



TECHNISCHE  
UNIVERSITÄT  
DRESDEN

Faculty of Computer Science Institute for System Architecture, Operating Systems Group

# Microkernel-based Operating Systems - Introduction

Nils Asmussen

Dresden, Oct 15 2019

- 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

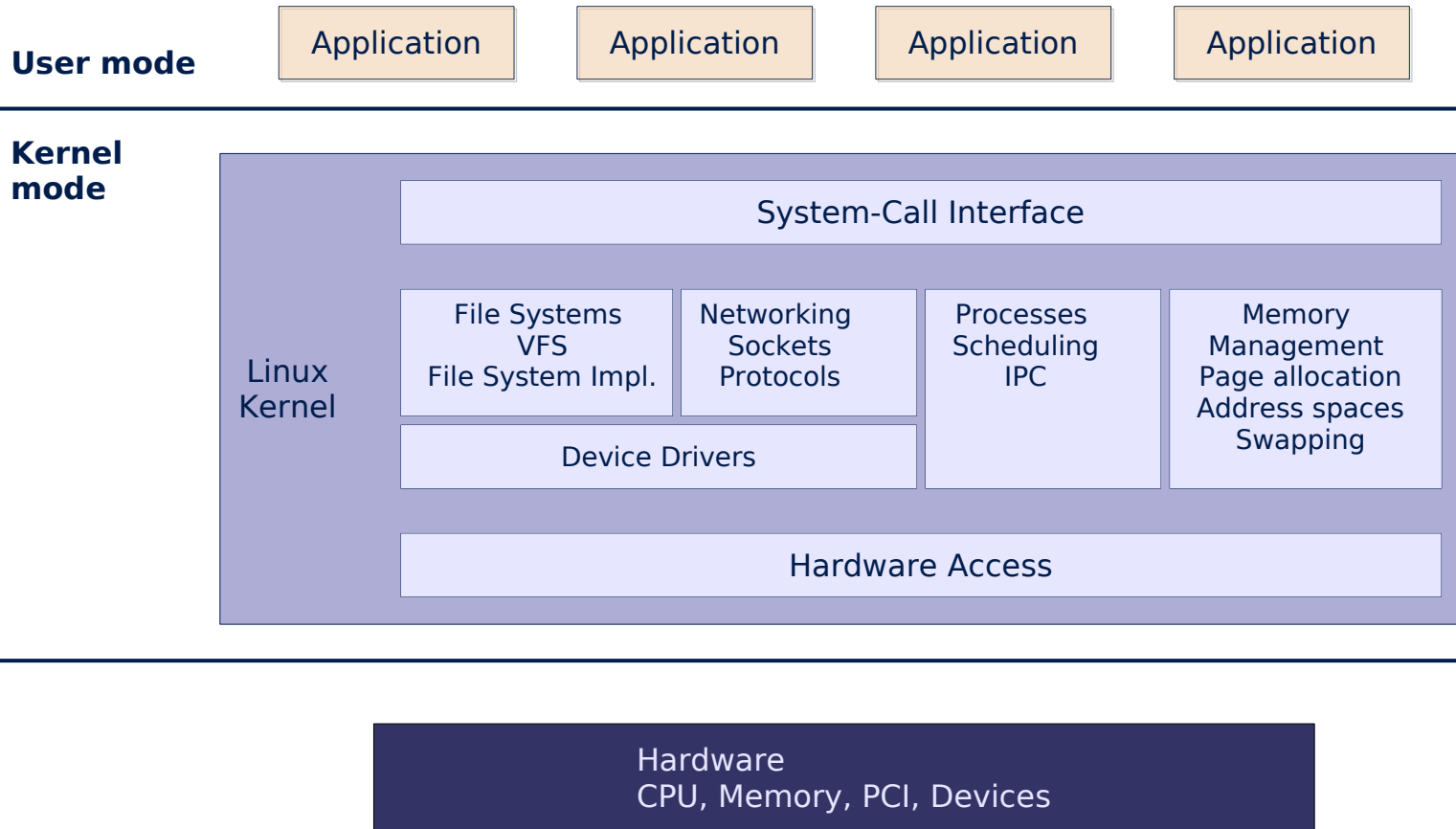
- Lecture every Tuesday, 4:40 PM, INF/E01
- Slides: <http://www.tudos.org> -> Teaching -> Microkernel-based Operating Systems
- Subscribe to our mailing list:  
<http://os.inf.tu-dresden.de/mailman/listinfo/mos2019>
- This lecture is not: Microkernel construction (in summer term)

- 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 on Nov 12<sup>th</sup>, 2:50 PM, INF/E08

- 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



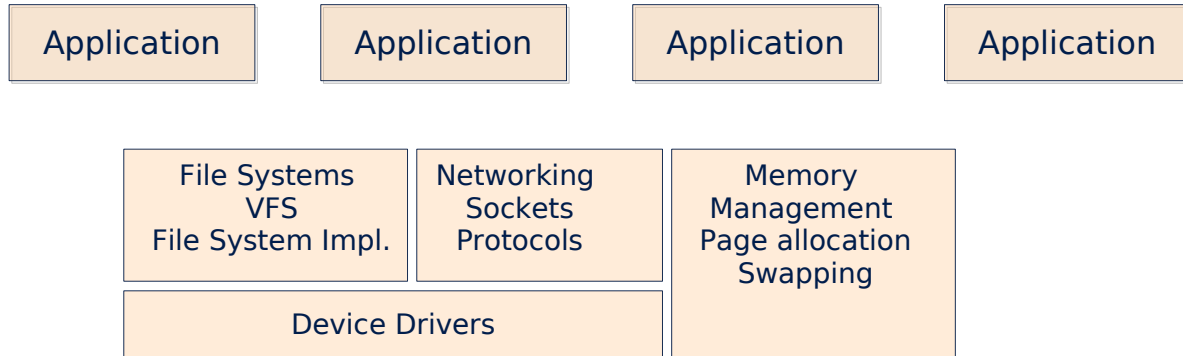
- Security issues
  - All components run in privileged mode.
  - Direct access to all kernel-level data.
  - Module loading → easy living for rootkits.
- Resilience issues
  - Faulty drivers can crash the whole system.
  - 75% of today's OS kernels are drivers.
- Software-level issues
  - Complexity is hard to manage.
  - Custom OS for hardware with scarce resources?



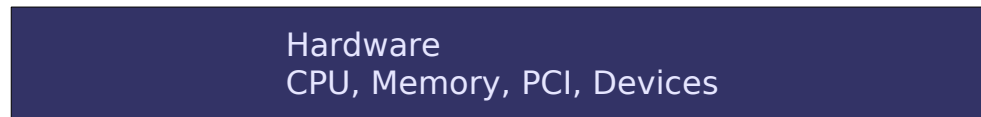
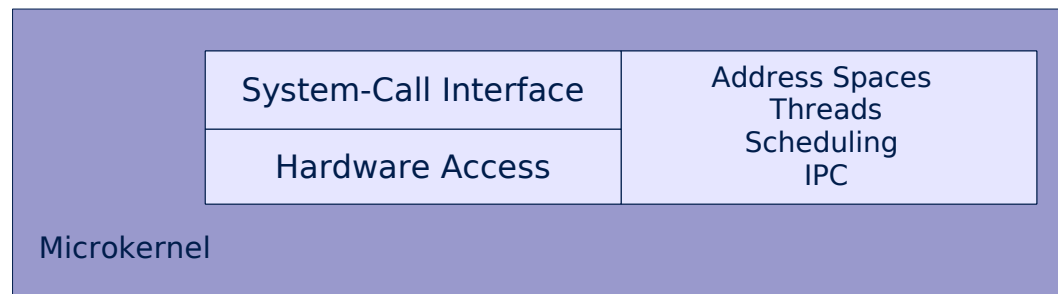
- Minimal OS kernel
  - less error prone
  - small *Trusted Computing Base*
  - suitable for verification
- System services in user-level *servers*
  - flexible and extensible
- Protection between individual components
  - More resilient – crashing component does not (necessarily...) crash the whole system
  - More secure – inter-component protection

# The microkernel vision

**User mode**



**Kernel mode**

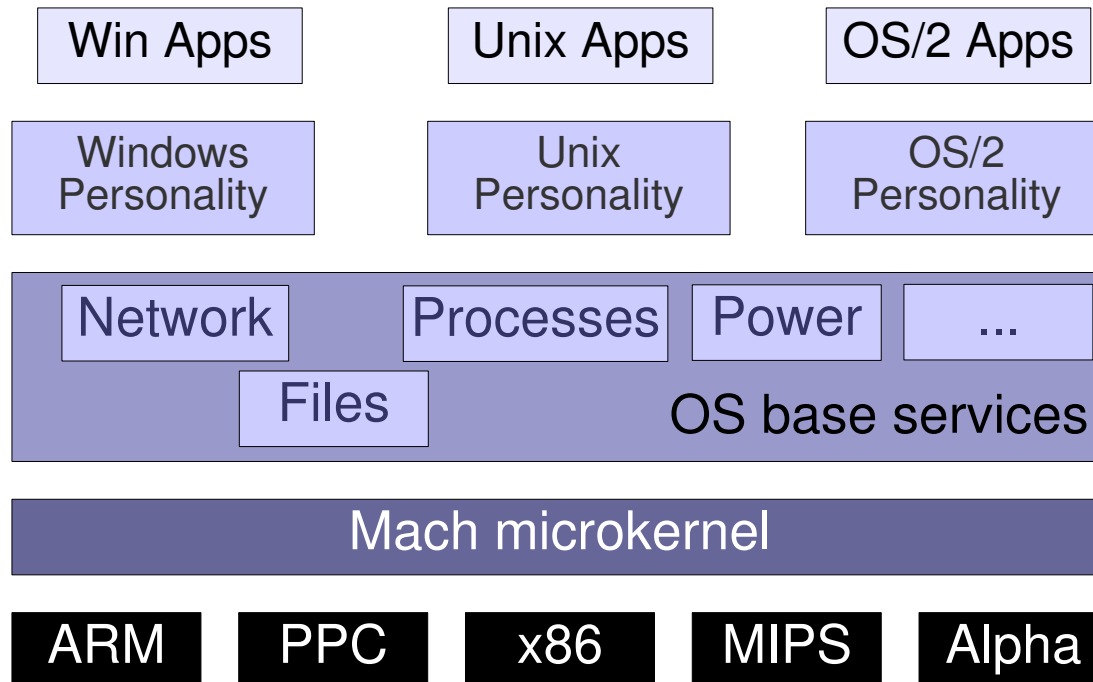


- OS personalities
- Customizability
  - Servers may be configured to suit the target system (small embedded systems, desktop PCs, SMP systems, ...)
  - Remove unneeded servers
- Enforce reasonable system design
  - Well-defined interfaces between components
  - No access to components besides these interfaces
  - Improved maintainability

- Mach – developed at CMU, 1985 - 1994
  - Rick Rashid (former 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

- 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

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



- 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

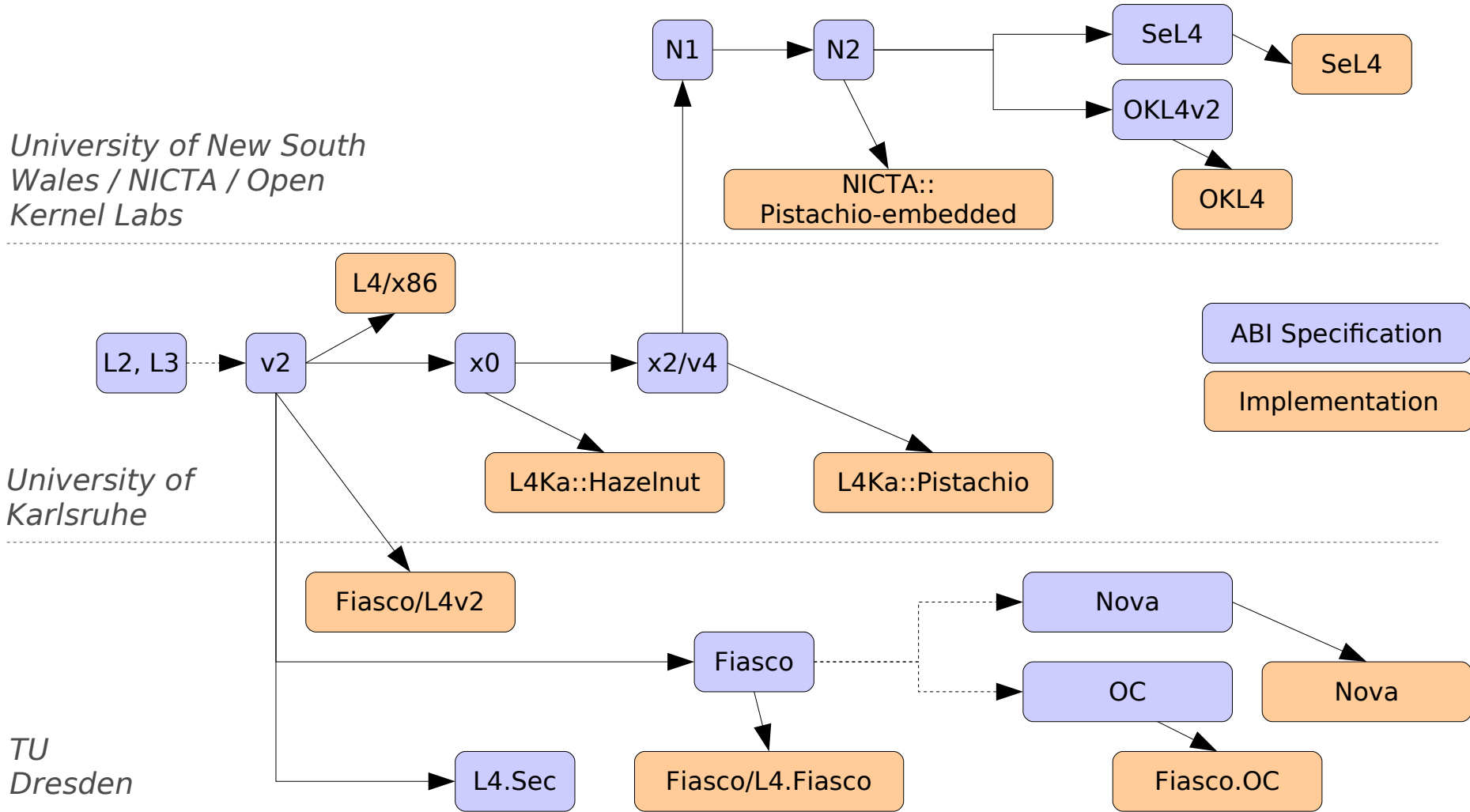


- Subsystem protection / isolation
- Code size
  - Microkernel-based OS
    - Fiasco kernel: ~ 34,000 LoC
    - “HelloWorld” (+boot loader +root task): ~ 10,000 LoC
  - Linux kernel (3.0.4., x86 architecture):
    - Kernel: ~ 2.5 million LoC
    - +drivers: ~ 5.4 million LoC
  - (*generated using David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

- 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

- 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<sup>nd</sup> generation microkernel
  - Several kernel ABI versions

# The L4 family - a timeline (or tree ...)



- 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

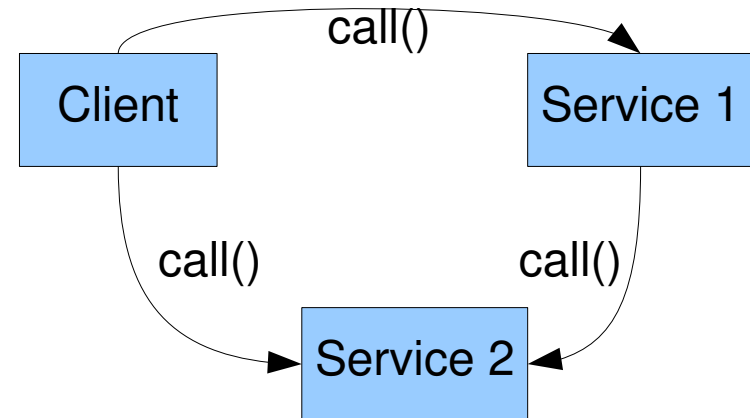
## Case study: **L4/Fiasco.OC**

- “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

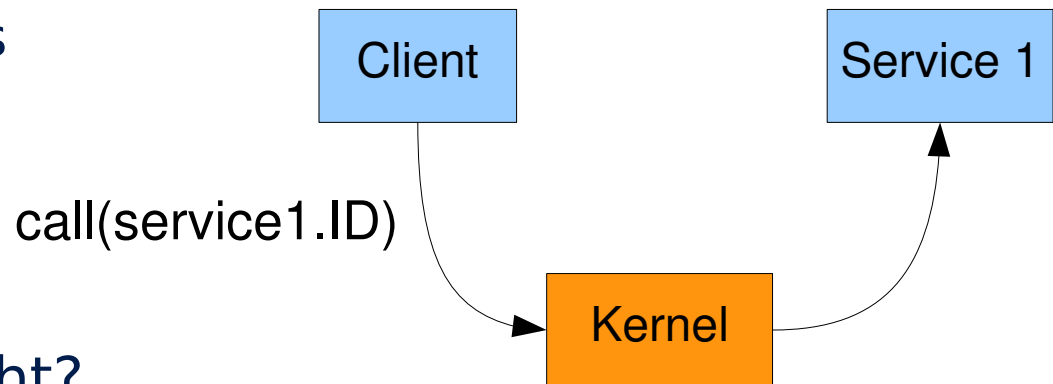
- Kernel-provided objects
  - Threads
  - Tasks
  - IRQs
  - ...
- Generic communication object: IPC gate
  - Send message from sender to receiver
  - Used to implement **new objects** in **user-level** applications



- 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)



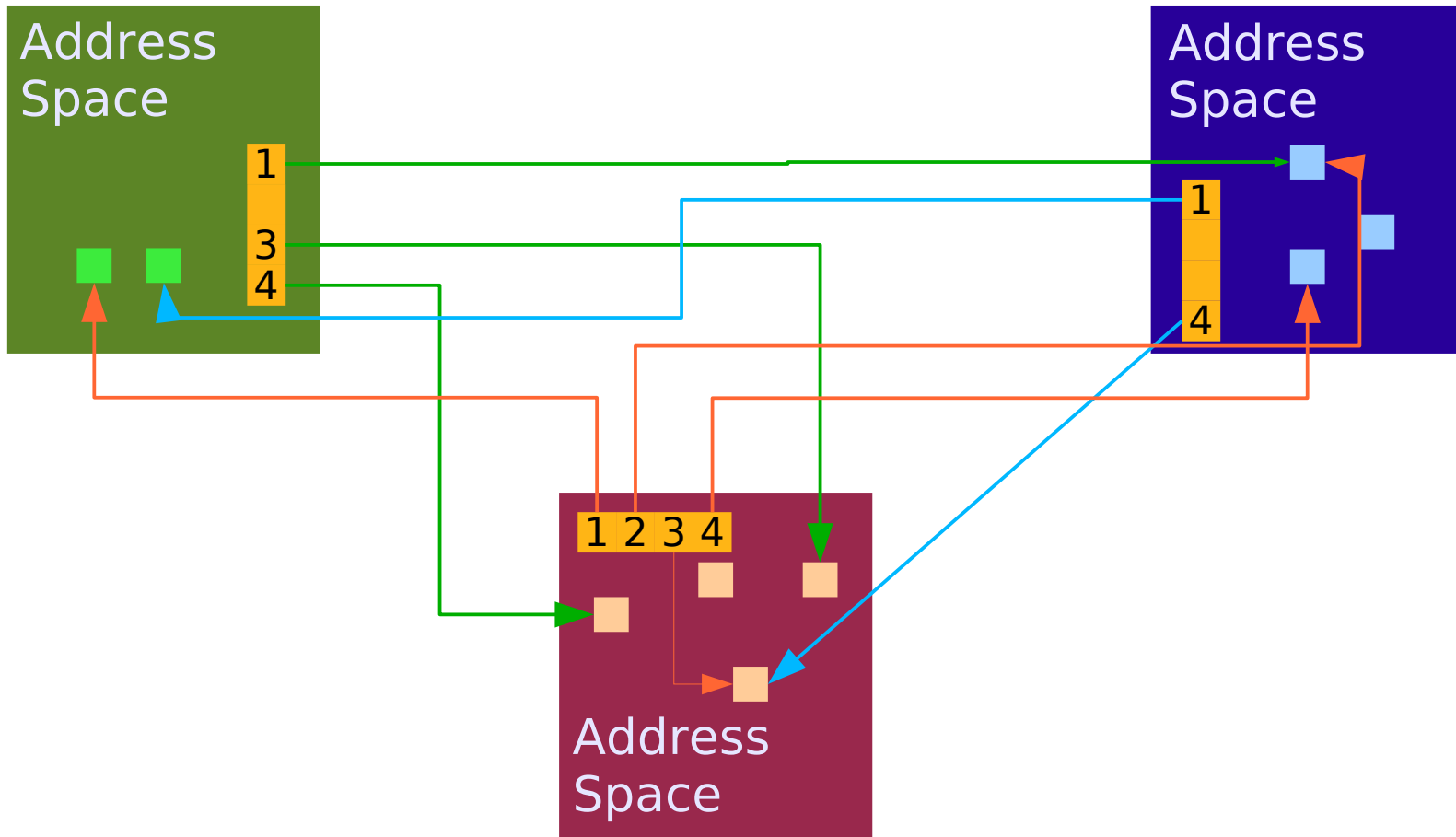
- To call an object, we need an address:
  - Telephone number
  - Postal address
  - IP address
  - ...



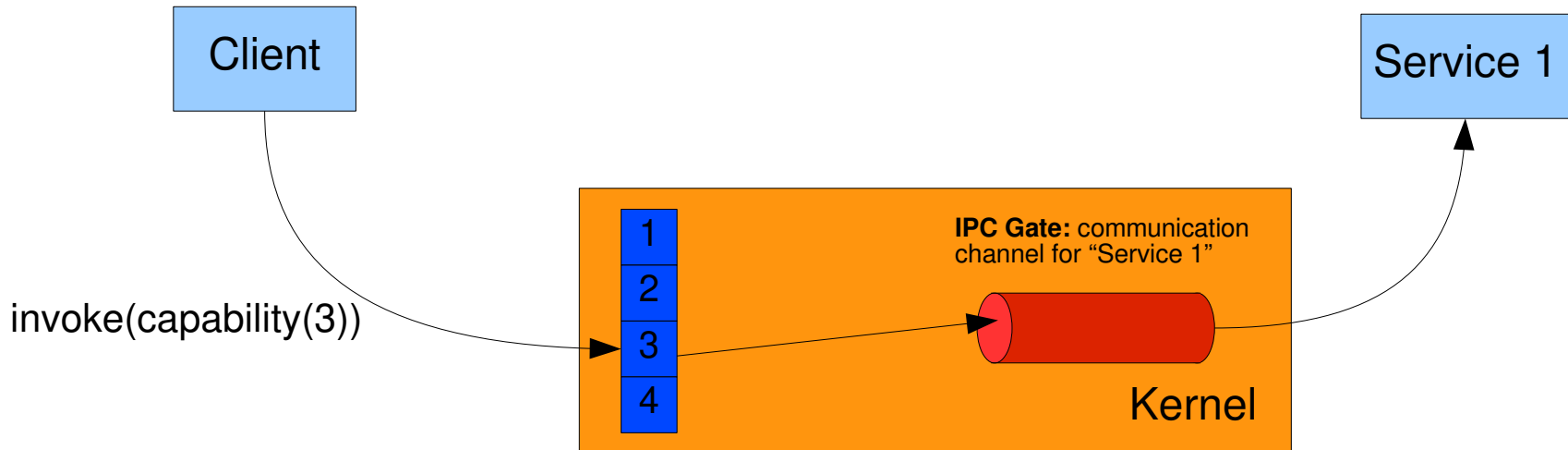
- Simple idea, right?
- ID is wrong? Kernel returns ENOTEXIST
- But not so fast! This scheme is insecure:
  - Client could simply “guess” IDs brute-force.
  - Existence/non-existence can be used as a covert channel

- 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.



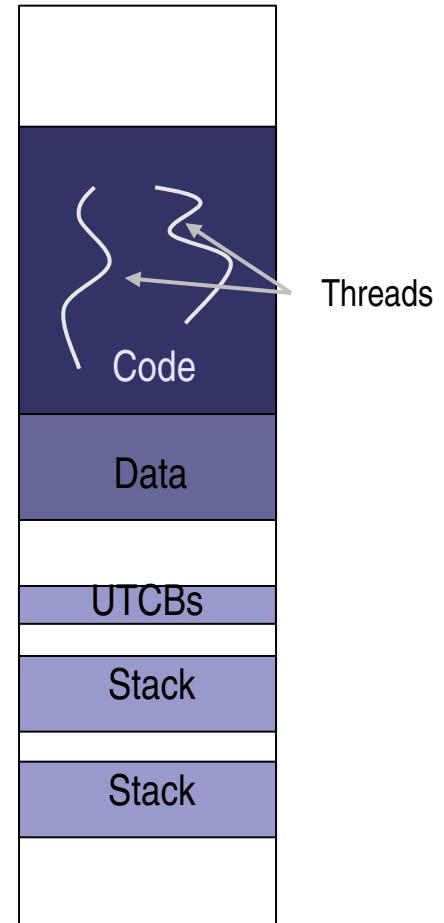
- Kernel object for communication: *IPC gate*
- Inter-process communication (IPC)
  - Between threads
  - Synchronous
- Communication using IPC gate:
  - Sender thread puts message into its UTCB
  - Sender invokes IPC gate, blocks sender until receiver ready (i.e., waits for message)
  - Kernel copies message to receiver thread's UTCB
  - Both continue, knowing that message has been transferred/received



## More L4 concepts

- 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
  - ...

Address Space





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

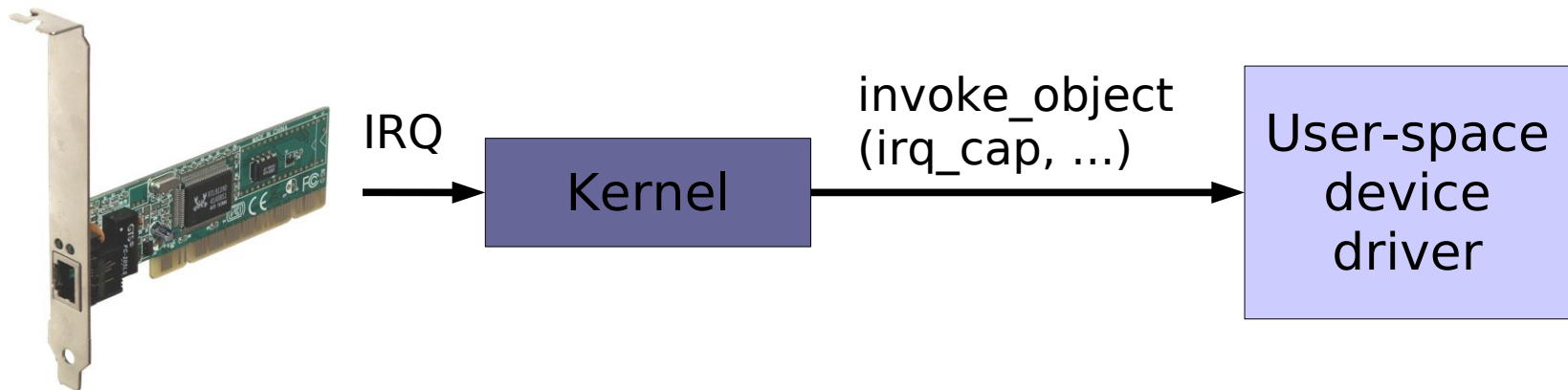
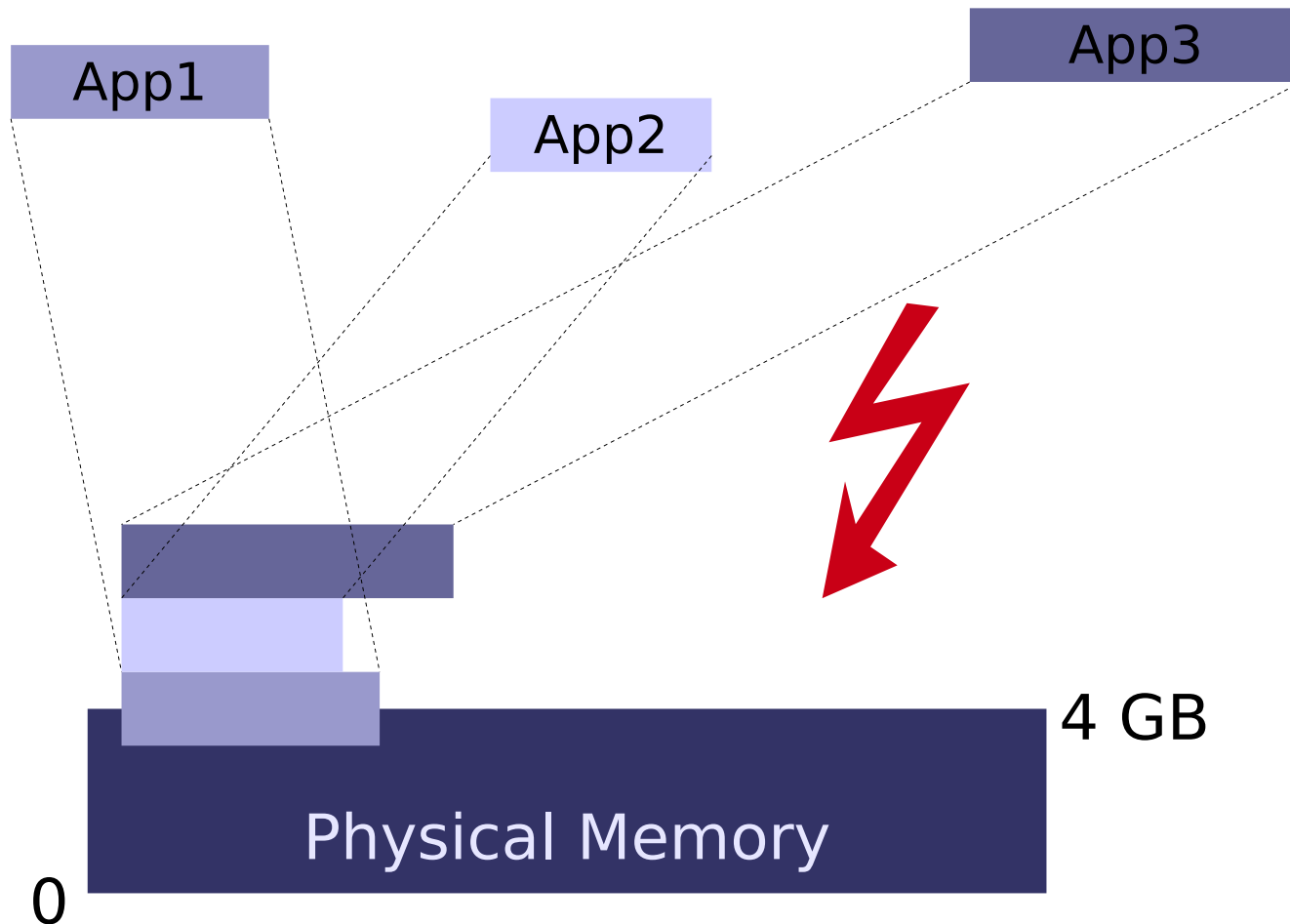
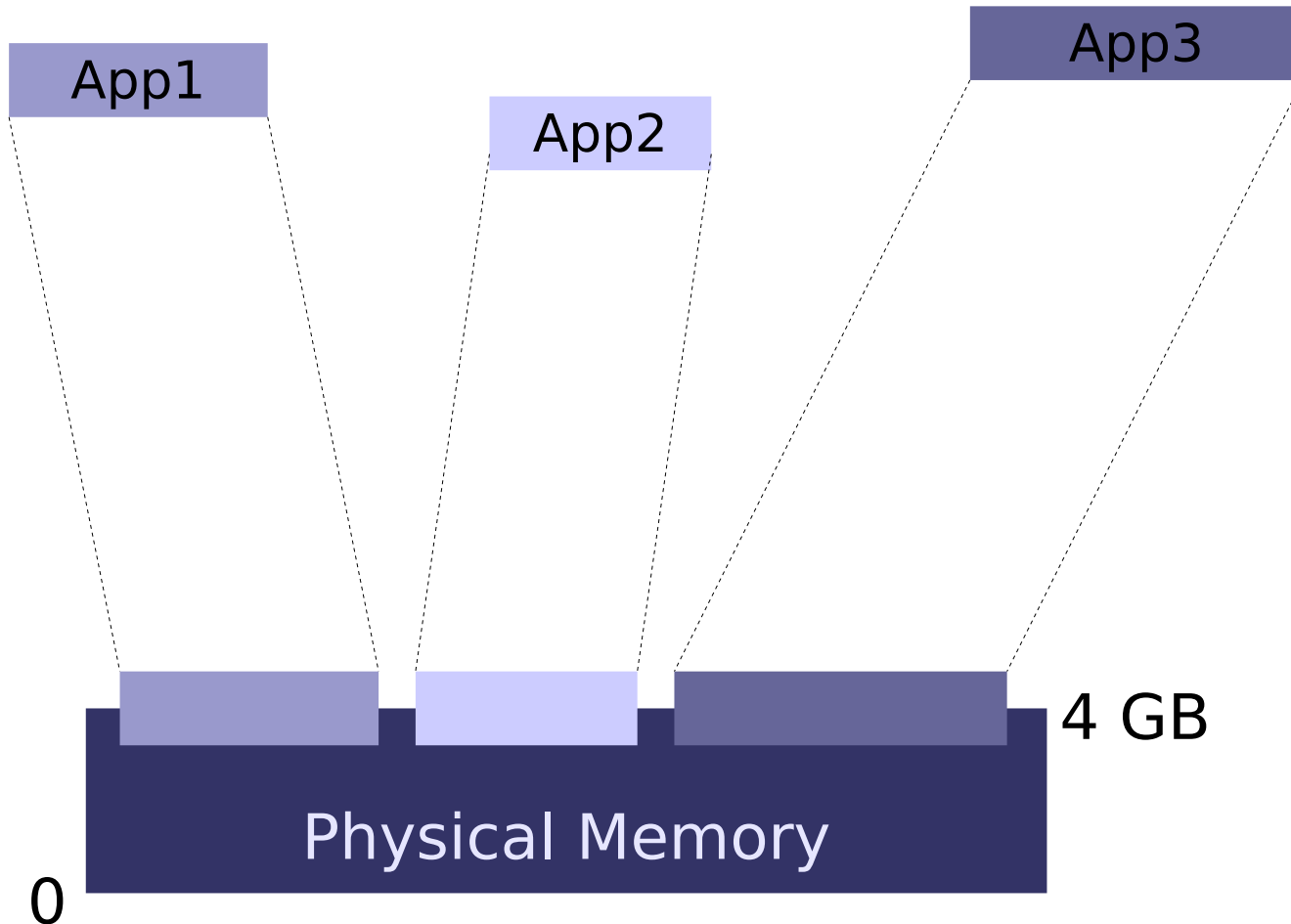


Image source:  
[https://commons.wikimedia.org/File:Ethernet\\_pci\\_card.jpg](https://commons.wikimedia.org/File:Ethernet_pci_card.jpg)

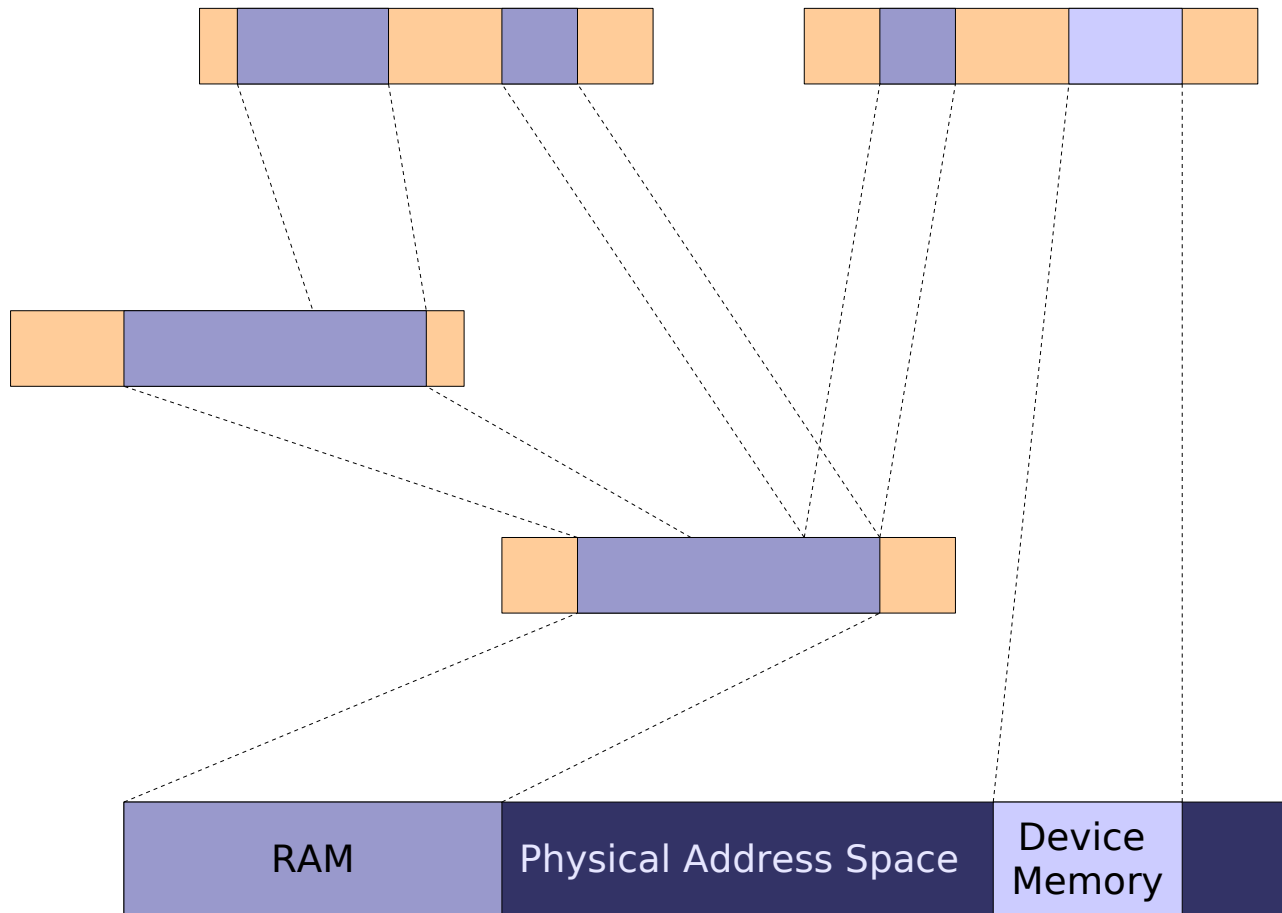
# Problem: Memory partitioning



# Solution: Virtual Memory



# 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)

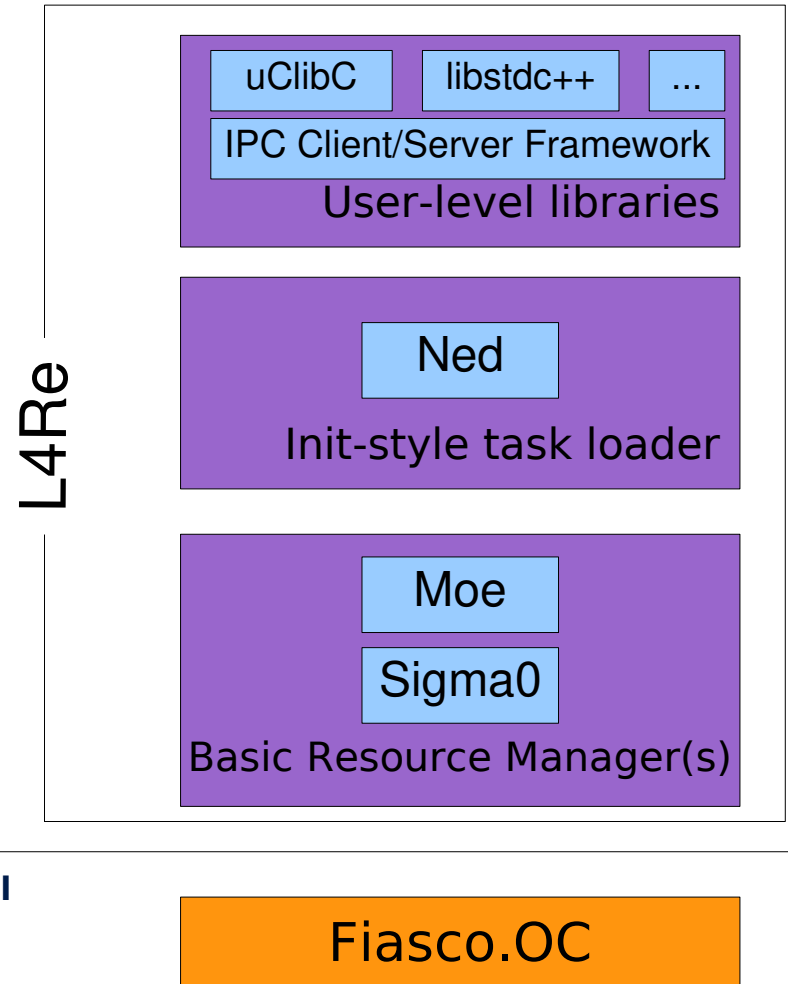
- Summary of object types
  - Task
  - Thread
  - IPC Gate
  - IRQ
  - Factory
- Each task gets initial set of capabilities for some of these objects at startup

What can we build with all this?

- 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

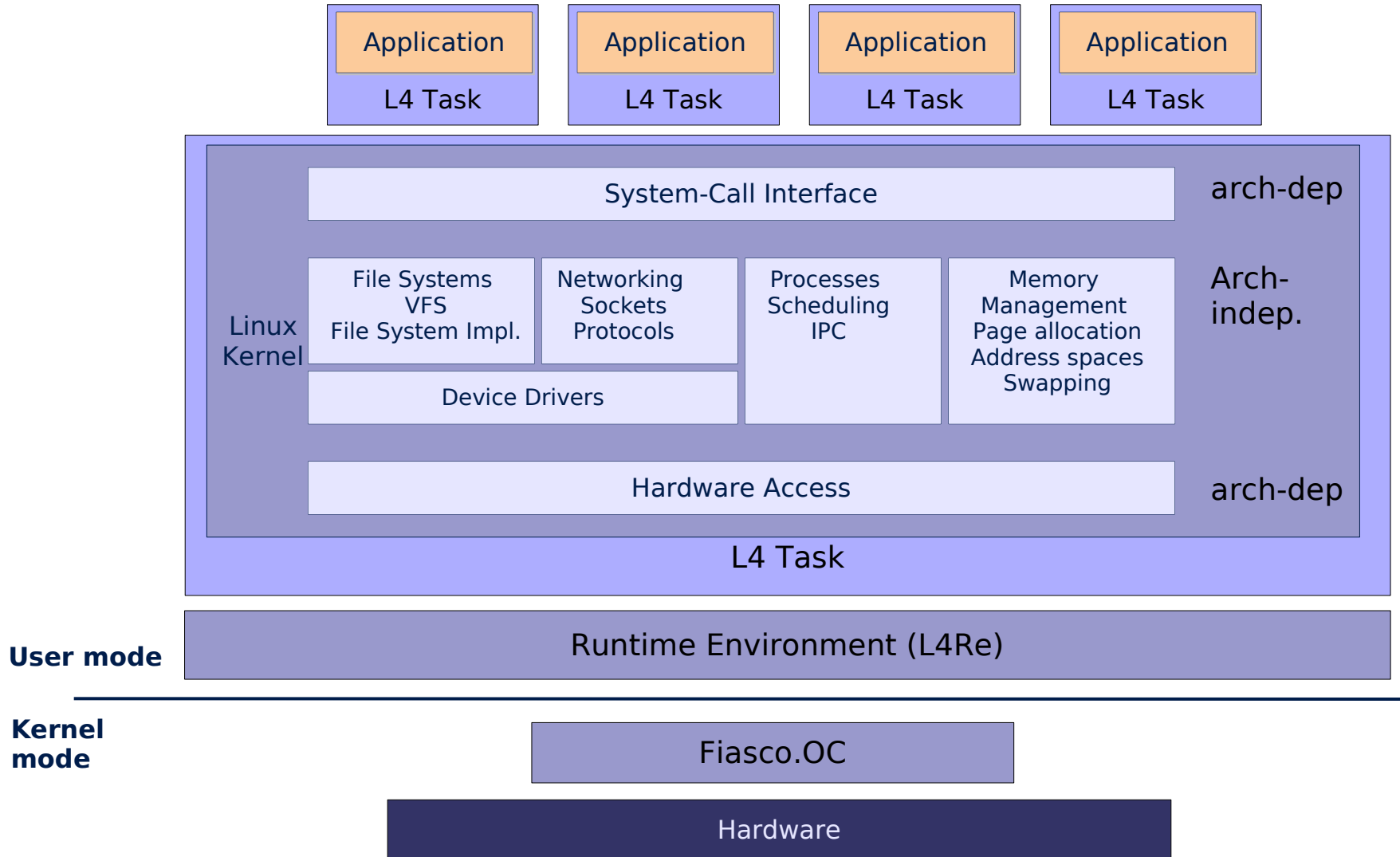


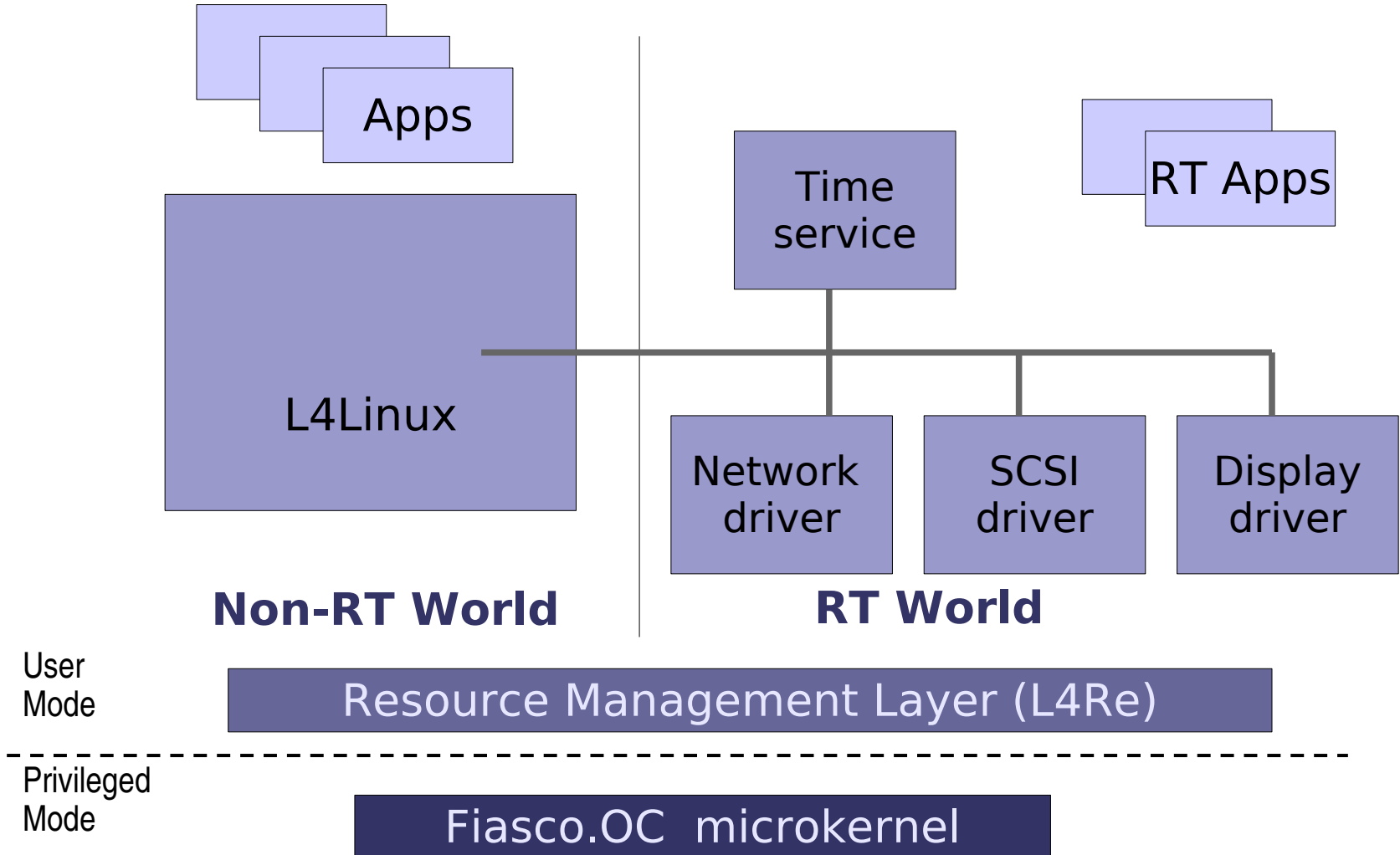
L4Re - L4 Runtime Environment



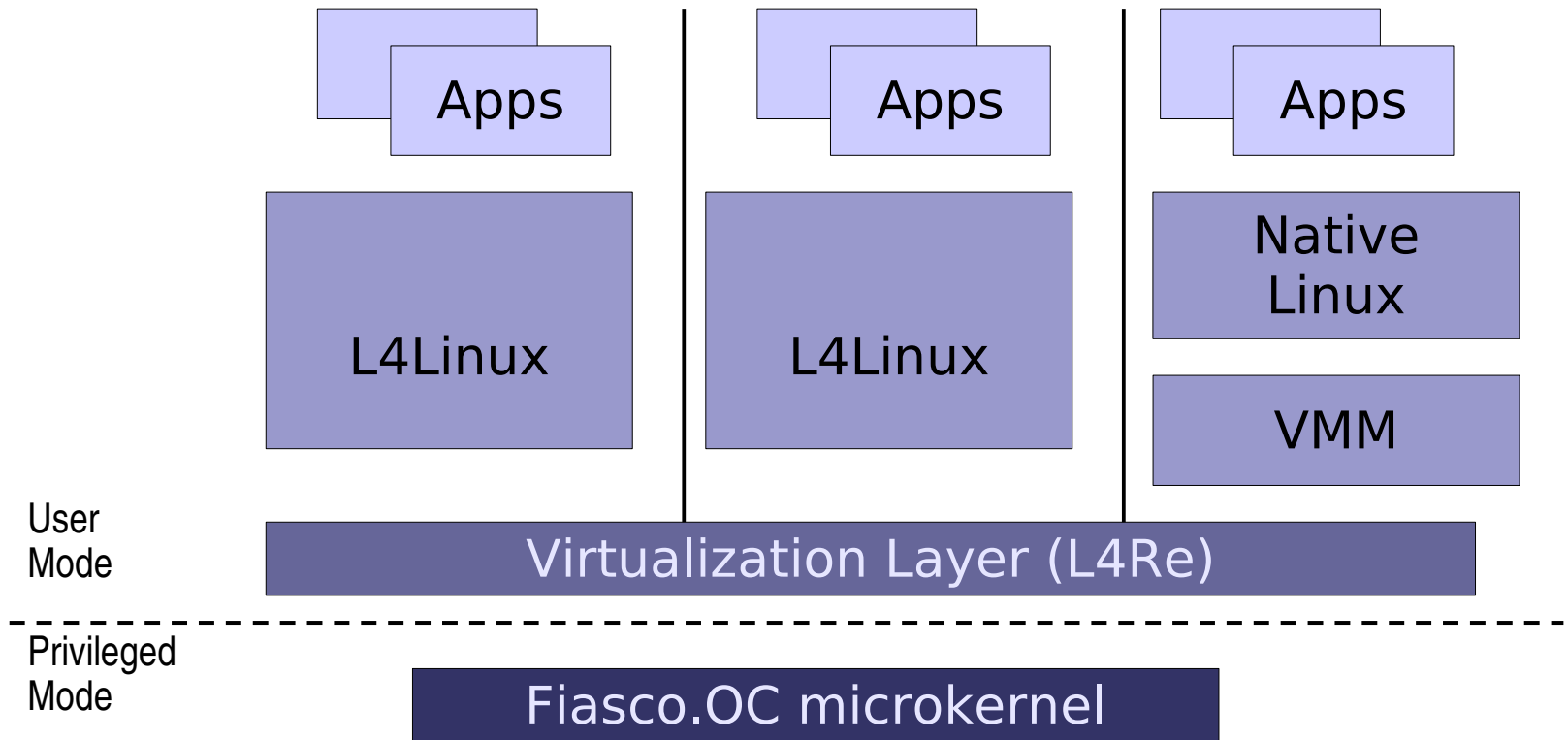


# Linux on L4

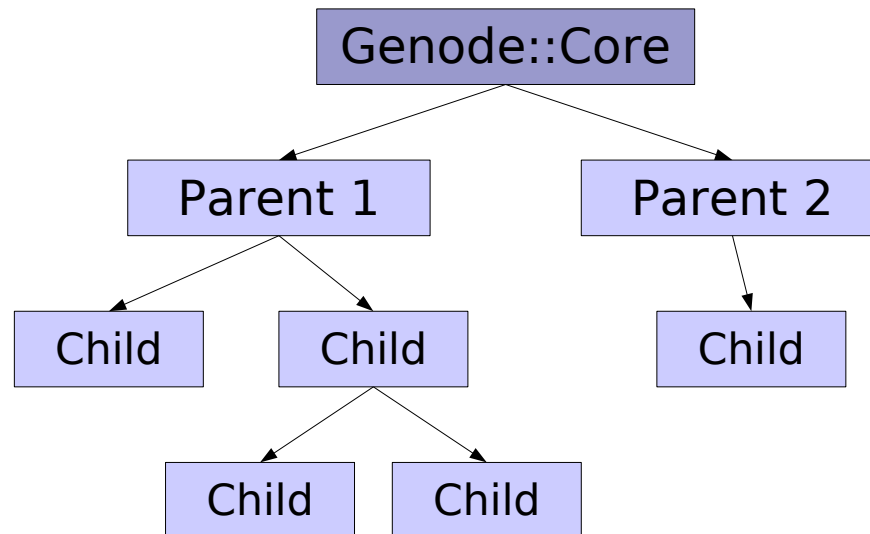




- Isolate not only processes, but also complete Operating Systems (compartments)
- “Server consolidation”



- 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?

- Next lecture:
  - “Threads & Synchronization”
  - Next week (Oct 22, 4:40 PM)
- First exercise:
  - Practical Exercise: *Booting*
  - Room will be announced on mailing list