



Microkernel-based Operating Systems —Introduction—

Jan Bierbaum
Carsten Weinhold

Dresden, October 13th 2026

- 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 particular

- Slides: <https://tud.de/inf/os> → ≡ → Studies → Lectures → MOS
- Mailing list:
<https://os.inf.tu-dresden.de/mailman/listinfo/mos2026>
- Matrix: <https://matrix.to/#/#inf-os-mos:tu-dresden.de>

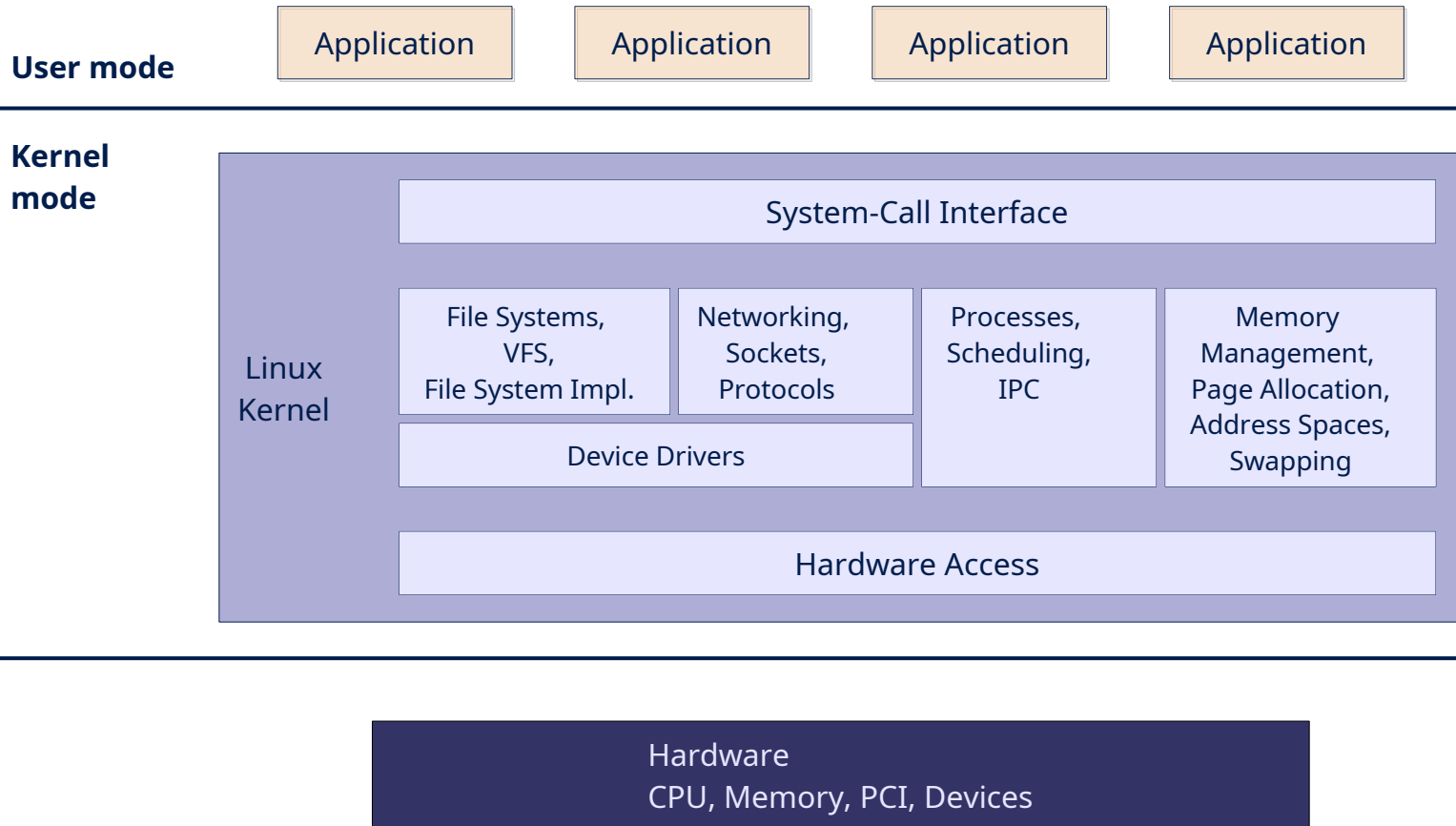
- Lecture:
 - 2 SWS
 - Tuesday, 14:50, APB E065
 - (Mostly) hybrid via BBB
- Exercises:
 - 4 SWS
 - Tuesday, 16:40, APB E065

- Announced on website and mailing list
- Practical exercises
 - Implement aspects of an MOS on your own
 - First one *today* after the lecture
- Paper reading exercises
 - Read a paper *beforehand*
 - Prepare questions/observations
 - Actively participate in the discussion
 - First one *next week (Oct 20th)*

- Oral, with Dr. Roitzsch, after the semester
 - Don't miss the *exam registration deadline* with the examination office!
 - Sign up for an appointment: <https://tud.de/inf/os> → 
→ Studies → Exams
 - Expect delays due to vacation period, etc.
 - Language: German or English (or mixture)
 - Withdrawal until 3 days before the appointment — make sure to de-register with “selma” *and* to cancel your appointment
- Topics: Lecture *and* exercise content

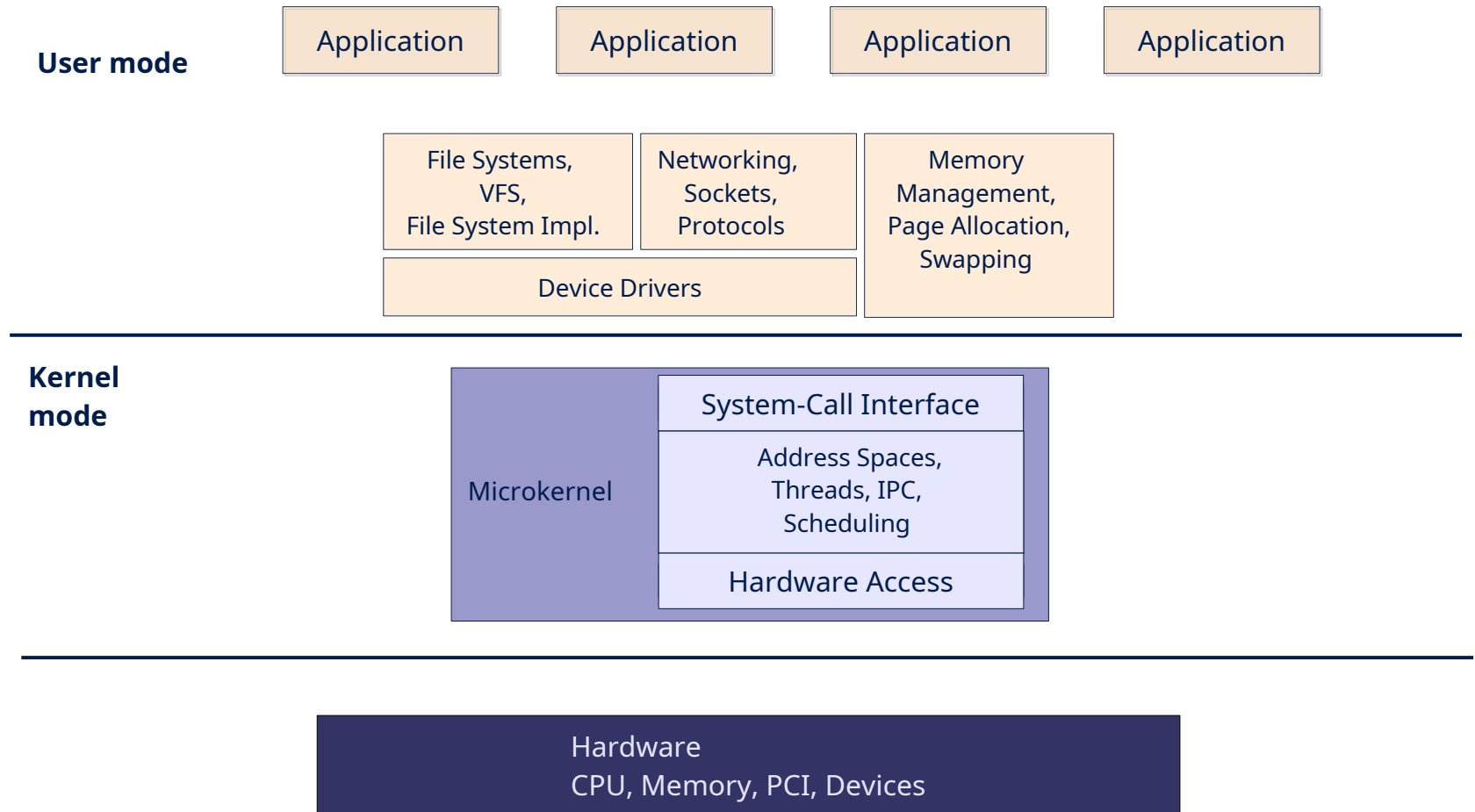
Monoliths vs. Microkernels

- Manage the available resources
 - Hardware (CPU, memory, storage, network, ...)
 - 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 or HW-specific operations
 - x86: ring 0 vs. ring 3
 - Device drivers
- Provide separation and means for collaboration
 - Isolate users / processes from each other
 - Allow cooperation if needed (e.g., sending messages between processes)



- Security issues
 - All components 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 (less code → fewer errors)
 - 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

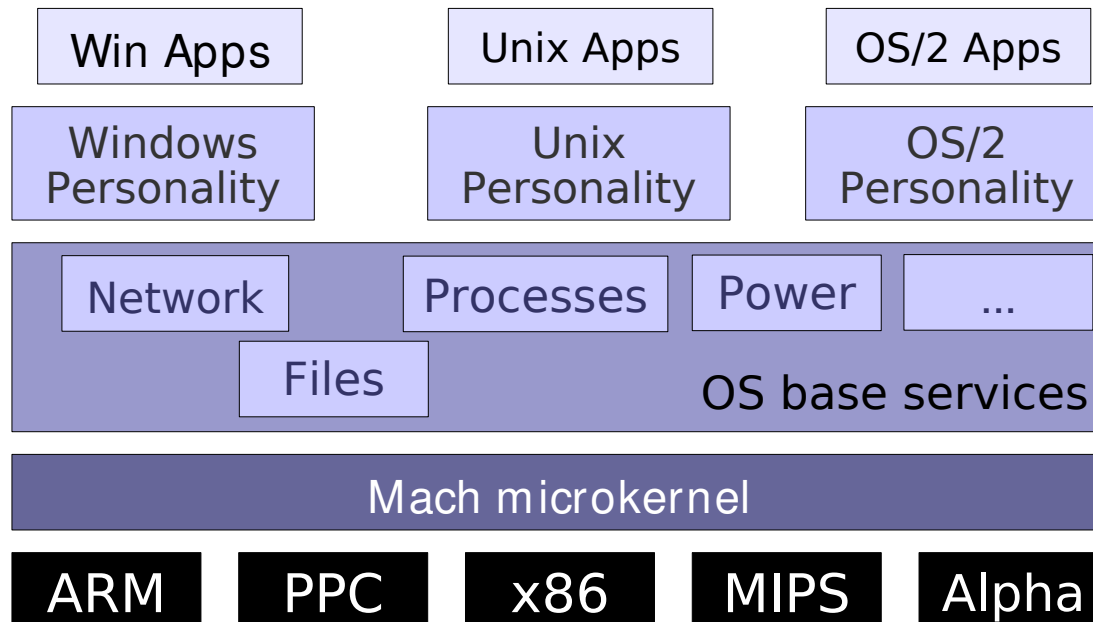


- OS personalities
- Customisability
 - Servers may be configured to suit the target system (small embedded systems, desktop PCs, SMP systems, ...)
 - Remove unnecessary servers
- Enforce reasonable system design
 - Well-defined interfaces between components
 - Access to components only via these interfaces
 - Improved maintainability

- Developed at CMU, 1985 – 1994
 - Rick Rashid (former head of MS Research)
 - Avie Tevanian (former Apple CTO)
 - Brian Bershad (professor @ Univ. of Washington)
 - ...
- Foundation for several real systems
 - Single Server Unix (BSD4.3 on Mach)
 - MkLinux (OSF, Apple)
 - IBM Workplace OS
 - NeXTSTEP

- 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 in the kernel

- Main goals:
 - Multiple OS personalities
 - Support for multiple HW architectures



- Never finished ... but almost 2 billion US-\$ spent
- Causes of failure:
 - Underestimated difficulties in creating OS personalities
 - Management errors: divisions forced 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 (e.g., 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 (generated using David A. Wheeler's "SLOCCount")
 - Microkernel-based OS, x86 architecture
 - Fiasco kernel: ~34k LoC
 - "HelloWorld" (+ boot loader + root task): ~10k LoC
 - Linux 6.17, x86 architecture
 - Kernel (kernel + arch/x86): ~600k LoC
 - Subsystems (net, fs, sound, ...): ~3.7M LoC
 - Drivers: ~20M LoC
- Customisability
 - Tailored memory management / scheduling / ... algorithms
 - Adaptable to embedded / real-time / secure / ... systems

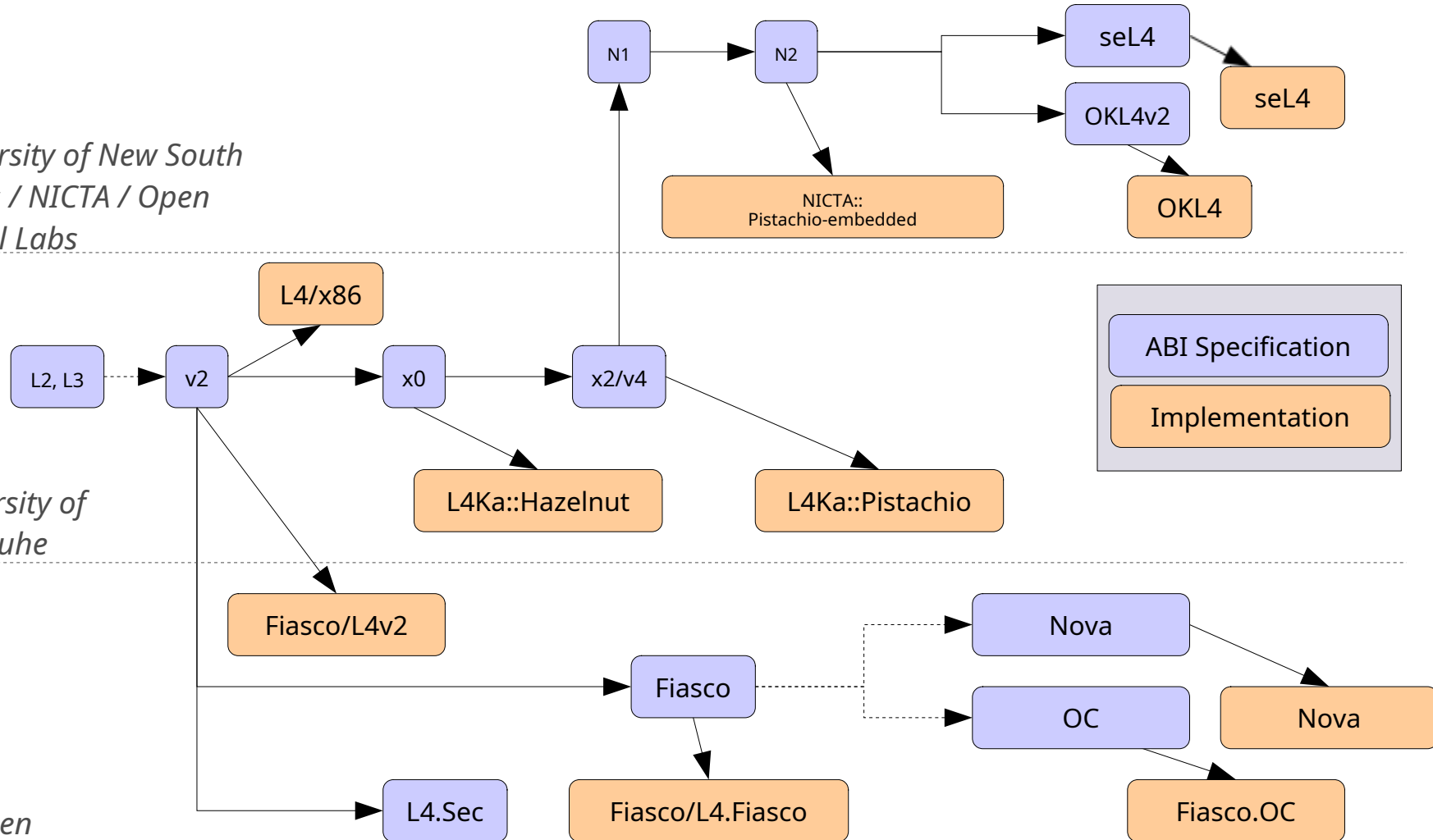
- We need fast and efficient kernels
 - Covered in the “Microkernel Construction” lecture in the summer term
- We need fast and efficient OS services
 - Memory and resource management
 - Synchronisation
 - Device Drivers
 - File systems
 - Communication interfaces
 - Subject of this lecture

- 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

University of New South
Wales / NICTA / Open
Kernel Labs

University of
Karlsruhe

TU
Dresden



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

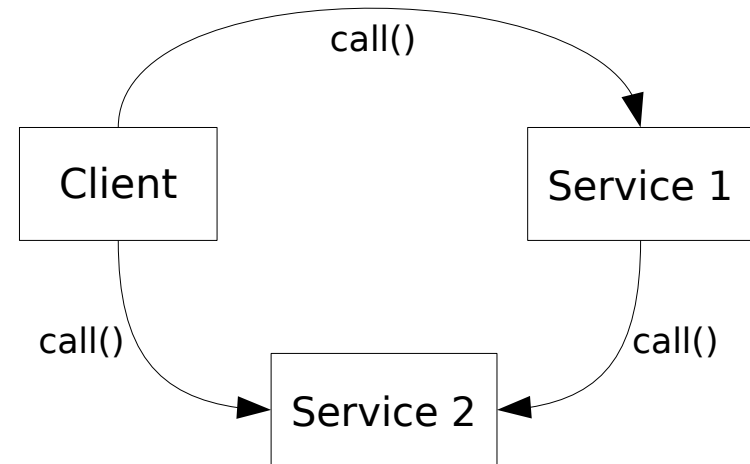
Taking a closer look at L4

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

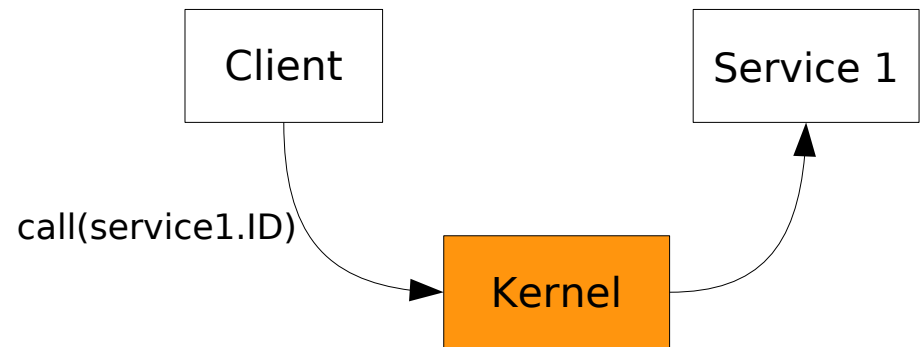
- “Everything is an object”
 - Task Address space
 - Thread Activities, scheduling
 - IPC Gate Communication, resource mapping
 - IRQ Communication
 - Factory Create other objects, enforce quotas
- One system call: **invoke_object()**
 - Parameters passed in UTCB
 - Types of parameters depend on type of object

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

- Build everything above kernel using user-level objects that provide a service
 - Networking stack
 - File system
 - ...
- Kernel provides
 - Object creation/management
 - Object interaction: Inter-Process Communication (IPC)



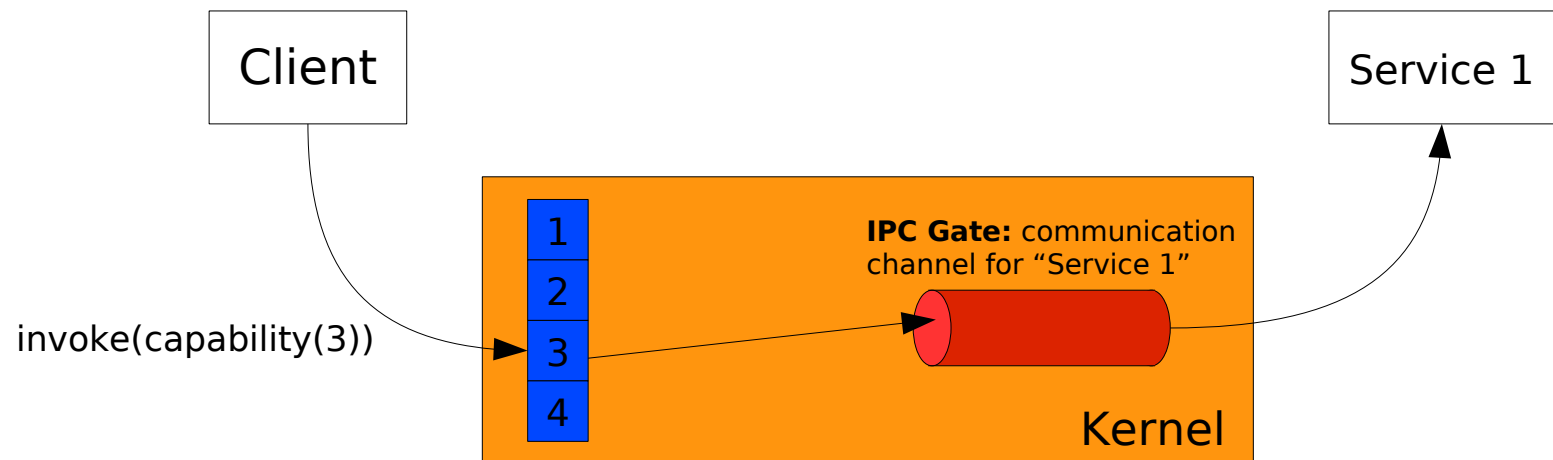
- To call an object, we need an address:
 - Telephone number
 - Postal address
 - IP address
 - ...



- Simple idea, isn't it?
- ID is wrong? Kernel returns ENOTEXIST
- But not so fast! This scheme is insecure:
 - Client could simply “guess” IDs brute-force
 - (Non-)Existence can be used as a covert channel

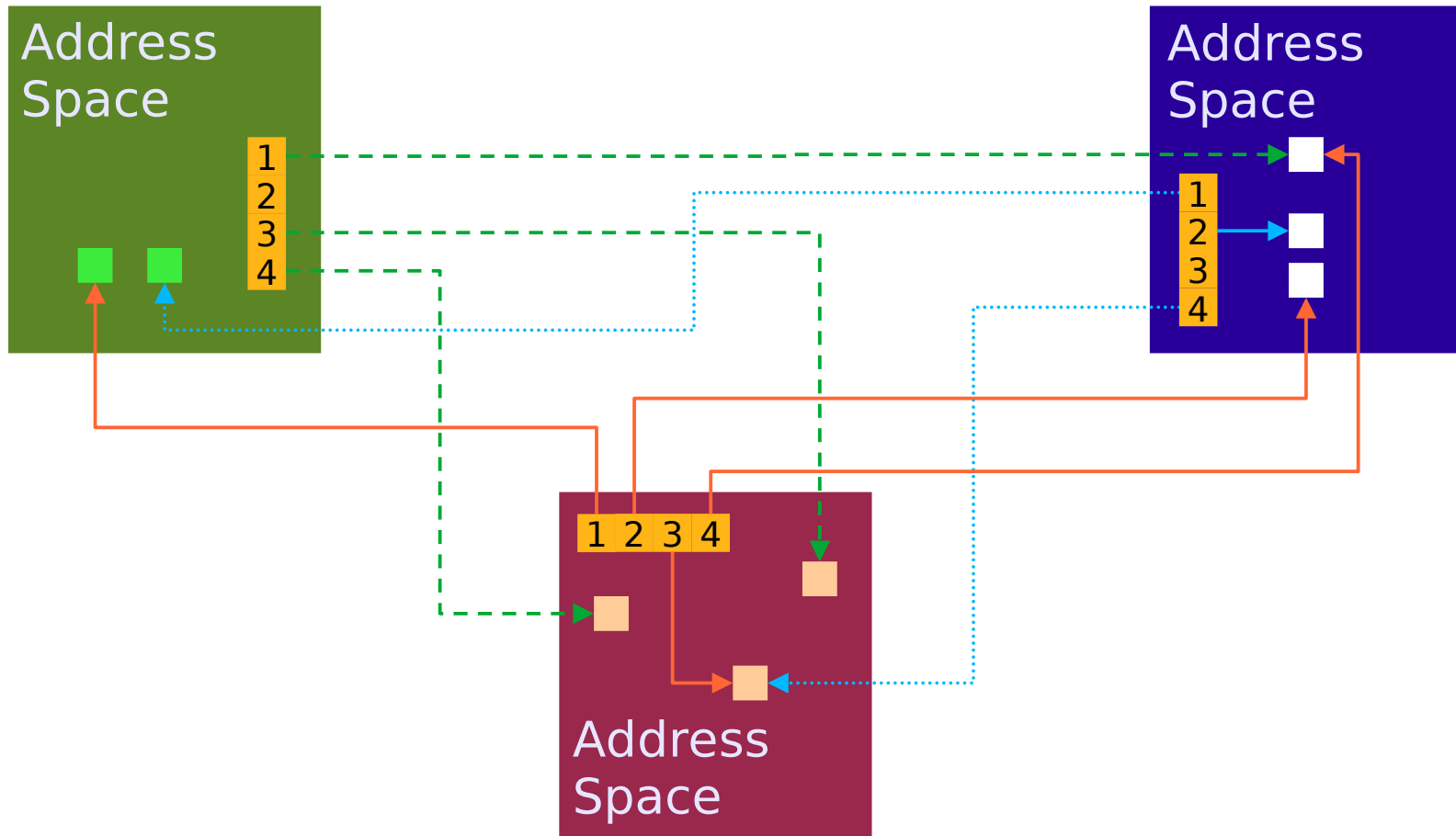
- Global object IDs are
 - insecure (forgery, covert channels)
 - inconvenient (programmer needs to know about partitioning in advance)
- Solution in Fiasco.OC
 - Task-local *capability space* as indirection
 - *Object capability* required to invoke object
 - Per-task name space
 - Maps names to object capabilities
 - Configured by task's creator

- Capability:
 - Reference to an object
 - Protected by the kernel
 - Kernel knows all capability-object mappings
 - Managed as a per-process capability table
 - User processes only use indices into this table



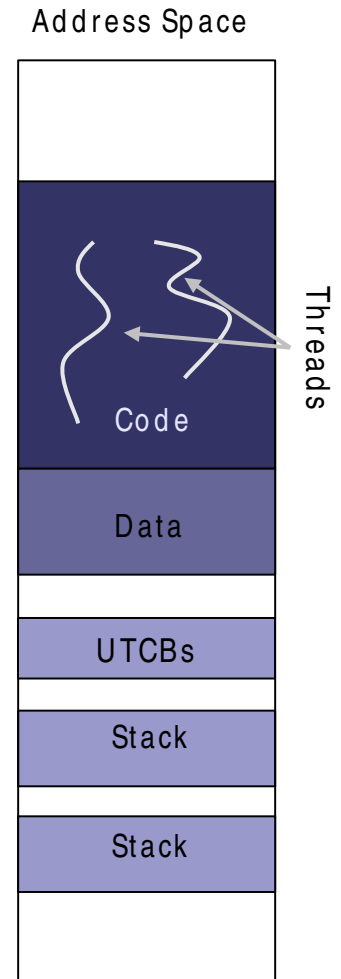
- Kernel object for communication: *IPC gate*
- Inter-process communication (IPC)
 - Between threads
 - Synchronous
- Sequence:
 - Sender writes message into its UTCB
 - Sender invokes IPC gate → blocks sender until receiver ready (i.e., waits for message)
 - Kernel copies message to receiver thread's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

Capabilities = Local Names

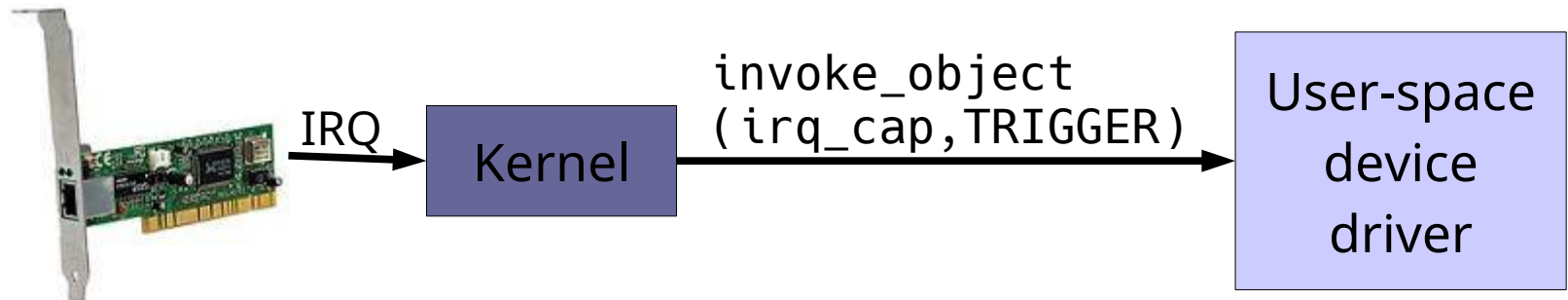


More L4 concepts

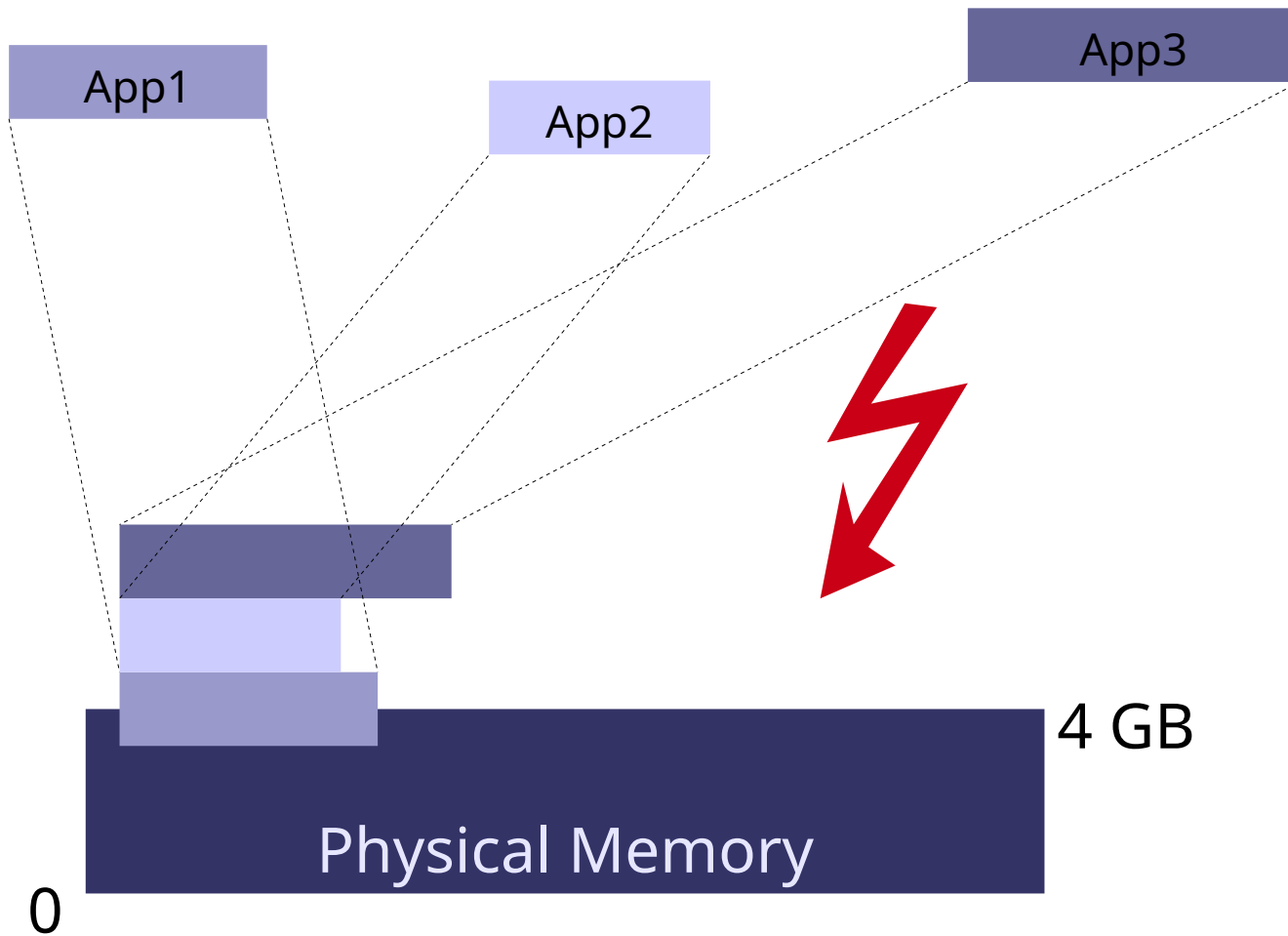
- Thread
 - Unit of execution
 - Implemented as kernel object
- Properties managed by the kernel
 - Instruction Pointer (EIP)
 - Stack (ESP)
 - Registers
 - User-level thread control block (UTCB)
- User-level applications need to
 - allocate stack memory
 - provide memory for application binary
 - find entry point
 - ...



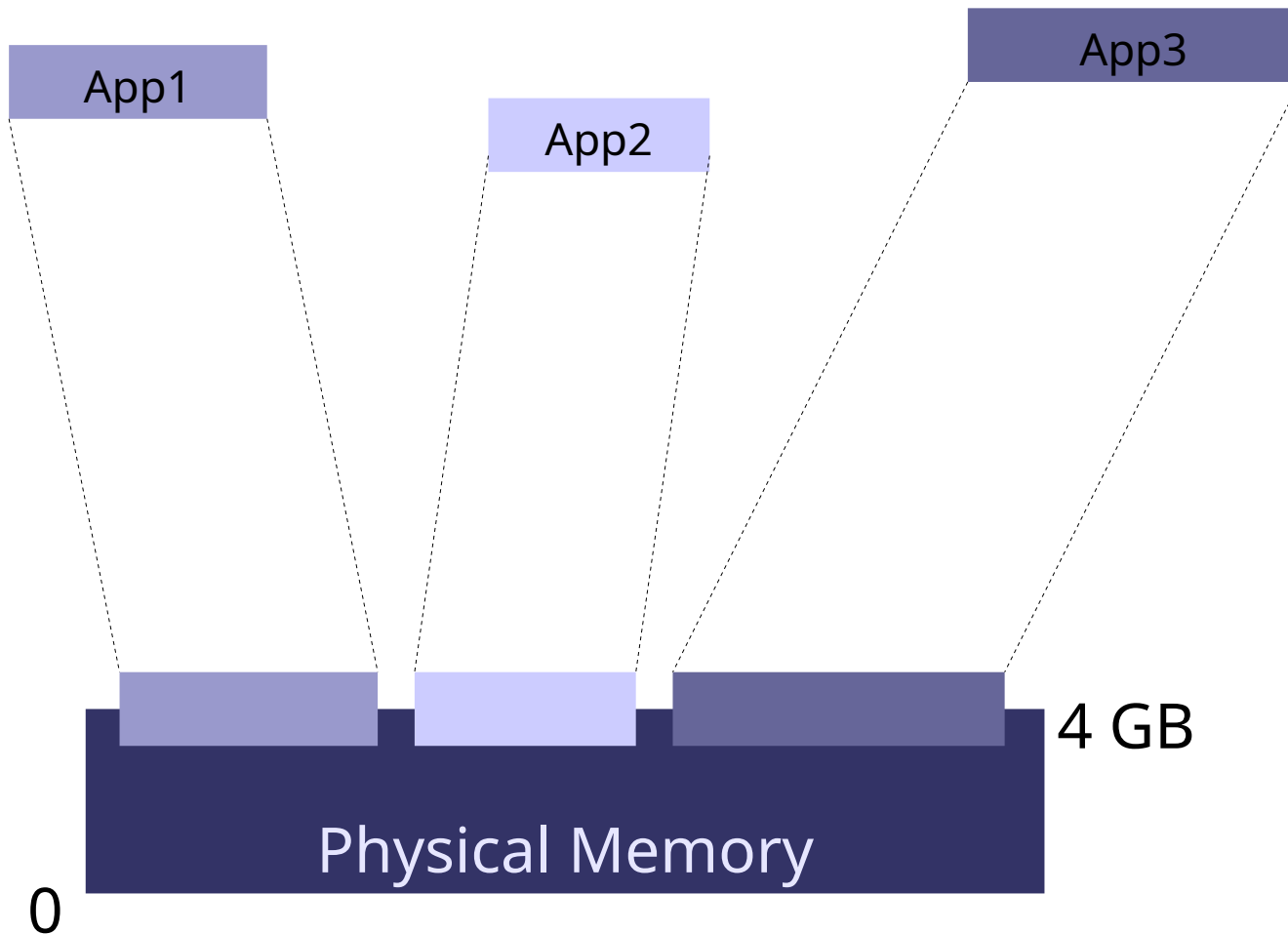
- Kernel object: IRQ
- Used for hardware and software interrupts
- Provides asynchronous signaling
 - `invoke_object(irq_cap, WAIT)`
 - `invoke_object(irq_cap, TRIGGER)`



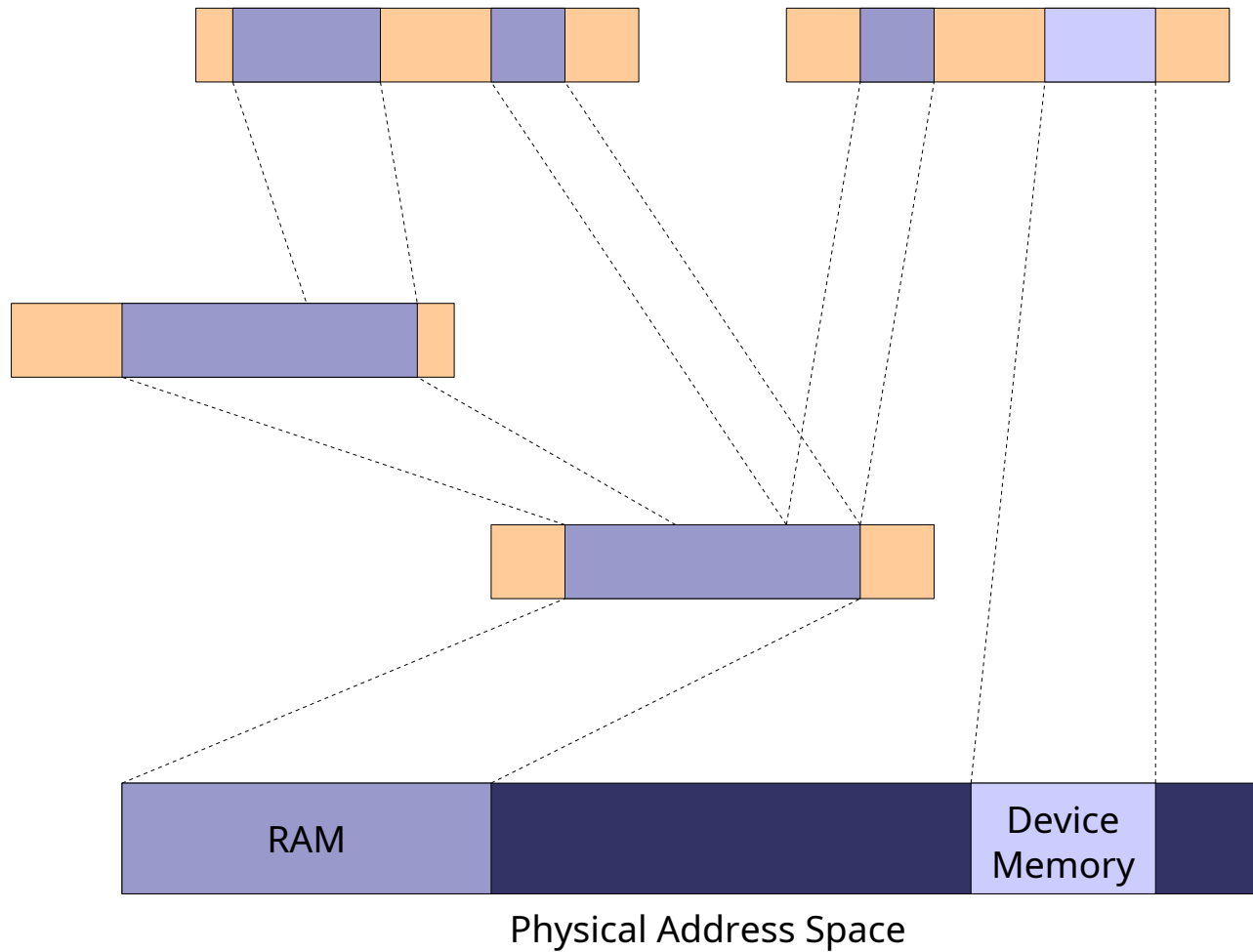
Challenge: Memory partitioning



Solution: Virtual Memory



L4: Recursive Address Spaces



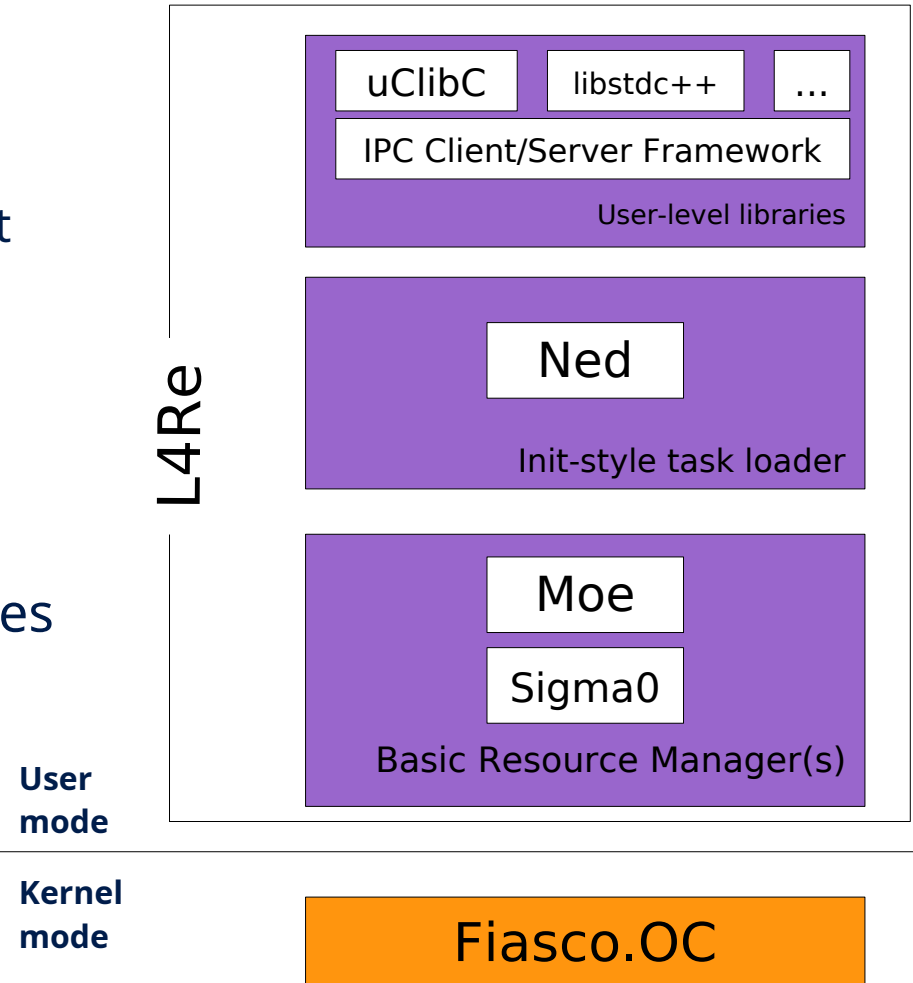
- Thread has access to a capability → can map this capability to another thread
- Mapping / not mapping of capabilities used to implement access control
- Abstraction for mapping: *flexpage*
 - Location and size of resource
 - Receiver's rights (read-only, mappable)
 - Type (memory, I/O, communication capability)

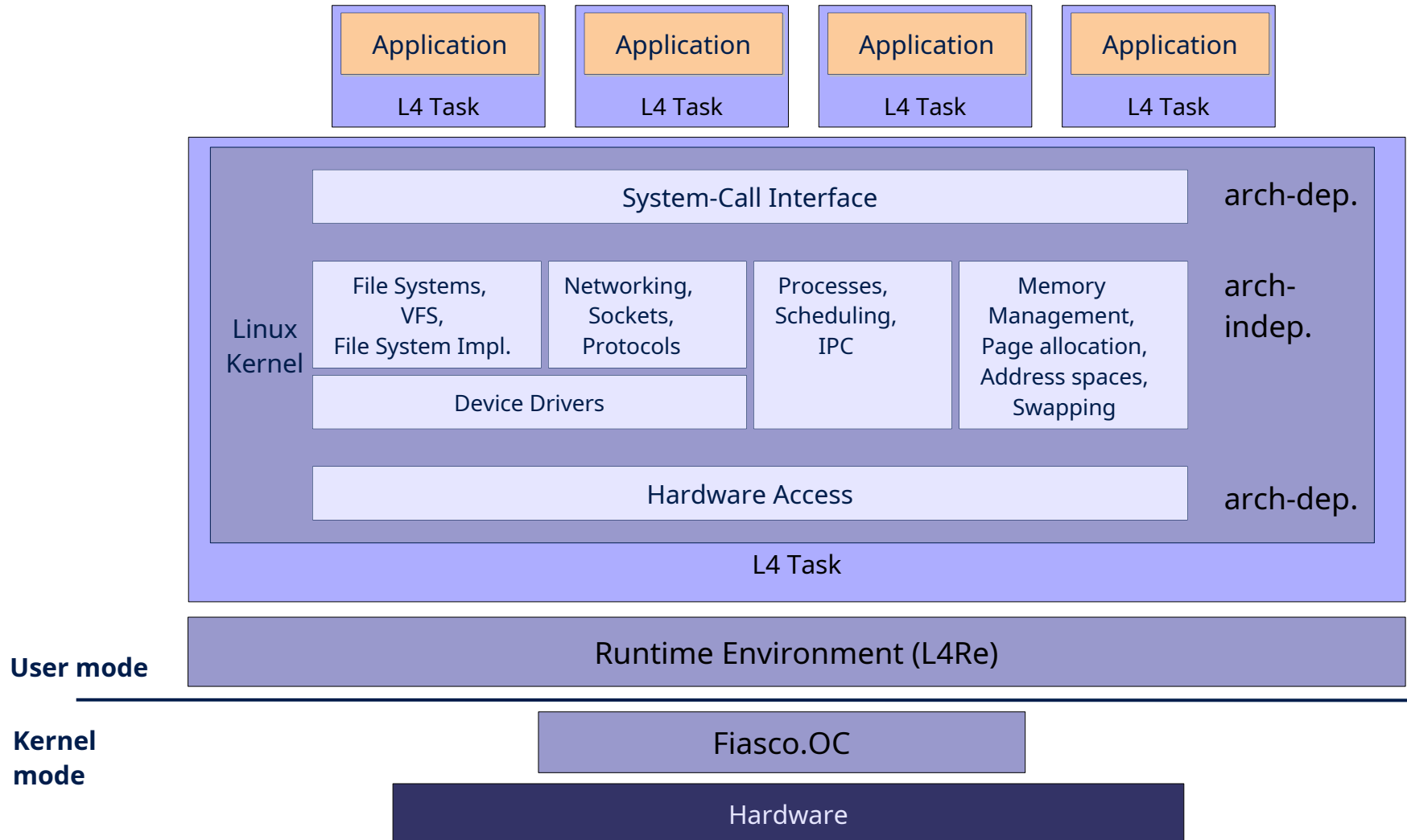
- Summary of object types
 - Task
 - Thread
 - IPC Gate
 - IRQ
 - Factory
- Each task gets initial set of capabilities for some of these objects at startup

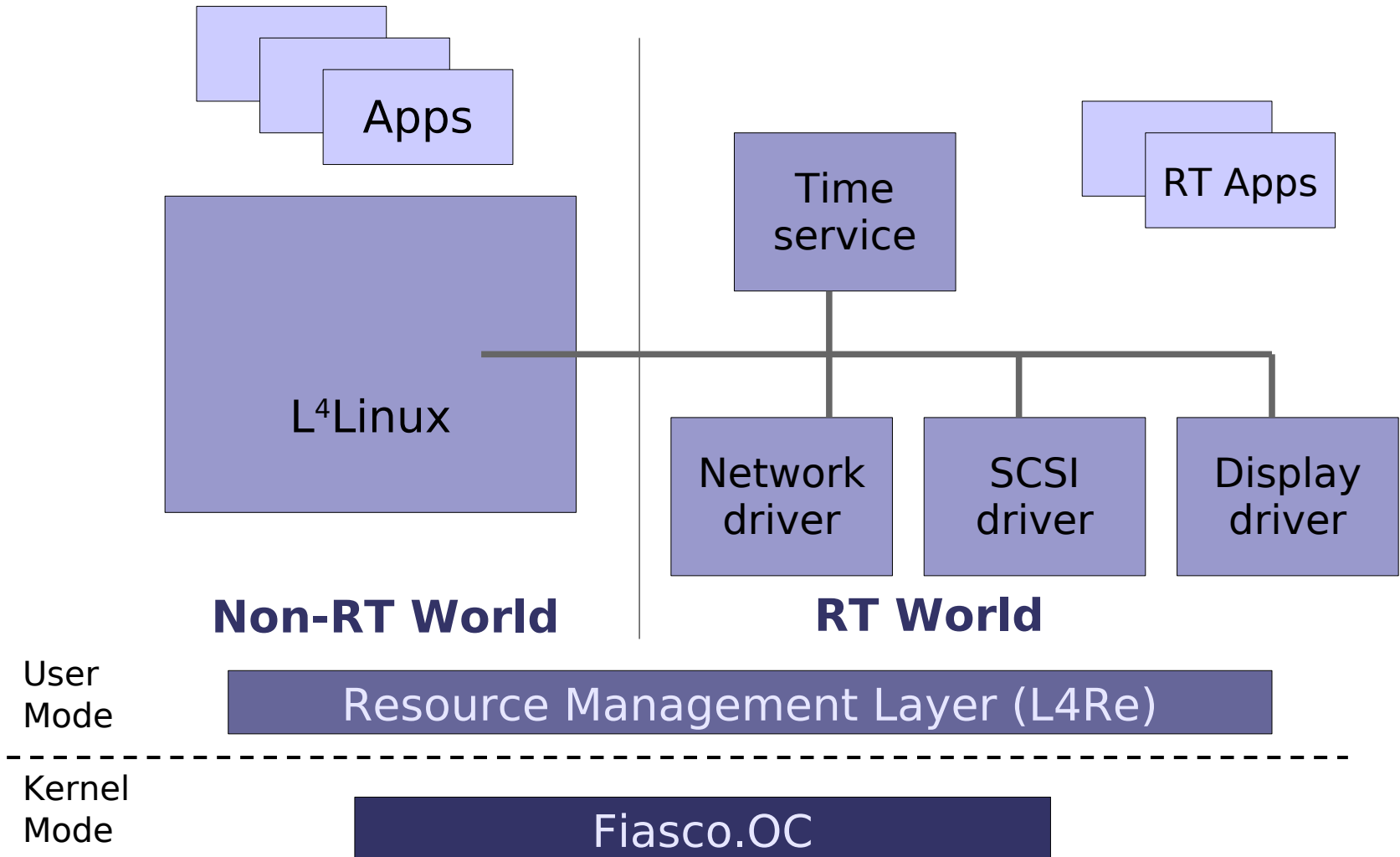
What can we build with this?

- 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

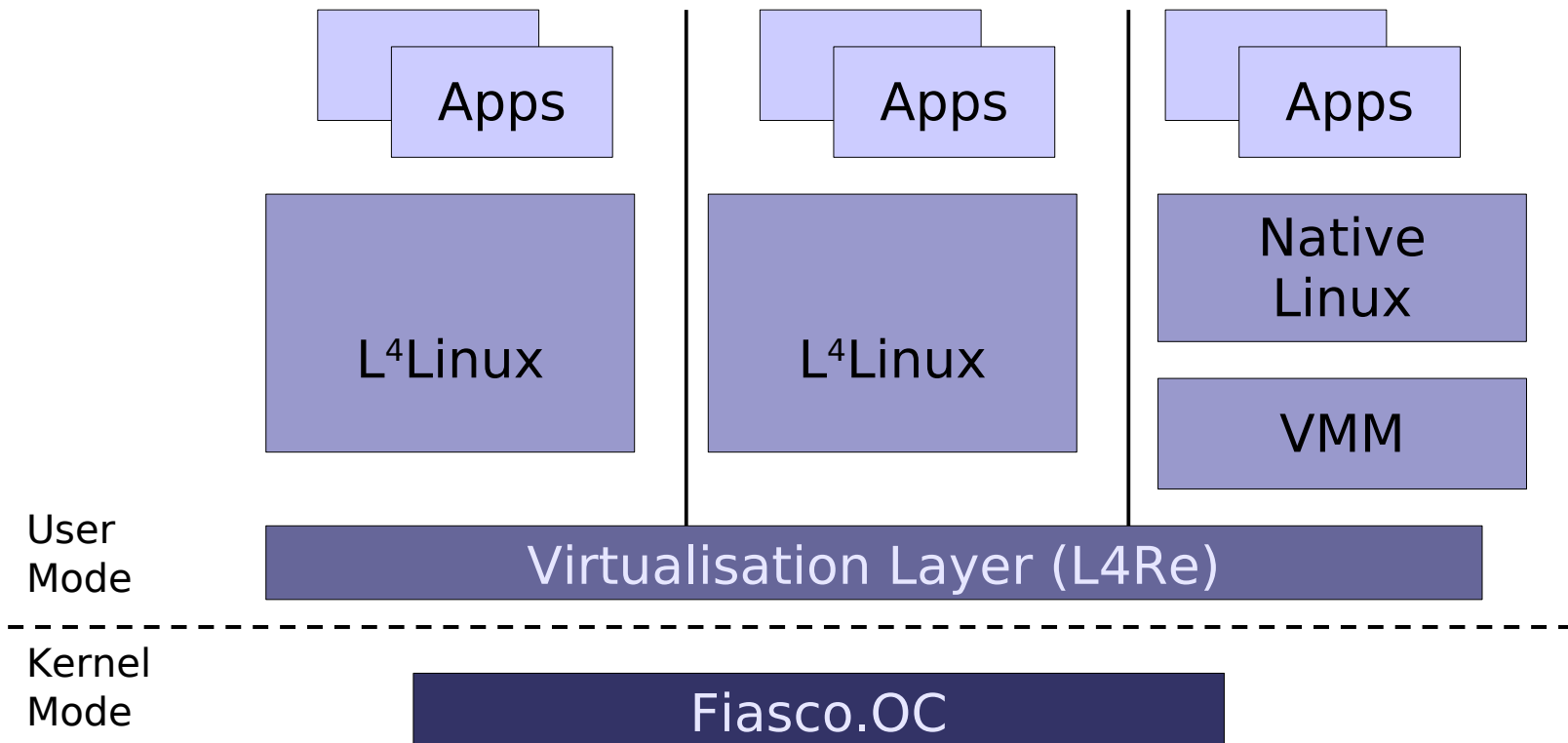
➔ L4Re = L4 Runtime Environment



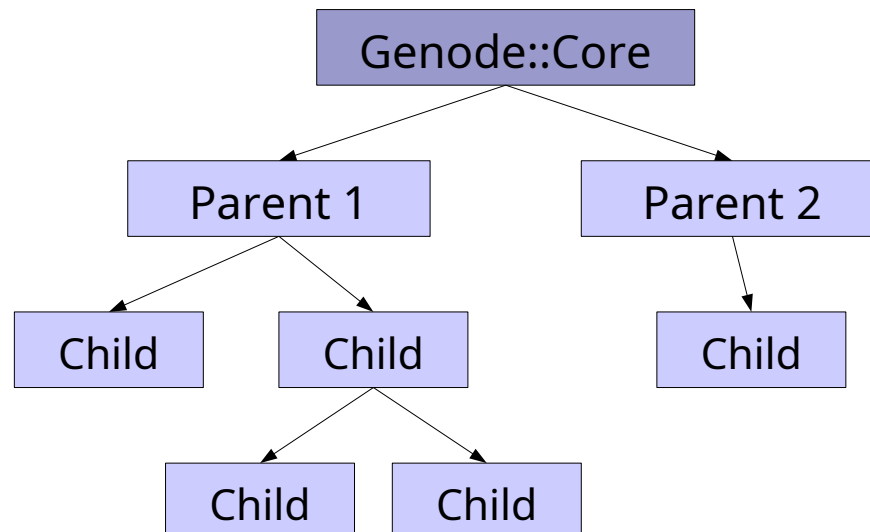




- Isolate not only processes, but also complete operating systems (compartments)
- “Server consolidation”



- Genode = C++-based OS framework developed here in Dresden
- Goal: hierarchical system that
 - supports resource partitioning
 - layers security policies on top of each other



What's to come?

- **Basic mechanisms and concepts**
 - Memory management
 - Tasks, Threads, Synchronisation
 - Communication
- **Building real systems**
 - What are resources and how to manage them?
 - How to build a secure system?
 - How to re-use existing code (Linux, standard system libraries, device drivers, ...)?
 - How to improve robustness and safety?

Date	14:50 @ APB E065	16:40 @ APB E065
Oct 13	Intro	Practical
Oct 20	IPC	Paper
Oct 27	Threads & Synchronization	Practical
Nov 3	Memory Management	—
Nov 10	Resource Management	Practical
Nov 17	—	<i>Practical</i>
Nov 24	Device Drivers	<i>Practical</i>
Dec 1	Virtualisation	Practical
Dec 8	Legacy Reuse	Paper
Dec 15	Security Architectures	—
Jan 5	Heterogenous Systems	<i>Practical</i>
Jan 12	Trusted Computing	Practical
Jan 19	—	<i>Paper</i>
Jan 26	—	Practical
Feb 2	Dependable Operating Systems	<i>Paper</i>