

一种符合 OSEK 标准的操作系统微内核设计

郭 景, 陈贤富

(中国科学技术大学 信息科学技术学院, 安徽 合肥 230027)

摘 要: 本文提出了一种符合 osek 标准的嵌入式操作系统微内核, 该内核对实时进程和一般进程采取了两种不同的调度策略, 既能满足实时性要求, 又能兼顾系统吞吐量. 此外, 还提出了一种通用的上下文切换模型, 使得该内核可以方便地移植到不同的平台. 相比直接使用商用操作系统或者前后台系统, 它具有通用性强、执行效率高、占用空间小等优点. 基于 STM32 处理器平台的实验表明, 该内核仅占用 924 Bytes ROM, 在 72 MHz 系统时钟驱动下, 完成一次任务切换仅需 $3.4 \mu\text{s}$.

关键词: 操作系统; 微内核; 任务调度; 上下文切换

中图分类号: TP316.2

文献标识码: A

文章编号: 1000-7180(2017)11-0016-05

An Operating System Micro Kernel Design Based on OSEK Standard

GUO Jing, CHEN Xian-fu

(School of Information Science and Technology, University of Science and Technology of China, Hefei 230027, China)

Abstract: This paper proposes a micro kernel based on OSEK standard. The kernel takes two different scheduling strategies for real-time process and general process, which can meet the real-time requirements, but also consider the system throughput. Compared with the direct use of commercial operating system or foreground and background system, it has the advantages of high efficiency, small space occupation. The experimental results based on STM32 processor platform show the kernel takes only 924 Bytes ROM and it requires only $3.4 \mu\text{s}$ for a task switching, driven by 72 MHz system clock.

Key words: operating system; micro kernel; task scheduling; context switching

1 引言

随着智能硬件的兴起, 嵌入式开发越来越复杂. 为了降低开发难度, 越来越多应用选择移植商用操作系统. 如果直接使用商用操作系统, 会引入很多冗余的功能, 比如内存管理、文件系统等. 这将占用过多的硬件资源, 给人一种“头重脚轻”的感觉. 另一方面, 大规模地使用商用操作系统需要付费, 对于成本敏感的应用来说无法接受.

在一些应用开发过程中, 为了完成不同任务的切换, 开发人员往往会采取“前后台系统”的设计方案. “前后台系统”的实时性比较差, 因为所有任务的优先级都是一样的, 任务的执行通过队列排队, 一些

实时性要求较高的任务就没有方法立即被执行. 更严重的情况是, 一旦中间任何一个任务崩溃, 那么所有其他任务都将得不到执行.

为了解决这些问题, 国内外的研究人员进行了很多研究. 文献[1, 2]设计了一种类 uCos_III 的实时系统内核, 但是都只支持实时任务调度器. 任务运行时间、何时挂起、挂起时间等繁琐的参数需要开发人员设置, 带来很多不便利. 并且对于很多非实时进程, 这种调度策略效率低下.

为此, 本文根据 OSEK/VDX 标准, 设计了一种操作系统微内核. 它为实时进程和非实时进程采取了两种不同的调度策略. 既保证了系统的实时性, 又兼顾系统的吞吐量. 提出的通用上下文切换模型, 使

得内核可以很方便的移植到不同的平台. 该内核很好地解决了使用商用操作系统过于庞大、“前后台系统”任务切换性能不佳的问题.

本文所述的操作系统微内核主要分为两个部分,一个是任务调度模块,另一个是上下文切换模型.

2 任务状态转换图

嵌入式系统中一般只有一个 CPU,每个时刻只能有一个任务处于运行状态. 本文根据任务是否占用 CPU、是否处于可调度状态,将任务划分任务运行、任务就绪和任务挂起 3 种状态. 任务运行表示该任务占用 CPU,正在运行. 任务就绪表示该任务等待调度,一旦被调度器选中即可投入运行. 任务挂起表示该任务不具备运行条件,需要其他任务调用恢复挂起函数 resume,才能转为就绪状态. 这三种状态的转换如图 1 所示,其中任务控制块 task_struct 中的任务状态标志 state 为 0 表示就绪状态、-1 表示挂起状态、1 表示就绪状态.

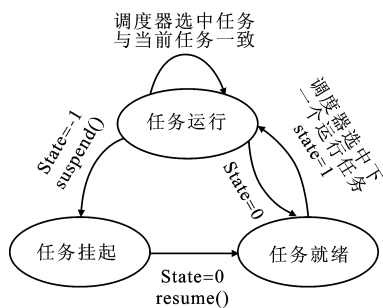


图 1 任务转台转化图

3 任务调度模块

本文提出的任务调度模块采用两种不同的调度器策略. 对于实时性要求较高,但是运行时间较短的任务采用实时调度器. 实时调度器以任务优先级为基础,优先级高的任务优先运行. 如果任务就绪队列中有实时任务,那么始终运行优先级最高的实时任务,直到该任务主动调用 suspend() 函数将自己挂起. 实时调度器是一种静态调度的策略,具有较高的响应速度,满足了实时系统的要求. 另外,对于每次运行时间比较长的非实时任务采用一般调度器. 一般调度器以剩余时间片为基础,剩余时间片较多的任务优先运行. 当所有一般任务的时间片都消耗完毕,再重新初始化任务的时间片. 一般调度器随着系统的运行,任务的运行优先次序在不断改变,所以它是一种动态调度的策略,提高了系统的任务吞吐量.

为了简化任务控制块的结构,不再为每个任务的调度器分配一个标识变量,而是直接采用任务优先级 prio 标识. 对于一般的任务,它的任务优先级大于 0,数值越大表示优先级越低,一般任务的最高优先级为 0. 对于实时任务,它的优先级小于 0,数值越小优先级越高. 另一方面,每个任务的任务优先级唯一,因此它还可以作为任务号使用.

3.1 任务调度算法

任务调度的过程为每个任务都计算出一个反映其运行“资格”的权值 weight,然后从中挑选权值最高的任务投入运行. 任务调度算法如图 2.

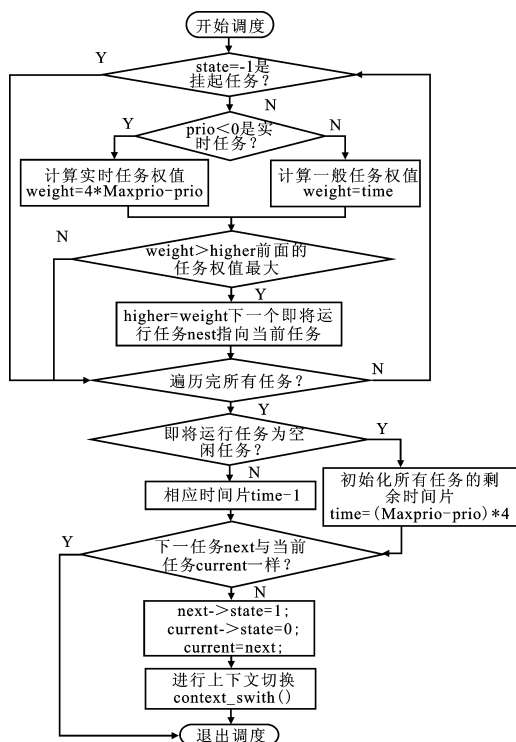


图 2 任务调度算法

对于一般任务,它们的权值就是其剩余时间片大小,如式 1 所示, time 表示剩余时间片. 一般任务的初始时间片大小由式 2 得出, MaxPrio 表示一般任务的最低优先级, prio 表示当前任务的优先级. 系统运行的过程一般任务的剩余时间片递减,其权值 wight 也递减. 当所有一般任务的时间片都消耗完毕时,重新由式 2 初始化剩余时间片 time.

$$\text{weight} = \text{time} \quad (1)$$

$$\text{time} = (\text{MaxPrio} - \text{prio}) * 4 \quad (2)$$

微内核中自带一个特殊的一般任务,它的优先级为 MaxPrio-1,它在一个周期中调用一次. 当系统没有任务可做时,不断的调用这个空闲任务.

$$\text{weight} = \text{MaxPrio} * 4 - \text{prio} \quad (3)$$

对于实时任务,它们的权值只需要满足两个限制条件.(1)实时任务的权值一定要大于一般任务.(2)优先级较高的实时任务权值大于优先级低的实时任务.因为实时任务的优先级是负数,所以它们的权重可根据式(3)得出,满足上述两个限制条件.

3.2 算法性能参数

系统中优先级最低的任务是空闲任务,优先级为 $\text{MaxPrio}-1$,由式(2)可得出,空闲任务的时间片 time 始终都是 4.当任务调度器选出的任务为空闲任务就认为当前系统没有任务或者任务的时间片都消耗完毕,这时就重新计算所有任务的剩余时间片.因此实际上一般任务的最小剩余时间片为 3.

当系统中只有一般任务,一般任务总数为 N ,最低优先级为 MaxPrio ,优先级为 prio_i ,那么把所有一般任务都调度一次的周期 T 可由式(4)得出.

$$T = 4 \sum_{i=0}^N (\text{MaxPrio} - \text{prio}_i) - 3N \tag{4}$$

由式(2)和式(4)可知,每个一般任务在一个调度周期内的时间占比为式(5).因此一般任务的优先级越高其运行的时间越多,在一个调度周期内运行的时间占比越大.

$$\text{percent}_i = \frac{4 * (\text{MaxPrio} - \text{prio}_i) - 3}{4 \sum_{i=0}^N (\text{MaxPrio} - \text{prio}_i) - 3N} \tag{5}$$

假如有 3 个一般任务 A、B、C 和 1 个空闲任务 D,它们的优先级、初始时间片以及这 4 个任务运行的次序如表 1 所示.

表 1 一般任务调度次序表

	A	B	C	D(空闲)
优先级	2	0	1	3
初始时间片	8	16	12	4
时间占比/%	17.86	46.43	28.57	3.57
运行次序	BBBBBCBCBCBCBCABCABCABCAD (周期为 28 个时间片)			

当系统中既有一般任务又有实时任务时,始终运行最高优先级的实时任务,直到该任务主动将自己挂起.所以该实时任务在调度周期内的时间占比为 100%.

假如有 2 个实时任务 A、C,1 个一般任务 B 和 1 个空闲任务 D,它们优先级、初始时间片以及任务的运行次序如表 2 所示.

表 2 实时任务调度次序表

	A	B	C	D(空闲)
优先级	-2	0	-1	1
初始时间片	x	8	x	4
时间占比	100%	0	0	0
运行次序	AAAAAAAAAAAAAAAAAAAA AAAAAAAA... (一直都是 A)			

4 上下文切换模型

任务调度算法挑选出下一个即将运行的任务后,就要进行上下文切换 context_switch ,如图 2 所示.任务调度算法与具体硬件平台无关,而上下文切换与硬件有较大的相关性.但不论是 Linux 系统还是一些嵌入式操作系统,其上下文切换的思路都是类似的,具体在硬件上的实现所有差异^[3].本文以 Linux 系统的上下文切换为框架^[4],对硬件相关特性进程抽象,提出了一种通用的操作系统上下文切换模型,如图 3 所示.该模型兼容了 Linux、Vx-Works 等主流操作系统的上下文切换技术的设计思想,不仅适用于 ARM、i386 处理器,甚至还可以应用到像 51 单片机这样的微控制器.

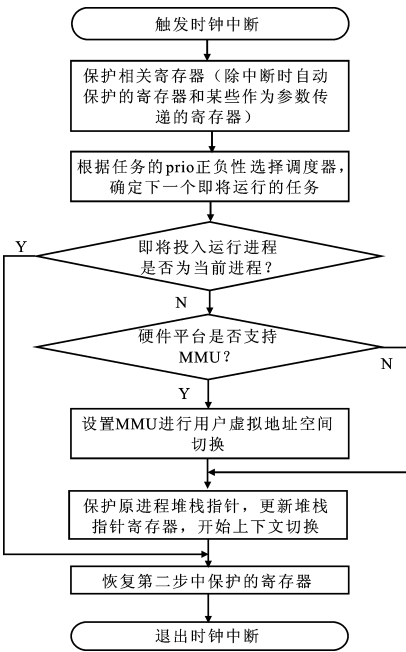


图 3 通用上下文切换模型

一般地说,上下文切换可以在软中断和外部中断中完成,软中断用于主动切换,而外部中断用于被动切换.为了简化任务切换方法,本文只在时钟中断中进行任务切换.如果将时钟中断的周期设置为 20 ms,那么对于实时任务来说,它的响应时间最多就是 20 ms.开发人员可以根据系统的要求,设置适当

的时钟中断周期。

本文提出的通用上下文切换模型的性能指标,依赖于开发者针对某个硬件平台的具体实现。上下文切换需要充分的利用具体硬件的特性,比如中断机制、处理器模式,这样才能最大程度提高上下文切换的速度。

5 实验结果及分析

5.1 实验目的和内容

在嵌入式开发中,STM32 芯片凭借较低的价格、强大的性能,得到了广泛的应用。为了验证本文提出的抢占式操作系统微内核,并测试其相关性能,本实验在 STM32F103 平台上移植了该内核。

(1)任务控制块是每个任务的“身份证”,记录着任务的状态、优先级、剩余时间片等信息。它也是调度器进行任务调度的基本单位。使用结构体表示任务控制块。

```
struct task_struct{
    uint * sp; //任务堆栈顶指针
    int state; //任务状态
    int prio; //任务优先级
    int time; //任务剩余时间片
    uint stack[StackDeep]; //任务堆栈
};
```

(2)用户需要指定中任务的个数 MaxTask,系统静态分配所有任务的任务控制块,使用结构体数组 PCB 的下标可以遍历所有任务控制块。

```
struct task_struct PCB[MaxTask];
```

(3)运行中的系统需要知道当前处于哪个任务, current 变量指向当前运行的任务,在任务调度的过程会使用该信息。

```
struct task_struct * current = 0;
```

(4)为了避免引入实时任务后,实时任务独占 CPU,本系统加入了简单的任务挂起功能 suspend(),挂起的任务号就是其优先级 prio。同理恢复挂起 resume()就是将 state 设置为就绪状态 0。

```
void suspend(int prio){
    int i;
    for(i=0;i<numTask;i++){
        if(PCB[i].prio == prio){
            PCB[i].state = -1; //该任务设置为挂起
            return;
        }
    }
```

(5)本文提出的任务切换方法只需要给用户提供了两个接口函数。

第一个是任务创建函数 create_Task。用户创建新的任务时需要指定其优先级 prio,以及该任务代码的指针 task。create_Task 创建新的任务的过程中,系统自动初始化任务控制块的各个参数。特别地是,因为 STM32 默认的堆栈类型是满减堆栈,所以初始的任务堆栈指针 sp=&stack[StackDeep-1]。void create_Task(int prio, void (* task)(void))

第二个接口是启动系统函数 osstart。系统启动过程首先创建空闲任务,设置当前任务 current 为空闲任务,然后启动周期性中断 systick,接着等待系统时钟中断发生,在周期性时钟中断中完成任务调度和上下文切换。

```
void os_start(void)
```

(6)任务调度算法与硬件无关,具体的实现过程如图 2,而上下文切换必须结合硬件的特性来实现。STM32 芯片在 ARM Cortex M3 内核中专门设置了一个系统滴答时钟 systick,本实验的周期性时钟中断借助该中断,周期为 10 ms。为了加快任务切换的速度,上下文切换一般使用汇编代码来实现。结合 STM32 芯片的硬件特性,在任务中使用过程堆栈指针 PSP,而中断中使用主堆栈指针 MSP。另外,STM23 中断的过程硬件自动保护了 R0-R3,R12,LR,PC 和 xPSR,所以软件只需要保护 R4-R11。

```
SysTick_Handler
CPSID I;关中断
……;保护中断现场,压栈 R4-R11
LDR R0, =scedule
BLX R0;调用任务调度算法,确定下一个运……;行任
务的任务控制块 next
……;判断下一个任务 next 是否与当前 CBZ R2, skip;
任务 current 一致?
LDR R0, =current
STR R1, [R0];不一致则 next->current
skip
LDR R0, =current
LDR R0, [R0]
LDR R0, [R0];取下一个任务的栈顶指针 sp
LDM R0, {R4-R11};恢复现场,弹出 R4-R11
ADDS R0, R0, #0x20
MSR PSP, R0;切换堆栈指针
ORR LR, LR, #0x04;返回使用进程指针 PSP
CPSIE I;开中断
BX LR;中断返回
```

5.2 实验验证与分析

为了验证本文提出的方法的可行性。本实验首先创建了如表 1 所示的一般任务。

```
int main(void) {
.....//初始化外设
create_Task(2, task_A);//创建任务 A
create_Task(0, task_B);//创建任务 B
create_Task(1, task_C);//创建任务 C
os_start();//启动系统,从此一去不复返
}

每个任务都是从串口输出对应任务的字母,例如任务 A,输出‘A’.
```

```
void task_A(void) {
while(1) { printf("A"); }
}
```

借助 SecureCRT 软件监视串口输出的结果. 由图 4 可知,一般任务以正确的次序输出,并且每 10 ms 切换一次.

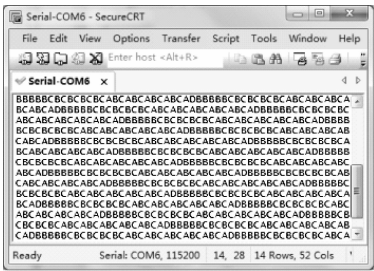


图 4 一般任务切换结果

接下来验证进程列表中既保护实时任务,又有一般任务的情形. 同样实现表 2 的任务,输出的结果. 由图 5 可知,系统始终运行优先级最高的实时任务 A. 并且任务 A 能在 10 ms 内得到响应,满足绝大多数应用实时性要求.

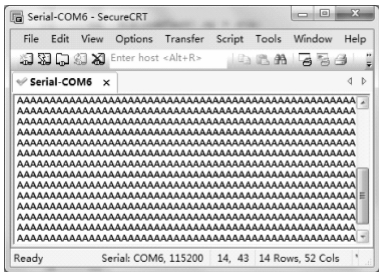


图 5 实时任务切换结果

图 4 和图 5 的结果验证了本文设计的嵌入式操作系统微内核的可行性.

因为 STM32 内核采用 ARM Cortex M3 架构,属于精简指令集 CPU,单条指令所占用的时间平均为 0.8 t(其中 t 表示系统时钟周期,本平台系统时钟频率为 72 MHz)^[5]. 在 keilu Vision 5 下编译该操作系统微内核,任务调度函数 scedule 和上下文切换 context_switch 大约使用了 310 条指令,所以每

次任务切换的耗时 $T \approx (1/72) * 0.8 * 310 = 3.4 \mu s$,任务切换耗时占任务运行时间比重 $3.4 \mu s/10 ms = 0.34\%$,非常高效. 另外,本文的微内核总共占用 924 bytes ROM. 本文设计的嵌入式系统微内核与其他系统内核等对比结果如表 3. 由表 3 看出,本文设计的微内核在各项参数上均具有明显的优势.

表 3 本文内核与其他内核对比

	文献[1]内核	uCOS_III	本文内核
任务切换时间/ μs	13.98	39	3.4
内核大小/Bytes	10 453	>24 000	924
支持任务类型	仅实时任务	仅实时任务	实时任务和一般任务

6 结束语

本文设计了一种符合 OSEK 标准的嵌入式操作系统微内核. 该内核具有通用性,适用于多数嵌入式开发平台. 通过引入两种不同的任务调度器,对实时任务和一般任务都提供的支持,既保证了实时性,又兼顾了系统吞吐量. 相比直接使用商用操作系统,该内核没有冗余的功能,占用空间小,大小只有九百多字节. 另外,相比传统的“前后台系统”,它的任务切换效率非常高,完成一次任务切换的总时间大概只有 3.4 μs . 可以满足多数实际应用需求,具有广阔的应用前景.

参考文献:

[1] 朱怡安,魏润之,苏世游,等. 一种基于 $\mu C/OS-II$ 符合 OSEK 标准的实时系统内核设计[J]. 计算机科学, 2016, 43(4):173-176.

[2] 王红玲,吕强,褚亚铭. 一个微内核操作系统的设计与实现[J]. 微电子学与计算机, 2008, 25(4): 9-12.

[3] Love R. Linux kernel development[M]. 3rd ed. USA, State of New Jersey, Pearson Education, 2010.

[4] 毛德操,胡希明. Linux 内核源代码情景分析:上册[M]. 浙江:浙江大学出版社, 2001.

[5] Yiu J. The definitive guide to the ARM Cortex-M3[M]. 2nd ed. 中国,北京:Newnes, 2009.

作者简介:

郭景男, (1992-), 硕士研究生. 研究方向为嵌入式系统操作系统、智能信息处理. E-mail: 2295952451@qq.com.

陈贤富男, (1963-), 博士, 副教授. 研究方向为复杂系统与计算智能.