Scheduling

*Throughout the course we will use overheads that were adapted from those distributed from the textbook website. Slides are from the book authors, modified and selected by Jean Mayo, Shuai Wang and C-K Shene.

C++ is an insult to the human brain.

Niklaus Wirth

Spring 2019

Main Points

- Scheduling policy: what to do next, when there are multiple threads ready to run
  - Or multiple packets to send, or web requests to serve, or …
- Definitions
  - response time, throughput, predictability
- Uniprocessor policies
  - FIFO, round robin, optimal
  - multilevel feedback as approximation of optimal
- Multiprocessor policies
  - Affinity scheduling, gang scheduling
- Queueing theory
  - Can you predict/improve a system's response time?

Example

- You manage a web site, that suddenly becomes wildly popular. Do you
  - buy more hardware?
  - implement a different scheduling policy?
  - turn away some users? Which ones?
- How much worse will performance get if the web site becomes even more popular?

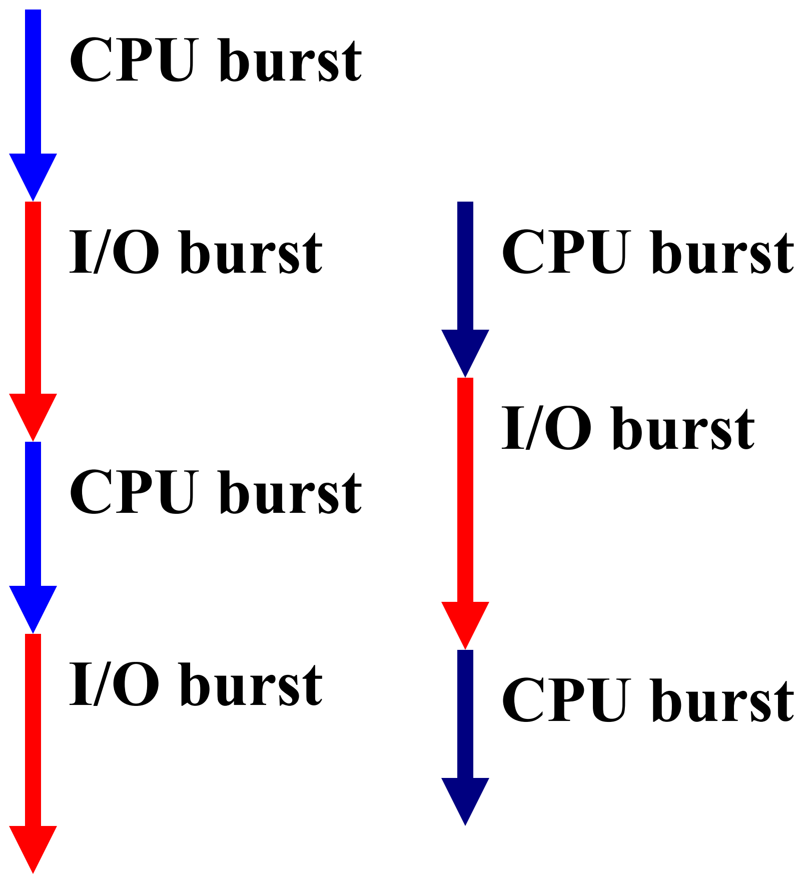
Definitions

- Task/Job
  - User request: mouse click, web request, shell command, …
- Latency/response time
  - How long does a task take to complete?
- Throughput
  - How many tasks can be done per unit of time?
- Overhead
  - How much extra work is done by the scheduler?
- Fairness
  - How equal is the performance received by different users?
- Predictability
  - How consistent is the performance over time?

More Definitions

- Workload
  - Set of tasks for system to perform
- Preemptive scheduler
  - If we can take resources away from a running task
- Work-conserving
  - Resource is used whenever there is a task to run
  - For non-preemptive schedulers, work-conserving is not always better
- Scheduling algorithm
  - takes a workload as input
  - decides which tasks to do first
  - Performance metric (throughput, latency) as output
  - Only preemptive, work-conserving schedulers to be considered

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.

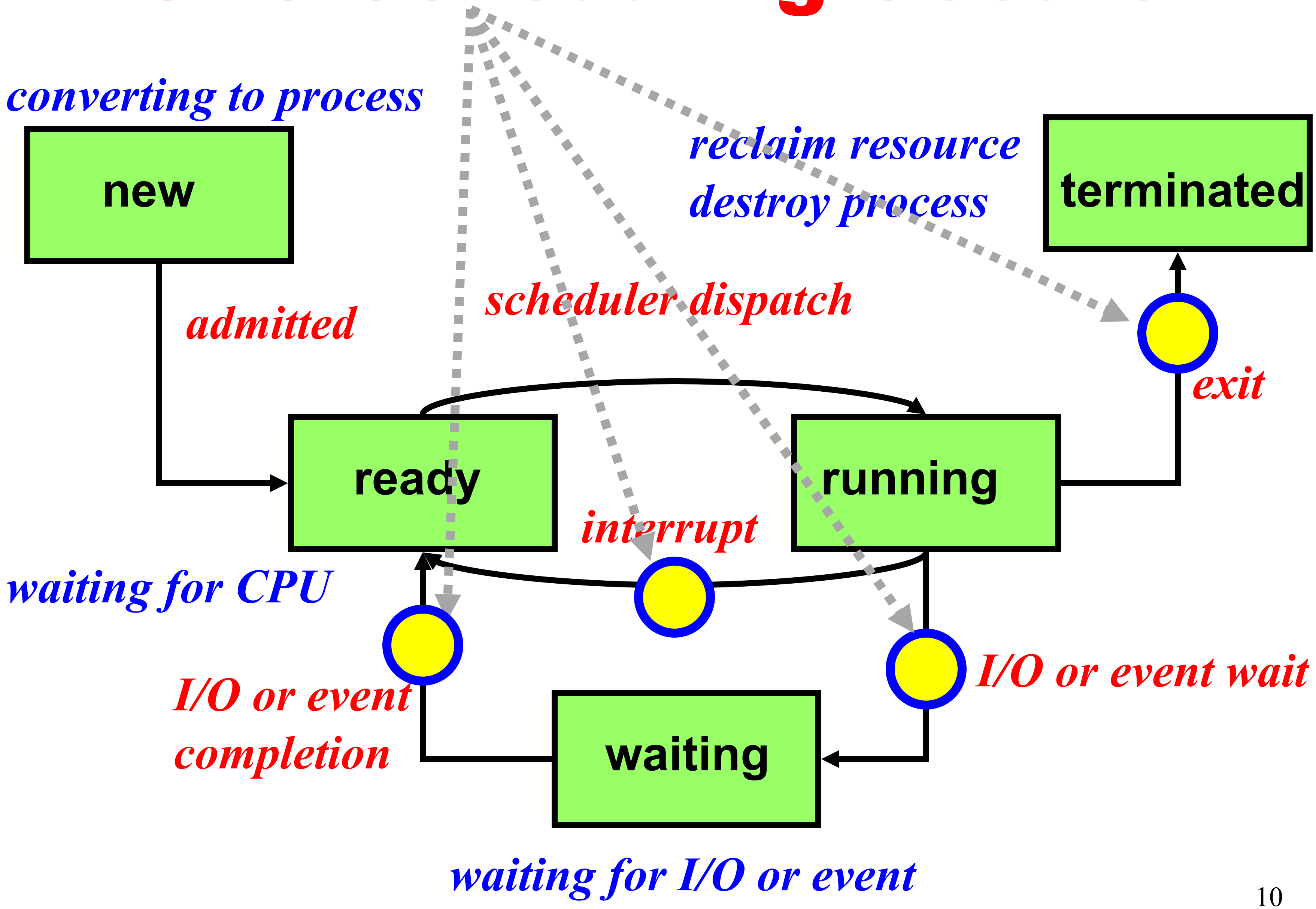
What Does a CPU Scheduler Do?

- 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 FIFO**. 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.*, due to an interrupt)
3. A process switches from the wait state to the ready state (*e.g.*, I/O completion)
4. A process terminates

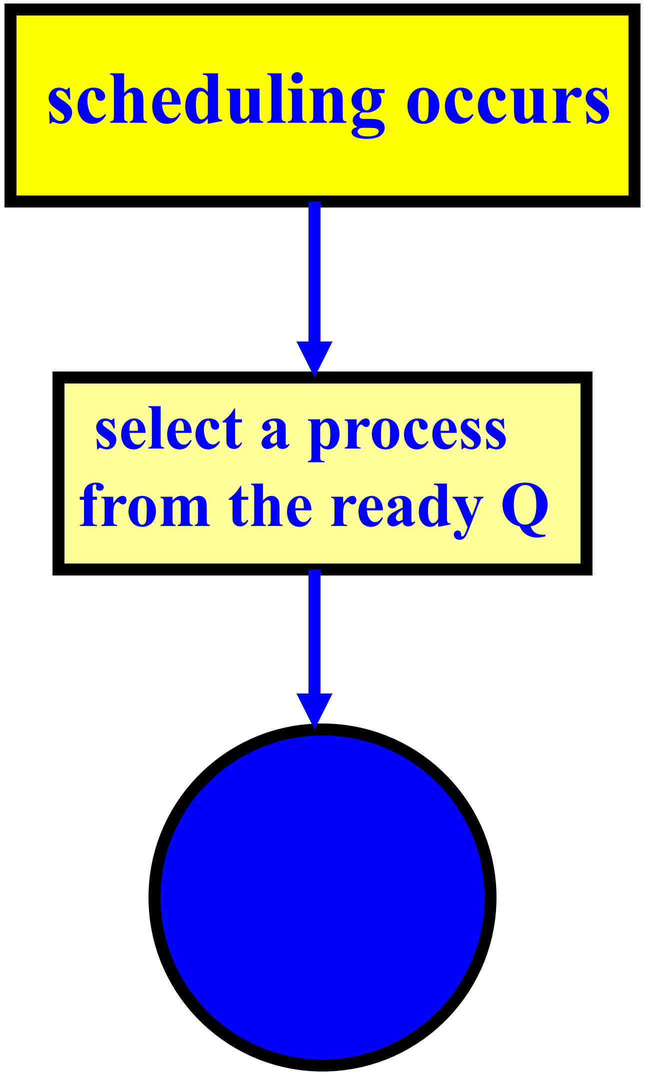
CPU Scheduling Occurs



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
  - Branches 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 may bring up a CPU usage meter to see CPU utilization on your system. Or, you may 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*: many 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: 1/9

- 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, processes in the ready queue are sorted in CPU burst length.
- SJF can be non-preemptive or preemptive.

Non-preemptive SJF: Example 2/9

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

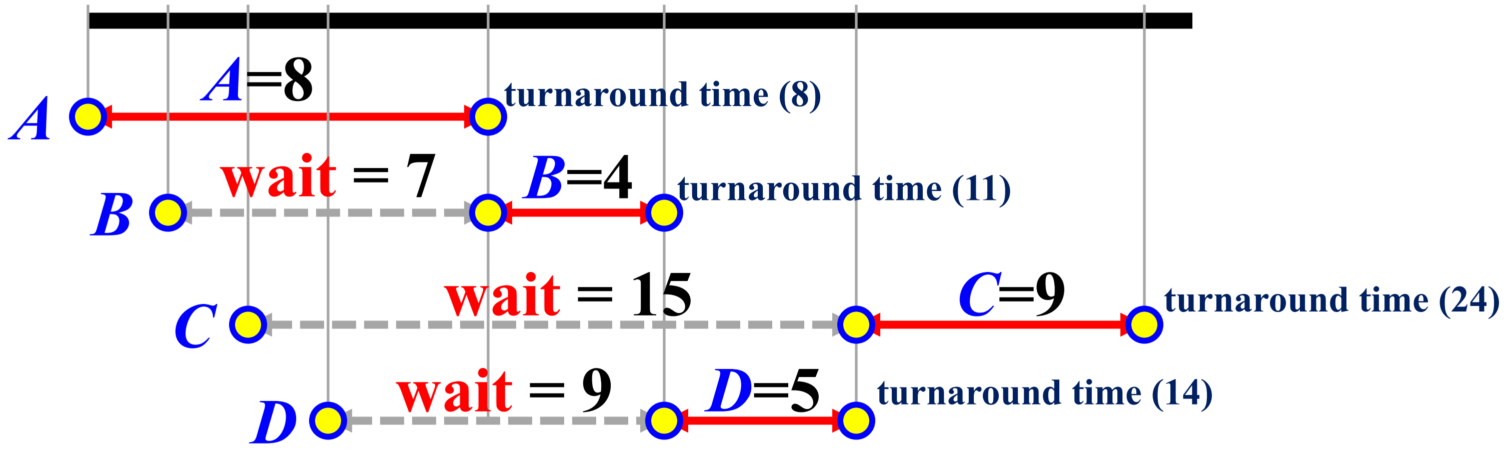
- Four jobs A, B, C and D come into the system in this order at about the same time.

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

Non-Preemptive SJF: 3/9



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

average turnaround time

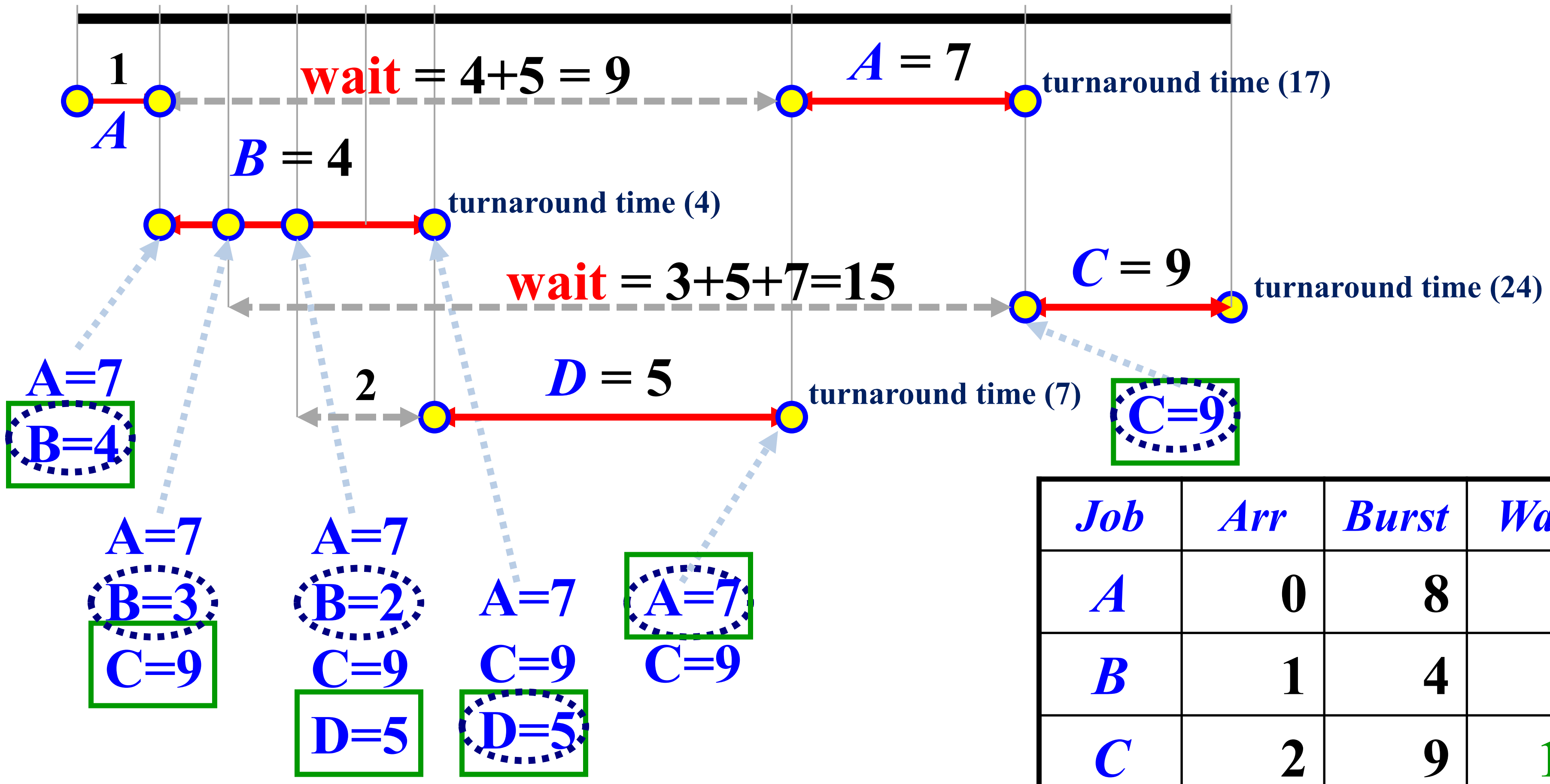
= (8+11+24+14)/4 = 14.25

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

Preemptive SJB: 4/9

- By “preemptive” it means when a new process comes in, the CPU scheduler may suspend the running process and update the remaining CPU burst of **EVERY** process, the newcomer included, and reorganize the ready queue.
- In this way, the current running process may not have the CPU if the newcomer has a shorter CPU burst.
- For example, suppose a newcomer has a CPU burst of 5 units of time. If the currently running process has 6 units of time to run, then the newcomer will have the CPU, because it has a shorter CPU-burst (5) than that of the currently running process (6).

Preemptive SJF: Example 5/9



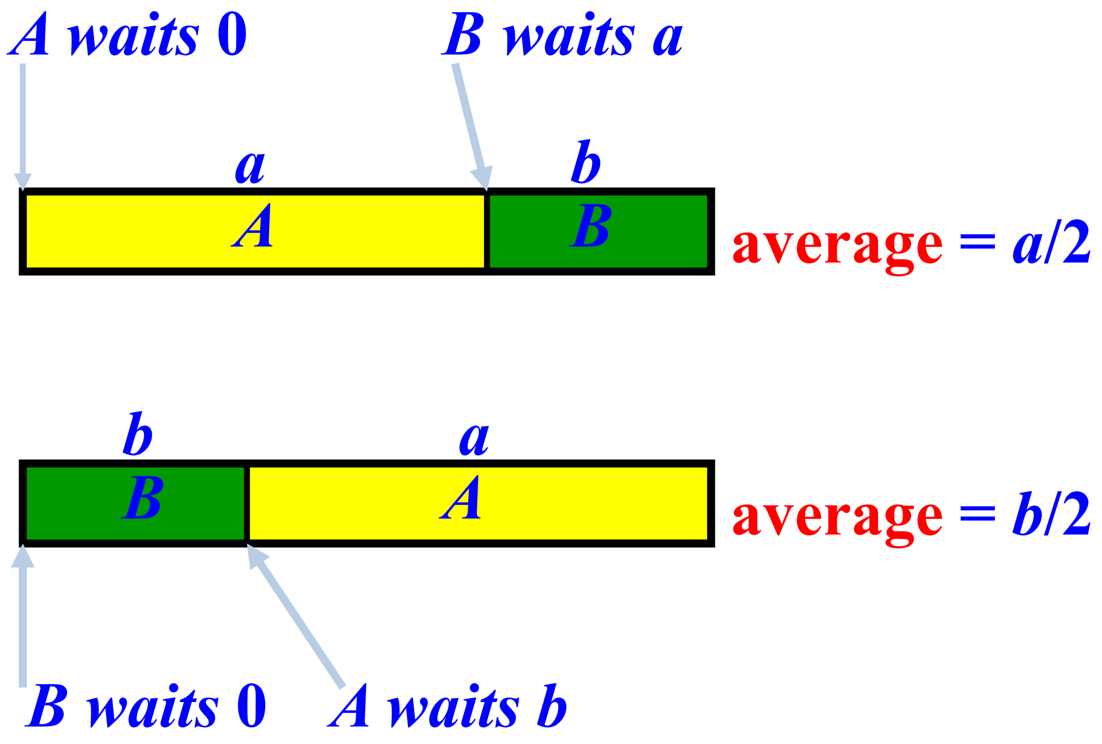
Job	*Arr*	*Burst*	*Wait*
A	0	8	9
B	1	4	0
C	2	9	15
D	3	5	2

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

average turnaround time = (17+4+24+7)/4 = 13

Preemptive means: when a new process arrives, an update of burst time is required

SJF Is Provably Optimal! 6/9



- What is optimal?
- It means **minimal average waiting time.**
- Every time we make a short job before a long one, we reduce average waiting time.
- We may switch out of order jobs until all jobs are in order.
- If jobs are sorted, job switching is impossible.
- Remember the bubble sort?

How Do We Know the Next CPU Burst? 7/9

- 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

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

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

Estimating the next burst: Example: 8/9

- **Initially, we have to use a default value p_1 because we have no history value.**
- **The following is an example with $\alpha = ½$.**

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: 9/9

- 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* (SRTF) scheduling, because scheduling is based on the "remaining time" of a process.

Priority Scheduling 1/4

- Each process is assigned 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 CPU scheduler always picks the process (in the 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 starvation 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).**

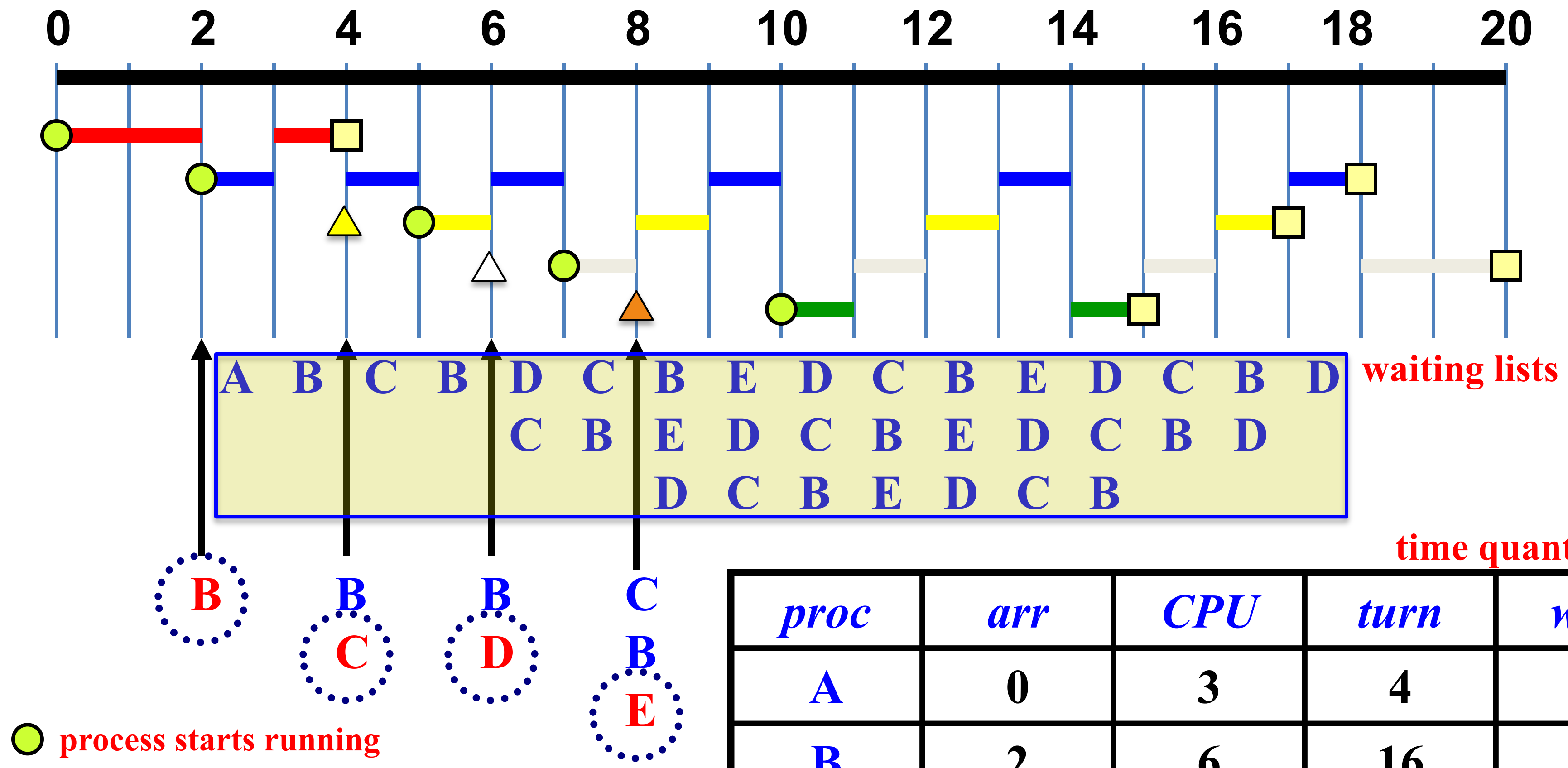
Round-Robin Scheduling: 1/5

- Round-Robin (RR) is similar to FCFS, except that each process is assigned a time quantum.
- Processes in the ready queue form 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 Scheduling: 2/5

- At the end of a time quantum, the currently running process is removed from the CPU and the next process is chosen.
- At this point, what if a new process arrives?
- Some systems add the newcomer to the tail of the list rather than running it immediately.
- Certainly, different systems handle this situation differently.

Round-Robin: Example 1 3/5



● process starts running

□ process ends

△ process arrives

◌ process arrives

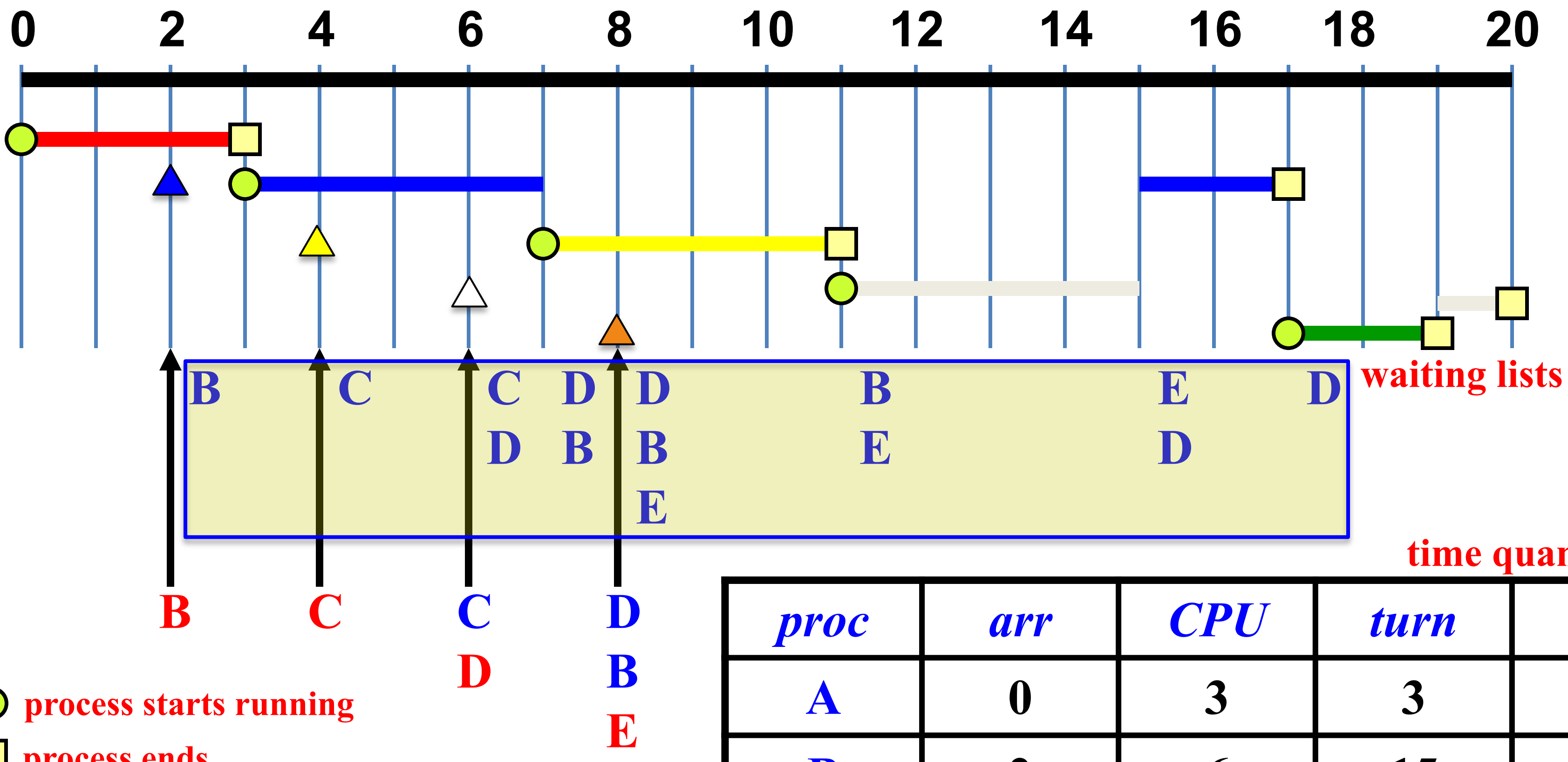
time quantum = 1

C turnaround = 17-4=13
C wait = 13-4=9
Avg turnaround=10.8
Avg wait = 4.25

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

The above diagram is constructed based on giving higher priority to the currently running process.

Round-Robin: Example 2 4/5



time quantum = 4

● process starts running

□ process ends

△ process arrives

D turnaround = 20-6=14

D wait = 14-5=9

Avg turnaround=10

Avg wait = 6

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

The above diagram is constructed based on giving higher priority to the currently running process.

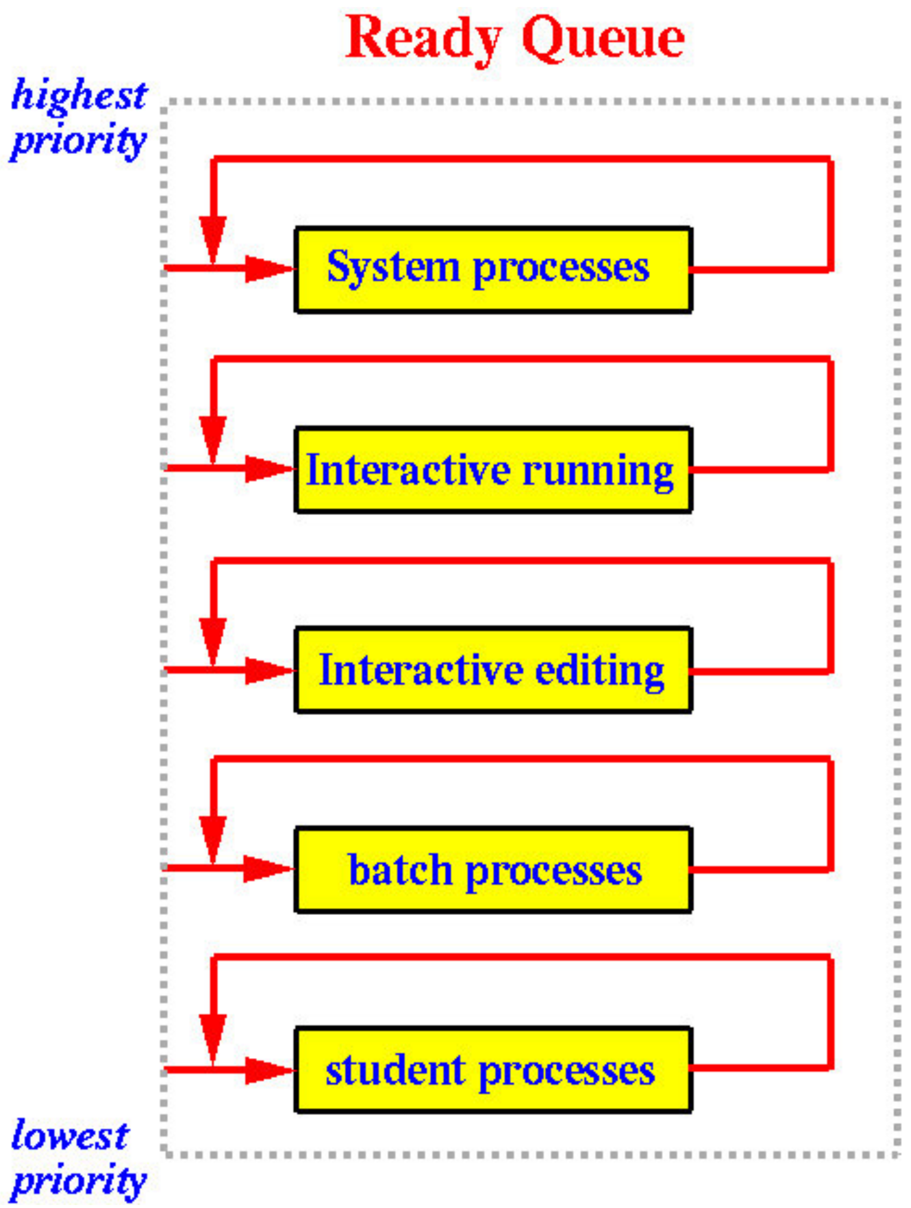
RR Scheduling: Some Issues 5/5

- 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

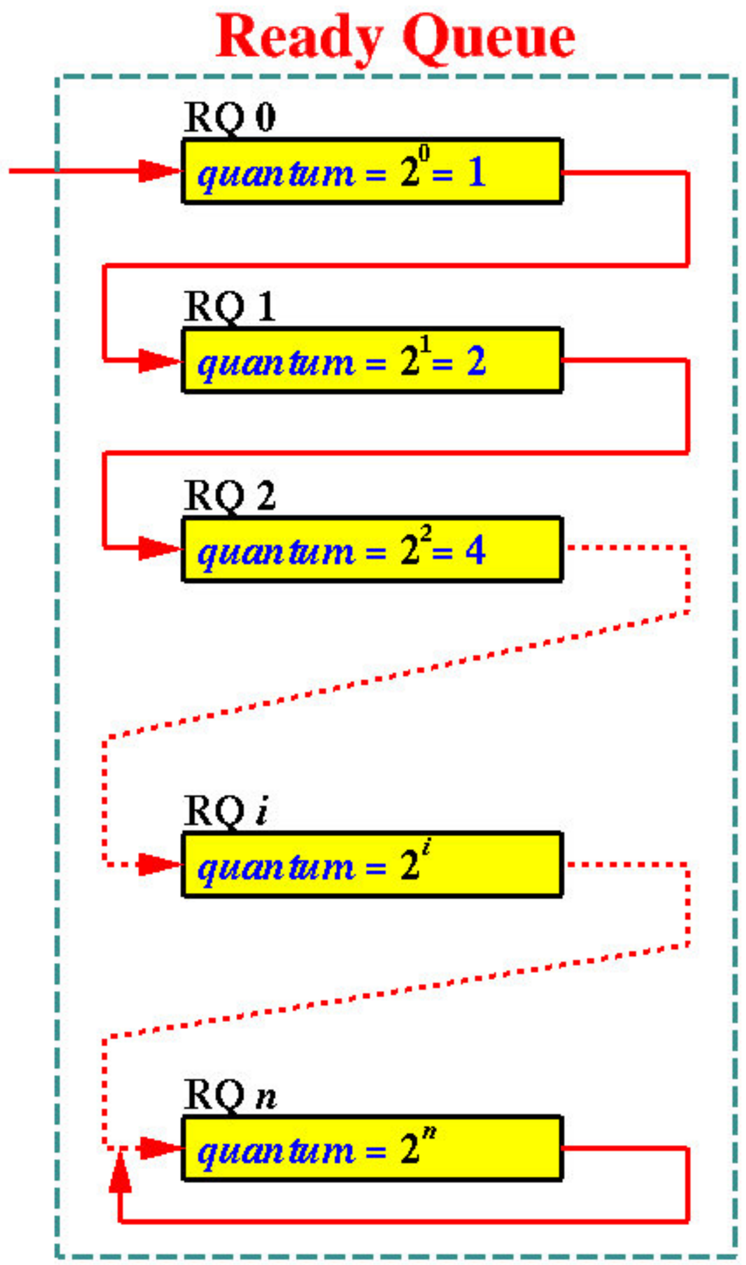


- A process P can run only if all queues above the queue that contains P are empty.
- When a process is running and a process in a higher priority queue comes in, the running process is preempted.

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 moved (*i.e.*, promoted or demoted) to a difference queue.
- Thus, when an I/O-bound (*resp*., CPU-bound) process starts to use more CPU (*resp*., do more I/O), it may be demoted (*resp*., promoted) to a lower (*resp*., higher) queue.

Uniprocessor Summary: 1/3

- FIFO is simple and minimizes overhead.
- If tasks are variable in size, then FIFO can have very poor average response time.
- If tasks are equal in size, FIFO is optimal in terms of average response time.
- Considering only the processor, SJF is optimal in terms of average waiting time.
- SJF is pessimal in terms of variance in response time.

Uniprocessor Summary: 2/3

- If tasks are variable in size, Round Robin approximates SJF. (**Why?**)
- If tasks are equal in size, Round Robin will have very poor average response time.
- Tasks that intermix processor and I/O benefit from SJF and can do poorly under Round Robin.

Uniprocessor Summary: 3/3

- Round Robin and Max-Min fairness both avoid starvation.
- By manipulating the assignment of tasks to priority queues, an MFQ scheduler can achieve a balance between responsiveness, low overhead, and fairness.
- In a large and complex system, scheduling is usually combined with memory or even resource management.

Case Study: IBM VM/370 1/7

- The precursor of VM/370 and CP/CMS, was conceived in 1964 as a second-generation time-sharing system for the newly announced IBM System/360*. (CP = Control Program, CMS = Conversational Monitor System)
- In 1966, CP-40 and CMS both became operational using an IBM System/360 Model 40.
- At about the same time, CP-67 was built to use the address translation feature of the new System/360 Model 67.
- IBM VM/CMS was released in 1972 and was a System/370 reimplementation of the earlier CP/CMS.
- VM/370 (Virtual Machine Facility/370) is a family of virtual machine operating systems. Versions include VM/SP and z/VM.

*For early history, see R. J. Creasy, The Origin of the VM/370 Time-Sharing System, *IBM Journal of Research and Development*, Vol. 25 (1981), No. 5, pp. 483-490.

Case Study: IBM VM/370 2/7

- Under VM/370, more than 16,000 virtual machines can be created, each of which is identical to the underlying hardware IBM System/370. Thus, these VM's are all **self-virtualized machines**.
- Each virtual machine (VM) can load an IBM operating system (e.g., DOS/VS, OS/MVS, even another level of VM).
- The CMS is a single user interactive system.
- The control program (CP) knows the behavior of each VM, CMS included; but, the CP does not know the behavior of the user processes running under an operating system in a VM. The CP considers each VM as a process!

Case Study: IBM VM/370 3/7

- The scheduler schedules the VMs, and the OS running in a VM schedules its processes.
- Because IBM systems are usually very large and run many CPU-bound and I/O-bound processes at the same time, and each process (and each VM) may use a large amount of resource (e.g., virtual memory), scheduling policy has to work with system resource managers.
- Note that the operating system running in a VM could be a very complex one (e.g., OS/MVS). This operating system could run both batch processes and interactive processes.
- As a result, CP classifies all the processes (i.e., virtual machines) it can see into eight categories.

Case Study: IBM VM/370 4/7

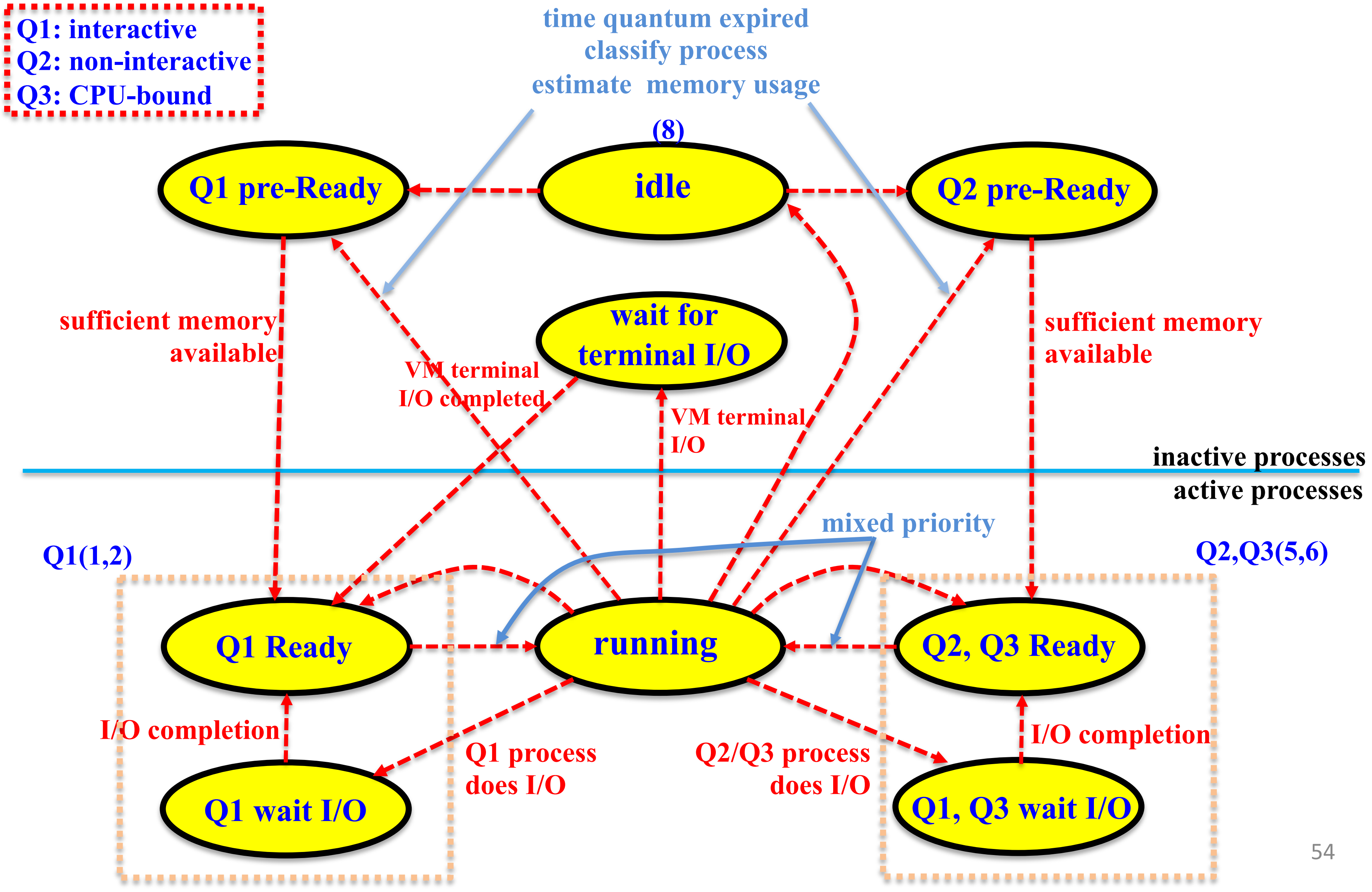
- There are in general three basic types of processes in CP:
  - **Interactive**: Processes use terminal interactive I/O frequently. A process finishing an interactive I/O is classified as interactive; otherwise, non-interactive.
  - **Waiting**: Processes waits on non-interactive I/O or waiting for a page frame (virtual memory).
  - **Idle**: process suspended due to insufficient resource (e.g., memory, number of pages)

Case Study: IBM VM/370 5/7

Eight Types of Processes under CP (Control Program)

Type	Description	
1	Interactive	Ready
2		Wait
3		Suspended
4	wait for completion of terminal I/O	
5	non-interactive	Ready
6		Wait
7		Suspended
8	waiting for external interrupts or stopped	

Case Study: IBM VM/370 6/7



Case Study: IBM VM/370 7/7

- This basically follows the foreground (interactive) background (batch) approach.
- Foreground processes always have higher priority.
- However, CP's scheduler dynamically classifies processes into Q1, Q2 and Q3 types rather than using only foreground and background.
- Time quantum of processes in Q1 and Q2 is shorter than those processes in Q3 (CPU-bound). Additionally, processes in Q3 are less likely to be scheduled to run.
- Therefore, the scheduler in a complex system does not usually work alone, and it has to work with other system components (e.g., memory management).

Real-Time Scheduling: 1/6

- A real-time system is one whose correctness depends on timing as well was functionality.
- Real-time systems have very different requirements, characterized by different metrics:
  - **Timeliness**: how close does it meet its timing requirement
  - **Predictability**: how much deviation is there in delivered timeliness
- More concepts:
  - **Feasibility**: whether or not it is possible to meet the requirements for a particular task set
  - **Hard Real-Time**: discuss later
  - **Soft Real-Time**: discuss later

Real-Time Scheduling: 2/6

- 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 rejects 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: 3/6

- 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.

Real-Time Scheduling: 4/6

- In the simplest real-time systems, where tasks and their execution times are all known, there might not be a scheduler. One task might simply call (or yield to) the next.
- In more complex real-time systems, with a larger (but still fixed) number of tasks that do not function in a strictly pipeline way, it may be possible to do **static scheduling**.
- Based on the list of tasks to be run, and the expected completion time for each, we can define a fixed schedule that will ensure timely execution of all tasks.

Real-Time Scheduling: 5/6

- For many real-time systems, the work-load changes from moment to moment, based on external events. **Dynamic scheduling** is required. There are two key questions:
  1) How they choose the next ready task?
     - ✓ SJF, static priority, soonest start-time deadline first (ASAP), soonest completion-time deadline first (slack time).
  2) How they handle overload (infeasible requirements)
     - ✓ Best effort, periodicity adjustment (run lower priority tasks les often), work shedding (stop running lower priority tasks completely).

Real-Time Scheduling: 6/6

- How about **preemption**?
  - Preempting a running task will almost surely cause it to miss its completion deadline
  - Because we normally know the expected execution time, we can schedule accordingly and should have little need for preemption
  - Embedded and real-time systems run fewer and simpler tasks than general purpose system, and the code is usually much better tested. Therefore, infinite loop and other odd bugs are rare.

Priority Inversion

- What if a high-priority process needs to access the data that is currently being held 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*.
  - The low priority process accessing the data inherits the high priority until it is done with the resource.
  - When the low-priority process finishes, its priority reverts back to the original.

The End