Japanese Concurrency with Groovy

Using threads and AtomicInteger

From Groovy you can use all of the normal concurrency facilities in Java and combine them with threads and closures as necessary. E.g. a (slightly modified) atomic counter from a Groovy example here:

```
import java.util.concurrent.atomic.AtomicInteger

def counter = new AtomicInteger()

synchronized out(message) {
    println(message)
}

def th = Thread.start {
    for( i in 1..8 ) {
        sleep 30
        out "thread loop $i"
        counter.incrementAndGet()
    }
}

for( j in 1..4 ) {
    sleep 50
    out "main loop $j"
    counter.incrementAndGet()
}

th.join()

assert counter.get() == 12
```

The output will be something like this:

```
thread loop 1
main loop 1
thread loop 2
thread loop 3
main loop 2
thread loop 4
thread loop 5
main loop 3
thread loop 6
main loop 4
thread loop 7
thread loop 8
```

Fibonacci with Executors

A Groovy script to naively calculate the Fibonacci series inspired by the example here.
Note: a version using memoizing will be much more efficient but this illustrates using Java's built-in concurrency primitives from Groovy.

```
import java.util.concurrent.*

CUTOFF = 12    // not worth parallelizing for small n
THREADS = 100

println "Calculating Fibonacci sequence in parallel..."
serialFib = { n -> (n < 2) ? n : serialFib(n-1) + serialFib(n-2) }
pool = Executors.newFixedThreadPool(THREADS)
defer = { c -> pool.submit(c as Callable) }
fib = { n ->
  if (n < CUTOFF) return serialFib(n)
  def left = defer{ fib(n-1) }
  def right = defer{ fib(n-2) }
  left.get() + right.get()
}

(8..16).each{ n -> println "n=$n => ${fib(n)}" }
pool.shutdown()
```

Which produces this:

```
Calculating Fibonacci sequence in parallel...
n=8 => 21
n=9 => 34
n=10 => 55
n=11 => 89
n=12 => 144
n=13 => 233
n=14 => 377
n=15 => 610
n=16 => 987
```

If you want to convince yourself that some threads are actually being created, replace the last `each` line with:

```
showThreads = { println Thread.allStackTraces.keySet().join('\n') }
(8..16).each{ n -> if (n == 14) showThreads(); println "n=$n => ${fib(n)}" }
```

which will add something like the following to your output:

```
Thread[Finalizer,8,system]
Thread[Reference Handler,10,system]
Thread[TimerQueue,5,system]
Thread[pool-1-thread-2,5,main]
Thread[Thread-3,6,main]
Thread[Thread-2,6,main]
Thread[pool-1-thread-3,5,main]
Thread[AWT-Shutdown,5,main]
Thread[Signal Dispatcher,9,system]
Thread[Attach Listener,5,system]
Thread[AWT-EventQueue-0,6,main]
Thread[DestroyJavaVM,5,main]
Thread[pool-1-thread-1,5,main]
Thread[Java2D Disposer,10,system]
Thread[pool-1-thread-5,5,main]
Thread[pool-1-thread-6,5,main]
Thread[pool-1-thread-4,5,main]
Thread[AWT-Windows,6,main]
```

Fibonacci with Functional Java

If you wish to make use of the functionaljava (tested with 2.1.3) library (see the link to the original example), you could use a Groovy program such as this:

```
import fj.*
import fj.control.parallel.Strategy
import static fj.Function.curry as fcurry
import static fj.P1.curry as pcurry
import static fj.P1.fmap
import static fj.control.parallel.Actor.actor
import static fj.control.parallel.Promise.*
import static fj.data.List.range
import static java.util.concurrent.Executors.*

CUTOFF  = 12   // not worth parallelizing for small n
START   = 8
END     = 16
THREADS = 4

pool = newFixedThreadPool(THREADS)
su   = Strategy.executorStrategy(pool)
spi  = Strategy.executorStrategy(pool)
add  = fcurry({ a, b -> a + b } as F2)
nums = range(START, END + 1)

println "Calculating Fibonacci sequence in parallel..."

serialFib = { n -> n < 2 ? n : serialFib(n - 1) + serialFib(n - 2) }

print = { results ->
  def n = START
  results.each{ println "n=${n++} => $it" }
  pool.shutdown()
} as Effect

calc = { n ->
  n < CUTOFF ?
    promise(su, P.p(serialFib(n))) :
    calc.f(n - 1).bind(join(su, pcurry(calc).f(n - 2)), add)
} as F

out = actor(su, print)
join(su, fmap(sequence(su)).f(spi.parMapList(calc).f(nums))).to(out)
```

Using java.util.concurrent.Exchanger

An example using `Exchanger`. We have two threads - one keeping evens and exchanging odd values with the other thread. Meanwhile the other thread is working in reverse fashion; keeping the odds and exchanging the evens. The algorithm here is dumb in that it relies on the same number of swaps for each side - which is fine for this example but would need to be altered for more general input values.

```
def x = new java.util.concurrent.Exchanger()
def first=1..20, second=21..40, evens, odds
def t1 = Thread.start{ odds = first.collect{ it % 2 != 0 ? it : x.exchange(it) } }
def t2 = Thread.start{ evens = second.collect{ it % 2 == 0 ? it : x.exchange(it) } }
[t1, t2]*.join()
println "evens: $evens"
println "odds: $odds"
```

Which has the following output:

```
evens: [2, 22, 4, 24, 6, 26, 8, 28, 10, 30, 12, 32, 14, 34, 16, 36, 18, 38,
20, 40]
odds: [1, 21, 3, 23, 5, 25, 7, 27, 9, 29, 11, 31, 13, 33, 15, 35, 17, 37,
19, 39]
```

Catching Exceptions with an Exception Handler

When catching Exceptions with an exception handler in Groovy Scripts, there is potential for interaction with Groovy's runtime which catches exceptions and filters stacktraces. The best way to avoid this interaction is to create your own thread and call `setDefaultUncaughtExceptionHandler` directly on that thread instance as per below:

```
def th = Thread.start {
    println 'start'
    println Thread.currentThread()
    sleep 1000
    throw new NullPointerException()
}
th.setDefaultUncaughtExceptionHandler({t,ex ->
    println 'ignoring: ' + ex.class.name
} as Thread.UncaughtExceptionHandler)
th.join()
```

More Information

See also:

1. Japanese Functional Programming with Groovy
2. Multithreading with SwingBuilder