分　类　号：TP399　　单位代码：10183
研究生学号：2010532185　　密　　级：公 开

吉 林 大 学

硕士学位论文

实时系统内存管理方案的设计与实现

Design and implementation of real-time system memory management scheme

作者姓名：高菲菲
专　　业：计算机应用技术
研究方向：计算智能
指导教师：于哲舟　教授
培养单位：计算机科学与技术学院

2013 年 4 月

实时系统内存管理方案的设计与实现

Design and implementation of real-time system memory management scheme

作者姓名：高菲菲

专业名称：计算机应用技术

指导教师：于哲舟

学位类别：工学硕士

答辩日期：2013 年 5 月 21 日

本人郑重声明：所呈交学位论文，是本人在指导教师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。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。

学位论文作者签名：高菲菲

日期：2013 年 5 月 21 日

提　　要

紧凑内存管理系统是一个紧密压缩的实时内存管理系统，可以完成分配、释放、以及对内存对象的引用。

实现方案：其一为可移动内存版本；其二为不可移动内存版本。在可移动内存的实现方案中，页面分配操作时间复杂度为常数级，如果没有内存紧缩操作，页面回收的时间复杂度会由线性级降为常数级。在不可移动内存实现方案中，分配操作和释放操作的时间复杂度均为线性级。间接引用的时间复杂度为常数级。可移动版本为一次间接跳转，不可移动版本为二次间接跳转。

指针概念：在紧凑内存模型中指针是指一个地址与一个偏移量的集合。因此，紧凑算法模型支持基于偏移量的寻址算法而不是基于地址指针。值得注意的是如果内存对象被分配在单一的，物理地址连续的一块内存中，可移动版本的实现方案也是可以支持基于地址的寻址算法的。在紧凑算法中紧凑操作是受限制的，它只会释放一块内存然后引起一块相同大小的内存对象的移动。

研究内容：1. 实时内存管理系统；　2.显式动态内存管理系统；　3. 隐式动态内存管理系统；　4.地址空间的意义，页块和页面碎块研究；5.各种版本的内存开销和算法实现。

预期达到的目标：

1、减少内存额外开销，我们可以通过将物理空间中的页面同样使用于抽象地址空间的方法来实现动态的抽象地址空间。

2、在当前版本中大于 16KB 的内存对象是不被支持的。我们希望在进一步的工作中通过使用 arraylets 的概念来实现大于 16KB 内存对象的支持。

摘 要

嵌入式实时操作系统以其良好的可靠性和卓越的实时性被广泛地应用在通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中，如卫星通讯、军事演习、弹道制导、飞机导航等。在美国的系列主力战机、爱国者导弹和火星探测器上都使用到了嵌入式实时操作系统。因技术垄断及其在航空、航天和军事领域的特殊作用，嵌入式实时操作系统过去一直对中国区禁运，自解禁以来在我国军事、通信、工业控制等领域得到了非常广泛的应用。但在这些涉及国家安全的核心领域受国外技术制约是十分危险的，如果我们能够在嵌入式实时操作系统方面取得一些突破性进展，我们便会拥有领先的，完全自主的嵌入式操作系统，以打破欧美对我国的技术封锁。因此无论从学术研究还是从实际的工程应用来讲，如今对嵌入式实时操作系统的研究都具有关系到国家利益的重大的意义。

内存管理系统作为实时性操作系统中至关重要的环节，其性能高低对于整个嵌入式系统性能起到了至关重要的作用。因此设计一种高效、可靠的内存管理方案显得尤为重要。本文中介绍了一种适用于实时操作系统的内存管理算法，通过它进行内存的分配，释放，以及间接引用内存对象的整个过程中内存都保持紧缩状态，因此我们称它内存紧凑分配算法。紧凑分配算法有两种实现方式：可移动版本和不可移动版本。在可移动版本的实现方案中内存分配操作的时间复杂度为常数时间，释放操作如果不需要紧缩其时间复杂度由线性时间变为常数时间。在不可移动版本中分配和释放操作的时间复杂度为线性时间。两种实现方案中间接引用都需要耗费常数时间。此外，在碎片角度来看，此系统提供了完全可预测的内存使用情况。一言以蔽之，它是一个优秀的实时内存管理系统方案。通过将现存的内存分配策略与本文提出的两种实现方案的紧凑分配算法相比较，实验确认了我们的理论可行性，并且证明了系统优异的性能表现。最后，我们引入了局部紧凑策略以实现性能和碎片处理的平衡。

关键词：内存管理 实时系统 页面链表

Design and implementation of real-time system memory management scheme

Embedded real-time operating system for its reliability and excellent real-time is widely used in the field of communications, military, aviation, aerospace and other sophisticated technology, and real-time requirements, such as satellite communications, military exercises, trajectory guidance, aircraft navigation. In the series main fighter plane in the United States, Patriot missiles and Mars probe to the embedded real-time operating system. Due to a technical monopoly and its special role in the field of aviation, aerospace and military, embedded real-time operating system has been embargo on China, a very wide range of applications, since the lifting of the ban in the military, communications, industrial control and other fields. However, in these core areas related to national security by foreign technical constraints is very dangerous, if we can made some breakthroughs in embedded real-time operating system, we will have a leading, completely independent of the embedded operating system, to break the blockade of Europe and the United States on China. Therefore, both from academic research or practical engineering applications in terms of today's embedded real-time operating system has great significance related to the national interest.

The memory management system as a vital link in the real-time operating system, played a crucial role in its performance level for the entire embedded system performance. Therefore the design of an efficient, reliable memory management scheme is particularly important. One for the real-time operating system memory management algorithms, through its memory allocation and release, as well as an indirect reference to the whole process of the memory object memory remain tight, so we call it compact memory allocation algorithm is described in this article. Compact allocation algorithm implemented in two ways: a mobile version and a non-mobile version. The time complexity of the implementation of the mobile version of the memory allocation operation is constant time, the release operation if you do not need to crunch time complexity of linear time to constant time. Allocation and deallocation operations in a non-removable version time complexity is linear. An indirect reference to the two implementation takes constant time. In addition, the debris

point of view, this system provides a completely predictable memory usage. In a nutshell, it is an excellent real-time memory management system. In the compact allocation algorithm compared to the two implementations of the program through the existing memory allocation strategies in this article, the experiments confirm the feasibility of our theory, and proved the excellent performance of the system. Finally, we introduced a locally compact strategy in order to achieve a balance of performance and debris processing.

Keywords: memory management real time system pages list

目　录

第一章 绪论

1.1 课题研究背景

自从上世纪七十年代以来，实时的计算机系统被更多的被开发使用，鉴于其拥有实时的特性，嵌入式实时系统被广泛利用在与人类密切相关的生活工作之中，比较典型的使用方向如：航天、飞行设备的控制系统、汽车中的电子控制设备、工业系统中的过程管理系统、医院病房中的监护系统、通信系统等非常重要的场景之中。嵌入式实时系统的运算结果的正确性非常的依赖于其时间上的敏感性，也就是嵌入式实时系统必须在系统所规定的时间范围内能够非常快速的响应请求并返回结果。

在目前科技领域中，嵌入式实时系统经常被用作要求很高的大型系统的核心系统模块，如果其不能性能稳定的满足整体系统所要求的快速的响应时间及精确的运算结果，便有可能引发重大的事故，造成大量的财产损失，甚至造成人身伤害、更甚至造成灾难性的后果。在人类的历史中也发生了很多这样的血淋淋的教训，如：1962 年水手 I 号发射失败，1978 年三里岛核电站的核泄漏事故，1990 年美国全国的长途电话业务瘫痪近 9 小时，1998 年火星探路者在着陆勘探 250 天后与美国航天局失去联系等。由以上的沉重的教训告诉我们，切实的保证嵌入式实时系统的安全性，可靠稳定性，容错性，是当前嵌入式实时系统中具有十分重大意义的研究领域。

根据实时系统的时间特性，有很多种版本的关于实时系统的定义。其中牛津计算技术字典中的定义如下[1][22]：

实时系统指系统输出的产生时间具有决定性意义的系统。这是因为系统的输入来自于物理世界的某个动作，而输出时对该动作的响应。输出和输出之间的延迟必须小于某个确定的时间间隔。

通过上面的定义我们可以得知实时系统并不是一个自我封闭的系统。其会接受来自于外部环境的输入，并经过自己的处理之后产生一个对应的输出以控制外部环境。在这个层面来讲，实时系统本身会处于一个更大的应用系统中用于控制整个应用系统的运行，因此我们又称之为嵌入式系统。

嵌入式系统出现后的很长的一段时间中，系统的设计者都是在平面和无保护的内存体系结构中进行开发的。操作系统没有内存管理单元（MMU），因此也不会支持虚拟内存机制，操作系统和应用程序可以访问系统中存在的所有的内存空间，进程与进程间的数据传递工作都是通过指针来完成的。

随着硬件技术的发展，处理器继续迅速发展，目前的计算机已经步入了一个新的时代，我们称之为”信息时代”。嵌入式设备被广泛应用至各种各样的机械设备之中。大部分的嵌入式应用都具有一定的时间限制及可靠性、正确性要求，也就是要能准确的在规定的时间内完成任务。在这样的前提下对系统中的软硬件资源进行管理，这样的系统可以被称之为嵌入式实时操作系统。相比传统类型的操作系统，嵌入式实时操作系统所需要的资源更加有限，操作系统系统的结构设计更加简单紧致，系统的处理时间更加快捷，系统的安全性更加可靠。这些实时操作系统不同于一般通用操作系统的特点要求其计算结果需要更加的精确，产生正确结果的时间更加的符合要求。

目前在各个领域中存在的超过百种各式各样的嵌入式实时操作系统，这是我们发展民族软件的重要机会。尽我们最大努力在嵌入式操作系统方向取得一些突破性的进展，使我们拥有领先的，自主的嵌入式实时操作系统，便可以彻底打破发达国家对我们的技术封锁。因此在学术研究以及实际的工程应用两个方面来讲，如今对嵌入式实时系统的研究具有关系到国家利益的重大的意义。

1.2 课题研究内容及意义

本文介绍了一种全新的针对嵌入式实时系统的内存管理方案，我们称之为紧凑内存管理系统。它是一个紧密压缩的实时内存管理系统，可以完成分配、释放、以及对内存对象的引用。文章提出了两种实现方案，其一为可移动内存版本，其二为不可移动内存版本。在本内存管理系统中内存碎片受编译时常数控制，当启用内存紧缩操作时，碎片量会进一步减少。紧凑内存管理系统使用虚拟地址空间实现内存对象的管理，因此由于内存紧缩导致的内存关联更新更加有限，我们只需要更新间接指针便可以了。

在可移动内存的实现方案中，页面分配操作时间复杂度为常数级，如果没有内存紧缩操作，页面回收的时间复杂度会由线性级降为常数级。在不可移动内存

实现方案中，分配操作和释放操作的时间复杂度均为线性级。间接引用的时间复杂度为常数级。可移动版本为一次间接跳转，不可移动版本为二次间接跳转。

进一步我们将会引入个新的指针概念：在紧凑内存模型中指针是指一个地址与一个偏移量的集合。因此，紧凑算法模型支持基于偏移量的寻址算法而不是基于地址指针。值得注意的是如果内存对象被分配在单一的，物理地址连续的一块内存中，可移动版本的实现方案也是可以支持基于地址的寻址算法的。在紧凑算法中紧凑操作是受限制的，它只会释放一块内存然后引起一块相同大小的内存对象的移动。

在本模型中，内存被划分为 16KB 大小的页面的集合。每个页面都被称为尺寸类的一个实例，页面会进一步被划分为大小相同的内存块。关于页面和尺寸类的概念改编于[3]中。当有内存块请求操作时，系统会分配一个最小能满足需求的页面块。大于 16KB 的内存块的请求虽然不能被满足，但是我们提供里一种解决大块内存分配问题的方法。

紧凑内存管理系统的核心问题是如何在整个生命周期中保持内存尺寸类处于一种紧凑的状态。换句话说也就是尽量保持所有的尺寸集合中的至多只有一个页面未被利用。当内存对象的释放操作导致同一个尺寸类中出现了两个未满的页面时，系统便会将一个页面中的内容移动到另一个页面中以保持不会出现两个同时未被完全利用的页面。如果一个未满的页变为空闲页面，它便可以被任何一个尺寸类重新使用。空闲页面会被放入空闲页面池中，通过一个智能的空闲链表，新的内存请求可以在常数时间内被满足。

在紧凑模型的移动版本的实现中，页面块直接与物理上连续的内存块映射，因此内存紧凑时需要移动所有的内存对象。这种实现导致分配内存是时间复杂度为常数级，但是释放操作如果需要内存紧凑的话时间复杂度变为线性级别。

在不移动版本的实现中，我们使用了一个内存块表（虚拟内存）来将页面块映射至同等大小的物理块帧，从而使物理块帧可以定位到内存中的任何地方。在这种情形下，仅需要修改内存块表便可以实现内存的紧缩过程,而不需要真正的移动内存对象，从而使得紧凑过程更加快捷。但是，尽管紧凑过程速度加快了，内存的释放操作由于对内存块表的管理仍然需要消耗线性时间级。同样，内存分配操作的时间复杂度也变为线性级。

在两种实现方案中，我们都可以放松对保持内存紧凑的要求，允许在单个尺寸类中出现不只一个的未满的页面块。这减少了释放操作所花费的时间，使得其时间复杂度变味常数级。我们所遵守、控制、实现的这种对于时间复杂度和内存碎片之间的权衡的概念称为局部紧凑。

最后，我们列出了实现于 Gumstix 裸机上轻量级 HAL 及 Linux 上本系统的两种实现方式，以及非紧凑嵌入式实时系统内存管理方案（Half-fit&TLSF），非实时内存管理方案（First-fit,Best-fit,Doug Lea’s allocator）在最坏及平均负荷下的基准测试结果。

本文提出了紧凑内存管理系统的模型，提供了可预测的内存概念（可预测的碎片）。基于此模型我们提出了两种实现方案。进一步提出了局部紧凑策略并在移动版本中实现了它。此外，我们还进行了大量的试验对本系统的三个版本及一些显示动态内存管理算法如 First-fit,Best-fit,Doug Lea’s allocator,Half-Fit 和 TLSF 进行了比较。实验室用了两个不同的平台作为基准：Gumstix 平台运行轻量级的硬件抽象层来衡量运行时间（处理器周期），Linux 操作系统来进行处理器指令测试。最后我们测量了对紧凑内存管理的移动版本与 TLSF 进行了碎片实验。得出了有用实验数据。

1.3 论文组织结构

论文以对嵌入式实时系统的内存管理系统的探讨为开始，引出了对于紧凑内存管理模型的描述，随后又提出了模型的两种实现方案。最后我们展示了本系统及基准程序的试验结果。

第一章， 绪论：本章节列出了论文的提纲及论文所做的主要工作及目的意义。

第二章， 嵌入式实时系统及动态内存管理系统：本章总结了内存管理概要并描述了嵌入式实时系统对于性能的需求，描述了内存碎片问题。文章还对各种内存管理方案进行了广泛研究。它们包括：非实时系统中的内存管理方案如 First-Fit，Best-Fit，Doug Lea’s 分配器。实时系统中的非紧凑内存管理系统如 Half-Fit，Two-level segregated fit。最后文章讨论了带垃圾回收机制的内存管理系统如 Treadmill，Metronome 以及 Jamaica。

第三章， 紧凑内存管理系统设计：本章详细描述了紧凑内存管理系统的系统模型。介绍了抽象地址空间及物理地址空间原型，讲解了尺寸类的概念。更进一步的描述了尺寸类对于内存碎片的限制作用。

第四章， 紧凑内存管理系统的实现：本章介绍了内存管理系统的实现方法。文章详细介绍了两种系统的实现方案，即移动版本与非移动版本实现，两种实现方案都经过了严格的检测，文章介绍了它们的复杂度及日常内存开销。此外，本章还引出了一种能够平衡性能与碎片的这种方案，称之为局部紧凑算法。最后我们讨论了本系统的可扩展性及进一步的优化方法。

第五章， 实验验证：本章讲述了实验的设计及基准测试程序。我们利用三个不同的函数模拟了最坏的工作负荷及平均负荷。两种紧凑内存管理系统的性能通过时钟周期及指令运行度量。实验的结果与 First-Fit,Best-Fit,Doug Lea’s allocator,Half-Fit,TLSF 等进行了详细的比较。在本章最后进行了内存碎片的测试，并将移动版本的紧凑算法与 TLSF 算法进行了比较。

结论与展望：本章我们总结出了本文的揭露，对本文进行了系统的回顾并列出了它的主要贡献。最后我们对未来要做的工作进行了讨论。

第二章 实时内存管理系统

本章首先对内存管理系统进行了概要描述,然后详细说明了实时性系统对性能方面的一些需求。作为内存管理中的一个很重要的问题，本章节也对内存碎片进行了积极的探讨。另外，本章讲述了动态内存管理系统的概念，并随之介绍了现存的内存管理策略，如：First-Fit, Best-Fit, Doug Lea’s allocator, Half-Fit 以及 Segregated fit。最后对采用垃圾回收机制的内存管理方案 Treadmill，Metronome，Jamaica 进行了简单介绍。

2.1 内存管理概述

内存管理在此处指的是动态内存管理.动态内存管理是现代操作系统中被深入研究的基础性部分,这个核心单元对内存各个部分的使用情况保持实时的追踪。应用程序使用动态内存管理系统以任意的顺序来分配和释放任意尺寸的内存，这便是称之为动态的原因。间接引用是指应用程序访问已经分配好的内存对象。在 C 语言中，内存管理系统并不支持间接引用操作。应用程序可以直接访问整个连续的内存空间。对于像 Java 虚拟机这样的虚拟机而言，它们调用显式的间接引用方法来获取对内存的访问权限。内存管理系统通过提供一个可用的内存块来相应请求操作，通过释放已占用的内存块来响应释放操作，通过提供一个已分配内存对象的内存地址来响应一个间接引用操作。内存释放操作会导致内存中存在空洞，如果这些空洞太小，就无法被未来的分配请求重新利用。动态内存管理系统必须将这个问题最优化，我们称之为内存碎片问题。分配操纵的复杂度即寻找空闲内存的困难程度随着碎片的增多越来越难，释放操作的复杂度也与碎片问题相关。因此，碎片问题是内存管理中的一个关键问题。在此，我们用术语碎片来表示内存空间的碎片化现象及被碎片化的内存的大小。

内存碎片可以分为以下两种形式：内部碎片和外部碎片。

内部碎片：由于内存分配策略的影响，系统分配的内存往往会大于实际需要的内存大小，比如说：系统内存只能以 2，4，8，16，32 等块为单位分配给应用程序，如果一个程序只请求 23 个字节，至少会有 24 字节分配给它，这就导致了 1 字节的浪费，在这种情况下无法使用的内存包含在被分配的内存块之中，因此

称之为内部碎片。

外部碎片：外部碎片的产生是由于空闲内存被分散成小块并被分配器散置，这是内存分配算法不能高效的整理程序使用的内存的的缺陷，这些内存块被分隔为太小的块，不能满足程序的需求，从而导致了它们的实际不可用性。“外部”的意思是指这些内存块在系统分配的区间之外。比如说应用程序请求了 3 块连续内存然后释放掉中间块，内存分配算法可以重新使用这块内存，但是当需要分配的内存块大小大于这块空闲内存时，它实际上是不可用的。

Johnstone 研究发现，大部分的应用程序趋向于分配大量的相等大小的小块内存。他认为如果为一个应用程序采取了合适的策略来分配内存，也许我们便可以忽略内存碎片所造成的性能上影响。对于短时间运行的应用程序，这种结论可能是正确的，但是对于安全的临界系统而言，这种结论是无法被信服的。特别是对于硬实时系统而言，最坏的运行状态及内存碎片必须被考虑。

应对碎片的一种方法便是紧凑算法，或者称为反碎片算法。初始的空闲内存是连续的。碎片导致了非连续的空闲内存空间。紧凑算法通过重新排列被使用的内存空间从而使得大块的内存变为可利用的。最好的情况便是所有的空闲内存经过整理后再次变为连续的内存空间。

动态内存管理是指从堆 Heap 中直接分配内存，然后将内存直接释放至堆中，现在主要有两种实现动态内存管理的方案：一是显示内存管理方案（Explicit Memory Management）：在显式内存管理方案中，内存从堆中进行分配，用完后手动释放至堆中。程序使用 malloc()函数分配整块的内存，然后使用 free()函数释放被分配的内存。自动内存管理（Automatic Memory Management）：也可称为垃圾回收器（Garbage Collection）。与显式内存管理不同，运行时系统关注已分配的内存空间，一旦内存空间不再被使用，将会立即被回收。

显式动态内存管理系统与自动内存管理系统相比，它实现了更多的底层的函数功能。因此在某种程度上，他们是不可比较的，但是 Berger et. Al 设计了一种显式动态内存管理与自动内存管理之间性能相比较的方法。

在此文中，我们更侧重于提出一个能够即拥有底层实现，又拥有高层实现的显式内存管理系统。

2.2 实时内存管理系统需求

传统动态内存管理策略是典型是非确定性的，其中大多数只在最好及平均情况下提供了良好的反应时间，但是在最坏情况下的响应时间确是不可衡量的。这中情况适合与非实时系统，但对于硬实时系统而言，这是不可原谅的。因此动态内存分配器已经在实时系统中被排除掉了。对于很多实时控制器而言，静态分配内存很好的满足了它们的要求。现代的嵌入式实时系统逐渐增加的复杂性对于其内存分配策略提出了新的要求，更高的灵活性。由此我们需要设计针对于嵌入式实时系统的内存管理方案。

在一个理想的动态内存管理系统中，每个操作单元（内存分配、释放、间接引用）的时间复杂度为常数级别。我们通常将此常数时间作为内存管理系统中各个操作的基准响应时间。如果我们无法达到常数级别的时间复杂度，那么有界的线性时间也是可以接受的。需要注意的是，内存管理系统的响应时间应该受所请求的内存块的大小限制，而非整个系统的内存状况。

如果系统的响应时间是有限的，则我们称其为可预测的。碎片问题会影响到系统响应时间的可预测性。让我们考虑下面这种情况。系统内存由 n 块大小相等的内存块组成，一个应用程序请求了所有的 n 块内存，然后其释放偶数编号的块。因此最终会有 50%的内存块被释放。但是，之后的所有超过两个连续块的请求都会被拒绝。在此种情况下，内存管理系统解决碎片问题的方式便会影响系统的响应时间。例如系统如果移动内存对象以便形成连续的内存空间来满足连续内存空间的请求，那么系统分配操作的响应时间便成了不可预测的，它可能会依赖于内存的整体情况。

可预测的可用内存意味着还有多少指定尺寸的内存可以分配成功依赖于实际分配的块的数量以及它们的大小，而与分配和释放操作的历史无关。可预测系统中的碎片量也是可预测的，仅仅依赖于实际分配的量。除了可预测性之外，系统中的碎片量必须被最小化以最大化可用内存的数量。

迄今为止，由于内存碎片的产生依赖于系统的分配及释放过程，没有任何现存的显式动态内存管理系统满足了对于空闲内存的可预测性。

综上所述，最小化碎片量的一种方法便是执行内存的紧凑过程。但是我们必须将内存紧凑过程的工作量平衡并且公正的分散开，以获得可预测的响应时间。

紧凑操作可以通过事件或者时间触发来完成。

- “在一个事件触发的系统中，进程的处理活动为特定意义的事件的发生所导致的进过。”[2]即内存的紧凑操作依赖于特殊意义的事件的发生。例如：通过内存管理系统接口之间的调用，内存块 M 可以被移动至另外一个地址。
- “在一个时间触发的系统之中，特定的动作在一个实时系统中特定的时间点被定期的触发[2]”即紧凑操作每隔 n 次时钟周期触发一次，内存块 M 可以移动至另外一个位置而不依赖于内存管理系统的接口调用。

  我们理想中的内存管理系统应该拥有常数级的反应时间，可预测的可用内存，最小化的碎片量。其中最小化的碎片量我们通过内存的紧凑操作达成目标。内存紧凑使用了事件触发机制。

2.3 显式动态内存管理算法

显式动态内存管理系统中,内存的分配(malloc)和释放(free)操作必须显式的被调用。Wilson et al.在[3]中为我们总结出了动态内存分配策略。Masmano et al.[4]和 Puaut[5]展示了动态内存管理系统在实时负载下的工作表现.在本节中,我们简单介绍以下现存的内存分配机制:First-Fit,Best-fit，Doug Lea’s allocator,Half-fit,以及 Two-level segregated fit.特别注意以上分配策略在一块连续内存上的分配机制.

以上所提及的显式动态内存管理系统并未直接的处理有关内存碎片的问题,这意味着它们并未执行内存紧凑操作.算法试图在一块连续的内存中以最优的方式来对齐以分配的内存对象.剩余的可利用内存依赖于应用程序分配和释放操作的过程.这便导致了可用内存的不可预估.对于安全攸关的硬实时系统而言,无法被预估的碎片量是不可接受的.

2.3.1 顺序适应算法

首次适应算法(First-fit),最优适应算法(Best-fit)以及最坏适应算法(worst-fit)皆为顺序适应算法(Sequential-fit).[6,7]对于此两种算法进行了详细的描述。顺序适应分配算法基于空闲内存块的一个单链表或者双链表,链表的指针被植于空闲内存块中,因此没有内存的浪费存在.

首次适应算法分配内存的方法为从空闲内存结构的表头指针开始依次查找，并将第一块大于等于所请求空闲块的内存空间分配给用户。空闲内存表中节点的

排序既不按照初始地址有序，也不按照节点大小有序。当要释放内存空间时，只需要将此空闲块插入至链表的表头，从而降低了算法的时间复杂度。

最佳适应算法是将可利用空闲链表中一个空间大于等于于所请求空间大小且最接近所申请空间大小的空闲块的一部分分配给请求用户。内存的分配与回收均需要从头至尾遍历可使用空闲链表。为了避免每次请求分配操作都要遍历整个空闲链表，算法要求链表中节点需要按照从大到小进行排序，因此只需找到第一个大于等于所申请空间的空闲块即可以将其分配。但是在内存回收时，需要把回收的空闲块插入在符合大小顺序的链表位置。最佳适应算法的缺点为在内存分配时会因为剩余内存片段太小而无法再次利用，从而生成内存碎片，同时这种算法也保留了一些很大的内存空间以响应将来要发生的用户申请的较大的内存空间，从而使整个链表中节点大小差别越来越大。最佳适应分配算法适合请求分配的内存空间大小范围较广的系统，此算法的时间复杂度较高。

最差适应算法的思路为将可利用空闲链表中的一个大小等于“请求分配的空间”且是链表中最大的一块空闲内存的一部分分配给用户。该算法要求空闲节点按照从大到小的顺序排序。在内存分配时不需遍历链表，在回收时需要遍历，以便插入到适当的位置。最差适应算法将使链表中的空闲内存大小逐渐趋于均匀分布，适合请求分配内存空间的大小范围较固定较窄的系统，此算法时间复杂度最高。

从上述描述可以看出,这些算法的分配机制都不适合嵌入式实时系统.让我们考虑一个大小为 m 的块,如果其已分配的空间为最小块 s,剩余的块为空闲块,在这种情况下空闲链表中的内存数量达到最大值.在最坏的情况下,要满足一次内存请求需要遍历整个空闲链表.需要 m/2s 次循环。释放内存对象花费常数时间。

2.3.2 Doug Lea 分配算法

Doug Lea 分配器[8]是广泛应用在多个系统环境（例如多个 linux 版本）中的混合分配算法.它使用了两个不同类型的空闲链表：第一种类型包含了 48 个拥有确切大小的空闲链表(16bytes 到 64bytes),称其为快速分配器.剩余的分隔的列表为普通分配器.分配操作通过寻找与内存请求相匹配的空闲链表来完成.分配器使用了延迟聚合策略.这种策略的使用意味着当内存释放操作完成后相邻的空闲块

并不会立刻合并.而是当请求操作不能被满足时,系统会在全局范围内进行块的合并操作.因此内存块的释放操作非常迅速,会在常数时间内完成.但是分配操作却因为可能发生的全局范围内的块合并操作而导致不确定的运行时间.我们用 m 表示内存块的大小,s 表示最小的内存块大小,那么$O(m/s)$表示当内存请求操作没有被满足时发生的内存块合并操作的复杂度.因此 Doug Lea 分配算法也是不可预测的,并不适用于硬实时系统.

2.3.3 Half-fit 算法

Half-fit 算法总共维护了 $\log^{2N}$ 个链表(N 表示了可分配的内存的大小),在每个链表中存放的内存块的大小范围从 2^i 到 2^{i+1},在分配内存时,首先按照申请内存块的大小找到所对应的链表,然后在链表中寻找第一个合适的块(First-Fit),将合适的块分配给内存,然后将剩余的块加入到对应的链表之中.假如我们请求的内存块大小为 s,那么算法将在第 i 个链表中开始搜索查询,其中 $i=\lfloor \log_2(s-1) \rfloor + 1$(s=1 时,i=0),如果链表 i 中没有空闲内存块,则到 i+1 个链表中查询,一次类推.当一个大块的内存被分配时，其剩余内存加入对应链表。Masmano et. al[9]的研究表明了在 Half-fit 分配算法中内存碎片率很高,特别是当所请求内存均非 2 的幂次时,碎片化现象尤为严重.

2.3.4 TLSF 算法

TLSF(Two-Level segregated fit),即二级隔离拟合算法在 RTLinux/GPL 中通过将空闲链表及位图分配算法相结合来实现对内存的管理.算法采用二级数组链表来管理内存块,第一级链表按照 2 的幂次对内存进行分块,分级的数目称为 FLI(First Line Index).第二级数组将第一组中的每块内存进行进一步的线性划分,我们可以自定义需要分配的级数 SLI(Second Level Index).每一个链表都有一个相关联的位图来记录该链表中是否存在空闲内存块.处理器指令可以在常数时间内完成内存块的查找工作.如果在释放操作后有相邻的空闲块,它们便会立刻通过伙伴标记技术合并.每一个被使用的块都包括 8 字节的头部字段,其中前四个字节包含了该块的大小,后四个字节包含了指向前一内存块的物理指针,这些信息对于在

常数时间内将相邻空闲内存块合并有重大意义.由于发现相邻空闲块后立即合并策略的运用,使得 TLSF 算法与 Half-fit 算法相比拥有更少的内存碎片,由于 TLSF 算法中最小的内存块为 16 字节,在最坏情况下头部字段导致的内存浪费会很大(50%).

2.3.5 算法复杂度

下表 2.1 列出了现存的显式动态内存管理系统中,内存的分配和释放算法的时间复杂度问题,通过表中可以看出,只有 Half-fit 以及 TLSF 算法可以提供有界(常数)时间内的分配和释放操作.

表 2.1 算法时间复杂度

	分配操作	释放操作
首次适应(First-fit)	$O(m/2s)$	$O(1)$
最佳适应(Best-fit)	$O(m/2s)$	$O(1)$
道格拉斯算法（DLmalloc）	$O(m/s)$	$O(1)$
Half-fit	$O(1)$	$O(1)$
TLSF	$O(1)$	$O(1)$

2.4 隐式动态内存管理算法

隐式的动态内存管理系统负责收集已分配但是并未使用的内存对象.隐式动态内存的释放机制即为我们通常所说的垃圾回收机制.垃圾回收机制通过释放大量的已分配却未被使用的内存块来满足未来可能发生的任意大小的内存请求.在本节中我们不深入研究垃圾回收机制,我们只把重点放在此机制中关于嵌入式实时系统的垃圾回收机制.

下面我们检测了现存的实时系统中的垃圾回收机制:Treadmill 算法[10]及其变种[11,12]，时间触发节拍器算法[13]，事件触发牙买加算法[14]。其中后两者

均为商业系统，Ritzau[15]在他的论文中详细讲述了它们。

2.4.1 Treadmill 算法

Baker’s Treadmill[10]算法作为一个实时、非复制的垃圾回收器为内存分配操作提供了常数级别的响应时间.垃圾回收策略为四色回收机制的组合.关于这种方法的具体实现细节可以在[10]中获得。该算法使用单一的内存块大小，内存请求通过在空闲链表中取下一块来满足。所有的内存块都被存储在一个双向循环链表中，因此内存的分配操作是在常数时间内完成的。使用单一的块大小是非常受限的而且导致了系统中存在大量的内部碎片.此算法的最主要的缺点是其不可预测的大量的垃圾回收操作导致的系统的不稳定性.

Wilson[11]提出了一种按照 2 的幂次分割的不同大小的空闲链表。分配操作由适合所请求的空闲链表完成。每个链表都由一个 Treadmill 回收器负责回收操作。只有当一个链表中空闲块为空时回收操作才会发生。由于此种策略的不可靠性，其很难适应硬实时系统。

一种基于 Treadmill 回收器的页面级别的内存管理方案在[12]中被提出，它在不影响内存分配操作时间复杂度的基础上提高了内存利用率。一种页面重映射机制被用来创建大块的连续可用内存。

2.4.2 Metronome 算法

在 Metronome 算法中内存被分割为大小相等的页面.每个页面再次被分割为固定大小的块.如果有 n 种不同大小的块,便存在 n 种不同大小尺寸类.所以包含相同尺寸的块的页面组成一个尺寸类集合.当有内存请求时便会在最小的能满足请求的尺寸类中完成分配操作,所有的这一切在常数时间内完成.未被使用的页面可以形成各自的尺寸类集合.

如果一个尺寸类中的页由于垃圾回收器导致碎片化问题达到一定的紧急程度，内存紧凑算法便会被执行。首先尺寸类中的页面是按照页面中所包含的未使用的页块数量来排序的，算法设置两个指针。对象的重新移动操作利用了存在于每个内存对象头部的指针指针一指向链表中第一个未满的页面，指针二指向了链表中最后一个页面。内存的紧凑操作是指将指针二指向的页面中的已分配的对象

移动至指针一所指向的页面中，对象的重新移动操作利用了存在于每个内存对象头部的指针，当两个指针相遇时，紧凑算法结束。

由于 Metronome 算法是一个时间出发的实时垃圾回收器，紧凑操作是预先定义的垃圾回收循环的一部分。研究表明紧凑操作大约占用回收循环 6%的时间，因此紧凑操作在 Metronome 算法中的开销是可控的。剩余的时间主要被用于探测哪些已经分配的对象不再被使用。回收循环的间隔时间需要按照不同的应用程序区别对待，不妥当的回收循环周期会导致性能下降或者内存溢出错误的产生。

2.4.3 Jamaica 算法

Siebert 提出了不执行紧凑操作的 Jamaica 实时系统垃圾回收机制。此算法引入了一种新的基于固定尺寸的内存块对象模型。整个内存空间被划分为固定大小的相等的内存块。小块的内存请求可以使用单块满足。大块的内存请求可能会使用一个非连续的内存块的集合来满足，每个内存块都持有一个指向其后继的指针。非连续的内存块集合便是不需要内存紧凑操作的原因。一个内存对象可以由任意个内存块构成，内存块之间通过一个单链表或者树形结构组合。

对于使用固定大小内存块的内存管理系统而言，最重要的决定便是选择一个合适的块尺寸。Siebert 建议块尺寸大小应在 16 到 64 字节之间。这个参数的选择取决于特定的应用程序。

Jamaica 算法的分配和回收操作的时间复杂度取决于所请求的内存对象的大小及所使用的内存块的大小，如果内存对象的大小为 s，那么它需要$n = [\frac{s}{b}]$个内存块。这意味着如果我们请求分配或者释放一个大小为 s 的内存对象，就需要 n 次的链表操作。因此分配的释放算法的时间复杂度接近于线性时间$O(\frac{s}{b})$,取决于对象的大小。

Jamaica 内存对象模型中，我们无法在常数时间内完成内存的间接引用操作。由于内存对象可能由非连续的内存块通过链表连接而成，访问内存对象的最后一块内存需要遍历对象的整个链表，因此内存间接引用操作为线性时间复杂度，并且依赖于所引用的块在对象中的位置。

2.5 本章小结

在显示动态内存管理系统中两种现存算法提供了常数级时间复杂度的分配和释放操作，分别为：Half-fit 和 TLSF。又由于 TLSF 的碎片处理性能要优于 Half-fit，它成为最适宜于嵌入式实时系统的候选内存管理系统。现在需要考虑的主要问题是，所有的这些内存管理系统都没有提供可靠的碎片处理能力。这意味着在某种情况下可能会导致很高的内存碎片率。因此，实时操作系统由于内存必须是可预测的，它使用上述内存管理系统可能会有问题。

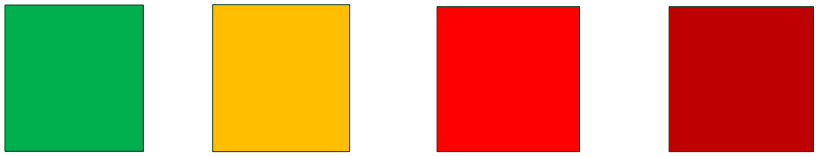
Treadmill 算法及其改进版本为我们提供了一种有意思的内存管理方法，但是它受无法预测的回收循环的困扰。Metronome 算法提供了受时间出发的垃圾回收机制。紧凑操作在垃圾回收循环中进行，但是必须为其提供一个精确的回收周期以保证实时系统能够按时完成任务并始终拥有充足的可用内存，否则系统便会出错。受事件触发的 Jamaica 系统使用一种新的内存对象来避免外部碎片，还需要通过选择一个适应于应用程序的内存块大小来最小化内部碎片量。垃圾回收器对内存的间接消耗及对于整体内存状态的以来降低了系统的可预测性。更糟的是，Jamaica 算法无法为内存的间接引用提供常数级的时间复杂度。

内存管理系统是可预测的硬实时系统的基础。上述概括的所有系统都未能提供一中可预测的内存操作能够很好的解决内存碎片问题，并且不依赖与整体内存状态。硬实时系统需要一个能够提供以上所有属性的动态内存管理系统。

第三章 紧凑内存管理系统框架设计

本章主要主要介绍了可压缩的嵌入式实时系统内存管理—紧凑适应算法模型，并提出了主要设计细节。本系统是从数据相关的概念如数据组织及管理方面抽象而出的。

本章中，状态图中不同的颜色的实体代表了在本内存管理系统模型中不同的内存状态，详细信息如下所示：



(a)未使用　(b)已使用　(c)页块内部碎片　(d)页面内碎片

3.1 紧凑操作

紧凑操作的确可以限制内存碎片的数量，然而不顾一切的进行全部内存的紧凑操作会导致系统性能大幅度降低，这种情况是实时操作系统所不允许的。因此内存紧凑操作的工作符合必须公平且逐步增加的均匀分配到整个内存操作过程中已获得良好的可预测的时间性能。

在 2.4.3 节提到的 Jamaica 系统[14]与上述情况不同，它所使用的对象模型消除了外部碎片，因此对于它来说紧凑操作是不必要的。但是对于 Jamaica 系统来说，它的性能并没有得到大幅提高，它的工作负荷转移到内存间接引用需要寻找内存对象最后一块内存块时，必须遍历整个链表，时间复杂度为线性级。我们期望在紧凑算法模型中，内存间接引用的时间复杂度被控制在常熟时间级，因此，Jamaica 内存对象模型并不能满足我们的目标。

紧凑操作的工作量主要由两个方面组成：复制内存对象及更新所有指向被移动内存对象的指针[16]。需要复制的内存对象的个数可以由对象的大小控制，但是需要更新的引用指针的数量却是不可预测的。在最坏的情况下会有 n 个已分配对象拥有指向被移动内存对象的指针，这将导致 n 次引用的更新。而且，我们要

找到内存中存在的这 n 个引用。只有当内存对象与引用他们的内存对象可以解耦和，才可以称紧凑操作为可预测的。关于解耦和的机制我们将会在下面的章节中提出。

3.1.1 地址空间概述

一般来言，有两种不同的内存视图：即抽象（虚拟）地址空间和具体（物理）地址空间。被分配的物理内存被放置在物理空间中连续的部分空间。对于每一个被分配的内存对象都有一个确切的抽象地址空间对应。应用程序无法直接映射到物理地址空间，它们首先对应抽象地址空间，然后抽象地址空间与物理地址空间一一对应进而确定了应用程序与物理地址空间中内存对象的对应关系。因此应用程序（抽象地址空间）与内存对象（物理地址空间）是解耦的。所有的内存操作都在抽象地址空间中。首先定义一些概念及符号：

定义 1：抽象地址空间为整数的有限集合，用 A 表示。

定义 2：虚拟地址 a 是抽象地址空间 A 中的一个元素，表示为 $a \in A$ 。

定义 3：物理地址空间为一个有限整数区间，用 C 表示。注意物理地址空间 C 为一个区间，所以它是连续的。此外物理地址和虚拟地址都是按照整数的大小顺序进行线性排序的。

定义 4：物理地址 c 是物理地址空间 C 中的一个元素，表示为 $c \in C$ 。

定义 5：一个内存对象是内存对象集合的一个元素，表示为 $i \in I(C)$ 。对于每个内存对象，在物理地址空间中都有两个元素 c_1，c_2（$c_1 < c_2$）来确定它的范围，比如说我们拥有对象 i，$i = [c_1, c_2] = \{x | c_1 \leq c_2\}$.

综上所述，每段抽象地址空间都唯一对应一段物理地址，一段物理地址对一个内存对象。反之亦然,每个已分配对象的物理地址对应唯一的一个虚拟地址。为了从形式上描述这种关系，我们为每个虚拟地址建立一个局部映射来指定它所关联的物理地址。

虚拟地址的映射关系$A \rightarrow T(C)$,将虚拟地址映射至内存对象。如果 Address(a) 被定义，我们则称虚拟地址 a 正在被使用。虚拟地址的映射是单映射的，也就是

说不同的虚拟地址空间会映射至不同的物理子区间，对于所有的正在被使用的虚拟地址a_1，$a_2 \in A$，如果$a_1 \neq a_2$,那么$address(a_1) \cap address(a_2) = \emptyset$.

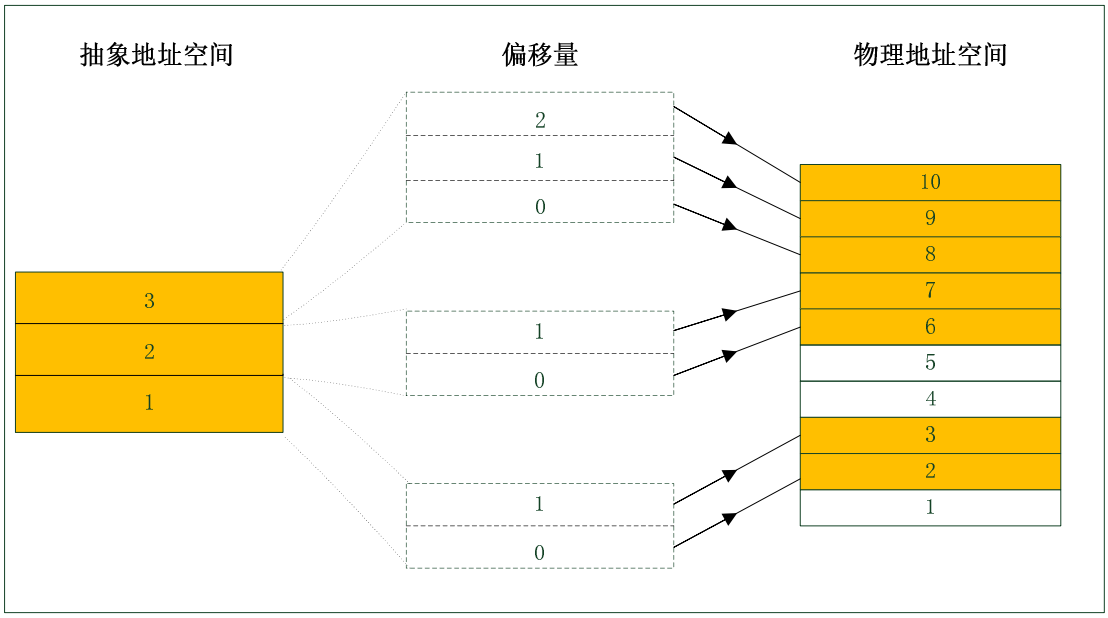
对于物理地址空间 C 中的元素进行访问需要两个信息：虚拟地址 a 和偏移量 o，这两个变量指出了想得到在内存对象$m = address(a)$中的哪个元素被请求访问，由此我们到处下一个定义。

定义 6：虚拟地址指针$a_p = (a,o)$,其中 a 代表使用的虚拟地址，o 表示一个偏移量，$o \in \{0,\ldots,|address(a)|-1\}$.

定义 7：虚拟地址指针 a_p 的集合称为虚拟地址指针空间，我们有 A_p 表示。

在虚拟地址指针空间 A_p 与物理地址空间 C 之间存在一一对应的关系，每一个虚拟地址指针 a_p 均通过虚拟指针映射关系 $A_p \rightarrow C$ 指向一个唯一的物理地址 c， 虚拟指针映射关系将虚拟指针 $a_p=(a,o)$映射至内存对象的物理地址 m=address(a)后顺序数第 o 个位置。

我们通过以下例子来阐明上面所涉及到的映射等定义的详细内容。假设抽象地址空间包含 3 个元素 A=｛1，2，3｝，物理地址空间 C 包含 10 个元素 C=｛1，2,……，10｝。现在我们拥有三个不同大小（包含有不同数量的物理地址）的内存对象被分配：address（1）=｛2，3｝，address（2）= ｛6，7｝，address（3）=｛8，10｝。虚拟地址加上其偏移量构成了映射至集合 C 的抽象指针。比如 pointer（1，1）= 3，pointer（3，2）= 10。如图 3.2 所示。



3.2 虚拟地址指针映射

下面我们结合具体的实现来描述虚拟地址空间 A 为我们带来的优势，假设一个应用程序通过一个连续的页表实现了在 A 中分配内存对象并有对 A 的引用。在此处我们将指针页表称为代理表，代理表中的实体指向物理地址空间 C 中实际的物理地址。

图 3.3 展示出了内存对象之间的相关性是如何被处理的。大型的数据结构通常是由一系列通过间接引用（比如链表，树等）链接的内存对象组成的。紧凑操作会导致该数据结构中引用的更新，如果引用均为直接引用，那么需要更新的引用的数量是不可预估的，因此每个内存对象的引用必须为间接引用。如每个引用均为虚拟指针。如下图所示。

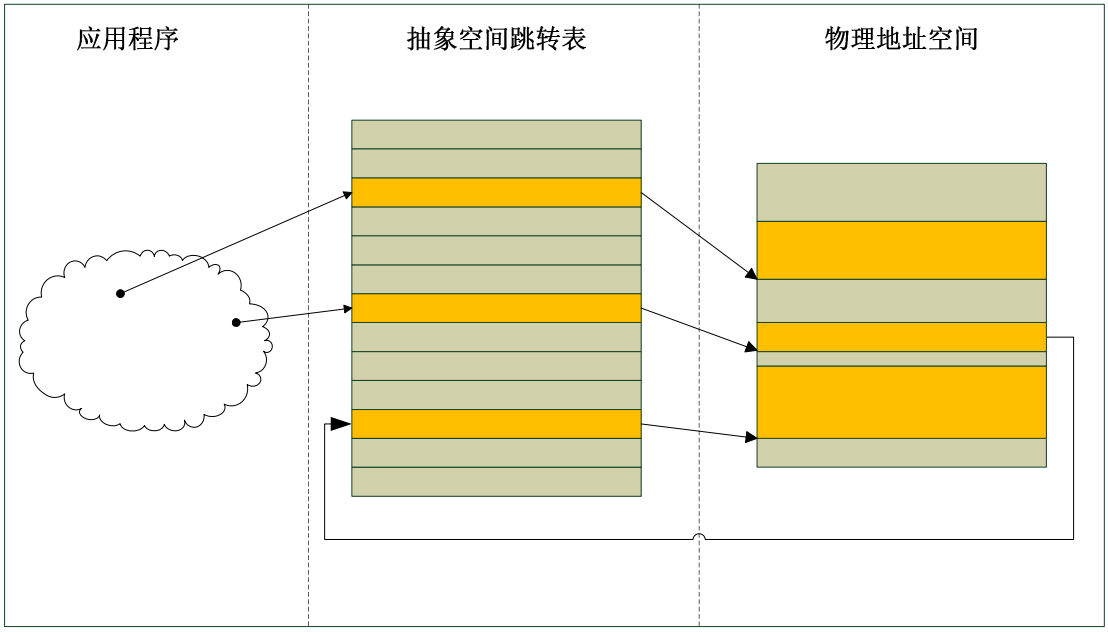


图 3.3 内存对象去耦合

间接引用使得紧凑操作时产生的引用的更新可以预估。如果出现了大量内存碎片时，物理地址空间 C 进行紧凑操作，对应的抽象地址空间 A 中的映射至 C 中的引用会随之更新。如图 3.4 及 3.5 所示，C 中的内存对象被移动，对应的 A 中的引用被更新。需要更新的引用是有限的：C 中的一个内存对象的移动对应 A 中一个引用指针的更新。对比而言，直接引用（依赖于内存对象大小）需要进行的的引用的更新是不可预估的。由此我们选择了拥有抽象地址空间的设计。

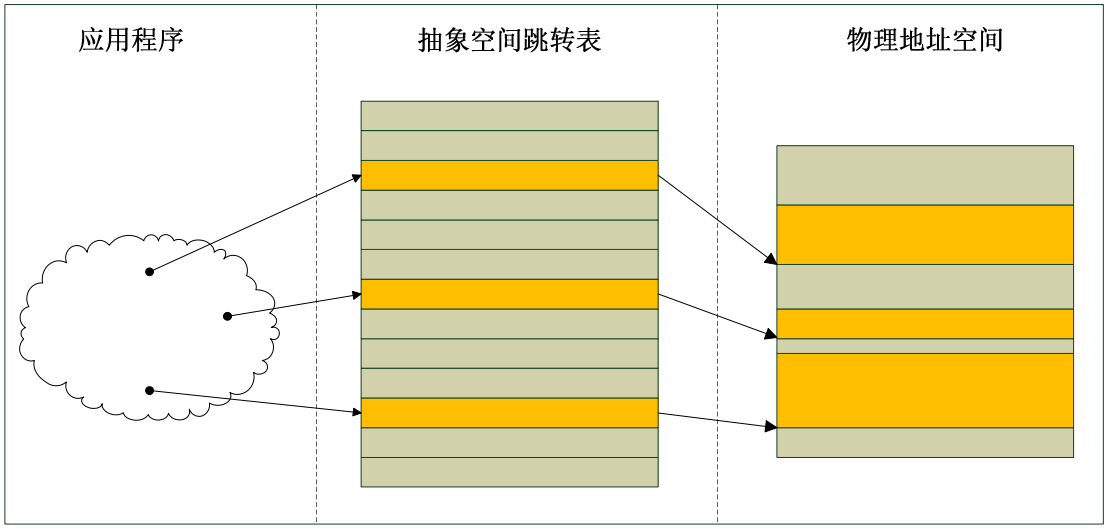


图 3.4 物理内存中存在碎片

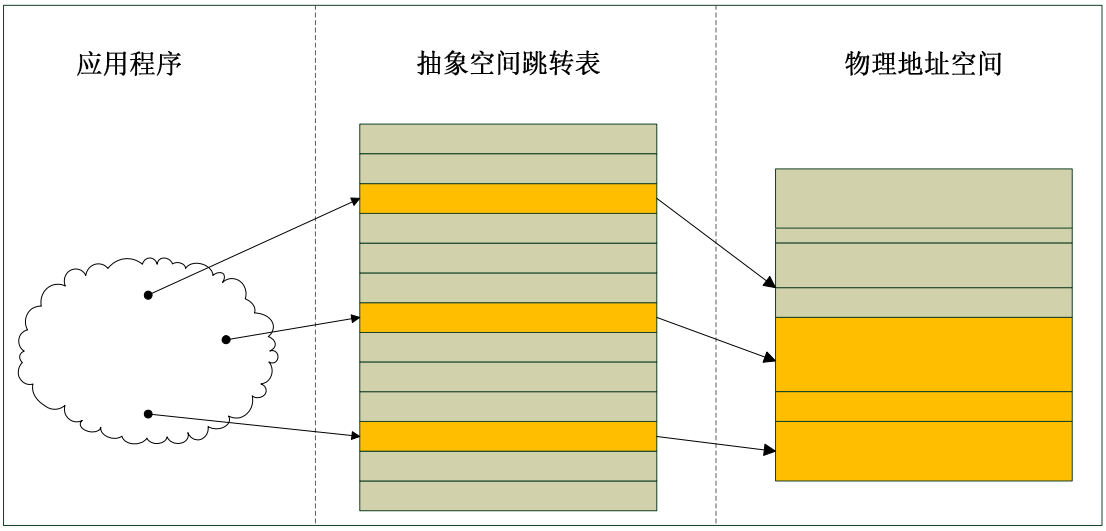


图 3.5 紧凑操作后的物理内存空间

3.2 系统接口函数

紧凑适应系统提供了三种显式内存操作函数，它们的实现部分将会在第四章中讲述。此处对它们进行抽象的描述：

- malloc（size）：内存分配操作，创建指定大小的内存对象。函数以正整数为参数，返回值为虚拟地址指针 a_p=（a，o），其中 a 表示虚拟地址指向了分配的内存对象，偏移量 o 设置为 0 时，为内存对象的起始位置。
- free（a）：内存释放操作以虚拟地址 a 为参数，将 a 指向的内存对象释放掉。
- dereference（a_p）：解引用操作，返回虚拟指针 a_p=（a，o）所对应的物理地址 c，其中 a 代表内存对象的虚拟地址，o 代表该内存对象内部的偏移量。

需要注意的是一个已分配对象的虚拟地址在其被释放之前不会发生改变。已分配对象的物理地址可能会因为紧凑操作而发生改变。

最后，我们指出物理地址空间与抽象地址空间的另外一个不同之处。在整个生命周期中，他们都会产生内存碎片，由于系统上层分配请求要求内存管理系统必须能够提供指定大小的连续的内存块，因此物理地址空间的碎片会产生一些问题。对于抽象地址空间而言，对于一次内存请求，内存管理系统只需要找到一个未被使用的虚拟地址而不需要考虑抽象地址空间的紧凑，此操作可以在常熟时间内完成。详细的实现细节将会在第四章提出。

3.3 系统中尺寸类概念

尺寸类的相关概念被用来进行物理地址空间的内存管理:

- 页面：内存被分为拥有固定大小的内存块的集合，我们称之为页，例如在我们的实现中每个页的大小 P=16KB。
- 页块：对每个页面进一步划分为更小的页块，在同一页中的页块拥有相同的大小。我们预先定义了一系列的页块大小 $S_1,\ldots,Sn$(当 $i<j$ 时 $S_i < S_j$)。共有 n 中页块大小，最大页块为 S_n。
- 尺寸类：页面根据其包含的不同大小的页块分类。由于有 n 中尺寸的页块，因此有 n 中尺寸类。每个拥有 S_i 尺寸页块的页面都属于第 i 个尺寸类。尺寸类中的页面通过双向循环链表连接。

内存分配请求由单一的页块来满足。通过所请求的大小 malloc（size）确定需要使用的尺寸类的大小，最适合 size 的页块大小为 S_i，其中 $S_{i-1} < size < S_i$。确定了最佳适合的页块大小后，我们便可以在尺寸类 i 中取出一页中的一块来满足所请求的内存。

如果页面被释放后，它便会从尺寸类中消除并仿佛空闲页面池中，该页面可再次被任何大小的尺寸类重新利用。

在本章结束的图 3.7 中我们给出了物理地址空间的组织形式的例子：有三种尺寸类，其中一个包含两个页面，另外两个类各包含一个页面。

3.4 系统碎片类型

尺寸类的概念引入了三种不同类型的内存碎片的概念：

- 页块内部碎片类型
- 页面内部碎片类型
- 外部碎片类型

下面我们将简要的阐述每种碎片类型的概念并描述它们对我们的系统设计所带来的影响。本章结尾的图 3.7 形象展示了各种碎片类型的区别。

3.4.1 页块内部碎片

页块内部碎片为每个页块尾部未使用的内存空间。假设在尺寸类 i 中存在一页面 p（p 中的页块的尺寸为 S_i），P 为页面大小，b_j（$j=1,\ldots,B_p$）为页面 p 中存在的页块，其中 $B_p=P/S_i$，对于其中的页块 b_j 而言，如果其被使用则定义 used（b_j）

=1，未使用则定义 used（b_j）=0。定义 data（b_j）为页块 b_j 中已经使用的内存的数量。通过以上数据，我们得知页面 p 中的页块内碎片的计算公式如下：

$$F_B(p) = \sum_{j=1}^{B_p} \text{used}(b_j) \cdot \left(S_i - \text{data}(b_j)\right).$$

页面 p 中已使用的内存数量为：

$$F_B(p) = \sum_{j=1}^{B_p} \text{used}(b_j) \cdot \text{data}(b_j).$$

如果要计算全部内存中存在的页块内碎片的数量，则需要将每个页面中的内存数量相加。假设 P_j（j=1,…,P_p）为系统中的所有的页面。如果 P_j 被使用了，则标记 used(P_j) = 1。那么内存中总共的页块内部碎片为；

$$TF_B = \sum_{j=1}^{P_p} F_B(p_j) \cdot \text{used}(p_j).$$

全部页面中被使用的内存数量为：

$$TU_M = \sum_{j=1}^{P_p} U_M(p_j) \cdot \text{used}(p_j).$$

单个页面中页块内部碎片所占的比例为：

$$f_B(p) = \frac{F_B(p)}{U_M(p)}.$$

所有已使用页面中页块内碎片所占的比例为：

$$Tf_B(p) = \frac{TF_B}{TU_M}.$$

如果谨慎仔细的选择页块的大小，那么页块内部碎片的总数可以限制在某因数 f。Berger【7】建议毗连的页块大小之间满足如下条件：

$$S_k = \lceil S_{k-1}(1+f) \rceil.$$

其中$k = 2, \ldots, n$。最小页块大小S_i及参数 f 都需要根据特定程序选定。Bacon et al[13 $f = 1/8$，这个参数使得小型尺寸类之间的尺寸差距较小，大型尺寸类之间的尺寸差距较大。

3.4.2 页块外部碎片

页块外部碎片为页面尾部的未使用的内存空间。如果页块的大小$S_1, \ldots, S_n$均为页面大小 P 的因子，那么便不存在页块外部碎片问题。但是如果我们利用公式（3.1）来选择页面内页块的大小，则我们必须要考虑页块外部碎片的问题。对于处于尺寸类 i 中的大小为 p 的页面而言，P 中的页面内部碎片定义如下：

$$F_p(p) = P \bmod S_i.$$

假设$p_j(j = 1, \ldots, Pp)$为系统中的所有页面，那么系统中所有页面中的页内碎片量为所有已使用的页面的$F_p(p)$的总和：

$$TF_p = \sum_{j=1}^{P_p} F_p(p) \cdot used(p_j).$$

一个页面中页块外部碎片的比例为：

$$f_p(p) = \frac{F_p(p)}{U_M(p)}.$$

所有已使用内存中页块外部碎片所占的比例为：

$$Tf_p = \frac{TF_p}{TU_M}.$$

3.4.3 页面内部碎片

页面内部碎片为已分配的页面中未使用的内存空间数量。这块内存被称为碎片或者说被浪费掉的原因为这段内存只有出现了在这个尺寸类中的分配请求时它才会被使用。例如，假设 p 为尺寸类 S_i = 32B 中的一页，如果只有一个内存块被分配，那么就有剩余的 P-32B 的内存空间未被使用，如果在将来一段时间内在此尺寸类中没有更多的分配请求到达，剩余的内存空间可能永远不会被使用，在这种情况下，32Byte 的内存块实际占用了整个页面。尺寸类 i 中的页面 p 的页面内部碎片量为：

$$F_S(p) = P - S_i.$$

因此通常而言系统中所有的页面内部碎片为所有页面中$F_S(p)$的总和。

假设有 n 个预先定义的页块大小$S_1,\ldots,S_n$，页面大小为 P，那么系统中所有页面内部碎片最大为：

$$TF_S = \sum_{i=1}^{n}(P - S_i)$$

在我们所设计的系统中，一个尺寸类至多包含一个拥有一个页块但却未完全使用的页面，因此整个尺寸类中的页面内部碎片被限制在单独一个页面的尺寸类中。关于这种性质我们会再接下来的一章中进行详细描述。

系统中存在的尺寸类越多，页块内部碎片的数量就会越少，但是页面内部碎片的碎片会增加，因此在我们设计尺寸类的时候应该考虑到页块内部碎片与页面内部碎片的平衡。例如在图 3.7 中的尺寸类 3 展示了更多的页块会导致更多的页面内碎片。

3.4.4 系统碎片总览

如图 3.6 展示了典型的物理地址空间组织结构：拥有三种类型的尺寸类，第一个尺寸类用了两个页面，后面两个尺寸类用了单一的页面。、

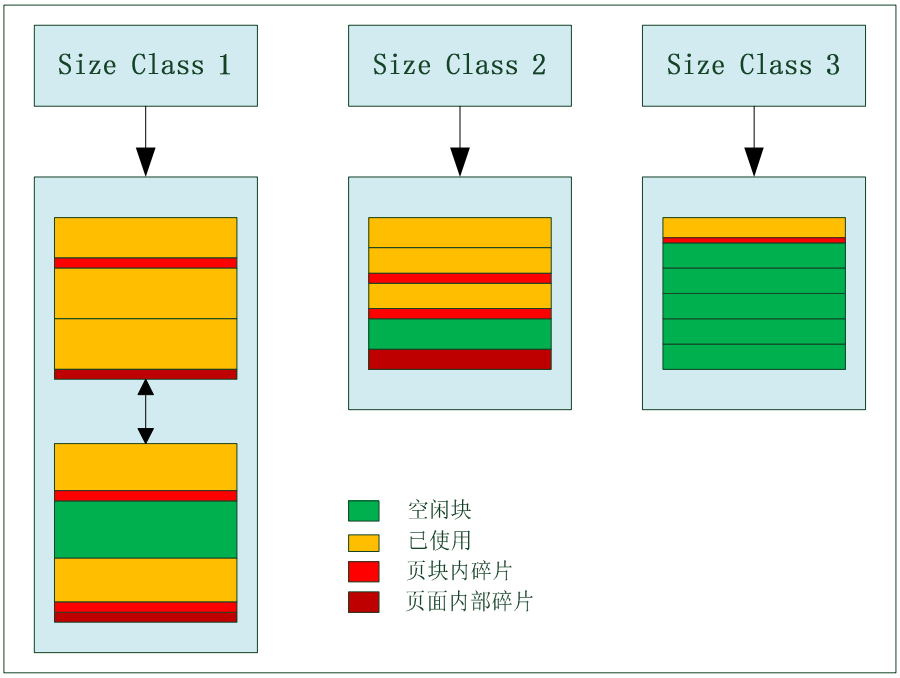


图 3.6 尺寸类及不同碎片类型

3.5 本章小结

在本章中我们阐释了紧凑系统中的尺寸类的概念，并介绍了抽象地址空间及物理地址空间的组织形式，进一步定义了抽象地址空间向物理地址空间的映射方法。在本章的结尾我们详细描述了不同类型的内存碎片的理论分析并呈现了各个碎片类型的可控范围，这些控制范围的存在使得我们的内存管理系统成为完全可预测的。

第四章 紧凑内存管理系统实现

本章主要包含一下两部分：第一部分我们介绍了紧凑内存管理系统的中紧凑算法的实现，然后我们讲述了自由链表的概念及物理地址空间管理的详情，比如说页面的管理及对于尺寸类概念的讨论。本章第二部分给出了紧凑系统的两种实现方式：可移动版实现和非移动版实现。另外我们更进一步的分析了每中系统实现的复杂度并展示了它们可控制的响应时间。

4.1 紧凑算法设计实现

正如 2.2 节所描述的那样，内存的紧凑过程是一个事件触发的操作，在紧凑操作执行后内存必须处于一个稳定的状态，或者称为“紧凑状态”。因此紧凑操作必须有足够的能力来保持内存的紧凑状态并且其响应时间必须能够满足实时系统的需求。

以图 4.1 为例，它展示了一个包含三个页面的尺寸类，每个页面都被分裂为特定的片段。如果我们想在严格的时间限制内要完成这些页面的紧凑操作是不可保证的。这些不必要的页面碎片的产生导致系统性能的严重降级。在最坏的情况下，每个页面纸杯占用了一个内存对象，其他内存对象均未使用，造成的内存的严重浪费。这种情况必须要被解决。

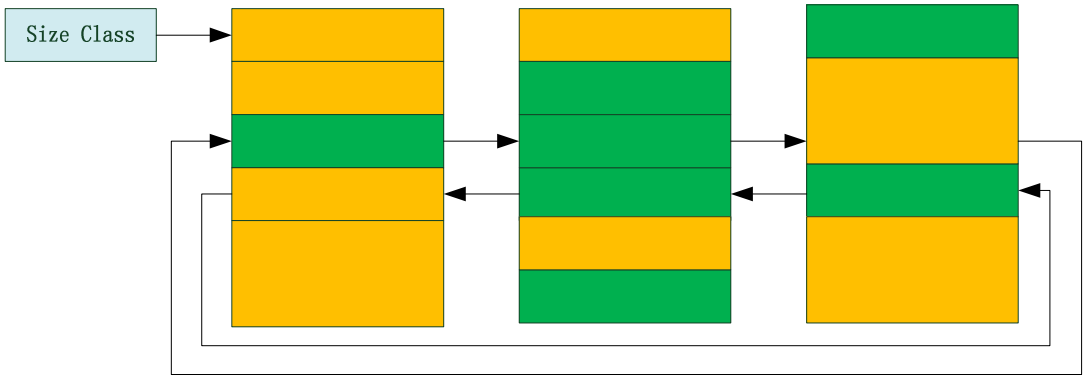


图 4.1 尺寸类中随机存在的碎片

紧凑适应模型中最关键的部分便是要保证内存中的尺寸类在所有时间内均为紧凑状态，也就是说在任意时间内，每个尺寸类中最多只能有一个页面没有被

完全使用，尺寸类中的其他页面必须被全部使用。内存紧凑算法通过如下行为来保证尺寸类始终保持一个紧凑的状态：即每当尺寸类中的内存对象被释放后，都应该执行紧凑操作以保证内存的紧凑状态。

与我们的内存紧凑策略不同的是 Metronome 的内存紧凑策略。它使用了时间触发机制，该策略存在一个专用函数用来记时，然后根据时间设定调用垃圾回收器在每个回收循环中有多少个页面应该被处理以消除内存碎片。如果空闲页面的数量第一某个给定的阈值，那么拥有很少对象的页面中的对象便会被移动至将满的尺寸类中。该策略中的紧凑操作负荷很大程度上取决于尺寸类中页面的碎片化程度。因而每个回收循环的周期的可预测性被严重降级了。我们提出的紧凑操作主要受到释放操作及被影响的尺寸类的范围的影响。其可确定性，可预测性要更加的优于 Metronome 的紧凑策略。

4.1.1 紧凑算法实现

在进一步讲解我们的紧凑策略算法之前，首先我们要声明两个约束条件及两条规则，再次说明每个尺寸类都是一个双向循环的链表。

约束条件 1：在一个尺寸类中未完全被使用的页面最多只有一个。

约束条件 2：如果尺寸类中存在一个未满的页面，那么这个页面必须是尺寸类链表的最后一个元素。

内存紧凑算法要遵循一下两条规则：

规则 1：如果一个满页 p 中的一个内存对象被释放后，该尺寸类中不存在其他未满的页面，那么 p 称为此尺寸类中的未满页并被放置在尺寸类链表的尾部。

规则 2：如果一个满页 p 中的一个内存对象被释放后，该尺寸类中存在一个未满的页面 p_n，那么 p_n 中的一个内存对象将移动至 p，如果 p_n 变为空，则将其再该尺寸类中移除。

并非所有释放工作都需要移动内存对象，以下情况都不需要移动内存对象：

- 发生释放操作的尺寸类中只有一个页面时，不需要移动内存对象
- 释放操作恰好发生在尺寸类中的未满页面中时，不需要移动内存对象，当释放掉的内存对象为该页面中的最后一个内存对象时，将此页面在尺寸类中移除。
- 内存对象释放操作发生在一个所有页面全满的尺寸类中，在这种情况下我们只需要更新一下页面在列表中的顺序，将发生改变的页面放置在尺

寸类链表中的最后一个位置。

当一个内存对象移动至另外一个页面时，由于对象的虚拟地址指向了另外一个物理位置，因此我们需要更新其在虚拟地址空间中的引用。

内存的紧凑算法在表 4.1 中展示出来，如果内存对象的释放操作导致一个全空页面，那么将全空页面在尺寸类链表中删除即可，尺寸类仍然保持在紧凑状态，在此种情况下 3　不需要调用紧凑算法。关于释放操作的更多细节模型我们将在 4.4.3 及 4.5.3 节进一步指出。

```
void compaction(changed_page, size_class) {
        if ( changed_page != last_page)
        {
            if ( is_full(last_page) )
            {
                set_last(changed_page)
            }
            else {
                move( object, last_page, affected_page);
                abstract_address_space_update(object);
            }
    }
}
```

如图 4.2 及 4.3 所示的例子使用了规则 1 来避免内存的碎片问题。在图 4.2 中的尺寸类中的页面几乎全部用完，绿色部分为刚刚释放的内存对象，由于释放操作后在该尺寸类中只存在一个未满页面，因此不需要执行内存紧凑操作，只需要将其移动至尺寸类链表的尾部即可，如图 4.3 所示。

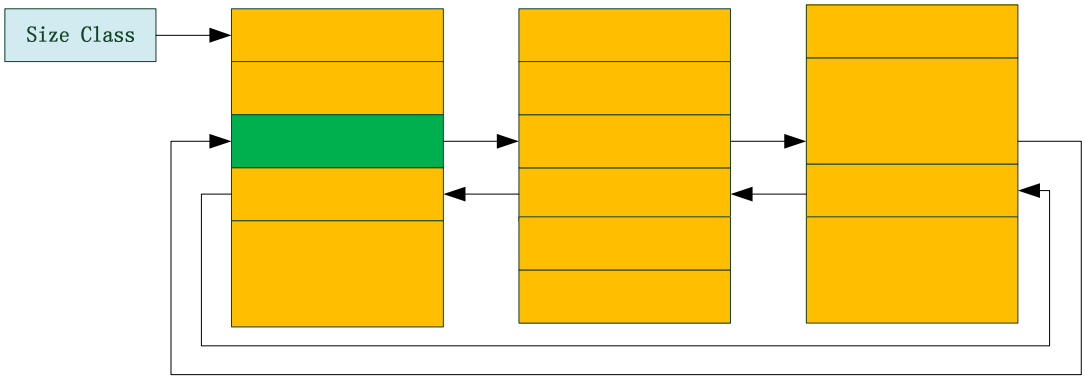


图 4.2 绿色内存块将被释放

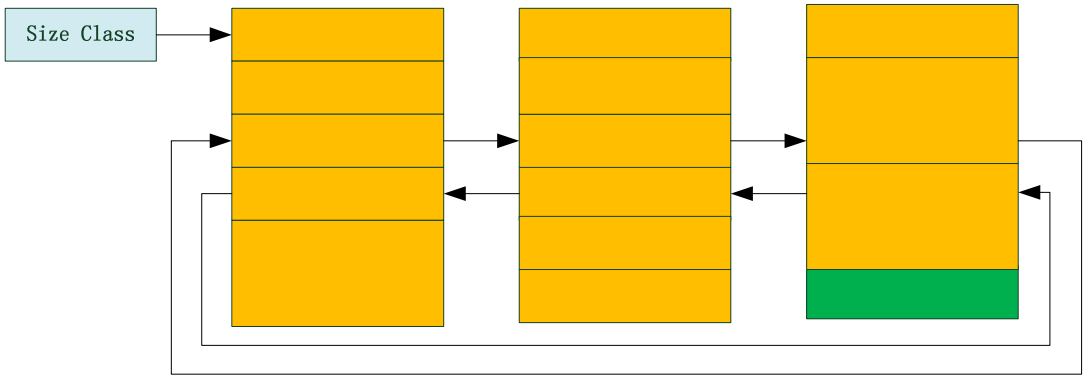


图 4.3 应用规则以后的尺寸类

如图 4.4 及 4.5 所示的例子使用了规则 2 来避免内存的碎片问题。图 4.4 中所示的尺寸类中有两个全满的页面和一个未满的页面。绿颜色标记的为一个刚刚被释放的内存对象，由于对象释放后存在两个未满的页面。约束条件被打破，因此要进行内存对象的移动操作来保持约束条件的成立，执行内存的紧凑算法后，未满的页面只有一个保持了约束条件的成立，尺寸类处于一种紧凑状态。

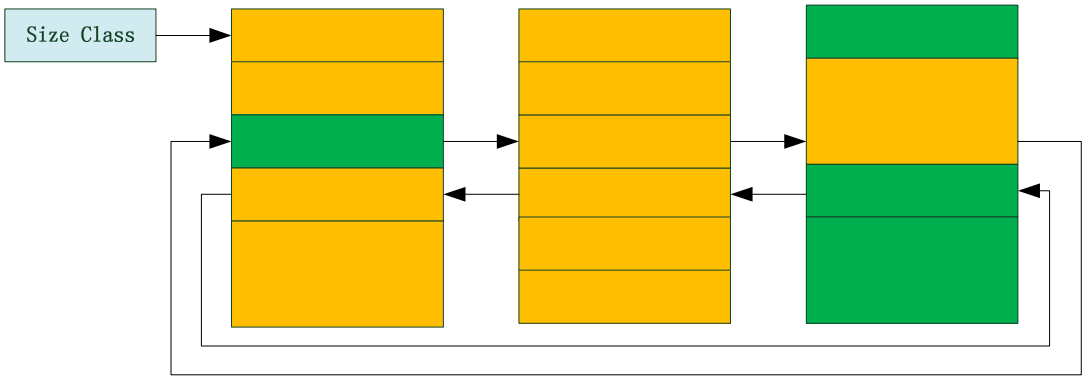


图 4.4 绿色内存块将被释放

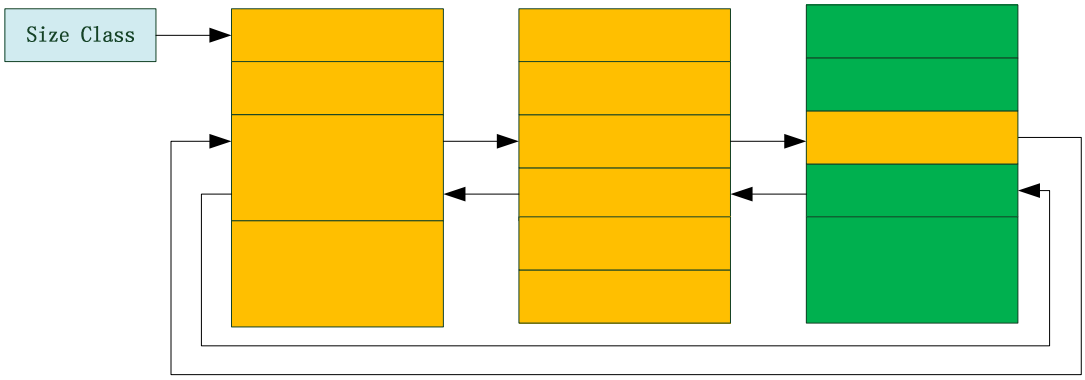


图 4.5 运用规则后的尺寸类

4.1.2 算法的复杂度分析

紧凑内存管理算法保持了尺寸类的紧凑状态，然而这是由增加释放操作的复杂度为代价的。假设 a 为一个内存对象的虚拟地址，C(a)为该内存对象的尺寸类。释放处于虚拟地址 a 的内存对象的复杂度包括:释放内存对象的复杂度 $\theta(remove(C(a)))$，取决于尺寸类中的内存块的大小。如果经过释放操作后，尺寸类 C(a)变为非紧凑状态，就必须执行内存对象的移动操作，将尺寸类链表尾部的页面中的内存对象移动至刚刚释放掉的内存对象所占用的空间.紧凑操作的复杂度为$\theta(compact(C(a)))$,另外参数 do_compaction(C(a))表示是否执行紧凑操作，如果在释放操作后存在非紧凑的尺寸类，则 do_compaction(C(a))=1，表示要执行内存紧凑操作。

释放尺寸类 C(a)中的对象 a 的复杂度可以表示为如下公式：

$$\theta(free(a)) = \theta\left(remove(C(a))\right) + \theta(compact(C(a))) \cdot do\_compaction(C(a)).$$

在上面所示的公式中，如果我们知道了所有子函数的渐进特性，那么我们便可以知道内存对象释放操作的渐进特性。关于子函数渐进特性的测量方法我们将在下面一节提出。

4.2 自由链表设计实现

在我们所设计的算法的很多关键点，都需要算法能够在常数时间内检测出众

多实体对象中的空闲对象。对象可以是抽象地址空间的一个元素，也可以是物理地址空间中的一个对象。例如，我们需要在常数时间内检测出页面中空闲页块，或者抽象地址空间中的一个虚拟地址块。

常用的组织空闲内存块的方法是使用一个 LIFO 链表将空闲内存块链接起来并保存链表的头部节点。链表中的每个实体都保留有指向下一个空闲块的指针，由于指针被存放在空闲块的一部分空间中，因此没有内存被浪费掉。假设链表已被初始化，则获得一个空闲的内存块只需要常数时间级，但是，链表的初始化工作需要耗费线性时间，其值依赖于我们需要追踪的内存块数量，起始状态下，所有的内存块都是空闲的，时间复杂度为 number_entries，因此为了获得常数时间内级别，我们必须使用更加高效的空闲链表。

空闲链表一般包含两部分指定的区域：next 域和 head 域，next 域指向下一个从未使用的内存块的位置，head 域是初始化阶段完成后空闲链表的头部。初始状态下 head = null,next = 0。

当一块未使用的空闲内存块被使用后，增加 next 值。已使用的内存块被释放后，我们将其放置在链表的头部。在申请获得一块空闲内存块时，我们首先需要判断 next 域与总共内存块数 number_entries 的关系，如果$next \leq number_entries$，则在 next 位置取得空闲内存块，否则，在空闲链表的首部指针处取得空闲内存块。

空闲链表的部分操作算法如下所示：

表 4.2 get_entry 函数实现

```
void free_list_entry *get_entry(free_list) {
        if ( free_list->next > free_list->number_entries)
        {
            free_entry = free_list->head;
            free_list->head = free_list->head->next_entry
        }
        else {
            free_entry = free_list->next;
            free_list->next += free_list->entry_size
            if ( mode_switch() ) {
                free_list->next = free_list_mode;
             }
          }
```

```
        return free_entry
    }
```

一块空闲内存块通过简单的入栈操作就可以加入空闲链表，整个操作的时间复杂度为常数时间范围.如公式 4.2

$$\theta(\text{add\_free\_entry}()) = \theta(1)$$

表 4.2 展示了通过使用空闲链表的概念我们可以在常数时间内得到一个空闲的内存块。假设使用$\theta(\text{next}())$表示找到一个从未使用过的内存空闲块所需要的时间复杂度，$\theta(\text{head}())$表示从头部返回空闲内存块的时间复杂度。更进一步的，如果空闲链表返回了由 next 指针指向的内存空间，我们设定$\text{fl_mode}() = 1$，否则设其为 0。申请空闲页的时间复杂度函数如下述公式所示：

$$\theta(\text{get\_free\_entry}()) = \begin{cases} \theta(\text{next}()) = \theta(1) \text{ if fl\_mode}() = 1 \\ \theta(\text{head}()) = \theta(1) \text{ if fl\_mode}() = 0 \end{cases} \quad (4.3)$$

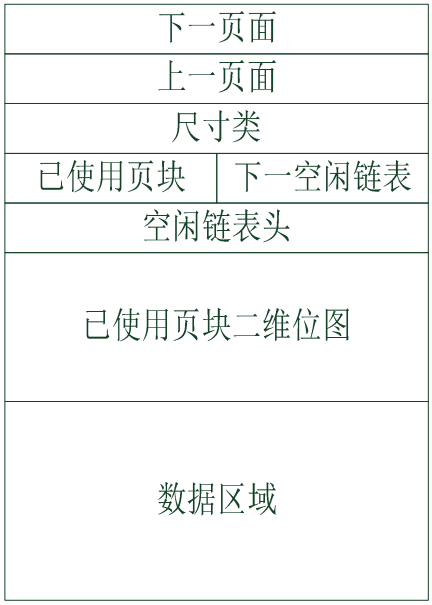
由上式可见，获取一个空闲内存空间的函数 get_entry()的时间复杂度为常数级别的。

$$\theta(\text{get\_free\_entry}()) = \theta(1) \qquad (4.4)$$

4.3 页面管理设计实现

物理地址空间被均分为大小相等页面空间，在我们的实现中我们使用了大小为 16KB 的页面。页面大小可以根据特定的应用程序而定制改变。在我们的实现中，最小的页块大小 S_i 为 32Byte，页块的大小也是可以改变的，但需注意的是页块最小为 16Byte，页块大小的最优选择方法取决于公式 3.1，在这里我们令参数$f = 1/8$。

每个页面的大小为 16KB，在其 16KB 内存中，每个页面都包含一个额外的头部用于对页面进行管理，其头部的管理信息如图 4.6 所表述：



4.6 页面数据结构视图

下面我们将详细介绍页面头部的每个区域的含义及内存紧凑管理系统如何使用它们。

4.3.1 尺寸类链表域

头部中的前两个区域分别为下一个页面（next page）和上一个页面（previous page），分别用于存储同一尺寸类中相邻的两个页面的地址信息。这两个引用信息构成了尺寸类的双向循环链表。在此处尺寸类使用单向链表是不可以的，因为如果我们需要将一个页面删除或者移到另外一个位置，我们将失去受影响的页面的前任页面。在一个单向链表中查询一个页面在最坏情况下需要遍历整个尺寸类链表，因此尺寸类中的每个页面都需要两个指针来指向其前驱以及后继，否则关于尺寸类的操作不可以在常数时间内完成。通过使用双向循环链表我们可以迅速的从链表头部达到链表尾部。正如之前 4.1.1 节讲述的那样，尺寸类的最后一个页面在紧凑算法中扮演着特殊的角色，它可以是未满的页面。

每个页面的头部都包含对内存的引用，占用 32 位内存。由于所有的空闲链表都被全局的页面空闲链表管理，尺寸类中基于双向循环链表的操作 remove_page()，add_page()，get_head()，get_tail()，get_predecessor()，get_successor() 都能够在常熟时间内完成。

$$\theta(\text{remove\_page}()) = \theta(\text{add\_page}()) = \theta(1) \qquad (4.5)$$

$$\theta(\text{get\_head}()) = \theta(\text{get\_tail}()) = \theta(1) \qquad (4.6)$$

$$\theta(\text{get\_predecessor}()) = \theta(\text{get\_successor}()) = \theta(1) \qquad (4.7)$$

4.3.2 尺寸类区域

尺寸类区域表示了页面中的尺寸类实体，它指向了本页所在的尺寸类链表。因此我们可以在任意一个页面都可以很轻松的直接访问到尺寸类链表的头部，这种性质对于内存紧凑操作具有重要意义。

为了进一步阐明该区域的重要性，我们考虑如下情况：当一个内存对象被释放时，内存对象的虚拟地址及其页块的物理地址都没有关于该内存对象的尺寸类的信息，但是尺寸类的信息决定了要释放的内存对象的大小。我们还要检查紧凑

操作是否必要被执行，因此每个页面都需要知道它的尺寸类。

该区域由于必须持有指向尺寸类的指针，它需要占用 32 位的内存空间，函数 get_sizeclass()及 get_size 占用常数时间。

$$\theta(\text{get\_sizeclass}()) = \theta(\text{get\_size}()) = \theta(1) \qquad (4.8)$$

4.3.3 已使用页块数

该区域包含了在该页中已经使用的页块的数量，用来检查该区域是否已满。函数 free_entries 进行了两个整型数的比较过程，时间复杂度为常数级。如果一个新的内存对象被使用，该尺寸类的最后一个页面的已使用页块数区域将被检查，如果该区域显示没有空闲的页块，则需要申请新的页面来满足分配请求。

$$\theta(\text{free\_entries}()) = \theta(1) \qquad (4.9)$$

4.3.4 空闲页块

页面包含有很多的页块，其中一部分为空闲，剩余部分以使用。如果一个页面中存在一个空闲的页块，空闲页块必须在常数时间内被发现。为达到此目的，我们必须维护一个空闲页块链表用以追踪所有的空闲页块。页面头部中的 Free-List Next 及 Free List Head 域结合起来组成了空闲链表。在我们的配置当中，最小的页块大小为 32Byte。因此一个页面中最多有 512 个页块。

通过此空闲链表的概念，我们避免了空闲页块链表的初始化工作。早期的初始化工作导致了不可预算的运行时间。向尺寸类中添加一个新的页面必须足够快能够在常数时间级完成。在一个页面全满的尺寸类中申请内存对象在反应时间是不应该让我们意识到一个新的页面必须添加至尺寸类来满足请求。

4.3.5 已使用页块

上面小结讲述了空闲块的链表，同样，我们也需要一个数据结构用来追踪一个页面中已经分配的页块。由于紧凑过程需要在常数时间级完成，由此在尺寸类

的最后一个未满页面中必须能够在常数时间内找到一个被使用的内存块并且在常数时间内将其移动到有内存对象被释放的位置。如果线性的遍历最后页面以查找已使用的页面块将会导致不可预测的运行时间，这种情况必须被避免。

下面两种方法被实现用来在常数时间内从页面中找到一个未使用的内存块。

已使用页块链表

我们可以使用一个双向循环链表来管理所有的页块。如果使用单向链表我们将面临一个链表元素的前驱节点不可查的缺点，这导致页块不能轻易的被移除。由于链表指针而耗费的页块中的内存空间对于已使用的页块来说不容小觑。这块内存位于页块的的最后一段位置，它必须被小心对待，以免造成此段数据的损坏。一个内存引用需要耗费 32bits 空间，由于我们使用了双向链表，每个页块需要 64bit 空间来存储引用地址，为了形成链表，我们总共话费了 64bit 空间。对于一块 32 字节的页块来说，其中的 64 位被用来存放其前驱和后继页块指针：这大约导致了每个页块中有 8/32=25%的内存浪费。解决如此之大的内存浪费的办法为使用 16 位的索引代替指向内存地址的指针。页块的索引标识了页块在页面中所处的位置，页面中的页块被顺序编号。通过此种方法，内存的浪费被减少了一半。

下面我们举例说明页块空闲链表及页面中已使用页块链表的具体形式，如图 4.7 及图 4.8 所示。顶部的顺序排放的整数表示了页面中页块的索引值，空闲页块用白色表示，在每个空闲页块的尾部都存放有指向下一个空闲页块的引用，此引用为物理地址。已使用页块链表进行了简化，我们使用页块索引进行组织。每个已使用的页块都包含两个索引值并由此形成一个双向链表。在图 4.7 中，空闲链表处于“下一页块”模式，因此它的每一个指针都指向了下一个未使用的内存块。已使用内存块指针指向第二块页块，该页块中索引又分别指出其前驱页块索引号 5，后继页块索引号 3。经过一段时间的分配操作后，页面的空闲链表策略改变为通过其头部指针来寻找空闲页块。图 4.8 展示出了空闲链表头部指向页块 2，页块 2 标识出其后继节点 4，从而可以通过指针头部获取空闲页块。

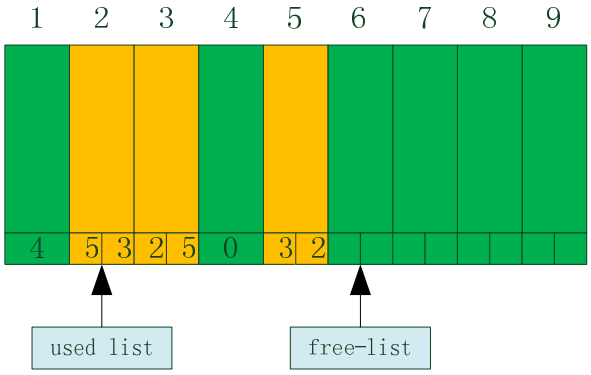


图 4.7 下一页块模式

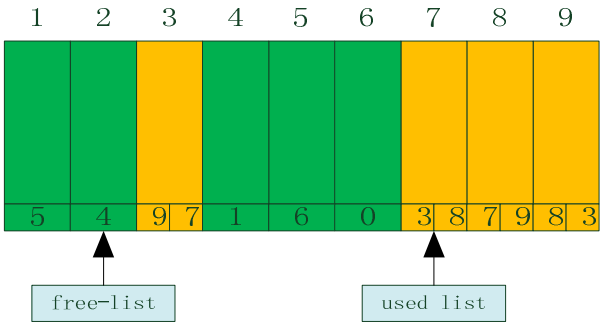


图 4.8 空闲链表模式

二维页块位图

在文献[9]中使用一个二维位图来管理整个内存空间。正如 2.3.4 节所描述的那样，我们同样使用一个二维位图来管理页面已使用的页块。页块的最小尺寸为 32B，一个页面中最多会有 512 块页块。由此，我们使用16×32尺寸的位图来记录页块状态。另外我们还需要另外的 16 位来记录每行中是否至少有一位被标记，这个附加的位串对于在常数时间内访问一个已使用页块来说是必要的条件。换句话说，一些 CPU 的指令可以在常数时间内访问一个位串，但是这些指令被限制在长度 32 位以内的位串中。因此为了获得一个已使用的页块，首先查找附加位串中一个被设置的位，然后在与该位对应的行中找到一个被标记的位。此处需要注意的是如果不存在这样的 CPU 指令，这些函数可以用 C 实现，这当然增加了算法的复杂度。

我们通过图 4.9 展示二维位图的概念。如果假设横向为 x 轴，纵向为 y 轴。附加位串为 y 轴，y 轴展示了存在两行，第 6 行和第 15 行中没有已使用的页块。系统找到一块已使用的页块的步骤如下：CPU 指令处理位串，并在常数时间内

返回该位串中以被标记为 1 的位 x 的位置。将此函数应用至图 4.9 y 轴方向的位串中将会返回值 0，意味着该位串中至少包含一个已经被标记的位，也就是某一行的 32 个页块中，至少有 1 块已经被使用。因此通过将 fls 函数应用至 x 轴方向的位串，我们便可以得出一个已使用的页块的位置。当查询到第一个已使用页块时返回 0，如果没有位被标记（没有页块被使用）则返回-1。

Listing 4.3: Returns the least significant bit of a bitstring x (IA-32 code)

```
static    inline int fls (int x){
        int    r;
        __asm__(“bsrl %1,%0\n\t”
                "jnz 1f\n\\t"
                "movl $-1,%0\n"
                "1:" : "=r" (r) : "g" (x);
        return r;
}
```

Listing 4.4: Returns the least significant bit of a bitstring x (ARM code)

```
static inline int (int x) {
int r;
__asm__("clz\t %0,%1" : "=r"(r) : "r"(x) : "cc");
return 31-r;
}
```

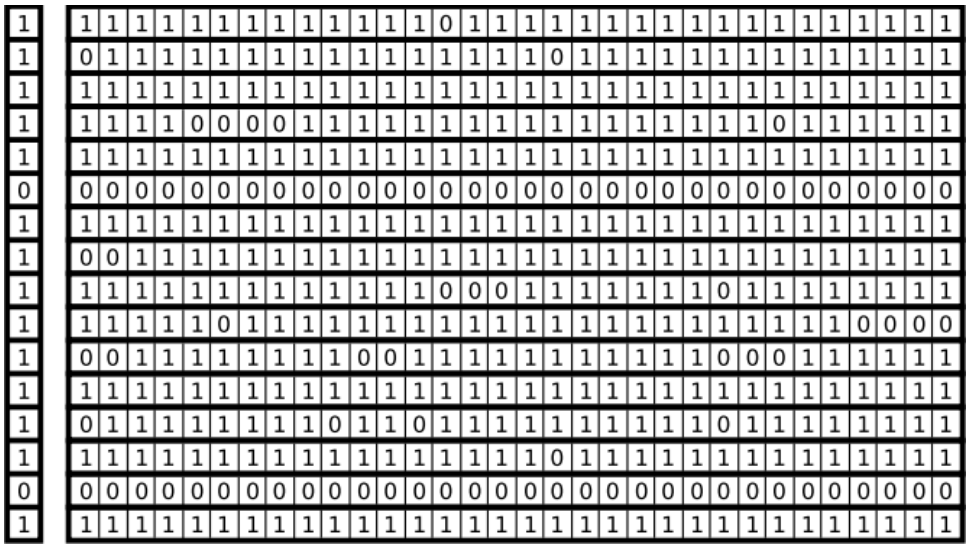


图 4.9 已使用页块二维位图

无论使用链表亦或使用二维位图（需要处理器支持）来管理已使用页块，我们都可以在常数时间内增加，获得或者删除一个元素，通过以下等式表式这种性质。

$$\theta(\text{add\_pb\_usedlist}()) = \theta(\text{get\_pb\_usedlist}()) = \theta(\text{remove\_pb\_usedlist}()) = \theta(1) \quad (4.10)$$

$$\theta(\text{add\_pb\_bitmap}()) = \theta(\text{get\_pb\_bitmap}()) = \theta(\text{remove\_pb\_bitmap}()) = \theta(1) \quad (4.11)$$

4.3.6 页面管理内存开销

页面头部的被用来管理已使用页块的数据结构决定了页面头部的大小，如果使用已使用页块链表来管理页面，页面头部为 24Byte，内存开销小于 0.15%。如果使用页块位图来管理，页面头部为 88Byte，由于此而引起的内存开销小于 0.6%。两种实现方案所引入的内存开销都处于可接受范围。

4.4 可移动版本算法设计实现

在紧凑内存管理系统的可移动版本中，内存对象会在紧凑的过程中在内存中移动。下面将展示出这种实现方案中内存的申请，释放，引用操作的算法实现，并对其复杂度进行详细分析。

4.4.1 概念介绍

抽象地址空间是一整块连续的内存空闲，抽象地址空间中的空闲对象通过一个空闲链表组织管理。

物理地址空间的组织形式正如 4.3 节所描述的那样，因此，每一个页面也为一块连续的内存空间。此外，页面中的每个页块都保持有其在抽象地址空间中虚拟地址的显式的引用。这个引用被放置在每个页块的尾部。在最坏的情况下，在最小的尺寸类中，这种开销占用了整个页块的 12.5%的空间。上述关系如图 4.10 所示：

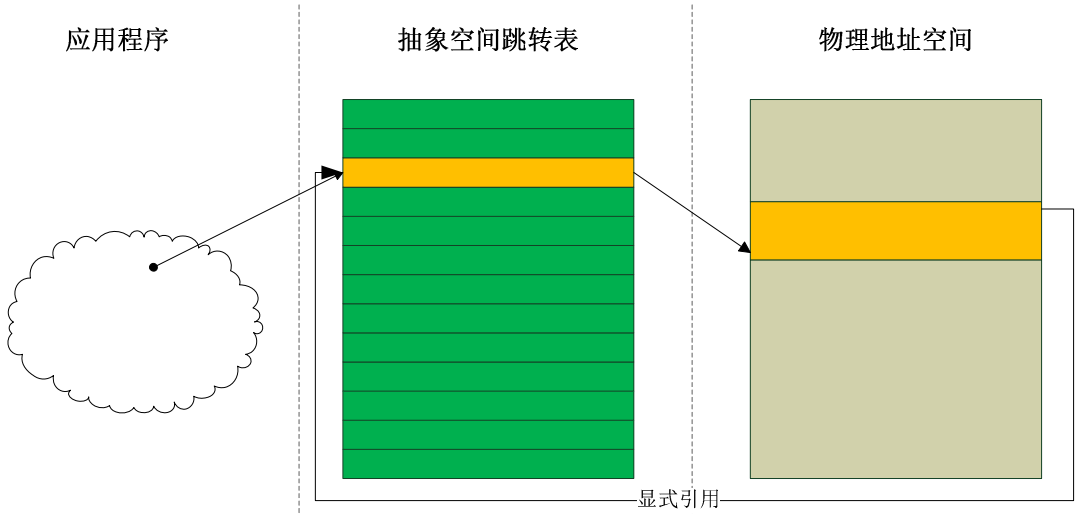


图 4.10 物理内存页块对于抽象地址的显式引用

4.4.2 分配操作实现

分配操作的主要实现细节如下表 4.5 所示。方法 get_page_of_size_class 会在常数时间内在一个相应的尺寸类中返回一个页面的引用，如果尺寸类中所有的页面均为满，那么我们将通过管理空闲页面的空闲列表获得一个新的页面。否则尺寸类中的未满页面将会被返回。整个函数时间复杂度在常数时间内。方法 get_free_page_block 的时间复杂度同样为常数时间级，通过使用页面中也空闲页块链表来获得页块，通过一个位操作即可声明一个页块是可用的。通过上面章节的介绍我们可知，抽象地址空间也是通过空闲链表来组织的。方法

create_abstract_address 同样花费常数时间来获取一个虚拟地址并将其指向物理地址空间中获得的空闲页块。总的来说，cfm_malloc 为常数级时间复杂度，即

$$\theta(\text{cfm\_malloc}(\text{size})) = \theta(1)$$

表 4.5 移动版本的内存分配操作

```
void **cmf_malloc(size) {
    page = get_page_of_size_class(size);
    page_block=get_free_page_block(page);
    set_used(page,page_block);
    return create_abstract_address(page_block);
}
```

4.4.3 释放操作实现

内存释放操作算法的详细信息如表 4.6 所示。方法 get_page_block 为常数级复杂度，因为他只需要获取虚拟地址所指向的内存位置。方法 get_page 进行了几次固定的算数运算，也即内存中的页面都是对齐的，对于一个给定的页块，我们可以计算出它所在的页面的开始位置，其时间复杂度仍然为常数级。方法 get_size_class 可以通过获取页面中的 Size-Class 域直接完成，常数级时间复杂度。方法 set_unused 只改变位图中某位的值，常数时间复杂度。方法 add_free_page_block 和 add_free_abstract_address 均可以在常数时间级内在对应的链表中增加一个元素。在尺寸类中删除一个页面的方法 remove_page 同样可以在常数时间级内完成：首先将页面在尺寸类链表中删除，然后将该页面加入空闲页

面链表。因此函数 cfm_free 的时间复杂度取决于紧凑算法的复杂度。

表 4.6 移动版本的内存释放操作

```
void cfm_free(abs_address) {
    page_block = get_page_block(abs_address);
    page = get_page(page_block);
    size_class = get_size_class(page);
    set_unused(page, page_block);
    add_free_page_block(page, page_block);
    add_free_abstarct_address(abs_address);
    if ( page == empaty)
    {
        remove_page(size_class, page);
    }else {
        compaction(size_class, page);
    }
}
```

由于可能需要移动内存对象等操作，紧凑算法 compaction 的时间复杂度鱼与尺寸类中的页块的大小成线性关系，由于页块与抽象地址空间的直接映射关系，函数 abstract_address_space_update 的时间复杂度为常数级别。

因此在最坏的情况下，cfm_free 函数的时间复杂度与页块的尺寸成线性关系：既可以表示为：$O(\text{cfm\_free}(\text{abs\_address})) = O(s)$,其中 s 为虚拟地址 abs_address 所指向的尺寸类中的页块的大小。对于固定尺寸的尺寸类而言，函数的时间复杂度为常数级别的。

4.4.4 解引用操作实现

在移动版本的实现中，对内存对象的物理地址的访问是通过简单的对虚拟地

址的解引用来实现的，一个显式的解引用操作并非必要的，他/她可以被实现为如表 4.7 所示的一行简单的代码。

表 4.7 移动版本-解引用操作

```
void *cfm_dereference(abs_address,offset) {
    return *abs_address + offset;
}
```

4.5 非移动版本算法设计实现

此种实现方法之所以称为“非移动版本”，是因为内存对象在它们的整个生命周期中，其在物理地址空间中的位置都不会发生改变，即便是紧凑操作执行时。在此种实现方法中存在三种形式地址空间：抽象地址空间，虚拟实际地址空间及物理实际地址空间，如图 4.11 所示。引入虚拟实际地址空间的想法来源于现代操作系统中广为人知的虚拟内存的概念，物理空间中非连续的内存空间可以表现为连续的地址。

4.5.1 概念介绍

在非移动实现方案中，抽象地址空间仍然为一块连续的内存空间。区别在于我们不在使用空闲链表来管理空闲的抽象地址，因为我们建立了由内存对象到抽象地址的隐式的映射。该映射可以用来在常数时间内对抽象地址空间进行更新操作。下面的章节将会详细的描述这种映射。

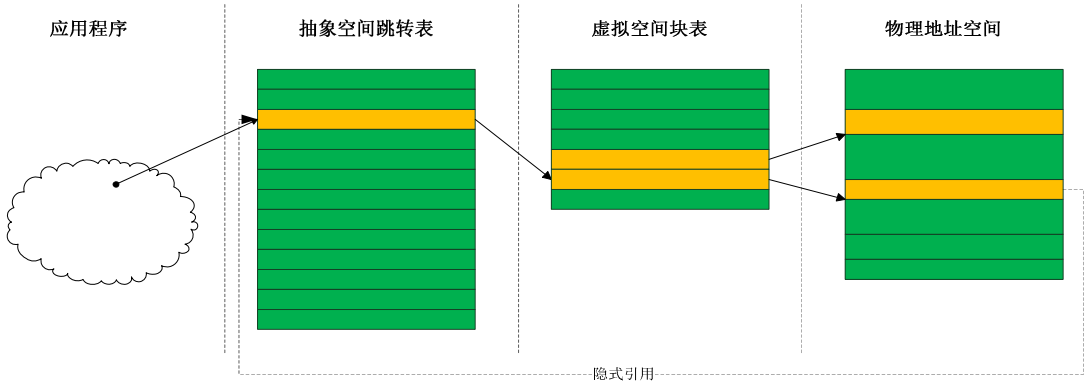


图 4.11 非移动版本的内存视图

首先，我们需要解释该实现方案中物理地址空间管理方式的改变。物理地址空间由一个虚拟内存管理，虚拟内存由相同大小的内存块组成。物理地址空间被连续相对的分为同样大小的块帧。下面我们用 Sb 表示块帧的尺寸，这里所讲的内存块及块帧的概念不要域之前说描述的页块的概念混淆，它们是全局的，面向整个内存的，在页面之上的概念。因此每个块帧都可以通过它的序列号直接访问。空闲的块帧由空闲链表组织。

抽象地址空间是预先分配的，它包含像块帧数量一样多的潜在的抽象地址。每个唯一的抽象地址对应一个内存对象 m：抽象地址空间中的第 k 个元素所对应的抽象地址即为物理地址空间中的第 k 个块帧的序号。因此，我们不需要显式的存储每个页块的地址，此处相比于移动版本的实现，节省了一些字节的开销。另外，只要我们在物理内存中分配了块帧，便可以立即获得它的抽象地址。

内存对象也即块帧与抽象地址的关系可以用如下所示的映射关系规范的表达出来。序号映射$T(C) \to N_0$将内存对象映射至正整数。内存对象$m \in T(C)$是指将对象 m 的第一个块帧映射至序号。假设$bf_adr(m)$为内存对象 m 的第一个块帧物理地址，bf 为连续内存块帧的起始地址，那么内存对象 m 的第一个块帧的序号可以通过一下公式计算得出：

$$ord(m) = \frac{bf_{adr(m)} - bf}{S_0}.$$

内存映射$T(C) \to A$将内存对象映射至抽象地址。内存对象$m \in T(C)$的抽象地址的位置为$ord(m)$，因此内存对象的第一块帧的位置决定了该内存对象的抽象地址。

块帧表记录了由抽象地址映射至物理块帧的映射关系。在我们的实现中，块帧表是建立在页面之上的，这简化了页选取的操作。非移动版本实现方案中内存的组织关系如图 4.11 所示。

内存对象在连续的虚拟内存中分配，但是可以在分散的物理块帧中存储。我们仍然使用页面，尺寸类，页块的概念，但不同的是此处的页面为一个虚拟页面，它不包含任何数据，在它的数据段中包含了可以指向物理内存中真是数据的块帧表。另外，为了管理方便的目的，所有的页块的大小都必须是块帧尺寸的整数倍。

因此每个页块中都会包含整数个块帧。此处我们不可以直接使用公式 3.1，我们需要聚集页块的大小变为块帧尺寸的更多倍数。

4.5.2 分配操作实现

非移动版本的分配算法由表 4.8 展示出来。与移动版本的实现相比，我们通过一个循环来处理分配对象块帧的相关操作。函数 number_of_blocks 对于指定的尺寸类而言是常数，获得一个空闲并创建相应的页块表也只需要常数时间级。因此分配算法的时间复杂度线性接近于页块中页帧的数量。可用如下公式表述 $\theta(\text{cfnm\_malloc(size)}) = \theta(n)$,其中$n = s/s_b$，s 为尺寸类中页块的大小。这也意味着其在尺寸类中拥有线性复杂度。

非移动方案中的函数 create_abstract_address 隐式的建立内存对象与抽象地址之间的引用关系。

表 4.8 非移动版本的内存分配操作

```
void **cfnm_malloc(size) {
    page = get_page_of_size_class(size);
    page_block = get_free_page_block(page);
    for ( i = 1 to number_of_blocks(page_block) )
    {
        block_frame = get_free_block_frame();
        add_to_block_table(page, page_block, block_frame);
    }
    set_used(page, page_block);
    return create_abstract_address(page_block);
}
```

4.5.3 释放操作实现

释放算法如表 4.9 所示，与移动方案不同的是，我们必须释放每个被要释放的内存对象使用的块帧。这就需要遍历内存对象中所有块帧，正如以前所提到的，总共有$n = s/s_b$个块帧，s 为尺寸类中页块的大小，sb 为一个块帧的大小。另外，内存的紧凑算法实现也有所不同：内存对象的移动操作只需要虚拟的更新内存的对象的块页表即可。当释放一个内存对象时，实际更新了页表中的 n 处位置。因此，非移动版本中的内存释放算法时间复杂度为$\theta(cfnmfree) = \theta(n)$，对于给定的尺寸类而言，也为常数级别。

表 4.9 非移动版本的内存释放操作

```
void cfnm_free(abs_address) {
    page_block = get_page_block(abs_address);
    page = get_page(page_block);
    size_class = get_size_class(page);
    for (i = 1 to number_of_blocks(page_block) )
    {
        block_frame = get_block_frame(page_blcok, i );
        add_free_block_frame(block_farme);
    }
    set_unused(page, page_block);
    add_free_page_block(page, page_block);
    add_free_abstract_address(abs_address);
    if ( page == empty )
    {
        remove_page(size_class, page);
    }
    else {
        compaction(size_class, page);
    }
}
```

4.5.4 解引用操作实现

非移动方案为我们提供一种直接访问内存位置的方法，而不是通过给出一个抽象的指针。解引用操作时间复杂度为常数级别，相比较移动版本的实现方案多次运算过程。算法的详细内容由表 4.10 给出，本算法中 s_b 代表了每个块帧的大小 sb。

表 4.10 非移动版本的内存解引用操作

```
void *cfnm_dereference(abs_address, offset) {
    return (*(*abs_address + ( offset / s_b)) + ( offset    % s_b));
}
```

综合以上分析我们可以得知，非移动实现方案中的紧凑算法获得了常数时间级，但是其分配算法的复杂度可以由常数级增长为线性级。两种方案的解引用和释放算法拥有相同的实现方式。

4.6 系统总内存开销

本小结我们对两种版本的实现方案中页面管理所带来的内存开销进行详细的分析。下面分别讨论了使用页块链表以及页块位图两种方法所带来的内存开销分析。

首先我们分析已使用页块链表的相关信息。已使用页块链表需要在每个已使用页块中保存其前驱及后继的引用指针。由于我们使用页块索引替代指针以减少内存浪费，每个已使用页块中包含 4Byte 的额外开销。在最坏的情况下，所分配的内存对象由最小尺寸类（<32Byte）分配，每个 32Byte 的页块中都会包含 4Byte 的管理信息，管理信息占据了总内存的 12.5%，在移动版本的实现中每个已使用的页块包含了显式的指向其抽象地址的指针。在最坏的情况下，这种内存开销也为 12.5%，在非移动版本中没有内存页块与抽象地址之间的直接引用关系，但是它使用了一个页块表来建立页块与抽象地址之间的映射关系。这个页块表的大小是不依赖与尺寸类的大小而改变的。每个页块表项占用 4B 空间。每个页面总的

页块表为 4*512B。

在最坏的情况下，可移动版本及不可移动版本的内存开销为 $\frac{512\times4+512\times4+24}{10384+24}=25.1\%$.由于非移动版本实现包含了一个静态的页块表而移动版本实现中的页面中的显示引用依赖于尺寸类的大小，由此通常来说移动版本的内存开销一般而言会小于或者等于非移动版本实现的内存开销。

使用位图的方法没有为每个已使用页块增加开销，它只是扩展了页面的头部。在最坏的情况下两种实现方案使用位图的内存开销为$\frac{514\times4+88}{16384+88}=13\%$，同样移动版本的内存开销一般而言会小于或者等于非移动版本实现的内存开销。

表 4.11 不同版本中的内存管理开销

	移动版本	非移动版本
已使用页块链表	<25.1%	<25.1%
已使用页块位图	<13.0%	<13.0%

4.7 局部紧凑算法实现

到目前为止，我们所遵循的基本原则为每个尺寸类在所有时间下都是保持该尺寸类的全部紧凑状态，也就是每个尺寸类中至多有一个页面非空。现在我们将此原则稍微修改一下使其更加宽松：一个尺寸类可以拥有多于一个未满的页面，我们不再要求尺寸类一直保持紧凑状态，而是允许其在预定的范围内可以处于非紧凑状态。通过变量 max_number_nf_pages 来改变每个尺寸类中所允许的未满的页面的个数，便可以获得不同的紧凑率。举例来说，如果 max_number_nf_pages = 1 表示允许的未满页面数为 1 个，也即尺寸类全部紧凑。进一步如果 max_number_nf_pages 变大，即允许未满的页面上增多，内存管理系统的反应速

度便会越快（最快的情况下便是没有紧凑操作），但是速度提高的代价是更高的碎片率。通过这种伸展性我们可以使内存紧凑系统适应更多的变化，达到系统运行速度与碎片化程度之间的平衡。我们可以通过程序性能分析来确定一个最优的max_number_nf_pages 值，其必须在编译时确定。

我们通过也许方案来实现内存的部分紧凑状态。一个尺寸类实例包含三个区域：满页链表（full_pages_list）,未满页面链表（not_full_pages_list）及未满页面数（number_not_full_pages）

由此尺寸类包含两个双向循环链表：一个链表包含了所有已满的页面，另外一个包含所有未满的页面。未满页面链表的元素的数量不会超过max_number_nf_pages。未满链表中的元素以一种弱顺序链接：已使用过半空间页面存放在链表的头部，未过半的页面存放在未满链表的尾部。

4.7.1 分配操作

由于大部分满的页面存放在未满页面链表的尾部，分配操作在未满链表的最后一个页面上进行以增加页面被完全使用的机会。如果页面变为被完全使用，则立即将其移动至该尺寸类中全满的页面中。另外如果尺寸类的未满链表中不存在未满页面，则需要将一个新的页面添加至尺寸类中以满足内存请求。

4.7.2 释放操作

内存部分紧凑策略为避免或者减少紧凑操作的发生，当发生内存页块的释放操作后进一步需采取怎样的操作依赖于该页面状态以及该页面所在的尺寸类状态。如果页面为满页面并且$number_not_full_pages \leq max_number_nf_pages$，则将该页面移动至未满页面链表的尾部。如果$number_not_full_pages > max_number_nf_pages$，紧凑操作将被执行。未满链表头部的页面中的内存对象将被移动至刚刚由于释放操作而产生的内存空洞中。如果一个未满的页面中的内存对象被释放后该页面中的空间过半未满，则需要将该页面移动值未满链表的头部。

4.8 系统扩展及优化

4.8.1 指针算法

由于我们的系统中对抽象地址空间与物理地址空间做出来明显区别，我们的内存分配操作 malloc 函数会返回我们一个抽象地址而不是物理地址空间中的内存地址。因此指针相关算法需要进行适当调整以满足我们的系统模型。为了能够使用标准的指针算法，我们需要一个抽象地址的指针结构体。它包含了一个抽象地址域和一个偏移域，如表 4.11 中的代码段所示。我们通过结构体的使用，在表 4.12 中的代码段也获得了同样的效果。

一段使用常规的指针算法的 C 代码可以自动的转换成一段使用抽象指针的代码。转换器只需要确定受影响的代码片段，用抽象指针结构替代传统指针变量，然后将指针算法应用至抽象指针结构中的偏移量区域。

4.8.2 紧凑系统初始化

内存管理系统的初始化也是一个非常重要的部分，当一个嵌入式实时系统需要重启操作时，重启过程必须非常迅速。通过使用 4.2 节中所讲述的空闲链表的概念来管理所有的内存资源（如页面，页块帧，抽象地址等）所需要的初始化时间为常数时间。

4.8.3 动态抽象地址空间

到目前为止我们一直假设为静态的抽象地址空间，在紧凑系统的初始化阶段静态抽象地址空间必须被预先建立起来。由于待申请的内存空间基于大量的可变因子，其大小是很难确定的，因此在静态地址空间中找到任意一块大小的地址空间操作是很复杂的。如果申请了很多小的内存对象，便会存在很多虚拟地址，然而如果申请了很大块的内存对象，便会造成内存的浪费。

还需注意的是在静态抽象地址空间中的紧凑操作可能会导致内存对象映射关系发生不可预测的引用更新操作，这对于一个嵌入式实时系统而言是很难实现的。

下面我们提出了对于动态抽象地址空间的可行的解决方案，在我们的紧凑内存管理系统中主要使用这些策略。

内存移动版本中的实现

抽象地址空间和物理地址空间共享同一个空闲页面池。一个 16KB 的页面包含 4096 个抽象地址。同一页面中的抽象地址由 4.2 节中所述的空闲链表统一管理。位于页面内部的页头部被用来管理整个页面中的抽象地址，抽象地址空间和物理地址空间不在被区分，两种地址空间中的页面任意分散在整个内存空间中。

如果抽象地址空间中所有的页面中的抽象地址均已经被使用，则需要在空闲页面链表中重新获取一个空闲页面。如果抽象地址空间中某页面的所有抽象地址均被释放，则该页面需要移动至空闲页面池。

内存不移动版本中的实现

在不移动内存版本的实现中，我们使用页帧在内存中所使用的位置编号将页帧显式的映射至抽象地址。在动态抽象地址空间中使用如此的映射映射关系需要进行一些变动。我们需要一个抽象地址空间表以获得连续的抽象地址空间。由于页块帧的数量要多于抽象地址，因此并不是每个页块帧都可以通过映射函数映射至一个抽象地址。页块帧必须被分类存储于不同的空闲链表：一是可以被用来生成抽象地址的块帧（它们必须是所申请的内存对象的第一个块帧），二是不可以被用来生成抽象地址的块帧（它们不是所申请的内存对象的第一个块帧）。

4.9 本章小结

在本章中我们提出了该内存管理系统的两种实现方案：内存可移动版本及内存不可移动版本，我们分别分析了它们的渐进特性以及其内存额外开销等。之后，我们进一步阐述了内存紧凑策略的主要思想。在本章的最后我们提出了对该内存管理系统的一些扩展及优化。经过以上设计优化后展示给我们的是适用于 IA-32 及 ARM 结构的经过优化的内存管理系统实现方法。

第五章 系统验证及结果分析

在本章中我们将对我们实现的移动版本紧凑内存管理系统和非移动版本的紧凑内存管理系统进行测试，主要测试其分配操作及释放操作的实际应用中的表现。然后我们将所得结果与现存的显式动态内存管理算法的性能进行了对比。另外我们还展现出了实验环境及实验中所运用的策略。

5.1 实验环境

验证实验使用两种不同的平台来运行基准测试程序：Gumstix connex400 运行轻量级的硬件抽象层（HAL）来对运行时间性能进行测量。Linux 系统来对 CPU 指令的执行速度进行测量。

5.1.1 运行时间测量环境

运行时间的测量基于 Gumstix connex400 平台，Gumstix 平台广泛应用于嵌入式系统中。Gumstix connex400 平台包含主频为 400MHz 的 Intel XScale PXA255，64MB 的 SDRAM 内存。通常其上可运行 linux 操作系统。此处为了减少不相关因素对所测的运行时间的干扰，我们只在其上运行轻量级的硬件抽象层。该硬件抽象层足以运行一个简单的测试程序。这样的设置为算法时间复杂度的检测提供了精确的环境。它可以被用来优化和分析算法中缓存所带来的影响，这在我们进一步的工作中将会被使用。

Intel XScale 微架构中的性能检测单元（PMU）用来进行算法性能的分析工作。很少的几条指令便可以用来对感兴趣的代码段进行分析。PMU 是由多个计数器的集合组成的，这些计数器用来收集性能有关数据或者应用程序的时间特性。对我们的测量实验而言最重要的计数器为 cpu 周期计数器。它被用来统计所运行程序总共的运行时间。还有一些统计特殊处理器时间的计数器存在，如统计缓存的未命中率，但是这些计数器在我们接下来的实验中不会被用到。用于测量时钟周期的时间计数器在使用之前必须被设定为特定的频率，初始化代码如下表 5.1 所示。由于 Gumstix 拥有 400MHZ 的 cpu，我们可以将我们的时钟周期设置

为 4，从而使 CCLKCFG 为 396MHz.

表 5.1 时钟初始化代码

```
void _init_clock(void)
{
    unsigned long clock_frequency = 4;
    asm( "mcr\tp14,0,%0,c6,c0,0" : : "r" (clock_frequency) );
}
```

下表 5.2 中的代码端功能为启动时钟计数器，值 0x00000707 用于将事件计数设置为 0，清除所有标记位，禁用所有中断，禁用分频器，重置所有计数器然后启用所有计数器。该函数必须在需测量代码段之前调用。

表 5.2 使能计数器代码

```
void _start_clock(void)
{
    unsigned long flags = 0x00000707;
    asm( "mcr\tp14,0,%0,c0,c0,0" : : "r" (flags) );
}
```

性能测试结果由下表 5.3 中的函数取得，该函数必须在待测量函数之后调用。它返回运行待测量代码段所耗费的时钟周期数。

表 5.3 读取计数器计数

```
unsigned long _read_clock(void)
{
    register unsigned val;
    asm volatile( "mrc\tp14,0,%0,c1,c0,0" : "=r" (val));
    return val;
}
```

更多的关于性能测试单元（PMU）的函数功能可以在 Intex XScale 微架构的

帮助文档中获得，此处不再进一步讲述。

5.1.2 处理指令数

一种有效的减少性能测试所受到的干扰（如缓存影响）的方式为直接统计在每个内存的分配和释放过程中所运行的指令的数量。因为直接对指令数量的统计可以排除上下文切换所带来的影响，我们可以直接在一台普通的 linux 机器上运行我们的基准测试程序。我们的通过使用 ptrace 系统调用[17，18]作为主要工具，因为它可以实现一个父进程对子进程运行的控制。为达到期望效果，我们使用了 ptrace 的单步模式，每当测试程序执行一条指令，便会发送一条通知，我们需要做的便是统计收到的通知的个数。同统计运行时间的工具一样，同样有一套工具来执行初始化，测试开始，测试结束返回结果的工作。

5.2 实验结果分析

在本小节中我们将展示测试结果。通常有两种方式来评估内存管理系统：第一种方式使用标准应用程序或者合成负载模型[19]；第二种使用最坏情况下的合成负载模型[20]。文章首先比较了移动版本与非移动版本的内存申请是释放操作的性能指标。然后比较了各个内存管理算法的性能指标。

5.2.2 移动版本与非移动版本对比

分配操作

第一个实验我们比较了紧凑内存管理系统的移动版本与非移动版本之间的性能差别。基准测试程序使用了很简单的分配策略：程序开始运行时申请 20B 的内存对象，以后每次分配增加 2B。最终当请求的内存对象大小为 8KB 时，程序停止。图 5.1 和图 5.2 展示的结果对应了移动版本与非移动版本分别对应了其常数级复杂度和线性时间复杂度。

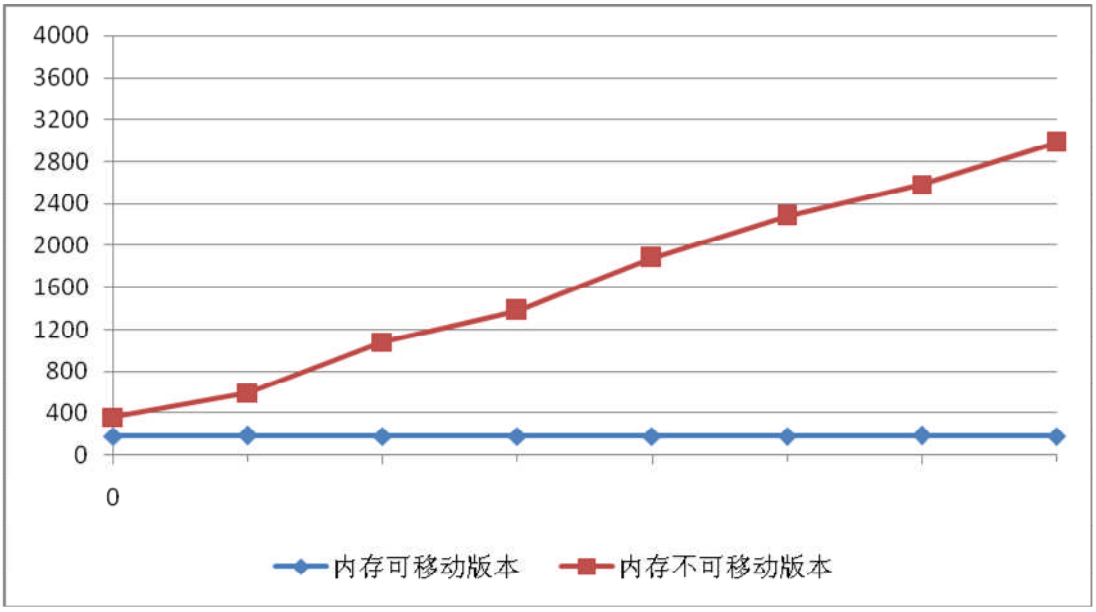


图 5.1 分配操作指令操作数

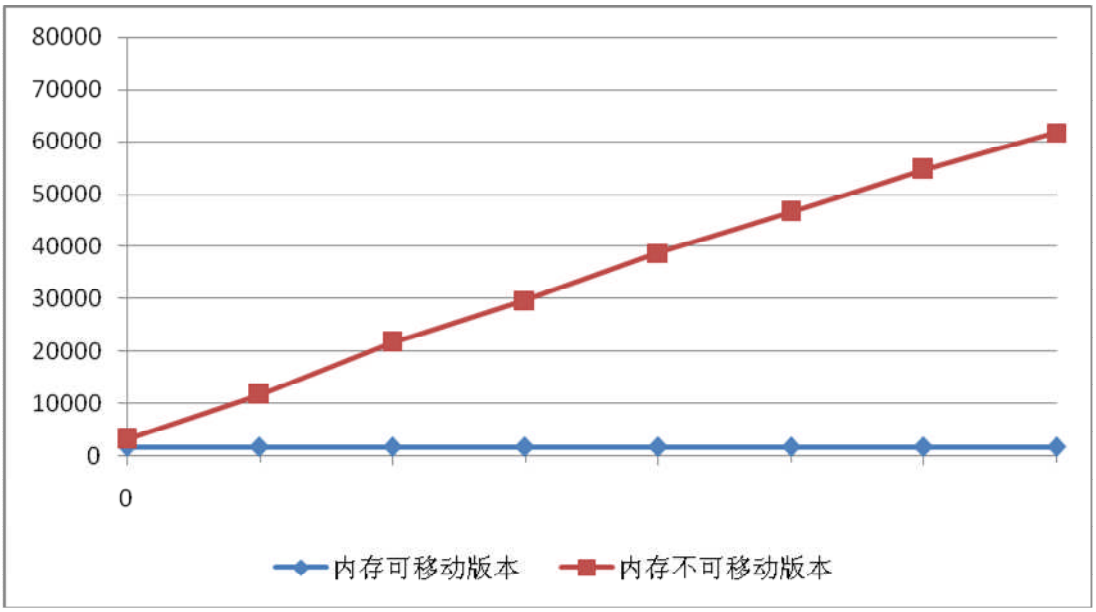


图 5.2 分配操作时钟周期数

指令计数和时钟计数很精确的描述了在可移动版本的实现中，分配操作的时间复杂度为常数时间级，其不依赖与它的尺寸类的变化。运行指令条数的平均值为 180.73 条，误差为 15.70.性能检测中心检测到的运行时间值为 1706.85.标准误差为 923.33。这些值都在表 5.1 中列出。

表 5.4：移动版本中分配操作测试结果

	指令条数	时钟 Ticks
均值	180.73	1706.85
误差	15.70	923.33

在非移动版本中分配操作的时间复杂度为线性级别，其值依赖于所请求分配的内存对象的大小。在一个给定的尺寸类大小范围内时，其时间复杂度为线性级别，这可以从结果图中得到验证。从图中又可以看出，随着时间的推移，函数的时间复杂度越来越大，这是因为后面的尺寸类中找到一块空闲块帧需要更多次循环遍历。

释放操作

上文中进行了大量的内存分配工作，在本小节中将按照同样的顺序将上面分配的内存释放掉。在图 5.3 和图 5.4 中展示了测的的指令数和时钟 Ticks 数目。图中很明显的展示了如果紧凑操作发生，紧凑操作的时间复杂度依赖于内存对象的大小。时钟 Ticks 数据显示如果紧凑操作发生，非移动版本的性能要优于移动版本。至于原因应为更好的缓存性能。

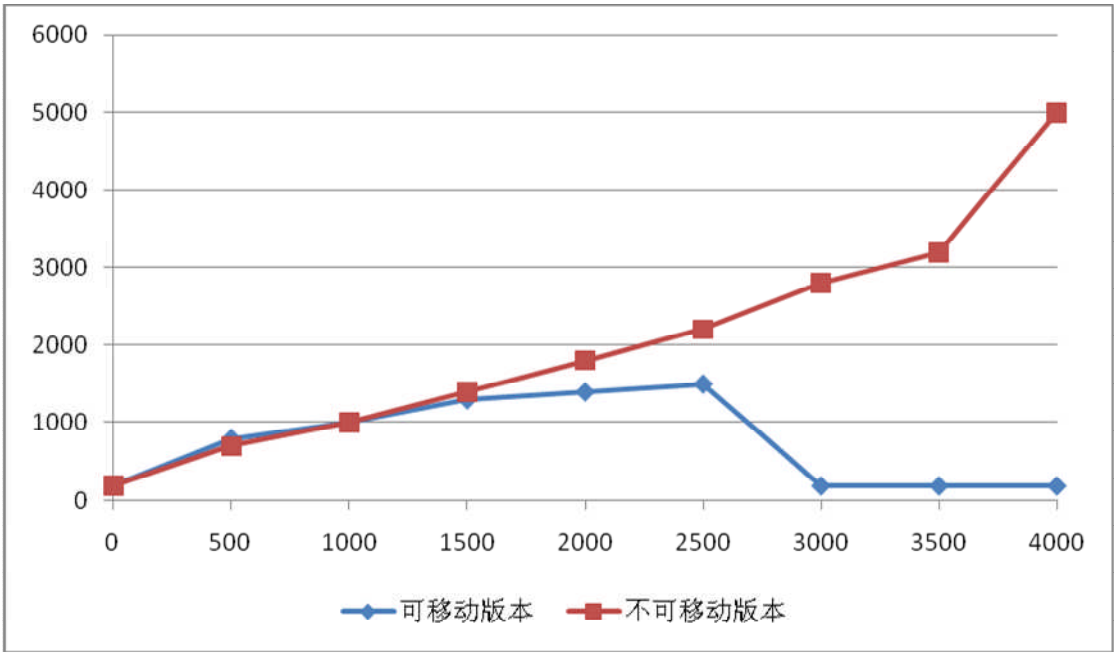


图 5.3 释放及紧凑操作指令操作数

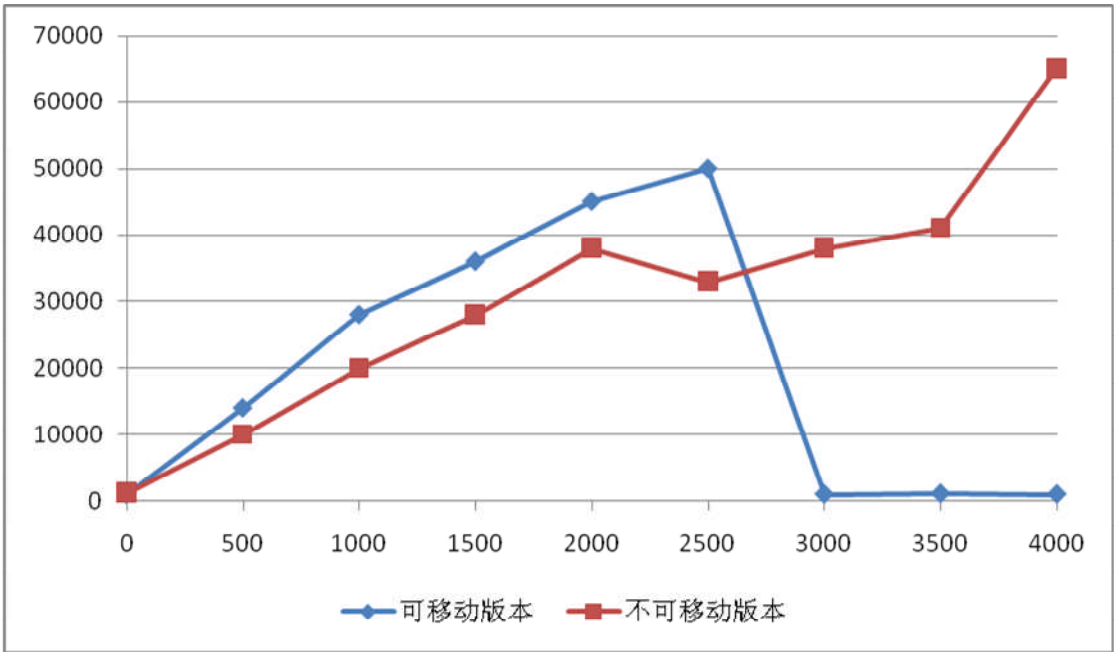


图 5.4 释放及紧凑操作时钟周期数

5.2.3 实时内存管理系统综合比较

在本小节中我们将紧凑内存管理系统与最快适应算法，最优适应算法，道格拉斯算法，Half-Fit 以及 TLSF 算法进行综合比较。基准测试程序的申请操作将不断增加申请的尺寸一直到全部内存被使用。然后每个偶数号对象被释放掉，最终每个被释放掉的内存对象重新被再次分配。

分配操作

第二次分配操作的统计数值将在下面给出。图 5.5 显示出由于空闲链表中存在大量元素，最快适应算法和最优适应算法极大的不可控性。道格拉斯算法也展现出了一些不可控性。Half-Fit 算法和 TLSF 算法的内存分配操作在常数级时间复杂度很快完成。紧凑内存算法的非移动版本的时间复杂度为线性级，移动版本为常数级。但是两个版本的紧凑内存算法都不是最快的。这是由于紧凑算法需要一些特殊的数据结构，而且内存要按照紧凑的目的组织。虽然其牺牲了性能，但是紧凑算法是可控的。表 5.3 展示了移动实现版本的测试数据。

表 5.5 可移动内存紧凑分配操作性能测试结果

	指令条数	时钟 Ticks
均值	169.61	1771.26
误差	8.63	538.90

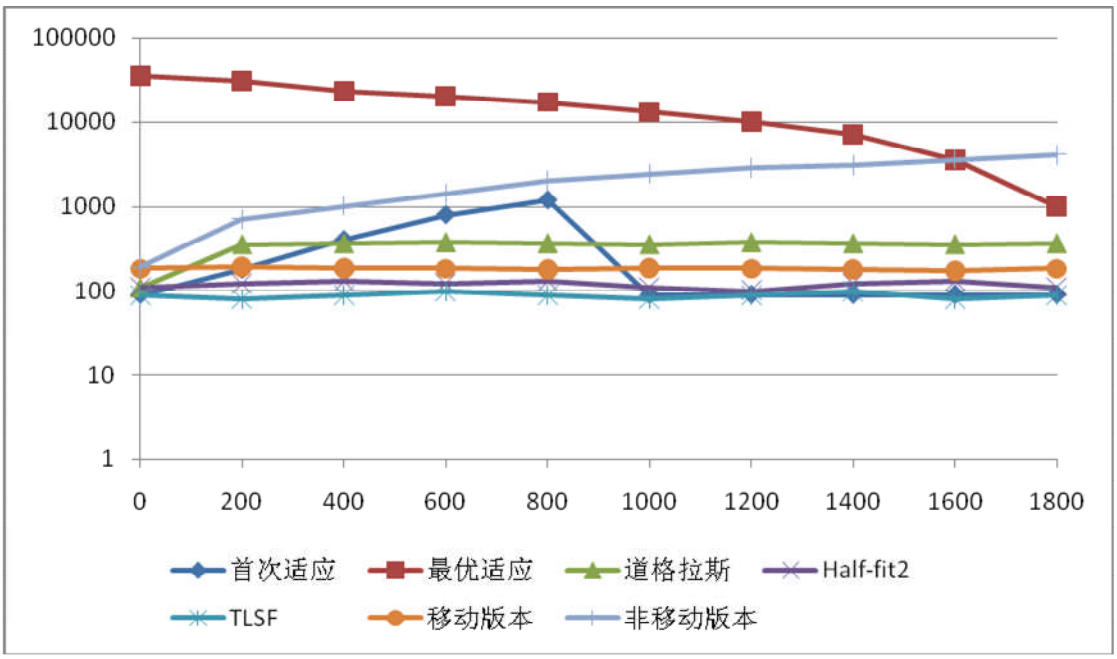


图 5.5 各算法指令操作条数

释放操作

内存释放操作的相关测试数据通过图 5.11 展示。最快适应算法，最优适应算法，道格拉斯算法，Half-Fit 算法，TLSF 算法的释放操作均为常数时间级。这些算法都将释放掉的内存范围添加至一个追踪空闲内存块的数据结构中。如果进行紧凑操作，紧凑内存算法的两种实现版本的释放操作均为线性级，其依赖于释放的内存对象的大小。内存释放操作均为可控的。紧凑操作的执行带来了额外的时间开销，但是两种紧凑内存系统都提供了一个完全可预测的内存空间。

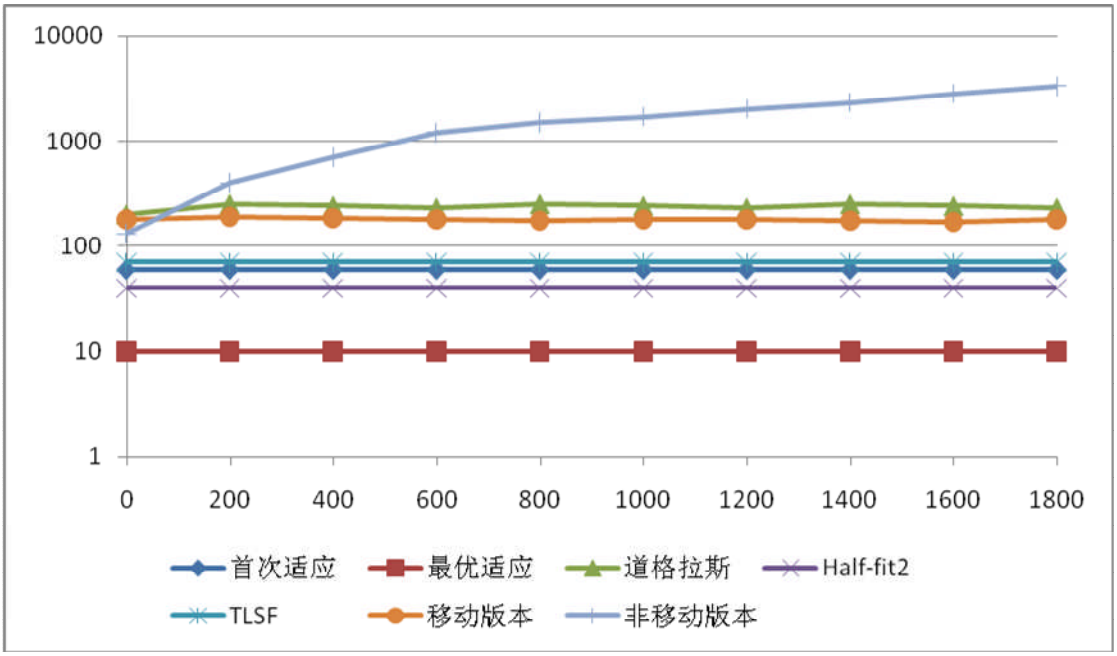


图 5.6 各算法释放操作运行指令数

5.2.3 实时内存管理系统综合比较

在本小节中我们将紧凑内存管理系统与最快适应算法，最优适应算法，道格拉斯算法，Half-Fit 以及 TLSF 算法进行综合比较。基准测试程序的申请操作将不断增加申请的尺寸一直到全部内存被使用。然后每个偶数号对象被释放掉，最终每个被释放掉的内存对象重新被再次分配。

分配操作

第二次分配操作的统计数值将在下面给出。图 5.5 显示出由于空闲链表中存在大量元素，最快适应算法和最优适应算法极大的不可控性。道格拉斯算法也展现出了一些不可控性。Half-Fit 算法和 TLSF 算法的内存分配操作在常数级时间复杂度很快完成。紧凑内存算法的非移动版本的时间复杂度为线性级，移动版本为常数级。但是两个版本的紧凑内存算法都不是最快的。这是由于紧凑算法需要一些特殊的数据结构，而且内存要按照紧凑的目的组织。虽然其牺牲了性能，但是紧凑算法是可控的。表 5.3 展示了移动实现版本的测试数据。

表 5.5 可移动内存紧凑分配操作性能测试结果

	指令条数	时钟 Ticks
均值	169.61	1771.26
误差	8.63	538.90

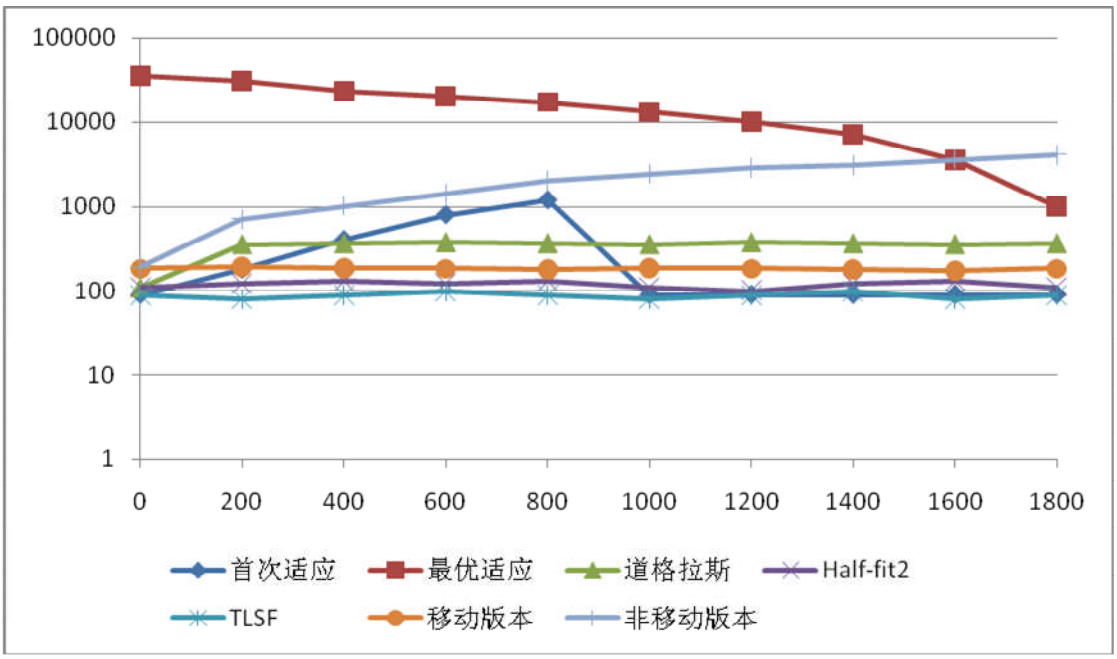
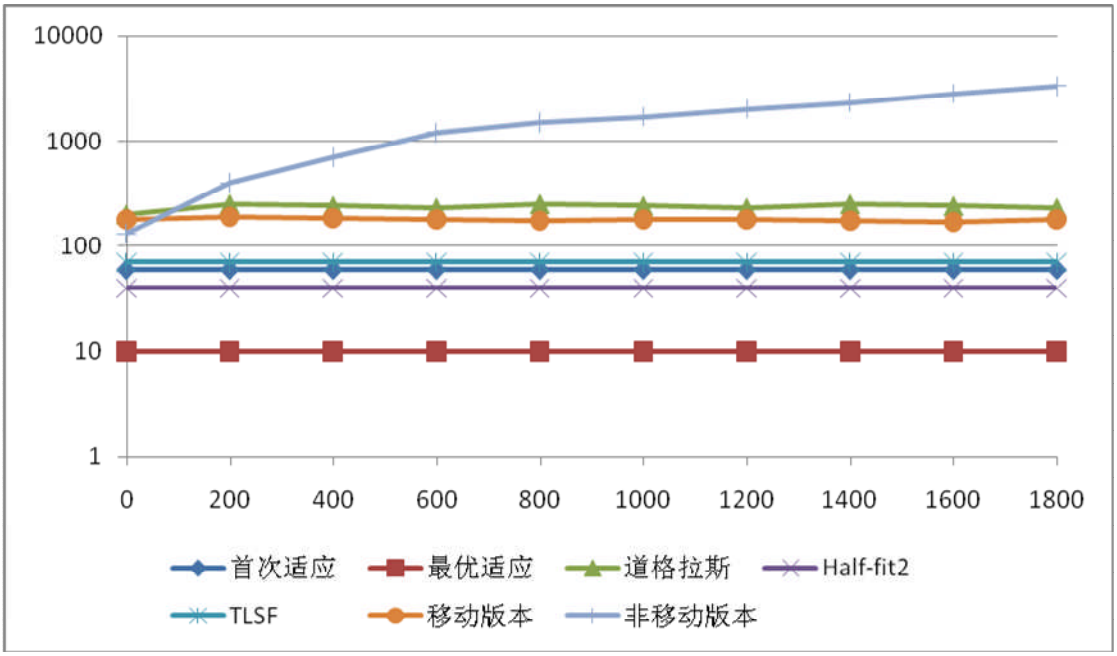


图 5.5 各算法指令操作条数

释放操作

内存释放操作的相关测试数据通过图 5.11 展示。最快适应算法，最优适应算法，道格拉斯算法，Half-Fit 算法，TLSF 算法的释放操作均为常数时间级。这些算法都将释放掉的内存范围添加至一个追踪空闲内存块的数据结构中。如果进行紧凑操作，紧凑内存算法的两种实现版本的释放操作均为线性级，其依赖于释放的内存对象的大小。内存释放操作均为可控的。紧凑操作的执行带来了额外的时间开销，但是两种紧凑内存系统都提供了一个完全可预测的内存空间。


10000
1000
100
10
1
0
200
400
600
800
1000
1200
1400
1600
1800
首次适应
最优适应
道格拉斯
Half-fit2
TLSF
移动版本
非移动版本

第六章 结论与展望

6.1 本文结论

在本文中我们首先提出了实时内存管理系统的定义[22]，然后介绍了一种适用于实时操作系统的内存管理方案，紧凑内存管理系统。我们在第二章中描绘了目前存在的显式非实时内存分配算法，它们为：首次适应算法，最佳适应算法以及道格拉斯分配算法。然后我们讨论了显式的嵌入式实时系统的内存分配算法 Half-fit 以及 TLSF 算法。以上这些算法都不能提供完全可以预测的内存使用情况，因此并不十分适合硬实时系统。我们还更进一步的讨论了垃圾回收机制的内存管理系统如 Treadmill 算法，jamaica 算法和 Metronome 算法。

第三章主要展示了紧凑内存管理系统的系统模型，我们分别介绍了抽象和物理地址空间的概念。另外还介绍了用于限定内存碎片范围的尺寸类的概念。

在第四章中我们详细介绍了紧凑内存管理系统的两种实现方式，分配为可移动内存及不可移动内存的两种实现方案。内存对象的分配操作在可移动版本中的时间复杂度为常数时间级，在不可移动版本中为线性时间级（受内存对象大小的影响）。内存对象的释放操作在两种实现方案中均为线性时间级。通过在可移动版本中使用部分紧凑的方法，我们可以将其释放操作的时间复杂度降低至常数时间内。两种实现方案的解引用操作均可以在常数时间内完成。由于我们可以保证每个（部分）尺寸类时刻保持紧凑状态，因此我们可以提供完全可以预测的内存使用状况。相对于其他现存的显式实时内存管理系统而言，我们的紧凑内存管理系统是完全适合嵌入式实时系统，甚至是安全相关的应用。最后，嵌入式实时系统的另外一条性质为初始化耗费常数时间，我们通过对所有需要初始化的资源（抽象地址空间，页面，块帧等）使用一个先进的空闲链表来达到此种要求。

在第五章中我们进行了性能的测试和基准测试程序的数值，它们的结果证实了我们提供的渐进复杂度。由于内存紧凑操作需要更多的管理操纵，我们系统的响应时间微微慢于现存的嵌入式实时系统。这是我们提供的完全可预测的内存状态的代价。

6.2 进一步工作展望

我们还需要进一步的对我们的设计与实现进行几个可能的优化。在我们的当前设计中，抽象地址空间是静态分配以满足最坏情况的。为了进一步减少内存额外开销，我们可以通过将物理空间中的页面同样使用于抽象地址空间的方法来实现动态的抽象地址空间，这个想法已经在 4.8.3 节中提出。

在当前版本中大于 16KB 的内存对象是不被支持的。我们希望在进一步的工作中通过使用 arraylets 的概念来实现大于 16KB 内存对象的支持。

另外一个需要完成的工作是实现一个程序分析小工具来分析最佳的内存部分紧凑阈值。使紧凑内存管理系统在可以保证内存可预测的基础上能够尽可能的提高效率，以降低响应时间。此外还需要进一步严谨验证过程，对 Linux 内存的动态监测技术[21]进行进一步研究。

参考文献

[1] A. Burns and A. Welling, Real-Time System and Programming Languages. Second Edition. Published by Addison-Weslay, 1996

[2] Hermann Kopetz. Event-triggered versus time-triggered real-time systems. In Proceedings of the International Workshop on Operating Systems of the 90s and Be-yond, pages 87–101, London, UK, 1991. Springer-Verlag.

[3] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic stor-age allocation: A survey and critical review. InIWMM ’95: Proceedings of the In-ternational Workshop on Memory Management, pages 1–116, London, UK, 1995.

[4] Miguel Masmano, Ismael Ripoll, and Alfons Crespo. A comparison of memory allo-cators for real-time applications. InJTRES ’06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems, pages 68–76, New York, NY, USA, 2006. ACM Press.

[5] Isabelle Puaut. Real-time performance of dynamic memory allocation algorithms. In ECRTS ’02: Proceedings of the 14th Euromicro Conference on Real-Time Systems, page 41, Washington, DC, USA, 2002. IEEE Computer Society.

[6] Donald E. Knuth. Fundamental Algorithms, volume 1 of The Art of Computer Pro-gramming. Addison-Wesley, Reading, Massachusetts, second edition, 10 January 1973.

[7] Andrew S. Tanenbaum. Modern operating systems. Prentice-Hall, Inc., Upper Sad-dle River, NJ, USA, 1992.

[8] Doug Lea. A memory allocator. Unix/Mail/, 6/96, 1996.

[9] M. Masmano, I. Ripoll, A. Crespo, and J. Real. Tlsf: A new dynamic memory al-locator for real-time systems. In ECRTS ’04: Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS’04), pages 79–86, Washington, DC, USA, 2004. IEEE Computer Society.

[10] Henry G. Baker. The treadmill: real-time garbage collection without motion sickness. SIGPLAN Not., 27(3):66–70, 1992.

[11] Paul R. Wilson and Mark S. Johnstone. Real-time non-copying garbage collection. In ACM OOPSLA Workshop on Memory Management and Garbage Collection. ACM Press, 1993.

[12] Tian F. Lim, Przemyslaw Pardyak, and Brian N. Bershad. A memory-efficient real-time non-copying garbage collector. In ISMM ’98: Proceedings of the 1st interna-tional symposium on Memory management, pages 118–129, New York, NY, USA,1998. ACM Press.

[13] David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. InProceed-ings of the Conference on Languages, Compilers, and Tools for Embedded Systems, pages 81–92, San Diego, California, 06 2003.

[14] Fridtjof Siebert. Hard real-time garbage-collection in the jamaica virtual machine.In RTCSA ’99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications, page 96, Washington, DC, USA, 1999. IEEE Computer Society.

[15] Tobias Ritzau. Memory Efficient Hard Real-Time Garbage Collection. Dissertation, Department of Computer and Information Science Linkoping University, SE-581 83 Linkoping, Sweden, 2003.

[16] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems.Commun. ACM, 12(11):611–612, 1969.

[17] Pradeep Padala. Playing with ptrace, part i. Linux J., 2002(103):5, 2002.

[18] Pradeep Padala. Playing with ptrace, part ii. Linux J., 2002(104):4, 2002

[19] Benjamin Zorn and Dirk Grunwald. Evaluating models of memory allocation. ACM Trans. Model. Comput. Simul., 4(1):107–131, 1994.

[20] I. Puaut. Real-time performance of dynamic memory allocation algorithms, 2002.

[21] 何杭军, 朱利. 基于 Linux 的动态内存检测工具的设计与实现[J]. 计算机工程, 2005, 31(21): 69-71。

[22] 徐蓉. 实时系统的内存管理技术研究与实现[D]. 电子科技大学,2004

[23] 胡滨;孙健力.一种内存管理技术的研究与实现[J].计算机工程与设计,2007 年 05 期

致 谢

在本论文即将完成之际，我很庆幸自己能够在三年的学习生活中遇到了许许多多的良师益友，他们不论在我的学习上还是我在生活中都给予了很大的关怀和帮助，让我在学习上、思想上都有了很大成长和进步。很难用言语来表达此刻我对大家的感激之情，谨以最简单的言语表达我最真诚的谢意！

首先，由衷的感谢我的导师于哲舟教授，他严肃的科学态度，严谨的治学精神，精益求精的工作作风，都无时无刻的不在深深地影响着我。他广泛而深入的信息安全实践也使我的视野得到了非常大的拓展。研究生学习生活期间，恩师在学业上给予了精心的指导，使我在学术上收获了丰厚的知识，受益匪浅，同时老师还在生活中给予了我无微不至的照顾。导师勤恳的工作精神、学术上严谨的态度、渊博的知识储备都令我无比钦佩。在此，向我的导师致以最崇高的敬意和最诚挚的感谢！

感谢各位审稿人，感谢你们在如此繁忙的工作任务中辛苦的审稿，祝你们事业顺利、身体健康！