Design and Implementation of a Multi-Task Scheduling Algorithm for Embedded Forth Virtual Machine Architecture: Postprint

Authors: Dai Hongbing, Zhou Yonglu, An Hongping, Huang Zhongjian

Date: 2018-05-20T00:00:00+00:00

Abstract

In response to the increasingly stringent practical demands on operating systems in embedded application domains regarding reconstruction, extension, portability, interaction, security, and efficiency, as well as the inherent characteristics of Forth systems, this paper employs Forth virtual machine technology to explore key technologies for embedded operating systems based on the Forth virtual machine architecture, and proposes an efficient and compact scheduling algorithm for embedded multitasking operating systems based on the Forth virtual machine architecture that possesses excellent extensibility and portability characteristics. This algorithm adopts a cooperative multitasking scheduling mechanism synchronized by Forth virtual machine instructions, which shortens task switching time and simplifies context switching operations to only requiring the preservation of the data stack pointer. Experimental results demonstrate that the multitasking scheduling algorithm based on the Forth virtual machine architecture leverages the inherent characteristics of Forth systems, improves efficiency for specific applications, and is suitable for resource-constrained embedded environments.

Full Text

Preamble

Design and Implementation of a Multitask Scheduling Algorithm for Embedded Forth Virtual Machine Architecture

Dai Hongbing, Zhou Yonglu†, An Hongping, Huang Zhongjian

(Digital Media Technology Key Laboratory of Universities in Yunnan, School of Information Science & Engineering, Yunnan University, Kunming 650223, China)

Abstract: In response to the increasingly stringent demands for reconstruction, expansion, portability, interaction, security, and efficiency in embedded operating systems, and leveraging the inherent characteristics of Forth systems, this paper explores key technologies for embedded operating systems based on Forth virtual machine architecture using Forth virtual machine technology. We propose an efficient and streamlined embedded multitask operating system scheduling algorithm built upon Forth virtual machine architecture that exhibits excellent extensibility and portability. The algorithm employs a cooperative multitask scheduling mechanism synchronized with Forth virtual machine instructions, which reduces task switching time and simplifies context switching operations to merely saving the data stack pointer.

Experimental results demonstrate that the multitask scheduling algorithm based on Forth virtual machine architecture harnesses the inherent characteristics of Forth systems, improves efficiency for specific applications, and is well-suited for resource-constrained embedded environments.

Keywords: Forth virtual machine; multitask; scheduling algorithm

0 Introduction

Forth is inherently a process control language and rapid development environment, not merely a programming tool. It possesses strong interactivity, configurability, portability, and self-extensibility capabilities, along with efficient code generation, and can even rapidly construct a real-time multitask operating system. The design philosophy of the Forth virtual machine (FVM) has gradually evolved into a unique embedded multitask operating system based on the Forth virtual machine architecture (FVMOS).

The choice of multitask scheduling strategy for FVMOS has long been a contentious issue within the Forth community, to the extent that none of the subsequently developed standards—including FIG-Forth, Forth-79, Forth-83, ANSI X3.215-1994, ISO/IEC 15145:1997, and FORTH-2012—address this topic. Unlike multitask operating systems built directly on CPU processor architecture, FVMOS implementation has unique characteristics. Introducing preemptive scheduling would inevitably compromise the virtual machine' s hardware abstraction and sacrifice inherent properties such as reconstruction, extensibility, and portability. Stand Alone Forth88 employs preemptive scheduling with three task types (concurrent—though merely a single terminal task—timer, and interrupt), but this real-time guarantee comes at the cost of dozens of complex CPU context save/restore operations and the loss of reconstruction, extensibility, and portability features. Moreover, if terminal tasks exist, this interrupt-driven forced scheduling severely disrupts the online interactive process unique to Forth systems. Is there a scheduling strategy that can both leverage FVM characteristics and maintain system real-time performance?

Analysis of FVM instruction execution reveals that if scheduling moments can be predetermined and precisely controlled at each Forth instruction boundary—

synchronizing with the FVM "heartbeat" —this dilemma can be resolved, providing higher-performance solutions for many specific embedded applications. Consequently, FVMOS multitask scheduling strategies have become a research hotspot in the Forth domain, with Forth Inc.' s SwiftOS being the most representative. SwiftOS adopts a non-preemptive strategy, supporting inter-process synchronization through semaphores and global variables, and employs Cross-Target Link for various terminal debugging interfaces. In handheld device application experiments, SwiftOS can place the device in sleep mode within one cycle of the multitask scheduling idle loop, saving 70% power consumption compared to previous unchangeable binary real-time kernels. Along with commercial SwiftOS, cooperative scheduling algorithms (Round-Robin) based on FVM have gradually become mainstream in Forth systems.

Driven by the IoT wave and facing increasingly complex embedded application requirements, along with pressing research challenges in embedded operating systems such as reconstruction, portability, trustworthiness, multi-core, and many-core architectures, FVMOS has regained attention. Compared with traditional operating systems that directly manage CPU resources, FVMOS represents a novel operating system with distinct differences in architecture, principle mechanisms, and scheduling strategies. Its real-time performance remains at a "coarse-grained" level, lacking necessary theoretical research and guidance. Building upon mainstream FVMOS multitask scheduling strategy research, this paper aims to improve real-time performance through algorithm optimization while preserving inherent characteristics such as reconstruction, extensibility, portability, and interaction. We fully exploit the uniqueness of the stack machine architecture and propose an efficient and streamlined embedded multitask operating system scheduling strategy based on Forth virtual machine architecture.

1 FVMOS System Structure

[Figure 1: see original paper] FVMOS System Structure

The FVMOS system structure is shown in Figure 1. Flash stores Forth code running on the FVM, with the FBS (Forth Basic System) 底层 comprising Code definitions and upper layers containing high-level definitions, upon which the FVMOS is built. From the FVMOS, multiple user tasks subordinate to stask can be created. In RAM, several user variable areas corresponding to user tasks consist of task control blocks (TCB), return stacks (RS), data stacks (DS), and other components, along with text input buffers (TIB), other user variable areas, and ordinary variable areas. Logically, both FBS and FVMOS can perform RAM operations (thin arrows), but physically, system access operations are encapsulated within the 底层 Code definitions of FBS, with actual operations completed through these categorized access instructions (thick arrows).

Unlike preemptive Stand Alone Forth88, cooperative FVMOS supports three task types: terminal tasks, background tasks, and interrupt tasks. Although

the TCB contents differ among the three task types, they are all connected to the multitask circular linked list (dashed line on the right).

2 Task Control Block Based on FVM

The cooperative scheduling of FVMOS is based on the virtual machine and supports three task types: terminal, background, and interrupt tasks. Under the aforementioned memory management approach, context saving only requires pushing the current return stack pointer RP onto the data stack and saving the current data stack pointer SP to the task' s user variable area. Context restoration only requires restoring SP from the task' s user variable area and storing the top-of-stack value into the RP pointer.

The FVMOS task control block (TCB) is a special area within the user variable area related to multitask scheduling. Its structure is shown in Table 1 . Each item in the TCB is defined through the user variable USER. The basic items (first six) are essential for every task, while additional items are specifically designed for terminal tasks. The additional items (I/O driver) store the I/O driver vectors for the terminal task, which are related to the specific terminal device connected. The additional items (interpreter operations) store operation vectors related to the terminal task' s text interpreter and state. It should be noted that the TCB does not retain the return stack pointer; instead, the current return stack pointer is placed at the top of the data stack.

Table 1: TCB Based on FVM

No.	Offset	Name	Description
1	0	STATUS	Task status xt, UP pointer
2	1	FOLLOWER	Next task TCB address
3	2	rp0	Return stack bottom pointer
4	3	sp0	Data stack bottom pointer
5	4	sp	Data stack top pointer (TOS)
6	5	CATCHER	Exception handler
7	6	···	Additional items (I/O driver)
8	7	···	···(Interpreter operations)

3 Task Scheduling Algorithm and Implementation Based on FVM

This section focuses on the cooperative multitask scheduling algorithm based on FVM. Given the characteristics of the Forth language and Forth systems, the algorithm description adopts the Forth2012 standard.

3.1 Task Scheduling

Typically, round-robin scheduling algorithms save task status in the STATUS unit of the TCB and cycle through searching for WAKE tasks during each scheduling event. However, in FVM-based systems, by introducing Forth vector words PASS and WAKE, the scheduling algorithm can be further optimized to eliminate status comparison and loop jumps. In this case, STATUS in the TCB no longer stores task status but instead stores replaceable PASS and WAKE vector words. The round-robin scheduling algorithm is as follows:

```
: MULTITASK-SCHEDULE
  SP @              \ Get current data stack pointer
  OVER !            \ Save to current task's tcb0[sp]
  FOLLOWER          \ Get tcb0[follower] address
  @                 \ Get next task's tcb1
  DUP
  8 + @             \ Get next task's status1
  I-CELL+           \ Get pfa of pass or wake process in status1 of next task
  >R ;              \ Push pfa of pass or wake in next task's status1 onto return stack
```

After >R, the FVM begins jumping to execute PASS or WAKE. When exiting MULTITASK-SCHEDULE, the FVM returns to the first word following PAUSE.

3.2 PASS and WAKE Processes

If PASS, rp0 and tcb1 remain on the data stack. The PASS process continuously loops, skipping tasks in PASS status until finding a WAKE task. The process is shown in Figure 2 [Figure 2: see original paper]. The PASS process definition algorithm is as follows:

```
:NONAME          \ rp0 tcb1 -- n: Number of PASS tasks in multitask circular list
  CELL+          \ Get address storing nth task's followern
  @              \ Get tcb n+1 of (n+1)th task
  DUP
  @              \ Get status n+1 (xts) of (n+1)th task
  I-CELL+        \ Get pfa of PASS or WAKE process in status n+1 of (n+1)th task
  >R ;           \ Push pfa of PASS or WAKE in status n+1 of (n+1)th task onto return stack
CONSTANT PASS    \ Store xt of nameless definition in PASS constant, PASS executes above nam
```

If WAKE, rp0 and tcbm remain on the data stack. The WAKE process restores the data stack and return stack of the task in WAKE status and implements jumping through the FVM. The WAKE process definition algorithm is as follows:

```
:NONAME          \ rp0 tcbm -- m: First WAKE task found during scanning multitask circular l
  UP!            \ Point user variable area pointer to tcbm of mth task (WAKE status)
  SP @           \ Get data stack pointer spm of mth task. Each task's data stack top stores
  @              \ Get spm of mth task
```

```
  SP!            \ Activate data stack of mth task
  RP! ;          \ Activate return stack of mth task
CONSTANT WAKE    \ Store xt of nameless definition in WAKE constant, WAKE executes above nam
```

3.3 Task Control

The multitask scheduler PAUSE can be switched on and off through SINGLE and MULTI definitions. SINGLE sets the multitask scheduler PAUSE to a no-operation, disabling multitask scheduling and retaining only one terminal task. MULTI sets PAUSE to point to MULTITASK-SCHEDULE, enabling multitask scheduling. The SINGLE, MULTI definitions, and task state control algorithms are as follows:

```
: SINGLE
  ['] NOOP IS PAUSE ;    \ Set PAUSE to no-operation, retain only one terminal task

: MULTI
  ['] MULTITASK-SCHEDULE IS PAUSE ;  \ Set PAUSE to MULTITASK-SCHEDULE, start multitask sche

: TASK-STOP
  PASS STATUS ! PAUSE ;  \ Enter multitask scheduling, stop current task immediately

: TASK-SLEEP
  PASS SWAP ! ;          \ Put tcb task to sleep in next multitask cycle

: TASK-WAKE
  WAKE SWAP ! ;          \ Wake tcb task in next multitask cycle
```

Before starting the multitask system, after task creation and initialization, each task must be defined. ACTIVATE sets the Forth commands following it as the task body for the task pointed to by the current tcb. Its definition algorithm is as follows:

```
\ tcb --
: ACTIVATE
  DUP
  6 + @ CELL-          \ Get data stack pointer where tcb task prepares to push first data
  OVER
  4 + @ CELL-          \ Get return stack pointer where tcb task prepares to push first data
  R> OVER I-CELL+ !    \ Push breakpoint after ACTIVATE onto return stack of tcb's correspon
  OVER !               \ Push rp onto data stack, preparing for WAKE process
  OVER 8 + !           \ Save data stack pointer to tcb[sp]
  TASK-WAKE ;
```

Taking SCAN defined in the main task as an example (Figure 3 [Figure 3: see original paper]), when the main task executes, ACTIVATE defines the following BEGIN···AGAIN loop as the task body for the task pointed to by tcb, rather than executing this loop. Only when the task pointed to by tcb (the slave

task) executes will it run this loop (the task body). Thus, the SCAN definition is implicitly decomposed into two parts: main task execution and slave task execution.

3.4 Background Tasks

Background tasks are suitable for scenarios requiring no continuous serial I/O, no terminal connection, and even no interpreter/compiler. Their task creation, initialization, and control are similar to TASK, TASK-INIT, ACTIVATE, and ADD-NEXTTASK processes.

3.5 Terminal Tasks

Forth is a unique "compilation/interpretation" dual-mode system supporting terminal interactive operations, facilitating online development, debugging, updating, and modification. Therefore, early minicomputer Forth systems (such as PDP-11 Stand Alone Forth-11) inherited the capability to support multiple terminal users and multiple terminal tasks. Even in resource-constrained embedded systems, Forth retains this advantage, assigning different work content to each terminal task.

Compared with background tasks, terminal tasks support serial ports and interpreter operations, allowing additional operational information to be appended to each task' s user variable area. If the target system has an interpreter, the corresponding terminal task needs to establish I/O drivers and operations related to the text interpreter input buffer in the user variable area according to different serial hardware. Terminal task creation is basically the same as background tasks, both calling the TASK process. The difference is that the additionally allocated user variable area can contain appended I/O driver vectors, interpreter input stream operation vectors, and input stream buffers. Terminal task initialization, besides completing the TASK-INIT process, also requires filling or replacing execution vectors for common operations such as character, carriage return, and cursor for different terminal types. It should be noted that when the system powers on, the core system already includes a terminal task by default, and the multitask circular list initially contains only one task pointing to its own TCB.

3.6 Interrupt Tasks

Unlike preemptive hard real-time OS, to reduce overhead while maintaining performance, FVM-based multitask scheduling does not directly use interrupts for task switching but instead places task switching between Forth definitions. When an interrupt occurs, if a corresponding interrupt task exists, after handling the interrupt 事务, the interrupt service routine performs operations similar to TASK-WAKE, causing the task to start running at the nearest PAUSE (Figure 4 [Figure 4: see original paper]). This interrupt event-driven approach is weakly real-time, requiring traversal of several predictable Forth definitions

before response, but in exchange provides system simplicity and guaranteed performance. Regardless of when the interrupt occurs, the moment to start the interrupt task is predictable and can only happen where PAUSE statements exist. Under this approach, interrupt task creation and initialization are the same as background tasks, offering greater flexibility.

Generally, except for special periodic timer tasks, interrupt tasks often need to complete execution from start to finish at each interrupt occurrence (without PAUSE process) rather than resuming from breakpoints. Furthermore, to improve system real-time performance and allow multiple priority interrupt tasks with preemptive capability, improvements to the multitask organization and scheduling are needed. One solution is to construct a preemptive multitask list (interrupt task list) outside the cooperative multitask circular list, adding TCB items such as interrupt number (INT-NO) and priority (PRIORITY) based on STATUS and FOLLOWER. Simultaneously, use processes similar to PAUSE, PASS, and WAKE to search for higher-priority WAKE tasks and launch interrupt tasks through the WAKE process.

4 Experimental Evaluation

The algorithm was implemented on the Arduino embedded hardware platform using an open-source Forth virtual machine. A multitask application containing three task types (terminal, background, and interrupt) was designed. The experimental code is as follows:

```
: MS ( n -- ) PAUSE 0 ?DO 1MS LOOP ;  \ Call PAUSE every n milliseconds to wait

VARIABLE N  \ Global variable N used by TASK3 in TASK1

40 0 BACKGROUND-TASK TASK2  \ Create background task TASK2, create task header in FLASH, all

: TASK2-BODY ( -- ) BEGIN 1 M +! &10 MS AGAIN ;  \ Define TASK2 task body

40 0 INTERRUPT-TASK TASK3  \ Create interrupt task TASK3, create task header in FLASH, alloc

: TASK3-BODY ( -- ) BEGIN 1 N +! TASK-STOP AGAIN ;  \ Define TASK3 task body

: STARTTASKER ( -- )   \ Initialize and start multitask
  TASK2 BACKGROUND-INIT   \ TASK2 initialization, create TCB2 in RAM
  TASK3 INTERRUPT-INIT    \ TASK3 initialization, create TCB3 in RAM
  MULTI ;                 \ Start multitask

: TASK-TURNKEY ( -- )     \ Power-on startup vector
  APPLTURNKEY INIT STARTTASKER ;
```

In the first terminal task TASK1 (STASK), background task TASK2 and interrupt task TASK3 are defined. After multitask startup, TASK1 executes

once each time TASK2 task body calls MS, with an average interval of 10 ms. TASK2 executes when TASK1 waits for keyboard input. TASK3 remains dormant (PASS) until a corresponding interrupt occurs, changing its status to WAKE, and starts execution at the nearest PAUSE (this distance is the interrupt latency), then enters dormant status again. Experiments show stable and reliable system operation, with efficient online interaction and background processing capabilities while maintaining certain real-time processing capabilities. The cooperative nature simplifies cumbersome resource mutual exclusion mechanisms between tasks, allowing real-time tracking of changes in M, N, and each task' s TCB through TASK1 terminal task.

In time-slice round-robin scheduling systems, system response time can generally be expressed as:

$$\mathrm{SRT} = \sum_{i=0}^{N-1} (TS_i + TST)$$

where TS_i is the time slice of each task, TST is task switching time, and N is the number of concurrent tasks. If a priority-based algorithm is adopted where each task' s time slice is proportional to its priority, then:

$$TS_i = TS_{\min} + M \times P_i$$

where M is a scaling factor, $TS_{\min}$ is the minimum time slice, and P_i is the priority of each task. When P_i takes values $0 \sim N-1$ and $M = 1$, then:

$$\mathrm{SRT} = N \times TS_{\min} + \frac{N(N-1)}{2} + N \times TST$$

When $TS_i = TS_{\min}$, the algorithm degenerates to equal time-slice round-robin scheduling:

$$\mathrm{SRT} = N \times TS_{\min} + N \times TST$$

To ensure the minimum time slice keeps system overhead below 5%, let $TST/TS_{\min} = 0.05$, then:

$$\mathrm{SRT} = 1.05 \times N \times TS_{\min}$$

To ensure average system overhead below 1%, let $TST/TS = 0.01$, then:

$$\mathrm{SRT} = 1.01 \times N \times TS$$

The system response time of the algorithm described in this paper can be approximated by equation (6). The difference is that FVM-based task switching is much faster than CPU-based switching, requiring only saving or restoring the SP pointer. Compared with the concurrent task scheduling of Stand Alone Forth88, the FVM-based task switching code (machine code) size is only 1/10 of its counterpart. Therefore, with equal overhead, system response time is significantly reduced.

System response times for CPU-based priority and fixed time-slice scheduling versus FVM-based cooperative scheduling are shown in Figure 5 [Figure 5: see original paper]. The proposed FVM-based scheduling algorithm shows no significant delay in task switching and online interaction as the number of tasks increases (which can be background or interrupt task types). Furthermore, each task' s time slice size can be flexibly determined by users according to application requirements, either by adjusting task body size or by increasing/decreasing PAUSE statements within each task body to change scheduling granularity without following predetermined system settings—that is, designed by users themselves.

5 Conclusion

Current mainstream FVMOS multitask scheduling still employs cooperative schedulers (cooperative scheme, Round-Robin), meaning each task' s start time is predetermined. Although preemptive multitask systems offer strong real-time performance, introducing them into FVMOS would inevitably affect the timely response and orderly execution of online interactive terminal tasks—features that the Forth community has long been proud of and unwilling to abandon. In practice, the trade-off between the two depends on the embedded application environment. Research shows that embedded systems often have heavy CPU loads, and preemptive scheduler overhead is significantly higher than virtual machine-based cooperative schedulers. In heavily loaded systems, the simplicity and performance of cooperative schedulers outweigh those of preemptive schedulers. Although FVMOS is only a lightweight cooperative scheduler, it can still leverage Forth' s flexible and simple characteristics to respond quickly to real-time events. Reports indicate that a four-axis bomb disposal machine running a standard cooperative scheduler operated 12 tasks, with applications in other complex domains even reporting up to 400 tasks.

Unlike traditional OS multitask scheduling algorithms, FVM-based scheduling code consists entirely of Forth high-level definitions without involving any CPU details. Through FVM abstraction, it achieves hardware independence, enabling platform portability without changing a single line of code, thus naturally providing reconstruction, extensibility, and portability. Embracing the Forth philosophy— "If you' re not satisfied, modify it" —this framework and algorithm can be continuously improved to further enhance system real-time performance. For example, interrupt task context switching time could be reduced from the task body level to interrupt service routine exit, but the complexity and overhead

introduced require further research and evaluation.

References

[1] McGuire T E. Kitt peak multi-tasking FORTH-11 [J]. The Journal of Forth Application and Reseach, 1984, 2 (2): 57-67.

[2] Dai Hongbing. Design and implementation of efficient microcomputer real-time multitask operating system [J]. Journal of the Graduate School of the Chinese Academy of Sciences, 1993, 10 (3): 283-292.

[3] Moore C. colorForth [EB/OL]. (2009). https://colorforth.github.io/cf.htm.

[4] Forth Inc. SwiftX cross compilers for embedded systems applications [EB/OL]. (2016). https://www.forth.com/embedded/.

[5] Yang Xia. Research on architecture of high trusted embedded operating system [D]. Chengdu: University of Electronic Science and Technology of China, 2013.

[6] Pelc S. Programming Forth [M]. Southampton: MicroProcessor Engineering Limited, 2011: 97.

[7] Dai Hongbing, Yang Weimin, Wang Liqing, et al. Research and implementation of multi-target Forth self-generator [J]. Application Research of Computers, 2014, 31 (4): 1109-1114.

[8] Yang Weimin, Dai Hongbing, An Hongping, et al. Research on a new embedded Forth real-time operating system [J]. Journal of Yunnan University: Natural Sciences Edition, 2013 (S2): 96-103.

[9] Frenger P. Forth and AI revisited: BRAIN.FORTH [J]. ACM SIGPLAN Notices, 2004, 39 (12): 11-16.

[10] Dai Hongbing. Development of new and efficient microcomputer Forth language [J]. Journal of the Graduate School of the Chinese Academy of Sciences, 1993, 10 (1): 62-69.

[11] FORTH Inc. Featured forth applications [EB/OL]. (2009). http://www.forth.com/resources/app Notes.

[12] IntellaSys, A TPL Group Enterprise. SEAforth 40C18 scalable embedded array processor [EB/OL]. (2008). http://www.intellasys.net/templates/trial/content/SEK_40C18_DataSheet

[13] Ieee B E. IEEE standard for boot (initialization configuration) firmware: bus supplement for IEEE 896 (futurebus+) [S]. 2002: i.

[14] The Forth Interest Group. Forth compilers page [EB/OL]. (2009). http://www.forth.org/compilers.html.

[15] Miller F P. Colorforth [M]. 2010: 28.

[16] Hanna D M, Jones B, Lorenz L, et al. An embedded Forth core with floating point and branch prediction [C]// Proc of IEEE, International Midwest Symposium on Circuits and Systems. 2013: 1055-1058.

Note: Figure translations are in progress. See original paper for figures.