Grand Central Dispatch
and
NSOperation

CSCI 5828: Foundations of Software Engineering
Lecture 29 — 04/26/2012

Credit Where Credit Is Due

- Most of the examples in this lecture were inspired by example code found in
  - Advanced Mac OS X Programming: The Big Nerd Ranch Guide
    - by Mark Dalrymple
  - iOS Programming: The Big Nerd Ranch Guide, 2nd Edition
    - by Joe Conway & Aaron Hillegass
- Both books come highly recommended, as do all Big Nerd Ranch Guides!

Goals

- Cover the main concepts of Grand Central Dispatch, a concurrency framework, first deployed by Apple in OS X 10.6 and iOS 4.
    - Queues and Tasks
    - Supporting Concept
        - Blocks, an extension to C, used heavily by Grand Central Dispatch
- Also cover NSOperation, a higher-level API that allows OS X and iOS applications to perform tasks concurrently
    - This higher-level API existed before Grand Central Dispatch
        - but has since been re-implemented to use GCD underneath

Design Approaches for Concurrent Systems

- This semester we looked at three approaches to the design of concurrent systems
    - Shared Mutability
        - Multiple threads can access mutable variables; need synchronization
    - Isolated Mutability
        - Multiple threads; but only one thread can access a mutable variable at a time; no locking; STM + Actor Model
- Pure Immutability
    - No mutable variables; values can be shared freely across multiple threads

And the Winner is...

- After looking at each approach, it's clear that
  - Shared mutability is really hard
    - starvation, race conditions, and deadlock, oh my!
  - Pure immutability is also hard
    - but primarily because it is unfamiliar;
    - Clojure/Scala are making headway here
- **Isolated mutability is the way to go!**
  - java.util.concurrent helps to create isolated mutability solutions
  - STM and the Actor Model encourage isolated mutability as well

Grand Central Dispatch: Up With Isolated Mutability

- Grand Central Dispatch is
  - an approach to creating concurrent software systems
  - it encourages developers to adopt the isolated mutability approach
- While initially deployed only on OS X 10.6 and iOS 4, the source code for libdispatch (the library that implements GCD) was released under the Apache License in September 2009 and has since been ported to
  - OS X 10.7 and iOS 5 (by Apple)
  - FreeBSD, Linux, Solaris (by the Open Source community)
- A Windows port is underway (according to Wikipedia)

Tasks and Work Items

- The first key abstraction in GCD is the **task**
  - A task is a discrete goal that your application wants to accomplish
    - display a web page; analyze a set of tweets; calculate primes
  - Tasks can depend on other tasks and might be decomposed into sub-tasks
    - Tasks themselves are ultimately decomposed into **work items**
      - work items are chunks of code (known as **blocks**) that need to be executed to make progress on a task
        - read a file; parse a tweet; break down a range of numbers
- Work items get **placed on queues**; when they get to the front of a queue, they are assigned to a thread and get executed

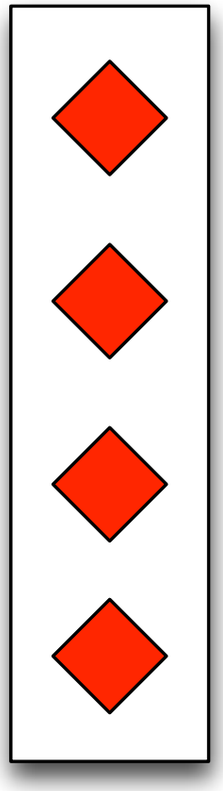
Queues

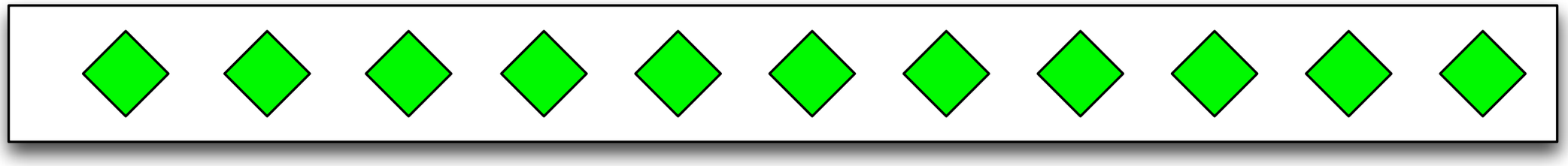
- The second key abstraction for GCD is the **queue**
- Queues can be either
    - a **serial queue**, created for a specific application
- or
    - a **global queue**, created by the operating system
        - three global queues of different priorities: low, default, and high
- Serial queues are lightweight constructs; you can have as many as you need
    - Each serial queue must “target” a global queue to get work done
        - Serial queues can also target other serial queues but ultimately they have to point at a global queue

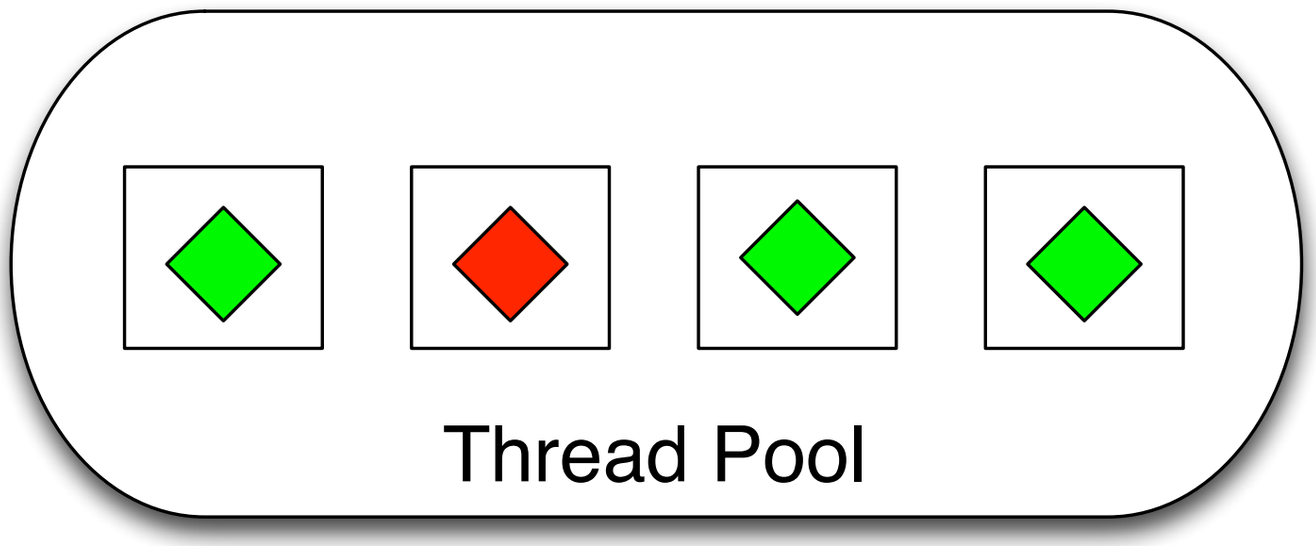
Semantics of Queues

- Both types of queues have FIFO semantics
    - The difference lies in the semantics of work completion
- Global queues
    - dispatch (**assign work items to threads**) in FIFO order
- Serial queues
    - dispatch **and complete** work items in FIFO order
- What does this mean?
    - A serial queue executes **only one** work item at a time
    - A global queue executes **many** work items at a time

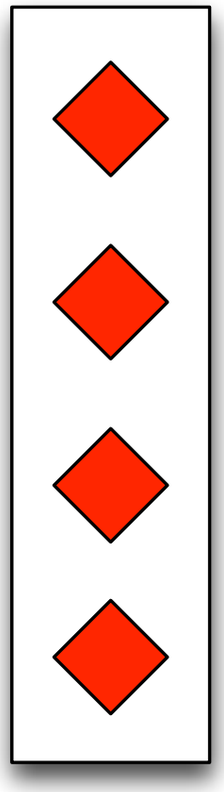
In this simplified view, GCD is managing one serial queue (for application A) and one global queue. The serial queue is targeting the global queue. Work items from the serial queue are shown in red. Work items from **other sources** are shown in green.



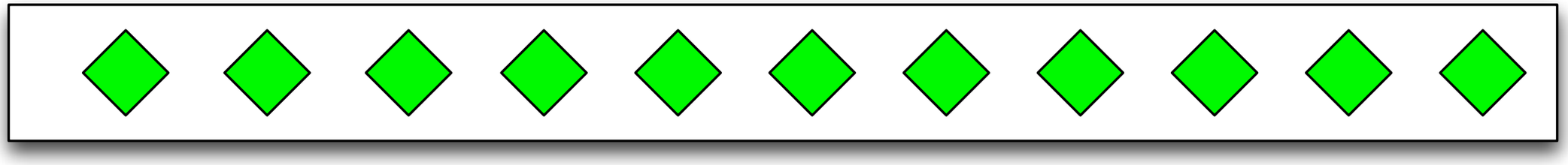




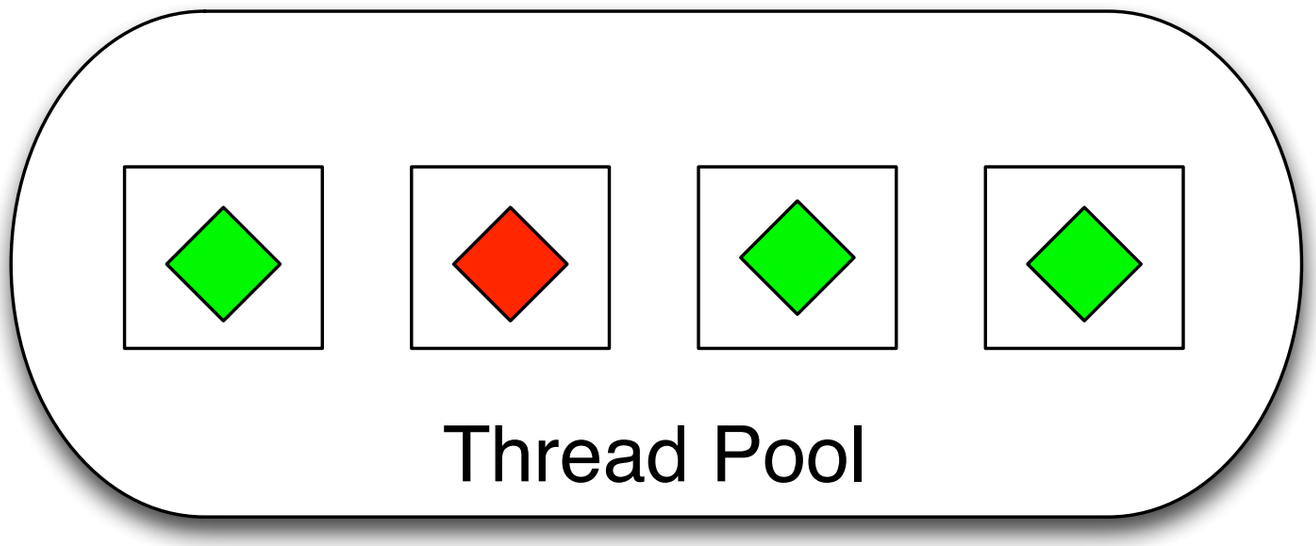
What “other sources” exist? Application A may have **other serial queues** targeting the global queue, or it might have **background threads** adding work items directly to the global queue. **Other applications** may also be adding work items to the global queue. Finally, the **OS** itself may add work items to the global queue.



Serial Queue

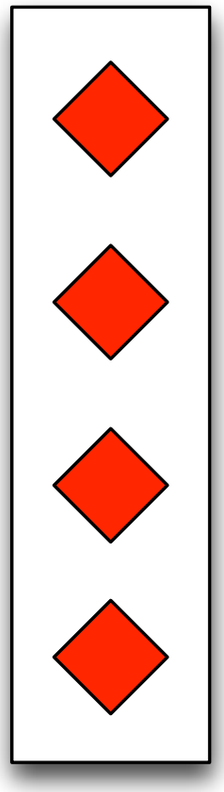


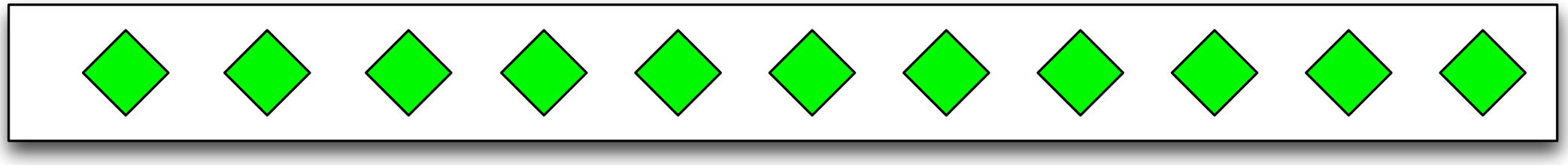
Global Queue

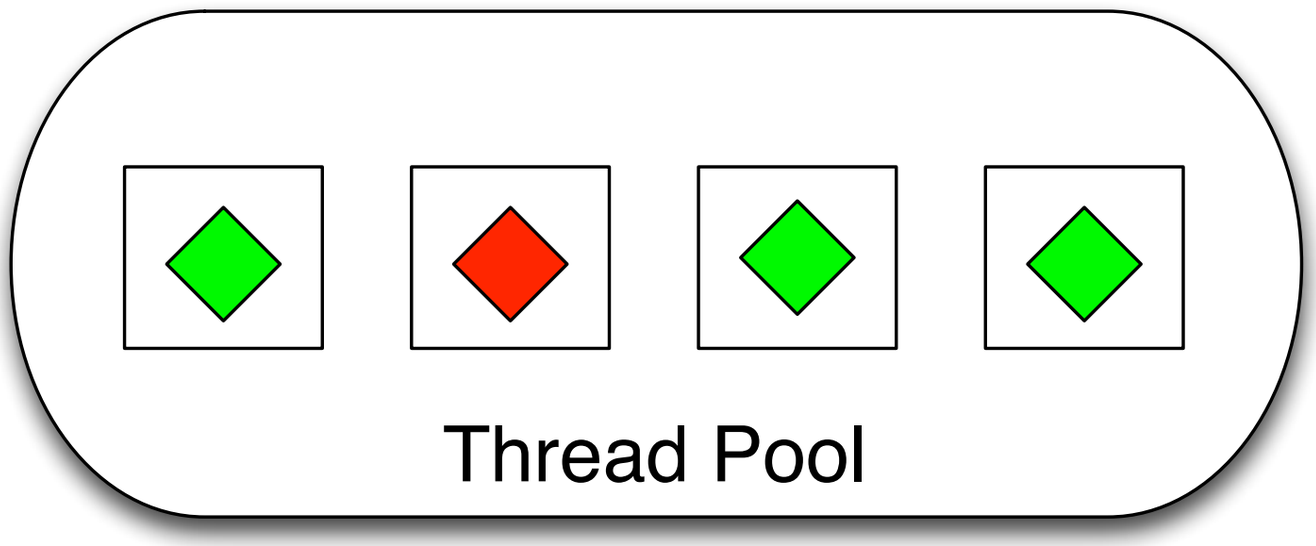


Thread Pool

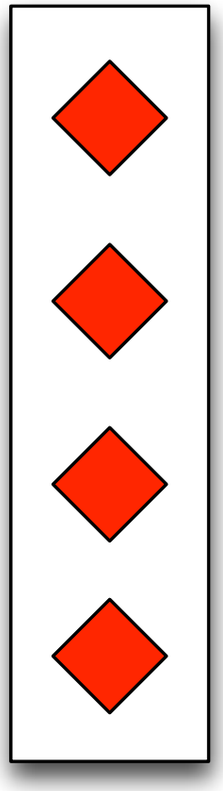
There are no red work items in the global queue because a serial queue **dispatches AND completes** work items in FIFO order. Since a red item is currently running, no additional work items in the serial queue can be added to the global queue.



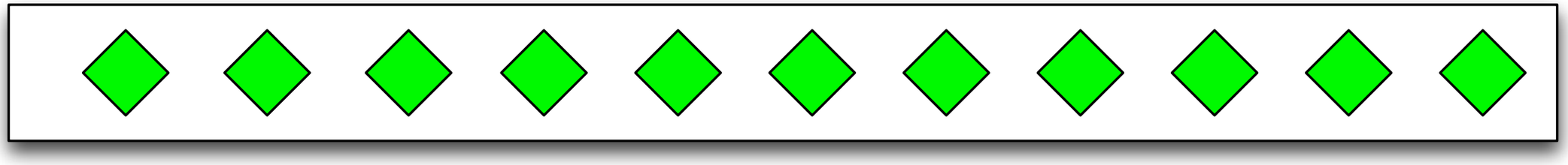




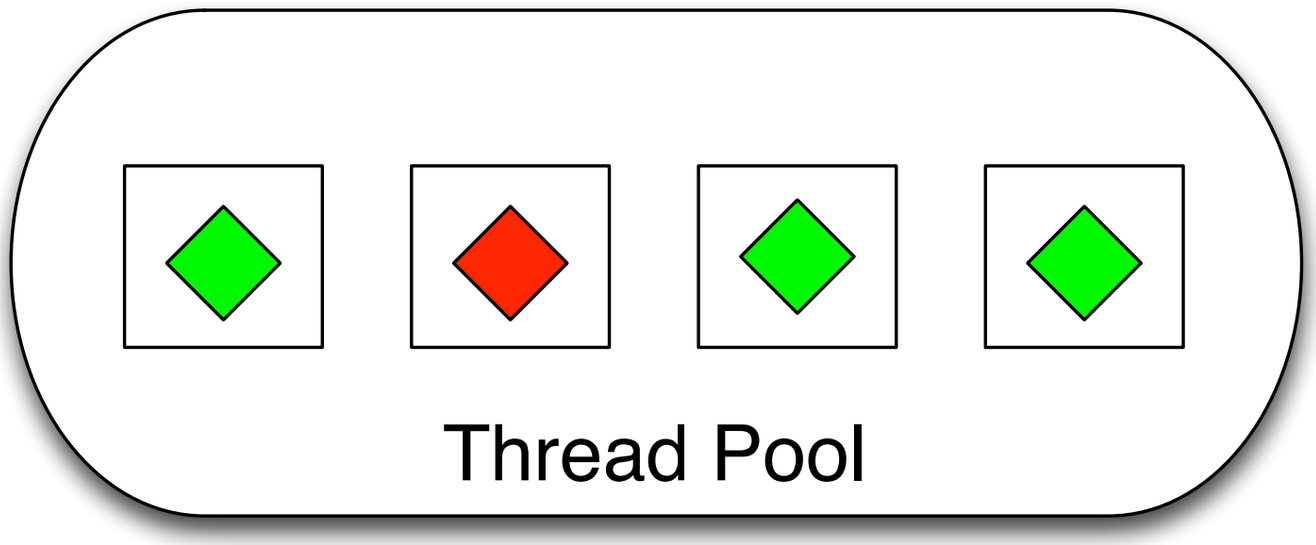
No such constraints exist for the green work items, they can be **dispatched immediately** as threads become available. The global queue does not care about work item completion, it only guarantees that work items are assigned to threads in FIFO order.



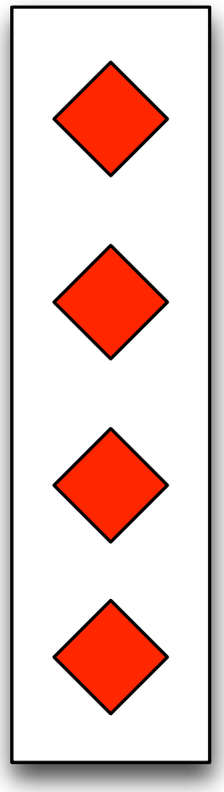
Serial Queue



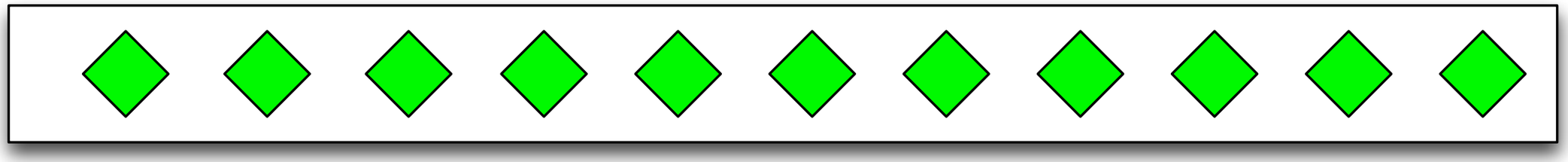
Global Queue



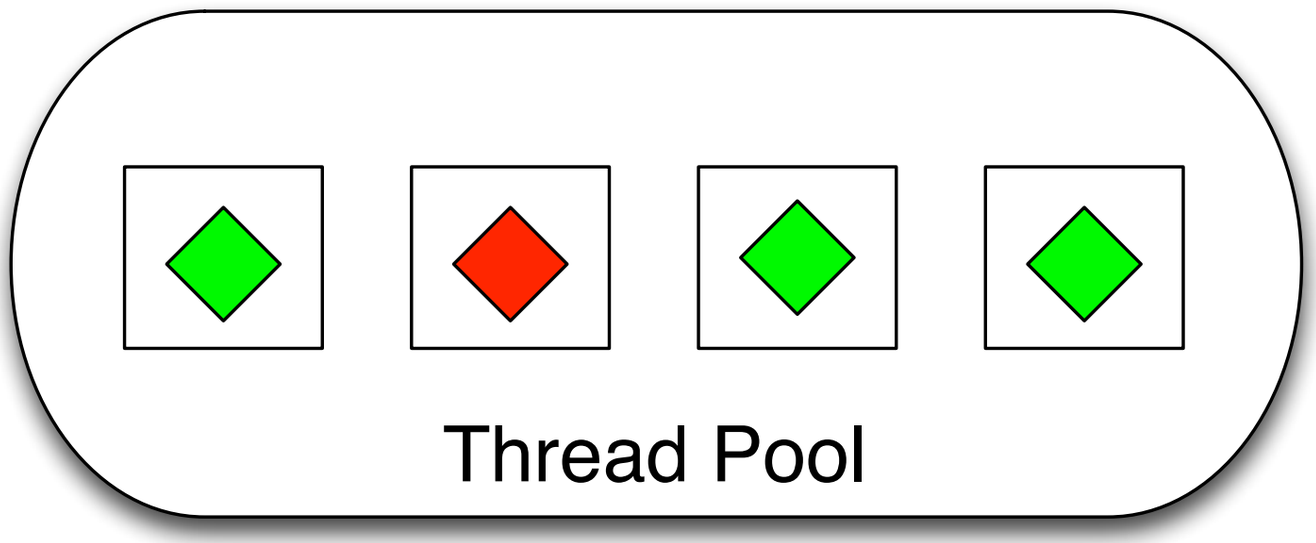
NOTE: GCD automatically determines how many threads it should allocate based on system load, number of cores, etc. It takes a global view of the workload on the **entire machine** and allocates threads accordingly.



Serial Queue

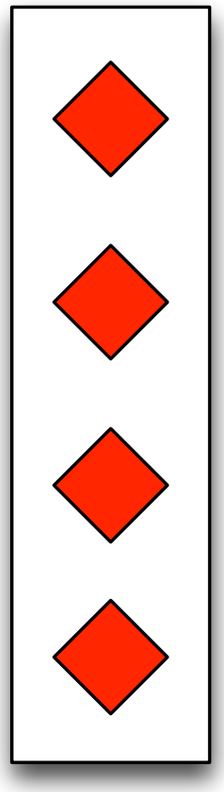


Global Queue

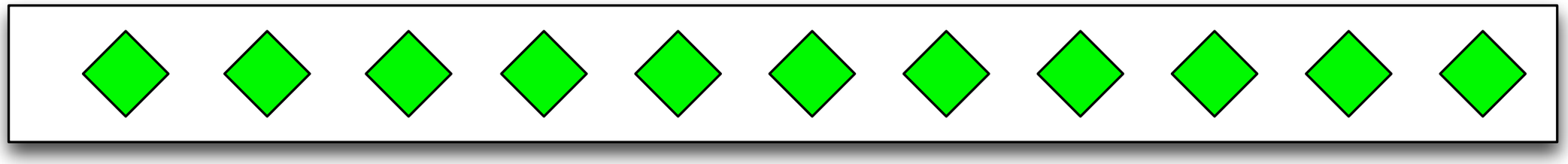


Thread Pool

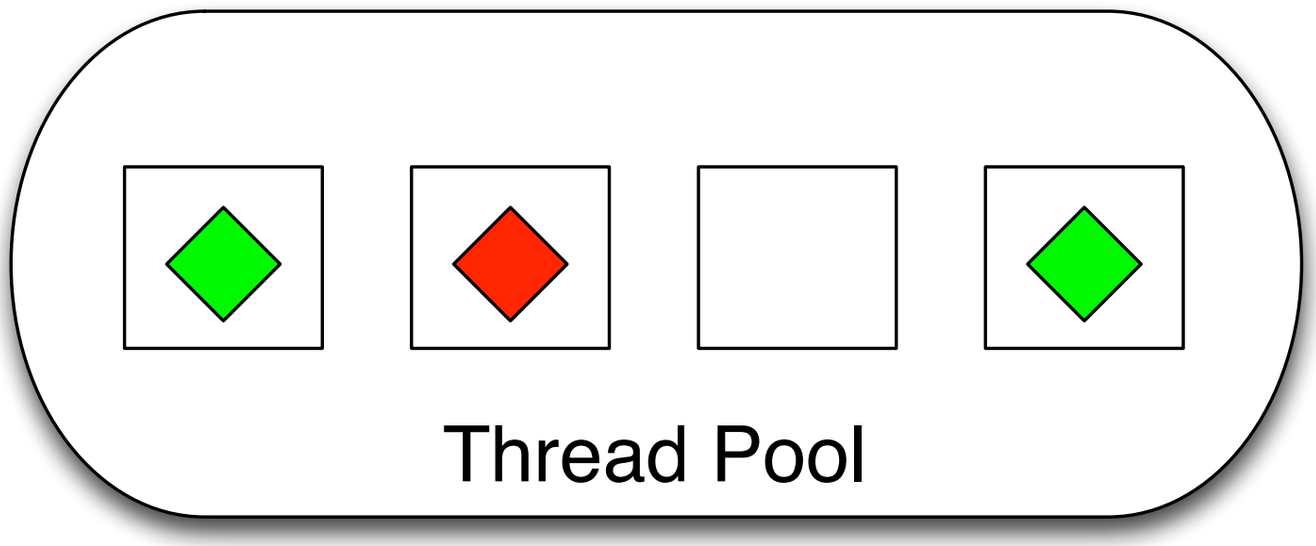
When a green work item completes...



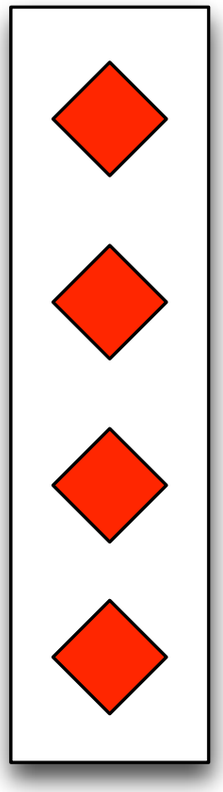
Serial Queue

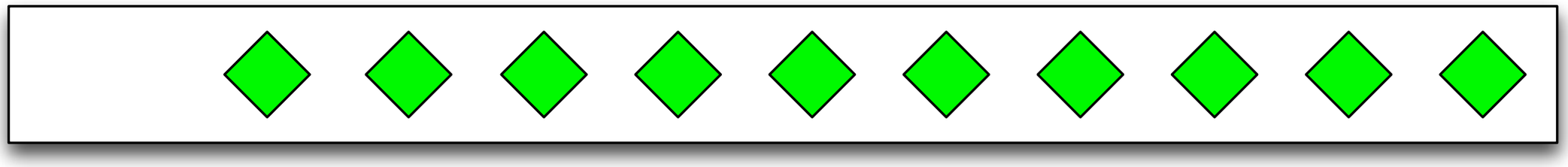


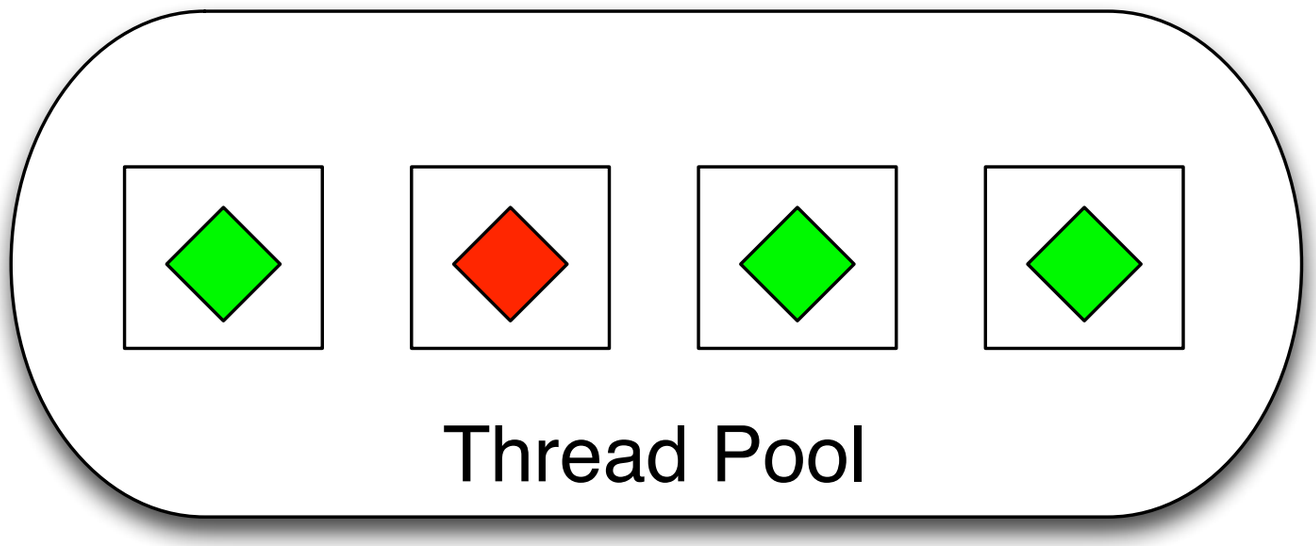
Global Queue



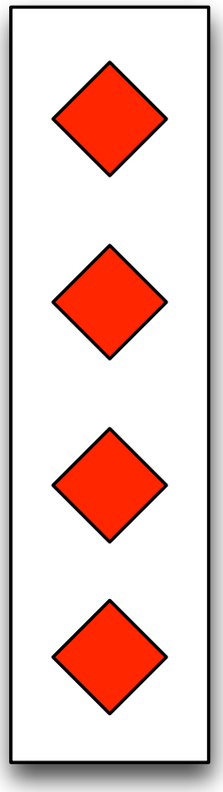
When a green work item completes, the global queue will immediately dispatch another item from the queue.



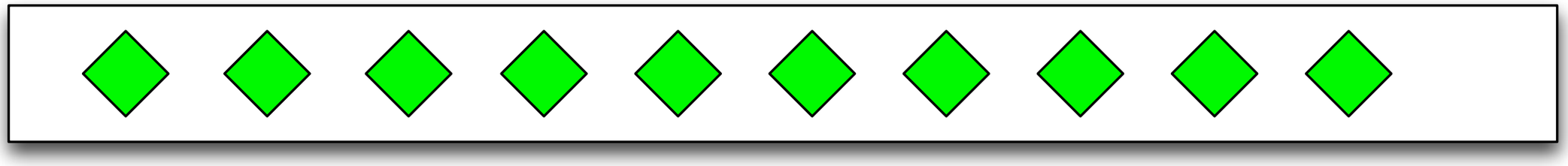




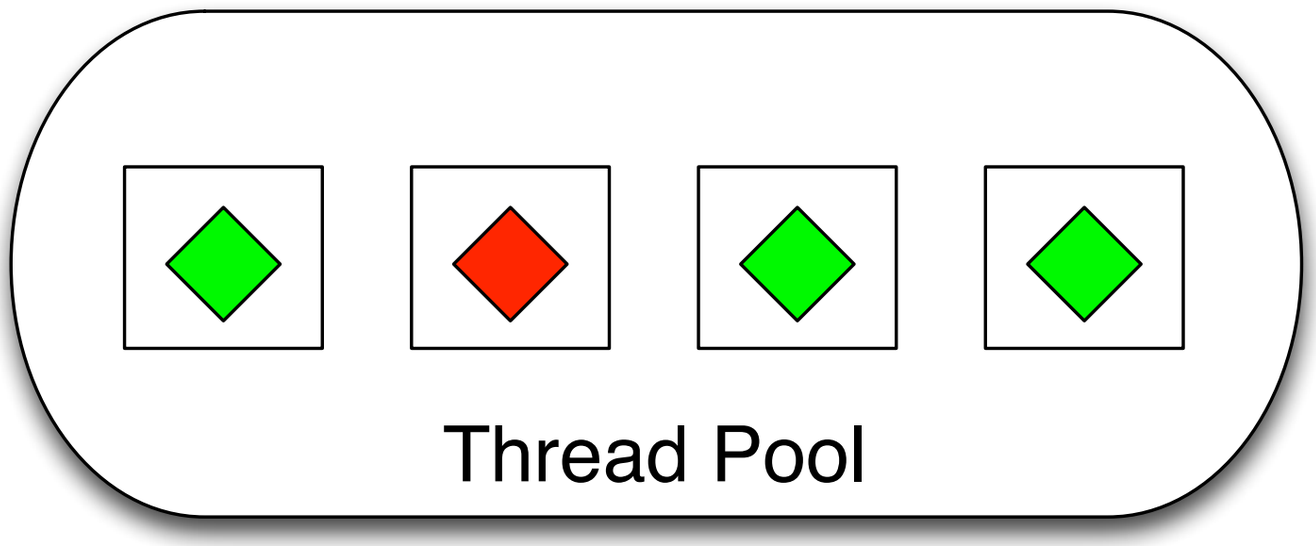
The queue will update...



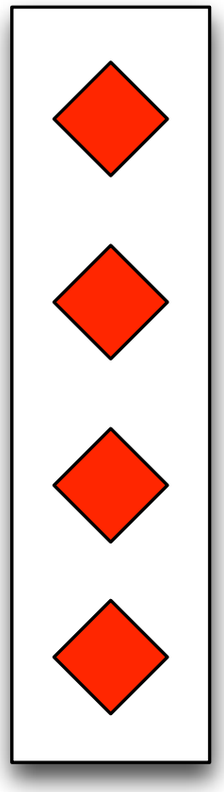
Serial Queue



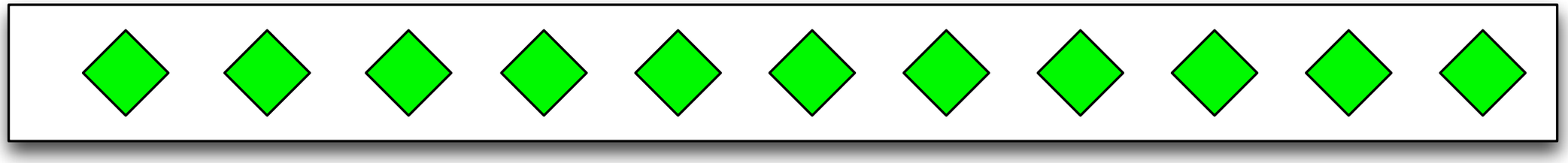
Global Queue



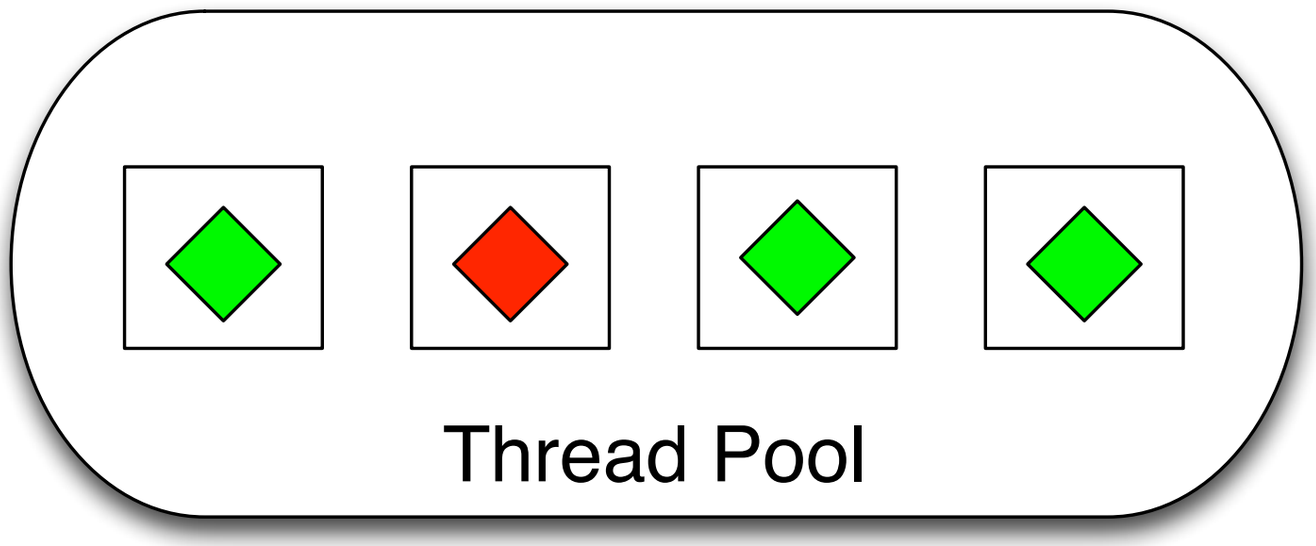
... and another green work item can be added.



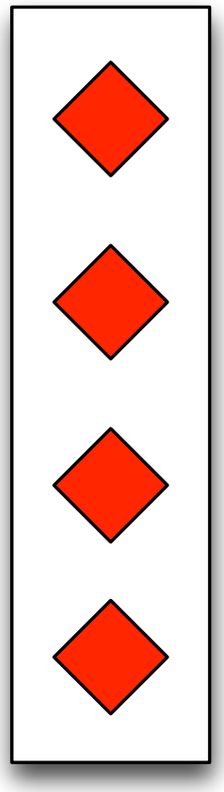
Serial Queue



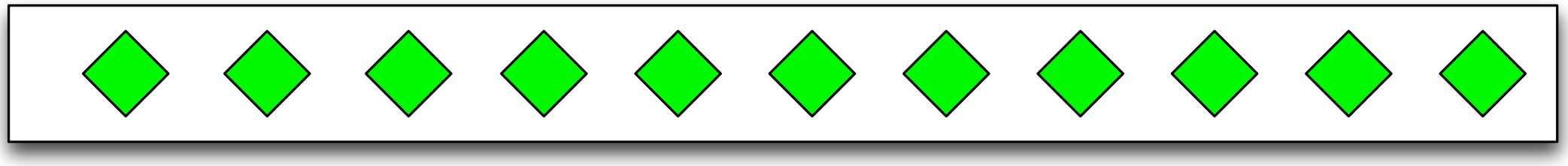
Global Queue



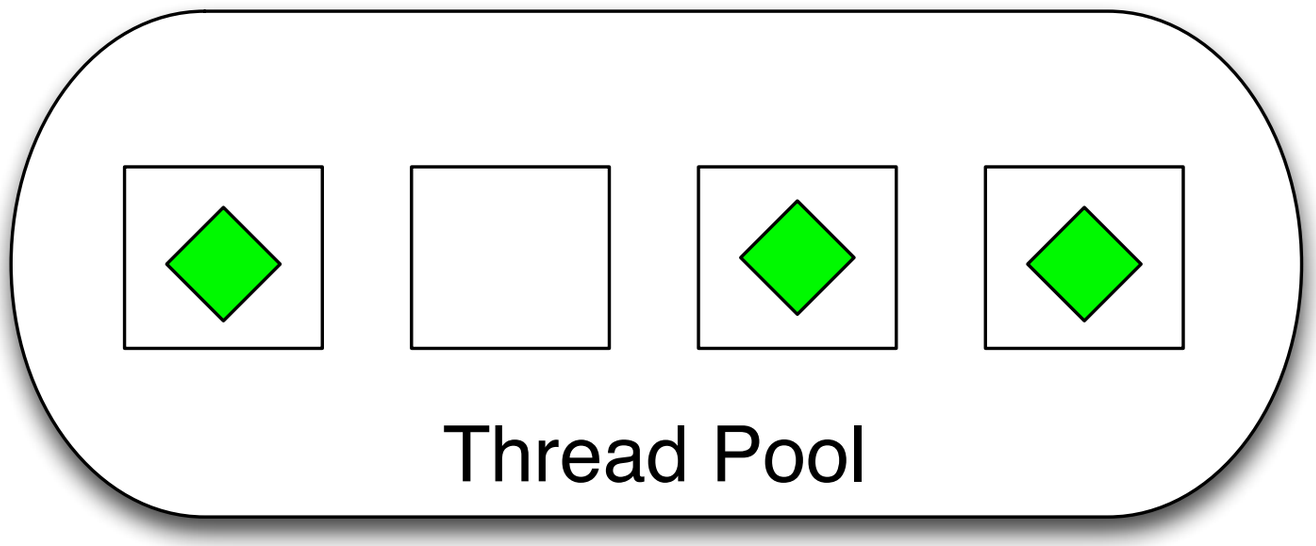
When a red work item completes...



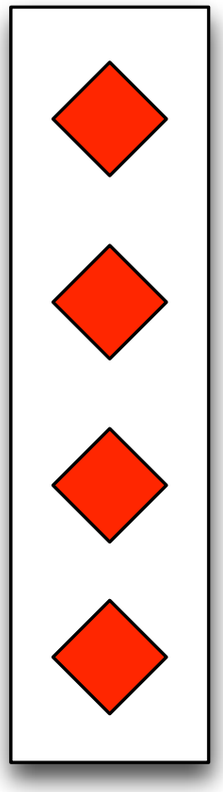
Serial Queue



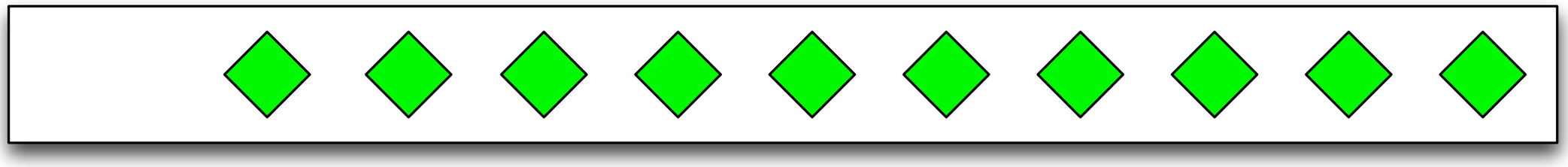
Global Queue



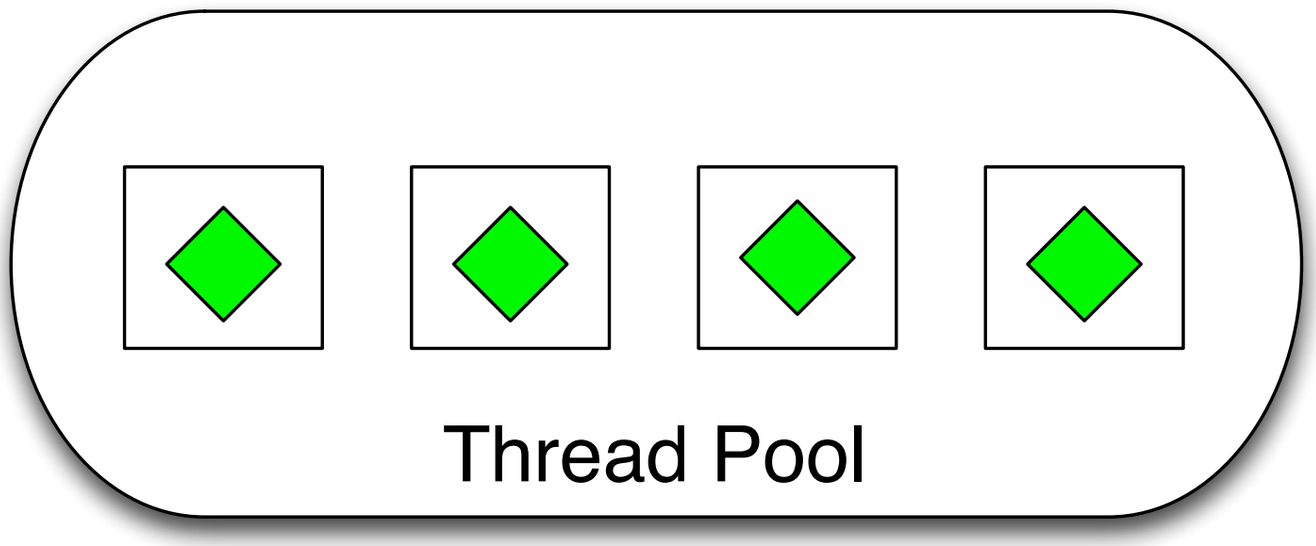
When a red work item completes, the global queue will immediately dispatch the next item from the queue.



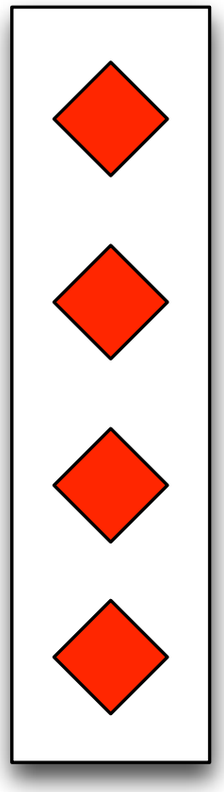
Serial Queue



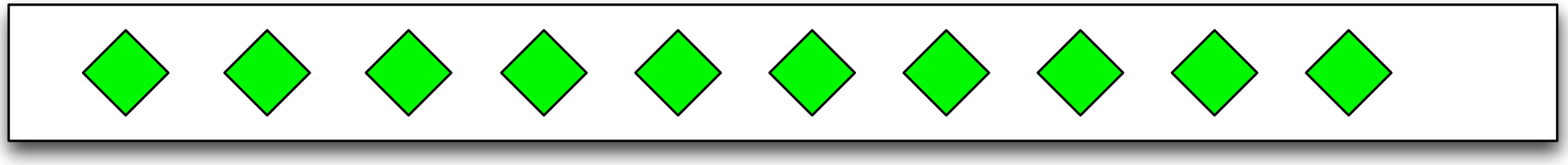
Global Queue



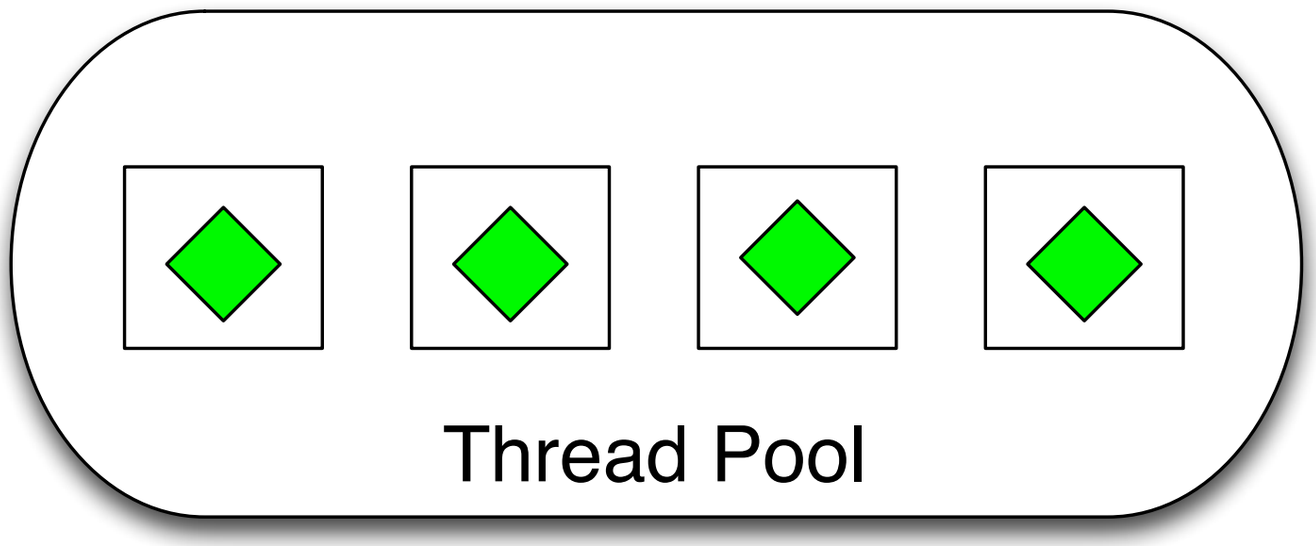
Once again, the queue will update...



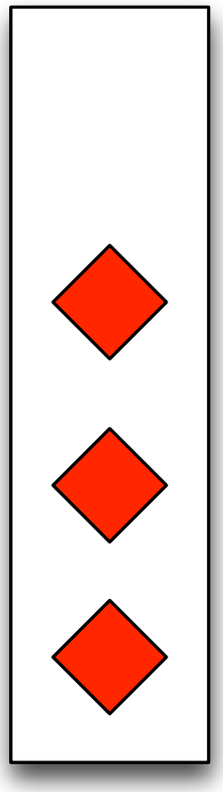
Serial Queue

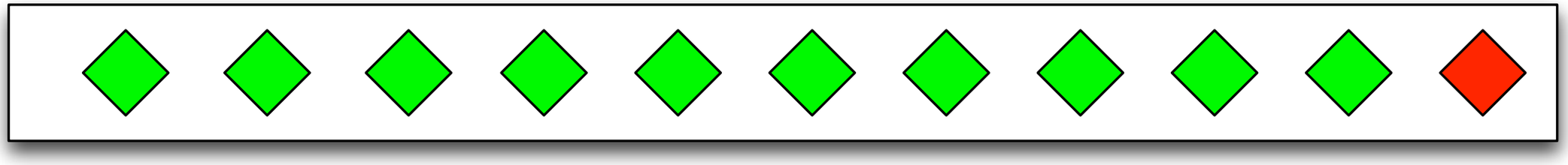


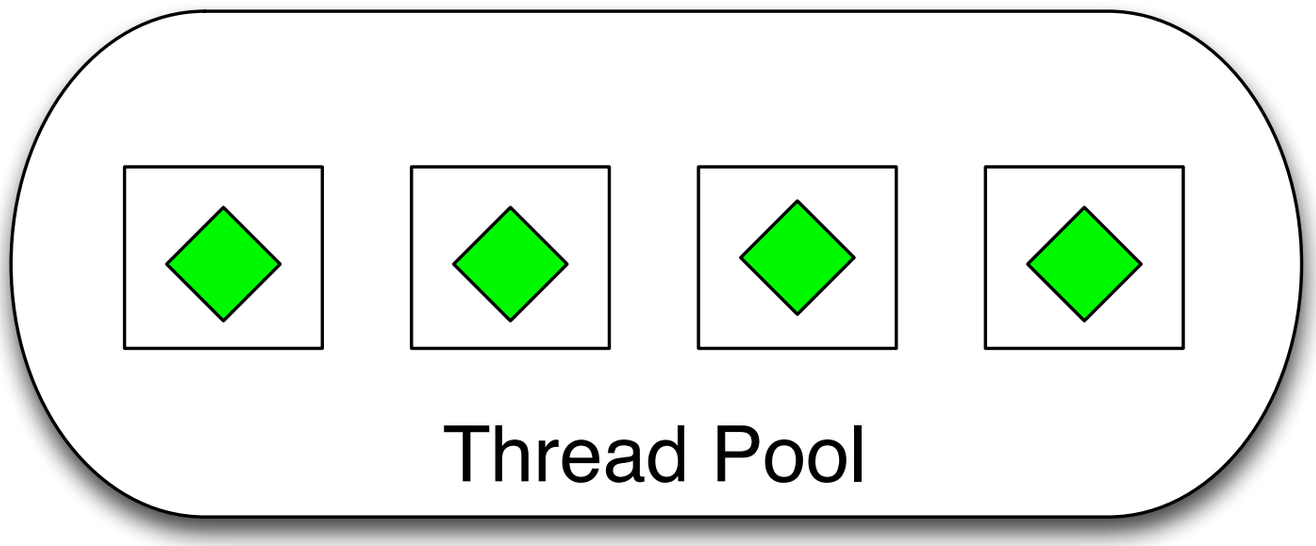
Global Queue



... but this time, the serial queue can add its work item to the global queue. That work item will then wait its turn to be dispatched to an available thread.







Of course, it's not this simple

- An actual implementation of GCD is more complex
  - Tens to hundreds of serial queues and thousands of work items
  - Lots of threads based on cores and overall load on system
  - Three global queues with different priorities
- Plus
  - tasks can be added to a global queue with a time constraint
  - a set of tasks can be added to a group; an app can then wait for the entire group to complete
  - GCD can monitor kernel-level events (signals, file reads, etc.) and then add a predefined block to one of its queues to handle the event

GCD is powerful (I)

- When you start an application in OS X or iOS,
  - you do some initialization
  - and then you start the main event loop
    - everything you do in your application is located in an event handler that is called by the main event loop when a particular event occurs
- With the advent of GCD, all “main event loops” were reimplemented to be serial queues!
  - Since GCD monitors the work load of the entire machine, **all of the events** being handled by **all of your running applications** are automatically taken into account
    - and will influence the number of threads allocated by GCD

GCD is powerful (II)

- Contrast this approach with Java's ExecutorService
    - An ExecutorService is an application-level construct
- When we did thread-allocation calculations in our Java programs earlier this semester, we did them within the context of a single application
    - Those programs have NO IDEA what else is running on the operating system
    - So, our IO intensive application is going to allocate HUNDREDS OF THREADS, even if four other IO-intensive applications are running at the same time
- With GCD, you get the benefits of not having to worry about thread allocation and guarantees that GCD will be allocating the number of threads it needs to efficiently handle all the current set of work items

GCD is powerful (III)

- A high level overview of GCD is available here:
  - <http://developer.apple.com/library/mac/#featuredarticles/BlocksGCD/_index.html>
- Take a look at the (short) subsection entitled “Thread Pools” for additional details
- Like an ExecutorService, it is best if your work items
  - do NOT themselves become blocked waiting for IO or other work items
- To avoid this, GCD provides ways to monitor work item completion and (as previously mentioned) to handle IO via a dispatch_source that generates work items automatically when their are bytes to read and/or write

GCD API (I)

- The primary API of GCD centers around queues; NOTE: this is a C API
- You can get hold of your application's main queue (a serial queue) like this
  - `dispatch_queue_t dispatch_get_main_queue (void);`
- You can get hold of one of the three global queues via:
  - `dispatch_queue_t dispatch_get_global_queue (priority, flags);`
- You can create your own queue with this function:
  - `dispatch_queue_t dispatch_queue_create (label, attr);`
- Note: I'm hiding parameter types for clarity
  - label is a reverse DNS string: "edu.colorado.cs.MyClient"

GCD API (II)

- To add a block to a queue for asynchronous execution:
  - `void dispatch_async(dispatch_queue_t queue; void (^block)(void));`
- To add a block to a queue for synchronous execution:
  - `void dispatch_sync(dispatch_queue_t queue; void (^block)(void));`
- If you have a for loop that is used to perform a set of calculations, where each calculation is independent, you can have GCD execute the calculations in parallel (!!):
  - `void dispatch_apply(size_t iterations, dispatch_queue_t queue, void (^block)(size_t index));`

What in the world is “void (^block)(void)”?

- **Welcome to the exciting world of blocks!**
  - blocks are an addition to the C programming language
- They allow the creation of anonymous functions that can be handed around for later execution.
- When a block is created, it acts as a closure, that captures the values of variables in scope at that time
  - We won’t go into memory management details but
    - blocks will automatically retain object instances to ensure they stay around for when the block executes (which may happen long after it is created)
    - blocks created on the stack will migrate to the heap (automatically) if the function that created them is about to go out of scope

Block Basics (I)

- If you need to store a block in a variable, you need to declare the variable correctly. Normally, a variable definition in C looks something like
  - int myInteger; int mySecondInteger = 42;
- With blocks, the name of the variable becomes embedded inside of the type information for the block; (very similar to function pointer declarations)
- Since the block is a function, we need to know its return type and its parameters
  - void (^**block**)(void);
- This declares an uninitialized variable called “block” that returns void and takes no parameters; The ^ symbol indicates we are dealing with a block
  - Assuming we point this variable at a block, we invoke it like any other function: block();

Block Basics (II)

- Here's a more involved example
  - NSString * (^**myVar**)(int a, int b);
- This declares an uninitialized variable called "myVar"
  - This variable can be pointed at any block that takes two integers as parameters and returns an NSString. For instance:
  - myVar = ^(int a, int b) {
    - return [NSString stringWithFormat:@"%d:%d", a, b];
  - };
- We could then invoke this block like this:
  - myVar(23, 42); // returns "23:42"

Block Basics (III)

- Blocks act as closures and will capture the values of any variables that are
  - in scope and referenced by the block
- For example
  - int value = 5;
  - void (^printIt)(void) = ^(void) {
    - NSLog(@"%d", value);
  - }
  - value = 10;
  - printIt(); // prints “5” to standard out

Let’s see this in action;
DEMO

Block Basics (IV)

- Most of the time, you will NOT explicitly store a pointer to a block in your own code; instead, you will create blocks and pass them into method calls
    - You can think of these blocks as anonymous functions being passed around for invocation at the later time
        - This is how blocks play the role of work items in GCD
            - You create a block and pass it to a queue using dispatch_async()
            - The block gets stored in the queue
            - At some point in the future, the block is assigned to a thread and the thread simply invokes it to cause the work item to execute:
                - block(); // assuming block is defined as: void (^block)(void);

Uses of Blocks and GCD

- One use of GCD in OS X apps is to free the main thread from having to perform a long running task
    - Example: Word Counting application
        - Without GCD, a long running count operation can generate the "Spinning Pizza of Death" icon, rendering the UI non-responsive
        - With GCD, the UI remains responsive while the count operation occurs in the background
        - Note: since computers are so dang fast these days, we simulate a long running operation in the program with the use of a call to sleep();
            - Sigh

Return to Prime Finder

- We can use our Prime Finder example from earlier in the semester to show how we can use GCD to handle a compute-intensive task
    - We'll add a GUI to this example, since XCode makes that straightforward
- Here we adopt the approach discussed by the textbook in Chapter 2 (and on slide 34 of Lecture 7) that
    - to keep the cores active for this compute-intensive problem
    - we'll just create a bunch of tasks
    - this will help us deal with the fact that some partitions take a LOT longer to calculate than others
        - by having lots of tasks, cores that finish the fast partitions can eventually be assigned one of the slower partitions

NSOperation

- GCD is nice but its API is fairly low-level
  - and while it adopts an OO-like design (not discussed), it does not provide the abstraction and encapsulation possible with a true OO API
  - In addition, it is difficult to cancel work items and managing dependencies between groups of work items is not straightfoward
- NSOperation addresses these concerns
  - It is a high-level OO API
  - It is easy to specify dependencies between operations
  - It is easy to cancel an operation
  - It is implemented on top of GCD; Apple has done the hard work for you!

Main Concepts: NSOperation

- NSOperation represents a unit of parallelizable work
  - what we've been calling "task" all semester and "work item" in this lecture
  - each NSOperation runs in its own thread
- To define a task, you subclass NSOperation and override its main() method
  - When main() returns, the NSOperation is considered complete
- An operation can be cancelled; as a result, an instance of NSOperation is expected to call isCancelled() to determine if it should shut down
  - it should call this right at the very beginning of main() since an NSOperation might be cancelled before it gets a chance to run
- Some operations can take priority over others; use setQueuePriority() to set
- Finally, some operations can depend on others: add/removeDependency()

Main Concepts: NSOperationQueue

- NSOperationQueue is a class that accepts instances of NSOperation and runs them
    - each in their own thread (as of OS X 10.6); requires care to detect the completion of operations
- NSOperationQueue manages its own thread allocation
    - the developer simply works in terms of NSOperations and NSOperationQueues
- Add operations to a queue with addOperation()
- You can cancel all operations in a queue with cancelAllOperations()
- You can also wait (synchronously) for all operations to finish with waitUntilAllOperationsAreFinished()

Example: MandelOpper

- Will use NSOperation to generate a picture of the Mandelbrot set
- Simple application design
    - MandelOpperAppDelegate: controller that kicks everything off
    - Bitmap: a class to store calculated byte values
    - BitmapView: a view inside of a window that will display the contents of Bitmap
    - CalcOperation: a subclass of NSOperation that calculates one row of the image
    - We create one CalcOperation per row and add it to the queue
- **DEMO**

Summary

- Reviewed concepts, techniques, and examples related to
  - GCD
    - Queue (Serial and Global), Task, Work Item
      - Work Items are implemented as blocks
  - NSOperation and NSOperationQueue
    - Higher-level OO API built on top of GCD
    - NSOperationQueues dynamically assign NSOperations to threads
      - Each NSOperation runs concurrently in its own thread
        - Requires work to ensure that UI is updated in the main thread

Coming Up Next

- Lecture 30: Semester Wrap-Up