Scheduling

Chapter 7 OSPP

Part I

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 …
- 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?

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?

Definitions

- Task/Job
  - User request: e.g., mouse click, web request, shell command, …
- Latency/response time
  - How long does a task take to complete?
- Throughput
  - How many tasks can be done per unit of time?
- 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?

More Definitions

- Workload
  - Set of tasks for system to perform
- 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
- Scheduling algorithm
  - takes a workload as input
  - decides which tasks to do first
  - Performance metric (throughput, latency) 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

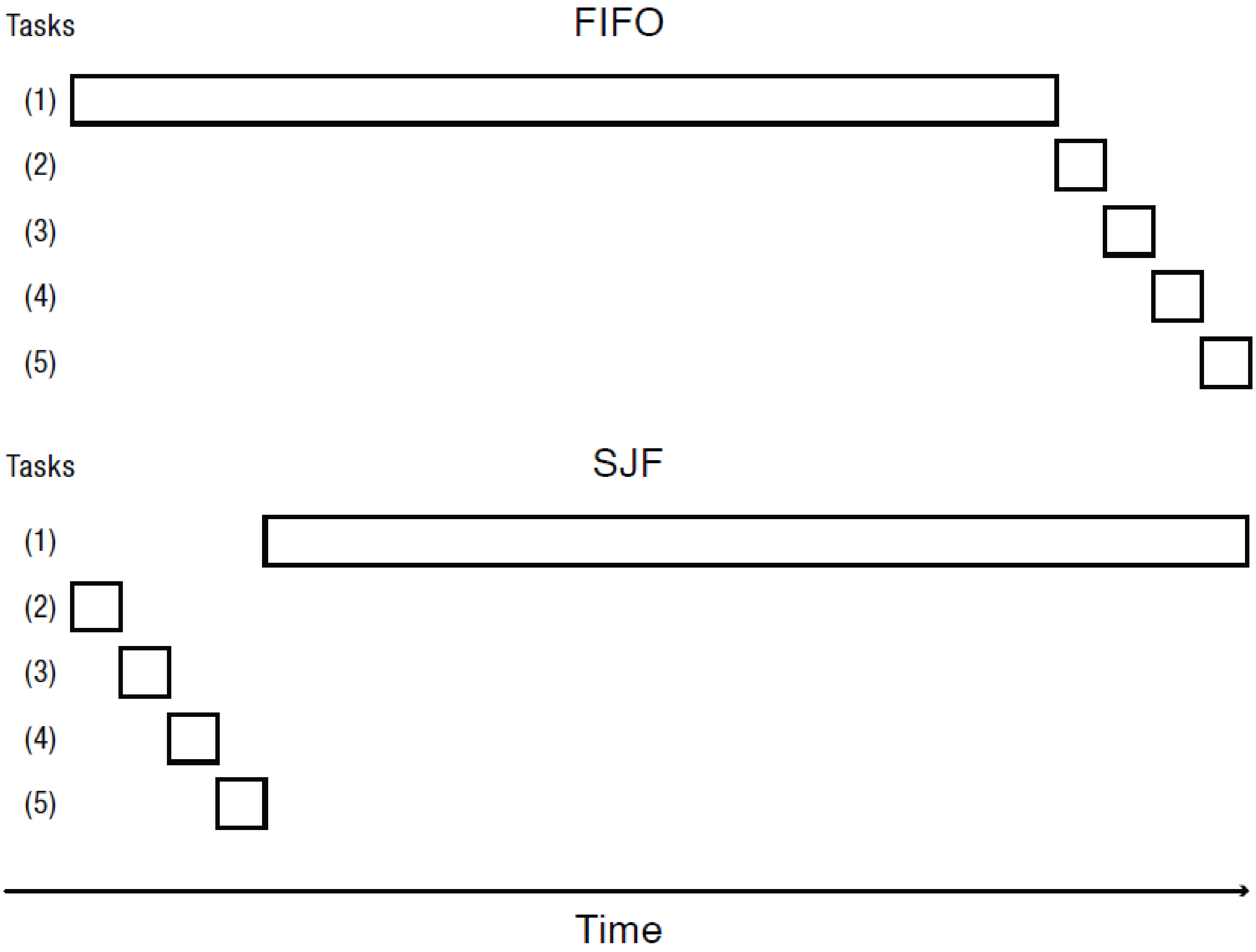
- On what workloads is FIFO particularly bad?

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? Next?
  - Which completes first in SJF? Next?

FIFO vs. SJF



Question

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

- Does SJF have any downsides?

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
  - Examples?

Question

- Is FIFO ever optimal?
  - Yes, when all requests are of equal length
- Why is it good?

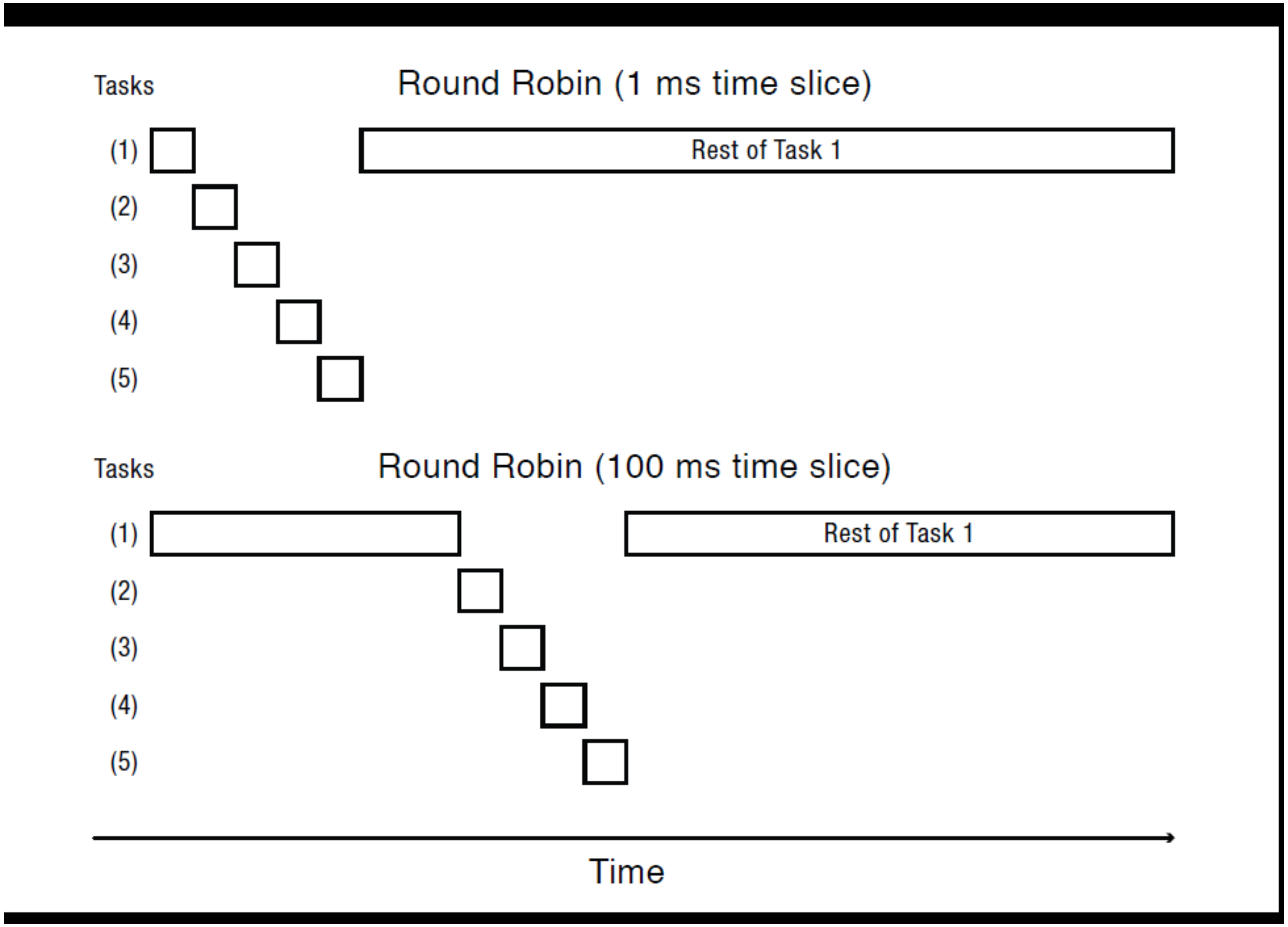
Starvation and Sample Bias

- 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

Round Robin

- Each task gets resource for a fixed period of time (time quantum)
  - If task doesn’t complete, it goes back in line
- Need to pick a time quantum
  - What if time quantum is too long?
    - Infinite?
  - What if time quantum is too short?
    - One instruction?

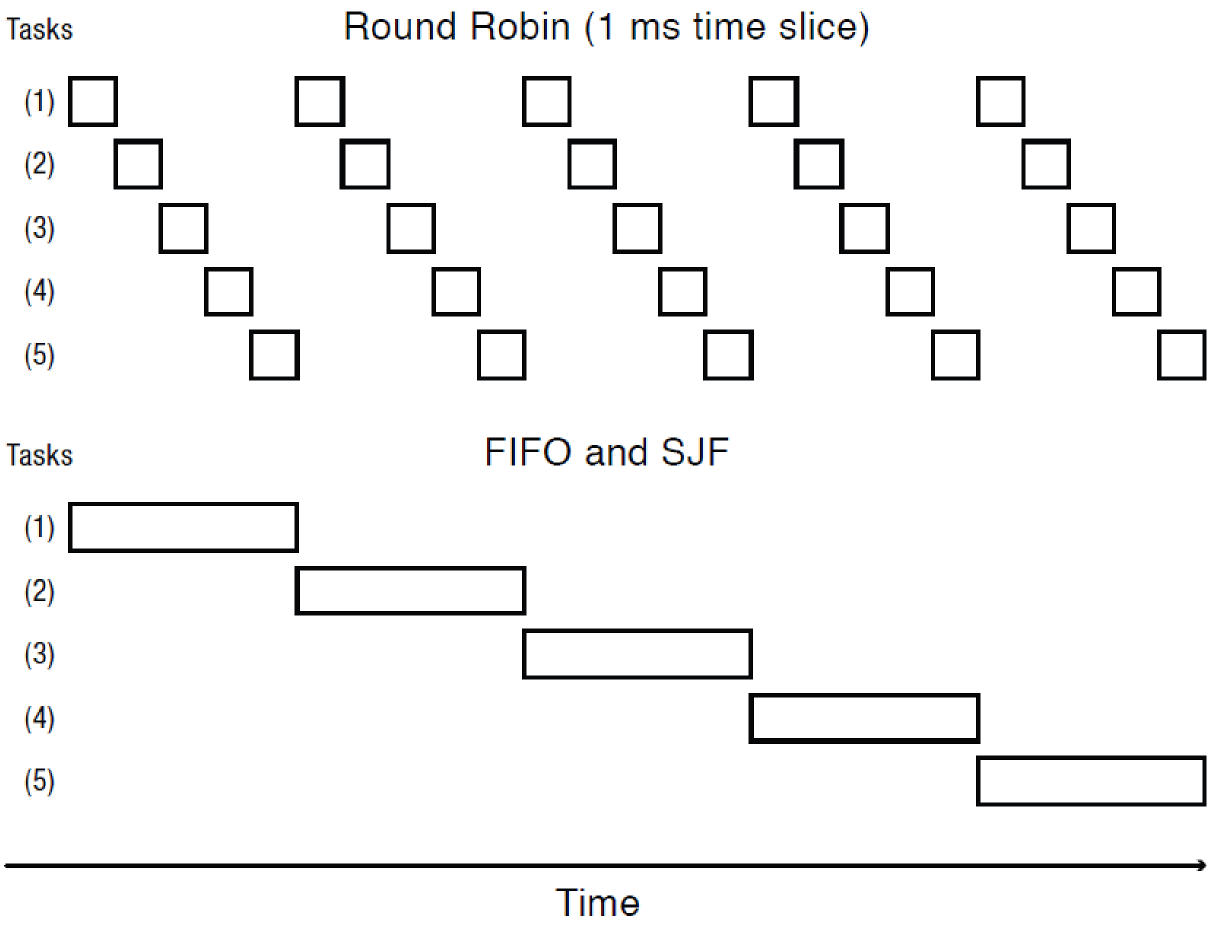
Round Robin


Tasks
Round Robin (1 ms time slice)
(1)
Rest of Task 1
(2)
(3)
(4)
(5)
Tasks
Round Robin (100 ms time slice)
(1)
Rest of Task 1
(2)
(3)
(4)
(5)
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
- Round robin for video streaming
  - Even for equal size streams this maintains stable progress for all

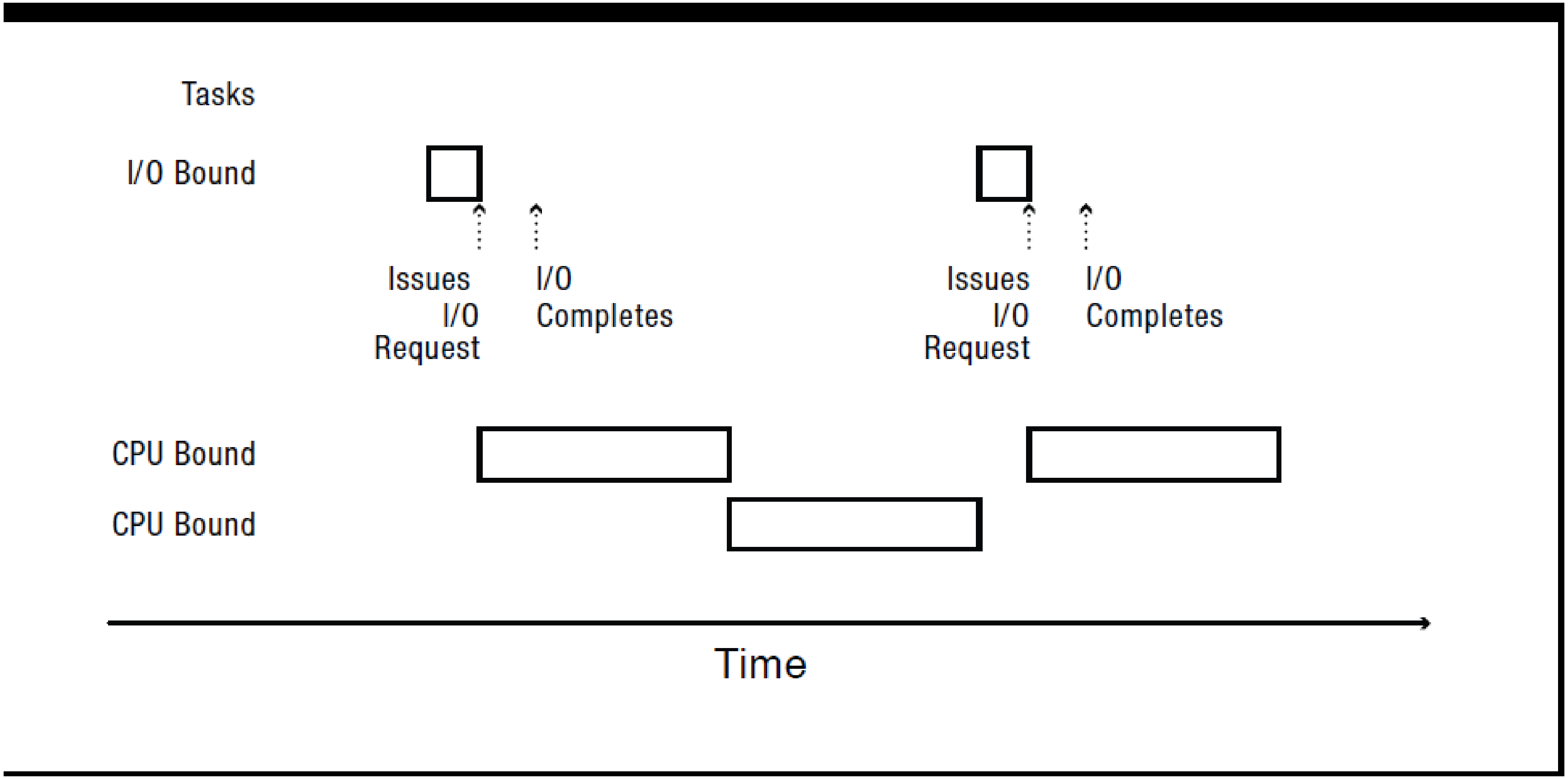
Round Robin vs. FIFO



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
    - Time task takes under scheduling algorithm

Mixed Workload



Max-Min Fairness

- How do we balance a mixture of repeating tasks:
  - Some I/O bound, need only a little CPU
  - Some compute bound, can use as much CPU as they are assigned
- One approach: maximize the minimum allocation given to a task
  - If any task needs less than an equal share, schedule the smallest of these first
  - Split the remaining time using max-min
  - If all remaining tasks need at least equal share, split evenly

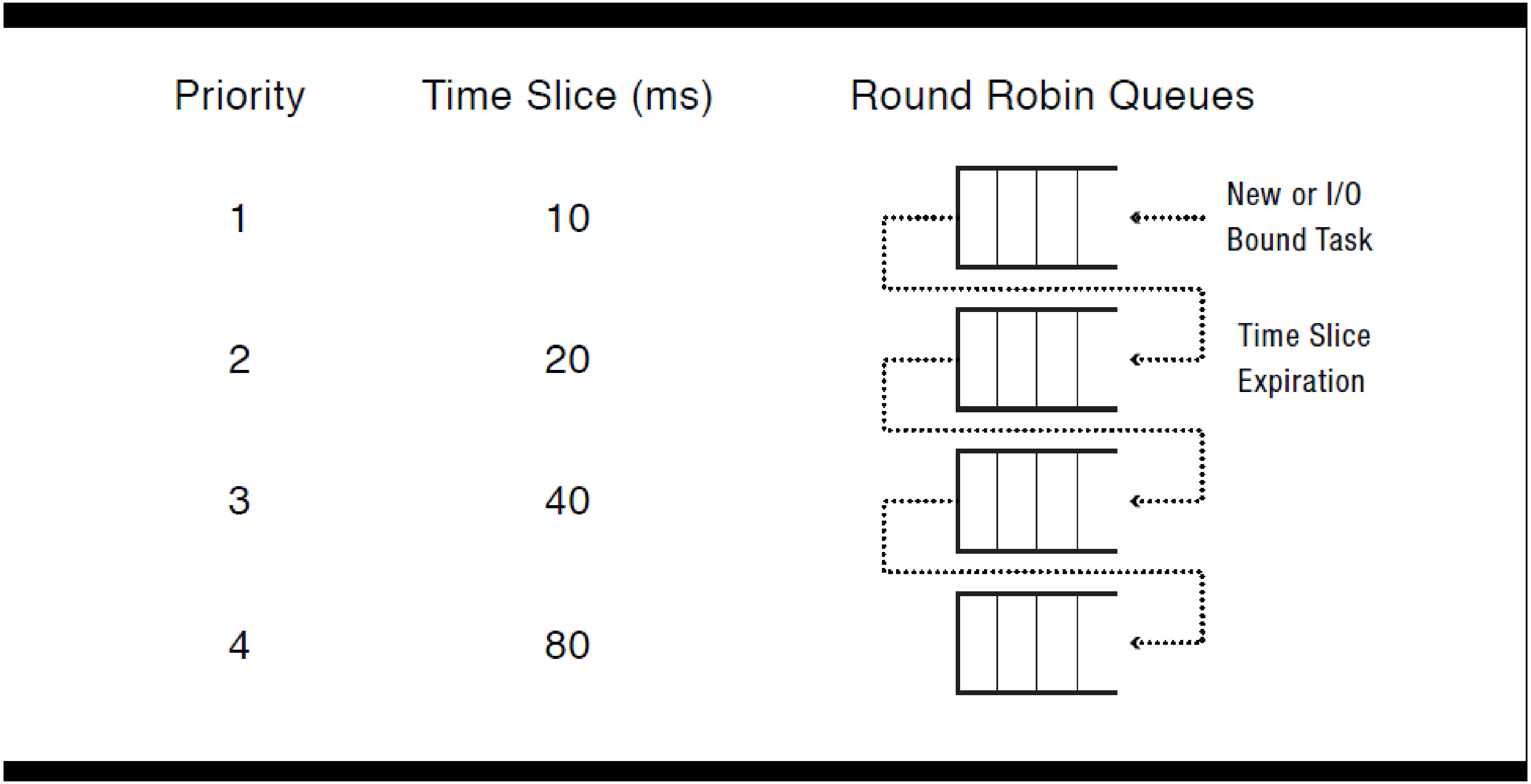
Multi-level Feedback Queue (MFQ)

- Goals:
  - Responsiveness
  - Low overhead
  - Starvation freedom
  - Some tasks are high/low priority
  - Fairness (among equal priority tasks)
- Not perfect at any of them!
  - Used in Linux

MFQ

- Set of Round Robin queues
  - Each queue has a separate priority
- 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)
1	10
2	20
3	40
4	80

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.
- Considering only the processor, 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.
- 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

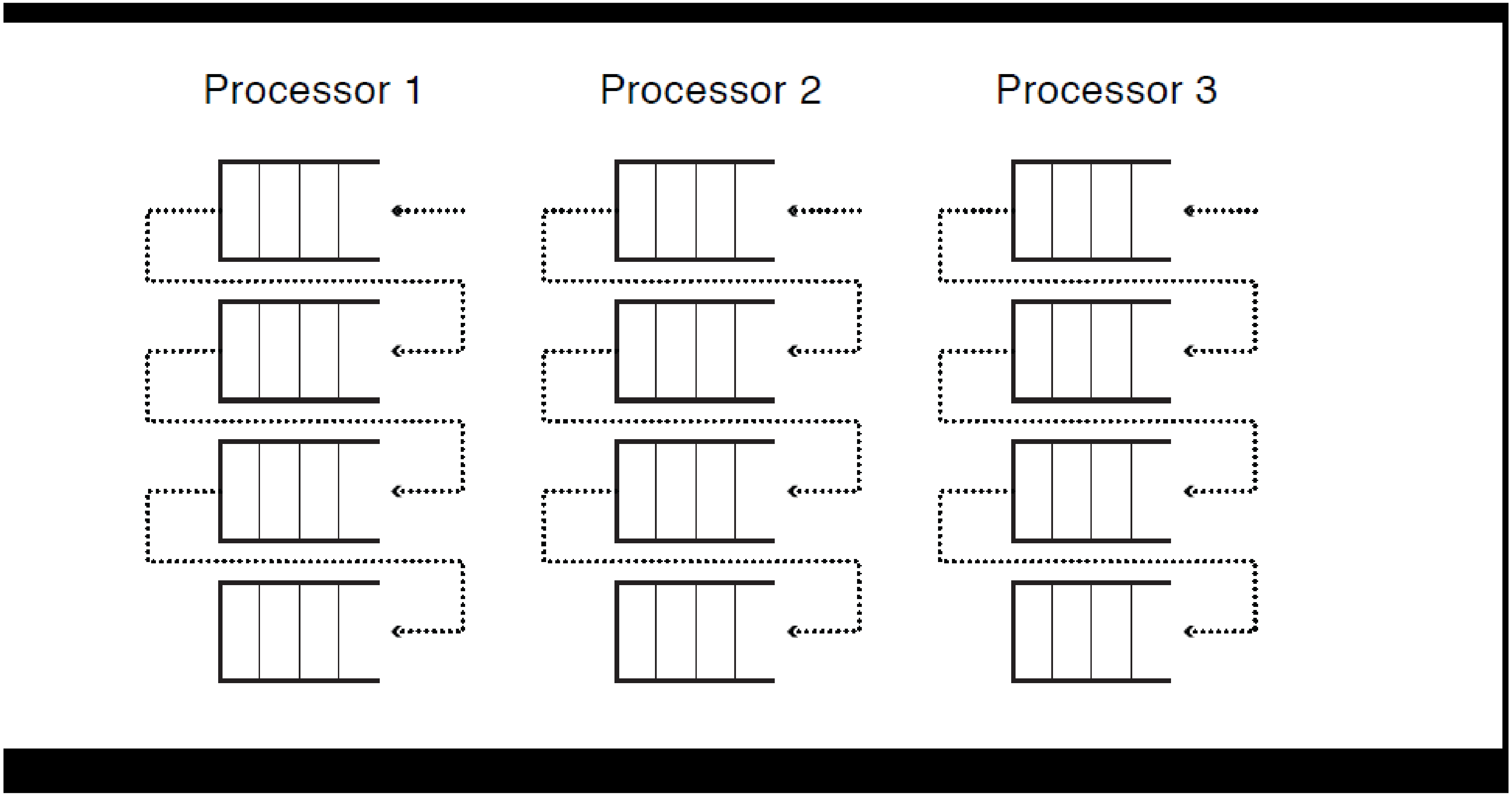
Multiprocessor Scheduling

- 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
  - Ex: when I/O completes, or on Condition->signal
- Idle processors can steal work from other processors

Per-Processor Multi-level Feedback with Affinity Scheduling



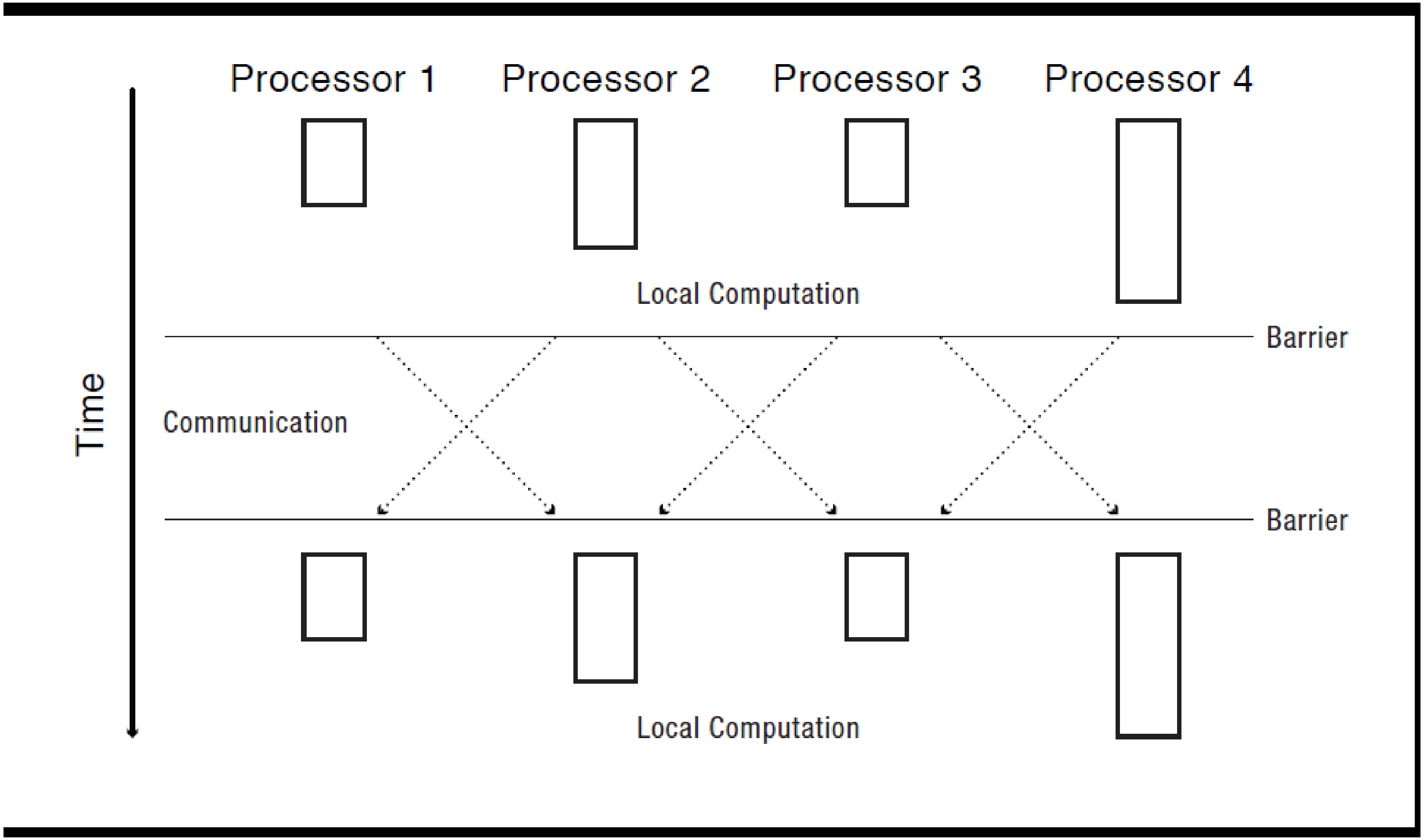
Scheduling Parallel Programs

- 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

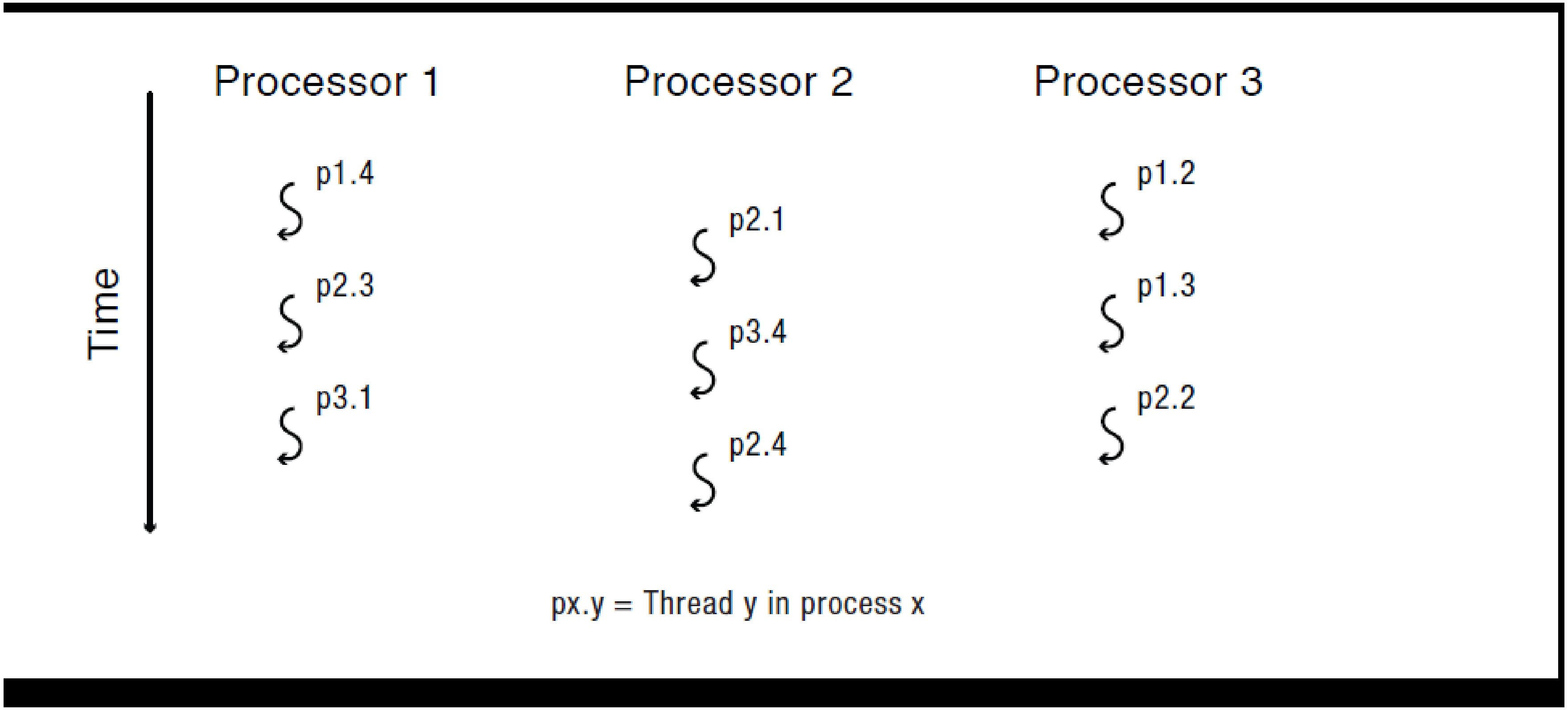
- Loop at each processor:
  - Compute on local data (in parallel)
  - Barrier
  - Send (selected) data to other processors (in parallel)
  - Barrier
- Examples:
  - MapReduce
  - Fluid flow over a wing
  - Most parallel algorithms can be recast in BSP

Tail Latency

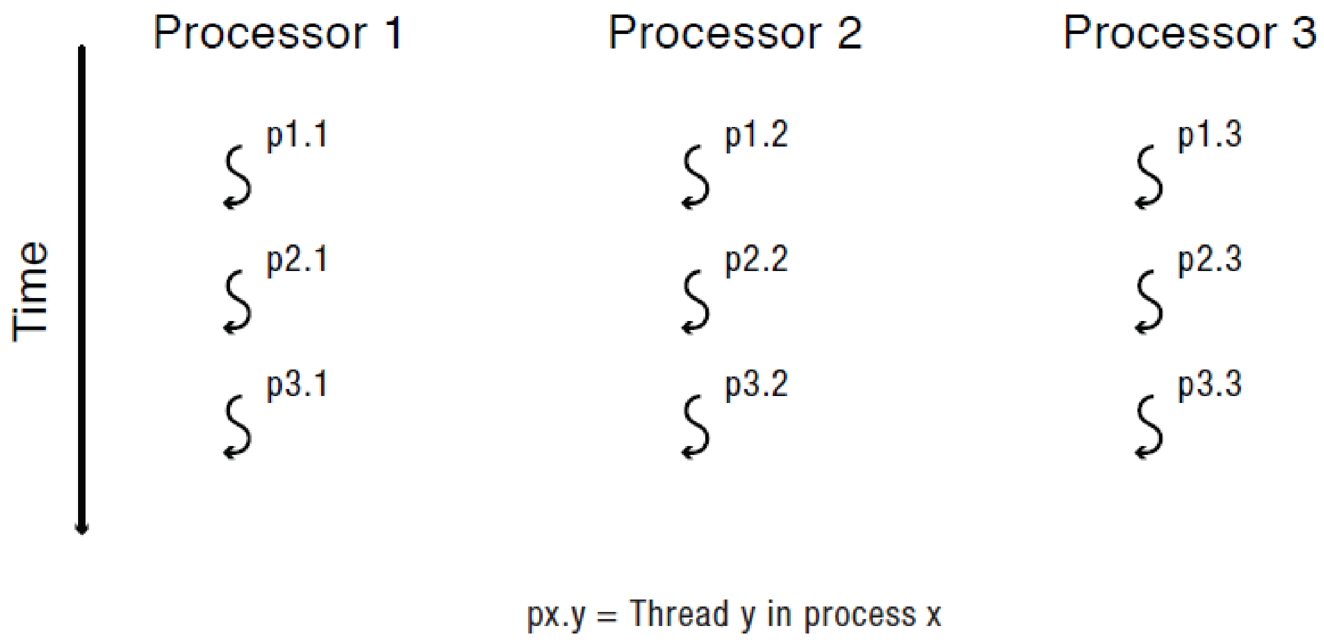


Scheduling Parallel Programs

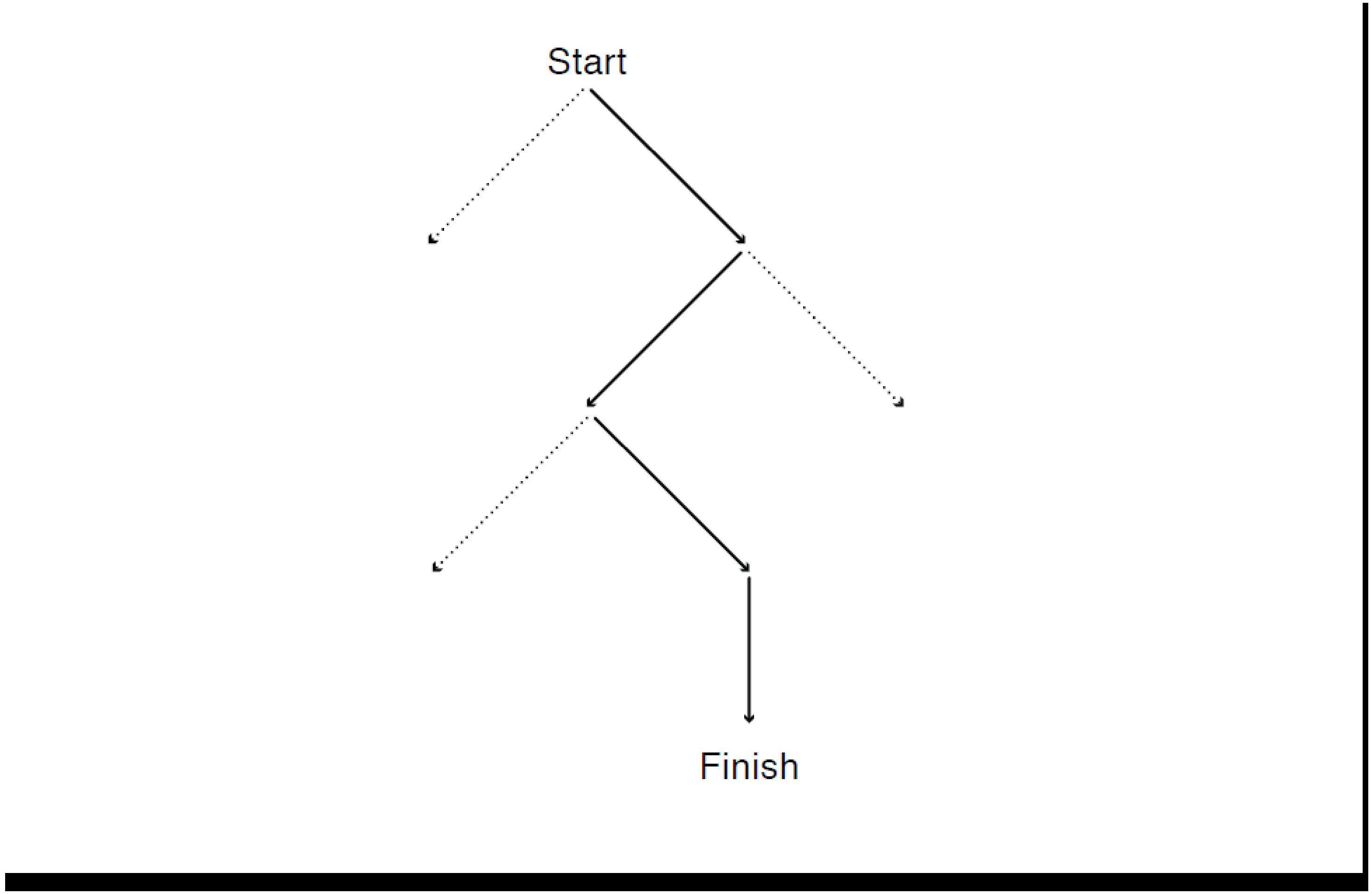
Oblivious: each processor time-slices its ready list independently of the other processors



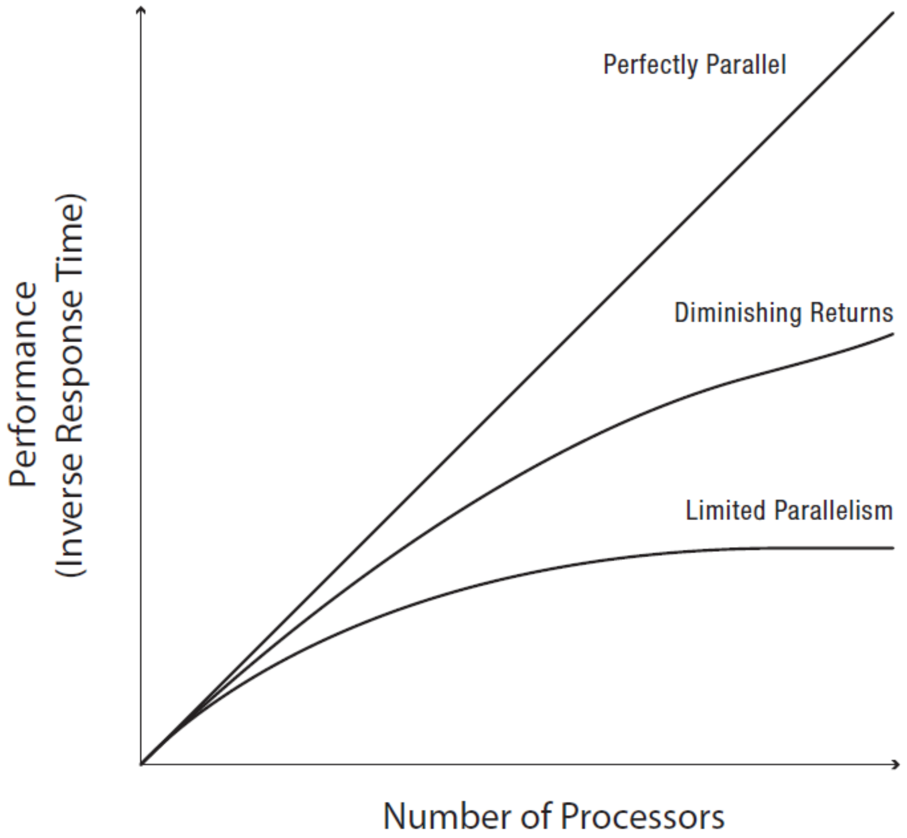
Gang Scheduling



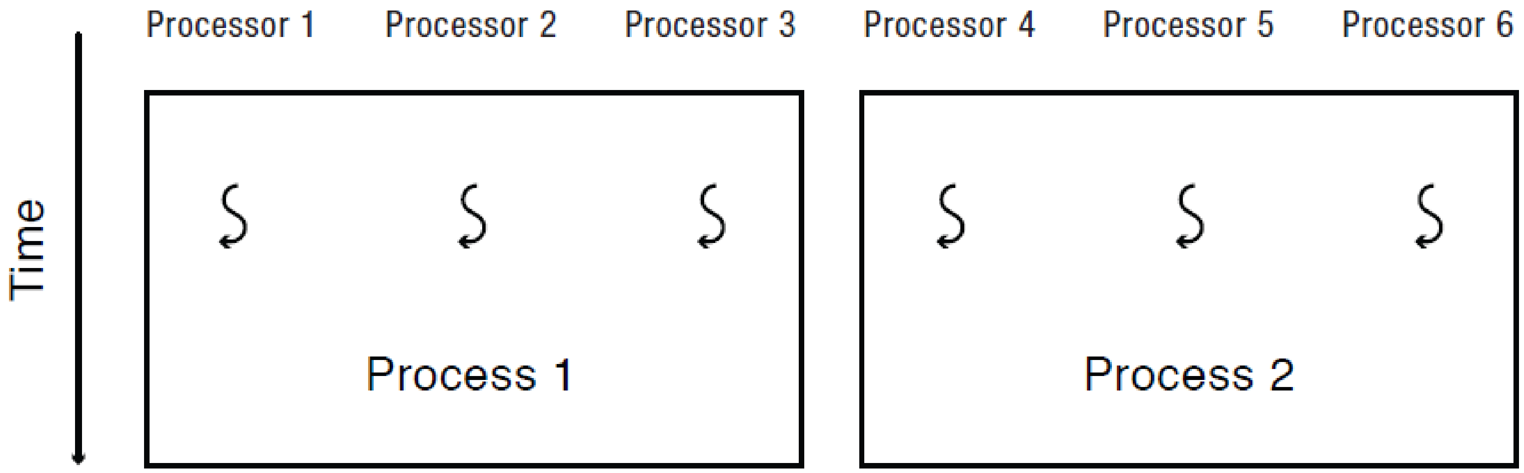
Critical Path Delay



Parallel Program Speedup



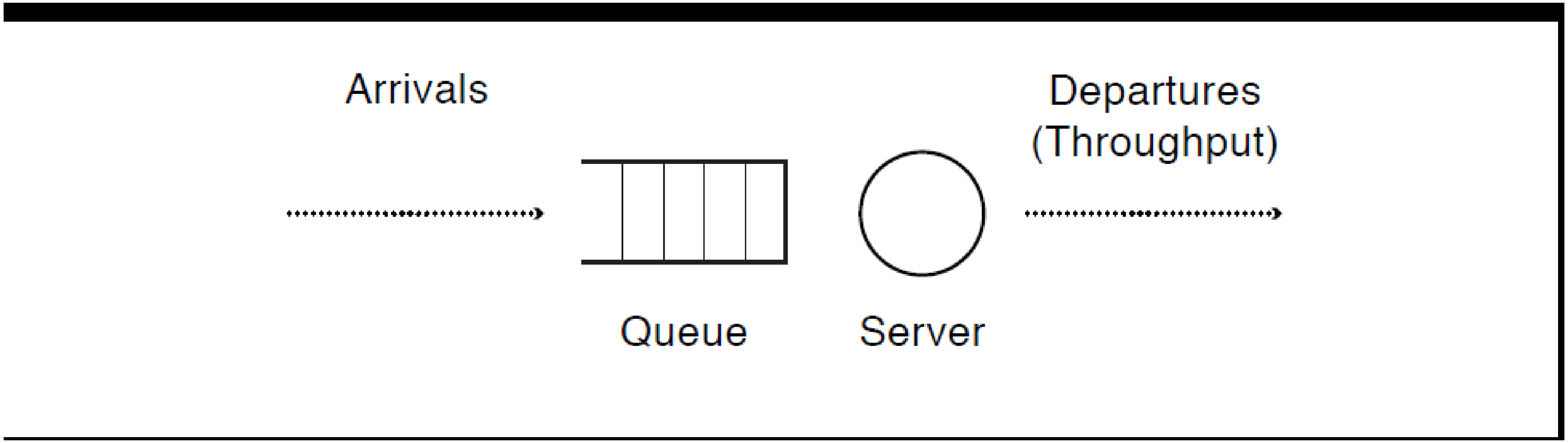
Space Sharing



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



Assumption: average performance in a stable system, where the arrival rate (ƛ) matches the departure rate (μ)

Definitions

- Queueing delay (W): wait time
  - Number of tasks queued (Q)
- Service time (S): time to service the request
- Response time (R) = queueing delay + service time
- Utilization (U): fraction of time the server is busy
  - Service time * arrival rate (ƛ)
- Throughput (X): rate of task completions
  - If no overload, throughput = arrival rate

Little’s Law

$$N = X * R$$

N: number of tasks in the system

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

Question

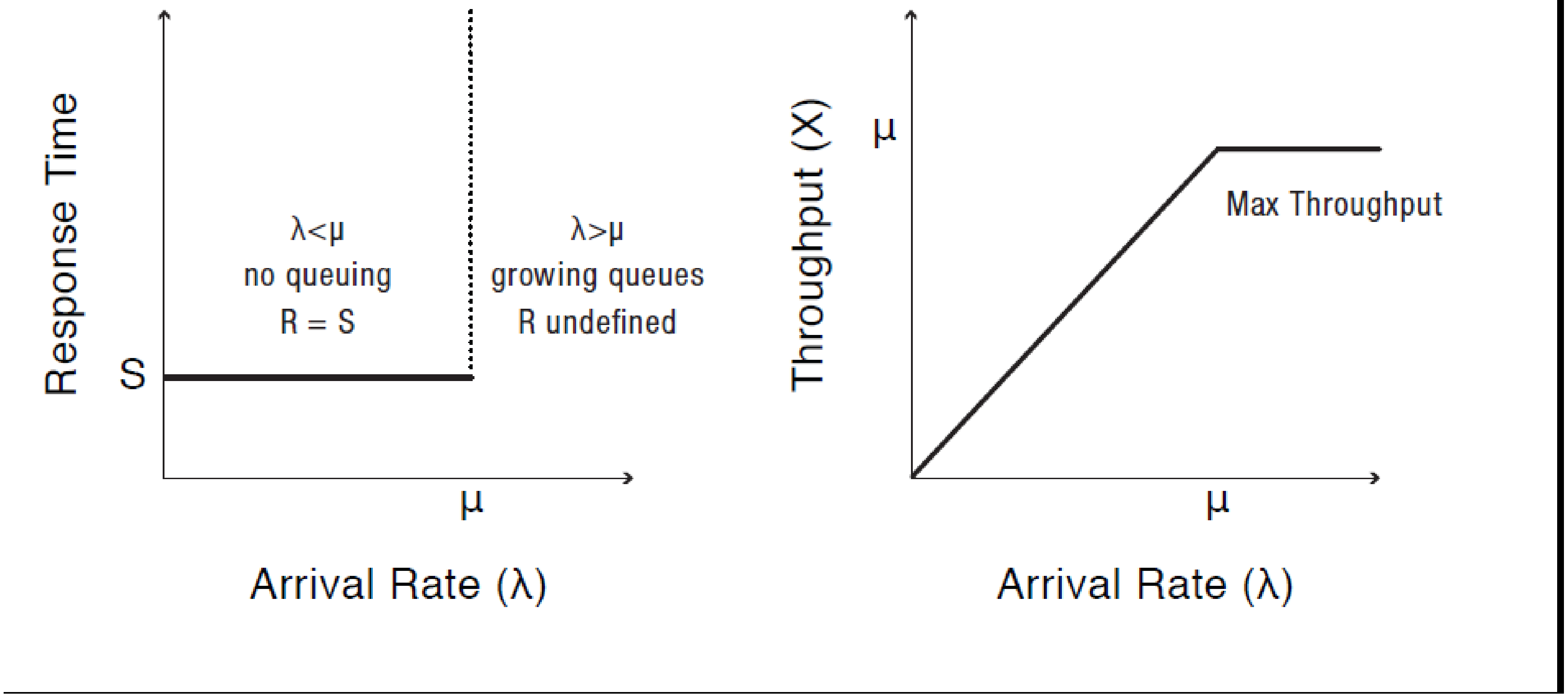
Suppose a system has throughput (X) = 100 tasks/s, average response time (R) = 50 ms/task

- 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?

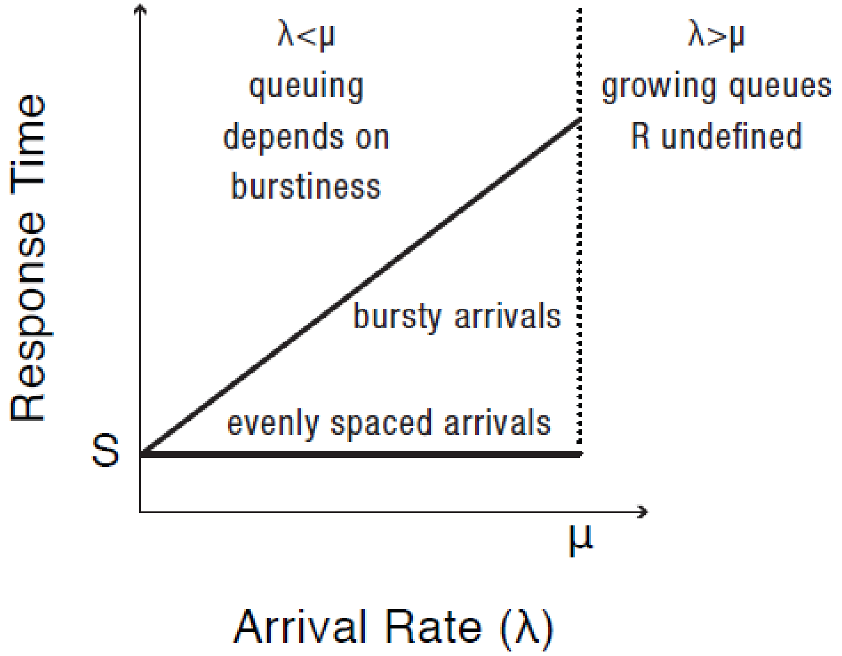
Queueing

- What is the best case scenario for minimizing queueing delay?

Queueing: Best Case



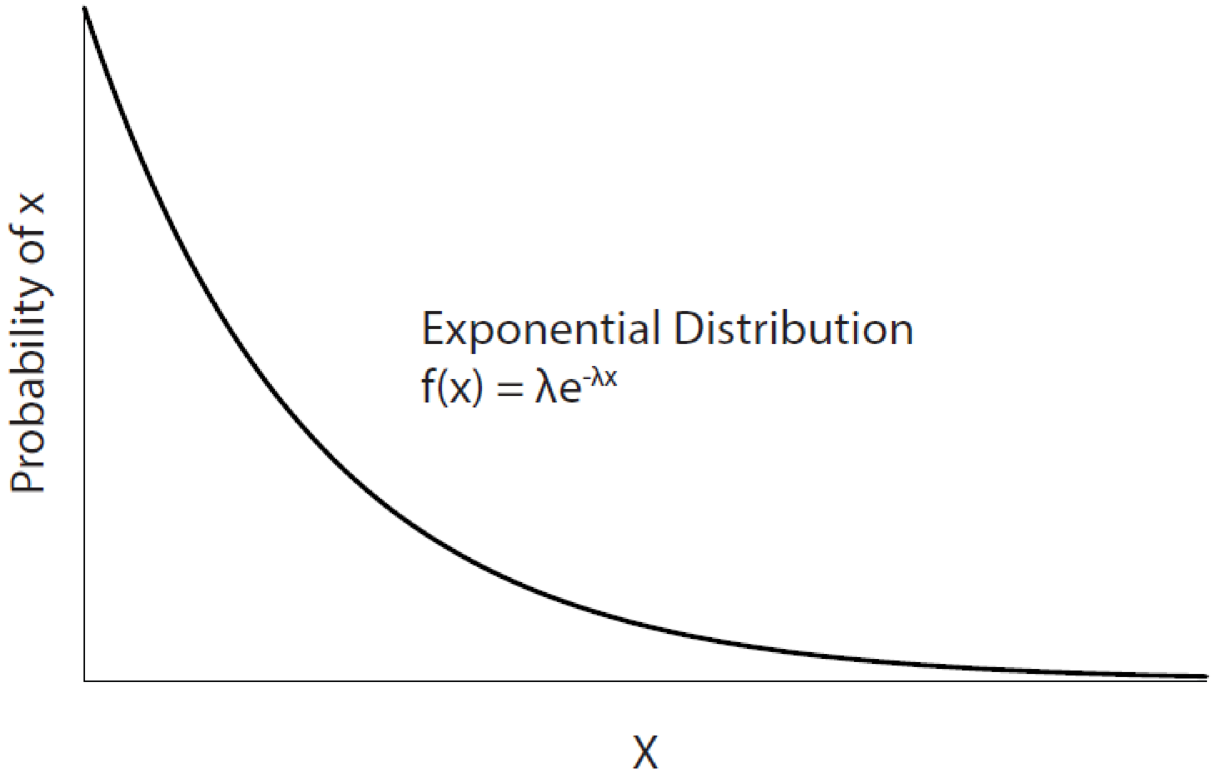
Response Time: Best vs. Worst Case



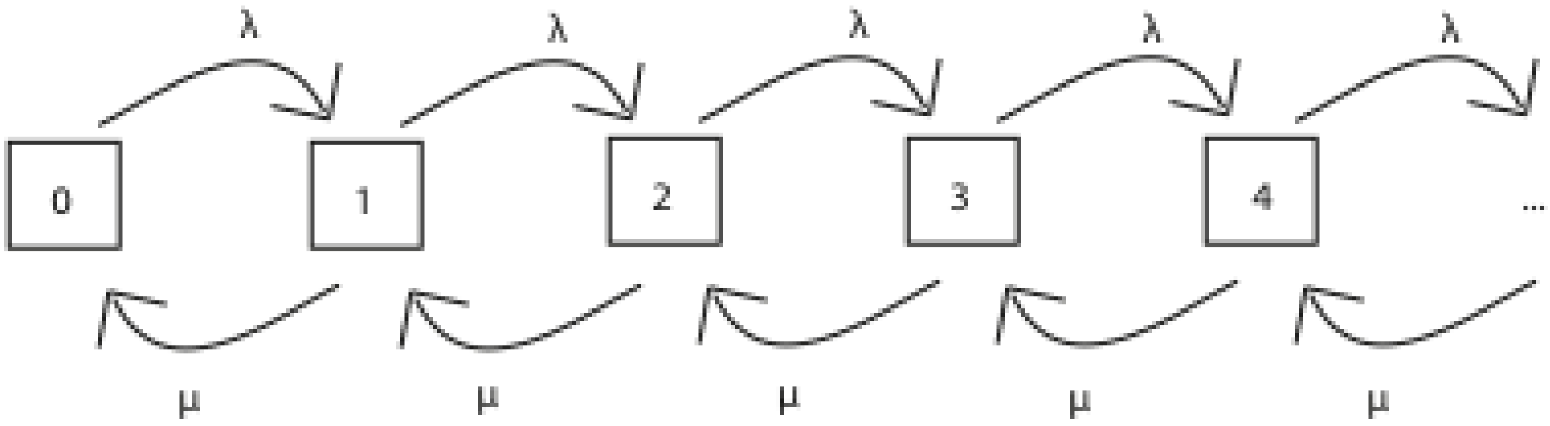
Queueing: Average Case?

- What is average?
  - Gaussian: Arrivals are spread out, around a mean value
  - Exponential: arrivals are memoryless
  - Heavy-tailed: arrivals are bursty
- Can have randomness in both arrivals and service times

Exponential Distribution

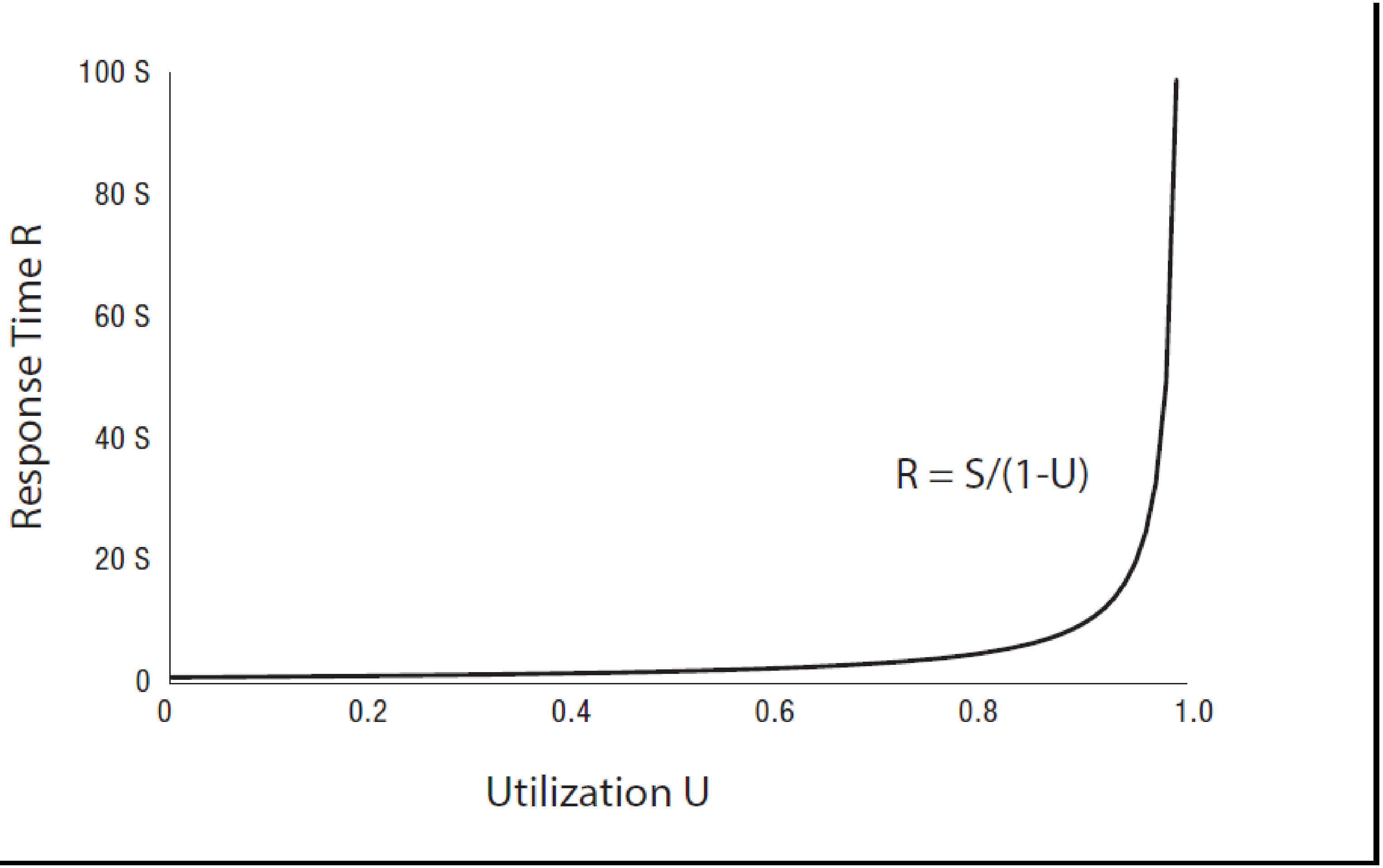


Exponential Distribution



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

Response Time vs. Utilization



Question

- Exponential arrivals: $R = S/(1-U)$
- If system is 20% utilized, and load increases by 5%, how much does response time increase?

- If system is 90% utilized, and load increases by 5%, how much does response time increase?

What if Multiple Resources?

- Response time =

  Sum over all i

    Service time for resource i /

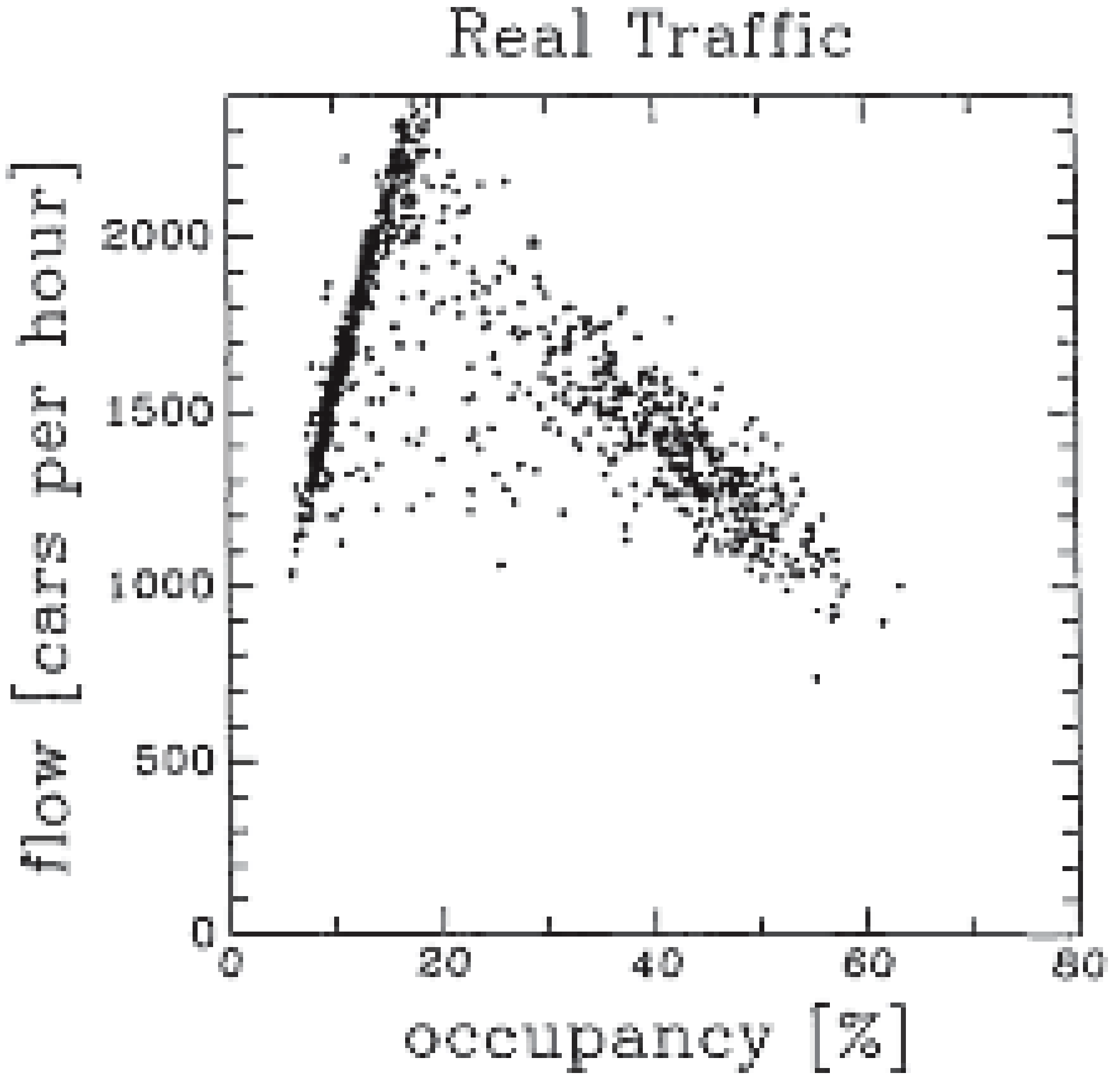
      (1 – Utilization of resource i)

- Implication
  - If you fix one bottleneck, the next highest utilized resource will limit performance

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
  - Example: Highway congestion
- Degrade service?
  - Compute result with fewer resources
  - Example: CNN static front page on 9/11

Highway Congestion (measured)



Data Center Case Study

- P. 361 to be added

Scheduling

Chapter 7 OSPP

Part III: Lottery Scheduling

Overview

- Scheduling Issues
- Lottery Scheduling
- Implementation
- Experiments
- Conclusions

Scheduling Issues

- Context
  - multiple scarce resources: CPU, I/O bw, mem
  - concurrently executing clients
  - service requests of varying importance and characteristics
- Quality of Service
- Modularity

Conventional Scheduling

- Priority Scheduling
  - absolute control (but crude)
  - decay-usage scheduling
    - fair, but hard to analyze, gives avg performance
  - Does p=1 vs. p=2 mean p=1 always gets the CPU or 2/3?
- Problems
  - often ad hoc
  - unable to control service rates
  - no modular abstraction

Solution: Lottery Scheduling

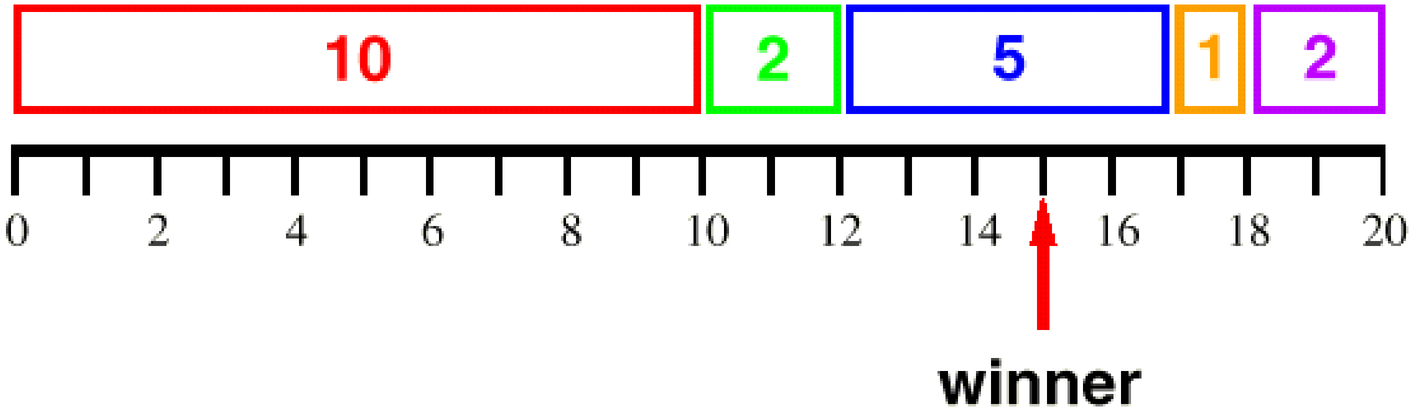
- Easily Understood Behavior
  - proportional share
- Resource Rights Vary Smoothly
  - resource consumption rate proportional to share allocated
- Flexible Control Over Service Rates
  - current schedulers are rigid
- Modular Abstraction
  - multiple resource management policies

Lottery Scheduling Basics

- Randomized Mechanism
- Lottery Tickets
  - encapsulate resource rights
  - issued in different amounts
  - first-class objects
- Lotteries
  - randomly select winning ticket
  - grant resource to client 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 (binomial distribution)
  - throughput proportional to ticket allocation
    - $E[w] = np$
  - response time inversely proportional to ticket allocation
    - $E[n] = 1/p$

Lottery Scheduling Advantages

- Proportional-Share Fairness
  - direct control over service rates
  - easily understood behavior
- Supports Dynamic Environments
  - immediately adapts to changes
  - fair chance to win each allocation
- No starvation
  - hold a non-zero # of tickets

Managing Diverse Resources

- Processor Time
- Lock Access
- I/O Bandwidth
  - disk bandwidth
  - network bandwidth
- Space-Shared Resources
  - memory

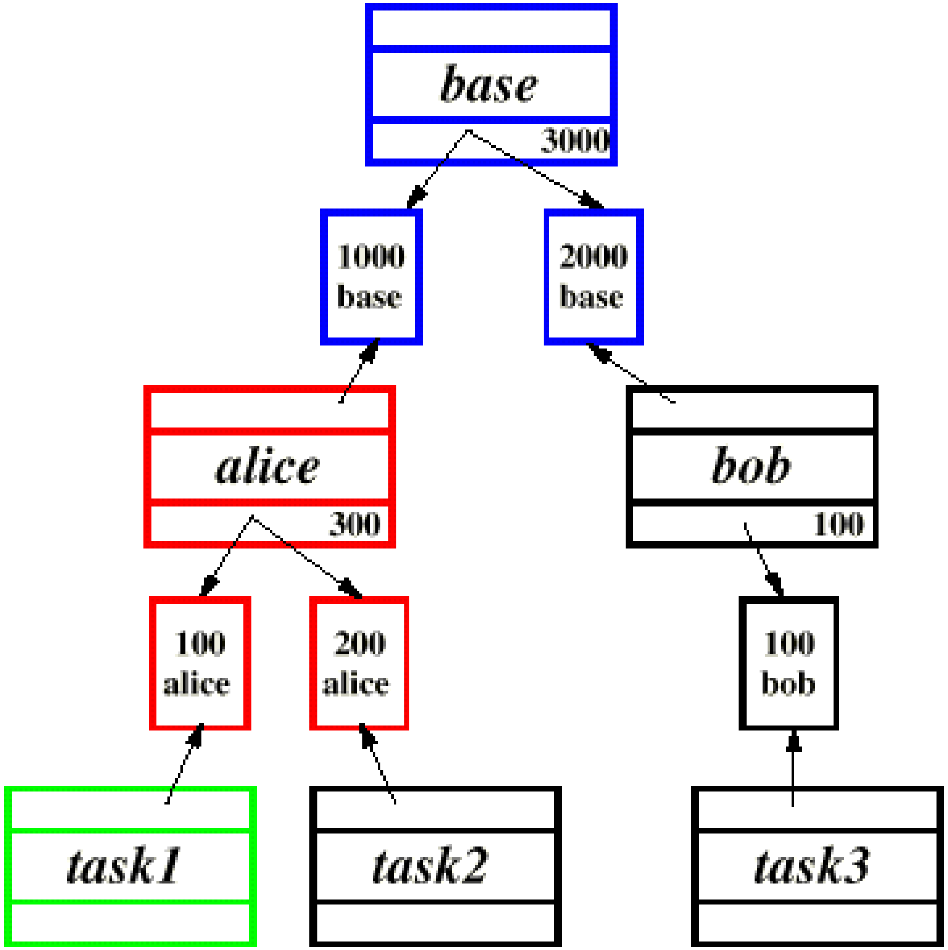
Flexible Resource Management

- Ticket Transfers
  - explicit transfer between clients
  - useful when client blocks while waiting
- Ticket inflation/deflation
  - client creates/removes tickets
  - violates modularity and load insulation
  - convenient among mutually trusting clients: no communication is needed

Ticket Currencies

- Tickets Denominated in Currencies
- Modular Resource Management
  - locally contain effects of inflation
  - isolates loads across logical trust boundaries
- Powerful Abstraction
  - name, share, and protect resource rights
  - flexibly group or isolate users and tasks

Currency Implementation



- **Computing Values**
  - currency: sum value of backing tickets
  - ticket: compute share of currency value
- **Example**
  - task1 funding in base units?
  - $\frac{100}{300} \times 1000$
  - 333 base units

Kernel Implementation

- Objects: Ticket, Currency
- Operations
  - create/destroy ticket, currency
  - fund/unfund currency
  - compute value of ticket, currency
- Algorithms
  - straightforward list-based lottery, O(lg # clients)
  - simple currency conversion scheme

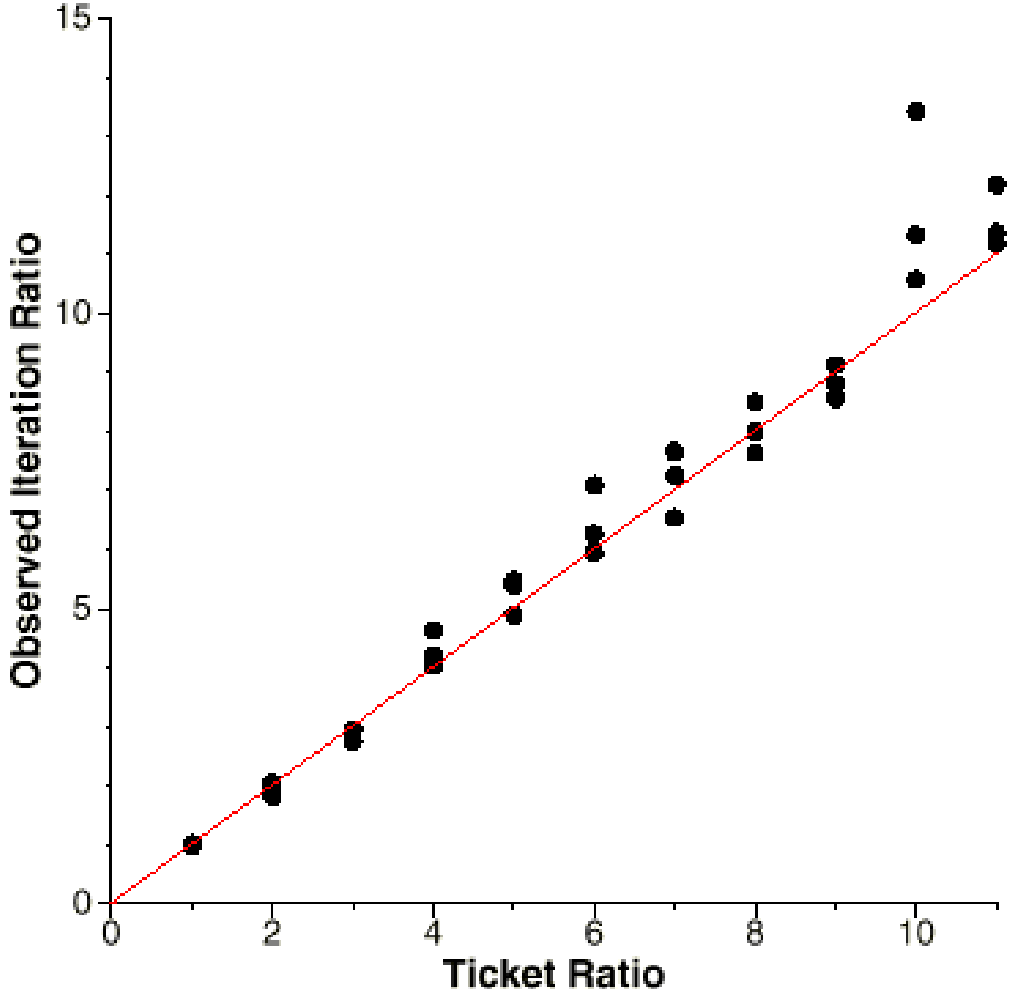
Prototype

- Platform
  - Mach 3.0 microkernel
  - 25 MHz DECStations
  - 100 msec quantum
- System Overhead
  - overhead comparable to standard scheduler
  - unoptimized prototype

Experiments

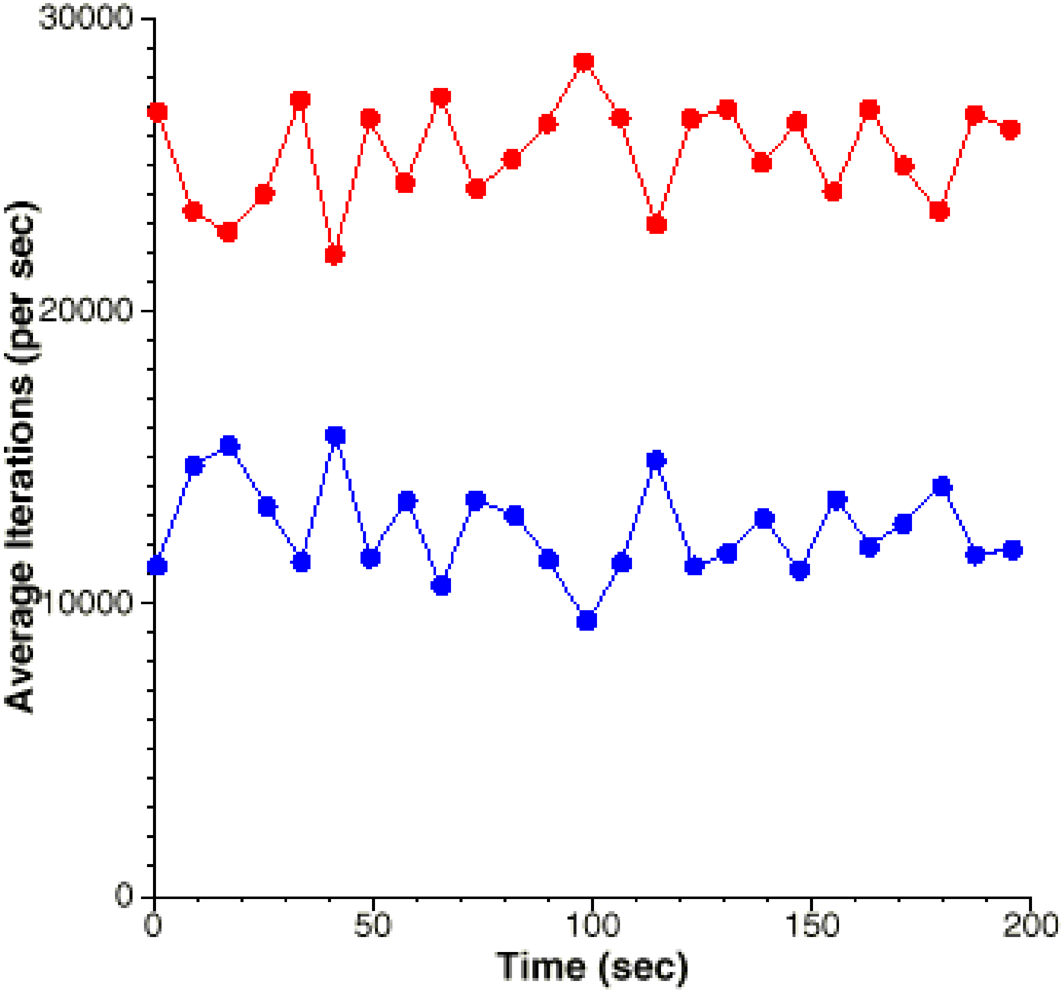
- Proportional-Share Service Rates
- Dynamic Ticket Inflation
- Client-Server Ticket Transfers
- Currency Load Insulation
- Lock Waiting Times

Relative Rates



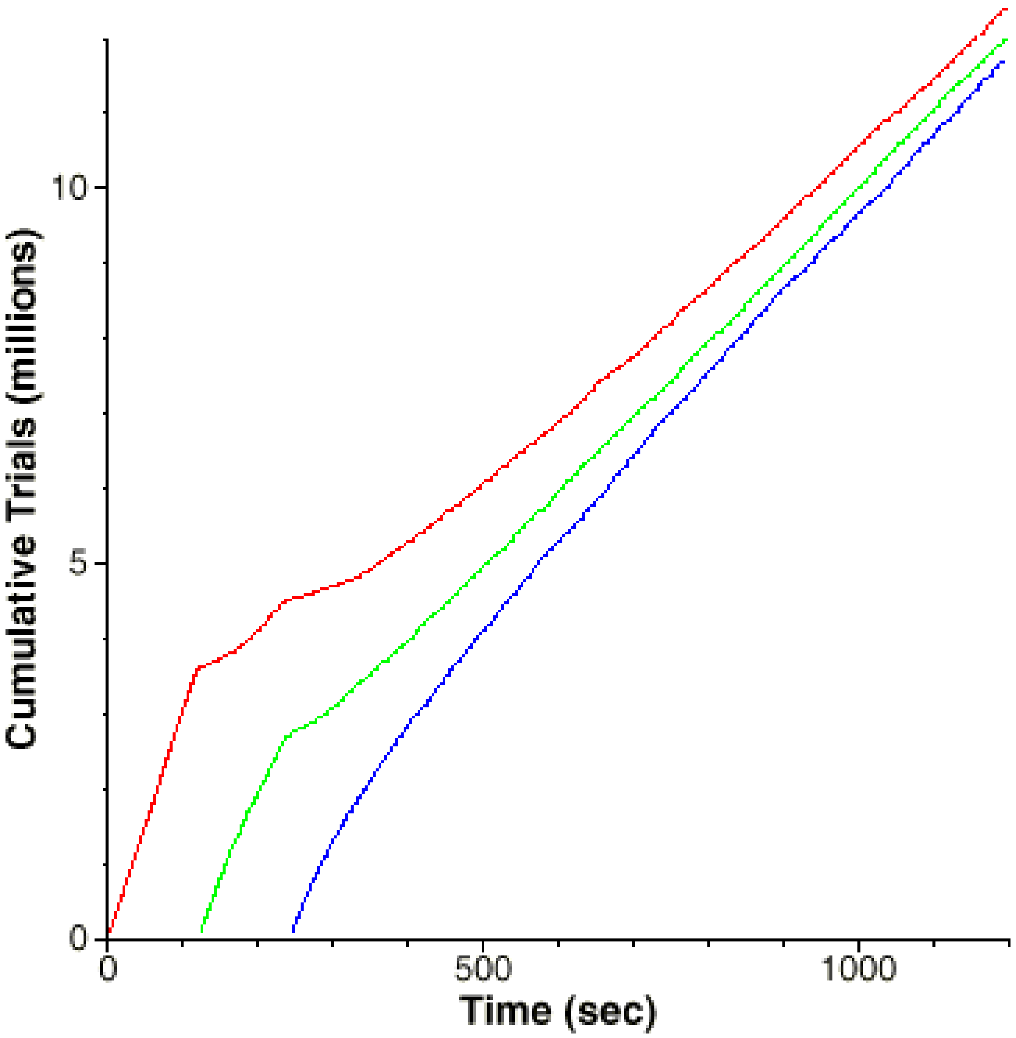
- Dhrystone benchmark
- two tasks
- three 60-second runs for each ratio

Fairness Over Time



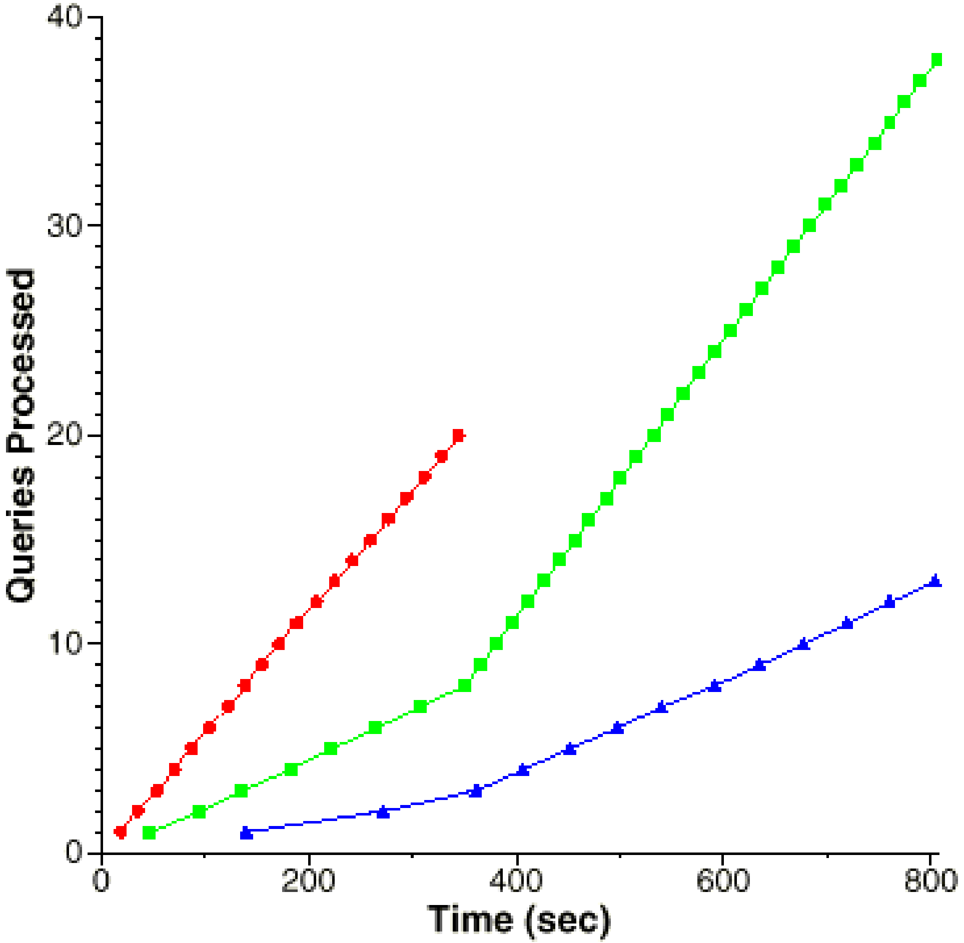
- **Dhrystone benchmark**
- **two tasks**
- **2 : 1 allocation**
- **8-second averages**

Monte-Carlo Rates



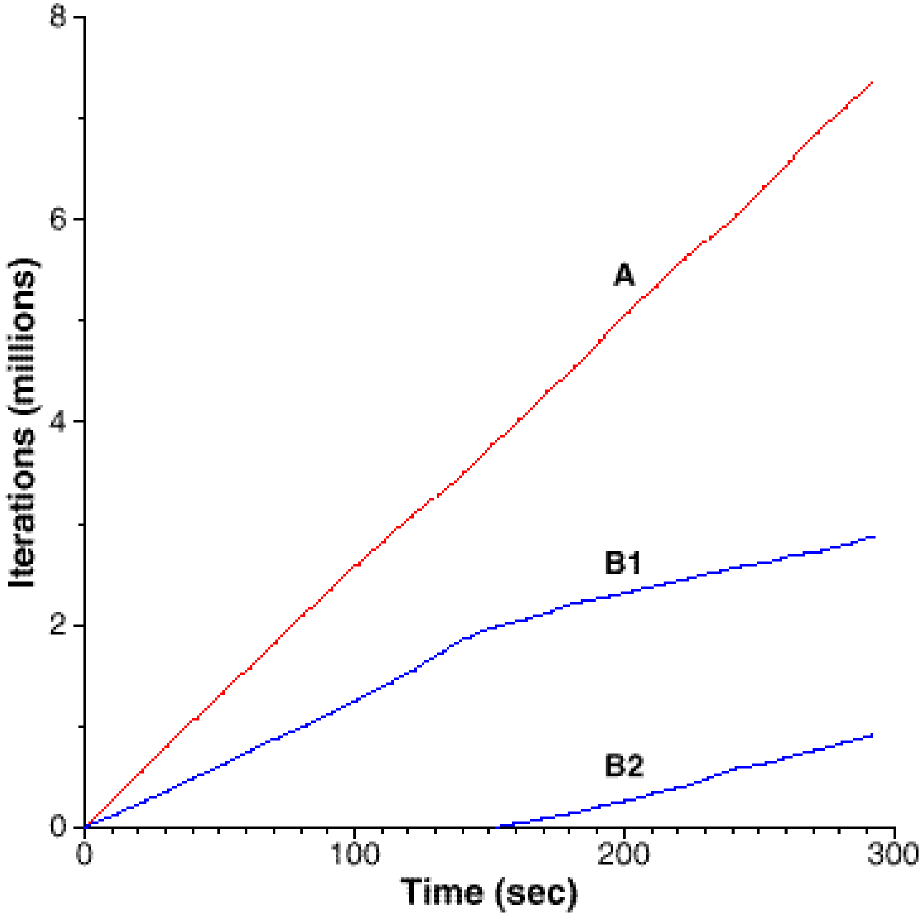
- many trials for accurate results
- three tasks
- ticket inflation
- funding based on relative error

Query Processing Rates



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

Currencies Insulate Loads



- currencies A, B
  2 : 1 funding
- task A
  funding 100.A
- task B1
  funding 100.B
- task B2 joins with
  funding 5 0 .B

Lottery-Scheduled Locks

- Waiting to Acquire
  - waiters transfer funding to lock owner
  - lock owner inhertis aggreagte 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

Conclusions

- Novel Randomized Scheduling Mechanisms
- Easily Understood Behavior
- Precise Control Over Service Rates
- Modular Resource Management
- Generalizes to Diverse Resources

Next

- Address Translation
- OSPP Chapter 8