TECHNISCHE UNIVERSITÄT DRESDEN

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

Microkernel-based Operating Systems - Introduction

Björn Döbel

Dresden, Oct 12th 2010

Lecture Goals

- 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

Administration - Lecture

- 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/mos2010
- This lecture is **not:** Microkernel construction (in summer term)

TECHNISCHE UNIVERSITÄT DRESDEN

Administration - 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*

Complex lab

- Complex lab in parallel to lecture
- Build several components of an OS
- “Komplexpraktikum” for (Media) Computer Science students
- “Internship” for Computational Engineering
- starts on Tuesday, Oct 26th, 14:50

TECHNISCHE UNIVERSITÄT DRESDEN

Schedule

Date	Lecture	Exercise
Oct 12	**Intro**	
Oct 19	**Tasks, Threads, Synchronization**	Paper: Nucleus of an MP system
Oct 26	**Memory**	
Nov 2	**Communication**	Practical: Booting
Nov 9	**Real-Time**	
Nov 16	**Device Drivers**	Paper: Singularity OS
Nov 23		
Nov 30	**Resource Management**	Practical: IPC
Dec 7	**Virtualization**	
Dec 14	**Legacy Containers**	Paper: Formal req. on virtualization
Dec 21	**Security Fundamentals**	
Jan 11	**Information Flow**	Paper: Cap myths demolished
Jan 18	**Secure Systems**	
Jan 25	**Trusted Computing**	Practical: Capability Systems
Feb 1	**Debugging Operating Systems**	

What's an Operating System good for?

Reliable

Secure

Comprehensible

Isolated

Centralized

Resource Management

Easy-to-use

Maintainable

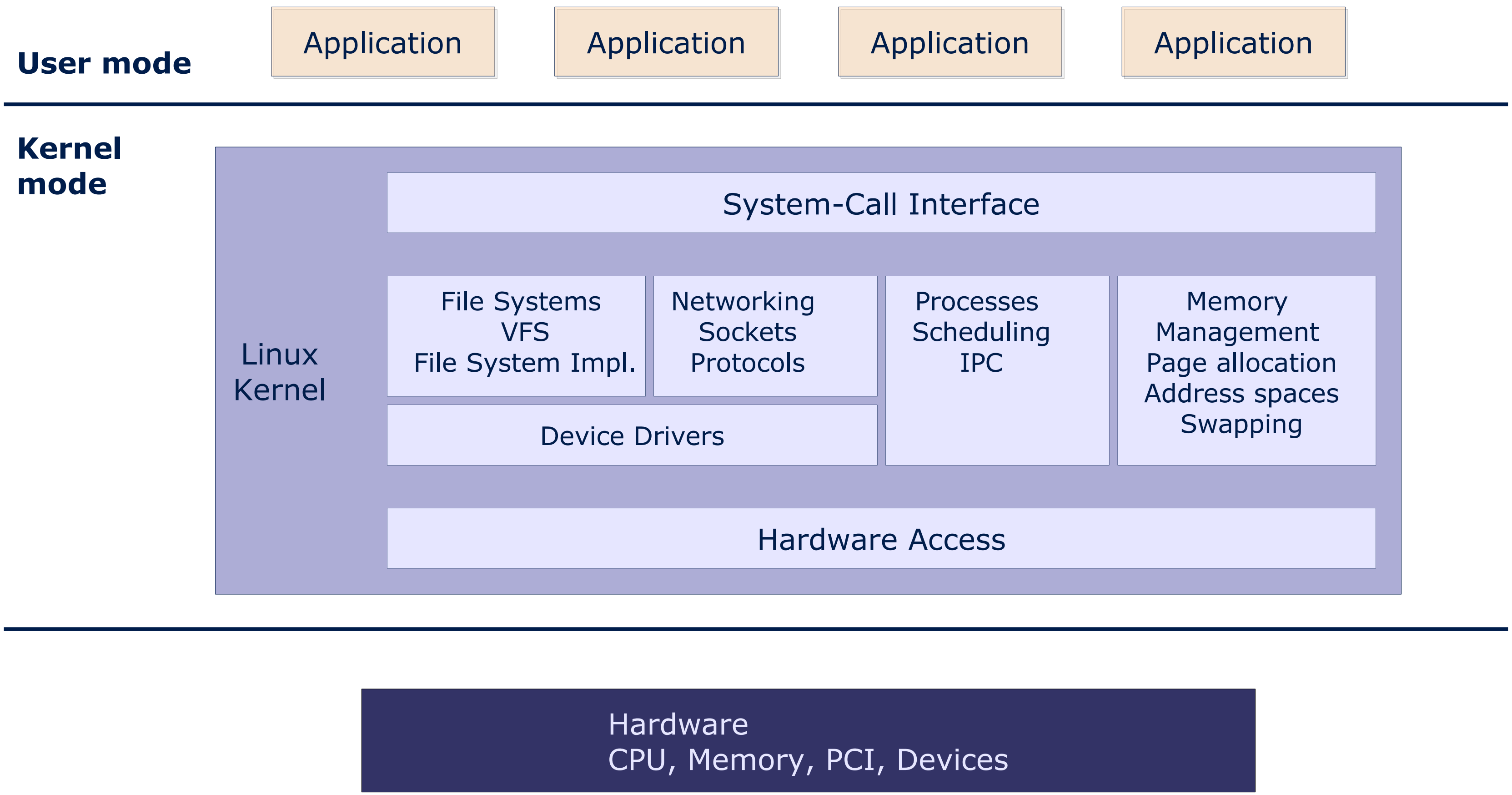
Distributed

Fast

Safe

TECHNISCHE UNIVERSITÄT DRESDEN

Monolithic kernels - Linux

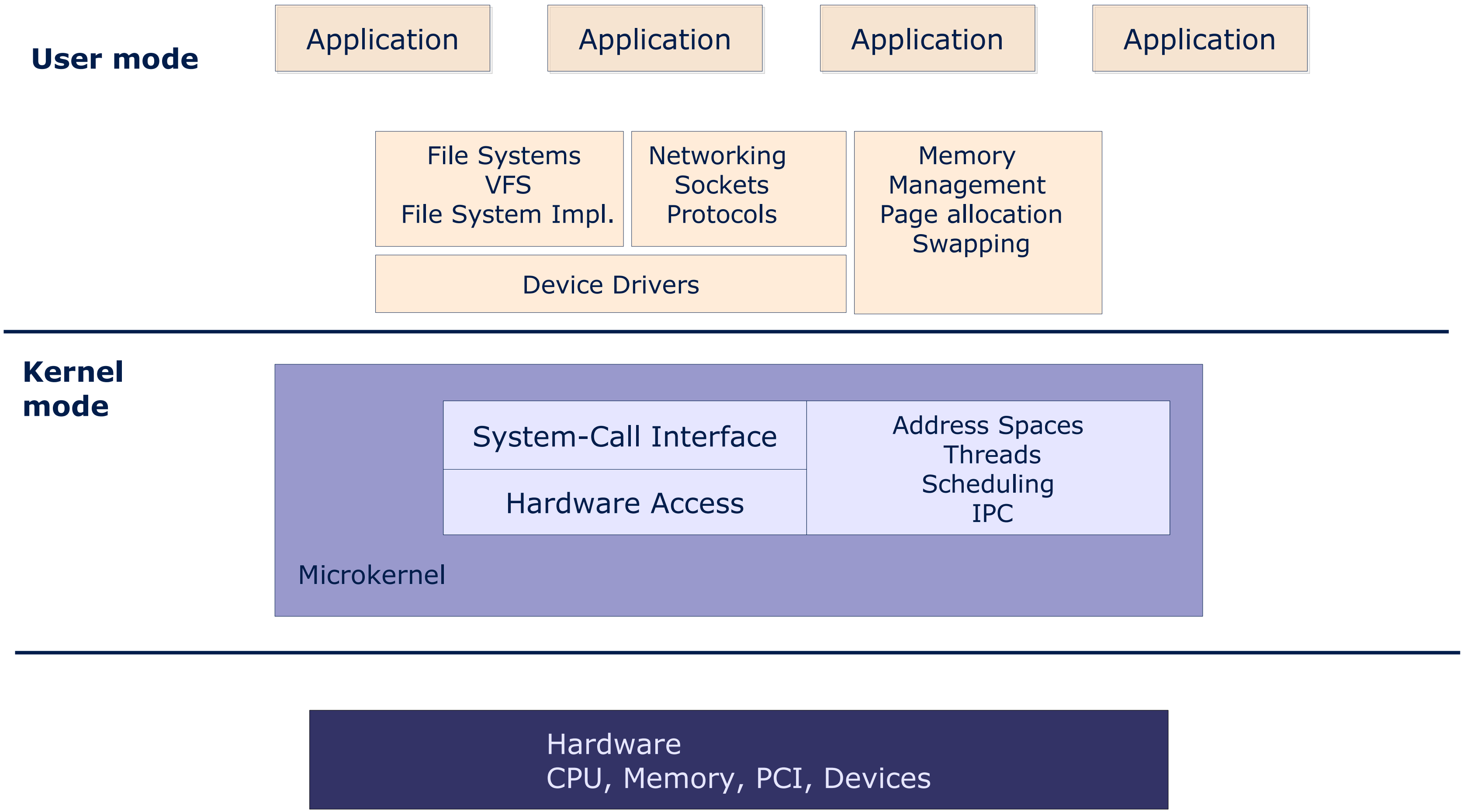


TECHNISCHE UNIVERSITÄT DRESDEN

What's the problem?

- All system components in privileged mode.
- No built-in isolation
  - Faulty driver crashes the whole system.
  - More then 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



One vision - microkernels

- 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

One vision – microkernels (2)

- 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

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)
  - …

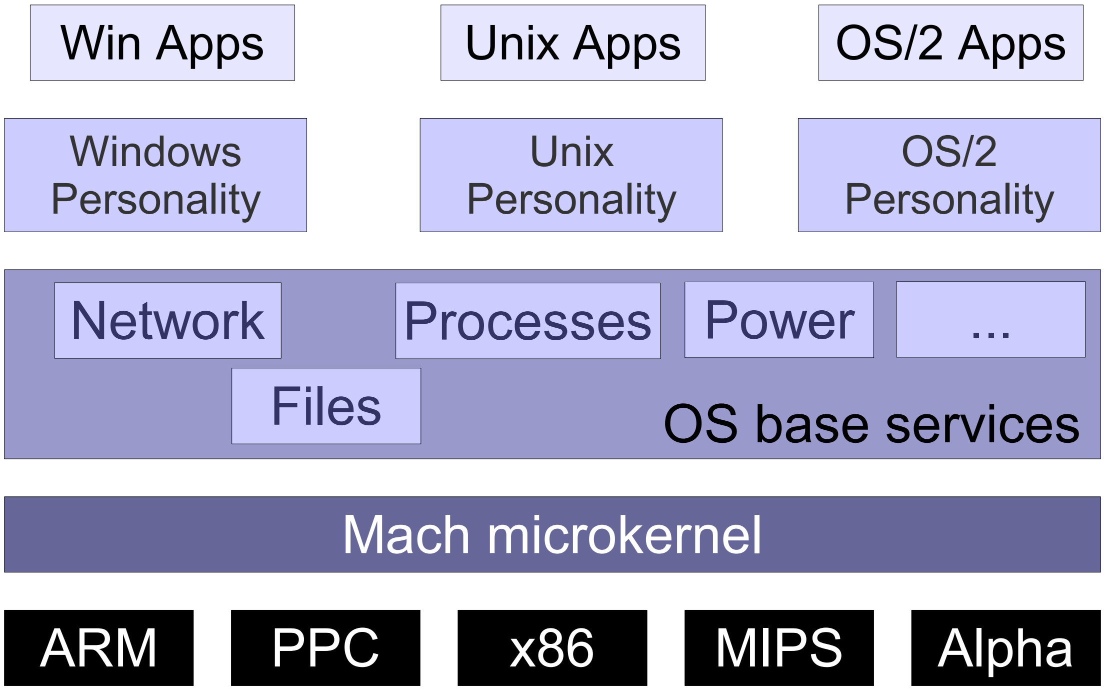
- Foundation for several real systems
  - Single Server Unix (BSD4.3 on Mach)
  - MkLinux (OSF)
  - IBM Workplace OS
  - NeXT OS → Mac OS X

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
- POSIX-compatibility
- Shortcomings
  - performance
  - drivers still in the kernel

Case study: IBM Workplace OS

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



IBM Workplace OS (2)

- 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

Lessons learned

- 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

Proven advantages

- Subsystem protection / isolation
- Code size
  - Fiasco kernel: ~ 15,000 LoC
  - Minimal application:
    (boot loader + “hello world”):
    ~ 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

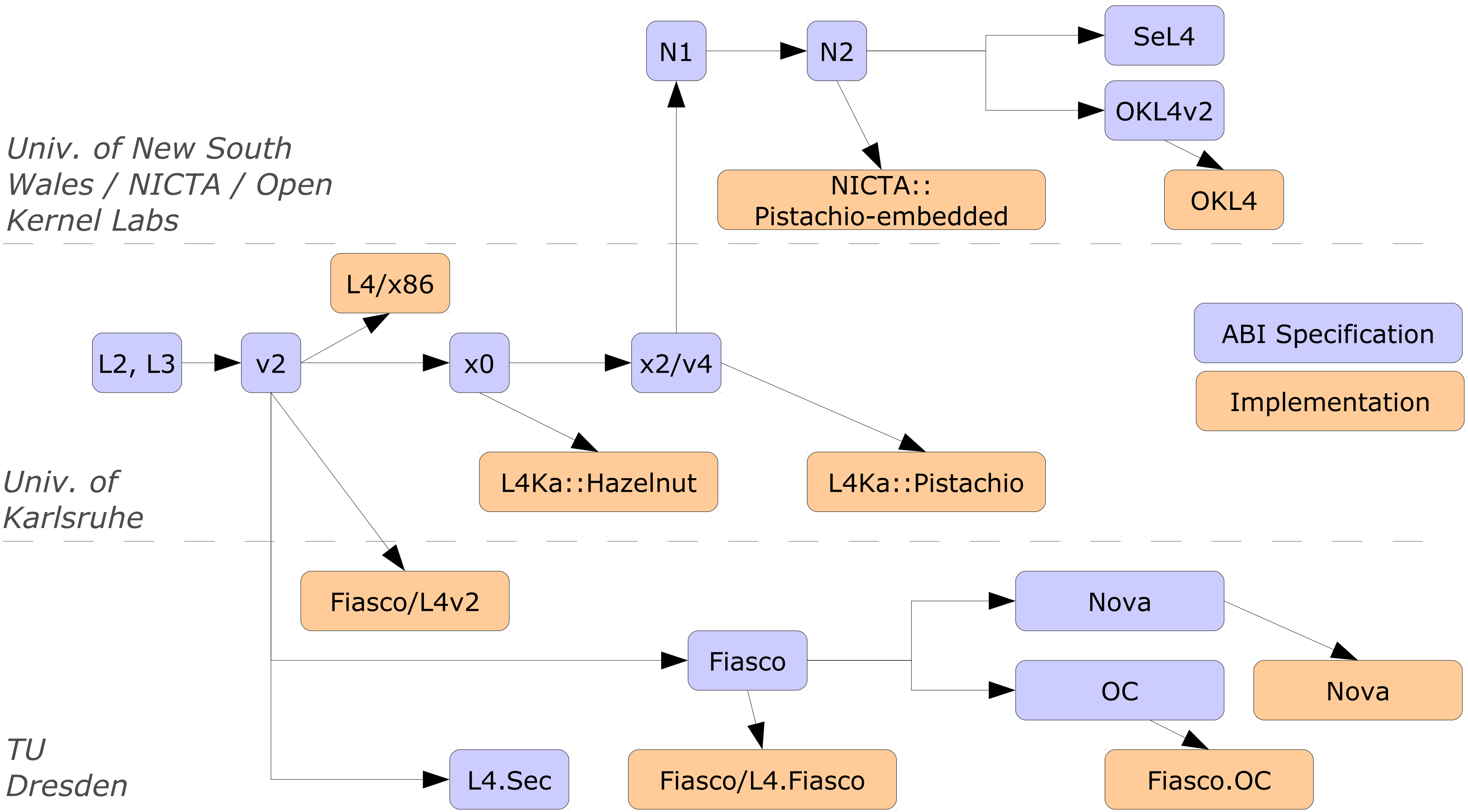
Challenges

- 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's out there?

- Minix @ VU 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



TECHNISCHE UNIVERSITÄT DRESDEN

L4 concepts

- 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)

Fiasco.OC: System Calls

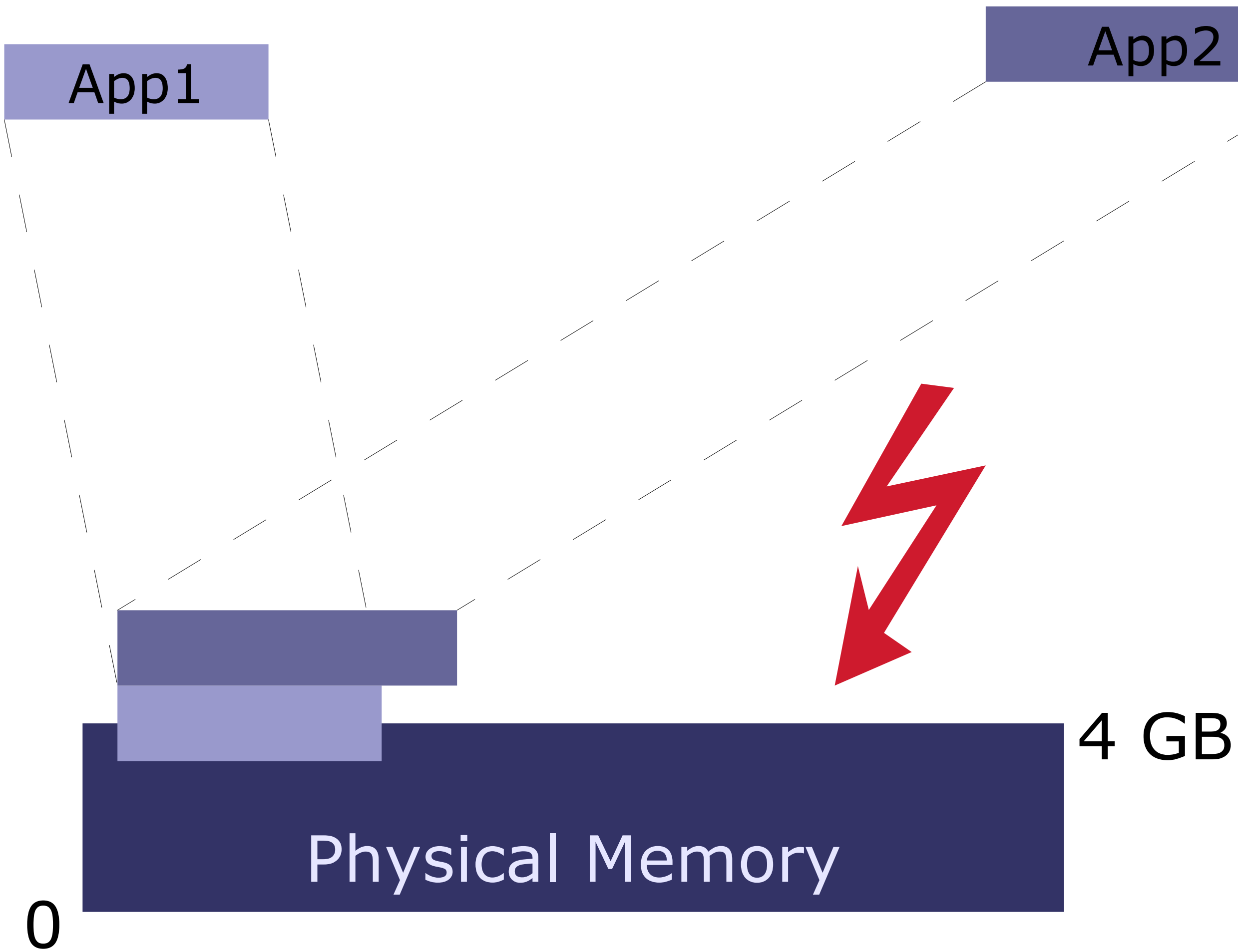
- "Everything is an object."
  - Objects posess internal state & behavior.
- 1 system call: *invoke_object()*
  - Parameters passed in UTCB
  - Types of parameters depend on type of object
- Objects referenced by capabilities
  - *invoke()* allowed for everyone possessing a capability to the object
  - Kernel mechanism: unforgeable
  - Can be mapped just like every other resource.

Fiasco.OC: Object types

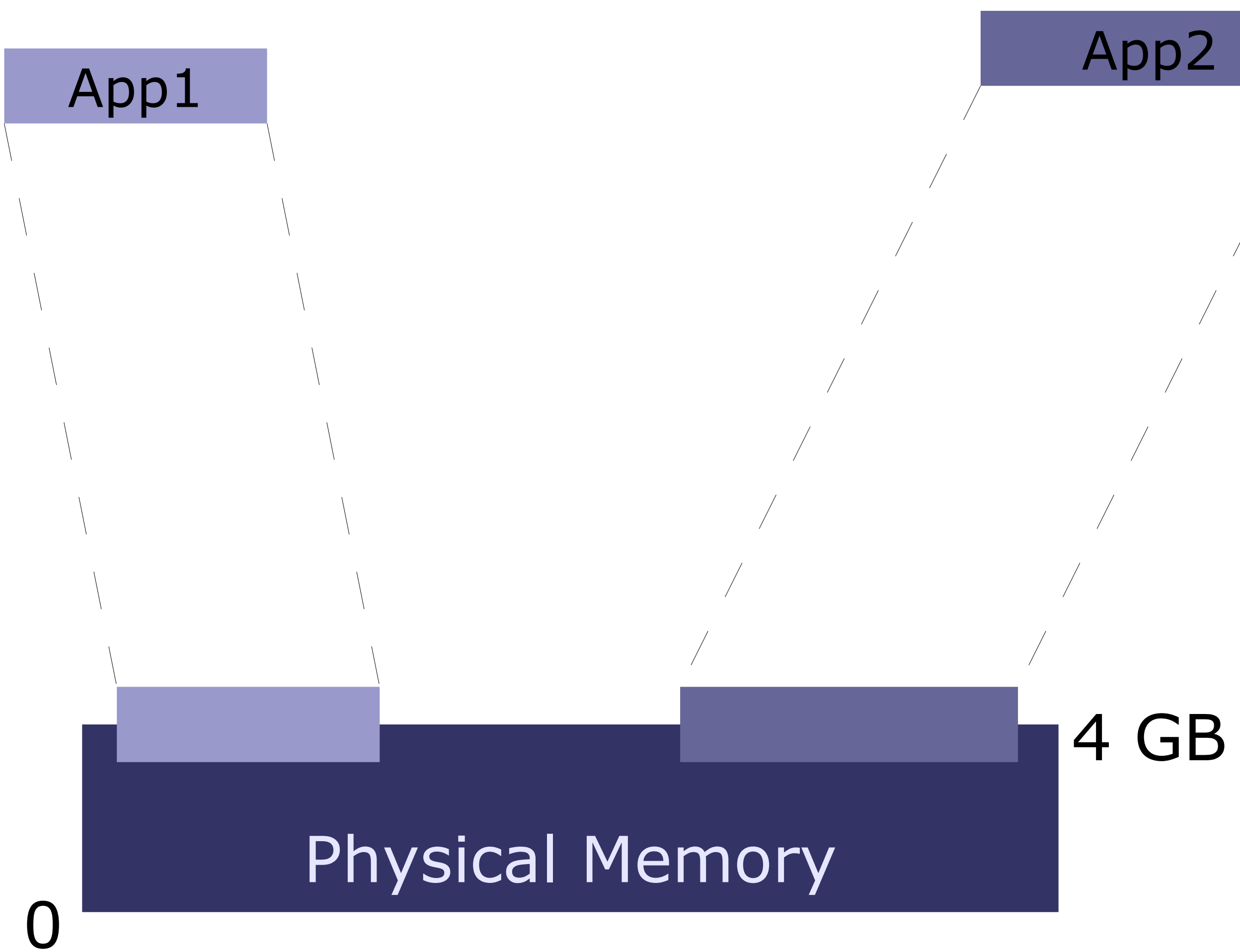
- Factory
  - Create other objects
  - Enforce resource quotas
- Task
  - Address + capability space
- Thread
- IPC Gate
- IRQ

- Each task gets initial set of capabilities upon startup.

Problem: Memory partitioning



Solution: Virtual Memory

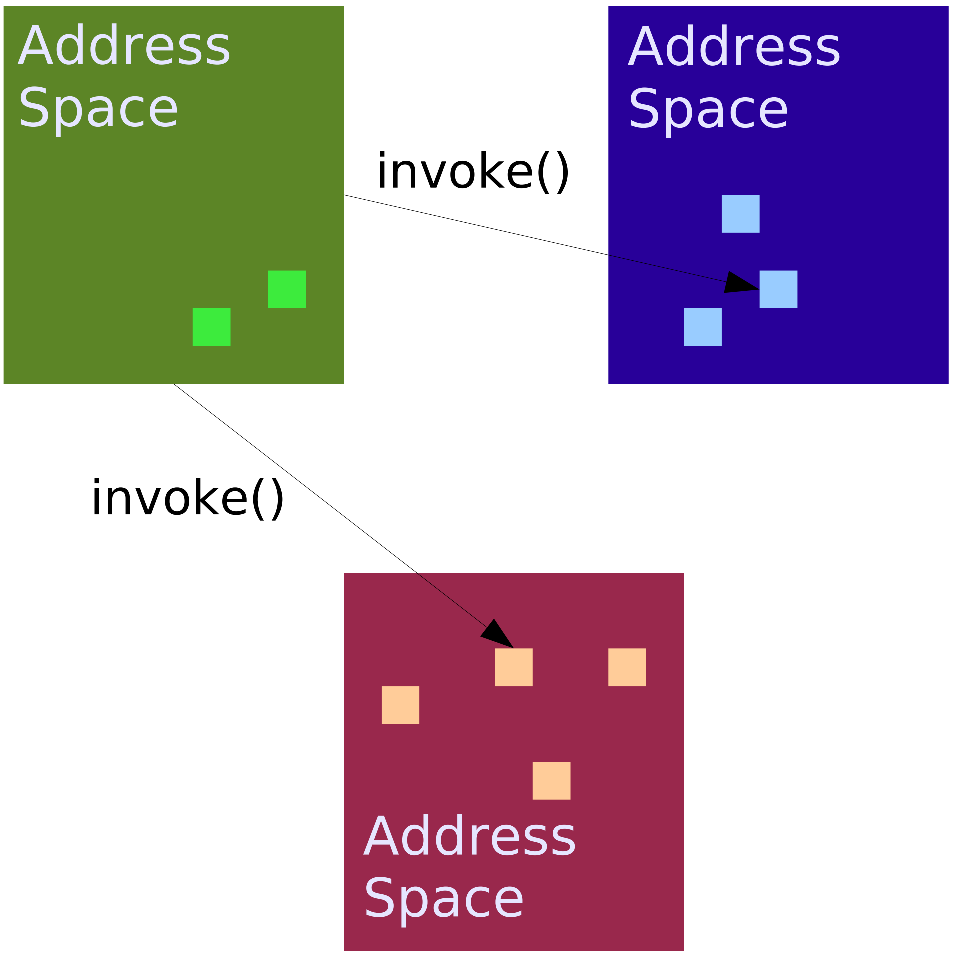


Resource partitioning

- 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

TECHNISCHE UNIVERSITÄT DRESDEN

Fiasco.OC: Partitioning objects



- 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?

Fiasco.OC: Local names

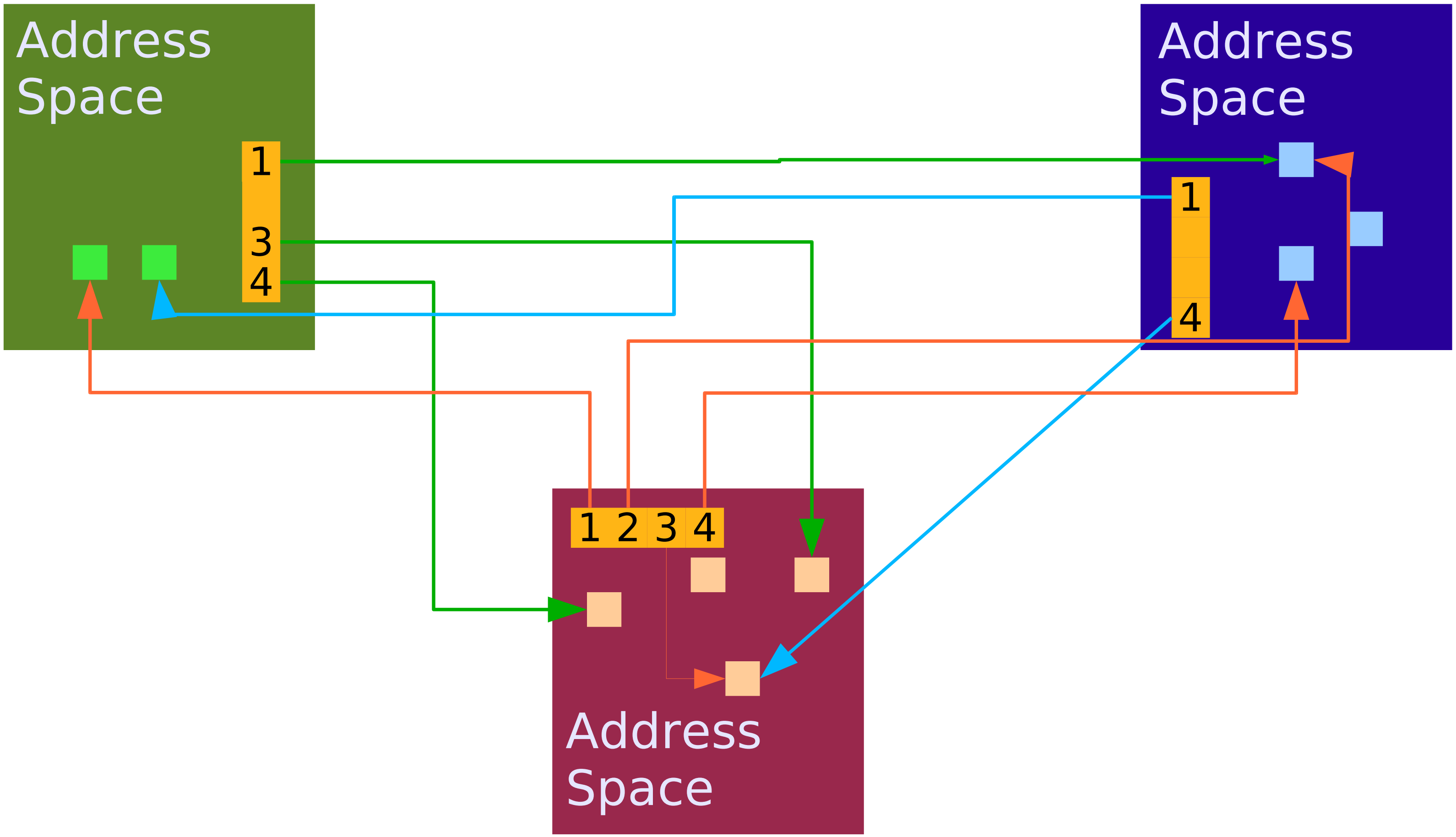
- Global object IDs are
  - insecure (forgery, covert channels).
  - inconvenient (programmer needs to know about partitioning in advance)

- Solution in Fiasco.OC: task-local *capability space* as an indirection

- Per-task name space (configured by task's creator) to map physical names (cap references) to object capabilities.

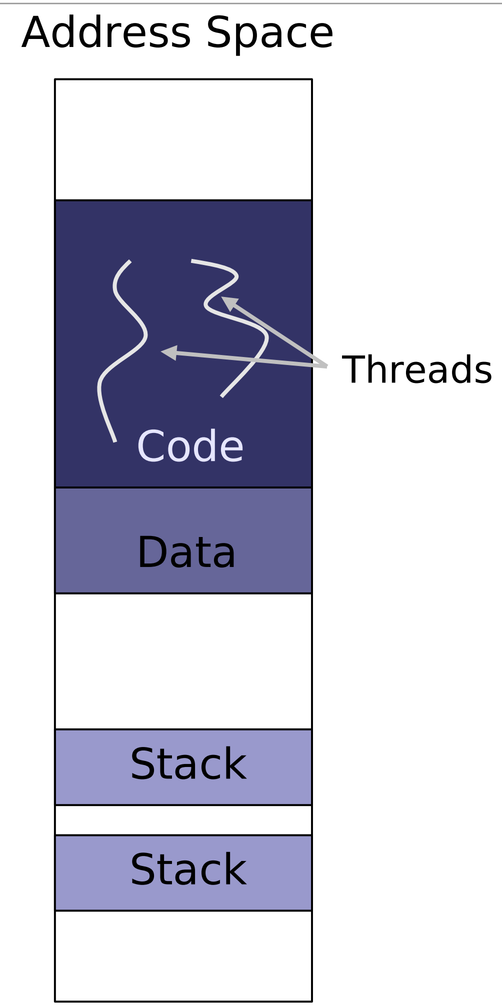
Capabilities / Local Names

Indirection allows for security and flexibility.



L4 - Threads

- 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
  - ...



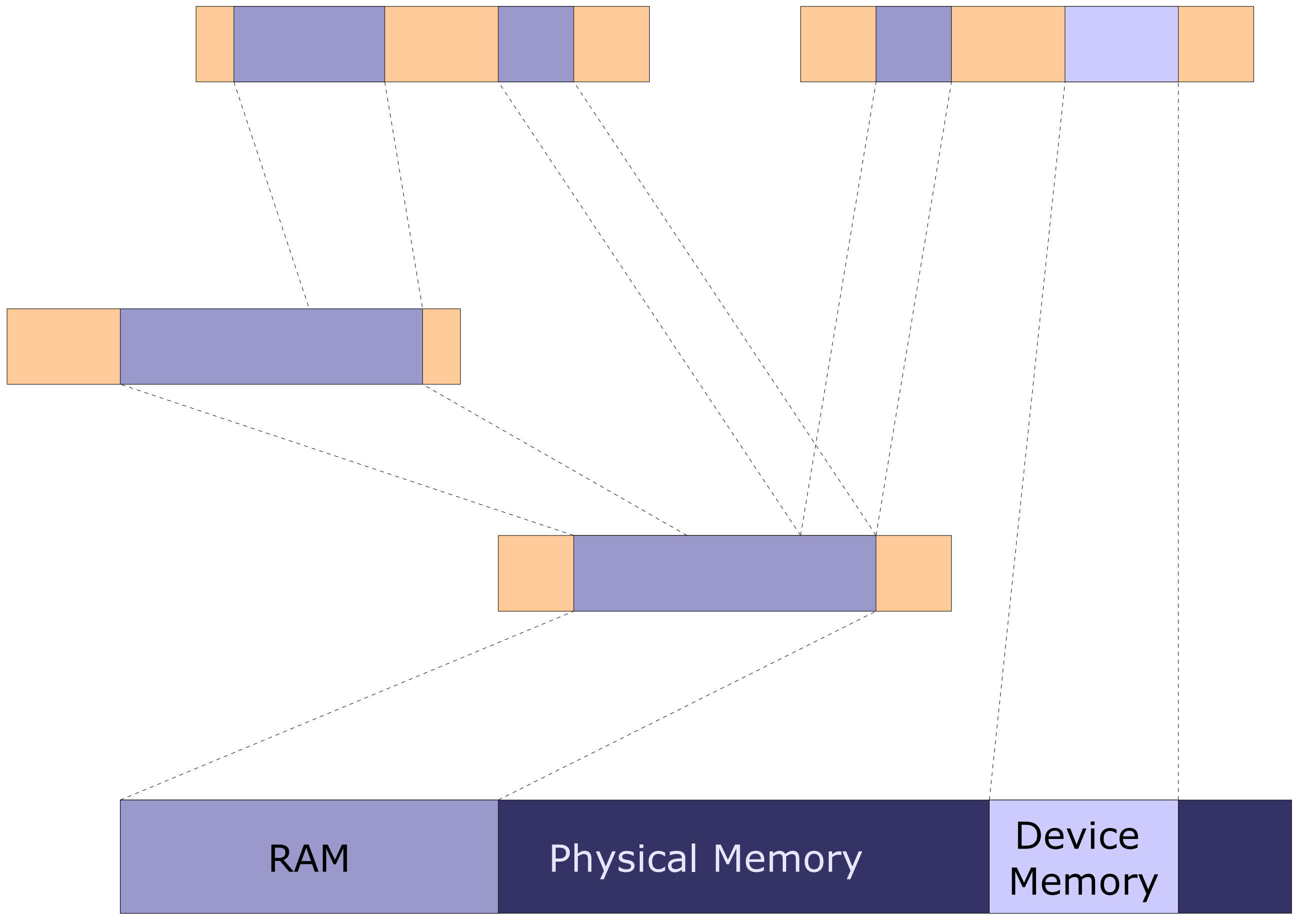
L4 - Communication

- 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

L4 – Resource Mappings

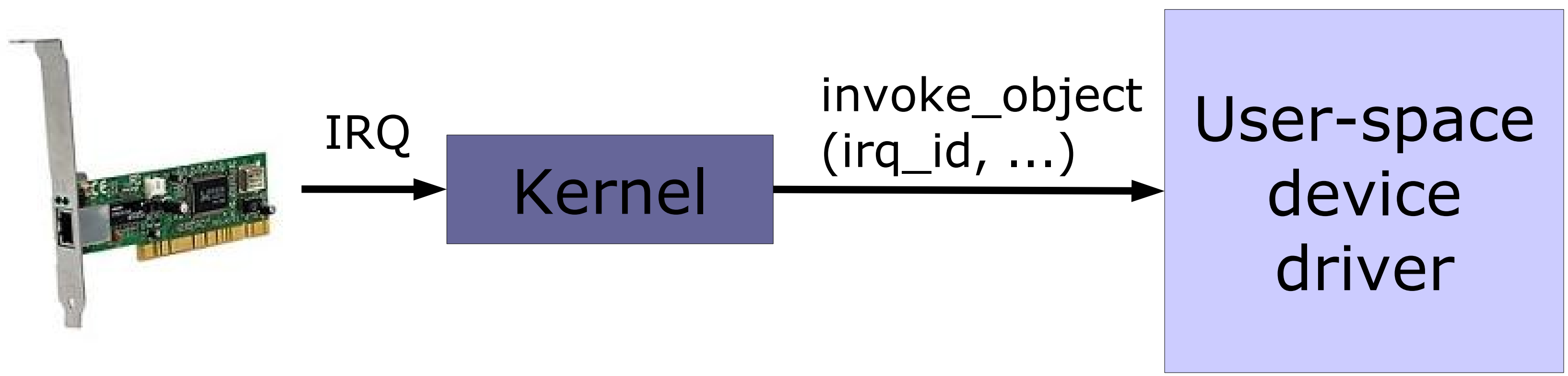
- If a thread has access to a capability, it can map this one to another thread.
- Abstraction for mapping: *flexpage*
- Flexpages describe mapping
  - location and size of resource
  - receiver's rights (read-only, mappable)
  - type (memory, IO, communication capability)

L4 – Recursive address spaces

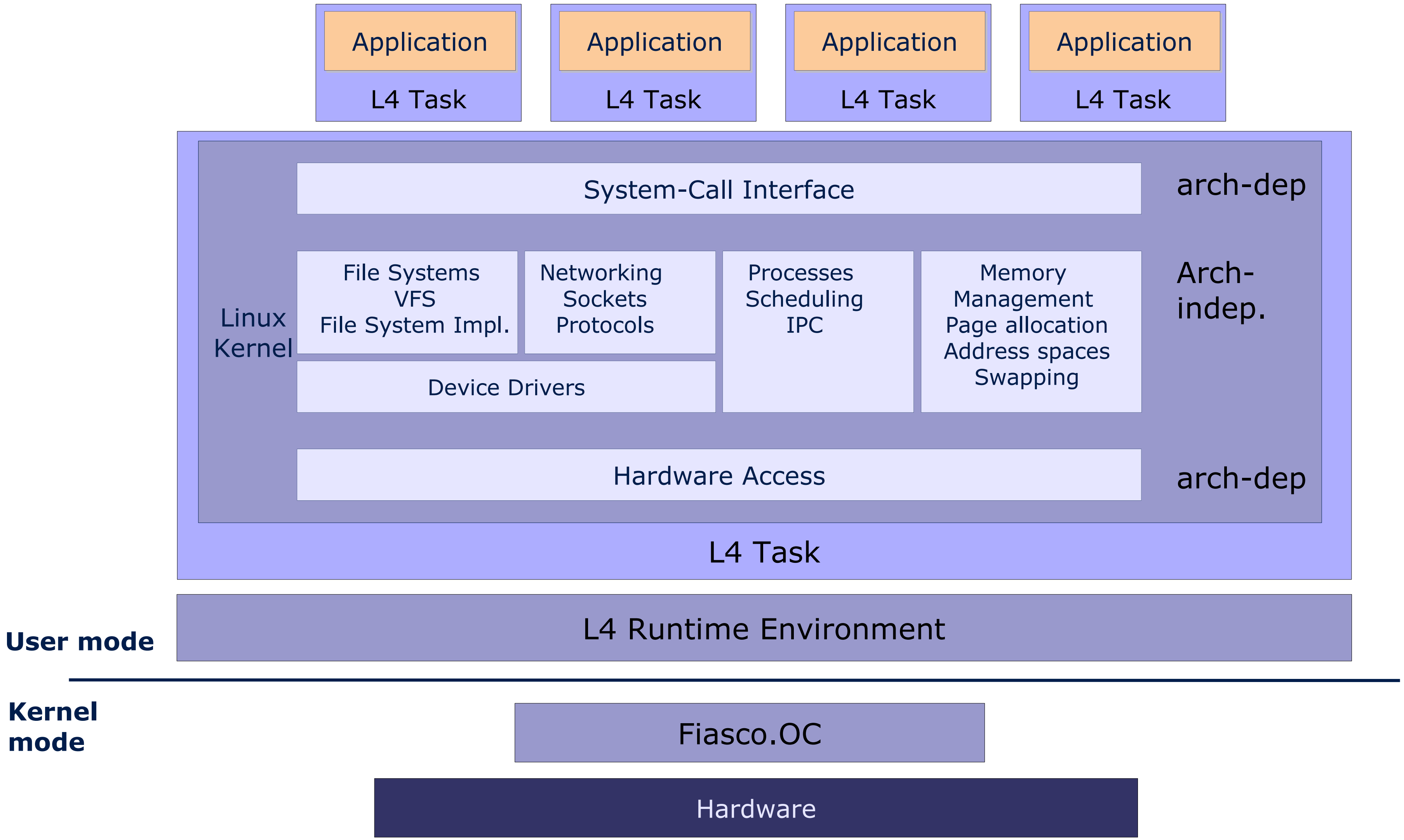


L4 – Hardware Interrupts

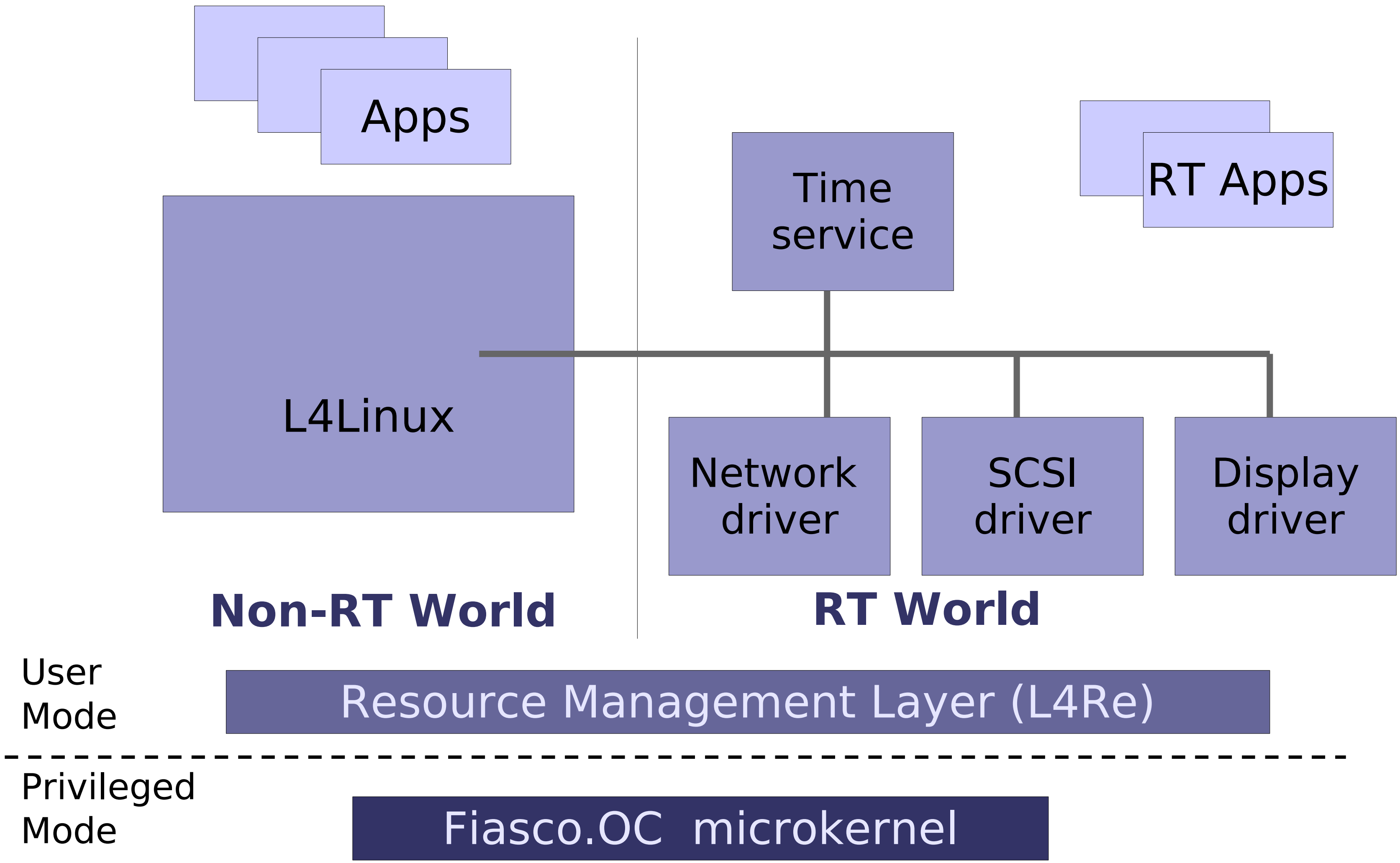
- Kernel object: IRQ
- Used for hardware and software interrupts (read: asynchronous signals)
- Wait for IRQ: *invoke_object(irq)*



Linux on L4

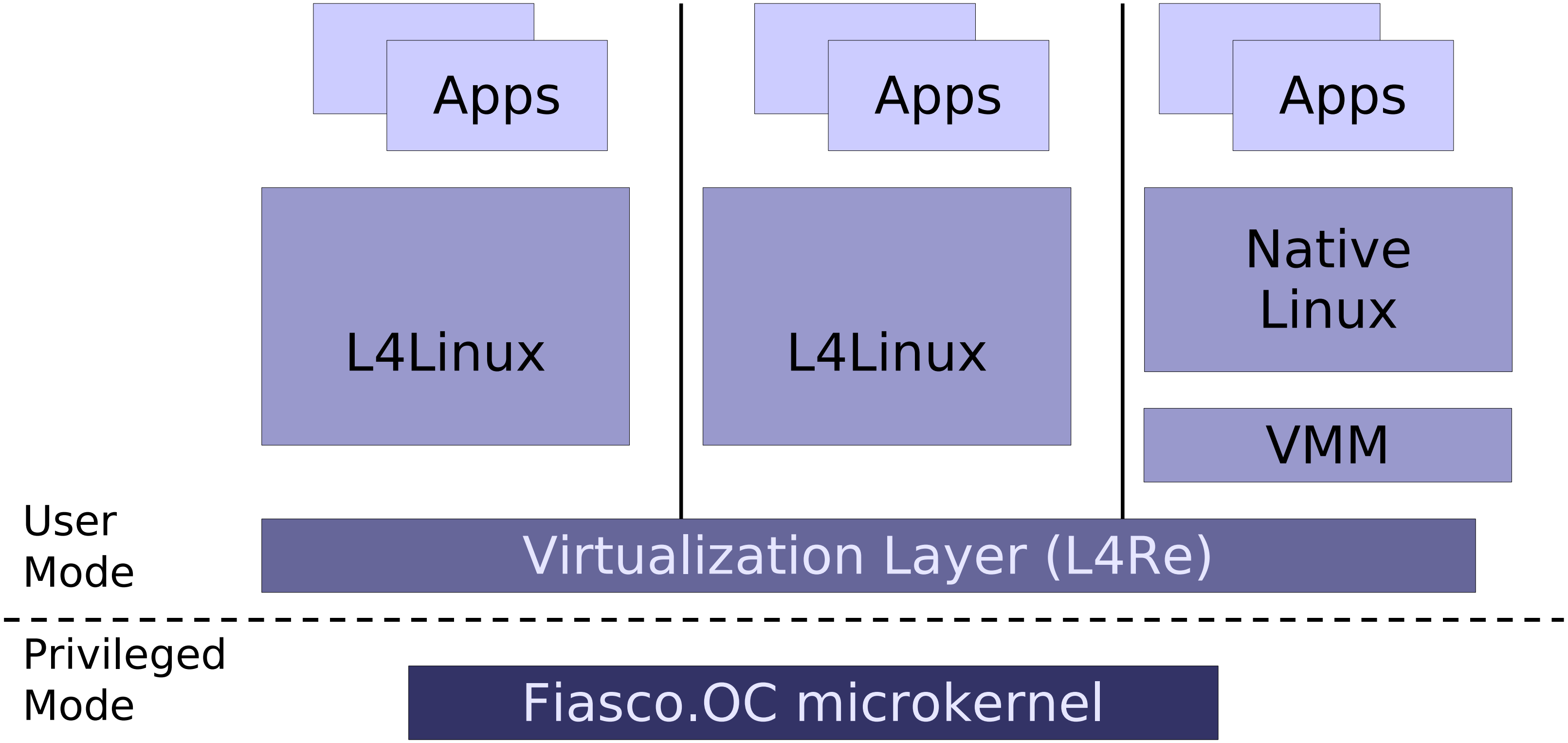

Application
L4 Task
Application
L4 Task
Application
L4 Task
Application
L4 Task
System-Call Interface
arch-dep
Linux Kernel
File Systems
VFS
File System Impl.
Networking
Sockets
Protocols
Processes
Scheduling
IPC
Memory Management
Page allocation
Address spaces
Swapping
Arch-indep.
Device Drivers
Hardware Access
arch-dep
L4 Task
User mode
L4 Runtime Environment
Kernel mode
Fiasco.OC
Hardware

The Dresden Real-Time Operating System



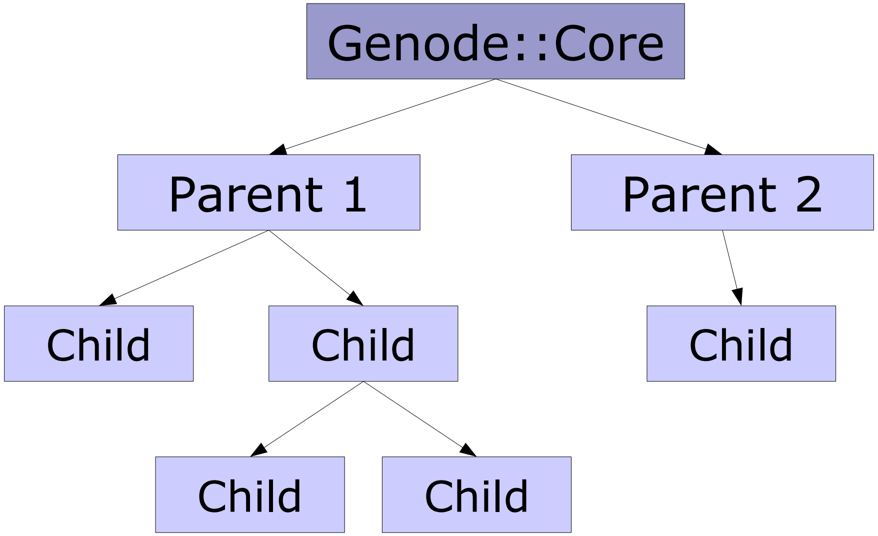
Virtual machines

- 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:
  - "Tasks, Threads and Synchronization" on Oct 19^{th}
- Next exercise:
  - Oct 19^{th}
  - Brinch-Hansen: "*The nucleus of a multiprogramming system*"