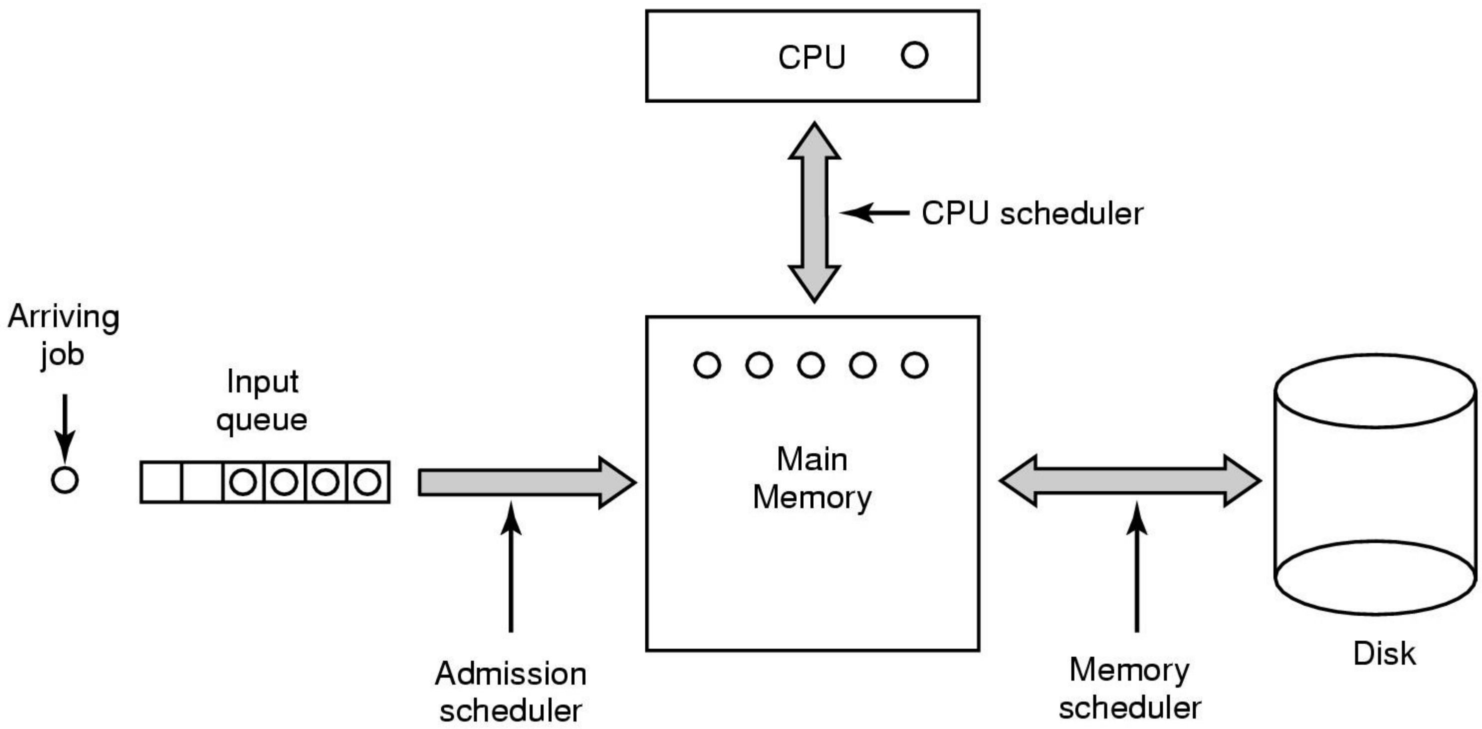
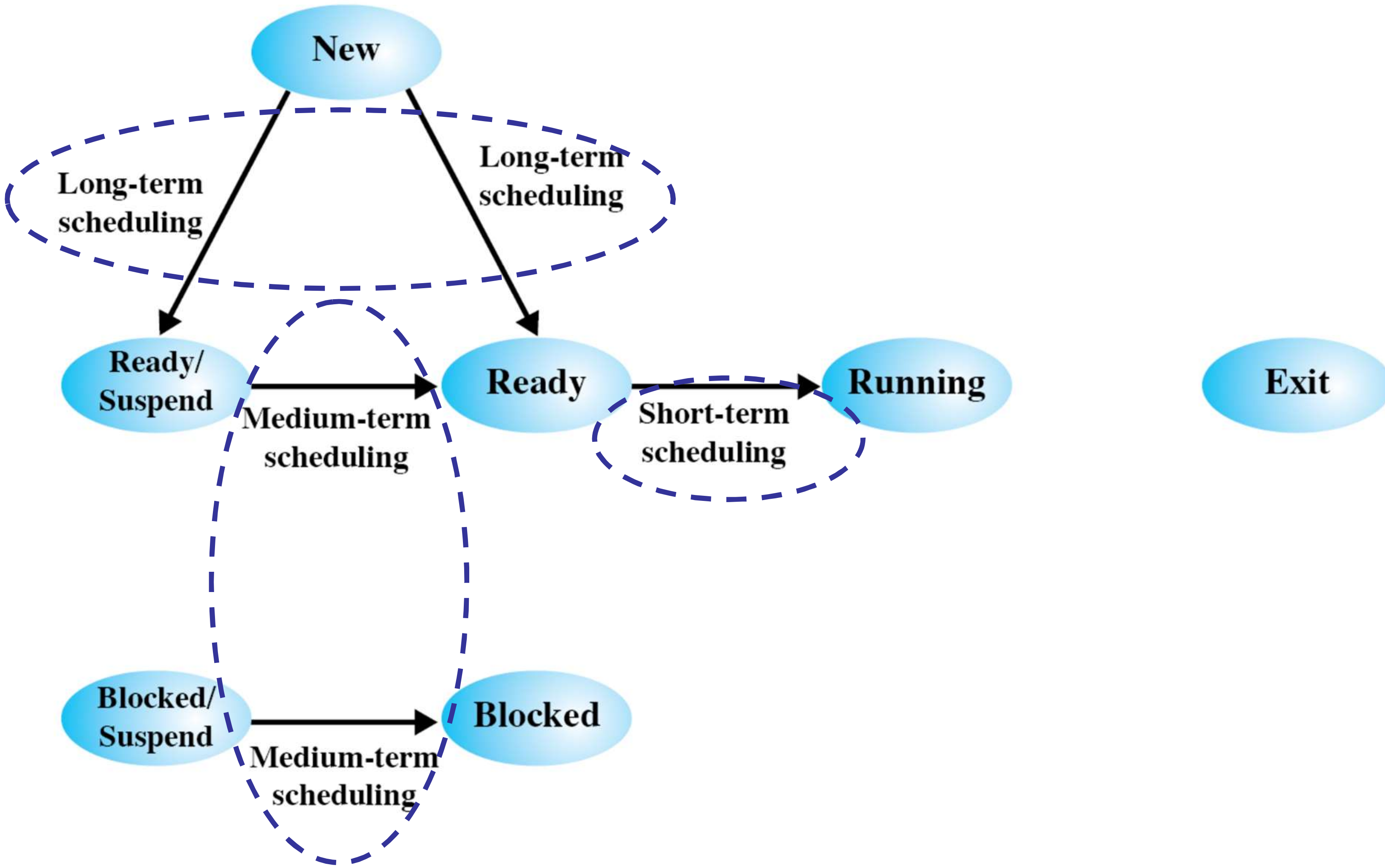
Operating Systems

3. Process Scheduling

Three Level Scheduling

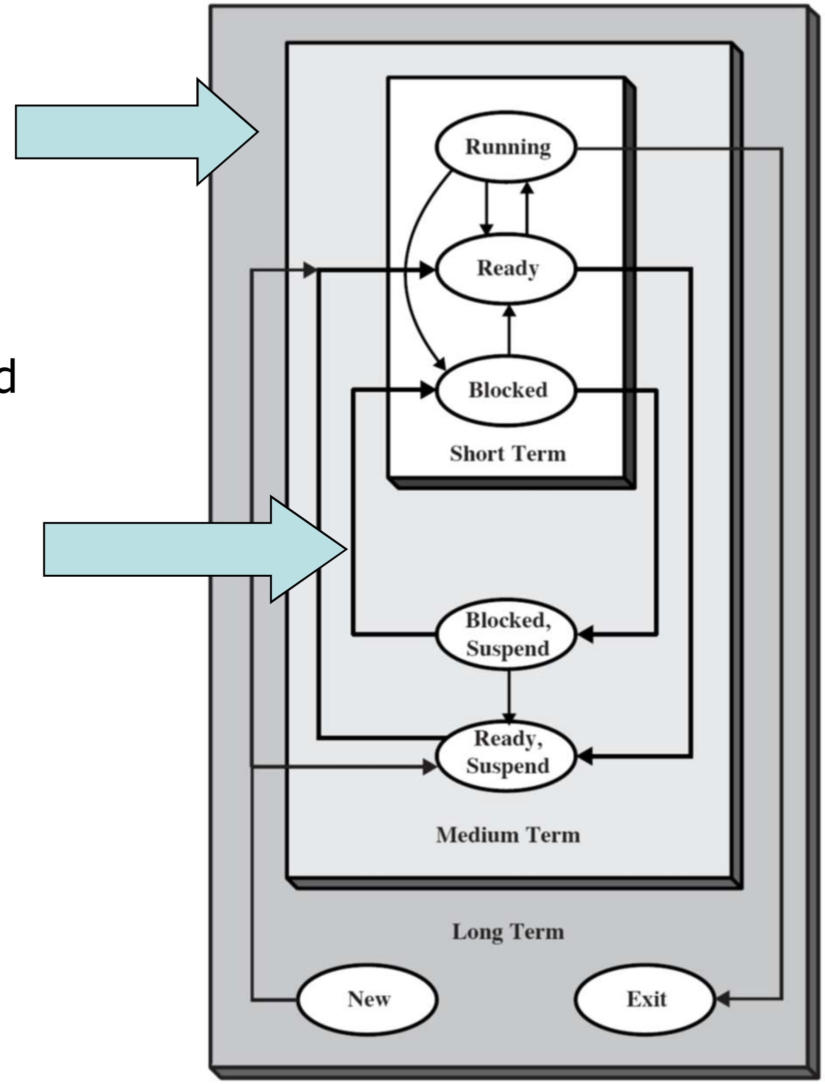


Scheduling and Process Transitions



Long- and Medium-Term Schedulers

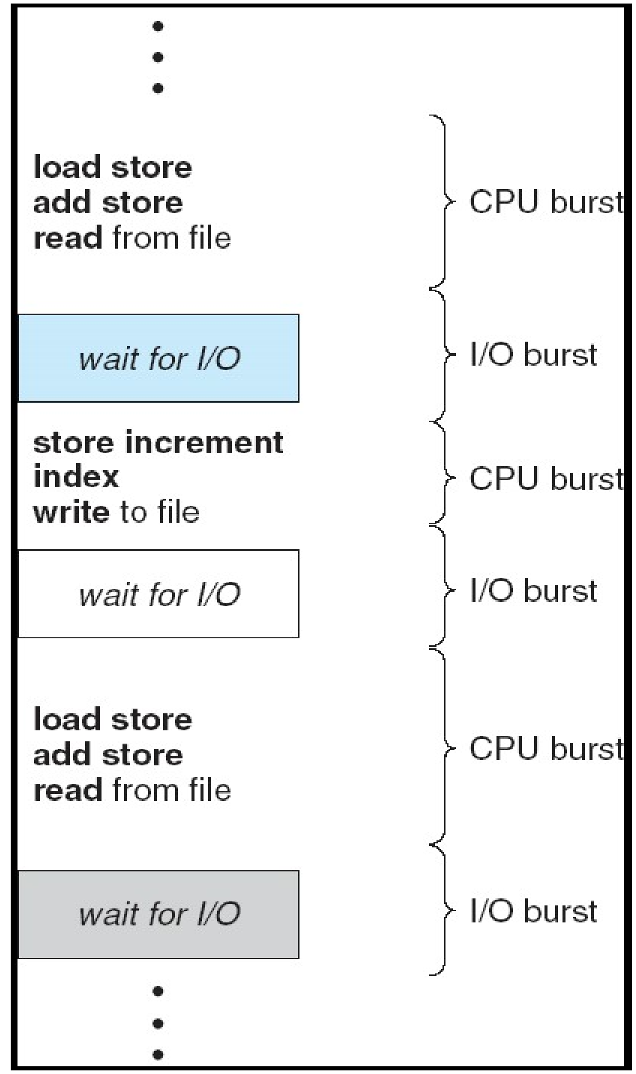
- Long-term scheduler
  - Determines which programs are admitted to the system
    - ➢ I.e., to become processes
  - Requests can be denied
    - ➢ E.g., in case of thrashing or overload
- Medium-term scheduler
  - Decides when/which processes to suspend/resume
- Both control the degree of multiprogramming
  - More processes, smaller percentage of time each process is executed



CPU–I/O Burst Cycle

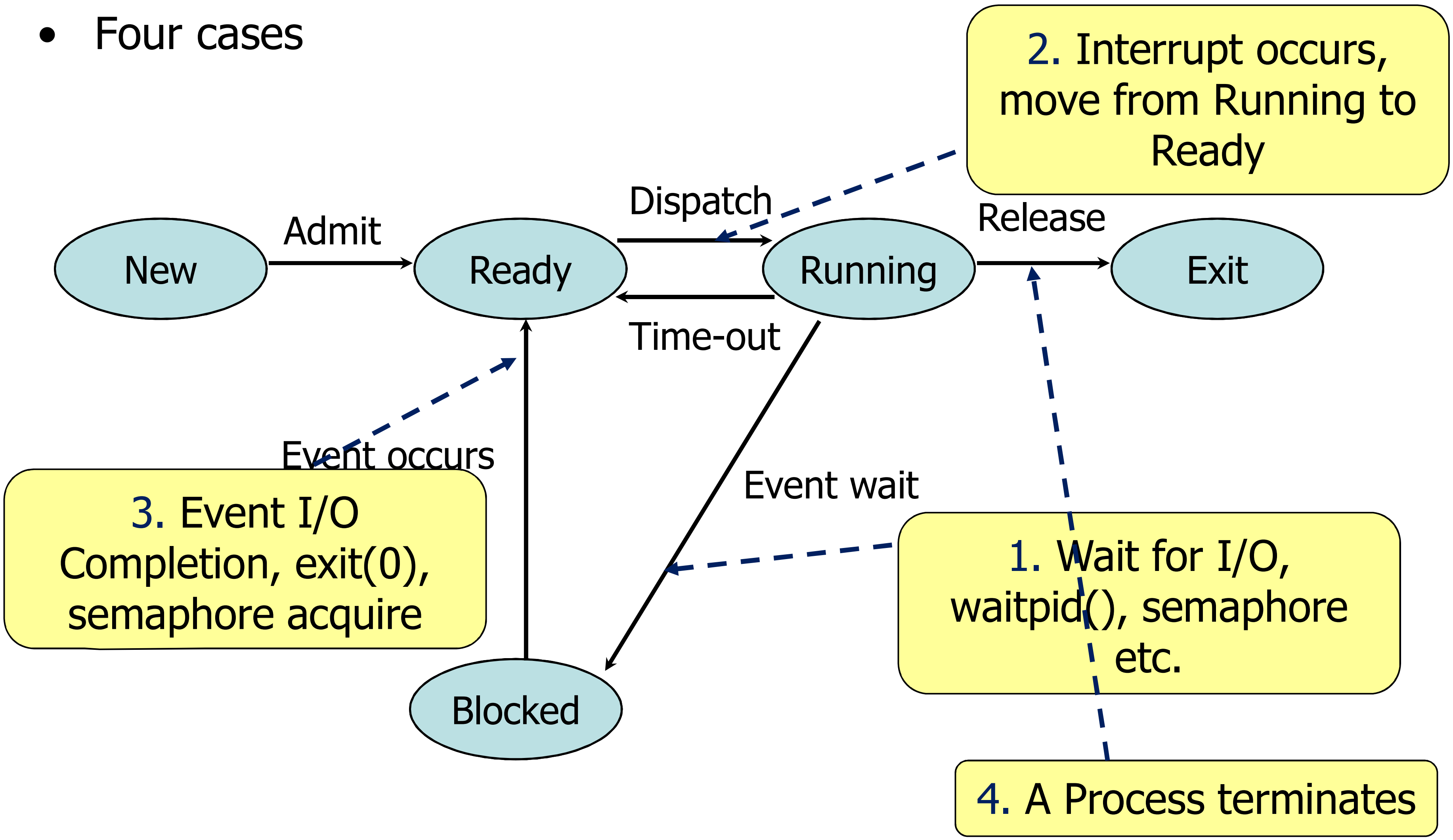
- Process execution consists of a cycle of
  - CPU execution and
  - I/O wait

- A process may be
  - CPU-bound : Long CPU bursts
  - IO-bound : Short CPU bursts



Short-Term Scheduler: Selection of a New Process

- Four cases



Decision Mode

Non preemptive

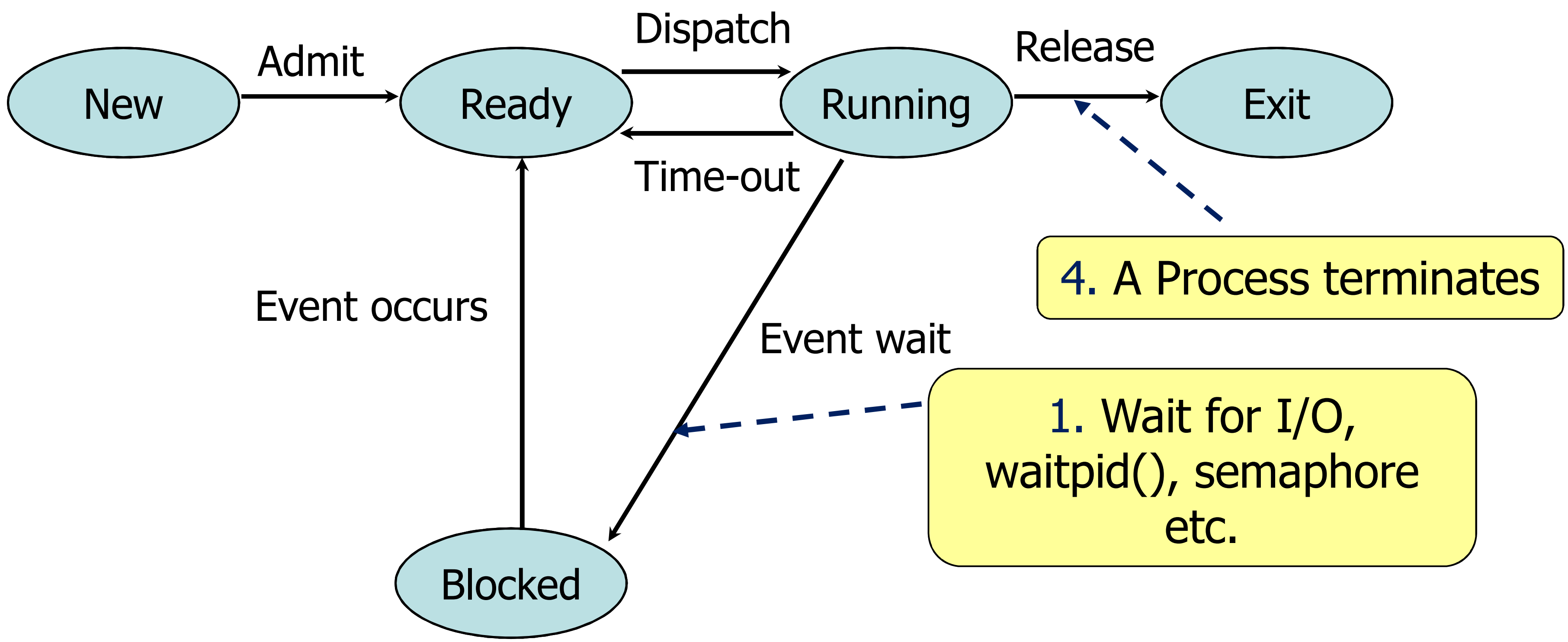
- Once a process is in the running state, it will continue until it terminates or blocks itself for I/O, child process, semaphore etc.

Preemptive

- Currently running process may be interrupted
- Now transition from Running to Ready state possible
- Prevents a single process to monopolize the processor for very long

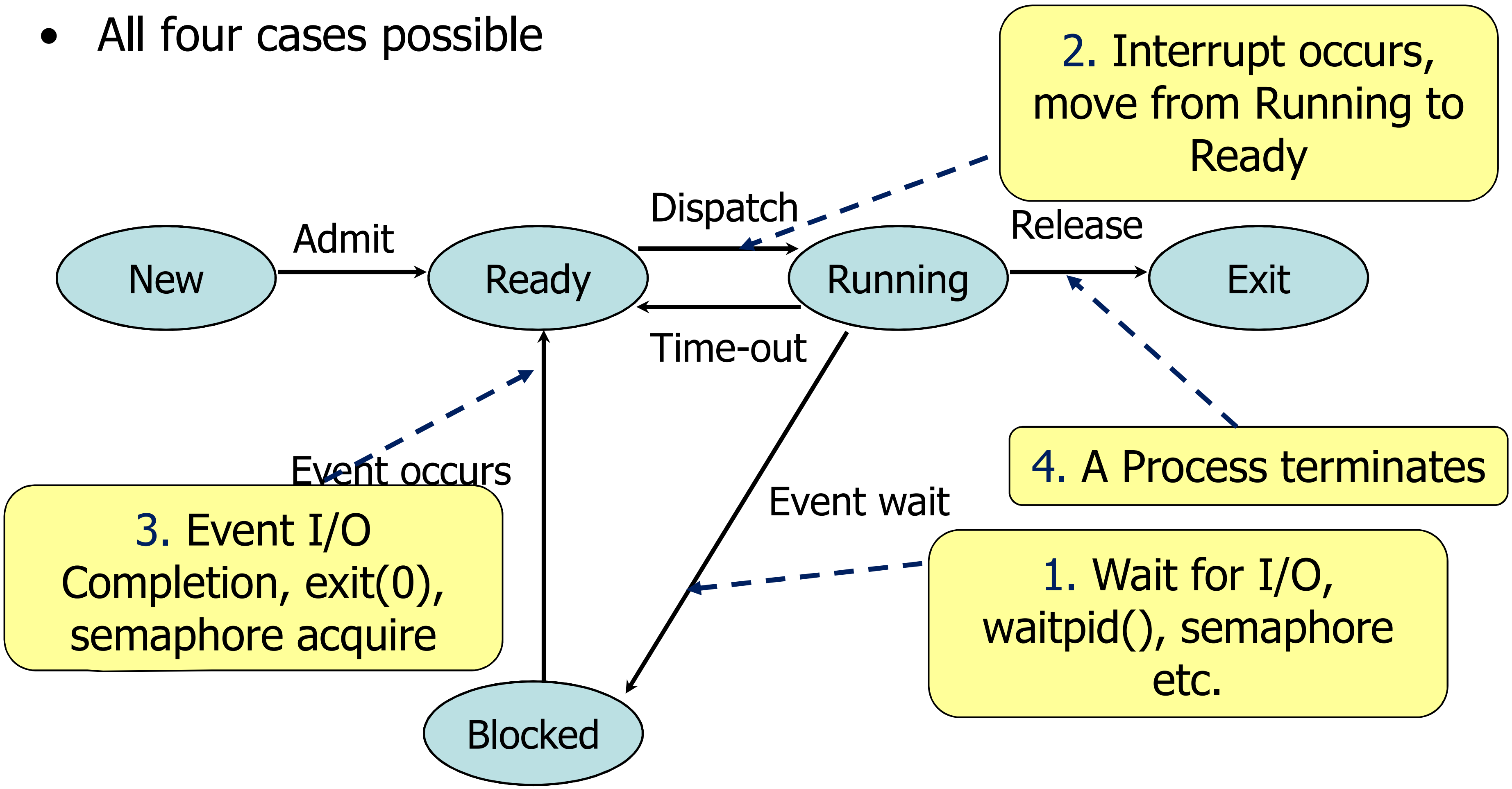
Non Preemptive Scheduling

- Only the case 1 and 4



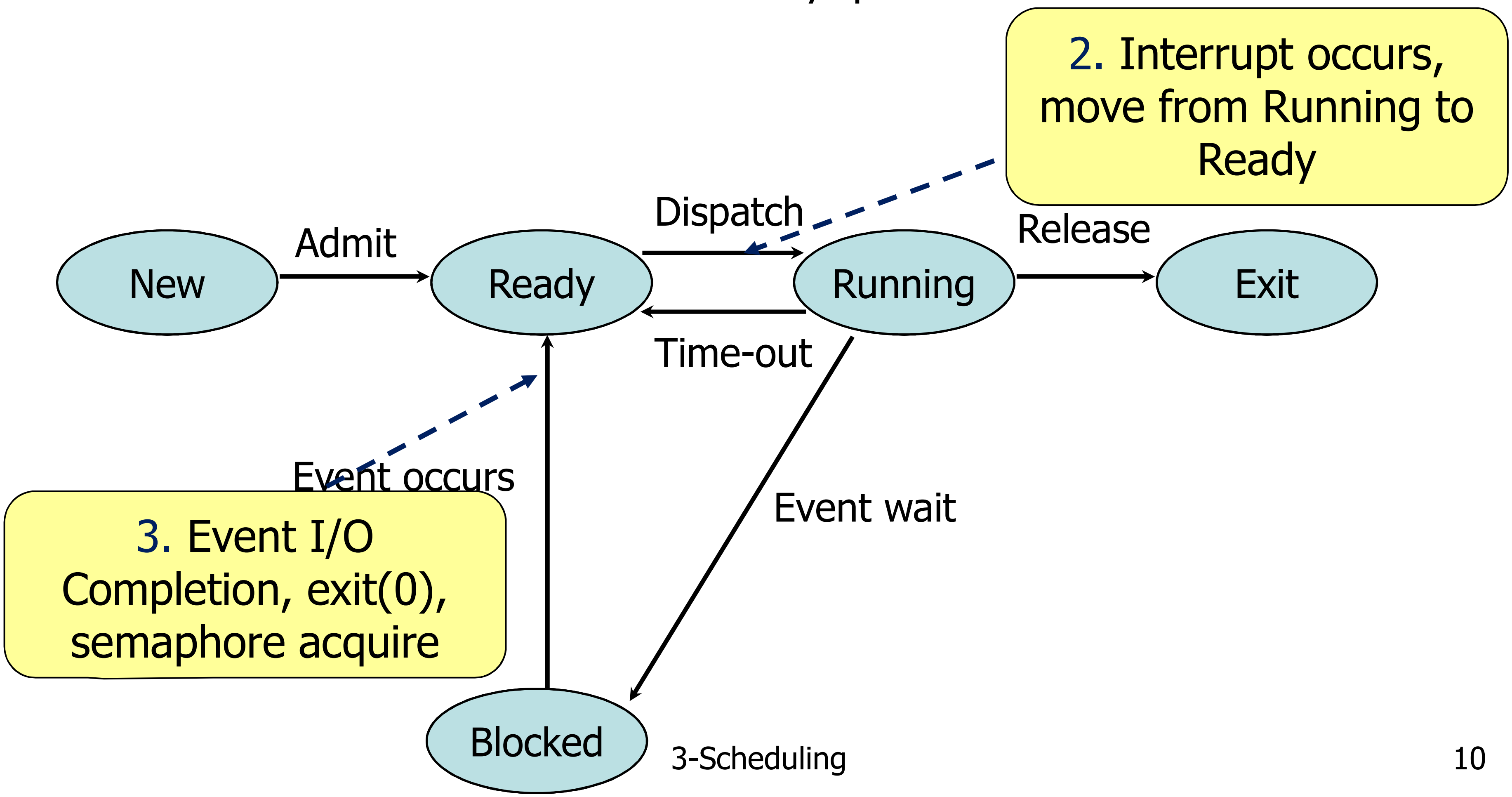
Preemptive Scheduling

- All four cases possible



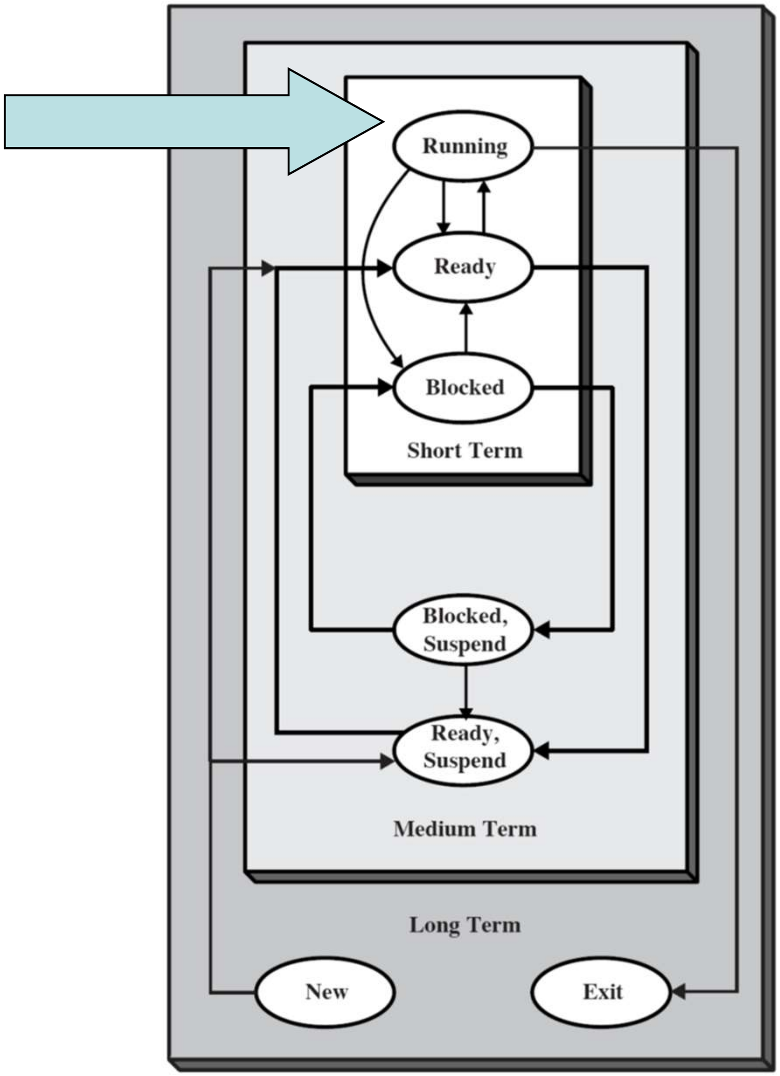
Preemptive Scheduling

- In case of 2 and 3, there is a choice
  - Whether to continue, with the same process
  - Or select a new one from the ready queue



Short-Term Scheduler

- Decides which process will be dispatched
  - Invoked upon Clock interrupts, I/O interrupts, Operating system calls, Signals
- Dispatch latency: Period of time the dispatcher needs to stop one process and start another running
- Dominating factors
  - Switching context
  - Selecting a process to dispatch
- Arrival time: Time when a process is admitted to the system
- Service Time: Period of time a process executes in running state



Scheduling Issues

- Fairness
  - Don’t starve process
- Priorities
  - Most important first
- Deadlines
  - Task X must be done by time t
- Optimization
  - Throughput, response time
- Scheduler Efficiency
  - Overhead, e.g., context switching, computing priorities, …
- Reality - No universal scheduling policy
  - Many models

Scheduling Criteria and Optimization goals

- CPU utilization
  - Percentage of time CPU is busy
  - Keep CPU as busy as possible

- Throughput
  - No. of processes that complete their execution per time unit

- Turnaround time (TAT)
  - Amount of time to execute a particular process
  - TAT = Time of completion – Arrival time
    - Execution + all the waiting for resources including CPU
    - Involves also the IO schedulers
  - Not a good criteria for interactive systems

Scheduling Criteria and Optimization goals

- Waiting time
  - Waiting Time = Sum of the periods spent waiting in the Ready queue
  - Scheduling Algorithm does not effect the waiting time in Block queue
    - ➢ Only effect the waiting time in the Ready queue

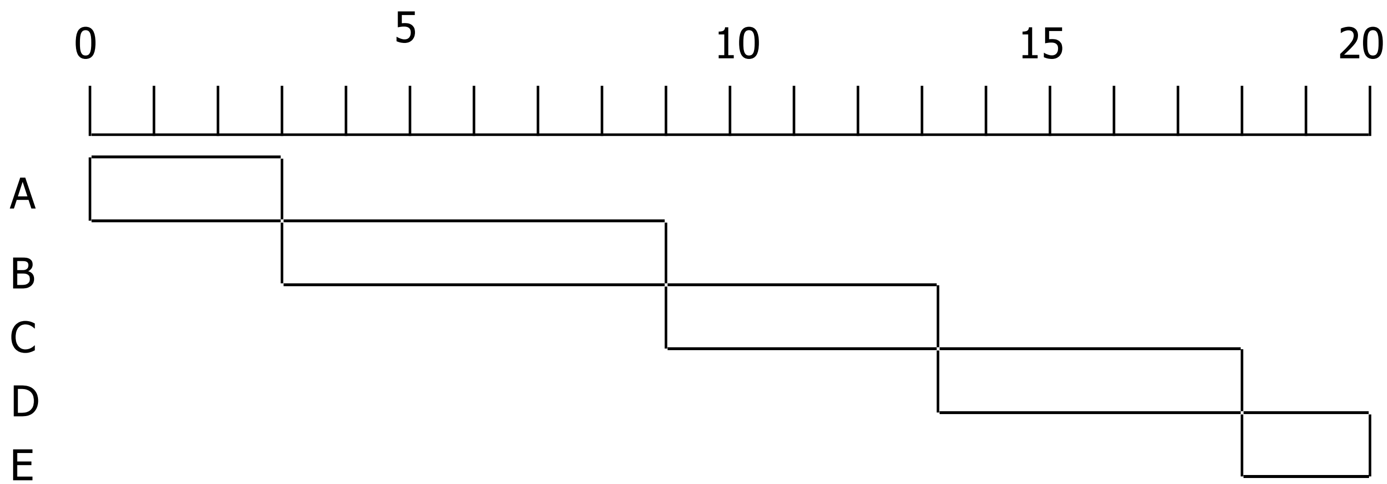
- Response time
  - Amount of time from submission of the request until first response
  - Response time = First response – Arrival time
    - ➢ Execution + waiting time in ready queue

- Summary
  - Maximize: CPU utilization, Throughput
  - Minimize: Turnaround time, Waiting time, Response time

First-Come-First-Served (FCFS)

- Non-preemptive
- Favors CPU-bound processes
- A short process may have to wait very long before it can execute
- Convoy effect – Many I/O processes may stuck behind CPU-bound process

Process	Arrival Time	Burst Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

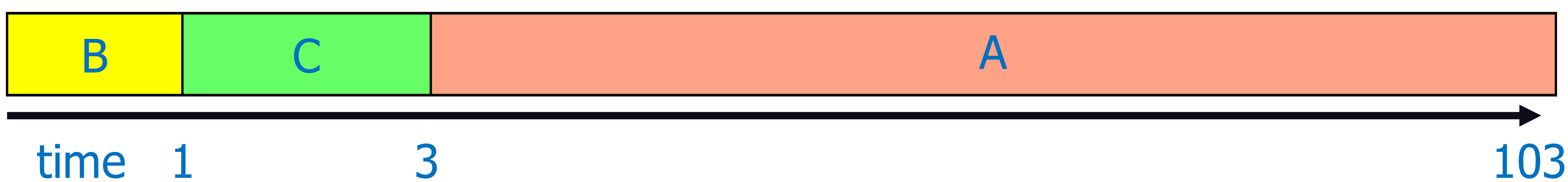


First-Come-First-Served (FCFS)

- Wait time depends on arrival order
- Worst case: long job arrives first
- Example: Three processes with service times: A=100, B=1, C=2
  - Case 1: Processes arrive in the order A, B, C

Average Waiting Time = (0 + 100 + 101) / 3 = 67

  - Case 2: Processes arrive in the order B, C, A



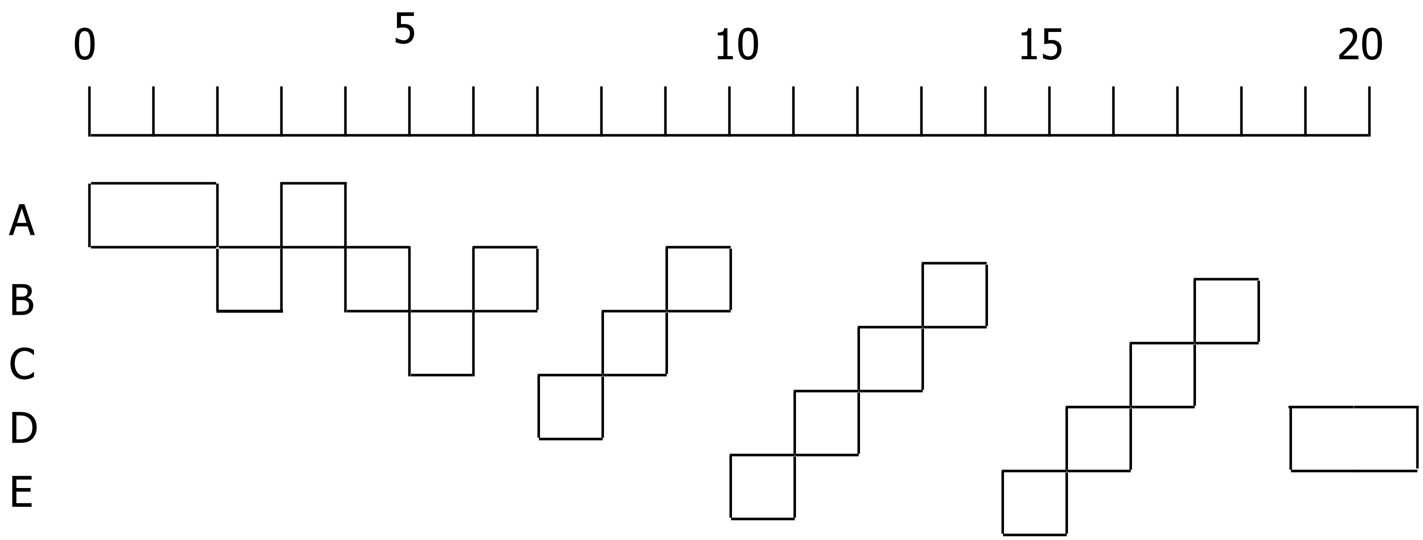
Average Waiting Time = (0 + 1 + 3) / 3 = 1.33

First-Come-First-Served (FCFS) – Convoy Effect

Consider one CPU-bound and many I/O-bound processes

1. The CPU intensive process blocks the CPU
2. A number of I/O intensive processes stuck behind this process
   - Leaving the I/O devices idle
   - Low I/O device utilization
3. When the CPU-bound process finally issues I/O request
   - I/O processes pass through the CPU quickly
   - Leaving the CPU idle while everyone queues up for I/O
4. The cycle repeats itself when the CPU intensive process gets back to the ready queue

Round Robin



Process	Arrival Time	Burst Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

- Preemption based on clock (interrupts on time slice or quantum -q- usually 10-100 msec)
- Fairness: for n processes, each gets $1/n$ of the CPU time in chunks of at most q time units
- Performance
  - q large $\Rightarrow$ FCFS
  - q small $\Rightarrow$ overhead can be high due to context switches

Round Robin – Example with Time Quantum = 4

Process	Burst time
P_1	24
P_2	3
P_3	3

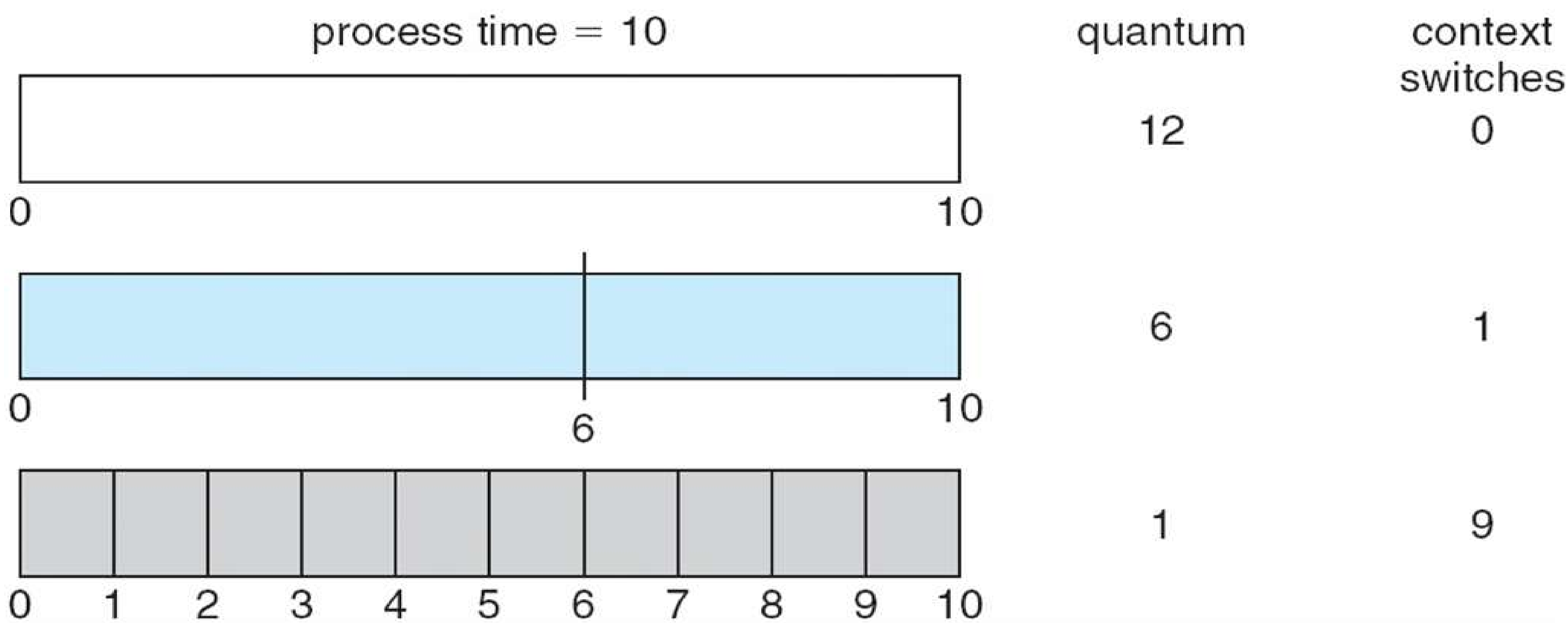
- Process may have CPU burst of less than 1 time quantum
  - Process itself release the CPU voluntarily
  - Dispatcher will proceed to the next process in ready queue

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

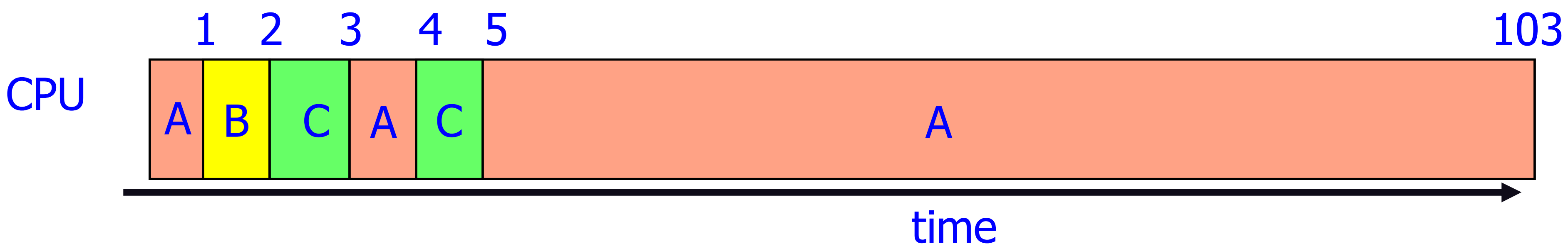
Time Quantum and Context Switch Time

- Quantum –q- should be large compared to context switch time
- Quantum –q- usually 10ms to 100ms, context switch < 10 usec



Round Robin

- Low average waiting time when job lengths vary
- Example: Three processes with service times: A=100, B=1, C=2



- Average waiting time
  - (3 + 1 + 3) / 3
- Average completion (Turn around time)
  - (103 + 2 + 5) / 3

Round Robin – Disadvantage

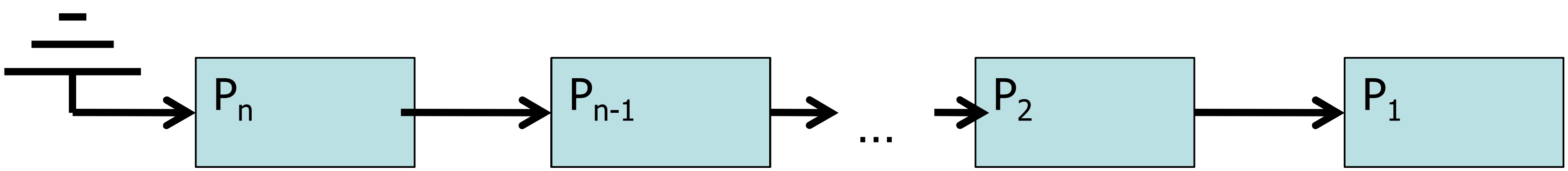
- Good for Varying sized jobs
- But what about same-sized processes?
- Example: Two processes with same service time =100

- Average completion time, i.e., Turn Around Time (TAT)?
  - (200 + 200) / 2 = 200
- How does this compare with FCFS for same two jobs?
  - (100 + 200) / 2 = 150

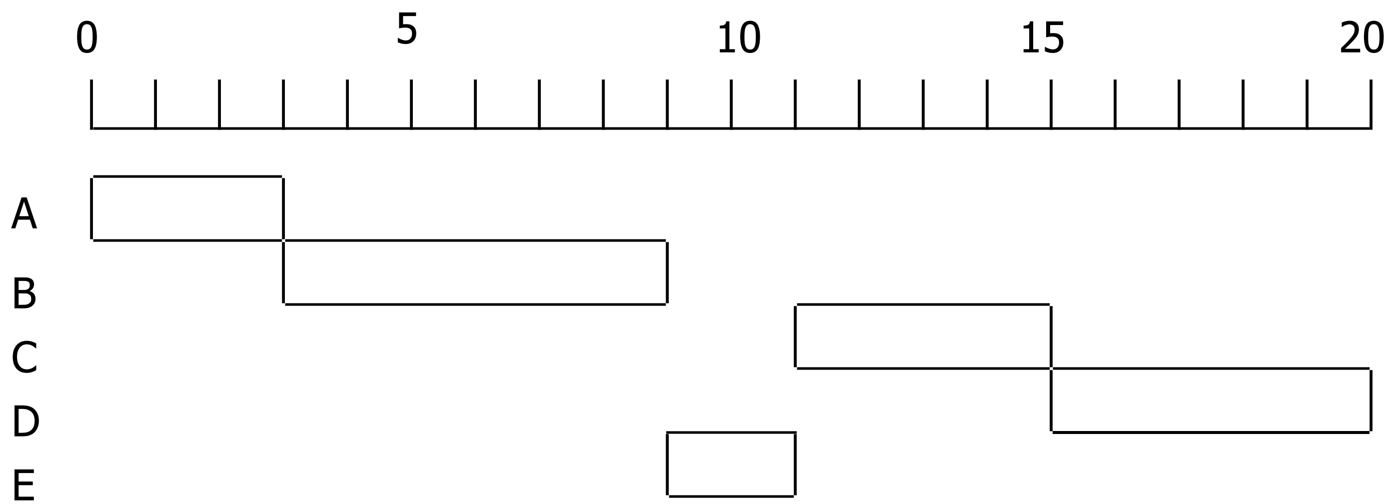
Dispatcher Performance: Round Robin and FCFS

- Maintain linked list
  - Dequeue: Remove head
  - Enqueue: Append tail

- Cost
  - Enqueue: O(1)
  - Dequeue: O(1)



Shortest Job First (SJF) – Non-preemptive

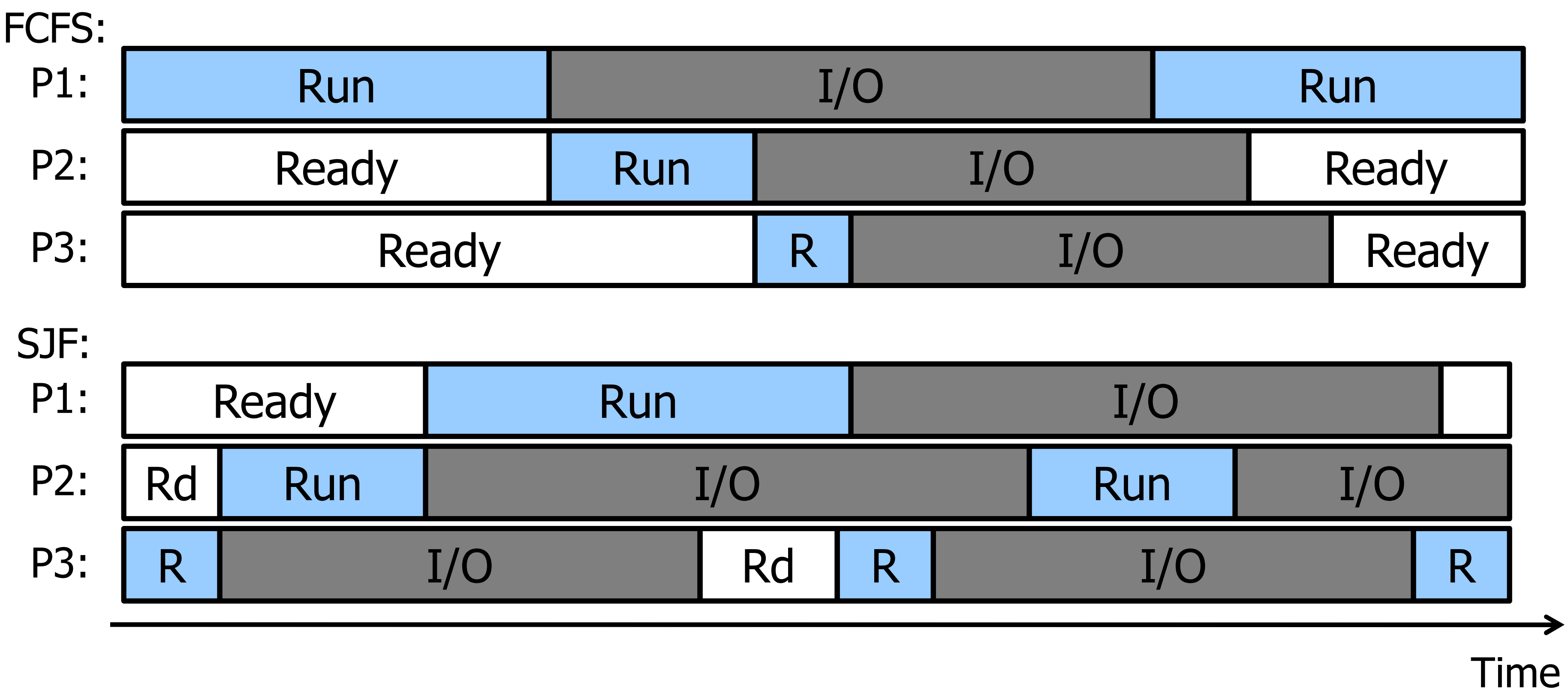


Process	Arrival Time	Burst Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

- Associate with each process the length of its next CPU burst
  - Short process jumps ahead of longer processes
- Avoid convoy effect

Performance of SJF Scheduling

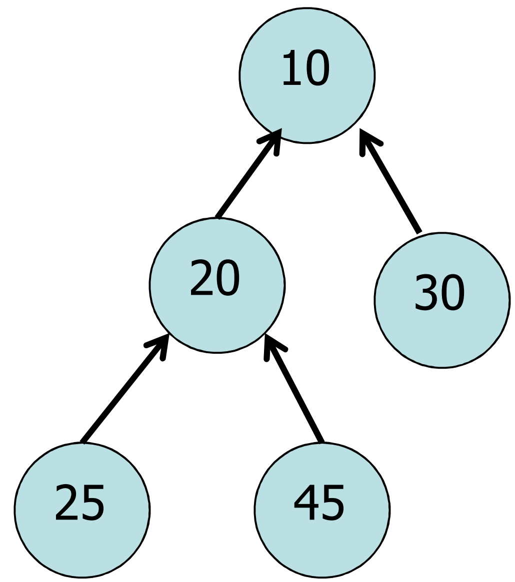
- Gives high throughput
- Gives minimum average waiting time for a given set of processes



Dispatcher Performance: SJF

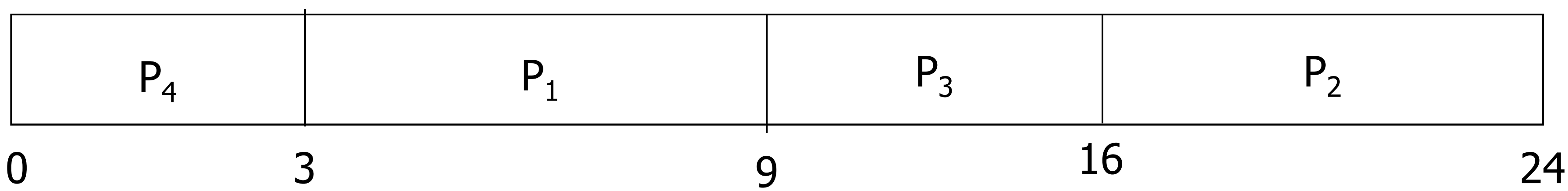
- Dispatcher utilizes heap operations
  - Enqueue: `insert_heap(`P_i`, `t_i` )`
  - Dequeue: `del_min()`
  - Decrease the priority dynamically

- Performance depends on implementation
  - Binary Heap
    - Insert : $O(\log_2 n)$
    - Dequeue: $O(\log_2 n)$



SJF (Non-preemptive) – Example

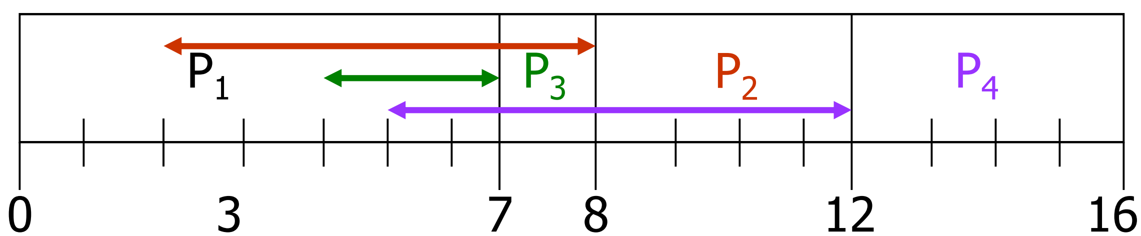
Process	Burst time
P_1	6
P_2	8
P_3	7
P_4	3



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

SJF (Non-preemptive) – Example

Process	Arrival time	Burst time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



- Average waiting time
  - $(0 + 6 + 3 + 7)/4 = 4$

Shortest Remaining Time First

- Shortest Job First (SJF) can be preemptive or non-preemptive
  - Preemptive SJF is also called Shortest Remaining Time First

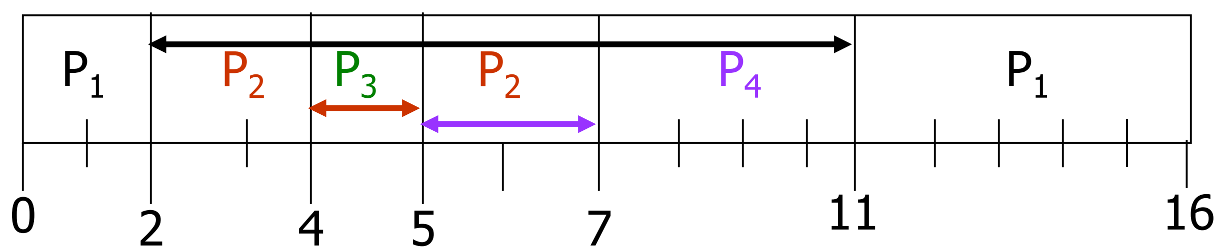
Overall Idea

- While an existing process is executing
  - A new process arrives at the ready queue

- Existing process is preemptive
  - If the CPU burst of the newly arrived process is shorter than the remaining execution time of existing process

Shortest Remaining Time First – Example

Process	Arrival time	Burst time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



- Average waiting time
  - $(9 + 1 + 0 + 2)/4 = 3$

Shortest Remaining Time First – Example

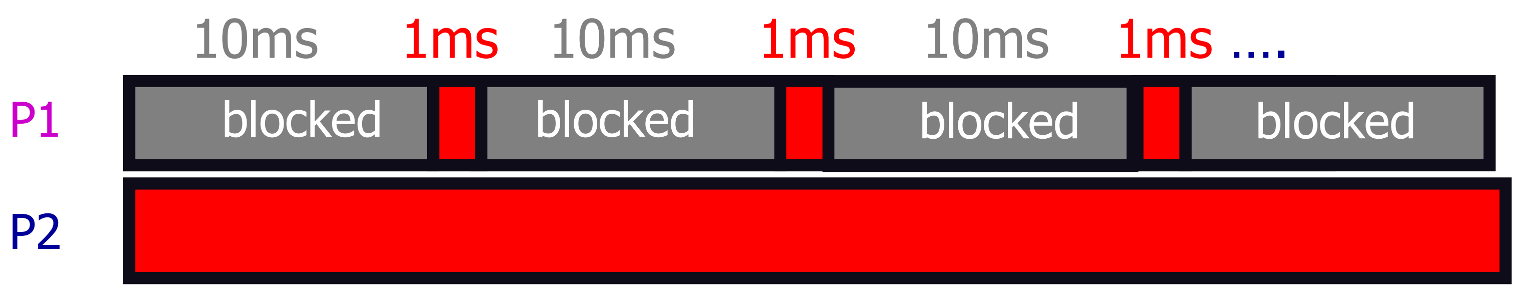
Process	Arrival time	Burst time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

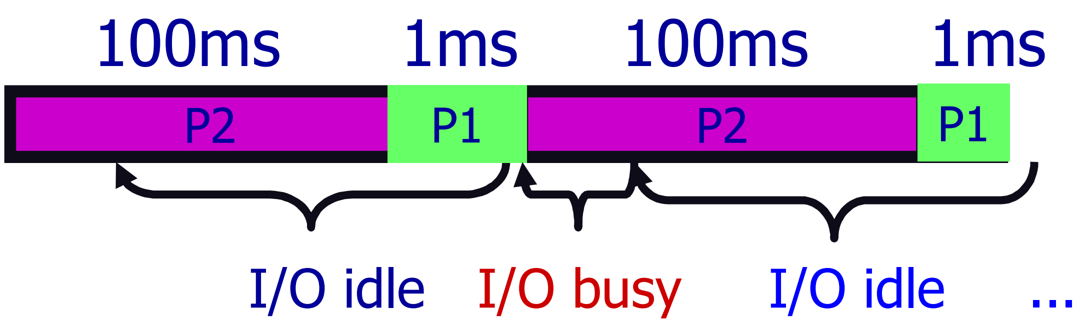
P_1	P_2	P_4	P_1	P_3	
0	1	5	10	17	26

- Average waiting time
  - [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5

SJF vs. RR

- Two processes P1, P2
  - Round robin with 100ms time slice





- SJF Offers better I/O utilization

Properties of SJF

- Possibility of starvation for longer processes

- Must estimate length of next CPU burst
  - Need for an algorithm that adapts the estimate dynamically
  - Algorithm must deal with variable length of CPU bursts
    - Exponential averaging

- What if estimates are not correct?
  - Preemption by the operating system

Priority Scheduling

Scheduler can choose a process of higher priority over one of lower priority

- Can be preemptive or non-preemptive
- Priorities can be static or dynamic
- E.g., in SJF priority is the predicted next CPU burst time

Problem of priority based schemes: Danger for starvation

- Low priority processes may never execute
- E.g., process with a long CPU burst in SJF may never execute

A solution: Aging

- As time advances the priority of a process is increased
- Longest running processes eventually reaches highest priority

Priority Scheduling – Example

Process	Priority	Burst time
P_1	3	10
P_2	1	1
P_3	4	2
P_4	5	1
P_5	2	5

P_2	P_5	P_1	P_3	P_4

0 1 6 16 18 19

- Average waiting time = 8.2

Priority Scheduling

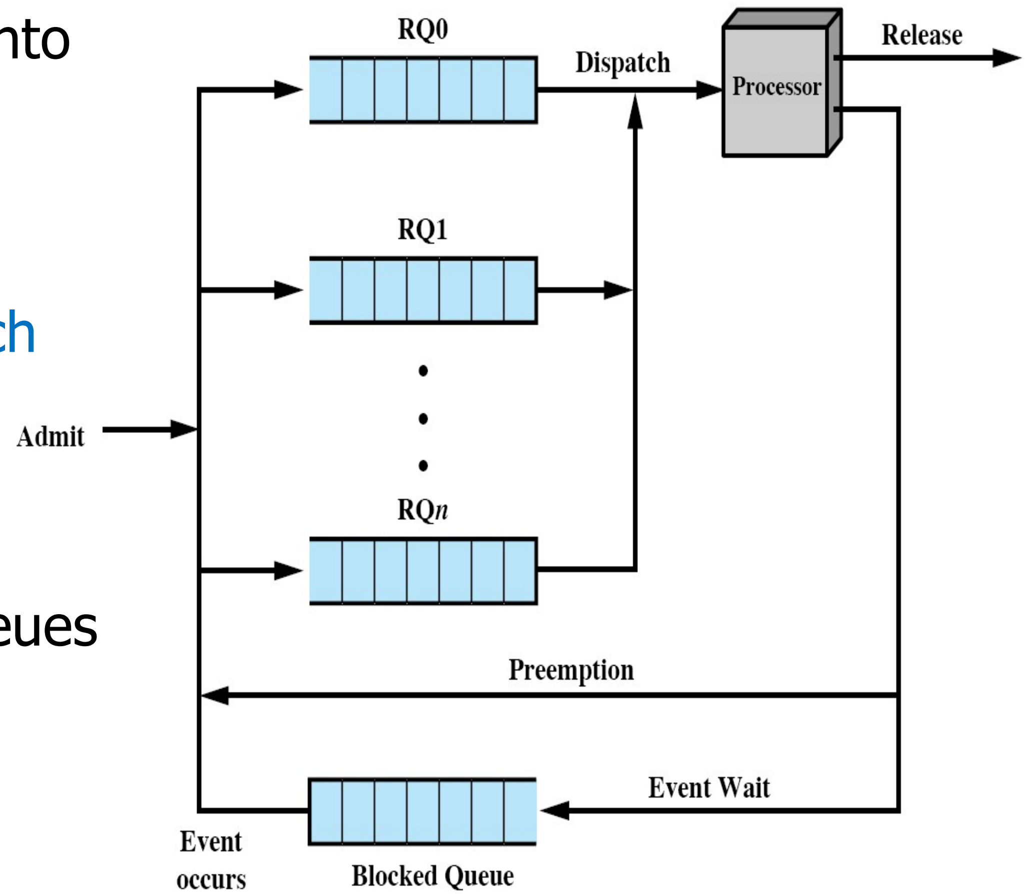
- If processes have equal priority
  - Round robin
  - FCFS

- I/O Bound vs. CPU Bound
  - Which type of processes should be given Higher Priority
  - In order to keep I/O busy increase priority for jobs that often block on I/O

Multilevel Queue

- Idea: Reduce dispatch latency by multilevel queues

- Ready queue is partitioned into separate queues, e.g.,
  - Foreground (interactive)
  - Background (batch)
- Scheduling algorithm for each queue, e.g.,
  - Foreground → RR
  - Background → FCFS
- And scheduling between queues …



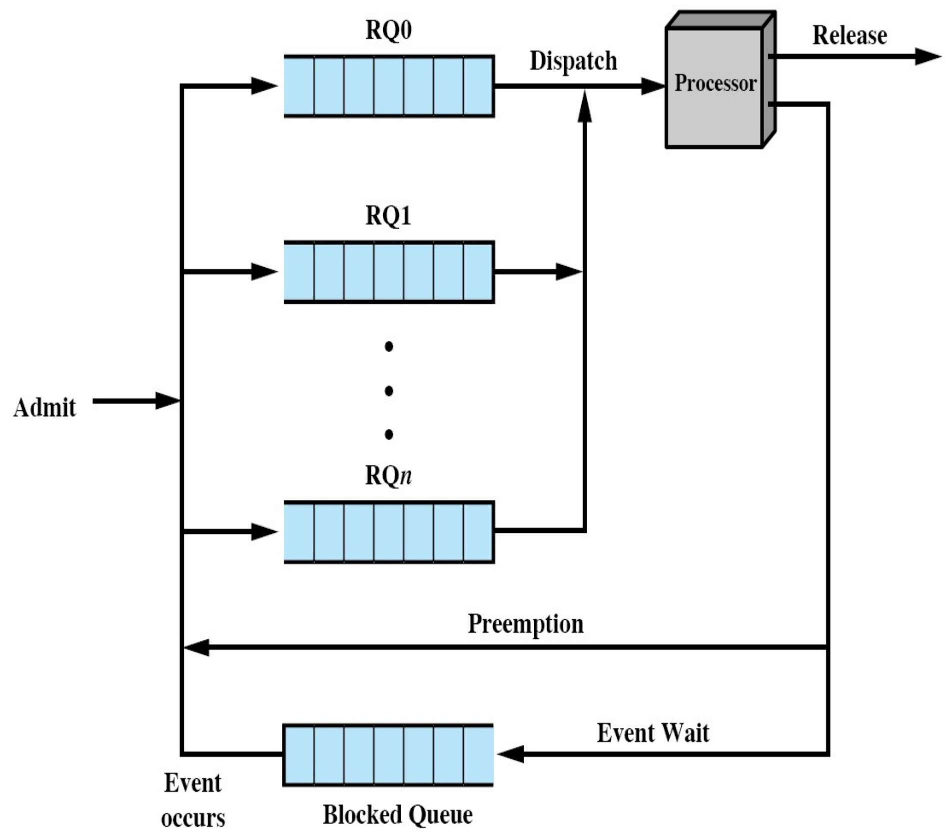
Multilevel Queues: Scheduling Between the Queues

Fixed order/priority e.g.,

- Serve all from foreground then from background
- Possible starvation

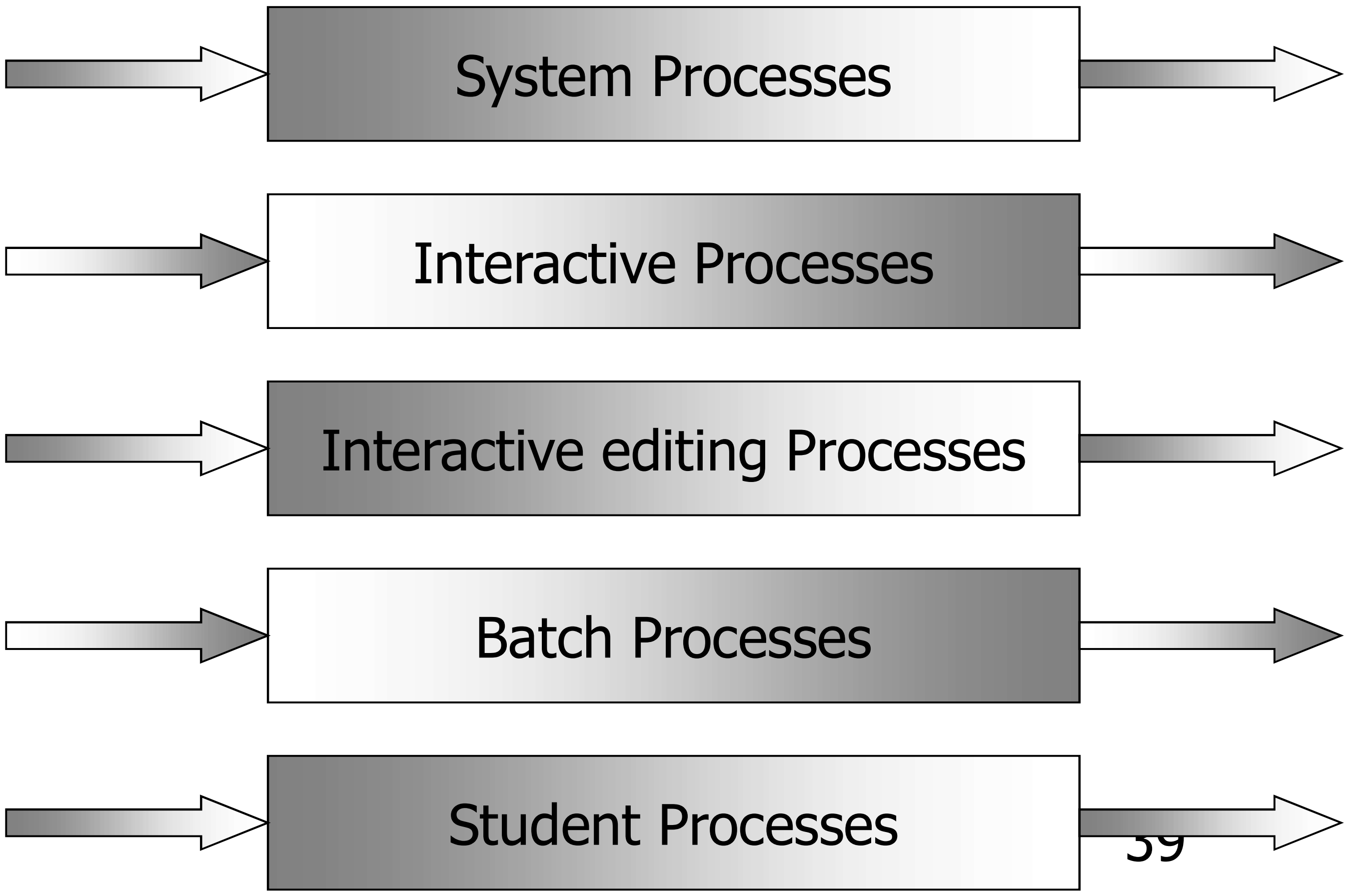
Time slice

- Each queue gets a fraction of CPU time to divide amongst its processes, e.g.,
  - 80% to foreground in RR
  - 20% to background in FCFS



Multilevel Queue Scheduling – Example

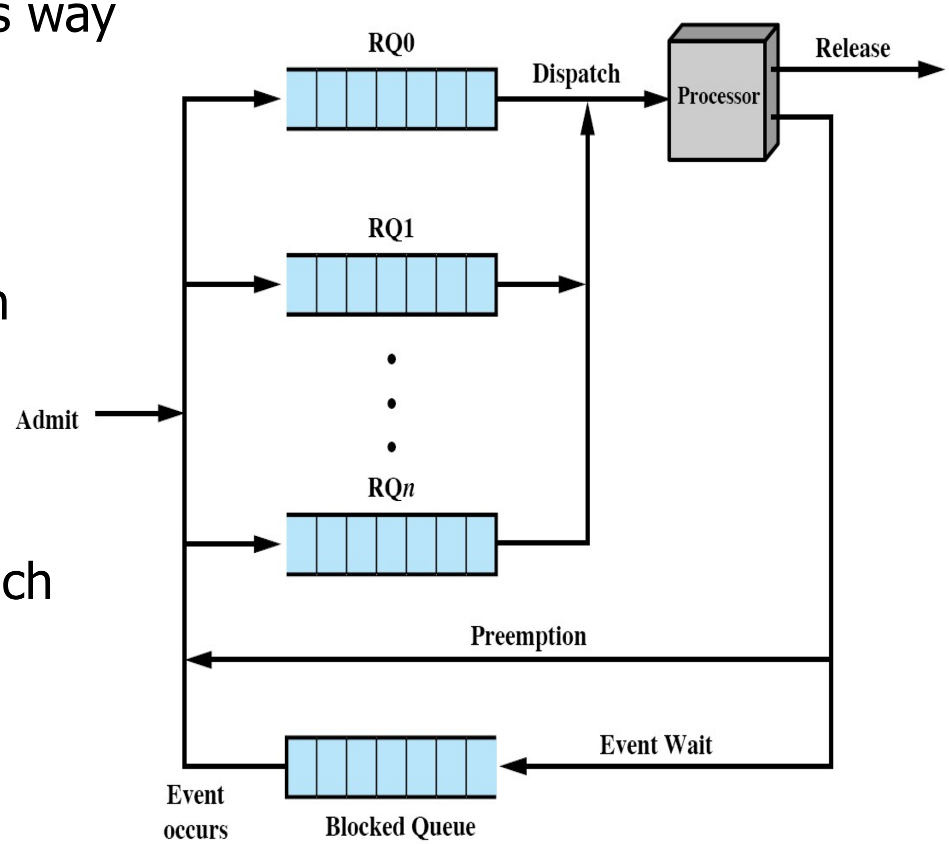
Highest Priority



Lowest Priority

Multilevel Feedback Queue

- A process can move between the various queues
  - Aging can be implemented this way
- Scheduling parameters
  - Number of queues
  - Scheduling algorithms for each queue
  - Method to upgrade a process
  - Method to demote a process
  - Method used to determine which queue a process enters first



Approximation of SJF: Multilevel Feedback Queue

Admission of P_i: $level(P_i) = 0$

- Assign process to RQ_0

Dispatch()

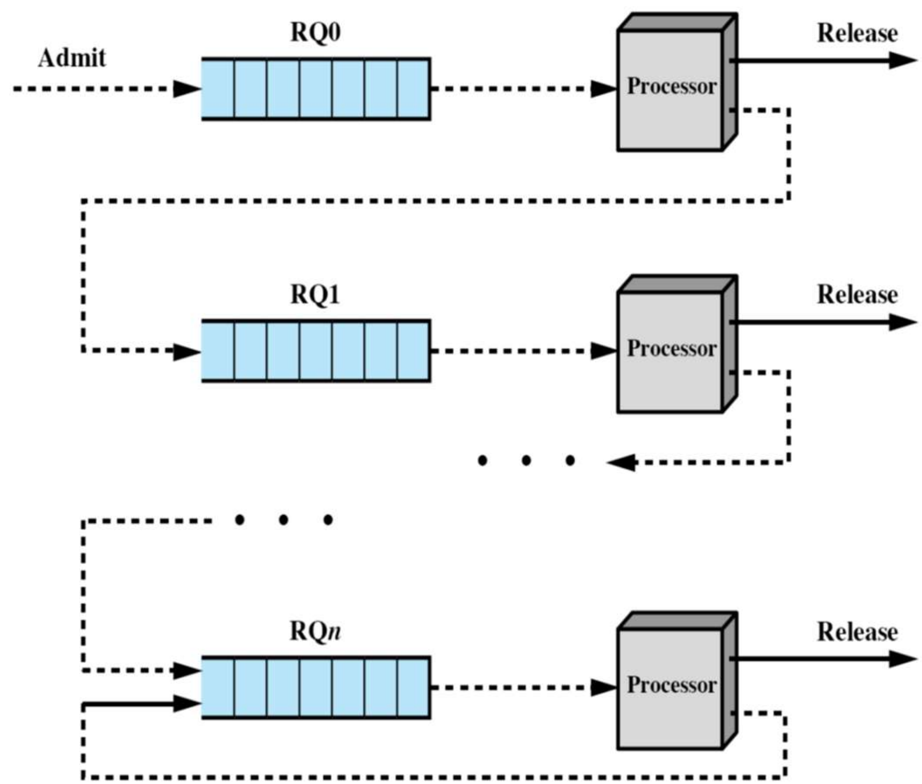
- Find smallest level s.t. $RQ_{level} \neq \emptyset$
- $P_j = RQlevel$.dequeue()
- Set time slice to fit level of process

Upon state change (P_i)

- Running $\rightarrow$ Ready
- $level(Pi) = \max\{level(Pi) + 1, n\}$
- RQ_{level}.enqueue(P_i)
- Dispatch()

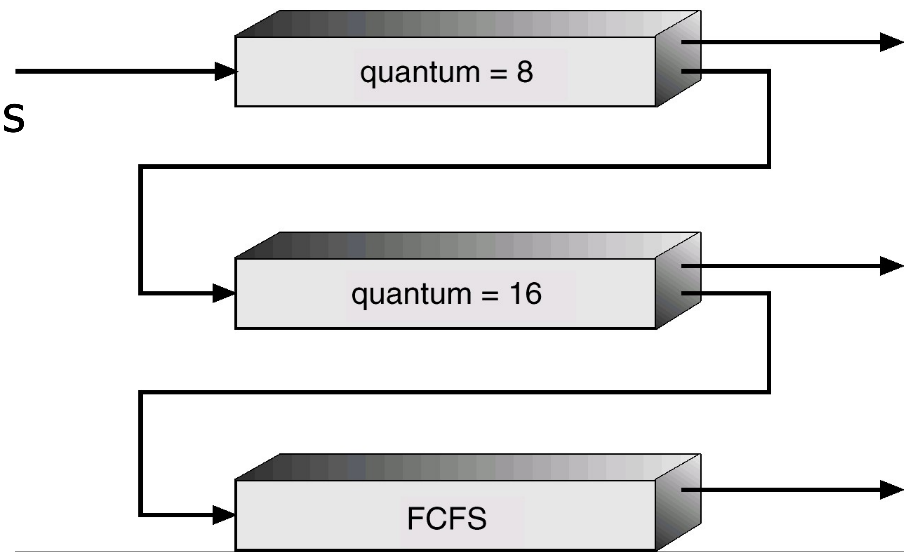
Upon state change (P_i)

- Running $\rightarrow$ blocked
- Enqueue P_i to Event Queue
- Dispatch()



Multilevel Feedback Queues – Example

- Three queues
  - Q_0 – RR with time quantum 8 ms
  - Q_1 – RR time quantum 16 ms
  - Q_2 – FCFS



Starvation is still possible here!

- Scheduling
  - A new process enters queue Q_0
  - When it gains CPU, process receives 8 ms
  - If it does not finish in 8 ms, process is moved to queue Q_1
  - At Q_1 process 16 additional ms
  - If it still does not complete, it is preempted and moved to queue Q_2

Any Question So Far?