一步步实现自己的内核调度程序(v1.0)

--基于 TQ2440(ARM9)的实现

作者：姜广伟
日期：2010-12-29
邮箱：Guangwei.jiang@gmail.com

很多程序员对“内核调度”概念并不陌生，但如果让大家去设计并实现“自己”的内核调度程序，也许并非易事。

出于好奇，我记下了自己设计和实现“内核调度”的四部曲。设计十分简单但对理解“内核调度”会有一些帮助，希望能给大家带来一点启发！

下面对“内核调度”的四部曲作简要的介绍：

1. 简易多任务设计(1)
   -- Task1 和 Task2 通过调用函数 Task_Sw_1To2/Task_Sw_2To1 实现两个任务间上下文的切换。
2. 简易多任务设计(2)
   -- 加入“轮转法调度”，Task1 和 Task2 通过调用函数 Task_Sw 即可实现多任务切换。
3. 简易内核调度(1)
   -- 加入时钟中断，当时钟中断发生时，内核调度程序自动切换不同的任务。
4. 简易内核调度(2)
   -- 加入“临界区(Critical Section)”,避免任务间的竞争；
   -- 中断发生时，增加判断“时钟中断”。

以下是在设计和实现“内核调度”程序时参考的书籍/文章，感谢这些书/文章的作者/译者：

1. 《嵌入式 Linux 开发应用开发完全手册》，韦东山编著，人民邮电出版社出版
2. 《ARM 嵌入式系统开发—软件设计与优化》，Andrwe N.Sloss/Dominic Symes/Chris Wright 著，沈建华译，北京航空航天大学出版社出版
3. 《嵌入式实时操作系统 uc/OS-II》，Jean J. Labrosse 著，邵贝贝等译，北京航空航天大学出版社出版
4. 《简易多任务 OS 设计》，lisuwei，http://group.ednchina.com/999/20520.aspx

简易多任务设计(1)

--基于 TQ2440(ARM9)的实现

作者：姜广伟
日期：2010-12-9
邮箱：Guangwei.jiang@gmail.com

1. 概述

多任务是现代操作系统的核心功能，深入理解多任务并非易事，但学习者如果能尝试去实现一个简易的多任务程序，对理解操作系统的“多任务”概念会有很大的帮助。若能进一步拓展这个多任务的程序，则可设计一个简易的内核(Kernel)。

本文就是介绍一个十分简易的多任务程序。

实验程序(TaskSwitch)中共有 2 个任务，Task1 和 Task2：
Task1 中，闪烁 LED1 灯 3 次后，主动将 CPU 使用权切换给 Task2；
Task2 中，闪烁 LED2 灯 3 次后，主动将 CPU 使用权切换给 Task1。
从而，实现 2 个任务的交替执行。

实验环境说明：

本实验的操作平台为 TQ2440(ARM9)开发板，程序代码在 S3C2440“SteppingStone”的 4KB 内部 RAM 中运行。程序就是利用这 4KB 的空间来实现多任务的切换。

请选择从 NAND Flash 启动，并将生成的二进制文件烧到 NAND Flash 中。

当选择从 NAND Flash 启动 CPU 时，CPU 会通过内部硬件将 NAND Flash 开始的 4KB 字节数据复制到“SteppingStone” 的 4KB 内部 RAM 中（此时内部 RAM 的起始地址是 0），然后跳到地址 0 开始执行。

Task1 和 Task2 的代码如下，

```
54 void Task1()
55 {
56     int i = 0;
57     int j = 0;
58
59     GPBDAT &= ~(0xf<<5);              // 点亮LED1-4
60     for(i=0; i<150000; i++);          // 延时循环
61
62     GPBDAT |= (0xf<<5);               // 关闭LED1-4
63     for(i=0; i<150000; i++);          // 延时循环
64
65     TaskCreate(Task2, TASK2_SP, TASK2_STK_PTR);
66
67     while(1)
68     {
69         GPBDAT |= (0xf<<5);           // 关闭LED1-4
70         GPBDAT &= ~(1<<5);            // 点亮LED1
71         for(i=0; i<150000; i++);      // 延时循环
72
73         GPBDAT |= (1<<5);             // 关闭LED1
74         for(i=0; i<150000; i++);      // 延时循环
75
76         if (++j>2)
77         {
78             j = 0;
79             Task_Sw_1To2();           // 从Task1切换到Task2
80         }
81     }
82 }
```

```
136 void Task2()
137 {
138     int i = 0;
139     int j = 0;
140
141     GPBDAT &= ~(0xf<<5);              // 点亮LED1-4
142     for(i=0; i<150000; i++);          // 延时循环
143
144     GPBDAT |= (0xf<<5);               // 关闭LED1-4
145     for(i=0; i<150000; i++);          // 延时循环
146
147     while(1)
148     {
149         GPBDAT |= (0xf<<5);           // 关闭LED1-4
150         GPBDAT &= ~(1<<6);            // 点亮LED2
151         for(i=0; i<150000; i++);      // 延时循环
152
153         GPBDAT |= (1<<6);             // 关闭LED2
154         for(i=0; i<150000; i++);      // 延时循环
155
156         if (++j>2)
157         {
158             j = 0;
159             Task_Sw_2To1();           // 从Task2切换到Task1
160         }
161     }
162 }
```

2. 详解设计/实现部分

程序分为 3 大部分，

--初始化

--存储模型

--任务切换

2.1 初始化

程序的初始化很简单，_start 是 CPU reset 后运行的第一段代码。

a. 因为要调用 C 函数，必须要先设置了 SVC 模式下的堆栈(CPU reset 后，默认进入 SVC 模式)；

b. 为了防止 CPU 不断自动重启，需要关闭 WatchDog；

c. 接下来初始化 4 个 LED 灯(关闭 4 个 LED 灯)；

d. 最后，启动第一个任务 Task1。

注意：Task1 堆栈的初始化指针指向#4000，它工作在 SVC 模式中。

初始化代码(_start)如下，

```
21 @*****************************************************************************
22 @ CPU reset后运行的第一段代码。
23 @ 初始化，包括设置SVC模式下的堆栈，关闭WatchDog,初始化LED灯，启动第一个任务Task1。
24 @*****************************************************************************
25 _start:
26     ldr sp, =4000           @ 设置SVC模式下的堆栈
27     bl  disable_watch_dog   @ 关闭WatchDog
28     bl  init_led            @ 初始化LED灯
29
30     ldr lr, =halt_loop      @ 设置lr
31     ldr pc, =Task1          @ 启动第一个任务Task1
32 halt_loop:
33     b   halt_loop
```

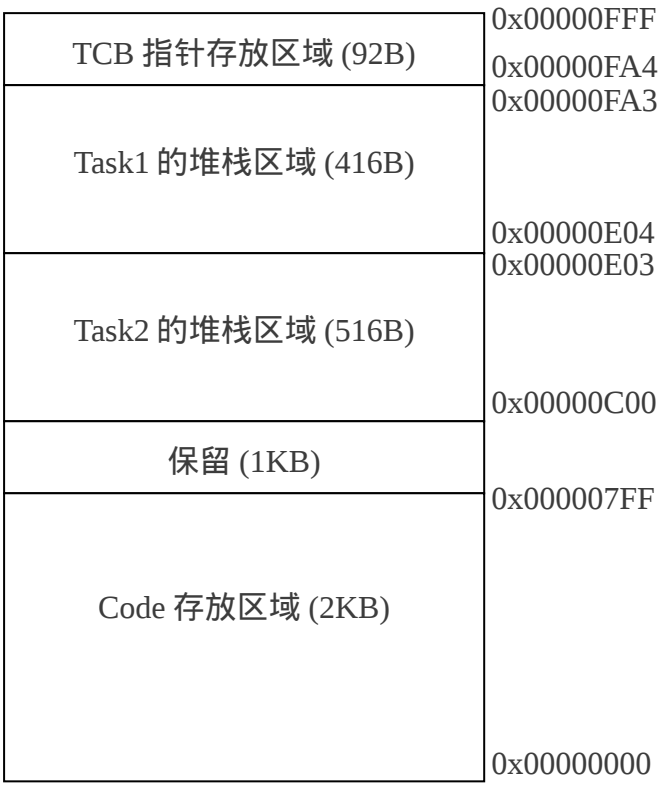
2.2 存储模型

Code 存放区域：　　0x00000000~0x000007FF (2KB)；
Task2 的堆栈区域：　0x00000C00~0x00000E03(516B)；
Task1 的堆栈区域：　0x00000E04~0x00000FA3(416B)；
TCB 指针存放区域：　0x00000FA4~0x00000FFF(92B)。

注意：

a. 栈的增长方向由高地址到低地址；
b. TCB 指针存放区域，存放的是 Task 栈顶指针，如 0x0000FFC 保存 Task1 的栈顶指针；0x00000FF8 保存 Task2 的栈顶指针。

存储划分图表如下，



2.3 任务切换

运行新任务之前，必须要为每个新任务初始化 PCB(Process Control Block)。PCB 是一个保留在 RAM 的数据结构，用于保存所有 ARM 寄存器的一个副本（见下表）。

任务控制块(PCB, Process Control Block)

测试代码中，任务 1 是主程序并首先运行，不需要初始化 PCB。

任务 2 运行之前，需要先初始化 PCB，初始化代码如下，

```
100 void TaskCreate(void(*Task)(void), unsigned long *p_Stack, unsigned long *Task_STK_PTR)
101 {
102     *(p_Stack)   = (unsigned long)Task;          /* 将任务的地址压入堆栈 */
103     *(--p_Stack) = (unsigned long)13;           /* lr  */
104     *(--p_Stack) = (unsigned long)12;           /* r12 */
105     *(--p_Stack) = (unsigned long)11;           /* r11 */
106     *(--p_Stack) = (unsigned long)10;           /* r10 */
107     *(--p_Stack) = (unsigned long)9;            /* r9  */
108     *(--p_Stack) = (unsigned long)8;            /* r8  */
109     *(--p_Stack) = (unsigned long)7;            /* r7  */
110     *(--p_Stack) = (unsigned long)6;            /* r6  */
111     *(--p_Stack) = (unsigned long)5;            /* r5  */
112     *(--p_Stack) = (unsigned long)4;            /* r4  */
113     *(--p_Stack) = (unsigned long)3;            /* r3  */
114     *(--p_Stack) = (unsigned long)2;            /* r2  */
115     *(--p_Stack) = (unsigned long)1;            /* r1  */
116     *(--p_Stack) = (unsigned long)0;            /* r0  */
117     *(--p_Stack) = (unsigned long)(SVCMODE);    /* CPSR */
118     *Task_STK_PTR = (unsigned long)p_Stack;     /* 保存Task栈顶地址*/
119 }
```

其中，函数“TaskCreate”的 3 个参数含义如下：

void(*Task)(void):	Task 地址
unsigned long *p_Stack:	Task 堆栈指针
unsigned long *Task_STK_PTR:	保存 Task 栈顶指针的地址

下面以 Task1 切换到 Task2 为例，来说明任务切换的过程。

第一阶段，保存 Task1 运行的上下文(Context)。

将 PC/LR/R0-R12/CPSR 压入堆栈中，然后将栈顶指针保存起来。代码如下，

```
35 @*******************************************************************************
36 @ 从Task1切换到Task2
37 @*******************************************************************************
38 Task_Sw_1To2:
39     stmfd sp!,{lr}              @ PC 入栈
40     stmfd sp!,{r0-r12,lr}       @ r0-r12，lr入栈
41
42     mrs r4,cpsr
43     stmfd sp!,{r4}              @ cpsr入栈
44
45     ldr r5,=Task1_STK_PTR       @ 取出存放Task1的Stack Pointer的地址
46     str sp,[r5]                 @ 保存Task1的Stack Pointer
```

第二阶段，恢复 Task2 的运行上下文(Context)。

首先取出 Task2 的栈顶指针；然后，PSR 先出栈并恢复 CPSR；接着 R0-R12/LR/PC 依次出栈。

至此，Task2 恢复了之前的运行上下文并接管 CPU 的控制权。

```
48     ldr r6, =Task2_STK_PTR      @ 取出存放Task2的Stack Pointer的地址
49     ldr sp, [r6]                @ 取出Task2的堆顶指针到SP
50     b   POP_ALL                 @ 根据设定的栈结构顺序出栈
```

```
70 @*******************************************************************************
71 @ 根据设定的栈结构顺序出栈
72 @*******************************************************************************
73 POP_ALL:
74     ldmfd sp!,{r4}              @ psr出栈
75     msr CPSR_cxsf,r4
76     ldmfd sp!,{r0-r12,lr,pc}    @ r0-r12,lr,pc出栈
```

简易多任务设计(2)

--基于 TQ2440(ARM9)的实现

作者：姜广伟
日期：2010-12-24
邮箱：Guangwei.jiang@gmail.com

1. 引言

上一个练习中(TaskSwitch)，我们通过“Task_Sw_1To2”和“Task_Sw_2To1”实现了“任务 1”和“任务 2”之间的切换。有没有办法，用统一的函数“Task_Sw”实现不同任务间的切换？

答案是肯定的。如何实现呢？

本章就带领大家一探究竟。

首先，还是先看看这个程序(TaskSwitchEx)的功能：
Task1 中，闪烁 LED1 灯 3 次后，主动将 CPU 使用权切换给其他任务；
Task2 中，闪烁 LED2 灯 3 次后，主动将 CPU 使用权切换给其他任务。
至于这个“其他任务”是谁，就不是 Task1/Task2 所能控制的了。

实验环境说明：

本实验的操作平台为 TQ2440(ARM9)开发板，程序代码在 S3C2440“SteppingStone”的 4KB 内部 RAM 中运行。程序就是利用这 4KB 的空间来实现多任务的切换。

请选择从 NAND Flash 启动，并将生成的二进制文件烧到 NAND Flash 中。

当选择从 NAND Flash 启动 CPU 时，CPU 会通过内部硬件将 NAND Flash 开始的 4KB 字节数据复制到“SteppingStone” 的 4KB 内部 RAM 中（此时内部 RAM 的起始地址是 0），然后跳到地址 0 开始执行。

下面，还是先看看 Task1 和 Task2 的代码，

```
53 void Task1()
54 {
55     int i = 0;
56     int j = 0;
57
58     GPBDAT &= ~(0xf<<5);              // 点亮LED1-4
59     for(i=0; i<150000; i++);          // 延时循环
60
61     GPBDAT |= (0xf<<5);               // 关闭LED1-4
62     for(i=0; i<150000; i++);          // 延时循环
63
64     TaskCreate(Task2, TASK2_SP, TASK2_STK_PTR);
65
66     while(1)
67     {
68         GPBDAT |= (0xf<<5);           // 关闭LED1-4
69         GPBDAT &= ~(1<<5);            // 点亮LED1
70         for(i=0; i<150000; i++);      // 延时循环
71
72         GPBDAT |= (1<<5);             // 关闭LED1
73         for(i=0; i<150000; i++);      // 延时循环
74
75         if (++j>2)
76         {
77             j = 0;
78             Task_Sw();                // 切换到其他任务
79         }
80     }
81 }
```

```
135 void Task2()
136 {
137     int i = 0;
138     int j = 0;
139
140     GPBDAT &= ~(0xf<<5);              // 点亮LED1-4
141     for(i=0; i<150000; i++);          // 延时循环
142
143     GPBDAT |= (0xf<<5);               // 关闭LED1-4
144     for(i=0; i<150000; i++);          // 延时循环
145
146     while(1)
147     {
148         GPBDAT |= (0xf<<5);           // 关闭LED1-4
149         GPBDAT &= ~(1<<6);            // 点亮LED2
150         for(i=0; i<150000; i++);      // 延时循环
151
152         GPBDAT |= (1<<6);             // 关闭LED2
153         for(i=0; i<150000; i++);      // 延时循环
154
155         if (++j>2)
156         {
157             j = 0;
158             Task_Sw();                // 切换到其他任务
159         }
160     }
161 }
```

2. 详解设计/实现部分

程序分为 3 大部分，

--初始化

--存储模型

--任务切换

2.1 初始化

程序的初始化很简单，_start 是 CPU reset 后运行的第一段代码。

a. 因为要调用 C 函数，必须要先设置了 SVC 模式下的堆栈(CPU reset 后，默认进入 SVC 模式)；

b. 为了防止 CPU 不断自动重启，需要关闭 WatchDog；

c. 接下来初始化 4 个 LED 灯(关闭 4 个 LED 灯)；

d. 初始化"下一个任务"的指向“任务 1”, 关于调度算法，后面会讲到；

e. 最后，启动第一个任务 Task1。

注意：Task1 堆栈的初始化指针指向#4000，它工作在 SVC 模式中。

初始化代码(_start)如下，

```
24 @*************************************************************************************
25 @ CPU reset后运行的第一段代码。
26 @ 初始化，包括设置SVC模式下的堆栈，关闭WatchDog,初始化LED灯，初始化"下一个任务"的堆栈指针，
27 @ 最后，启动第一个任务Task1。
28 @*************************************************************************************
29 _start:
30     ldr sp, =4000           @ 设置SVC模式下的堆栈
31     bl  disable_watch_dog   @ 关闭WatchDog
32     bl  init_led            @ 初始化LED灯
33
34     @ "下一个任务"的堆栈指针指向任务1
35     ldr r0, =Task01_STK_PTR
36     ldr r1, =TaskNext_STK_PTR
37     str r0, [r1]            @ mem[r1] =r0
38
39     ldr lr, =halt_loop      @ 设置lr
40     ldr pc, =Task1          @ 启动第一个任务Task1
41 halt_loop:
42     b   halt_loop
```

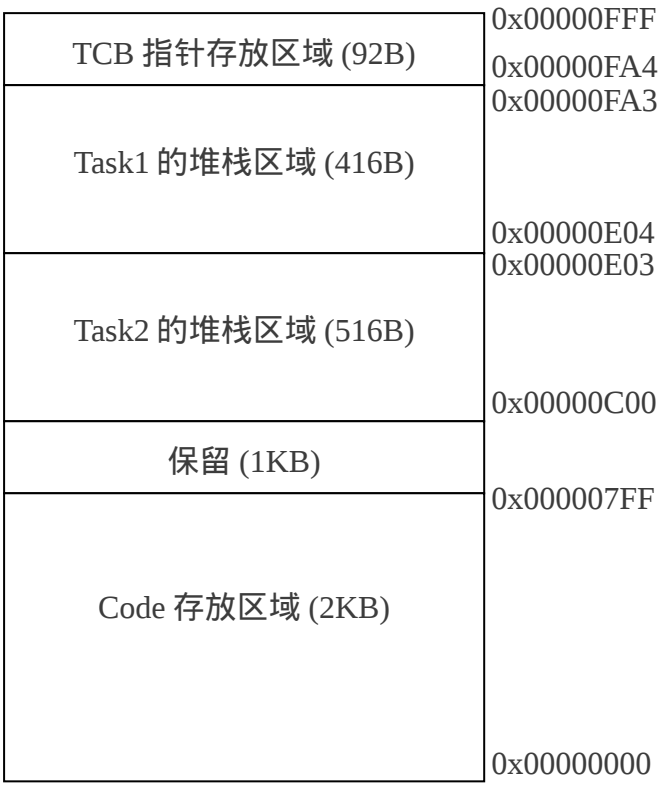
2.2 存储模型

Code 存放区域：　　0x00000000~0x000007FF (2KB)；
Task2 的堆栈区域：　0x00000C00~0x00000E03(516B)；
Task1 的堆栈区域：　0x00000E04~0x00000FA3(416B)；
TCB 指针存放区域：　0x00000FA4~0x00000FFF(92B)。

注意：

c. 栈的增长方向由高地址到低地址；
d. TCB 指针存放区域，存放的是 Task 栈顶指针，如 0x0000FF4 保存 Task1 的栈顶指针；0x00000FF0 保存 Task2 的栈顶指针。

存储划分图表如下，



2.3 任务切换

运行新任务之前，必须要为每个新任务初始化 PCB(Process Control Block)。PCB 是一个保留在 RAM 的数据结构，用于保存所有 ARM 寄存器的一个副本（见下表）。

任务控制块(PCB, Process Control Block)

PC	高地址
LR	
R12	
R11	
R10	
R9	
R8	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	
CPSR	低地址

测试代码中，任务 1 是主程序并首先运行，不需要初始化 PCB。
任务 2 运行之前，需要先初始化 PCB，初始化代码如下，

```
100 void TaskCreate(void(*Task)(void), unsigned long *p_Stack, unsigned long *Task_STK_PTR)
101 {
102     *(p_Stack)   = (unsigned long)Task;          /* 将任务的地址压入堆栈 */
103     *(--p_Stack) = (unsigned long)13;           /* lr  */
104     *(--p_Stack) = (unsigned long)12;           /* r12 */
105     *(--p_Stack) = (unsigned long)11;           /* r11 */
106     *(--p_Stack) = (unsigned long)10;           /* r10 */
107     *(--p_Stack) = (unsigned long)9;            /* r9  */
108     *(--p_Stack) = (unsigned long)8;            /* r8  */
109     *(--p_Stack) = (unsigned long)7;            /* r7  */
110     *(--p_Stack) = (unsigned long)6;            /* r6  */
111     *(--p_Stack) = (unsigned long)5;            /* r5  */
112     *(--p_Stack) = (unsigned long)4;            /* r4  */
113     *(--p_Stack) = (unsigned long)3;            /* r3  */
114     *(--p_Stack) = (unsigned long)2;            /* r2  */
115     *(--p_Stack) = (unsigned long)1;            /* r1  */
116     *(--p_Stack) = (unsigned long)0;            /* r0  */
117     *(--p_Stack) = (unsigned long)(SVCMODE);    /* CPSR */
118     *Task_STK_PTR = (unsigned long)p_Stack;     /* 保存Task栈顶地址*/
119 }
```

其中，函数“TaskCreate”的 3 个参数含义如下：

void(*Task)(void):	Task 地址
unsigned long *p_Stack:	Task 堆栈指针
unsigned long *Task_STK_PTR:	保存 Task 栈顶指针的地址

首先，解释任务调度的原则。
我们采用最简单的“轮转法调度”，即 Task1->Task2->Task3->…
在运行第一个任务之前，要先初始化“下一个任务”，即指向“任务 1”；
任务切换过程中，要计算出“当前任务”和“下一个任务”的指向：
“当前任务”就是上一次调度之前的“下一个任务”；
“下一个任务”就是“当前任务”的 id 加 1，如“当前任务”是“任务 1”，则“下一个任务”是“任务 2”。如果“下一个任务”的 id 大于任务总数（我们也可称之为“越界”），则“下一个任务”指向第一个任务。

接着，介绍两个变量的作用。
TaskCur_STK_PTR：保存“当前任务”堆栈指针的地址；
TaskNext_STK_PTR：保存“下一个任务”堆栈指针的地址。

然后，介绍“任务 x”堆栈指针保存的地址。
在我们的练习中，为了简化难道，任务 x”堆栈指针是保存在连续的地址空间中。如，保存 Task1 堆栈指针的地址是 0x00000FF4; 保存 Task2 堆栈指针的地址是 0x00000FF0。

下面是具体实现，
第一阶段，保存“当前任务”运行的上下文(Context)。
a. 首先，将“当前任务”寄存器值压栈；
b. 接着，计算存放“当前任务”堆栈指针的地址；
c. 最后，保存“当前任务”的栈顶指针。

```
44 @*****************************************************************************
45 @ 任务切换。这里使用最简单的"轮转法调度"！
46 @*****************************************************************************
47 Task_Sw:
48     @ 将“当前任务”寄存器值压栈。
49     stmfd sp!,{lr}                @ PC 入栈
50     stmfd sp!,{r0-r12,lr}         @ r0-r12, lr入栈
51     mrs r4,cpsr
52     stmfd sp!,{r4}                @ cpsr入栈
53
54     @ 计算存放“当前任务”堆栈指针的地址。
55     @ 因为采用最简单的“轮转法调度”，所以，“当前任务”是上一次调度之前的“下一个任务”！
56     ldr r0, =TaskCur_STK_PTR
57     ldr r1, =TaskNext_STK_PTR
58     ldr r1, [r1]
59     str r1, [r0]
60
61     @ 保存“当前任务”的栈顶指针。
62     ldr r5,=TaskCur_STK_PTR       @ 取出存放“当前任务”的堆栈指针的地址
63     ldr r5, [r5]
64     str sp,[r5]                   @ 保存“当前任务”的堆栈指针
```

第二阶段，恢复“下一个任务”的运行上下文(Context)。

a. 首先，计算“下一个任务”堆栈指针；

b. 接着，取出“下一个任务”的栈顶指针；

c. 最后，将寄存器依次出栈。

```
66      @ 计算“下一个任务”堆栈指针。
67      @ “下一个任务”的堆栈指针，保存在“当前任务”堆栈指针的地址的后面(-4)；
68      @ 如果“下一个任务”超出了界限，则指向第一个任务。
69      ldr r0, =TaskNext_STK_PTR
70      ldr r1, =TaskCur_STK_PTR
71      ldr r1, [r1]
72      sub r1, r1, #4                  @"下一个任务"保存地址 = “当前任务”保存地址 - 4
73      str r1, [r0]
74      ldr r2, =TaskMin_STK_PTR
75      ldr r3, =TaskMax_STK_PTR
76      cmp r1, r3
77      strmi r2, [r0]                  @ 如果TaskNext_STK_PTR越界，
78                                      @ 则将TaskMin_STK_PTR的值赋给TaskNext_STK_PTR.
79
80      @ 取出“下一个任务”的栈顶指针，并将寄存器依次出栈。
81      ldr r6, =TaskNext_STK_PTR       @ 取出存放"下一个任务"的Stack Pointer的地址
82      ldr r6, [r6]
83      ldr sp, [r6]                    @ 取出"下一个任务"的堆顶指针,赋给SP
84      b   POP_ALL                     @ 根据设定的栈结构顺序出栈
85
86  @*********************************************************************************
87  @ 根据设定的栈结构顺序出栈
88  @*********************************************************************************
89  POP_ALL:
90      ldmfd sp!,{r4}                  @ psr出栈
91      msr CPSR_cxsf,r4
92      ldmfd sp!,{r0-r12,lr,pc}        @ r0-r12,lr,pc出栈
```

简易内核调度(1)

--基于 TQ2440(ARM9)的实现

作者：姜广伟
日期：2010-12-28
邮箱：Guangwei.jiang@gmail.com

1. 引言

本章的主题是“内核调度程序的设计与实现”，“内核调度”是所有多任务内核的必须模块，调度算法设计的优劣，直接影响到整个系统的效能。

本章的实例，没有考虑系统“效能”，只是为了向大家展示如何设计和实现一个最简单的内核调度程序，我们采用最简单的“轮转法调度”，大家可以尝试采用其他算法来优化内核的调度性能。

首先，还是先看看这个程序(KernelSched)的功能：
Task1 中，闪烁 LED1 灯；
Task2 中，闪烁 LED2 灯。
调度程序根据时钟产生的中断，循环调度 Task1/Task2。

实验环境说明：

本实验的操作平台为 TQ2440(ARM9)开发板，程序代码在 S3C2440“SteppingStone”的 4KB 内部 RAM 中运行。程序就是利用这 4KB 的空间来实现多任务的切换。

请选择从 NAND Flash 启动，并将生成的二进制文件烧到 NAND Flash 中。

当选择从 NAND Flash 启动 CPU 时，CPU 会通过内部硬件将 NAND Flash 开始的 4KB 字节数据复制到“SteppingStone” 的 4KB 内部 RAM 中（此时内部 RAM 的起始地址是 0），然后跳到地址 0 开始执行。

下面，还是先看看 Task1 和 Task2 的代码，

```
49 void Task1()
50 {
51     int i = 0;
52 
53     GPBDAT &= ~(0xf<<5);              // 点亮LED1-4
54     for(i=0; i<150000; i++);          // 延时循环
55 
56     GPBDAT |= (0xf<<5);               // 关闭LED1-4
57     for(i=0; i<150000; i++);          // 延时循环
58 
59     TaskCreate(Task2, TASK2_SP, TASK2_STK_PTR);
60 
61     while(1)
62     {
63         GPBDAT |= (0xf<<5);           // 关闭LED1-4
64         GPBDAT &= ~(1<<5);            // 点亮LED1
65         for(i=0; i<150000; i++);      // 延时循环
66 
67         GPBDAT |= (1<<5);             // 关闭LED1
68         for(i=0; i<150000; i++);      // 延时循环
69     }
70 }
```

```
124 void Task2()
125 {
126     int i = 0;
127 
128     GPBDAT &= ~(0xf<<5);              // 点亮LED1-4
129     for(i=0; i<150000; i++);          // 延时循环
130 
131     GPBDAT |= (0xf<<5);               // 关闭LED1-4
132     for(i=0; i<150000; i++);          // 延时循环
133 
134     while(1)
135     {
136         GPBDAT |= (0xf<<5);           // 关闭LED1-4
137         GPBDAT &= ~(1<<6);            // 点亮LED2
138         for(i=0; i<150000; i++);      // 延时循环
139 
140         GPBDAT |= (1<<6);             // 关闭LED2
141         for(i=0; i<150000; i++);      // 延时循环
142     }
143 }
```

2. 详解设计/实现部分

程序分为 3 大部分，

--初始化

--存储模型

--内核调度

2.1 初始化

CPU 初次加电后，跳到中断向量表中执行 Reset 分支。

a. 因为要调用 C 函数，必须要先设置了 SVC 模式下的堆栈(CPU reset 后，默认进入 SVC 模式)；

b. 为了防止 CPU 不断自动重启，需要关闭 WatchDog；

c. 分别设置中断/系统模式下的栈指针；

d. 初始化 clock/LED/Timer0/Irq，开 IRQ 中断；

e. 初始化"下一个任务"的指向“任务 1”, 关于调度算法，后面会讲到；

f. 最后，启动第一个任务 Task1。

初始化代码(_start)如下，

```
59 Reset:
60     ldr sp, =4000            @ 设置SVC模式下的堆栈
61     bl  disable_watch_dog    @ 关闭WatchDog
62
63     msr cpsr_c, #0xd2        @ 进入中断模式
64     ldr sp, =2560            @ 设置中断模式栈指针
65
66     msr cpsr_c, #0xdf        @ 进入系统模式
67     ldr sp, =4000            @ 设置系统模式栈指针
68
69     bl  clock_init           @ 设置MPLL，改变FCLK、HCLK、PCLK
70     bl  init_led             @ 初始化LED灯
71     bl  timer0_init          @ 初始化定时器0
72     bl  init_irq             @ 调用中断初始化函数，在init.c中
73     msr cpsr_c, #0x5f        @ 设置I-bit=0，开IRQ中断
74
75     @ "下一个任务"的堆栈指针指向任务1
76     ldr r0, =Task01_STK_PTR
77     ldr r1, =TaskNext_STK_PTR
78     str r0, [r1]             @ mem[r1] =r0
79
80     ldr lr, =halt_loop       @ 设置lr
81     ldr pc, =Task1           @ 启动第一个任务Task1
82 halt_loop:
83     b   halt_loop
```

2.2 存储模型

Code 存放区域：　　0x00000000~0x000007FF (2KB)；
IRQ 栈区域：　　0x00000800~0x000009FF(512B);
Task2 的堆栈区域：　0x00000C00~0x00000E03(516B)；
Task1 的堆栈区域：　0x00000E04~0x00000FA3(416B)；
TCB 指针存放区域：　0x00000FA4~0x00000FFF(92B)。

注意：

e. 栈的增长方向由高地址到低地址；
f. TCB 指针存放区域，存放的是 Task 栈顶指针，如 0x0000FF4 保存 Task1 的栈顶指针；0x00000FF0 保存 Task2 的栈顶指针。

存储划分图表如下，

2.3 内核调度

运行新任务之前，必须要为每个新任务初始化 PCB(Process Control Block)。PCB 是一个保留在 RAM 的数据结构，用于保存所有 ARM 寄存器的一个副本（见下表）。

任务控制块(PCB, Process Control Block)

测试代码中，任务 1 是主程序并首先运行，不需要初始化 PCB。
任务 2 运行之前，需要先初始化 PCB，初始化代码如下，

```
100 void TaskCreate(void(*Task)(void), unsigned long *p_Stack, unsigned long *Task_STK_PTR)
101 {
102     *(p_Stack)   = (unsigned long)Task;          /* 将任务的地址压入堆栈 */
103     *(--p_Stack) = (unsigned long)13;           /* lr  */
104     *(--p_Stack) = (unsigned long)12;           /* r12 */
105     *(--p_Stack) = (unsigned long)11;           /* r11 */
106     *(--p_Stack) = (unsigned long)10;           /* r10 */
107     *(--p_Stack) = (unsigned long)9;            /* r9  */
108     *(--p_Stack) = (unsigned long)8;            /* r8  */
109     *(--p_Stack) = (unsigned long)7;            /* r7  */
110     *(--p_Stack) = (unsigned long)6;            /* r6  */
111     *(--p_Stack) = (unsigned long)5;            /* r5  */
112     *(--p_Stack) = (unsigned long)4;            /* r4  */
113     *(--p_Stack) = (unsigned long)3;            /* r3  */
114     *(--p_Stack) = (unsigned long)2;            /* r2  */
115     *(--p_Stack) = (unsigned long)1;            /* r1  */
116     *(--p_Stack) = (unsigned long)0;            /* r0  */
117     *(--p_Stack) = (unsigned long)(SVCMODE);    /* CPSR */
118     *Task_STK_PTR = (unsigned long)p_Stack;     /* 保存Task栈顶地址*/
119 }
```

其中，函数“TaskCreate”的 3 个参数含义如下：

void(*Task)(void):	Task 地址
unsigned long *p_Stack:	Task 堆栈指针
unsigned long *Task_STK_PTR:	保存 Task 栈顶指针的地址

首先，解释任务调度的原则。
我们采用最简单的“轮转法调度”，即 Task1->Task2->Task3->…
在运行第一个任务之前，要先初始化“下一个任务”，即指向“任务 1”；
任务切换过程中，要计算出“当前任务”和“下一个任务”的指向：
“当前任务”就是上一次调度之前的“下一个任务”；
“下一个任务”就是“当前任务”的 id 加 1，如“当前任务”是“任务 1”，则“下一个任务”是“任务 2”。如果“下一个任务”的 id 大于任务总数（我们也可称之为“越界”），则“下一个任务”指向第一个任务。

接着，介绍两个变量的作用。
TaskCur_STK_PTR：保存“当前任务”堆栈指针的地址；
TaskNext_STK_PTR：保存“下一个任务”堆栈指针的地址。

然后，介绍“任务 x”堆栈指针保存的地址。
在我们的练习中，为了简化难道，任务 x”堆栈指针是保存在连续的地址空间中。如，保存 Task1 堆栈指针的地址是 0x00000FF4; 保存 Task2 堆栈指针的地址是 0x00000FF0。

下面是具体实现，
第一阶段，处理 Timer0 中断。我们要求 Time0 中断发生后，要执行 KernelSched，而不是返回中断发生前的函数。
首先，将前一种模式的寄存器压栈；
接着，把中断发生前的 PC 指针保存起来，以备之后使用；
然后，在堆栈中，用 KernelSched 替换中断发生前的 PC 指针，使中断后能直接跳到 KernelSched；
最后，寄存器值出栈，PC 跳转到 KernelSched。

```
 86 HandleIRQ:
 87     sub lr, lr, #4                   @ 计算返回地址
 88     stmdb   sp!,    { r0-r12,lr }    @ 保存使用到的寄存器
 89                                      @ 注意，此时的sp是中断模式的sp
 90     @ 我们假定只有Time0中断。
 91     @ 我们要求Time0中断发生后，要执行KernelSched，而不是返回中断发生前的函数，为此要做：
 92     @ a. 把中断发生前的PC指针保存起来，以备之后使用；
 93     @ b. 在堆栈中，用KernelSched替换中断发生前的PC指针，使中断后能直接跳到KernelSched!
 94     mov r4, sp                       @ r4 = 堆栈指针
 95     add r5, r4, #52                  @ +52，找到PC(进入intrrupt之前模式的)存储的位置
 96     ldr r6, [r5]                     @ 将PC(进入intrrupt之前模式的)指针值载入到r6
 97     ldr r7, =PC_TEMP_PTR             @ r7 = PC指针暂时存放地址
 98     str r6, [r7]                     @ 将PC(进入intrrupt之前模式的)暂存
 99     ldr r8, =KernelSched             @ r8 = KernelSched
100     str r8, [r5]                     @ 将PC指针置换成KernelSched
101
102     ldr lr, =int_return              @ 设置调用ISR即EINT_Handle函数后的返回地址
103     ldr pc, =Timer0_Handle           @ 调用中断服务函数，在interrupt.c中
104
105 int_return:
106     ldmia   sp!,    { r0-r12, pc }^  @ 中断返回，^表示将spsr的值复制到cpsr
```

第二阶段，保存“当前任务”运行的上下文(Context)。

d. 首先，将“当前任务”寄存器值压栈；

e. 接着，取出中断发生前的 PC 值，并将它压入栈中；

f. 然后，计算存放“当前任务”堆栈指针的地址；

g. 最后，保存“当前任务”的栈顶指针。

```
111 KernelSched:
112     @ 将“当前任务”寄存器值压栈。
113     stmfd sp!,{lr}              @ PC 入栈
114     stmfd sp!,{r0-r12,lr}       @ r0-r12，lr入栈
115
116     @ 取出中断发生前的PC值，并将它压入栈中
117     mov r4, sp                  @ r4 = 堆栈指针
118     add r5, r4, #56             @ +56，找到PC(进入intrrupt之前模式的)存储的位置
119     ldr r6, =PC_TEMP_PTR        @ r6 = PC指针暂时存放地址
120     ldr r7, [r6]                @ r7 = 进入intrrupt之前模式的PC值
121     str r7, [r5]                @ 将PC指针置换成进入intrrupt之前模式的PC值
122
123     mrs r4,cpsr
124     stmfd sp!,{r4}              @ cpsr入栈
125
126     @ 计算存放“当前任务”堆栈指针的地址。
127     @ 因为采用最简单的“轮转法调度”，所以，“当前任务”是上一次调度之前的“下一个任务”！
128     ldr r0, =TaskCur_STK_PTR
129     ldr r1, =TaskNext_STK_PTR
130     ldr r1, [r1]
131     str r1, [r0]
132
133     @ 保存“当前任务”的栈顶指针。
134     ldr r5,=TaskCur_STK_PTR     @ 取出存放“当前任务”的堆栈指针的地址
135     ldr r5, [r5]
136     str sp,[r5]                 @ 保存“当前任务”的堆栈指针
```

第三阶段，恢复“下一个任务”的运行上下文(Context)。

d. 首先，计算“下一个任务”堆栈指针；

e. 接着，取出“下一个任务”的栈顶指针；

f. 最后，将寄存器依次出栈。

```
138     @ 计算“下一个任务”堆栈指针。
139     @ “下一个任务”的堆栈指针，保存在“当前任务”堆栈指针的地址的后面(-4)；
140     @ 如果“下一个任务”超出了界限，则指向第一个任务。
141     ldr r0, =TaskNext_STK_PTR
142     ldr r1, =TaskCur_STK_PTR
143     ldr r1, [r1]
144     sub r1, r1, #4              @"下一个任务"保存地址 = “当前任务”保存地址 - 4
145     str r1, [r0]
146     ldr r2, =TaskMin_STK_PTR
147     ldr r3, =TaskMax_STK_PTR
148     cmp r1, r3
149     strmi r2, [r0]              @ 如果TaskNext_STK_PTR越界，
150                                 @ 则将TaskMin_STK_PTR的值赋给TaskNext_STK_PTR.
151
152     @ 取出“下一个任务”的栈顶指针，并将寄存器依次出栈。
153     ldr r6, =TaskNext_STK_PTR   @ 取出存放"下一个任务"的Stack Pointer的地址
154     ldr r6, [r6]
155     ldr sp, [r6]                @ 取出"下一个任务"的堆顶指针,赋给SP
156     b   POP_ALL                 @ 根据设定的栈结构顺序出栈
157
158 @*******************************************************************************
159 @ 根据设定的栈结构顺序出栈
160 @*******************************************************************************
161 POP_ALL:
162     ldmfd sp!,{r4}              @ psr出栈
163     msr CPSR_cxsf,r4
164     ldmfd sp!,{r0-r12,lr,pc}    @ r0-r12,lr,pc出栈
```

简易内核调度(2)

--基于 TQ2440(ARM9)的实现

作者：姜广伟
日期：2010-12-28
邮箱：Guangwei.jiang@gmail.com

1. 引言

本章，我们继续完善 KernelSched 这个程序(本章程序名为 KernelSchedEx)：

a. 加入 Critical Section, 防止同时操作 LED，造成 LED1 和 LED2 同时点亮；

b. 在 HandleIRQ 中，加入对 Timer0 中断源的判断。

实验环境说明：

本实验的操作平台为 TQ2440(ARM9)开发板，程序代码在 S3C2440“SteppingStone”的 4KB 内部 RAM 中运行。程序就是利用这 4KB 的空间来实现多任务的切换。

请选择从 NAND Flash 启动，并将生成的二进制文件烧到 NAND Flash 中。

当选择从 NAND Flash 启动 CPU 时，CPU 会通过内部硬件将 NAND Flash 开始的 4KB 字节数据复制到“SteppingStone” 的 4KB 内部 RAM 中（此时内部 RAM 的起始地址是 0），然后跳到地址 0 开始执行。

下面，还是先看看 Task1 和 Task2 的代码，

```
54 void Task1()
55 {
56     int i = 0;
57 
58     GPBDAT &= ~(0xf<<5);              // 点亮LED1-4
59     for(i=0; i<150000; i++);          // 延时循环
60 
61     GPBDAT |= (0xf<<5);               // 关闭LED1-4
62     for(i=0; i<150000; i++);          // 延时循环
63 
64     TaskCreate(Task2, TASK2_SP, TASK2_STK_PTR);
65 
66     while(1)
67     {
68         EnterCriticalSection();       // 进入临界区
69 
70         GPBDAT |= (0xf<<5);           // 关闭LED1-4
71         GPBDAT &= ~(1<<5);            // 点亮LED1
72         for(i=0; i<150000; i++);      // 延时循环
73 
74         GPBDAT |= (1<<5);             // 关闭LED1
75         for(i=0; i<150000; i++);      // 延时循环
76 
77         ExitCriticalSection();        // 退出临界区
78     }
79 }
```

```
133 void Task2()
134 {
135     int i = 0;
136 
137     GPBDAT &= ~(0xf<<5);              // 点亮LED1-4
138     for(i=0; i<150000; i++);          // 延时循环
139 
140     GPBDAT |= (0xf<<5);               // 关闭LED1-4
141     for(i=0; i<150000; i++);          // 延时循环
142 
143     while(1)
144     {
145         EnterCriticalSection();       // 进入临界区
146 
147         GPBDAT |= (0xf<<5);           // 关闭LED1-4
148         GPBDAT &= ~(1<<6);            // 点亮LED2
149         for(i=0; i<150000; i++);      // 延时循环
150 
151         GPBDAT |= (1<<6);             // 关闭LED2
152         for(i=0; i<150000; i++);      // 延时循环
153 
154         ExitCriticalSection();        // 退出临界区
155     }
156 }
```

2. 详解改进部分

--加入 Critical Section, 防止同时操作 LED，造成 LED1 和 LED2 同时点亮；

--在 HandleIRQ 中，加入对 Timer0 中断源的判断。

2.1 加入 Critical Section, 防止同时操作 LED，造成 LED1 和 LED2 同时点亮

a. 进入临界区，关闭 Irq 中断；

b. 退出临界区，恢复 Irq 中断。

```
@ 进入临界区
EnterCriticalSection:
    stmfd sp!, {r0-r4, lr}      @ 保存寄存器
    mrs r0, CPSR                @ 将CPSR的值保存到r0
    ldr r3, =CPSR_TEMP_PTR
    str r0, [r3]                @ 保存CPSR的值
    ldr r1, =NOINT
    orr r2, r0, r1              @ 将CPSR的"I"位置1
    msr CPSR_c, r2              @ 设置CPSR的控制位
    ldmfd sp!, {r0-r4, pc}      @ 寄存器恢复

@ 退出临界区
ExitCriticalSection:
    stmfd sp!, {r0-r4, lr}      @ 保存寄存器
    ldr r0, =CPSR_TEMP_PTR
    ldr r0, [r0]                @ 获取保存的CPSR的值
    msr CPSR_c, r0              @ 恢复CPSR的值
    ldmfd sp!, {r0-r4, pc}      @ 寄存器恢复
```

2.2 在 HandleIRQ 中，加入对 Timer0 中断源的判断

读取“Interrupt Offset Register”，判断是否是 Timer0 中断源，若是，则跳转到 Timer0_Int，执行内核调度。

```
 94 HandleIRQ:
 95     sub lr, lr, #4                   @ 计算返回地址
 96     stmdb   sp!,    { r0-r12,lr }    @ 保存使用到的寄存器
 97                                      @ 注意，此时的sp是中断模式的sp
 98     @ 检查是否Timer0中断，若是，则跳转到Timer0_Int
 99     ldr r0, =INTOFFSET
100     ldr r0, [r0]
101     ldr r1, =INT_TIMER0_OFFSET
102     cmp r0, r1
103     beq Timer0_Int
104
105     @ 处理其他类型中断
106
107 int_return:
108     ldmia   sp!,    { r0-r12, pc }^  @ 中断返回，^表示将spsr的值复制到cpsr
109
110 Timer0_Int:
111     @ 我们要求Time0中断发生后，要执行KernelSched，而不是返回中断发生前的函数，为此要做：
112     @ a. 把中断发生前的PC指针保存起来，以备之后使用；
113     @ b. 在堆栈中，用KernelSched替换中断发生前的PC指针，使中断后能直接跳到KernelSched!
114     mov r4, sp                       @ r4 = 堆栈指针
115     add r5, r4, #52                  @ +52，找到PC(进入intrrupt之前模式的)存储的位置
116     ldr r6, [r5]                     @ 将PC(进入intrrupt之前模式的)指针值载入到r6
117     ldr r7, =PC_TEMP_PTR             @ r7 = PC指针暂时存放地址
118     str r6, [r7]                     @ 将PC(进入intrrupt之前模式的)暂存
119     ldr r8, =KernelSched             @ r8 = KernelSched
120     str r8, [r5]                     @ 将PC指针置换成KernelSched
121
122     ldr lr, =int_return              @ 设置调用ISR即EINT_Handle函数后的返回地址
123     ldr pc, =Timer0_Handle           @ 调用中断服务函数，在interrupt.c中
```