on portability of microkernel?

➔ Poor management of huge project
  • eg. wrt shared PSNs

➔ Don't mention microkernel performance as an issue
Microkernel Performance

- Performance problems of Mach became generally known $\approx 93$
- Reasons are investigated by [Chen & Bershad 93]:
  - Instrumented user and system code to collect execution traces
  - Run on DECstation 5000/200 (25MHz R3000)
  - Run under Ultrix and Mach with Unix server
  - Traces fed to memory system simulator
  - Analyse MCPI (memory cycles per instruction)
    - Baseline MCPI (i.e. excluding idle loops)
Ultix vs. Mach-Unix MCPI

- sed+U +M: 0.227, 0.495
- egrep+U +M: 0.015, 0.081
- yacc+U +M: 0.067, 0.129
- gcc+U +M: 0.434, 0.690
- compress+U +M: 0.250, 0.418
- ab+U +M: 0.427, 0.534
- espresso+U +M: 0.041, 0.068
- lisp+U +M: 0.015, 0.038
- eqntott+U +M: 0.154, 0.157
- fpppp+U +M: 0.262, 0.243
- doduc+U +M: 0.338, 0.341
- liv+U +M: 0.117, 0.158
- tomcatv+U +M: 0.674, 0.675

- system i-cache misses
- system d-cache misses
- system wbuffer stalls
- system uncached reads
- user i-cache misses
- user d-cache misses
- user wbuffer stalls

MCPI
Interpretation

→ Observations:
  • Mach memory penalty higher
    − i.e. cache misses or write stalls
  • Mach VM system executes more instructions than Ultrix
    − But has more functionality

→ Claim:
  • Degraded performance is (intrinsic?) result of OS structure
  • IPC cost is not a major factor [Ber92]
    − IPC cost known to be high in Mach
**Assertions**

→ **OS has less instruction & data locality than user code**
  - System code has higher cache and TLB miss rates
  - Particularly bad for instructions

→ **System execution is more dependent on instruction cache behaviour than is user execution**
  - MCPI’s dominated by system i-cache misses
  - Now: most benchmarks were small, i.e. user code fits in cache

→ **Competition between user & system code no problem**
  - Few conflicts between user and system caching
  - TLB misses are not a relevant factor
  - Note: the hardware used has direct-mapped physical caches
    - Split system/user caches wouldn't help
Self-Interference

- Only examine system cache misses
- Shaded: System cache misses removed by associativity
- MCPI for system-only, using R3000 direct-mapped cache
- Reductions due to associativity were obtained by running system on a simulator and using a two-way associative cache of the same size
Assertions

→ 4 Self-interference is a problem in system instruction reference streams.
  • High internal conflicts in system code
  • System would benefit from higher cache associativity

→ 5 System block memory operations are responsible for a large percentage of memory system reference costs
  • Particularly true for I/O system calls

→ 6 Write buffers are less effective for system references.
  • Write buffer allows limited asynchronous writes on cache misses

→ 7 Virtual-to-physical mapping strategy can have significant impact on cache performance
  • Unfortunate mapping may increase conflict misses
  • “Random" mappings (Mach) are to be avoided
Other Experience with Microkernel Performance

→ System call costs are (inherently?) high
  • Typically hundreds of cycles, 900 for Mach/i486

→ Context (address-space) switching costs (inherently?) high
  • Getting worse (in terms of cycles) with increasing CPU/memory speed ratios [Ous90]
  • IPC (involving system calls and context switches) is inherently expensive

→ Microkernels heavily depend on IPC

→ IPC is expensive
  • Is the microkernel idea flawed?
  • Should some code never leave the kernel?
  • Do we have to buy flexibility with performance?
A Critique of the Critique

→ Data presented earlier:
  • Are specific to one (or a few) system,
  • Results cannot be generalised without thorough analysis
  • No such analysis had been done
→ Cannot trust the conclusions [Lie95]
Re-Analysis of Chen & Bershad's Data

MCPI for Ultrix and Mach
Re-Analysis of Chen & Bershad's Data

MCPI caused by cache misses: conflict (black) vs capacity (white)
Conclusion

→ Match system is too big
  • Kernel + UNIX server + emulation library
→ UNIX server is essentially same
→ Emulation library is irrelevant (according to Chan & Bershad)
→ Inevitable conclusion: Mach kernel working set is too big

Can we build microkernels which avoid these problems?
Requirements for Microkernels

- Fast (system call costs, IPC costs)
- Small (almost inevitably big $\Rightarrow$ slow)
- Must be well designed
- Must provide a minimal set of operations

Can this be done?

- Example: kernel call cost on i486
  - Mach kernel call: $\approx$900 cycles
  - Inherent (hardware-dictated cost): 107 cycles
    - $\approx$800 cycles kernel overhead
  - L4 kernel call: 123–180 cycles (15–73 cycles overhead)
  - Obviously, *Mach’s performance is a result of design and implementation*
    - It is not the result of the microkernel concept!
Microkernel Design Principles [Lie96]

- **Minimality:**
  - If it doesn't have to be in the kernel, it shouldn't be in the kernel

- **Appropriate abstractions**
  - which can be made fast and allow efficient implementation of services

- **Well written:**
  - It pays to shave a few cycles off TLB refill handler or the IPC path

- **Unportable:**
  - must be targeted to specific hardware
  - no problem if it's small, and higher layers are portable
  - Example: Liedtke reports significant rewrite of memory management when porting from 486 to Pentium
    - Eg size and associativity of cache, TLB
  - Hardware abstraction layer is too costly

We'll revisit those principles later
What Must a Microkernel Provide?

→ Virtual memory/address spaces
  • required for protection
→ Threads (or equivalent, eg scheduler activations)
  • as execution abstraction
  • for exploiting multiple CPUs
→ Fast IPC
  • the most critical operation
→ Unique identifiers (for IPC addressing)
  • Actually, not true: can use local names
  • Example: shared memory:
    − “physical” identifiers (physical addresses) only known to kernel
    − Mapped into local name space (virtual addresses)
Microkernel Should Not Provide

- File system
  - User-level server (as in Mach)
- Device drivers
  - User-level driver invoked via interrupt (= IPC)
- Page-fault handler
  - Use user-level pager
L4 Implementation Techniques [Liedtke '93]

→ Appropriate system calls to minimise number of kernel invocations
  • e.g. reply & receive next
  • As many syscall args as possible in registers
→ Efficient IPC
  • Rich message structure
  • Value and reference parameters in message
  • Copy message only once (i.e. not user→kernel→user)
→ Fast thread access
  • Thread UIDs (containing thread ID)
  • TCBs in (mapped) VM, cache-friendly layout
  • Separate kernel stack for each thread (fast interrupt handling)
→ General optimisations
  • “hottest” kernel code is shortest
  • Kernel IPC code on single page, critical data on single page
  • Many H/W specific optimisations
## Microkernel Performance [95/97]

<table>
<thead>
<tr>
<th>System</th>
<th>CPU</th>
<th>MHz</th>
<th>RPC [μs]</th>
<th>cyc/IPC</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>L4</td>
<td>MIPS R4600</td>
<td>100</td>
<td>2</td>
<td>100</td>
<td>full</td>
</tr>
<tr>
<td>L4</td>
<td>Alpha 21164</td>
<td>433</td>
<td>0.2</td>
<td>43</td>
<td>full</td>
</tr>
<tr>
<td>L4</td>
<td>Pentium</td>
<td>166</td>
<td>1.5</td>
<td>125</td>
<td>full</td>
</tr>
<tr>
<td>L4</td>
<td>i486</td>
<td>50</td>
<td>10</td>
<td>250</td>
<td>full</td>
</tr>
<tr>
<td>IBM µk</td>
<td>PPC 604</td>
<td>60</td>
<td>14</td>
<td>420</td>
<td>full</td>
</tr>
<tr>
<td>QNX</td>
<td>i486</td>
<td>33</td>
<td>76</td>
<td>1254</td>
<td>full</td>
</tr>
<tr>
<td>Mach</td>
<td>MIPS R2000</td>
<td>16.7</td>
<td>190</td>
<td>1587</td>
<td>full</td>
</tr>
<tr>
<td>Mach</td>
<td>i486</td>
<td>50</td>
<td>230</td>
<td>5750</td>
<td>full</td>
</tr>
<tr>
<td>Amoeba</td>
<td>MC 68020</td>
<td>15</td>
<td>800</td>
<td>6000</td>
<td>full</td>
</tr>
<tr>
<td>Spin</td>
<td>Alpha 21064</td>
<td>133</td>
<td>102</td>
<td>6783</td>
<td>full</td>
</tr>
<tr>
<td>Mach</td>
<td>Alpha 21064</td>
<td>133</td>
<td>104</td>
<td>6916</td>
<td>full</td>
</tr>
<tr>
<td>Exo-tlrpc</td>
<td>MIPS R2000</td>
<td>116.7</td>
<td>6</td>
<td>350</td>
<td>restricted</td>
</tr>
<tr>
<td>Spring</td>
<td>SPARC V8</td>
<td>40</td>
<td>11</td>
<td>220</td>
<td>restricted</td>
</tr>
<tr>
<td>DP-Mach</td>
<td>i486</td>
<td>66</td>
<td>16</td>
<td>528</td>
<td>restricted</td>
</tr>
<tr>
<td>LRPC</td>
<td>CVAX</td>
<td>12.5</td>
<td>157</td>
<td>981</td>
<td>restricted</td>
</tr>
</tbody>
</table>
## L4Ka::Pistachio IPC Performance

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Optimisation</th>
<th>C/C++ Intra AS</th>
<th>C/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>Itanium 2</td>
<td>NICTA</td>
<td>508</td>
<td>508</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>MIPS64</td>
<td>UNSW/NICTA</td>
<td>276</td>
<td>276</td>
<td>109</td>
<td>109</td>
</tr>
<tr>
<td>- inter-CPU</td>
<td>UNSW/NICTA</td>
<td>3238</td>
<td>3238</td>
<td>690</td>
<td>690</td>
</tr>
<tr>
<td>PowerPC-64</td>
<td>UNSW/NICTA</td>
<td>330</td>
<td>518</td>
<td>~200</td>
<td>~200</td>
</tr>
<tr>
<td>Alpha 21264</td>
<td>UNSW/NICTA</td>
<td>440</td>
<td>642</td>
<td>~70</td>
<td>~70</td>
</tr>
<tr>
<td>ARM/XScale</td>
<td>UNSW/NICTA</td>
<td>340</td>
<td>340</td>
<td>151</td>
<td>151</td>
</tr>
</tbody>
</table>
Case in Point: \( L^4 \)Linux [Härtig et al. 97]

- Port of Linux kernel to L4 (like Mach Unix server)
  - Single-threaded (for simplicity, not performance)
  - Is pager of all Linux user processes
  - Maps emulation library and signal-handling code into AS
  - Server AS maps physical memory (& Linux runs within)
  - Copying between user and server done on physical memory
    - Use software lookup of page tables for address translation
- Changes to Linux restricted to architecture-dependent part
- Duplication of page tables (L4 and Linux server)
- Binary compatible to native Linux via trampoline mechanism
  - But also modified libc with RPC stubs
Signal Delivery in L⁴Linux

→ Separate signal-handler thread in each user process
  (1) Server IPCs signal-handler thread
  (2) Handler thread manipulates main user thread to save state
      - Exchange_Registers
  (3) User thread IPCs Linux server
  (4) Server does signal processing
  (5) Server IPCs user thread to resume
L4Linux Performance: Microbenchmarks

getpid():

<table>
<thead>
<tr>
<th>System</th>
<th>Time [µs]</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>1.68</td>
<td>223</td>
</tr>
<tr>
<td>L4Linux (mod libc)</td>
<td>3.95</td>
<td>526</td>
</tr>
<tr>
<td>Li4Linux (trampoline)</td>
<td>5.66</td>
<td>753</td>
</tr>
<tr>
<td>MkLinux in-kernel</td>
<td>15.66</td>
<td>2050</td>
</tr>
<tr>
<td>MkLinux server</td>
<td>110.6</td>
<td>14710</td>
</tr>
</tbody>
</table>

Cycle breakdown:

Hardware cost: 82 cycles (133MHz Pentium)
L4Linux Performance

Microbenchmarks: lmbench

Macrobenchmarks: kernel compile
Conclusions

- Mach sux \[\Rightarrow\] microkernels suck
- L4 shows that performance might be deliverable
  - L^4Linux gets close to monolithic kernel performance
  - Need real multi-server system to evaluate microkernel potential
- Recent work substantially closer to native performance
  - NICTA Wombat, OK Linux (1–2% overhead)
- Microkernel-based systems can perform
- Mach has prejudiced community (see Linus...)
  - Getting microkernels accepted is still uphill battle
Present State

→ Microkernels deployed for years where reliability matters
  • QNX, Integrity
  • Military, aerospace, automotive

→ OKL4 is now being deployed where performance matters
  • Mobile wireless devices
    - Qualcomm chipsets
    - Mobile phones
  • Estimated deployment: 300 million devices (March ’09)
  • First fully-virtualized phone
    - Single-core phone outperforming multicores
Liedtke's Design Principles: What Stands?

→ **Minimality**: definitely

→ **Appropriate abstractions**: yes
  • but no agreement about some of them
  • L4 API still developing
  • NICTA seL4 is most advanced model
    - Integration with commercial OKL4 will set a new standard

→ **Well-written**: absolutely

→ **Unportable**: *no*
  • Pistachio is proof
  • but highly optimised IPC fast path (assembler)
How About His Implementation Techniques?

→ **Appropriate system calls:** *yes*
  - But probably less critical than thought

→ **Efficient IPC, rich message structure:** *less so*
  - OKL4 has abandoned structured messages
  - Passing data in registers beneficial on some architectures
  - single-copy definitely wins
  - Note introduction of asynchronous notification and memcopy syscall in OKL4

→ **Fast thread access:** *no* (at least as propagated by Liedtke)
  - Thread UIDs maybe nice but are a security issue
    - Covert storage channel through global names
    - Segregates caps are the way to go (se OKL4)
  - virtually-mapped linear (sparse) TCB array: *no*
    - Performance impact negligible [Nourai 05]
    - Wastes address space, requires exception handling in kernel (complexity)
  - per-thread kernel stacks: *no*
    - Performance impact negligible [Warton 05]
    - Wastes physical memory (very significant for embedded use)
    - Creates multiprocessor scalability issues