Java Channels:
a Learning Experience

P.H.Welch
Computing Laboratory
University of Kent at Canterbury, CT2 7NF, England

Abstract. This document transcribes a series of emails in May, 1996, which account for my learning experience with Java threads. The motivation for this arose from the WoTUG-19 conference at Nottingham-Trent University, which included a pre-conference weekend for the WoTUG committee that was convened to examine technical relationships between the Java and occam programming languages.

Java had drawn itself to our attention as it had drawn itself to everyone's attention. However, what was of special interest to us was that a major part of its *hype* echoed many of the key messages that the occam community had been trying to get across (without, exactly, total success) for the past decade. For example, the need for security to be built in to the language (motivated in the case of occam by the requirements for safety-critical embedded applications, and in the case of Java by the danger of pulling executables straight off the web). Another aspect that struck a chord was the binding of a model for concurrency directly into the language (which gives much greater awareness to the system designer of what is actually being designed, and information to the compiler that can be used to prevent a wide class of error and to optimise code generation).

The particular problem exercised in these emails is the construction of robust and efficient *glue* objects (channels/buffers) for connecting active processes (i.e. Java objects containing one or more threads). It proved surprisingly hard to get this right.

1 The Original Example Problem

At the WoTUG-19 meeting, Oyvind Teig (from Autronica in Trondheim, Norway — a company involved in the design and construction of embedded control systems for maritime applications) presented a Java version of a standard occam benchmark for measuring process (i.e. thread) context-switch times.

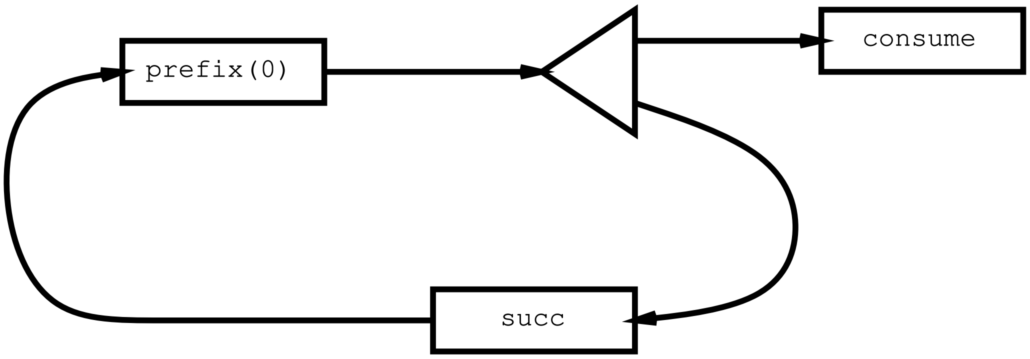

Figure 1: The `CommsTime` benchmark

This benchmark (see Figure 1) consists of four *processes* (`prefix(0)`, `delta`, `succ` and `consume`) connected by *channels*. In Java, each process is an *active object* (i.e. thread) and each channel is a *passive object* (i.e. with no life of its own — its methods being called as part of threads within active objects).

Now, our experience from the occam world in writing multi-threaded code shows that designing and implementing *active* objects is very natural and easy. Indeed, active objects are all you can have in occam so we don't have much choice! What we wanted to discover was whether working with active objects in Java would be equally straightforward. What we discovered was that this is (probably) the case, but only once we had constructed the *passive* objects needed to connect them together. Passive objects contain shared data that active objects manipulate simultaneously — such manipulation must be properly managed if chaos (i.e. non-determinism) is not to ensue. Again, in occam we are spoilt because the only passive objects needed are channels (which contain the shared data-structures used for synchronisation) and all the tricky algorithms for manipulating them are pre-packaged as language features (called *channels* and `SHARED` *channels*) and pre-implemented in software (or hardware) nano-kernels. So, problems such as *race hazards* don't occur! In Java, we have to program these passive objects ourselves and this turns out to require a surprising degree of care.

Back to the benchmark: the `prefix(0)` thread kicks things off by outputting `0` on its output channel and then going into an infinite *input-and-forward* cycle. The `delta` thread – the triangle in the picture – just cycles by waiting for input and then forwarding two copies (in parallel) to each of its output channels. The `succ` thread cycles through waiting for input, incrementing the number that arrives and forwarding the result to its output channel. Finally, the `consume` thread simply consumes the arriving numbers (which will be the sequence of natural numbers) and times how fast they arrive.

The `prefix(0)` and `succ` threads synchronise with other threads twice per cycle, the `delta` thread three times and the `consume` thread once per cycle. That makes 8 synchronisations (or context-switches) per number that `consume` consumes. Dividing the cycle time reported by `consume` by 8 gives us the basic overhead for thread synchronisation (the time spent by the threads themselves doing other things being negligible). The KRoC occam system on a 60-MHz SPARC-20 gives a cycle time of around 3.7 micro-seconds — i.e. a context-switch time of under half a micro-second. The first implementation of Java channels presented by Oyvind was based upon the `Piped I/O Stream` channels provided by Sun. What he showed was that the active objects were, indeed, not-too-bad to write in Java ... but the cycle time (on the same SPARC-20) was around 2 seconds. Even allowing a factor of ten for the Java interpreter, the factor of one million wasn't too good. What was the matter? Now, read on ...

2 Please check out the Appendices

For this workshop, the emails are reproduced verbatim in the appendices:

1. Analysis of the trouble with `Piped I/O Stream`; a quick summary of Java threads; a zero-buffered synchronised (i.e. occam) channel class for Java with no busy-waits; a blocking FIFO class for Java with no busy-waits; single reader/writer only; discussion.
2. Small problem with `IllegalThreadStateException` (it doesn't seem to do quite what Sun's Java Tutorial says); wanted as a possible route to allowing multiple readers/writers.
3. Reply from Gerald Hilderink to the above.
4. Reply to Gerald.
5. Solution to the *infinite overtaking problem* when supporting buffers with multiple readers/writers (described in Appendix 1); doesn't use `IllegalThreadStateException`; performance times; discussion on the non-compositional nature of Java monitor semantics compared with the compositional (i.e. *WYSIWIG*) semantics of CSP/occam processes; note that designing only with active objects synchronising through our pre-packaged *glue* gives occam *WYSIWIG* semantics for Java.
6. Simplified version of the above; use of `synchronized` blocks rather than methods; minimalist programming — the use of classes with no user-defined data or methods (only those inherited automatically from the basic `Object` class).
7. A channel class also secure for multiple readers and writers; small fly in the ointment?
8. Patch to squash above fly; brief discussion.
9. Nothing to do with the above, but may be interesting: problem of accidental introducing of aliases ... leading to complex semantic singularities.