Locks and Threads and Monads—OOo My

Stephan Bergmann
StarOffice/OpenOffice.org
Sun Microsystems

Locks and Threads and Monads—OOo My

1 - Tomorrow's hardware...

2 - ...and today's software

3 - Stateful vs. functional...

4 - ...in parallel

5 - UNO to the rescue?

CPUs: Broader, Not Faster

- Today, CPU speed no longer increases the way it did all those decades.
- Instead, consumer machines are equipped with increasing numbers of parallel execution units (multiple CPUs, hyperthreading).
- Herb Sutter: “[...] applications will increasingly need to be concurrent if they want to fully exploit CPU throughput gains [...]”

OOo Today

- Mostly single-threaded application, based around a GUI event loop.
- Few additional threads:
  - filename autocompletion in file picker, ...,
  - remote UNO connections.
- An example consequence: Opening a large writer document takes a while, you cannot start searching through it right away.

OOo Today

- Much of the OOo code written with a single-threaded application in mind.
- Multi-threading support added afterwards (“global solar mutex”).
- An example consequence: Multiple incoming remote UNO connections (i.e., multiple active threads) likely crash OOo.

Shared state threading

- Extremely hard to get right.
- Example: What is a recursive mutex good for?
  > David Butenhof: “A correct and well understood design does not require recursive mutexes.”
- Example: Issue 67191, `osl_waitCondition` not working properly from day one, detected years later.
- Example: Are `Old/NewValue` in `PropertyChangeEvent` of any use?

Dilemma

- Can we reasonably expect to make use of multiple parallel execution units in OOo using the shared state threading model we love and hate?
- No!
- What then?

A little rant intermezzo

- Many CS concepts seem to be little known across the industry:
  - “So, what your suggestion amounts to is to add closures to OOo Basic, right?” — “Closures???”
  - “But UNO does not support structural subtyping.” — “C struct types???”
  - Scott Meyers: “I have a Ph.D. in Computer Science, and I’d never heard of F-bounded polymorphism.”

Look around!

- Other approaches to programming (concurrent) applications:
  - Declarative models with logic variables (e.g., Oz).
  - Non-strict functional models (e.g., Haskell).
- Philip Greenspun: “Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified bug-ridden slow implementation of half of Common Lisp.”
- Remember: There are many interesting approaches, and there is no silver bullet.

Oz

- Logic (dataflow) variables and lightweight threads:

```
thread List = "a"|X1 end
thread X1 = "b"|X2 end
thread X2 = "c"|nil end
{Length List}
```

```
fun {Map F Xs}
  case Xs of nil then nil
  [] X|Xr then thread {F X} end|{Map F Xr}
  end end
```

- *Concepts, Techniques, and Models of Computer Programming* by van Roy and Haridi.

Haskell

- Non-strict (“lazy”):

```
f :: Float -> Float
f _ = 5.0
f (1.0 / 0.0)                    -- 5.0

squares :: Int -> [Int]
squares n =
  take n (map (\x -> x * x) [1 ..])
```

infinite

- Monadic IO:

```
main :: IO a

wordCount :: IO Int
wordCount = do putStr "input: "
               l <- getLine
               return (length (words l))
```

Software Transactional Memory

- Don't pessimistically lock data, but optimistically use the data and then commit a bunch of operations: Either succeeds or fails and restarts.
  - > Easier to program.
  - > Works best in low-contention scenarios.
  - > Nicely integrates into Haskell:

```
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
atomically :: STM a -> IO a
```

And its not only concurrency

- For example, resource management:
  - > C `malloc/free`: a nightmare to get them properly paired.
  - > C++ RAII: better, but (a) often not used (witness many OOo crash reports), and (b) bad when destruction can fail (`fclose`).
  - > Java `try/finally`: cumbersome, esp. when using multiple resources.
  - > Haskell: higher order functions!

And its not only concurrency

- ```
  withOpenFile :: Handle ->
                  (Handle -> IO a) ->
                  IO a
  withOpenFile h f = finally (f h) (hClose h)

  copyAndClose :: Handle -> Handle -> IO ()
  copyAndClose h1 h2 =
    withOpenFile h1 (_ ->
    withOpenFile h2 (_ ->
    do x <- hGetContents h1
       hPutStr h2 x
       return () ))

  do h1 <- openFile "input" ReadMode
     h2 <- openFile "output" WriteMode
     return copyAndClose h1 h2
  ```
  ```

UNO

- Conceptually, UNO consists of threads concurrently invoking methods on (shared) objects.
- Each UNO object has to ensure that concurrent invocations of its methods are safe.
  > Hard to avoid deadlock.
  > Single method calls are often the wrong locking granularity.
  > Unnecessary locking costs in single-threaded use.
  > Java had the same problem (e.g., `StringBuffer` → `StringBuilder`).

UNO

- Does this fit a (massively) concurrent world?
- Not really:
  > The emerging threading framework tends to cluster objects in cages when they should be free (individual paragraphs of a text document model).

  > The two-level approach (language-independent model on top of language bindings) hampers innovation (e.g., language-supported lightweight threads, language-supported STM).
- (UNO *does* help to integrate new languages.)

Conclusion

- *An OOo running correctly on 1–2 processing units is important, but an OOo running efficiently on 8–16 processing units will become just as important.*
  - > Find places in OOo where things can be done in parallel.
  - > Know how to write good code that achieves this.
  - > Have fun with a snappy application.

Mistrust all enterprises that require new clothes.

—E. M. Forster