

# Amdahl's Law & The New Rules of Concurrency (or, What's Old Is New Again)

---

David Moskowitz  
Productivity Solutions, Inc.





# Agenda

---

- Amdahl's Law
  - Brief overview
- What's old is new again
  - Lessons learned from OS/2
- Safety
  - Rules and annotations
- Structure
  - Architecture and Design
- Implementation
  - Scalability, deadlocks and annotations





# Introduction

---

- Survey, can't cover all possibilities, contingencies, or testing issues...
  - Purpose: basis for discussion regarding a way to think about concurrency that crosses languages and platforms
  - Could spend 3 days and not cover everything
  - There are some Java-specific references
- We're talking about patterns to make code less brittle or breakable in a concurrency-based world
- Old world => choice, single- or multi-threaded...
  - ...in a multi-core world, it's less of an option



# Agenda

---

- **Amdahl's Law**
  - Brief overview
- What's old is new again
  - Lessons learned from OS/2
- Safety
  - Rules and annotations
- Structure
  - Architecture and Design
- Implementation
  - Scalability, deadlocks and annotations





# Ultimate Goal is Linear Scalability

---

- Perfect world: system throughput would be directly proportional to number of CPUs
  - Impossible for a variety of reasons
- Detractors from perfect linear scalability
  - Design issues, poor...
    - Transaction, connection management, schema
  - Implementation
    - Bad I/O strategies, different execution environment (compared to test environment), memory-intensive runtime memory management, inefficient memory usage (including leaks)
  - Sizing
    - Bad capacity planning
  - Operating system overhead



# Amdahl's Law

---

- The formula:

$$\frac{1}{\sum_{k=0}^n \left( \frac{P_k}{S_k} \right)}$$

- $P_k$  is a percentage of instructions that can be improved (or slowed)
- $S_k$  is a speed-up multiplier (where 1 is no speed-up and no slowing)
- $K$  is an enumerator for each different % speed-up
- $n$  is the number of different speed-up/slow-down resulting from change.



# What it means...

---

- Gross approximation:

- If 20% of the code can run in parallel, then...
  - $1/(1-P)$  or...
- $1/(1-.2) = 1.25$  theoretical maximum performance increase

$$\frac{1}{1 - P}$$

- More precise

- If 20% of the code can run in parallel... and that portion runs twice as fast
  - $1/((1-.2) + (.2/2)) = 1.11$  more likely gain

$$\frac{1}{(1 - P) + \frac{P}{S}}$$





# Bottom Line: It isn't linear

---

- CPU utilization tool.
  - If you're not using both processors, you have work to do
- As N gets large, (1-P)/S gets small, so...
  - 50% parallel means max 2x gain, independent of cores
- Measured on Mac (x86) & Solaris (SPARC) averaged
  - 1 processor (core) baseline
  - 2 processor 1.78 x 1 processor
  - 4p 1.79 x 2 processors (3.18 x 1)
  - 8p 1.51 x 4 processors (4.81x1)

$$P + \frac{1 - P}{S}$$

Numbers are similar with Linux; Windows Server is worse: 1.71, 1.63 (2.79), 1.40 (3.9) respectively



# A Note About Amdahl's Law

---

- Ultimately this covers the basics you need to deal with concurrency.
  - If you have tasks that can't be run in parallel throwing hardware at performance will have little impact.
- You don't need to understand the mathematics to appreciate the implications



# Agenda

---

- Amdahl's Law
  - Brief overview
- **What's old is new again**
  - Lessons learned from OS/2
- Safety
  - Rules and annotations
- Structure
  - Architecture and Design
- Implementation
  - Scalability, deadlocks and annotations

# What's old is new: Lessons from OS/2

---

- OS/2 was the first desktop operating system to support threads.
    - The primary purpose for threads, in an OS/2 context: provide extraordinary **responsiveness** to the end-user.
      - Not overlap, but parallel operation
    - Thread: unit of execution
    - Process: "contains" allocated resources
  - Most operating systems used a process model instead of threads.
- Fork a process in Linux





# Lessons from OS/2

---

- Threads had to be designed in, not added as an afterthought
  - An application was partitioned into threads based upon tasks, not arbitrarily or merely to overlap operations.
- Lesson 1: Think in terms of tasks
- Lesson 2: Start with a plan for concurrency
  - It's 10 times harder and costly to retrofit!
- Lesson 3: It's not just about parallel execution, there are side effects including race, deadlock, task synchronization, long-running, *etc.*



# Lessons: Task Partition

---

- The first step is to identify the tasks
  - Easiest starting point: Independent activities
  - Within a single task: often smaller tasks that can run concurrently
    - Lesson: Look for parallel possibilities
  - Communications
    - Resource synchronization (semaphores)
    - Data sharing (locking)
    - Inter-task communication (queues / messages)
    - *etc.*
- Required policy : synchronize with known, published identical semaphores (locks)



# Task Partition

---

- Server-side: Find the balance between throughput and responsiveness
- User-side: Extraordinary responsiveness
  - For example, avoid illogical UI blocking
- Every read has a write
  - (concept → pairs or sets)
  - What can be done in parallel?
  - Always look for parallel possibilities!!!
- New way of thinking about implementation
  - Not one application, but set of parallel tasks



# Learning From...

---

- Thread pools: same techniques still work
  - More expensive to create / destroy than to reuse.
  - Avoids the possibility of hitting a JVM limit
- Important to provide a mechanism to (self) terminate threads in the pool
  - Why?
  - Allows JVM to exit.
- Deadlock, semaphores, and annotation
- OO languages changes the rules
  - You don't command; you send a message.
  - Thread-safe means class manages itself



# Agenda

---

- Amdahl's Law
  - Brief overview
- What's old is new again
  - Lessons learned from OS/2
- **Safety**
  - Rules and annotations
- Structure
  - Architecture and Design
- Implementation
  - Scalability, deadlocks and annotations



# Rules of the Road: Optional???

---

- Failure to follow is fraught with fragility.
- Write thread-safe code
  - Use annotations (JSR 305)
  - Immutable is your friend
- Structure everything for concurrency
  - (JSR 166)
  - It's a multi-core world
  - Manage resources for concurrency
  - You never know when a "little" piece of code...
- Scalability is inexorably linked to concurrency
  - Serialization doesn't work



# Thread-Safe Code in OOP

---

- What is it? What does it mean?
  - Not just executable unit
- Class must be thread-safe
  - Internal class behavior is unknown outside class
  - No **external** coordination or synchronization
  - Class handles its own synchronization
  - **No** side effects when accessed from multiple threads
- In a multi-core world, concurrency isn't (can't be) an afterthought!



# Thread Safety

---

- Requires coordinated access to mutable data
  - Shared access requires special attention
  - `static` data should be `final`
- General rule of thumb: If it's really safe code in a single-threaded environment and it continues to be safe in a multi-threaded environment... then it **might** be thread-safe.
  - **Don't assume, test it!**



# JSR 305 Defect Annotations

---

- Includes contributed code from *Java Concurrency in Practice* (Brian Goetz *et al*)
  - <http://www.javaconcurrencyinpractice.com/annotations/doc/index.html>
  - @GuardedBy
  - @Immutable
  - @NotThreadSafe
  - @ThreadSafe
- Not final – there is still much discussion. For example, should these annotations be inherited?



# JSR 305 – General Thoughts

---

- Be consistent
- `@GuardedBy` **doesn't mean** `@ThreadSafe`
  - If something isn't explicitly annotated as safe, assume that it isn't!
  - **Safest:** Be explicit, use both `@ThreadSafe` & `@NotThreadSafe`
- `@ThreadSafe` **is class, not instance**
  - Observe the annotation
- If you inherit from a `@ThreadSafe` **class**
  - Observe the annotation



# Annotation Advantages

---

- Used consistently & correctly, it's easier to build thread safety starting with code that is already known to be safe.
  - How do you know?
  - Can you rely on the documentation?
    - Really?
- Annotations (properly used) allow tools to help identify errors.
  - Facilitates turning code over to someone else
  - Tricks (in the code) prevent turn-over



# Related Considerations

---

- Coding policies required
  - Access a resource that might be accessed from another thread requires that both threads synchronize with a common lock.
    - Hold/own semaphore X to access resource y.
    - You must stick to it!!!
- Has to be part of review process
  - Any exception causes grief
  - @GuardedBy helps



# Graceful Termination

---

- When can you cancel (or shutdown) a task?
  - You don't, you send it a message and...
  - What if you need to cancel a task?
- Plan first!
- Need a *cancellation policy* that defines:
  - **How** the mechanism works and how cancellation is requested,
  - **What** has to be done to initiate the cancellation request,
  - **What** the receiving object does in response, and
  - **When** the receiving object checks for the cancellation request.



Cooperative cancellation, what if...

# Thread Interrupt

---

- `java.lang.thread` provide methods to interrupt a thread (method == message in this context)
  - `void interrupt() // message, not command`
  - `boolean interrupted()`
  - `boolean isInterrupted()`
- Code needs to handle exceptions
  - `SecurityException`
  - `InterruptedException`
  - `ClosedByInterruptException`
- Probably best way to terminate a thread that also does not reinvent the wheel



# Thread Interrupt

---

- Beyond the scope of this presentation to delve into how thread interrupt exceptions should be handled.
  - There's more that we haven't covered, for example: need a way to handle non-interruptible blocking methods
  - See the resources section for more information.
- It has to be part of policy
  - One reason: need a way to gracefully handle JVM shutdown



# Agenda

---

- Amdahl's Law
  - Brief overview
- What's old is new again
  - Lessons learned from OS/2
- Safety
  - Rules and annotations
- **Structure**
  - Architecture and Design
- Implementation
  - Scalability, deadlocks and annotations





# Structure

---

- 2 points: Tasks & Amdahl
- How do you identify parallel tasks
  - Decompose the task. What is really involved?
    - Every `read` has a corresponding `write` (pairs)
      - ✓ File, buffer, screen, cache...
    - What type of communication is required to keep the two in sync?
- Amdahl's law suggests the benefit attached to parallel processing.
  - Difference between CPU and threads



# Structure for Success

---

- Encapsulation is your friend!
  - Make everything that does not have to be externally accessible `private`
    - Are you sure...???
- Each class must manage its mutable data
  - No outside access permitted unless the access is available only with possession of a defined and unique lock...
    - Not a guarantee of thread safety!
  - **Consider scope of control & scope of effect**



# Best Practice

---

- Plan... resource management
  - Programmer controlled,
    - Not request arrival or request volume / frequency driven
  - Put bounds on every resource used
  - Need policy for resource depletion
    - For example: queue fills
  - Measure, monitor, adjust and control
- Like threads, it's much more costly and time consuming to retrofit than to accommodate



# Serialization

---

- Simplistic response: remove it
  - Think about parallelization instead
- Throughput is governed by Amdahl's Law
  - Caps maximum derived performance based upon what can be run in parallel
  - If 50% must be serialized, you only get 2x maximum performance benefit no matter how many cores are available or added.
  - Remember Brooks (Mythical Man Month)



# Agenda

---

- Amdahl's Law
  - Brief overview
- What's old is new again
  - Lessons learned from OS/2
- Safety
  - Rules and annotations
- Structure
  - Architecture and Design
- **Implementation**
  - Scalability, deadlocks, and testing



# Implement Immutability

---

- Immutable objects are thread safe
  - `@Immutable` is `@ThreadSafe`
  - What is an immutable object?
- Most problems occur coordinating access to mutable data or state
- How often do you use `final`?
  - Something to think about: Can you use it more often?
- If you can't make the class immutable, can you limit mutability?



# Resource Management

---

- Two related pieces...
- Limitation on resource usage
  - If you don't limit resource usage, the JVM will
  - Threads, queues, connections (database, outgoing, incoming), file handles, ...
- Concurrency issues
  - `private` vs `public`, don't assume, be explicit
    - `final` when you can
  - Out-of-date data might not be a problem in a single-threaded environment, but...

# Thread Pools

---

- Different in a multi-core (CPU) environment.
- How many threads?

- Depends upon the number of cores

```
numCores =
```

```
Runtime.getRuntime().availableProcessors();
```

- Starting point for thread pool size: `numCores + 1`

- Explicit threads, not implicit

- Modifiers

- Scarce resources (*e.g.*, JDBC connections)
- Mix of multiple types of operations: long running, I/O, blocking, ...

- Based upon empirical testing



# Avoid Pools...

---

- If there is a possibility that an object from the pool will immediately experience resource contention block.
  - Contention is expensive *vs.* allocation
  
- Plan for concurrency!
  - Failure to plan **IS** planning to fail!



# Scalability

---

- Scalability is not the same as performance.
  - Scalability refers to the ability to take advantage of additional resources (CPU, memory, bandwidth).
  - Does the data scale?
  - Performance refers to throughput and speed
  - Can operations change thread count?
- Performance is related to code optimization
  - Outside of optimizations performed by compiler
  - Consider Knuth's warnings: "If you optimize everything, you will always be unhappy."



# Performance

---

- SOP: Get it working, then refactor and optimize to improve performance
- Refactoring also considers tradeoffs
- Ask questions during the process
  - Does this optimization scale? Prevent scaling?
  - How much faster? Is that really important?
  - Optimize a situation: how often does it occur?
  - Are there side effects to the optimization?
  - Are there hidden costs (*e.g.*, increased costs for maintenance or deployment or...)



# Hidden Challenges

---

- If you don't have access to the source code you have to infer from observation
  - Use a profiler that shows CPU/core use and threads
- Add threads and see what happens to throughput
  - Easier on a multi-core system
  - Start with one thread and add until  
`numThreads == numCores`
- Does the throughput increase as the number of threads approaches the number of cores?
  - Yes, the code scales; no the code doesn't scale



# Testing Correctness

---

- Concurrency introduces complexity
  - Still use JUnit for conformance (correctness)
  - Can't test concurrency without multiple threads
    - Better with multiple cores
- How do you test to determine if a thread blocks when it is supposed to do so?
  - Wait and interrupt (`InterruptedException`)
  - Modify wait period and try again...



# Testing Thread Safe

---

- More challenging than testing for correctness
- You must **predict** what is likely to fail when something breaks the notion of Thread Safe
  - Things get more interesting if the test requires synchronization
    - Hint: don't do that
- Testing thread safety requires multiple threads – test threads can be harder to create than the classes under test.



# Thread Safety

---

- Testing race conditions or deadlock is easier with more threads.
  - Test with same number of cores as the production system
  - Test with as many threads as planned in the production environment...
    - ...then add more to be sure
- **Reminder: predicting what is likely to fail is difficult, but necessary.**
  - ... so is adjustment for (at) each test iteration
  - Test to establish bounds when ops tunes





## Testing / Measuring Performance

---

- Generate expected metrics for performance, CPU utilization, serialization, etc.
- Profile to observe
  - Make sure the profiler supports threads
    - If you find one that also incorporates core profiling...
- Monitor CPU utilization and compare to profile output.
- Measure and monitor responsiveness
  - That still is one of the purposes for concurrency
- Find and eliminate dead code



# Additional Thoughts

---

- Pay attention to both the hardware & virtualization spaces
- There is more support for concurrency coming in both spaces
  - *e.g.*, Hardware supported virtualization
- Be prepared to accommodate hardware changes at the atomic instruction level
- Think about what it would take to create non-blocking algorithms where- and when-ever possible



# Not the Last Word

---

- John Soyring talked about it, and it's another driver for concurrency think.
  - More computing power → more cores at slower clock speed, not faster processors
- Green is another driver
  - Need more computing power with a reduced carbon footprint
- Either way it means concurrency is no longer an option!



# Resources

---

- *Java Concurrency in Practice* by Brian Goetz et al, Addison-Wesley, © 2006 Pearson Education, ISBN: 0-321-34960-1
- *Mythical Man Month Essays on Software Engineering, 20th Anniversary Ed* by Frederick Brooks, Addison-Wesley Professional, © 1995 by Frederick Brooks, ISBN: 0-201-83595-9
- <http://jcp.org/en/jsr/detail?id=305>
- <http://groups.google.com/group/jsr-305>
- <http://groups.google.com/group/jsr-305/web/proposed-annotations>
- <http://jcp.org/en/jsr/detail?id=166>
- Align business and IT goals and objectives see the BoF slides "What is ITIL" on CSS 2007 CD



# Questions ? ? ?

---

**If you  
don't ask,  
who will?**

**If not now,  
when?**



**There  
aren't any  
dumb  
questions.**

**The only dumb  
question is the  
one not asked!**





# Thank You

---

For more information:

David Moskowitz

Productivity Solutions, Inc.

147 Ashland Avenue

Bala Cynwyd, PA 19004

+1-610-726-9925

[davidm2@usa.net](mailto:davidm2@usa.net)

SkypeID: davidmosk

