CSC 112: Computer Operating Systems
Lecture 5

Scheduling

Zonghua Gu

Department of Computer Science,

Hofstra University

Acknowledgement: Lecture slides based on UC Berkeley CS 162: Operating Systems and System Programming

CPU/IO Bursts

- A typical task alternates between bursts of CPU and I/O
  - It uses the CPU for some period of time, then does I/O, then uses CPU again (A job may be pre-empted and forced to give up CPU before finishing current CPU burst)

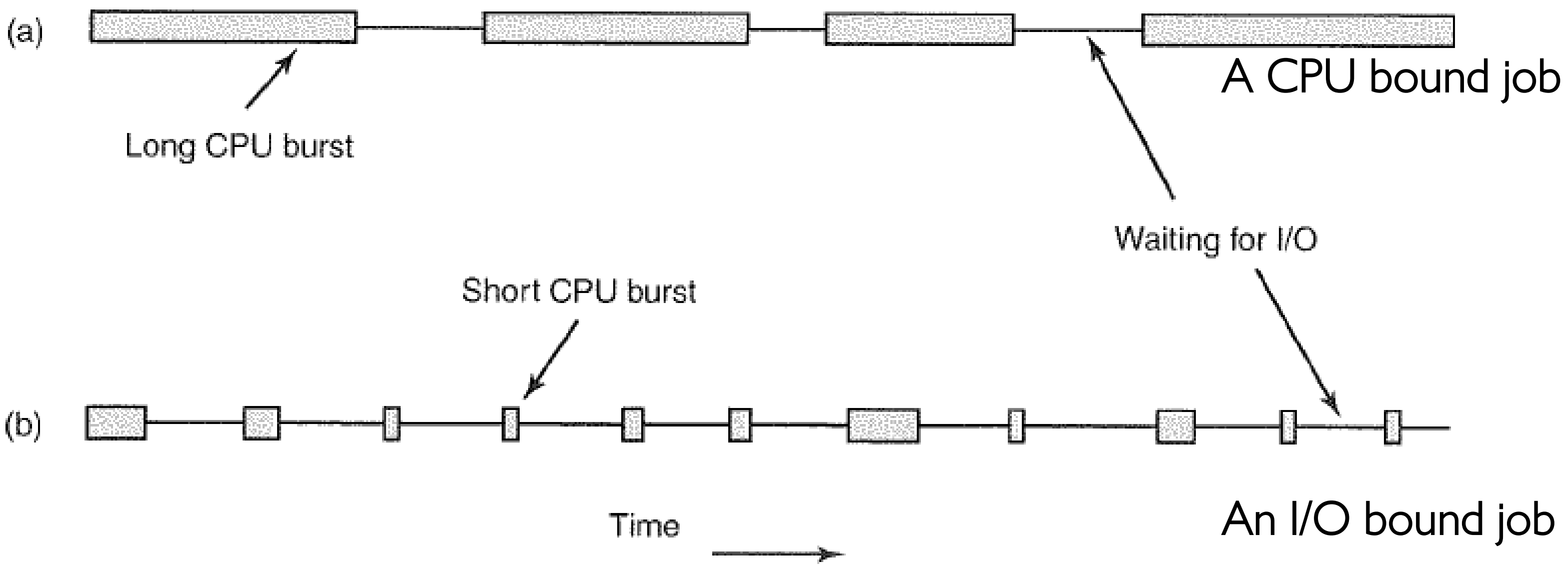
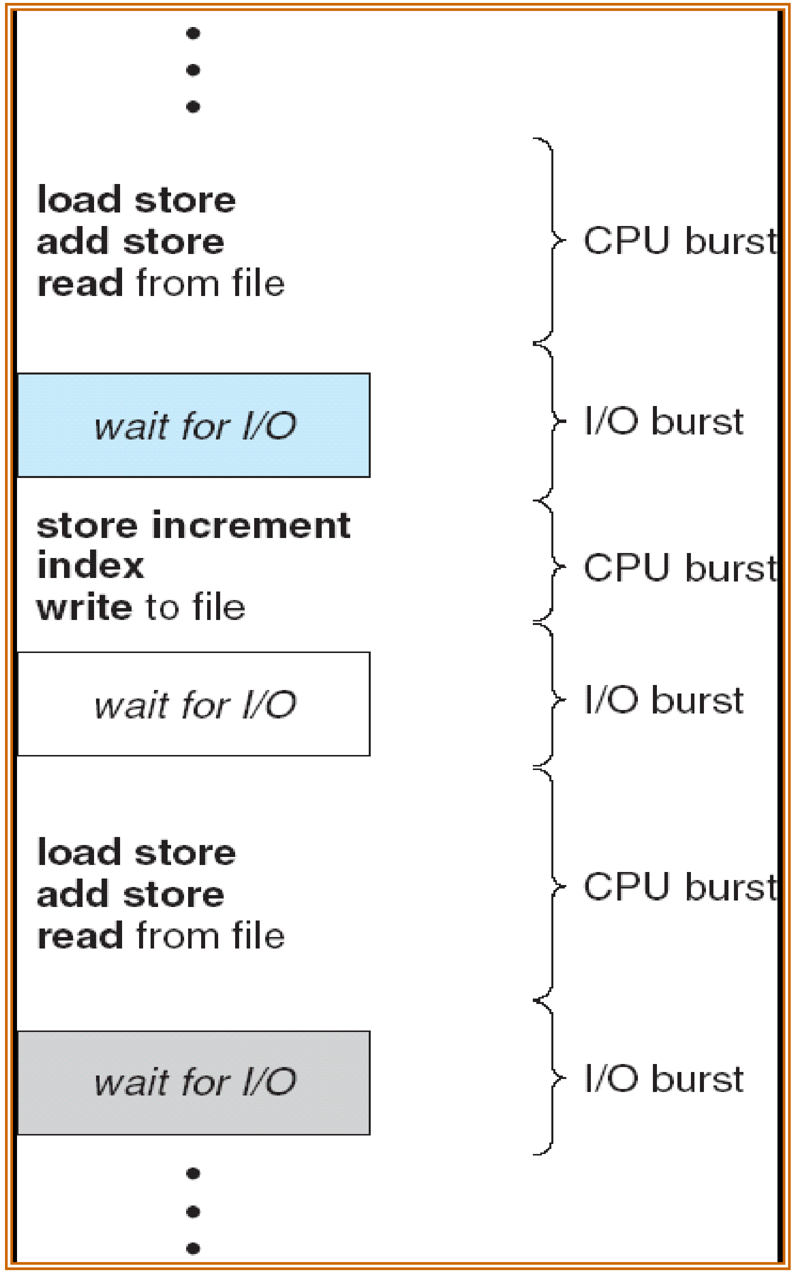
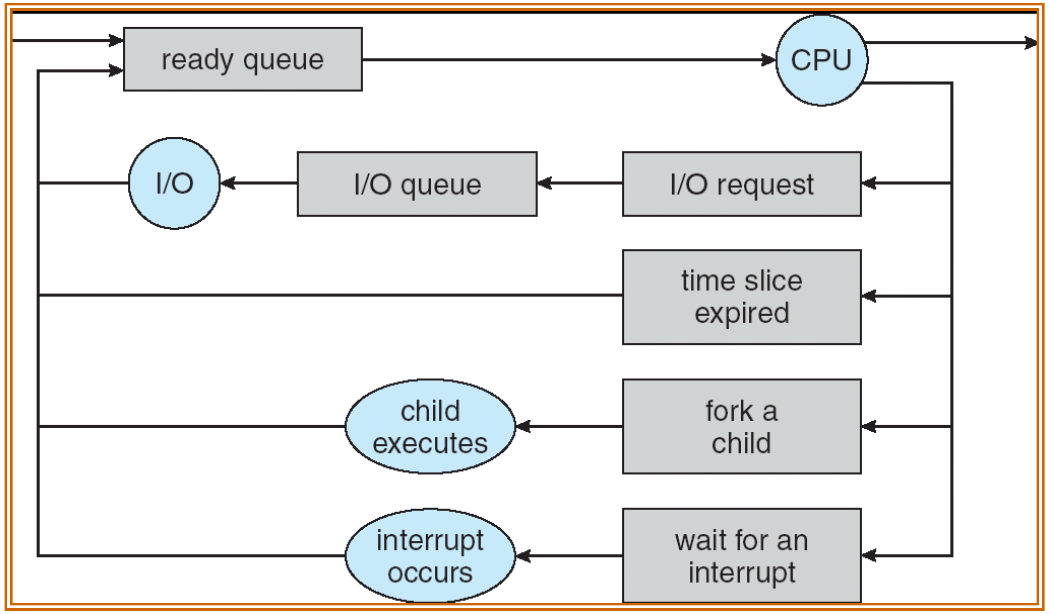


Figure 2-38. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.



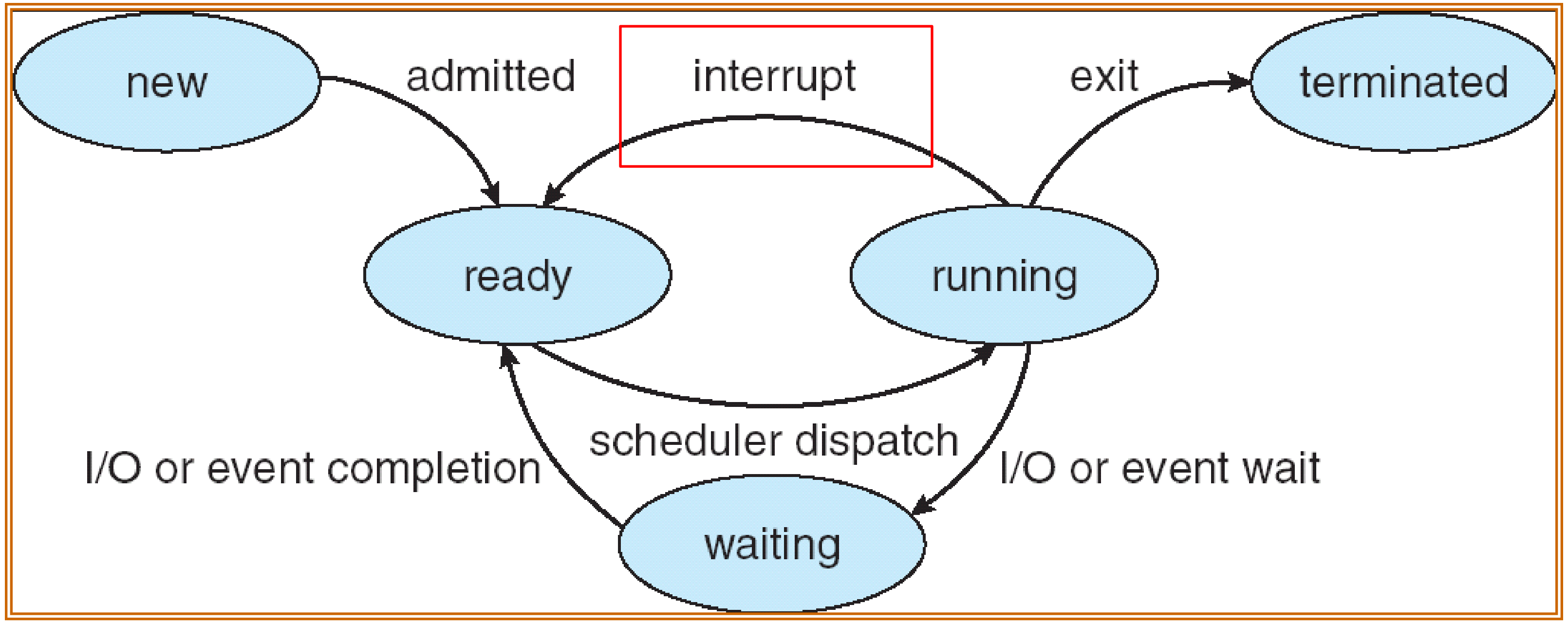
The Scheduling Problem



- Scheduling: When multiple jobs are ready, the scheduling algorithm decides which one is given access to the CPU
  - We use the term "task" to refer to a runnable entity in the OS, which may be a process or a thread. We use the term "job" to refer to a CPU burst of a task

Preemptive vs. Non-Preemptive Scheduling

- With non-preemptive scheduling, once the CPU has been allocated to a process, it keeps the CPU until it releases the CPU either by terminating or by blocking for IO.
- With preemptive scheduling, the OS can forcibly remove a process from the CPU without its cooperation
- Transition from "running" to "ready" only exists for preemptive scheduling



Performance Metrics

- Response time = CompletionTime – ArrivalTime: the total time taken for a job to complete its execution, starting from its arrival time until it finishes. It includes all phases of the process lifecycle: waiting in queues, execution on the CPU, and any I/O operations.
  - Called turnaround time in most textbooks (Please use my definition in this class!)
- Initial waiting time: the time a job spends waiting in the ready queue before it gets its first chance to execute on the CPU
  - Called response time in most textbooks (Please use my definition in this class!)
- Waiting time: the total time a job spends waiting in the ready queue until it finishes
- CPU utilization: percent of time when CPU is busy
- Throughput: # of jobs that complete their execution per time unit
- Different systems may have different objectives. Typically, they cannot be optimized simultaneously by a single scheduling algorithm
  - Maximize CPU utilization
  - Maximize Throughput
  - Minimize Average Response time
  - Minimize Average Waiting time

Common Scheduling Algorithms

- First-Come-First-Served (FCFS) Scheduling
- Round-Robin (RR) Scheduling
- Shortest-Job-First (SJF) Scheduling
- Priority-Based 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"
- Example:

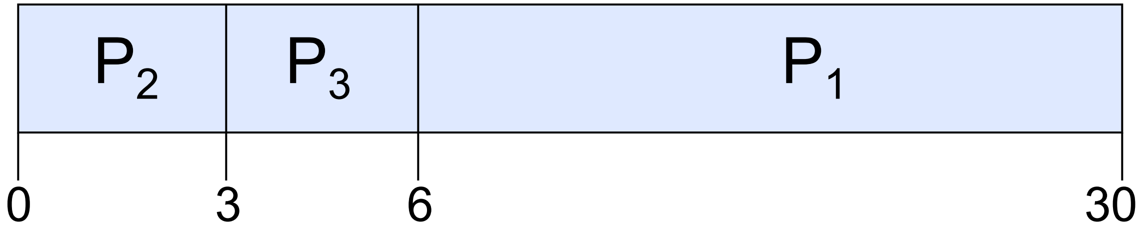
job	Burst Time
P_1	24
P_2	3
P_3	3

  - Suppose jobs arrive in the order of P_1, P_2, P_3 at time 0, i.e., P_1 arrives at time 0, P_2 arrives at time ϵ, P_3 arrives at time 2ϵ
    The Gantt Chart for the schedule is:

  - Initial waiting times: P_1: 0; P_2: 24; P_3: 27
  - Response times: P_1: 24; P_2: 27; P_3: 30
  - Average initial waiting time: (0 + 24 + 27)/3 = 17
  - Average response time: (24 + 27 + 30)/3 = 27
- *Convoy effect:* short job stuck behind long job

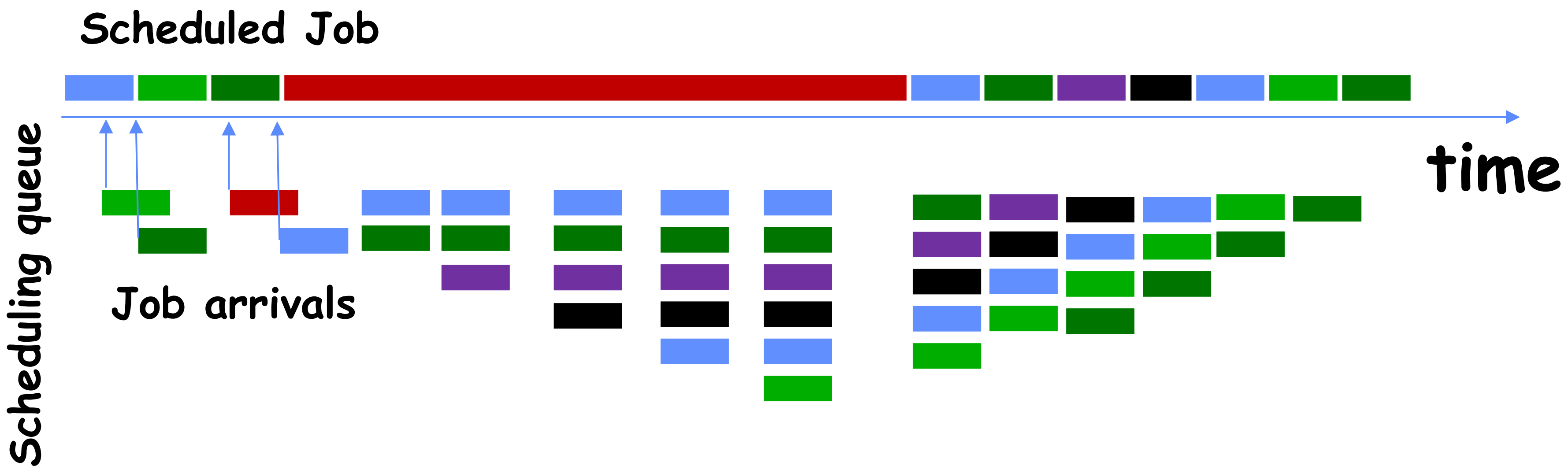
FCFS Scheduling (Cont.)

- Example continued:
  - Suppose jobs arrive in the order: P2 , P3 , P1 at time 0:



  - Initial waiting times: P_1: 6; P_2: 0; P_3: 3
  - Response times: P_1: 30; P_2: 3; P_3: 6
  - Average initial waiting time: (6 + 0 + 3)/3 = 3 (vs. 17 before)
  - Average response time: (30 + 3 + 6)/3 = 13 (vs. 27 before)

Convoy Effect



- With FCFS non-preemptive scheduling, convoys of small tasks tend to build up when a large one is running.

Round Robin (RR) Scheduling

- Round Robin Scheme:
  - Each job gets a small unit of CPU time (*time slice or time quantum*), usually 10-100 milliseconds
  - When quantum expires, the job is preempted and added to the end of the ready queue
  - If the current CPU burst finishes before quantum expires, the job blocks for IO and is added to the end of the ready queue
  - n jobs in ready queue and time quantum is $q \Rightarrow$
    - » Each job gets (roughly) $1/n$ of the CPU time
    - » In chunks of at most q time units
    - » No job waits more than $(n\text{-}1)q$ time units
- OS implementation:
  - Use a periodic timer interrupt to preempt the running job every time quantum, and send it to the back of the ready queue

Example of RR with Time Quantum = 20

- Example:

job	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

  – Suppose jobs arrive in the order of P_1, P_2, P_3, P_4 at time 0. Gantt chart:

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

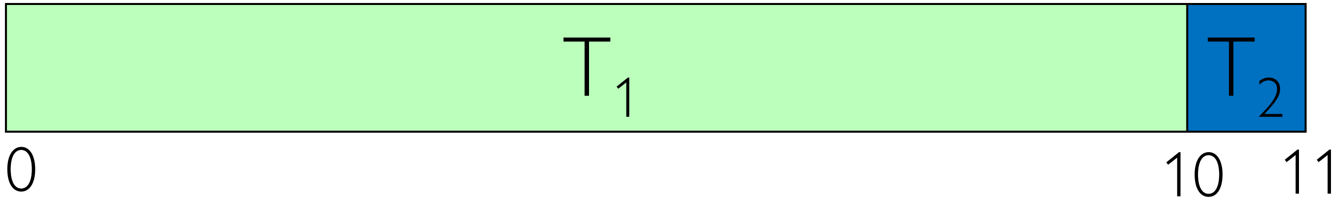
  – Waiting times:
    P_1=(68-20)+(112-88)=72
    P_2=(20-0)=20
    P_3=(28-0)+(88-48)+(125-108)=85
    P_4=(48-0)+(108-68)=88
  – Average waiting time = (72+20+85+88)/4=66¼
  – Response times: P_1: 125; P_2: 28; P_3: 153; P_4: 112
  – Average response time = (125+28+153+112)/4 = 104½
- Round-Robin scheduling
  – Pro: Better for short jobs, Fair
  – Con: Context-switching overhead adds up for long jobs

Quantum size

- Choice of time quantum size *q*:
  - – *q* must be large with respect to context-switching overhead,
  - – *q* too large: fairness is reduced. RR with infinite time quantum is equivalent to FCFS
  - – *q* too small: too many context-switches with high overhead
- Typical time quantum in modern OSes is between 10ms – 100ms
- Typical context-switching overhead is 0.1ms – 1ms
  - – Roughly 1% overhead due to context-switching

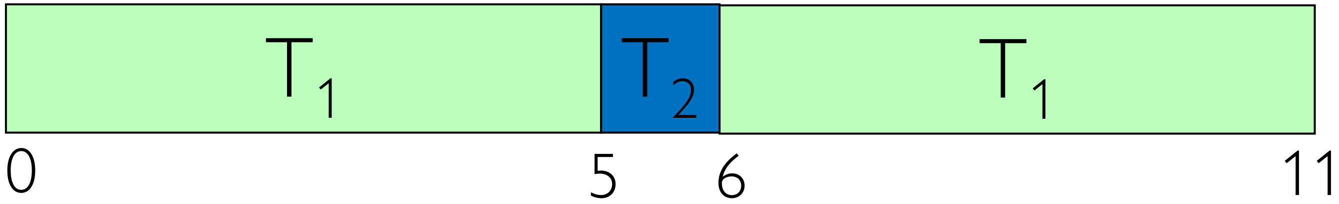
Decrease Response Time w. Decreasing Quantum

- T_1: Burst Length 10
- T_2: Burst Length 1
- Suppose jobs arrive in the order of T_1, T_2 at time 0
- *Q* = 10



  – Average Response Time = (10 + 11)/2 = 10.5

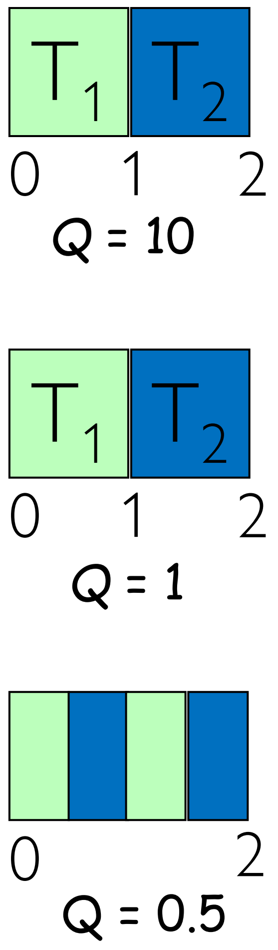
- *Q* = 5



  – Average Response Time = (11 + 6)/2 = 8.5

Response Time vs. Time Quantum

- T_1: Burst Length 1
- T_2: Burst Length 1
- Suppose jobs arrive in the order of T_1, T_2 at time 0
- $Q = 10$
  - Average Response Time = (1 + 2)/2 = 1.5
- $Q = 1$
  - Average Response Time = (1 + 2)/2 = 1.5
- Q = 0.5
  - Average Response Time = (1.5 + 2)/2 = 1.75



FCFS vs. Round Robin

- Assuming zero-cost context-switching time, RR may not be better than FCFS, e.g., when all jobs have equal execution time
- Simple example:

  10 jobs, each take 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time

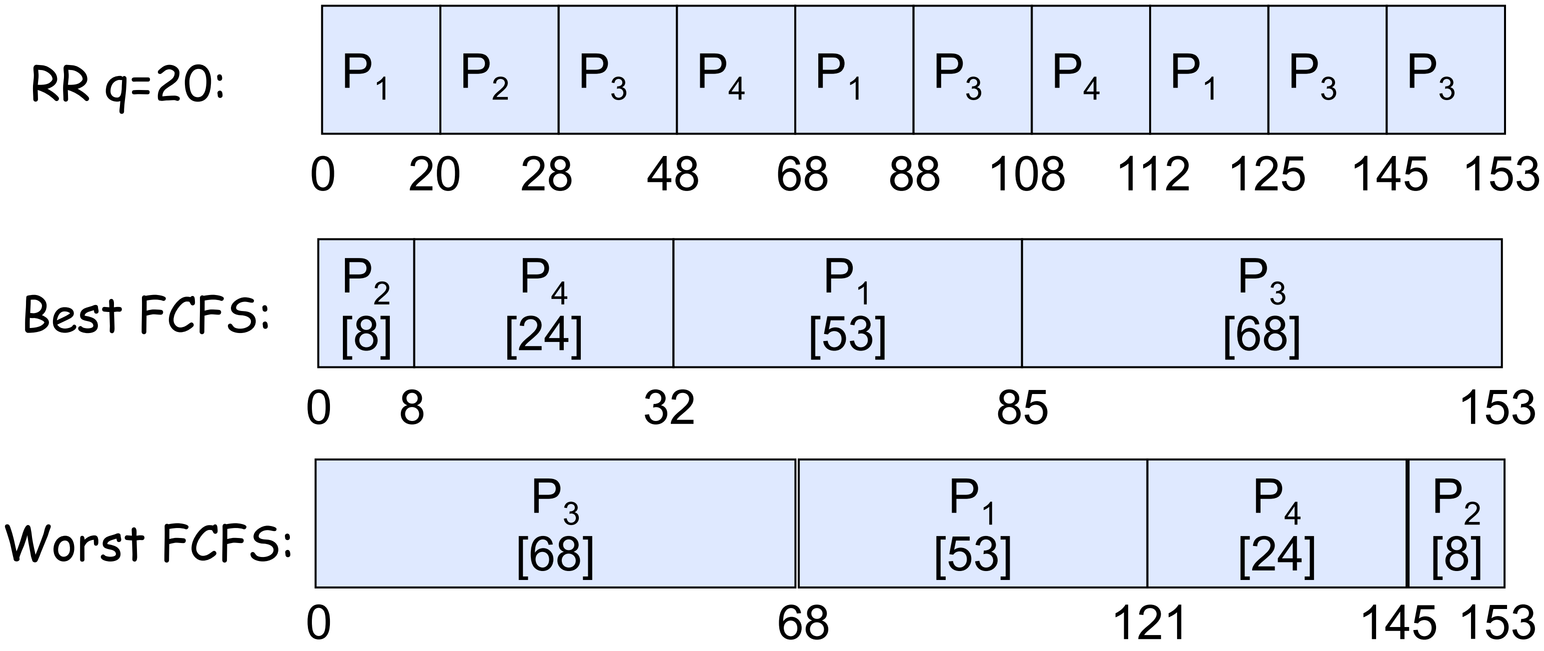
- response times:

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

  – Both RR and FCFS finish at the same time
  – Average response time is much worse under RR than FCFS
- Frequent context switches under RR hurts cache locality and increases job execution time due to increased cache miss rate

Consider the Previous Example

Job	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24



- When jobs have uneven length, it seems to be a good idea to run short jobs first!

Earlier Example with Different Time Quantum

	Quantum	P_1	P_2	P_3	P_4	Average
Wait 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

Average Wait Time and Response Time (Completion Time) may increase or decrease with varying time quantum

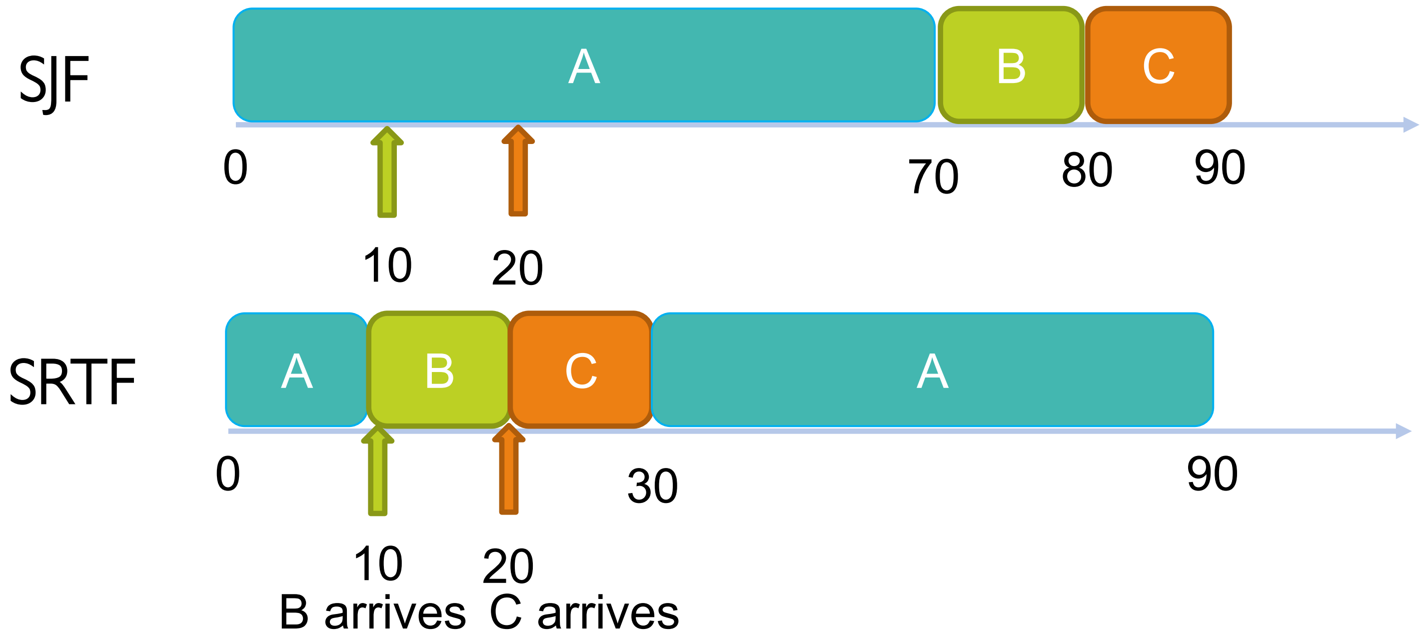
SJF and SRTF

- If we know job execution times at arrival time (predict the future), then we can implement SJF and SRTF
- Shortest Job First (SJF):
  - Non-preemptive scheduling: Run whatever job has least amount of computation to do
  - Still suffers from convoy effect due to non-preemption
- Shortest Remaining Time First (SRTF):
  - Preemptive scheduling: if a new job arrives with remaining time less than remaining time of currently-executing job, preempt the current job
    - » In case of a tie (a new job arrives with remaining time equal to remaining time of currently-executing job), then do not preempt the current job
- Key idea: Give higher priority to short jobs and finish them quickly
  - Big benefit for short jobs, only small delay effect on long ones
  - Result is better average response time

SJF and SRTF Example

- SRTF achieves shorter average response time (Avg RT) than SJF, thanks to preemptive scheduling

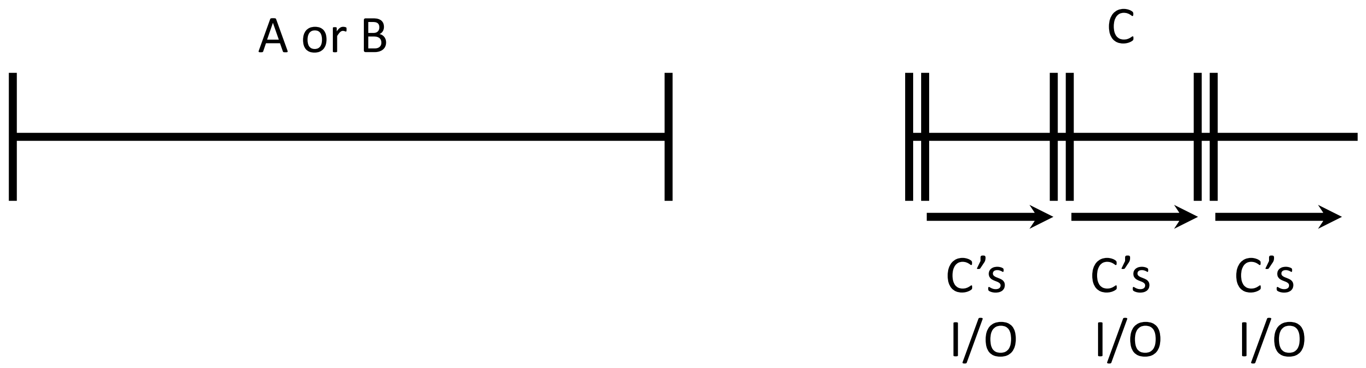
Job	Arrival time	Exec Time	SJF Finishing Time	SJF Response Time	SRTF Finishing Time	SRTF Response Time
A	0	70	70	70	90	90
B	10	10	80	70	20	10
C	20	10	90	70	30	10
				Avg RT 70		Avg RT 37



Optimality of SJF and SRTF

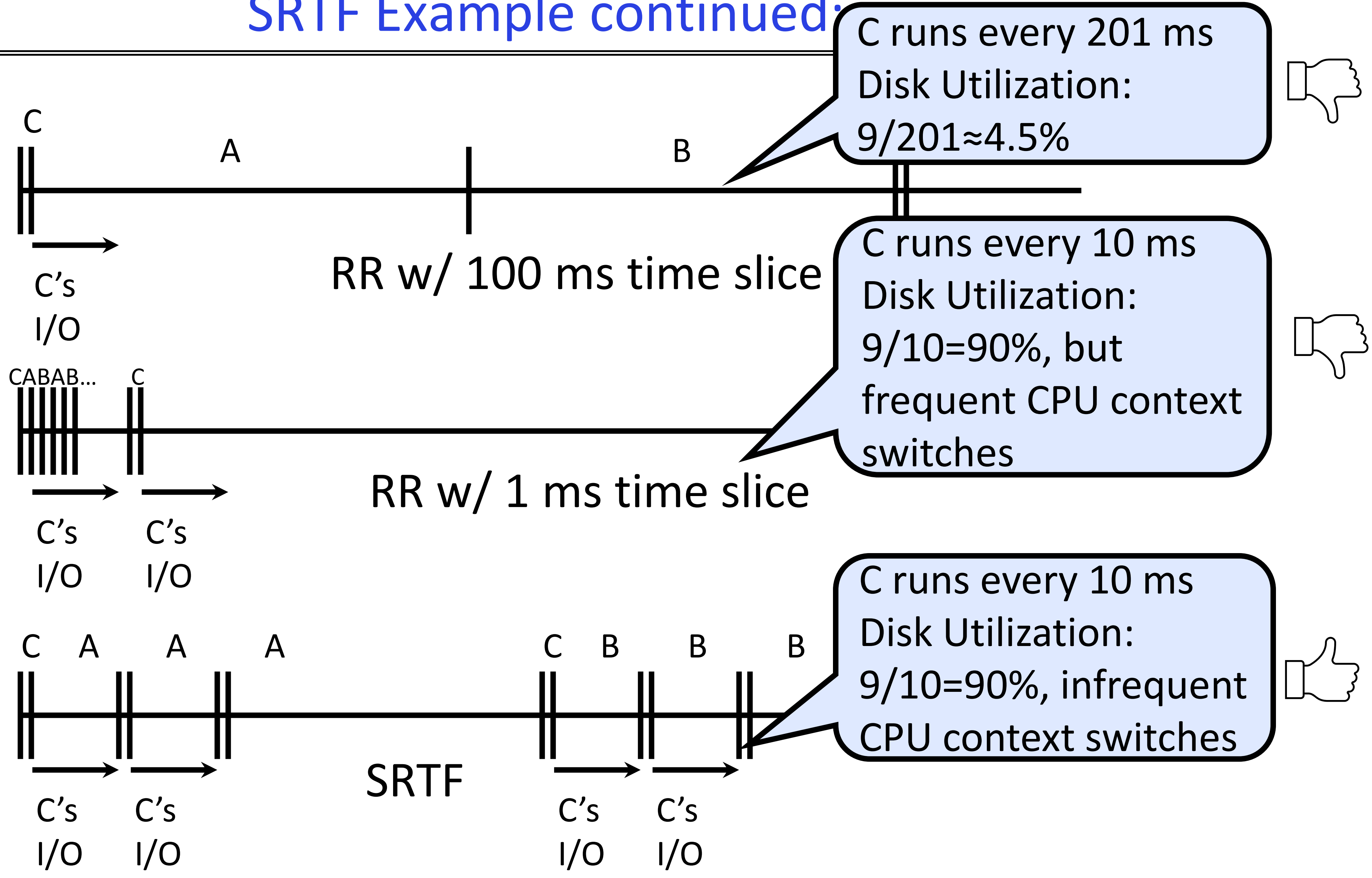
- SJF is the optimal scheduling algorithm for minimizing the average response time under the following assumptions:
  - All jobs only use the CPU (no I/O)
  - All jobs arrive at the same time
  - Job execution times are known in advance
  - Non-preemptive scheduling
- SRTF is the optimal scheduling algorithm for minimizing the average response time under the following assumptions:
  - All jobs only use the CPU (no I/O)
  - Job execution times are known in advance
  - Preemptive scheduling
- Comparison of SRTF with FCFS
  - If all jobs have the same length (execution time)
    - » SRTF becomes the same as FCFS (i.e. FCFS is optimal if all jobs the same length)
  - If jobs have varying length
    - » SRTF is better, since short jobs are not stuck behind long ones

Example to illustrate benefits of SRTF



- Three jobs:
  - A, B: both CPU bound, run for an hour
    C: I/O bound, runs in a loop of 1ms CPU followed by 9ms disk I/O
  - If each job runs alone without interference, then C uses 90% of disk, A or B uses 100% of CPU
- With FCFS:
  - A and B may arrive and keep CPU busy for two weeks before C is scheduled
- What about RR or SRTF?

SRTF Example continued:



SRTF Discussions

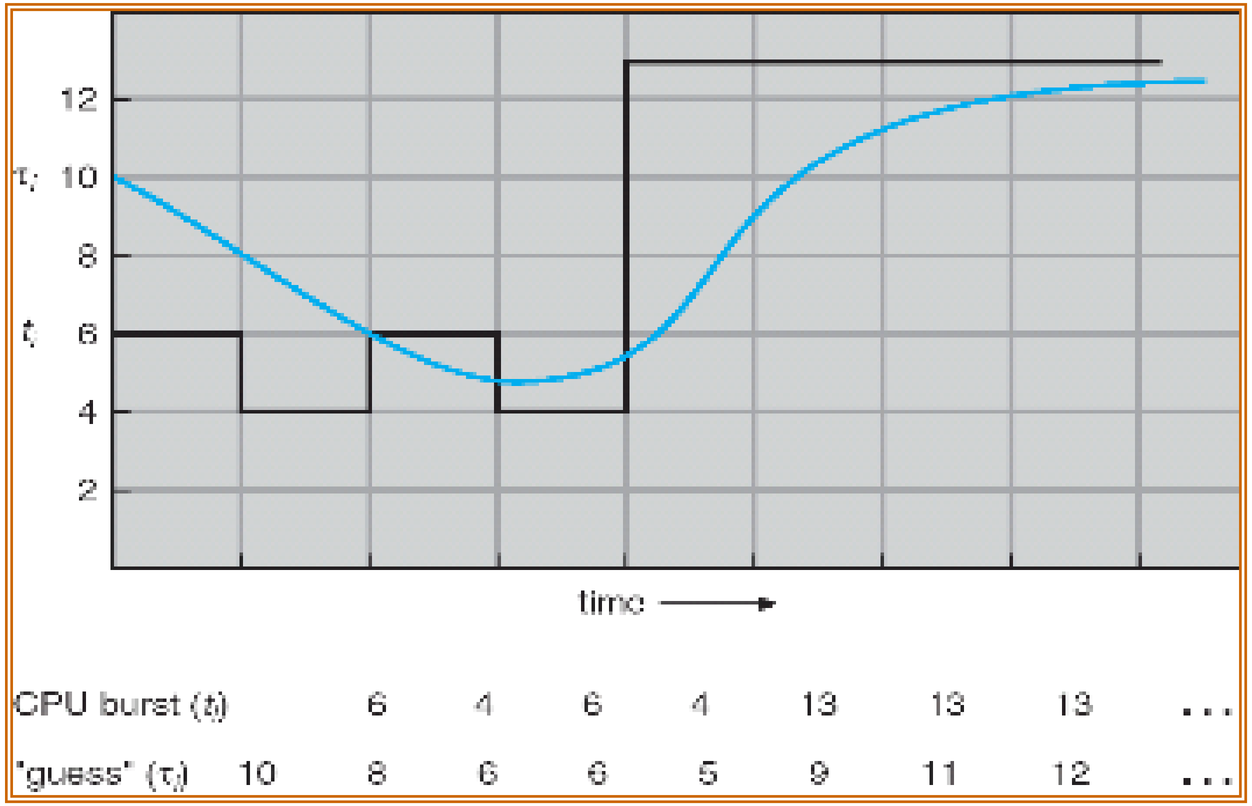
- How to predict job execution time?
  - Runtime measurement and profiling for typical inputs
  - Offline static analysis
  - Difficult and error-prone in general
- Unfair
  - SRTF can lead to starvation if many small jobs arrive so large jobs never get to run
- SRTF Pros & Cons
  - Pros: Optimal in minimizing average response time)
  - Cons: Hard to predict job execution time; Unfair

Predicting Length of the Next CPU Burst

- Adaptive: Changing policy based on past behavior
  - Works because programs have predictable behavior
    - » If program was I/O bound in recent past, it is likely to be I/O bound in future
- We can use exponential moving averaging $t_n = \alpha x_n + (1 - \alpha) t_{n-1}$, where:
  - x_n is the new input data point
  - t_{n-1} is the previous exponential moving average
  - α is the smoothing factor (0<α<1)
    - α large: fast update of τ_n based on new input. $\alpha = 1 \rightarrow t_n = x_n$ is equal to the new input data point at each step
    - α small: slow update of τ_n based on new input. $\alpha = 0 \rightarrow t_n = t_0$ stays constant and not affected by new input data point
    - Appropriate choice of α lets t_n track the input data points while smoothing out sensor noise

Predicting Length of the Next CPU Burst: α=0.5

- Compute $t_n = \alpha x_n + (1-\alpha)t_{n-1}$ with initial guess $t_0 = 10$, assuming smoothing factor α=0.5
- $t_1 = \alpha x_1 + (1-\alpha)t_0 = 0.5*6 + 0.5*10 = 8$
- $t_2 = \alpha x_2 + (1-\alpha)t_1 = 0.5*4 + 0.5*8 = 6$
- $t_3 = \alpha x_3 + (1-\alpha)t_2 = 0.5*6 + 0.5*6 = 6$
- $t_4 = \alpha x_4 + (1-\alpha)t_3 = 0.5*4 + 0.5*6 = 5$
- $t_5 = \alpha x_5 + (1-\alpha)t_4 = 0.5*13 + 0.5*5 = 9$
- $t_6 = \alpha x_6 + (1-\alpha)t_5 = 0.5*13 + 0.5*9 = 11$
- $t_7 = \alpha x_7 + (1-\alpha)t_6 = 0.5*13 + 0.5*11 = 12$



Time Series 101: Exponential Moving Average, A Visual Guide
https://www.youtube.com/watch?v=joHKNtPYtLo

Predicting the Length of the Next CPU Burst: α=0.1 or 0.9

- Compute $t_n = \alpha x_n + (1-\alpha)t_{n-1}$ with initial guess τ_0 = 10, assuming α=0.1.
- $t_1 = \alpha x_1 + (1-\alpha)t_0 = 0.1*6 + 0.9*10 = 9.6$
- $t_2 = \alpha x_2 + (1-\alpha)t_1 = 0.1*4 + 0.9*9.6 = 9.0$
- $t_3 = \alpha x_3 + (1-\alpha)t_2 = 0.1*6 + 0.9*9.0 = 8.7$
- $t_4 = \alpha x_4 + (1-\alpha)t_3 = 0.1*4 + 0.9*8.7 = 8.3$
- $t_5 = \alpha x_5 + (1-\alpha)t_4 = 0.1*13 + 0.9*8.3 = 8.7$
- $t_6 = \alpha x_6 + (1-\alpha)t_5 = 0.1*13 + 0.9*8.7 = 9.2$
- $t_7 = \alpha x_7 + (1-\alpha)t_6 = 0.1*13 + 0.9*9.2 = 9.5$

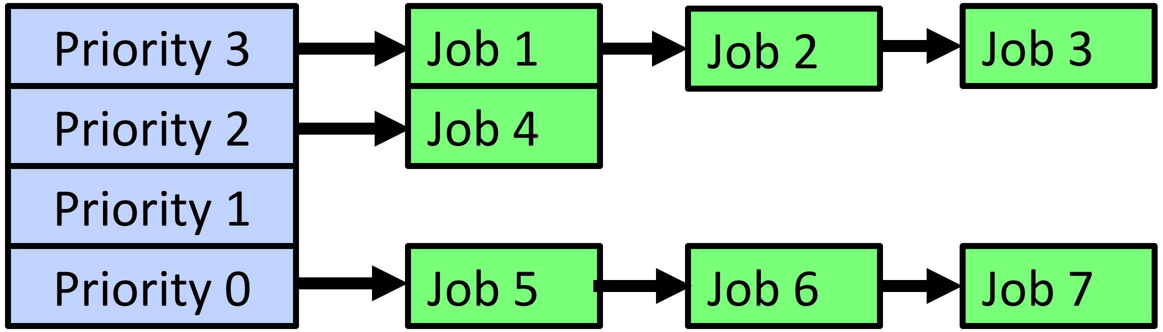
- Compute $t_n = \alpha x_n + (1-\alpha)t_{n-1}$ with initial guess τ_0 = 10, assuming α=0.9.
- $t_1 = \alpha x_1 + (1-\alpha)t_0 = 0.9*6 + 0.1*10 = 6.4$
- $t_2 = \alpha x_2 + (1-\alpha)t_1 = 0.9*4 + 0.1*6.4 = 4.2$
- $t_3 = \alpha x_3 + (1-\alpha)t_2 = 0.9*6 + 0.1*4.2 = 5.8$
- $t_4 = \alpha x_4 + (1-\alpha)t_3 = 0.9*4 + 0.1*5.8 = 4.2$
- $t_5 = \alpha x_5 + (1-\alpha)t_4 = 0.9*13 + 0.1*4.2 = 12.1$
- $t_6 = \alpha x_6 + (1-\alpha)t_5 = 0.9*13 + 0.1*12.1 = 13.0$
- $t_7 = \alpha x_7 + (1-\alpha)t_6 = 0.9*13 + 0.1*13.0 = 13.0$

With low $\alpha = 0.1$, the EMA changes gradually and reacts slowly to new data, staying closer to the starting value.
With high $\alpha = 0.9$, the EMA responds quickly and closely tracks the latest data points.

Comparison Chart

Property	FCFS	SJF	SRTF	RR
Optimize Average Response Time		✓	✓	
Prevent Starvation	✓			✓
Prevent Convoy Effect			✓	✓
No Need to Predict Exec Time	✓			✓

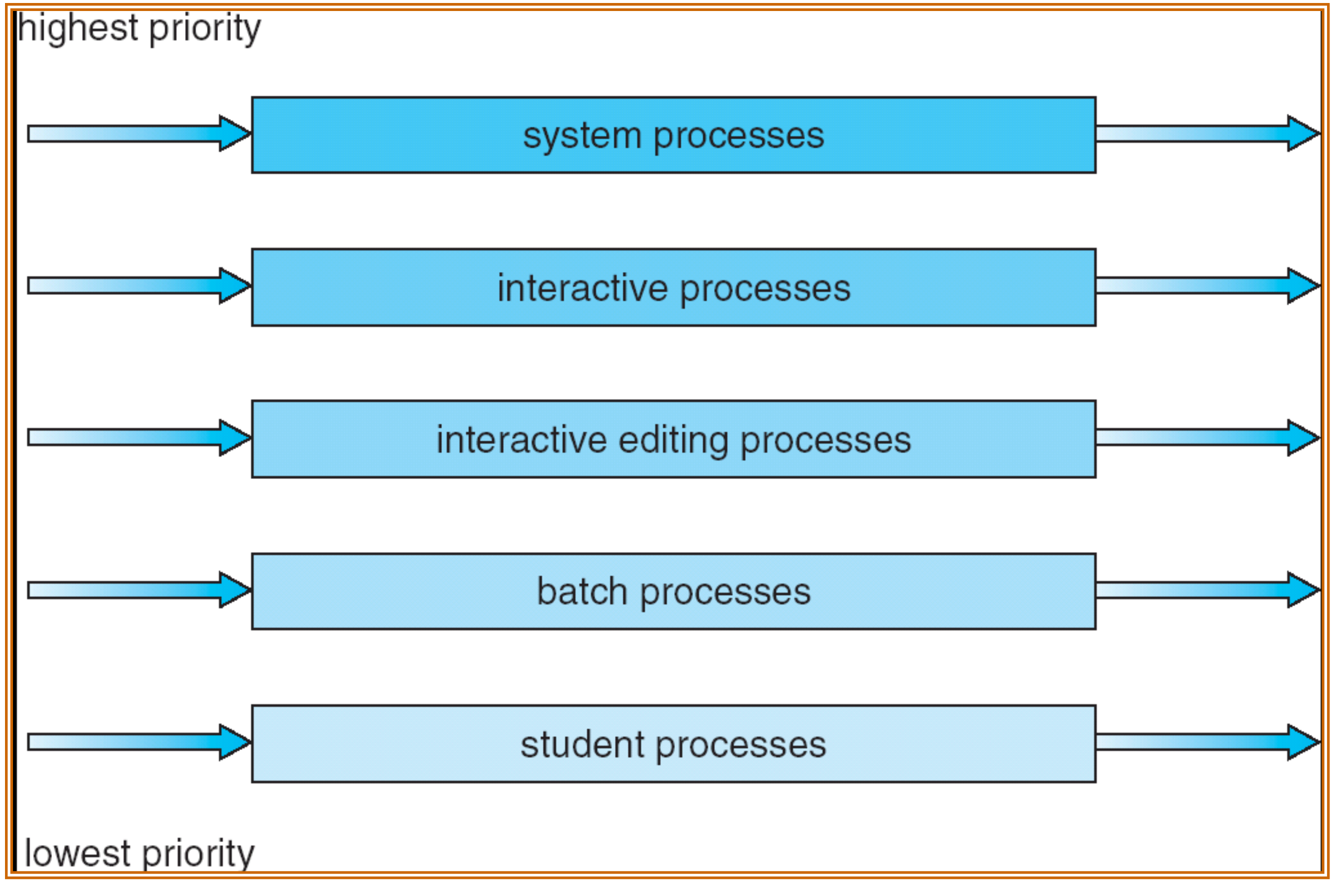
Fixed-Priority Scheduling



- Fixed-Priority Scheduling
  - Each job is assigned a fixed priority
  - Run the highest-priority job in the ready queue at any given time (may be preemptive or non-preemptive)
  - Jobs of equal priority are scheduled with RR
- SJF/SRTF are special cases of priority-based scheduling where priority is the predicted (remaining) job execution time
- Problem: starvation – low priority jobs may never execute
  - Sometimes this is the desired behavior!

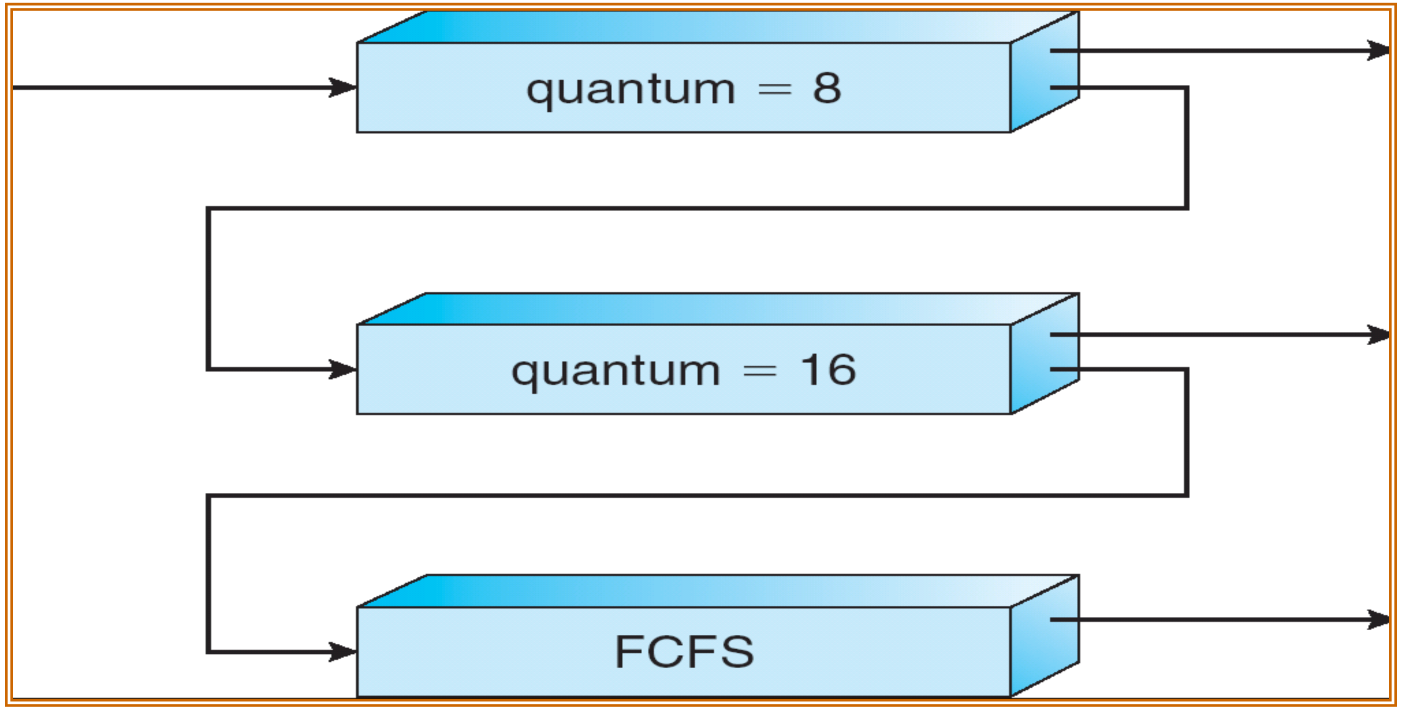
Multi-Level Queue Scheduling

- Ready queue is partitioned into multiple queues, each with different priority
  - Higher priority queues often considered “foreground” tasks
- Each queue has its own scheduling algorithm
  - e.g., foreground queue (interactive jobs/processes) with RR scheduling; background queue (batch jobs/processes) with FCFS scheduling
  - Typically time quantum increases with decreasing priority (highest:1ms, next: 2ms, next: 4ms, etc)
- Scheduling between the queues
  - Fixed priority, e.g., serve all from foreground queue, then from background queue



Multi-Level Feedback Queue Scheduling

- Based on Multi-Level Queue Scheduling, but dynamically adjust each job's priority as follows:
  - It starts in highest-priority queue
  - If quantum expires before the CPU burst finishes, drop down one level
  - If it blocks for I/O before quantum expires, push up one level (or to top, depending on implementation)
  - Time quantum increases with decreasing priority, from 8, to 16, to infinity (RR with infinite time quantum is equivalent to FCFS)



Multi-Level Feedback Queue Scheduling Discussions

- MLFQ approximates SRTF:
  - Long-running CPU-bound jobs/processes are punished and drop down like a rock
  - Short-running I/O-bound processes are rewarded and stay near top
  - No need for prediction of job éxecution time; rely on past behavior to make decision
- User can game the scheduler:
  - e.g., put in a bunch of meaningless I/O like printf() to keep process in the high-priority queue
  - Of course, if everyone did this, this trick wouldn’t work!

Conclusion

- FCFS Scheduling:
  - Run jobs in the order of arrival
  - Cons: Short jobs can get stuck behind long ones
- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  - Run whatever job has the least execution time/least remaining execution time
  - Pros: Optimal (in terms of average response time)
  - Cons: Hard to predict execution time, Unfair
- Priority-Based Scheduling
  - Each job is assigned a fixed priority
- Multi-Level Queue Scheduling
  - Multiple queues of different priorities and scheduling algorithms
- Multi-Level Feedback Queue Scheduling:
  - Automatic promotion/demotion of jobs between queues to approximate SJF/SRTF