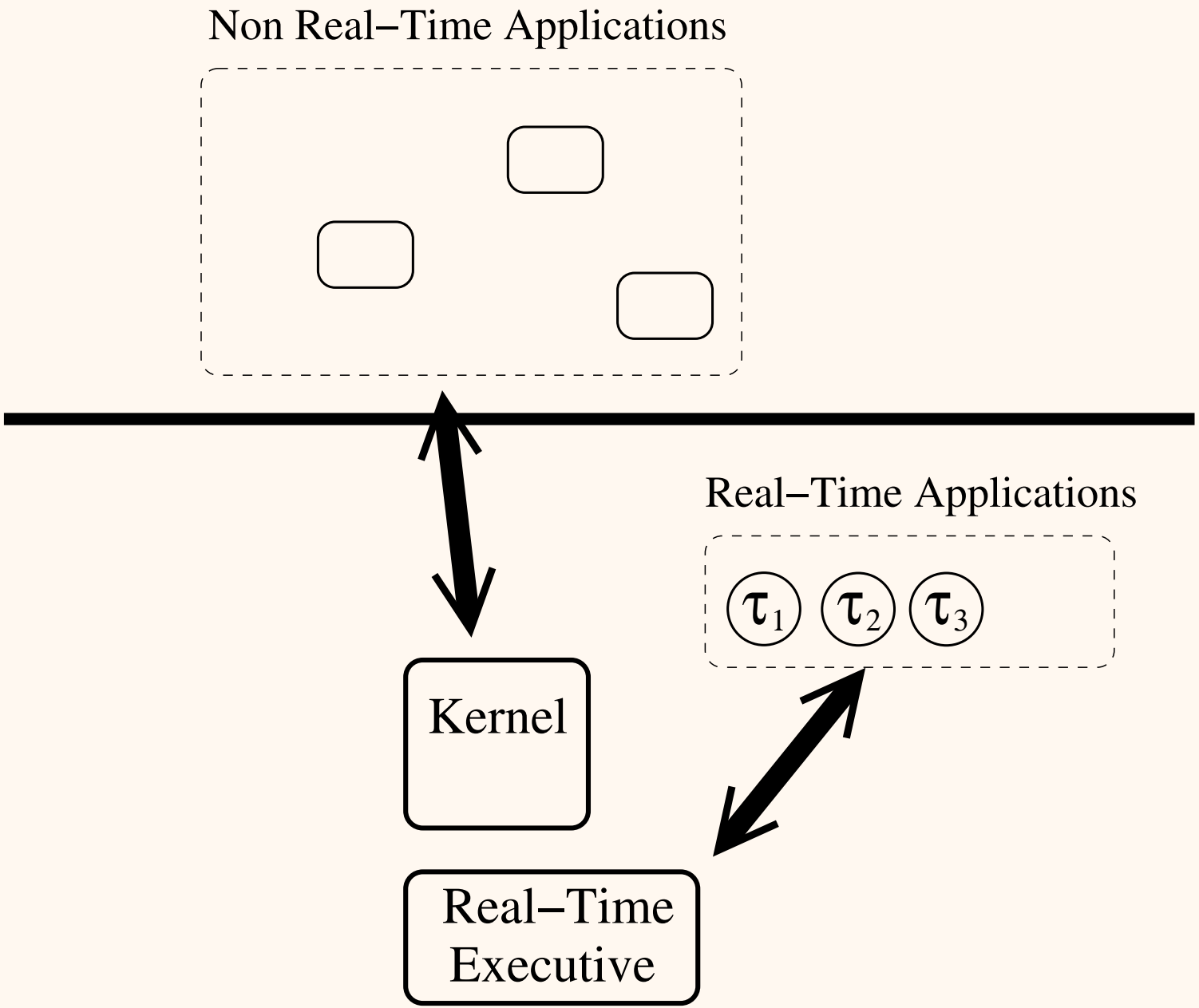
Something About MicroKernels

Luca Abeni
luca.abeni@santannapisa.it

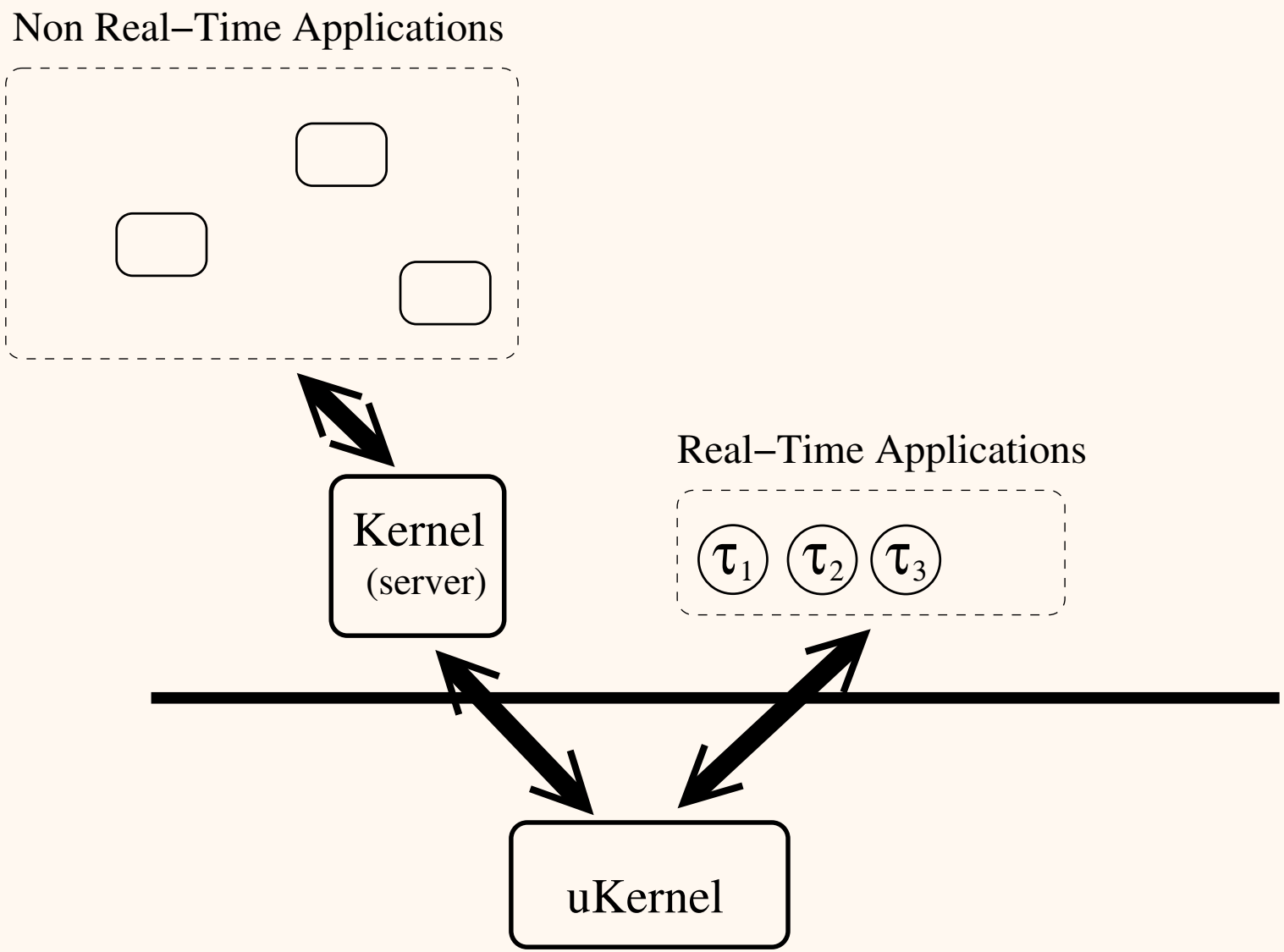
Reducing Kernel Latencies: Using HLP

- Strong distinction between real-time tasks and non real-time tasks
- Real-time tasks cannot use kernel functionalities!!!
- Two ways to achieve this result:
  - In dual kernel systems, real-time tasks are kernel modules (based on a small real-time executive!)
  - In μkernel systems, real-time tasks execute in user space

Dual Kernel System



μKernel-Based System



Traditional OS Structure

- OS: Set of computer programs interfacing user applications with the hardware
  - Kernel: part of the OS running at high privilege level
  - Traditionally includes a lot of things (drivers, network stack, fs, ...)
  - Do we really need high privileges for all of this?
- Special-purpose OSs often propose different structures
  - For virtualization, hypervisor running “below the kernel”
  - For real-time, dual-kernel approach
  - For security, reduced trusted code base...

μKernels - The Idea

- Basic idea: simplify the kernel
  - Reduce to the number of abstractions implemented by the kernel
    - Address Spaces
    - Threads
    - IPC mechanisms (channels, ports, etc...)
  - Most of the “traditional” kernel functionalities implemented in user space
  - Even device drivers can be in user space!

μKernels and Servers

- Interactions via IPC (IRQs to drivers as messages, ...)
- Servers: US processes implementing OS functionalities
  - OS kernel as a single user-space process: Single-server OSs
  - Multiple user-space processes (a server per driver, FS server, network server, ...): Multi-server OSs

μ-Kernels as Resource Managers

- A μ-kernel handles some “software resources”
  - Sometimes referred as “objects”
    - Debatable name: an object is generally (encapsulated) data + methods
  - Must be protected
- Example: address spaces, tasks, ...
  - The number / kind of abstractions / resource types depends on the μ-kernel details
- Tasks can “operate” on these resources
  - How to control the accesses / implement protection?

Message Passing Interactions

- Most of the interactions happen through message passing
  - Operation on a resource: send a message (and eventually wait for a reply)
  - Send message... To who?
    - The μ-kernel / kernel?
    - The resource / its manager?
    - Something else?
- Different IPC mechanisms depending on the μ-kernel
  - Synchronous / Asynchronous
  - Different security mechanisms
  - ...

Capabilities

- μ-kernels are often capability-based systems
  - Why? Because this is the most natural solution for an IPC-based system
- What is a “capability”? Informally speaking: reference to a software resource, associated with access rights
  - The exact definition might vary from system to system
- Intuition: to operate on something I must “own” the right capability
  - More advanced than a simple access control list

Capabilities as (Protected) References

- To access resource $\mathcal{R}$, task τ needs a reference to it
  - Example: you cannot open a file if you do not know its name
- Protected reference: tasks cannot forge capabilities
  - Capabilities are created and manipulated by the capability system
  - So, a file name is not a good example!
- Capabilities are opaque
  - You do not really know what a capability is: you just use it to access a resource
- Think about pointers

Capabilities and Access Rights

- A capability is not a simple protected reference
- It is associated to access rights
  - A capability can be used to perform an action on a resource, but not other actions
  - Example: read/receive capability, write/send capability, ...
- Using the read capability for a file, I can read it, but I cannot write on it!
- Each task owns capabilities for accessing some resources
- The μ-kernel / capability system is responsible for enforcing the respect of capabilities

Capabilities Management

- Tasks cannot create capabilities $\rightarrow$ a task "receives" a capability from someone else
  - Can be the μ-kernel / capability system
  - ... Someone else? ...
- Capabilities can be transferred
  - A task owning a capability can send it to another task
  - What happens when a capability is tranferred?
- The capability system defines the exact behaviour of capability transfer

Capabilities and Messages

- Capabilities can be used for IPC access control
  - Used to send / receive messages
  - Used to check if a task has the rights to send / receive a message
  - ...
- Object Oriented vision: resource $\rightarrow$ Object
  - Invocation of a method $\rightarrow$ send a message to the object

Example: Mach

- Mach μ-kernel: capability-based, tries to implement some OO concepts
  - Tasks can operate on “objects” by sending messages to them
  - IPC mechanism provided by the (μ)kernel
- Mach IPC: indirect addressing
  - The destination of a message is indicated by specifying a “communication channel”
  - Tasks send messages to ports $\rightarrow$ queues of messages
- Capabitilies implemented through ports
  - Capability $\leftrightarrow$ read or write reference to a port)

Mach Ports and Capabilities

- Port rights: secure, location-independent way of naming ports
- The receive right for a port can be owned only by one single task
  - A task can send a message to a port by using a “send capability” (send right) for the port
  - A task can receive a message from a port by using a “receive capability” (receive right) for the port
- Each task is created with some ports for communicating with the kernel (and owns the “send rights” - capabilities for such ports)

Sending Capabilities

- A task can send one of its “port right” in a message
- The task receiving the message will be able to access the port
- When a receive right is contained in the message, the right is revoked from the sender
  - The sender “donates” its capability to the receiver
  - Remember: the receive right for a port cannot be owned by multiple tasks!

Example: Scheduling Capabilities

- Capabilities can be used for scheduling too!
  - Temporal capabilities
- Distinction between scheduling context and execution context
  - In traditional systems, a “task context” contains both scheduling information and the task state used for dispatching
  - Scheduling context: data structure used by the scheduler
  - Execution context: rest of the task state
- Scheduling capability: reference to the scheduling context

Scheduling and Dispatching

- A task can execute only if it owns a scheduling capability
  - Will be dispatched when the scheduler selects the corresponding scheduling context
- Generally, a scheduling context is associated to a core / CPU
  - Migration: the scheduling capability of a task is replaced with a different one (referencing a scheduling context associated to a different core / CPU)

Flexibility

- Inheritance is simple: donate the scheduling capability to a different task!
  - Easy form of proxy execution... Useful for client/server interactions!
- User space scheduling is possible: a user-space server manages the scheduling capabilities of the tasks!