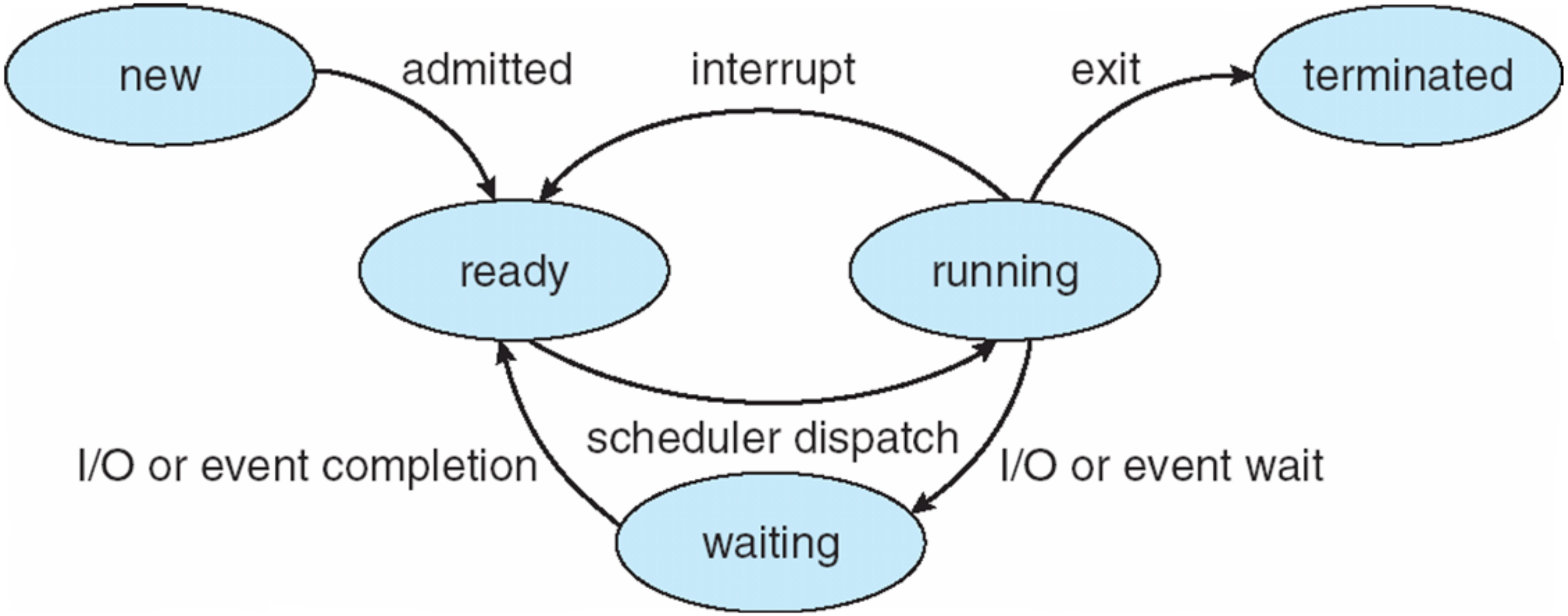
Lecture 14
Process Scheduling

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

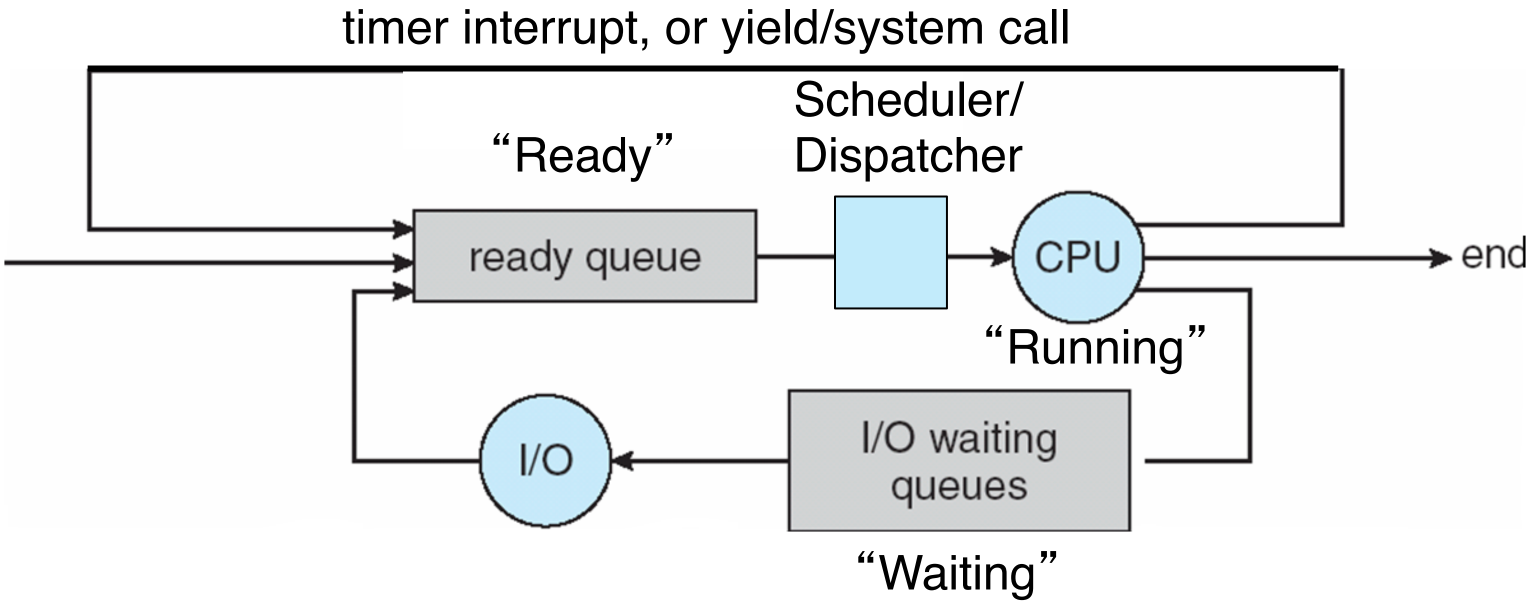
Policy

- 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

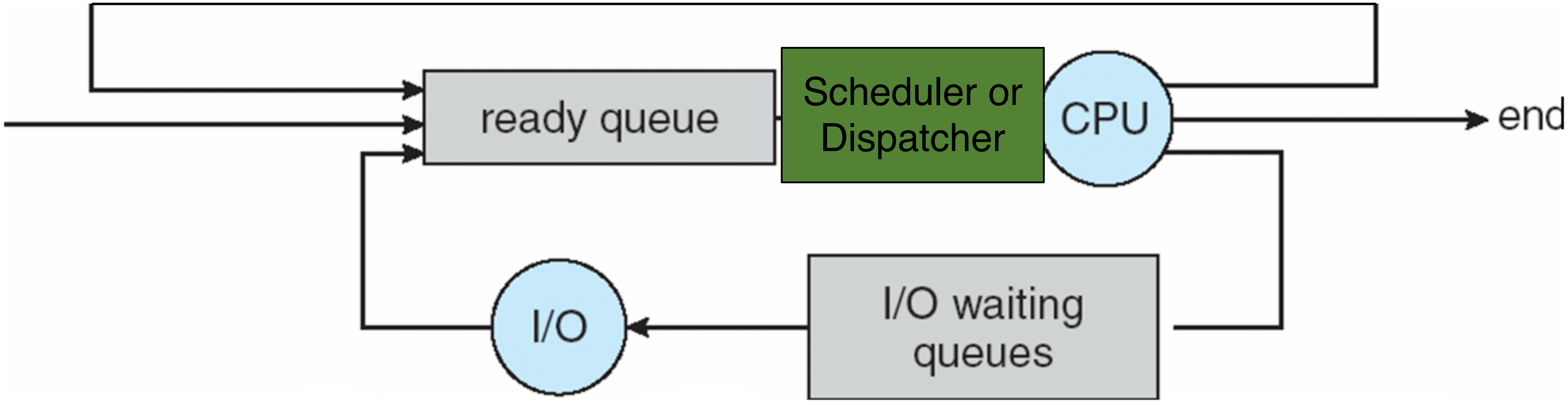


If threads are implemented as kernel threads, then OS can schedule threads as well as processes

Modified version of Silberschatz et al slides

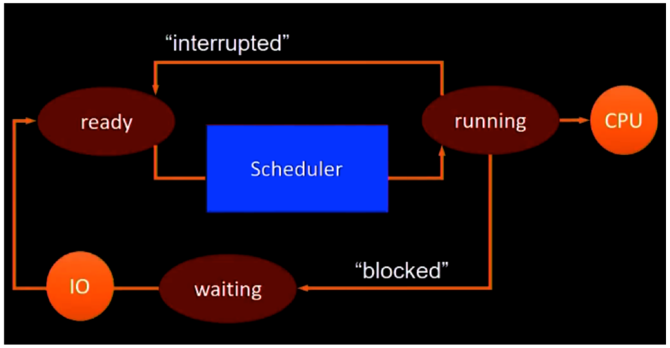
Scheduling Policy

- 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

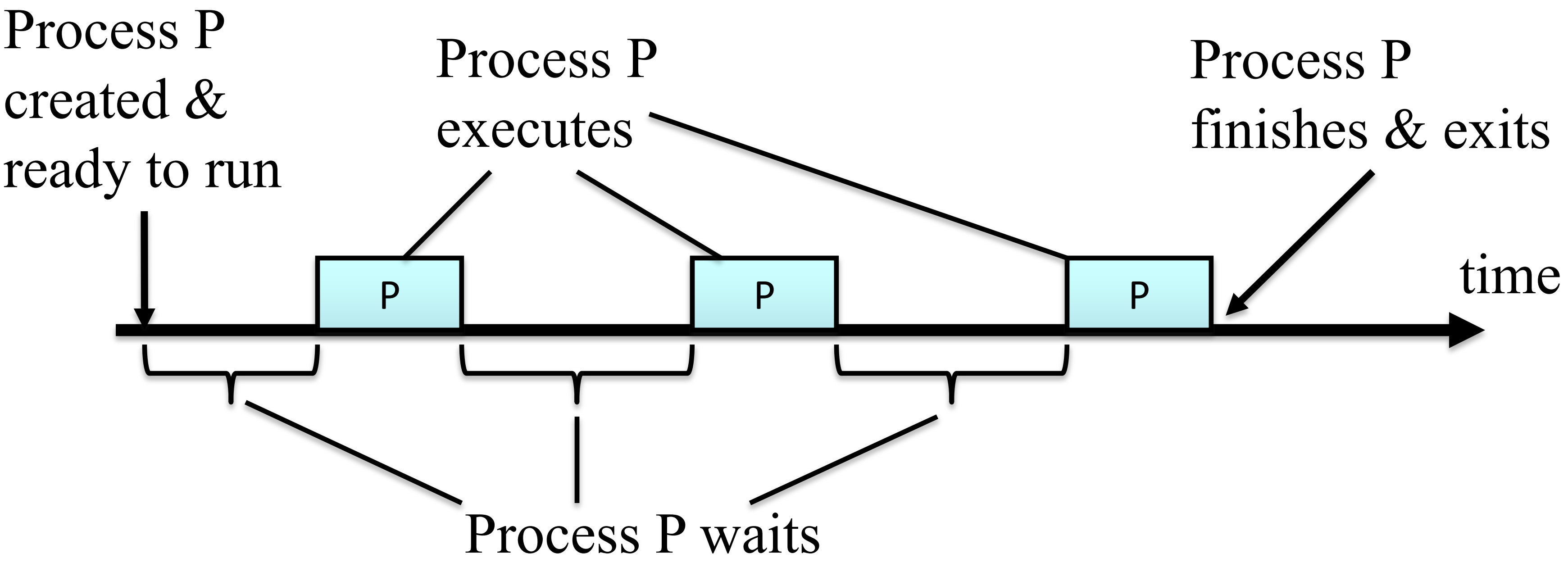


Scheduling Policy

- 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.

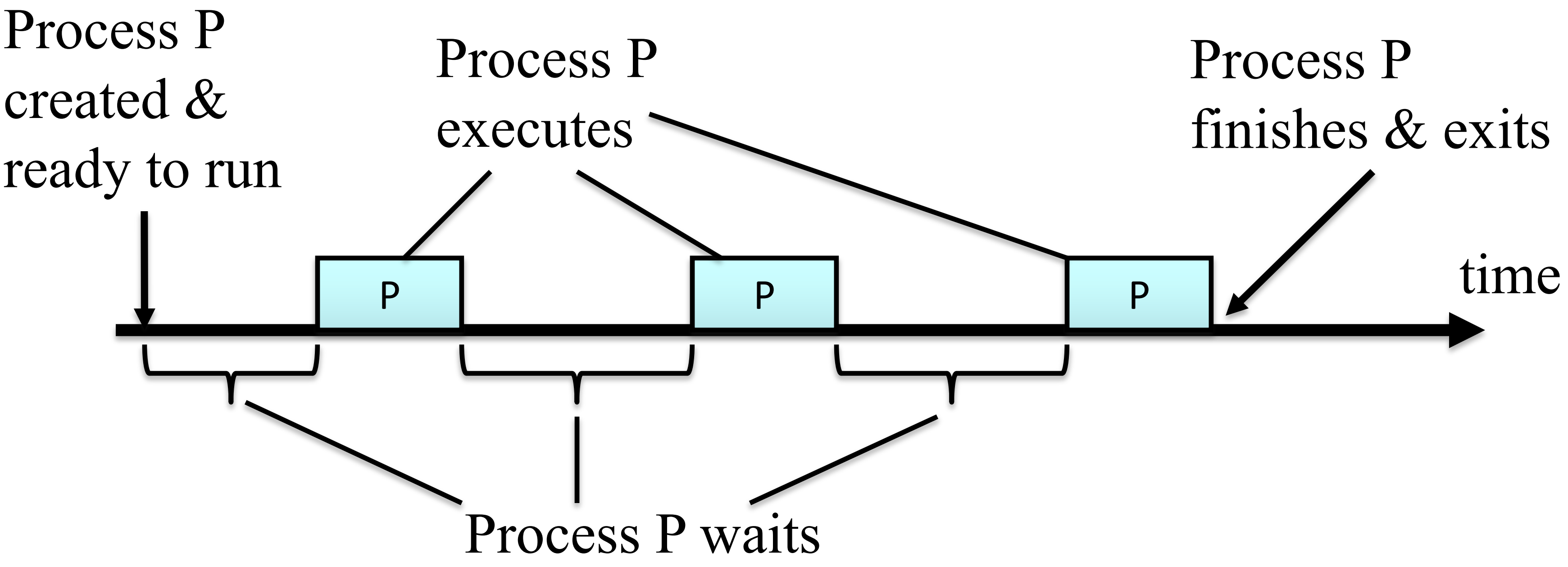


Scheduling Definitions



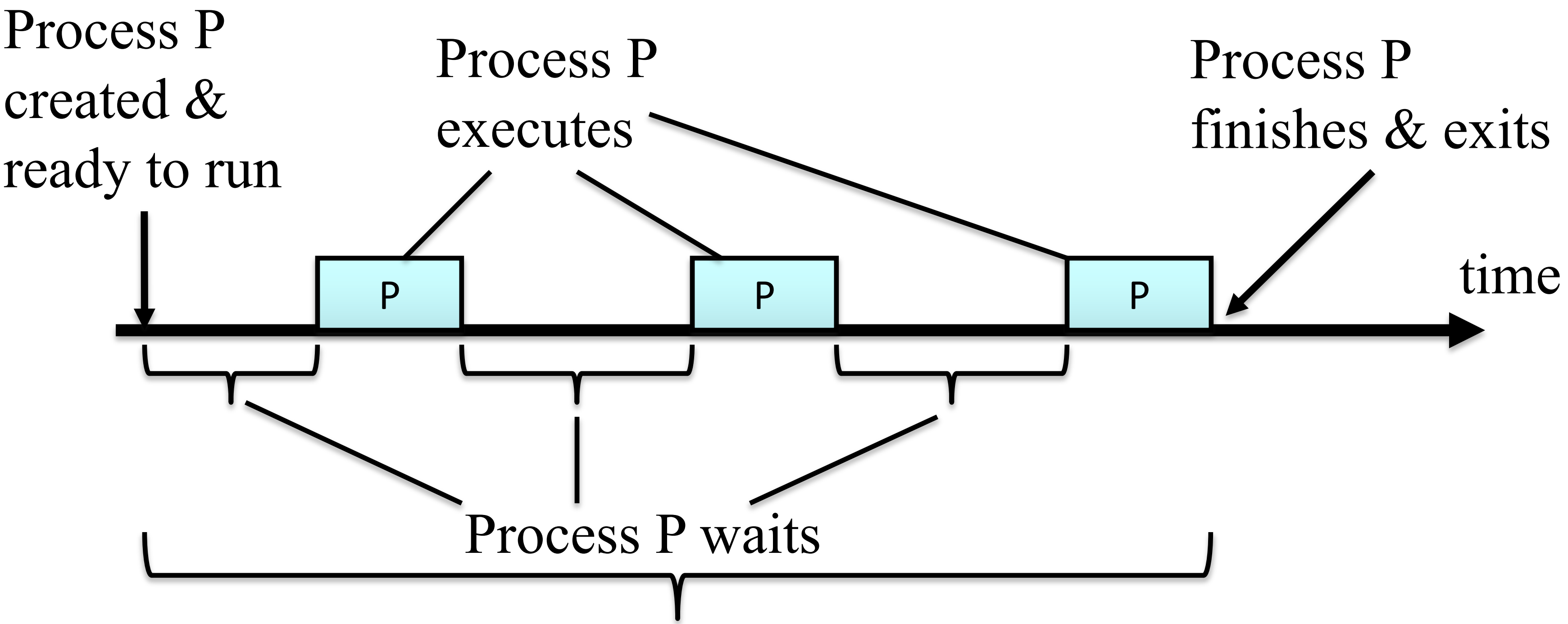
- *execution time* $E(P_i)$ = the time on the CPU required to fully execute process i
  - Sum up the time slices given to process i
  - Also called the “burst time” by textbook

Scheduling Definitions



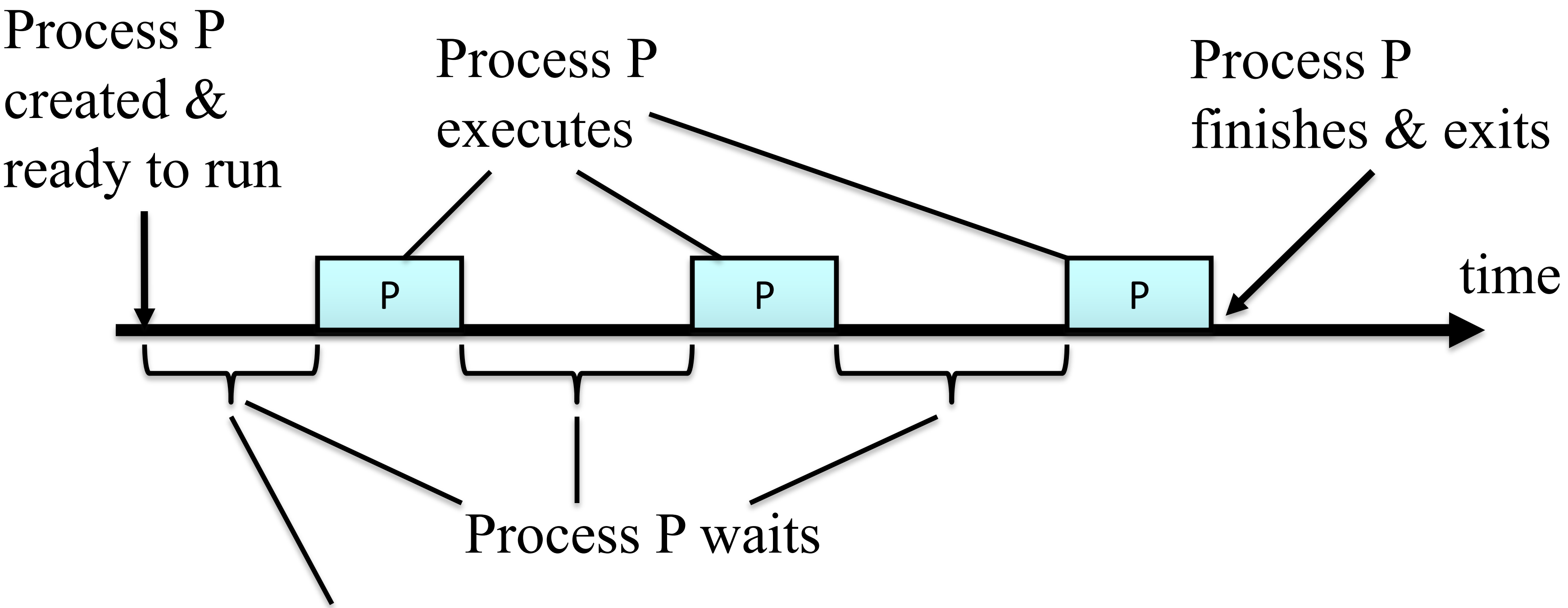
- ***wait time*** W(P_i) = the time process i is in the ready state/queue waiting but not running
  - Sum up the gaps between time slices given to process *i*
  - But, does NOT include I/O waiting time

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

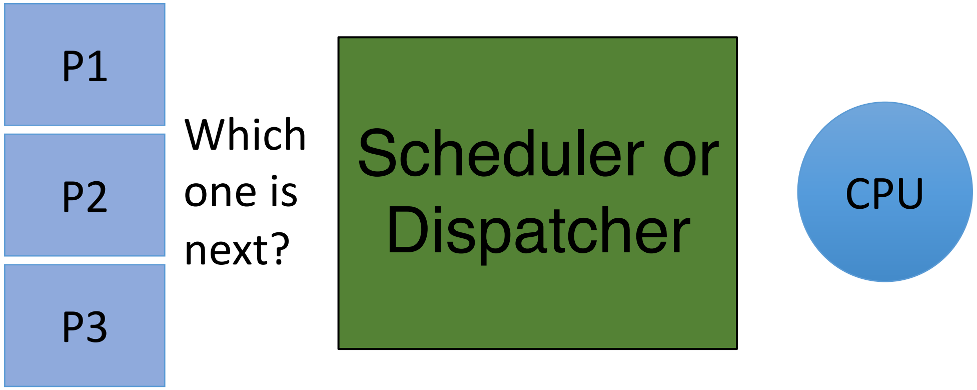
Scheduling Policies

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?

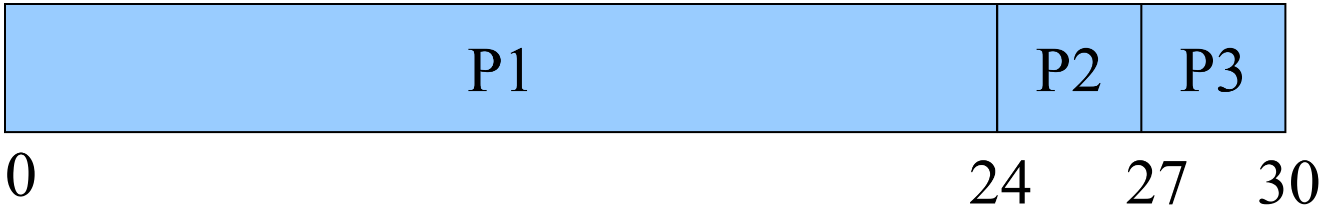


First Come First Serve (FCFS) Scheduling

- 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:

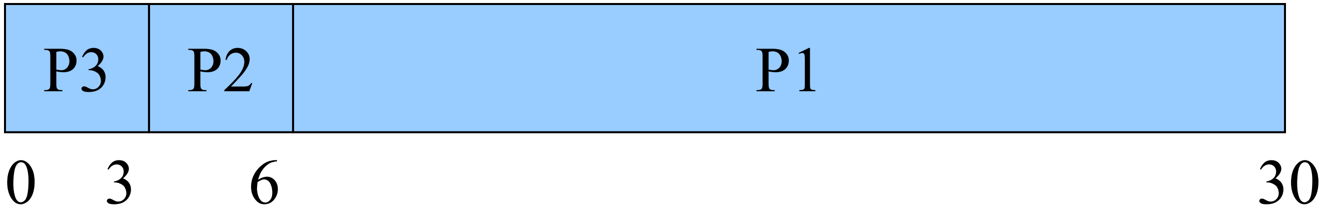
Process	CPU Execution Time
P1	24
P2	3
P3	3



FCFS Scheduling

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

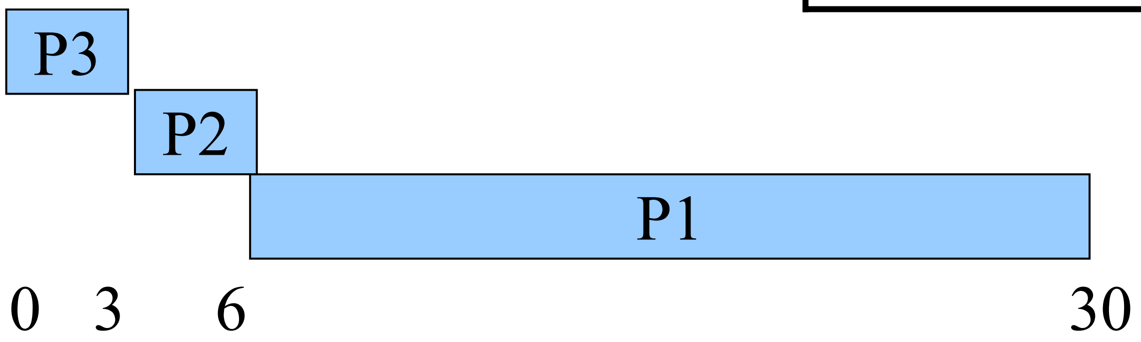
Process	CPU Execution Time
P1	24
P2	3
P3	3



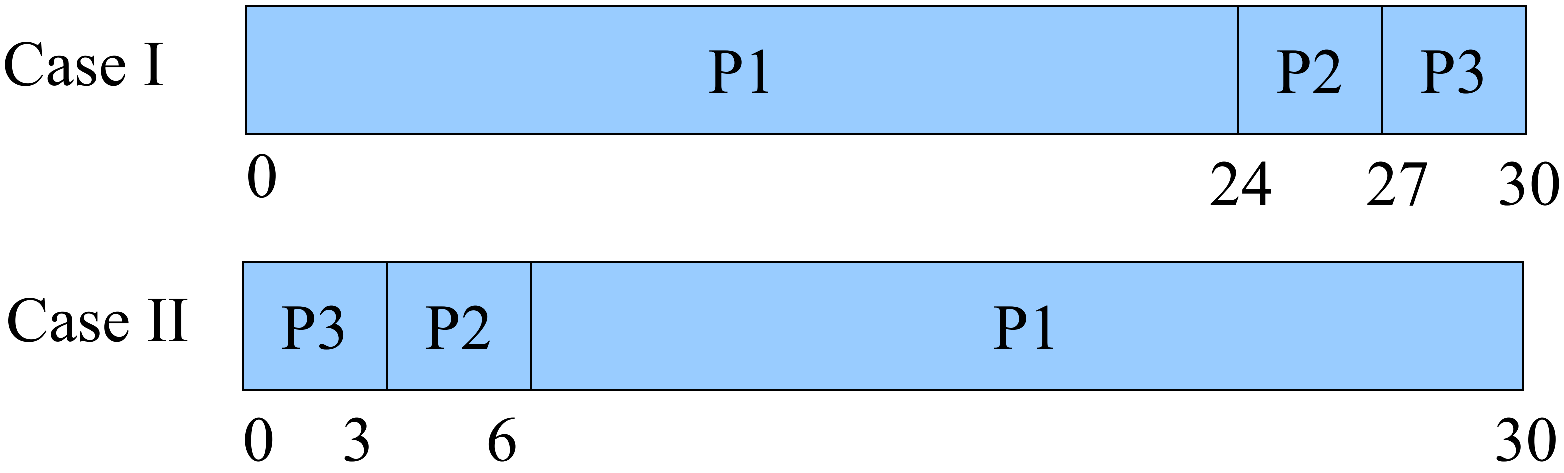
Gantt Chart

- 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

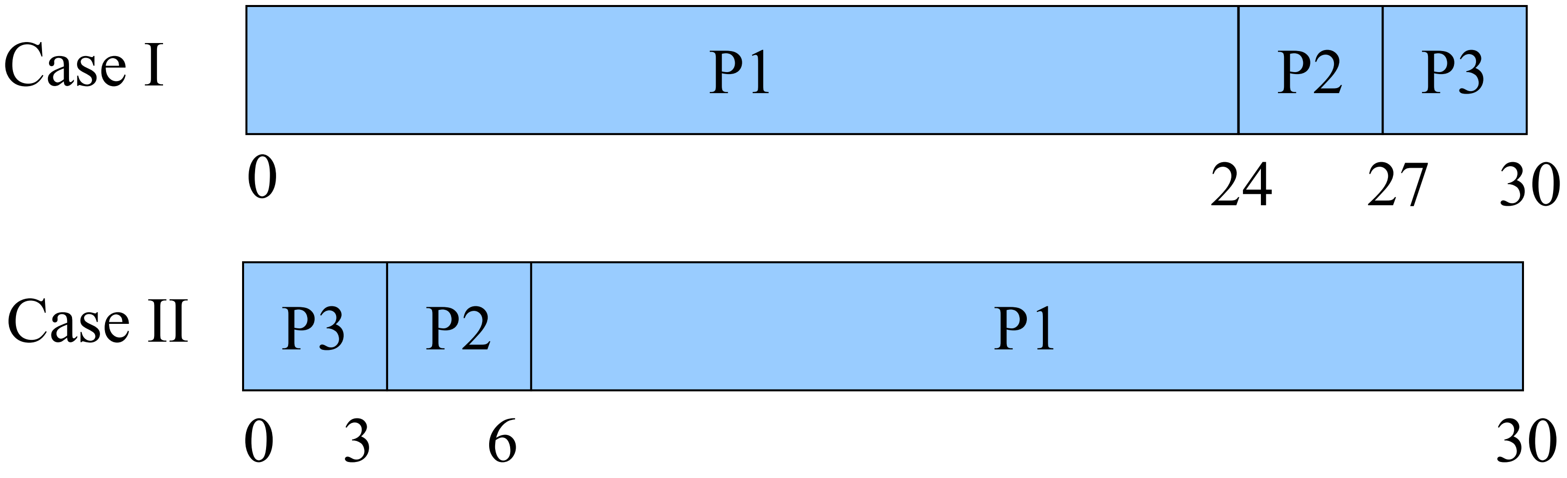
P1 does not wait at all

P2 waits for P1 to complete before being scheduled

P3 waits until P2 has completed

(0 + 24 + 27)/3 = 17

FCFS Scheduling

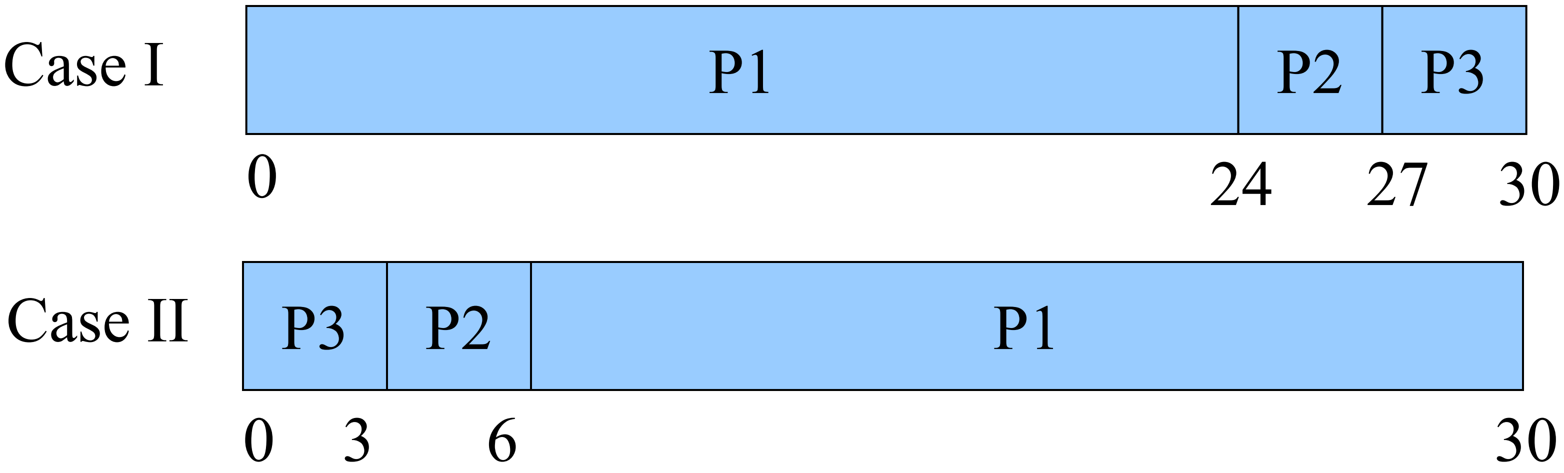


- Case I: **average wait time** is (0+24+27)/3 = 17 seconds

- 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



Case I

P1 | P2 | P3

0 24 27 30

Case II

P3 | P2 | P1

0 3 6 30

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

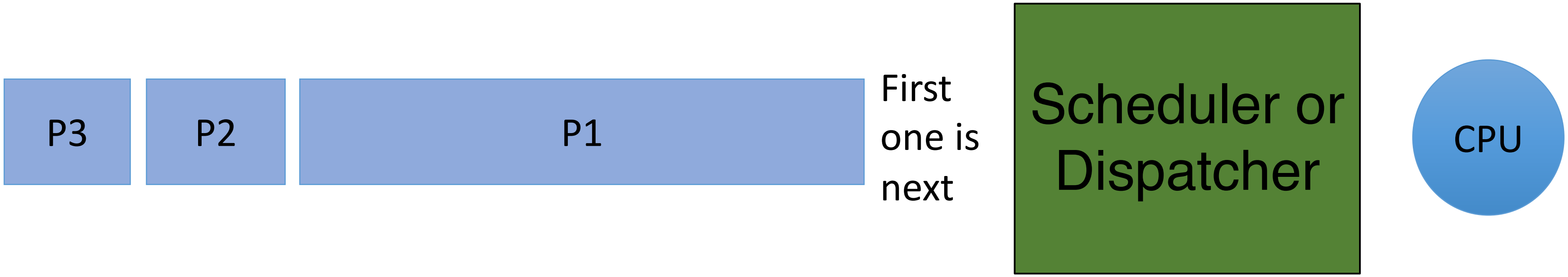
- Case I: average turnaround time is (24+27+30)/3 = 27 seconds
- Case II: average turnaround time is (3+6+30)/3 = 13 seconds

- 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

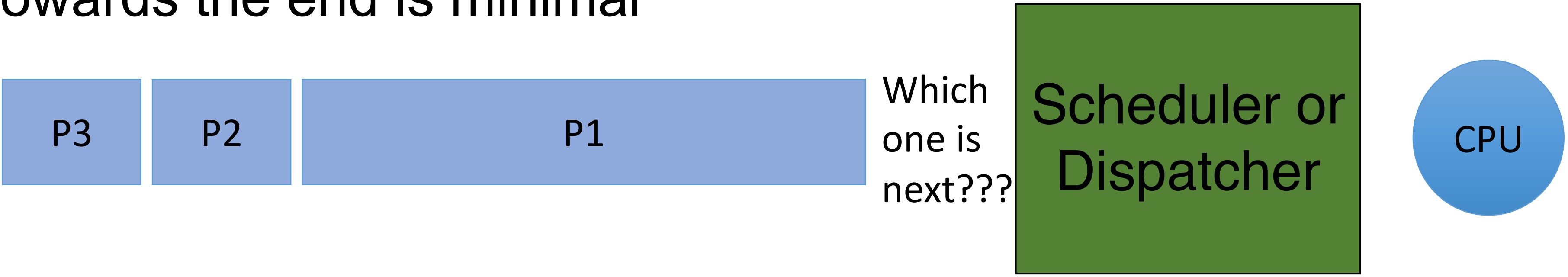
- Does it meet our goals?
  - maximize CPU utilization: 40% to 90%?
  - maximize throughput: # processes completed/second?
  - minimize average or peak wait, turnaround, or response time?
  - meet deadlines or delay guarantees?
  - maximize fairness?



Shortest Job First (SJF) Scheduling

Choose the process/thread with
the lowest execution time

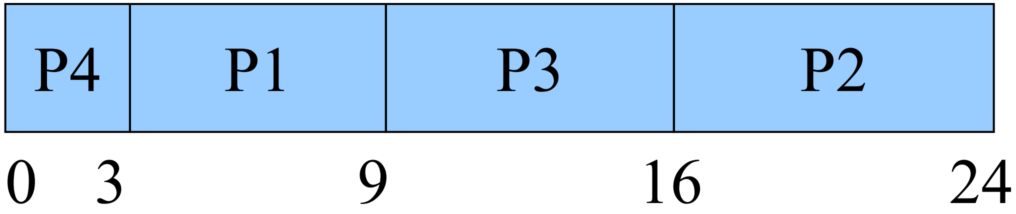
- gives priority to shortest or briefest processes
- minimizes the average wait time
  - intuition: one long process will increase wait time for all short processes that follow
- the impact of the wait time on other long processes moved towards the end is minimal



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


P4 P1 P3 P2

0 3 9 16 24

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

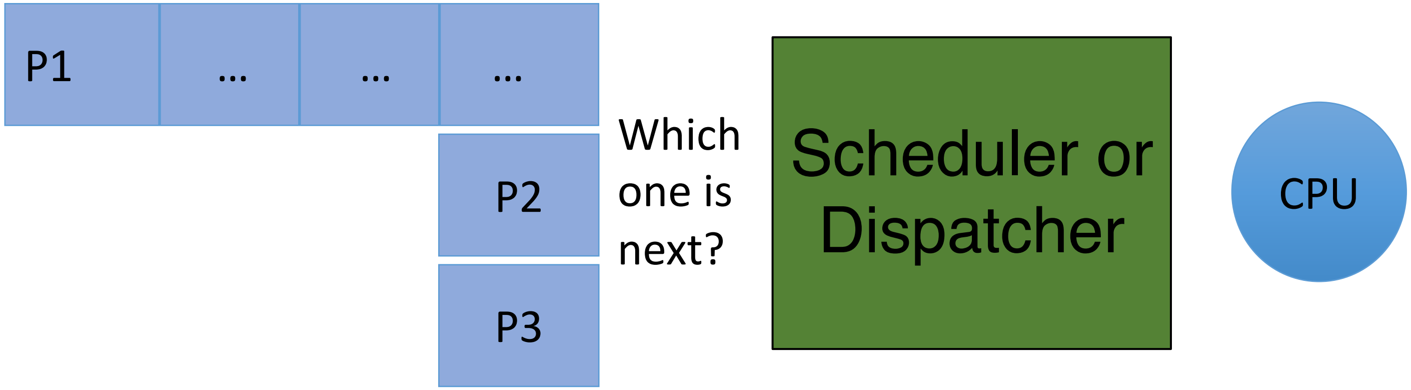
- **SJF can be preemptive:**
  - i.e. when a new job arrives in the ready queue, if its execution time is less than the currently executing job’s remaining execution time, then it can preempt the current job
  - Compare to FCFS: a new process can’t preempt earlier processes, because its order is later 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

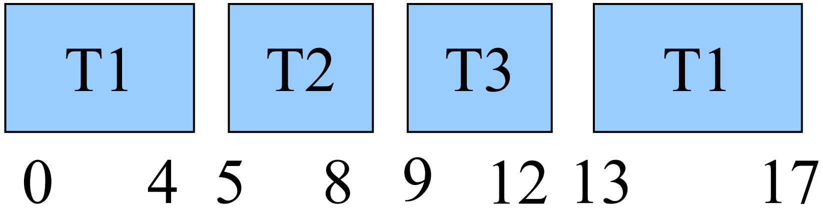
- **Use preemptive <u>time slicing</u>**
  - a task is forced to relinquish the CPU before it’s necessarily done

- **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

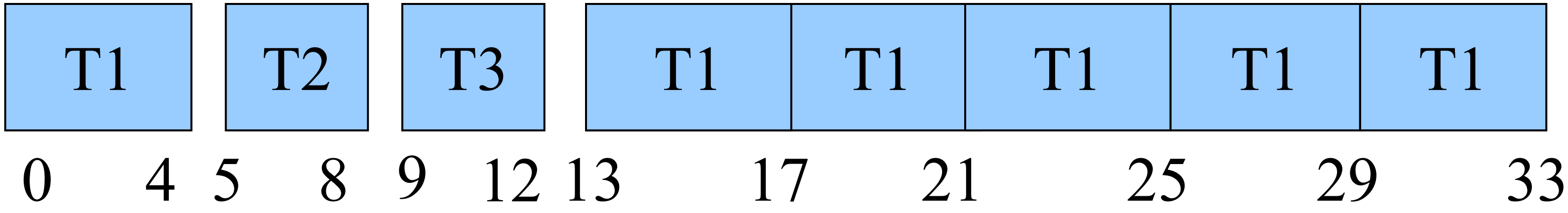
Task	CPU Execution Time (ms)
T1	24
T2	3
T3	3



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 4.66ms*
  - 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

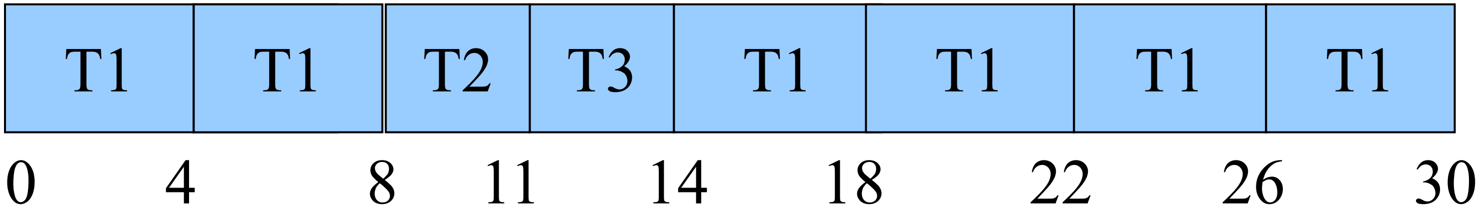
Weighted Round Robin

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



Deadline Scheduling

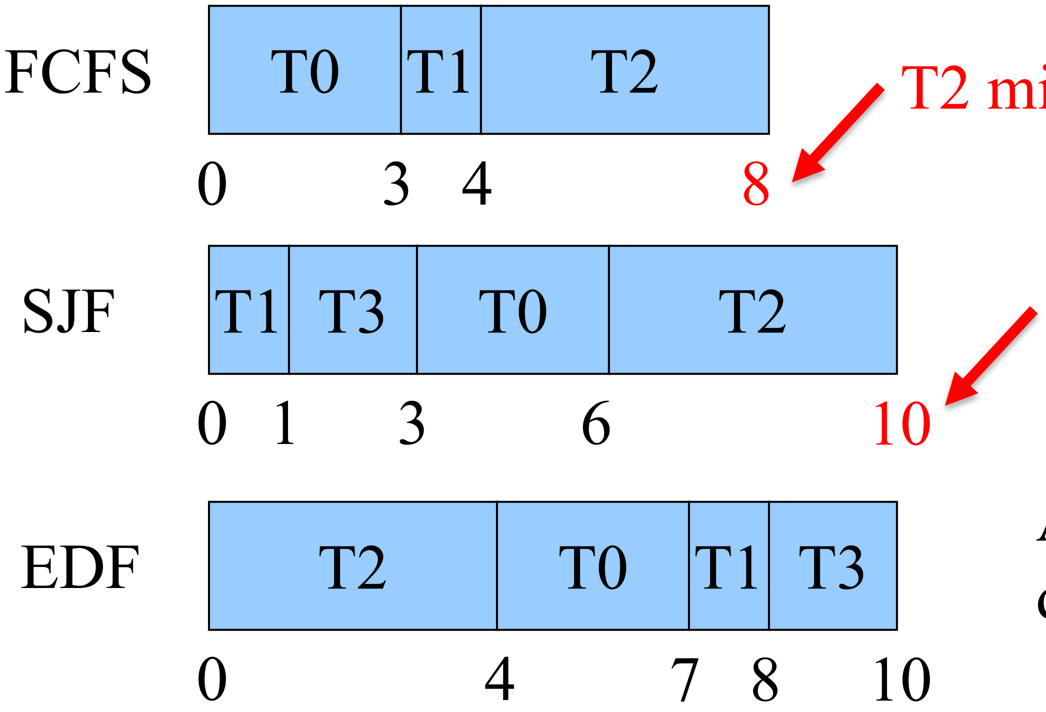
- Hard real time systems require that certain tasks *must* finish executing by a certain time, or the system fails
    - e.g. robots and self-driving cars need a real time OS (RTOS) whose tasks (actuating an arm/leg or steering wheel) must be scheduled by a certain deadline

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



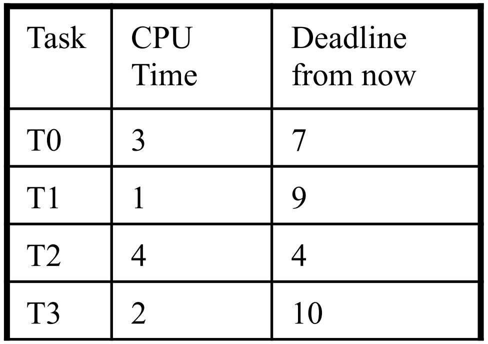
T2 missed deadline of t=4

T2 missed deadline of t=4

All tasks meet their deadlines (just barely)

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 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

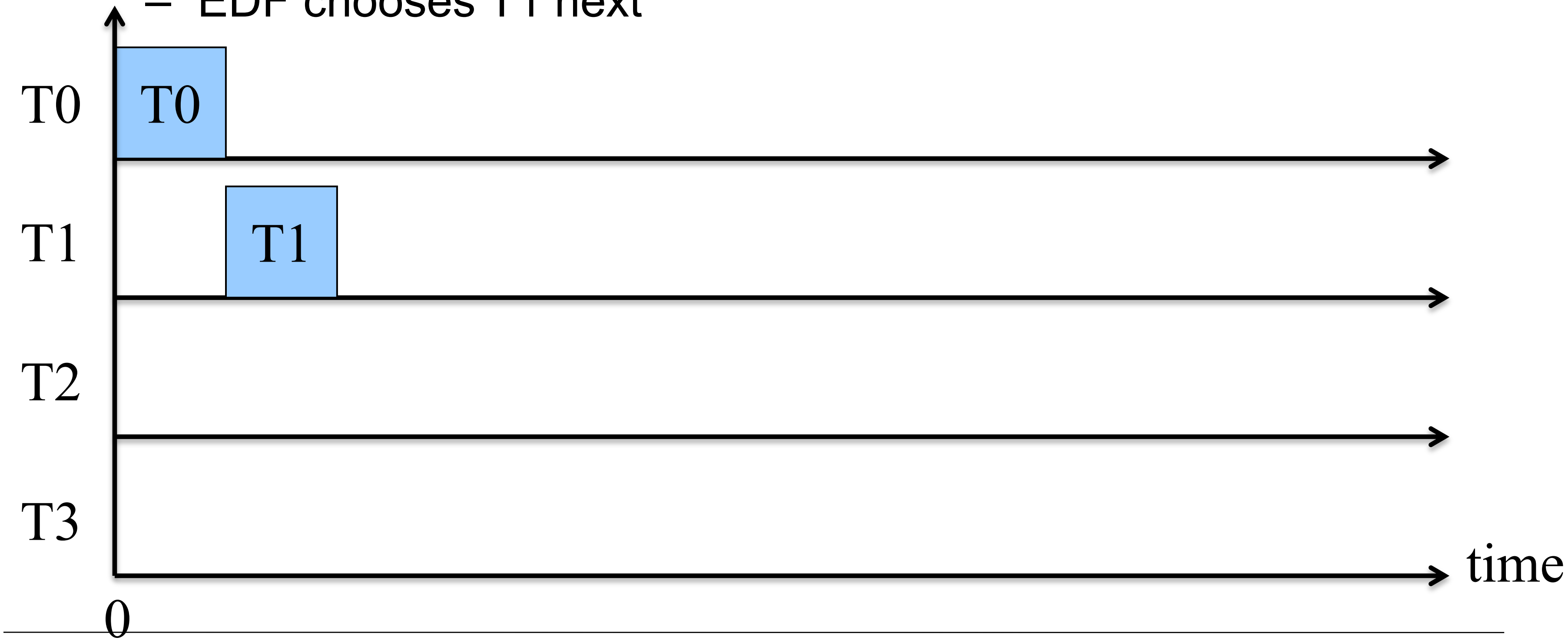
- Assume a preemptively time sliced system
  - A task arriving with an earlier deadline can preempt one currently executing with a later deadline.

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

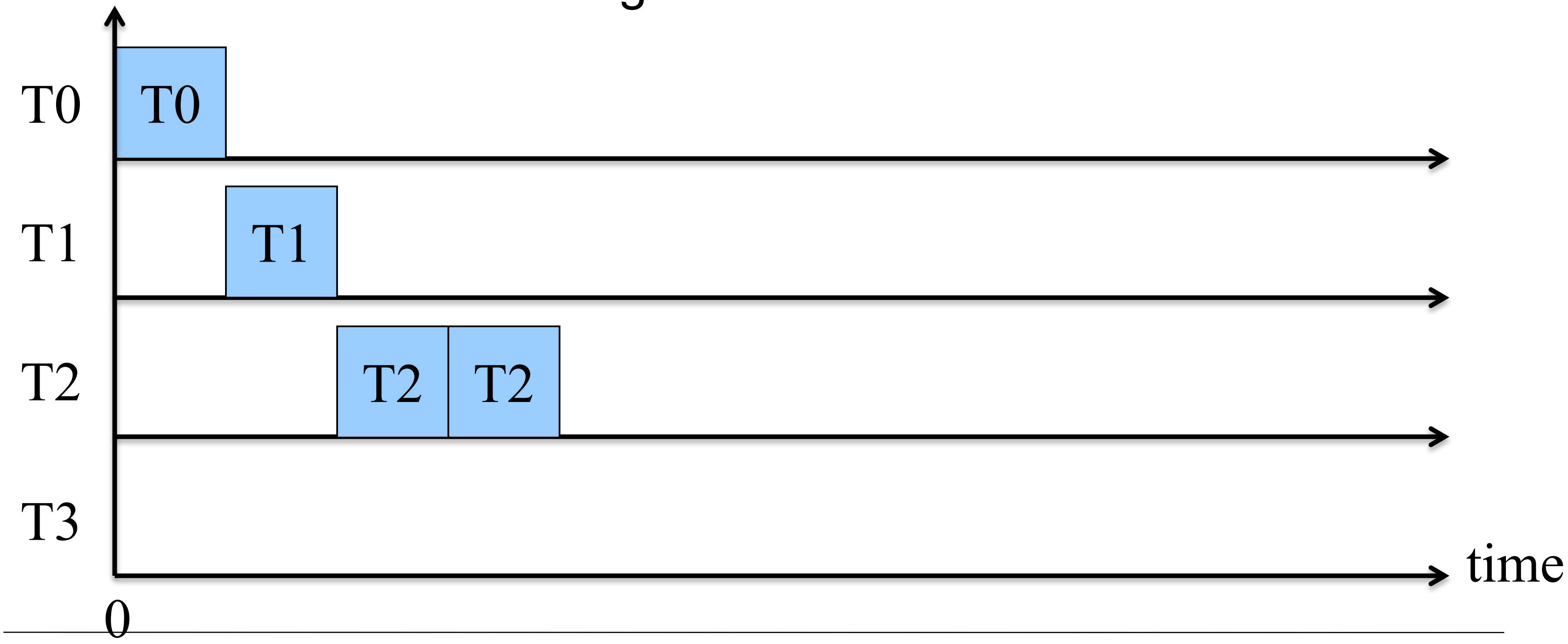
EDF and Preemption

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



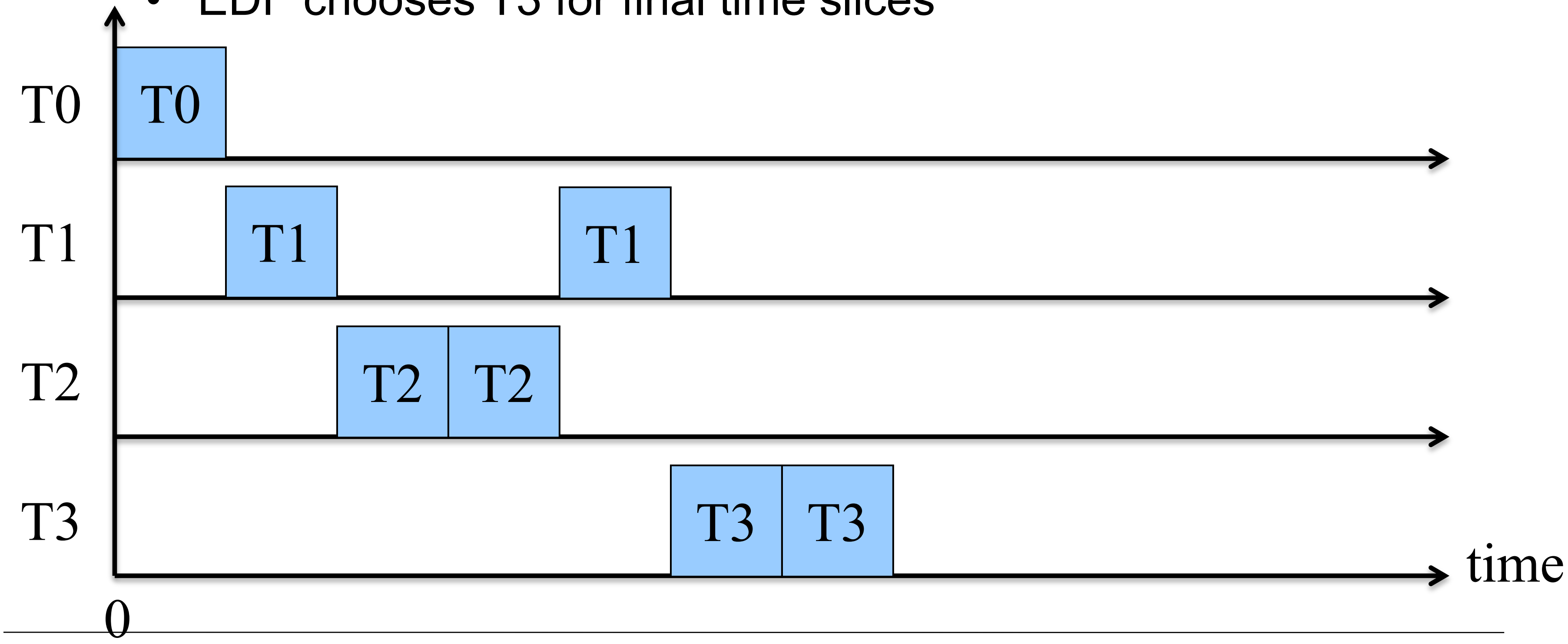
EDF and Preemption

- 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 = 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