slides kindly provided by:

Tam Vu, Ph.D
Professor of Computer Science
Director, Mobile & Networked Systems Lab
Department of Computer Science
University of Colorado Boulder

CSCI 3753 Operating Systems Summer 2020

Christopher Godley

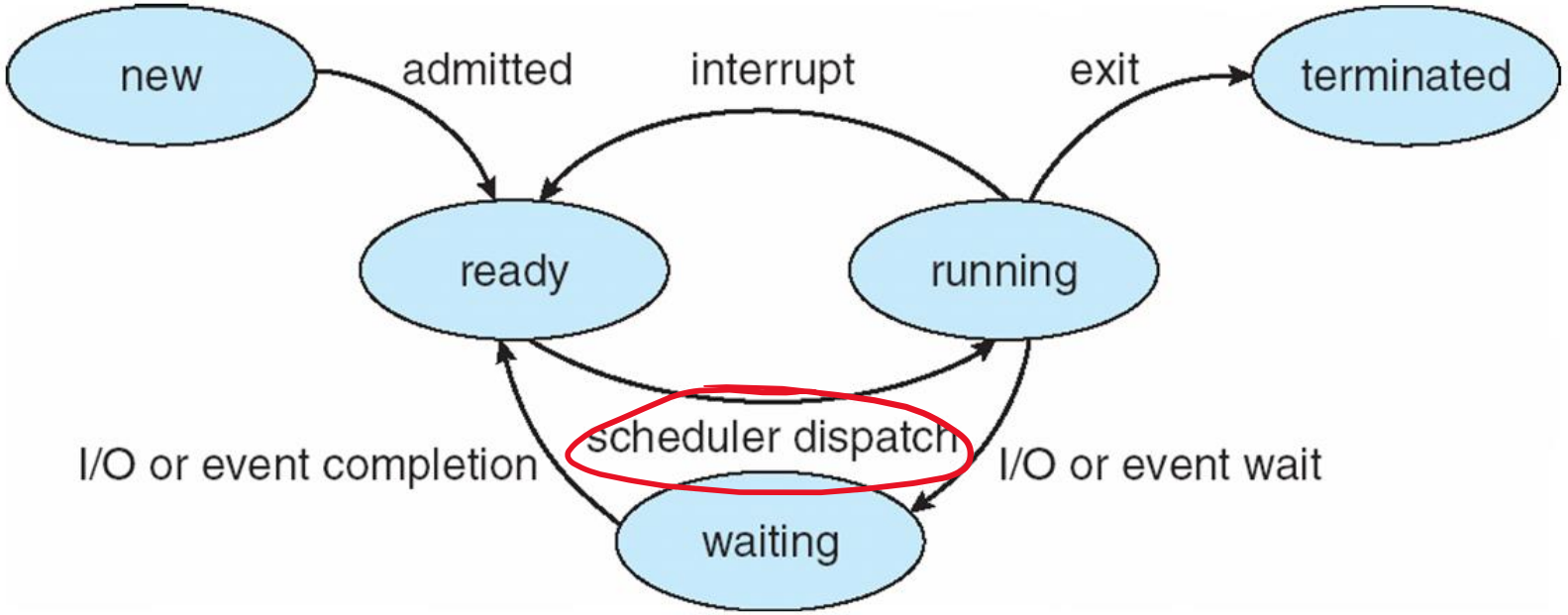
PhD Student

Department of Computer Science

University of Colorado Boulder

Process Scheduling

Diagram of Process State



Also called “blocked” state

Switching Between Processes

- A process can be switched out due to:
  - blocking on I/O
  - voluntarily yielding the CPU, e.g. via other system calls
  - being preemptively time sliced, i.e interrupted
  - Termination

Switching Between Processes

- The dispatcher gives control of CPU to the process selected by the scheduler, causing context switch:
  - save old state
  - select next process
  - load new state
  - switch to user mode, jumping to the proper location in the user process to restart that process

Mechanism

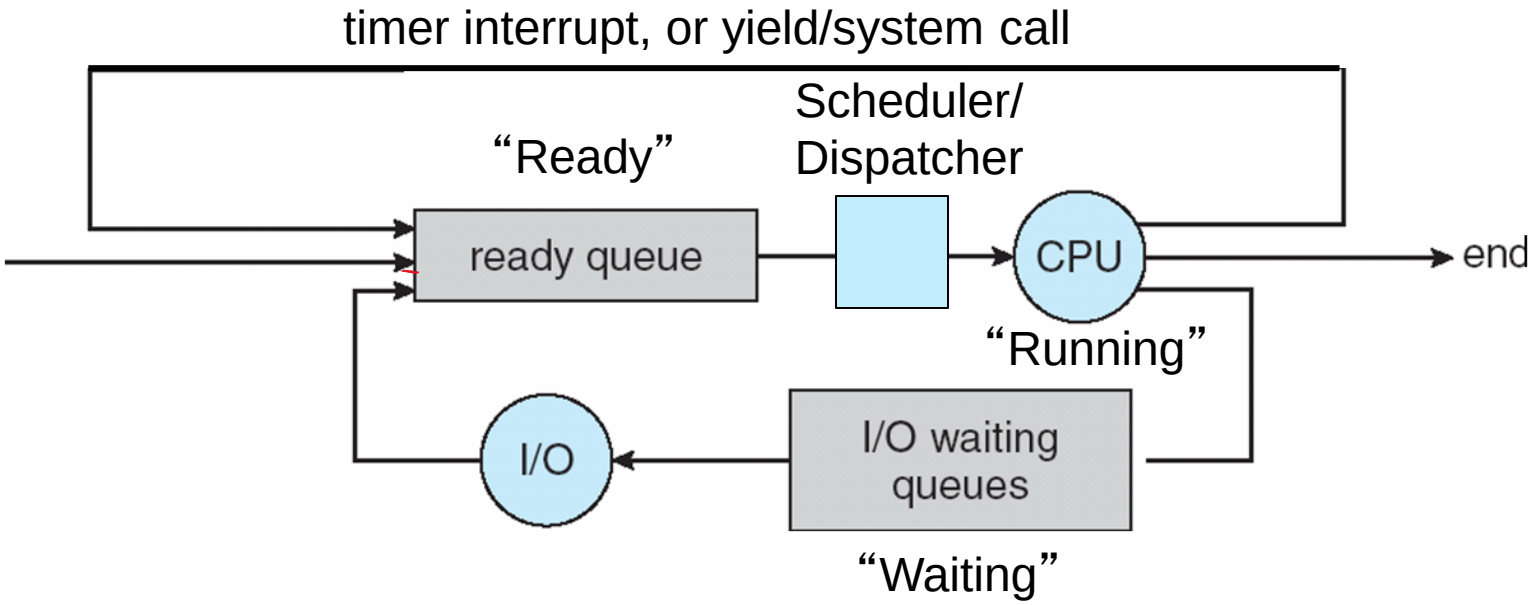
Policy

- Separate
  - *mechanism* of scheduling
  - from the *policy* of scheduling

Context Switch Overhead

- Typically take 10 microseconds to copy register state to/from memory
  - on a 1 GHz CPU, that's 10000 wasted cycles per context switch!
- if the time slice is on the order of a context switch, then CPU spends most of its time context switching
  - Typically choose time slice to be large enough so that only 10% of CPU time is spent context switching
  - Most modern systems choose time slices of 100 us

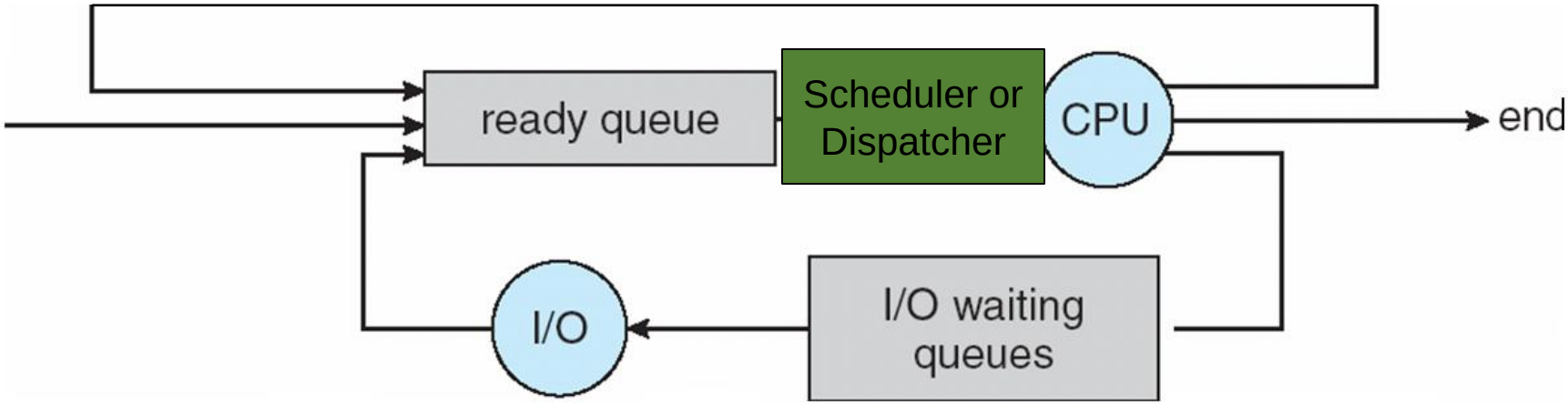
Process Scheduling



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

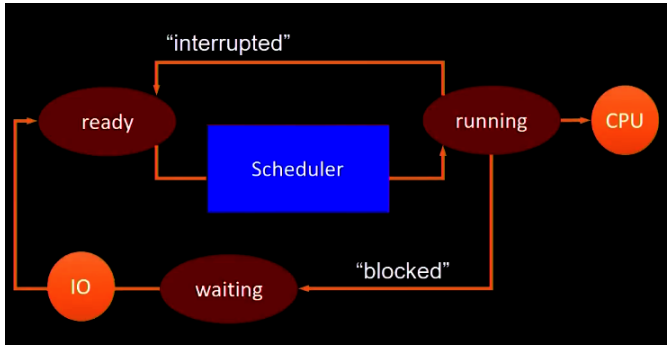
Scheduling Policy

- Scheduler's job is to decide the next process (or kernel thread) to run
  - From among the set of processes/kernel threads in the ready queue

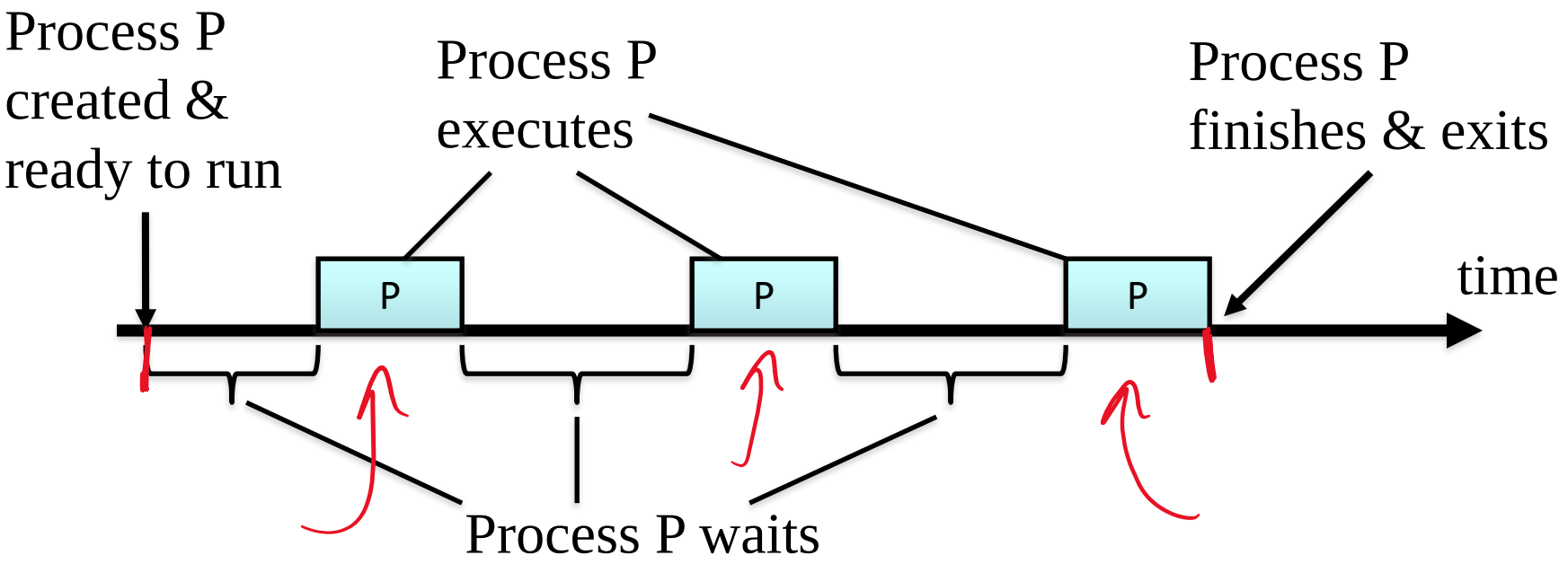


Scheduling Policy

- Scheduler implements a scheduling policy based on some of the following **goals**:
  - maximize CPU utilization: 40 % to 90 %
  - maximize throughput: # processes completed/second
  - maximize fairness
  - Meet deadlines or delay guarantees
  - Ensure adherence to priorities
  - Minimize average or peak turnaround time: from 1st entry to termination
  - Min avg or peak waiting time: sum of time in ready queue
  - Min avg or peak response time: time until first response.

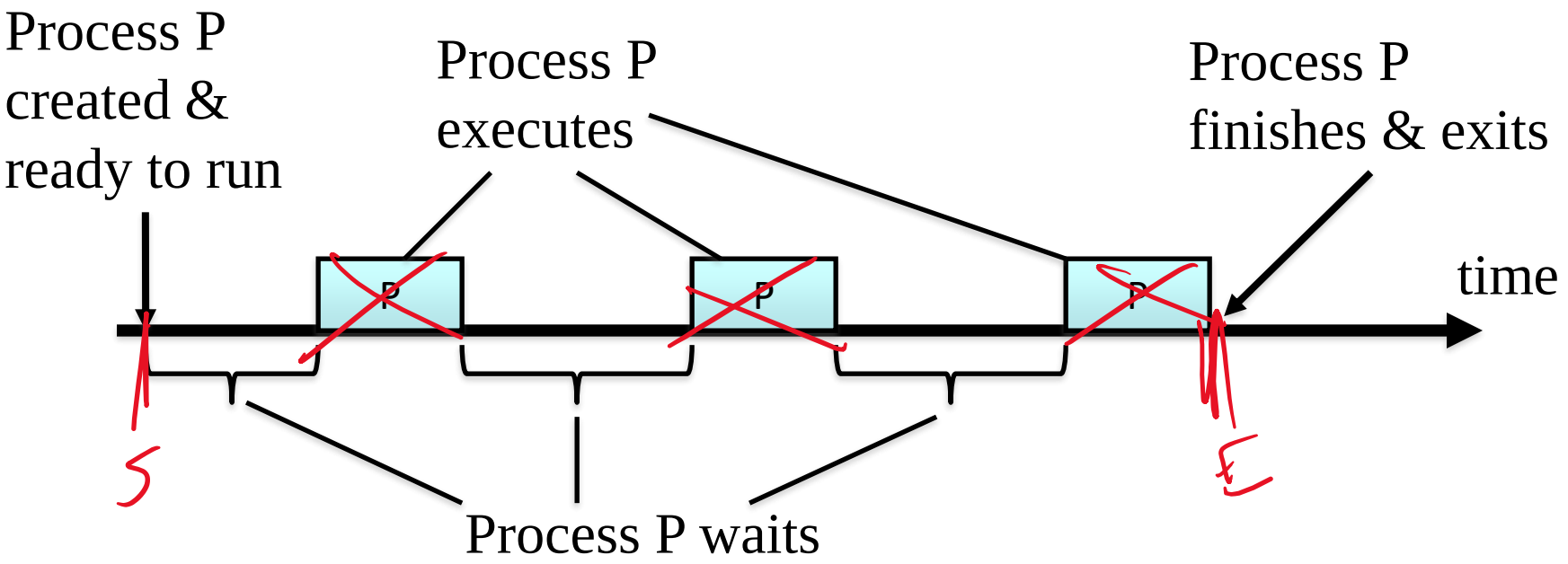


Scheduling Definitions



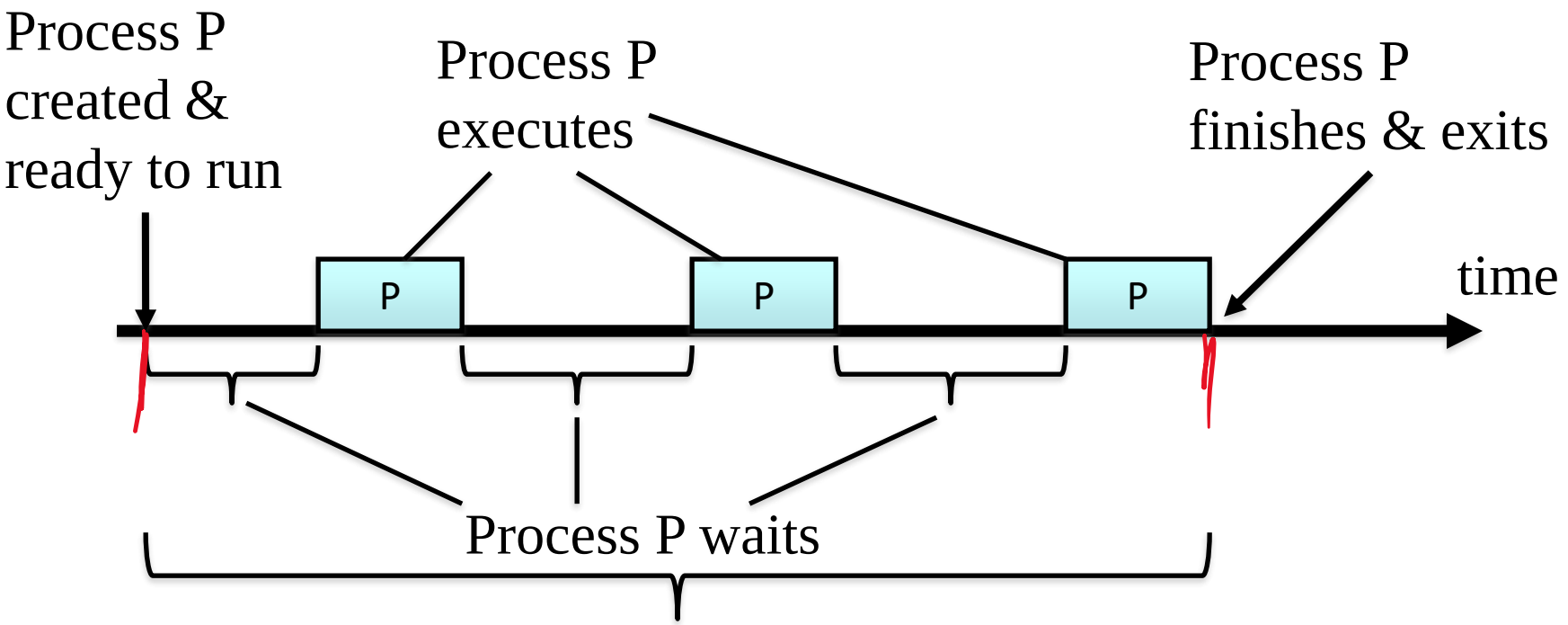
- *execution time* $E(P_i)$ = the time on the CPU required to fully execute process i
  - Sum up the time slices given to process i
  - Also called the "burst time" by textbook

Scheduling Definitions



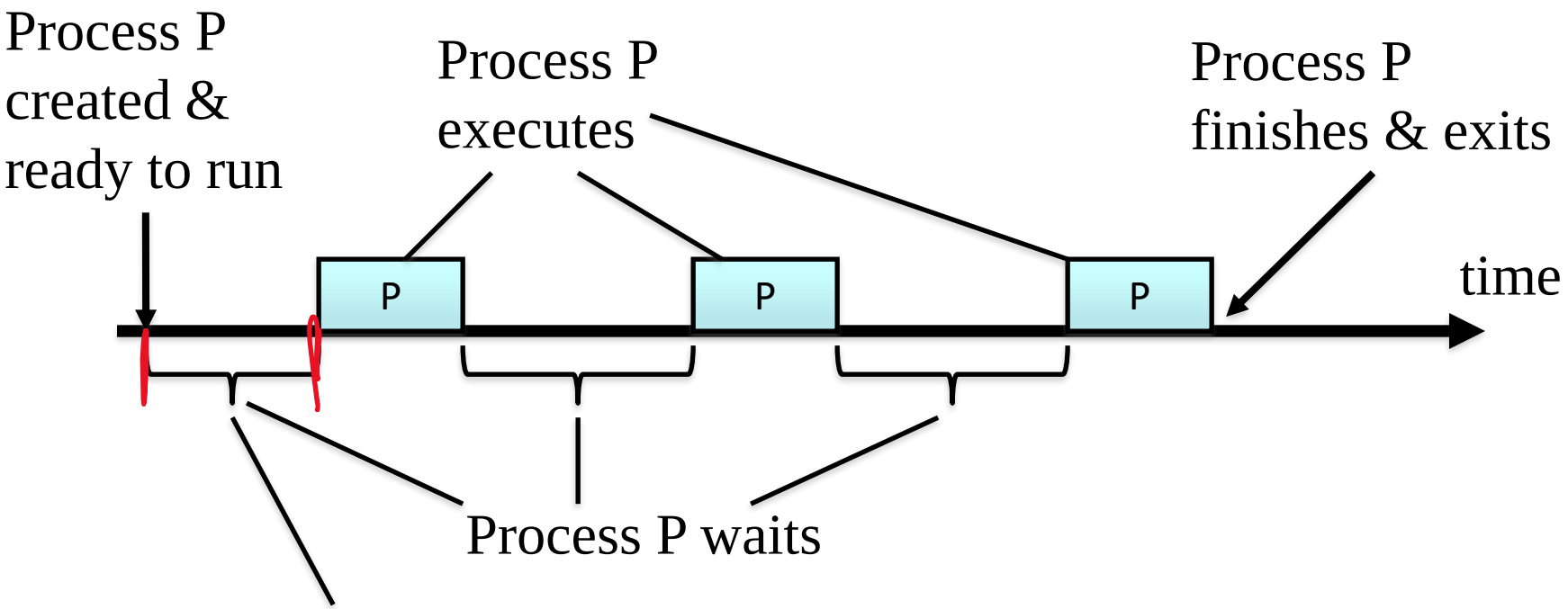
- ***wait time*** $W(P_i)$ = the time process i is in the ready state/queue waiting but not running
  - Sum up the gaps between time slices given to process *i*
  - But, does NOT include I/O waiting time

Scheduling Definitions



- ***turnaround time*** $T(P_i)$ the time from 1st entry of process i into the ready queue to its final exit from the system (exits last run state)
  - Does include time waiting and time for IO to complete

Scheduling Definitions



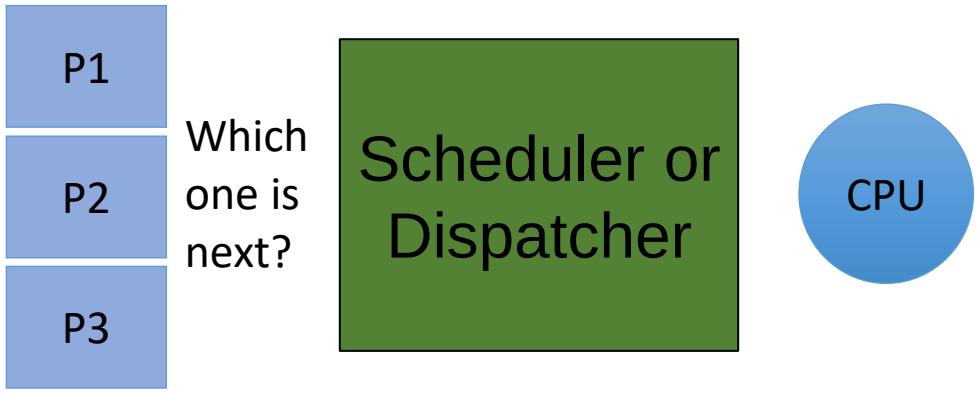
- ***response time*** $R(P_i)$ = the time from 1st entry of process i into the ready queue to its 1st scheduling on the CPU (1st occurrence in running state)
  - Useful for interactive tasks

Scheduling Analysis

- We analyze various scheduling policies to see how efficiently they perform with respect to metrics
  - wait time, turnaround time, response time, etc.
- Some algorithms will be optimal in certain metrics
- To simplify our analysis, assume:
  - No blocking I/O. Focus only on scheduling processes/tasks that have provided their execution times
  - Processes execute until completion, unless otherwise noted

Scheduling Policy

- How does Scheduler pick the next process to be run?
  - depends on which policy is implemented
- What is the simplest policy you can think of for picking from a group of processes?
- How about something complicated?

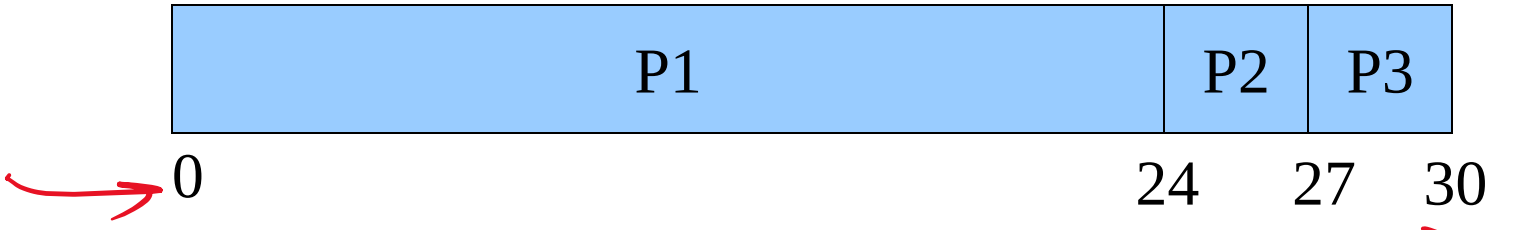


First Come First Serve (FCFS) Scheduling

- order of arrival dictates order of scheduling
  - Non-preemptive, processes execute until completion

- If processes arrived in order P1, P2, P3 before time 0, then CPU service time is:

Process	CPU Execution Time
P1	24
P2	3
P3	3



FCFS Scheduling

- If processes arrive in reverse order P3, P2, P1 before time 0, then CPU service time is:

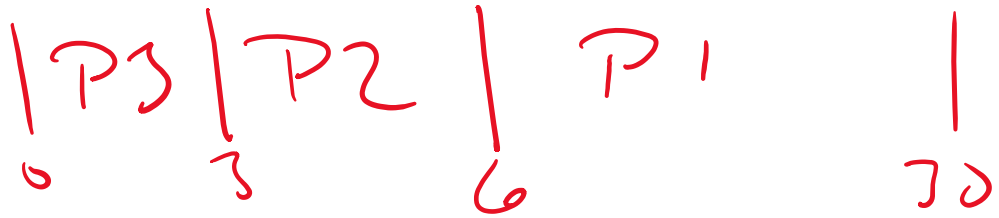
Process	CPU Execution Time
P1	24
P2	3
P3	3

Gantt Chart

- The chart we have been using to visualize the sequence of events and time allocated to processes is call a Gantt Chart
- We used this same concept in our first descriptions of scheduling using multi-process and multi-tasking

Process	CPU Execution Time
P1	24
P2	3
P3	3

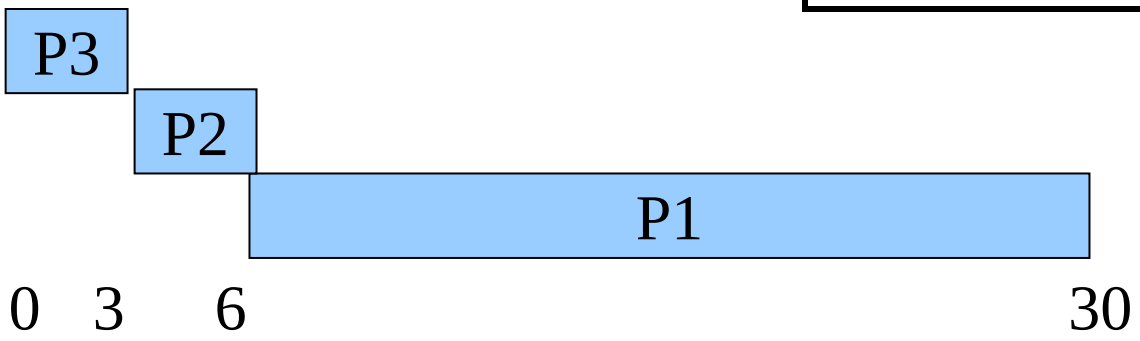
Gantt Chart



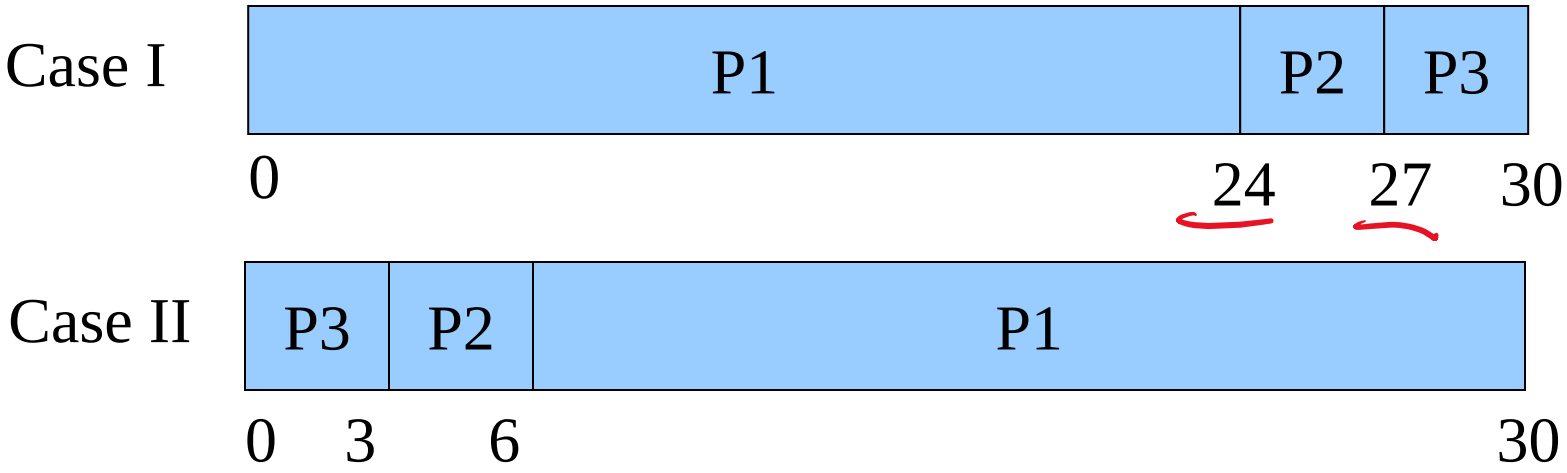
- We used a different format to describe the process times of multi-tasking and multi-programming

- These formats are equivalent

Process	CPU Execution Time
P1	24
P2	3
P3	3



FCFS Scheduling



Lets calculate the ***average wait time*** for each of the cases

All processes arrived just before time 0

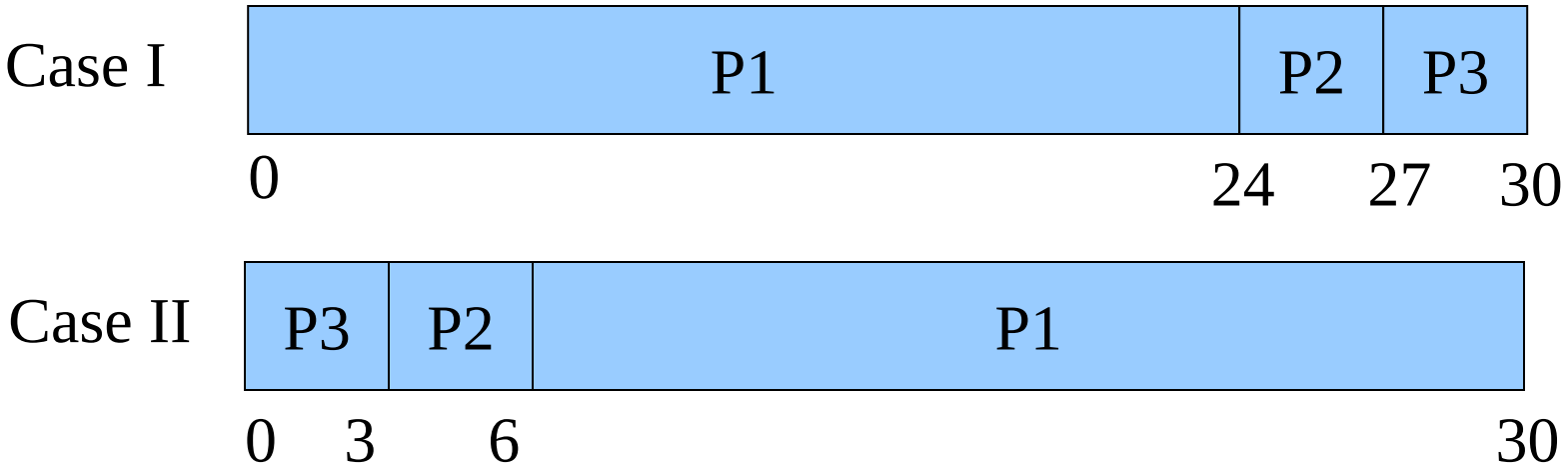
P1 does not wait at all

P2 waits for P1 to complete before being scheduled

P3 waits until P2 has completed

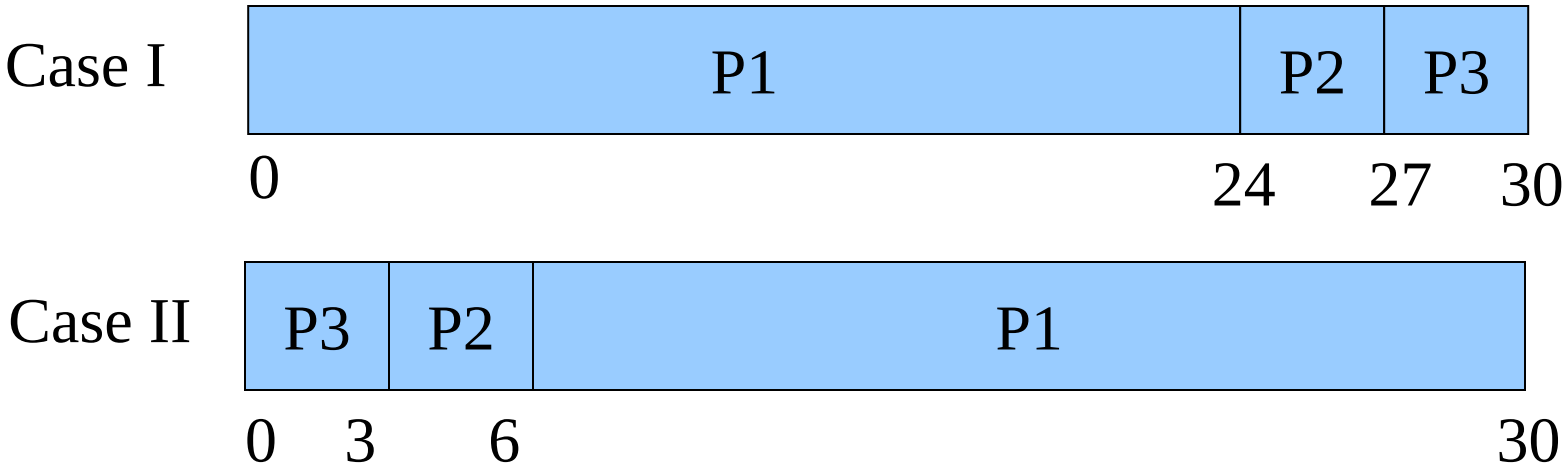
(0 + 24 + 27)/3 = 17

FCFS Scheduling



- Case I: **average wait time** is (0+24+27)/3 = 17 seconds
- Case II: **average wait time** is (0+3+6)/3 = 3 seconds
- FCFS wait times are generally not minimal - vary a lot if order of arrival changed, which is especially true if the process service times vary a lot (are spread out)

FCFS Scheduling



Lets calculate the ***average turnaround time*** for each of the cases

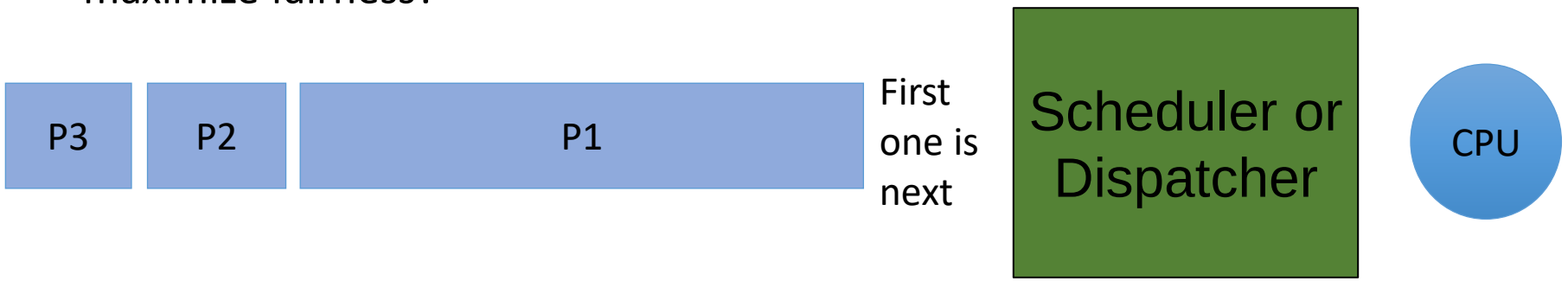
- Case I: average turnaround time is (24+27+30)/3 = 27 seconds
- Case II: average turnaround time is (3+6+30)/3 = 13 seconds

- A lot of variation in turnaround time too, depending on the task's arrival.

FCFS Scheduling

- Just pick the next process in the queue to be run?
  - No other information about the process is required

- Does it meet our goals?
  - maximize CPU utilization: 40% to 90%?
  - maximize throughput: # processes completed/second?
  - minimize average or peak wait, turnaround, or response time?
  - meet deadlines or delay guarantees?
  - maximize fairness?

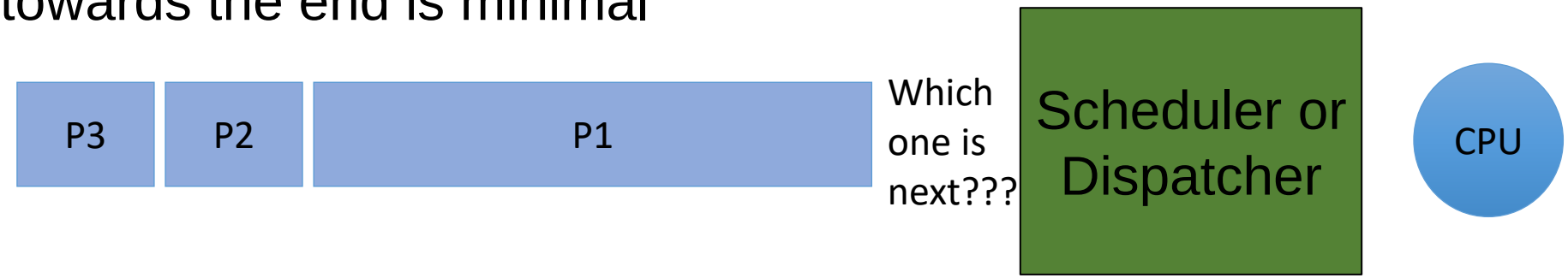


Scheduling Policies

Shortest Job First (SJF) Scheduling

Choose the process/thread with
the lowest execution time

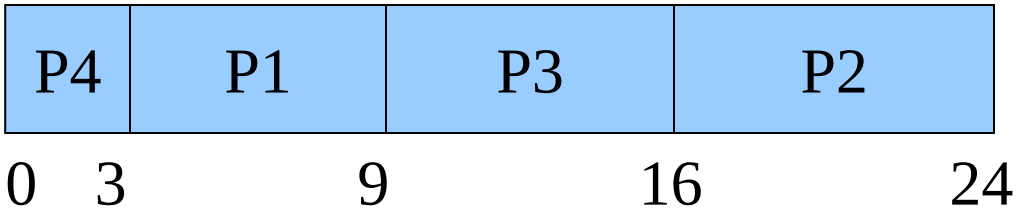
- gives priority to shortest or briefest processes
- minimizes the average wait time
  - intuition: one long process will increase wait time for all short processes that follow
- the impact of the wait time on other long processes moved towards the end is minimal



Shortest Job First Scheduling

- In this example, P1 through P4 are in ready queue at time 0:
  - *can prove SJF minimizes wait time*
  - out of 24 possibilities of ordering P1 through P4, the SJF ordering has the lowest average wait time

Process	CPU Execution Time
P1	6
P2	8
P3	7
P4	3



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

Shortest Job First (SJF) Scheduling

- ***It has been proved that SJF minimizes the average wait time out of all possible scheduling policies.***
- ***Sketch of proof:***
  - Given a set of processes {P_a, P_b, …, P_n}, suppose one chooses a process P from this set to schedule first
  - The wait times for all the remaining processes in {P_a, …, P_n}-P will be increased by the run time of P
  - If P has the shortest run time (SJF), then the wait times will increase the least amount possible

Shortest Job First (SJF) Scheduling

- ***Sketch of proof (continued)***:
  - Apply this reasoning iteratively to each remaining subset of processes.
    - At each step, the wait time of the remaining processes is increased least by scheduling the process with the smallest run time.
  - The average wait time is minimized by minimizing each process' wait time,
  - Each process' wait time is the sum of all earlier run times, which is minimal if the shortest job is chosen at each step above.

Shortest Job First Scheduling

- **Problem?**
  - must know run times $E(p_i)$ in advance unlike FCFS

- **Solution: estimate CPU demand in the next time interval from the process/thread's CPU usage in prior time intervals**

  - Divide time into monitoring intervals, and in each interval n, measure the CPU time each process P_i takes as CPU(n,i).

  - For each process P_i, estimate the amount of CPU time EstCPU(n,i) for the next interval as the average of the current measurement and the previous estimate

Shortest Job First Scheduling

- **Solution (continued):**

  EstCPU(n+1,i) = α*CPU(n,i) + (1-α)*EstCPU(n,i)
  where $0<\alpha<1$

  - If $\alpha>1/2$, then estimate is influenced more by recent history.
  - If $\alpha<1/2$, then bias the estimate more towards older history
  - This kind of average is called an **exponentially weighted average**
  - See textbook for more

Shortest Job First Scheduling

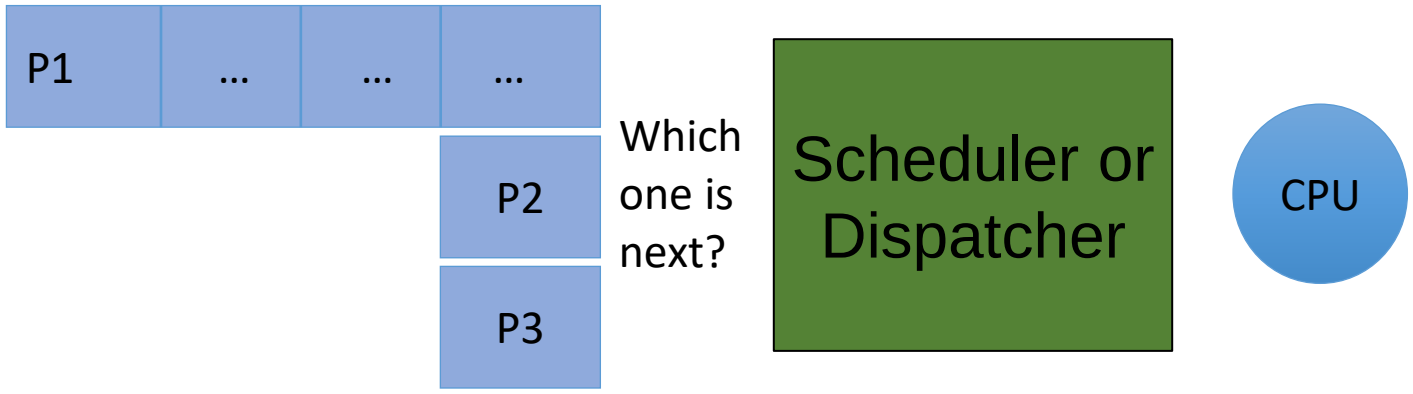
- **SJF can be preemptive:**
  - i.e. when a new job arrives in the ready queue, if its execution time is less than the currently executing job's remaining execution time, then it can preempt the current job
  - Compare to FCFS: a new process can't preempt earlier processes, because its order is later than the earlier processes
  - For simplicity, we assumed in the preceding analysis that jobs ran to completion and no new jobs arrived until the current set had finished

Scheduling Criteria

Is it FAIR that long processes use the CPU as much as they need?

Is it FAIR to make long processes wait for all shorter processes?

How can be we more fair with the CPU resource?



Round Robin Scheduling

- **Use preemptive time slicing**
  - a task is forced to relinquish the CPU before it's necessarily done

- **Rotate among the tasks in the ready queue.**
  - 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, …

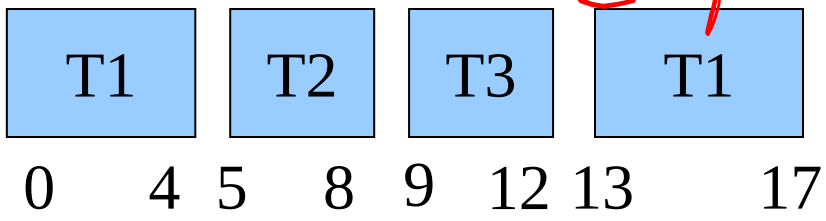
  - treats the ready queue as a circular queue (or removes the scheduled item from the front and places it at the back)

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?***
  - assuming a 1ms switching time

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

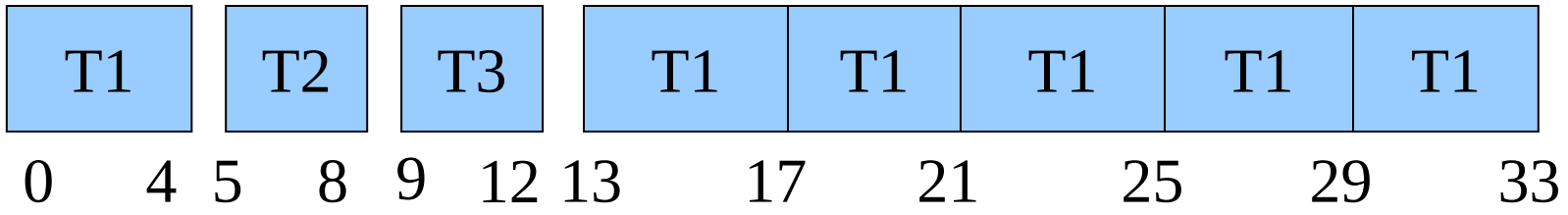
sched oppt. @ time slice
@ process termination



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 3.66ms*
  - assuming a 1ms switching time
  - 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

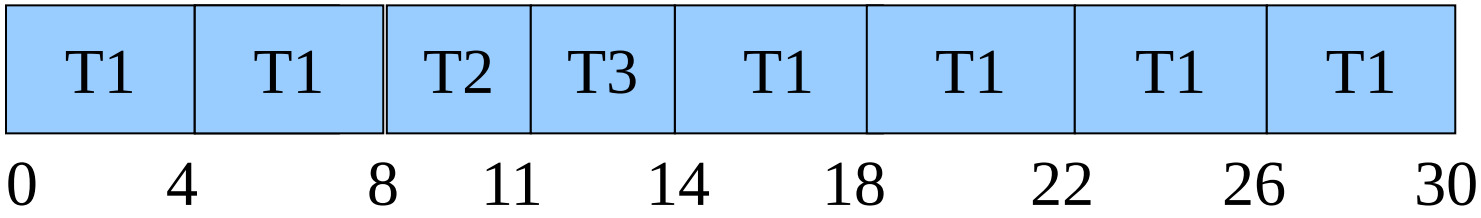
Weighted Round Robin

- **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:

$$\alpha_i = \frac{N_i}{\sum_i N_i}$$

Weighted Round Robin

- In previous example – assuming switching time = 0ms:
  - could give T1 two time slices
  - T2 and T3 only 1 each round



Deadline Scheduling

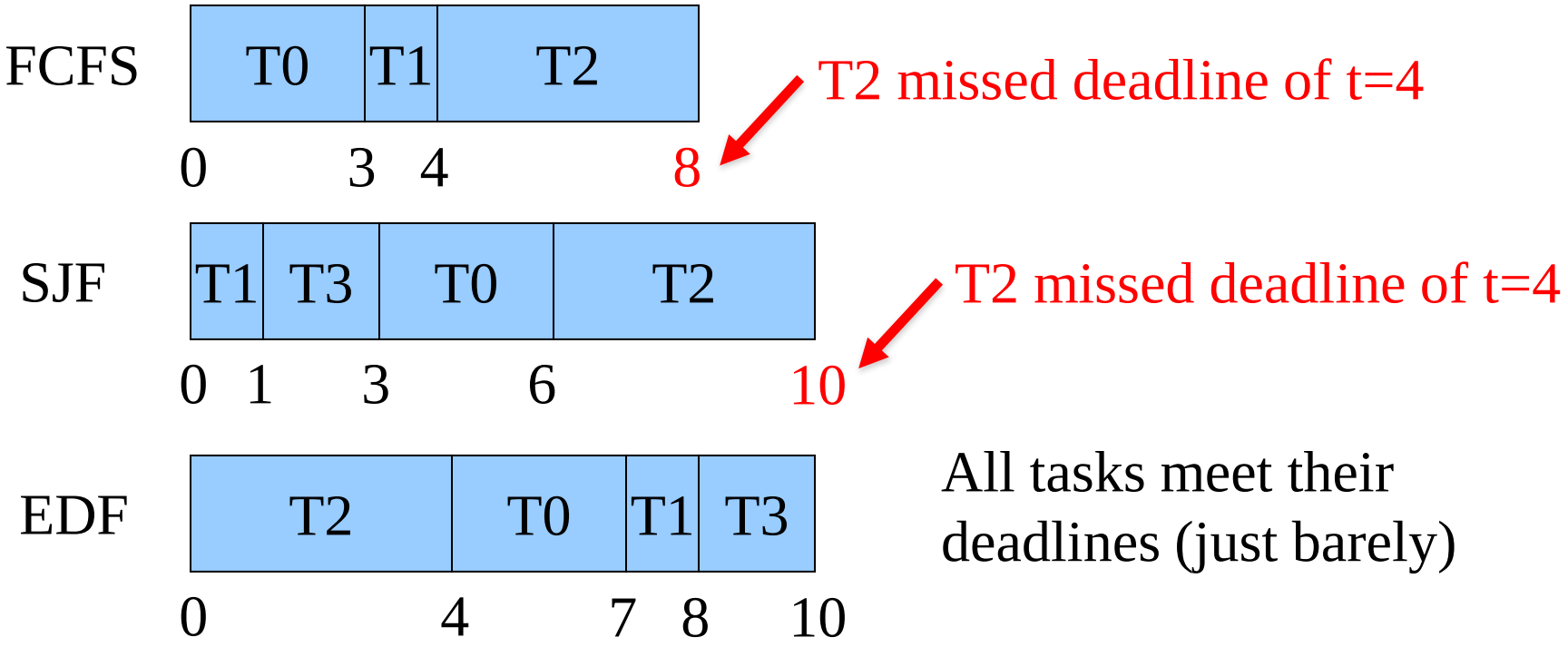
- Hard real time systems require that certain tasks *must* finish executing by a certain time, or the system fails
  - e.g. robots and self-driving cars need a real time OS (RTOS) whose tasks (actuating an arm/leg or steering wheel) must be scheduled by a certain deadline

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**
  - Pick task that most urgently needs to be completed

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



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

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

- **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 (accepting new tasks) 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 when deadlines cannot be met with new task,
      deny admission to this task if its deadline can't be met
  - Note FCFS, SJF and priority/weighted RRobin 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.

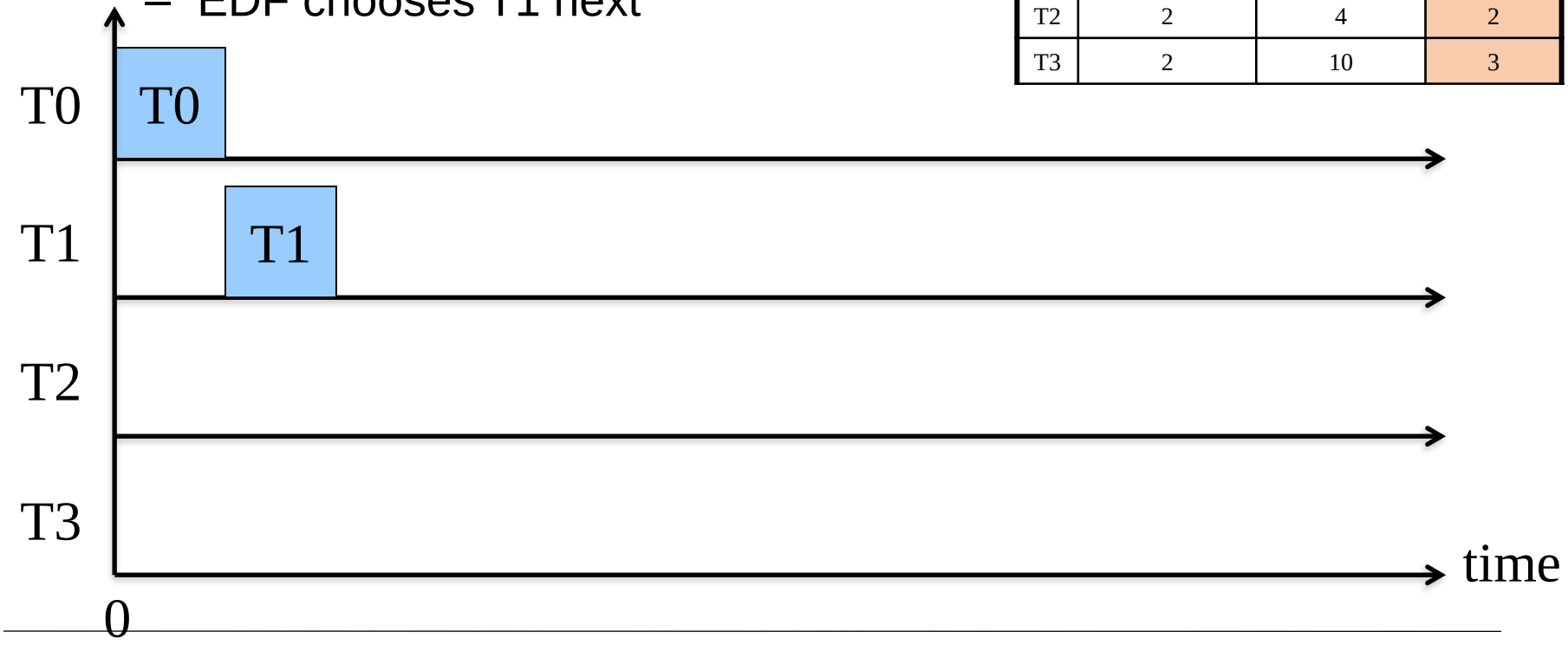
Task	CPU Execution Time	Absolute Deadline	Arrival time
T0	1	2	0
T1	2	5	0
T2	2	4	2
T3	2	10	3

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

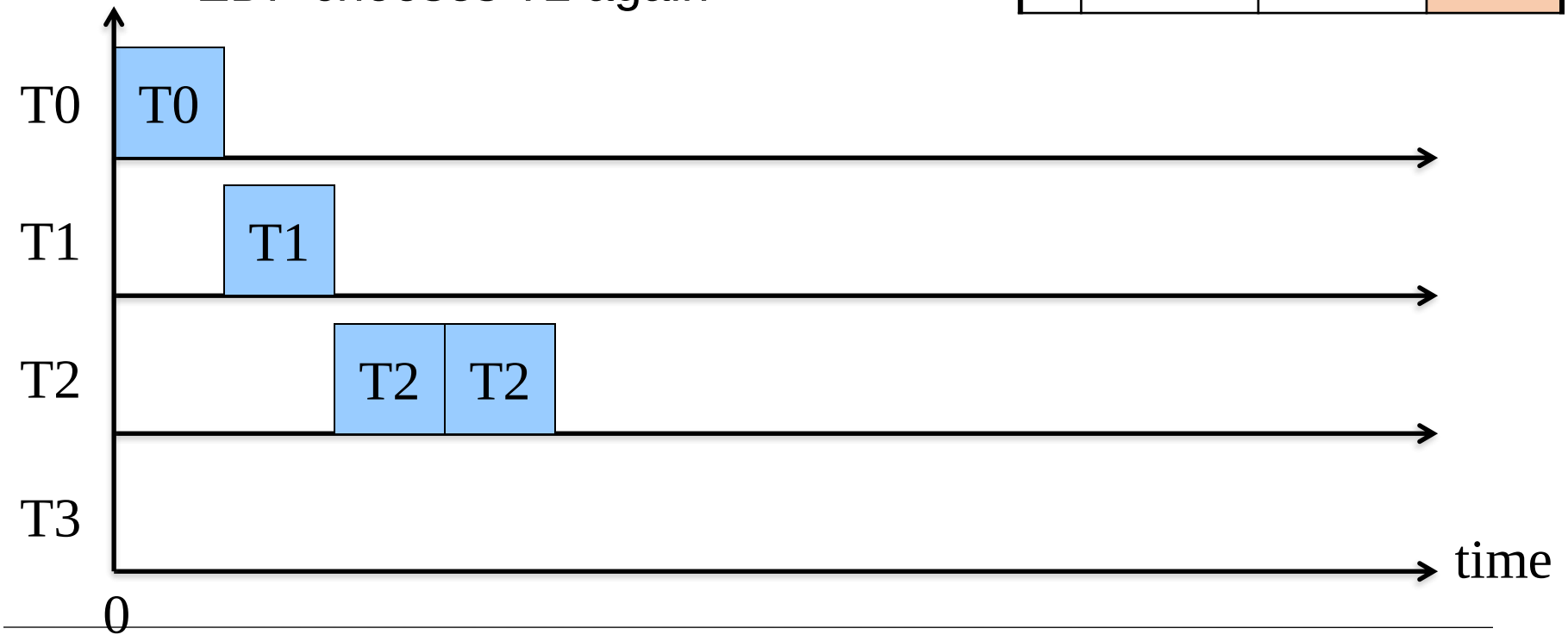
Task	CPU Execution Time	Absolute Deadline	Arrival time
T0	1	2	0
T1	2	5	0
T2	2	4	2
T3	2	10	3



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

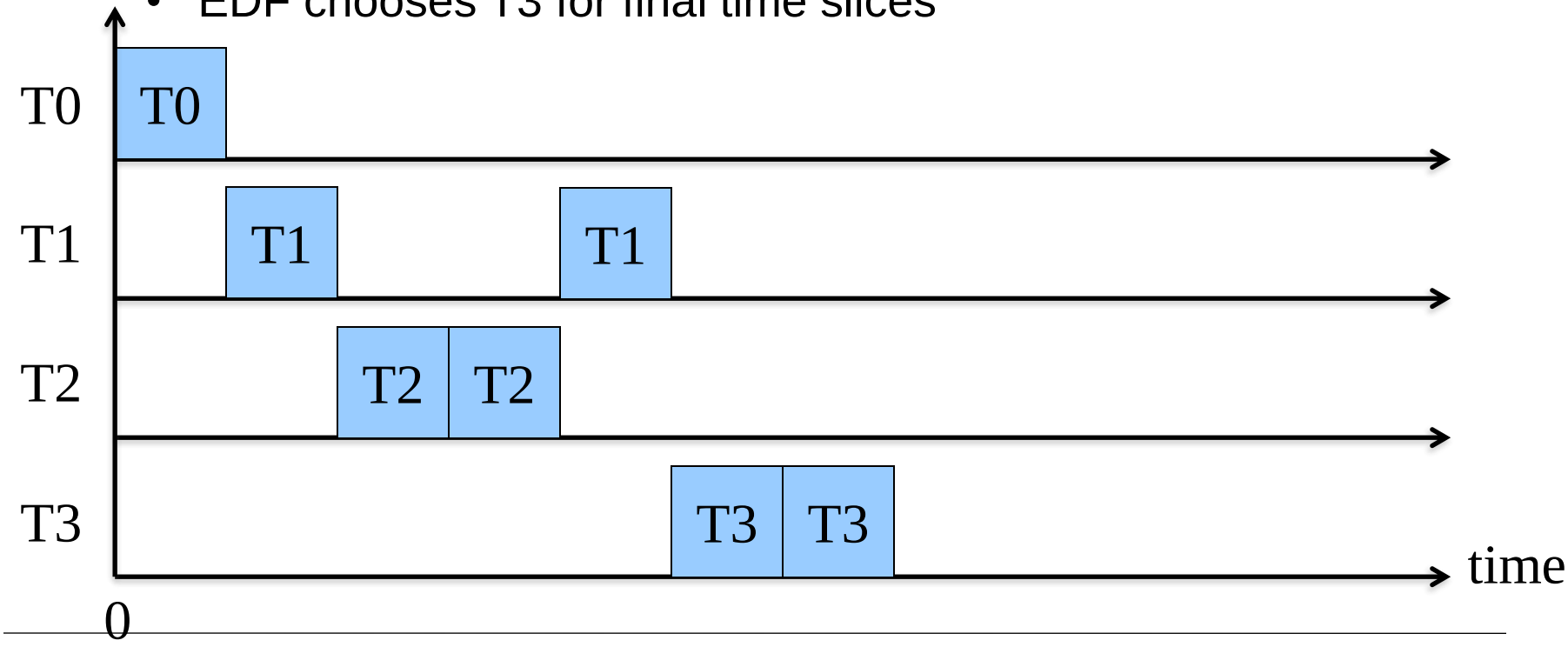
Task	CPU Execution Time	Absolute Deadline	Arrival time
T0	1	2	0
T1	2	5	0
T2	2	4	2
T3	2	10	3



EDF and Preemption

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

- 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 = 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

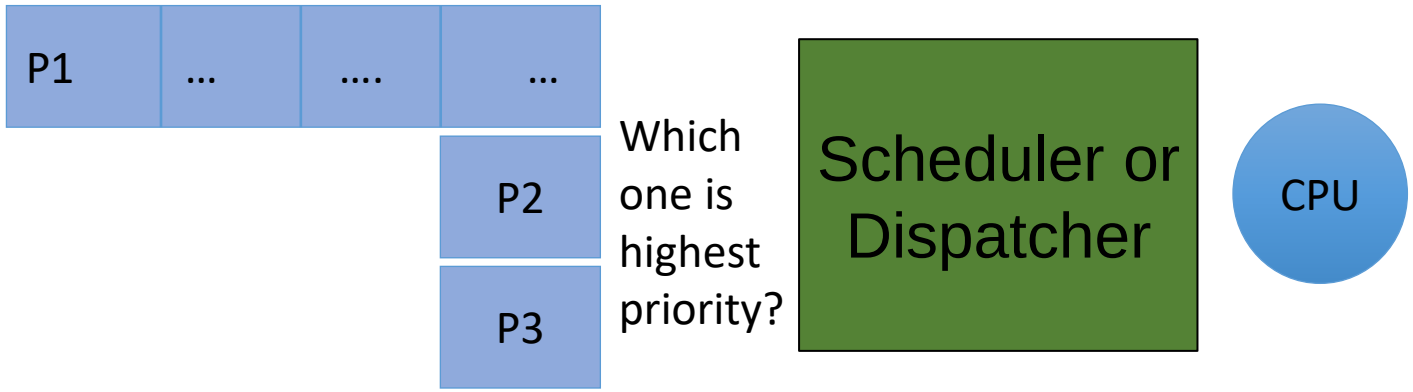
- ***Soft* real time systems seek to meet most deadlines, but allow some to be missed**
  - Unlike hard real time systems, where every deadline must be met or else the system fails
  - Soft real time scheduler may seek to provide probabilistic guarantees
    - e.g. if 60% of deadlines are met, that may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

More Scheduling Policies

Priority-based Scheduling

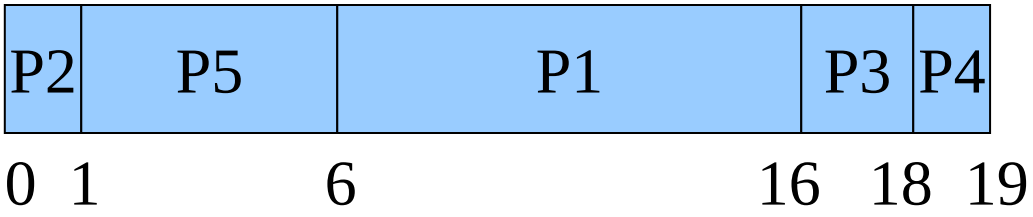
- **Assign each task a priority, and schedule higher priority tasks first, before lower priority tasks**

- **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



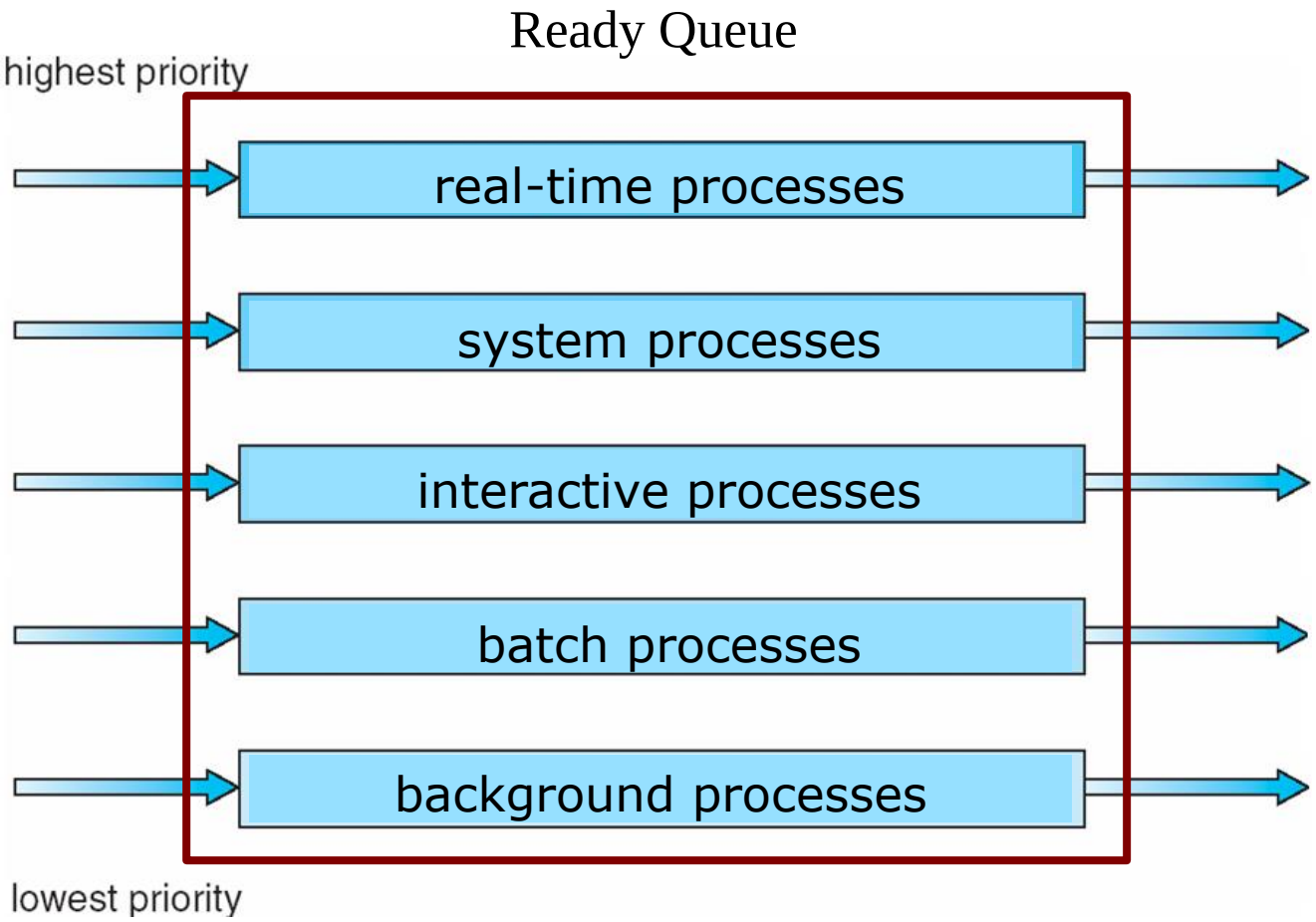
Priority-based Scheduling

- Can be preemptive:
  - A higher priority process arriving in the ready queue can preempt a lower priority running process
  - Switch 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

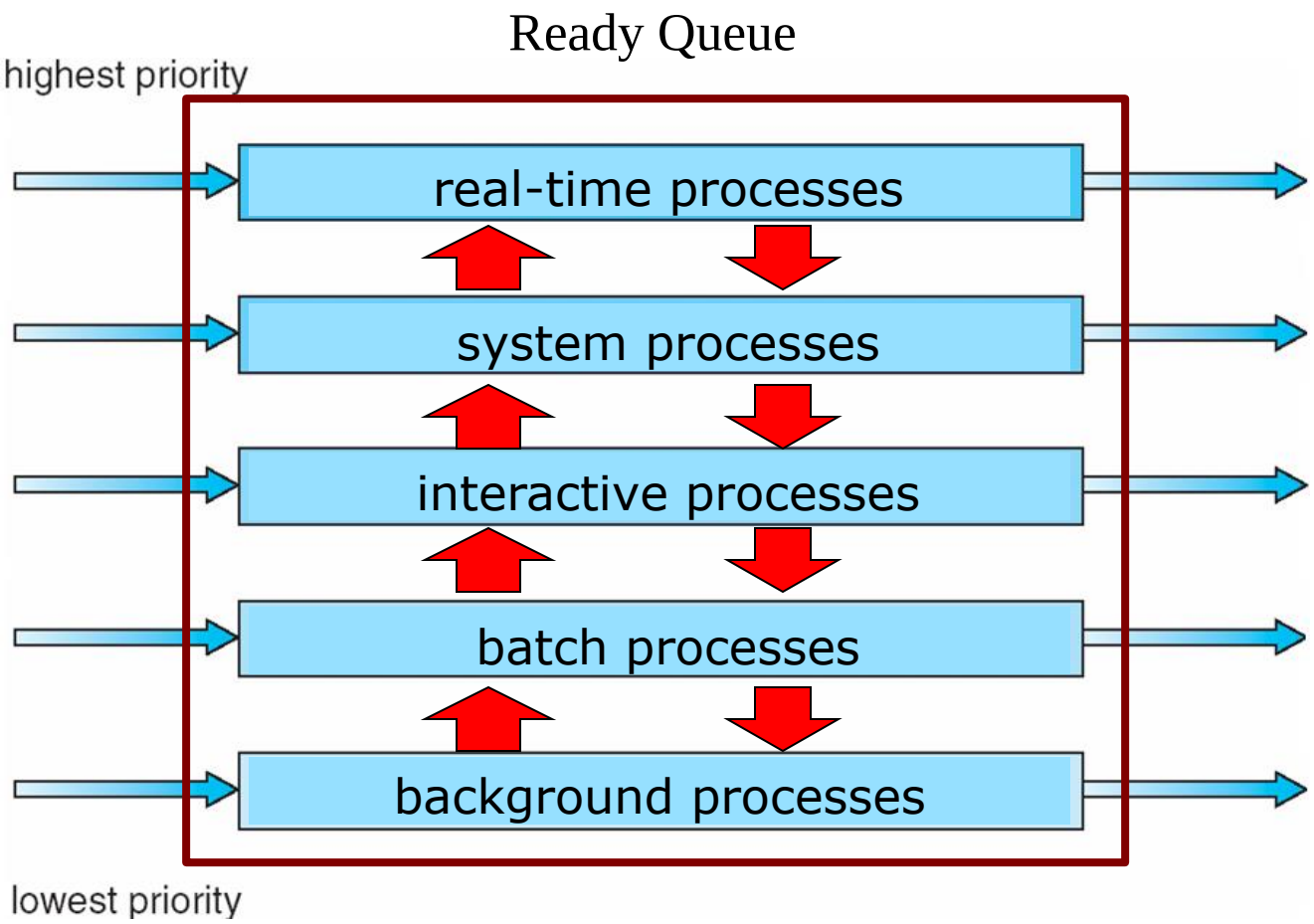


Priority-based Scheduling

- **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

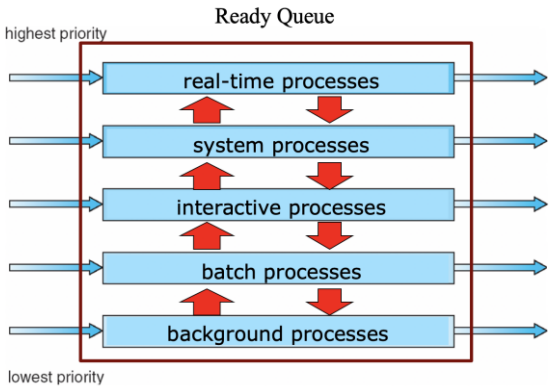
- **Some starvation-free solutions**:
  - Assign each priority level a proportion of time, with higher proportions for higher priorities, and rotate among the levels
    - Similar to weighted round robin, except across levels
  - Create a *multi-level feedback queue* that allows a task to move up/down in priority
    - Avoids starvation of low priority tasks

Multilevel Feedback Queue Scheduling



Multilevel Feedback Queue

- **Multilevel-feedback-queue scheduler defined bv 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

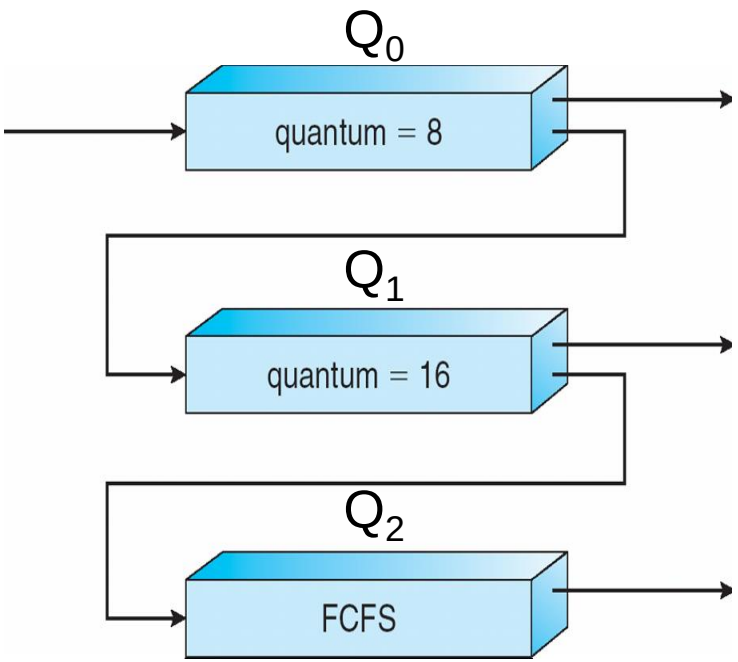
Multi-level Feedback Queues

- **Criteria for process movement among priority queues could depend upon behavior of a process:**
  - 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/lower 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, job receives 8 ms
  - If it does not finish in 8 ms, job is preempted and moved to Q_1
  - At Q_1 job receives additional 16 ms.
  - If it still does not complete, it is preempted and moved to Q_2
  - Interactive processes are more likely to finish early, processing only a small amount of data
  - Compute-bound processes will exhaust their time slice



Interactive processes will gravitate towards higher priority queues, while Compute bound will move to lower priority queues

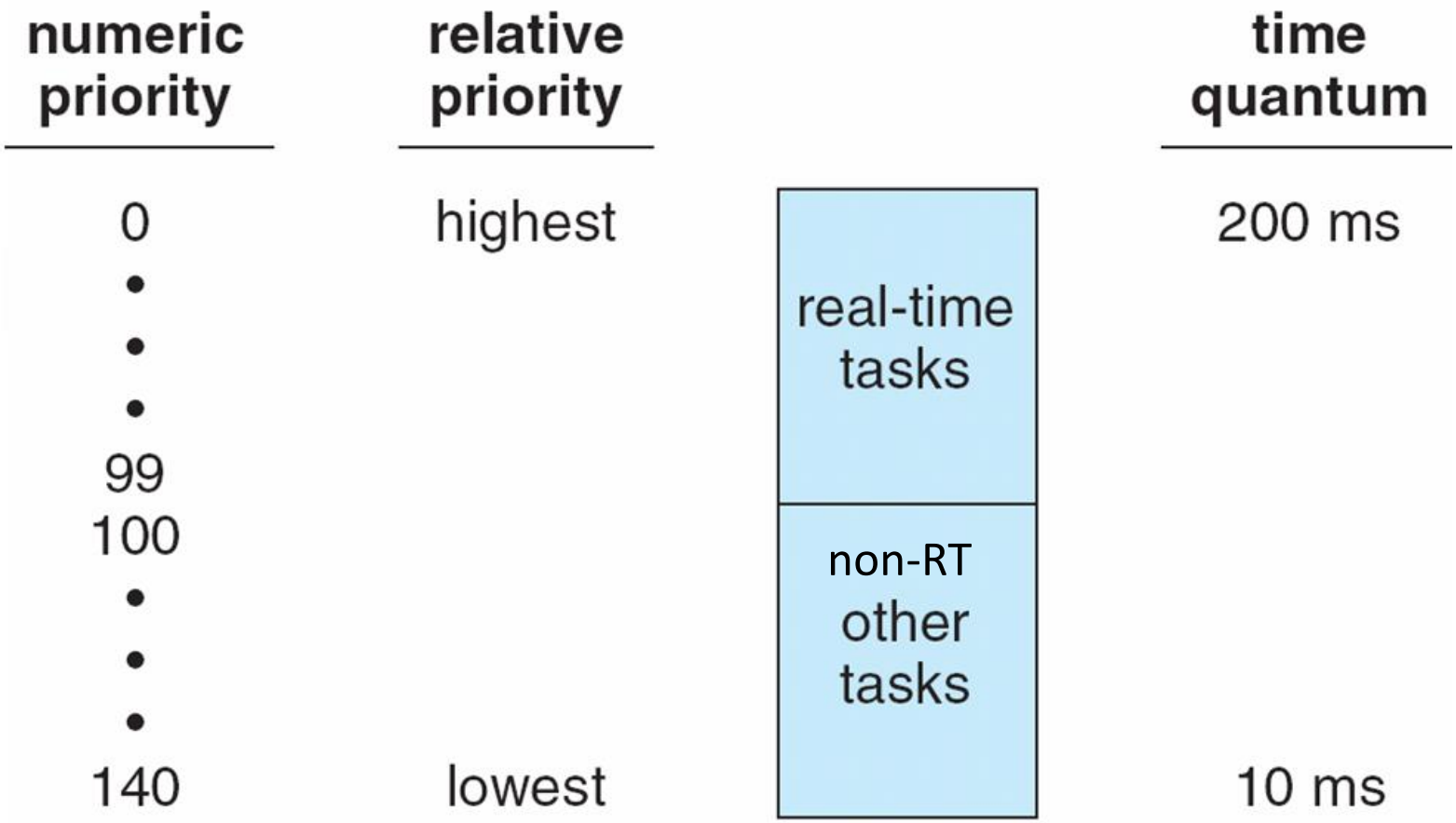
Priority-based Scheduling

- **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 higher than the priority range for 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
  - Linux has
    - priorities 0-99 are for important/real-time processes
    - 100-139 are for 'nice' user processes.
    - Lower values mean higher priorities.
    - Also, longer time quanta for higher priority tasks
      - 200 ms for highest priority
      - Only 10 ms for lowest priority

Linux Priorities and Timeslice 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

More Linux Scheduler History

- **Linux 2.4 introduced an O(N) scheduler – help interactive processes**
  - If an interactive process yields its time slice before it's done, then its "goodness" is rewarded with a higher priority next time it executes
  - Keep a list of goodness of all tasks.
  - But this was unordered. So had to search over entire list of N tasks to find the "best" next task to schedule – hence O(N)
    - doesn't scale well

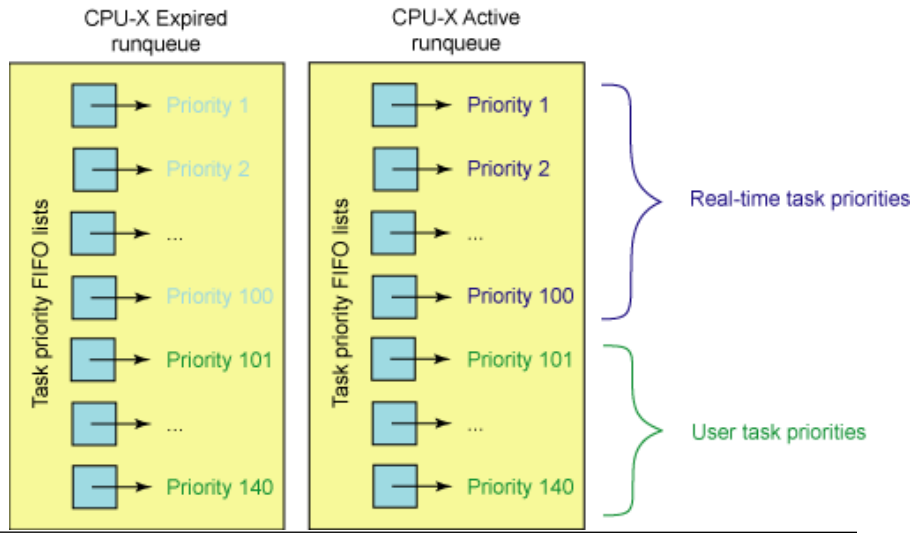
More Linux Scheduler History

- **Linux 2.6-2.6.23 uses an O(1) scheduler**
  - Iterate over fixed # of 140 priorities to find the highest priority task
  - The amount of search time is bounded by the # priorities, not the # of tasks.
  - Hence O(1) is often called “constant time”
  - scales well because larger # tasks doesn’t affect time to find best next task to schedule

O(1) Scheduler in Linux

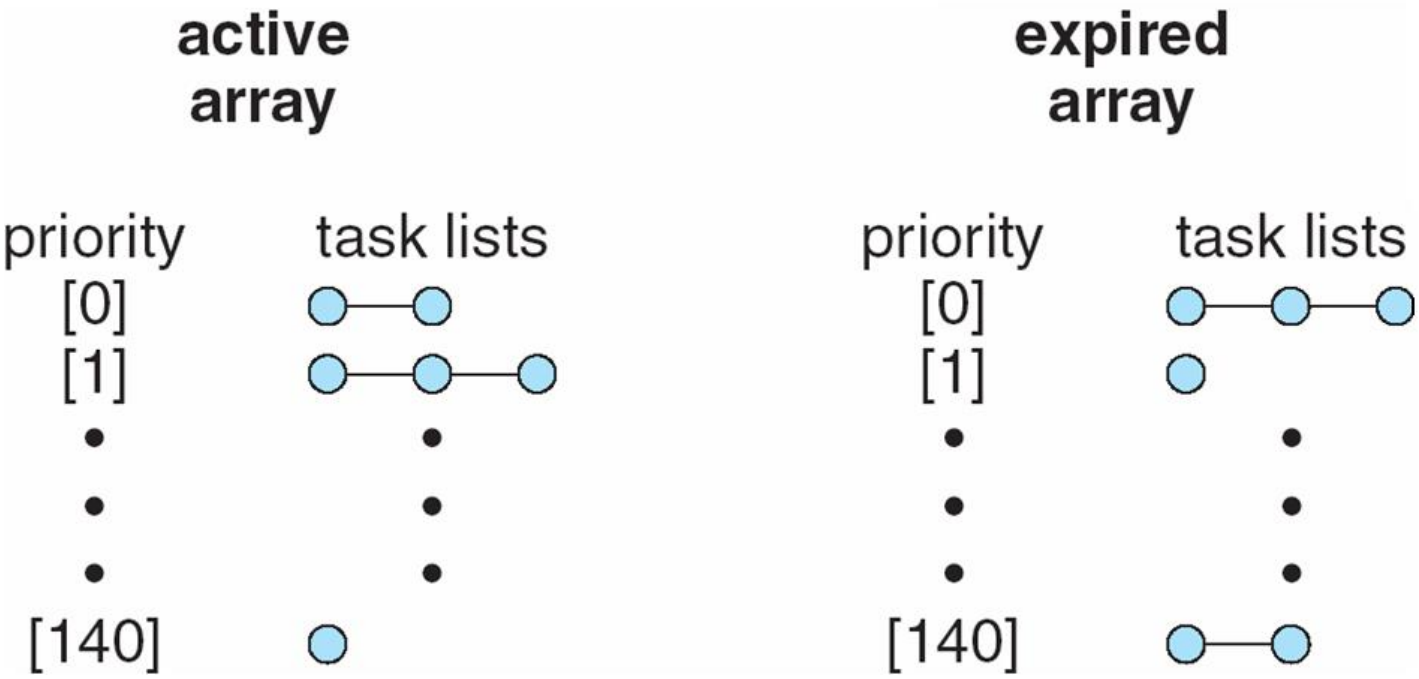
- **Linux maintains two queues:**
  - an active array or run queue and an expired array/queue, each indexed by 140 priorities
- **Active array contains all tasks with time remaining in their time slices, and expired array contains all expired tasks**

- Once a task has exhausted its time slice, it is moved to the expired queue



From http://www.ibm.com/developerworks/linux/library/l-scheduler/

O(1) Scheduler in Linux

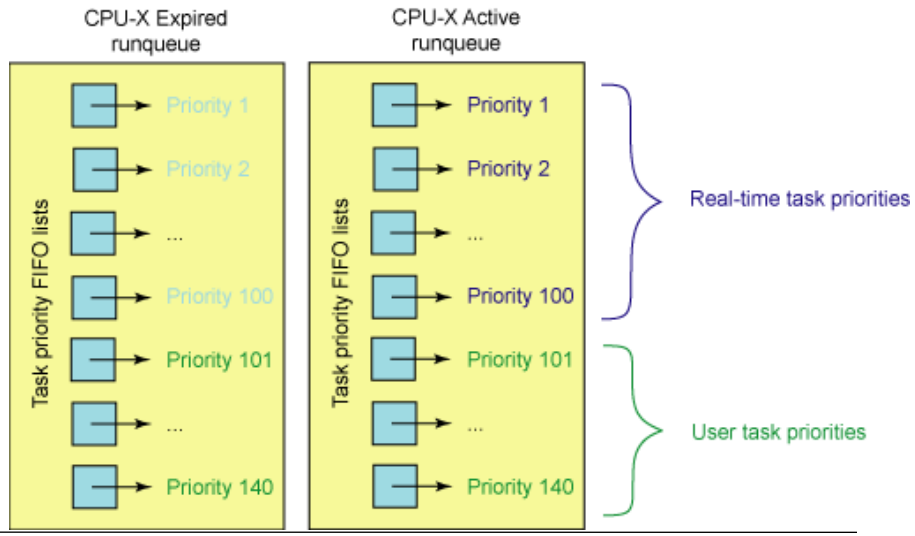


O(1) Scheduler in Linux

- **An expired task is not eligible for execution again until all other tasks have exhausted their time slice**

- **Scheduler chooses task with highest priority from active array**

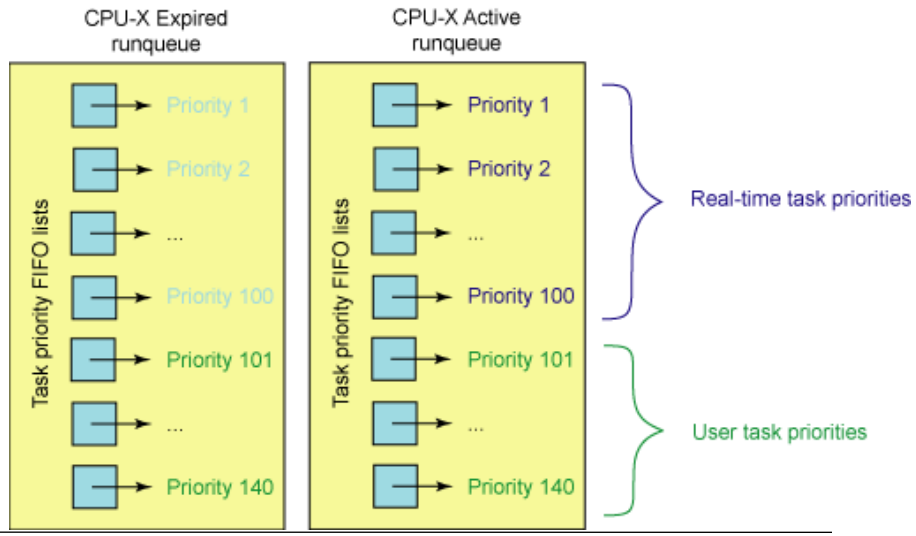
  - Just search linearly through the active array from priority 1 until you find the first priority whose queue contains at least one unexpired task



O(1) Scheduler in Linux

- **# of steps to find the highest priority task is in the worst case 140**
  - This search is bounded and depends only on the # priorities, not # of tasks, unlike the O(N) scheduler
  - hence this is O(1) in complexity

- When all tasks have exhausted their time slices, the two priority arrays are exchanged
  - the expired array becomes the active array



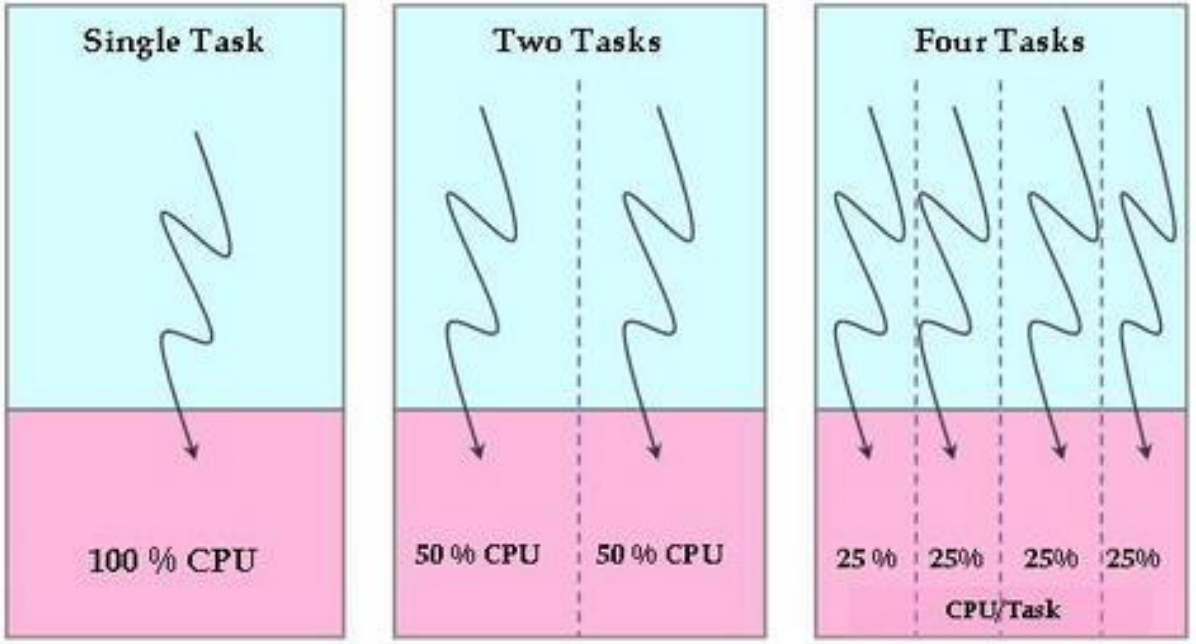
O(1) Scheduler in Linux

- **When a task is moved from run to expired, Linux recalculates its priority according to a heuristic**
  - New priority = nice value +/- f(interactivity)
    - f() can change the priority by at most +/-5, and is closer to -5 if a task has been sleeping while waiting for I/O
    - interactive tasks tend to wait longer times for I/O, and thus their priority is boosted -5, and closer to +5 for compute-bound tasks
  - This dynamic reassignment of priorities affects only the lowest 40 priorities for non-RT/user tasks (corresponds to the nice range of +/- 20)
  - The heuristics became difficult to implement/maintain

Linux CFS

Completely Fair Scheduler (CFS) in Linux

- Linux 2.6.23+/3.* has a "completely fair" scheduler
- Based on concept of an "ideal" multitasking CPU
- If there are N tasks, an ideal CPU gives each task 1/N of CPU *at every instant of time*



CFS Intuition

- On an ideal CPU, N tasks would run truly in parallel, each getting 1/N of CPU and each executing at every instant of time
  - Example: for a 4 GHz processor, if there are 4 tasks, each gets a 1 GHz processor for each instant of time
  - Each such task makes progress at every instant of time
  - This is “fair” sharing of the CPU among each of the tasks

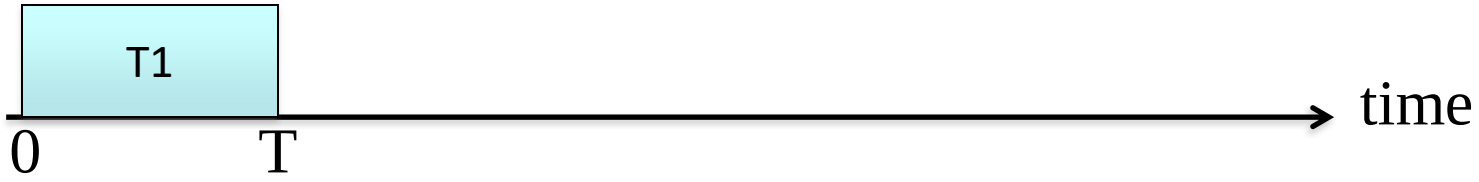
CFS Intuition

- In practice, we know a real (1-core) CPU cannot run N tasks truly in parallel
  - Only 1 task can run at a time
  - Time slice in/out the N tasks, so that in steady state each task gets ~ 1/N of CPU
  - This gives the illusion of parallelism
  - Thus, what we have is concurrency, i.e. the N tasks run concurrently, but not truly in parallel

CFS Intuition

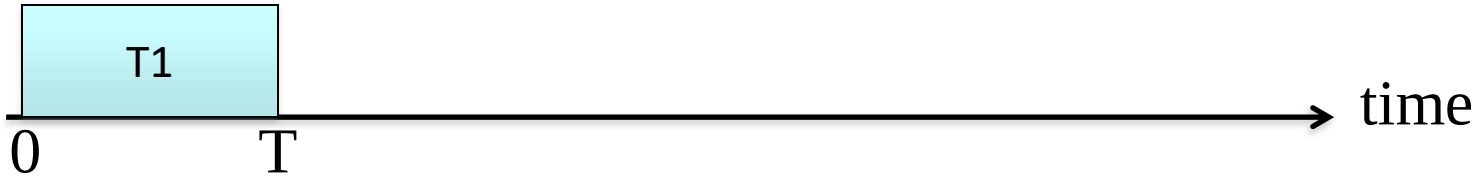
- Ingo Molnar (designer of CFS):
  - “CFS basically models an 'ideal, precise multitasking CPU' on real hardware.”
- So CFS’s goal is to approximate an ideally shared CPU
- Approach: when a task is given T seconds to execute, keep a running balance of the amount of time owed to other tasks as if they all ran on an ideal CPU

CFS Intuition



- Example:
  - Task T1 is given a T second time slice on the CPU
  - Suppose there are 3 other tasks T2, T3, and T4
  - On an ideal CPU, in any interval of time T, then T1, T2, T3 and T4 would each have had the equivalent of time T/4 on the CPU
  - Instead, on a real CPU
    - T1 is given T instead of T/4, so T1 has been overallocated 3T/4
    - T2, T3 and T4 are owed time T/4 on the CPU, i.e. they have each been forced to *wait* the equivalent of T/4

CFS Intuition

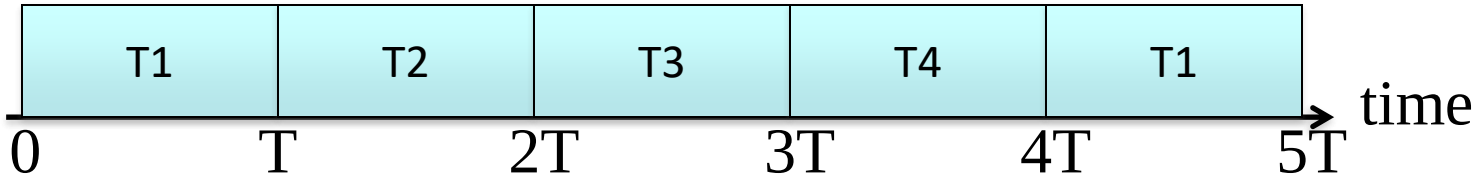


- Example:
  - The current accounting balance is summarized in the table below

	Time owed to task, i.e. wait time W_i:			
Giving time T to task:	T1	T2	T3	T4
T1	-3T/4	T/4	T/4	T/4

  - In general, at any given time t in the system, each task T_i has an amount of time owed to it on an ideal CPU, i.e. the amount of time it was forced to wait, its *wait time $W_i(t)$,*

CFS Intuition



- Example: let's have round robin over 4 tasks

	Time owed to task, i.e. wait time W_i:			
Giving time T to task:	T1	T2	T3	T4
T1	-3T/4	T/4	T/4	T/4
then T2	-T/2	-T/2	T/2	T/2
then T3	-T/4	-T/4	-T/4	3T/4
then T4	0	0	0	0

- After 1 round robin, the balances owed all = 0, so every task receives its fair share of CPU over time 4T

CFS Intuition

- Suppose a 5th task T5 is added to the round robin
  - Now the amount owed/wait time is calculated as T/5 for each task not chosen for a time slice, and as -4T/5 for the chosen task for a time slice
  - In general, if there are N runnable tasks, then
    - (N-1)T/N is subtracted from the balance owed/wait time of the chosen task
    - T/N is added to the balanced owed/wait time of all other ready-to-run tasks
  - T5 is initially owed no CPU time, so $W_5 = 0$
    - Example: If T5 had arrived just after T2's time slice, then T5's wait time =0 would place it above T1 and T2 but below T3 and T4 in terms of amount of time owed on the CPU

CFS Scheduler in Linux

- Goal of CFS Scheduler: *select the task with the longest wait time*
  - i.e. choose max W_i
    i
  - This is the task that is owed the most time on the CPU and so should be run next to achieve fairness most quickly

Wait Time Calculation

- Each scheduling decision at time k incurs a wait time $W_i(k)$, either positive or negative, to each task i
- Total accumulated wait time for each task i at time k is:

$$Wtotal_i(k) = \sum_{j=1}^{k} W_i(j)$$

Wait Time Calculation

- Each wait time $W_i(k)$ =
  - Either a penalty of T/N added to $Wtotal_i$ if task i is not chosen to be scheduled, or
  - (N-1)T/N is *subtracted* from the sum (=T-T/N) if task i is chosen to be scheduled
- So $W_i(k)$= either T/N or –T+T/N
  - note how T/N is added regardless of the case!
- Hence $W_i(k)$ = T/N - execution/run time given to task i at time k, which may be zero
  - Define run time $R_i(k)$ as the execution/run time given to task i at time k, which may be zero
  - So $W_i(k) = T/N - R_i(k)$

Wait Time Calculation

- In general, each scheduling decision at time k may choose:
  - An arbitrary amount of time T(k) to schedule the chosen task, i.e. it doesn't have to be a fixed time slot T
  - The number of runnable tasks N(k) may change at each decision time k
- So $W_i(k) = T(k)/N(k) - R_i(k)$

Wait Time Calculation

- Total accumulated wait time for each task i at time k is:

$$Wtotal_i(k) = \sum_{j=1}^{k} W_i(j) = \sum_{j=1}^{k} [T(j)/N(j) + R_i(j)]$$

$$= \sum_{j=1}^{k} T(j)/N(j) + \sum_{j=1}^{k} R_i(j)$$

Global fair clock measuring how system time advances in an ideal CPU with N varying tasks, also called rq->fair_clock in CFS' 1st implementation

Total run time given task i. Let's define it as $Rtotal_i(k)$

CFS Scheduler in Linux

- Recall: CFS scheduler chooses task with max $Wtotal_i(k)$ at each scheduling decision k
- Maximizing $Wtotal_i(k)$ equivalent to minimizing the quantity [Global fair clock - $Wtotal_i(k)$]
- 1st CFS scheduler:
  - Had to track global fair clock and $Wtotal_i(k)$ for each task i
  - Then would compute the values [Global fair clock - $Wtotal_i(k)$]
  - Then ordered these values in a Red-Black tree
  - Then selected leftmost node in tree (has minimum value) and scheduled the task corresponding to this node

CFS Scheduler in Linux

- Revised CFS scheduler:
  - We note that [Global fair clock - $Wtotal_i(k)$] = run time $Rtotal_i(k)$!
  - Minimizing over the quantities [Global fair clock - $Wtotal_i(k)$] is equivalent to minimizing over the accumulated run times $Rtotal_i(k)$
  - 1st CFS scheduler had to track complex values like the global fair clock, and accumulated wait times
    - These both needed the # runnable tasks N(k) at each scheduling decision time k, which keeps changing
  - New approach just sums run times given each task
  - this simple approach still achieves fairness according to our derivation

Virtual Run Time

- Revised CFS scheduler simply sums the run times given each task and chooses the one to schedule with the minimum sum
  - This is equivalent to choosing the task owed the most time on an ideal fair CPU according to our derivation, and thus achieves fairness
  - Caveat: when a new task is added to the run queue, it may have been blocked a long time, so its run time may be very low compared to other tasks in the run queue
    - Such a task would consume a long time before its accumulated run time rises to a level close to the other executing tasks' total run times, which would effectively block other tasks from running in a timely manner

Virtual Run Time

- Revised CFS scheduler accommodates new tasks as follows:
  - Define a virtual run time *vruntime*
    - As before, each normally running task i simply adds its given run times to its own accumulated sum $vruntime_i$
  - When a new task is added to the run queue (or an existing task becomes unblocked from I/O), assign it a new virtual run time = minimum of current vruntimes in the run queue
    - This quantity is defined as *min_vruntime*
  - This approach re-normalizes the newly active task's run time to about the level of the virtual run times of the currently runnable tasks

Virtual Run Time

- Since each newly active task's is given a re-normalized run time, then the run time calculated is not the actual execution time given a task
  - Hence we need to define a new term *vruntime$_i$(k)*, rather than use the absolute accumulated run time Rtotal$_i$(k)
- *Intuitively, CFS choosing the task with the minimum virtual run time prioritizes the task that been given the least time on the CPU*
  - *This is the task that should get service first to ensure fairness*

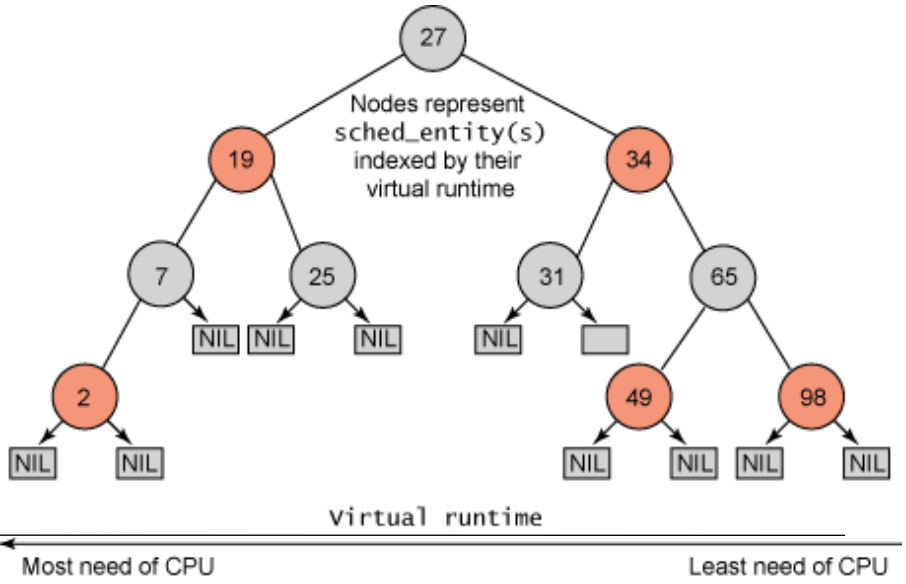
CFS Scheduler in Linux

- *So revised CFS scheduler chooses the task with the minimum $vruntime_i(k)$ at each scheduling decision time k*
- This approach is responsive to interactive tasks!
  - They get instant service after they unblock from their I/O
  - This is because they are given a re-normalized $vruntime_i(k) = min\_vruntime$,
  - Since CFS chooses the next task to schedule as the one with the minimum vruntime, then the interactive task will be chosen first and get service immediately

CFS' Red Black Tree

- To quickly find the task with the minimum vruntime, order the vruntimes in a Red-Black tree
  - This is a balanced tree, ordered from left (minimum vruntime) to right (maximum vruntime)

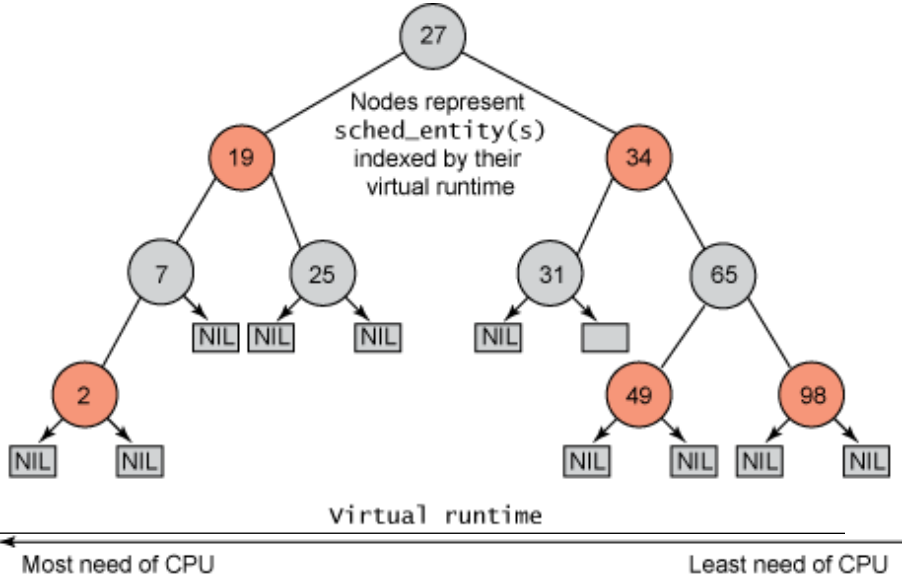
- Finding the minimum is fast, simple and constant time!
  - Choose leftmost task in tree with lowest virtual run time to schedule next



CFS' Red Black Tree

- As tasks run more, their virtual run time increases
  - so they migrate to other positions further to the right in the tree
  - Must re-insert nodes to tree, and rearrange tree, but the RB tree is self-balancing

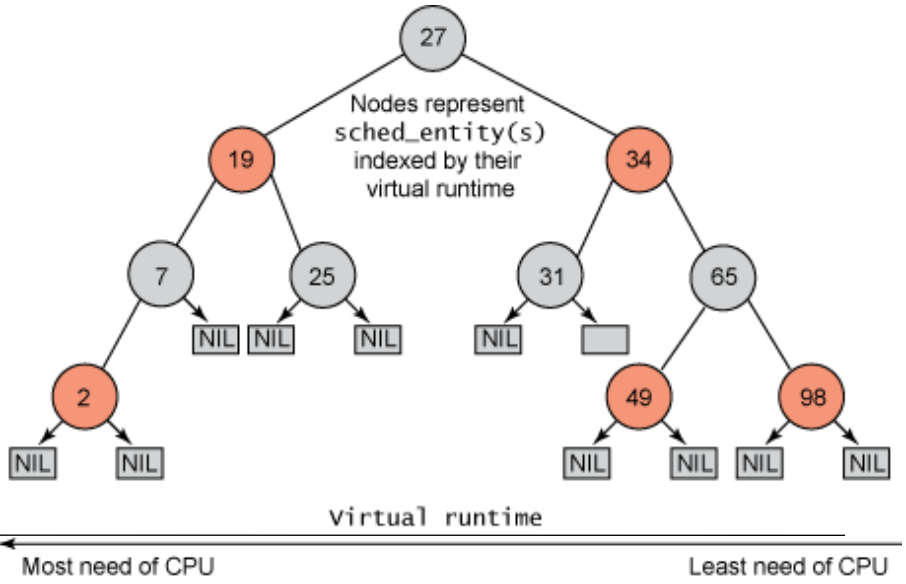
- Inserting nodes is an O(logN) operation due to RB tree
  - This is viewed as acceptable overhead



CFS' Red Black Tree

- Tasks that haven't had CPU execution in a while will migrate left and eventually get service
  - Intuitively, this eventual migration leftwards makes CFS fair

- Newly active tasks, e.g. interactive ones, will be added to the left of the tree and get service quickly



CFS Scheduler and Priorities

- All non-Real Time tasks of differing priorities are combined into one RB tree
- don’t need 40 separate run queues, one for each priority - elegant!
- Higher priority tasks get larger run time slices
- lower niceness => higher the priority => more run time is given on the CPU

CFS Scheduler and Priorities

- Higher priority tasks are scheduled more often
  - *virtual runtime += (actual CPU runtime) * NICE_0 / task's weight*

  - Higher priority
    $\Rightarrow$ higher weight
    $\Rightarrow$ less increment of *vruntime*
    $\Rightarrow$ task is further left on the RB tree and is scheduled sooner

CFS Scheduler and Priorities

- While CFS is fair to tasks, it is not necessarily fair to applications
  - Suppose application A1 has 100 threads T1-T100
  - Suppose application A2 is interactive and has one thread T101
  - CFS would give
    - A1 100/101 of CPU
    - A2 only 1/101 of the CPU
- Instead, Linux CFS supports fairness across groups:
  - A1 is in group 1 and A2 is in group 2
  - Groups 1 and 2 each get 50% of CPU – fair!
  - Within Group 1, 100 threads share 50% of CPU
- Multi-threaded apps don't overwhelm single thread apps

Realtime and Multi-Core Scheduling

Real Time Scheduling in Linux

- Linux also includes three **real-time** scheduling classes:
  - Real time FIFO – soft real time (SCHED_FIFO)
  - Real time Round Robin – soft real time (SCHED_RR)
  - Real time Earliest Deadline First – hard real time as of Linux 3.14 (SCHED_DEADLINE)

- Only processes with the priorities 0-99 have access to these RT schedulers

Real Time Scheduling in Linux

- A real time FIFO task continues to run until it voluntarily yields the processor, blocks or is preempted by a higher-priority real-time task
  - no timeslices
  - all other tasks of lower priority will not be scheduled until it relinquishes the CPU
  - two equal-priority Real time FIFO tasks do not preempt each other

Real Time Scheduling in Linux

- SCHED_RR is similar to SCHED_FIFO, except that such tasks are allotted timeslices based on their priority and run until they exhaust their timeslice

- Non-real time tasks continue to use CFS algorithm

- SCHED_DEADLINE uses an Earliest Deadline First algorithm to schedule each task.