Technical Notes

Segment-Threaded Code

Robert Berkey
2334 Dumbarton Ave.
Palo Alto, CA 94303

Abstract

An 83 Standard Forth was implemented for the 8086 in which the threaded code is maintained outside the 64k Forth address space by using segments for the compilation address. Execution and compilation speeds are equivalent to a comparable 64k system.

Introduction

The Forth-83 Standard does not require that the threaded code, headers, constants, stacks, etc., reside in the Forth address space. Because the 8086 conveniently addresses beyond the 64k Forth address space, it is possible to offload the largest of these data structures. The result is a two-part dictionary structure, described here as the inner space and the outer space. As implemented, the inner space contains the stacks, user tables, block buffers, editor buffers, dictionary data fields, and code definitions. The outer space contains headers, colon definitions, constants, and user-variable offsets.

This allows one to boot a FORTH-83 system, type a command, and see

```
Inner space     53248 bytes free
Outer space    499172 bytes free
```

This system was implemented on an 8086 and became operational in the fall of 1982.

Segment Threading

The Intel 8086 architecture provides for as many as 65536 segments. Segments may be as large as 65536 bytes and segments can overlap in absolute memory, though their origins must be separated by at least sixteen bytes of memory. 65536 segments times sixteen bytes per segment equals a total of one megabyte of 8086 memory directly-accessible through segmentation.

The FORTH-83 "compilation address" is a misnomer because this value is not necessarily an address. To quote from FORTH-83, p. 4, "address, compilation. The numerical value compiled for a Forth word definition which identifies that definition." The system implemented on the 8086 uses segment values for the compilation address. An Intel salesman suggested that, in thinking of 8086 segments, a programmer should think in terms of modular parts of the code. The implemented system treats each Forth word as one modular component. Each Forth word has its own segment, which can be up to 64k in length. This technique is here called "segment threading".

A starting point for thinking of using extended 8086 memory is to look at an implementation in which each compilation address is 32 bits, consisting of an offset and a segment. This approach would double the size of the threaded code, and increase the overall size of a system by perhaps 80%. In contrast, with segment threading the threaded code is no larger; however, word headers

must be aligned on segment boundaries. The resultant space cost is seven bytes per header and an overall 30% larger system. Part of this increase in size is for extra memory-access operators needed to manage the extended memory.

This particular implementation was optimized for execution speed and compilation speed on the 8086. An in-line NEXT of nine bytes was the fastest found. As implemented on the 8086, execution cycles and cycles needed for instruction and data fetches on the bus are roughly balanced. Because the 8088 bus interface unit operates at only half the speed of the 8086, this same implementation on the 8088 would run at only about half speed, and noticeably different coding approaches would probably be required for optimum performance with the 8088.

Header Structure

Each Forth word resides within one segment; the header is maintained at the low end of this segment. Each count byte is located at offset 31 within its segment. The last byte of the name field is at offset 30, and if the name were the maximum of 31 characters, its first byte would be at offset zero. For shorter names there will be unused bytes at the bottom end of the segment, and these will physically overlap with bytes from the previous word's segment. In absolute memory from zero to fifteen bytes per header will be used only for segment alignment. The actual average of unused space was 7.03 bytes per header. The count byte, at offset 31, contains a pure number with a possible smudge bit.

Offset 32 contains the dictionary link to a previous segment with a word.

Offset 34 contains a VIEW field containing the absolute block number of source. The absolute block number was preferred to the relative block number because of the use of a fixed disk in the installation.

Offset 36 contains the precedence value, either −1 or 1 as returned by FIND in the standard. In practice, bits 1 through 14 are not used by the system and are available for application uses.

Offset 38 contains the pointer into the 64k inner space; this pointer points to the parameter field of the inner space. This pointer is needed in all words so that FORGET can reduce inner space as well as outer space. This pointer is also used by CREATE >BODY DOES> and CODE. The parameter field of the inner space contains the data field of VARIABLEs, ALLOTed dictionary space, and machine code sequences.

Offset 40 contains the code field. All code is assembled in the inner space, so the code field contains an offset into the inner space. Consolidating the code in the inner space conferred a significant speed benefit for this particular implementation, because the memory in the inner space of this system was faster than in the outer space, and instruction fetches represent the majority of memory accesses. Additionally, no cycles are used to modify the code segment register.

This is the end of the header. Offset 42, although contiguous with the header, marks the beginning of the part of the parameter field in the outer space. CONSTANTs and USER variables keep values in this part of the parameter field (the outer space part), but more important, the threaded code, that is, the data structure of the colon definitions, is maintained in this part of the parameter field.

Return Stack

The return stack contains two entries for each return address: the compilation address (i.e., segment) and the offset. This is useful for UNRAVEL because it is possible to print the name of the word whose execution has been suspended as well as the word that would be next executed.

Transportability

The transporting of conventional indirect threaded code has been easier than expected. This is because this system uses a form of indirect-threading and contains analogues for equivalent

conventional indirect threaded structures. A case-statement structure, at first deemed as too system dependent to be easy to transport, actually required only a few simple twiddles to bring up.

Limitations

The only structure not known to have trivial analogues is the multiple code field. There has been no need to implement this, but thought experiment suggests the use of adjacent segments for each additional code field, using sixteen bytes in the process.

General Overall Criticisms of this System

(1) Non-conventional

The system is non-conventional. This means that a programmer off the street needs some study to understand the access to the outer space. Of course, because this is a standard implementation, delving into the implementation's internal details can be indefinitely postponed.

(2) Multiplication of memory access operators.

`L@ L! LC@ LC! L,` `LC,` and `LCMOVE` seemed reasonable enough. Even `LDUMP` seemed to have its reasonable place. However, as use of the system progressed `LCREATE LFILL` and `LERASE` were added. With RAM so cheap, we used a lot of it and ran out of inner space. This recalls one of the laws of programming, "Memory usage expands to the available limits." The conclusion drawn is that multiple memory spaces multiply the operators needed to access memory, multiply the opportunities to run out of memory, and complicate the management of memory.

(3) 80286

Segment registers can be set in as few as two cycles on the 8086. Later processors being produced by Intel require a minimum of eighteen cycles to effect the same operation. As `NEXT` sets one segment register and `DOCOLON` sets two, segment threading on these processors in their non-8086 mode is problematic.

Related Comment

Segment-threaded code (SgTC) is not limited to implementing the standard; it is also a candidate threading system for a 32-bit Forth system.

Conclusion

As implemented this segment-threaded system was slightly faster (5%) in both compilation speed and execution speed than the traditional system it supplanted, and also satisfied the Forth-83 Standard. In discussing Forth threading techniques, segment-threaded code (SgTC) can be considered with indirect-threaded code (ITC), subroutine-threaded code (STC), direct-threaded code (DTC), and token-threaded code (TTC).

Reference

FORTH-83 Standard. Forth Standards Team, 1983.

Manuscript received August 1984.