



**TECHNISCHE
UNIVERSITÄT
DRESDEN**

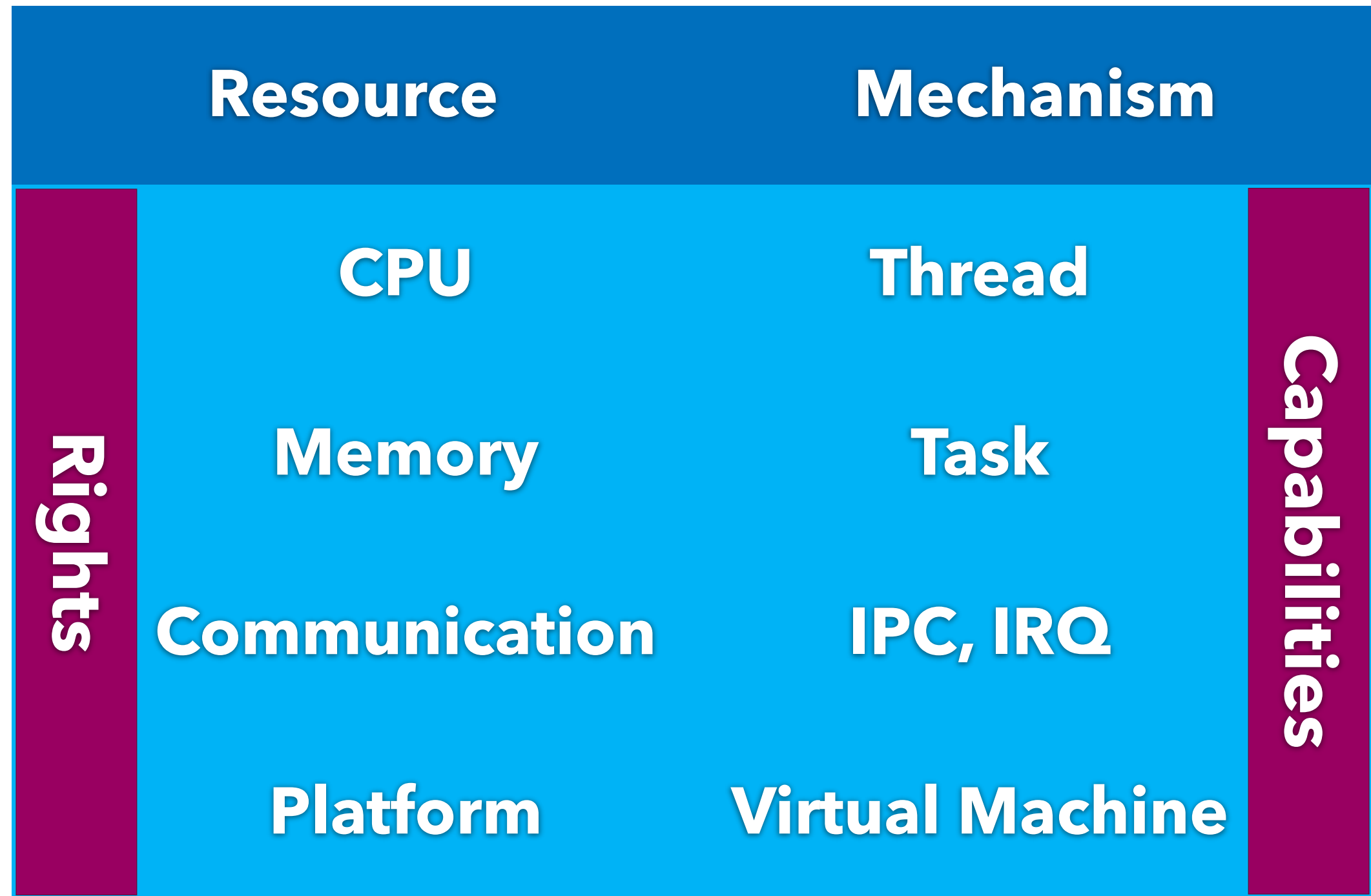
Faculty of Computer Science Institute of Systems Architecture, Operating Systems Group

THREADS

MICHAEL ROITZSCH

RECAP

- kernel:
 - provides system foundation
 - usually runs in privileged CPU mode
- microkernel:
 - kernel provides mechanisms, no policies
 - most functionality implemented in user mode, unless dictated otherwise by
 - security
 - performance

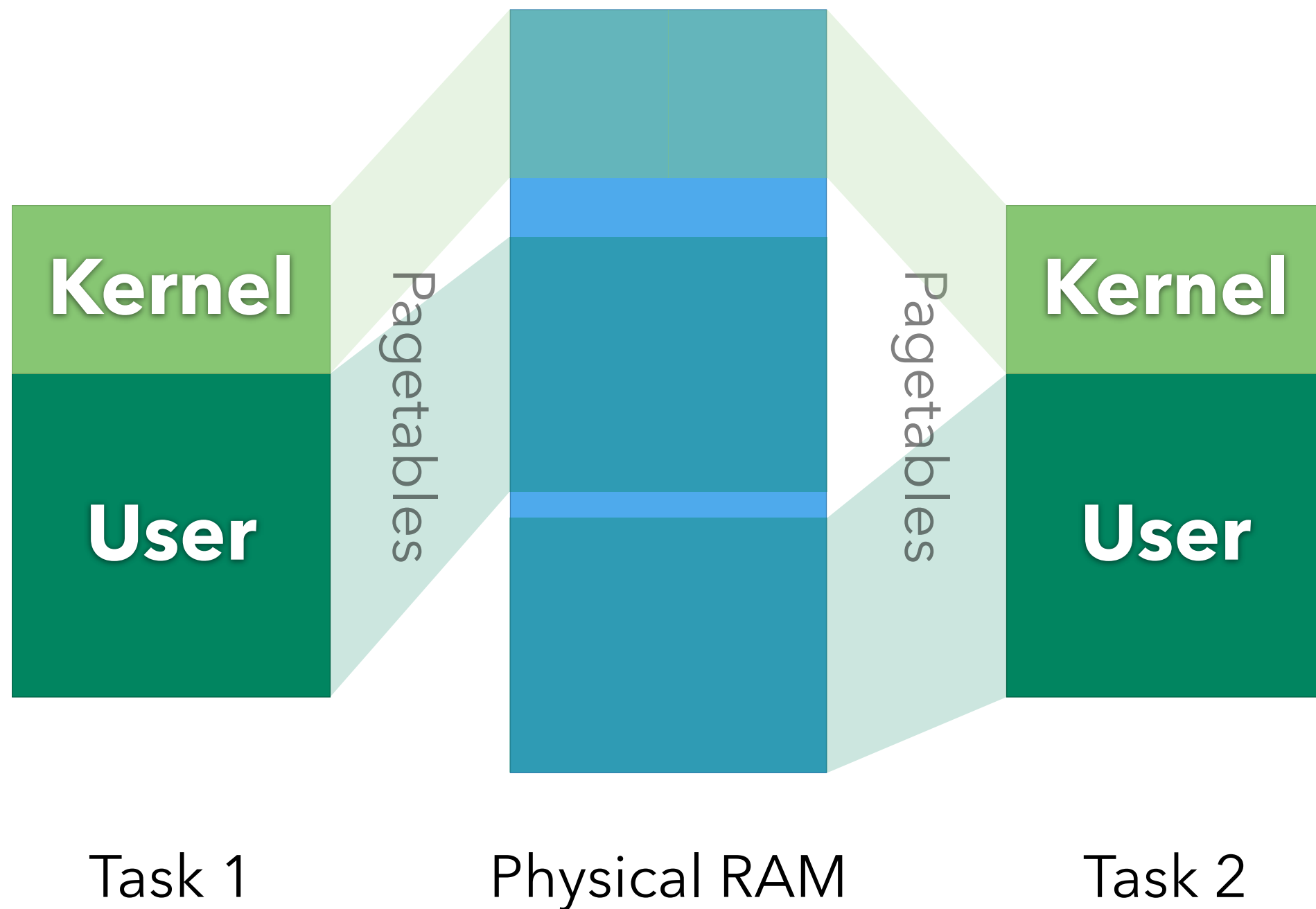


- provides an exclusive instance of a full system platform
- may be a synthetic platform (bytecode)
- full software implementations
- hardware-assisted implementations in the kernel (hypervisor)
- see virtualization lecture

- inter-process communication
- between threads
- two-way agreement, synchronous
- memory mapping with flexpages
- see communication lecture

- (virtual) address space
- unit of memory management
- provides spatial isolation
- common memory content can be shared
 - shared libraries
 - kernel
- see memory lecture

SHARED KERNEL



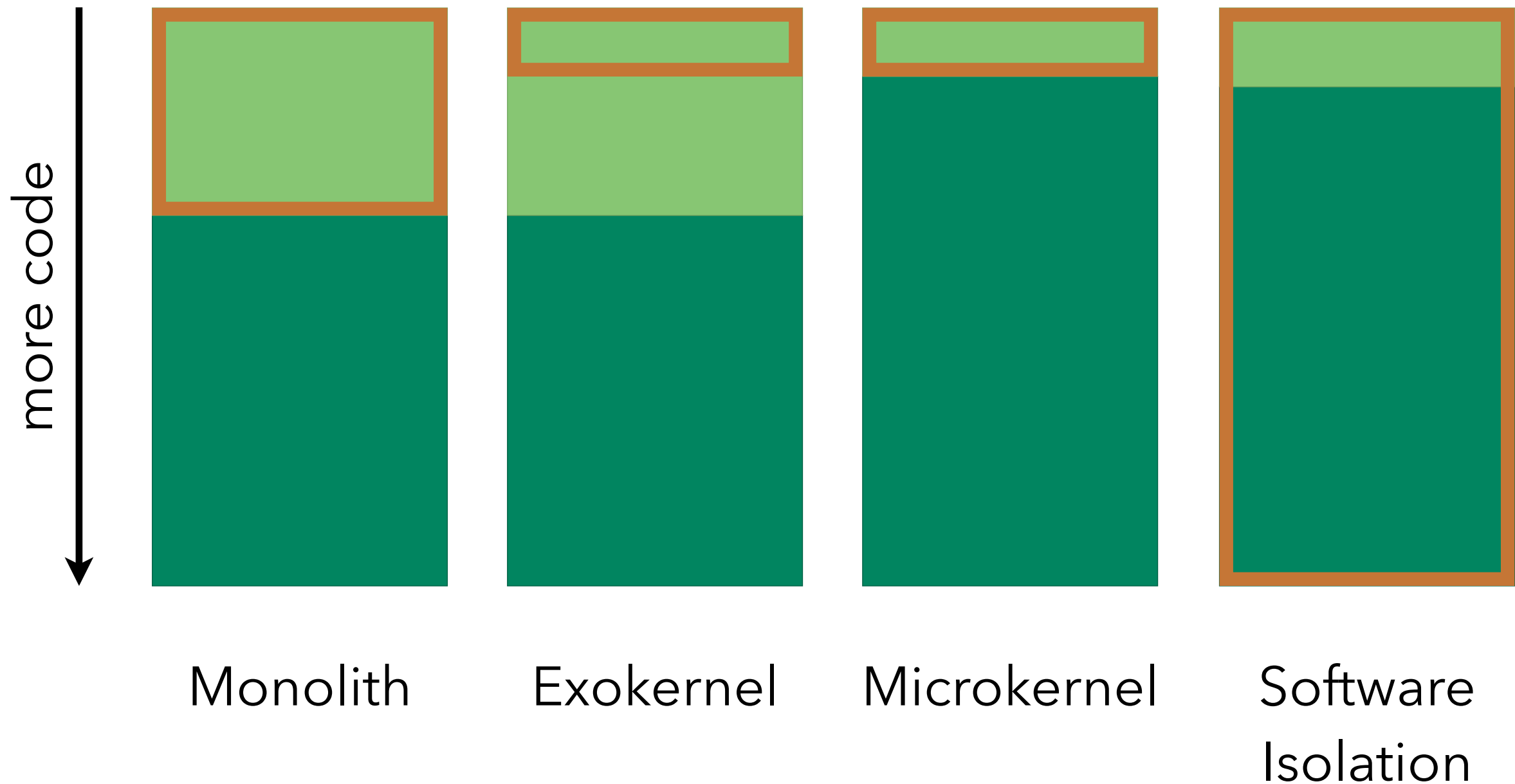


User Address Space

Kernel Address Space

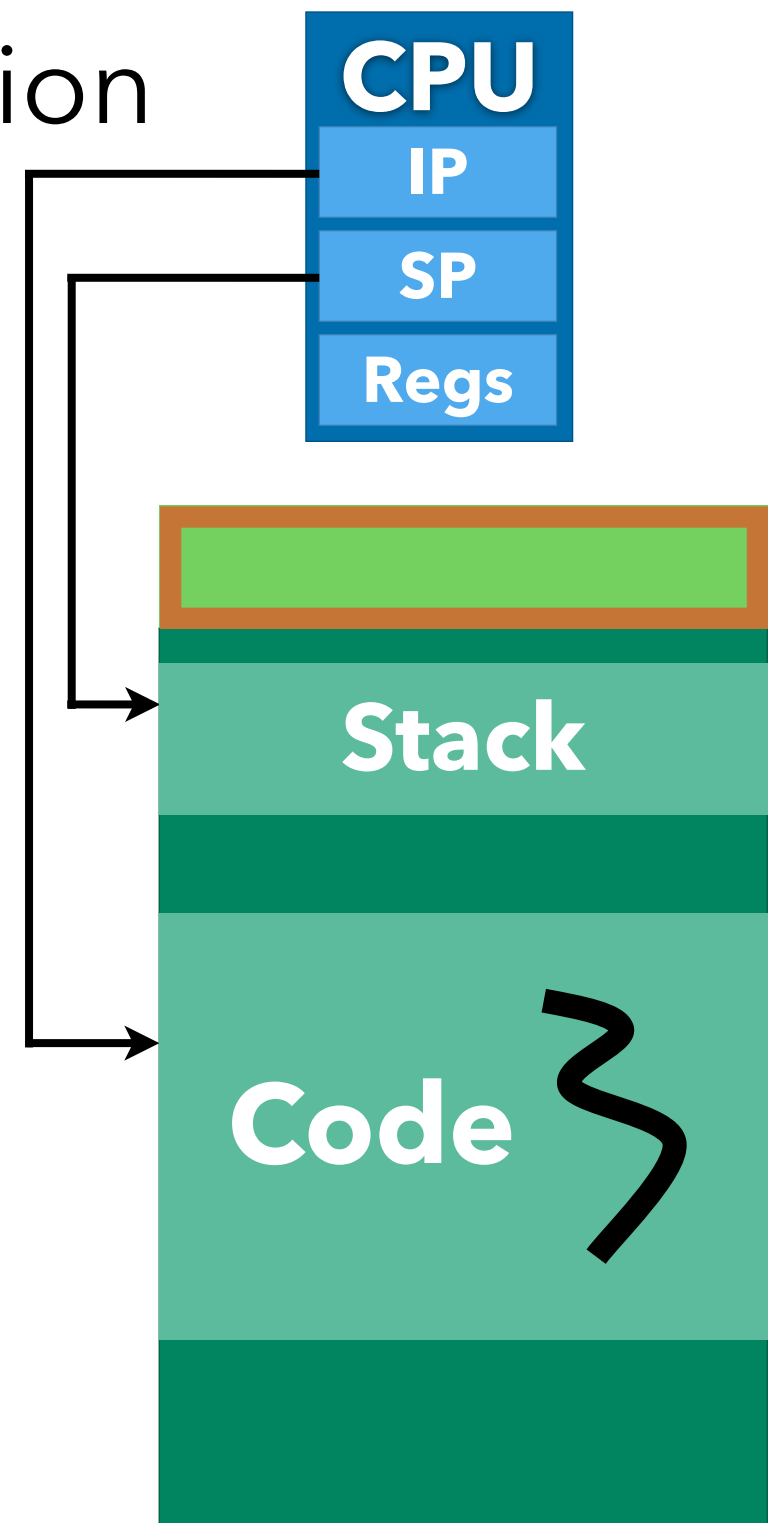
ALTERNATIVES

user
 shared system
 privileged

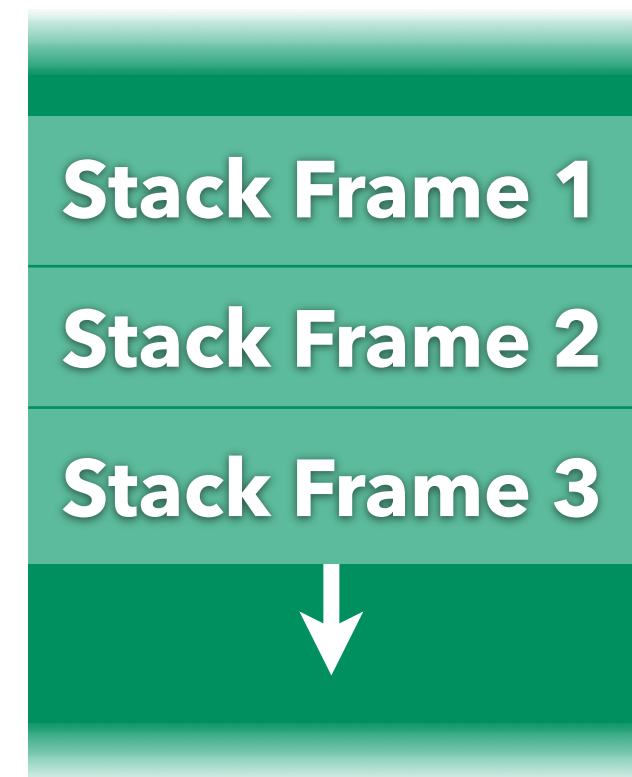


THREADS

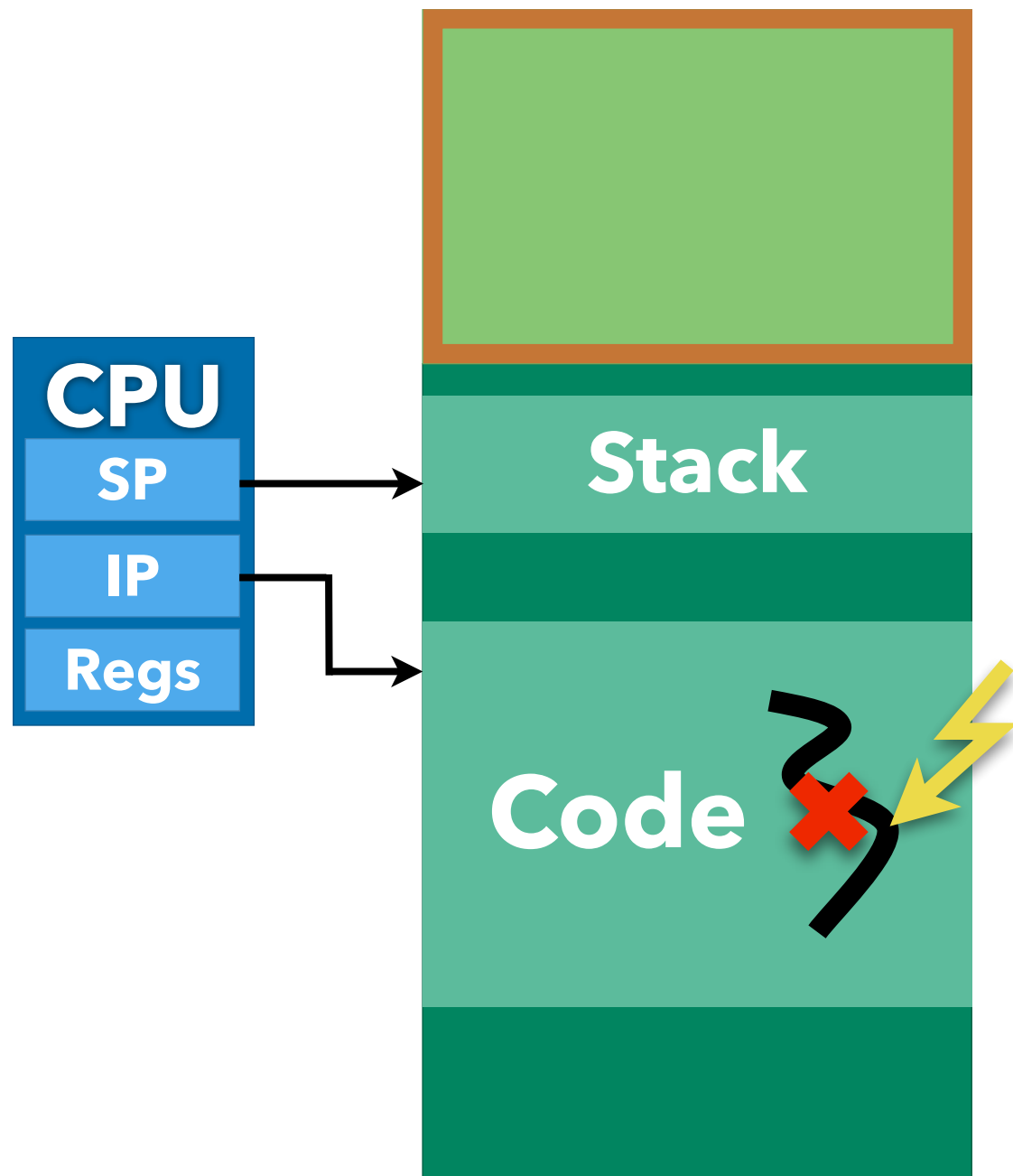
- abstraction of code execution
- unit of scheduling
- provides temporal isolation
- typically requires a stack
- thread state:
 - instruction pointer
 - stack pointer
 - CPU registers, flags



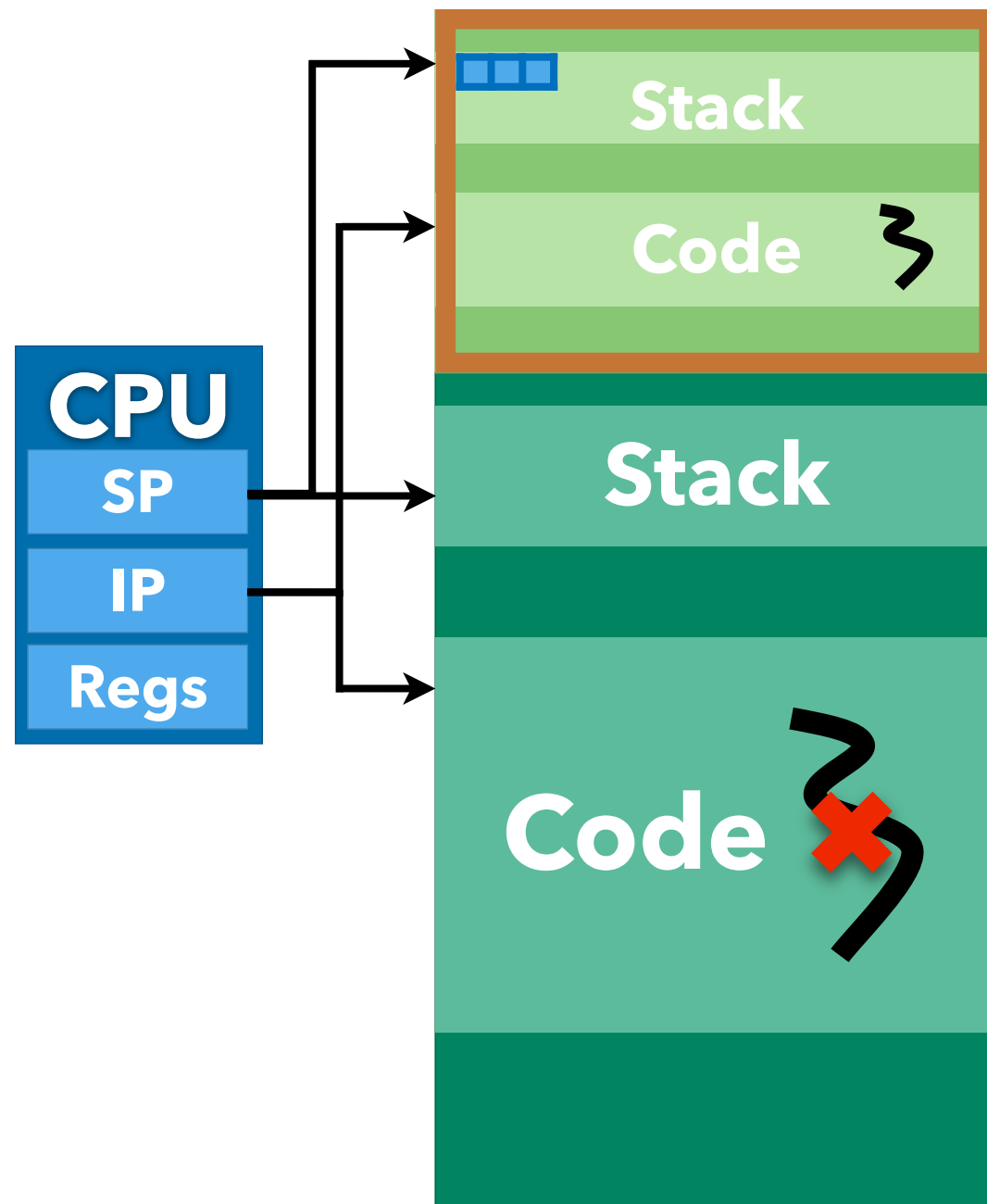
- storage for function-local data
 - local variables
 - return address
- one stack frame per function
- grows and shrinks dynamically
- grows from high to low addresses



- maps user-level threads to kernel-level threads
 - often a 1:1 mapping
 - threads can be implemented in userland
- assigns threads to hardware
- one kernel-level thread per logical CPU
- with hyper-threading and multicore, we have more than one hardware context



- thread can enter kernel:
- voluntarily
 - system call
- forced
 - interrupt
 - exception



- IP and SP point into kernel
- user CPU state stored in TCB
 - old IP and SP
 - registers
 - flags
 - FPU state
 - MMX, SSE, AVX

- thread control block
- kernel object, one per thread
- stores thread's userland state while it is not running
- untrusted parts can be stored in user space
 - separation into KTCB (kernel TCB) and UTCB (user TCB)
 - UTCB also holds system call parameters

- once the kernel has provided its services, it returns back to userland
- by restoring the saved user IP and SP
- the same thread or a different thread
- the old thread may be blocking now
 - waiting for some resource
- returning to a different thread might involve switching address spaces

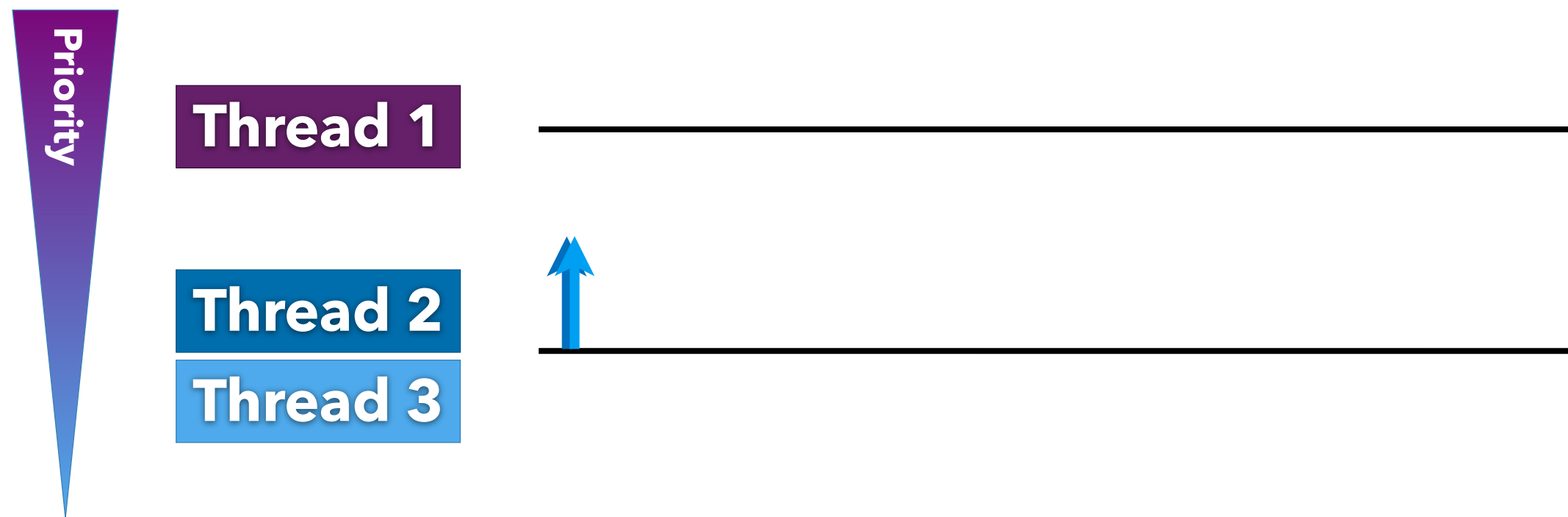
SCHEDULING

- scheduling describes the decision, which thread to run on a CPU at a given time
- When do we schedule?
 - current thread blocks or yields
 - time quantum expired
- How do we schedule?
 - RR, FIFO, RMS, EDF
 - based on thread priorities

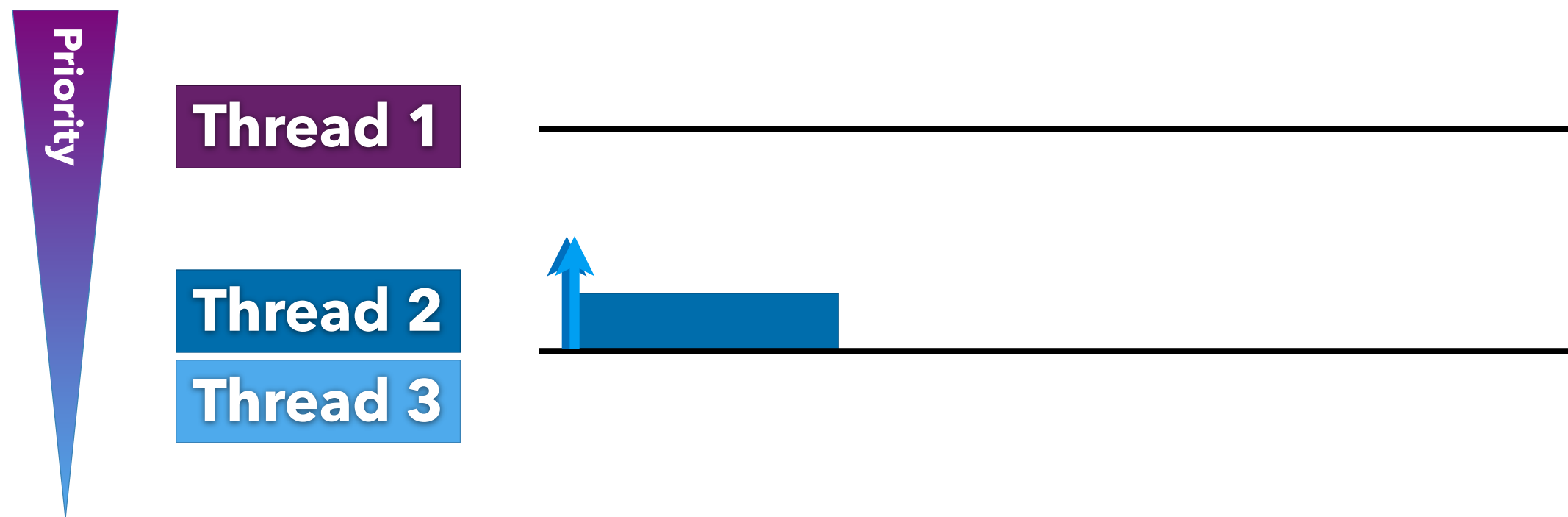
- scheduling decisions are policies
- should not be in a microkernel
- L4 used to have facilities to implement scheduling in user land
 - each thread has an associated preempter
 - kernel sends an IPC when thread blocks
 - preempter tells kernel where to switch to
- no efficient implementation yet
- scheduling is the only in-kernel policy in L4

- scheduling in L4 is based on thread priorities
- time-slice-based round robin within the same priority level
- kernel manages priority and timeslice as part of the thread state
- see scheduling lecture

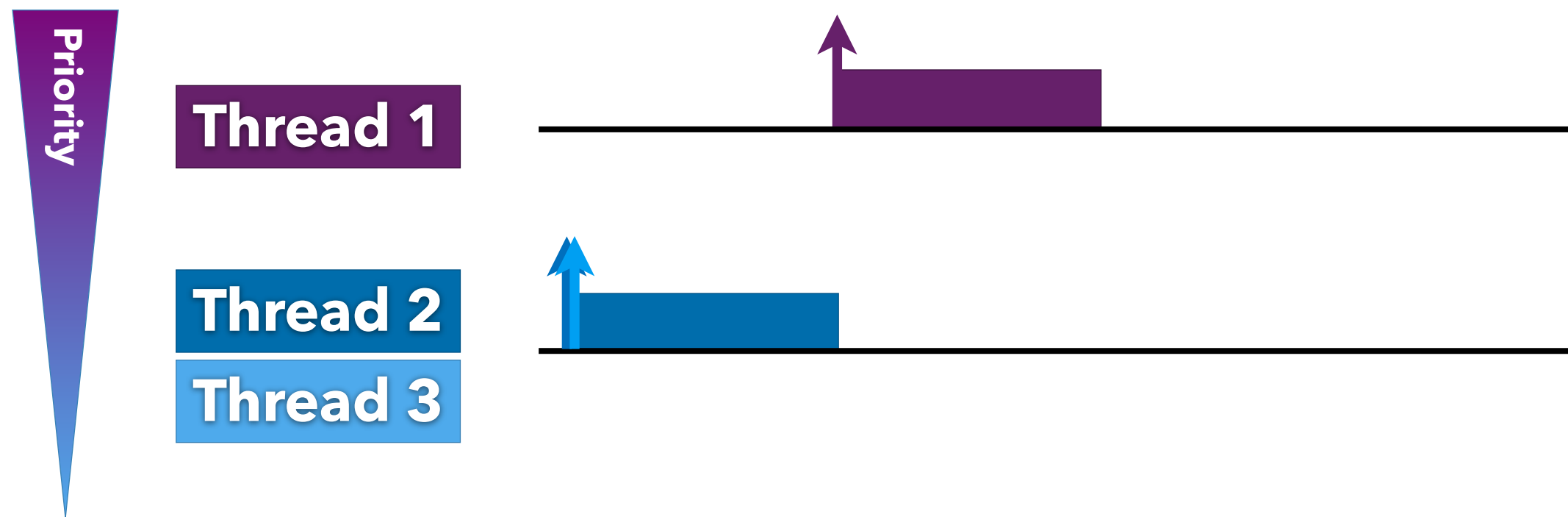
- thread 1 is a high priority driver thread, waiting for an interrupt (blocking)
- thread 2 and 3 are ready with equal priority



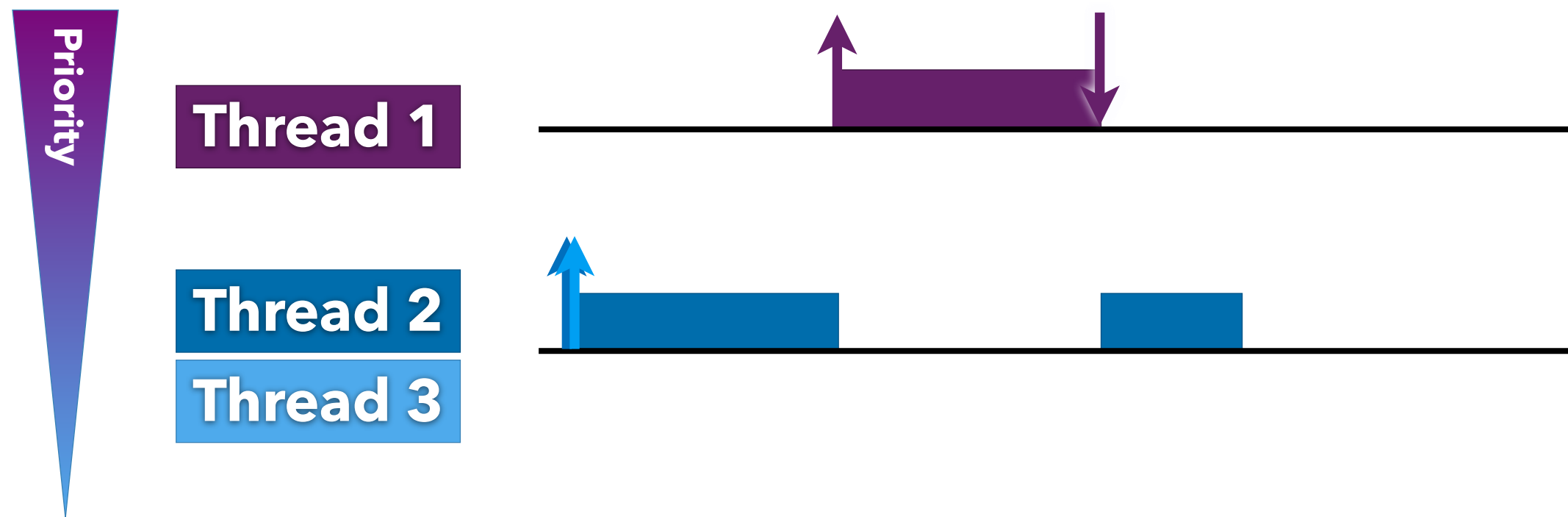
- 1 hardware context
- kernel fills time slices of threads 2 and 3
- scheduler selects 2 to run



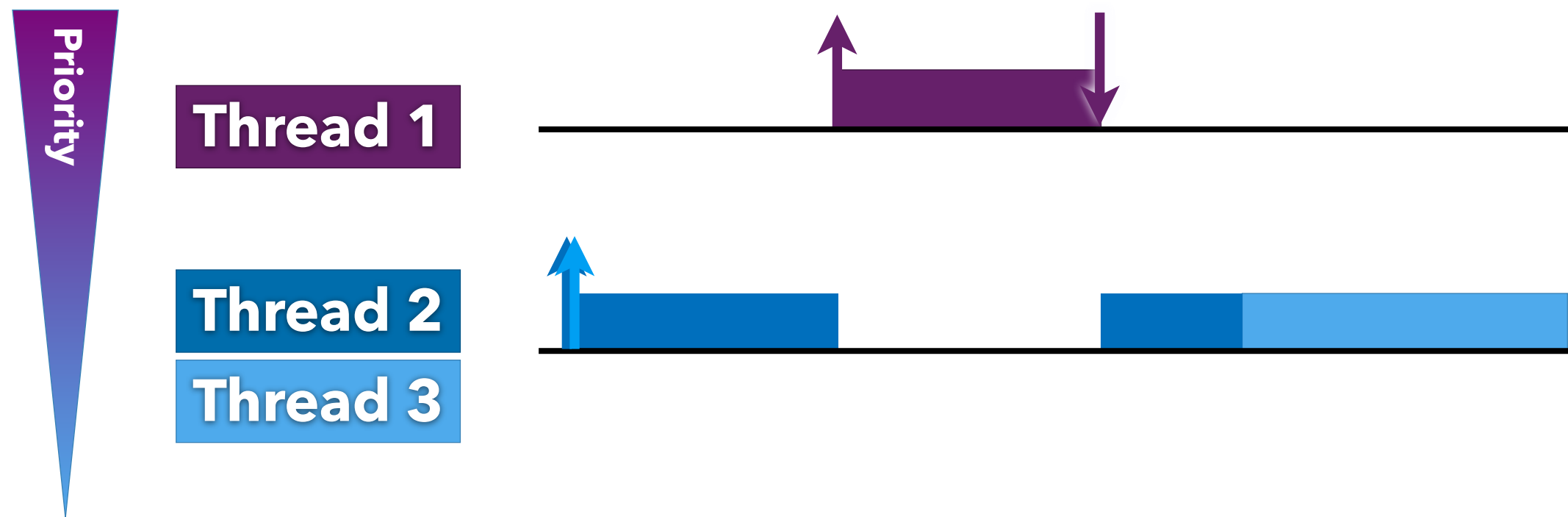
- device interrupt arrives
- thread 2 is forced into the kernel, where it unblocks thread 1 and fills its time slice
- switch to thread 1 preempts thread 2



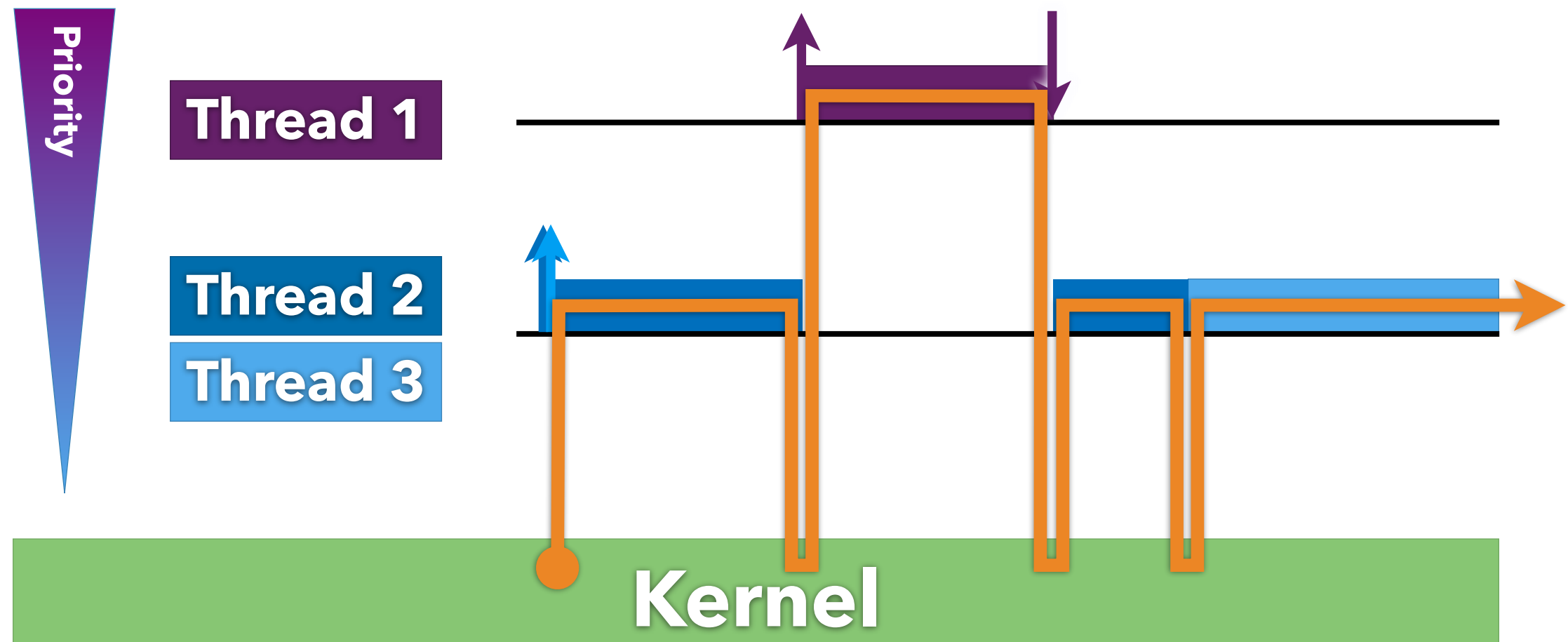
- thread 1 blocks again (interrupt handled, waiting for next)
- thread 2 has time left



- thread 2's time slice has expired
- timer interrupt forces thread 2 into kernel
- scheduler selects the next thread on the same priority level (round robin)



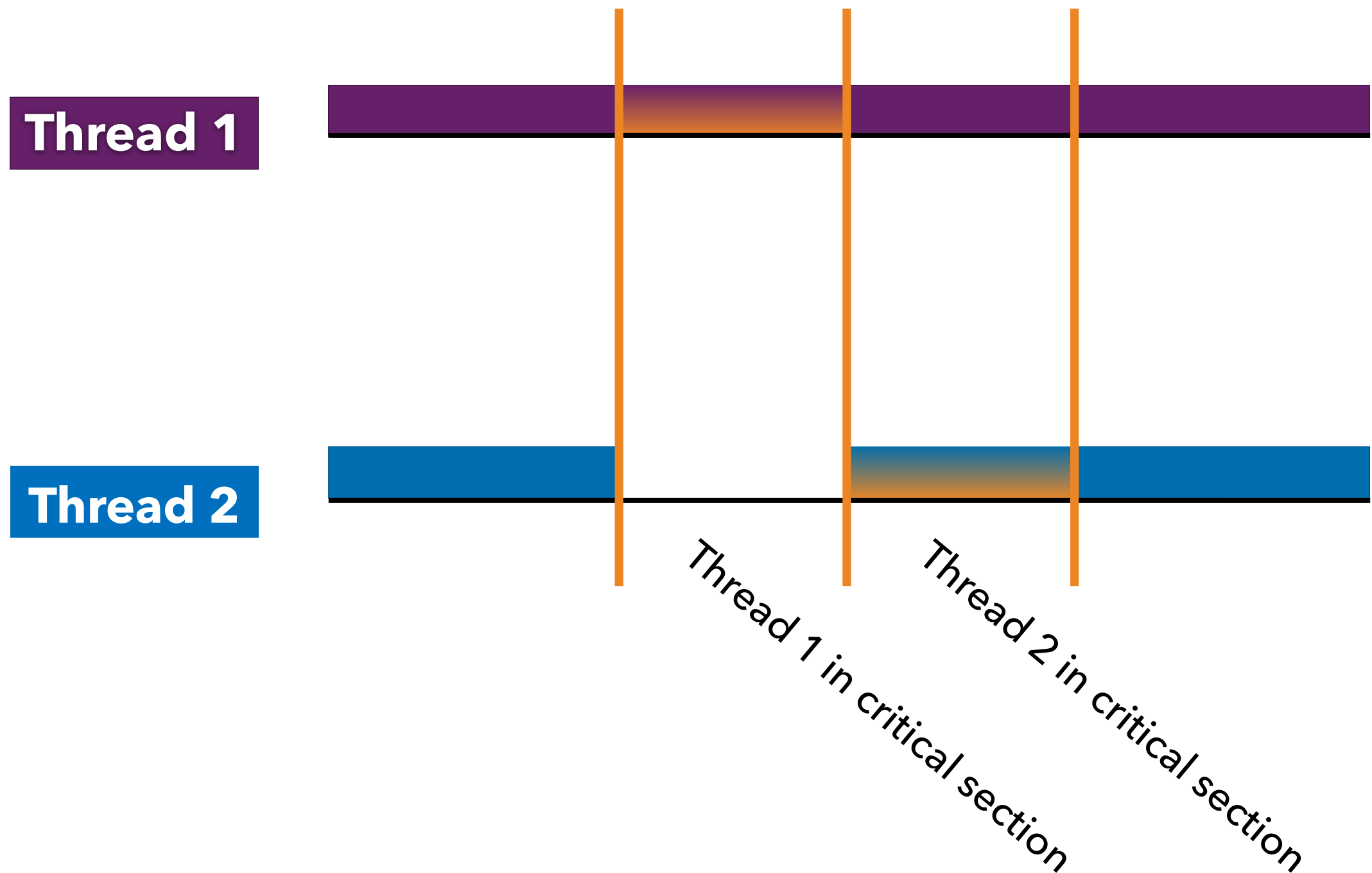
it's really only one hardware thread being multiplexed

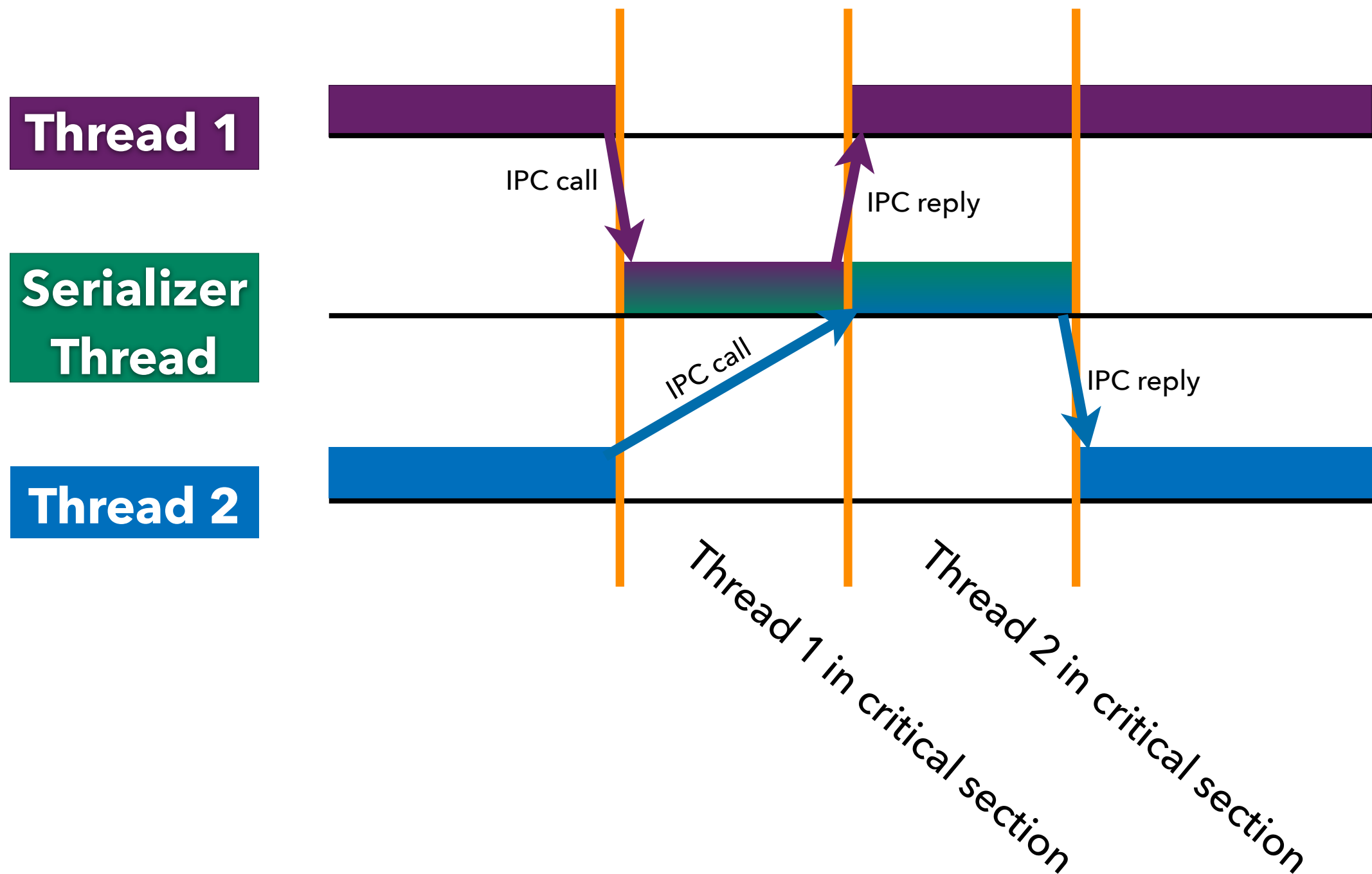


SYNCHRONIZATION

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

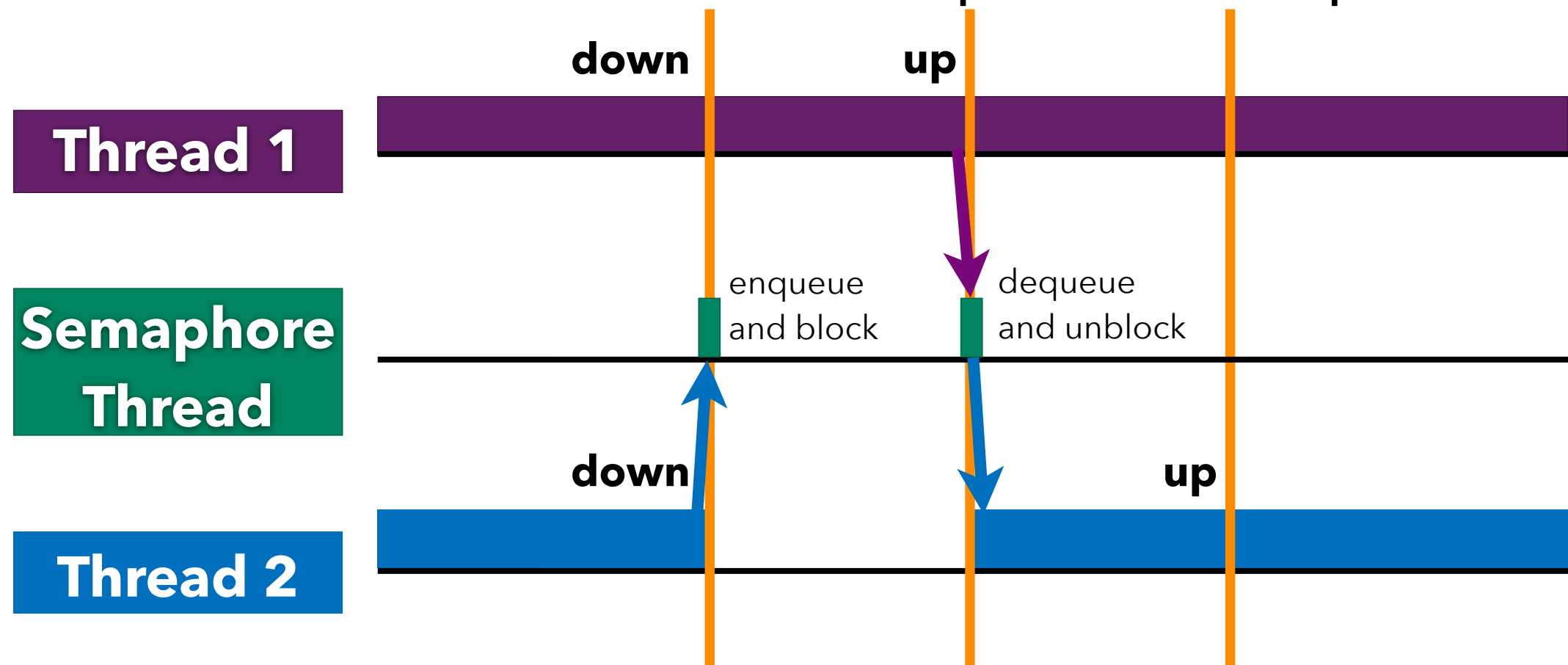
- for concurrent access to data structures
- use atomic operations to protect manipulations
- only suited for simple critical sections





- 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

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



- 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
- optimisation for small critical sections in scheduling lecture

- repeated basic microkernel concepts
 - paradigm, resource abstractions
- closer look on threads
 - TCB, kernel entry
- scheduling
 - time slices, priorities, preemption
- synchronization
 - atomic ops, serializer thread, semaphore
- next up: memory