Microkernel-based Operating Systems - Introduction

Carsten Weinhold

Dresden, Oct 10 2017
Lecture Goals

• Provide deeper understanding of OS mechanisms

• Illustrate alternative OS design concepts

• Promote OS research at TU Dresden

• Make you all enthusiastic about OS development in general and microkernels in particular
• Lecture every Tuesday, 4:40 PM, INF/E01

• **Slides:** [http://www.tudos.org -> Teaching -> Microkernel-based Operating Systems](http://www.tudos.org)

• Subscribe to our mailing list: [https://os.inf.tu-dresden.de/mailman/listinfo/mos2017](https://os.inf.tu-dresden.de/mailman/listinfo/mos2017)

• This lecture is **not**: Microkernel construction (in summer term)
Organization: Exercises

- Exercises (roughly) bi-weekly Tuesday, 2:50 PM, INF/E08
- Practical exercises in the computer lab
- Paper reading exercises
  - Read a paper beforehand.
  - Sum it up and prepare 3 questions.
  - We expect you to actively participate in discussion.
- First exercise: next week
  - Practical Exercise: *Booting*
  - Room: to be announced
• Complex lab in parallel to lecture
• Build several components of an OS
• “Komplexpraktikum” for (Media) Computer Science students
• Starts in 2 weeks, 2:50 PM, INF/E08
## Schedule

<table>
<thead>
<tr>
<th>Date</th>
<th>Lecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oct 10</td>
<td>Intro</td>
</tr>
<tr>
<td>Oct 17</td>
<td>IPC</td>
</tr>
<tr>
<td>Oct 24</td>
<td>Threads &amp; Synchronization</td>
</tr>
<tr>
<td>Nov 7</td>
<td>Memory Management</td>
</tr>
<tr>
<td>Nov 14</td>
<td>Real-Time</td>
</tr>
<tr>
<td>Nov 21</td>
<td>Device Drivers</td>
</tr>
<tr>
<td>Nov 28</td>
<td>Resource Management</td>
</tr>
<tr>
<td>Dec 5</td>
<td>Heterogeneous Systems</td>
</tr>
<tr>
<td>Dec 12</td>
<td>Legacy Reuse</td>
</tr>
<tr>
<td>Dec 19</td>
<td>Virtualization</td>
</tr>
<tr>
<td>Jan 9</td>
<td>Secure Systems</td>
</tr>
<tr>
<td>Jan 16</td>
<td>Trusted Computing</td>
</tr>
<tr>
<td>Jan 23</td>
<td>Faults, Failures &amp; Resilience</td>
</tr>
<tr>
<td>Jan 30</td>
<td>Spare slot</td>
</tr>
</tbody>
</table>
Purpose of Operating Systems

• Manage the available resources
  – Hardware (CPU, memory, ...)
  – Software (file systems, networking stack, ...)

• Provide easier-to-use interface to access resources
  – Unix: read/write data from/to sockets instead of fiddling with TCP/IP packets on your own

• Perform privileged / HW-specific operations
  – x86: ring 0 vs. ring 3
  – Device drivers

• Provide separation and collaboration
  – Isolate users / processes from each other
  – Allow cooperation if needed (e.g., sending messages between processes)
Monolithic kernels: Linux

User mode

Kernel mode

Linux Kernel

- System-Call Interface
- File Systems
  - VFS
  - File System Impl.
- Networking
  - Sockets
  - Protocols
- Processes
  - Scheduling
  - IPC
- Memory
  - Management
  - Page allocation
  - Address spaces
  - Swapping
- Device Drivers
- Hardware Access

Hardware

CPU, Memory, PCI, Devices
What's the problem?

- **Security issues**
  - All components run in privileged mode.
  - Direct access to all kernel-level data.
  - Module loading → easy living for rootkits.

- **Resilience issues**
  - Faulty drivers can crash the whole system.
  - 75% of today's OS kernels are drivers.

- **Software-level issues**
  - Complexity is hard to manage.
  - Custom OS for hardware with scarce resources?
The microkernel vision

- Minimal OS kernel
  - less error prone
  - small *Trusted Computing Base*
  - suitable for verification

- System services in user-level *servers*
  - flexible and extensible

- Protection between individual components
  - More resilient – crashing component does not (necessarily...) crash the whole system
  - More secure – inter-component protection
The microkernel vision

User mode
- Application
- Application
- Application
- Application

- File Systems
  - VFS
  - File System Impl.

- Networking
  - Sockets
  - Protocols

- Memory
  - Management
  - Page allocation
  - Swapping

- Device Drivers

Kernel mode
- System-Call Interface
- Hardware Access
- Address Spaces
  - Threads
  - Scheduling
  - IPC

Microkernel

Hardware
- CPU
- Memory
- PCI
- Devices

TU Dresden, 2017-10-10
MOS - Introduction
Slide 11 von 47
What microkernels can give us ...

• OS personalities

• Customizability
  - Servers may be configured to suit the target system (small embedded systems, desktop PCs, SMP systems, ...)
  - Remove unneeded servers

• Enforce reasonable system design
  - Well-defined interfaces between components
  - No access to components besides these interfaces
  - Improved maintainability
The mother of all microkernels

- **Mach** – developed at CMU, 1985 - 1994
  - Rick Rashid (today head of MS Research)
  - Avie Tevanian (former Apple CTO)
  - Brian Bershad (professor @ U. of Washington)
  - ...

- Foundation for several real systems
  - Single Server Unix (BSD4.3 on Mach)
  - MkLinux (OSF)
  - IBM Workplace OS
  - NeXT OS → Mac OS X
Mach: Technical details

- Simple, extensible *communication kernel*
  - “Everything is a pipe.” – *ports* as secure communication channels
- Multiprocessor support
- Message passing by mapping
- Multi-server OS personality
- POSIX-compatibility
- Shortcomings
  - performance
  - drivers still in the kernel
Case study: IBM Workplace OS

- Main goals:
  - multiple OS personalities
  - run on multiple HW architectures

- Win Apps
  - Windows Personality
  - Files
  - Network
- Unix Apps
  - Unix Personality
  - Processes
  - Power
- OS/2 Apps
  - OS/2 Personality
  - OS base services

Mach microkernel

ARM | PPC | x86 | MIPS | Alpha
IBM Workplace OS: Why did it fail?

- Never finished (but spent 1 billion $)
- Failure causes:
  - Underestimated difficulties in creating OS personalities
  - Management errors, forced divisions to adopt new system without having a system
  - “Second System Effect”: too many fancy features
  - Too slow
- Conclusion: Microkernel worked, but system atop the microkernel did not
IBM Workplace OS: Lessons learned

- OS personalities did not work
- Flexibility – but monolithic kernels became flexible, too (Linux kernel modules)
- Better design – but monolithic kernels also improved (restricted symbol access, layered architectures)
- Maintainability – still very complex
- Performance matters a lot
Microkernels: Proven advantages

- Subsystem protection / isolation
- Code size
  - Microkernel-based OS
    - Fiasco kernel: ~ 34,000 LoC
    - “HelloWorld” (+boot loader +root task): ~ 10,000 LoC
  - Linux kernel (3.0.4., x86 architecture):
    - Kernel: ~ 2.5 million LoC
    - +drivers: ~ 5.4 million LoC
      - (generated using David A. Wheeler's 'SLOCCount')
- Customizability
  - Tailored memory management / scheduling / ... algorithms
  - Adaptable to embedded / real-time / secure / ... systems
Challenges

• We need fast and efficient kernels
  – covered in the “Microkernel construction” lecture in the summer term

• We need fast and efficient OS services
  – Memory and resource management
  – Synchronization
  – Device Drivers
  – File systems
  – Communication interfaces
  – Subject of this lecture
Who is (or was) out there?

- Minix @ FU Amsterdam (Andrew Tanenbaum)
- Singularity @ MS Research
- EROS/CoyotOS @ Johns Hopkins University
- The L4 Microkernel Family
  - Originally developed by Jochen Liedtke at IBM and GMD
  - 2nd generation microkernel
  - Several kernel ABI versions
The L4 family – a timeline (or tree ...)

University of New South Wales / NICTA / Open Kernel Labs

L2, L3 v2 x0 x2/v4

L4/x86

L4Ka::Hazelnut L4Ka::Pistachio

NICTA:: Pistachio-embedded OKL4

OKL4v2

SeL4

University of Karlsruhe

Fiasco/L4v2

L4.Sec

Fiasco/L4.Fiasco

TU Dresden

Fiasco

Novo

OC

Fiasco.OC

TU Dresden, 2017-10-10

MOS - Introduction

Slide 21 von 47
L4 concepts

• Jochen Liedtke: “A microkernel does no real work.”
  - Kernel only provides inevitable mechanisms.
  - Kernel does not enforce policies.

• But what is inevitable?
  - Abstractions
    • Threads
    • Address spaces (tasks)
  - Mechanisms
    • Communication
    • Resource mapping
    • (Scheduling)
Taking a closer look at L4

Case study: L4/Fiasco.OC
• “Everything is an object”
  - Task Address spaces
  - Thread Activities, scheduling
  - IPC Gate Communication, resource mapping
  - IRQ Communication
  - Factory Create other objects, enforce resource quotas

• One system call: invoke_object()
  - Parameters passed in UTCB
  - Types of parameters depend on type of object
L4/Fiasco.OC: Types of Objects

- Kernel-provided objects
  - Threads
  - Tasks
  - IRQs
  - ...

- Generic communication object: IPC gate
  - Send message from sender to receiver
  - Used to implement new objects in user-level applications
L4/Fiasco.OC: User-level objects

- Everything above kernel built using user-level objects that provide a service
  - Networking stack
  - File system
  - ...

- Kernel provides
  - Object creation/management
  - Object interaction: Inter-Process Communication (IPC)
To call an object, we need an address:
- Telephone number
- Postal address
- IP address
- ...

Simple idea, right?
ID is wrong? Kernel returns ENOTEXIST
But not so fast! This scheme is insecure:
- Client could simply “guess” IDs brute-force.
- Existence/non-existence can be used as a covert channel
- Global object IDs are
  - insecure (forgery, covert channels).
  - inconvenient (programmer needs to know about partitioning in advance)

- Solution in Fiasco.OC
  - Task-local *capability space* as an indirection
  - *Object capability* required to invoke object

- Per-task name space
  - Maps names to object capabilities.
  - Configured by task's creator
Indirection allows for security and flexibility.
• Capability:
  - Reference to an object
  - Protected by the Fiasco.OC kernel
    • Kernel knows all capability-object mappings.
    • Managed as a per-process capability table.
    • User processes only use indexes into this table.
• Kernel object for communication: **IPC gate**

• Inter-process communication (IPC)
  - Between threads
  - Synchronous

• Communication using IPC gate:
  - Sender thread puts message into its UTCB
  - Sender invokes IPC gate, blocks sender until receiver ready (i.e., waits for message)
  - Kernel copies message to receiver thread's UTCB
  - Both continue, knowing that message has been transferred/received
More L4 concepts
L4/Fiasco.OC: Threads

- **Thread**
  - Unit of Execution
  - Implemented as kernel object

- **Properties managed by the kernel:**
  - Instruction Pointer (EIP)
  - Stack (ESP)
  - Registers
  - User-level TCB

- **User-level applications need to**
  - allocate stack memory
  - provide memory for application binary
  - find entry point
  - ...

Diagram:
- Address Space
  - Code
  - Data
  - UTCBs
  - Stack
L4/Fiasco.OC: Interrupts

- Kernel object: IRQ
- Used for hardware and software interrupts
- Provides asynchronous signaling
  - invoke_object(irq_cap, WAIT)
  - invoke_object(irq_cap, TRIGGER)

Image source: https://commons.wikimedia.org/File:Ethernet_pci_card.jpg
Problem: Memory partitioning

- Physical Memory
- App1
- App2
- App3

4 GB
Solution: Virtual Memory

Physical Memory

App1

App2

App3

4 GB
L4: Recursive address spaces
• If a thread has access to a capability, it can map this capability to another thread

• Mapping / not mapping of capabilities used for implementing access control

• Abstraction for mapping: flexpage

• Flexpages describe mapping
  – location and size of resource
  – receiver's rights (read-only, mappable)
  – type (memory, I/O, communication capability)
L4/Fiasco.OC: Object types

- Summary of object types
  - Task
  - Thread
  - IPC Gate
  - IRQ
  - Factory

- Each task gets initial set of capabilities for some of these objects at startup
Building microkernel-based systems

What can we build with all this?
Kernel vs. Operating System

- **Fiasco.OC** is *not* a full operating system!
  - No device drivers (except UART + timer)
  - No file system / network stack / …

- A microkernel-based OS needs to add these services as user-level components

L4Re - L4 Runtime Environment

---

**User mode**

<table>
<thead>
<tr>
<th>L4Re</th>
<th>User-level libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moe</td>
<td></td>
</tr>
<tr>
<td>Sigma0</td>
<td></td>
</tr>
</tbody>
</table>

**Kernel mode**

- uClibC
- libc++
- ...
Linux on L4

Linux Kernel

- System-Call Interface
  - File Systems (VFS)
  - Networking (Sockets, Protocols)
  - Processes (Scheduling, IPC)
  - Memory Management (Page allocation, Address spaces, Swapping)
  - Device Drivers
  - Hardware Access

L4 Task

Application

L4 Task

Application

L4 Task

Application

L4 Task

User mode

Runtime Environment (L4Re)

Kernel mode

Fiasco.OC

Hardware
The Dresden Real-Time Operating System

Fiasco.OC microkernel

Resource Management Layer (L4Re)

Non-RT World

RT World

User Mode

Privileged Mode

Apps

L4Linux

Time service

Network driver

SCSI driver

Display driver

RT Apps
Virtual machines

- Isolate not only processes, but also complete Operating Systems (compartments)
- “Server consolidation”
Genode

- Genode := C++-based OS framework developed here in Dresden
- Aim: hierarchical system in order to
  - Support resource partitioning
  - Layer security policies on top of each other
• **Basic mechanisms and concepts**
  - Memory management
  - Tasks, Threads, Synchronization
  - Communication

• **Building real systems**
  - What are resources and how to manage them?
  - How to build a secure system?
  - How to build a real-time system?
  - How to reuse existing code (Linux, standard system libraries, device drivers)?
  - How to improve robustness and safety?
Outlook

• Next lecture:
  – “Inter-Process Communication”
  – Next week (Oct 17, 4:40 PM)

• First exercise:
  – Practical Exercise: *Boot*ing
  – Room will be announced on mailing list