CPU Scheduling and Queueing Theory

Last Time

- Scheduling policy: what to do next, when there are multiple threads ready to run
- Uniprocessor policies
  - FIFO, round robin, shortest job first
  - Multi-level feedback queues as approximation of shortest CPU task first

Main Points

- Multiprocessor scheduling
  - Affinity scheduling
  - Space vs. time sharing
- Queueing theory
  - Can you predict a system's response time?

Multiprocessor Scheduling

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

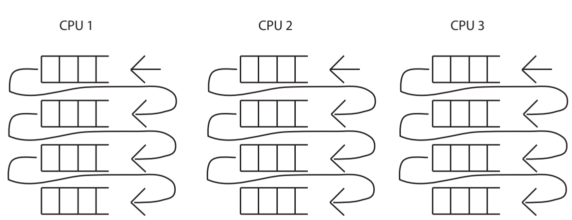
Multiprocessor Scheduling

- 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

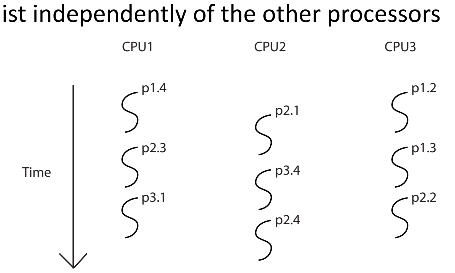
- Speedup on a multiprocessor limited by whatever runs sequentially
- Runtime >=
  Sequential portion + parallel portion/# CPUs
- Example:
  - Suppose scheduler lock used 0.1% of the time
  - Suppose scheduler lock is 50x slower because of cache effects
  - Runtime >= 5% + 95%/# CPUs
    - System is 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

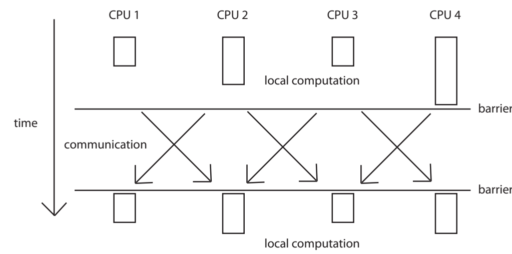


px.y = thread y in process x

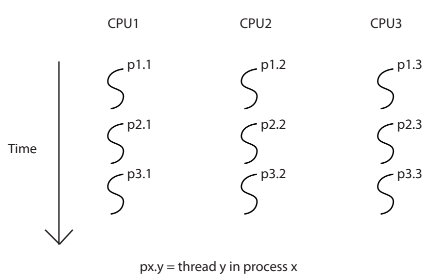
Scheduling Parallel Programs

- 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?

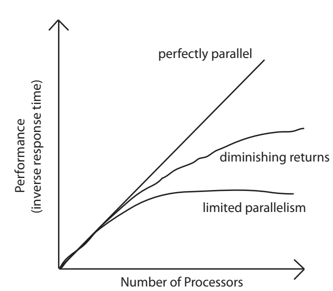
Bulk Synchronous Parallel Program



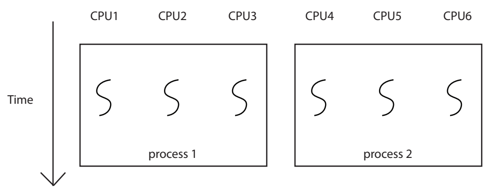
Co-Scheduling



Amdahl's Law, Revisited



Space Sharing

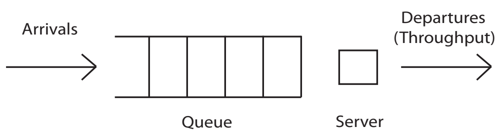


Scheduler activations: kernel informs user-level library as to # of processors assigned to that application, with upcalls every time the assignment changes

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



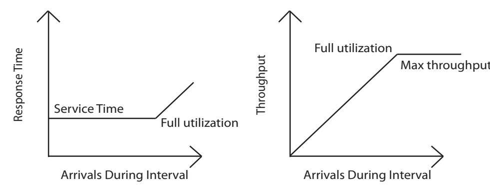
Definitions

- 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

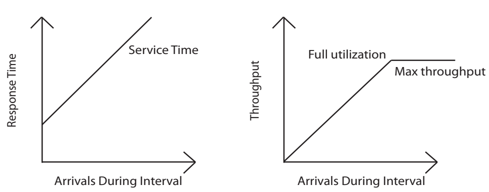
Queueing

- 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

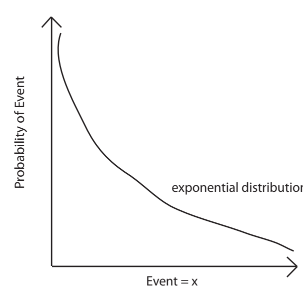


Queueing: Worst Case

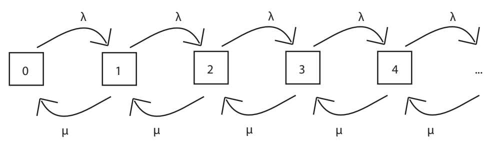


Queueing: Average Case?

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



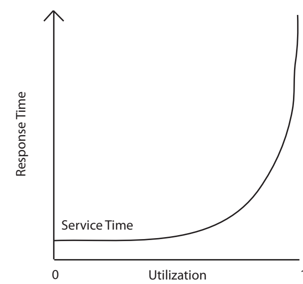
Exponential Distribution



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