

注意:

- (1) 任务中使用的变量应为静态变量
- (2) 线程内不能使用纯 while(1) --即含 PT_WAIT_UNTIL() 等宏的 while(1) 是可以的。
不能在 switch() {case...} 中调用任务 Protothreads API 带有 case 的语句 (即只能单向嵌套)。
- (3) 线程内可以使用:
for() {...}
switch() {case...} -- case 与 case 之间必须是一个完整的语句或者语句段
if() {...} else {...}
含宏的 while(1) {...}
- (4) ProtothreadS 系统可以仍然还是个大 while(1) 循环。但也可设计为根据定时器产生的恒定间隔的中断来触发和管理任务 -- 时间触发方式的嵌入式系统, 此时可更改 pt 结构体为 (见《时间触发模式下的 ProtothreadS 设计应用》):

```
struct pt {  
    lc_t lc;  
    unsigned short count; // 每次中断都减 1  
    unsigned short load; // 初始计数值  
    char ready; // 任务就绪标志  
}
```

- (5) 在 ProtothreadS 系统中延时:
 - 1) 如果 ProtothreadS 系统是基于时间触发, 则延时可基于该触发 -- 即基于系统时钟。
 - 2) 如果 ProtothreadS 系统中无系统时钟,
- (6) Protothreads 虽然提供了在各自线程内的条件阻塞机制, 但对于在该线程内调用的其它函数, 则无法阻塞其运行。所以, 如果要在线程内调用占用时间较多的函数, 为保证各个线程的实时性要求, 需要将这类函数进一步划分为更小的函数, 分步执行。
- (7) **Protothread 的精华: 当 Protothread 程序运行到 PT_WAIT_UNTIL 时, 判断其运行条件是否满足, 若不满足, 则阻塞。**
Protothread 的阻塞其实质就是函数返回。
Protothread 仅能在程序员指定位置阻塞。
- (8) 能满足系统实时性要求的条件是: (当且仅当) TaskA、TaskB、TaskC 三个任务的运行时间之和要小于系统实时响应的
时间要求。
- (9) 由于事件驱动模型没有阻塞机制, 因此需要由程序员构造一个有限状态机来实现顺序流控制。

ProtothreadS 函数说明 1:

函数	说明
PT_INIT(pt)	初始化任务变量, 只在初始化函数中执行一次就行
PT_BEGIN(pt)	启动任务处理, 放在函数开始处
PT_END(pt)	结束任务, 放在函数的最后
PT_WAIT_UNTIL(pt, condition)	等待某个条件 (条件可以为时钟或其它变量, IO 等) 成立, 否则直接退出本函数, 下一次进入本函数就直接跳到这个地方判断
PT_WAIT_WHILE(pt, cond)	和上面一个一样, 只是条件取反了
PT_WAIT_THREAD(pt, thread)	等待一个子任务执行完成
PT_SPAWN(pt, child, thread)	新建一个子任务, 并等待其执行完退出
PT_RESTART(pt)	重新启动某个任务执行
PT_EXIT(pt)	任务后面的部分不执行, 直接退出重新执行
PT_YIELD(pt)	锁死任务
PT_YIELD_UNTIL(pt, cond)	锁死任务并在等待条件成立, 恢复执行

ProtothreadS 函数说明 2:

函数	说明
struct pt { lc_t lc;};	
#define PT_INIT(pt) LC_INIT((pt)->lc)	Initialize a protothread 初始化 Protothread 线程, 实际上是将线程控制结构体里的 lc 置 0
#define PT_THREAD(name_args) char name_args	Declaration of a protothread 声明线程 例如: PT_THREAD(task1(struct pt *pt))
#define PT_BEGIN(pt) { char PT_YIELD_FLAG = 1; LC_RESUME((pt)->lc)	Declare the start of a protothread 启动任务处理
#define PT_END(pt) LC_END((pt)->lc); PT_YIELD_FLAG = 0; \	Declare the end of a protothread

PT_INIT(pt); return PT_ENDED; }	结束任务
<pre>#define PT_WAIT_UNTIL(pt, condition) \ do { \ LC_SET((pt)->lc); \ if(!(condition)) { \ return PT_WAITING; \ } \ } while(0)</pre>	Block and wait until condition is true 等待某个条件成立
#define PT_WAIT_WHILE(pt, cond) PT_WAIT_UNTIL((pt), !(cond))	Block and wait while condition is true 某个条件成立则一直等待
#define PT_WAIT_THREAD(pt, thread) PT_WAIT_WHILE((pt), PT_SCHEDULE(thread))	Block and wait until a child protothread completes 等待一个子任务执行完成
<pre>#define PT_SPAWN(pt, child, thread) \ do { \ PT_INIT((child)); \ PT_WAIT_THREAD((pt), (thread)); \ } while(0)</pre>	Spawn a child protothread and wait until it exits 新建一个子任务，并等待其执行完退出
<pre>#define PT_RESTART(pt) \ do { \ PT_INIT(pt); \ return PT_WAITING; \ } while(0)</pre>	Restart the protothread 线程的重新开始
<pre>#define PT_EXIT(pt) \ do { \ PT_INIT(pt); \ return PT_EXITED; \ } while(0)</pre>	Exit the protothread 线程的退出
#define PT_SCHEDULE(f) ((f) < PT_EXITED)	Schedule a protothread 线程的调度。f 传递线程的状态，当 f 为 PT_WAITING 或 PT_YIELDED 才参与调度 例如在主循环中：PT_SCHEDULE(task1(&pt_task1));
<pre>#define PT_YIELD(pt) \ do { \ PT_YIELD_FLAG = 0; \ LC_SET((pt)->lc); \ if(PT_YIELD_FLAG == 0) { \ return PT_YIELDED; \ } \ } while(0)</pre>	Yield from the current protothread 无条件唤醒线程。把唤醒标志设为 0，接着给线程设置局部连续块，然后判断如果唤醒标志为 0 则返回 PT_YIELDED 的状态
<pre>#define PT_YIELD_UNTIL(pt, cond) \ do { \ PT_YIELD_FLAG = 0; \ LC_SET((pt)->lc); \ if((PT_YIELD_FLAG == 0) !(cond)) { \ return PT_YIELDED; \ } \ } while(0)</pre>	Yield from the protothread until a condition occurs 有条件唤醒线程。与无条件唤醒的区别就在于在判断唤醒标志是否为 0 的同时还判断条件是否为成立
<pre>struct pt_sem { unsigned int count; };</pre>	信号量数据结构的定义 count 就是此信号量的值
#define PT_SEM_INIT(s, c) (s)->count = c	Initialize a semaphore 信号量初始化
<pre>#define PT_SEM_WAIT(pt, s) \ do { \ PT_WAIT_UNTIL(pt, (s)->count > 0); \ --(s)->count; \ } while(0)</pre>	Wait for a semaphore 等待信号量可用
#define PT_SEM_SIGNAL(pt, s) ++(s)->count	Signal a semaphore 信号量增加

	此时将(s)->count 加 1, 由 PT_SEM_WAIT 程序可知, 只有当(s)->count 的值变为大于等于 0, 线程才继续执行
--	--

简单例子:

<pre>#include "pt.h" static int counter; static struct pt example_pt; int main(void) { counter = 0; PT_INIT(&example_pt); while(1) { example(&example_pt); counter++; } return 0; }</pre>	<pre>static PT_THREAD(example(struct pt *pt)) { PT_BEGIN(pt); while(1) { PT_WAIT_UNTIL(pt, counter == 1000); printf("Threshold reached\n"); counter = 0; } PT_END(pt); }</pre>
<pre>static PT_THREAD(example(struct pt *pt)) { PT_BEGIN(pt); while(1){ PT_WAIT_UNTIL(pt, counter == 1000); printf("Threshold reached\n"); counter = 0; } PT_END(pt); }</pre>	<pre>static char example(struct pt *pt) { switch(pt->lc) { case 0: while(1) { pt->lc = 12; case 12: if(! (counter == 1000)) return 0; printf("Threshold reached\n"); counter = 0; } pt->lc = 0; return 2; } }</pre>

一个实时通信的例子:

<pre>PT_THREAD(sender(struct pt *pt)) { PT_BEGIN(pt); do { send_packet(); /* Wait until an acknowledgement has been received, or until the timer expires. If the timer expires, we should send the packet again. */ timer_set(&timer, TIMEOUT); PT_WAIT_UNTIL(pt, acknowledgment_received() timer_expired(&timer)); } while(timer_expired(&timer)); PT_END(pt); }</pre> <pre>PT_THREAD(receiver(struct pt *pt)) { PT_BEGIN(pt); /* Wait until a packet has been received, and send an acknowledgment. */ PT_WAIT_UNTIL(pt, packet_received()); send_acknowledgement(); PT_END(pt); }</pre>

```

//一个实现 delay---用于 LCD 显示时延时，当然需要增加时间信息
struct state {
    char *text;
    char *scrollptr;
    struct pt pt;
    struct timer timer;
};

PT_THREAD(display_text(struct state *s))
{
    PT_BEGIN(&s->pt);

    /* If the text is shorter than the display size, show it right away. */
    if(strlen(s->text) <= LCD_SIZE)
    {
        lcd_display_text(s->text);
    }
    else
    {
        /* If the text is longer than the display, we should scroll in the text from the right with a delay of
        one second per scroll step. We do this in a for() loop, where the loop variable is the pointer to the first character
        to be displayed. */
        for(s->scrollptr = s->text; strlen(s->scrollptr) > LCD_SIZE;    ++s->scrollptr)
        {
            lcd_display_text(s->scrollptr);
            /* Wait for one second. */
            timer_set(&s->timer, ONE_SECOND);
            PT_WAIT_UNTIL(&s->pt, timer_expired(&s->timer));
        }
    }
    PT_END(&s->pt);
}

```

官网第一页示例:

```

#include "pt.h"

struct pt pt;
struct timer timer;

PT_THREAD(example(struct pt *pt))
{
    PT_BEGIN(pt);

    while(1) {
        if(initiate_io()) {
            timer_start(&timer);
            PT_WAIT_UNTIL(pt,
                io_completed() ||
                timer_expired(&timer));
            read_data();
        }
    }
    PT_END(pt);
}

```

官方生产者消费者例子 (example-buffer.c):

(1) 全局变量

```

#define NUM_ITEMS 32          // 定义了生产或消费的货物有 32 个
#define BUFSIZE 8             // 定义了缓冲区的大小为 8
static int buffer[BUFSIZE];   // 缓冲区
static int bufptr;             // 缓冲区数组下标
static int item = 0;           // 定义了货物的标号
static int produced;           // 定义了生产的次数
static int consumed;           // 定义了消费的次数
static struct pt_sem full, empty; // 定义了两个信号量

```

(2) 数据操作

- 1) 往缓冲区里添加货物 add_to_buffer()
- 2) 往缓冲区里取出货物 get_from_buffer()
- 3) 生产一个货物 produce_item()
- 4) 消费一个货物 consume_item()

(3) 三个线程的定义

1) 生产者线程

```

static PT_THREAD(producer(struct pt *pt))
{
    static int produced;

    PT_BEGIN(pt);
    for(produced = 0; produced < NUM_ITEMS; ++produced) {
        PT_SEM_WAIT(pt, &full);
        add_to_buffer(produce_item());
        PT_SEM_SIGNAL(pt, &empty);
    }
    PT_END(pt);
}

```

可以发现，通过 PT_BEGIN() 和 PT_END() 保证线程操作的原子性，而通过 PT_SEM_WAIT() 和 PT_SEM_SIGNAL() 保证生产者和消费者根据缓冲区内容的多少来行为。

2) 生产者线程

```

static PT_THREAD(consumer(struct pt *pt))
{
    static int consumed;

    PT_BEGIN(pt);
    for(consumed = 0; consumed < NUM_ITEMS; ++consumed) {
        PT_SEM_WAIT(pt, &empty);
        consume_item(get_from_buffer());
        PT_SEM_SIGNAL(pt, &full);
    }
    PT_END(pt);
}

```

3) 驱动线程

```

static PT_THREAD(driver_thread(struct pt *pt))
{
    static struct pt pt_producer, pt_consumer;

    PT_BEGIN(pt);

    PT_SEM_INIT(&empty, 0);
    PT_SEM_INIT(&full, BUFSIZE);

    PT_INIT(&pt_producer);
    PT_INIT(&pt_consumer);

    PT_WAIT_THREAD(pt, producer(&pt_producer) & consumer(&pt_consumer));

    PT_END(pt);
}

```

(4) 主函数

```
int main(void)
{
    struct pt driver_pt;

    PT_INIT(&driver_pt);

    while(PT_SCHEDULE(driver_thread(&driver_pt))) {
        /*
         * When running this example on a multitasking system, we must
         * give other processes a chance to run too and therefore we call
         * usleep() resp. Sleep() here. On a dedicated embedded system,
         * we usually do not need to do this.
         */
#ifdef _WIN32
        Sleep(0);
#else
        usleep(10);
#endif
    }
    return 0;
}
```

```
int a_prototread(struct pt *pt) {
    PT_BEGIN(pt);

    PT_WAIT_UNTIL(pt, condition1);
    if(something) {
        PT_WAIT_UNTIL(pt, condition2);
    }

    PT_END(pt);
}
```

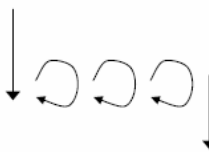
Hierarchical protothreads

```
int a_prototread(struct pt *pt) {
    static struct pt child_pt;

    PT_BEGIN(pt);

    PT_INIT(&child_pt);
    PT_WAIT_UNTIL(pt2(&child_pt) != 0);

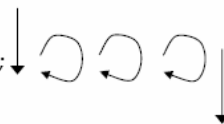
    PT_END(pt);
}
```





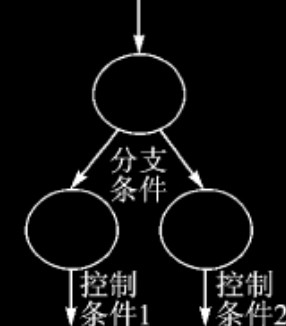
```
int pt2(struct pt *pt) {
    PT_BEGIN(pt);

    PT_WAIT_UNTIL(pt, condition);

    PT_END(pt);
}
```



事件驱动程序中的顺序流控制情况 (《使用 Protothread 简化嵌入式系统中的顺序流控制》，其有个密码锁例子):

情况	 <p>(a) 顺序</p>	 <p>(b) 重复</p>	 <p>(c) 分支</p>
Protothread 实现结构	顺序结构: PT_BEGIN() ... PT_WAIT_UNTIL(condition) ... PT_END()	重复结构: PT_BEGIN() ... while(cond1) PT_WAIT_UNTIL(cond1 or cond2) ... PT_END()	分支结构: PT_BEGIN() ... if(condition) PT_WAIT_UNTIL(cond2a) else PT_WAIT_UNTIL(cond2b) ... PT_END()