C 语言介绍

(第二篇)

轻量级线程库 protothread

周华军

2202877@163.com

这篇文章源于我个人的计划，在使用了几年 C 语言之后，略有感悟，打算总结一下，也锻炼一下自己的文笔，写出来与大家分享。

如果你能从中学到你以前不知道的知识，那就最好了。如果你感觉本文有哪些不正确或者不妥当的地方，也希望你能告诉我。

下面是对应 CSDN 文章地址：在这里会有其他一些文章的链接，大家也可以在这里发表意见和建议。

http://blog.csdn.net/zhouhuajun/article/details/7589379

周华军

2202877@163.com

2012 年 5 月 26 日

目录

1 前言

一般单片机程序的主框架都是一个大循环，在循环里做各种各样的事情，接收外部数据、扫描键盘、显示等等。理想情况下我们希望这些任务能同时执行，但由于 CPU 的工作方式是线性的，在同一个时刻下只能做一件事情。所以我们只好把这些任务放在一个大循环中，让 cpu 不间断循环的执行，这样在宏观上看来可以认为这些任务是同时执行的。

代码 1: 大循环

```
1 void main(){
2     init(); // 初始化
3 
4     while(1){
5         task1(); //任务1
6         task2(); //任务2
7         task3(); //任务3
8     }
9 }
```

这种方式简单高效，几乎不需要额外的硬件资源。但是它的实时性比较难保证，因为有些任务是有时间限制的，比如扫描按键时间间隔太长，就会感觉到延迟；接收到串口数据可能也需要在一定时间里及时处理；系统在检测到有异常情况时需要马上执行异常处理代码。在这些情况下，在上面的大循环结构显然不能满足要求，因为在这种结构中所有任务都必须按顺序轮流执行，不管哪个任务有多紧急，都不能插队。

引入 RTOS 可以很好地解决这个问题，RTOS 最核心的功能就是任务管理，可以每次让最紧急的任务投入运行，甚至在一个任务运行过程中，如果出现一个紧急任务,RTOS 可以先让运行中得任务退出运行，让紧急任务先运行。这样可以很好地保证任务的实时性。当然任务管理功能是 RTOS 的功能之一，通常 RTOS 还提供消息邮箱，消息队列，信号量，时间管理等功能。完善的任务管理和任务间通信功能可以使软件更加模块化，很好地简化开发和维护的复杂度。

但是这些功能不是没有代价的，RTOS 自身的代码和数据结构需要占用一定的代码和数据存储区；而且要实现多任务，就必须具有保存任务状态的功能，因为当一个任务可能在运行之中退出让其他任务投入运行，但是当这个任务再次投入运行时应该从被打断的点之后继续运行。所以应该在任务退出时保持这个任务当前的状态，当任务再次运行时应将这些状态恢复过来，这样可以让任务看起来跟没被打断过一样。

一个任务的完整状态包括所有的寄存器和栈。在实际 RTOS 中每个任务都要有一个任务堆栈，用于保存任务状态信息，但是任务堆栈的大小是很难确定的。一个 CPU 寄存器的个数是确定的，但是栈的大小和函数调用深度，局部变量多少有关，甚至可能在任务运行中间发生中断的话，中断处理函数也要占用一些栈空间。所以我们很难预先知道一个任务该需要多大的栈，在给任务分配任务栈的时候我们只能给它分配足够大的栈，这会使得 RTOS 占用比较大的 RAM。

RTOS 在移植的时候通常都需要编写一部分汇编代码，因为一般情况下 C 语言不能访问 CPU 的寄存器。而我们在任务切换的时候需要将寄存器的值存入任务栈中。所以我们在使用一个 RTOS 之前需要做一个移植工作，虽然移植所需要的汇编代码不多，但通常都需要对 CPU 结构，汇编，C 语言以及编译器比较了解的情况下才能做到。

相比较而言 protothreads 就简单的多，可以说是非常简单，protothreads 可以以非常少的代价实现简单的多任务机制。运行时需要占用的 RAM 和任务切换时间都非常少。这个线程库非常简单，总共也就几十行的代码。但已经提供了比较完善的功能了。当然简单是它的有点也是它的缺点，简单意味着功能的缺乏，使用它也有诸多的限制。我们需要了解它的优点和缺点，才可以在合适的地方正确使用它。由于这个库的源代码非常简单，我们完全可以理解源代码之后再去用它。下面我们就了解一下它的源代码。完整源代码可以从这个网站上下载:

http://dunkels.com/adam/pt/

2 源码解析

protothreads 的源代码非常简单，实际上仅仅只有两个简单的头文件，我们先看文件 lc-swtich.h，这个文件虽然只有 5 行代码，当可以说是 protothreads 的内核了，我们这里就可以看到它的实现原理。

代码 2: lc-switch.h

```
1 typedef unsigned short lc_t;
2
3 #define LC_INIT(s) s = 0;
4
5 #define LC_RESUME(s) switch(s) { case 0:
6
7 #define LC_SET(s) s = __LINE__; case __LINE__:
8
9 #define LC_END(s) }
```

初一看可能觉得很奇怪，不理解。不着急，为了让大家以最快的速度理解它的工作原理。下面先举一个使用它的例子，为了使这个例子尽可能的简单，我重定义了 LC_SET 宏，下面先看示例代码

代码 3: test-lc.c

```
1  #include <stdio.h>
2  #include "pt\lc-switch.h"
3
4  #undef LC_SET
5  #define LC_SET(s) s = __LINE__; return; case __LINE__:
6
7  lc_t s;
8  void task(void){
9      LC_RESUME(s);
10         printf("hello␣1\n");
11     LC_SET(s);
```

```
12          printf("hello␣2\n");
13      LC_SET(s);
14          printf("hello␣3\n");
15      LC_END(s);
16  }
17  int main(int argc,char *argv[]){
18      LC_INIT(s);
19      task();
20      printf("===␣in␣main␣===\n");
21      task();
22      printf("===␣in␣main␣===\n");
23      task();
24      printf("===␣in␣main␣===\n");
25  }
```

我们在 main 函数中调用 3 次 task 函数，下面是这个示例程序的输出结果。

代码 4: test-lc.c的输出结果

```
1  hello 1
2  === in main ===
3  hello 2
4  === in main ===
5  hello 3
6  === in main ===
```

这个例子说明什么？我们实现了保存任务状态的的功能，前面提到过，实现多任务的关键就在于保存任务退出时的状态，在任务下一次进入运行时任务要按照退出时的状态继续往下运行。使得任务看起来没被打断过一样。

从这个角度上看，在这个例子中我们把 task 函数打印三个字符串看成是一个任务的话，那么这个任务的执行中断过两次，每次中断之后都从

中断点往后继续执行了，所以虽然中断了两次但还是连续地输出了三个字符串。在代码中 LC_SET 所处的位置是一个断点，当任务运行到这里的时候，就会退出，让其他任务运行。而这个任务再次运行时就会从这个断点的下一条语句开始执行。

要理解这个例子的工作原理，我们把所有的宏都展开，再看起来应该就一目了然了。

代码 5: test-lc.i

```
1  lc_t s;
2  void task(void){
3  //------- LC_RESUME ------
4      switch(s) {
5          case 0:;
6  //------------------------
7              printf("hello␣1\n");
8
9  //------- LC_SET ---------
10             s = 11;
11             return;
12         case 11:;
13 //------------------------
14             printf("hello␣2\n");
15 //------- LC_SET ---------
16             s = 13;
17             return;
18         case 13:;
19 //------------------------
20             printf("hello␣3\n");
21 //------- LC_END ---------
22     };
23 //------------------------
```

```
24
25 }
26 int main(int argc,char *argv[]){
27  s = 0;;
28  task();
29  printf("===␣in␣main␣===\n");
30  task();
31  printf("===␣in␣main␣===\n");
32  task();
33  printf("===␣in␣main␣===\n");
34 }
```

看到了吗，实际上 LC_RESUME,LC_SET,LC_END 组成一个 switch 结构，把整个任务包在其中，每次退出时 LC_SET 将当前的行号保存在全局变量 s 中，当下次运行时 switch 语句就会根据 s 的值直接跳到 LC_SET 的下一条语句。所以我们每个任务只需要额外的一个整型变量保存行号就可以了。好理解吧。

lc-switch.h 中定义的宏是非常基础的，不是直接给用户用的。上面的例子也看到，我们为了举一个直接使用 lc-switch.h 的例子，还重定义了一下 LC_SET 宏。下面我们看一下文件 pt.h，它在 lc-switch.h 的基础上定义了一系列的宏，这些宏是直接面对用户的。在 pt.h 中封装了 struct pt 结构，这个结构也就是 protothreads 的任务控制块了，看它的定义就知道，struct pt 结构内部就一个成员 lc 用于包存行号。`struct pt {lc_t lc; };`

我们首先看四个宏:PT_INIT,PT_BEGIN,PT_WAIT_UNTIL,PT_END 的定义，然后再看 protothreads 自带的一个例子 example-small.c。

代码 6: pt.h(部分)

```
1 #define PT_INIT(pt) LC_INIT((pt)->lc)
2
```

```
3  #define PT_BEGIN(pt) { char PT_YIELD_FLAG = 1; LC_RESUME((pt)
       ->lc)
4
5  #define PT_WAIT_UNTIL(pt, condition) \
6    do {          \
7      LC_SET((pt)->lc);      \
8      if(!(condition)) {      \
9        return PT_WAITING;     \
10     }          \
11   } while(0)
12
13
14 #define PT_END(pt) LC_END((pt)->lc); PT_YIELD_FLAG = 0; \
15                   PT_INIT(pt); return PT_ENDED; }
```

这四个宏结合 lc-switch.h(代码 2) 中的四个宏来看的话，应该不难理解。

PT_INIT 宏是线程初始化宏，在线程开始运行之前，需要调用该宏初始化线程控制块，它直接调用 LC_INIT 宏其实也就是将线程控制块保存的行号初始化为 0，这样使得线程从头开始执行。

PT_BEGIN 宏是 LC_RESUME 的封装，表示线程的开始，在线程中它必须放在最开始处，这是因为 protothreads 是通过一个 switch 结构实现多线程编程的，PT_BEGIN 是构成 swtich 结构的起始部分。

PT_WAIT_UNTIL 宏是阻塞这个线程直到某个条件成立。它先调用 LC_SET 宏设置一个断点，然后判断条件是否成立，如果不成立直接返回，当线程第二次进入时会直接跳到这个 if 语句上继续判断条件是否成立。直到条件成立才继续运行下面的代码。

PT_END 宏先调用 LC_END 宏结束整个 switch 结构，然后调用 LC_INIT 宏重新初始化线程控制块，这样在下次运行时可以从头开始，

在它之后就不应该调用类似 PT_WAIT_UNTIL 宏了，因为不能构成 swtich 结构，如果在 PT_END 之后调用，编译器是会报错的。

3 使用例子

因为 protothreads 是由一系列的宏定义组成，理解它的最好办法就是对比分析宏展开后的源代码。在 linux 中可以用 gcc -E 选项获得预处理后的源代码。我们下面看一下 protothreads 源码中的一个示例程序。

代码 7: example-small.c

```
1  static int protothread1_flag, protothread2_flag;
2
3  static int
4  protothread1(struct pt *pt)
5  {
6    PT_BEGIN(pt);
7
8    while(1) {
9      PT_WAIT_UNTIL(pt, protothread2_flag != 0);
10     printf("Protothread␣1␣running\n");
11
12     protothread2_flag = 0;
13     protothread1_flag = 1;
14   }
15
16   PT_END(pt);
17 }
18
19
20 static int
21 protothread2(struct pt *pt)
```

```
22 {
23   PT_BEGIN(pt);
24
25   while(1) {
26     protothread2_flag = 1;
27     PT_WAIT_UNTIL(pt, protothread1_flag != 0);
28     printf("Protothread␣2␣running\n");
29
30     protothread1_flag = 0;
31   }
32   PT_END(pt);
33 }
34
35 static struct pt pt1, pt2;
36 int
37 main(void)
38 {
39   PT_INIT(&pt1);
40   PT_INIT(&pt2);
41
42   while(1) {
43     protothread1(&pt1);
44     protothread2(&pt2);
45   }
46 }
```

由于这个例子比较简单，我把注释去掉了，我们看在 main 函数中初始化两个线程控制块 pt1，pt2 之后就在一个 while 死循环里不停地调度两个线程，在线程中用两个标志 protothread1_flag,protothread2_flag 来协调两个线程，使他们能交错执行，

当然这个示例程序纯粹是为了演示 protothreads 的基本使用方法，因为我们只要在 main 函数中简单写两个 printf 语句就可以实现这个示例程序的功能。

看完了第一个例子，我们在看看 pt.h 除了上面 (代码 6) 列出的四个宏的其他宏。

代码 8: pt.h(部分)

```
1  #define PT_WAIT_WHILE(pt, cond) PT_WAIT_UNTIL((pt), !(cond))
2  #define PT_WAIT_THREAD(pt, thread) PT_WAIT_WHILE((pt),
       PT_SCHEDULE(thread))
3  #define PT_SPAWN(pt, child, thread)  \
4    do {      \
5      PT_INIT((child));    \
6      PT_WAIT_THREAD((pt), (thread));  \
7    } while(0)
8
9  #define PT_RESTART(pt)    \
10   do {      \
11     PT_INIT(pt);    \
12     return PT_WAITING;   \
13   } while(0)
14
15 #define PT_EXIT(pt)    \
16   do {      \
17     PT_INIT(pt);    \
18     return PT_EXITED;   \
19   } while(0)
20
21 #define PT_SCHEDULE(f) ((f) < PT_EXITED)
22 #define PT_YIELD(pt)    \
23   do {      \
```

```
24      PT_YIELD_FLAG = 0;      \
25      LC_SET((pt)->lc);       \
26      if(PT_YIELD_FLAG == 0) {    \
27        return PT_YIELDED;    \
28      }       \
29    } while(0)
30  #define PT_YIELD_UNTIL(pt, cond)   \
31    do {        \
32      PT_YIELD_FLAG = 0;      \
33      LC_SET((pt)->lc);       \
34      if((PT_YIELD_FLAG == 0) || !(cond)) { \
35        return PT_YIELDED;    \
36      }       \
37    } while(0)
```

PT_WAIT_WHILE 这个宏和 PT_WAIT_UNTIL 类似，但条件相反，PT_WAIT_UNTIL 的意思是一直等待直到条件成立，而 PT_WAIT_WHILE 是只要条件成立就一直等待。我们从名字上也能看出来这个差别。

PT_WAIT_THREAD 这个宏让线程停止执行，等待另一个线程执行完才继续往下执行，它调用 PT_WAIT_WHILE 宏等待其它线程结束，PT_SCHEDULE 宏在线程退出时会返回非真值，可以让 PT_WAIT_WHILE 退出。

PT_SPAWN 这个宏分离出一个子线程并等到子线程执行结束，它和 PT_WAIT_THREAD 是类似的，只不过它多了一步初始化子线程的动作。所以它有多了一个 child 参数，这个参数是子线程的线程控制块，因为我们只能通过这个线程控制块才可以初始化子线程。

PT_RESTART 这个宏可以重新初始化本线程，然后退出，当线程再次运行时就从头开始运行了。

PT_EXIT 这个宏是线程退出，与 PT_END 不同，PT_END 是在线程的最后结束线程，而它是中间退出而已，PT_END 不可以在线程的中间调用。实际上 PT_EXIT 和 PT_RESTART 几乎是一样的，仅仅是返回值不同而已。

PT_SCHEDULE 这个是线程调度宏，在线程执行结束时会返回非真值，我们必须在线程调度时判断它的返回值，如果为真则继续调度，如果非真则退出。

PT_YIELD 这个宏可以让线程退出，让其他线程执行，第二次再进入时则继续执行，相当于线程暂停一次，为了支持这个 yield 操作，专门定义了一个变量 PT_YIELD_FLAG ，这是在线程开始时在 PT_BEGIN 中定义的，并且初始化为 1，在 PT_YIELD 中将 PT_YIELD_FLAG 赋为 0，这样后面的 if 判断就通过了，然后就返回 PT_YIELED；线程第二次再进入时在 PT_BEGIN 中将 PT_YIELD_FLAG 初始化为 1，使得线程继续往下执行。假如不用 yield 功能的话我们可以把 PT_YIELD_FLAG 变量删掉。

PT_YIELD_UNTIL 这个宏是结合了 PT_WAIT_UNTIL 和 PT_YIELD 的功能，它先执行 PT_YIELD 的功能，不管条件有没有成立，就要先退出一次，让其他线程进入执行。然后执行 PT_WHILE_UNTIL 的功能，一直阻塞等待条件成立。

学完了这几个宏实际上已经学完了 protothreads 的所有内容，我建议仔细分析一下源码中另外两个示例程序 example-buffer.c 和 example-codelock.c。如果还有不明白的，那就写段代码试试吧，别忘了分析一下预处理后的代码。

下面我们讲一下 protothreads 的一些特点。

4 protothreads 的特点

实际上从本质来看，protothreads 线程的调度是两次独立的函数调用。这就说明线程内部的局部变量是无法保存到下一次调用的。我们看下

面的例子：

代码 9: test-local.c

```
1  #include<stdio.h>
2  #include "pt.h"
3
4  struct pt local_pt;
5
6
7  static PT_THREAD(local_thread(struct pt *pt)){
8      char var = 1;
9
10     PT_BEGIN(pt);
11
12     printf("var␣=␣%d\n",var);
13     var ++;
14     PT_YIELD(pt);
15     printf("var␣=␣%d\n",var);
16     PT_END(pt);
17 }
18
19 int main(int argc,char *argv[]){
20     PT_INIT(&local_pt);
21
22     while(PT_SCHEDULE(local_thread(&local_pt)));
23 }
```

我们可能期望两次打印的时候 var 的值分别是 1 和 2。但是运行之后两次打印 var 的值都是 1。我们线程切换时保存的状态只有行号，所以出现了这样的现象。我们可以将 var 声明为 static 静态变量，这样可以在两次函数调用中保持值不变。

protothreads 所有的宏都只能在线程函数中出现，不能在线程函数调用的其他函数中使用，理解了 protothreads 的实现原理，这一点很容易理解，在其他函数中调用就不能构成 switch 结构。除非以子线程的形式调用，为它分配线程控制块，在函数内部有 PT_BEGIN 和 PT_END 宏。

protothreads 的宏不能嵌在别的 switch 语句中，因为 protothreads 宏本身会构成一个 switch 语句，如果在和其他 switch 交织在一起，就会引起错误。

最后留给大家一个思考题：在 PT_BEGIN 之前的代码运行起来会和其他地方的代码有什么不一样的地方？