Postprint: Research on Real-Time Scheduling Algorithms for Embedded Forth Operating Systems

Authors: Huang Zhongjian, Dai Hongbing, Wang Lei

Date: 2018-06-19T00:00:00+00:00

Abstract

To address the lack of real-time scheduling mechanisms in current embedded Forth operating systems, this paper investigates the key technologies of multitasking scheduling in embedded operating systems based on the Forth virtual machine architecture. By employing Forth virtual machine technology, a novel interrupt task type is defined to handle real-time emergent events, and a new task scheduling algorithm is proposed to coordinate the execution of terminal tasks, background tasks, and interrupt tasks in the Forth system. Experimental results demonstrate that the improved Forth system can handle emergent events through real-time scheduling with high real-time responsiveness, making it particularly suitable for embedded environments with real-time requirements, thereby meeting the application demands of increasingly complex embedded environments for efficient operating systems and Forth technology.

Full Text

Preamble

Title: Research on Real-Time Scheduling Algorithm for Embedded Forth Operating System

Authors: Huang Zhongjian, Dai Hongbing†, Wang Lei
(Digital Media Technology Key Laboratory of Universities in Yunnan, School of Information Science & Engineering, Yunnan University, Kunming 650223, China)

Abstract:
For the problem of lacking real-time scheduling mechanism in the embedded Forth operating system at present, this paper investigated the key technologies of multitask-scheduling of embedded operating system based on Forth virtual machine architecture. On the basis of Forth virtual machine technology, this paper defined a new interrupt task type to deal with real-time emergencies and

proposed a new scheduling algorithm, which enabled the scheduling of terminal tasks, background tasks and interrupt tasks in Forth system to be successfully executed. The experimental results demonstrate that the improved Forth system can handle emergencies through real-time scheduling and has a high degree of real-time response and is especially suitable for the embedded environment which is required for real-time performance, so as to meet the increasingly complex application requirements of embedded system for efficient operating system and Forth technology.

Keywords: Forth system; multitask; real-time scheduling

Introduction

Forth was invented by Chuck Moore in 1969 in the United States [1]. It is a collection of an operating system, a set of development tools, a high-level programming language, an assembly language, and a set of software design criteria [2]. In recent years, research on Forth both domestically and internationally has mainly focused on Forth system construction and Forth processors. Since its inception, Forth has been continuously expanded and supplemented, resulting in multiple versions such as AmForth, SwiftX, and PunyForth, among which SwiftX [3] is currently the most representative version. Regarding Forth processors, Edvin et al. [4] introduced the design of a stack-based 32-bit Forth microprocessor, elaborating on its architecture and instruction design. Hanna et al. [5] proposed a 32-bit Forth kernel architecture called rForth, which includes a floating-point unit, an interrupt controller, and accessible extended memory. Ting et al. [6–11] described the design of Forth processors using VHDL on FPGA. Additionally, Chuck Moore' s GreenArrays company has integrated 144 Forth processors on its GA144 chip [12].

Just as multitask scheduling is a core function of other operating systems, Forth operating systems also include multitask scheduling as a core function. The earliest Forth operating system scheduling adopted the same approach as traditional operating systems, namely CPU-based scheduling. This method requires saving and restoring the CPU context and Forth state during task switching, as seen in early versions like Forth11 and Forth88 [13]. The problem with CPU-based multitask scheduling is its high overhead. To address this issue, virtual machine-based Forth multitask operating systems (hereinafter referred to as Forth systems) emerged, characterized by simplicity, efficiency, hardware abstraction, reconfigurability, extensibility, portability, and interactivity. These systems use stack-based scheduling with a cooperative round-robin approach, typified by ColorForth, AmForth, PunyForth, and SwiftX. Due to these advantages, virtual machine-based Forth operating systems have become a research hotspot and gradually mainstream. While virtual machine-based Forth multitask scheduling successfully solves the problem of high overhead during task switching, it introduces another issue: non-real-time multitask scheduling.

Forth systems are mainly applied in embedded fields, where the ability to handle

various emergencies in a timely manner is self-evidently significant for an operating system. However, a review of current Forth research reveals that real-time scheduling based on virtual machines remains a blank area. Therefore, by investigating key issues related to multitask scheduling mechanisms in current virtual machine-based Forth systems, this paper proposes an implementation method to improve the real-time performance of Forth system multitask scheduling.

1. Existing Cooperative Multitask Scheduling Mechanism in Forth Systems

Currently, mainstream Forth systems generally support cooperative multitask mechanisms with round-robin scheduling. These systems provide two different task types: terminal tasks and background tasks. There is only one terminal task, while users can create multiple background tasks. Both terminal and background tasks have only two states: ready and sleeping. Only tasks in the ready state can obtain CPU control during scheduling. In single-task mode, the terminal task obtains CPU control. In multitask mode, the terminal task and background tasks alternately obtain CPU control through cooperative round-robin scheduling. Terminal tasks and background tasks are organized through a circular linked list, as shown in [Figure 1: see original paper].

In Forth system multitask scheduling, *pause* is the task scheduling primitive. Its function is to first save the execution breakpoint of the currently running task, then find the next ready task in the task chain, and finally transfer CPU control to the found ready task, which then begins execution. As shown in the task chain of [Figure 1: see original paper], assuming the terminal task is running, the principle of cooperative round-robin scheduling in Forth systems is as follows: (a) The terminal task encounters the scheduling primitive *pause* during execution. In *pause*, the terminal task first saves its execution breakpoint, then finds the next ready background task Bg-taski (1 i n-1) in the task chain, and finally transfers CPU control to Bg-taski, which then begins running. (b) Bg-taski encounters *pause* during execution. In *pause*, Bg-taski first saves its execution breakpoint, then finds the next ready background task Bg-taskj (i < j n-1) in the task chain, and finally transfers CPU control to Bg-taskj, which then begins running. (c) Background task Bg-taskj encounters *pause*. If there are no more ready tasks between Bg-taskj and background task Bg-taskn-1, since the task chain is organized as a circular linked list, Bg-taskj will transfer CPU control to the terminal task when executing *pause*, and the terminal task continues execution from its last breakpoint. (d) During terminal task execution, CPU control is transferred to Bg-taski via *pause*, and Bg-taski continues from its last breakpoint. During Bg-taski execution, CPU control is transferred to Bg-taskj via *pause*, and Bg-taskj continues from its last breakpoint. This cycle continues until both Bg-taski and Bg-taskj finish execution, after which the system enters single-task mode and the terminal task obtains CPU control.

In Forth systems, tasks are the smallest execution units competing for system resources. Currently, Forth systems only have terminal tasks and background

tasks. During execution of terminal tasks and background tasks (Bg-task1~Bg-taskn-1), if an external real-time emergency requests processing, existing Forth systems lack a real-time scheduling mechanism to handle such emergencies, which severely affects Forth system real-time performance and limits Forth' s application scope in embedded environments with real-time requirements.

2. Real-Time Task Scheduling in Forth Systems

2.1. Real-Time Task Scheduling Mechanism

(1) Handling Real-Time Emergencies in Forth Systems

Building upon the cooperative round-robin scheduling of Forth systems and combining it with interrupt technology, this paper defines a new task type—the interrupt task type—to handle external real-time emergencies. Terminal tasks, background tasks, and interrupt tasks are organized through a circular linked list, as shown in [Figure 2: see original paper]. Int-task1~Int-taskn represent interrupt tasks. The task states for terminal tasks, background tasks, and interrupt tasks are only two: ready and sleeping. The process for interrupt tasks handling emergencies is: (a) An external real-time emergency generates an interrupt request, causing the currently executing task to transfer to the interrupt service routine, where the corresponding interrupt task is readied; (b) The interrupt task is scheduled to execute.

(2) Interrupt Task Scheduling

The original Forth task scheduling primitive *pause* cannot schedule interrupt tasks. According to the task chain organization shown in [Figure 2: see original paper], the basic idea of *INT-PAUSE* scheduling is: (a) Introduce critical resources *intTask* and *readyTask[]*. *intTask* is initialized to 0 and records the number of ready interrupt tasks in the current interrupt task chain. *readyTask[0]* stores the TCB of either the terminal task or a background task, with the purpose of returning to the terminal or background task where the interrupt occurred after the interrupt task completes execution, allowing the next ready task in the terminal-background task chain to run. Whenever an interrupt request readies an interrupt task, *intTask* increments by 1, and *readyTask[1]~readyTask[intTask]* store the TCB addresses of the readied interrupt tasks in the interrupt service routine. (b) If an interrupt occurs while a terminal or background task is running: (i) store the TCB address of the currently running terminal or background task in *readyTask[0]*; (ii) increment *intTask* by 1, and store the TCB address of the readied interrupt task in *readyTask[intTask]*. If an interrupt occurs while an interrupt task is running: increment *intTask* by 1, and store the TCB address of the readied interrupt task in *readyTask[intTask]*. (c) Each time the currently running task executes *INT-PAUSE* for scheduling: if $intTask > 0$, schedule the ready interrupt task TCB stored in *readyTask[intTask]* to run, then decrement *intTask* by 1; if $intTask = 0$ and $readyTask[0] = 0$, schedule the next ready task in the terminal-background task chain; if $intTask = 0$ and *readyTask[0]* 0, schedule the next ready task in the terminal-background task chain starting from the terminal or background task TCB stored in *ready-*

Task[0], then set *readyTask[0]* to 0.

According to the task chain organization in [Figure 2: see original paper], the waiting time for a ready interrupt task Int-taskk to transition from ready to running is given by:

$$T_{wait-k} = T_i + \sum_{m=k+1}^{intTask} (T_{m-exec} + T_{INT-PAUSE}) + T_{INT-PAUSE}$$

where: 1 k n, 0 i n-1. When i = 0, Ti represents the time for the terminal task to execute from the interrupt location to the scheduling primitive *INT-PAUSE*. When i 0, Ti represents the time for background task Bg-taski to execute from the interrupt location to *INT-PAUSE*. Tm-exec is the execution time for interrupt task Int-taskm to complete its execution body once, with k+1 intTask. TINT-PAUSE is the execution time of the scheduling primitive *INT-PAUSE*.

When k+1 > intTask, meaning Int-taskk is the last ready task in the interrupt task chain, the waiting time simplifies to:

$$T_{wait-k} = T_i + T_{INT-PAUSE}$$

When only one task Int-task1 is ready in the interrupt task chain, the waiting time is:

$$T_{wait-1} = T_i + T_{INT-PAUSE}$$

When all n interrupt tasks are ready in the interrupt task chain, the waiting time becomes:

$$T_{wait-k} = T_i + \sum_{m=k+1}^{n} (T_{m-exec} + T_{INT-PAUSE}) + T_{INT-PAUSE}, \quad 1 \le k \le n$$

Since where to start scheduling in the terminal or background task execution body is determined by the programmer' s manual insertion of *INT-PAUSE*, Ti is uncertain. Meanwhile, Tm also depends on how long the interrupt task execution body takes to execute.

2.2. Real-Time Scheduling Algorithm

(1) Interrupt Task Type

With the addition of the interrupt task type created in this paper, Forth systems can now provide three task types: terminal tasks, background tasks, and interrupt tasks. Interrupt tasks are specifically designed to handle external

emergencies. Each interrupt task has its own Task Control Block (TCB). The TCBs of the terminal task, all background tasks, and all interrupt tasks are located in the user area, organized as a circular linked list as shown in [Figure 2: see original paper], and scheduled through the newly defined task scheduling primitive *INT-PAUSE*, as illustrated in [Figure 3: see original paper].

The TCB of an interrupt task defines runtime information including: task status (ready or sleeping); pointer to the next TCB in the circular linked list; stack start location and current number of entries; and a flag identifying the task as an interrupt task. The structure is shown in .

(2) Interrupt Task Creation

Interrupt tasks are created using *INT-TASK:*. For example, to create an interrupt task Int-task1: `32 32 INT-TASK: Int-task1`. The first 32 indicates that Int-task1' s return stack has 32 storage units, and the second 32 indicates that its parameter stack has 32 storage units. Using the Forth2012 standard [14], the *INT-TASK:* algorithm is as follows:

```
: INT-TASK:
  <builds
    here , (1)
    -1 ,
    (2) *mintTaskkwaitimexecINTPAUSEmkTTTintTaskkT      kwaitiINTPAUSETTT  1waitiINTPAUSETTT
    [ s" /user" environment search-wordlist drop execute ] literal
    ( rstacksize ) allot here , ( dstacksize ) allot here ,
  does>
    ;
```

The storage mapping for Int-task1 is shown in [Figure 4: see original paper]. After creating interrupt task Int-task1, several steps are required before it can run: (a) Initialize Int-task1' s TCB by setting the parameter and return stack pointers to correct positions, and placing the interrupt task execution body (the code for handling emergencies) at the top of the return stack so Int-task1 can execute this code when running. (b) If Int-task1 is the first interrupt task created, build a circular interrupt task queue by making Int-task1' s TCB *follower* point to itself; if not the first, add Int-task1' s TCB to the existing interrupt task queue to form a circular queue. (c) Since external emergencies request CPU processing through interrupts, and interrupt tasks are specifically designed for handling external emergencies, the interrupt service routine must ready the interrupt task Int-task1 by storing the execution address of the word *WAKE* into the *status* storage unit of Int-task1' s TCB, enabling Int-task1 to be scheduled.

(3) Task Scheduling Primitive INT-PAUSE

As discussed above, the task scheduling primitive *INT-PAUSE* schedules terminal tasks, background tasks, and interrupt tasks. Using the Forth2012 standard, the *INT-PAUSE* algorithm is:

```
: INT-PAUSE
  ENTER_CRITICAL
```

```
  rp@ sp@ sp !
  0 intTask @ > if branch1 else branch2 then
  EXIT_CRITICAL
;
```

During *INT-PAUSE* execution, CPU interrupt response is disabled; the code between *ENTER_CRITICAL* and *EXIT_CRITICAL* is a critical section. The sequence `rp@ sp@ sp !` saves the execution breakpoint of the currently running task. If $intTask > 0$, indicating ready interrupt tasks exist, execution branches to *branch1* to schedule an interrupt task. If $intTask = 0$, indicating no ready interrupt tasks, execution branches to *branch2* to find a terminal or next ready background task.

In *branch1*, the ready interrupt task TCB address stored in *readyTask[intTask]* is assigned to *tmpTCB*, then *intTask* is decremented by 1, and finally the TCB stored in *tmpTCB* is scheduled to run. Specifically, the currently running task executes *WAKE* stored in the *status* unit of the TCB pointed to by *tmpTCB*, which transfers CPU control to the ready interrupt task, which then resumes execution from its breakpoint.

In *branch2*, if $readyTask[0] = 0$, indicating task scheduling occurred in a terminal or background task execution body, *branch3* finds the next ready task in the terminal-background task chain and transfers CPU control to it. If *readyTask[0]* 0, indicating scheduling occurred in an interrupt task execution body, *branch4* returns to the terminal or background task where the interrupt occurred, allowing the next ready task in the terminal-background task chain to run, then sets *readyTask[0]* to 0.

In *branch3*, the *status* content of the next task' s TCB in the task chain is retrieved. If it contains the execution address of *PASS* (sleeping), *PASS* is executed to continue searching until a task with *WAKE* status is found. Then *WAKE* is executed to transfer CPU control to the found ready task, which resumes execution from its breakpoint.

In *branch4*, the terminal or background task TCB address stored in *readyTask[0]* is assigned to *tmpTCB*, *readyTask[0]* is set to 0, and finally the TCB stored in *tmpTCB* is scheduled to run.

Compared with CPU-based scheduling, virtual machine-based Forth real-time operating systems only need to push the current return stack pointer *rp* onto the parameter stack and save the current parameter stack pointer *sp* to the task' s user variable area during context protection. For context restoration, they only need to restore *sp* from the task' s user variable area and store the parameter stack top value into the *rp* pointer.

3. Experimental Verification and Analysis

The specific implementation of the Forth system real-time task scheduling mechanism in this paper is based on the open-source AmForth. AmForth is a 16-bit

Forth system running on the ATmega328 chip, a high-performance, low-power AVR 8-bit microcontroller supporting interrupt processing with a 16MHz clock frequency, 32KB flash, 2KB RAM, and 1KB EEPROM. ATmega328P provides various interrupt types including external request interrupts, pin change request interrupts, timer interrupts, and serial interrupts. All interrupts have separate vectors in the interrupt vector table with priorities corresponding to their positions—lower vector addresses have higher priorities.

As analyzed in Section 2.1, according to the *INT-PAUSE* scheduling method, the number of background tasks does not affect the waiting time for a ready interrupt task Int-taskk to transition from ready to running. The waiting time depends on: (a) the time for the terminal or background task to execute from the interrupt location to *INT-PAUSE*; (b) the number of interrupt tasks and their execution time; (c) the execution time of *INT-PAUSE*.

In AmForth, a background task Bg-task1 was created. The terminal task Sys-task execution body consists of program segments , while background task Bg-task1 consists of program segments . Emergencies successively ready interrupt tasks Int-task1~Int-task5 through interrupts in program segment , with Int-task1~Int-task5 execution bodies being program segments ~, as shown in [Figure 5: see original paper].

After background task Bg-task1 finishes executing program segment , the interrupt task ready table *readyTask[]* storage status is shown in [Figure 6: see original paper].

presents the waiting times from ready to running for interrupt tasks Int-task1~Int-task5 under *INT-PAUSE* scheduling. TINT-PAUSE represents the execution time of *INT-PAUSE*, and T1 represents the time for background task Bg-task1 to execute from the interrupt location to *INT-PAUSE* in program segment . From and the *INT-PAUSE* scheduling process:

a) Interrupt task Int-task5 waiting time from ready to running: $T_{wait-5} = T_1 + T_{INT-PAUSE} = 9\,\text{ms}$

b) T5-exec represents Int-task5 execution time. Int-task4 waiting time: $T_{wait-4} = T_1 + T_{5-exec} + 2 \times T_{INT-PAUSE} = 11\,\text{ms}$

c) T4-exec represents Int-task4 execution time. Int-task3 waiting time: $T_{wait-3} = T_1 + T_{5-exec} + T_{4-exec} + 3 \times T_{INT-PAUSE} = 12\,\text{ms}$

d) T3-exec represents Int-task3 execution time. Int-task2 waiting time: $T_{wait-2} = T_1 + T_{5-exec} + T_{4-exec} + T_{3-exec} + 4 \times T_{INT-PAUSE} = 16\,\text{ms}$

e) T2-exec represents Int-task2 execution time. Int-task1 waiting time: $T_{wait-1} = T_1 + T_{5-exec} + T_{4-exec} + T_{3-exec} + T_{2-exec} + 5 \times T_{INT-PAUSE} = 18\,\text{ms}$

4. Conclusion

This paper investigated key issues related to multitask scheduling mechanisms in Forth systems. To address the lack of real-time scheduling for handling real-time emergencies in Forth systems, this work combined interrupt technology to create a new interrupt task type *INT-TASK:* for handling emergencies, and defined a new task scheduling primitive *INT-PAUSE* to schedule terminal tasks, background tasks, and interrupt tasks. Theoretically, this improves the real-time response of Forth systems to emergencies. The correctness of this theory was verified through experiments on AmForth, advancing the development of real-time multitask scheduling technology in Forth systems and providing theoretical support and implementation reference for large-scale application of virtual machine-based embedded Forth systems in real-time embedded environments.

References

[1] Wikipedia. Charles H. Moore [EB/OL]. (2018) [2018-01-01]. http://en.wikipedia.org/wiki/Charles_H._Moo
[2] Leo B. Starting Forth [M]. Hermosa Beach: FORTH Inc, 1981: 1-5.
[3] FORTH Inc. SwiftForth IDE for Windows, Linux, and macOS [EB/OL]. (2018) [2018-1-12]. https://www.forth.com/embedded/.
[4] Hjrtland E, Chen L. EP32-A 32 bit Forth micorprocessor [C]// Proc of CCECD. New York: Institute of Electrical and Electronics Engineers Inc, 2007: 518-521.
[5] Hanna D M, Jones B, Lorenz L, et al. An embedded Forth core with floating point and branch prediction [C]// Proc of the 56th International Midwest Symposium on Circuits and Systems. New York: Institute of Electrical and Electronics Engineers Inc, 2013: 1055-1058.
[6] Ting C H. VHDL Design of eP32 Microprocessor [EB/OL]. (2010) [2018-2-5]. http://www.forth.org/svfig/kk/08-2010-Ting.pdf.
[7] James B. The J1 Forth CPU [EB/OL]. (2018) [2018-1-10]. http://www.excamera.com/sphinx/fpga-j1.html.
[8] Don G. Forth Processor in VHDL [EB/OL]. (2018) [2018-1-15]. http://www.ultratechnology.com/4thvhdl.htm.
[9] John R. Using Forth as a VHDL [EB/OL]. (2018) [2018-1-15]. http://testra.com/Forth/VHDL.htm.
[10] John R. QSP16 [EB/OL]. (2018) [2018-1-12]. http://www.sandpipers.com/.
[11] Paweł G, Wojciech M Z. Tethered Forth system for FPGA applications [C]// Proc of Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2013.
[12] Chuck M. GreenArrays [EB/OL]. (2018) [2018-1-15]. http://www.greenarraychips.com/.
[13] 代红兵. 高效微机实时多任务操作系统设计与实现 [J]. 中国科学院研究生院学报, 1993, 10 (3): 283-292. (Dai Hongbing. The design and realization of efficent microcomputer operating system of real-time multitask [J]. Journal of the Graduate School of the Chines Academy of Sciences, 1993, 10 (3): 283-292.)
[14] Forth200x committee. Forth 2012 Standard [EB/OL]. (2018) [2018-3-12]. http://forth-standard.org/.