Operating System Concepts

Lecture 13: CPU Scheduling — Part 3

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

Today’s class

- Scheduling algorithms
  - Priority scheduling: SJF and beyond
  - Multilevel queue scheduling
  - Multilevel feedback queue scheduling
  - Proportional-share scheduling (lottery scheduling)
- Multiprocessor scheduling
- Scheduling on real-time systems

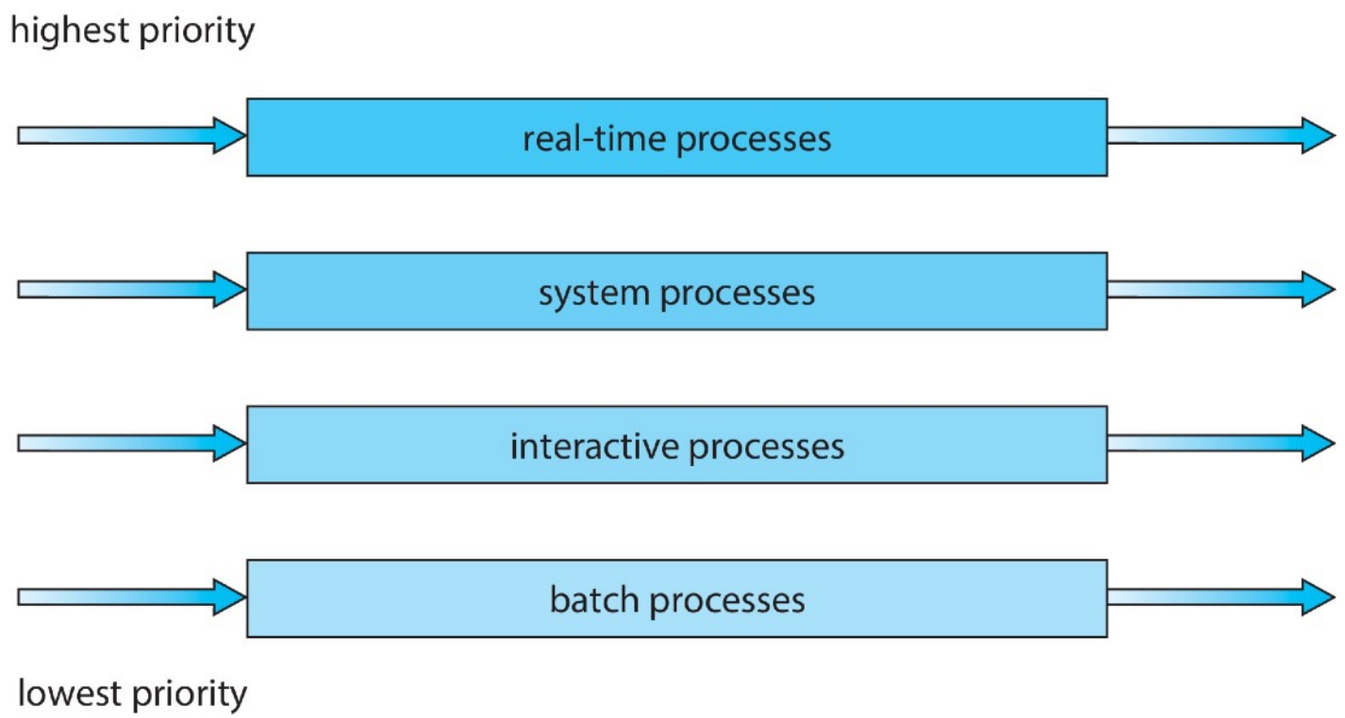
Can we design a policy that minimizes response time, is fair and starvation-free, and has low overhead?

Priority scheduling

- Priority scheduling (preemptive and non-preemptive)
  - sort ready queue according to priority; so high-priority processes will run before low-priority processes and equal priority processes are scheduled in FCFS order
  - SJF is a simple priority scheduling policy with process priority being the inverse of its (predicted) CPU burst
- Low-priority processes may be indefinitely blocked (starvation)
- Two solutions for dealing with **starvation**
  - (a) **aging:** gradually increasing priority of processes waiting for a long time
  - (b) **combine round robin and priority scheduling:** add different queues, each for a distinct priority level, and time-slice among them (e.g., 70% to high priority, 20% to mid priority, 10% to low priority)
- These solutions improve fairness at the cost of increased response time

Multilevel queue scheduling

- Separate queues for each distinct priority
  - a process remains in the same queue for the duration of its runtime
- How to lower the response time?
  - high-priority queues have shorter time slices; low-priority queues have longer time slices

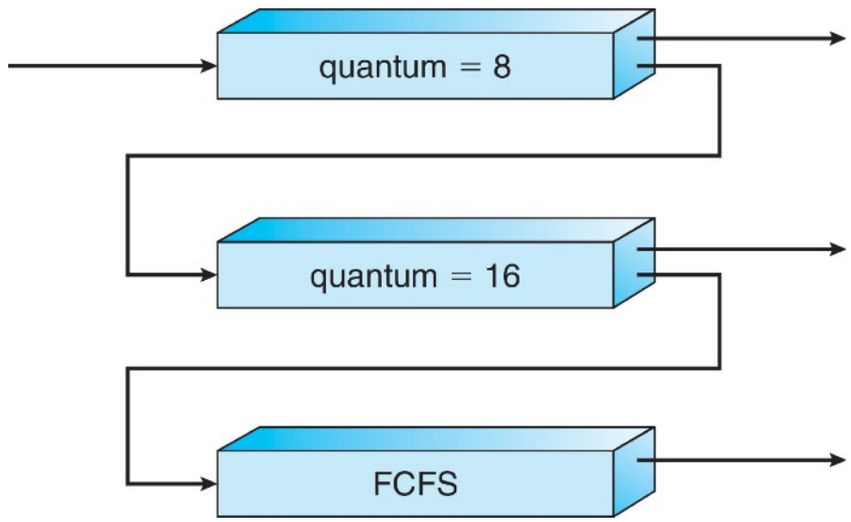


Priority inversion

- In priority scheduling, a task can be prevented from running by a higher priority task… with some exceptions
  - a high-priority task can be blocked waiting on a low priority task, for example, to release a lock
  - so the low-priority task must run for the high-priority task to make progress
- **Solution:** a high-priority task temporarily grants a low-priority task its “highest priority” to run on its behalf and release the lock
  - the high priority task then acquires the lock and runs again

MLFQ scheduling

- Multilevel feedback queue (MLFQ) is a set of **round robin** queues with different **priorities**
  - use Round Robin scheduling for each priority level
  - run the jobs in the highest priority queue first, once those finish, run jobs at the next highest priority queue, etc.
  - jobs can move between queues
- RR time slice increases exponentially as we move to lower priority queues



Using feedback to adjust priorities

- Jobs start in highest priority queue
  - if a job's time slice expires, drop its priority one level
  - if a job's time slice does not expire (i.e. context switch happens due to an I/O request), then increase its priority one level (up to the top priority level)
- CPU-bound jobs drop in priority and I/O-bound jobs stay at a high priority
  - a process that waits too long in a low priority queue may be moved up to avoid starvation (**aging**)
- By manipulating the assignment of tasks to priority queues, a MFQ scheduler can achieve a balance between responsiveness, low overhead, and fairness

Improving fairness

- Since SJF is optimal but unfair, any improvement in fairness by giving long jobs a fraction of CPU cycles when shorter jobs are available will degrade average waiting time
- Possible solutions
  - give each queue a fraction of the CPU time
    - this solution is only fair if there is an even distribution of jobs among queues
  - adjust the priority of jobs if they do not get serviced (UNIX originally did this)
    - this ad hoc solution avoids starvation but average waiting time suffers when the system is overloaded because all the jobs end up with a high priority

Lottery scheduling

- Give every process a number of lottery tickets; in each time slice, randomly pick a winning ticket and run the winning process
- On average, the share of CPU cycles is proportional to the number of tickets given to each job
  - give a job 10% of tickets is equivalent to giving it 10% of CPU cycles on average
- How to assign tickets? to avoid starvation, every job gets **at least one ticket**
  1. give most tickets to short running processes, and fewer to long running processes (approximating SJF scheduling)
  2. give most tickets to high-priority processes, and fewer to low-priority processes (approximating priority scheduling)
- Performance degrades gracefully as load changes
  - adding or deleting a process affects all processes proportionately, independent of the number of tickets a process has
- To address priority inversion: donate tickets to the process you are waiting for

Lottery scheduling example

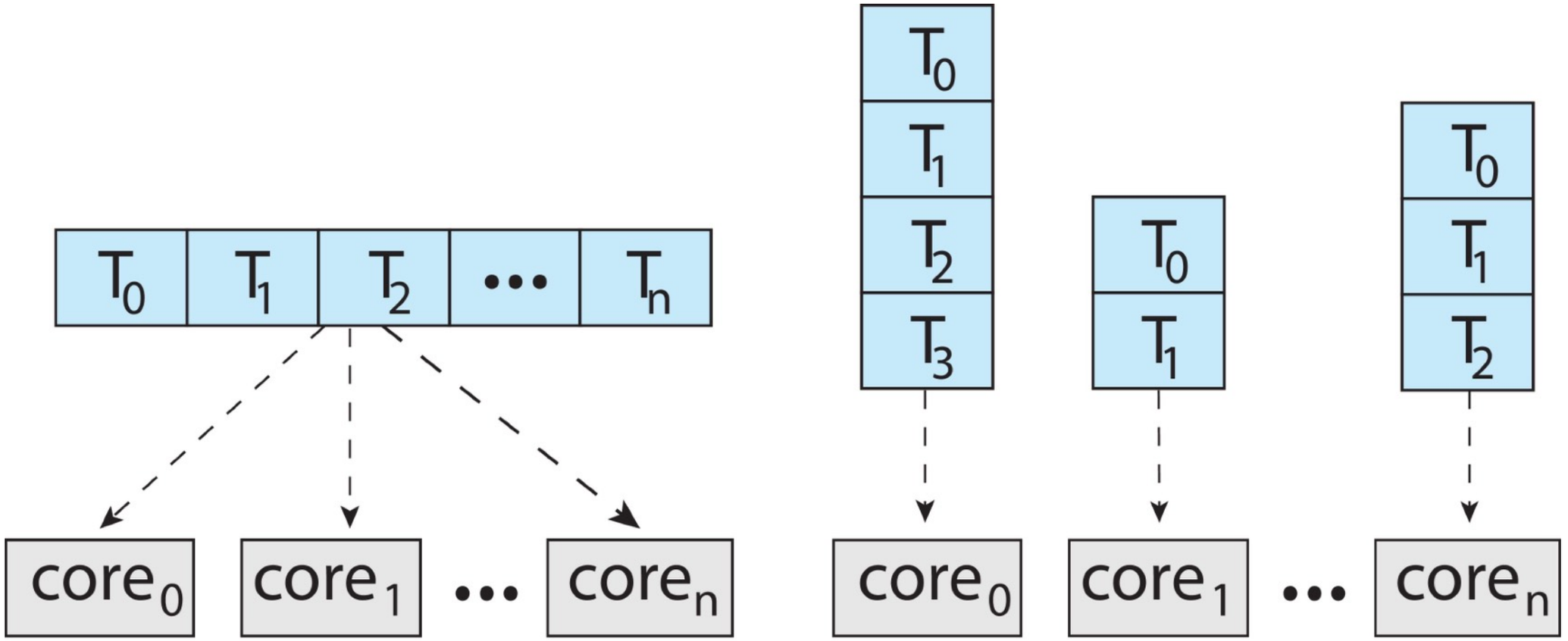
short jobs receive **8 tickets**; long jobs receive **2 tickets**

	#tickets	#short jobs	#long jobs	%CPU each short job gets	%CPU each long job gets	
	10	1	1	80	20	
long job added to queue	12	1	2	66.6	16.6	-16.6%
short job added to queue	20	2	2	40	10	-40%
long job removed from queue	18	2	1	44.4	11.1	+11.1%
short job removed from queue	10	1	1	80	20	+80%

Scheduling on multi-processor systems

- Asymmetric multiprocessing
  - a single processor is responsible for scheduling, I/O processing, and other system activities; other processors run user tasks/threads only
  - advantage: simplicity as only one processor needs to access kernel data structures
  - disadvantage: that processor may become a bottleneck
- Symmetric multiprocessing (SMP)
  - each processor is self scheduling

Ready queues and scheduling in SMP



common ready queue
(a)

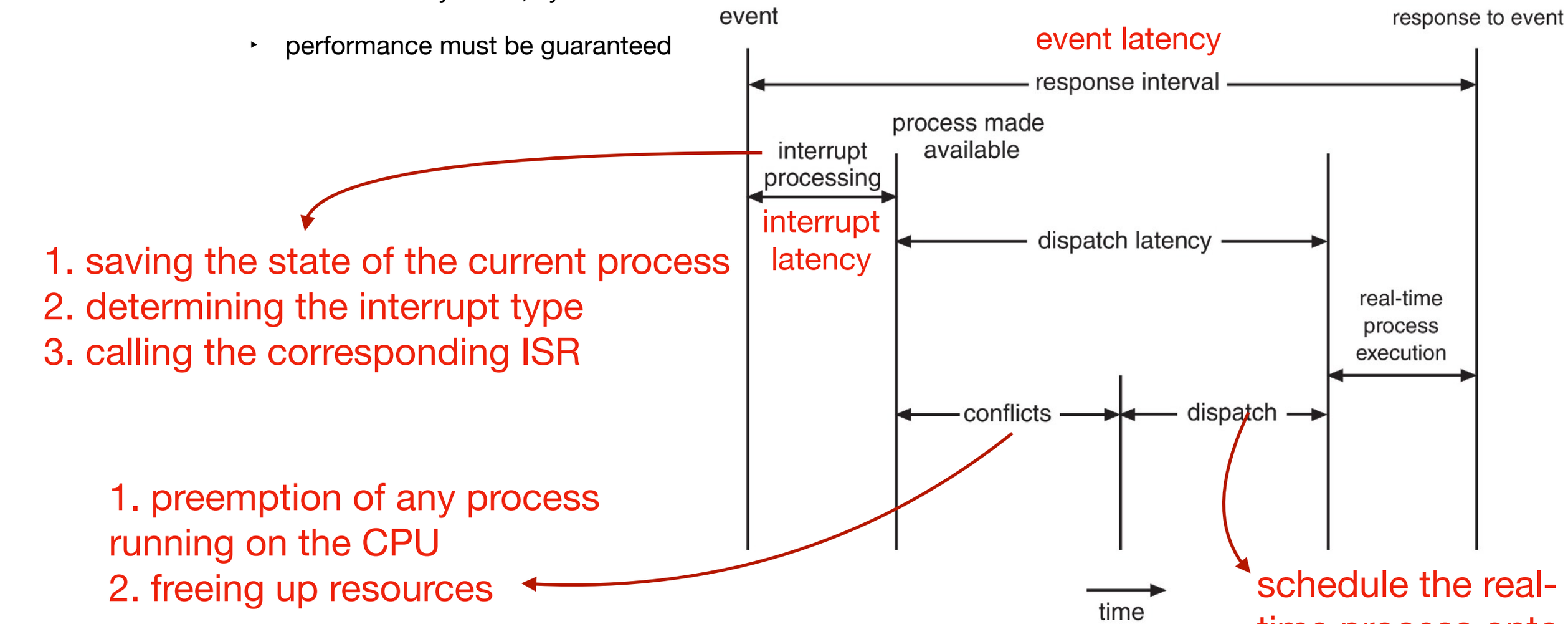
locking the ready queue is necessary
to avoid race conditions
(lock would be highly contended)

per-core run queues
(b)

most common approach in SMP
load balancing may be necessary

Real-time systems

- A real-time system is a system in which task **completion by deadline** is crucial to its performance
  - in soft real-time systems, performance degrades if deadlines cannot be met (event latency > time budget)
    - the scheduler is best effort (no guarantee)
  - in hard real-time systems, system **fails** if deadlines cannot be met
    - performance must be guaranteed



Real-time scheduling

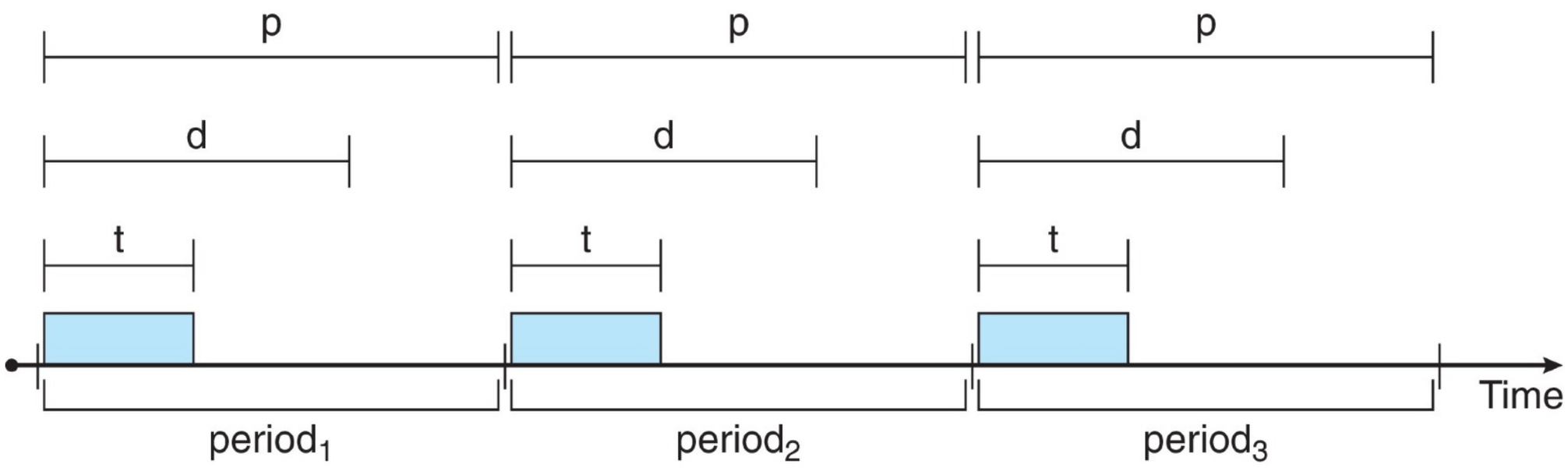
- A preemptive priority-based scheduler is often used in soft real-time systems
  - assigns real-time processes the highest priority
    - e.g., Windows has 32 priority levels, levels 16 to 32 are reserved for real-time processes
- Admission control is necessary in hard real-time systems
  - each task has to announce its deadline to the scheduler when becomes runnable
  - the scheduler admits the task if it can guarantee that it will finish execution by the deadline
  - the scheduler rejects the task if it cannot guarantee that

Assumption: periodic tasks

- A periodic task
  - requires the CPU at constant intervals (period = *p*; rate = *1/p*)
  - has a fixed deadline *d*, by which it must be serviced
  - once acquired the CPU, has a fixed processing time *t*

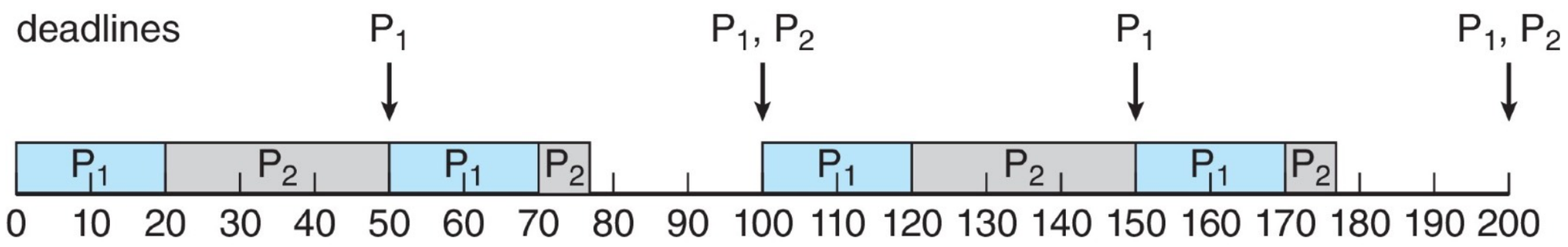
$$0 \leq t \leq d \leq p$$

  - $\frac{t}{p}$ is the percentage of CPU it needs



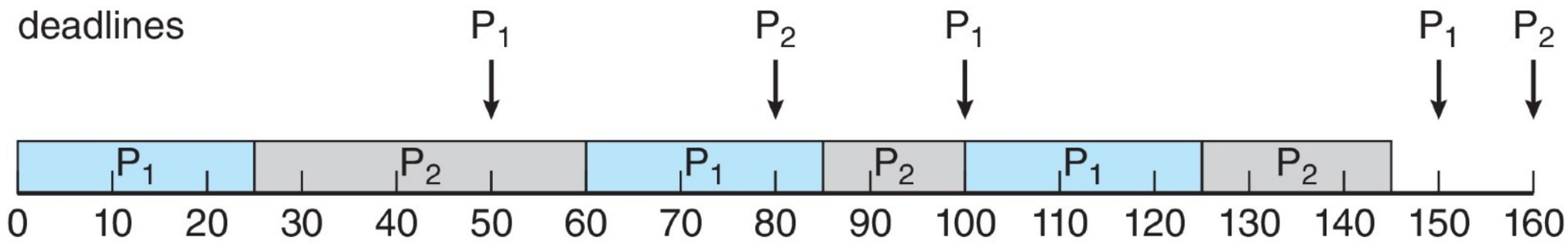
Rate-Monotonic scheduling

- Priority assigned to each task based on the inverse of its period
  - shorter periods = higher priority; longer periods = lower priority
  - simple implementation without requiring knowledge of deadline
- Example: consider two tasks P1 and P2
  - P1: p=50, t=20, d=50
  - P2: p=100, t=35, d=100
  - P1 is assigned a higher priority than P2
- Admission control rule (assuming deadline is equal to period): reject a submitted task if $\sum_i \frac{t_i}{p_i} > 1$

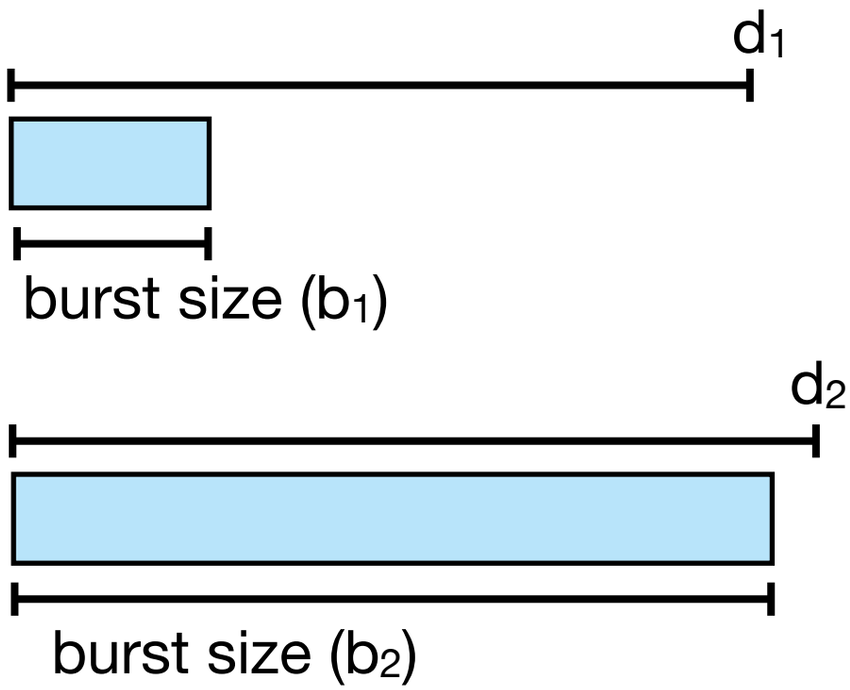


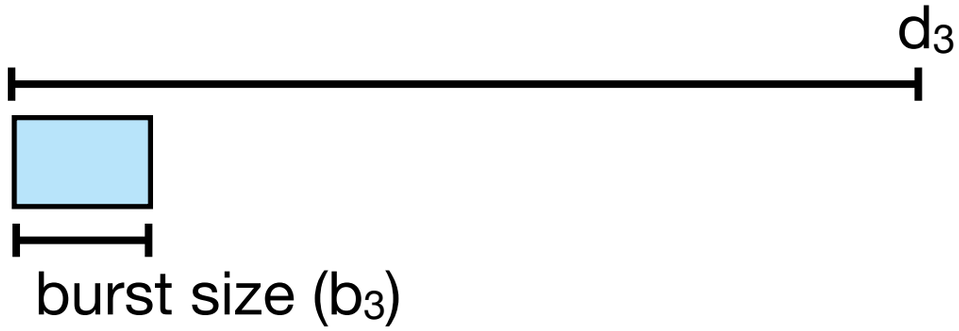
Earliest-Deadline-First (EDF) scheduling

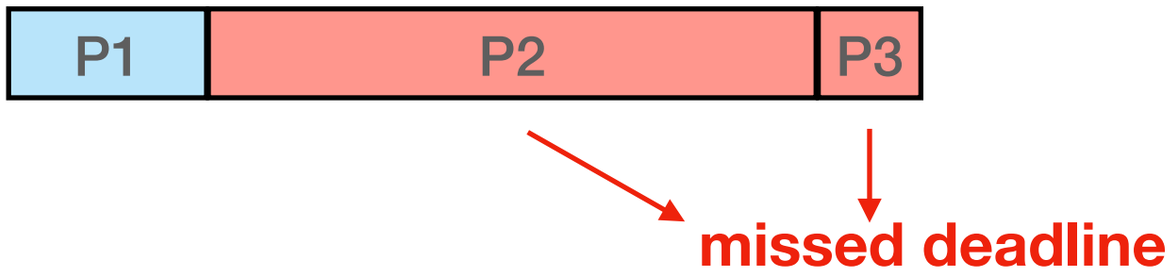
- Priorities are assigned according to deadlines
  - the earlier the deadline, the higher the priority
  - it's a preemptive scheduling policy
  - does not require tasks to be periodic
  - does not require knowledge of the CPU burst so it can be easily implemented in the real world
- EDF is theoretically optimal with respect to CPU utilization if there exists a feasible schedule



EDF in overloaded systems

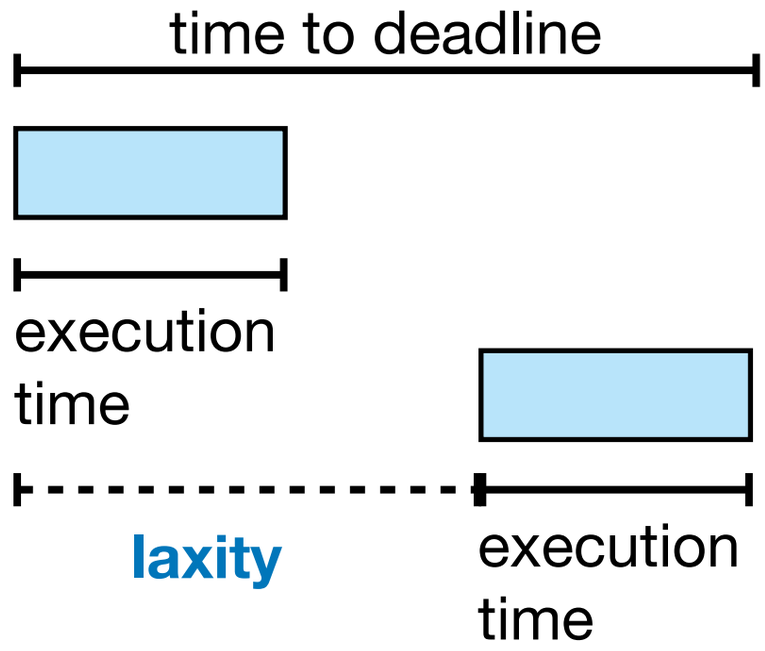






Least-Laxity-First (LLF) scheduling

- Laxity is **slack time** defined as
  time until deadline - remaining execution requirement =
  deadline - current time - remaining execution requirement



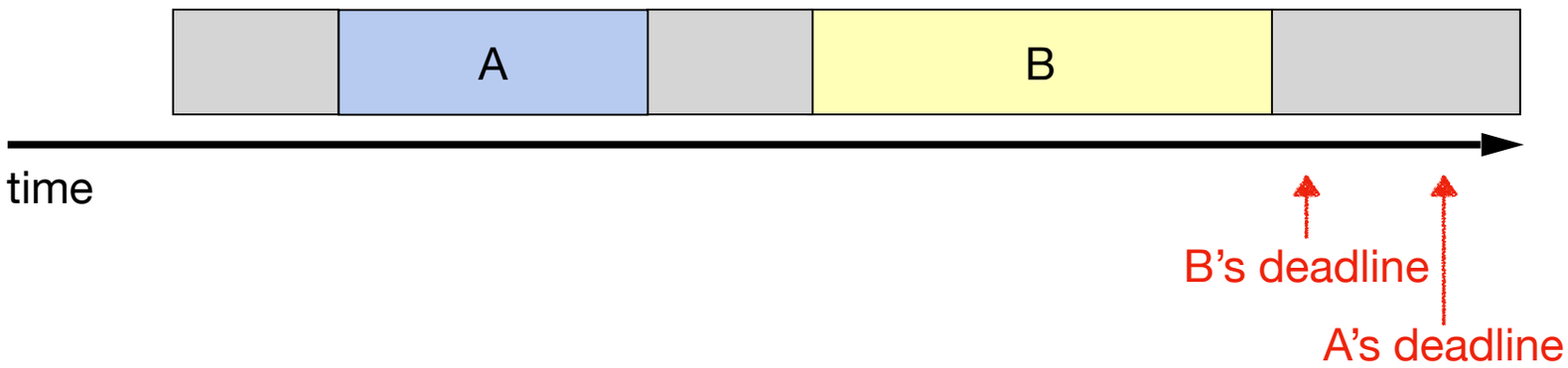
- Assign priority to tasks according to their laxity
  - the lower the laxity, the higher the priority
  - it's a preemptive scheduling policy
- If laxity of a process becomes negative it is impossible to meet the deadline even if we give all CPU cycles to that process from this point onwards
- LLF is a more complex scheduling policy compared to EDF
  - priorities must be updated dynamically
  - but it can detect missed deadlines early and do not execute those jobs at all

Homework

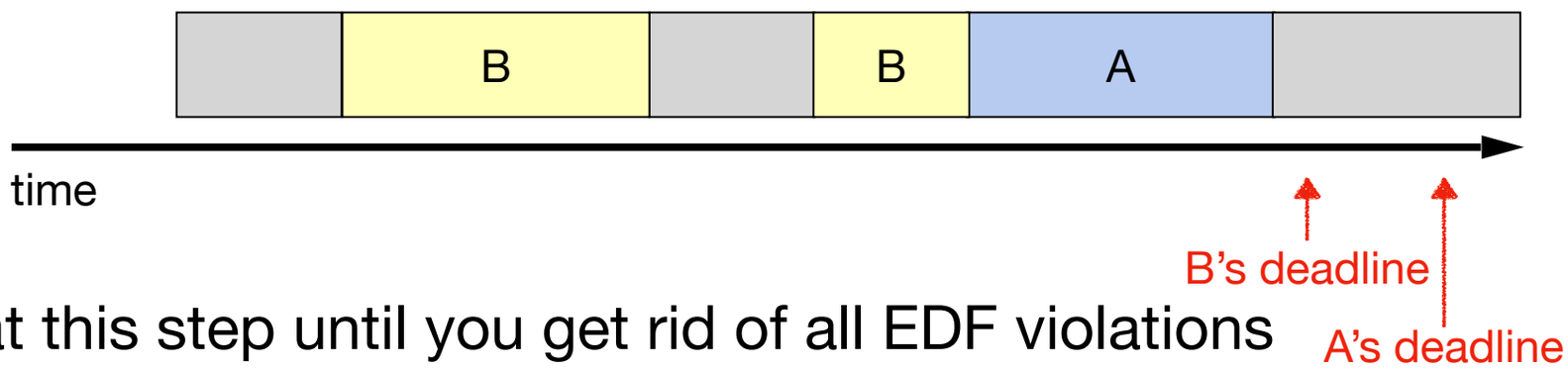
- Prove that when preemption is allowed, the EDF algorithm can produce a feasible schedule of a set S of independent tasks with arbitrary release times and deadlines on a processor if and only if S has a feasible schedule (i.e., a schedule that meets all deadlines)

Proof sketch

- We show that any feasible schedule of *S* can be transformed to an EDF schedule
- Suppose the following is a feasible schedule of *S*, but A and B are scheduled in non-EDF order



- This can be transformed into another feasible schedule in EDF order



- Repeat this step until you get rid of all EDF violations
- Shift the schedule ahead if there is a gap between two tasks