Scheduling

Chapter 7 OSPP

Part I

(skip 7.3, 7.4)

Today

- HW #2 due Thursday
- Lab #1 teams formed

Main Points

- Scheduling policy: what to do next, when there are multiple threads ready to run
  - Or multiple packets to send, or web requests to serve, or …
  - We will focus on processes
  - Equally applies to threads

Example

- You manage a web site, that suddenly becomes wildly popular. Do you?
  - Buy more hardware?
  - Implement a different scheduling policy?
  - Turn away some users? Which ones?
- How much worse will performance get if the web site becomes even more popular?
- Provide some insight into this problem

Roadmap

- Definitions
  - response time, throughput, predictability, …
- Uniprocessor policies
  - FIFO, round robin, optimal
  - multilevel feedback as approximation of optimal
- Multiprocessor policies
  - Affinity scheduling, gang scheduling
- Queueing theory
  - Can you predict/improve a system’s response time?

Definitions

- Task/Job
  - User request: e.g., mouse click, web request, shell command, … (I/O and computation)

- Workload
  - Set of tasks for system to perform

- ~~Latency/~~response time
  - How long does a task take to complete?
  - Response can also be the *first* CPU slice

- Throughput
  - How many tasks can be done per unit of time?

Definitions

- Overhead
  - How much extra work is done by the scheduler?
- Fairness
  - How equal is the performance received by different users?
- Predictability
  - How consistent is the performance over time?

Definitions

- Preemptive scheduler
  - If we can take resources away from a running task
- Work-conserving
  - Resource is used whenever there is a task to run
  - For non-preemptive schedulers, work-conserving is not always better (i.e. sometimes holding a resource idle is better)
- Scheduling algorithm
  - takes a workload as input
  - decides which tasks to do first
  - Performance metric (throughput, response) as output
  - Only preemptive, work-conserving schedulers to be considered

First In First Out (FIFO)

- Schedule tasks in the order they arrive
  - Continue running them until they complete or give up the processor

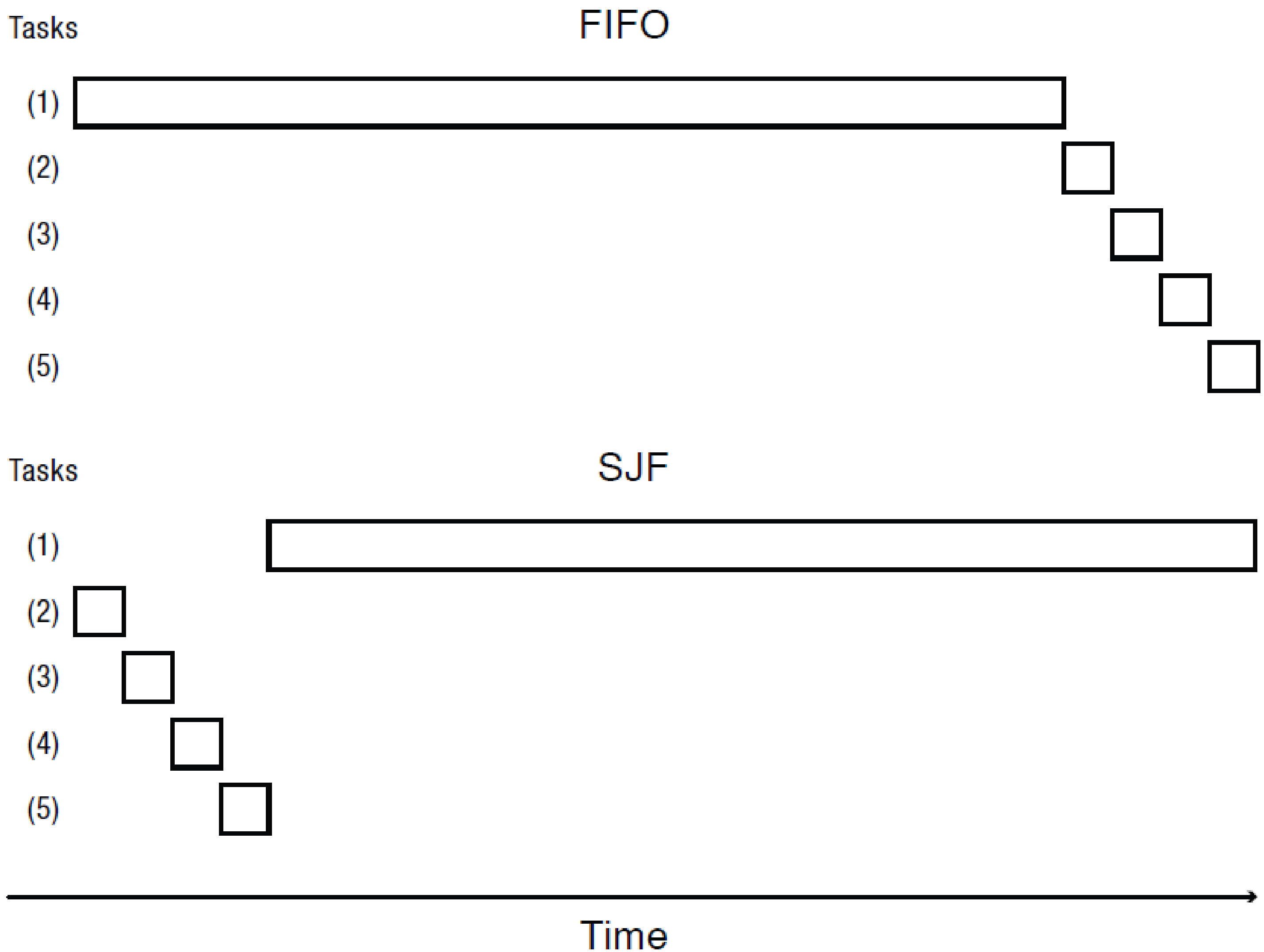
- On what workloads is FIFO particularly bad?
- Lots of small jobs, a few big ones
- Small ones get stuck behind big ones
- Jobs that never end

Shortest Job First (SJF)

- Always do the task that has the shortest remaining amount of work to do
  - Often called Shortest Remaining Time First (SRTF)

- Suppose we have five tasks arrive one right after each other, but the first one is much longer than the others
  - Which completes first in FIFO?
  - Which completes first in SJF?

FIFO vs. SJF



Question

- Claim: SJF is optimal for average response time
  - Why? Easy to prove by contradiction.

- Does SJF have any downsides?
- Starvation (particularly for STRF: pre-emptive)
- Wide variation in response (short are short, long are LOOONNNGGGGG)
- Have to know run lengths

Can we do SJF in practice?

- May be hard at OS level since tasks are black boxes but concept can be widely applied
- Think about Web requests
  - You can queue web requests
  - Prioritize small ones v. large ones
  - Other examples?
    - FB post: text only, image
    - Disk I/O: favor short ones?

Question

- Is FIFO ever optimal?
  - Yes, when all requests are of equal length
- Why is it FIFO generally good?
- No context switches
- Simple (i.e. fast)
- Seems fair

Aside: Starvation and Sample Bias

- Suppose you want to compare two scheduling algorithms
  - Create some infinite sequence of arriving tasks
  - Start measuring
  - Stop at some point
  - Compute average response time as the average for completed tasks between start and stop
- Problem is at time *t*: one algorithm has completed fewer tasks
- Solution?
  - Create fixed trace from infinite

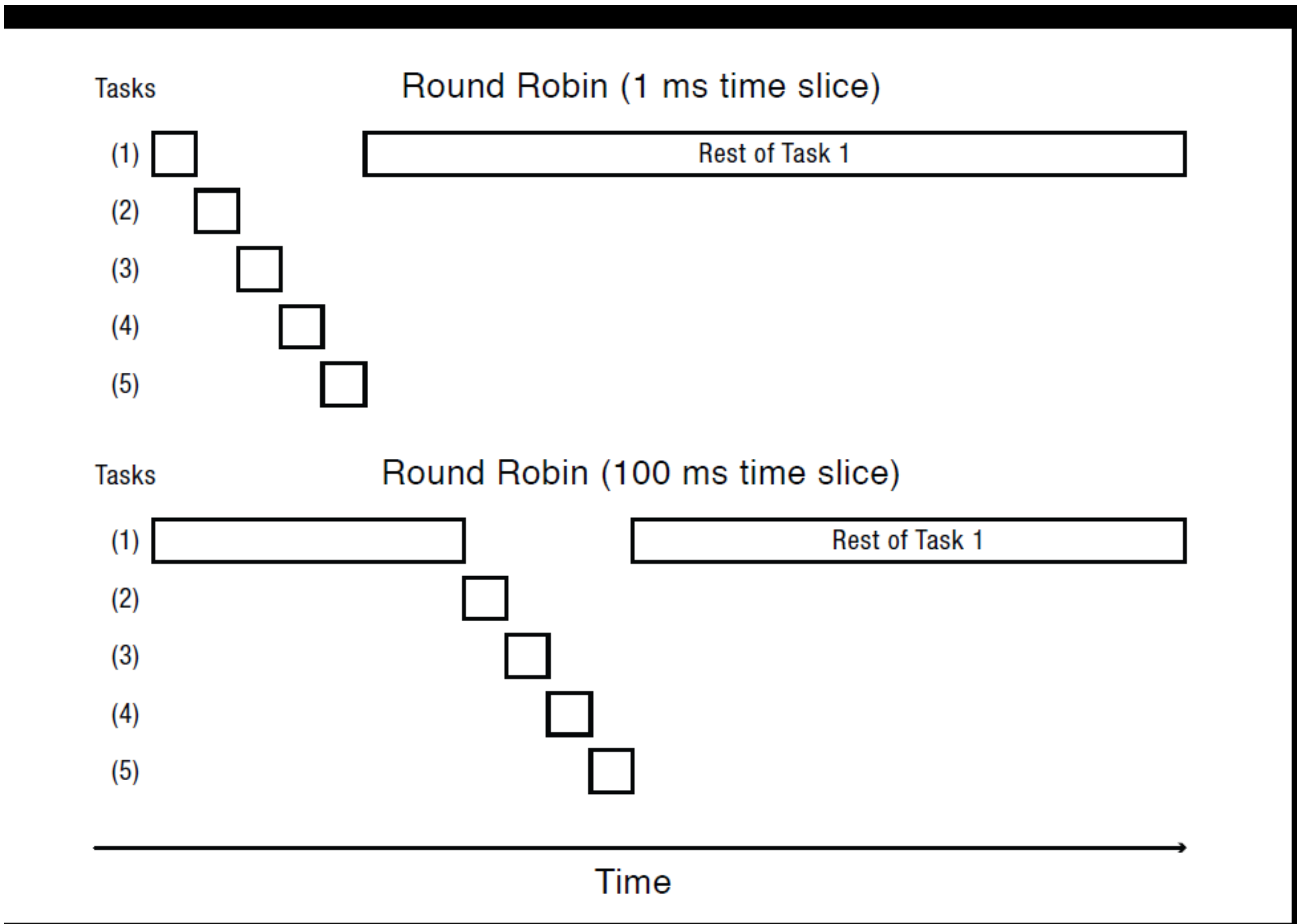
Round Robin

- Each task gets resource for a fixed period of time (time quantum Q)
  - No starvation, no favoritism
  - If task doesn't complete, it goes back in line
  - Pre-emptive as is SJF (STRF)
- Also good:
  - Guaranteed "first response" good for interactive jobs
  - Q quanta, N jobs, what is worst-case first response?

Round Robin

- Need to pick a time quantum!!
  - What if time quantum is too long?
    - Infinite?
  - What if time quantum is too short?
    - One instruction?

Round Robin

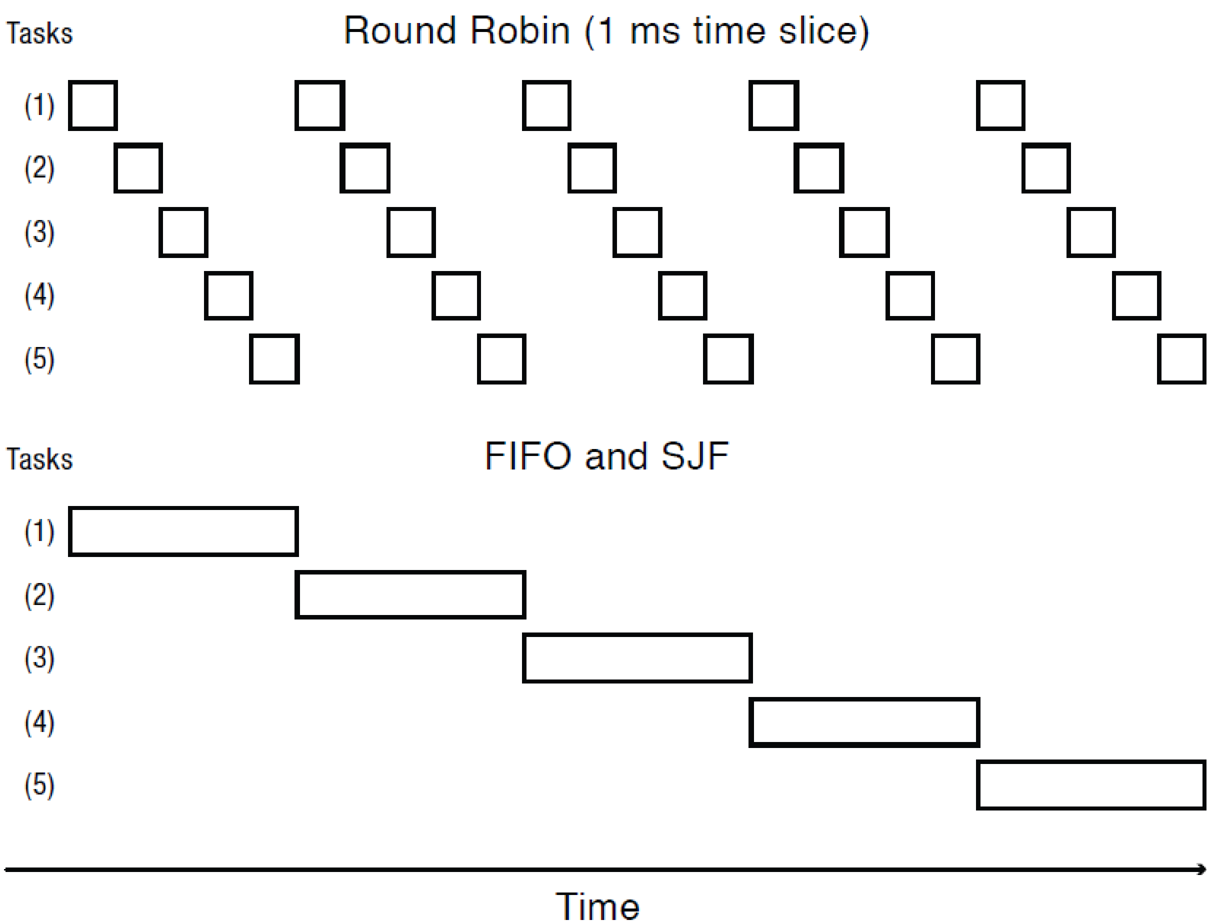

Tasks
Round Robin (1 ms time slice)
(1)
(2)
(3)
(4)
(5)
Rest of Task 1
Tasks
Round Robin (100 ms time slice)
(1)
(2)
(3)
(4)
(5)
Rest of Task 1
Time

Round Robin vs. FIFO

- Assuming zero-cost time slice, is Round Robin always better than FIFO?
  - Same size jobs time-slicing may serve little purpose except “initial” response
  - Poor average response time
  - Mixed workloads can be a problem

- However, for long-running interactive jobs …
  - round robin for video streaming
  - Even for equal size streams this maintains stable progress for all

Round Robin vs. FIFO



What about average response time?

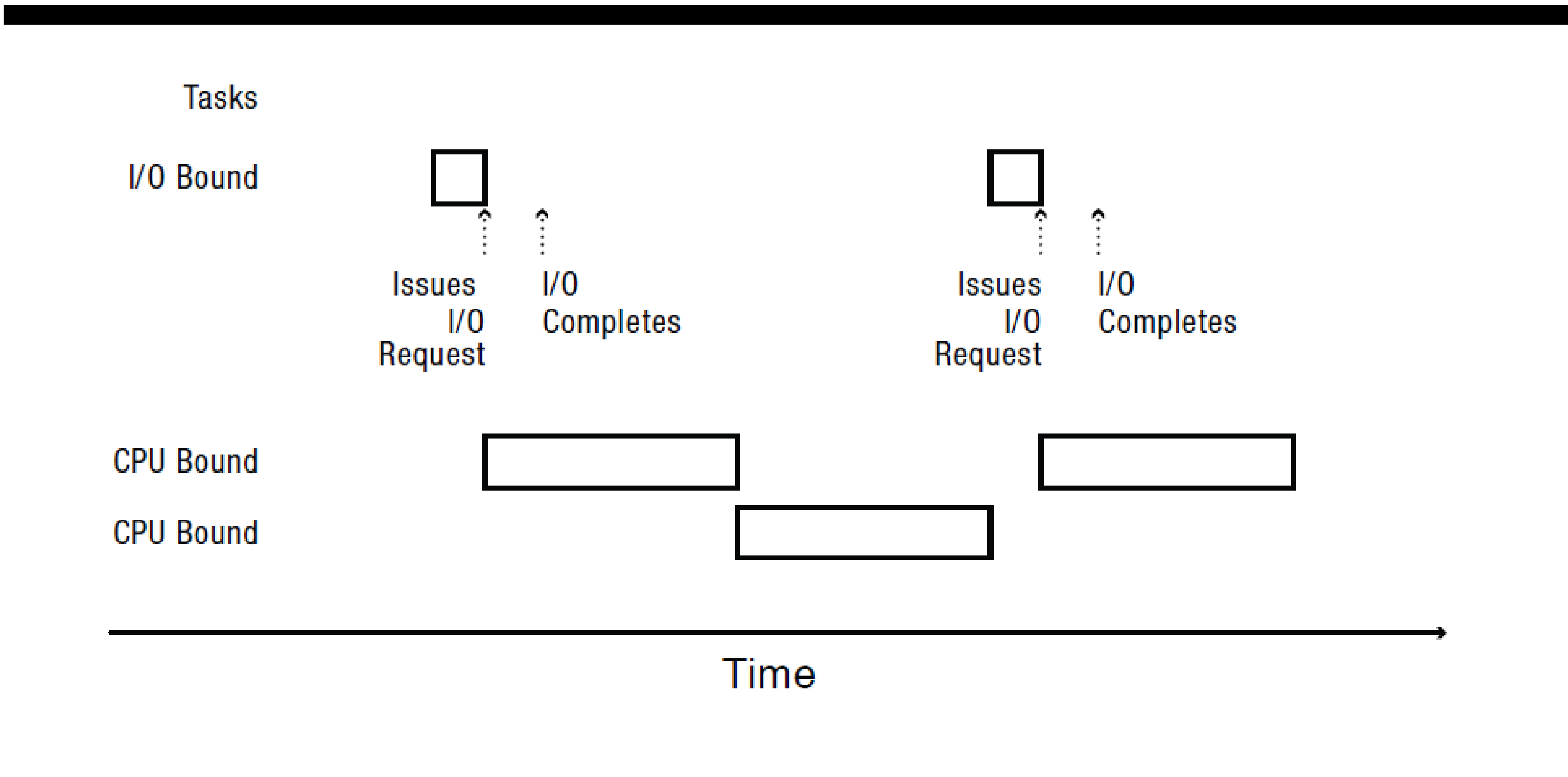
Round Robin = Fairness?

- Is Round Robin always fair?
  - Sort of but short jobs finish first!
- What is fair?
  - FIFO?
  - Equal share of the CPU?
  - What if some tasks don’t need their full share?
  - Minimize worst case divergence?
    - time task would take if no one else was running vs.
    - time task takes under scheduling algorithm with other jobs

Scheduling

Chapter 7 OSPP

Mixed Workload: Fairness



Problem: work conserving: CPU bound job is ready to roll ….

Max-Min Fairness

- One approach: maximize the minimum allocation given to a task (~ min worst case divergence)
  - If any task needs less than an equal share, schedule the smallest of these first; but how?
  - Split the remaining time using max-min
  - If all remaining tasks need at least equal share, split evenly
  - example

Multi-level Feedback Queue (MFQ)

- Hybrid solution: see any before?
- Goals:
  - Responsiveness (i.e. for interactive job)
  - Low overhead (i.e. limited context switching)
  - Starvation freedom: maybe
  - Some tasks are high/low priority (mixed workload)
  - Fairness (among equal priority tasks)
- Not perfect at any of them!
  - Used in Linux, Windows, …

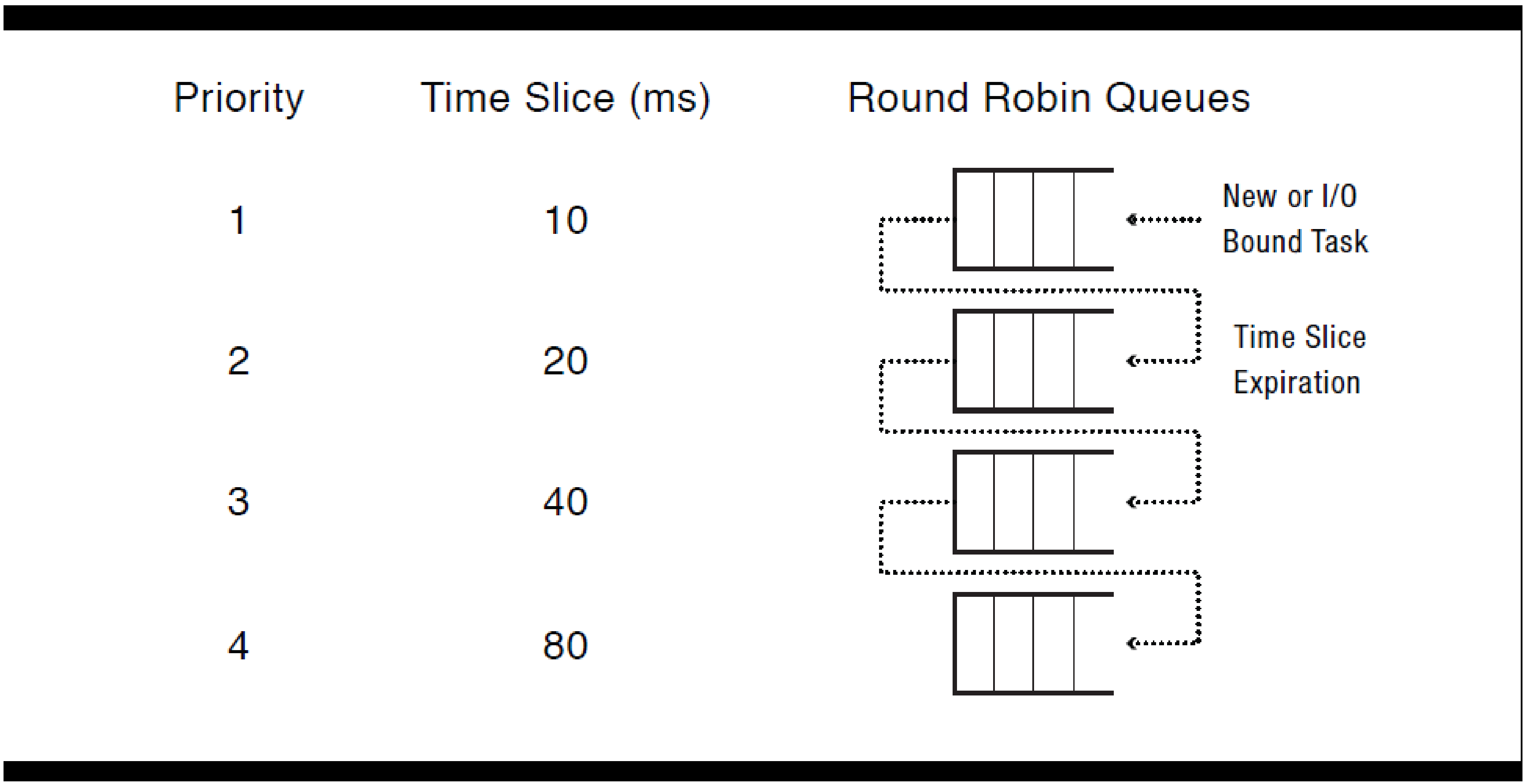
MFQ

- Set of Round Robin queues
  - Each queue has a separate priority

Why?

- High priority queues have short time slices
  - Low priority queues have long time slices
- Scheduler picks first thread in highest priority queue
- Tasks start in highest priority queue
  - If time slice expires, task drops one level

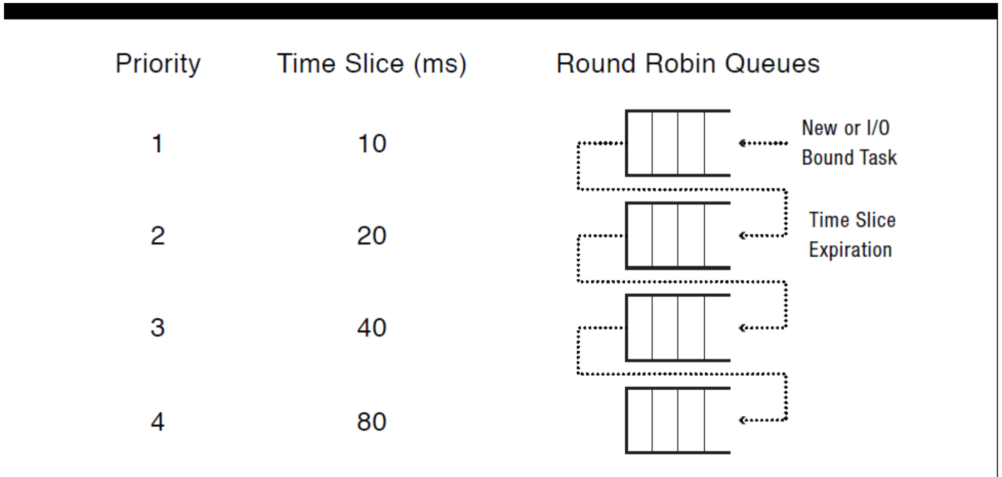
MFQ



Priority
Time Slice (ms)
Round Robin Queues
1
10
2
20
3
40
4
80
New or I/O
Bound Task
Time Slice
Expiration

Starvation Freedom

- How can starvation still happen?
  - Lots of arriving I/O bound jobs
- Solution
  - Keep track of how much a job gets over time relative to other jobs
  - Can promote a job that has received less than its fair share (e.g. 20/150 % for a job sitting in P2)



Uniprocessor Summary (1)

- FIFO is simple and minimizes overhead.
- If tasks are variable in size, then FIFO can have very poor average response time.
- If tasks are equal in size, FIFO is optimal in terms of average response time.
- If tasks are variable in size, SJF is optimal in terms of average response time.
- SJF is poor in terms of variance in response time.

Uniprocessor Summary (2)

- If tasks are variable in size, Round Robin approximates SJF.
- If tasks are equal in size, Round Robin will have very poor average response time but good interactive response time.
- Tasks that intermix processor and I/O benefit from SJF and can do poorly under Round Robin.

Uniprocessor Summary (3)

- Max-Min fairness can improve response time for I/O-bound tasks.
- Round Robin and Max-Min fairness both avoid starvation.
- By manipulating the assignment of tasks to priority queues, an MFQ scheduler can achieve a balance between responsiveness, low overhead, and fairness.

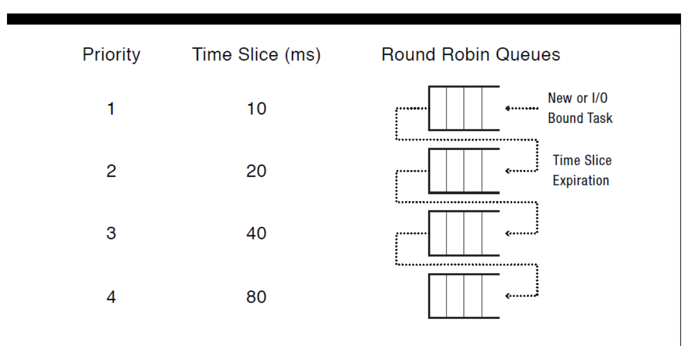
Scheduling

Chapter 7 OSPP

Part II

Multiprocessor Scheduling

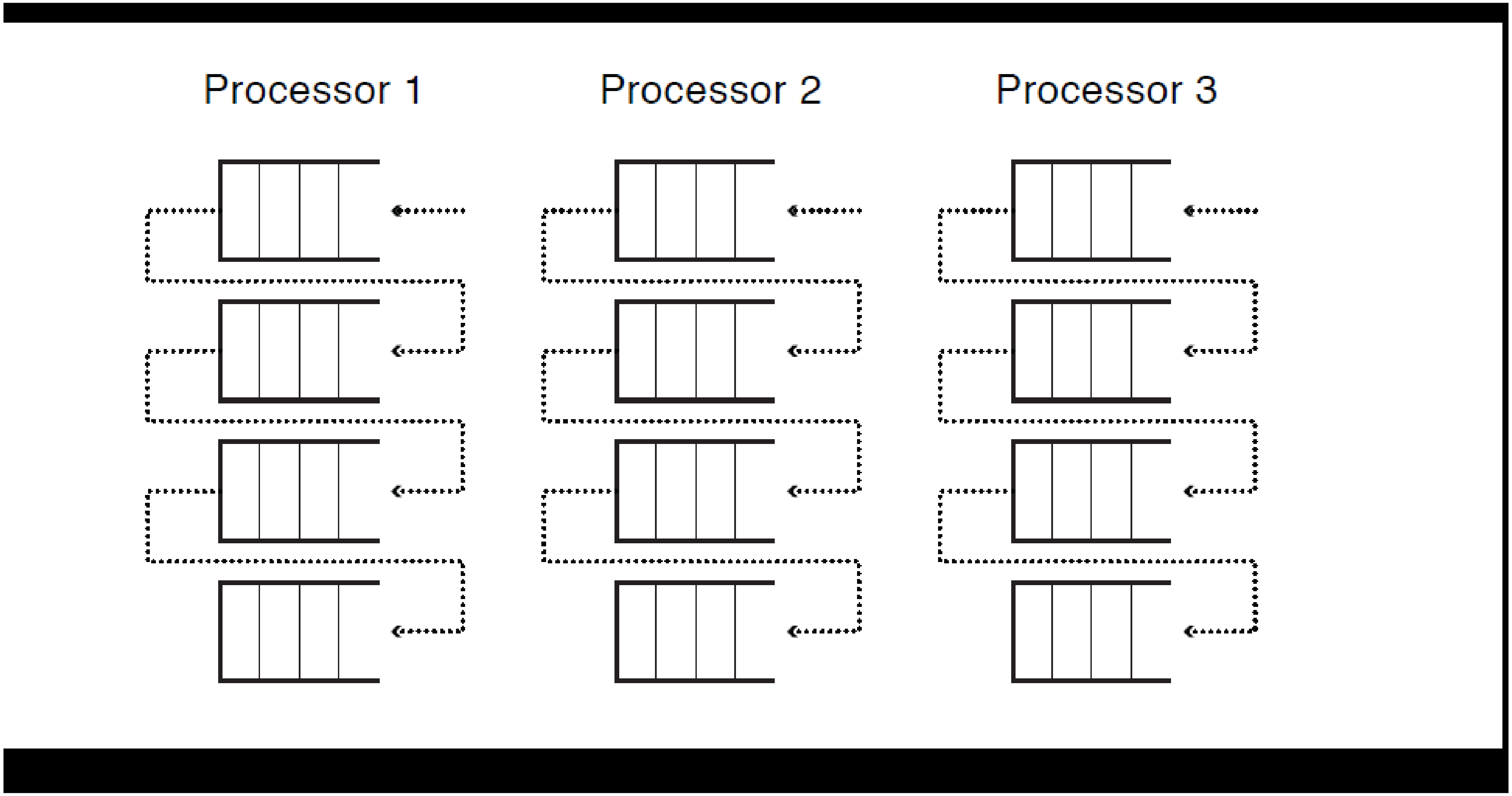
- What would happen if we used MFQ on a multiprocessor?
  - Contention for scheduler spinlock
  - Cache slowdown due to ready list data structure pinging from one CPU to another
  - Limited cache reuse: thread's data from last time it ran is often still in its old cache



Per-Processor Affinity Scheduling

- Each processor has its own ready list
  - Protected by a per-processor spinlock
- Put threads back on the ready list where it had most recently run: why?
  - Ex: when I/O completes, or on Condition->signal
- Work conserving?
- Idle processors can steal work from other processors

Per-Processor Multi-level Feedback with Affinity Scheduling



Load balancing is an issue: may need work stealing

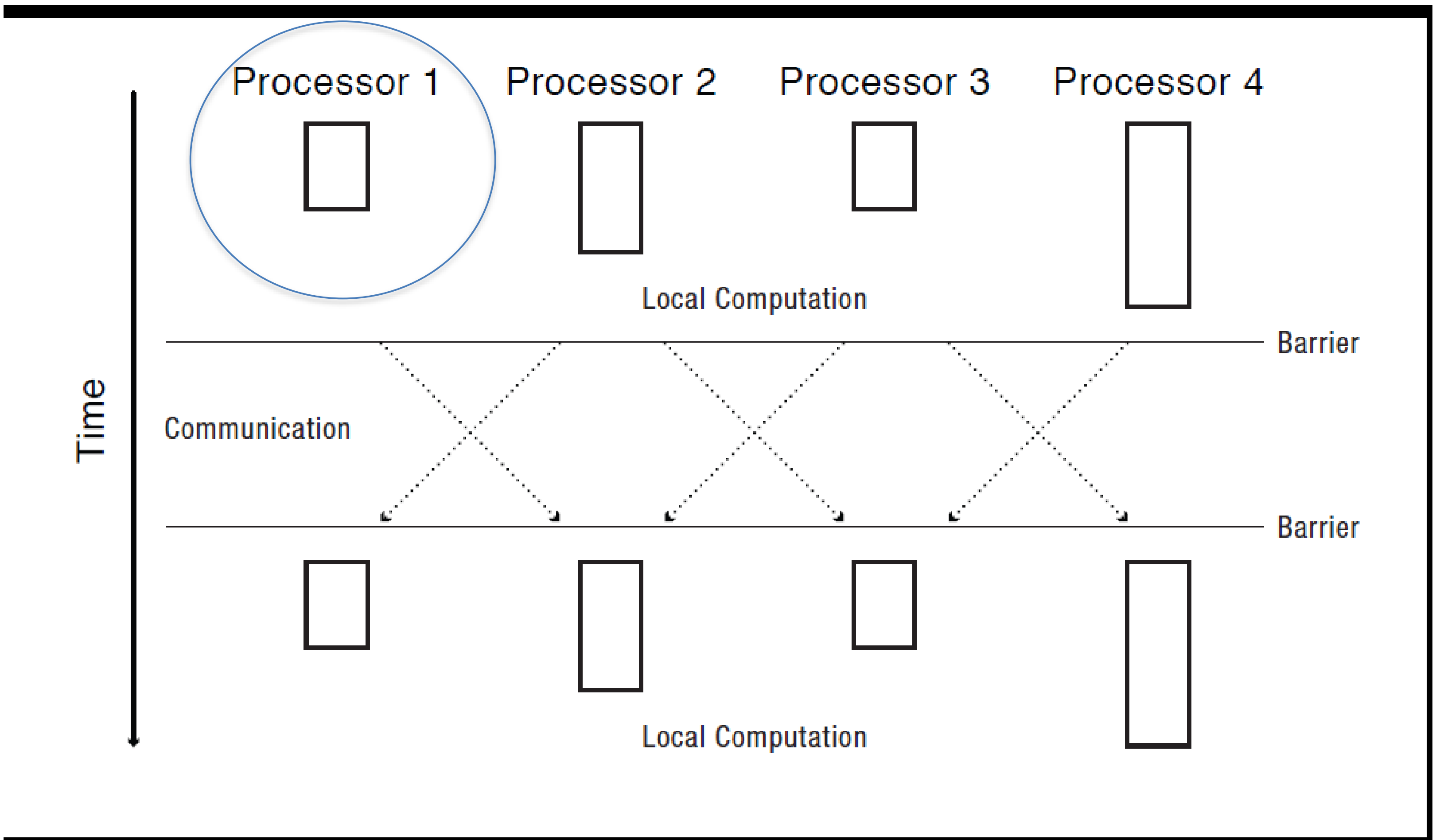
Scheduling Parallel Programs

- What happens if one thread gets time-sliced while other threads from the same program are still running?
  - Assuming program uses locks and condition variables, it will still be correct
  - What about performance?

Bulk Synchronous Parallelism: Single Program Multiple Data (SPMD)

- Loop at each processor:
  - Compute on local data (in parallel)
  - Barrier
  - Send (selected) data to other processors (in parallel)
  - Barrier
- Examples:
  - MapReduce
  - Fluid flow over a wing
  - Most parallel algorithms can be recast in BSP

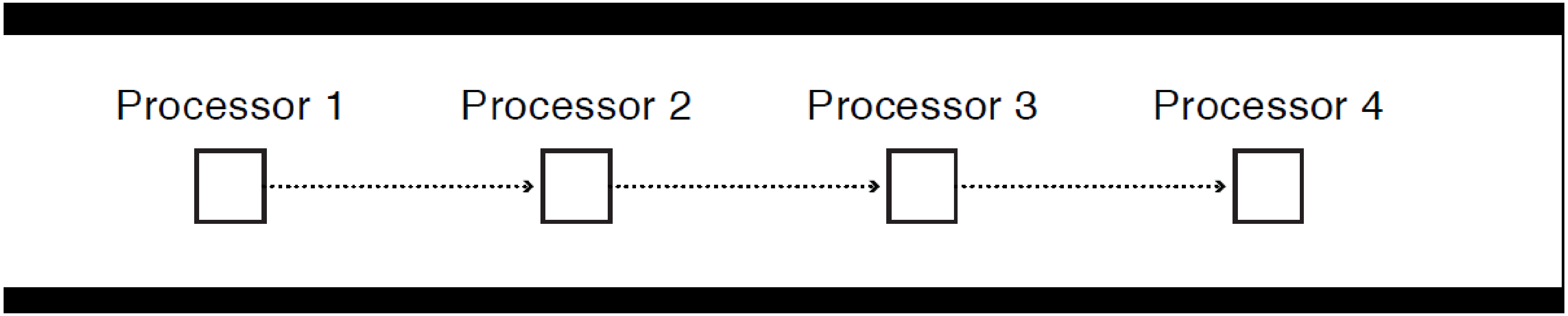
Tail Latency or Makespan



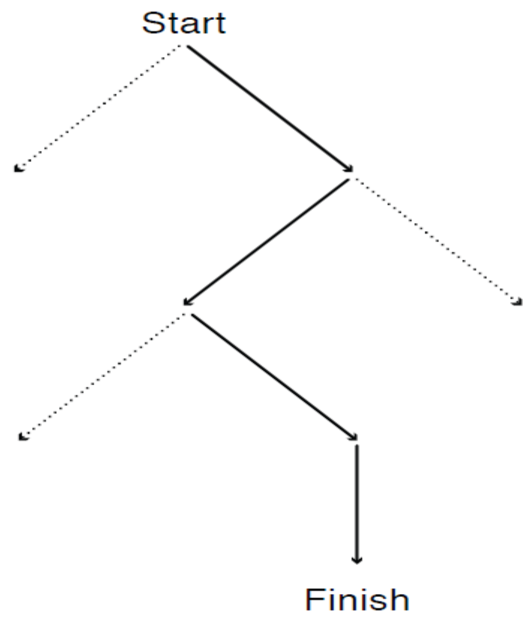
Problem: Limited by the slowest

Dependencies: Pipelines

- What can happen?



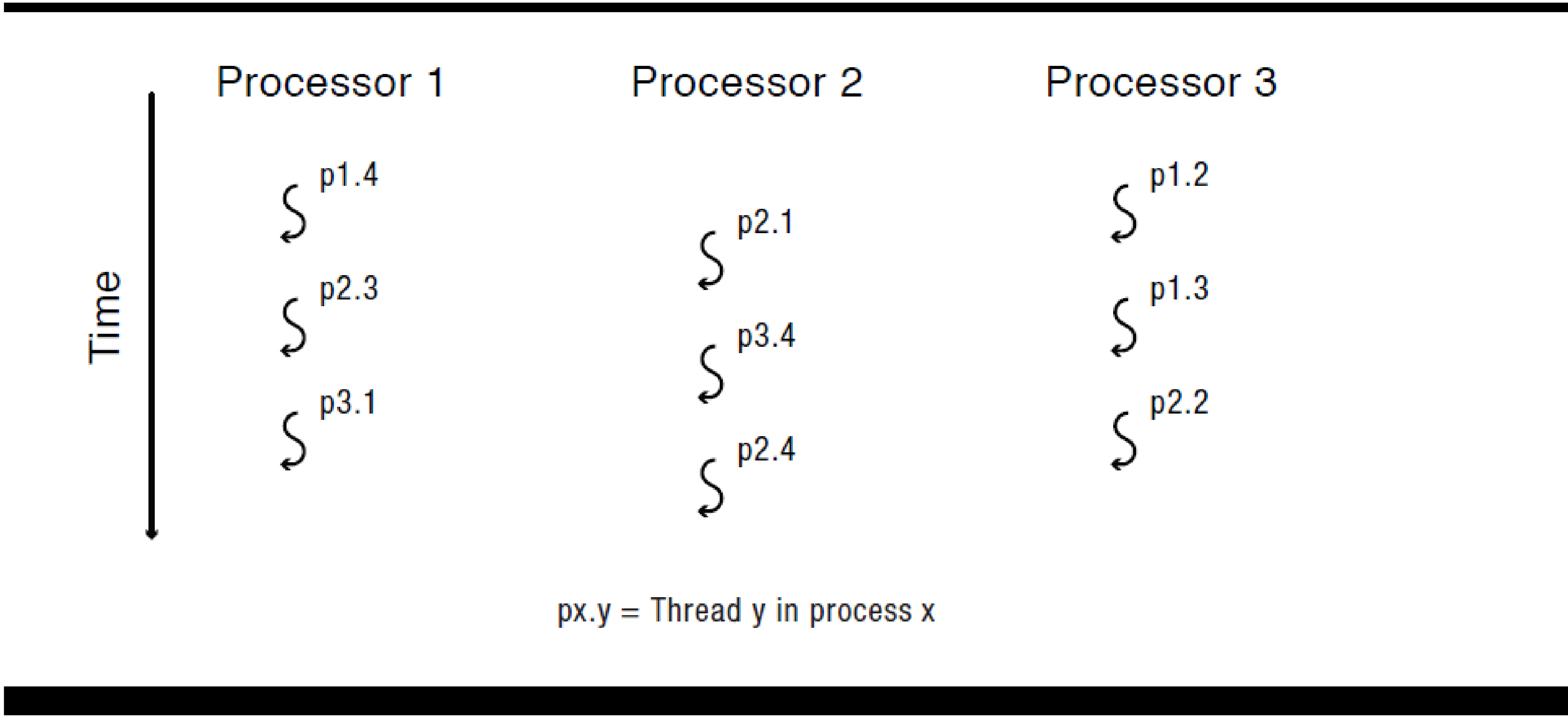
Dependencies: Critical Path Delay



What matters is the dark path – why?
Scheduling can be tricky

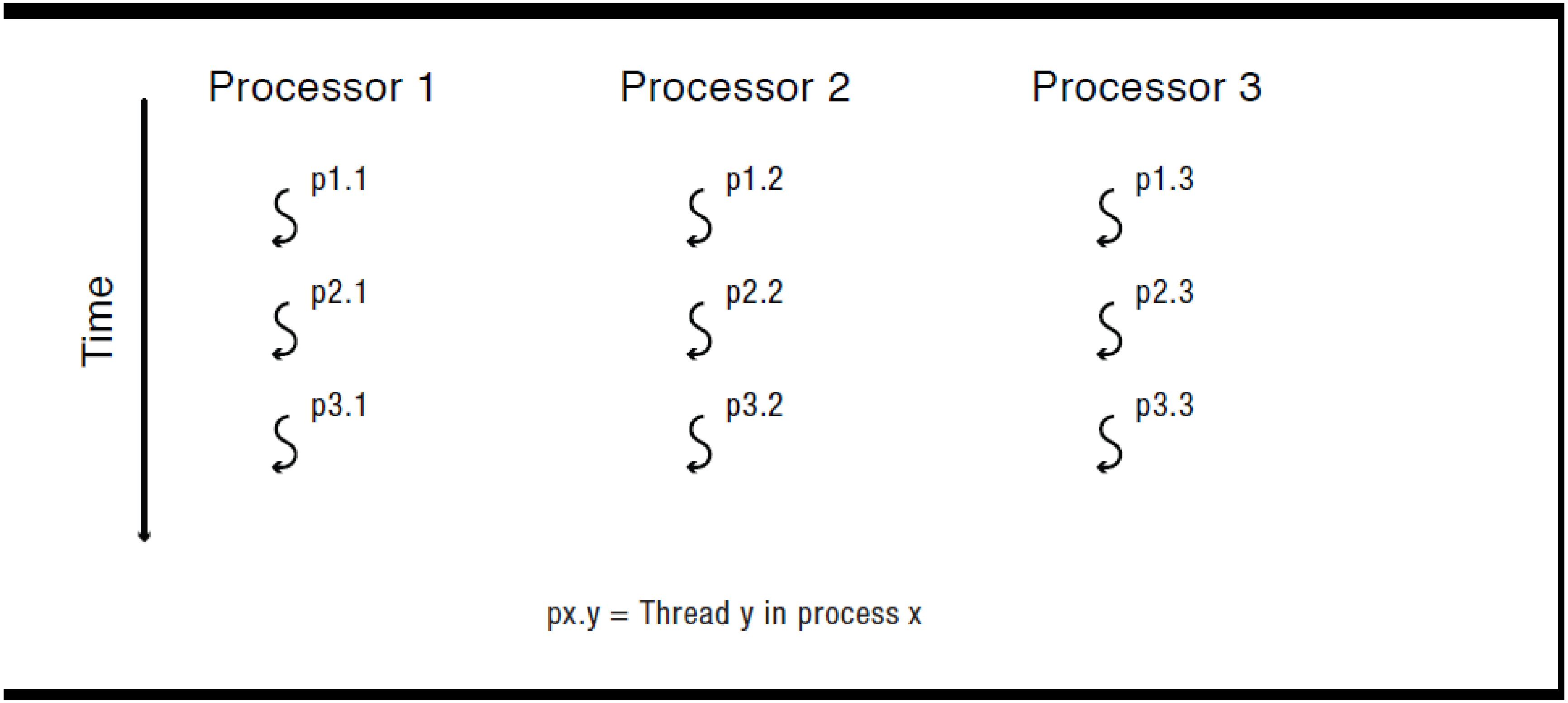
Scheduling Parallel Programs

Oblivious: each processor time-slices its ready list independently of the other processors



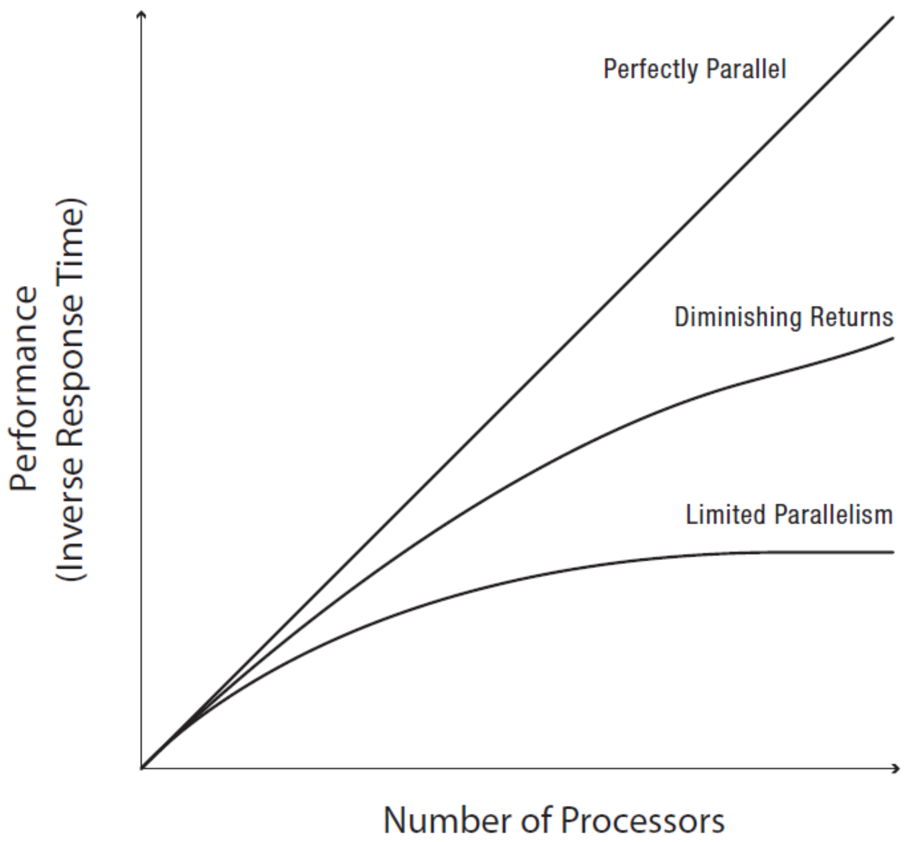
Can yield very poor tail latency: even for identical tasks/threads!

Gang Scheduling



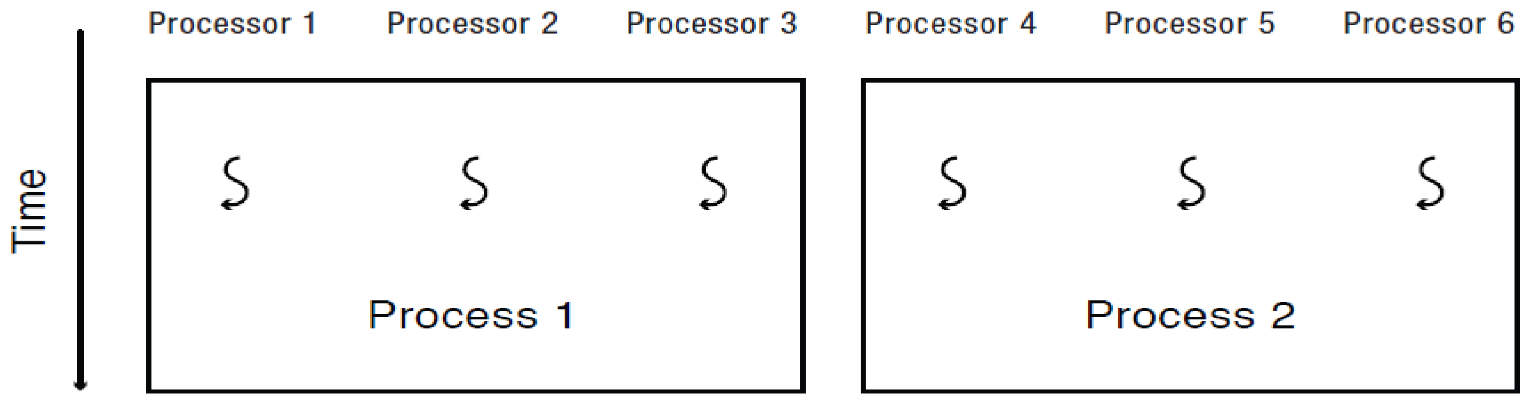
Time-slice at the level of an application: OS ensures all threads of an application run at the same time; Each app gets all processors
Problem?

Parallel Program Speedup



How many processors to use?

Space Sharing



If job can live with a smaller # of dedicated processors …
May be better than time-slicing per job

Standard practice for many years: job declares how many processors it wants (may wait) and runs to finish

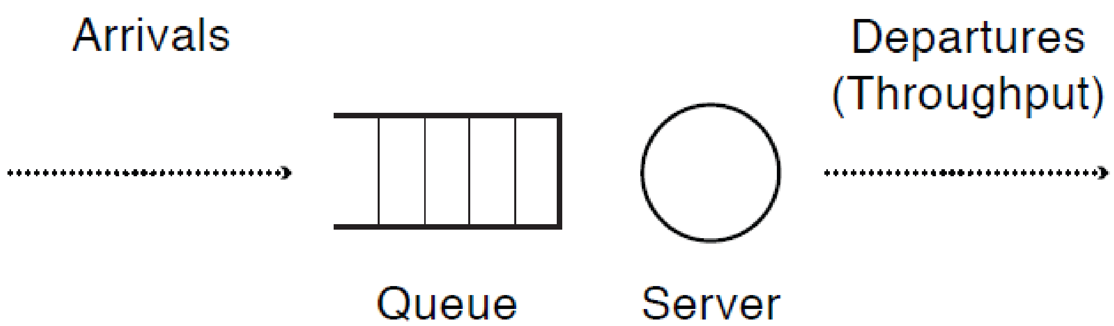
Problem?

- Solution: Backfilling

Queueing Theory

- Can we predict what will happen to user performance:
  - If a service becomes more popular?
  - If we buy more hardware?
  - If we change the implementation to provide more features?

Queueing Model



arrival rate (ƛ): tasks/time   max service/departure rate (μ) : tasks/time

FIFO, work-conserving

Assumption: average performance in a stable system, where ƛ ~= μ;   suppose ƛ > μ?
suppose ƛ < μ?

Definitions

- Queueing delay (W): wait time, avg is key
- Number of tasks queued (Q), avg is key
- Service time (S): time to service the request
  - $\mu = 1/S$ (departure rate)
- Response time (R) = W + S: improve?

Definitions

- Utilization (U): fraction of time the server is busy
  - Service time * arrival rate (ƛ)
  - S = 1 msec, ƛ = 1000 tasks/sec =>
  - S = 1 msec, ƛ = 100 tasks/sec =>
  - S = 1 sec, ƛ = 100 tasks/sec =>

- Throughput (X): actual rate of task completions
  - X = U * μ
  - If stable (no overload), throughput = arrival rate

Little's Law

Applies to *any* stable system – where arrivals match departures.

N: number of tasks in the system on average (stable system):

N = X * R

throughput (i.e. arrival rate) * avg response time

(# tasks/time * avg time)

where N ~= # on the Q and # running

what happens when R goes up?

why is knowing N useful?

Question

Suppose a system has throughput (X) = 100 tasks/s, average response time (R) = 50 ms/task

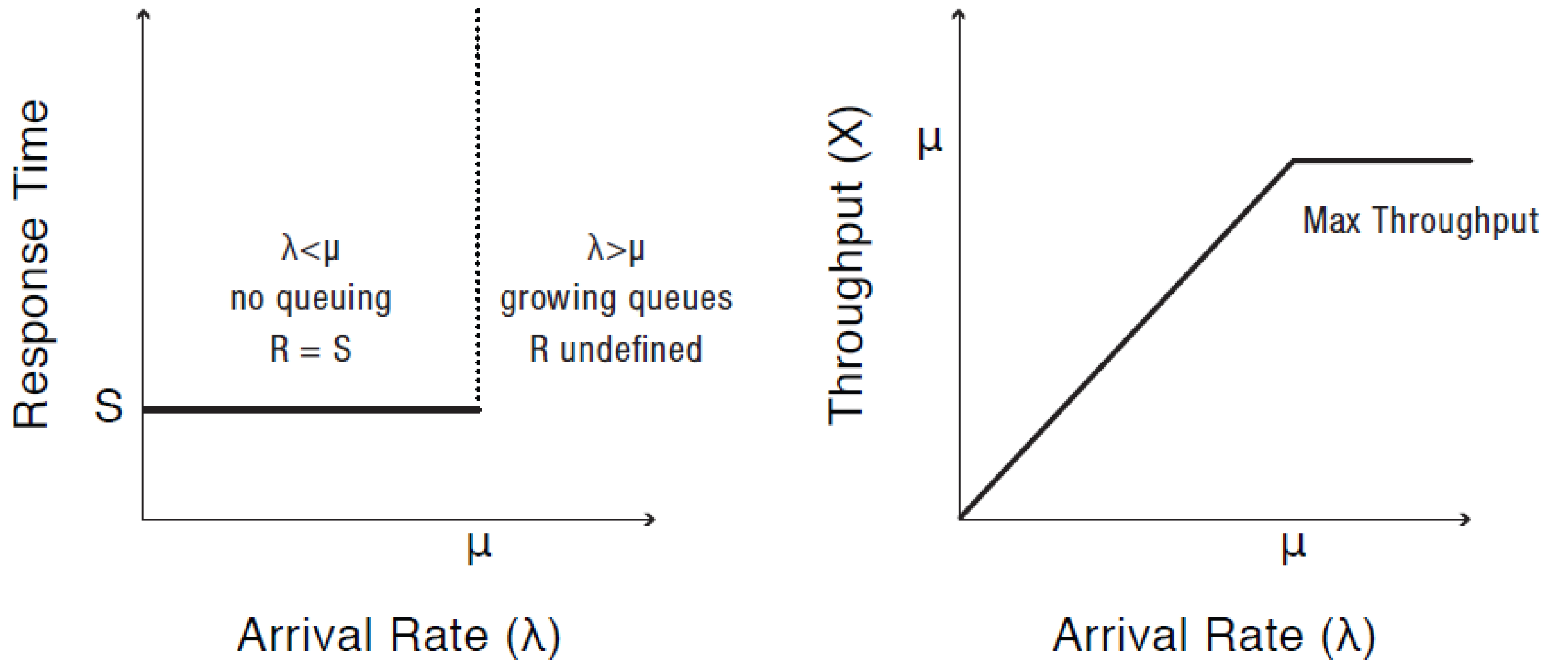
- How many tasks are in the system on average?
- If the server takes 5 ms/task, what is its utilization?
- What is the average wait time?
- What is the average number of queued tasks?

Queueing

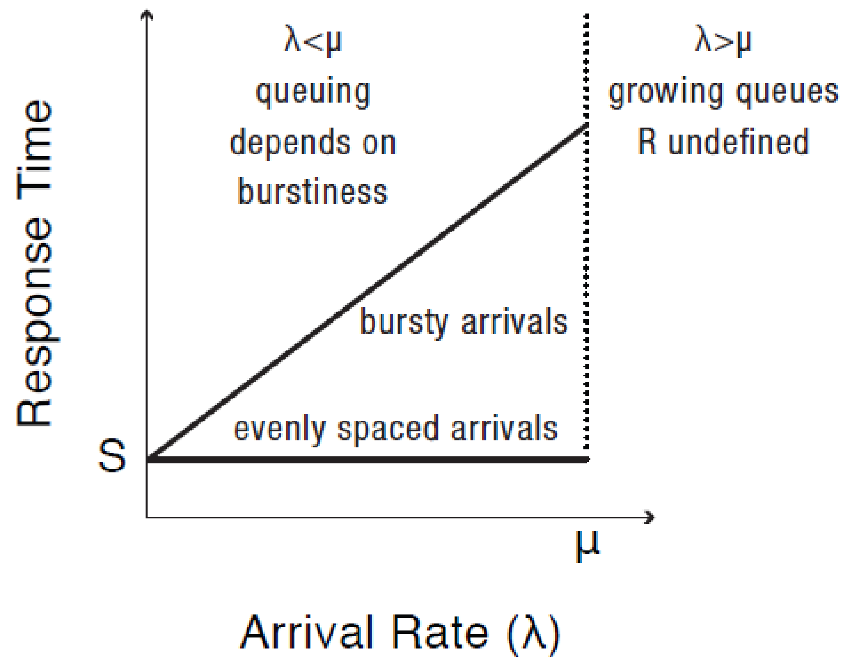
- What is the best case scenario for minimizing queueing delay (assuming $\lambda <= \mu$) ?
  - Keeping arrival rate even, service time constant, no queueing!
  - Why was there queueing in the previous example?
    - Arrivals are not uniform at small time scales: 100 tasks/sec with 5 ms service time

- When do things worsen (assuming $\lambda <= \mu$)?
  - Highly bursty arrivals

Queueing: Best Case



Response Time: Best vs. Worst Case



Next Week

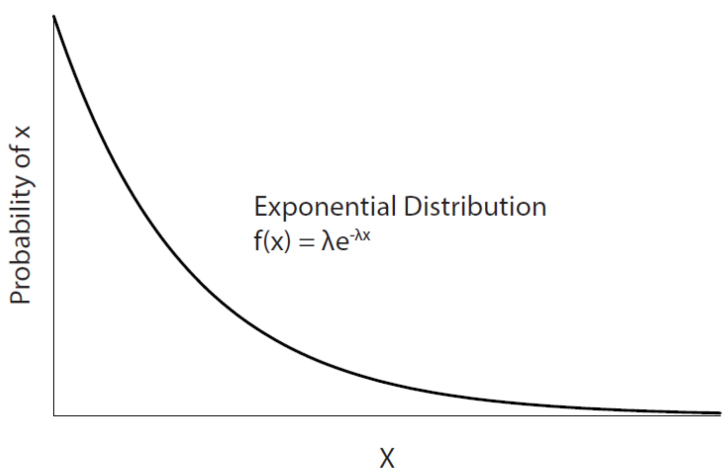
- Queuing theory
- Lottery scheduling
- Start: Address Translation - OSPP Chapter 8
- Have a great weekend!

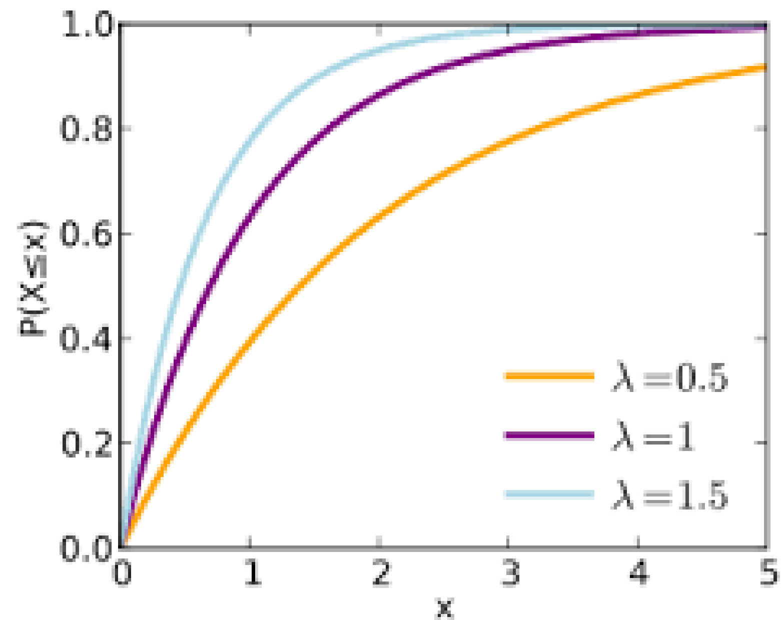
Queueing: Average Case?

- What is average?
  - Gaussian: arrivals are spread out, around a mean value
    - Longer you wait, more likely to be done
  - Exponential: same but arrivals are memoryless
  - Heavy-tailed: arrivals are very bursty
    - Longer you wait, longer you will wait
- Can have randomness (with a distribution) in both arrivals and service times

Exponential Distribution

ƛ : arrival rate
1/ƛ: mean of distribution (e.g. inter-arrival rate)





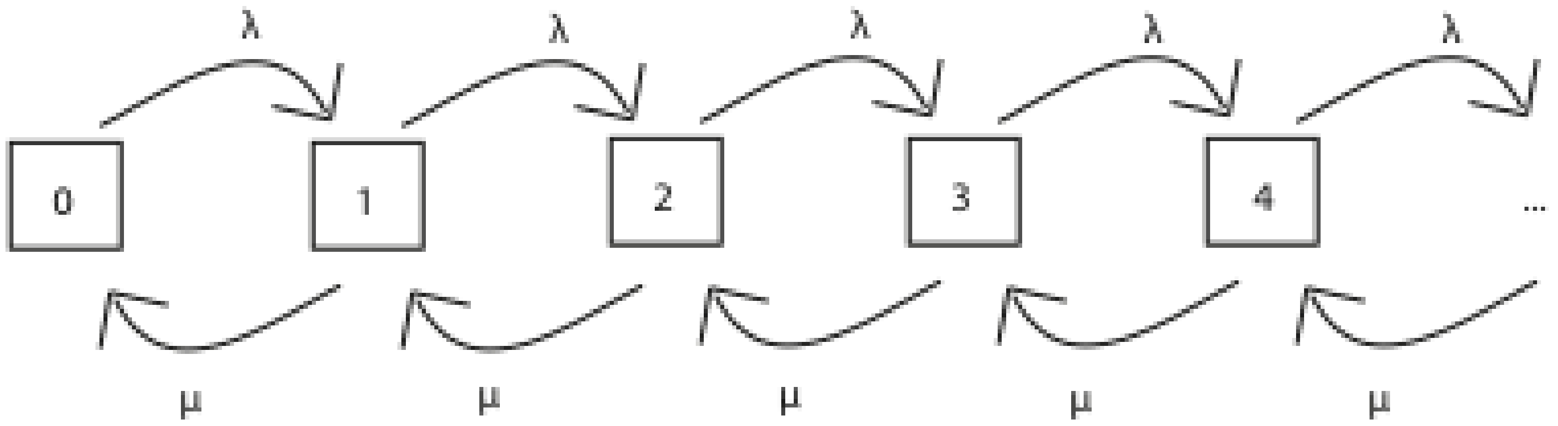
Surprisingly accurate!

$F(x) = 1 - e^{-\lambda x}$

E.g. Prob of next arrival <= 2 sec

Exponential Distribution

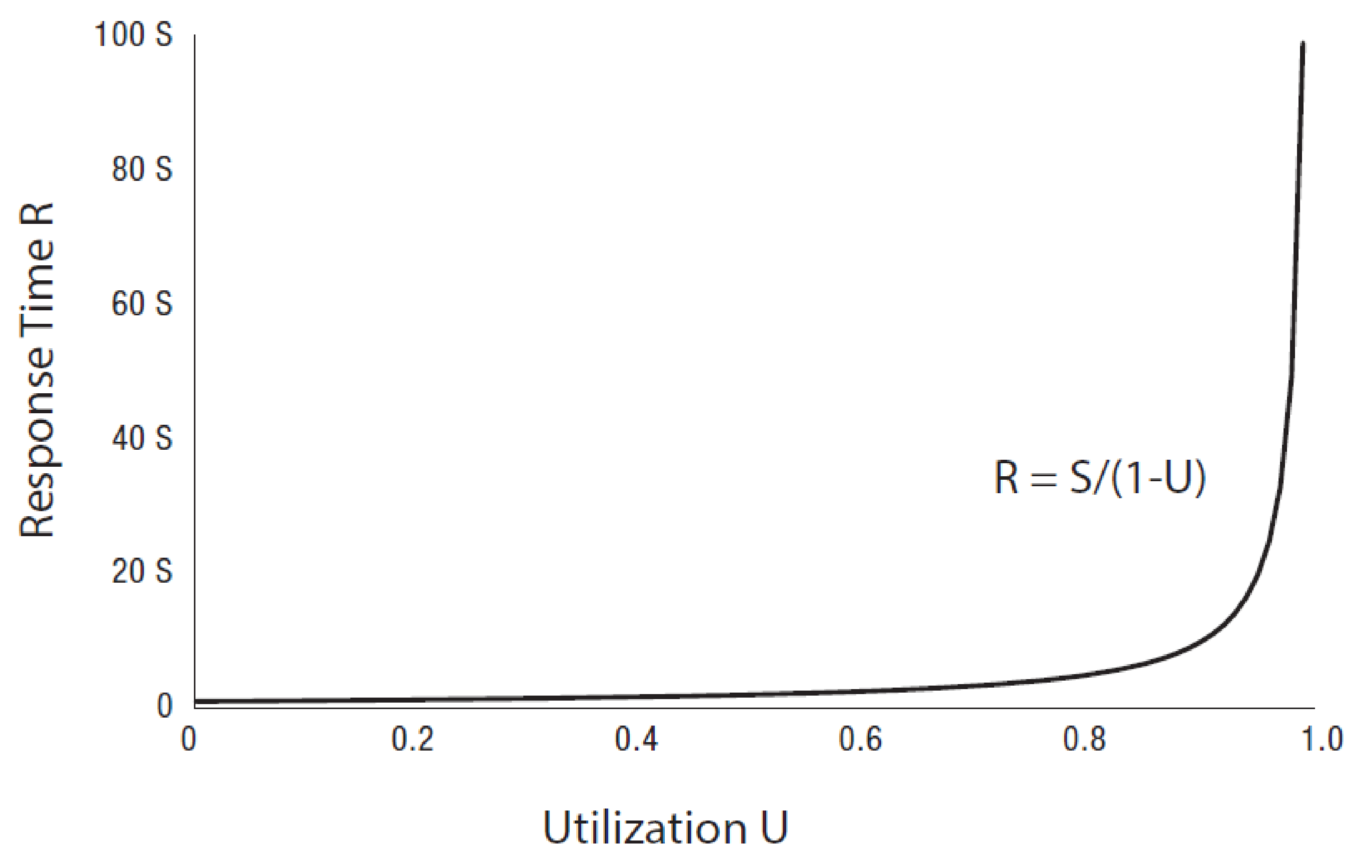
State is queue length, e.g.



Memoryless:
Probability of state transition independent of how long you have been in a particular state

Permits closed form solution to state probabilities, as function of arrival rate and service rate

Response Time vs. Utilization



For exponentially distributed arrivals (stable)

Question

- Exponential arrivals: $R = S/(1-U)$
- If system is 20% utilized, and load increases by 5%, how much does response time increase?
  - 1.25S vs. 1.33S => few percent
- If system is 90% utilized, and load increases by 5%, how much does response time increase?
  - 10S vs. 20S = 100%!
- So, the upshot is that monitoring U is key

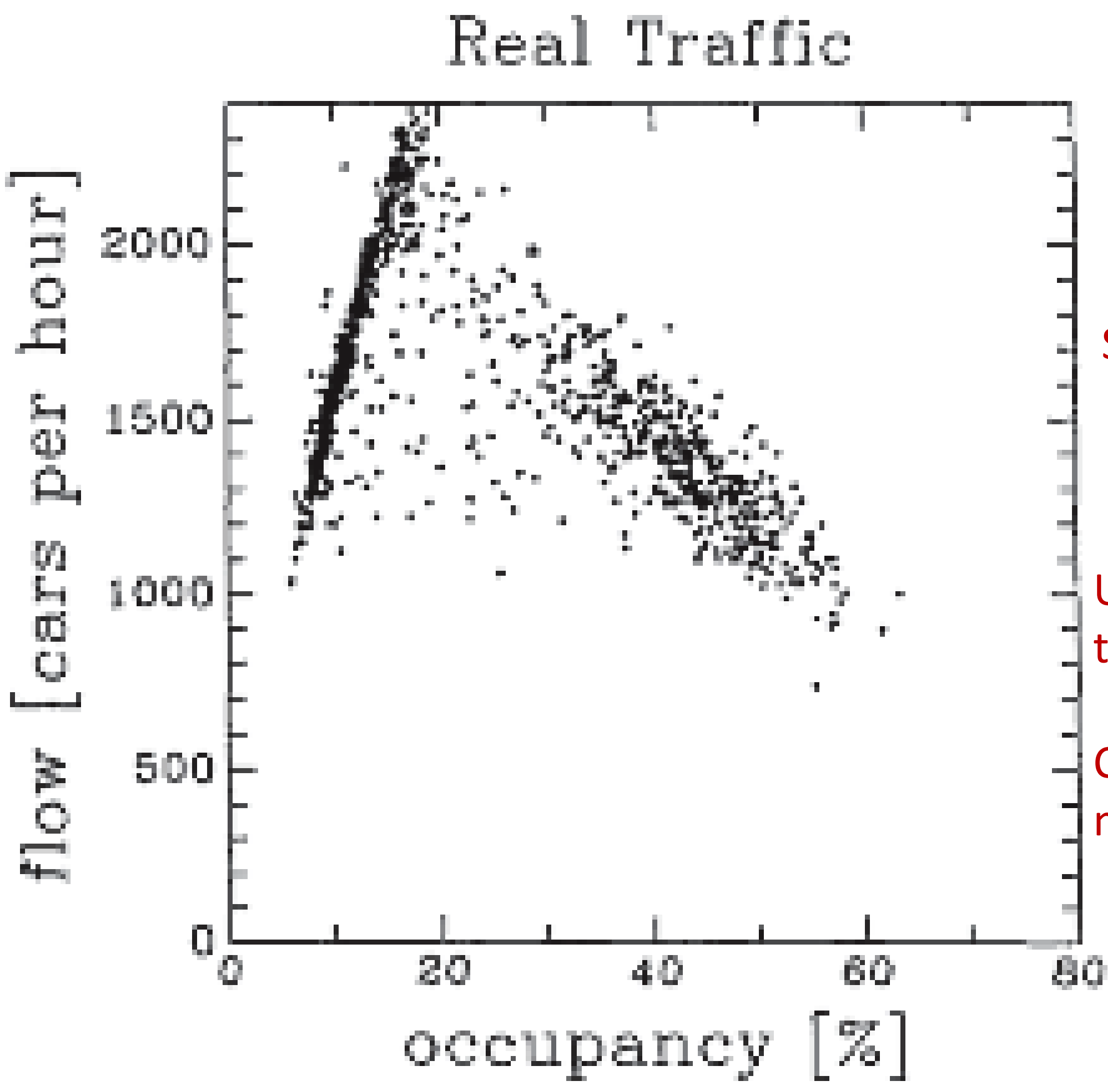
What if Multiple Resources?

- Response time =
  $\Sigma S_i/(1-U_i)$ over all i resources needed assuming seq.
  - network bandwidth, disk I/O, CPU, … (web request)
- Implication
  - If you fix one bottleneck, the next highest utilized resource will limit performance
  - Doubling # of CPUs may not half response time

Overload Management

- What if arrivals occur faster than service can handle them
  - If do nothing, response time will become infinite
- Turn users away?
  - Which ones? Average response time is best if turn away users that have the highest service demand
- Degrade service?
  - Compute result with fewer resources
  - Example: CNN static front page on 9/11

Highway Congestion (measured)



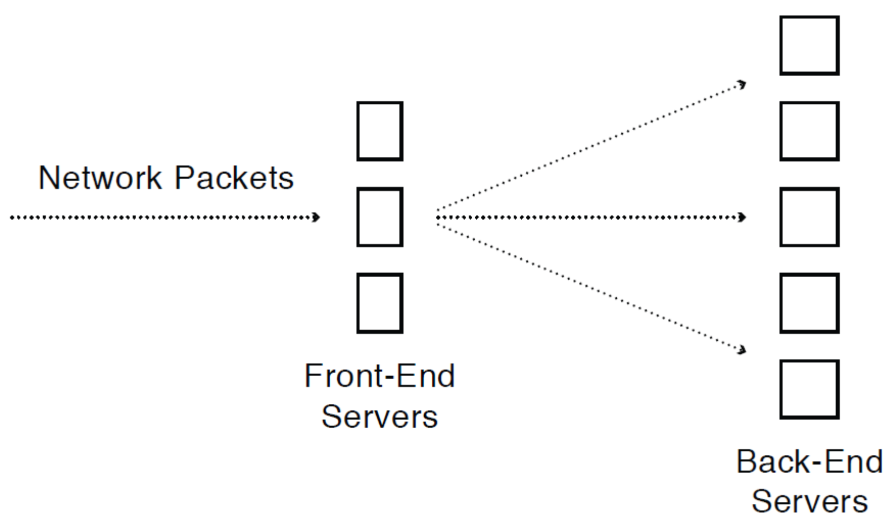
Solution: on ramps

Unlike best case, throughput collapse!

Can happen for multithreaded servers

Data Center Case Study

- Load balance requests
- Affinities
- SJF +/ Fair share
- If sustained U gets too high, provision more
- Ideally, try to predict increase in U



Scheduling

Chapter 7 OSPP

Part III: Lottery Scheduling

(much shortened)

Scheduling Issues

- Context
  - multiple scarce resources: CPU, I/O bw, mem
  - concurrently executing clients (~ tasks)
  - service requests of varying importance and characteristics

- Quality of Service needs differ
  - editor, video playback, compilation, simulation, …

Conventional Scheduling

- We know that SJF does not reflect needs per-se and has other problems, as does FIFO, as does RR
- Priority Scheduling
  - what does it really mean?
  - does p=1 vs. p=2 mean p=1 always gets the CPU or just 2/3?
- Problems
  - often ad hoc
  - unable to control service rates to tasks

Solution: Lottery Scheduling

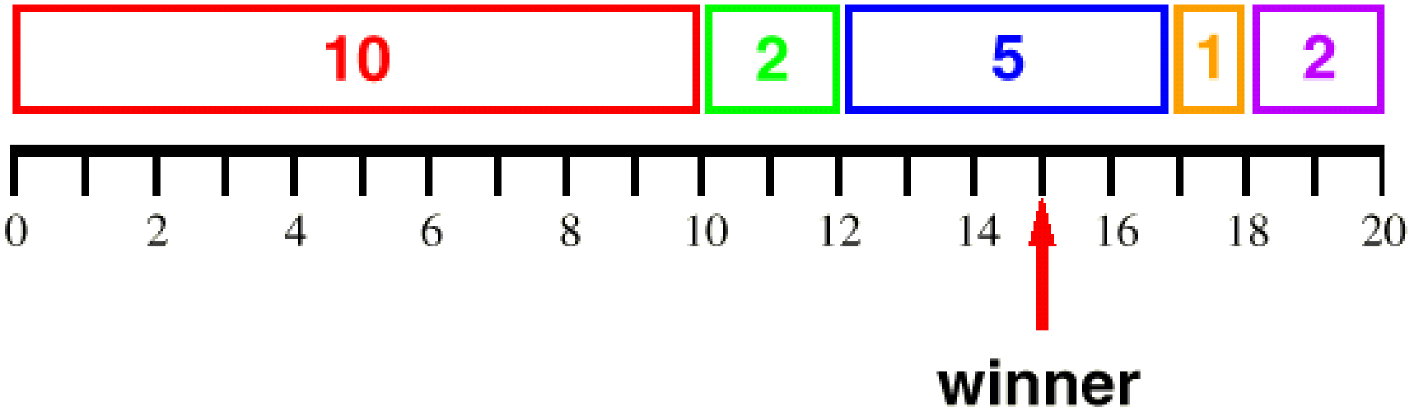
- Easily Understood Behavior
  - proportional share
- Flexible Control Over Service Rates
  - current schedulers are rigid (e.g. RR-> fixed Q)
- No starvation
  - hold a non-zero # of tickets

Lottery Scheduling Basics

- Randomized Mechanism
- Lottery Tickets
  - encapsulate resource rights
  - issued in different amounts
- Lotteries
  - randomly select winning ticket
  - grant resource to client/task holding winning ticket

Example Lottery

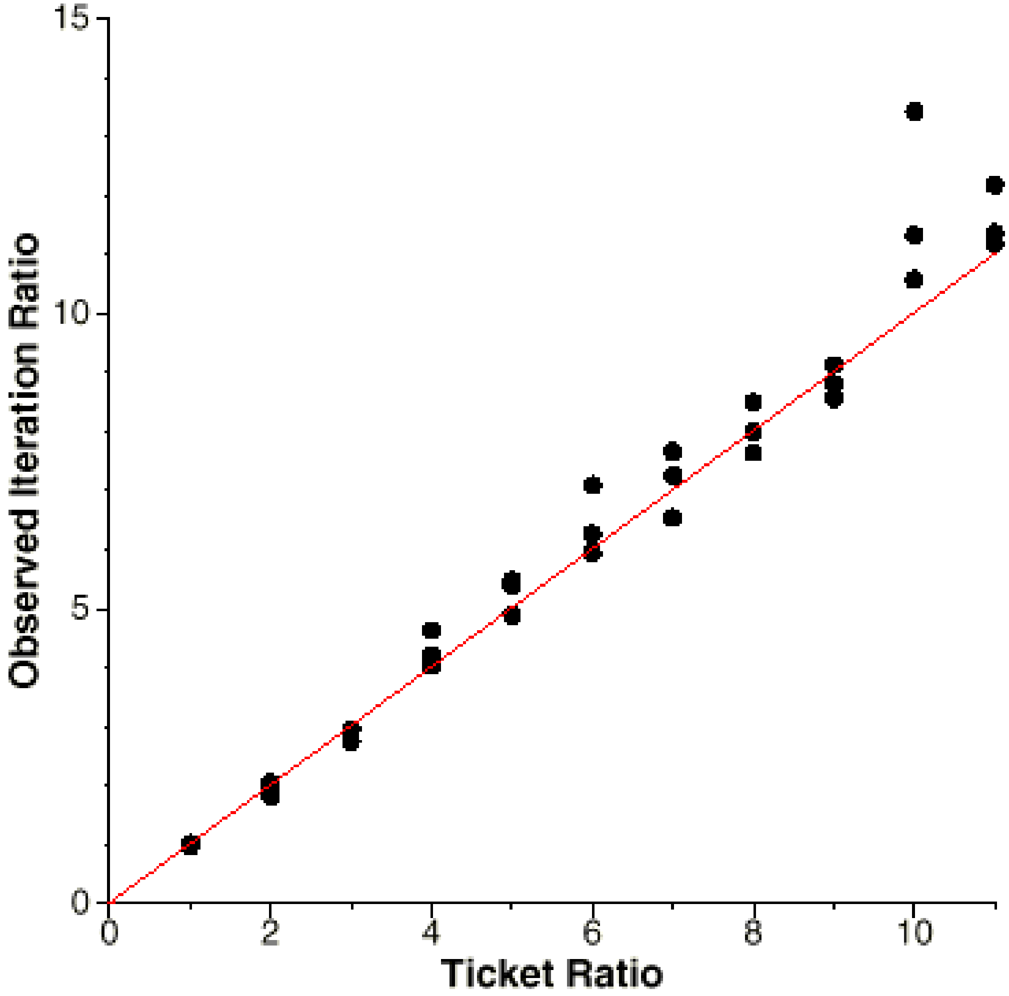
total = 20
random [1 .. 20] = 15



Lottery Scheduling Advantages

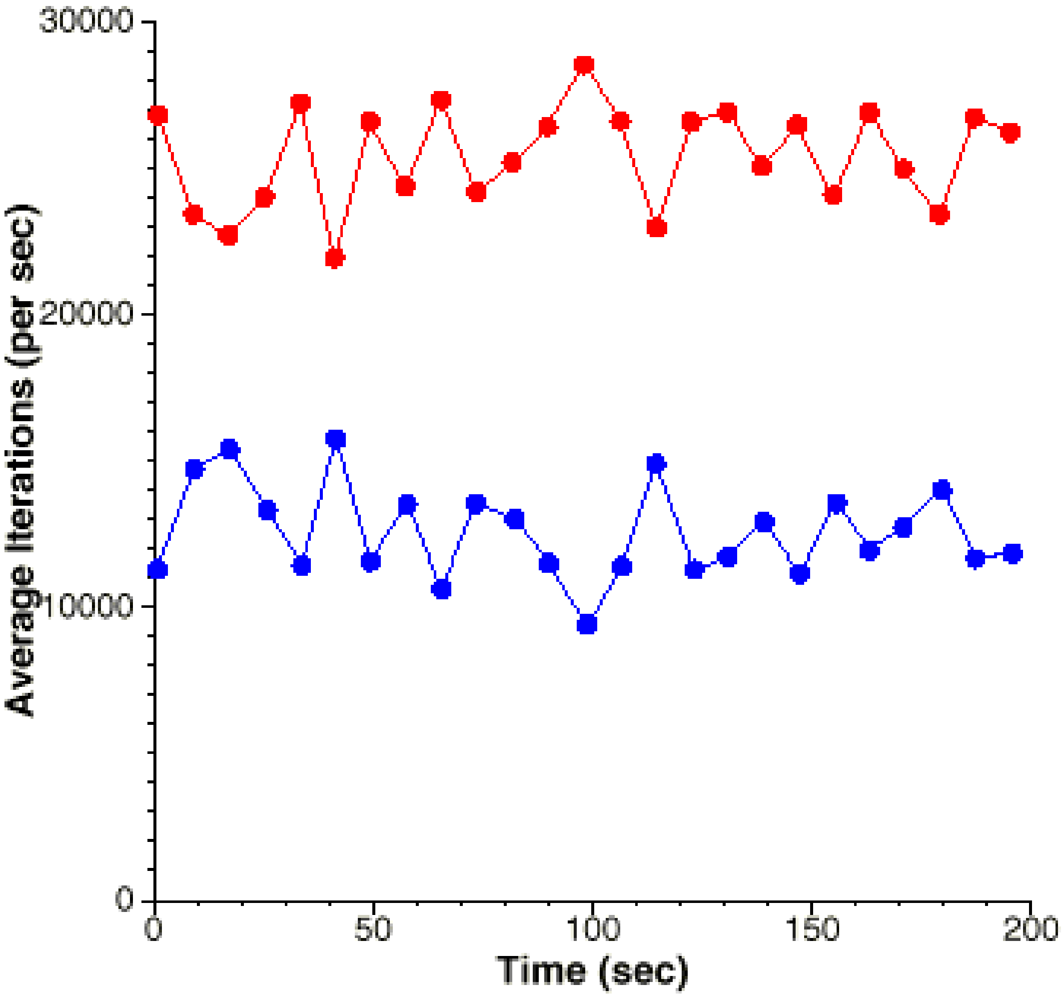
- Probabilistic Guarantees
  - n lotteries, client holds t tickets, T total tickets
  - $p = t/T$ (prob. of winning = binomial distribution)
  - throughput proportional to ticket allocation
    - $E[w] = np$ (how many lotteries I will win)
  - response time (# of lotteries b4 winning) inversely proportional to ticket allocation
    - $E[n] = 1/p$

Relative Rates



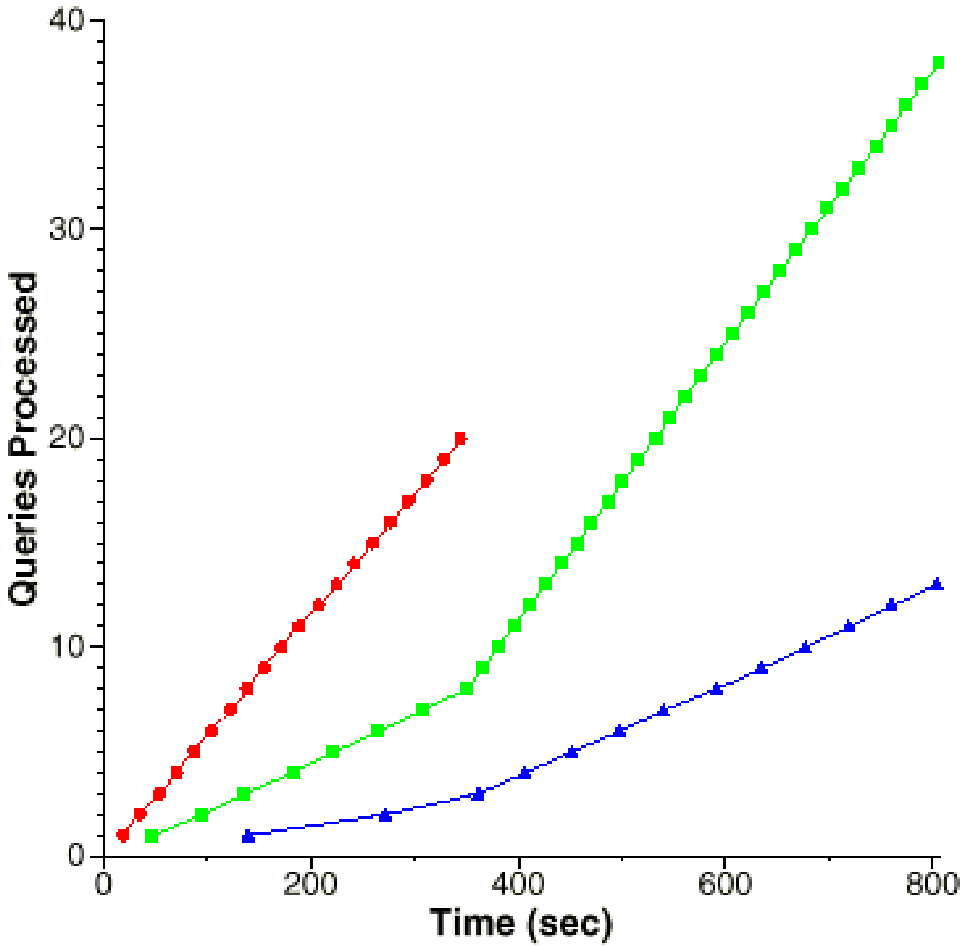
- Dhrystone benchmark
- two tasks
- three 60-second runs for each ratio

Fairness Over Time



- Dhrystone benchmark
- two tasks
- 2 : 1 allocation
- 8-second averages

Query Processing Rates



- multithreaded "database" server
- three clients
- 8 : 3 : 1 allocation
- ticket transfers

Lottery-Scheduled Locks

- Waiting to Acquire
  - waiters transfer funding to lock owner
  - lock owner inherits aggregate funding to acquire CPU
- Release
  - return funding to waiters
  - hold lottery among waiters
  - new winner inherits funding
- Avoids Priority Inversion

Lock Experiment

- Groups of threads A, B with 2:1 Allocation
- Acquire, Hold 50 ms, Release, Compute 50 ms
- Average Waiting Time
  - A waits 450 ms, B waits 948 ms
  - 1:2.11 response time ratio
- Lock Acquisitions
  - A completes 763, B completes 423
  - 1.80 : 1 throughput

Next Time

- Address Translation
- OSPP Chapter 8