Design and Analysis of Operating Systems

CSCI 3753

Dr. David Knox
University of Colorado Boulder

Department of Computer Science
UNIVERSITY OF COLORADO **BOULDER**

Design and Analysis of Operating Systems CSCI 3753

Process Scheduling SJF, RR, and EDF

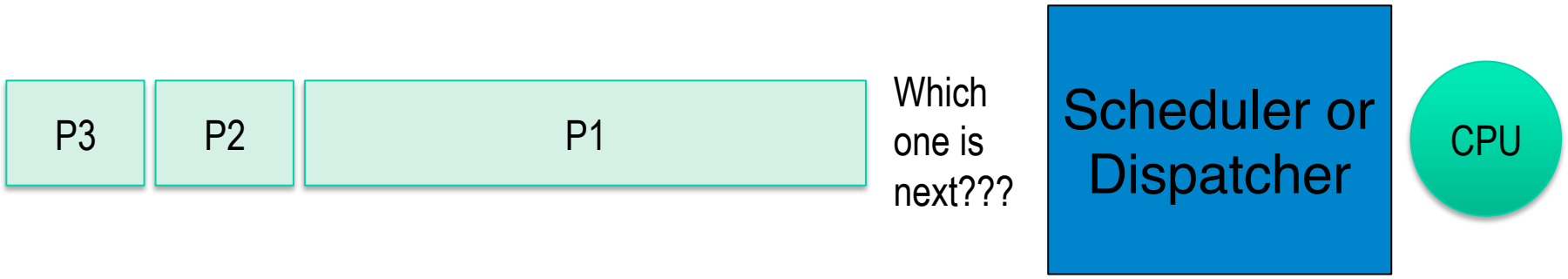
Dr. David Knox

University of
Colorado Boulder

Shortest Job First (SJF) Scheduling

Choose the process/thread with the lowest execution time

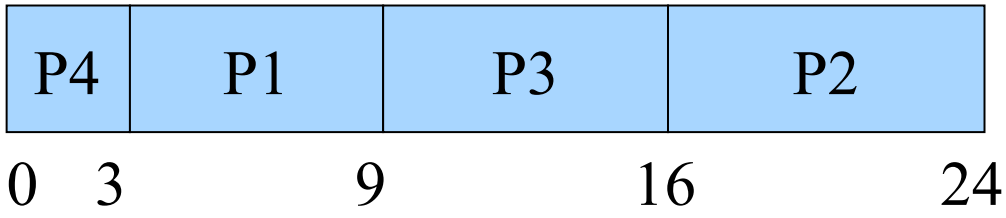
- **gives priority to shortest or briefest processes**
- **minimizes the average wait time**
  - intuition: one long process will increase wait time for all short processes that follow
- **the impact of the wait time on other long processes moved towards the end is minimal**



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 $\{P_a, P_b, \ldots, P_n\}$, suppose one chooses a process P from this set to schedule first
  - The wait times for all the remaining processes in $\{P_a, \ldots, P_n\}$-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<α<1

  - If α>1/2, then estimate is influenced more by recent history.
  - If α<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

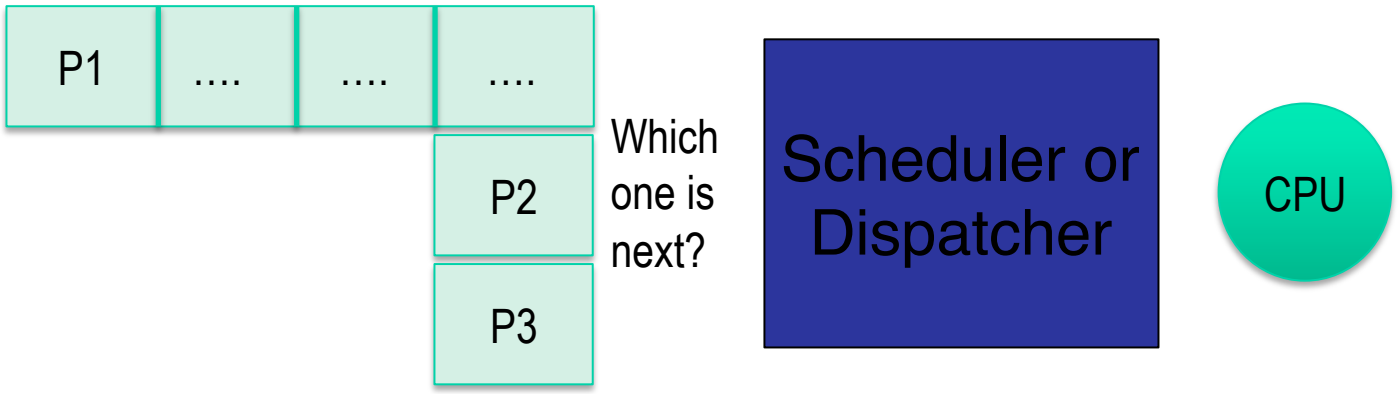
- **SJF can be preemptive:**
  - i.e. when a new job arrives in the ready queue, if its execution time is less than the currently executing job's remaining execution time, then it can preempt the current job
  - For simplicity, we assumed in the preceding analysis that jobs ran to completion and no new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

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?



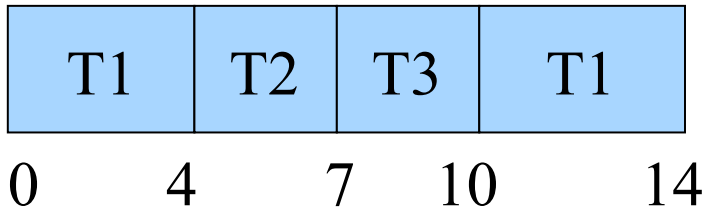
Round Robin Scheduling

- **Use preemptive time slicing**
  - a task is forced to relinquish the CPU before it's necessarily done
- **R*otate* among the tasks in the ready queue**
  - periodic timer interrupt transfers control to the CPU scheduler, giving each task 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*** is fast at 3.66ms
  - assuming a 1ms switching time
  - Compare to FCFS w/ long 1st task

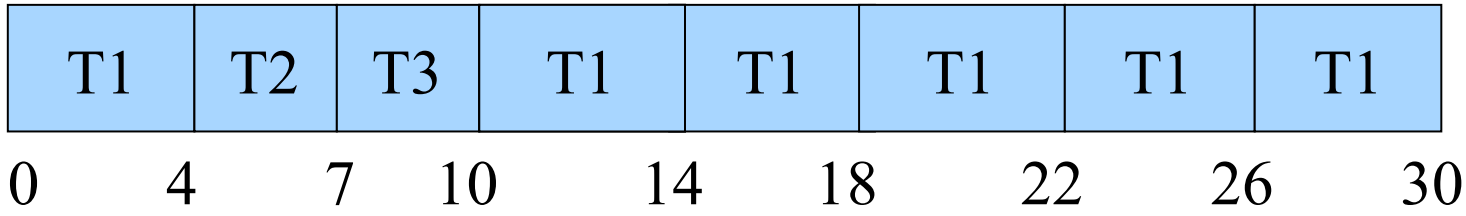
Task	CPU Execution Time (ms)
T1	24
T2	3
T3	3



Round Robin Scheduling

- **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

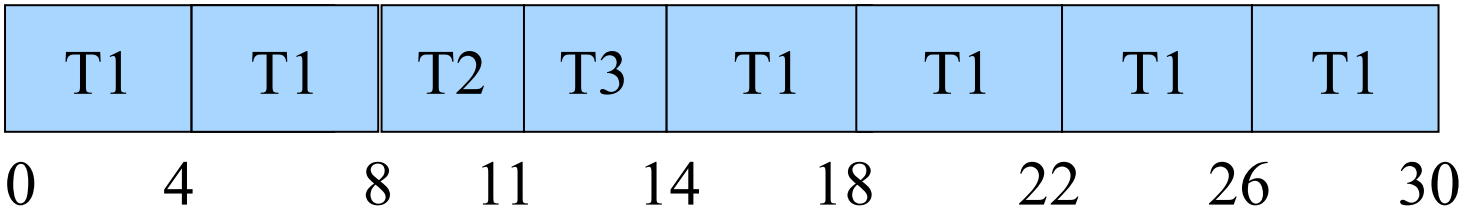
Weighted Round Robin

- **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:

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

Weighted Round Robin

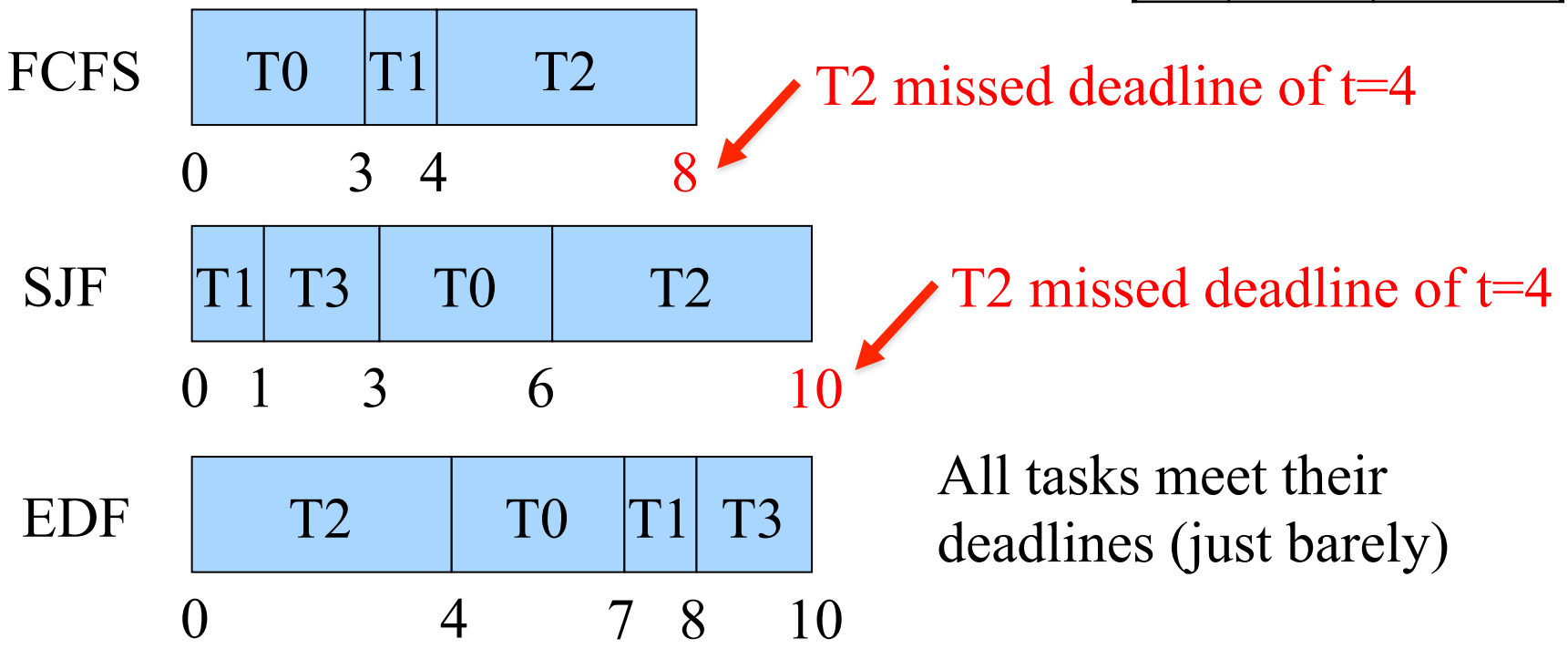
- **In previous example,**
  - could give T1 two time slices
  - T2 and T3 only 1 each round



Earliest Deadline First (EDF) Scheduling

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

- **Choose the task with the earliest deadline**
  - Pick task that 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 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 policies had no notion of refusing admission

EDF and Preemption

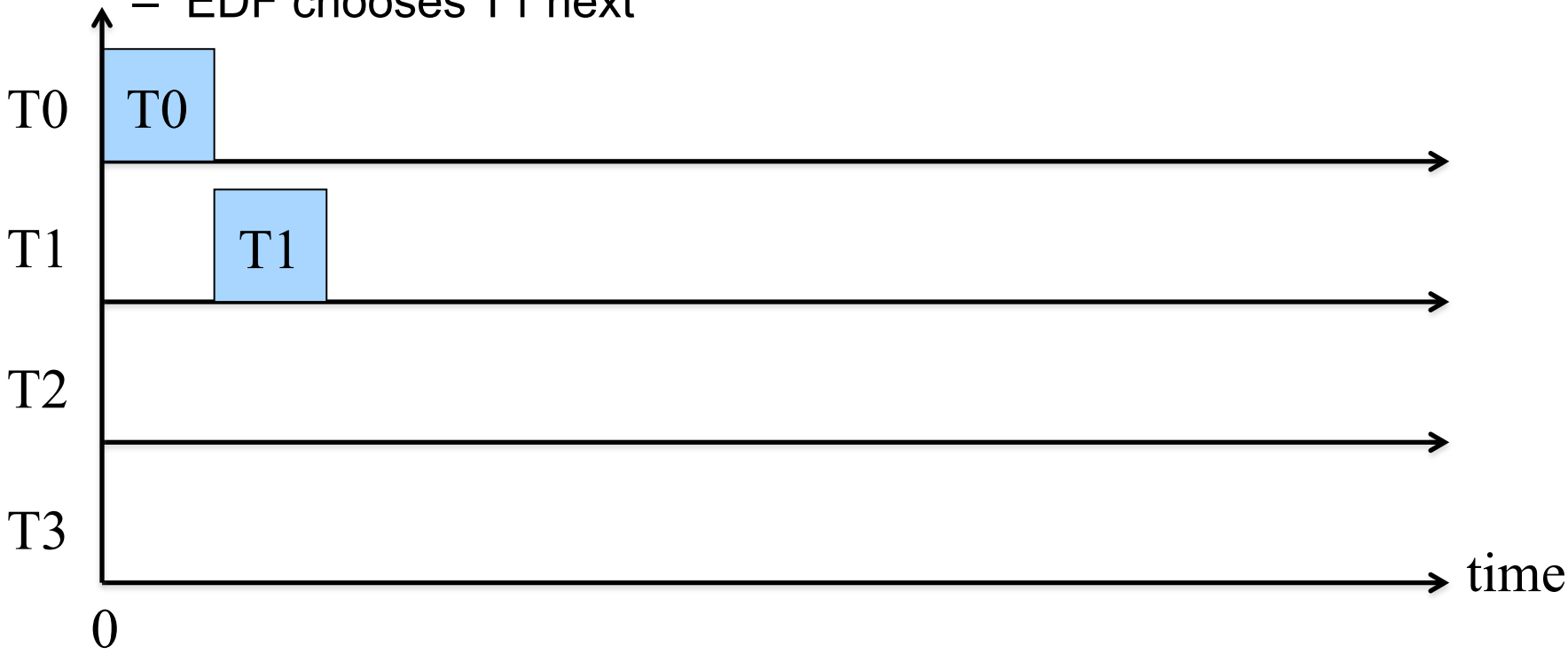
- **Assume a preemptively time sliced system**
  - A task arriving with an earlier deadline can preempt one currently executing with a later deadline.

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

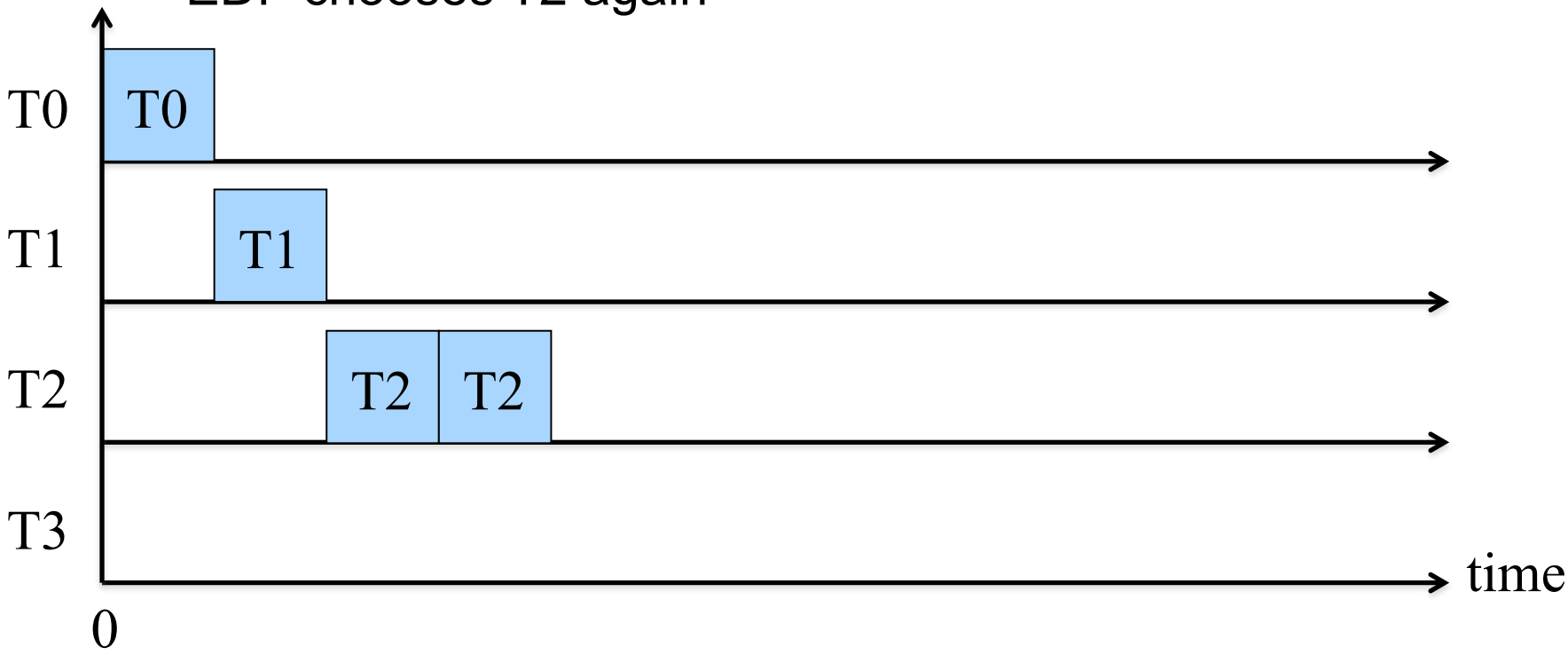
EDF and Preemption

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



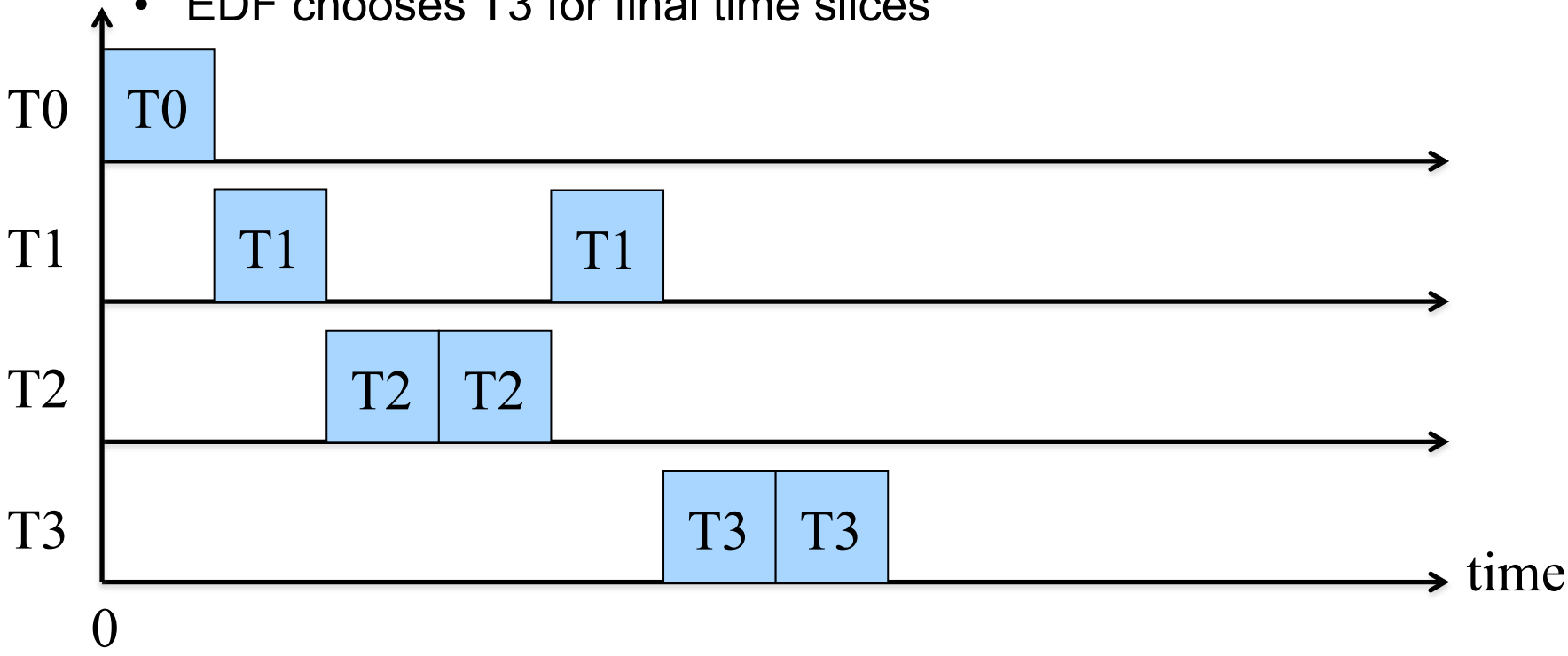
EDF and Preemption

- **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



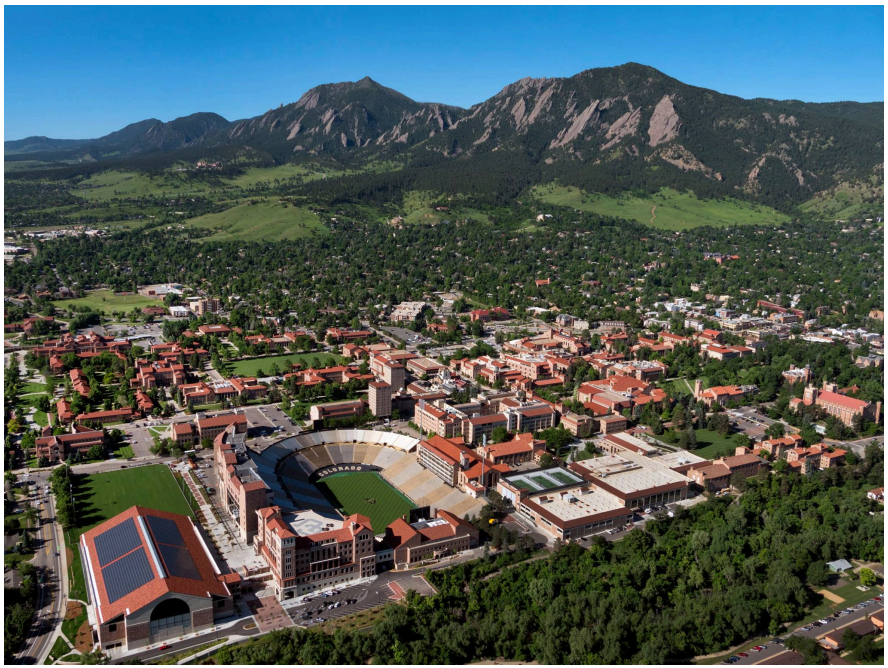
Deadline Scheduling

- **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

Design and Analysis of Operating Systems CSCI 3753



Dr. David Knox
University of
Colorado Boulder