Operating System Concepts

Lecture 13: CPU Scheduling — Part 2

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

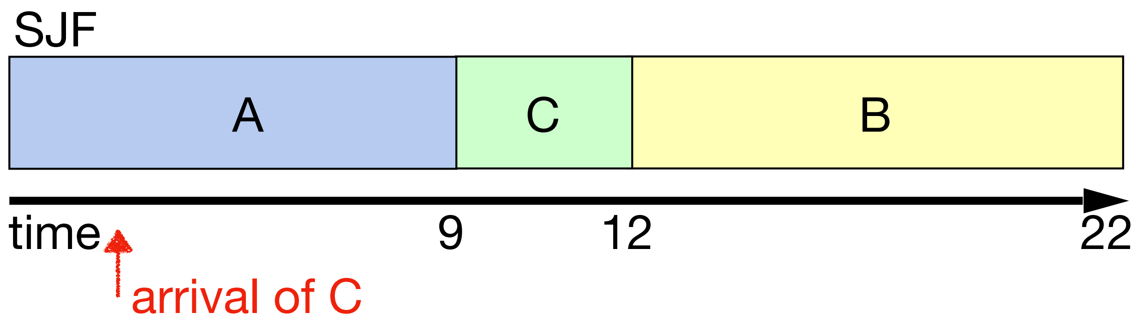
MWF 12:00-12:50 VVC 2 215

Today's class

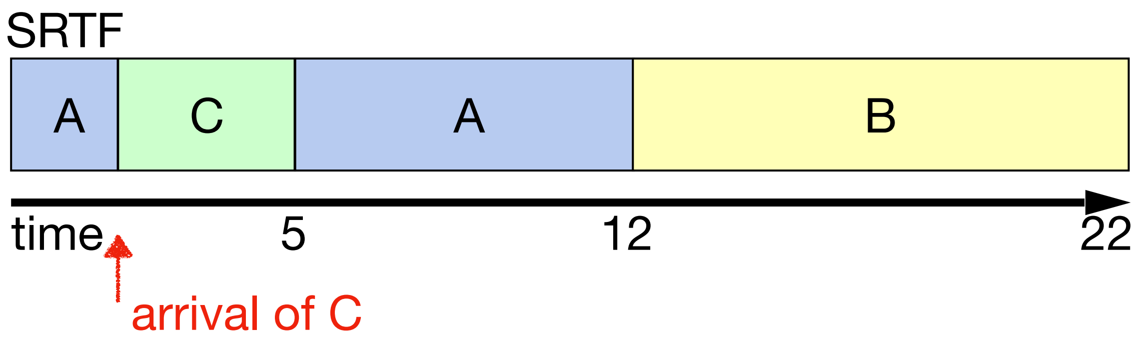
- Scheduling algorithms
  - SJF/SRTF scheduling
  - Priority scheduling
  - MFQ: Multilevel Feedback Queues
  - Proportional-share scheduling (lottery scheduling)
- Multiprocessor scheduling
- Scheduling on real-time systems

Comparing SRTF and SJF

Process	Burst length	Arrival time
A	9	0
B	10	0
C	3	2



Avg. turnaround time:
[9+(12-2)+22]/3=13.6



Avg. turnaround time:
[(5-2)+12+22]/3=12.3

But do we know the burst length of a process?

- no, the burst length is not known a priori

But do we know the burst length of a process?

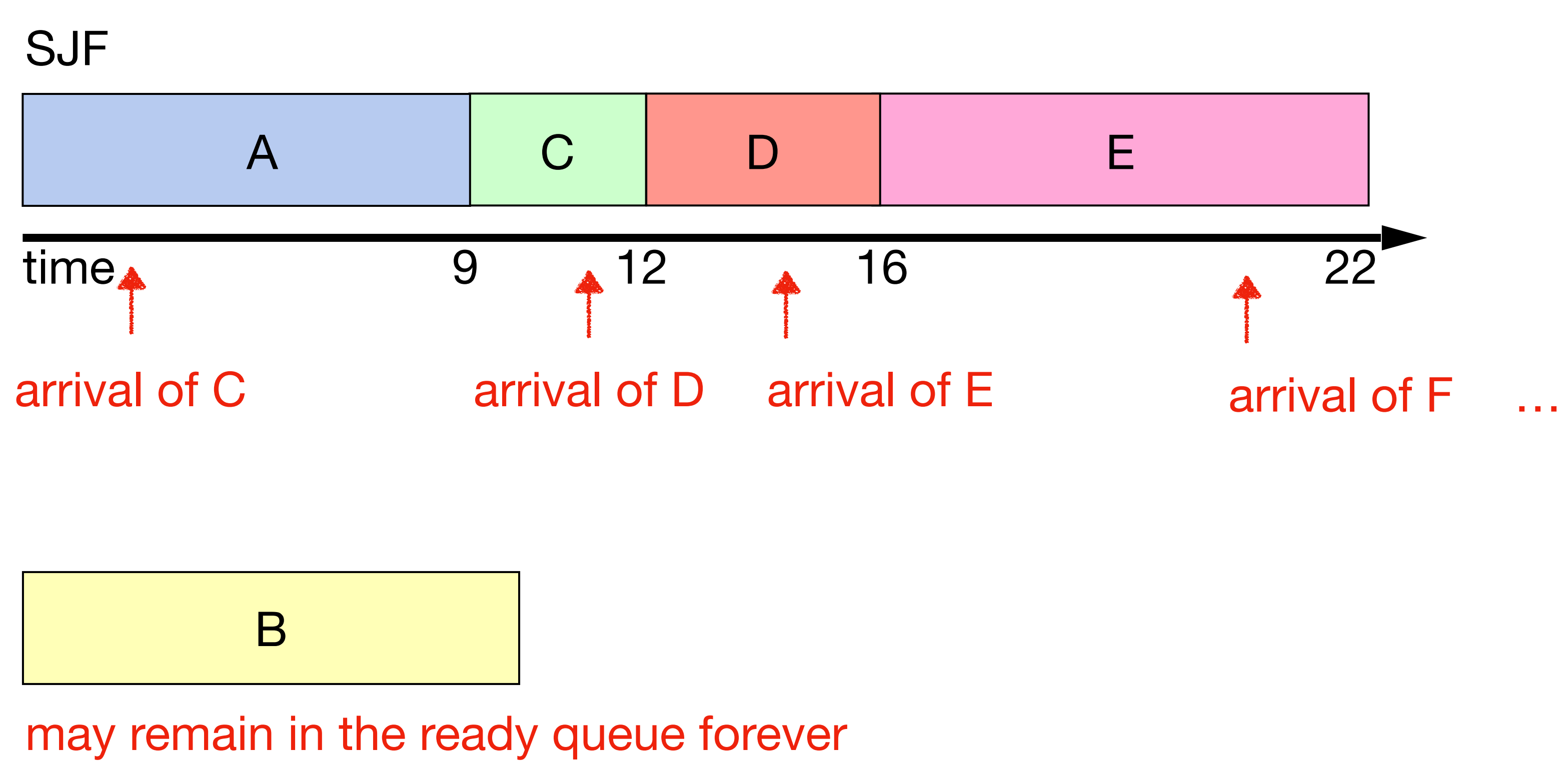
- no, the burst length is not known a priori
- can estimate based on previous burst lengths
  - the exponential moving average estimator
    - $\hat{b}[1] = b[1]$
    - $\hat{b}[t] = \eta b[t] + (1 - \eta)\hat{b}[t-1]$ for $\eta \in (0,1]$

But do we know the burst length of a process?

- no, the burst length is not known a priori
- can estimate based on previous burst lengths
  - the exponential moving average estimator
    - $\hat{b}[1] = b[1]$
    - $\hat{b}[t] = \eta b[t] + (1 - \eta)\hat{b}[t-1]$ for $\eta \in (0,1]$
- can ask users
  - they user can lie (declare a shorter burst length) to game the system
  - how to encourage truthfulness?
    - terminate execution after the specified burst length has passed

SJF is starvation prone

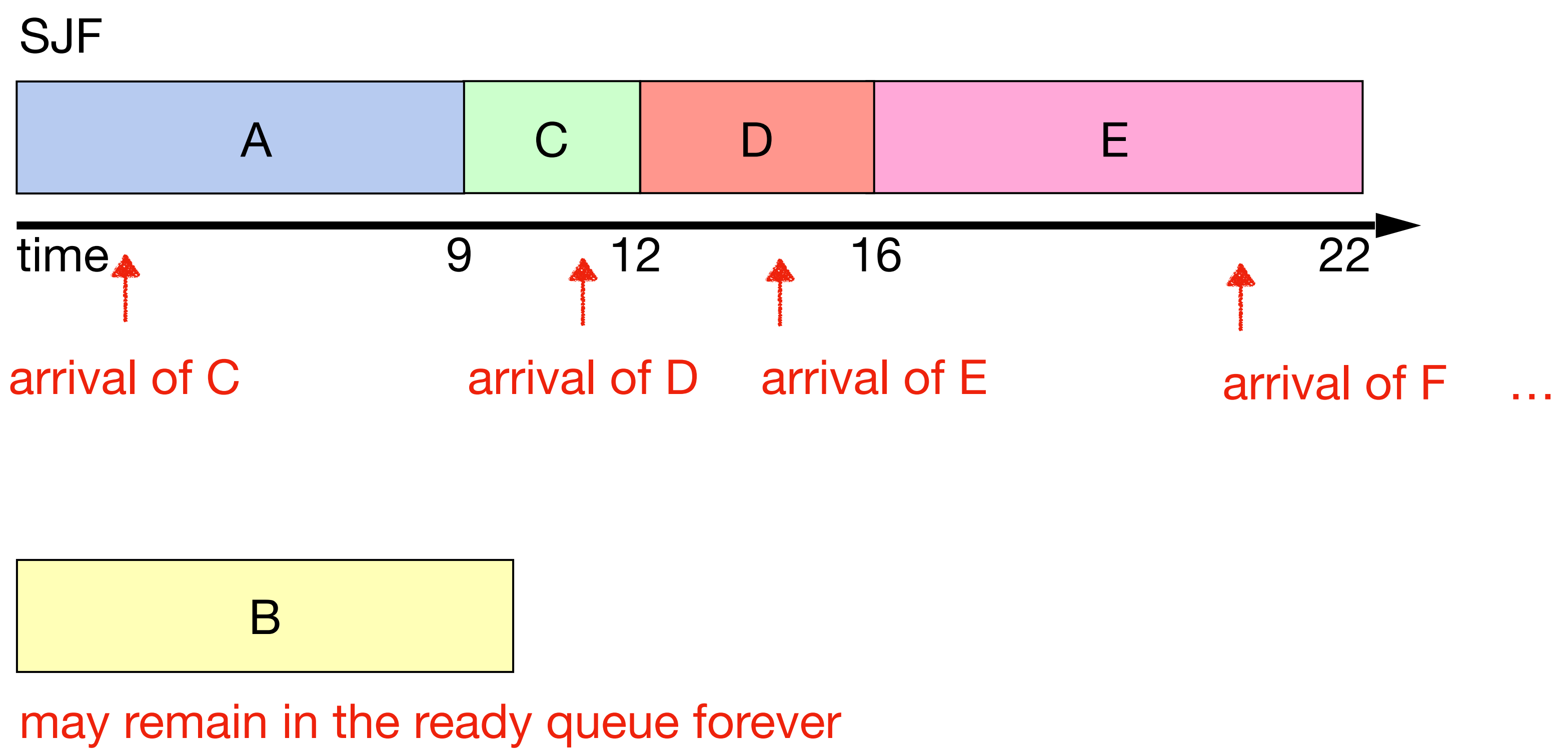
Process	Burst length	Arrival time
A	9	0
B	10	0
C	3	2
D	4	11
E	6	14



SJF is starvation prone

Is SRJT starvation prone too?

Process	Burst length	Arrival time
A	9	0
B	10	0
C	3	2
D	4	11
E	6	14



Can we design a policy that is responsive, fair, and starvation free with low overhead?

Priority scheduling

- is a multilevel queue scheduling policy
  - each queue has a certain priority (usually defined by user based on the nature of their task)
  - tasks cannot switch from one queue to another queue

Priority scheduling

- is a multilevel queue scheduling policy
  - each queue has a certain priority (usually defined by user based on the nature of their task)
  - tasks cannot switch from one queue to another queue
- scheduler picks the ready process/thread with the highest priority
  - tasks start running from the highest priority queue
  - again low-priority tasks may never run (starvation)

Priority scheduling

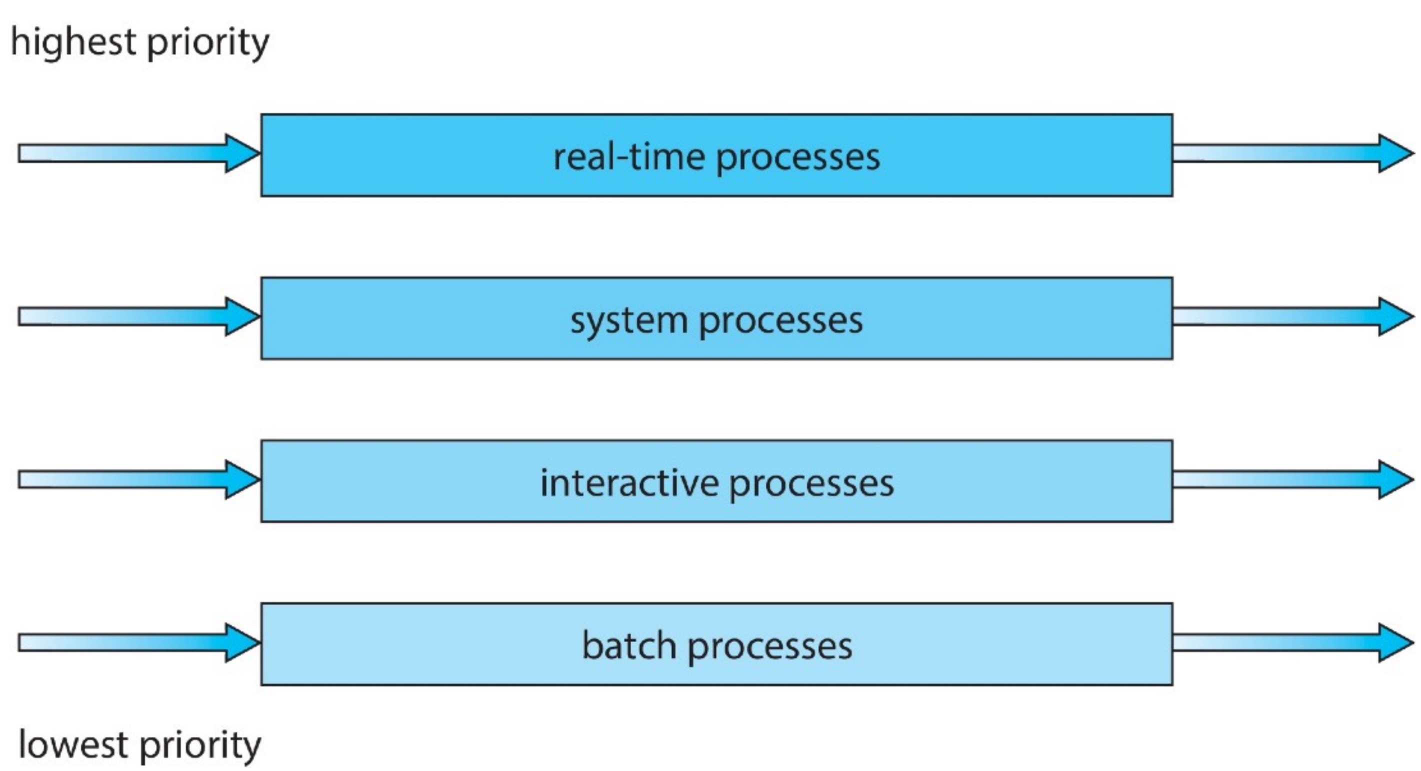
- is a multilevel queue scheduling policy
  - each queue has a certain priority (usually defined by user based on the nature of their task)
  - tasks cannot switch from one queue to another queue
- scheduler picks the ready process/thread with the highest priority
  - tasks start running from the highest priority queue
  - again low-priority tasks may never run (starvation)
- two solutions for **dealing with starvation**
  - (a) if time slice expires, task drops one level; long-running compute tasks are demoted to low priority
  - (b) time-slice among the queues (e.g., 70% to highest priority, 20% to middle, 10% to low)

Priority scheduling

- is a multilevel queue scheduling policy
  - each queue has a certain priority (usually defined by user based on the nature of their task)
  - tasks cannot switch from one queue to another queue
- scheduler picks the ready process/thread with the highest priority
  - tasks start running from the highest priority queue
  - again low-priority tasks may never run (starvation)
- two solutions for **dealing with starvation**
  - (a) if time slice expires, task drops one level; long-running compute tasks are demoted to low priority
  - (b) time-slice among the queues (e.g., 70% to highest priority, 20% to middle, 10% to low)
- these solutions could improve fairness but increase response time

How to lower the response time?

- high priority queues have short time slices
- low priority queues have longer time slices

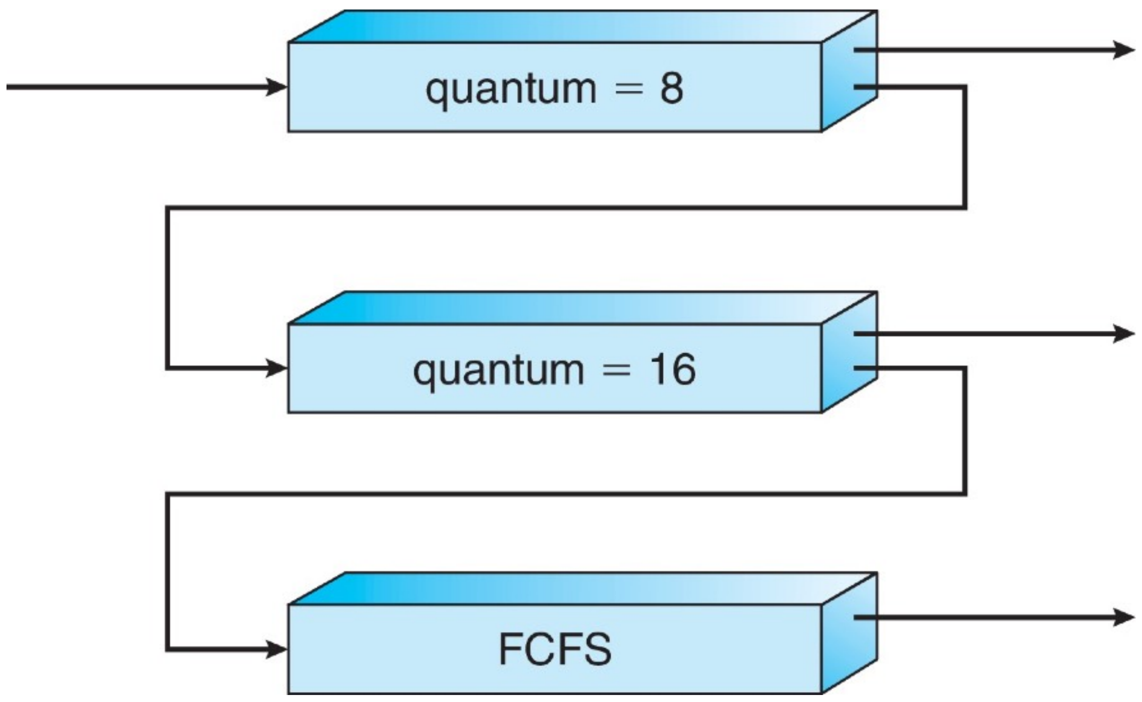


Priority inversion

- where a high priority task is blocked waiting on a low priority task
- the low priority task must run for the high priority task to make progress
- solution: a high priority task temporarily grants a lower 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

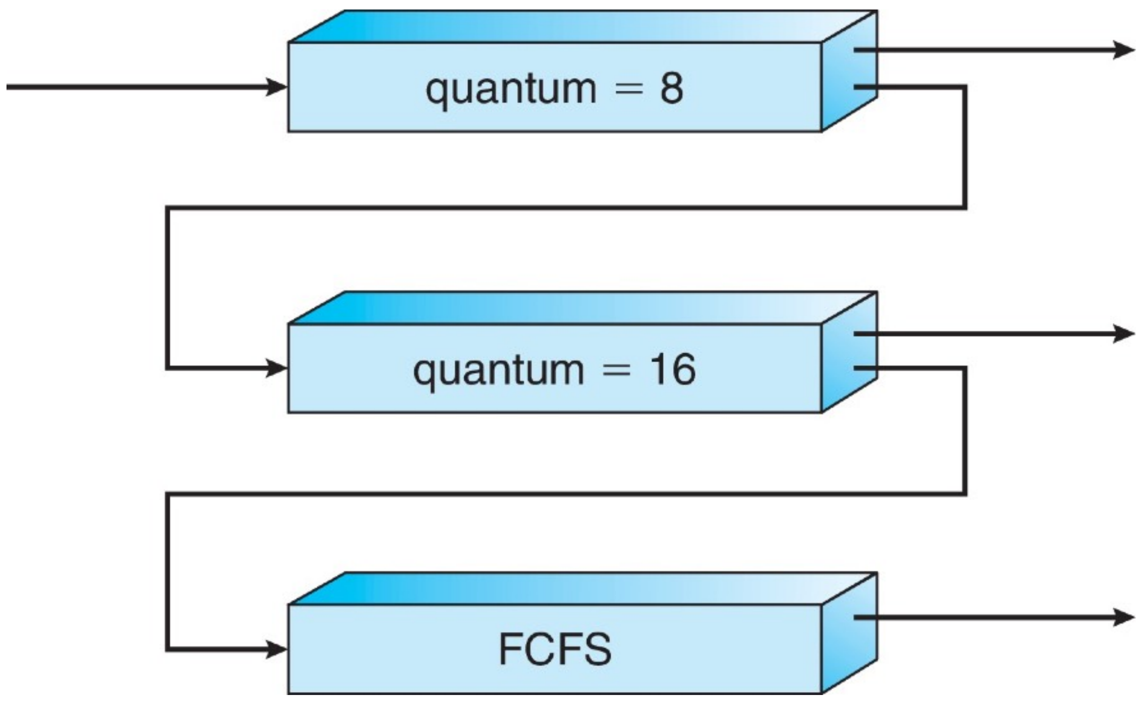
MFQ scheduling

- multilevel feedback queue (MFQ) is a set of round robin queues with different priorities
  - use Round Robin scheduling at each priority level, running the jobs in highest priority queue first
  - once those finish, run jobs at the next highest priority queue, etc.



MFQ scheduling

- multilevel feedback queue (MFQ) is a set of round robin queues with different priorities
  - use Round Robin scheduling at each priority level, running the jobs in highest priority queue first
  - once those finish, run jobs at the next highest priority queue, etc.
- round robin time slice increases exponentially at lower priorities



How to adjust priorities?

- job starts in highest priority queue
  - if job's time slice expires, drop its priority one level
  - if job's time slice does not expire (the context switch comes from an I/O request instead), then increase its priority one level (up to the top priority level)

How to adjust priorities?

- job starts in highest priority queue
  - if job's time slice expires, drop its priority one level
  - if job's time slice does not expire (the context switch comes from an I/O request instead), 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

How to adjust priorities?

- job starts in highest priority queue
  - if job's time slice expires, drop its priority one level
  - if job's time slice does not expire (the context switch comes from an I/O request instead), 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
- 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 increase in fairness by giving long jobs a fraction of the CPU when shorter jobs are available will degrade average waiting time

Improving fairness

- since SJF is optimal but unfair
  - any increase in fairness by giving long jobs a fraction of the CPU 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 some number of lottery tickets
  - on each time slice, randomly pick a winning ticket and run the winning process

Lottery scheduling

- give every process some number of lottery tickets
  - on each time slice, randomly pick a winning ticket and run the winning process
- on average, CPU time 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

Lottery scheduling

- give every process some number of lottery tickets
  - on each time slice, randomly pick a winning ticket and run the winning process
- on average, CPU time 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 <u>at least one ticket</u>
  1. give the most to short running processes, and fewer to long running processes (approximating SJF scheduling)
  2. give the most to high-priority processes, and fewer to low-priority processes (approximating priority scheduling)

Lottery scheduling

- give every process some number of lottery tickets
  - on each time slice, randomly pick a winning ticket and run the winning process
- on average, CPU time 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 <u>at least one ticket</u>
  1. give the most to short running processes, and fewer to long running processes (approximating SJF scheduling)
  2. give the most 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

Lottery scheduling

- give every process some number of lottery tickets
  - on each time slice, randomly pick a winning ticket and run the winning process
- on average, CPU time 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 <u>at least one ticket</u>
  1. give the most to short running processes, and fewer to long running processes (approximating SJF scheduling)
  2. give the most 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
- for priority inversion: donate tickets to process you are waiting on

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

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%

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%

Lottery scheduling example

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

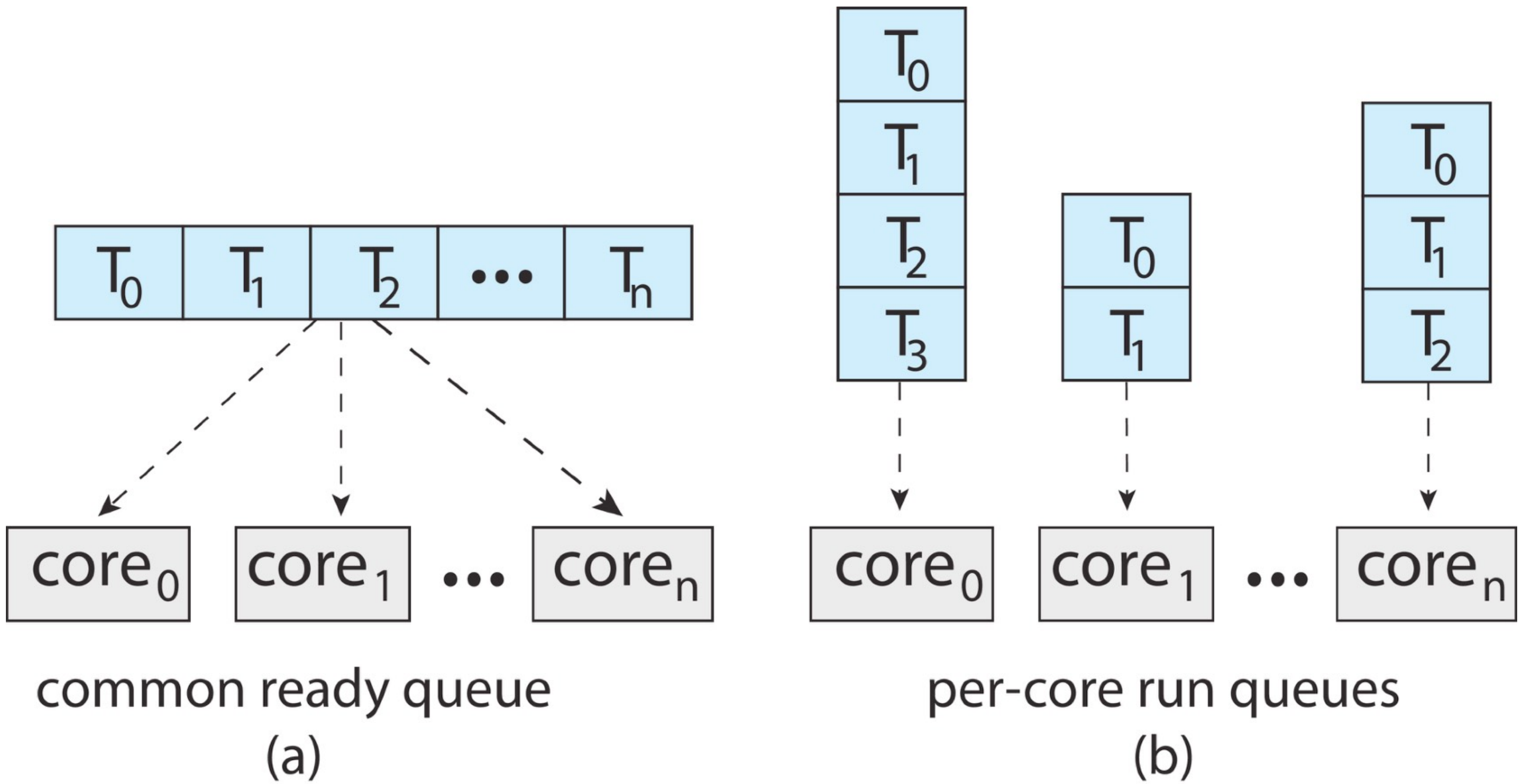
	#tickets	#short jobs	#long jobs	%CPU each short job gets	%CPU each long job gets	
	10	1	1	80	20	
long job added to queue	12	1	2	66.6	16.6	-16.6%
short job added to queue	20	2	2	40	10	-40%
long job removed from queue	18	2	1	44.4	11.1	+11.1%

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 multicore systems



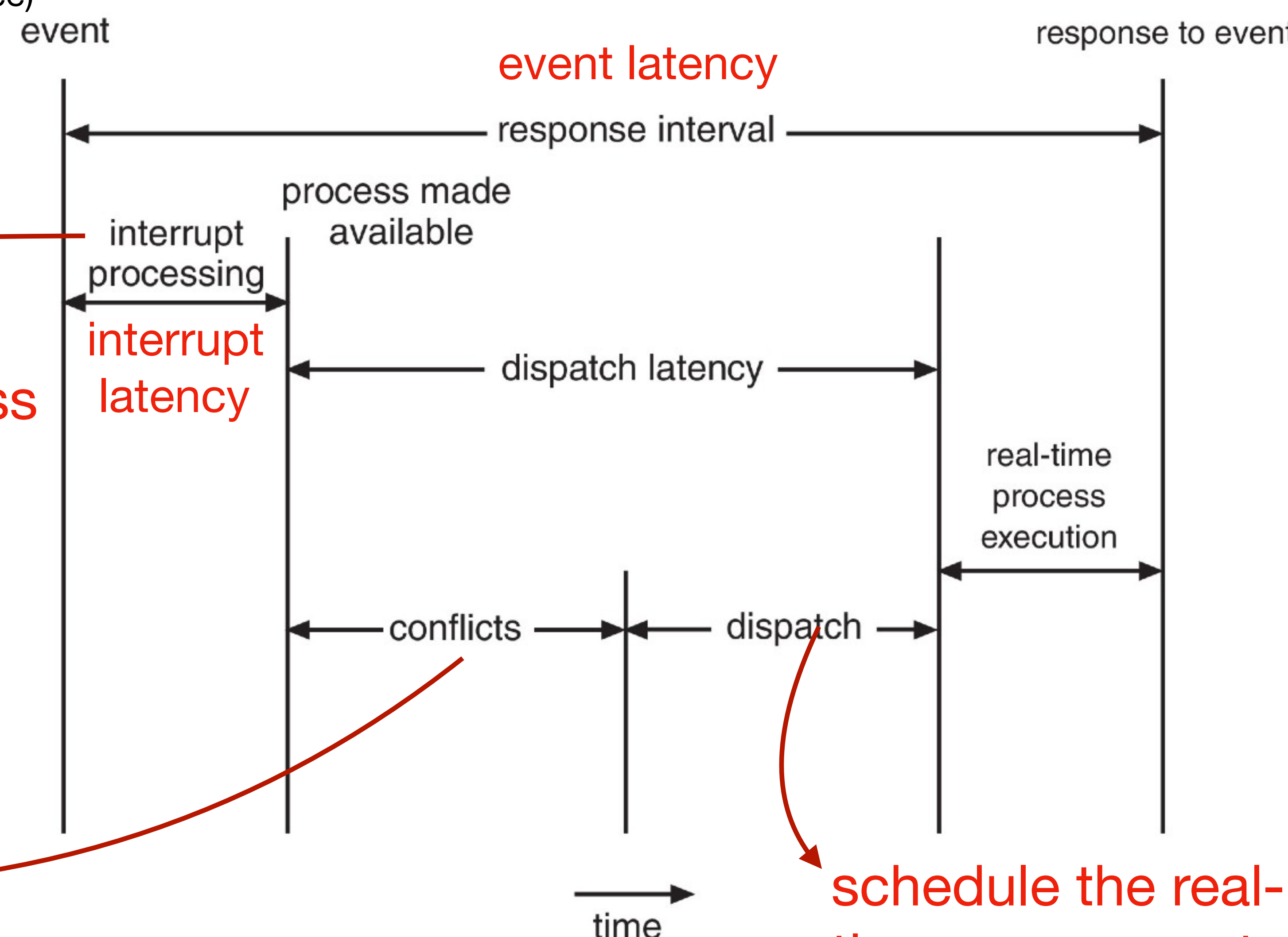
load balancing may be necessary

Real-time systems

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

Real-time systems

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



1. saving the state of the current process
2. determining the interrupt type
3. calling the corresponding ISR

1. preemption of any process running on the CPU
2. freeing up resources

schedule the real-time process onto an available CPU

Real-time scheduling

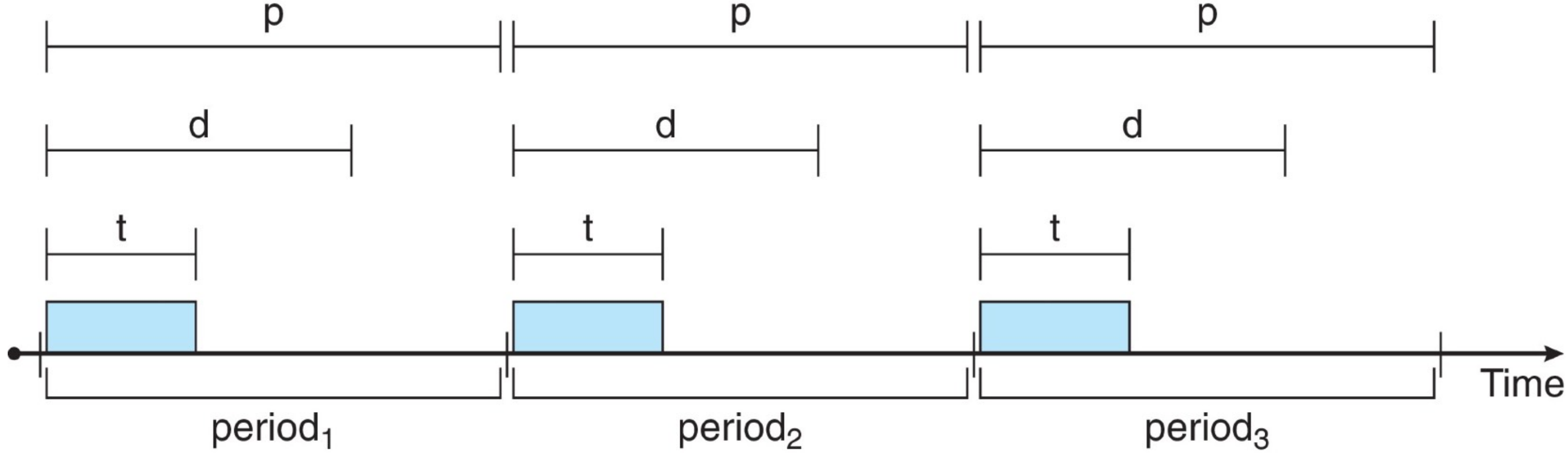
- a preemptive priority-based scheduler is often used in <u>soft real-time systems</u>
  - assigns real-time processes the highest priority
    - e.g., Windows has 32 priority levels, levels 16 to 32 are reserved for real-time processes

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

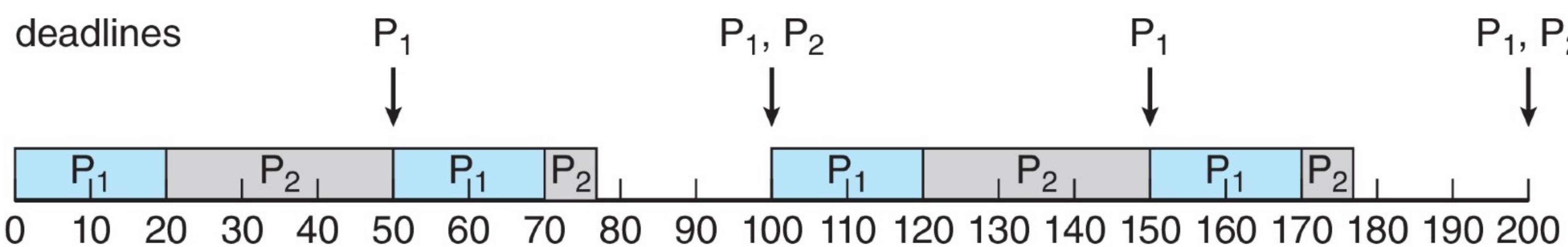
Assumption: periodic tasks

- 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



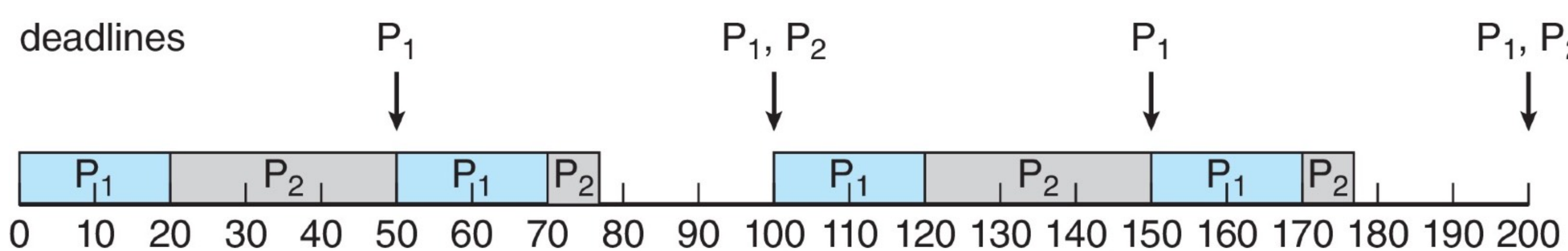
Rate-Monotonic scheduling

- a priority is assigned to each task based on the inverse of its period
  - shorter periods = higher priority;
  - longer periods = lower priority



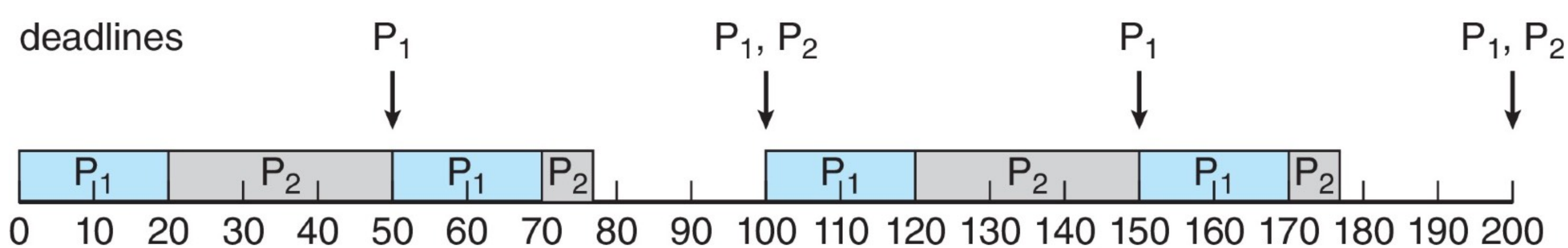
Rate-Monotonic scheduling

- a priority is assigned to each task based on the inverse of its period
  - shorter periods = higher priority;
  - longer periods = lower priority
- 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



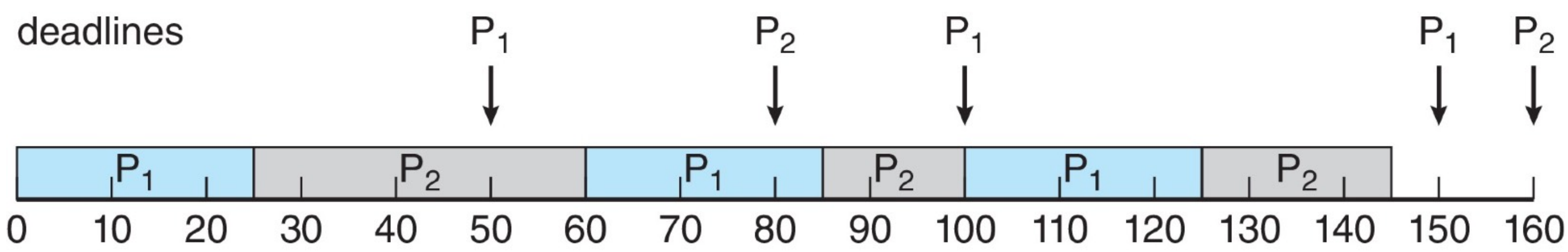
Rate-Monotonic scheduling

- a priority is assigned to each task based on the inverse of its period
  - shorter periods = higher priority;
  - longer periods = lower priority
- 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: reject a submitted task if $\sum_i \frac{t_i}{p_i} > 1$



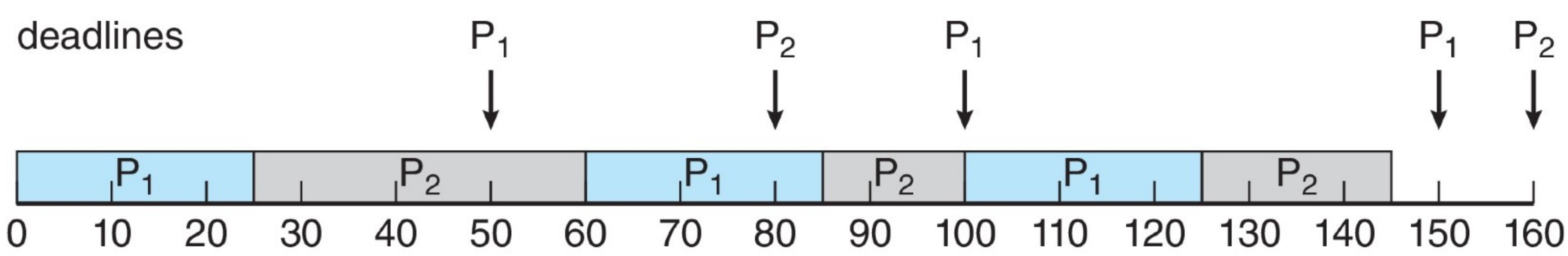
Earliest-Deadline-First (EDF) scheduling

- priorities are assigned according to deadlines
  - the earlier the deadline, the higher the priority
  - does not require tasks to be periodic!



Earliest-Deadline-First (EDF) scheduling

- priorities are assigned according to deadlines
  - the earlier the deadline, the higher the priority
  - does not require tasks to be periodic!
- EDF is theoretically optimal

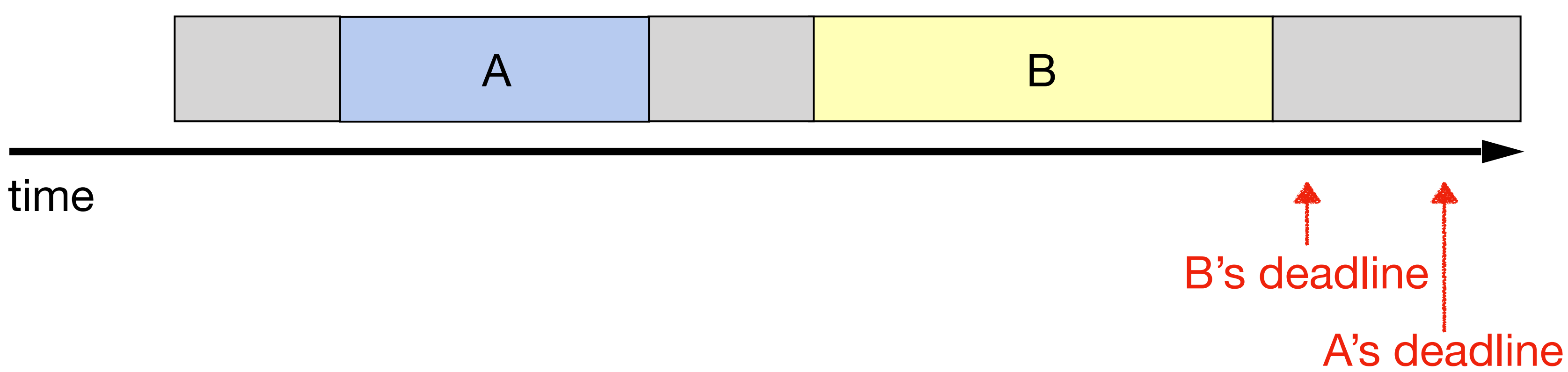


Homework

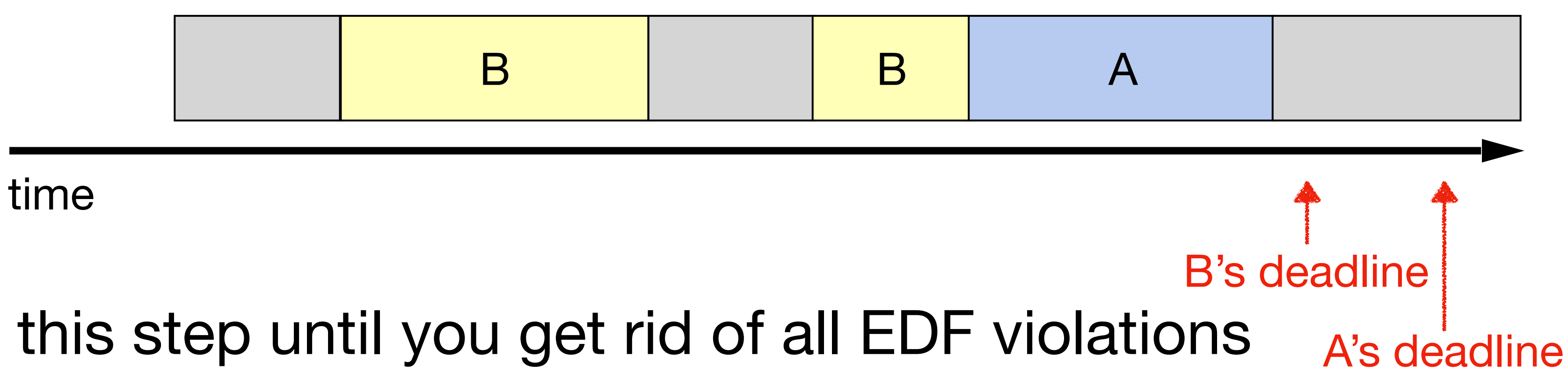
- Prove that when preemption is allowed, the EDF algorithm can produce a feasible schedule of a set S of independent tasks with arbitrary release times and deadlines on a processor if and only if S has a feasible schedule (i.e., a schedule that meets all deadlines)

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