Microkernel-based Operating Systems - Introduction

Carsten Weinhold

Dresden, Oct 11 2016
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
Organization: Lecture

- Lecture every Tuesday, 4:40 PM, INF/E01
- Slides: http://www.tudos.org -> Teaching -> Microkernel-based Operating Systems
- Subscribe to our mailing list: http://os.inf.tu-dresden.de/mailman/listinfo/mos2016
- 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
  - Brinch-Hansen: *Nucleus of a multiprogramming system*
More Practical Stuff: Complex lab

- 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
<table>
<thead>
<tr>
<th>Date</th>
<th>Lecture</th>
<th>Exercise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oct 11</td>
<td>Intro</td>
<td></td>
</tr>
<tr>
<td>Oct 18</td>
<td>IPC</td>
<td>Paper: Nucleus of an MP System</td>
</tr>
<tr>
<td>Oct 25</td>
<td>Threads &amp; Synchronization</td>
<td>(Complex Lab)</td>
</tr>
<tr>
<td>Nov 1</td>
<td>Memory Management</td>
<td>Practical: Booting Fiasco/L4Re</td>
</tr>
<tr>
<td>Nov 8</td>
<td>Real-Time</td>
<td></td>
</tr>
<tr>
<td>Nov 15</td>
<td>Device Drivers</td>
<td>Paper: Singularity</td>
</tr>
<tr>
<td>Nov 22</td>
<td>Resource Management</td>
<td>(Complex Lab)</td>
</tr>
<tr>
<td>Nov 29</td>
<td>Heterogeneous Systems</td>
<td>Practical: IPC</td>
</tr>
<tr>
<td>Dec 6</td>
<td>Legacy Containers</td>
<td>(Complex Lab)</td>
</tr>
<tr>
<td>Dec 13</td>
<td>Virtualization</td>
<td>Paper: TBA</td>
</tr>
<tr>
<td>Dec 20</td>
<td>Secure Systems</td>
<td>(Complex Lab)</td>
</tr>
<tr>
<td>Jan 10</td>
<td>Trusted Computing</td>
<td>Practical: L4Linux</td>
</tr>
<tr>
<td>Jan 17</td>
<td>Faults, Failures &amp; Resilience</td>
<td></td>
</tr>
<tr>
<td>Jan 24</td>
<td>L4/Android</td>
<td>Paper: TBA.</td>
</tr>
<tr>
<td>Jan 31</td>
<td>Spare slot</td>
<td>(Complex Lab)</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

The diagram illustrates the components and structure of a monolithic kernel, specifically focusing on Linux. The diagram is divided into two main sections: User mode and Kernel mode.

**User mode** includes:
- Application

**Kernel mode** includes a more detailed breakdown:
- 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 Access** includes:
- 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

File Systems
VFS
File System Impl.

Networking
Sockets
Protocols

Memory
Management
Page allocation
Swapping

Device Drivers

Kernel mode

System-Call Interface

Address Spaces
Threads
Scheduling
IPC

Hardware Access

Microkernel

Hardware
CPU, Memory, PCI, Devices
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

Unix Apps

OS/2 Apps

Windows Personality

Unix Personality

OS/2 Personality

Network

Processes

Power

Files

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
• 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: \( \sim 34,000 \text{ LoC} \)
    - “HelloWorld” (+boot loader +root task): \( \sim 10,000 \text{ LoC} \)
  - Linux kernel (3.0.4., x86 architecture):
    - Kernel: \( \sim 2.5 \text{ million LoC} \)
    - +drivers: \( \sim 5.4 \text{ million LoC} \)
      - (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
  - 2\textsuperscript{nd} generation microkernel
  - Several kernel ABI versions
The L4 family – a timeline (or tree ...)

University of New South Wales / NICTA / Open Kernel Labs

University of Karlsruhe

TU Dresden

TU Dresden, 2016-10-11  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
• 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
L4/Fiasco.OC: Local names for objects

- 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.
L4/Fiasco.OC: Object capabilities

- **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
**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
- ...
• Kernel object: IRQ
• Used for hardware and software interrupts
• Provides asynchronous signaling
  - invoke_object(irq_cap, WAIT)
  - invoke_object(irq_cap, TRIGGER)
Problem: Memory partitioning

Physical Memory

App1

App2

App3

4 GB
Solution: Virtual Memory

App1

App2

App3

Physical Memory

0

4 GB
L4: Recursive address spaces

Diagram showing the relationship between RAM, Physical Address Space, and Device Memory.
• 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
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
Linux on L4

Application
L4 Task

Application
L4 Task

Application
L4 Task

Application
L4 Task

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

L4 Task

User mode

Runtime Environment (L4Re)

Kernel mode

Fiasco.OC

Hardware

TU Dresden, 2016-10-11
MOS - Introduction
Slide 42 von 47
The Dresden Real-Time Operating System

- Privileged Mode
- User Mode

L4Linux

Non-RT World

RT World

Resource Management Layer (L4Re)

Fiasco.OC microkernel

Apps

Time service

RT Apps

Network driver

SCSI driver

Display driver
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
Lecture outline

• 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 18, 4:40 PM)

• First exercise:
  – Per Brinch-Hansen: *The nucleus of a multiprogramming system*
  – Next week (Oct 18, 2:50 PM)
  – **Read the paper! Link is on website!**