why another Thread Model for Forth ?

Alvaro G. S. Barcellos,
agsb at github,
version 1.0, 2024.09

why another thread model ?

I rediscovered Forth in 2019, with the Forth2020 forum at Facebook.

Since the pandemic reclusion, I got a opportunity for review the native code implementation of diverse Forths, to learn what happens from a insider view.

My focus was on the PDP-11, 6502 and X86 Forths, to create a simple and minimalist Forth for use with the ATMega328 MCU and then also port to 6502 and RISC-V.

By the way, Forth uses thread code to be independent virtual machine.

the question.

A little ATMega328, have 32k flash and 2k ram.

It is a 16-bit MCU, then 32k is really 16k code space.

And Forth dictionary grows fast. How reduce it ?

By split the headers and the code into separate memory segments, then you can delete the headers when do not needed then. *GOOD!*

And about the code, *what can be take off* ?

Only native code of primitive words is really executed by CPU.

All references of compound words are just “pushed into” and “pulled from” a FIFO stack.

Thread Code

Where does Thread Code comes from ?

the ISA of PDP-11 shows the metaphor.

The Jump to sub-routine (JSR) and Return from sub-routine (RTS)

```
        JSR Ri, Src      // Jump into subroutine
            -(SP) ← Ri; Ri ← +PC; PC ← Src;

        RTS Ri           // Return from subroutine
            PC ← Ri; Ri ← (SP)+;
```

Uses 3 registers, the program counter PC(R7), the stack pointer SP(R6) and the instruction pointer Ri(R5).

decompose the JSR/RTS

```
push and pull  // same, only use SP, as many CPUs ISA:

           CALL Src
                -(SP) ← Ri; Ri ← +PC; PC ← Src;
           RETURN
                PC ← Ri; Ri ← (SP)+;

link and jump  // same, do not use SP, in RISC-V ISA:

           JAL Ri, Src
                -(SP) ← Ri; Ri ← +PC; PC ← Src;
           JR Ri
                PC ← Ri; Ri ← (SP)+;
```

question

The primitive words does “link and jump” and compound words uses “call and return”.

A “link and jump” is done using the “instruction and word” pointers.

A “call and return” by “docol e semis” routines. Both using the return stack and made around the “next” inner interpreter.

In Direct Thread Code and Indirect Thread Code, all compound words must have (some sort of) a ***docol*** at init and a ***semis*** at ends.

What waste of resources !

question is

What waste of resources.

If is a primitive word just jump to it, then jump to next.

If is a compound word, must jump to it, then jump to *docol*, *to push the following reference*, then jump to *next*, to process the compound words inside it, until appears a primitive word or ends at *semis*, *to pull the reference to follow*, then jump to next.

That takes time from processor and space from memory.

Then the question is *how recognize primitive words before take jumps.*

(next, docol and semis are primitives).

solution

Alternatives, using the reference (16bit-address) of the words.

1. Create a table of tokens and map some references to it. Eg any reference with MSB equal 0x0 is a primitive and the LSB is a index to a table of routines. *(usually called Token-Thread-Code. It didn't reduce at all)*.
2. Mark all primitives by starting with NULL (0x0000) then test and jumps to follow address. *Does less jumps but uses same amount memory as using `docol at first`. (first version of Minimal-Thread, it partially reduces)*
3. Group all primitives words down a memory limit, then jumps only when the reference is bellow. *Less jumps and less use of memory. (Minimal-Thread, it reduces everything possible)*

simplify

How test and tune without waste time in write-erase flash cycles and change components inside a proto-boards?

Creating a minimal Forth for a 6502, using the Minimal Thread Code (MTC) model and parts and ideas of milliforth, a revised version sectorForth for x86.

All tests are done with a Linux/SWL x86 notebook, by using CA65, as compiler, and run6502, as emulator.

Next step will be translate and expand the MTC with all core Forth as primitives, for use in ATmega328.

In August 2024, it was shown that MTC is a reduced version of the internal Tachyon Forth interpreter. I'm glad to have come up with the same solution by myself.

thread code models

STC: header | call word | sequence of “call words” | jmp EXIT

ITC: header | DOCOL | list of references to words | EXIT

DTC: header | call DOCOL | list of references to words | EXIT

TTC: header | DOCOL | list of references and tokens | EXIT

MTC: header | list of references to words | EXIT

STC subroutine thread code, ITC indirect thread code

DTC direct thread code, TTC token thread code

MTC minimal thread code

That’s all, folks.