Scheduling

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. Performance starts to degrade. 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?
  - Tail latency: worst case response time inflation factor?
- Throughput
  - How many tasks can be done per unit of time?
- Overhead
  - How much extra work is done by the scheduler?
- Fairness
  - Do multiple users share resource evenly?
- Strategy-proof
  - Can a user manipulate the system to gain better performance?
- Predictability
  - How consistent is a user's 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)

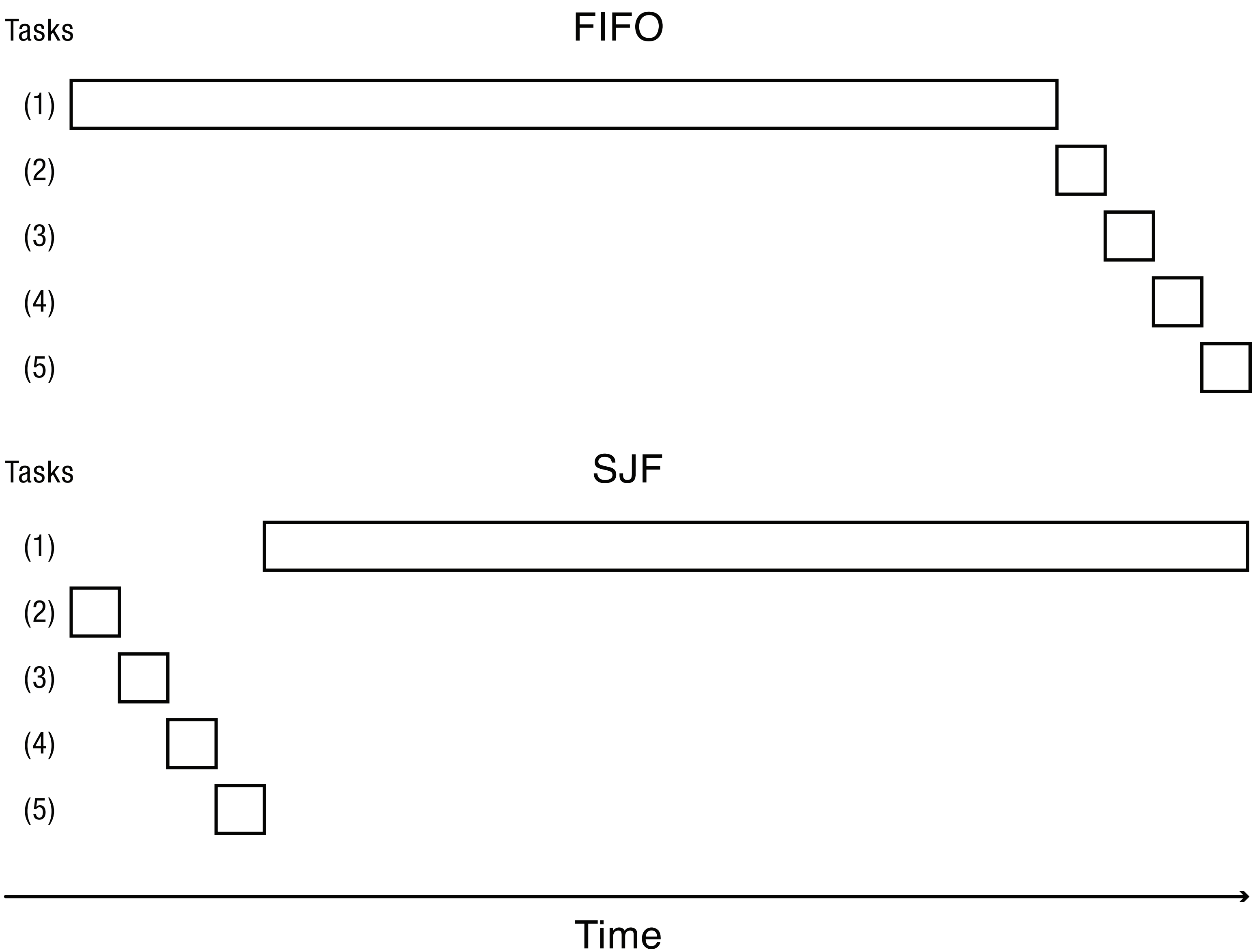
- Schedule tasks in the order they arrive
  - Continue running them until they complete or give up the processor
- Example: memcached
  - Facebook cache of friend lists, …
- On what workloads is FIFO particularly bad?

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?

- Does SJF have any downsides?

Question

- Is FIFO ever optimal (for average response time)?

- Pessimal?

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
- Is this valid or invalid?

Sample Bias Solutions

- Measure for long enough that # of completed tasks >> # of uncompleted tasks
  - For both systems!
- Start and stop system in idle periods
  - Idle period: no work to do
  - If algorithms are work-conserving, both will complete the same tasks

Tail Latency

- What if we are optimizing for tail latency and not average responsiveness?
  - Ex: mapreduce needs to wait for the slowest task
  - Starvation of some jobs not an option
- Many cloud systems provide service level agreements (SLA) to applications
  - Average response time, throughput, …
  - Tail behavior: 99% (or 99.9%) latency, downtime, …

Question

- What does a cache do to tail latency?

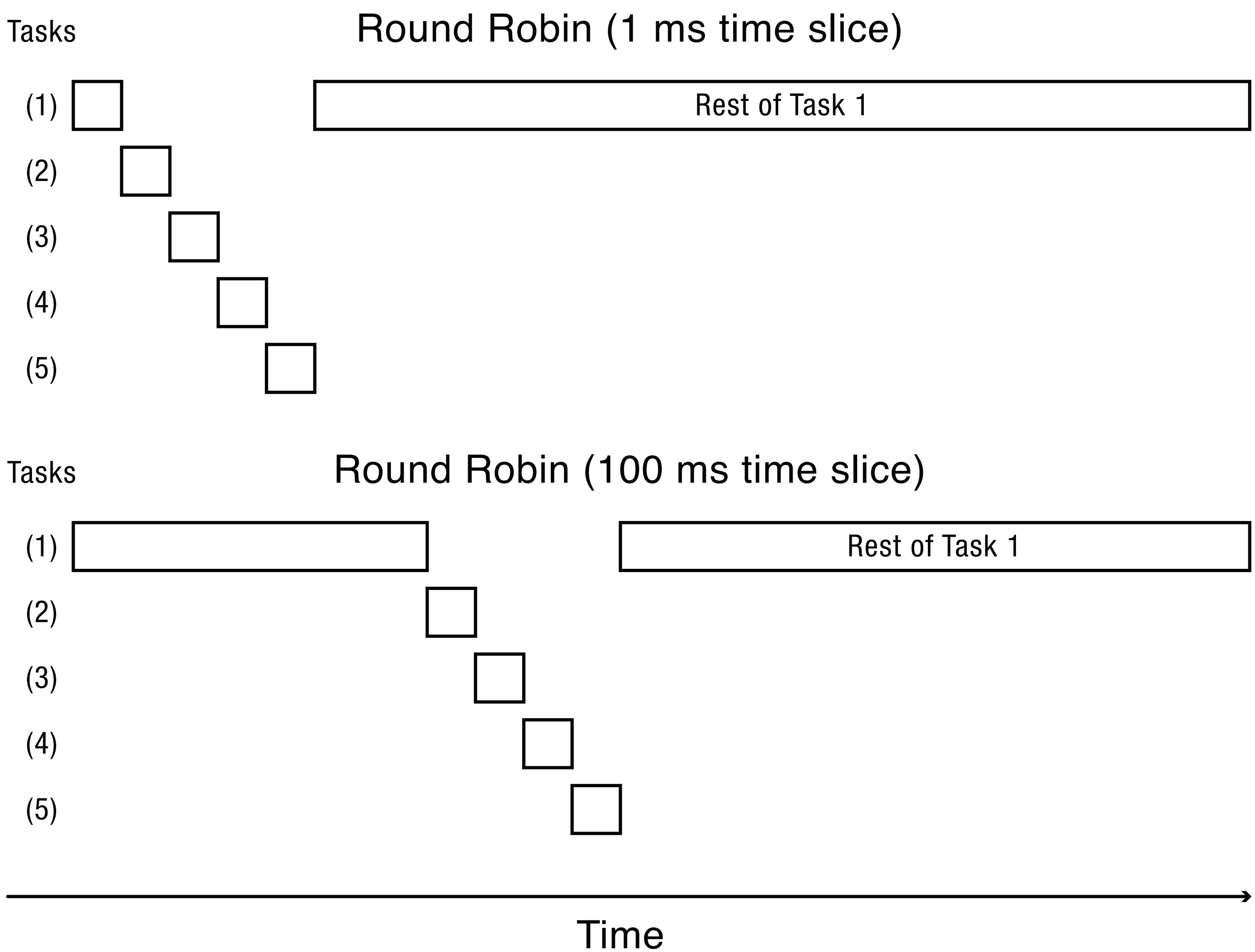
Earliest Deadline First (EDF)

- EDF: run task with the earliest deadline first
  - If it is possible to meet deadlines, EDF will meet them
- SLA + EDF
  - Deadline is arrival time + tail latency goal
- What is optimal for tail latency if all tasks are the same size?
- What if tasks have a mixture of sizes?
  - If it is not possible to meet deadlines, discard longest remaining (or lowest priority) task first
  - Requires predicting the future

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

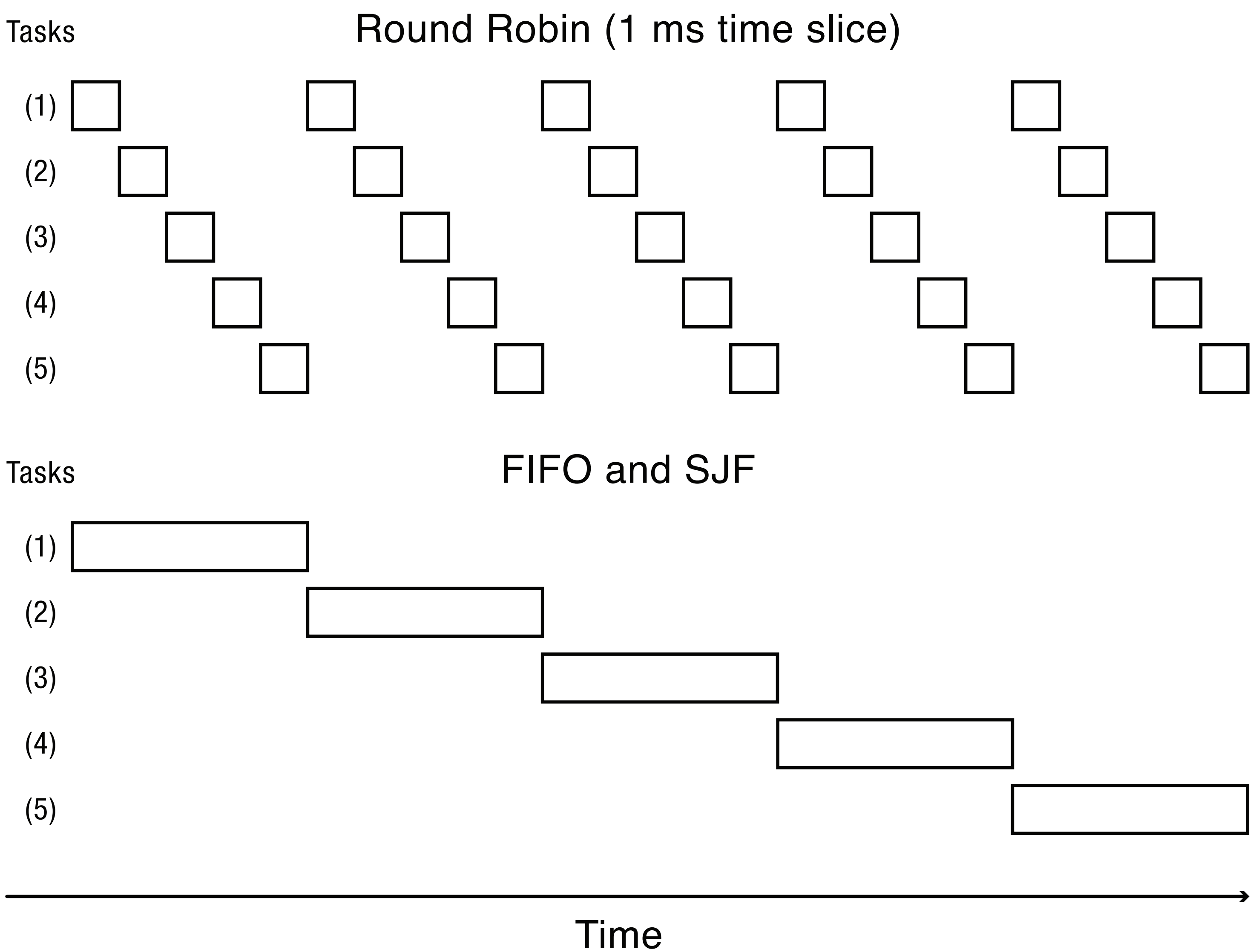


Tasks
Round Robin (1 ms time slice)
(1)
Rest of Task 1
(2)
(3)
(4)
(5)
Tasks
Round Robin (100 ms time slice)
(1)
Rest of Task 1
(2)
(3)
(4)
(5)
Time

Round Robin vs. FIFO

- Assuming zero-cost time slice, is Round Robin always better than FIFO?
  - Average response time?
  - Tail latency?

Round Robin vs. FIFO



Round Robin = Fairness?

- Is Round Robin fair?
- What is fair?
  - Equal share of the CPU?
  - What if some tasks don’t need their full share? How do we allocate the remainder?

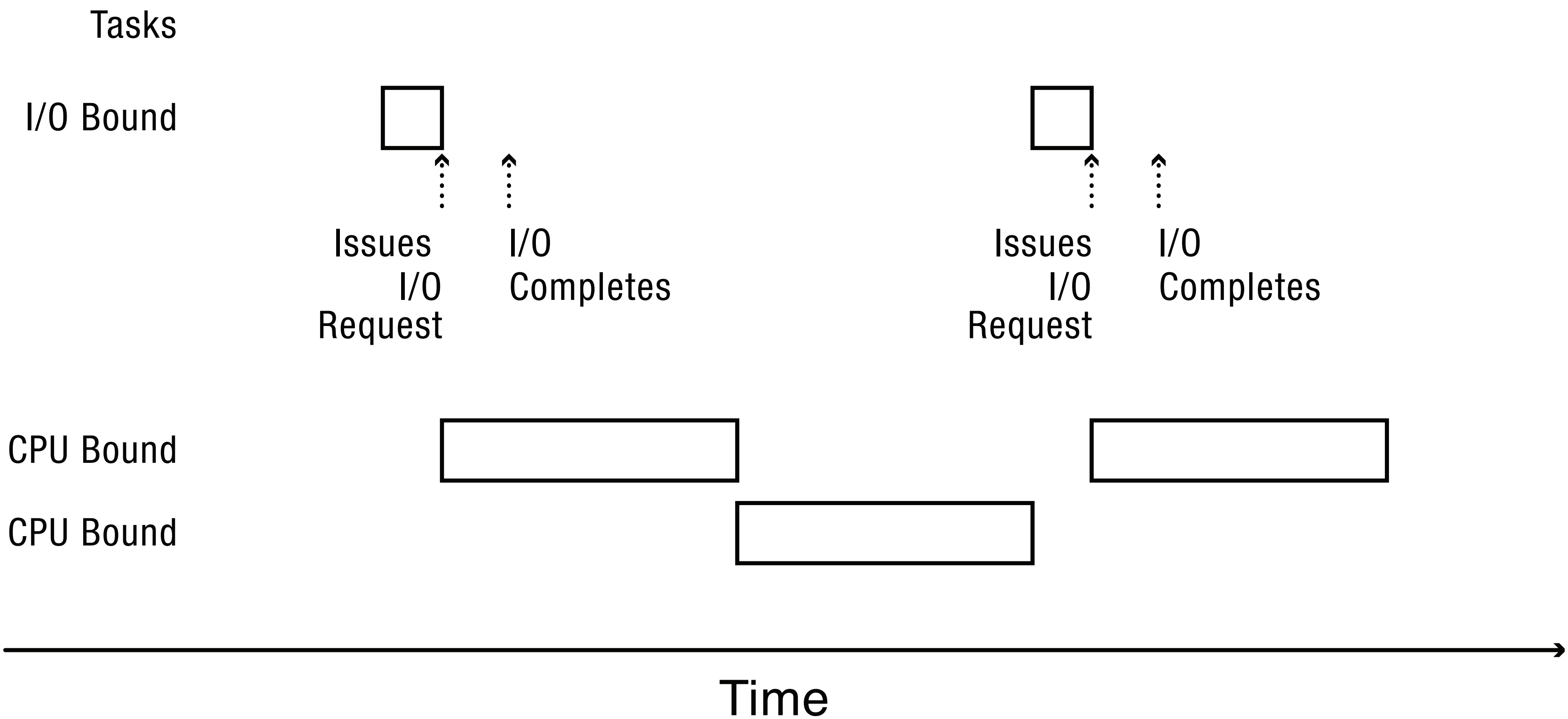
Max-Min Fairness

- Applies to repeating tasks
  - Ex: network bandwidth allocation
- Maximize the min 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

Leaky Bucket (Max-Min)

- Every task gets a leaky bucket
  - Add credits to each task at same rate
  - Debit as task uses resource
  - Cap accumulated credits at some maximum
- Simple scheduling policy
  - Choose task with largest # of credits
  - Or randomly choose proportional to # of credits

Mixed Workload



Scheduling Multiple Resources

- How do we balance a tasks that need a mixture of resources:
  - Some I/O bound, need only a little CPU
  - Some compute bound, can use as much CPU as they are assigned
  - Queue for CPU reduces I/O throughput
- Max-min over each resource separately?
- Min-max inflation relative to system with no competing tasks?

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 (and probably Windows, MacOS)

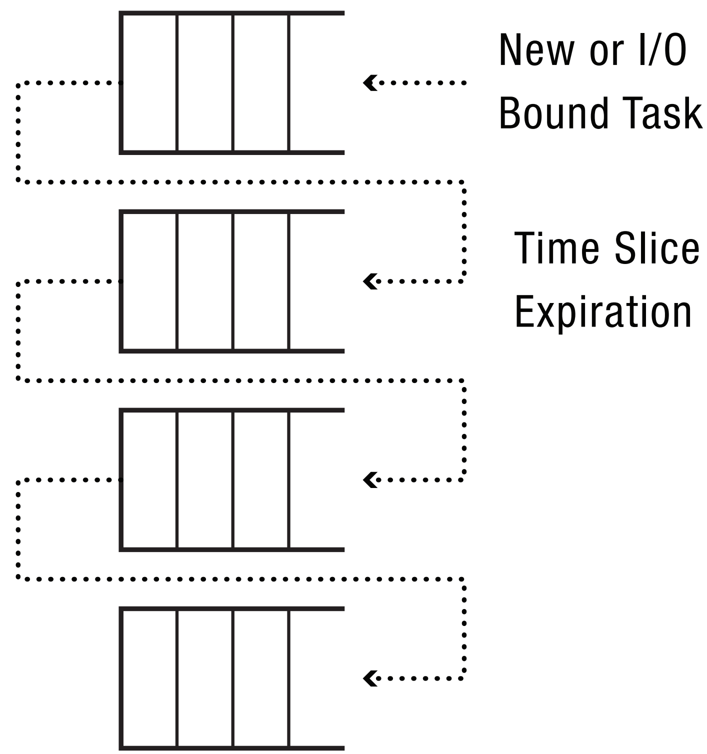
MFQ

- 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

Round Robin Queues



MFQ and Tail Latency

- How predictable is a task's performance?
  - Can it be affected by other users?

- Linux boosts priority to tasks being starved

MFQ and Strategy

- Can a user get better performance (response time, throughput) by doing useless work?

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 pessimal 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 both avoid starvation.
- MFQ can adjust priorities to balance responsiveness, overhead, and fairness.
- MFQ approximates SJF
  - High variance for long jobs; vulnerable to strategy

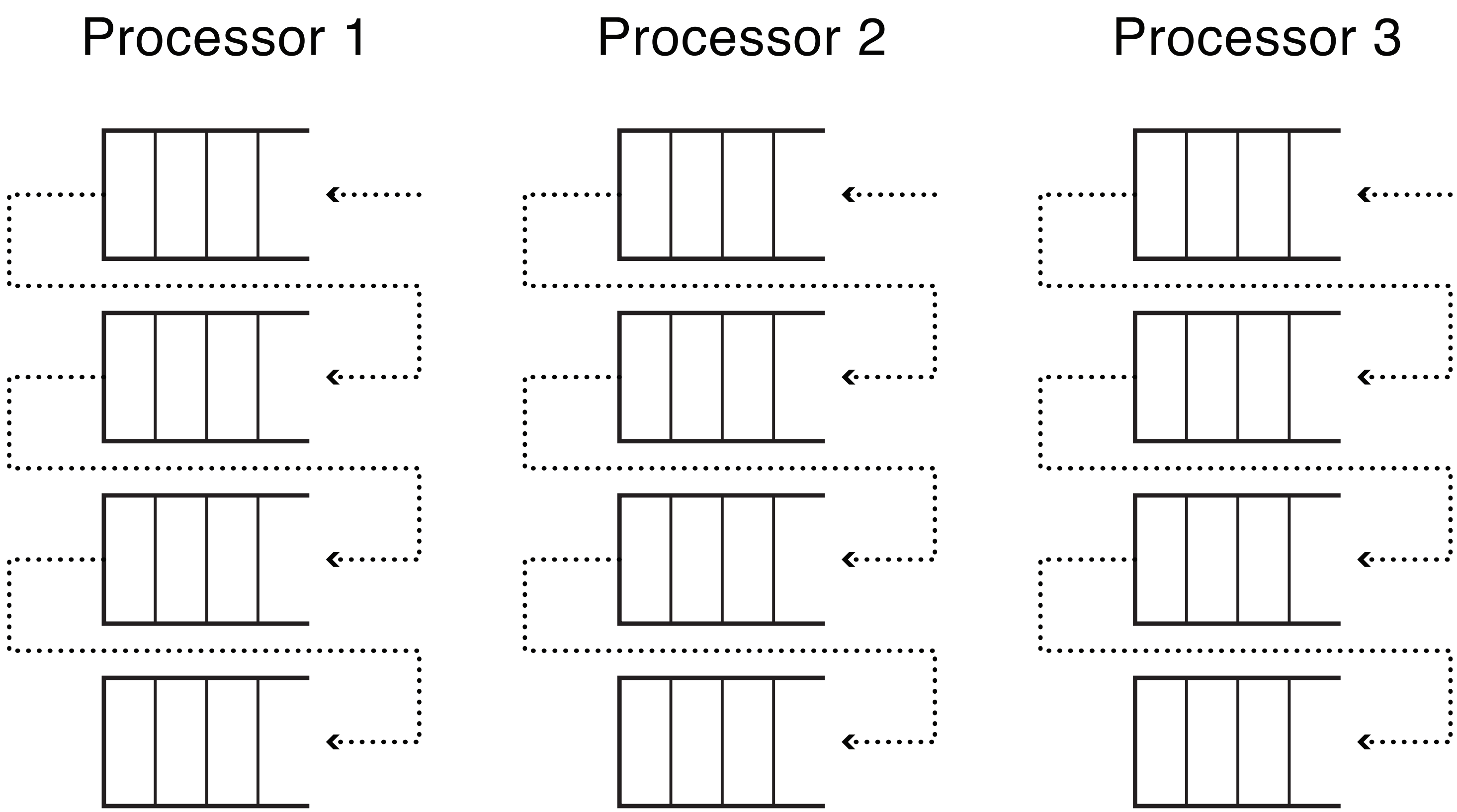
Multiprocessor Scheduling

- 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



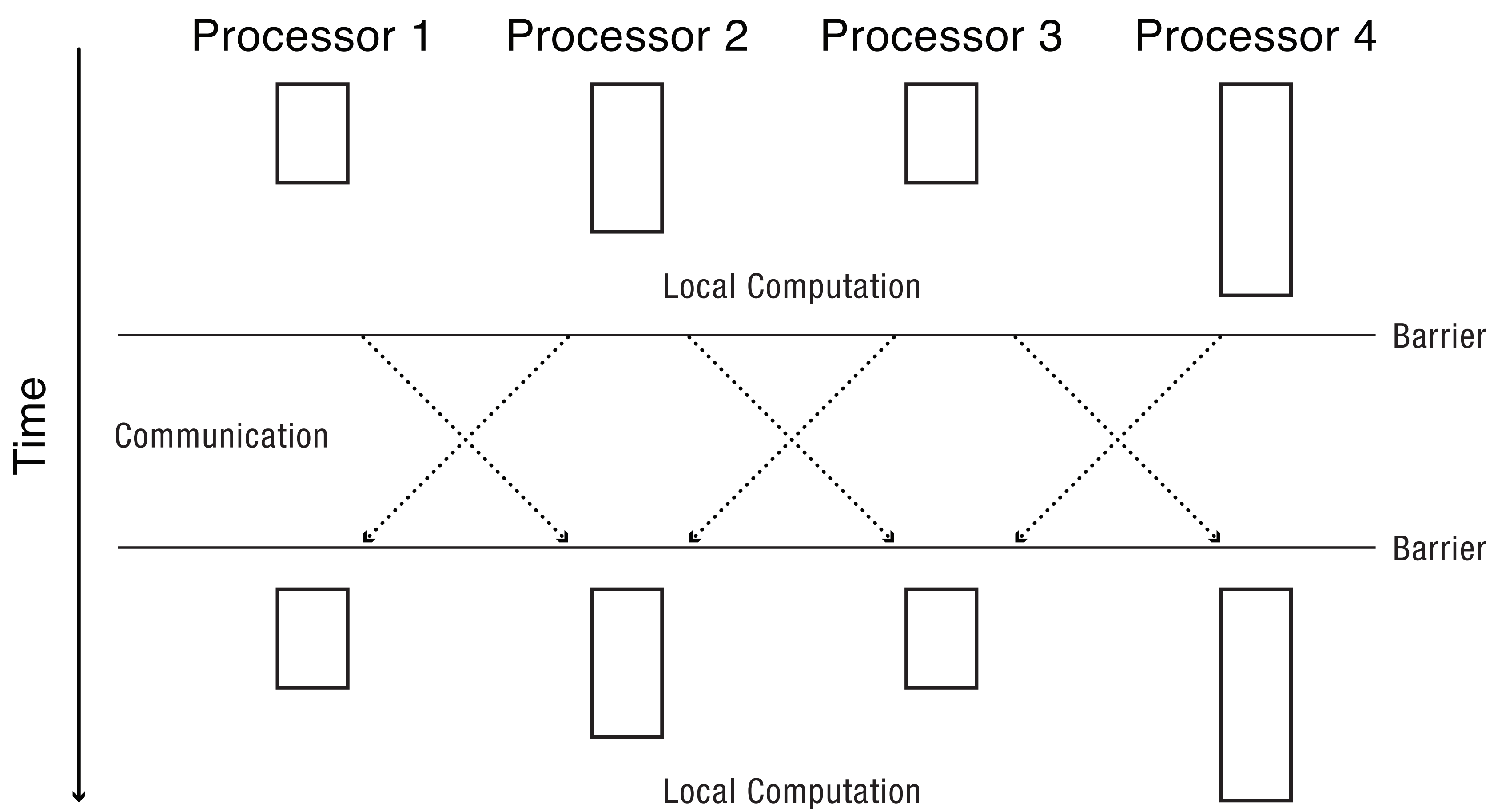
Scheduling Parallel Programs

- 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

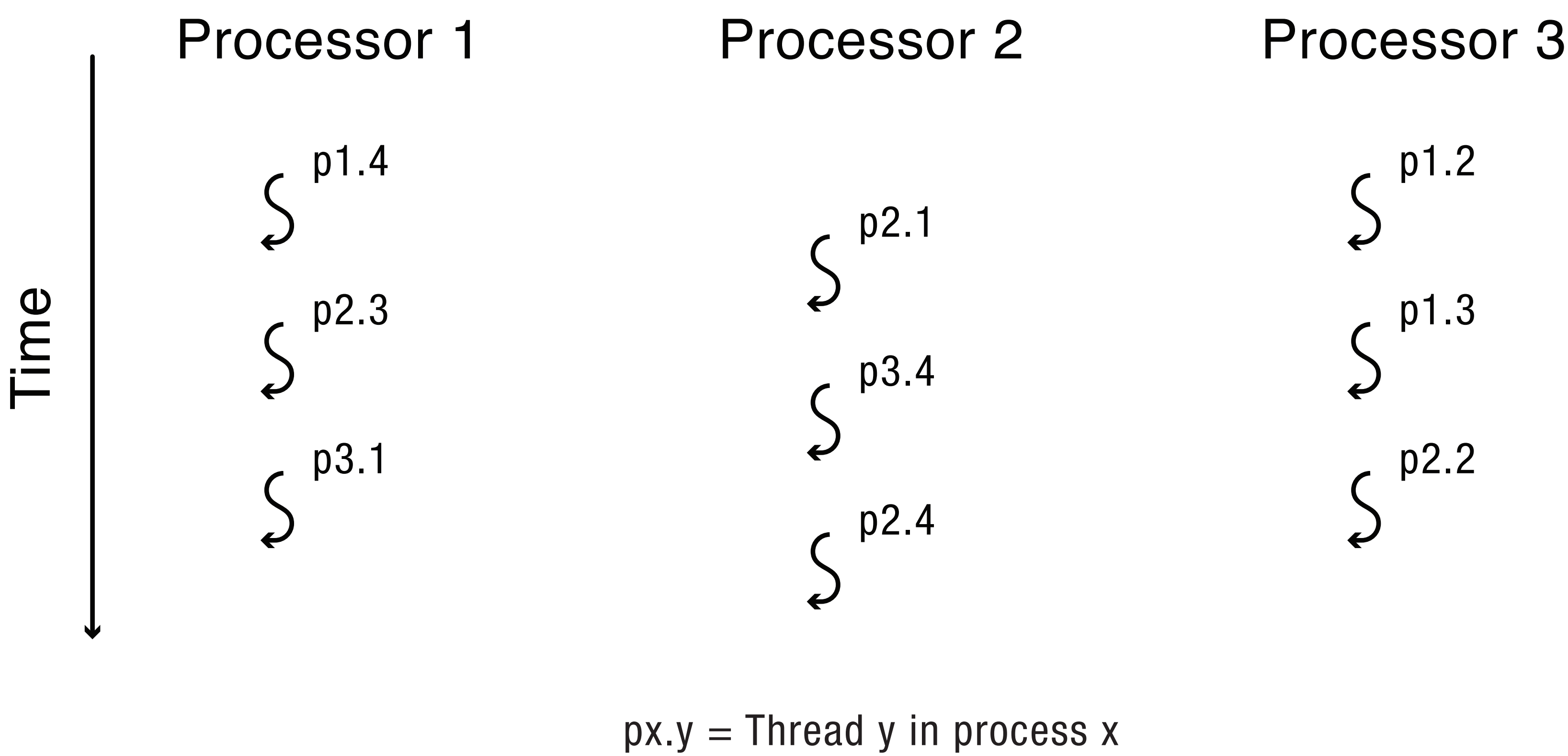
- Loop at each processor:
  - Compute on local data (in parallel)
  - Barrier
  - Send (selected) data to other processors (in parallel)
  - Barrier
- Examples:
  - MapReduce
  - Fluid flow over a wing
  - Most parallel algorithms can be recast in BSP, sacrificing at most a small constant factor in performance

Tail Latency

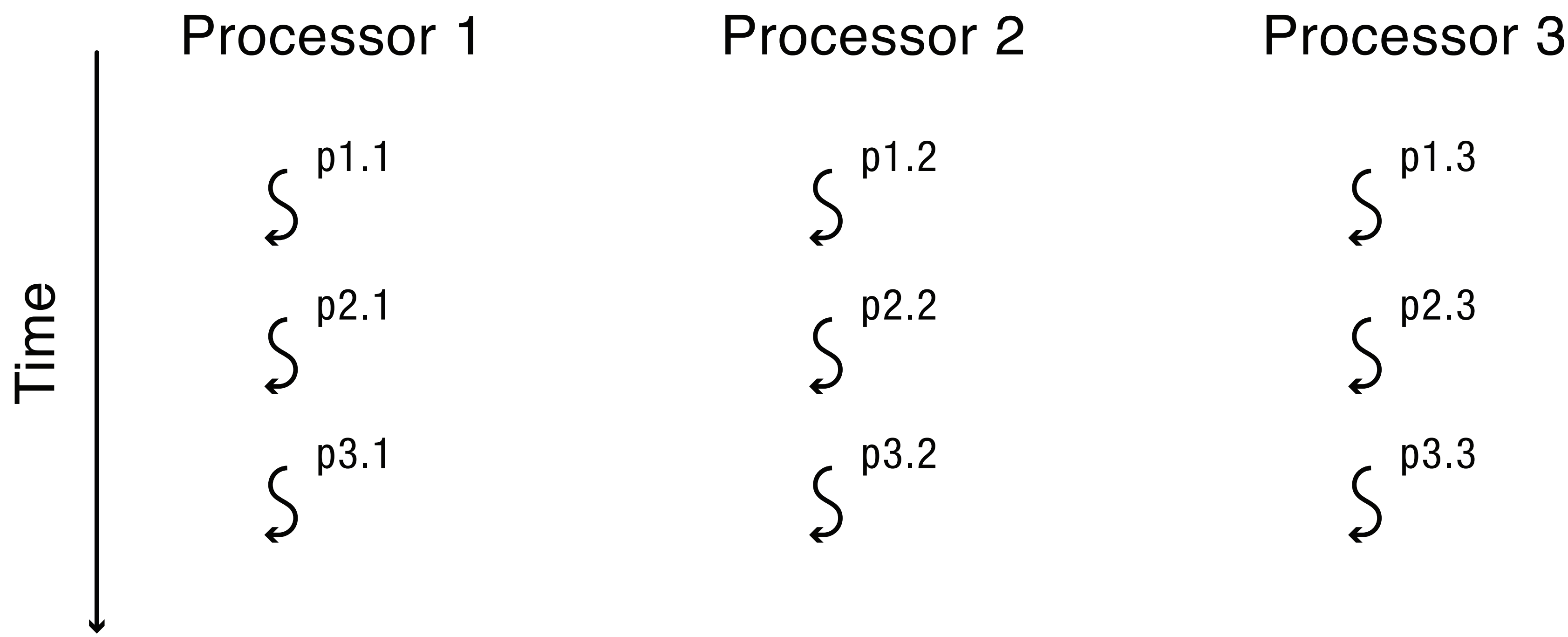


Scheduling Parallel Programs

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

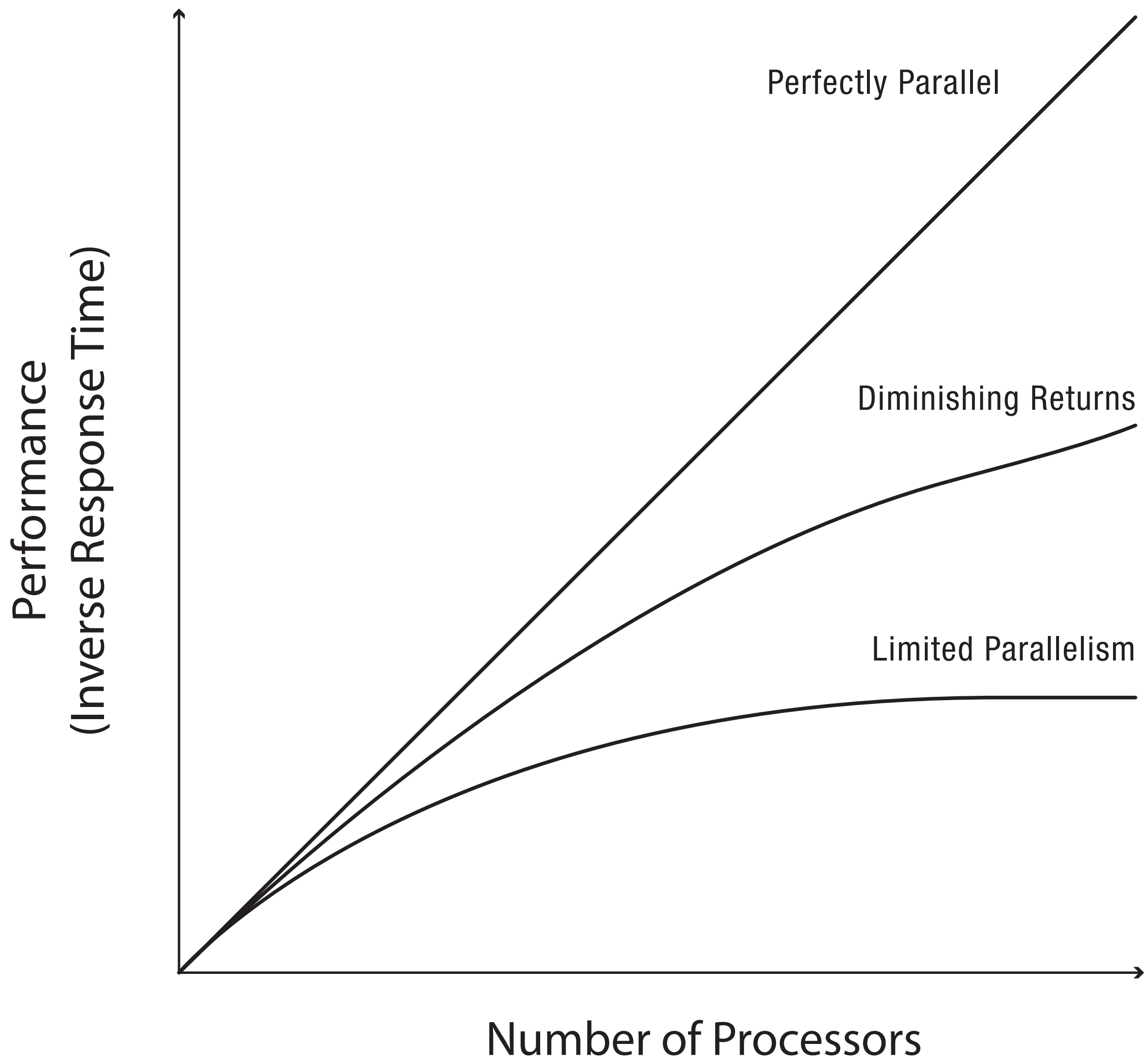


Gang Scheduling

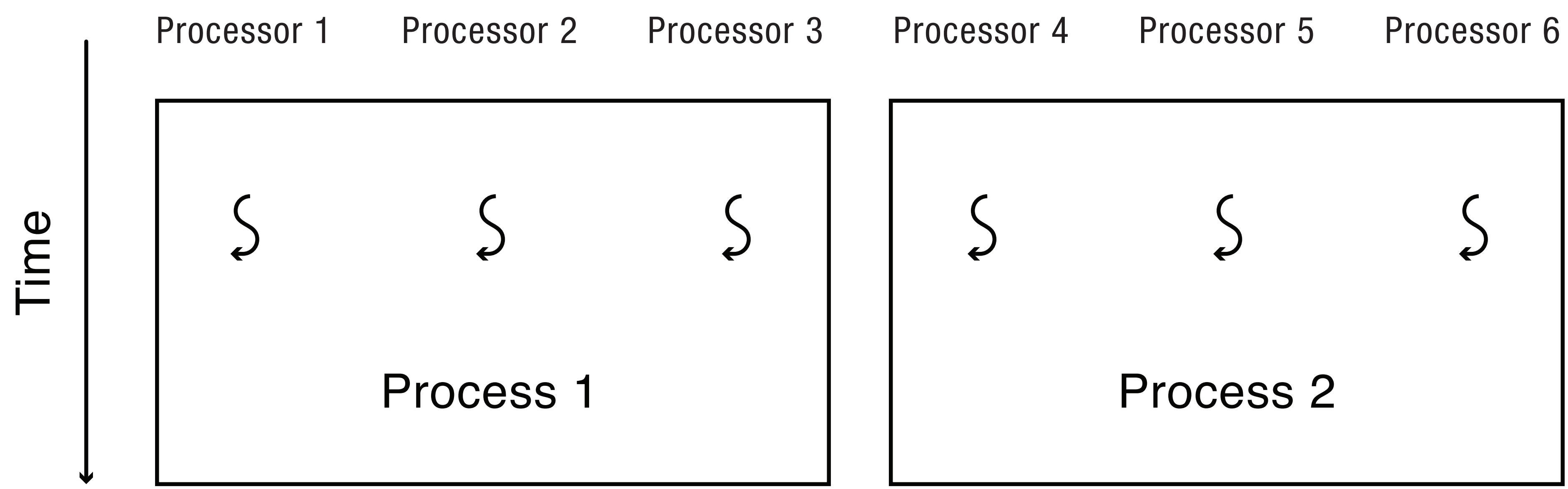


px.y = Thread y in process x

Parallel Program Speedup



Space Sharing

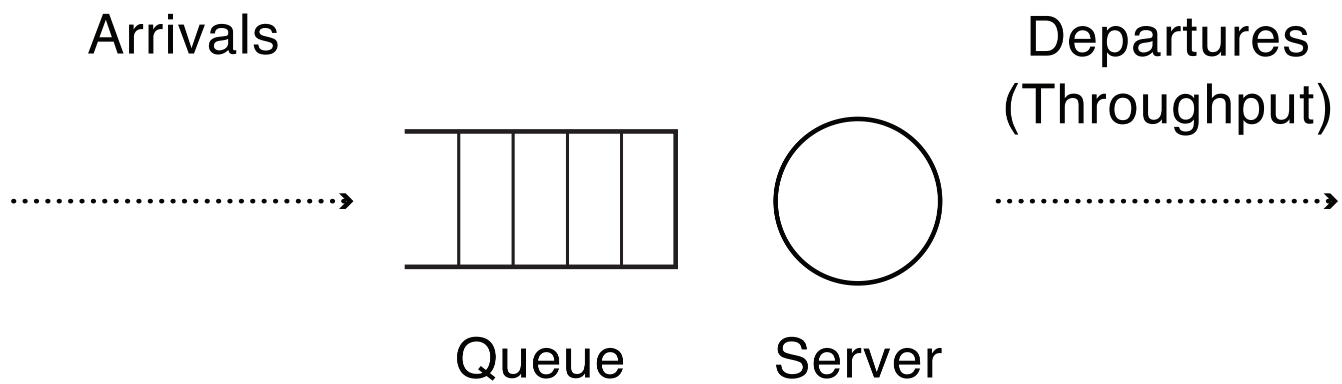


Scheduler activations: kernel tells each application its # of processors with upcalls every time the assignment changes

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

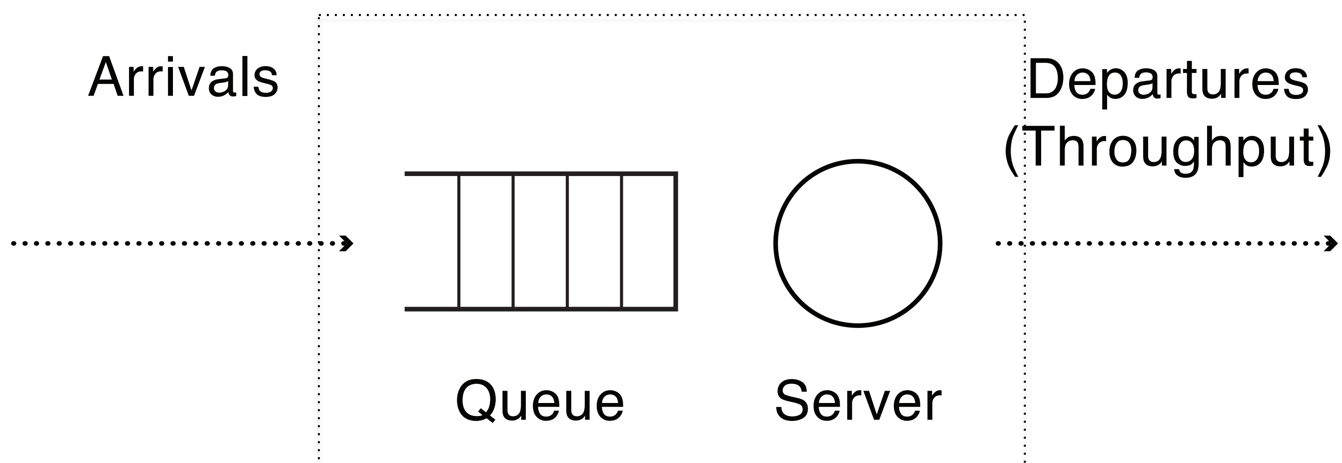
Applies to *any* stable system – where arrivals match departures.

- Independent of scheduling discipline and burstiness

Question

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

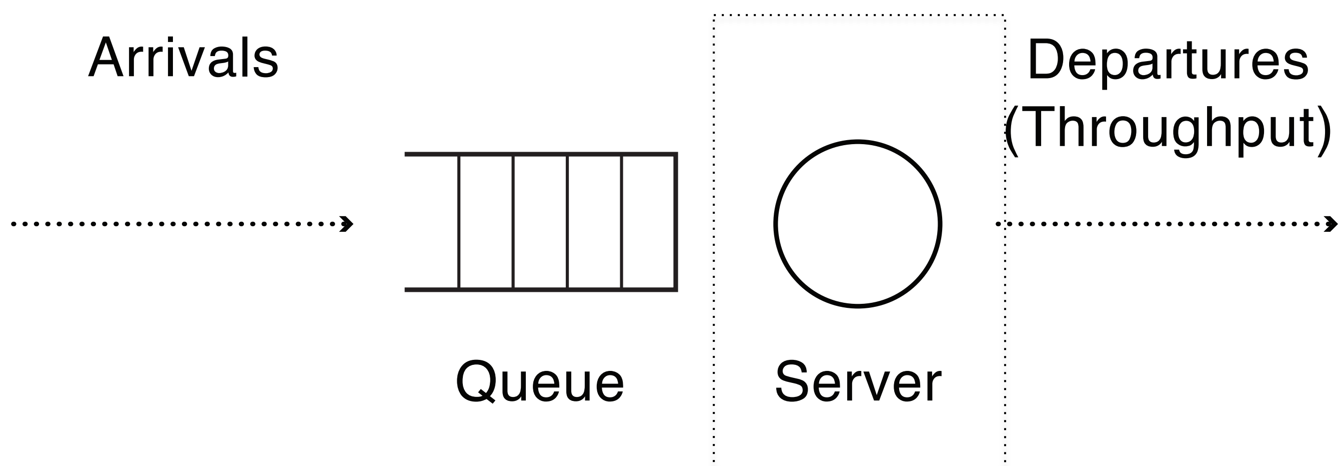
- How many tasks are in the system on average?
  - Hint: Little's Law N = X * R



Question

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

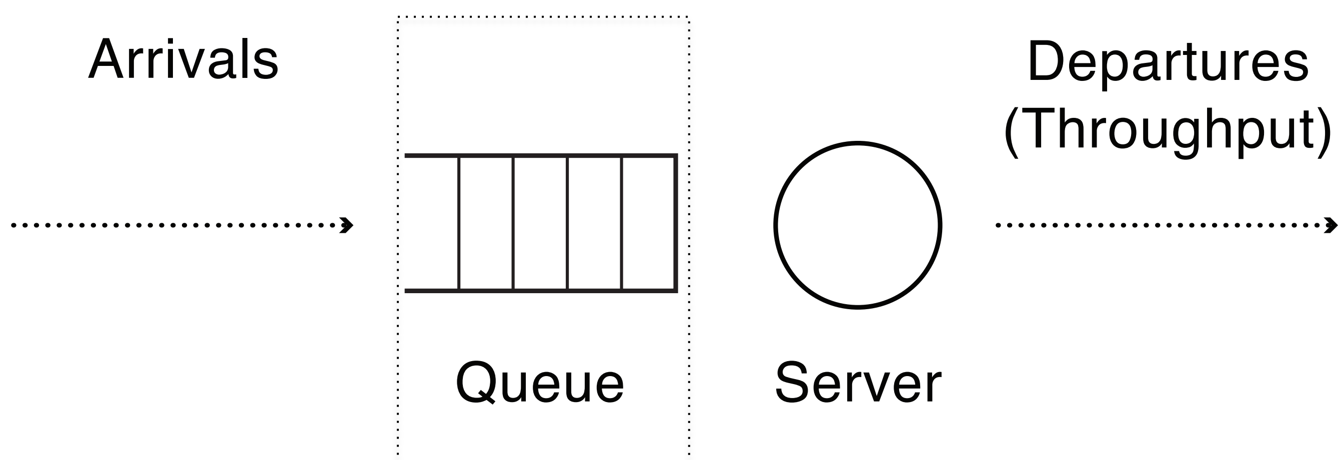
- If the server takes 5 ms/task, what is its utilization? (N = X * R)



Question

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

- What is the average wait time?
- What is the average number of queued tasks?



Question

- From example:

  X = 100 task/sec

  R = 50 ms/task

  S = 5 ms/task

  W = 45 ms/task

  Q = 4.5 tasks

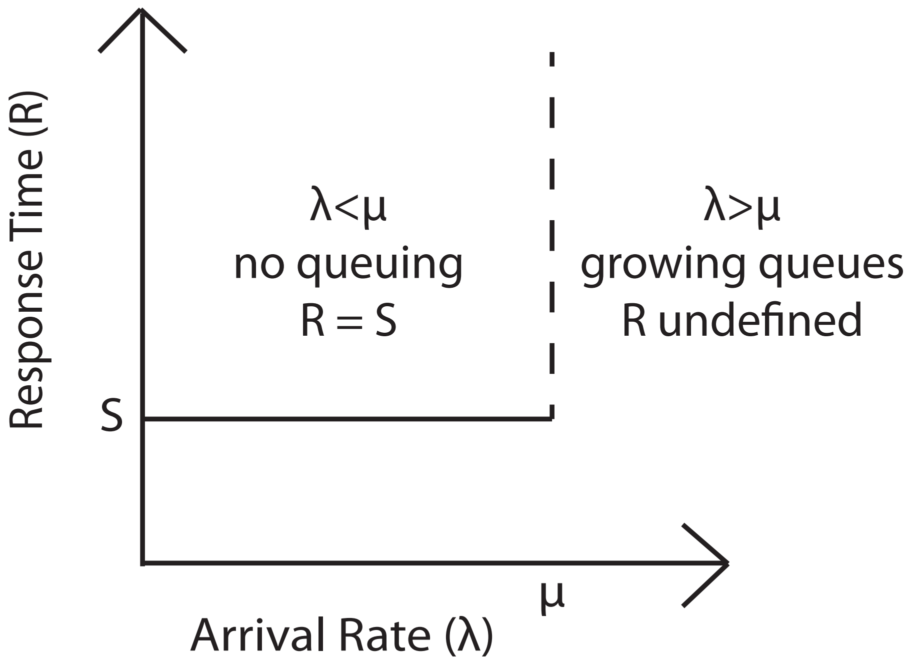
- What gives? W = 45 ms while S * Q = 22.5 ms
  - Hint: what if S = 10ms? S = 1ms?

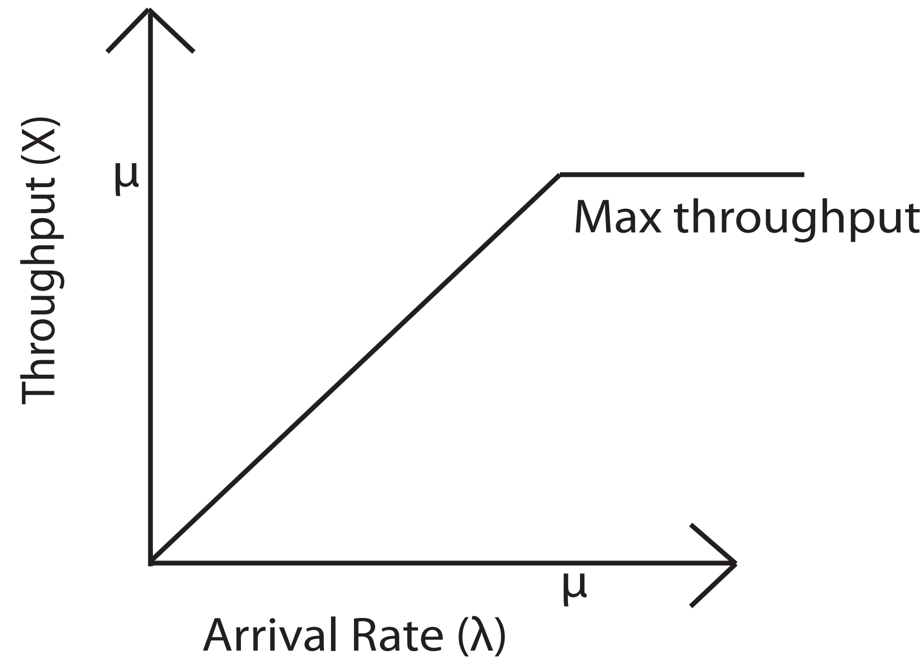
Queueing

- What is the best case scenario for minimizing queueing delay?
  – Keeping arrival rate, service time constant

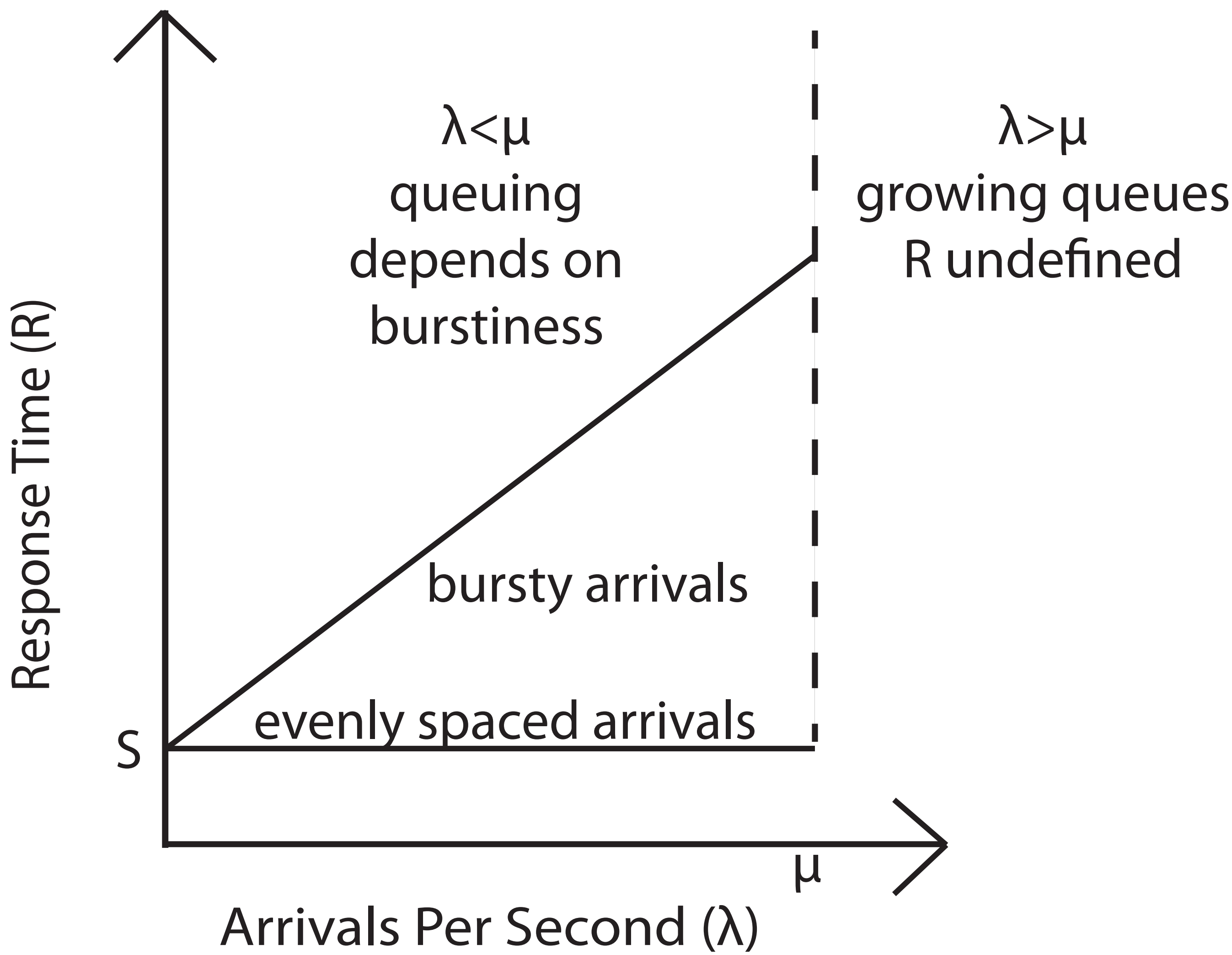
- What is the worst case scenario?

Queueing: Best Case





Response Time: Best vs. Worst Case

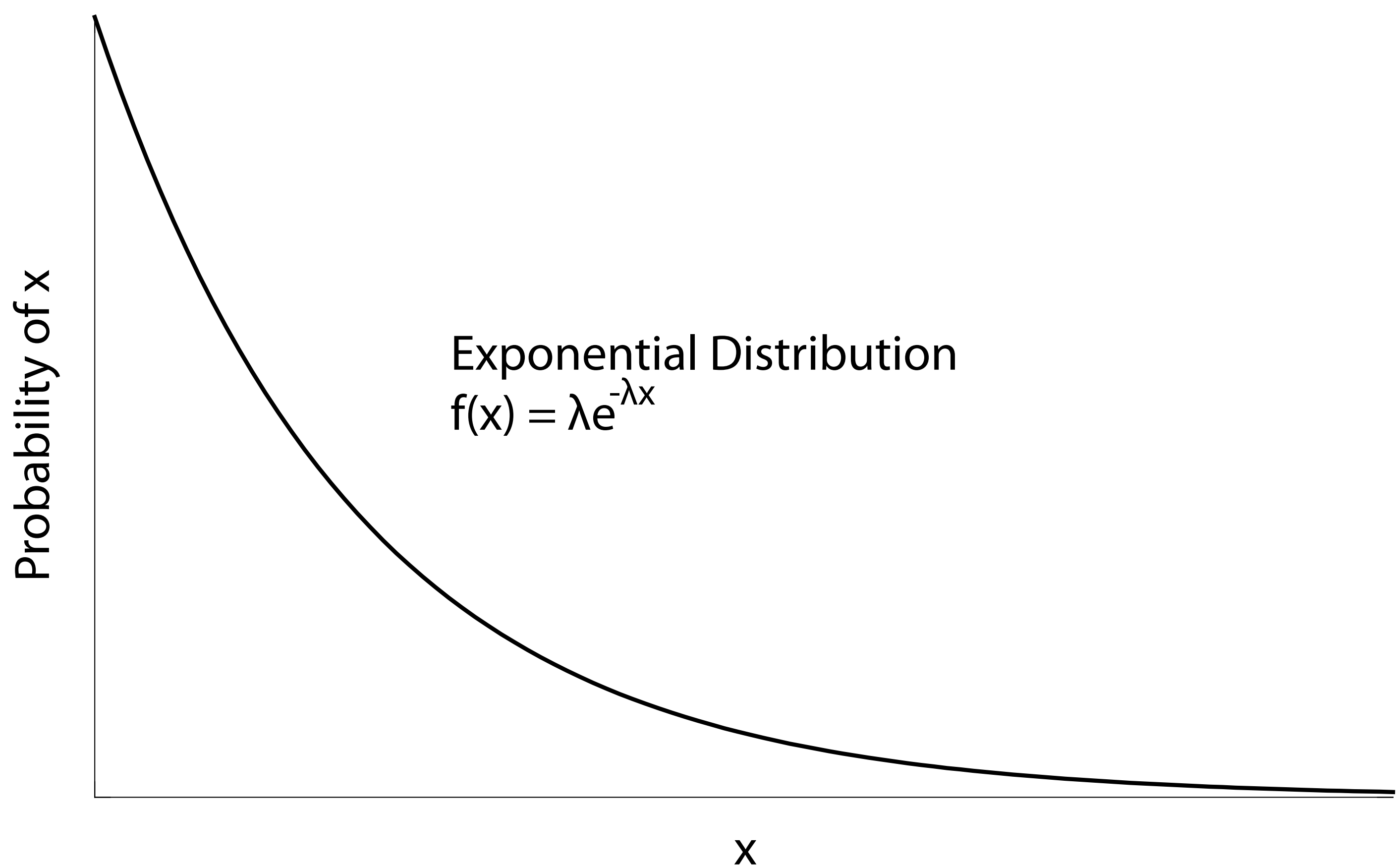


Queueing: Average Case?

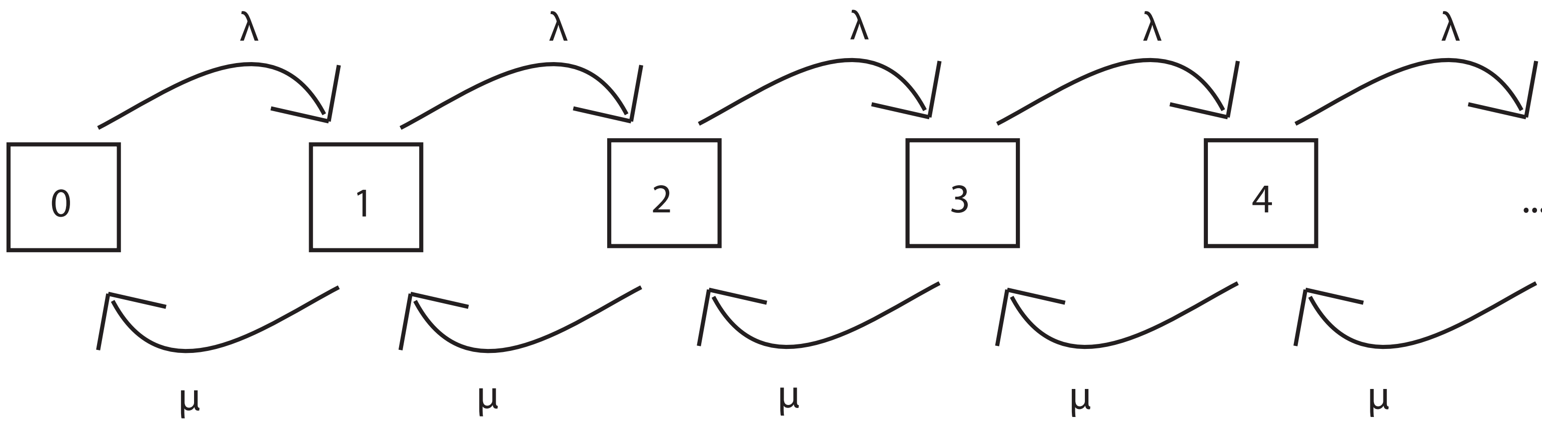
- What is average?
  - Gaussian: Arrivals are spread out, around a mean value
  - Exponential: arrivals are memoryless
  - Heavy-tailed: arrivals are bursty

- Can have randomness in both arrivals and service times

Exponential Distribution

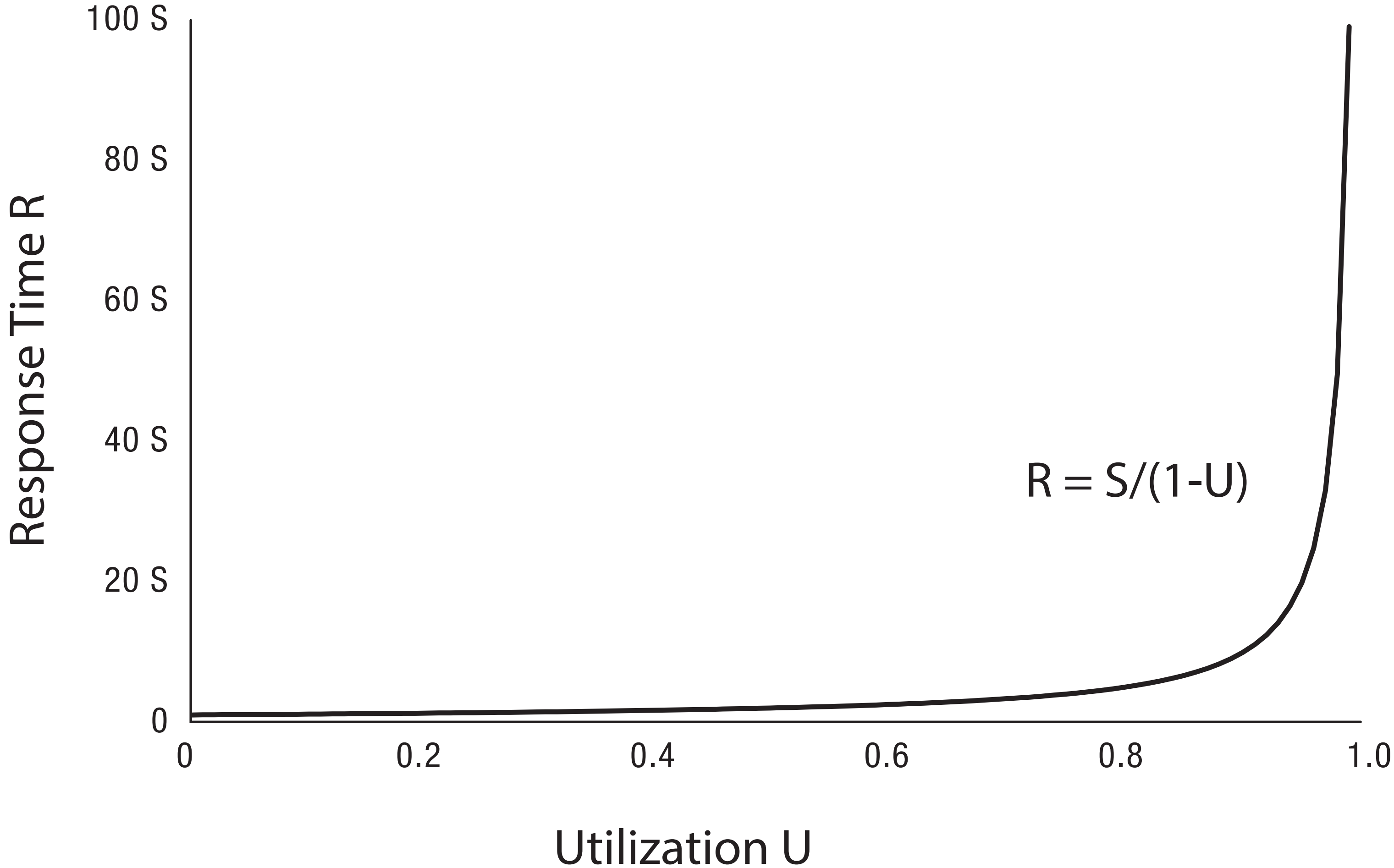


Exponential Distribution



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?

Variance in Response Time

- Exponential arrivals
  - Variance in R = S/(1-U)^2
- What if less bursty than exponential?
- What if more bursty than exponential?

What if Multiple Resources?

- Assuming exponential arrival, service times
- Response time =

  Sum over all i

  Service time for resource i /

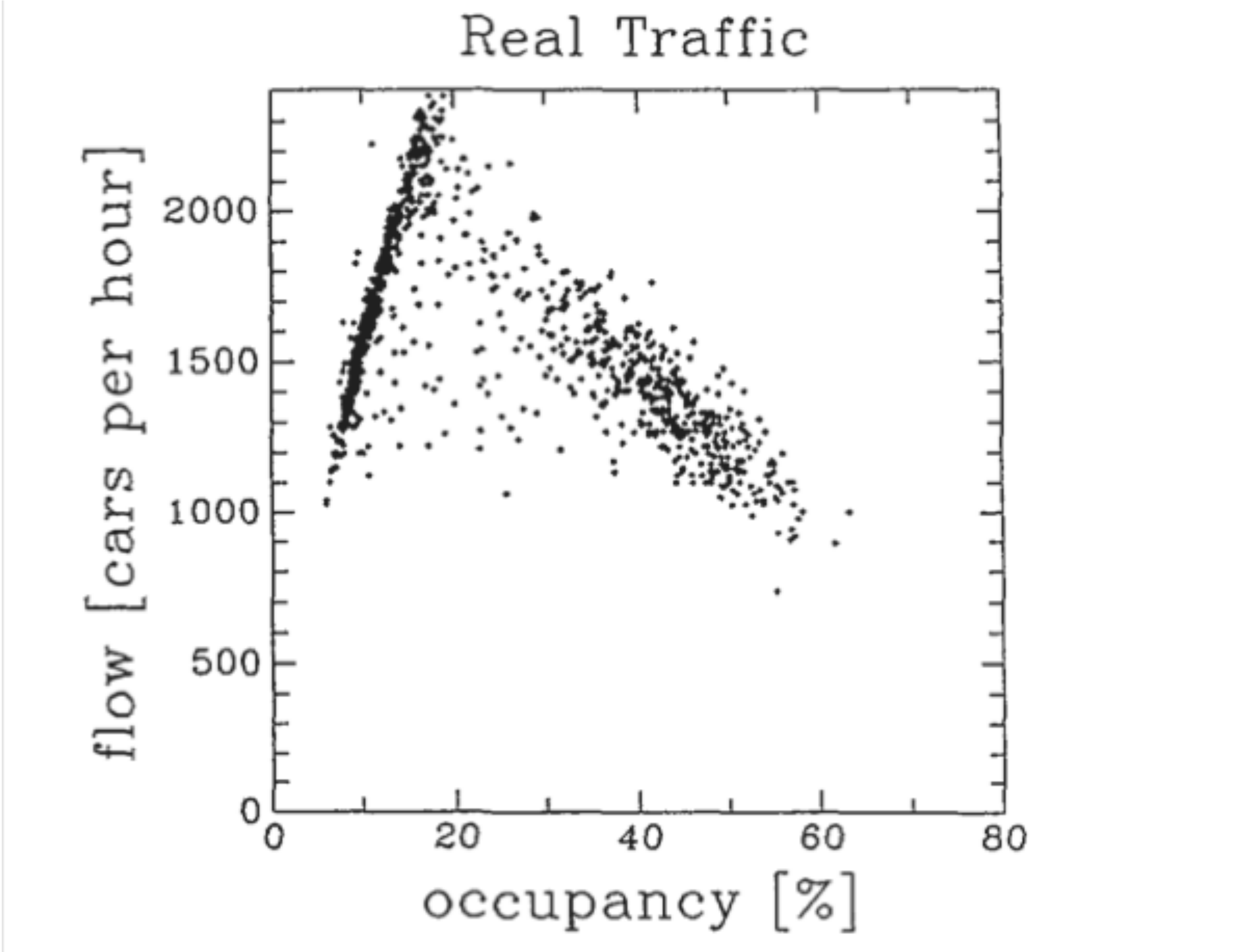
  (1 – Utilization of resource i)

- 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)



Why Do Metro Buses Cluster?

Suppose two Metro buses start 10 minutes apart.

Why might they arrive at the same time?