TECHNISCHE
UNIVERSITÄT
DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

OPERATING-SYSTEM CONSTRUCTION

Material based on slides by Olaf Spinczyk, Universität Osnabrück

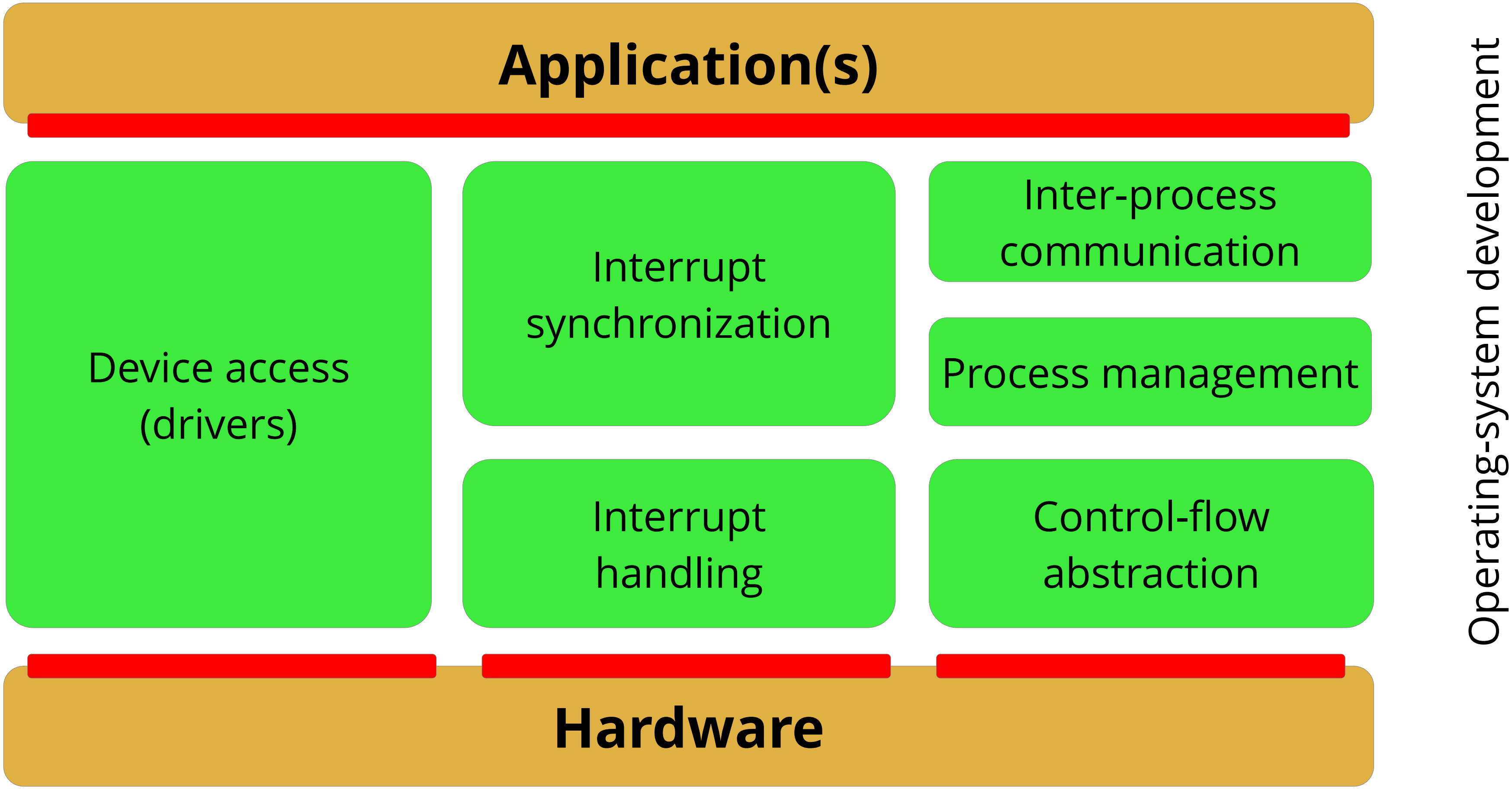
Scheduling

https://tud.de/inf/os/studium/vorlesungen/betriebssystembau

HORST SCHIRMEIER

Overview: Lectures

Structure of the “OO-StuBS” operating system:



Application(s)

Device access (drivers)

Interrupt synchronization

Inter-process communication

Process management

Interrupt handling

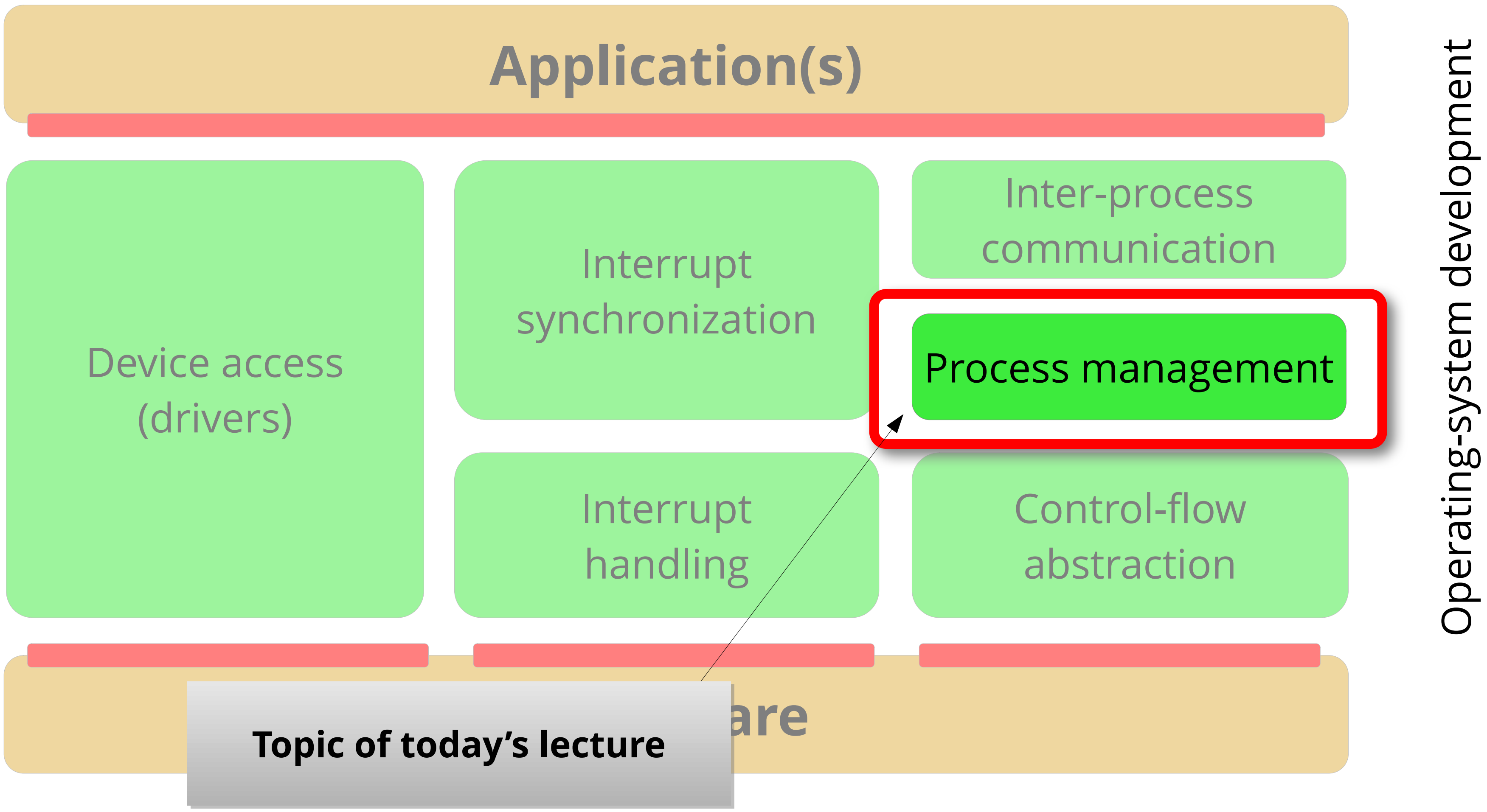
Control-flow abstraction

Hardware

Operating-system development

Overview: Lectures

Structure of the “OO-StuBS” operating system:



Topic of today’s lecture

Agenda

- Kernel-Level Threads
  - Motivation
  - Cooperative Thread Switch
  - Preemptive Thread Switch
- Scheduling
  - Basic Terms and Classification
  - in Windows (8–11)
  - in Linux
- Summary

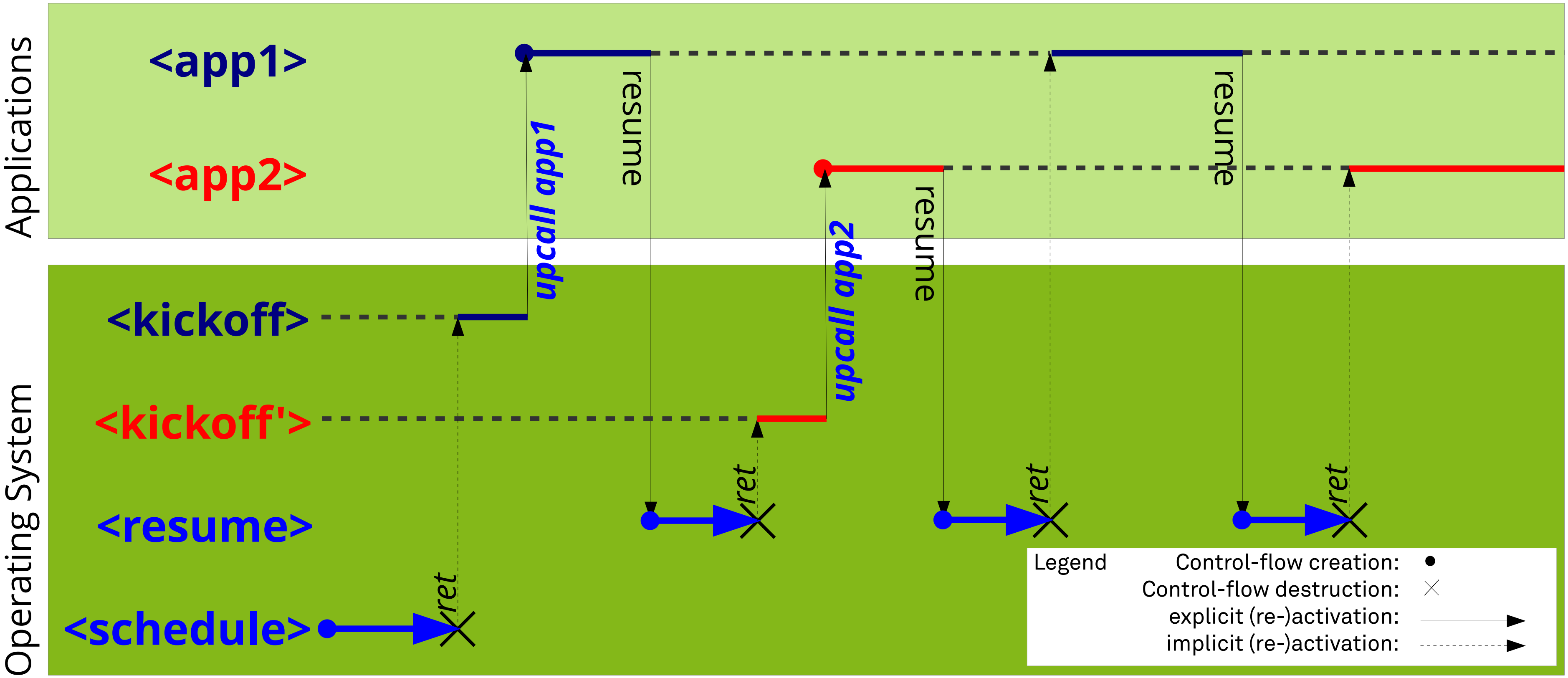
Agenda

- **Kernel-Level Threads**
  - Motivation
  - Cooperative Thread Switch
  - Preemptive Thread Switch
- Scheduling
  - Basic Terms and Classification
  - in Windows (8–11)
  - in Linux
- Summary

Kernel-Level Threads: Motivation

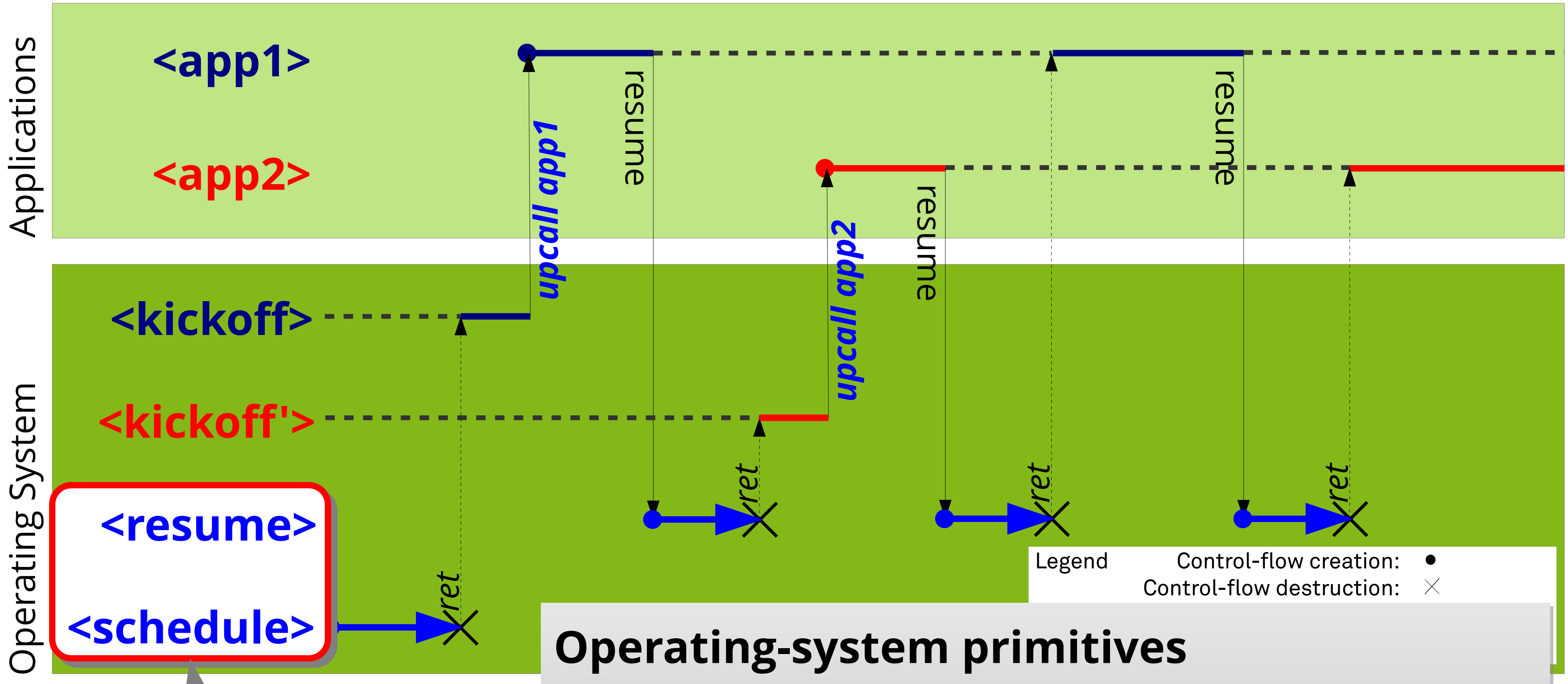
- **Approach:** Run applications “unnoticeably” as independent threads
  - One OS coroutine per application
  - Application is activated by being called
  - Coroutine switch: indirect by system call
- **Advantages:**
  - Independent application development
  - Central **scheduler** implementation
  - An application waiting for I/O can be “blocked” by the OS and “awakened” later
  - Additional **preemption mechanism** can prevent CPU monopolization

Cooperative Thread Switch



Legend
Control-flow creation: ●
Control-flow destruction: ×
explicit (re-)activation: ⟶
implicit (re-)activation: ⇢

Cooperative Thread Switch

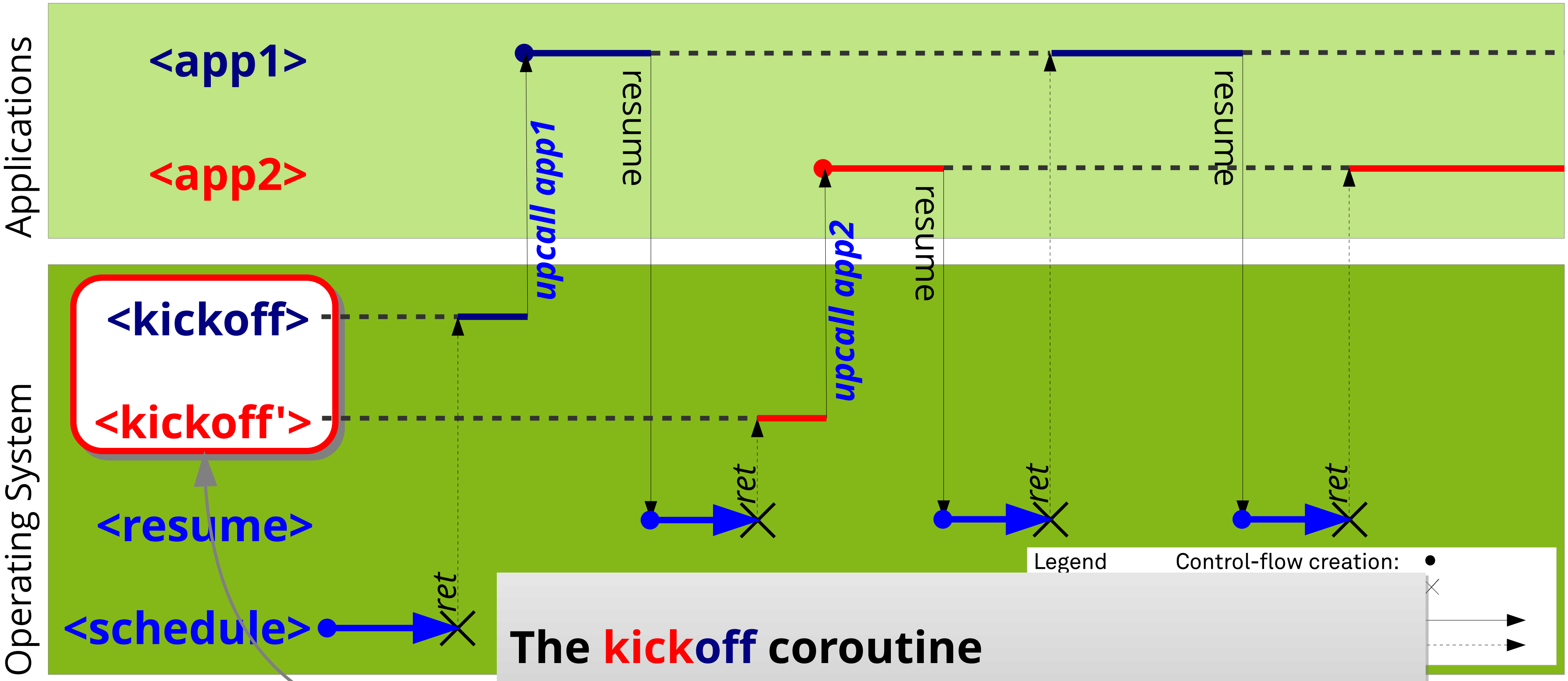


Operating-system primitives

- **schedule** (indirectly) starts the first application thread, does not return
- **resume** switches from one application thread to the next.

Both make a **scheduling decision**.

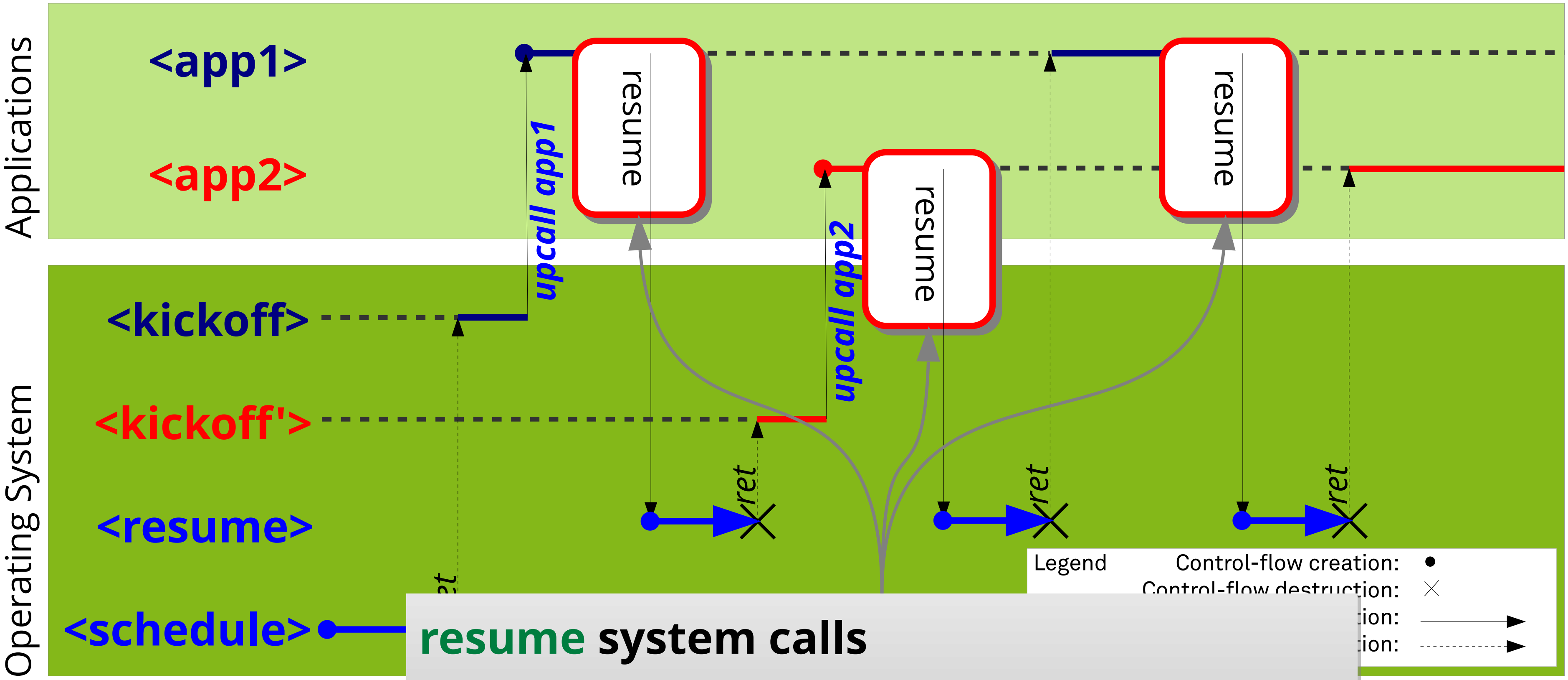
Cooperative Thread Switch



The kickoff coroutine

- runs once for each application thread
- activates the respective application through an **upcall**

Cooperative Thread Switch

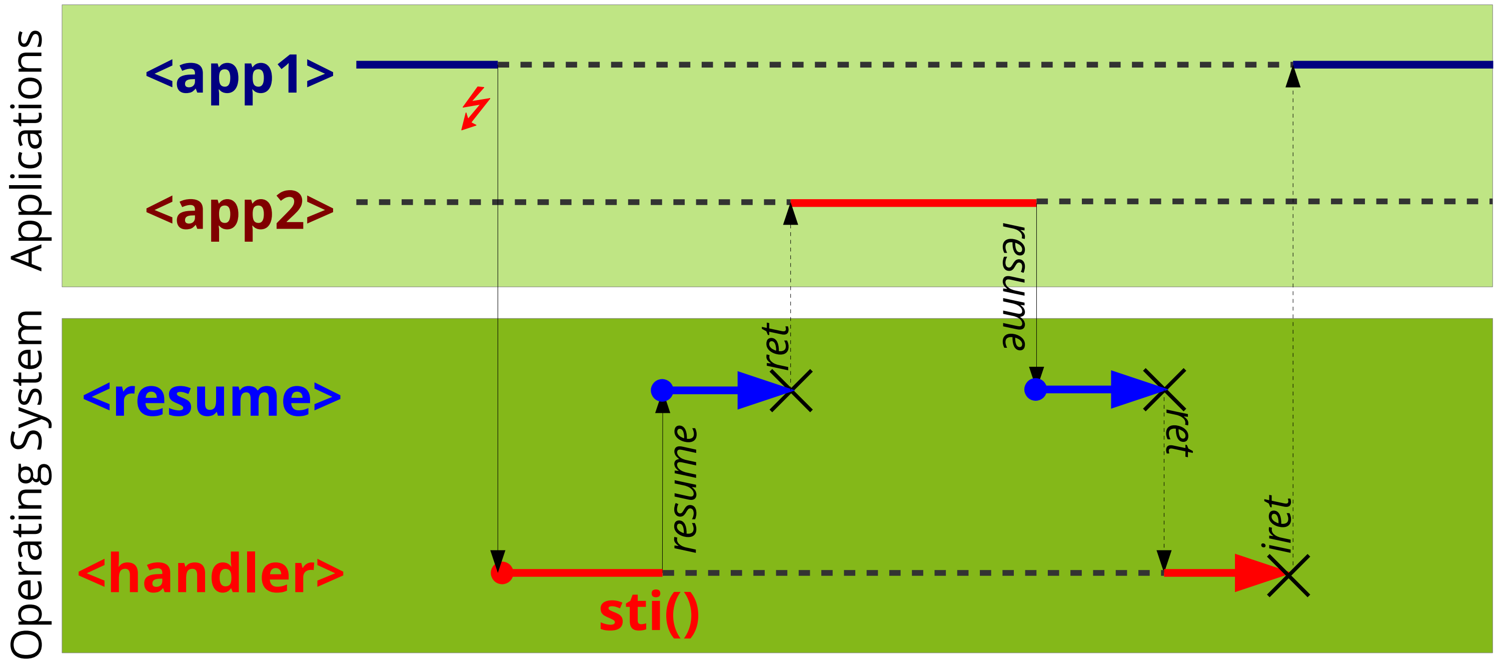


resume system calls

- the mechanism for applications to yield the CPU voluntarily
- possibly combined with a CPU mode switch (in this case we additionally need a wrapper)

Preemptive Thread Switch

- Forced CPU removal via timer interrupt
  - the interrupt is “just” an implicit call
  - handler routine can call *resume*



Careful: In general it does not work this way, because **resume** makes a **scheduling decision**. We need to apply **interrupt synchronization** for the involved data structures!

Thread Switch in the Epilogue

- Implementation
  - Scheduler data (list of ready threads) reside on the epilogue level
  - All system functions that manipulate these data
    must acquire the epilogue lock before (enter/leave)
    - Create thread, terminate thread, voluntary thread switch, ...
- Basic rule for thread switches:
  - the **yielding thread** requests the lock
    (e.g. implicitly in interrupt handling)
  - the **activated thread** must release the lock
- Tips:
  - Never call **enter** from the epilogue (double request)
  - Basic rule (see above) also holds **for the first** thread activation(!)

More on that in the exercises.

Agenda

- Kernel-Level Threads
    - Motivation
    - Cooperative Thread Switch
    - Preemptive Thread Switch
- **Scheduling**
    - Basic Terms and Classification
    - in Windows (8–11)
    - in Linux
- Summary

Scheduling: Classification by …

- **Resource type** of the scheduled hardware resource
- **Operation mode** of the controlled computer system
- **Point in time** when the schedule is determined
- **Determinism** of timing and duration of process runs
- **Cooperation behavior** of (user/system) programs
- **Computer architecture** of the system
- **Decision-making level** when scheduling resources

… by Resource Type

- **CPU scheduling** of the resource “CPU”
  - Process count at times higher than CPU count
  - CPU(s) must be multiplexed for several processes
  - Admission via waiting queue
- **I/O scheduling** of the resource “device”, particularly “disk”
  - Device-specific scheduling of I/O jobs generated by processes
  - e.g., **disk scheduling** usually takes into account these factors:
    - (1) Positioning time, (2) rotation time, (3) transfer time
  - Device parameters and device state determine the next I/O operation
  - Scheduling decisions possibly not conforming to CPU scheduling

… by Mode of Operation

- **Batch scheduling** of interaction-less programs
  - **non-preemptive** scheduling
    (or preemptive scheduling with long time slices)
  - Context-switch count minimization
- **Interactive scheduling** of interactive programs
  - **Event-driven**, **preemptive** scheduling with short time slices
  - Partly response-time minimization by heuristics
- **Real-time scheduling** of time-critical programs
  - Event- or time-driven **deterministic** scheduling
  - Guarantee of keeping environment-specific deadlines
  - Focus: **Timeliness**, not performance

… by Point in Time

- **Online scheduling** dynamic, during actual program execution
  - Interactive and batch systems, but also soft real-time systems
- **Offline scheduling** static, before actual program execution
  - If **complexity** prohibits scheduling at runtime
    - Guarantee keeping all deadlines: NP-hard
    - Critical if we must react to any preventable catastrophic situation
  - Result: Complete schedule (in tabular form)
    - (Half) automatically generated via source-code analysis of a specialized “compiler”
    - Often executed by a time-triggered scheduler
  - Usually limited to hard real-time systems

… by Determinism

- **Deterministic scheduling** of known, exactly pre-computed processes
  - **Process runtimes and deadlines are known**, possibly calculated offline
  - Exact prediction of CPU load is possible
  - System guarantees and enforces process runtimes/deadlines
  - Time guarantees are valid regardless of system load
- **Probabilistic scheduling** of unknown processes
  - **Process runtimes and deadlines are unknown**
  - (Probable) CPU load can only be estimated
  - System cannot give and enforce time guarantees
  - Timing guarantees conditionally achievable by application mechanisms

… by Cooperation Behavior

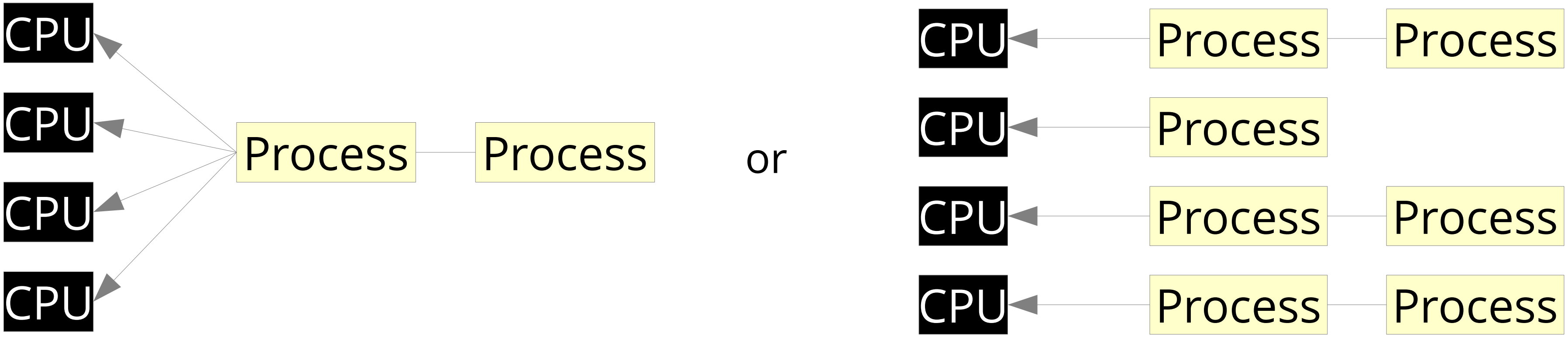
- **Cooperative scheduling** of interdependent processes
  - Processes must **voluntarily yield the CPU** in favor of other processes
  - Program execution must (directly/indirectly) trigger **system calls**
  - System calls must (directly/indirectly) activate the scheduler
- **Preemptive scheduling** of independent processes
  - Processes are forcibly **deprived of the CPU** in favor of other processes
  - **Events** can trigger preemption of the running process
  - Event processing (directly/indirectly) activates the scheduler

… by Computer Architecture

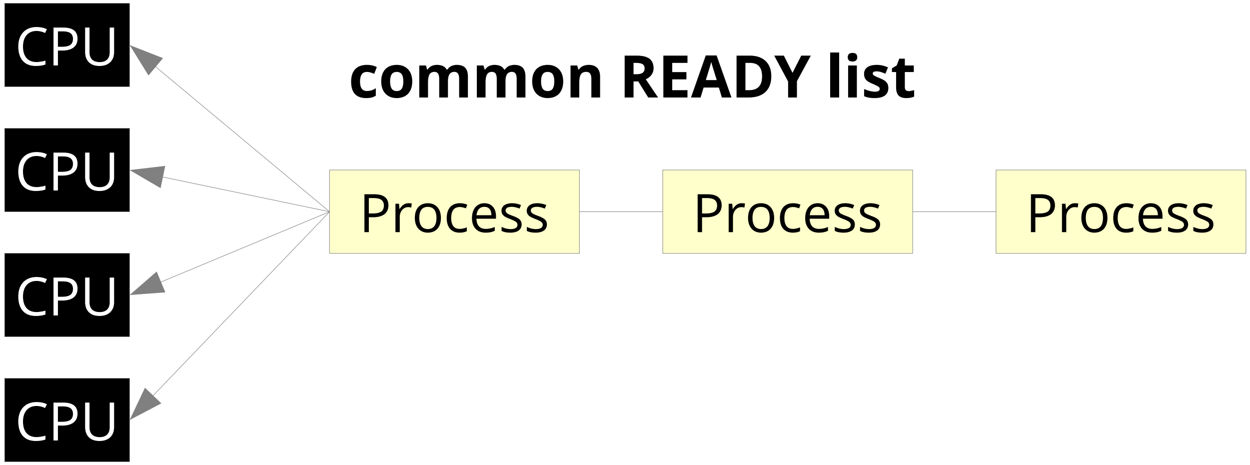
- **Uni-processor scheduling**

  in multiprogramming/processing systems
  - Process execution only pseudo parallel
- **Multi-processor scheduling**

  in shared-memory systems
  - Parallel process execution possible
    - Each processor processes its **local ready list**
    - All processors process **one global ready list**

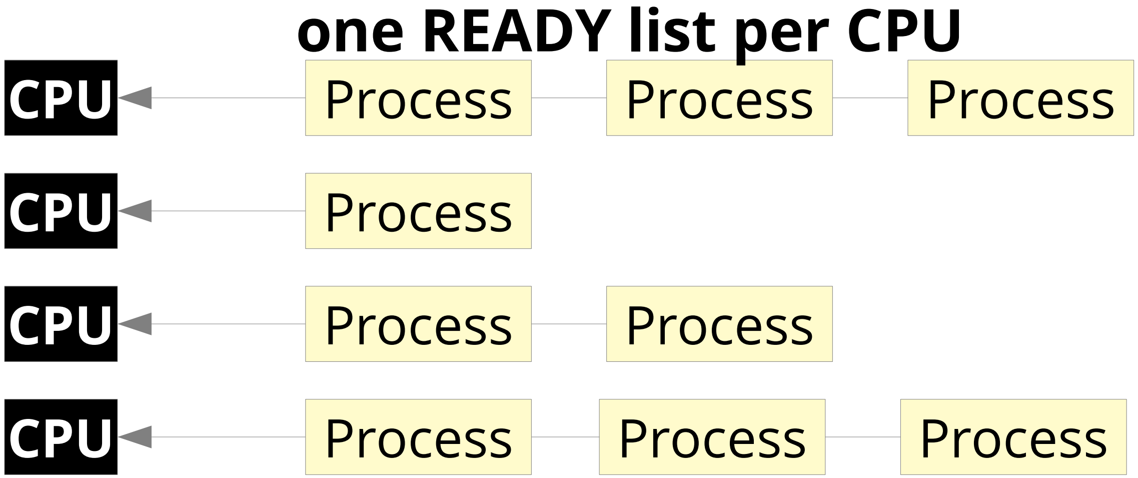


Multiprocessor CPU Scheduling



- Automatic load balancing
  - No CPU runs empty
- Processes are not bound to particular CPUs
- Accesses to the READY list must be synchronized
  - **Spinlock**
  - Conflict probability grows with CPU count!

Multiprocessor CPU Scheduling



- Processes stay on one CPU
  - Better cache utilization
- Less synchronization costs
- CPU can drain (empty list)
  - Solution: On-demand load balancing (**pull**)
    - When a READY list is empty
  - By a load-balancer process (**push**)

Modern PC operating systems nowadays use **separate READY lists.**

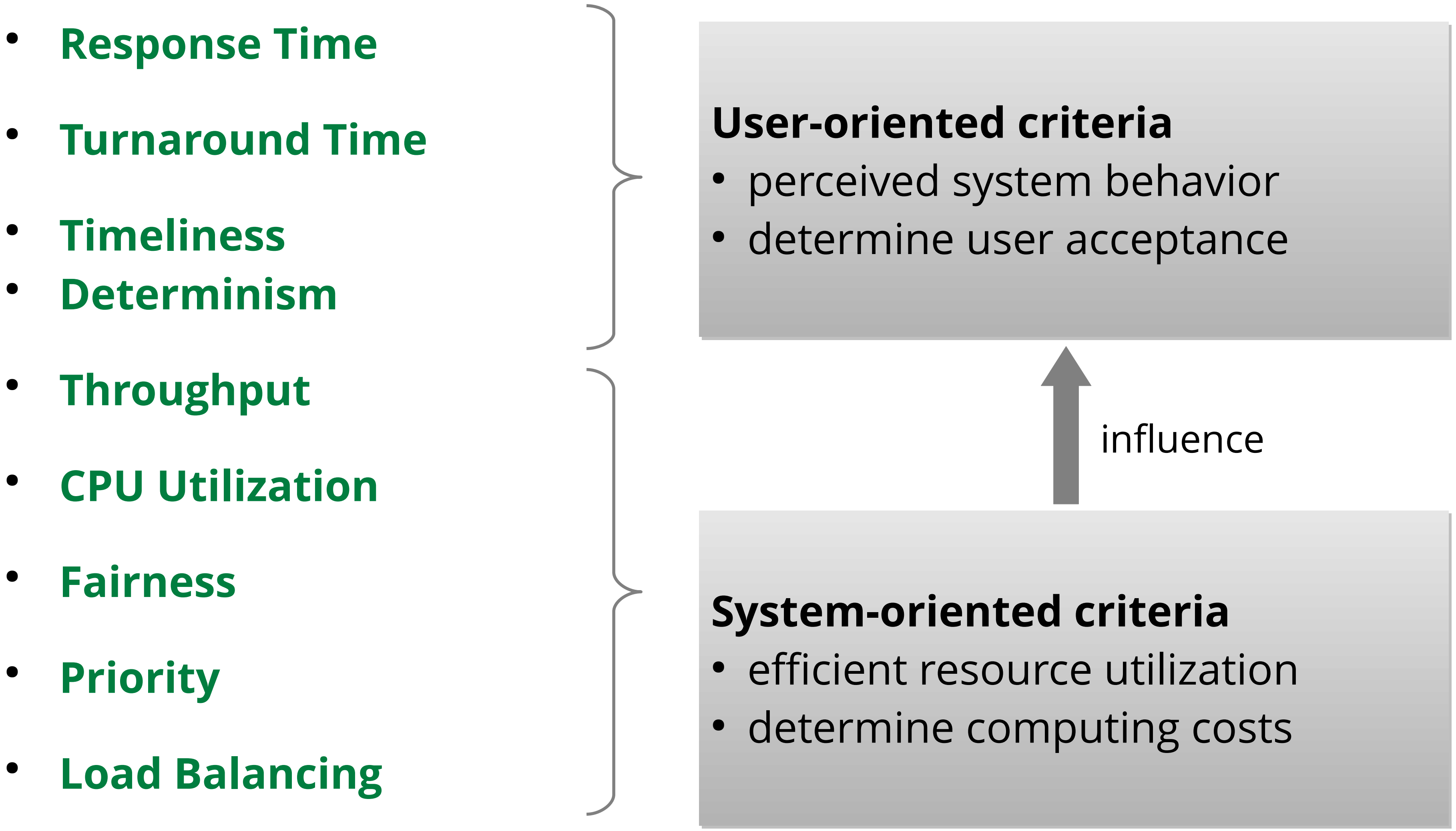
… by Decision-Making Level

- **Long-term scheduling** controls the degree of multiprogramming **[s – min]**
  - Admission for users and processes
  - Hand over processes to medium- and short-term scheduling
- **Medium-term scheduling** as part of swapping **[ms – s]**
  - Move processes back and forth between RAM and disk
  - **swapping**: swap-out, swap-in
- **Short-term scheduling** schedules processes on the CPU(s)
  - Event-driven scheduling: Interrupts, system calls, signals **[µs – ms]**
  - Blocking / preemption of the running process

Scheduling Criteria

- **Response Time** Minimizing the time from a system-service **request until the response**, while maximizing the number of interactive processes.
- **Turnaround Time** Minimizing the time between **process submission and completion**, i.e. the effective process runtime and all waiting times.
- **Timeliness** Start and/or termination of a process at **fixed points in time**.
- **Determinism** Deterministic execution of a process **regardless of the current system load**.
- **Throughput** Maximizing the number of completed processes **per predefined time unit**. A measure for the performed “work” in a system.
- **CPU Utilization** Maximizing the **percentage of time** the CPU executes processes, i.e. does useful work.
- **Fairness** Equal treatment of processes, and guarantee to schedule processes within certain time frames (no starvation).
- **Priority** Executing processes with the highest (statically/dynamically assigned) priority first.
- **Load Balancing** Uniform resource utilization, or prioritized execution of processes that rather seldomly allocate heavily utilized resources.

Scheduling Criteria



- **Response Time**
- **Turnaround Time**
- **Timeliness**
- **Determinism**
- **Throughput**
- **CPU Utilization**
- **Fairness**
- **Priority**
- **Load Balancing**

User-oriented criteria
- perceived system behavior
- determine user acceptance

influence

System-oriented criteria
- efficient resource utilization
- determine computing costs

Operating Modes and Criteria

- in general
  - Fairness
  - Load balancing
- **Batch systems**
  - Throughput
  - Turnaround time
  - CPU utilization
- **Interactive systems**
  - Response time (Proportionality – Processing time corresponds to expectation)
- **Real-time systems**
  - Priority
  - Timeliness
  - Determinism

Agenda

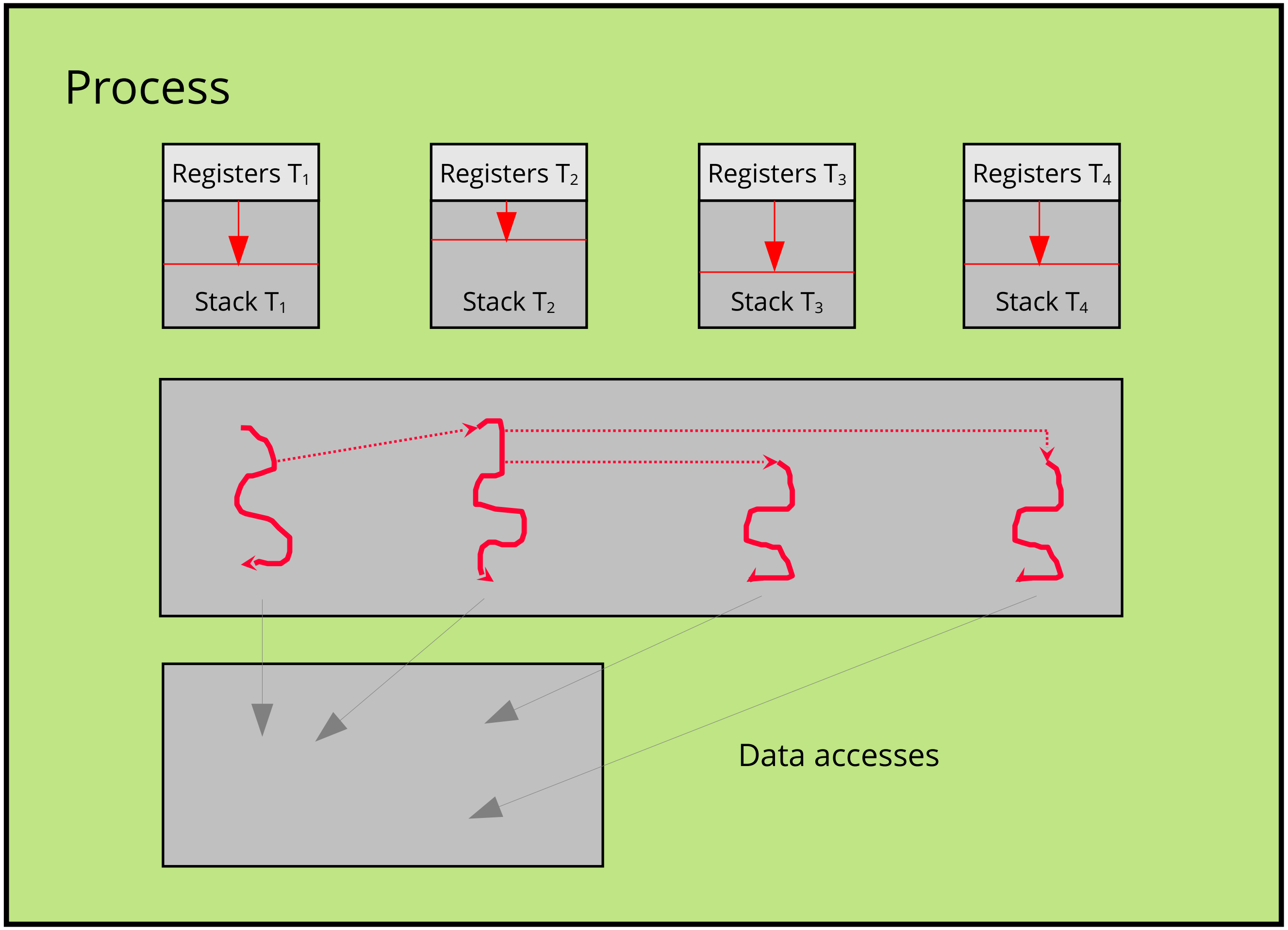
- Kernel-Level Threads
  - Motivation
  - Cooperative Thread Switch
  - Preemptive Thread Switch
- Scheduling
  - Basic Terms and Classification
  - **in Windows (8–11)**
  - in Linux
- Summary

Processes and Threads in Windows

Stack +
Register file
(1 per thread)

Code

Global and
static data



Processes and Threads in Windows

- **Process:** Environment and address space for threads
  - A Win32 process always contains at least one thread
  - **Thread:** Code-executing entity
- Thread implementation by NT kernel
  - User-mode threads possible (“**fibers**”), but unusual
- Scheduler assigns processing time to threads

The Windows Scheduler

- Preemptive, priority-based scheduling:
  - High-priority thread preempts thread with lower priority
    - Regardless whether thread currently in user or kernel mode
    - Most functionality of the **Executive** (“kernel”) implemented as threads, too
  - **Round-Robin** for threads with same priority
    - Round-robin assignment of one time slice (“**Quantum**”)
- Thread priorities
  - 0 to 31, subdivided in three ranges
    - **Variable Priorities**: 1 to 15
    - ***Real-time Priorities***: 16 to 31
    - Priority 0 is reserved for the Zero-Page Thread
  - Threads of the Executive maximally use priority 23

Time Slice (*Quantum*)

- *Quantum* is decreased
  - by 3 at every clock tick (every 15 ms)
  - by 1 if the thread voluntarily enters a waiting state
- Time-slice length: 20 – 180 ms

	short Quantum values (Desktop)				**long Quantum values (Server)**			
	variable			**fixed**	**variable**			**fixed**
Thread in backgr. process	6			18	12			36
Thread in FG process	6	12	18	18	12	24	36	36

Priority Classes, Relative Thread Priority

		Process Priority Class					
Relative Thread Priority		Idle	Below Normal	Normal	Above Normal	High	Realtime
		4	6	8	10	13	24
Time Critical	=15	15	15	15	15	15	31
Highest	+2	6	8	10	12	15	26
Above Normal	+1	5	7	9	11	14	25
Normal		**4**	**6**	**8**	**10**	**13**	24
Below Normal	-1	3	5	7	9	12	23
Lowest	-2	2	4	6	8	11	22
Idle	=1	1	1	1	1	1	16

Priorities: *Variable Priorities*

- **Variable Priorities** (1–15)
  - Scheduler strategies to prioritize “important” threads
    - **Quantum Stretching** (preference for the active GUI thread, cf. 2 slides back)
    - Dynamic **priority boost** for a few time slices at events
  - Progress guarantee
    - Every 3 to 4 seconds, up to 10 “disadvantaged” threads are raised to priority 15 for two time slices
  - Thread priority is calculated using this (simplified) formula:

Process priority class + Thread priority + Boost

Priorities: *Realtime Priorities*

- **Realtime Priorities** (16–31)
  - Pure priority-based **Round-Robin**
    - No progress guarantee
    - No dynamic boost
    - Operating system itself can be negatively affected
    - Special user privilege necessary (`SeIncreaseBasePriorityPrivilege`)

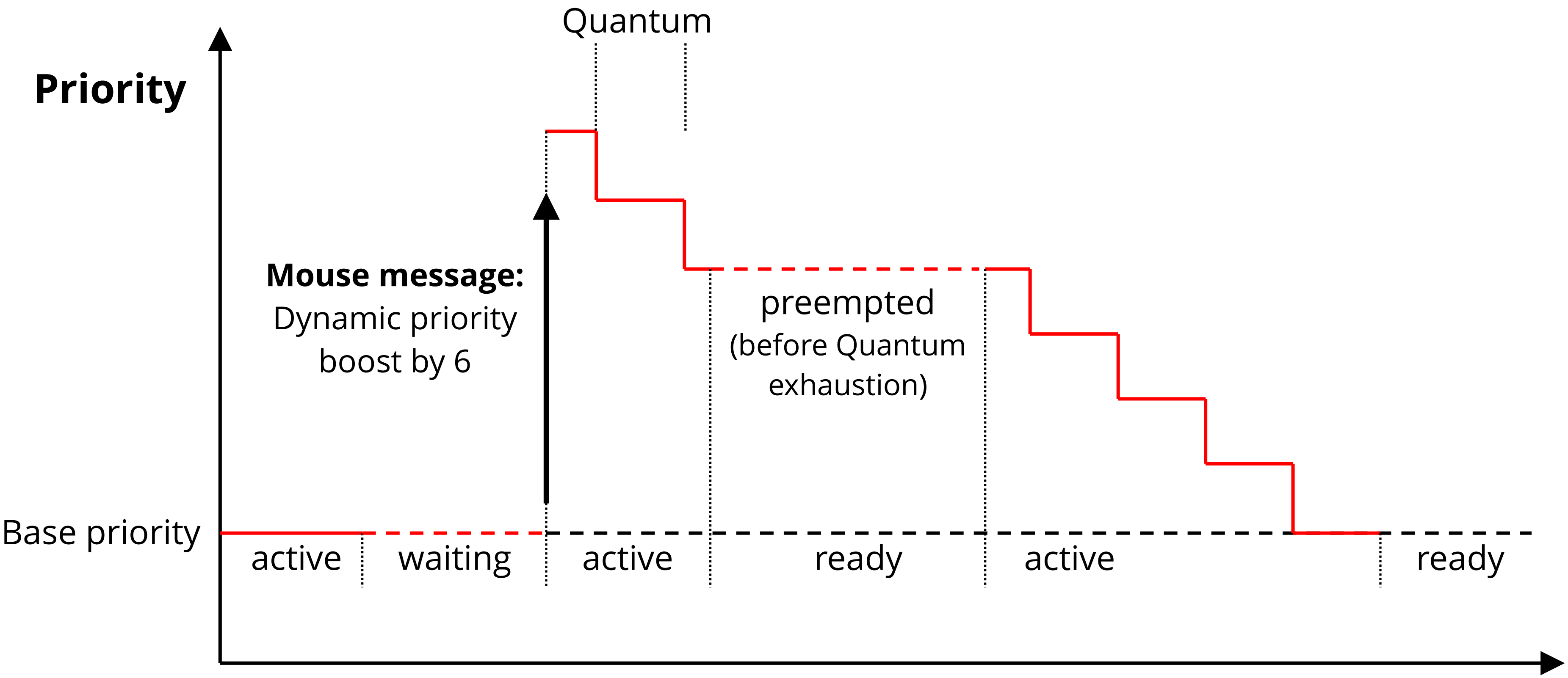
  - Thread priority is calculated using this formula:

REALTIME_PRIORITY_CLASS + Thread priority

Dynamic Priority Boosts

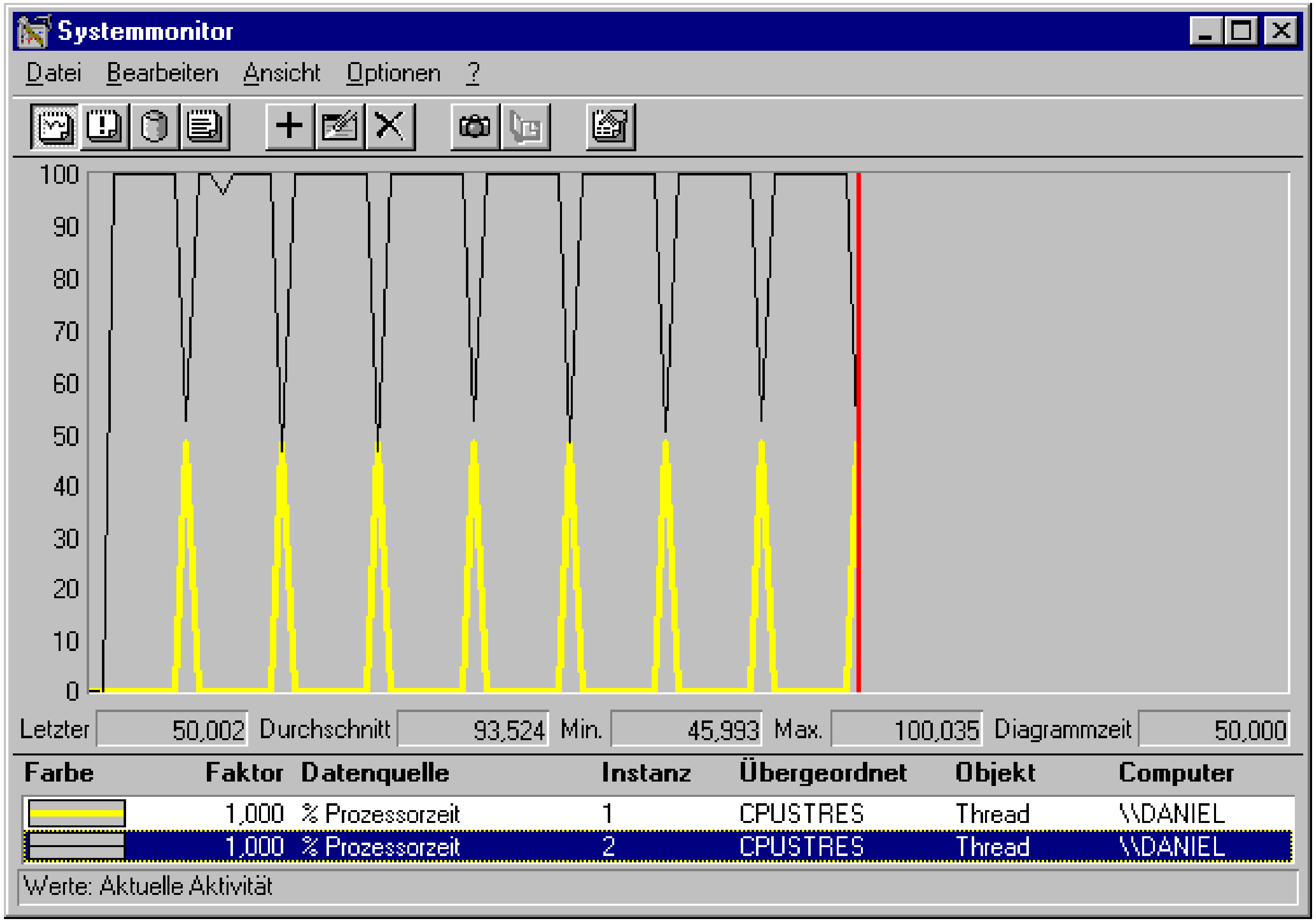
- ***Dynamic Boosts***
  - The system dynamically raises thread priorities in specific situations (not for REALTIME_PRIORITY_CLASS)
    - Disk input or output complete: +1
    - Mouse, keyboard input: +6
    - Semaphore, Event, Mutex: +1
    - Other events (network, pipe, …) +2
    - Event in the foreground application +2
  - Dynamic Boost gets “used up”
    (one level per Quantum)

Priority Change after a Boost



The *Balance-Set Manager*

- About every 3–4 seconds, up to 10 “disadvantaged” threads are raised to priority 15 for two time slices



Letzter 50,002 Durchschnitt 93,524 Min. 45,993 Max. 100,035 Diagrammzeit 50,000

Farbe	Faktor	Datenquelle	Instanz	Übergeordnet	Objekt	Computer
🟨	1,000	% Prozessorzeit	1	CPUSTRES	Thread	\\DANIEL
▬	1,000	% Prozessorzeit	2	CPUSTRES	Thread	\\DANIEL

Werte: Aktuelle Aktivität

Progress guarantee!

Multiprocessor Scheduling

- Goal: “fair” Round-Robin at maximum throughput
- Problem: Cache effects
- Since Windows 8 / Windows Server 2012: “Ready queue” per priority level and **CPU group** (before: per CPU)
  - Groups aligned to SMT-set/multicore package/NUMA information
  - **Ready summary:** 32-bit bitmask to speed up finding the highest-priority non-empty queue
  - Guarantee: Each CPU group runs ≥1 highest-priority thread

Multiprocessor Scheduling

- Threads can be restricted with **CPU affinity**
  (mapping CPUs ⇔ thread)
  - **hard_affinity**: Fixed mapping
    → via `SetThreadAffinity()`
  - **ideal_processor**: “Ideal” mapping (NUMA: also **ideal_node**)
    → assigned at creation time (“random”)
    → modifiable via `SetThreadIdealProcessor()`
  - **soft_affinity**: Previous CPU the thread ran on
    → internally managed by the scheduler
  - **last_run**: Point in time the thread ran last
    → internally managed by the scheduler

Multiprocessor Scheduling

- Algorithm: CPU *n* calls `KiSelectNextThread()`
  - Use *ready summary* to pick highest-prioritized non-empty ready list of the CPU group this CPU belongs to
  - Pick head of this ready list
  - If `ReadyQueue` completely empty, activate *Idle Loop*
  - In *Idle-Loop*: Search `ReadyQueue` of other CPU groups
    (taking NUMA-node topology and other factors into account)
- more features:
  - Heterogeneous scheduling
    (Arm big.LITTLE, Intel Performance Hybrid Architecture)
  - Dynamic Fair Share Scheduling (DFSS)

Conclusion Windows

- *"interactive, probabilistic, online, preemptive, multi-processor CPU scheduling"*
- Priority model allows fine-grained CPU-time allocation
  - Dynamic modifications
  - User-mode threads with high real-time priorities take precedence over all system threads!
  - Threads in the Executive are generally preemptible
- Continuous SMP/NUMA improvements since Windows 2003
- **Heuristics** to accommodate interactive users

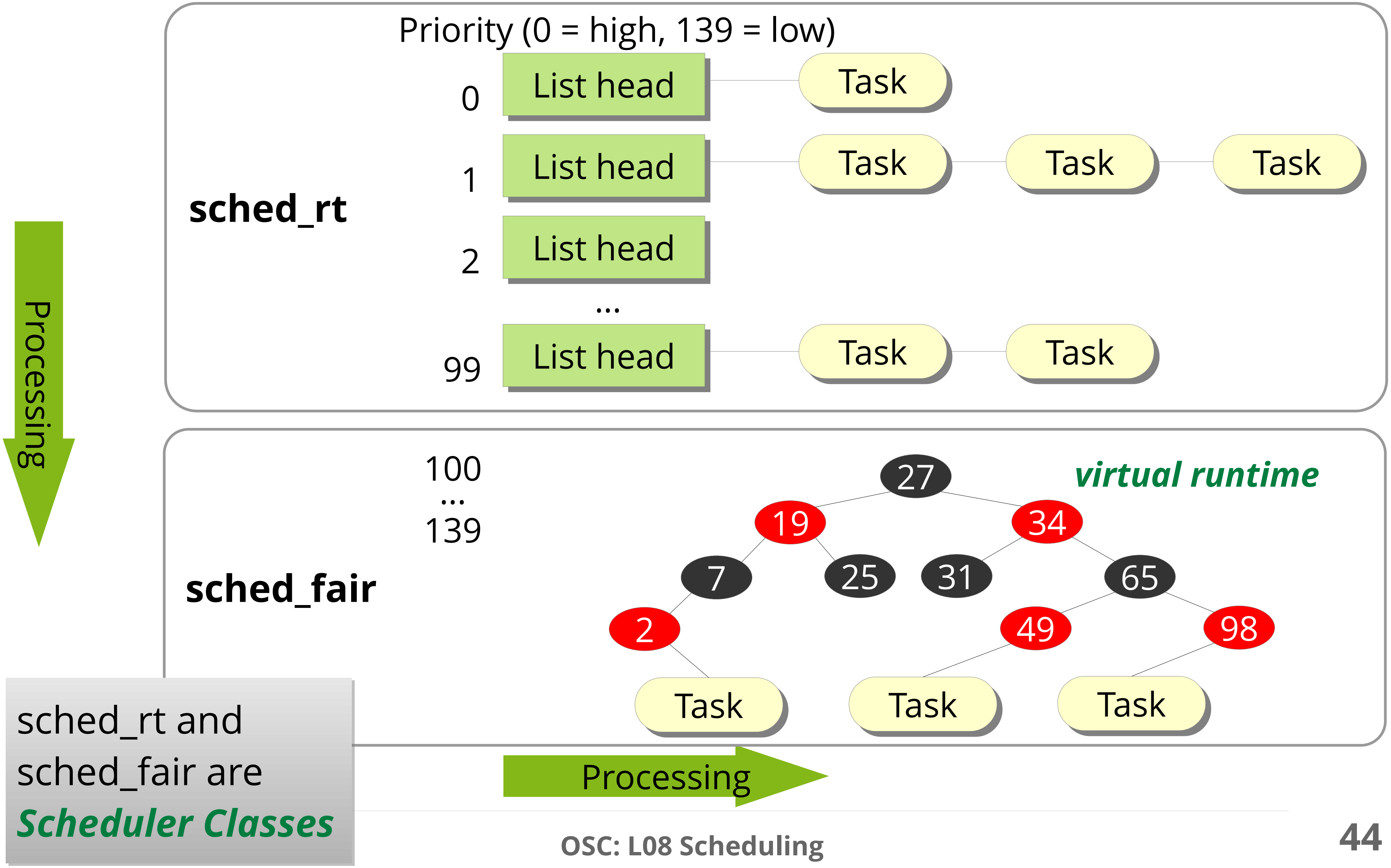
Agenda

- Kernel-Level Threads
  - Motivation
  - Cooperative Thread Switch
  - Preemptive Thread Switch
- Scheduling
  - Basic Terms and Classification
  - in Windows (8–11)
  - **in Linux**
- Summary

Linux *Tasks* ...

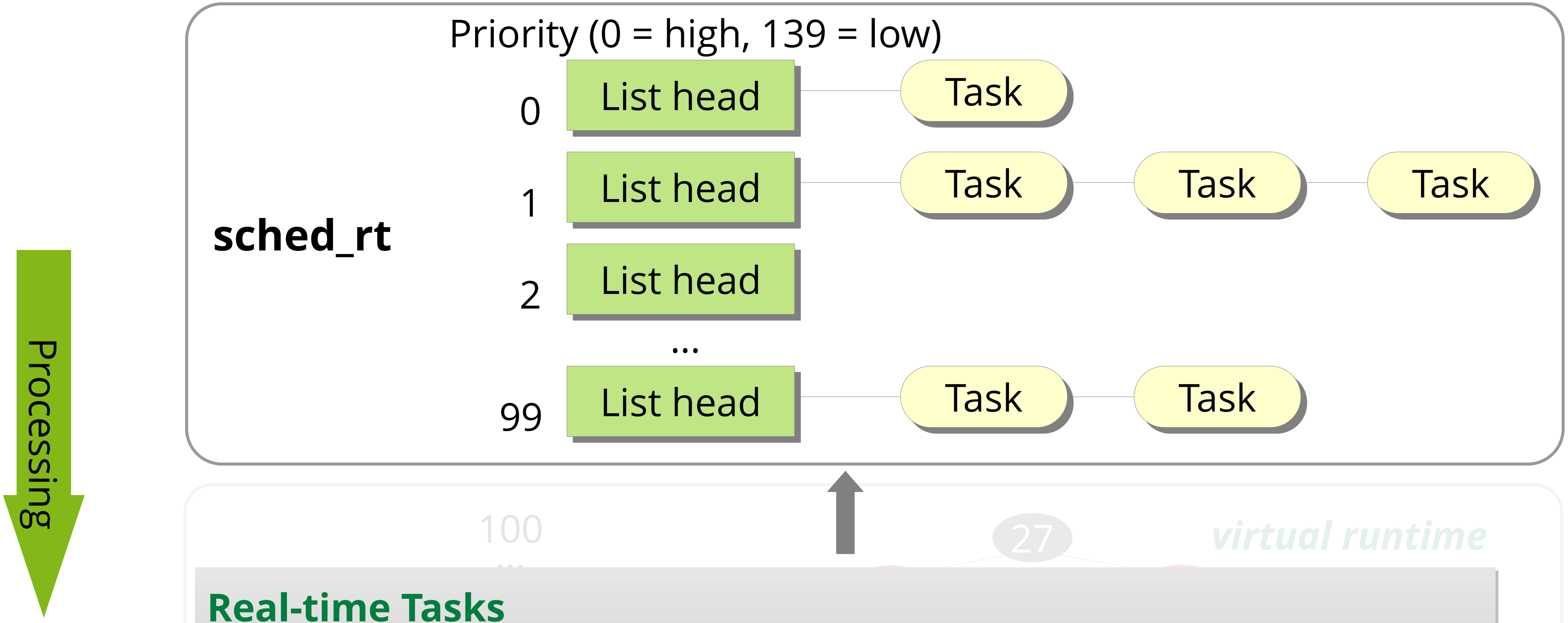
- are the **Linux-Kernel** abstraction for ...
  - **UNIX processes:** one thread in one address space
  - **Linux Threads:** special process that shares its virtual address space with at least one other thread
- are the activities considered by the scheduler
  - Up to Linux 2.6.23 (introduction of CFS, the *Completely Fair Scheduler*) a program with many threads received more computation time than a single-threaded process
    - similarly a program with one process and many child processes

Linux’ Modular Scheduler



sched_rt and sched_fair are ***Scheduler Classes***

Linux’ Modular Scheduler



Priority (0 = high, 139 = low)

sched_rt

0 List head — Task

1 List head — Task — Task — Task

2 List head

…

99 List head — Task — Task

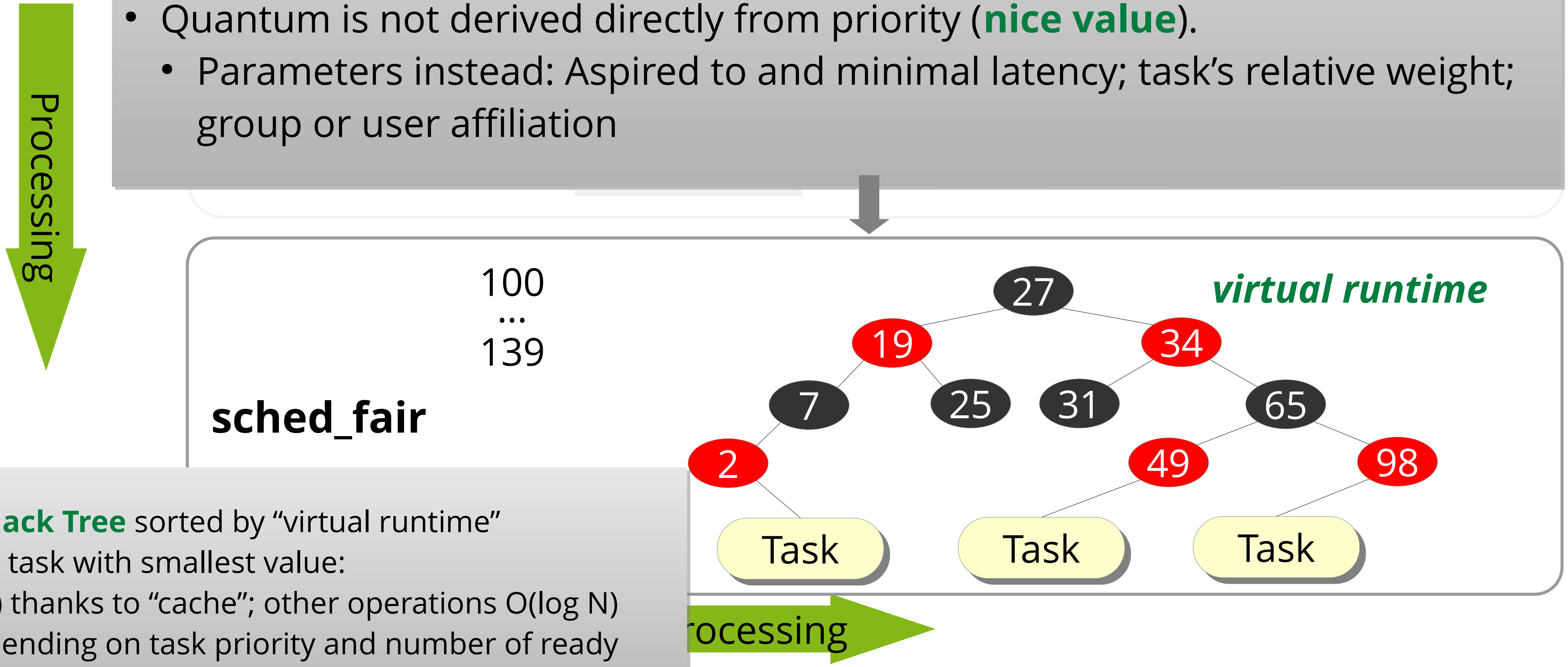
Real-time Tasks

- SCHED_FIFO never preempted
- SCHED_RR preempted when fixed time slice expires
- Real-time tasks preempt any other regular task.
- Due to the simple strategy, the behavior in a real-time environment can be very well predicted.

Linux’ Modular Scheduler

Regular tasks: **Completely Fair Scheduler (CFS)**

- Geared to an idealized “multitasking processor”
  - Infinitesimally tiny time slices
  - Runtime of two same-priority tasks distributed equally
- Quantum is not derived directly from priority (**nice value**).
  - Parameters instead: Aspired to and minimal latency; task’s relative weight; group or user affiliation



Red-black Tree sorted by “virtual runtime”

- Pick task with smallest value:
  O(1) thanks to “cache”; other operations O(log N)
- Depending on task priority and number of ready tasks, virtual runtime passes with different speeds.

Multiprocessor Support

- Multiple READY lists
  - Parallel scheduler execution possible
- Support for CPU affinity
- Takes “warm” caches into account
- CPU load balancing
  - “push” by load-balancer process
    → spin-locking still necessary
  - “pull” when a READY list runs empty

Conclusion Linux

- *“interactive, probabilistic, online, preemptive, multi-processor CPU scheduling“*
- Modular architecture
  - Arbitrary scheduler hierarchy possible
  - Support for soft real-time applications
- CFS focuses on fairness
  - Goal: Fair distribution of CPU-time **shares**
    - Fairness not guaranteed if very many processes are ready
  - Progress guarantee for all processes
  - No arbitrary heuristics
- CFS solves many problems of classic UNIX schedulers
  - CPU-time limits for users or groups
  - Provides semantics for *nice* values (+1 corresponds to CPU share * 1.25)
- Modern multiprocessor support

Agenda

- Kernel-Level Threads
  - Motivation
  - Cooperative Thread Switch
  - Preemptive Thread Switch
- Scheduling
  - Basic Terms and Classification
  - in Windows (8–11)
  - in Linux
- **Summary**

Summary

- Threads are operating-system coroutines
  - OS has a preemption mechanism
- *Scheduling* has profound impact on system performance. It determines …
  - which process wait and which progress
  - which resources are utilized how much
- There exist many variants of schedulers
  - only little differences at mainstream PC/workstation OSs
  - large differences in other application domains