Scheduling

David E. Culler

CS162 – Operating Systems and Systems Programming

Lecture 11

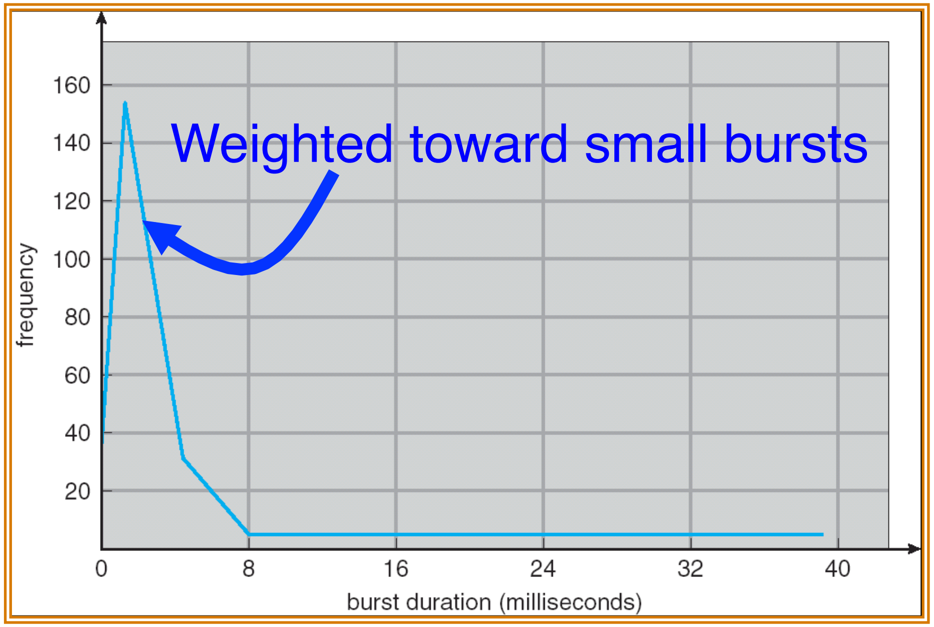
Sept 24, 2014

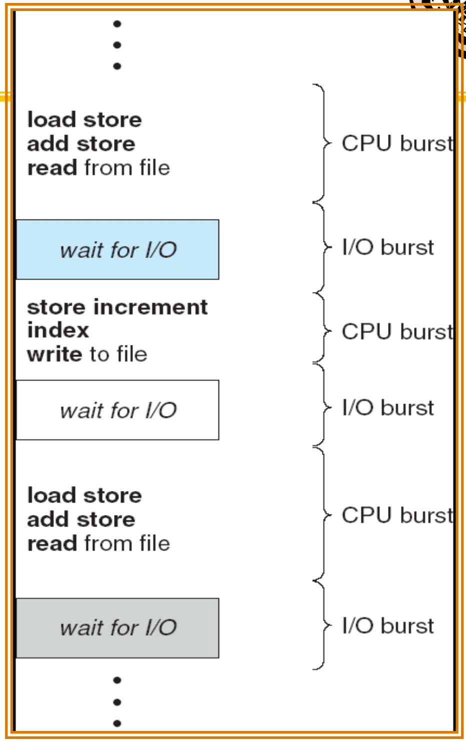
Reading: A&D 7-7.1
HW 2 due 9/26
Proj 1 design review
MT1: 9/29 6:00-7:00

Recall: Objectives

- Introduce the concept of scheduling
- General topic that applies in many context
  - rich theory and practice
- Fundamental trade-offs
  - not a simple find the "best"
  - resolution depends on context
- Ground it in OS context
- Ground implementation in Pintos
- … after synch implementation wrap-up

Recall: CPU Bursts





- 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 by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

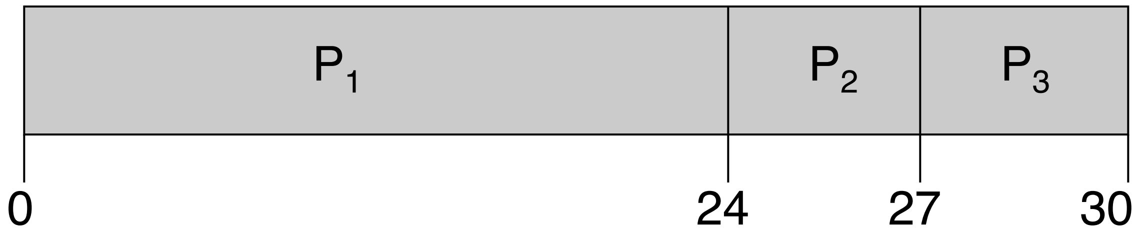
Recall: First-Come, First-Served (FCFS) Scheduling

- 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 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
- *Convoy effect:* short process behind long process

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P_2 , P_3 , P_1
    Now, the Gantt chart for the schedule is:

  - Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
  - 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 (-)
    - Safeway: Getting milk, always stuck behind cart full of small items

Recall: Round Robin (RR)

- 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 Scheme

  - 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 $q \Rightarrow$
    - 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
- Performance
  - q large $\Rightarrow$ FCFS
  - q small $\Rightarrow$ 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	Remaining Time
P_1	53	53
P_2	8	8
P_3	68	68
P_4	24	24

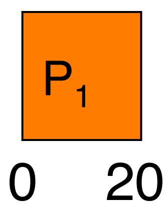
  – The Gantt chart is:

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	33
P_2	8	8
P_3	68	68
P_4	24	24

  – The Gantt chart is:

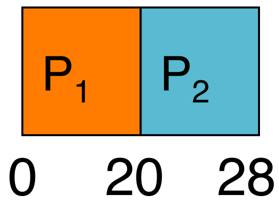


Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	33
P_2	8	0
P_3	68	68
P_4	24	24

  - The Gantt chart is:

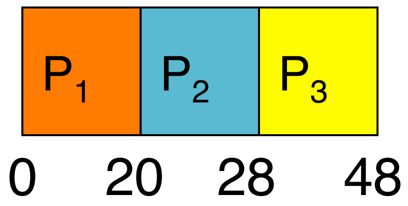


Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	33
P_2	8	0
P_3	68	48
P_4	24	24

  – The Gantt chart is:

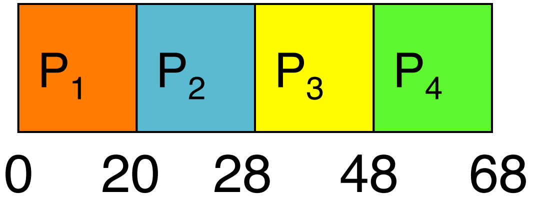


Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	33
P_2	8	0
P_3	68	48
P_4	24	4

  – The Gantt chart is:

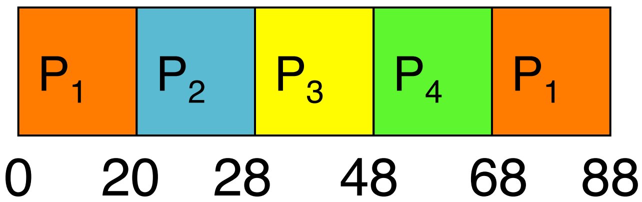


Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	13
P_2	8	0
P_3	68	48
P_4	24	4

  – The Gantt chart is:

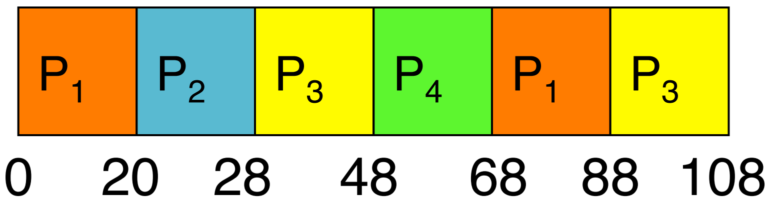


Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	13
P_2	8	0
P_3	68	28
P_4	24	4

  – The Gantt chart is:

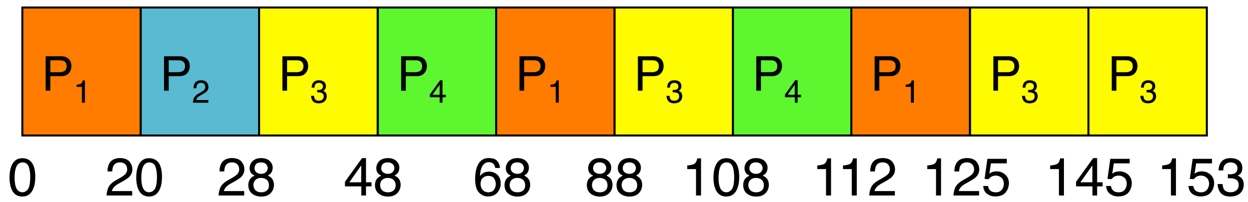


Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	0
P_2	8	0
P_3	68	0
P_4	24	0

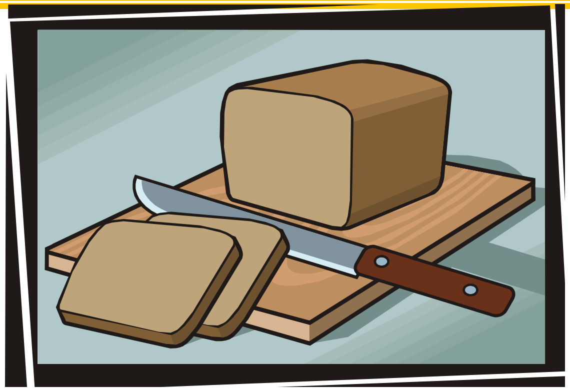
  – The Gantt chart is:



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

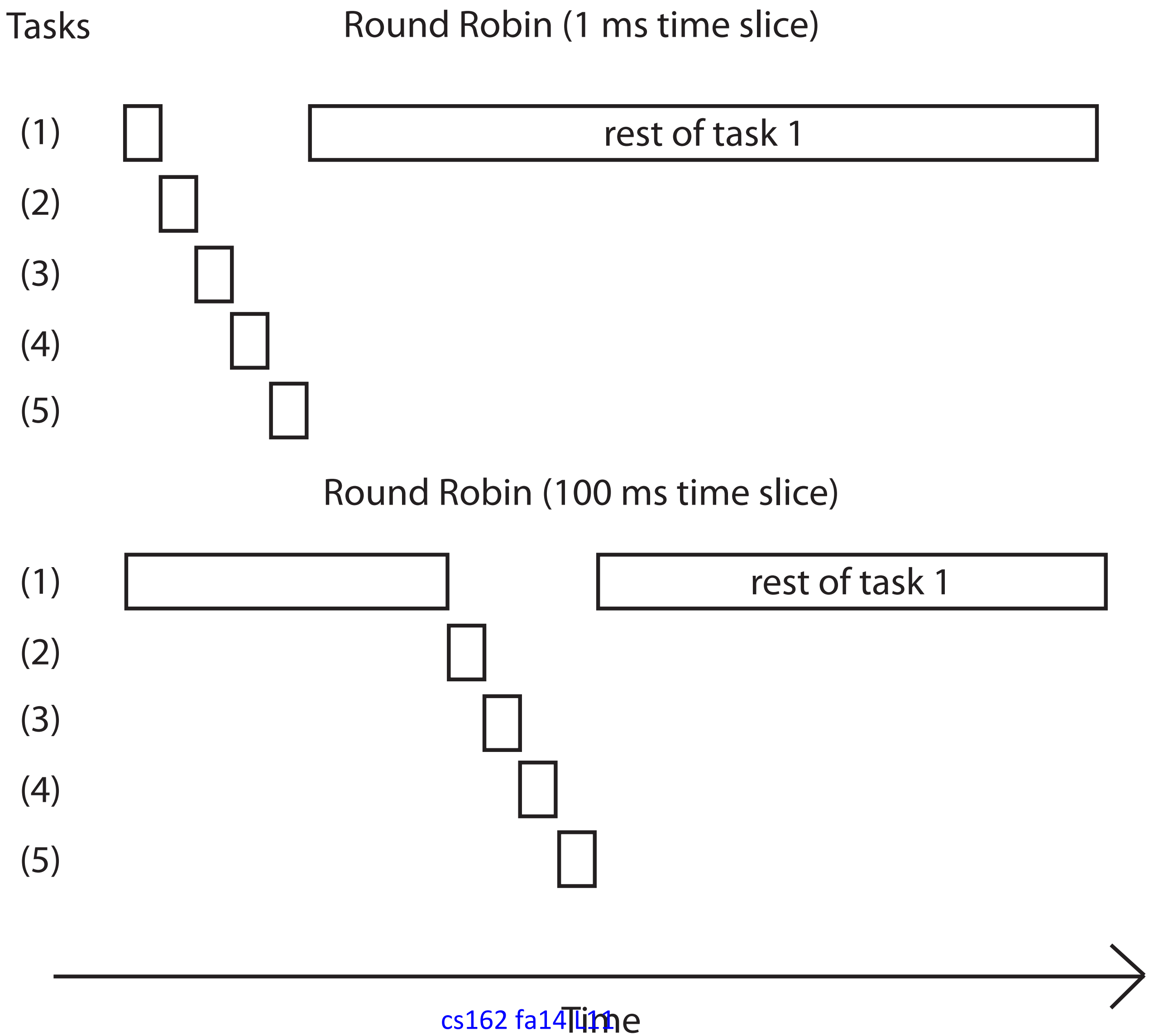
- Thus, Round-Robin Pros and Cons:
  – Better for short jobs, Fair (+)
  – Context-switching time adds up for long jobs (-)

Round-Robin Discussion



- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (∞)?
    - Get back FCFS/FIFO
  - 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.
    - 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

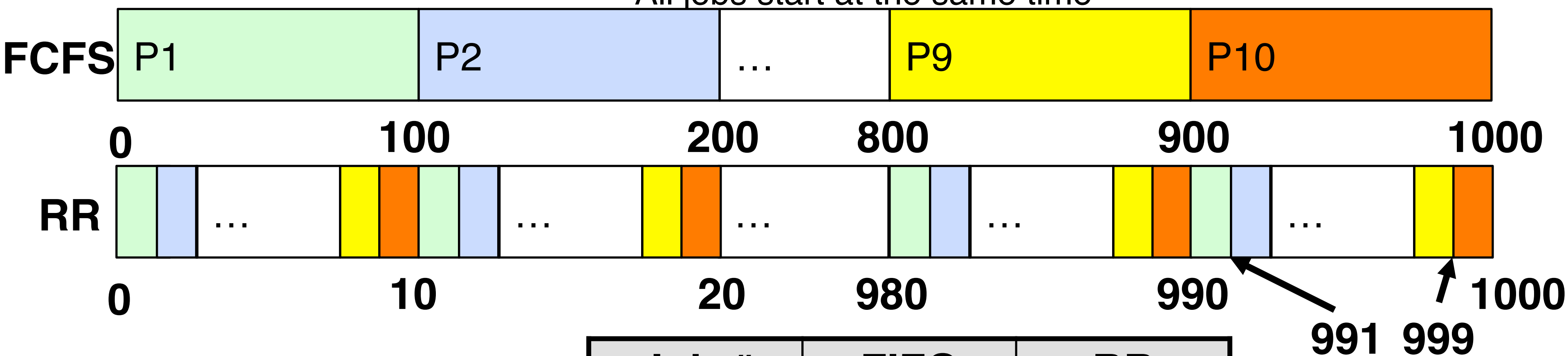
Round Robin Slice



Comparisons between FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example:

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



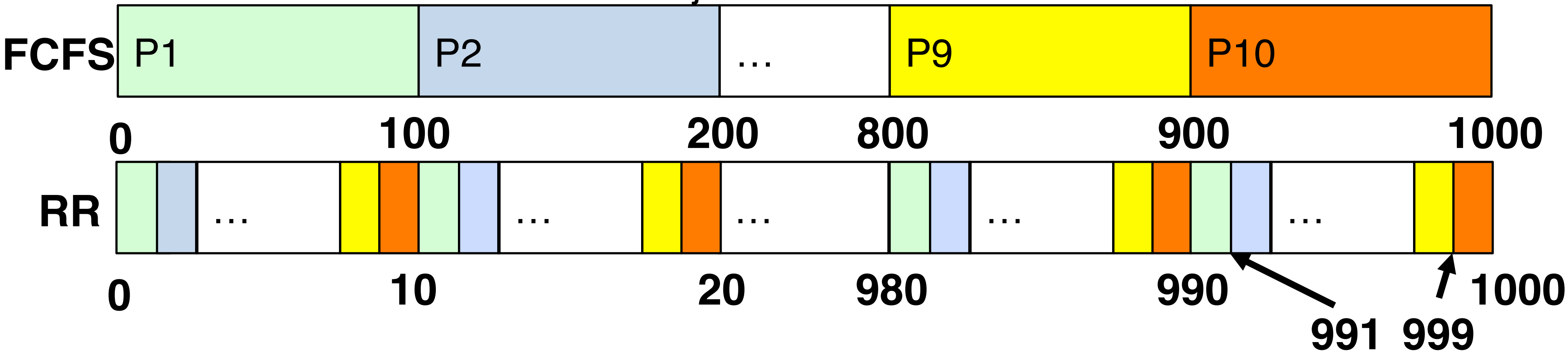
- Completion Times:
- FIFO average 550
- RR average 995.5!

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

Comparisons between FCFS and Round Robin

- ~~Assuming zero-cost context-switching time, is RR~~ always better than FCFS?
- Simple example:

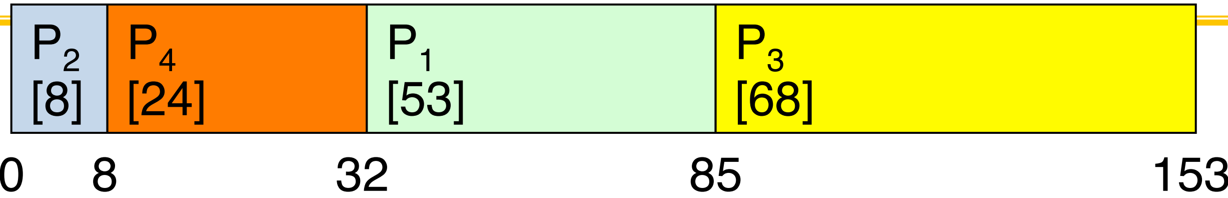
  10 jobs, each takes 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time



- Both RR and FCFS finish at the same time
- Average 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 FCFS
  - Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quantum

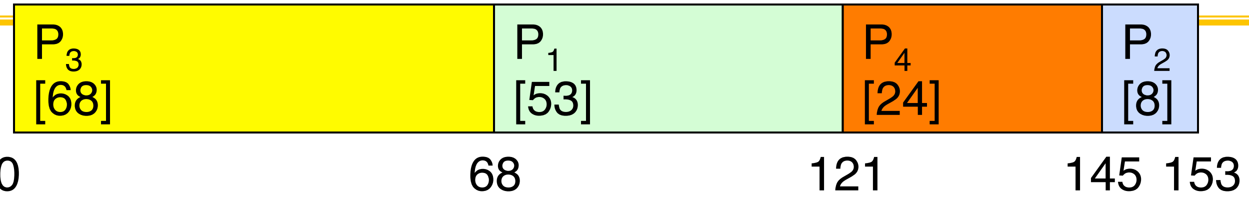
Best FCFS:



	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
Completion Time	Best FCFS	85	8	153	32	69½

Earlier Example with Different Time Quantum

Worst FCFS:



	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Worst FCFS	121	153	68	145	121¾

Earlier Example with Different Time Quantum

Worst FCFS:

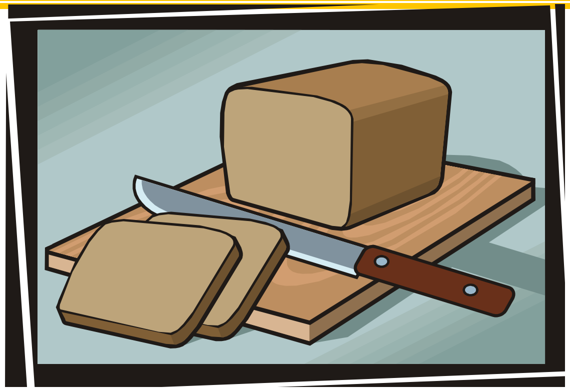
P_3 [68]	P_1 [53]	P_4 [24]	P_2 [8]
0	68	121	145 153

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_1	P_3	P_1	P_3	P_1	P_3	P_3	P_3

0 8 **16** 24 32 40 48 56 64 72 **80** 88 96 104 112 120 128 **133** 141 149 **153**

	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	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

Round-Robin Discussion



- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (∞)?
    - Get back FCFS/FIFO
  - 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.
    - 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

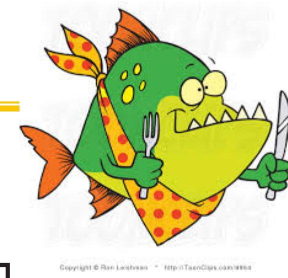
Administrative Break

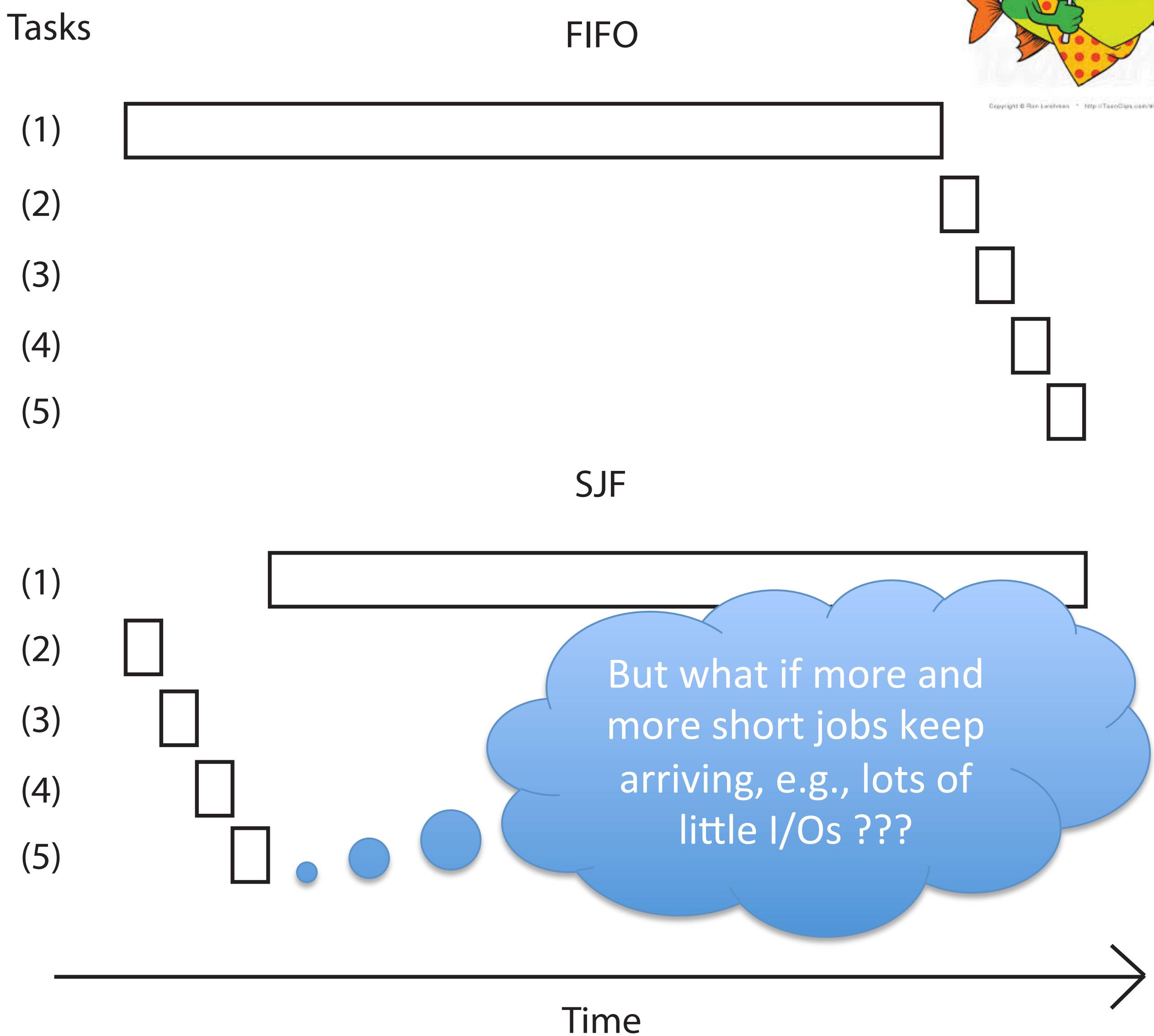
- Survey thanks
- Midterm Monday 6pm
  - 145 DWINELLE (aa – ft)
  - 2040 VALLEY LSB (fu – jl)
  - 2060 VALLEY LSB (jm – ni)
  - review session 1-3:00 pm on Sat 9/26 @100 GPB
- Vote: Q&A Monday ???
- Design review is to help you get a clear path to completion – not a big grading hurdle
- HWs are to help you internalize the concepts
- project test jigs …

What if we Knew the Future?

- Shortest Job First (SJF):
  - Run whatever job has the least amount of computation to do
- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
    - but how do you now???
- 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
- Want a simple approximation to SRTF …

FIFO vs. SJF

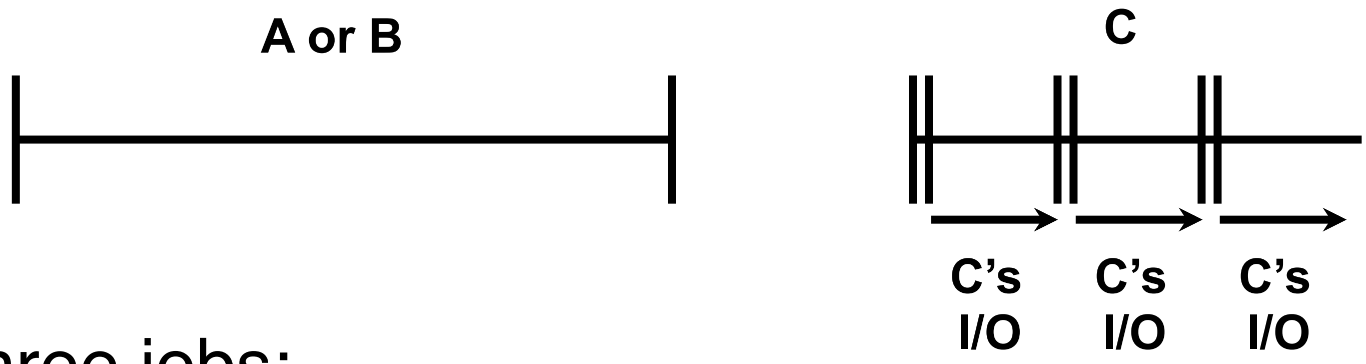



Tasks
FIFO
(1)
(2)
(3)
(4)
(5)
SJF
(1)
(2)
(3)
(4)
(5)
But what if more and more short jobs keep arriving, e.g., lots of little I/Os ???
Time

Discussion

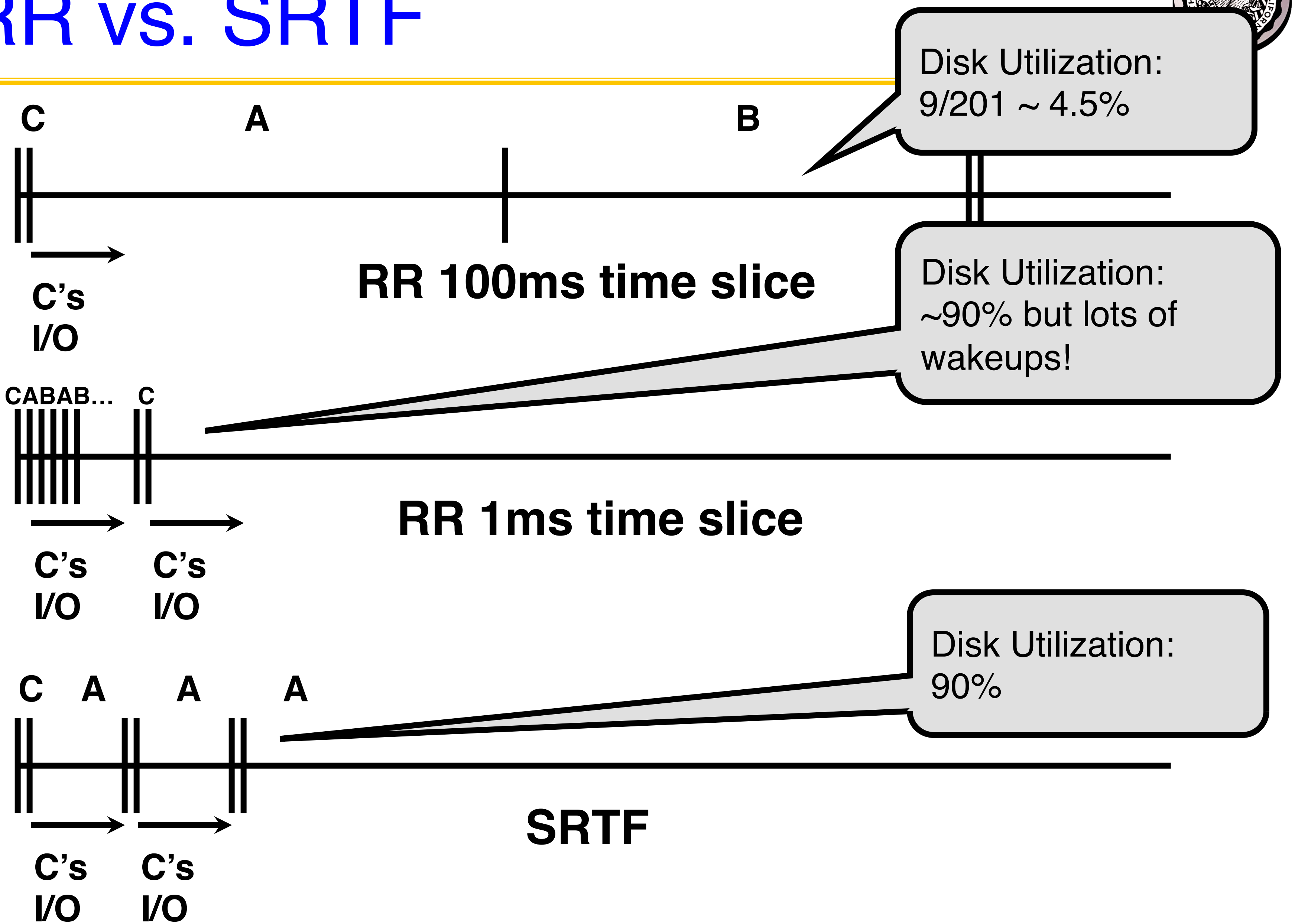
- SJF/SRTF are best at minimizing average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - SJF becomes the same as FCFS (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

Example to illustrate benefits of SRTF



- Three jobs:
  - A,B: CPU bound, each run for a week
    C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B use 100% of the CPU
- With FIFO:
  - Once A or B get in, keep CPU for one week each
- What about RR or SRTF?
  - Easier to see with a timeline

RR vs. SRTF



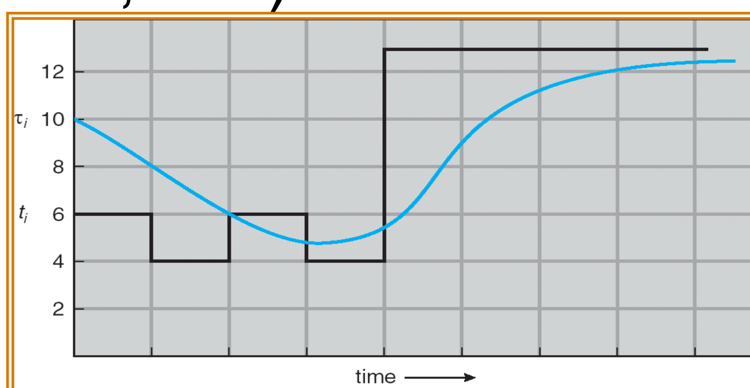
C
A
B
Disk Utilization:
9/201 ~ 4.5%
C's
I/O
RR 100ms time slice
Disk Utilization:
~90% but lots of
wakeups!
CABAB…
C
C's
I/O
C's
I/O
RR 1ms time slice
Disk Utilization:
90%
C
A
A
A
C's
I/O
C's
I/O
SRTF

SRTF Further discussion

- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- 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: even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal => Practical approximations?
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

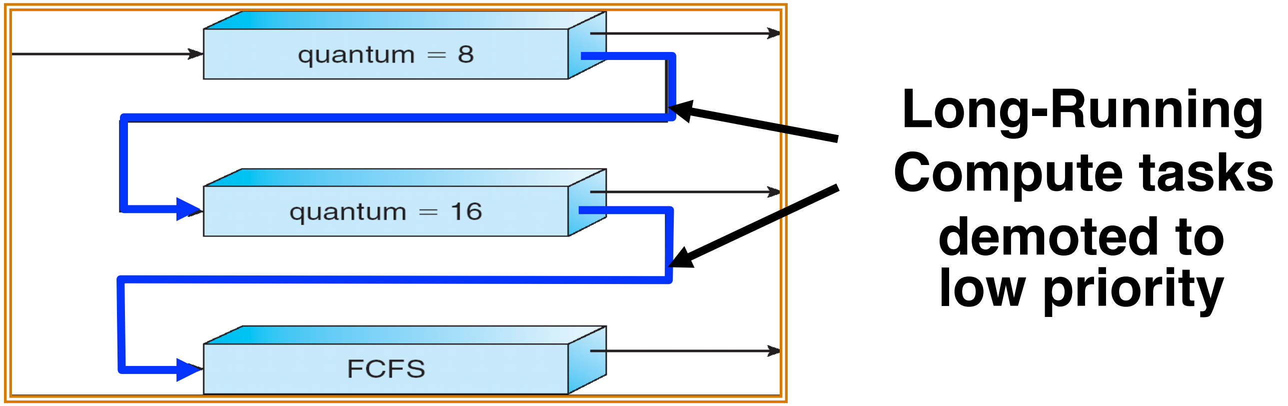
Predicting the Length of the Next CPU Burst

- Adaptive: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc.
  - Works because programs have predictable behavior
    - If program was I/O bound in past, likely in future
    - If computer behavior were random, wouldn't help
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts: Let t_{n-1}, t_{n-2}, t_{n-3}, etc. be previous CPU burst lengths. Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots)$
  - Function f could be one of many different time series estimation schemes (Kalman filters, etc.)
  - Example: Exponential averaging
    $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$
    with $(0 < \alpha \leq 1)$



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

Multi-Level Feedback Scheduling



- Another method for exploiting past behavior
  - First used in Cambridge Time Sharing System (CTSS)
  - Multiple queues, each with different priority
    - Higher priority queues often considered "foreground" tasks
  - Each queue has its own scheduling algorithm
    - e.g., foreground – RR, background – FCFS
    - Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc.)
- Adjust each job's priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or to top)

Scheduling Details

- Result approximates SRTF:
  – CPU bound jobs drop like a rock
  – Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  – Fixed priority scheduling:
    - Serve all from highest priority, then next priority, etc.
  – Time slice:
    - Each queue gets a certain amount of CPU time
    - e.g., 70% to highest, 20% next, 10% lowest

Scheduling Fairness

- What about fairness?
  - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    - Long running jobs may never get CPU
    - In Multics, shut down machine, found 10-year-old job
  - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
  - Tradeoff: fairness gained by hurting average response time!
- How to implement fairness?
  - Could give each queue some fraction of the CPU
    - What if one long-running job and 100 short-running ones?
    - Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
  - Could increase priority of jobs that don't get service
    - What is done in UNIX
    - This is ad hoc—what rate should you increase priorities?

Lottery Scheduling

- Yet another alternative: Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
  - To approximate SRTF, short running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Lottery Scheduling Example

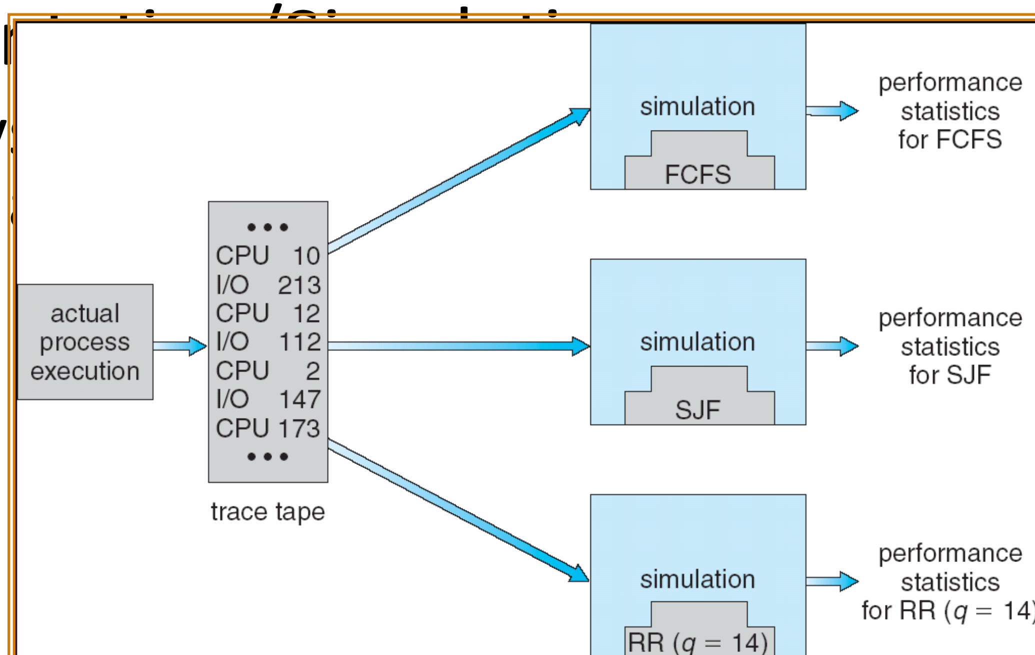
- Lottery Scheduling Example
  - Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

  - What if too many short jobs to give reasonable response time?
    - In UNIX, if load average is 100, hard to make progress
    - One approach: log some user out

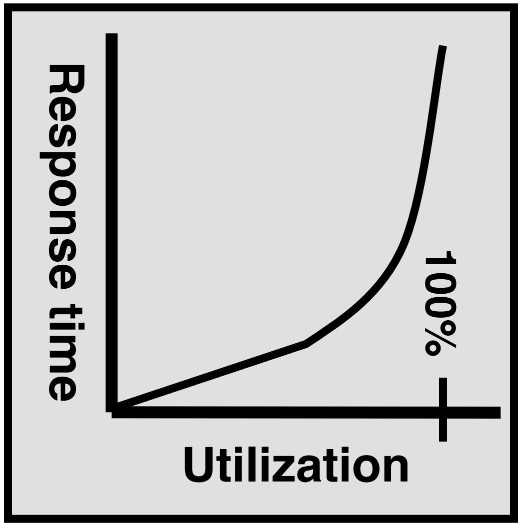
How to Evaluate a Scheduling algorithm?

- Deterministic modeling
  - Takes a predetermined workload and compute the performance of each algorithm for that workload
- Queuing models
  - Mathematical approach for handling stochastic workloads
- Implemen
  - Build sy ns to be run against .



A Final Word On Scheduling

- When do the details of the scheduling policy and fairness really matter?
  - When there aren't enough resources to go around
- When should you simply buy a faster computer?
  - (Or network link, or expanded highway, or …)
  - One approach: Buy it when it will pay for itself in improved response time
    - Assuming you're paying for worse response time in reduced productivity, customer angst, etc…
    - Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization⇒100%



- An interesting implication of this curve:
  - Most scheduling algorithms work fine in the "linear" portion of the load curve, fail otherwise
  - Argues for buying a faster X when hit "knee" of curve

Scheduling Summary

- Scheduling: selecting a process from the ready queue and allocating the CPU to it
- FCFS Scheduling:
  – Run threads to completion in order of submission
  – Pros: Simple (+)
  – Cons: Short jobs 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 (+)
  – Cons: Poor when jobs are same length (-)
- Shortest Remaining Time First (SRTF)
  – Run whatever job has the least remaining amount of computation to do !!!
  – Pros: Optimal (average response time)
  – Cons: Hard to predict future, Unfair

Summary (cont'd)

- Multi-Level Feedback Scheduling:
  – Multiple queues of different priorities
  – Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
- Lottery Scheduling:
  – Give each thread a number of tokens (short tasks $\Rightarrow$ more tokens)
  – Reserve a minimum number of tokens for every thread to ensure forward progress/fairness