第 46 卷 第 1 期 计 算 机 工 程 2020 年 1 月
Vol. 46 No. 1 Computer Engineering January 2020

·体系结构与软件技术· 文章编号：1000-3428(2020)01-0208-08 文献标志码：A 中图分类号：TP316

嵌入式 Forth 操作系统多任务空间复用算法

梅 浩，代红兵，刘 静

（云南大学 信息学院，昆明 650000）

摘 要： 针对现有嵌入式 Forth 操作系统多任务空间无法复用且多任务管理仅支持任务创建的问题，提出一种基于 Forth 虚拟机架构的嵌入式操作系统多任务空间复用算法。将任务控制块作为空闲任务映像分区链表头结点，利用任务控制块中的链接地址变量维护系统删除的后台任务映像，且仅需修改一个用户变量指针即可实现任务映像空间的回收和重分配。实验结果表明，该算法在保证 Forth 系统稳定及其固有特性的同时，提高了 Forth 系统内存资源利用率，适用于资源有限的嵌入式环境。

关键词： Forth 虚拟机；嵌入式环境；多任务；内存管理；多任务管理；可移植性

开放科学（资源服务）标志码（OSID）：

中文引用格式： 梅浩，代红兵，刘静. 嵌入式 Forth 操作系统多任务空间复用算法[J]. 计算机工程，2020，46(1)：208-215，221.

英文引用格式： MEI Hao，DAI Hongbing，LIU Jing. Multitask space reuse algorithm in embedded Forth operating system[J]. Computer Engineering，2020，46(1)：208-215，221.

Multitask Space Reuse Algorithm in Embedded Forth Operating System

MEI Hao，DAI Hongbing，LIU Jing

（School of Information Science and Engineering，Yunnan University，Kunming 650000，China）

【Abstract】 In existing embedded Forth operating systems，multitask space cannot be reused and multitask management supports only task creation. To address the problem，this paper proposes a space reuse algorithm for multitask in embedded operating systems based on Forth Virtual Machine（FVM）. The algorithm takes the task control block as the header node of the idle task image partition list. The link address variable in the task control block is used to track the background task images deleted by the system，and both collection and redistribution of task image space can be realized by modifying only one user variable pointer. Experimental results show that the proposed algorithm improves memory resource utilization of Forth system while ensuring the stability and inherent features of Forth system. It is applicable to embedded environments with limited resources.

【Key words】 Forth Virtual Machine（FVM）；embedded environment；multitask；memory management；multitask management；portability

DOI： 10.19678/j.issn.1000-3428.0053552

0 概述

Forth 是基于堆栈、交互式且具有简单性哲学思想的计算机编程语言和环境。Forth 语言以可延伸的词典为核心，采用以两个堆栈为基础的高度模块化结构，是一种具有解释态和编译态的双态系统[1]。区别于其他程序设计语言，Forth 具有可交互、可重构、可移植、可扩展和高效精简的代码生成等特点，甚至可以根据需要快速构造出一个实时多任务操作系统[2]。Forth 语言自发明以来，先后形成了 FIG-Forth、Forth-79、Forth-83、ANS-Forth、ISO-Forth 和 FORTH-2012 标准[3]，并且已经越来越多地用于设计嵌入式软件和固件[4-7]，几乎覆盖了所有主流的嵌入式平台[8-10]。

在资源条件有限的嵌入式硬件平台上，为满足复杂应用的需求，出现了一种全新的可重构、可扩展和可在线交互的基于 Forth 虚拟机（Forth Virtual Machine，FVM）架构的嵌入式多任务操作系统（FVM Architecture-based Embedded Multitask Operating System，FVMOS）[11]。与传统操作系统架构不同[12]，

基金项目： 国家自然科学基金（61640205）。

作者简介： 梅 浩（1993—），男，硕士研究生，主研方向为嵌入式系统；代红兵（通信作者），正高级工程师；刘 静，硕士研究生。

收稿日期： 2019-01-02 **修回日期：** 2019-02-18 **E-mail：** hbdai_it@126.com

FVM 架构实现了上层 Forth 操作系统和底层硬件平台的抽象,整个系统运行在 FVM 上。与采用抢占式调度的 Forth 操作系统[2]不同,任务之间的切换需要对 CPU 调度现场和 Forth 系统状态进行保护,FVMOS 大多采用基于 Forth 虚拟机的协同调度算法[13],系统事先勾链好一个调度队列,通过基于 FVM 的调度原语 pause 实现整个调度过程全部基于 Forth 虚拟机之上,不涉及底层硬件细节[14],多任务调度现场保护只需将该任务数据堆栈栈顶地址保存在其任务控制块(Task Control Block,TCB)的数据栈指针变量 sp 中,每个任务的任务空间相互独立[15]。

在多任务内存空间管理方面,Forth 实时多任务操作系统 Forth11 的任务空间划分为逻辑上独立的 4 个区,利用页地址映射机制实现了任务逻辑地址到物理地址的映射。系统提供专门的公共资源接口(Common Resource Interface,CRI)进行任务空间分配和回收[16]。基于 X86 架构的 PC-Forth 系统提供了一个任务列表(Task List,TL)用来存放待执行任务的代码域地址(Code Field Address,CFA),任务列表常驻内存,系统创建和删除任务是在 TL 中增加或擦除一个 CFA[17]。SwiftX 采用 FVM 思想,系统内部包含一个基于 FVM 的实时多任务操作系统 SwiftOS,任务空间由 TCB 和堆栈区组成,任务空间一次性全部加载进内存才能运行。在多任务启动后,任务无法删除,只能强制阻塞任务使其不再继续调度执行[18],但是任务空间一直驻留在内存中且无法释放。

嵌入式环境由于资源的限制,处理器一般不具备复杂的内存管理单元(Memory Management Unit,MMU)[19]。现有 FVMOS 的静态分配算法严格按照地址递增的方向为任务一次性分配指定大小的内存空间,以满足嵌入式系统对任务内存分配快速、可靠和高效的现实需求[20],但其任务空间静态分配后不能对其回收复用且无法删除调度队列中的任务,造成内存资源浪费及系统不必要的开销,导致系统性能降低。另外,其只能指定位置勾链多任务调度队列,不能满足复杂的嵌入式应用需求。

为解决上述问题,本文通过对 FVMOS 的 TCB、任务内存分配和多任务调度队列等关键问题的研究,在保证 FVMOS 原有优势的前提下,提出一种多任务空间复用算法。

1　FVMOS 框架结构及多任务管理

1.1　FVMOS 框架结构

FVMOS 框架结构如图 1 所示,FVM 采用核心算法 DO_NEXT 进行 Forth 虚拟机指令与目标物理机指令的一一对应,实现了系统底层抽象。FVMOS 中词(words)是完成某个功能的一段代码或一些数据的封装,FVM 和核心词典组成了 Forth 基本系统(Forth Basic System,FBS)。操作系统上电启动运行一个终端任务实现用户交互。终端任务下可动态定义多个后台任务,最终形成一个终端任务和多个后台任务的嵌入式操作系统 FVMOS。

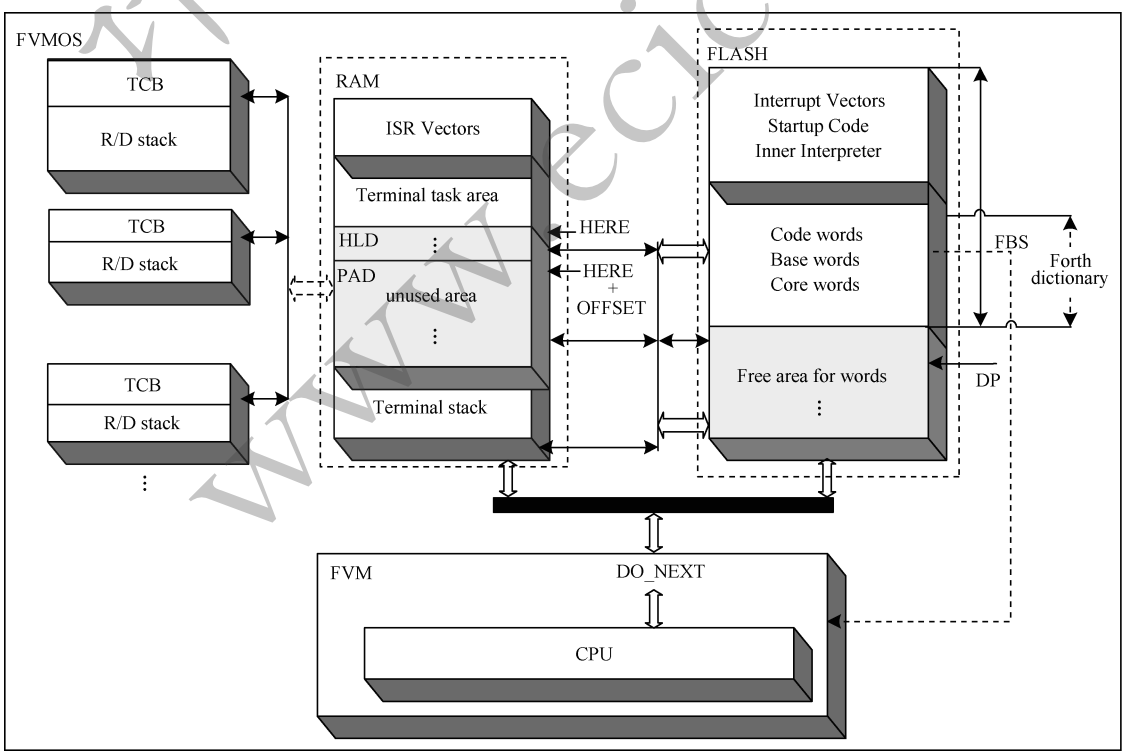

图 1　FVMOS 框架结构

Fig. 1　FVMOS framework structure

1.2 FVMOS 多任务管理

1.2.1 任务控制块

保存任务运行信息的用户变量组成了 FVMOS 的任务控制块 TCB,其结构如表 1 所示。

表 1 基于 FVM 的任务 TCB

Table 1 FVM-based task TCB

相对偏移	名称	作用描述
0	status	任务状态
2	follower	下一个任务地址
4	RP0	返回栈底指针
6	SP0	数据栈底指针
8	SP	数据栈顶
10	NFA	任务名域地址
12	HANDLER	异常处理句柄
14	BASE	数字基制
16	EMIT	输出字符
18	EMIT?	输出设备是否就绪
20	KEY	接收键盘输入
22	KEY?	键盘是否有输入
24	SOURCE	终端输入缓冲区首地址
26	>IN	终端输入字符串偏移
28	REFILL	填充输入缓冲区

每个表项占两个内存单元,用户可根据应用需要裁剪或添加 TCB 表项。终端任务较特殊,该任务连接终端设备,提供用户交互,需要在 TCB 中存放相关操作用到的信息,表 1 中的所有表项都为终端任务所需表项。后台任务不占用 I/O 设备,任务调度运行过程中只需 TCB 表项的前 5 项。

1.2.2 后台任务映像

TCB 和任务堆栈共同构成任务映像(Task Image,TI),它们在内存中连续存放。FVMOS 在建立任务时以词的方式保存任务映像在 RAM 空间中的 TCB 起始地址、返回堆栈起始地址和数据堆栈起始地址。以上信息全部保存在词的参数域,构成了任务信息块(Task Information Block,TIB)。任务映像结构如图 2 所示。

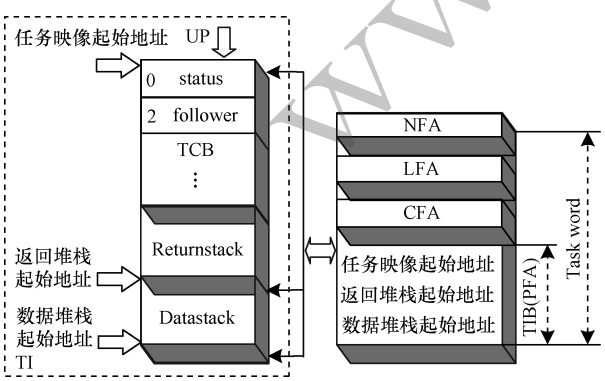

图 2 任务映像

Fig. 2 Task image

系统用户指针(User Pointer,UP)指向当前运行的任务映像首地址,任务映像各自独立,内部以 UP 为基准,采用相对偏移编址,返回堆栈(Return Stack,RS)和数据堆栈(Data Stack,DS)共同构成任务堆栈(Task Stack,TS)。堆栈向下生长,避免了任务之间互相干扰和破坏,起到了存储保护作用。

1.2.3 多任务创建

在现有 FVMOS 中,在多任务管理方面只提供了简单的多任务创建算法 tasks:。tasks:首先根据用户给定的任务名在 FLASH 中建立一个任务词,然后调用静态分配算法 allot 依次分配 TCB、RS 和 DS,并在任务词的参数域(PFA)记录相应的起始地址。

当一个任务经过创建、初始化、勾链具体执行任务体后,即可加入多任务调度队列等待调度执行。系统通过 Forth 词 onlytask 和 alsotask 实现多任务调度队列。其中,onlytask 实现了只有一个终端任务的调度队列,后台任务只能通过 alsotask 将其勾链在终端任务之后。

1.3 现有 FVMOS 存在的问题

一般嵌入式操作系统任务删除功能就是回收任务 TCB,任务代码仍驻留在内存中,只是不被再次调用[21],回收的 TCB 可以进行下次分配。

现有 FVMOS 在多任务内存空间管理方面只提供基本的静态按需分配方式,针对不需要继续运行的任务无法删除并回收利用其内存空间,造成嵌入式硬件平台的资源浪费。另外,对不需要继续运行的任务只能通过置任务为睡眠状态来防止被再次调度执行,系统在维护调度队列中可能再次调度执行任务的同时还要维护删除的任务,调度过程中会不断执行其状态原语,导致系统性能下降。在调度队列中,新建后台任务只能勾链在终端任务后,不能满足复杂应用的需求。

2 FVMOS 多任务空间复用算法及实现

针对现有 FVMOS 存在的问题,本文提出一种基于 FVM 的多任务空间复用算法 Re_allot,该算法能够可靠、快速、高效地对整个任务映像空间进行回收和重分配。同时,对调度队列勾链算法 alsotask 进行改进,改进算法能够根据实际应用需要,动态勾链其任务调度队列。

本文先提出基于 FVM 的多任务空间复用管理框架,再给出框架中空闲任务映像分区链表结构,并设计 Re_allot 算法具体的任务映像空间分配和回收算法,同时对多任务调度队列的勾链算法进行改进。

2.1 多任务空间复用管理框架

静态分配算法 allot 的每次分配都是从 HERE 处按照地址递增的方向开始分配,为不影响其他任务的运行和保持 FVMOS 的稳定,HERE 值不能更改。多任务空间复用管理框架如图 3 所示。

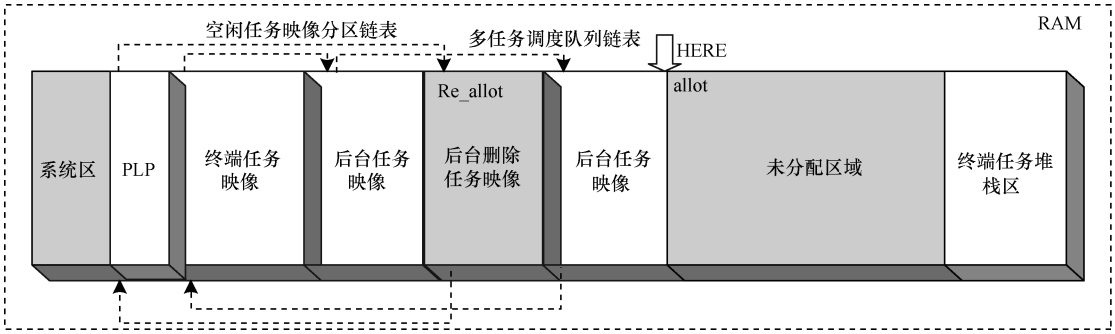

图 3　多任务空间复用管理框架

Fig. 3　Management framework of space reuse for multitask

现有系统仅提供 allot 算法利用内存未分配区域创建新任务,并通过 alsotask 算法勾链在终端任务后,形成多任务调度队列链表。在实现了 Re_allot 算法的 FVMOS 中,存在空闲任务映像分区链表和多任务调度队列链表两个链表数据结构。

2.2　空闲任务映像分区链表

空闲任务映像分区链表是一个带头结点的单向循环链表,以一个只有 TCB 控制块的任务映像 plp 作为链表头结点。plp 以词的形式定义,当有任务删除时,首先通过 FLASH 和 RAM 之间的互操作 tib > tcb 得到 plp 的 TCB 地址,然后修改 TCB 中的 follower 变量信息,把删除的任务映像添加到分区链表中。空闲任务映像分区链表结构设计过程如下:

步骤 1　入口参数全为 0,使用 task:在词典中建立一个任务名为 plp 的任务词,其对应任务映像空间只有 TCB 块。

步骤 2　建立空闲任务映像分区链表。首先初始化 plp 的任务映像空间,然后通过互操作得到 TCB 地址,根据 TCB 内变量的相对位置,获取到 follower 变量并使用 RAM 写操作,赋该值为 plp 的 TCB 首地址。

步骤 3　采用 FVMOS 独有的延迟词技术,把空闲任务映像分区链表的建立过程定义为系统的启动行为。

根据 FORTH2012 标准,算法实现如下:

```
0 0 task:plp (RS DS- )
partition_list \ 链表结构建立
plp task-init \ 初始化
plp tib > tcb dup cell + !
'partition_list' turnkey defer!
```

至此,系统上电就会存在一个只有 plp 的单向循环链表——空闲任务映像分区链表。当有任务删除时,勾链到该分区链表即可,具体过程如算法 1 所示。

算法 1　空闲任务映像分区链表勾链

输入　删除待回收的任务映像空间首地址,即 TCB 首地址

输出　删除任务被勾链到 plp 之后

步骤 1　终端任务执行 plp,通过定义 plp 时 DOES > 后的默认行为得到其 TIB,互操作获取 TCB 中的 follower 变量值,并写入输入的删除任务 TCB 对应表项中。

步骤 2　同步修改任务 plp 的 TCB 表项 follower 为输入的删除任务 TCB。

采用 FORTH2012 标准,算法实现如下:

```
partition_list_insert ( d_tcb- )
dup plp tib > tcb cell + @ swap cell + !
plp tib > tcb cell + !
```

plp 的主要作用是系统用来感知空闲任务映像分区链表的存在,不能勾链任务体投入运行。同时,待回收任务映像空间直接勾链在 plp 之后,不用遍历分区链表,算法时间花费主要集中在 follower 变量指针的修改和赋值方面,时间复杂度为 $O(1)$,整个算法执行期间最多仅需 6 个内存单元。

2.3　空闲任务映像分配

在系统进行任务空间分配时,实现了空闲任务映像分区链表之后就可采取两种方式分配,一种是使用系统原分配算法 allot,另一种是本文实现的多任务空间复用算法 Re_allot。FVMOS 后台任务必须在可接受的时间范围内运行完毕,终端任务与用户交互才不会感受到明显延迟,并由于后台任务堆栈空间分配后不能动态增加,其所执行的任务体往往都是多个 Forth 定义嵌套组合而成,很难精确计算出运行期间所需的堆栈空间大小,因此后台任务的映像空间基本控制在一个范围内。鉴于嵌入式操作系统中对内存分配的要求和 FVMOS 中多任务的特殊情况,本文采用快速高效的首次适应(First Fit,FF)算法进行空闲任务映像分区分配,算法执行流程如图 4 所示。

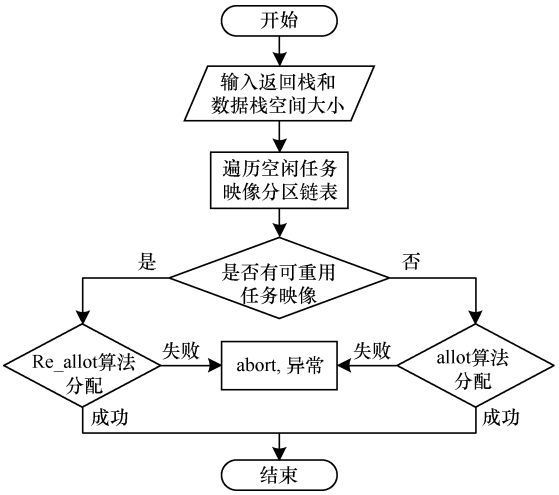

图 4 空闲任务映像分区分配算法流程

Fig. 4 Procedure of the distribution algorithm for idle task image partition

在进行空闲任务映像分配时,沿着空闲任务映像分区链表 plp 开始遍历,查找到首个大小能满足申请要求的空闲任务映像分区给用户。如果空闲任务映像分区链表中没有满足大小要求的空闲分区,则调用静态分配算法 allot 来分配任务空间,整个过程不涉及任务映像移动,分配过程中只需修改一个 follower 变量指针,具体过程如算法 2 所示。

算法 2 空闲任务映像分区分配

输入 任务所需 RS、DS 大小

输出 重利用的任务映像空间首地址

步骤 1 执行 plp,获取空闲任务映像分区链表地址并开始链表遍历。

步骤 2 通过获取 TCB 中 SP0 变量存放的数据栈起始地址,与 TCB 首地址和 TCB 大小相减,得到当前访问映像空间堆栈大小。

步骤 3 判断当前任务映像空间大小是否满足要求,若满足要求,则执行步骤 4;否则转到步骤 5。

步骤 4 从空闲任务映像分区链表中得到满足要求的任务映像空间并利用终端任务给出该映像空间地址和 ok 提示,清理算法临时数据并退出。

步骤 5 继续向下遍历空闲任务映像分区链表。

步骤 6 对空闲任务映像分区链表是否遍历完毕进行判断,若是则通过终端任务给出提示并结束算法;否则转到步骤 2 继续执行算法流程。

根据 FORTH2012 标准,算法实现如下:

```
1. Re_allot ( RS DS- )
2. + plp tib > tcb
3. begin
4. cell + @ dup dup 6 + @ swap – 10-
5. swap > r \ 返回堆栈暂存数据
6. over swap 1 + <
7. if
8. r > dup partition_list_delete @
9. – 1 abort ‘ok’
10. else
11. r > cell + @ dup plp tib > tcb =
12. until
13. ‘no suitable task TI’ 2drop
```

步骤 4 中置换出满足要求的任务映像空间对应的算法 partition_list_delete 和任务映像回收算法原理一致,在保证链表结构不被破坏的情况下修改 follower 值,逻辑上置换出将要重分配的分区,具体过程如算法 3 所示。

算法 3 partition_list_delete

输入 待置换出进行分配的任务映像空间 TCB 首地址

输出 置换出指定任务映像空间后的空闲任务映像分区链表

步骤 1 输入数据合法性判断,如果是 plp,则给出提示并终止算法。

步骤 2 对当前访问任务映像的 follower 执行 RAM 读操作@,获取该值并判断是否为需要摘取的目标任务映像,若是则执行步骤 3;否则执行步骤 4。

步骤 3 获取目标任务映像所在分区链表中的下一个任务映像 TCB 首地址,并以此值修改当前访问任务映像的 follower。plp 的 TCB 首地址压入数据堆栈,转步骤 5,提前结束算法流程。

步骤 4 继续遍历访问链表结构中的下一个任务映像空间。

步骤 5 对空闲任务映像分区链表是否遍历完毕进行判断,如果遍历完毕,即栈中存放的下一个要访问的任务映像地址是 plp,则执行步骤 6;否则程序跳转到步骤 2 继续执行。

步骤 6 丢弃算法执行过程中的临时数据并结束算法。

根据 FORTH2012 标准,算法实现如下:

```
1. partition_list_delete ( tcb- )
2. plp tib > tcb 2dup =
3. if
4. – 1 abort ‘forbidden’
5. then
6. begin
7. 2dup cell + @ =
8. if
9. dup cell + @ cell + @
10. swap cell + ! plp tib > tcb
11. else
```

```
12. cell + @
13. then
14. dup plp tib > tcb =
15. until
16. 2drop
```

2.4　任务映像回收

若多任务调度队列中某个任务的任务体执行完毕且以后不需要再次执行,则对其运行载体——任务映像进行回收,具体过程如算法 4 所示。

算法 4　任务映像回收

输入　多任务调度队列链表中待回收任务 TCB 首地址

输出　多任务调度队列链表中删除指定任务映像并回收到空闲任务映像分区链表

步骤 1　算法输入数据合法性判断,如果待删除任务映像是终端任务,则终止算法并给出禁止提示。

步骤 2　从终端任务映像开始遍历调度队列。

步骤 3　判断当前访问任务映像 follower 是不是指向待回收任务,若是则执行步骤 4;否则执行步骤 6。

步骤 4　使用待回收任务映像的 follower 值对应修改当前访问的任务映像,逻辑上从调度队列中删除待回收任务映像并把该任务映像首地址压入终端任务数据堆栈。

步骤 5　终端任务数据堆栈弹出栈顶数据作为 partition_list_insert 的输入,逻辑勾链到空闲任务映像分区链表后跳转到步骤 7。

步骤 6　继续多任务调度队列的遍历,如果队列遍历完毕则执行步骤 7;否则转到步骤 3。

步骤 7　清除算法执行过程中终端任务数据堆栈中暂存的临时数据并结束算法。

根据 FORTH2012 标准,算法实现如下:

```
1. task_delete ( d_tcb- )
2. dup status =
3. if
4. -1 abort ‘forbidden’
5. then
6. status
7. begin
8. 2dup cell + @ =
9. if
10. dup cell + @ cell + @ swap cell + !
    dup partition_list_insert status
11. else
12. cell + @
13. then
14. dup status =
15. until
16. 2drop
```

由以上分析可以得出,Re_allot 算法的整个过程都采用 FVMOS 的基本词实现,涉及的都是堆栈操作,算法效率高。算法时间复杂度为 $O(N)$,其中,N 为空闲任务映像分区链表的长度,空间复杂度为$O(1)$。

2.5　多任务调度队列

FVMOS 任务调度采用的是协同式调度算法,任务按照事先确定的调度队列顺序依次调度执行,系统不能完成实时性要求高的任务。

多任务空间复用算法的前提是能够根据需求操作多任务调度队列,系统原有的 alsotask 算法只能在一个位置勾链多任务,这样的方式不能满足复杂的嵌入式应用需求,因此设计并实现了基于 FVM 的多任务调度队列勾链算法 queue_schedule,具体过程如算法 5 所示。

算法 5　queue_schedule

输入　指定勾链位置和目标勾链任务的 TCB 首地址

输出　勾链完毕后的调度队列

步骤 1　调度队列中指定勾链位置对应的任务映像 TCB 中的 follower 值更新目标勾链任务的 follower。

步骤 2　重新修改指定勾链位置所在任务映像的 follower 值为目标勾链任务首地址。

根据 FORTH2012 标准,算法实现如下:

```
1. queue_schedule ( ftcb ntcb- )
2. \ 任务 ftcb 之后勾链 ntcb
3. 2dup swap cell + @
4. swap cell + !
5. swap cell + !
```

alsotask 算法都是采用基于 FVM 的基本词实现,整个过程只需要一次 RAM 读操作和两次 RAM 写操作,算法时间复杂度都为 $O(1)$,alsotask 需要 4 个内存单元,queue_schedule 需要 8 个内存单元。

FVMOS 以词的形式定义任务体并添加到 Forth 词典,任务体运行所需的数据放在任务映像空间的 DS,执行代码地址放在 RS,实现程序和数据分离。后台任务所执行的具体任务体代码中都要插入 pause 原语作为调度点,当任务执行到 pause 就转到调度队列中的下一个任务,同时任务体必须用 begin…again Forth 词包围。因为任务投入运行前需把任务体的代码域地址 CFA 压入 RS——勾链任务体,所以基于 FVM 的多任务运行无需进行任务虚拟地址到物理地址的映射,整个过程如图 5 所示。

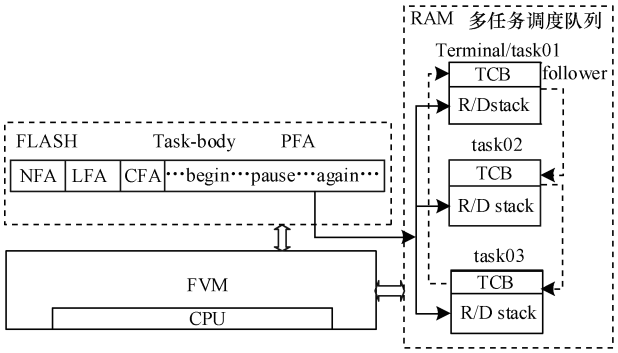

图 5 基于 Forth 虚拟机的协同式调度队列

Fig. 5 Collaborative scheduling queue based on Forth virtual machine

3 实验评估

本文在 AVR8 架构的 Arduino Uno Rev3 嵌入式开发板上,移植一个开源的 FVMOS,实现多任务空间复用算法。在实验环境中,利用 FVMOS 提供的内存地址访问接口和一些硬件传感器,设计了一个环境温度监控程序,程序的整个过程分为采集、处理、检测和控制,环境温度监控程序流程如图 6 所示。

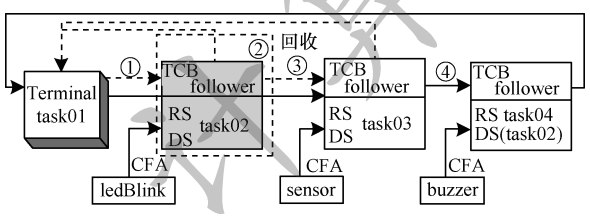

图 6 环境温度监控程序流程

Fig. 6 Environment temperature monitoring program flow

下面给出程序主要步骤的核心代码,根据 FORTH2012 标准,算法实现如下:

```
1. 20 20 task:task02
2. :startup (-) \ 定义系统启动任务
3. init
4. task02 task-init
5. \ 任务映像空间初始化
6. task02 tib > tcb activate ledBlink
7. \ task02 勾链 ledBlink 任务体
8. onlytask
9. task02 tib > tcb alsotask
10. multi; \ 开启多任务调度
11. 20 20 task:task03
12. task03 task-init
13. :linkTask03
14. task03 tib > tcb activate sensor
15. \ activate 勾链其后的任务体
16. task02 tib > tcb task03 tib > tcb
17. queue_schedule \ task03 添加到 task02 之后
18. task02 tib > tcb task_delete
19. \ 删除任务 task02
20. 20 20 task:task04
21. task04 task-init
22. :linkTask04
23. task04 tib > tcb activate buzzer
24. task03 tib > tcb task04 tib > tcb
25. queue_schedule
```

步骤 1 ~ 步骤 10(对应图 6 中的①)定义了任务 task02 并在 startup 中勾链控制 led 闪烁的任务体,加入到多任务调度队列。开启多任务,若 led 闪烁则表示多任务运行正常。步骤 11 ~ 步骤 17(对应图 6 中的②)添加后台任务 task03 并勾链控制温度传感器并获取外界温度的任务体 sensor。步骤 18、步骤 19(对应图 6 中的③)回收不再需要的指示任务 task02。步骤 20 ~ 步骤 25(对应图 6 中的④)添加 task04,勾链对 task03 获取到的温度进行判断触发警报的任务体 buzzer。

本文改进的系统程序中各任务的 TCB 块和堆栈大小(包含返回堆栈和数据堆栈)对比原系统中采用同样方法创建的任务,内存单元数据如表 2 所示。

表 2 程序任务内存单元对比

Table 2 Comparison of program task memory units

任务	原系统	本文改进的系统
task02	70(30 +40)	50(10 +40)
task03	70(30 +40)	50(10 +40)
task04	70(30 +40)	50(10 +40)

在 TCB 表项中,有关终端控制的变量在后台任务中无法使用,对原系统的任务创建过程进行分析和实验,系统启动即对终端任务 TCB 进行初始化,有关配置信息存放在 E^2PROM 中,在创建任务时只对 TCB 中前 5 项进行初始化,本文在实现多任务空间复用算法时对系统创建任务也进行了改进,保持原有接口不变,TCB 精简至前 5 项。

在环境温度监控程序的每一步操作后采集 HERE 值、查看后台任务数和计算当前系统内存利用率,并与原系统作对比,结果如表 3 所示。其中 HERE 值表示下一个可分配利用的地址空间,系统通过 unused 词获取当前系统可用 RAM 内存空间大小,实验硬件平台的 RAM 空间大小为 2 KB。

表 3　嵌入式 Forth 操作系统综合性能对比

Table 3　Comparison of overall performance between embedded Forth systems

任务阶段	原系统			本文改进的系统		
	HERE 值	任务数量	内存利用率/%	HERE 值	任务数量	内存利用率/%
系统启动	450	1	13.4	461	1	13.9
task02	521	2	16.8	512	2	16.4
task03	592	3	20.3	563	3	18.9
删除 task02	592	3	20.3	563	2	18.9
task04	663	4	23.8	563	3	18.9

在系统启动阶段,由于多任务空间复用算法引入了空闲任务映像分区链表来维护后台删除任务映像分区信息,plp 占用一部分空间,但其精简了后台任务 TCB 和 Re_allot 算法,使得系统空间资源得到更有效的利用。task04 重利用了 task02 的任务映像,HERE 值没有改变。整个应用程序内存使用率为 18.9% ,没有采用本文实现算法下的原系统最终程序内存使用率为 23.8% 。同样地,以 Arduino Uno Rev3 平台为例,假设任务堆栈空间大小均为 40,对比高负荷情况下两种方式能支持系统运行的最大任务数,如表 4 所示。

表 4　堆栈空间大小为 40 时高负荷情况下的任务数量

Table 4　Number of tasks under high loads when the stack size is 40

比较项目	原系统	本文改进的系统
无回收任务	24	34
回收 N 个任务	24 – N	34

当无回收任务时,本文多任务空间复用算法改进的系统和原系统均采用静态分配,TCB 表项精简至前 5 项,使得前者的任务数明显多于后者。当系统运行过程中有任务回收时,由于前者实现了多任务空间复用算法且对比实验中堆栈空间都为 40,回收的所有任务映像空间均能再次分配利用,相比无多任务空间复用算法下的原系统,最终运行的任务数为 24 – N,其中 N 小于系统最大支持任务数。

本文实现的算法基于 FVM 架构,不涉及宿主机底层硬件细节,平台移植可以不更改代码,保证了 FVMOS 的可移植、可交互、可扩展和可重构等特性,算法时间复杂度为 $O(N)$,空间复杂度为 $O(1)$。

4　结束语

FVMOS 系统在运行过程中无法删除后台任务映像,导致内存利用率低和系统性能下降。本文算法在保留 FVM 固有特性的前提下,时间复杂度控制在线性时间范围内,空间复杂度也在一个较小的常数范围内。实验结果表明,该算法在资源有限的嵌入式环境下,提高了内存利用率,系统调度队列中无需维护删除的任务,系统性能也有所提升,支持运行更多的任务。但目前系统采用的是协同调度算法,不能满足实时性任务的需求,因此下一步将改进多任务调度算法,对强实时任务采用静态分配方式,一般任务采用多任务空间复用算法,增强算法实时性,并且在实现基于优先级的可抢占调度算法时,需对算法中的相关操作进行互斥访问,保证系统内存分配的可靠性。

参考文献

[1] BRODIE L. Thinking Forth[M]. Los Angeles, USA: Punchy Publishing, 2004.

[2] DAI Hongbing. Design and implementation of high-efficiency microcomputer real-time multitask operation system[J]. Journal of University of Chinese Academy of Sciences, 1993, 10(3): 283-292. (in Chinese)
代红兵. 高效微机实时多任务操作系统设计与实现[J]. 中国科学院大学学报, 1993, 10(3): 283-292.

[3] Forth200x Standardization Committee. Forth-2012-ANSI X3. 215-1994[S]. Washington D. C., USA: ANSI Technical Committee, 2012: 1-248.

[4] HANNA D M, JONES B, LORENZ L, et al. An embedded Forth core with floating point and branch prediction[C]//Proceedings of the 56th International Midwest Symposium on Circuits and Systems. Washington D. C., USA: IEEE Press, 2013: 1055-1058.

[5] YANG Weimin, DAI Hongbing, AN Hongping, et al. A new Forth framework for embedded systems[J]. Journal of Yunnan University(Natural Sciences Edition), 2013, 35(S2): 96-103. (in Chinese)
杨为民,代红兵,安红萍,等. 一种新的嵌入式 Forth 实时操作系统的研究[J]. 云南大学学报(自然科学版), 2013, 35(S2): 96-103.

[6] FENIELLO A N. Programming the F18[EB/OL]. [2018-08-17]. https://blogs.msdn.microsoft.com/ashleyf/2013/10/13/programming-the-f18/.

[7] EDVIN H, CHEN L. Ep32——a 32-bit Forth microprocessor[C]//Proceedings of the 20th Annual Canadian Conference on Electrical and Computer Engineering. Washington D. C., USA: IEEE Press, 2007: 518-521.

[8] HASKELL R E, HANNA D M. A VHDL——Forth core for FPGAs[J]. Microprocessors and Microsystems, 2004, 28(3): 115-125.

[9] FORTH CPU cores. Forth interest group[EB/OL]. [2018-08-17]. http://www.forth.org/cores.html.

[10] HUANG Weifu, SHEN Chiliu. Control language for public——an example based on 328eforth[J]. Applied Mechanics and Materials, 2013, 418: 116-119.

[11] DAI Hongbing, ZHOU Yonglu, AN Hongping, et al. Research on embedded multitasking operating system architecture based on Forth virtual machine[J]. Application Research of Computers, 2019, 36(2): 476-480. (in Chinese)
代红兵,周永录,安红萍,等. 基于 Forth 虚拟机的嵌入式多任务操作系统体系架构研究[J]. 计算机应用研究, 2019, 36(2): 476-480.

(下转第 221 页)

(上接第 215 页)

[12] HE Li, SONG Lihong, DONG Lingfang, et al. Operating system practical tutorial [M]. Beijing: Tsinghua University Press, 2012. (in Chinese)
何丽,宋丽红,董林芳,等. 操作系统实用教程[M]. 北京:清华大学出版社,2012.

[13] FORTH, Inc. SwiftX cross compilersfor embedded systems applications[EB/OL]. [2018-08-17]. https://www.forth.com/embedded/.

[14] CHOU Wenlong, MEI Kuizhi, GAO Zenghui, et al. Design and Implementation of Multitask Scheduling for Embedded ARM GPU[J]. Journal of Xi'an Jiaotong University, 2014, 48(12): 87-92. (in Chinese)
丑文龙,梅魁志,高增辉,等. ARM GPU 的多任务调度设计与实现[J]. 西安交通大学学报,2014,48(12):87-92.

[15] DAI H, ZHOU Y, WANG L. Forth virtual machine task scheduling method, involves returning current stack pointer, scanning next process task status according to presses return stack, and returning pause breakpoint tostart execution process: CN107391251-A[P]. 2017-11-24.

[16] MCGUIRE T E. Kitt peak multi-tasking FORTH-11[J]. Journal of Forth Application and Reseach, 1984, 2(2): 56-57.

[17] JIN Xiangfeng, ZHOU Shuqin. Background task in PC/FORTH——the modified clock utility[J]. Computers and Applied Chemistry, 1988, 5(4): 57-59. (in Chinese)
金祥凤,周淑琴. FORTH 后台任务——时钟实用程序的改进[J]. 计算机与应用化学,1988,5(4):57-59.

[18] Forth Inc. SwiftX configurable _ multitasking _ kernel [EB/OL]. [2018-08-18]. https://www.forth.com/embedded/#Configurable_Multitasking_Kernel.

[19] WU Wenfeng. Design and implementation of embedded real-time system dynamic memory allocation manager[D]. Chongqing: Chongqing University, 2013. (in Chinese)
吴文峰. 嵌入式实时系统动态内存分配管理器的设计与实现[D]. 重庆:重庆大学,2013.

[20] CHI Yuanwu. Research on dynamic memory management optimization scheme of embedded real-time operating system [D]. Shanghai: Shanghai Jiao Tong University, 2016. (in Chinese)
池元武. 嵌入式实时操作系统动态内存管理优化方案的研究[D]. 上海:上海交通大学,2016.

[21] ZHOU Hangci. Programming technology based on embedded real-time operating system [M]. Beijing: Beijing University of Aeronautics and Astronautics Press, 2011. (in Chinese)
周航慈. 基于嵌入式实时操作系统的程序设计技术[M]. 北京:北京航空航天大学出版社,2011.

编辑　陆燕菲