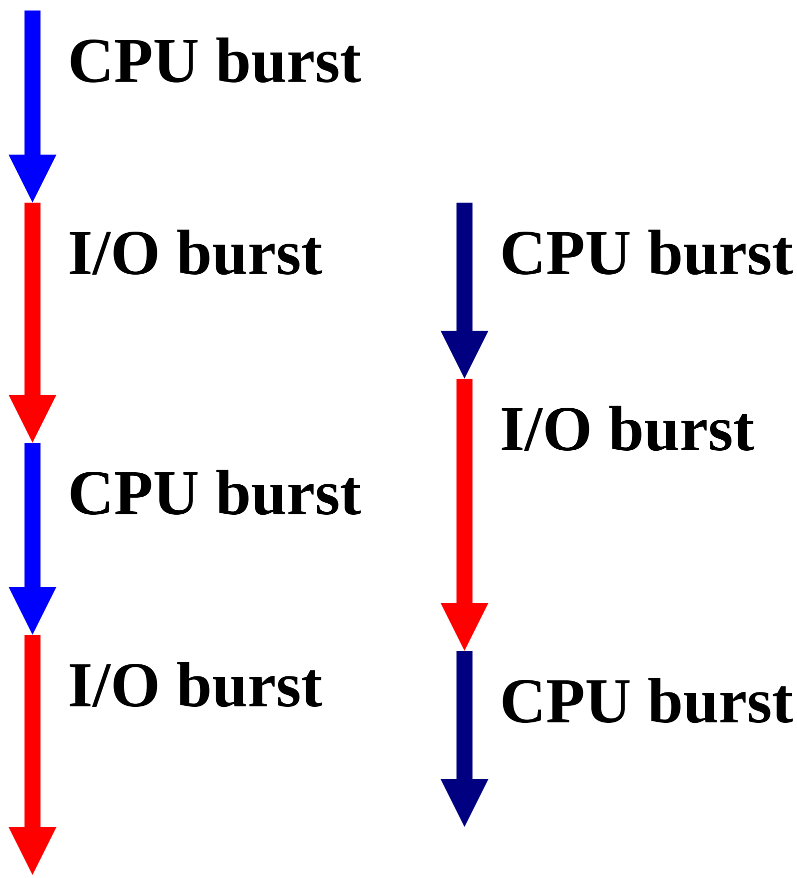
Part II
Process Management
Chapter 5 CPU Scheduling

CPU-I/O Burst Cycle



- **Process execution repeats the CPU burst and I/O burst cycle.**
- **When a process begins an I/O burst, another process can use the CPU for a CPU burst.**

CPU-bound and I/O-bound

- **A process is *CPU-bound* if it generates I/O requests infrequently, using more of its time doing computation.**
- **A process is *I/O-bound* if it spends more of its time to do I/O than it spends doing computation.**
- **A CPU-bound process might have a few very long CPU bursts.**
- **An I/O-bound process typically has many short CPU bursts.**

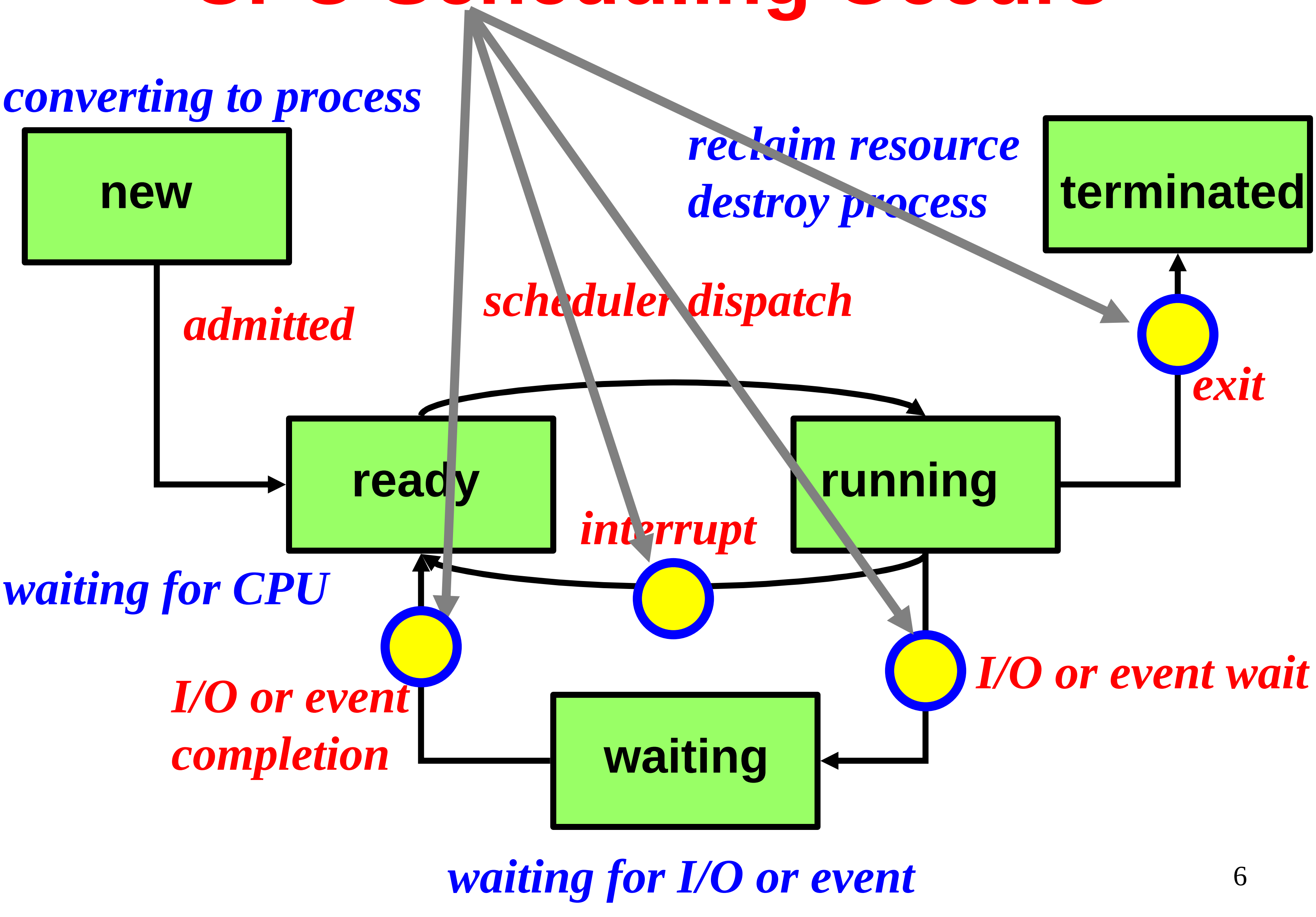
What does a CPU scheduler do?

- **When the CPU is idle, the OS must select another process to run.**
- **This selection process is carried out by the *short-term scheduler* (or *CPU scheduler*).**
- **The CPU scheduler selects a process from the ready queue, and allocates the CPU to it.**
- **The ready queue does not have to be a FIFO one. There are many ways to organize the ready queue.**

Circumstances that scheduling may take place

1. **A process switches from the running state to the wait state (*e.g.*, doing for I/O)**
2. **A process switches from the running state to the ready state (*e.g.*, an interrupt occurs)**
3. **A process switches from the wait state to the ready state (*e.g.*, I/O completion)**
4. **A process terminates**

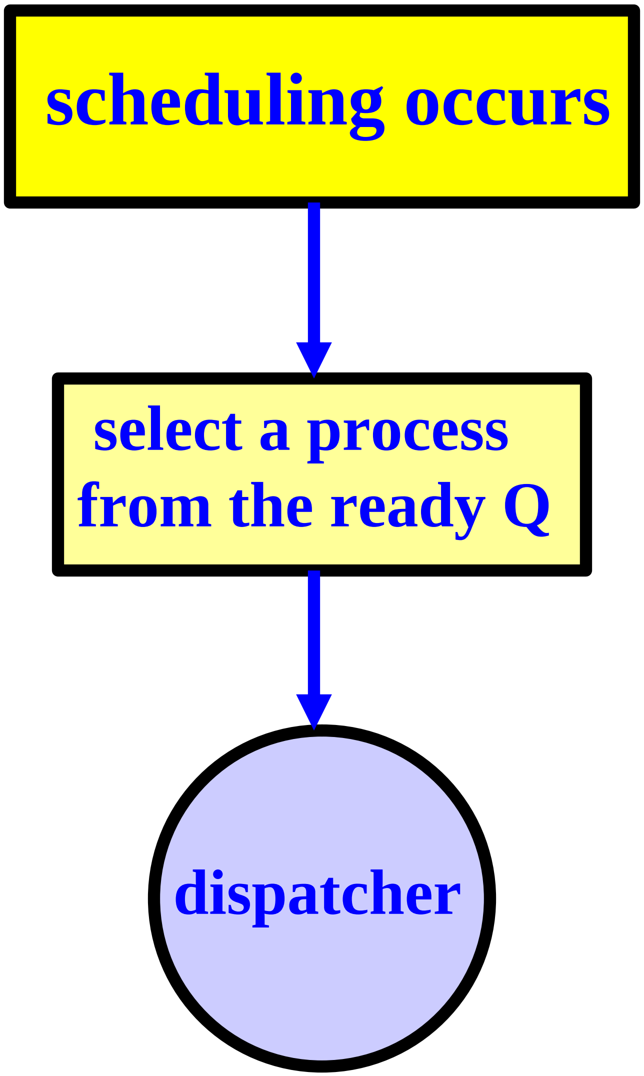
CPU Scheduling Occurs



Preemptive vs. Non-preemptive

- ***Non-preemptive scheduling*: scheduling occurs when a process voluntarily enters the wait state (case 1) or terminates (case 4).**
  - **Simple, but very inefficient**
- ***Preemptive scheduling*: scheduling occurs in all possible cases.**
  - **What if the kernel is in its critical section modifying some important data? Mutual exclusion may be violated.**
  - **The kernel must pay special attention to this situation and, hence, is more complex.**

Scheduling Flow and Dispatcher



- **The dispatcher is the last step in scheduling. It**
  - **Switches context**
  - **Switches to user mode**
  - **Braches to the stored program counter to resume the program's execution.**
- **It has to be very fast as it is used in every context switch.**
- ***Dispatcher latency*: the time to switch two processes.**

Scheduling Criteria: 1/6

- **There are many criteria for comparing different scheduling algorithms. Here are five common ones:**
  - **CPU Utilization**
  - **Throughput**
  - **Turnaround Time**
  - **Waiting Time**
  - **Response Time**

Criterion 1: CPU Utilization 2/6

- We want to keep the CPU as busy as possible.
- CPU utilization ranges from 0 to 100 percent.
- Normally 40% is lightly loaded and 90% or higher is heavily loaded.
- You can bring up a CPU usage meter to see CPU utilization on your system. Or, you can use the `top` command.

Criterion 2: Throughput 3/6

- **The number of processes completed per time unit is called *throughput*.**
- **Higher throughput means more jobs get done.**
- **However, for long processes, this rate may be one job per hour, and, for short (student) jobs, this rate may be 10 per minute.**

Criterion 3: Turnaround Time 4/6

- **The time period between job submission to completion is the *turnaround time*.**
- **From a user's point of view, turnaround time is more important than CPU utilization and throughput.**
- **Turnaround time is the sum of**
  - **waiting time before entering the system**
  - **waiting time in the ready queue**
  - **waiting time in all other events (*e.g.*, I/O)**
  - **time the process actually running on the CPU**

Criterion 4: Waiting Time 5/6

- ***Waiting time*** **is the sum of the periods that a process spends waiting in the ready queue.**
- **Why only ready queue?**
  - **CPU scheduling algorithms do not affect the amount of time during which a process is waiting for I/O and other events.**
  - **However, CPU scheduling algorithms do affect the time that a process stays in the ready queue.**

Criterion 5: Response Time 6/6

- The time from the submission of a request (in an interactive system) to the first response is called *response time*. It does not include the time that it takes to output the response.
- For example, in front of your workstation, you perhaps care more about the time between hitting the Return key and getting your first output than the time from hitting the Return key to the completion of your program (*e.g.*, turnaround time).

What are the goals?

- **In general, the main goal is to maximize CPU utilization and throughput and minimize turnaround time, waiting time and response time.**
- **In some systems (*e.g*., batch systems), maximizing CPU utilization and throughput is more important, while in other systems (*e.g*., interactive) minimizing response time is paramount.**
- **Sometimes we want to make sure some jobs must have guaranteed completion before certain time.**
- **Other systems may want to minimize the variance of the response time.**

Scheduling Algorithms

- **We will discuss a number of scheduling algorithms:**
  - **First-Come, First-Served (FCFS)**
  - **Shortest-Job-First (SJF)**
  - **Priority**
  - **Round-Robin**
  - **Multilevel Queue**
  - **Multilevel Feedback Queue**

First-Come, First-Served: 1/3

- **The process that requests the CPU first is allocated the CPU first.**
- **This can easily be implemented using a queue.**
- **FCFS is not preemptive. Once a process has the CPU, it will occupy the CPU until the process completes or voluntarily enters the wait state.**

FCFS: Example 2/3

A	*B*	*C*	*D*
10	5	7	6

- **Four jobs A, B, C and D come into the system in this order at about the same time.**

Process	Start	Running	End
A	0	10	10
B	10	5	15
C	15	7	22
D	22	6	28

Average Waiting Time
= (0 + 10 + 15 + 22)/4
= 47/4 = 11.8

Average turnaround
= (10 + 15 + 22 + 28)/4
= 75/4 = 18.8

FCFS: Problems 3/3

- **It is easy to have the *convoy effect*: all the processes wait for the one big process to get off the CPU. CPU utilization may be low. Consider a CPU-bound process running with many I/O-bound process.**
- **It is in favor of long processes and may not be fair to those short ones. What if your 1-minute job is behind a 10-hour job?**
- **It is troublesome for time-sharing systems, where each user needs to get a share of the CPU at regular intervals.**

Shortest-Job First: 1/8

- **Each process in the ready queue is associated with the length of its *next* CPU burst.**
- **When a process must be selected from the ready queue, the process with the smallest next CPU burst is selected.**
- **Thus, the processes in the ready queue are sorted in CPU burst length.**
- **SJF can be non-preemptive or preemptive.**

Non-preemptive SJF: Example 2/8

A	*B*	*C*	*D*
10	5	7	6

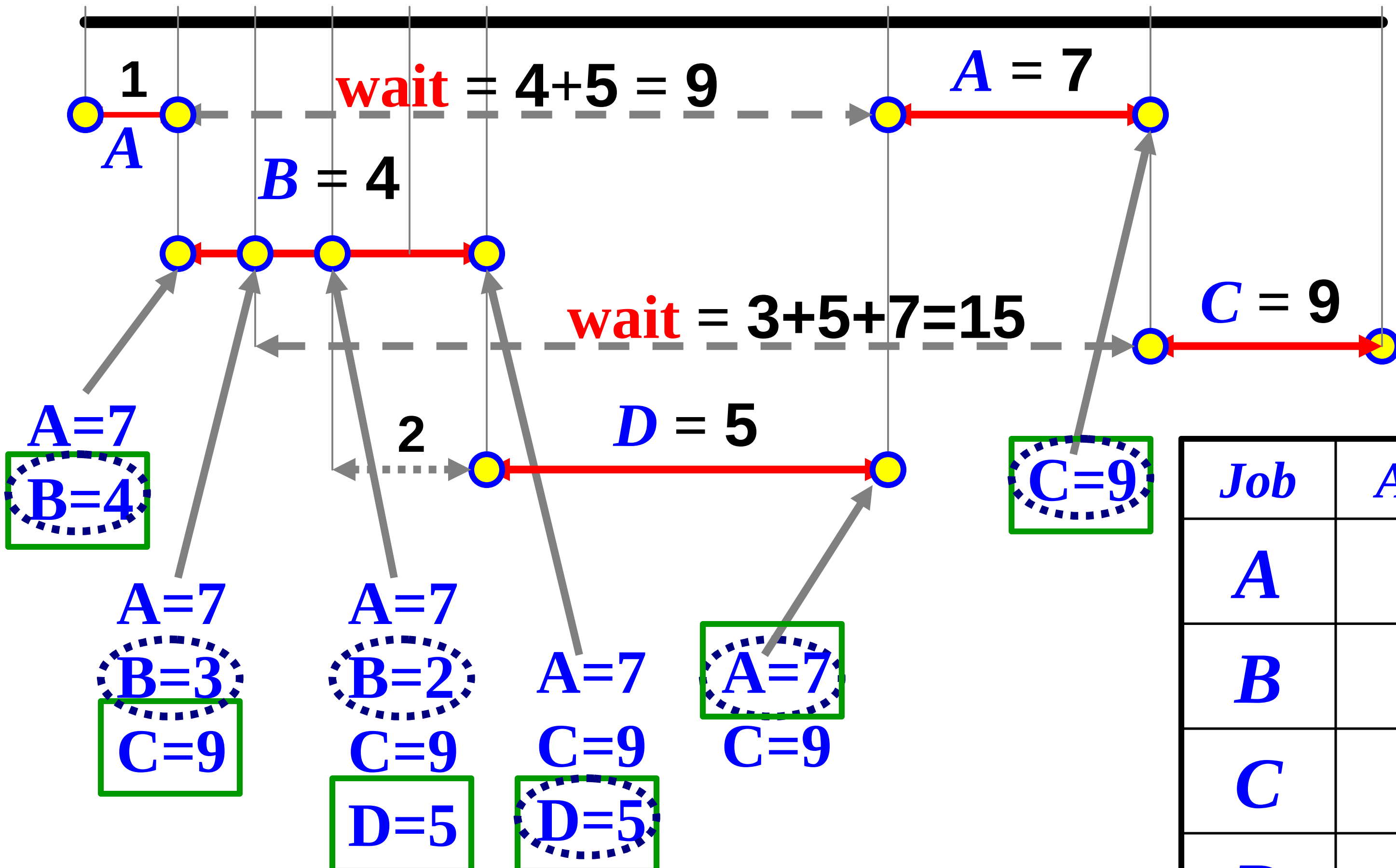
- **Four jobs A, B, C and D come into the system in this order at about the same time.**

Process	Start	Running	End
B	0	5	5
D	5	6	11
C	11	7	18
A	18	10	28

Average waiting time
= (0 + 5 + 11 + 18)/4
= 34/4 = 8.5

Average turnaround
= (5 + 11 + 18 + 28)/4
= 62/4 = 15.5

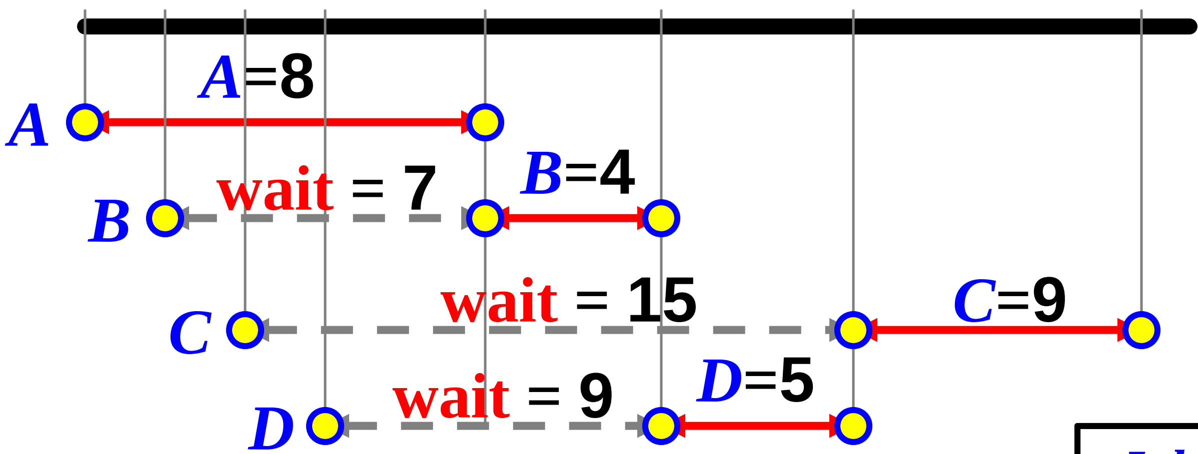
Preemptive SJF: Example 3/8



Avg wait = (9+0+15+2)/4 =26/4=6.5

Job	*Arr*	*Burst*	*Wait*
A	0	8	9
B	1	4	0
C	2	9	15
D	3	5	2

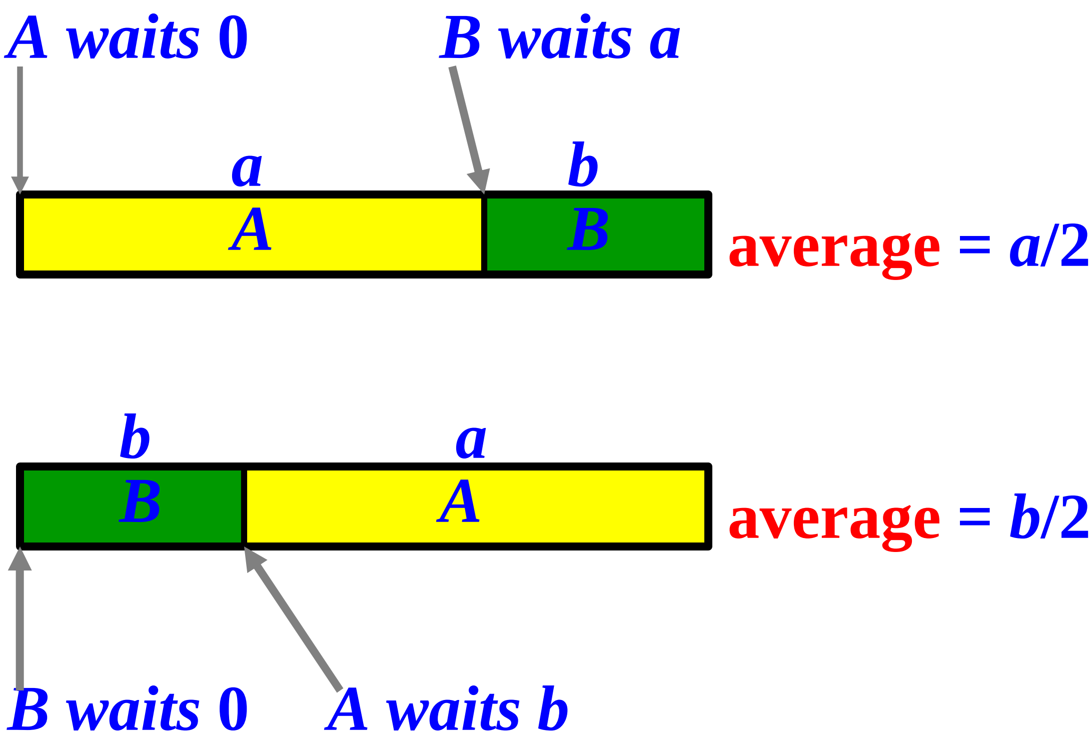
Non-preemptive SJF: 4/8



Job	*Arr*	*Burst*	*Wait*
A	**0**	**8**	**0**
B	**1**	**4**	**7**
C	**2**	**9**	**15**
D	**3**	**5**	**9**

average wait = (0+7+15+9)/4=7.75

SJF is provably optimal! 5/8



- Every time we make a short job before a long job, we reduce average waiting time.
- We may switch out of order jobs until all jobs are in order.
- If the jobs are sorted, job switching is impossible.

How do we know the next CPU burst? 6/8

- Without a good answer to this question, SJF cannot be used for CPU scheduling.
- We try to **predict** the next CPU burst!
- Let t_n be the length of the nth CPU burst and p_{n+1} be the prediction of the next CPU burst

$$p_{n+1} = a\ t_n + (1 - a)\ p_n$$

  where a is a weight value in [0,1].
- If a = 0, then $p_{n+1} = p_n$ and recent history has no effect. If a = 1, only the last burst matters. If a is ½, the actual burst and predict values are equally important.

Estimating the next burst: Example: 7/8

- **Initially, we have to guess the value of p_1 because we have no history value.**
- **The following is an example with a = ½.**

CPU burst		6 t_1	4 t_2	6 t_3	4 t_4	13 t_5	13 t_6	13 t_7
Guess	10 p_1	8 p_2	6 p_3	6 p_4	5 p_5	9 p_6	11 p_7	12 p_8

SJF Problems: 8/8

- **It is difficult to estimate the next burst time value accurately.**
- **SJF is in favor of short jobs. As a result, some long jobs may not have a chance to run at all. This is *starvation*.**
- **The preemptive version is usually referred to as *shortest-remaining-time-first* scheduling, because scheduling is based on the "remaining time" of a process.**

Priority Scheduling 1/4

- **Each process has a *priority*.**
- **Priority may be determined internally or externally:**
  - **internal priority: determined by time limits, memory requirement, # of files, and so on.**
  - **external priority: not controlled by the OS (*e.g.*, importance of the process)**
- **The scheduler always picks the process (in ready queue) with the highest priority to run.**
- **FCFS and SJF are special cases of priority scheduling. (Why?)**

Priority Scheduling: Example 2/4

A_2	B_4	C_1	D_3
10	5	7	6

- Four jobs A, B, C and D come into the system in this order at about the same time. Subscripts are priority. Smaller means higher.

Process	Start	Running	End
C	0	7	7
A	7	10	17
D	17	6	23
B	23	5	28

average wait time
= (0+7+17+23)/4
= 47/4 = 11.75

average turnaround time
= (7+17+23+28)/4
= 75/4 = 18.75

Priority Scheduling: Starvation 3/4

- Priority scheduling can be **non-preemptive** or **preemptive**.
- With preemptive priority scheduling, if the newly arrived process has a higher priority than the running one, the latter is preempted.
- **Indefinite block** (or **starvation**) may occur: a low priority process may never have a chance to run.

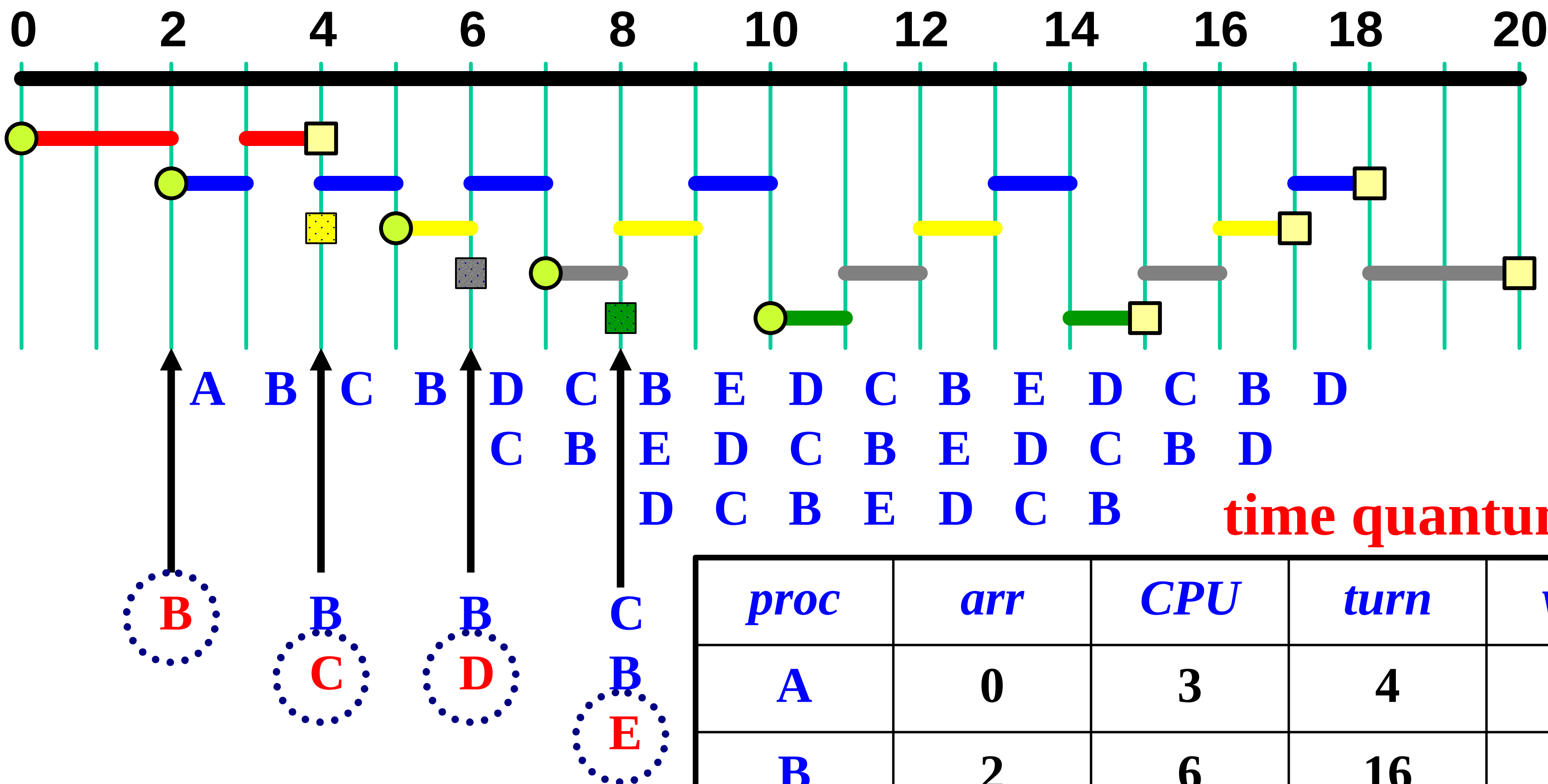
Priority Scheduling: Aging 4/4

- **Aging is a technique to overcome the starvtion problem.**
- ***Aging*: gradually increases the priority of processes that wait in the system for a long time.**
- **Example:**
  - **If 0 is the highest (*resp*., lowest) priority, then we could decrease (*resp*., increase) the priority of a waiting process by 1 every fixed period (*e.g*., every minute).**

Round-Robin (RR) Scheduling: 1/4

- **RR is similar to FCFS, except that each process is assigned a time quantum.**
- **All processes in the ready queue is a FIFO list.**
- **When the CPU is free, the scheduler picks the first and lets it run for one time quantum.**
- **If that process uses CPU for less than one time quantum, it is moved to the tail of the list.**
- **Otherwise, when one time quantum is up, that process is preempted by the scheduler and moved to the tail of the list.**

Round-Robin: Example 1 2/4



time quantum = 1

proc	*arr*	*CPU*	*turn*	*wait*
A	0	3	4	1
B	2	6	16	10
C	4	4	13	9
D	6	5	14	9
E	8	2	7	5

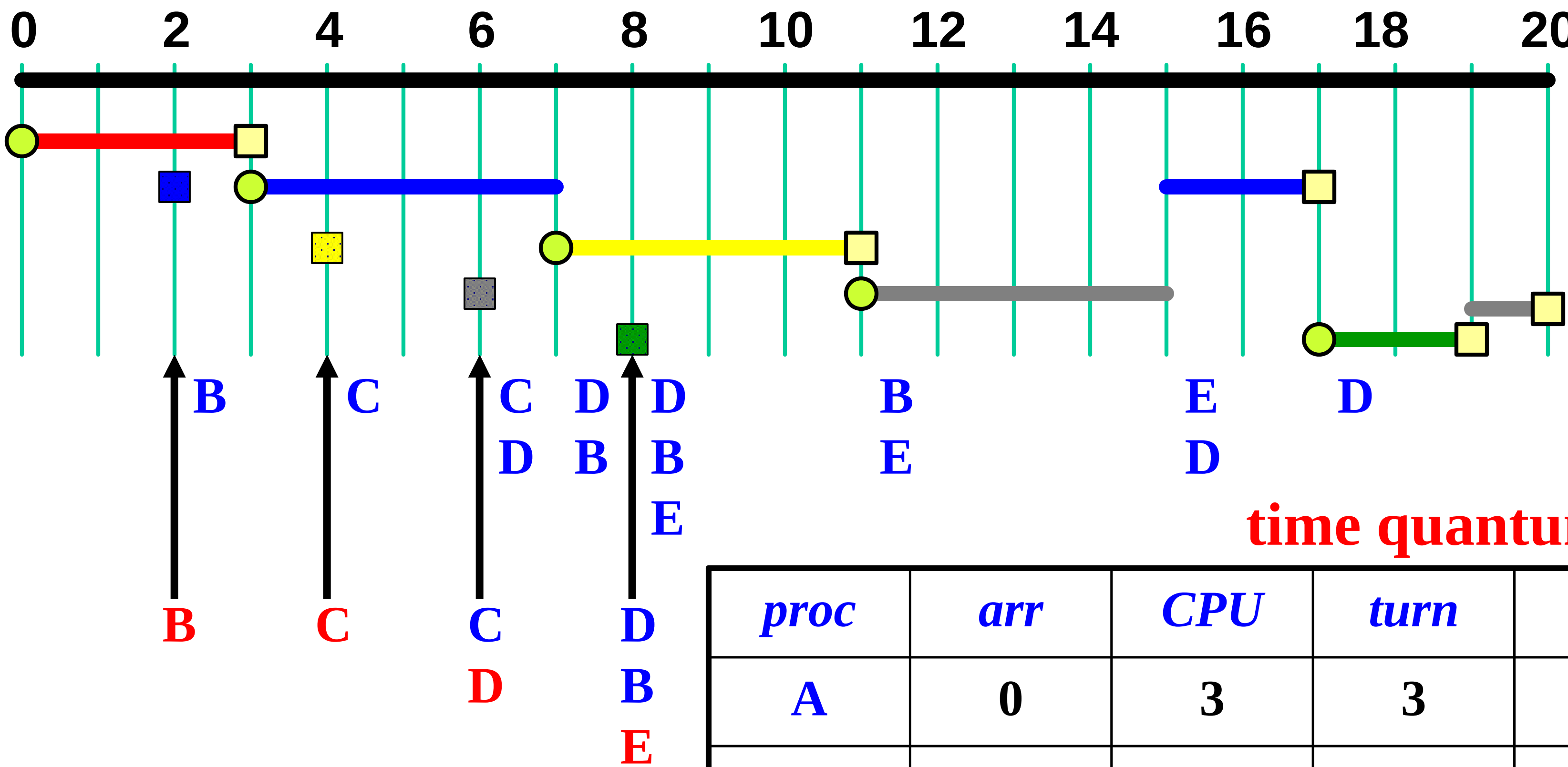
C turnaround = 17-4=13

C wait = 13-4=9

Avg turnaround=10.8

Avg wait = 4.25

Round-Robin: Example 2 3/4



time quantum = 4

proc	*arr*	*CPU*	*turn*	*wait*
A	0	3	3	0
B	2	6	15	9
C	4	4	7	3
D	6	5	14	9
E	8	2	11	9

D turnaround = 20-6=14

D wait = 14-5=9

Avg turnaround=10

Avg wait = 6

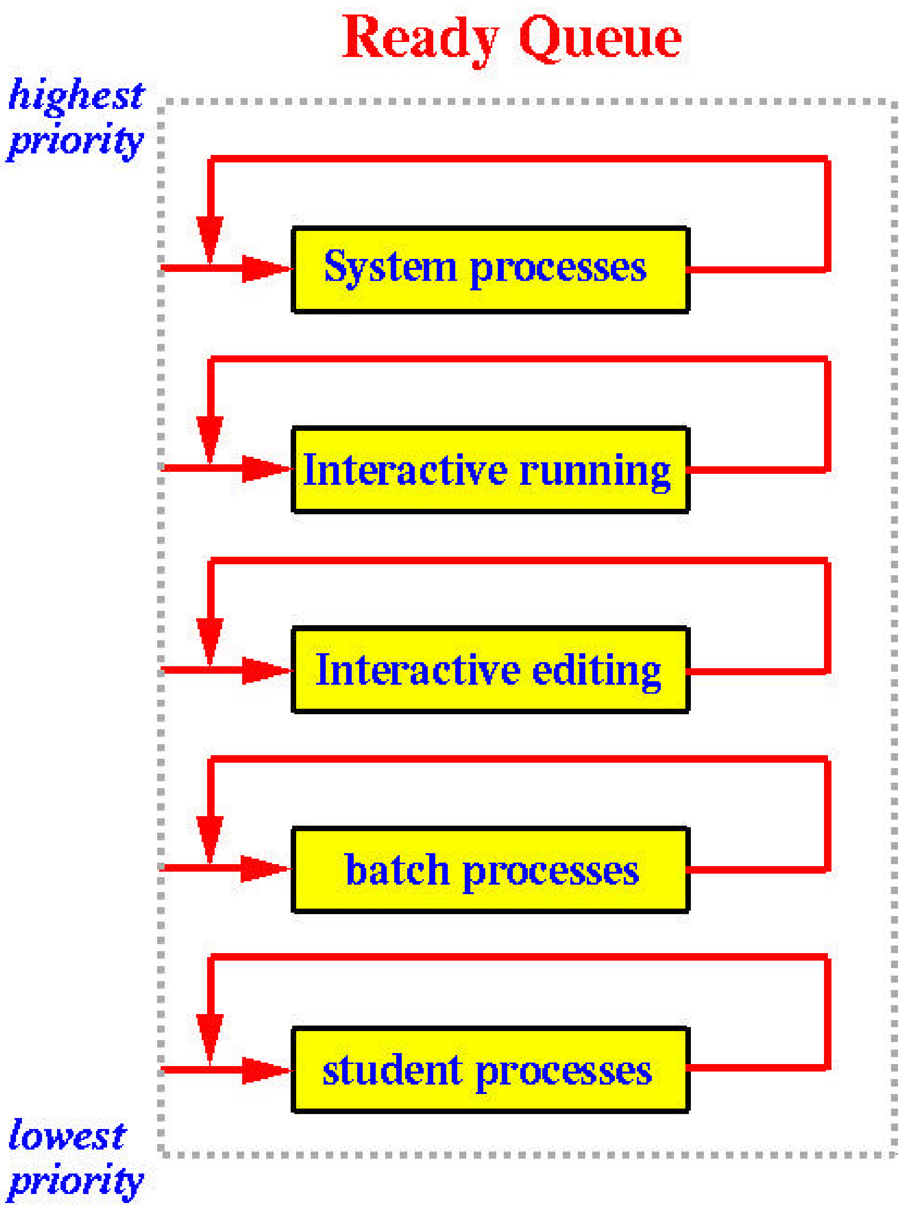
RR Scheduling: Some Issues 4/4

- If time quantum is **too large**, RR reduces to **FCFS**
- If time quantum is **too small**, RR becomes **processor sharing**
- Context switching may affect RR's performance
  - Shorter time quantum means more context switches
- Turnaround time also depends on the size of time quantum.
- In general, 80% of the CPU bursts should be shorter than the time quantum

Multilevel Queue Scheduling

- **A *multilevel queue scheduling* algorithm partitions the ready queue into a number of separate queues (*e.g.*, foreground and background).**
- **Each process is assigned permanently to one queue based on some properties of the process (*e.g.*, memory usage, priority, process type)**
- **Each queue has its own scheduling algorithm (*e.g.*, RR for foreground and FCFS for background)**
- **A priority is assigned to each queue. A higher priority process may preempt a lower priority process.**

Multilevel Queue

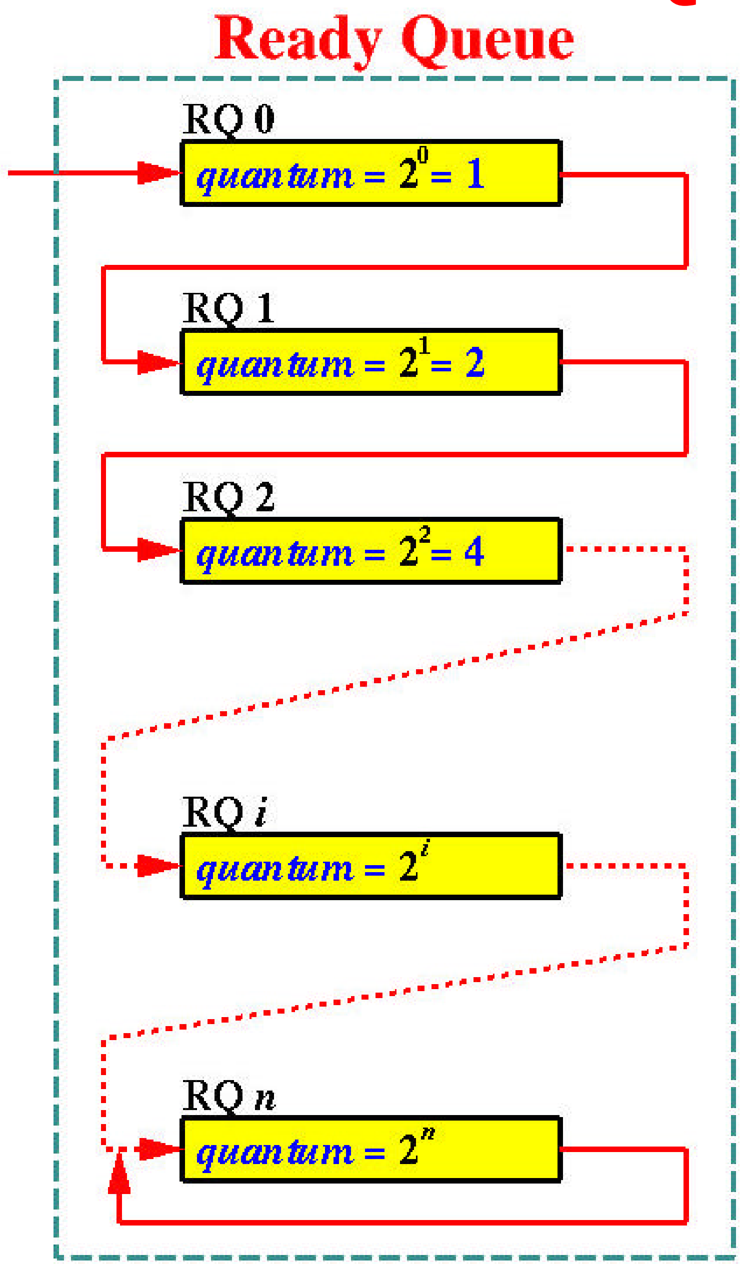


- **A process P can run only if all queues above the queue that contains P are empty.**
- **When a process is running and a process in a higher priority queue comes in, the running process is preempted.**

Multilevel Queue with Feedback

- ***Multilevel queue with feedback scheduling*** **is similar to multilevel queue; however, it allows processes to move between queues.**
- **If a process uses more (*resp*., less) CPU time, it is moved to a queue of lower (*resp*., higher) priority.**
- **As a result, I/O-bound (*resp*., CPU-bound) processes will be in higher (*resp*., lower) priority queues.**

Multilevel Queue with Feedback



- **Processes in queue *i* have time quantum 2^i**
- **When a process' behavior changes, it may be placed (*i.e.*, promoted or demoted) into a difference queue.**
- **Thus, when an I/O-bound (*resp*., CPU-bound) process starts to use more CPU (*resp*., more I/O), it may be demoted (*resp*., promoted) to a lower (*resp*., higher) queue.**

Real-Time Scheduling: 1/2

- **There are two types of real-time systems, hard and soft:**
  - ***Hard Real-Time*: critical tasks must be completed within a guaranteed amount of time**
    - **The scheduler either admits a process and guarantees that the process will complete on-time, or reject the request (*resource reservation*)**
    - **This is almost impossible if the system has secondary storage and virtual memory because these subsystems can cause unavoidable delay.**
    - **Hard real-time systems usually have special software running on special hardware.**

Real-Time Scheduling: 2/2

- **There are two types of real-time systems, hard and soft:**
  - ***Soft Real-Time*: Critical tasks receive higher priority over other processes**
    - **It is easily doable within a general system**
    - **It could cause long delay (starvation) for non-critical tasks.**
    - **The CPU scheduler must prevent aging to occur. Otherwise, critical tasks may have lower priority.**
    - **Aging can be applied to non-critical tasks.**
    - **The dispatch latency must be small.**

How do we reduce dispatch latency?

- **Many systems wait when serving a system call or waiting for the completion of an I/O before a context switch to occur. This could cause a long delay.**
- **One way to overcome this problem is to add *preemption points* in long-duration calls. At these preemption points, the system checks to see if a high-priority process is waiting to run.**
- **If there are, the system is preempted by a high-priority process.**
- **Dispatch latency could still be high because only a few preemption points can be added. A better solution is to make the whole system *preemptible*.**

Priority Inversion

- **What if a high-priority process needs to access the data that is currently being accessed by a low-priority process? The high-priority process is blocked by the low-priority process. This is *priority inversion*.**
- **This can be solved with *priority-inheritance protocol*.**
  - **All processes, including the one that is accessing the data, inherit the high priority until they are done with the resource.**
  - **When they finish, their priority values revert back to the original values.**