redhat.

A multi-tasking wordset for Standard Forth

Andrew Haley
Consulting Engineer
8 September 2017

Background

Forth has been multi-tasking for almost 50 years. It's time to standardize it.

- Chuck Moore's early Forths had a simple and efficient multi-tasker
- This was refined by others at Forth, Inc. and eventually became the core of polyFORTH
- Many Forths have used a version of this multi-tasker since then, and because of that there is some practical portability of multi-tasking programs between Forth systems. These include products from MPE and Forth, Inc. as well as free systems such as F83

Goals

- To make it possible to write multi-tasking programs in Standard Forth
- These standard multi-tasking programs will run unaltered on both co-operative (round-robin) and time-sliced schedulers, on hosted and freestanding systems

Design criteria

- No innovation!
  - Wherever possible, use established practice from Forth systems
  - Where no established practice exists in Forth, take inspiration from other programming languages, especially C
- This should be a low-level wordset
  - Standardize the most basic elements of multi-tasking, eschewing more complex objects such as queues and channels. These can be provided by libraries, based on this wordset

Design criteria

- Completeness
  - This wordset must provide everything that is necessary to write libraries and multi-tasking programs without such needing to use carnal knowledge
- Simplicity
  - Given that this is a Forth standard, simplicity hardly needs mentioning, but it must be paramount after completeness
  - Simplicity mostly “falls out” of the design as a result of following common Forth practice

Design criteria

- Efficiency
  - While the greatest possible efficiency will always result from a system-specific wordset, we can get very close with a standard wordset
  - This wordset should work well on large multi-core systems but not impose a significant burden on very small single-core systems
- Portability
  - The wordset shouldn't require anything that is unavailable on a system that is capable of realistic multi-tasking. This means that the wordset should be usable on a machine with some kbytes of memory, not megabytes

Round-robin versus pre-emptive scheduling

- One of the surprising things (well, it surprised me!) was the realization that we need to say hardly anything about the differences between round-robin and pre-emptive schedulers
  - We make no guarantees about forward progress (doing so in a portable standard in a meaningful way is almost impossible) so it's not necessary to discriminate between these
  - Non-normative language must point out that on some systems you need to PAUSE or perform I/O from time to time, but that's all
  - Programs written with this wordset will work well with either type of scheduler

Memory ordering

- We have to say something about what happens when more one task accesses the same memory at the same time
- Real systems have some surprising behaviours when you do this. These include, but are not limited to
  - *Word tearing*, where a fetch sees a partial update of a multibyte word
  - Memory updates to different cells appear in different orders to different tasks
  - Memory reads can appear to go backwards in time, so that a counter is not monotonic
  - Memory can temporarily have unexpected values.
  - Many other things

Memory ordering: SC-DRF

Or, “sequentially consistent / data race free.”

- I believe that the best memory ordering model for Forth is SC-DRF.
  - The best reference for this is Hans Boehm, *Foundations of the C++ Concurrency Memory Model*, www.hpl.hp.com/techreports/2008/HPL-2008-56.pdf
  - Hans Boehm: “IMHO, the closest we have [to a language-independent memory model] that is actually solid and understandable is the basic DRF model, with undefined semantics for data races.”

Memory ordering: SC-DRF

Or, “sequentially consistent / data race free.”

- A *data race* is defined as a concurrent (non-atomic) access to shared memory
- SC-DRF allows tasks to use relaxed memory ordering locally, but requires them to use SC atomic operations when communicating with other tasks
- We give *no* semantics to programs with data races. The hardware might do all manner of things. We don't have to care: a data race might be benign on some hardware, but it won't be portable
- This isn't the dreaded nasal daemons: we only have to warn that tasks may observe misordering, word tearing, apparent loss of causality, and so on

Memory ordering: SC-DRF

Or, “sequentially consistent / data race free.”

- Sequential consistency, defined by Lamport, is the most intuitive model. Memory operations appear to occur in a single total order (i.e., atomically); further, within this total order, the memory operations of a thread appear in the program order for that thread
- We could define all Forth memory operations to be SC, but this would seriously restrict many compiler and hardware optimizations
- The best route is to allow tasks to use relaxed memory ordering locally, but require them to use SC atomic operations when communicating with other tasks

Memory ordering: SC-DRF

Or, “sequentially consistent / data race free.”

- A program which has no data races can be proved equivalent to a program in which every fetch and store are SC, i.e. they appear to all threads to happen in the same order
- This is easy for programmers to understand and it is reasonably easy to specify
- Other weaker memory models exist, but such mixed memory models become far more complicated and unintuitive

Creating a task

TASK <taskname> [polyFORTH]

Define a task. Invoking taskname returns the address of the task's Task Control Block (TCB).

/TASK (- n) [new]

n is the size of a Task Control block. [This word allows arrays of tasks to be created without having to name each one.]

CONSTRUCT (addr --) [polyFORTH]

Instantiate the task whose TCB is at addr. This creates the TCB and and possibly links the task into the round robin. After this, user variables may be initialized before the task is started

Starting a task

ACTIVATE (xt addr –) [polyFORTH]

Start the task at addr asynchronously executing the word whose execution token is xt. [This differs from Forth, Inc. practice, which uses the “word with an exit in the middle” technique of DOES>.]

What should we say about a task which reaches the end of this word, i.e. it hits the EXIT ? Traditionally, Forth systems would crash, and in order to prevent that you'd have to end an activation with

```
BEGIN STOP AGAIN
```

IMO, we'd be better saying that the task terminates

USER variables

A programmer may define words to access variables, with private versions of these variables in each task (such variables are called “user variables”).

USER (n1 --) [polyFORTH]

Define a user variable at offset n1 in the user area.

+USER (n1 n2 -- n3) [polyFORTH]

Define a user variable at offset n1 in the user area, and increment the offset by the size n2 to give a new offset n3.

#USER (– n) [polyFORTH]

Return the number of bytes currently allocated in a user area. This is the offset for the next user variable when this word is used to start a sequence of +USER definitions intended to add to previously defined user variables.

USER variables

A programmer may define words to access variables, with private versions of these variables in each task (such variables are called “user variables”).

HIS (addr1 n -- addr2) [polyFORTH]

Given a task address addr1 and user variable offset n, returns the address of the ref- erenced user variable in that task's user area. Usage:

<task-name> <user-variable-name> HIS

- This is very useful for initializing user variables before a task runs

STOP and **AWAKEN**

- These have been present in some form since the earliest days of Forth
  - **STOP** blocks the current task unless or until **AWAKEN** has been issued
  - Calls to **AWAKEN** are not “counted”, so multiple **AWAKEN**s before a **STOP** only unblock a single **STOP**
  - A task invoking **STOP** might return immediately because of a "leftover" **AWAKEN** from a previous usage. However, in the absence of an **AWAKEN**, its next invocation will block
  - **STOP** is the most OS-independent low-level blocking primitive I know of: it is a leaky one-bit semaphore
  - **STOP** and **AWAKEN** can fairly easily be used to create locks, blocking queues, and so on
  - **STOP** and **AWAKEN** correspond to BSD UNIX's `_lwp_park(2)` and `_lwp_unpark(2)`

Atomic operations

All of these are data race free

`ATOMIC@ ( a-addr -- x )[new]`

Atomically load x from a-addr. The load is sequentially consistent. Equivalent to C11's atomic_load().

`ATOMIC!( x a-addr -- )[new]`

Atomically store x at a-addr. The store is sequentially consistent. Equivalent to C11's atomic_store().

- These words are part of the total order so can be used for synchronization between threads

Atomic operations

All of these are data race free

`ATOMIC-XCHG` (x1 a-addr – x2) [new]

Atomically replace the value at a-addr with x1. x2 is the value previously at a-addr. This is an atomic read-modify-write operation. Equivalent to C11's atomic_exchange().

`ATOMIC-CAS` (expected desired a-addr – prev) [new]

Atomically compare the value at a-addr with expected, and if equal, replace the value at a-addr with desired. prev is the value at a-addr immediately before this operation. This is an atomic read-modify-write operation. Equivalent to C11's atomic_compare_exchange_strong().

- These words are part of the total order so can be used for synchronization between tasks

GET and RELEASE

MutExes provide mutual exclusion

`MUTEX-INIT` (addr) [new]

Initialize a mutex. Set its state to released.

[In polyFORTH, this was just `0 addr ! .`]

`/MUTEX` (– n) [new]

n is the number of bytes in a mutex.

[In polyFORTH, a mutex was a simple variable.]

GET and RELEASE

MutExes provide mutual exclusion

GET (addr --) [polyFORTH]

Obtain control of the mutex at addr. If the mutex is owned by another task, the task executing GET will wait until the mutex is available.

[In a round-robin scheduler, this word executes PAUSE before attempting to acquire the mutex.]

RELEASE (addr –) [new]

Relinquish the mutex at addr

- These words are part of the total order

And also...

PAUSE (–) [polyFORTH]

Causes the execuiting task temporarily to relinquish the CPU.

- In a system which uses round-robin sheduling, this can be used to allow other tasks to run
- However, this isn't usually needed because I/O causes a task to block. All words which do I/O should, unless they are extremely high priority, execute PAUSE