



TECHNISCHE
UNIVERSITÄT
DRESDEN

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

Microkernel-based Operating Systems - Introduction

Björn Döbel

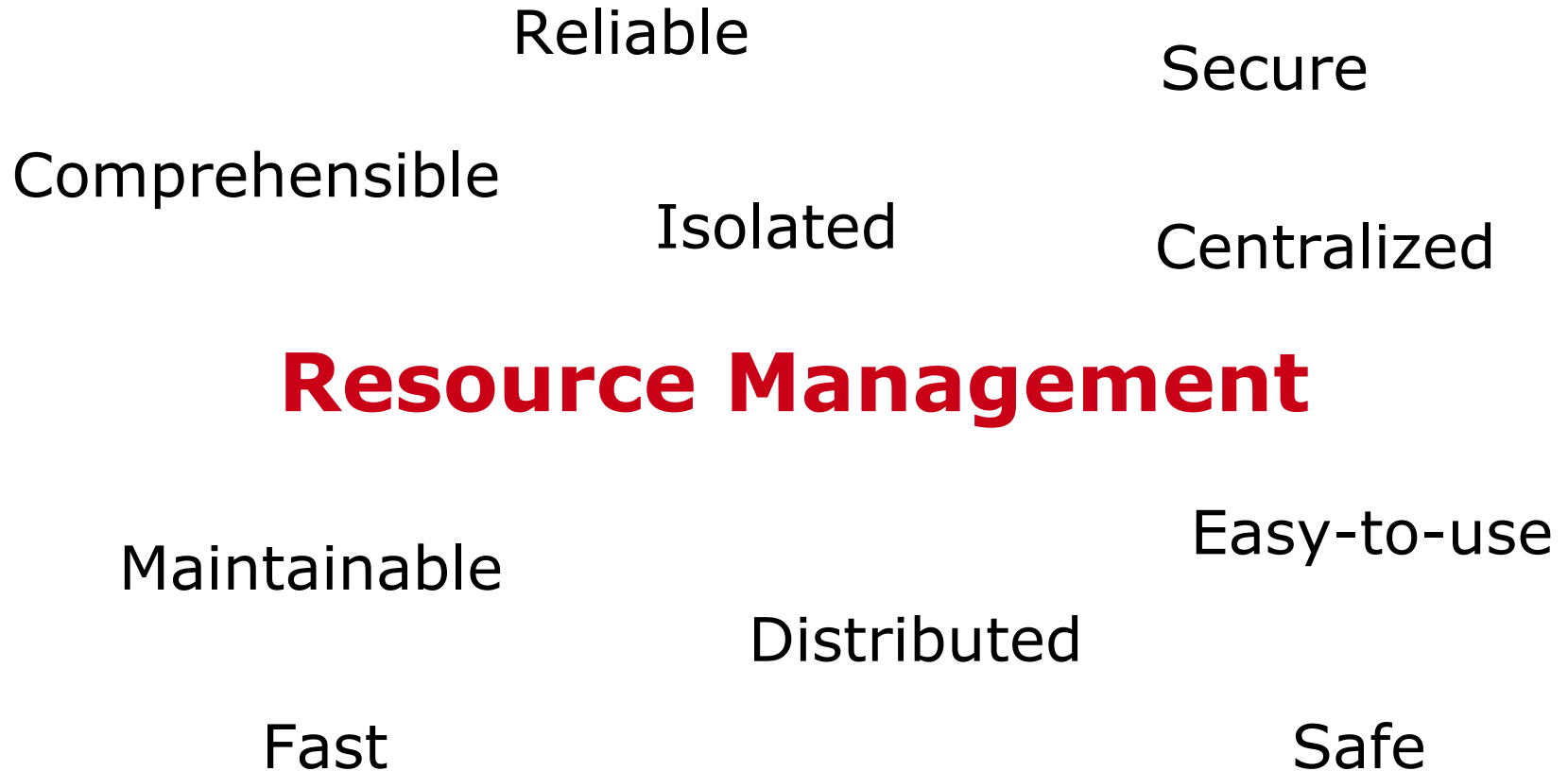
Dresden, Oct 13th 2009

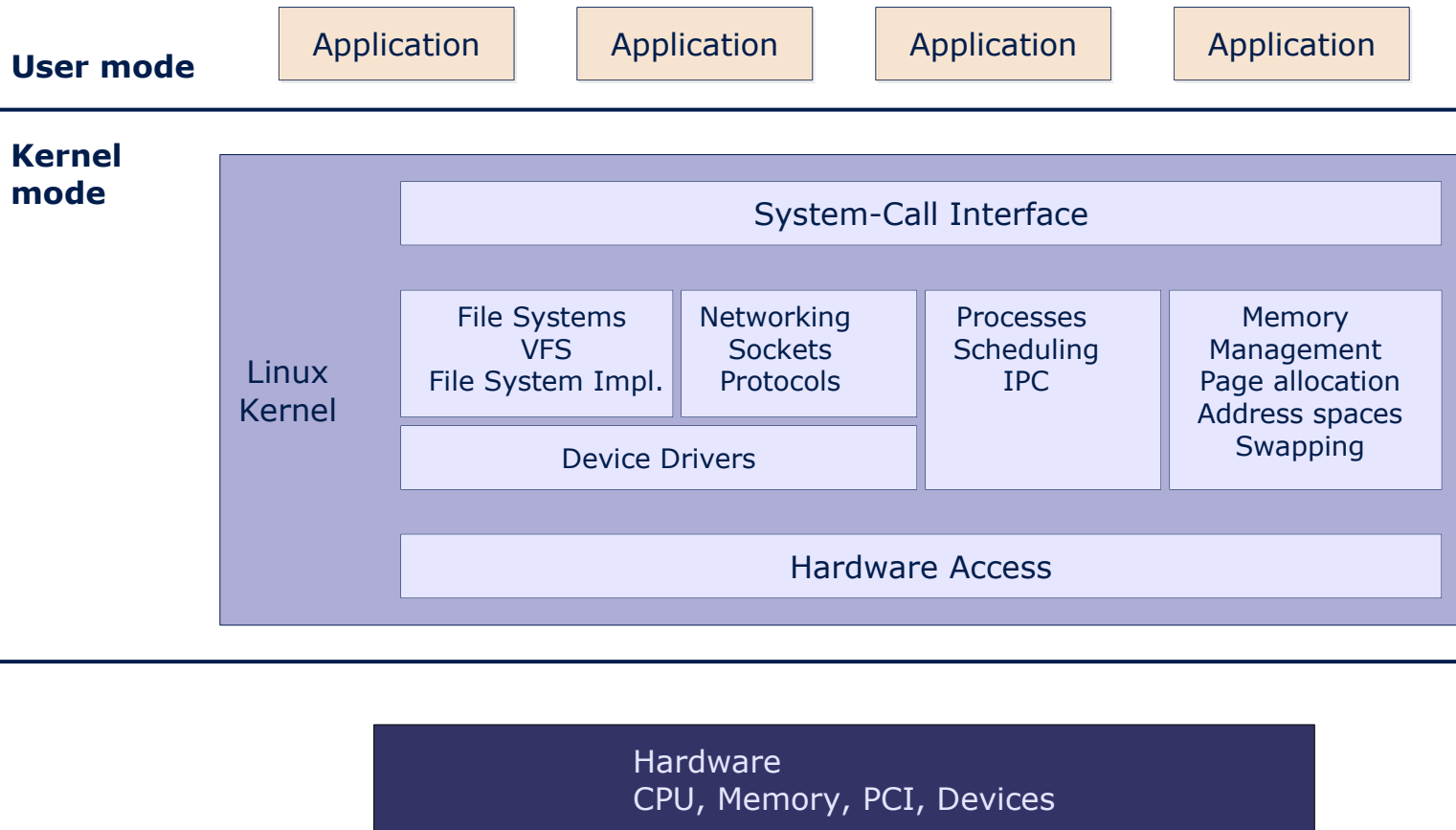
- Provide deeper understanding of OS mechanisms
- Illustrate alternative design concepts
- Promote OS research at TU Dresden
- Make you all enthusiastic about OS development in general and microkernels in special

- Lecture every Tuesday, 1:00 PM, INF/E08
- Slides: <http://www.tudos.org> -> Teaching -> Microkernel-based Operating Systems
- Subscribe to our mailing list:
<http://os.inf.tu-dresden.de/mailman/listinfo/mos2009>
- 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, computer lab
 - You'll need a quota raise.

- Complex lab in parallel to lecture
- Groups of 2-3 students.
- Build several components of an OS
- “Komplexpraktikum” for (Media) Computer Science students
- “Internship” for Computational Engineering
- starts on Tuesday, Oct 27th, 5th block (14:50), INF/E08

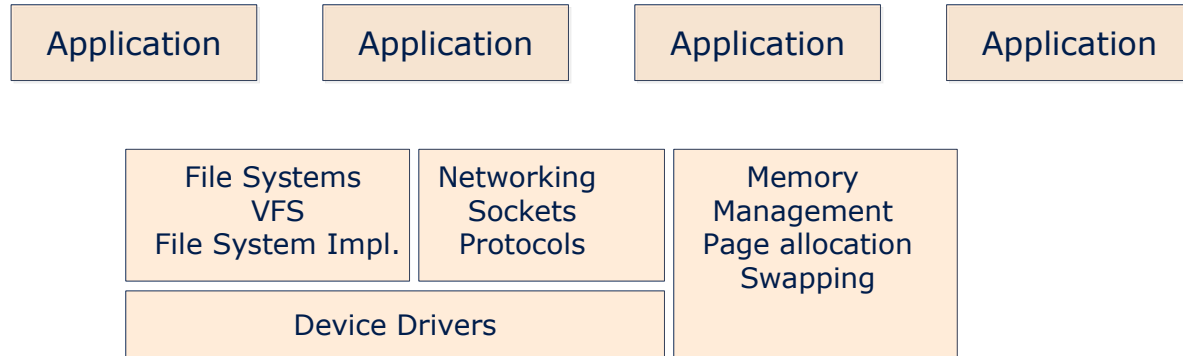




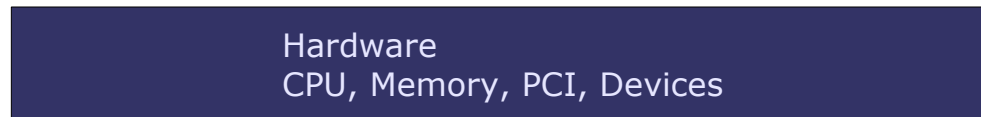
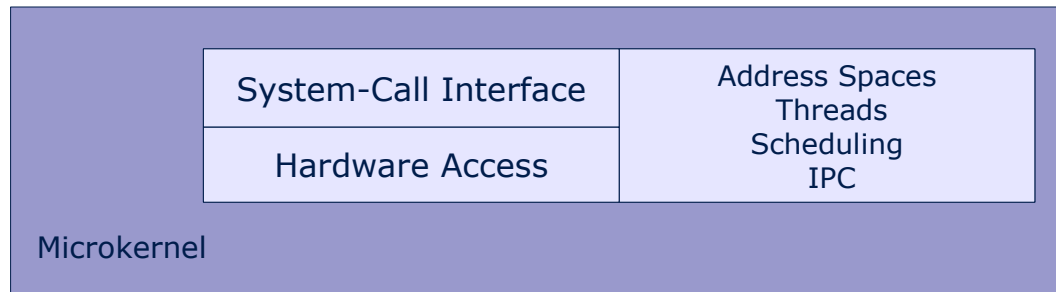
- All system components in privileged mode.
- No built-in isolation
 - Faulty driver crashes the whole system.
 - More than 2/3 of today's systems are drivers.
- No enforcement of good system design
 - can directly access all kernel data structures
- Size and inflexibility
 - Not suitable for embedded systems.
 - Difficult to replace single components.
- Increasing complexity becomes more and more difficult to manage.

The microkernel vision

User mode



Kernel mode



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

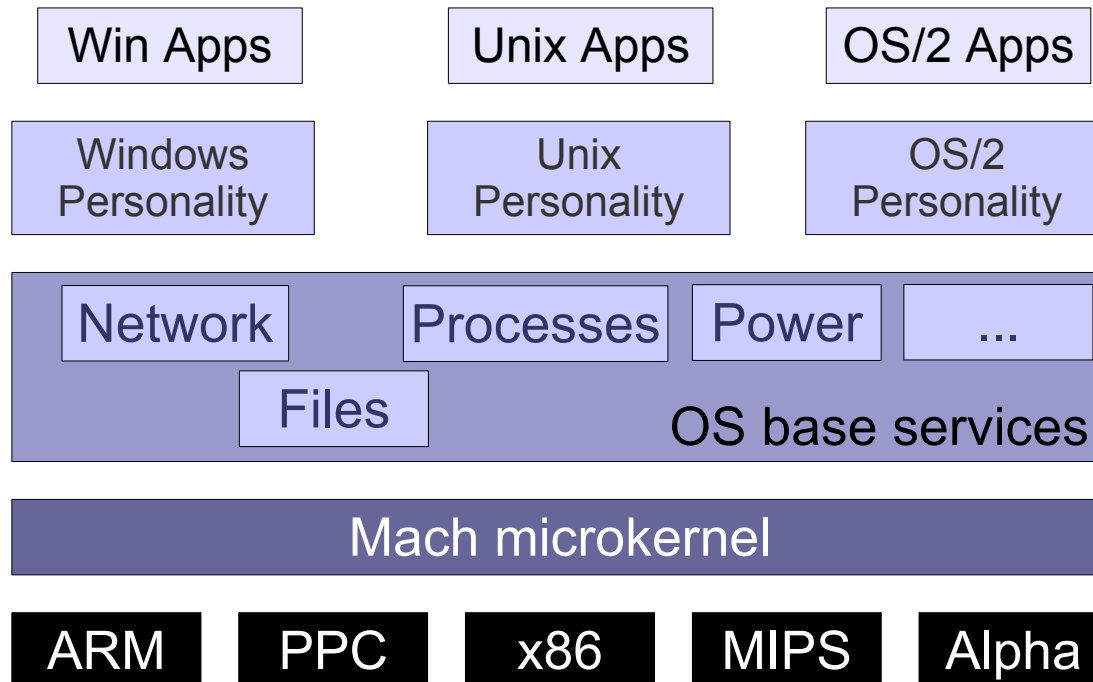
- OS personalities
- Servers may be configured to suit the target system (small embedded systems, desktop PCs, SMP systems, ...)
- Enforce reasonable system design
 - Well-defined interfaces between components
 - No access to components besides these interfaces
 - Improved maintainability

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

- Simple, extensible *communication kernel*
 - “Everything is a pipe.” – *ports* as secure communication channels
- Multiprocessor support
- Message passing by mapping
- Multi-server OS
- 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

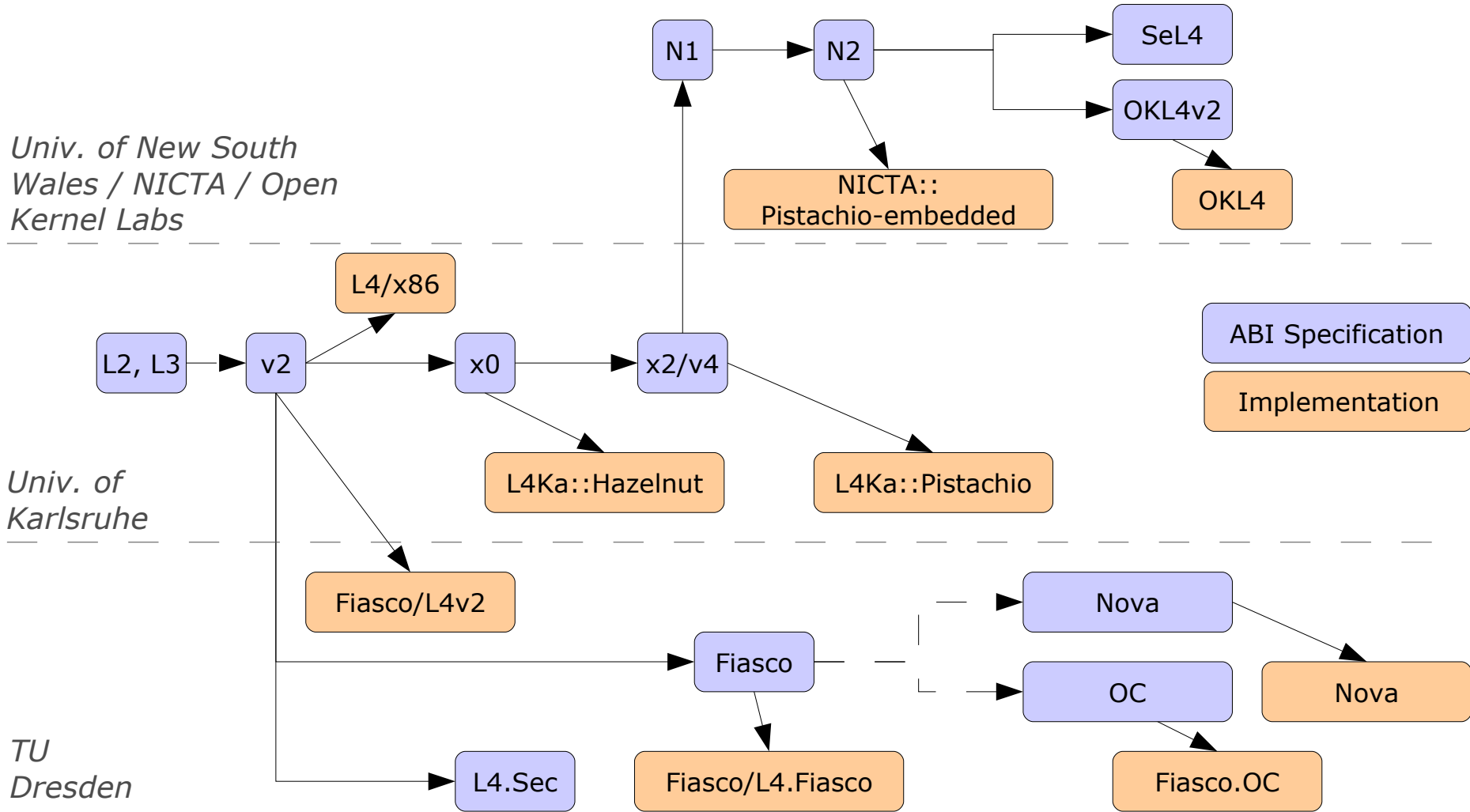
- Subsystem protection / isolation
- Code size
 - Fiasco kernel: $\sim 15,000$ LoC
 - Minimal application:
(boot loader + "hello world"):
 $\sim 6,000$ LoC
 - Linux kernel (2.6.24, x86 architecture):
 ~ 1.6 million LoC
(+drivers: ~ 2.8 million LoC)

(generated using David A. Wheeler's 'SLOCCount')
- Customizable
 - 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
 - 2nd generation microkernel
 - Several kernel ABI versions

The L4 family – a timeline



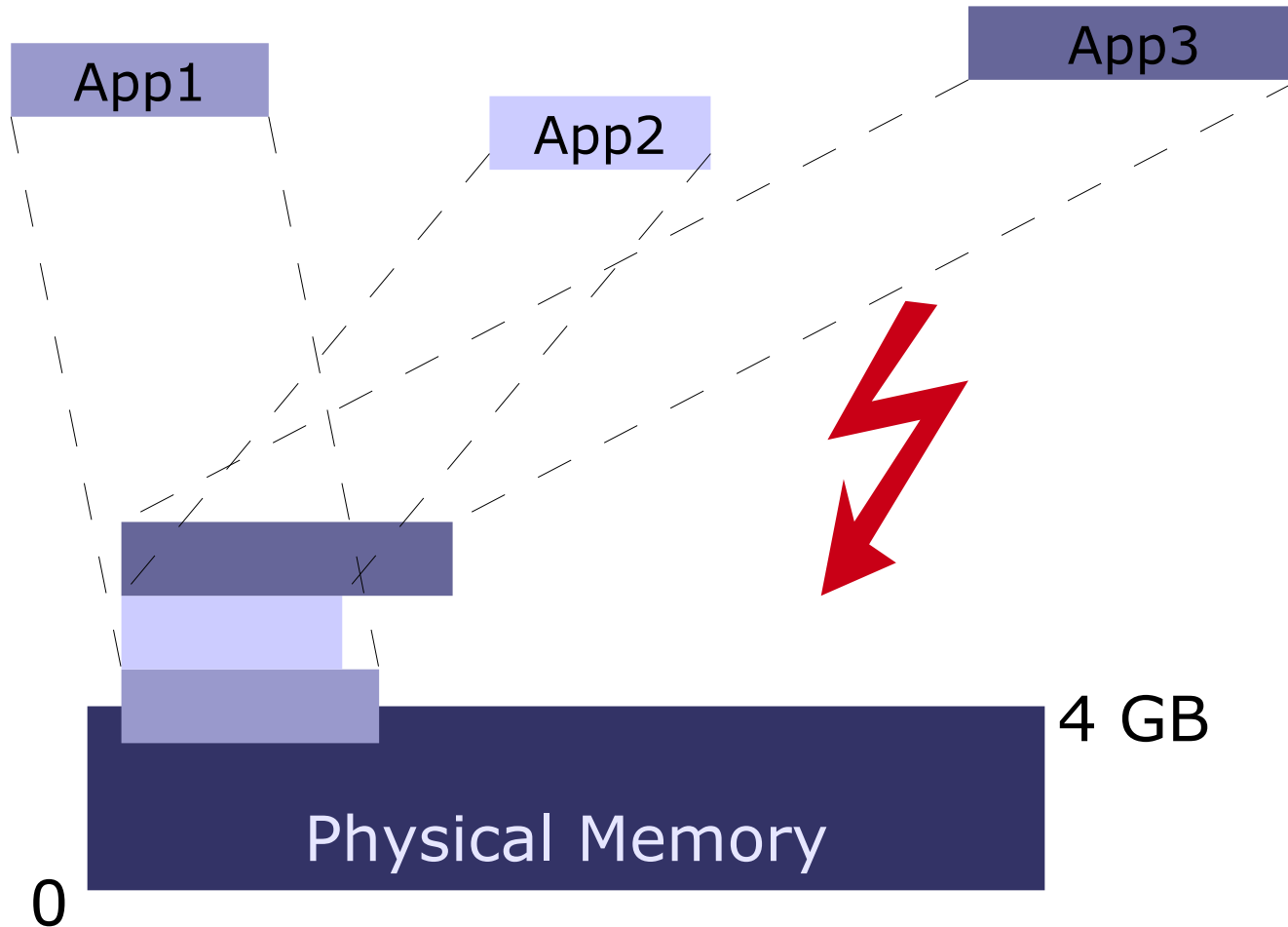
- Jochen Liedtke:
"A microkernel does no real work."
 - kernel provides inevitable mechanisms
 - kernel does not enforce policies
- But what **is** inevitable?
 - Abstractions
 - Threads
 - Address spaces (tasks)
 - Mechanisms
 - Communication
 - Resource Mapping
 - (Scheduling)

- “Everything is an object.”
- 1 system call: *invoke_object()*
 - Parameters passed in UTCB
 - Types of parameters depend on type of object
- Objects referenced by capabilities
 - *invoke()* allowed for everyone possessing a capability to the object
 - Capabilities can be mapped just like every other resource.

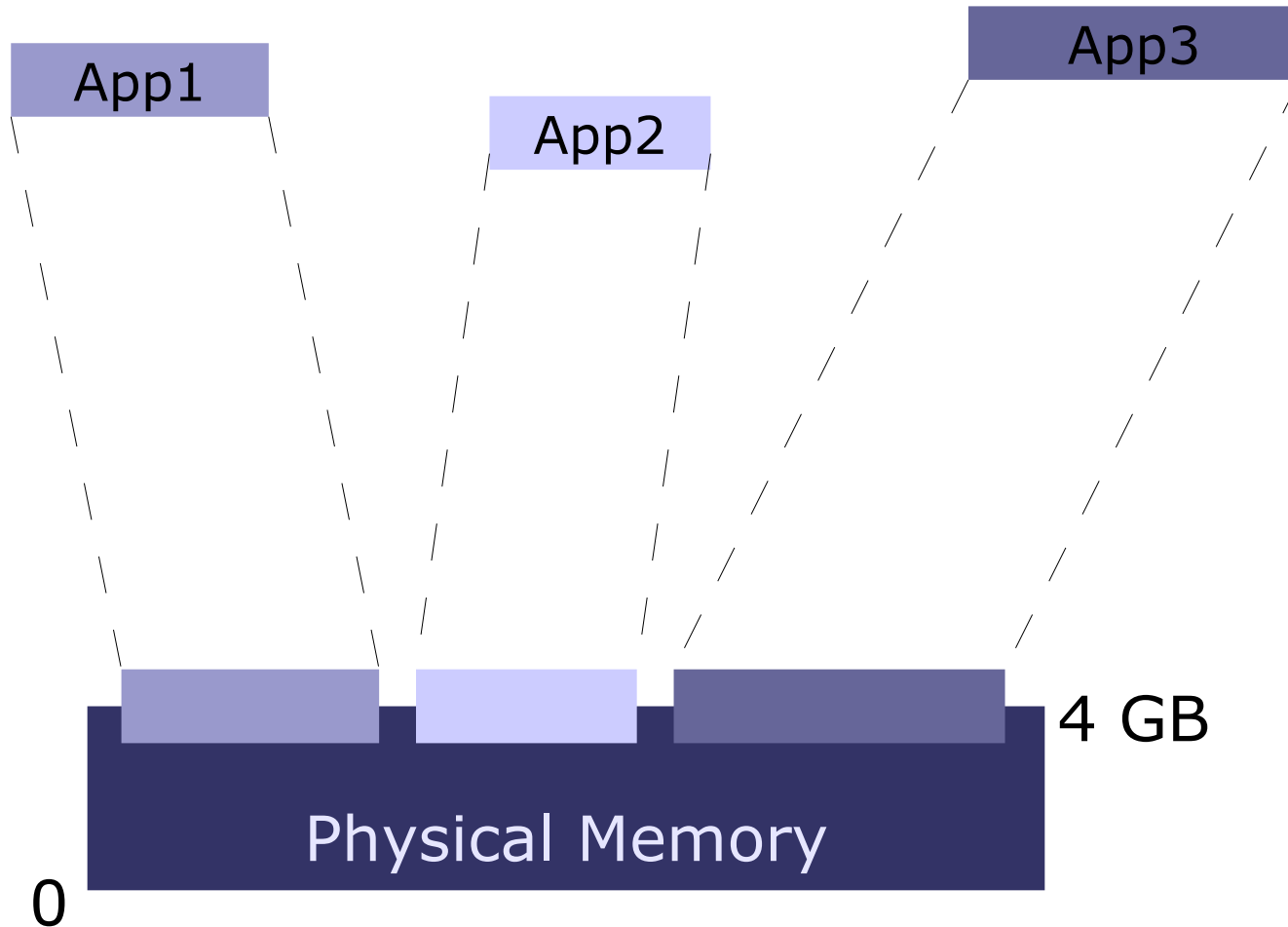
- Factory
 - Create other objects
 - Enforce resource quotas
- Task
 - Address + capability space
- Thread
- IPC Gate
- IRQ

- Each task gets initial set of capabilities upon startup.

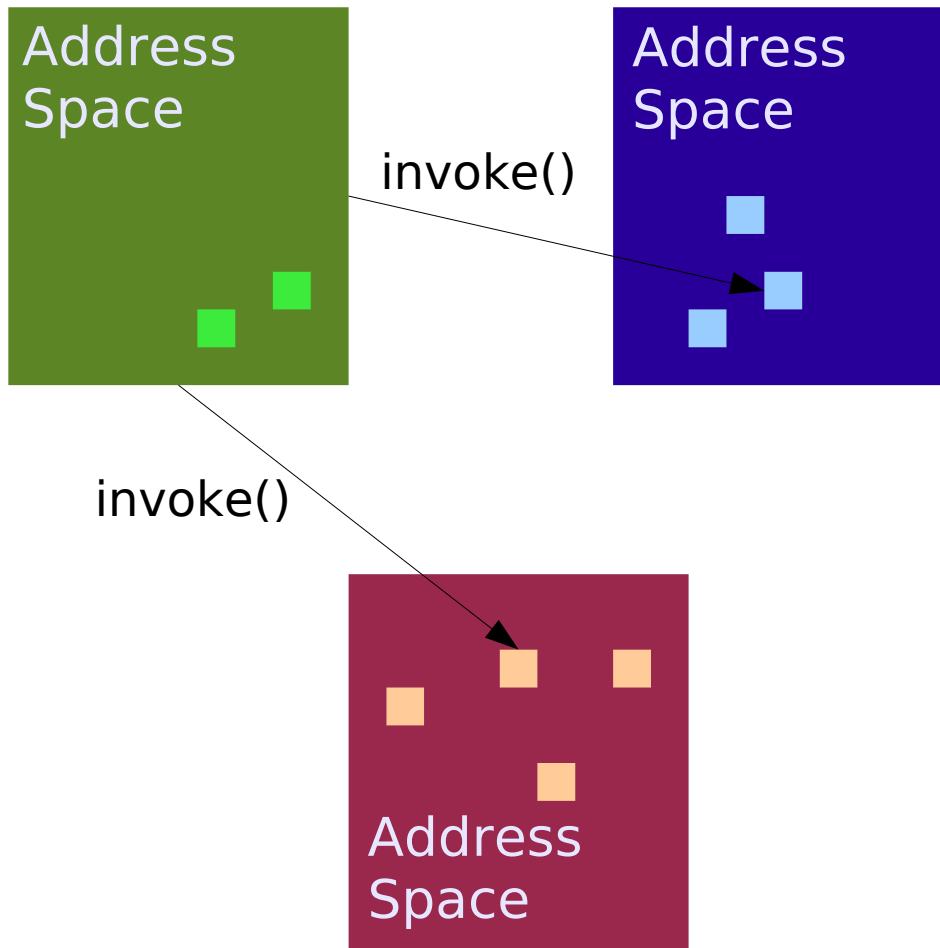
Problem: Memory partitioning



Solution: Virtual Memory



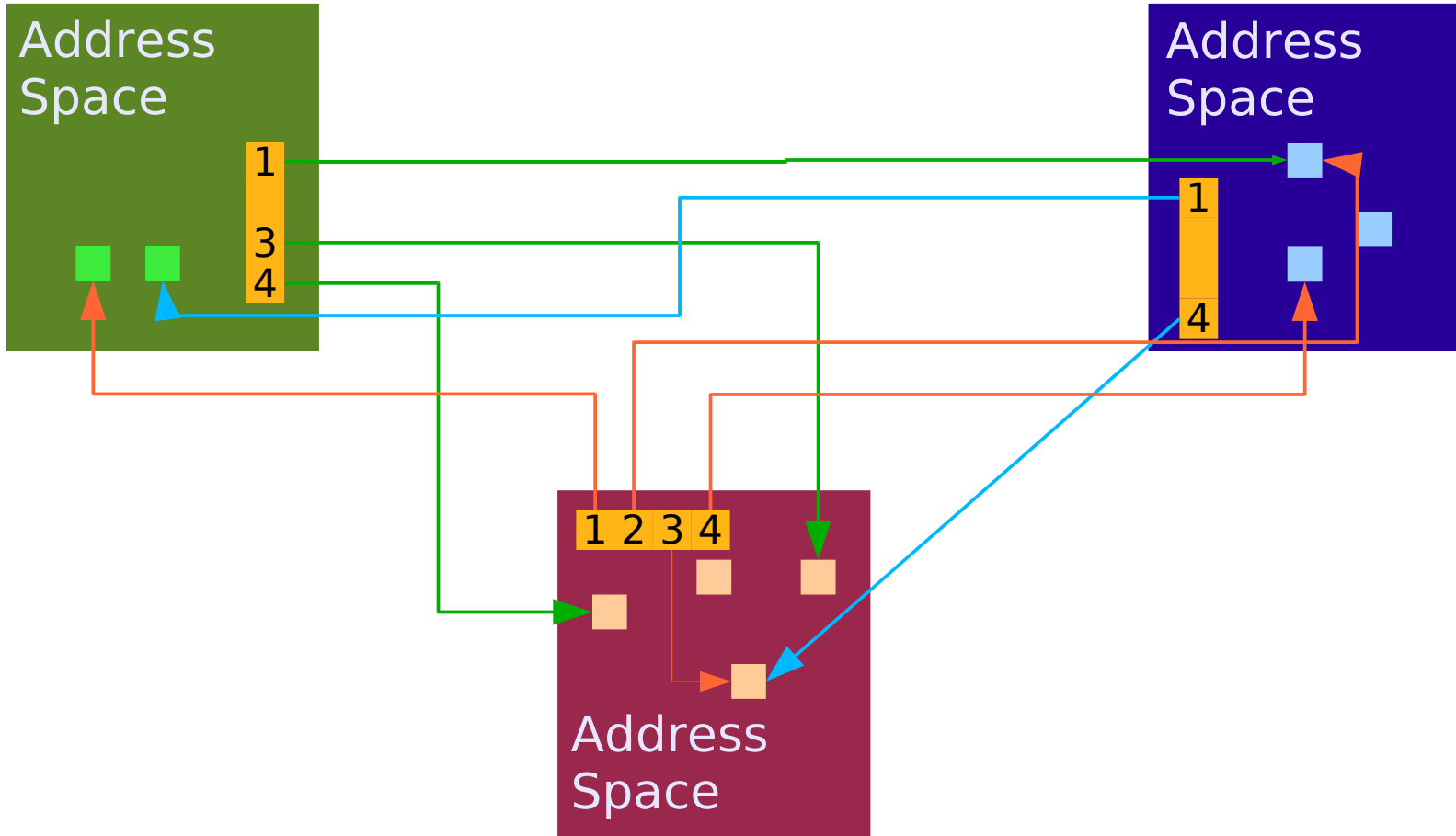
- General problem:
 - Partition resources for concurrent use by different applications
- Examples
 - CPU partitioning → Scheduling
 - Memory partitioning → Virtual memory
 - Hard disks → multiple logical drives
 - Computer partitioning → virtual machines
 - IP address ranges



- Addressing: How does green know how to find objects? → need object ID
- Security: global IDs can be forged.
- Flexibility: what happens to object ID if blue object implementation is moved to red task?

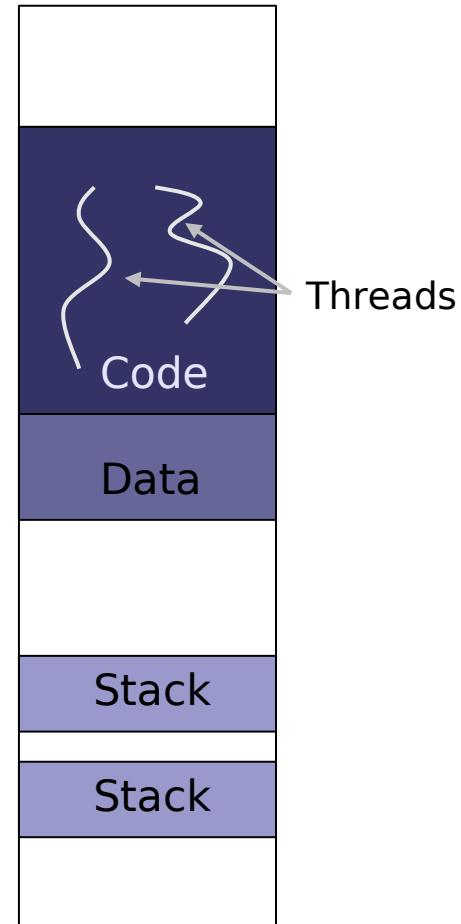
- Global object IDs are
 - insecure (forgery, covert channels).
 - inconvenient (programmer needs to know about partitioning in advance)
- Solution in Fiasco.OC: task-local *capability space* as an indirection
- Per-task name space (configured by task's creator) to map physical names to object capabilities.

Indirection allows for security and flexibility.



- Thread ::= Unit of Execution
- 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
 - ...

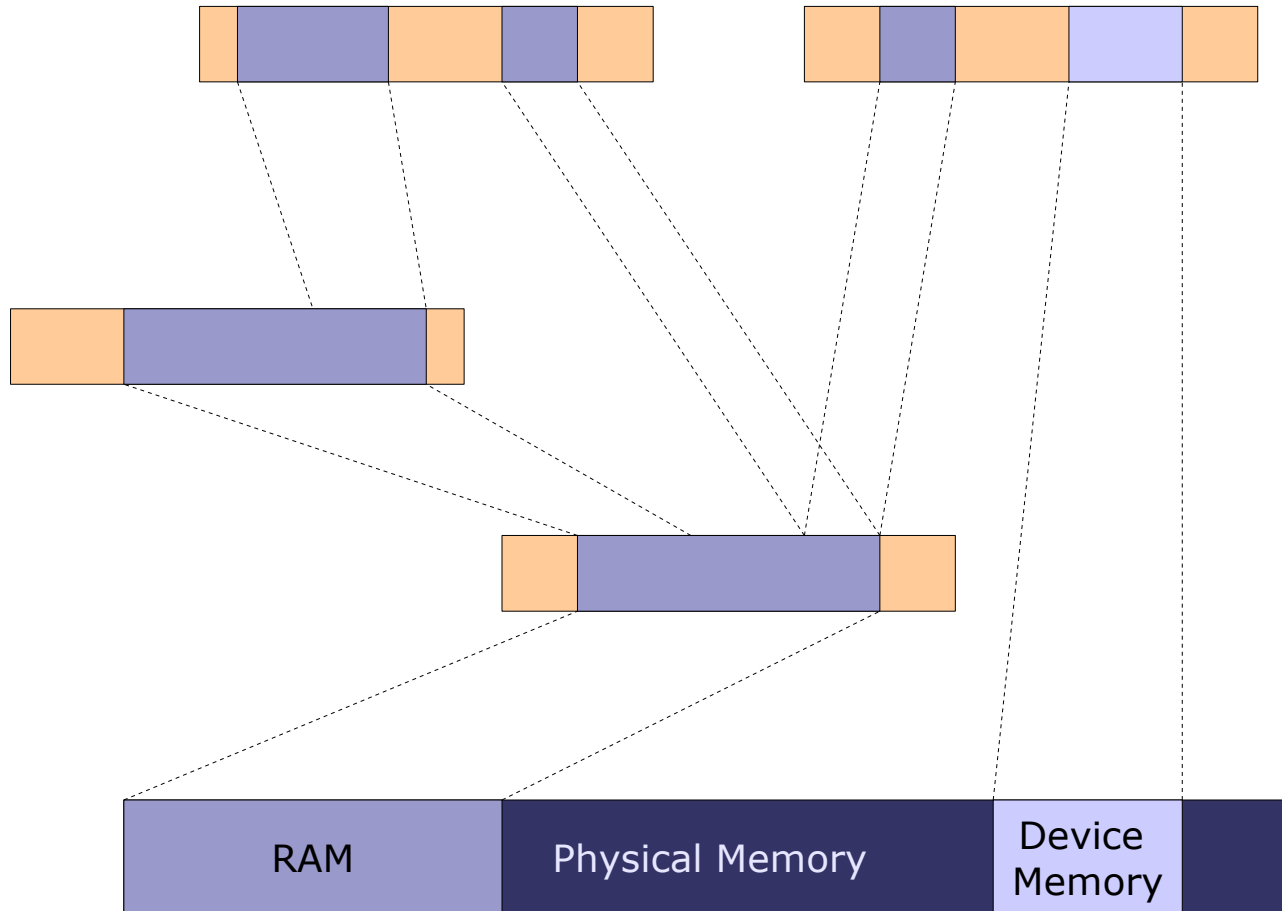
Address Space



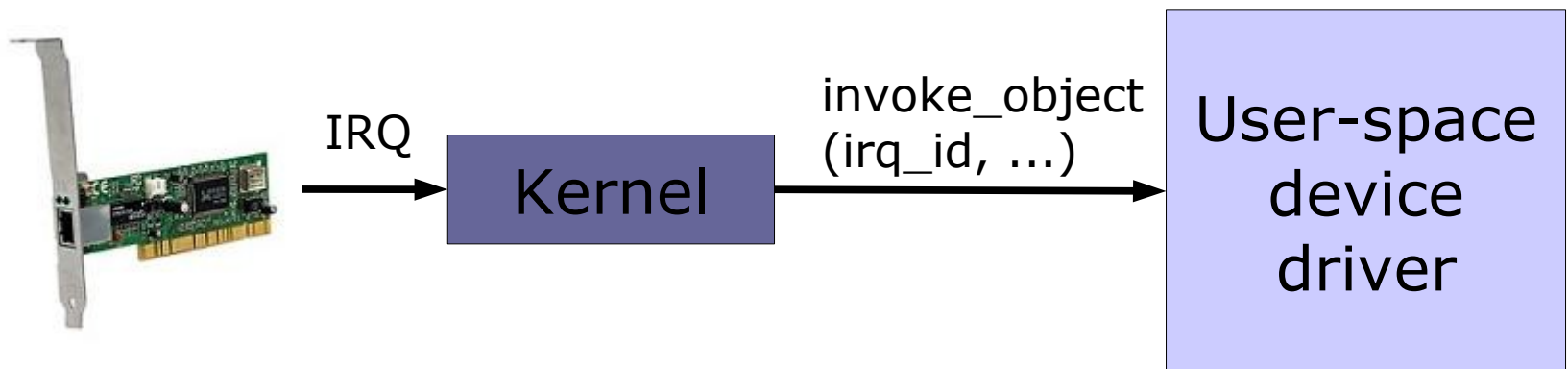
- Synchronous inter-process communication (IPC) between threads
- Kernel object: *IPC gate*
- Communication:
 - Put message into sender's UTCB
 - Invoke IPC gate (blocks until receiver ready)
 - Kernel copies message to receiver UTCB
- Note: This is the same procedure as for any other *invoke_object()*
 - allows object interpositioning

- If a thread has access to a capability, it can map this one to another thread.
- Abstraction for mapping: *flexpage*
- Flexpages describe mapping
 - location and size of resource
 - receiver's rights (read-only, mappable)
 - type (memory, IO, communication capability)

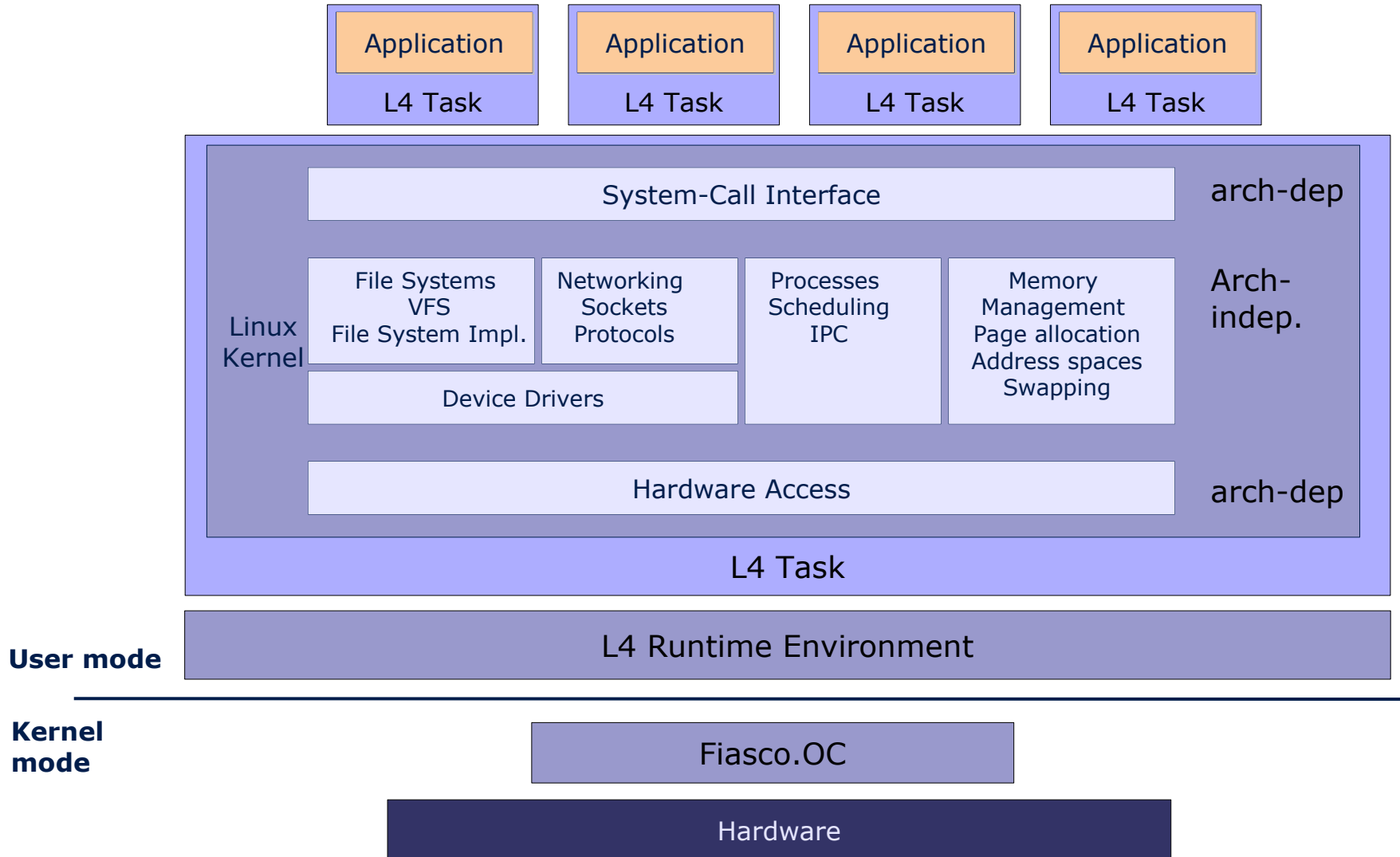
L4 – Recursive address spaces

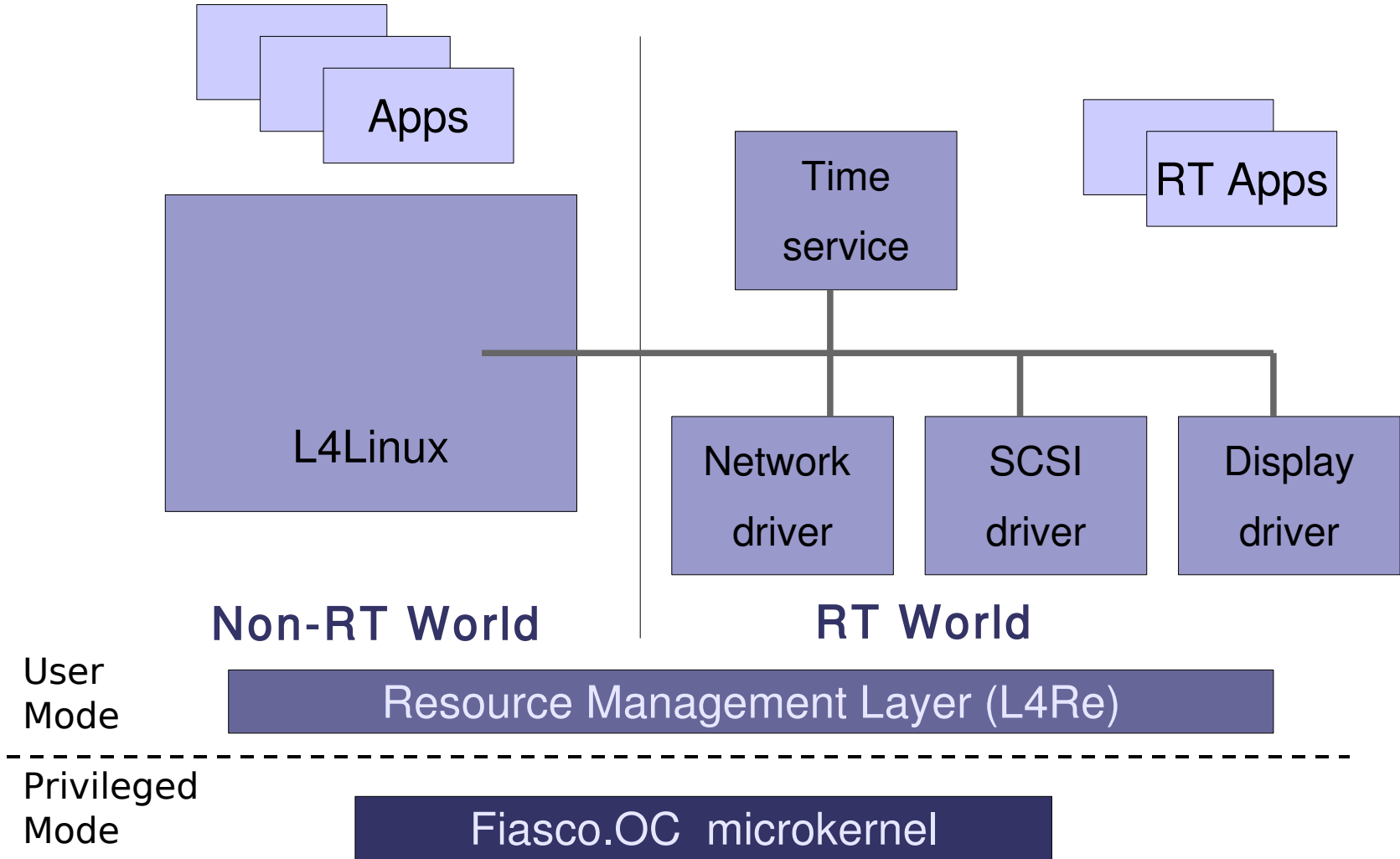


- Kernel object: IRQ
- Used for hardware and software interrupts (read: asynchronous signals)
- Wait for IRQ: *invoke_object(irq)*

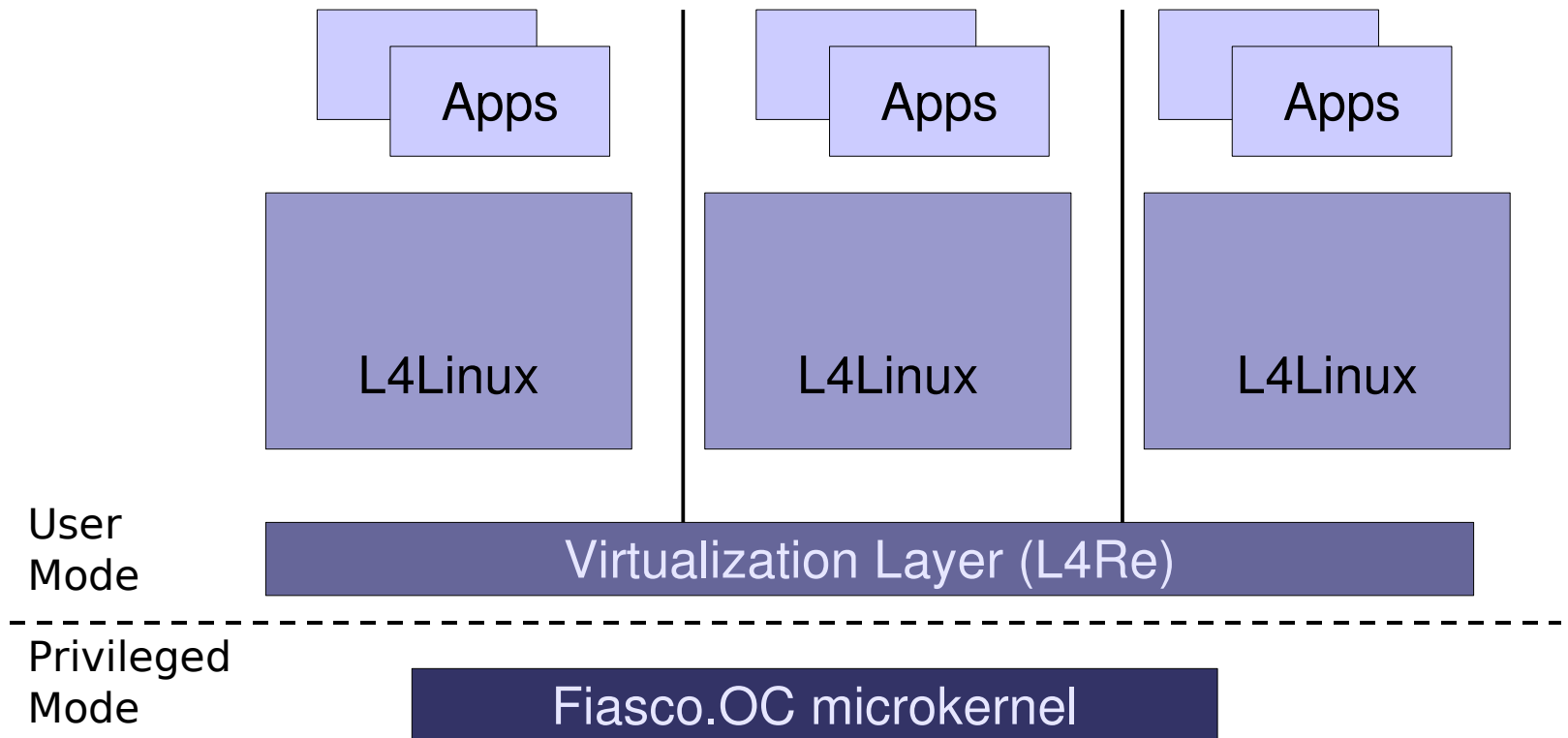


Linux on L4

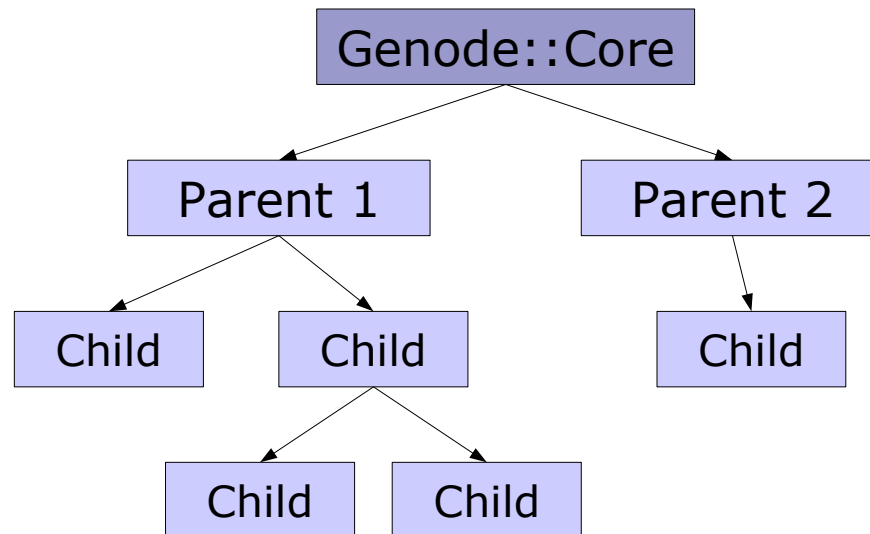




- Isolate not only processes, but also complete Operating Systems (compartments)
- “Server consolidation”



- 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:
 - “Tasks, Threads and Synchronization”
on Oct 21st
- Next exercise:
 - Oct 21st
 - Building and booting an L4 system