设计自己的嵌入式操作系统内核之二

--- 队列结构与内核队列

一、概述

在嵌入式操作系统中，总是存在多个任务并发运行。这些任务间需要共享和竞争资源，也可能存在某种“协作”关系。在资源不可用时，或“协作”条件未达到时，任务需要等待。有时，因同一原因需要等待的任务存在多个。比如，如果将ＣＰＵ也视为一种资源，在一般的单处理器中，总会存在一个或多个任务在等待ＣＰＵ的空闲而被置入“就绪队列”中。

不仅是ＣＰＵ的就绪队列，对于诸如信号量、邮箱队列等内核提供的服务，操作系统也需要为这些结构实现任务的阻塞队列。当任务申请延时时，需要将任务置于某种队列结构中，以备内核在系统时钟中断发生时或之后对这些任务进行处理。一种极端的实现机制是，不设置任务的队列结构，在事件发生或资源可用时，操作系统遍历每一个任务，检查是否阻塞，如果是，则将任务置为就绪态。这种实现机制，在任务数少，且硬件资源如 RAM 等不够用的情况下非常有效，但对于任务数较多，且调度算法相对而言复杂的情况下，显然效率太低。

这里，针对整个系统内核的各个对像，信号量、消息队列、就绪队列等，实现一种适合的队列结构。

首先，会简要的介绍在 eos 中任务的实际表示，然后会明确设计的需要，提出相应的设计要求，再介绍队列的实现。最后，就整个内核，简要说明内核中的各种队列是如何以此为基础实现的。

二、任务的表示

首先明确的是，eos 并不支持多进程，而只支持多线程。整个内核作为一个大的进程体，而任务则对应于进程体内的多个线程。任务作为进程内的活动单元，表现为如下形式。

```
void task0( task_data_t pdata )
{
    uint8    n;

    LED_INIT();
    n = (uint8)(( uint16 )pdata & 0xff );
    while(1)
    {
        semaphore_wait( led1_sem, 0 );          /*  等待信号        */
        PORTB ^= n;                             /* LED1  闪烁       */
    }
}
```

从上很明确可以看出，线程执行的代码体对应于函数，而进程执行的代码则是对应于程序，二者明显是不同的。正因为此，线程之间可以共享全局数据。

在嵌入式系统中，一般直接称线程为“任务”。任务只是逻辑上的一种抽像。cpu 只知道简单的取指－译码－执行，没有多任务的概念。为了表征这一抽像，这里采用了以下数据

类型表示：

```
struct   _task_struct                                   /* task_t 控制块          */
{
    stack_t * stk_top;                                  /*  任务栈顶              */
    node_t   tmo_node;                                  /*  延时链接              */
    node_t   wait_node;                                 /*  事件链接              */

    uint8    state;                                     /*  任务当前状态          */
    uint8    prio;                                      /*  任务优先级            */
    uint16 delay;                                       /*  任务延时值            */

    uint8    slice;                                     /*  任务运行时间片        */
    uint8    total_slice;                               /*  任务运行总的时间片    */

#if OS_TASK_SET_SUSPEND == 1
    uint8    suspend_cnt;                               /*  任务挂起的次数        */
#endif

#if OS_TASK_SET_DESTORY_FUN == 1
    void     ( * destroy )( task_t task, void * pdata );/*  任务删除析构函数      */
    void * pdata;
#endif

    uint8    wait_state;                                /*  等待状态              */
    void * event;                                       /*  任务等待的事件        */
};
```

在实际的系统中，一个任务除了包含它所执行代码（即前面的函数）外，还包含了一块内存区域。该块内存为上述结构体的实例。task_struct 结构体存储了任务的运行状态。在 os 中，通过保存和恢复任务的状态至上述结构体中，来实现任务的停止和恢复运行。可以认为 task_struct 结构唯一对应一任务。任务通过执行任务代码的活动状态反映其存在。相关的概念请参考操作系统原理方面的书籍。

三、队列结构的设计

在 eos 中，同时存在多个任务，任务间相互竞争资源。然会资源的数量总是有限的，以 CPU 为例，cpu 只有一个，而任务有多个。除了某一个任务外，其的任务必须等待。具体的表现为：资源被其它资源占用时，将需要等待的任务置入某个 FIFO 结构，或是 LIFO 结构中；资源可用时，再从队列中取 task_struct 结构，将资源分配给对应的任务。等待 cpu 的队列称为就绪队列，队列结构的设计以及如何向队列中插入与删除 task_struct 结构就决定了操作系统中说的调度器的调度策略。

当然，队列的实现是多样的。ucos 中使用的是位图，占用空间小，操作快速。但其是以每个优先级只能有一个任务为前提的。eos 中，设置了多个优先级，同优先级内允许多个

任务存在，显然再使用位图是不合适的。因而，需要一种不同的实现。

首先考虑队列实现的要求。

1、eos 中任务有多种优先级，而优先级最高的任务具有获取资源的最高优先权。因而，有必要维护多级队列，每个队列对应于某一优先级。同一优先级内的各任务间获取资源的方式按 FIFO/或 LIFO 方式进行，以保证一定的公平性；

2、提供依据任务的优先级插入到相应的队列结构操作；

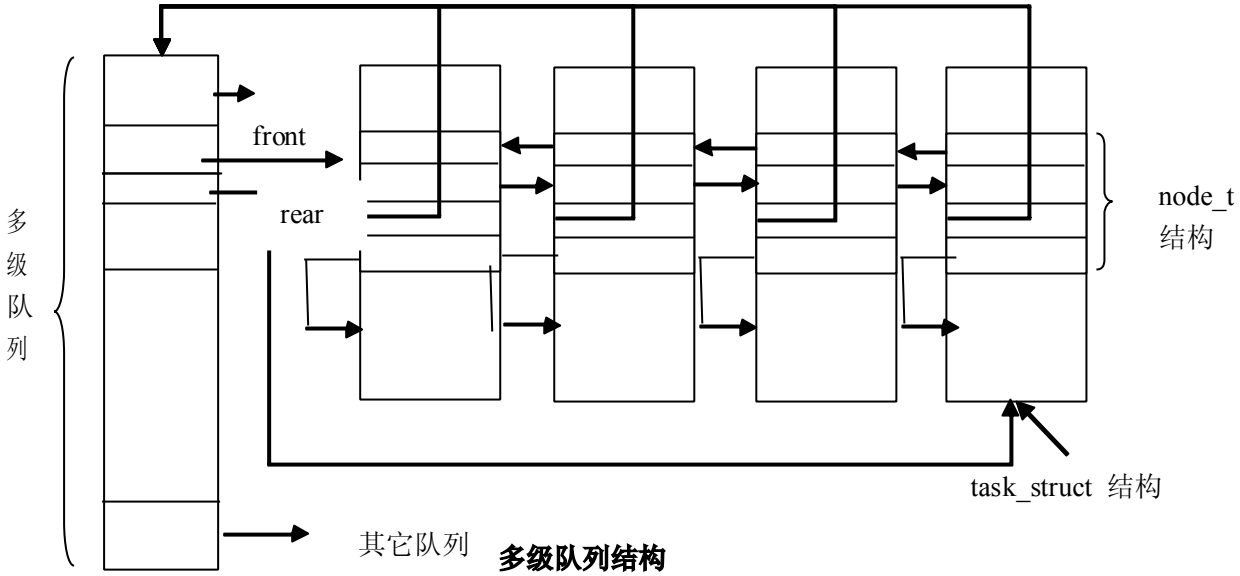
3、提供从队列中取出最高优先级任务的操作，因为优先级任务高的任务有优先权，这样当事件发生时，从队列中唤醒的首先是最高优先级的任务；

4、提供队列中任意任务的删除操作；一种情况是，当任务被删除时，如果其被阻塞在某个队列中，必须将其从队列中移除；

5、其它的一些操作：队列的判空，返回队列中任务的最高优先级等。

四、队列结构的实现

下图是一多级队列的示意图，为方便起见，多级队列中只画出了一级队列。该单级队列被组织成双向链表的结构。同时为使得操作能按 FIFO/LIFO 方式，队列提供了.front, .rear 指针分别指示队首、队末。



多级队列结构

一）、结点结构

node_t 对应的结构实现如下：

```
struct   _node_t                                /*  双向队列结点类型                */
{
    struct _node_t * pre, * next;               /*  结点的链接指针                  */
    union
    {
        struct   _slist_t * slist;              /*  结点所在的队列                  */
        struct   _mlist_t * mlist;
    }xlist;
#define xslist   xlist.slist
#define xmlist   xlist.mlist
    task_t task;                                /*  结点的所有者                    */
};
```

node_t 结构用于链接 task_struct 结构. 前面的 task_struct 结构中, 维护了两个 node_t 结构:

```
node_t    tmo_node;                    /* 延时链接          */
node_t    wait_node;                   /* 事件链接          */
```

其中 tmo_node 用于将任务链入延时队列中, wait_node 用于将任务置入资源等待队列, 如就绪队列, 信号量阻塞队列. 提供两个结点是必要的, 比如, 任务在使能了超时等待信号量, 那么任务除了挂起在信号量阻塞队列中外, 还有必要置入延时队列, 显然仅用一个结点是不够的.

既然是任务可能需要同时置入两组队列中, 就有必要维护两组指针用于将 task_struct 链入队列, 而考虑到这两种出队入队操作基本一致, 因而对这两组指针进行封装成 node_t, 以提供统一的操作.

node_t 结构中包含有 task_struct 结构(即 task_t). task_t 结构保存了当前结点所在 task_struct 块的指针。考虑到队列中任务间的链接是借助于 node_t 结构的, 即队列的操作的基本元素为 node_t 结构, 而非 task_struct 结构. 当从队列中取出一结点是, 得到的是 node_t 结构, 而我们要得到的是 task_struct, 这时可以从. task 域中得到.

另一有用的域为. xlist，其保存了当前结点所在队列的指针。因为结点可能处于多级队列或普通的 slist 队列中(后面会说明)，因而用联合体。. xlist 主要方便于将任意结点从队列中删除，这在后面会说明。

对 node_t 结构提供了一初始化操作:

```
    void    task_node_init( task_t task );
```

同时完成了 task_t 结构中 tmo_node，wait_node 的初始化。

二)、单级队列结构

对应于上图所示的多级队列图，单级队列只是图中的一级队列。

对其中一级队列，其表示为:

```
struct _slist_t                          /* 双向队列        */
{
    struct _node_t  * front;             /* 队首指针        */
    struct _node_t  * rear;              /* 队尾指针        */
};
```

. front 为队列出指针，指向队列的第一个结构，. rear 为队列出指针，指向队列中最后一结点。

其基本的操作为:

```
void      slist_init( slist_t * list );                  /* slist 线性表初始化          */
void      slist_add_node( slist_t * list, node_t * node );/* 添加结点至 slist 头部       */
void      slist_append_node( slist_t * list, node_t * node ); /* 添加结点至 slist 尾部   */
task_t    slist_get_task( slist_t * list );              /* 从 slist 头部取 TCB         */
void      slist_del_node(   node_t * node );             /* 将结点从所在队列移除        */
bool_t    slist_is_empty( slist_t * list );              /* slist 判空.TRUE-空, FLASE - 非空
*/
```

可以看出，slist 的操作并不是严格按照 FIFO 方式进行的，同时提供了 LIFO 方式，以及任意结点的删除操作。

注意 void slist_del_node(node_t * node)操作。其参数只有 node，而没有队列指针，原因在于结点所在的队列可以通过 node_t 结构中的 xslist 域获得。当然，可以在 slist_del_node 中引入 slist_t * list 参数，但这个参数由何而来？当删除一个任务时，如果任务此时挂在某个队列中，必然要将其移除，但挂在哪个队列，显然队列的信息必然要保存在 task_struct 中。更进一步，保存在 node_t 中，可达到同样的效果，并不多占用存储空间，而且操作更为方便，此时只需要关注 node_t 结构，而不需考虑 task_struct 结构。

五、mlist 多级队列结构

mlist 队列单级队列 slist_t 的组合，但增加了优先级的限定。

```
struct  _mlist_t                                    /* 多级双向队列 */
{
    uint8 prio_grp;                                 /* 队列位图组 */
    uint8 prio_tbl[ OS_PRIO_MAX / 8 + 1 ];          /* 队列位图表 */
    struct _slist_t task_tbl[ OS_PRIO_MAX + 1 ];    /* 多级任务队列 */
};
```

task_tbl 域为 slist_t 的数组，数组 0 对应的队列优先级最高，意味着在竞争资源时有较高的优先权。prio_grp 和 prio_tbl 域用于位图式的快速查找，使用的是 ucos 的位图查找法，也是查找最高优先级，但这里查找的是有任务的队列，而不是任务。对于其方法，这里也不作介绍，请参考源码或 ucos 源码分析的书籍。

基本操作为：

```
void    mlist_init( mlist_t * list );                   /* mlist 线性表初始化 */
void    mlist_add_task( mlist_t * mlist, task_t task ); /* 添加 task 至线性表 */
void    mlist_del_task( task_t task );                  /* 将 task 从 mlist 移除 */
task_t  mlist_get_task( mlist_t * mlist );              /* 从 mlist 取优先级最高的首个 task */
task_t  mlist_prio_max_task( mlist_t * mlist );         /* 返回 mlist 优先级最高的首个 task */
bool_t  mlist_is_empty( mlist_t * list );               /* mlist 判空.TRUE-空, FLASE - 非空
```

对 mlist_t 操作的对像是 task，而不再是 node_t，实际的链接结点仍为 node_t。考虑 mlist 只用于阻塞队列，而任务链入任何阻塞队列只用到 wait_node，所以在操作的输入输出上均使用 task 对像，应用起来更方便。

六、多样的内核队列

eos 是如何应用前面提到的多级队列(mlist_t), 单级队列(slist_t)的?

延时队列的实现：

在任务需要暂停执行，等待指定的时间后再恢复时，会被置入延时队列中等待。在系统每个时钟中断发生后，对延时队列中的任务进行处理，当任务的延时期到后，将任务从队列移除。显然，我们只需要关心任务的延时值是否会达到，而不考虑优先级。所以，直接应用

slist_t 来表示即可。任务入队列从队末入，只要延时到，就从队列删除，slist_t 对应的操作可以完成这类需要。

就绪队列的实现：

就绪队列里的任务在竞争 cpu 的运行。前面提到，eos 的任务有多种优先级，优先级最高的任务有优先获得 cpu 的机会。为达到此目的，使用 mlsit_t 来组织就绪的任务。

延时队列和就绪队列的定义如下：

```
extern   mlist_t OSRdyQ;                          /* 就绪队列      */
extern   slist_t OSDelayQ;                        /* 延时队列      */
```

阻塞队列的实现：

任务在等待获取资源时，需要进入相应的资源等待队列。当资源可用时，依据不同的策略，即可以按照 FIFO 的方式将资源分配给最先进入队列的任务，也可以按照优先级分配资源给优先级最高的任务，同一优先级间按 FIFO 方式进行。对于后者，队列结构要占有更多的存储空间。在一些情况下，进入同一队列的各任务的优先级总是相同的，因为就可选用单级队列，以减少存储空间，同时也提高了操作效率。在 eos 的信号量实现中，同时支持这两种方式。以下是信号量的结构，通过条件编译可选择阻塞队列的类型.

```
typedef struct _semaphore_t                       /* 信号量结构          */
{
    uint8     cnt;                                /* 当前计数            */
    uint8     max_cnt;                            /* 最大计数            */
#if OS_SEM_SET_MLIST == 1
    mlist_t wait_list;                            /* 阻塞队列            */
#else
    slist_t wait_list;
#endif
}* semaphore_t;
```

六、总结

上述的 slist_t, mlist_t 结构及其操作还是比较简单和易于理解的，在一般的《数据结构》教材中都会有所讲述。但是相对于教材中提到的结构，这里的实现相对而言要复杂些，但效率更高，操作也更加方便。特别是关于 node_t 结点，我最初的想法是直接在 task_struct 结构体中提供两组指针，用于分别进行链接，后经反复修改，同时参考了 freertos 等内核的源码，稍作了一些调整，从后面的关于任务调度和诸如信号量等待实现的效果来看，这种队列还是比较有效的。并且提供的操作，其执行是可确定，可预知的。

很显然，这类实现占用的存储空间较大。当优先级数增多时，多级队列占用的存储空间也会相应增大。这对于只有2K ram 的单片机来说，显得有些臃肿。

七、源码与资料

涉及源码文件：

list.c / list.h ---------- 队列结构实现

core.h ---------- task_struct 结构定义

李述铜 于 四川大学 电气信息学院

2009-7-26