

电子科技大学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

工程硕士学位论文

ENGINEERING MASTER DISSERTATION

论文题目

Linux 内存管理分析与研究

工程领域

软件工程

指导教师

李毅 教授

作者姓名

姜力波

学 号

200892333046

分类号_____ 密级_____

UDC_____

学 位 论 文

Linux 内存管理分析与研究

(题名和副题名)

姜 力 波

(作者姓名)

指导教师姓名_____ 李 毅 _____ 教授

电子科技大学 _____ 成都

姜 俊 文 _____ 高工

成都锐维广告有限公司 _____ 成都

申请学位级别_____ 硕士 _____ 专业名称_____ 软件工程 _____

论文提交日期_____ 2011.5 _____ 论文答辩日期_____ 2011.6 _____

学位授予单位和日期_____ 电子科技大学 _____

答辩委员会主席_____

评阅人_____

年 月 日

独 创 性 声 明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

签名：姜 力 波

日期：2011 年 6 月 1 日

关于论文使用授权的说明

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

签名：姜 力 波

导师签名：李毅

日期：2011 年 6 月 1 日

摘 要

内存管理系统是操作系统中最为重要的部分，因为系统的物理内存总是少于系统所需要的内存数量。为发挥内存的最大作用，各种操作系统采用了不同的管理策略。在 Linux 操作系统中采用分页式的内存管理方式，而它的内存管理算法采用的是经典的伙伴算法。即：把所有的空闲页面分为 10 个块组，每组中块的大小是 2 的幂次方个页面，例如，第 0 组中块的大小都为 2^0 （1 个页面），第 1 组中块的大小都为 2^1 （2 个页面），第 9 组中块的大小都为 2^9 （512 个页面）。也就是说，每一组中块的大小是相同的，且这同样大小的块形成一个链表。

但伙伴算法合并要求太过严格，只允许两个块大小相同，地址连续并且同属于一个大块的伙伴才能进行合并。伙伴算法还容易产生碎片，当一个连续的内存中仅仅一个页面被占用，这将导致这整个内存区都不具备合并的条件。伙伴算法涉及了比较多的计算还有链表和位图的操作，开销还是比较大的，如果每次 2^n 大小的伙伴块就会合并到 $2^{(n+1)}$ 的链表队列中，那么 2^n 大小链表中的块就会因为合并操作而减少，但系统随后立即有可能又有对该大小块的需求，为此必须再从 $2^{(n+1)}$ 大小的链表中拆分，这样的合并又立即拆分的过程是无效率的。

本文针对伙伴算法这一缺陷，设计一种算法，放宽伙伴关系限制，使两个块大小相同，地址连续但不属于一个大块的空闲空间形成伙伴。对伙伴算法的数据结构进行扩展，同时修改原算法中的分配和释放函数。

实验表明，经过修改的伙伴算法在一定程度上提高了内存的利用效率，使 Linux 操作系统具有更大的适用性。

关键词：Linux，内存管理，伙伴系统

Abstract

Operating system memory management system is the most important part, because the system's physical memory is always less than the amount of memory required for the system. To play the biggest role of memory, a variety of operating systems using different management strategies. Used in the Linux operating system paged memory management, and its memory management algorithms used in the classic buddy. Ie: all the free page is divided into 10 blocks each group in the block size is a power of 2 pages, for example, group 0 to 2^0 block size are (1 page), Group 1 both in the size of the block 2^1 (2 pages), 9 blocks in the size of the group are 2^9 (512 pages). That is, the size of blocks in each group is the same, and this same size to form a linked list of blocks.

But the buddy system consolidation requirements too stringent, allowing only two blocks of the same size, contiguous and belong to a large merger partner. The buddy system is also prone to fragmentation, when a continuous memory occupied only a page, which will result in the whole memory area do not have the merger conditions. Partners algorithm involves a lot more computing also list and bitmap operations, the overhead is quite large, if the size of each partner 2^n will be merged into block $2^{(n+1)}$ in the linked list queue, then the size of the list 2^n The block will be reduced because of the merge operation, but the system then may have an immediate demand for the size of the block, this must be from the $2^{(n+1)}$ split the size of the list, such a merger and now demolition sub-process is inefficient.

In this thesis, the buddy system this deficiency, an algorithm designed to relax the restrictions partnership, so that the two blocks of the same size, but do not belong to a contiguous chunk of free space for the formation of partnerships. Algorithm for partners to extend the data structure, while the original algorithm to modify the allocation and release functions.

Experimental results show that modified buddy been to a certain extent, improve the efficiency in the use of memory, so Linux operating system has greater applicability.

Keywords: Linux, memory management, Buddy system

目 录

第一章 绪 论	1
1.1 课题开发背景	1
1.2 国内外发展情况	1
1.3 应用前景	2
1.4 问题的提出	2
1.5 本文的主要工作	2
1.6 章节安排	3
第二章 存储系统结构简述	4
2.1 三级存储体系结构	4
2.1.1 主存储器.....	5
2.1.2 外存储器.....	6
2.1.3 高速缓冲存储器.....	7
2.2 虚拟存储器	12
2.2.1 页式虚拟存储器.....	13
2.2.2 段式虚拟存储器.....	14
2.2.3 段页式虚拟存储器.....	15
2.2.3 快表.....	16
2.3 本章小结	16
第三章 LINUX 操作系统的内存管理模式	17
3.1 LINUX 简介.....	17
3.2 LINUX 内核主要特点.....	18
3.3 LINUX 内存管理器.....	21
3.3.1 地址映射.....	21

3.3.2 虚拟地址管理.....	21
3.3.3 物理内存管理.....	22
3.3.4 建立地址映射.....	23
3.3.5 内核空间管理.....	23
3.3.6 用户空间内存管理.....	24
3.3.7 用户的栈.....	25
3.4 本章小结	25
第四章 LINUX 伙伴系统及改进方案.....	26
4.1 LINUX 伙伴系统.....	26
4.2 代码分析	27
4.2.1 有关的数据结构.....	29
4.2.2 alloc_pages()/alloc_page()实现分析.....	32
4.2.3 free_pages()的相关实现.....	41
4.3 伙伴算法的不足	46
4.4 LINUX 伙伴算法的改进.....	46
4.4.1 主要数据结构的改进.....	46
4.4.2 内存分配函数的改进.....	51
4.4.3 内存释放函数的改进.....	58
4.5 本章小结	60
第五章 算法效率分析	62
5.1 算法有效性分析	62
5.2 改进算法的测试和分析	62
5.3 本章小结	65
第六章 结论	67
致谢辞	68
参考文献	69

第一章 绪 论

1.1 课题开发背景

Linux 内核始于 1991 年，是由当时还是芬兰赫尔辛基大学学生的 Linus Torvalds 为他的 Intel 80386 开发的一个类 Unix 的操作系统。随后 Linux 1.0 的官方版发行于 1994 年 3 月，包含了 386 的官方支持，但仅支持单 CPU 系统。1995 年 3 月，Linux 1.2 发行，它是第一个支持多平台的官方内核版本。1996 年 6 月，Linux 2.0 版本开始发行，它是第一个支持 SMP 体系结构的内核版本。Linux 2.2 版于 1999 年 1 月发行，它使得 SMP 系统上的性能得到极大提升，同时也支持更多的硬件设备。2001 年 1 月，Linux 2.4 发布，它进一步扩展 SMP 系统，同时它也开始支持桌面系统。2003 年年底 Linux 2.6 版开始发布，对于企业服务器和嵌入式系统来说，这是一个巨大的进步。对高端的机器来说，新特性针对的是性能改进、吞吐率、可扩展性，以及对 SMP 机器 NUMA 的支持。在文件系统、网络、进程调度、系统安全、内存管理、设备驱动等方面进行了大幅度改进。

从产品角度来说，计算机操作系统可分为三类：桌面操作系统、服务器操作系统和嵌入式操作系统。大名鼎鼎的微软公司只是在桌面系统方面暂时占据上风，而在服务器系统方面，Linux，Windows NT 和 Unix 三家瓜分了市场的大部分份额。现在 Linux 服务器操作系统已经非常成熟，完全可以替代 NT 和 Unix。随着 Linux 内核的不断成熟发展，为了增进 Linux 做为服务器操作系统的性能，Linux 内核在后续版本中使用了许多新技术来改进系统内核，使得 Linux 更符合企业服务器的角色，也巩固了它在服务器操作系统市场的位置。在嵌入式市场上，Linux 正在与 Windows CE 展开激烈的市场份额争夺战，并且已经略显优势，在亚洲地区，Linux 已经成为使用最普遍的嵌入式操作系统。

1.2 国内外发展情况

作为最重要的计算机系统资源，内存的分配和释放策略对于系统的运行效率发挥着举足轻重的作用。系统内核和所有进程都是通过共享有限的物理内存来运行，一个系统的稳定性与高效性往往取决于它内存管理机制，因此，一个高效的内存

管理系统不仅要能够有效地管理系统内存，保证内存分配和回收的公平性，减少频繁回收和分配内存而产生的内存碎片，还要依靠回收和分配的速度来提高系统的运行效率。

计算机操作系统的存储管理机制可以分为动态存储管理和静态存储管理两种。其中动态存储管理理论从提出到现在，被认为是永远都不可能彻底解决的问题。动态存储管理机制的设计目标是尽量减少空间资源的浪费，减少释放和申请过程的时间消耗。理想的动态存储分配器应该是在时耗极少，不浪费空间的情况下，能满足任何次序的存储空间释放和申请。然而，理想的分配器是很难甚至是不可能实现的。现实中增加空间利用率与减少时间消耗往往相互矛盾。设计动态存储管理机制只能根据特定的环境来作出适当的决策。

1.3 应用前景

Linux 内核在进程调度设计上采用多策略与机制相结合的灵活设计思想，使得针对不同的应用可以很方便、很灵活的采用不同的调度机制和调度模型来满足调度性能，因此在内存管理上进行性能优化是提高 Linux 操作系统性能的重要途径。作为一种类 Unix 操作系统主要强调系统整体效率和稳定性。因此在一定程度上势必会对于资源的使用效率上采取折中的策略。内存是计算机中非常重要资源，内存管理实现的优劣对系统性能有决定性的作用。

随着计算机技术的发展，计算机软件的规模越来越大。计算机系统中运行的进程越来越多、内存资源有限，对于内存的利用率也提出了更高的要求。因此能够提高内存的利用率对于整个系统的健壮性和稳定性起到了关键的作用。有效地提高 Linux 的内存利用率可以使得 Linux 在更广泛的环境中得到使用。

1.4 问题的提出

在 Linux 操作系统中采用伙伴算法来管理内存。伙伴算法是一种经典的内存管理算法，它严格要求伙伴内存块地址连续，且出自同一个更大的内存块，而且伙伴块必须大小相等，这就使得有些连续的内存因为不符合伙伴算法的要求却无法合并加以利用，尽管它是足够大并且连续的内存，这将对整个系统的内存利用率造成很大的影响。

1.5 本文的主要工作

本课题打算设计出一种更合理的更高效的关于内存管理的方式，使 Linux 更适合多进程系统，现代大内存机器的应用。首先介绍存储系统结构和存储设备的概念，然后简单了解 Linux 内存管理方式，着重分析伙伴系统的相关数据结构和函数代码，针对伙伴系统中存在的缺陷，添加一种新的类伙伴关系来对只满足大小相等地址连续的内存块进行分配，从而在原伙伴算法无法进行分配内存时工作，查找这样满足类伙伴关系内存块对进程进行内存分配，并对该算法进行测试。最后给出结论。

1.6 章节安排

本论文的组织结构如下：

第一章综述本课题的背景和意义，国内外内存管理技术的研究现状和需要解决的问题。

第二章简要介绍内存管理模型的工作方式，了解内存管理器是如何对于内存进行高效、快捷地管理。

第三章通过对于 Linux 内存管理器进行分析，了解 Linux 操作系统对于内存管理的方式，从整体上对于 Linux 操作系统的内存管理进行分析。

第四章对于 Linux 内存管理中使用的伙伴算法进行细致地剖析，了解伙伴算法的工作方式。在此基础上对于伙伴算法的自身不足提出解决方案。

第五章对于改进后的算法进行性能分析、验证改进算法的可行性。

第六章给出课题研究的结论，并提出了对于 Linux 内存管理技术研究的展望。

第二章 存储系统结构简述

一个高性能的计算机系统要求存储容量大，用户可使用的编程空间大，存取速度快，成本低廉，存储器能支持复杂系统结构。这些要求往往是相互矛盾的，彼此形成制约。因此在一个计算机系统当中，常采取几种不同的存储器，构成多级存储体系，以适应不同层次的需要。并可采取虚拟存储技术，使用户获得更大，更方便的编程空间。

2.1 三级存储体系结构

如图 2-1 所示是一个典型的三级存储体系结构，分为高速缓存、主存（内存）和外存三个层次，即在这个体系中，计算机有若干兆快速但昂贵且易失性的高速缓存（cache）；数千兆速度与价格适中的且同样易失性的内存；以及大容量的低速，廉价，非易失性的磁盘存储；另外还有诸如 DVD 和 USB 等可移动的存储装置。高速缓存和主存能够直接被 CPU 访问，而外存中的程序和数据只能先调入内存才能被 CPU 访问。

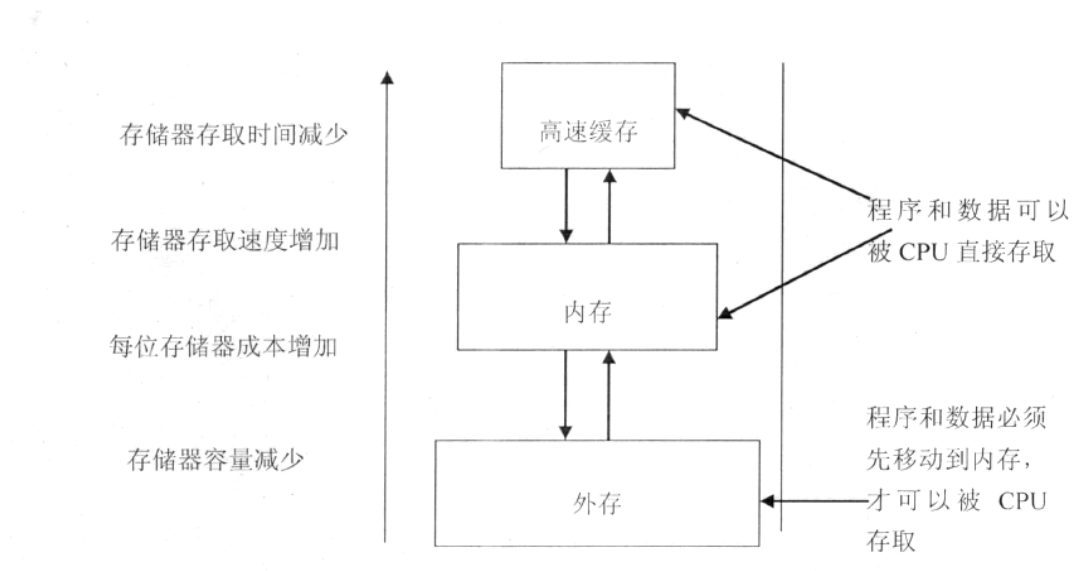


图 2-1 存储器层次结构

把存储器分为几个层次主要基于下述原因：

第一，合理解决速度与成本的矛盾，以得到较高的性能价格比。半导体存储器速度快，但价格高，容量不宜做得很大，因此仅用作与 CPU 频繁交流信息的内存存储器。磁盘存储器价格较便宜，可以把容量做得很大，但存取速度较慢，因此用作存取次数较少，且需存放原始数据（许多程序和数据是暂时不参加运算的）、大量程序和运行结果的外存储器。计算机在执行某项任务时，仅将与此有关的原始数据和程序从磁盘上调入容量较小的内存，通过 CPU 与内存进行高速的数据处理，然后将最终结果通过内存再写入磁盘。这样的配置价格适中，综合存取速度则较快。

为解决高速的 CPU 与速度相对较慢的主存的矛盾，还可使用高速缓存。它采用速度很快、价格更高的半导体静态存储器，甚至与微处理器做在一起，存放当前使用最频繁的指令和数据。当 CPU 从内存中读取指令与数据时，将同时访问高速缓存与主存。如果所需内容在高速缓存中，就能立即获取；如没有，再从主存中读取。高速缓存中的内容是根据实际情况及时更换的。这样，通过增加少量成本即可获得很高的速度。

使用磁盘作为外存，不仅价格便宜，可以把存储容量做得很大，而且在断电时它所存放的信息也不丢失，可以长久保存，且复制、携带都很方便。

2.1.1 主存储器

主存储器是能由 CPU 直接编程访问的存储器，它存放着需要执行的程序与需要处理的数据。因为它位于通常所谓的主机范畴之内，所以常常也称为内存。

从 70 年代起，主存储器已逐步采用大规模集成电路构成。用得最普遍的也是最经济的动态随机存储器芯片（DRAM）。1995 年集成度为 64Mb（可存储 400 万个汉字）的 DRAM 芯片已经开始商业性生产，16Mb DRAM 芯片已成为市场主流产品。DRAM 芯片的存取速度适中，一般为 50–70ns。有一些改进型的 DRAM，如 EDO DRAM（即扩充数据输出的 DRAM），其性能可较普通 DRAM 提高 10%以上，又如 SDRAM（即同步 DRAM），其性能又可较 EDO DRAM 提高 10%左右。1998 年 SDRAM 的后继产品为 SDRAM II（或称 DDR，即双倍数据速率）的品种已上市。在追求速度和可靠性的场合，通常采用价格较贵的静态随机存储器芯片（SRAM），其存取速度可以达到了 1–15ns。无论主存采用 DRAM 还是 SRAM 芯片构成，在断电时存储的信息都会“丢失”，因此计算机设计者应考虑发生这种情况时，设法维持若干毫秒的供电以保

存主存中的重要信息，以便供电恢复时计算机能恢复正常运行。鉴于上述情况，在某些应用中主存中存储重要而相对固定的程序和数据的部分采用“非易失性”存储器芯片（如 EPROM，快闪存储芯片等）构成；对于完全固定的程序，数据区域甚至采用只读存储器（ROM）芯片构成；主存的这些部分就不怕暂时供电中断，还可以防止病毒侵入。

为满足 CPU 直接编程访问的需要，对主存储器的基本要求是：

第一，采取随机存取的方式。可以按照地址随机地访问任一存储单元，这一点使得 CPU 可以按字节或字存取数据，并加以处理。访问任一存储单元所需要的读写时间相同，与地址无关，因此可以用存取周期表明其工作速度。

第二，工作速度快。尽管存储芯片的存取周期能够达到毫微秒级，但仍然远不能达到 CPU 的需要，为使存储系统的整体速度能与 CPU 匹配，一般采取两种措施：在主存与 CPU 之间增加一级高速缓冲存储器或是采用多存储体交叉访问方式的存储体系结构。

第三，具有较大的存储容量。程序和数据进入主存后才能运行，如果主存容量不足，CPU 将很难有效的运行大程序，因为过分频繁地在主存与外存之间调入调出会大幅度增加系统开销，使运行效率下降。所以主存的容量是影响系统性能的一个重要指标。

2.1.2. 外存储器

由于主存容量受地址位数、成本、速度等指标的制约，其容量指标一般只能满足当前需要执行的程序与数据的需要，所以在大多数计算机系统中都设置有一级大容量存储器，如硬盘、U 盘、光盘等，以此作为对主存的补充和后援。它位于传统主机范畴之外，常被称为外存储器，简称外存。

外存储器用来存放需要联机保存但暂时不使用的程序和数据。计算机系统常能提供丰富的软件资源，如操作系统、多种语言的编译程序、编辑程序、调试环境等，但某个用户可能只需要其中的一部分，如某语言的编译程序，或在工作的某一阶段暂时只使用其中的一部分。所以我们将各种软件资源存入硬盘，需要使用哪个文件时，再将它调入主存使用。运算处理结果，修改后或新形成的文件，可写入外存储器中保留。

外存储器本身可以是多台彼此独立的存储器，直接作为主存的后援。也可以分级构成，例如将调用频繁的信息保存于硬盘存储器中，作为主存的直接后援；将

调用不太频繁的信息保存在光盘存储器中，作为硬盘的后援，构成一种硬盘、光盘外存体系。

如前所述，CPU 不能直接运行在外存中的程序和数据，需先将外存当作一种外围设备，将程序和数据调入主存中后才能运行。这种调用往往是以文件的形式组织的，一个文件为一次调用的单位，可以按照文件名调用。从 CPU 操作的层次，则可以数据块为调用单位。

根据外存储器所担负的任务以及工作原理，它有以下几个特点：

第一，信息组织采取文件、数据块的结构，采取顺序存取的方式。

第二，具有很大的存储容量，而且价格低。在某些系统中，联机外存容量极大，有海量存储器之称。这就为整个系统提供了充分的存储空间，从而可以存储大量的软件和数据。

第三，一般都能在断电后长久保存信息，这种性能被称为“非易失性”。

第四，许多外存储器的记录介质，如磁盘，光盘等，可以脱机保存信息。反之，记录有软件信息或数据信息的光盘片只要装入驱动器，就可以快速地将信息输入主机。

第五，由于现在外存储器多属于在机械运动中进行读写，其可靠性要低于半导体存储器，相应的需要采取较复杂的校验技术。

2.1.3 高速缓冲存储器

由于主存的工作速度常常不能满足高速 CPU 的要求，为了解决 CPU 与主存之间的速度匹配问题，许多计算机在主存和 CPU 之间增加了一级高速缓冲存储器，即 cache。高速缓冲存储器速度快，容量小，但价格昂贵，其中存放着最近使用的数据和程序，作为主存中当前活跃信息的一个副本。其容量约为数 K 字节到数百 K 字节，由于容量比较小，可以选用高速半导体存储器，使 CPU 的访存速度得到极大提高。

当 CPU 需要访存时，同时将地址送往主存和 cache。若所需要访问的内容已经复制在 cache 中，则可直接从 cache 中快速地读取信息，称为访问 cache 命中。若访问区间的内容不在 cache 中，则不命中，需要从主存中读取信息并且要考虑更新 cache 中的内容，使其为当前活跃部分。为此需要实现访问地址与 cache 物理地址的映像转换，并采取某种算法刷新 cache 中的内容。

如图 2-2 所示给出了 cache 的基本结构。cache 和主存都被分成若干个大小相等的块，每块由若干字节组成。由于 cache 的容量远小于主存的容量，所以 cache 的块数要远少于主存的块数，它保存的信息只是主存中最活跃的若干块的副本。用主存地址的块号字段访问 cache 标记，并将取出的标记和主存地址的标记字段相比较，若相等，说明访问 cache 有效，称 cache 命中，若不相等，说明访问 cache 无效，称 cache 不命中或失效，而此时需要从主存中将该块取出至 cache 中。

当 CPU 发出读请求时，如果 cache 命中，就直接对 cache 进行读操作，与主存无关；如果 cache 不命中，则仍需访问主存，并把该块信息一次从主存调入 cache 内。若此时 cache 已满，则须根据某种替换算法，用这个块替换掉 cache 中原来的某块信息。

当 CPU 发出写请求时，如果 cache 命中，有可能会遇到 cache 与主存中的内容不一致的问题，处理的方法主要有两种，一是同时写入 cache 和主存，称为写直达法；二是将信息暂时只写入 cache，并用标志将该块加以注明，直到该块从 cache 中替换出来时才一次写入主存，称为写回法。如果不命中，就直接把信息写入主存，而与 cache 无关。

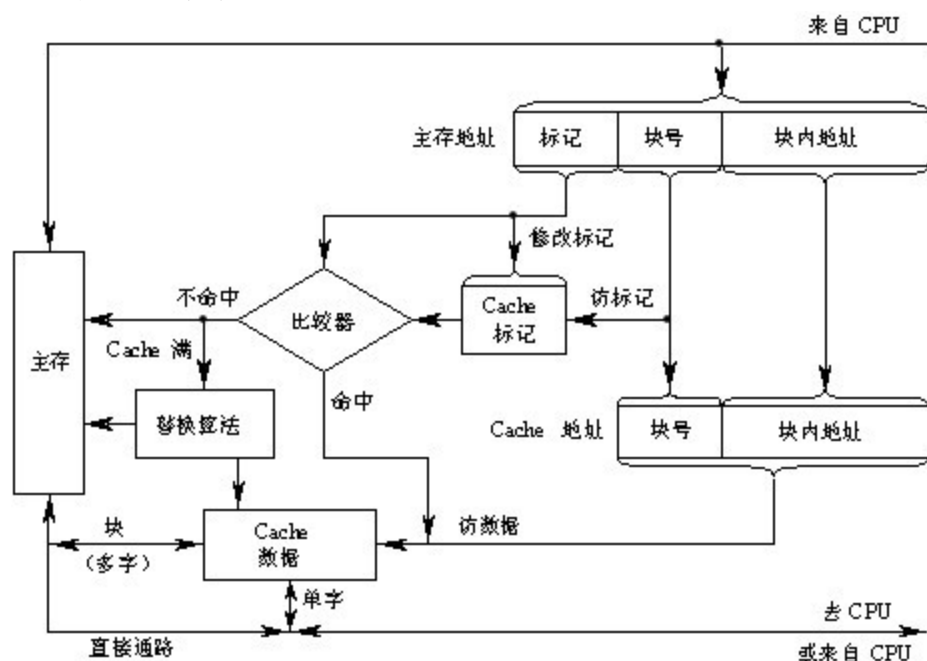


图 2-2 cache 的基本结构

为了把信息放到 cache 中，必须应用某种函数把主存地址映像到 cache 中定位，

称为地址映像（映射）。而将主存地址变换成 cache 地址，称做地址变换，它们之间是密切相关的。常用的地址映像方式有三种，分别是全相联映像、直接映像和组相联映像。

1. 全相联映像

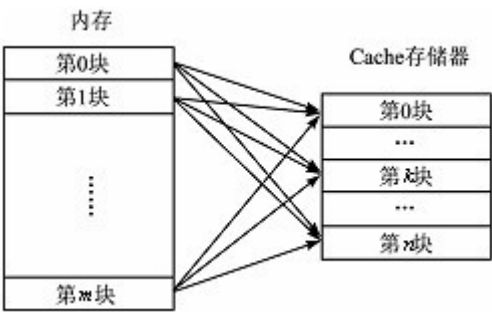


图 2-3 全相联映像

全相联映像是指将内存和 cache 按照固定的相同的大小进行分块。内存的块和 cache 的块可以任意对应，即内存的任何一块都可以映像到 cache 的任何一块。在 cache 的存储空间被占满的情况下，也允许确实已被占满的 cache 存储器中替换出任何一个旧块，具体如图 2-3 所示。

2. 直接映像

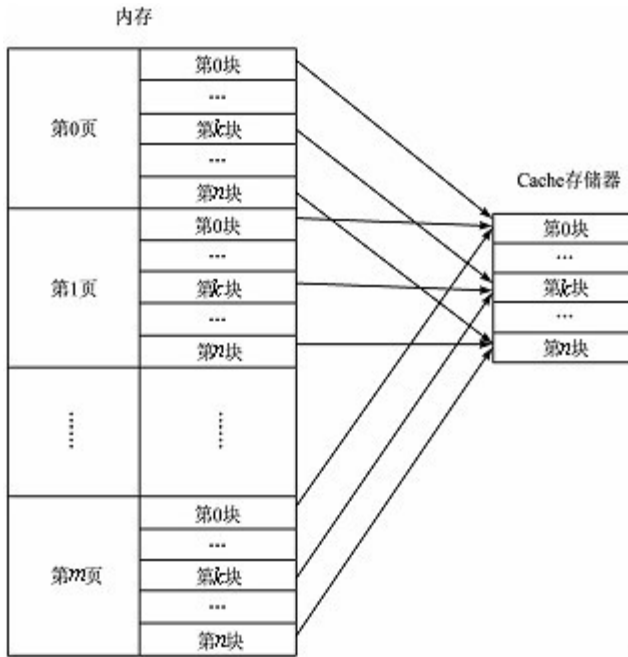


图 2-4 直接映像

直接映像先将 cache 分成若干块，每个块的大小相同，并对每个块进行编号。同时根据 cache 容量大小将内存分成若干区（页），每个区的容量都跟 cache 的容量相同，然后对内存进行分块，每块的大小跟 cache 块的大小相同，同样对区内的块进行编号。映像时，内存的某个区的块只能保存在与其块号相同的 cache 块中。如图 2-4 所示，内存各区中的第 0 块只能映像到 cache 的第 0 块，而不能映像到其他块。

3. 组相联映像

组相联映像实际上是直接映像和全相联映像的折中方案。组相联映像方式先将 cache 分成大小相同的若干区（组），对每个区按照直接映像的方式进行分块，并且编号，因此，cache 中有多个编号相同的块。对内存按照 cache 区的大小进行分页，再对每页按照 cache 块的大小进行分块，每个内存块可以对应不同 cache 区中的相同块号的块。如图 2-5 中内存第 0 页的第 0 块，可以对应 cache 的第 0 区的第 0 块，也可以对应第 j 区的第 0 块。

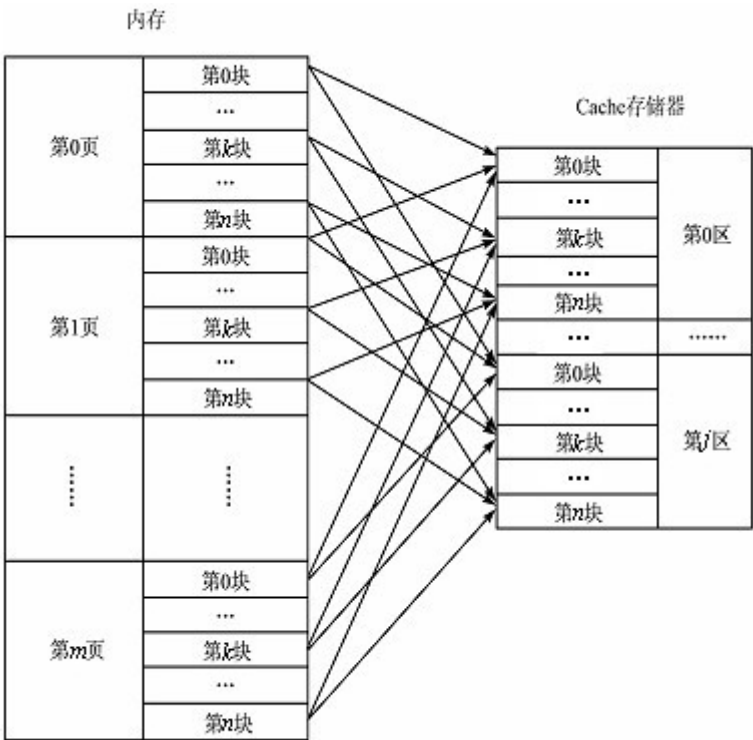


图 2-5 组相联映像

在三种方式中，全相联映像方式比较灵活，cache 的块冲突概率最低、空间利

用率最高，但是地址变换速度慢，而且成本高，实现起来比较困难；直接映像方式是最简单的地址映像方式，成本低，易实现，地址变换速度快，而且不涉及其他两种映像方式中的替换算法问题。但这种方式不够灵活，cache 的块冲突概率最高、空间利用率最低；组相联映像实际上是全相联映像和直接映像的折中方案，其优点和缺点介于全相联和直接映像方式的优缺点之间。

由于 cache 容量有限，为了让 CPU 能及时在缓存中读取到需要的数据，就不得不让内存中的数据替换掉 cache 中的一些数据。但要怎样替换才能确保缓存的命中率呢？这就牵涉到替换策略问题了。

在三种缓存设计方式中，直接映像的替换最简单，因为每个内存块对应的缓存位置都是固定的，只要直接把原来的换出即可。但对于全相联和路组相联来说，就比较复杂了，因为内存块在缓存中组织比较自由，没有固定的位置，要替换的时候必须考虑哪些块 CPU 可能还会用到，而哪些是不会用到。当然，最轻松的就是随机取出一块 CPU 当时不用的块来替换掉，这种方法在硬件上很好实现，而且速度也很快，但缺点也是显而易见的，那就是降低了命中率，因为随机选出的很可能是 CPU 马上就需要的数据。所以，Cache 的替换策略出现了以下几种算法：

先进先出算法 (First In First Out, FIFO)，即替换最早进入缓存的那一块。这种算法，在早期的 CPU 缓存里比较多使用，那时候的 cache 的容量还很小，每块内存块在缓存的时间都不会太久，经常是 CPU 一用完就不得不被替换下来，以保证 CPU 所需要的下块内存块能在缓存中找到。但这样命中率也会降低，因为这种算法所依据的条件是在缓存中的时间，而不能反映其在缓存中的使用情况，最先进入的也许在接下来还会被 CPU 访问。

最不经常使用算法 (Least Frequency Used, LFU)，即替换被 CPU 访问次数最少的块。LFU 算法是将每个缓存块设置个计数器，起始为 0，每被 CPU 访问一次，就加 1，当需要替换时，找到那个计数最小的替换出来，同时将其它块的计数置 0。这种算法在一定程度上还是很理想的，它利用了时间局限性原理，但是每次替换完都把其它块置 0，使得把这种局限性的时间限定在了两次替换之间的时间间隔内。由于替换太频繁，让这时间间隔太短了，并不能完全反映出 CPU 近期的访问情况。

近期最少使用算法 (Least Recently Used, LRU)，即替换在近段时间里，被 CPU 访问次数最少的块，它是 LFU 的基础上实现的。它的原理是在每个块中设置一个计数器，哪块被 CPU 访问，则这块置 0，其它增 1，在一段时间内，如此循环，待到要替换时，把计数值最大的替换出去。这种算法更加充分利用了时间局部性，

既替换了新的内容，又保证了其命中率。有一点要说明的是，有时候“块”替换出，并不代表它一定用不到了，而是缓存容量不够了，为了让要进去的内存块腾出空间，以满足 CPU 的需要。这种算法是目前最优秀的，大部分的 Cache 的替换策略都采用这种算法。

相比于写入操作来说，读取操作是最主要的，并且复杂得多，而写入操作则要简单的多，下面我们来看看 cache 的写入策略。

在一台 PC 中，任何设备都是从内存读取数据的，所以一旦 CPU 更改了 cache 的数据，就必须把新的数据放回内存，使系统其它设备能用到最新的数据，这就涉及到写入了，目前的写入策略有三种：

写回法：当 CPU 写 cache 命中时，只改变其缓存的内容，而不写入内存，直到替换策略把该块替换出来时才写入内存。这种方法减少了访问内存的次数，缩短了时间，也提高了内存带宽利用率，但在保持与内存内容的一致性上存在在隐患，并且使用写回法，必须为每个缓存块设置一个修改位，来反映此块是否被 CPU 修改过。

全写法：当写 cache 命中时，立即在所有的等级存储介质里更新，即同时写进 cache 与内存，使主存和 cache 相关页内容始终保持一致；而当未命中时，则直接向内存写入，而 cache 不用设置修改位或相应的判断器。这种方法的好处是，当 Cache 命中时，由于缓存和内存是同时写入的，所以可以很好的保持缓存和内存内容的一致性，但缺点也很明显，插入慢速的访问主存操作，影响工作速度。

写一次法：这是一种基于上面两种方法的写策略，它的特点是，除了第一次写 cache 命中的时候要写入内存，其它时候都和写回法一样，只修改缓存。其实这也就是一种对缓存一致性的妥协，使得在缓存一致性和延迟中取的一个较好的平衡。

2.2 虚拟存储器

所谓虚拟存储器，就是采用一定的方法将一定的外存容量模拟成内存，同时对程序进出内存的方式进行管理，从而得到一个比实际内存容量大得多的内存空间，使得程序的运行不受内存大小的限制。因此，虚拟存储器是由内存和部分辅存一起组成的。

在虚拟存储器中，主存或部分辅存的地址空间是统一编址的，形成一个庞大的存储空间。在这个大空间里，用户可以自由编程，完全不必考虑程序在主存是否装得下以及这些程序将来在主存中的实际存放位置。当然，这样的编程地址是虚

地址（逻辑地址），而不是实际的主存单元地址。在程序运行时，CPU 以虚地址来访问主存，由辅助硬件找出虚地址和实地址（物理地址）之间的对应关系，并判断这个虚地址指示的存储单元内容是否已装入主存。如果已在主存中，则通过地址变换，CPU 可直接访问主存的实际单元；如果不在主存中，则把包含这个字的一页或一个程序段调入主存后再由 CPU 访问。如果主存已满，则由替换算法从主存中将暂不运行的一页或一段调回辅存，再从辅存调入新的一页或一段到主存。从原理的角度看，虚拟存储器和 cache-主存层次有不少相同之处。

虚拟存储方法的实现也依赖于程序的局部性原理，另外虚拟存储方法还依赖于程序的顺序性特性。

2.2.1 页式虚拟存储器

在页式虚拟存储系统中，将程序按统一的大小划分成多个页，同时也将虚拟存储器划分为同样大小的页，其中虚拟空间的页称为虚页（逻辑页），而主存空间的页称为实页（物理页），并对这些页按地址从低到高的顺序编号。

在编程时，程序的虚地址由高位字段的虚页号和低位字段的页内地址两部分组成，虚页号标识页。虚地址到实地址之间的变换是由页表来实现的。页表是一张存放在主存中的虚页号和实页号的对照表，记录着程序的虚页调入主存时被安排在主存中的位置。若计算机采用多道程序工作方式，则可为每个用户作业建立一个页表，硬件中设置一个页表基址寄存器，存放当前所运行程序的页表的起始地址。

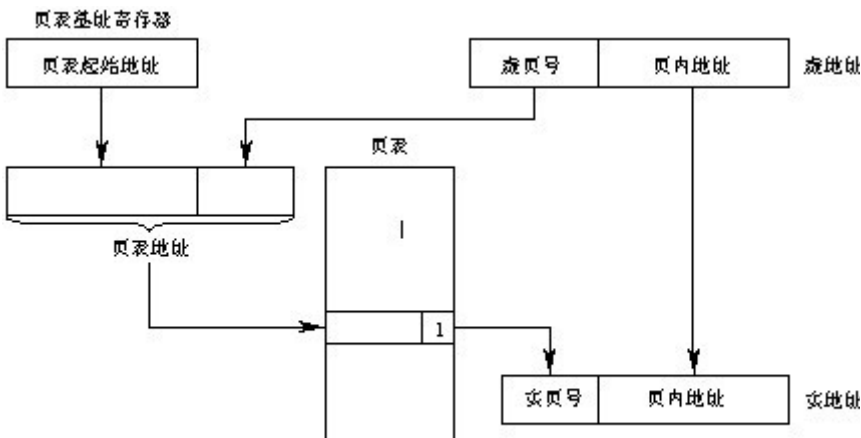


图 2-6 页式虚拟存储器的虚-实地址的变换过程

页表中的每一行记录了与某个虚页对应的若干信息，包括虚页号、装入位和实页号等。页表基址寄存器和虚页号拼接成页表索引地址。根据这个索引地址可读到一个页表信息字，然后检测页表信息字中装入位的状态。若装入位为 1，表示该页面已在主存中，将对应的实页号与虚地址中的页内地址相拼接就得到了完整的实地址；若装入位为 0，表示该页面不在主存中，于是启动 I/O 系统，把该页从辅存中调入主存后再供 CPU 使用，若主存已满，还需要使用替换算法替换页。如图 2-6 所示给出了页式虚拟存储器的虚-实地址的变换过程。

页式虚拟存储器虽然能实现虚拟存储，但它还存在一些不足。

(1) 由于采用定长的页，虽然建立页表方便，页的调入也容易实现。但由于程序不可能正好是页面的整倍数，那么最后一页的零头将无法利用而造成空间浪费。

(2) 由于页不是逻辑上独立的实体，这给程序的处理、保护和共享等带来了麻烦。

2.2.2 段式虚拟存储器

在段式虚拟存储器系统中，将程序按其逻辑结构划分为段，各个段的长度因程序而异。段式虚拟存储器借助于段表来实现虚地址与实地址的转换。段表中每一行记录了某个段对应的若干信息，包括段号、装入位、段起点和段长等。装入位为 1，表示该段已调入主存；装入位为 0，则表示该段不在主存中。段表其实本身也是一个段，可以存放在辅存中，但一般存放在主存中。

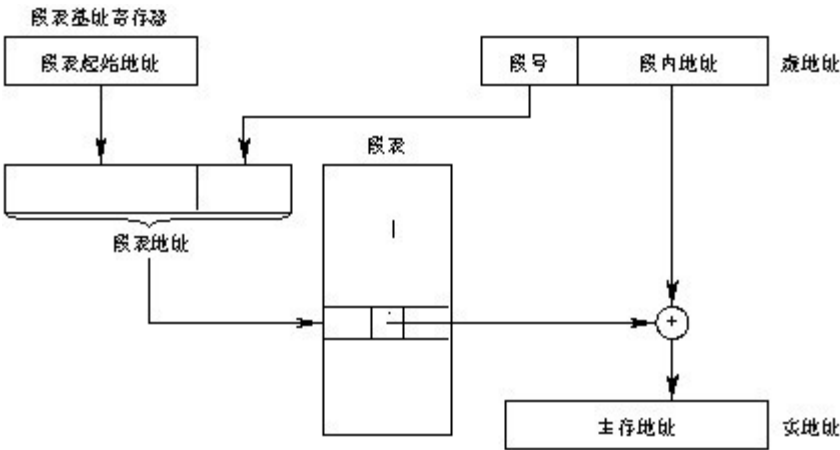


图 2-7 段式虚拟存储器的虚-实地址的变换过程

在段式虚拟存储器系统中，虚地址由段号和段内地址两部分组成，如图 2-7 所示给出了段式虚拟存储器的虚-实地址的变换过程。

由于段式虚拟存储器的段具有逻辑独立性，因此它易于程序的处理、保护和共享等操作，但是，因为段的长度参差不齐，给主存空间分配带来了麻烦，同时很可能也会带来一定的空间浪费。

2.2.3 段页式虚拟存储器

段页式虚拟存储器是对段式、页式虚拟存储器的综合，它先将程序按其逻辑结构分段，再将每段划分为若干大小相等的页，同时将主存空间划分为同样大小的块。如图 2-8 所示。

因为段页式存储管理对逻辑地址进行了两次划分，第一次将逻辑地址划分为若干段，第二次将每个段划分为若干页。因此，要对内存正常寻址，不仅要知道将要访问的地址属于哪个段，也要知道该地址属于该段的哪个页。逻辑页与物理块一一对应，所以需要页表来记录各页对应的块号，且因为每个段都分成了很多页，所以每个段都需要一个页表。同时，作业分成了很多段，为了统一管理，系统需要知道每个段的分页情况，所以又要设置一个段表来记录每个段所对应的页表。

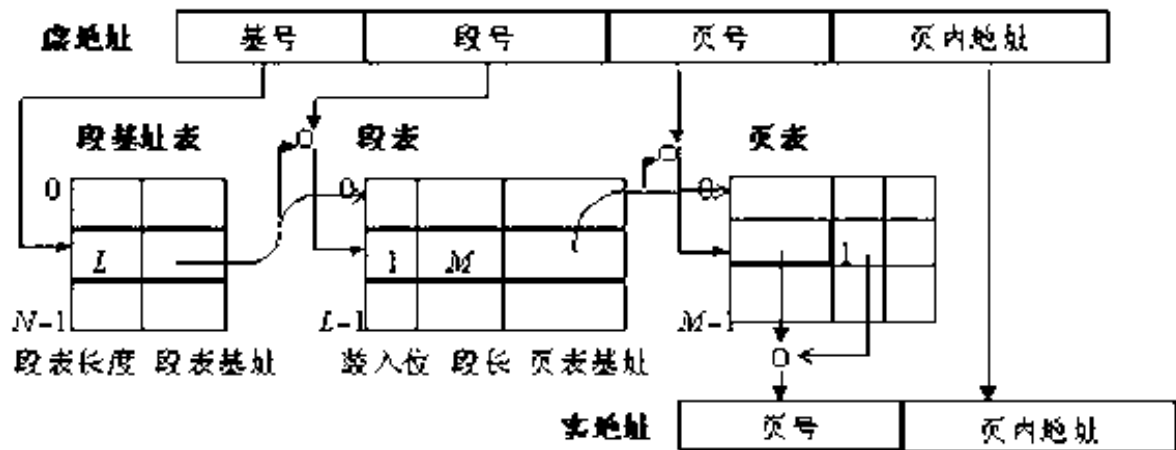


图 2-8 段页式虚拟存储器地址变换

作业将要执行其中的某个语句时，根据其地址计算出段号、页号和页内地址。首先根据段号查找段表，得到该段的页表的起始地址，然后查找页表，得到该页对应的块号，最后根据块的大小和页内地址计算出该语句的内存地址。

2.2.4 快表

TLB (Translation Lookaside Buffer, 旁路转换缓冲) 也称为页表缓冲或快表, 是用来加速虚拟地址到物理地址映射的过程。TLB 中具体的内容对于不同的 CPU 都会有差异, 但是每个表项基本上都会包括虚拟地址和页表的对应关系, 具体的实现策略基本上对于软件来说是透明的。

快表是一块小容量的相联存储器, 由高速缓存器组成, 速度快, 并且可以从硬件上保证按内容并行查找, 一般用来存放当前访问最频繁的少数活动页面的页号。快表的用途是加快线性地址的转换。当一个线性地址第一次使用时, 通过慢速访问 RAM 中的页表计算出相应的物理地址。同时, 物理地址被存放在一个 TLB 表项中, 以便以后对同一个线性地址的引用时可以快速地得到转换。

使用分页内存管理会存在一个二次访问内存的问题, 即要读取一个数据两次访问内存。首先, 从内存中读取页表, 从中找到对应页的首地址; 然后, 程序再次访问内存, 读取数据。这样就降低了计算机的性能 (因为内存远比 CPU 要慢)。所以现代 CPU 中一般都有一个快表寄存器, 里面存储着最近使用过的页表记录。要查询的页表记录已经调入快表寄存器, 就不用再从内存读取而是直接从快表寄存器中读取。由于寄存器的读取速度要远快于内存, 从而提高了系统性能。

2.3 本章小结

本章主要介绍了内存管理模型的基本概念以及一些基本管理方式。对于存储器结构和存储器的管理方式进行了介绍, 其中对于 cache 和虚拟存储器进行了细致地分析。

第三章 Linux 操作系统的内存管理模式

3.1 Linux 简介

1991 年, Linus Torvalds 开发出最初的 Linux, 它作为一个适用于基于 Intel 80386 微处理器的 IBM PC 兼容机的操作系统。现在, Linus 依然不遗余力地改进 Linux, 使它保持与各种硬件平台同步发展, 并协调世界各地数百名 Linux 开发者的开发工作。几年来, 开发者已经使 Linux 可以在其他平台上运行, 包括 HP 的 Alpha, Intel 的 Itanium, AMD 的 AMD64, Power PC 及 IBM 的 zSeries。

Linux 最吸引人的一个优点就在于它不是商业操作系统: 它的源代码在 GNU 公共许可证 (General Public License, GPL) 下是开放的, 任何人都可以获得源代码并研究它; 只要你下载源代码, 或者在 Linux 光盘上找到源代码, 你就可以从用户接口层到与硬件密切相关的操作系统核心层, 对这个最成功而又最现代的操作系统进行由表及里的研究。

从技术角度来说, Linux 是一个真正的 Unix 内核, 但它不是一个完全的 Unix 操作系统, 这是因为它不包含全部的 Unix 应用程序, 诸如文件系统实用程序、窗口系统及图形化桌面、系统管理员命令、文本编辑程序、编译程序等等。不过, 因为以上大部分应用程序都可在 GNU 许可证下免费获得, 因此, 可以把它们安装在任何一个基于 Linux 内核的系统中。

市场上各种类 Unix 系统在很多重要的方面有所不同, 其中有些系统已经有很长的历史, 并且显得有点过时。所有商业版本都是 SVR4 或 4.4B SD 的变体, 并且都趋向于遵循某些通用标准, 诸如 IEEE 的 POSIX (Portable Operating Systems based on Unix 基于 Unix 的可移植操作系统) 和 X/Open 的 CAE (Common Applications Environment, 公共应用环境)。

现有标准仅仅指定了应用程序编程接口 (application programming interface, API), 也就是说, 指定了用户程序应当运行的一个已定义好的环境。因此, 这些标准并没有对内核的内部设计施加任何限制。

为了定义一个通用用户接口, 类 Unix 内核通常采用相同的设计思想和特征。在这一点上, Linux 和其他的类 Unix 操作系统是一样的。

Linux 内核 2.6 版的目标是遵循 IEEE POSIX 标准。这意味着在 Linux 系统下,很容易编译和运行目前现有的大多数 Unix 程序,只需少许或根本无需为源代码打补丁。此外, Linux 包括了现代 Unix 操作系统的全部特点, 诸如虚拟存储、虚拟文件系统、轻量级进程、Unix 信号量、SVR4 进程间通信、支持对称多处理器 (Symmetric Multiprocessor, SMP) 系统等。

Linus Torvalds 在写第一个内核的时候, 参考了 Unix 内幕方面一些经典的书, 比如 Maurice Bach 的(((The Design of the Unix Operating System))) (Prentice Hall, 1986)。实际上, Linux 始终对 Bach 的书(即 SVR4)中所描述的 Unix 基准有些偏爱。但是, Linux 没有拘泥于任何一个特定的变体, 相反, 它尝试采纳了几种不同 Unix 内核中最好的特征和设计选择。

3.2 Linux 内核主要特点

Linux 内核相比其它的操作系统内核, 有如下特点:

单块结构的内核 (Monolithic kernel): 它是一个庞大、复杂的自我完善 (do-it-yourself) 程序, 由几个逻辑上独立的成分构成。在这一点上, 它是相当传统的, 大多数商用 Unix 变体也是单块结构。

编译并静态连接的传统 Unix 内核: 大部分现代操作系统内核可以动态地装载和卸载部分内核代码(典型的例子如设备驱动程序), 通常把这部分代码称做模块 (module)。Linux 对模块的支持是很好的, 因为它能自动按需装载或卸载模块。在主要的商用 Unix 变体中, 只有 SVR4.2 和 Solaris 内核有类似的特点。

内核线程: 一些 Unix 内核, 如 Solaris 和 S V R4.2/MP, 被组织成一组内核线程 (kernelthread)。内核线程是一个能被独立调度的执行环境 (context); 也许它与用户程序有关, 也许仅仅执行一些内核函数。线程之间的上下文切换比普通进程之间的上下文切换花费的代价要少得多, 因为前者通常在同一个地址空间执行。Linux 以一种十分有限的方式使用内核线程来周期性地执行几个内核函数, 但是, 它们并不代表基本的执行上下文的抽象。

多线程应用程序支持: 大多数现代操作系统在某种程度上都支持多线程应用程序, 也就是说, 这些用户程序是根据很多相对独立的执行流来设计的, 而这些执行流之间共享应用程序的大部分数据结构。一个多线程用户程序由很多轻量级进程 (lightweight process, LWP) 组成, 这些进程可能对共同的地址空间、共同的物理内存页、共同的打开文件等等进行操作。Linux 定义了自己的轻量级进程版本,

这与 SVR4, Solaris 等其他系统上所使用的类型有所不同。当 LWP 的所有商用 Unix 变体都基于内核线程时, Linux 却把轻量级进程当作基本的执行上下文, 通过非标准的 clone() 系统调用来处理它们。

抢占式(preemptive)内核: 当采用“可抢占的内核”选项来编译内核时, Linux 2.6 可以随意交错执行处于特权模式的执行流。除了 Linux 2.6, 还有一些传统的、通用的 Unix 系统(如 Solaris 和 Mach3.0)是完全的抢占式内核。SVR4.2/MP 通过引入一些固定抢占点(fixed preemption point)的方法获得有限的抢占能力。

多处理器支持: 几种 Unix 内核变体都利用了多处理器系统。Linux 2.6 支持不同存储模式的对称多处理(SMP), 包括 NUMA: 系统不仅可以使使用多处理器, 而且每个处理器可以毫无区别地处理任何一个任务。尽管通过一个单独的“大内核锁”使得内核中的少数代码依然串行执行, 但公平地说, Linux 2.6 以几乎最优化的方式使用 SMP。

文件系统: Linux 标准文件系统呈现出多种风格。如果你没有特殊需要, 就可以使用普通的 Ext2 文件系统。如果你想避免系统崩溃后冗长的文件系统检查, 就可以切换到 Ext3。如果你不得不处理很多小文件, ReiserFS 文件系统可能就是最好的选择。除了 Ext3 和 ReiserFS, 还可以在 Linux 中使用另外几个日志文件系统; 这些文件系统包括 IBM AIX 的日志文件系统(Journaling File System, JFS)和 SGI 公司 IRIX 系统上的 XFS 文件系统。有了强大的面向对象虚拟文件系统技术(为 Solaris 和 SVR4 所采用), 把外部文件系统移植到 Linux 比移植到其他内核相对要容易。

STREAMS: 尽管现在大部分的 Unix 内核内包含了 SVR4 引入的 STREAMS I/O 子系统, 并且已变成编写设备驱动程序、终端驱动程序及网络协议的首选接口, 但是 Linux 并没有与此类似的子系统。

对 Linux 的评价充分说明, 与商业化的操作系统相比, Linux 已经具备足够的竞争力。而且, Linux 一些独具特色的特点使其成为一种趣味盎然的操作系统。商业化的 Unix 内核为了赢得更大的市场份额通常也引入了新特征, 但这些特征本是可有可无, 其稳定性和效率都值得商榷。事实上, 现代 Unix 内核有向更臃肿变化的倾向, 而 Linux 以及其他开放源代码的操作系统不受市场因素的制约, 因此可以根据设计者的想法(主要是 Linus Torvalds 的想法)自由地演进。尤其是, 与商用竞争对手相比, Linux 有如下优势:

Linux 是免费的。除硬件之外, 你无需任何花费就能安装一套完整的 Linux 系

统。

Linux 的所有成分都可以充分地定制。通过内核编译选项，你可以选择自己真正需要的特征来定制内核。而且有了通用公共许可证 (GPL)，你就可以自由地阅读、修改内核和所有系统程序的源代码。

Linux 可以运行在低档、便宜的硬件平台上。你可以用一个 4MB 内存的旧 Intel 80386 系统构建网络服务器。

Linux 是强大的。由于充分挖掘了硬件部分的特点，使得 Linux 系统速度非常快。Linux 的主要目标是效率，所以，商用系统的许多设计选择由于有降低性能的隐患而被 Linux 舍弃，如 STREAMSI/O 子系统。

Linux 的开发都是非常出色程序员。Linux 系统非常稳定，有非常低的故障率和非常少系统维护时间。

Linux 内核非常小，而且紧凑。甚至可以把一个内核映像和一些系统程序放在一张 1.4MB 的软盘上。据了解，现在还没有一个商用 Unix 变体能从一张软盘上启动。

Linux 与很多通用操作系统高度兼容。Linux 可以让你直接安装以下文件系统的所有版本:MS-DOS 和 MS Windows, SVR4, OS/2, Mac OS X, Solaris, SunOS, NEXTSTEP,还有很多BSD变体等等。另外,Linux也能对很多网络层进行操作, 这些网络层如以太网、光纤分布式数据接口(Fiber Distributed Data Interface, FDDI)、高性能并行接口(High Performance Parallel Interface, HIPPI), IEEE 802.11(无线局域网)和 IEEE802.15(蓝牙)。通过使用适当的库函数, Linux 系统甚至能直接运行为其他操作系统所编写的程序。例如, Linux 能执行为以下操作系统所编写的应用程序: MS-DOS, MS Windows, SVR3 及 SV R4, 4.4BSD, SCO Unix, Xenix, 以及其他在 Intel 80x86 平台上运行的操作系统程序。

Linux 有很好的技术支持。Linux 比任何有版权的操作系统更容易获得补丁和更新。如果你把遇到的难题发给一些新闻组或邮件列表, 经常在几个小时内就会得到回应。此外, 当新的硬件产品投放市场以后, 其 Linux 驱动程序通常在几周内就可得到。与此相反, 硬件厂商仅仅给少数商业操作系统发布设备驱动程序, 通常只有微软一家。因此, 所有商用 Unix 变体只能运行在有限的硬件上。

因为有了数千万台安装 Linux 的基础, 那些习惯了其他操作系统某些标准特征的用户开始期望 Linux 也具有相同的特征。在这种情况下, 对 Linux 开发者的需求也在不断增加。值得庆幸的是, 在 Linus 的密切指导下, Linux 始终在不断发展以满足如此众多的需求。

3.3 Linux 内存管理器

3.3.1 地址映射

linux 内核使用页式内存管理，应用程序给出的内存地址是虚拟地址，它需要经过若干级页表一级一级的变换，才变成真正的物理地址。当访问一个由虚拟地址表示的内存空间时，需要先经过若干次的内存访问，得到每一级页表中用于转换的页表项，才能完成映射。也就是说，要实现一次内存访问，实际上内存被访问了 $N+1$ 次（ N =页表级数），并且还需要做 N 次加法运算。

所以，地址映射必须要有硬件支持，MMU（内存管理单元）就是这个硬件。并且需要有 cache 来保存页表，这个 cache 就是 TLB（Translation lookaside buffer）。

尽管如此，地址映射还是有着不小的开销。假设 cache 的访存速度是内存的 10 倍，命中率是 40%，页表有三级，那么平均一次虚拟地址访问大概就消耗了两次物理内存访问的时间。于是，一些嵌入式硬件上可能会放弃使用 mmu，这样的硬件能够运行 VxWorks 操作系统、linux 等系统。但是使用 mmu 的优势也是很大的，最主要的是出于安全性考虑。各个进程都是相互独立的虚拟地址空间，互不干扰。而放弃地址映射之后，所有程序将运行在同一个地址空间。于是，在没有 mmu 的机器上，一个进程越界访存，可能引起其他进程莫名其妙的错误，甚至导致内核崩溃。

在地址映射这个问题上，内核只提供页表，实际的转换是由硬件去完成的。这个过程涉及两方面的内容：虚拟地址空间的管理和物理内存的管理。

3.3.2 虚拟地址管理

每个进程对应一个 task 结构，它指向一个 mm 结构，这就是该进程的内存管理器。对于线程来说，每个线程也都有一个 task 结构，但是它们都指向同一个 mm，所以地址空间是共享的。

mm->pgd 指向容纳页表的内存，每个进程有自己的 mm，每个 mm 有自己的页表。于是，进程调度时，页表被切换，一般会有一个 CPU 寄存器来保存页表的地址，比如 X86 下的 CR3，页表切换就是改变该寄存器的值。所以，各个进程的地址空间互不影响，因为页表都不一样了，当然无法访问到别人的地址空间上。但是共享内存除外，这是故意让不同的页表能够访问到相同的物理地址上。

用户程序对内存的操作都是对 mm 的操作，具体来说是对 mm 上的 vma（虚拟内存空间）的操作。这些 vma 代表着进程空间的各个区域，比如堆、栈、代码区、数据区、各种映射区、等等。

用户程序对内存的操作并不会直接影响到页表，更不会直接影响到物理内存的分配。比如 malloc 成功，仅仅是改变了某个 vma，页表不会变，物理内存的分配也不会变。

假设用户分配了内存，然后访问这块内存。由于页表里面并没有记录相关的映射，CPU 产生一次缺页异常。内核捕捉异常，检查产生异常的地址是不是存在于一个合法的 vma 中。如果不是，则给进程一个“段错误”，让其崩溃；如果是，则分配一个物理页，并为之建立映射。

3.3.3 物理内存管理

下面来看看物理内存的分配过程。

首先，linux 支持 NUMA，物理内存管理的第一个层次就是介质的管理。pg_data_t 结构就描述了介质。一般而言，我们的内存管理介质只有内存，并且它是均匀的，所以可以简单地认为系统中只有一个 pg_data_t 对象。

每一种介质下面有若干个 zone。一般是三个，DMA、NORMAL 和 HIGH。

DMA：因为有些硬件系统的 DMA 总线比系统总线窄，所以只有一部分地址空间能够用作 DMA，这部分地址被管理在 DMA 区域；

HIGH：高端内存。在 32 位系统中，地址空间是 4G，其中内核规定 3-4G 的范围是内核空间，0-3G 是用户空间。前面提到过内核的地址映射是写死的，就是指这 3-4G 的对应的页表是写死的，它映射到了物理地址的 0-1G 上。所以，大于 896M 的物理地址是没有写死的页表来对应的，内核不能直接访问它们，必须要建立映射，称它们为高端内存；

NORMAL：不属于 DMA 或 HIGH 的内存就叫 NORMAL。

在 zone 之上的 zone_list 代表了分配策略，即内存分配时的 zone 优先级。一种内存分配往往不是只能在一个 zone 里进行分配的，比如分配一个页给内核使用时，最优先是从 NORMAL 里面分配，不行的话就分配 DMA 里面的好了，这就是一种分配策略。

每个内存介质维护了一个 mem_map，为介质中的每一个物理页面建立了一个 page 结构与之对应，以便管理物理内存。

每个 zone 记录着它在 mem_map 上的起始位置。并且通过 free_area 串连着这个 zone 上空闲的 page。物理内存的分配就是从这里来的，从 free_area 上把 page 摘下，就算是分配了。

3.3.4 建立地址映射

内核需要物理内存时，很多情况是整页分配的，这在上面的 mem_map 中摘一个 page 下来就好了。比如前面说到的内核捕捉缺页异常，然后需要分配一个 page 以建立映射。

内核代码所访问的地址都是虚拟地址，因为 CPU 指令接收的就是虚拟地址。但是，建立地址映射时，内核在页表里面填写的内容却是物理地址，因为地址映射的目标就是要得到物理地址。mem_map 中的 page 就是根据物理内存来建立的，每一个 page 就对应了一个物理页。

虚拟地址的映射是靠这里 page 结构来完成的，是它们给出了最终的物理地址。然而，page 结构显然是通过虚拟地址来管理的。在内核初始化时，内核的地址空间就已经把地址映射写死了。page 结构显然存在于内核空间，所以它的地址映射问题已经通过“写死”解决了。

由于内核空间的页表项是写死的，NORMAL（或 DMA）区域的内存可能被同时映射到内核空间 and 用户空间。被映射到内核空间是显然的，因为这个映射已经写死了。而这些页面也可能被映射到用户空间的，在前面提到的缺页异常的场景里面就有这样的可能。映射到用户空间的页面应该优先从 HIGH 区域获取，因为这些内存被内核访问起来很不方便，拿给用户空间再合适不过了。但是 HIGH 区域可能会耗尽，或者可能因为设备上物理内存不足导致系统里面根本就没有 HIGH 区域，所以，将 NORMAL 区域映射给用户空间是必然存在的。

但是 NORMAL 区域的内存被同时映射到内核空间 and 用户空间并没有问题，因为如果某个页面正在被内核使用，对应的 page 应该已经从 free_area 被摘下，于是缺页异常处理代码中不会再将该页映射到用户空间。反过来也一样，被映射到用户空间的 page 自然已经从 free_area 被摘下，内核不会再去使用这个页面。

3.3.5 内核空间管理

除了对内存整页的使用，有些时候，内核也需要像用户程序使用 malloc 一样，分配一块任意大小的空间。这个功能是由 slab 系统来实现的。

slab 相当于为内核中常用的一些结构体对象建立了对象池，比如对应 task 结构的池、对应 mm 结构的池、等等。

而 slab 也维护有通用的对象池，比如“32 字节大小”的对象池、“64 字节大小”的对象池、等等。内核中常用的 kmalloc 函数（类似于用户态的 malloc）就是在这些通用的对象池中实现分配的。

slab 除了对象实际使用的内存空间外，还有其对应的控制结构。有两种组织方式，如果对象较大，则控制结构使用专门的页面来保存；如果对象较小，控制结构与对象空间使用相同的页面。

除了 slab，linux 2.6 还引入了 mempool（内存池）。其意图是：某些对象我们不希望它会因为内存不足而分配失败，于是我们预先分配若干个，放在 mempool 中存起来。正常情况下，分配对象时是不会去动 mempool 里面的资源的，照常通过 slab 去分配。到系统内存紧缺，已经无法通过 slab 分配内存时，才会使用 mempool 中的内容。

3.3.6 用户空间内存管理

malloc 是 libc 的库函数，用户程序一般通过它（或类似函数）来分配内存空间。libc 对内存的分配有两种途径，一是调整堆的大小，二是 mmap 一个新的虚拟内存区域。

在内核中，堆是一个一端固定、一端可伸缩的 vma。可伸缩的一端通过系统调用 brk 来调整。libc 管理着堆的空间，用户调用 malloc 分配内存时，libc 尽量从现有的堆中去分配。如果堆空间不够，则通过 brk 增大堆空间。

当用户将已分配的空间 free 时，libc 可能会通过 brk 减小堆空间。但是堆空间增大容易减小却难，考虑这样一种情况，用户空间连续分配了 10 块内存，前 9 块已经 free。这时，未 free 的第 10 块哪怕只有 1 字节大，libc 也不能够去减小堆的大小。因为堆只有一端可伸缩，并且中间不能掏空。而第 10 块内存就死死地占据着堆可伸缩的那一端，堆的大小没法减小，相关资源也没法归还内核。

当用户 malloc 一块很大的内存时，libc 会通过 mmap 系统调用映射一个新的 vma。因为对于堆的大小调整和空间管理还是比较麻烦的，重新建一个 vma 会更方便。

在 malloc 的时候不能总是去 mmap 一个新的 vma。有两个原因：第一，对于小空间的分配与回收，被 libc 管理的堆空间已经能够满足需要，不必每次都去进行

系统调用。并且 vma 是以 page 为单位的，最小就是分配一个页；第二，太多的 vma 会降低系统性能。缺页异常、vma 的新建与销毁、堆空间的大小调整、等等情况下，都需要对 vma 进行操作，需要在当前进程的所有 vma 中找到需要被操作的那个（或那些）vma。vma 数目太多，必然导致性能下降。在进程的 vma 较少时，内核采用链表来管理 vma；vma 较多时，改用红黑树来管理。

3.3.7 用户的栈

与堆一样，栈也是一个 vma，这个 vma 是一端固定、一端可伸的。这个 vma 比较特殊，没有类似 brk 的系统调用让这个 vma 伸展，它是自动伸展的。

当用户访问的虚拟地址越过这个 vma 时，内核会在处理缺页异常的时候将自动将这个 vma 增大。内核会检查当时的栈寄存器，访问的虚拟地址不能超过 ESP 加 n（n 为 CPU 压栈指令一次性压栈的最大字节数）。也就是说，内核是以 ESP 为基准来检查访问是否越界。

但是，ESP 的值是可以由用户态程序自由读写的，用户程序如果调整 ESP，不能将栈划得很大。内核中有一套关于进程限制的配置，其中就有栈大小的配置，栈只能这么大，再大就出错。

对于一个进程来说，栈一般是可以被伸展得比较大（如：8MB）。线程的 mm 是共享其父进程的。虽然栈是 mm 中的一个 vma，但是线程不能与其父进程共用这个 vma（两个运行实体显然不用共用一个栈）。于是，在线程创建时，线程库通过 mmap 新建了一个 vma，以此作为线程的栈（大于一般为：2M）。

可见，线程的栈在某种意义上并不是真正栈，它是一个固定的区域，并且容量很有限。

3.4 本章小结

本章是对于 Linux 操作系统及其特点作了简要介绍，并对 Linux 内存管理器进行了分析。对 Linux 物理内存管理，虚拟内存管理，用户空间内存管理作了详细描述。

第四章 Linux 伙伴系统及改进方案

4.1 Linux 伙伴系统

Linux 的伙伴算法把所有的空闲页面分为 10 个块组，每组中块的大小是 2 的幂次方个页面，例如，第 0 组中块的大小都为 2^0 （1 个页面），第 1 组中块的大小都为 2^1 （2 个页面），第 9 组中块的大小都为 2^9 （512 个页面）。也就是说，每一组中块的大小是相同的，且这同样大小的块形成一个链表。

假设请求分配 4 个页面，根据该算法先到第 2（ $2^2=4$ ）个组中寻找空闲块，如果该组中没有空闲块就到第 3（ $2^3=8$ ）个组中寻找，假设在第 3 个组中找到空闲块，就把其中的 4 个页面分配出去，剩余的 4 个页面放到第 2 个组中。如果第 3 个组还是没有空闲块，就到第 4（ $2^4=16$ ）个组中寻找，如果在该组中找到空闲块，就把其中的 4 个页面分配出去，剩余的 12 个页面被分成两个部分，其中的 8 个页面放到第 3 个组中，另外 4 个页面放到第 2 组，……，以此类推。同理，释放时会尽量向上合并空闲块。

伙伴算法和 `free_area_t free_area[MAX_ORDER]` 结构密切相关，下面来看看它的详细描述。

```
type struct free_area_struct
{
    struct list_head    free_list
    unsigned int        *map
} free_area_t;
```

其中 `list_head` 域是一个通用的双向链表结构，链表中元素的类型将为 `mem_map_t`（即 `struct page` 结构）。`Map` 域指向一个位图，其大小取决于现有的页面数。`free_area` 第 k 项位图的每一位，描述的就是大小为 2^k 个页面的两个伙伴块的状态。如果位图的某位为 0，表示一对兄弟块中或者两个都空闲，或者两个都被分配，如果为 1，肯定有一块已被分配。当兄弟块都空闲时，内核把它们当作一个大小为 2^{k+1} 的单独块来处理。如图 4-1 给出该数据结构的示意图。

`free_aera` 数组的元素 0 包含了一个空闲页（页面编号为 0）；而元素 2 则包含了两个以 4 个页面为大小的空闲页面块，第一个页面块的起始编号为 4，而第二

个页面块的起始编号为 56。当需要分配若干个内存页面时，用于 DMA 的内存页面必须是连续的。其实为了便于管理，从伙伴算法可以看出，只要请求分配的块大小不超过 512 个页面（2KB），内核就尽量分配连续的页面。

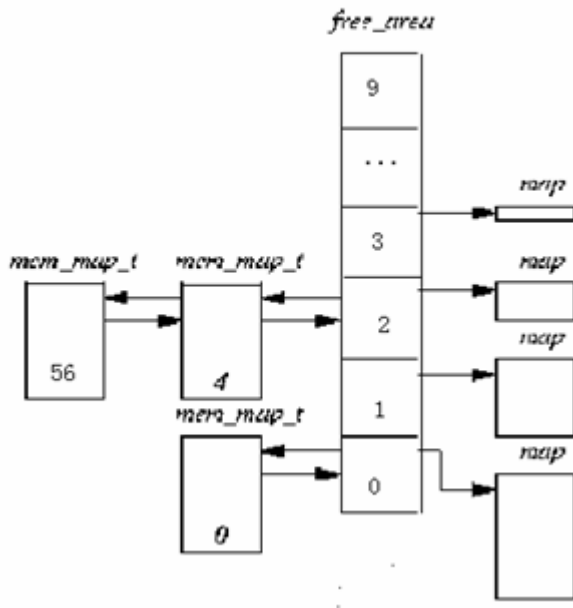


图 4-1 伙伴系统使用的数据结构

4.2 代码分析

下面对涉及伙伴算法的几个代码片段进行简略的分析。在内核模块的编写中，我们用 `alloc_page()`/`alloc_pages()` 来分配大内存(超过一个页面，即 4K)。用 `free_page()`/`free_pages()` 来释放用前者分得的内存。

一个操作系统的内存管理方式很大程度上决定了它的效率，时间与空间的对立统一在内存管理上体现得最为明显。首先，分配和释放内存是一个发生频率很高的操作，所以它要求有一定的实时性，另外，内存又是一种非常宝贵的资源，所以要尽量减少内存碎片的产生。

Linux 采用了伙伴系统算法来管理内存，即把内存按 $2^0, 2^1, 2^2 \dots 2^{10}$ 大小进行分组。每次分配内存时，从相应大小的池中分配内存，然后再把余下的内存分配给它的下一级缓存池。如图 4-2 所示。

在 Linux 中，每一个大小的缓存池都对应一个位图，然后，位图中的每一位对应一对空闲区的空闲标志。每分配一个分配区，都对相应的位图取反。例如：位图

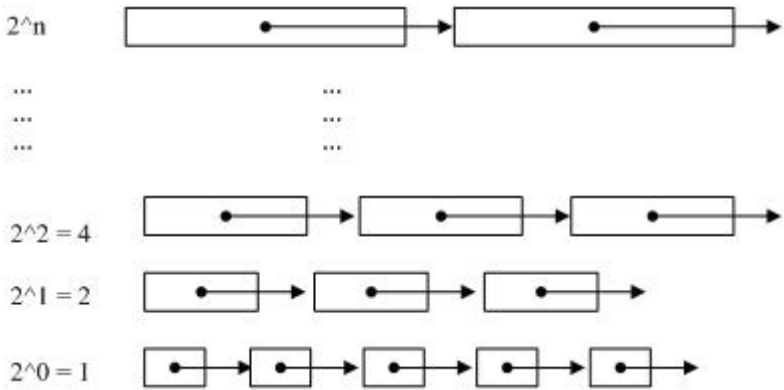


图 4-2 伙伴系统分组

管理 A, B 两个分配块，初始化的时候，该位图置为 0。此时，若把 A 分配出去，更新位图为 1，若再把 B 分配出去，再取反，变为了 0。不难发现有以下规律：

若位图为 1:表示其中有一个已经分配出去，有一个空闲。

若位图为 0:表示两个分配块都空或者都已经分配出去。

关于分配位图，特别需要注意两个地方：

第一，位图的大小只跟总内存有关，跟当前缓存池的空闲内存无关。例如：总内存为 4M 的系统。总共有 $4M/4K = 1024$ 个页，那对应的有：

2^0 位图有： $1024/1/2=512$ 项

2^1 位图有： $1024/2/2=256$ 项

.....

2^{10} 位图有： $1024/2^{10}/2=1$ 项

因为 1 位表示二个页面，所以还需要除以 2。

第二，为了更好的管理伙伴系统，设计了一位管理一对内存，也有减少位图大小的因素。

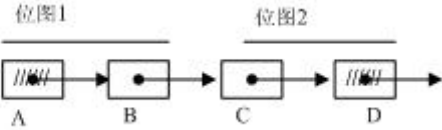


图 4-3 伙伴系统位图

如图 4-3 所示，位图 1 表示 A, B 的分配情况，位图 2 表示 C, D 的分配情况。A, D 已经分配出去了，B, C 处于空闲状态。虽然 B, C 处理连续的页，且都空闲，但不可以把 B, C 合位为一块大内存。

4.2.1 有关的数据结构

在系统中，每一个页面对应一个页描述符，它的结构如下：

```
struct page
{
    page_flags_t flags;        //页面所对应的标志，这与页表项的标志不同
    atomic_t _count;           //页面引用计数
    atomic_t _mapcount;        //有多少个页表项映射到了此页面
    unsigned long private;      //私有数据区，这个成员将在交换与磁盘高速
                                //缓存中使用
    struct address_space * mapping; //用于磁盘高速缓存
    pgoff_t index;             //同上
    struct list_head lru;       //LRU 链表，链至管理区的相应链表中
#ifdef WANT_PAGE_VIRTUAL
    void *virtual;              /* Kernel virtual address (NULL if
                                not kmapped, ie. highmem) */
#endif /* WANT_PAGE_VIRTUAL */
}
```

在 Linux 中，用一个全局的 page 结构数组 mem_map 描述系统中可管理的物理页面项。管理区这个概念有一定的历史形成原因，由于计算机系统的发展，老的 ISA 设备的 DMA 只能使用前 16M 的内存，这就是 ZONE_DMA。另外，由于内核空间默认配置是 1GB，高于 1GB 的物理地址要通过特殊的方式才能供内核使用，这就是 ZONE_HIGHMEM，余下的就是 ZONE_NORMAL 了。

具体的看一下管理区的结构

```
struct zone
{
    spinlock_t lock; //互斥锁
    unsigned long free_pages; //这个区中现有的空闲页面数
```

```

unsigned long    pages_min,  pages_low,  pages_high; //pages_min、
               pages_low 及 pages_high 是对这个区最少、此少及最多页面个数的描述
unsigned long    protection[MAX_NR_ZONES]; //每个类型的管理区的，所
               保护的页面数
spinlock_t      lru_lock;
struct list_head active_list;           //活跃列表
struct list_head inactive_list;         //不活跃列表
unsigned long    nr_scan_active; //内存回收时，所扫描的活跃列表中的
               页面数
unsigned long    nr_scan_inactive; //内存回收时，所扫描的非活跃列表
               中的页面数
unsigned long    nr_active; //活跃链表中的页面个数
unsigned long    nr_inactive; //非活跃链表中的页面个数
int all_unreclaimable; /* All pages pinned */
unsigned long    pages_scanned;         /* since last reclaim */
int temp_priority; //temp_priority 与 prev_priority 在内存回收算法中
int prev_priority; //使用
struct free_area free_area[MAX_ORDER]; //伙伴分配系统中的位图数组和
               页面链表
wait_queue_head_t * wait_table;
unsigned long    wait_table_size;
unsigned long    wait_table_bits;
struct per_cpu_pageset  pageset[NR_CPUS]; //PCP 结构
struct pglist_data    *zone_pgdat; //本管理区所在的存储节点
struct page          *zone_mem_map; //该管理区的内存映射表
unsigned long    zone_start_pfn; // 起始页面号
char    *name; //管理区中的名字
unsigned long    spanned_pages; /* total size, including holes */
unsigned long    present_pages; /* amount of memory (excluding
               holes) */
}

```

在计算机系统中，并不是每一块内存处理的访问时间对于 CPU 来说都是一样的，例如，靠近 CPU 的内存插槽的内存访问速度要较快于另外的插槽。CPU 缓存的访问速度要大于内存芯片的访问速度。基于这种情况，内核为每个区域都定义了一个结点，认为 CPU 访问结点所表示的内存消耗的时间是一样的。看一下结点的数据结构：

```
typedef struct pglist_data
{
    struct zone node_zones[MAX_NR_ZONES];    //结点中的管理区数组
    struct zonelist node_zonelists[GFP_ZONETYPES]; //按分配内存的先后顺序排列的管理区
    int nr_zones;    //该结点总共有多少个管理区
    struct page *node_mem_map;    //结点中的页描述符数组
    struct bootmem_data *bdata;    //只用在初始化阶段 (boot_mem)
    unsigned long node_start_pfn;    //起始物理页号
    unsigned long node_present_pages; //总共的物理页面数目
    unsigned long node_spanned_pages; //物理页面的总大小
    int node_id;    //结点的 ID
    struct pglist_data * pgdat_next; //下一个结点
    wait_queue_head_t kswapd_wait;
    struct task_struct *kswapd;    //上面这两个成员跟 kswapd 有关
}
```

如图 4-4 可以表示出上述三个结构的关系：

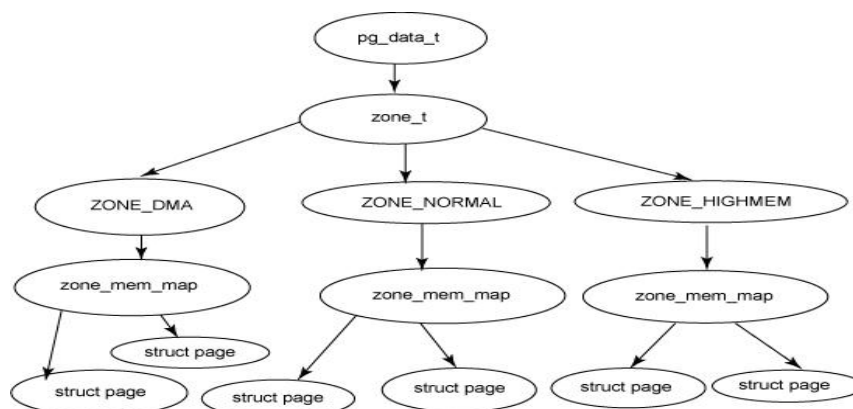


图 4-4 管理区结构

4.2.2 alloc_pages()/alloc_page() 实现分析

函数原型:

```
alloc_pages(unsigned int gfp_mask, unsigned int order);  
alloc_page(unsigned int gfp_mask);
```

其中 alloc_pages() 用来分配多页内存; alloc_page() 只用来分配单页内存。而在内核中, alloc_page() 就是调用了 alloc_pages(gfp_mask, 0) 来实现分配的。gfp_mask 表示分配的标志。

此外, alloc_pages 返回的是 page 结构, 如果要使用分配到的内存, 还需使用函数 void *page_address(struct page *page) 将其转换一下。

下面来看一下具体的代码:

```
static inline struct page * alloc_pages(unsigned int gfp_mask, unsigned  
int order)  
{  
    if (unlikely(order >= MAX_ORDER)) //参数有效性判断. MAX_ORDER 通常被  
        return NULL; //定义为 11, 所以, 最大只能分得  $2^{10}$  大小的内存  
    return alloc_pages_current(gfp_mask, order);  
}  
  
struct page *alloc_pages_current(unsigned gfp, unsigned order)  
{  
    //当前进程的内存分配策略, 通常为 NULL  
    struct mempolicy *pol = current->mempolicy;  
    if (!pol || in_interrupt())  
        pol = &default_policy;  
    if (pol->policy == MPOL_INTERLEAVE)  
        return alloc_page_interleave(gfp, order, interleave_nodes(pol));  
    return _ _alloc_pages(gfp, order, zonelist_policy(gfp, pol));  
}
```

zonelist_policy() 的作用是根据参数标志和当前 CPU 节点得到合适的 zone_list。下面进入 _ _alloc_pages() 看具体实现。

```
struct page * fastcall
```

```

__alloc_pages(unsigned int gfp_mask, unsigned int order, struct zonelist
*zonelist)
{
    const int wait = gfp_mask & __GFP_WAIT;
    unsigned long min;
    struct zone **zones, *z;
    struct page *page;
    struct reclaim_state reclaim_state;
    struct task_struct *p = current;
    int i;
    int alloc_type;
    int do_retry;
    int can_try_harder;
    might_sleep_if(wait); //自旋锁睡眠调试函数，在没有开启相关 DEBUG 开关
        的情况下，此函数即为空。
    can_try_harder = (unlikely(rt_task(p)) && !in_interrupt()) || !wait;
        //取得 zonelist 中的 zone 数组。
    zones = zonelist->zones; /* the list of zones suitable for gfp_mask */
    if (unlikely(zones[0] == NULL)) //如果管理区为空，退出。
    {
        /* Should this ever happen?? */
        return NULL;
    }
    alloc_type = zone_idx(zones[0]); //所要分配的类型，比如说 ZONE_DMA
    /* Go through the zonelist once, looking for a zone with enough free
    */
    for (i = 0; (z = zones[i]) != NULL; i++)
    {
        //找到一个合适大小的 zone 区
        //判断分配之后的内存是否超过所充许的低位 pages_low
        min = z->pages_low + (1<<order) + z->protection[alloc_type];
        if (z->free_pages < min)

```



```

        continue;
//从此 zone 中分配内存页面
page = buffered_rmqueue(z, order, gfp_mask);
if (page)
    goto got_pg;
}
for (i = 0; (z = zones[i]) != NULL; i++)
//分配内存失败了, 唤醒 kswapd 进行内存回收
    wakeup_kswapd(z);
for (i = 0; (z = zones[i]) != NULL; i++) {
    //进行内存回收之后, 重新寻找有空闲的 zone 区, 这次降低了低位要求
    min = z->pages_min;
    if (gfp_mask & __GFP_HIGH)
        min /= 2;
    if (can_try_harder)
        min -= min / 4;
    min += (1<<order) + z->protection[alloc_type];
    if (z->free_pages < min)
        continue;
    page = buffered_rmqueue(z, order, gfp_mask);
    if (page)
        goto got_pg;
}

```

/*运行至此, 说明页面分配依然是失败的, 如果请求内存分配的进程是“内存分配工作者”, 比如说 kswapd, 这类型进程是为内存分配而工作的, 进程 mm 带有标志 PF_MEMALLOC, 或者进程是在做“out of memory”之类的工作, 那么只要管理区中有内存就分配给它*/

```

if ((p->flags & (PF_MEMALLOC | PF_MEMDIE)) && !in_interrupt()) {
    /* go through the zonelist yet again, ignoring mins */
    for (i = 0; (z = zones[i]) != NULL; i++) {
        page = buffered_rmqueue(z, order, gfp_mask);
        if (page)

```

```

        goto got_pg;
    }
    goto nopage;
}
/* Atomic allocations - we can't balance anything */
if (!wait)
    goto nopage;

```

/*运行至此，说明 zone 区的空闲内存实在太少，调用 try_to_free_pages 进行内存回收。把磁盘缓存区，slab 缓存区中的页面释放，将 inactive_list 中的页面交换至磁盘，然后再回收该项页面*/

rebalance:

```

/* We now go into synchronous reclaim */
p->flags |= PF_MEMALLOC;
reclaim_state.reclaimed_slab = 0;
p->reclaim_state = &reclaim_state;
try_to_free_pages(zones, gfp_mask, order);
p->reclaim_state = NULL;
p->flags &= ~PF_MEMALLOC;
/* go through the zonelist yet one more time */
for (i = 0; (z = zones[i]) != NULL; i++) //回收过后，再请求内存
{
    min = z->pages_min;
    if (gfp_mask & __GFP_HIGH)
        min /= 2;
    if (can_try_harder)
        min -= min / 4;
    min += (1<<order) + z->protection[alloc_type];
    if (z->free_pages < min) & min /= 2;
    if (can_try_harder)
        min -= min / 4;
    min += (1<<order) + z->protection[alloc_type];
    if (z->free_pages < min)

```

```

        continue;
    page = buffered_rmqueue(z,  order,  gfp_mask);
    if (page)
        goto got_pg;
}
do_retry = 0;
if (!(gfp_mask & __GFP_NORETRY))
{
    if ((order <= 3) || (gfp_mask & __GFP_REPEAT))
        do_retry = 1;
    if (gfp_mask & __GFP_NOFAIL)
        do_retry = 1;
}
if (do_retry)
{
    blk_congestion_wait(WRITE,  HZ/50);
    goto rebalance;
}
nopcode:
    if (!(gfp_mask & __GFP_NOWARN) && printk_ratelimit()) {
        printk(KERN_WARNING "%s: page allocation failure."
            " order:%d,  mode:0x%x\n",
            p->comm,  order,  gfp_mask);
        dump_stack();
    }
    return NULL;
got_pg:
    zone_statistics(zonelist,  z);
    kernel_map_pages(page,  1 << order,  1);
    return page;
}

```

从上面的分配过程，可以看到整个分配过程非常繁杂，不过在大部份情况下，第一次循环就应该可以请求到内存。至此对 alloc_pages 的轮廓已经有大致了解，从上面的代码可以看到，从相应的 zone 分配内存的接口是 buffered_rmqueue()。其代码如下：

```
static struct page *
buffered_rmqueue(struct zone *zone, int order, int gfp_flags)
{
    unsigned long flags;
    struct page *page = NULL;
    int cold = !(gfp_flags & __GFP_COLD);
    if (order == 0) //单页面的分配，每个 cpu 都维持着一个”冷”，”热”页
        //面的内存池
    {
        struct per_cpu_pages *pcp; //取得当前 cpu 对应的 pcp
        pcp = &zone->pageset[get_cpu()].pcp[cold];
        local_irq_save(flags);
        if (pcp->count <= pcp->low) //如果剩余页低于指定的数值，就从 zone
            //中请求一大块内存，将之放入 pcp
            pcp->count += rmqueue_bulk(zone, 0, pcp->batch, &pcp->list);
        if (pcp->count) //如果有内存剩余，直接从 pcp 中取从页面即可
        {
            page = list_entry(pcp->list.next, struct page, lru);
            list_del(&page->lru);
            pcp->count--;
        }
        local_irq_restore(flags);
        put_cpu();
    }
    if (page == NULL) //多页面分配，或者是单页面的上述分配过程失败
    {
        spin_lock_irqsave(&zone->lock, flags);
        page = __rmqueue(zone, order);
    }
}
```

```

        spin_unlock_irqrestore(&zone->lock, flags);
    }
    if (page != NULL)
    {
        BUG_ON(bad_range(zone, page));
        mod_page_state_zone(zone, pgalloc, 1 << order);
        prep_new_page(page, order);
        if (order && (gfp_flags & __GFP_COMP))
            prep_compound_page(page, order);
    }
    return page;
}

```

pcp 结构是 2.6 中新加入的 per-cpu 结点, 据统计, 冷热页面缓存区的存在, 使整个系统效率提高了 17%。_rmqueue() 是从相应 zone 中取得多页面的操作, 它是整个过程的核心代码, 代码如下:

```

static struct page *_rmqueue(struct zone *zone, unsigned int order)
{
    struct free_area * area;
    unsigned int current_order;
    struct page *page;
    unsigned int index;
    for (current_order = order; current_order < MAX_ORDER;
        ++current_order) //从 zone 中取得相应大小的 free_area
    {
        area = zone->free_area + current_order; //如果 free_area 为空, 则
        从下一个空闲区分配
        if (list_empty(&area->free_list))
            continue;
        page = list_entry(area->free_list.next, struct page, lru); //
        从空闲区中分得内存
        list_del(&page->lru); //脱链
    }
}

```

```

        index = page - zone->zone_mem_map; //对应页面对 zone 的 page 数组
            中的序号
        if (current_order != MAX_ORDER-1)
            MARK_USED(index, current_order, area); //更新分配位图
        zone->free_pages -= 1UL << order; //更新 zone 空闲页面计数
        return expand(zone, page, index, order, current_order, area);
    }
    return NULL;
}

static inline struct page *
expand(struct zone *zone, struct page *page,
        unsigned long index, int low, int high, struct free_area *area)
{
    unsigned long size = 1 << high; //求得总的内框个数
    while (high > low)
    {
        area--;
        high--;
        size >>= 1;
        BUG_ON(bad_range(zone, &page[size]));
        list_add(&page[size].lru, &area->free_list);
        MARK_USED(index + size, high, area); //更正空闲区的页面位图
    }
    return page;
}

```

函数 `expand()` 把剩余的内存归还给下一个空闲区。其中参数的含义如下：

zone: 管理区

page: 要归还内存的起始内框 page

index: 在管理区中的序号

low: 欲分配的内框大小

high: 已经分配的内框大小

area: 空闲区

这段代码比较晦涩，用下图表示操作过程：

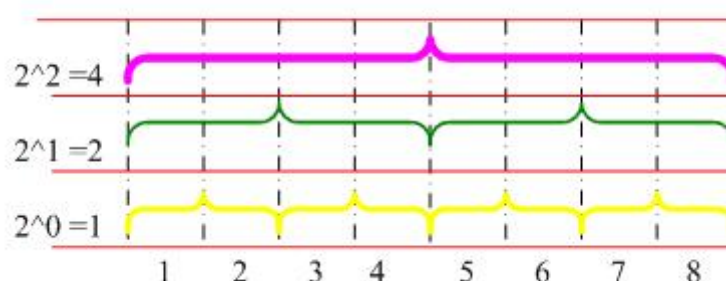


图 4-5 页面位图

如图 4-5 所示，每一个框表示一个页面，大括号表示相应的位图。以下需要考虑几个问题。

第一，如何知道页面在相应空闲所在的分配位图位。结合前面讨论的伙伴系统分配位图知道，空闲区的每一组用一个位来表示其分配情况，所以，在 2^n 的空闲链表中，每一个空闲块是 2^n 大小，每个位图表示 2×2^n 个页。在 zone 中序号为 index 的页在 2^n 空闲链中相应的位图是 $\text{index} / 2 \times 2^n = \text{index} \gg (n+1)$ 。如上图中 2^0 空闲链中，1, 2 属于第一个位，3, 4 属于第二个位。由此我们就可以分析出专门用来改变位图相应位的宏应为：

```
#define MARK_USED(index, order, area) \
    _change_bit((index) >> (1+(order)), (area)->map)
```

其中 index 是页面的序号， 2^{order} 即为 area 每一个空闲块的大小。这个宏对页对应的位的值进行了改变 (0 变成 1，1 变成 0)

第二，在一个大空闲中请求一个小空闲区，这时，大空闲的剩余部份该如何并入到小的空闲链表。如上图，如果要请求 2 个页面，首先它会到 2^1 中寻找有没有空闲页面。如果有，正好从这个链表中分配；如果没有，则要到 2^2 这条链表中分配，假设 2^2 中的第 5 到 8 是空闲的，这时会把 5—8 这四个页面从 2^2 链中脱落，更新 2^2 的相应分配位图。然后，再把它的一半加入 2^1 链，并更新 2^1 链的位。总之是按照半数递减的关系放入前一个链表。因为空闲链是按 2 的倍数增加的，所以按半数递减可以把剩余内存尽量地分成大内存。

第三，剩余内存归入到低链表的时候，底链表会不会合成大的空闲块？如上例中：将 4 页面放入到 2^2 链的时候，有没有机会合成一个 8 页面空闲块？分析后不难

得出: $2^{(n+1)}$ 每个空闲块刚好可以表示 2^n 位图中所表示的两个空闲块。例如上例中 2^1 的第一个块,就是 2^0 的第位图第一个位所表示的两个空闲块。结合前面所讨论的内容,只有同一个位图位所表示的两个空闲块都空闲的时候,才能将它合并。把剩余内存归入前一空闲链的时候,不可能满足上面所讲的要求,所以,不可能在低链上合并成高内存。

4.2.3 free_pages() 的相关实现

```
fastcall void __free_pages(struct page *page, unsigned int order)
{
    if (!PageReserved(page) && put_page_testzero(page))
    {
        //PageReserved(page):页面被保留或者没有分配出去
        //put_page_testzero(page):把页面的引用计数减1,并判断是否为零
        if (order == 0) //这个页没有被使用了,可能释放到伙伴系统了
            //单页面的释放
            free_hot_page(page);
        else
            //多页面的释放
            __free_pages_ok(page, order);
    }
}
```

如前面所讲述的那样,为了提高内存管理的效率,每个cpu维持了一个单页面的缓冲区.所以当释放一个单面的时候,把它释放到缓存池就好了。

先看一下单页面释放的代码:

```
void fastcall free_hot_page(struct page *page)
{
    free_hot_cold_page(page, 0);
}

static void fastcall free_hot_cold_page(struct page *page, int cold)
{
    struct zone *zone = page_zone(page); //page 所在 zone
    struct per_cpu_pages *pcp;
```



```

unsigned long flags;
arch_free_page(page, 0); //只有配置了 HAVE_ARCH_FREE_PAGE 才有效
kernel_map_pages(page, 1, 0); //只有配置了 CONFIG_DEBUG_PAGEALLOC
才有效
inc_page_state(pgfree);
if (PageAnon(page))
    page->mapping = NULL;
free_pages_check(_ _FUNCTION_ _, page);
pcp = &zone->pageset[get_cpu()].pcp[cold]; //得到 pcp
local_irq_save(flags);
if (pcp->count >= pcp->high)
    pcp->count -= free_pages_bulk(zone, pcp->batch, &pcp->list, 0);
//如果剩余量超过了允许的最大值, 把它释放到伙伴系统
list_add(&page->lru, &pcp->list); //将页面链入
pcp->count++; //更新计数
local_irq_restore(flags);
put_cpu();
}

```

当 pcp 中缓存的页面超过了最大值, 需要将 pcp->batch 个页面释放到伙伴系统。再看多页面的释放操作, 调用 `_ _free_pages_ok()`, 其代码如下:

```

void _ _free_pages_ok(struct page *page, unsigned int order)
{
    //释放的起始页面:page
    //页面个数:2order
    LIST_HEAD(list); //声明并初始化一个链表
    int i;
    arch_free_page(page, order);
    mod_page_state(pgfree, 1 << order);
    for (i = 0 ; i < (1 << order) ; ++i)
        free_pages_check(_ _FUNCTION_ _, page + i);
    list_add(&page->lru, &list);
    kernel_map_pages(page, 1<<order, 0);
    free_pages_bulk(page_zone(page), 1, &list, order);
}

```

```
}
```

它的核心处理都是在函数 `free_pages_bulk()` 中完成的。下面分析该函数。

```
static int free_pages_bulk(struct zone *zone, int count, struct list_head
*list, unsigned int order)
{ //count:释放的次数; order:每次释放  $2^{\text{order}}$  个页面。例如:要释放 x 个单页面,
//count=x, order=0, 释放  $2^x$  大小的连续页面 count=1, order=x
    unsigned long flags;
    struct free_area *area;
    struct page *base, *page = NULL;
    int ret = 0;
    base = zone->zone_mem_map;
    area = zone->free_area + order; //找到要释放大小页面的空闲链
    spin_lock_irqsave(&zone->lock, flags);
    zone->all_unreclaimable = 0;
    zone->pages_scanned = 0;
    while (!list_empty(list) && count--) //如果页面释放完了, 则循环结束
    {
        page = list_entry(list->prev, struct page, lru);
        /* have to delete it as __free_pages_bulk list manipulates */
        list_del(&page->lru);
        __free_pages_bulk(page, base, zone, area, order);
        ret++;
    }
    spin_unlock_irqrestore(&zone->lock, flags);
    return ret;
}
```

再看调用的 `__free_pages_bulk()` 函数, 其参数含义如下:

page:要释放的起始 page

base:对应 zone 的页数组的首地址

area:将页面释放到此空闲区

order:释放的连续空闲区大小

其代码如下:

```

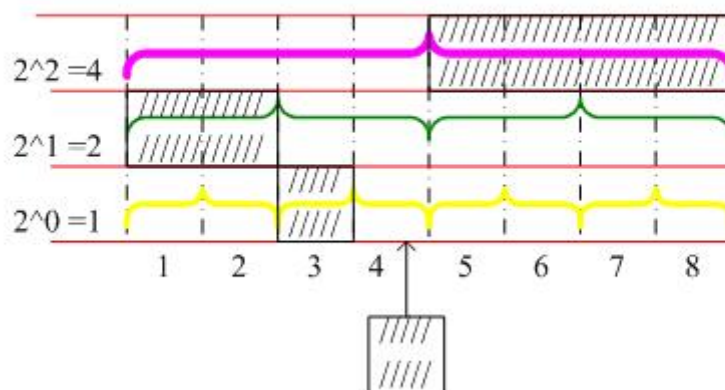
static inline void __free_pages_bulk (struct page *page, struct page
*base, struct zone *zone, struct free_area * area, unsigned int order)
{
    unsigned long page_idx, index, mask;
    if (order)
        destroy_compound_page(page, order);
    mask = (~0UL) << order; //假设 order = 3, mask = 1111 1000
    page_idx = page - base; //取得页在 zone 中页数组中的序号
    if (page_idx & ~mask)
        /*~mask:0000 0111, 判断 page_idx 是不是 order 位对齐的, 结合前面 alloc
所分析的, 不难得出在  $2^{\text{order}}$  大小空闲区中, 空闲块的首地址必须是  $2^{\text{order}}$ 
的倍数*/
        BUG();
    index = page_idx >> (1 + order); //得到 page 在链表中的分配位图对应位
    zone->free_pages += 1 << order; //更新 zone 的空闲区统计计数
    while (order < MAX_ORDER-1) //循环, 合并内存
    {
        struct page *buddy1, *buddy2;
        BUG_ON(area >= zone->free_area + MAX_ORDER); //判断相邻块是否空
        //闲。为 0 的时候, 是两者都空闲或两者都已分配
        if (!_test_and_change_bit(index, area->map))
            break;
        /* Move the buddy up one level. */
        buddy1 = base + (page_idx ^ (1 << order)); //它的邻居块
        buddy2 = base + page_idx; //它自己
        BUG_ON(bad_range(zone, buddy1)); //判断是否超过 zone 所允许的 page
        BUG_ON(bad_range(zone, buddy2)); //范围
        list_del(&buddy1->lru); //将其从现有的空闲链中脱落
        mask <<= 1; //到它的上一级, 判断是否有能合并的内存块
        order++;
        area++;
        index >>= 1;
    }
}

```

```

    page_idx &= mask;
}
list_add(&(base + page_idx)->lru, &area->free_list);

```



}

图 4-6 页面释放

如图 4-6 所示, 标记为阴影的代表空闲块, 此时系统将第 4 个单页表释放到伙伴系统。首先, 它会到相应大小的 (2^0) 的空闲链表入队。判断相邻块的空闲情况, 此时, 因为第三个是空闲的, 所以可以合并为一块 2^1 的空闲块。继续判断 2^1 链中相邻块是否是空闲的, 上图中可以将 2^1 的两个块合成一个大块, 然后继续判断 2^2 中是否可以继续合并, 依次类推。上述代码中, 涉及到几个位操作, 分析如下:

第一, 如何得到相邻块的起始 page。如图 4-7 示:

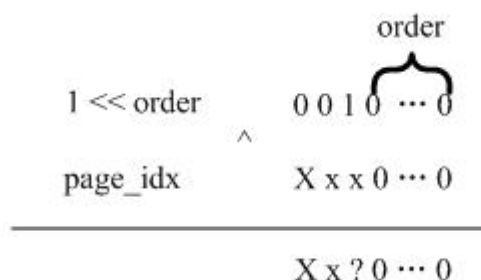


图 4-7 位操作

```
buddy1 = base + (page_idx ^ (1 << order))
```

根据上面的分析, 可得知 `page_idx` 本身就是 `order` 位对齐的, 所以, 它的低 `order` 位为零。此外, 再根据 0 与任何数异或值不变, 1 与数异或都相反的规律, 可以得知, 位运算结果只跟 `order+1` 位有关。据此就可以计算出它的“伙伴块”

第二，如何得到高一级空闲链的首空闲块序号：

```
mask <<= 1; page_idx &= mask;
```

只需按着高一级链表的 order 位对齐即可。

第三，如何得到空闲块在高一级链表中对应的分配位图位。

```
index >>= 1;
```

在前面分析过对应位的计算方法，在高一级空闲链中的位对应当前除二。其实这一个过程在操作系统设计中也叫“内存拼凑”，就是把剩余小内存，拼成连续的大内存，以满足某些程序的需要。

4.3 伙伴算法的不足

尽管伙伴算法非常经典，但是也存在缺陷。比如它的合并要求过于严格，只能是满足伙伴关系的块。第 1 和第 2 个块虽然相邻但是由于它们不能满足伙伴关系却不能合并。伙伴算法还容易产生碎片，根据算法，如果在一片连续的内存中如果有一个页面被占用，那么有可能将导致整个内存区都不具备合并条件。

伙伴算法中还存在浪费现象。由于伙伴算法是按 2 的幂次方分配内存区的，当需要 513 (2^9+1) 个页面时，就不得不申请 2^{10} 的页面。于是就有 511 个页面被浪费掉了。

伙伴算法的效率问题也值得商榷。伙伴算法涉及了比较多的计算还有链表和位图的操作，开销巨大，如果每次 2^n 大小的伙伴块就会合并到 2^{n+1} 的链表队列中，那么 2^n 大小链表中的块就会因为合并操作而减少，但系统随后立即有可能又有对该大小块的需求，为此必须再从 2^{n+1} 大小的链表中拆分，这样的合并又立即拆分的过程是无效率的。

4.4 Linux 伙伴算法的改进

4.4.1 主要数据结构的改进

内存管理器对于内存的管理基本单位是页，每一个物理页的大小是 4K。系统中所有的物理页用一个 mem_map 数组描述，这是一个 mem_map_t 结构的数组，该数组的大小由系统中实际物理内存的大小决定。每一个 mem_map_t 结构描述系统中的一个物理页面。在系统启动时，对 mem_map 数组每一个元素进行初始化，标

示出了系统中所有可用的物理页。由于 mem_map 是 page 结构指针，操作的结果也是个 page 结构指针。

```
struct page
{
    struct list_head list;
    struct address_space * mapping;
    struct node * inode;
    unsigned long offset;
    atomic_t count;
    unsigned long age;
    .....
}mem_map_t;
```

在该数据结构中使用到了数据结构 list_head，它是一个通用的双向链表，将整个内存中的所有页面链接到一起。inode 和 offset 用来表示当前物理页面所存储的文件的节点号和文件中的位置。两者合起来可以唯一地标识一个文件中的页。属于同一个 inode 的所有页，由数据结构 list_head 建立起一个双向链表，count 是一个引用计数。表示物理页在系统中出现的次数。如果物理页正在使用，其 count 值大于或是等于 1。flag 是一个标志域，表示页的状态。当页面的内容来自一个文件时，index 代表着该页面在文件中的序号。当页面的内容被换出到交换设备上，但还保留着内容作为缓冲时，则 index 指明了页面的去向。

在内存管理中的内存分配和释放最常见的方法就是将位图和链表相结合使用。位图的作用是跟踪并记录内存单元的使用情况。可以通过位图的检测来简化对内存的分配。位图中的每一位表示一个页的使用情况。如果分配 10 页内存内存管理器只需要在位图中找到 10 个连续的 1 即可。对于物理内存很大的设备中，它的位图也会相应的很大，并且经常会出现跨字节边界操作，因此仅仅使用位图并不是最理想的管理方法。

链表方式则是使用数据结构来表示内存单元，并可分别建立分配内存和空闲内存的链表。通过这些链表可以方便地完成内存的分配和回收。这种方法比位图更加灵活，也更加有效。

Linux 内存管理器采用的是将链表和位图相结合的方法。Linux 内核定义了一个 free_area 的数组。该数组按照大小记录系统中的所有的连续空闲页块。在 free_area[0] 中排列着大小为 1 页的空闲页，在 free_area[1] 中排列的是大小为

2 页的空闲页，在 free_area[2]中排列着大小为 4 页的空闲页，依此类推页面的大小按照 2 的指数增加。

```
free_area 数组定义如下：
struct free_area_struct{
    struct list_head free_list;
    unsigned int* map;
};
struct free_area_struct free_area[NR_MEM_LISTS];
```

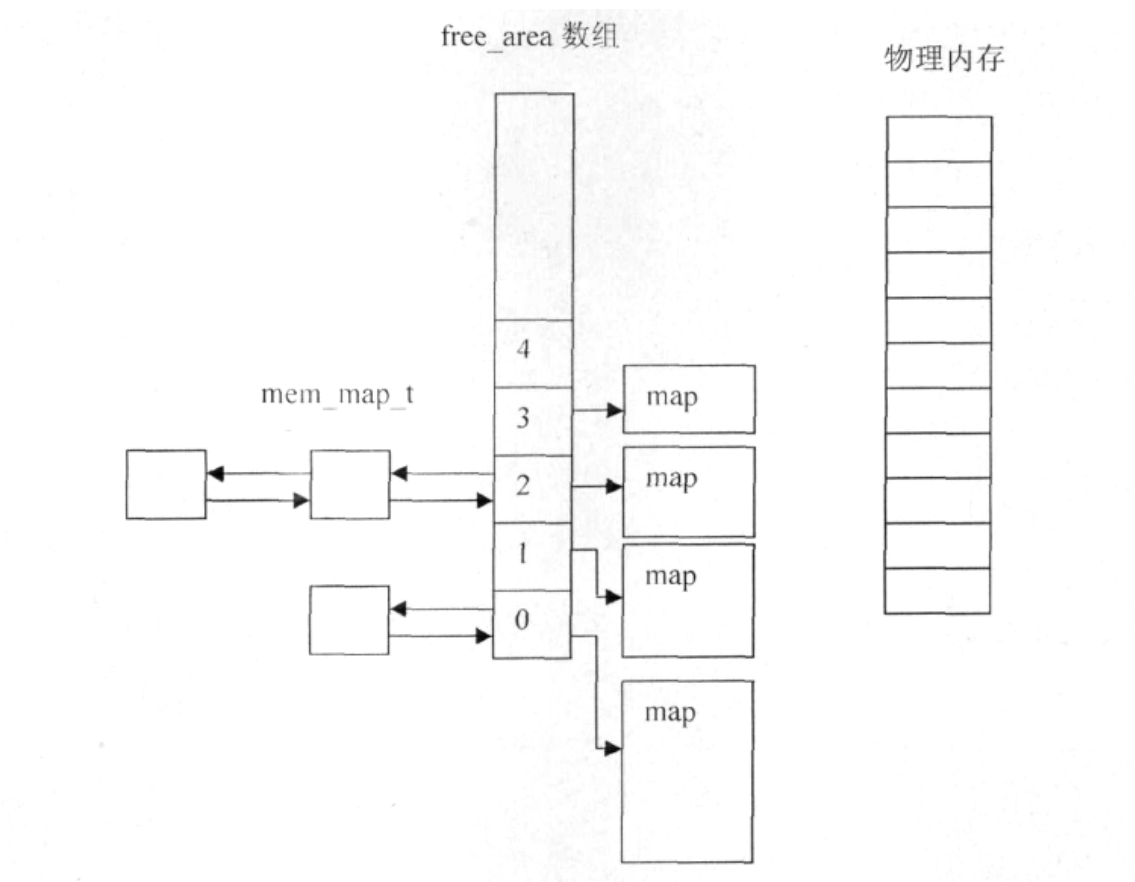


图 4-8 伙伴算法的空闲内存组织形式

数组 free_area 的每一项包括 2 个元素, free_list 和指针 map。在 free_list 中包含有两个指针 next 和 prev 用于将物理页结构 page 连成一个双向链表, 而 map 则是一个指向位图的指针, 该位图用于记录页块及其伙伴是否在列表中, 两个伙伴合用位图中的一位。在位图中, 两伙伴合用位的规则如下:

两个伙伴全都不在队列中(已分配或已组成了一个更大的块)，它们合用的位为 0。有一个在队列中，一个不在队列中，则它们合用的位为 1。两个都在队列中，则它们应该合并成更大的块，加入到 `free_area` 数组中更上面的队列，所以它们合用的位应为 0。

在 `free_area` 数组的最大一个元素中不使用位图，因为最大的元素不存在合并的问题，因此其位图也不再起作用。在 `freearea` 中的每一项的位图都要表示整个物理内存的使用情况，因此每一项的位图的大小各不相同，页块越小位图越大。

物理内存管理器所管理的内存最小为 1 页(4KB)，最大的内存块为 512 页(2MB) 内存管理器使用 `free_area` 来分配和回收物理页系统初始化时已经对数组 `free_area` 数组进行初始化工一作。全局变量 `nr_free_pages` 记录系统中当前的空闲页数空闲链表组织形式如图 4-8 所示。

伙伴算法对于伙伴关系的定义过于严格。要求内存块满足以下 3 个条件：

- (1)两个内存块的大小相等。
- (2)两个内存块的物理地址连续。
- (3)两个内存块出自同一个更大的内存块。

如果在系统中存在只满足大小相等、物理地址连续这两个条件，而不是出自同一个更大的内存块时伙伴系统并不会对于这样的空闲内存进行管理和维护，因此在无法找到空闲内存时也不会来查看这样的空闲内存。在系统内存使用量比较大而不存在符合伙伴关系的连续内存时系统会认为内存分配失败。而实际上系统中存在连续的足够大的内存块，只是伙伴系统没有办法识别出来这样的连续的内存因而无法将内存块分配给进程。

对于上面提到的问题可以定义一种类似于原有伙伴关系的新的关系，符合这的关系的两个内存块只需要满足大小相等的、地址连续这两个条件即可，称这种关系为放宽的伙伴关系，满足这样关系的两个内存块互称为放宽的伙伴块。为了这样的放宽的伙伴块或类伙伴关系，使用另一个链表 `new_free_list` 和一张位图 `new_map` 来进行管理。它的管理思想和原伙伴算法的管理思想类似，通过位图示放宽的伙伴块是否空闲链表 `free_list` 中，如果在则将这两块内存块进行合并，将合并成的大的内存块链接到在新增加链表 `new_free_list` 上。合并完成后同样修改 `new-map` 所指向的新的位图。改进后的 `free_area_struct` 数据结构为：

```
typedef struct free_area_struct
{
    struct list_head free_list;
```



```

struct list_head new_free_list;
unsigned int* map;
unsigned int* new_map;
}free_area_t;

```

其结构如图 4-9 所示。

为了更加详细地了解对于数据结构 `free_area_struct` 的修改，有必要对两个空闲链表和位图指针进行一些比较。在修改后的 `free_area_t` 中定义了两个空闲内存链表，分别为 `free_list` 和 `new_free_list` 这两个空闲内存链表都是 `list_head` 数据结构，两者之间的不同是链接的空闲内存块不同。首先，在 `free_list` 上链接着原伙伴算法中的空闲内存块，对于 `free_area_t[n]` 中的空闲

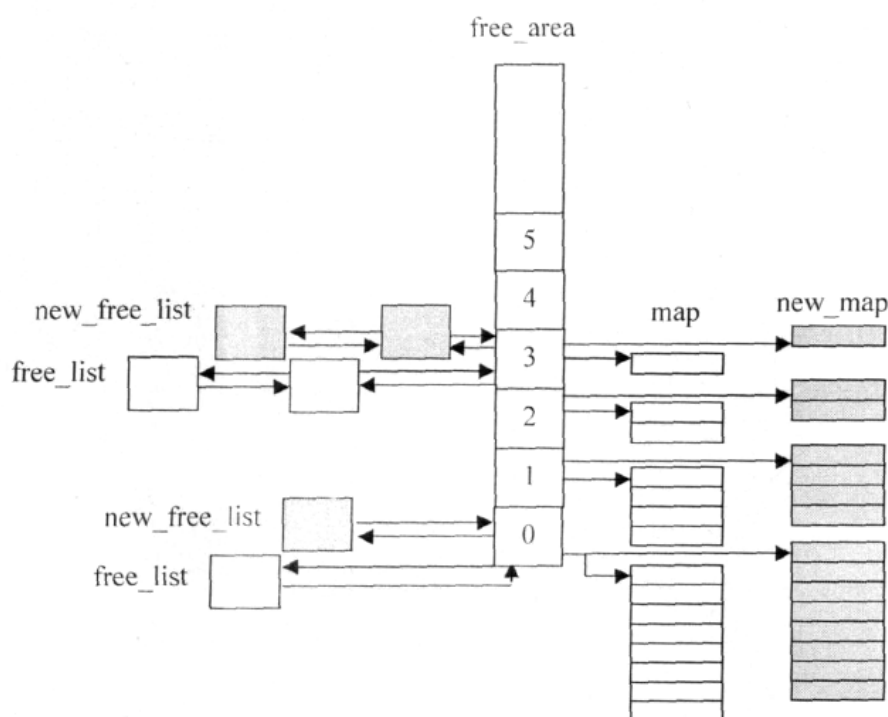


图 4-9 修改后的空闲内存组织形式

内存块是由两个 `free_area_t[n-1]` 中的空闲内存块合并而成的。

这两个内存块满足伙伴定义的两个条件。而在 `free_area_t[n]` 中的 `new_free_list` 中的内存块也是由两个 `free_area_t[n-1]` 中 `free_list` 的两个内存块合并而成，但是这两个内存块只满足大小相等、物理地址连续这两个条件。其次是对这两个空闲链表中的内存块处理方法的不同。在 `free_list` 中的内存

块进行合并时会将用来合并的两个内存块先从链表中删除，然后再将合并成的大内存块添加到上一级的 `free_list` 中，而在 `new_free_list` 中的内存块在合并时并不会将这两个内存块从空闲链表中删除而是继续在 `free_list` 中保留着两个内存块，同时在 `new_free_list` 中再次添加由这两个内存块合并而成的大的内存块。最后，在 `free_list` 中的内存块被使用时不会影响到它的上一层的 `free_list` 中的内存块，它只是简单地将空闲链表中的内存块从链表中摘下来分配给进程使用，而在 `free_list` 中某一个内存块被分配出去有可能会影响到 `new_free_list` 中的空闲内存块，因为在 `free_list[n-1]` 和 `new_free_list[n]` 中可能对于同一个内存块存在着以不同的形式表示，因此会存在分配 `free_list[n-1]` 中的内存块是 `new_free_list` 中的某个内存块的一部分，这时就需要对于 `new_free_list` 中的内存块从链表中摘下来保持系统对于内存管理的正确性。

对于 `free_area_t` 中的两个位图指针所指向的位图的操作也是不同的。这两个位图的作用相同，都是用来表示内存的各个伙伴块的使用情况。但是对于同一个内存块这两个位图表示的伙伴块不同，一个表示内存的原伙伴块，另一个表示的是放宽的伙伴块。在原伙伴算法中的两个伙伴块的起始地址只有一位不同，这样可以通过一个简单的位运算就可以找到某个内存块的伙伴块。而在放宽的伙伴关系中，互为放宽伙伴关系的两个内存块的起始地址有两位不同，因此需要对位图的操作更为复杂一些。

4.4.2 内存分配函数的改进

在 Linux 中我们可以通过 6 个稍有差别的函数和宏申请页内存。一般情况下，它们都返回第一个所分配页的线性地址，或者分配失败则返回 `NULL`。

`alloc_pages(gfp_mask, order)`：用这个函数请求 2^{order} 个连续的页面。它返回第一个所分配页面描述符的地址，或者如果失败，则返回 `NULL`。

`alloc_page(gfp_mask)`：用于获得一个单独页面的宏，它其实只是 `alloc_pages(gfp_mask, 0)`。它返回所分配页面描述符的地址，或者如果分配失败，则返回 `NULL`。

`_get_free_pages(gfp_mask, order)`：该函数类似于 `alloc_pages()`，只不过它返回第一个所分配页对应的内存线性地址。

`_get_free_page(gfp_mask)`：用于获得一个单独页面的宏，它也只是 `_get_free_pages(gfp_mask, 0)`

`get_zeroed_page(gfp_mask)`：函数用来获取满是 0 的页面，它调用 `alloc_pages(gfp_mask | __GFP_ZERO, 0)`，然后返回所获取页面的线性地址。

`__get_dma_pages(gfp_mask, order)`：该宏获取用于 DMA 的页面，它扩展调用 `__get_free_pages(gfp_mask | __GFP_DMA, order)`。

参数 `gfp_mask` 是一组标志，它指明了如何寻找空闲的页面，如表 4.1 所示。

标志	说明
<code>__GFP_DMA</code>	所请求的页框必须处于 ZONE_DMA 管理区。
<code>__GFP_HIGHMEM</code>	所请求的页框处于 ZONE_HIGHMEM 管理区。
<code>__GFP_WAIT</code>	允许内核对等待空闲页框的当前进程进行阻塞。
<code>__GFP_HIGH</code>	允许内核访问保留的页框池。
<code>__GFP_IO</code>	允许内核在低端内存页上执行 I/O 传输以释放页框。
<code>__GFP_FS</code>	如果清 0，则不允许内核执行依赖于文件系统的操作。
<code>__GFP_COLD</code>	所请求的页框可能为“冷的”。
<code>__GFP_NOWARN</code>	一次内存分配失败将不会产生警告信息。
<code>__GFP_REPEAT</code>	内核重试内存分配直到成功。
<code>__GFP_NOFAIL</code>	与 <code>__GFP_REPEAT</code> 相同。
<code>__GFP_NORETRY</code>	一次内存分配失败后不再重试。
<code>__GFP_NO_GROW</code>	slab 分配器不允许增大 slab 高速缓存。
<code>__GFP_COMP</code>	属于扩展页的页框。
<code>__GFP_ZERO</code>	任何返回的页框必须被填满 0。

表 4-1 参数 `gfp_mask` 标志

对一组连续页面的每次请求实质上是通过执行 `alloc_pages` 宏来处理的。接着，这个宏又依次调用 `__alloc_pages()` 函数，该函数是分配页面的核心。它接收以下 3 个参数：

`gfp_mask`：在内存分配请求中指定的标志。

`order`：将要分配的一组连续页面数量的对数。

`zonelist`：指向 `zonelist` 数据结构的指针，该数据结构按优先次序描述了适于内存分配的内存管理区。

```

__alloc_pages() 函数概括起来执行以下代码：
for (i = 0; (z=zonelist->zones[i]) != NULL; i++)
{
    if (zone_watermark_ok(z, order, ...))
    {
        page = buffered_rmqueue(z, order, gfp_mask);
        if (page)
            return page;
    }
}

```

__alloc_pages() 函数首先扫描包含在 zonelist 数据结构中的每个内存管理区。

对于每个内存管理区，该函数将空闲页面的个数与一个限定值作比较，该限定值取决于内存分配标志、当前进程的类型以及管理区被函数检查过的次数。

实际上，如果空闲内存不足，那么每个内存管理区一般会被检查几遍，每一遍在所请求的空闲内存最低量的基础上使用更低的限定值扫描。因此前面一段代码在 __alloc_pages() 函数体内被复制了几次，每次变化很小。__alloc_pages() 函数调用 buffered_rmqueue() 函数：它返回第一个被分配的页面的页描述符；如果内存管理区没有所请求大小的一组连续页面，则返回 NULL。

buffered_rmqueue() 函数在指定的内存管理区中分配页面。其参数为内存管理区描述符的地址，请求分配的内存大小的对数 order，以及分配标志 gfp_flags。该函数本质上执行如下操作：

1. 如果 order 等于 0，则使用每 CPU 页面高速缓存，这里咱不讨论；如果 order 不等于 0，则表明请求跨越了几个连续页面，每 CPU 页面高速缓存就不能被使用，函数执行下面步骤：

2. 调用 __rmqueue() 函数从伙伴系统中分配所请求的页面。

3. 如果内存请求得到满足，函数就初始化（第一个）页面的页描述符：清除一些标志，将 private 字段置 0，并将页面引用计数器置 1。此外，如果 gfp_flags 中的 __GFP_ZERO 标志被置位，则函数将被分配的内存区域填充 0。

4. 返回（第一个）页面的页描述符地址，如果内存分配请求失败则返回 NULL。

这里要提一下 zone_watermark_ok() 辅助函数，他的目的就是来探测对应的内存管理区中有没有足够的空闲页面，该函数接收几个参数，它们决定对应内存管

理区 z 中空闲页面个数的限定值 \min 。下面解释一下 `zone_watermark_ok` 同时满足下列两个条件则返回 1 的情况：

1. 除了被分配的页面外，在内存管理区中至少还有 \min 个空闲页面，不包括为内存不足保留的页面（管理区描述符的 `lowmem_reserve` 字段）。

2. 除了被分配的页面外，这里在 order 至少为 k 的块中起码还有 $\min/2^k$ 个空闲页面，其中，对于每个 k ，取值在 1 和分配的 order 之间。因此，如果 order 大于 0，那么在大小至少为 2 的块中起码还有 $\min/2$ 个空闲页面；如果 order 大于 1，那么在大小至少为 4 的块中起码还有 $\min/4$ 个空闲页面；…依此类推。

在内核中，真正的 `_alloc_pages()` 函数是很复杂的，需要结合内核回收页面机制来做分析，它所做的工作步骤如下：

1. 执行对内存管理区的第一次扫描。在第一次扫描中，限定值 \min 被设定为 $z \rightarrow \text{pages_low}$ ，其中的 z 指向正在被分拆的管理区描述符，参数 `can_try_harder` 和 `gfp_high` 被设定为 0。

2. 如果函数在第一步没有终止，就表示没有剩下多少空闲内存，将唤醒 `kswapd` 内核线程来异步地开始回收内存。

3. 执行对内存管理区的第二次扫描，将值 $z \rightarrow \text{pages_min}$ 作为限定值 `base` 传递。标志 `can_try_harder` 和 `gfp_high` 决定了实际的限定值，这一步与第一步类似，不同的是该函数使用了较低的限定值。

4. 如果函数在上一步没有终止，那么可以肯定系统内存不足。如果产生内存分配请求的内核控制路径不是一个中断处理程序或一个可延迟函数，并且它试图回收内存，或者是 `current` 的 `PF_MEMALLOC` 标志被置位，或者是它的 `PF_MEMDIE` 标志被置位，那么函数将随即执行对内存管理区的第三次扫描，试图分配内存并忽略内存不足的限定值，也就是说在这个时候不去调用 `zone_watermark_ok()` 函数。唯有这种情况下才允许内核控制路径耗用为内存不足时预留的页。此时管理区描述符的 `lowmem_reserve` 字段已指定。其实在这种情况下产生内存请求的内核控制路径最终将试图释放页面，因此只要有可能它就应当得到它所请求的。如果没有任何内存管理区包含足够的页，函数就返回 `NULL` 来提示调用者发生了错误。

5. 在这里，正在调用的内核控制路径并不试图回收内存。如果 `gfp_mask` 的 `_GFP_WAIT` 标志没有被置位，函数就返回 `NULL` 来提示该内核控制路径内存分配失败，在这种情况下，如果不阻塞当前进程就无法满足要求。

6. 在这里当前进程能够被阻塞，调用 `cond_resched()` 检查是否有其它的进程需要 CPU。

7. 设置 `current` 的 `PF_MEMALLOC` 标志来表示进程已经准备好执行内存回收。

8. 将一个指向 `reclaim_state` 数据结构的指针存入 `current->reclaim_state`。这个数据结构只包含一个字段 `reclaimed_slab`，被初始化为 0。

9. 调用 `try_to_free_pages()` 函数寻找一些页面来回收。该函数有可能阻塞当前进程。一旦函数返回，`_alloc_pages()` 就重设 `current` 的 `PF_MEMALLOC` 标志并再次调用 `cond_resched()` 函数。

10. 如果在上一步中已经释放了一些页面，那么该函数还要执行一次与第三步相同的操作，即扫描内存管理区。如果 `_GFP_NOFAIL` 标志被清除，并且内存分配请求跨越超过 8 个页面或者是 `_GFP_REPEAT` 和 `_GFP_NOFAIL` 标志其中之一被置位，那么函数就调用 `blk_congestion_wait()` 使进程休眠一会儿，并且跳回第 6 步。否则，函数返回 `NULL` 来提示调用者内存分配失败。

11. 如果在第 9 步中没有释放任何的页面，这就表示内核遇到了大问题，因为空闲页面已经少到了危险的境地，并且不可能回收任何页面，这就到了该作出重要决定的时刻。当 `gfp_mask` 中的 `_GFP_FS` 标志被置位，即允许内核控制路径执行依赖于文件系统的操作来杀死一个进程，并且 `_GFP_NORETRY` 标志为 0，那么执行如下的步骤：

- a. 使用等于 `z->pages_high` 的限定值再一次扫描内存管理区。
- b. 调用 `out_of_memory()`，通过杀死一个进程开始释放一些内存。
- c. 跳回到第 1 步。

由于第 11. a 步使用的限定值远比前面扫描时使用的限定值要高，所以这个步骤很容易失败。实际上，只有当另一个内核控制路径已经杀死一个进程回收它的内存后，第 11. a 步才会成功执行。因此第 11. a 步避免了两个无辜的进程被杀死。

修改后的内存块分配函数会仍然使用原分配函数中对于空闲内存块的搜索过程，只是在空闲链表中无法找到合适的内存块之后并不是直接调用函数 `_alloc_page_limit()`，而是继续查找新增加的空闲链表 `new_free_list`。在这个空闲链表中链接着放宽限制的伙伴块，从 `new_free_list` 中进行查找。这时只要查找 `new_free_list[5]` 就可以，如果其中不存在空闲内存块的话就可以直接调用 `_alloc_page_limit()` 函数。在查找 `new_free_list` 时不需要在查找更大的链表。因为更大的 `new_free_list` 也是有 `free_list` 中的空闲内存块合并而成，如果在 `free_list[5]` 中没有空闲内存块的话在 `new_free_list[6]` 中也不会有元素存在，因此没有必要对于更大的 `new_free_list` 进行查找。搜索顺序对比如图 4-10 所示。

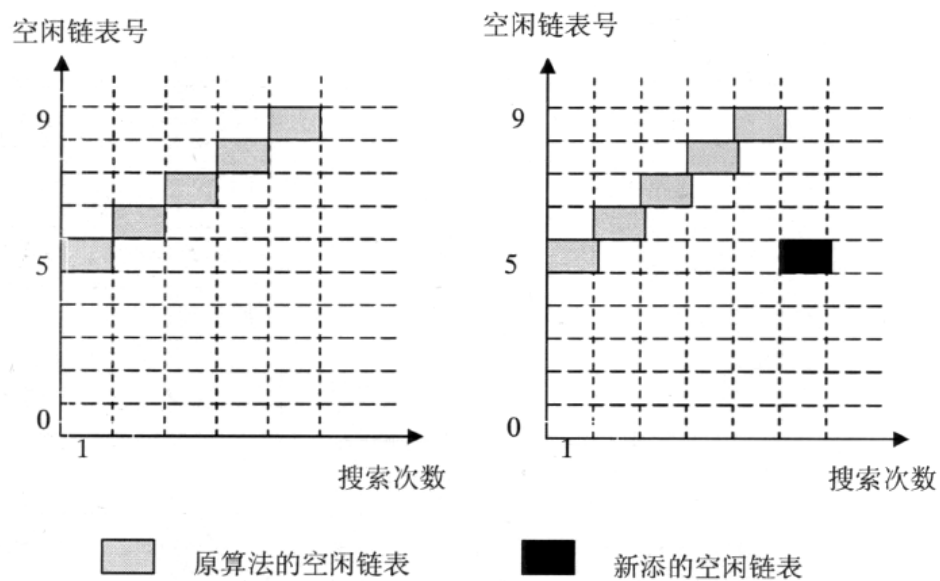


图 4-10 改进后的空闲链表搜索顺序

修改后的内存分配函数 `_alloc_pages(int gfp_mask, unsigned long order)` 流程图如图 4-11 所示。

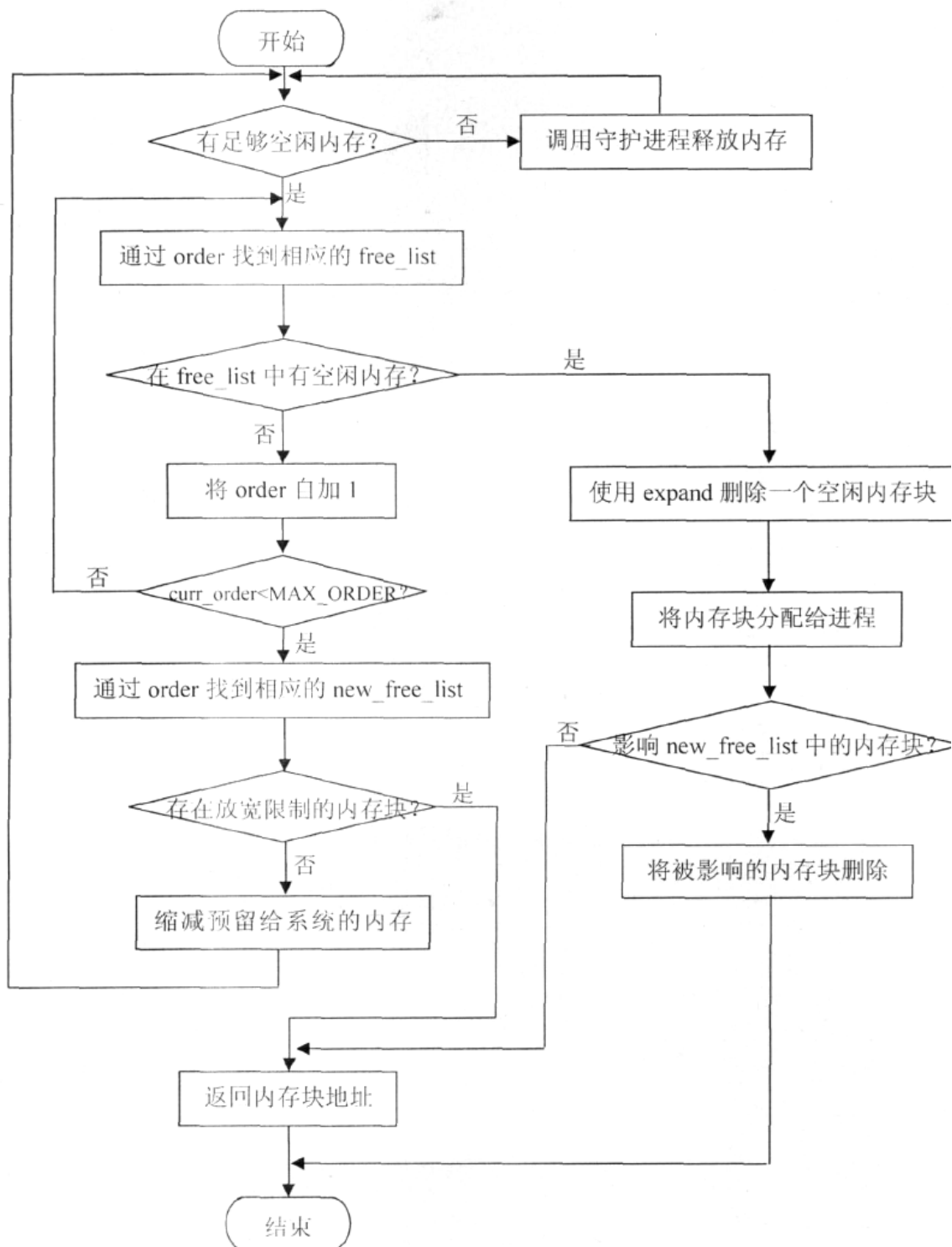


图 4-11 改进后的内存分配函数流程图

伙伴系统会首先使用原有的算法对内存进行分配，如果在 `free_list` 中查找

到存在空闲内存块时，会将查找到的空闲内存块分配给进程。然后查看分配出去的内存块是否在 `new_free_list` 中以更大内存块的形式出现。如果在 `new_free_list` 中存在更大的内存块包括该块内存则直接将 `new_free_list` 中的内存块删除如果在 `free_list` 中无法找到合适的内存块时会查看空闲链表 `newfreehst` 中是否有内存块可以分配给进程，如果仍然不存在就会退出 `_alloc_pages()`。

4.4.3 内存释放函数的改进

释放页框的所有内核宏和函数都依赖于 `_free_pages()` 函数。它接收的参数为将要释放的第一个页框的页描述符的地址 (`page`) 和将要释放的一组连续页框的数量的对数 (`order`)。该函数执行如下步骤：

1. 检查第一个页框是否真正属于动态内存（它的 `PG_reserved` 标志被清 0）；如果不是，则终止。
2. 减少 `page->_count` 使用计数器的值；如果它仍然大于或等于 0，则终止。
3. 如果 `order` 等于 0，那么该函数调用 `free_hot_page()` 来释放页框给适当内存管理区的每 CPU 热高速缓存。
4. 如果 `order` 大于 0，那么它将页框加入到本地链表中，并调用 `free_pages_bulk()` 函数把它们释放到适当内存管理区的伙伴系统中。

页释放分配页块的过程中会将大的页块划分为小的页块，这将导致内存块越来越小。为了不使内存过分零散，因此在释放过程中必须尽可能地合并页块。事实上，页的释放过程就把小页块合并成大页块的过程。

函数 `free_pages_ok()` 用于释放页块，其定义如下：

```
void free_pages_ok(unsigned long addr, unsigned long order);
```

这里：`addr` 是要释放页块的首地址；`order` 表示要释放的页块的大小，被释放的内存块为 2 的 `order` 次幂个物理页。该函数所做的工作如下：

根据页块的首地址 `addr` 算出该页块的第一页在 `mem_map` 数组中的索引 `map-nr`。

(1) 如果该页是保留的（内核在使用），不允许释放则返回。

(2) 将该页块第一页相应的 `mem_map_t` 结构中的 `count` 减 1，表示该页的使用者减少一个。如果 `count` 域的值不足 0，说明它还有别的使用者，因此不能释放它，返回。

(3) 页的引用计数为 0，释放该页。清除页块第一页对应的 mem_map_t 结构的 flag 域中的 PG_REFERENCED 位，表示该页块不再被引用。

(4) 调整全局变量 nr_free_pages，将其值加上回收的物理页数。

(5) 将页块加入到数组 free_area[order] 的相应队列中。

在该程序中对于释放的内存块查找其相应的伙伴块，如果存在则进行合并。合并后的内存块会作为新的一个待合并对象来进行再次合并直至没有其对应的伙伴块或是合并到 n=9 为止。

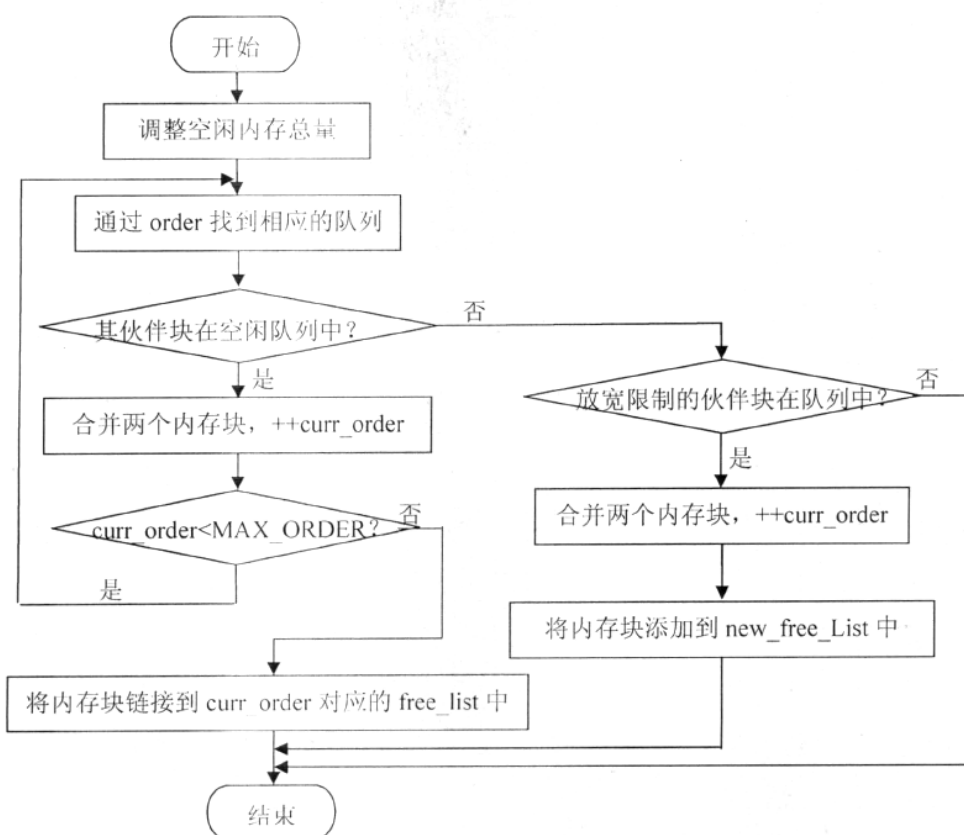


图 4-12 改进后的内存合并函数流程图

改进后的伙伴算法在内存块释放函数中除了查看原有伙伴算法中的位图来判断其相应的伙伴内存块是否在空闲链表。同时在另一张位图表示符合新的伙伴关系的内存块是否在空闲链表中。当一个内存块被释放时，例如 n 值为 5 的内存块，伙伴算法会首先查看 free_area[5] 中的位图 map 看其伙伴块是否在空闲链表 free_list 中，如果存在则与释放的内存块进行合并。合并后的内存块会重新作为

一个新的释放的内存块添加到空闲链表中，如果在 map 中查找不到其对应的伙伴块就会直接将这个内存块添加到空闲内存链表中并且同时修改其在位图中的相应的位的值。在修改完原伙伴算法中的数据结构之后还需要查找该释放的内存块其所在的空闲内存队列中是否有新的伙伴块在其中，如果存在则需要将这两个内存块进行合并，将合并成的内存块链接到 new_free_list[6] 中，而组合成 new_free_list[6] 中的内存块的两块内存并不会因为会有更大的内存块出现在 new_free_list 中而将 free_list[5] 中的内存块删除。这样就有一些内存块会在两个链表中以不同形式同时出现，合并函数流程图如图 4-12 所示。

除了上述改进以外还要在原有算法的基础上进行一些变动。如当内存页号为 1、2、3 的这三个页面被释放时，这三个页面会和原有的空闲内存页面 0 合并成一个起始页面为 0、大小为 4 页的空闲内存块。而这个内存块会进一步和起始页面为 4、大小为 4 的内存块合并形成一个大小为 8 页的内存块链接到 free_area[3] 中。这时就会出现起始页面为 4、大小为 4 页的内存块同时出现在 free_area[3] 的 free_list 和 new_free_list 中。这时就需要重新调整 new_free_list 中的内存块，将 new_free_list 的内存块删除，这样保证了不影响原有伙伴算法能够正常的工作。

在维护数据结构方面同样需要进行些改进，原有的伙伴算法中对于位图的修改和维护使用一个简单的宏来完成。该宏的工作原理是通过内存块的地址索引来找到该内存块在位图中的对应的位置，并且在该位上根据操作的不同来对位图进行不同的修改。

对于一个内存块 block1 起始地址 AR 为 11000001111111000000000000000000，它的 n 值为 6，它的伙伴块的起始地址即 11000001111111000000000000000000。可以通过一个位运算计算得到，计算公式为 $AR \wedge (1 \ll (12+n))$ 。而对于放宽的伙伴块的起始地址的计算公式则不同，它分为两种情况：第一种情况是空闲内存块号最后一位为 0，这时其伙伴块的起始地址为 $AR - 1 \ll (12+n)$ ；第二种情况是空闲内存块号的最后一位为 1，这时其伙伴块的起始地址为 $AR + 1 \ll (12+n)$ 。对于内存块 block1 的放宽的伙伴块的起始地址为 11000010000000000000000000000000。

4.5 本章小结

本章主要对于 Linux 操作系统的内存管理算法伙伴算法进行详细地分析。对于伙伴算法中对于伙伴块的要求过高而导致的内存使用率较低的缺陷提出了一种

改进方案，并且对于内存管理中使用的数据结构以及内存分配和内存释放函数提出相应的改进方案。

第五章 算法效率分析

5.1 算法有效性分析

在原伙伴算法中对于内存块的使用都是在 `free_list` 中查找, 这样的内存块都是严格满足伙伴定义的内存块, 因此对于内存的利用率比较低。改进后的伙伴算法重新利用了满足大小相等、内存地址连续的内存块进行合并, 这样会将更多的连续内存合并成更为大的内存块供系统分配使用。在 `new_free_list[n]` 中的每一个内存块都是由两个 `free_area[n-1]` 中的两个内存块合并而成的。在系统分配给进程 `new_free_list[n]` 中的内存块时, 实际上是分配的是两个 `free_list[n-1]` 中的内存块。这样可以使得更多的内存块得到利用, 更加有效、更加合理地利用内存资源。对于嵌入式下内存资源非常有限、且对于响应时间没有实时性要求的系统中可以使用这样的改进作为原伙伴算法的一个补充, 使得整个系统对于资源的利用率更高。

5.2 改进算法的测试和分析

在 Linux 内存管理中主要是内存的分配和释放两部分, 因此对于改进后的算法的测试也是针对于分配和释放进行的。通过将修改后伙伴算法中的内存分配和内存释放函数在 Linux 操作系统下进行编译, 通过应用程序模拟系统分配和释放内存。

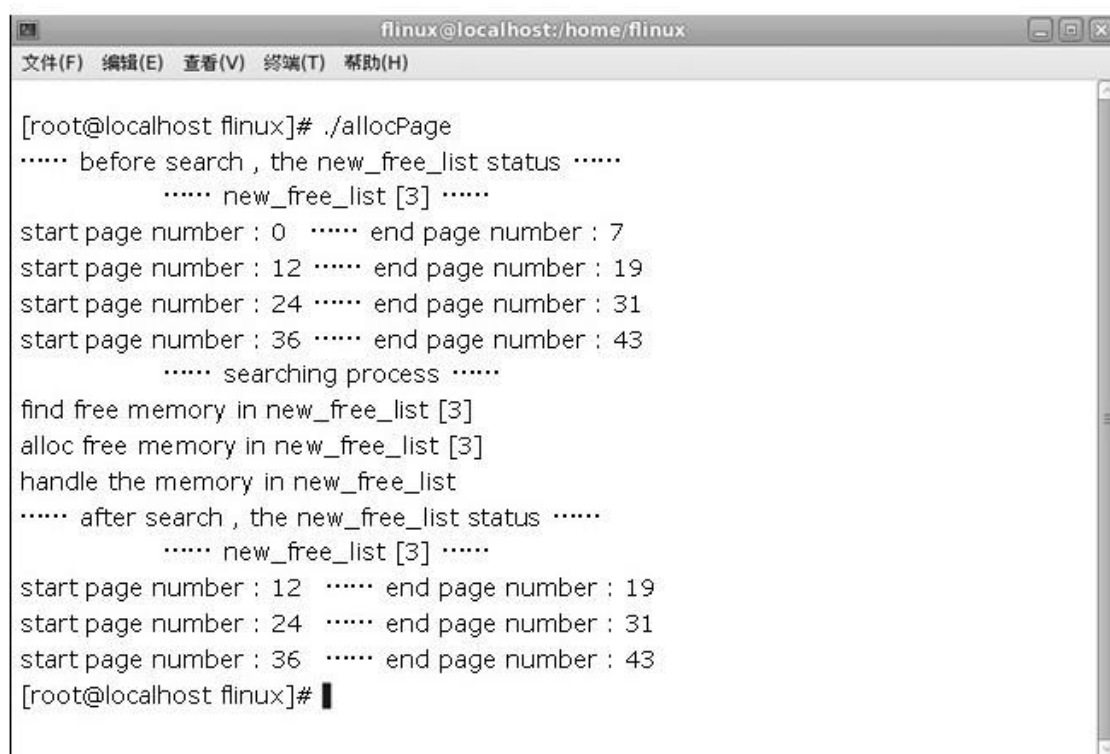
测试程序一:

将空闲链表 `new_free_list` 链接上符合放宽的伙伴关系的内存块, 通过改进后的内存分配函数 `_ _alloc_pages()` 在原搜索函数失败后来搜索放宽的伙伴块。在测试程序中进程申请 8 页物理内存, 其 `n` 值为 3。

如图 5-1 所示, 在空闲链表 `new_free_list` 链接着 4 个符合放宽关系的伙伴块, 它们都是由 `free_list` 中 `n` 值为 2 的内存块合并而成, 当 `_ _alloc_pages()` 搜索空闲内存时会首先查找 `free_list` 中的空闲内存块, 对 `free_list` 搜索失败之后就会继续搜索。

`free_area[3]` 中 `new_free_list` 上的空闲内存块。在进行搜索之前,

new_free_list[3]中链接着 4 个空闲内存块，起始地址分别为 0、12、24、36。当某个进程需要分配内存时，它会首先查看 free_list 中是否有空闲内存，当发现其中没有空闲内存时会从空闲内存链表 new_free_list 中查找空闲内存。它在空闲内存链表 new_free_list 中查找到空闲块，这时它会从中取下一个空闲内存块。



```
flinux@localhost:/home/flinux
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)

[root@localhost flinux]# ./allocPage
..... before search , the new_free_list status .....
..... new_free_list [3] .....
start page number : 0 ..... end page number : 7
start page number : 12 ..... end page number : 19
start page number : 24 ..... end page number : 31
start page number : 36 ..... end page number : 43
..... searching process .....
find free memory in new_free_list [3]
alloc free memory in new_free_list [3]
handle the memory in new_free_list
..... after search , the new_free_list status .....
..... new_free_list [3] .....
start page number : 12 ..... end page number : 19
start page number : 24 ..... end page number : 31
start page number : 36 ..... end page number : 43
[root@localhost flinux]#
```

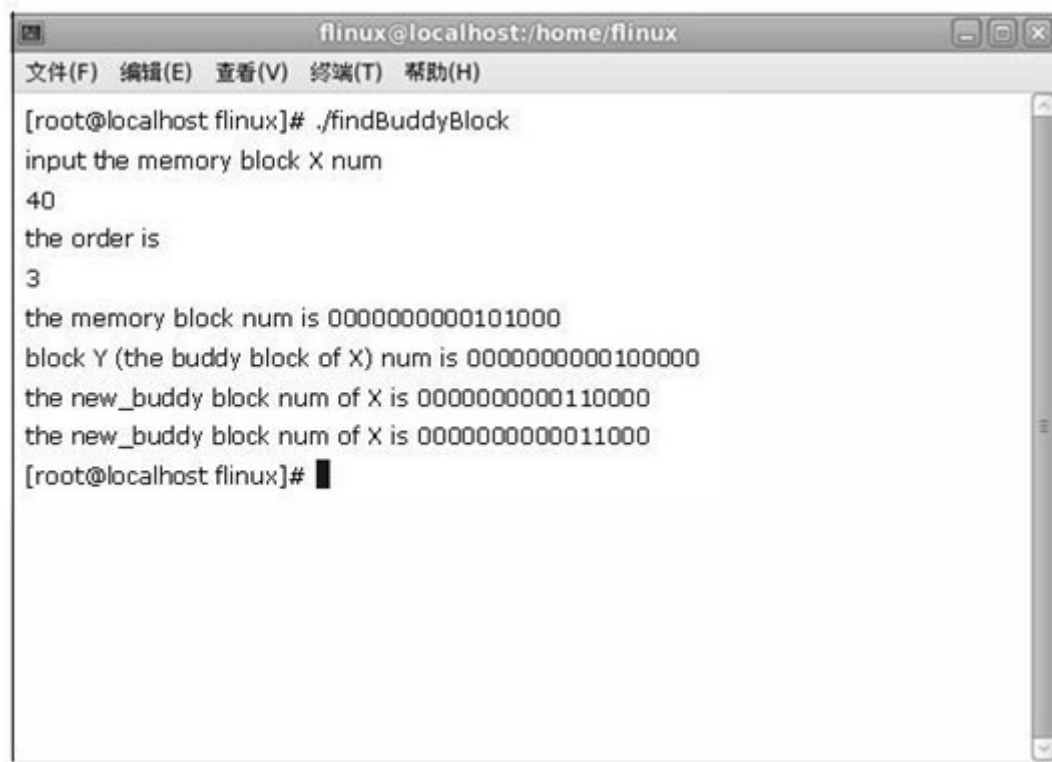
图 5-1 改进的内存分配算法

在原伙伴算法中，对于失败的内存分配的时间复杂度是进行 $(9-n)$ 次搜索，而在改进后的伙伴算法中需要多进行一次搜索，即对于 new_free_list 空闲链表的搜索，在搜索到空闲内存块之后原伙伴算法中只需要删除一个空闲内存块，也就是对于空闲链表进行一次操作。而对于修改后的伙伴算法，当从 new_free_list 中取下一个内存块时需要对空闲链表进行三次删除操作，不仅需要删除掉 new_free_list 中的内存块，还要删除掉组成这个内存块的两个 free_list 中的内存块。这样增加的时间就是一次空闲链表的搜索时间加上三次空闲链表的操作时间。

测试程序二：

对于伙伴内存块的查找操作。当一个内存块释放时被释放的内存块首先需要找到它对应的伙伴块的起始地址，然后根据伙伴块的起始地址来计算出其在位图中

的相应位置进而察看其伙伴块是否在空闲链表中。改进后的伙伴算法中对于伙伴块的查找是通过一个位操作来完成的。



```
flinux@localhost:/home/flinux
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)

[root@localhost flinux]# ./findBuddyBlock
input the memory block X num
40
the order is
3
the memory block num is 0000000000101000
block Y (the buddy block of X) num is 0000000000100000
the new_buddy block num of X is 0000000000110000
the new_buddy block num of X is 0000000000110000
[root@localhost flinux]#
```

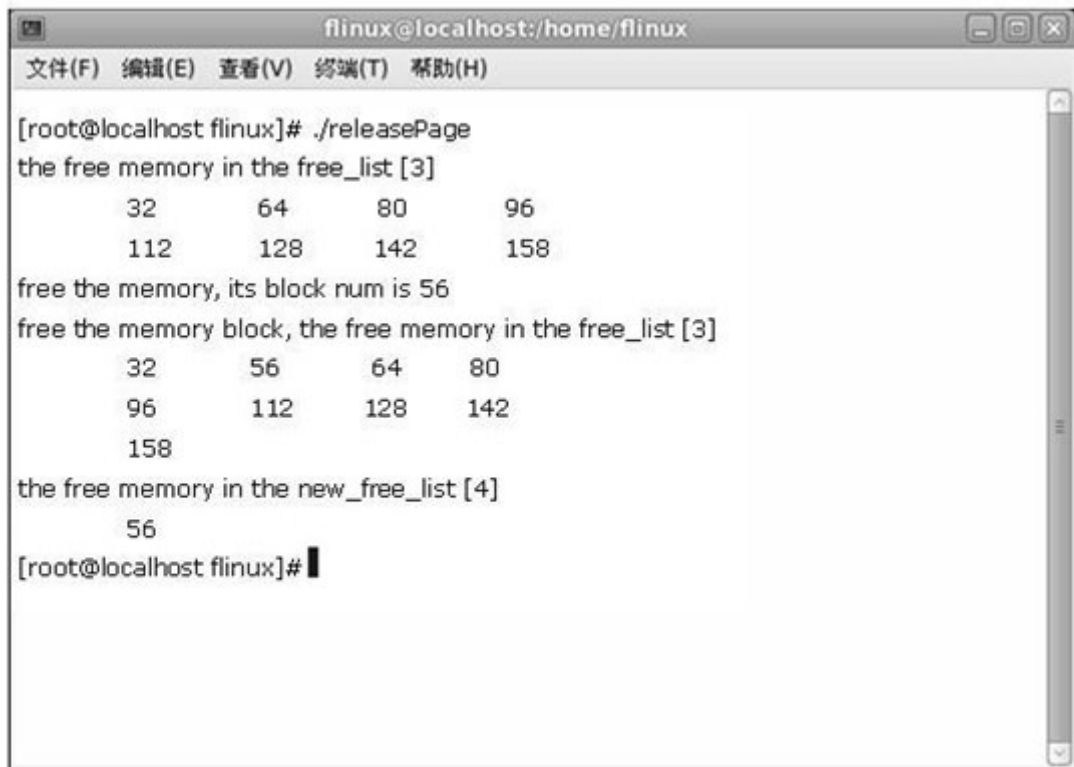
图 5-2 改进后的查找伙伴块算法

如图 5-2 所示，对于块号为 40 的空闲内存来说它的原伙伴块号位 32。以二进制表示这两个内存块号之间只有一位不同，它可通过一个异或操作完成查找伙伴块的操作。而对于放宽的伙伴关系来说，放宽的伙伴块与内存块的差别不只是某一位数字的差距。它需要更具内存块号最后一位的值来判断如何计算出放宽的伙伴关系内存块的块号，计算复杂度进一步加大。在这里内存块号为 40 的内存块的伙伴块的块号为 32，而它的放宽的伙伴块的块号为 48，而块号为 32 的放宽的伙伴块的块号为 24。

测试程序三：

内存块释放过程是向一个内存空闲链表链接空闲内存的过程，他会在将被释放的内存链接到链表之前首先查看是否存在伙伴块，如果存在则进行合并。不存在则直接连接到空闲内存链表中。

如图 5-3 所示，在释放块号为 56 的内存之前在空闲链表中有 8 个内存块，它们的起始地址分别为 32、64、80、96、112、128、142 和 158。当块号为 56 的内



```
flinux@localhost:/home/flinux
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)

[root@localhost flinux]# ./releasePage
the free memory in the free_list [3]
    32      64      80      96
   112     128     142     158
free the memory, its block num is 56
free the memory block, the free memory in the free_list [3]
    32      56      64      80
    96     112     128     142
   158
the free memory in the new_free_list [4]
    56
[root@localhost flinux]#
```

图 5-3 改进后的内存合并算法

存块被释放时，系统会首先计算它的原伙伴块，它的伙伴块的块号为 48。而该块内存不在空闲链表中，因此会将块号为 56 内存块链接到空闲链表 `free_list[3]` 中。其后系统会继续查看该块内存的放宽的伙伴块是否在空闲内存链表中，它的放宽的伙伴块的块号为 64，在空闲链表中，可以将这两块大小为 2^3 个页的内存块合并成一个大小为 24 个页的内存块链接到 `new_free_list` 中。

5.3 本章小结

对于内存的管理和改进主要是在内存分配和内存释放两部分进行改进，伙伴算法中对于内存管理是基于伙伴关系来进行的，在一定程度上影响到了内存的使用率，但是伙伴算法中对于内存分配和释放的操作比较简便，节省了对于内存管理的时间。本次改进的目的就是提高内存的利用率，因此增加了系统分配和回收内存块的复杂度，增加了操作所使用的时间。

在本章中的改进部分重新设计了一种寻找伙伴块的方法用来寻找和维护新建

立的空闲内存链表 `new_free_list` 和位图指针 `new_map` 所指向的新的位图。并且在分配内存时增加了 `new_free_list` 这个新增加的链表作为空闲内存的搜索目标。同时在释放内存时也增加了对于新建立的位图和链表的维护操作。在寻找伙伴块、释放内存和分配内存三个方面都增加了其操作时间、增加了时间复杂度，但是与之相对应的是提高了系统资源的利用率。在一些对于资源很少、响应时间的实时性要求不是很高的应用可以发挥很好的效果。

第六章 结 论

Linux 操作系统内存管理中使用伙伴算法进行内存的分配和回收。本文针对伙伴算法对大小相等、地址连续但不满足伙伴关系的内存块不能合并回收形成一个大的空闲内存块的问题，对伙伴算法进行了改进，主要工作如下：提出了按原伙伴关系和新的放宽限制伙伴关系维护两套空闲块链表，在原伙伴算法无法找到合适的空闲内存块时进一步寻找放宽限制伙伴关系形成的空闲内存块链表，为应用程序分配内存的内存管理机制。

该机制可有效地提高内存资源的利用率，对资源紧要的嵌入式系统应用提供底层内存管理支持。本文扩展了原有伙伴算法中内存管理的数据结构 `free_area_t`。在其中添加了新的空闲链表和新的位图指针。为新算法实现奠定基础；添加了用于查找放宽限制的伙伴块的函数，根据放宽限制的伙伴块出现位置的不同，设计了新的查找函数；实现原伙伴块和放宽限制的伙伴块的联合查找；修改了伙伴算法中内存分配和内存回收的函数，同时对原伙伴块和放宽限制的伙伴块进行管理和维护。

实验结果表明，改进后的伙伴算法实现了对放宽限制的伙伴块的分配和回收。能够在原有伙伴算法无法找到合适的内存块的时一候，通过放宽限制的伙伴关系来进行内存块的合并，进而将合并后的内存块分配给进程。改进后的伙伴算法能够更加有效地管理整个系统的内存资源，提高了内存资源的利用率。

在本文实现的 Linux 内存管理算法中，需要按原伙伴关系和放宽限制的伙伴关系维护两套空闲内存块链表，需要付出一定的管理开销。如何在提高内存资源利用率的前提下减少内存管理的开销是课题需要进一步深入研究的内容。

致 谢 辞

首先我要由衷地感谢我的导师李毅教授，在这两年多的时间中，李老师给了我巨大的关怀和无私的支持，在李老师的指导我才懂得如何学习和研究，李老师给予我的学术影响使我终生受益。李老师学识渊博，治学严谨，严于律己，宽以待人，在工作和学习中永远保持着旺盛的精力，是我一生学习的榜样。李老师给我参与项目的机会，但由于工作的原因没能继续深入下去，是我永久的遗憾。在此祝愿李老师身体健康，工作顺利。

其次要感谢王小松，李桂忠，施嵘和教研室的各位师兄，在我面对难题的时候给了我无私的帮助。还要感谢工作单位的领导和同事，在工作中给予我的照顾。

特别要感谢我深爱的父母，感谢你们对我的所有付出，你们是我前进的动力，使我有勇气面对工作和学习中的困难。在此致以我对父母最美好的祝愿。

对评审论文和参加答辩的各位专家，老师表示感谢，谢谢你们在论文评审和答辩过程中提出的宝贵意见。

最后向以上所有人士以及关注我成长的朋友致以最真诚的感谢！

参 考 文 献

- [1] D. (Daniele) Bovet and Marco Cesati. Understanding the Linux kernel (2nd Edition) [M]. O'Reilly, 2003.
- [2] Mel Gorman. Understanding The Linux Virtual Memory Manager [M]. Prentice Hall PTR, Inc., 2004.
- [3] Richard W. Stevens. Advanced programming in the Unix environment, 2005
- [4] Paul Larson. Kernel comparison: Improved memory management in the 2.6 kernel. IBM[Z]. <http://www-106.ibm.com/devel2 operworks/Linux/library/1-mem26/2004>.
- [5] Roberto Arcomano berto. Linux documents [DB / OL]. <http://www.tldp.org>, 2004
- [6] Andrew S. Tanenbaum. Modern Operating Systems (2nd Edition) [M]. Prentice-Hall, 2001.
- [7] Robert L. Linux Kernel Development [M]. 2nd ed. US: Novel Press, 2005.
- [8] 毛德操, 胡希明. Linux 内核源代码情景分析 [M]. 杭州: 浙江大学出版社, 2001.
- [9] 陈莉君, Linux 操作系统内核设计与实现 [M]. 机械工业出版社, 2006.
- [10] 林伟, Linux 内存管理子系统在龙芯 2 号上的优化 [M]. 硕士学位论文. 北京: 中国科学院计算技术研究所 2005.
- [11] 李善平, 等. Linux 内核 2.4 版源代码分析大全 [M]. 机械工业出版社. 2002.
- [12] William Stallings. 操作系统精髓与设计原理 [M]. 北京: 清华大学出版社, 2002.
- [13] 倪继利. Linux 内核分析及编程 [M]. 电子工业出版社, 2006
- [14] 谢长生, 刘志斌. Linux 2.6 内存管理研究. 计算机应用研究, 2005 (3): 58 — 60.
- [15] 吴晓勇, 曾家智, 操作系统内核中动态内存分配机制的研究, 成都信息工程学院学报, 1671-1742 (2005) 01-0027-04
- [16] 沈勇, 王志平, 庞丽萍, 对伙伴算法内存管理的讨论. 计算机与数字工程. 2004 (3)
- [17] 肖竟华, 陈岚, Linux 内存管理实现的分析与研究. 计算机技术与发展. 2007. 2
- [18] 高峰, Linux 内存管理设计与实现. 硕士学位论文. 沈阳工业大学. 2007
- [19] 栾建海, 李众立, 黄晓芳. Linux 2.6 内核分析 [J]; 兵工自动化软件技术, 2005. 2. 24
- [20] 田祖伟. Linux 内存管理机制分析和改进. 湖南人文科技学院学报, 2004 (6): 138 — 140
- [21] 俸远帧, 等. 计算机组成原理与汇编语言程序设计. 电子工业出版社. 1997
- [22] 陈莉君, 康华. Linux 操作系统原理与应用 [M]. 清华大学出版社, 2006

- [23]史芳丽,周亚莉.Linux 系统中虚拟文件系统内核机制研究[J];陕西师范大学学报,2005
- [24]夏煜.Linux 操作系统的文件系统研究[M].硕士学位论文.西安:西北工业大学,2000
- [25]陈燕晖,罗宇.Linux 2.6 存储管理子系统新特性分析[J];计算机工程与应用 2005。
- [26]Linux 的文件系统[EB/OL].<http://www.ironwareinfo.com.cn>,2004.
- [27]文件系统[EB/OL].<http://www.todayhero.net>,2004.
- [28]Evi Nemeth,Garth Snyder,Trent R Hein.Linux 系统管理技术手册[M].张辉译.北京:人民邮电出版社,2004.53~66,113~152.
- [29]李善平,陈丈智,等.边学边干--Linux 内核指导[M].浙江:浙江大学出版社,2002.
- [30]刘生平.桌面 Linux 内存管理性能优化技术研究 with 实现[M].硕士学位论文.北京:清华大学 2004 年.
- [31]李云华.独辟蹊径品内核:Linux 内核源代码导读[M].电子工业出版社,2009
- [32]Wolfgang Mauerer.深入 Linux 内核构架[M].郭旭译.人民邮电出版社,2010

电子科技大学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA



工程硕士学位论文

ENGINEERING MASTER DISSERTATION