OPERATING SYSTEMS

CPU Scheduling

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

中山大學
SUN YAT-SEN UNIVERSITY

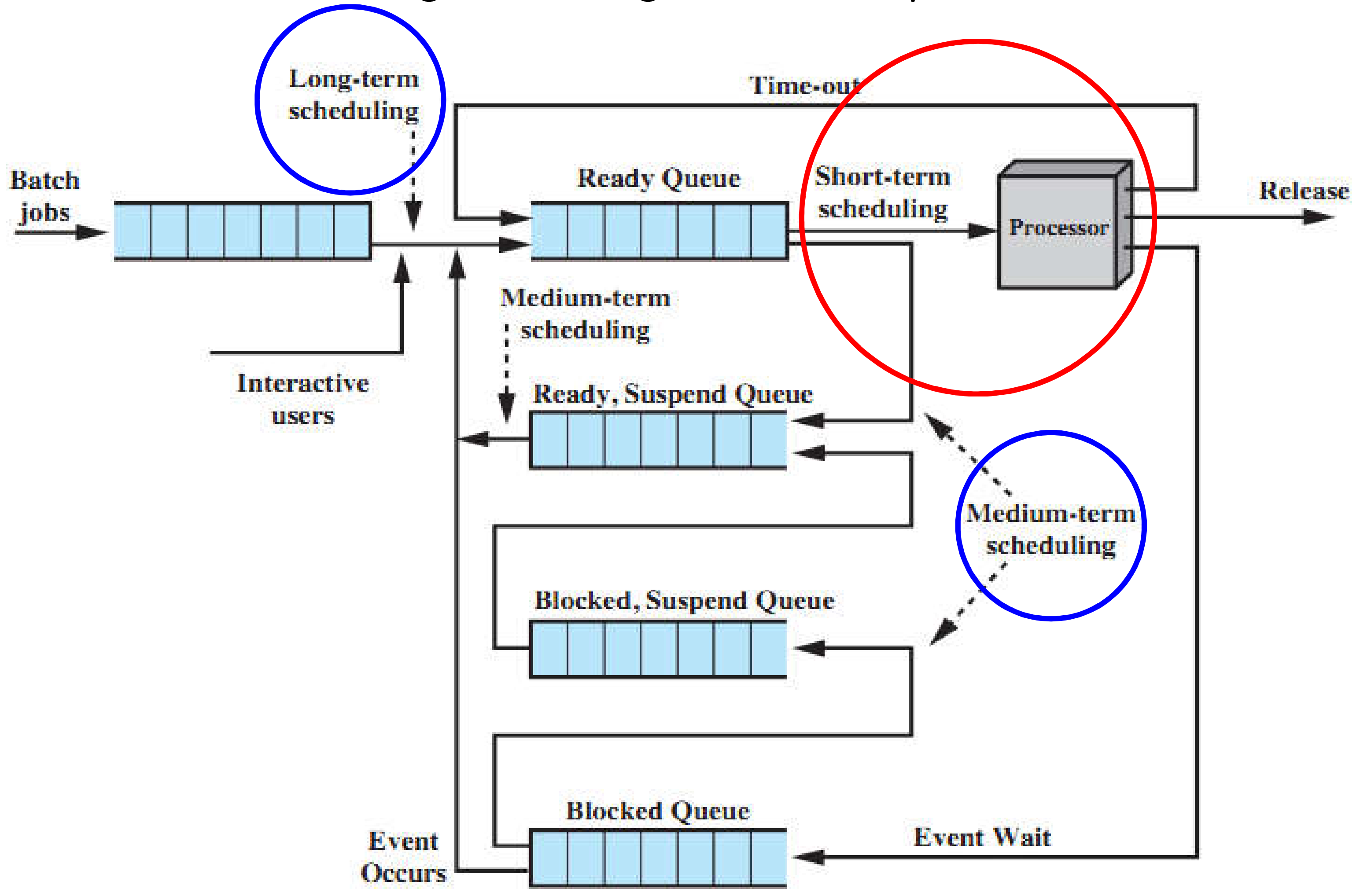
CPU Scheduling

Contents

- Basic Concepts
- Scheduling Criteria
- Simple Scheduling Algorithms
  - First-Come, First-Served Scheduling
  - Shortest-Job-First Scheduling
  - Priority Scheduling
- Advanced Scheduling Algorithms
  - Round-Robin Scheduling
  - Multiple-Priority Queues Scheduling
  - Multilevel Feedback Queue Scheduling
  - Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Algorithms Evaluation

Overview

- Review: queuing diagram for process scheduling
  - Processes migrate among the various queues.



Basic Concepts

Overview

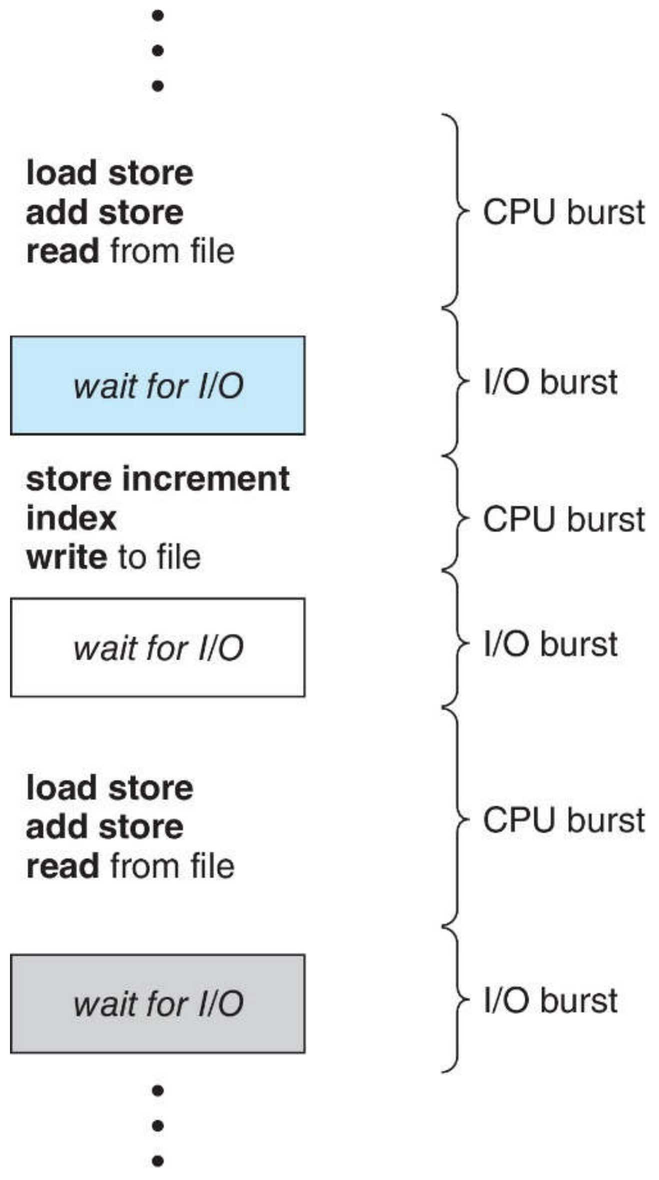
- In the single CPU situation, the objective of multiprogramming is to have processes running at all times, to maximize CPU utilization.
  - Keep several processes in memory at one time
  - Every time one process has to wait, another process can take over use of the CPU.
- CPU is one of the primary computer resources. CPU Scheduling is a fundamental operating-system function and is central to operating-system design.
- Process Scheduling and Thread Scheduling
  - Kernel-level Threads (not processes) are scheduled by the operating system.
  - The terms "process scheduling" and "thread scheduling" are often used interchangeably.
    - We use “process scheduling” when discussing general scheduling concepts.
    - We use “thread scheduling” to refer to thread-specific ideas.

CPU-I/O Burst Cycle

- Process execution consists of a *cycle* of CPU execution and I/O wait. Processes alternate between these two states.
  - Process execution begins with a *CPU burst* (CPU 执行期).
  - That is followed by a (usually longer) *I/O burst* (I/O 执行期), which is followed by another CPU burst, then another I/O burst, and so on.
  - A process usually terminates on a CPU burst.
- CPU burst distribution is of main concern.

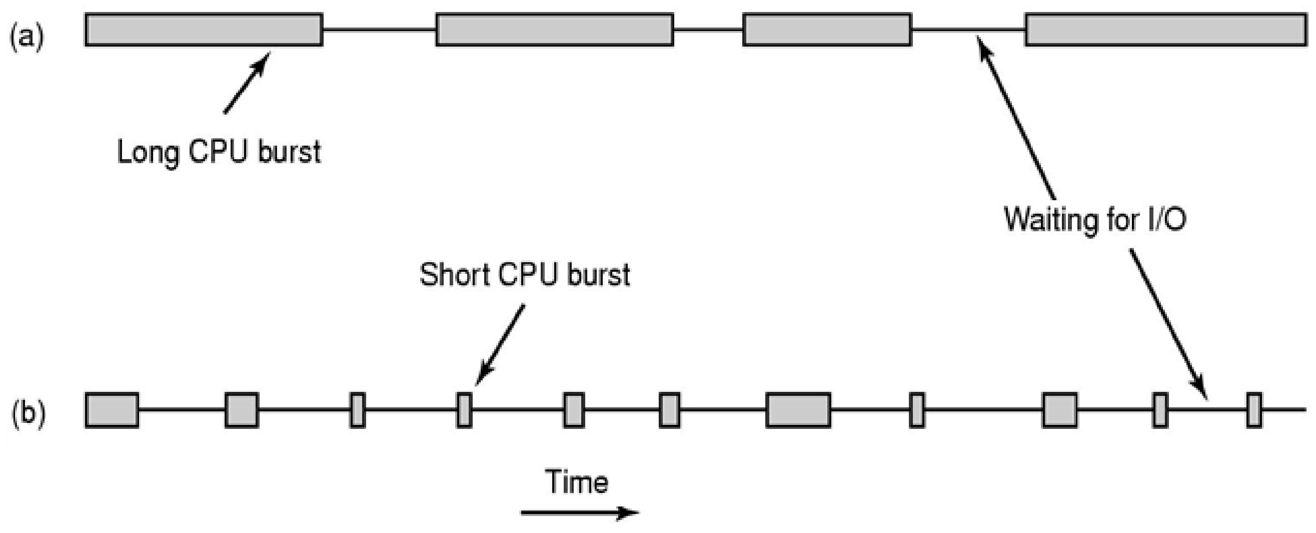
CPU-I/O Burst Cycle

- A *CPU-I/O burst cycle* (CPU-I/O 执行周期) may consist of many CPU bursts and I/O bursts.
- The CPU bursts and I/O bursts are applied in an alternating sequence.



CPU-I/O Burst Cycle

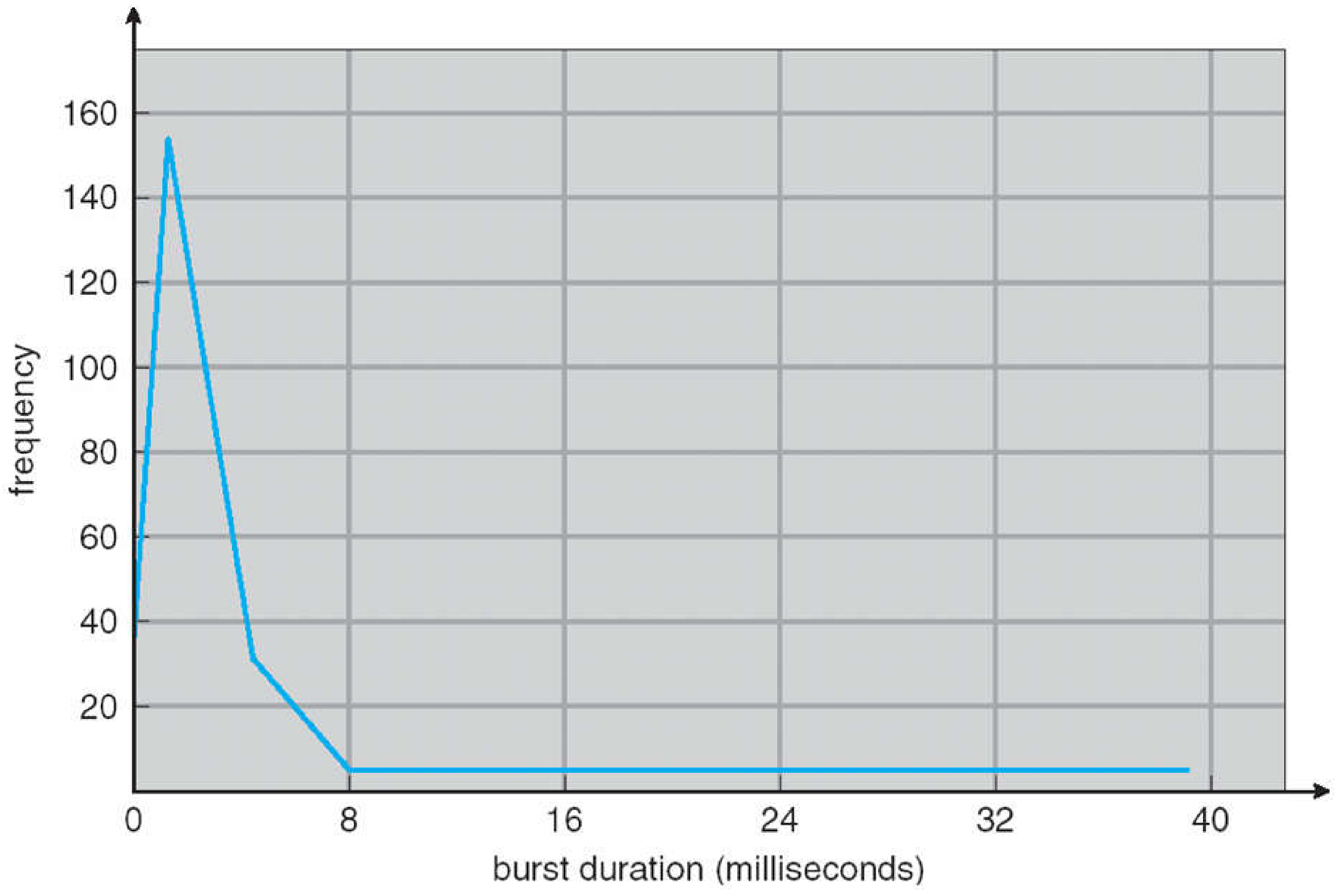
- CPU Burst Time
  - *CPU Burst Time* (CPU 执行时间, or Service Time) is the total processor time needed in one CPU-I/O burst cycle.
- Jobs with long CPU burst time are CPU-bound jobs and are also referred to as "Long Jobs". They might have a few long CPU bursts.
- Jobs with short CPU burst time are I/O-bound jobs and are also referred to as "Short Jobs". They typically have many short CPU bursts.



Long Job (a) vs. Short Job (b)

Basic Concepts

- **CPU-I/O Burst Cycle**
  - Histogram of CPU burst Times (CPU 执行时间的统计直方图)
    - with a large number of short CPU bursts (less than 5ms) and a small number of long CPU bursts



CPU Scheduler

- The short-term scheduler, or *CPU scheduler* determines
  - which process in the *ready queue* in memory is selected next for execution, and
  - allocates the CPU to that process whenever the CPU becomes idle.
- A ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. The records in the queues are generally *process control blocks* (PCBs) of the processes.
- CPU-scheduling decisions may take place when a process:

  (1) switches from the running state to the waiting state

  (2) switches from the running state to the ready state

  (3) switches from the waiting state to the ready state

  (4) terminates.

Basic Concepts

- **CPU Scheduler**
  - Under these four circumstances, there are two *Decision Modes*:
    - *Non-preemptive* or *cooperative* (非抢占的/合作的)
    - *Preemptive* (抢占的)
  - Non-preemptive or cooperative
    - Once the CPU has been allocated to a process, the process will keeps the CPU until it terminates or blocks itself for I/O.
  - Preemptive
    - Currently running process may be interrupted and moved to the ready state.
    - allows for better service since any one process cannot monopolize the processor for very long
  - Preemptive scheduling decision mode is used by most of current operating systems. It can result in race conditions, introducing data inconsistency when data are shared among several processes.

Preemptive Scheduling

- Preemption during *system calls*
  - During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O queues).
  - What happens if the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure?
    - Certain operating systems, including most versions of UNIX, deal with this problem by waiting either for a system call to *complete* or for an I/O *block* to take place before doing a context switch.
      - This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state.
      - It is not good for supporting real-time computing where tasks must complete execution within a given time frame.

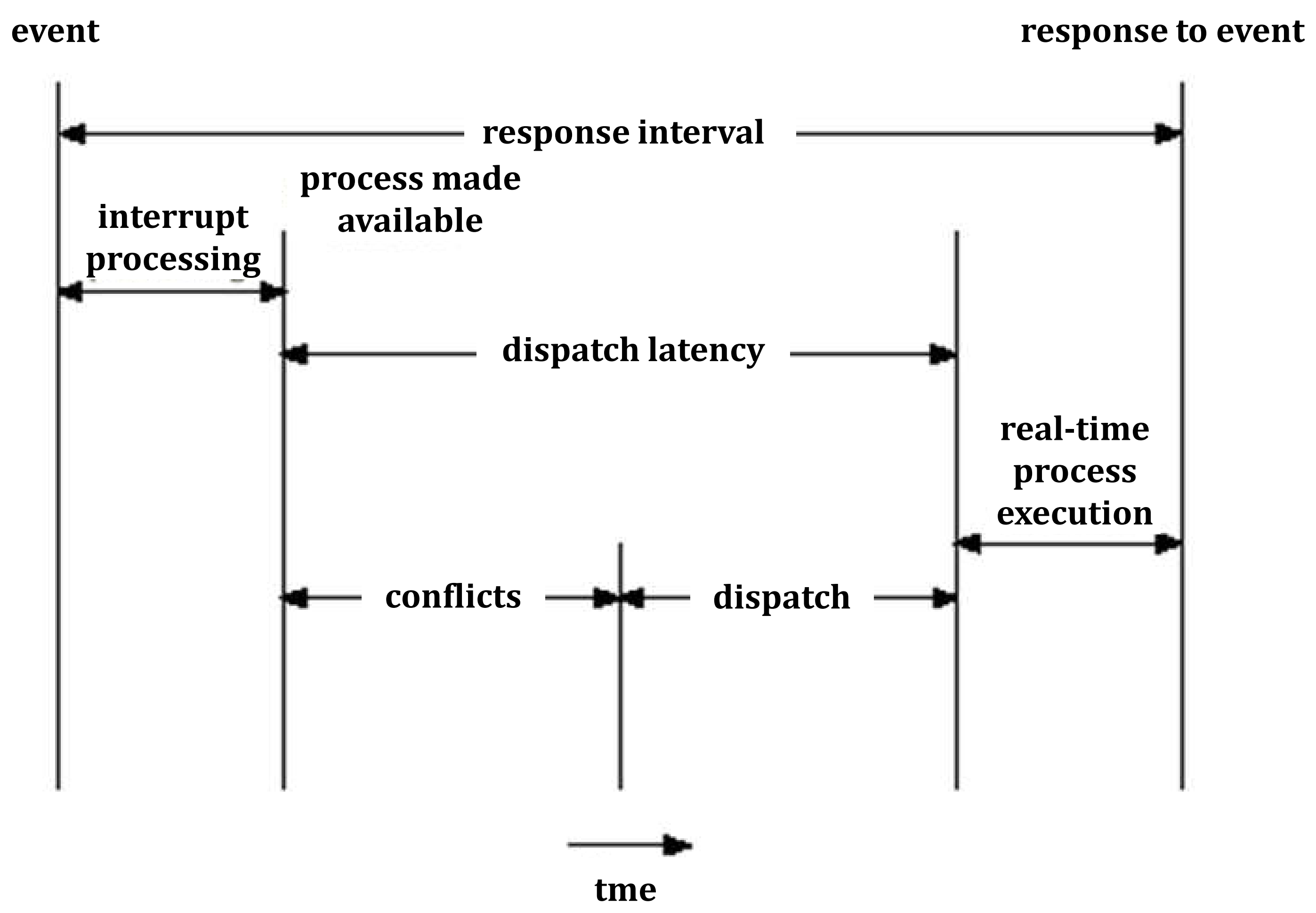
Preemptive Scheduling

- Preemption during *interruption*
  - Interrupts can occur at any time by its definition.
  - OS needs to accept interrupts at almost all times.
    - otherwise, input might be lost or output overwritten
  - The sections of code affected by interrupts must be guarded from simultaneous use.
    - These sections of code are not accessed concurrently by disable interrupts at entry and re-enable interrupts at exit.
    - It is important to note that sections of code that disable interrupts do not occur very often and typically contain few instructions.

Dispatcher 调度器

- *Dispatcher* is another component involved in the CPU-scheduling function. It is the module that gives control of the CPU to the process selected by the short-term scheduler.
- The function of dispatcher involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- Dispatch Latency (分发延时)
  - The dispatcher should be as fast as possible, since it is invoked during every process switch.
  - The time it takes for the dispatcher to stop one process and start another running is known as the *dispatch latency*.

- **Dispatcher**
  - Dispatch Latency



Scheduling Criteria

- **Scheduling Goals of Different Systems**
  - All Systems
    - Fairness (公平性)
      - giving each process a fair share of the CPU
    - Policy enforcement (策略执行能力)
      - seeing that states policy is carried out
    - Balance (平衡能力)
      - keeping all parts of the system busy
  - Batch systems
    - Throughput (吞吐量)
      - maximize jobs per hour
    - Turnaround time (周转时间)
      - minimize time between submission and termination
    - CPU utilization (CPU利用率)
      - keep the CPU busy all the time

Scheduling Criteria

- **Scheduling Goals of Different Systems**
  - Interactive systems
    - Response time (响应时间)
      - respond to requests quickly
    - Proportionality (相当性/平衡性)
      - meet users' expectation
  - Real-time systems
    - Meeting deadlines (及时性/截止期限前完成)
      - avoid losing data
    - Predictability (预见性)
      - avoid quality degradation in multimedia systems (避免多媒体系统的质量退化)

Scheduling Criteria

CPU-Scheduling Criteria

- Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:
  - CPU utilization
    - keeping CPU as busy as possible
  - Throughput
    - the number of processes that complete their execution per time unit
  - Turnaround time
    - amount of time to execute a particular process from start to end
  - Waiting time
    - amount of time a process has been waiting in the ready queue
  - Response time
    - amount of time it takes from when a request was submitted until the first response is produced (for time-sharing environment)

Scheduling Criteria

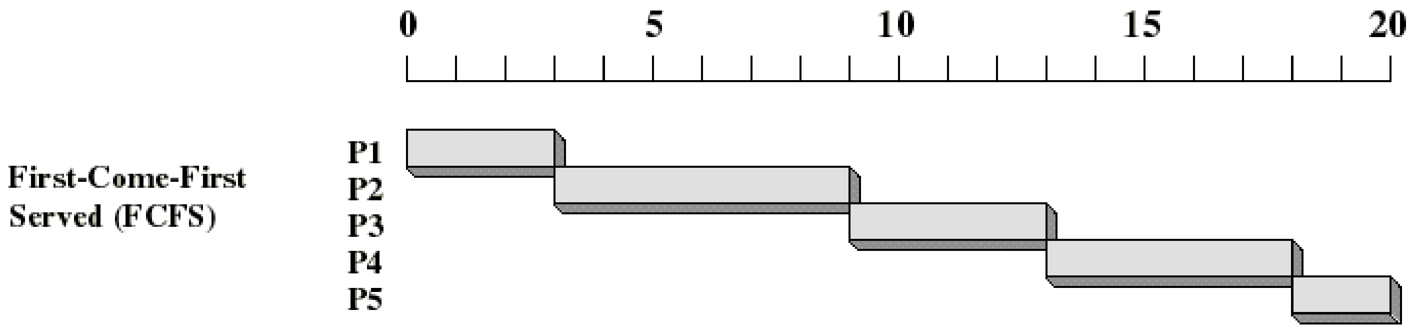
- **CPU-Scheduling Criteria**
  - User-oriented:
    - Response Time
    - Turnaround Time
  - System-oriented:
    - CPU utilization
    - Throughput
    - Fairness
  - Optimization Criteria
    - maximize CPU utilization
    - maximize throughput
    - minimize turnaround time
    - minimize waiting time
    - minimize response time

Simple Scheduling Algorithms

First-Come, First-Served Scheduling

- Selection function: the process that has been waiting the longest in the ready queue – hence called *First-Come, First-Served* (FCFS).
- Implementation: FIFO ready queue
- Decision mode: Non-preemptive
- Example.

Process	Arrival Time	Burst Time
P_1	0	3
P_2	2	6
P_3	4	4
P_4	6	5
P_5	8	2



Simple Scheduling Algorithms

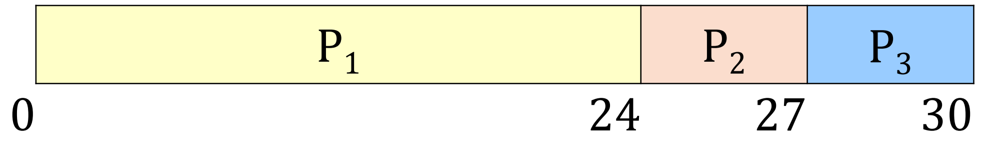
First-Come, First-Served Scheduling

- A Simpler FCFS Example.

Process	Burst Time
P_1	24
P_2	3
P_3	3

甘特图

- Suppose that the processes arrive in the order: P_1, P_2, P_3. The *Gantt Chart* for the schedule is:



- Waiting time: $P_1 = 0$; $P_2 = 24$; $P_3 = 27$.
- Average waiting time:

  $(0 + 24 + 27)/3 = 17$.

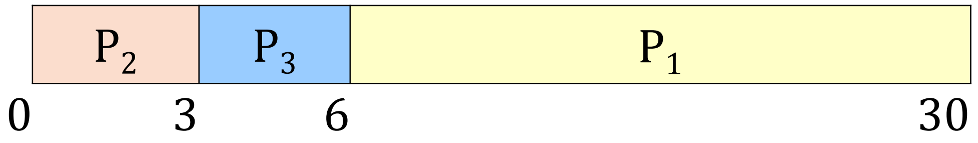
- *Convoy effect*: (护航效应) short process behind long process.
  - Consider one CPU-bound and many I/O-bound processes

Simple Scheduling Algorithms

- **First-Come, First-Served Scheduling**
  - A Simpler FCFS Example.

Process	Burst Time
P_1	24
P_2	3
P_3	3

  - Now suppose that the processes arrive in the order: P_2, P_3, P_1. The *Gantt* Chart for the schedule is:



  - Waiting time: $P_1 = 6$; $P_2 = 0$; $P_3 = 3$.
  - Average waiting time:

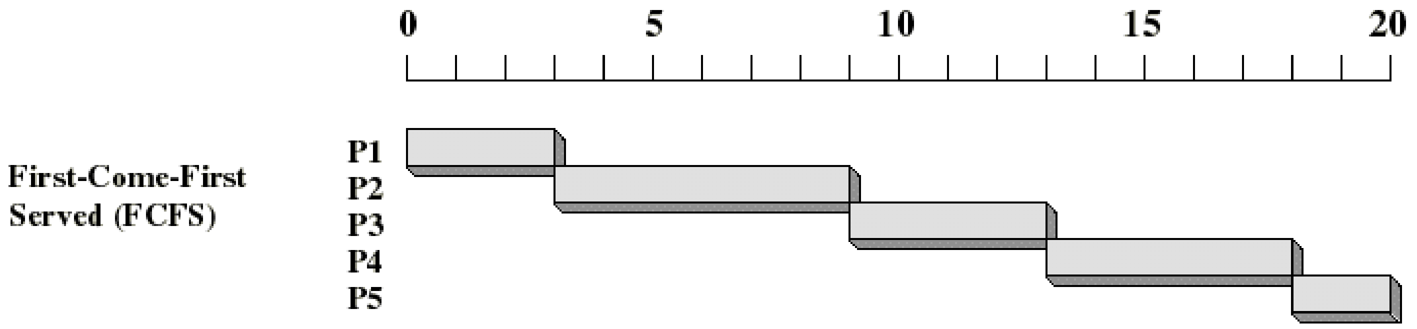
    `(6 + 0 + 3)/3 = 3.`

  - Much better than previous case

Simple Scheduling Algorithms

- **First-Come, First-Served Scheduling**
  - Another FCFS Example.

Process	Arrival Time	Burst Time
P_1	0	3
P_2	2	6
P_3	4	4
P_4	6	5
P_5	8	2



- Waiiting time: P_1 = 0, P_2 = 1, P_3 = 3, P_4 = 7, P_5 = 10.
- Average waiting time:

  (0 + 1 + 3 + 7 + 10)/5 = 21/5 = 4.2

Simple Scheduling Algorithms

- **First-Come, First-Served Scheduling**
  - FCFS Drawbacks
    - A process that does not perform any I/O will monopolize the processor. All the other processes wait for the (one big) process to get off the CPU (Convoy Effect).
    - Favors CPU-bound processes:
      - I/O-bound processes have to wait until CPU-bound process completes.
      - I/O devices may have to wait even when their I/O are completed (poor device utilization).
  - We could have kept the I/O devices busy by giving a bit more priority to I/O bound processes.

Simple Scheduling Algorithms

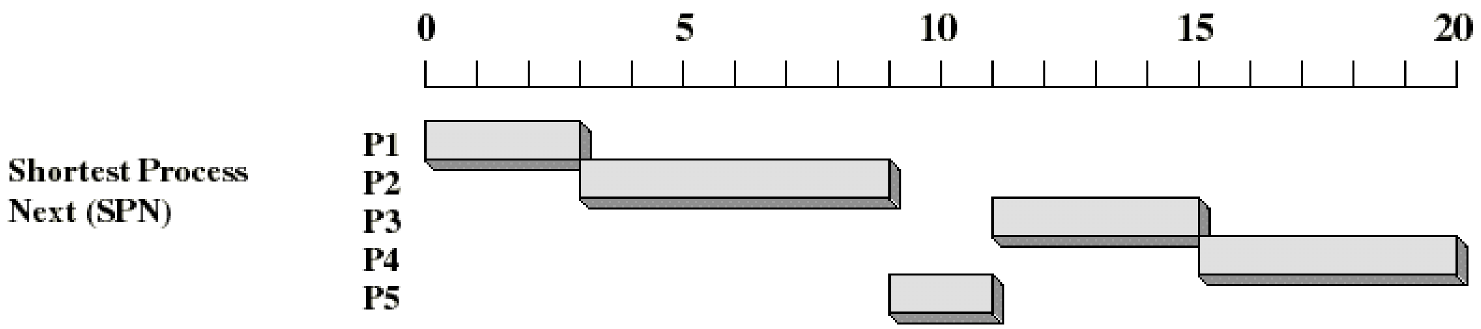
Shortest-Job-First Scheduling

- Selection function: the process with the shortest expected CPU burst time in the ready queue – *Shortest-Job-First* (SJF), called also Shortest-Time-First (STF) and Shortest-Process-Next (SPN)
  - more appropriately Shortest-Next-CPU-Burst
  - need to associate with (somehow estimate) the required processing time (next CPU burst time) for each process
  - If the next CPU bursts of two processes are the same, FCFS used.
- Decision mode: Non-preemptive or Preemptive
- I/O bound processes will be picked first.
- SJF is optimal.
  - It gives minimum average waiting time for a given set of processes.
- How do we know the next CPU burst time?
  - SJF scheduling is used frequently in long-term scheduling where a user specifies the process time when he submits the job to a batch system.
  - With short-term scheduling, there is no way to know the length of the next CPU burst. We may have to approximate SJF scheduling by picking the process with the shortest predicted next CPU burst.

Simple Scheduling Algorithms

- **Shortest-Job-First Scheduling**
  - Example.

Process	Arrival Time	Burst Time
P_1	0	3
P_2	2	6
P_3	4	4
P_4	6	5
P_5	8	2



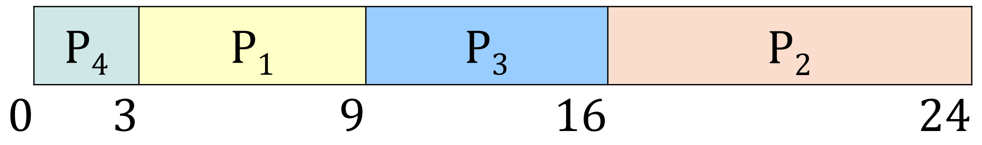
Simple Scheduling Algorithms

Shortest-Job-First Scheduling

- A Simpler SJF Example: suppose all processes arrive at the same time

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart (decision mode: non-preemptive)



- Average waiting time:

  `(3 + 16 + 9 + 0)/4 = 7.`

- Average turnaround time:

  `(9 + 24 + 16 + 3)/4 = 13.`

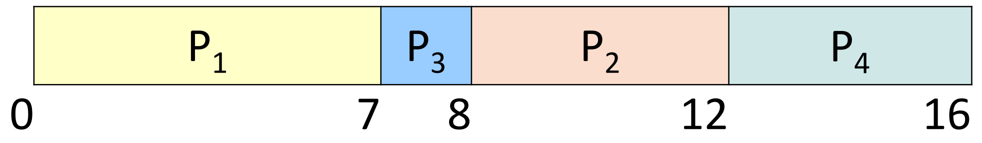
Simple Scheduling Algorithms

Shortest-Job-First Scheduling

- Another SJF Example: processes with different arrival times

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

- SJF scheduling chart (decision mode: non-preemptive)



- Average waiting time:

```
[(0-0)+(8-2)+(7-4)+(12-5)]/4 = 4.
```

- Average turnaround time:

```
[(7-0)+(12-2)+(8-4)+(16-5)]/4 = 8.
```

Simple Scheduling Algorithms

Shortest-Job-First Scheduling

- Prediction of the Length of the Next CPU Burst
  - The next CPU burst of a process is generally predicted as an *exponential average* of the measured lengths of previous CPU bursts of that process:

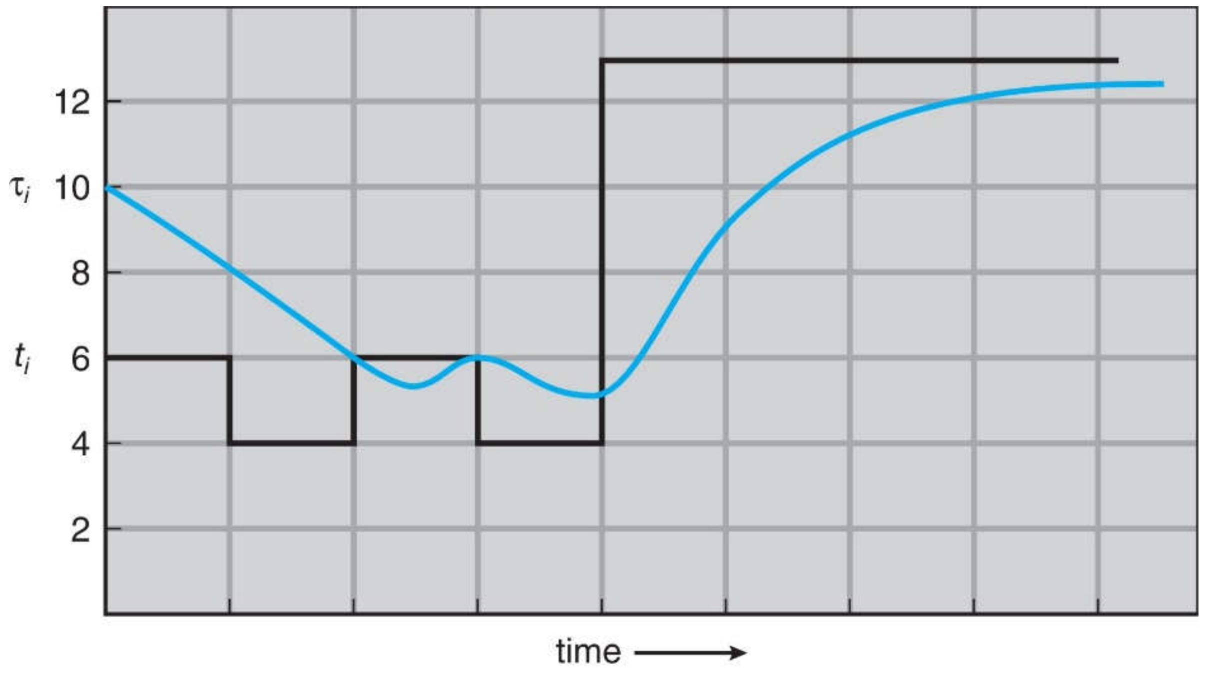
指數平均值

    (1) t_n = actual length of the nth CPU burst
    (2) τ_{n+1} = predicted value for the next CPU burst
    (3) α, $0 \leq \alpha \leq 1$
    (4) Define $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

  - How to set α in?
    - $\alpha = 0$
      - $\tau_{n+1} = \tau_n$,
      - Recent history t_n does not count.
    - $\alpha = 1$
      - $\tau_{n+1} = t_n$,
      - Only the actual last CPU burst counts.
    - Being balanced, let $\alpha = 0.5$ and $\tau_0 = 10$.

Shortest-Job-First Scheduling

- Prediction of the Length of the Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	. . .
"guess" (τ_i)	10	8	6	6	5	9	11	12	. . .

Simple Scheduling Algorithms

- **Shortest-Job-First Scheduling**
  - Idea of Exponential Averaging
    - Expanding the formula $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$, we get:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots$$
$$+(1 - \alpha)^j \alpha t_{n-j} + \ldots$$
$$+(1 - \alpha)^{n+1}\tau_0$$

预测可信度

指数级下降

    - Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor; term weights are decreasing exponentially.
    - Exponential averaging here is better than simple averaging.
  - SJF scheduling can be either preemptive or non-preemptive.
    - The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.
    - Preemptive SJF scheduling is sometimes called *shortest-remaining-time-first* (SRTF) scheduling.

Simple Scheduling Algorithms

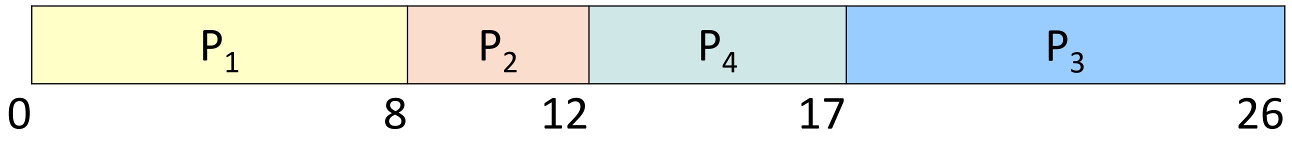
- **Shortest-Job-First Scheduling**
  - Shortest-Job-First Drawbacks
    - Possibility of starvation for longer processes as long as there is a steady supply of shorter processes
    - Lack of preemption is not suited in a time sharing environment:
      - CPU-bound process gets lower priority (as it should) but a process doing no I/O could still monopolize the CPU if he is the first one to enter the system.
    - SJF implicitly incorporates priorities: shortest jobs are given preferences. (优先级隐含)

Simple Scheduling Algorithms

- **Shortest-Job-First Scheduling**
  - SJF Example: non-preemptive scheduling

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

- SJF scheduling chart (decision mode: non-preemptive)



- Average waiting time:

```
[(0-0)+(8-1)+(17-2)+(12-3)]/4 = 7.75.
```

Simple Scheduling Algorithms

- **Shortest-Job-First Scheduling**
  - SRTF Example: preemptive scheduling of SJF

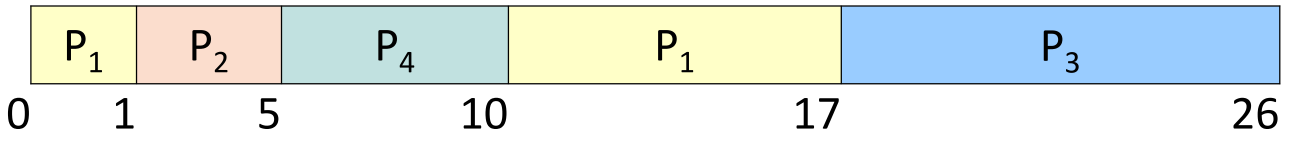
可抢占

最短剩余时间优先级

最高

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

- SRTF scheduling chart (decision mode: preemptive)



- Average waiting time:

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

Simple Scheduling Algorithms

- **Priority Scheduling**
  - A priority number is associated with each process.
  - The CPU is allocated to the process with the highest priority.
    - smallest integer = highest priority
    - SJF is a priority scheduling where priority is the predicted next CPU burst time.
  - Priorities can be defined either internally or externally.
    - Internally defined priorities use some measurable quantity(s) to compute the priority of a process.
      - E.g., time limits, memory requirements, the number of open files, the ratio of average I/O burst to average CPU burst, etc.
    - External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Simple Scheduling Algorithms

- **Priority Scheduling**
  - Decision modes: Preemptive or Non-preemptive.
    - When a process arrives at the ready queue and the priority of this newly arrived process is higher than the priority of the currently running process, what happens next?
      - Preemptive priority scheduling will let the newly arrived process preempt the CPU, putting the running process at the head of the ready queue.
      - Non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
  - Starvation – indefinite blocking
    - Low priority processes may never execute.
    - Solution: Aging strategy
      - gradually increasing the priority of processes that wait in the system for a long time

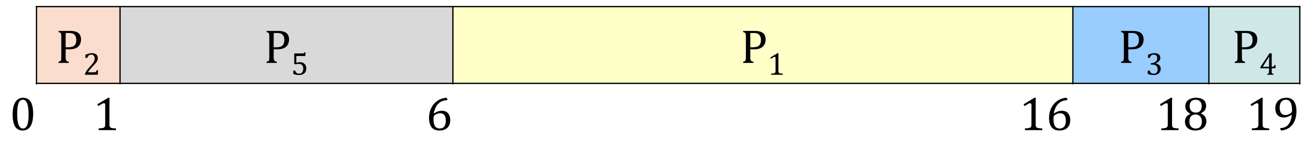
Simple Scheduling Algorithms

Priority Scheduling

- Example of Priority Scheduling: all processes arrive at the same time

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

- Priority scheduling *Gantt* Chart.



- Average waiting time:

```
(6 + 0 + 16 + 18 + 1)/5 = 8.2
```

Advanced Scheduling Algorithms

Round-Robin Scheduling

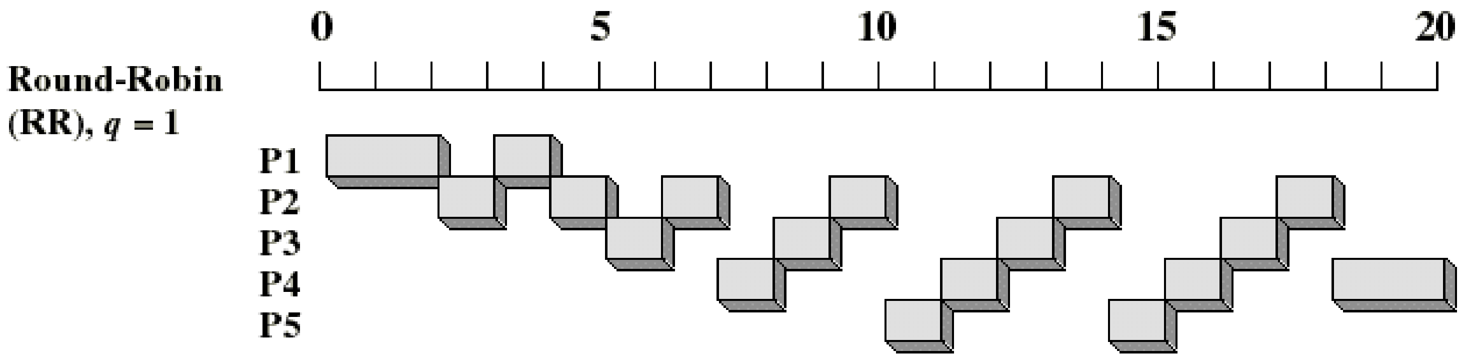
- The *Round-Robin* (RR, 轮转) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
  - each process assigned a small unit of CPU time (*Time Quantum*)
  - a process allowed to run until the set time slice period (the time quantum) is reached. A clock interrupt occurs
  - the running process preempted and added to the end of the ready queue, while a context switch will be executed
- Timer interrupts every time quantum to schedule next process. The ready queue is treated as *a circular queue*. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- Selection function: (initially) same as FCFS
- Decision mode: Preemptive

Round-Robin Scheduling

- Example of RR with Time Quantum q = 1.

进程到达时间可能不在 Quantum 边界

Process	Arrival Time	Burst Time
P_1	0	3
P_2	2	6
P_3	4	4
P_4	6	5
P_5	8	2



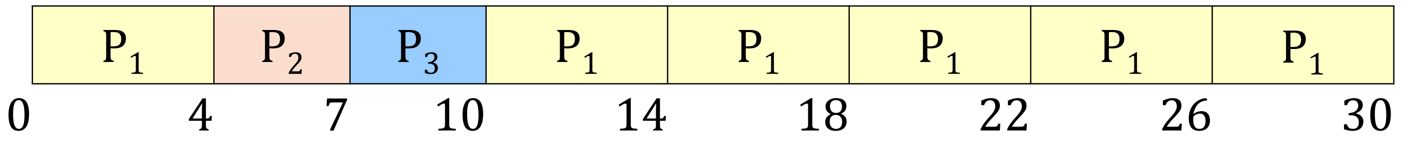
Advanced Scheduling Algorithms

Round-Robin Scheduling

- Example of RR with Time Quantum q = 4.

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The *Gantt* chart is

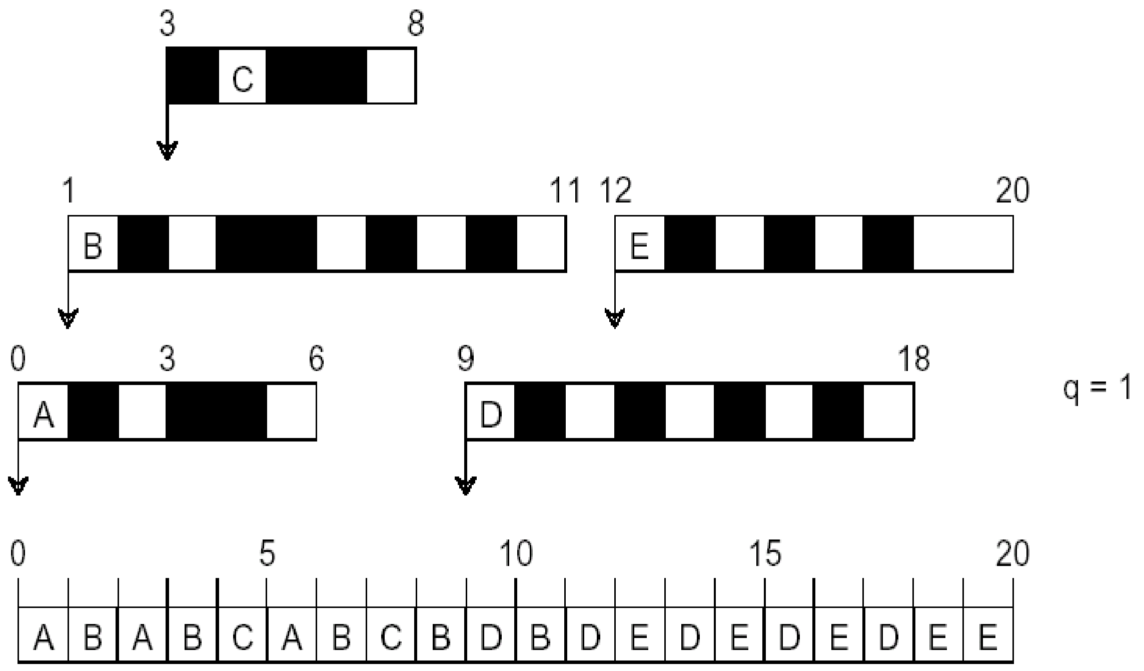


- Typically, more average waiting time and better *response*
- The time quantum q should be large compared to context switch time, or high overhead occurs.
  - q usually from 10ms to 100ms, and context switch < 10μs
  - Extremely large q will make RR scheduling degrade into FCFS.

Round-Robin Scheduling

- Example of RR with Time Quantum q = 1.

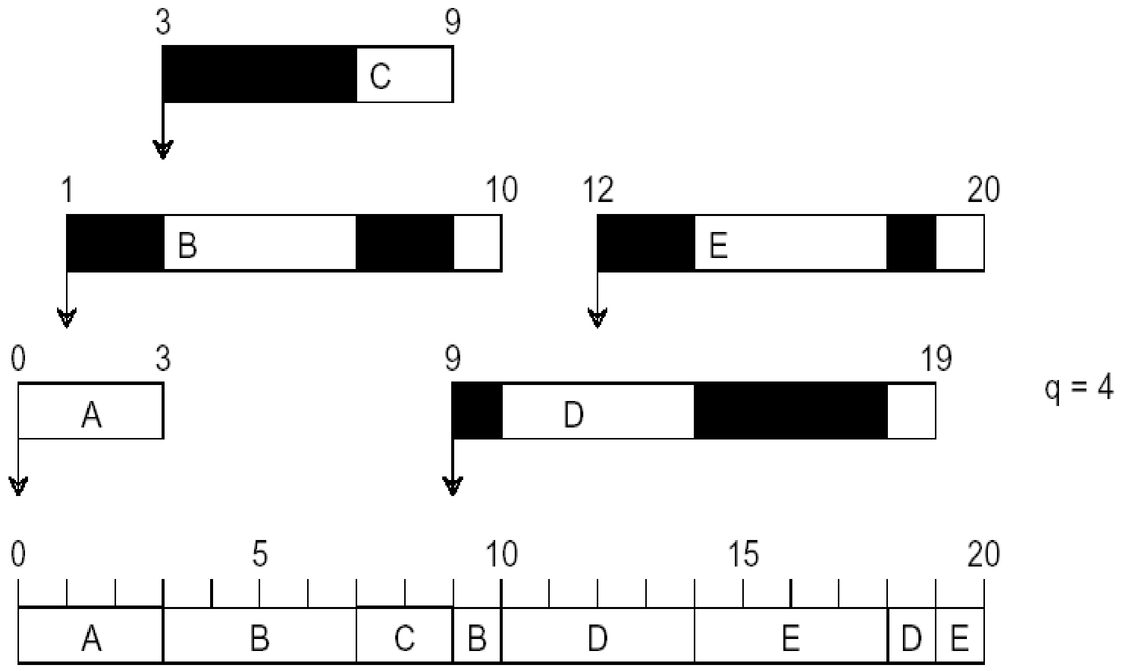
Process	Arrival Time	Burst Time
A	0	6
B	1	10
C	3	5
D	9	9
E	12	8



Round-Robin Scheduling

- Example of RR with Time Quantum q = 4.

Process	Arrival Time	Burst Time
A	0	3
B	1	9
C	3	6
D	9	10
E	12	8

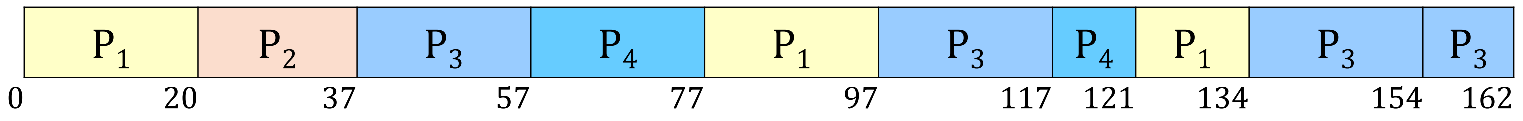


Round-Robin Scheduling

- Example of RR with Time Quantum q = 20.

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

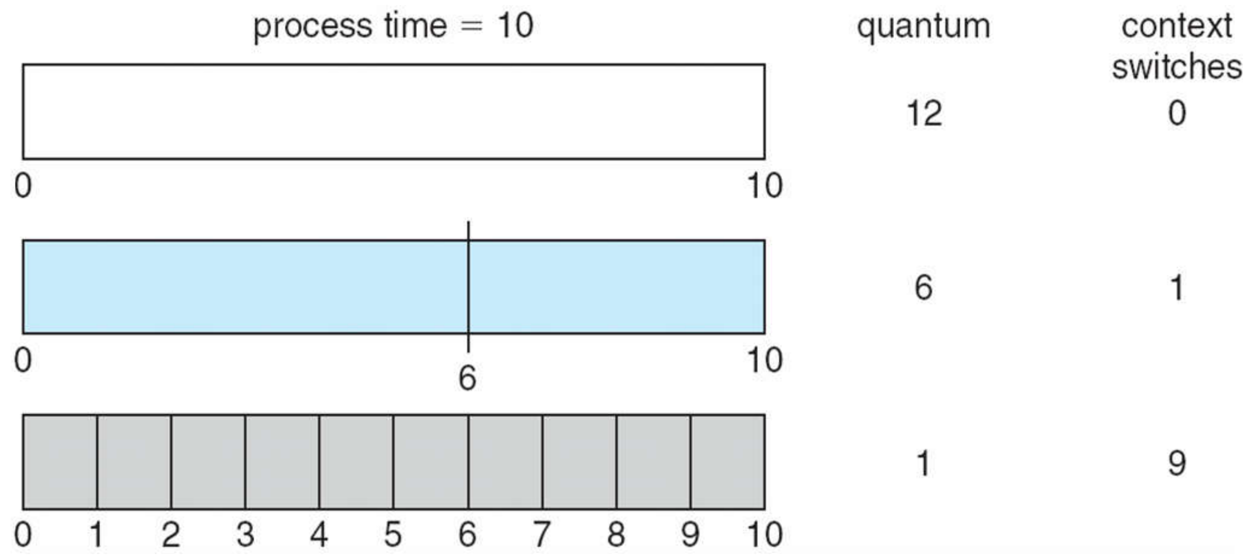
  - The *Gantt* chart is



- If there are n processes in the ready queue and the time quantum is q, then no process waits more than (n - 1) × q time units.

Round-Robin Scheduling

- Example: Small time quantum increases context switches.
  - Suppose we have only one process of 10 time units.

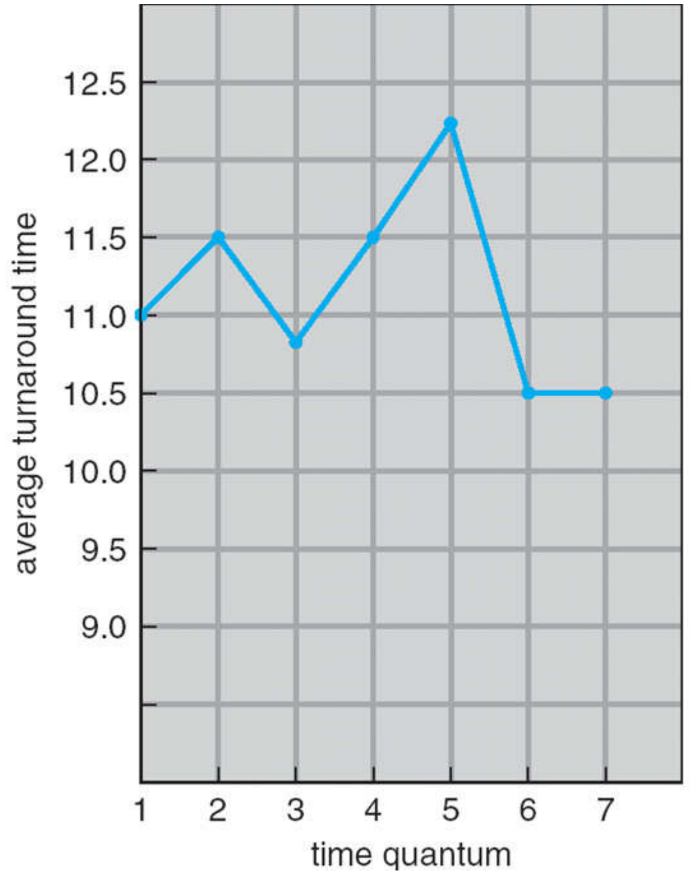


  - If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching.
  - In practice, most modern systems have time quanta ranging from 10ms to 100ms. The context switch time is typically less than 10μs, a small fraction of the time quantum.

Advanced Scheduling Algorithms

Round-Robin Scheduling

- Example: Turnaround time varies with the time quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

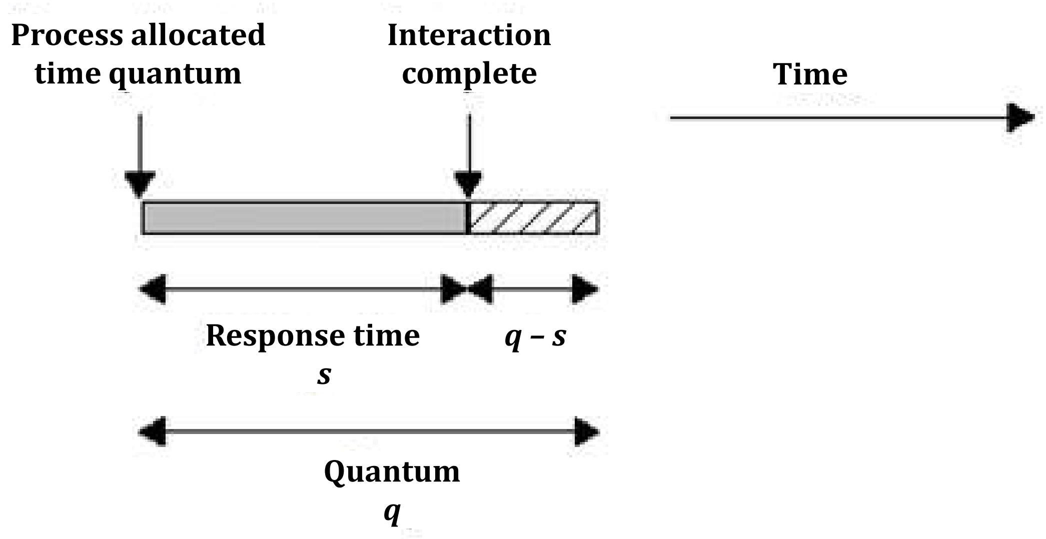
- The average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases.

Round-Robin Scheduling

- Example: Turnaround time varies with the time quantum
  - Turnaround time also depends on the size of the time quantum.
    - The average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases.
  - In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.
    - For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29. If the time quantum is 10, however, the average turnaround time drops to 20.
  - In addition, if context-switch time is added in, the average turnaround time increases even more for a smaller time quantum, since more context switches are required.

Round-Robin Scheduling

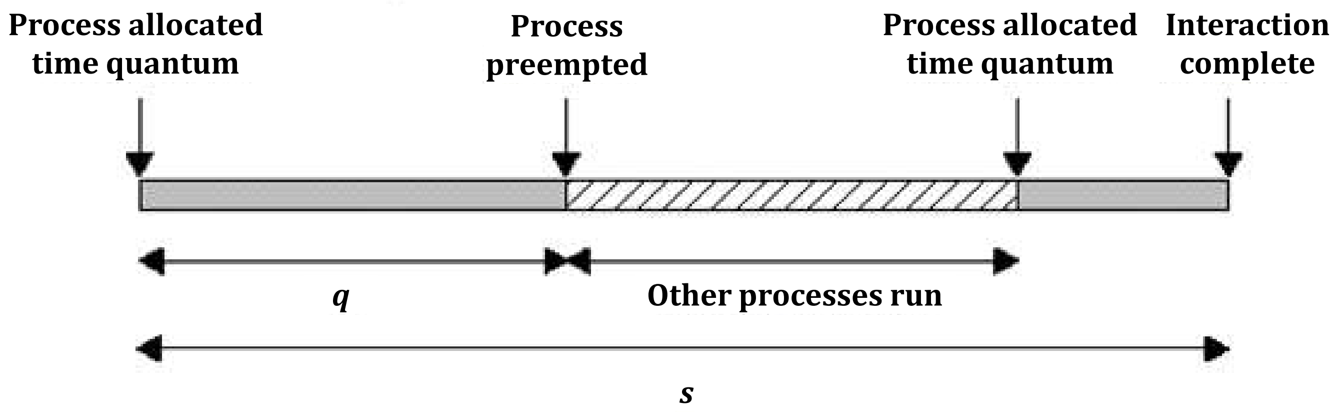
- Time Quantum for Round Robin - more discussion
  - Time Quantum must be substantially larger than the time required to handle the clock interrupt and dispatching.
    - It should be larger than the typical interaction (but not much more to avoid penalizing I/O-bound processes).



Time quantum greater than typical interaction

Round-Robin Scheduling

- Time Quantum for Round Robin - more discussion
  - Time Quantum must be substantially larger than the time required to handle the clock interrupt and dispatching.
    - It should be larger than the typical interaction (but not much more to avoid penalizing I/O-bound processes).


Process allocated time quantum
Process preempted
Process allocated time quantum
Interaction complete
q
Other processes run
s

Time quantum less than typical interaction

Round-Robin Scheduling

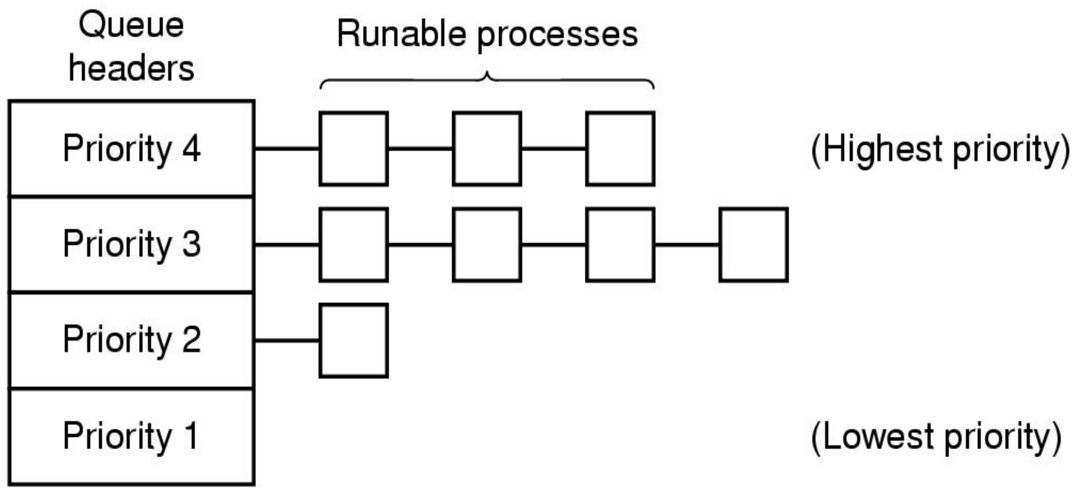
- Round Robin Drawbacks
  - Still favors CPU-bound processes
    - An I/O-bound process uses the CPU for a time that is less than the time quantum and then blocked waiting for I/O.
    - A CPU-bound process runs for all its time slice and is put back into the ready queue (thus getting in front of blocked processes).
  - A solution - virtual round robin
    - When an I/O has completed, the blocked process is moved to an auxiliary queue which gets preference over the main ready queue.
    - A process dispatched from the auxiliary queue runs no longer than the basic time quantum minus the time spent running since it was selected from the main ready queue.

Multiple-Priority Queues Scheduling

- In priority scheduling, every process has a priority number. Consider the situation that the number of processes is far larger than the level of priorities.
- Multiple-Priority Queues Scheduling is implemented by having multiple ready queues to represent each level of priority.
  - Scheduler will always choose a process of higher priority over one of lower priority.
  - With the processes in the same priority ready queue, FCFS scheduling is used to select the next process to allocate CPU.
  - Lower-priority may suffer starvation. Then allow a process to change its priority based on its age or execution history with so-called dynamic multiple priority mechanisms.

Advanced Scheduling Algorithms

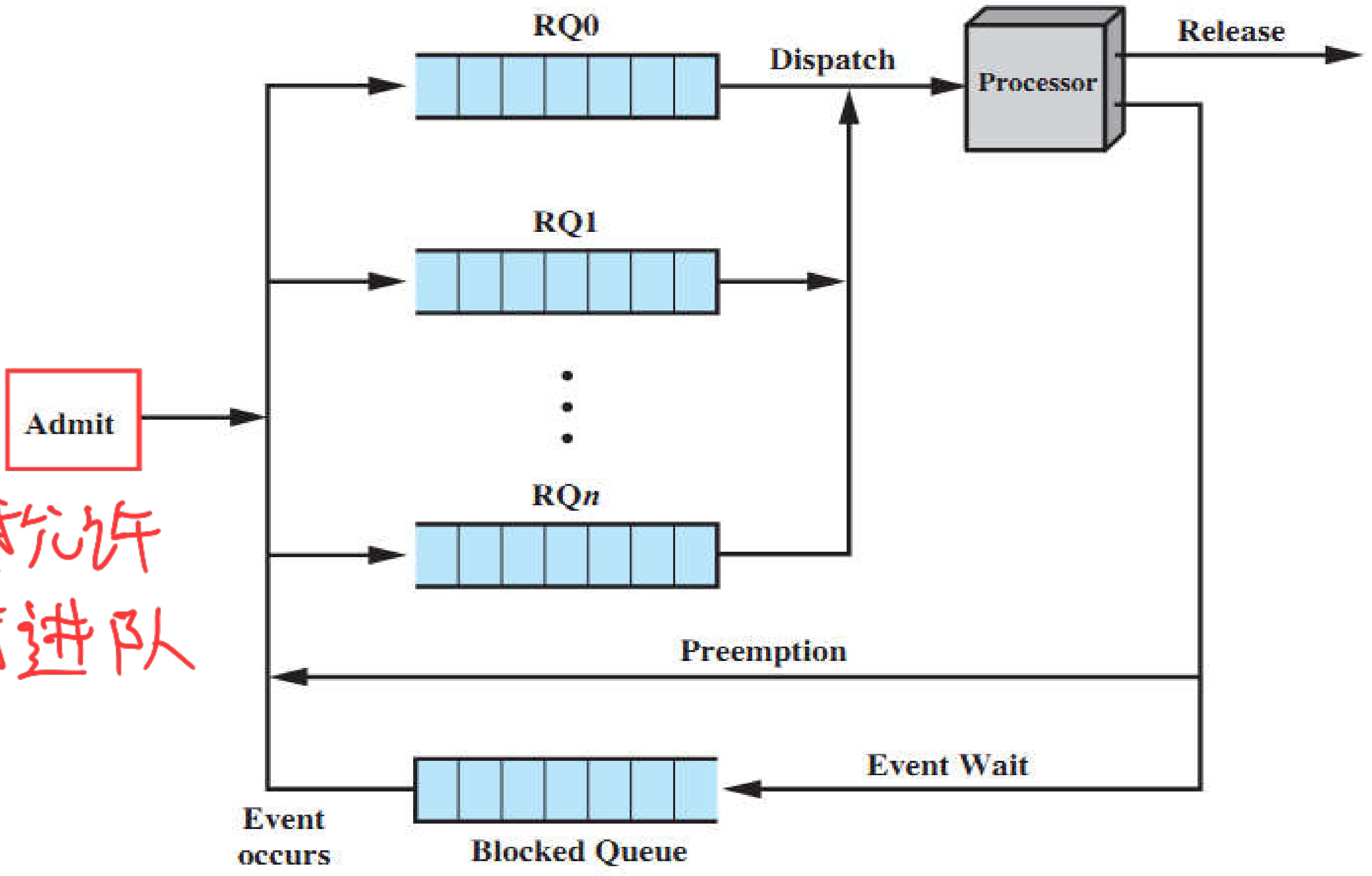
- **Multiple-Priority Queues Scheduling**
  - Priority Scheduling with Queues.



Advanced Scheduling Algorithms

Multiple-Priority Queues Scheduling

- Priority Queuing.



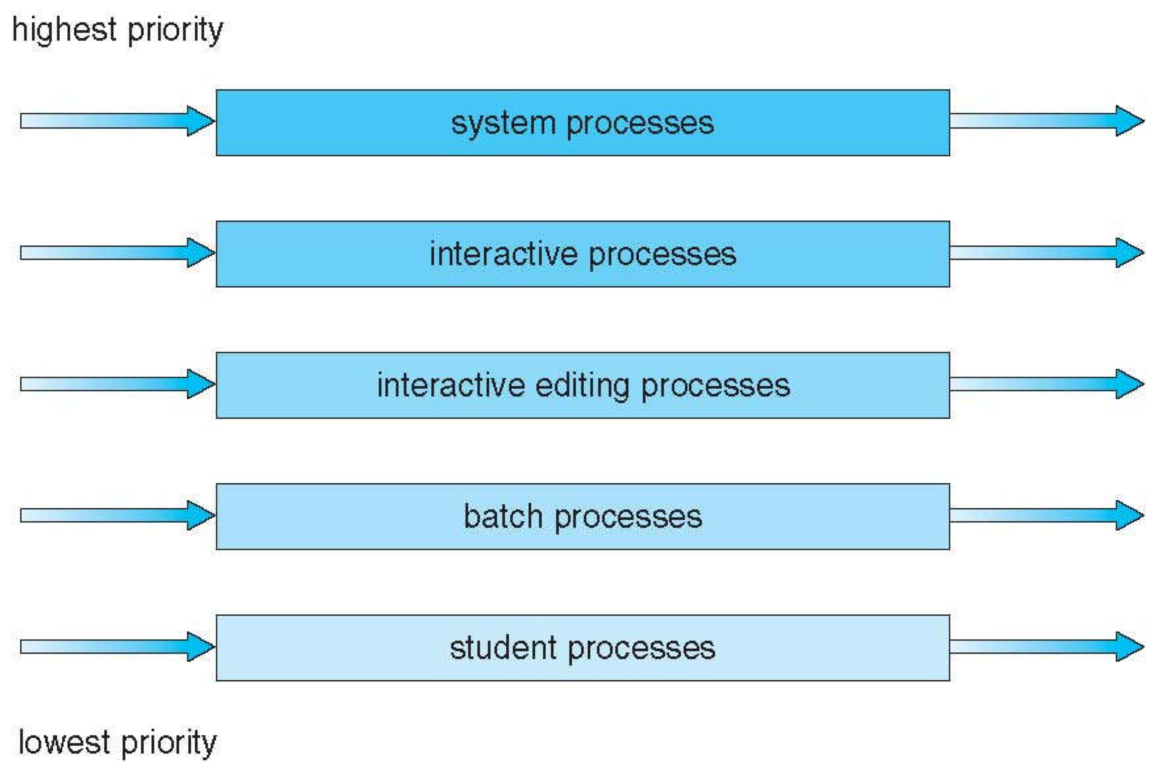
经过允许
才可进队

Multilevel Queue Scheduling

- The *Multilevel Queue Scheduling* algorithm partitions the ready queue into several separate queues.
  - The processes are permanently in a given queue, generally based on some property of the process, such as memory size, process priority, or process type.
  - Each queue has its own scheduling algorithm.
  - e.g., ready queue is partitioned into separate queues:
    - Foreground (interactive) scheduled by an RR algorithm
    - Background (batch) scheduled by an RR algorithm
- Scheduling must be done among the queues.
  - Fixed-priority preemptive scheduling
    - e.g., the foreground queue may have absolutely priority over the background queue – possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule among its various processes
    - e.g., 80% for foreground queue in RR; 20% for background in FCFS.

Multilevel Queue Scheduling

- Example: multilevel queue scheduling algorithm with five queues, listed below in order of priority



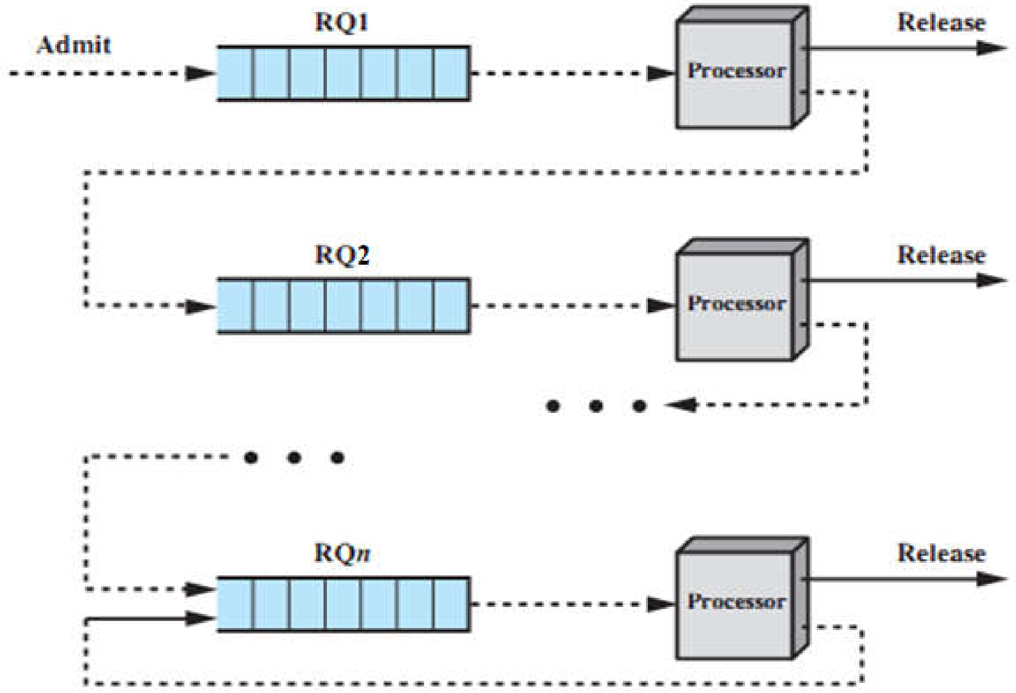
  - If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Multilevel Feedback Queue Scheduling

- The *Multilevel Feedback Queue Scheduling* algorithm allows a process to move between different level queues.
  - The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
  - In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.
  - A multilevel feedback queue scheduler is defined by the following parameters:
    - The number of queues
    - The scheduling algorithms for each queue
    - The method used to determine which queue a process will enter when that process needs service
    - The method used to determine when to upgrade a process
    - The method used to determine when to demote a process

Multilevel Feedback Queue Scheduling

- Example.



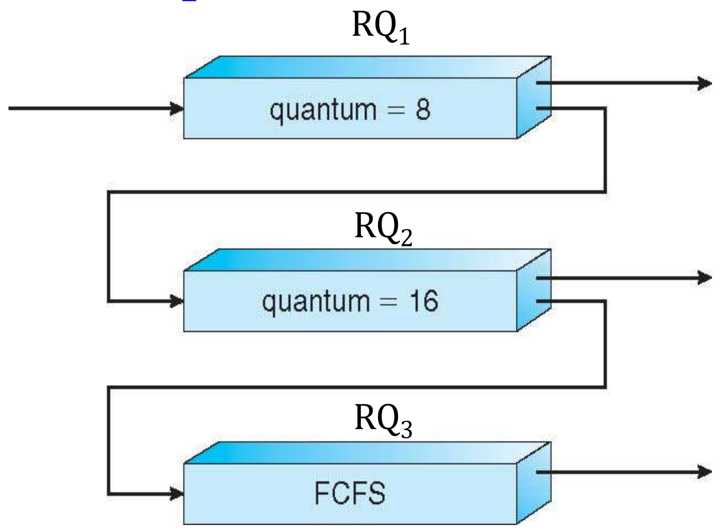
Multilevel Feedback Queue Scheduling

- Example.
  - Several ready queues RQ_1, RQ_2, …, RQ_n with decreasing priorities:

    $P(RQ_1) > P(RQ_2) > ... > P(RQ_n)$.
  - New process are placed in RQ_1.
  - When they reach the time quantum, they are placed in RQ_2. If they reach it again, they are place in RQ_3, …, until they reach RQ_n.
  - I/O-bound processes will tend to stay in higher priority queues. CPU-bound jobs will drift downward.
  - Dispatcher chooses a process (for execution) in RQ_i only if RQ_1 to RQ_{i-1} are empty.
  - Hence long jobs may starve.

Multilevel Feedback Queue Scheduling

- Example.
  - Considering three ready queues:
    - RQ_1 – RR with time quantum 8 milliseconds and FCFS
    - RQ_2 – RR with time quantum 16 milliseconds and FCFS
    - RQ_3 – FCFS
  - Scheduling:
    - A new job P enters queue RQ_1 which is served FCFS. When it gets CPU, job P receives 8 milliseconds. If it does not finish in 8 milliseconds, job P is moved to queue RQ_2.
    - At RQ_2 job P is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue RQ_3.
    - Could be also vice versa.



Multilevel Feedback Queue Scheduling

- Time quantum for Feedback Scheduling
  - With a fixed quantum time, the turnaround time of longer processes can stretch out alarmingly.
  - To compensate this we can increase the time quantum according to the depth of the queue:
    - Example: define time quantum of the i^{th} ready queue as

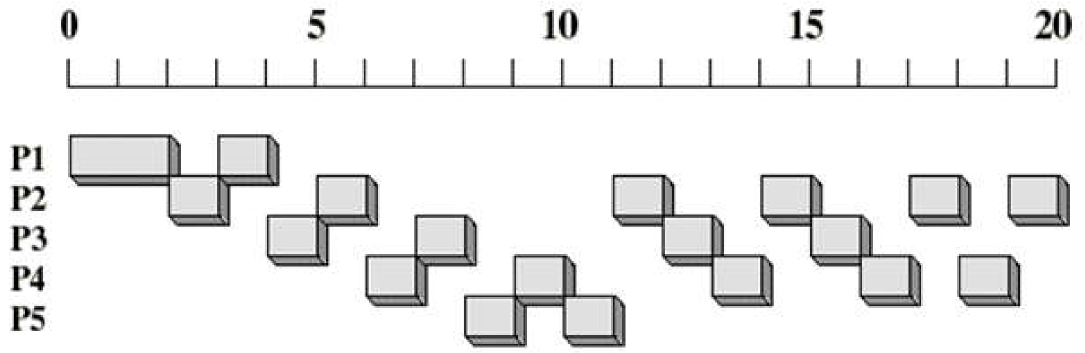
$$q_i = 2^{i-1}$$

  - See next slide for an example
  - Longer processes may still suffer starvation.
    - Possible fix: promote a process to higher priority after some time.

Advanced Scheduling Algorithms

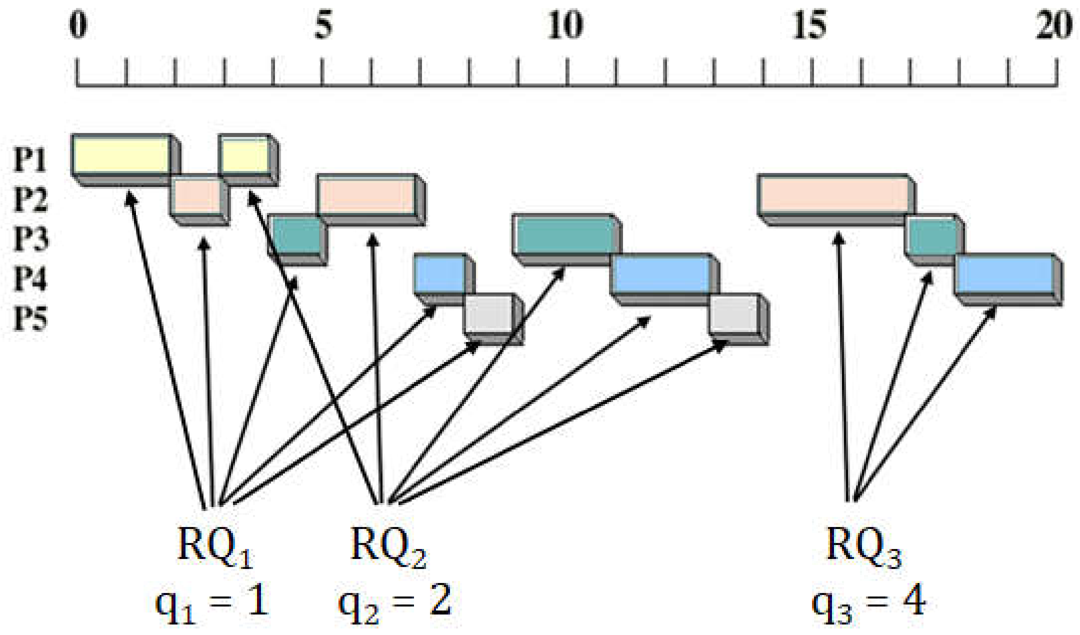
Multilevel Feedback Queue Scheduling

- Example: Feedback Scheduling with q = 1.



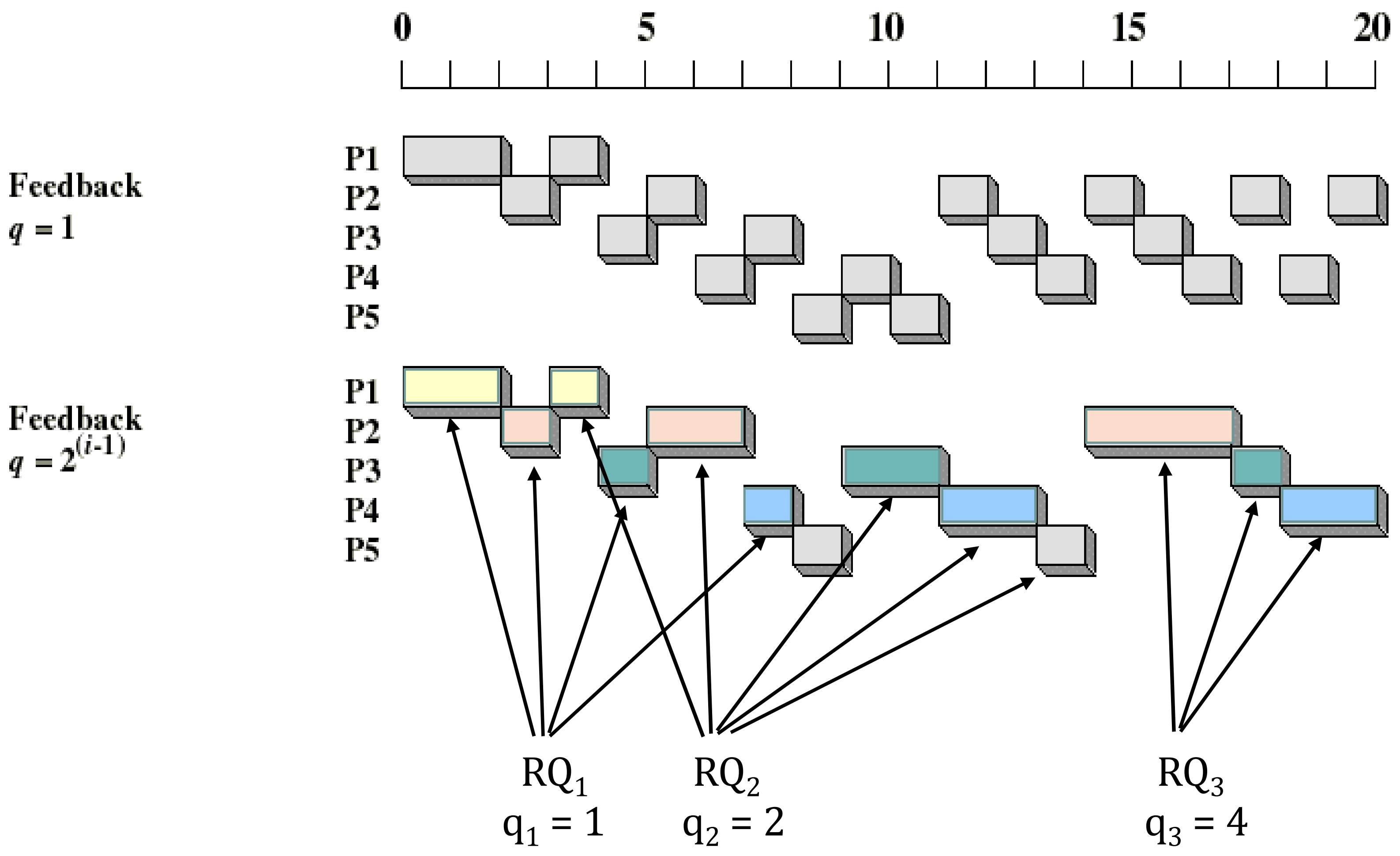
Multilevel Feedback Queue Scheduling

- Example: Feedback Scheduling with $q_i = 2^{i-1}$.



Multilevel Feedback Queue Scheduling

- Example: Feedback Scheduling with $q = 1$ and $q_i = 2^{i-1}$.



Thread Scheduling

- On operating systems that support User-level threads and Kernel-level threads, it is KLTs—not processes—that are being scheduled by the operating system. ULTs are managed by a thread library. To run on a CPU, ULTs must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP).
- Process-contention Scope
  - On systems implementing the many-to-one and many-to-many models, the thread library schedules ULTs to run on an available LWP.
  - This scheme is known as *process-contention scope* (PCS, 进程竞争域), since competition for the CPU takes place among threads in the same process.
    - When we say the thread library *schedules* user threads onto available LWPs, we do not mean that the threads are actually running on a CPU. That would require the operating system to schedule the kernel thread onto a physical CPU.

Thread Scheduling

- Process-contention Scope (cont.)
  - Typically, PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run. ULT priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread. It is important to note that PCS will typically preempt the thread currently running in favor of a higher-priority thread; however, there is no guarantee of time slicing among threads of equal priority.
- System-contention Scope
  - The kernel uses *system-contention scope* (SCS, 系统竞争域) to decide which KLT to schedule onto a CPU. Competition for the CPU with SCS scheduling takes place among all threads in the system.
    - Systems using the one-to-one model, such as Windows, Linux, and Solaris, schedule threads using only SCS.

Threads Scheduling

- **Thread Scheduling**
  - Pthreads Scheduling
    - Pthreads identifies the following contention scope values:

      ```
      PTHREAD_SCOPE_PROCESS
      ```

      - It schedules threads using PCS scheduling.

      ```
      PTHREAD_SCOPE_SYSTEM
      ```

      - It schedules threads using SCS scheduling.
    - The Pthread IPC provides two functions for getting and setting the contention scope policy:

      ```
      pthread_attr_setscope(pthread_attr t *attr, int scope)
      pthread_attr_getscope(pthread_attr t *attr, int *scope)
      ```
    - Next slide shows a Pthread scheduling API. It first determines the existing contention scope and sets it to PTHREAD_SCOPE_SYSTEM. It then creates five separate threads that will run using the SCS scheduling policy. Note that on some systems, only certain contention scope values are allowed. For example, Linux and Mac OS X systems allow only PTHREAD_SCOPE_SYSTEM.

Thread Scheduling

- Example: Pthreads Scheduling API
  - pthread_scope.c

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void *runner(void *);
int main(int argc, char *argv[])
{
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
      /* get the default attributes */
    pthread_attr_init(&attr);
      /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("\nscope = PTHREAD_SCOPE_PROCESS\n");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("\nscope = PTHREAD_SCOPE_SYSTEM\n");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
```

Thread Scheduling

- Example: Pthreads Scheduling API
  - pthread_scope.c (2)

```
          /* set the scheduling algorithm to PCS or SCS */
        pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
          /* create the threads */
        for (i = 0; i < NUM_THREADS; i++)
            pthread_create(&tid[i], &attr, &runner, NULL);
          /* now join on each thread */
        for (i = 0; i < NUM_THREADS; i++)
            pthread_join(tid[i], NULL);
    }

      /* Each thread will begin control in this function */
    void *runner(void *param)
    {
          /* do some work ... */
        pthread_exit(0);
    }
```