VxWorks 内核

余争

版 本 历 史

版本/状态	责任人	起止日期	备注
V1.0/草稿	余争	2011Nov21	创建文档。

目 录

1. 前言

很幸运能够有时间认真的研究了 VxWorks5.5 的源码。惊叹前人能够运用这样的逻辑，写出这么好的代码！惊叹之余，我做下笔记，并经过一定的组织，于是形成现在这份文档。

如果你有机会接触到 VxWorks 的源码，那么你与我一样幸运，你可以就这份文档与我一起探讨。如果你没有 VxWorks 的源码，那么你也可以从这份文档里窥视出 VxWorks 内核的实现。但请不要问我要它的源码，因为我现在坐在家里，我也没有。不过我认真的模仿了 VxWorks5.5 内核源码的精华，并且从开源的 UCOS_MIPS 那里拿出一些头文件，写了一个简单的内核，可以提供一些练习（不过只提供了.a 与头文件，因为里面有好多代码是赤裸裸的模仿，怕引发不必要的麻烦）。

说到内核的实现，大家可能都会想到 Linux 的内核。VxWorks 内核的实现与 Linux 的内核的实现思路相差是很大的，如果想看到这份差异，请阅读这份文档☺。

1.1 文档特点

写这份文档的时候，我尽量保持每一章节的独立性。每一章都是一个独立的整体，并且为了尽力保持独立，在讲解本章的逻辑的时候，可能会用到的其它章节的内容，我将不会再有过多的重复。

Rain——VxWorks Like Kernel

2. VxWorks 对象

2.1 函数接口

2.1.1 类定义

```
typedef struct _obj_class
{
  struct _obj_core objCore;

  struct _mem_part* *objPartId;
  unsigned objSize;
  unsigned objAllocCnt;
  unsigned objFreeCnt;
  unsigned objInitCnt;
  unsigned objTerminateCnt;

  int coreOffset;
  FUNCTION createRtn;
  FUNCTION initRtn;
  FUNCTION destroyRtn;
  FUNCTION showRtn;
}OBJ_CLASS;

typedef struct _obj_class *CLASS_ID;       /* CLASS_ID */
```

从上面的定义可以看出，类的定义里有四个函数指针，同时还有用于内存分配的内存分区 ID。从四个函数指针的定义可以初步推测出，createRtn 用于创建对象，initRtn 用于初始化对象，destroyRtn 用于销毁对象，showRtn 用于显示对象的信息。

2.1.2 创建类

用下面的接口创建一个类。比如创建一个 semClassId 类：

```
classInit (semClassId, sizeof(SEMAPHORE), OFFSET(SEMAPHORE,objCore),
          (FUNCPTR) NULL, (FUNCPTR) NULL, (FUNCPTR) semDestroy) == OK)
```

2.1.3 创建对象

从一个类中创建一个对象，相当于 C++的 new 操作。

```
typedef struct semaphore *SEM_ID;
SEM_ID semId;
semId = (SEM_ID) objAlloc (semClassId)
```

2.1.4 识别对象

在每个对象进行初始化的时候，都有 objCoreInit 这么一步初始化。它的作用是将一个对象关联到一个类。比如对象 pSemaphore 是属于 semClass 类型的，它将进行下面的操作：

```
objCoreInit (&pSemaphore->objCore, semClassId);
void objCoreInit
    (
    OBJ_CORE    *pObjCore,
    OBJ_CLASS   *pObjClass
    )
    {
    pObjCore->pObjClass = pObjClass;
    pObjClass->objInitCnt ++;
    }
```

这给验证对象的合法性带来很大的好处。对象的验证可参考宏 OBJ_VERIFY(objId,classId)。在阅读内核代码时常常看到这个宏。

2.2 对象的创建流程

对象总有一个根。那么 VxWorks 的对象的开始在哪里？

上文提到过 semClass 通过下文的函数进行创建，如果继续跟踪到 classInit 的函数实现的话

```
classInit (semClassId, sizeof(SEMAPHORE), OFFSET(SEMAPHORE,objCore),
    (FUNCPTR) NULL, (FUNCPTR) NULL, (FUNCPTR) semDestroy) == OK)
```

——（接上文）会看到以下代码。

```
/* initialize class as valid object */
objCoreInit (&pObjClass->objCore, classClassId);
```

从类型的识别一节当中知道，这段代码是把 pObjClass（即为传入的 semClassId）设置为类型 classClass 的一个对象。即很容易看出，所有的类型即为 classClass 的一个对象。

继续跟踪 classClassId 的初始化，如下面代码所示。

```
classInit(classClassId,sizeof(OBJ_CLASS),OFFSET(OBJ_CLASS,objCore),(FU
NCPTR)classCreate,(FUNCPTR)classInit,(FUNCPTR)classDestroy);
```

可以看出 classClassId 把自己也初始化为一个 classClass 的一个对象。

把上面的推导过程总结一下是：（TODO 图示）

VxWorks 对象模块先创建了一个 classClass 的对象，并初始化自身为 classClass 类。后续的类的创建，都是相当于创建了一个 classClass 的对象。有了类，便可以从类创建出无究无尽的对象来。

2.3 对象设计对内核的意义

（TODO）

3. VxWorks 动态内存

3.1 内存结构

VxWorks 使用内存分区(memory partition)、内存池(memory pool)、内存块(memory block)来管理运行需要的内存。一个内存分区包括多个内存池。内存池是一块连续的区域，包含许多个内存块。内存分区和内存池对用户是透明的。

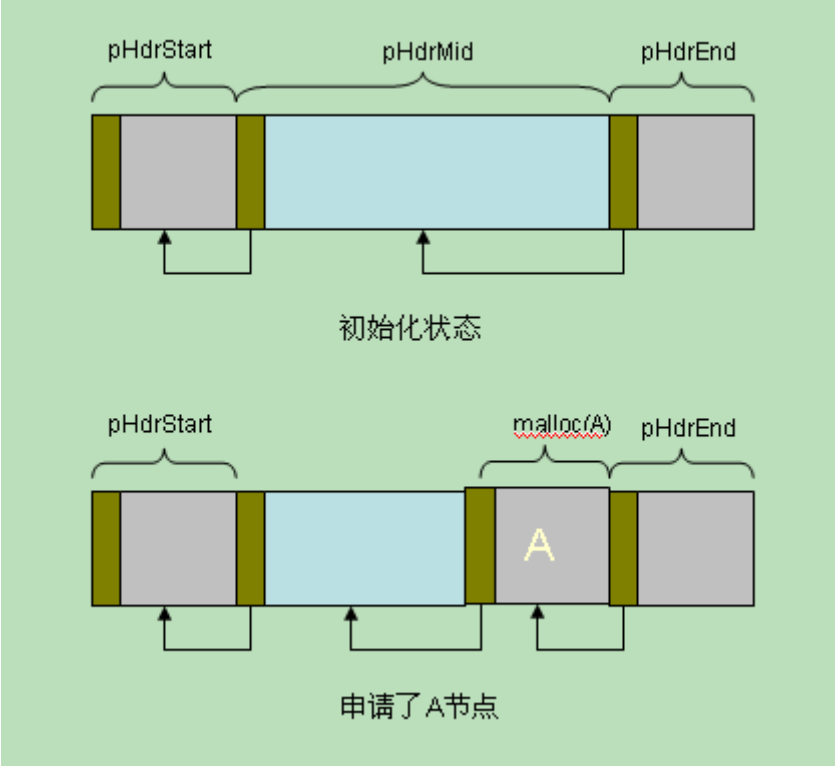
系统起来后的时候，会先创建一个叫 memSysPartition 的系统内存分区。可以使用 memPartCreate 来创建新的内存分区，而后独立在此分区分配内存不会受到其它内存的影响。（强烈建议一些使用内存比较大的模块自己创建自己的内存分区，可以避免对系统内存的影响，包括因为内存碎片产生的影响）。

（内存结构关系图）

3.2 内存分布

在 VxWorks 的分区当中，内存的组织形态是以一个单链表的形式存在的（图 4-1 所示）。每个节点称之为一个内存块，具体代码如下：

```
typedef struct blockHdr        /* BLOCK_HDR */
    {
    struct blockHdr *    pPrevHdr;    unsigned        nWords : 31;
    unsigned        free    : 1; /* TRUE = this block is free */
  } BLOCK_HDR;
```



图表 3- 1

3.3 内存池

内存分区已释放（未被使用）的内存块，都把它放入一个 freeList 的双连表当中。这个双链表，就是一个内存池。双链表的每个节点如下：

```
typedef struct                  /* FREE_BLOCK */
    {
    struct
        {
  struct blockHdr *   pPrevHdr;      /* pointer to previous block hdr
*/
  unsigned        nWords : 31;/* size in words of this block */
  unsigned        free   : 1;  /* TRUE = this block is free */
  } hdr;
    DL_NODE         node;       /* freelist links */
    } FREE_BLOCK;
```

3.4 FIRST FIT 算法

VxWorks 内存分配采用 FIRST FIT 算法。基本操作是，从 freeList 内存池一个节点一个节点搜索，找第一个大小合适，对齐方式也合适的内存块；如果这个内存块正好合适，那么从 freeList 里删除，并设置为被使用状态；如果内存块比需要申请的还大，那么要将余下的内存作为新的未使用的内存块放入 freeList 当中。具体请参见代码~

（TODO 详细解释）

3.5 内存释放

内存释放类似于使用伙伴算法。释放的内存块检查前面的内存块是否是也是处于释放状态，如果是释放状态，则将它们两合并成新的内存块。

（TODO 详细解释）

3.6 系统对象与内存

4. VxWorks 内核特性

都知道 VxWorks 是基于优先级的抢占式的系统，那么这个内核在内部实现上又会有什么样的特性呢？

4.1 内核数据

系统运行过程中的全局信息被放入以下全局变量中：

4.1.1 activeQHead

是一个先入先出队列，记录着所有的任务控制块（WND_TCB）的信息。任务创建后就被挂入此队列，一直到任务结束或被删除。

4.1.2 readQHead

是一个优先级队列，记录着所有处于就绪状态的任务。队列的头结点意味着是优先级最高的任务控制块。

4.1.3 tickQHead

是一个以时间为 key 的优先级队列。如果任务需要延迟操作，那么会被挂入此队列。

4.2 内核态

VxWorks 用一个整形 kernelState 来标识系统处于内核态。如果 kernelState=TRUE，表示正是处于内核态，接下来很可能是对内核数据进行修改；如果 kernelState=FALSE 表示处于用户状态运行。

很明显看出来 VxWorks 所谓的内核态并不是体现在系统操作权限的内核态（TODO VxWorks 所有代码都运行在 xxx 态。），而不过只是一个标识，表示接下来我将对内核数据进行修改，并可能产生内核的调度。

4.3 内核互斥

很容易想象，既然 VxWorks 用几个全局数据来记录系统的任务，那么这些全局数据是如何进行互斥的。先假设在单 CPU 的场景，现在正将把一个任务控制块挂入 readyQHead，此时产生中断，到了另外的上下文，又将另外一个任务控制块加入 readyQHead，那么很可能就会破坏 readyQHead 数据。（TODO 代码示例）当然想避免这种情况，最简单的方法是在任务切换的时候禁止中断，但是如果这样又会带来太大的中断延迟。VxWorks 采用另外的办法——这亏功于 kernelState 的作用。

当需要任务切换的时候，比如往 readyQHead 加入一个任务，先将 kernelState 设置为 TRUE 表示当前系统需要操作内核数据，然后进行内核操作，这个时候是不禁止中断的。

进行内核操作时，若没有产生中断，那么内核操作结束后，将 kernelState 设置为 FALSE。

若此时产生中断，并且中断又要求对内核数据进行修改（比如释放信号量操作），因为此时 kernelState=TRUE，系统会把修改内核数据的操作加入到一个延迟操作的队列，中断完成后再统一进行处理。（TODO 具体操作流程是？）（TODO 图示）

4.4 特性总结

5. 任务

下面是一个任务的基本定义，我只列出了比较重要的部分。

```
typedef struct windTcb          /* WIND_TCB - task control block */
    {
    Q_NODE          qNode;          /*   0x00:   multiway   q   node:
rdy/pend q */
    Q_NODE          tickNode;   /* 0x10: multiway q node: tick q */
    Q_NODE          activeNode; /* 0x20: multiway q node: active q */

    OBJ_CORE        objCore;    /* 0x30: object management */
    Q_HEAD *        pPendQ;         /* 0x5c: q head pended on (if any)
*/
    UINT            safeCnt;    /* 0x60: safe-from-delete count */
    Q_HEAD          safetyQHead;    /*  0x64: safe-from-delete  q  head
*/

    FUNCPTR         entry;          /* 0x74: entry point of task */

    char *          pStackBase; /* 0x78: points to bottom of stack */
    char *          pStackLimit;    /* 0x7c: points to stack limit */
    char *          pStackEnd;  /* 0x80: points to init stack limit */

    int             errorStatus;    /*  0x84:  most  recent  task  error
*/
    int             exitCode;   /* 0x88: error passed to exit () */

#if (CPU_FAMILY==MIPS)
    EXC_INFO        excInfo;    /* 0x108: exception info */

    /* REG_SET must be aligned on a ?? byte boundary */

    REG_SET         regs;       /* 0x128: register set */
#endif  /* CPU_FAMILY==MIPS */

    } WIND_TCB;
```

每个字段的作用其实后面的注释说得很清楚了。Q_NODE 等字段是用于挂入某些特定的队列的，比如 readQHead、activeQHead、tickQHead 等。

5.1 创建

任务的创建相对比较简单，无非是：

申请 WIND_TCB 的内存空间

初始化任务栈空间

初始化 WIND_TCB.REGS 的寄存器

把任务加入 activeQHead 队列；如果任务可以执行，那么加入 readQHead 队列

执行任务调度

5.2 切换

（TODO 汇编代码）

5.3 删除

任务的删除会比较复杂。流程如下：

6. 信号量

6.1 同步单元

6.2 二进制信号量

6.2.1 初始化

6.2.2 创建

6.2.3 获取

6.2.4 释放

6.3 互斥信号量

6.3.1 初始化

6.3.2 创建

6.3.3 获取

6.3.4 任务反转

6.3.5 释放

7. 消息队列(TODO)

7.1 结构与数据

名称	属性名	类型	允许空	说明
BugID	BugID	Uniqueidentifier	否	自动生成唯一标识
所属项目	ProjectName	Nvarchar(50)	是	出现这个 Bug 的项目

Bug 状态	BugState	Nvarchar(50)	否	记录 Bug 所处状态