The L4Re Microkernel

Adam Lackorzynski

June 2019
• What is L4Re?
• History
• The L4Re Microkernel / Microhypervisor
• Fiasco Interfaces
• Multiprocessor systems
• Virtualization
• …
• L4Re is a microkernel-based operating system framework
• Provides building blocks to build systems
  • Security
  • Safety
  • Real-time
  • Virtualization

• Framework for building „µApps“
  • Libraries and services, libc, pthread, libstdc++, program loading, POSIX subset, shared libraries, memory allocators, ...
• ~1995+: Jochen Liedtke develops L4 microkernel
  • New minimalistic approach; pure assembly; performance
  • Interface: L4-v2
• ~1996+: First application: L4Linux
• ~1997+: Development of Fiasco starts
  • Original L4 kernel restrictive
  • Modern C++-based design
• DROPS project (Dresden Real-time OPerating System)
  • Fiasco is a real-time kernel
• Uni Karlsruhe: L4-x0, L4-x2/v4 interfaces
• L4Env – L4 Environment
  • Environment to run applications on the system
• General shift of focus from real-time to security
• All interfaces (v2 + x2/v4) not suited
  • Global identifiers
• New interface:
  • Capability-based naming
  • Needs redesign of whole user-level framework and applications
• New: L4Re
  • Capability-based run-time environment
  • Kernel/user co-design
  • Uniform & transparent service invocation
• Today: L4Re developed & maintained by Kernkonzept
L4Re Microkernel

• Also called:
  • L4Re Hypervisor, L4Re Microhypervisor
  • Fiasco.OC – Added with capabilities
  • Fiasco

• Continuously developed since 1997
• Started as a single-processor kernel for x86-32
  • Initially on the Pentium-1
• Followed by:
  • ARM, 32bit
  • Itanium IA64
  • PPC32, Sparc
  • MIPS: 32bit + 64bit (r2+r6, LE/BE)
  • ARM, 64bit
L4Re Microkernel – Today

- Capability-based access model
- Multi-processor
- Multi-architecture: ARM, MIPS, x86 (Main architectures)
- 32 & 64 bit
- Generic virtualization approach
  - Paravirtualization
  - Hardware-assisted virtualization: ARM VE, MIPS VZ, Intel VT, AMD SVM
  - For most people it’s a hypervisor
Objects and Capabilities

- Every accessible function is called through an object
  - C++ object, derived from L4::Kobject
  - E.g. Thread, Task, Irq
- Each such object has a pointer
- A capability is such a pointer
  - But protected by the kernel
  - By an indirection through an array in kernel’s address space
  - Invocation is done with an integer indexing into the array
  - Implemented with a sparse array
- Crucial object: Communication channel (between threads)
- One thread listens to messages
- Multiple threads can send to it
Inter-Process Communication – IPC

- Inter-Process Communication (IPC)
  - Actually between threads

- UTCB – „User Thread Control Block“
  - Storage space to exchange data with the kernel
  - Special memory that does not fault
    - Simplifies kernel code

- Capability invocation → sending a message to an object
  - Data is transferred using the UTCB
  - Message transfer is a memcpy() between source and destination UTCBs
• IPC can transfer data and resources
• Resources:
  • Access to memory pages (RAM and IO memory)
  • Access to kernel objects (capabilities)
• Sending access → Granting tasks access to resources

• Data type: Flexpage
  • Flexpages describing resources in the sender’s address space

• Map & Unmap operations
L4 Virtual Memory Model

Linux App

Native L4 App / Driver

L4Linux Kernel

Resource Management

RAM

Device Mem

1  2  3  4
Invoking / Calling the Kernel

Thread

UTCB
MRs
BRs
TCRs

capability

Object

Message

Opcode
Parameters

LABEL
TAG

p  i  w  f
• **TAG** message descriptor
  - Number of words
  - Number of items
  - Flags
  - Protocol ID (type of payload)

• **LABEL** protected message payload
  - Secure identification of a specific capability a message was sent through
UTCB Content

- **MR** message registers
  - Untyped message data
  - Message items (capabilities, memory pages, IO ports)
- **BR** buffer registers
  - Receive buffers for capabilities, memory, IO-ports
  - Absolute timeouts
- **TCR** thread control registers
  - Error code
  - User values
Communication – IPC

Sender Thread

- UTCB
- MRs
- BRs
- TCRs

LABEL
TAG

p i w f

capability

IPC GATE

LABEL

Receiver Thread

- UTCB
- MRs
- BRs
- TCRs

LABEL
TAG

p i w f
• Synchronous
  • Sender waits until the receiver is ready to receive
  • Blocking can be limited by a timeout
• Data-only IPC is atomic
  • Mapping IPC can block (not atomic)
• Atomic send→receive transition in call
  • Reply can have a zero timeout
Asynchronous IRQ Message

Device

User App

HW IRQ
trigger
sys-call

IRQ
LABEL

UTCB
MRs
BRs
TCRs

LABEL
TAG

p i w f

Receiver Thread
Page-fault & Exception Message

Faulting Thread

IPC GATE

UTCB

 RECEIVER THREAD

Faulting Thread

capability

Page-fault message

Fault Address

... p i w f

p i w f
Multi Processor Model

- Supports multi-processor (SMP) on all supported architectures
- Model:
  - Explicit migration by user-level (no policy in the kernel)
  - Address spaces span over cores
  - Transparent cross-core IPC
    - Proxying, programming interface, ...
- User-level provides mechanisms to control migration
Kernel Objects

- L4::Task / L4::VM
- L4::Thread / L4::vCPU
- L4::IPC_gate
- L4::IRQ
- L4::ICU
- L4::PFC
- L4::Vcon
- L4::Scheduler
- L4::Factory
- L4::Debugger
Tasks

- Passive protection domain
  - No threads
- Memory protection (incl. x86 IO-ports)
  - Address space(s)
- Access control (kernel objects, IPC)
  - Object space
  - User-level managed
  - Managed by sparse array
  - Lock free access (using RCU)
Thread

- Executes in a task
  - Access to virtual memory and capabilities of that task
- States: ready, running, blocked
- UT CB
- Active endpoint in synchronous IPC
- Needs scheduling parameters to run
- VCPU mode (later)
• Create (kernel) objects
  • Limited by kernel-memory quota
  • Secondary kernel memory also accounted: page tables, KU memory, FPU state buffers, mapping nodes, ..

• Generic interface
  • User for kernel and user-level objects
• Messages forwarded to a thread
• Including protected label for secure identification
• Fundamental primitive for user-level objects
• Asynchronous signaling
  • Signal forwarded as message to thread
  • No payload
  • Including protected label for identification
  • Fundamental primitive for hardware IRQs and software signaling
• Interrupt controller abstraction
  • Binds an IRQ object to a hardware IRQ pin / source
  • IRQ object gets triggered by hardware interrupt
  • Control parameters of IRQ pin / source

• Generic interface
  • Also used for virtual IRQ sources (triggered by software)
Scheduler

- Manage CPUs and CPU time
  - Bind thread to CPU
  - Control scheduling parameters
  - Gather statistics
- Generic interface
  - Used to define resource management policies
Remaining Kernel objects

- **PFC: Platform control:**
  - Enable/disable CPUs
  - Suspend/resume
  - Reset / Poweroff (depending on platform)

- **Vcon:**
  - Console access

- **Debugger:**
  - Access to in-kernel debugger
• Generic virtualization approach
  • Paravirtualization
    ▪ Making an OS a native L4Re application, e.g. L4Linux
  • Hardware-assisted virtualization
    ▪ Using CPU features: Intel VT-x, AMD SVM, ARM VE, MIPS VZ

• Based on concept of a vCPU
Standard Threads & Virtualization

- Standard threads are synchronous in execution, i.e.:
  - Can execute, OR
  - Can receive messages

- The execution of a CPU is asynchronous, i.e. interrupt driven
  - Can execute, AND
  - Can be ready to receive messages
  i.e. message can be received while code is executed

→ Standard threads are hard to use for executing OS kernels
• A vCPU is a unit of execution
  • It's a thread with more features
  • Every thread can be a vCPU
• Enhanced execution mode: Interrupt-style execution
  – Events (incoming IPCs and exceptions) transition the execution to a user-defined entry point
  – Virtual interrupt flag allows control
    • Behaves like a thread with disabled virtual interrupts
• Virtual user mode
  – A vCPU can temporarily switch a different task (address space)
  – Returns to kernel task for any receiving event
vCPU State Area

- Entry information
  - Entry-point program counter
  - Entry stack pointer
- vCPU state
  - Current mode
  - Exception, page-fault, interrupt acceptance, FPU
vCPU State Area (2)

- State save area
  - Entry cause code
  - Complete CPU register state
  - Saved vCPU state, saved version of the vCPU state
- IPC/IRQ receive state
vCPU – User Processes with Separate L4 Tasks
## Real CPU vs. vCPU

<table>
<thead>
<tr>
<th></th>
<th>Physical CPU</th>
<th>vCPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency control</td>
<td>Interrupt flag</td>
<td>Virtual interrupt flag</td>
</tr>
<tr>
<td>Control flow transfer</td>
<td>Interrupt entry vector(s)</td>
<td>Entry point</td>
</tr>
<tr>
<td>State recovery</td>
<td>Kernel stack or registers by CPU, mostly kept</td>
<td>State-save area</td>
</tr>
<tr>
<td>MMU</td>
<td>Page-tables</td>
<td>Host tasks</td>
</tr>
<tr>
<td>Protection</td>
<td>Modes: Kernel / User</td>
<td>vCPU User / vCPU Kernel</td>
</tr>
</tbody>
</table>
Hardware-Assisted Virtualization

• Requires privileged instructions
  → Implemented in the host kernel (hypervisor) using the vCPU execution model.
• State save area extended to hold
  • x86: VMCB / VMCS (hardware defined data structures, see manuals)
  • ARM / MIPS: kernel-mode state + interrupt controller state
• Gust memory for VM
  • Hardware provides nested paging
  • x86: L4::VM, a specialized L4::Task
  • ARM / MIPS: L4::Task
  • Maps guest physical memory to host memory
vCPU extended mode

- Extended mode: has hardware state
- vCPU-resume implements VM handling
  - Plus sanity checking of provided values
- VMM can run with open and closed vCPU interrupts
  - Open: VMM continues in entry upon VM-exit
  - Closed: VMM continues after resume call upon VM-exit
- Nested paging vs. vTLB
• Type-1 Hypervisor
• Microkernel has virtualization features
  – “Microhypervisor”
• Split functionality:
  – Hypervisor: privileged mode functionality (e.g. VM switching)
  – VMM: user-level program providing the virtual platform for VMs
- Used with hardware-assisted virtualization
- VMM: Virtual Machine Monitor
- Typical model: One VMM per VM
  - Application-specific VMM (simple vs. feature-rich)
  - Multi-VM VMMs possible
- VMM is an **untrusted** user application
VMM Example – uvmm

- uvmm:
  - VMM for ARM and MIPS
  - uvmm for x86 is WiP
  - Small
  - Uses VirtIO for guests
• Feature-full virtulization
• Runs Windows
• x86-based
• Uses L⁴Linux to run KVM & QEMU
• Used in production
• Use C++ to get type-safety in the kernel
• Kernel handles different kind of integer types, e.g.
  • Physical addresses
  • Virtual addresses
  • Page-frame numbers
• Prevent that one can easily convert one into another
  • I.e.: unsigned long phys, virt;
    virt = phys; ← This should not work
• Explicit types:
  • Virt_addr, Phys_addr, V_pfn, Cpu_number, Cpu_phys_id, …
• Provides virtual memory for devices
  → Needs page-table
• Kernel provides mechanisms
• User-level component manages the page-table (task)
• Uses standard mapping/unmapping mechanism
Scheduling

- Each cores run an independent scheduler
- Scheduling type selected at compile time
  - Standard: Fixed-priority round-robin
  - WFQ

- Further topics
  - Scheduling contexts
  - Flattening hierarchical scheduling
  - VMs
  - Budgets / Quotas / QoS
  - …
• Security Appliances
  • (Application) Firewalls, Data-Diodes, ...
  • Up to level GEHEIM / NATO SECRET
• IoT devices
• Automotive
  • New industry-wide paradigm: Consolidate ECUs
  • Central High Performance Controllers
  • Need virtualization etc.
• ... and more...