嵌入式实时操作系统VxWorks的内存管理方案

万逸珠，戚文芽

（信息工程大学 信息工程学院，河南 郑州 450002）

摘要：文章介绍了嵌入式实时操作系统VxWorks的内存结构及其提供的内存管理接口，讨论了它的内存分配和内存保护策略，并以USB接口模块为例说明了一种一次分配多次使用的内存分配方案。

关键词：嵌入式实时操作系统；VxWorks；内存管理

中图分类号：TP316　　文献标识码：A　　文章编号：1671－0673(2004)04－0054－03

Memory Management of the Embedded Real Time Operation System VxWorks

WAN Yi-zhu, QI Wen-ya

(Institute of Information Engineering, Information Engineering University, Zhengzhou, 450002, China)

Abstract: This paper introduces the memory structure and the memory management interface of the Embedded Real Time OS VxWorks, and discusses the memory allocation and memory protection strategy in VxWorks. It also explains a method for decreasing memory fragment with an example of the USB Driver module.

Key words: Embedded Real Time OS; VxWorks; Memory Management

1 引言

由于嵌入式系统的实时性、灵活性和稳定性要求，它们对内存的管理也有着特殊的策略。VxWorks是一种高性能的嵌入式实时OS，其简捷实用的内存分配方案以及较为完善的内存保护手段值得我们探讨。

2 VxWorks的内存结构

VxWorks的内存结构如图1所示，在宏LOCAL_MEM_LOCAL_ADRS为嵌入式系统内存的起点，一般为0，宏RAM_LOW_ADRS是VxWorks系统镜象的起点而FREE_RAM_ADRS为系统镜象的终点。多数BSP都定义sysPhysMemTop()为物理RAM的结束地址，而sysMemTop()为系统内存池的结束地址，两者之差则为可配置的常数USER_RESERVED_MEM。这些宏都在板级支持包(BSP)中定义。

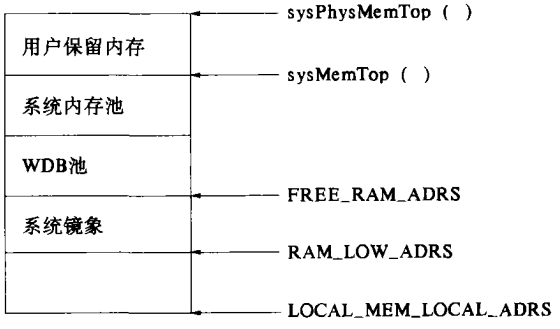

图1 VxWorks的内存结构

3 VxWorks的内存分配

3.1 系统内存分配方式

VxWorks的内存分配采用简单的先匹配先分配(First-Fit)算法，通过调用malloc从系统分区中分

收稿日期：2003－12－08

作者简介：万逸珠(1974－)，女，四川绵阳人，信息工程大学硕士研究生，主要研究方向为宽带网络嵌入式系统。

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

配, 通过 free 释放的内存将被聚合以形成更大的空闲块。分配时可以要求一定的对齐格式, 不同的 CPU 架构有不同的对齐要求。为了优化性能, malloc 返回的指针是经过对齐的, 为此的开销随架构不同而不同。例如, 68k 为 4 字节对齐, 开销 8 字节; SPARC 为 8 字节对齐, 开销 12 字节; MIPS 为 16 字节对齐, 开销 12 字节; I960 为 16 字节对齐, 开销 16 字节。

3.2　内存分配接口

VxWorks 的内存管理函数分为 2 个库, 紧凑的内存分区管理器(memPartLib)和完整的内存分区管理器(memLib)。memPartLib 提供的接口函数用于从内存分区中分配内存块。该库包含两类程序, 一类是用于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 程序提供与内存分区的接口。MemLib 是 memPartLib 的扩展, 进一步提供内存管理功能, 包括错误处理, 按对齐分配和 ANSI 分配函数。系统内存分区(其 ID 为 memSysPartId 是一个全局变量)在内核初始化时由 usrRoot 调用 memInit 创建。其开始地址为 RAM 中紧接着 VxWorks 的 BSS 段之后, 大小为所有空闲内存。

3.3　一次分配多次使用的内存分配方案

malloc()和 free()可在任意的时间段中, 为任务分配任意大小的内存块, 随着内存块的使用和释放, 在整个系统内存分区中, 分配给堆栈的存储区将混杂着许多正在使用或已经释放的存储块, 而未被使用的任何小块内存区将变得无法使用, 即产生了内存碎片。在嵌入式系统设计中, 经常有一些需要频繁申请释放内存的场合, 为了减少碎片, 提高效率, 常采取一次分配多次使用的内存分配方案。

下面以 USB 接口驱动模块为例来说明这一方案的具体实施细节。VxWorks 对 USB 总线数据传输时的这种内存分配方案主要由库 ossLib 提供的函数 ossPartMalloc()来实现。

```
void * ossPartMalloc ( UINT32 numBytes)
{
    …
    if (ossPartInitFlag= = TRUE)/ * 首次调用本函数 */
        {
        / * 申请 USB 分区所需的缓冲区 */
        pUsbMemSpace= (char *) cacheDmaMalloc (usbMemPartSize);
        …
        / * 创建 USB 分区 */
        usbMemPartId= memPartCreate (pUsbMemSpace, usbMemPartSize);
        …
        ossPartInitFlag= FALSE;
    }
    / * 从 USB 分区中分配一片内存 */
    if (( pBfr = memPartAlignedAlloc (usbMemPartId,
            ROUND_UP(numBytes, _CACHE_ALIGN_SIZE), _CACHE_ALIGN_SIZE)) = = NULL)
    {  / * 如果分配不成功则扩大 USB 分区 */
        growSize= 0x2000;
        if (numBytes> 0x1000)
        growSize += numBytes;
        pMoreMemory = cacheDmaMalloc (growSize);
        …
        if (memPartAddToPool (usbMemPartId, pMoreMemory, growSize)= = ERROR)
        {      …      }
        usbMemPartSize+= growSize;
        / * 扩大 USB 分区后再次从中分配所需缓冲区 */
        pBfr= memPartAlignedAlloc (usbMemPartId,
            ROUND_UP(numBytes, _CACHE_ALIGN_SIZE), _CACHE_ALIGN_SIZE);
    }
    …
    / * 分配成功返回指向所需缓冲区的指针 */
    return pBfr;
}
```

在首次被调用时, 它调用 cacheDmaMalloc(), 分配一片 64k 的缓冲区, 然后调用 memPartCreate(), 把它建立为 USB 专用分区, 保存下它的 ID, 以后再被调用时, 它首先调用 memPartAlignedAlloc(), 根据所存 ID 尝试从这片 USB 专用分区中分配内存, 如果不能分配成功, 则重新从系统内存分区中再分配一片缓冲区, 并调用 memPartAddToPool(), 把这片新分配的缓冲区添加到已有的 USB 分区中去。USB 驱动程序在需要分配临时缓冲区时, 总是调用 ossPartMalloc()来分配, 在释放时则调用 ossPartFree()来把不用的缓冲区释放回 USB 分区中。

4　VxWorks 的内存保护

VxWorks 是一个基于任务(task)的实时操作系统(RTOS), 在 VxWorks 下, 所有的应用程序都运行在内核模式下。在内核模式下的所有操作都可以以实地址的方式访问所有的物理内存。毫无疑问,

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved.　http://www.cnki.net

运行在内核模式下的应用程序有更完备的功能和灵活性,但需要付出的代价是,内存的任何区域,例如说应用程序驻留区域、数据区,甚至内核自身所处的区域都面临着被改写的危险。

目前许多处理器都提供了内存管理单元(MMU),这个硬件组件所负责的功能是将内存的逻辑地址转换为实际的地址。操作系统通常利用此内存管理功能,达成虚拟内存及主存储器的保护功能。VxWorks也提供了对MMU的基本支持,除此之外,WINDRIVER公司还提供了虚拟内存接口(VxVMI)用于增强内存保护。

4.1 保护私有数据

如果不采取任何措施的话,一个任务的所有数据都将被存放在一个全局区域内,任何其它任务都可以方便地读取和改写它。这种错误,不易调试。因此,任务经常需要保护自身的私有数据。生成能被保护的私有内存区域的方法是使用taskSwitchHookAdd()在所有任务切换时动态地改变"私有内存"页面的状态。先调用宏VM_STATE_SET(),它将调用vmBaseStateSet()以寻求基本MMU支持,然后在进入任务时,把内存页面的属性设成为VM_STATE_VALID,从而使该任务生成它自已的"私有内存";而在离开任务时,把内存页面的属性设成为VM_STATE_VALID_NOT,从而使别的任务不能访问这片内存区域。这样,任务就创建了它的私有内存,如果别的任务企图访问该区域,调试过程中就会产生一个"转换旁视缓冲"(TLB)错误。

4.2 保护共享数据

任务之间也经常需要以一种受保护的模式共享数据,例如,在一个生产—消费模型里,生产任务产生数据并把它们放在一个专用的数据结构里,然后消费任务会来读取数据,当存在多个任务可能同时读写数据结构的情况时,我们就需要把数据保护起来,以免产生误操作或被非法用户改写,从而增加系统的可靠性。生成被保护的共享数据的方法是,首先,申请一片共享内存并创建一个控制它的信号灯,然后,用宏VM_STATE_SET()和VM_STATE_WRITABLE_NOT把这片内存区域设成只读属性,最后,创建一个专门改写这片共享内存的例程,在获取到信号量时,用宏VM_STATE_SET()和VM_STATE_WRITABLE把这共享内存区域属性设成可写,任何只对共享内存进行读操作的任务都可以直接访问共享内存,但对共享内存的写操作必须通过上述例程来完成,任何不通过该例程来对共享内存的访问都会在调试过程中产生一个TLB错误。

4.3 检测缓冲越界

内存保护也能用于检测代码编写中的缓冲越界错误。通过在一个缓冲区的两端申请"卫兵"页面,能使系统检测到这类错误。生成"卫兵"页面的方法是,首先,申请所要使用的缓冲区,该缓冲区应该是按页对齐的;然后,把申请到的缓冲区增大额外的相当于两页的大小,实际使用的缓冲区位于中间,其高端和低端各一页用来做"卫兵"页面;最后,把这两页"卫兵"页面用宏VM_STATE_SET和VM_STATE_VALID_NOT设成不可访问页。这样,任何对该缓冲两端的页的访问都会被在调试过程中产生一个TLB错误。

4.4 保护代码区和中断向量表

如果安装了VMI,通过简单的两条宏指令INCLUDE_PROTECT_TEXT和INCLUDE_PROTECT_VBC_TABLE就可以分别实现代码区保护和中断向量表保护功能。

5 结束语

通过灵活的内存分配方案和有效的内存保护机制,VxWorks不仅提供了较高的实时性和稳定性保障,还为嵌入式系统的软件的开发降低了难度,缩短了开发周期。实际上这种内存管理的策略也可以在我们以后的开发过程中推广和运用。

参考文献:

[1] 孔祥营,柏桂枝. 嵌入式实时操作系统VxWorks及其开发环境Tornado[M]. 北京:中国电力出版社,2002.

[2] Wind River Systems, Inc. Tornado BSP Training Workshop. [EB/OL]. http://www.windriver.com/training, 1998.

[3] Wind River Systems, Inc. VxWorks Real-Time Kernel[EB/OL]. http://www.windriver.com, 1998.

[4] Wind River Systems, Inc. VxWorks Programmer's Guide 5.4 [EB/OL]. http://www.windriver.com, 2001.

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net