中山大学硕士学位论文

Linux 桌面环境下内存去重技术的研究与实现

Research and Implementation of Memory Deduplication in The Linux Desktop Environment

学位申请人：　周霄
导师姓名及职称：　王常吉副教授
专业名称：　软件工程
院、系（所）：　软件学院

答辩委员会主席：_____

答辩委员会成员：_____

_____

_____

二零一三年五月

论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，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：

日期：　年　月　日

学位论文使用授权声明

本人完全了解中山大学有关保留、使用学位论文的规定，即：学校有权保留学位论文并向国家主管部门或其指定机构送交论文的电子版和纸质版；有权将学位论文用于非赢利目的的少量复制并允许论文进入学校图书馆、院系资料室被查阅；有权将学位论文的内容编入有关数据库进行检索；可以采用复印、缩印或其他方法保存学位论文；可以为存在馆际合作关系的兄弟高校用户提供文献传递服务和交换服务。

保密论文保密期满后，适用本声明。

学位论文作者签名：　　导师签名：

日期：　年　月　日　　日期：　年　月　日

论文题目： Linux 桌面环境下内存去重技术的研究与实现

专业：软件工程
硕士生：周霄
指导教师：王常吉

摘 要

内存管理是操作系统中至关重要的部分。优秀的操作系统高效地管理有限的内存资源。Linux 内核的内核同页合并（KSM）模块是在虚拟化兴起后为节省虚拟机使用的内存而发展的一种节约内存的技术。KSM 通过合并相同内容页面的方式显著地降低了多个基于内核虚拟机（KVM）技术的虚拟机同时运行时系统使用的内存。实际上，KSM 也能合并一般应用程序中的相同内容页面。但一般应用程序为了使用 KSM 技术合并内存，需要在源代码级别显式地调用相应的系统调用来告知 KSM 需要扫描的内存区域。对于一般用户来说，需要修改应用程序的源代码显然是一个太高的门槛。

本论文在认真研究了 KSM 模块的实现的基础上，增强了 KSM 的功能。新设计实现的 KSM+使得用户可以不必修改应用程序的源代码就能利用 KSM+合并相同内容的内存页面。另外原来的 KSM 算法依赖程序指定的内存区域有大量的重复内存页面，而实际上，一般应用程序并没有大量的重复内存。在重复内存非常少的情况下，对这些程序应用 KSM 技术非但不能降低内存使用，而且因为运行 KSM 本身所需内存而增加了内存开销。本论文针对这一点在 KSM 的算法上做了改进，减少了运行 KSM+算法本身所消耗的内存。然后论文用 Linux 桌面下的常用应用程序验证了 KSM+的实际效果，并构造了一个实验比较了 KSM 和 KSM+。最后，为了了解应用程序的重复内存页面的特点，还编写了专门统计重复页面的内核模块，发现应用程序的重复页面的一些规律，为持续改进 KSM+提供了依据。

关键词：KSM，重复内存合并 ，内存管理

Title: Research and Implementation of Memory Deduplication in The Linux Desktop Environment

Major: Software Engineering

Name: Zhou Xiao

Supervisor: Wang Changji

Abstract

Memory management is one of the most important parts of the operating system. KSM (Kernel Samepage Merging) in Linux kernel is a kind of memory saving technology developed after the emerging of virtual machine. KSM can dramatically decrease the memory usage of the hypervisor running several virtual machines. Actually, KSM can also be applied to normal applications. But in order to use the KSM, application must explicitly evoke a system call in source code level to tell KSM the memory area where the KSM will scan. To normal users, modifying the source code is impossible at most of the time.

Base on the full grasp of the implementation of KSM, a new implementation named KSM+ is created, which allows users to merge same-content pages on the specified applications without modifying corresponding source code. Moreover, the original KSM algorithm relies heavily on the specified area is rich in same-content pages, while normal applications have much less same-content pages compared to virtual machines. So, when KSM is applied to those applications, it is possible that memory usage will rise rather than decrease. To combat with this situation, KSM+ employs a new algorithm to decrease the memory usage for running itself. Several experiments prove that the KSM+ can be easily applied to specified applications and memory usage can be reduced. And a case is designed to compare the effect between

KSM and KSM+, which shows KSM+ behaves better in deduplication when the same-content pages' density is very low. At last, in order to inspect the characteristics of same-content pages from normal desktop applications, an ad-hoc kernel module is developed to do the statistics which supplies significant data for the further development of KSM+.

Key Words： KSM, Deduplication, Memory Management

目　录

第一章 绪论

1.1 课题背景

内存是操作系统中最为重要的资源之一。为了最有效地利用好有限的内存资源，现代操作系统使用了很多技术以节省内存的使用。在 Linux 程序的内存布局中，代码段具有只读属性，永远是共享的。具体地说，即使一个程序被实例化了多个进程，但是在物理内存中对于该程序的代码只有一份拷贝，同一个程序的不同进程之间共享这份拷贝。而且，所有共享库（Windows 下被称之为动态链接库）的代码段在物理内存中也只有一份拷贝[1]，所有加载了该共享库的进程虽然把共享库加载到了不同的虚拟地址中，但实际上都映射到相同的物理地址上以实现对共享库代码区的共享。除了共享只读的代码区外，可写的页面也可能在进程之间被共享，Linux 在创建进程时的 fork 操作使用写时拷贝技术在父子进程之间共享内存。当父进程调用 fork()函数后，所有的可写页面转成写时拷贝页面。只要父子进程都不写入某页面，则该页面一直被共享。

近年来，Linux 操作系统中出现了一种新的节省内存的技术，它的出现源于云计算的发展。云计算能够通过互联网按需分发计算机的软硬件资源，被称之为 IT 产业的又一次革命[2]。虚拟机技术是云计算的基础，Xen 和 KVM(Kernel Virtual Machine，内核虚拟机)是开源社区最流行的两种虚拟化技术[3]。通过运用虚拟化技术，越来越多的 Linux 主机上同时运行着多个虚拟机，而虚拟机使用了相同的操作系统，操作系统中运行着相同的大型程序，如数据库服务器，邮件服务器，Web 服务器等。在运行多个虚拟机的环境下，人们发现虚拟机之间存在着大量的相同内存页面，但 Linux 主机没有办法共享这些内存页面。为了解决虚拟机之间大量重复内存的问题，Linux 社区诞生了 KSM(Kernel Samepage Merging，内核相同页面合并)技术。KSM 技术通过扫描内存区域，找到相同的内存页面，并把它们合并。该技术使得运行虚拟机的主机使用的内存明显下降，从而让同样的 Linux 主机在使用 KSM 技术后能运行更多的虚拟机。

KSM 技术虽然是为基于 KVM 技术的虚拟机设计，但也能运用到普通的桌面应用程序。随着软硬件技术的不断进步，Linux 桌面应用程序占用的内存也越来越多。普通的 Linux 桌面应用程序中存在多少重复页面，重复页面的分布有些什么特点，把 KSM 技术运用在桌面应用程序上存在什么问题，这些是本论文将要探讨的内容。针对桌面应

用程序的重复内存，本文在 KSM 的基础上开发了 KSM+，能方便有效地合并桌面应用程序的重复内存。

1.2 国内外研究现状

对于传统的节约内存技术，如共享只读代码区，fork 的写时拷贝技术。这些技术非常成熟，现代操作系统中已经普通采用。

另外，文献[4]研究了一种比较特殊的节省内存技术。它针对的是科学计算中的特殊程序，通过用注释驱动的工具把源代码转译到更高效的代码，于是编译出节省内存的程序。

近年来研究较多的是基于合并相同内存页面的内存去重技术：通过查找系统中的相同内存页面，把这些页面合并从而节省系统内存。但国内外的研究基本集中在运行虚拟机的场景中。如 VMware 公司较早进行了虚拟机内存去重技术的研究工作，文献[5]研究了 VMware 公司的 Hypervisor(虚拟机管理器)使用的基于合并相同内容内存页面的去重技术，以减少虚拟机之间的内存冗余。文献[6]详细论述了此类技术对于基于 Xen 的虚拟机的实现。Linux 社区的类似技术称之为 KSM。文献[7]以实证研究的方式通过大量实验研究了 KSM 用于合并虚拟机之间冗余内存的性能。文献[8]研究的是在数据中心中如何提高在虚拟机之间合并相同内存的效率。它实现了通过把虚拟机在各物理机中动态迁移到最适合的位置从而提高合并效率。文献[9]使用 KSM 技术解决了主机和虚拟机都需要维护自己的缓存的问题，提高了缓存的效率。KSM 技术并不是总能高效地去除虚拟机之间的冗余内存，也存在一些问题。内存地址的 ASLR(Address Space Layout Randomization，空间布局随机化技术)是为了提高程序安全性的一种技术。该技术提高了程序的安全性，却影响了页面对齐，从而影响了 KSM 的合并效率[10]。另外，KSM 的运用也暴露了安全问题。运行在某台虚拟机上的程序可以通过构造特定的页面尝试让主机的 KSM 合并它，再计算写该页面所用的时间。因为写被 KSM 合并的页面的时候会发生写时拷贝，所以这个时间会比正常情况下延迟。通过利用这个时间差，运行在一台虚拟机上的程序可能会发现其它虚拟机上运行的程序[11]，甚至让虚拟机之间秘密通信[12]。

KSM 技术应用在非虚拟机的场景下的研究则比较少。欧洲核子研究组织(CERN)的核子实验模拟程序产生海量的数据，并且并行工作的科学程序同时载入这些数据，使

得内存中有大量的相同内容页面。CERN 通过使用钩子技术修改 C 运行库的 malloc()从而运行 KSM，解决了内存中的大量重复页面的问题，节省了 8%-48%的内存[13]。

本论文将在一般桌面应用环境下研究并增强新的内存节省技术 KSM，使之更适合运用在桌面应用环境下，更有效地减少应用程序的内存使用量。

1.3 研究意义

内存资源总是操作系统中稀缺的资源，研究节省内存的技术具有普遍的意义。另外，随着基于 Linux 内核的操作系统如 Android 在消费电子终端上的迅速普及[14]，以及可预见到基于 Linux 的 Chrome OS 在 PC 桌面市场也将占有一定市场，而目前的应用程序普遍占用越来越多的内存，研究 Linux 平台下新的节省内存的技术 KSM，对于开发 Android 平台等基于 Linux 内核的平台下的商用优化管理软件具有重要的技术指导作用。

1.4 论文的主要工作

本论文重点研究了 Linux 内核的 KSM 模块的设计与实现，并在原来的基础上做了针对桌面应用环境的增强与改进，同时认真研究了 Linux 内核代码的组织方式，配置与编译方式，内核开发及调试的技术，以及 Linux 的内存管理单元的实现。增强后的 KSM 命名为 KSM+,它比原来的 KSM 更方便地自动采集到应用程序的相同内容页面并合并这些页面，以达到节省应用程序消耗的内存的目的。论文用多个实验展示了 KSM+的功能，并与修改前的 KSM 做了比较。此外，本论文还编写了内核模块用于统计 Linux 桌面应用程序的重复内存页面的特点，同时可以用来应证 KSM+运行的效果。

1.5 论文的整体结构布局

本论文的第二章介绍了 x86 体系的段页式内存管理方式，以及 Linux 的内存管理的实现，还介绍了编写 Linux 内核代码的一些基本理论。

第三章介绍了如何使用 KSM 以及 KSM 的设计和实现原理。

第四章讲述 KSM+的设计与实现。KSM+是在 KSM 的基础上的增强与改进。

第五章用多个实验展示了 KSM+合并程序的重复内存页面的效果，并设计了一个实验比较了 KSM 和 KSM+的差别。最后编写了用于统计应用程序重复内存页面的内核

模块，并用统计数据应证了 KSM+的实验结果。这些统计数据反映了 Linux 桌面应用程序中重复内存页面的一些基本特点，对进一步改进 KSM+的算法提供了依据。

1.6 本章小结

本章首先介绍了 Linux 操作系统中传统的节省内存技术，然后引入了因为虚拟化兴起而发展的新的内存节省技术，它通过合并内容相同的内存页面从而减少系统的内存使用，也被称之为内存去重技术，接着叙述了国内外在该技术方面的相关研究。目前对该技术的研究多针对虚拟机环境，而本论文则是在 Linux 桌面环境中研究内存去重技术。

第二章 相关理论与技术

2.1 内存管理

内存管理是操作系统内核中极为重要的部分，它涉及物理地址、虚拟地址、分段、分页等诸多概念。

2.1.1 内存地址

程序使用地址来访问内存单元。对于 x86 架构的中央处理器，使用逻辑地址，虚拟地址，物理地址三种不同的地址[15]。

逻辑地址包括段和偏移地址两部分，它们共同用来指定机器指令或者操作数在内存中的位置。逻辑地址对应下文提到的分段式内存管理。

虚拟地址也被称作线性地址，它表示一个寻址空间。在 Linux 操作系统中，使用一个无符号长整型存储虚拟地址。如果是针对 32 位 x86 处理器的内核，这是一个 32 位的无符号长整型，可以寻址至多 2^{32}，即 4GB 空间。而 Linux 在 x86_64 处理器上，一个无符号长整型有 64 位，但实际上只使用了 48 位，即可寻址到 2^{48} ，即 256TB 的内存空间。引入虚拟地址使得用户空间的进程都独自拥有一个地址空间，方便操作系统在硬件的协助下隔离进程访问内存的操作，从而实现更高的安全性和便捷性。在 32 位处理器架构下，4G 虚拟地址空间划分成两部分，高端的 1G 空间被称之为内核空间，低端的 3G 空间称之为用户空间。内核的代码和数据存在内核空间，用户空间的进程通过系统调用陷入内核空间执行内核空间的代码。对于 64 位处理器，虚拟地址同样分为内核空间和用户空间，只是针对 64 位处理器的内核拥有大得多的寻址范围，对内核空间和用户空间的地址划分有多种实现。

物理地址被中央处理器用来寻址内存芯片上的实际的物理单元。它对应了送往处理器的针脚上的高低电平信号。

Linux 中的内存管理单元 (Memory Management Unit, MMU)在硬件电路的帮助下把逻辑地址译成线性地址；而另一个硬件电路则帮助把线性地址翻译成物理地址。

2.1.2 段页式内存管理

内存分段是由 80x86 处理器引入的概念，它鼓励程序员在设计程序时把程序使用的内存按照逻辑上的意义相关性分成若干个部分，比如有些段存放全局数据，有些段存放程序代码[16]。

内存管理单元的分页单元协助把虚拟地址译为物理地址，为了提高翻译的效率，虚拟地址被按照固定长度切分成一个个小单元，它们则被称之为页面。页面可以看成是 MMU 管理内存的最小单位。

分段和分页都用来划分内存，在某种程度上，它们的功能有一定重合。实际上，Linux 内核的内存管理机制倾向于使用分页技术。Linux 进程运行在用户空间时使用的标志代码段的代码段选择子(segmentation selector)和数据段选择子是相等的；进程运行在内核空间时使用的代码段选择子和数据段选择子也是相等的。而且这四个段选择子的段基址都设置为 0。这意味着在 Linux 操作系统中，无论寻址数据和还是寻址指令，无论在用户空间还是内核空间，逻辑地址的段基址部分为 0，偏移地址部分刚好等于虚拟地址。

在 Linux 所支持的大多数硬件平台上，默认的页面大小是 4K。但 Linux 同时也支持大于 4K 的页面的内存管理。运行在用户空间的代码可以使用 glibc 库中的 getpagesize()函数得到当前使用的页面大小。本论文只针对在 x86_64 架构上页面大小是 4K 的 Linux 内核。实际上从内核 2.6.38 开始，内核中开始支持透明巨页(Transparent Huge Page)[17]，内核中有些页面的大小实际上是 2M，但这不影响按照 4K 的页面大小展开讨论。

Linux 使用分级页表的方式把虚拟地址映射到页面。在 32 位处理器上使用二级页表，在 x86_64 架构的处理器上则使用四级页表。

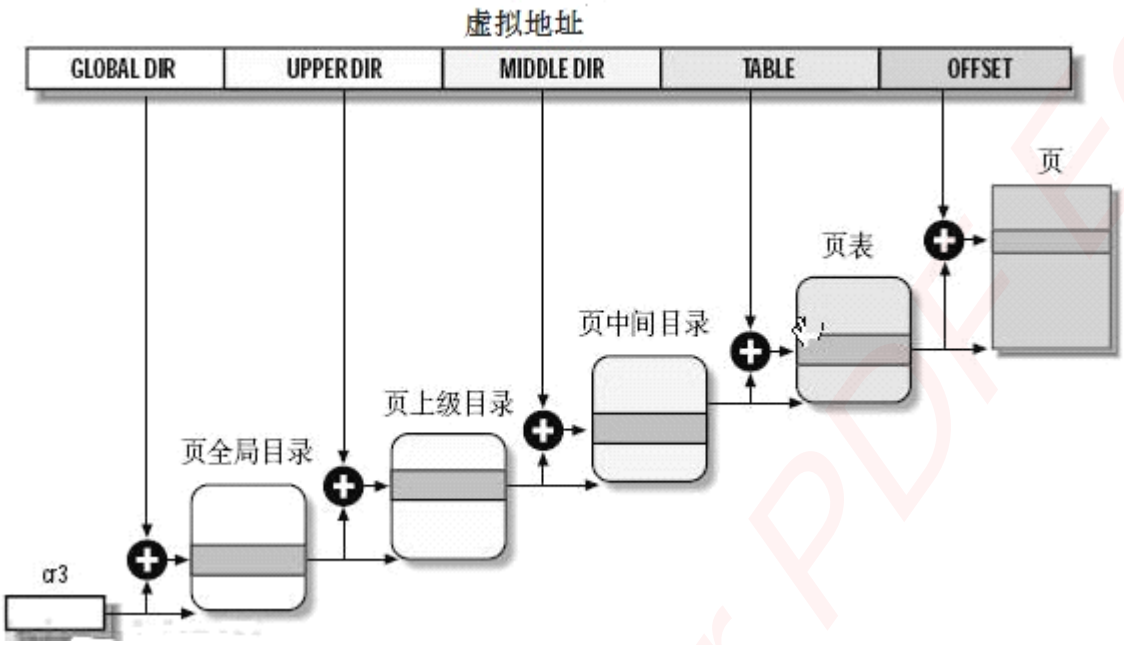


图 2-1 四级页表

如图 2-1 所示，以针对 x86_64 架构处理器的 Linux 内核为例，64 位虚拟地址的高 16 位不被使用，低 48 位被平均分为 5 段，offset 段占 12 位，其它四个段各占 9 位。这个划分在源代码文件 arch/x86/include/asm/pgtable_64_types.h 中定义。cr3 寄存器存储了页全局目录的物理地址。在把虚拟地址翻译到物理地址的过程中，MMU 先从 cr3 寄存器中取出全局目录的物理地址，再加上 GLOBALDIR 段算出的偏移量，从而在页全局目录中得到页上级目录的物理地址。与此类似，从页上级目录中得到页中间目录的地址，然后得到页表的地址，再得到页面的地址，最后的 OFFSET 字段则定位到页面中的一个字节。至此，虚拟地址被翻译到了物理地址。

页全局目录，页上级目录，页中间目录，以及页表中的每一项都由相同的数据结构表示。它记载了一些重要的信息，与本论文相关的信息有：该项指向的下一级表或者页面是否存在的标志，如果访问时不存在则产生缺页中断，由 do_page_fault 函数负责把页面调入内存[18]；下一级表或页面的物理地址；下一级表或页面是否可读可写的标志，利用该标志位可以写保护一个页面。

Linux 内核用一个 struct page 结构体描述一个页面的信息。这个结构体定义在 include/linux/mm_types.h 文件中[19]。图 2-2 列出这个结构体中与本论文直接相关的字段。

```
struct page {
        unsigned long flags;
        atomic _count;
        atomic _mapcount;
        struct address_space *mapping;
        void *virtual;
        ...
};
```

图 2-2 struct page 的部分字段

其中的 flags 字段存储了页面的状态信息，例如：表示页面刚被写了数据的脏位；该页面是否被锁定在内存中不充许置换到交换分区的标志。_count 字段和_mapcount 字段都是引用计数，它们用来共同维护 page 页面的生命期。_mapcount 表示一个页面拥有多少页表项指向它，_count 被称为 page 的使用计数，所有的_mapcount 计数只相当于_count 计数中的一次计数。如果内核代码中某执行序列在访问某个页面时需要确保该页面存在，则在访问前给_count 计数加一，访问结束后_count 计数减一。当_count 计数减到负数时表示没有任何内核需要使用该页面，则表示该页面没被使用。内核代码不应该直接访问_count 计数，而应该使用 page_count 函数。该函数用一个 struct page 的指针做为参数，当该页空闲时函数返回 0，否则返回一个正数表示参数指向的页面正被使用。当页面被页高速缓冲使用时，mapping 域指向一个 address_space 对象，该对象与 Linux 文件系统中的文件是一对一映射的关系，描述了一个文件分配到的物理页面及相关数据结构。如果页面并不被页高速缓冲使用时，mapping 则有其它的意义，比如 mapping 字段的最低位记载这是否是一个匿名页面。virtual 字段是映射到该页面的内核空间的虚拟地址，如果该页面是属于高端内存区域则 virtual 字段为 NULL。对于 x86_64 架构上的 Linux 内核，已经没有高端内存，因此该字段不会为 NULL。

2.1.3 内核中的巨页

现代处理器的内存管理单元几乎都能处理除 4KB 大小外的多种页面尺寸。然而，Linux 内核在几乎所以平台上的实现都选用最小的页面大小，即 4KB，比 4KB 更大的页面，都被称之为巨页(huge page)。在某些工作环境下，巨页可以给操作系统带来性能上的提高。巨页能提高操作系统的性能主要是因为两点。第一，使用巨页可以减少发

生页面出错处理的频率，因为在每次页面出错处理时内核调入比使用小页面更多的内存。第二，使用巨页减少翻译虚拟地址的时间。在 x86_64 加构的处理器上，从虚拟地址翻译到物理地理需要依次访问四级页表，非常耗时。而使用巨页可以减少页表的级数。实际上更大的性能提高源自使用巨页后提高了旁路转换缓冲区(Translation Lookaside Buffer，TLB)的命中效率[20]。

在内核 2.6.38 版本之前，唯一使用巨页的方式是通过非常复杂的 hugetlbfs 文件系统。应用程序开发者和系统管理员都需要注意一些特别事项才能够开启巨页功能。因此只有极少数真正需要巨页带来性能提高的用户，如专用数据库系统，才使用巨页。

情况从内核版本 2.6.38 以后发生了变化，一个被称之为透明巨页的功能被合并进了内核主干代码。之前内核在 VMA（Virtual Memory Area，虚拟内存区域）中的所有页面大小都是一样大的，而加入透明巨页后，VMA 中的页面的大小可能不只一种。透明巨页的实现代码改写了页面出错处理函数，当一个出错发生时，内核尽力分配一个巨页，如果成功的话，其它相应的在巨页地址范围内的小页面就会被释放，巨页被插入到 VMA 中。如果不能分配到一个巨页，则仍然按照以前的方式分配一个小页面。当巨页需要被置换到交换分区时，透明巨页机制简单地把巨页切割成小页面，其它逻辑和处理小页面时一样。实际上不仅在置换到交换分区时需要切割巨页，很多其它的操作，如 mprotect()和 mlock()页面时也需要。本论文将详述的 KSM 在合并页面时同样需要把巨页切割成小页面。

透明巨页这种轻便地利用巨页的方式使得很多内核代码并没有感知到巨页的存在，对于应用程序则更没有感知到巨页，因此被称之为透明巨页。

2.1.4 内存描述符

Linux 内核使用一个叫内存描述符(Memory Descriptor)的结构体描述每一个进程的地址空间[21]。这个结构体包括了所有和进程地址空间相关的信息。内存描述符和 struct page 一样在文件 include/linux/mm_types.h 中定义。图 2-3 列出这个结构体中与本论文直接相关的字段。

```
struct mm_struct {
        struct vm_area_struct *mmap;
        struct rb_root mm_rb;
        atomic_t mm_users;
        atomic_t mm_count;
        struct rw_semaphore mmap_sem;
        struct list_head mmlist;
        unsigned long flags;
        ...
};
```

图 2-3 struct mm_struct 的部分字段

其中，mmap 字段是用来实现一个链表。这个链表链接了属于这个内存描述符的所有 vm_area_struct 结构体。vm_area_struct 结构体描述了一个内存区域。由于属于一个内存描述符的内存区域可能非常多，为了加快内存区域的查找以及添加删除等操作的速度，内核用 mm_rb 表示一棵链接了所有内存区域的红黑树。红黑树是一种二叉树，它被广泛地应用在 Linux 内核代码中。mmap 和 mm_rb 是用两种不同的数据结构表示同一批数据。mm_users 和 mm_count 是内存描述符的引用计数，它的实现原理和 struct page 的引用计数的原理一样。每一个进程如果拥有一个内存描述符，则会增加 mm_users 的计数，所有 mm_users 的计数只相当于 mm_count 的一个计数。比如 n 个 Linux 线程共享同一个内存描述符，那么对应的内存描述符的 mm_users 计数则为 n，而 mm_count 则可能只是 1。如果有内核执行序列想要访问一个内存描述符，则该执行序列先增加 mm_count 的计数，使用结束后减少 mm_count 的计数。一但 mm_count 减为 0，表示该内存描述符没有任何引用，则它会被内核销毁。mmap_sem 是一个读写锁，凡是需要操作内存描述符中的内存区域时，则需要先得到相应的读锁或者写锁，使用结束后释放该锁。mm_list 字段是一个循环双链表。它链接了系统中所有的内存描述符。flags 字段定义了内存描述符的标志，这些标志标记了内存描述符的一些状态和属性，内核代码需要使用原子操作访问该字段。

2.1.5 虚拟内存区域

同样被定义在 inlude/linux/mm_types.h 中的结构体 vm_area_struct 用来描述一个内存区域。在 Linux 内核中，内存区域经常被称之为虚拟内存区域(Virtual Memory Areas,VMA)。VMA 描述的内存区域是一个地址空间中的地址连续的部分。而且该部分

拥有一些特定的属性，比如访问权限和相关联的操作等。不同的 VMA 可以用来描述不同的内存区域。比如，映射到内存的文件，用户空间的栈，特别是本论文重点关注的匿名内存区域(Anonymous VMA)。下面讨论本论文关心的 vm_area_struct 的一些重要字段，这些字段如图 2-4 所示。

```
struct vm_area_struct {
        struct mm_struct *vm_mm;
        unsigned long vm_start;
        unsigned long vm_end;
        struct vm_area_struct *vm_next;
        struct rb_node vm_rb;
        unsigned long vm_flags;
        struct anon_vma *anon_vma;
        struct file *vm_file;
        ...
};
```

图 2-4 struct vm_area_struct 的部分字段

vm_mm 字段指向该 VMA 属于的内存描述符。vm_start 和 vm_end 字段表示该 VMA 描述的内存区域的起始和终点虚拟地址，但不包括 vm_end 指向的地址，即 vm_end 是虚拟内存区域的最后一个有效字节的后一个字节。vm_next 和 vm_rb 分别把内存描述符拥有的内存区域用链表，和红黑树链接起来。vm_flags 表示内存区域的属性。anon_vma 与内核的页面回收机制回收匿名页时用到的页面反射表有关。vm_file 是该内存区域对应的文件，如果内存区域是匿名的，则该字段被置为 NULL。

2.1.6 Slab 层

频繁地申请和回收同一类型的数据结构是内核中极为常见的操作。为了提高申请和回收某种特定数据结构的效率。Linux 内核中引入了 slab 层的概念，通过 slab 层来管理某种数据结构的频繁申请和回收[22]。

Slab 层把不同的数据结构划分到不同的高速缓存(cache)中，每个缓存都用来维护某种数据结构的申请的回收。比如，从维护进程描述符的缓存中申请进程描述符时，缓存把一个标记为可用的进程描述符返回给调用者，同时把该描述符标记为已用；释放进程描述符则使得这个进程描述符在存放它的缓存中又被标记为可用。

每个缓存被划分为好几个 slab，这就是这个子系统被称之为 slab 层的原因。每个 slab 是一页或几页连续的内存。缓存维护的某种数据结构，称之为对象，就全部放在 slab 中。每个 slab 处于满，部分满或空三种状态。当 slab 处于满状态时，表示该 slab 中维护的对象已经全部被分配。当 slab 处于空状态时，和半满状态时，slab 可以有空闲的对象可供分配。当内核需要申请一个新对象时，就向对应的缓存申请。缓存找到自己维护的 slab，从 slab 中拿出预先分配好的空闲对象返回给申请者。当内核使用完了该对象，这个对象就被缓存回收，一般情况下并不真正释放，而是在对应的 slab 上做空闲标记。缓存通过灵活的策略维护 slab 的大小和数量。使用 slab，能够加速常用数据结构的分配和释放，并减少分配内存产生的碎片。

2.2 内核开发

操作系统内核的开发和普通用户空间的程序开发有非常大的差异。本节将介绍理解 KSM 和在内核空间进行开发所需熟知的一些内容。

2.2.1 Linux 内核简介

Linux 由 1991 诞生开始就受到开源社区的热烈欢迎，并且一直处于极为活跃的持续开发状态。Linux 的代码以 GNU 计划倡导的 GPL 协议自由分发，全世界的操作系统爱好者，硬件驱动编写者都非常活跃地为 Linux 贡献代码。Linux 的开发如此成功，以至 GNU 计划中自己的操作系统内核 GNU Hurd 都未得到足够多的关注，Linux 成为事实上 GNU 计划的操作系统内核。

Linux 内核沿用 Unix 的单内核设计。单内核设计把整个内核设计成运行在同一个地址空间的一个“大”进程。所有的内核执行序列存在于同一个地址空间中，并在同一个地址空间内运行。因此，内核的不同执行序列之间的通信是极为快速的，基本不需要额外的开销，但由此也带来内核代码的复杂度过高的问题。较之单内核，另一个设计操作系统的思路是微内核，它的很多设计理念和单内核的设计相反，如 windows XP, Vista 还有 windows 7 等等。Linux 内核在发展过程中引入了内核模块的概念，这使得 Linux 内核既保持了单内核设计性能快速的特点，又使得内核代码相对纯粹的单内核简洁和易于维护。

本论文选定的内核版本是 2013 年 2 月发布的 3.0.66 稳定版，在它的基础上研究开发。

2.2.2 内核线程

Linux 的内核线程实际上是只存在于内核空间的一个进程。内核通常创建内核线程让它在后台周期性的处理一些事务。内核线程和普通进程一样可调度，可被抢先。他们的最显著的区别是内核线程的进程描述结构体 task_struct 的 mm 字段为 NULL。而一般进程的进程描述结构体的 mm 字段指向该进程的地址空间。因为内核线程永远只运行在内核态，永远不必切换至用户空间，并且所有用户态进程的地址空间的内核虚拟地址部分都是一样的，所以当处理器调度到内核进程时，内核进程可以随便使用某个用户态进程的地址空间的内核虚拟地址部分。Linux 内核线程的作法是借用上一个普通用户态进程的用户空间[23]。

内核线程由内核 API 函数 kthread_create()创建，也可由 kthread_run()创建。他们的区别是前者创建的是一个处于非运行状态的内核线程，需要使用 wake_up_process()把它转换为可运行状态；而 kthread_run()创建的内核线程立即处于可运行状态，随时可能被调度而获得运行的机会。内核线程开始后会一直运行，直到它显式地调用 do_exit()或者其它内核代码调用 kthread_stop()。kthread_stop()函数需要传入先前创建内核线程函数返回的 task_struct 作为参数。该函数在调用后会一直阻塞，直到等待的内核线程完全退出了才返回。

2.2.3 内核的同步机制

竞争条件指的是程序设计中的一种缺陷，这种缺陷使得程序的输出会因为不受控制的事件出现的顺序或发生时间而发生改变。Linux 内核提供了一套同步方法，正确地使用同步方法可以消除竞争条件[24]。

atomic_t 是内核中的特殊整型数，应用在该整型数上的方法使得对该整数的操作是原子操作，从而对该整数的操作不会被其它执行序列中断。如 atomic_inc()和 atomic_dec()分别是原子地给一个整型数加一或减一。

Linux 内核中最常用的锁是自旋锁。对于一个自旋锁，内核在任何时候只能有一个执行序列持有该锁。如果在该自旋锁被其它执行序列持有时，当前执行序列也申请持

有该锁，则这个执行序列就会使得处理器一直忙等该锁被持有者释放。为遵循迅速占有，迅速释放的原则，一个内核执行序列一但持有某自旋锁，则内核在该内核序列运行的当前处理器上被禁止抢先。另外，持有自旋锁的内核执行序列应该运行完简短的代码后迅速释放该自旋锁，而不应该在释放持有的自旋锁前执行可能造成睡眠的代码，也不能显式地调用放弃处理器而申请调度其它进程的代码，因为这样会使得其它申请该自旋锁的执行序列等待非常长的时间而大大损害了整个系统的性能。自旋锁的基本有法是：用 DEFINE_SPINLOCK 在编译期定义一个未被占用的自旋锁变量，用函数 spin_lock()得到自旋锁，用 spin_unlock()函数释放自旋锁。与普通自旋锁类似的还有读写自旋锁。当对自旋锁保护的资源的访问多数是读访问，少部分是写访问时，使用读写自旋锁比使用普通自旋锁的效率更高。

Linux 内核中另一种常见的锁是信号量。信号量只能在进程上下文中才能使用，当一个进程或内核线程尝试获取一个不可用的信号量时，该进程或者该内核线程会被置于一个等待队列中，然后进入睡眠状态，处理器转而执行其它的代码。当该信号量被某持有者释放，信号量变为可用时，等待队队中的一个进程或内核线程会被唤醒而获得该信号量。因为未能立即获得锁时会使得执行序列进入睡眠，所以在不能被调度的中断上下文中不能使用信号量。同样因为未能立即获得信号量则被置于睡眠状态的原因，信号量非常适合需要长时间持有锁的情况。信号量的另一个显著的特点是它充许同时有若干个持有者，这个数量在声明信号量时指点。但大多数情况下信号量被设置为只能有一个持有者，这种情况下，信号量被称为二值信号量或者互斥锁。互斥锁常见的使用方法是：用 DECLARE_MUTEX 宏在编译期定义一个未被占用的互斥锁变量，或者用 init_MUTEX 在运行时初始化互斥锁，用 down()函数获得信号量，用 up()函数释放信号量。与读写自旋锁类似，Linux 内核也有针对读多写少的情况做优化的读写信号量，它的特征和普通信号量基本相同。

2.2.4 内核的调试技术

调试内核代码是内核开发技术中非常困难的一部分，这是因为内核代码运行在软件栈的最底层，不像用户空间的进程可以轻易地被调试器跟踪。另外，因为很多情况下的内核错误会让整个操作系统崩溃或者运行在不可预测的不稳定状态，让开发者几

乎没有机会收集到足够多的错误是如何出现的线索，所以内核错误在很多情况下也不能轻易地被重现。

用 printk()函数打印出程序的相关信息到日志文件或控制台是最古老也最常用且有效的调试方法[25]。printk()函数与用户空间的 printf()库函数相似，其中的差别之一是 printk()函数可以让用户指定打印信息的级别。目前 printk()函数有八个信息级别，它们被定义在 linux/kernel.h 头文件中。级别的数字越小则表示级别越高。基于信息级别，内核可能把信息打印到当前的控制台。它可能是一个字符终端，可能是一个串口连接的设备，也可能是一个并口打印机。内核中的整型变量 console_loglevel 表示能被发往控制台的信息的最小级别。可以用 sys_syslog 系统调用改变它的值，用可以通过写文件/proc/sys/kernel/printk 改变它。所有信息，包括信息级别的值大于 console_loglevel 而没能被发往控制台的信息都被添加到文件/var/log/messages 的末尾。

除了最为常用的打印函数 printk()是调试内核必备之外。内核开发者使用的内核在编译前通常都会打开一些方便调试的内核选项[26]。比如 CONFIG_KALLSYMS 选项使得内核代码的符号信息被编译进内核，这样当内核崩溃时，系统会用符号打印出出错时栈回溯信息，否则只有二进制的地址信息。还有一些特定的编译选项帮助开发者排查特定的错误，比如 CONFIG_DEBUG_SLAB 可以帮助排查忘记初始化内存和内存越界访问的错误；而 CONFIG_DEBUG_SPINLOCK 可以排查非法使用自旋锁的错误。

KGDB 是常用的调试 Linux 内核的调试器。最开始，它只是作为一个内核补丁发布，从内核 2.6.26 开始，它被合并到了内核的主干代码。用 KGDB 调试时，需要两台用串口连接的 Linux 机器，被调试的机器运行 KGDB，另一台(调试机)则运行普通的 GDB。GDB 远程协议被用于它们之间的通讯[27]。配置 KGDB 比较麻烦，但目前桌面虚拟化技术非常成熟，开发者可以用两台 Linux 虚拟机使用 KGDB，用虚拟的串口硬件连接两台机器。仅管如此，使用内核调试器还是相当之麻烦。被调试的内核在编译后输出的 vmlinux 文件是未压缩的 linux 内核，需要被拷贝到调试机；对应的源代码需要被拷贝到调试机调试时的当前目录；另外，调试时并不能在所有代码语句设置断点，因为内核代码的编译依赖于 GCC 的编译优化选项，开启相应的优化选项内核代码才能通

过编译，而很多语句的变量经过 GCC 优化后会被去掉，调试者从源代码上看到的执行路径经常并非内核实际的执行路径。

2.3 本章小结

本章首先介绍了 Linux 内核的内存管理的一些理论和技术，然后介绍了进行内核开发所必须掌握的基础知识。本章的内容是理解以后章节的基础。

第三章 KSM 的设计与实现

3.1 KSM 简介

KVM 是 Linux 内核主干代码中的虚拟机解决方案[28]。当使用了 KVM 虚拟化技术的 Linux 主机同时运行多个操作系统时。主机的内存里存在着大量的相同内容的页面，而内核的内存管理不能发现它们，不能够共享这些相同内容的页面。KSM 的出现就是为了解决这种大量冗余内存的问题。KSM 是内核共享内存(Kernel Shared Memory)或者内存相同页面合并(Kernel Samepage Merging)的简称，是随着 Linux 内核的虚拟化技术的发展而产生的一种节省内存的技术。KSM 通过扫描运行虚拟机的内存找到内容相同页面并将他们合并为一页，从而节省了内存，于是可以运行更多的虚拟机。更准确地讲，KSM 扫描并合并相同的匿名页面。而基于 KVM 虚拟运行操作系统时用 malloc()分配内存，malloc()分配到的内存是 KSM 可以合并的匿名区域。

KSM 最开始以内核补丁的方式发布，但因为一些原因未能进入内核的主干代码。首先，Linux 社区的开发者们担心 KSM 的实现代码中存在安全漏洞。KSM 的最初算法通过比较两个页面内容的 SHA1 哈希值来判断两个页面是否相同。如果攻击者通过哈希碰撞的方式攻击内核，那么攻击者有可能把恶意数据或代码注入到系统中[29]。另外，社区开发者认为 VMWare 公司持有的一个算法专利和 KSM 的最初实现非常类似，Linux 开发者认为在内核中包含 KSM 可能会给 Linux 招致法律方面的不必要麻烦。

后来，KSM 改变了实现的方法，并顺利通过了代码审阅进入了 Linux 内核的主干代码，成为了 KVM 虚拟机技术的一个重要的内存优化解决方案[30]。

3.2 KSM 的使用

KSM 的使用包括如何配置 KSM，如何理解 KSM 的一些参数，以及如何通知 KSM 合并程序的重复内存。

3.2.1 通过 sysfs 使用 KSM

KSM 功能在内核配置文件.config 中由开关 CONFIG_KSM=y 控制是否编译进内核。KSM 目前的实现不能以模块的方法编译，也就是 CONFIG_KSM 不能置为值 m。如果 KSM 被编译进内核，内核在启动后会创建 ksmd 内核线程。

sysfs 是 Linux 内核的一个特殊的文件系统，通过它可以实现内核空间和用户空间的交互。管理和配置 KSM 也是通过 sysfs 进行，位置在/sys/kernel/mm/ksm 目录下[31]。在该目录下可以找到一系列文件，有一些文件用来控制 KSM，有一些文件用来监视 KSM 的运行状态。该目录下的文件：run，用来开启和停用 KSM。在默认配置下，KSM 是被停用的，如果写入值 1 到文件 run，则会开启 KSM。例如：echo 1 > /sys/kernel/mm/ksm/run。在 run=1 的状态下，再写入值 0 到 run 文件，则会停用 KSM。此外，在 run=1 的状态下，还可以写值 2 到 run 文件，要求把合并的所有页面再拆分出来。

当 KSM 在运行的时候，可以通过三个参数配置 KSM 的运行状态。其中，文件 sleep_millisecs 指明了内核线程 ksmd 在执行完每次扫描之后需要至少睡眠多长的时间。文件 pages_to_scan 定义了每次扫描需要扫描多少个页面。每个用户都可以查看该文件，但只有有系统权限的用户可以更改 pages_to_scan 的值。共有 5 个文件用来显示 KSM 的运行情况，这 5 个文件都是只读的。文件 full_scans 指示 KSM 已经完成了多少次全面扫描。文件 pages_shared 指明了 KSM 中的稳定树上已收集了多少个 KSM 页面。文件 pages_sharing 指明了节省了多少个页面的内存。文件 pages_unshared 则显示不断被扫描但又未能合并的页面数。文件 pages_volatile 显示因为页面内容变化太快而不被合并的页面数。用文件 pages_sharing 的值比上 pages_shared 的值，如果这个值越大，说明 KSM 运行的效率越高，反之则运行效率越低。

3.2.2 KSM 的编程接口

虽然 KSM 可以扫描并合并系统中相同的匿名页面，但在这个扫描和合并过程中，KSM 需要消耗处理器资源，同时也需要消耗一部分内存。如果 KSM 扫描的大量内存区域并没有足够多的内容相同的匿名页面，那么 KSM 非但没有降低内存的使用量，相反

还会因为支撑自己的运行而消耗更多的内存。KSM 的解决办法是把寻找具有大量相同内容内存的责任转给程序编写者。

Linux 的系统调用 madvise[32]用来建议内核如何处理从虚拟地址 addr 开始，长 length 个字节的内存区域。该系统调用的函数原型为：int madvise(void *addr, size_t length, int advice)。KSM 假设程序员知道哪些区域包含大量的可合并内存。于是，从内核版本 2.6.32 开始 KSM 在系统调用 madvise 中新加入了两个参数：MADV_MERGEABLE 和 MADV_UNMERGEABLE。应用程序代码中通过用参数 MADV_MERGEABLE 调用函数 madvise 建议 KSM 扫描哪些内存区域，KSM 则把相应的内存区域 VMA 加上 VM_MERGEABLE 标志位；用 MADV_UNMERGEABLE 取消之前的建议，相应的 VMA 确保没有 VM_MERGEABLE 标志位。分析 KSM 的源代码可知，实际上 KSM 找到指定区域所在的 VMA，并把该 VMA 置 VM_MERGEABLE 标志，之后将扫描整个 VMA 的内存区域，并不严格遵守 madvise 设定的起始范围。

3.3 KSM 的设计与实现

本节将介绍设计和实现 KSM 的主要思路，特别介绍了最为关键的稳定树和不稳定树两个数据结构，并对 KSM 的效率作简要的分析。

3.3.1 KSM 的基本算法

KSM 算法的核心思想是通过把需要扫描的页面插入两棵红黑树之一来查找到内容相同的页面。红黑树用页面的内容作为树的索引，用函数 memcmp()逐字节的比较两个页面来判断页面是否相等。如果两页面内容不等，memcmp()函数返回小于 0 的值或者大于 0 的值决定下一步往红黑树的左边搜索还是往右边搜索。这两棵红黑树其中一棵称为稳定树(如图 3-1 所示)，另一棵称为不稳定树。稳定树用来存放已经合并的页面，被称之为 KSM 页面。不稳定树用来筛选合适的候选合并页面[33]。

KSM的主要算法流程如图3-1所示。对于每一个被扫描的匿名页面，KSM先尝试在稳定树中查找是否有页面内容与被扫描页面的内容一致。如果在稳定树中找到了这样的匹配，被扫描的页面则被合并到匹配到的KSM页面。如果在稳定树中未能找到匹配项，KSM计算被扫描的匿名页面的校验和。如果校验和与上次KSM扫描时计算的值不一样，那么KSM更新这个页面的校验和，推迟以该页面内容为索引搜索不稳定树。这

样的好处是避免了把一个更新频率很快的页面放入不稳定树。如果检验和与上次扫描时没有发生变化，KSM在不稳定树中开始搜索匹配页面。如果在不稳定树中找到相应

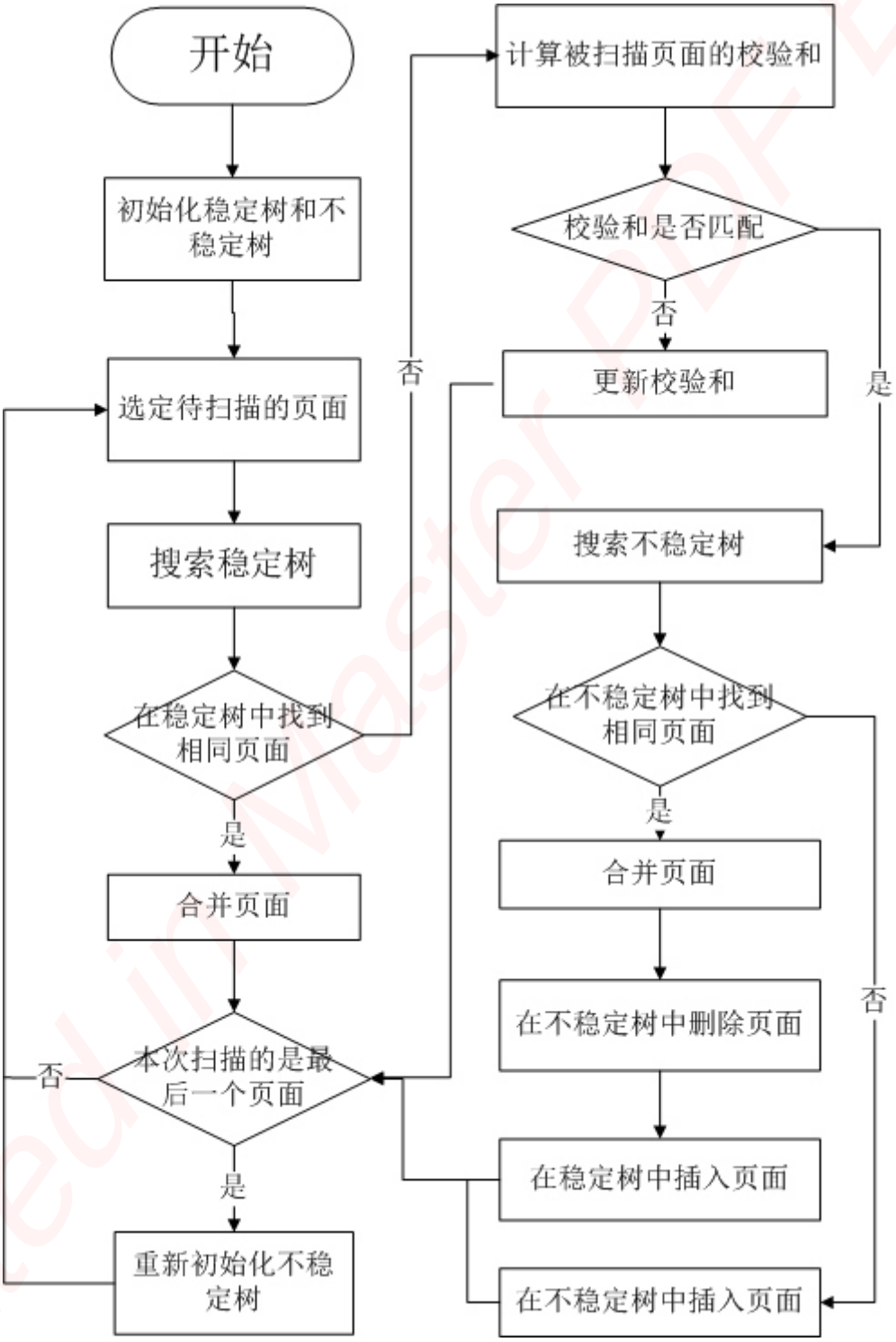


图 3-1 KSM 算法流程图

的匹配项，KSM 合并匹配页面和当前被扫描的页面，并把匹配页面从不稳定树中删除。合并后生成一个 KSM 页面被插入稳定树，并把 KSM 页面写保护。如果未能在不稳定树中找到匹配项，KSM 把被扫描的页面插入不稳定树，该页面不被写保护。这里检查页面的校验和实际上对整个 KSM 算法只起来辅助的作用。校验和并不作为索引搜索不稳定树，而是用来筛选一个足够稳定的页面用来合并。如果一个页面更新的太快，则失去了合并的意义，相反会影响不稳定树的效率。如果一个经常变化的页面被合并到了稳定树，则可预见合并后立即发生一个写时拷贝的页面错误，又要把该变化的页面重新分出来，浪费了处理器资源。在整个 KSM 算法中，如果去掉检查校验和这一步骤，也可以正常运行，但会降低 KSM 的效率。

3.3.2 稳定树

红黑树是一种平衡二叉查找树，在Linux内核代码中有广泛的应用。鲁道夫·贝尔于1972年发明了红黑树，他当时把它称之为"对称二叉B树"[34]，而红黑树是现代的名字。红黑树的每个节点都必须带有颜色属性，或红色或黑色。除此之外，红黑树还必须满足其它一些特性：根节点具有黑色属性，每个叶节点也是黑色属性，每个红色属性的节点的两个子节点都是黑色属性(即从每个叶子到根的所有路径上不可能出现两个连续的红色节点)，从任一个节点到其每个叶子的所有路径都包含相同数目的黑色节点。对红黑树的查找，插入和删除都可以在O(log n)时间内完成，其中的n是树中元素的数目。

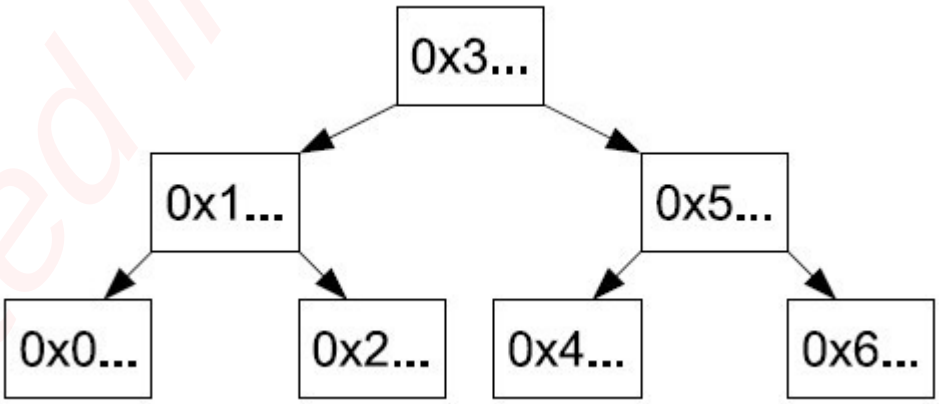


图 3-2 稳定树

稳定树是一棵红黑树。被扫描的页面在不稳定树中找到匹配页面后，它们被合并，产生的KSM页面就进入稳定树。稳定树的每个结点都存放了已经被共享的页面。为了保证作为树索引的页面内容不被修改，页面被写保护。如果内核中某执行序列写入存于稳定树中的KSM页面，就会产生一个写时拷贝的错误。执行写操作的程序得到被写保护页面的一份拷贝。稳定树如图3-2所示：

3.3.3 不稳定树

KSM中的不稳定树也是一棵红黑树，但与内核中其它绝大多数应用红黑树的数据结构不同，不稳定树的索引即页面内容。而页面内容随时可能会发生变化，因此该树会因此部分不可用。但KSM不能像对待稳定树那样写保护不稳定树中的页面。这是因为如果不稳定树中的页面被写保护的话将会导致大量的写时拷贝，致使消耗大量的处理器资源处理页面错误。好消息是向红黑树插入或删除数据的方法rb_insert()和rb_erase()并不需要查询红黑树上的结点的索引值就能够让树保持平衡。一个仍然处于完全可工作状态的不稳定树如图3-3如示。此时在树顶的的页面的索引的第一个字节是3，如果这个索引的第一个字节又被程序修改为7，则如图3-4如示，那么在这棵树中查找第一个字节为4，5，或者6的页面时将得不到正确结果。为了应对这样一种不利情况，KSM的解决方案是在每次扫描完所有注册的内存区域后，将不稳定树设置为空树，即重建不稳定树。这样，因为树结点的索引变化而造成的负面效果不会持续影响新的一次扫描。另一种应对不稳定树部分不可用的解决方案是，KSM可以在每次查询不稳定树的过程中检查经过的树结点的校验和，如果检验和与上一次的旧校验和不一致就把该结点从不稳定树中删除，但这样做会比仅仅用memcmp()函数比较两个页面的内容要慢得多，因此KSM采取重建不稳定树的策略。

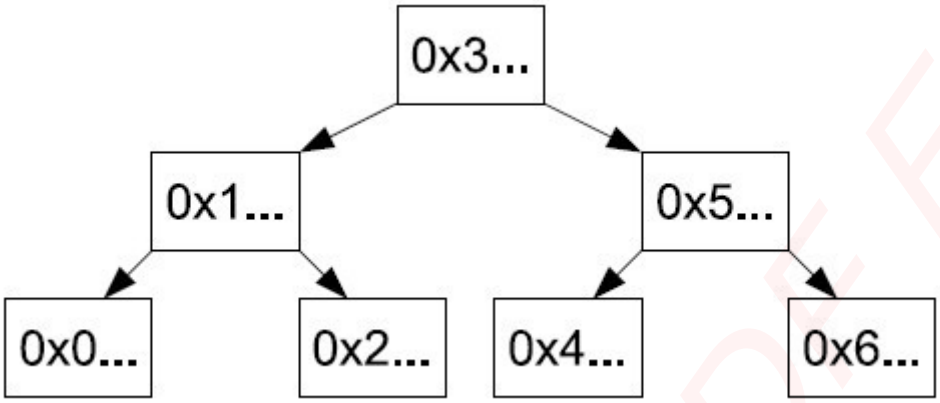


图3-3 完全可用的不稳定树

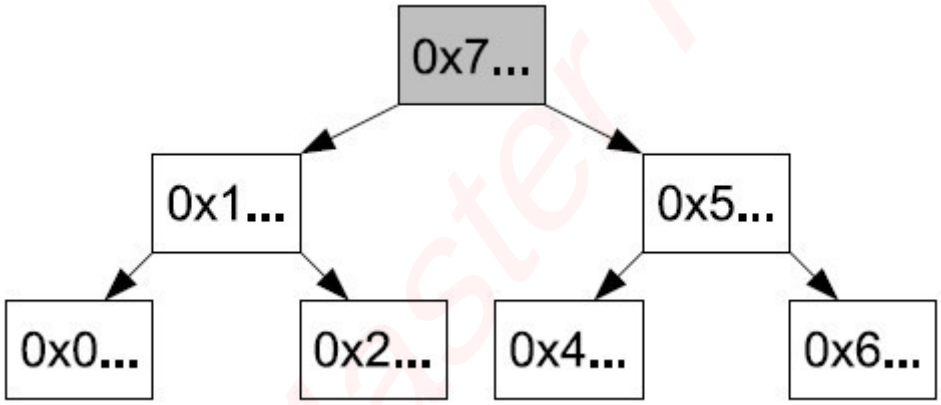


图3-4 失去部分可用性的不稳定树

3.3.4 KSM 的效率

Linux内核在x86架构上的实现中，和其它大多数平台的实现中，一个页面的大小通常是4096个字节。当用memcmp()函数比较两个页面时，最坏的情况下要比较4096个字节，平均情况下的效率则快得多，这是因为在大多数情况下如果两个页面不一致，在比较页面内容的前几个字节时就能使memcmp()函数退出了。在一棵红黑树中查找到内容相同的页面的复杂度是用memcmp()函数的开销乘以红黑树的高度。由于红黑树的良好性能，它的插入、搜索、删除的时间复杂志是O(log(n))，其中n表示KSM扫描的页面数。所以，即使在最坏的极端情况下，memcmp()函数每次都要比较到最后一个字节才退出。KSM算法的性能也不会非常差。

3.4 关键代码分析

本节将在源代码层面介绍 KSM 的主要数据结构，以及合并重复页面时的关键函数。

3.4.1 KSM 的主要数据结构

KSM中主要的数据结构有mm_slot，ksm_scan，stable_node，rmap_item。对应每一个被扫描的内存描述符mm_struct，KSM都有一个mm_slot描述了一些相关信息。它的定义如图3-5所示：

```
struct mm_slot {
        struct hlist_node link;
        struct list_head mm_list;
        struct rmap_item *rmap_list;
        struct mm_struct *mm;
};
```

图 3-5 mm_slot 的定义

其中，link 字段用来链接被哈希到相同哈希表位置的 mm_slot，mm_list 字段链接 KSM 算法中所有的 mm_slot，rmap_list 字段用来链接属于该 mm_slot 的所有 rmap_item，mm 字段是该 mm_slot 对应的内存描述符。

ksm_scan 记载了当前正扫描到的位置。它的定义如图 3-6 所示：

```
struct ksm_scan {
        struct mm_slot *mm_slot;
        unsigned long address;
        struct rmap_item **rmap_list;
        unsigned long seqnr;
};
```

图 3-6 ksm_scan 的定义

其中，mm_slot 字段表示当前正在扫描的 mm_slot 结构体；address 字段表示当前正在扫描到的虚拟地址；rmap_list 是个双指针，它指向一个指向 rmap_item 的指

针，表示在 rmap_list 中下一个将被扫描的 rmap_item；seqnr 记下当前扫描次数的计数。

stable_node 表示 KSM 算法中稳定树的结点。它的定义如图 3-7 所示：

```
struct stable_node {
        struct rb_node node;
        struct hlist_head hlist;
        unsigned long kpfn;
};
```

图 3-7 stable_node 的定义

其中，node 字段表示在稳定树中的哪个结点；hlist 链接了所有使用该结点表示的 KSM 页面的 rmap_items；kpfn 字段则是物理页框的编号。

rmap_item 是 KSM 算法中最重要的数据结构。如果把从虚拟地址到页面的映射称之为正映射，那么 rmap_item 是从页面信息到虚拟地址的反向映射。它的定义如图 3-8 所示：

```
struct rmap_item {
        struct rmap_item *rmap_list;
        struct anon_vma *anon_vma;
        struct mm_struct *mm;
        unsigned long address;
        unsigned int oldchecksum;
        union {
                struct rb_node node;
                struct {
                        struct stable_node *head;
                        struct hlist_node hlist;
                };
        };
};
```

图 3-8 rmap_item 的定义

其中，rmap_list 用来链接上文指到的 mm_slot 结构体中属于某内存描述符的所有 rmap_item 链成的单链表；anon_vma 是与内核中回收匿名页面的算法相关的数据结构，与 KSM 算法关系不大；mm 表示该反向映射 rmap_item 所属的内存描述符；

address 表示虚拟地址；oldchecksum 用来存储上一次计算的校验和；node 字段用在当 rmap_item 存于不稳定树中时表示在不稳定树中的结点；head 字段指向稳定树中的结点；hlist 字段用来链接所有共享稳定树中某结点的 KSM 页面的 rmap_items。

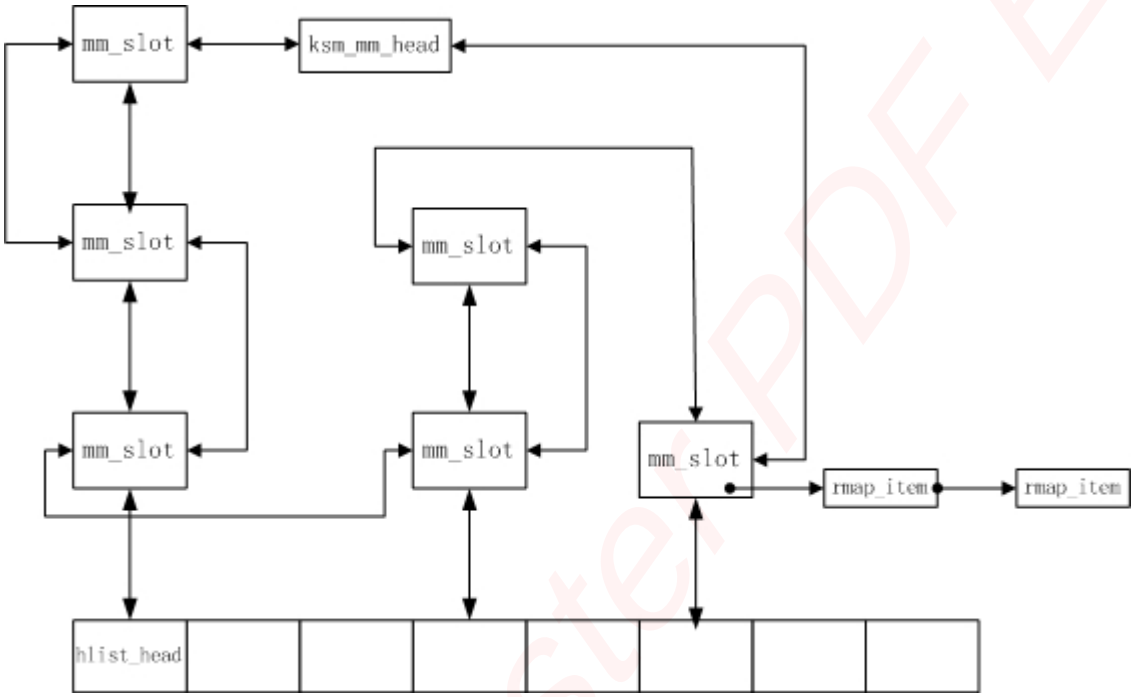


图 3-9 mm_slot，rmap_item，mm_slots_hash

如图 3-9 所示，mm_slots_hash 是一个哈希数组，每个数组元素存放了一个哈希链表的头结点。所有的 mm_slot 在计算哈希值后链接到哈希数组的相应的哈希链表上。同时，所有的 mm_slot 用双链表结点两两链接成一个环形链表。对于每个 mm_slot，它还有所属的 rmap_item 链接在 mm_slot 上。

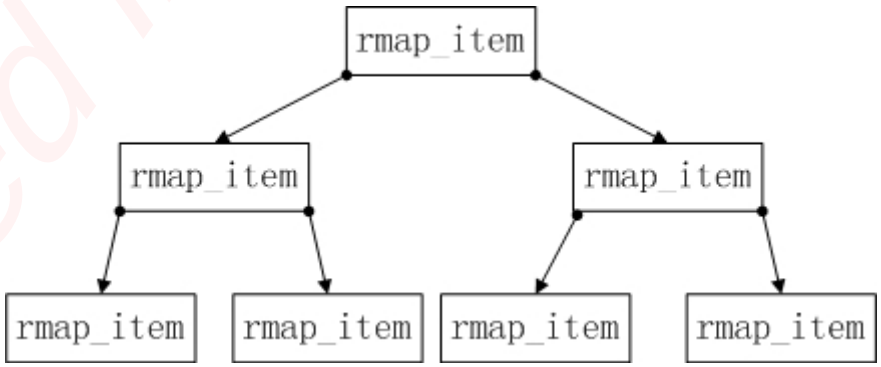


图 3-10 rmap_item 链接在不稳定树上

如图 3-10 所示，不稳定树的树结点嵌入在 rmap_item 结构体中。每个不稳定树的树结点对应的页面由 rmap_item 表示。

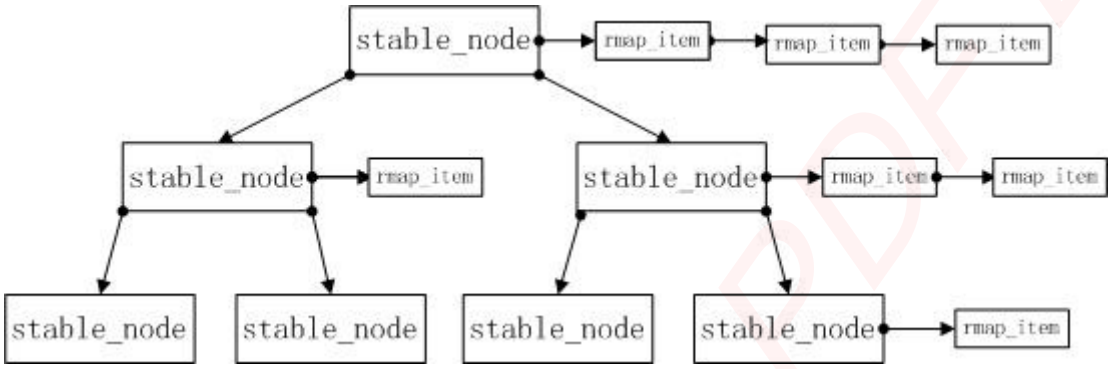


图 3-11 rmap_item 链接在稳定树上

如图 3-11 所示，每个稳定树的树结点维护了一个链表。链表链接了使用该结点对应的 KSM 页面的所有 rmap_item。

3.4.2 合并页面

```
1         ptep = page_check_address(page, mm, addr, &ptl, 0);
2         if (!ptep)
3                 goto out;
4
5         if (pte_write(*ptep) || pte_dirty(*ptep)) {
6                 pte_t entry;
7
8                 swapped = PageSwapCache(page);
9                 flush_cache_page(vma, addr, page_to_pfn(page));
10                entry = ptep_clear_flush(vma, addr, ptep);
11                if (page_mapcount(page) + 1 + swapped != page_count(page)) {
12                        set_pte_at(mm, addr, ptep, entry);
13                        goto out_unlock;
14                }
15                if (pte_dirty(entry))
16                        set_page_dirty(page);
17                entry = pte_mkclean(pte_wrprotect(entry));
18                set_pte_at_notify(mm, addr, ptep, entry);
19        }
20        *orig_pte = *ptep;
21        err = 0;
22
23 out_unlock:
24        pte_unmap_unlock(ptep, ptl);
25 out:
26        return err;
```

图3-12 write_protect()函数关键代码

KSM算法中涉及合并页面的两个最关键的函数是write_protect_page()和replcace_page()。其中前一个函数传入指定页面做为参数，然后该函数把指向指定页面的页表项的写标志位清掉。第二个函数replace_page()有两个参数分别指向两个页面，该函数若执行成功则把指向其中一个页面的页表项改为指向第二个页面，这样便完成了合并页面的功能。

图3-12所示为write_protect()函数的关键部分代码。代码1-3行调用函数page_check_address()，该函数检查page指向的页面是否在内存描述符mm中被映射到虚拟地址addr。如果该函数成功，则返回在内存描述符mm中指向页面page的页表项的指针，并且被自旋锁锁定。代码行5检查页表项上有没有可写标志或者有脏位。如果有可写标志，那么在接下的逻辑中就把可写标志擦掉，完成写保护；如果页表项上有脏位则需要清掉脏位后再写保护。第8行代码检查该页面是否被置换到了交换分区。有一种情况有内存中的页面被置换到了交换分区，但内存中的页面并没有被立即释放；另一种情况是交换分区的页面被重新唤入了内存，而交换分区中的页面仍然有效。对

于这两种情况，一个页面在内存和交换分区中都拥有一份拷贝。代码行9清空处理器高速缓冲区和页面page有关的缓冲。代码行10清空指定的页表项并清空处理器中的旁路转换缓冲，该函数返回清空该页表项前的原来的页表项。代码行11-14检查当前是否有在这个页面上的I/O，如果是，则把以前的页表项设回，退出write_protect()函数，写保护失败。第15，16行代码检查旧页表项是否置脏位，如果是则把页page置脏位。第17,18行代码写保护新的页表项并清掉页表项上的脏位，然后把新的页表项写入页表。此时，write_protect()函数成功。

图 3-13 是 replace_page()函数的关键部分代码，它完成合并页面逻辑。该函数接受的其中两个参数表示两个页面，一个是 page，它的页表项在合并后会被改写，另一个是 kpage，在合并后，page 对应的页表项会和 kpage 指向相同的页面。第 1 行代码调用 pte_offset_map_lcok()得到在内存描述符 mm 中的虚拟地址为 addr 的对应页表项，并用自旋锁锁定。它的功能类似 page_check_address()函数。第 2-4 行检查待合并的两个页表项是否是同一页表项，如果是，则不需要合并。第 7 行增加 kpage 页面的引用计数。第 8 行维护了一个用于从页面到虚拟地址的逆映射的数据结构，与 KSM 算法无直接关联。第 10，11 行代码清空 page 页面对应的高速缓冲和旁路转换缓冲，与 write_protect()函数中的用法一样。第 12 行代码完成合并的动作，原先指向 page 的页表项也指向 kpage。第 14-17 行代码完成释放 page 的逻辑。第 19 行代码给刚被改写的页表项解锁。

```
1         ptep = pte_offset_map_lock(mm, pmd, addr, &ptl);
2         if (!pte_same(*ptep, orig_pte)) {
3                 pte_unmap_unlock(ptep, ptl);
4                 goto out;
5         }
6
7         get_page(kpage);
8         page_add_anon_rmap(kpage, vma, addr);
9
10        flush_cache_page(vma, addr, pte_pfn(*ptep));
11        ptep_clear_flush(vma, addr, ptep);
12        set_pte_at_notify(mm, addr, ptep, mk_pte(kpage, vma->vm_page_prot));
13
14        page_remove_rmap(page);
15        if (!page_mapped(page))
16                try_to_free_swap(page);
17        put_page(page);
18
19        pte_unmap_unlock(ptep, ptl);
20        err = 0;
21 out:
22        return err;
```

图3-13 replace_page()关键代码

3.5 本章小结

本章详细介绍了如何使用 Linux 内核的 KSM 模块，该模块的设计思路以及主要的数据结构，并分析了 KSM 中最核心的代码。

第四章 KSM+的设计与实现

4.1 KSM+的设计

KSM+是在 KSM 的基础上针对桌面应用程序的应用场景而设计开发的合并内存页面工具。本节介绍 KSM+的设计思路和最核心的算法。

4.1.1 KSM+的设计目标

KSM+主要针对桌面应用程序，而 KSM 主要针对基于 KVM 的虚拟机环境。这两种环境主要有两个区别：第一，KVM 在源代码级别通知 KSM 合并哪些内存，而桌面应用程序除非更改源代码后重新编译才能使用 KSM，或者使用钩子技术截获程序对内存的申请再作相应的与 KSM 的交互处理，但实现这种方案也不简单。第二，虚拟机之间有大量的重复内存页面，而较之虚拟机环境，一般的桌面应用没有大量的重复内存页面。因此，KSM 如果不作改动地运用在桌面场景下会存在两个问题。第一，应用程序通知 KSM 需要合并的内存区域非常困难。第二，KSM 原有的扫描算法对扫描过的每一个页面都生成一个 rmap_item 结构体，而桌面应用程序的重复页面比较少，那么 KSM 对大量不能合并成功的页面生成 rmap_item 结构体非但不能节省内存，相反还浪费了内存。因此，KSM+的设计目标亦为两点：第一，用户空间易于通知 KSM+需要扫描的内存区域。第二，KSM+的扫描算法既能在重复内存页面较少的内存中找到需要合并的页面，又能够最大限度地减少自身的内存消耗。

4.1.2 新的用户交互方式

KSM+达到第一个设计目标的解决方案相对简单，通过扩展 KSM 使用的 sysfs 文件系统，加入了让用户指定和撤销扫描某程序的功能。用户空间可以通过 sysfs 文件系统通知 KSM+需要扫描哪个程序的内存。

KSM+设计具体交互方式为：用户通过操作 sysfs 文件系统上位于位置 /sys/kernel/ksm+/processes_to_scan 的文件，通知 KSM+需要扫描和撤消扫描哪些程序。约定的操作方式是：写入字符串“+ app_name”，则会扫描程序名为 app_name

的所有进程；写入字符串“- app_name”则撤消扫描程序名为 app_name 的所有进程。比如，用户可以方便地使用 echo “+ app_name” > /sys/kernel/ksm+/processes_to_scan 通知 KSM 扫描 app_name 程序

4.1.3 新的扫描算法

为了达成 4.1.1 中提到的第二个目标，KSM+需要对 KSM 原有的扫描算法做较大的改动。KSM 的描述算法依赖指定的地址空间存在高密度的重复内容的页面。分析 KSM 的 scan_get_next_rmap_item()函数，可以看到 KSM 尽力为每一个页面分配一个 rmap_item 结构体，如果该页面不能合并，那么不仅没有减少内存使用，相反还增加了一个 rmap_item 结构体大小的内存开销。要解决这个问题，需要采样另外一种 rmap_item 的生成算法，对于合并不成功的 rmap_item，能够回收该 rmap_item。

考虑到不同的 VMA 的重复内存页面的密度基本上是不一样的，针对不同的 VMA，KSM+可能需要使用不同的策略生成 rmap_item。因此，KSM+使用的新扫描算法做的第一个改动就是为每个 VMA 设置一个数据结构。较之 KSM 使用的数据结构，KSM+在 mm_slot 和 rmap_list 之间引入了一个新的维护 VMA 相关数据的结构体 vma_node。原来的算法把 rmap_item 挂接到对应的 mm_slot，即 mm_slot，rmap_item 两级数据结构。加入 vma_node 结构体后，每个 mm_slot 维护一棵红黑树，树中链接了 vma_node，而 vma_node 维护了一个链表，链表中链接了 rmap_item，即 mm_slot，vma_node，rmap_item 三级数据结构。它们的关系如图 4-1 所示。

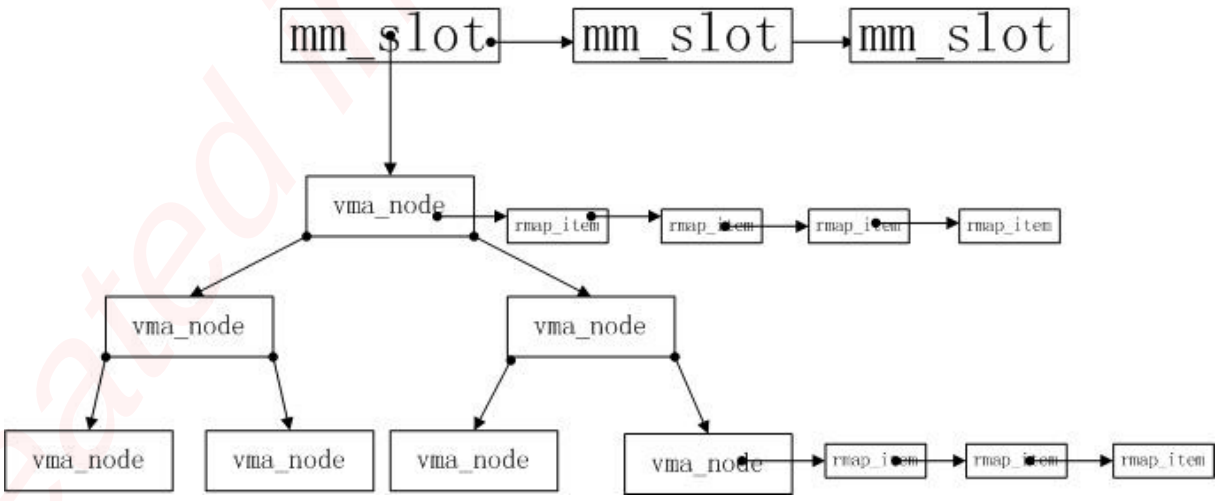


图 4-1 KSM+的三级数据结构

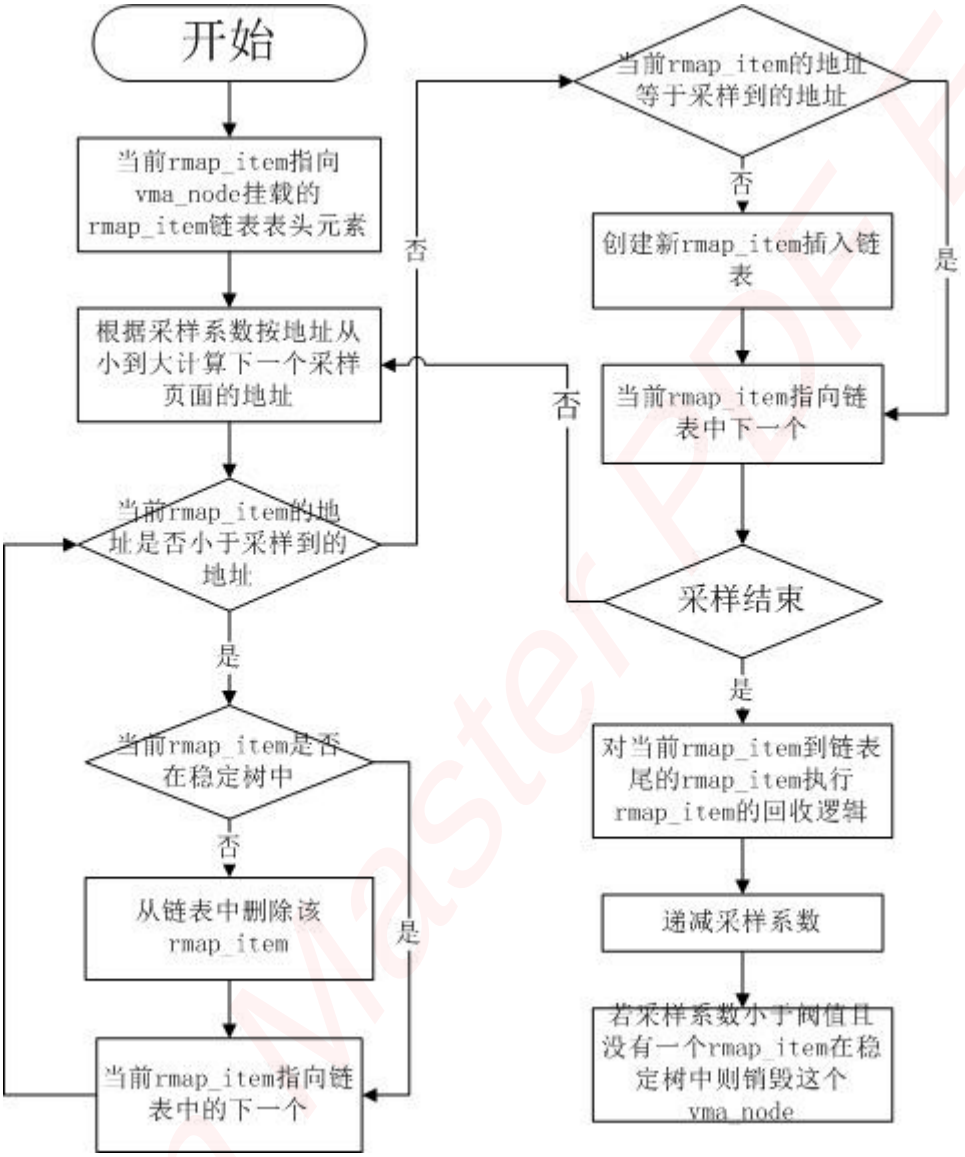


图 4-2 VMA 的采样算法

KSM+的扫描算法的主要部分在 vma_node_do_sampling()函数中。该函数对 VMA 进行采样，生成 rmap_item。该函数简化后的逻辑如图 4-2 所示，它主要做三件事情：第一，采样值得尝试进行合并的页面并生成 rmap_item 结构体；第二，销毁未能够合并成功的页面对应的 rmap_item，因为 KSM 在扫描重复页面并不多的虚拟内存区域时为生成的一些 rmap_item 本身消耗了一些内存，如果不回收这部分内存，会大大抵消 KSM 合并后节省的内存；第三，如果探测到基本上没有太大希望存在可合并内存

页的 VMA，则把对应的 vma_node 回收，节省了扫描这个 VMA 的所需要的处理器和内存资源。

函数 vma_node_do_sampling()中的采样逻辑部分使用了结构体 vma_node 的一个重要的变量，采样系数：coefficient。该系数取 1 到 100 之间的一个值 n，表示 vma_node 对应的 VMA 中有 n%的页面将被采样，如果 n 为 100 ，则 VMA 中所有的页面将被采样。采样逻辑根据采样系数计算出需要采样多少个页面 m，再把虚拟内存区域平均分成 m 个区间，然后在每个区间中随机取一个页面。KSM+设定初始采样系数是 100，即采样 VMA 中的所有页面，然后在每轮采样后给采样系数减少一定的值，这样每次采样就会比上次采样时选到的页面要少。因为前几轮的采样系数较高，所以大部分重复页面都会在前几轮采样中被选中，随后被合并。前几轮高密度的采样过后会进入低密度的采样，这样又能使一些内容有变化的页面有机会被选中进行合并操作。

函数 vma_node_do_sampling()中回收 rmap_item 的算法是：对于本轮采样到的 rmap_item，无条件加入 vma_node 维护的 rmap_item 链表；对于上轮采样中的 rmap_item，如果已经找到了可以合并的相同内容页面，即该 rmap_item 链接到了稳定树中，就保留该 rmap_item；如果既不在稳定树中，而且不被本轮采样命中，则销毁该 rmap_item。vma_node 维护的 rmap_item 链表的结点按照虚拟地址从小到大排序，销毁 rmap_item 的算法只需要遍历一次 rmap_item 链表，它的时间复杂度与链表的长度是线性的关系。

KSM+的扫描算法做的第三件事是销毁 vma_node 的逻辑。函数 vma_node_do_sampling()在每完成一轮采样后，判断本轮采样过后是否有 rmap_item 处在稳定树中，如果没有 rmap_item 在稳定树中且采样系数 coefficient 下降到了一定阀值，则把描述对应的虚拟内存空间的结构体 vm_area_struct 中的标志字段 vm_flags 置 VM_NOT_MERGEABLE 位。VM_NOT_MERGEABLE 标志位是本工程自定义的标志，表示某 vm_area_struct 将不被 KSM 采样。该标志被添加在内核源代码 /include/linux/mm.h 文件中。KSM 遍历所有 vma_node 的时候，代码将检查到 VM_NOT_MERGEABLE 标志位，于是把 vma_node 从 mm_slot 中删除。

4.1.4 KSM+的性能

从理论上分析，KSM+和 KSM 在比较两个页面是否相同时使用同样的函数，该函数的性能在 3.3.4 小节有较详细的分析。稳定树和不稳定树也仍然是最主要的数据结构，所以 KSM+对 KSM 的整体性能基本一致。考虑最主要的两大修改部分，新的用户交互方式只是用来通知 KSM+需要扫描的区域，对性能几乎没有影响。再考虑 KSM+采用的新的扫描算法，KSM 依次为需要扫描的内存区域中的每个页面生成 rmap_item，而 KSM+根据采样系数生成 rmap_item。当采样系数不为 100 时，KSM+生成的 rmap_item 要比 KSM 生成的 rmap_item 要少。而采样系数以某个数值为步长，从 100 递减，这意味着大多数情况下，扫描同样大小的内存区域，KSM+生成的 rmap_item 比 KSM 少，因此 KSM+的时间效率要优于 KSM。

KSM+对系统的性能影响与 KSM 一样，主要取决于两个参数，一个是两次扫描之间的时间间隔，另一个是每次扫描的页面数量。这两个参数实际上都可以通过 sysfs 文件系统进行调整，两次扫描之间的时间间隔越短，每次扫描的页面数量越多，则 KSM+对系统的性能影响越大，反之越小。

4.2 KSM+的实现

本节介绍实现 KSM+的关键代码，它包括新的用户交互方式和新的扫描算法两部分。

4.2.1 用户交互方式的核心代码

KSM+中用一个循环链表维护需要扫描的程序。用户通知扫描某程序时，程序名被加入到循环链表，用户通知取消扫描某程序时，程序名从循环链表中删除。对应的核心代码如下：

```
struct process_to_scan *pts;      // 一个记载程序名的结构体的指针
sscanf(buf, "%c %s", &op, comm);  // 处理格式 “+ 程序名”
len = strlen(comm);       // 得到程序名的长度
spin_lock(&processes_to_scan_lock); // 得到操作循环双链表的自旋锁
exist = process2scan_exist(comm); // 是否已经存在这个程序名
if (op == '+' && !exist)   // 当操作是+，程序名又未被记载时
```

```
{
    pts = alloc_process_to_scan();  // 分配一个 process_to_scan 的结构体
    pts->comm = kmalloc(len+1, GFP_KERNEL); // 分配内存存下文件名
    if (!pts->comm) {   // 极端情况下的出错处理.
        free_process_to_scan(pts);  // 释放掉之前分配的 process_to_scan
结构体
        goto _err;      // 跳到出错处理.
    }
    strcpy(pts->comm, comm);  // 拷贝程序名
    list_add(&pts->list, &processes_to_scan_head); // 加入循环双链表
}
spin_unlock(&processes_to_scan_lock);  // 释放同步用自旋锁
```

KSM+的内核线程周期性地检查系统的系统的进程链表，发现需要扫描的程序创建了新的进程时，就创建一个 mm_slot 结构分配给该进程。如 3.4.1 小节所述，KSM 把所有的 mm_slot 链接在一起，每个 mm_slot 对应一个进程的内存描述符，KSM 将通过这些信息进行扫描操作。KSM+基本沿用 KSM 中的 mm_slot 结构体。对应的代码片断是：

```
read_lock(&tasklist_lock);  // 以读的方式得到系统中进程链表的读写锁
for_each_process(p) {     // 遍历每一个进程
    get_task_comm(comm, p);  // 得到进程的程序名
    matched = 0;
    spin_lock(&processes_to_scan_lock);//得到需扫描程序名称的链表的自旋
锁
    matched = process2scan_exist(comm); // 是否是需要被扫描的程序
    spin_unlock(&processes_to_scan_lock); // 释放自旋锁
    if (matched)  // 如果是需要被扫描的程序
    {
            task_ksm_enter(p);  // 该函数以 task_t 为参数做进一步处理
    }
```

```
    }
    read_unlock(&tasklist_lock); // 释放保护系统中进程链表的读写锁。
```

4.2.2 扫描算法的核心代码

KSM+与 KSM 在数据结构方面的主要区别是 KSM+引入了 vma_node 结构体。KSM 使用 mm_slot，rmap_item 两级数据结构，KSM+使用 mm_slot，vma_node，rmap_item 三级数据结构。rmap_item 挂在 vma_node 上。结构体 vma_node 的定义如下：

```
struct vma_node {
    struct rb_node; // 红黑树结点，用于把 vma_node 链接到 mm_slot 的红黑树上
    struct vm_area_struct *vma; // 对应的虚拟内存区域
    unsigned long start;  // 虚拟地址的起点
    unsigned long end;   // 虚拟地址的终点
    struct rmap_item *rmap_list;  // 挂接到该虚拟内存区域的 rmap_item 链表
    struct rmap_item *rmap_current;  // 当前扫描使用的 rmap_item
    u8 coefficient;  // 该虚拟内存区域使用的采样系数
};
```

更改数据结构后 KSM+的扫描算法概括为：遍历每个 mm_slot；对于 mm_slot，遍历每个 vma_node；对于每个 vma_node，调用采样函数 vma_node_do_sampling()生成挂接在 vma_node 上的 rmap_item；对于每个 rmap_item，调用寻找相同页面的逻辑。

其中最为核心的是采样函数 vma_node_do_sampling()，它完成三个主要职能：采样页面；回收没有合并成功的 rmap_item；回收 vma_node。

简化后采样页面的实现如下：

```
pages_count = (vma_node->end - vma_node->start) >> 12; // 计算有多少个页面
sample_count = pages_count * vma_node->coefficient / 100;// 计算要采样多少个页面
```

```
gap_len = pages_count / sample_count ; // 平均多少个页面中采样一个页面
left = right = 0;        // 预置左区间和右区间都是 0
while (1)
{
        left = right;   // 本次的左区间是上次的右区间
        right = left + gap_len;  // 本次的右区间
        if ( right > pages_count ) {  // 右区间大于页面总数.
            break;  // 采样结束
        }
        index = left + random32() % gap_len;  // 在区间内随机抽取一个页面
        address = vma_node->start + (index << 12); // 计算该页面的虚拟地址
 }
  vma_node->coefficient -= 10; // 本轮采样后减少采样系数的取值
```

简化后回收 rmap_item 的实现代码如下：

```
 // 双指针指向指向链表中第一个 rmap_item 的指针
struct rmap_item **item = &vma_node->rmap_list;
// 遍历链表，一直到当前指向的 rmap_item 的地址比当前采样的地址大.
while (*item && ((*item)->address & PAGE_MASK) < address)
{
            addr = (*item)->address;
            if(!(addr & STABLE_FLAG)) // 如果 rmap_item 不在稳定树上
            {
                    struct rmap_item *ri = *item;
                    *item = ri->rmap_list;//双指针指向指向下一个 rmap_item
的指针
                    free_rmap_item(ri); // 销毁该 rmap_item
                    continue;
                }
```

```
            item = &(*item)->rmap_list; // 双指针指向指向下一个 rmap_item 的
指针
        }
        if (*item) { // rmap_item 在稳定树或不稳定树上，又被本次采样命中
            addr = ((*item)->address & PAGE_MASK);
            if (addr == address) {
                item = &(*item)->rmap_list;
                continue;
            }
        }
        new = alloc_rmap_item();      // 分配新的 rmap_item
        ......  // 填充新 rmap_item 的字段
        *item = new;  // 把新 rmap_item 挂接到链表上
        item = &new->rmap_list;
```

到本轮采样的 sample_count 个页面全部处理完后，如果双指针 item 不为 NULL，即指向 rmap_item 链表尾部的剩余的结点，对这些结点执行上文一样的销毁 rmap_item 的算法。

回收 vma_node 的实现如下：

```
// 本轮过后没有任何 rmap_item 在稳定树上且采样系数小于 10
if (0 == stable_node_count && vma_node->coefficient < 10)
{
    struct rmap_item *item;
    while (vma_node->rmap_list) // 销毁 vma_node 上维护的所有 rmap_item
    {
        item = vma_node->rmap_list;
        vma_node->rmap_list = item->rmap_list;
        remove_rmap_item_from_tree(item);
        free_rmap_item(item);
    }
    vma_node->vma->vm_flags |= VM_NOT_MERGEABLE; // 置标志位
```

```
}
```

4.3 本章小结

本章详细介绍了 KSM+在 KSM 的基础上做的改进。KSM+主要针对桌面应用程序的应用场景改进了内核模块和用户空间的交互方式，使得用户可以方便的指定程序进行内存去重操作；以及模块中数据结构的优化，使 KSM+较之 KSM 自身更节省内存。最后，本章分析了 KSM+的实现部分的核心代码。

第五章 实验结果及分析

5.1 KSM+的实验结果

本小节用两类程序作为实验对象分析 KSM+的实际执行效果。第一类程序有大量的重复内存可以合并，第二类程序则是 Linux 桌面环境上的常用程序。

5.1.1 针对有大量重复页面的程序

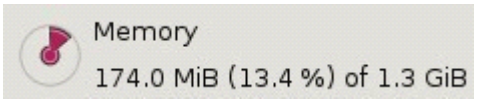
一般情况下，应用程序本身或应用程序之间并没有太多重复页面。但这样的例子并非不存在。如 KSM 模块的出现就是为了解决运行相同操作系统的虚拟机程序之间的存在大量重复内容页面的问题，再如 1.2 小节中提到的欧洲核子研究中心并行执行多个程序，载入大量相同的数据使得内存里有大量的重复内存页面。

为了实验方便，可以编写一个模拟程序，但仍能达到验证本工程的目的。验证所用的模拟程序 test.c 如下：

```
int main()
{
    char *mem;
    int i, size;
    size = 0x1400000;
    mem = (char*)malloc(size);
    for (i = 0; i < size; i++)
        mem[i] = (i%256);
    getchar();
    free(mem);
    return 0;
}
```

该程序用来测试一个极端的情况，它申请了 $20 * 2^{20}$ 字节，即 20M 内存空间，并把 20M 内存空间按字节依次循环设置为 0 到 255 之间的数。这样，如果分配的 20M 内存的起始地址按 4K 对齐，那么 20M 内存共有 $20 * 2^{20} / 2^{12}$ 个页面，即 5120 个页

面，且每个页面的内容一样。编译这个程序：$ gcc test.c –o test。并运行该程序的两个进程实例。如图 5-1 所示，这两个进程的 PID 分别为 2207 和 2206，它们都使用了 20.1MB 内存，且此时整个系统使用内存 174MB。



Process Name	Status	% CPU	Nice	ID	Memory
test	Sleeping	0	0	2207	20.1 MiB
test	Sleeping	0	0	2206	20.1 MiB

图 5-1 合并页面前系统消耗的内存和 test 程序消耗的内存

下面让 KSM+合并内存，命令如下：

#echo "+ test" > /sys/kernel/mm/ksm+/processes_to_scan

#echo 1 > /sys/kernel/mm/ksm+/run

第一条命令使用了改进后的交互方式，它通知 KSM 合并程序名为 test 的重复内存。第二条命令让 KSM 进入合并模式，如果给/sys/kernel/mm/ksm+/run 写入 0 值，则 KSM+停止工作。目前本工程的实现中，KSM+默认在停止状态。

KSM+完成所有合并的时间长短取决于对 KSM+设置的描述轮数之间的间隔时间和每轮扫描的页面多少，采用越小的时间间隔，越多的每轮描述页面数量，会对系统的整体响应时间造成越大的负面影响。更改后的默认设置为每 600MS 描述一轮，每轮描述 100 个页面。因此，在最好的情况下，每个描述页面都能被合并，那么默认的合并速度是 667K/S。由此得出，在默认设置下，需要 40*1024/667 秒，大概一分钟的时候合并完 40M 内存空间。本次实验中经过大约 65 秒后，重复页面被合并完毕。如图 5-2 所示，两个 test 进程在重复页面被合并后都减少了近 20M 内存占用量，即共节省了 40MB 内存。其中 PID 为 2207 的进程比 PID 为 2206 的进程多占用了 4K，即一个页面的内存量。这是因为近 40M 大小的相同页面被合并到一个页面，而这个页面在 PID 为 2207 的进程中。另外，从系统全局来看，系统占用的内存从 174.0MB 减少到 134.1MB，共减少 39.9MB，非常接近减少 40MB 的预期。这 0.1MB 的细微差距受到系统中其它进程申请和释放内存的影响。

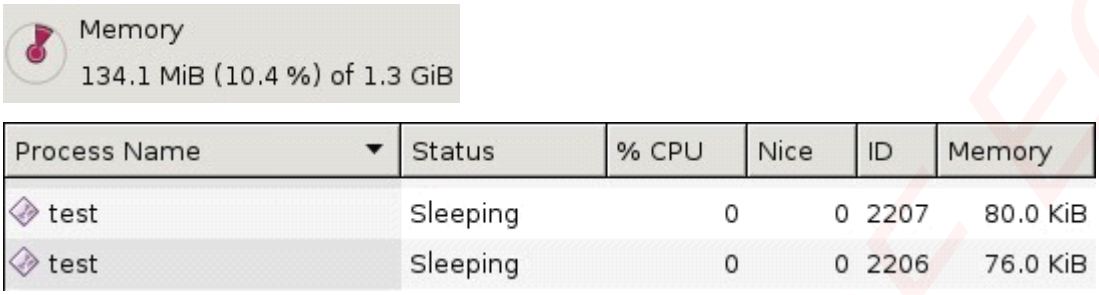


图 5-2 合并页面后系统消耗的内存和 test 程序消耗的内存

此时，如果需要撤消所有合并的页面，可用命令# echo 2 > /sys/kernel/mm/ksm+/run。

撤消合并页面后，两个 test 进程消耗的内存又回到各 20MB。

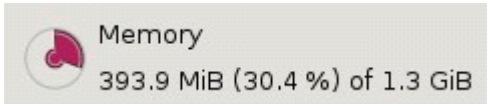
本小节的实验证明 KSM+可以让用户方便地指定合并某程序的重复内存，对于有大量重复页面的程序，改进后的 KSM+能显著地减少它们的内存使用量。

5.1.2 针对 Linux 桌面的常用程序

上一小节是对假想程序的实验效果，而本节将在实际的 Linux 桌面应用环境中检验 KSM+。

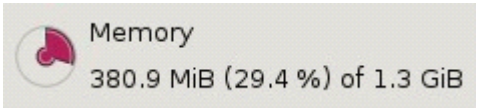
实验一。Okular 是 Linux 平台下一个常用的 pdf 阅读软件。本次实验同时运行 okular 程序的四个进程实例，每个进程打开不同的 pdf 文件。四个 okular 进程所消耗的内存资源以及系统消耗的总内存资源如图 5-3 所示。在默认设置下，经过大约两分钟，KSM+基本完成了合并重复页面。此时，四个 okular 进程以及整个系统所消耗的内存资源如图 5-4 所示。经计算，与合并前相比，四个进程共节省内存 14.4MB。比较整个系统在合并前后的内存使用，也减少了 13M 内存。因为比较整个系统的内存减少量时把 KSM+本身的内存消耗计算在内，所以数值会比单独考察被合并进程的内存减少量要少一些，但比较整个系统的内存变化更合理。另外，查看 /sys/kernel/mm/ksm+/pages_sharing 文件可以得到共节省了多少个页面。可使用命令： # cat /sys/kernel/mm/ksm+/pages_sharing，此时得到输出 3668。因为每个页面占 4K 大小，即共节省内存 14672K，非常接近之前计算得到的 14.4MB。如果合并效

率定义为节省的内存除以合并前的内存之和。那么此次内存合并效率为：14.4/(22.8+21.4+19.9+15.3) = 18.14%。



Process Name	Status	% CPU	Nice	ID	Memory
okular	Sleeping	0	0	2361	22.8 MiB
okular	Sleeping	0	0	2378	21.4 MiB
okular	Sleeping	0	0	2351	19.9 MiB
okular	Sleeping	0	0	2370	15.3 MiB

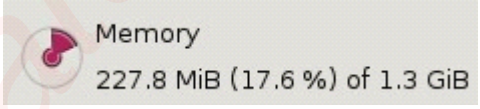
图 5-3 合并页面前系统消耗的内存和 okular 程序消耗的内存



Process Name	Status	% CPU	Nice	ID	Memory
okular	Sleeping	0	0	2361	18.8 MiB
okular	Sleeping	0	0	2378	15.6 MiB
okular	Sleeping	0	0	2351	16.9 MiB
okular	Sleeping	0	0	2370	13.7 MiB

图 5-4 合并页面后系统消耗的内存和 okular 程序消耗的内存

实验二。Gimp 是 Linux 平台下最常用的图像编辑软件，相当于 Windows 平台下的 Photoshop。本次实验用 gimp 打开 5 张图片。合并页面前后，系统消耗的内存和 gimp 单独消耗的内存分别如图 5-5 和图 5-6 所示。计算得到合并页面前后，系统消耗的内存减少 13.7M，而 gimp 进程消耗的内存减少 14.7M，这两个数值非常接近。内存合并效率为 14.7/91.2 = 16.12%。本次实验说明 gimp 程序也有可观的重复内存用来合并。



Process Name	Status	% CPU	Nice	ID	Memory
gimp-2.6	Sleeping	0	0	2158	91.2 MiB

图 5-5 合并页面前系统消耗的内存和 gimp 程序消耗的内存

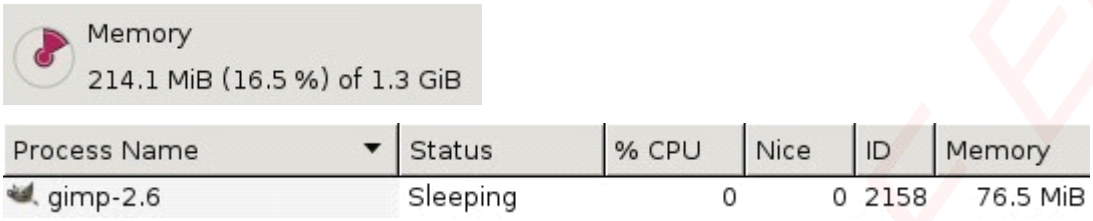


图 5-6 合并页面后系统消耗的内存和 gimp 程序消耗的内存

实验三。Epiphany 是 Linux 下 GNOME 桌面的默认浏览器。为了防止浏览器和 WEB 服务器持续交互影响测试效果，本次实验用该浏览器打开四个已经下载好的网页并工作在脱机状态。这四个页面分别是：腾讯门户的首页，搜狐门户的首页，以及两个新闻页面。

合并页面前后，系统消耗的内存和浏览器单独消耗的内存分别如图 5.7 和图 5.8 所示。计算得到合并页面前后，系统消耗的内存减少 1.1M，而浏览器消耗的内存减少 3.6M。内存合并效率仅为 3.6/90.5 = 3.97%。本次实验说明 epiphany 浏览网页时没有很多的可合并页面。

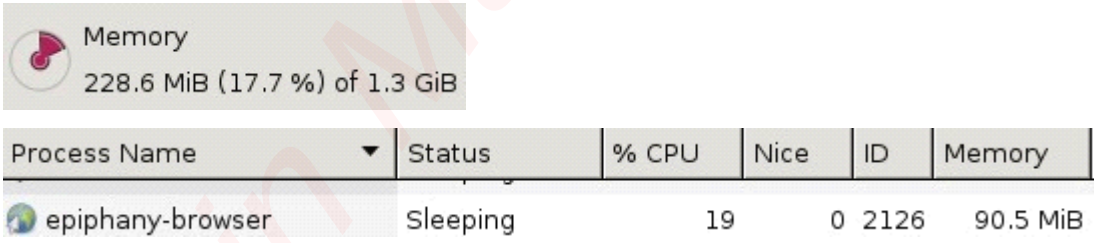


图 5-7 合并页面前系统消耗的内存和浏览器程序消耗的内存

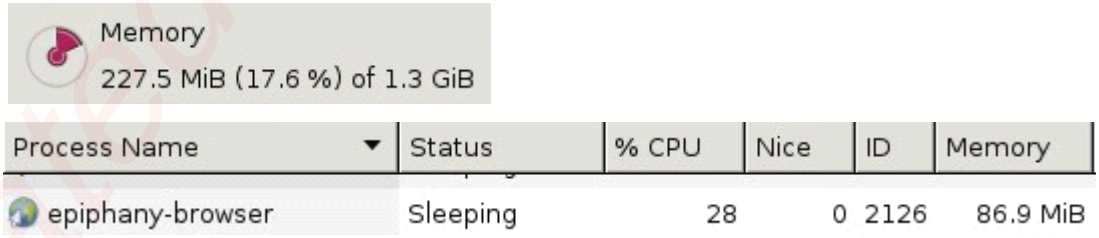


图 5-8 合并页面后系统消耗的内存和浏览器程序消耗的内存

实验四。本实验模拟常见的 Linux 桌面用户使用环境，测试增强后的 KSM+对整个系统使用内存的影响。实验假设用户打开了一个 okular 进程阅读 pdf，一个 epiphany 浏览器进程浏览网页，一个 openoffice 进程处理文档，以及一个控制台打开四个标签(五个进程)。实验用一个简单的 Bash 脚本控制 KSM+扫描这几个进程。脚本如图 5.9 所示。

```
1 #!/bin/bash
2
3 KSMPLUS_DIR=/sys/kernel/mm/ksm+/
4 PROS2SCAN=${KSMPLUS_DIR}processes_to_scan
5 RUN=${KSMPLUS_DIR}run
6
7 echo "+ okular" > ${PROS2SCAN}
8 echo "+ epiphany-browse" > ${PROS2SCAN}
9 echo "+ soffice.bin" > ${PROS2SCAN}
10 echo "+ gnome-terminal" > ${PROS2SCAN}
11 echo "+ bash" > ${PROS2SCAN}
12 echo 1 > ${RUN}
```

图 5-9 用脚本控制 KSM+合并程序内存

脚本的第 3 行设置 KSM+在 sysfs 文件系统中的位置，第 4 行指定了 KSM+接受用户指示扫描哪些进程的文件路径，第 5 行指定了开启关闭 KSM+的文件路径，第 7 到第 11 行把本次实验要求扫描的进程名通知给 KSM+，第 12 行开启 KSM+进行合并。

比较本次实验前后系统占用的总内存如图 5.10 所示。计算可得系统总占用内存在 KSM+合并内存后减少 5.7MB。

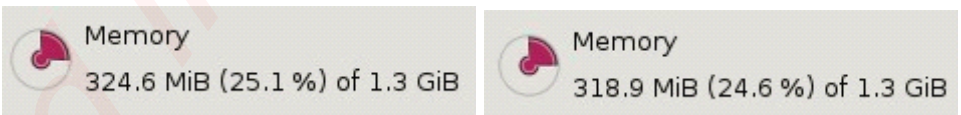


图 5-10 合并内存前后系统内存占用对比

本次实验扫描的八个进程在合并前后占用的内存如表 5.1 所示。从表 5.1 的数据我们可以看到，大多数 Linux 桌面应用程序并没有大量的重复内存页面。对比从表 5.1 得到的节省内存量的数据为 11.7MB，而与从图 5.10 得到的数据为 5.7MB，导致它们之间有 6MB 的差距部分因为系统中其它程序的影响，部分因为 KSM 本身所消耗的内存。

表 5-1 内存合并效果表(单位：MB)

进程名	合并前	合并后	节省内存	合并效率
okular	18.0	13.4	4.6	25.56%
epiphany	42.1	39.0	3.1	7.36%
soffice.bin	37.9	37.7	0.2	0.53%
terminal	3.4	3.3	0.1	2.94%
bash	2.5	2.3	0.2	8.00%
bash	2.5	2.4	0.1	4.00%
bash	2.5	2.3	0.2	8.00%
bash	2.5	2.3	0.2	8.00%
合计	111.4	102.7	11.7	10.50%

本小节的四个实验说明，KSM+算法对于部分有可观数量的重复内存的应用程序，如 pdf 阅读器 okular，图像编辑软件 gimp 有明显的节省内存使用量的作用。但对于内存中重复页面非常少的应用，且本身使用的内存量也不多的程序，则没有太大的意义。

5.2 KSM 与 KSM+的对比

本节比较了在实际实用中 KSM 和 KSM+的差异，证明 KSM+达到了当初的设计要求。

5.2.1 自动采集代替源代码级显式指定

3.2.2 小节叙述了改进前的 KSM 需要在源代码级显式调用 madvise 通知 KSM 有哪些合并区域。该函数的第一个参数需置为 MADV_MERGEABLE，第二个参数是内存的起始地址，第三个参数是内存区域的长度。需要注意的是，传入的内存地址必须按页面的大小对齐，否则 madvise 会返回-1，错误信息为“Invalid argument”。为保证内存地址按页面大小对齐，可使用函数 posix_memalign。该函数在头文件 sys/mman.h 中声明，它的函数原型为：

int posix_memalign(void **memptr, size_t alignment, size_t size);

第一个参数用来返回函数成功时内存起始地址；第二个参数指明对齐边界的值，它必须是 2 的整数次方；第三个参数指明内存区域的大小。该函数在成功时返回 0 值，失败时返回错误码，但不设置全局错误 errno。

而改进过的 KSM+通过 sysfs 文件系统指定程序名即可，用户不必要修改代码，也不必考虑指定内存区域，KSM+自动寻找可合并的内存页面。

5.2.2 更省内存

KSM 为指定内存区域的每一个页面生成一个 rmap_item，而改进后会在合并失败时收回 rmap_item，释放它占用的内存。在内核对 x86_64 处理器架构的实现代码中，一个 rmap_item 占 64 字节大小。如果改进前的 KSM 扫描 100MB 内存区域没有一个页面能够合并成功，理论上计算，那么 KSM 要消耗 1600K 内存分配给 rmap_item。KSM+则有收回分配的 rmap_item 的机制，合并失败的页面可能会被收回 rmap_item，且当某虚拟内存区域被认为没太大希望存在重复页面时则不会再分配 rmap_item 到此区域。具体的收回 rmap_item 的算法见 4.1.3 小节所示。但因为申请和回收 rmap_item 通过内核的 Slab 层进行，改进的 KSM 代码中用 kmem_cache_free 内核 API 回收 rmap_item 时，取决于 Slab 能够容纳多少空闲的 rmap_item，收回的 rmap_item 可能被彻底释放给操作系统，也有可能并没有被操作系统完全回收且能分配给其它内核代码使用，而只是返回给了原先的 Slab，等待下一次分配 rmap_item 时 Slab 能够迅速地把闲置地 rmap_item 返回给申请者。更详细的叙述见 2.1.5 小节。但通过比较 rmap_item 的数量，可以发现改进后 KSM 使用更少的 rmap_item。

5.2.3 实验验证

下面用一个实验程序比较 KSM 与 KSM+的不同。该程序去掉无关代码和出错处理代码后的核心代码片断如图 5-11 所示：

```
1 size = 0x6400000;  // 100M
2
3 ret = posix_memalign((void**)&mem0, 4096, size);
4 ret = posix_memalign((void**)&mem1, 4096, size);
5
6 ptr0 = (unsigned long*)mem0;
7 ptr1 = (unsigned long*)mem1;
8
9 for (i = 0; i < size/sizeof(unsigned long); i++){
10         ptr0[i] = i;
11         ptr1[i] = -i;
12 }
13 memset(mem0, 0xff, 0x100000);
14 memset(mem1, 0xff, 0x100000);
15
16 ret = madvise(mem0, size, MADV_MERGEABLE);
```

图 5-11 KSM 与 KSM+对比实验用程序代码

第 1 行代码指定了一个 100MB 大小的数值。第 3 行和第 4 行在堆上分配了两块内存，它们的起始地址都以一个页面的大小对齐；第 9-12 行确保两块内存没有内容相同的页面；第 13 行设置第一块内存的前 1MB 全是 0xff，即前 256 个页面都是全 1 页面；第 14 行设置第二块内存的前 256 个页面都是全 1 页面；第 15 行通知 KSM 需要合并第一个内存区域中的重复内存。

这段程序分配了两个匿名内存区域。它们都是 100MB 大小，即 21500 个页面。两个内存区域加在一起共有 51200 个页面，其中 512 个页面是全 1 页面，其它的页面都各不相同。可见，这两个内存区域的重复页面密度都很低，平均一百个页面里才有一个页面可以被合并。本实验程序特意只把其中一个内存区域注册给 KSM，用以说明 KSM 只能合并在源代码级别被显示要求合并的内存区域。另外，本实验还旨在说明 KSM+比 KSM 更省内存。按照前两节的理论分析， KSM 只能合并第一个内存区域中的 256 个重复页面，却消耗 25600 个 rmap_item，浪费掉 64*(25600-256)/1024 字节，即 1584KB 内存。而 KSM+能合并两个内存区间的 512 个全 1 页面，几轮扫描过后，消耗的 rmap_item 会远比 KSM 少。KSM+除了有 512 个 rmap_item 链接在 KSM+稳定树的结点上，还需要一些 rmap_item 用来周期性采样内存区域，但应远远小于 KSM 分配的 rmap_item 的数量。

由表 5-2 的数据证实了之前的理论推导：（1）KSM 只能合并 1MB 内存，因为只有注册了的内存区域才能执行合并的逻辑；而 KSM+自动发现了所有有重复页面的内存区域，所以合并了 2MB 内存。（2） KSM 扫描的内存区域的大小只有大约 KSM+扫描的内存区域大小的一半，但 KSM 使用的 rmap_item 却比 KSM+要多得多，直接导致合并后系统消耗的内存数量比合并前还要多 0.6MB，但 KSM+使得系统总消耗内存有 1.3MB 的下降。经简单的计算可以看出 KSM+合并了 2MB 大小的重复页面，自身消耗了大约 0.7MB 内存空间。因此，可以得出结论是：KSM+能自动找到可以合并的内存页面，并且在应对重复页面非常稀少的进程时，仍然能减少系统的总内存使用量。而 KSM 只合并在源代码中显示指定的内存区域，而且在应对重复页面非常稀少的进程时，可能不仅没能降低内存使用，相反还会增加系统的内存使用。

表 5-2 KSM 和 KSM+的比较

	合并前系统所用内存	合并后系统所用内存	合并前程序所用内存	合并后程序所用内存	合并后 rmap_item 数量
KSM	334.4MB	235.0MB	200.1MB	199.1MB	25600
KSM+	325.8MB	324.5MB	200.1MB	198.1MB	1027

5.3 重复页面的统计分析

为了能进一步改进 KSM+，同时也可验证 5.1.2 小节的实验结果。本节用一个专门的内核模块来统计程序的重复页面的数量和重复页面的内容。通过分析统计数据，可以发现 Linux 桌面应用程序中重复内存页面的一些特点。

5.3.1 编写统计重复页面的内核模块

由 5.1.2 小节的实验可知各个程序之间合并内存页的效果有较大的差导。为了持续改进 KSM+的合并效率，可以从分析这些应用程序中的重复页面的特点着手。比如，重复次数最多的是哪个页面，重复了多少次，每个重复页面的内容是什么，哪些常用应用有较多的重复页面。为了统计这个信息，可以在目前 KSM+代码的基础上修改，

增加统计信息的功能，但这样会让 KSM+的代码变得更加复杂。考虑到内核代码难以调试的特点，本工程把统计重复页面的功能独立出来作为一个单独的模块。这样的好处是：第一，独立的模块易于调试，出错几率低；第二，只专注于做统计信息这一个简单的功能，不必像 KSM+中的代码一样必须防止分配过多内存，当有足够多的内存可供使用时，代码编写的难度大大降低；第三，单独模块的统计结果可以和 KSM+代码的相关结果作对比，协助排除程序的错误。

该统计模块设计的目标是让用户方便地指定统计哪些程序，统计模块输出所有的重复页面的重复次数，以及有多少页面没有重复页面。该统计模块与用户交互的部分的需求和 KSM+的用户交互需求一致，所以沿用了设计 KSM+时的办法：让用户通过 sysfs 文件系统与模块交互，指定统计哪些程序。约定的格式也保持一致，即写入“+程序名”即通知模块统计哪个程序，写入“- 程序名”即通知模块撤消统计哪个程序。

统计部分的思路是：初建一棵红黑树。每扫描一个页面，计算出该页面的 MD5 值，以该 MD5 值为索引插入红黑树。如果树中没有该 MD5 值，则增加一个结点插入到树中，结点的计数置为 1。如果树中已经存在该 MD5 值，则给对应结点的计数增加 1。该计数则是重复页面的计数。

虽然统计模块和 KSM 中一样都用了红黑树的数据结构，但 KSM 中的树的索引是用了页面的内容。如果页面的内容发生变化，则红黑树会部分无效。KSM 中的不稳定树的解决方案是每轮扫描都重建不稳定树。而稳定树的解决方案是写保护稳定树中的页面。在本统计模块中，以扫描时计算得到的 MD5 值为索引，即使在扫描过后，页面的值发生了变化，但之前扫描得到的 MD5 值并不会被重新计算，这样就保证了树仍然是有效的。理论上，虽然这样的解决方案得到的统计结果会和真实情况有小小的差距，但因为整个统计过程发生在极短的时间里，而在很短的时间内发生变化的页面只是极少数，并不影响统计数据的代表性。另一方面，如果 KSM 中的红黑树用 MD5 值作为索引，则会给每个页面带来 16 个字节的额外内存开销。此外，MD5 值存在安全性问题[35]。所以 KSM 中的红黑树和统计模块中的红黑树采用不同的索引方案。

当全部页面扫描结束后，红黑树上的每一个结点则记录了对应的页面的重复次数。本工程使用堆排序的方法把重复的页面次数按顺序排列。执行堆排序时需要一棵完全二叉树，而红黑树并不是完全二叉树。本工程的解决思路是：计算得到红黑树共

有 n 个结点，再分配一个有 n 个元素的一维数组，每个元素都是一个指针。遍历红黑树，依次把一维数组上的每个无素指向红黑树上的结点。至此，这个一维数组存储了一棵完全二叉树。于是，对该完全二叉树使用一次堆排序后，该一维数组中的元素即按重复页面次数从小到大或从大到小排列了。

该统计模块被命名为 parser.ko。

5.3.1 重复页面的统计分析

在 5.1.2 小节中，实验发现 pdf 阅读工具 okular 和图像编辑工具 gimp 有比较多的重复内容页面，epiphany 浏览器浏览网页时的重复页面也有一定数量，openoffice 的重复页面较少。本小节用统计模块分析在 5.1.2 小节的实验中测试过的程序。

分析一。本次测试也运行程序 okular 的四个进程实例，打开四个不同的 pdf 文档，各随机阅读一些页面。之后用命令“insmod parser.ko”把统计模块插入内核，再转到/sys/kernel/mm/parser 目录，用命令“echo + okular > processes_to_scan”通知 parser 模块采集程序名为 okular 的进程的页面信息，最后读文件 statistics 即得到统计信息。本次输出如图 5-12 所示。

图 5-12 所示的第一栏是重复页面的次数，第二栏是该重复页面的 MD5 值。可以明显看到 okular 的内存中绝大多数页面不是重复的，即该图的最后一行所示 13249 个页面没有重复，而重复页面的分布非常集中，排名第一的重复页面有 2746 页，占所有页面的 16%。而其它的所有重复页面全部加上也只有 565 个页面，数量上只有排名第一的 20.6%。排名前三的重复页面共 3217 项，占据了所有重复页面的 97.2%。理论上合并这 3311 个重复页面可以合并到 27 个页面，因此可节省(3311-27)*4KB，即 13136KB 内存，与实验一的数据基本一致。

因为把一个页面的内容打印在屏幕上需要 8192 个字符，这个数量非常大，所以统计模块把页面的内容打印到了内核日志文件中。查看/var/log/kern.log 文件，可以看到，排名第一的重复页面是所有比特位被置 1 的页面，称为全 1 页面；排名第二的重复页面是所有比特位被置 0 的页面，称为全 0 页面；排名第三的重复页面以 32 个字节的长度为周期循环，每个周期为一串奇怪的字节。联想到白色的 RGB 值是(255,255,255)，黑色的 RGB 值是(0,0,0)，而 pdf 阅读器显示文章时有很大一部分的白

色区域，以此为根据本论文认为排名第一和第二的大量的重复页面和程序的图形显示有关。

```
root@infatuate:/sys/kernel/mm/parser# cat statistics
2746 6AE59E64850377EE5470C854761551EA
317 620F0B67A91F7F74151BC5BE745B7110
154 1DB2CF3C7A232CE360F50E0D2089A28D
14 285A60C8BB93DCE155F83A32272ECBA1
8 A23F076CDF2758059B7DE2EC01192FDD
7 90B4A05E6ED239C22758ADBEEBE6CD1F
7 1F955[illegible]C74CC0AF4
4 AAD8[illegible]3A660D2A7A
4 534E7C6EEC2F7D77AAB28923AA8955D6
4 969F5FAAC83FC68FA3F4AE7512D287EB
4 258B78AED70BE6C8C4147E757B316953
4 F218D89CA538AC61C3C5EEC8B3D15582
4 76AD83F8EE9B6DA89D8D076FDF4A43CE
4 B5987215C94533ADE40856EE72637E74
3 EA6FDABE9DEA92DD135D6A700AA2D7DF
3 91C0CEA0DF5884B70CCFE00E83F195E2
3 C5AD18E5FDFBF008A915793B299E37DB
3 110974D59D9637115E7359E63586A890
2 AE80F7E3A9EC0CF15F3B8445E6A76B11
2 5BC4216FECB42670225915A7B3EFA117
2 4854B7A6970A954972BA1D87F0A641CA
2 21A199C53F422A380E20B162FB6EBE9C
2 897ED296D6A9FB71879BC9120D9FA76D
2 080BDB20431E036B7FDEA54AA971F9D9
2 F78FCB9455E957B0036EDDF67CC1457E
2 DBE8D45809A1D59F88567260F8626DCB
2 D9B16C991D908E4C11D85C57F7532D3F
13249 pages have no matches.
```

图 5-12 四个 okular 进程的重复页面统计

分析二。此次测试与 5.1.2 节中的实验二一样，用 gimp 打开五张图片。分析结果显示 gimp 中的的重复内存页面分布较之 okular 要分散得多。共扫描了 23012 个页面，其中 3702 个页面是重复页面。在分析 okular 程序的分析一中排名第一，占据了总页面的 16%的全 1 页面在分析 gimp 的测试中仅有区区 6 个重复次数。原排名第二的全 0 页面在此次测试中排名第一位，页面总共重复了 1029 次。排名第二位的页面重复了 796 次，这个页面几乎是全 0 页，只有少数几个字节不是 0，排名第三到第五的页面均重复了 100 次以上，页面的内容是一些特殊内容的字节循环填充了整个页面。

此外，重复次数在 10 次以上，100 次以下的页面有 13 个；重复次数在两次以上(包括两次)，10 次以下的页面有 166 个。理论上合并这些重复页面可以节省(3702-5-13-166)*4KB，即 14072KB，与实验二的数据基本一致。

分析三。此次测试对应 5.1.2 节中的实验三，用 epiphany 以脱机状态打开四个网页。分析结果显示一共扫描到 21165 个页面，有 688 个重复页面，占总页面数的 3.2%。占据重复页面次数第一位的是全 0 页面，共有 366 个全 0 页；排名第二位的是全 1 页面，共 29 个全 1 页；排名第三位的共有 13 个重复页，这个重复页面的内容为以四字节为周期不断重复“FFFFFF00”；其它重复页面的重复次数都是 5 次以下，重复 5 次的有 1 个，重复 4 次的有 3 个，重复 3 次的有 53 个，重复两次的有 52 个。理论上合并这些重复页面可以节省(688-3-1-3-53-52)*4KB，即 2304KB，与实验三的数据基本一致。

以上三个实例分析验证了 5.1.2 小节中改进后的 KSM 算法基本上合并了所有的重复页面。另外从本节也可以看到应用程序中的重复页面的一些特点。1.全 0 和全 1 的特殊页面占了重复页面的绝大部分。2.其它重复页面也多有周期规律，是以一定长度内容为周期不断重复填充了整个页面。3.程序与程序之间重复页面所占总页面的比例有较大变化。这些规律为进一步改进 KSM 的算法提供了部分思路。

5.4 本章小结

本章通过实验验证了用户可以方便地指定 KSM+合并程序的相同内存页面。实验结果显示 KSM+对部分存在比较多重复内存的程序的合并效果明显。然后，本章用一个实验对 KSM 和 KSM+做了比较。实验证明 KSM+达到了设计的目的，在桌面应用程序的应用场景下，KSM+较之 KSM 更方便也更省内存。最后，还编写了一个专门统计重复内存页面的内核模块，对 Linux 桌面环境下的常用程序的重复内存页面情况进行统计分析，揭示了这些重复内存页面的一些特点，对持续改进 KSM+提供了依据。

第六章 总结与展望

6.1 全文总结

活跃的 Linux 社区为顺应云计算，虚拟化的兴起，在内核主干代码中加入了内核自己的虚拟机模块 KVM。紧接着，为解决虚拟机之间运行同样的操作系统或者大型服务器程序带来的大量重复内存的问题，内核中又加入合并虚拟机程序中重复页面的模块 KSM。目前大量对 KSM 的研究集中在虚拟化的场景，实际上 KSM 也可以应用在桌面环境中。本文在 KSM 的基础上专门针对 Linux 桌面环境设计并实现了 KSM+，能方便有效地合并桌面应用程序的重复内存。

论文的主要工作体现在以下几个方面：

（1）详细研究了 Linux 内核的内存管理子系统以及 Linux 内核开发的相关技术。

（2）详细分析了 KSM 模块的实现原理和页面合并技术。

（3）增强并改进了 KSM 应用在桌面 Linux 环境下的功能。改进后的 KSM+能让用户方便地指定需要合并重复内存页面的程序。并且，KSM+比改进前更节省内存。论文用大量的实验验证了 KSM+合并应用程序的重复内存的实际效果，并设计实验比较了 KSM 和 KSM+。

（4）编写了用于统计应用程序内存中重复页面数量的内核模块。通过对应用程序中重复内存页面的统计分析，为持续改进 KSM+提供了依据和线索。

6.2 未来展望

KSM+对系统性能的影响有待进一步研究。比如，虽然 KSM+消耗的处理器资源和配置参数有关，但究竟消耗处理器资源的情况如何，因为时间关系，本论文还没有这方面的实验数据。另外，KSM+在合并了应用程序的重复页面之后对应用程序的性能有多大的影响，也没有这方面的实验数据。

KSM+还有很多可以研究和改进的方向。比如通过钩子技术监视虚拟内存区域(VMA)的创建，销毁，变动，可以让 KSM+更快速的发现 VMA 的变化。另外，KSM+可以从单机程序扩展到有服务端的网络程序。服务端通过收集大量常用应用的重复页面

情况，根据这些重复信息精准地通知客户端应该在哪些应用程序的 VMA 中搜索重复页面，从而避免在重复内存非常少的内存区域消耗大量的处理器资源。

虽然计算机硬件不断飞速发展，但软件也越来越占用更多的内存。内存资源仍然是计算机系统中稀缺的资源。又因为应用程序消耗的内存越来越多，程序之间存在相同内容页面的几率也随之增大。如果能再降低 KSM+在合并相同页面的过程中消耗的处理器资源，那么就可以在占据了消费电子类市场相当市场份额的 Android 操作系统中移植该技术，对于其它基于 Linux 的操作系统也是如此，对 KSM+技术的研究有着广泛的应用前景。

6.3 本章小结

本章总结了全文的主要内容，对主要工作进行了归纳，并指出了进一步改进 KSM+的方向和未来的应用前景。

参考文献

[1] 俞甲子，石凡，潘爱民.程序员的自我修养，链接、装载、与库.电子工业出版社，2010.

[2] 李烨. 云计算的发展研究. 硕士论文.北京邮电大学.3.2011

[3] A. Binu, G. Santhosh Kumar. Virtualization Techniques: A Methodical Review of XEN and KVM. Communications in Computer and Information Science, v 190 CCIS, n PART 1, p 399-410, 2011

[4] Jin, Guohua1; Mellor-Crummey, John1.Improving performance by reducing the memory footprint of scientific applications. International Journal of High Performance Computing Applications,2005,19 433-451

[5] Carl A. Waldspurger, Memory resource management in VMware ESX server, ACM SIGOPS Operating Systems Review, v.36 n.SI, Winter 2002

[6] Eto, Mikinon1 ; Umeno, Hidenori1.Design and implementation of content based page sharing method in Xen for the International Conference on Control, Automation and Systems (ICCAS) 2008

[7] C.-R. Chang, J.-J. Wu, and P. Liu. An empirical study on memory sharing of virtual machines for server consolidation. In Proceedings of the 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications, ISPA '11, Washington, DC, USA, 2011. IEEE Computer Society.

[8] Timothy Wood , Gabriel Tarasuk-Levin , Prashant Shenoy , Peter Desnoyers , Emmanuel Cecchet , Mark D. Corner, Memory buddies: exploiting page sharing for smart colocation in virtualized data centers, Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, March 11-13, 2009, Washington, DC, USA

[9] Prateek Sharma and Purushottam Kulkarni. Singleton: system-wide page deduplication in virtual environments. In Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing, HPDC '12, pages 15-26. ACM, 2012.

[10] Yagi, T., Artho, C., Suzaki, K., Iijima, K.: Effects of memory randomization, sanitization and page cache on memory deduplication. In: European Workshop on System Security, EuroSec 2012

[11] Suzaki, K., Iijima, K., Yagi, T., Artho, C.: Memory deduplication as a threat to the guest os. In: Proceedings of the Fourth European Workshop on System Security, EUROSEC 2011, pp. 1:1 – 1:6. ACM, New York ,2011

[12] K. Suzaki, K. Iijima, T. Yagi, and C. Artho. Software side channel attack on memory deduplication. SOSP POSTER, 2011.

[13] Rauschmayr, Nathalie, and Achim Streit. Reducing the Memory Footprint of Parallel Applications with KSM. Facing the Multicore-Challenge III. Springer Berlin Heidelberg, 2013. 48-59.

[14] International Data Corporation. Android and iOS Combine for 91.1% of the Worldwide Smartphone OS Market in 4Q12 and 87.6% for the Year. http://www.idc.com/getdoc.jsp?containerId=prUS23946013

[15] Bovet, Daniel P./ Cesati, Marco .Understanding The Linux Kernel.3rd edition. Oreilly & Associates Inc.2005

[16] 赵炯. Linux 内核完全剖析. 机械工业出版社.2006

[17] Jonathan Corbet: Transparent huge pages in 2.6.38 http://lwn.net/Articles/423584/ 2011

[18] 刘立圆,于松波,吕晓娴. Linux 虚拟内存管理.数字技术与应用.2011

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

[20] 陈鸣春,潘金贵.一个基于 IA-64 体系的内存管理大页面的实现模型.计算机科学.2007,4: 276－278

[21] Claudia Salzberg Rodriguez,Gordon Fischer,Steven Smolski . The Linux Kernel Primer : A Top-Down Approach for x86 and PowerPC Architectures. Prentice Hall PTR. 2006

[22] 李毅.Slab 内存分配策略与移植.计算机技术与发展.2007.

[23] Robert Love. Linux Kernel Development. 3rd Edition. Addison-Wesley Professional.2011

[24] Wolfgang Mauerer . Professional Linux Kernel Architecture. Wrox.2010

[25] 洪永学,余红英,姜世杰,林丽蓉.Linux 内核调试技术的方法研究.电子测试.2012.

[26]Jonathan Corbet,Alessandro Rubini,Greg Kroah-Hartman.Linux Device Drivers.3rd edition. Oreilly & Associates Inc.2005

[27] wikipedia: KGDB. http://en.wikipedia.org/wiki/KGDB

[28] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the linux virtual machine monitor. In Ottawa Linux Symposium (July 2007), pp. 225–230.

[29]Jonathan Corbet: /dev/ksm: dynamic memory sharing. http://lwn.net/Articles/306704/ 2008

[30] Jonathan Corbet: KSM tries again. http://lwn.net/Articles/330589/ 2009

[31] The Linux Kernel Organization, Inc. How to use the Kernel Samepage Merging feature. https://www.kernel.org/doc/Documentation/vm/ksm.txt

[32] Kerrisk, M.: Linux programmer's manual. 2012 http://man7.org/linux/man-pages/man2/madvise.2.html

[33] Arcangeli, A., Eidus, I., Wright, C.: Increasing memory density by using KSM. In:
OLS 2009: Proceedings of the Linux Symposium, pp. 19 – 28 (July 2009)

[34] Bayer, R.: Symmetric binary b-trees: Data structure and maintenance algorithms.
Acta Informatica 1, 290 – 306 ,1972

[35] Sasaki, Y., Aoki, K.: Finding Preimages in Full MD5 Faster Than Exhaustive Search. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 134–152.

致　谢

本论文是在导师王常吉副教授的悉心指导下完成的。从选题到最后的终稿，王老师一丝不苟地对我的论文工作进行指导，提出了很多宝贵的意见。谨在此向王老师表示衷心的感谢。

感谢阿里巴巴公司的工程师李凡希，和金山安全公司的工程师项柱给予的技术上的建议和意见。

感谢 Linux 内核邮件列表的各位不知名开发者提供的帮助。

感谢父母在我调试内核代码最困难时期给予的精神上和后勤上的大力支持。

感谢徐天池同学，韦宇同学，蒋正阳同学，高静伟同学等各位同学提供的各种帮助。

嵌入式资源免费下载

总线协议：

1. 基于 PCIe 驱动程序的数据传输卡 DMA 传输
2. 基于 PCIe 总线协议的设备驱动开发
3. CANopen 协议介绍
4. 基于 PXI 总线 RS422 数据通信卡 WDM 驱动程序设计
5. FPGA 实现 PCIe 总线 DMA 设计
6. PCI Express 协议实现与验证
7. VPX 总线技术及其实现
8. 基于 Xilinx FPGA 的 PCIE 接口实现
9. 基于 PCI 总线的 GPS 授时卡设计
10. 基于 CPCI 标准的 6U 信号处理平台的设计
11. USB30 电路保护
12. USB30 协议分析与框架设计
13. USB 30 中的 CRC 校验原理及实现
14. 基于 CPLD 的 UART 设计
15. IPMI 在 VPX 系统中的应用与设计
16. 基于 CPCI 总线的 PMC 载板设计
17. 基于 VPX 总线的工件台运动控制系统研究与开发
18. PCI Express 流控机制的研究与实现
19. UART16C554 的设计
20. 基于 VPX 的高性能计算机设计
21. 基于 CAN 总线技术的嵌入式网关设计
22. Visual C 串行通讯控件使用方法与技巧的研究
23. IEEE1588 精密时钟同步关键技术研究
24. GPS 信号发生器射频模块的一种实现方案
25. 基于 CPCI 接口的视频采集卡的设计
26. 基于 VPX 的 3U 信号处理平台的设计
27. 基于 PCI Express 总线 1394b 网络传输系统 WDM 驱动设计

VxWorks：

1. 基于 VxWorks 的多任务程序设计

2. 基于 VxWorks 的数据采集存储装置设计
3. Flash 文件系统分析及其在 VxWorks 中的实现
4. VxWorks 多任务编程中的异常研究
5. VxWorks 应用技巧两例
6. 一种基于 VxWorks 的飞行仿真实时管理系统
7. 在 VxWorks 系统中使用 TrueType 字库
8. 基于 FreeType 的 VxWorks 中文显示方案
9. 基于 Tilcon 的 VxWorks 简单动画开发
10. 基于 Tilcon 的某武器显控系统界面设计
11. 基于 Tilcon 的综合导航信息处理装置界面设计
12. VxWorks 的内存配置和管理
13. 基于 VxWorks 系统的 PCI 配置与应用
14. 基于 MPC8270 的 VxWorks BSP 的移植
15. Bootrom 功能改进经验谈
16. 基于 VxWorks 嵌入式系统的中文平台研究与实现
17. VxBus 的 A429 接口驱动
18. 基于 VxBus 和 MPC8569E 千兆网驱动开发和实现
19. 一种基于 vxBus 的 PPC 与 FPGA 高速互联的驱动设计方法
20. 基于 VxBus 的设备驱动开发
21. 基于 VxBus 的驱动程序架构分析

Linux：

1. Linux 程序设计第三版及源代码
2. NAND FLASH 文件系统的设计与实现
3. 多通道串行通信设备的 Linux 驱动程序实现
4. Zsh 开发指南-数组
5. 常用 GDB 命令中文速览
6. 嵌入式 C 进阶之道
7. Linux 串口编程实例
8. 基于 Yocto Project 的嵌入式应用设计
9. Android 应用的反编译
10. 基于 Android 行为的加密应用系统研究
11. 嵌入式 Linux 系统移植步步通
12. 嵌入式 CC++语言精华文章集锦
13. 基于 Linux 的高性能服务器端的设计与研究
14. S3C6410 移植 Android 内核
15. Android 开发指南中文版
16. 图解 Linux 操作系统架构设计与实现原理（第二版）

17. 如何在 Ubuntu 和 Linux Mint 下轻松升级 Linux 内核
18. Android 简单 mp3 播放器源码
19. 嵌入式 Linux 系统实时性的研究
20. Android 嵌入式系统架构及内核浅析
21. 基于嵌入式 Linux 操作系统内核实时性的改进方法研究
22. Linux TCP IP 协议详解

Windows CE:

1. Windows CE.NET 下 YAFFS 文件系统 NAND Flash 驱动程序设计
2. Windows CE 的 CAN 总线驱动程序设计
3. 基于 Windows CE.NET 的 ADC 驱动程序实现与应用的研究
4. 基于 Windows CE.NET 平台的串行通信实现
5. 基于 Windows CE.NET 下的 GPRS 模块的研究与开发
6. win2k 下 NTFS 分区用 ntldr 加载进 dos 源代码
7. Windows 下的 USB 设备驱动程序开发
8. WinCE 的大容量程控数据传输解决方案设计
9. WinCE6.0 安装开发详解
10. DOS 下仿 Windows 的自带计算器程序 C 源码
11. G726 局域网语音通话程序和源代码
12. WinCE 主板加载第三方驱动程序的方法
13. WinCE 下的注册表编辑程序和源代码
14. WinCE 串口通信源代码
15. WINCE 的 SD 卡程序[可实现读写的源码]
16. 基于 WinCE 的 BootLoader 研究

PowerPC:

1. Freescale MPC8536 开发板原理图
2. 基于 MPC8548E 的固件设计
3. 基于 MPC8548E 的嵌入式数据处理系统设计
4. 基于 PowerPC 嵌入式网络通信平台的实现
5. PowerPC 在车辆显控系统中的应用
6. 基于 PowerPC 的单板计算机的设计
7. 用 PowerPC860 实现 FPGA 配置
8. 基于 MPC8247 嵌入式电力交换系统的设计与实现

9. 基于设备树的 MPC8247 嵌入式 Linux 系统开发
10. 基于 MPC8313E 嵌入式系统 UBoot 的移植

ARM：

1. 基于 DiskOnChip 2000 的驱动程序设计及应用
2. 基于 ARM 体系的 PC-104 总线设计
3. 基于 ARM 的嵌入式系统中断处理机制研究
4. 设计 ARM 的中断处理
5. 基于 ARM 的数据采集系统并行总线的驱动设计
6. S3C2410 下的 TFT LCD 驱动源码
7. STM32 SD 卡移植 FATFS 文件系统源码
8. STM32 ADC 多通道源码
9. ARM Linux 在 EP7312 上的移植
10. ARM 经典 300 问
11. 基于 S5PV210 的频谱监测设备嵌入式系统设计与实现
12. Uboot 中 start.S 源码的指令级的详尽解析
13. 基于 ARM9 的嵌入式 Zigbee 网关设计与实现
14. 基于 S3C6410 处理器的嵌入式 Linux 系统移植

Hardware：

1. DSP 电源的典型设计
2. 高频脉冲电源设计
3. 电源的综合保护设计
4. 任意波形电源的设计
5. 高速 PCB 信号完整性分析及应用
6. DM642 高速图像采集系统的电磁干扰设计
7. 使用 COMExpress Nano 工控板实现 IP 调度设备
8. 基于 COM Express 架构的数据记录仪的设计与实现
9. 基于 COM Express 的信号系统逻辑运算单元设计
10. 基于 COM Express 的回波预处理模块设计
11. 基于 X86 平台的简单多任务内核的分析与实现
12. 基于 UEFI Shell 的 PreOS Application 的开发与研究