CPU Scheduling

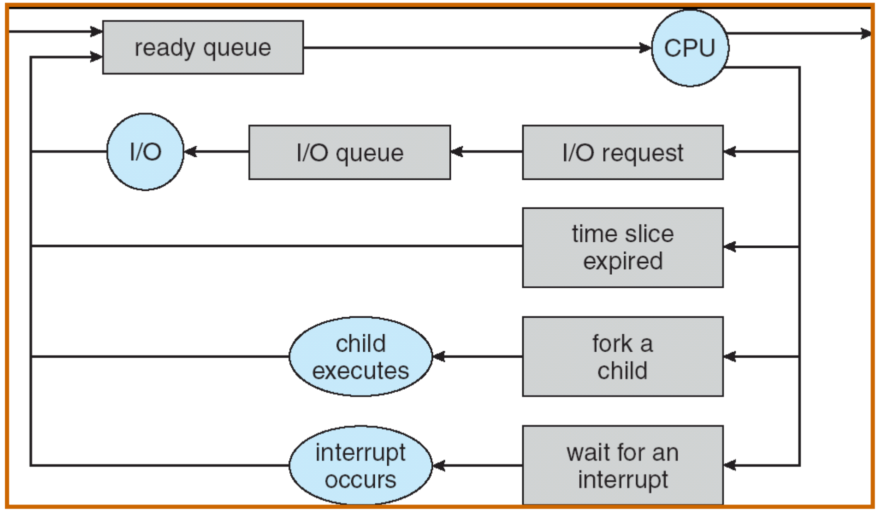
Basic Concepts

Scheduling

- Long-Term Scheduling
  - Determines which programs are admitted to the system for processing
  - Controls the degree of multiprogramming
- Medium-Term Scheduling
  - Part of the swapping function
  - Based on the need to manage the degree of multiprogramming
- Short-Term Scheduling
  - Known as the dispatcher
  - Executes most frequently

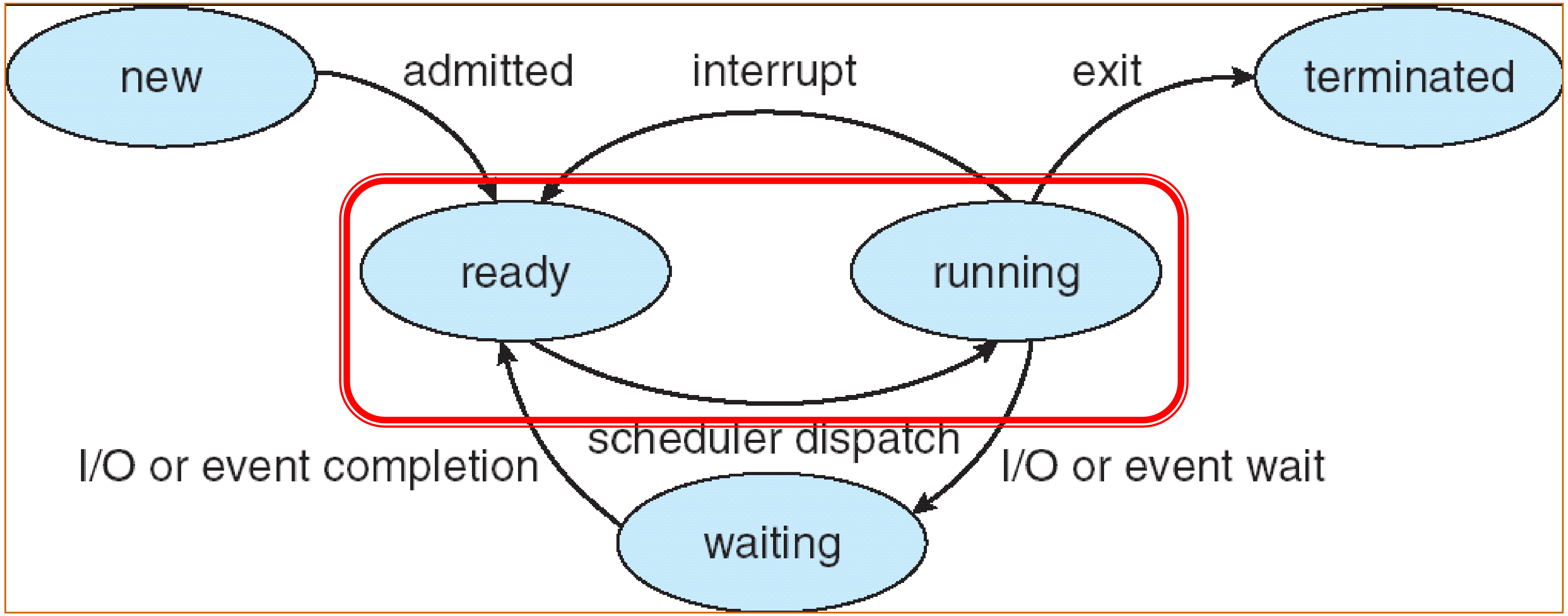
CPU Scheduling

- Earlier, we talked about the life-cycle of a process



- Question: How is OS to decide which of several tasks to take off a queue?
  - Obvious queue to worry about is ready queue
  - Others can be scheduled as well, however

Diagram of Process State

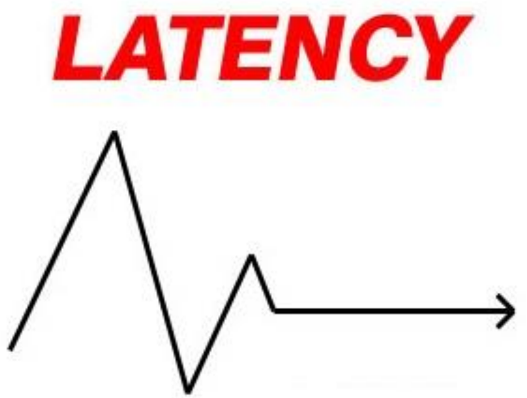


CPU Scheduler

- Selects from the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  - 1. Switches from running to waiting state
  - 2. Switches from running to ready state
  - 3. Switches from waiting to ready
  - 4. Terminates
- Scheduling under 1 and 4 (no choice in terms of scheduling) is nonpreemptive
- All other scheduling is preemptive

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running



Scheduling Assumptions

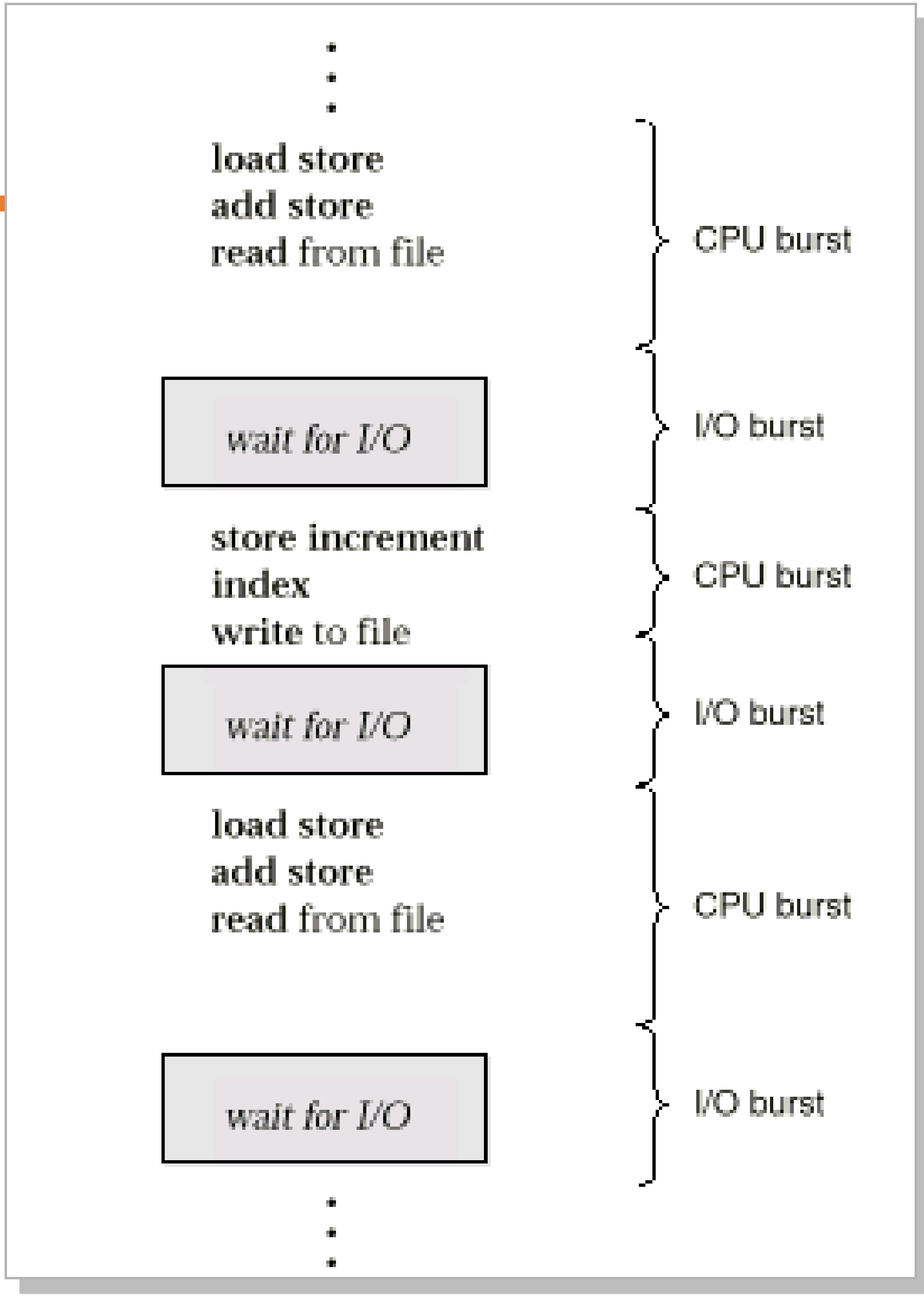
- CPU scheduling is a big area of research in early 70's
- Many implicit assumptions for CPU scheduling:
  - One program per user
  - One thread per program
  - Programs are independent

Scheduling Assumptions

- Clearly, these are unrealistic but they simplify the problem so it can be solved
  - For instance: is “fair” about fairness among users or programs?
    - If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system

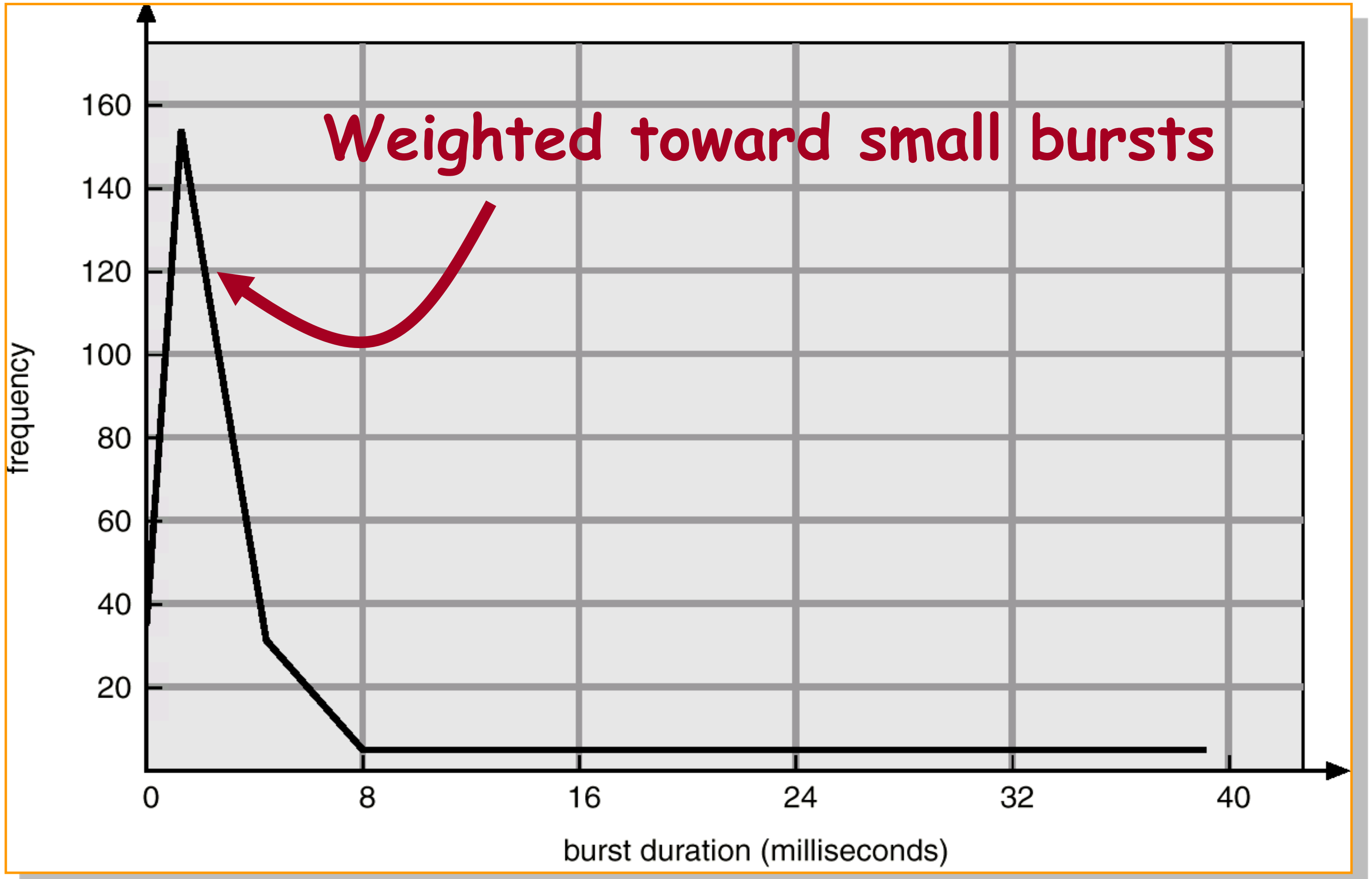
Basic Concepts

- **CPU-I/O burst cycle**: Process execution consists of a cycle of CPU execution and I/O wait (i.e., **CPU burst** and **I/O burst**).
- Execution model: programs alternate between bursts of CPU and I/O



CPU Burst Distribution

- Histogram of CPU-burst Times (exponential or hyper-exponential)



Scheduling Criteria

Scheduling Policy Goals/Criteria

- Minimize Response Time
- Maximize Throughput
- Fairness

Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Response time (of a request)
    - submission ~ the first response is produced
  - Response time is what the user sees:
    - Time to echo a keystroke in editor
- Maximize Throughput
- Fairness

Scheduling Policy Goals/Criteria

- Minimize Response Time
- Maximize Throughput
  - Throughput
    - number of completed processes per time unit
  - Maximize operations (or jobs) per second
- Fairness

Scheduling Policy Goals/Criteria

- Minimize Response Time
- Maximize Throughput
- Fairness
  - Share CPU among users in some equitable way

Scheduling Policy Goals/Criteria

- CPU utilization
  - theoretically: 0%~100%
  - real systems: 40% (light)~90% (heavy)
- Turnaround time (of a process)
  - submission ~ completion
- Waiting time (of a process)
  - total waiting time in the ready queue

Scheduling Algorithms

Scheduling Algorithms

- First-Come-First-Served (FCFS) Scheduling
- Round-Robin (RR) Scheduling

- Shortest-Job-First (SJF) Scheduling
- Shortest-Remaining-Time-First (SRTF) Scheduling

- Priority Scheduling

- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

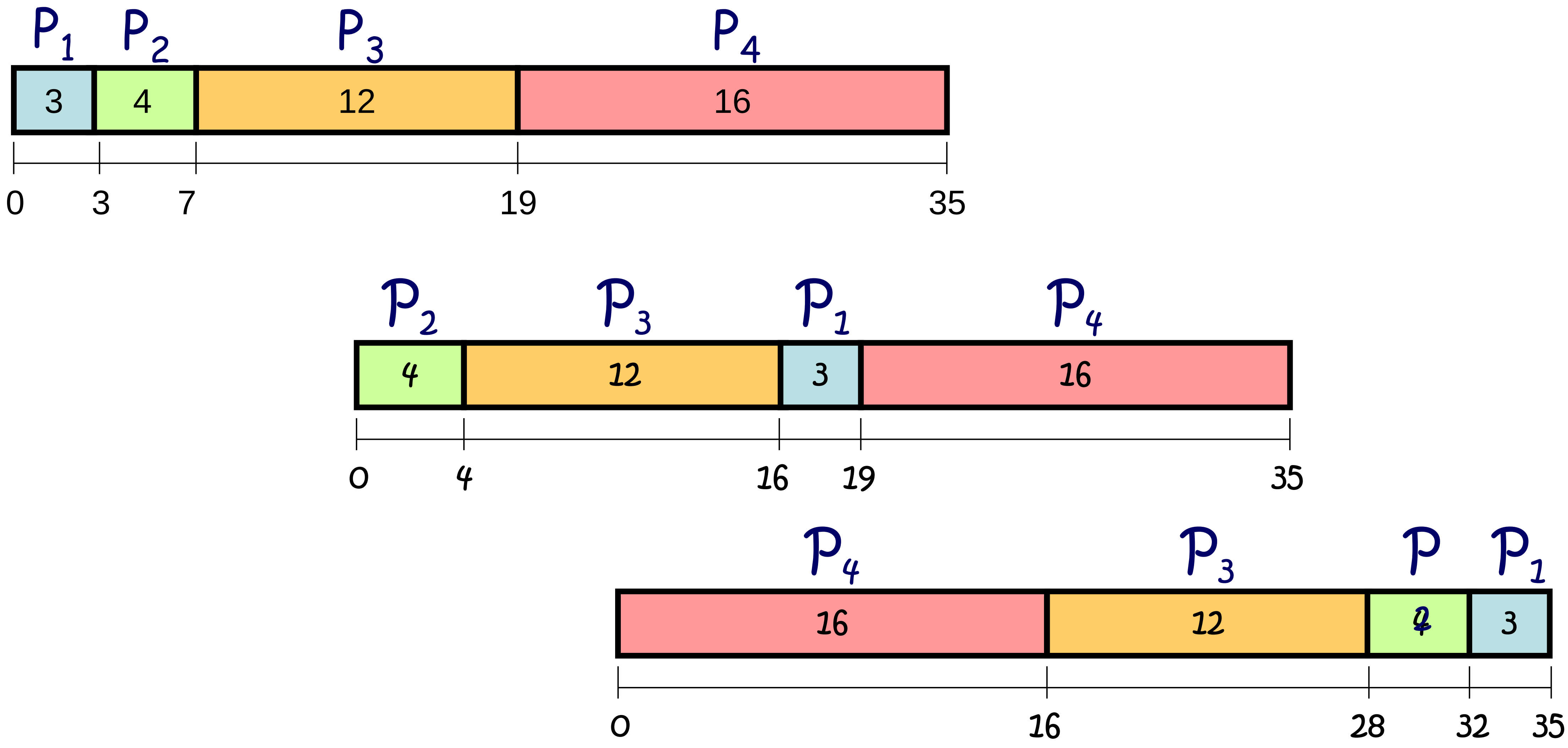
First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also “First In, First Out” (FIFO) or “Run until done”
    - In early systems, FCFS meant one program scheduled until done (including I/O)
    - Now, means keep CPU until process blocks

FCFS Scheduling

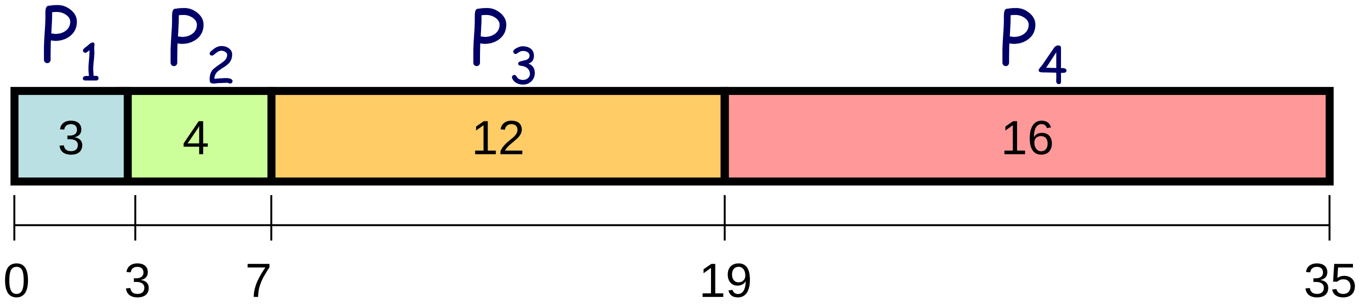
- Each process joins the Ready queue
- When the current process ceases to execute, the **oldest** process in the ready queue is selected

FCFS Scheduling



FCFS Scheduling

- Example: 4 processes arrived almost simultaneously



- Waiting time:

  P_1=0; P_2=3; P_3=7; P_4=19

- Average waiting time:

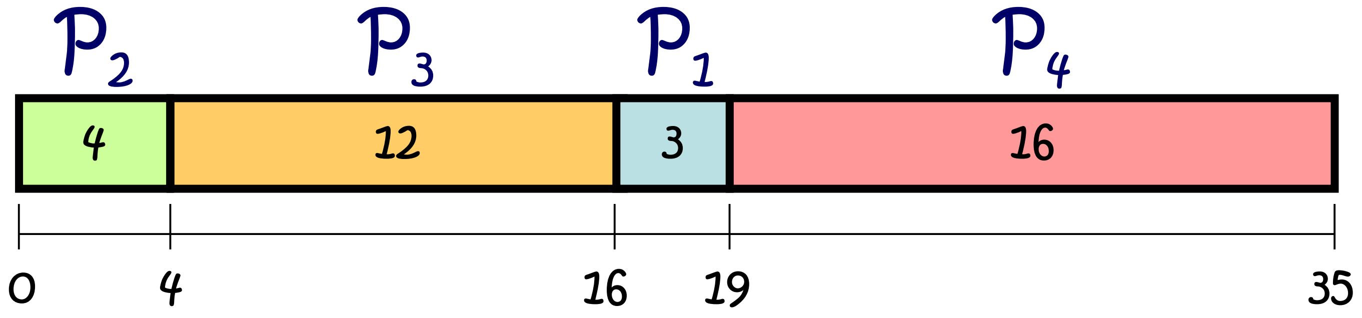
  (0+3+7+19)/4 =7.25

- Average completion time:

  (3+7+19+35)/4=16

FCFS Scheduling

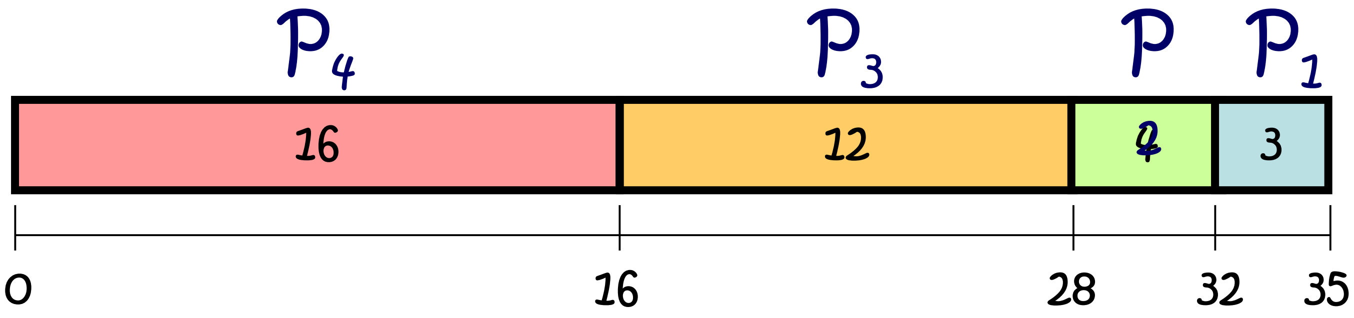
- Example: 4 processes arrived almost simultaneously



- Waiting time: P_2=0; P_3=4; P_1=16; P_4=19
- Average waiting time: (0+4+16+19) / 4 = 9.75
- Average completion time:(4+16+19+35) / 4 = 18.5

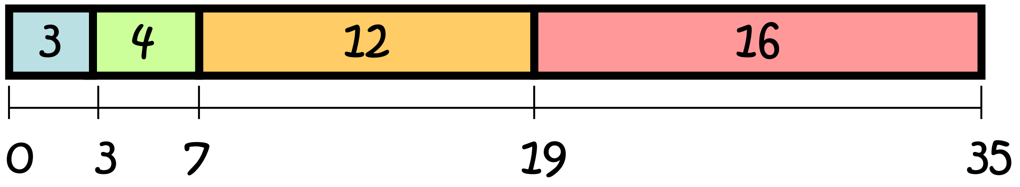
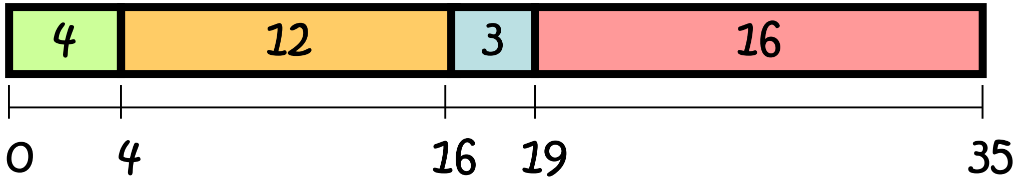
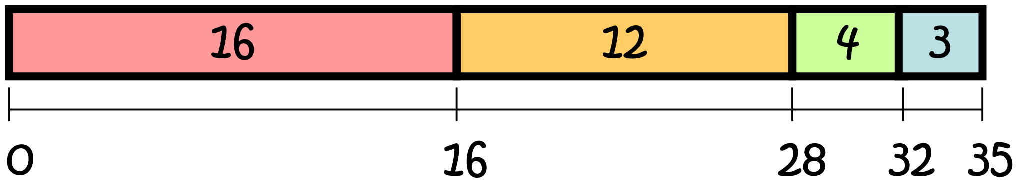
FCFS Scheduling

- Example: 4 processes arrived almost simultaneously



- Waiting time: P_4=0; P_3=16; P_2=28; P_1=32
- Average waiting time: (0+16+28+32) / 4 = 19
- Average completion time: (16+28+32+35) / 4 = 27.75

FCFS Scheduling

Scheduling Example	Average waiting time	Average completion time
	7.25	16
	9.75	18.5
	19	27.75

FCFS Scheduling

- Simplest(+), non-preemptive, often having a long average waiting time(-)
- A short process may have to wait a very long time before it can execute(-)
- Favors CPU-bound processes
  - I/O processes have to wait until CPU-bound process completes

FCFS Scheduling

- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand…

Round Robin (RR)

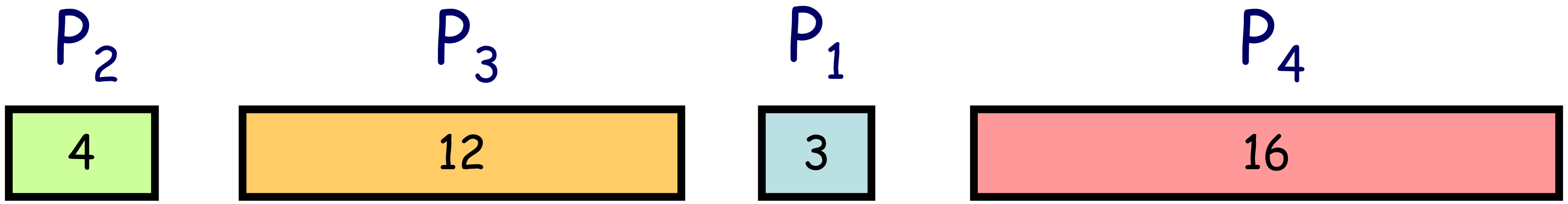
- Round Robin Scheme
- The modern use of the term dates from the 17th Century French ruban rond (round ribbon)

Round Robin (RR)

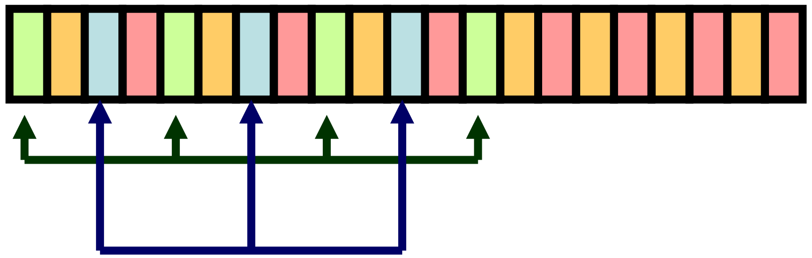
- Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds
- After quantum expires, the process is **preempted** and **added to the end of the ready queue**.
- n processes in ready queue, time quantum is $q \Rightarrow$
  - Each process gets $1/n$ of the CPU time
  - No process waits more than $(n\text{-}1)q$ time units

Example of RR with Time Quantum = 1

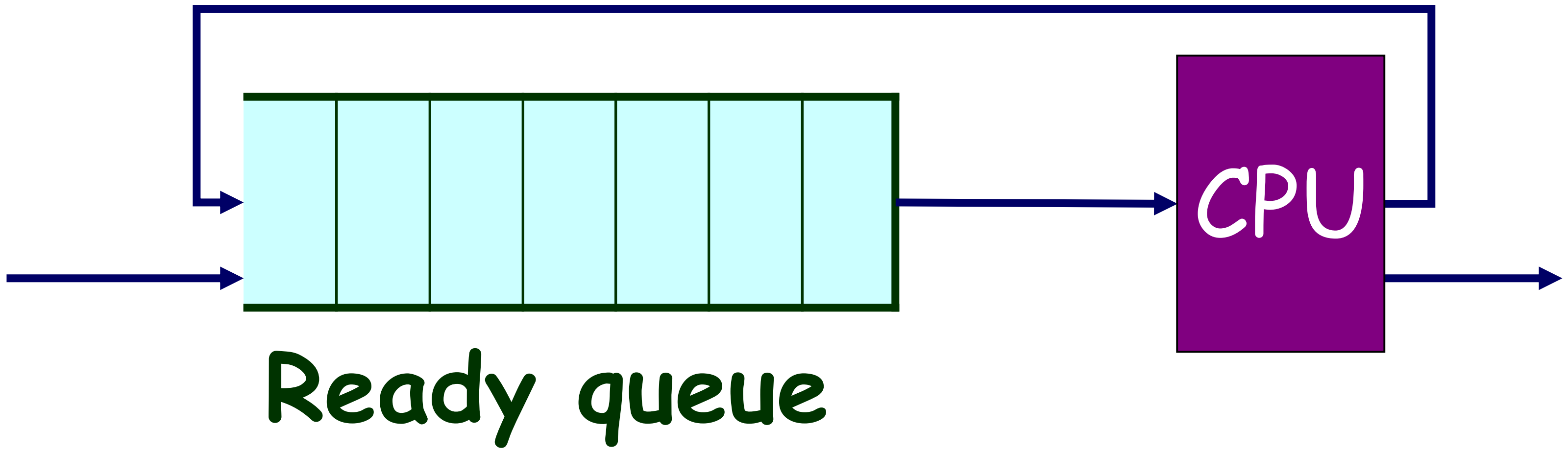
- 4 processes arrived almost simultaneously



- The Gantt chart is:

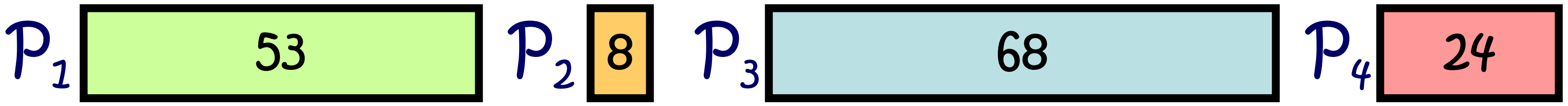


Round Robin (RR)

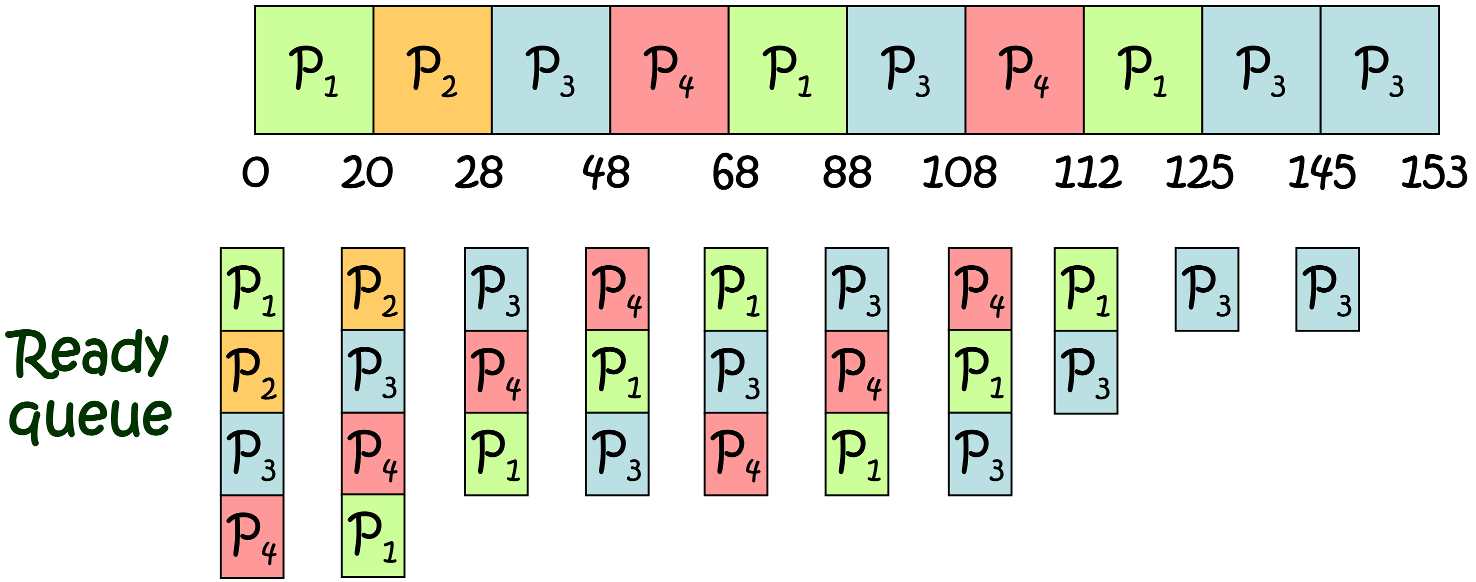


Example of RR with Time Quantum = 20

- Example: 4 processes arrived almost simultaneously



- The Gantt chart is:



Example of RR with Time Quantum = 20

- Example: 4 processes arrived almost simultaneously

P_1	P_2	P_3	P_4
53	8	68	24

- The Gantt chart is:

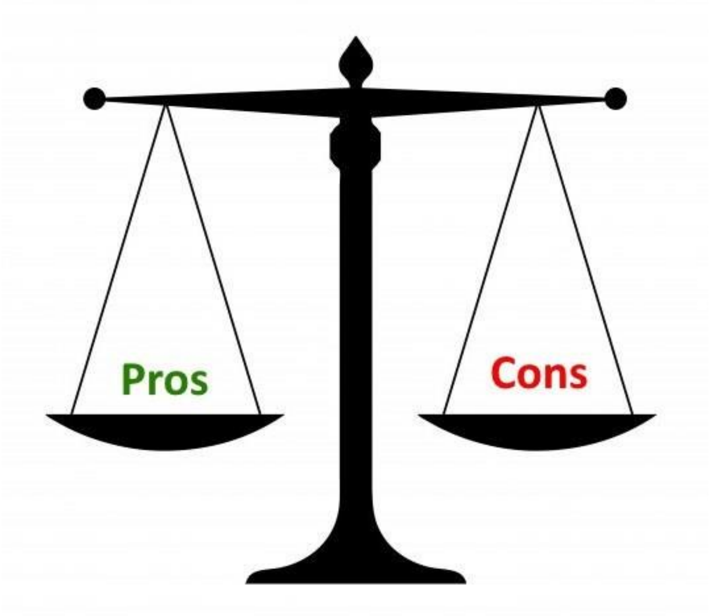
P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 28 48 68 88 108 112 125 145 153

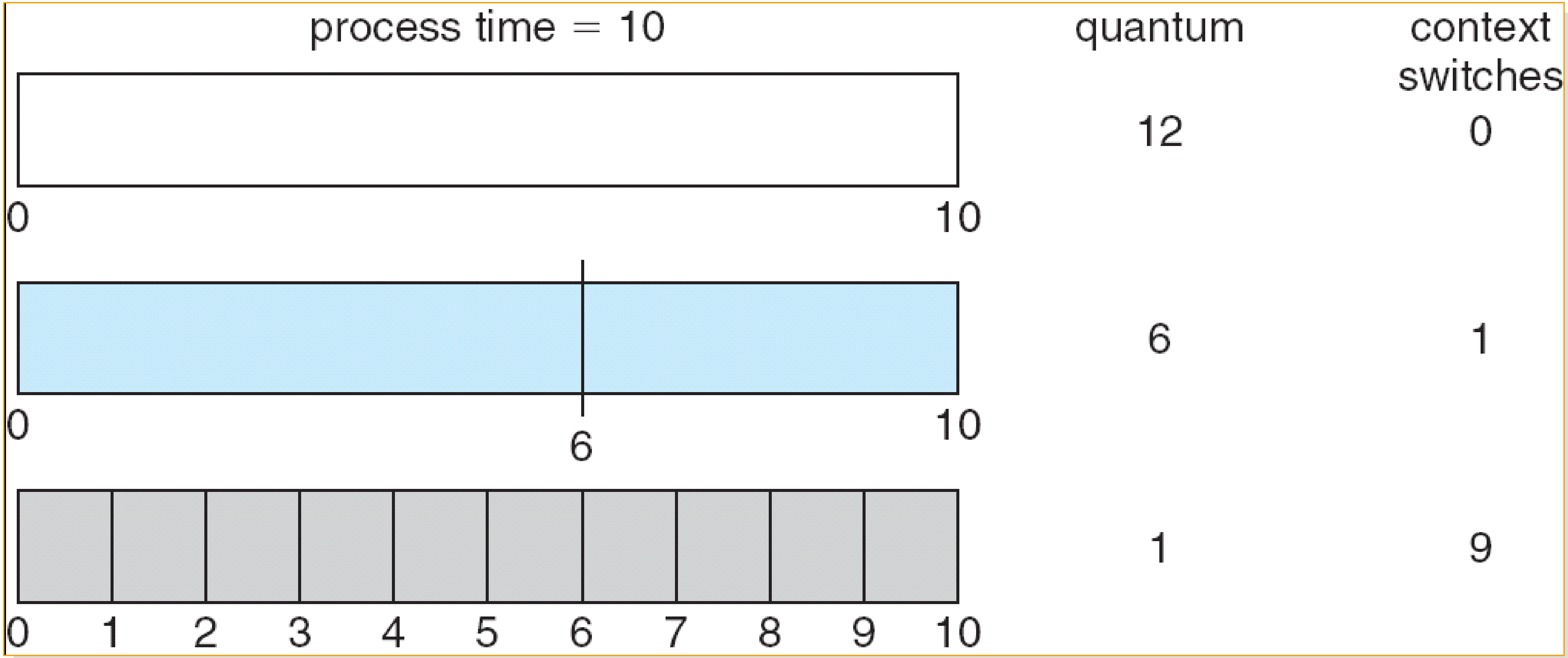
- **Waiting time**
  - P_1=(68-20)+(112-88)=72; P_2=(20-0)=20; P_3=(28-0)+(88-48)+(125-108)=85; P_4=(48-0)+(108-68)=88
- **Average waiting time** = (72+20+85+88)/4 = 66.25
- **Average completion time** = (125+28+153+112)/4 = 104.5

Round-Robin Pros and Cons

- Better for short jobs, Fair (+)
- Context-switching time adds up for long jobs (-)

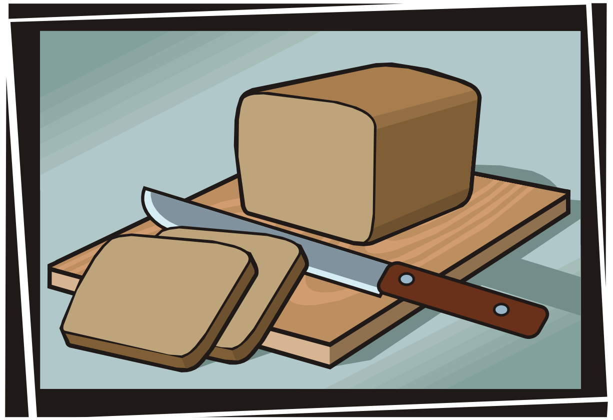


Time Quantum & Context Switch Time

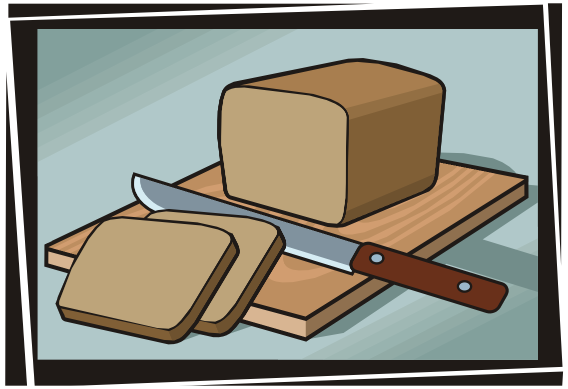


Round-Robin Discussion

- How do you choose **time slice**?
  - What if too big?
    - Response time suffers
  - What if infinite (∞)?
    - Get back FCFS
  - What if time slice too small?
    - Throughput suffers!
  - Must be large with respect to context switch, otherwise overhead is too high (all overhead)



Round-Robin Discussion



- Actual choices of time slice:
  - Initially, the UNIX time slice is 1s:
    - Worked ok when UNIX was used by one or two people.
    - What if three compilations going on? 3 seconds to echo each keystroke!
  - In practice, need to balance short-job performance and long-job throughput:
    - Typical time slice today is between 10ms – 100ms
    - Typical context-switching overhead is 0.1ms – 1ms
    - Roughly 1% overhead due to context-switching

Comparisons between FCFS and RR

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example:
  - 10 jobs, each take *100s* of CPU time
  - RR scheduler quantum of *1s*
  - All jobs start at the same time

Job #	FCFS	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

- Completion Time:
  - Both RR and FCFS finish at the same time
  - Average completion time is much worse under RR!
    - Bad when all jobs same length

Example with Different Time Quantum

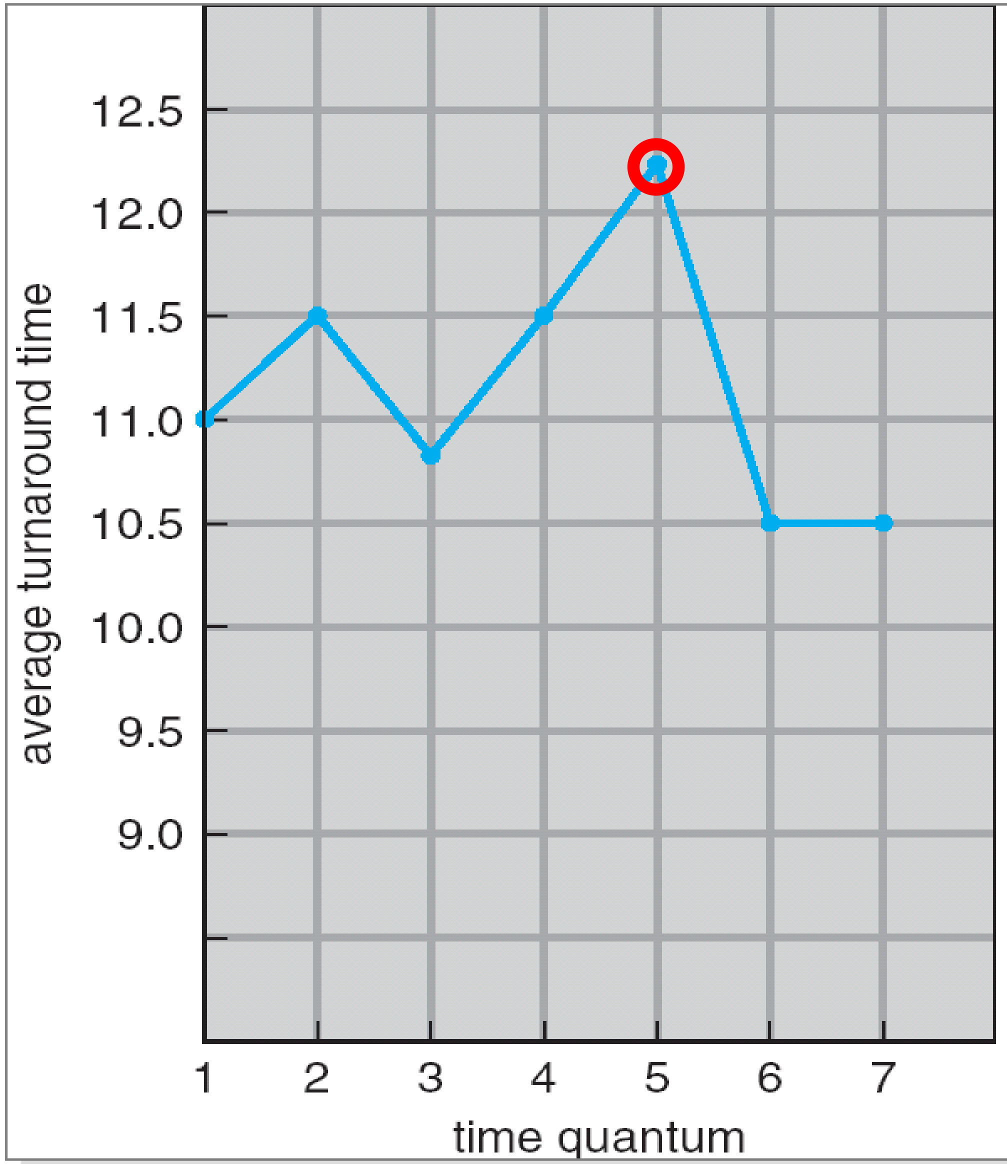
Best FCFS:

P_2 [8]	P_4 [24]	P_1 [53]	P_3 [68]

0 8 32 85 153

	Quantum	P_1	P_2	P_3	P_4	Average
Waiting Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

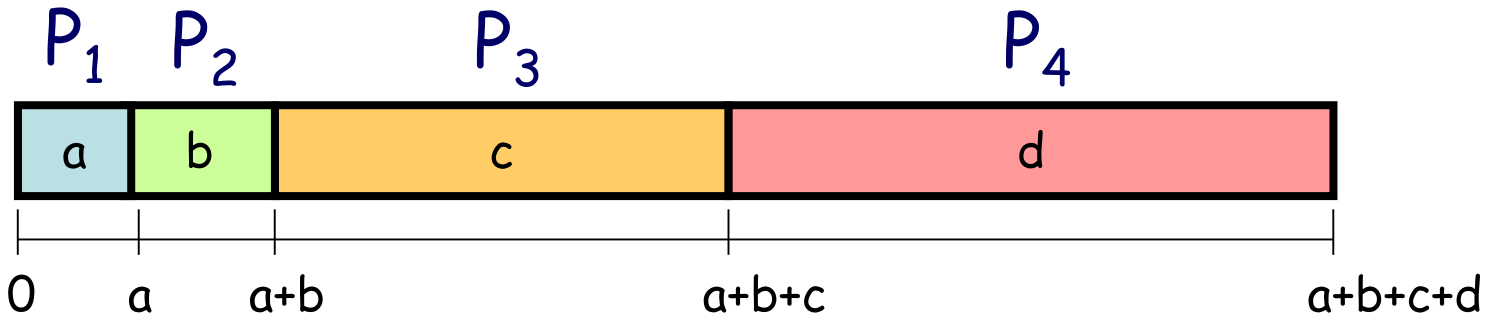
5 3 1 5 1 2

15+
8+
9+
17 = 49

49/4 = 12.25

Discussion

- Average waiting time = (3a+2b+c)/4
- Average completion time = (4a+3b+2c+d)/4



What if we knew the future?

- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  - Idea is to get short jobs out of the system
  - Run whatever job has the **least amount of computation** to do
  - Sometimes called “Shortest Time to Completion First” (STCF)
- Associate with each process the length of its next CPU burst.

What if we know the future?

- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF
    - if a job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  - Sometimes called "Shortest Remaining Time to Completion First" (SRTCF)

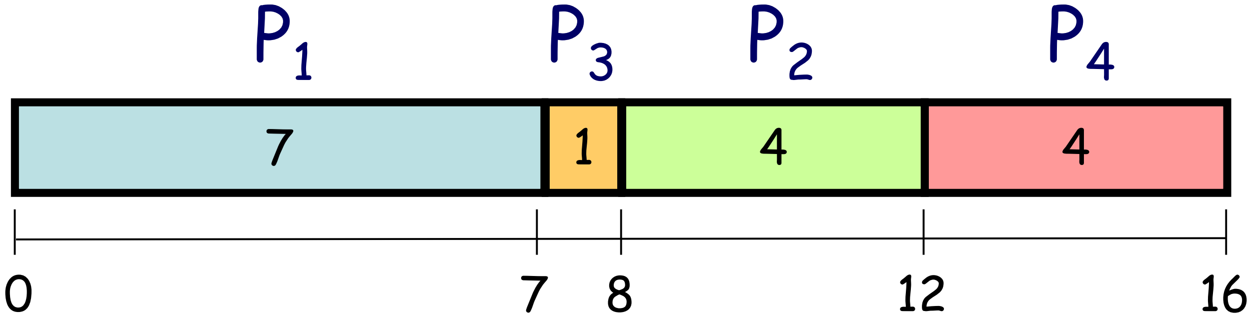
Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
~~P_1~~	~~0.0~~	~~7~~
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)



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

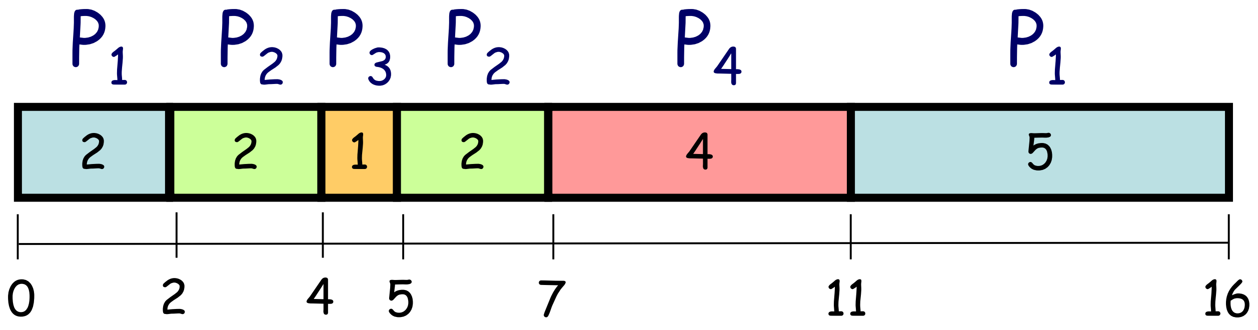
Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	~~7~~ 5
P_2	2.0	~~4~~ ~~2~~
P_3	4.0	~~1~~
P_4	5.0	~~4~~

- SRTF (preemptive)



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

Discussion

- SJF/SRTF are the **best** you can do at minimizing average waiting time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, focus on SRTF

Discussion

- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    - SRTF (and RR): short jobs not stuck behind long ones

SRTF Further discussion

- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Short-running I/O bound jobs stay near top
  - CPU bound jobs drop like a rock
  - Large jobs may never get to run

SRTF Further discussion

- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal, so can't do any better
- SRTF Pros & Cons
  - Optimal (average waiting time) (+)
  - Hard to predict future (-)
  - Unfair (-)

SRTF Further discussion

- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    - When you submit a job, have to say how long it will take
    - To stop cheating, system kills job if takes too long
  - But: users have trouble predicting runtime of their jobs

SRTF Further discussion

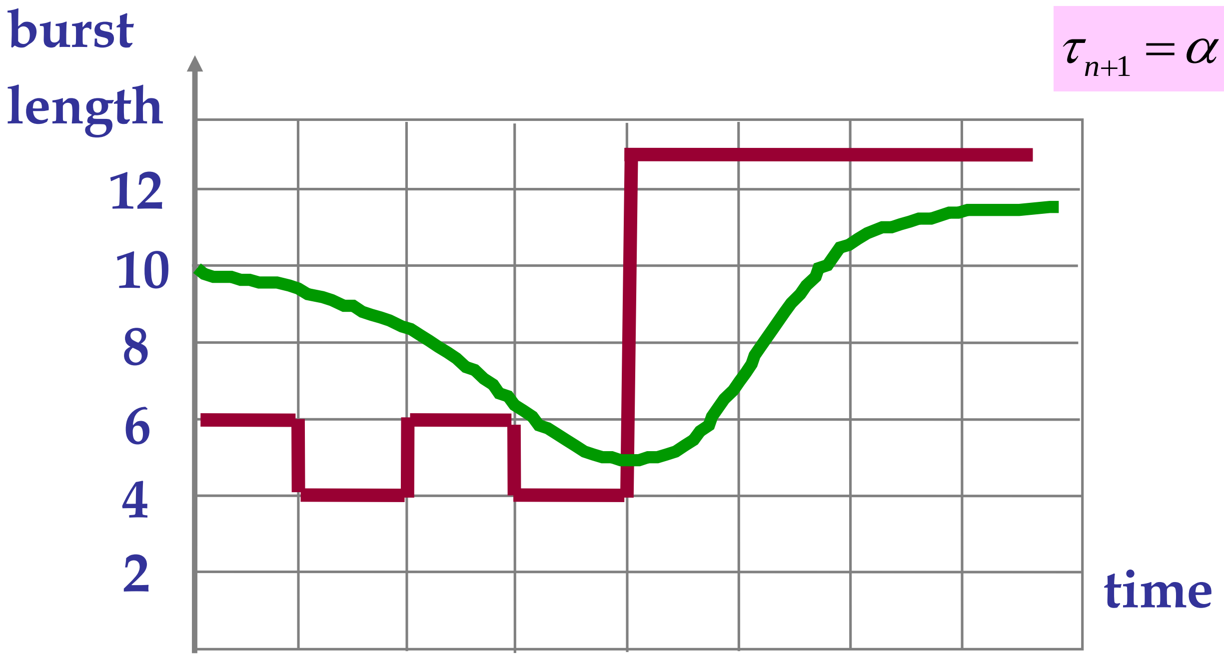
- **Difficulty:** no way to know length of the next CPU burst. Thus, it cannot be implemented in short-term scheduler
- **Approximate SJF**: the next burst can be predicted as an exponential average of the measured length of previous CPU bursts

Determining Length of Next CPU Burst

- Can only **estimate** the length
- Using the length of previous CPU bursts by exponential averaging

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

Prediction of the Length of Next CPU Burst



$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$

CPU burst t_n	t_0=	6	4	6	4	13	13	13	…
guess τ_n	τ_0=	10	8	6	6	5	9	11	…
Diff.		4	4	0	2	-8	-4	-2	…

Examples of Exponential Averaging

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n .$$

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts

Examples of Exponential Averaging

- If we expand the formula, we get:

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

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Scheduling algorithms

- First-Come-First-Served (FCFS) Scheduling
- Round-Robin (RR) Scheduling

- Shortest-Job-First (SJF) Scheduling
- Shortest-Remaining-Time-First (SRTF) Scheduling

- Priority Scheduling

- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

Priority scheduling

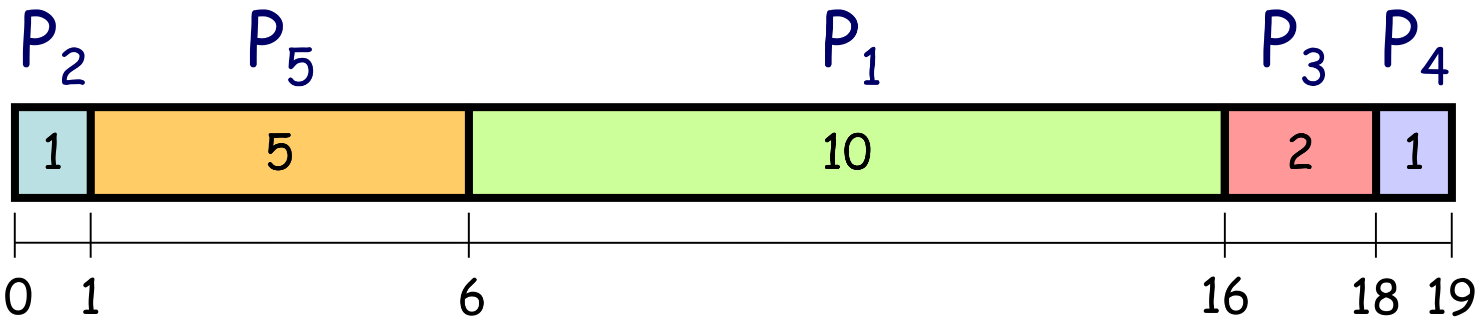
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
  - Preemptive
  - Non-preemptive
- SJF?
  - is a priority scheduling where priority is the predicted next CPU burst time (smallest integer ≡ highest priority)

An Example

Example:

5 processes arrived almost simultaneously

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



Average waiting time = 8.2

An Example

Example:

5 processes arrived almost simultaneously

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Priority Scheduling

- Problem: **Starvation** – low priority processes may never execute
  - At MIT, there was a job submitted in 1967 that had not be run in 1973.
- Solution: **Aging** – as time progresses increase the priority of the process

Static priority scheduling

- Static priority scheduling
  - Prioritizing a process when it is created and keeping the priority unchanged until the process is finished.
- How to prioritize?
  - The type of process
    - System process › user process, interactive process › batch process
  - The need for system resources
    - Higher priority for processes with less CPU and memory requirements
  - User requirements
    - User level & payment, e.g. military computers, commercial computers

Dynamic priority scheduling

- Dynamic priority scheduling
  - The priority given to the process when it is created can be dynamically changed during process operation for better scheduling performance
- To prevent “starvation”, priority is adjusted based on running time and waiting time
  - A process lowers its priority every time it executes.
  - In the ready queue, the priority of the process is gradually increased as the waiting time increases.

Highest response ratio next

- **Highest Response Ratio Next (HRRN)** scheduling
  - a non-preemptive discipline, similar to SJF
  - the priority of each job is dependent on its estimated run time, and also the amount of time it has spent waiting.

$$Priority = \frac{estimated\ run\ time + waiting\ time}{waiting\ time}$$

$$= 1 + \frac{estimated\ run\ time}{waiting\ time}$$

Highest response ratio next

- Jobs gain higher priority the longer they wait, which prevents indefinite postponement (process starvation).
- In fact, the jobs that have spent a long time waiting compete against those which are estimated to have a short running time.

$$Priority = \frac{estimated\ run\ time + waiting\ time}{waiting\ time}$$

$$= 1 + \frac{estimated\ run\ time}{waiting\ time}$$

Example 1

JOB	ARRIVAL TIME	RUNNING TIME
0	10：00	24 minutes
1	10：10	60 minutes
2	10：15	36 minutes
3	10：20	12 minutes

- Assume that all jobs are completely CPU bound.
- For each of the scheduling algorithms, determine the scheduling order and the average turnaround time
  - FCFS
  - SJF

Example 1

JOB	ARRIVAL TIME	RUNNING TIME
0	10：00	24 minutes
1	10：10	60 minutes
2	10：15	36 minutes
3	10：20	12 minutes

- FCFS
  - (24+74+105+112)/4=78.75 min
- SJF
  - (24+122+57+16)/4=54.75 min

Example 2

- 5 processes arrived almost simultaneously

P1 P2 P3 P4 P5

- For each of the scheduling algorithms, determine the scheduling order and the average turnaround time
  - FCFS
  - RR (time slice = 1)
  - SJF
  - Priority Scheduling

Process	Running Time	Priority
P1	8	4
P2	4	3
P3	6	2
P4	2	1
P5	10	5 (highest)

Example 2

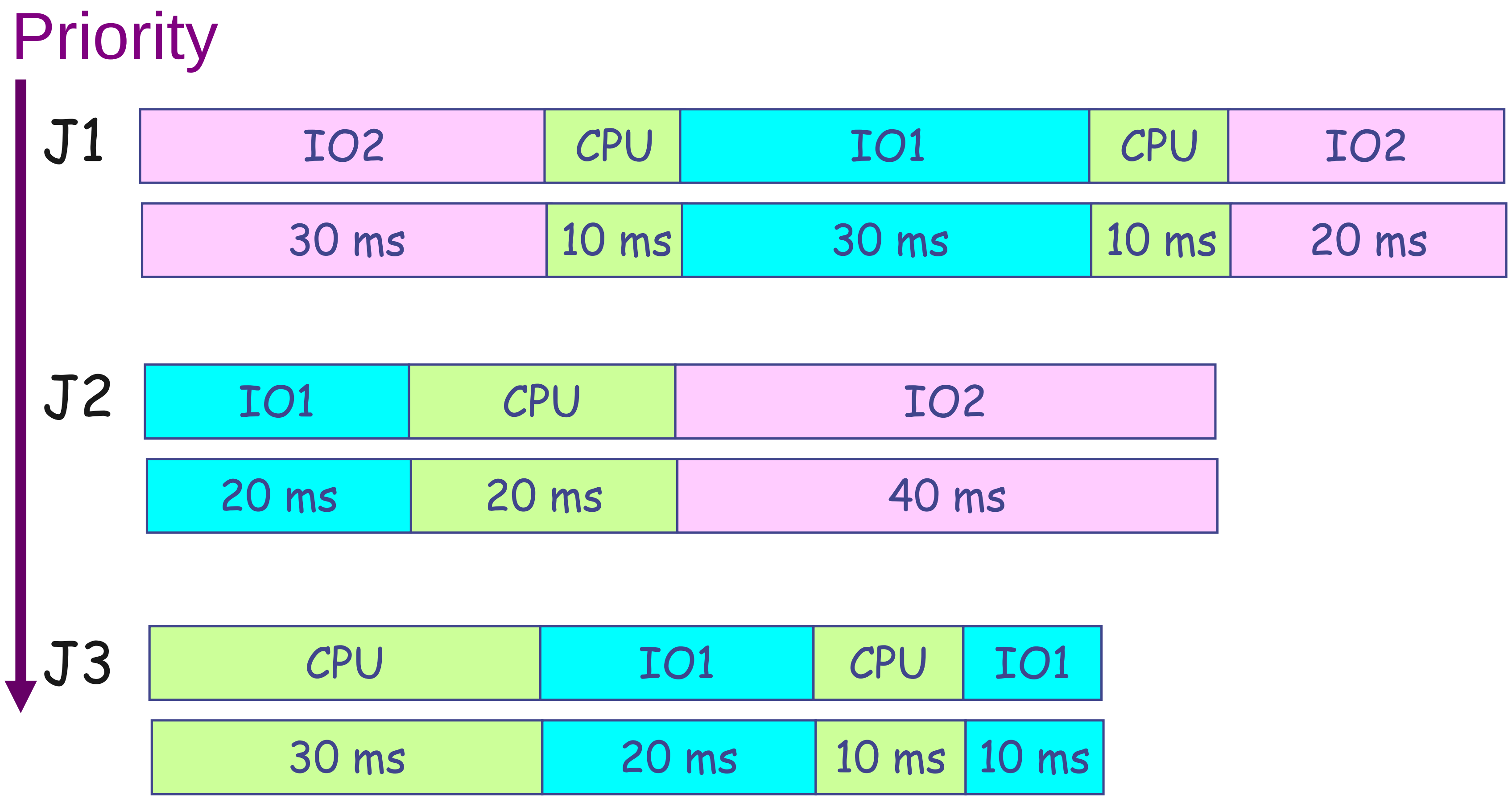
- Average turnaround time
- FCFS
  - 17.6
- RR (time slice = 1)
  - 21
- SJF
  - 14
- Priority Scheduling
  - 21.6

Process	Running Time	Priority
P1	8	4
P2	4	3
P3	6	2
P4	2	1
P5	10	5 (highest)

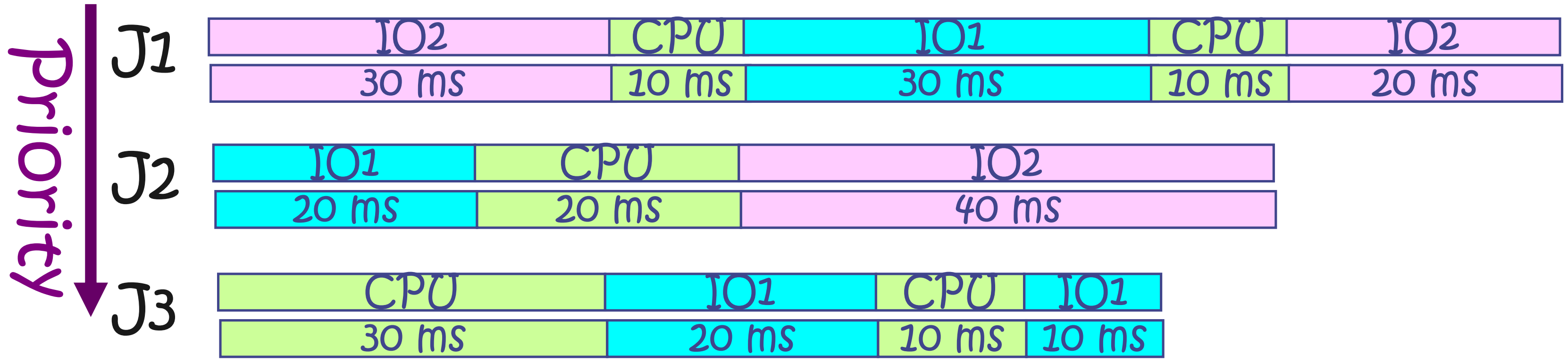
Example 3

- In a multi programming system it has 1 processor and 2 IO devices. 3 jobs come at once, called J1、J2 and J3. The running time queue they use as follows.
  - J1: IO2(30ms); CPU(10ms); IO1(30ms); CPU(10ms); IO2(20ms)
  - J2: IO1(20ms); CPU(20ms); IO2(40ms)
  - J3: CPU(30ms); IO1(20ms); CPU(10ms); IO1(10ms)
- Assume that CPU、IO1 and IO2 work parallel, J1 has highest priority, J2 has second one, J3 has third. The higher priority job can preempt the CPU, but can not preempt IO.

Example 3


Priority
J1
IO2
CPU
IO1
CPU
IO2
30 ms
10 ms
30 ms
10 ms
20 ms
J2
IO1
CPU
IO2
20 ms
20 ms
40 ms
J3
CPU
IO1
CPU
IO1
30 ms
20 ms
10 ms
10 ms

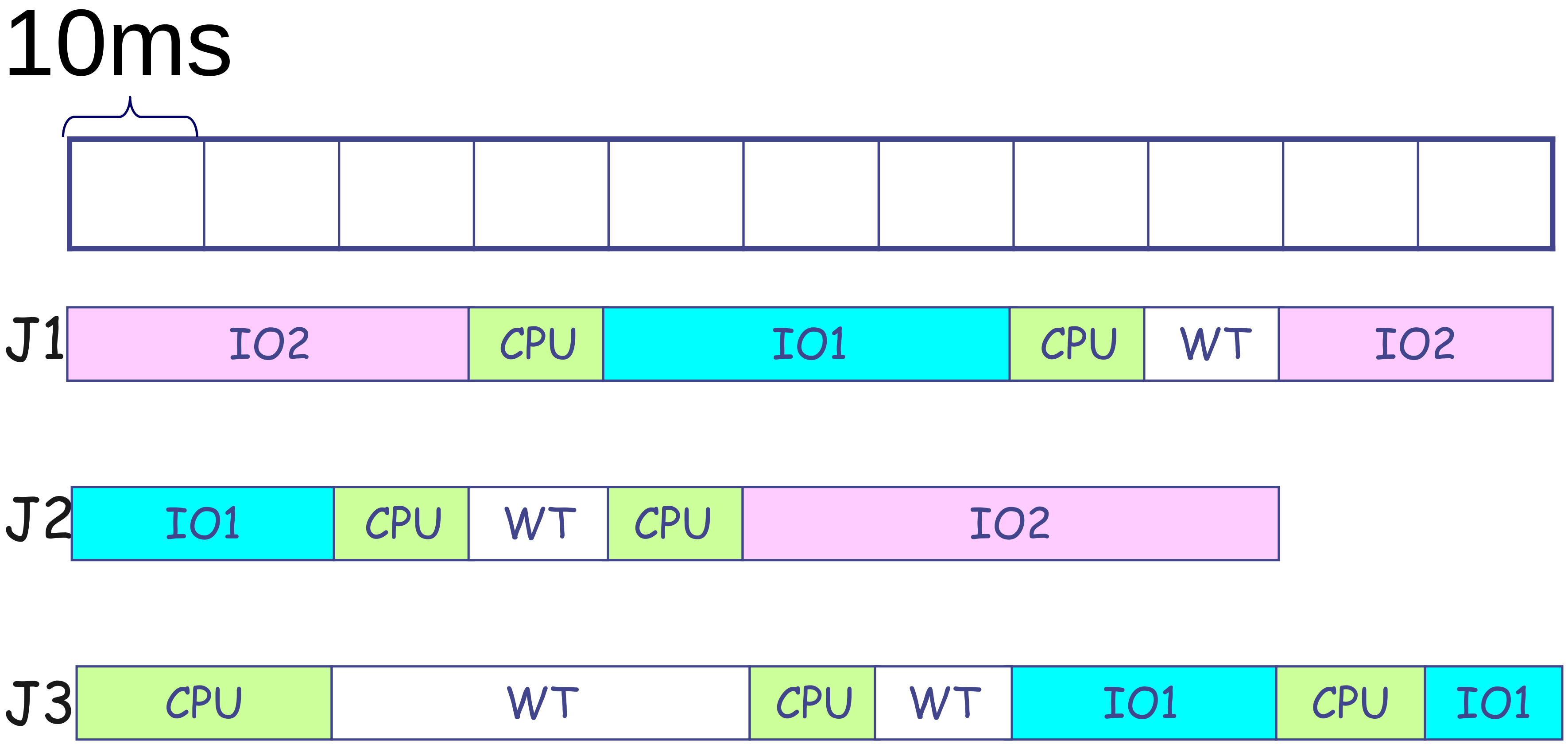
Example3



- 10ms

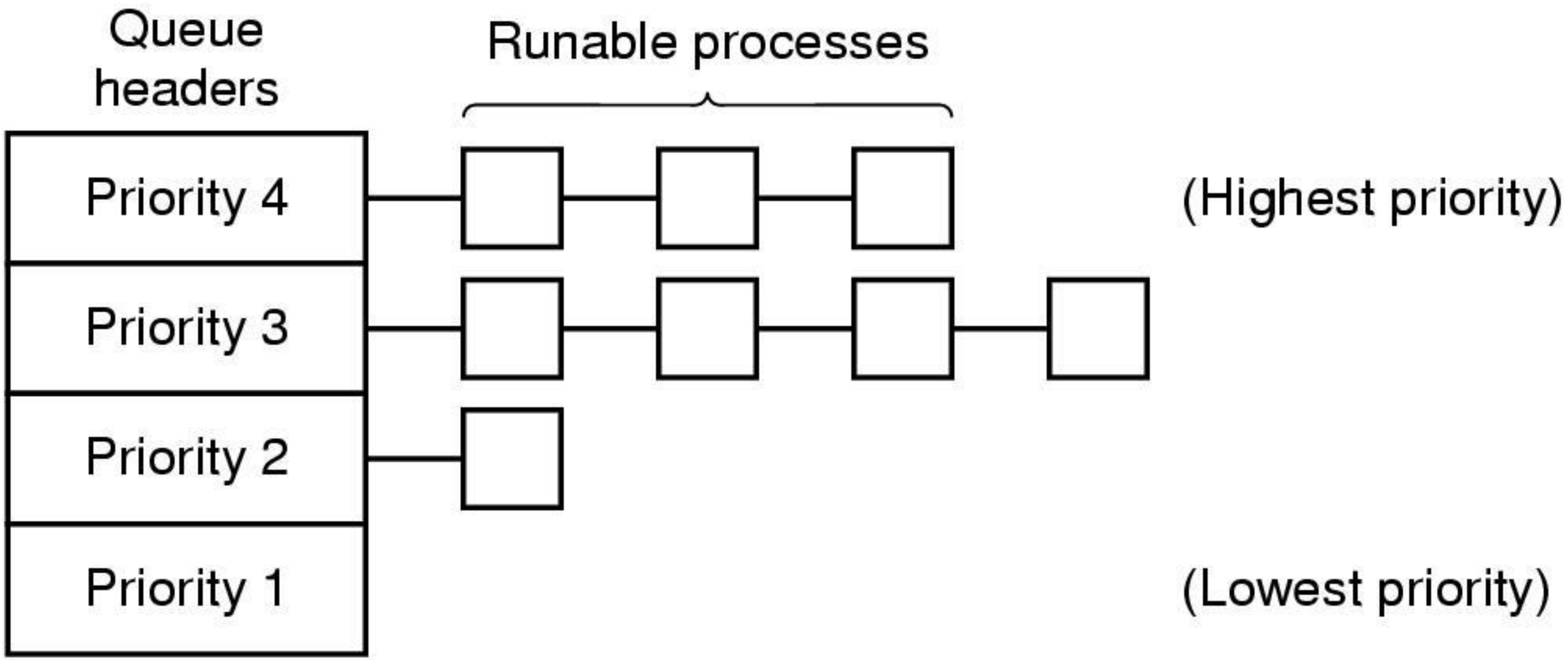
J1	IO2	IO2	IO2	CPU	IO1	IO1	IO1	CPU	wait	IO2	IO2
J2	IO1	IO1	CPU	wait	CPU	IO2	IO2	IO2	IO2		
J3	CPU	CPU	wait	wait	wait	CPU	wait	IO1	IO1	CPU	IO1

Example 3

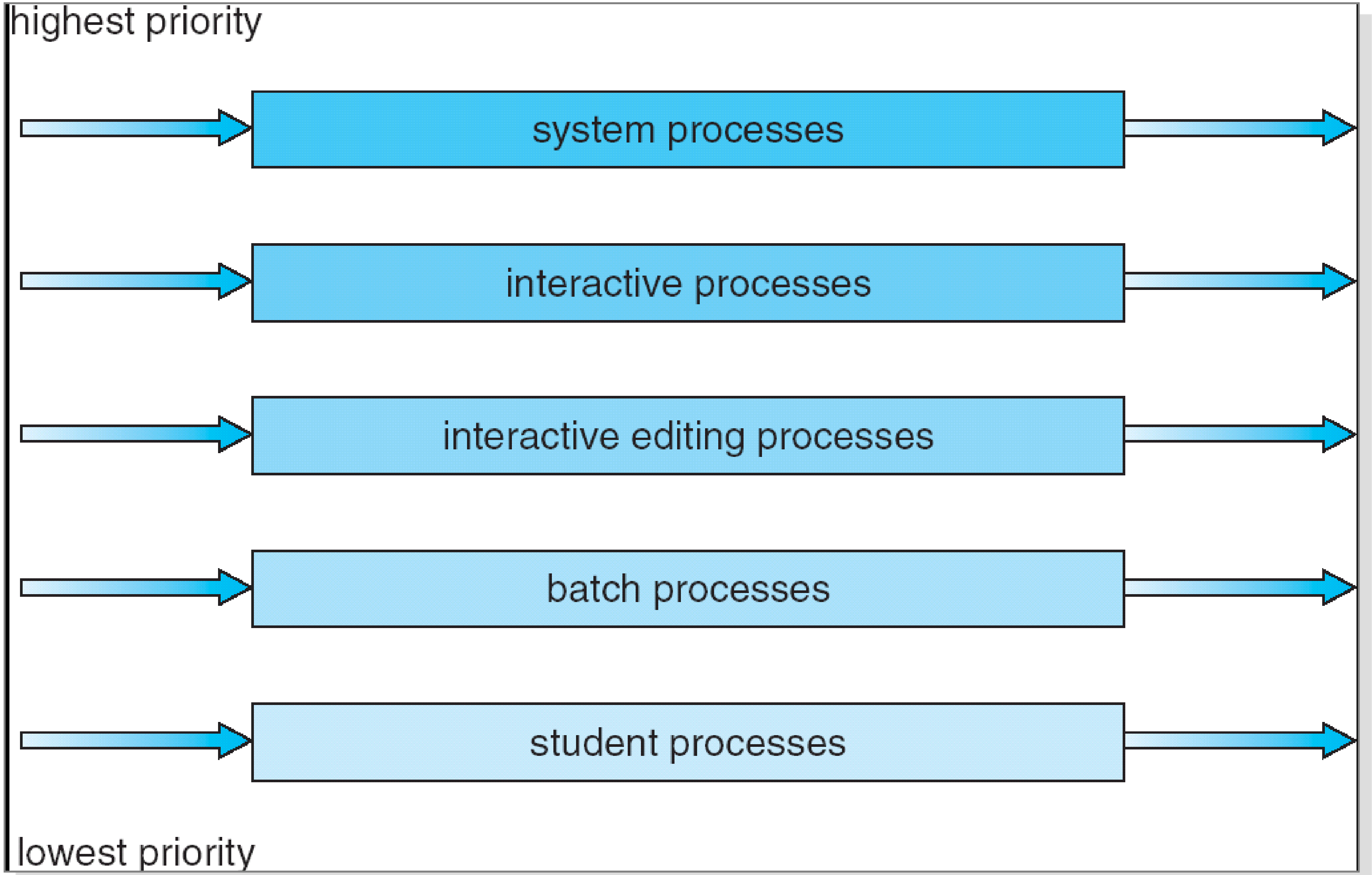


10ms
J1
IO2
CPU
IO1
CPU
WT
IO2
J2
IO1
CPU
WT
CPU
IO2
J3
CPU
WT
CPU
WT
IO1
CPU
IO1

Multilevel Queue Scheduling



Multilevel Queue Scheduling


highest priority
system processes
interactive processes
interactive editing processes
batch processes
student processes
lowest priority

Multilevel Queue Scheduling

- Example:
  - Ready queue is partitioned into separate queues: foreground (interactive) / background (batch)
  - Each queue has its own scheduling algorithm
    - foreground – RR
    - background – FCFS

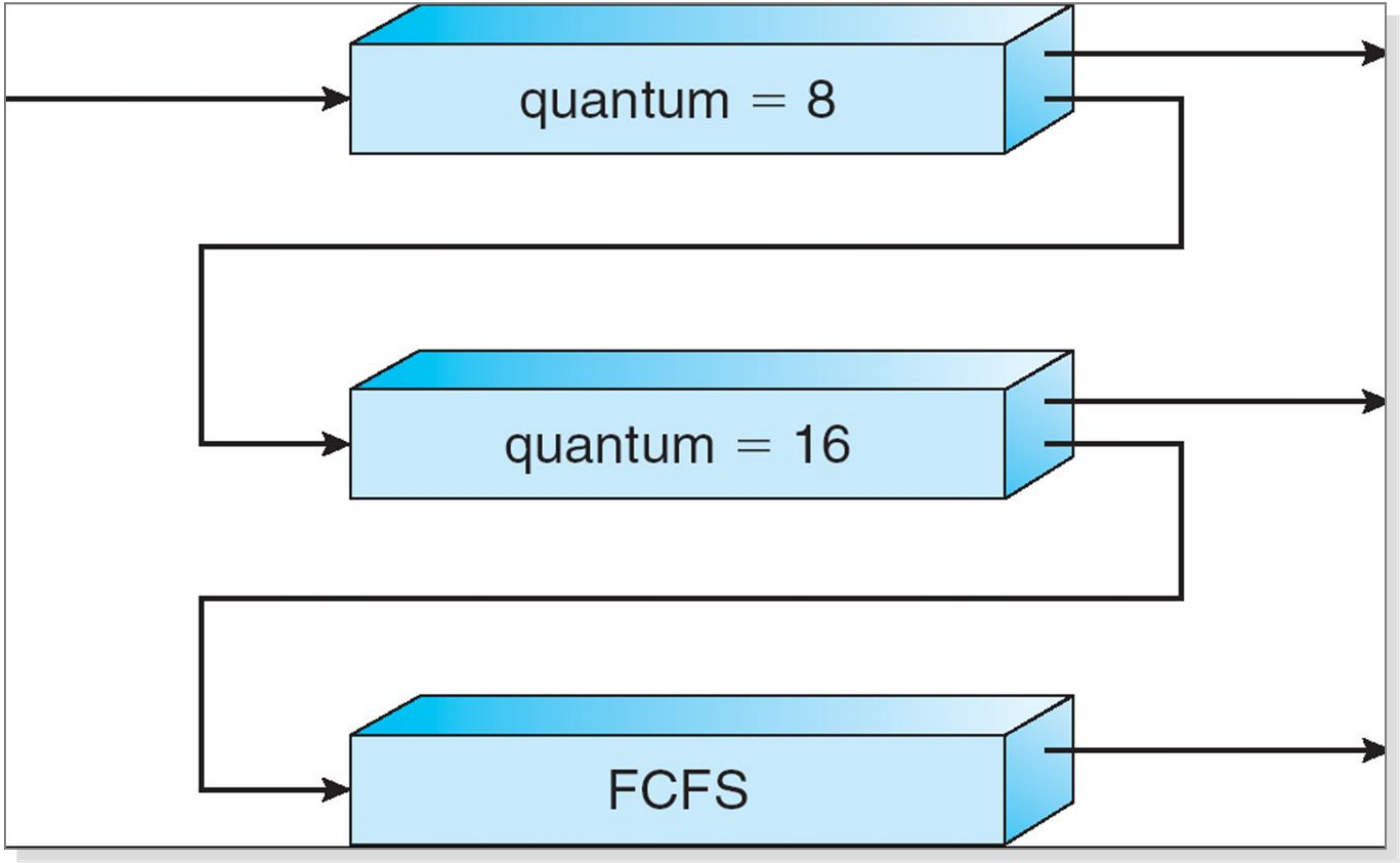
Multilevel Queue Scheduling

- Scheduling must be done between the queues
  - **Fixed priority scheduling**
    - serve all from highest priority, then next priority, etc.
      - e.g., serve all from foreground then from background.
    - Possibility of starvation.
  - **Time slice**
    - each queue gets a certain amount of CPU time which it can schedule amongst its processes
      - e.g., 80% to foreground in RR, 20% to background in FCFS

Multilevel Feedback Queue Scheduling

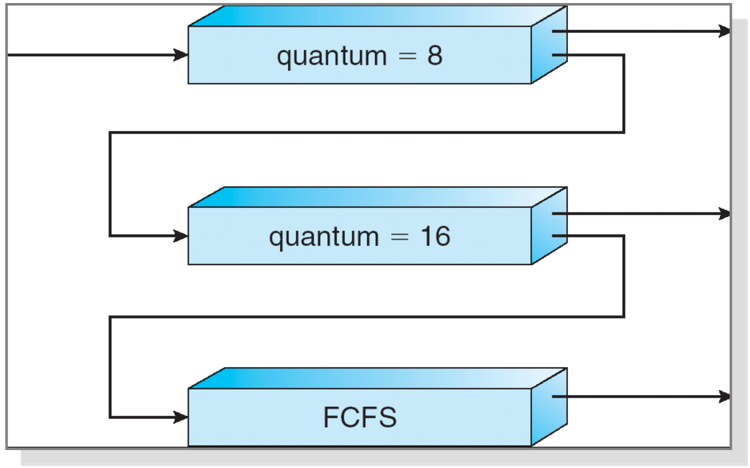
- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler is defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method to determine *when to upgrade a process*
  - method to determine *when to demote a process*
  - method to determine *which queue a process will enter* when that process needs service

Example of Multilevel Feedback Queue



Example of Multilevel Feedback Queue

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



- Scheduling
  - A new job enters queue Q_0 which is served RR. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served RR and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queue

- I/O bounded
  - The queue with highest priority
- CPU bounded
  - Lower and lower

- No need to estimate the length of the next CPU bursts

Scheduling Details

- Countermeasure: user action that can foil intent of the OS designer
  - To avoid moving to a lower priority queue, a user may put in a bunch of meaningless I/O , e.g., output a meaningless char on the screen regularly, to keep its priority high
  - Of course, if everyone did this, wouldn’t work!

Other issues

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor system
  - Only homogeneous systems are discussed here
- Scheduling
  1. Each processor has a separate queue
  2. All processes use a common ready queue
     a. Each processor is self-scheduling
     b. Appoint a processor as scheduler for the other processes (the master-slave structure)

Real-Time Scheduling

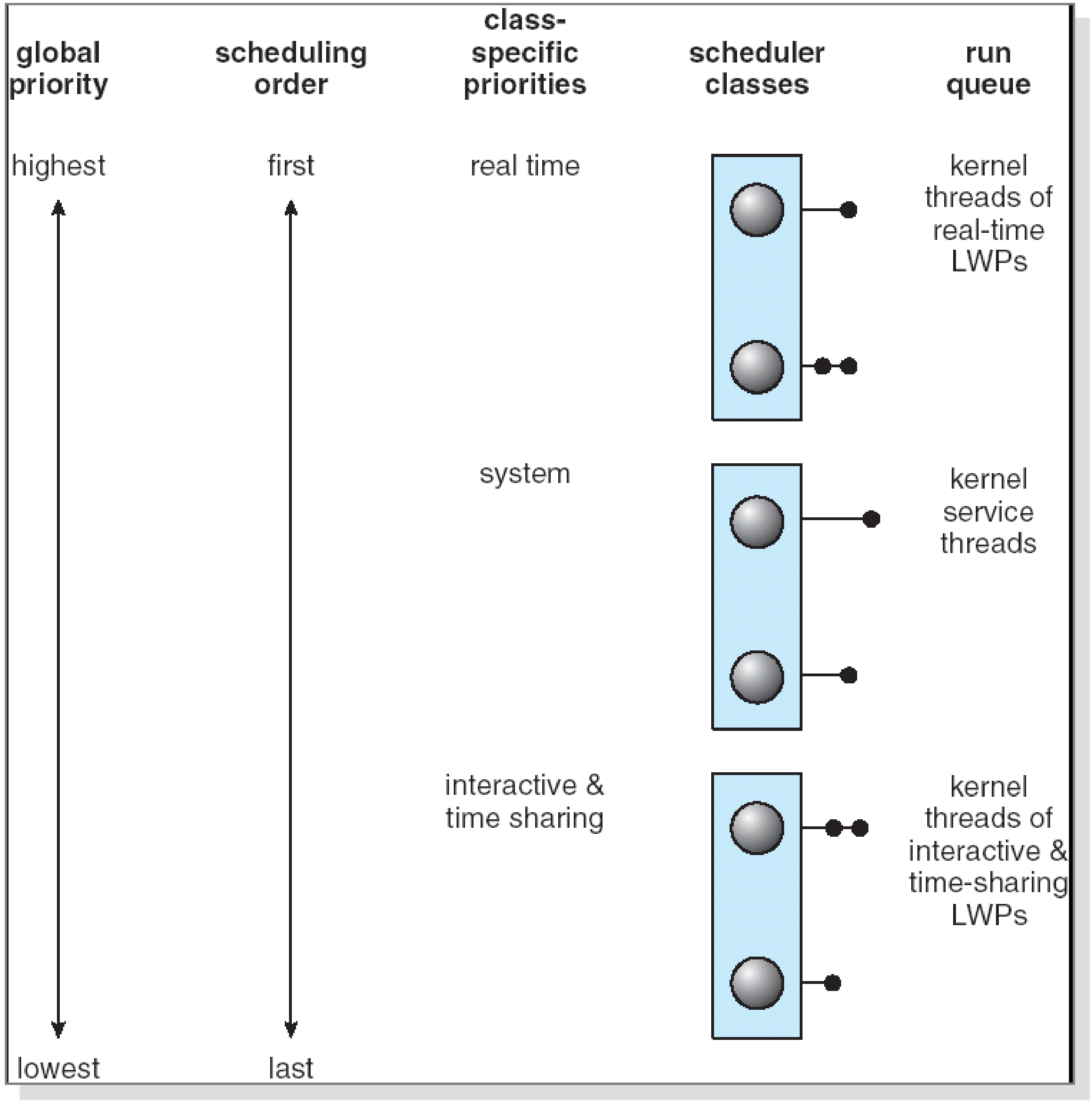
- Timesharing vs Realtime
- **Hard real-time** systems – required to complete a critical task within a guaranteed amount of time
- **Soft real-time** computing – requires that critical processes receive priority over less fortunate ones

Operating Systems Examples

Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Solaris Scheduling (Priority-based)



Windows XP Priorities (priority-based, preemptive)

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Win32 APIs

- Suspend/ResumeThread: 挂起正在运行的线程或激活一个暂停运行的线程。
- Get/SetPriorityClass：读取或设置进程的基本优先级类型
- Get/SetThreadPriority：读取或设置线程相对优先级
- Get/SetProcessPriorityBoost：读取或设置当前进程缺省优先级提升控制
- Get/SetThreadPriorityBoost：读取或设置暂时提升线程优先级状态
- Get/SetProcessAffinityMask：读取或设置进程的亲合处理器集合
- SetThreadAffinityMask：设置线程的亲合处理器集合，只允许该线程在指定处理器集合运行
- SetThreadIdealProcessor：设置特定线程的首选处理器
- SwitchToThread：当前线程放弃一个或多个时间配额的运行
- Sleep：使当前线程等待指定的一段时间（单位为毫秒）。0表示放弃该线程的剩余时间配额。
- SleepEx：使当前线程进入等待状态，直到I/O处理完成

Linux Scheduling

- Two algorithms: time-sharing and real-time
- Time-sharing
  - Prioritized credit-based – process with most credits is scheduled next
  - Credit subtracted when timer interrupt occurs
  - When credit = 0, another process is chosen
  - When all processes have credit = 0, recrediting occurs
    - Based on factors including priority and history
- Real-time
  - Soft real-time

Algorithm Evaluation

Algorithm Evaluation

- Criteria to select a CPU scheduling algorithm may include several measures, such as:
  - **Maximize CPU utilization** under the constraint that the maximum response time is 1 second
  - **Maximize throughput** such that turnaround time is (on average) linearly proportional to total execution time

How to Evaluate a Scheduling Algorithm?

- Deterministic modeling
- Queuing models
- Simulation/Implementation

Algorithm Evaluation

- **Deterministic modeling**
  - Analytic evaluation
    - Input: a given algorithm and a system workload
    - Output: performance of the algorithm for that workload

Algorithm Evaluation

- **Deterministic modeling**
  - A simple and fast method. It gives the exact numbers, allows the algorithms to be compared.
  - It requires exact numbers of input, and its answers apply to only those cases. In general, deterministic modeling is too specific, and requires too much exact knowledge, to be useful.
  - Usage
    - Describing algorithm and providing examples
    - A set of programs that may run over and over again
    - Indicating the trends that can then be proved

Algorithm Evaluation

- **Queuing Models** - Queuing network analysis
  - Using
    - the distribution of service times (CPU and I/O bursts)
    - the distribution of process arrival times
  - The computer system is described as a network of servers. Each server has a queue of waiting processes.
  - Determining
    - utilization, average queue length, average waiting time, etc.

Algorithm Evaluation

- **Simulations**
  - Simulations involve programming a model of the system. Software data structures represent the major components of the system.
  - Coding a simulator can be a major task
  - Generating data to drive the simulator
    - a random number generator.
    - trace tapes: created by monitoring the real system
  - More accurate
  - Expensive (several hours of computer time).
  - Large storage
    - Recording the sequence of actual events.

Algorithm Evaluation

- **Implementation**
  - Put data into a real system and see how it works.
  - The only accurate way
    1. cost is too high
    2. environment will change (All methods have this problem!)
       - e.g., To avoid moving to a lower priority queue, a user may output a meaningless char on the screen regularly to keep itself in the interactive queue.

北京交通大學

Summary

Summary

- Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it
- **FCFS Scheduling:**
  - Run processes to completion in order of submission
  - Pros: Simple
  - Cons: Short jobs get stuck behind long ones
- **Round-Robin Scheduling:**
  - Give each process a small amount of CPU time when it executes; cycle between all ready processes
  - Pros: Better for short jobs
  - Cons: Poor when jobs are same length

Summary

- **Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):**
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: optimal (average response time)
  - Cons: hard to predict future, unfair
- **Multi-Level Feedback Queue Scheduling:**
  - Multiple queues of different priorities
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF

Summary

- Countermeasure: user action that can foil intent of the OS designer
- Evaluation of mechanisms:
  - Analytical/Deterministic modeling
  - Queuing Theory
  - Simulation
  - Implementation

Process Schedule