Performance Based Programing Assignment

Eamonn Mc Gonigle

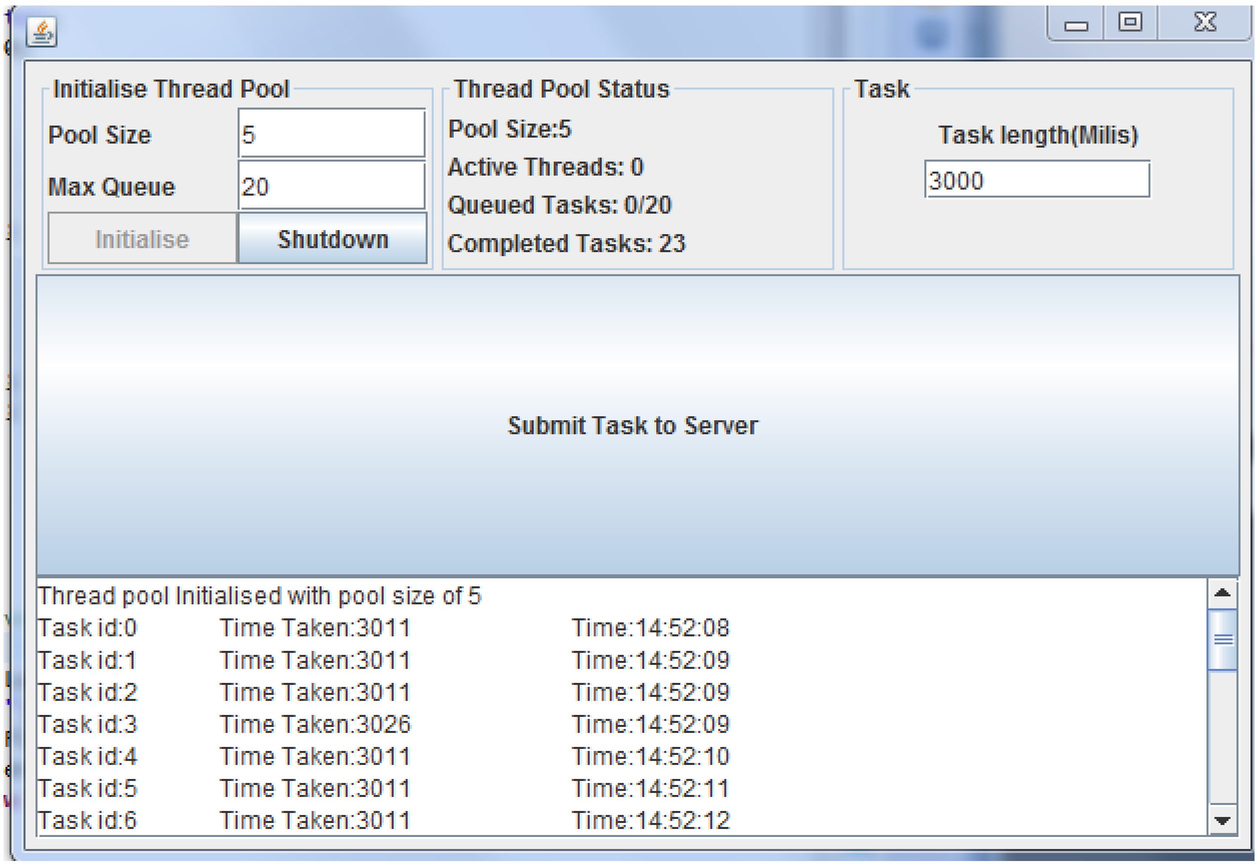
L00093050

8th December 2014

1 Introduction

Thread pools offer a number of performance advantages when building server applications that process a large number of small tasks. However programming with thread pools does not remove the typical concurrency concerns associated with all multi threaded application development(Goetz, 2002). Measures must be taken to ensure that access is coordinated amongst all active threads.

This artefact aims to implement a thread pool, while ensuring access to shared resources is synchronised.



Pool Size: 5
Max Queue: 20

Pool Size:5
Active Threads: 0
Queued Tasks: 0/20
Completed Tasks: 23

Task length(Milis): 3000

```
Thread pool Initialised with pool size of 5
Task id:0        Time Taken:3011        Time:14:52:08
Task id:1        Time Taken:3011        Time:14:52:09
Task id:2        Time Taken:3011        Time:14:52:09
Task id:3        Time Taken:3026        Time:14:52:09
Task id:4        Time Taken:3011        Time:14:52:10
Task id:5        Time Taken:3011        Time:14:52:11
Task id:6        Time Taken:3011        Time:14:52:12
```

Figure 1.1: Graphical User Interface

2 Artefact Description

The program or artefact allows the user create a fixed thread pool with a bounded queue of their own specification. The user initialises a thread pool then submits tasks (of a specified length) to be processed by the threads in the pool. When a task is processed it appends to to a log file and a text area on the GUI. As there is many threads writing to the same file, potentially at the same time, a synchronised method is employed. When they the user attempts to add more tasks than the queue and thread pool can hole they are notified.

3 Concepts Under Review

The artefact utilises two techniques from the following two performance based programming topic areas; thread pools and synchronisation.

3.1 Thread Pools

Thread pools allow the reuse of threads in a program. This limits the number of threads running in a program, cutting down on the computational and memory overhead associated with creating and maintaining multiple threads at a time. A common problem with having to many active threads is resources thrashing, where the system runs out of memory due to an unforeseen number of threads running concurrently. Thread pools provide a solution to this problem. Instead of creating a new thread for every new task, the task will be put into a queue (Blocking Queue). When a thread in the thread pool becomes idle it will dequeue the next task from the queue.

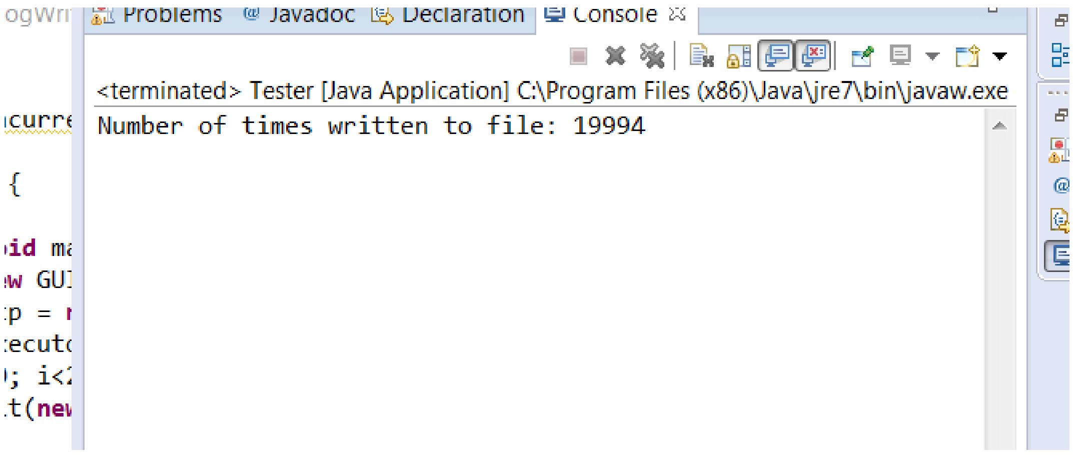
The artefact implements a thread pool by utilising Java's TheadPoolExecutor class(Java, 2014). This class was used instead of the more convenient factory methods also provided by Java API. Java recommends using these methods in most situations, however by using the ThreadPoolClass more control can be had over the thread pool. This facilitated the monitoring of the state of the pool as well as the implementation of the RejectedExecutionHandler (Java class) to handle situations where the pool queue is full.

3.2 Synchronisation

Synchronisation in Java programming is the practise of guarding concurrent access to a block of code or variable. This is particularly important when multiple threads are attempting to access the same resource such as in the case of the artefact. Each task which is submitted to the thread pool writes to a common log text file upon completion. Each task (an implementation of Java's Runnable interface) is does this by calling a method printToLog(). To prevent two tasks calling this method at the same time and attempting to access the same file, the method was changed to a synchronised method. To test whether this in fact made a difference and protected the code from concurrency issues, a tester class and a static counter was created to ensure the method is called as many times as expected. Figure's 3.1 and 3.2 below demonstrate the results of the tester class.

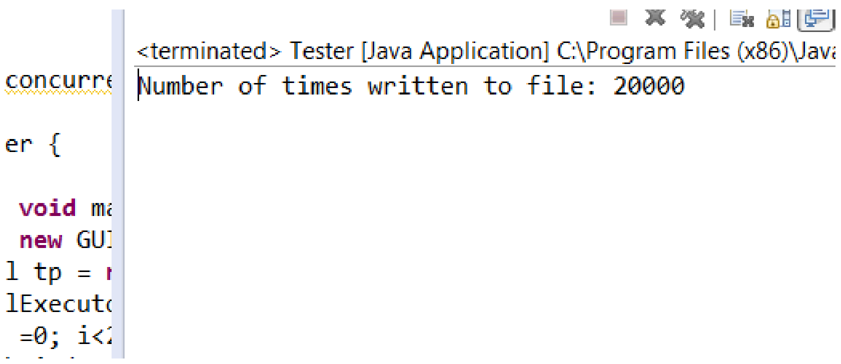
The method was called 20000 times. As is shown in the screen shots below, when multiple threads tried to access the shared resource a race condition was created. A race condition can occur when multiple processes are attempting to change a shared resource. To change the resource (increment an int counter variable in this example), a few steps must be followed. The thread must first read the original value. Then increment that value by one. Finally the variable is updated with the new value. However if another thread attempts to carry out the same action before these steps complete, it will be working with outdated data the variable will only be incremented once.

By ensuring that the programs multiple can safely access and modify the int counter as expected when it is encapsulated in a synchronised method, it can be assumed that writing to a text file will be no different and the program is indeed thread safe.



```
<terminated> Tester [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe
Number of times written to file: 19994
```

Figure 3.1: Test performed calling unsynchronised method



```
<terminated> Tester [Java Application] C:\Program Files (x86)\Javi
Number of times written to file: 20000
```

Figure 3.2: Test performed calling synchronised method

4 Wider Implications

The techniques investigated and implemented in this artefact are of great use in many industry programming scenarios.

The artefact's thread pool implementation is similar in its operation to a multi threaded server application. It can for example be thought of as a HTTP server: A task or request is sent to a remote server and either processed or queued up to be processed.

The use of a bounded queue size graceful rejection of any task once the the queue is filled, provides protection against server overload, either from legitimate traffic or from denial of service (DOS)attacks.

The artefact has multiple threads writing to a single log file. This is a common scenario, that introduces the risk of multiple threads attempting to append to a file at the same time and could result in incorrect data being logged (ex. does not records logs in correct order). By declaring the method that writes to the file as synchronised this problem is solved as the thread must wait until no other thread is running the method until it can gain access.

5 Conclusions and Results

By implementing a thread pool in Java an appreciation of the power and elegance of the design pattern was gained. Thread pools offer an effective way to more efficiently utilise system resources in many multi threaded scenarios.

Server applications that process a large number of small requests are perhaps the most obvious implementation in industrial programming. However the thread pool should be considered in any situation where a large number of small tasks must be processed and the overhead associated with creating and maintaining threads is particularly expensive. The way that thread pools facilitate thread reuse make it particularly suitable for situations where the velocity of tasks to be process fluctuates (ex. a web server). Using a thread pool limits the number of idle threads and prevents resource thrashing by limiting the creation of new threads during times of unforeseen activity.

Thread synchronisation was successfully achieved in developing the artefact. This was achieved by wrapping interaction with a shared resource in a synchronised method. Testing this method with and without the synchronised keyword highlighted the importance of restricting concurrent access to shared resources.

References

Goetz, B., 2002. Java theory and practice: Thread pools and work queues. [accessed 27/11/2014]. URL `http://www.ibm.com/developerworks/library/j-jtp0730/j-jtp0730-pdf.pdf`

Java, 2014. Class threadpoolexecutor. [accessed 27/11/2014]. URL `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html`

Code References

- Java Advanced Programming Tutorial 11 How To Append To A File, 25 April 2013, viewed 8 December 2014, <http://www.youtube.com/watch?v=sLWfRzgo__4&NR=1>.
- Kumar, P. (2013). Java Thread Pool Example using Executors and ThreadPoolExecutor | Java Code Geeks. [online] Java Code Geeks. Available at: http://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html [Accessed 8 Dec. 2014].