Microkernels
In a Bit More Depth

COMP9242
2007/S2 Week 4
UNSW
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**

(comp9242 07s2 w04)
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

![Graph showing the evolution of the Linux kernel size from 1994 to 2008. The x-axis represents years from 1994 to 2008, and the y-axis represents the size of the Linux kernel in MB. The graph shows a steady increase in kernel size over time.]
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
Microkernel Idea: Break Up the OS
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

- Result 2: Interdependency grows exponentially with version!

- **The present Linux model is doomed!**

- There is no reason to believe that others are different
  - eg 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 (syscall)**
  - Enters kernel mode
  - Implemented via trap

- **Upcall**
  - Enters user mode
  - Implemented via IPC
Microkernel-Based Systems

- **classic +**
  - classic OS
  - Security
  - RT MM
  - L4
  - HW

- **thin**
  - native Java
  - embedded app
  - L4
  - HW

- **specialized**
  - highly-specialized component
  - L4
  - HW
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 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
- 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
  - has exactly one receiver, but possibly many senders
  - can have "send-once" capability to a port
- Can pass the receive capability for a port to another process
  - give up read access to the port
- Kernel detects 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
  - except with send-once cap (used for server replies)
Mach IPC: Messages

• Segregated capabilities:
  – threads refer to them via local indices
  – kernel marshalls 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

Message

| header | port rights (capabilities) | out-of-line data | in-line data |

virtual address space

task 1

virtual address space

task 2

mapping before IPC

mapping after IPC

physical memory

IPC
Mach Virtual Memory Management

Address space constructed from memory regions

- initially empty
- populated by:
  - explicit allocation
  - explicitly mapping a memory object
  - inheriting from "blueprint" (as in Linux clone()),
    - inheritance: not, shared or copied
  - allocated automatically by kernel during IPC
- when passing by-reference parameters
- sparse virtual memory use (unlike UNIX)
Copy-on-Write in Mach

• When data is copied ("blueprint" or passed by-reference):
  – source and destination share single copy,
  – both virtual pages are mapped to the same frame

• Marked as read-only

• When one copy is modified, a fault occurs

• Handling by kernel involves making a physical copy
  – VM mapping is changed to refer to the new copy

• Advantage:
  – efficient way of sharing/passing large amounts of data

• Drawbacks
  – expensive for small amounts of data (page-table manipulations)
  – data must be properly aligned
Mach Address Maps

- Address spaces represented as **address maps**: Any part of AS can be mapped to (part of) a memory object
- Compact representation of **sparse** address spaces
  - Compare to multi-level page tables?
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 uncached page is touched
    • used by file system server to provide data

• Support data sharing
  – by mapping objects into several address spaces

• Memory is only cache for memory objects
User-Level Page Fault Handlers

- All actual I/O performed by \textit{pager}, can be
  - default pager (provided by kernel), or
  - \textit{external} pager, running at user-level.

- Intrinsic page fault cost: 2 IPCs
Handling Page Faults

(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
Remote Communication

• Client A sends message to server B on remote node
  (1) A sends message to local proxy port for B's receive port
  (2) User-level network message server receives from proxy port
  (3) NMS converts proxy port into (global) network port
  (4) NMS sends message to NMS on B's node
      • may need conversion (byte order...)
  (5) Remote NMS converts network port into local port (B's)
  (6) Remote NMS sends message to that port

• Note: networking built into kernel
Mach Unix Emulation

- 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
  – Size: 200k instructions

• 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
    - Bought by Sun a number of years back, closed down later
    - Chorus team spun out to create Jaluna, renamed Virtual Logix
    - Now market embedded virtualization technology
  - **QNX**: “first commercial microkernel” (early '80s)
    - highly successful in automotive
  - **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

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

2 System execution is more dependent on instruction cache behaviour than is user execution
   - MCPIs dominated by system i-cache misses
   Note: most benchmarks were small, i.e. user code fits in cache

3 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
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
MCPI caused by cache misses: conflict (black) vs capacity (white)
Conclusion

• Mach system is too big!
  – kernel + UNIX server + emulation library

• UNIX server is essentially same

• Emulation library is irrelevant (according to Chan & Bershad)

• 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 (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: 900 cycles
  - Inherent (hardware-dictated cost): 107 cycles
    $\Rightarrow$ 800 cycles kernel overhead
  - L4 kernel call: 123-180 cycles (15-73 cycles overhead)
  - Mach's performance is a result of design and implementation
    $\Rightarrow$ 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
    • hardware abstraction layer is too costly
Non-Portability Example: i486 vs. Pentium

• Size and associativity of TLB
• Size and organisation of cache
  – larger line size — restructured IPC
• Segment regs in Pentium used to simulate tagged TLB
  – different trade-offs

With the benefit of hindsight:
• Non-portability is not essential
  – Pistachio is proof
  – >80% architecture-independent code, all C/C++
  – performance rivals that of original x86 assembler kernel
What Must a Microkernel Provide?

• Virtual memory/address spaces
  – required for protection

• Threads (or equivalent, eg scheduler activations)
  – as execution abstraction

• Fast IPC

• Unique identifiers (for IPC addressing)
  – actually, no: can use local names
  – as with shared memory:
    • “physical” identifiers only know to kernel
    • “mapped” into local name space
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]

- Appropriate system calls to minimise # 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

<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>45</td>
<td>full</td>
</tr>
<tr>
<td>L4</td>
<td>Pentium</td>
<td>166</td>
<td>1.5</td>
<td>121</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>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>1584</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>53</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>
## Pistachio IPC Performance

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Optimisation</th>
<th>C/C++ Optimised</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Intra AS</td>
<td>Inter AS</td>
</tr>
<tr>
<td>Pentium-3</td>
<td>UKA</td>
<td>180</td>
</tr>
<tr>
<td>Pentium-4</td>
<td>UKA</td>
<td>385</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>NICTA</td>
<td>508</td>
</tr>
<tr>
<td>MIPS64</td>
<td>UNSW/NICTA</td>
<td>276</td>
</tr>
<tr>
<td>inter-CPU</td>
<td>UNSW/NICTA</td>
<td>3238</td>
</tr>
<tr>
<td>PowerPC-64</td>
<td>UNSW/NICTA</td>
<td>330</td>
</tr>
<tr>
<td>Alpha 21264</td>
<td>UNSW/NICTA</td>
<td>440</td>
</tr>
<tr>
<td>ARM/XScale</td>
<td>UNSW/NICTA</td>
<td>340</td>
</tr>
</tbody>
</table>
Case in Point: L4Linux [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
  - server IPCs signal-handler thread
  - handler thread ex_regs main user thread to save state
  - user thread IPCs Linux server
  - server does signal processing
  - 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>L4Linux (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: Imbench

Macrobenchmarks: kernel compile
Conclusions

• Mach sux △ microkernels suck

• L4 shows that performance might be deliverable
  – L4Linux 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

Microkernel-based systems can perform

• Mach has prejudiced community (see Linus...)
  – getting microkernels accepted is still an uphill battle
Present State

- Microkernels deployed for years where **reliability** matters
  - QNX, Velocity
  - military, aerospace, automotive

- OKL4 is now being deployed where **performance** matters
  - mobile wireless devices
    - Qualcomm chipsets
    - mobile phones
  - estimated deployment: 10s of millions devices (August '07)
  - estimated pipeline: 100s of millions devices in '08
Liedtke's Design Principles: What Stands?

- **Minimality**: definitely
- **Apropriate abstractions**: yes
  - but no agreement about some of them
  - L4 API still developing
- **Well-written**: absolutely
- **Unportable**: *no*
  - Pistachio is proof
  - but highly optimised IPC fast path (assembler)
How About 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

• **Fast thread access**: *no* (at least as propagated by Liedtke)
  - thread UIDs maybe nice but are a security issue
  - virtually-mapped linear (sparse) TCB array: *no*
    • performance impact negligible [Nourai 05]
    • wastes address space
  - per-thread kernel stacks: *no*
    • performance impact negligible [Warton 05]
    • wastes physical memory
    • creates multiprocessor scalability issues