ISSN 1009-3044
Computer Knowledge and Technology 电脑知识与技术
Vol.7, No.7, March 2011,pp.1646-1648

E-mail: kfyj@cccc.net.cn
http://www.dnzs.net.cn
Tel:+86-551-5690963 5690964

嵌入式操作系统的多线程机制研究与实现

陈雪芳

(东莞理工学院 计算机学院,广东 东莞 523808)

摘要:通过对嵌入式操作系统的多线程机制的理论进行研究与分析,提出了一种应用在嵌入式操作系统中的多线程机制实现方案。方案以多线程机制的理论为基础,建立了多线程机制的实现模型,并以 Cortex-M3 内核为例,深入的分析了这种多线程机制模型的调度实现方案,以及多线程的创建,切换,延时功能函数的实现。嵌入式操作系统的多线程机制的实现以实际应用为基础,以堆栈溢出为例,着重探讨了多线程机制在实现过程当中需要注意的安全因素。

关键词:嵌入式;操作系统;多线程;进程;Cortex-M3

中图分类号:TP316 **文献标识码**:A **文章编号**:1009-3044(2011)07-1646-03

Research and Implement of Multi-threads Technology Based on Embedded OS

CHEN Xue-fang

(Computer School, Dongguan University of Technology, Dongguan 523808, China)

Abstract: A new implement of multi-threads technology in embedded operating system is introduced in this paper, which is developed from traditional theory of multi-threads technology. A model of this new implement of multi-threads is bulit up and then the schedule how to create create, switch and delay multi-threads are deeply analyzed. All these operation is implemented in Cortex-M3 core. The implement of multi-threads technology in embedded operating system is based on practical applications. And the stack overflow action is taken as an example, to show design security factor while implement multi-threads.

Key words: embedded; operating system; multi-threads; process; Cortes-M3

随着嵌入式技术的发展,基于对整个系统(产品)软件的安全性,可靠性,开发周期等多方面因素的考虑,嵌入式操作系统正成为嵌入式系统不可缺少的重要组成部分。目前广泛使用在嵌入式系统中的操作系统各有优劣势,比如开源 linux 操作系统需要较多的硬件资源支持,在小型和有着特殊要求的嵌入式系统中,设计自己的嵌入式操作系统是一种比较有价值方案。

多线程机制是操作系统的重要组成部分,线程是在操作系统的进程设计过程当中演化出来的。线程可以与进程(任务)进行优劣势互补,以满足系统要求,比如线程之间可以很方便的共享进程里的所有资源,甚至包括了进程的全局变量;而且,线程的切换(调度)开销比进程更少,比如在一个嵌入式终端的网络通信程序当中,需要同时处理接收与发送,而且接收和发送的信息有较大的相关性,这种情况下采用多线程方式是最优选择。

本文通过研究多线程的实现原理,给出了一种基于嵌入式操作系统的多线程机制实现模型,并以此为基础深入研究了其调度,创建,切换和延时的实现。

1 多线程机制模型

传统上线程可以分为用户级与内核级两种[1],用户级线程完全通过在用户级提供的一个程序调用库来实现多线程,比如 MACH 操作系统的 C-threads 库和 POSIX 标准定义实现的 pthreads 库,这些库提供了线程的所有操作,比如创建,同步,调度等。用户级线程最大的好处是无需操作系统内核的支持,因而调度开销较少,内核级线程则是使用内核来维护线程的调度,管理,这时,操作系统提供一系列的系统调用接口让用户程序来管理线程。采用内核级线程设计方式更有利于并发使用多处理器的资源。在现代的操作系统设计当中,往往合并使用这两种线程机制,在操作系统的设计过程当中,可以根据不同的需要,把线程的管理部分放在不同的级别。

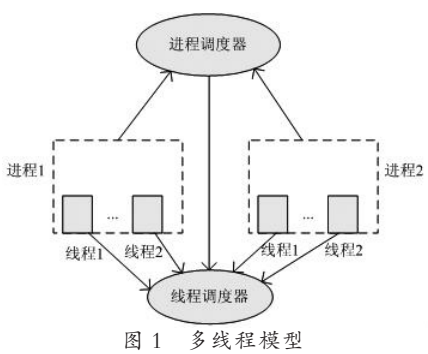

图 1 多线程模型

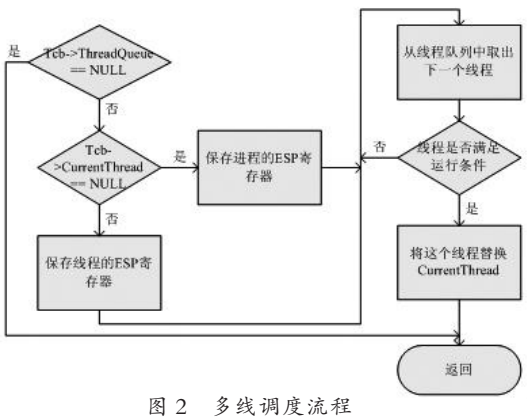

图 2 多线调度流程

在嵌入式系统的设计过程中,较少的硬件资源,较高的实

收稿日期:2011-01-28
基金项目:东莞市高等院校科研机构重点科技计划项目(200910814002)
作者简介:陈雪芳(1978-),女,广东英德人,讲师,工程师,硕士,主要研究方向为嵌入式系统设计。

时性要求是对嵌入式操作系设计的基本要求。在操作系统当中可以同时存在多个进程,一个进程内又可以同时存在多个线程[2]。从理论上分析,一个操作系统中可以同时存在大量的进程和线程。但是嵌入式系统具有很强的功能专一性,同时运行大量的线程的情况很少存在。针对嵌入式系统的这些情况,本文设计的多线程模型如图 1 所示。

在这个多线程模型中,进程调度器与线程调度器都是内核的一部分,进程调度器只负责对进程的调度,线程调度器只对线程进行调度。当发生进程调度时,进程调度器从运行队列中选择将要运行的进程,然后启动线程调度器。线程调器启动后从该进程的线程队列中选择将要运行的线程,然后切换进程。当线程运行时发生调度,将直接启动线程调度器,进行线程之间的切换。

2 多线程机制在内核级的实现

多线程机制在内核级主要完成调度工作,接下来以 ARM 的 Cortex-M3 内核为例对多线程的实现进行研究。为了实现对多线程的支持, 需要在进程 TCB (task control block) 结构中增加对线程的描述变量, 其中最关键的两个变量是 ThreadQueue 和 CurrentThread。ThreadQueue 是线程队列的头指针,CurrentThread 指向当前运行的线程。进程调度器完成对进程调度后启动线程调度器,图 2 是线程调度器的流程图,多线程调度器首先对 ThreadQueue 判断,查看是否有线程存在,如果没有线程存在,则返回。当有线程存在时,如果 CurrentThread 为空,则说明这是第一次运行线程,这时保存进程的 ESP 寄存器的内容,否则,保存当前线程(也就是 CurrentThread 所指向的线程)的 ESP 寄存器的内容。如果线程实现分时调度算法,则从当前线程位置开始,依次在线程队列中找到下一个满足执行条件的线程。因为线程可能处于睡眠状态,所以要判断线程是否处于等待状态。对于线程的睡眠,不必在每次启动线程调度器时都遍历线程链表中的所有的线程来判断线程的睡眠时间是否超时,这样做会增加调度器的运行时间,对于小型的嵌入式系统,这是不能容忍的。所以,可以在找到该线程时通过对线程进入睡眠时系统计数器加睡眠时间与当前系统计数器进行比较来判断是否睡眠超时[3]。因而,对于线程 TCB 块必须存在两个变量 ThreadDly 和 ThreadCnt,ThreadDly 用于保存需要睡眠的时间(在内核,一般用嘀嗒数来表示),ThreadCnt 用于保存进入睡眠时系统的嘀嗒数。系统嘀嗒数的使用比较普遍。比如,在 linux 中用 jiffies 来表示系统嘀嗒数[4]在线程调度器中,之所以只关心 ESP 寄存器,是因为我们要把所有寄存器的内容都存在堆栈中。对于 Cortex-M3 内核,当发生中断时,MCU 首先会自动将 xPSR,PC,LR,R12,R3-R0 压入堆栈[5]。对于操作系统而言,这些显然不够,因为我们不能保证应用程序在进入中断时会用到哪些寄存器。所以为了安全考虑我们需要把所有的寄存器压入堆栈。当线程(进程)恢复时,只要把所有的寄存器依次弹出就可以。所以,ESP 寄存器非常关键,因而在进程 TCB 和线程 TCB 中都要使用一个变量来保存这个寄存器的值。而在调度器返回后,操作系统内核不必去关心它要运行的是进程还是线程,它只会根据 ESP,依次弹出堆栈的内容到寄存器。因而与无线程的操作系统相比,我们增加的调度开销几乎只是对线程队列查找的开销。对于 ucos 系统,采用了一种比特位映像的方法来查找下一个运行的进程,这种方法对于小型操作系统的调度具有很高的效率[6]。

3 多线程在用户级的实现

除了线程调度,对于应用程序,还需要更多的其它功能来管理线程。为了减少内核的复杂度,对内核实现模块化设计,需要将一些要求不高的功能放到内核之外。在 linux 系统中,对线程的处理就全部放在用户级。为了便于应用程序对线程的管理,嵌入式操作系统提供了一系列的接口函数供应用程序调用,比如线程创建,线程删除,线程切换,线程延时等。对于这些接口函数,要满足安全,稳定,高效的要求。

3.1 线程的创建

多个线程之间可以共享进程资源, 因而线程没有必要设计得象进程那样庞大。一个线程创建函数的例子为 int ThreadCreate (void *stack, unsigned int dwStackSize, void *StartAddress, unsigned int Parameter, unsigned int ThreadId)。参数 stack 指向调用时申请的线程堆栈空间,dwStackSize 是线程堆栈的大小,StartAddress 是线程运行函数的地址,Parameter 是调用者传递给创建函数的参数,ThreadId 是线程 ID 号。在这个函数当中,只有一个参数(Parameter)用于调用时将参数传递给创建函数,并且定义为 unsigned int 类型。这是因为有些编译器(比如 realview 和 keil MDK 的编译器)不能识别结构体作为函数的参数。所以当需要传递多个参数时,可以将结构体的指针作为传递参数。参数 stack 指向了调用时申请用来作为线程堆栈的空间。严格来说,stack 指向的是申请到的 heap,它向下增长。线程创建函数首先进行安全检查, 比如 stack 所指向的空间大小是否大于最小需求等。然后在线程堆栈中填充线程 TCB,之后便是填充寄存器的值,寄存器的顺序必须是和调度时压堆的顺序相对应,最后是将线程加入进程的线程队列, 如果需要马上运行线程, 则将线程状态置为就绪。图 3 是基于 Cortrex-M3 内核的线程创建时堆栈的示例图。对于 Cortex-M3,堆栈指针必须 4 字节对齐,这一点可能和别的处理器不一样,因为 Cortex-M3 的 ESP 的最后两位没有用。而应用程序申请的空间并不能一定保证会是 4 字节对齐的 5,所以必须做好对齐处理。前面已经提到过,应用程序申请到的是 heap,它向下增长,但是堆栈是向上增长的,如图 3 中 stack 就是这段空间的地址,而堆栈是向上增长,所以填充线程堆栈时要注意顺序。图 3 的“模拟中断压栈”部分是按 Cortex-M3 产生中断时压栈顺序填充的, 在这部分之后就是我们自己的压栈部分, 如图 3 中的 R14-R0 和 ESP。完成 ESP 的填充后,就是将线程 TCB 中的状态变量置为就绪,最后打开中断。因为 ThreadCreate()函数是在应用程序中被调用的,所以在 ThreadCreate()函数的执行过程当中任何情况都有可能发生,因而必须在关键步骤时关中断。

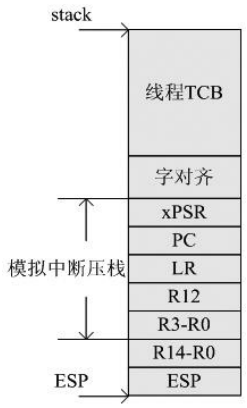

图 3 线程创建时的堆栈

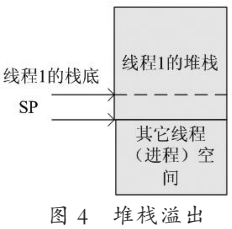

图 4 堆栈溢出

线程创建后,就在进程的 ThreadQueue 队列中加入了一个节点,当下一次启动线程调度器后就会运行这个线程。

3.2 线程切换

线程切换广泛的应用于其它线程功能函数中,比如线程延时,自旋锁,旗标等调用当中[7]。线程切换有三个基本的工作要做,分别是:安全检查,将线程置为等待状态和启动线程调度器。安全检查是对线程的状态和资源进行简要的检查,有时程序的运行安全要求要高于速度等其它一切要求,特别是在医疗电子产品的设计当中。以堆栈的安全检查为例,图 4 是一个在没有 MMU(内存管理单元)的处理器上产生的堆栈溢出情况。在图 4 中,堆栈指针寄存器已经处于线程 1 的堆栈底部时,如果再次压栈就会产生堆栈溢出。如果线程 1 的堆栈底下还有内存,则不会产生硬件异常,这时线程 1 并不知道发生了溢出,在这种情况将会对紧接在线程 1 下面的应用程序造成严重的破坏。这时线程 1 正在运行,所以,只要线程 1 一直占据处理器,对整个系统的安全性能没有产生严重的危害,但是如果把线程 1 切换出去,运行到被破坏的程序时,后果将是很严重。比如一个非常重要的变量被线程 1 修改了,这时运行的程序将有可能使系统输出一个错误的信号。所以,在线程切换时,必须将当前堆栈指针寄存器与堆顶和堆栈大小比较。如果发生了堆栈溢出,线程切换程序产生一个严重异常信息给操作系统。

线程切换接下来的一个重要工作就是启动线程调度器。线程切换只是启动线程调度器,它只对进程内的线程切换,相比线程切换,可以减小开销。线程切换可以用软中断或其它方式来模拟压栈。

3.3 线程延时

一个线程延时函数的例子为 int ThreadDly(unsigned int dwMilliseconds),如果调用时发生异常,则 ThreadDly 返回错误码,参数 dwMilliseconds 是延迟时间,单位是毫秒。线程延时函数首先检查线程,看看线程是否满足睡眠条件,然后将延迟时间 dwMilliseonds 与当前系统嘀嗒相加,放入线程的 ThreadDly 变量中,最后调用线程切换。这里要注意的一个问题是要防止线程延迟操作时将进程睡眠。

4 结束语

嵌入式操作系统的多线程机制研究与实现从理论与实际出发,在研究了多线程机制的原理后,给出了一种针对小型嵌入式操作系统的实现方案,这种多线程实现方案,在实际的项目中得到了很好的应用。本文虽然是主要针对小型的嵌入式操作系统,对于硬件资源非富的大型嵌入式系统,多线程的实现原理相同,只要根据系统追求速度,存储空间等要求做出相应的取舍。对于多线程机制的研究还有许多工作值得开展,比如调度算法,多线程库的实现等。

参考文献:

[1] 罗宇,商临锋.操作系统多线程实现技术研究[J].小型微型计算机系统,2000,21(5).
[2] Mrva M.Reuse factors in embedded systems design[J].Computer,1997,30(8):93-95.
[3] William Stalling.Operating systems:internals and design principles[M].Prentice Hall,1998.
[4] 毛德操,胡希明.Linux 内核源代码情景分析[M].杭州:浙江大学出版社,2001.
[5] 宋岩,译.ARM Cortex-M3 权威指南[M].北京:北京航空航天大学出版社,2009.
[6] 何先波,芦东昕,李志蜀,等.一种面向通信领域的高效嵌入式操作系统进程队列模型[J].哈尔滨工程大学学报,2006,27(2).
[7] KATCHEK D I,SATHAYE S S,STROSNIDER J K.Fixed priority scheduling with limited priority levels [J].IEEE Trans on Computers,1955,44(9):1140-1144.

(上接第 1645 页)

4 设计仿真与验证

本设计在搭建完成的验证平台环境上,采用 VERA 验证语言编写 TestCase,对电路加输入激励,采用 Nanosim 数模仿真工具仿真验证电路如图 7 所示。

设计结果符合设计要求,数据写入安全,延时调节有效,达到设计目标。

参考文献:

[1] Jacob Baker R.CMOS 电路设计布局与仿真[M].陈中建,译.北京:机械工业出版社,2006.
[2] Kuriyama H,Ishigaki Y.A C-Switch cell for low-voltage and high-density SRAM's [J].IEEE Transactions on electron devices,1998,45(12):2483-2487.
[3] Gaillard R,Poirault G.Numerical simulation of hard errors induced by heavy ions in 4T hig density SRAM cells [J].IEEE Trans.On Nuclear Science,1994,41(3):613-618.
[4] Levy H J,Daniel E S,Mcgill T C.A Transistorless-Current-Mode static RAM architecture[J].IEEE JSSC,1998,33(4):669-672.