分类号：_____ 密级：_____

U D C：_____ 编号：_____

工学硕士学位论文

多核系统内存管理算法的研究

硕 士 研 究 生 ：杨新波

指 导 教 师 ：李静梅 教授

学 科 、 专 业 ：计算机系统结构

论 文 主 审 人 ：门朝光 教授

哈尔滨工程大学

2011年3月

分类号：_____ 密级：_____

UDC：_____ 编号：_____

工学硕士学位论文

多核系统内存管理算法的研究

硕士研究生：杨新波

指导教师：李静梅　教授

学位级别：工学硕士

学科、专业：计算机系统结构

所在单位：计算机科学与技术学院

论文提交日期：2010年12月30日

论文答辩日期：2011年 3 月17日

学位授予单位：哈尔滨工程大学

Classified Index:

U.D.C:

A Dissertation for the Degree of M. Eng

Research on Multi-core System Memory Of Management Algorithm

Candidate: Yang Xinbo

Supervisor: Prof. Li Jingmei

Academic Degree Applied for: Master of Engineering

Specialty: Computer Architecture

Date of Submission: December, 2010

Date of Oral Examination: Mar., 2011

University: Harbin Engineering University

哈尔滨工程大学
学位论文原创性声明

本人郑重声明：本论文的所有工作，是在导师的指导下，由作者本人独立完成的。有关观点、方法、数据和文献的引用已在文中指出，并与参考文献相对应。除文中已注明引用的内容外，本论文不包含任何其他个人或集体已经公开发表的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

作者（签字）：杨新波

日期：2011 年3 月18日

哈尔滨工程大学
学位论文授权使用声明

本人完全了解学校保护知识产权的有关规定，即研究生在校攻读学位期间论文工作的知识产权属于哈尔滨工程大学。哈尔滨工程大学有权保留并向国家有关部门或机构送交论文的复印件。本人允许哈尔滨工程大学将论文的部分或全部内容编入有关数据库进行检索，可采用影印、缩印或扫描等复制手段保存和汇编本学位论文，可以公布论文的全部内容。同时本人保证毕业后结合学位论文研究课题再撰写的论文一律注明作者第一署名单位为哈尔滨工程大学。涉密学位论文待解密后适用本声明。

本论文（☑在授予学位后即可　☐在授予学位12个月后　☐解密后）由哈尔滨工程大学送交有关部门进行保存、汇编等。

作者（签字）：杨新波　　导师（签字）：李静梅

日期：2011 年3 月18日　　2011 年3月18日

摘　要

传统超标量处理器在典型程序指令流中不能发掘出更多的并行性，阻碍了单核处理器性能的提升。多核处理器成为继续提升系统性能的一种新技术途径，为构建高性能处理器提供了良好的硬件基础。

为充分发挥多核处理器的高性能优势，需要设计更好的操作系统策略与之匹配，这就给操作系统的设计者提出了新的技术挑战。内存管理作为操作系统的重要组成部分，对构建高性能多核操作系统起着至关重要的作用。在Linux操作系统中，其内核代码是开放的，使内核数据结构及算法的改进成为可能，为多核处理器内存管理的研究提供了一个良好实验平台。

Linux操作系统内存管理主要采用分页式管理方式，其典型算法为Buddy算法和SLAB算法。但是，在Buddy算法中由于某些连续内存块不能满足伙伴块的要求，导致内存中出现了许多较大的不能进行合并的连续内存块，降低了内存利用率。基于Buddy算法的不足，本文提出了一种部分延迟放宽合并算法，通过将连续的内存块进行合并，为内存分配提供了充足的空间，减少了系统中的内存碎片，提高了多核处理器内存利用率；同时，在原伙伴的基础上，通过将空闲链表分为伙伴忙空闲链表和伙伴闲空闲链表，延迟了伙伴块的合并时间，减少了内存的分裂与合并频率，提高了系统的运行时间。SLAB算法通过将频繁分配释放的对象缓存起来提高了内存系统的空间和时间性能，但是该算法对缓冲区内存的回收比较复杂。本文在原有SLAB算法的基础上引入了本地SLAB队列，通过本地SLAB队列和半满SLAB队列的使用，减少了自旋锁的使用，降低了核间共享内存的争用，提高了多核处理器内存访问的速度。

为验证算法的可行性、高效性，本文在Linux系统中采用SimOS模拟器搭建了实验平台，在实验平台上使用不同的内存管理算法进行了测试。测试结果表明：新算法减少了内存碎片的产生，提高了多核处理器系统的并行性。

关键词：多核系统；Linux内存管理；Buddy算法；SLAB算法；部分放宽延迟合并算法

Abstract

The traditional superscalar processor in the typical program instruction stream can not discover more parallelism, which hinders the performance improvement of single-core processors Multi-core processors considered as a new technical way to enhance the system performance continually provide a good foundation for building high-performance hardware processors.

The changes in maximize the performance advantages of multi-core processors needs to design better operating system strategies to match it, which presents new technical challenges for the operating system designers. As an important component of the operating system, memory management plays a vital role of building high-performance multi-core operating systems. The kernel code in the Linux operating system is open, which makes possible for the kernel data structure and algorithm and provides a good test platform for the research on the memory management.

The memory management in the Linux operating system mainly uses page management, whose typical methods are Buddy algorithm and Slab algorithm. However, since some memory blocks can not meet the requirements of the partners blocks in the Buddy algorithm, it results in the emergence of many contiguous memory blocks that can not be combined which leads to reduce the memory utilization. Based on the deficiency of the Buddy algorithm, this paper presents a part of the delayed relaxation of merging algorithm .The algorithm mergers contiguous memory blocks, which provides ample space for allocating the memory, reduces the memory fragmentation and finally increases the rate of multi-core processor memory utilization. At the same time, the free list is divided into the ones of partner-bussy and partner-free based on the original partners, which delays the time of mergering the partner blocks, reduces the frequency of splitting and mergering the memory and then improves system run-time. The Slab algorithm improves the performance of space and time by caching the memory allocated and released frequently. But the recovery of the buffer memory is more complex in the Slab algorithm. The paper introduces the local slab queue based on the Slab algorithm. Using the local slab queue and half-full queue reduces the use of spin locks, reducing the race for the shared memory in the kernel, and improves the speed of accessing

the memory of the multi-core processors.

An experimental platform is built with SimOS simulator in the Linux system to verify the feasibility and high efficiency of the method in the paper, Different memory management algorithms are tested in the experimental platform. The test result shows that the new algorithm reduces the memory fragmentation and increases the parallelism of multi-core processor systems.

Keywords: Multi-core System; Linux memory management; Buddy algorithm; Slab algorithm; Part of the delayed relaxation of merging algorithm

目　录

第 1 章　绪论

1.1 课题的背景及意义

近年来，随着集成电路制造技术的迅猛发展，单芯片上集成晶体管的数量呈几何级数增长，单核处理器的频率得到了大幅度提升，目前已高达 4GB[1]。然而，由于单芯片的面积有限，单芯片上集成晶体管的数量存在限制。根据摩尔定律，单芯片上集成晶体管的数量以每 18 个月翻一番的速度增加，但是，处理器芯片的面积毕竟是有限的，按照摩尔定律预测的速度发展下去，芯片集成度将很快达到极限，同时，不断提高处理器的主频，也会增加成本，并且造成功耗越来越大，散热问题越来越严重，如果这些问题得不到解决，单核处理器的性能提升将很快达到瓶颈。业界已有专家预计，芯片性能的增长速度将在今后几年内趋缓，一般认为，摩尔定律能再适用 10 年左右。为获得较高的处理器性能，目前许多专家都将目光投向了多核处理器。

多核处理器，又称单芯片多处理器(CMP，Chip Multiprocessor)，是指在一个芯片上同时集成两个或者两个以上微处理器内核，整个芯片作为一个整体向外界提供服务[2]。从理论上讲，多核处理器由于将两个或多个运算核集成在一个芯片内部，首先节省了大量的晶体管和封装成本，同时还能显著提高处理器的性能。另外，由于多核处理器对外的“界面”还是统一的，所以整个计算机系统需要为此做出的改变很有限，这意味着用户不会在主板、硬件体系方面做大的改变，从兼容性和系统升级成本方面来考虑有诸多优势。

在多核处理器系统中，多个处理器核心虽然可以并行访问内存，但是对于非统一内存访问的多核处理器系统来说，所有核心只能访问同样的内存单元。为了提高多核处理器系统的内存空间利用率，充分发挥多核处理器内核访问内存的并行性，需要应用自旋锁技术来实现访存动作的同步进行。但是，自旋锁的获取和释放会影响到系统的性能，因此如何减少访存时对锁的竞争，如何提高获取和释放锁的速度，是提高多核处理器系统内存利用率及降低防存时间的关键。

随着多核处理器的出现，在多核系统中如何有效利用内存空间，减少并行访问内存冲突、减少共享Cache失效率以及核间碎片等问题成为当前多核处理器的研究热点。为提高多核处理器的性能和内存利用率，许多研究者将多核处理器与内存管理系统进行结合，既保留了多核处理器结构简单的特性，又提高了多核系统的并行性。对多核系统内存管理而言，如何减少内存碎片，提高内存利用率和内存访问效率，减少多个核心并发

访问内存的冲突量，提高多核访问共享Cache的准确率，是衡量内存管理系统好坏的关键标准。但是，这些标准是相互冲突的，因此如何平衡多核内存管理中各因素间的关系，使多核内存管理系统的性能达到最好，已经成为目前研究的热点之一。内存管理算法是将系统中的多个进程均匀分配给各个核心，在多个内核上并行执行。在多核系统中，页面分配程序和回收程序都会被每个处理器核心不断地执行。当它们发生冲突时，一些处理器核心可能会处在忙等状态，而只有一个处理器核心在进行内存的分配或回收操作，极大地降低了处理器内存的利用率，降低了系统的并行性。因此，内存管理算法的好坏直接影响着多核处理器的性能，如果内存管理算法设计不当，将会降低多核处理器的并行性，降低内存利用率。综上所述，一个好的内存管理算法对提升多核处理器的性能发挥着至关重要的作用。

1.2 国内外发展现状

目前，在多核处理器系统的内存管理的研究领域中，为提高内存的利用率，同时使多核系统的并行性得到更好的发挥，国内外专家学者在该领域进行了大量的研究。

1.2.1 多核体系结构研究现状

1996年美国斯坦福大学第一次提出了片上多处理器的思想及结构原型。2001年IBM首次推出了第一款商用多核处理器即双核处理Power4[3]。UNIX阵营中的HP和Sun也相继在2004年2月和3月发布了PA-RISC8800和UltraSPARC IV双核处理器[4]。现在市面上流行的多核处理器包括Intel的Pentium D、Core 2 duo、Core 2 Quard系列，AMD的Phonem、Opteron系列，IBM的Power和Cell系列和SUN的UltraSparc等。

目前多核处理器主要有两种：同构多核处理器和异构多核处理器。同构多核处理器的设计原理简单，硬件更容易实现。这种处理器结构是当前市场上比较流行的双核和四核处理器的首选[5]。在同构多核处理器的设计中，随着芯片集成度的不断提高，单芯片上集成核心的数量越来越大，如何保持各个核心之间的数据一致；如何实现核心的存储访问和I/O访问；如何挑选一个好的处理器，能够让各方面性能达到均衡、且占用面积小，功耗低；如何使若干处理器的负载和任务达到最好的平衡效果等是目前急需解决的问题。

与同构多核处理器相比，异构多核处理器的优点是优化了处理器内部结构，通过将具有不同特点的核心相互组合，极大的提高了处理器的性能，降低了运行时的功耗。异构多核处理器存在的问题是：多个核心间如何搭配；如何完成任务在多个核心间的分工，

以更好的发挥多核系统的并行性；如何增加多核系统的扩展性，使结构不受核心数量的限制；如何更好地完成处理器指令系统的设计和实现。由于不同核心使用的指令系统制约着系统的实现，所以在不同内核的使用中，要求考虑运行时间，相同指令系统和不同指令系统的选择，操作系统的运行等问题[1,2,3]。

与单核处理器相比，多核处理器具有更好的结构优势：性能发展潜力大、集成度高、并行度强、结构简单，设计验证方便。多核处理器的出现为处理器性能提升和科学计算等领域开辟了一个新的研究方向[4]。然而，由于多核处理器是一种新型结构，设计人员难免会在设计和应用开发时遇到一些新问题，给多核处理器的片上通信、存储结构设计、操作系统设计、平衡设计、低功耗等方面提出了挑战。多核处理器的出现给计算机行业带来了巨大变革，主要包括体系结构、嵌入式系统设计和解决方案、编译技术和操作系统核心算法设计等领域。过去，对单核处理器的研究主要集中在提高处理器的频率方面，而多核处理器的研究则更注重内核间的协作、线程级并行和资源的共享。

在多核领域的研究中，目前，阻碍多核性能提升的因素有多种，其中低功耗和应用开发是束缚多核处理器发展的最主要因素。其中，低功耗设计涵盖了多方面的内容，包括电路级、结构级、算法级和操作系统级等，在设计过程中要求对问题进行全方面的综合考虑[4,5]。在多核应用开发中，由于目前在现实的应用开发中，程序员开发的程序大多数还是在C或C++的单线程环境中编写的，影响了多核系统应用开发的并行性，因此多核应用开发的并行行受到很大限制。多核处理器的到来要求软件开发者必须寻找新的软件开发方法，选择新的程序执行模型，这是程序员必须面对和解决的问题。

1.2.2 内存管理的研究现状

为适应多核技术和多核应用的不断发展，需要对多核环境下的内存管理进行研究，以寻求更好的提高内存利用率和多核并行性的方法。在多核系统中，内存的分配、利用和回收如何高效实现，已成为计算机领域比较热门的研究方向之一，许多大学和科研机构都进行了积极的研究，并取得了许多成果。

Marchand A，BalbaStre P，RipolI I，MaSmano M，和 Crespo A，从动态内存分配算法与回收算法的角度，对如何提高算法的可预测性问题进行了研究，提出了内存碎片的定义。Bach 在 1986 年提出了公平分享调度的策略，解决了多线程系统下负载不均的问题。随着多核的广泛应用，Emery D.berger，Hoar 提出了多线程应用程序内存分配算法，随后多位外国学者又相继提出了 Streamflow、PtMalloc 等内存分配算法。这些分配算法从不同的角度解决了多核内存分配中遇到的问题[6]。

在国内，许多专家也对内存管理算法进行了研究。对多核技术的研究已经被纳入国家863高技术研究发展计划基金项目。国防科技大学对多线程技术进行了一系列研究，提出了具有前瞻性的多线程系统结构。中科院计算所对低功耗SMA体系结构进行了研究，电子科技大学计算机系统结构研究所提出了一种基于多核网络平台的内存管理算法。

但是，到目前为止，多核处理器的内存管理算法仍不够成熟，仍需对多核处理器的内存管理算法进行不断深入的研究和探索。

当前，本文主要从三方面对多核系统的内存管理进行优化：第一，尽量减少多核处理器各个核心的共享数据量或减少共享数据的访问，让各个核心尽可能多的访问自己的私有数据；第二，尽量减少锁消耗的开销，在访问共享数据时，尽可能少的使用同步和互斥锁；第三，尽量提高每个核心对自己私有数据的访问性能，提高每个处理器核心的性能，从而改善多核系统的整体性能[7]。

无论是基于并行系统的多线程内存分配技术，还是基于多核的内存分配技术，目前都不够成熟。为提高内存的利用率和多核的并行性，要求内存管理算法不断减少系统中处理器核心间的通信开销和程序调度的执行开销。一个好的内存管理算法，不但可以提高 Cache 的命中率，减少多线程的并发访问冲突，而且还可以提高多核系统的并行性。如何实现多核内存分配的可管理性，在系统出现问题时能进行准确的定位，让内存使用效率尽可能地提高，是目前内存管理算法研究所面临的瓶颈。目前还没有一种算法适用于所有的多核系统，每一种分配算法都存在各自的不足，因此对内存管理算法的研究具有重要的意义。

1.3 论文的主要工作

本论文主要研究多核系统内存管理算法。通过对CMP体系结构和技术特点的深入研究与分析，总结了CMP的高性能优势和亟待解决的关键问题，介绍了多核系统的操作模型，明确了多核操作系统的发展过程；详细分析了现有Linux内存管理器中的内存管理算法。在上述研究的基础上，针对现有内存管理算法中存在的内存利用率低和缓冲区内存回收复杂等不足，本文提出了一种部分延迟放宽合并算法，通过将连续的内存块进行合并，为内存分配提供了充足的空间，减少了系统中的内存碎片，提高了多核处理器内存利用率；同时，在原伙伴的基础上，通过将空闲链表分为伙伴忙空闲链表和伙伴闲空闲链表，延迟了伙伴块的合并时间，减少了内存的分裂与合并频率，提高了系统的运行时间。并且在原有SLAB算法的基础上引入了本地SLAB队列，通过本地SLAB队列和半

满SLAB队列的使用，减少了自旋锁的使用，降低了核间共享内存的争用，提高了多核处理器内存访问的速度。

最后通过在Linux系统中采用SimOS模拟器搭建实验平台对算法的可行性和高效性测试结果表明：新算法减少了内存碎片的产生，提高了多核处理器系统的并行性，具有更好的应用前景。

1.4 论文的组织结构

本文内容共分 4 章，组织结构如下：

第 1 章，首先介绍本文的背景及意义，概括多核的国内外发展情况和内存管理算法研究现状，最后总结本论文的主要工作，给出本论文的组织结构。

第 2 章，对 CMP 技术进行总体分析，其中包括 CMP 的结构、优势和设计所涉及的关键技术等。

第 3 章，对 Linux 系统的内存管理算法进行研究，分析了内存管理算法中的相关技术，详细分析了伙伴算法的工作原理、物理页面的分配和释放流程、SLAB 算法的数据结构以及它们之间的关系，并提出了本文在多核内存管理算法上的改进思路。

第 4 章，详细设计了部分放宽延迟合并算法的实现，给出了该算法的数据结构、页面分配和释放流程。同时，对改进的 SLAB 算法进行详细设计，从理论上分析算法的可行性。最后对部分放宽延迟合并算法进行了性能测试。

第 2 章　多核相关技术研究

多核是随着大规模集成电路技术的发展而出现的一种新型处理器结构，为处理器性能的提升提供了良好的基础。同时，由于多个核心对内存的争用，使多核处理器的内存管理面临新的技术挑战。本章通过对多核处理器体系结构、多核技术以及多核操作系统的深入研究，为后续多核处理器内存管理算法的研究提供良好的理论基础[8]。

2.1 CMP 技术的研究

为更好的对 CMP 中涉及的相关技术进行研究，本节主要从 CMP 的体系结构和技术特点两个方面进行分析。

2.1.1 CMP 体系结构

CMP可被划分为同构多核和异构多核两种，其标准是按计算内核中集成的多个微处理器的核心是否对等一致进行的划分。同构CMP是一种多核处理器，这种处理器里都是类型相同的内核，而且各个内核的地位是对等的，通用处理器一般被作为同构多核处理器的核心。其中，一种典型的同构多核处理器，就是1996年美国斯坦福大学 (Stanford University)研制的Hydra多核处理器[7]。异构多核处理器是指处理器是由类型不同的内核组成，并且各个内核之间的地位是不对等的多核处理器，异构多核处理器多被采用“主处理核+协处理核”的设计模式，通用处理器通常被作为主处理器核，而对于一些特定的应用计算部件则采用协处理器核，其中，一种典型的异构多核处理器是Cell处理器，它是由索尼、IBM和东芝等合作研发推出的。下面分别对这两种体系结构进行介绍。

1．同构多核处理器的典型代表是 Hydra CMP 原型系统，Hydra 处理器在一个片的多核处理器上集成 4 个 MIPS 内核，在这 4 个内核中，每个内核拥有自己的指令和数据缓存，这个缓存被称为一级缓存，为了提高多核的利用率，在所有内核上设置了一个统一的、可共享的片上二级缓存，其大小为 1M，该系统采用一种基于监听的一致性协议来对这些核心之间的数据的一致性进行维护。它们既能完成在 MIPS 指令集中的 LL(Load Locked)和 SC(Store Conditional)操作，并实现同步原语，又能够完成常用的读取和保存操作[9]。

该系统除了共享片上的二级Cache，处理器内核还可以通过总线接口单元直接与片外高速SRAM构成三级Cache，并使得主存和输入、输出设备相连。Hydra支持线程间的猜测执行，对于整数应用，只能获得1~2倍的性能加速比，但对于大多数浮点应用和多

道程序，Hydra可获得传统单内核3倍以上的性能加速比。但是，同构多核处理器自身的结构特征，限定了它对大量非同质信息数据的处理速度，同时会带来功耗过大的问题。其实现的结构如图2.1所示。

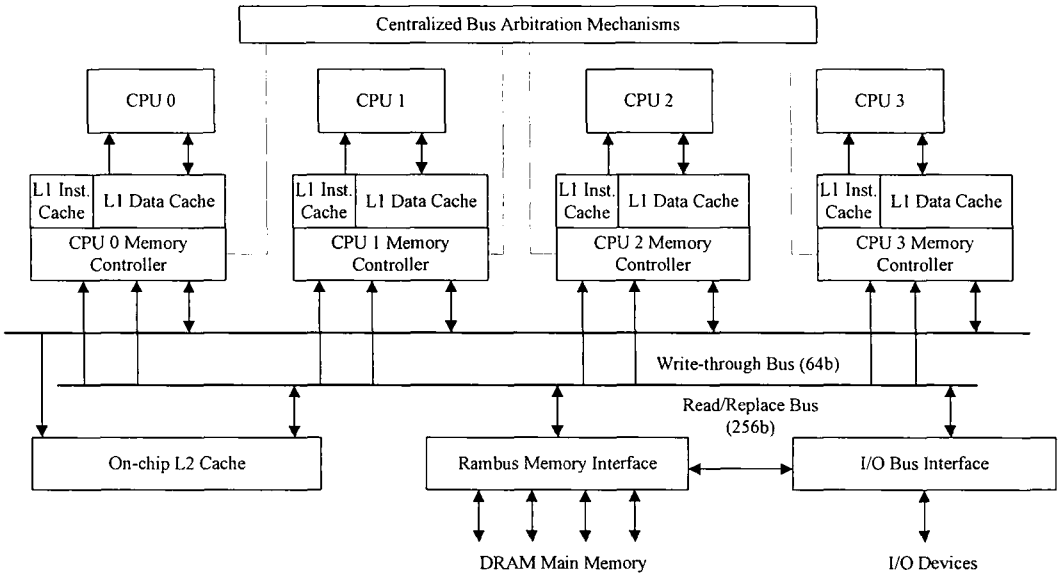


图 2.1　Hydra CMP 结构示意图

2．异构多核处理器的典型代表是由索尼、东芝与IBM公司联合研发的Cell处理器。它包括一个运行Power指令的主核(PPE)和8个SIMD辅助核，这些核间使用一条高速总线连接，该处理器的工作频率已超过4GHz，根据应用领域的需要，Cell处理器中核心的数目可以是一个变化的值。IBM作为Cell处理器的开发合作伙伴，将为Cell处理器的发展提供一个开放的平台，并为其提供源代码编码工作[10]。

它是一款性能超高的异构 CMP，运算性能峰值为 256Gflops，主频高达 4.6GHz。它含有一个通用的 PowerPC 处理器核 PPE(PowerPC Processing Element)和 8 个专用的 SPE(Synergistic Processing Element)处理器核。这 8 个 SPE 处理器核结构相同，都是 SIMD(单指令多数据流)结构的向量处理器，同时均具有各自的本地存储器。所以，每个 SPE 都可以独立地作为完整的处理器使用。每个 SPE 是一个 RISC 处理器，支持 128 位单精度和双精度浮点 SIMD 操作，局部存储采用 256KB 的嵌入式 SRAM。8 个 SPE 和 1 个 PPE 通过单元接口总线 EIB(Element Interface Bus)连接在一起。每一个 SPE 可以通过 EIB 和其它的 SPE 进行数据传输。PPE 采用 Power 结构，支持双线程，控制 8 个负责计算任务的 SPE。PPE 可以运行在传统的操作系统上，而 SPE 主要用它来运行向量浮点指令。PPE 包含 32KB 的一级指令 Cache、32KB 的一级数据 Cache 和 12KB 的二级 Cache。其实现的结构如图 2.2 所示。

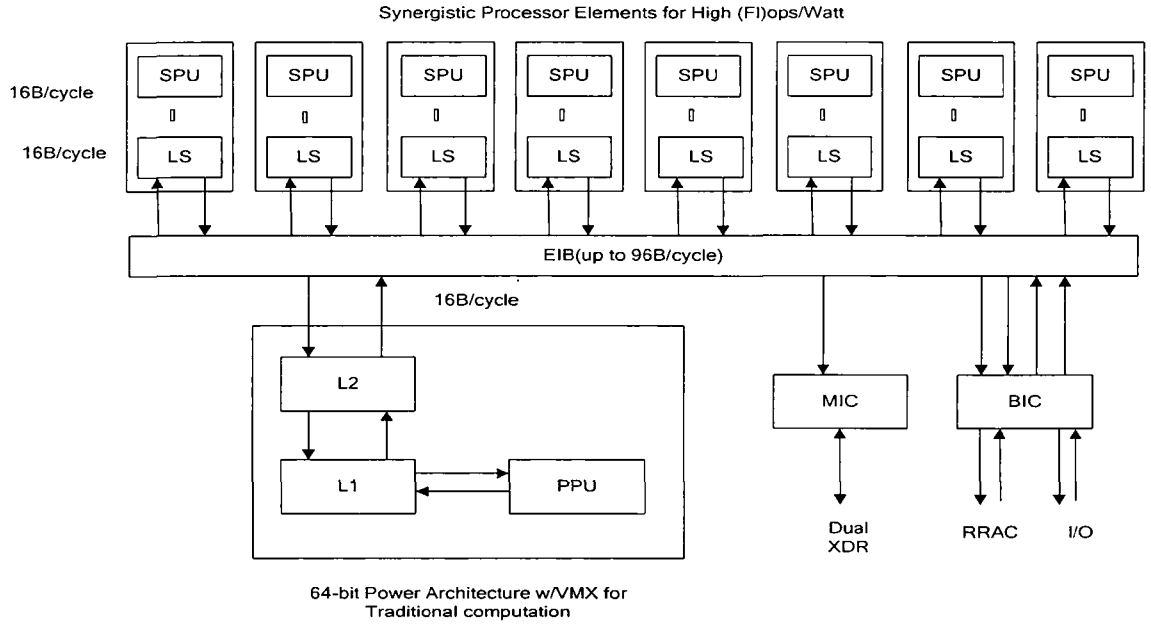


图 2.2　Cell 体系结构示意图

2.1.2 CMP 技术的特点

CMP结构的基本思想是将多个简单的超标量处理器核心集成在一个处理器芯片上，其中，利用了大量的晶体管资源，并且对超标量结构设计进行简化，消除了线延迟所带来的影响。在CMP中，多核处理器的多个处理器核心可以并行的工作，让多线程的并行性以及应用程序的指令级并行性得到更好的发挥，通过逐层的提高各个级别的并行性，系统的整体性能得到了很好的提高[11]。

CMP结构的优点是把多个处理器核心集成在一块处理器芯片上，从而构成一个整体，并且简化了单核处理器的设计结构，这样不但消除了线延迟所带来的影响，而且一定程度上可以让处理器主频达到极限值，降低处理器设计和验证的时间周期。而且，CMP结构还具有很好的可扩展性，可以适应工艺尺寸比例的缩放[12]。

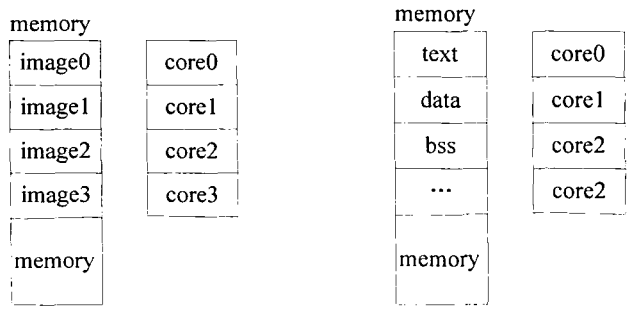
CMP存在的主要问题在于一个处理器芯片上集成了多个单处理器核心，而这些处理器核心要共享同一个资源，CMP是通过划分方式完成资源共享的，由于访问线程的不足，可能会导致资源的利用不充分。

从目前微处理器发展的形势看，CMP结构必将成为未来微处理器结构发展的主要方向，这是因为CMP处理器更新换代简单且工作量小，只需在一个芯片上采用一定的技术就可以集成多个处理器核心，为了让处理器内核的通信带宽与延迟得到满足，需要对其内核间的连线逻辑进行适当的变动[13]。因此，CMP处理器的更新换代与一个具有高速流水线逻辑的处理器的重新设计相比较，无论从时效还是从工效，都显示出一定的优势。

此外，CMP处理器的主板采用了新的I/O技术标准来提高内存和I/O的带宽。在发生几代变革的硅工艺之后，基于CMP的处理器给工程设计减少了很大成本，这是因为升级换代处理器后，新的设计只是多集成了几个处理器内核，这比传统的设计要容易得多。一些共性的工程工作可以被不同的处理器设计厂家分摊研究，这也就导致了该处理器能够满足不同硬件的需求。

2.2 CMP操作系统

多核系统的工作模型可以分为 Multiple Image 模型和 Single Image 模型[14]。Multiple Image 模型属于一种松耦合的方式，系统中的每个 Core 运行独立的操作系统，具有独立的系统映像，多个 Core 间通过操作系统提供的共享内存和管道等机制实现多个 Core 间的通信，采用多个 Core 这种方式下可以运行不同的操作系统，每个 Core 运行的代码需要独立编译和加载。Single Image 模型则属于紧耦合方式，所有的 Core 都由一套操作系统进行管理，并且可以共享同一份系统映像(包括代码段，数据段等)，系统中所有的 Core 共享相同的内存空间，它们之间可以互相直接访问彼此的数据。其结构图如图 2.3 所示。



(a) Multiple Image 模型图　　(b)Single Image 模型图

图 2.3　多核系统工作模型

2.2.1 多核操作系统的发展

在多核系统中，多个执行线程能够真正地并行执行，它与以往的单处理器单线程和单处理器多线程环境有着本质的区别，它对应用程序，编译器和操作系统等各层次软件的开发都提出了新的要求和挑战[15]。目前，很多操作系统都是在单核处理器系统上开发的，还没有为多核处理器设计一种操作系统，很多操作系统都是通过支持SMP(Symmetrical Multi Processing)来支持多核系统的，但是由于多核处理器(CMP)和对称多处理器(SMP)并不是完全一样的，支持SMP的操作系统和多核系统之间的平行移植不能使多核处理器的效率得到充分发挥，在单机多道程序系统上发展起多处理器操作系

统，具有以下特征[16]：

1．并行性：由于多处理器操作系统支持多个处理器并行的运行，这就可以让多个进程并行的在各个处理器上执行，从而提高了系统的吞吐率和减少了运行时间。

2．分布性：指多处理器操作系统中的进程不再只是在某个处理器上运行，它可以分散到多个处理器中运行。

3．通信和同步：多处理器系统与单核处理器系统相比，由于进程间的资源共享和相互合作，增加了处理器间进程的通信和同步。

4．可重构性：在多处理器系统中，当某个处理器发生故障时，操作系统把它换出来，并使该处理器重新启动。

2.3 SMP 系统

对称多处理器(SMP)是一种紧耦合形式的多处理器系统。它能够实现真正意义上的并行执行，因为系统中存在多个处理器，每个处理器在同一时刻都可以执行任务，并且能够让多条指令同时执行[17]。对称多处理器系统的体系结构如图2.4所示。

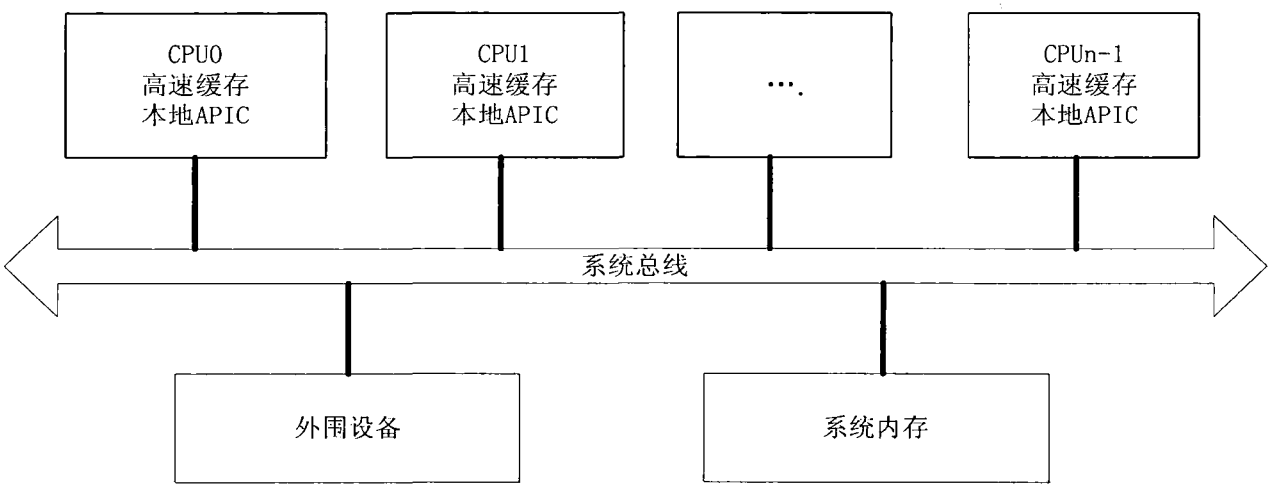


图 2.4　SMP 体系结构图

SMP 系统是指在一个计算机系统中同时共享一个内存并汇集了多个处理器的系统，多个处理器共享系统总线带宽、系统内存以外围设备。在这样的系统中，PC 机是由多个处理器而不是单个处理器构成。虽然如此，外表上仍然是一台单机系统，只是此时多个处理器可以同时执行同一个操作系统的副本，多个任务在每个处理器上可以并行执行，其他资源被系统共享，从而整体上使系统的性能得到提高。

2.4 本章小结

本章对 CMP 和 SMP 技术进行了分析和比较。首先对 CMP 的技术进行了研究，包括 CMP 设计的关键问题、核结构、CMP 系统的工作模型、CMP 与 SMP 的异同、SMP 的系统结构等问题。接下来对现有的操作系统进行了阐述，包括 SMP 操作系统和 CMP 操作系统。

第 3 章　Linux 内存管理算法的研究

Linux 虚拟内存管理器采用段页式管理模型，其目的是让较大的程序能够在有限的内存空间上顺利执行，从而提高内存的容量，因而，便出现了虚拟内存和实际物理内存两个概念[18]；这样把 Linux 内存管理器进行了分类：即虚拟内存管理和实际物理内存管理，实际物理内存的管理进一步可分为物理内存管理、内核管理和用户空间管理[19]。在本章中，物理内存管理算法和内核管理算法是主要研究对象。

3.1 Linux内存管理器

有多种类型的存储器存在于计算机系统中，包括 Cache、内存、外存。内存和外存是系统的重要资源，需要为其提供专门的管理系统。

在 Linux 系统中，内存管理作为一项复杂的工作，把它分为几个不同的内存管理器来管理内存[20,21]。在对内存进行管理时，这些内存管理器，相互促进，互相补救，共同把内存的管理工作处理好。Linux 中通常分为 5 个内存管理器[22]。组成如下：

1. 物理内存管理器。物理内存管理器以页为单位，对物理内存进行分配、回收和释放，从而提高了内存的使用效率，减少了内存碎片的产生[23]。伙伴算法是 Linux 操作系统经典的物理内存管理器。

2. 内核内存管理器。内核建立各种管理结构是通过小块内存来完成的，而物理内存管理器不能满足其要求，因此 Linux 采用了一种专门负责分配和释放内核中小块内存的管理器——内核内存管理器，其专门负责分配和释放内核中小块内存。SLAB 算法是典型的 Linux 内核内存管理器。

3. 虚拟内存管理器。虚拟内存管理器的出现使得用户进程的运行不再受实际内存空间大小的限制，它使用了页目录和页表核交换机制，并在物理内存管理器的基础上，为系统中每一个进程模拟了一个虚拟地址空间，并使其与实际请求的内存大小相等[24]。

4. 内核虚拟内存管理器。内核虚拟内存管理器利用虚拟内存管理器的原理并通过内核虚拟地址来实现的，从而满足了 Linux 内核对大内存的要求。

5. 用户空间内存管理器。该管理器不属于内核，其目是在进程堆中完成用户态虚拟内存的动态分配和回收。一般用于在库中实现，受内核的支持[25]。

关系如图 3.1 所示。

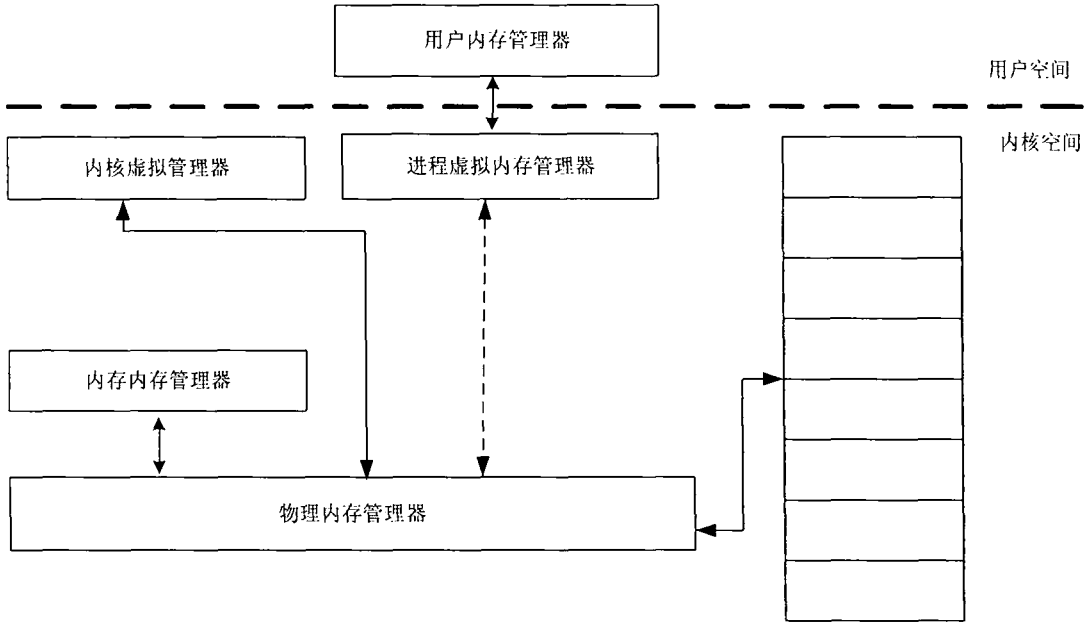


图 3.1 Linux 内存管理体系关系图

3.2 Linux 内存管理器的分析

操作系统的重要组成部分之一是内存管理子系统，它的主要任务是对操作系统的内存进行高效的管理，有效地完成内存的分配，回收和使用操作，提高系统对内存的利用率，从而减少内存碎片的产生，并让内存资源在系统运行时发挥最大的作用[26]。

3.2.1 物理内存管理器

Linux 采用分页机制来管理内核中的物理内存，整个内存被它划分成无数 4k(在 i386 体系结构中)大小页，从而使内存页成为分配和回收内存的基本单位。物理内存管理器是整个内存管理的基础，它负责管理系统中物理内存资源，并负责对系统中的内存进行分配和回收。显然，物理内存采用什么样的分配和回收机制对整个系统的操作性能有很大的影响。利用分页的管理机制，可以在分配时减少内存碎片的产生，这是因为分配时并不要求大块连续的内存存在[27]。尽管如此，但在实际的系统应用上，人们还是倾向于在分配内存时使用连续的内存块，因为连续内存块的分配，页表不需要进行更改操作，从而可以使 TLB 的刷新率降低。目前有许多关于管理物理内存的算法，如最优匹配算法和位图算法等，这些都是由相关的系统设计人员设计完成的。研究者通常把效率和利用率作为衡量内存管理算法优劣的主要标准[26]，而两者在实际设计中有时是冲突的，算法的内存利用率高而其效率却不一定高；或是某算法对内存的访问效率很高，但其内存

利用率却很低。Linux 采用了折中的算法：物理内存管理器以页为单位运用伙伴算法(Buddy Algorithm)进行实现，高效率地分配和回收物理内存，并充分的利用内存空间，减少内存碎片的产生。为了让物理内存得到更充分的利用，物理内存管理器中的内核内存管理器为内核用户提供管理物理内存策略来管理大小不同的内存块。虚拟内存管理器占用很大的虚拟空间[28]。

3.2.2 内核内存管理器

在实际运行中，内存经常需要频繁地被 Linux 操作系统内核访问，而所有内核最普遍的访问内存的操作是分配、回收和释放数据操作。系统内核经常访问的内存的数据结构与一般用户使用的内存相比较，有一些自己的特性；如，专供内核使用而不参与交换的内存，它在进行分配和响应时，需要访问的时间很短，其大小也比一个普通页面小很多，占用时间也较小，因此，在物理内存管理器上直接分配空闲的内存块时，容易造成内存空间的浪费。为了提高内存的使用率，Linux 操作系统在物理内存管理器上增加了内核内存管理器[29]。

内核内存管理器的实现策略随着操作系统的发展不断的进行演变，逐步地被优化改进。在系统初始化时，系统为每种数据结构提前预留一定大小的内存，那种不具有专门的内核内存的管理器是目前最简单的内核内存管理器。这种策略虽然简单，但是运用时无法满足系统对动态内存分配的需求。内存预留的多少是一个问题，如果预留的过大，则会造成内存资源的浪费，如果预留的过小，则可能会造成系统崩溃。因此，操作系统一般根据自己的需要设有专门的内核内存管理器。目前现有内核内存管理器常见的分配策略有以下几种：简单 2 次幂空闲表、资源映射图分配器、伙伴系统、McKusick. Karels 分配器、SVR4 Lazy 伙伴算法、Solaris2.4 的 Slab 分配器和 Mach-OSF/1 的 Zone 分配器。通过分析每种管理器，发现它们都有其自身的优缺点，但目前最好的一种管理器是 Solaris2.4 的 SLAB 分配器。Linux 利用 SLAB 分配器的思想设计了内核内存管理器，并在其缓存层数据结构中分享了它的名字和基本设计思想。内核通用数据结构发挥着缓存层的作用。SLAB 分配器作为 Linux 内核内存管理器分配模式，有如下优点[30]：

1．数据结构、构造函数以及析构函数组成 Linux 内核使用的数据对象，其大小一般小于普通的物理页面。内核内存管理器在分配数据对象时初始化对象的内存区用构造函数，回收数据对象时，清除对象的内存区需要用析构函数。一般情况下，为了减少对数据对象的重复初始化，SLAB 分配器把这些释放的数据对象保存在内存中，防止内核以后请求同样类型的数据对象并重新对其初始化，这样可以从内存中直接获得已经初始

化好的数据对象，减少了内存访问时间。从另一个意义上讲，SLAB 分配器的存在，减少了 Linux 内核中物理内存管理器对伙伴系统的调用，从而提高系统的整体性能。

2. 同一类型的数据对象在 Linux 的内核中有时需要反复申请，而 Slab 分配器能更好的满足此种请求分配，从而减少对伙伴算法的直接调用。

3. Linux 系统请求内存的大小不同，根据频率把这些大小不同的内存请求进行分类。第一类请求频繁的内存块，专门划分出相应大小的内存对象对其进行内存管理，一定程度上可以减少内存碎片的产生；另一类是请求不频繁的内存块，则将其按照 2 的幂次方的内存块大小来进行划分，从而实现对其有效地管理。这种方法尽管会产生一些内存碎片，但一定程度上满足了内核的需求。

4. 为了使系统达到较好的性能， Linux 操作系统通过硬件高速缓存来处理起始地址。内存对象如果不是 2 的幂次方(满足严格的对齐的内存数据结构)，使用硬件高速缓存可以减少 Linux 系统中物理内存管理器调用伙伴算法的次数，从而提高了内存的分配和回收的效率，增加了平均访问内存的次数。

3.3 伙伴算法

伙伴算法是 Linux 内存管理管理中经典的算法之一，本节从该算法的结构、页面的分配和释放三个方面进行详细的分析。

3.3.1 算法的设计原理

在 Linux 内存管理系统中，伙伴算法设计原理是把所有的空闲页面分为 10 个大小相等的块组，其中每组中块的大小为 2^n 页面。例如，第 0 组中包含的都是大小为 2^0 个页面块，第 1 组中包含的都是大小为 2^1 个页面块，第 9 组中包含的都是大小为 2^9 个页面块。即每一组都是由同样大小的块连接成一个链表组成，这样的块的链表共 10 个，分别包含大小为 1、2、4、8、16、32、64、128、256 和 512 个连续的页面[8]。在系统中，RAM 块是可供分配的最大的空闲内存链表块。它是对 512 个页面的最大请求所对应着 2MB 大小的连续 RAM 块。系统在进行初始启动时，对所有的物理页面来讲，都是首先把大的空闲内存块链接在较大的空闲链表中，然后再挂在比较小的链表中。

对于伙伴算法的伙伴块来说，伙伴是由满足以下定义的两个块组成：

（1）两个块的大小相同

（2）两个块的物理地址连续

（3）两个块是从一个大块中分离出来的

伙伴算法在进行内存分配和回收时，把满足上面三个条件的两个块合并为一个大块，如果合并后的块还满足条件，那么继续合并，直到最大值[31]。

3.3.2 Buddy 算法的数据结构

Linux 系统提供了对非一致内存访问(Non-Uniformly Memory Access，NUMA)模型的支持。在该模型中，同一个 CPU 对不同内存单元的访问时间有可能不同。在系统的物理内存所划分的每一个单独的节点(node)内，对于相同的 CPU 来说，同一个 CPU 所访问的页面需要消耗的时间是相同的。但是，对于不同的 CPU，通常情况下消耗的时间是不同的。一般 CPU 通过减少对耗时节点的访问次数，来提高自身的速度。这就要求内核要选择 CPU 经常使用的内核数据结构所对应的位置。由于内存节点和 pg_data_t 的描述符是一一对应的关系，系统中内存节点和 pg_data_t 的描述符在数量上是一样的。节点描述符组成了内存节点单链表，它们用指针串联在一起，并由 pgdat_list 变量指向作为其第一个元素。同时，每个节点的物理内存被划分成四个管理区(Zone)，分别为 ZONE DMA32(未使用)、ZONE DMA 管理区、ZONE HIGHM 管理区和 ZONE NORMAL 管理区，每个管理区拥有自己的一个伙伴系统，通过伙伴系统来管理自己的物理内存[32]。每个物理内存又被划分为若干个物理页面，并且这些物理页面的大小相等，使用 page(或 mem_map_t)描述符来对一个物理页面进行描述，所有的物理页面在管理区内组成一个 page 描述符的数组 mem_map。

page 结构定义为 mem_map_t 类型，定义在/include/linux/mm.h 中：

```
typedef struct page {
        struct page *next;
        struct page *prev;
        struct inode *inode;
        unsigned long offset;
        struct page *next_hash;
        atomic_t count;
        unsigned flags;
        unsigned dirty,age;
        struct wait_queue    *wait;
        struct buffer_head * buffers;
        unsigned long swap_unlock_entry;
```

```
    unsigned long map_nr;
  } mem_map_t;
```

Count 共享该页面的进程计数。

Age 标志页面的“年龄”。

Dirty 表示是该页面的修改标志。

prev 和 next：把 page 结构体链接成一个双向循环链表的指针。

prev_hash 和 next_hash 把有关 page 结构体连成哈希表。

对不同管理区，这种算法使用单独伙伴算法管理。在每一种类中，不同的页框都由一个 free_area_struct 结构体进行管理。系统将 10 个 free_area_struct 结构体组成一个 free_area[]数组。free_area_struct 结构体定义如下：

```
#define MAX_ORDER 10
Type struct free_area_struct
{
  struct list_head, free_list;
  unsigned int *map;
}free_area_t
```

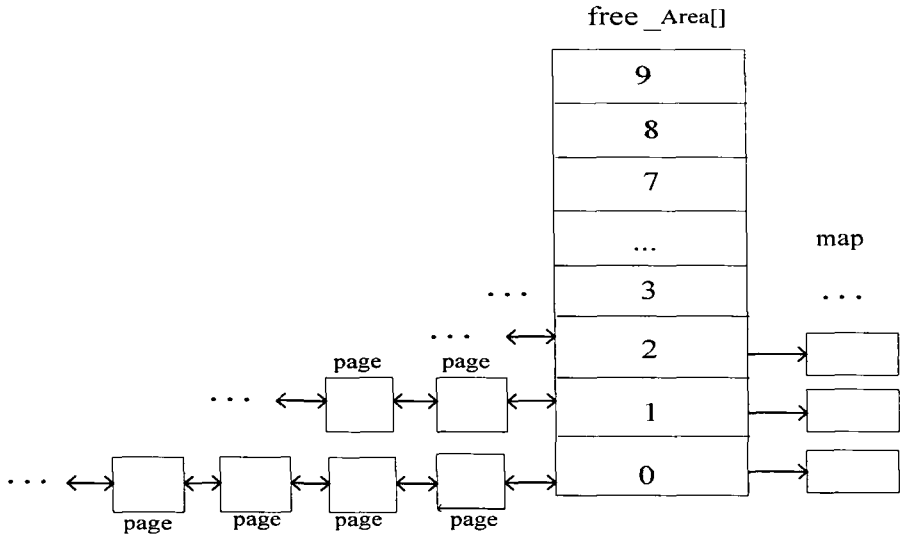


图 3.2　Buddy 算法使用结构

其中 list_head 域表示一个通用双向链表结构，链表中元素类型是 Mem_map_t(即 struct page 结构)。它的大小由现有的页面数的 Map 域所指向的一个位图决定。free_area 指第 k 项位图的每一位，这一位是用来描述两个伙伴块的使用情况的。两个页面块是空闲的，或都被全部占用时，则位图对应的位是 0，如果当一对 Buddy 的两个页面块中有

一个是空闲的，则位图对应的位是 1。

如果伙伴块都为空闲，内核则将其作为一个的单独块来处理，其大小为 2^{k+1}，(k=1、2、3、4、5、6、7、8、9)如图 3.2 所示。

3.3.3 Buddy 算法物理页面分配

在进行内存分配时，Linux 系统调用伙伴算法中的内存分配函数来完成内存的分配。依据申请的空闲内存块空间的大小，内存分配函数到 free_area 数组中查找与其相对应的空闲块链表，查找是否有满足申请空闲内存块[33]。内存分配函数对空闲物理页面块的大小的分配是以 2^n 为单位的，如果页块满足请求则进行分配，然后返回当前块的首个物理页面的起始地址；如果没有满足请求的页块，则分配失败，返回出错信号。

Linux 伙伴算法在内存分配中使用 alloc_ pages()函数，其具体流程如下。

1．函数定义：

static inline struct page*alloc_pages(int gfp_mask,unsigned long order);

2．参数解析：

① gfp_mask：标志字。

② order：用来表示请求页的大小，一般为 2 的 order 幂，这里是在 free_area 数组中的页块索引。

3．page*alloc_pages 运行过程如下：

① 检查是否有内存块满足需求。其方法是：首先对第二个参数order与MAX_ORDER的值的大小进行判断，其中第二个参数order是由配函数传递的，MAX_ORDER的值的大小是系统定义变量(MAX_ORDER是最大空闲内存块的幂次)，如果前者的值大于或后者定义的值，则系统中不存在空闲内存满足所请求的内存块，函数返回空，则内存分配不成功；如果前者的值小于后者所定义的值，则说明系统中有合适的空闲块满足内存请求，此时在free_area[order] 中直接查询空闲内存块链表，把满足请求的内存块进行分配。

②统计系统中空闲物理页数，判断当前空闲页的大小是否满足页面的请求，如果满足，则系统根据请求页面的大小直接分配内存，如果没有满足请求页面的大小的内存块，则需先对空闲内存进行回收操作，然后再进行内存分配操作。空闲物理页面在进行内存的回收时，首先需要调取函数try_to_free_pages(gfp_mask)，接着由内核交换守护进程kswapd负责回收物理页，该进程负责内存的回收工作，并且kswapd必须是处于唤醒状态的。在一般情况下，该进程是处于休眠状态的，需要调用wake_up_process函数将其唤醒，

唤醒后即为守护进程，然后负责内存的回收操作。如果在gfp_mask中设置了_GFP_WAIT标志(该标志一般情况下都设置)，则直接调用do_try_free_pages函数，强行对内存进行回收。如果在gfp_mask中没有设置_GFP_WAIT标志，则唤醒kswapd，然后立刻转到③，直接分配内存。

③进行正常的分配内存。首先在free_area数组中查找第order项，它是struct page结构的链表。在该链表中如果存在满足请求的页块，则把该节点从链表中摘下；同时对此页块在free_area数组中与之对应的位图位作取反操作，该位图用来表示内存页块的使用情况；同时修改全局变量nr_free_pages；接着算出该页块的起始物理地址，这可以根据其在mem_map数组中的该页块位置，可以计算出页块的起始物理地址；并对该页结构的引用计数加1；然后返回该物理页块所对应的首地址。在当前链表中若不存在符合所请求内存页块时，则需要在free_area数组中进行逐增查找满足请求的内存块，最终可以得出两种结果：a)在整个free_area数组中不存在满足请求的页块，此时分配失败。在gfp_mask中如果没有设置_GFP_WAIT标志，就由别的进程在CPU中进行，自己等待一个时间段。如果进程又一次执行，则返回0。b)在free_area数组的其它链表中查找到了满足请求的内存页块，把该链表节点从相应的链表中取下，并把该页块在free_area数组中相应的位图位取反。位图作为区分该页块是否被应用或被分配还是空闲的标志，需要同步对全局变量nr_free_pages进行更新。因为此时所查到的满足请求的空闲的内存页块通常比实际需求的页块要大，所以要对此内存页块进行分割操作。由于在Linux中，页块分割是以2为指数次幂为单位的，将刚查到的满足请求的页块分为大小相等的两个伙伴。地址较小的伙伴块的地址存放在free_area数组对应的链表中，并将其链表位图中对应的位进行修改。如果分割后的大伙伴块仍比实际请求页块大，采用前面所运用的分割操作对其继续进行分割，直到所需要的页块的大小与请求的内存大小相等。如果大伙伴分割操作完成，然后获得该页分配的物理页的首地址，把与之相应的页结构的引用计数加1，并把获得的地址返回调用内存分配函数的进程。如图3.3所示。.

3.3.4 Buddy 算法的页面释放

通过对 Buddy 算法中物理页面的分配函数进行研究，本节接下来会介绍 Buddy 算法中的内存释放函数，由于内存页面分配是将大页块划分为小页块的过程，随着内存块分配频度的增加会造成内存块变小，为此，在页面释放过程中，空闲页块应尽可能的进行合并，把小页块合并成大页块，以便提高内存的利用率。

在Linux系统的函数free_pages中，其内存页块的释放界定如下：

void free_pages(unsigned long addr, unsigned long order)，其中addr表示将被释放页块的首地址，order表示释放页块的大小为2的order次幂个物理页。

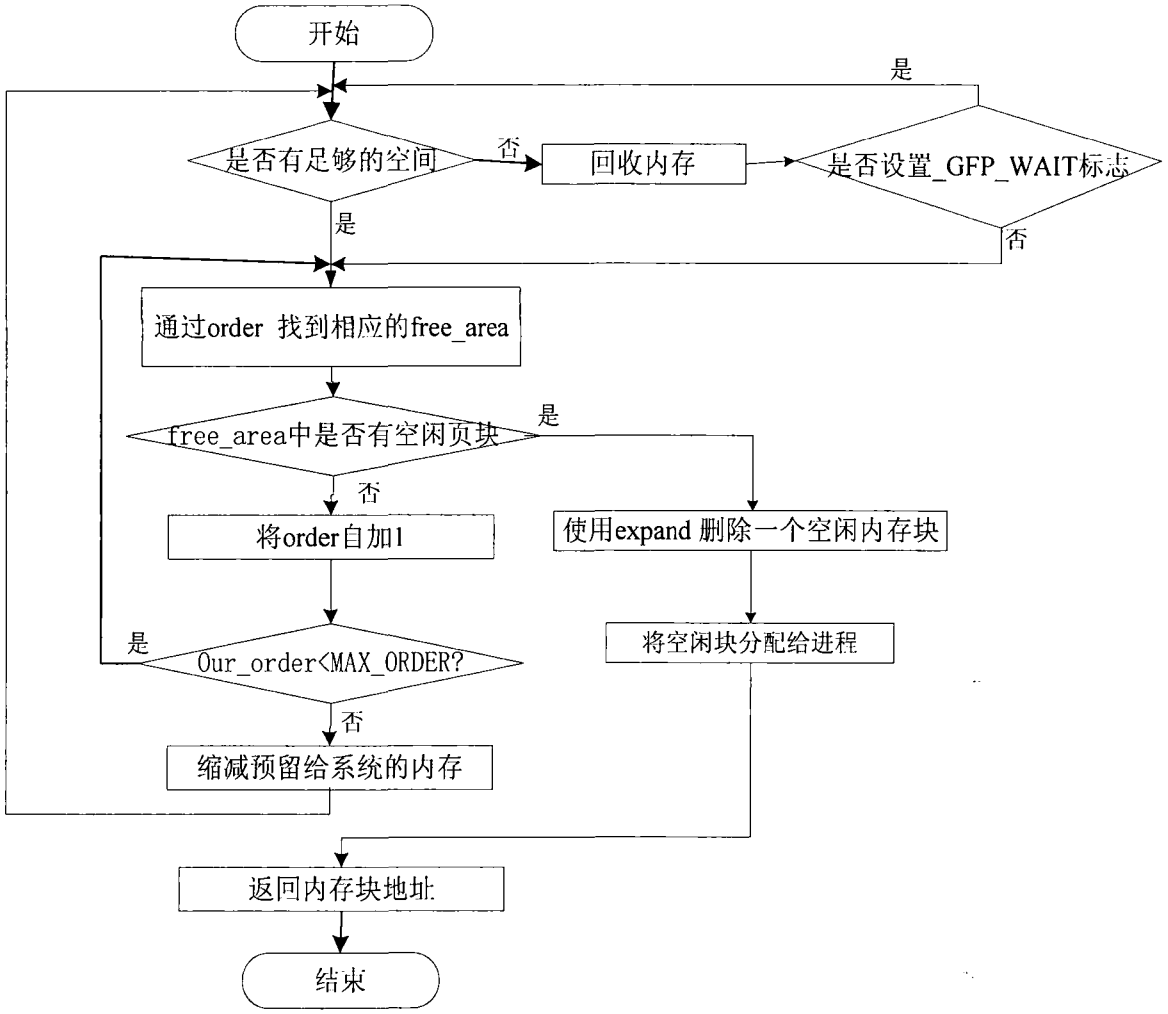


图3.3　分配函数流程图

Free_pages函数的工作过程如下：

1. 页块的首页在mem_map数组中的索引map_br中可以由该页块的首地址计算得到。

2. 如果该页是被内核使用或是已分配的，则作为保留页，设置保留标志，不允许释放，并且函数直接返回开始。

3．一个页块可能被多次引用，所以在进行释放操作时，需要判断该页上的引用计数器，看其是否为0，count域的值变为0。其中count域是用来记录引用者的个数。它存在于页块第一页对应的mem_map_t结构中，当其值为0时，只有该页块才能被释放，否则直接返回。

4. 若该页对应的count域的值为0，则将其释放。同时将与该页块的对应的PG_referenced位消除，该位在mem_map_t结构的flags域中，清楚该位后，则该页块不可

以被引用。

5．把nr_free pages的值加上回收的物理页数，同时对其进行调整。

6．把内存页块存入到数组free_area[order]的相应的队列里。

释放页面的流程如图3.4所示。

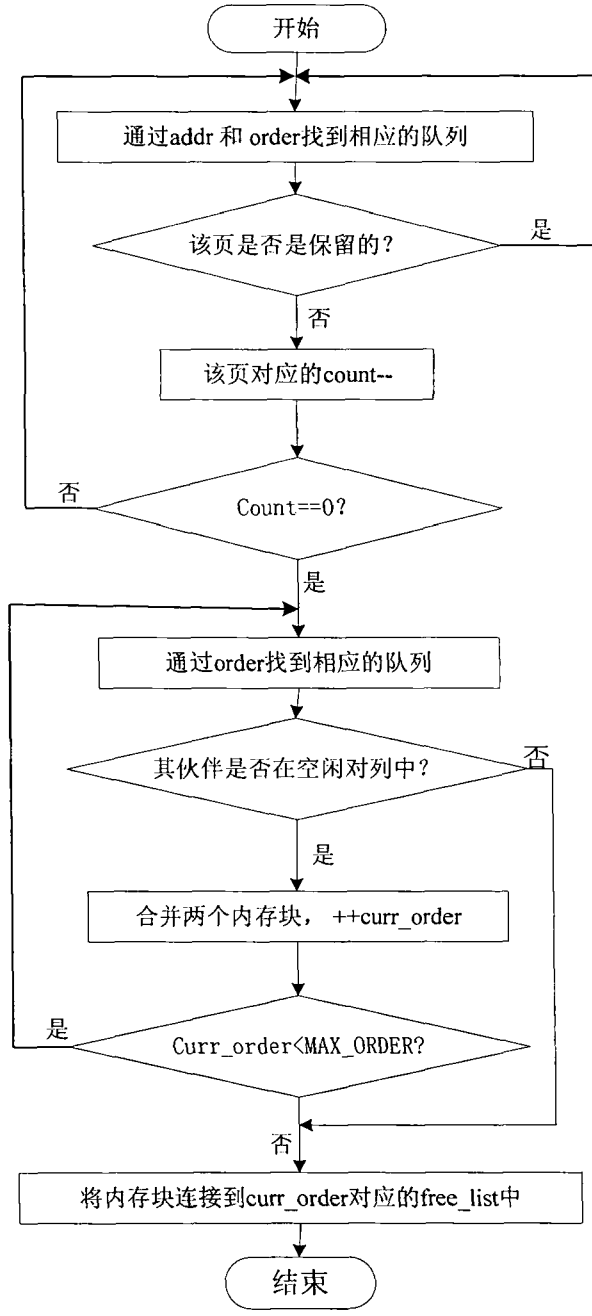


图3.4 Buddy释放页面流程图

在内存的释放过程中，需要对已经释放的所有内存块进行查找，查找其相对应的伙伴块，伙伴如果在队列中，说明该伙伴和其内存块均被释放，则将其合并为更大的内存块。如果合并后的伙伴块，依然满足伙伴块合并的定义，则该伙伴块将继续向上进行合

并操作，直到没有满足要求的伙伴块进行合并时为止，或者是已经达到最高一级的链表，此时则会停止合并。

3.3.5 Buddy 算法的改进思路

伙伴算法虽然是经典的算法，且具有分配和回收速度快，算法简单的优点，并且克服了大量存储空间浪费的问题，但也存在不足之处。专家学者针对其不足之处提出了一些改进策略。

首先，根据伙伴的定义可以得出，必须达到两个块物理地址连续、大小相等以及由同一大块分裂的，满足这三个条件的两块才被称为伙伴块，由此研究者发现了一些缺点，也就是说，内存块如果只满足前两个条件的块则不能进行伙伴块的合并，从而使本来连续的内存空间不能得到充分的利用，减少了内存的利用率[33]。

针对这一缺点，有学者对伙伴块的定义进行了适当的放宽，从而会有更多的机会合并成连续的大块内存。即在满足前两个条件的基础上提出了一种新的伙伴关系的定义。新的伙伴关系在互相邻接的两级链表之间是有效的[34]。为了避免与原伙伴算法位图产生矛盾，整个算法的结构需要改进。如图3.5所示。

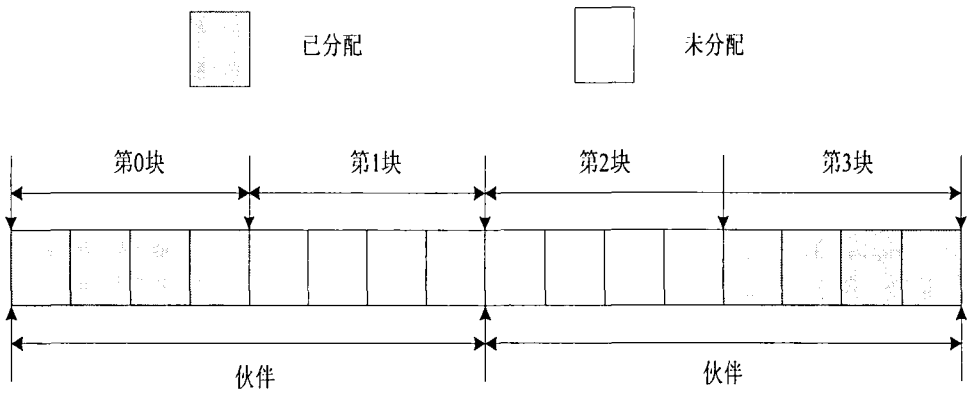


图3.5　部分放宽伙伴的定义块

改进前的伙伴算法由于第一块和第二块不满足第三条，所以不能合并，从而使连续的空间不能得到充分的利用。部分放宽伙伴关系的伙伴的定义，可以把第一和第二块内存块进行合并，从而减少了内存碎片，提高了内存的利用率。

伙伴算法另一个缺点，专家提出了另一种改进的策略，它在进行内存的分配和回收时需要较多的运算，特别是内存块的合并和拆分还涉及到链表和位图的操作。它要完成这些操作需要很大的系统开销[35]。如果一直是小的页面块向上合并成较大的页面块直到最大的页面块时，较小页面块会越来越少。由于系统并不是只申请大的页面块，系统如

果申请较小的页面块，则需要从该链的上级页面块甚至再上一级页块的链表中拆分出与其请求相适应的页块。这种操作会使内存块频繁的合并和拆分，从而增加内存的访问时间。

如果没有合适的内存块满足请求，经典的伙伴算法在管理内存时就需要消耗很大的内存管理时间，其时间主要消耗在分配内存上，并且主要集中在内存块的分裂和内存块的回收时内存块的合并和拆分[36]。针对这个问题，有学者提出了延迟合并算法[37]。这种算法提高了系统的运行效率。

如果对象的创建生存周期很短，经典伙伴算法的运行效率会降低。在这种情况下，由于频繁的申请或释放同样大小的内存块而致使内存块频繁地分裂合并，而使系统产生过大的负载[38]。延迟合并算法释放内存块后，先等待一定的周期，然后再进行内存块的合并，这就使内存块具有一定的生命周期来满足下一次的同种内存的请求，从而减少内存的合并和分裂，提高了系统的运行时间。

3.3.6 Buddy 算法的改进策略

以上针对伙伴算法的不足，分析了专家学者们提出的改进策略，它们都存在一些不足。由于原有伙伴关系的链表和位图操作对时间消耗已经很大，部分放宽伙伴算法而又增加了新的关系，所以将更加加大对时间的消耗，但其优点是提高了内存的利用率。延迟合并伙伴算法通过减少由于频繁访问内存而导致的内存合并和分裂的时间，提高了访问内存的效率。

通过对部分放宽内存管理方法和延迟合并内存管理方法的分析和研究，本节提出了自己的改进策略——基于伙伴系统的部分放宽延迟合并算法。

算法的合理性分析：理论上，部分放宽伙伴算法提高了内存的使用率，降低了内部碎片的产生，但同时也降低了系统的时效；延迟合并算法提高了内存的时效，提高了访问效率；如果采用一个算法来实现这两个算法，可能会从整体上提高系统的性能，有利于多核的并行性的提高。

3.4 SLAB 分配算法

在二进制的伙伴系统中，内部碎片是内存管理不可避免的问题。虽然专家预计在28%的空间中存在碎片，但是与1%的首次适应分配器相比，碎片面积已达到60%。为了减少碎片面积，Linux又采用了SLAB分配算法，它将页面块切割为小的内存块进行分配[39]。Linux内存管理系统采用SLAB算法和Buddy算法相组合的方式来管理内存，从而使

内核中的内碎片保持最低，也更好的满足多核的需求。

3.4.1 SLAB 算法

在Linux内核内存管理器中，SLAB把内存区看作对象，把对象分组放进高速缓存区中，每种对象类型对应一个高速缓存；由小到大的排列顺序为：对象、页框、slab、高速缓存。它们之间是多对一的关系。其结构图如图3.6所示。

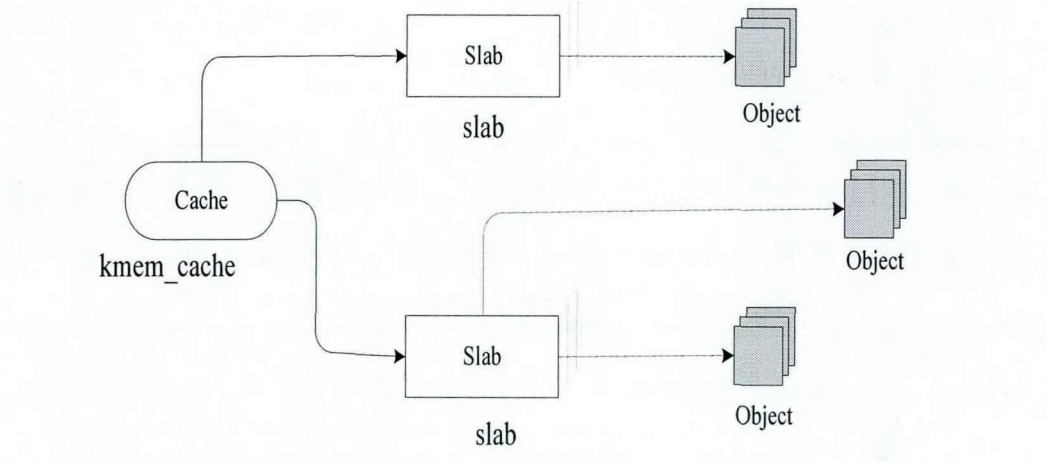


图3.6　Slab的结构关系图

SLAB算法的基本思路：

SLAB算法是为了在系统管理内存时，减少对伙伴系统的调用而提出的。它将那些频繁分配释放的对象进行缓存，下次分配时，可以直接分配，这样不仅从时间上提高了内存系统的性能，而且从空间上提高了内存的利用率，也有利于改善硬件高速缓存的效率，提供了对多核的支持[40]。

分配：

if(适应请求的缓存区有空闲位置)

使用这个位置，不必再初始化；

else{

分配内存；

初始化对象；

}

释放：

在缓存中标记空闲，不再调用析构函数；

资源不足：

寻找未使用的对象空间；

对符合要求的对象做析构；

释放对象占用的空间；

3.4.2 SLAB 的数据结构及关系

SLAB 算法中有两个主要的数据结构，分别是 kemem_cache_t 结构和 kemem_slab_t 结构。kemem_slab_t 结构用于描述 Slab，kemem_cache_t 结构用于描述缓冲区。

下面针对这两个结构进行分析。

1．SLAB

SLAB 是 SLAB 管理模式中最基本的结构，连续的物理页面组成了 SLAB，对象在这些页面中是顺序存放的。在 mm/slab.c 中结构界定如表 3.1 所示。

表 3.1　结构体 kemem_slab_t 中字段的说明

类型	字段	说明
List_head	list	是一个双向链表它将前后两个 Slab 连接起来
unsigned long	colouroff	Slab 上着色区大小
void *	s_mem	是指向对象区的起点的指针
unsigned int	inuse	Slab 中所分配对象的个数
Kmem_bufctl_t	free	其值表示的是空闲对象链中首个对象

Slab 管理结构的存放方式为：on-slab 和 off-slab[41]。其中 on-slab 通常存放小于 1/8 页的小对象，在 SLAB 块中存放 SLAB 管理结构。也就是每个 SLAB 的开头位置存放每块的 SLAB 控制信息 slab_t。描述符在 SLAB 内部时结构情况如图 3.7 所示。

off-slab 则适用于大于等于 1/8 页的大对象，它在 cache_slab 中完成分配对象和 SLAB 块的管理。根据 SLAB 的要求，SLAB 不适合进行大对象的分配和释放。显示描述符在 SLAB 外部时，off_slab 结构如图 3.8 所示。

着色区位于SLAB的首部，它是从每一个SLAB的未用空间中，划出的很少一部分区域。由于每个SLAB都是从一个页面边界开始且每个SLAB又都是由一个页面或多个页面(最多为32)组成。这就使每个着色区的起始地址都按高速缓存中的“缓存行(Cache line)”大小进行对齐[42]。但是，在SLAB中，对象的大小是透明的。SLAB算法设置着色区，目的是将SLAB中第一个对象的起始地址往后推到与缓冲行对齐的位置。由于每个缓冲区有多个SLAB，每个缓冲区中的SLAB着色区大小需要尽量的安排不等，着色区的

作用是让高速缓存中的起始地址错开，使不同的SLAB，处于同一相对位置，从而可以改善高速缓存的存取效率[37]。

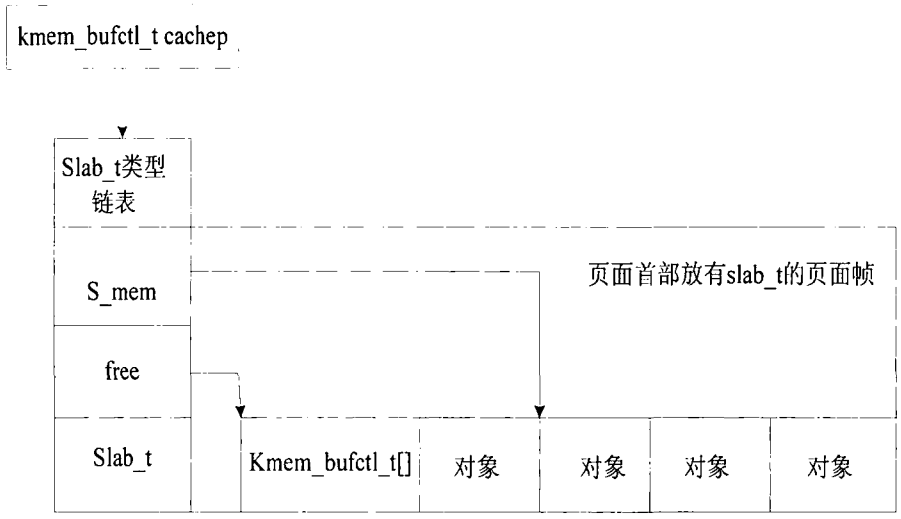


图 3.7　描述符存储在内部的 on_slab 结构

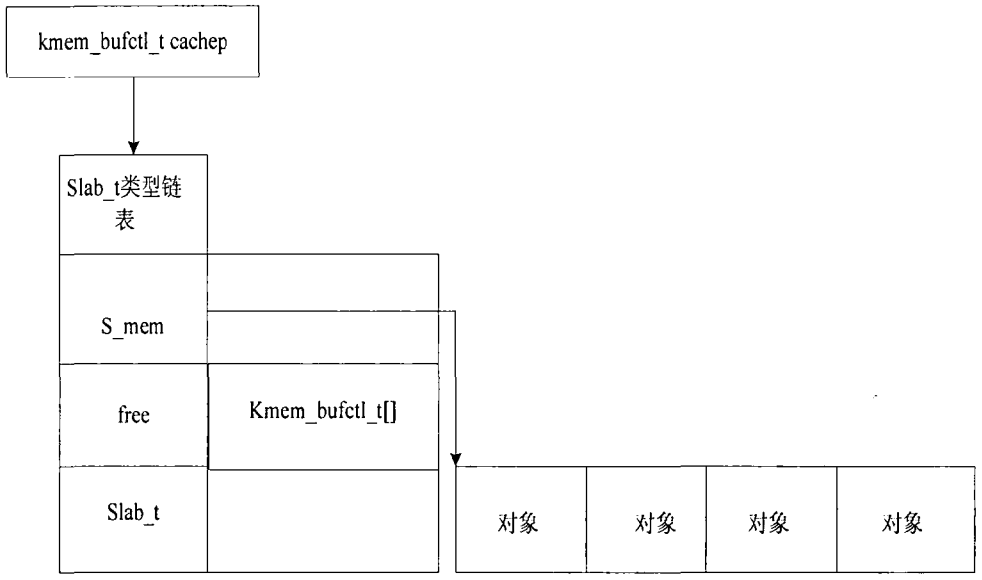


图3.8　描述符存储在外部的off_slab结构

2．Cache结构

缓冲区与SLAB链表是一一对应的，一个缓存区管理一个链表，SLAB分为三组。1)全部对象可分配的空闲SLAB。2)部分对象可分配的半满的SLAB。3)无对象分配的全满的SLAB。这样划分有利于SLAB的管理。其结构定义如表3.2所示。

现有的SLAB算法通常将高速缓存划分为两种：

通用缓存一般是由内存中那些不被经常访问的内存区SLAB创建的，这样做不但可以适量的降低碎片量，提高内存利用率，而且对系统的性能影响不大。

专用缓存是为了更好的提高内存的利用率，降低碎片量，SLAB专门为需要经常被访问的内存区创建了专用的缓冲区，缓冲区的大小是特定的。

符合Slab描述符之间的关系的高速缓存如图3.9所示。

表3.2 结构体kemem_cache_s中字段的说明

类型	字段	描述
struct List_head	slabs_full	全满的Slab
struct List_head	slabs_partial	半满的Slab
struct List_head	slabs_free	空闲的Slab
unsigned int	objsize	原始的数据结构大小
unsigned int	num	每个Slab上的缓冲区数
unsigned int	colour_off	颜色的偏移量
unsigned int	gfporder	每个Slab大小的对数
size_t	colour	着色区的大小
size_t	colour_next	着色区下一个可用位置
unsigned int	flags	标志
char	name	Cache名
struct List_head	next	下一个Cache结构

在内核函数中，当有一个新的对象请求时，系统优先在半满的SLAB中查询是否有满足请求的对象，如果找不到满足请求的，则再到全空的SLAB中查找满足请求的对象；如果在全空的SLAB中也找不到满足请求的对象，则调用Buddy系统申请新的页面生成一个新的SLAB。

3.4.3 SLAB 算法的改进思路

Linux 内核针对 SMP 做了很多改进，SLAB 算法虽然提高了内存的利用率和访问速度，但在当前的共享存储模型的多核系统中，内存被所有的核心所共享，因此多核系统的整体性能，不仅受到内存自身性能的影响，而且还受核间竞争内存情况的约束。在多核系统中，内存管理系统把内存分为核心私有数据和多个核心共享的数据[38]。优化思路包括：一是通过访问私有数据来尽量减少共享数据的访问；二是减少同步/互斥的操作，这是针对共享数据进行操作的；三是提高访问私有数据的性能，这第三点钟可以引入单

核系统优化的一些策略。

Linux 系统为了更好地支持 SMP 和多核处理器，在内核中引入了每 CPU 变量的声明和操作的接口。

作为操作系统核心的部分之一的内存管理系统，除了操作系统使用的内存外，其它部分如果也需要频繁的访问内存时，则内存的利用率及运行效率对系统的整体性能有一定的影响，因此一个高效简单的内存管理系统设计的算法，对于提高多核系统的并行性具有很大的促进作用。

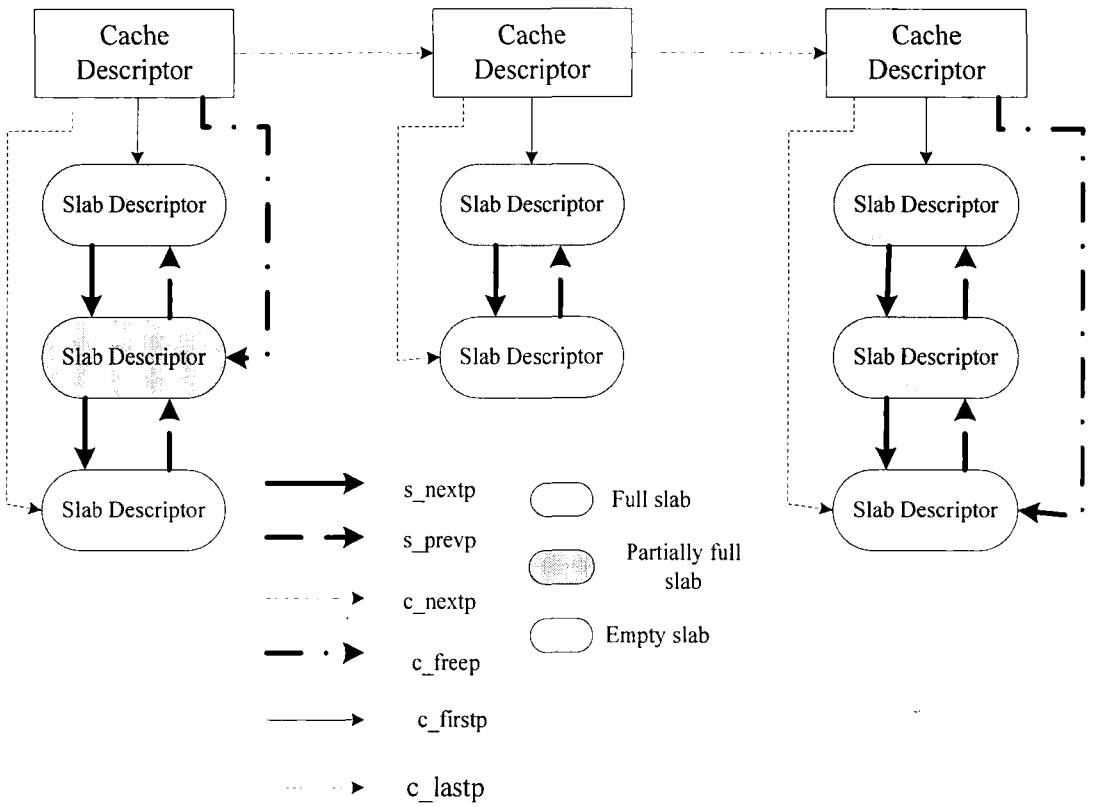


图3.9　Slab描述符之间的结构图

3.5 每 CPU 页框高速缓存

每 CPU 页框高速缓存定义了一个“每 CPU”页框高速缓存，所有“每 CPU”高速缓存包含一些预先分配的页框，它们被用于满足本地 CPU 发出的单个页的内存请求。该内核的实际思路与设计 cache 类似，以便更好让多核处理器的并行性得到更大的发挥。提高页框分配和释放的效率。其原理是把某些页框提前分配给在系统中的每个核中，以便系统在进行分配时优先分配该页框。

每 CPU 页框是为单页准备的(大多数分配都是单页的)，用于避免频繁与频繁伙伴系统核心的代码打交道，从而减少了自旋琐的使用，避免对 cache 的破坏。

每CPU页框Cache被分为热Cache和冷Cache。热Cache包含的页框可能在硬件Cache中，指刚刚被释放的Cache。热Cache对分配后的页框的使用情况是即时写入的，这样能够提高硬件Cache的准确率，从而提高了系统的整体性能；而冷Cache则比较适于不必利用CPU, 也对硬件高速缓存不作任何修改的与DMA操作相似的操作。现在的Linux内核对都是在链表的首部对这两种高速缓存进行增加或减少页框的操作[39]。但是有时这两种操作也在链表的尾部进行，这样的情况目前只存在一种即在冷Cache标志下被指定了_GFP_COLD的情况下，这两种操作才在链表的尾部进行。Linux界定了两种类型用于每CPU页框Cache的表示。其结构体如下：

```
Struct per_cpu_pages{
    Int count;                  //缓存中空闲页数目
    Int high;                 //空闲页数的上限，如果超过这个值，则被删除
    Int batch;                  //在对页框进行修改时变动的数量
    Struct list_head list;     //空闲页框链表
  }
struct per_cpu_pageset{
    struct per_epu_pages pcp;
#ifdef CONFIG_NUMA
    s8 expire;
#endif
#ifdef CONFIG_SMP
    s8 stat_threshold;
    s8 vm_stat_di 州 R_VM_ZONE_STAT_ITEMS];
#endif
}_cacheline_aligned_in _smp;
```

这个结构体只用在 struct zone 中，通过 zone__pep 宏访问。为内存管理区中每个CPU 定义页框 Cache, 即字段 struct per_cpu_pageset[NR_CPUS]。

static void free_hot_cold_page(struet page *page, int cold)函数用来向 Cache 中释放页框，其中 cold 的标志为：在缓存的空闲链表中 page 指向的被释放的页框的在该链表中是开始还是结尾[40]。如果未被使用的页框，在释放该页框后，其个数大于指定的上限值，则将指定数量的页框释放给 Buddy 系统。这里利用 free_pages_bulk 函数释放 pcp->batch 指定值，其工作流程如图 3.10 所示。

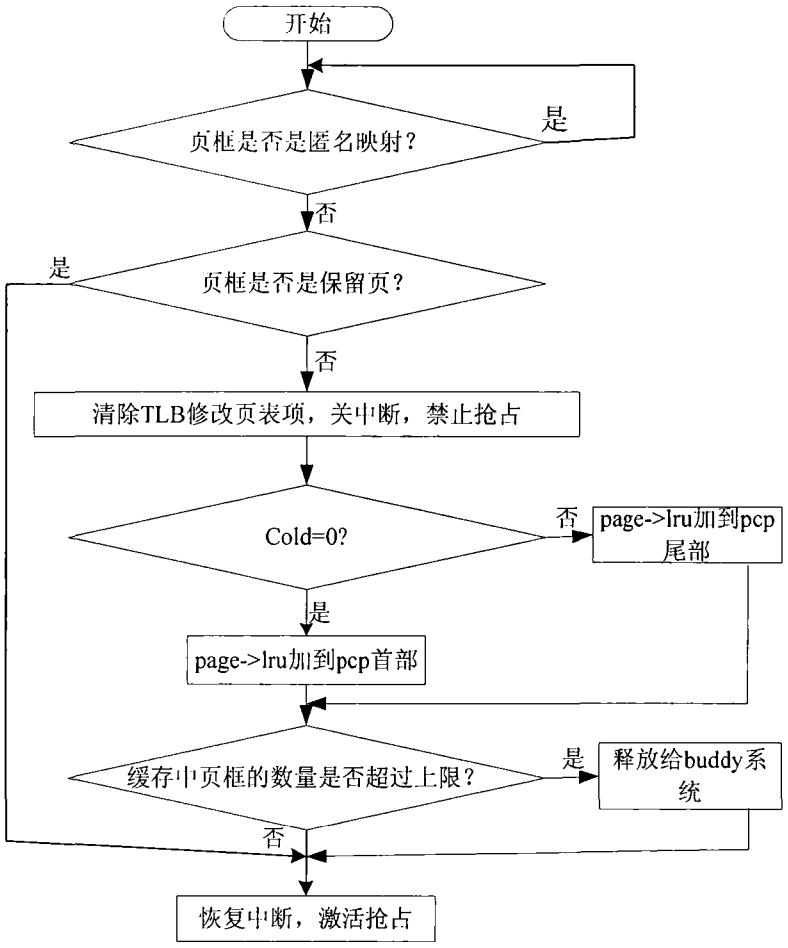


图 3.10　每页框高速缓存流程图

3.6 本章小结

本章以 Linux 源码为基础，并对 Linux 内存管理算法进行了研究和分析，重点对 Buddy 算法和 SLAB 算法进行了深入的解析，接下来对多核支持的每页框高速缓存结构进行了设计，论述了部分放宽伙伴算法和延迟合并算法的优缺点，并针对这两种算法的特点，通过改进 Buddy 算法提出了自己的改进思路。最后，为了提高多核系统的并行性，对 SLAB 算法提出了自己的想法，改进的思路从理论上可以提高内存的利用率，使多核系统的并行性得到更好的发挥。

第 4 章　改进的 Linux 内存管理算法

通过以上对 Linux 内存管理算法的深入研究，本章针对经典的 Buddy 算法和 SLAB 算法在多核处理器上存在的不足进行了改进，分别提出了部分放宽延迟合并算法和基于 SLAB 的改进算法，并搭建的实验模拟平台，对该算法进行性能上的测试。

4.1 部分放宽延迟合并算法的实现

本节对基于伙伴系统的部分放宽延迟合并算法进行了分析和设计，主要包括对提出的算法的数据结构、内存的分配流程和内存的释放流程等进行设计。

4.1.1 数据结构的改进

通过对第 3 章中伙伴算法的分析及改进策略的描述可以看出，Buddy 算法存在着一定的不足，因此，通过对 Buddy 算法的改进，提出一种新的基于伙伴系统的部分放宽延迟合并的伙伴算法就成为本节研究的重点。为了方便对现有内存块进行管理和维护，本文对现有的 Buddy 算法的数据结构进行了相应的改进，free_area 的数据结构如下：

```
Typedef struct free_area{
    struct list_head active_list;
    struct list_head inactive_list;
    struct list_head ex_free__list;
    unsigned int      *map;
    unsigned int      *ex_map;
}free_aree_t;
```

其中active_list 表示满足原伙伴块定义的伙伴块中有一个伙伴正在被访问，另一个伙伴块处于空闲状态的队列；inactive_list 表示满足原伙伴块定义的其中伙伴块都未被使用的伙伴块中地址较小的那个空闲伙伴；ex_free__list 表示新增的双向链表；ex_map 表示满足放宽伙伴块定义的伙伴位图；map 表示满足原伙伴块定义的伙伴的状态位图。Buddy算法修改后的队列结构如图4.1所示。

下面对部分放宽延迟合并算法中的链表和位图进行详细的阐述：

首先，active_list，inactive_list 和 ex_free__list 都是 list_head 类型，对应的空闲内存块的大小都是 2^n，其中，active_list 和 inactive_list 链表所链接的伙伴块符合 Buddy 算法对伙伴块的定义，而 ex_free__list 链表是满足部分放宽 Buddy 算法中伙伴的定义。

ex_free__list 链表中的空闲块是由 free_area[n-1]中的 active_list 和 inactive_list 的空闲内存块按照放宽伙伴算法定义的伙伴进行合并而组合而成的。

其次， ex_free__list 都是来自下一级的 active_list 和 inactive_list，active_list 和 inactive_list 对满足原 Buddy 关系的伙伴内存块进行合并，形成更大的空闲内存块。而 ex_free__list 链表中合并的伙伴块都是满足放宽 Buddy 中伙伴块的内存块定义，它并不删除在 active_list 和 inactive_list 链表中满足放宽伙伴关系定义的内存块，而是继续保留在 active_list 和 inactive_list 中，同时将合并后的更大内存块添加到 ex_free__list 的上一级中。

最后，map 和 ex_map 都是表示空闲内存块中伙伴的使用状况，map 位图表示的是满足原 Buddy 关系中满足伙伴定义的空闲块中伙伴的使用状况；而 ex_map 位图则是对满足放宽 Buddy 关系中伙伴定义的空闲块中伙伴的使用状况。

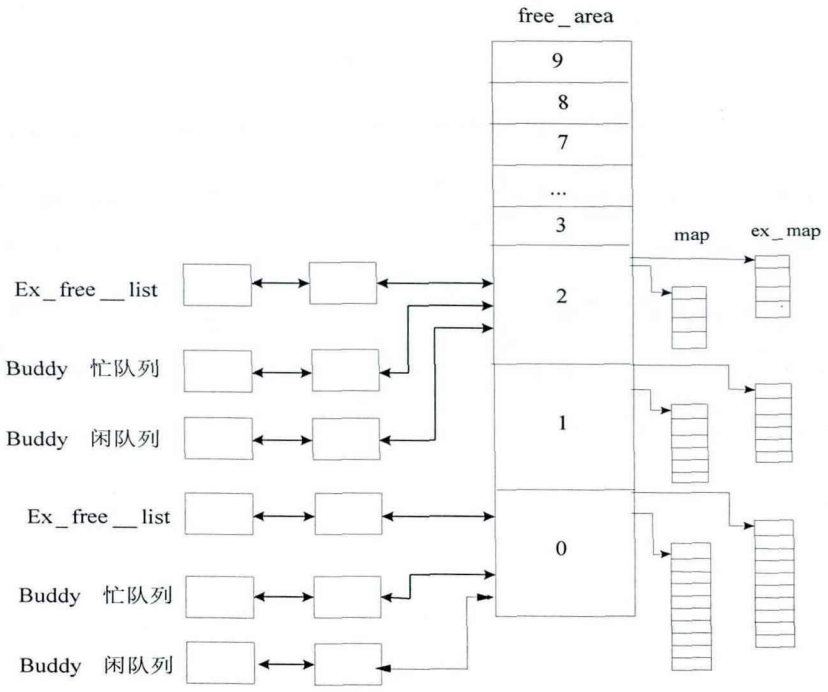


图 4.1 修改后的 Buddy 算法结构

4.1.2 内存分配的改进

上一小节中详细分析了部分放宽延迟合并算法，并写出了该算法的数据结构，其中，加了对满足部分放宽伙伴关系中伙伴块管理和延迟合并策略的数据结构，下面对该算法的内存分配函数进行详细的论述，其总体思路是：内存搜索策略不变，还是运用原伙伴算法的内存搜索策略来查找满足请求的空闲块，首先在 Buddy 的忙队列链表中查找满足请求的空闲块，如果其中没有满足请求的内存块，则进入 Buddy 的闲队列链表中查找相

应的空闲块，如果在以上两个链表中查到满足请求的内存块，则内存分配成功，否则，进入到 ex_free_list 链表的空闲内存块中查找满足要求的空闲块，查看该链表中是否有满足部分放宽伙伴关系定义的内存块，如果有，则分配成功，这样相当于给内存又增加了一次分配机会，从而一定程度上可以提高内存的利用率。否则，按照上述查找流程到上一级链表中继续进行查找。下面给出了部分放宽延迟合并的伙伴算法内存分配函数的流程图，如图 4.2 所示。

由于把满足原伙伴关系的链表进行了分类，所以其链表的执行流程图也有一定的改进，其中 inactive_list 链表流程图如图 4.3 所示。

4.1.3 内存释放函数的改进

上节中，对部分放宽延迟合并伙伴算法内存分配工作流程进行了分析，并给出了其执行流程图，下面对其内存释放过程进行详细的分析。首先通过 order 找到相应的队列，看它是否在空闲队列中，此时，可以分为两种情况：1．如果在空闲队列中，则看 order 的值是否达到最大，如果达到最大，则将释放的页面链接在闲空闲链表上，如果不是最大，判断当前位图的伙伴块是分配的还是为分配的，如果是未分配的，则放入忙空闲链表上，如果伙伴是分配的，则释放内存块，链接到闲空闲链表上；2．如果不在空闲对列中，则看是否在部分放宽的伙伴队列中，如果在则将其合并后添加到 ex_free_list 链表中，如果不在则返回。流程图如图 4.4 所示。

4.2 SLAB 算法的改进

为了更好的支持多核处理器，提高系统的并行性，Linux 系统内核引入了每 CPU 变量的声明和操作接口。但是在 SLAB 算法中缓冲区内存的回收比较复杂，特别是对多个 CPU 和多核处理器显得更为复杂。本节中摒弃了 SLAB 中的空闲对列，引入了本地活跃的 SLAB，也就是为每个处理器或核心设置自己的一个本地活跃的 SLAB。

4.2.1 改进 SLAB 的架构

改进 SLAB 算法中，每个节点维护一个半满队列，每个核心拥有自己的半满队列。

在这个算法中，有两个关键的锁要求注意：

1．slab_lock：它是用来保护元数据的，该数据是在某个对象所在的 SLAB 和它的所在页的页描述符里。如果拥有这个锁，则不能分配和释放该对象，也不能对该对象进行加入或移出半满队列的操作。

2．slab->list_lock：它是用来保护 SLAB 计数的，该计数存在于每个节点的半满队列里。如果拥有该锁，则不能对半满队列作添加 SLAB 或删除 SLAB 操作，也不能对该队列里的 SLAB 计数进行修改。这是一个全局锁，实际设计中应该尽可能少的应用该锁。

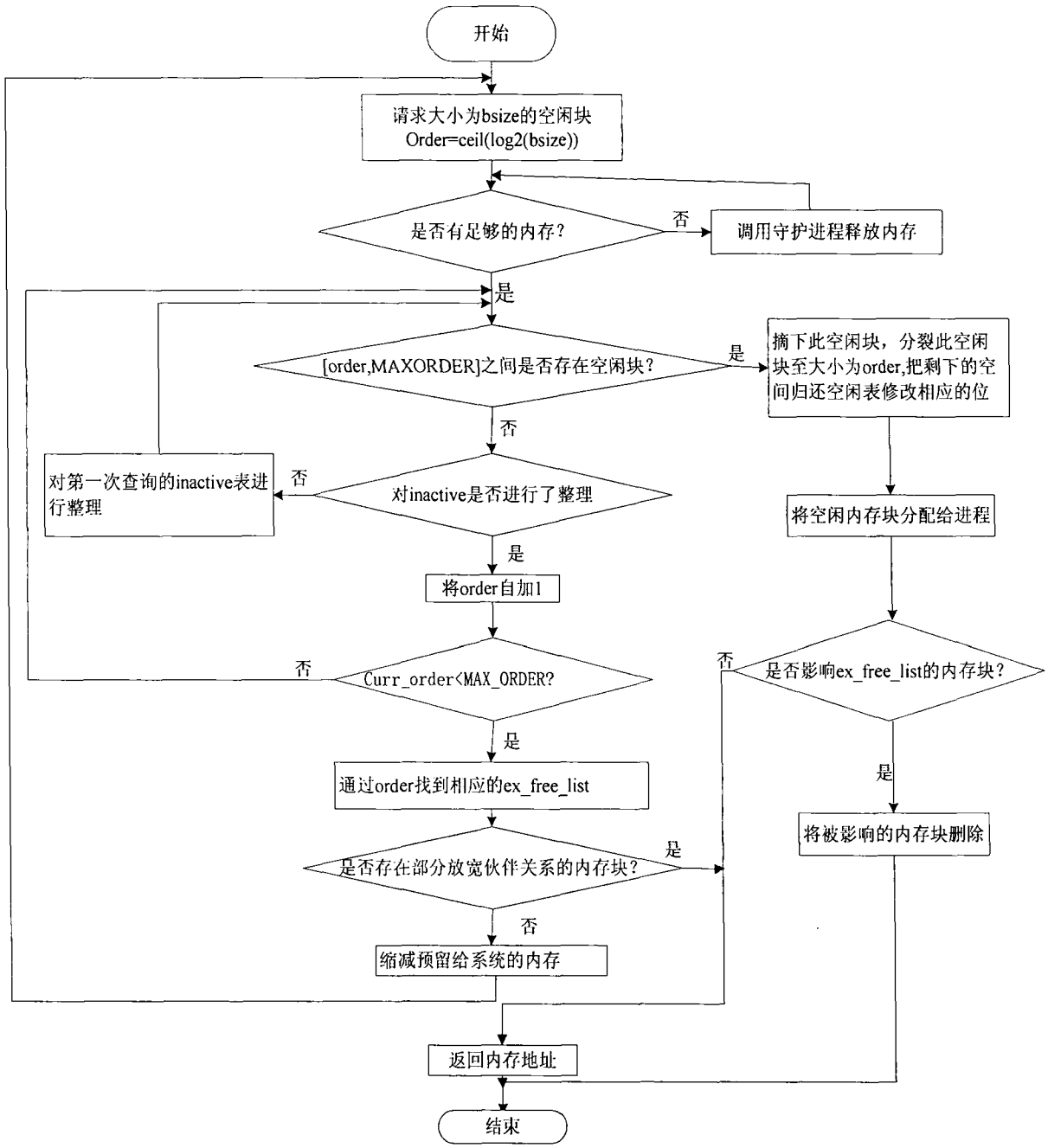


图 4.2 部分放宽延迟合并的伙伴算法内存分配流程图

一般情况下，它们的使用顺序是先使用 slab_lock，后使用 slab->list_lock, 但也有个别的情况。例如，如果在队列中取出 SLAB 时，则首先在队列里获取 slab->list_lock 锁，然后从里边搜寻可用的 SLAB, 如果找到相应 SLAB，则有可能会被释放该 SLAB，只有拥有了该 SLAB 的 slab_lock，这时才可以对其进行操作。如果把所有的队列都查询完了，

仍然没有找到可用的 SLAB，就需要调用伙伴系统为其分配新的 SLAB，以供其使用。其架构图如图 4.5 所示。

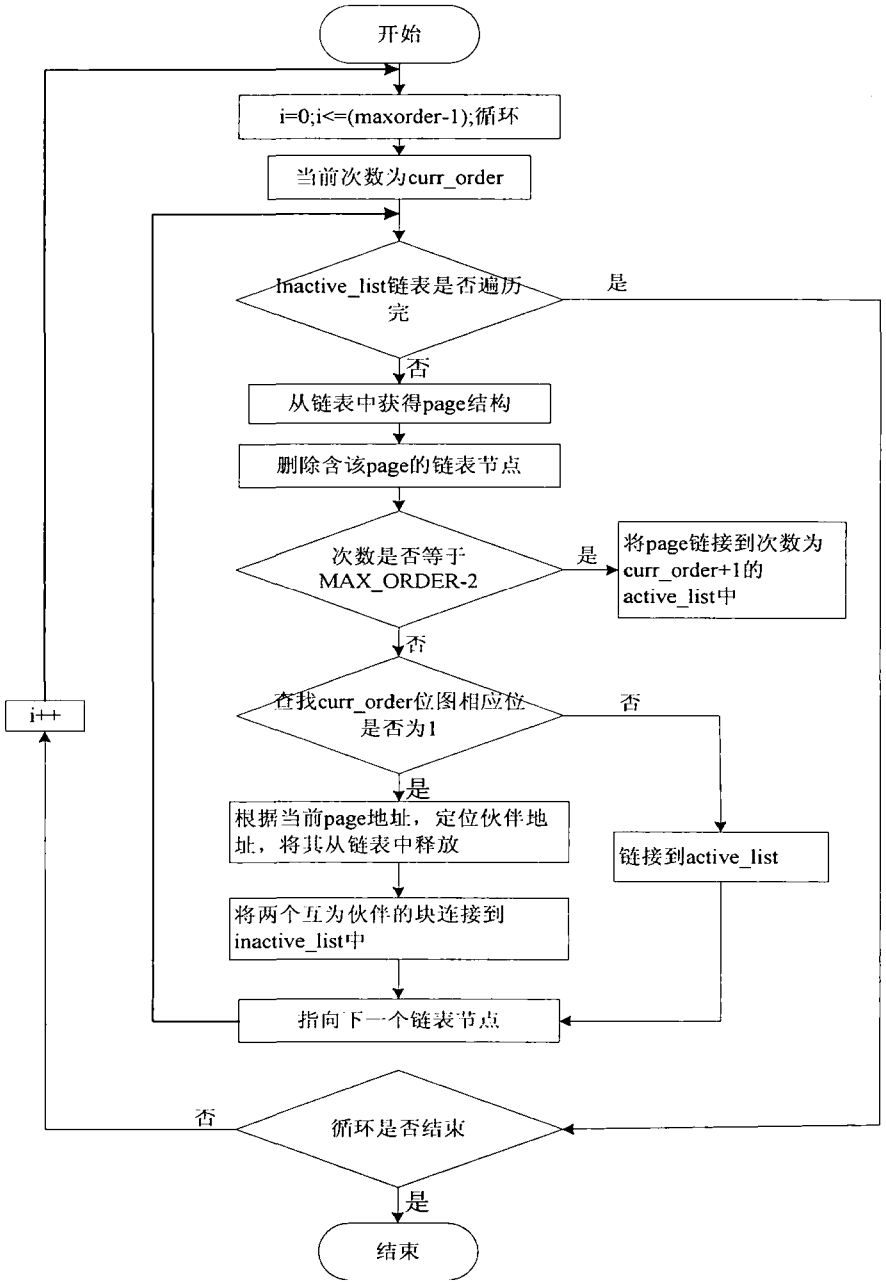


图 4.3　inactive_list 链表整理流程图

在上述改进算法的基础上，本文进行了新的改进，其改进算法的架构图如图 4.6 所示。每个核心拥有一个本地的 SLAB，内存块在进行分配时，首先在本地 SLAB 中查找要的获取对象，如果本地的 SLAB 已经被用完，则需要去半满队列中查询，在这需要对半满队列设置互斥锁，然后扫描整个半满队列，看是否有满足需求的空闲对象，如果有满足的对象，则直接进行分配，如果没有满足的对象，则需要 Buddy 系统分配一个新的

SLAB 放入本地的 SLAB 中。如果分配对象过于频繁，则可能会导致本地的 SLAB 迅速被用完，以至经常扫描共享的半满队列，从而造成频繁的在半满队列上进行加锁/释放锁的操作，这样，就会影响多核环境下内存和核的并行性的发挥。为了强化内存在多核系统中并行性的分配，本文引入了局部化半满队列 SLAB，即每个核拥有自己的半满队列，以此来消除互斥的开销。

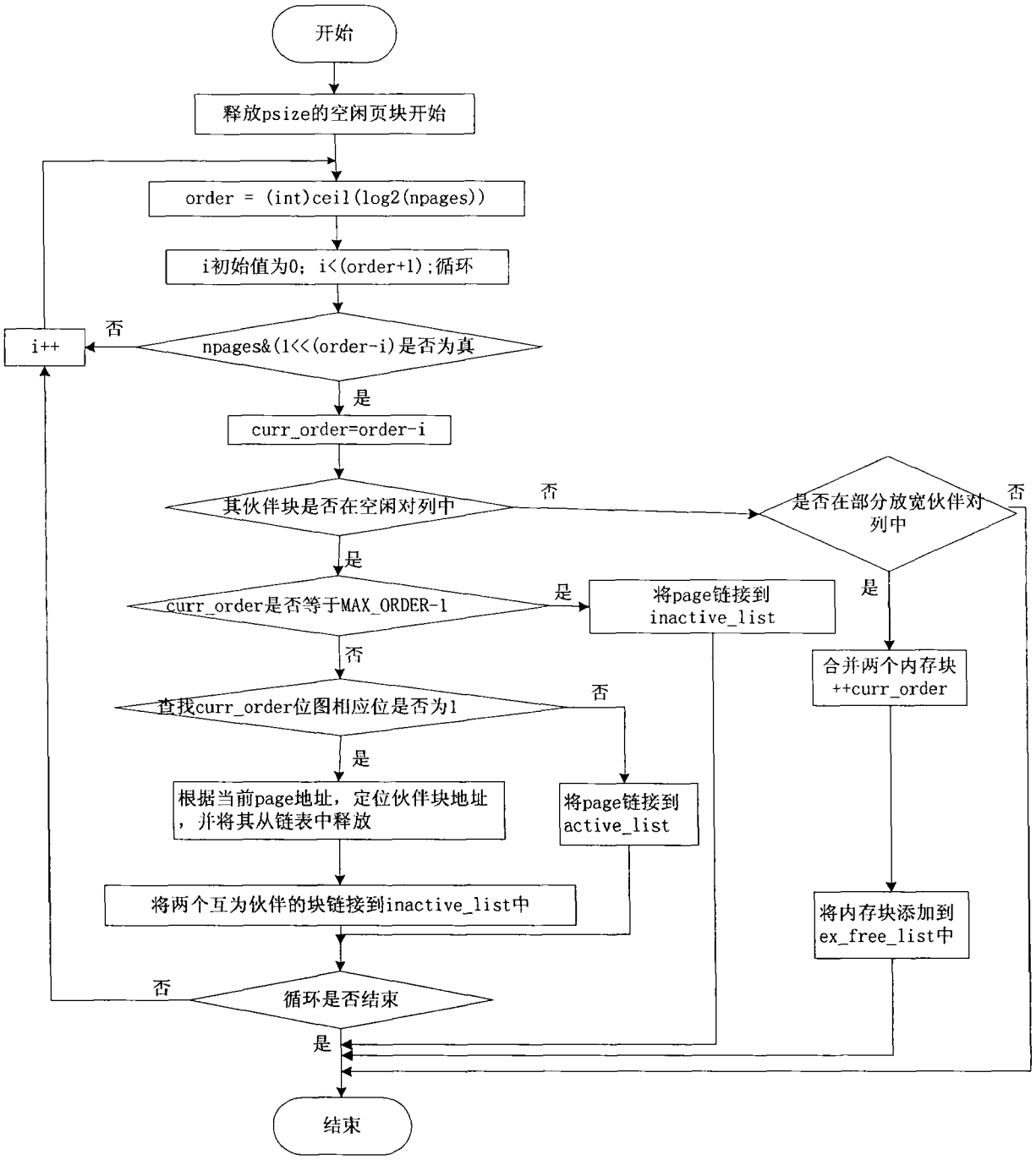


图 4.4　部分放宽延迟合并的伙伴算法内存释放流程图

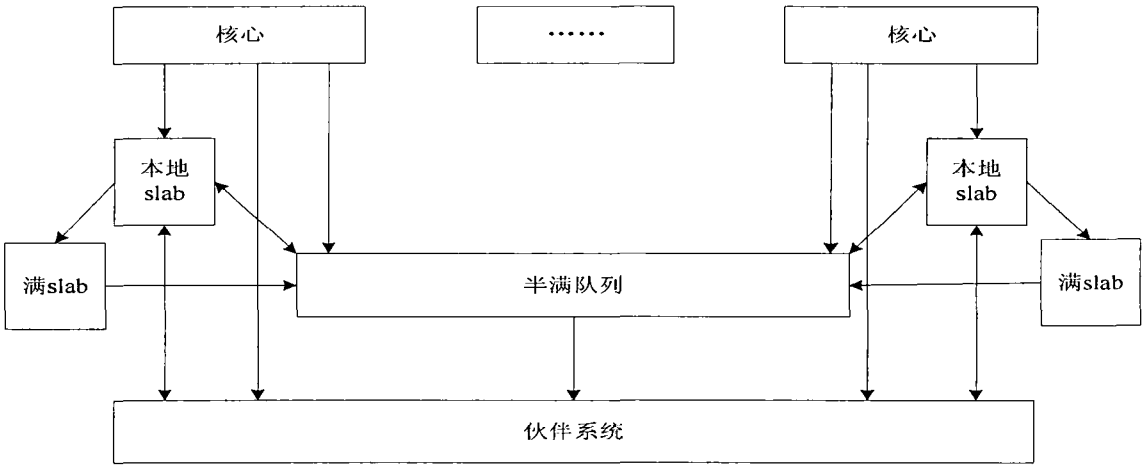


图 4.5　改进的 slab-1 架构图

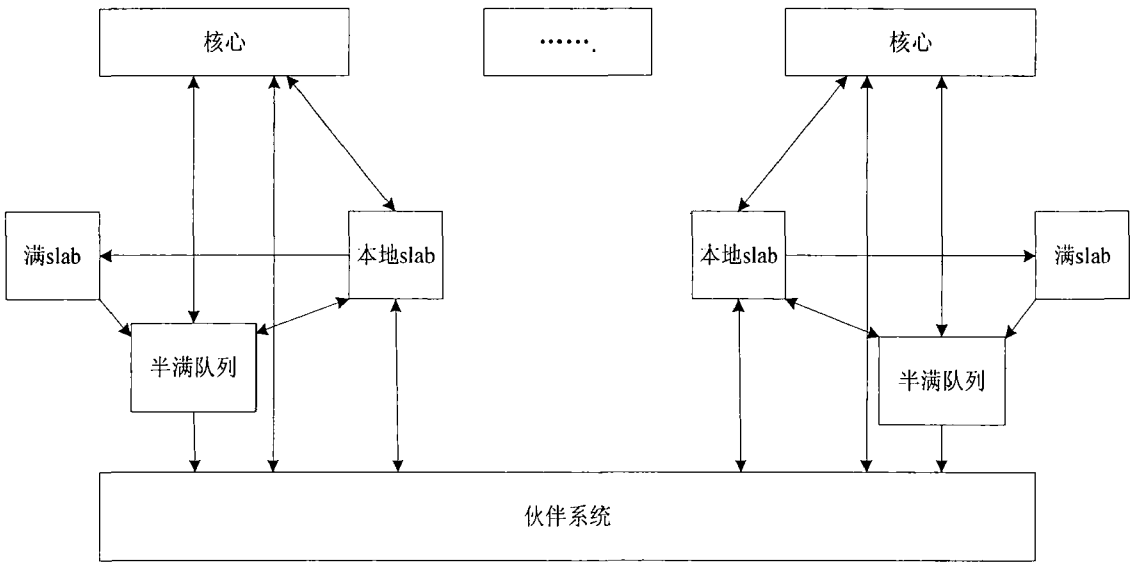


图 4.6　改进后的 Slab-2 架构图

4.2.2 改进的 SLAB 算法的数据结构

修改位于 kmem_cache_cpu 的数据结构，如表 4.1 所示。

该 SLAB 算法针对多核处理器/CPU 进行了优化设计，提高了多核系统运行的并行性，减少了内存空间的浪费，同时简化了 kmem_cache 数据结构的设计，为 SLAB 分配器的发展带来了新的挑战。由于改进的 SLAB 算法中所有接口的 API 函数与原 SLAB 算法相同，从而保证了内核的其它模块能顺利迁移到改进算法的 SLAB 上。

kmem_cache 类型的数据结构描述每个处理器/内核中对象缓冲区：

```
struct kmem_cache{
unsigned long flags;          //指一组标志用于描述缓冲区属性
```

```
int size;        //对象内存的大小
int objsize;     //对象的实际大小
int offset;      //指指针的位移用于存放空闲指针的对象
int order;       //表示一个 slab 需要 2^order 个物理页面
kmem_cache_node local node; //指刚创建 Cache 节点的 slab 信息
int objects;     //一个 slab 中的对象总数
gfp_t allcflags; //一个标志在创建 slab 时使用
int refcount;    // 缓冲区计数器。
void(*)void(*) ctor; //构造函数其作用在于在创建 slab 时初始化对象
int inuse;  //元数据的位移
int align;  // 对齐
const char*  name;  //缓冲区名字
struct list_head list; //双向循环队列——其包含所有缓冲区描述结构的
kmem_cache_core *[NR_CPUS] cpu_slab;//每个处理器/核心的对应的本地上 slab
}
```

表 4.1 结构体 kmem_cache_cpu 中的字段说明

类型	字段	说明
void**	freelist	指向首个每 CPU 空闲对象的指针
page*	page	当前正在分配的页
int	node	Page 所属的节点
unsigned long	min_partial	半满 slab 对象数目的下限
unsigned long	nr_partial	半满队列中 slab 数目
struct list_head	partial	半满对列
unsigned int	offset	空闲对象偏移
unsigned int	objsize	对象的大小

改进后的 SLAB 算法不再使用 struct kmem_cache_node, 一般在自己对应的核心中进行释放和分配操作，较少的调用伙伴算法进行分配和释放内存块的操作。

改进的 SLAB 算法对原有的 SLAB 算法的初始化函数和销毁函数进行了改进，同时对分配和释放对象的过程进行了相应的修改，由于每个核拥有自己的半满队列且需要与核心中自己的本地 SLAB 进行转换，所以还要对半满队列的清理函数进行修改。

系统启动时，在start kernel函数中调用kmem_cache_init()，并对kmalloc_cache数组

进行初始化，而后这个函数不断的调用create_kmalloc_cache()来初始化kmalloc_cache数组中的每个成员，其中create_kmalloc_cache()函数最重要的代码片段如下：

```
for(i=KMALLOC_SHIFT_LOW; i<=PAGE_SHIFT; i++){
create_kmalloc_cache(&kmalloc_caches[i],” kmalloc”, 1<<i, GFP_KERNEL);
caches++;
)
```

KMALLOC_SHIFT_LOW 的默认值为 3，这样 kmalloc_cache 数组的第 3 个成员维护对象的大小为 2^3 个字节，一直到第 11 个成员维护的对象大小为 2^{11} 个字节，空出来的第 1 和第 2 个成员分别维护的对象大小为 96 和 128 个字节。PAGE_SHIFT 被定义为 12，这样出去第 0 个，还剩 11 个 slab 分配对象，slab_cache 指向这 11 个对象的头部。

create_kmalloc_cache 函数又调用函数 kmem_cache_open 来完成缓存创建。

kmem_cache_open 函数先后调用 calculate_size 计算 SLAB 中页框的数目及对象的大小；接着调用 init kmem_cache_nodes 完成对各 SLAB 节点的初始化；调用函数 alloc_kmem_cache_cpu 为每个核心建立自己的本地的 SLAB，现在改进后的 SLAB 算法不需要维护每个 SLAB 的节点信息，所以改进的 SLAB 删掉了在 kmem_cache_open 函数中对 init kmem_cache_nodes 的初始化调用，并在 alloc_kmem_cache_cpus 函数中，加上了对半满队列的初始化。

其实现代码如下：

```
For_each_inline_cpu(cpu){
    c=alloc_kmem_cache_cpu(s,cpu.flags);
    s->cpu_slab[cpu]=c;
}
```

alloc_kmem_cache_cpus 函数调用 alloc_kmem_cache_cpu。而 alloc_kmem_cache_cpu 则调用函数 init kmem_cache_cpu, 最终完成对每个核的本地 SLAB 的初始化。由于每个核拥有自己的半满队列，所以在这里需要在函数 init kmem_cache_cpu 中的初始化代码中加入对半满队列的初始化，初始化半满队列的代码，如下：

```
static void init_krnem_cache_cpu(kmem_cache*s; struct kmem_cache_cpu*c)
{
    ……
    INI_IST_HEAD(&c->partial);
    e->nr_partial=0;
```

```
    ……
}
```

已经存在的 SLAB 缓存使用 kmem_cache_destory 函数进行销毁，它会调用 kmem_cache_close 函数，并且用它清理每个核所对应的半满对列，本文把遍历所有节点的函数改为遍历所有核心，其代码如下：

```
for_each_inline_core(core){
    c=get _cpu_slap(s,core);
    free_partial(s,c);   //释放该核的半满队列
}
```

由于每半满队列都有自己的核与之对应，所以去掉了原算法对半满队列和页的加锁操作，核在其半满队列中获取 SLAB 的函数代码如下：

```
static struct page*get_partial_node(struct kmem_cache__core *c){
    if(!c ||!c->nr_partial)
    return NULL;
  retunl list_entry(c->partial.next,page,lm);
 }
```

由于修改后的 SLAB 算法拥有自己的半满队列，所以在释放自己半满队列中空闲的 SLAB 时，同样不需要互斥加锁，遍历代码如下：

```
for each_inline_cpu(core){ / / 遍历所有核心
        struct kmem_cache_core *c=get_cpu_slab(s，core);
        if(!s->nr_partial)
        continue;
    list for each_entry_safe(page,t,&c->partial,lru ){ / / 遍历核心的半满队列
        if(!page->inuse){ / / 释放空闲 slab
        list_del(&pag->hu);
        c->nr_partial--;
        discard_slab(s，page);
         }
    }
}
```

改进后的 SLAB 算法与原 SLAB 算法可以使用相同的接口，不会影响其他部分代码

的实现功能。这样的改进减少了多核之间因为抢占共享数据而产生的冲突，也消除了每个核访问部分空队列对象时所带来的自旋锁的开销。同时减少对伙伴系统的调用，从时效上提高了内存的访问速度，也提高了内存的使用率，一定程度对多核系统并行性的提高起到了一定的作用。

4.2.3 SLAB 算法的分配请求和释放

假设在 start_kernel() 中，内核对 11 个 SLAB 进行了初始化，并使所有初始值为空，下面是本文对分配请求过程进行描述：

1．首先根据分配请求的字节大小，确定 kmem_cache 结构。

2．查看 kmem_cache 结构中的 freelist，可以根据 kmem_cache 和当前核已经确定的 kmem_cache_cpu 结构，如果该结构中的 freelist 不为空，则直接从 freelist 上分配，初始化时该指针为空。

3．检查该核对应的 partial 链是否为空，不为空则直接从该链上分配，并假设初始时该链的分配对象为空。

4．进行页面初始化，其大小为 size 的 object 链表，同时“批发”2^{order} 个页面，将 kmem_cache_cpu 结构中的 page 指向首个页面对应的 page 结构。Page 结构中的 freelist 结构指向一个 object。page 结构中的 inuse 初始化为这个 Page frame 所包含的 object 的总数。

5．首先对 object 进行分配，然后把 page 结构中的 freelist 修改为空，并使 kmem_cache_cpu 的 freelist 指向 page Frane 中的第二个 object。如果以后还想继续分配，则从 kmem_cache_cpu 结构中的 freelist 上进行分配。

以上介绍的是分配的流程，下面分析他的释放流程，假设 Page Frame 中的全部 object 都被分配了并且没有被释放，如果有新的分配请求，就分配新的 Page Frame，同时把 kmem_cache_cpu 中的 page 指针指向新的 page 结构，SLAB 并不记录每个 Page Frame 的相关信息，这些信息的释放函数 kfree()可以根据 object 的首地址进行计算，下面对页面的释放步骤进行描述：

1．根据要释放的 object 的首地址计算出 PFN,并根据该 PFN 得到 page 的结构指针。

2．如果根据 PFN 得到的 page 结构指针与当前 kmem_cache_cpu 中的 page 指针一致，就把释放的对象直接添加到当前的 kmem_cache_cpu 中的 freelist 链表中，并结束以上过程，否则继续进行下面的步骤。

3．检查当前要释放的 object 的 page 中 freelist 是否为空，如果为空，则说明这是第

一次释放某个 Page Frame 中的对象，就把这个 page 结构添加到 partial 链上，并且把 page 结构中的 freelist 结构指向这个 object，在下次释放 page 结构中的 object 时，如果发现 page 结构中的 freelist 不为空，就可以把这个 object 直接链接到 page 结构中的 freelist 链上。

4．page 结构中的 inuse 初始化为某个 page Frame 所包含的 object 的总数，现在每释放一次就把 inuse 减 1，并查看 inuse 是否为 0，如果等于 0，则说明某个 Page Frame 中的 object 全部被释放了，此时就可以把这个 Page Frame 释放到整个页面管理块中。

4.3 性能测试

通过使用 C 语言编程，并利用 SimOS 模拟器在 Linux 系统环境下对原伙伴算法和改进后的部分放宽延迟算法进行仿真实验。该实验虽然没有把该算法放置在 Linux 内核里的多核环境下进行测试，但在一定程度上可以从运行时间的角度对比出改进的算法与原算法相比在对多核系统的支持上具有一定的优势，同时该仿真平台对内存碎片产生量的仿真测试还是比较准确的。

下面是进行测试前宏的定义

```
#define BUDDYSIZE 64*1024*128
#define PAGESIZE 128
#define BUDDYPAGES(BUDDYSIZE/PAGESIZE)
#define MAX_ORDER 10
```

仿真平台：

虚拟机配置：VMware-workstation-5.5.1、Redhat9.0、SimOS、Gcc-3.4.3

Windows 作为主机配置，操作系统是 windows XP、处理器是 Intel(R) Core(TM) Duo CPU T 2600、内存大小为 1G。

性能指标：分配的时间和内存碎片。

4.3.1 分配时间

有两种方法可以用来衡量时间的开销：一是通过比较分配或者是释放时的搜索链表次数、伙伴块裂开次数和伙伴块合并次数，二是直接通过仿真测试分配和释放的时间[41]。

利用仿真测试，在测试平台中输入随机的分配请求页面，请求页面的随机范围在 1 到 512 之间，通过仿真实验的结果表明：随着输入次数的变小，时间开销抖动会逐渐变大。由表 4.2 中测试结果的对比数据可以得出两种算法的在多核环境下内存的分配时间

都大于内存的释放时间，并且新的算法无论在内存分配还是释放时所用的时间开销均高于原算法的时间开销。产生以上现象的原因是两种算法在内存分配时，都是先找到合适大小的内存块，然后才有可能对块进行裂开和进行位图的相关操作，而内存的释放无需查找合适的内存块搜索链表，直接进行测试和位图的设置，最后对内存块进行合并；改进的算法在分配和释放过程中，如果在满足原伙伴算法的内存块中没有搜索到合适的伙伴块，则需要在满足部分放宽伙伴算法的伙伴块链表中搜索合适的伙伴块，从而增加了内存分配和释放时的时间消耗。

如表 4.2 所示。

表 4.2　算法时间消耗测试对比图

算法种类	次数	分配时间 us	释放时间 us	总时间 us
原伙伴算法	200000	1.294375	1.205468	1.437847
部分放宽延迟合并算法	200000	2.209735	1.625498	2.904653

4.3.2 内存碎片量的计算

无论是在多核系统还是单核系统中，内存碎片都分为内部碎片和外部碎片。本文引用文献[17]中关于内部碎片的定义的公式(4-1)。

$$Fi = \frac{\sum Ri}{\sum Qi} \tag{4-1}$$

在公式(4-1)中，Ri 代表请求分配内存的大小，Qi 代表实际分配内存的大小，Fi 代表内部碎片的数量。

当系统请求分配一块或多块地址连续的内存块时，如果现有的内存空闲空间无法满足系统的请求，则会产生内存溢出，并产生外部碎片。本文采用衡量外部碎片的公式(4-2)。

$$Fe = \frac{MEM\_UNUSED}{T} \tag{4-2}$$

在公式(4-2)中，MEM_UNUSED 表示系统中未使用的内存，T 表示系统中总的内存资源，Fe 表示外部碎片的数量。

参考文献[18]、参考文献[19]和参考文献[20]中都有相关的内部碎片的推导过程，本文中将不做详细的讲述。当输入{2,8,10,15,25,30,35,40,50,70,100,200}的请求分布数时，预测的内部碎片为 0.381[17]。

测试结果如表 4.3 所示。

表 4.3　内碎片测试结果

算法的种类	内碎片均值 (%)	外碎片均值(%)	碎片总量(%)
原伙伴算法	25.6311	0.0214	25.6525
部分放宽延迟合并算法	16.3231	0.0201	16.3432

为了使测试得出的结论更具有真实性和完整性，更能满足实际的需求，下面本文采用一种常见的输入请求测试方法对两种算法所产生的碎片数量进行相应的测试，并在程序中，采用均匀的输入方式，在数值范围 1 到 512 之间，均值是 256 的页框中输入数据。纵坐标表示两种算法产生的碎片数量的百分比，仿真得到的图形如图 4.7 所示。

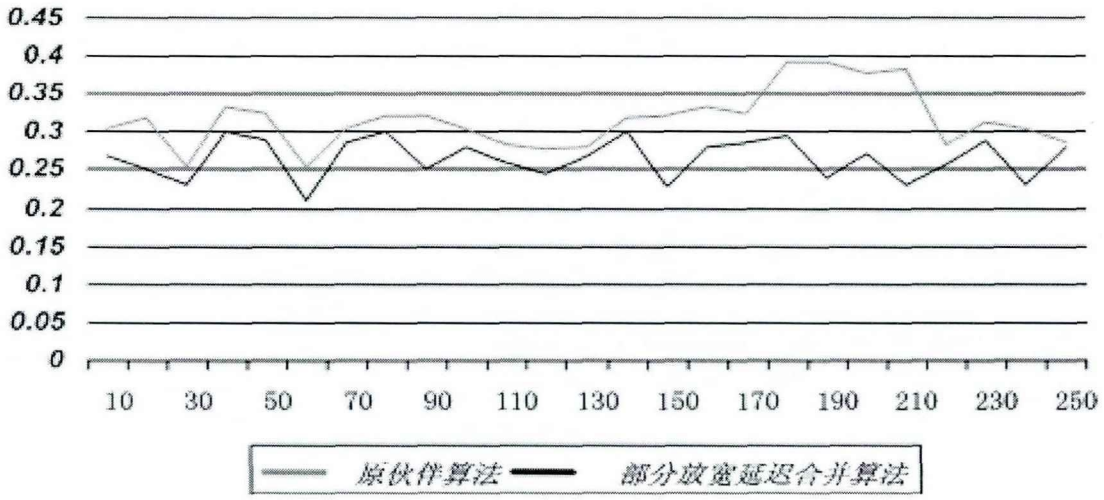


图 4.7　碎片量对比图

从图 4.7 可以看出，部分放宽延迟合并算法在做每次测试时内存产生的总碎片量低于原伙伴算法中产生的总碎片数量，无论是部分放宽延迟合并算法还是原伙伴算法，其内存碎片的产生量都随着均值的增大而逐渐上升。得出以上结论的原因是改进的算法是针对多核环境下提出的，并不是按照实际需要进行内存的分配，这就在一定程度上减少了内存碎片的数量。

4.4 本章小结

本章首先介绍了 Linux 操作系统中两个经典的内存管理算法——Buddy 算法和 SLAB 算法，并结合前人改进的策略在原有 Buddy 算法的基础上设计了一种新的改进算法——部分放宽延迟合并算法，并深入的分析了该算法的数据结构、内存分配和释放过程；接下来对 SLAB 算法的框架进行了改进，给出了改进后相关的代码，并深入的分析

了改进的算法分配和释放页框的过程。从理论上分析，改进后的算法不仅可以有效的提高内存的使用率，同时提高了整个存储系统的性能和多核系统的并行性。最后对改进后的 Buddy 算法进行了仿真测试，测试结果表明该算法具有更高的效率，对以后内存管理算法的研究具有一定的理论和应用价值。

结　论

针对多核系统内存管理算法的研究普遍存在内存利用率低和并行性差等问题，本文在现有的内存管理算法的基础上提出一种基于伙伴系统的部分放宽延迟合并算法，并在SLAB 算法的基础上进行了改进，提出一种改进的 SLAB 算法。本文的主要工作有以下4 个方面：

1．分析多核系统中 Linux 系统的经典内存管理算法，并对现有的内存管理算法存在的问题进行了阐述，结合经典内存管理算法的理论特点，对基于 Linux 系统的内存管理算法进行了深入的研究。

2．提出一种基于伙伴系统的部分放宽延迟合并算法。该算法改进了原算法中关于伙伴块的定义，改进算法的伙伴块要求两个块大小相等、地址连续，这就从一定程度上减少了多核系统中内存碎片的产生量，提高了内存的利用率。为了更好的适应多核环境，提高内存的访问时间，该算法引入了延迟合并算法的分块结构，减少了内存合并和分开时对多核的使用时间，从而提高了内存的访问速度。

3．提出一种改进的 SLAB 算法。该算法引入了本地 SLAB 队列，并且每个核或 CPU 都拥有自己的本地 SLAB 队列和半满 SLAB 队列，从而减少了核间自旋锁的使用。当核访问内存时，首先检查自己的本地 SLAB 队列，如果满足则直接分配，如不满足，则查找自己的半满队列，若半满队列满足则直接分配内存，不满足则调用伙伴系统创建新的SLAB 队列，分配内存。通过这种方式减少了核与核之间对共享内存的争用，提高了访问内存的速度，同时整体上提高了多核系统的并行性。

4．为了验证提出算法的可行性，本文通过对新的算法进行实验测试，从内存碎片和访问时间两个方面分析了改进算法与原算法在内存管理上的优势。

尽管本文提出的内存管理算法具有很好的性能，但是也存在着一定的不足，主要表现为部分放宽延迟合并算法在进行内存分配时，如果在原伙伴块的列表中查找不到满足请求的空闲页面，则会增加访问内存的时间。这种不足还希望在后续的研究工作中进一步加以改进。

参考文献

[1] 刘必慰，陈书明，汪东．先进微处理器体系结构及其发展趋势．计算机应用研究．2007，24(3):16-26 页

[2] 谢向辉，胡苏太，李宏亮．多核处理器及其对系统结构设计的影响．高性能计算技术．2007，11(4):1-6 页

[3] 郝松，都志辉，王曼等．多核处理器降低功耗技术综述．计算机科学．2007，34(11):259-263 页

[4] 张骏，樊晓桠，刘松鹤．多核多线程处理器的低功耗设计技术研究．计算机科学．2007，134(10):301-305 页

[5] 章承科．基于多核处理器实时操作系统的扩展．成都电子科技大学硕士学位论文．2006:23-24 页

[6] 覃中．基于多核系统的线程调度．成都电子科技大学硕士学位论文．2009:17-20 页

[7] 多核系列教材编写组．多核程序设计．北京：清华大学出版社．2007:30-50 页

[8] 赵炯．Linux 内核完全剖析．基于 0.12 内核．北京：机械工业出版社，2008:70-100 页

[9] 郭玉东．Linux 操作系统结构分析．西安:西安电子科技大学出社，2002:60 -90 页

[10] 刘近光，梁满贵．多核多线程处理器的发展及其软件系统架构．微处理机．2007，(1):1-3 页

[11] 何进仙．基于多核系统的内存管理研究．成都电子科技大学硕士学位论文．2009:37-61 页

[12] 刘俊海．多核系统内存管理算法的设计与实现．天津大学硕士学位论文．2009:15-26 页

[13] 李江雄．嵌入式 Linux 内存管理设计与实现．华中科技大学硕士学位论文．2009:26-42 页

[14] 谢子光．多核处理器间通信技术研究．成都电子科技大学硕士学位论文．2009:46-52 页

[15] 焦莉娟．浅析伙伴系统的分配与回收．科技情报开发经济．2005，15(15):15 页

[16] 冯国富，董小社．面向 Cell 宽带引擎架构的异构多核访存技术．西安交通大学学报．2009，43(2):1-5 页

[17] James Reinders. Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Pamllelism, Sebastopol California USA, OReilly Media, hc. 2007, 12(7):120-130P

[18] McKee, S. A. Dynamic Access Ordering: Bounds on Memory Bandwidth.University of Virginia, Technical Report CS-94-14. 1994, 4(3):3-10P

[19] McKee S. A. Aylor J H, Salinas M H, et al. Dynamic Access Ordering for Streamed Computations. IEEE Transactions On Computers. 2000, 49(11):1255-1271P

[20] Mckee S A, Wulf W A. Access ordering and memory-conscious cache utilization. In Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture(HPCA'95). 1995:253-262P

[21] Hong S I, McKee S A, Salinas M H, et al. Access Order and Effective Bandwidth for Streams on a Direct Ram bus Memory. In Proceedings of the 5th International Symposium on High Performance Computer Architecture(HPCA'99). 1999:80-89P

[22] David Tawei Wang. Modern DRAM Memory Systems: Performance Analysis and Scheduling Algorithm. PhD Thesis. University of Maryland. 2005:10-11P

[23] Panda P R, Catthoor F, Dutt N D, et al. Data and memory optimization techniques for embedded systems . ACM Transactions on Design Automation of Electronic Systems. 2001, 6(2):149-206P

[24] Jan Christian Meyer, Anne C.Elster. Latency Impact On Spin-lock Algorithm For Modem Shared Memory Multiprocessors . Scalable Computing : Practice and Experience. 2006, 5(11):3-8P

[25] Maged M. Michael. Michael L. Scott. Non-Blocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. 1997, 6(6):222-280P

[26] Michael Creel William L. Goffe. Multi-core CPUs, Clusters, and Grid Computing: a Tutorial. 2007, 5(11):160-200P

[27] Sevin Fide. Architectural Optimizations in Multi-Core Processors, VDM Verlag Dr. Muellere. K. Publisher, VDM Verlag Publisher. 2008, 18(11):14-17P

[28] Cory Isaacson. Software Pipelines and SOA: Releasing the Power of Multi-Core Processing. London England UK, Addison-Wesley Professiona. 2009, 2(1):88-90P

[29] Shuwei Sun, Dong Wang and Shuming Chen. A Highly Efficient Parallel Algorithm for H. 264 Encoder Based on Macro-block region Partition. In Proceedings of the 2007

International Conference on High Performance Computing and Communications (HPCC'07), Houston, USA, September 26-28, 2007:577-585P

[30] Sundamrajan Sriram, Shuvra S. Bhattacharyyn, Embedded Multiprocessors: Scheduling and Synchronization, Second Edition, Cleveland USA, CRC Press. 2009, 3(2):189-190P

[31] Kunle Olukotun, Lance Hammond, James Laudon. Chip Multiprocessor ArchRocture: Techniques to Improve Throughput and Latenccy. Mob, an and Claypool Publishers. 2009:1-7P

[32] Winthir Brunnbauer. Methodology for System Partitioning of Chip-Level Multiprocessor Systems. Dudweiler Germany, VDM Verlag Dr.Mueller e. IC Publisher: 2008, 15(4):15P

[33] Shameem Akhter, Jason Roberts. Multi-Core Programming: Increasing Performance Through Software Multithreading. Santa Clam USA, Intel Press. 2006:215-222P

[34] Ioannis E. Venetis, Thedore S. Papatheodorou, Typing Memory Manangment to Parallel Progamming Models. Springer Berlin/Heidelberg. 2006, 4(11):156-210P

[35] Clay Breshears. The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications, Sebastopol California USA, O'Reilly Media, Inc. 2009:51-56P

[36] Martin J David H. Linux memory management on later machines. proceedings of the Linux Symposium. 2003:55-68P

[37] http://www.Eefocus.com/article/09-06/74976s.html

[38] M. Tim Jones. Anatomy of the Linux slab allocator. 2007(5):111-115P

[39] Rashmi Bajaj, Dharma P. Agrawal. Improving Scheduling of Tasks in a Heterogeneous Environment. IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS. 2004, 15(2):107-117P

[40] 许建卫，杨伟，潘晓雷等. 体系结构模拟器的技术和发展. 信息技术快报. 2008, 6(1):2-13 页

[41] MENDEL ROSENBLUM, EDOUARD Bugnion, Scott Devine, and STEPHEN A. HERROD. Using the SimOS Machine Simulator to Study Complex Computer Systems. 2005, 15(3):99-110P

[42] Rosenblum M, Herrod SA, Witchel E, Gupta A. Complete computer system simulation: The SimOS approach, IEEE Parallel and Distributed Technology: Systems and

Applications. 1995，3(4):34-43P

攻读硕士学位期间发表的论文和取得的科研成果

[1]李静梅，杨新波．改进的 Gabor 算法在指纹识别系统中的应用．网络安全技术与应用．2010，(4)7:9-11 页

致　　谢

经过近一年时间的努力，论文终于完成了，两年半的研究生生活也即将画上句号。

在此论文完成之际，我怀着激动和高兴的心情，对帮助过我的老师和同学表示衷心的感谢。

首先感谢我的导师李静梅教授，李老师的言传身教让我的学术水平得到了提高。在论文撰写期间，李老师多次帮我审阅论文，指出论文中存在的问题，我才得以完成论文的撰写。同样，她严谨的治学态度，渊博的知识都是我学习和努力的方向，她的人格魅力深深的感染了我，在她的影响下我懂得了“要成才先成人”的道理。她对我学习上的指导和深深的教诲令我难忘，也将使我受用一生。

同时感谢实验室的同门师弟师妹们，实验室是一个大家庭，大家在一起朝夕相处，学习的日子很融洽，放松的时候很开心，与你们在一起我感受到了家的温暖。这样温馨的环境使我能够安心的学习。

感谢同寝室的同学及我的朋友们，研究生的生活有你们相伴才更多彩。

最后感谢我深爱的家人，是你们从精神和物质上的支持才有我的今天。

感谢在百忙中评阅我论文的老师们，你们提出宝贵的意见，才使我的论文得以完善。

感谢所有关心我、爱护我、支持我的人。愿幸福永远陪伴你们。