TECHNISCHE UNIVERSITÄT DRESDEN

Department of Computer Science Institute of System Architecture, Operating Systems Group

REAL-TIME

MICHAEL ROITZSCH

OVERVIEW

SO FAR

- talked about in-kernel building blocks:
  - threads
  - memory
  - IPC
- drivers will enable access to a wide range of non-kernel resources
- need to manage resources

COURSE EIP

Applications

System Services

Basic Abstractions

COMPARISON

Memory	Time
discrete, limited	continuous, infinite
hidden in the system	user-perceivable
managed by pager	managed by scheduler
page-granular partitions	arbitrary granularity
all pages are of equal value	value depends on workload
active policy decisions, passive enforcement	active policy decisions, active enforcement
hierarchical management	Fiasco: flattened in-kernel view

REAL-TIME

DEFINITION

- a **real-time system** denotes a system, whose correctness depends on the timely delivery of results
- „it matters, **when** a result is produced"
- **real-time** denotes a predictable relation between system progress and wall-clock time

EXAMPLES

- engine control in a car
- break-by-wire
- avionics
- railway control

focused
catastrophic failures

- set-top box DVD player
- GSM-stack in your cell phone

benign failures
complex

THEMES

① Predictability

② Guarantees

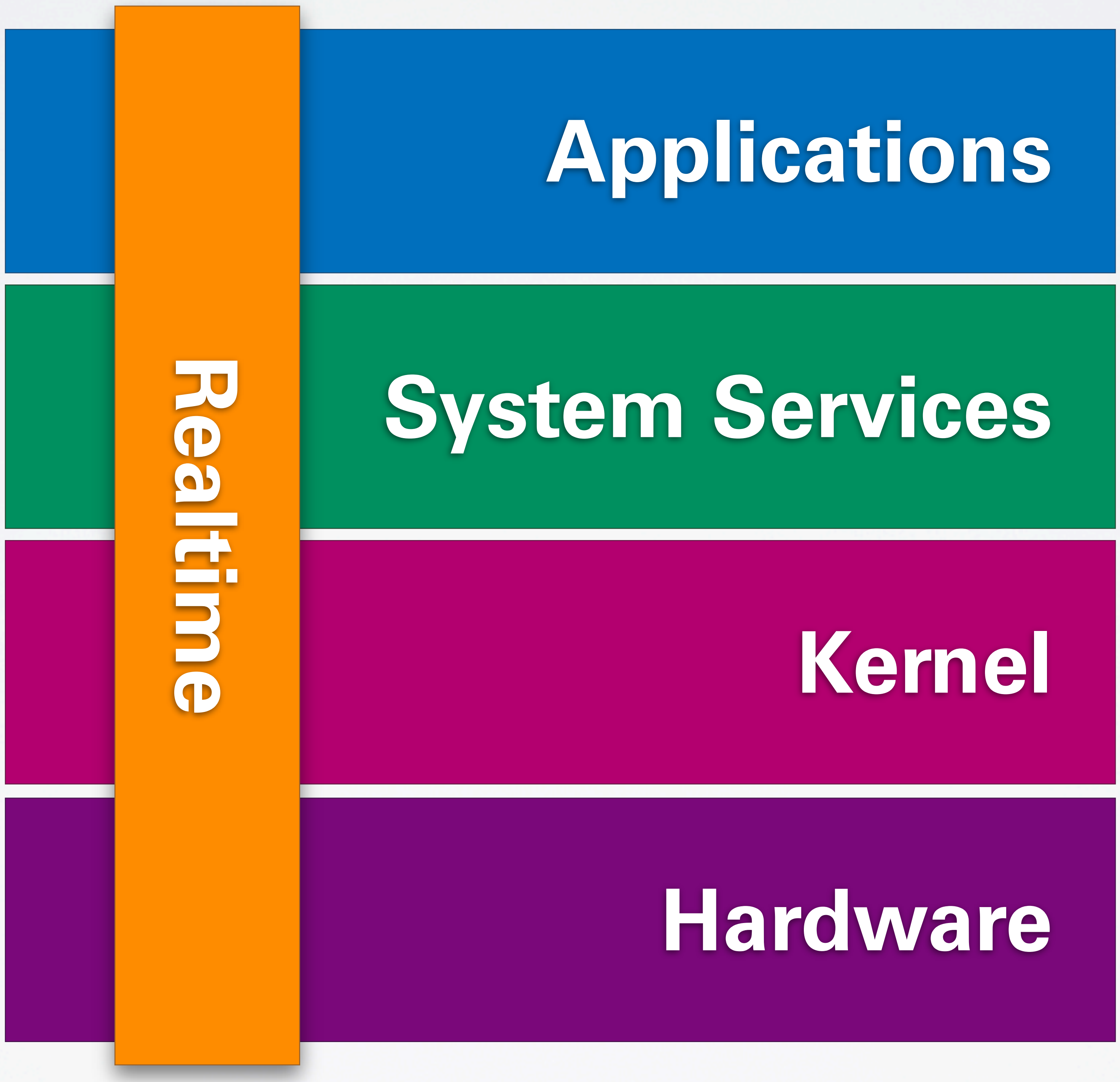
③ Enforcement

PREDICTABILITY

ENEMIES

- gap between worst and average case
  - memory caches, disk caches, TLBs
- „smart“ hardware
  - system management mode
  - disk request reordering
- cross-talk from resource sharing
  - servers showing O(n) behavior
  - SMP
- unpredictable external influences
  - interrupts

CROSSCUTTING



CUSTOM RTOS

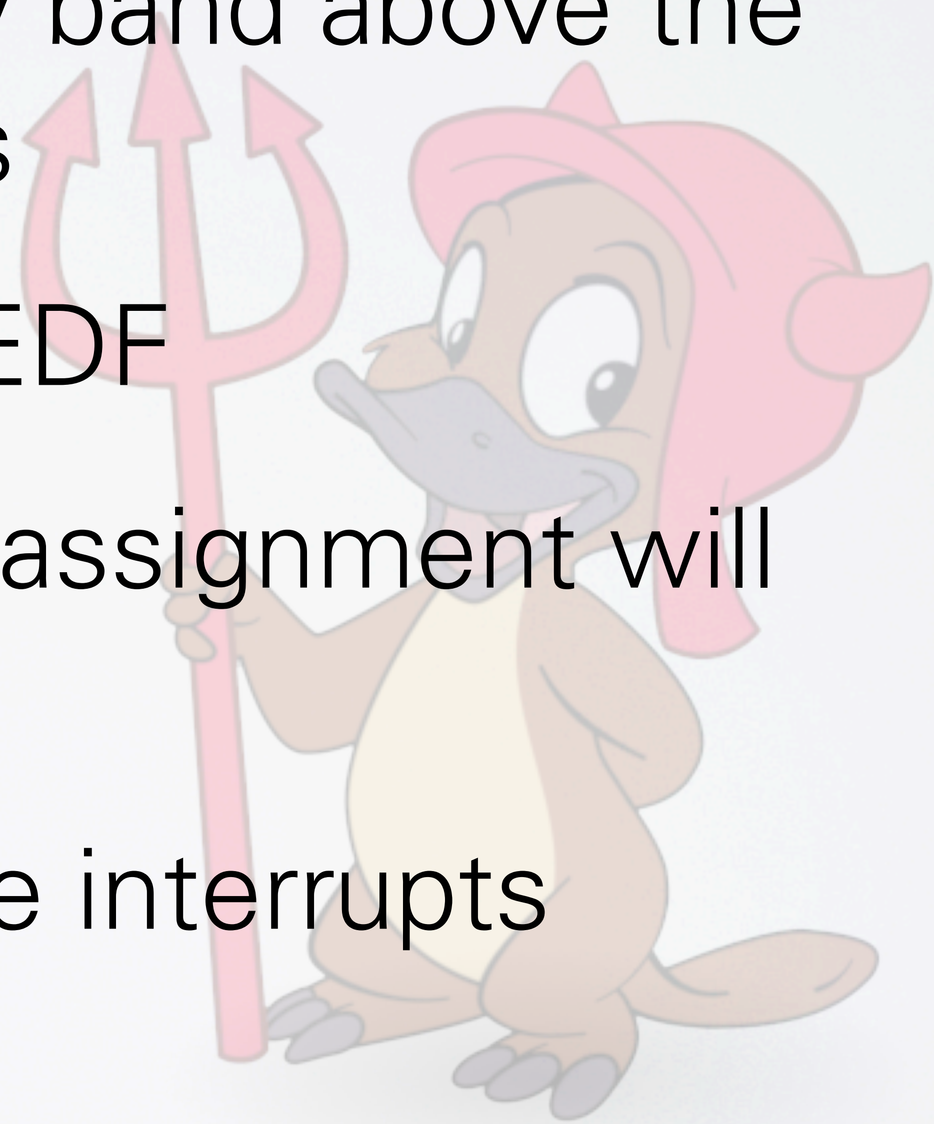
- small real-time executives tailor-made for specific applications
- fixed workload known a priori
- pre-calculated time-driven schedule
- used on small embedded controllers
- benign hardware

RTLINUX

- full Linux kernel and real-time processes run side-by-side
- small real-time executive underneath supports scheduling and IPC
- real-time processes implemented as kernel modules
- all of this runs in kernel mode
- no isolation

XNU

- the kernel used in Mac OS X
- offers a real-time priority band above the priority of kernel threads
- scheduled using global EDF
- threads exceeding their assignment will be demoted
- all drivers need to handle interrupts correctly

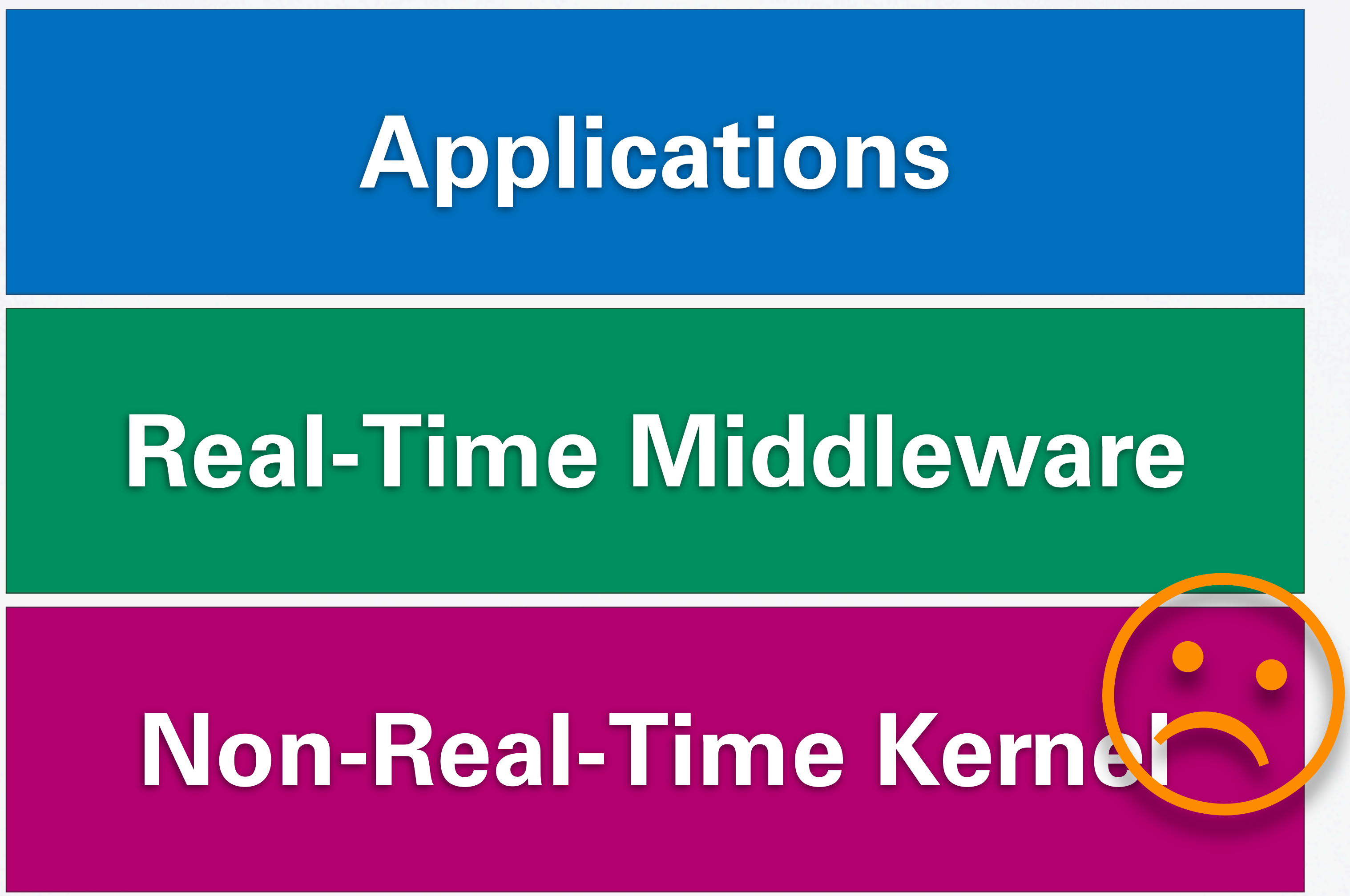


FIASCO

- static thread priorities
- O(1) complexity for most system calls
- fully preemptible in kernel mode
  - bounded interrupt latency
- lock-free synchronization
  - uses atomic operations
- wait-free synchronization
  - locking with helping instead of blocking

BE AFRAID

- architecture for those afraid of touching the OS
  - example: Real-Time Java



RESOURCES

- a real-time kernel alone is not enough
- the microkernel solution:
  - real-time kernel enables temporal isolation
    - simplifies cross-talk and interrupt problems
  - user-level servers on top act as resource managers
    - implement real-time views on specific resources
- real-time is not only about CPU
- details in the resource management lecture

GUARANTEES

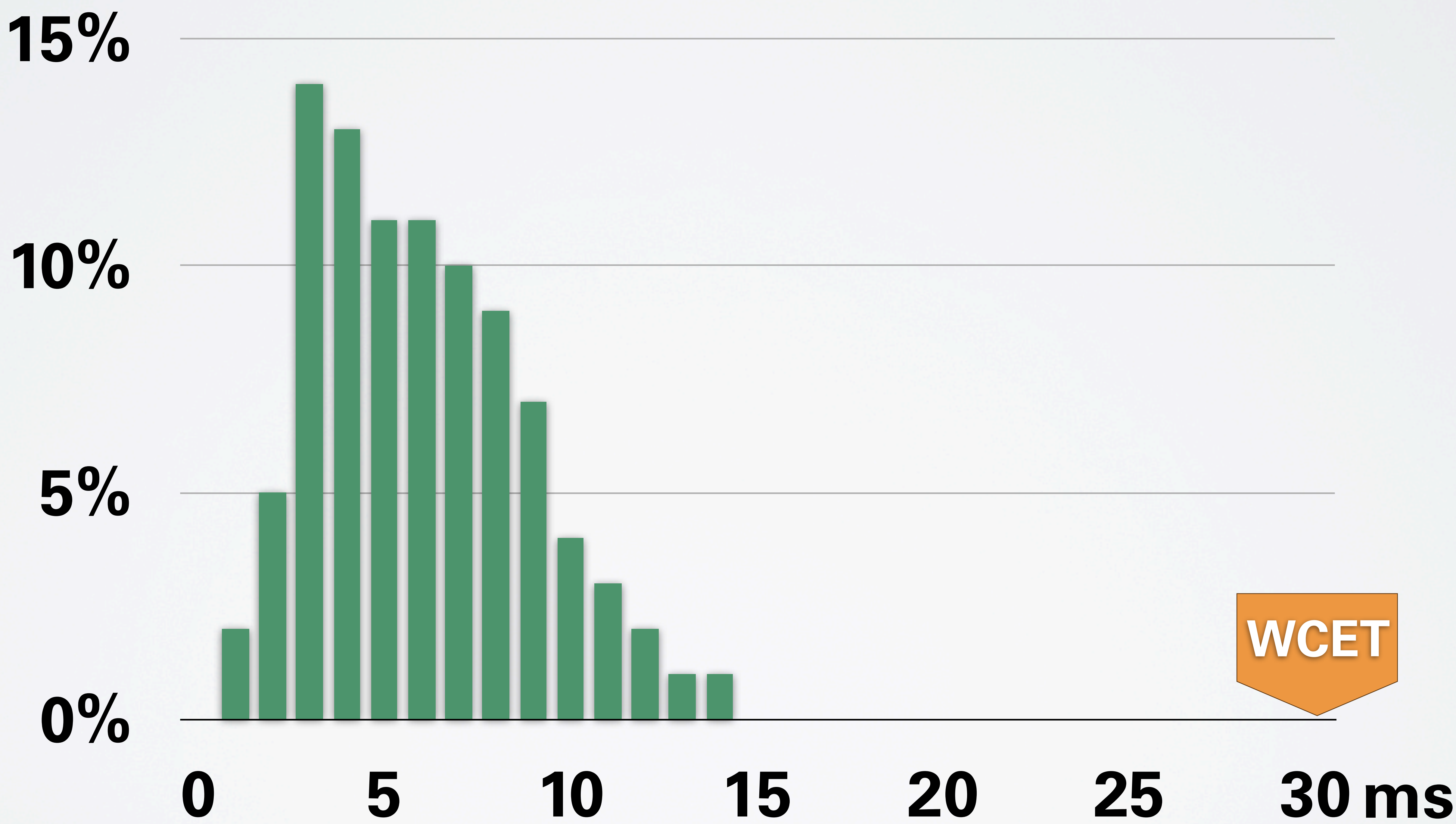
PROBLEM

- worst case execution time (WCET) largely exceeds average case
- offering guarantees for the worst case will waste lots of resources
- missing some deadlines can be tolerated with the firm and soft real-time scheme

MOTIVATION

- desktop real-time
- there are no hard real-time applications on desktops
- there is a lot of firm and soft real-time
  - low-latency audio processing
  - smooth video playback
  - desktop effects
  - user interface responsiveness

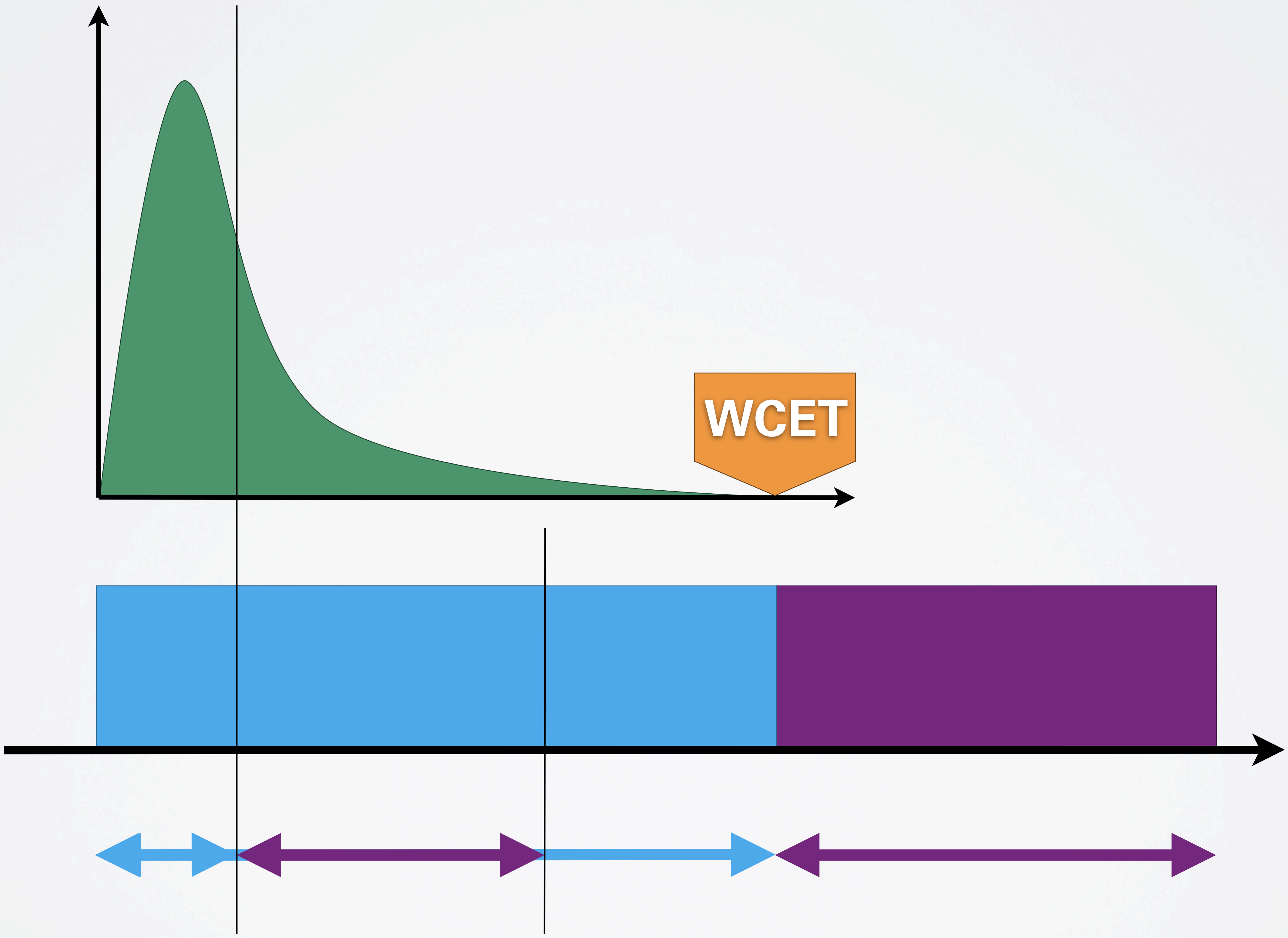
H.264 DECODING



KEY IDEA

- guarantees even slightly below 100% of WCET can dramatically reduce resource allocation
- unused reservations will be used by others at runtime
- use probabilistic planning to model the actual execution
- quality *q*: fraction of deadlines to be met

KEY IDEA



RESERVATION

$$r'_i = \min(r \in \mathbb{R} \mid \frac{1}{m_i} \sum_{k=1}^{m_i} \mathbf{P}(X_i + k \cdot Y_i \leq r) \geq q_i)$$

$$r_i = \max(r'_i, w_i) \quad i = 1, \ldots, n$$

- to fully understand this:
  see real-time systems lecture
- good for microkernel: reservation can be calculated by a userland service
- kernel only needs to support static priorities

SCHEDULING

- **scheduling = admission + enforcement**
- admission = scheduling analysis
  - evaluates the requests from clients, which follow some task model
  - calculates task parameters
  - verifies the feasibility of the guarantees
  - can reject requests
- enforcement
  - executing the schedule

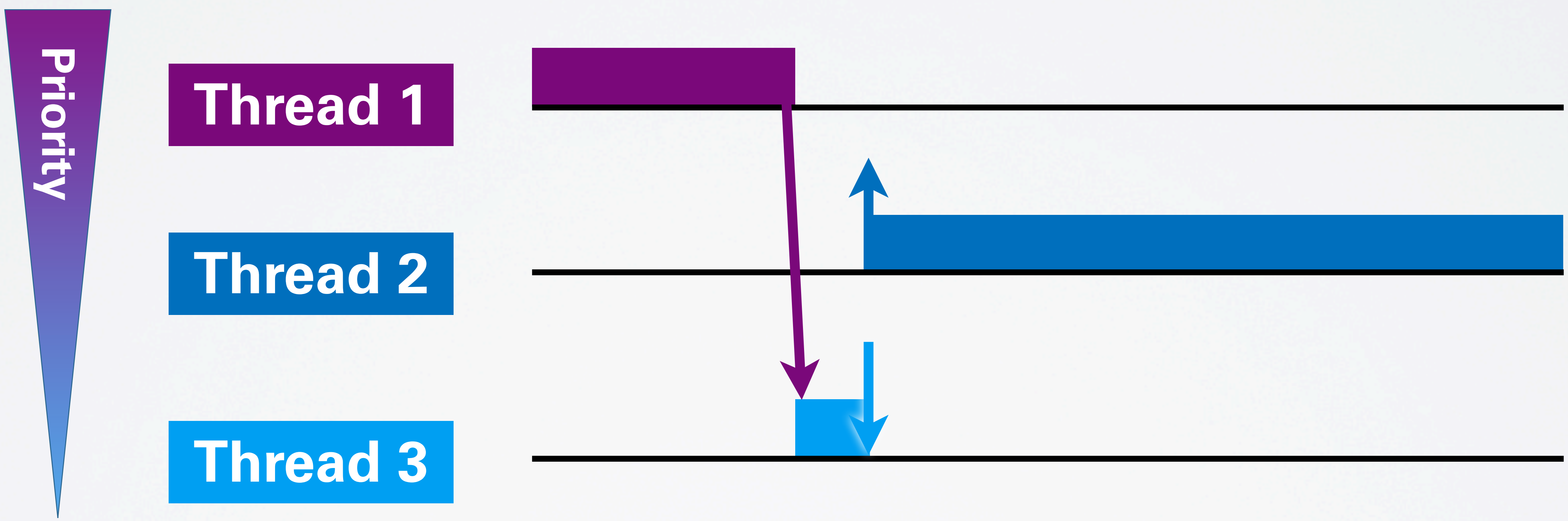
ENFORCEMENT

DISPATCHER

- executed at specific events
- enforces task parameters by preemption
  - e.g. on deadline overrun
- picks the next thread
  - static priorities (e.g. RMS, DMS)
  - dynamic priorities (e.g. EDF)
- seems simple…

PROBLEM

- high priority thread calls low priority service with a medium priority thread interfering:



1 waits for 3 ✔

3 waits for 2 ✔

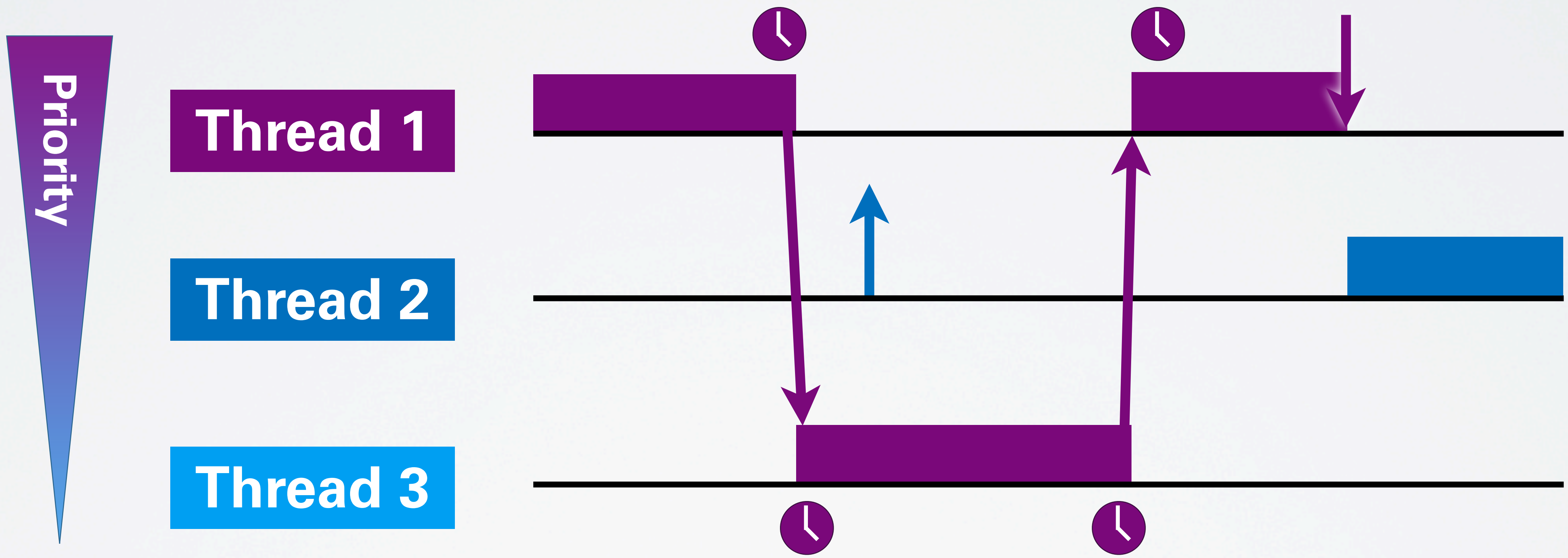
= 1 waits for 2 ✘

Priority Inversion

SOLUTION

- priority inheritance, priority ceiling
- nice mechanism for this in Fiasco, NOVA: timeslice donation
- implemented by splitting thread control block
  - **execution context:** holds CPU statue
  - **scheduling context:** time and priority
- on IPC-caused thread switch, only the execution context is switched

DONATING CALL



- IPC receiver runs on the sender's scheduling context
- priority inversion problem solved with priority inheritance

ACCOUNTING

- servers run on their clients' time slice
  - when the server executes on behalf of a client, the client pays with its own time
- this allows for servers with no scheduling context
  - server has no time or priority on its own
  - can only execute on client's time
  - relieves dispatcher from dealing with servers

TECHNISCHE UNIVERSITÄT DRESDEN

TIMELESS

- servers could be malicious, so you need timeouts to get your time back
- now, malicious clients can call the server with a very short timeout
  - server's time will be withdrawn while the server is in the middle of request handling
- servers need to do session cleanup
  - on whose time?
  - open research problem

SMP

- timeslice donation does not work in multiprocessor case:
  - server is running on a different CPU
  - cross-CPU donation would be needed
- you cannot donate time between CPUs
  - think of real-time guarantees
  - if one CPU is 100% loaded, donating time to it will overload it
- servers need to migrate to client CPU?

TECHNISCHE UNIVERSITÄT DRESDEN

SYNCHRONIZATION

BASICS

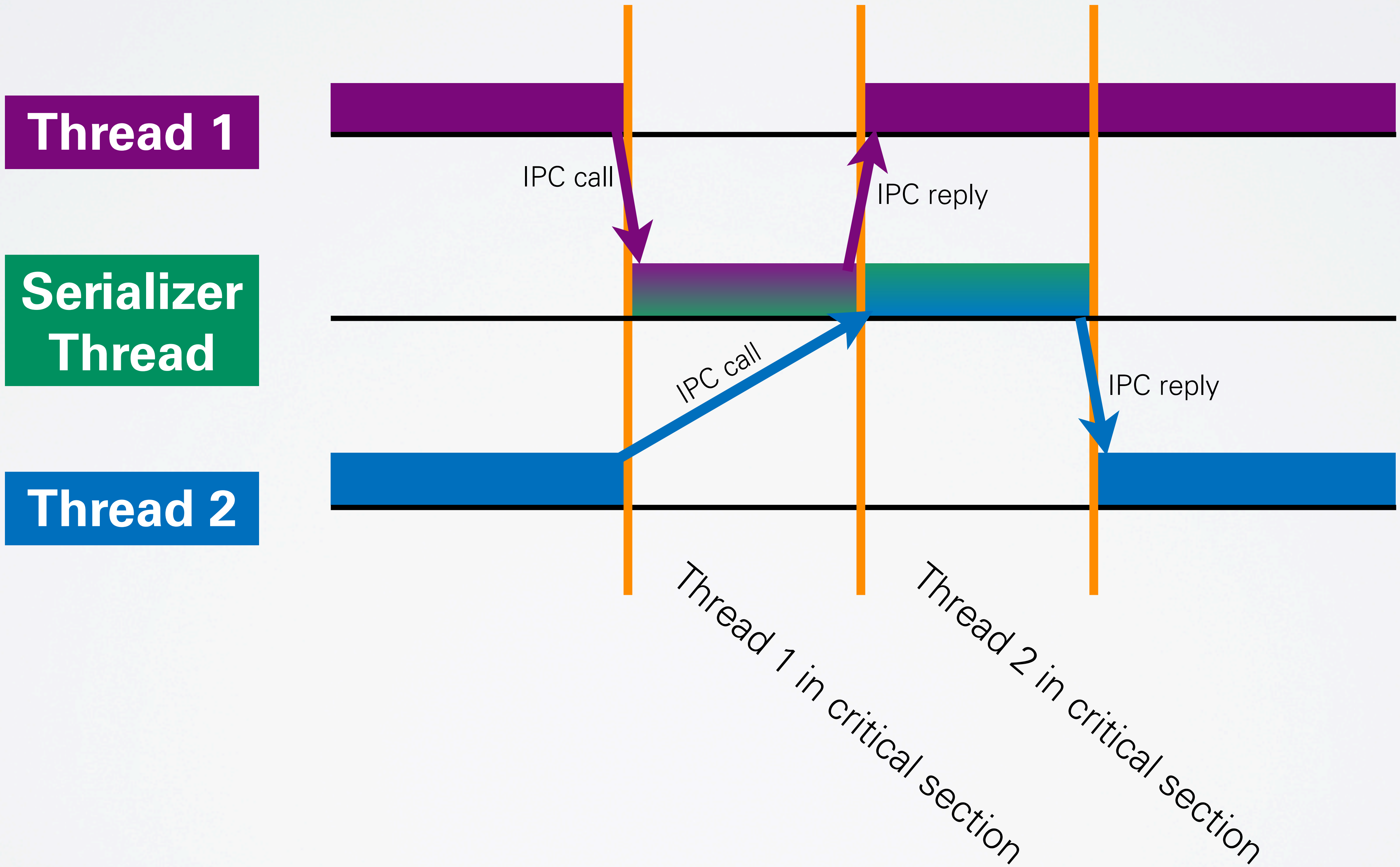
- synchronization used for
  - mutual exclusion
  - producer-consumer-scenarios
- traditional approaches that do not work
  - spinning, busy waiting
  - disabling interrupts

ATOMIC OPS

- for concurrent access to data structures
- use atomic operations to protect manipulations
- only suited for simple critical sections

EXPECTATION

SOLUTION

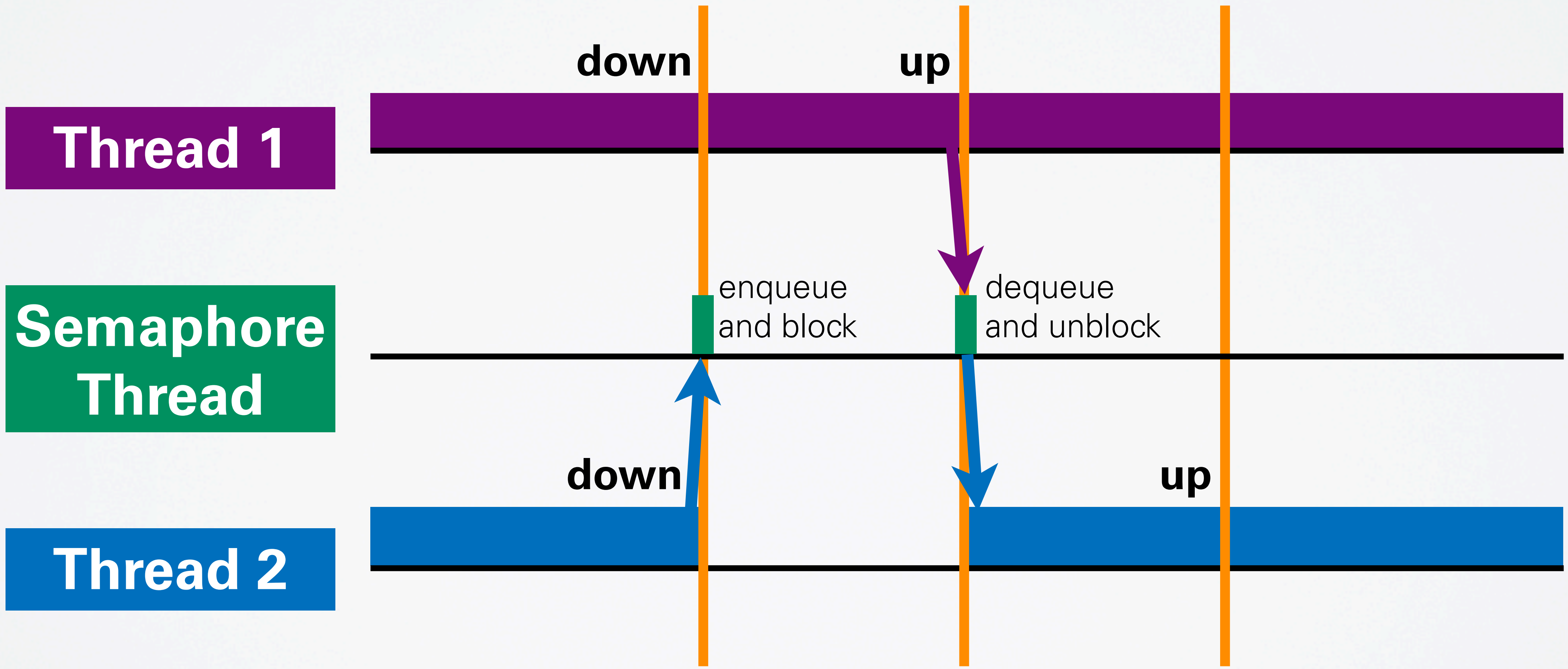

Thread 1
Serializer Thread
Thread 2
IPC call
IPC reply
IPC call
IPC reply
Thread 1 in critical section
Thread 2 in critical section

SEMAPHORES

- serializer and atomic operations can be combined to a nice counting semaphore
- semaphore
  - shared counter for correctness
  - wait queue for fairness
  - down (P) and up (V) operation
  - semaphore available iff counter > 0

SEMAPHORES

- counter increments and decrements using atomic operations
- when necessary, call semaphore thread to block/unblock and enqueue/dequeue



BENEFITS

- cross-task semaphores, when counter is in shared memory
- IPC only in the contention case
  - good for mutual exclusion when contention is rare
  - for producer-consumer-scenarios, contention is the common case

SUMMARY

- managing time is necessary
  - we interact with the system based on time
- real-time is a cross-cutting concern
- „hard real-time is hard, soft real-time is even harder“ (E. Douglas Jensen)
- priority inheritance by timeslice donation
- synchronisation, ~~delayed preemption~~
- next week: drivers