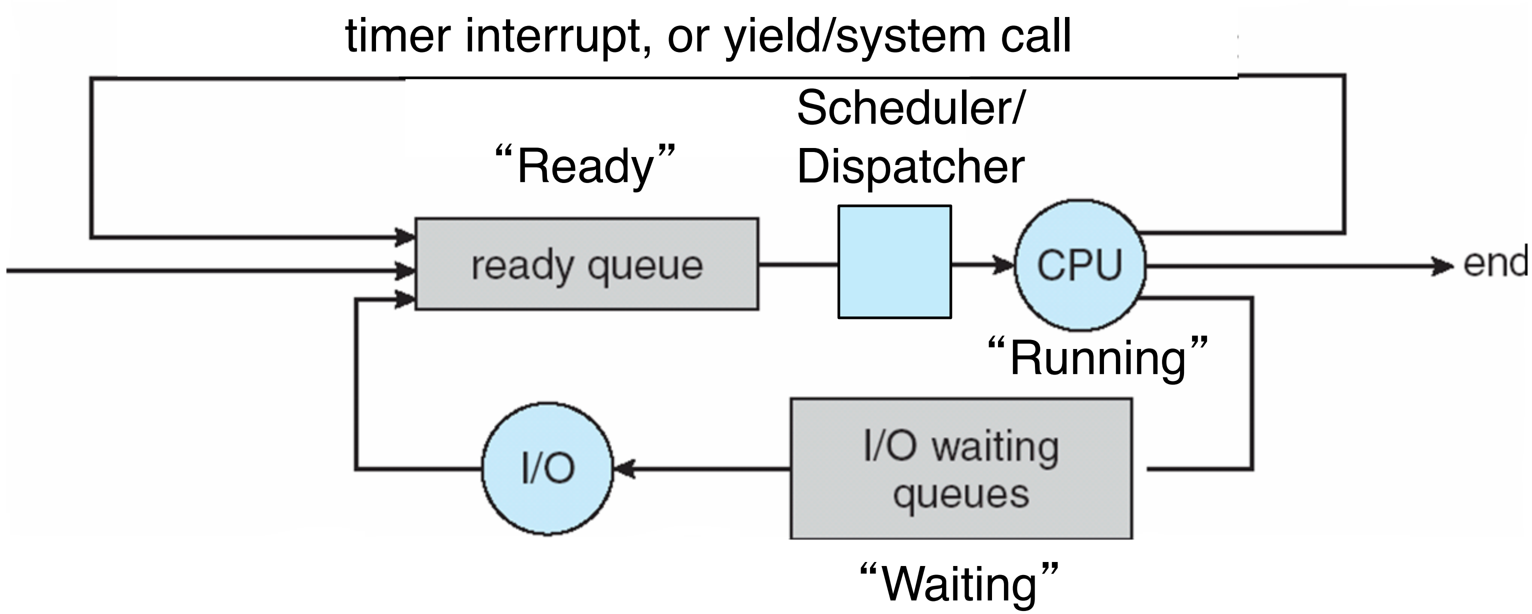
Scheduling: Round Robin, Deadlines, Priorities, Multi-level Feedback Queues

CSCI 3753 Operating Systems

William Mortl, MS and Rick Han, PhD

University of Colorado Boulder

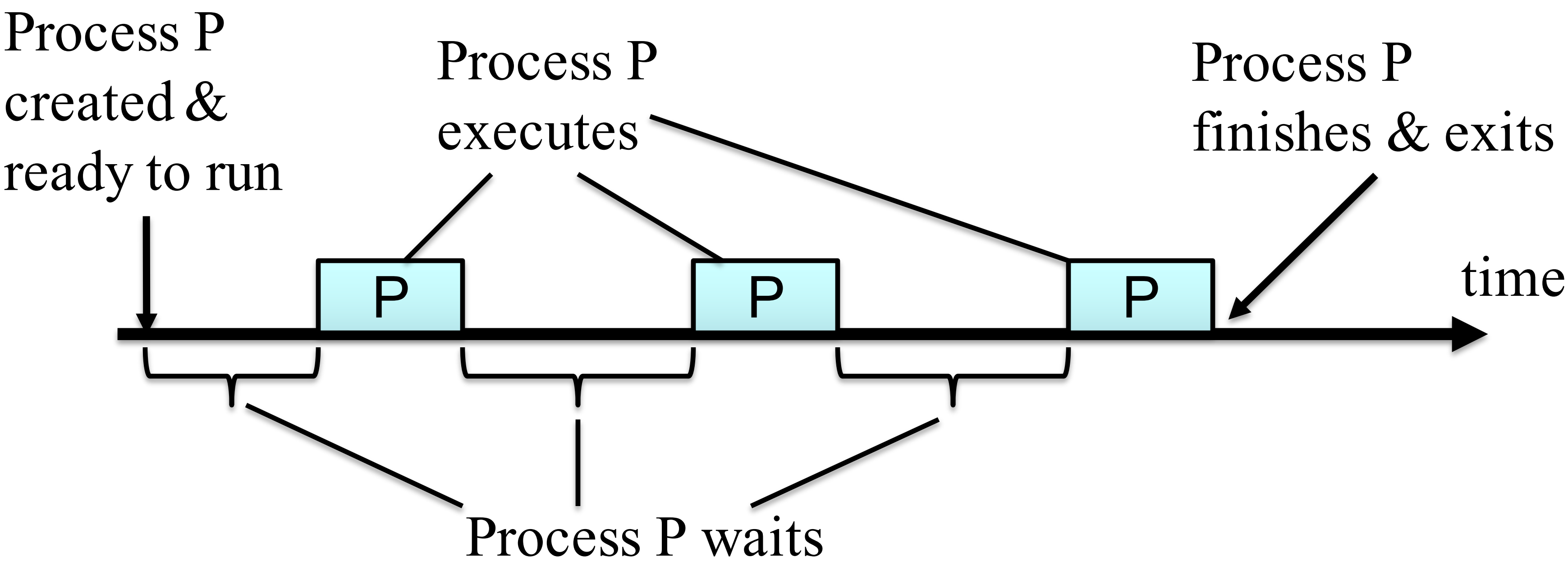
Process Scheduling



If threads are implemented as kernel threads, then OS can schedule threads as well as processes

Modified version of Silberschatz et al slides

Recap

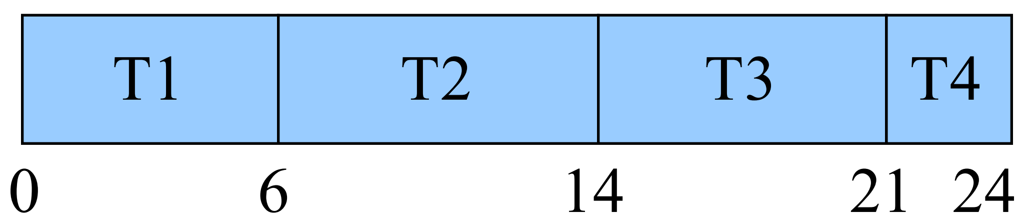


- CPU Scheduler implements a scheduling policy
  – Minimize ave/peak wait time, response time, turnaround time, etc.

First Come First Serve (FCFS) Scheduling

- Tasks are scheduled according to the order they arrive
  - Simple to implement
  - Can result in high variance

Task	CPU Execution Time
T1	6
T2	8
T3	7
T4	3

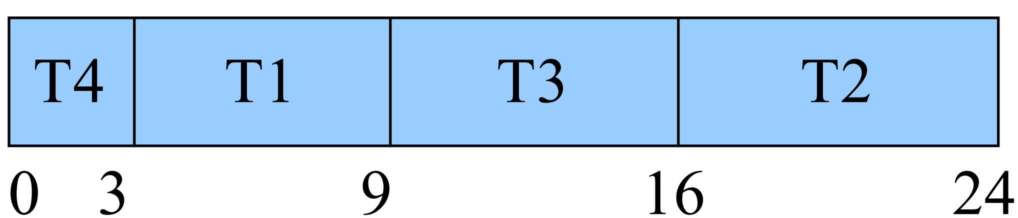


average wait time
= (0+6+14+21)/4
= 10.25 seconds

Shortest Job First Scheduling

- Schedule tasks with the shortest execution times first
  - Can prove this results in the lowest average wait time

Task	CPU Execution Time
T1	6
T2	8
T3	7
T4	3



average wait time
= (0+3+9+16)/4
= 7 seconds

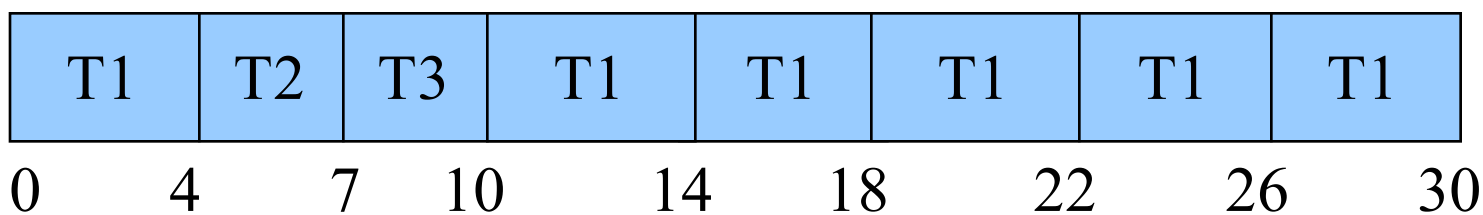
Round Robin Scheduling

- Assume preemptive time slicing
  - A task is forced to relinquish the CPU before it's necessarily done
  - Periodic timer interrupt transfers control to the CPU scheduler, which *rotates* among the processes in the ready queue, giving each a time slice,
  - e.g. if there are 3 tasks T1, T2, & T3, then the scheduler will keep rotating among the three: T1, T2, T3, T1, T2, T3, T1, …
  - treats the ready queue as a circular queue

Round Robin Scheduling

- Example: let time slice = 4 ms
- Now T1 is time sliced out, and T2 and T3 are allowed to run sooner than FCFS
- average response time is fast at 3.66 ms
  - Compare to FCFS w/ long 1st task

Task	CPU Execution Time (ms)
T1	24
T2	3
T3	3



Round Robin Scheduling

- useful to support interactive applications in multitasking systems
  - hence is a popular scheduling algorithm
- Properties:
  - Simple to implement: just rotate, and don't need to know execution times a priori
  - Fair: If there are n tasks, each task gets 1/n of CPU
- A task can finish before its time slice is up
  - Scheduler just selects the next task in the queue

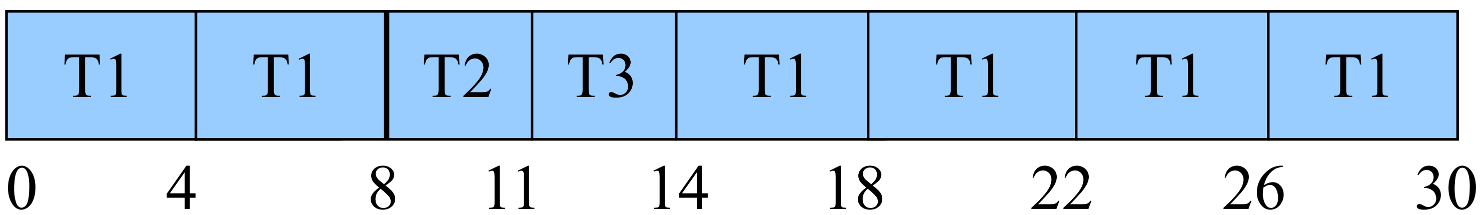
Weighted Round Robin

- Give some some tasks more time slices than others
  - This is a way of implementing priorities – higher priority tasks get more time slices per round
  - If task T_i gets N_i slots per round, then the fraction α_i of the CPU bandwidth that task i gets is:

$$\alpha_\iota = \frac{N_i}{\sum_i N_i}$$

Weighted Round Robin

- In previous example, could give T1 2 time slices, and T2 and T3 only 1 each round



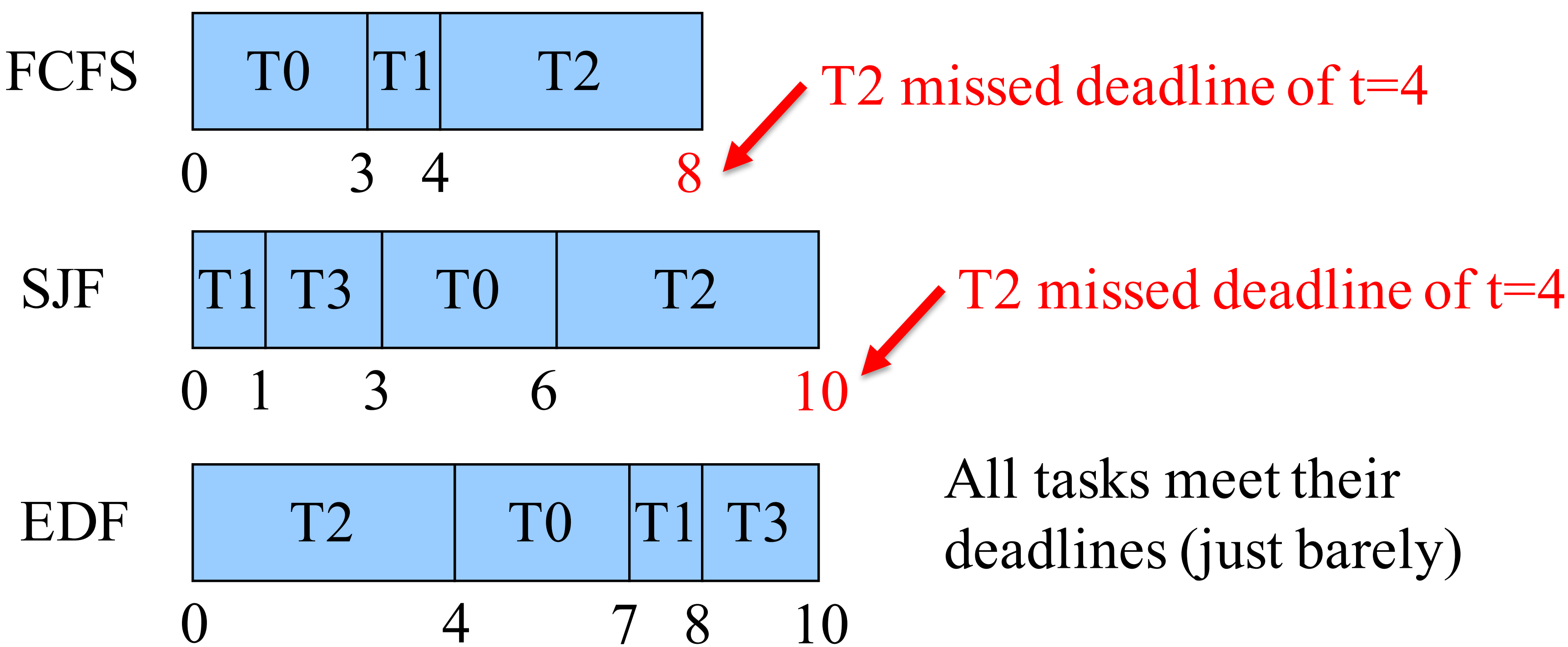
Deadline Scheduling

- Hard real time systems require that certain tasks *must* finish executing by a certain time, or the system fails
  - e.g. robots and self-driving cars need a real time OS (RTOS) whose tasks (actuating an arm/leg or steering wheel) must be scheduled by a certain deadline

Task	CPU Execution Time	Deadline from now
T0	3	7
T1	1	9
T2	4	4
T3	2	10

Earliest Deadline First (EDF) Scheduling

- Choose the task with the earliest deadline
  - This task most urgently needs to be completed



Deadline Scheduling

- Even EDF may not be able to meet all deadlines:
  - In previous example, if T3's deadline was t=9, then EDF cannot meet T3's deadline
- When EDF fails, the results of further failures, i.e. missed deadlines, are unpredictable
  - Which tasks miss their deadlines depends on when the failure occurred and the system state at that time
    - Could be a cascade of failures
  - This is one disadvantage of EDF

Deadline Scheduling

- Admission control policy
  - Check on entry to system whether a task's deadline can be met,
    - Examine the current set of tasks already in the ready queue and their deadlines
    - If all deadlines can be met with the new task, then admit it. The *schedulability* of the set of real-time tasks has been verified.
    - Else, deny admission to this task if its deadline can't be met.
  - Note FCFS, SJF and priority had no notion of refusing admission

EDF and Preemption

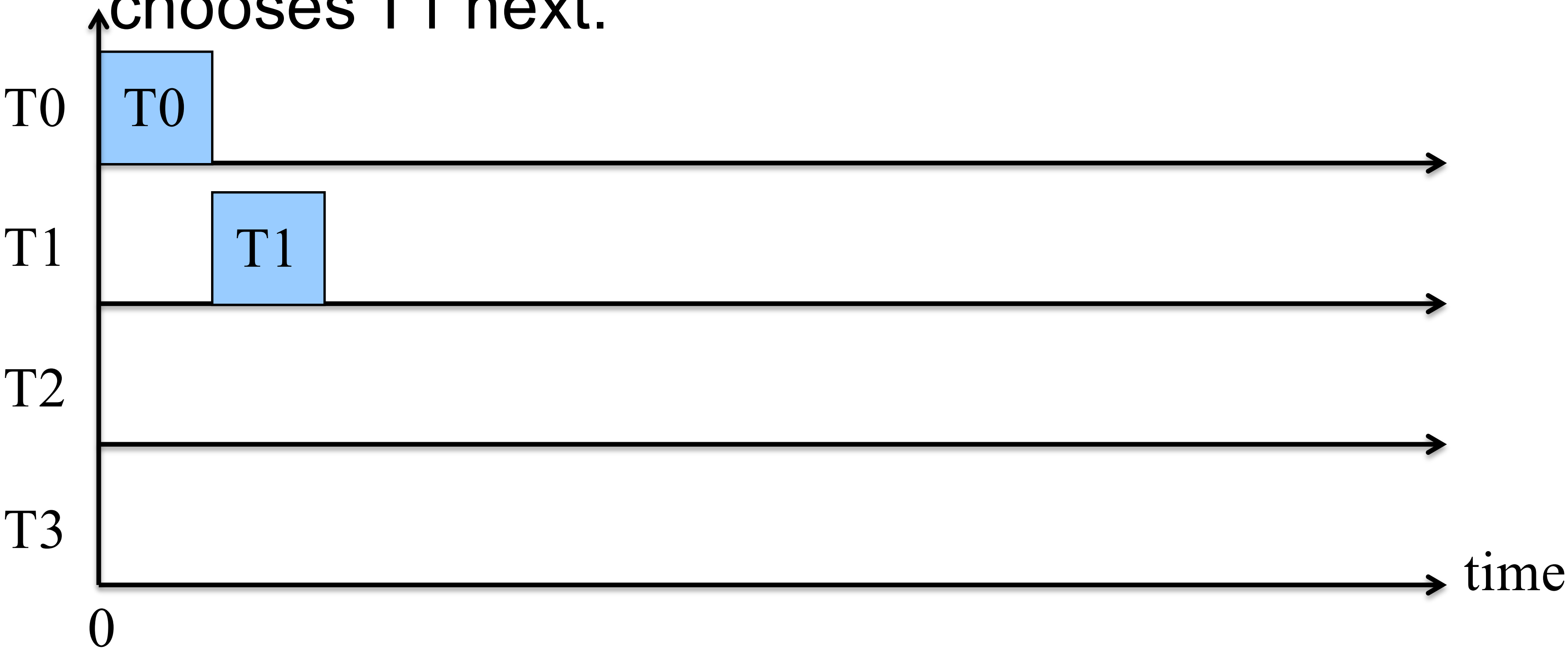
- Assume a preemptively time sliced system
  - A task arriving with an earlier deadline can preempt one currently executing with a later deadline.

Task	CPU Execution Time	Absolute Deadline	Arrival time
T0	1	2	0
T1	2	5	0
T2	2	4	2
T3	2	10	3

Assume in this example time slice = 1, i.e. the executing task is interrupted every second and a new scheduling decision is made

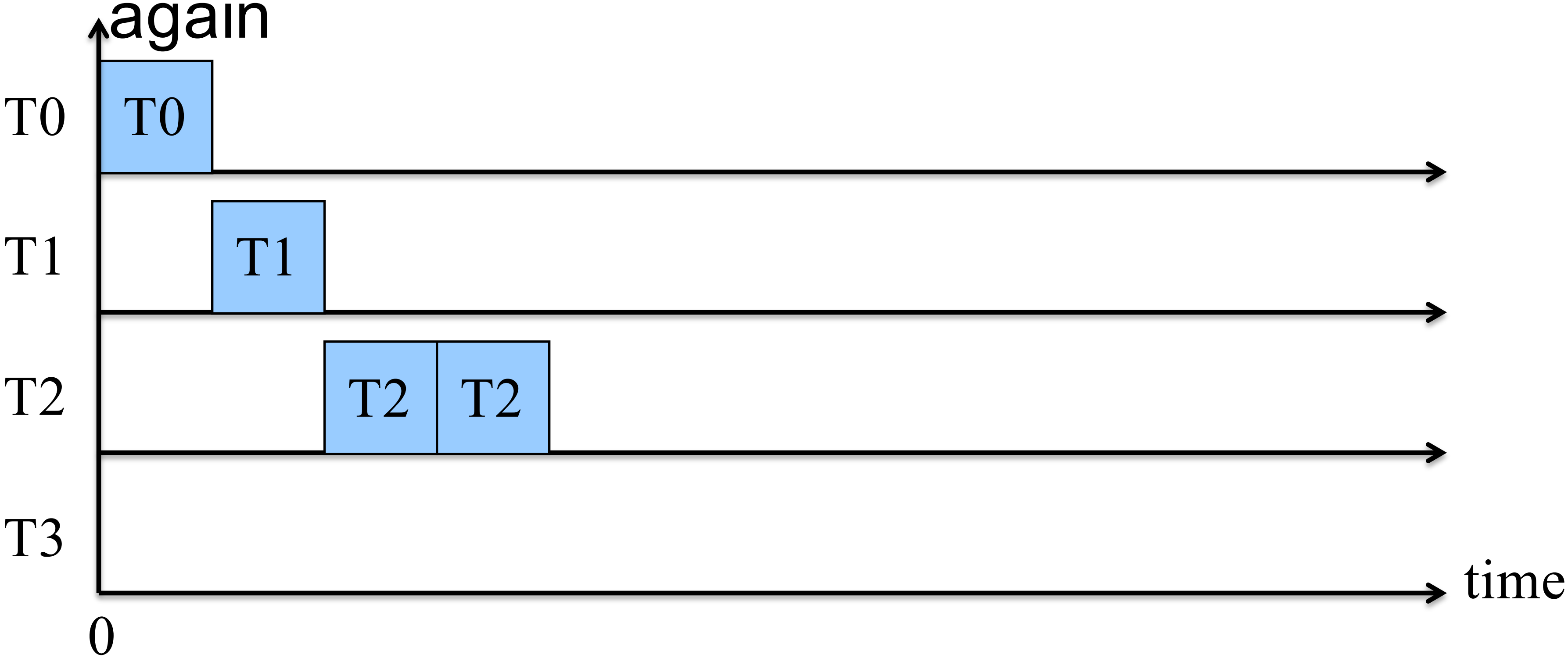
EDF and Preemption

- At time 0, tasks T0 and T1 have arrived. EDF chooses T0.
- At time 1, T0 finishes, makes deadline. EDF chooses T1 next.



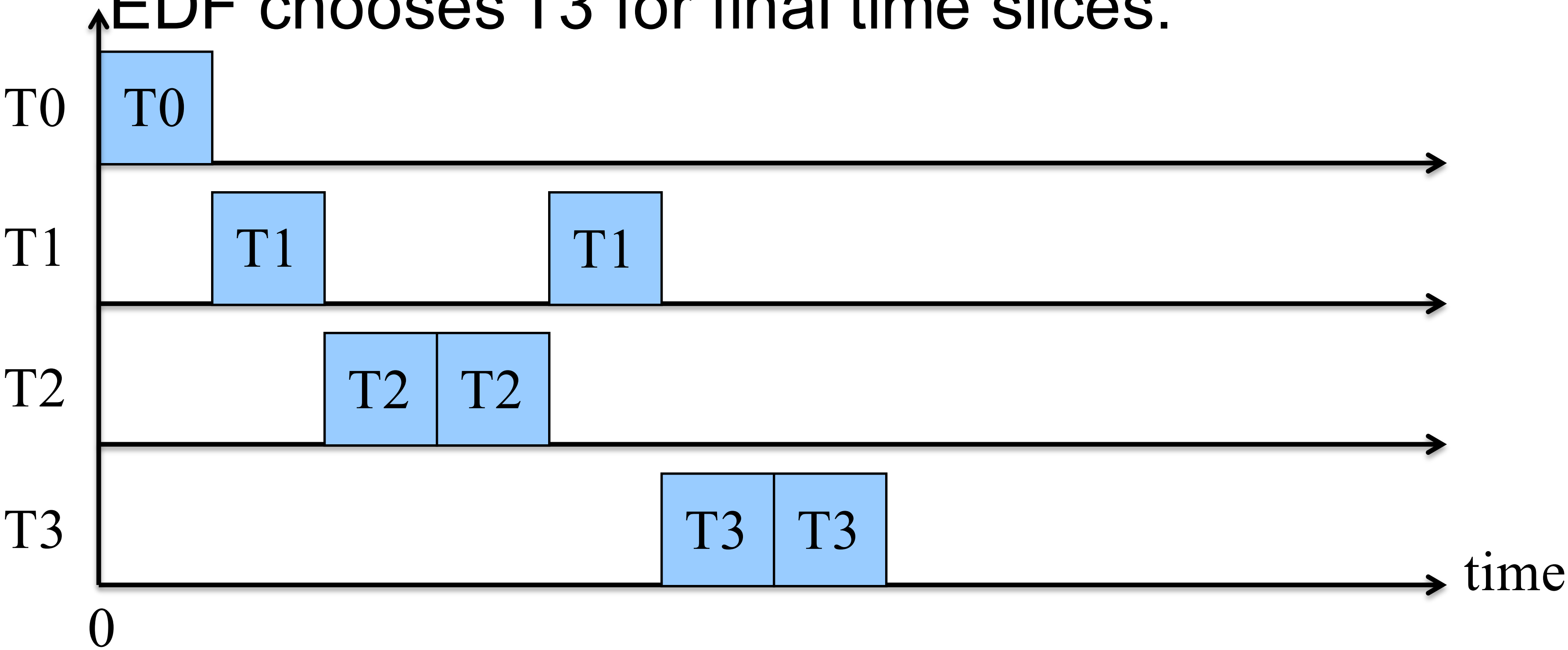
EDF and Preemption

- At time 2, preempt T1. EDF chooses newly arrived T2 with earlier deadline.
- At time 3, preempt T2. EDF chooses T2 again



EDF and Preemption

- At time 4, T2 finishes and makes deadline. EDF chooses T1.
- At time 5, T1 finishes and makes deadline. EDF chooses T3 for final time slices.



Deadline Scheduling

- There are other types of deadline schedulers
  - Example: a Least Slack algorithm chooses the task with the smallest slack = time until deadline – remaining execution time
  - i.e. slack is the maximum amount of time that a task can be delayed without missing its deadline
    - Tasks with the least slack are those that have the least flexibility to be delayed given the amount of remaining computation needed before their deadline expires
- Both EDF and Least Slack are optimal according to different criteria

Soft Real Time Systems

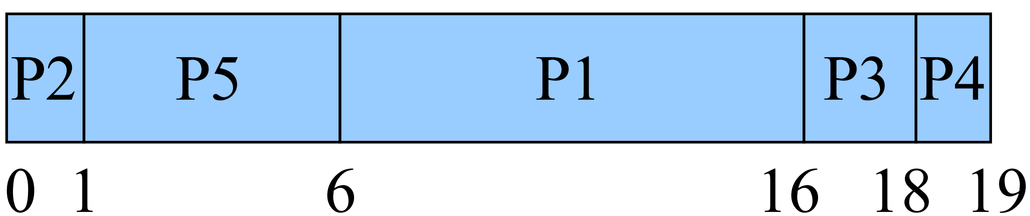
- *Soft* real time systems seek to meet most deadlines, but allow some to be missed
  - Unlike hard real time systems, where every deadline must be met or else the system fails
  - Soft real time scheduler may seek to provide probabilistic guarantees
    - e.g. if 60% of deadlines are met, that may be sufficient for some systems
  - Linux supports a soft real-time scheduler based on priorities – we'll see this next

Priority-based Scheduling

- Assign each task a priority, and schedule higher priority tasks first, before lower priority tasks
- Any criteria can be used to decide on a priority
  - measurable characteristics of the task
  - external criteria based on the “importance” of the task
  - Example: foreground processes may get high priority, while background processes get low priority

Priority-based Scheduling

Process	CPU Execution Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



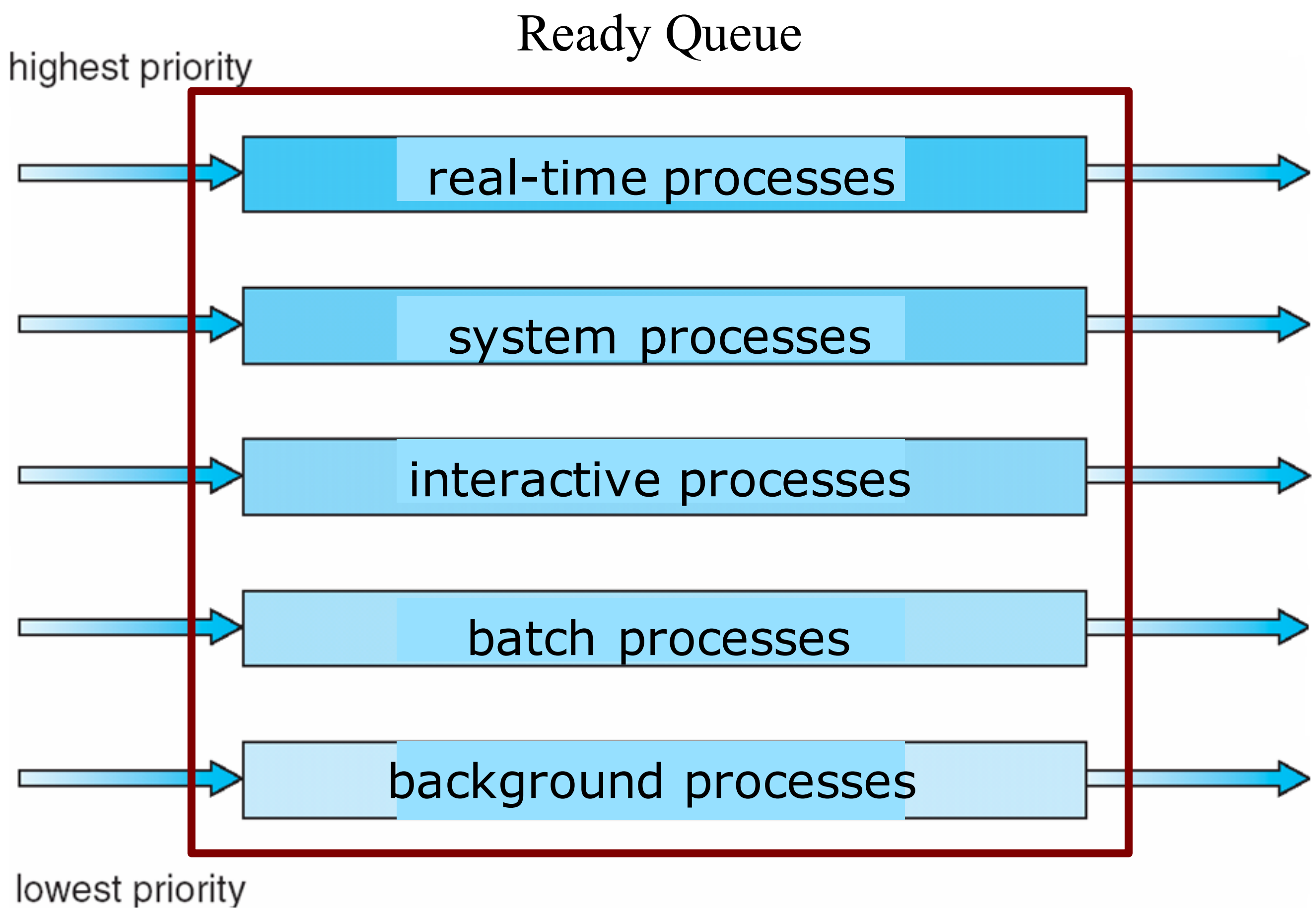
Priority-based Scheduling

- Can be preemptive:
  - A higher priority process arriving in the ready queue can preempt a lower priority running process
  - Can occur if the lower priority process …:
    - Yields CPU with a system call
    - Is interrupted by a timer interrupt
    - Is interrupted by a hardware interrupt
  - Each of these cases gives control back to the OS, which can then schedule the higher priority process

Priority-based Scheduling

- Multiple tasks with the same priority are scheduled according to some policy
  - FCFS, round robin, etc.
- Each priority level has a set of tasks, forming a *multi-level queue*
  - Each level's queue can have its own scheduling policy
- We use priority-based scheduling and multi-level queue scheduling interchangeably

Multilevel Queue Scheduling

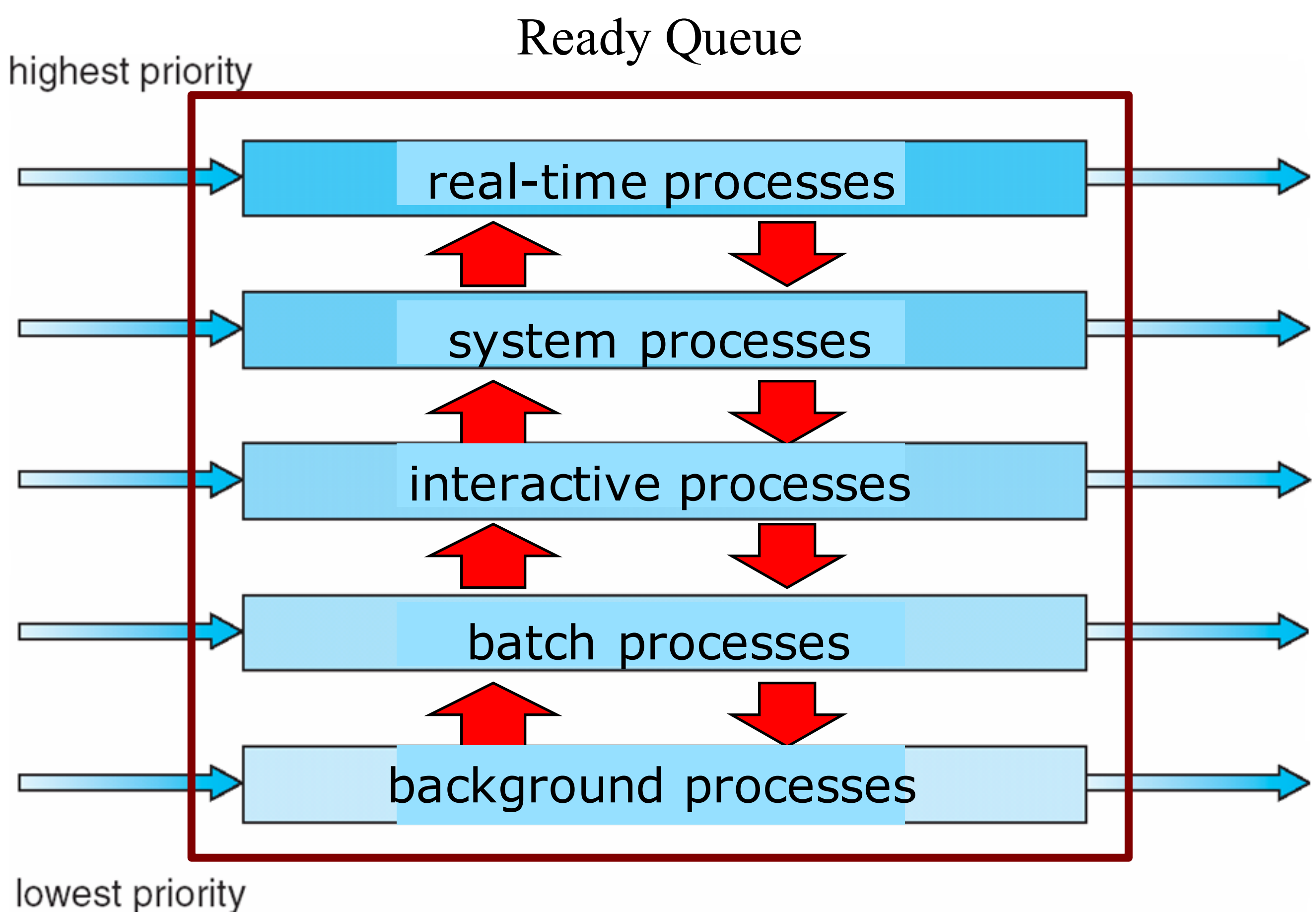


(modified)

Priority-based Scheduling

- Preemptive priorities can starve low priority processes
  - A higher priority task always gets served ahead of a lower priority task, which never sees the CPU
- Some starvation-free solutions:
  - Assign each priority level a proportion of time, with higher proportions for higher priorities, and rotate among the levels
    - Similar to weighted round robin, except across levels
  - Create a *multi-level feedback queue* that allows a task to move up/down in priority
    - Avoids starvation of low priority tasks

Multilevel Feedback Queue Scheduling



(modified)

Multilevel Feedback Queue

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Multi-level Feedback Queues

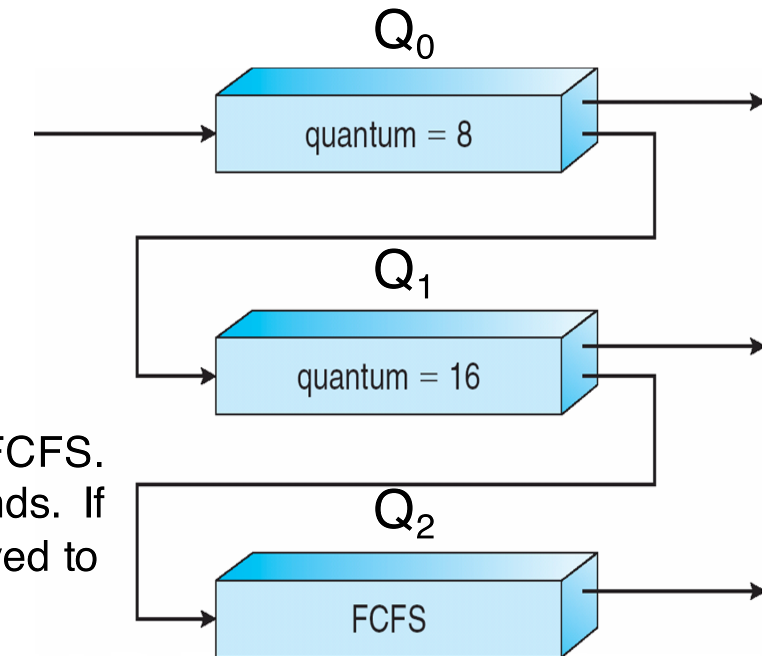
- Criteria for process movement among priority queues could depend upon age of a process:
  - old processes move to higher priority queues, or conversely, high priority processes are eventually demoted
  - sample aging policy: if priorities range from 1-128, can decrease (increment) the priority by 1 every T seconds
  - eventually, the low priority process will get scheduled on the CPU

Multi-level Feedback Queues

- Criteria for process movement among priority queues could depend upon behavior of a process:
  - could be CPU-bound processes move down the hierarchy of queues, allowing interactive and I/O-bound processes to move up
  - give a time slice to each queue, with smaller time slices higher up
  - if a process doesn’t finish by its time slice, it is moved down to the next lowest queue
  - over time, a process gravitates towards the time slice that typically describes its average local CPU burst

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.
  - Interactive processes are more likely to finish early, processing a small amount of data, while compute-bound processes will exhaust their time slice. So interactive processes will gravitate towards higher priority queues.



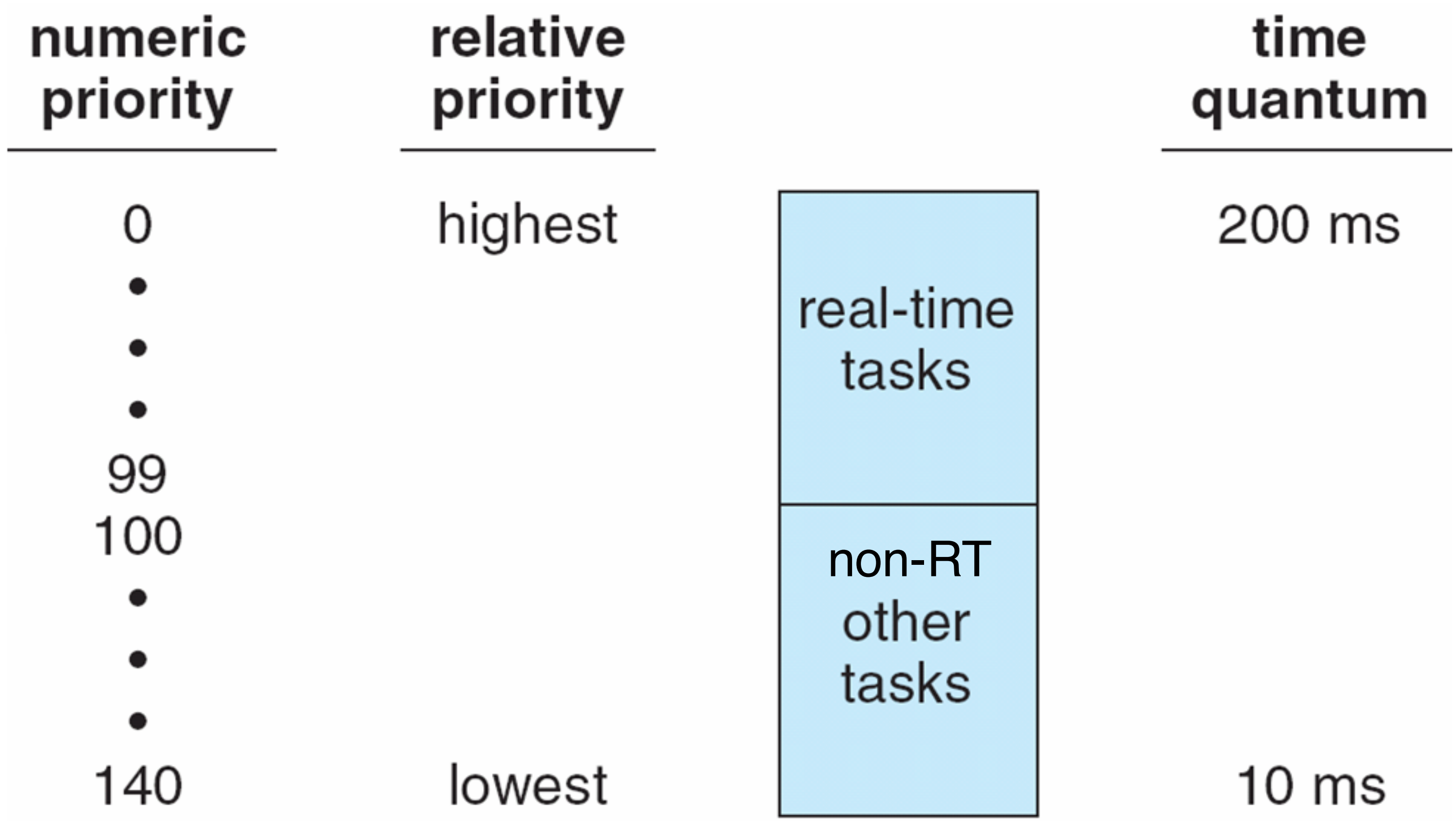
Priority-based Scheduling

- In Unix/Linux, you can *nice* a process to set its priority, within limits
  - e.g. priorities can range from -20 to +20, with lower values giving higher priority, a process with 'nice +15' is "nicer" to other processes by incrementing its value (which lowers its priority)
    - E.g. if you want to run a compute-intensive process compute.exe with low priority, you might type at the command line "nice –n 19 compute.exe"
  - To lower the niceness, hence increase priority, you typically have to be root
  - Different schedulers will interpret/use the nice value in their own ways

Multi-level Feedback Queues

- In Windows XP and Linux, system & real-time tasks are grouped in a priority range that is higher in priority than the priority range of non-real-time tasks
  - XP has 32 priorities. 1-15 are for normal processes, 16-31 are for real-time processes. One queue for each priority.
    - XP scheduler traverses queues from high priority to low priority until it finds a process to run
  - In Linux, priorities 0-99 are for important/real-time processes while 100-139 are for 'nice' user processes. Lower values mean higher priorities.
    - Also, longer time quanta for higher priority tasks (200 ms for highest) and shorter time quanta for lower priority tasks (10 ms for lowest).

Linux Priorities and Time-slice length



Multi-level Feedback Queues

- Most modern OSs use or have used multi-level feedback queues for priority-based preemptive scheduling
  - e.g. Windows NT/XP, Mac OS X, FreeBSD/NetBSD and Linux pre-2.6
  - Linux 1.2 used a simple round robin scheduler
  - Linux 2.2 introduced scheduling classes (priorities) for real-time and non-real-time processes and SMP (symmetric multi-processing) support
  - Linux 2.4 and above next lecture…