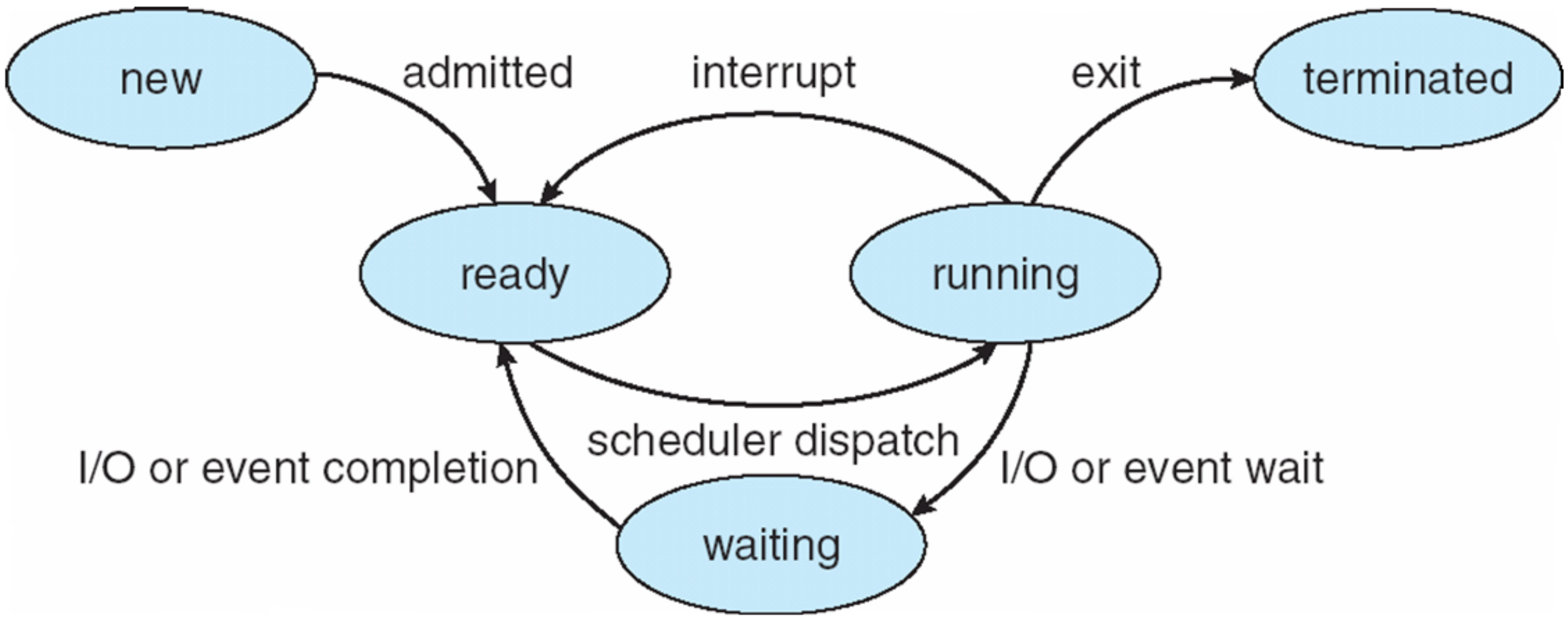
Chapter 6: Scheduling

CSCI 3753 Operating Systems

William Mortl, MS and Rick Han, PhD

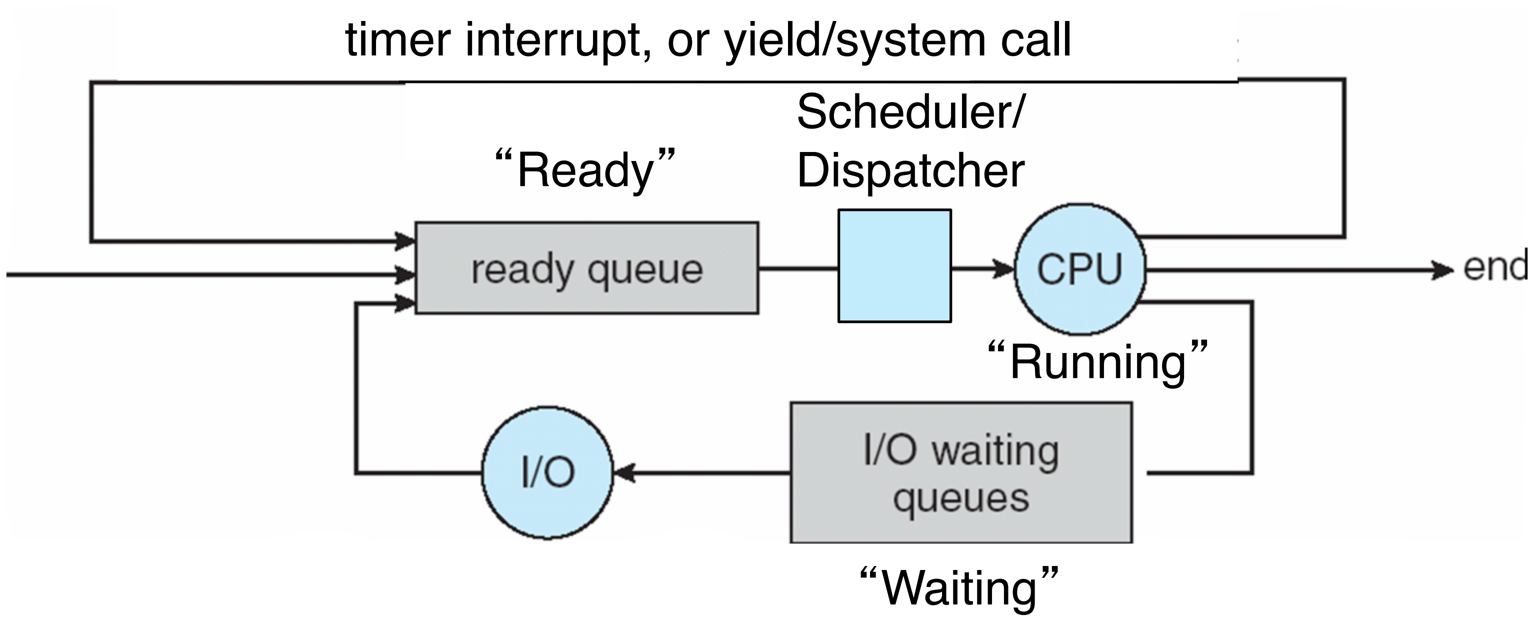
University of Colorado Boulder

Diagram of Process State



Also called “blocked” state

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

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

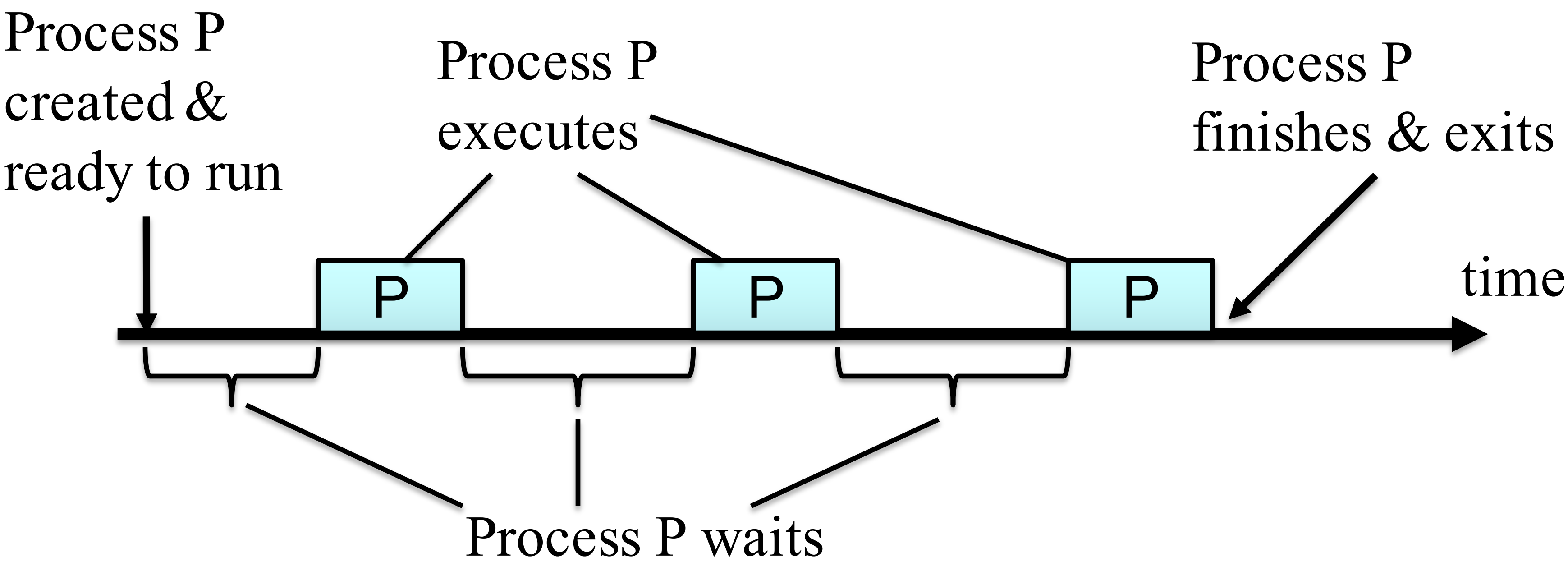
Policy

- Separate the mechanism of scheduling from the policy of scheduling

Context Switch Overhead

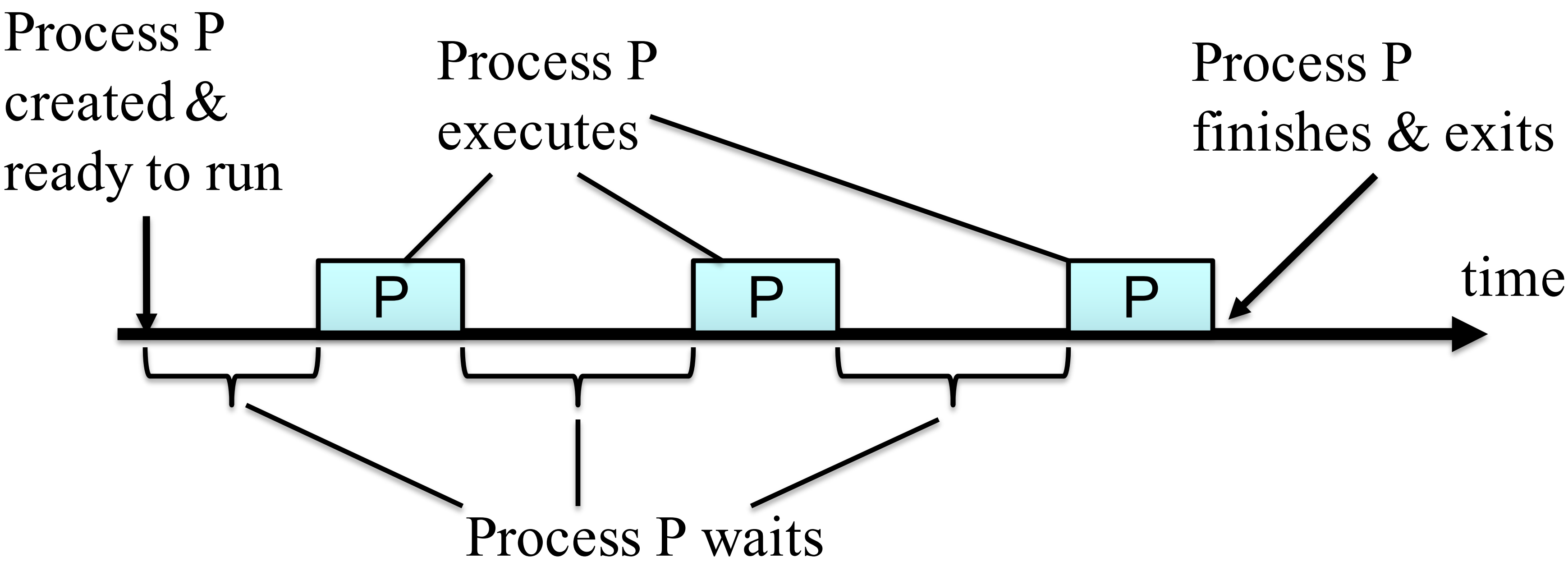
- Typically take 10 microseconds to copy register state to/from memory
  – on a 1 GHz CPU, that's 10000 wasted cycles per context switch!
- if the time slice is on the order of a context switch, then CPU spends most of its time context switching
  – Typically choose time slice to be large enough so that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

Scheduling Definitions



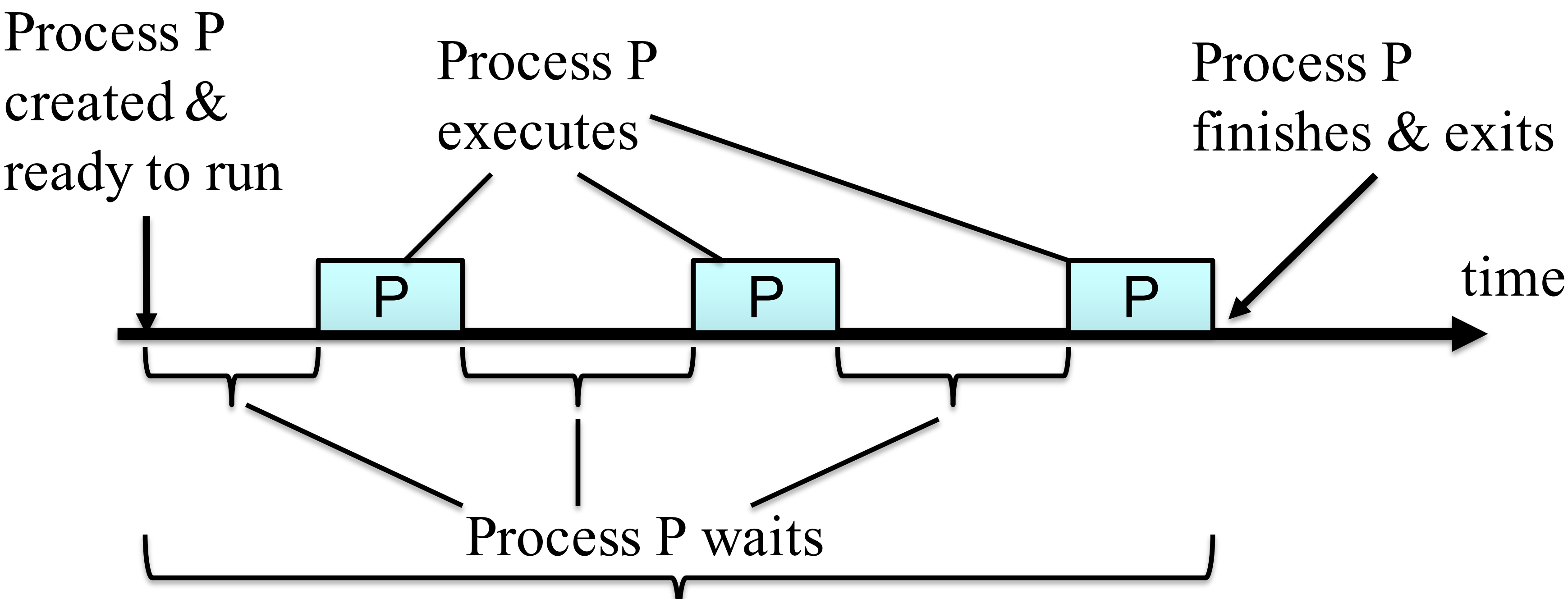
- *execution time* $E(P_i)$ = the time on the CPU required to fully execute process i
  - Sum up the time slices given to process i
  - Also called the “burst time” by textbook

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 doesn't include I/O waiting time

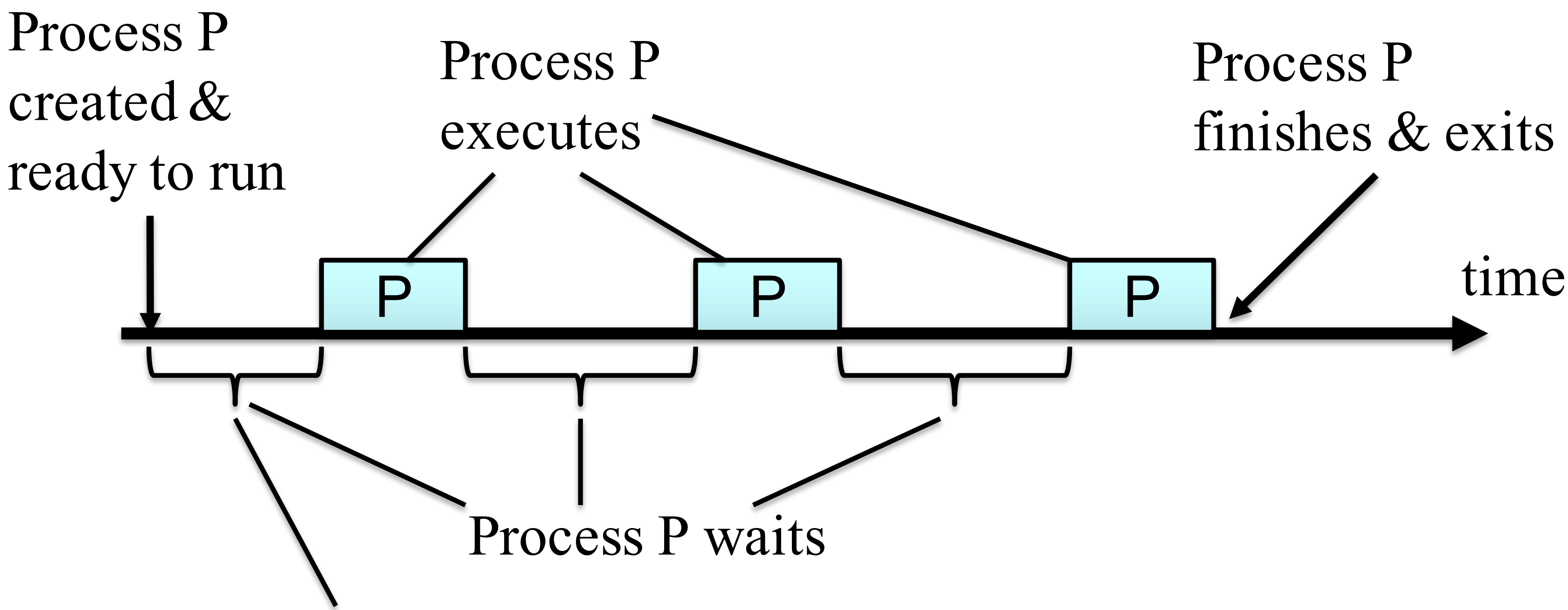
Scheduling Definitions



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)

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 run state)

Scheduling Criteria

- 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
- Scheduler implements a *scheduling policy*, that may adhere to one or more of the following goals:
  - maximize CPU utilization: 40% to 90%
  - maximize throughput: # processes completed/second
  - See next slide…

Scheduling Criteria

- Scheduling goals (continued):
  - minimize average or peak turnaround time: how long it takes to finish executing a process from 1st entry to final exit
  - min ave/peak waiting time: sum of time in ready queue
  - min ave/peak response time: time until first response
    - Some processes can generate early results, so if they get some CPU time quickly, they can start producing output sooner. A quick response time from the scheduler benefits such processes.

Scheduling Criteria

- Scheduling goals (continued):
  - maximize fairness
  - meet deadlines or delay guarantees
  - ensure priorities are adhered to

Scheduling Analysis

- We analyze various scheduling policies to see how efficiently they perform with respect to metrics like:
  - Wait time, turnaround time, response time, etc.
- Some algorithms will be optimal in certain metrics
- To simplify 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, e.g round robin.

FCFS Scheduling

- First Come First Serve: order of arrival dictates order of scheduling
  - Nonpreemptive, processes execute until completion
- If processes arrived in order P1, P2, P3 before time 0, then *Gantt chart* of CPU service time is:

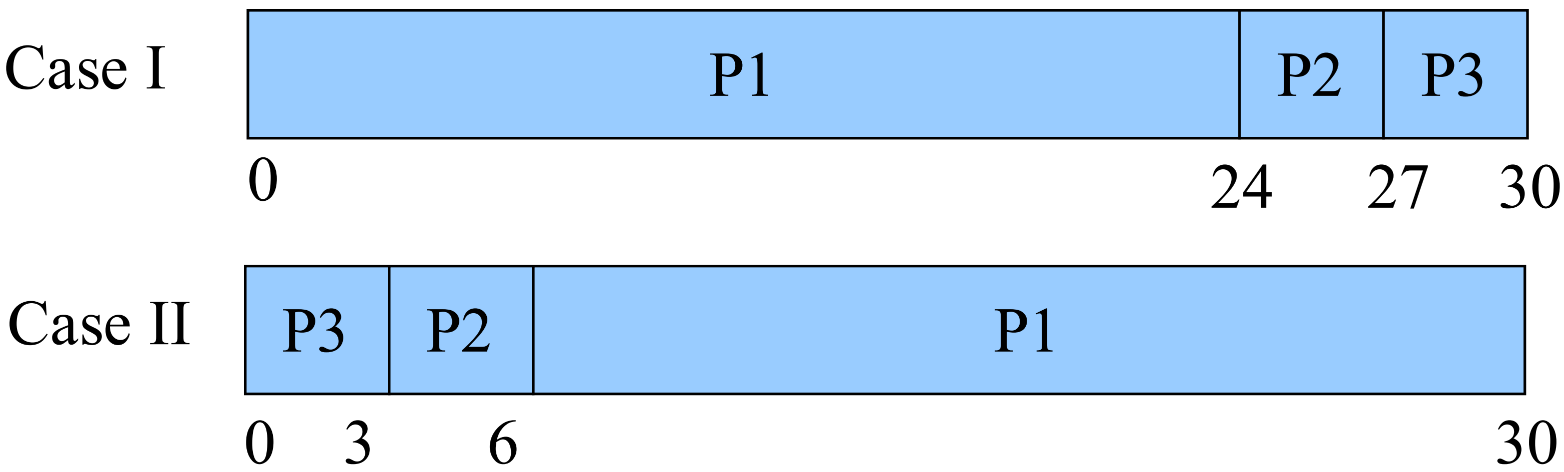
Process	CPU Execution Time
P1	24
P2	3
P3	3

FCFS Scheduling (2)

- If processes arrive in reverse order P3, P2, P1 around time 0, then Gantt chart of CPU service time is:

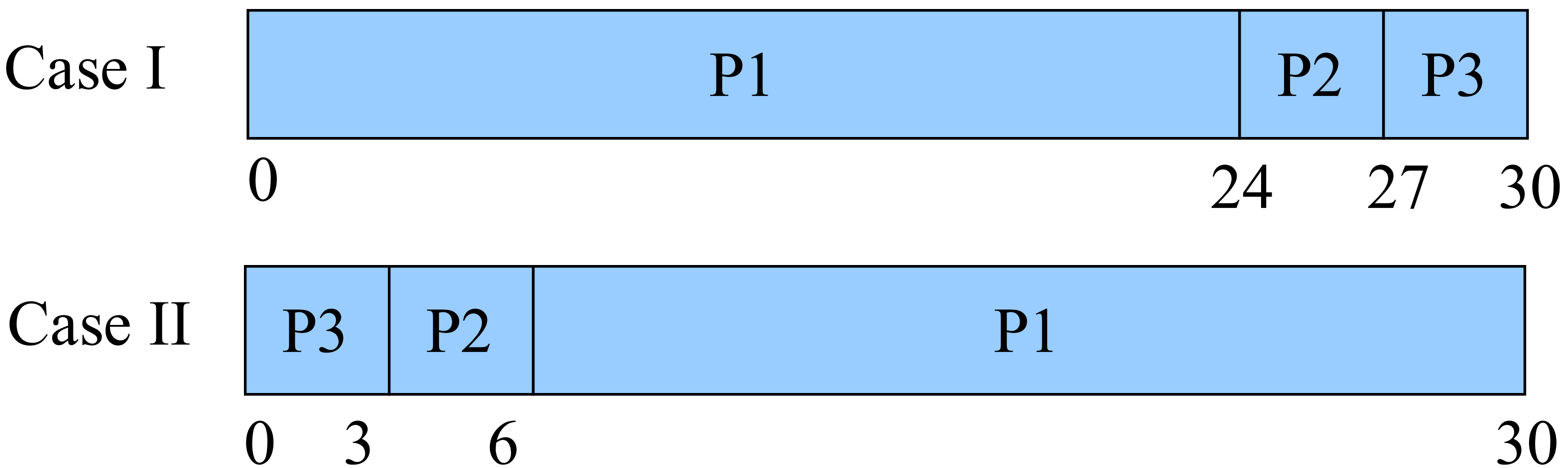
Process	CPU Execution Time
P1	24
P2	3
P3	3

FCFS Scheduling (3)



- 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 (3)



- 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.

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: moving a long process before a short one increases the wait time of short processes a lot.
    - Conversely, moving long process to the end decreases wait time seen by short processes
      - Also, the impact of the wait time on long processes moved towards the end is minimal

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.

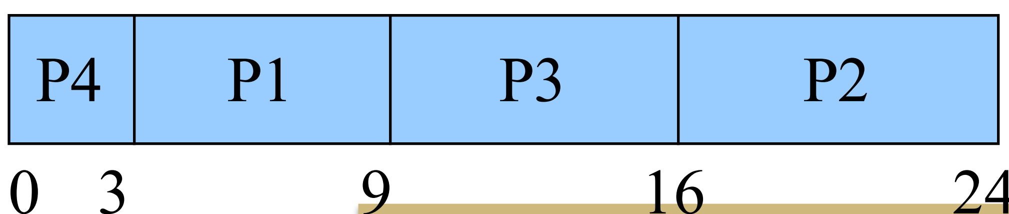
Shortest Job First (SJF) Scheduling

- *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

- 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 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 Pi takes as CPU(n,i).
  - For each process Pi, 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):

  $EstCPU(n+1,i) = \alpha*CPU(n,i) + (1-\alpha)*EstCPU(n,i)$

  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

- 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