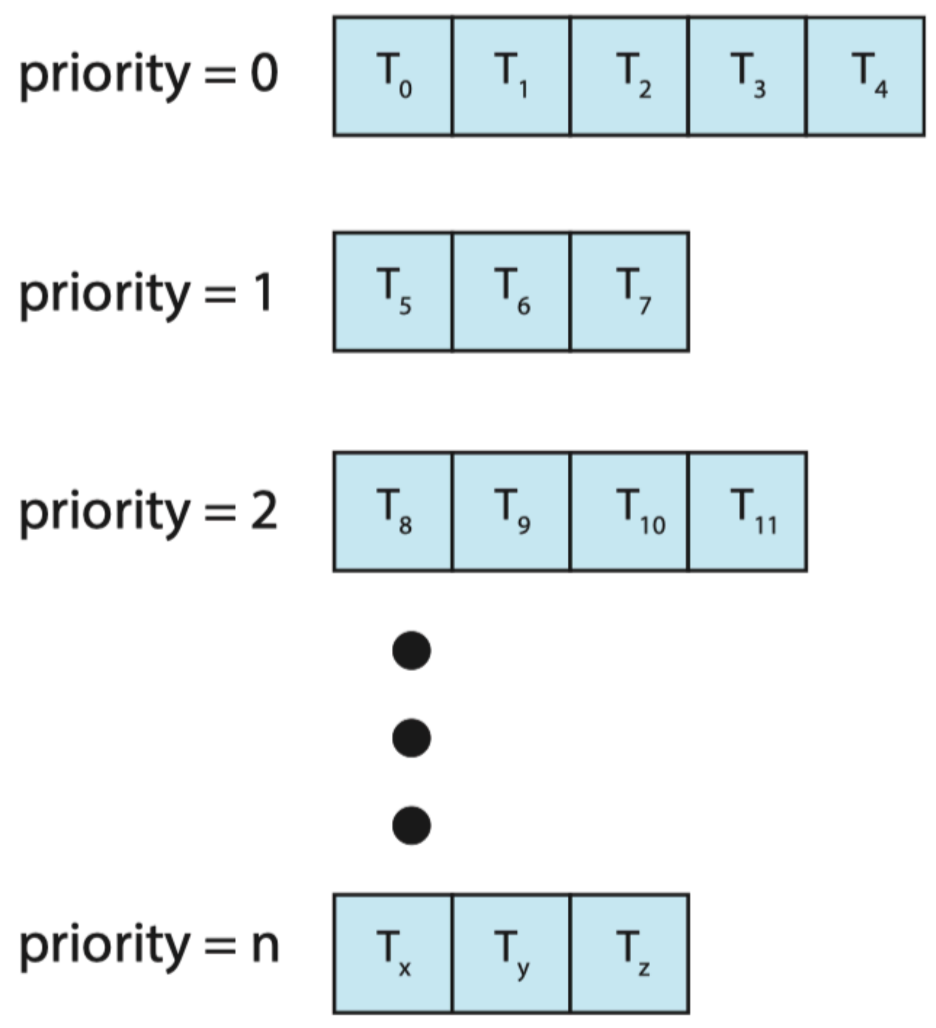
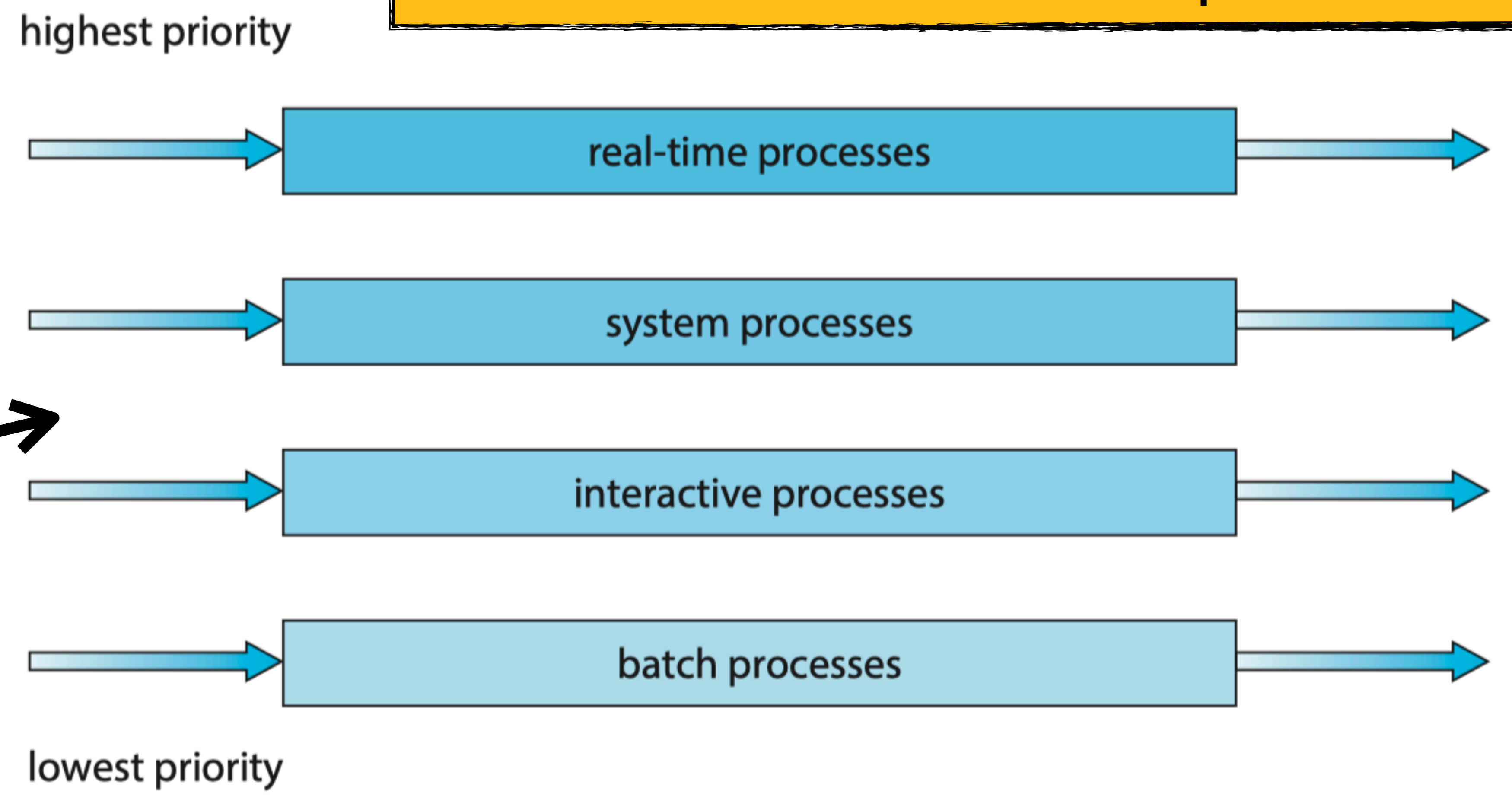
More Scheduling

Queue Size a Problem



- Round Robin: Very simple idea, but the problem is a job needs to wait at most $(n-1) \times q$ times units before running again
- Can we reduce the number of jobs in the queue?
- Not put all the jobs in a single queue - classify the jobs into classes of equal priority
- Run RR within each class - one queue for a class
- Priority across the classes

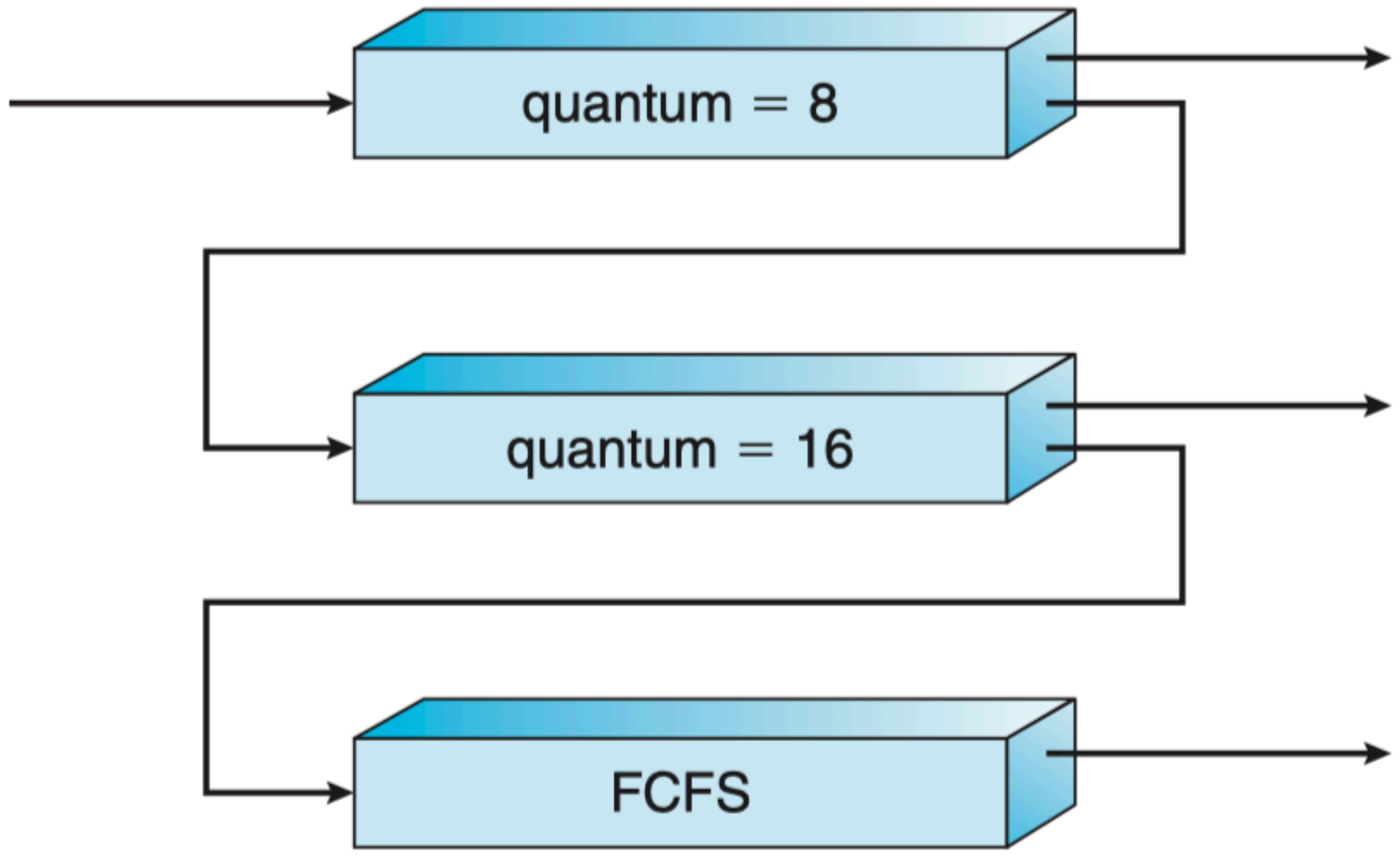
System processes run only when there are no real-time processes



Multilevel Queue Scheduling

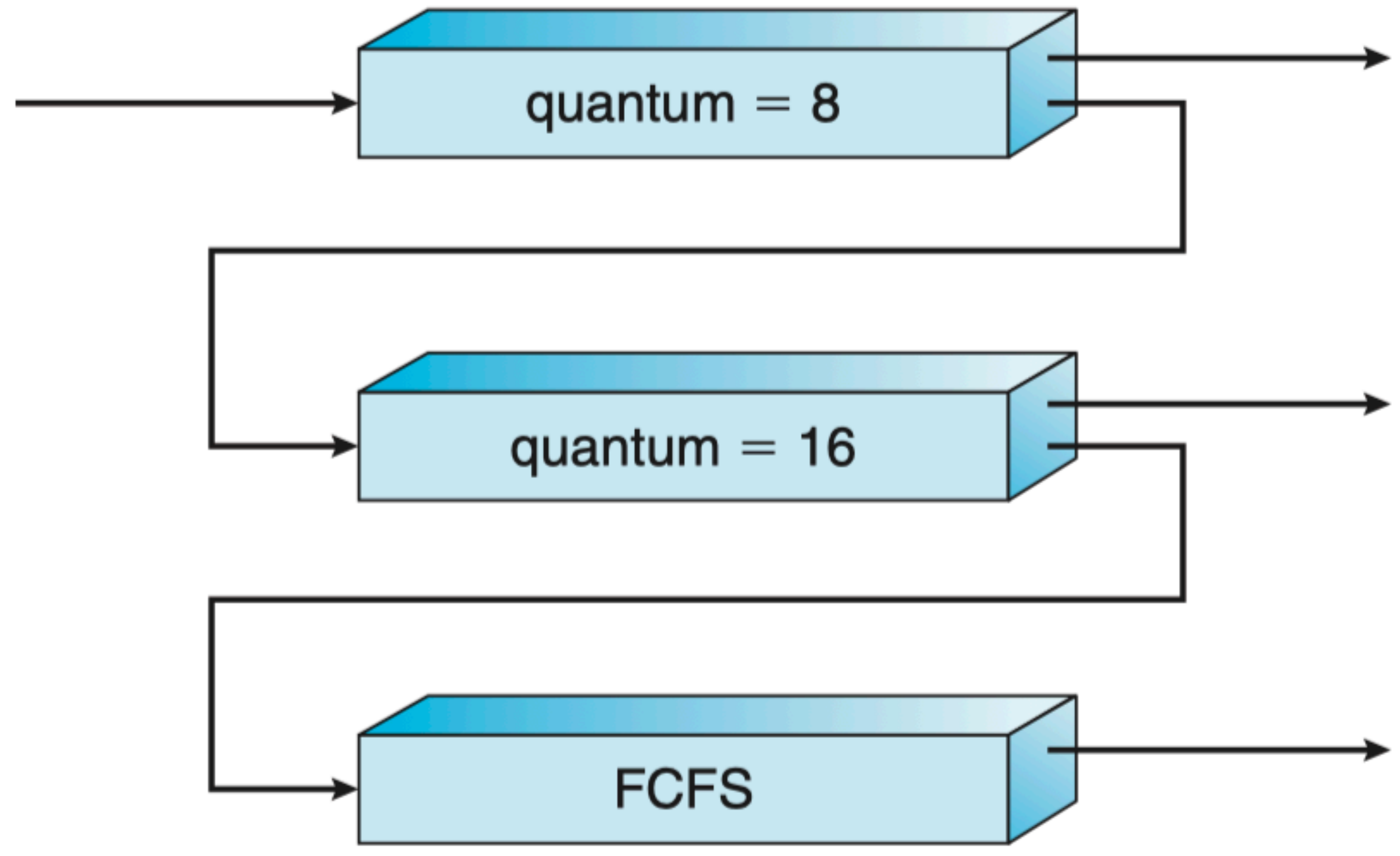
Multilevel Feedback Queue Scheduling

- Multilevel queue scheduling - jobs are assigned statically to a class
- Problem: we don't have much info about a job to assign it properly
- Jobs can change nature dynamically - for example background jobs can become foreground and vice-versa
- Multilevel feedback queue - separate the jobs into queues based on their CPU usage patterns - burst sizes



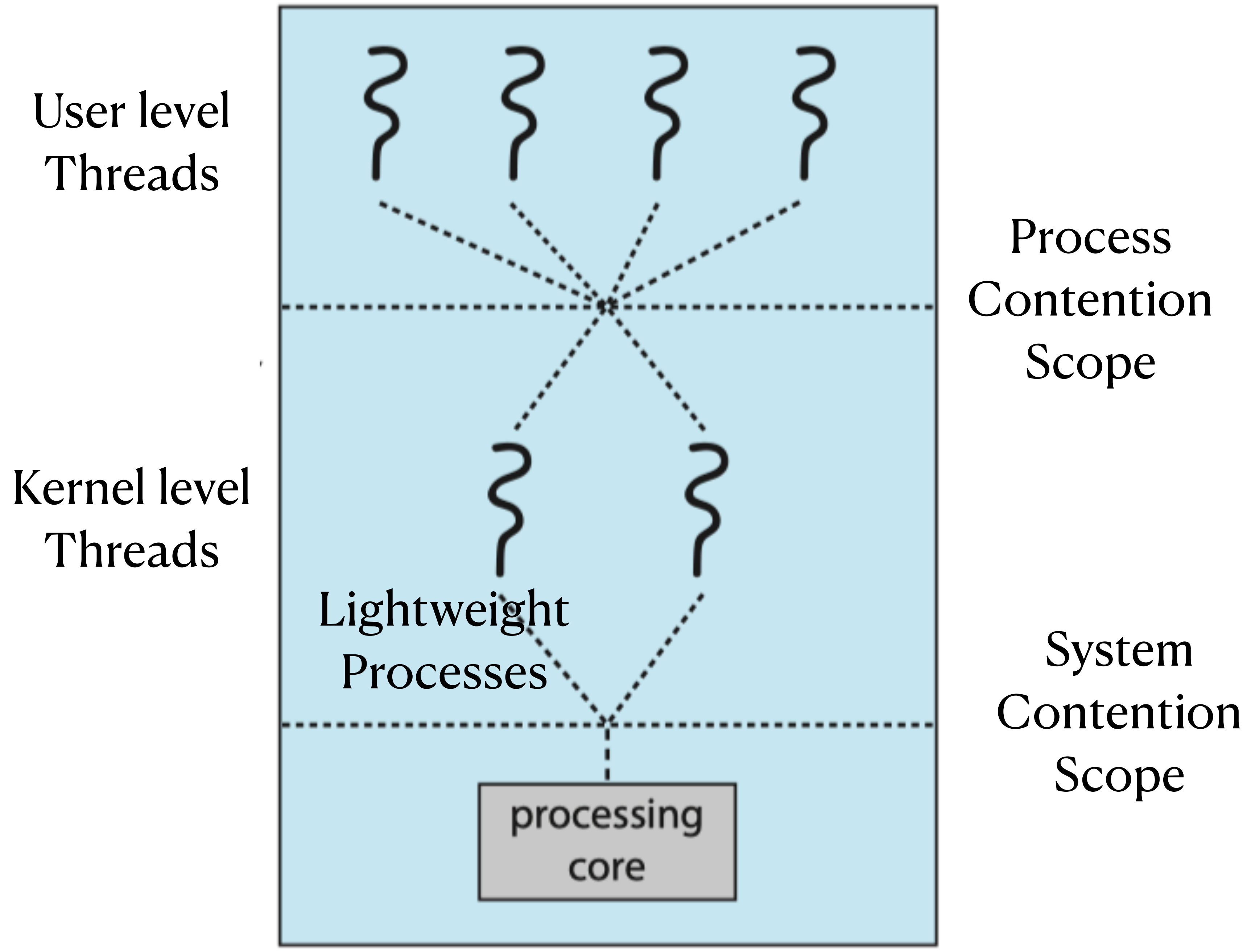
Multi-Level Feedback Scheduling

- Multilevel feedback queue scheduler is defined by the following parameters:
    - Number of queues
    - Scheduling algorithm for each queue
    - Method to upgrade a process to a higher-priority queue
    - Method to downgrade a process to a lower priority queue
    - Method uses to determine which a queue a process will enter when it needs service
- This is a very general CPU scheduling algorithm - each queue can use a different sub scheduler



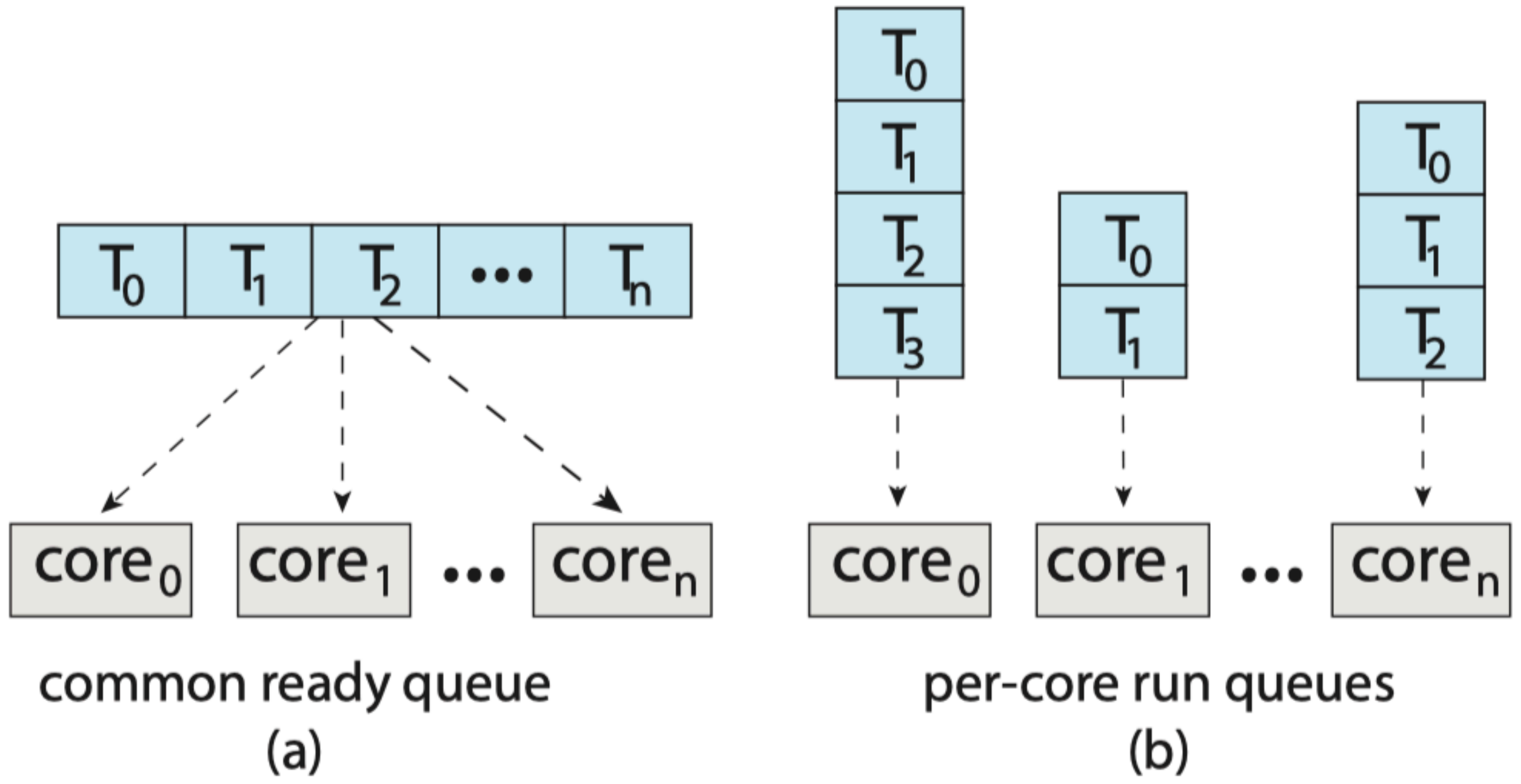
Thread Scheduling

- Threads are of two types: user-level and kernel-level
- Most modern OS schedulers manage kernel-level threads as the basic scheduling unit of CPU time
- User level threads are managed by a thread library - kernel does not know about tit

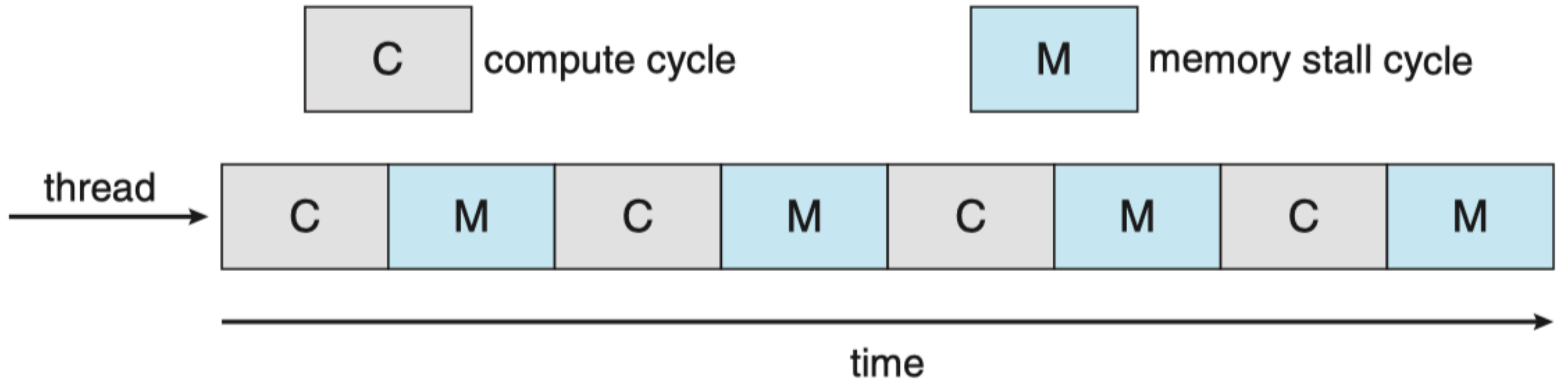


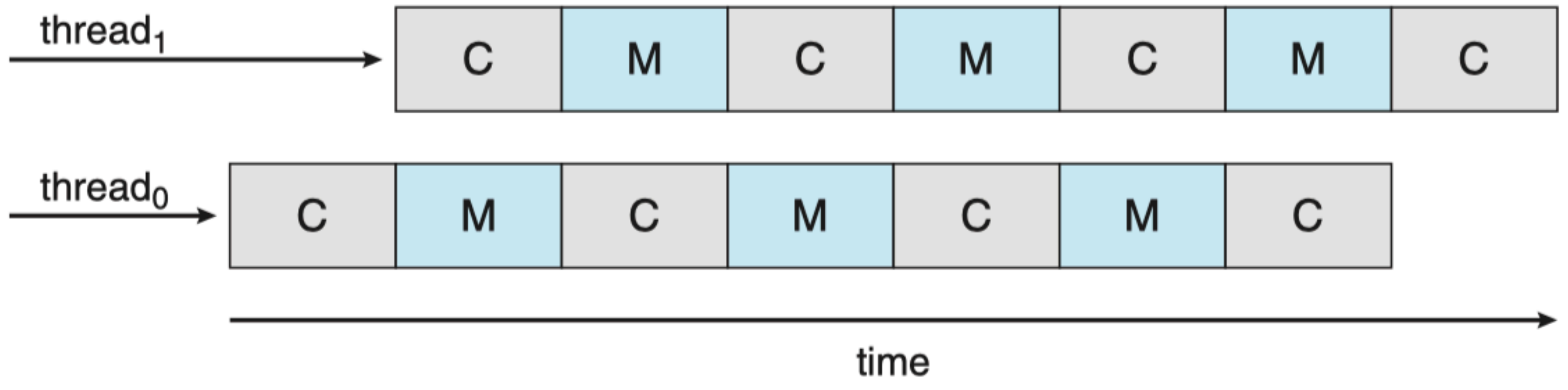
Multi-Processor Scheduling

- With multi-processor scheduling, one important problem is how to organize the ready queue
- Common ready queue - need locking of the queue to avoid race conditions - threads not duplicated or lost
- Lock access can create performance bottleneck
- Private ready queue avoid locking - workload distribution is problem with private queues
  - How to find a lightly loaded queue?
  - How to efficiently transfer work from one queue to another to improve performance?
  - How to deal with workloads with different sizes?



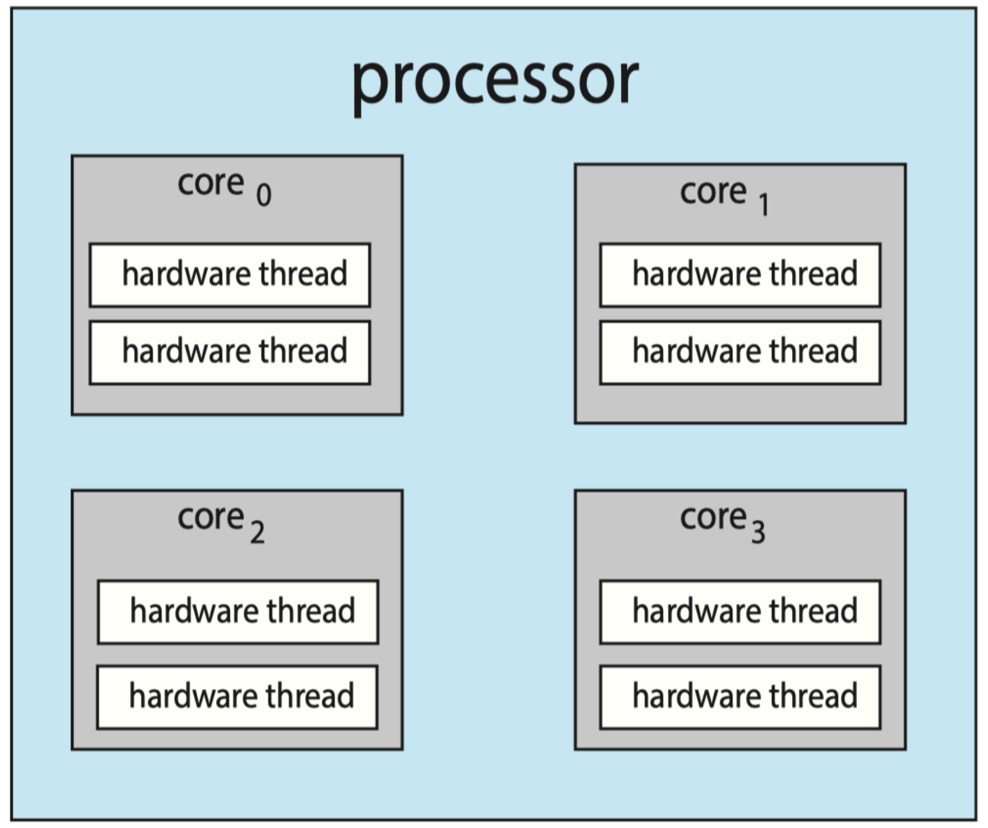
Hardware Multi-Threading

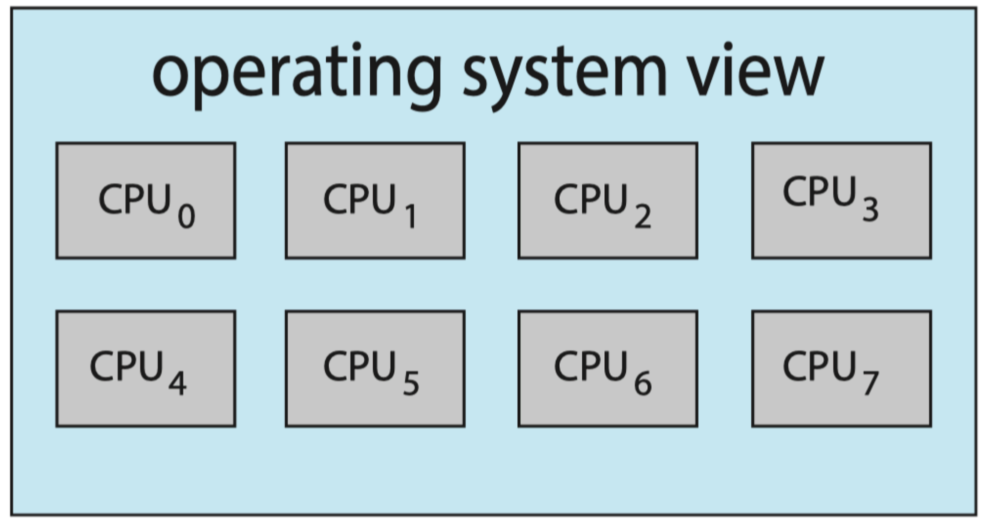




- CPUs are much faster than memory
- Assembly programs have “load” and “store” instructions - IO to memory
- These instructions cause memory stalls - execution has to wait until data comes in - very similar to IO wait in programs!
- Idea is to have two hardware threads - when one hardware thread stalls we are going to switch to the other hardware thread

Chip Multithreading



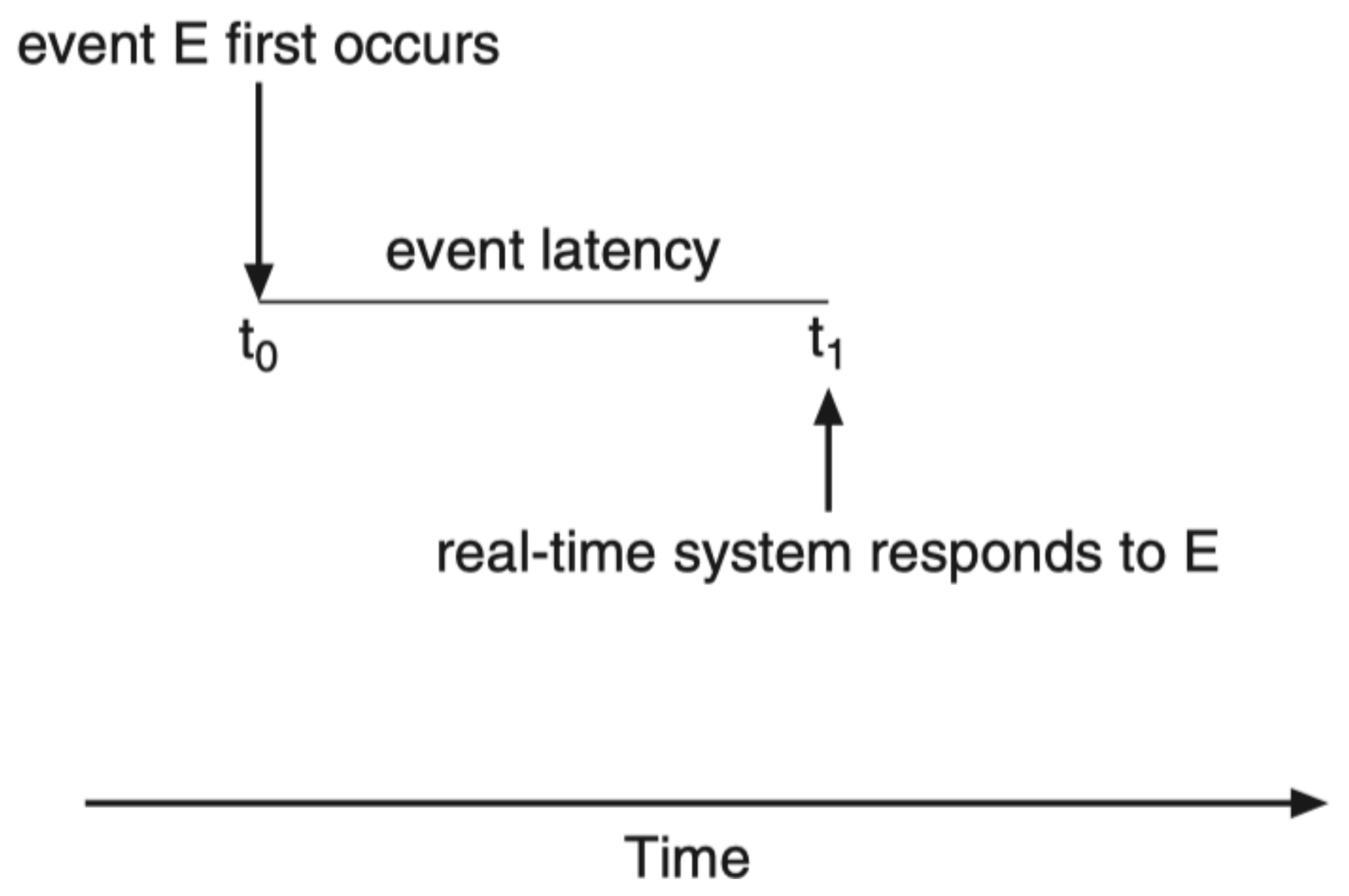


Real-Time Scheduling

- Real-time scheduling can be of two types
  - Soft real-time scheduling - suitable for multimedia computing or other non mission critical applications. We give priority processing for soft real-time processes
  - Maximize deadline compliance in a soft real-time scheduler
  - Hard real-time scheduling - tasks need guarantees on deadline compliance - no deadline must be missed

Minimizing Latency

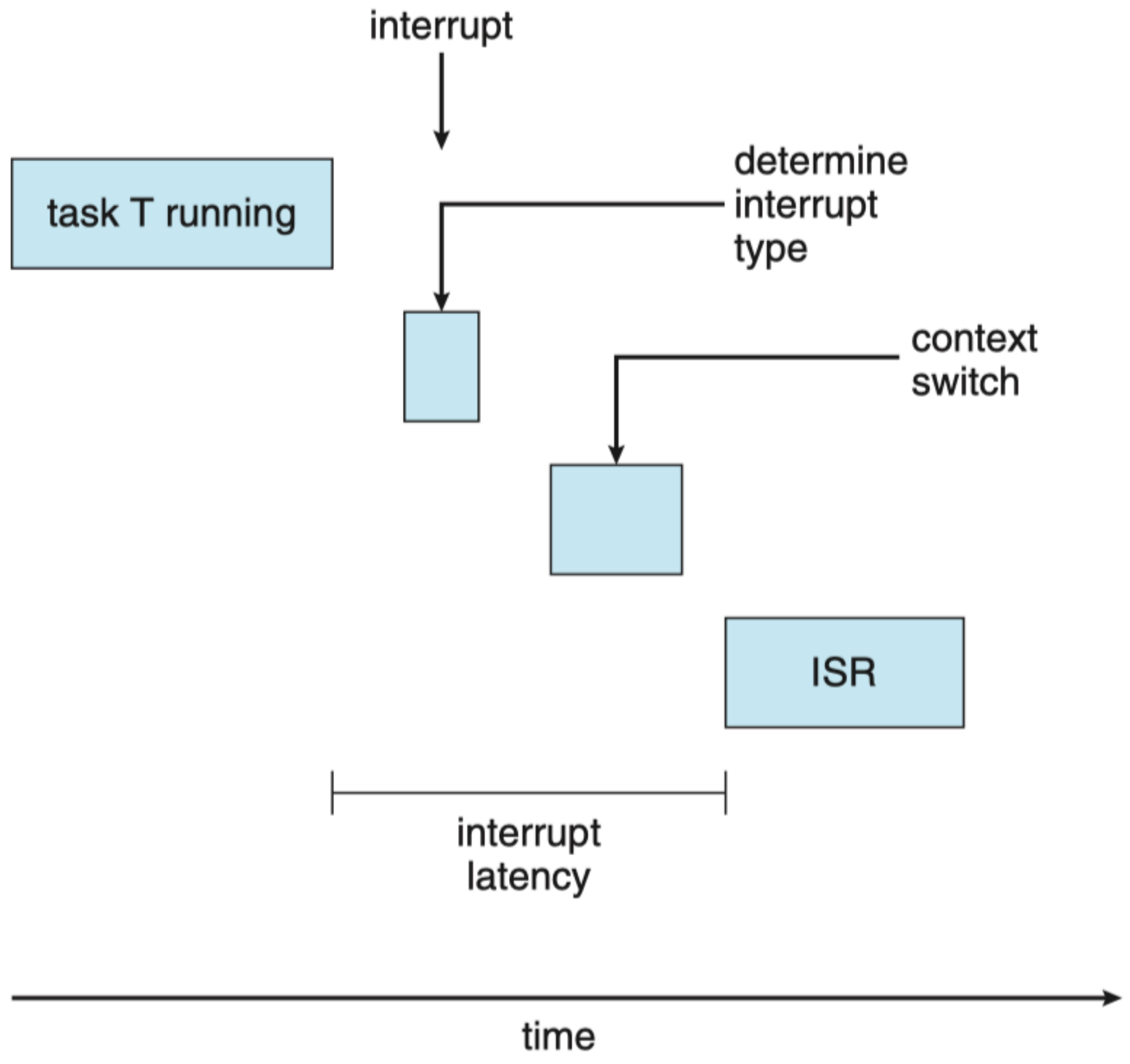
- Minimizing event latency (similar to response time) is very important in real-time systems
- Time elapsed from the event occurrence to event processing
- Different systems would have different event latency requirements:
    - ABS has 3-5 ms event latency requirements
    - Controller for airliner radar can be in seconds



- Two types of latencies affect performance of real-time systems
    - Interrupt latency
    - Dispatch latency

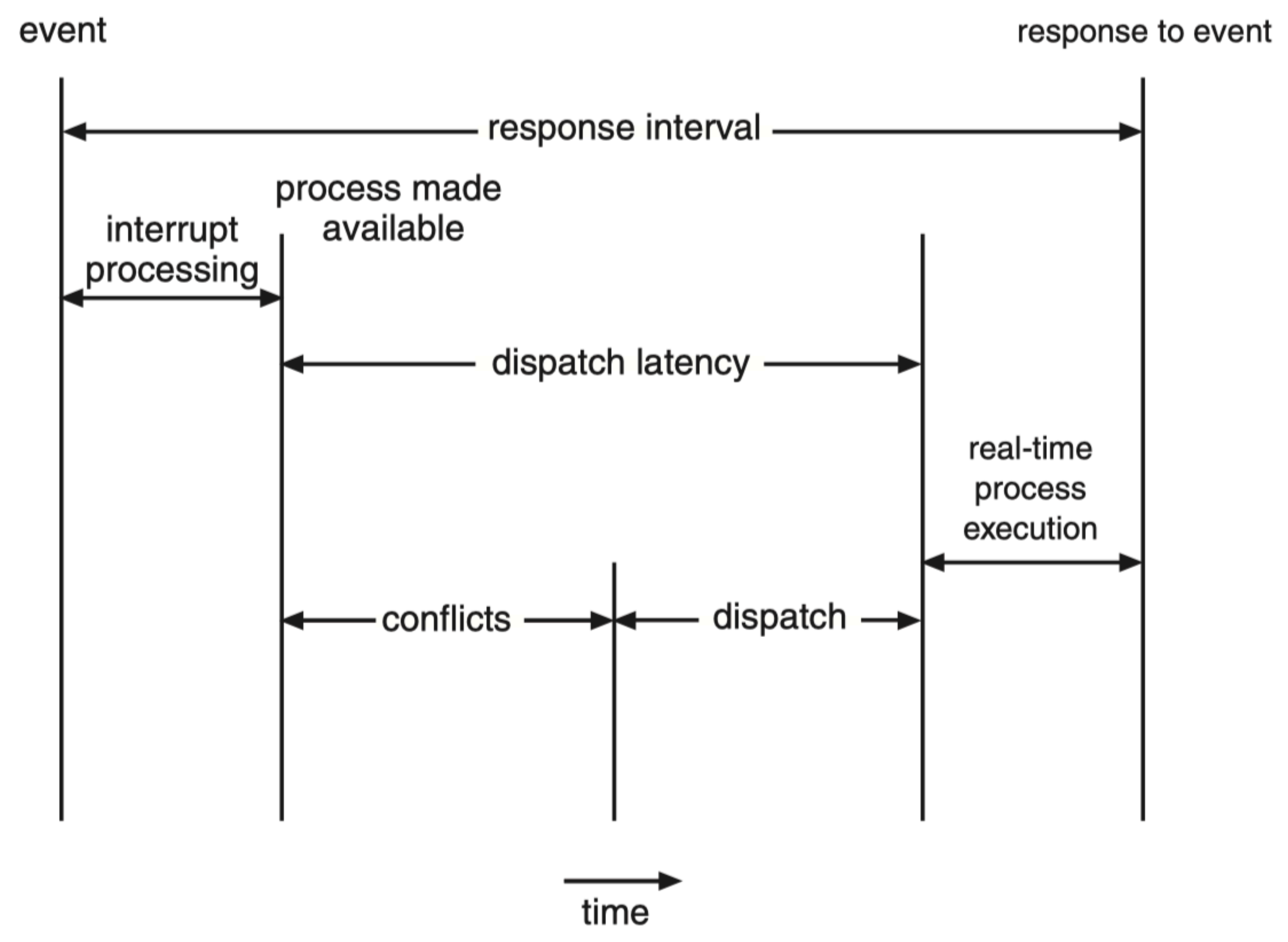
Interrupt Latency

- Defined as the time elapsed since the arrival of the interrupt to the time to start interrupt servicing
- Real-time OSes need to have a very small interrupt latency
- Kernels might mask the interrupts because they are doing something critical - this masking should be for very tiny periods to keep interrupt latency as small as possible



Dispatch Latency

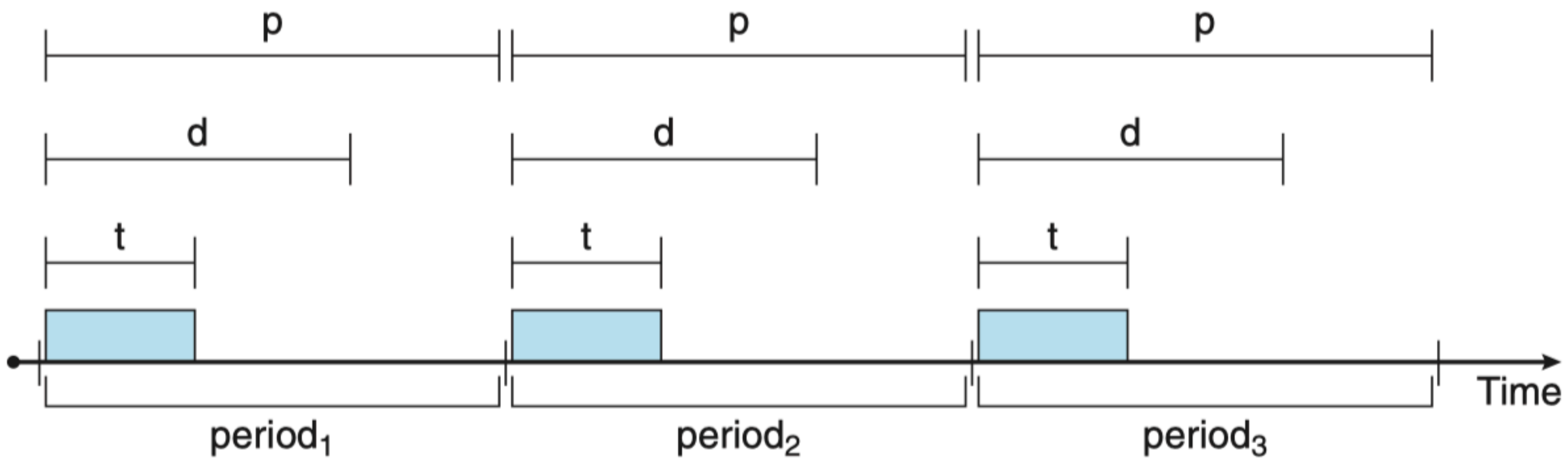
- Time required for the scheduler dispatcher to stop one process and start another process is the dispatch latency
- Preemptive kernels reduce the dispatch latency to a minimal value - no deferring of process switch
- Conflict phase includes - preemption of any running process, time to release resources held by other (non real-time processes) for running this process



Real-Time Scheduling Jobs

- Real-time scheduling jobs have to be known precisely unlike non real case
- So, we start with a rigorous specification of the job requirements
  - Jobs have to stick to the requirements
  - System has to provide computing support for the accepted jobs
  - System can reject some jobs as “cannot” support
  - Specification is used to determine the feasibility

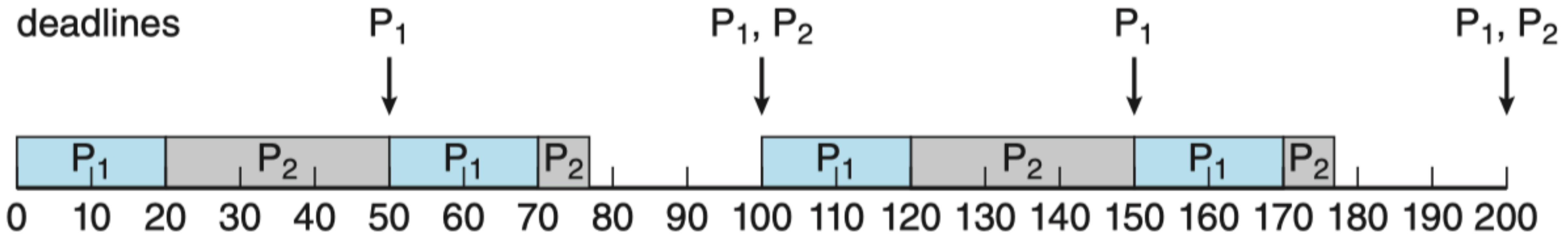
- Processes are periodic
- Runs a definite amount of time
- There is a deadline before which each process needs to complete at every execution



Rate Monotonic Scheduling

- Rate-monotonic scheduler uses a static priority policy with preemption
- High priority incoming process will kick out low priority process
- Periodic task is assigned a priority inversely proportional on its period
- Higher priority is assigned to processes that require CPU more often
- CPU bursts for a process remains the same for all its arrivals

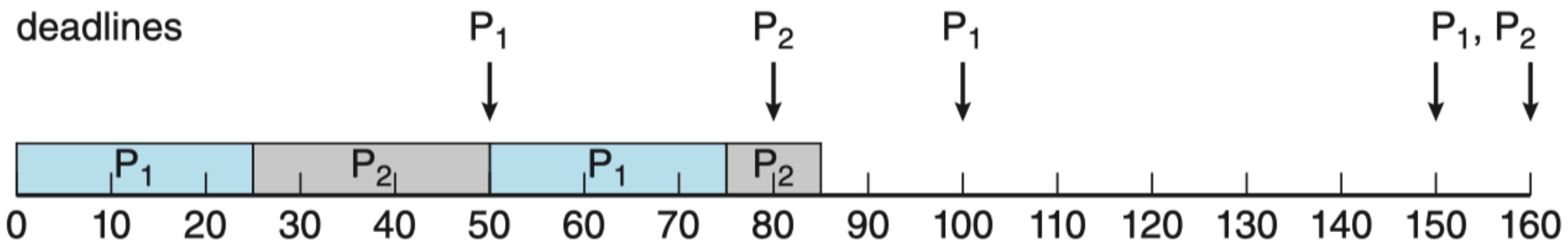
- Two processes: P_1 and P_2
- Period of P_1 and P_2 are 50 and 100
- Processing times are $t_1 = 20, t_2 = 35$
- CPU utilization for both tasks $\sum t_i/p_i = 20/50 + 35/100 = 0.4 + 0.35 = 0.75$
- Schedule should be feasible



Rate Monotonic Scheduling

- RMS is considered optimal - if RMS cannot schedule, no algorithm with static priority assignment can schedule the tasks
- Consider P_1 with a period of 50 and burst $t_1 = 25$. For P_2 the values are 80 and $t_2 = 35$. RMS assigns P_1 higher priority as it has shorter period (same as the deadline for this case).
- Total CPU utilization is $(25/50) + (35/80) = 0.94$

- Initially, P_1 runs until it completes its CPU burst at 25 units. P_2 begins running, but at 50 it is preempted by P_1 (it has higher priority)
- P_1 runs from 50 to 75. And P_2 runs afterwards and it has 10 left and needs to complete by 80 - misses the deadline!

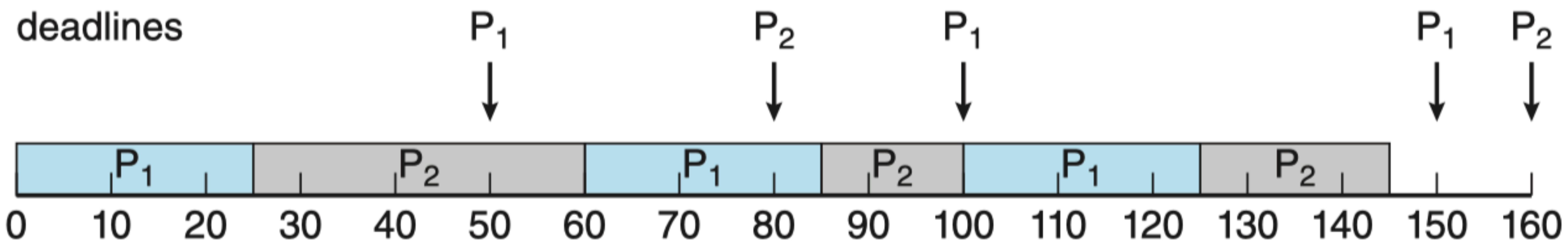


Rate Monotonic Scheduling

- RMS can schedule up to a CPU utilization given by $N(2^{1/N} - 1)$, where N is the number of processes
- $N = 1$, we can reach 100% utilization
- $N = 2$, we can reach a max of 83% utilization - that is why we were not able to schedule the previous problem

Earliest Deadline First Scheduling

- EDF assigns priorities dynamically to the jobs
- Earlier the deadline, higher the priority
- Process needs to announce its deadline as soon as it become runnable
- Consider P_1 with a period of 50 and burst $t_1 = 25$. For P_2 the values are 80 and $t_2 = 35$.
- P_1 has the earlier deadline - higher priority
- EDF also preempts when a high priority task arrives (task with an earlier deadline)
- EDF can theoretically reach 100% CPU utilization



Proportional Share Schedulers

- Proportional share schedulers allocate T shares among all applications. If an application gets N shares out of T, it is assured of getting N/T share of the processor time.
- Suppose T = 100 shares divided among A, B, C
- A gets 50 shares, B gets 15 shares, and C gets 20 shares.
- A is assured 50% of processor time, B has 15% of the processor time, etc
- Proportional share schedulers work with an admission control policy which limits the share distribution - admit a client if there are sufficient number of shares available

Implementing Proportional Share

- Example: Consider an application that is written in two different ways (a) with multi-threading (kernel level), (b) with no threading
- If we use non proportional share scheduling, the multi-threaded version will get higher CPU share - why? CPU scheduler has many kernel threads in the ready queue
- Lets say we have 5 users and 4 of them running multi-threaded programs (5 KT each). One user has non multi-threaded version
- There are 21 kernel threads. The user running non MT version get 1/21 CPU fraction
- Others get 5/21 each - unfair to the one using non MT version - they are all equal in terms of eligibility to use CPU

- Proportional share scheduling can fix the problem
- Lottery scheduling is one simple way of doing proportional share scheduling
- Create 25 lottery tickets distribute them equally among users
- Draw ticket and the holder of the ticket gets the CPU
- The non MT version will get picked more often because it has more tickets!