Technical Brief

James Peach

12 March 2015

1 Introduction

Threads are used within applications that wish do accomplish many things at once. Most languages provide support for threading and java is no different. Java allows threading via the use of the ***Thread*** class and the ***Runnable*** Interface. These provide a way of defining a task for a thread to accomplish and also a way of representing a thread and interacting with it.

Each java process is inherently created with a single *Main* thread. This thread can then be used to create other threads. Once the last thread of execution has exited, the application will then exit. The new threads have access to any of the fields in scope of its creation and this is the main way of the threads interacting with each other and the operating system. Threads have some intrinsic actions provided by the ***Thread*** API, they can sleep and return execution to the other running threads by calling the sleep method:

$$Thread.sleep();$$

This will sleep the current thread for the specified amount of time or until another thread causes this one to be interrupted. this can then be caught by the application and handled correctly.

Threads also have the ability to wait for child threads to finish executing. For example they might start two threads and pass each some computation work. and then want to wait till each thread has finished processing results before continuing on the main thread. If t is a ***Thread*** then:

$$t.join();$$

Will cause the current thread to wait for t to finish before continuing.

When multiple threads are working at the same time and share access to memory there is the ability for two threads to access the same value or reference and for them to read incomplete changes and also to tread on each others values. This family of problems is called concurrency issues. Java supports different ways of dealing with this one such method is with the *Synchronized* modifier. this can be applied to methods and will ensure that the method is not called by two different threads at the same time for the same instance of an object.

Another way to minimise effects is to create immutable objects. these are objects that do not change state. they must be passed by reference from one thread to the next and because they cant be changed then they will always be up to date. This circumvents part of the problem but creates another. How do we pass objects between threads in a thread safe manner? The Java class **BlockingQueue** allows objects to pass objects throughout a queue interface with a buffer. This can be used to ensure objects are correctly send and received on both ends as a write to the queue will wait to ensure there is a place to put it in. And on the other hand a read will wait until there is something to read.

2 Relevance

The ability to run several threads at once has always been sought after by programmers even before the ability to have many cores in a single computer. You always wanted inputs to be handled to the system no matter what it was doing before, and you always want it co go back to doing what it was doing before. This is in essence the most intrinsic use of threads. For example if you were printing a document and you wanted to check the time you wouldn't want to have a half printed document and the correct time.

However in a modern CPU there is often the ability to run many threads truly concurrently and this is important in order to achieve the amount of computation many of the more demanding applications need in a sensible amount of time, or to run many different applications at the same time so called multitasking. There are also a family of algorithms that are designed to work well when divided up into many different threads of execution.

3 How it's used

There are many different ways and places ***Thread*** 's might be used. One such example is in a HTTP web server. The HTTP server is based on a TCP socket server. The socket server should accept the new connection and then move the new connection to a new socket so that the original port can continue to accept new connections. Typically this is done with one thread responsible for accepting a new connection and when it does it spawns a new worker thread to deal with the input output of the socket.

This is just one application of threads within an application many other applications exist in every day computing. For example the Chrome browser from Google is famous from having a different process for each tab this is for security as well as performance reasons. However while this is technically still multi-threading, the different processes do not share memory. Instead the developers must use a technique called IPC or inter process communication. This is accomplished in different ways on different systems, for instance on windows sockets are used to send messages between processes and on Linux pipes are used to send buffered messages.

In many ways programming between multiple threads, can be complicated and issues can arise when threads are accessing the same piece of data at the same time as in modern processors there may be hardware support for multiple concurrent threads to be running at the same time. There are ways around this by way of using Locks and in java, Synchronized methods and blocks. Java also makes it easy for developers to schedule tasks on separate threads by way of an ***Executor*** class. This class provides a way of assigning tasks to a queue and having them execute on a pool of pre created threads. This overcomes much of the boilerplate code necessary to get processing done in a multi-threaded application.

4 Resources

- **Oracle - Java Tutorials: Concurrency** :- http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html