Lab 5

Memory Barriers

Compilers, run-time systems, and hardware might process the code in a way that you might not intuitively expect. For example, any of the following might apply to the execution of a method:

- The compiler may rearrange the order of the statements. If the method is inlined, the compiler may further rearrange the order with respect to yet other statements.
- The processor may rearrange the execution order of machine instructions corresponding to the statements, or even execute them at the same time.
- The memory system (as governed by cache control units) may rearrange the order in which writes are committed to memory cells corresponding to the variables. These writes may overlap with other computations and memory actions.
- The compiler, processor, and/or memory system may interleave the machine-level effects of two consecutive statements. For example on a 32-bit machine, a high-order word of a variable may be written first, followed by the write to another variable, followed by the write to the low-order word of the first one.
- The compiler, processor, and/or memory system may cause the memory cells representing variables not to be immediately updated after each operation, but instead to maintain the corresponding values (for example in CPU registers) in such a way that the code still has the intended effect.

In a sequential language, none of this can matter as long as program execution obeys as-if-serial semantics. Sequential programs cannot depend on the internal processing details of statements within simple code blocks, so they are free to be manipulated in all these ways. This provides essential flexibility for compilers and machines. Things are different in concurrent programming. Not only may concurrent executions be interleaved, but they may also be reordered and otherwise manipulated in an optimized form that bears little resemblance to their source code. This can be the source of subtle concurrent programming errors. Thus, on multiprocessors, guaranteeing conformance often requires emitting barrier instructions.

Memory barriers, or **fences**, are a set of processor instructions used to apply ordering limitations on memory operations. Memory barriers are only indirectly related to higher-level notions described in memory models such as "acquire" and "release". And memory barriers are not themselves "synchronization barriers". Memory barrier instructions directly control only the interaction of a CPU with its cache, with its write-buffer that holds stores waiting to be flushed to memory, and/or its buffer of waiting loads or speculatively executed instructions. A properly placed memory barrier prevents non-deterministic behaviour by forcing the processor to serialize pending memory operations. Finally, memory barriers are used to achieve an equally important element of concurrent programming: **visibility**.

Java benefits of a **Just-in-Time (JIT)** compiler that optimizes the execution of the application. The JIT compiler is activated when a Java method is called and compiles the bytecode of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it.

In order to see the assembler code, the application must be run at the command line with the following Java options:

```
$ java -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly Application
```

In order to fine tune the output further, a specific method can be indicated with the options

`-XX:PrintAssemblyOptions=hsdis-print-bytes`

`-XX:CompileCommand=print,Application.method`.

For this to work, you need the disassembler plugin hsdis.

Note that the JIT only compiles methods considered "hot", i.e., that are executed for a sufficiently large number of times during the run of the application. The runtime option `–XX:+PrintCompilation` can be used to check if the desired method was compiled or not.

The output might vary depending on the hardware. However, memory barriers and other details can be observed on any architecture. See more information on the websites below.

In addition to the command line tool, there is also a graphical interface version that lets the user check the assembly code and generates useful statistics, called JITWatch.

Credits:
http://www.infoq.com/articles/memory_barriers_jvm_concurrency
http://www.beyondjava.net/blog/java-programmers-guide-assembler-language/
http://mechanical-sympathy.blogspot.ch/2011/07/memory-barriersfences.html
http://gee.cs.oswego.edu/dl/jmm/cookbook.html