SCHEDULING

Slides adapted from Tom Anderson's "Operating Systems: Principles and Practice"

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 a system’s response time?

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

- Example: memcached
  - Facebook cache of friend lists, …

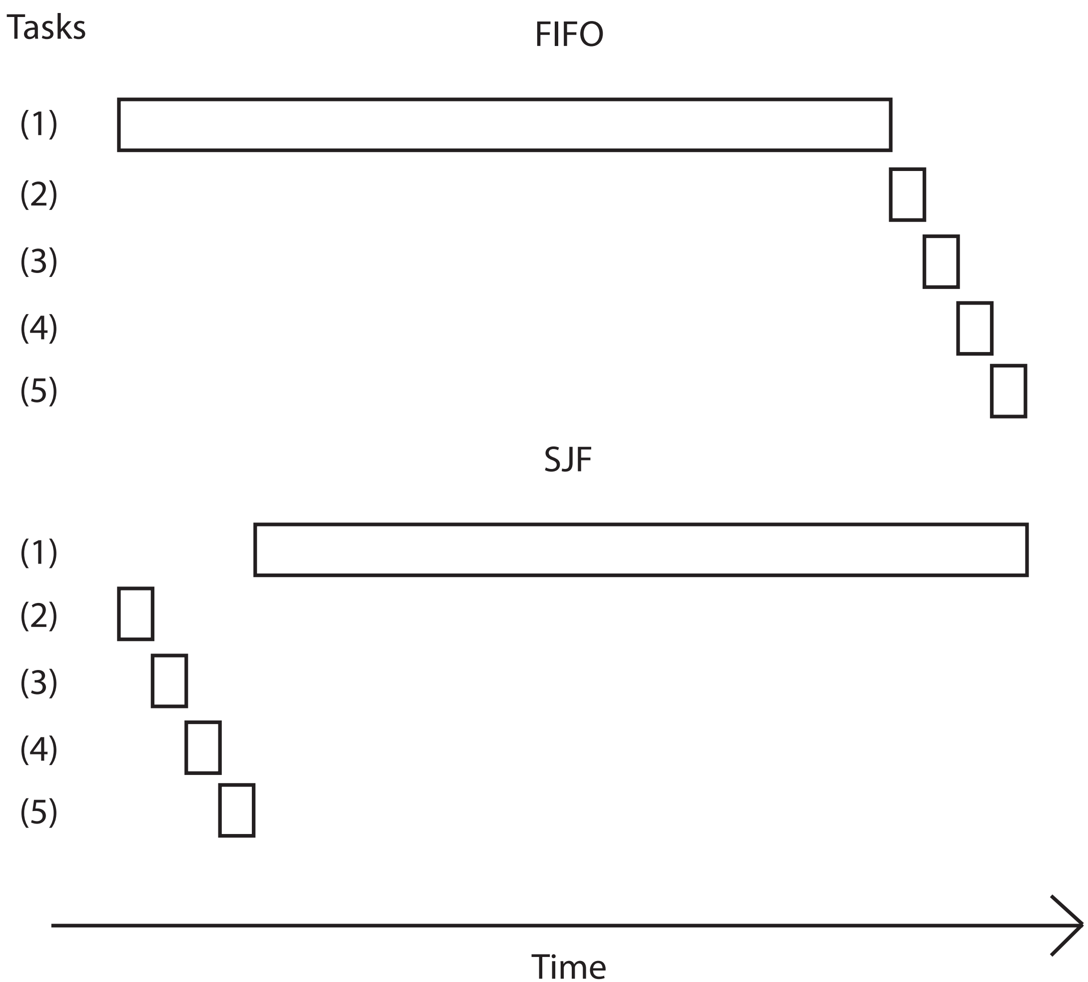
- On what workloads is FIFO particularly bad?

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



Slides adapted from Tom Anderson's "Operating Systems: Principles and Practice"

Shortest Job First

- Claim: SJF is optimal for average response time
  - Why?
- Pessimal?
- Does SJF have any downsides?
  - starvation
  - variance in response time.

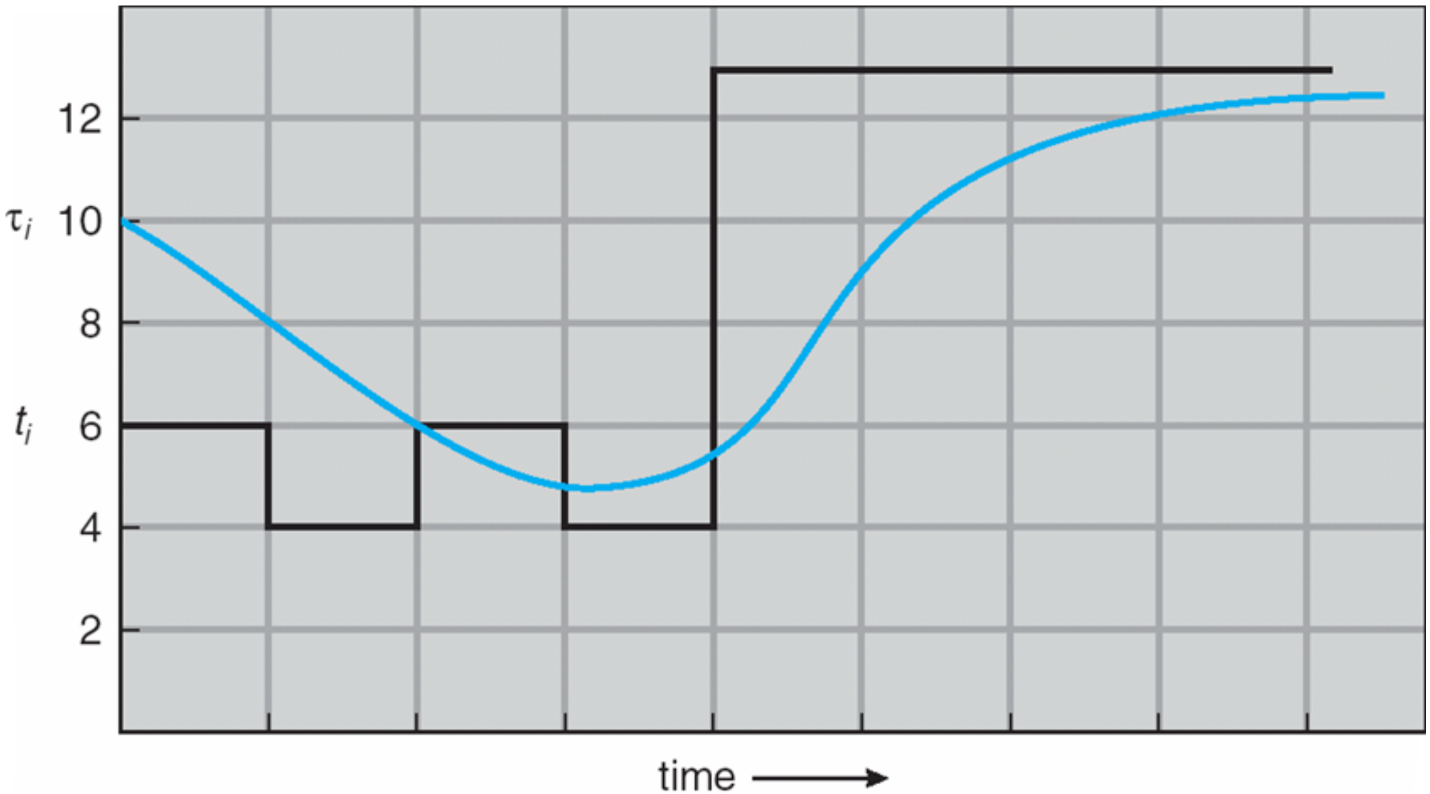
Determining Length of Next CPU Burst

- Can only estimate the length

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

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define: $\tau_{n=1} = \alpha\, t_n + (1 - \alpha) * \tau_n$

Prediction of the Length of the Next CPU Burst



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

Examples of Exponential Averaging

- α =0
  - $\tau n+1 = \tau n$
  - Recent history does not count
- α =1
  - $\tau n+1 = \alpha\ tn$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau n+1 = \alpha\ tn + (1 - \alpha)\alpha\ tn -1 + \ldots + (1 - \alpha\ )j\ \alpha\ tn -j + \ldots + (1 - \alpha\ )n +1\ \tau 0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

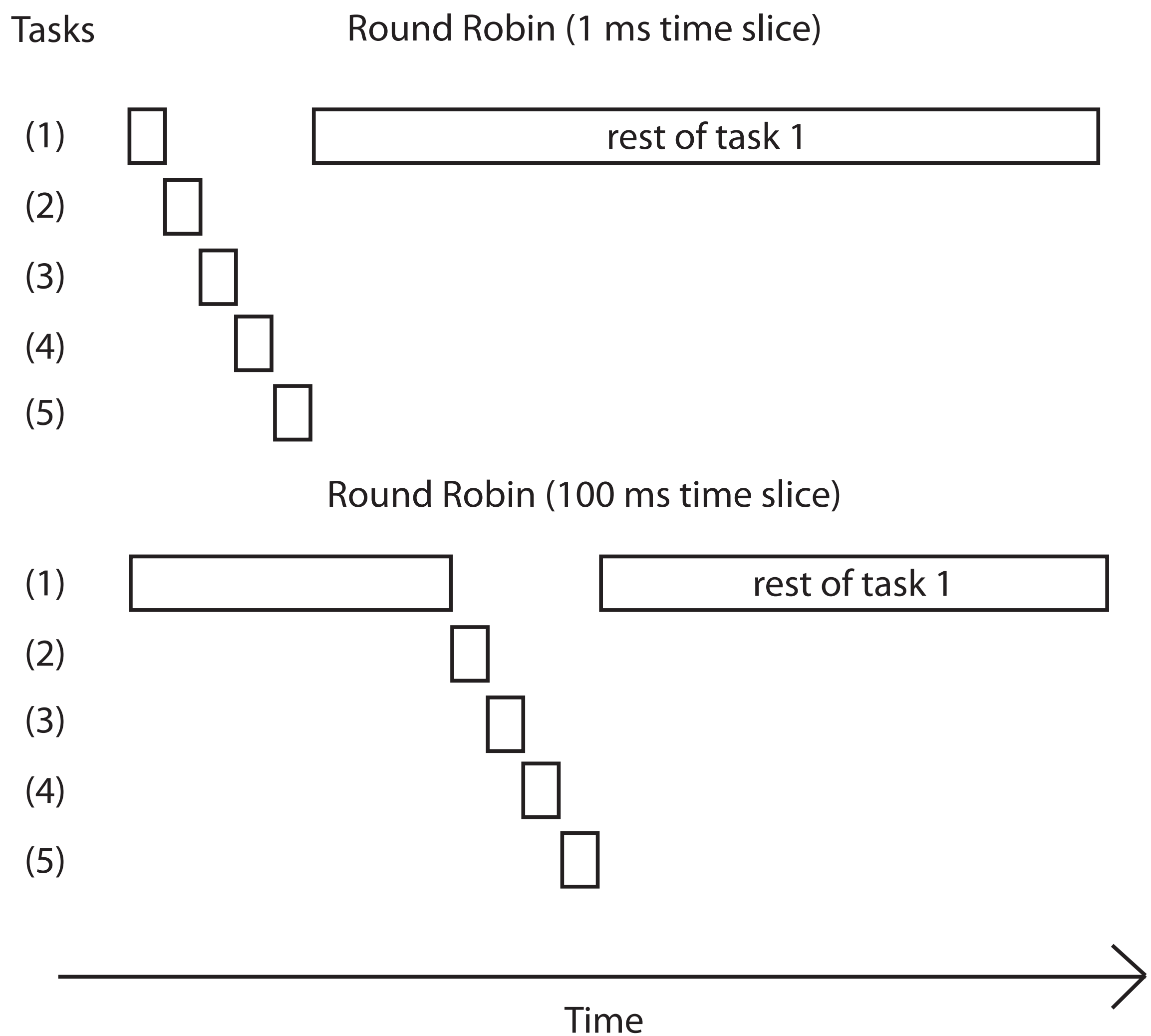
Round Robin

- 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?

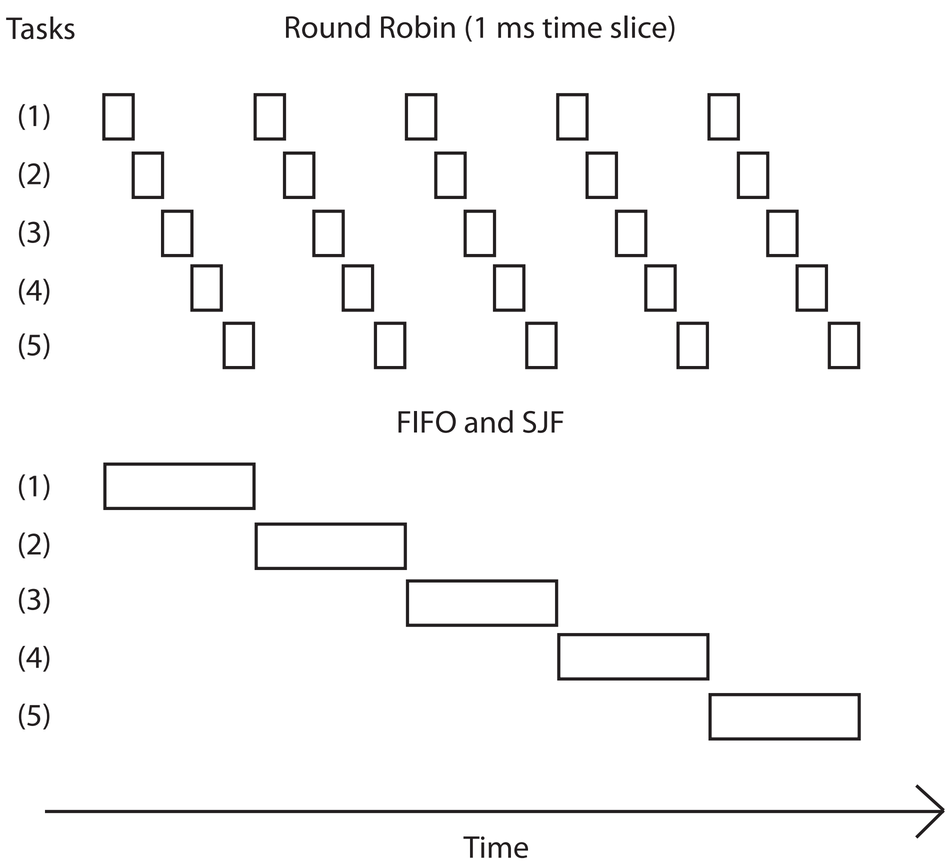
- Usually it is set between 10 and 100ms
  - A common approach is have 80% of the tasks complete in a single execution

Round Robin


Tasks
Round Robin (1 ms time slice)
(1)
rest of task 1
(2)
(3)
(4)
(5)
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?

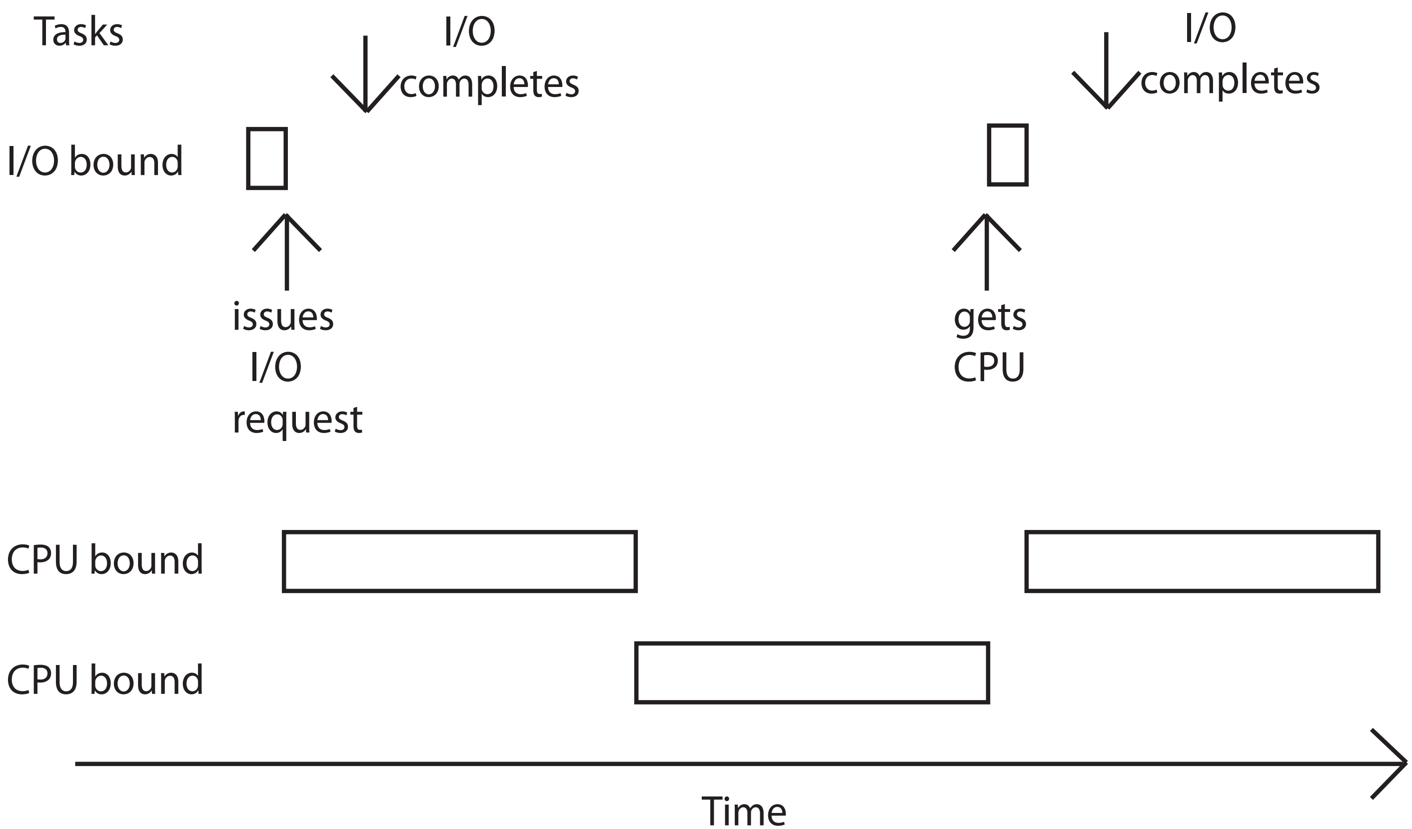


No

Round Robin vs. Fairness

- Is Round Robin always fair?

Mixed workload



No

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
  - Schedule the smallest task first, then split the remaining time using max-min

Multi-level Feedback Queue (MFQ)

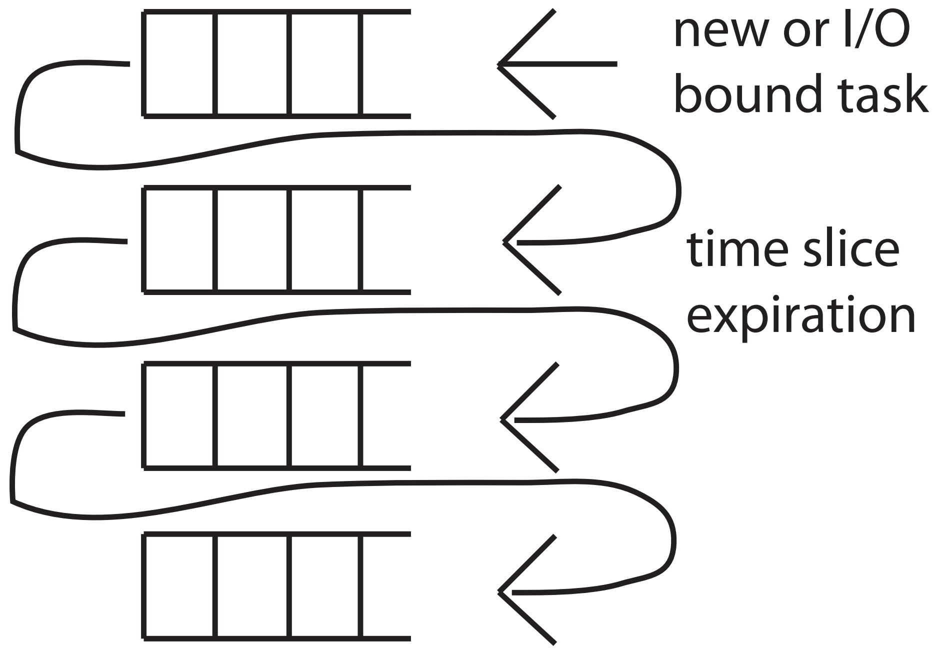
- Goals:
  - Responsiveness
  - Low overhead
  - Starvation freedom
  - Some tasks are high/low priority
  - Fairness (among equal priority tasks)
- Not perfect at any of them!
  - Used in Linux (and probably Windows, MacOS)

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
- Preemption
  - Tasks with more priority preempt tasks with less priority

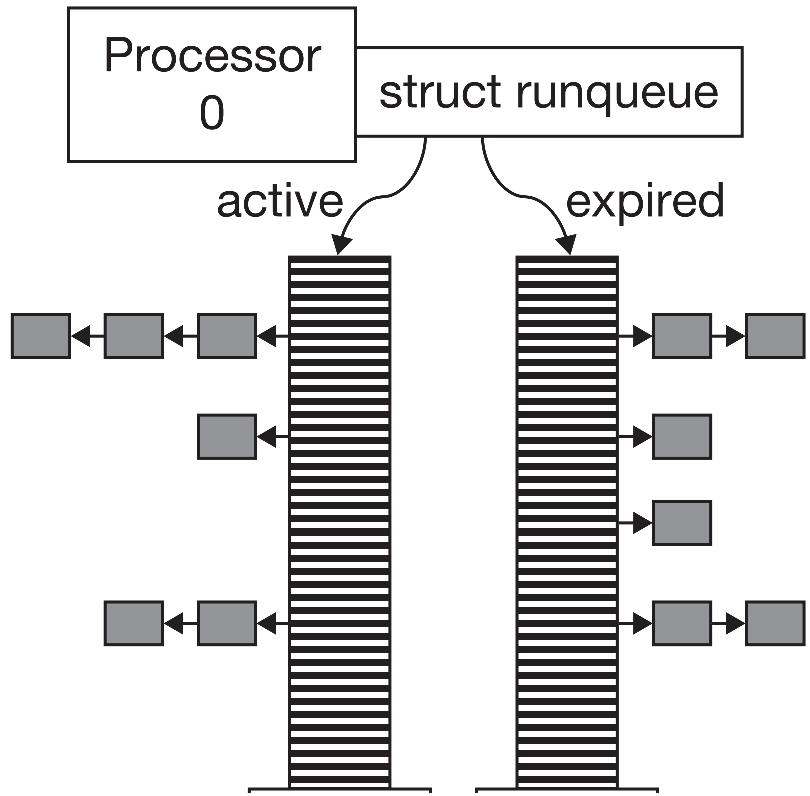
MFQ

Priority	Time Slice (ms)	Round Robin Queues
1	10	
2	20	
3	40	
4	80	



Fairness

- Give priority to tasks that have quantum left
- Example the O(1) scheduler previously implemented in Linux



Uniprocessor Summary

- 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

- 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.
- 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.

Multiprocessor Scheduling

- What would happen if we used MFQ on a multiprocessor?
  - Contention for scheduler spinlock

Multiprocessor Scheduling

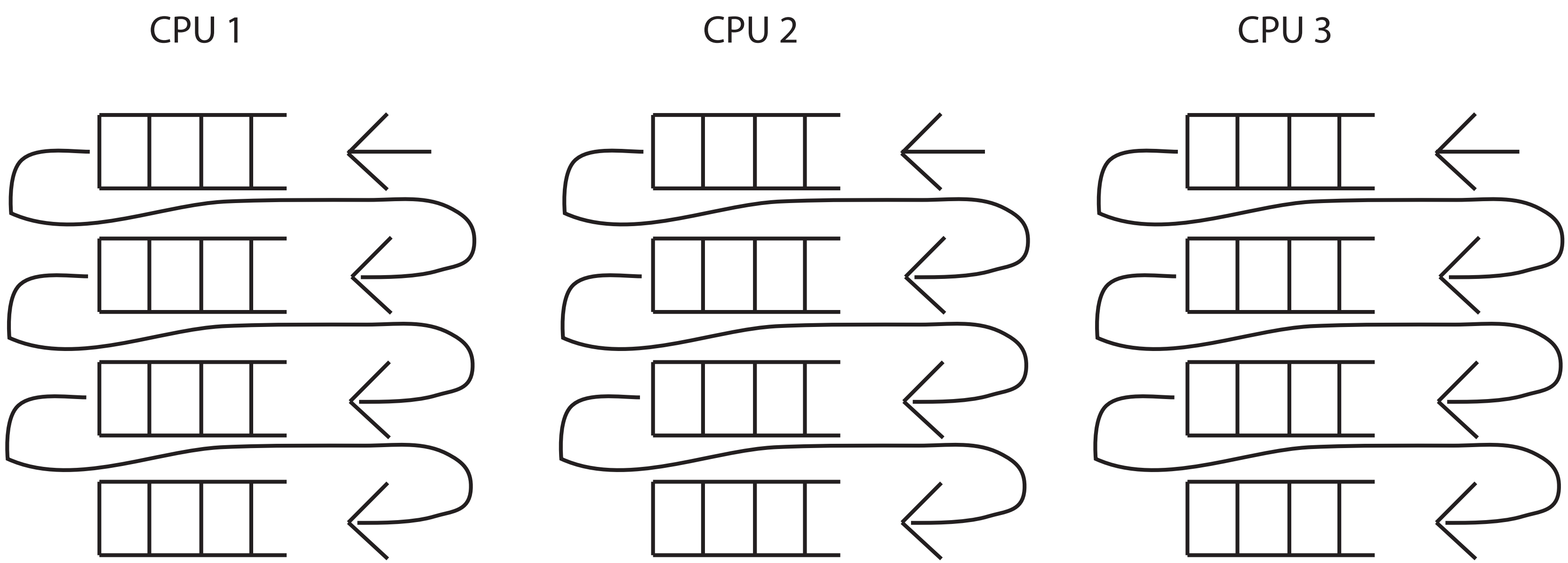
- On modern processors, the CPU is 100x slower on a cache miss
- Cache effects of a single ready list:
  - Cache coherence overhead
    - MFQ data structure would ping between caches
    - Fetching data from other caches can be even slower than re-fetching from DRAM
  - Cache reuse
    - Thread's data from last time it ran is often still in its old cache

Amdahl's Law

- Speedup on a multiprocessor limited by whatever runs sequentially
- Runtime >= Sequential portion + parallel portion/#CPUs

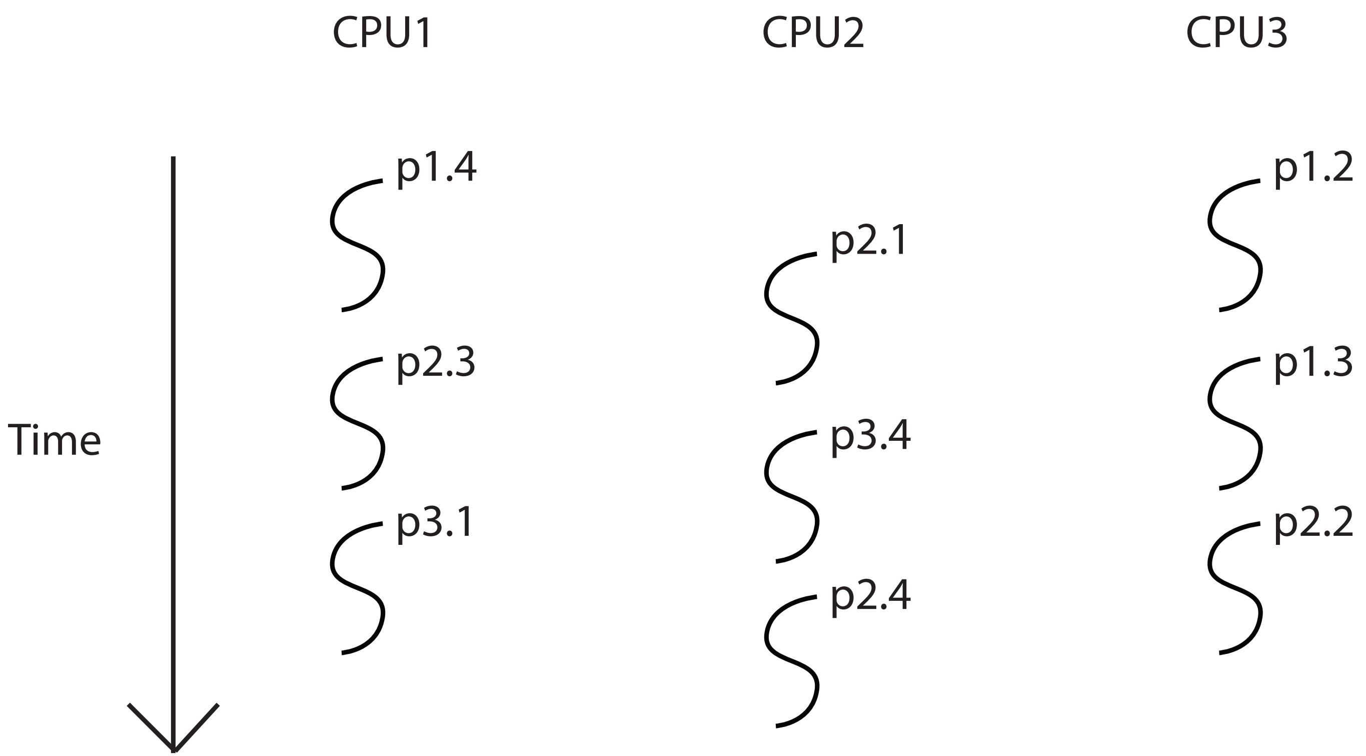
- Example:
  - Suppose scheduler lock used 0.1% of the time
  - Suppose scheduler lock is 50x slower because of cache effects
  - Runtime >= 5% + 95%/# CPUs
    - System is only 2.5x faster with 100 processors than 10

Per-Processor Multi-level Feedback: Affinity Scheduling



Scheduling Parallel Programs

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

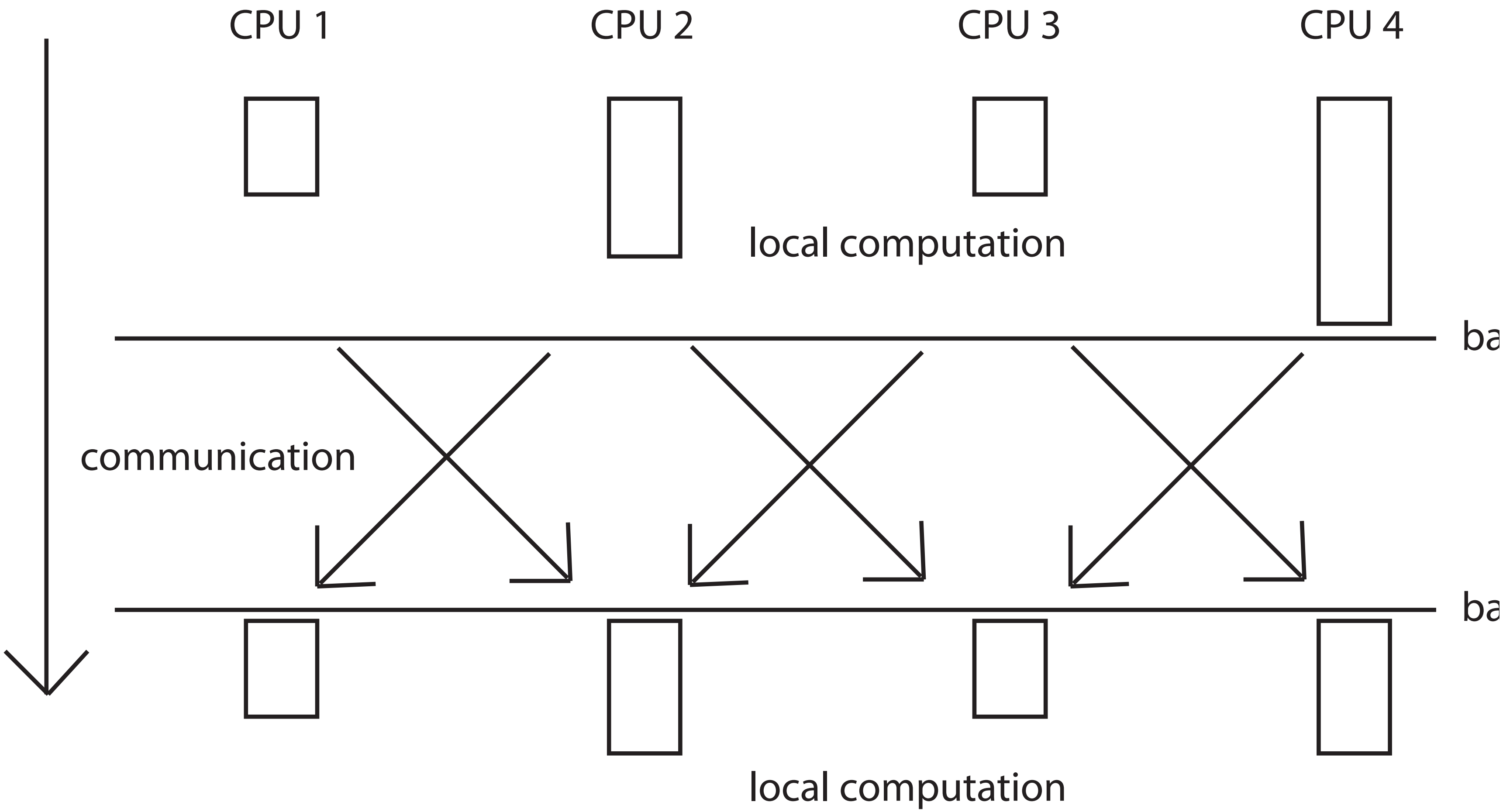


px.y = thread y in process x

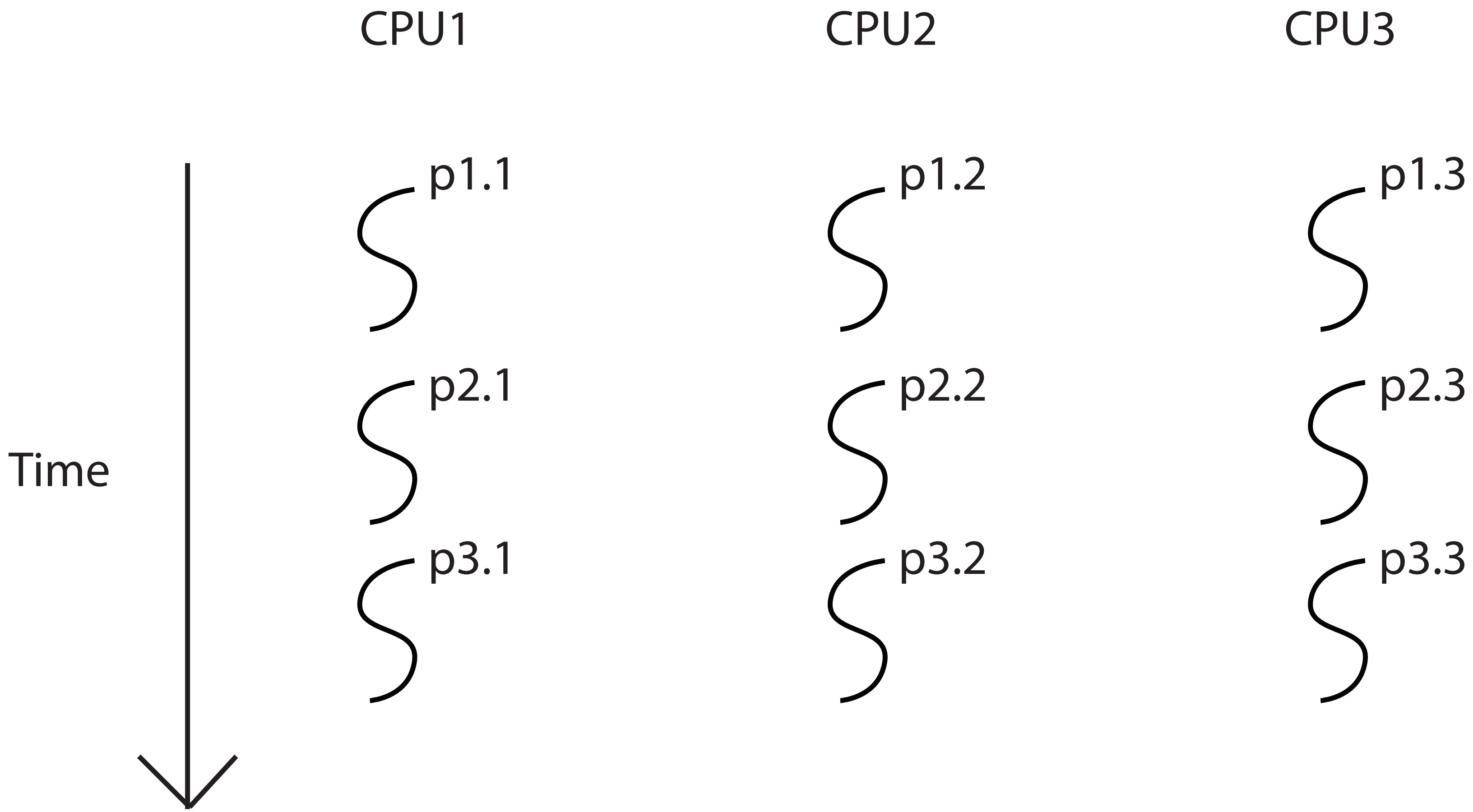
Scheduling Parallel Programs

- 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 Parallel Program

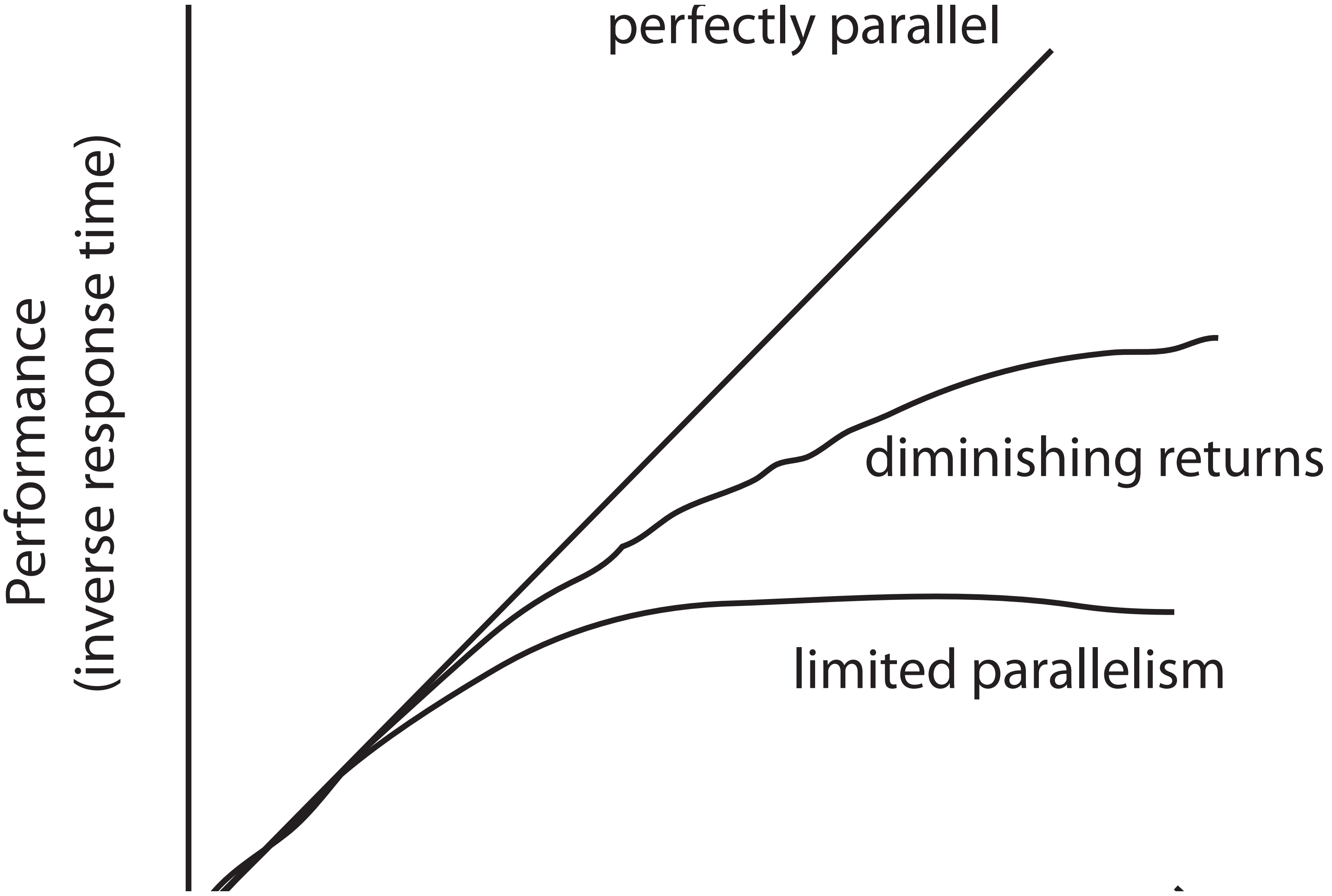


Co-Scheduling

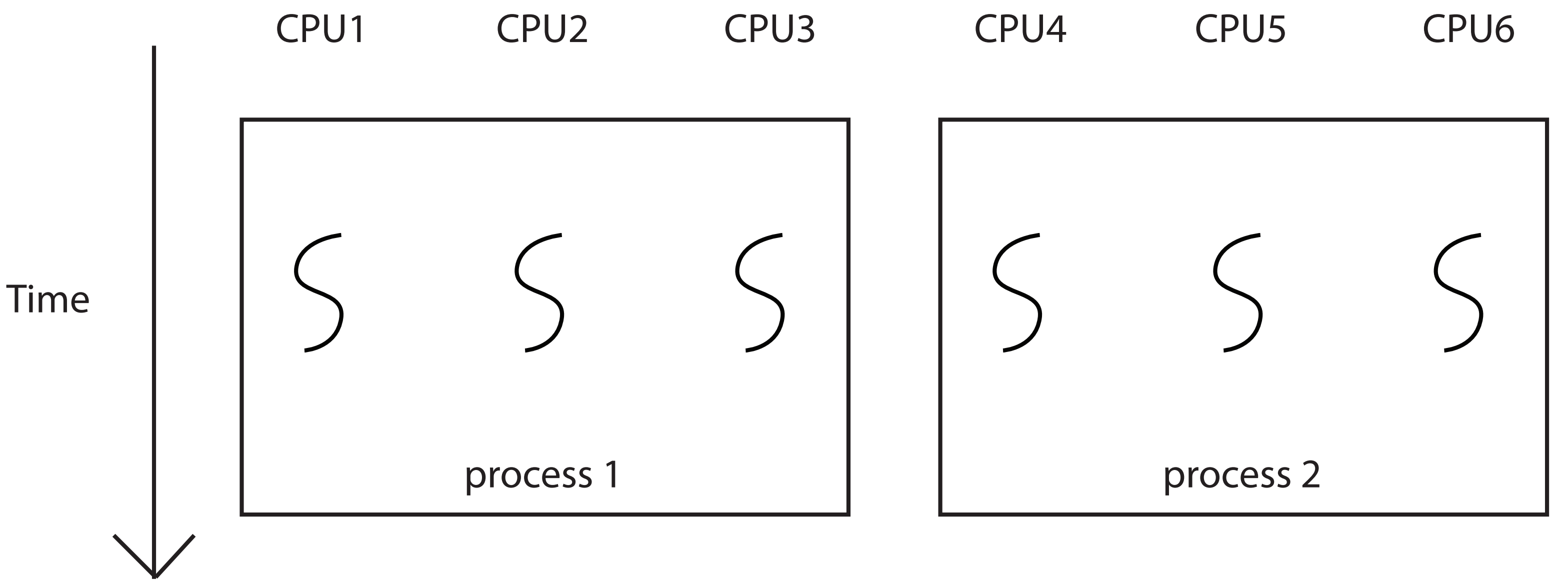


px.y = thread y in process x

Amdahl's Law, Revisited



Space Sharing

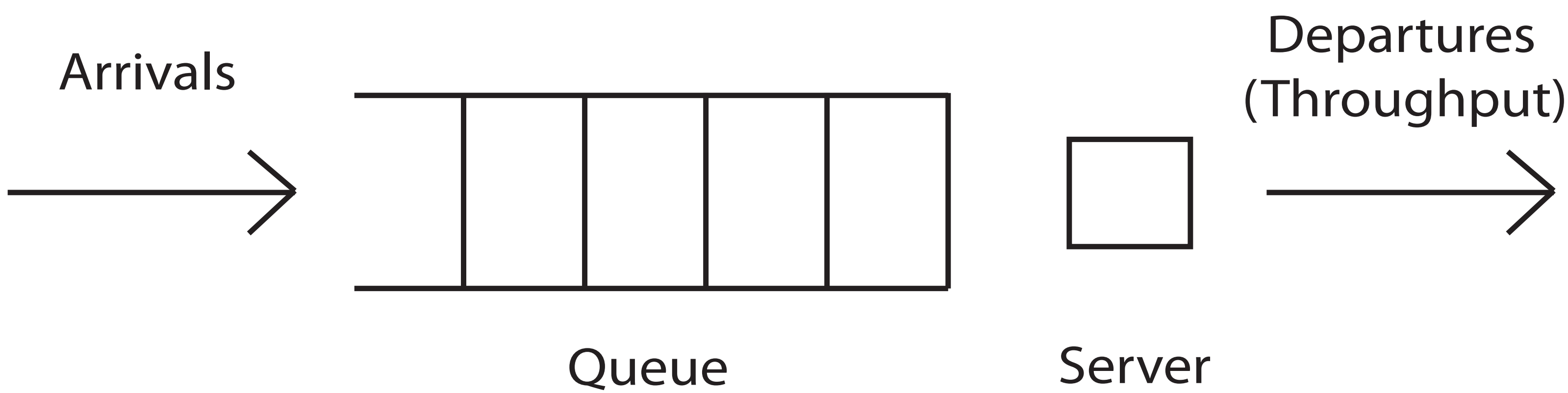


Scheduler activations: kernel informs user-level library as to # of processors assigned to that application, with upcalls every time the assignment changes

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



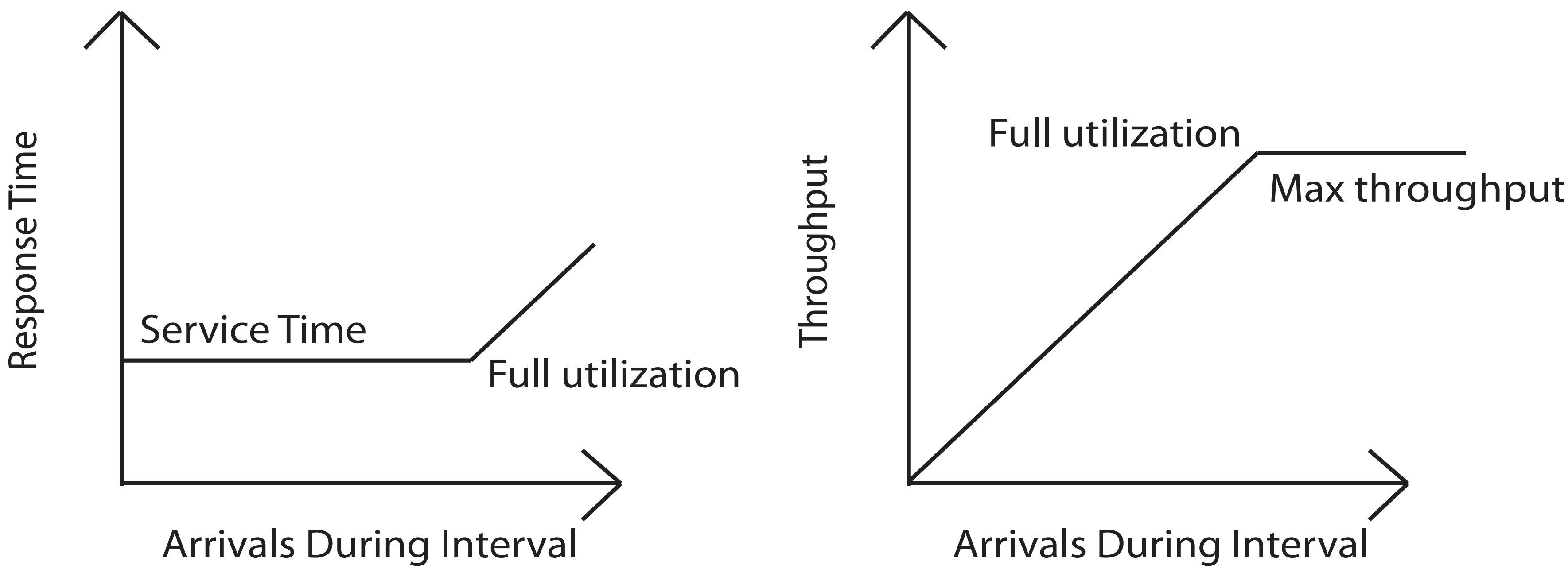
Definitions

- Queueing delay: wait time
- Service time: time to service the request
- Response time = queueing delay + service time
- Utilization: fraction of time the server is busy
  - Service time * arrival rate
- Throughput: rate of task completions
  - If no overload, throughput = arrival rate

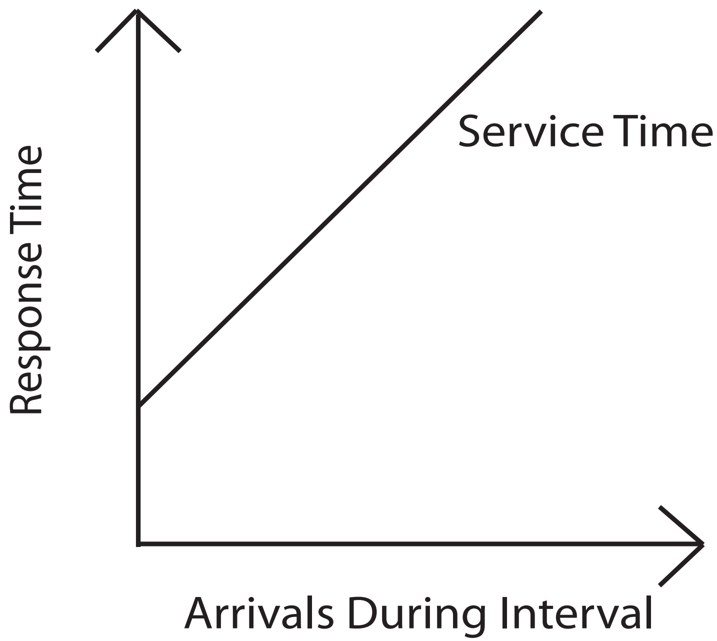
Queueing

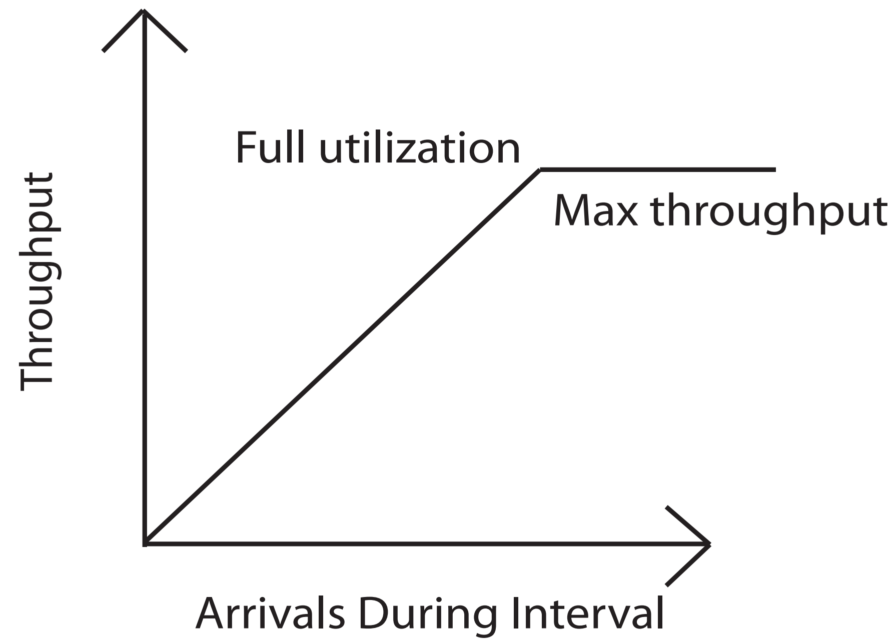
- What is the best case scenario for minimizing queueing delay?
  - Keeping arrival rate, service time constant
- What is the worst case scenario?

Queueing: Best Case



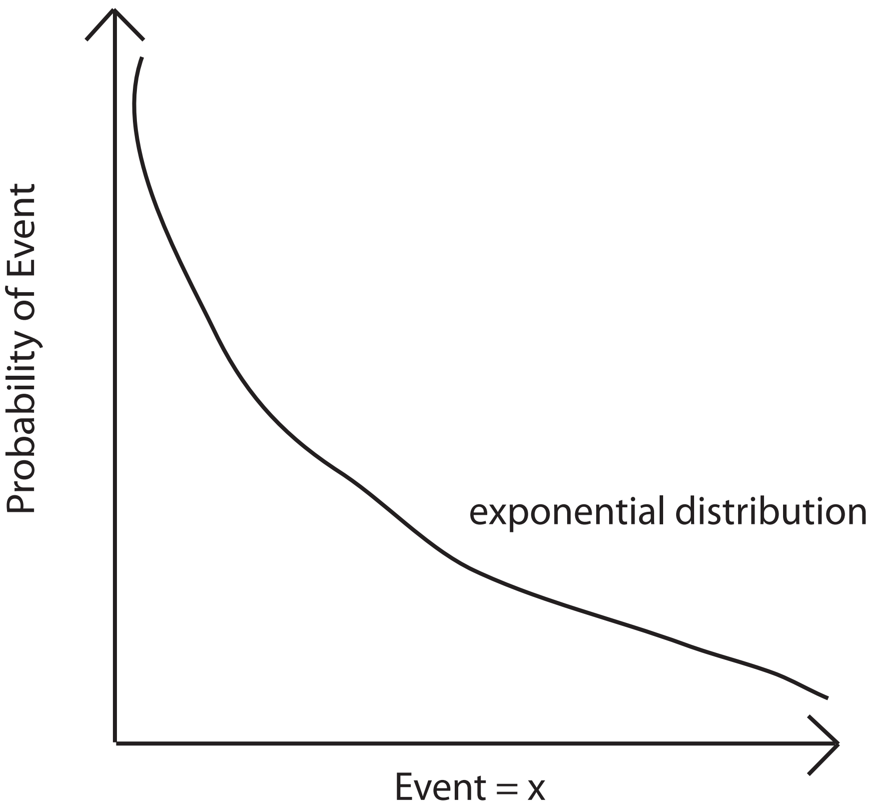
Queueing: Worst Case



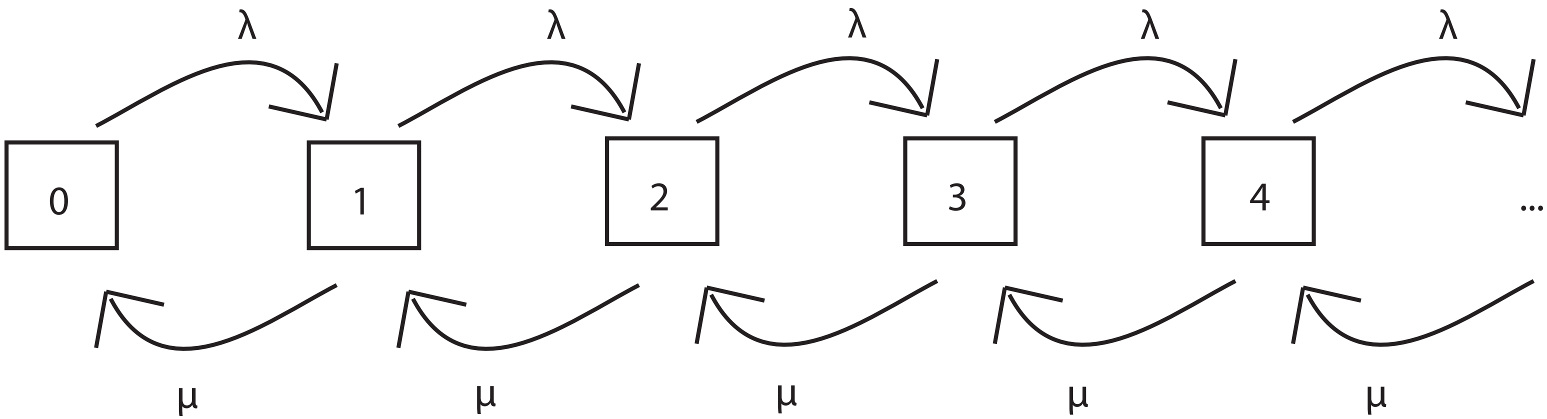


Queueing: Average Case?

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



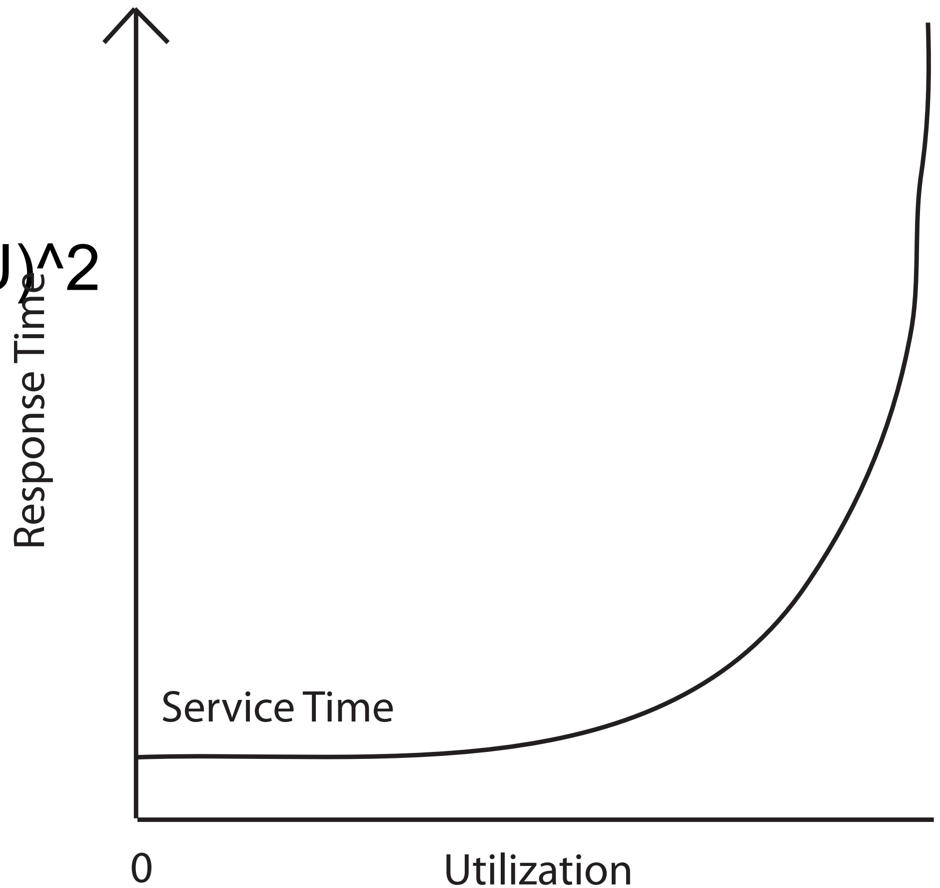
Exponential Distribution



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

Response Time vs. Utilization

- R = S/(1-U)
  - Better if gaussian
  - Worse if heavy-tailed
- Variance in R = S/(1-U)^2



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

Why Do Metro Buses Cluster?

- Suppose two Metro buses start 15 minutes apart
  - Why might they arrive at the same time?