学科门类：工 学　　单位代码：10287
中图分类号：TP316

硕 士 学 位 论 文

动态存储管理机制的改进及实现

硕士生姓名： 曹全欣
一级学科： 计算机科学与技术
学科、专业： 计算机应用
研究方向： 系统软件
指导教师： 顾宝根 教授

南京航空航天大学
二〇〇三年一月十日

摘 要

动态存储管理是计算机系统必不可少的资源管理手段之一。现有的动态存储分配算法存在着效率不高、时间开销较大等缺点。

本文在自主开发操作系统这个教学课题的基础上，深入研究了操作系统内存分配及回收机制的基本理论，结构，工作过程及有关的概念，算法，技术等等。详细阐述了对边界标识法，伙伴系统算法改进后的具体实现；介绍了一个用多级位示图目录来实现存储资源动态分配的算法，给出了多级位示图目录的结构和相应的算法，并详细介绍了一个二级位示图目录的具体实现过程。同时将这三种存储分配算法进行了性能对比与分析，并得出结论。随后，文章还描述了内核缓冲区特有的内存管理模式，详细阐述了其具体实现过程，并进行了性能分析。最后文章针对具体的实现提出了进一步的改进设想，并确定了今后的发展方向。

关键词： 动态存储管理，边界标识，伙伴系统，多级位示图目录，管理开销

Abstract

Dynamic memory management is necessary for memory resource management in computer system. The existing dynamic memory management methods are inefficient, and have high cost of time.

Based on the development of our own operating system for teaching object, this thesis describes the basic theories, architectures, procedures, concepts, algorithms and technologies about the memory management in an operating system.

The thesis recounts the realization of the boundary tag method and the buddy system allocator method, which had been improved. A dynamic memory management method using multilevel bitmap catalogue structure is introduced in the paper. A detailed description of the structure of multilevel bitmap catalogue as well as algorithms using multilevel bitmap catalogue is given. The thesis also describes the implement of a 2-level bitmap catalogue method in detail. Then it compares the performance of the three different methods, and draws a conclusion by the analysis. Subsequently, the thesis describes the particular memory management mode of the kernel buffer, depicts the implement of the mode, and analyzes its performance. Finally, the paper proposes the assumption of the betterment and confirms the way of future works.

Key Words: Dynamic Memory Management, Boundary Tag, Buddy System, Multilevel Bitmap Catalogue, Managing Overhead

目 录

图清单

表清单

第一章 引言

1.1 课题研究的原因及意义

当前，随着计算机科技水平的进步和提高，操作系统也飞速发展起来。尤其是从出现了 Linux 操作系统以后，更是极大地促进了操作系统的研究与发展，在相当广泛的范围内掀起了研究操作系统的热潮。现今我国操作系统的研究水平有了相当大的提高，出现了一批操作系统核心，但是应用还不够广泛、普及。目前国内采用的操作系统绝大部分是由外商提供的，如 Windows 系列，Unix 和 LINUX 等等。计算机应用软件是建立在操作系统的基础上的，强烈依赖于操作系统，没有自主开发的操作系统，或者是操作系统性能低下，无疑都会严重地制约我国软件产业的发展。因此，开发自主的操作系统核心，提高自主操作系统的开发水平，跟踪世界先进的操作系统的发展趋势，对于国家安全以及民族产业发展都有着极其重要的意义。为了提高操作系统课程的教学水平，培养出更多的操作系统方面的人才，南航信息科学与技术学院系设立了《基于教学平台的操作系统开发研制》这个教改项目，其目标就是开发一个基于教学目的的通用操作系统原型，这个系统覆盖了现代操作系统的核心技术，通过这个系统的开发，可以使学生对于《操作系统》这门课程有一个较感性的认识，同时也有了比较具体的开发模式可供参考，同时，为进一步开发出商用的操作系统打下一个良好的基础。

内存管理是计算机系统的重要组成部分，是操作系统的核心之一。近年来，随着计算机技术的发展，系统软件和应用软件在种类、功能及其所需存储空间等方面，都在急剧的膨胀，虽然存储器的容量也一直在不断地扩大，但仍不能满足现代软件发展的需要，因此，内存仍然是一种宝贵且紧俏的资源。如何对它们实施有效的管理，不仅直接影响到内存的利用率，而且还对系统性能有重大影响。

另一方面，随着互联网的快速普及以及电信事业的发展，操作系统的发展也呈现了两种趋势，一种是随着适应多种异构平台的操作系统的出现，对于性能、可用性、可靠性、安全性以及数据管理能力提出了更高的要求；另一种是随着各种信息电器的发展，实时嵌入式操作系统的蓬勃兴起，对于实时性，效率，系统规模提出了更高的要求。无论是何种发展方向，都离不开操作系统内存管理机制的进步和提高。

在《基于教学平台的操作系统开发研制》的项目实施中，我主要针对内存管理这部分进行了深入的研究和探讨，在深入学习理论和阅读了大量现代操作系统核心实现代码的基础上，对 i386 体系结构上的内存管理机制进行了一些尝试和改进。内存管理是一个相当复杂的综合体，按功能的差异，可大体上分为：内存的申请与释

放、物理内存的管理、虚存的组织、以及页面交换等部分，本文主要是关于内存的申请与释放机制及其管理这些方面的探讨、研究及其实现。

1.2 本文的主要工作

本文的内容组织如下：

第一章介绍了课题研究的原因和意义以及本文的重要内容。

第二章讲述操作系统及内存管理的基本理论及其基本设施的构建。

第三章讨论了边界标示法内存分配机制的改进，描述了此种机制下的内存管理框架，详细阐述了具体的边界标识法的实现细节。

第四章讲述了伙伴系统算法的理论，描绘了在伙伴系统下内存机制的框架，详细阐述了其具体的实现细节。

第五章介绍了多级位示图目算录法的内存管理机制，并详细描述了二级位示图目录法的具体的实现过程。同时对上述三种机制进行了比较分析。

第六章讲述了在对 slab 算法进行改进的基础上，对内核缓冲区的动态管理机制的具体实现。

第七章阐述了本文工作得出的结论，并讲述了今后进一步的改进构想。

第二章 内存管理基础设施的构建

内存管理是与 CPU 以及计算机系统结构密切相关的，本文的主要研究工作是基于 i386 体系结构的计算机上展开的，所以有必要介绍一下 i386 体系结构的特点和相关功能。

2.1 内存管理的硬件基础

2.1.1 分段机制的硬件基础

我们知道，i386 体系结构的 CPU 提供了保护机制，在访存指令给出的逻辑地址转化成物理地址的过程中，在某个环节上对访问权限进行比较，以防止不具备特权的用户通过一些特殊手段（例如修改段寄存器的内容，修改段描述结构的内容等）得以非法访问其他进程的空间或系统空间。i386 划分了四个特权级可供使用，其中 0 级最高，即内核态，3 级最低，是用户态[6][17]，大多数操作系统都只使用了这两个级别，这里的实现也是如此。

Intel 的 i386 体系结构中设置了两个寄存器：一个是全局段描述符表寄存器 GDTR（Global Descriptor Table Register），另一个是局部段描述表寄存器 LDTR（Local Descriptor Table Register），分别用来指向存储在内存中的一个段描述符结构数组，这两个数组分别叫全局描述符表（GDT）和局部描述符表（LDT）[25]。访问这两个寄存器的指令都设计成特权指令，也即只能在内核态中运行。比如装入和存储 GDTR 和 LDTR 的指令 LGDT、LLDT 和 SGDT、SLDT 等都是特权指令。正是由于这些指令都只能在系统内核态下才能使用，才使得用户程序（运行在用户状态下）不但不能改变 GDTR 和 LDTR 的内容，并且由于无法确定其段描述符表在内存中的具体位置，又不能访问其所在的空间（只有在内核方式下才能访问），从而无法通过修改段描述项来打破系统的保护机制。

段寄存器的内容意义不再与 8086 下的相同，成为了段选择子，见图 1。

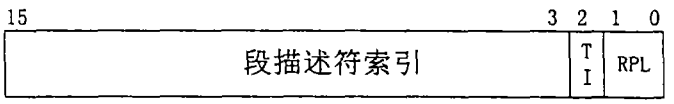

图 1　段选择子示意图

其中第 3~15 位这高 13 位是描述符索引，标识描述符在描述符表中的序号。TI 是描述符表指示位，TI=0 指示从 GDT 中读取描述符，TI=1 从 LDT 中读取描述符。RPL 是请求特权级，用于特权检查，只有当 RPL≤段描述符中的 DPL 才允许执行。

每个段描述符表项的长度是 8 个字节，其结构的各个意义字段见图 2。

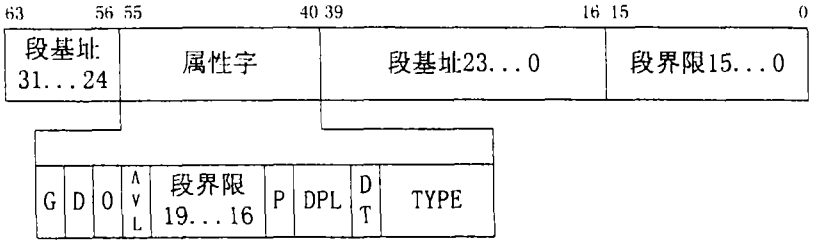

图 2　段描述符示意图

其中段界限 0~15 和段界限 16~19 合并起来表示出一个段的长度；段基址 0~23 和段基址 24~31 合并起来表示一个段的基址；G 位是段界限粒度位，G=1 表示界限粒度为 4K 字节，G=0 表示粒度为字节；D 位表示默认下指令及操作数是 32 位（D=1）还是 16 位（D=0）；P=1 表示段存在，描述符在地址转换时有效；DPL 规定了描述段的特权级；DT=1 表示该描述符时系统段描述符;TYPE 说明存储段的具体特性。

在 i386 体系分段机制作用下，一条访存指令通过如下步骤获得实际数据总线地址：

1. 根据指令的性质来确定应该使用哪个段寄存器，例如：转移指令中的地址在代码段中，取数指令的地址在数据段中。
2. 根据段寄存器的内容作为基址来找到相应的段描述符。
3. 从段描述符中获得段的基地址。
4. 将指令中发出的地址作为偏移，与段描述符中规定的段的长度比较，确定是否越界。
5. 根据指令的性质和段描述符中的访问权限来确定是否越权。
6. 将指令中发出的地址作为偏移，与基地址相加来获得实际的物理地址。

这样的访问机制给 CPU 提供保护模式奠定了基础，但是，由于段描述符在内存中，所以一条访存指令实际上要两次访问内存才能获得实际内容，（这里讨论的是只有分段机制的情况，如果同时有分页机制，那么实际上访问内存的次数还要更多），这样就降低了 CPU 的工作效率，为了解决这个问题，Intel 在实现上在 CPU 中添加了一组影子结构，这个影子结构只对 CPU 可见，对于程序员是不可见的，段描述符开始在内存中，但在实际使用时将其装入 CPU 的影子结构中，CPU 运行时则是使用影子结构中的内容。

如果将每个段寄存器都指向同一个描述表项，并且在这个表项中将基地址设置为 0，将段长度设成最大，就形成了一个从 0 开始到整个 32 位地址空间的段。并且由于基地址为 0，物理地址就和逻辑地址相同了，CPU 放到地址总线上去的地址就是指令中给出的地址，这种方式称为“平坦地址方式”。平坦地址方式的使用并不意味着绕过了段描述符表、段寄存器这一套段式管理内存的机制，而只是段式内存管理的一种特例。在本文实现的系统中就采用了这种地址方式。

段式存储管理按逻辑划分段，方便了段之间的共享，但其灵活性和效率都比较差，一方面段是可变长度的，这就给盘区交换操作带来了不便；另一方面，如果为了增加灵活性而将一个进程划分成很多小段时，就势必要求在程序中频繁地改变段寄存器地内容；同时，段描述表容纳的最大 8192 个段描述表项并不一定能够保证够用。

2.1.2 分页机制的硬件基础

在 i386 中的保护模式的实现是与段式存储管理密不可分的，这决定了页式管理只能建立在段式管理的基础上。页式存管的作用是在由段式存管所映射而成的地址上再加一层地址映射。此时由段式映射而成的地址不再是物理地址了，而称之为线性地址。因此，段式存管先将逻辑地址映射成线性地址，然后再由页式存管将线性地址映射成物理地址；如果不用页式存管时，就将线性地址直接用作物理地址。分页机制是将用户程序的地址空间划分成若干个固定大小的区域，称其为页。同时，实际物理内存也被划分成若干物理块，块与页的大小是相等的，采用此种方式的好处在于可以将用户程序的一页放到物理内存中的一块上，从而实现了内存的离散分配，有利于内存的共享以及减少碎片。以 i386 结构计算机为例，将线性地址解释成三部分，见图 3。

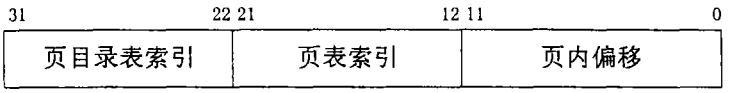

图 3　线性地址解释示意图

将物理地址解释成两部分，见图 4。

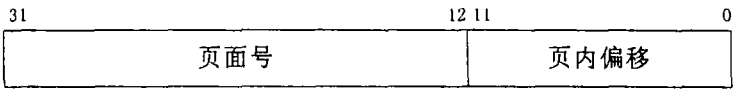

图 4　物理地址解释示意图

CPU 中有个 CR3 寄存器用来指示当前页目录表的位置，由于页目录表必然是与 4K 对齐的，所以，CR3 中的低 12 位为 0，见图 5。

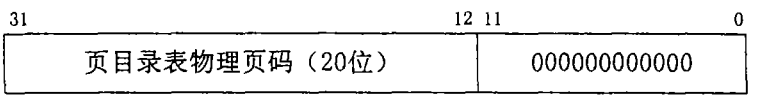

图 5　CR3 寄存器示意图

由此可见，在页面目录中共有 $2^{10}=1024$ 个目录项，每个目录项指向一个页面表，而在每个页面表中又共有 1024 个页面描述项。在 i386CPU 中有 MMU（内存管理单元）这个部件，MMU 的运作是硬件动作，负责将逻辑地址划分为页目录索引、页表索引和页内的偏移量三部分，进行映射。映射的具体过程见图 6。

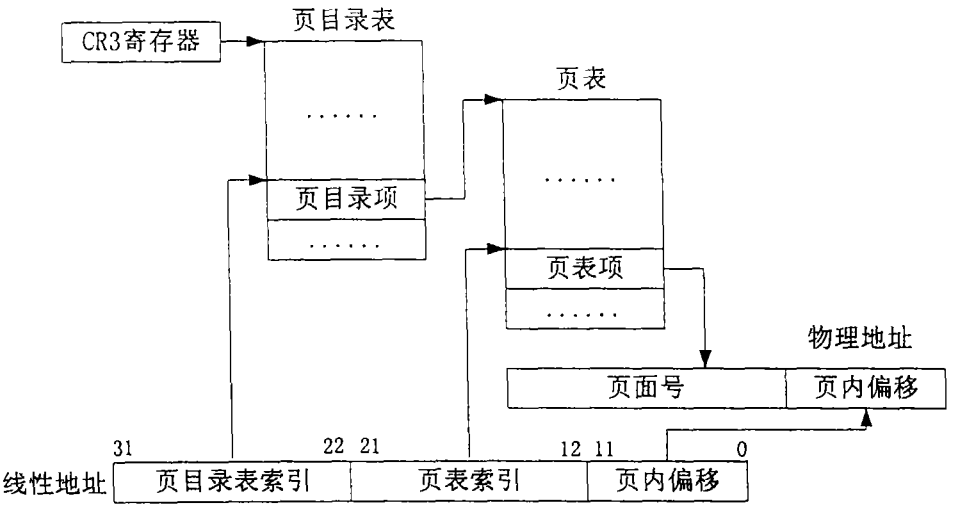

图 6　地址映射示意图

映射的步骤说明如下：

- 首先从 CR3 中取得页目录表的基地址，其低 12 位必为 0；
- 以线性地址中的高 22~31 位为下标，在页目录中找到相应页目录表项；
- 以页目录表项中的高 20 位，再在低 12 位补 0，获得页面表的基地址；
- 以线性地址中的页面索引为下标，在页面表中取得相应的页表项。
- 以页表项的高 20 位，拼接线性地址的页内偏移量这低 12 位，获得物理地址；

页面表和页面的地址都总是在 4K 字节的边界上，所以在目录项和页表项中都只要有 20 位用于指针就可以了，其他的 12 位则用于一些控制信息。见图 7。

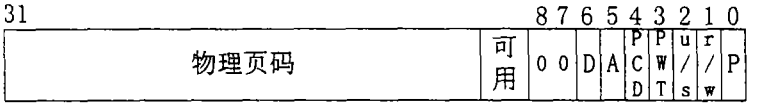

图 7　页表项示意图

其中：

- 第 0 位 P 表示目录项或是页表项是否有效，也即是否存在在内存中，所以称为存在位，当 P＝0 时表示此表项无效；当 P＝1 时表示有效。
- 第 1 位 RW 表示页面读写属性，当 RW＝0 时表示页为只读和可执行；当 RW＝1 时表示可读、可写、可执行。这主要是用于页面的保护性检查过程中，要注意的是，这种检查只是在 CPU 处于用户态下才进行，当 CPU 处于核心态时，这种检查将忽略。
- 第 2 位 U/S 表示页面时系统级页面还是用户级页面，当此位为 0 时表示页面是系统级页面；当此位为 1 时表示是用户页面。这也是用在页面保护过程中的，系统级的页面只能被处于核心态下的程序访问，而用户级的页面可以由任何特权级的程序访问。

- 第 3 位 PWT 和第 4 位 PCD 是用于缓冲 Cache 的标记位，这里将它们设置为 0，表示启用高速缓存，对写操作总是采用写回策略。
- 第 5 位 ACCESS 表示是否已经访问过，在页面刚被调入内存中时，ACCESS＝0，在这个页面被访问过了以后，CPU 会自动将这位设置成 1。这一位主要是用于页面换入换出的处理过程中。
- 第 6 位 DIRTY 表示该页面写访问位。当页面被修改后，CPU 会将这位设置为 1，使得操作系统在调出此页面时可以判断是放弃页面或是将其写回到磁盘中去。

虽然页式存管是建立在段式存管的基础上的，但是如果启动了页式存管，所有的线性地址都要经过页式映射，包括类似 GDTR 和 LDTR 中给出的段描述符表的起始地址也不例外。

页式管理的好处在于，页面都是固定大小的，便于管理；便于磁盘交换操作；便于程序间共享。页式管理是当前操作系统的主流趋势。

2.2 基本的数据结构及操作

2.2.1 进入保护模式的方法

要充分利用 i386 体系结构的优越性，就要进入 CPU 的保护模式，使控制寄存器 CR0 的 PE 位为 1。CR0 的各个位字段如下图 8 所示：

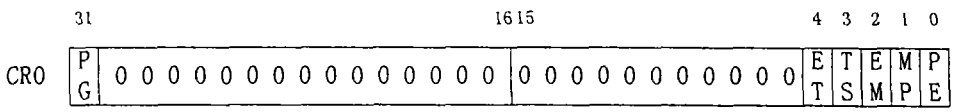

图 8　CR0 寄存器示意图

与存储管理相关的是 PG 位和 PE 位，根据 PG 位和 PE 位的不同组合情况，CPU 有以下几种寻址方式：

PE＝0，PG＝0 实方式，系统加电后最初的方式

PE＝1，PG＝0 保护模式，支持分段但不支持分页

PE＝1，PG＝1 保护模式，支持分段也支持分页

PE＝0，PG＝1 虚拟 8086 方式（V86）不支持分段，仅支持分页

在从开机的实方式切换到保护方式之前，要建立合适的全局描述符表，并使 GDTR 指向该 GDT，这是因为切换时至少要把代码段的选择子装载到 CS 寄存器中。装载 GDTR 的语句如下：

```
asm lgdt Gdt_table
```

通常在进入保护模式时要关闭所有的中断。将 IDTR 的界限设置为 0，CPU 自动关闭所有中断，包括 NMI。

另外，为了访问 1M 以上的内存空间，应该让 8042 打开 A20 地址线，其指令如下：

```
asm {
        push ax
        in al,92h
        or al,2
        out 92h,al
        pop ax
        ret
}
```

由实方式切换到保护方式只要将 CR0 中的 PE 位置 1 即可，执行如下指令序列：

```
mov eax,cr0
or eax,1
mov cr0,eax
```

由于 BC3.1 连接器不能处理 32 位寄存器，所以直接在代码中加入操作码前缀 0x66 和地址前缀 0x67 后，指令变成如下：

```
asm{
        db 0x66,0x0F,0x20,0xC0      // MOV EAX,CR0
        db 0x66
        or AX,1                     // OR EAX,1
        db 0x66,0x0F,0x22,0xC0      //MOV CR0,EAX
        jmp Flush
      }
```

其中 JMP Flush 将保护方式下的代码段选择子装入 CS，同时刷新预取指令队列。接下来，就要对页映射表进行初始化，安排页表物理空间布局，然后启动分页机制：

```
asm{
        db 0x66,0x0F,0x20,0xC0              //MOV EAX,CR0
        db 0x66,0x0D,0x00,0x00,0x00,0x80 //OR EAX,0x80000000
        db 0x66,0x0F,0x22,0xC0              //MOV CR0,EAX
        JMP PageStart
}
```

在启用分页机制前，线性地址就是物理地址；启动分页机制后，线性地址要通过分页机制的转换，才能成为物理地址。为了保证顺利过渡，在启用分页机制之后的过渡阶段，仍要维持线性地址等同于物理地址，所以在建立页映射表时，必须使实现过渡的代码所在的线性空间页映射到具有相同地址的物理空间页上。所以在页表中设置了一个初始化页表 0，它的作用就是使线性地址空间映射到相同的物理地址空间。然后启动分页机制，到此时，系统已经进入了保护模式下的分页状态。

2.2.2 物理内存空间的抽象

为了使系统能够方便快捷地进行内存管理，仅仅依靠页目录，页表项等结构是远远不够的，还需要添加一些必要的管理元素，下面大致描述一下在此系统中所运用到的一些主要数据结构及其功能。

首先，要对整个系统的内存有一个抽象的描述体，这里采用了一个结构数组，数组中的每一个结构项都对应着系统的一个物理页面，整个数组就代表了系统中的全部物理页面。有了这个数组的定义，页表项的高 20 位地址就被赋予了两个意义：对于 MMU 硬件来说，在高 20 位地址的后面自动添加 12 位 0，就得到了物理页面的起始地址；对于内存中的这个数组来说，以这 20 位作为下标，就可以检索到数组中代表了这个物理页面的 page 数据结构项。

Page 数据结构中主要相关的数据项描述如下：

```
struct page {
        struct page * prev;
        struct page * next;
        unsigned long index;
        unsigned long count;
        struct manage_struct * man_zone;
        ……
};
```

其中：

- next，prev 是两个结构指针，通过它们把 page 数组项又连成一个双向循环链表，这样做的目的是为了将一些相关的页面链接在一起，从而便于操作；
- index 是该页面的索引号；
- count 表示当前使用这个页面的进程的数目；
- man_zone 指向一个内存分配回收管理结构 manage_struct，将在后文中详细讲述；

其次对于内存管理中要使用到的 CPU 内部寄存器以及各种表项也都定义了相关的数据结构，比如页表项，页目录项，GDT 表项等等，同时，为了加快这些数据结构的操作速度，大量的采用了宏定义的方式进行处理，由于内容较多较杂，这里就不再赘述了。

2.2.3 虚拟内存空间的抽象

上面的数据结构都是用于物理内存空间的管理的。由于进程运行在虚拟空间中，对于虚存空间也要进行管理，虚存空间管理不同于物理空间管理之处在于：物理空间有一个物理页面的总量，是和系统中实际配置的物理内存量相关的，可以随

着系统物理配置的不同而不同；而虚拟空间是以进程为基础的，创建了一个进程，就产生了一个新的虚存空间，每个进程都有其各自的虚存空间，且大小都为 4G（对于 i386 来说），彼此之间是独立的，不会相互干扰。

在虚拟空间的划分上，仿效 Linux 及 Windows2000 的作法[10][14]，将 4G（字长 32 位）的空间分成了两部分。将最高的 1G 字节（从虚地址 0xC0000000 到 0xFFFFFFFF）用于内核本身，称为“系统空间”。而将较低的 3G 字节（从虚地址 0x0 到 0xBFFFFFFF）用于各个进程的用户空间。这样每个进程可以使用的用户空间都是 3G 字节。当然，实际空间的大小受到物理存储器的限制。系统空间是由所有进程共享的。虽然系统空间占据了每个虚存空间中最高的 1G 字节，但是为了便于具体实现，系统空间在物理内存中却总是从最低的地址 0 开始的，这个映射由宏来实现。从虚存地址转化到物理地址：

#define get_phya（x） （（unsigned long）（x）－ 0xC0000000）

从物理地址转化到虚存地址：

#define get_vira（x） （（unsigned long）（x）+ 0xC0000000）

因为一个进程一般不会真的需要使用其全部的 3G 字节的虚存空间，同时，一个进程所需要使用的虚存空间的各个部分也不一定是连续的，很可能是由若干个离散的区域组成，所以，需要一个有效的数据结构来描述它，称其为“虚存控制项”，这个结构的描述如下：

```
struct vir_mem_struct {
        struct mm_struct * mem;
        unsigned long vm_start;
        unsigned long vm_end;
        struct vir_mem_struct * vm_next;
        unsigned long page_attr;
        unsigned long flag;
};
```

其中的各项解释如下：

- vm_start 这个 32 位地址，是该虚存控制项所描述的虚存空间的起始地址，vm_end 是该虚存控制项所描述的虚存空间的结束地址＋1，此块虚存空间的大小也就是：vm_end－vm_start；
- page_attr 和 flag 分别是标明了页面属性和段属性，由于程序中存在了各种各样具有不同属性的段，比如数据段，代码段等等，所以虚存空间的区域的划分也是带有属性的，此外，在同一个区域中的所有页面都必须具有相同的访问权限，这就是 page_attr 和 flag 项在这里的作用；
- 同一进程的所有区间按照其起始虚存地址的高低从低到高顺次排列，并通过 vm_next 将它们链接起来；

- mem 是一个指向 mm_struct 结构的指针。在整个虚存空间的最高层次上，还有一个数据结构来统揽大局，它就是 mm_struct 结构，在进程控制块中设定了一个指针指向这个 mm_struct 结构，这样每个进程就有其各自的进程虚拟空间了。

mm_struct 的主要相关数据项描述如下：

```
struct mem_struct {
        struct vir_mem_struct * virmem;
        struct vir_men_struct * virmemcache;
        pd_t * pt;
        unsigned long codestart, codeend;
        unsigned long datastart, dataend;
        unsigned long stackstart;
        ……
};
```

其中：

- virmem 即指向虚存控制项队列，它表示了这个进程所有的虚拟地址空间；
- virmemchche 是指向最近一次访问过的虚存空间的虚存控制项，根据局部性原理，刚访问过的数据、指令，很可能再次被访问，添加了这个指针，可以在一定的程度上避免再次查找链表的时间花销，提高系统访问虚存的效率；
- pt 是一个指向该进程页目录表的指针，就是通过它找到这个进程的页目录表，从而可以通过页面地址转换机制将虚拟空间和物理空间联系起来；
- codestart，codeend，datastart，dataend，stackstart 等分别表示了程序中的代码段、数据段和堆栈段的起始/结束地址，当然一个程序中还有其他不同类型的段，因为这并不是本课题的研究重点，这里就不作详细的介绍了。

在上述数据结构的作用下，虚拟地址和物理内存联系关系见图 9：

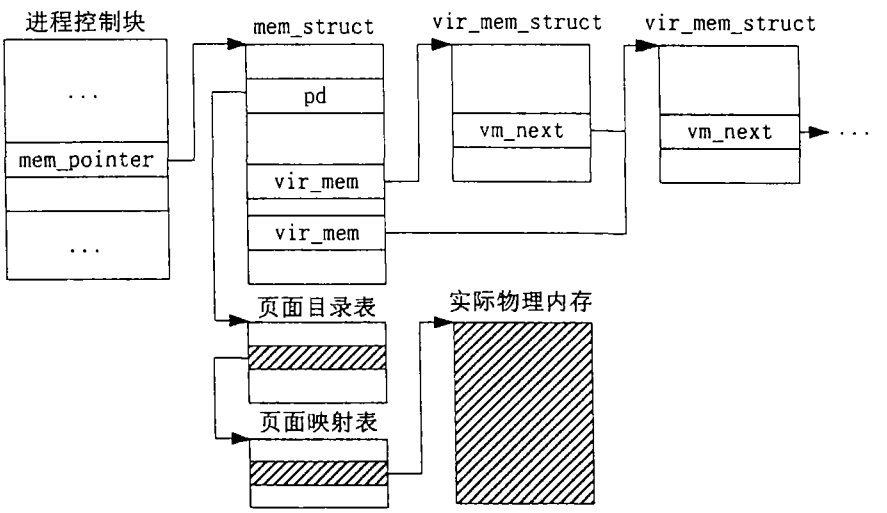

图 9　虚存管理数据结构关系图

mem_struct 结构及其指向的 vir_mem_struct 等结构仅仅是表明了应用程序对虚存空间的需求，更重要的一点在于即使虚存空间分配给了应用程序，并不代表了其页面已经映射到了物理页面空间。CPU 在使用页式存储管理机制，通过页目录表和页面表将线性地址转换为物理地址，并访问最终的物理内存单元这个过程中，如果受到了某种阻碍而无法访问到最终的物理内存单元，映射就失败了，这时 CPU 会产生一次“Page Fault”异常。阻碍一般有三种情况： a.该线性地址与物理地址的映射关系没有建立，也就是相应的页面目录项或者页面表项为空；b.对应的物理页面不在内存中，也就是页表项中的 P 位为 0；c.访问方式与页面的权限不符合，这是页式保护机制的功能。

当 CPU 产生“Page Fault”异常时，CPU 将导致映射失败的线性地址放到控制寄存器 CR2 中，同时，系统的中断/异常响应机制还要传过来另外两个参数。一个是当前 CPU 中各个寄存器内容的现场记录，另一个是错误代码。根据错误代码，系统可以得知是什么原因导致物理内存不能被正确的访问到，然后系统会进行相应的处理，如果页表项还没有建立就建立页表项形成映射关系；如果是越界访问，要根据访问的地段不同进行不同的处理，如增长堆栈段，非法访问报错等等；如果是页面不在物理内存中，就要启用页面交换机制进行页面调度。

第三章 改进的边界标识法的具体实现

本文中介绍的几种内存管理机制都是以 i386 的分页机制为基础的，也就是说都使用了页表的地址映射功能。

3.1 内存分配回收机制的基本要求

对于操作系统来说，存储管理是一个复杂但又重要的问题。动态存储管理的基本问题是：系统如何针对用户提出的请求分配内存？又如何回收那些用户不再使用而释放的内存，以供新的请求到来时再次分配？怎样合理而又有效地管理内存，使系统高效的使用有限的内存资源，这些问题都是非常值得研究的。

我们将已经分配给用户使用的内存区域称为“占用块”，未被分配的内存区域称为“空闲块”。系统经过一段时间的运行后，整个内存区会呈现出占用块和空闲块犬牙交错的状态，因此需要有控制结构控制内存空闲块的信息，一般称其为“可利用空间表”。可利用空间表一般有以下三种不同的应用场合：

- 系统运行期间所有用户请求分配的内存量大小相同，这种情况在实际中不多见，而且系统实现较简单，这里不做讨论。
- 系统运行期间所有用户请求分配的内存量有若干种大小规格，这种情况下，一般是按照规格建立起若干个可利用空间表，每个表中有相同规格的空闲块若干，在分配时按不同规格从不同的可利用空间表中获得内存块。这种情况在实际中也不多见，而且其实现也较简单，这里也不做讨论。
- 最常见的情况就是用户请求内存块的大小不固定，是随意的，本文是针对这种情形进行研究的。

由于系统中内存分配大小不固定，可利用空间表中的节点大小不同，所以在节点中除了要提供必要的链表元素外，还需要有表示节点大小的元素。

3.2 常用的分配策略

假如说：用户申请内存量为 n，而可利用空间表中有若干个不小于 n 的空闲块存在，那么如何分配呢？这时一般有以下三种不同的分配策略[9]：

1. 首次拟合法。分配时，从可利用空间表的表头开始查找，将第一个找到的可用的空闲块分配给用户。在回收时，只要将释放的空闲块插入到链表的表头就可以了。这种方式下链表本身不按节点地址排序，也不按节点大小排序。
2. 最佳拟合法。将可利用空间表中的一个不小于 n 且大小最接近 n 的空闲块分配给用户。通常为了节省时间，此表是按节点的大小从小至大排序的。这种

方式决定了系统分配前要对可利用空间表从头到尾的扫描，同时回收时也要将回收的节点按大小放到合适的位置上。此策略可能会在系统中产生一些小的无法利用的内存碎片。

3. 最差拟合法。这种方式与最佳拟合法刚好相反，它是把可利用空间表中的一个大小不小于 n 且比 n 大的最多的空闲块分配给用户。这种表一般是按节点大小从大至小排序的。这样，在分配时，只要确定第一块空闲块有足够的容量，就可以将它分配给用户而无需查找。这种方式在回收时也要将节点按大小放置到合适的位置上。

为了获得最好的响应性能，在本文中分别实现的动态存储管理机制：改进的边界标识法、伙伴系统以及多级位示图目录法中，均采用了首次拟合法策略。

3.3 边界标识法原理及其改进

一般边界标识法的特点在于利用内存空闲块的上下边界作为标识的区域来存储内存块控制信息[9]。边界标识法将所有的空闲块链接在一个双重循环链表结构中，形成了可利用空间表。系统刚开始工作时，整个内存空间是一个大的空闲内存块，即空闲表上只有一个大小为整个内存区的节点，随着分配和回收的进行，空闲表中的节点大小和个数也随之改变。可利用空间表各节点的相关的管理信息，如节点的前向、后向指针，本内存块大小和起始地址等都存入空闲内存块中 ，一般占用该内存块的起始部分的固定字节。一般边界标识法可利用空间表中的节点结构如图 10 所示[19]：

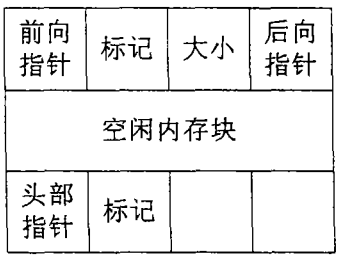

图 10　边界标识法节点结构示意图

但是我们知道，出于系统安全保护方面的需求，必须将这些重要的内存控制信息与用户使用的区域分离开来，把内存块控制信息放置到系统数据区域，只有在内核状态下，通过系统调用才能够读取、改变这些控制信息[20]。比如对内存空闲块链表进行内存申请，释放，合并空闲区域等操作时，如果不对控制信息进行保护，那么系统的安全保护机制就形同虚设了。所以这里将内存块控制信息从内存块自身移出，放到一个头结构中。每个头结构控制一块内存区。头结构是保存在系统内核空间中的。假如应用程序在申请到的内存块中产生了非法指针，指针指向了该程序无权访问的空间，如指向了内存块头结构区域，就产生了非法访问中断。

3.4 边界标识法实现中所用的主要数据结构

首先要说明一点的是因为本文分别实现了改进的边界标识法分配机制、伙伴系统分配机制、多级位示图目录法分配机制，为了最大限度的追求它们之间的共性，同时也为了减少编程的复杂度，这三个策略都使用了名为 manage_struct 数据结构，但是在具体实现不同的机制时，manage_struct 结构的定义是不同的。可以认为 manage_struct 是不同分配机制的控制结构的抽象，在不同的分配机制下，manage_struct 所代表的意义和使用方式是不同的。

在边界标识法中，操作系统在对底层的系统数据区如中断向量表、页面映射表等所需内存分配完后，将所剩下的内存创建为一个大的动态内存池。这个内存池专用于满足运行在操作系统上的任务的相关需求，比如为任务分配堆栈段、代码段和数据段、及任务运行中对内存的动态分配。任务在需要时可从中申请所需大小的内存块，并将该内存创建为一个新的内存池，也就是新生成一个 manage_struct 结构，这里称其为“内存池控制项”。任务的后续内存请求可直接从该内存池中申请。当任务的内存需求增加而现有内存池中没有足够数量的内存时，系统要从别的内存池中找到可用的内存空间，并将这些内存添加到这个任务的内存池中。这种机制的特点在于尽量保证任务数据存在于一片连续的内存空间，同时也方便任务间的共享和映射。

一个 manage_struct 主要是用来管理内存块队列的结构，其主要数据项描述如下：

```
struct manage_struct {
        struct manage_struct * prev;
        struct manage_struct * next;
        struct mem_border * block;
        unsigned long maxfreesize;
        unsigned long allsize;
        unsigned  long numofblock;
        struct mem_border * searchpointer;
        struct mem_border * maxblock;
} alloc_policy;
```

其中各个数据项解释如下：

- prev 和 next 将内存池控制项连接成一个双项链表，为了能够适应内存操作中的多样性和复杂性以及快速性的需求，内存控制信息多是用双向链表的形式来组织的；
- maxfreesize 是这个内存池所包含的内存中当前能够分配的最大空闲块的大小；
- allsize 表示了这个内存池包含的全部内存块的容量；
- numofblock 说明了这个内存池中有多少个内存块构成的；

- maxblock 指向内存块队列中具有最大容量的第一个块体的头结构。
- block 指针指向一个 mem_border 类型的内存块控制头结构，mem_border 自身也形成了一个双向链表，一个 mem_border 控制一个内存块，整个 mem_border 链表就形成了这个 manage_struct 内存池控制项所控制的全部内存空间；
- searchpointer 也是一个指向 mem_border 结构的指针，用来记录上次已经搜索到的内存块头结构队列的位置，当下次搜索操作时可以从上次搜索完成的位置开始，跳过了前面已经分配了的内存块，从而加快查询空闲块的速度。

mem_border 内存块头结构项的数据结构定义如下：

```
struct mem_border {
        struct mem_border * prev;    //形成双向链表
        struct mem_border * next;    //形成双向链表
        unsigned long size;
        unsigned long startaddress;
        unsigned long endaddress;
        unsigned long flag;
};
```

其中：

- size 是该头结构所代表的可用内存区域的大小；
- startaddress、endaddress 标识了可用内存块的起始地址和结束地址（物理地址），由于采用的是按页分配的原则，不足一页也按一页分配，所以地址的低 12 位均为 0。
- flag 是标记字，用来描述一下该内存块的特性，这里主要是用于判断该内存块是否已经分配了。

一个内存块头结构的示意图如图 11 所示：

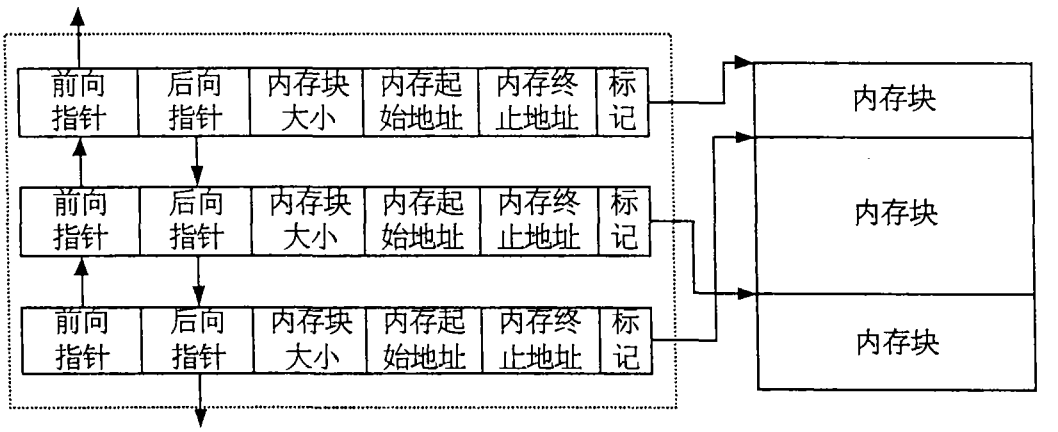

图 11　内存块头结构示意图

有了如上的数据结构定义，此时系统内存动态管理结构如图 12 所示：

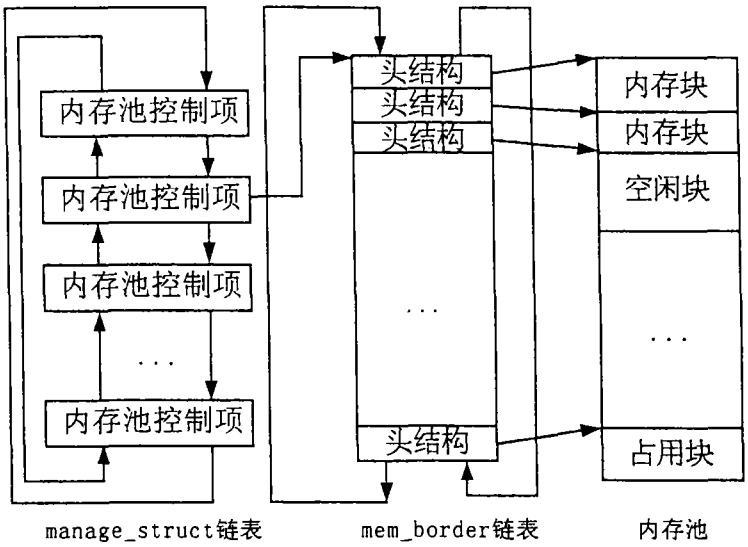

图 12　边界标识法内存管理结构关系图

3.5 边界标识法管理机制分配算法

在内存分配回收时，动态内存管理系统主要需要解决三个问题：

a. 对于任务请求中要求的内存长度，分配程序从内存池中寻找出合适的空闲区分配之；

b. 分配空闲区后，更新内存块头结构链表；

c. 任务释放内存资源时，和相邻的空闲区进行合并，并更新内存头结构信息；

从内存池中寻找空闲区的策略采用了最先适配法。要注意的一点是：内存是以一页（4k）为分配单位的，需求不足一页仍按一页分配，这种做法在简化了系统内存管理机制，降低了内存管理的复杂度的同时，也带来了不利的一点：造成内存空间的利用率的降低，形成了内碎片。

在内存保护模式下，采用虚拟存储器管理的方法扩展了物理内存空间，使每个任务都拥有独立的逻辑地址空间。系统的物理内存由内核进行管理，以内存池、内存块链表和内存池控制项链表等结构管理系统内存。任务需要向系统提出内存申请时，通过系统调用接口来实现。动态内存管理模块将系统分配给任务的内存自动映射到任务空间。

边界标识法内存块分配算法流程如图 13 所示：

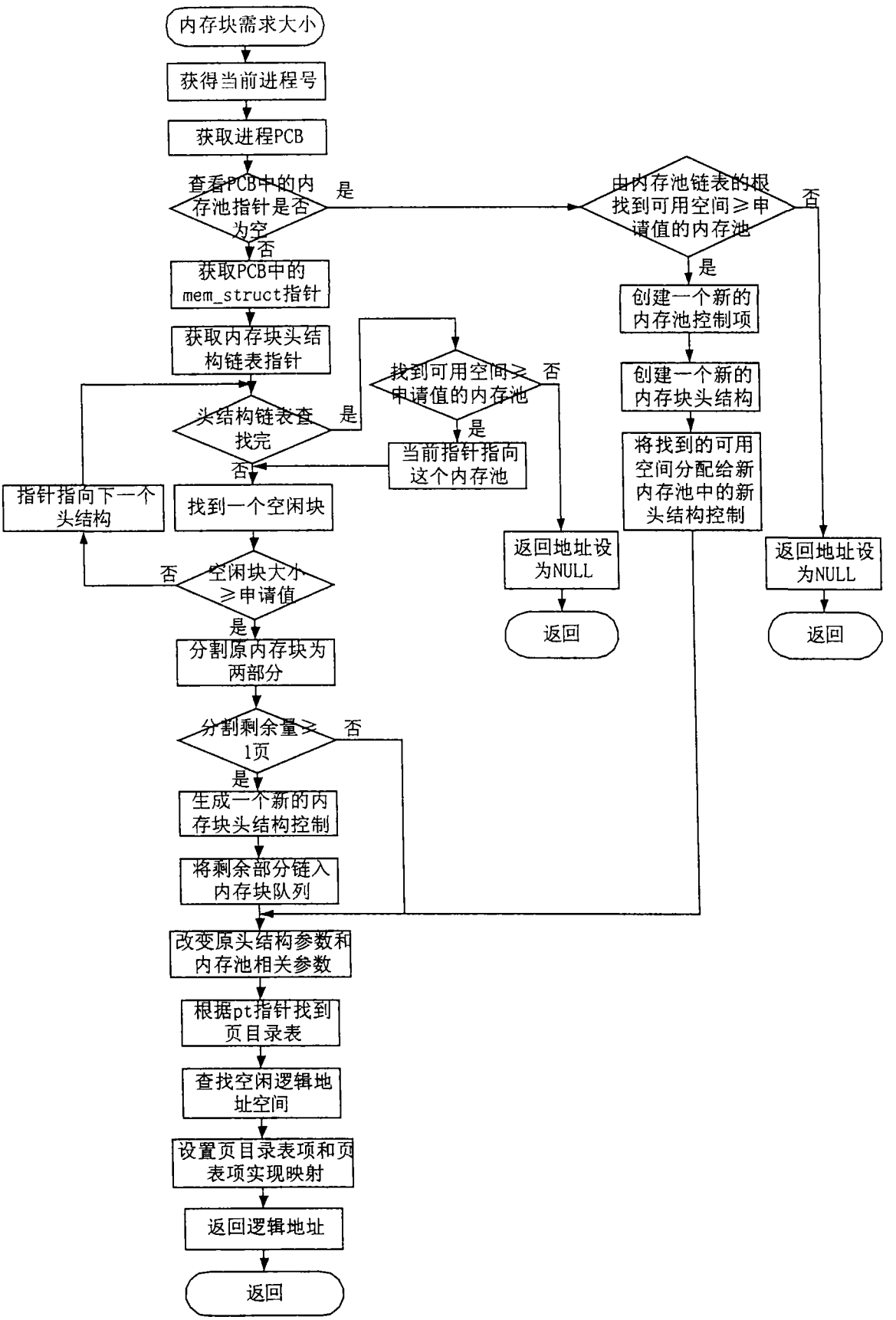

图 13　边界标识法内存分配流程图

在分配算法中，我们可以看出：

- 在每个进程控制块中都有一个 manage_struct 类型的指针，指向该进程的内存池控制项，当一个进程是新创建的时，这个指针初始化为 NULL，也就是还没有分配任何内存。系统内核中保留有一个全局变量，其指向内存池控制项的双向链表，所以也称其为内存池的“根”，当开始分配一个新的内存池时，就需要从这个根出发，找到一个新的可用空间，并将其创建为一个新的内存池。所以，当系统一开始时，只有一块大的内存池，随着内存分配的进行，会从这个大内存池中分裂出许多小的内存池。
- 在进程控制块中还有一个指向 mem_struct 结构的指针，在 mem_struct 结构中既包含有物理内存页面目录地址 pt，同时也有虚拟地址控制结构 vir_mem_struct 的指针，通过这个结构可以将虚拟地址和物理地址关联起来。按照获得的物理内存块的页号和虚拟地址来填写页目录项和页表项，大体的过程是：首先将虚拟地址分解出页目录索引、页表索引，检查对应的目录项和页表项是否存在，如无，则生成新的目录项和页表项，并按物理内存页号填写；如有，则说明此虚拟地址以前曾分配过了，要查看对应的页面的实际情况分别进行处理。
- 根据 vir_mem_struct 的结构，在虚拟空间寻找可用空间的算法也是有其一整套机制的。对 vir_mem_struct 的查找效率也是对系统性能有很大影响的，由于研究的重点在于不同物理内存分配策略对于系统效率的影响，所以这部分就不做详细叙述了，但要注意的是：在实现的三种分配策略中，虚拟空间的分配都采用了相同的算法，所以在比照时的条件是相同的。
- 当在进程自身的内存池中找不到合适的空间分配时，就需要沿着 manage_struct 链表查找下一个有足够连续空间的内存池，并在这个内存池中查找到一个有足够空间的内存块（按照最先适配法），将这个内存块从这个内存池中摘下来放入到申请内存的进程的内存池中，所以，在系统运行了一段时间以后，内存池中也可能并不都是连续空间了，但是，分配的基本原则仍然是在有连续空间可用的时候，优先保证分配连续空间给用户。
- 在内存池中有合适的内存块可以满足分配要求的情况下，要比较空闲内存块和需求内存量的大小，如果空闲内存块的大小比需求内存量多一页（4K）以上，就要将空闲内存块分裂成 2 块，例如，假设需求内存量为 A 字节，空闲内存块的大小为 B 页，那么其中一块的大小是：（A/4096）页，将原来的内存块头结构 P 按此块进行修改，并将其分配给用户；剩下另一块的大小是 B－A/4096（页），此时要新生成一个内存块头节点 Q，按这个剩余的块修改 Q 的信息，并将 Q 插入到头节点链表队列中 P 的后面。当空闲内存块的大小比需求内存量多不到一页的空间时，就将此内存块全部分配给用户。可以看出，这种分配是以页为基本单位进行的，不足一页按一页分配，所以存在一定的浪费现象，在极端情况下，用户申请一个字节的空间，就要分配一个页面的空间给用户，当

然，由于页表映射的作用，当用户在此区域再申请小空间内存时，可以通过已经映射了的空间分配给它。

3.6 边界标识法管理机制回收算法

边界标识法内存块回收算法如图 14 所示：

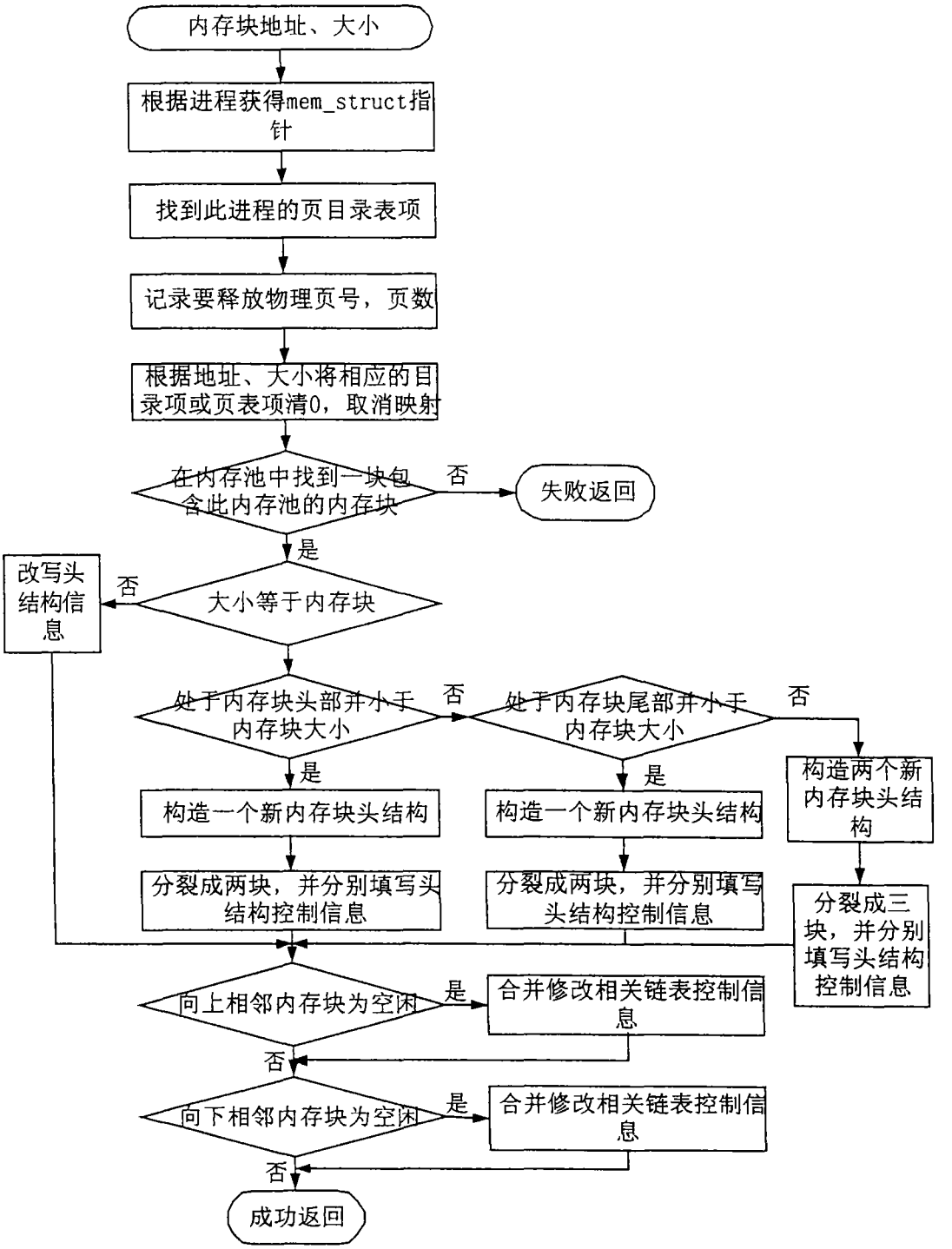

图 14　边界标识法内存回收流程图

关于回收算法的一些说明：

- 内存回收的步骤实际上分两步：第一步是解除物理内存和虚拟内存之间的映射关系；第二步是修正内存块控制信息，使已分配内存块变为空闲内存块，以备再次分配使用；
- 根据逻辑地址查找到相应的页目录项，再根据释放内存的大小计算需要释放的内存页数目，因为分配时是以页面为基本单位的，所以释放也是以页面为单位的，对于不到一页的部分，因为可能还有别的虚拟地址映射到了此页上，所以系统暂时不将其映射取消，而是等到进程退出时，将内存池中的剩余占用内存释放。
- 要释放的内存块（释放块）在内存中的位置可能有几种情况：第一种最简单，就是内存块与释放块完全重合，这样就直接修改头结构就可以了；另一种情况是：释放块只是此内存块中的一部分内存，这又分为三种可能：一种是释放块占据内存块的头部，这种情况要分裂成两块，生成一个新的内存块头结构来管理后半截内存块；另一种与这种情况类似，不同在于释放块占据内存块的尾部，这种情况也是需要分裂成两块的；还有一种是释放块占据内存块的中间部分，这种情况要将此内存块分裂成三块来处理。
- 由上可见，不论是在分配还是回收时，内存块头结构的链表都可能在不断分裂增长的，如果不采取措施，过长的链表会严重损害系统性能。所以在回收机制中要加入合并的过程，合并的准则如下:一个空闲块 A，若其在头结构链表中的上一个相邻的内存块 B 空闲，并且 B 块的 endaddress=A 块的 startaddress，就将 A 块和 B 块合并，并改写 B 块头结构的信息为新的大空闲块的信息，同时删除 A 块的头结构；若其在头结构链表中的下一个相邻的内存块 C 空闲，并且 C 块的 startaddress 等于 A 块的 endaddress，就将 A 块和 C 块合并，并改写 A 块的头结构的信息为新的大空闲块的信息，同时删除 C 块的头结构。
- 如果在内存池中找不到一个可以包含释放块的内存块，这是不正常的情况，所以要返回释放失败的信息给用户，以便用户来进行判断处理。

3.7 边界标识算法的分析

从上述的算法可以看出，由于采用双向链表结构控制，在链表中插入和删除节点的时间基本上是可以忽略的，分配和回收算法的时间花费主要是集中在链表中查找、定位内存块的时间，时间复杂度与链表的长度成正比。一般应用中，由于使用了搜索指针以及采用了最先适配法策略，通常情况下可以很快的找到需要的内存块。但是随着系统的运行，可能会使链表的长度越来越长，也就使得算法的时间复杂度明显增加，在最坏情况下，当查找完一个内存池中长长的链表以后，还没有找到适配块，那么就需要到其他的内存池中进行查找，从而又涉及到另一条链表，所以花费的时间更多。

这种算法的优点在于：

1. 它是对边界标识法的改进，在边界标识法的基础上提高了系统的可靠性和安全性。原来在边界标识法中，内存块控制信息就放在该内存块内部。改进

后，操作系统对内存块头结构的维护保证了普通用户任务无法对其进行非法操作；

2. 该算法趋向于分配连续的空间给用户使用，比较适合于内存需求不旺盛的系统，也就是内存的分配释放活动不频繁的系统。在当前的大量的嵌入式操作系统应用中，有很大一部分应用程序在启动后就不再进行内存请求了，因此，这种分配策略比较适合这部分嵌入式操作系统；
3. 算法比较简单，没有复杂的理论，实现起来比较直接、方便。

但同时这种算法也带来了几个问题：

1. 任务释放内存块时，系统处理的复杂度增加了。原先的边界标识法中，内存空闲块中就包含了控制信息，分别位于内存块的首部和尾部，当进行内存空闲块操作时，只需要从内存块的起始地址开始的固定偏移处将标记位设置为空闲，同时去读出内存块前后向指针的内容，然后判断是否需要进行合并操作就可以了。当把这些控制信息从内存块本身中移出来后，这些控制信息都被存放在链表中，系统需要在链表中遍历查找被释放的内存块对应的头结构，才能进行相关的处理，算法的复杂度增加了。
2. 当系统对实时性要求较高时，这种策略可能会力不从心，原因有以下几点：a.当没有足够的连续的空间给用户使用时，要从非连续的空间中寻找，此时算法的时间复杂度是较高的；b.依靠链表进行控制，算法的时间复杂度依赖于链表的长度，当链表较短时，可以获得很好的运行效率，而当链表的长度很长时，系统的运行效率可能会受到严重伤害。对于实时性操作系统来说，一个实时性的衡量标准就是时间响应的确定性，也就是指：系统从外部事件发生，到开始为这事件执行的第一条指令所需要的时间不应该超过某个上界[23]。而边界标识法中链表的长度是不能确定的，所以其不能保证系统响应的确定性。
3. 再有，这种分配策略会导致内存空间分配上存在一定的浪费，这是以页为基本分配单位的分配方法都会遇到的共同问题，解决这个问题的方法是缩小基本分配单位的大小，降低分配的粒度。

第四章 伙伴系统算法的具体实现

4.1 外碎片的产生原因和解决方法

由前述可以知道，频繁的请求和释放不同大小的一组内存块，必然会导致在已经分配的内存块内散布了许多小块的空闲可用块，这就是所谓的外碎片问题，由此可能会产生：即使有总量足够的空闲空间的内存能够满足内存请求，但要分配一个大块的连续区域却不能得到满足。

从本质上说，避免或减少外碎片的方法有两种：

- 利用分页单元把一组非连续的空闲内存块映射到连续的线性地址区域。
- 开发一种适当的技术来记录现存的空闲连续内存块的状态，以尽量避免为满足对小块内存请求而把大块的空闲块进行分割。

这里采用的是第二种方法，原因在于：a.在请求给 DMA 处理器分配内存缓冲时，DMA 是忽略分页单元而直接访问地址总线的，所以其缓冲必须位于连续的物理内存中；b.如果采用第一种方法利用分页单元来映射成连续地址的方式，会频繁的修改页表，必将导致 CPU 频繁地刷新 TLB 的内容，使得平均访问内存次数增加。因为程序运行在连续的物理空间可以提升系统的整体性能，所以现代操作系统都倾向于优先分配连续的物理空间给用户。

Linux 采用了著名的伙伴（Buddy）系统算法来解决外碎片的问题[12]，这里将伙伴算法在自己的系统上予以实现，并对伙伴算法中的主要数据结构进行了改进，同时也改进了分配和回收算法的流程。

4.2 伙伴（Buddy）系统原理

首先解释一下何谓“伙伴”（buddy）。伙伴系统将所有的空闲内存块组织成多条链表，每个块链表中分别包含大小为 1，2，4，8，16，32，64，128，256，512，1024 个连续页面的内存块若干个，每个块的起始物理地址不是任意的，而是与该块的大小相关，也就是每个块的物理起始地址必须是该块大小的整数倍，比如说：大小为 32 个页面的内存块，其起始地址必须是 32×2^{12} 的整数倍。伙伴系统的分配和释放都是基于伙伴块的基础的，比如释放时，系统总是试图把大小为 2^k 的一对空闲块合并为一个大小为 2^{k+1} 的空闲块。假设伙伴系统中的一个空闲块的起始地址 p，大小为 2^k，则计算其伙伴块的起始地址 q 的公式如下：

$$\text{if}\ (\ p\ \text{MOD}\ 2^{k+1} == 0\ )\quad q=p+2^k;$$

$$\text{if}\ (\ p\ \text{MOD}\ 2^{k+1} == 2^k\ )\quad q=p-2^k;$$

综上所述，将满足以下三个条件的两个块互称为伙伴[10]：

1. 两个块具有相同的大小，记作 2^k；
2. 它们的物理地址是连续的；
3. 第一个块的起始物理地址是 $2^{k+1} \times 2^{12}$ 的整数倍；

4.3 对伙伴系统数据结构的改进

在伙伴系统中，无论是内存占用块还是空闲块，其大小均为 2 的 k 幂次，其中 k≥0，且 k 为整数。比如：应用程序申请 1352 个字节的内存量，基本分配单位为 1 页 4k 字节，最终分配给用户的空间是 2^0=1 个页面，当申请量为 4100 字节时，就要分配给 2^1=2 个页面。假设总的可用内存容量为 2^m 个页面，则内存块只能划分成大小分别为 2^0，2^1，2^2，…2^m 个页面的规模。这种作法的好处在于便于分配连续的大块物理内存页面给用户。

基于前述的内存管理数据结构，适用于伙伴系统的 manage_struct 结构描述如下：

```
struct manage_struct {
        struct manage_struct * prev;
        struct manage_struct * next;
        struct mem_buddy membuddy[MAXBUDDY];
};
```

这里的 membuddy 数组用于管理不同连续长度页面的链表，页面的连续长度与数组的下标成正比，比如：membuddy[0]管理着一条规模为 2^0 个页面大小的内存块的链表，其中每个内存块大小都是 2^0 个页面，而 membuddy[5]管理着一条 2^5 个页面大小内存块的链表。MAXBUDDY 是一个常数定义，表示最大的连续页面的数量，这里定义 MAXBUDDY 为 11，也就是说在 membuddy[10]数组中管理着 2^{10} 个页面大小的连续空间，即 1024×4k=4M 字节，4M 的连续空间已经可以满足绝大部分应用程序的要求，即使不够，再通过查找下一个空闲的内存块分配出去即可。

mem_buddy 的主要数据定义如下：

```
struct mem_buddy {
        struct page * freearea;
        unsigned long * bitmap;
        unsigned long count;
        ……
};
```

其中：

- bitmap 是一个位图的指针，伙伴系统通过这个位图来标识了当前内存的使用状况，bitmap 位图的大小是由两个因素决定的：一个是和位图所属的 membuddy 数组项有关；另一个是当前系统现有的可用内存容量的大小。数组项

membuddy[k]中的 bitmap 的每一位都描述了一对大小为 2^k 个页面的伙伴块的状态，假设该位图的某位为 0，表示一对伙伴块中要么两个都空闲、要么两个都已被占用；如果某位为 1，则这一对伙伴块中肯定有一块已经被分配。假如两个伙伴块都是空闲时，伙伴系统首先将它们作为一个大小为 2^{k+1} 的单独块来处理。举个例子来说，假设当前系统可用的内存总量为 128MB，首先，128MB 可以分成 32768 个页面（每页面占 4K），16384 个大小为 2 个页面的内存块，8192 个大小为 4 个页面的内存块，依次类推下去。因此，对应于 membuddy[0]中的 bitmap 位图的大小是 32768/2=16384 位，对应于 membuddy[1]中的位图的大小为 16384/2=8192 位，对应于 membuddy[2]中的位图的大小为 8192/2=4096 位，依次类推可得出下表 1：

表 1 伙伴系统位图规模表

内存量（字节）	membuddy 索引	内存块大小（页面数）	位图大小（位）
128M	0	1	16384
	1	2	8192
	2	4	4096
	3	8	2048
	4	16	1024
	5	32	512
	6	64	256
	7	128	128
	8	256	64
	9	512	32
	10	1024	16

- count 即为该 mem_buddy 数组项的种类标记，用来指出该数组项是属于哪一个级别的，同时也就指出了该数组项的 bitmap 中包含了多少位的信息。
- freearea 指针是指向 page 结构的指针，前面说过，系统中定义了一个 page 数组，数组中的每一项都是实际物理内存的一页的抽象，是一一对应的关系。在伙伴系统中，membuddy[k]中的 freearea 指向了一个大小为 2^k 个页面的空闲内存块的头一个页面所对应的 page 结构，而随后这个 page 结构的 next 指针指向了下一个大小为 2^k 个页面的空闲块的头一个页面所对应的 page 结构，这样，就形成了一条大小为 2^k 个页面的空闲块的链表；同时，这些 page 结构中的 prev 指针也进行了适当的设置，使其指向它的前一个节点，最终 freearea 指向的是一个由若干个空闲块头一个页面所组成的双向链表。这个链表中的内存块都是空闲可用的，当一个内存块被分配了以后，就将代表它的头页面的 page 结构从链表中摘除。freearea 的作用可以理解成主要用于内存的分配，而 bitmap 的作用则主要在于内存的回收和合并。

- 需要说明的是，系统在刚初始化的时候，各个级别的空闲链表中，除了最大的membuddy[10]的那条以外，其他链表还都是空的，系统的全部可用物理内存都放在 membuddy[10]的链表中，随着系统的运行，从大块分裂成小块，然后再插入到相应的空闲队列中。

  伙伴系统内存管理结构关系图见图 15。

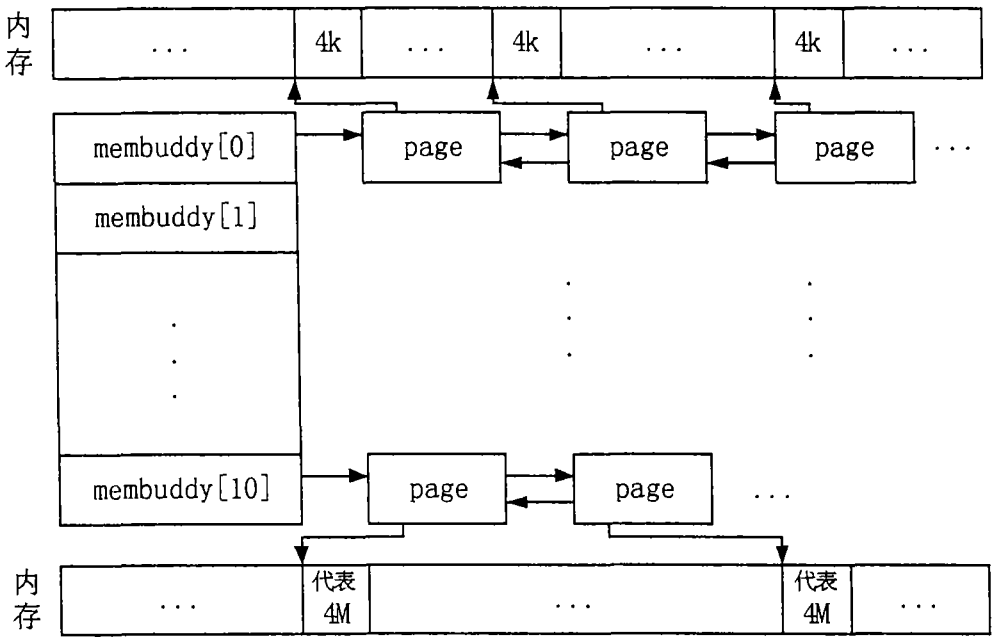

图 15 伙伴系统内存管理结构关系图

4.4 伙伴系统分配算法

有了上述的管理结构，就可以利用伙伴系统对内存进行有效的管理了，实际应用中伙伴系统是比较复杂的，这里只是对于其中最关键、核心的行为进行了一番描述，而在系统在实现中涉及到许多细小、琐碎的环节，就不做进行一一陈述了。

伙伴系统内存分配算法主流程图如图 16 所示：

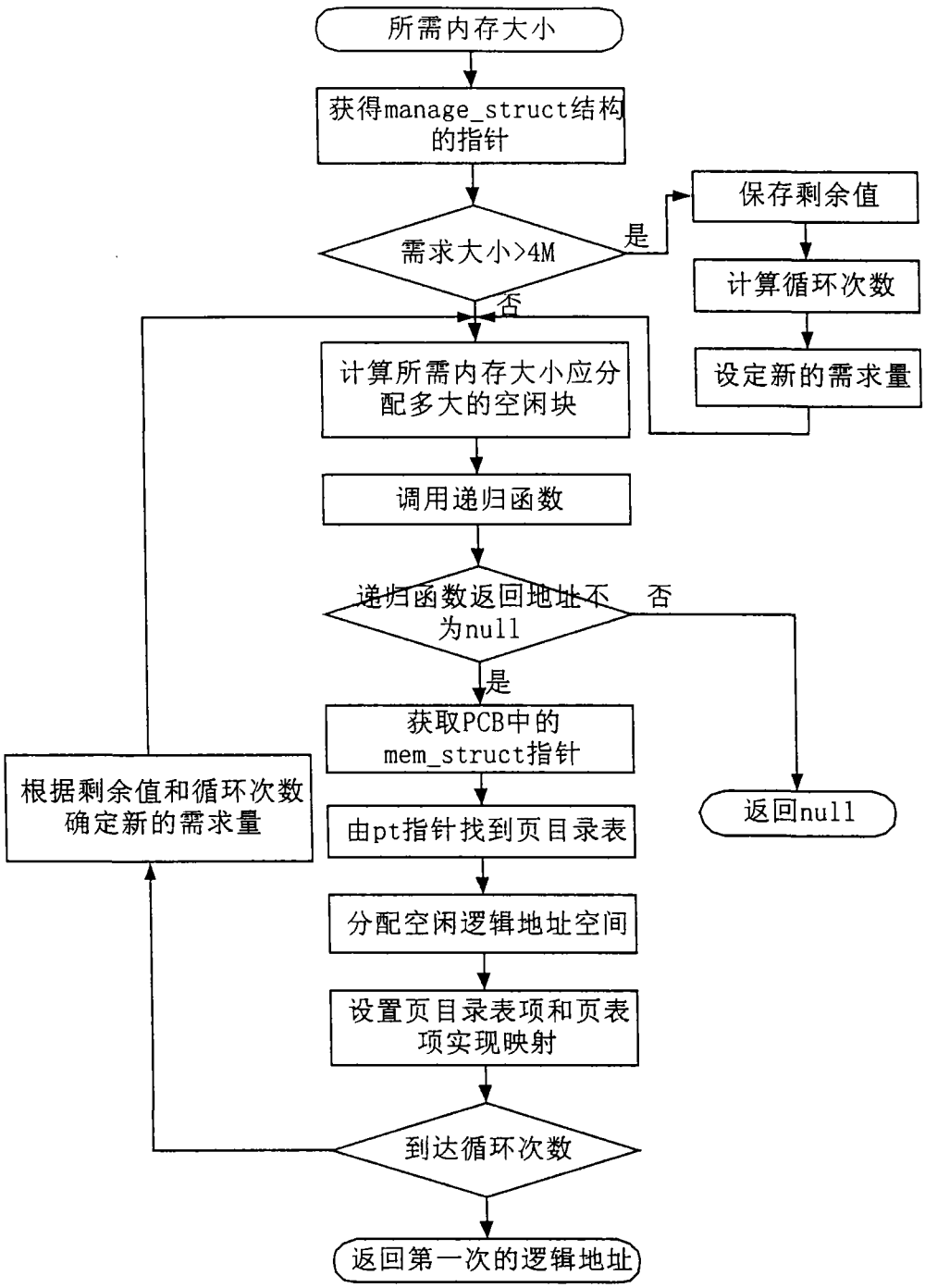

图 16　伙伴系统内存分配算法主流程图

对于分配算法流程有以下几点说明：

- 首先，这里实现的伙伴系统算法，与边界标识法不同之处在于：伙伴系统的每个进程中的 manage_struct 指针都指向了同一个 manage_struct 结构，而边界标识法中各个进程的 manage_struct 指针是指向不同结构的，代表了不同的内存池空间。可以认为伙伴系统使用了一个大的、公共的内存池来统一管理。

- 从入口参数内存需求大小 A，可以很容易的得出应该分配给用户的物理内存块的实际大小 B：
  如果 $2^{k+1}\times2^{12}>A>2^{k}\times2^{12}$（其中 k 是界于 0 和 MAXBUDDY 之间的整数值），则 $B=2^{k+1}\times2^{12}$；如果 $A=2^{k}\times2^{12}$，则 $B=2^{k}\times2^{12}$。
- 还有一点要注意的是当请求超过了 4M 内存，也就是说按最大的块（4M）分配一次不能满足请求，这时系统要有一些特殊处理：先按照内存请求为 4M 进行，然后再根据剩余的请求量依次来进行处理。当然，就目前的应用程序状况来说，一次分配 4M 内存的连续空间可以满足绝大多数应用程序的需求。
- 算法在获得了物理内存块地址以后的处理，如分配虚拟地址空间、页表映射等过程与前面的边界标识法中讲述的一致，这里就不再多作陈述了。

分配流程的核心是一个递归函数，通过这个递归函数可以获得要分配的物理内存块的地址，该函数的流程图见图 17：

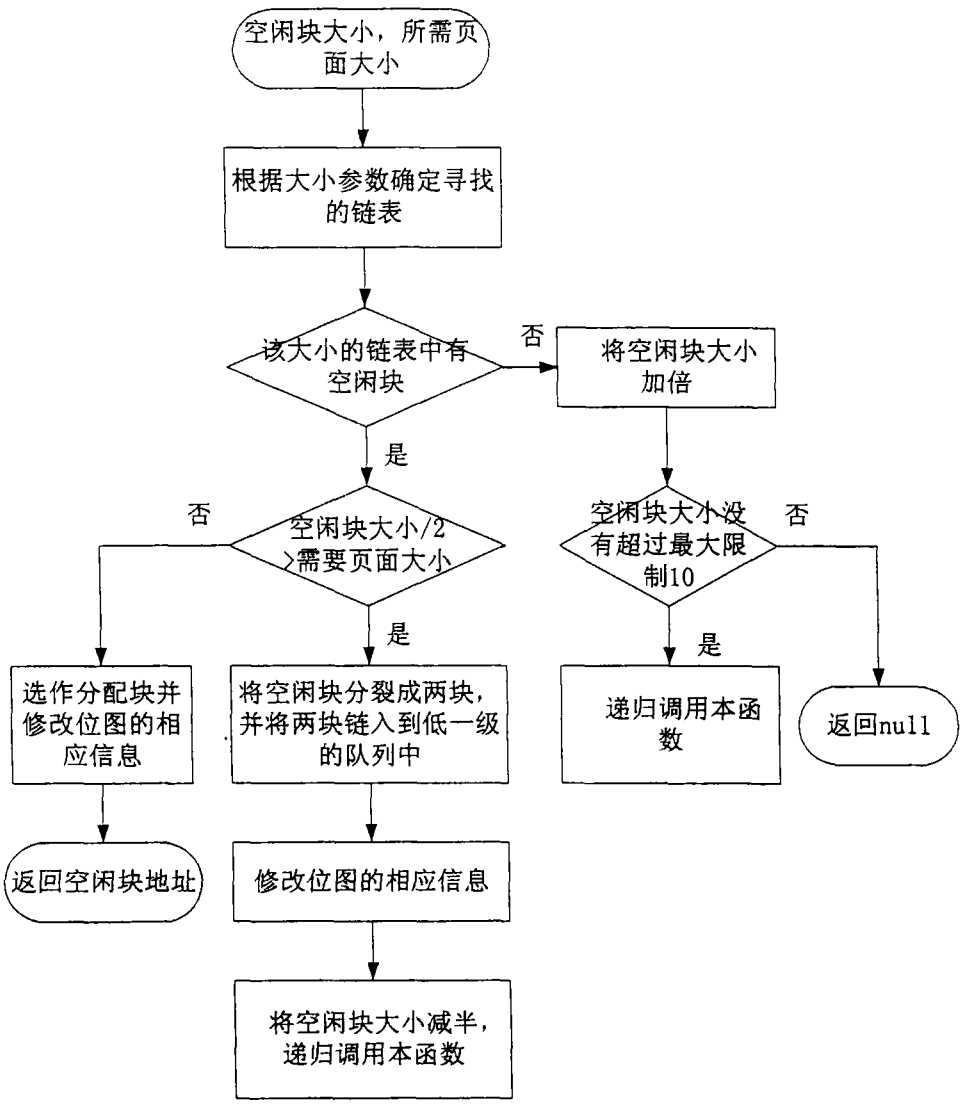

图 17　伙伴系统分配核心函数流程图

- 这个函数的入口参数是空闲块大小以及所需页面大小，函数原型为：get_free_block（unsigned long blocksize, unsigned long needsize）；
- 此函数根据空闲块的大小，去查看相应的 mem_buddy 数组项的 freearea 队列有无空闲块可用，若存在空闲块，就用空闲块的一半与需求量进行比较，如果空闲块的一半小于需求量，就把这个空闲块分配给用户；相反的，若空闲块的一半不小于需求量，说明用这个空闲块分配就太浪费了，它至少可以满足 2 倍的需求量，所以要把这个空闲块分裂成两个伙伴块，同时将它从队列中摘除，将其两个子伙伴块分别链入其对应的 freearea 队列中，然后再利用第一个伙伴块进行后续处理，这个过程是递归的，如果有一个很大的块，而需求量又很小，就有可能要进行多次的分裂才能得到需要的空闲小块。如果相应的 freearea 队列中没有可用的空闲块，那么算法就要向上一级（容量多一倍）的 freearea 队列中查询有没有空闲块，如果找到空闲块，就又开始递归调用自身；如果还是没有空闲块，就再向上查找，一直找到最大的 mem_buddy[10]数组项中，如果还是没有找到，那么程序就返回 NULL 值表示分配失败。
- 与每个队列相对应的位图是用来表示当前内存分配状况的，系统初始时，位图中全部的位均为 0，当有一个块被分配的时候，根据此块的起始页面的地址 p（页面号）和大小 b（2 的幂次）可以很容易的算出此块在相应位图中的位置 C：C = p>>（1+b），就可以通过数组下标直接获得包含该位的信息的字 M（这里一个字长为 32），M 在位图数组中的下标应该为：C / 32，该位在字中的位置 n 是: n = C MOD 32，然后对该位进行异或操作，M ⊕（1<<n），这样这个块的位图位就会变为 1（0 ⊕ 1=1）；如果这个块的伙伴块也分配的时候，此位又经过了一次异或操作从而变为 0（1 ⊕ 1=0）。

4.5 伙伴系统回收算法

伙伴系统在分配页块的过程中将大的页块分裂为小的页块，将会使内存越来越零散。因此页回收的代码只要有可能就把零散页组合成大的页块，伙伴系统之所以采用页块的大小为 2 的整数次幂的原因之一就在于这样能够很容易将页块组成大的页块。

伙伴系统回收算法的流程图见图 18：

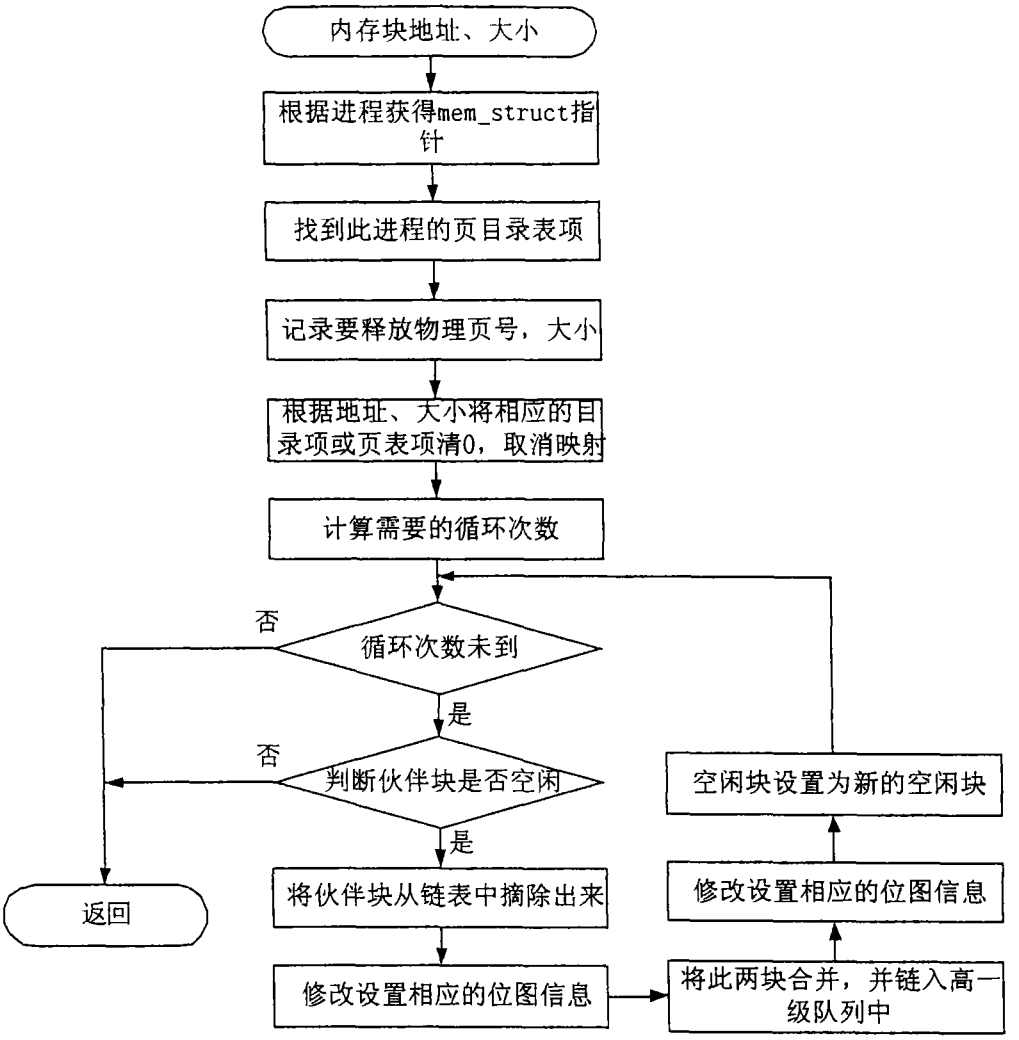

图 18　伙伴系统回收算法的流程图

回收算法流程中的一些说明：

- 只要一个页块被回收，就检查它的伙伴块是否空闲，如果是空闲，就把它和新释放的页块一起组成一个新的两倍大小的空闲页块。然后会继续检查这个新形成的空闲块的伙伴是否空闲，若有则继续合并，这样一直合并到最大规模的空闲块。
- 在伙伴系统中回收空闲块时，只当其伙伴为空闲块时，才归并成大块。也就是说，若有两个空闲块，即使大小相同且地址相邻，但不是由一个大块分裂出来的，就不能进行合并。比如说：前面一块的起始地址 p MOD 2^{k+1}!＝0，则不把它们归并在一起，而是分别链入到小一级的链表中去。
- 位图起到加速查询伙伴块状态的目的，当一个块释放时，它所对应的位图位就会经过一次异或操作，假设原来此位为 0，表示这一对伙伴块都忙，这样把此块放入到空闲链表中，然后对该位进行异或，得到 1，表示一对伙伴块中至少有一块忙，所以其伙伴块忙，不再合并；如果接着这个伙伴块也释放了，同样的，

将它放入到空闲队列中，然后对位图位进行异或，此时由 1 变成了 0，说明这一对伙伴都空闲了，此时进行合并工作：将两个块从空闲队列中摘除，合并成一个大的空闲块，然后放到高一级的空闲队列中去，到这时，就相当于释放了一个大的空闲块，程序的操作又重新开始，不同的是现在是对高一级队列的处理了，这样的处理过程一直到最大的队列为止。

4.6 伙伴系统算法讨论

伙伴系统是一种非常强调时间效率的算法，其运行的效率比较高，从上述算法流程中可以看出，在一般分配过程中，通过对应大小的链表队列可以直接找到空闲的内存块供分配使用，这个时间几乎可以忽略，要考虑的时间消耗在于，当没有合适空闲块时，把大块空闲块分裂成合适空闲块这个过程；同样的，在回收过程中，其时间消耗主要集中在把小块空闲块合并形成大块空闲块的过程中。在位图的辅助下，分裂与合并的速度还是较快的，因此伙伴系统是一种快速分配回收机制，正是由于这个原因，现在广为流行的 Linux 操作系统才采用了这种内存管理机制。

虽然伙伴系统的运行效率较高，但是并不意味着其一定可以适用于实时操作系统中，这是因为其块分裂或块合并的过程都是依赖于队列中空闲块的状态，不能够满足实时系统中对时间响应的确定性的要求，只可以说，在一定的程度上，伙伴系统提高了系统的相应实时性能。

伙伴系统的一个重要的缺点就是其空间浪费是比较严重的，举例来说：如果一个用户申请 512 个页面+1 个字节的连续空间，那么系统必须要分配给它 1024 个页面，仅仅是因为其多出的那一个字节，这时的空间利用率下降到 50%左右，浪费了几乎一半的空间，而且随着空闲块的大小的增加，浪费的现象也越来越严重。产生这种现象的主要原因在于伙伴系统是建立在“伙伴块”这个概念的基础上的，而伙伴块是一个固定的，静态的概念，对于内存中的一个物理块，它的伙伴块的位置是固定的，所以就不能很好的适应动态的需求，因此，可以认为伙伴系统是利用空间来换取时间的一种算法。要解决这个问题，同时又要保证获得良好的运行时间效率，就必须在伙伴系统的基础上，再引入别的机制来降低分配的粒度，在本文后面讲述的内核缓冲区的管理机制中就很好的解决了这个问题。

第五章 多级位示图目录法的具体实现

5.1 多级位示图目录法的提出

前面讲述了两种内存动态分配的方法，对于边界标识法来说，它将运行中形成的长度不相等的空闲存储块链接成一个可利用空间队列，无论是采取何种查找策略，都会在队列中形成许多小碎片，这些小碎片难以分配出去，使得可利用空间队列增长，也就意味着通过链表查找适配块的时间开销会增大，在这种情况下的时间开销不是常数；对于伙伴系统来说，在最坏的情况下，一次分配可能要进行多次分割才能得到适配块，一次回收可能要进行多次合并，其最差情况下的时间开销也不是常数。

用位示图管理空闲块的效率较高。无论是分配、回收、还是合并操作，其时间开销都是常数[13]。不足之处在于必须将存储池划分为长度相等的基本块，当存储池容量较大而基本块长度较小时，要用多个计算机字构成位示图，这涉及到多个字的联合移位等逻辑操作，是比较麻烦的操作。

这里，使用了一种经过改进的位示图算法——多级位示图目录法，用类似多级目录的方式将位示图组织起来，提高了系统的实时性能，比起上述方法都有效一些。

5.2 二级位示图目录结构

首先介绍一下最简单的二级位示图目录算法。设一个 4M 的存储池，将其分成等长的 32 个大块，则每个大块的长度为 128K，又将每个大块划分成 32 个等长的小块，每个小块的长度为 4k（即一个页面的长度）。用一个 32 位的字 A（用 32 位是因为 i386 字长是 32 位的，处理起来最方便）来表示存储池的分配情况，将 $A=a_0a_1a_2...a_{31}$ 称为存储池的位示图。A 中的每位 a_i 元素对应于位示图中的一位，代表了一个大块（128K）。同时，为了便于实现首次适应或最佳适应分配策略，将大块 a_i 中的最大连续空闲小块的数目存放在整型数组 B 的 B[i]元素中，再用一组 32 位的字 $C_i=c_0c_1...c_{31}$ 来表示第 a_i 大块中的 32 个 4K 小块的位示图。如此以来，就可以用三元组（a_i，c_i，B[i]）来描述第 i 大块的内存动态分配情况了。如果 4M 内存池中的第 i 大块空闲，那么 $a_i=1$（用 1 来表示空闲可用），c_i 中的所有位均为 1，且 B[i]＝32；若第 i 大块中含有被占用的小块，则 $a_i=0$（为 0 表示该块已被占用），c_i 中至少有一位为 0，B[i]为一个小于 32 的整数；如果第 i 大块已经被完全分配了，则 $a_i=0$，c_i 中也全部为 0，B[i]等于 0。举个例子，比如在大块 5 中，如果第 17 个小块空闲，则

c_{17}=1，否则 c_{17}=0。系统未执行任何动态存储分配操作之前，A 和 C 中的所有位均为 1，B[i]均为 32。

在上述的关系构造下，一块内存池的管理模式如图 19 所示：

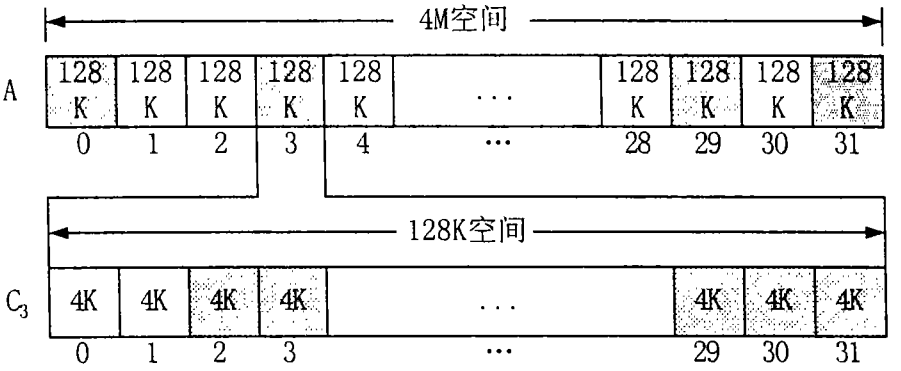

图 19　二级位示图目录法内存池构造关系图

图中的阴影部分表示已经被占用的空间，由上图可以看出，对于大块位示图 A 有：A = 01101...1010B；对于小块位示图 C_3 则有: C_3 = 1100...000B；若省略处均为空闲的，则此时的计数元素 B_3 为：B_3 = 25；

5.3 多级位示图目录

上述动态存储分配算法可以推广到三级位示图目录的情形。假设 4M 字节的存储池被等分成 32 个大块，每块 128K 字节。每个大块分成 32 个中块，每块 4K 字节。每个中块又可分成 32 个小块，每块 128 字节。此时存储池的动态分配情况可用 $L=a_0a_1...a_{31}$ 来描述，称 L 为存储池的位示图。L 中的每个 a_i 对应一个数组元素 A[i]和一个 32 位的字 $M_i=b_0b_1...b_{31}$（$0\leqslant i\leqslant 31$）。（a_i，M_i）就是第 i 大块的位示图，并且是 L 的第二级子位示图。而 M_i 中的每个 b_j 又对应一个数组元素 B[i，j]和一个 32 位的字 $S_{ij}=c_0c_1...c_{31}$（$0\leqslant i$，$j\leqslant 31$）。（$a_i \cdot b_j$，S_{ij}）就是第 i 大块中第 j 中块的位示图。与二级位示图目录的情形相似，大块 a_i 中的最大剩余记录在 A[i]中，而中块 $a_i \cdot b_j$ 的最大剩余记录在 B[i，j]中。如果存储池中第 i 大块空闲，则 a_i=1；若其中夹杂着被占用的中块或小块，则 a_i=0。在某一大块中，若第 j 中块空闲，则 b_j=1；若其间夹杂有被占用的小块，则 b_j=0。三级位示图目录的结构如图 20 所示。

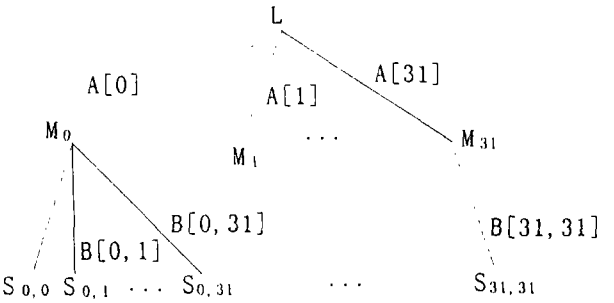

图 20　三级位示图目录结构图

利用三级位示图的算法和二级的情况类似，只是要考虑的分支情况更多一些，同理，位示图目录的结构和算法可以推广到三级以上的多级位示图目录的情形。

5.4 二级位示图目录法的主要数据结构

因为二级位示图目录是最简单的多级位示图目录，实现起来比较方便，也具有一定代表性，所以这里实现的是一个具体的二级位示图目录法存储管理机制。

在前述的内存管理基础设施的基础上，对主要控制结构 manage_struct 进行了修改，使其适用于位示图目录法的特点，其具体形式如下：

```
struct manage_struct {
        struct manage_struct * prev;
        struct manage struct * next;
        struct mem_bitview * membitview;
};
```

其中 membitview 是一个指向 mem_bitview 结构的指针，mem_bitview 的主要结构如下：

```
struct mem_bitview {
        struct mem_bitview * prev;
        struct mem_bitview * next;
        unsigned long address;
        unsigned long firmap;
        unsigned long secmap[32];
        unsigned long maxnum[32];
};
```

其中：

- address 是此大块（4M）的起始物理地址；
- firmap 是大块的位示图，即前面所述的 A 字；
- secmap 是小块的位示图，即前面所述的 C 字；
- maxnum 是表示的是大块中的最大连续空闲小块的数目，即前述的 B 数组。

系统在初始化时，将全部空闲内存都分割成了若干个 4M 字节大小的内存池，在 membitview 中设置好相应的地址信息和其它初始状态信息，通过 membitview 结构来把它们链接起来。

通过上面描述的数据结构，系统的内存管理体系结构如图 21 所示：

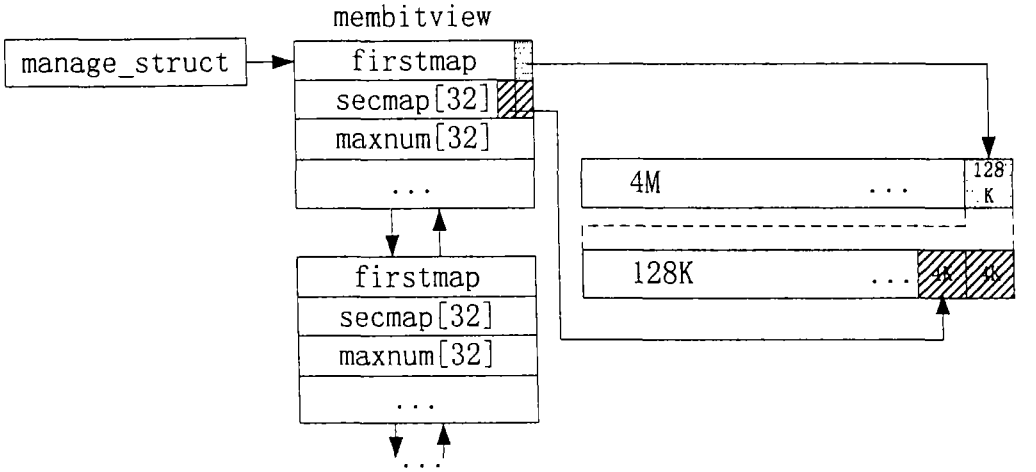

图 21　二级位示图目录法内存管理结构关系图

5.5 二级位示图目录法的分配算法

5.5.1 主分配流程

同伙伴系统算法一致，二级位示图目录法每个进程控制块中的 manage_struct 指针都指向同一个 manage_struct 结构。这个结构包含了 membitview 队列，后者描述了全部系统可用的物理内存空间。

二级位示图目录法分配算法的主流程如图 22 所示：

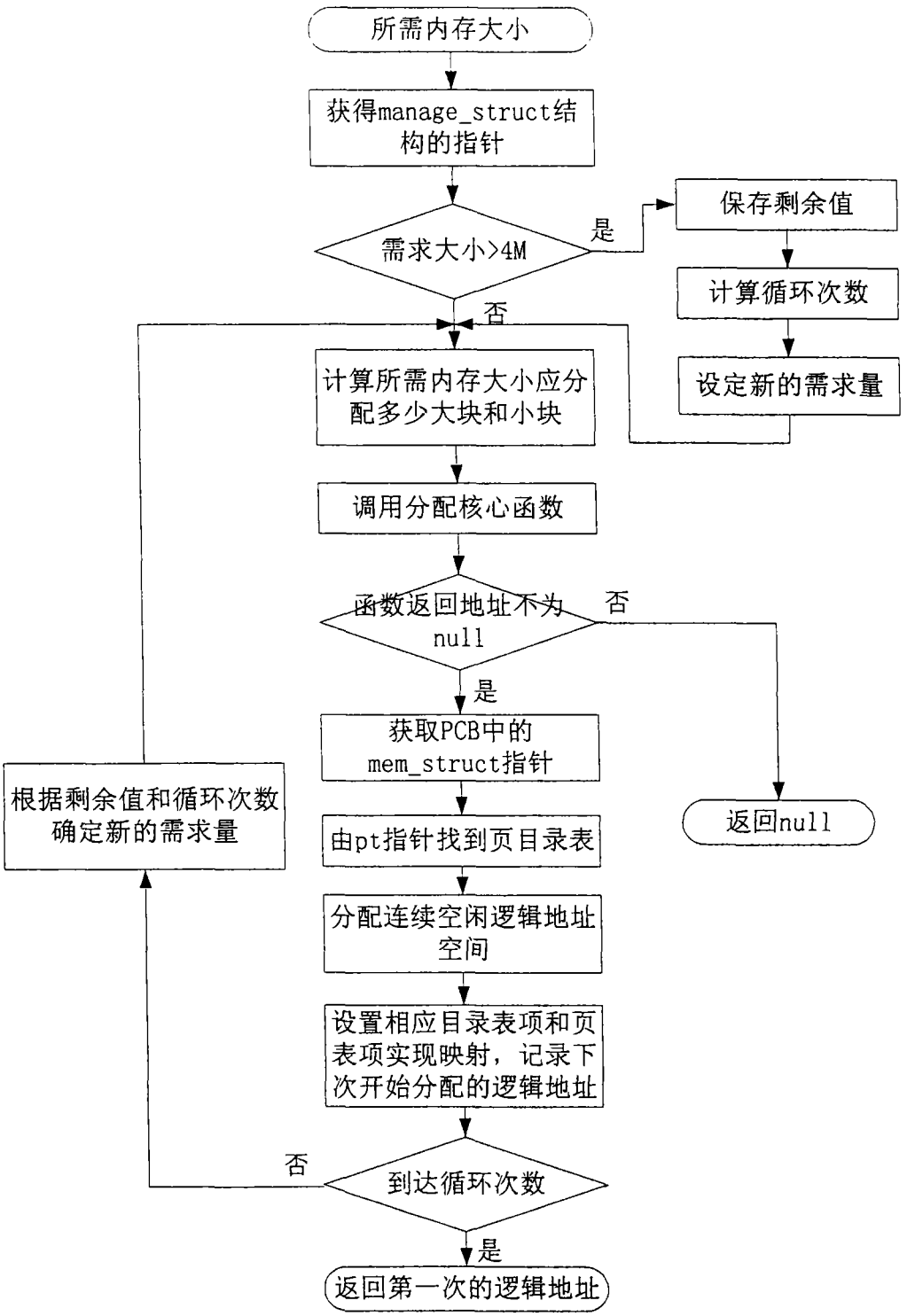

图 22　二级位示图目录法分配算法的主流程图

关于这个主流程的一些说明:

- 算法的主要实现在核心函数中，这个函数返回了分配的内存的首地址，因为最小分配单位为 4K，所以此地址是页对齐的；

- 在调用分配核心函数之前，先要通过作为输入参数的用户需求内存量 T 的大小计算出一共需要多少个大块（128K）和多少个小块（4K），这个计算非常简单，利用整除和模除就可完成了，用户需要的大块的数量 m 为：m = T / （128 $\times 2^{10}$）；令 P=T－（m × 128 × 2^{10}），则需要的小块的数量 n 为：

  n = P / 4096 + 1 ；　　（ if （P MOD 4096） ≠ 0）

  n = P / 4096 ；　　（ if （P MOD 4096） = 0）

- 如果一个请求不能得到满足，比如队列中没有足够的连续空间可用于分配，那么函数会返回 NULL 值，这时会引发后台的交换守护进程，守护进程将一些不常用的页交换到磁盘交换区上，腾出足够的空间，然后再次调用分配函数给用户分配空间。因此，后台交换的效率和策略也是至关重要的，好的后台交换方式是对分配算法的强有力的支持。当然，一般应用中这种情况发生的频率并不太高，因为 4M 的空间已经可以满足大多数应用程序的使用了。
- 与前面的分配算法相似，在核心函数返回了分配的内存物理地址以后，要通过页目录表项和页表项的映射，将物理地址和虚拟地址联系起来。要注意的一点是，如果需求超过了 4M 空间，就必须按 4M 大小将用户需求进行分割，循环多次进行分配，但是要求分配的虚拟空间是连续的，这样当返回了虚拟空间的首地址以后，就可以在页表的帮助下，将不连续的物理空间映射为连续的虚拟空间了。而且这个不连续的物理空间也不会存在过多的分段，因为核心算法中要求必须是连续的空间才可以分配。

5.5.2 核心分配函数

在分配过程中，分配核心函数可能会遇到三种情况，这里分别进行讨论。

第一种情况：用户申请量正好是大块的长度（128k）的整数倍数，在这种情况下，只要分配大块就可以了，其分配核心函数流程图如图 23：

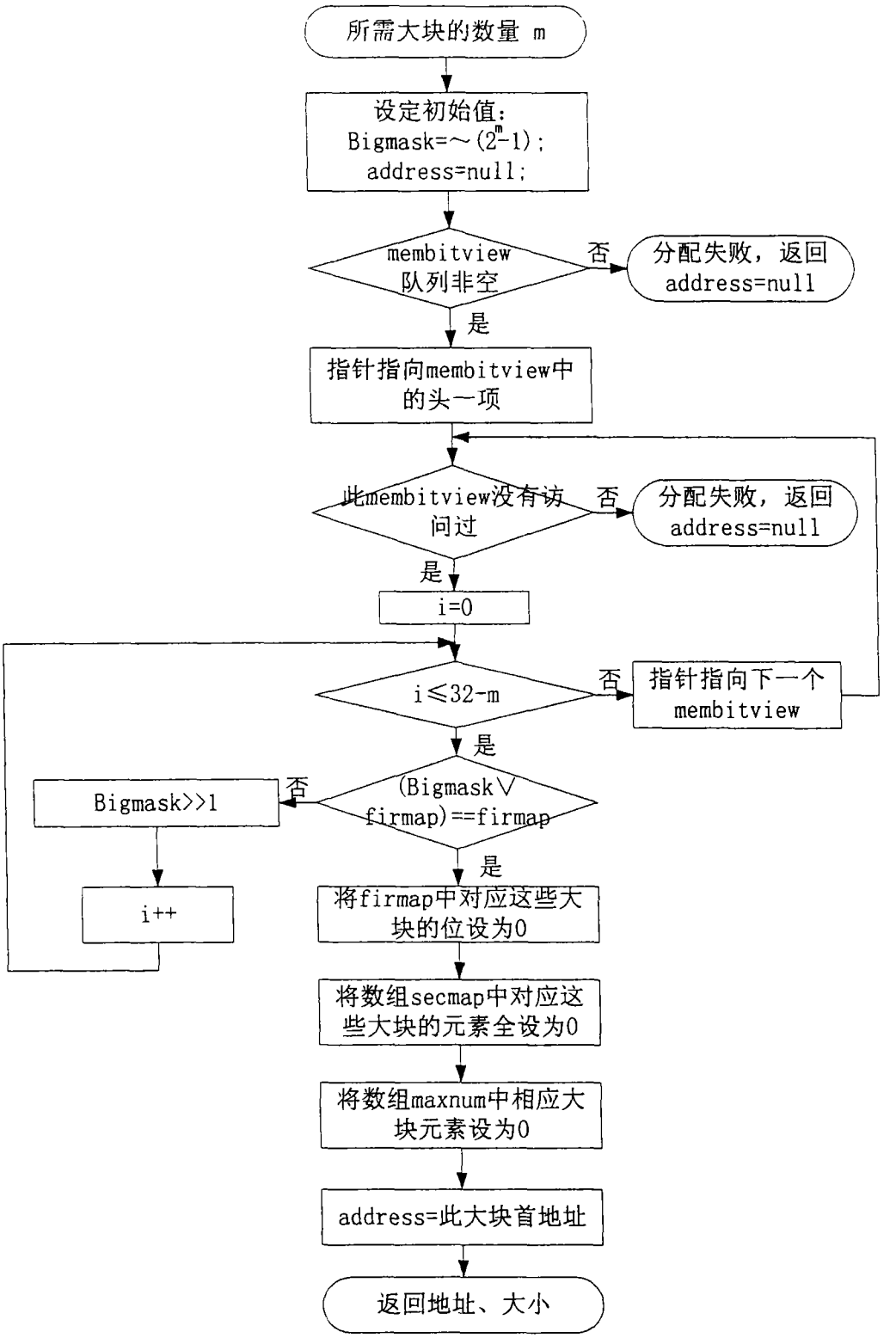

图 23　二级位示图目录法大块分配核心函数流程图

第二种情况：用户申请量正好是小块的长度（4k）的整数倍数，而需求量不到128k（大块长度），在这种情况下，只要选中一个大块分配其中的小块就可以了，其分配核心函数流程图如图 24 所示：

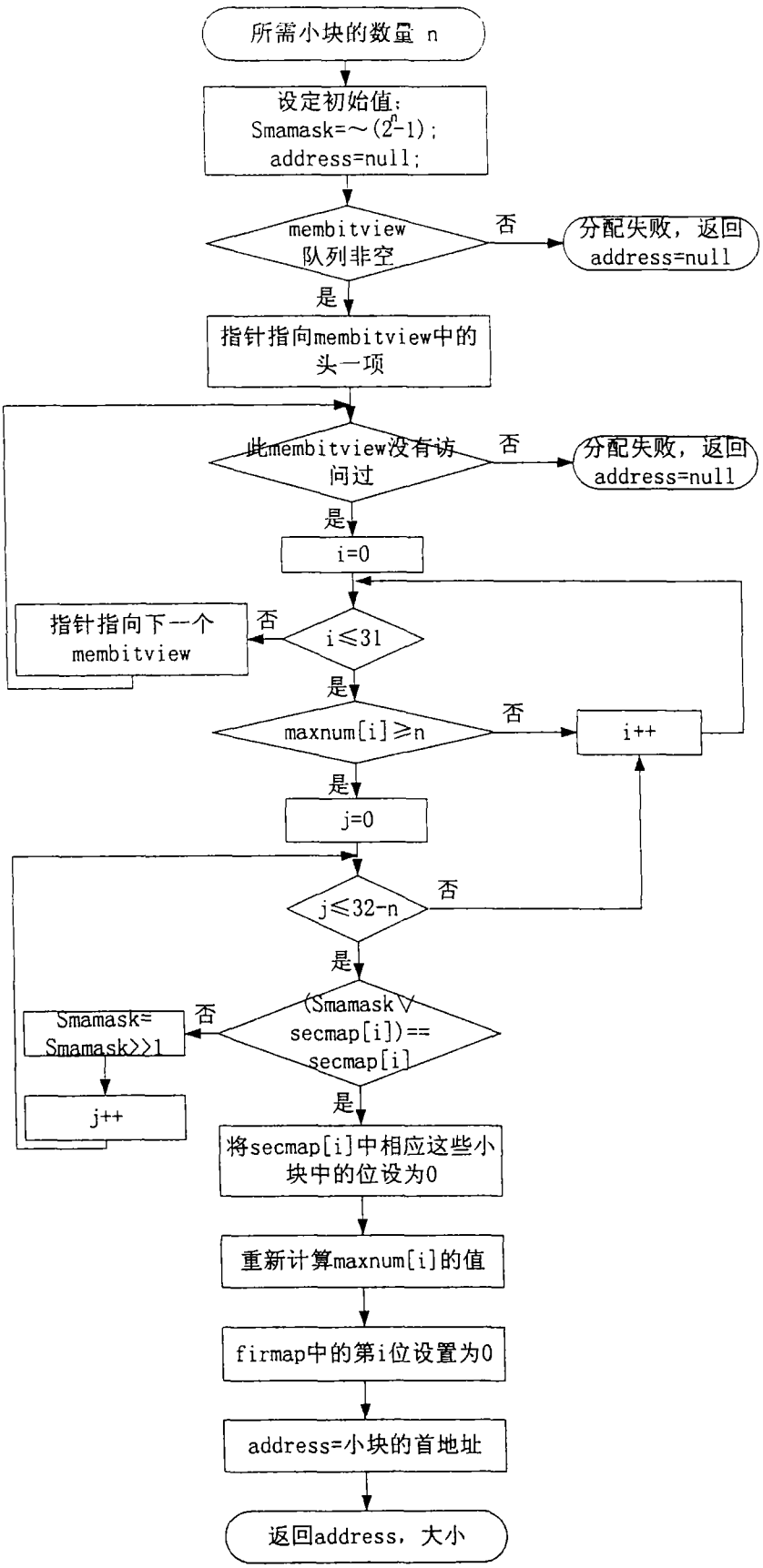

图 24　二级位示图目录法小块分配核心函数流程图

第三种：既要有大块分配，同时也要有小块分配，此时分配核心函数的流程图如图 25 所示：

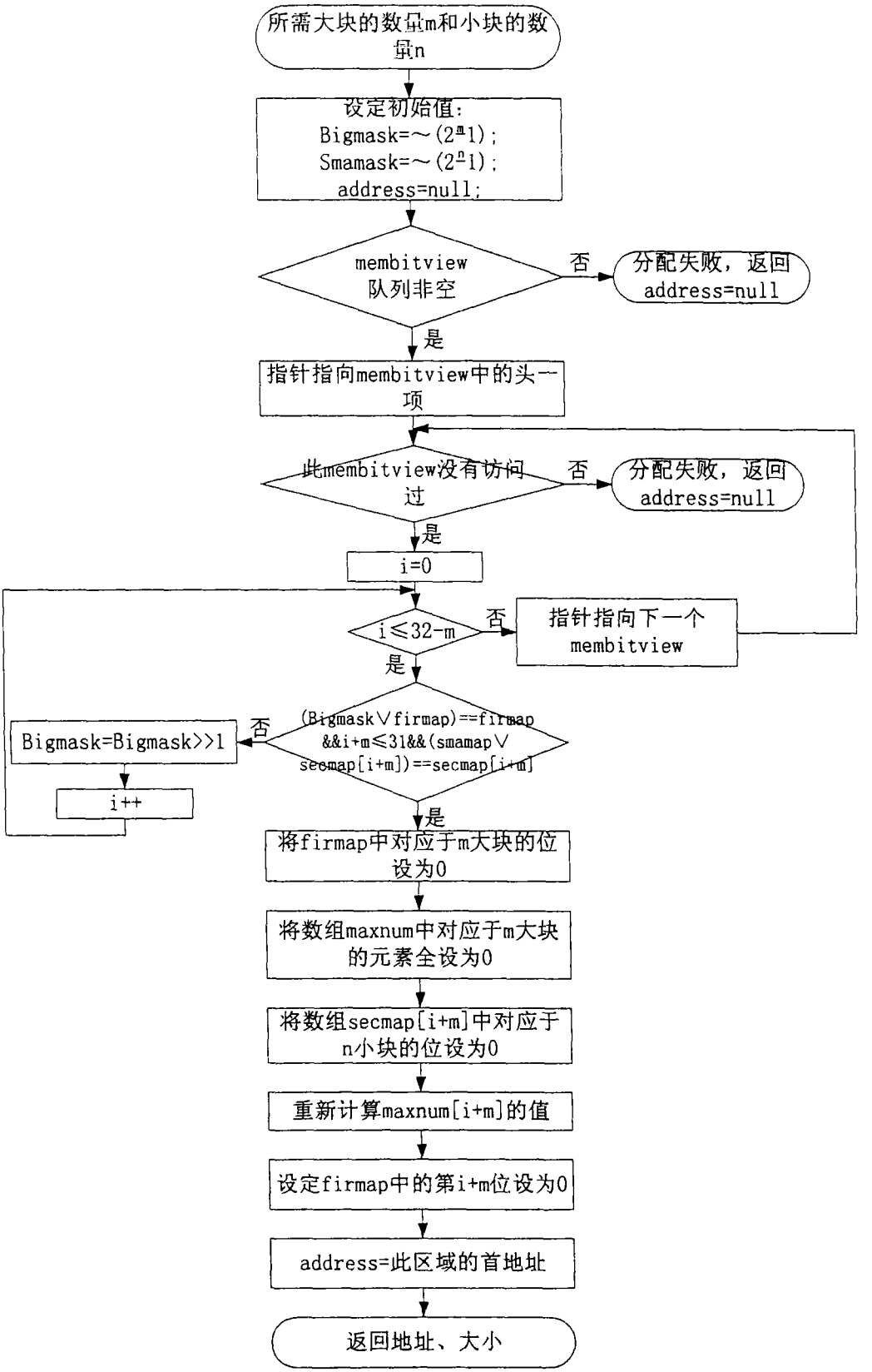

图 25　位示图目录法大小块同时分配核心函数流程图

分配核心函数的三种情况在其流程图中都有比较详细的描述，这里就不再对核心函数的算法进行赘述了。

5.6 回收算法流程

回收时也可按三种情况分别考虑，当然首先要通过已知的归还存储区的首地址和长度进行必要的计算:

由首地址 address 可以计算出大块索引数 i 和小块索引数 j：

首先 address/0x400000 得到该地址属于哪个 membitview 结构控制，也就得到了该 membitview 控制内存区的起始地址 a＝（address/0x400000）×0x400000；所以大块索引数 i＝（address－a）/0x20000；小块索引数 j=（（address－a）－i×0x20000）/0x1000;

由归还存储区的长度 d 可以计算出大块数目 m 和小块数目 n，这在前面已经介绍过了。

在得到了这些数据后，调用回收核心函数来进行回收。回收核心函数中同样又按三种情况来分别处理：

第一种情况：当只归还大块存储区时，回收核心函数的流程图如图 26：

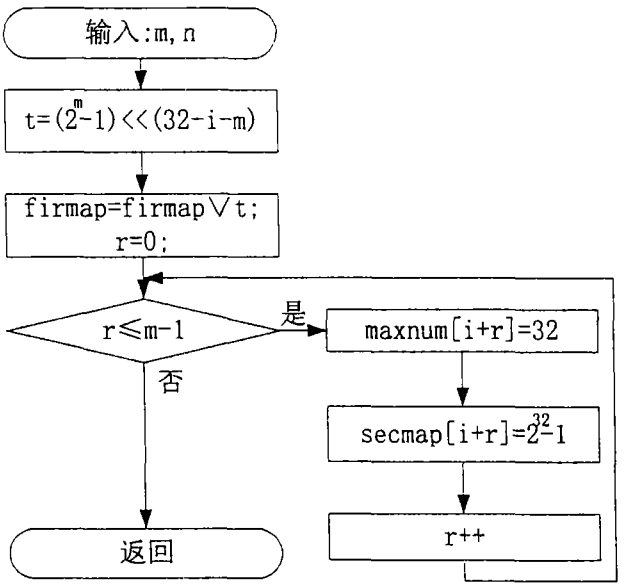

图 26　位示图目录法大块回收核心函数流程图

第二种情况：当只归还小块存储区时，回收核心函数的流程图如图 27：

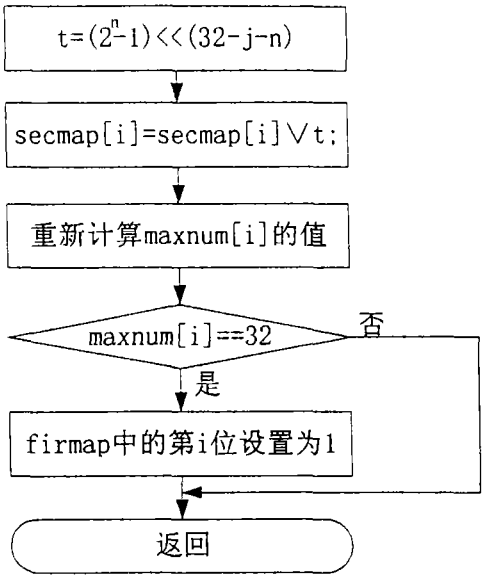

图 27　位示图目录法小块回收核心函数流程图

第三种情况：当大小块同时都要归还时，回收核心函数的流程见图 28。

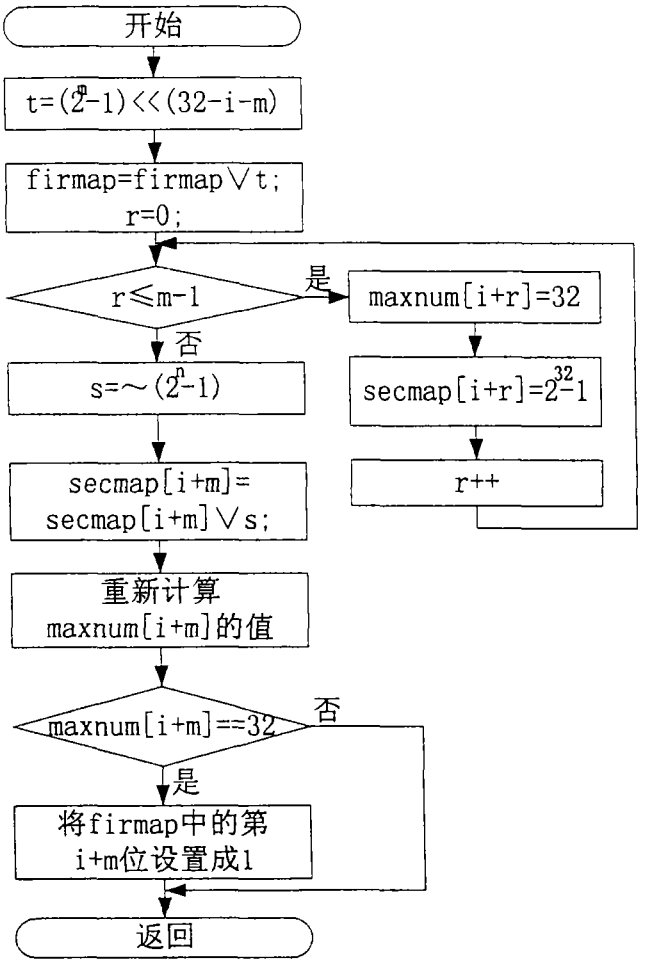

图 28　位示图目录法大小块同时回收核心函数流程图

有一点需要注意的是：当回收的内存大于 4M 的时候，说明不止是一个内存池进行回收，所以要先将回收的部分按 4M 进行划分，进行循环回收，与分配时的方式类似。回收流程中的其他部分和前面陈述的两个算法是相类似的，这里就不再赘述了。

5.7 算法分析及讨论

我们知道一个membitview代表了4M的内存空间，这对于一般的应用程序来说是够的了，所以，一般在分配过程中仅仅通过一个membitview就可以满足绝大部分的用户需求了；而且membitview的数量是固定的值，对于有128M内存的系统来说，其数量不会超过32个，所以通常可以忽略对于membitview链表的查询访问时间，而只考虑对一个membitview的操作代价。通过对以上算法流程图的分析，可以看出：

分配过程中运行时间开销较大的有两种情况：m=0或是（m!=0）&&（n!=0）。当m=0时，表面上看这是一个两层循环。但是外层循环最多执行31次，且仅当maxnum[i] ≥n为真时才一次性进入内层循环；其余时间都只是在循环头和布尔表达式maxnum[i] ≥n判定之间反复。可见，除去内层循环执行时间之外的时间开销是常数C_1。同理，内层循环最多执行31次，且仅当Smmask∨secmap[i]==secmap[i]为真时才一次性进入复合语句。故复合语句执行时间除外的内层循环时间开销是常数C_2。重新计算maxnum[i]的值这个操作是一个单层循环，执行31次移位、加法和赋值操作等，其时间开销等价于32位整数求积的时间开销，故设定其时间开销是常数C_3。所以总的算来，在只分配大块（m=0）的情况下，分配操作在最坏的情况下的时间开销是O（C_1+C_2+C_3）=O（C）是一个常数。同样类似可以推得，当既分配大块同时也分配小块（（m!=0）&&（n!=0））的情况下，其时间开销在最差情况下也是一个常数。回收算法比较简单，时间复杂度一目了然，在最差情况下也是一个常数。由此可见，此种存储管理算法具有可预测性，最坏的时间开销不会大于特定的界限值，较适合于用在实时操作系统或对实时性要求较高的系统中。

在这里实现的是二级位示图目录法，最小的分配粒度是一页（4K），所以在页内空间还是存在一定的浪费的。要解决这个问题，可以采用更多级的目录，比如对于同样的4M内存池，如果采用三级位示图目录的话，分配粒度可以下降到128字节，如果采用四级位示图目录的话，分配粒度就可以下降到4字节，这样就可以解决内存空间浪费的问题，当然，随着级数的增多，分配粒度下降，带来的是算法复杂度的增加，运行的效率会受到影响。但不论怎样，其在最坏情况下的时间开销不会超过一个常数，这是这种算法带来的最大好处。

5.8 三种管理机制性能比较及分析

下表2中给出了以上三章中不同管理机制在分配回收不同大小的存储区域的平均

时间开销和最坏时间开销，实验的平台为：Intel Celeron 600MHz，128M 内存，硬盘：Maxtor 20GB。

表 2　三种管理机制性能对比表

存储区域大小（字节）	测试次数	分配方法	平均开销（μs）	最坏开销（μs）
4096	40000	边界	4.6	7.9
		伙伴	2.8	4.7
		位示	3.3	5.1
40960	20000	边界	25.8	45.9
		伙伴	15.9	24.3
		位示	16.1	27.7
135168	1000	边界	450.5	1027.8
		伙伴	210.4	442.4
		位示	330.6	784.3
4198400	150	边界	35400	47300
		伙伴	22600	24100
		位示	27100	29400

从表中可以看出，无论是平均开销还是最坏开销，伙伴系统的性能都是最好的，其次是二级位示图目录法，边界标识法的运行效率是最差的。这主要是因为伙伴系统采用了高效的数据结构，利用位图来掌握空闲块的信息，这样就省去了在链表中查找空闲块这个相当耗时的过程。这点正是边界标识法运行效率低的主要原因。

从具体每个分配机制内部来看，随着分配内存量的增大，消耗的时间是越来越多的，这是因为：a.小的内存块很容易就可以得到满足，而较大的内存块可能要经过较长时间的查找才能得到一个符合要求的内存区域；b.在页面映射这个步骤里显然是小块内存的映射要比大片内存的映射节省时间的多。

从表中还可以看出，前三种大小内存的分配中，消耗的时间还基本上与大小成正比例，而当分配块超过了4M字节时（比如第四种分配4198400字节），消耗的时间大大增加了，这种现象的主要原因在于分配算法中是以4M作为一个内存池来进行处理的，当需求超出了内存池的全部容量以后，就需要将需求分割成多个，然后循环几次进行分配，所以消耗的时间大幅度增加。

此外还可以看出，对于二级位示图目录算法来说，第3、4两行的时间开销相对较大，除了上面讲到的原因以外，还因为此两行所分配的数据区域的大小映射到位示图的结构上时，需要考虑的边界条件较为复杂，存储分配算法中需要考虑多种分

支情况，导致时间消耗的增加。

总的说来，伙伴系统是分配时间效率最高的一种算法，边界标识法是效率最低的一种，位示图目录法的效率界于两者之间。

但是边界标识法的实现比较简单，适用于要求不高的简单系统中；伙伴系统适用于要求较高的通用操作系统中，利用它的高效性可以满足多用户、多任务的需求，但其不能保证系统响应的确定性；多级位示图目录法虽然运行效率不如伙伴系统，但是其最差时间复杂度为常数，具有可预测性，因此适用在实时操作系统中，并且其运行的效率也是较高的。

第六章 内核缓冲区动态管理的实现机制

上述的内存分配管理机制是对于整个内存空间而言的。系统的内核空间具有其不同于用户空间的特点，上述机制并不能完全满足内核空间的需求。在现今操作系统中，内核空间往往具有以下几个特点：

1. 内核空间是某些系统变量获得分配的空间。比如说，当建立一个新的进程空间的时候需要构建进程控制块（PCB）的结构，就需要在内存中动态分配一块空间给PCB结构，并且，这个结构的使用并不是局限于某个子程序的，否则就可以作为某个子程序的局部变量而使用用户态中的堆栈空间就可以了。这样的例子还有上面讲述的那些动态管理需要的内存控制数据结构，如：manage_struct，mem_vir_struct等等。
2. 这些存储空间常常是变化频繁的，而不是象page结构数组那样，具有和物理内存容量相关的静态长度。并且一般不能事先预测出运行中各种不同的数据结构对内核缓冲区的需求，这也就决定了不能为每一种可能用到的数据结构事先建立起一块“缓冲池”，因为那样很可能会出现某些要用的缓冲池已经用尽而其它缓冲池却有大量空闲的情况。
3. 内核空间的缓冲区的分配和回收操作应该具有较高的时间效率，因为内核空间的动作频率往往是最频繁的，所以对操作的时间效率有要求。同时，内核空间的资源又是最匮乏的，如果采用按页分配的方式，对于页内空间会造成一定的浪费，使得本就有限的资源更加匮乏。
4. 内核中频繁地使用一些数据结构，这些数据结构中有相当一部分需要某些特殊的初始化（如初始化链表指针等等）而不是简单地清成全0就可以了。假如能够使数据结构在一次初始化以后，即使释放了下次也可以重用而无需初始化，那么系统内核的效率就会有很大的提高。

内核缓冲区的这些特点，就决定了前面讲述的内存管理机制不再能够满足内核空间管理的需求，必须要有特别处理的机制。

Slab算法是1994年开发出来的并首先用于Sun Microsystem Solaris 2.4操作系统的一种分配器模式[12]，现在在Linux系统上得到了广泛的应用，在深入研究了Linux系统的内核缓冲机制后，以其为基础，并对其进行了一定的改进，形成了这里具有一定面向对象特性的内核缓冲区管理机制。如果我们将内核缓冲区中的各种数据结构作为对象来看待，根据其独自的特性和用途，可以为它们建立不同的初始化和释放机制。对于不同的对象，可以为其构造不同的“构造函数”和“析构函数”及其各种对象所需的不同的操作处理过程，然后再利用一定的数据结构将这些对象组织起来，统一管理，这就是本方法的基本思想。目前，在Linux系统中还没有充分利用到这些特性。

6.1 主要数据结构

首先要介绍一下该机制所使用的一些特定的数据结构，对于内核中的各种数据结构，现在开始称其为“对象”。

1．一个block的队列

每种对象的缓冲队列并非是由各个对象直接构成的，而是由一连串的block所组成，每个block中包含了若干个同种的对象。并且为了便于分配物理内存，这个block的大小必须是页面大小的整数倍。具体大小根据不同的对象的大小而不同，在系统初始化的时候通过计算得出block合适的大小。一个block的结构如图29所示：

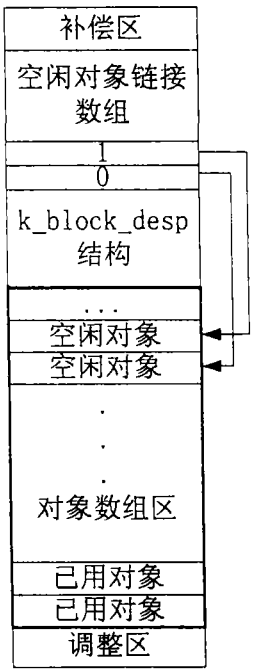

图 29 block 示意图

其中的主要组成介绍如下：

- 调整区：在每个 block 的起始处是一块有特定功能的区域，叫做调整区。调整区的大小不是固定的，其作用是使 block 中的每个对象的起始地址都和 CPU 的高速缓冲存储器 Cache 的一行缓存管线对齐。

  Cache 是位于 CPU 与主存储器之间的一种存储器。它的容量比主存储器小，但是访问速度比主存储器要快得多。Cache 中的内容是主存储器某一部分内容的副本，而这一部分是 CPU 当前正在使用的指令和数据[16]。在具有 Cache 的存储系统中，Cache 和主存储器都被机械地划分为尺寸（容量大小）相同的块，称之为“缓存管线”，每条缓存管线由若干个字节组成，并且将缓存管线有序的编号。Cache 的优势在于能够简单快速地确定某一内存区域是否被某条缓存管线命中（已被存放到 Cache 中），如果命中则直接从 Cache 中高速获取数据，

而不需要对主存进行操作。对于使用相当频繁的内核访问，有效利用 Cache 的特性，可以节省大量的访存时间。

因为 Cache 的容量远远小于主存，一条 Cache 管线要对应许多主存块，因此需要按某种规则把主存块装入 Cache 中，这就是 Cache 的地址映像。一般映像分为三种情况[16]：一种是主存中的一行只能装入到 Cache 中唯一的特定管线中，这种方式称为直接映像；另一种是主存中的任意一行可以被存放在 Cache 中的任意一条管线中，这称为全相联映像；还有一种是主存中的任意一行可以被存放在 Cache 的 N 行中的任意一行中，称为 N 路组相联映像。

直接映像方式下，相当于把主存空间按 Cache 空间划分为许多区，区中的块仅能装入 Cache 中特定的块位置上。因此地址变换的速度快且实现简单。如果对应 Cache 中的某一块的主存块有两个或两个以上需同时装入 Cache，即使 Cache 中还有许多未使用的块，这些主存块也不能装入 Cache，这就产生了“块冲突”，所以直接映像的块冲突率最高，空间利用率最低。为了进一步提高系统性能，要设法解决或减少直接映射方式下的块冲突率。

举例来说，Intel Celeron CPU 中的一级 Cache 容量为 32KB，每条缓存管线包含 32 个字节，设主存块的块号为 i，Cache 中的管线号为 j，2^n 为 Cache 中的管线数，则直接映像关系如下公式所示：j = i MOD 2^n。如果系统内存为 128MB，则每个高速缓存管线负责映射到的内存区域为 128MB/1024=128K。其直接映像示意图见图 30：

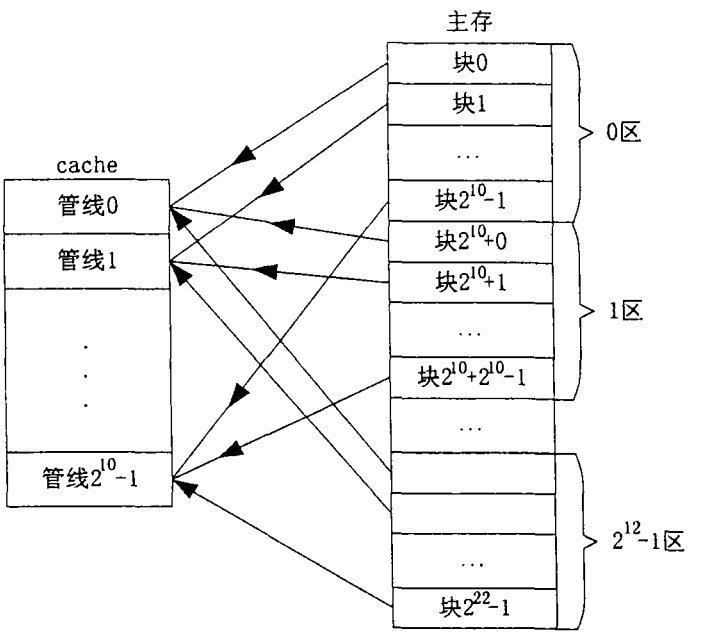

图 30 Cache 直接映像方式示意图

为了避免内核中的那些频繁使用的对象在访问时能以最快的速度装入 Cache 中，需要调整对象的起始地址，将其调整到与高速缓存管线长度对齐的最佳位

置上；同时由于在不同 block 中具有相同偏移量的对象最终很可能被映射到同一高速缓存管线中，所以要尽可能的将同一位置上的对象的起始地址相互错开，这样可以保证这些对象可以同时装入 Cache 中，最大限度的提高 Cache 的效率。这里要注意的是：每个 block 都是从一个页边界开始的，因此必然和高速缓存管线对齐，调整区的作用是将 block 中的第一个对象的起始地址向后推到一个与高速缓存管线对齐的边界上。

- 对象数组区：见图中的粗黑框部分。这个区域是 block 的主体，也就是内核缓存所保存的数据结构，它是一个数据结构对象数组。采用数组的方式组织对象，可以以对象的序号作为下标来访问数组，也就可以快速准确的获得对象的起始地址。对象数组区的大小不是统一的，其包含的对象的个数也不是固定的，这些量值的确定，要在内核缓冲队列初始化时，根据具体对象（数据结构）的大小来计算出合适的个数。并且，数组中每个对象的大小不与高速缓存管线对齐时，要在对象后增补一些字节，来保证一个 block 中的所有对象的起始地址都是和高速缓存管线对齐的。
- 空闲对象链接数组：这个数组是用来描述对象数组区中空闲的、还没有分配的对象。链接数组中每个元素都是下一个空闲对象的序号，定义 0xFFFFFFFF 作为数组结束的标记，这样可以顺次描述出一个空闲对象的队列。只要给出 block 上的第一个空闲对象，就可以依次找出全部的空闲对象，这个控制在下面的 k_block_desp 结构中有描述。
- k_block_desp 是一个 block 的描述结构，它的定义如下：

```
struct k_block_desp{
        struct k_block_desp * prev;
        struct k_block_desp * next;
        struct k_cache_head * head;
        unsigned long firstfree;
        unsigned long numofused;
        void * object_region;
};
```

其中 prev 和 next 用来将一个 block 链入一个 block 队列中，以便于统一管理；这个队列中的 block 都是用于同一种对象的；head 是指向该队列的队列头（在下面介绍）的指针。firstfree 是该 block 中第一个可用的空闲对象的序号，它实际上是对象数组的一个下标，用来指示头一个空闲对象；numofused 是 block 中已经使用了的对象的个数，当有一个空闲对象被分配时，numofused 就加 1，当有一个对象被释放时，numofused 就减 1；region 是指向对象数组区的指针，通过它找到对象数组区。

- 补偿区：这也是一块不使用的区域，其目的是使整个 block 的大小与页面大小 4K 对齐，所以其大小是取决于调整区的大小以及各种对象的大小。

为每种对象建立的 block 队列都有个队列头结构，队列头结构中除了用来维持 block 队列的各种指针外，还配有适用于队列中的各种参数。对于各种不同的内核对象，有各自不同的行为模式和处理方法，所以在这里引入了面向对象的方法，把不同对象的构造函数（Constructor）和析构函数（Destructor）的指针放在队列头中，同时增加了一个 Callback 函数指针，为不同的函数功能保留一个接口，通过这个接口，可以使不同的对象具有不同行为的能力，从而具有一定的动态特性。这个队列头结构的定义如下：

```
struct k_cache_head{
        struct k_cache_head * followprev;
        struct k_cache_head * followhext;
        struct k_block_desp * block;
        struct k_block_desp * allocableblock;
        unsigned long objsize;
        unsigned long objnum;
        unsigned long blocknum;
        unsigned long blocksize;
        unsigned long align_offset;
        unsigned short align_bit;
        unsigned short objectid;
        unsigned long align_next;
        unsigned long align_all;
        void （*Constructor） （void *, k_cache_head *, unsigned long）;
        void （*Destructor） （void *, k_cache_head *, unsigned long）;
        void （*Callback） （void *, k_cache_head *, unsigned long）;
};
```

首先要说明一下整个内核缓冲区的结构，在系统中有一个总的block队列，其中的对象都是k_cache_head数据结构，换句话说，这个总队列中存放的都是各个不同对象的缓冲队列的队列头对象，这些队列头对象分别维护各自不同对象的缓冲队列；同时系统总block队列自身也有一个队列头叫做“总根”（cache_root），它也是一个k_cache_head结构。很明显，这样就形成了一种层次式的树形结构：

- 总根cache_root用来维护第一层的block队列（总队列），这些队列上的对象都是k_cache_head结构；
- 每个第一层block上的每个对象（k_cache_head）都是队列头，分别用来维护一个第二层的block队列；
- 第二层上的block队列都是为某种对象（数据结构）所专用的，同时通过每个block上的block_desp结构来维护着一个空闲对象队列。

整体的内核缓冲区结构示意见图31所示：

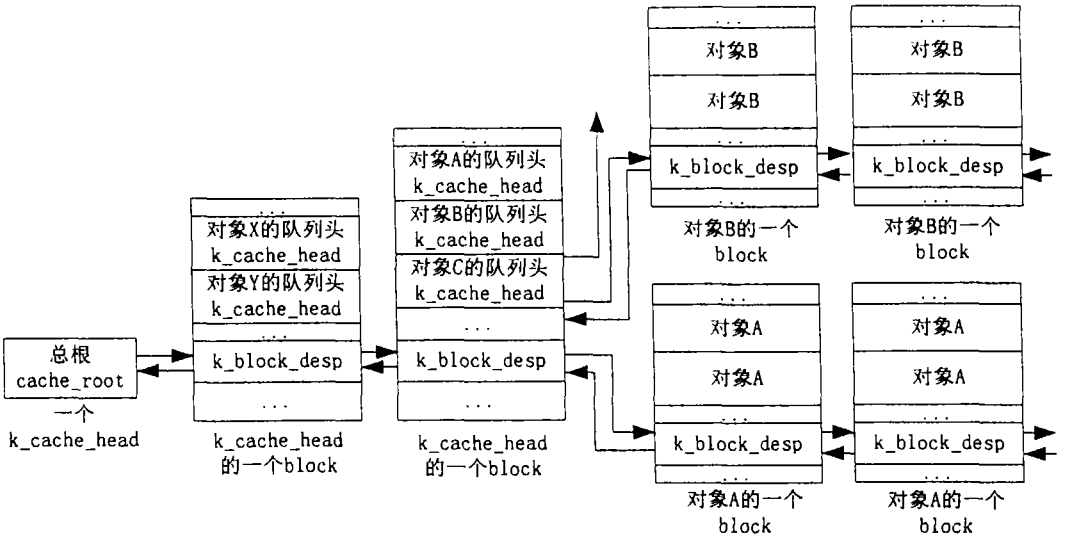

图 31　内核缓冲区内存管理结构关系图

如此，对于k_cahce_head数据结构的各项就可以有一个比较清晰的解释了：

- followprev和followprev的作用是将k_cache_head结构链接起来形成一个双向队列；
- block是指向k_block_desp结构的指针，它指向对象缓冲队列中的第一个block的k_block_desp控制结构；
- allocableblock是指向对象缓冲队列中第一个含有空闲对象的block的k_block_desp控制结构，通过它可以快速地找到可用对象空间；
- objectid是不同内核对象的标识，用它来识别不同的对象，系统规定了一系列的内核对象的标识；
- objsize表示了原始的数据结构（对象）的大小，一般是调用函数通过sizeof（ ）的形式传入的；
- objnum代表了每个block上存放对象的个数，需要注意的是：虽然不同对象所对应的缓冲队列中的block上对象个数是不一定相同的，但是对于同一种对象，其每个缓冲队列中block上存放的对象的个数是相同的，并且是根据对象的大小计算出来的；
- blocknum表示队列中block结构的数量；
- blocksize表示了一个block的大小，比如blocksize=2，则其控制的block的大小占2个页面，也就是2×4096 = 8K字节，一个block的大小必须是页面的整数倍，当然也不能过大，定义block大小最大不超过64个页面；
- align_offset、align_bit和align_all都是用于控制对象地址的变量。前面说过，为了提高Cache的效率，需要将同一block队列中不同block上对象区的起始地址相互错开，具体分析如下：考虑某个block中的对象已经与Cache的缓冲管线对齐，为了对齐而在对象后添加的字节数为align，align存放在align_offset变量中；对象的

大小为osize（已经对齐后的大小），对象的数目为num，k_block_desp结构的大小为dsize，在block中未使用的字节数是free，那么一个block的总长度就为：（num×osize）+ dsize + free，显然一个block是页对齐的，所以可调整的实际上是free的大小。free总是小于osize的，否则，就可以再添加新的对象到block中。将不同block中的对象错开的过程也常常称作“着色”过程，align_bit是一个控制量，称为“颜色”，假设align_bit中值为color，那么block中第一个对象的相对偏移量就等于color×align，这个值计算出来后保存在align_all变量中。同时“颜色”的个数是有限的，为free/align+1，所以，只有当空闲量free足够大时，着色才能起作用，如果free<align，那么这个block就只有“0”这个颜色可用了。每一个block创建所用的颜色都与前面一个不同，从0开始，一直到最大可用颜色free/align，接下来又从0开始循环。

- 接下来定义了三个函数指针，其中Constructor和Destructor指针分别代表了内核对象专用的构造函数和析构函数，当缓冲队列建立一个新block时，就使用构造函数对其对象数组区内的对象进行一定的初始化，这样，当进行实际分配时就不再需要进行初始化了；同样，当把一个block从队列中摘除时，就调用析构函数使其恢复原状。Callback提供了一个接口，可以通过加载不同的内核驱动程序的方式，使内核对象具有不同的行为。

6.2 缓冲队列建立流程

要通过缓冲区的 block 队列来分配和回收内存，首先要建立队列及其控制结构，这实际上也是一个分配过程，不同的是，这里要分配的对象是 k_cache_head 结构，这个结构自身又是参与缓冲区动态管理的结构而已。

系统规定了每个 k_cache_head 对象的 block 占 1 个页面的大小，block 中的每个 k_cache_head 对象占 60 个字节，为了与 Cache 管线对齐，所以需添补 4 字节，所以一个对象扩充到了 64 字节；空闲对象链接数组每个占 4 字节，数目和对象数目相同，一个 k_block_head 结构占 24 个字节，由此可计算出一页中共存放了（4096－24）/（64+4）= 59 个 k_cache_head 对象，每个 block 的空闲区域为 60 字节，可用的“颜色”数为 60/2＋1＝31 种。确定了这些参数以后，再申请 1 页的物理内存作为 block 的载体，将这些信息写入到这块内存的特定的地址上，这样，一个总队列中的 block 就可以很快构造出来了。

于是，当有需求要建立一个新的对象的内核缓冲队列时，就首先查看系统总队列中有无可用的 k_cache_head 结构，如果有就将它分配，并将其内部成员 block 指针设为空，等待缓冲区分配时建立其 block 队列；如果没有可用的 k_cache_head 结构，就再请求一页内存空间构造一个存放 k_cache_head 对象的 block，并将这个 block 链接入系统总队列中。

对象专用缓冲队列的建立的大体流程如图32所示：

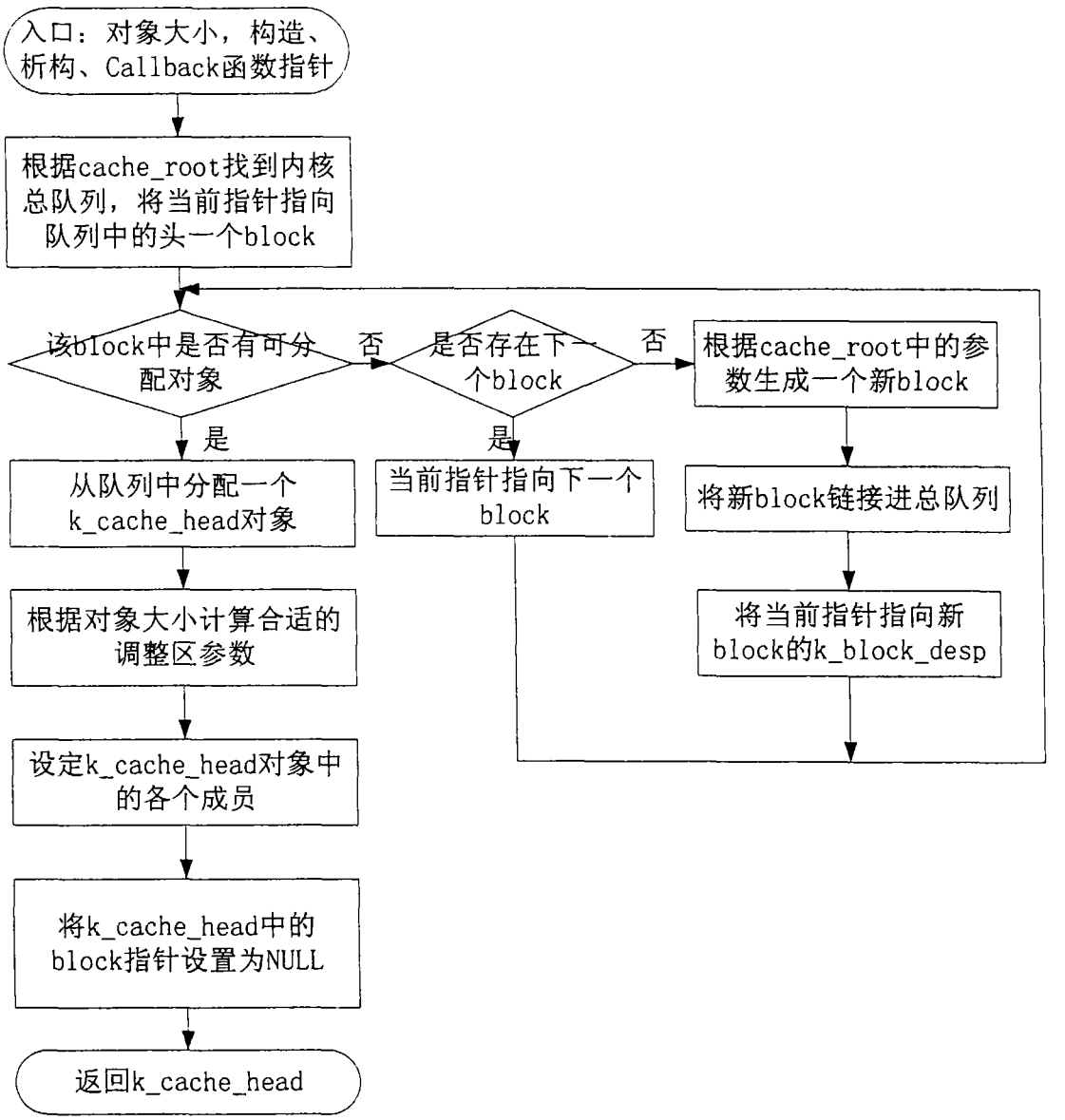

图 32　对象专用缓冲队列建立流程图

6.3 缓冲区的分配算法流程

当用户有请求申请一个内核对象时，系统首先要检查总队列中是否已经建立了该对象的缓冲队列，如果已经建立，就返回该队列的k_cache_head指针；如果没有，就要先调用缓冲队列建立函数建立这个队列，然后返回其k_cache_head指针。接着再调用缓冲区分配函数为其分配实际对象，所以，缓冲区函数的入口参数是一个队列头指针，其主要负责返回一个可用的、用户所要种类的对象的地址。

内核缓冲区分配算法流程见图33

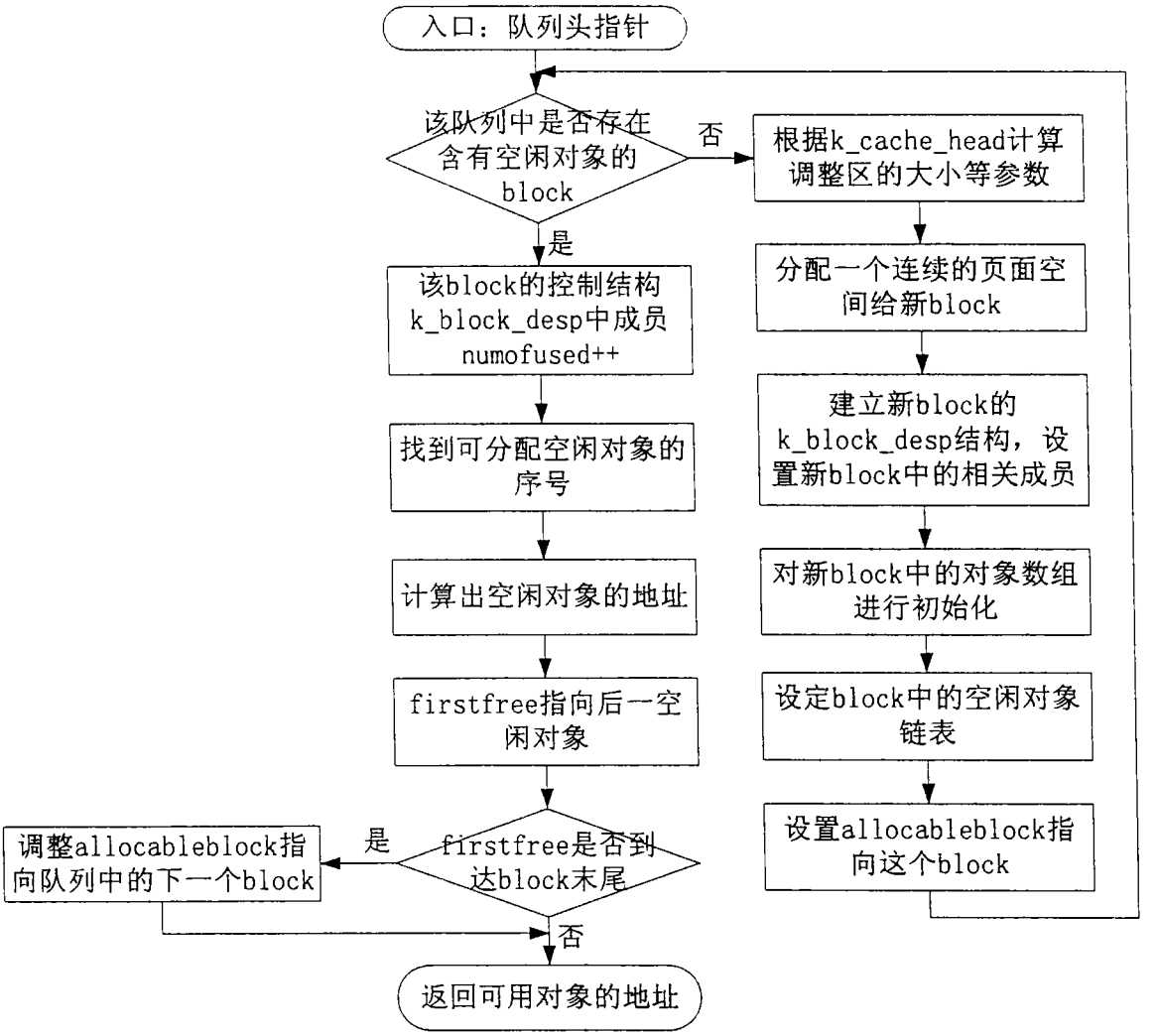

图 33　内核缓冲区分配流程图

在分配流程中要注意：

- 首先队列头中的allocableblock指针如果不为空时，说明队列中有空闲的对象可用于分配，并且该指针指向含有空闲对象的block的k_block_desp结构；
- 在k_block_desp中的firstfree就是下一个可用的空闲对象在该block的对象数组中的序号，所以空闲对象的地址objectaddress就为：
  object_region+firstfree×(k_cache_head－>objsize＋k_cache_head－> align_offset)
- 然后需要将firstfree指向空闲对象链表中的下一个对象，以便于下一次分配时使用，如果firstfree＝0xFFFFFFFF，表示该block中已经没有空闲可用对象了，这时要将队列头k_cache_head中的allocableblock指向队列中的下一个block的k_cache_desp结构，这可以通过上一个k_cache_desp的双向指针找到。
- 在分配页面空间这个过程中，可以使用上面讲述过的三种分配机制，设置好页目录表和页表及其页面的权限位、属性位等，实现映射；同时，页面的page结

构中的指针prev原来用于指向前一个页面形成双向链表，现在将这个指针指向本页面所属的block。

- 在对block的初始化中包括了对具体构造对象函数的调用，在这里就已经对对象中的一些成员进行了设置，比如双向链表的指针等；
- 在生成了一个新的block后，要重新设置队列头中的allocableblock指向这个新生成的block，同时程序跳回到开始处，这时，已经有可用空闲对象可供分配了。

6.4 缓冲区回收算法流程

内核缓冲区回收流程见图34：

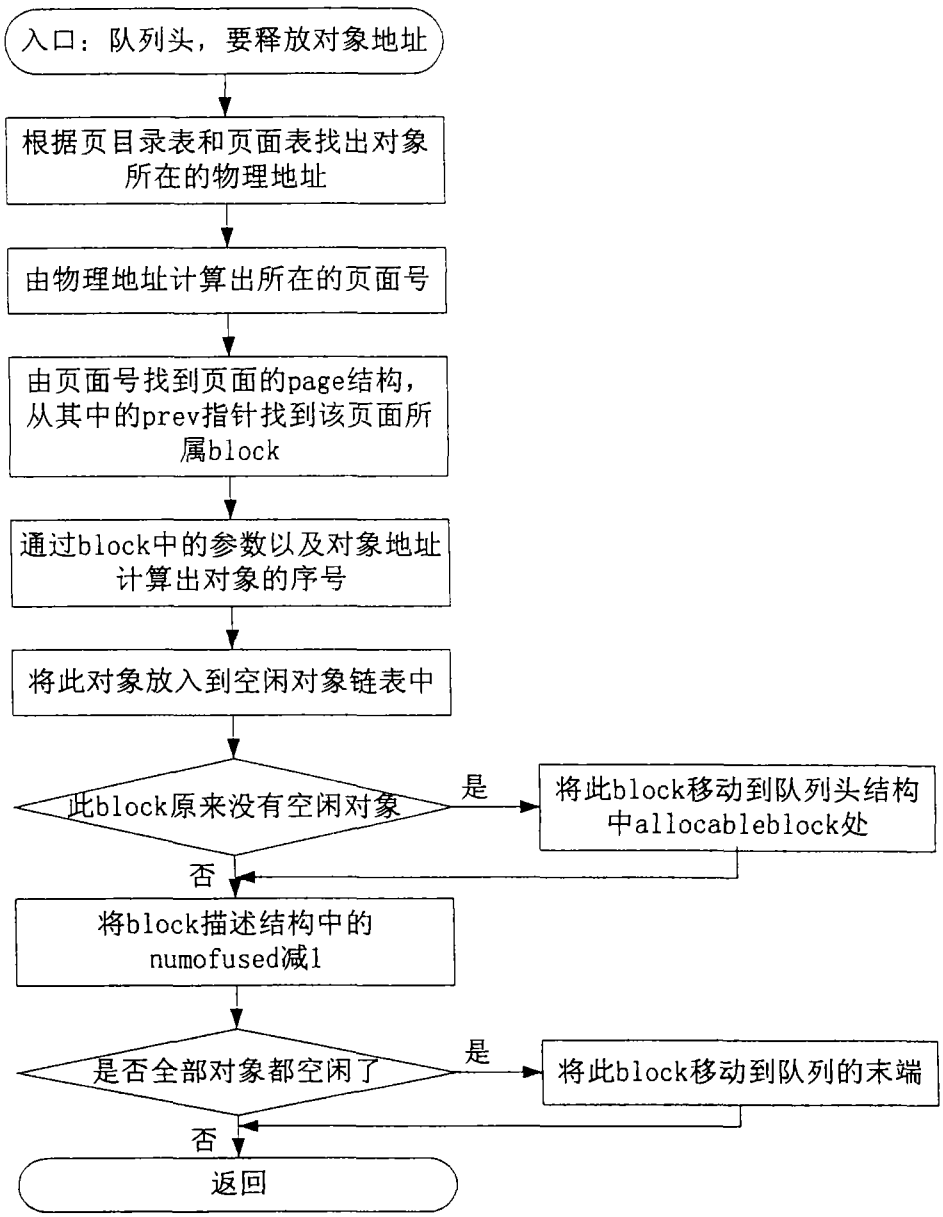

图 34 内核缓冲区回收流程图

关于回收流程的一些说明：

- 从回收流程中可以看出，回收时并不释放内存空间，那么随着分配过程中不断的申请内存，缓冲区队列是单调增长而不缩短的，这样下去内核空间会迅速的消耗光，所以系统通过增加后台进程k_recyle定期的回收内核缓冲空间。k_recyle定期检查若干个专用缓冲的block队列，看看是否有已经完全空闲的block存在，如果有，则将这个block从队列中摘除，并释放其占用的内存页面。在内核中有全局变量记录了k_recyle上次检查到的位置，下一次k_recyle接着这个位置继续检查缓冲队列。
- 为了提高内核缓冲的效率，一个对象的缓冲队列中block的排列是有一定的顺序的：在队列头开始处的一段若干个block中的对象是已经被全部分配出去了的，这段block称为满载区；接下来的一段block中只有一些对象已经分配出去了，但是并没有全部分配出去的，称为半载区，队列头中的allocableblock指针就是指向这个区域的第一个block的；剩下的称为空闲区，其中block中的对象一个都没有分配出去，每次释放对象的过程中如果得到了一个完全没有对象被分配的block，就将其链接到队列的最后，也就是空闲区的最后。
- page结构中的prev指针的意义对于内核缓冲页面中和一般页面是不同的，一般页面中prev指针和next指针互相配合使page形成一个双向链表；而正如前面所说的，对于内核缓冲队列来说，prev指针是用来指向这个页面被分配给的block中的k_cache_desp结构的。这两个指针的类型是不同的，所以在实际编程时要有个强制转型的过程。
- 根据队列头k_cache_head中的对象大小、调整区参数和一个block的k_cache_desp中的对象数组区起始地址，可以确定block中每个对象的起始地址，所以与对象地址比较即可得到此对象在对象数组中的序号。

6.5 算法分析及讨论

从上述的算法描述可以看出，无论是缓冲区的分配还是回收算法，其时间复杂度都是比较低的。在分配时，除非一个 block 中的对象已经全部分配出去了，这时需要重新构造一个新的 block，这个操作要耗费一些时间以外，在一个 block 中的对象的分配过程即没有循环也没有复杂的运算，所以时间复杂度较低；在回收对象时也没有什么复杂的动作，而且由于回收只是将对象置为空闲状态，并不真正从内存中清除（清除的动作是交给后台进程定期进行的），所以也具有较低的时间复杂度。

实际上，内核缓冲区的管理机制是建立在前述的内存动态分配回收机制的基础之上的，每次构造 block 的时候，还是要调用前述的内存分配机制为其分配实际物理空间。

采用这种方式实现的目的在于：

1. 降低内存的分配粒度，减少物理内存的浪费；
2. 有效的利用了 Cache 机制，加快内核对象的存取速度；
3. 引入了内核对象的一些特定相关的初始化动作，在释放对象时并不清除对象，这样下次再分配该对象时就可以省去了一些初始化动作，同样加快了操纵内核对象的速度。

第七章 结论及今后改进方向

本文深入研究了操作系统动态内存分配和回收机制的相关理论及其具体实现过程，既在理论上进行了一些探讨，也在实现上有所改进。

文章首先从实现具体的系统入手，介绍了与内存管理相关的一些重要理论和概念，接着介绍了实现过程中必要的抽象手段。为了使多个分配回收策略可以运行在系统上，在数据结构上添加了一层抽象单元 manage_struct，使得对于不同的管理策略都可以适用。

对传统的边界标识法进行了改进，将控制信息从空闲块中移动到了空闲块外，同时添加了搜索指针等元素，针对特定的数据结构，详细的介绍了分配和回收的流程，并对算法进行了分析。

在对 Linux 中使用的伙伴系统进行了深入的研究后，提出了对伙伴系统的改进方案，扩充了系统中队列的规模，修改了数据结构，并针对自身特定的系统实现，去除了一些繁琐复杂而又不经常使用的部分细节，简化了程序流程，使其更具运行效率。

同时文章研究了一种基于多级位示图目录结构的内存管理机制，并具体实现了一个二级位示图目录算法的管理机制，通过这种算法，可以用较少的效率降低的代价，获得系统响应的可预测性的保证。

通过对于这几种内存动态管理机制的研究，测试，经过分析比较，最终得出结论，这三种方案各有个的优势，但又有各自的缺点：

边界标识法是一种较经典的理论，实现起来相对容易些，但是效率是较差的，已不能适用于现代操作系统的复杂性，它较适合简单的操作系统中；

伙伴系统运行效率在本文实现的三种机制中最好，但是其实现起来要比边界标识法复杂些，并且存在较严重的内碎片的问题，适用于一般的通用操作系统中。

多级位示图目录法的性能比起伙伴系统稍微差些，但是其具有在最差情况下时间复杂度也为一个常数的优点，具有可预测性，所以其实时性能得到确定的保证，较适合于实时操作系统中。

在文章的第六章中，对内核缓冲的管理进行了研究。在 slab 算法的基础上，对内核缓冲的管理模式进行了改进，引入了面向对象的机制，同时通过补位着色等措施，使得 Cache 能够发挥出最高的效率。如果将伙伴系统和 slab 机制结合起来，就能获得最好的内核缓冲区性能。

今后的改进方向应当是三个：

1.将 slab 机制扩充到整个内存管理中，而不再仅仅局限于内核缓冲区管理，这样做可以大大减少内存浪费现象，提高内存的利用率，当然，随着算法的复杂化，

不可避免的会带来时间复杂度的上升，所以同时要进一步对数据结构、算法进行研究，应该把时间的损失降低到最小。

2.对于多级位示图目录法的改进和提高。由于现今嵌入式应用越来越广泛，且多为实时系统，所以实时性能较佳的多级位示图目录法应该有着广阔的应用天地。本文中仅仅实现了最简单的二级位示图目录法，通过将级数增加，可以控制更大的内存区域，并且可以降低分配粒度，减少系统资源的浪费。

3.改进操作系统结构，使得对于一个进程能够动态选择不同的存储管理机制，这样就可以针对不同特点的应用，采用相应的效率最高的管理机制，使系统性能大大提高。

当然，一个操作系统的好坏，并不仅仅取决于内存管理系统，还和很多方面相关，比如进程调度，进程通讯机制，文件系统，磁盘交换机制等等，是一个相当巨大、复杂的综合体，本文只是针对了其中的一个小方面进行了研究和探讨，也是研究操作系统，改进操作系统的一个 有益的尝试。

致　谢

首先我要衷心地感谢我的导师顾宝根教授，研究生学习期间，他不论是指导我做学问，做项目，还是在指导我撰写论文等多方面都给予了我极大的帮助。他渊博的知识，深厚的学术造诣，严谨的治学态度使我在多方面获益匪浅。对于顾宝根老师所给予的指导与教诲、关心与鼓励作者将铭记在心。

我还要向支持我的学业的父母致以最深切的感谢，他们多年来为我默默的付出，从精神上的鼓励到生活上的细心关怀，没有他们的鞭策就没有我今天的成绩，他们的支持永远是我前进的动力。

最后还要对所有在这两年半中给过我帮助和鼓励的老师和朋友们致以最诚挚的谢意。

在学期间研究成果

[1] 参与编写《操作系统实验教程——核心技术与编程实例》一书，2003.3 科学出版社出版，约占 3 万字

[2] 天象仪系统的设计与实现，微机发展，2003.1

参考文献

[1] Andrew S. Tanebaum, Albert.S. Woodhull. Operating Systems: Design and Implementation (in Chinese). 第 2 版. 北京: 清华大学出版社, 1997

[2] 李善平，郑扣根. Linux 操作系统及实验教程. 北京：机械工业出版社，2000

[3] M. Rosenblum, E. Bugnion, S. A. Herrod, E.Witchel, and A. Gupta. The Impact of Architectural Trends on Operating System Performance. In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, pages 285~298. ACM, 1995.12

[4] 彭晓明，王强. Linux 核心源代码分析. 北京：人民邮电出版社，2000

[5] L. Hosking and J. E. B. Moss. Protection Traps and Alternatives for Memory Management of an Object Oriented Language. In Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, pages 106~119. ACM, 1993.12

[6] David A.Rusling 等著， 朱珂，涂二舰等译. Linux 编程白皮书. 北京：机械工业出版社，2000

[7] William S. Operating System internals and design princinples (第 4 版). 北京：电子工业出版社，2001

[8] David G. Korn and Kiem-Phong Vo. In search of a better malloc. In USENIX Summer 1985, pages 189-506, Portland, Oregon, 1985.7

[9] 汤子瀛，哲凤屏，汤小丹. 计算机操作系统. 西安：西安电子科技大学出版社，1999

[10] Daniel P. Bovet, Marco Cesati 著，陈莉君，冯锐，牛欣源译. 深入理解 Linux 内核. 北京：中国电力出版社，2001

[11] K. Harty and D. R. Cheriton. Application-Controlled Physical Memory using External Page-Cache Management. In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 187~197. ACM, 1992.

[12] 毛德操，胡希明. Linux 内核源代码情景分析. 杭州：浙江大学出版社，2001

[13] 郭顺福，王世铀，臧天仪.一种动态存储管理机制. 计算机研究与发展，1999，36（1）：62~66

[14] 尤晋元，史美林等. Windows操作系统原理. 北京：机械工业出版社，2001

[15] T. H. Romer, D. Lee, B. N. Bershad, and J. B. Chen. Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware. In Proceedings of the First USENIX Symposium on Operating System Design and Implementation, pages 255~266. 1994.11

[16] 仇玉章. 32位微型计算机原理与接口技术. 北京：清华大学出版社，2000

[17] 汤子瀛，杨成忠. 计算机操作系统.西安：西安电子科技大学出版社，1984

[18] B. N. Bershad, J. B. Chen, D. Lee, and T. H. Romer. Avoiding Cache Misses Dynamically in Large Direct-Mapped Caches. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 158~170. ACM, 1994.10

[19] 严蔚敏，吴伟民. 数据结构（C语言版）. 北京：清华大学出版社，1992

[20] 封斌，龚灼，杨学军. 实时操作系统保护模式下的内存管理策略. 华中科技大学学报（自然科学版），2002，30（3）：94~96

[21] 刘福岩，尤晋元. 存储系统的层次性与进程数据存储模型.计算机研究与发展，2000，37（11）：1367~1373

[22] 潘立登，李婷. 有效实现内存管理的方法. 北京化工大学学报，2000，27（3）：98~102

[23] 李江，常葆林. 嵌入式操作系统设计中的若干问题. 微型机与应用，2000，8：13~14

[24] 王世铀，郭福顺，臧天仪. 微内核操作系统的结构对性能的影响. 计算机研究与发展，1999，36（1）：57~61

[25] 杨季文. 80X86 汇编语言程序设计教程. 北京：清华大学出版社，1998