Lecture 15
Performance and Scalability
SOME DEFINITIONS
Latency

• The time delay between the moment an action is initiated, and the moment the first of its effects occurs.

• Examples:
  – Time it takes a web browser to begin displaying the page.
  – Time it takes Orbitz to say it is initiating your search.
Service Time

• **Total Time required to service a request.** Defined as the elapsed time from when the request is initiated to the time when the full response has been produced.

• Examples:
  - Full time required to get the results from Orbitz.
  - On mobile devices, Google Maps used to show a low-res version and then load the full-res one. This improves latency at a slight cost to service time.
Capacity

• Total workload that can be handled without violating predetermined performance criteria.

• Example
  – Assuming all requests must be responded to within 5 ms, Orbitz has a capacity of 1000 requests/sec
Throughput

• Number of units of work that can be handled per unit of time.

• Example:
  – Number of Orbitz searches that are completed per second
Capacity / Throughput Relationship

• Throughput is requests / sec, regardless of any other performance criteria
  – e.g., time per request, CPU utilization, memory consumption, etc.

• Capacity is throughput under certain other restrictions
Bottleneck

• A point in an application is a bottleneck if adding resources at this point (while keeping other resources fixed) will increase performance
  – E.g., service time, throughput, or capacity

• Example
  – If the CPU usage on the Orbitz search server is never more than 50%, then CPU time is not a bottleneck for the search
Scalability

• The scalability of a system is its ability to improve throughput and capacity when given additional resources
  – E.g., CPU time, I/O bandwidth, network bandwidth
• If an application is perfectly scalable, throughput will double if resources are doubled (up to Amdahl’s law restrictions)
  – May need to increase work to do
Amdahl’s Law

- Speedup \leq \frac{1}{(F + \frac{1-F}{N})}
  - where Speedup = \frac{ST \text{ time}}{MT \text{ time}}

- Utilization = \frac{\text{Speedup}}{N}

\(N\) is the \# processors.
\(F\) is fraction of work that is serial.
Reducing Lock Contention

![Graph showing throughput comparison between ConcurrentLinkedQueue and synchronized LinkedList](image-url)
Reducing Lock Contention

• Reduce duration that locks are held
  – narrower synchronized blocks
• Reduce the frequency of lock requests
  – Use lock splitting or striping
• Replace exclusive locks with coordination mechanisms that permit greater concurrency
  – E.g., reader/writer locks
• Avoid hot fields
Lock splitting

• Replace a single lock in a program with two or more locks
  – Example: separate locks for protecting Nodes and Edges in ImperativeGraphs
  – Example: separate locks for protecting the upper and lower bound of MutableRange
Danger of more locks: Deadlock!

• DynamicOrderDeadlock.java
  - Shows how bank transfer between two different accounts, each with its own lock, can deadlock
    • Due to fixing the order that the locks are acquired
  - Fix: use dynamic information to determine ordering
    • .hashCode() method, plus a tie lock
Lock striping

• Like lock splitting, but based on a dynamically determined policy
  – ConcurrentHashMap uses 16 locks, each for 1/16 of the hashtable
  – Key idea: prior to splitting/striping, program contends more for the lock than the data it protects
Example: Striped Map

- StripedMapBad.java
  - Removes some benefit of striping because each method increments count using single lock
    - To ensure consistent semantics for size() and similar methods

- StripedMap.java
  - Removes problem by computing size per bucket
    - But no longer guaranteed atomic

- StripedMapBest.java
  - Computes counts incrementally, so size() faster
Reader/writer locks

• A pair of locks (R,W)
  – Multiple threads can acquire R
  – Only one thread can acquire W
  – No thread can acquire R if W is held
  – No thread can acquire W if R is held
• Readers acquire R, writers acquire W
  – Improves performance when there are many readers and few writers
• Java interface ReadWriteLock
  – Implemented by ReentrantReadWriteLock class
  – Policy: what to do when readers/writers waiting? What is the proper notion of fairness?
Avoiding hot fields

• When fields are accessed frequently from multiple threads, they are considered hot
• In this case, consider recomputing within separate threads, rather than contending for shared fields via locks
  – Extra per-thread computation cost scales according to Amdahl’s law, but serializing on a lock doesn’t
  – E.g., avoid “object pooling,” i.e., custom allocators: synchronization bottleneck