CS 310   Operating Systems

Lecture 24 Scheduling – FIFO, SJF, SRTF

Ravi Mittal
IIT Goa

In this lecture we will study

- Scheduling in Batch System
    - FIFO
    - SJF
    - SRTF

Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - CS162, Operating System and Systems Programming, University of California, Berkeley
  - Book: Modern Operating Systems, Andrew Tenenbaum, and Herbert Bos, 4th Edition, Pearson

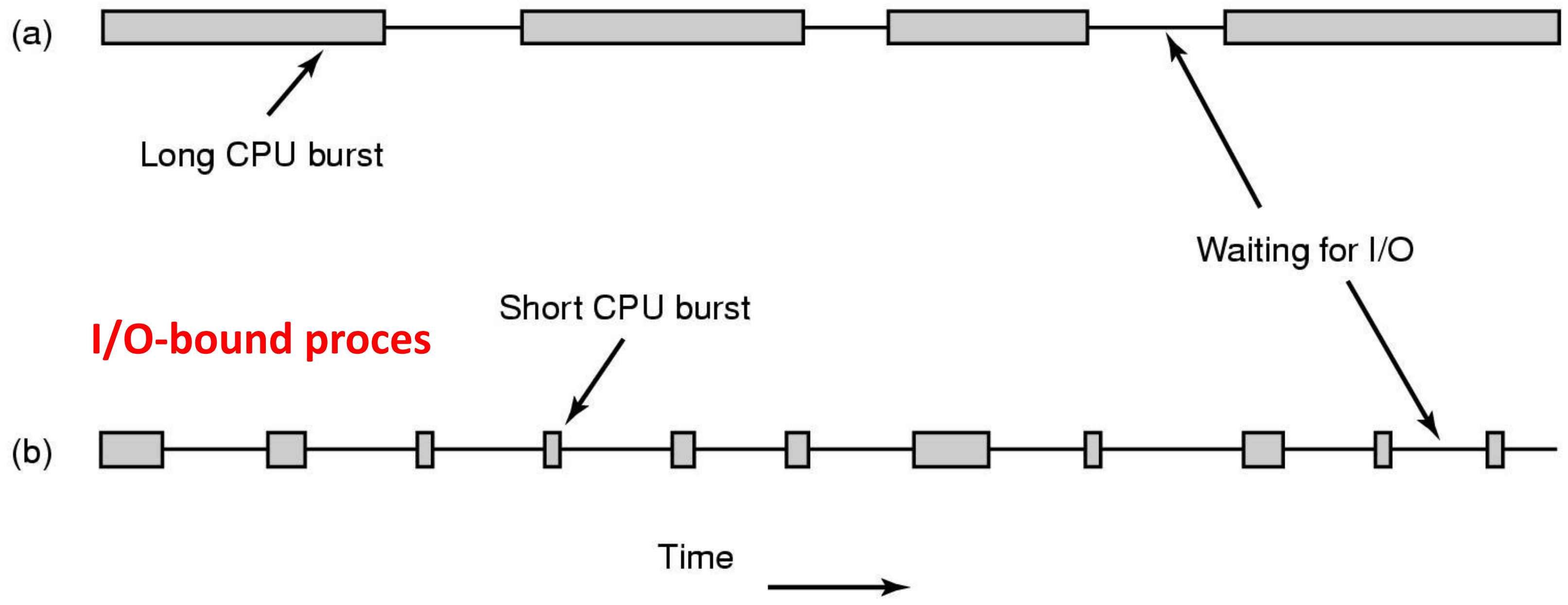
Reading

- Book: Modern Operating Systems, Andrew Tenenbaum, and Herbert Bos, 4th Edition, Pearson
  - Chapter 2

Last Class

CPU Bursts

CPU-bound process



I/O-bound proces

- As processors become faster, processes tend to become more I/O bound
  - Why?
  - CPU is becoming faster than the I?O

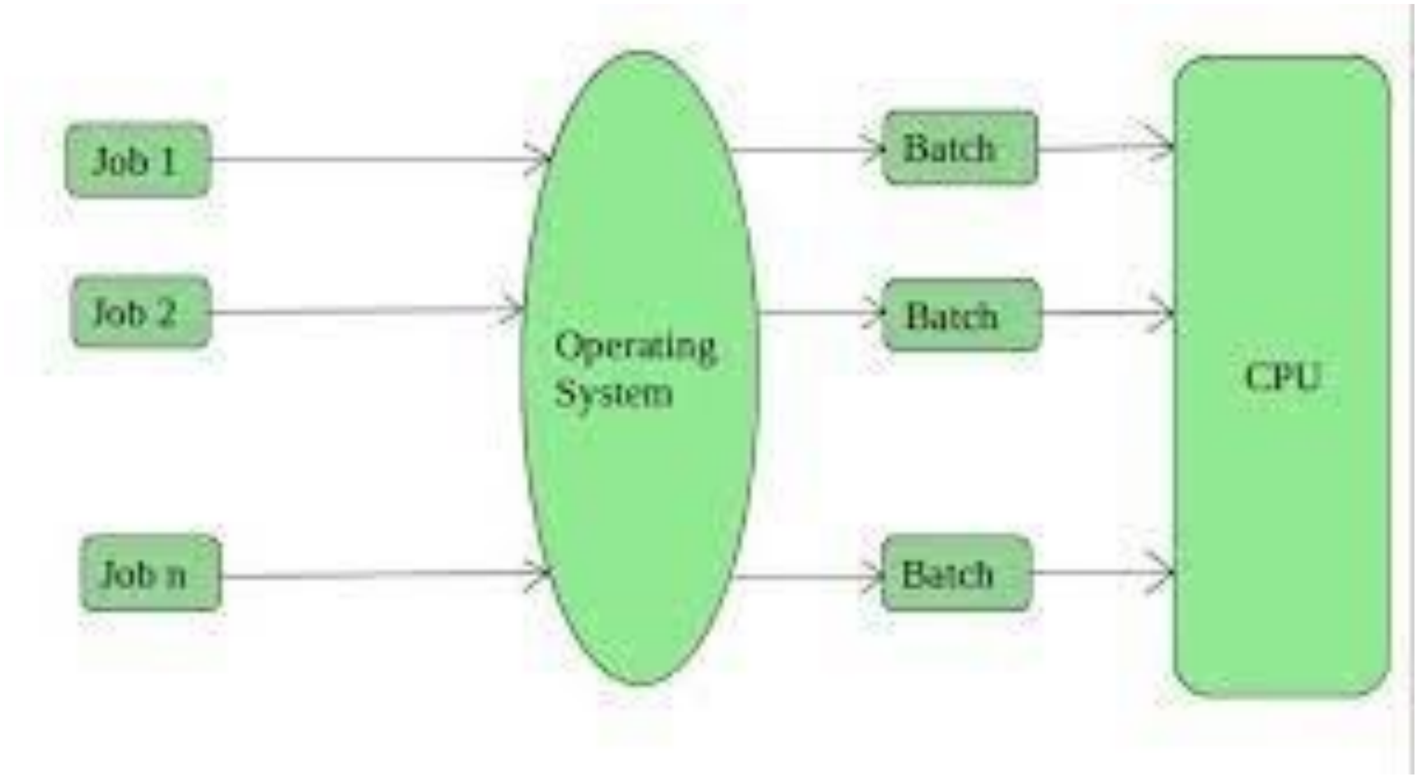
Non-preemptive vs Preemptive scheduling

- Non-preemptive Scheduling Algorithm
  - Picks up a process to run
  - Let the process run until it blocks (for I/O or waiting for another process) or voluntarily releases the CPU
  - A process may run for hours; it will not be forcibly suspended
    - No scheduling decisions are made during clock interrupts
- Preemptive Scheduling
  - Picks up a process to run
  - lets the process run for a maximum of some fixed time
  - At the end of time period, timer interrupt occurs
  - In Kernel mode, scheduler picks up another ready process to run

Scheduling Algorithms - types

- Different environment require different scheduling algorithms
  - Different applications have different goals □ appropriate scheduling algorithms
- Categories of Scheduling Algorithms
  - Batch
  - Interactive
  - Real time (deadlines)

Batch System



Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Time between issuing a command and getting result
- Maximize Throughput
  - Maximize operations (or jobs) per second
- Minimize Turnaround time
  - Average elapsed time – primarily for batch system
- Fairness
  - Share CPU among users in some equitable way

Scheduling in Batch Systems

Scheduling in Batch Systems

- First-come First-served (FCFS)
- Shortest Job First (SJF)
- Shortest Remaining Time First (STRF)
    - Preemptive version of SJF

First Come First Served

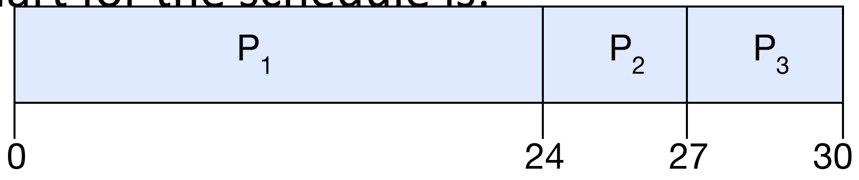
First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO) or "Run until done"
    - In Batch systems, FCFS meant one program scheduled until done (including I/O)
    - In interactive system, means keep CPU until thread blocks

- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

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



  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17
  - Average Completion time: (24 + 27 + 30)/3 = 27

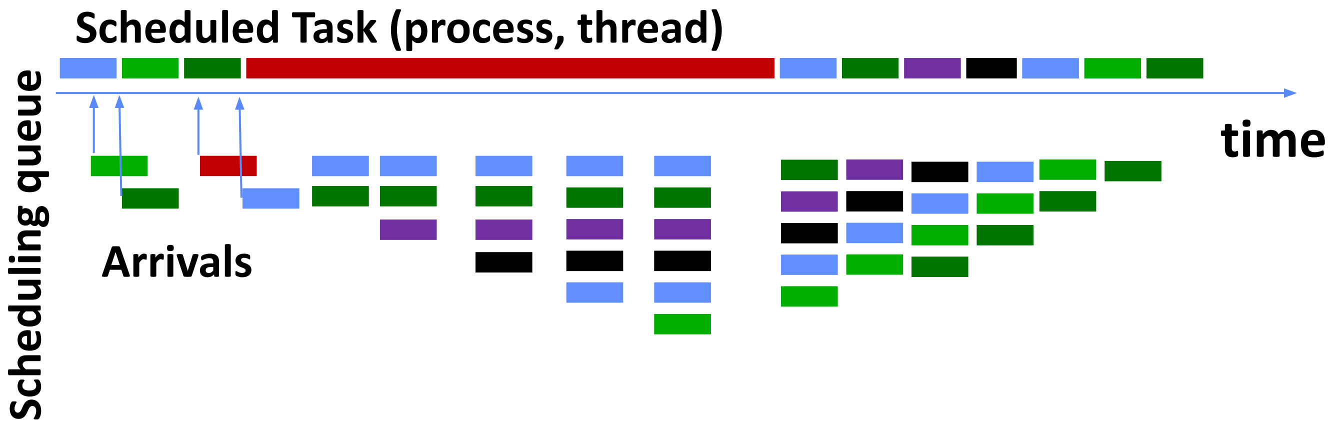
First-Come, First-Served (FCFS) Scheduling

- Advantages
  - Simple algorithm
  - Easy to implement

- Disadvantage
  - Convoy Effect

FCFS: Convoy effect

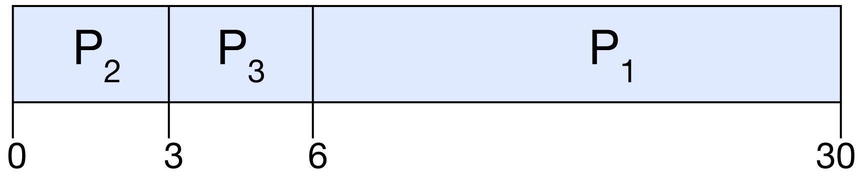
Short process stuck behind long process



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

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P2 , P3 , P1
    Now, the Gantt chart for the schedule is:



  - Waiting time for P1 = 6; P2 = 0; P3 = 3
  - Average waiting time: (6 + 0 + 3)/3 = 3
  - Average Completion time: (3 + 6 + 30)/3 = 13
- In second case:
  - Average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)
- FCFS Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
  - Non-preemptive (-)

Disadvantages of 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…

Shortest Job First

Shortest Job First (SJF)

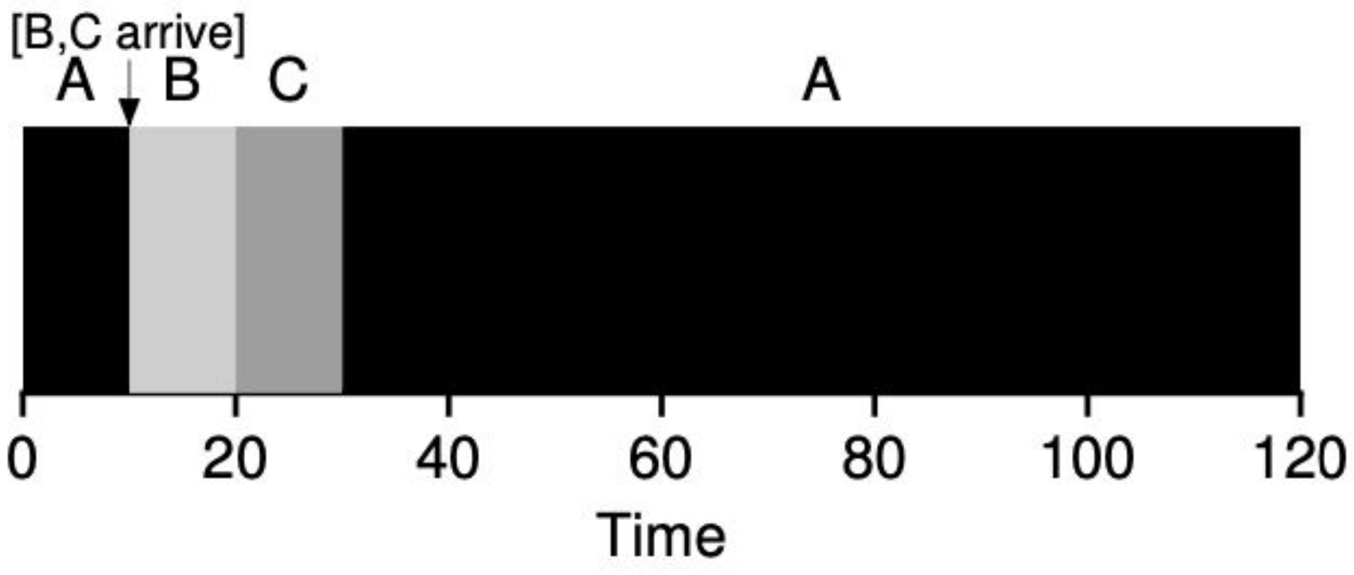
- Non-preemptive
- Run whatever job has least amount of computation to do
  - Shortest job first scheduling runs a process to completion before running the next one
- Sometimes called "Shortest Time to Completion First" (STCF)
- Need to know run times in advance
- Provably optimal
  - 4 jobs with runs times of a,b,c,d
  - First finishes at a, second at a+b,third at a+b+c, last at a+b+c+d
  - Mean turnaround time is (4a+3b+2c+d)/4
  - Smallest time has to come first to minimize the mean turnaround time

Shortest Remaining Time First

Shortest Remaining Time First (SRTF)

- Preemptive version of Shortest job first
- If 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)
- The queue of jobs is sorted by estimated job length so that short programs get to run first and not be held up by long ones
- Both SJF and SRTF:
  - These can be applied to whole program or current CPU burst
    - Idea is to get short jobs out of the system
    - Big effect on short jobs, only small effect on long ones
    - Result is better average response time

Shortest Remaining Time First (SRTF)



- Execution time of A = 100, B = 10, C =10 seconds
- A will be preempted when jobs B and C arrive
- Average Turn Around Time = 50 seconds

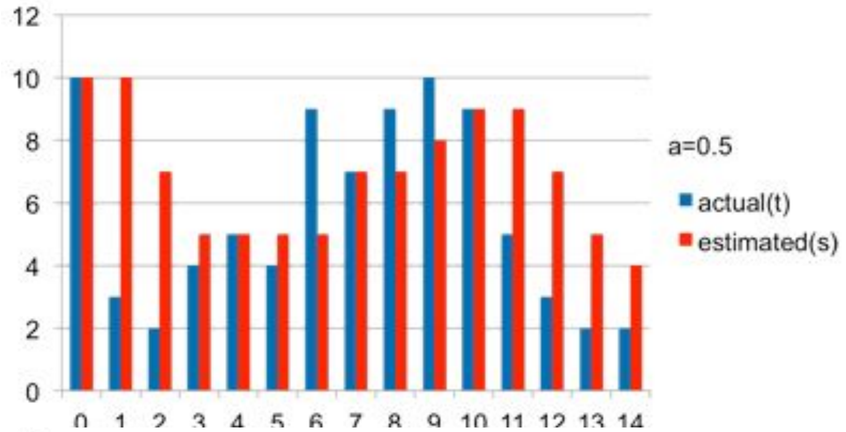
Estimating future CPU Burst time

- The SRTF algorithm can be applied to whole program or current CPU burst
- Sorting based on future burst time?
  - How do we know it ?
- Solution:
  - Predict future burst based on the past history
  - Use an estimator function on previous bursts:
    Let tn-1, tn-2, tn-3, etc. be previous CPU burst lengths.
    Estimate next burst τn = f(tn-1, tn-2, tn-3, …)
  - For example: Exponential Averaging
    - $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$ with $(0 < \alpha \leq 1)$, where
      - τn : predicted size of the *n*th CPU burst
      - tn-1 : the measured time of the (n-1)th burst
      - α : a weighing factor

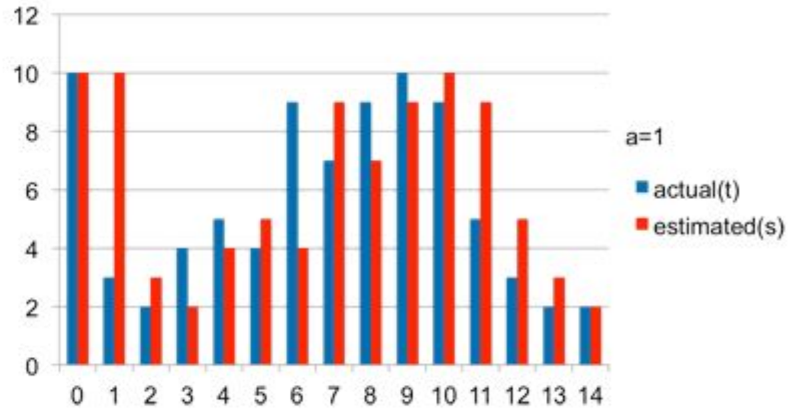
Estimating future CPU Burst time

- Weighing factor α can be adjusted based on how much to weigh past history versus last observation
- If $\alpha = 1$ then only the last observation of the CPU burst period counts
- If $\alpha = ½$ then the last observation has as much weight as the historical weight

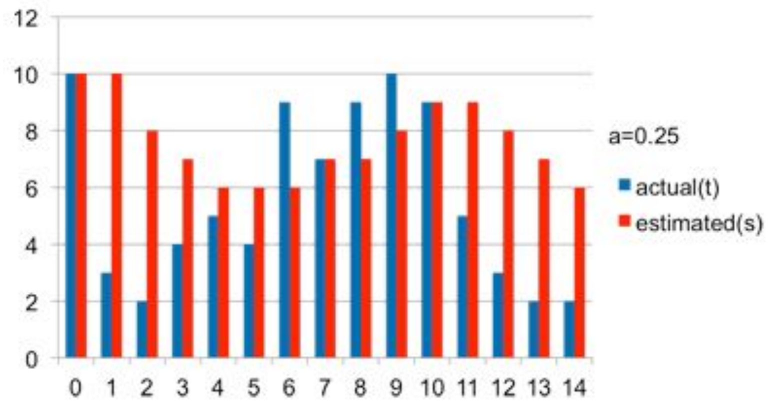
Exponential Average with α = 0.5, 1, 0.25



Exponential Average (α =0.5)



Exponential Average (α = 1)



Exponential Average (α = 0.25)

Advantages and Disadvantages of SRTF

- Advantages
  - This scheduling is optimal in that it always produces the lowest average response time
  - Processes with short CPU bursts are given priority and hence run quickly
- Disadvantages
  - Long-burst (CPU-intensive) processes are hurt with a long average waiting time
  - In fact, if short-burst processes are always available to run, the long-burst ones may never get scheduled
    - Starvation
  - the effectiveness of meeting the scheduling criteria relies on our ability to estimate the length of the next CPU burst

Useful metrics

- Waiting time for process *P:* time before *P* got scheduled
- Average waiting time: Average of all processes’ wait time.
- Completion time (response time): Waiting time + Run time.
- Average completion time (response time): Average of all processes' completion time

Lecture Summary

- In Batch Systems, the following scheduling algorithms are used
  - First Come First Served
    - Simple, Convoy effect
  - Shortest Job First
    - Need to know job length in advance
  - Shortest Remaining Time First
    - Preemptive
    - Performs well - lowest average response time
    - Long burst job may suffer

Scheduling in Interactive Systems

Scheduling in Interactive Systems

- Round Robin
- Priority

Round Robin

Round Robin (RR) Scheduling

- Uses Preemption!
- 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 and time quantum is q
  - Each process gets 1/n of the CPU time
  - In chunks of at most q time units
  - No process waits more than (n-1)q time units

The magic number

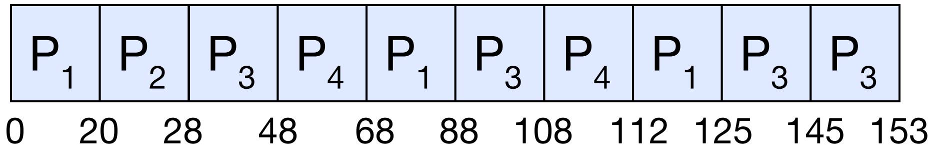
- What should q be?
  - *q* large ⇒ FCFS
  - *q* small ⇒ Interleaved
  - *q* must be large with respect to context switch, otherwise overhead is too high (all overhead)

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

  - The Gantt chart is:



  - Waiting time for P_1= 0 + (68-20)+(112-88)=72
    P_2=(20-0)=20
    P_3=(28-0)+(88-48)+(125-108) + 0 =85
    P_4=(48-0)+(108-68)=88

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

  - Average completion time = (125+28+153+112)/4 = 104½

Round-Robin Quantum

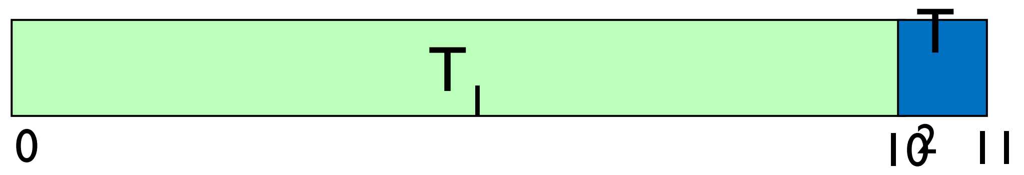
- Assume that context switch overhead is 0
- What happens when we *decrease q*?

1. Avg. response time always **decreases** or **stays the same**
2. Avg. response time always **increases** or **stays the same**
3. Avg. response time can **increase**, **decrease**, or **stays the same**

Note: Response time: Time between issuing a command and getting result

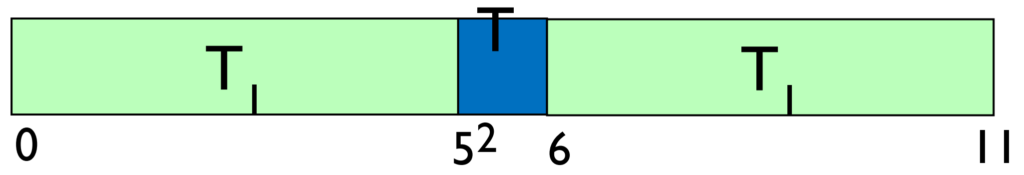
Decrease Response Time

- T_1: Burst Length 10
- T_2: Burst Length 1
- Q = 10



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

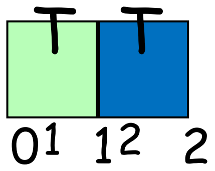
- Q = 5



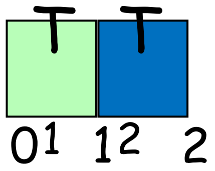
  - Average Response Time = (6 + 11)/2 = 8.5

Same Response Time

- T1: Burst Length 1
- T2: Burst Length 1
- *Q* = 10



- Average Response Time = (1 + 2)/2 = 1.5
- *Q* = 1

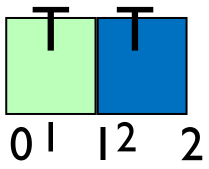


- Average Response Time = (1 + 2)/2 = 1.5

Increase Response Time

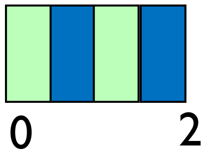
- T1: Burst Length 1
- T2: Burst Length 1

- Q = 1



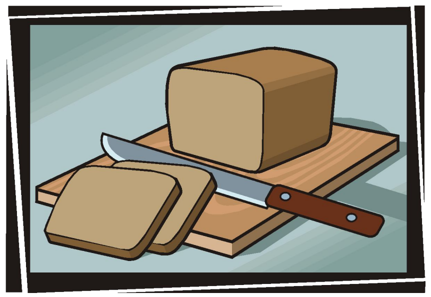
  - Average Response Time = (1 + 2)/2 = 1.5

- Q = 0.5



  - Average Response Time = (1.5 + 2)/2 = 1.75

Round-Robin Scheduling: Discussion



- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if time slice too small?
    - Throughput suffers!
- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - Worked ok when UNIX was used by one or two people
  - 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 Round Robin

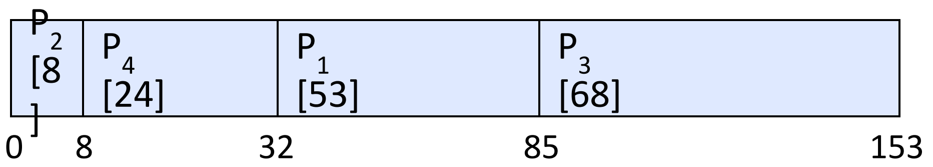
- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
  - RR scheduler quantum of 1s
  - All jobs start at the same time
- Completion Times:

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 completion(response) time is much worse under RR!
    - Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
  - Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quantum

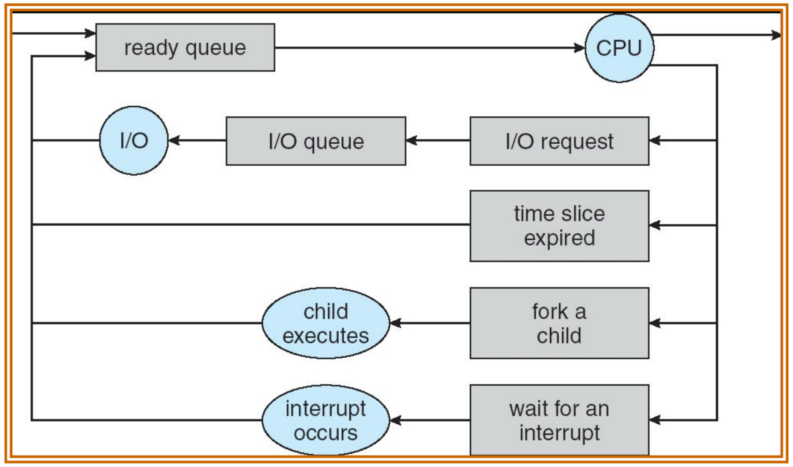
Best FCFS:



	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

How to Implement RR in the Kernel?

- FIFO Queue, as in FCFS
- But preempt job after quantum expires, and send it to the back of the queue
    - How? Timer interrupt!



Priority Scheduling

Round Robin Scheduling – Treat all equal

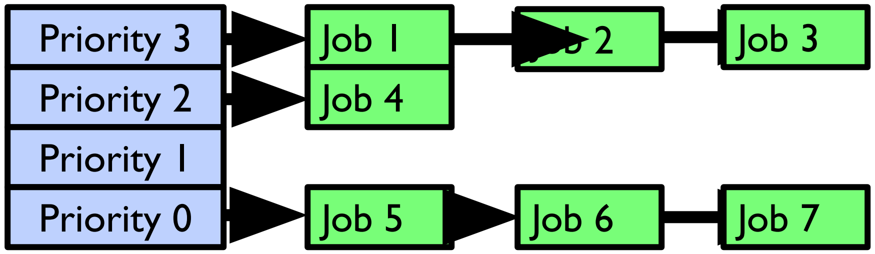
- Round robin scheduling assumes that all processes are equally important
- This is not always practical solution
- If someone wants
  - Long CPU intensive process to run with lower priority than interactive processes
  - Round Robin is not a good solution
- What about giving higher priority to Sys Admin processes?

Priority Scheduling

- Each process is assigned a priority (a number)
- the scheduler simply picks the highest priority (ready) process to run
- In preemptive scheduling , a process is preempted whenever a higher priority process is available in the ready queue

- In RR or other Scheduling there is one queue of jobs.
  - Scheduler selects the highest priority process
  - It takes O(n) time
- Solution
  - Separate queue for each distinct priority
  - Schedule the process in the highest priority queue

Multilevel Queue Scheduling – Strict Priority



- Execution Plan
  - Always execute highest-priority runnable jobs to completion
  - Each queue can be processed in RR with some time-quantum
  - A priority is assigned statically to each process, and a process remains in the same queue for the duration of the run time
- Problems:
  - Starvation:
    - Lower priority jobs don't get to run because higher priority jobs
  - Deadlock: Priority Inversion
    - Happens when low priority task holds a lock needed by high-priority task

Multi level queue: Strict Priority Scheduling: Fairness Issues

- Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    - long running jobs may never get CPU
    - When you shut down machine, you may find 10-year-old job still waiting to run ??

- Must give long-running jobs (with low priority) a fraction of the CPU even when there are higher priority jobs to run

How to implement Fairness

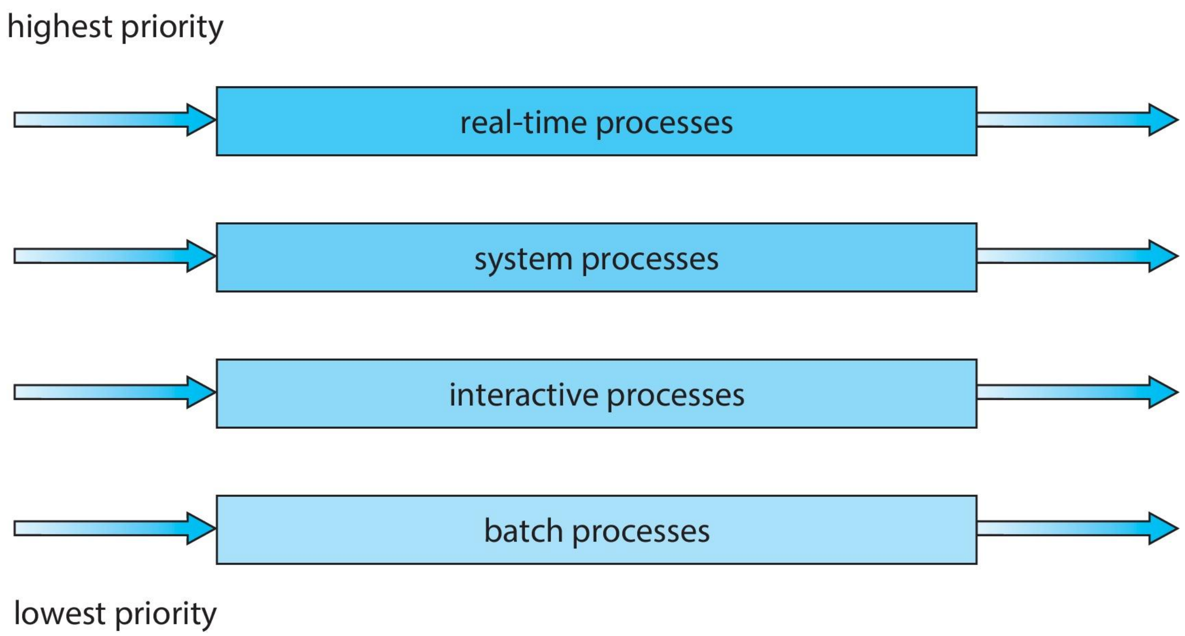
- Could give each queue some fraction of the CPU
- Could increase priority of jobs that don't get service
  - what rate should you increase priorities?
  - And, as system gets overloaded, no job gets CPU time, so everyone increases in priority

  ⇒ Interactive jobs suffer

⇒ Multilevel Feedback Queue

Multilevel Queue Scheduling (for different process types)

- Prioritization based upon process type

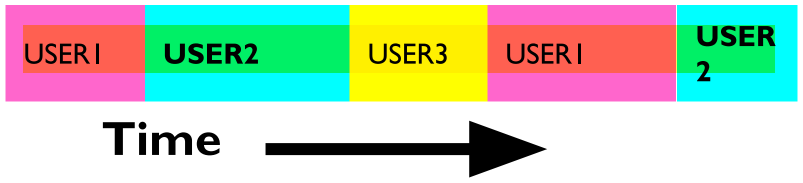


Types of Resource

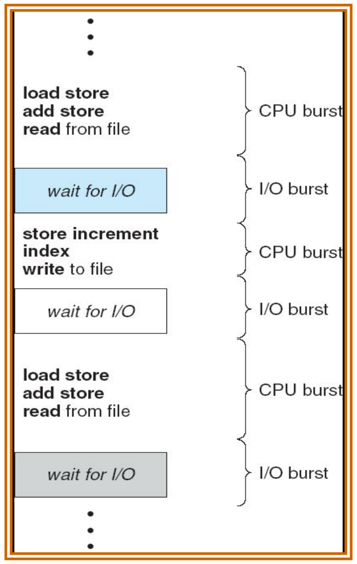
- Preemptible
  - OS can take resource away, use it for something else, and give it back later
    - E.g., CPU
- Non-preemptible
  - Once given resource, it can’t be reused until voluntarily relinquished
    - E.g., disk space
- Given set of resources and set of requests for the resources, types of resource determines how OS manages it

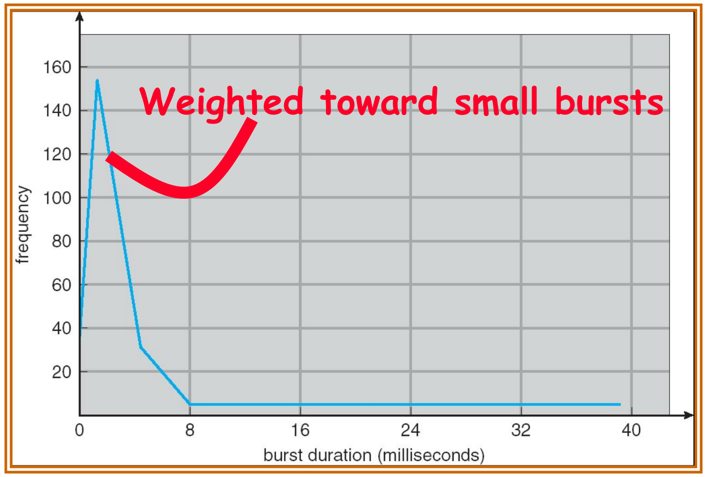
Scheduling Assumptions

- Many implicit assumptions for CPU scheduling:
  - One program per user
  - One thread per program
  - Programs are independent
- 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



Assumption: CPU Bursts





- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use in next CPU burst
    - If a process comes back after I/O wait, it counts as a fresh CPU burst
  - With time-slicing, thread may be forced to give up CPU before finishing current CPU burst

Lecture Summary

- Interactive systems use scheduling that reduce average response time
- Round Robin is the oldest, simplest, fairest, and widely used scheduling algorithm
- In round robin algorithm, proper choice of time quantum is very important
    - If the time quantum is too short, there will too many context switches lower CPU utilization
    - If the time quantum is too big, there will be high values of wait time and response time poor response to short interactive requests
- Priority scheduling
    - Multilevel Queue Scheduling – Strict Priority