Principles of Java Multicore Programming: Highlights

Interrupts: **Join Function**

The Join function has an optional timeout value. This will cause the thread to give up waiting after a certain amount of time. The waiting thread can then attempt to manually interrupt the long running thread, possibly to restart it, or just to give up and move on to another operation. This type of error recovery is important for building stable real world applications.

```
public static void main(String args[])
                    throws InterruptedException {
                        Thread t = new Thread(new WorkerObject());
                        t.start();
                        // Wait maximum of 1 second
                        // for Thread
                        // to finish.
                        *t.join(1000);*

                        *if ( t.isAlive() )* {
                            // Tired of waiting!
                            *t.interrupt();*
                            // Shouldn't be long now
                            // -- wait indefinitely
                            *t.join();*
                        }
                    }
```

Executor Pattern: Thread Pools

Imagine a naive program in which a new thread is created every time the program needs a unit of work done, creating new threads ad infinitum. The compiler and runtime environment would respect these requests, allocating memory and cpu time until the system reaches capacity (and crashes). For small programs thread pools are overkill, but with large-scale applications, it makes sense to separate thread management and creation from the rest of the application.

Thread pools implement what is known in java as the **executor pattern**. This is actually a class of process scheduling algorithms, similar to operating system schedulers.

You give the the executor a unit of work to do, it then sends that unit of work off to be completed, and returns when it is finished. From the high level perspective of our naive program logic, nothing has changed except that instead of directly creating new threads, we create new work units for the executor. What has changed is that the executor is much more conservative than the the naive program. Instead of creating a new thread every time work needs to be done, it creates a pre-defined set of threads at initialization, and then recycles those same threads as it needs them. This creates a much more stable memory footprint, especially for large programs.

From the Oracle documentation: “Consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to all requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be serving HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.”

An important heuristic for thread pools is how many threads can a system sustain, i.e. the optimum thread pool size. This is heavily dependent on the system and the workload of each thread, and often requires large amounts of experimentation. One of the best mathematical frameworks for solving for optimal thread pool size is Littles Law. Kirk Pepperdine states, “Little’s Law says that the number of requests in a system equals the rate at which they arrive, multiplied by the average amount of time it takes to service an individual request.”

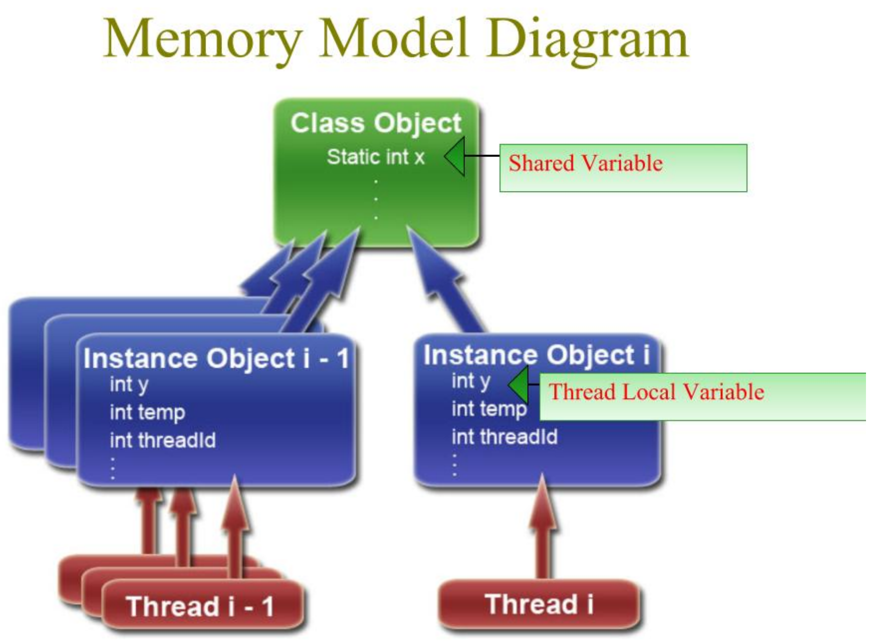
$$L = \lambda W$$

Where L is the average number of customers in the system, λ is the arrival rate, and W is the average time it takes to service a customer.

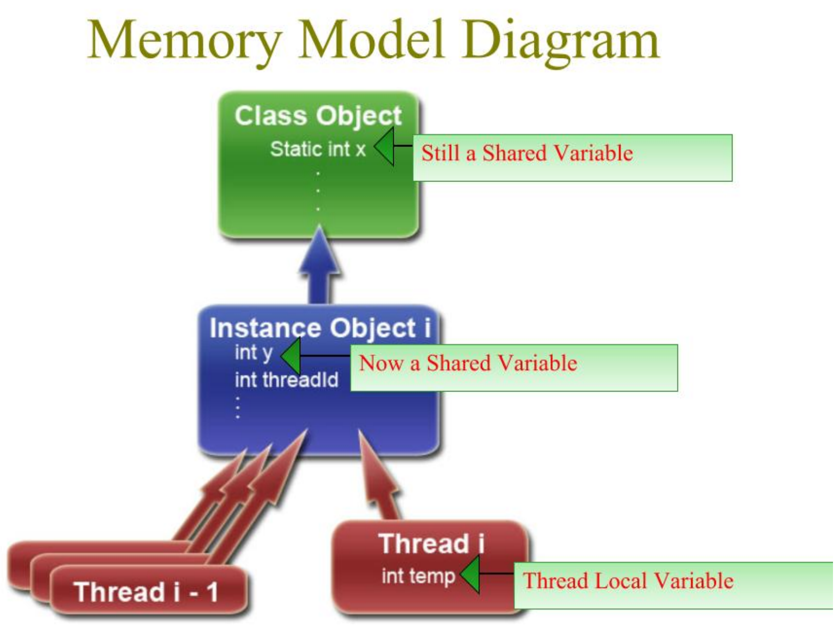
We measure the rate at which work unit requests arrive and the average amount of time to service them. We can then plug those measurements into Little’s Law to calculate the average number of requests in the system. If that number is less than the size of our thread pool then the size of the pool can be reduced accordingly. In the case where we have more requests waiting than being executed, we need to determine is if there is enough capacity in the system to support a larger thread pool. To do that we must determine what resource is most likely to limit the application’s ability to scale.

Memory Models

One Thread / One Object



Multiple Threads / One Object



Java Locking Concepts

Monitor Locks Review

A Monitor Lock consists of a mutex object and a condition variable. The condition variable is basically a container of threads that are waiting on a certain predicate. Monitors provide a mechanism for threads to temporarily give up a critical section in order to wait for some condition to be met, before regaining access to the critical section and resuming their task. Every time a thread releases the mutex, it wakes up all waiting threads so that they may test their condition. Note: the predicate is optional. If we let the predicate be the trivial predicate (true), then the a monitor lock becomes a mutex with a queueing system. This is how the synchronized keyword in java works.

Synchronized in Java

The built in locking mechanism used in Java is initialized with the **synchronized** keyword. This is a monitor lock! Note: This means that all objects in java have: *wait()*, *notify()*, and *notifyAll()* available to them.
Why Java chose monitor locks:

- All the synchronization code is centralized in one location and the users of this code don't need to know how it's implemented.
- The code doesn't depend on the number of processes, because monitors have a built in queuing system. This is opposed to a semaphore counter.
- You don't need to release something like a mutex, so you cannot forget to do it.
- Java implements Monitors with mutexes that have a feature known as *reentrant synchronization.*

From the Oracle documentation: "if synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block." Reentrant synchronization solves this problem.

Java Atomic Operations

Volatile

Volatile only guarantees order inside a single thread is preserved, but there is no ordering guarantees between multiple threads. See the example below:

```
class VolatileTest {
                static *volatile* int i = 0, j = 0;

                static void threadOne() { *i++; j++;* }

                static void threadTwo() { System.out.println(*"i=" + i* + *" j=" + j*); }

                }
```

Note: i will always increment before j, but thread 2 could read a value for i, sleep for a very long time, and then come back and read a MUCH larger value for j. This would not happen with *synchronized*. Volatile is important for implementing complex non-blocking data-structures and algorithms, but it is not a replacement for locks / synchronized.

Compiler Optimizations

Volatile

Examine the example below. This is a very tricky scenario. The Java compiler has no way to check for *writes from another thread.* It will see that loop, not see any changes to to the checkMe variable, and *optimize* it to be an infinite loop. This is obviously incorrect, as the the second thread changes the value of checkMe. The solution to this problem is to declare checkMe as **volatile**. Volatile guarantees that all reads of the variable must be up to date from the last write. Since the compiler has no way of knowing when or where that write came from, the compiler will not optimize the code.

```
class OptimizeMe {

            static boolean checkMe = true;

            public threadOne() {
             // will be optimized to be while(true)
              *while(checkMe == true)* {
                //Do something that does not change the value of checkMe.
              }
            };

            public threadTwo() {
              Thread.sleep(4000);
              *checkMe = false;*
            }
          }
```

Java String Types

	String	**StringBuilder**	**StringBuffer**
Thread Safe	Yes	Yes	No
Synchronization Overhead	Low	High	None
Memory Usage	High	Low	Low

References

- The Official Java Tutorials on Concurrency
- Java Concurrency Package API
- Java Memory Model
- Java Concurrency in Practice (Book)
- Java Performance Tuning
- Java Specification on Final Fields
- Comparison of String Class Alternatives
- Five things you didn't know about Java Multithreading
- Thread Pool Performance Tuning
- Implementing a Monitor Lock in Java
- Prisoner Riddle Case Study with Solutions