Introduction to Operating Systems

CPSC/ECE 3220 Summer 2018

Lecture Notes
OSPP Chapter 7 – Part B

(adapted by Mark Smotherman from Tom Anderson's slides on OSPP web site)

Main Points

- Levels of Scheduling
  - Short-term, medium-term, and long-term
- Priority-based policies
  - Problems with static priority (e.g., deadlocks)
  - Priority aging and boosting
- Multiprocessor policies
  - Affinity scheduling, gang scheduling
- Queueing theory
  - Can you predict/improve a system's response time?

Levels of Scheduling

- Short-term scheduler – dispatcher
  - Runs after every interrupt
- Medium-term scheduler – swapper
  - Runs every few seconds
  - Provide a mix of CPU-bound and I/O-bound ready tasks allocated in main memory
- Long-term-scheduler – batch job initiator

Priority-Based Scheduling

- Static priority
  - Can lead to problems including deadlock
- Dynamic priority
  - Priority aging
    - Thread’s priority decreases as it runs
    - Thread’s priority increases as it waits
  - Priority boosting
    - When signaled – e.g., on I/O completion

Spinlock-Based Deadlock

- Two threads – one has low priority and one has high priority
- Spinlocks in each

low priority thread A		high-priority thread B	
...	...		
A1:	spinlock.acquire();	B1:	spinlock.acquire();
A2:	critical section	B2:	critical section
A3:	spinlock.release();	B3:	spinlock.release();
...	...		

Thread A starts with thread B waiting on some other event:

A1 acquires lock
A2 in CS
✉ event wakes up B ✉
B1 in spinlock

Question

- Would priority aging break the deadlock?

Priority Inversion

- Three threads – low, medium, and high priority
- Blocking locks used in low and high priority threads

low priority A	medium priority B	high-priority C
…	…	…
A1: lock.acquire();	B1: work	C1: lock.acquire();
A2: critical section	…	C2: critical section
A3: lock.release();		C3: lock.release();
…	…	…

Thread A starts with threads B and C waiting on events:

A1 acquires lock
A2 in CS
// event wakes up C
C1 blocks in call to acquire
// dispatcher runs
A2 continues
// event wakes up B
B1 nows runs

Priority Inheritance (Donation)

- Priority aging would eventually allow A to run and release lock – but not clear how long this would take

- Is there a way to avoid B preempting A?
  - Priority inheritance – when a thread waits for a lock held by a lower priority thread, the lock holder's priority is temporarily increased to the waiter's priority until the lock is released

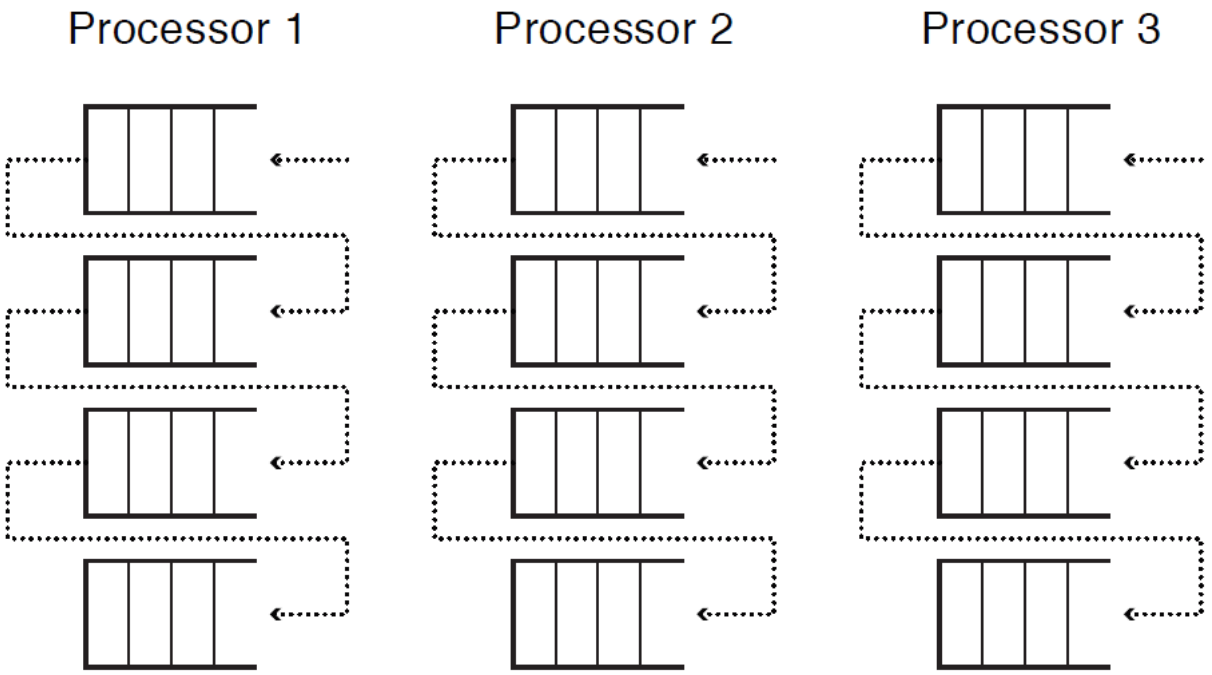
Multiprocessor Scheduling

- 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 CV signal
- Idle processors can steal work from other processors

Per-Processor Multi-level Feedback with Affinity Scheduling



- Rebalance workload only when queues consistently unbalanced (this can be decided by medium-term scheduler)

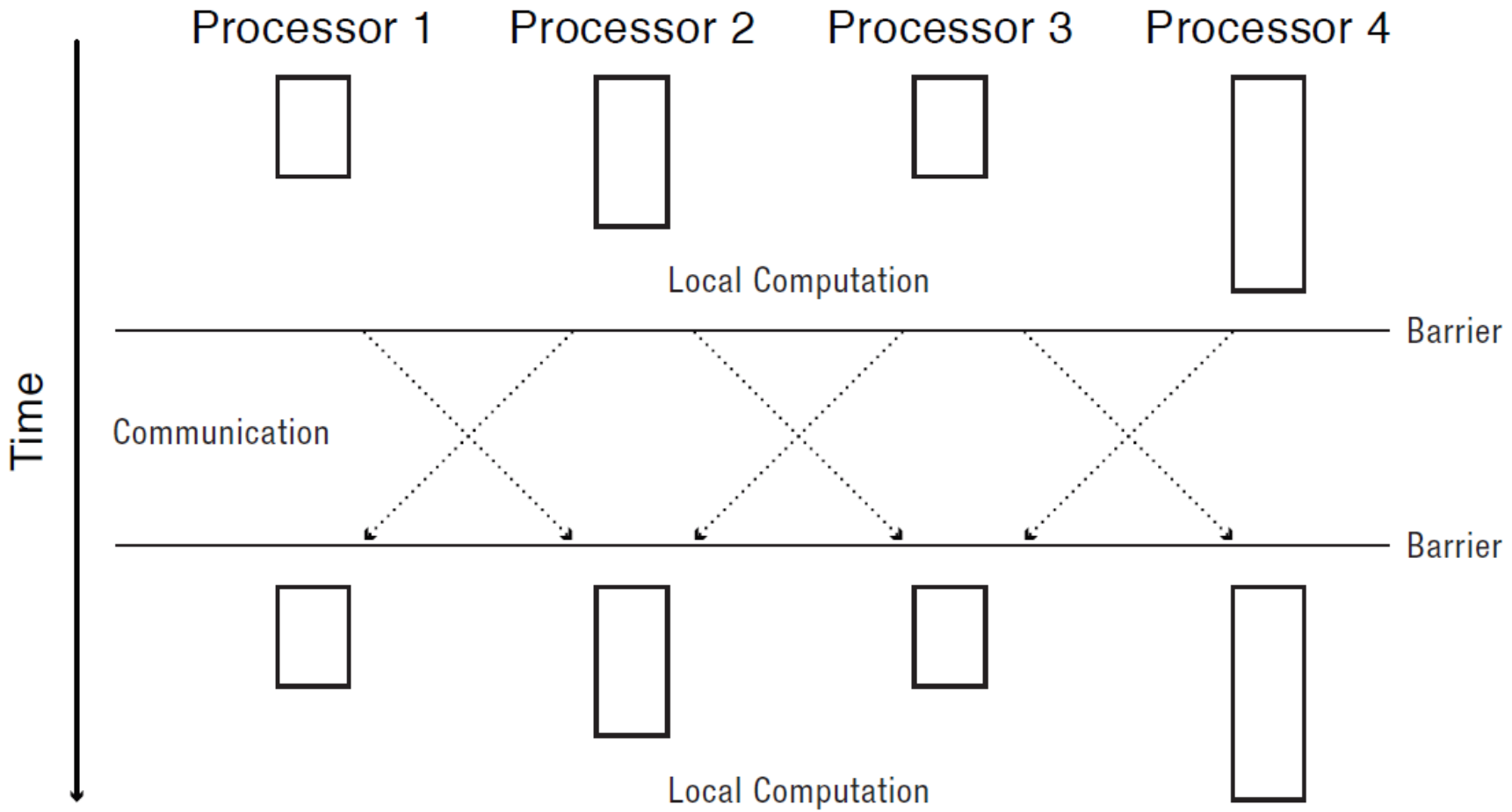
Scheduling Parallel Programs

- 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

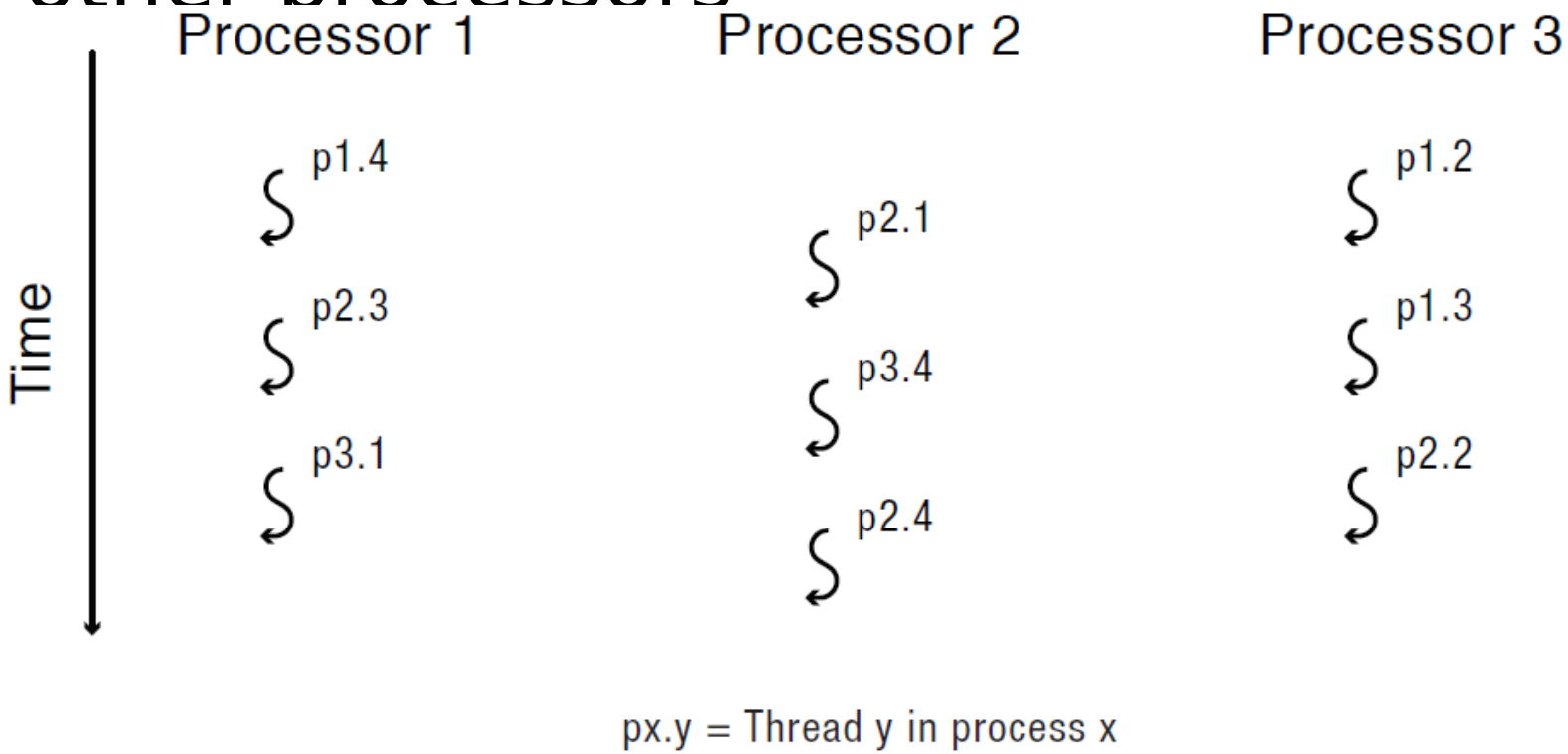
- Loop at each processor:
  - Compute on local data (in parallel)
  - Barrier
  - Send (selected) data to other processors (in parallel)
  - Barrier
- Examples:
  - MapReduce
  - Fluid flow over a wing
  - Most parallel algorithms can be recast in BSP
    - Sacrificing a small constant factor in performance

Longest Thread Controls Overall Performance

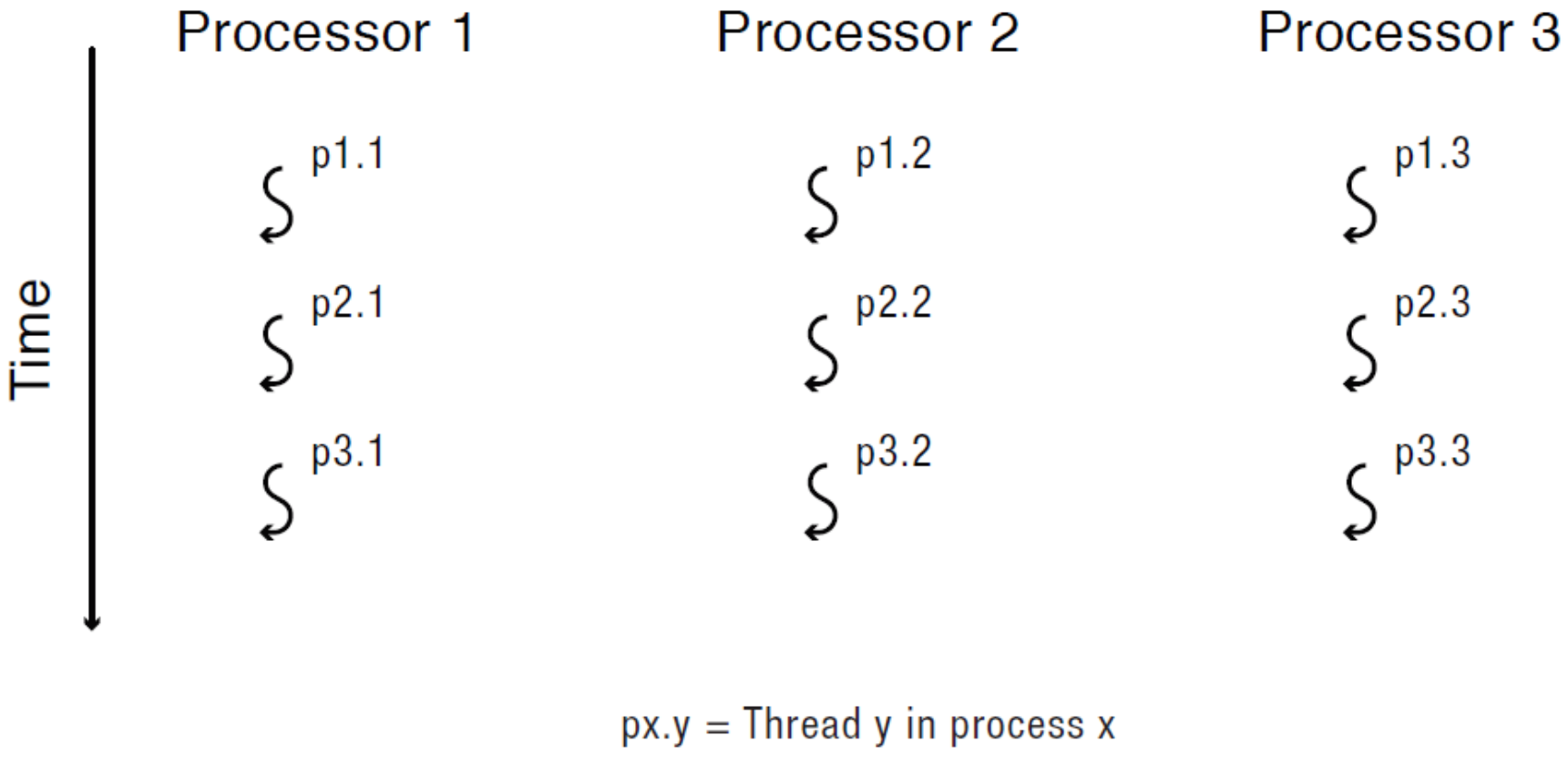


Scheduling Parallel Programs

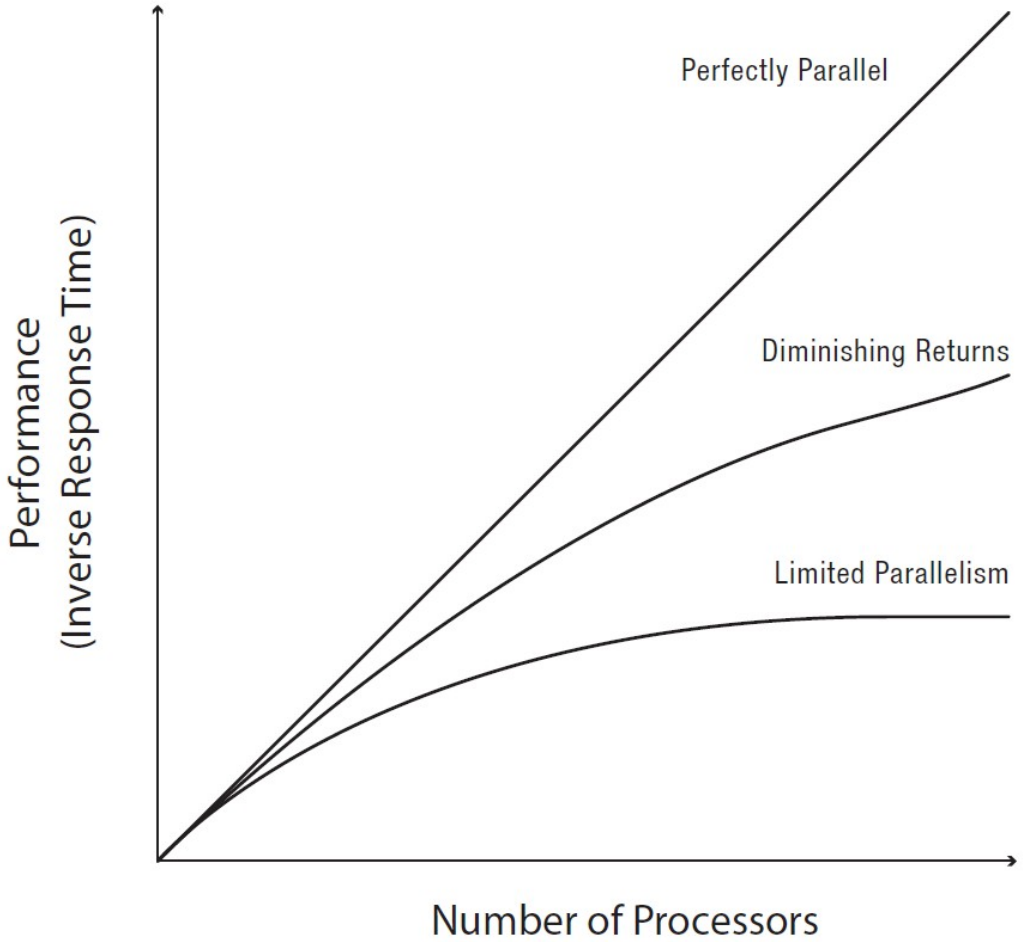
Oblivious: each processor time-slices its ready list independently of the other processors



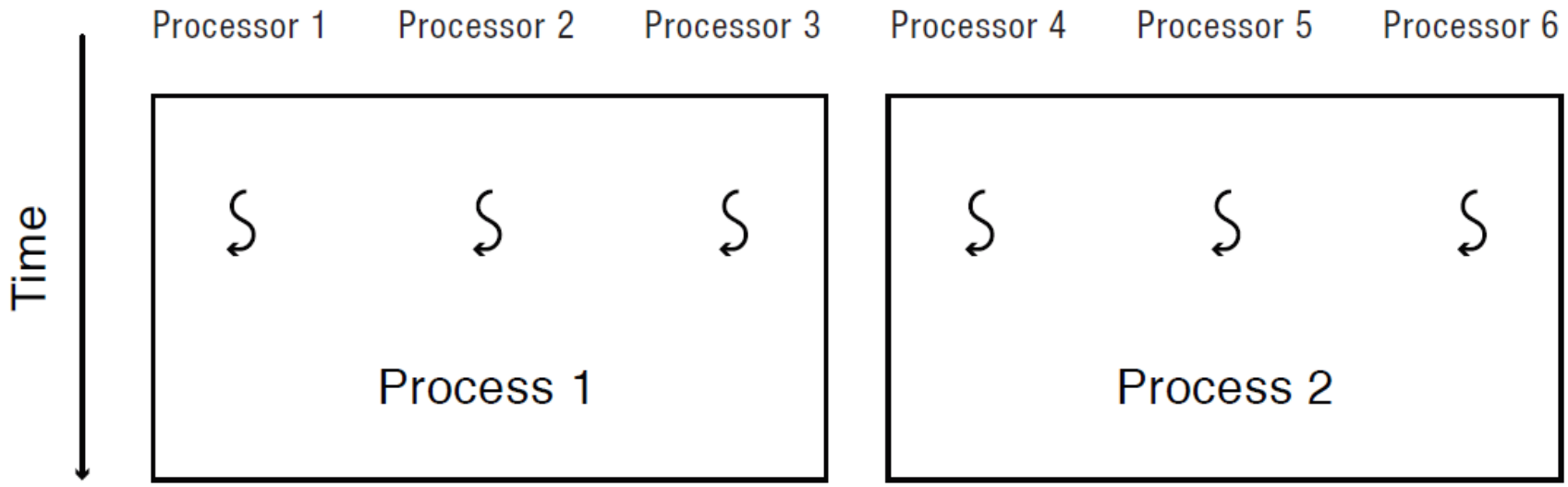
Gang Scheduling



Parallel Program Speedup



Space Sharing



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

Energy-Aware Scheduling

- Turn off some of the processors (cores)
- Adjust the clock frequency of a processor
- Choose among asymmetric cores
  - e.g., ARM big.LITTLE

Real-Time Scheduling

- Computation that must be completed by a deadline if it is to have value
- Example policies
  - EDF – earliest deadline first
  - LST - least slack time first

    (slack time = deadline - current time - service time)
  - RM - rate monotonic

Various Factors In Scheduling Policies

- arrival order
- service time (future knowledge)
- preemption
- time quantum
- number of time slices or time used so far (aging)
- wait time (aging)
- static (base) priority
- dynamic priority
- priority boost
- load factor
- type of task (CPU bound / I/O bound)
- source of task (for fair share or priority bands)
- number of processors
- processor affinity
- load balancing
- priority inheritance
- energy-aware
- deadlines

Overload Management

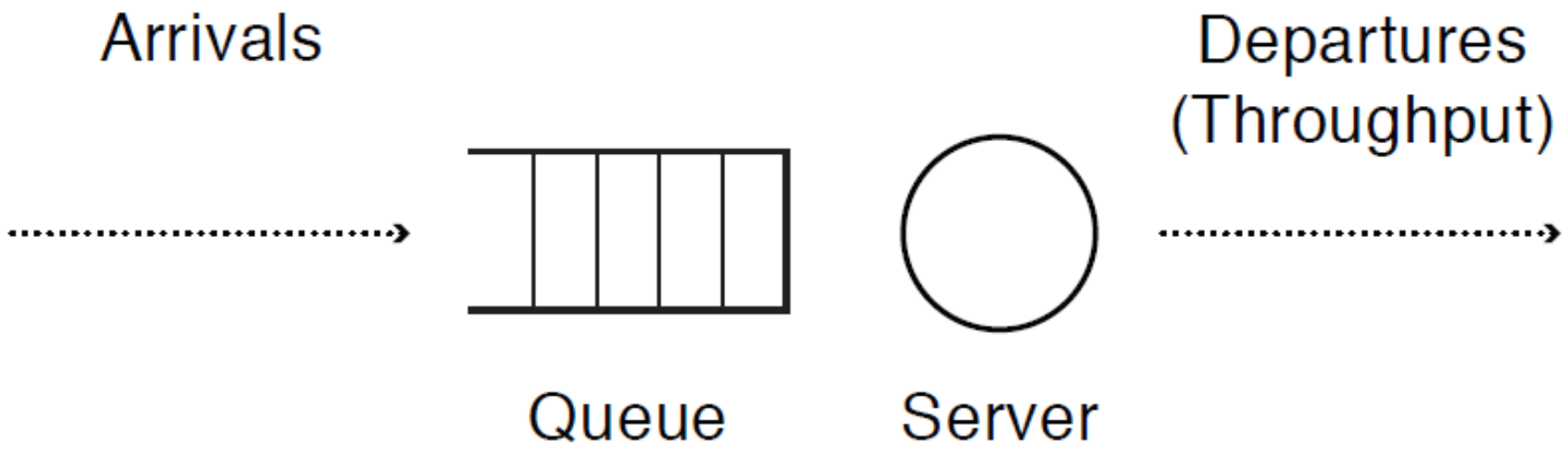
- What if arrivals occur faster than service can handle them
  - If do nothing, response time will become infinite
- Turn users away?
  - Which ones? Average response time is best if turn away users that have the highest service demand
  - Example: Highway congestion
- Degrade service?
  - Compute result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

(if time permits)

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



ssumption: average performance in a stable system
here the arrival rate (ƛ) matches the departure rate

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?

Question

- From example:

  X = 100 task/sec

  R = 50 ms/task

  S = 5 ms/task

  W = 45 ms/task

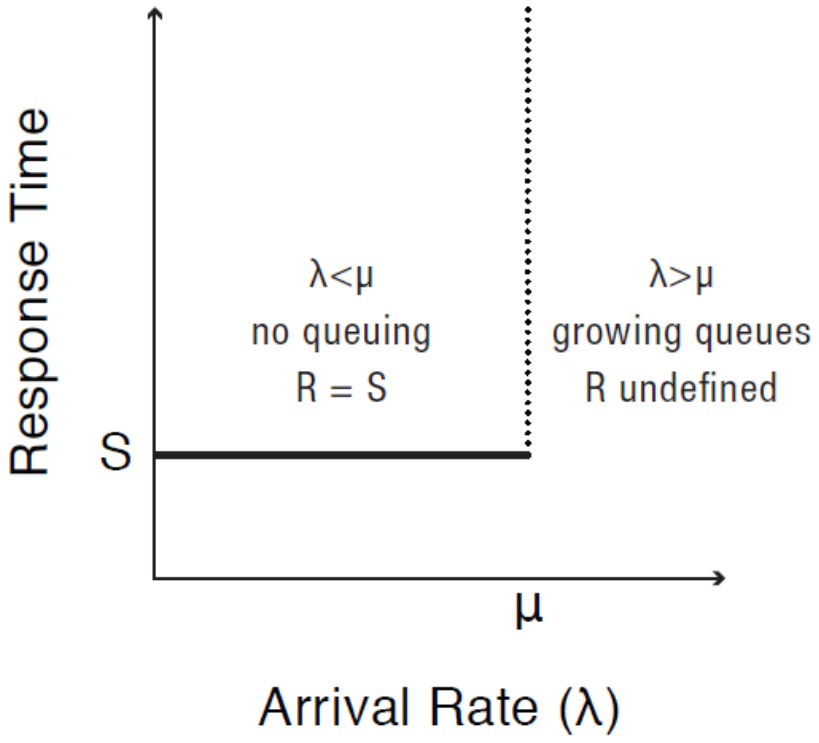
  Q = 4.5 tasks

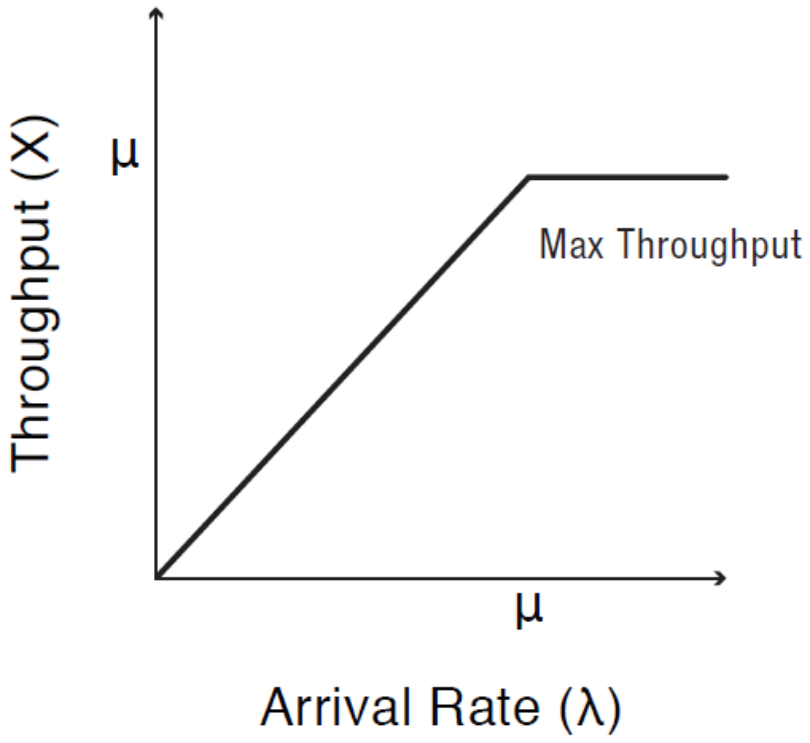
- Why is W = 45 ms and not 4.5 * 5 = 22.5 ms?
  - Hint: what if S = 10ms? S = 1ms?

Queueing

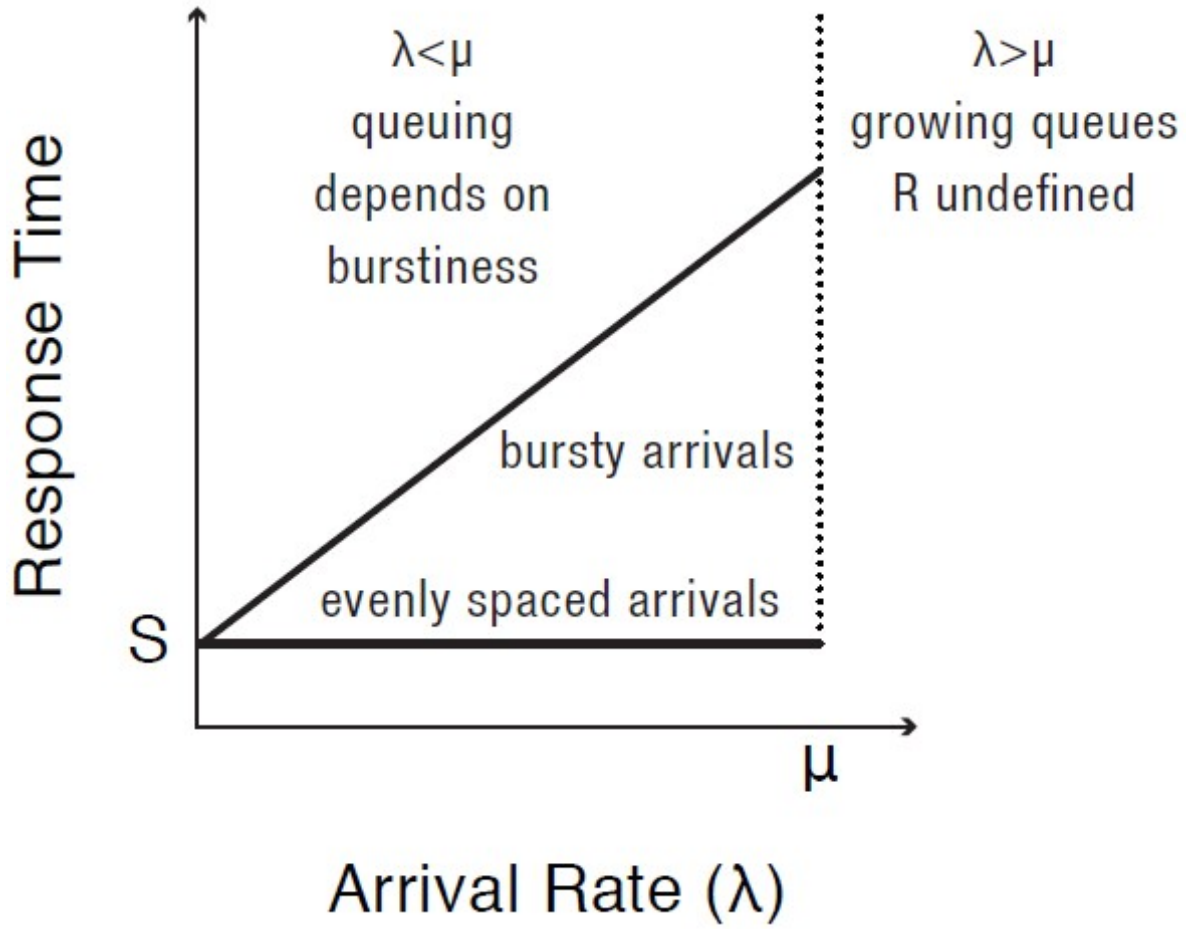
- What is the best case scenario for minimizing queueing delay?
  - Keeping arrival rate, service time constant
- What is the worst case scenario?

Queueing: Best Case





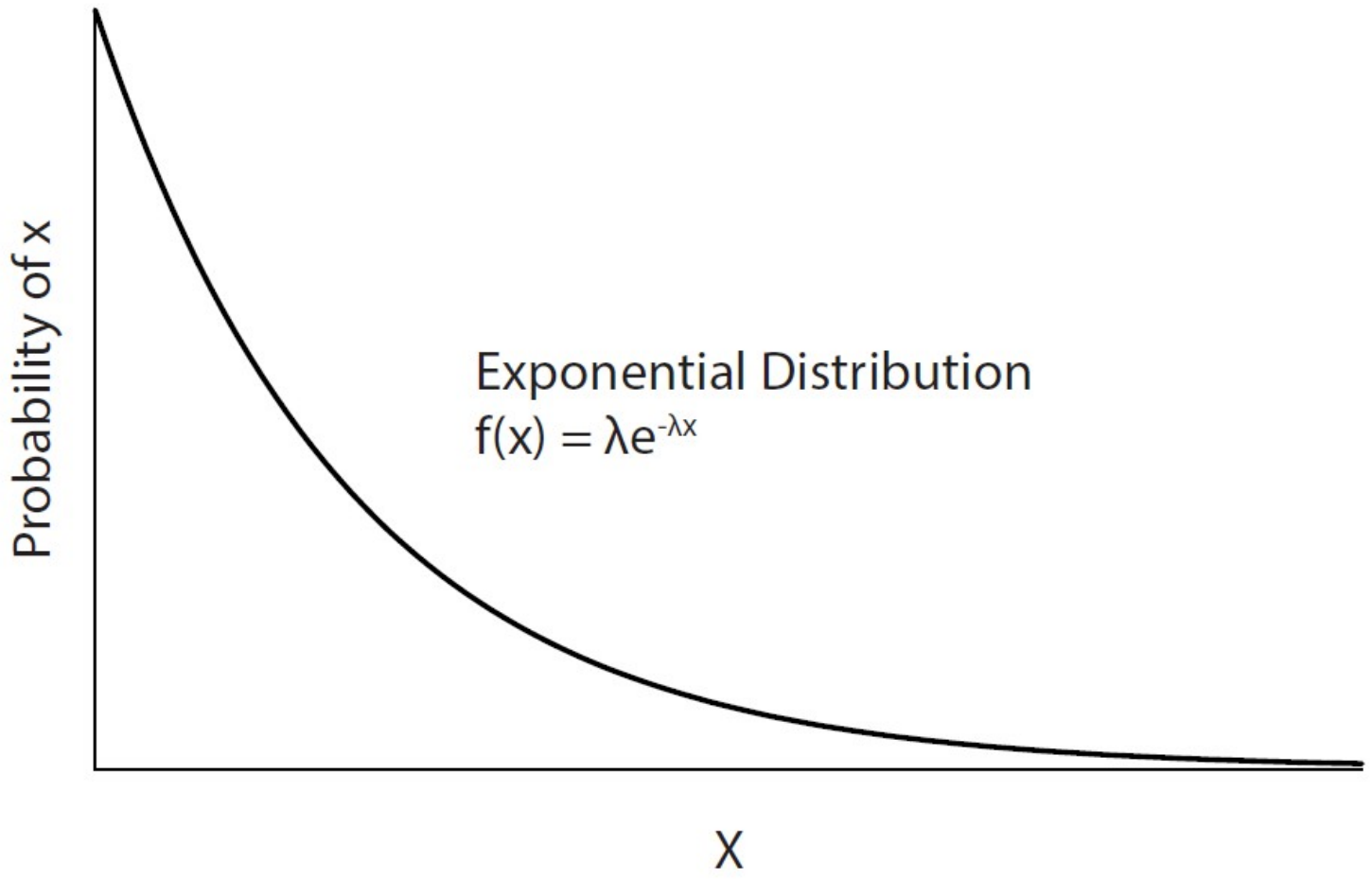
Response Time: Best vs. Worst Case



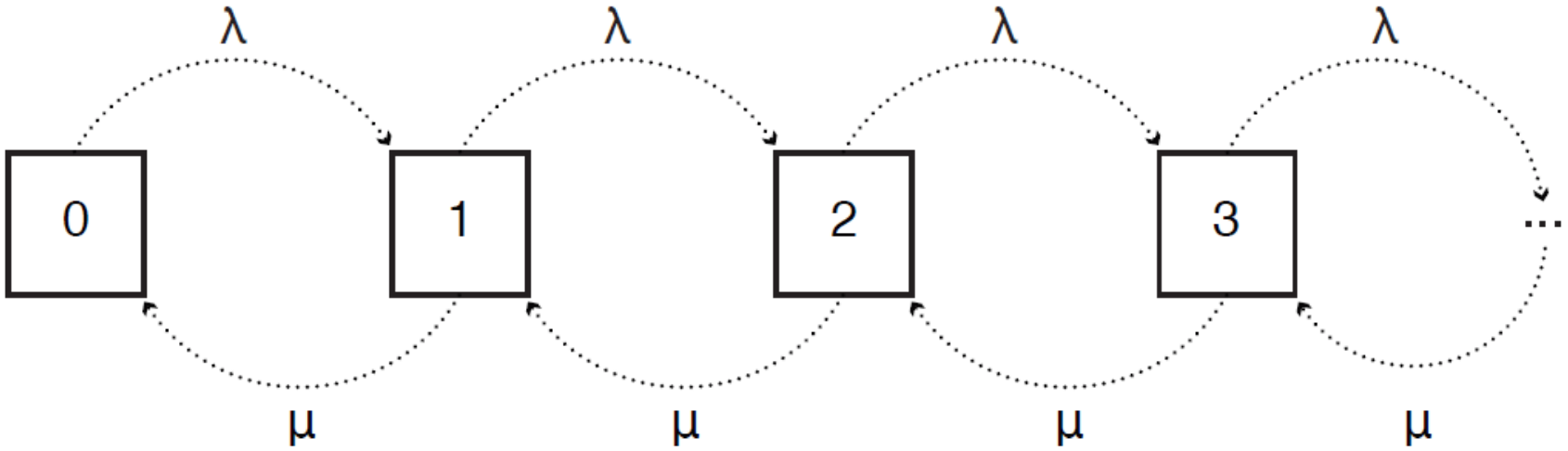
Queueing: Average Case?

- 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

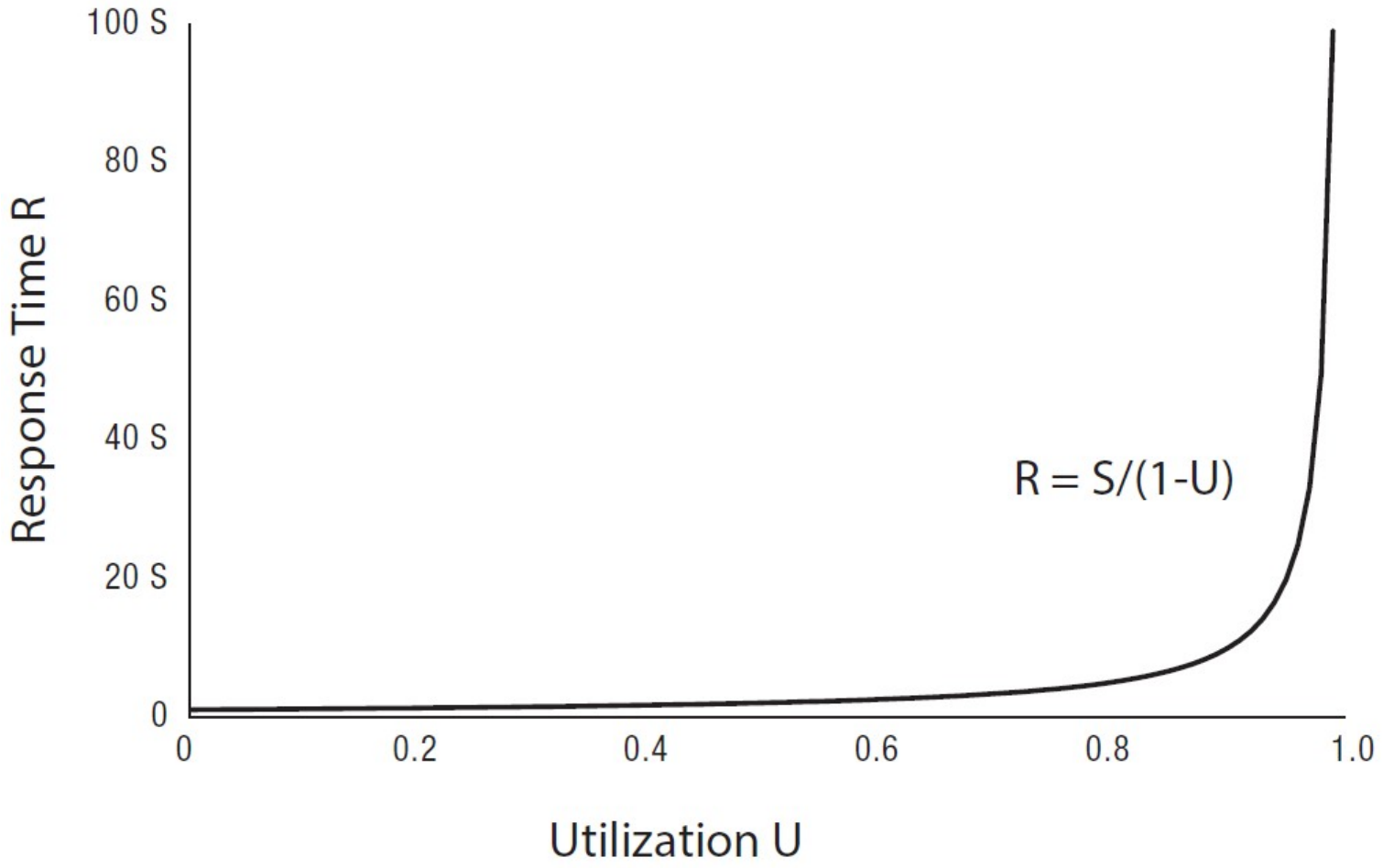


Exponential Distribution



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?

- 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