**->https://blog.csdn.net/redseazhaojianertao/article/details/53843493?utm\_source=blogxgwz6**

**VxWorks内核解读-5**

2016年12月23日 20:16:22 [redseazhaojianertao](https://me.csdn.net/redseazhaojianertao" \t "_blank) 阅读数：2040

本篇博文，我们该谈到Wind内核的内存管理模块了，嵌入式操作系统中, 内存的管理及分配占据着极为重要的位置, 因为在嵌入式系统中, 存储容量极为有限, 而且还受到体积、成本的限制, 更重要的是其对系统的性能、可靠性的要求极高, 所以深入剖析嵌入式操作系统的内存管理, 对其进行优化及有效管理, 具有十分重要的意义。在嵌入式系统开发中, 对内存的管理有很高的要求。概括地说, 它必须满足以下三点要求:

* 实时性, 即在内存分配过程中要尽可能快地满足要求。因此, 嵌入式操作系统中不可能采取通用操作系统中的一些复杂而完备的内存分配策略, 而是要采用简单、快速的分配策略, 比如我们现在讨论的VxWorks 操作系统中就采用了“ 首次适应”的分配策略。当然了具体的分配也因具体的实时性要求而各异。
* 可靠性, 即在内存分配过程中要尽可能地满足内存需求。嵌入式系统应用千变万化, 对系统的可靠性要求很高, 内存分配的请求必须得到满足, 如果分配失败, 则会带来灾难性的后果。
* 高效性, 即在内存分配过程中要尽可能地减少浪费。在嵌入式系统中, 对体积成本的要求较高, 所以在有限的内存空间内, 如何合理的配置及管理, 提高使用效率显的尤为重要

实时嵌入式系统开发者通常需要根据系统的要求在RTOS提供的内存管理之上实现特定的内存管理，本章研究 VxWorks的Wind内核内存管理机制。

**5.1 VxWorks内存管理概述**

**5.1.1 VxWorks内存布局**

VxWorks5.5 版本提供两种虚拟内存支持（Virtual MemorySupport）：基本级（Basic Level）和完整级（Full Level）。基本级虚拟内存以Cache 功能为基础。当DMA 读取内存前必须确保数据以更新到内存中而不是仍缓存在Catch 中。对于完整级需要购买可选组件VxVMI，其功能为：提供对CPU 的内存管理单元的编程接口，同时提供内存保护，它在功能上覆盖了基本级虚存。VxWorks的内存配置宏如下：

INCLUDE\_MMU\_BASIC：基本级虚存管理，无需VxVMI；

INCLUDE\_MMU\_FULL:完整级虚存管理,需组件VxVMI；

INCLUDE\_PROTECT\_TEXT：写保护text，需组件VxVMI；

INCLUDE\_PROTECT\_VEC\_TABLE：写保护异常向量表，需组件VxVMI；

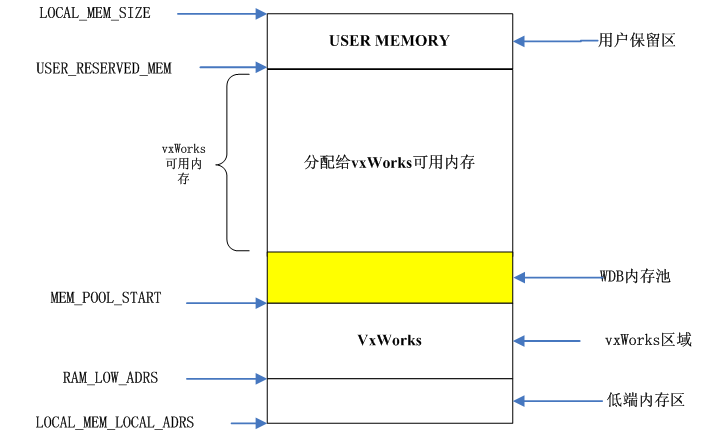
备注：VxVMI,即虚拟内存接口，是VxWorks的一个功能模块，它利用用户片上或板上的内存管理单元（MMU）,为用户提供了对内存的高级管理功能。

在VxVMI的最小配置中，它写保护了几个关键资源，其中包括VxWorks程序代码体、异常向量表、以及通过VxWorks装载器下载的应用程序代码体。保护特性让开发人员集中精力编写自己的程序，无需担心无意中修改关键代码段或引发耗时的系统错误。这在开发阶段是很有用的，因为它简化了对致命性错误的诊断。在产品的定型阶段也是如此，因为它提高了系统可靠性。VxVMI提供的其它工具主要用于修改这些被保护的区域，如修改异常表或者插入断点。

以上配置可在Tornado 开发环境中MMU 配置管理工具中改变，也可在BSP 的config.h 中完成。内存页表的划分和属性配置，包括BSP 的config.h 中VM\_PAGE\_SIZE 和sysLib.h 中的sysPhysMemDesc 定义。本篇博文只考虑VxWorks基本级内部保护，即INCLUDE\_MMU\_BASIC配置模块。

VxWorks 的内存管理函数存在于2 个库中 ：memPartLib (紧凑的内存分区管理器) 和memLib (完整功能的内存分区管理器)。memPartLib 提供的工具用于从内存分区中分配内存块。该库包含两类程序， 一类是通用工具memPartXXX()，包含创建和管理内存分区并从这些分区中分配和管理内存块；另一类是标准的malloc/free内存分配接口。系统内存分区(其ID为memSysPartId 是一个全局变量，关于它的定义在memLib.h 中)在内核初始化kernelInit() 是由usrRoot() (包含在usrConfig.c 中) 调用memInit 创建。其开始地址为RAM 中紧接着VxWorks 的BSS段之后的地址，大小为所有空闲内存。

VxWorks 5.5在目标板上有两个内存池（Memmory Pool）用于动态内存分配：系统内存池（System Memory Pool）和WDB内存池。对于VxWorks上程序的设计者来说，需要关注的是系统内存池的动态内存管理机制。嵌入式系统的动态内存管理实际上就是需要尽量避免动态分配和释放内存，以最大程度地保证系统的稳定性，VxWorks5.5的内存布局如图5.1所示。



5.1 VxWorks内存布局

如上图所示，VxWorks 5.5的内存按照装载内容不同从内存的低地址开始依次分为低端内存区、VxWorks内存区、WDB内存池、系统内存池和用户保留区五部分。各部分在内存中的位置由一些宏参数来决定，内存区域的划分及相关参数的定义与CPU的体系结构相关，这里只介绍VxWorks 5.5的典型内存布局。

整个目标板内存的起始地址是LOCAL\_MEM\_LOCAL\_ADRS，大小是LOCAL\_MEM\_SIZE，sysPhysMemTop()一般返回内存的物理最高地址。各部分的内容及参数如下：

(1)低端内存区

在低端内存区中通常包含了中断向量表、bootline（系统引导配置）和exception message(异常信息)等信息。

LOCAL\_MEM\_LOCAL\_ADRS是低端内存区的起始地址，典型设置是0，但是在Pentium平台一般设置位0x100000(即1M)。RAM\_LOW\_ADRS是低端内存区最高地址，也即vxWorks系统映像加载到内存中的地址，在Pentium平台一般为0x308000。

(2)VxWorks区域

VxWorks区存放操作系统映像，其中依次是VxWorks image的代码段、数据段和BSS段。

由RAM\_LOW\_ADRS和FREE\_MEM\_ADRS决定该区的大小和位置。

在Pentium平台RAM\_LOW\_ADRS为0x308000，FREE\_MEM\_ADRS通过链接脚本中的全部变量end指定，一般紧随VxWorks系统映像BSS段末尾。

(3)系统可用内存

系统可用内存主要是提供给VxWorks内存池用于动态内存的分配(如malloc())、任务的堆栈和控制块及VxWorks运行时需要的内存。这部分内存有VxWorks管理，开销位于目标板上。系统内存池在系统启动时初始化，它的大小是整个内存减去其他区的大小。在启动后可以通过函数memAddToPool()向系统内存池中增加内存，sysMemTop()返回系统内存池（System Memory Pool）的最高地址。

(4)WDB内存池

又叫Target Server内存池，是在目标板上为Tornado工具（如Wind Debugger）保留的一个内存池，主要用于动态下载目标模块、传送参数等。这个内存池由Target Server管理，管理的开销位于宿主机。Target Server内存池必要时（如Target Server内存池中内存不够）可以从系统内存池中分配内存。

其起始地址是WDB\_POOL\_BASE，通常等于FREE\_RAM\_ADRS，初始大小由WDB\_POOL\_SIZE定义，默认值是系统内存池的1/16，在installDir/target/config/all/configAll.h中定义:

#define WDB\_POOL\_SIZE((sysMemTop（）-FREE\_RAM\_ADRS)/16)

系统中不一定包含Target Server内存池.只有当系统配置中包含组件INCLUDE\_WDB时才需要Target Server内存池。

(5)用户保留区

用户保留区是用户为特定应用程序保留的内存。该区的大小由USER\_RESERVED\_MEM决定，默认值为0。以上所涉及的大部分宏定义和函数在目标板的BSP或configAll.h中定义。

**5.1.2 VxWorks内存分配策略**

嵌入式系统中为任务分配内存空间有两种方式，静态分配和动态分配。静态分配为系统提供了最好的可靠性与实时性，如可以通过修改USER\_RESERVED\_MEM 分配内存给应用程序的特殊请求用。对于那些对实时性和可靠性要求极高的需求，只能采用静态分配方式。但采用静态分配必然会使系统失去灵活性，因此必须在设计阶段考虑所有可能的情况，并对所有的需求做出相应的空间分配，一旦出现没有考虑到的情况，系统就无法处理。此外静态分配方式也必然导致很大的浪费，因为必须按照最坏情况进行最大的配置，而在实际运行中可能只用到其中的一小部分。因此一般系统中只有1 个内存分区，即系统分区，所有任务所需要的内存直接调用malloc()从其中分配。分配采用First-Fit算法(空闲内存块按地址大小递增排列，对于要求分配的分区容量size，从头开始比较，直至找到满足大小≥size 的块为止，并从链表相应块中分配出相应size 大小的块指针)，通过free释放的内存将被聚合以形成更大的空闲块。这就是VxWorks的动态内存分配机理。但是使用动态内存分配malloc/free时要注意到以下几个方面的限制：

* 因为系统内存分区是一种临界资源，由信号量保护，使用malloc 会导致当前调用挂起，所以它不能用于中断服务程序；
* 因为进行内存分配需要执行查找算法，其执行时间与系统当前的内存使用情况相关，是不确定的，所以对于有规定时限的操作它是不适宜的；
* 采用简单的最先匹配算法，容易导致系统中存在大量的内存碎片，降低内存使用效率和系统性能。

一般在系统设计时采用静态分配与动态分配相结合的方法。也就是说，系统中的一部分任务有严格的时限要求，而另一部分只是要求完成得越快越好。按照RMS(RateMonotonic Scheduling)理论，所有硬实时任务总的CPU 时间应小于70％，这样的系统必须采用抢先式任务调度；而在这样的系统中，就可以采用动态内存分配来满足那一部分可靠性和实时性要求不那么高的任务。

VxWorks采用最先适应法来动态分配内存，

优点：

* 满足嵌入式系统对实时性的要求；
* 尽可能的利用低地址空间，从而保证高地址空间有较大的空闲来放置要求内存较多的任务；

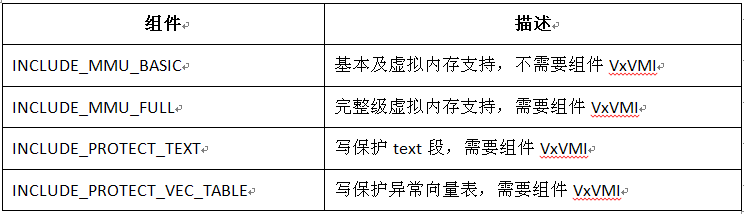
缺点：

VxWorks没有清除碎片的功能，只在内存释放时，采用了上下空闲区融合的方法，即把相邻的空闲内存块融合成一个空闲块。

**5.1.3 VxWorks对虚拟内存的支持**

VxWorks 5.5提供两级虚拟内存支持（Virtual Memory Support）：基本级（Basic Level）和完整级（Full Level）。后者需要购买可选组件VxVMI。在VxWorks 5.5中有关虚拟内存的配置包括两个部分。第一部分是对vxWorks虚拟内支持级别和保护的配置，下表列出了这些选项。

表5.1 VxWorks虚拟内存配置常量



以上配置一般在BSP的config.h中完成。

第二部分是内存页表的划分和属性配置。配置包括BSP的config.h中的VM\_PAGE\_SIZE和sysLib.c中的sysPhysMemDesc。

VM\_PAGE\_SIZE定义了CPU默认的页的大小。需参照CPU手册的MMU部分定义该值。

sysPhysMemDesc用于初始化MMU的TLB表，它是以PHYS\_MEM\_DESC为元素的常量数组。PHYS\_MEM\_DESC在vmLib.h中定义，用于部分内存的虚拟地址到物理地址的映射：

typedef struct phes\_mem\_desc

{

         void virtualAddr ;        /\*虚拟内存的物理地址\*/

         void \*physicalAddr ;   /\*虚拟地址\*/

         UNIT len ;                      /\*这部分的大小\*/

         UNIT initialStateMask ;

         UNIT initialState ;       /\*设置这部分内存的初始状态\*/

}PHYS\_MEM\_DESC;

sysPhysMemDesc中的值需要根据系统的实际配置进行修改。sysPhysMemDesc中定义的内存地址必须页对齐，且必须跨越完整的页。也就是说PHYS\_MEM\_DESC结构中的前三个域的值必须能被VM\_PAGE\_SIZE整除，否则会导致VxWorks初始化失败。

基本级虚拟内存库vmBaseLib提供了系统中所需最低限度的MMU支持，其主要目的是为创建Cache-safe缓冲区提供支持。

**5.2 VxWorks内存分配算法**

**5.2.1 VxWorks核心数据结构**

VxWorks系统初始化时创建系统内存分区，VxWorks定义了全局变量memSysPartition来管理系统内存分，用户也可以使用memPartLib库的函数实现自己的分区。用于分区中内存管理的数据结构是在memPartLib.h中定义的mem\_part，包含对象标记、保护该分区的信号量、一些分配统计信心（如当前分配的块数）及分区中所有的空闲内存快形成的一个双向链表freeList(在VxWorks6.8中，空闲内存块采用平衡二叉树来组织)。

具体定义如下：

typedef struct mem\_part

{

    OBJ\_CORE    objCore;         /\* 对象标识 \*/

    DL\_LIST     freeList;              /\* 空闲链表 \*/

    SEMAPHORE   sem;            /\* 分区信号量，保护该分区互斥访问 \*/

    unsigned    totalWords;     /\* 分区中的字数(一个字=两个字节) \*/

    unsigned    minBlockWords;  /\* 以字为单位的最小块数，包含头结构 \*/

    unsigned    options;        /\* 选项，用于调试和统计\*/

    /\*分配统计信息\*/

    unsigned curBlocksAllocated;    /\*当前分配的块数 \*/

    unsigned curWordsAllocated;     /\* 当前分配的字数 \*/

    unsigned cumBlocksAllocated;    /\* 累积分配的块数 \*/

    unsigned cumWordsAllocated;     /\* 累积分配的字数\*/

} PARTITION;

备注：需要注意的是内存分区信号量sem是分区描述符中的一个成员，而不是指向动态创建的信号量结构的指针，采用静态分配的方式，是从提高系统性能的角度考虑。

VxWorks在初始化的过程中，通过usrInit()->usrKernelInit()->kernelInit()将系统分配给vxWorks内存池的基地址MEM\_POOL\_START和大小，通过usrRoot()的参数传递给memInit()函数。换句话说：vxWorks系统分区memSysPartition的初始化由usrRoot()->memInit()来完成，而其中内存的布局则通过usrInit()->usrKernelInit()->KernelInit()传递的参数来确定，

usrKernelInit()传递给kernelInit()的参数如下：

kernelInit ((FUNCPTR) usrRoot, ROOT\_STACK\_SIZE, MEM\_POOL\_START,

                sysMemTop (), ISR\_STACK\_SIZE, INT\_LOCK\_LEVEL);

其中ROOT\_STACK\_SIZE为10000=0x2710，

MEM\_POOL\_START为end，这里我们假定为0x3c33a0

sysMemTop ()返回的是最大可用内存，这里假定为32M，即sysMemTop ()返回值为0x200 0000。

ISR\_STACK\_SIZE值为1000=0x3e8

INT\_LOCK\_LEVEL值为0

我们假设系统可用内存为32M，VxWorks的入口地址为0x30800c，end的值，即VxWorks内核映像BSS段末尾地址；

sysMemTop ()返回的值为0x2000000，作为VxWorks内存池的最高地址pMemPoolEnd。VxWorks在内存中的布局如图5.2所示。

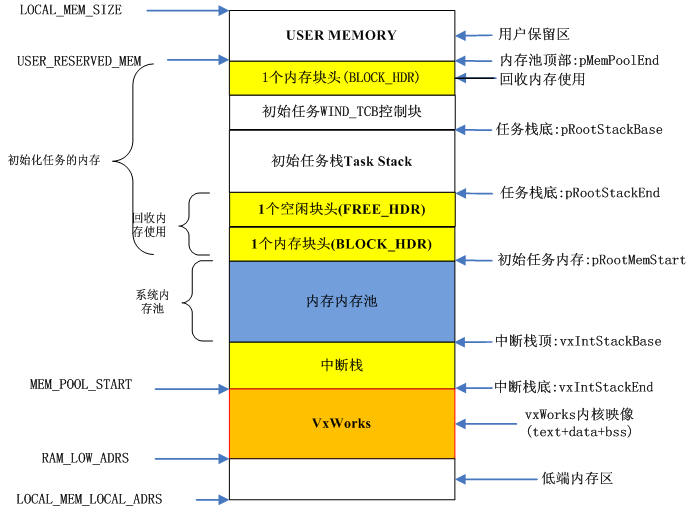


图5.2 VxWorks内部布局

由于我们不配置WDB模块，故没有标出WDB内存池的分配。

我们仍以Pentium平台为例，这里假设VxWorks内核映像加载到内存的地址RAW\_LOW\_ADRS为0x30 8000，LOCAL\_MEM\_LOCAL\_ADDR定位为0x10 0000，即1M字节位置。

Pentium平台的全局描述符表放在的0x10 0000+0x1000=1M+4K位置处的80字节内存范围。

中断描述符表放置在0x10 0000开始的2K字节内存范围。

这样中断栈的栈底vxIntStackEnd=end=0x3c33a0，

中断栈的栈基址为vxIntStackBase= end+ISR\_STACK\_SIZE=0x3c33a0+0x3e8=0x3c3788

将要分配给系统分区内存基地址pMemPoolStart=vxIntStackBase=0x3c3788

用于初始任务tRootTask的内存起始地址为：

pRootMemStart       =pMemPoolEnd-ROOT\_STACK\_SIZE

=pMemPoolEnd-10000

=0x2000000-0x2710

=0x1ff d8f0

用于初始任务tRootTask的任务栈大小为：

rootStackSize  = rootMemNBytes - WIND\_TCB\_SIZE - MEM\_TOT\_BLOCK\_SIZE

=10000 – 416 –( (2 \* MEM\_BLOCK\_HDR\_SIZE) + MEM\_FREE\_BLOCK\_SIZE)

=10000 – 416 – 32=9532=0x253c

用于tRootTask任务栈的栈顶：

pRootStackBase= pRootMemStart + rootStackSize + MEM\_BASE\_BLOCK\_SIZE

=pRootMemStart+rootStackSize+(MEM\_BLOCK\_HDR\_SIZE+MEM\_FREE\_BLOCK\_SIZE)

=0x1ff d8f0+0x253c+0x8+0x20

= 0x1ff fe54

这样32M的内存就被分成了三个部分：

0x10 0000到0x3c33a0 用于存放vxWorks内核映像的代码段、数据段、BSS段；

0x3c33a0到0x3c3788 用作vxWorks内核的中断栈

0x3c3788(即end+1000) 到0x1ff d8f0(即0x200 0000-0x2710)用于VxWorks的内存池；

0x1FFFE54-0x253c(即十进制数9532)到0x1FFFE54用作tRootTask任务栈的栈底；

这样的话，用于创建初始任务tRootTask的TCB控制块构架接口如下：

    taskInit (pTcb, "tRootTask", 0, VX\_UNBREAKABLE | VX\_DEALLOC\_STACK,

               pRootStackBase, (int) rootStackSize, (FUNCPTR) rootRtn,

               (int) pMemPoolStart, (int)memPoolSize, 0, 0, 0, 0, 0, 0, 0, 0);

这里的pMemPoolStart就是图中vxIntStackBase，即end+1000=0x3c 3788

内存池的范围从vxIntStackBase到pRootMemStart的内存空间，

即0x1ff d8f0-0x3c 3788=1C3 A168,约为28.2M，这一段区域作为初始任务tRootTask执行代码usrRoot()的入口参数传递给memInit ()函数用于创建初始分区。

memInit()函数的最重要作用是创建内存分区类memPartClass和内存分区类的一个全局的系统对象memSysPartition，并用指定的内存块(pMemPoolStart, pMemPoolStart + memPoolSize)来初始化系统内存对象memSysPartition的分区空闲链表。

这样得到的VxWorks内存分区逻辑图如图5.3所示。

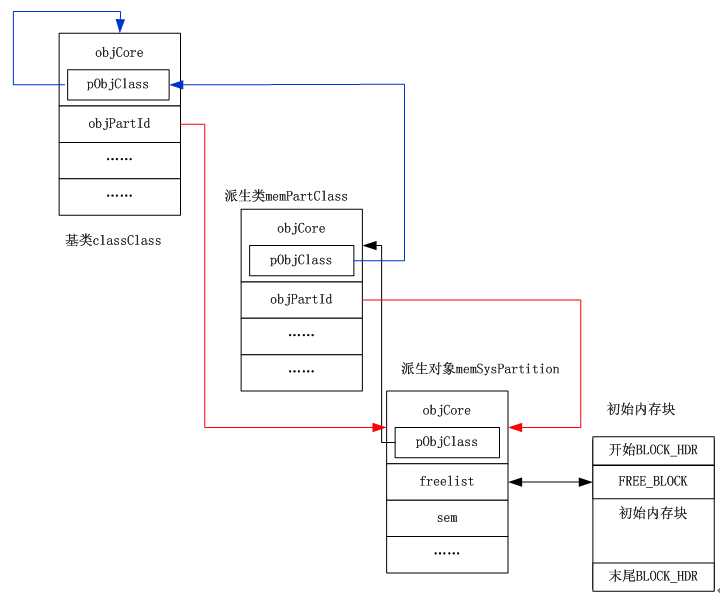


图5.3 VxWorks内存分区逻辑图

**6.2.2 VxWorks核心分区初始化**

usrInit()->usrKernelInit()->kernelInit()构建并启动初始化任务tRootTask，执行usrRoot()

usrRoot()-> memInit (pMemPoolStart, memPoolSize)，我们从memInit()开始分析。

还是使用上面的示例：

pMemPoolStart =0x3c3788(即end+1000) 到0x1ff d8f0(即0x200 0000-0x2710)用于VxWorks的初始化内存分区。

memPoolSize=0x1ff d8f0-0x3c3788=0x1C3 A168,约为28.2M，示意图如图5.2所示。

memInit()代码如下：

STATUS memInit ( char \*pPool, unsigned poolSize)

{

memLibInit ();           /\*初始化化的内存管理函数 \*/

//初始化内存对象

//初始化系统内存分区

//将分配的(pMemPoolStart, memPoolSize)约28.2M加入系统内存分区

    return (memPartLibInit (pPool, poolSize));

 }

**6.2.2.1 memLibInit ()分析**

STATUS memLibInit (void)

{

    if (!memLibInstalled)

         {

                  \_func\_valloc                          = (FUNCPTR) valloc;

                  \_func\_memalign                  = (FUNCPTR) memalign;

                  memPartBlockErrorRtn         = (FUNCPTR) memPartBlockError;

                  memPartAllocErrorRtn          = (FUNCPTR) memPartAllocError;

                  memPartSemInitRtn           = (FUNCPTR) memSemInit;

             memPartOptionsDefault          = MEM\_ALLOC\_ERROR\_LOG\_FLAG |

                                      MEM\_BLOCK\_ERROR\_LOG\_FLAG |

                                      MEM\_BLOCK\_ERROR\_SUSPEND\_FLAG |

                                      MEM\_BLOCK\_CHECK;

         memLibInstalled = TRUE;

         }

    return ((memLibInstalled) ? OK : ERROR);

}

分析：初始化VxWorks提供的更高层的内存管理接口，例如：

\_func\_valloc初始化为虚拟地址按页边界对齐的从系统分区分配内存的函数valloc;

\_func\_memalign初始化为安装自定义对齐方式从系统分区分配内存的函数memalign；

memPartSemInitRtn用于指定初始化系统分区memSysPartition的信号量sem的类型，这里使用具有优先级队列、安全删除和防止优先级翻转的互斥信号量的初始化函数memSemInit；

另外两个接口定义出错处理方式。

**6.2.2.2 memPartLibInit (pPool, poolSize)分析**

memPartLibInit()初始化了VxWorks的内存对象类和内存对象类的一个实例系统分区，并将配置的28.2M的内存块加入到系统分区当中。

STATUS memPartLibInit (char \*pPool, unsigned poolSize )

{

    if ((!memPartLibInstalled) &&

           (classInit (memPartClassId, sizeof (PARTITION),

                       OFFSET (PARTITION, objCore), (FUNCPTR) memPartCreate,

                       (FUNCPTR) memPartInit, (FUNCPTR) memPartDestroy) == OK))

         {

                  memPartInit (&memSysPartition, pPool, poolSize);

                  memPartLibInstalled = TRUE;

         }

    return ((memPartLibInstalled) ? OK : ERROR);

}

分析：

我们在第1章概述中提到，vxWorks采用类和对象的思想将wind内核的任务管理模块、内存管理模块、消息队列管理模块、信号量管理模块、以及看门狗管理模块组织起来，各个对象类都指向元类classClass，每个对象类只负责管理各自的对象，如图5.4。

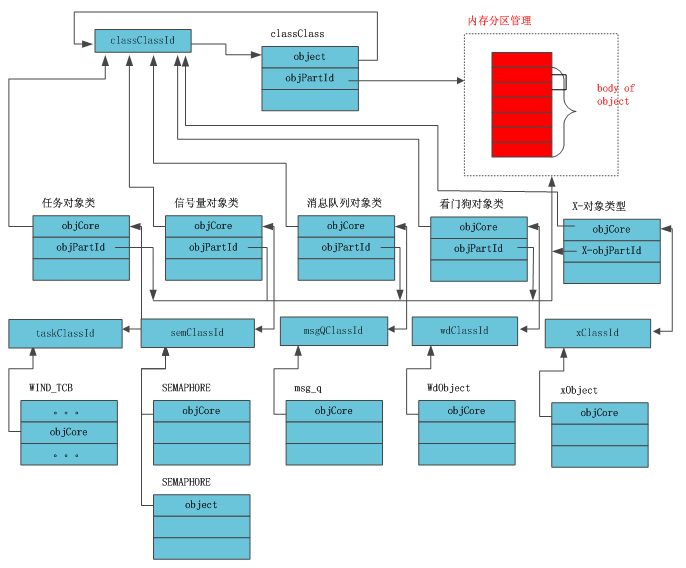


图5.4 Wind内核对象组织关系图

本篇所分析的内存分区memSysPartition就是内存分区对象类memPartClass的一个实例，即系统分区，其红色部分的组织形式如图5.3所示。

classInit (memPartClassId, sizeof (PARTITION),

OFFSET (PARTITION, objCore), (FUNCPTR) memPartCreate,

(FUNCPTR) memPartInit, (FUNCPTR) memPartDestroy)初始化图中的内存对象类memPartClass，其代码如下：

STATUS classInit

(

    OBJ\_CLASS   \*pObjClass,    /\* pointer to object class to initialize \*/

    unsigned    objectSize,     /\* size of object \*/

    int         coreOffset,           /\* offset from objCore to object start \*/

    FUNCPTR     createRtn,    /\* object creation routine \*/

    FUNCPTR     initRtn,          /\* object initialization routine \*/

    FUNCPTR     destroyRtn    /\* object destroy routine \*/

)

{

    /\* default memory partition is system partition \*/

    pObjClass->objPartId       = memSysPartId;     /\* partition to allocate from \*/

    pObjClass->objSize           = objectSize;    /\* record object size \*/

    pObjClass->objAllocCnt   = 0;            /\* initially no objects \*/

    pObjClass->objFreeCnt    = 0;            /\* initially no objects \*/

    pObjClass->objInitCnt      = 0;            /\* initially no objects \*/

    pObjClass->objTerminateCnt  = 0;            /\* initially no objects \*/

    pObjClass->coreOffset    = coreOffset;   /\* set offset from core \*/

    /\* initialize object methods \*/

    pObjClass->createRtn     = createRtn;    /\* object creation routine \*/

    pObjClass->initRtn            = initRtn;          /\* object init routine \*/

    pObjClass->destroyRtn   = destroyRtn;  /\* object destroy routine \*/

    pObjClass->showRtn                 = NULL;    /\* object show routine \*/

    pObjClass->instRtn           = NULL;             /\* object inst routine \*/

    /\* 初始化内存对象类memPartClass为合法的对象类 \*/

//内存对象类memPartClass指向其wind内核的元类classClass

    objCoreInit (&pObjClass->objCore, classClassId);

    return (OK);

}

接着由memPartInit (&memSysPartition, pPool, poolSize)初始化内存队列类memPartClass的实例对象memSysPartition，并将配置的内存池分配给系统分区memSysPartition。

**6.2.2.3 memPartInit (&memSysPartition, pPool, poolSize)分析**

memPartInit()将初始化系统分区memSysPartition，并将配置的内存池分配给系统分区memSysPartition，是代码实现如下：

void memPartInit ( FAST PART\_ID partId, char \*pPool, unsigned poolSize )

{

    /\*初始化分区描述符 \*/

    bfill ((char \*) partId, sizeof (\*partId), 0);

    partId->options          = memPartOptionsDefault;

    partId->minBlockWords = sizeof (FREE\_BLOCK) >> 1;

    /\* initialize partition semaphore with a virtual function so semaphore

     \* type is selectable.  By default memPartLibInit() will utilize binary

     \* semaphores while memInit() will utilize mutual exclusion semaphores

     \* with the options stored in \_mutexOptionsMemLib.

     \*/

//通过调用一个函数指针memPartSemInitRtn，来初始化分区描述符的信号量，采用这种方

//式的好处是信号量类型的选择是可选的。默认情况下

//memPartLib库中将其初始化二进制信号量，但是memInit()将会使用存放在

//\_mutexOptionsMemLib中的属性值来初始化互斥信号量。

    (\* memPartSemInitRtn) (partId);

    dllInit (&partId->freeList);                          /\*初始化空闲链表 \*/

    objCoreInit (&partId->objCore, memPartClassId);          /\* initialize core \*/

    (void) memPartAddToPool (partId, pPool, poolSize);

}

分析：

系统分区memSysPartition的属性初始化为：

memPartOptionsDefault          =  MEM\_ALLOC\_ERROR\_LOG\_FLAG |

MEM\_BLOCK\_ERROR\_LOG\_FLAG |

MEM\_BLOCK\_ERROR\_SUSPEND\_FLAG |

MEM\_BLOCK\_CHECK;

memSysPartition的分区做小大小为sizeof (FREE\_BLOCK)个字节，在Pentium平台为16个字节。

**6.2.2.4 将初始内存块加入系统分区内存池中**

VxWorks调用函数memPartAddToPool (&memSysPartition, pPool, poolSize)来完成这一功能，我们来分析这一函数的具体实现过程：

我们要加入的内存块区域是从0x3c3788(即end+1000) 到0x1ff d8f0(即0x200 0000-0x2710)用于vxWorks的初始化内存分区，大小为0x1ff d8f0-0x3c3788=0x1C3A168=29598056字节

从0x3c3788开始的8个字节作为块的开头，其初始化代码如下：

pHdrStart                   = (BLOCK\_HDR \*) pPool;// 0x3c3788

pHdrStart->pPrevHdr = NULL;

pHdrStart->free     = FALSE;

pHdrStart->nWords   = sizeof (BLOCK\_HDR) >> 1;//nWorks=8>>1=4

从0x3c3790开始的8个字节，其初始化代码如下：

pHdrMid           = NEXT\_HDR (pHdrStart);// 0x3c3788

pHdrMid->pPrevHdr   = pHdrStart;

pHdrMid->free       = TRUE;

pHdrMid->nWords     = (poolSize - 2 \* sizeof (BLOCK\_HDR)) >> 1;

//(29598056-2\*8)/2=14799020=0xE1D0AC

由于pHdrMid->free       = TRUE，所以0x3c3790出的值为0x80e1d0ac。

由于通过BLOCK\_HDR的结构体中，nWords和free共用了4个字节的长度，nWords占用了低0~30bit位，free占用了第31bit位。

typedef struct blockHdr            /\* BLOCK\_HDR \*/

{

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 \*/

} BLOCK\_HDR;

所以其在内存中的布局如图5.5。

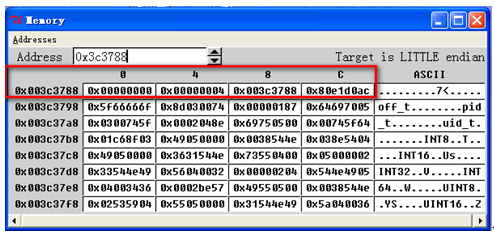


图5.5 内存块头在内存中的布局

以上是内存块的头部的设置，我们在看下尾部的设置，对要加入的内存块区域是从0x3c3788(即end+1000) 到0x1ff d8f0(即0x200 0000-0x2710)用于vxWorks的初始化内存分区，最后的8个字节表示的块头的初始化结果。

特别需要注意的块头地址0x1ffd8e8，是从初始化内存分区尾端分出8个字节，其初始化代码如下：

pHdrEnd            = NEXT\_HDR (pHdrMid);//指向的是0x373C90

pHdrEnd->pPrevHdr   = pHdrMid;

pHdrEnd->free       = FALSE;

pHdrEnd->nWords     = sizeof (BLOCK\_HDR) >> 1;

其初始化的结构，正如图5.6的内存区域所示。

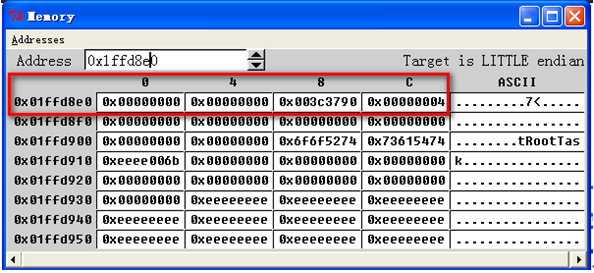


图5.6 内存块尾部结构布局

从上面的初始化过程，我们看到vxWorks把内存块区域是从0x3c3788(即end+1000) 到0x1ff d8f0(即0x200 0000-0x2710)用于vxWorks的初始化内存分区加入的分区描述符memSysPartition.freeList中去时，真正加入的区域从送0x3c3788+8到0x1ff d8f0-8这个区域，即从这个初始分区的开头和末尾处分别划出了一个8字节的区域填写两个内存块的头结构BLK\_HDR。以表示与初始化分区相邻的两个内存块，只不过这两个内存块只有头部，数据部分为0。这样做的目的是为了将来在释放0x3c3788+8开始的内存块时，可以0x3c3788位置的空白块合并；以及在释放以0x1ff d8f0-8结束的内存块时可以和0x1ff d8f0位置的空白内存块合并。以使得vxWorks分区内存管理模块对0x3c3788+8开始的内存块和0x1ff d8f0-8结束的内存块才操作进行统一。此时sysMemPartition的 布局如图5.7所示。

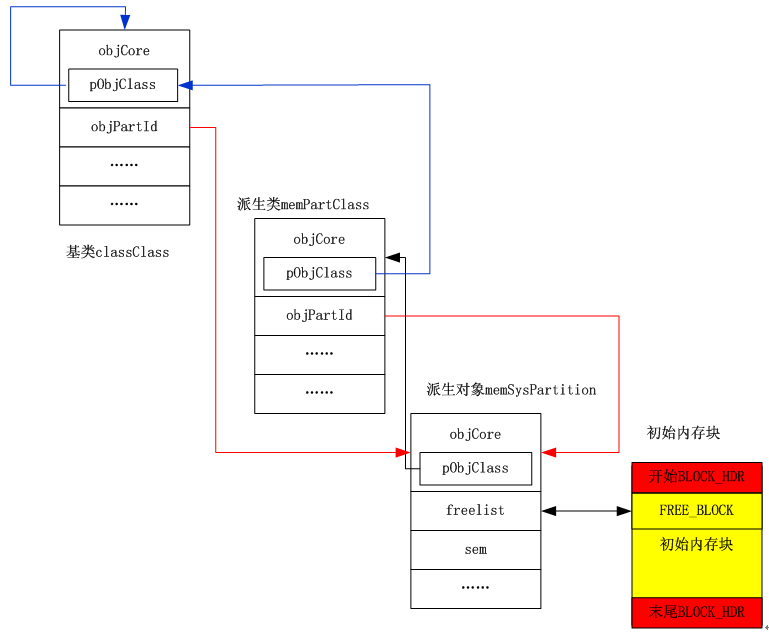


图5.7 sysMemPartition的 内部布局

通过上面的描述，memPartAddToPool (&memSysPartition, pPool, poolSize)的具体实现如下：

STATUS memPartAddToPool ( PART\_ID partId,      char \*pPool,   unsigned poolSize )

{

    FAST BLOCK\_HDR \*pHdrStart;

    FAST BLOCK\_HDR \*pHdrMid;

    FAST BLOCK\_HDR \*pHdrEnd;

    char \*          tmp;

    int             reducePool;                 /\*内存池的实际减少量\*/

    if (OBJ\_VERIFY (partId, memPartClassId) != OK)//验证partId合法性

         return (ERROR);

    /\*确保内存池的实际开始地址是4字节对齐(假设是Pentium平台) \*/

    tmp       = (char \*) MEM\_ROUND\_UP (pPool);          /\* 获取实际的其实地址 \*/

    reducePool = tmp - pPool;

    if (poolSize >= reducePool)                       /\* 调整内存池的长度\*/

        poolSize -= reducePool;

    else

        poolSize = 0;

    pPool     = tmp;//调整内存池的开始位置

    /\*

\*确保内存池大小poolSize是4字节的整数倍，并且至少包含3个空闲内存块的块头和

\*1个空闲内存块(仅有包含一个块头)

     \*/

    poolSize = MEM\_ROUND\_DOWN (poolSize);

    if (poolSize < ((sizeof (BLOCK\_HDR) \* 3) + (partId->minBlockWords \* 2)))

         {

         errno = S\_memLib\_INVALID\_NBYTES;

        return (ERROR);

         }

    /\* 初始化化3个内存块头 -

     \* 内存块的开头和结束各有一个块头，并且还有一个代码真正的内存块 \*/

    pHdrStart                   = (BLOCK\_HDR \*) pPool;

    pHdrStart->pPrevHdr = NULL;

    pHdrStart->free     = FALSE;

    pHdrStart->nWords   = sizeof (BLOCK\_HDR) >> 1;

    pHdrMid           = NEXT\_HDR (pHdrStart);

    pHdrMid->pPrevHdr   = pHdrStart;

    pHdrMid->free       = TRUE;

pHdrMid->nWords     = (poolSize - 2 \* sizeof (BLOCK\_HDR)) >> 1;

//中间的内存块头代码真正的内存块，其大小应除去开头和结尾两个内存块头结构占用

//的内存

    pHdrEnd            = NEXT\_HDR (pHdrMid);

    pHdrEnd->pPrevHdr   = pHdrMid;

    pHdrEnd->free       = FALSE;

    pHdrEnd->nWords     = sizeof (BLOCK\_HDR) >> 1;

    semTake (&partId->sem, WAIT\_FOREVER);

//从中我们可以看出sysMemPartition中的信号量是保证空闲块链表必须被互斥访问

    dllInsert (&partId->freeList, (DL\_NODE \*) NULL, HDR\_TO\_NODE (pHdrMid));

    partId->totalWords += (poolSize >> 1);

    semGive (&partId->sem);

    return (OK);

}

分析：这里有必要分析一下将内存块插入系统分区sysMemPartition的空闲链表freeList的操作：

dllInsert (&partId->freeList, (DL\_NODE \*) NULL, HDR\_TO\_NODE (pHdrMid));

内存池的第二个内存块块(pHdrMid指向)插入freeList，这里存在一个强制类型转换过程。

typedef struct blockHdr            /\* 内存块头BLOCK\_HDR \*/

{

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 \*/

} BLOCK\_HDR;

空闲内存块头结构是在BLOCK\_HDR的基础上，加上DL\_NODE类型的成员变量node。

typedef struct                    /\* 空闲内存块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;

因此其加入sysMemPartition.freeList的过程如下：

dllInsert (&partId->freeList, (DL\_NODE \*) NULL, HDR\_TO\_NODE (pHdrMid));

#define HDR\_TO\_NODE(pHdr)        (& ((FREE\_BLOCK \*) pHdr)->node)，其组合示意如图5.8所示。

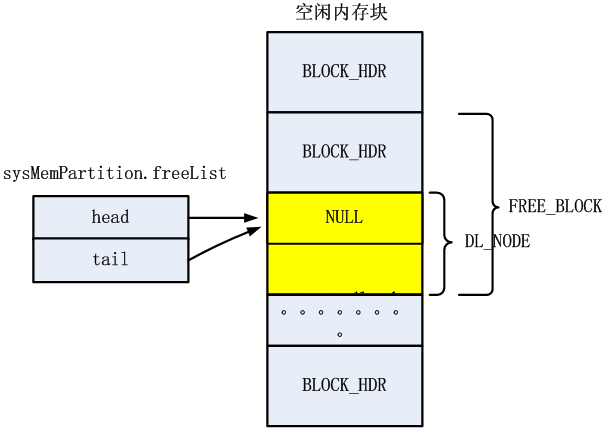


图5.8 空闲块头示意图

**5.2.3 VxWorks内存分配机制**

VxWorks中的动态内存分配采用最先匹配(First-Fit)算法，即从空闲链表中查找内存块，然后从高地址开始查找，当找到第一个满足分配请求的空闲内存块时，就分配所需内存并修改该空闲块的大小。空闲块的剩余部分仍然保留在空闲链表中。当从大的内存块中分配出新的内存块时，需要将新内存块头部的一块空间(称为“块头”)用来保存分配、回收和合并等操作的信息，因此，实际占用的内存大小是分配请求的内存大小与块头开销之和。

**6.2.3.1 malloc()分配内存机制**

分配函数malloc()返回的地址是分配请求所获可用内存的起始地址，为优化性能，malloc()会自动对齐边界，分配的内存的起始地址和大小都是边界之的整数倍，因此有可能造成内部碎片。块头的开销与处理器的体系结构有关，对于X86体系结构来说，，块头的开销是8字节，通常这部分会被自动以4字节边界对齐，以减少出现碎片的可能并优化性能。

void \*malloc (size\_t nBytes )

 {

    return (memPartAlloc (&memSysPartition, (unsigned) nBytes));

 }

malloc()是memPartAlloc()的进一步封装，并且从系统内存分区memSysPartition分配内存。

memPartAlloc()函数如下：

void \*memPartAlloc (PART\_ID partId, unsigned nBytes)

{

    return (memPartAlignedAlloc (partId, nBytes, memDefaultAlignment));

}

memPartAlloc()又是memPartAlignedAlloc()的进一步封装，其中memDefaultAlignment在Pentium平台是4字节。

memPartAlignedAlloc()从指定的系统分区memSysPartition中分配nBytes字节的内存，分配的字节数必须要被memDefaultAlignment整除，其中memDefaultAlignment必须为2的幂数。

首次适应算法就体现在函数memPartAlignedAlloc()的实现上：

void \*memPartAlignedAlloc (PART\_ID    partId, unsigned int nBytes, unsigned int alignment)

{

    FAST unsigned nWords;

    FAST unsigned nWordsExtra;

    FAST DL\_NODE \*      pNode;

    FAST BLOCK\_HDR \* pHdr;

    BLOCK\_HDR \*          pNewHdr;

    BLOCK\_HDR            origpHdr;

    if (OBJ\_VERIFY (partId, memPartClassId) != OK)

                  return (NULL);

    /\* 实际分配的内存大小为请求分配内存大小+块头大小 \*/

    nWords = (MEM\_ROUND\_UP (nBytes) + sizeof (BLOCK\_HDR)) >> 1;

    /\*检查是否溢出，如果溢出，设置errno，并返回NULL\*/

    if ((nWords << 1) < nBytes)

         {

                  if (memPartAllocErrorRtn != NULL)

                      (\* memPartAllocErrorRtn) (partId, nBytes);

                  errnoSet (S\_memLib\_NOT\_ENOUGH\_MEMORY);

//如果分区设置了MEM\_ALLOC\_ERROR\_SUSPEND\_FLAG，则当前请求内存的任务被阻塞

                  if (partId->options & MEM\_ALLOC\_ERROR\_SUSPEND\_FLAG)

             {

                      if ((taskIdCurrent->options & VX\_UNBREAKABLE) == 0)

                                     taskSuspend (0);                         /\* 阻塞自己\*/

             }

                  return (NULL);

         }

//如果当前请求分配的内存小于分区允许的最小内存(Pentium平台，8个字节)，则以最小内

//存为准

    if (nWords < partId->minBlockWords)

         nWords = partId->minBlockWords;

    /\* 获取分区信号量，互斥范围分区空闲块链表，在这里如果信号量已经被占用，则当

\*前请求分配内存的任务将会被阻塞，等待该信号量释放，这也是malloc()会导致被调用\*任务阻塞的根源\*/

    semTake (&partId->sem, WAIT\_FOREVER);

    /\* 首次适应算法，就体现在下面的链表查找中 \*/

    pNode = DLL\_FIRST (&partId->freeList);

/\* 我们需分配一个空闲的块，并带有额外的空间用于对齐，最坏的情况是我们需要一\*个对齐的额外字节数

      \*/

    nWordsExtra = nWords + alignment / 2;

    FOREVER

         {

                  while (pNode != NULL)

             {

//如果当前块大于包含额外对齐字节空间的请求字节数，或者

//当前块和请求的字节数刚好一致，并且满足对齐条件

//则当前块满足要求

             if ((NODE\_TO\_HDR (pNode)->nWords > nWordsExtra) ||

                     ((NODE\_TO\_HDR (pNode)->nWords == nWords) &&

                     (ALIGNED (HDR\_TO\_BLOCK(NODE\_TO\_HDR(pNode)), alignment))))

                     break;

             pNode = DLL\_NEXT (pNode);

             }

             //如果找不到满足要求的当前块

         if (pNode == NULL)

         {

             semGive (&partId->sem);

             if (memPartAllocErrorRtn != NULL)

                            (\* memPartAllocErrorRtn) (partId, nBytes);

             errnoSet (S\_memLib\_NOT\_ENOUGH\_MEMORY);

//默契情况下memLibInit()中设置memSysPartition.options如下表所示

// memPartOptionsDefault     = MEM\_ALLOC\_ERROR\_LOG\_FLAG |

//                                   MEM\_BLOCK\_ERROR\_LOG\_FLAG |

//                                   MEM\_BLOCK\_ERROR\_SUSPEND\_FLAG |

//                                   MEM\_BLOCK\_CHECK

             if (partId->options & MEM\_ALLOC\_ERROR\_SUSPEND\_FLAG)

                  {

                            if ((taskIdCurrent->options & VX\_UNBREAKABLE) == 0)

                               taskSuspend (0);                         /\* suspend ourselves \*/

                   }

             return (NULL);

          }

//找到满足的内存块

         pHdr = NODE\_TO\_HDR (pNode);

         origpHdr = \*pHdr;

//从当前块中分配出用户指定的内存，需要注意的是vxWorks从空闲块的末端开始分配捏成

//这样的好处是，余下的部分可以仍然在空闲链表中。如果memAlignedBlockSplit()返回为

//NULL，意味着将会使第一个内存块余下的部分调小，而不能放在空闲链表中,

//这种情况下，vxWorks会继续尝试使用下一个内存块

         pNewHdr =

             memAlignedBlockSplit (partId, pHdr, nWords, partId->minBlockWords,

                                       alignment);

         if (pNewHdr != NULL)

             {//找到并分配处用户指定的内存块，退出循环

             pHdr = pNewHdr;                       /\* give split off block \*/

             break;

             }

         //当前块不合适的话，继续尝试

         pNode = DLL\_NEXT (pNode);

         }

    /\*标记分配出来的空闲块为非空闲 \*/

    pHdr->free = FALSE;

    /\* 更新memSysPartition分区的统计量 \*/

    partId->curBlocksAllocated++;

    partId->cumBlocksAllocated++;

    partId->curWordsAllocated += pHdr->nWords;

    partId->cumWordsAllocated += pHdr->nWords;

//是否信号量

    semGive (&partId->sem);

//跳过块头，返回可用内存的地址

    return ((void \*) HDR\_TO\_BLOCK (pHdr));

}

这里比较复杂的函数是，当memPartAlignedAlloc()找到合适的块后，从该空闲块分割出用户所需要内存所考虑的情况，下面我们会分析这个分割函数。

**6.2.3.2 内存块的分割**

分配内存时，首先找到一个合适的空闲内存块，然后从该块上分割出一个新块（包含头部）。最后把该新块的内存区域返回给用户，因此头部对用户的透明的。

分配时，有时为了对齐，会将一个块一分为三，第一块是申请后剩余空间构成的块，第二块是申请块的目标块，第三块是用第二块为了对齐而剩余的内存空间构成的块。如果第三块太小，则不将其独立出来，而是作为第二块的一部分。

示意图如5.9所示。

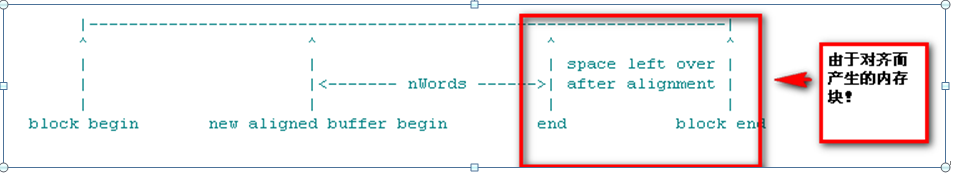


图5.9 内存块分割示意图

memAlignedBlockSplit()函数是空闲内存块分割的核心函数，设从A块申请nWords字节的内存大小，要求alignment字节对齐，其代码分析如下：

（1）将从A块的尾部endOfBlock往上偏移nWords-BLOCK\_HDR字节处pNewBlock作为新块的头部地址。

（2）pNewBlock往上偏移，以使之alignment字节对齐。这一步导致申请块的后面出现了剩余空间,正如上图中所示。

（3）计算块A的剩余空间blockSize=(char \*) pNewHdr - (char \*) pHdr，若blockSize小于最小块大小minWords，且新块的起始地址pNewHdr等于块A的起始地址pHdr，那么新快就从块A的起始地址开始，即将块A分配出去，否则申请失败。若blockSize大于最小块minWords，则更新块A的大小，即块A仍然存在只是其内存空间变小了。

（4）计算新块尾部剩余多小空间endOfBlock - (UINT) pNewHdr - nWords，若剩余的空间小于最小块大小minWords，则将该剩余空间也分配给申请的块，否则为该剩余空间构建一个新块，并插入内存分区的空闲链表。

（5）返回pNewHdr。

memAlignedBlockSplit()实现如下：

LOCAL BLOCK\_HDR \*memAlignedBlockSplit

(

    PART\_ID partId,

    BLOCK\_HDR \*pHdr,

    unsigned nWords,            /\* 需要分割出来的字数，包含块头 \*/

    unsigned minWords,          /\* 所允许的最小字数 \*/

    unsigned alignment                          /\* 边界对齐数 \*/

)

{

    FAST BLOCK\_HDR \*pNewHdr;

    FAST BLOCK\_HDR \*pNextHdr;

    FAST char \*endOfBlock;

    FAST char \*pNewBlock;

    int blockSize;

    /\*计算出当前块的末尾位置 \*/

    endOfBlock = (char \*) pHdr + (pHdr->nWords \* 2);

    /\* 计算出新块的起始位置 \*/

//通过memPartAlignedAlloc()调用函数的分析，我们指定nWords中起始已经包含了

//块头的位置，所以这里在分配内存是只考虑实际使用的内存。

    pNewBlock = (char \*) ((unsigned) endOfBlock -

                   ((nWords - sizeof (BLOCK\_HDR) / 2) \* 2));

    /\* 通过边界向内对齐调整内存块起始位置,这将使得分配的内存偏大 \*/

    pNewBlock = (char \*)((unsigned) pNewBlock & ~(alignment - 1));

//将确定的内存块起始位置假设块头大小，这里才考虑进了块头的大小

    pNewHdr = BLOCK\_TO\_HDR (pNewBlock);

    /\* 分割之后剩下的块的大小 \*/

    blockSize = ((char \*) pNewHdr - (char \*) pHdr) / 2;

    if (blockSize < minWords)

         {

//如果分割之后剩下的内存块，小于分区规定的最小内存，并且切换分割出去的内存块

//恰好就是原来的内存块，则将原来的内存块从空闲块链表中删除

//否则，分割之后剩余的内存太小，不足以继续挂载空闲块链表上，则函数返回NULL；

// memPartAlignedAlloc()将会尝试这从下一个空闲块中继续分割

     if (pNewHdr == pHdr)

         dllRemove (&partId->freeList, HDR\_TO\_NODE (pHdr));

     else

         return (NULL);

         }

    else

         {

         pNewHdr->pPrevHdr = pHdr;

         pHdr->nWords = blockSize;

         }

//检查由于新块地址对齐导致的多出来的内存碎片，是否足够大

//足够足够大，则单独作为一个空闲块插入空闲块链表；

//否则并入新分割出来的块中。

    if (((UINT) endOfBlock - (UINT) pNewHdr - (nWords \* 2)) < (minWords \* 2))

         {

         /\* 将产生的碎片全部并入新分割出来的块中 \*/

         pNewHdr->nWords = (endOfBlock - pNewBlock + sizeof (BLOCK\_HDR)) / 2;

         pNewHdr->free     = TRUE;

         /\*调整后面的空闲块，使其指向新分割出来的块\*/

         NEXT\_HDR (pNewHdr)->pPrevHdr = pNewHdr;

         }

    else

         {

         /\* the extra bytes are big enough to be a fragment on the free list -

          \* first, fix up the newly allocated block.

          \*/

//余下的碎片最够大，首先让其成为一个单独的块

         pNewHdr->nWords = nWords;

         pNewHdr->free     = TRUE;

         /\*将这个单独的块加入空闲块链表\*/

         pNextHdr = NEXT\_HDR (pNewHdr);

         pNextHdr->nWords = ((UINT) endOfBlock - (UINT) pNextHdr) / 2;

         pNextHdr->pPrevHdr = pNewHdr;

         pNextHdr->free = TRUE;

//加入空闲块链表

         dllAdd (&partId->freeList, HDR\_TO\_NODE (pNextHdr));

         /\* fix next block to point to the new fragment on the free list \*/

//调整新块后面的空闲链表

         NEXT\_HDR (pNextHdr)->pPrevHdr = pNextHdr;

         }

    return (pNewHdr);

}

分析：在vxWorks的内存池中，所有的内存块，无论是空闲块还是非空闲块，均是通过其每一个块内部的第一个块头和最后一个块头将内存池中的所有块连接成一个整体；

就单个内存块而言，紧接着前面第一个块头后一个块头才代表着该内存块中处于收尾两个块头大小的那块内存；

并且这第二内存块头和空闲块头相比仅少了DL\_NODE成员域，用于链入空闲链表。

示意图如下：

typedef struct blockHdr            /\* BLOCK\_HDR \*/

{

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 \*/

} BLOCK\_HDR;

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;

正因为如此，在第二个内存块后面只需要额外包含sizeof(DN\_NODE)大小的空间，两个类型就可以互相装换。

换句话说内存块除去收尾块头，只要剩下的内存可以存放一个空闲头的大小，这个内存块就

可以参与管理。

这也是为什么memSysPartition必须强调最小内存必须为一个空闲块头的大小。

**5.2.3.3 内存块的释放free()**

内存释放时，根据块头中的信息判断相邻的内存块是否空闲，如果相邻内存块空闲,将进行内存块合并，并修改空闲块长度；否则就把新释放的内存插入到空闲链表中。这里，空闲链表采用的是双链表的数据结构，它将紧临块头的8个字节(2个指针长度)用来存放双链表的指针，所有由malloc()分配的内存必须显式调用free()进行释放，以避免内存溢出，如图5.10。

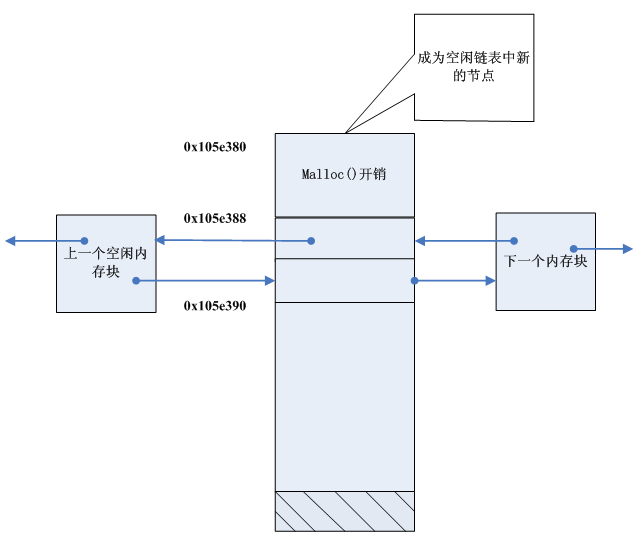


图5.10 内存释放

free()将malloc()分配的内存块释放到内存分区memSysPartition的空闲块链表中，其代码如下：

void free (void \*ptr )

{

    (void) memPartFree (&memSysPartition, (char \*) ptr);

}

free()是memPartFree()的封装。

如果pPrevHdr指向的前一块也为空闲块，则只需扩大前一块的内存空间大小即可；否则，把空闲块链入分区的freeList链表。空闲块链入的时候，还要判断能否与后一块内存空间合并（当前块的头部地址加上当前块的大小即为下一块的头部地址）。

内存释放的关键函数是memPartFree，设内存分区为partId，释放的内存地址为pBlock，分析如下：

（1）通过BLOCK\_TO\_HDR宏，获得内存头pHdr。

（2）获得内存块大小pHdr->nWords。

（3）如果前一块空闲PREV\_HDR (pHdr)->free为TRUE，则扩大前一块即可，扩大后的前一块为当前块。否则将该块作为独立空闲块链入分区空闲链表partId->freeList。

（4）检查后一块是否空闲，若是则将后一块并入到当前块上。

（5）更新后一块的pPrevHdr指针指向当前块。

代码如下：

STATUS memPartFree( PART\_ID partId, char \*pBlock )

{

    FAST BLOCK\_HDR \*pHdr;

    FAST unsigned   nWords;

    FAST BLOCK\_HDR \*pNextHdr;

    if (OBJ\_VERIFY (partId, memPartClassId) != OK)

         return (ERROR);

    if (pBlock == NULL)

                   return (OK);                                  /\* ANSI C compatibility \*/

    pHdr   = BLOCK\_TO\_HDR (pBlock);

    /\* 获取memSysPartition的信号量 \*/

    semTake (&partId->sem, WAIT\_FOREVER);

    /\* optional check for validity of block \*/

    if ((partId->options & MEM\_BLOCK\_CHECK) &&

        !memPartBlockIsValid (partId, pHdr, FALSE))

         {

         semGive (&partId->sem);                           /\* release mutual exclusion \*/

         if (memPartBlockErrorRtn != NULL)

             (\* memPartBlockErrorRtn) (partId, pBlock, "memPartFree");

         if (partId->options & MEM\_BLOCK\_ERROR\_SUSPEND\_FLAG)

             {

             if ((taskIdCurrent->options & VX\_UNBREAKABLE) == 0)

                   taskSuspend (0);

             }

         errnoSet (S\_memLib\_BLOCK\_ERROR);

         return (ERROR);

         }

    nWords = pHdr->nWords;

//检查该内存块前面的内存块是否空闲，如果空闲将该内存块合并到前面的空闲内存块中

    if (PREV\_HDR (pHdr)->free)

         {

         pHdr->free = FALSE;                          /\* this isn't a free block \*/

         pHdr = PREV\_HDR (pHdr);                          /\* coalesce with prev block \*/

         pHdr->nWords += nWords;

         }

    else

         {//否则单独作为空闲块，掺入memSysPartition分区的空闲块链表

         pHdr->free = TRUE;                   /\* add new free block \*/

         dllInsert (&partId->freeList, (DL\_NODE \*) NULL, HDR\_TO\_NODE (pHdr));

         }

//检查与其相邻的后面的内存块释放空闲，如果空闲将其后面的内存块从空闲队列中

//删除，并并入当前空闲块。

    pNextHdr = NEXT\_HDR (pHdr);

    if (pNextHdr->free)

         {

         pHdr->nWords += pNextHdr->nWords;   /\* coalesce with next \*/

         dllRemove (&partId->freeList, HDR\_TO\_NODE (pNextHdr));

         }

    /\* fix up prev info of whatever block is now next \*/

    NEXT\_HDR (pHdr)->pPrevHdr = pHdr;

    /\* adjust allocation stats \*/

    partId->curBlocksAllocated--;

    partId->curWordsAllocated -= nWords;

    semGive (&partId->sem);

    return (OK);

}

**5.3 VxWorks与动态内存相关的API**

表5.2 VxWorks5.5动态内存管理相关API



VxWorks 5.5允许用户建立并管理自己的内存分区（Memory Partition）,并提供相应的函数，如下表5.3所示。

表5.3 创建内存分区函数



每个分区又各自的分区ID。ID实际是指向分区内存管理数据结构mem\_part的指针，定义在memLib.h中。

typedef struct mem\_part \*PART\_ID;

**5.4 VxWorks内存分配机制总结**

在可用内存块上建立空闲块，并交给内存分区进行管理。内存分区记录了空闲块链表和一些统计信息。申请时从分区的空闲链表中找到合适的空闲块，并从该空闲块上分割出一块交给用户使用，当剩余空闲块太小时，整块交给用户使用。释放时，判断能否进行前向、后向合并，若可以则直接合并，否则作为一个独立块链入空闲链表中。

用户申请到的内存块，在它的前面隐藏了该块的基本信息（BLOCK\_HDR，DL\_NODE等信息），内存释放就是根据这个隐藏信息进行的。

内存块之间建立了两种联系：pPrevHdr元素存放了物理上连续的前一块内存的地址，用于空闲内存块的合并；链表节点node用于将空闲内存块链入到分区的freeList链表中，用于空闲块的管理。空闲内存块合并机制保证了任意两个空闲内存块在物理上不是连续的，从而避免了内存碎片。

内存分配采用首次适应（first fit）算法，即每次分配时，都是分配空闲链表中第一个合适块。设申请nWords字节的内存大小，要求alignment字节对齐，那么合适的要求是内存块的可用空间大于nWords + alignment / 2或者内存块可用空间等于nWords且alignment字节对齐。

这里需要注意的是当初始任务终结时，为初始化任务预留的内存10000字节，是作为一个独立的空闲块，加入到分区freeList链表中，为啥vxWorks没有这样做？这是因为在为初始任务tTaskRoot分配栈空间时，分区内存管理机制还没有建立，即malloc()服务例程还不能实现，只能手动进行分配，当初始任务被删除时，为其手动分配的任务只能作为一个独立的空闲块，加入到分区freeList链表中，而不能和其它内存开合并。

至此，本篇介绍的VxWorks内存管理机制就告一段落了，严格意义上本篇介绍的内存管理模块，比如malloc()/free()已经超过了Wind内存管理的范畴，但是为了描述的完整性，稍微增加了一点点O(∩\_∩)O~。

大家在阅读的过程，如果发现任何的问题，都可以直接给我留言，或者给我发邮件cwsun@mail.ustc.edu.cn，我会做进一步的修正，使其更为完善，如果需要注册为本网站的会员，请给我发邮件获取邀请码的邮件。

待续.......