Slides from

Silberschatz, Galvin and Gagne ©2003

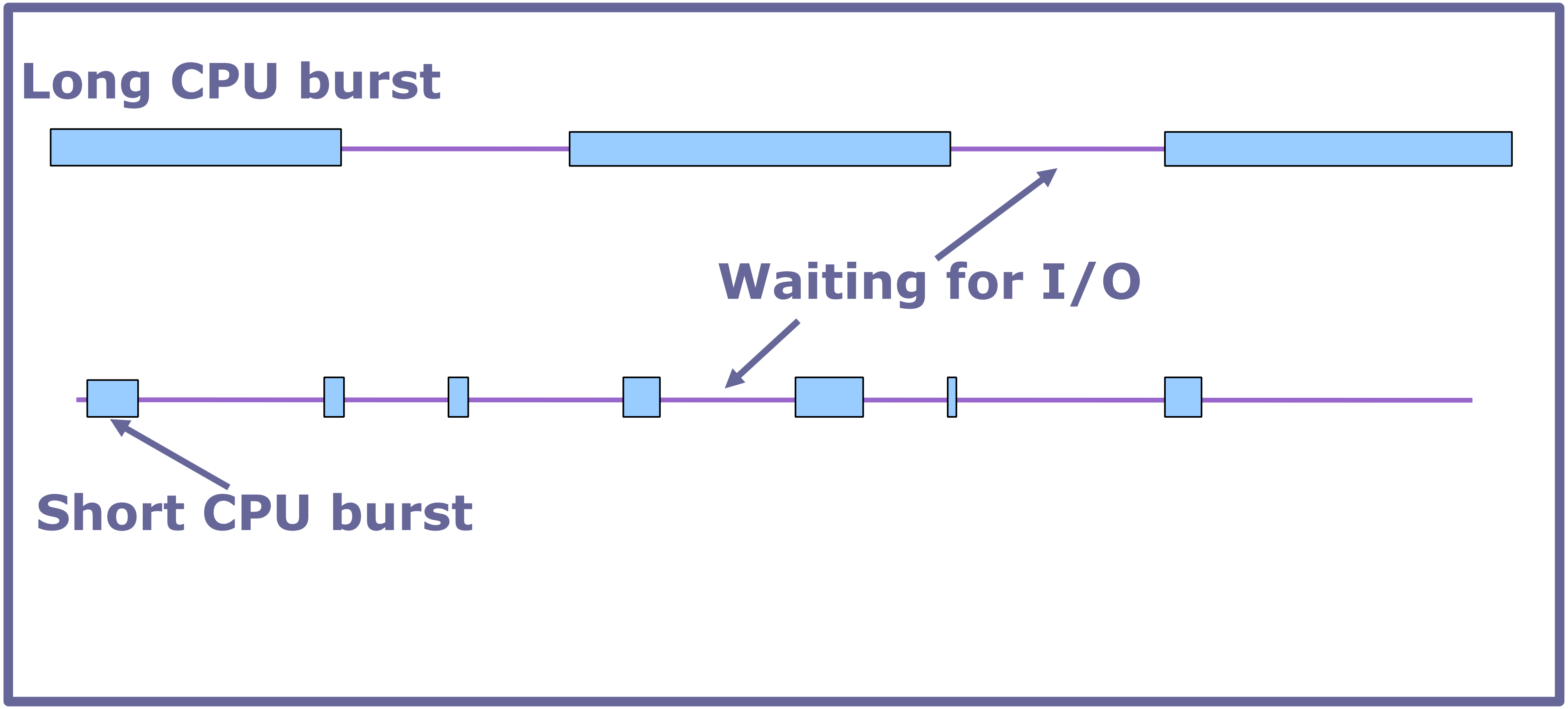
Thu D. Nguyen

Michael Hicks

Basic Concepts

- CPU-I/O burst cycle – Process execution consists of a *cycle* of CPU execution and I/O wait.
- CPU burst distribution
  - What are the typical burst sizes of a process's execution?

Process Behavior

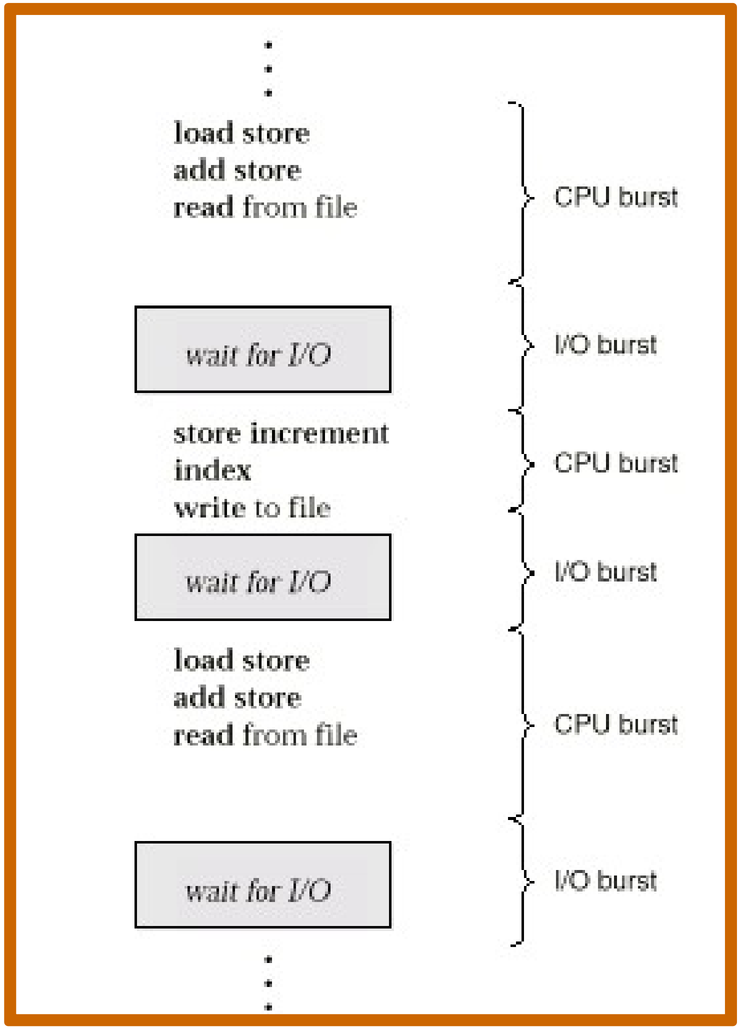


Long CPU burst

Waiting for I/O

Short CPU burst

Alternating Sequence of CPU And I/O Bursts



⋮

load store
add store
read from file

wait for I/O

store increment
index
write to file

wait for I/O

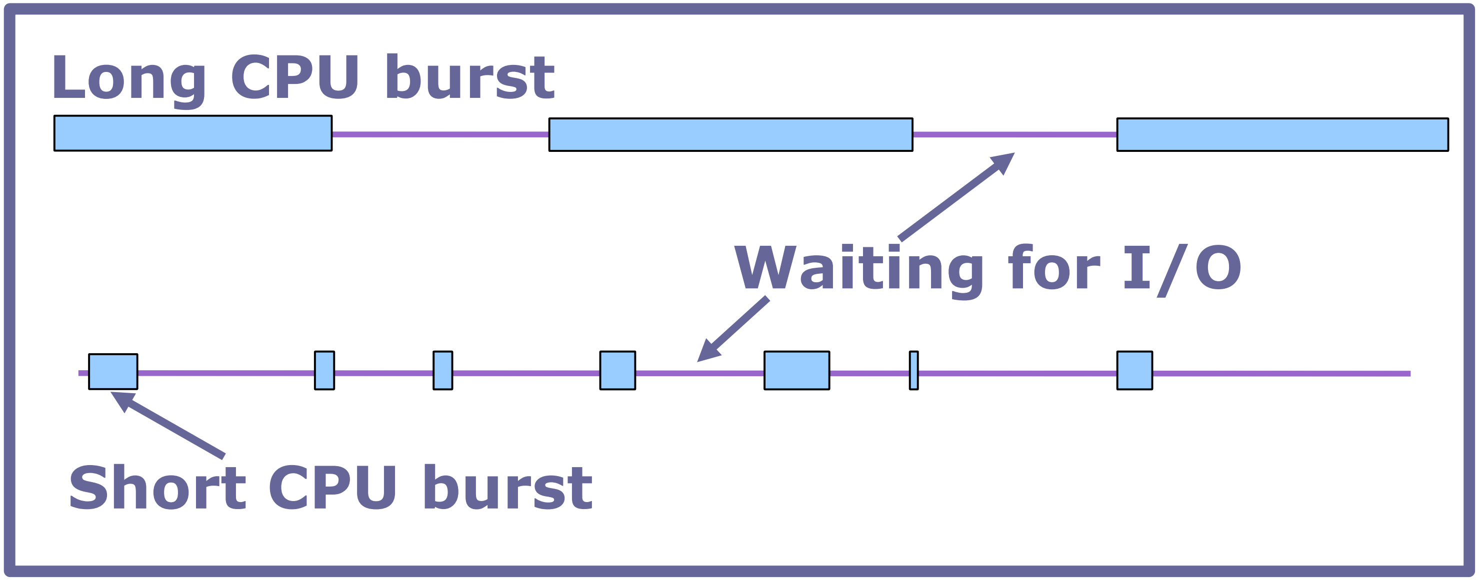
load store
add store
read from file

wait for I/O

⋮

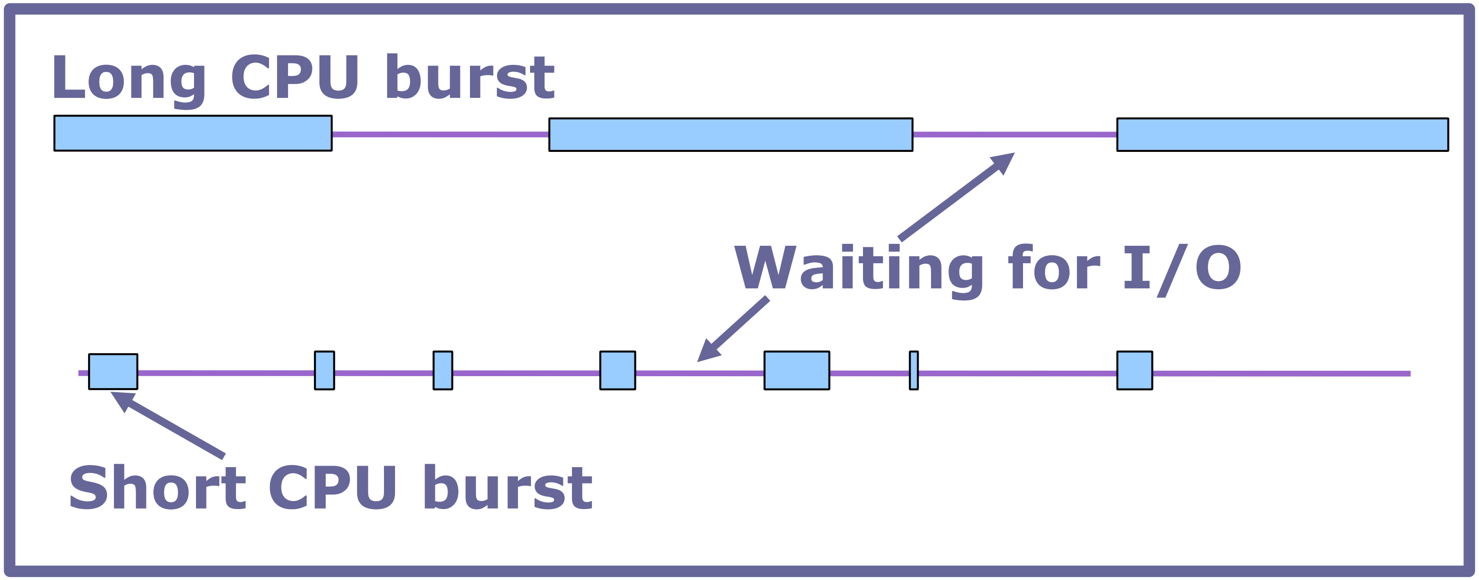
Job Behavior

- I/O bound jobs: Jobs that perform lots of I/O tend to have short CPU bursts
- CPU bound jobs: Jobs that perform very little I/O tend to have very long CPU bursts

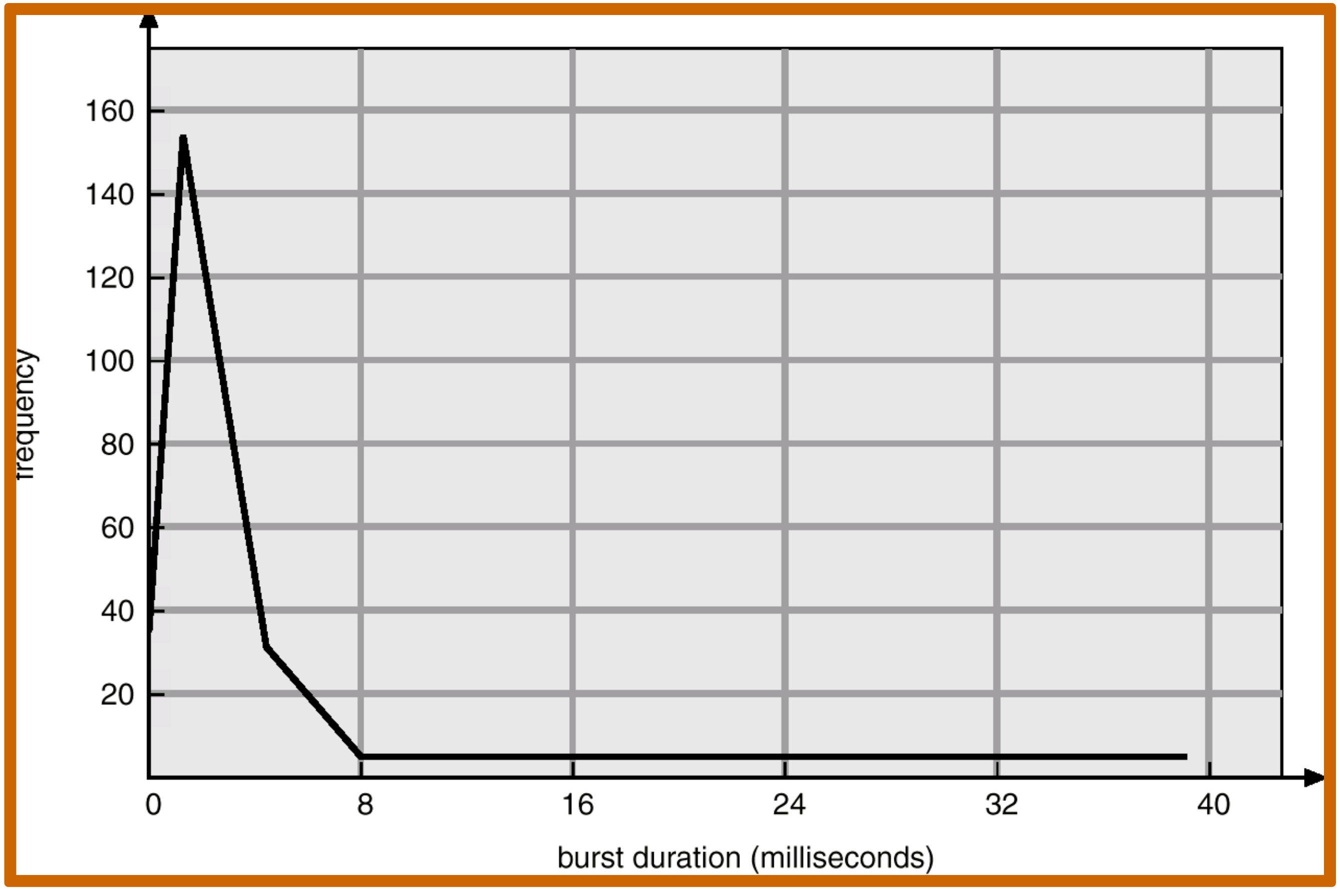


Job Behavior

- Distribution tends to be hyper-exponential: Very large number of very short CPU bursts. Small number of very long CPU bursts.



Histogram of CPU-burst Times



CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them

<When scheduling decisions 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

<When scheduling decisions 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 is *nonpreemptive*

<When scheduling decisions 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

- All other scheduling *preemptive*

<nonpreemptive>

once the cpu has been allocated to a process, the process keeps the cpu until it terminates or switches to a waiting state.

<nonpreemptive>

Aka cooperative

assume a cooperative process.
let that process decide and let's not interrupt it.

<modern operating systems>

preemptive
or nonpreemtive?

<Cooperative>

- Microsoft Windows 3.x. (pre 1995)
- Mac pre OS X (1984 - 2000)
- Commodore 64

<preemptive>

- Microsoft Windows OSs starting w/ Windows 95
- Mac OS X uses preemptive scheduling
- Pre-OS x versions used "cooperative scheduling"
- Linux, BeOS, Symbian OS, Solaris, Android, iOS, all preemptive

<Dispatcher>

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

<Dispatch Latency>

<Dispatch Latency>

Dispatch latency - time it takes for the dispatcher to stop one process and start another running

<summary>

- Algorithms important for knowing history
- Algorithms in current use today
  - Multi-level feedback queue
- Algorithms that give better results but take too long to run.
- Multicore challenges

<What criteria>

What criteria should we use to schedule processes?

<What criteria>

- CPU utilization – keep the CPU as busy as possible (40-90%)

<What criteria>

- CPU utilization - keep the CPU as busy as possible (40-90%)

Poll: how busy are our CPUs?

<What criteria>

- CPU utilization - keep the CPU as busy as possible (40-90%)
- Throughput - # of processes that complete their execution per time unit

<What criteria>

- CPU utilization - keep the CPU as busy as possible (40-90%)
- Throughput - # of processes that complete their execution per time unit
- Latency (avg) - average time from when a task arrives until it completes

Maximizing throughput may not minimize latency

Example: 100 jobs arrive together. One takes 100 seconds and the rest 1.

Running the first first and then running the rest yields and avg. latency of 149.5 s (100 + 101 + 102 …)

Running the short jobs first yields an avg. latency of 51.5s.

Latency is 3x better but throughput is the same.

<What criteria>

- CPU utilization - keep the CPU as busy as possible (40-90%)
- Throughput - # of processes that complete their execution per time unit
- Latency (avg) - average time from when a task arrives until it completes
- Latency (99%) - time required by 99% of tasks to complete.

(minimize variance of latencies)

users would be more satisfied with an UI that processes each input in 100ms than one that usually processes input in 50ms but with an occasional 5 sec. pause

<What criteria>

- CPU utilization – keep the CPU as busy as possible (40-90%)
- Throughput – # of processes that complete their execution per time unit
- latency – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue

<What criteria>

- CPU utilization - keep the CPU as busy as possible (40-90%)
- Throughput - # of processes that complete their execution per time unit
- Turnaround time - amount of time to execute a particular process
- Waiting time - amount of time a process has been waiting in the ready queue
- Response time - amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

<What criteria>

- CPU utilization - keep the CPU as busy as possible (40-90%)
- Throughput - # of processes that complete their execution per time unit
- Turnaround time - amount of time to execute a particular process
- Waiting time - amount of time a process has been waiting in the ready queue
- Response time - amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)
- Real time guarantees - guaranteeing a certain amount of resources by a deadline.

<Optimization Criteria>

- MAX
    - CPU utilization
    - Throughput

- MIN
    - Turnaround time
    - Waiting time
    - Response time

<Optimization Criteria>

- MAX
  - CPU utilization
  - Throughput

- MIN
  - Turnaround time
  - Waiting time
  - Response time

- These are performance related

<Optimization Criteria> non performance related

- Predictability
  - Job should run in the same amount of time regardless of total system load
  - Response times should not vary
- Fairness
  - Don't starve any processes
- Enforce priorities
  - Favor high priority processes
- Balance resources
  - Keep all resources busy

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

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

avg. wait time?

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_2 , P_3 , P_1

The Gantt Chart for the schedule is:

avg. wait time?
Better or worse than previous ordering?

http://youtu.be/mAPRrdgYU7o

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_2 , P_3 , P_1

The Gantt Chart for the schedule is:

avg. wait time?
Better or worse than previous ordering?

Convoy effect short process behind long process

Shortest-Job-First (SJF) Scheduling

Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time

Shortest-Job-First (SJF) Scheduling

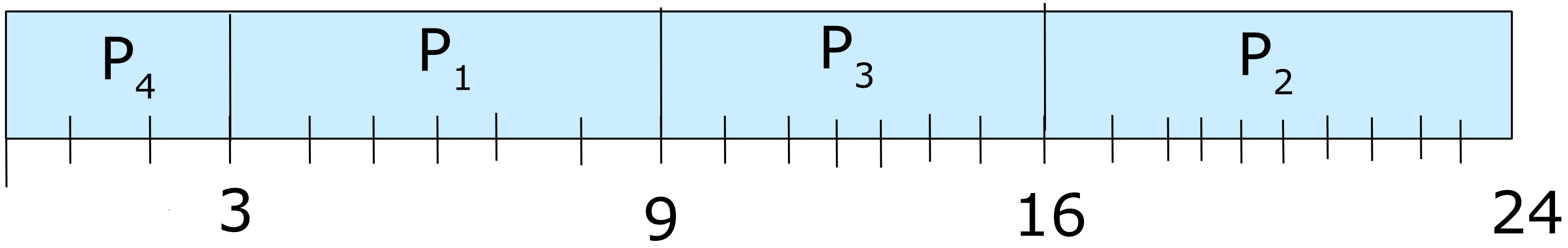
Two schemes:

- nonpreemptive - once CPU given to the process it cannot be preempted until completes its CPU burst

- preemptive - if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF)

Example of Non-Preemptive SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



- Average waiting time

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0	6
P_2	0	8
P_3	1	7
P_4	2	3

- Average waiting time

Team Work

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

FCFS
Nonpreemtive SJF
SRTF (Shortest Remaining Time First)

<team work>

SJF

- Optimal

<recap>

Shortest-Job-First (SJF) Scheduling

Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time

<anybody see a problem w/ this?>

Shortest-Job-First (SJF) Scheduling

Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time

Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

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

1. t = actual length of nth CPU burst
   τ_n = past history.
2. τ_{n+1} = predicted value for next CPU burst
3. $0 < \alpha < 1$

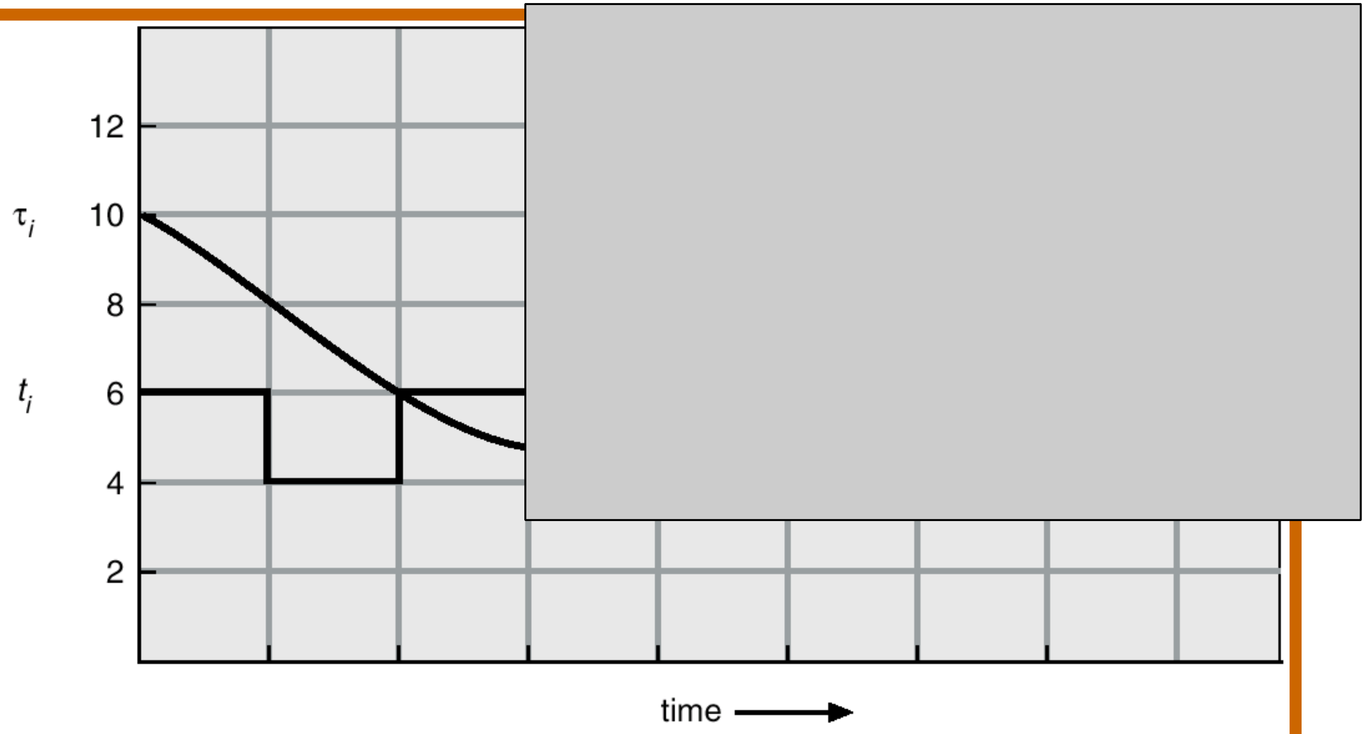
<exponential averaging>

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

- $\alpha = 1?$
- $\alpha = 0?$
  - Good estimates with 1/2
  - each successive term has less weight than its predecessor

Prediction of the Length of the Next CPU Burst

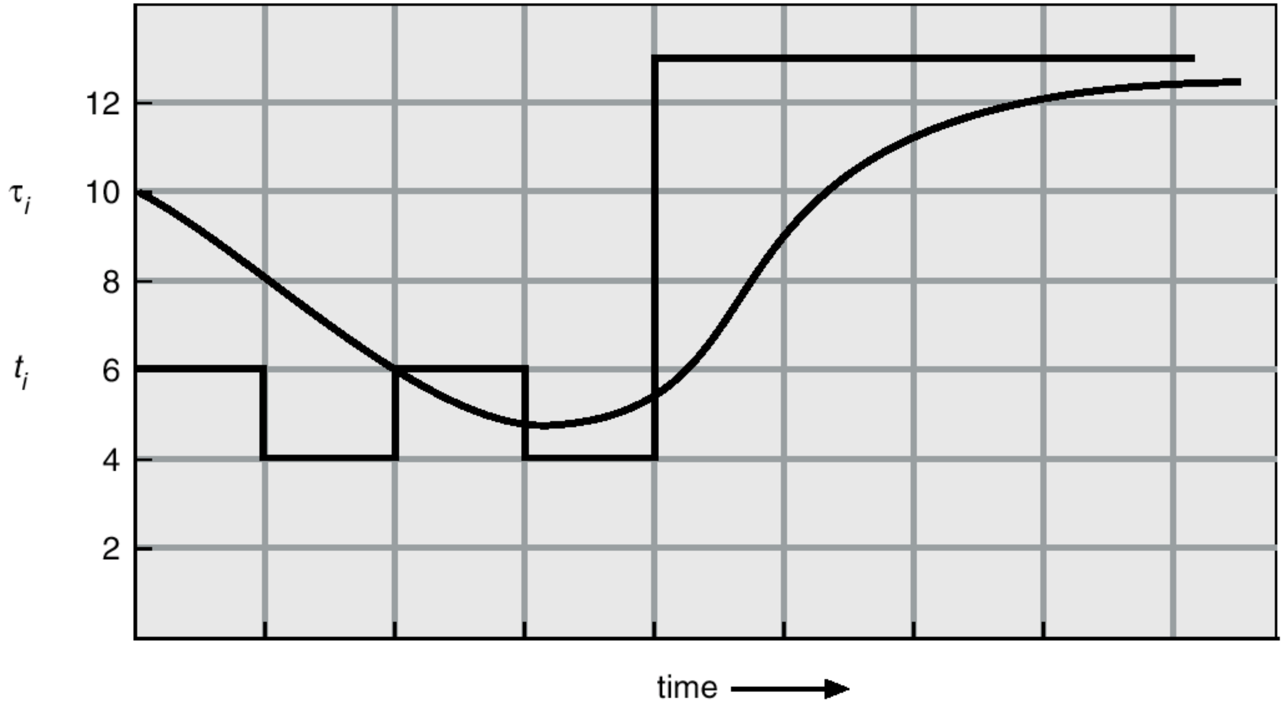
a = ½ t0 = 10



CPU burst (t_i)		6	4	6	4	13	13	13	. . .
"guess" (τ_i)	10	8	6	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	. . .

Prediction of the Length of the Next CPU Burst

a = ½   t0 = 10



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

<round robin>

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds.
  - Once this time has elapsed, the process is preempted and placed at the end of the ready queue.

- If there are *n* processes in the ready queue and the time quantum is *q*, then no process waits more than __ time units.

Round Robin (RR)

- To implement RR:
  - we keep the ready queue as a FIFO queue of processes.
  - New processes are added to the tail of the ready queue.
  - The CPU scheduler picks the first process on the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

Round Robin (RR)

- Then one of two things will happen:

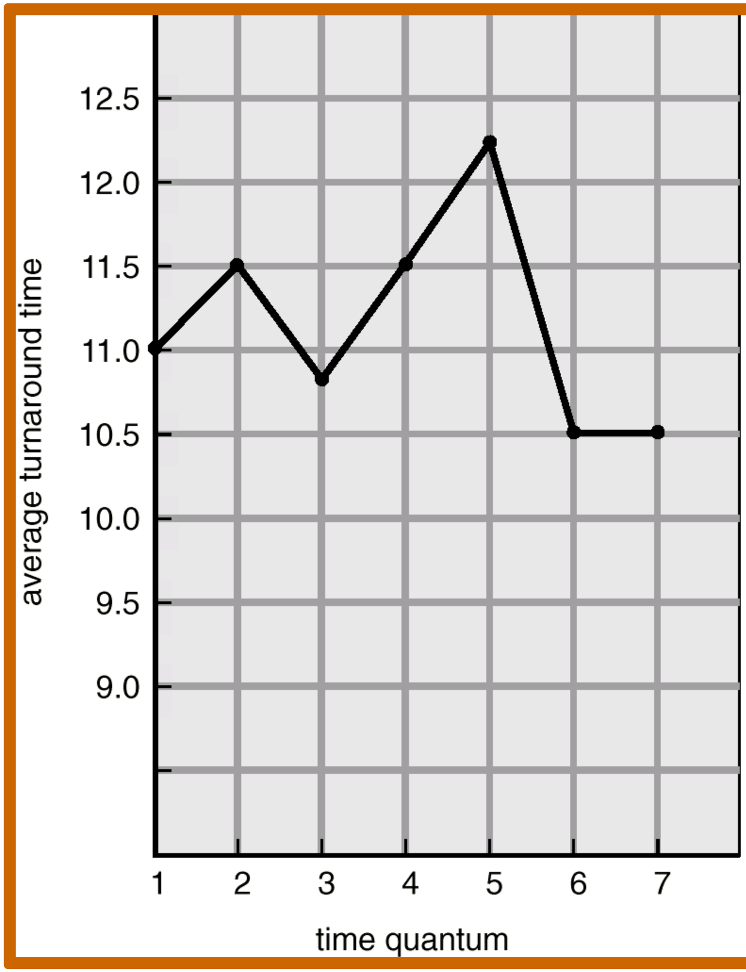
Round Robin (RR)

- Then one of two things will happen:
  - The process will have a CPU burst of less than 1 quantum and process releases CPU voluntarily
  - If the process has a CPU burst greater than 1 quantum the timer goes off causing an interrupt to the OS. Context switch occurs and process gets put on tail of queue

<Choosing *q*>

- Very large-degenerates to which scheduler?
- Very small-dispatch time dominates
- Rule of thumb-for better turnaround time, quantum should be slightly greater than time of 'typical job' CPU burst

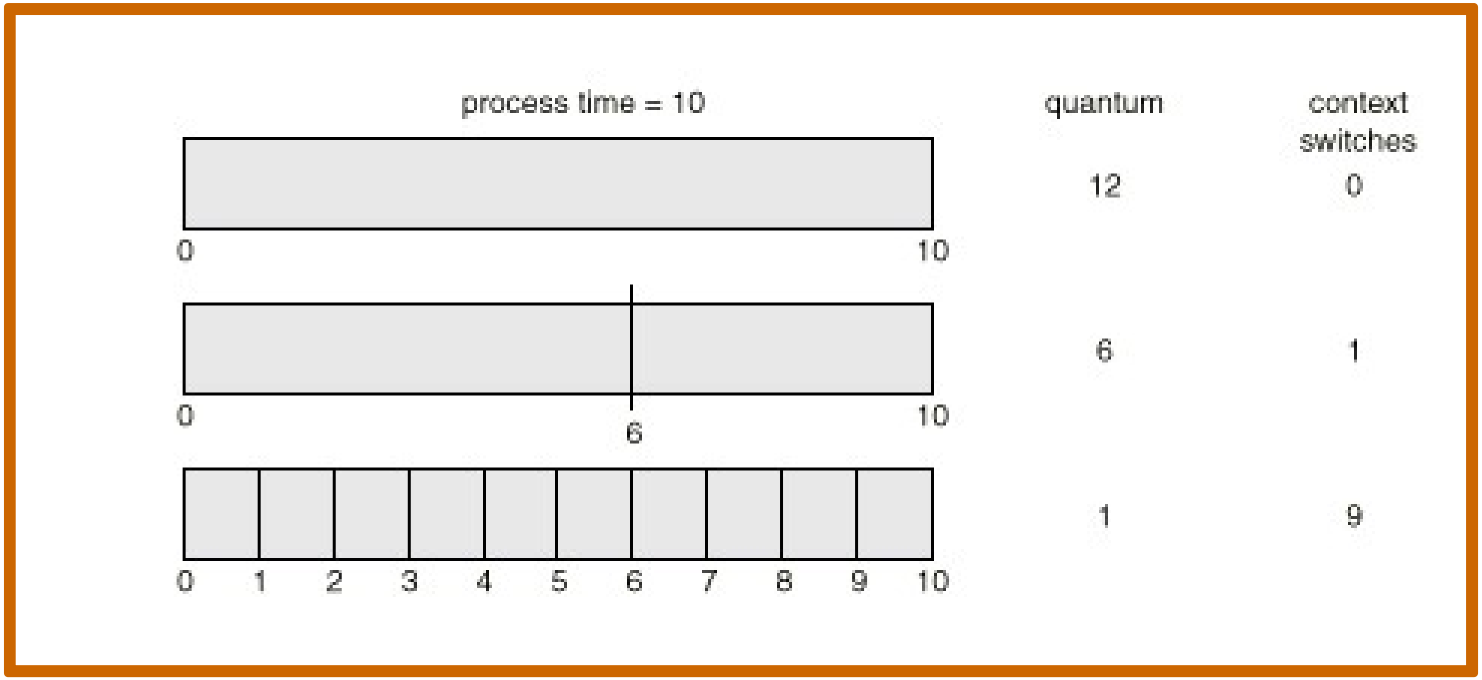
Turnaround time varies /w time quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Rule of thumb: 80% of CPU bursts should be shorter than time quantum

Time quantum and context switch time



Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- What is the Gantt chart?
- Typically, higher average turnaround than SJF, but better *response*

<round robin>

- Typically higher average turnaround than SJF
- But better *response*

<Priority Scheduling>

- Prefer one process over another
- One common implementation
  - A priority number (integer) is associated with each process
  - OS schedules the process w/ the highest priority (smallest integer = highest priority) - mac/linux -20 to 20
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- nice renice

<Priority Scheduling>

- Anybody see any problems with this?

<Priority Scheduling>

- Problem: Starvation - low priority processes may never execute

- Solution: Aging-as time progresses increase the priority of a process.

Multilevel Priority Queue

- Ready queue is divided into *n* queues, each w/ its own scheduling algorithm, e.g.,
    - Foreground (interactive) - RR
    - Background - FCFS
  - Now need to schedule between queues

Linux 2.4 → 2.6

- Linux 2.4 all processes one ready queue
- When scheduler ran it looked for the highest priority job on the queue.
- What do people think?

Linux 2.4 → 2.6

- Linux 2.4 all processes one ready queue
- When scheduler ran it looked for the highest priority job on the queue.
- What do people think?
- How could we improve that?

<Scheduling done between queues>

- Fixed priority scheduling (serve all from foreground then from background)
  - Doesn't solve starvation
- Time slice – each queue gets a certain amount of CPU time which it can schedule among its processes. e.g.
  - 80% to foreground in RR
  - 20% to background in FCFS.

<Multilevel Scheduling Design>

How to avoid undue increase in turnaround time for longer processes when short new jobs regularly enter the system

<Multilevel scheduling design>

- Solution 1: vary preemption times according to queue
  - Processes in lower priority queues have longer time slices.
- Solution 2: promote a process to a higher queue
  - After it spends a certain amount of time waiting for service in its current queue move it up
- Solution 3 ...

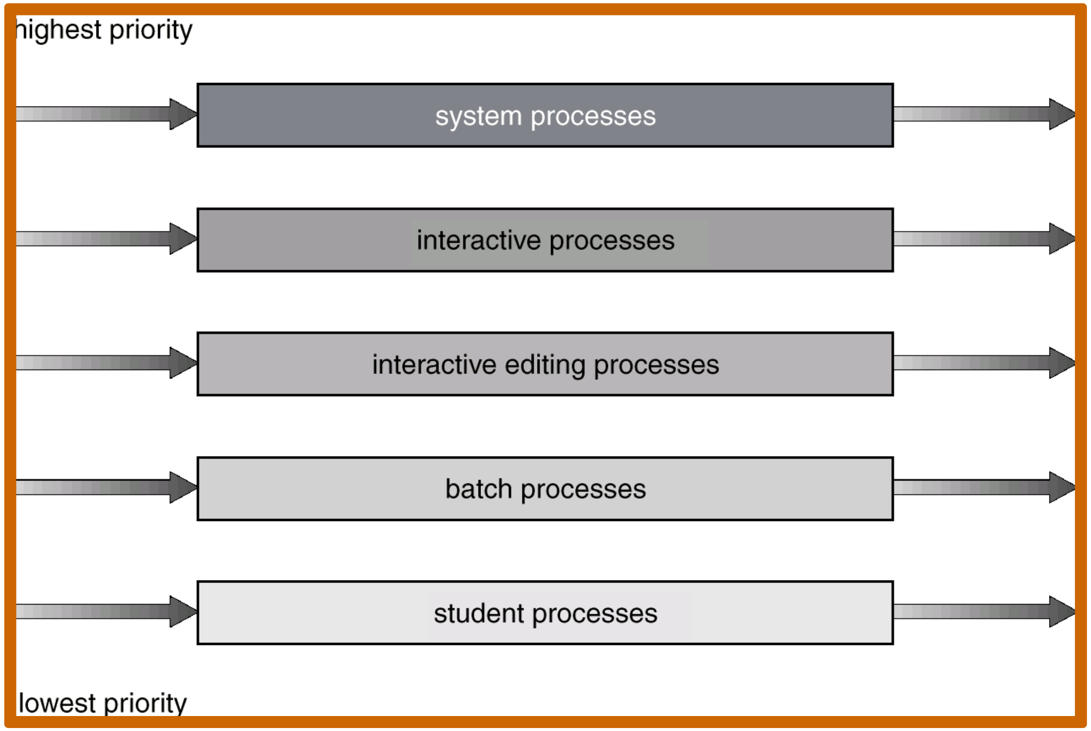
<Multilevel scheduling design>

- Solution 3: allocate fixed share of CPU time to jobs
  - If process doesn't use its share give it to other processes

  for, ex. Linux Q=200ms

  - Variation on this idea: lottery scheduling
    - Assign a process "tickets" (# of tickets is share)
    - Pick random number and run the process w/ the winning ticket

Multilevel Queue Scheduling



Processes permanently assigned to one queue based on some property

each queue has own scheduling algorithm

scheduling among queues:
- absolute
- timeslice

Multilevel Feedback Queue

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

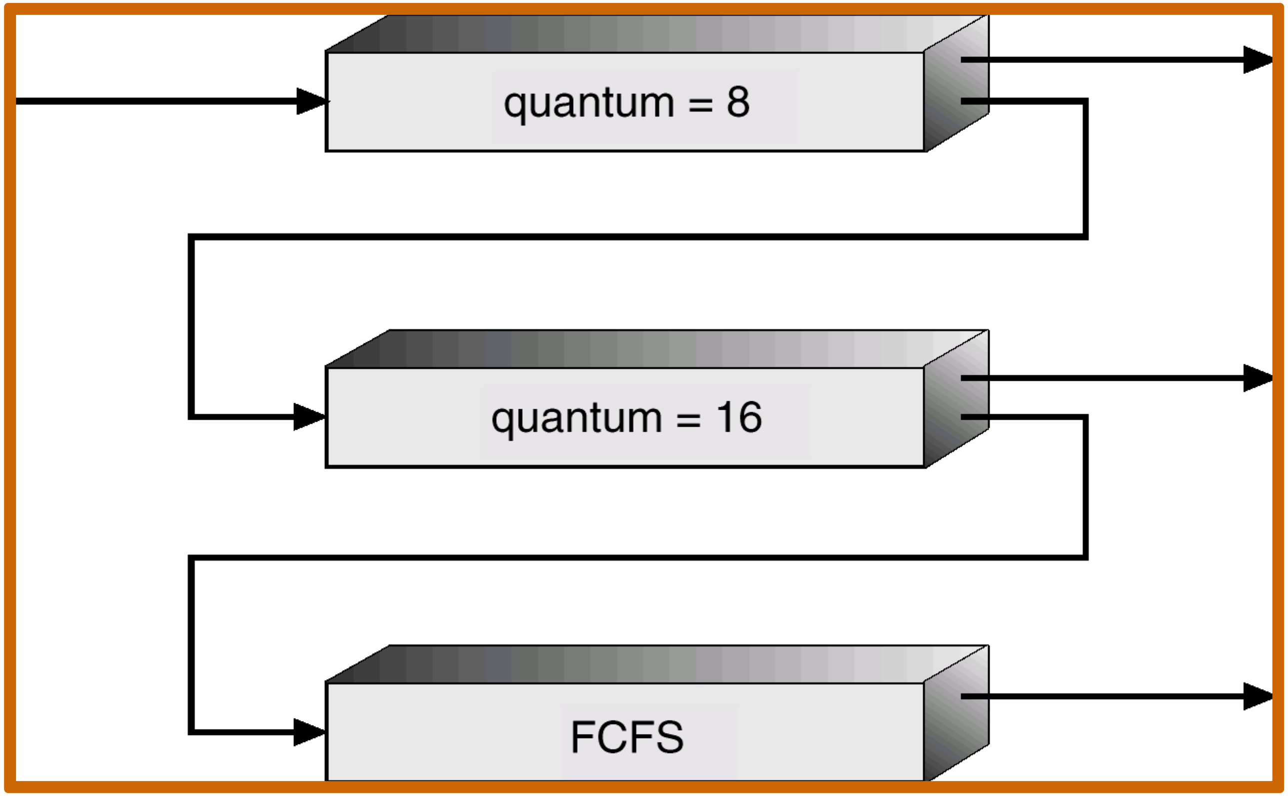
<example of multilevel feedback queue>

- Three queues
    - q0-time quantum 8 milliseconds
    - q1-time quantum 16 milliseconds
    - q2-FCFS

<example of multilevel feedback queue>

- Scheduling
  - A new job enters queue q0, which is served RR. When it gains CPU, job receives 8ms. If it doesn't finish, it is moved to queue q1
  - At q1 job is again serviced RR and receives 16 additional ms. If it does not complete it is preempted and moved to queue q2.
  - At q2 the job is serviced FCFS

Multilevel Feedback Queues



Overload Control

servers typically have highly variable load

Flash crowd: emergency servers

Ebay auction

Ticketmaster

Live blogging of event

Can't solve w/ scheduling

Solution 1: reduce work

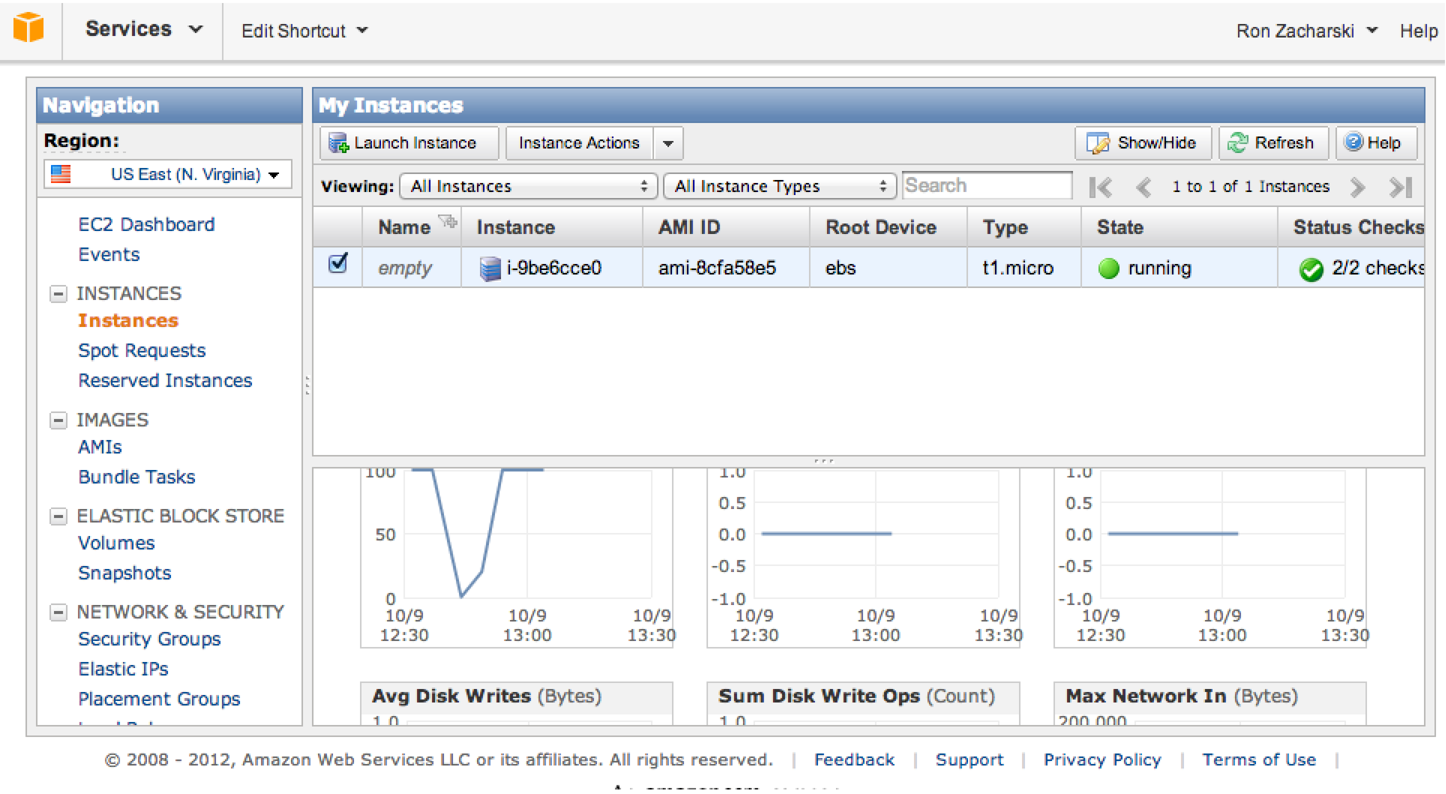
Distasteful but sometimes necessary

- reject requests
- do less work per request
    - Switch from 720P to 480i
    - Serve static pages instead of dynamically generated ones
- turn off other services (mail server)

Solution 2: increase resources

- cloud services
  Amazon aws

- squarespace example



	Name	Instance	AMI ID	Root Device	Type	State	Status Checks
☑	*empty*	i-9be6cce0	ami-8cfa58e5	ebs	t1.micro	● running	✓ 2/2 checks

© 2008 - 2012, Amazon Web Services LLC or its affiliates. All rights reserved.

Worksheet