

The L4Re Operating System and Hypervisor Framework

The L4Re Microkernel

Adam Lackorzynski

June 2024


What is the L4Re Operating System and Hypervisor Framework?

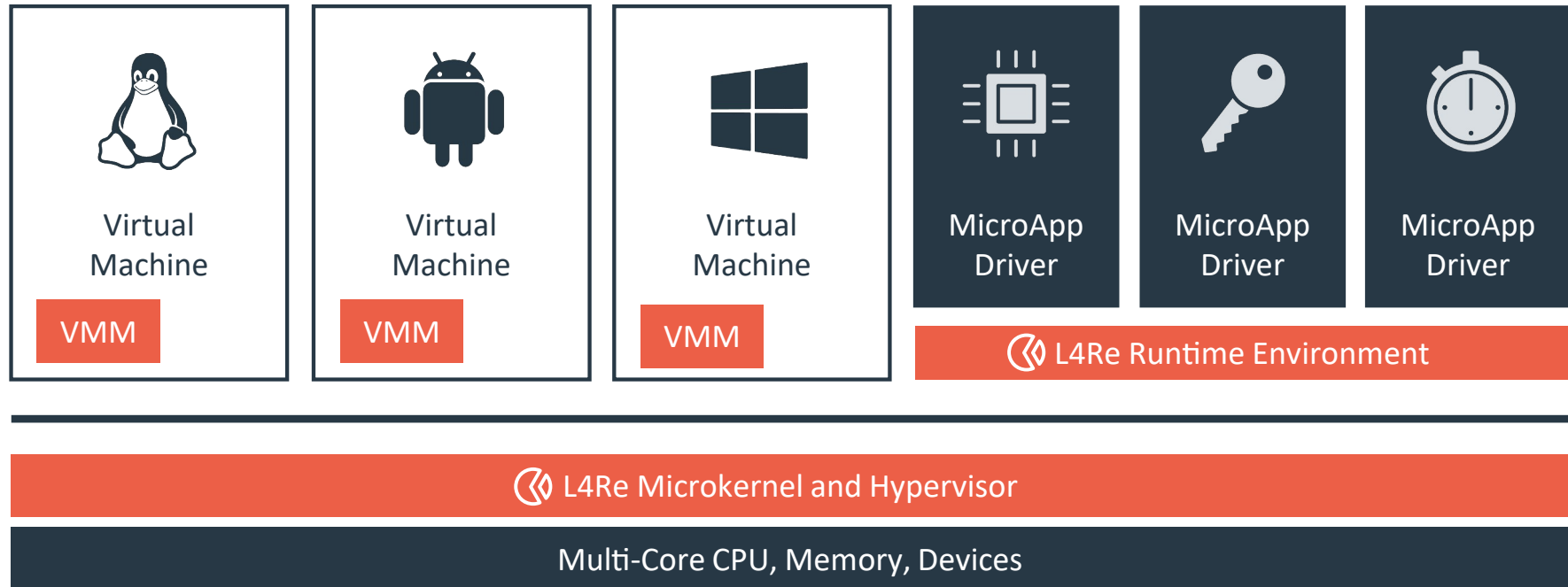
- History
- The L4Re Microkernel and Hypervisor Family
- Interfaces of the Microkernel
- Virtualization
- ...

Overview of the L4Re Microkernel, not the whole L4Re Framework

- L4Re is an open-source microkernel-based Operating System Framework
- Provides building blocks (framework) to build systems, focusing on:
 - Security
 - Safety
 - Real-time
 - Virtualization → Hypervisor
- Framework for building L4Re applications „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 the Fiasco microkernel starts
 - Original L4 kernel has a too restrictive license
 - Modern C++-based design
- DROPS project (**D**resden **R**real-time **O**Perating **S**ystem)
 - 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 open-source project,
maintenance done by  **KERNKONZEPT**



- Also called:
 - Fiasco
 - Old: Fiasco.OC – Redesigned version with added capabilities
- 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, Armv8-A 64bit
 - RISC-V
 - ARM, Armv8-R 32bit + 64bit

- Capability-based access model
- Multi-processor
- Multi-architecture: ARM, MIPS, x86, RISC-V
- 32 & 64 bit
- Generic virtualization approach
 - Para-virtualization
 - Hardware-assisted virtualization: ARM VE, MIPS VZ, Intel VT, AMD SVM, RISC-V H
 - Often approached as a hypervisor
- Continuously open source

- Goals:
 - Security, Safety, Real-Time
- Isolation

- Goals:
 - Security, Safety, Real-Time
- Isolation & defined communication

Implementation

- Goals:
 - Kernel as small as possible
 - Use hardware features for isolation
- Techniques:
 - Microkernel
 - Privilege levels
 - Memory protection
 - Preemption

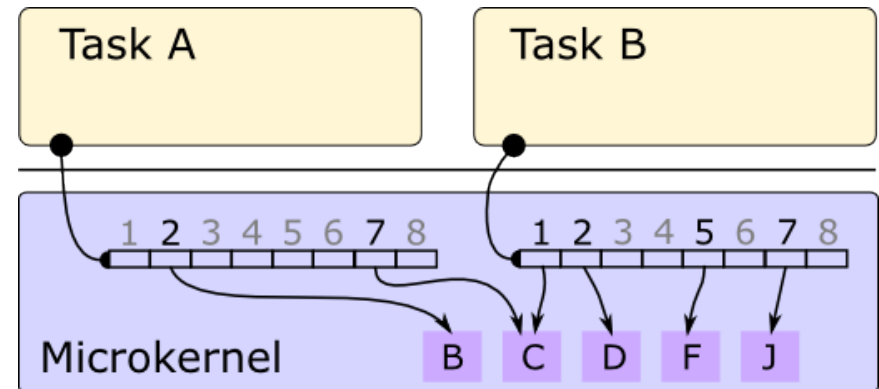
- Microkernel → Kernel as small as reasonably possible
- Privilege levels → Run functionality in user-mode
- Memory protection → Address spaces for spatial isolation
- Preemption → Temporal isolation

- Protection boundary / resource container: Address space (Task)
- Unit of execution: Thread
- Communication: shared memory, data exchange, notification
- Hardware Interrupts
- Low-level platform management
- Scheduling
- Object management
- Debugging

Object	Purpose
L4::Task / L4::Vm / L4::Dma_task	Resource container
L4::Thread	Unit of execution
L4::lpc_gate	Communication channel
L4::Irq	Interrupt
L4::lcu	Set of Interrupts (Interrupt Controller)
L4::Vcon	Textual I/O
L4::Scheduler	Scheduler
L4::Platform_control	Platform Management
L4::Factory	Object creation
L4::Debugger	Debugging facilities

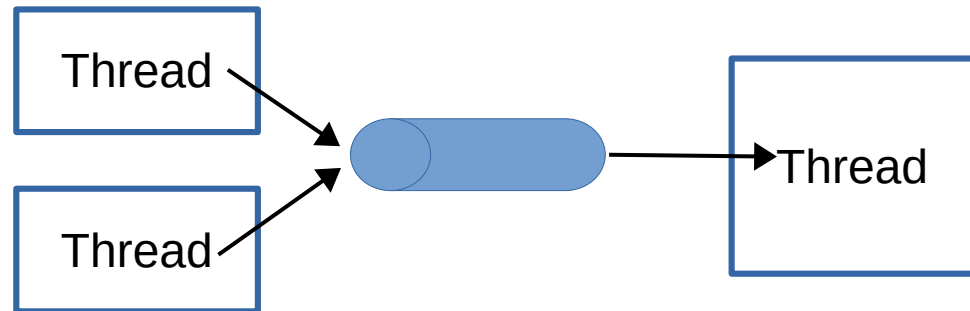
- System call?
 - Yes
- What to invoke?
 - A Capability!

- Every accessible function is called through an object
 - C++ object, derived from base object 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



- It is the same
- Calling an object is like doing an RPC (remote procedure call)
- The kernel has objects
- User-level has objects
 - But how are two user-level threads connected?

- Crucial object: Communication channel (between threads)
- One thread listens to messages
- Multiple threads can send to it



- 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

- Support for 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 trigger thread migration

- 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)
- Variants:
 - Dma_task
 - VM

- Executes in a task
 - Access to (virtual) memory and capabilities of that task
- States: ready, running, blocked
- One UTCB
- Active endpoint in synchronous IPC
- Needs scheduling parameters to run (L4::Scheduler)
- 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)

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

- 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 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
 - Used for
 - Para-virtualising multi-user kernels
 - Switching the vCPU between the VMM and VM

- Entry information
 - Entry-point program counter and stack pointer
- vCPU state
 - Current mode
 - Exception, page-fault, interrupt acceptance, FPU
- State save area
 - Entry cause code
 - Complete CPU register state
 - Saved vCPU state, saved version of the vCPU state
- IPC/IRQ receive state

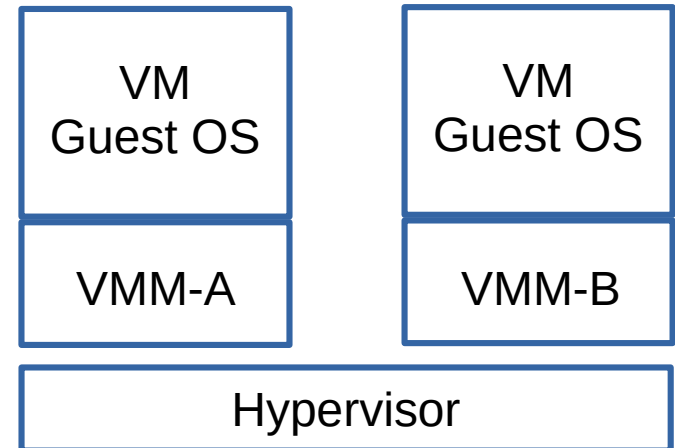
	Real CPU	vCPU
Concurrency Control	Interrupt flag	Virtual interrupt flag
Control flow transfer	Entry vector(s)	Entry point
State recovery	Kernel stack or registers by CPU, mostly kept	State-save area
MMU	Page-tables	Host tasks
Protection	Modes: Kernel / User	vCPU kernel / vCPU user

- 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 / RISC-V: kernel-mode state + interrupt controller state
- Guest memory for VM
 - Hardware provides nested paging
 - x86: L4::VM, a specialized L4::Task
 - ARM / MIPS / RISC-V: L4::Task
 - Maps guest physical memory to host memory

- Extended mode: adds 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

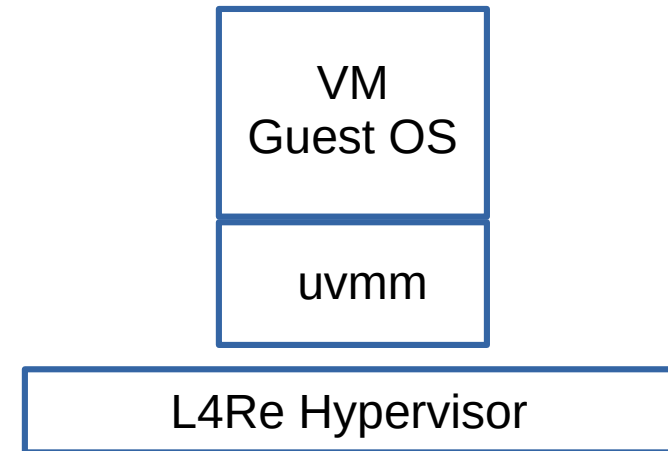
- Microkernel has virtualization features
 - “Hypervisor”
- Split functionality:
 - Hypervisor: privileged mode functionality (e.g. VM switching)
 - VMM: user-level program providing the virtual platform for VMs

- 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

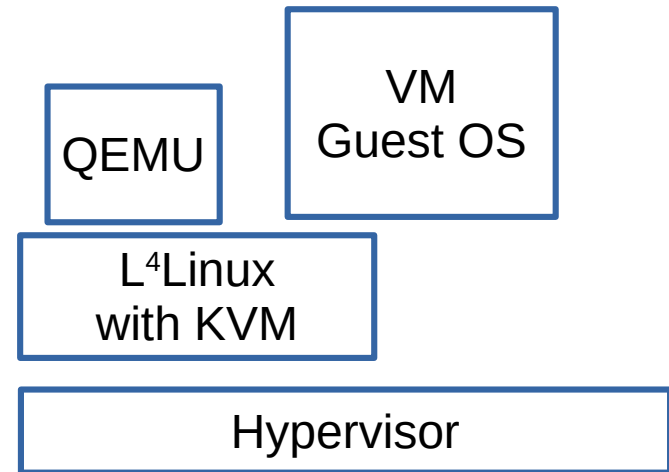


uvmm:

- VMM for ARM, MIPS, RISC-V and x86
- Uses VirtIO for guests



- Feature-full virtualization
- Runs complex guests (e.g., 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

- Each cores run an independent scheduler
- Scheduling type selected at compile time
 - Standard: Fixed-priority round-robin
 - WFQ
- Further topics
 - First-class 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.
- Similar in avionics
- ... and more...

- (Serious) Deployment requires certification
- Security & Safety
 - Security: Common Criteria EAL4+, BSI
 - Safety: ISO 26262, IEC 61508
- Convince others (the assessor) that the system holds the claimed properties:
 - Architecture, documentation of properties
 - Development process
 - Roles
 - Tests, Coverage, ...
 - ...

L4
Re

l4re.org

kernkonzept.com

 KERNKONZEPT

jobs@kernkonzept.com

adam@l4re.org

