CSC 112: Computer Operating Systems
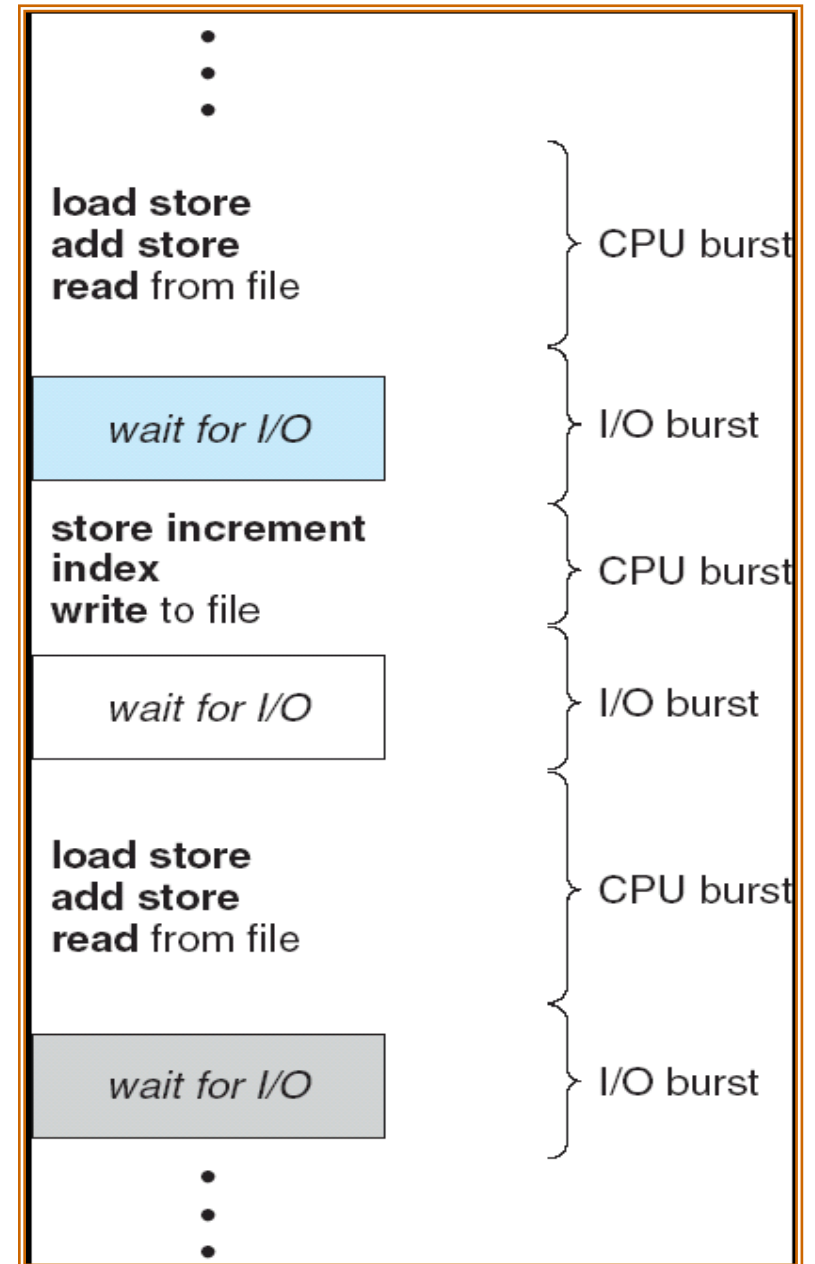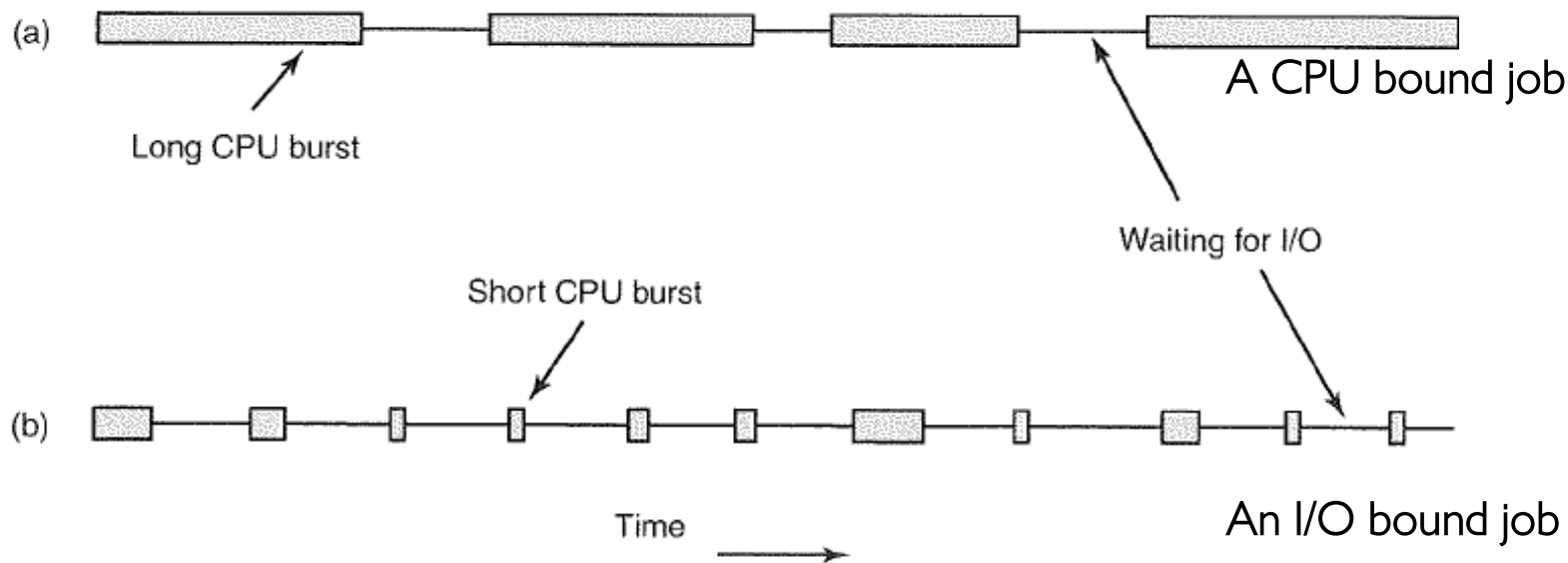## Lecture 5

Scheduling

Department of Computer Science,
Hofstra University
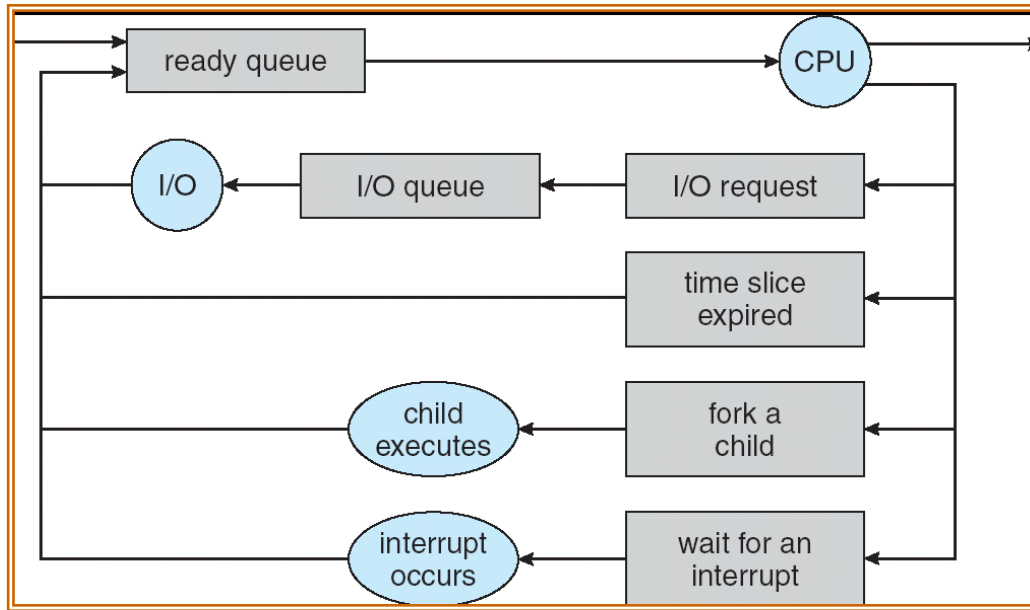
- A typical job alternates between bursts of CPU and I/O

  - 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)

(a) A CPU bound job

Long CPU burst

Waiting for I/O

Short CPU burst

(b) An I/O bound job

Time

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.

load store
add store
read from file

CPU burst

wait for I/O

I/O burst

store increment
index
write to file

CPU burst

wait for I/O

I/O burst

load store
add store
read from file

CPU burst

wait for I/O

I/O burst

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 "job" to refer to a runnable entity in the OS, which may be a process or a thread

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
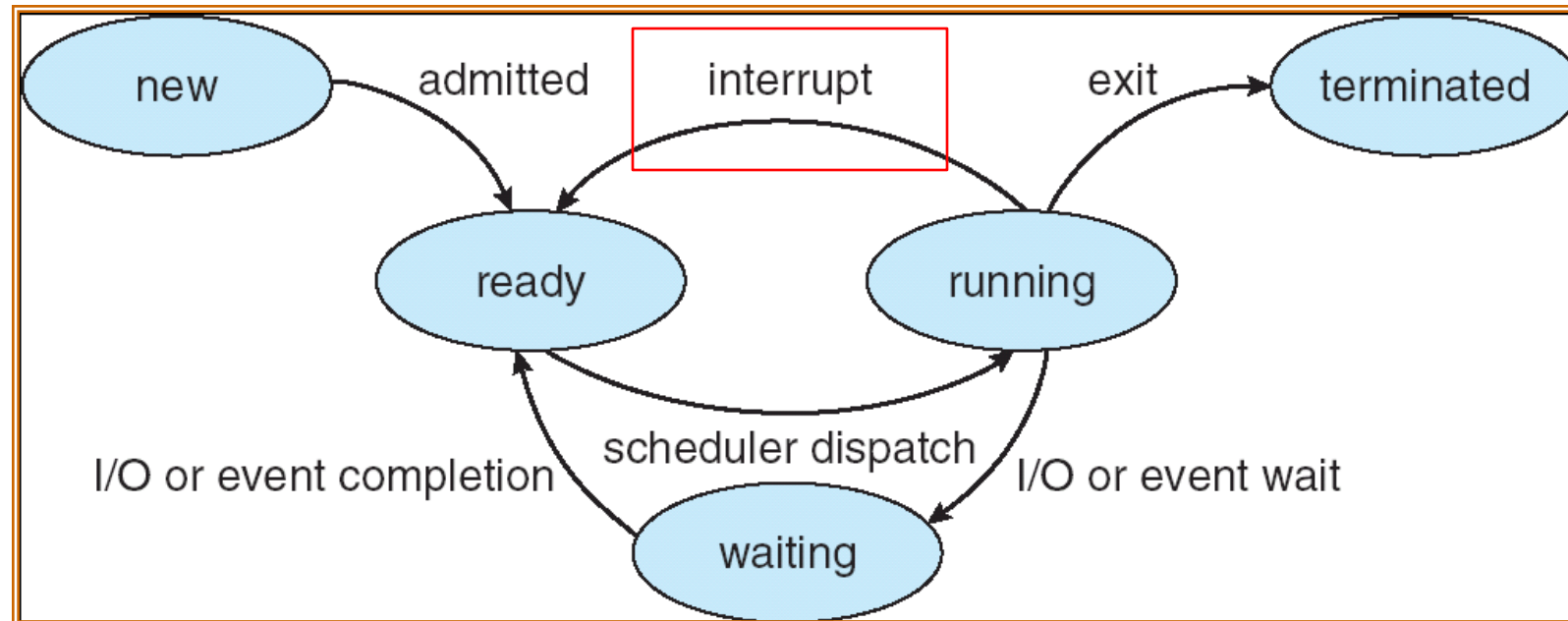
Performance Metrics

- Response time: the total time taken for a job to complete its execution, starting from the moment it arrives until it finishes. It includes all phases of the process lifecycle: waiting in queues, execution on the CPU, and any I/O operations. It can be calculated as CompletionTime – ArrivalTime.
  - Also called turn-around time
- Initial waiting time: the time a job spends waiting in the ready queue before it gets its first chance to execute on the CPU
- CPU utilization: percent of time when CPU is busy
- Throughput: # of jobs that complete their execution per time unit
- Different systems may have different requirements
  - Maximize CPU utilization
  - Maximize Throughput
  - Minimize Average Response time
  - Minimize Average Waiting time
  - Typically, these goals cannot be achieved simultaneously by a single scheduling algorithm

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)
  - 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: 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:

P_1	P_2	P_3

0   24   27   30

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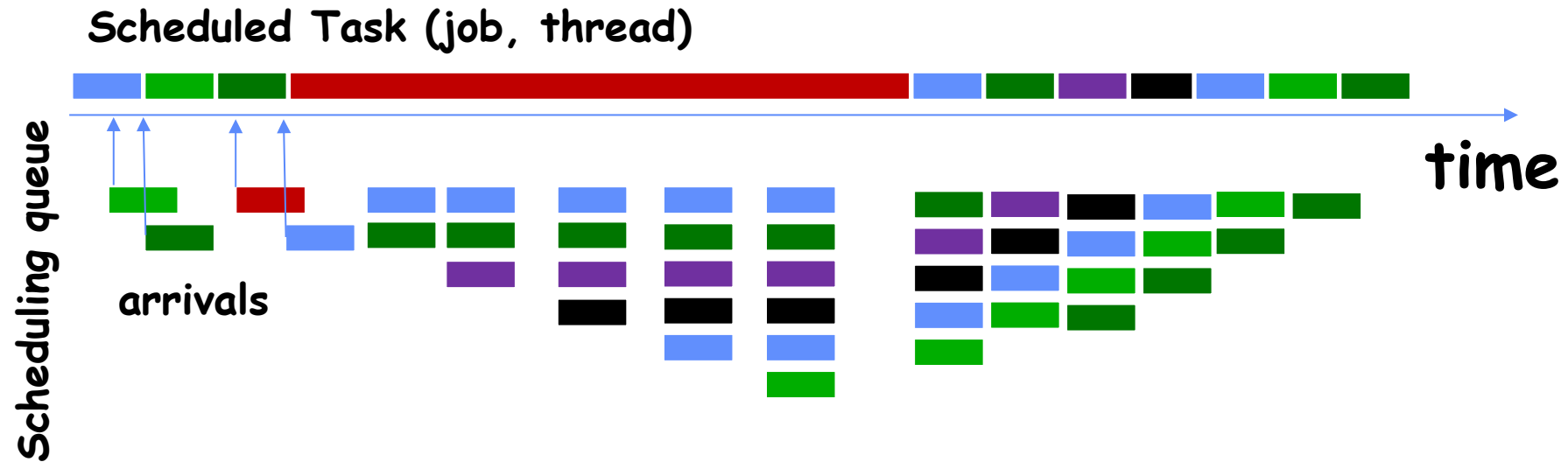- *Convoy effect:* short job stuck behind long job

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

P_2	P_3	P_1

0   3   6   30

  - Initial waiting time for P1: 6; for P2: 0; for P3: 3
  - Average initial waiting time: (6 + 0 + 3)/3 = 3 (vs. 17 before)
  - Average response time: (3 + 6 + 30)/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.

Long job   Short job   Short job
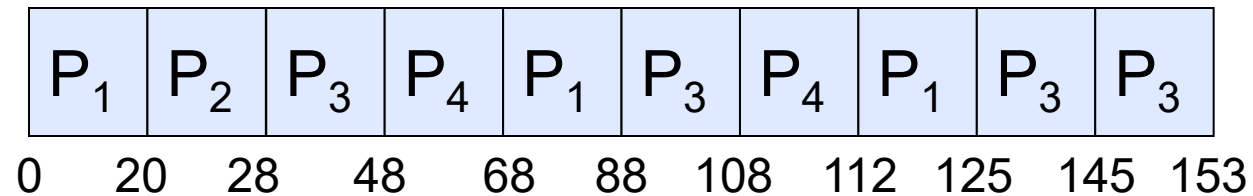
Round Robin (RR) Scheduling

- 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:

job	Burst Time
P_1	53
P_2	8
P_3	68
P_4	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 for $P_1 = (68-20)+(112-88)=72$
  $P_2 = (20-0)=20$
  $P_3 = (28-0)+(88-48)+(125-108)=85$
  $P_4 = (48-0)+(108-68)=88$

  – Average waiting time = $(72+20+85+88)/4 = 66¼$

  – Average response time = $(125+28+153+112)/4 = 104½$

- ## Round-Robin scheduling
  – Pro: Better for short jobs, Fair
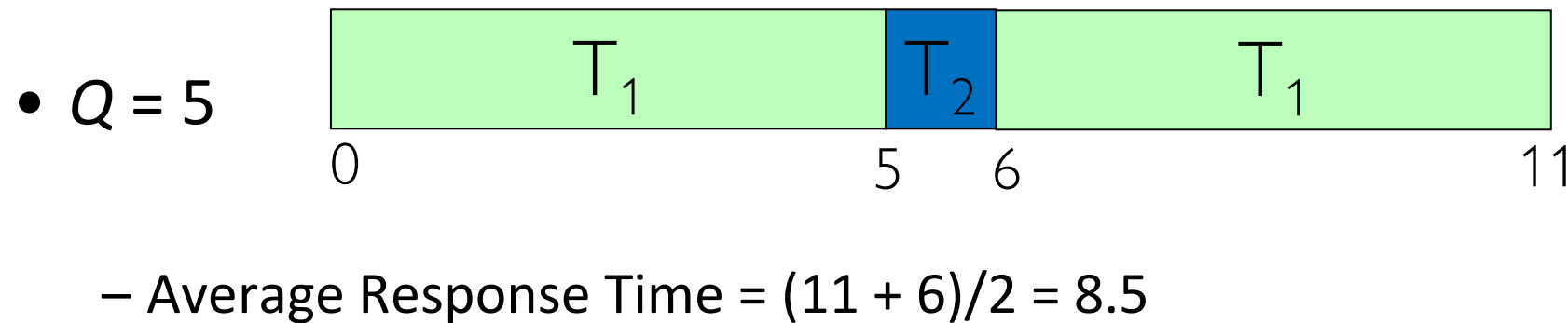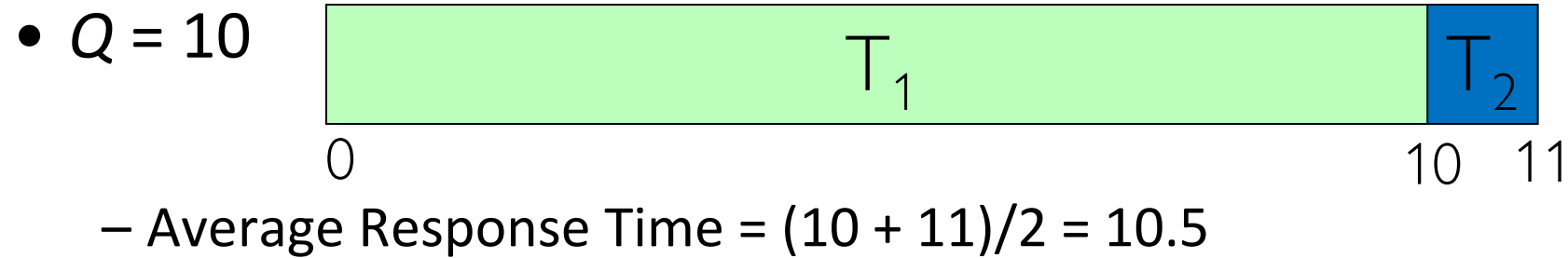 – Con: Context-switching overhead adds up for long jobs

Quantum size

- Choice of quantum size q:
  - q must be large with respect to context-switching overhead,
  - q too large: response time will be long. q very large $\Rightarrow$ FCFS
  - q too small: too many context-switches with high overhead
- Typical time slice in modern OS is between $10\text{ms} - 100\text{ms}$
- Typical context-switching overhead is $0.1\text{ms} - 1\text{ms}$
  - Roughly 1% overhead due to context-switching

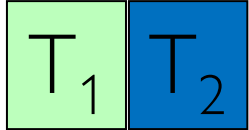# Decrease Response Time w. Decreasing Quantum

- T_1: Burst Length 10

- T_2: Burst Length 1

- $Q = 10$

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

- $Q = 5$

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

- T_1: Burst Length 1
- T_2: Burst Length 1

- $Q = 10$

T_1	T_2

0   1   2

  – Average Response Time = (1 + 2)/2 = 1.5

- $Q = 1$

T_1	T_2

0   1   2

  – Average Response Time = (1 + 2)/2 = 1.5

- T_1: Burst Length 1

- T_2: Burst Length 1

- $Q = 1$

  | T_1 | T_2 |

  0   1   2

  – Average Response Time = $(1 + 2)/2 = 1.5$

- $Q = 0.5$

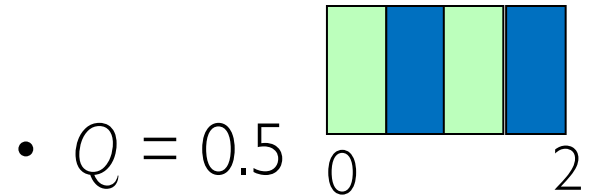

  0   2
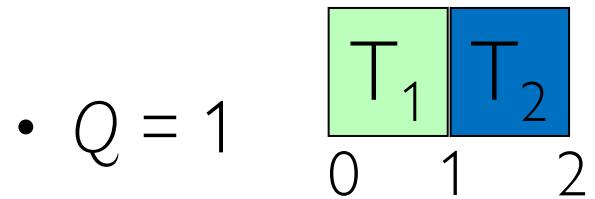
  – Average Response Time = $(1.5 + 2)/2 = 1.75$

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

RR q=20:

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

Best FCFS:

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

0   8   32   85   153

Worst FCFS:

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

0   68   121   145 153
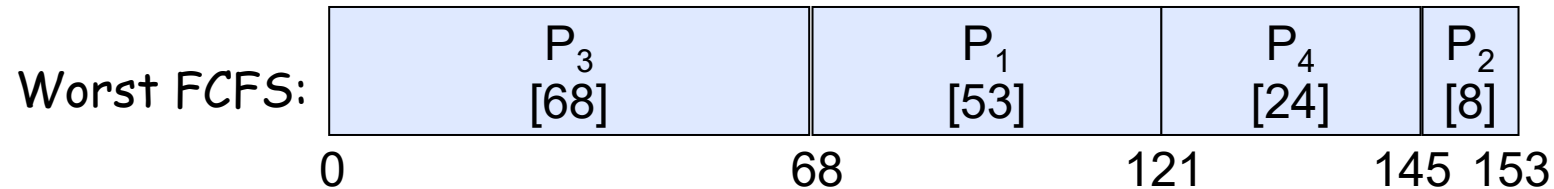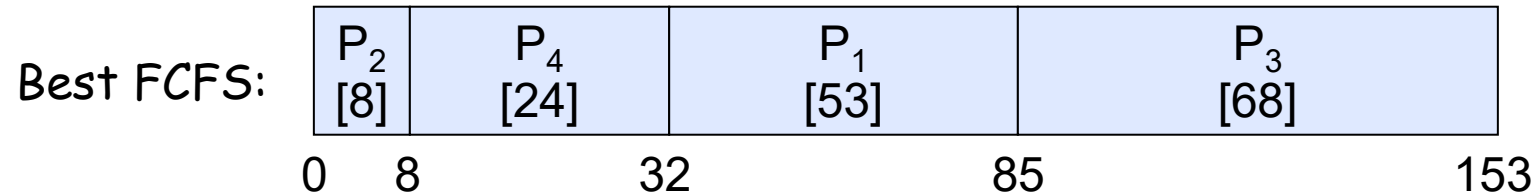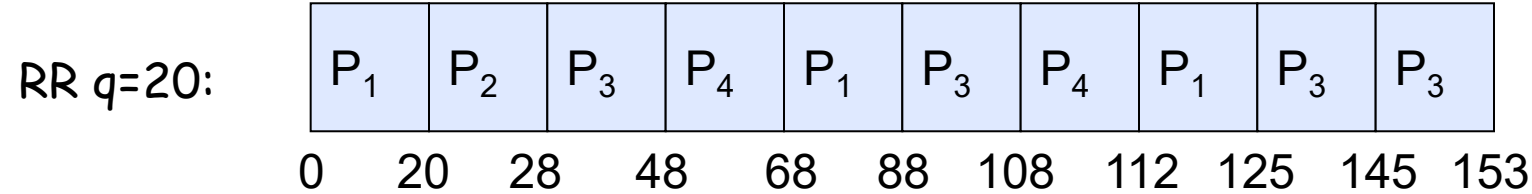
- When jobs have uneven length, it seems to be a good idea to run short jobs first!

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

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.

- 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

- 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
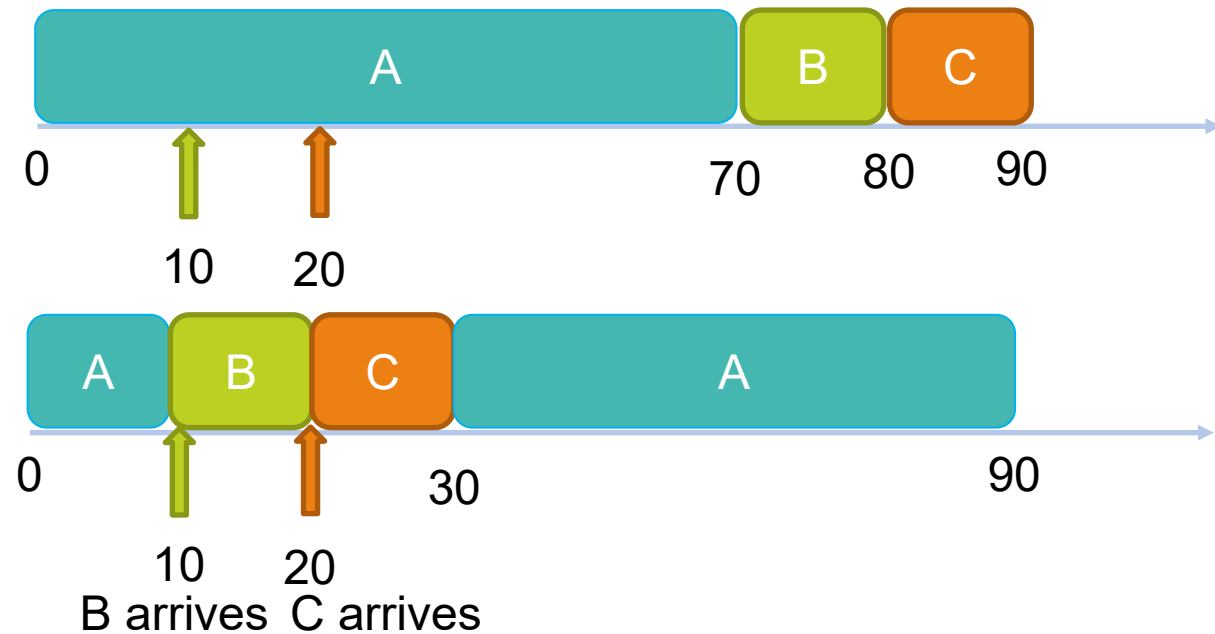
Optimality of SJF and SRTF

- 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
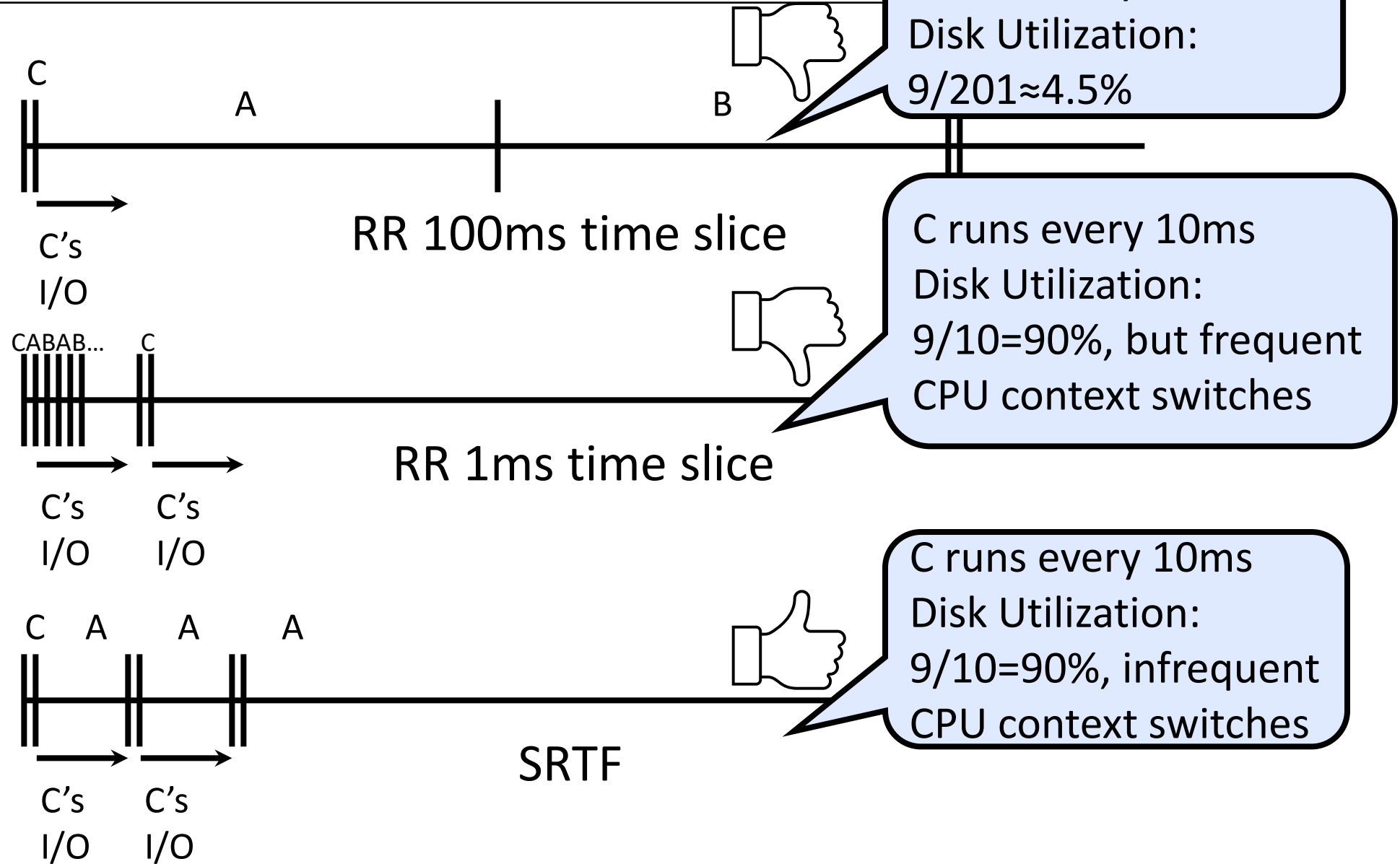   - » SRTF is better, since short jobs are not stuck behind long ones

A or B

C

C's
I/O

C's
I/O

C's
I/O
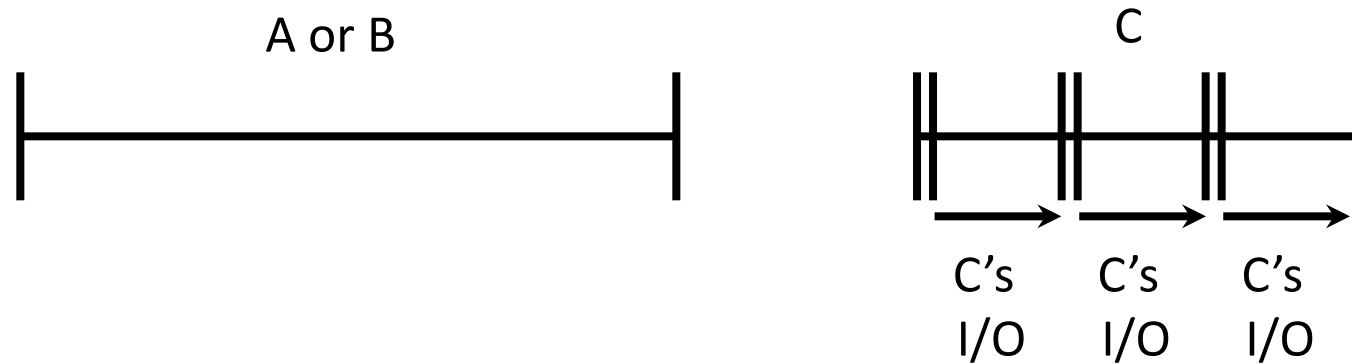
- Three jobs:
  - A, B: both CPU bound, run for a week
    C: I/O bound, runs in a loop of 1ms CPU followed by 9ms disk I/O
  - If each job runs alone without interference, then C uses 90% of disk, A or B uses 100% of CPU

- With FCFS:
  - A and B may arrive and keep CPU busy for two weeks before C is scheduled

- What about RR or SRTF?

C runs every 201ms
Disk Utilization:
9/201≈4.5%

C
A
B

RR 100ms time slice

C's I/O

CABAB... C

RR 1ms time slice

C's I/O   C's I/O

C runs every 10ms
Disk Utilization:
9/10=90%, but frequent
CPU context switches

C   A   A   A

SRTF

C's I/O   C's I/O

C runs every 10ms
Disk Utilization:
9/10=90%, infrequent
CPU context switches

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)
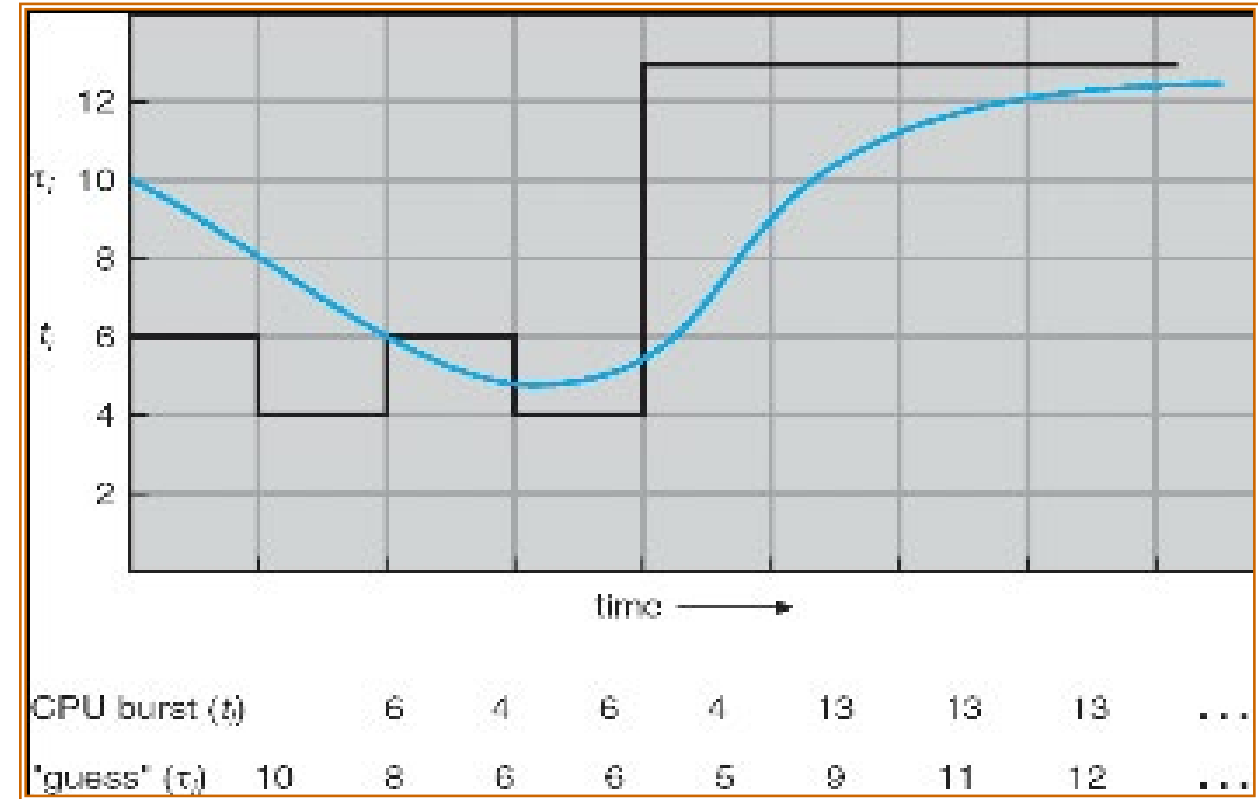
  - Cons: Hard to predict job execution time; Unfair

Predicting the 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 past, likely in future
- Let t_{n-1}, t_{n-2}, t_{n-3}, etc. be previous CPU burst lengths. We need to estimate/predict next burst length $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots)$ based on previous burst lengths.
  - Function f may be one of many different time series estimators (Kalman filters, etc)
- We can use exponential averaging $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1,}$ where t_{n-1}, t_{n-2}, t_{n-3}, etc. are previous CPU burst lengths, and τ_n is the predicted next CPU burst length.
  - t_i = actual burst time of process P_i, i = n, n-1, n-2, …
  - τ_n = predicted burst time for process P_n
  - α is the smoothing factor (0 <= α <=1)
    - α large: fast update of τ_n based on new input.
    - α small: slow update of τ_n based on new input.

Predicting the Length of the Next CPU Burst: Example

- Computing $\tau_n = \alpha t_{n-1}+(1-\alpha)\tau_{n-1}$ with initial guess $\tau_0 = 10$. Assume $\alpha=0.5$.

- $\tau_1 = \alpha t_0+(1-\alpha)\tau_0=0.5*6 + 0.5*10 = 8$

- $\tau_2 = \alpha t_1+(1-\alpha)\tau_1=0.5*4 + 0.5*8 = 6$

- $\tau_3 = \alpha t_2+(1-\alpha)\tau_2=0.5*6 + 0.5*6 = 6$

- $\tau_4 = \alpha t_3+(1-\alpha)\tau_3=0.5*4 + 0.5*6 = 5$

- $\tau_5 = \alpha t_4+(1-\alpha)\tau_4=0.5*13 + 0.5*5 = 9$

- $\tau_6 = \alpha t_5+(1-\alpha)\tau_5=0.5*13 + 0.5*9 = 11$

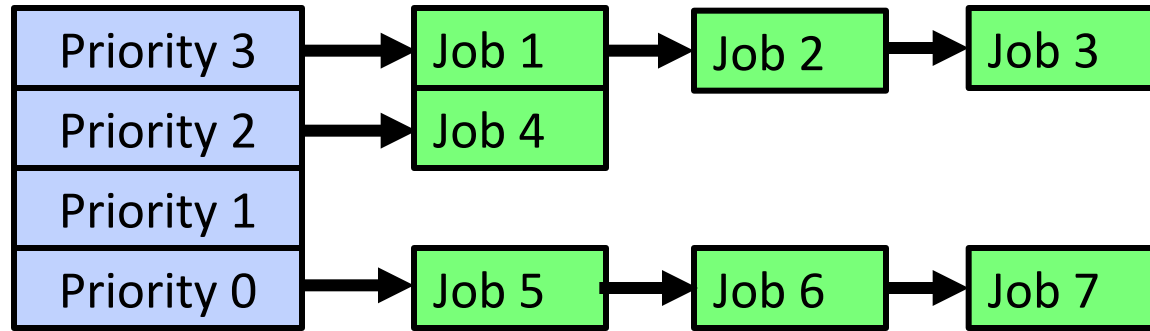- $\tau_7 = \alpha t_6+(1-\alpha)\tau_6=0.5*13 + 0.5*11 = 12$

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

Comparison Chart

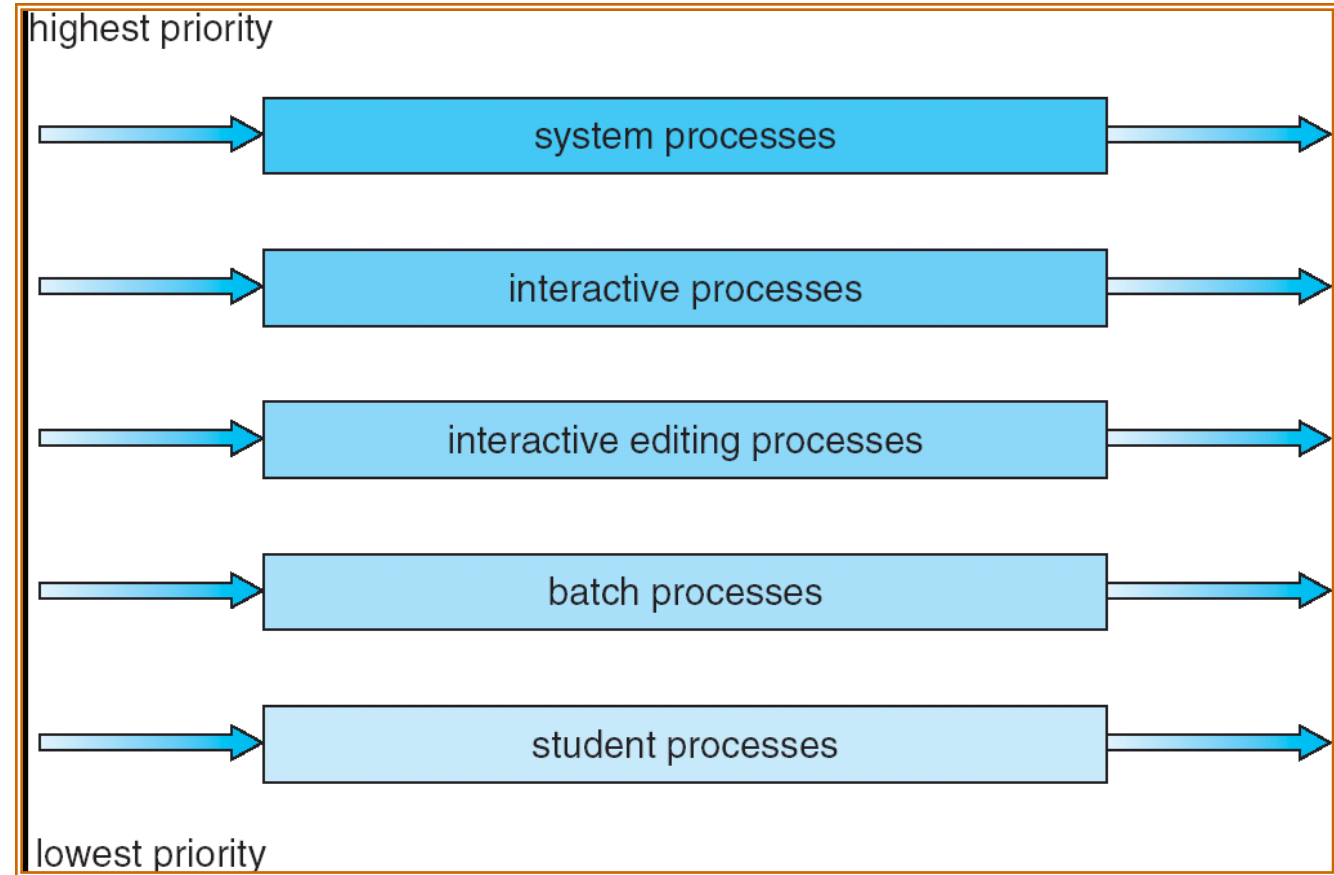| Property	FCFS	SJF	STCF	RR
Optimize Average Response Time		✔	✔	
Prevent Starvation	✔			✔
Prevent Convoy Effect			✔	✔
No Need to Predict Exec Time	✔			✔

- 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
  - Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next: 2ms, next: 4ms, etc)
- Scheduling between the queues
  - Fixed priority, e.g., serve all from foreground queue, then from background queue

highest priority

system processes

interactive processes

interactive editing processes

batch processes

student processes

lowest priority

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)
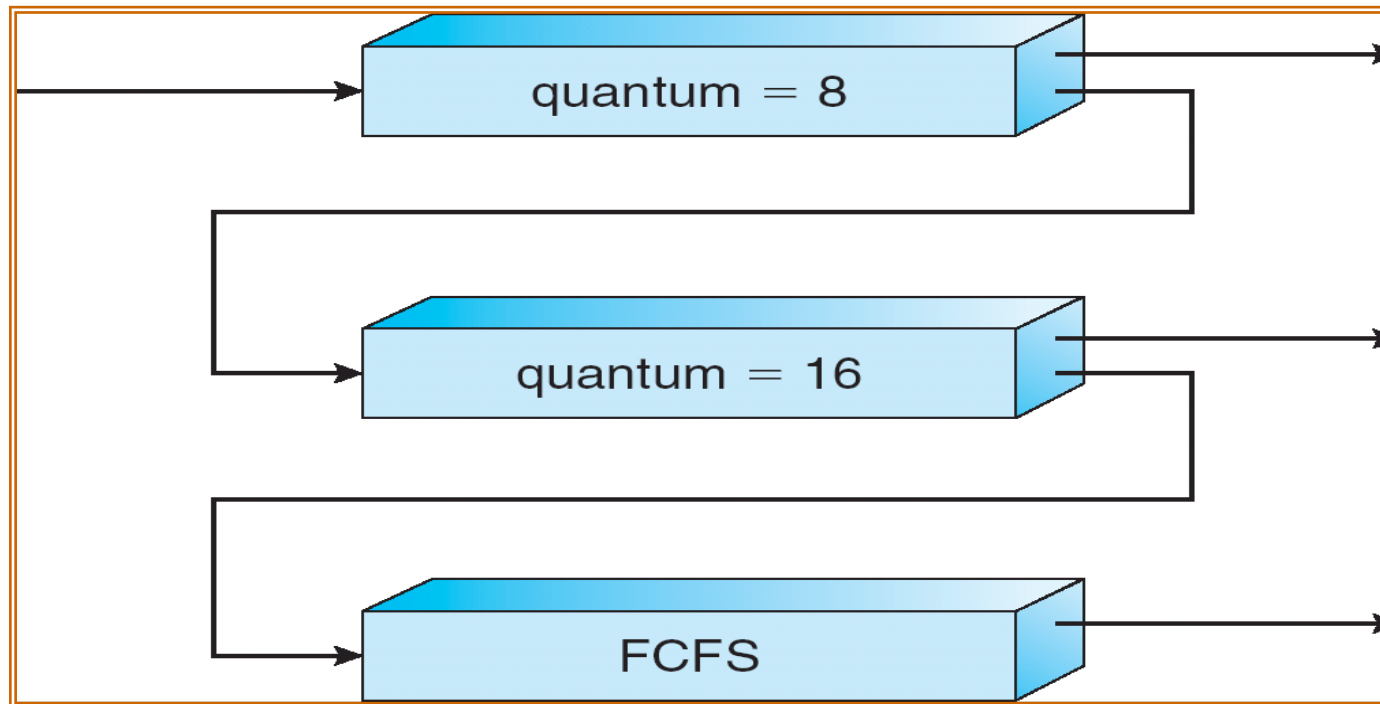
Multi-Level Feedback Queue Scheduling Discussions

- 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