

北京交通大学

硕士专业学位论文

面向 Linux 内核空间的内存分配隔离方法的研究与实现

Research and Implementation of Memory Allocation and Isolation
Method Oriented to Linux Kernel Space

作者：段鑫峰

导师：翟高寿

北京交通大学

2021 年 6 月

学位论文版权使用授权书

本学位论文作者完全了解北京交通大学有关保留、使用学位论文的规定。特授权北京交通大学可以将学位论文的全部或部分内容编入有关数据库进行检索，提供阅览服务，并采用影印、缩印或扫描等复制手段保存、汇编以供查阅和借阅。同意学校向国家有关部门或机构送交论文的复印件和磁盘。学校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。

（保密的学位论文在解密后适用本授权说明）

学位论文作者签名：段鑫峰

导师签名：程高寿

签字日期：2021年6月1日

签字日期：2021年6月1日

学校代码：10004

密级：公开

北京交通大学

硕士专业学位论文

面向 Linux 内核空间的内存分配隔离方法的研究与实现

Research and Implementation of Memory Allocation and Isolation
Method Oriented to Linux Kernel Space

作者姓名：段鑫峰

学 号：18140056

导师姓名：翟高寿

职 称：副教授

工程硕士专业领域：软件工程

学位级别：硕士

北京交通大学

2021 年 6 月

致谢

首先，要特别感谢翟高寿老师在选题阶段对我的帮助和信任，使我有机会接触、学习并探索系统安全里 Linux 内核安全这一块的内容。在整个研究过程和论文写作过程中，翟高寿老师初期给我提供了非常好的外文文献参考平台和相关学习课件，使我对 Linux 内核安全有了初步的认知。在中期进行相关内核实验的过程中，老师经常性地对我进行指导，并将每次讨论的内容都详细地记录在课题进展检查及讨论指导纪要里，方便我后续回顾总结，老师严谨的学术作风令我深深折服。在后期的论文写作工程中，老师对论文框架和内容也提出了有针对性的修改意见，帮助我更好地完成了学位论文。在此要向老师表示衷心的感谢。

其次，要感谢曾努力拼搏的自己，即使边工作边学习，还要备考公务员考试，但始终对研究结果精益求精，不畏艰难，勇于学习新知识。感谢每一次的成长，让我成为更好的自己。

再次，要感谢在这个过程中给予我帮助的学长，正是他们对我在 Linux 内核编译时出现的问题进行及时的解惑，才让我的研究得以顺利进行。他们对内核编译技巧的传授也使我少走了很多弯路。

然后，还要感谢研究生阶段我遇到的所有老师，正是基于平时点滴知识的传授才使我具备了一定的知识储备和编码能力，最终顺利完成学位论文的写作。感谢在此过程中一直陪伴着我的家人，对我提供的物质和精神上强有力的支持。

最后，感谢本文所有的审稿者，你们对本文提出的宝贵修改意见，使本文更加完善。

摘要

随着信息化时代的到来, 计算机核心地位日益凸显, 与之相关的系统安全也成为人们日益关注的焦点。对于主流开源的 **Linux** 操作系统而言, 其安全运行和管理控制的核心是内核。面对频繁的系统安全威胁, 尤其是内存损坏攻击, 如何有效地保护内核安全十分重要。内核隔离作为系统安全重要而热门的研究方向, 近些年受到广泛的关注。其中, 基于内存分配的内核隔离是主要解决途径之一。

本文首先针对国内外内核隔离的方法和相关技术进行了调研分析, 发现目前内核隔离主要采用基于进程、页表或虚拟机的内存隔离手段。其中, 基于页表构建独立执行空间, 完成内存分配隔离的思想得到广泛的应用。但在基于页表的实现中, 内核存在将属于主内核或不同内核模块的数据分配到同一物理页框上的混合页问题。这种混合页的存在, **Linux** 操作系统难以进行页级别的保护, 极易发生内存泄漏或缓冲区溢出等内存损坏攻击。然后, 本文围绕 **Linux** 内核内存分配机制展开深入的研究, 发现导致混合页问题的根本原因是 **Linux** 内核在进行内存分配时, 除使用常规的伙伴系统分配器外, 还使用一种可以称之为块分配器的小块内存分配器。因此, 为有效提升系统安全性, 解决混合页的问题, 必须对块分配器进行改进, 从而更好地实现主内核和内核模块在内存分配时的隔离。通过对 **Linux** 内核源码的分析, 本文发现在 **Linux** 内核空间中, 存在 **SLAB**、**SLUB** 和 **SLOB** 三种块分配器, 其中 **SLUB** 分配器是当前 **Linux** 内核默认块分配器。接着, 本文重点论述了如何改进 **SLUB** 分配器。具体而言, 改进后的 **SLUB** 分配器通过添加标识主内核和内核模块的分配标志, 并构建专用的内存缓存, 使内核可以根据分配标志调用专用的内存缓存完成相应的内存分配请求, 从而解决混合页的问题。在此基础上, 对伙伴系统分配器进行了相应的改进, 构建了面向 **Linux** 内核空间的内存分配隔离方法的新分配器处理模型。最后, 本文对新模型进行了实验检测和评估, 原型实验结果表明, 基于该方法构建的新模型可在不改变内核原有处理逻辑的前提条件下, 实现对主内核和内核模块在内存分配时的隔离, 并且对内核原有内存分配性能影响很小。

需要说明的是, 本文实现的新模型只是一种解决混合页问题的理论模型, 距离真正应用和集成到 **Linux** 内核并提供安全可靠的内存分配和内核隔离服务还需有不少的工作要做。

关键词: 系统安全; 内核隔离; 内存分配; 块分配器; 混合页

ABSTRACT

With the advent of the information age, the core position of computers has become increasingly prominent, and system security related to it has also arisen increasing attention. The core of its safe operation and management control is the kernel for the mainstream open-source Linux operating system. In the face of frequent system security threats, especially memory corruption attacks, effectively protecting kernel security is very important. As a critical and popular research direction for system security, kernel isolation has received extensive attention in recent years. Among them, kernel isolation based on memory allocation is one of the leading solutions.

This paper investigates and analyzes the methods and related technologies of kernel isolation at home and abroad and finds that the current kernel isolation mainly uses memory isolation methods based on processes, page tables or virtual machines. Among them, the idea of building an independent execution space based on page tables and completing memory allocation and isolation has been widely used. However, in the implementation based on the page table, the kernel has a mixed page problem in which data belonging to the main kernel or different kernel modules are allocated to the same physical page frame. With such mixed pages, it is difficult for the Linux operating system to perform page-level protection, and it is highly prone to memory corruption attacks such as memory leaks or buffer overflows. Then, this article carried out in-depth research around the Linux kernel memory allocation mechanism and found that the root cause of the mixed page problem is that the Linux kernel uses a conventional partner system allocator and the conventional partner system allocator, which can be called a block. The allocator is a small memory allocator. Therefore, in order to effectively improve system security and solve the problem of mixed pages, the block allocator must be improved to better realize the isolation between the main kernel and the kernel module during memory allocation. Through the analysis of the Linux kernel source code, this article found that in the Linux kernel space, there are three block allocators, SLAB, SLUB and SLOB, among which SLUB allocator is the default block allocator of the current Linux kernel. Next, this article focuses on how to improve the SLUB allocator. Specifically, the improved SLUB allocator adds allocation flags that identify the central core and kernel modules and builds a dedicated memory cache so that the kernel can call the dedicated memory cache to complete the corresponding memory allocation request according to the allocation flag, thereby solving the problem of

mixing page problem. On this basis, the partner system allocator has improved accordingly, and a new allocator processing model is constructed for the memory allocation isolation method of Linux kernel space. Finally, this paper conducts experimental testing and evaluation of the new model. The prototype experiment results show that the new model built based on this method can achieve the memory allocation of the central core and core modules without changing the original processing logic of the core. The isolation and has little impact on the original memory allocation performance of the kernel.

It should be noted that the new model implemented in this article is only a theoretical model to solve the mixed page problem. There is still much work before the application and integration into the Linux kernel and providing safe and reliable memory allocation and kernel isolation services.

KEYWORDS: System security; Kernel isolation; Memory allocation; Block allocator; Mixed pages

目录

摘要	iii
ABSTRACT	iv
1 引言	1
1.1 研究背景	1
1.2 国内外研究现状	2
1.3 研究意义与技术路线	4
1.4 论文结构	5
2 基本原理	6
2.1 Linux 内核空间	6
2.2 内存损坏	7
2.3 传统内存隔离	7
2.4 Linux 内核物理内存的组织	8
2.4.1 页框	8
2.4.2 区域	9
2.5 内存分配	10
2.5.1 以页框为单位的分配	10
2.5.2 以字节为单位的分配	11
2.5.3 分配标志位	11
2.6 分配器	14
2.6.1 伙伴系统分配器	14
2.6.2 块分配器	15
2.7 本章小结	17
3 面向 Linux 内核空间的内存分配隔离方法	18
3.1 基于 SLUB 分配器的内核空间内存分配隔离方法	18
3.1.1 SLAB 分配器中内存缓存的数据结构	18
3.1.2 SLUB 分配器中内存缓存的数据结构	20
3.1.3 整体设计	21
3.2 基于 SLUB 分配器的内核空间内存分配隔离机制的实现	23
3.2.1 分配策略	23

3.2.2	创建新的分配标志	24
3.2.3	建立专用 kmem_cache 结构数组	26
3.2.4	修改接口函数	29
3.3	基于伙伴系统分配器的内核空间内存分配隔离方法	31
3.4	完整的面向 Linux 内核空间的内存分配隔离方法	33
3.5	本章小结	34
4	实验结果分析	35
4.1	实验环境	35
4.2	实验方法	35
4.2.1	sysfs 文件系统	35
4.2.2	实验过程	36
4.3	讨论	42
4.4	本章小结	43
5	总结与展望	44
5.1	工作总结	44
5.2	研究展望	45
	参考文献	46
	图表索引	48
	作者简历及攻读硕士学位期间取得的研究成果	49
	独创性声明	50
	学位论文数据集	51

1 引言

1.1 研究背景

随着信息化时代的到来，计算机得到了广泛的应用。操作系统是计算机的重要组成部分，其核心地位日益凸显。Windows 和 Linux 操作系统是目前主要使用的操作系统，相比于 Windows 操作系统的办公高效性、操作简便性，Linux 操作系统以其开源免费性^[1]、安全稳定性，成为高校科学研究平台或网络安全设备的首选操作系统。

在 Linux 操作系统中，内核是程序运行和管理硬件设备的核心^[2]，它的存在使计算机系统能正常稳定地运行。出于安全的考虑，Linux 操作系统对自身内存空间进行了划分，并规定了不同的特权级。在此基础上，Linux 操作系统将内核置于区别于用户空间的受保护的内核空间，并赋予内核高级别的访问控制权限。

但由于内核采用不安全的程序设计语言如 C/C++ 编写，此类语言缺乏对未初始化内存、数组越界等错误的检查机制^[3]；且内核代码规模庞大，结构复杂，无论是资深的内核研发程序员，还是强大的代码分析辅助工具，都很难在内核发生问题的第一时间精确检测出问题所在；加之内核包含大量的中断异常入口点和系统调用点等攻击窗口和第三方内核模块，使得内核中存在着内存泄露、缓冲区溢出等漏洞和错误^[4]，容易引发攻击者任意执行代码、恶意操作设备、随意读取重要数据等提权攻击。

近年来，为了提升内核运行时的安全性，许多研究课题针对内核展开，且取得了一定的研究成果。从安全需求的角度而言，现有的系统安全工作可分为内核完整性保护、内核监控、内核最小化（微内核）、应用程序保护和内核隔离五类^[5]。内核完整性保护包含对控制流完整性（Control Flow Integrity, CFI）^[6]、代码指针完整性（Code-Pointer Integrity, CPI）^[7]和数据流完整性（Data Flow Integrity, DFI）^[8]的保护。内核监控通过使用一个安全的内核监控器，基于特定的私密性或完整性安全策略，实时监测内核行为和运行状态。内核最小化旨在提供内核的精简版本，只保留内核的线程调度、资源管理等核心基本功能，或缩小内核代码规模，移除非必需代码，减少内核攻击点。应用程序保护通过软硬件相结合的手段，确保提供给敏感应用程序的均为可信安全接口。内核隔离主要将主内核和内核模块进行安全隔离，减少更易遭受攻击、安全漏洞更多的内核模块对主内核带来的安全威胁。其中，内核隔离是当前最主要、应用最广泛、研究成果最丰富的研究方向。

1.2 国内外研究现状

当前系统安全面临的最大威胁就是内存损坏攻击。许多研究团体为此付出了大量的努力，也提出了许多有效的防御技术。虽然研究方向不尽相同，但归根结底，无论是哪种隔离或保护的方法，其本质都是为了有效区分内核可信部分和不可信部分，确保特定的安全操作仅能由可信部分进行。因此，从某种程度上而言，系统安全的最终落脚点往往都是要实现内核隔离。近些年，国内外均有科研人员进行内核隔离这方面的研究。

X. Xiong et al. (2013) 设计了一个提供透明保护域原语的 SILVER 框架^[9]，通过跟踪内核空间中数据对象的安全属性，如所有者主体和完整性级别等，建立了基于动态数据对象的元数据安全存储管理方案，有效地获取了细粒度的访问控制权限。与此同时，SILVER 框架引入了基于系统统一完整性模型的安全保护域原语，可有效保护操作系统内核与模块之间进行安全的数据通信，相关的保护域原语允许开发人员显式定义单个程序数据的安全属性和完成控制特权的委派，在给数据的传输和服务的导出带来很大灵活性的同时，减少了内核模块通过传递某些恶意参数而进行内核攻击，实现了主内核和内核模块之间安全有效的隔离。

R. Nikolaev et al. (2013) 认为很多时候操作系统为用户进程提供保护和隔离，但不为关键系统组件（即内核模块，例如设备驱动程序模块）提供保护和隔离是导致这些组件中错误频发，很容易带来内核安全隐患的重要原因。为此他们设计了基于内核隔离思想的 VirtuOS 系统^[10]。VirtuOS 系统利用虚拟化技术来隔离和保护单独的服务域，每个服务域代表一个内核现有的分区，并实现了内核功能的子集，如存储功能和网络服务功能等。VirtuOS 系统在完成原本需要内核系统调用才能完成的工作时，只需要使用服务域提供的接口即可完成相关的内核服务，从而避免内核频繁的系统调用和数据交互。基于 Xen 虚拟机管理程序的 VirtuOS 整体上实现了一种高效的内核组件和主内核隔离的模型，通过服务域的安全保护机制，避免了某个服务域故障威胁整个系统安全情况的发生。

D. Oliveira et al. (2014) 指出，在现代操作系统内核中，内核模块占据了很大一部分，在 Linux 中约为 70%。虽然内核模块的设立对内核功能进行了有效的扩展，诸如设备驱动程序模块等在一定程度上为 I/O 设备间的便捷通信提供了条件，但不可否认的是，内核模块本身存在的安全漏洞，很容易对内核带来安全威胁。因此，论文作者提出一种 Linux 系统下基于“免疫”范式的 Ianus 原型^[11]，并在 Bochs 这一硬件平台模拟器上对其进行了验证。实验结果表明，该原型成功地最小化了内核模块和主内核间的交互。

D. Evtyushkon et al. (2015) 提出了一种灵活的、由硬件支持的 Iso-X 框架^[12]。

Iso-X 提供页级别的细粒度隔离保护, 通过创建和管理隔离区, 来实现代码和重要数据的托管。其具有灵活的内存分配机制及低复杂度的运行逻辑, 仅需要少量的附加硬件, 使用少量的新 ISA 指令, 即可有效管理隔离区。即使在不受信任的系统运行环境中, 使用 Iso-X 框架的系统也可有效保障重要应用程序的安全性。

Y. Liu et al. (2015) 提出了 Secage 的内核隔离方法^[13]。该方法基于 Intel VT 支持的扩展页表 (Extended Page Tables, EPT) 实现, 通过构建两套 EPT, 即用于常规内存的 EPT-N 和受保护内存的 EPT-S, 实现内核中敏感可信数据与普通数据的隔离。两套 EPT 之间的快速切换由受信任的 vmfunc 调用指令完成, 从而有效保护敏感数据不被非法获取。

钱振江等 (2016) 指出了微内核和宏内核实现中各自存在的问题。微内核中的问题主要源于进程间通信, 由于通信双方不在同一内存空间, 频繁的消息复制带来的时间开销导致了微内核性能低下。宏内核虽然将主内核和载入模块置于同一内存空间, 便利了进程间通信, 提升了系统性能, 但却由此带来了安全问题。在此理论认知基础上, 他们提出了一种根据权能将主内核和载入模块隔离的 KCapISO 方案^[14]。该方案为主内核和载入模块分别构建和维护对应的页表, 借助 HybridHP 这一硬件化虚拟内存监控机制, 实现权限控制, 从而使载入模块只能访问自身页表, 而无法访问主内核页表, 也无法以任何方式直接调用主内核或跳转到内核空间执行。

邓良 (2016) 提出了四类同层可信基的方法^[14]。第一类方法基于硬件虚拟化实现, 在对敏感应用程序的请求进行验证后, 为其构建可信的执行空间, 从而保护应用程序的安全。第二类方法在结合 CPI 和 CFI 这两种内核完整性保护技术的基础上, 通过限定指令地址长度, 对内核映射地址的可访问区间进行限制, 实现对主内核的保护。第三类方法将地址空间隔离与传统的沙箱技术结合, 实现对不可信内核主动有效的监控。第四类方法利用 x86 的 NXE 和 WP 硬件机制, 实现对不可信内核主动有效的监控。四类方法均是在同一特权层进行可信基^[15]的构建, 这样做既提升了系统安全性, 又避免了因不同层之间频繁切换而导致的系统开销。

陈可昕等 (2016) 针对目前系统安全威胁主要来自内存泄露和缓冲区溢出的现状, 在论证了内存安全的重要性后, 提出了一种以保护安全敏感数据为目标的内存隔离方案^[3]。该方案提供了基于数据流的分析算法, 利用代码转化工具, 而非手动修改代码, 实现了由编译器自动进行代码转换和分析的功能。在程序指令执行的过程中, 该方案通过跟踪数据流动性和具体的操作指令, 将敏感数据代码和安全敏感数据本身与其他代码和数据分隔开, 并通过跳板函数实现敏感数据与普通数据间的交互, 以及敏感数据代码和普通数据代码编译时的切换。相关实验结果表明, 该方案能有效防止 Heartbleed 攻击。

胡志希等（2016）结合 Xen 虚拟机中内存安全管理机制和内存映射的特点，基于 XSM（一种 Xen 安全模块）的 Flask 框架，提出了一种可配置的特权域 Domain0 与普通域 DomainU 之间内存安全隔离的方法^[16]。

F. Tommaso et al.（2018）提出了一种基于 Intel x86 CPU 的轻量级进程内存隔离扩展技术 IMIX^[17]。该技术使用新的内存访问权限扩展了 x86 ISA 指令集，通过将内存页标记为安全敏感区，只有使用新引入的 smov 指令才能访问带有特殊标志的隔离页面，从而实现了内存分配时的隔离。

通过对国内外研究现状的调查与分析可知，当前内核隔离方案主要分为硬件隔离方案、软件隔离方案、软硬件相结合的隔离方案。以硬件为主的隔离方案提供基于系统级的隔离机制，由于硬件资源本身的封闭性和硬件支持的特异性，相关研究工作主要由硬件提供商独立开展或委托科研机构合作完成，典型的代表有 PC 领域的 Intel SGX^[18]和 AMD SVM，嵌入式领域的 Texas Instruments M-Sheild 和 ARM TrustZone^[19]。软件主导的隔离方案主要依托软件技术构建独立执行环境，以内存隔离作为核心，可用的隔离手段包括进程、页表和虚拟机。其中，基于页表构建独立执行空间，完成内存分配隔离的思想得到广泛的应用。由于页表存放有逻辑地址到物理地址的映射关系^[20]，其限制了进程可读地址范围，针对主内核和内核模块分别构建不同的内核页表，可有效控制内核模块的内存访问权限^[5]，实现主内核和内核模块的安全隔离。

1.3 研究意义与技术路线

本文相关的研究工作在前述认知基础上开展。通过对 Linux 系统内存分配机制的分析研究，本文发现在 Linux 内核空间中，Linux 内存分配的方式主要分为以页框为单位的分配和以字节为单位的分配。其中，以页框为单位的分配使用伙伴系统分配器，这种基于页粒度的分配方式，比较容易将主内核和内核模块隔离。而以字节为单位的分配使用块分配器。块分配器的出现，虽然既为内核运行提供了内存缓存，极大地提升了内核运行效率，又通过合理分配小块内存，减少了内存碎片的产生，提高了页表的利用率，但不可避免地带来了安全隐患。其根源在于块分配器在分配小块内存时，会将属于主内核或不同内核模块的数据分配到同一物理页框上，这种混合页的存在，难以进行页级别的保护，攻击者可通过触发可控溢出而实现提权攻击。目前，造成缓冲区溢出或内存泄露等安全威胁的原因正是由于在内存分配时，Linux 系统使用了这种块分配器。为了有效提升系统安全性，解决混合页问题，更好地实现主内核和内核模块在基于页表进行内存分配时的安全隔离，对块分配器的改进具有重要的理论意义和学术科研价值。

为此,本文面向 Linux 内核空间,对内存分配隔离方法进行了研究。研究工作的重点是对以字节为单位的小块内存分配时采用的块分配器进行改进,通过新增标识主内核和内核模块内存分配请求的分配标志,并构建专用的内存缓存,使内核在进行小块内存分配时,可以根据分配标志分别给主内核和内核模块分配对应的内存缓存,同一类内存缓存位于同一种页表中,即同一类内存分配请求对应同一种页表,从而有效解决混合页的问题;在此基础上,对以内存页为单位进行内存分配的伙伴系统分配器进行改进,使其可以根据分配标志限制内存分配请求可读取页表权限,使特定的页表只能由对应的分配请求访问,从而基于页表构建面向 Linux 内核空间的内存分配隔离方法的新分配器处理模型。基于该方法实现的新分配器处理模型,可实现主内核和内核模块在内存分配时的隔离。

1.4 论文结构

本论文正文由五章构成,各章节安排如下:

第一章:简述系统安全的研究背景,综述国内外关于内核隔离的研究现状,并对内存分配隔离的研究意义与本文采用的技术路线进行说明。

第二章:进行必要的技术背景和相关原理说明,提供研究工作开展的理论依据,以支持面向 Linux 内核空间的内存分配隔离机制的分析、设计和实现。本章首先概述了 Linux 内核空间的相关概念;其次对当前系统安全面临的主要威胁,即内存损坏攻击进行了说明;之后简述了传统内存隔离的段/页式管理技术;接着引出了 Linux 内核物理内存的组织;然后论述了内存分配相关的知识;最后对分配器进行了重点说明,细致讲解了伙伴分配器和块分配器。

第三章:本章首先结合实验采用的内核源码,完整分析了块分配器,给出了当前内核采用的 SLUB 分配器与之前内核采用的 SLAB 分配器的对比,并通过对内核源码的研读展示了两种块分配器的数据结构,在此基础上给出了基于 SLUB 分配器的内核空间内存分配隔离方法的整体设计框架;其次详细介绍了该方法的实现,主要包括新建分配标志、创建相应内存缓存数组以及完成相应接口函数修改等;然后给出了基于伙伴系统分配器的内核隔离方法;最后完整构建了面向 Linux 内核空间的内存分配隔离方法的新分配器处理模型。

第四章:本章对本文提出的新分配器处理模型进行原型测试评估,并结合其他研究人员的相关工作,进行了必要的讨论。

第五章:总结与展望,通过分析回顾整个研究过程,结合相关实验结果,对整个研究工作进行总结,在此基础上得出最终结论。同时,指出在进行研究的过程中或研究本身存在的一些不足,提出展望,对后续研究人员提供指导性意见。

2 基本原理

本章主要介绍研究涉及的基本原理和理论，通过对相关原理和理论的阐述，说明面向 Linux 内核空间的内存分配隔离方法研究工作的理论依据，以便在此基础上开展相应隔离机制的设计和实现。

2.1 Linux 内核空间

Linux 操作系统是一种支持多线程、多任务、多用户，基于模块化构建思想，内核可调度的类 Unix 开源操作系统。由于操作系统自身可用资源有限，Linux 操作系统引入了特权概念。根据特权等级，Linux 操作系统将自身运行状态分为了系统态和用户态。运行于系统态的核心程序区别于普通应用程序，其独立运行于特定的受保护的内存空间^[21]。这类核心程序具有高级别的特权权限，而特权级别越高，意味着调用系统资源和访问硬件设备优先级越高，对文件的可读写权限也越大。这类核心程序组成了 Linux 内核，其运行的受保护的内存空间被称为内核空间。

同样的，普通应用程序运行的内存空间被称为用户空间。用户空间相比于内核空间，由于其内存空间并不受保护，运行于其中的普通应用程序被 Linux 操作系统赋予了低级别的特权权限，而特权级别越低，意味着调用系统资源和特定的系统功能受到的限制越大，这类普通应用程序往往只能访问特定的内存范围和使用某些公用的系统调用功能，访问硬件设备时也可能因为权限过低而无法直接访问，除此之外，还有其他一些使用限制。

Linux 这种针对自身内存空间的划分原则一定程度上提升了系统安全性，有效避免了一定量恶意用户对于系统的不友好行为，使系统可以在大多数情况下稳定可靠运行。但必须指出，虽然 Linux 操作系统对自身内存空间进行了划分，但划分后的两种内存空间并不是完全独立的，二者往往需要协作完成任务，二者之间的通信通过内核空间中的系统调用接口完成，通过这种方式，普通应用程序可短暂运行在内核空间。同时，内核为了完成 Linux 操作系统的各项任务，往往离不开各种硬件设备的支持，通过对系统硬件设备的管理，内核合理地调度硬件资源，二者之间的通信通过内核空间中的设备驱动程序完成。

因此，为了满足内核与硬件设备和用户空间普通应用程序的交互，完整的 Linux 内核空间除包括内核子系统外，还应包括系统调用接口和设备驱动程序^[22]。而设备驱动程序的存在正是导致内核频繁遭受安全威胁的主要原因之一，所以对内核隔离的研究有着重要的意义。

2.2 内存损坏

内存损坏（Memory Corruption）是 Linux 内核编程中最棘手的错误之一。由于 C 和 C++ 的灵活性和高效性，它们被广泛用于 Linux 内核的编程设计。但这两种语言本身缺乏对指针使用后释放、数组越界等不安全未定义行为的错误检查机制，无法提供相应的内存保护；且整个编程设计对开发人员要求高，开发人员不仅需要维护庞大的内核代码，还往往需要手动进行内存管理，因此很容易导致内存损坏漏洞的出现，使攻击者能够实施提权攻击。

内存损坏主要表现为缓冲区溢出和内存泄露。缓冲区溢出是最常见的编程漏洞之一，如果在访问缓冲区期间缺少边界检查，导致攻击者能够操纵超出边界的相邻内存区域，有了相应写权限后，攻击者就能实现对目标不同级别的控制，如更改应用程序内的数据流或劫持控制流^[23]。当进行数据流攻击时，攻击者操纵在条件语句中使用的数据指针和变量泄露加密密钥之类的重要信息；当进行控制流劫持时，攻击者将覆盖随后用作间接映射目标地址的代码指针，更改控制流以执行代码注入操作或进行代码重用攻击。内存泄露一般是导致的错误，内存管理由于对读权限缺乏保护，攻击者可以根据该错误获取安全敏感信息。

研究团体针对内存损坏提出了许多有效的防御技术，如第一章提到的完整性保护就是在这种应用背景下发展起来的，此外，细粒度的代码随机化也是其中典型有效的缓解技术^[13]。但这些技术均离不开强大的内存隔离的支持，如要保护代码指针完整性的安全区域不被破坏、随机化密钥不被获取、控制流完整性的影子堆栈不受能执行任意读写访问权限对手的攻击^[24]，均需要做好内存隔离工作。

因此，为了解决当前 Linux 内核空间所面临的内存损坏攻击的挑战性问题，有必要针对内存隔离进行相应的分析、研究和改进。

2.3 传统内存隔离

传统内存隔离即 Linux 内核自带的内存隔离保护机制，其采用段/页式管理。分段和分页都为可由操作系统配置的内存访问建立了一个间接层，该间接层也被称为存储管理单元（MMU），处理器在使用间接层时强制执行访问控制权限。具体而言，处理器通过存储管理单元间接访问物理内存，在访问过程中，存储管理单元会根据某种策略判断当前操作是否具有访问权限，即是否合法，如果访问操作非法则访问失败，从而有效保护内存不因非法访问造成内存泄露。

分段允许开发人员定义由起始地址、大小和访问权限组成的段，但是在现代 64 位操作系统中，分段不再强制执行访问权限限制。

在现代操作系统中，分页会建立虚拟内存地址和物理内存地址之间的映射关系，从而实现间接寻址的功能。在这个过程中，分页采用的页表结构包含了转换信息和各种访问权限。为了将不同的进程彼此隔离，操作系统应确保每个进程都拥有自身独立的页表。

2.4 Linux 内核物理内存的组织

为了实现内存隔离，研究工作的重点是做好内存分配时的隔离。而要实现内存分配隔离，研究过程中首先要对 Linux 内核内存的组织情况进行深入的了解。

在 Linux 操作系统中，内存组织方式是 Linux 内核的核心内容之一。合理高效的内存组织方式有助于提升内核运行效率。

Linux 内核空间中关于内存的组织，主要分为针对虚拟内存的组织 and 针对物理内存的组织^[25]。针对虚拟内存的组织以进程空间虚拟页为单位，完成虚拟页的分配工作，以便实现内存映射，主要服务于进程。针对物理内存的组织在合理安排物理内存后，完成相应的管理工作，包括物理内存的分配、迁移和释放等工作。本文的研究工作主要针对物理内存的组织展开，下面将对研究工作涉及的物理内存的组织的相关理论进行详细论述。

Linux 内核采用页框（page frame，即内存页或物理页）、区域（zone）和节点（node）三级结构组织物理内存。节点主要说明当前架构为 NUMA 架构还是 UMA 架构，本章不对其进行讲解，重点关注页框和区域结构。

2.4.1 页框

内核进行物理内存的组织的基本单位是页框。虽然在处理器中，通常将字节作为最小可寻址单位，但对于完成内存组织工作，并负责完成虚拟地址到物理地址映射的内存管理单元而言，页框是其进行相应管理操作的基本单位。而如果从虚拟内存角度出发，页面（或称为页）即为最小单位^[26]。同一系统中，页框的大小与页面的大小保持一致。

对于不同体系结构的计算机而言，所支持页框的大小不完全相同。为了准确获知当前系统支持的页框大小，可通过 `getconf PAGE_SIZE` 命令查询。经查询，实验采用的 x86_64 结构该结果为 4096，即页框大小为 4 KB。

每个页框对应一个 `page` 结构体，该结构体在 `<include/linux/mm_types.h>` 中定义^[27]。其成员 `flags` 用来表示页框的状态，`page_to_nid()` 函数用于获取页框所属内存节点的编号，`page_zonenum()` 函数用于获取页框所属内存区域的类型。该结构与

虚拟页无关。内核可通过该结构获取页框信息，来判断某一页是否被分配，如果已被分配，则进一步判断是分配给了磁盘高速缓存（或称为页高速缓存），还是分配给了进程，其对应的是静态存放的内核代码还是动态执行的内核数据。

2.4.2 区域

因为某些历史原因和硬件设备自身缺陷，使得内核在进行内存寻址时会出现一些问题。这些问题主要是由于在某些计算机体系结构中虚拟寻址的范围与物理寻址的范围之间大小不匹配。这种大小差异带来的最直接的影响就是导致有些内存无法实现在内核空间的永久映射，以及有些设备无法直接访问所有内存。这些制约条件存在的情况下，内核中的页框并不是完全相同的。内核基于这一现实问题，对页框进行了分组，将页框归于不同的区域，同一区域中的页框具有相似的特性。

Linux 中的区域主要分为五种类型：

①DMA 区域（ZONE_DMA），该区域包含的物理页可执行 DMA（Direct Memory Access，直接内存访问）操作。DMA 区域的产生和出现，主要原因在于工业标准体系结构下的 ISA 设备只能直接访问 16 MB 以下的区域。

②DMA32 区域（ZONE_DMA32），该区域包含的物理页可执行 DMA 操作，且只有 32 位设备可访问此类页框。具体而言，对于 64 位系统，如果既要支持只能直接访问 16 MB 以下内存的设备，又要支持只能直接访问 4 GB 以下内存的 32 位设备，那么必须使用该区域。

③普通区域（ZONE_NORMAL），该区域指的是直接将物理内存映射到内核虚拟地址空间的内存区域，也被称为直接（或称线性）映射区域，该区域中包含的物理页均能正常映射。相应的内核虚拟地址和物理地址是线性映射的关系，即虚拟地址=（物理地址+常量）。

④高端内存区域（ZONE_HIGHEM），该区域中的页框无法与内核虚拟地址空间实现永久的映射关系。这是 32 位时代的产物，当时按一定比例划分内存空间，内核地址空间占据低内存段，而把不能直接映射的高内存段划分为高端内存区域。通常把 DMA 区域、DMA32 区域和普通区域统称为低端内存区域。

⑤可移动区域（ZONE_MOVABLE），该区域是一个伪内存区域，用来防止内存碎片。

区域的存在并不具有实际的物理意义，但这种逻辑上的分组方式，有利于内核对页框进行管理。Linux 内核根据区域的划分，形成相应的内存池，之后可根据具体的用途高效分配内存。分配有时并不是固定的，比如普通内存的分配既可由

ZONE_NORMAL 完成,也可由 ZONE_DMA 完成,但出于要将 ZONE_DMA 最大限度保留以满足未来 DMA 使用需求的考虑,除非迫不得已,一般建议普通用途的内存分配请求均由 ZONE_NORMAL 完成。此外,需注意分配不能同时跨区进行。

每个区域对应一个 zone 结构体,该结构体在<include/linux/mmzone.h>中定义。其成员 watermark 数组表示分配器使用的水线,通过水线设定了每个区域的合理内存消耗基准。成员 name 表示区域名称,相关代码位于<mm/page_alloc.c>中。

2.5 内存分配

在对 Linux 内核内存管理进行研究分析后,研究工作重点对 Linux 内核内存分配方式进行探究。

由于 Linux 内核的特殊性和重要性,内核空间区别于用户空间,简单的内存分配方式并不适用于内核内存分配。考虑到内核不能随意睡眠、内核对于内存分配错误敏感等特性, Linux 内核建立了一套特有的内存分配机制^[28]。

2.5.1 以页框为单位的分配

内核以页框为单位进行内存分配时,基于伙伴系统分配器实现,并提供了以下接口用于内存分配请求,相关声明位于<include/linux/gfp.h>中。

(1) alloc_pages (gfp_mask, order)

该函数请求分配一个阶数为 order 的内存页块,即分配 2^{order} ($1 \leq \text{order}$) 个连续的内存页,并返回一个 page 结构体的指针,该指针指向第一个内存页。

(2) alloc_pages (gfp_mask)

该函数是 alloc_pages()函数在阶数为 0 情况下的简化形式,只分配一个内存页。

(3) _get_free_page (gfp_mask, order)

该函数对 alloc_pages()函数进行了封装,直接返回分配到的第一个内存页的逻辑地址的指针,但只能从低端内存区域来分配内存页。程序在开始运行前应调用该函数进行内存分配错误检查,以免因内存分配失败导致已进行的工作前功尽弃。

(4) __get_free_page (gfp_mask)

该函数是 __get_free_page()函数在阶数为 0 情况下的简化形式,只分配一个内存页。

(5) get_zeroed_page(gfp_mask)

该函数是将 __get_free_page()函数的参数列表设置为 (__GFP_ZERO, 0),表示只分配一个内存页,并且用零初始化。

2.5.2 以字节为单位的分配

内核以字节为单位进行内存分配时，基于块分配器实现，分配的这些内存也被称为内存缓存（准确而言，应为内核内存缓存），并提供了相应的接口函数。其中，最主要的 `kmalloc()` 函数用于完成小块内存的分配，由该函数分配的内存也被称为通用的内存缓存，该函数定义在 `<include/linux/slab.h>` 中，其声明为：`void *kmalloc (size_t size, gfp_t flags)`。

在该函数中，`size` 表示需要的内存长度；`flags` 表示传给页分配器的分配标志位，当内存缓存没有空闲对象，需要向页分配器请求分配页用于内存缓存的分配时使用这个分配标志位。块分配器根据 `size` 大小，找到一块内存长度刚好大于或等于请求内存长度的内存缓存，然后从该内存缓存中分配对象。如果分配成功，返回分配得到的对象的指针，如果分配失败，返回 `NULL` 空指针。使用该函数所分配的内存块在物理层面上是连续的。

2.5.3 分配标志位

无论是在以页框为单位进行分配时，还是在以字节为单位进行分配时，内存分配都用到了分配标志位。这些标志按功能可划分为位置性的区域修饰符、操作性的行为修饰符和类型性的类型标志三类。所有的标志均在 `<include/linux/gfp.h>` 中声明。

（1）区域修饰符

区域修饰符解决从哪儿问题，即指定从哪个类型的区域分配物理页。由于内核对物理内存进行了划分，将其分为多个区，每个区都有不同的用途，内存分配时可根据区域修饰符获知当前分配首选区域类型。

一般而言，尽管内核在进行内存分配时可以从任意的一个区域开始，但实际上，内核的选择策略是优先分配 `ZONE_NORMAL` 区域，这样做的目的是保证其他区有充足的空白页以备不时之需。

传统区域修饰符有 `__GFP_DMA`、`__GFP_DMA32` 和 `__GFP_HIGHMEM` 三种（标志位名称中 `GFP` 是 `Get Free Pages` 的缩写），后又新增了 `__GFP_MOVABLE` 区域修饰符。指定区域修饰符中的一种就可以控制内核基于区域的内存分配行为。普通区域在 `x86` 体系结构中物理内存地址空间为 `16 MB~896 MB`，高端内存区域在 `x86` 中为 `896 MB~4 GB`^[29]。如果没有指定任何区域修饰符，则内核从 `DMA` 区域或普通区域进行内存分配，且优先从普通区域进行内存分配。区域修饰符及相关的功能描述如表 2-1 所示。

表 2-1 传统区域修饰符
Table 2-1 Traditional zone modifiers

区域修饰符	功能描述
<code>__GFP_DMA</code>	表示从 DMA 区域分配物理页。
<code>__GFP_DMA32</code>	表示只在 DMA32 区域分配物理页。
<code>__GFP_HIGHMEM</code>	表示从高端内存区域或普通区域分配物理页,且优先从高端内存区域分配物理页。
<code>__GFP_MOVABLE</code>	表示从可移动区域分配物理页。

(2) 行为修饰符

行为修饰符解决怎么做问题,即指定内核在进行内存分配时的具体方式。它的出现是因为对于不同的应用场景,需要一种特定的内存分配方法来满足这种特殊情况下的需要。常见的行为修饰符及相应的功能描述如表 2-2 所示。

表 2-2 行为修饰符
Table 2-2 Behavior modifiers

标志	功能描述
<code>__GFP_HIGH</code>	指明调用者是高优先级的,为了使系统能向前推进,必须准许这个请求。例如,创建一个 I/O 上下文,把脏页回写到存储设备。
<code>__GFP_ATOMIC</code>	指明调用者是高优先级的,不能回收页或睡眠。
<code>__GFP_IO</code>	允许读写存储设备。
<code>__GFP_FS</code>	允许向下调用底层文件系统。当文件系统申请页时,如果内存严重不足,直接回收页,并把脏页回写到存储设备,可能导致死锁。为了避免死锁,文件系统申请页的时候应该清除这个标志位。
<code>__GFP_RECLAIM</code>	允许直接回收页和异步回收页。
<code>__GFP_COLD</code>	调用者不期望分配的页很快被使用,应尽可能分配缓存冷页(数据不在处理器的缓存中)。
<code>__GFP_NOWARN</code>	分配器在分配失败后不输出失败警告。
<code>__GFP_REPEAT</code>	分配器在分配失败后自动重试,在失败次数达上限前成功或失败次数超过上限后放弃,分配允许失败。
<code>__GFP_NOFAIL</code>	分配器必须无限次重试,因为调用者不能处理分配失败。
<code>__GFP_NORETRY</code>	不要重试,当直接回收页和内存碎片整理分配失败时,应该放弃。
<code>__GFP_ZERO</code>	分配器在分配成功时,把页用零初始化。
<code>__GFP_HARDWALL</code>	实施 <code>cpuset</code> 内存分配策略。 <code>cpuset</code> 是控制组(cgroup)的一个子系统,提供了把处理器和内存节点的集合分配给一组进程的机制。

（3）类型标志

实际进行内存分配时，一般不直接指定行为修饰符，而是采用类型标志。

类型标志解决怎么用问题，即对区域修饰符和行为修饰符进行组合，降低修饰符的选用难度，通过将两类修饰符可能的组合进行归纳，总结出可完成各种需求的类型组合，并给每个这样的类型组合指定特定的标志，之后进行内存分配时，直接使用相应的类型标志即可完成需求的内存分配操作。

类型标志的出现，既简化了内存分配的过程，又一定程度上减少了内存分配时错误的发生。常见的类型标志及功能描述如表 2-3 所示。

表 2-3 类型标志
Table 2-3 Type flags

标志	功能描述
GFP_ATOMIC	原子分配，分配内核使用的页，指明调用者是高优先级的，为了使系统能向前推进，必须准许这个请求。
GFP_NOWAIT	对于内核分配而言，直接回收而不能延迟，也不能启动物理输入输出、也不能使用任何文件系统回调机制。
GFP_NOIO	采用直接回收方式来放弃干净的页面或那些无需启动任何物理输入输出的 slab 页面。
GFP_NOFS	不允许调用到底层文件系统，允许异步回收页和直接回收页，允许读写存储设备。
GFP_KERNEL	分配内核使用的页，可能睡眠。调用者请求 ZONE_NORMAL 或低址区域以直接访问，但也可直接回收。调用者在分配期间可以睡眠、对换或刷新一些页面到硬盘。该标志为首选标志。
GFP_USER	分配用户空间使用的页，内核或硬件也可以直接访问，从普通区域分配，允许异步回收和直接回收页，允许读写存储设备，允许调用到文件系统，允许实施 cpuset 内存分配策略。
GFP_HIGHUSER	分配用户空间使用的页，内核不需要直接访问，从高端内存区域分配，物理页在使用的过程中不可以移动。
GFP_DMA	从 ZONE_DMA 进行分配，一般与其他类型标志组合使用

类型标志与关联修饰符的对应关系如表 2-4 所示。因为在实际分配时，普通区域即可满足一般需求，所以在类型标志的关联修饰符中，一般不需要指明相应的区域修饰符。

表 2-4 类型标志与修饰符的关联关系
Table 2-4 Association between Type flags and modifiers

标志	关联修饰符
GFP_ATOMIC	__GFP_KSWAPD_RECLAIM __GFP_HIGH __GFP_ATOMIC
GFP_NOWAIT	__GFP_KSWAPD_RECLAIM
GFP_NOIO	__GFP_RECLAIM
GFP_NOFS	__GFP_RECLAIM __GFP_IO
GFP_KERNEL	__GFP_RECLAIM __GFP_IO __GFP_FS
GFP_USER	__GFP_RECLAIM __GFP_IO __GFP_FS __GFP_HARDWALL
GFP_HIGHUSER	GFP_USER __GFP_HIGHMEM
GFP_DMA	__GFP_DMA

2.6 分配器

2.6.1 伙伴系统分配器

内核初始化完毕后，由页分配器负责接下的物理页管理与分配工作。当前页分配器使用的是伙伴系统分配器。伙伴系统分配器的特点是算法简单且效率高。

在伙伴系统分配器中，连续的物理页称为页块（page block）。伙伴系统分配器中另一个重要的概念是阶（order）。阶是页的数量单位， 2^n 个连续页称为 n 阶页块。满足以下条件的两个 n 阶页块称为伙伴（buddy）：①两个页块是相邻页块，即具有连续的物理地址；②页块的第一页的物理页号必须是 2^n 的整数倍；③如果合并成 $(n+1)$ 阶页块，第一页的物理页号必须是 2^{n+1} 的整数倍^[30]。基于伙伴构建的系统被称为伙伴系统（buddy system）。以单页为例，0 号页和 1 号页是伙伴，2 号页和 3 号页是伙伴，1 号页和 2 号页不是伙伴，因为 1 号页和 2 号页合并组成一阶页块，第一页的物理页号不是 2 的整数倍。

伙伴系统分配器分配和释放物理页的数量单位是阶。分配 n 阶页块的过程为：①查看是否有空闲的 n 阶页块，如果有，直接分配；如果没有，继续执行下一步。②查看是否存在空闲的 $(n+1)$ 阶页块，如果有，就把 $(n+1)$ 阶页块分裂为两个 n 阶页块，一个插入空闲 n 阶页块链表，另一个分配出去；如果没有，继续执行下一步。③查看是否存在空闲的 $(n+2)$ 阶页块，如果有，把 $(n+2)$ 阶页块分裂为两个 $(n+1)$ 阶页块，一个插入空闲 $(n+1)$ 阶页块链表，另一个分裂为两个 n 阶页块，一个插入空闲 n 阶页块链表，另一个分配出去；如果没有，继续查看更高

阶是否存在空闲页块。释放 n 阶页块时，查看它的伙伴是否空闲，如果伙伴不空闲，那么把 n 阶页块插入空闲的 n 阶页块链表；如果伙伴空闲，那么合并为 $(n+1)$ 阶页块，接下来释放 $(n+1)$ 阶页块。

内核在基本的伙伴系统分配器上做了一些扩展以便更好地完成内核功能，具体表现在以下方面：①支持内存节点和区域，构成分区的伙伴系统分配器；②为了预防内存碎片的产生，将物理页根据可移动性进行分组；③为减少处理器之间的锁竞争，分配单页时做了性能优化，在内存区域增加一个每处理器页集合，并采用 `per_cpu_pageset` 结构体描述。

分区的伙伴分配器相关声明存放在 `<include /linux /mmzone.h>` 中。内存区域的结构体 (`struct zone`) 成员 `free_area` 用来维护空闲页块，数组下标对应页块的阶数。成员 `managed_pages` 是伙伴分配器管理的物理页的数量。定义的 `MAX_ORDER` 常量表示最大阶数，实际上是可分配的最大阶数加 1，默认值是 11，这也意味着伙伴分配器一次最多可以分配 2^{10} 页，即 1024 页。

2.6.2 块分配器

在内核运行过程中，相比于以页框为单位的内存分配和释放，内核中更多进行的是基于数据结构的内存分配和释放。这些数据结构往往很小，但却频繁地被调用，内核将这些数据结构组织起来放入空闲链表中，当程序运行过程中请求相应结构的实例时，就可以直接从空闲链表中分配，而不再需要现场分配内存。当不再需要时，就将其放回空闲链表中。此时空闲链表相当于对这些频繁使用的数据结构建立了对象内存缓存，以帮助内核快速调用相应的对象类型^[31]。

空闲链表虽然既满足了快速分配频繁使用的数据结构的需求，减少内存现场分配带来的时间开销，又因为空闲链表在存储的时候连续存放，已被大量使用并处理完的数据结构重新释放回空闲链表空间，降低了内存碎片的总体体量，但因为其不能全局控制的缺陷，导致内核在内存紧张时，无法控制空闲链表收缩缓存大小以便释放更多的内存满足必要的需求^[32]。

在此背景下，Linux 内核在伙伴系统分配器的基础上，又提供了用于小块内存分配的块分配器。其中最原始的块分配器是 SLAB 分配器，从内核代码 3.x 开始，新增了 SLUB 分配器和 SLOB 分配器。

SLAB 分配器不仅能分配小块内存，更重要的是它可充当经常分配和释放对象的缓存。SLAB 分配器为每种对象类型创建一个内存缓存，每个内存缓存由多个 slab（大块，原意是大块的混凝土，可形象化描述内存块）组成，一个 slab 是一个或多个连续的物理页。每个 slab 都包含多个对象，这里的对象特指被缓存的数据

结构。内存缓存的组成如图 2-1 所示。

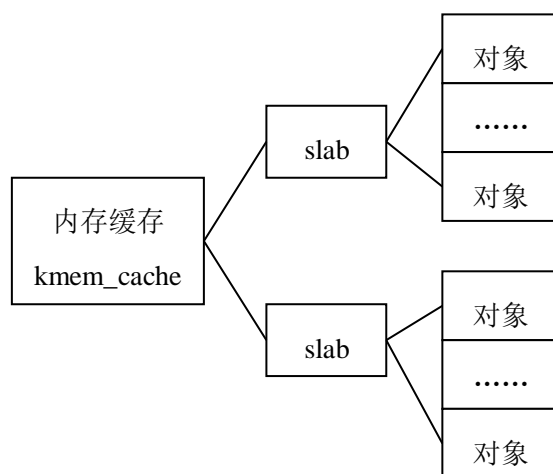


图 2-1 内存缓存的组成

Figure 2-1 The composition of kmem_cache

每个 slab 有满、部分满或空三种状态。当 slab 已将自身所含的所有对象都分配出去时，slab 处于满的状态，无法满足后续分配请求。当 slab 仅将一部分对象分配出去，还保留一些空闲对象时，slab 处于部分满的状态，可满足后续分配请求。当 slab 未分配任何对象时，slab 处于空的状态，此时可满足后续分配请求。为了减少内存碎片，内核在实际分配时，应优先分配部分满的 slab，没有部分满的 slab，就分配空的 slab，没有空的 slab，就创建一个新的 slab。新 slab 通过上文提到的 `_get_free_pages()` 页分配函数实现。

如对于 inode 结构而言，该结构是索引节点对象，其会在内核中被频繁地创建和释放，此时可采用 SLAB 分配器管理该对象。SLAB 分配器为 inode 结构创建 `inode_cachep` 内存缓存，这种内存缓存包含一个或多个（一般为多个）slab，每个 slab 包含大量的 `struct inode` 对象。当内核发出分配请求，需要调用新的 inode 结构时，内核就参照上一段提及的分配策略进行分配，分配成功返回相应的指针。当该 inode 对象使用完毕后，分配器就把相应对象的状态标记为空闲。

slab 的出现，可有效弥补内存紧缺时空闲链表无法动态调整缓存大小的缺陷。完成 slab 释放的函数是 `kmem_freepages()`，当内存缓存被显示撤销，或内存变得紧缺需要 slab 释放内存以供使用时，内核将调用该函数。

SLAB 分配器在某些情况下表现不太好，所以 Linux 内核提供了两种改进的块分配器：①在配备了大量物理内存的大型计算机上，SLAB 分配器的管理数据结构的内存开销比较大，所以设计了 SLUB 分配器。②在小内存的嵌入式设备上，SLAB 分配器的代码既多又复杂，所以设计了一种精简的 SLOB 分配器。由于 SLOB 分配器主要用于嵌入式设备，在研究工作中并未涉及，在此不再进行相关说明。

2.7 本章小结

本章主要介绍了面向 Linux 内核空间的内存分配隔离方法的相关技术背景 and 理论。

本章首先阐述了 Linux 内核空间的概念。简要交代了内核安全的相关背景，正是因为内存损坏等不安全因素的存在，为了保护内核，相关的系统安全工作才具有实际意义，这些研究工作大多离不开内存隔离的支持，内存隔离作为一种强大有效的内核安全隔离防御技术，本文的研究工作也围绕该方向展开，为此本章对传统内存隔离即 Linux 内核自带的段/页式内存隔离保护机制做了相关说明。

因为内存隔离一个很重要的方面是要做到内存分配隔离，本文的研究工作为实现内存隔离，进而实现将主内核和内核模块分隔开的内核隔离保护机制，即采用了内存分配隔离的思想，而要实现内存分配隔离，必须事先掌握内存组织的相关内容，所以本文接着对 Linux 内核物理内存组织形式进行了讲解，并对物理内存三级组织结构中的页框和区域进行了详细介绍。

本章之后分析了 Linux 内核内存分配机制。Linux 内核内存分配主要分为以页框为单位的内存分配和以字节为单位的内存分配，由于二者在分配过程中均使用了分配标志位，本文后续的研究工作正是基于这一点重要发现而展开，为此本章对分配标志位进行了详细说明。

本章最后重点介绍了 Linux 内核中的分配器，因为内存分配离不开分配器的支持，本文的研究工作最终目的也是为了改进分配器，使改进后的分配器最终能有效实现基于内存分配隔离方法的主内核和内核模块的隔离，所以本章最后对 Linux 内核中的两类分配器，即伙伴系统分配器和块分配器，结合相关代码进行了详细论述。

3 面向 Linux 内核空间的内存分配隔离方法

本章详细论述基于 SLUB 和伙伴系统分配器的面向 Linux 内核空间的内存分配隔离方法，并对具体实现过程进行了细致的说明。

3.1 基于 SLUB 分配器的内核空间内存分配隔离方法

对于 Linux 内核而言，其内核版本一直在更新，涉及内核代码更改的工作，必须建立在大量内核源码分析的基础上。本文实验所采用的内核版本为 4.13.2，通过对内核源码的分析，尤其是对 slab.c、slub.c 和 slab_common.c 三个源文件的分析，本文得出了 SLAB 分配器和 SLUB 分配器中各自内存缓存的数据结构，并通过分析内核函数间的调用关系，明确了该版本中默认使用的块分配器为 SLUB 分配器。

3.1.1 SLAB 分配器中内存缓存的数据结构

SLAB 分配器中内存缓存的数据结构如图 3-1 所示。

(1) 每个内存缓存对应一个 kmem_cache 实例

kmem_cache 结构中成员 size 是包括填充对象的总长度，成员 object_size 是对象的原始长度，成员 num 是每个 slab 包含对象的数量，成员 gfporder 是 slab 的阶数，成员 node[] 指针数组指向 kmem_cache_node 结构。

(2) 每个内存节点对应一个 kmem_cache_node 实例

kmem_cache_node 结构包含三个 slab 链表：链表 slabs_partial 把部分对象空闲的 slab 链表链接起来，链表 slabs_full 把没有空闲对象的 slab 链表链接起来，链表 slabs_free 把所有对象空闲的 slab 链表链接起来。kmem_cache_node 结构中成员 total_slabs 是 slab 总数量。

(3) 每个处理器对应一个 array_cache 实例

kmem_cache 结构中成员 cpu_slab 指针指向 array_cache 结构，该结构也被称为数组缓存，用来缓存刚被释放的对象，分配时首先从当前处理器的数组缓存分配，避免每次都要从 slab 分配，从而减少锁操作和链表操作，提高分配速度。array_cache 结构中成员 avail 是 entry[] 指针数组存放的对象数量，成员 limit 是 entry[] 指针数组的大小，entry[] 指针数组存放对象的地址。

(4) 每个 slab 对应一个或多个 page 实例

在 slab 对应的 page 结构体中，成员 flags 设置标志位 PG_slab，表示页属于 SLAB

分配器；成员 s_mem 存放 slab 第一个对象的地址；成员 active 表示已分配对象的数量；成员 lru 作为链表节点加入其中一条 slab 链表；成员 slab_cache 指针指向 kmem_cache 实例；成员 freelist 指针指向空闲对象链表。

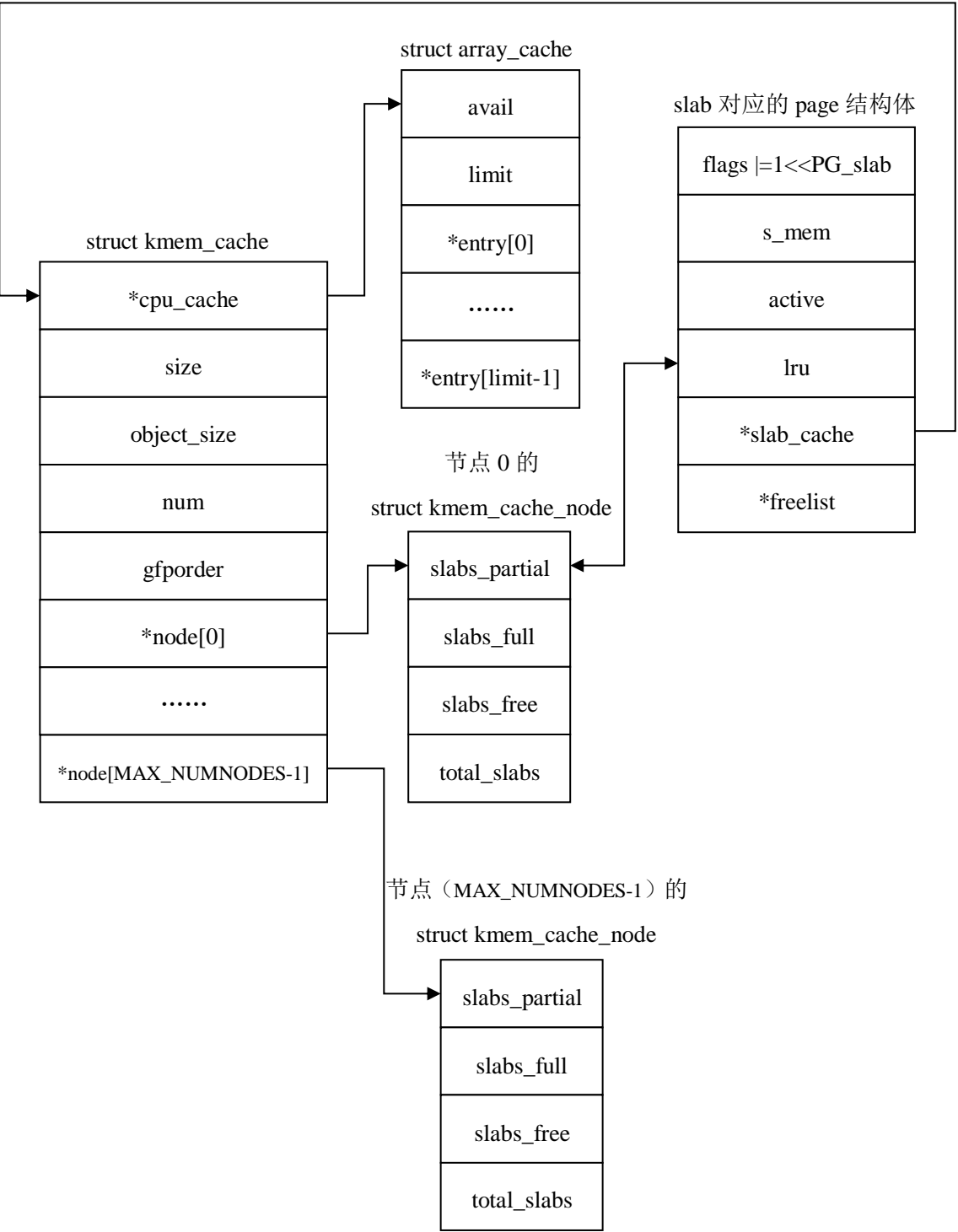


图 3-1 SLAB 中内存缓存的数据结构
Figure 3-1 Data structure of kmem_cache in SLAB

3.1.2 SLUB 分配器中内存缓存的数据结构

SLUB 分配器中内存缓存的数据结构如图 3-2 所示。

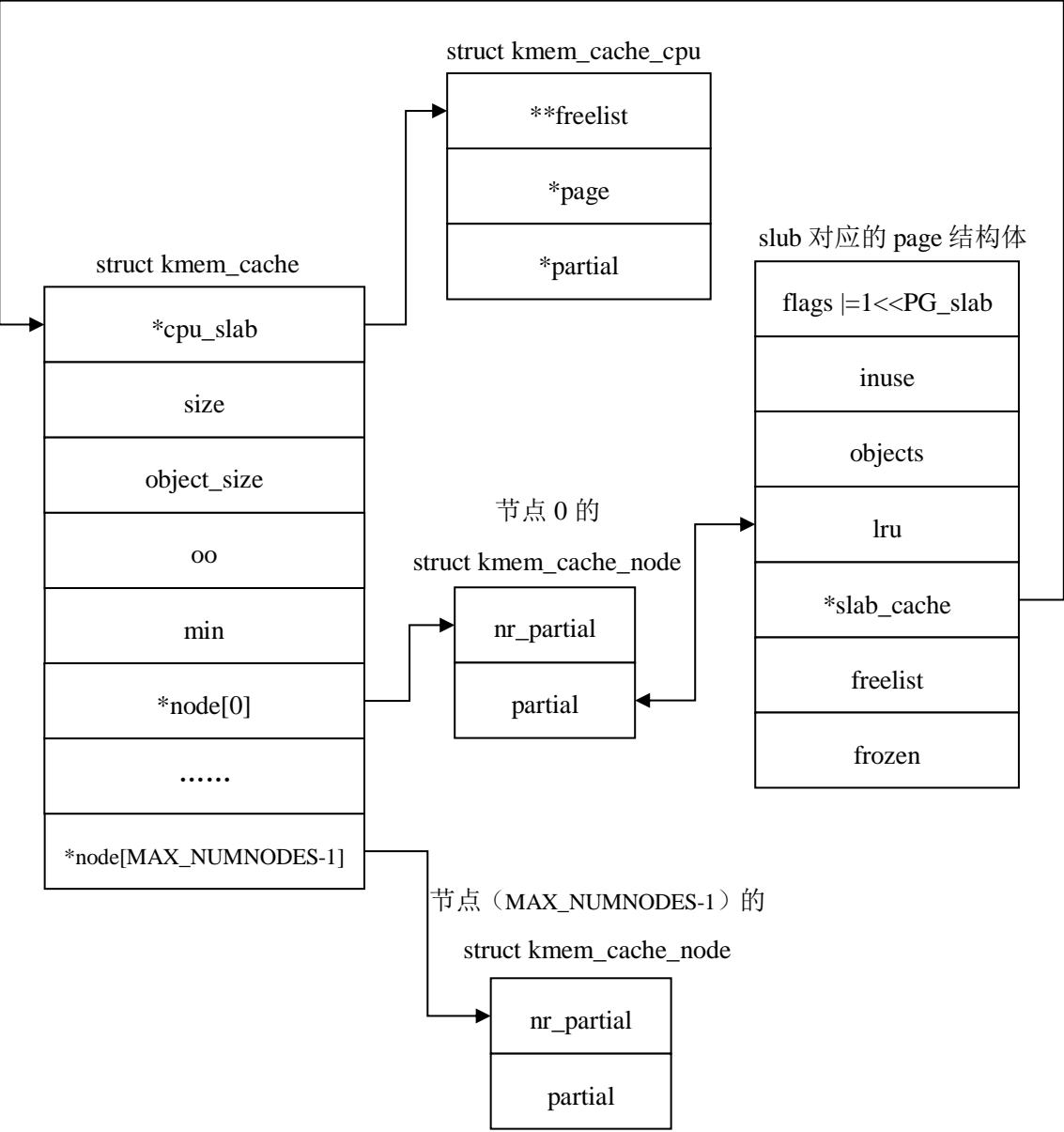


图 3-2 SLUB 中内存缓存的数据结构
Figure 3-2 Data structure of kmem_cache in SLUB

(1) 每个内存缓存对应一个 kmem_cache 实例

kmem_cache 结构中成员 size 是包括元数据对象的总长度；成员 object_size 是对象的原始长度；成员 oo 存放最优 slab 的阶数和对象数，低 16 位是对象数，高 16 位是 slab 的阶数，即 oo 等于 ((slab 的阶数<<16) |对象数)，最优 slab 是剩余部分最小的 slab；成员 min 存放最小 slab 的阶数和对象数，格式和 oo 相同，最小

slab 只需要足够存放一个对象;成员 `node[]` 指针数组指向 `kmem_cache_node` 结构。

(2) 每个内存节点对应一个 `kmem_cache_node` 实例

链表 `partial` 把部分空闲 slab 链接起来,成员 `nr_partial` 是部分空闲 slab 的数量。

(3) 每个处理器对应一个 `kmem_cache_cpu` 实例

`kmem_cache` 结构中成员 `cpu_slab` 指针指向 `kmem_cache_cpu` 结构,该结构也称为每处理器 slab 缓存。`kmem_cache_cpu` 结构中成员 `freelist` 二级指针指向当前使用的 slab 的空闲对象链表,成员 `page` 指针指向当前使用的 slab 对应的 `page` 实例,成员 `partial` 指针指向每处理器部分空闲 slab 链表。

(4) 每个 slab 对应一个或多个 `page` 实例

在 slab 对应的 `page` 结构体中,成员 `flags` 设置标志位 `PG_slab`,表示页属于 SLUB 分配器;成员 `inuse` 表示已分配对象的数量;成员 `objects` 是 slab 的对象数量;成员 `lru` 作为链表节点加入部分空闲 slab 链表;成员 `slab_cache` 指针指向 `kmem_cache` 实例;成员 `freelist` 指针指向第一个空闲对象,成员 `frozen` 表示 slab 是否被冻结在每处理器 slab 缓存中。

由上文对 SLAB 和 SLUB 数据结构的分析可知,SLUB 分配器继承了 SLAB 分配器的核心思想,基本的底层逻辑和函数实现大致相同,并在某些方面进行了改进。主要的变化体现在 SLUB 分配器中只保留了部分空闲 slab 链表;SLAB 分配器的每处理器缓存以对象为单位,而 SLUB 分配器的每处理器 slab 缓存以 slab 为单位。

3.1.3 整体设计

在对块分配器进行深入研究,明确当前 Linux 内核默认分配器为 SLUB 分配器后,为了解决混合页的问题,对 SLUB 分配器改进的整体思路框架如图 3-3 所示。其处理思想如下。

首先,对于调用块分配器的内存分配请求,不同于以往采用 SLUB 分配器直接分配,改进后的 SLUB 分配器通过判定分配标志位中是否同时包含或不包含本文新增的两种分配标志,即代表主内核分配请求的 `GFP_COME_FROM_KERNEL` 分配标志和代表内核模块分配请求的 `GFP_COME_FROM_MODULE` 分配标志,来决定后续的内存分配操作。如果内存分配请求来源于本文第二章中提到的内核自带的 `kmalloc()` 接口函数,其并未包含新增的分配标志,继续使用内核原有的通用 `kmem_cache` 结构。如果内存分配请求分配标志位中同时包含两种分配标志,说明此时内核因为某种特殊需求,应同时满足来源于主内核和内核模块的内存分配需求,此时不再对分配来源进行区分,也继续使用内核原有的通用 `kmem_cache` 结构。

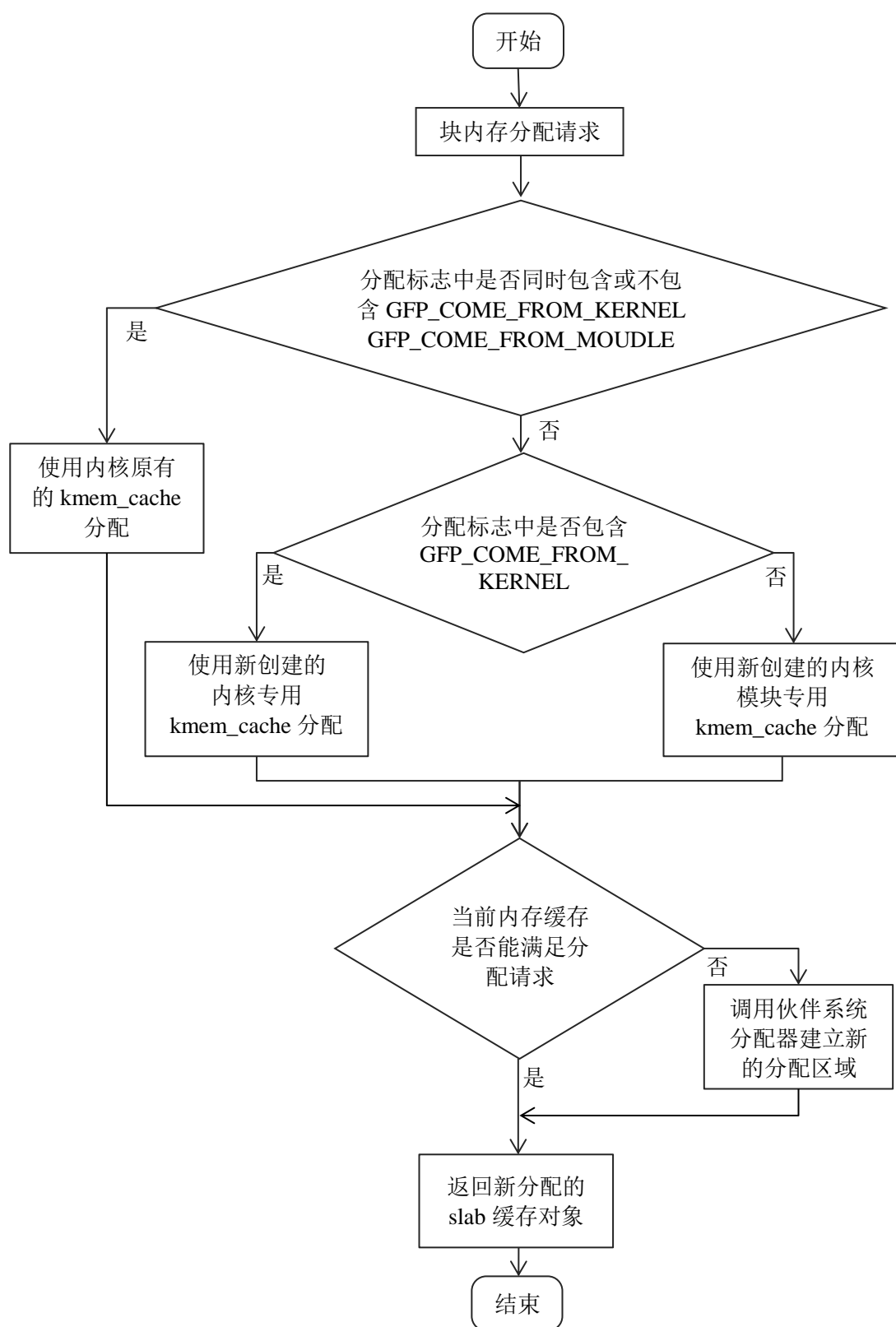


图 3-3 SLUB 分配器改进的方法框架
Figure 3-3 Improved method framework of SLUB allocator

其次，对于内存分配请求分配标志位只含有一种分配标志的情况，通过判断属于哪种分配标志，使用本文新建立的专用 `kmem_cache` 结构数组完成内存分配。

最后，本方法对内存分配可能失败的情况进行了考虑，因为在 `SLUB` 分配器中三种内存分配情形都用到了内存缓存，容易使内存缓存区域在分配一段时间后无法满足分配请求，所以应及时检查当前内存缓存是否能满足分配请求。如果无法满足，则调用伙伴系统分配器建立新的内存缓存区域，继续之后的内存分配操作。如果满足，在已分配的 `kmem_cache` 基础上，调用 `kmem_cache` 对应的 `slab`，完成分配请求并返回分配到的内存地址。

3.2 基于 `SLUB` 分配器的内核空间内存分配隔离机制的实现

3.2.1 分配策略

`SLUB` 分配器基本的底层逻辑是为每种对象类型创建一个内存缓存，每个内存缓存由多个 `slab` 组成。每个内存缓存对应一个 `kmem_cache` 结构，这种结构也被称为通用的内存缓存，其内存长度由 `kmalloc()` 函数在分配的时候指定。这些通用的内存缓存分为两类，一类是从普通区域分配页的内存缓存，对应的名称是“`kmalloc-<size>`”（`size` 是内存长度，单位是字节），另一类是从 `DMA` 区域分配页的内存缓存，对应的名称是“`dma-kmalloc-<size>`”，这些通用的内存缓存可通过执行“`cat /proc/slabinfo`”命令看到，如图 3-4 所示。

<code>dma-kmalloc-8192</code>	0	0	8192	4	8 : tunables	0	0	0 : slabdata
<code>dma-kmalloc-4096</code>	0	0	4096	8	8 : tunables	0	0	0 : slabdata
<code>dma-kmalloc-2048</code>	0	0	2048	16	8 : tunables	0	0	0 : slabdata
<code>dma-kmalloc-1024</code>	0	0	1024	32	8 : tunables	0	0	0 : slabdata
<code>dma-kmalloc-512</code>	64	64	512	64	8 : tunables	0	0	0 : slabdata
<code>dma-kmalloc-256</code>	0	0	256	64	4 : tunables	0	0	0 : slabdata
<code>dma-kmalloc-128</code>	0	0	128	64	2 : tunables	0	0	0 : slabdata
<code>dma-kmalloc-64</code>	0	0	64	64	1 : tunables	0	0	0 : slabdata
<code>dma-kmalloc-32</code>	0	0	32	128	1 : tunables	0	0	0 : slabdata
<code>dma-kmalloc-16</code>	0	0	16	256	1 : tunables	0	0	0 : slabdata
<code>dma-kmalloc-8</code>	0	0	8	512	1 : tunables	0	0	0 : slabdata
<code>dma-kmalloc-192</code>	0	0	192	42	2 : tunables	0	0	0 : slabdata
<code>dma-kmalloc-96</code>	0	0	96	42	1 : tunables	0	0	0 : slabdata
<code>kmalloc-8192</code>	503	524	8192	4	8 : tunables	0	0	0 : slabdata
<code>kmalloc-4096</code>	1019	1032	4096	8	8 : tunables	0	0	0 : slabdata
<code>kmalloc-2048</code>	1383	1408	2048	16	8 : tunables	0	0	0 : slabdata
<code>kmalloc-1024</code>	3328	3488	1024	32	8 : tunables	0	0	0 : slabdata
<code>kmalloc-512</code>	17699	17728	512	64	8 : tunables	0	0	0 : slabdata
<code>kmalloc-256</code>	2589	2880	256	64	4 : tunables	0	0	0 : slabdata
<code>kmalloc-192</code>	3780	3780	192	42	2 : tunables	0	0	0 : slabdata
<code>kmalloc-128</code>	2240	2240	128	64	2 : tunables	0	0	0 : slabdata
<code>kmalloc-96</code>	5629	7644	96	42	1 : tunables	0	0	0 : slabdata
<code>kmalloc-64</code>	9641	10624	64	64	1 : tunables	0	0	0 : slabdata
<code>kmalloc-32</code>	9344	10240	32	128	1 : tunables	0	0	0 : slabdata
<code>kmalloc-16</code>	9728	9728	16	256	1 : tunables	0	0	0 : slabdata
<code>kmalloc-8</code>	10752	10752	8	512	1 : tunables	0	0	0 : slabdata

图 3-4 内存缓存的类型
Figure 3-4 `kmem_cache` types

由图 3-4 可知, `kmem_cache` 的内存长度一般是 2^n 大小, 只有 96 和 192 特殊, 内存长度最大值为 8192, 即 8 KB 大小。超过 8 KB 的内存分配请求将调用伙伴系统分配器完成。

3.2.2 创建新的分配标志

在使用 `kmalloc(size_t size, gfp_t flags)` 函数给通用的内存缓存 `kmem_cache` 分配内存空间后, 接着创建 `kmem_cache`。`kmem_cache` 通过以下函数创建, 相应的函数声明位于 `<mm/slab_common.c>` 中:

```
struct kmem_cache *kmem_cache_create( *name, size, align, flags, (void *))
```

其中 `size` 表示对象的长度; `align` 表示对象需要对齐的数值, 通常 0 即可满足要求, 即进行标准对齐; `flags` 表示分配标志位; `ctor` 表示对象的构造函数, 通常赋值为 `NULL`。如果创建成功, 返回内存缓存的地址, 否则返回 `NULL` 空指针。由于在 `slab_common.c` 中具体声明了该函数, 在 `slab.c` 和 `slub.c` 中可直接将该函数简化为 `kmem_cache_create(struct kmem_cache *s, unsigned long flags)` 使用。

创建 `kmem_cache` 后, 就可通过以下函数从指定的内存缓存中分配对象, 相应的函数声明位于 `<mm/slab.c>` 和 `<mm/slub.c>` 中:

在 `slab.c` 中: `void *kmem_cache_alloc(*cachep, flags)`

在 `slub.c` 中: `void *kmem_cache_alloc(*s, gfpflags)`

成员 `cachep` 或 `s` 均表示从指定的内存缓存分配; `flags` 或 `gfpflags` 均表示传给页分配器的分配标志位, 当内存缓存的所有 `slab` 中都没有可用于内存分配的空闲对象, `slab` 层向页分配器请求分配页时使用这个分配标志位, 该值将传递给本文第二章提到的 `__get_free_page(gfp_mask, order)` 函数。

观察上述函数参数列表可以发现, 在内存分配涉及的接口函数中, 都包含分配标志位参数, 而该参数由本文第二章分配标志位的介绍可知, 可用于标识内存分配请求来源和指定内存分配行为。在原有的 `SLUB` 分配器中, 分配标志位中的类型标志包含标识内核空间内存分配请求的 `GFP_KERNEL` 和标识用户空间内存分配请求的 `GFP_USER`。为了实现面向 Linux 内核空间的内存分配隔离, 可以在不增加接口函数参数的情况下, 对原有分配标志位进行扩展, 新增用于标识主内核内存分配请求的分配标志位和标识内核模块内存分配请求的分配标志位。

(1) 在 `<include/linux/gfp.h>` 增加代码如下:

<code>#define __GFP_COME_FROM_KERNEL</code>	<code>0x2000000u</code>
<code>#define __GFP_COME_FROM_MODULE</code>	<code>0x4000000u</code>

```

#define __GFP_COME_FROM_KERNEL ((__force gfp_t) __GFP_COME_FROM_KERNEL)
#define __GFP_COME_FROM_MODULE ((__force gfp_t) __GFP_COME_FROM_MODULE)
#define GFP_COME_FROM_KERNEL    (__GFP_RECLAIM | __GFP_IO | __GFP_FS |
__GFP_COME_FROM_KERNEL)
#define GFP_COME_FROM_MODULE    (__GFP_RECLAIM | __GFP_IO | __GFP_FS |
__GFP_COME_FROM_MODULE)

```

(2) 修改如下代码行:

① #define __GFP_NOLOCKDEP 0x2000000u

改为: #define __GFP_NOLOCKDEP 0x8000000u

② #define __GFP_BITS_SHIFT (25 + IS_ENABLED(CONFIG_LOCKDEP))

改为: #define __GFP_BITS_SHIFT (27 + IS_ENABLED(CONFIG_LOCKDEP))

此处分配标志偏移量 __GFP_BITS_SHIFT 由 25 改为 27 是因为原分配标志位定长地址截止到 0x1000000u, 转化为二进制为 25 位, 由于新增了两个分配标志位定长地址截止到 0x4000000u, 转化为二进制为 27 位, 所以移位的时候要增加 2 位才能正确指示当前分配标志位的位数。

最后将新增的两个分配标志位归并到 GFP_MOVABLE_MASK 中, 相关定义和函数修改如下:

```

#define GFP_MOVABLE_MASK (__GFP_RECLAIMABLE|__GFP_MOVABLE)
#define GFP_MOVABLE_SHIFT 3a
static inline int gfpflags_to_migratetype(const gfp_t gfp_flags)
{
    VM_WARN_ON((gfp_flags & GFP_MOVABLE_MASK) == GFP_MOVABLE_MASK);
    BUILD_BUG_ON((1UL << GFP_MOVABLE_SHIFT) != __GFP_MOVABLE);
    BUILD_BUG_ON((__GFP_MOVABLE >> GFP_MOVABLE_SHIFT)
                                                         != MIGRATE_MOVABLE);

    if (unlikely(page_group_by_mobility_disabled))
        return MIGRATE_UNMOVABLE;

    /* Group based on mobility */
    return (gfp_flags & GFP_MOVABLE_MASK) >> GFP_MOVABLE_SHIFT;
}
#undef GFP_MOVABLE_MASK
#undef GFP_MOVABLE_SHIFT

```

3.2.3 建立专用 kmem_cache 结构数组

(1) 研究思路

由前文对分配策略的描述和对创建新的分配标志的背景说明可知，要实现面向 Linux 内核空间的内存分配隔离方法，关键在于对 kmem_cache 结构的相关操作。为了分别响应主内核和内核模块的内存分配请求，需要在通用的内存缓存 kmem_cache 结构的基础上，建立新的专用 kmem_cache 结构数组。

由于通用 kmem_cache 结构共有两类，每类又根据分配到的内存长度细分为 13 种，原有的 SLUB 分配器便根据这两类的划分构建了相应的 kmem_cache 结构数组，分别是用于处理普通区域内存分配请求的 kmem_caches 数组和用于处理 DMA 区域内存分配请求的 kmem_dma_caches 数组。

本文为了建立专用 kmem_cache 结构数组，需要在原有 SLUB 分配器基础上，建立四个新的 kmem_cache 结构数组。其中，新建的 kmem_caches_kernel 数组用于处理来自主内核的针对普通区域的内存分配请求，新建的 kmem_dma_caches_kernel 数组用于处理来自主内核的针对 DMA 区域的内存分配请求，因为这两种数组处理的都是来自主内核的分配请求，所以对应的分配标志均是新建的 GFP_COME_FROM_KERNEL 标志。

新建的 kmem_caches_module 数组用于处理来自内核模块的针对普通区域的内存分配请求，新建的 kmem_dma_caches_module 数组用于处理来自内核模块的针对 DMA 区域的内存分配请求，因为这两种数组处理的都是来自内核模块的分配请求，所以对应的分配标志均是新建的 GFP_COME_FROM_MODULE 标志。

(2) 具体过程

首先，在 `<include /linux /slab.h>` 头文件中添加对四个新增数组的定义，新增代码如下：

```
#ifdef CONFIG_COME_FROM_KERNEL
extern struct kmem_cache *kmem_caches_kernel[KMALLOC_SHIFT_HIGH + 1];
extern struct kmem_cache *kmem_dma_caches_kernel[KMALLOC_SHIFT_HIGH + 1];
#endif
#ifdef CONFIG_COME_FROM_MODULE
extern struct kmem_cache *kmem_caches_module[KMALLOC_SHIFT_HIGH + 1];
extern struct kmem_cache *kmem_dma_caches_module[KMALLOC_SHIFT_HIGH + 1];
#endif
```

其次，在定义了新的 kmem_cache 结构数组后，需要对这四个新增的数组进行初始化操作，以方便 kmem_cache() 接口函数后续使用这些新增的数组进行相应的内存

分配。与内存缓存 `kmem_cache` 结构数组初始化相关的代码存放在 `<mm/slab_common.c>` 文件中。

具体操作过程为先新增对新 `kmem_cache` 结构数组调用的指针，以提供相应的调用接口。后重点对 `create_kmalloc_caches()` 函数进行修改，以完成对新数组的初始化操作。新增代码如下：

```
struct kmem_cache *kmalloc_caches_kernel[KMALLOC_SHIFT_HIGH + 1];
EXPORT_SYMBOL(kmalloc_caches_kernel);

#ifdef CONFIG_ZONE_DMA
struct kmem_cache *kmalloc_dma_caches_kernel[KMALLOC_SHIFT_HIGH + 1];
EXPORT_SYMBOL(kmalloc_dma_caches_kernel);
#endif

struct kmem_cache *kmalloc_caches_module[KMALLOC_SHIFT_HIGH + 1];
EXPORT_SYMBOL(kmalloc_caches_module);

#ifdef CONFIG_ZONE_DMA
struct kmem_cache *kmalloc_dma_caches_module[KMALLOC_SHIFT_HIGH + 1];
EXPORT_SYMBOL(kmalloc_dma_caches_module);
#endif
```

对 `create_kmalloc_caches()` 函数的修改，主要包含建立专用 `kmem_cache` 结构数组和建立这些数组对应的名称标识两部分。用于处理主内核普通区域分配请求的 `kmalloc_caches_kernel` 数组和用于处理内核模块普通区域分配请求的 `kmalloc_caches_module` 数组由于针对的都是普通区域，这两种数组可通过修改 `for (i = KMALLOC_SHIFT_LOW; i <= KMALLOC_SHIFT_HIGH; i++)` 这一循环结构创建，在 `for` 循环结构中增加的代码如下：

```
if (!kmalloc_caches_kernel[i])
    new_kmalloc_cache(i, flags);
if (!kmalloc_caches_module[i])
    new_kmalloc_cache(i, flags);
```

创建过程调用 `<mm /slab_common.c>` 中原有的 `new_kmalloc_cache()` 函数完成，该函数基于 `create_kmalloc_cache()` 函数实现（注意此处是 `create_kmalloc_cache()` 函数，而非 `create_kmalloc_caches()` 函数，后续在针对 DMA 区域的专用 `kmem_cache` 结构数组创建时也用到该函数）。

在完成 `kmalloc_caches_kernel` 数组和 `kmalloc_caches_module` 数组的创建后，需要为其建立对应的名称标识，新增的代码如下：

```
for (i = 0; i <= KMALLOC_SHIFT_HIGH; i++) {
    struct kmem_cache *s = kmalloc_caches_kernel[i];
    char *n;
    if (s) {
        n = kasprintf(GFP_NOWAIT, "kmalloc-kernel-%d", kmalloc_size(i));
        BUG_ON(!n);
        s->name = n;
    }
}

for (i = 0; i <= KMALLOC_SHIFT_HIGH; i++) {
    struct kmem_cache *s = kmalloc_caches_module[i];
    char *n;
    if (s) {
        n = kasprintf(GFP_NOWAIT, "kmalloc-module-%d", kmalloc_size(i));
        BUG_ON(!n);
        s->name = n;
    }
}
```

考虑到 `kmem_cache` 结构数组存在内存长度不是 2^n 的类型，即存在 96 和 192 两种特殊内存长度的类型，需要专门对其进行内存分配操作。新增的代码如下：

```
if (KMALLOC_MIN_SIZE <= 32 && !kmalloc_caches_kernel[1] && i == 6)
    new_kmalloc_cache(1, flags);
if (KMALLOC_MIN_SIZE <= 64 && !kmalloc_caches_kernel[2] && i == 7)
    new_kmalloc_cache(2, flags);
if (KMALLOC_MIN_SIZE <= 32 && !kmalloc_caches_module[1] && i == 6)
    new_kmalloc_cache(1, flags);
if (KMALLOC_MIN_SIZE <= 64 && !kmalloc_caches_module[2] && i == 7)
    new_kmalloc_cache(2, flags);
```

在完成针对普通区域新增数组的初始化工作后，针对 DMA 区域的内存分配请求，用于处理主内核 DMA 区域分配请求的 `kmalloc_dma_caches_kernel` 数组和用

于处理内核模块 DMA 区域分配请求的 `kmalloc_dma_caches_module` 数组可根据 (`SLAB_CACHE_DMA | flags`) 组合标志, 通过调用 `create_kmalloc_cache()` 函数完成, 并同样对其创建相应的名称标识, 新增的代码如下:

```
for (i = 0; i <= KMALLOC_SHIFT_HIGH; i++) {
    struct kmem_cache *s = kmalloc_caches_kernel[i];
    if (s) {
        int size = kmalloc_size(i);
        char *n = kasprintf(GFP_NOWAIT,
            "dma-kmalloc-kernel-%d", size);
        BUG_ON(!n);
        kmalloc_dma_caches_kernel[i] = create_kmalloc_cache(n,
            size, SLAB_CACHE_DMA | flags);
    }
}

for (i = 0; i <= KMALLOC_SHIFT_HIGH; i++) {
    struct kmem_cache *s = kmalloc_caches_module[i];
    if (s) {
        int size = kmalloc_size(i);
        char *n = kasprintf(GFP_NOWAIT,
            "dma-kmalloc-module-%d", size);
        BUG_ON(!n);
        kmalloc_dma_caches_module[i] = create_kmalloc_cache(n,
            size, SLAB_CACHE_DMA | flags);
    }
}
```

3.2.4 修改接口函数

在创建完新的分配标志, 建立了四个专用 `kmem_cache` 结构数组并对数组进行初始化后, 为了实现面向 Linux 空间的内存分配隔离方法, 完成 Linux 内核空间主内核和内核模块在内存分配时的隔离, 还需要对接口函数进行修改。通过修改接口函数, 改造后的 SLUB 分配器可通过对分配标志位的判定, 对特定的分配标志分配对应的 `kmem_cache` 结构数组, 进而完成后续的内存分配操作。

经过对内核源码的研究发现，在众多的接口函数中，<mm/slab_common.c>中的 `kmalloc_slab()` 函数提供了最原始的基于分配标志的分配行为。

具体而言，`kmalloc_slab()` 函数的处理逻辑是如果内存分配标志中含有 `GFP_DMA`，则启用针对 `DMA` 区域的内存分配机制，如果内存分配标志中不含 `GFP_DMA`，则启用针对普通区域的内存分配机制。

两种分配机制类似，都先判断分配标志是否同时包含新定义的标识主内核内存分配请求的 `GFP_COME_FROM_KERNEL` 标志和标识内核模块内存分配请求的 `GFP_COME_FROM_MODULE` 标志。如果同时包含或都不包含，就返回原有的通用 `kmem_cache` 结构数组，即根据区域的不同分别返回对应的 `kmalloc_dma_caches` 或 `kmalloc_caches` 数组。如果只包含 `GFP_COME_FROM_KERNEL` 标志，则返回新创建的主内核专用 `kmem_cache` 结构数组，即根据区域的不同分别返回对应的 `kmalloc_dma_caches_kernel` 数组或 `kmalloc_caches_kernel` 数组。如果只包含 `GFP_COME_FROM_MODULE` 标志，则返回新创建的内核模块专用 `kmem_cache` 结构数组，即根据区域的不同分别返回对应的 `kmalloc_dma_caches_module` 数组和 `kmalloc_caches_module` 数组。新增的代码如下：

```
if (unlikely((flags & GFP_DMA))) {
    if (unlikely((flags & GFP_COME_FROM_KERNEL & GFP_COME_FROM_MODULE)))
        return kmalloc_dma_caches[index];
    if (likely((flags & GFP_COME_FROM_KERNEL)))
        return kmalloc_dma_caches_kernel[index];
    if (likely((flags & GFP_COME_FROM_MODULE)))
        return kmalloc_dma_caches_module[index];
    return kmalloc_dma_caches[index];
} else {
    if (unlikely((flags & GFP_COME_FROM_KERNEL & GFP_COME_FROM_MODULE)))
        return kmalloc_caches[index];
    if (likely((flags & GFP_COME_FROM_KERNEL)))
        return kmalloc_caches_kernel[index];
    if (likely((flags & GFP_COME_FROM_MODULE)))
        return kmalloc_caches_module[index];
    return kmalloc_caches[index];
}
```

代码中 `unlikely()` 和 `likely()` 用于描述判断条件为真的可能性大小，是种常用的内核代码编程手段，可在一定程度上提升代码执行速度。

3.3 基于伙伴系统分配器的内核空间内存分配隔离方法

在完成对块分配器中 SLUB 分配器的改进后，本文解决了内核在以字节为单位进行内存分配时的混合页问题。为了完整实现面向 Linux 内核空间的内存分配隔离方法，还需要对内核以页框为单位的内存分配进行改进，这种分配方式由伙伴系统分配器完成。

由于要具体实现面向 Linux 内核空间的内存分配隔离方法，不仅需要为主内核和内核模块构建多套不同的页表，还需要对页表修改、创建和切换等过程进行完整的监控和保护，相关的技术实现复杂且工作量庞大，且目前国内外研究人员中先进团体已经对此提出了完善而成熟的方案，本文在结合相关研究人员工作的基础上，假定已为主内核和内核模块建立了独立的页表，重点关注如何在对 SLUB 分配器改进的基础上，继续对伙伴系统分配器进行改进，从而构建完整的解决混合页问题，可实现面向 Linux 内核空间的内存分配隔离的新分配器处理模型。

由第二章对内存分配方式的说明可知，在内核两种分配方式中都用到了分配标志，且由第三章经分析内核源码提出的 SLUB 分配器整体设计框架可知，SLUB 分配器通过伙伴系统分配器完成初始化操作，因而分配标志在两种分配器中是通用的。因为已经假定为主内核和内核模块构建了多套不同的页表，此时要完成主内核和内核模块内存分配时的隔离，只需使伙伴系统分配器根据分配标志限制内存分配请求可读取的页表权限即可，从而使特定的页表只能由对应的分配请求访问，而没有读取权限的内存访问操作将触发缺页异常。

由于建立多套页表和建立两套页表处理逻辑上类似，区别只是多套页表增加了更多判断流程，为了降低流程图绘制的难度，增加流程图的可读性，此处仅使用两套页表进行说明，这两套页表分别用于主内核和内核模块的内存分配操作。

由本文第二章对内核使用伙伴系统分配器以页框为单位进行内存分配时的相关说明可知，在伙伴系统分配器中提供了多个接口函数，但图 3-5 中仅涉及 `__alloc_pages_nodemask()` 一个接口函数，这是因为多个接口函数最终都会进入到 `__alloc_pages_nodemask()` 这一核心函数中处理，所以仅需对其进行修改即可完成对伙伴系统分配器的改进。

改进后的伙伴系统分配器的处理流程如图 3-5 所示。改进后的伙伴系统分配器的处理思想为先判断分配操作是否成功，如果分配失败，与未分配类似，此时不涉及主内核和内核模块在内存分配时的隔离，不对其进行后续操作。

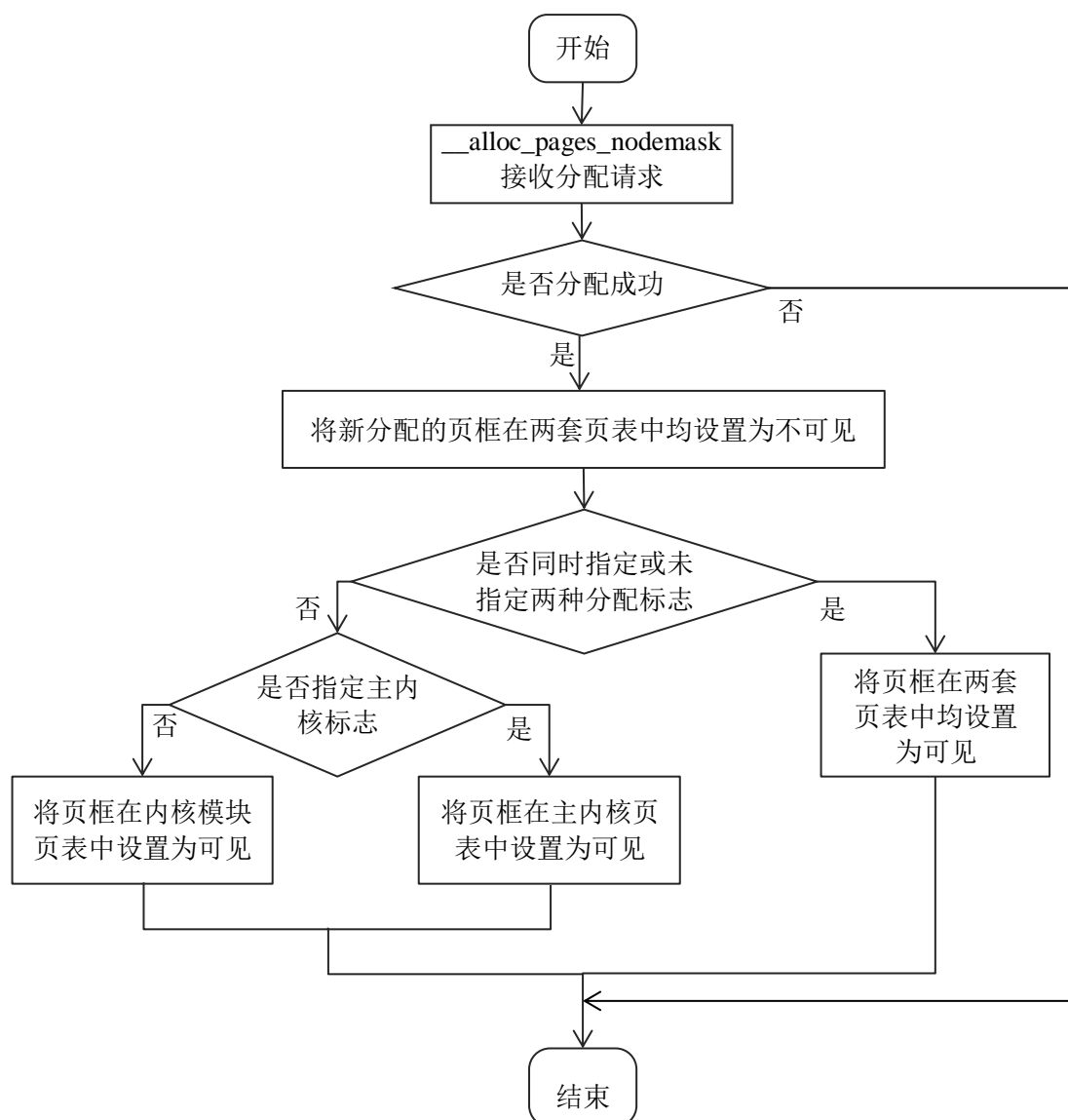


图 3-5 伙伴系统分配器改进的方法框架

Figure 3-5 Improved method framework of buddy system allocator

如果分配成功，将新分配到的页框在两套页表中均设置为不可见，继续进行后续的判断。如果内存分配请求中同时指定了来源于主内核和内核模块的内存分配标志，说明此时应同时满足来源于主内核和内核模块的内存分配需求，此时和同时未指定分配标志的情况类似，不再对分配来源进行区分，相应的页框在两套页表中均应设置为可见。如果内存分配请求中只指定了一种分配标志，需要对其进行进一步判断。如果指定了主内核分配标志，说明此时新分配的页框应专门用于主内核的内存分配请求，将页框在主内核页表中设置为可见，由于在内核模块页表中仍保持不可见，伙伴系统分配器通过限定页框的可见性可使不可见的页框在后续内存分配操作中触发缺页异常，从而使特定的页框仅能由指定的主内核或内核模块访问并使用，进行实现主内核和内核模块在内存分配时的隔离。

3.4 完整的面向 Linux 内核空间的内存分配隔离方法

在完成对 SLUB 分配器的改进和对伙伴系统分配器的改进后，本文在此基础上，建立了完整的面向 Linux 内核空间的内存分配隔离方法的新分配器理论模型。新模型的总体架构如图 3-6 所示。

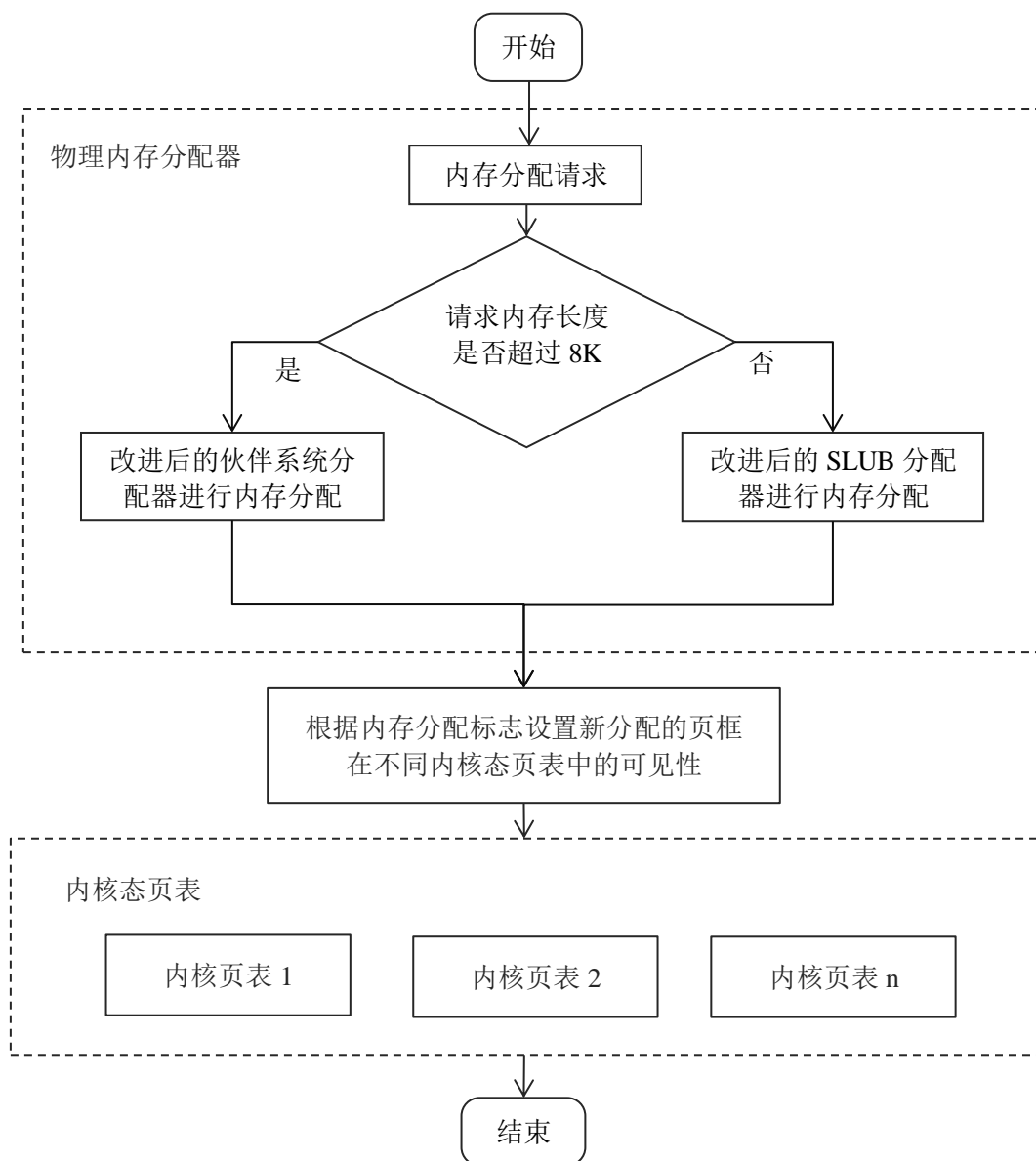


图 3-6 完整的内存分配隔离方法理论模型

Figure 3-6 Complete theoretical model of memory allocation isolation method

对于物理内存分配器而言，当内核接收到内存分配请求时，对当前请求分配的内存长度进行判断。如果请求分配的内存长度大于 8 KB，则调用改进后的伙伴系统分配器进行内存分配，否则调用改进后的 SLUB 分配器进行内存分配。本文 3.2.1 中已对为什么以 8 KB 作为判断条件做出必要说明，在此不再赘述。在完成分

配操作后，需根据内存分配标志设置新分配的页框在不同内核态页表中的可见性，使特定的页框仅能由指定分配标志的分配请求访问，从而实现主内核和内核模块在内存分配时的隔离。

3.5 本章小结

本章首先通过对内核源码的分析，给出了 SLAB 分配器和 SLUB 分配器的数据结构图，并在此过程中，经过对内核源码中各函数相互调用关系的研究，明确了实验采用的 4.13.2 内核使用的默认块分配器是 SLUB 分配器。

因为本文研究工作旨在解决混合页问题，所以研究工作的重心在于对 SLUB 分配器的改进。因而本文接着给出了基于 SLUB 分配器的内核空间内存分配方法的整体框架，并对框架进行了必要的说明。

之后本文对 SLUB 分配器的改进进行了详细的说明，结合对内核源码的修改，给出了从创建分配标志，到建立专用 `kmem_cache` 结构数组，再到修改接口函数的完整改进流程。

然后本章论述了基于伙伴系统分配器的内核空间内存分配隔离方法，并给出了伙伴系统分配器改进的方法框架。

本章最后结合对 SLUB 分配器和伙伴系统分配器的改进，提出了完整的面向 Linux 内核空间的内存分配隔离方法，并创建了相应的新分配器概念模型。

下一章将在本章基础上，对本章提出的方法和模型进行验证和评价。

4 实验结果分析

本章介绍实验环境和实验方法，得出实验结果，对比其他研究人员的研究成果，对本文提出的面向 Linux 内核空间的内存分配隔离方法进行分析与评价。

4.1 实验环境

本文的研究工作在 VMware Workstations11 虚拟机平台进行。创建的虚拟机基于 Ubuntu 18.04 (64 位)，对应的内核版本号为 4.15.0，分配的内存大小为 2 GB，硬盘空间为 60 GB，处理器为双核处理器。实验编译和修改的内核版本号为 4.13.2，代码编译采用内核中默认的 GCC 编译器。

4.2 实验方法

4.2.1 sysfs 文件系统

在本文第三章对通用的内存缓存 `kmem_cache` 类型访问时，使用了 `<cat /proc /slabinfo>` 命令，该命令的实质是用户通过查看存储在 `proc` 这一虚拟文件系统中的特殊文件，可以获知当前处于运行状态的内核信息，具体包括系统硬件设备信息，内核组成信息和当前运行进程的相关信息等。

在 Linux 内核 2.6 及以上版本中，还存在一种更为强大的 `sysfs` 文件系统。该文件系统也是一种虚拟文件系统，它不仅可以把驱动程序信息和系统硬件设备信息等内核信息以文件的形式提供给用户查看或用户程序调用，还可以用来设置驱动程序和管理系统硬件设备。相比于 `proc` 文件系统，`sysfs` 文件系统有很多优点，最大优点在于其设计结构的清晰，是一种更为理想化的访问和管理内核及内核模块的途径。该文件系统挂载于 `/sys` 挂载点，构建了面向对象层次结构的视图，使得它在帮助用户以文件系统方式观察系统设备拓扑结构的同时，也提供了一种显式描述内核对象和对象属性，说明对象关系的机制。具体的机制为利用用户空间中的目录描述内核对象，利用用户空间中的文件描述对象属性，利用用户空间中的链接描述对象关系。在此基础上，`sysfs` 文件系统提供分别针对内核和针对用户的接口。

为了更好地对本文提出的面向 Linux 内核空间的内存分配隔离方法进行评估，

显示当前修改后的内核在内存分配时的具体运行情况，本文将基于 sysfs 文件系统进行相关说明。

4.2.2 实验过程

(1) 实验一：验证专用 kmem_cache 结构数组是否建立

首先，为了获知修改后的内核当前内存缓存 kmem_cache 的生成情况，本文借助 sysfs 文件系统实时观测。而要实现基于 sysfs 文件系统的追踪显示功能，需要对 sysfs 文件系统相关的 create_unique_id() 函数进行修改，该函数存放在 <mm/slab.c> 文件中，新增的代码如图 4-1 所示。

```
//START_AddedbyDuanxinfeng
if(!memcmp(s->name,"dma-kmalloc-kernel",18) || !memcmp(s->name,"kmalloc-kernel",14))
    *p++ = 'k';
if(!memcmp(s->name,"dma-kmalloc-module",18) || !memcmp(s->name,"kmalloc-module",14))
    *p++ = 'm';
//END_AddedbyDuanxinfeng
```

图 4-1 create_unique_id() 函数新增代码
Figure 4-1 New code for create_unique_id() function

其处理逻辑是利用 memcmp() 比较函数对第三章新增的专用 kmem_cache 结构数组的名称标识进行判断，如果名称标识中含有“dma-kmalloc-kernel”或含有“kmalloc-kernel”，就新增字符“k”标志；如果名称标识中含有“dma-kmalloc-module”或含有“kmalloc-module”，就新增字符“m”标志。新增的两个特殊标志将在 sysfs 文件系统展示内核信息时出现，其实际显示效果将在之后的测试结果中呈现。

在完成了对 sysfs 文件系统的改进后，可以进入编译成功并启用的新内核 </sys/kernel/slab> 中查询有关内存缓存 kmem_cache 的动态运行信息，基于 sysfs 文件系统的测试结果可通过使用“ll | grep kernel”和“ll | grep module”命令分类显示。如图 4-2 显示的是当前针对主内核的专用 kmem_cache 结构数组情况，如图 4-3 显示的是当前针对内核模块的 kmem_cache 结构数组情况。

由图 4-2 和图 4-3 可知，当前 SLUB 分配器已经根据分配来源的不同，分别建立了基于主内核的专用 kmem_cache 结构数组和基于内核模块的专用 kmem_cache 结构数组。其中，专用 kmem_cache 结构数组用“dk”标志（“d”标志原 create_unique_id() 函数本身含有）表示主内核针对 DMA 区域的内存分配，用“k”标志表示主内核针对普通区域的内存分配，用“dm”标志表示内核模块针对 DMA 区域的内存分配，用“m”标志表示内核模块针对普通区域的内存分配。

```

lrwxrwxrwx 1 root root 0 May 2 21:15 dma-kmalloc-kernel-1024 -> :dk-0001024/
lrwxrwxrwx 1 root root 0 May 2 21:15 dma-kmalloc-kernel-128 -> :dk-0000128/
lrwxrwxrwx 1 root root 0 May 2 21:15 dma-kmalloc-kernel-16 -> :dk-0000016/
lrwxrwxrwx 1 root root 0 May 2 21:15 dma-kmalloc-kernel-192 -> :dk-0000192/
lrwxrwxrwx 1 root root 0 May 2 21:15 dma-kmalloc-kernel-2048 -> :dk-0002048/
lrwxrwxrwx 1 root root 0 May 2 21:15 dma-kmalloc-kernel-256 -> :dk-0000256/
lrwxrwxrwx 1 root root 0 May 2 21:15 dma-kmalloc-kernel-32 -> :dk-0000032/
lrwxrwxrwx 1 root root 0 May 2 21:15 dma-kmalloc-kernel-4096 -> :dk-0004096/
lrwxrwxrwx 1 root root 0 May 2 21:15 dma-kmalloc-kernel-512 -> :dk-0000512/
lrwxrwxrwx 1 root root 0 May 2 21:15 dma-kmalloc-kernel-64 -> :dk-0000064/
lrwxrwxrwx 1 root root 0 May 2 21:15 dma-kmalloc-kernel-8 -> :dk-0000008/
lrwxrwxrwx 1 root root 0 May 2 21:15 dma-kmalloc-kernel-8192 -> :dk-0008192/
lrwxrwxrwx 1 root root 0 May 2 21:15 dma-kmalloc-kernel-96 -> :dk-0000096/
lrwxrwxrwx 1 root root 0 May 2 21:15 kmalloc-kernel-1024 -> :k-0001024/
lrwxrwxrwx 1 root root 0 May 2 21:15 kmalloc-kernel-128 -> :k-0000128/
lrwxrwxrwx 1 root root 0 May 2 21:15 kmalloc-kernel-16 -> :k-0000016/
lrwxrwxrwx 1 root root 0 May 2 21:15 kmalloc-kernel-192 -> :k-0000192/
lrwxrwxrwx 1 root root 0 May 2 21:15 kmalloc-kernel-2048 -> :k-0002048/
lrwxrwxrwx 1 root root 0 May 2 21:15 kmalloc-kernel-256 -> :k-0000256/
lrwxrwxrwx 1 root root 0 May 2 21:15 kmalloc-kernel-32 -> :k-0000032/
lrwxrwxrwx 1 root root 0 May 2 21:15 kmalloc-kernel-4096 -> :k-0004096/
lrwxrwxrwx 1 root root 0 May 2 21:15 kmalloc-kernel-512 -> :k-0000512/
lrwxrwxrwx 1 root root 0 May 2 21:15 kmalloc-kernel-64 -> :k-0000064/
lrwxrwxrwx 1 root root 0 May 2 21:15 kmalloc-kernel-8 -> :k-0000008/
lrwxrwxrwx 1 root root 0 May 2 21:15 kmalloc-kernel-8192 -> :k-0008192/
lrwxrwxrwx 1 root root 0 May 2 21:15 kmalloc-kernel-96 -> :k-0000096/

```

图 4-2 主内核的专用 kmem_cache 结构数组

Figure 4-2 The dedicated kmem_cache structure array of the core kernel

```

lrwxrwxrwx 1 root root 0 May 2 21:26 dma-kmalloc-module-1024 -> :dm-0001024/
lrwxrwxrwx 1 root root 0 May 2 21:26 dma-kmalloc-module-128 -> :dm-0000128/
lrwxrwxrwx 1 root root 0 May 2 21:26 dma-kmalloc-module-16 -> :dm-0000016/
lrwxrwxrwx 1 root root 0 May 2 21:26 dma-kmalloc-module-192 -> :dm-0000192/
lrwxrwxrwx 1 root root 0 May 2 21:26 dma-kmalloc-module-2048 -> :dm-0002048/
lrwxrwxrwx 1 root root 0 May 2 21:26 dma-kmalloc-module-256 -> :dm-0000256/
lrwxrwxrwx 1 root root 0 May 2 21:26 dma-kmalloc-module-32 -> :dm-0000032/
lrwxrwxrwx 1 root root 0 May 2 21:26 dma-kmalloc-module-4096 -> :dm-0004096/
lrwxrwxrwx 1 root root 0 May 2 21:26 dma-kmalloc-module-512 -> :dm-0000512/
lrwxrwxrwx 1 root root 0 May 2 21:26 dma-kmalloc-module-64 -> :dm-0000064/
lrwxrwxrwx 1 root root 0 May 2 21:26 dma-kmalloc-module-8 -> :dm-0000008/
lrwxrwxrwx 1 root root 0 May 2 21:26 dma-kmalloc-module-8192 -> :dm-0008192/
lrwxrwxrwx 1 root root 0 May 2 21:26 dma-kmalloc-module-96 -> :dm-0000096/
lrwxrwxrwx 1 root root 0 May 2 21:26 kmalloc-module-1024 -> :m-0001024/
lrwxrwxrwx 1 root root 0 May 2 21:26 kmalloc-module-128 -> :m-0000128/
lrwxrwxrwx 1 root root 0 May 2 21:26 kmalloc-module-16 -> :m-0000016/
lrwxrwxrwx 1 root root 0 May 2 21:26 kmalloc-module-192 -> :m-0000192/
lrwxrwxrwx 1 root root 0 May 2 21:26 kmalloc-module-2048 -> :m-0002048/
lrwxrwxrwx 1 root root 0 May 2 21:26 kmalloc-module-256 -> :m-0000256/
lrwxrwxrwx 1 root root 0 May 2 21:26 kmalloc-module-32 -> :m-0000032/
lrwxrwxrwx 1 root root 0 May 2 21:26 kmalloc-module-4096 -> :m-0004096/
lrwxrwxrwx 1 root root 0 May 2 21:26 kmalloc-module-512 -> :m-0000512/
lrwxrwxrwx 1 root root 0 May 2 21:26 kmalloc-module-64 -> :m-0000064/
lrwxrwxrwx 1 root root 0 May 2 21:26 kmalloc-module-8 -> :m-0000008/
lrwxrwxrwx 1 root root 0 May 2 21:26 kmalloc-module-8192 -> :m-0008192/
lrwxrwxrwx 1 root root 0 May 2 21:26 kmalloc-module-96 -> :m-0000096/

```

图 4-3 内核模块的专用 kmem_cache 结构数组

Figure 4-3 The dedicated kmem_cache structure array of the kernel modules

在完成对专用 kmem_cache 结构数组的验证后,接下来进行对分配标志的验证。

(2) 实验二：验证是否可以根据分配标志进行对应的内存分配

在第三章面向 Linux 内核空间的内存分配隔离方法实现的新分配器模型中, 已经完成了对相关接口函数的修改。为了验证改造后的新分配器模型是否可以根据新增的分配标志进行不同来源的内存分配, 本文通过构建内核测试模块的方式完成相应的测试工作。

具体实验过程为先构建测试模块, 测试模块通过调用 `kmalloc()` 函数, 根据提供的分配标志和申请的内存长度进行对应的分配操作。为了得出验证结论, 调用测试模块前, 要先记录 `/sys/kernel/ slab` 文件夹下待分配内存长度对应的内存缓存 `kmem_cache` 相关属性值, 加载测试模块后, 再次记录测试模块指定内存长度对应的 `kmem_cache` 相关属性值, 通过属性值的变化得出最终的验证结论。

根据分配标志的不同, 实验过程分四步进行, 由于内存长度的不同并不影响实验得出结论的正确性, 所以随机选 16 字节长度为例进行说明。

1) 只指定 `GFP_COME_FROM_KERNEL` 分配标志

首先, 构建请求分配内存长度为 16 字节, 使用 `GFP_COME_FROM_KERNEL` 分配标志的测试模块 1, 由于 `GFP_COME_FROM_KERNEL` 标识了主内核内存分配请求, 且测试模块 1 未请求针对 DMA 区域进行分配, 为普通区域内存分配, 所以该测试模块对应的专用 `kmem_cache` 结构数组为 `kmalloc-kernel-16`, 测试模块代码如图 4-4 所示。

测试模块 1 代码中的 `MODULE_LICENSE("GPL")` 表示模块的许可证声明。

```
#include <linux/slab.h>
#include <linux/module.h>

static int __init test_init(void)
{
    buffer=kmalloc(16,GFP_COME_FROM_KERNEL);
    return 0;
}

static void __exit test_exit(void)
{
    kfree(buffer);
}

module_init(test_init);
module_exit(test_exit);
MODULE_LICENSE("GPL")
```

图 4-4 只指定主内核分配标志的测试模块 1

Figure 4-4 Only specify test module 1 with main kernel allocation flags

其次, 进入 `kmalloc-kernel-16` 目录下记录当前重要属性值 `slabs`、`order` 和 `objects`。三个属性值表示的意义如表 4-1 所示。

表 4-1 属性值说明
Table 4-1 Property value description

属性值	相关说明
slabs	表示内存缓存 kmem_cache 中 slab 个数
order	表示物理页框的阶数 即每个 slab 对应的物理页框数为 2^{order}
objects	表示该内存缓存 kmem_cache 中包含的所有对象的个数

测试模块 1 加载前记录的属性值如图 4-5 所示。

```
dx&XF4-13:/sys/kernel/slab/kmalloc-kernel-16$ sudo cat slabs order objects
0
0
0
```

图 4-5 测试模块 1 加载前的属性值
Figure 4-5 Attribute value before test module 1 loaded

加载测试模块 1 后，记录的相关属性值如图 4-6 所示。

```
dx&XF4-13:/sys/kernel/slab/kmalloc-kernel-16$ sudo cat slabs order objects
1 N0=1
0
1024 N0=1024
```

图 4-6 测试模块 1 加载后的属性值
Figure 4-6 Attribute value after test module 1 loaded

最后，由前后 slabs、order 和 objects 属性值的变化可知，内存缓存 kmem_cache 中 slab 个数由 0 变为 1 个；每个 slab 对应的物理页框数前后一致，均为 $2^0=1$ ；内存缓存 kmem_cache 中包含的所有对象的由 0 变为 1024 个。

该实验结果表明，对于指定了 GFP_COME_FROM_KERNEL 分配标志的请求，面向 Linux 内核空间的内存分配隔离方法实现的新分配器模型，可以成功调用针对主内核分配请求创建的专用 kmem_cache 结构数组顺利完成相应的内存分配操作。

2) 只指定 GFP_COME_FROM_MODULE 分配标志

首先，构建请求分配内存长度为 16 字节，使用 GFP_COME_FROM_MODULE 分配标志的测试模块 2，由于 GFP_COME_FROM_MODULE 标识了内核模块内存分配请求，且测试模块 2 未请求针对 DMA 区域进行分配，为普通区域内存分配，所以该测试模块对应的专用 kmem_cache 结构数组为 kmalloc-module-16，测试模块代码如图 4-7 所示。


```

#include <linux/slab.h>
#include <linux/module.h>

static int __init test_init(void)
{
    buffer=kmalloc(16,GFP_COME_FROM_MODULE);
    return 0;
}

static void __exit test_exit(void)
{
    kfree(buffer);
}

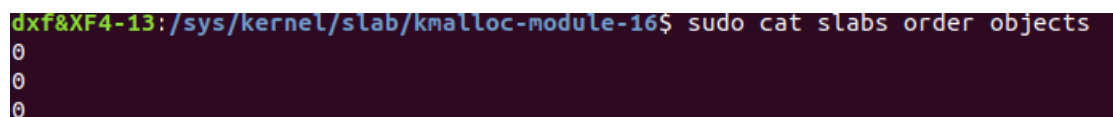
module_init(test_init);
module_exit(test_exit);
MODULE_LICENSE("GPL")

```

图 4-7 只指定内核模块分配标志的测试模块 2

Figure 4-7 Only specify test module 2 with kernel module allocation flags

其次，进入 `kmalloc-module-16` 目录下记录当前重要属性值 `slabs`、`order` 和 `objects`。测试模块加载前记录的属性值如图 4-8 所示。



```

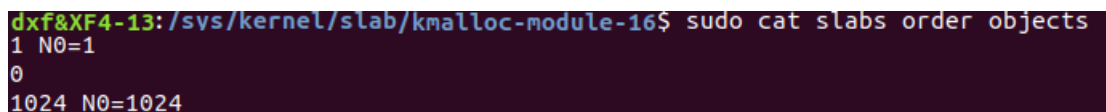
dx&XF4-13:/sys/kernel/slab/kmalloc-module-16$ sudo cat slabs order objects
0
0
0

```

图 4-8 测试模块 2 加载前的属性值

Figure 4-8 Attribute value before test module 2 loaded

加载测试模块后，记录的相关属性值如图 4-9 所示。



```

dx&XF4-13:/sys/kernel/slab/kmalloc-module-16$ sudo cat slabs order objects
1 NO=1
0
1024 NO=1024

```

图 4-9 测试模块 1 加载后的属性值

Figure 4-9 Attribute value after test module 1 loaded

由图 4-8 和图 4-9 知，`kmalloc-module-16` 目录下属性值的变化与 `kmalloc-kernel-16` 目录下属性值的变化相同。

该实验结果表明，对于指定了 `GFP_COME_FROM_MODULE` 分配标志的请求，新分配器模型可以成功调用针对内核模块分配请求创建的专用 `kmem_cache` 结构数组顺利完成相应的内存分配操作。

3) 同时指定两种分配标志

构建请求分配内存长度为 16 字节，同时使用 `GFP_COME_FROM_KERNEL` 和 `GFP_COME_FROM_MODULE` 分配标志的测试模块 3，由于测试模块 3 未请求针对 DMA 区域进行分配，为普通区域内存分配，相关的测试模块代码如图 4-10 所示。

```

#include <linux/slab.h>
#include <linux/module.h>

static int __init test_init(void)
{
    buffer=kmalloc(16,GFP_COME_FROM_KERNEL | GFP_COME_FROM_MODULE);
    return 0;
}

static void __exit test_exit(void)
{
    kfree(buffer);
}

module_init(test_init);
module_exit(test_exit);
MODULE_LICENSE("GPL")

```

图 4-10 同时指定两种分配标志的测试模块 3

Figure 4-10 At the same time specify the test mode 3 of the two allocation signs

测试模块 3 加载前后，分别进入 kmalloc-kernel-16 目录、kmalloc-module-16 目录和 kmalloc-16 目录查看 slabs、order 和 objects 前后属性值的变化情况，发现 kmalloc-kernel-16 目录和 kmalloc-module-16 目录下各个属性值均未变化，而 kmalloc-8 中 slabs 和 objects 值发生变化。

该实验结果表明，对于同时指定了 GFP_COME_FROM_KERNEL 和 GFP_COME_FROM_MODULE 分配标志的请求，新分配器模型调用原有的通用 kmem_cache 结构数组完成相应的内存分配操作。

4) 不指定分配标志

不指定分配标志的实验过程与第三步类似，区别只是调用测试模块时，未指定分配标志，得到的实验结果相同，这也意味着新构建的分配器模型并未破坏系统原有的分配机制。

(3) 实验三：内存分配性能测试

内核内存分配时的性能采用内存带宽测试工具 mbw 进行测试。该工具可测试内核在内存(MEMCPY)、字符串(DUMB)、内存块(MCBLOCK)三种方式(Method)下内存拷贝的速率，并由三种方式各自的平均值(AVG)得出相应的内存分配速率^[33]。内存分配速率越大，表示内存分配性能越好。

分别启用修改前的 4.13.2 内核和修改后的内核，利用 mbw 对其进行测试。执行命令：./mbw -q -n 10 256，其中 -q 表示隐藏日志，-n 10 表示运行 10 次，256(MB) 表示测试所用内存大小，测试结果如表 4-2 所示。

表 4-2 性能测试表
Table 4-2 Performance test table

方式	修改前	修改后
MEMCPY (AVG)	耗时：0.03997	耗时：0.04069
	数据大小：256.00000	数据大小：256.00000
	速率：6404.852 MiB/s	速率：6291.828 MiB/s
DUMB (AVG)	耗时：0.12242	耗时：0.13344
	数据大小：256.00000	数据大小：256.00000
	速率：2091.165 MiB/s	速率：1918.408 MiB/s
MCBLOCK (AVG)	耗时：0.04300	耗时：0.04462
	数据大小：256.00000	数据大小：256.00000
	速率：5952.935 MiB/s	速率：5737.723 MiB/s

由表 4-2 可知，本文提出的方法由于并未引入额外的内存分配机制，而是基于内核现有的内存分配机制重点对块分配器进行了改进，对内核原有的内存分配性能影响很小。

4.3 讨论

在目前基于 Linux 内核页表完成内存分配隔离的方法中，主要存在三类突出的问题。

第一类是没有意识到存在块分配器导致的混合页问题，只实现了基于页粒度的内存分配隔离，并未真正有效的解决混合页引发的缓冲区溢出和内存泄露问题。第二类是内存分配隔离的实现，依赖于频繁的合法性检查和中断保护机制，以及对数据流的实时跟踪检测，相关技术复杂，性能开销大。第三类是构建了独立的内存管理和分配模型^[34]，没有有效利用 Linux 内核原有的内存分配机制，尤其完全抛弃 Linux 原有的块分配器，自己独立实现，而最终的结果往往有很大的局限性，相比功能强大的块分配器，新实现的模型过于简单，分配效率低^[35]。

本文提出的面向 Linux 内核空间的内存分配隔离方法，从混合页问题出发，在 Linux 内核原有伙伴分配器和块分配器的基础上，通过改进目前默认的 SLUB 块分配器，让其可以根据新增的标识内存分配请求来源的分配标志，调用新增的主内核和内核模块专用内存缓存完成对应的内存分配操作，从而实现主内核和内核模块在内存分配时的隔离。

对比其他人员研究成果，本文提出的方法不仅考虑到了混合页，针对以字节

为单位的小块内存分配带来的系统安全威胁问题提出了有效的解决方案，还基于 Linux 原有的内存分配机制实现，在很好地保留了原有分配机制优点的基础上，对缺点进行了改进，技术实现相对简单，性能开销小，便于推广，具有一定的参考意义和实用价值。

4.4 本章小结

本章首先对实验环境进行了说明。之后重点介绍了实验方法。在对实验方法的讲解中，先对测试时用到的 `sysfs` 文件系统进行了必要的说明，后具体介绍了实验过程。实验整体上分三类进行，第一类是验证主内核和内核模块对应的专用内存缓存 `kmem_cache` 结构数组是否建立；第二类是验证新生成的分配器处理模型是否可以根据分配标志进行对应的内存分配操作，对第二类实验验证过程的讲解，本章根据不同的情形都给出了从测试模块构建到得出结论的完整说明；第三类是对新模型内存分配性能进行测试。

本章最后结合其他研究人员研究过程中存在的问题，对本文提出的方法进行了分析和评价。

5 总结与展望

5.1 工作总结

内核是 Linux 操作系统安全运行和管理控制的核心,面对目前频繁的系统安全威胁,保护内核安全具有重要的意义。为了提升内核运行时的安全性和可信性,许多研究课题针对内核展开,且取得了一定的研究成果。内核隔离作为其中一个重要而热门的研究方向,研究人员基于软件和硬件的支持提出了许多种解决方案。由于硬件资源本身的封闭性和硬件支持的特定性,以软件隔离为主,以内存隔离为核心,采用进程、页表和虚拟机等隔离手段进行内核隔离的思想得到了更广泛的应用。本文在此背景下,结合国内外研究现状,提出了一种面向 Linux 内核空间的内存分配隔离的方法。研究过程和方法实现中得出的结论如下。

(1) 研究工作开展前经调研得出的内存分配隔离背景相关的结论。内存损坏是内核安全面临的最主要的威胁,其主要表现为缓冲区溢出和内存泄露。内存损坏相关的防御技术,往往离不开内存隔离的支持。内存隔离作为一种强大有效的内核安全隔离防御技术,重要的是做到内存分配时隔离。

(2) 研究工作开展时结合国内外研究现状及 Linux 内核原有内存分配机制得出的结论。Linux 内核内存分配分为以页为单位的内存分配和以字节为单位的内存分配,以页为单位的内存分配采用伙伴系统分配器,以字节为单位的内存分配采用块分配器。目前内核中默认的块分配器是 SLUB 分配器。块分配器在分配小块内存时,会将属于主内核或不同内核模块的数据分配到同一物理页框上,这种混合页的存在,难以进行页级别的保护,这也是导致缓冲区溢出和内存泄露最主要的原因。但可惜的是,现有的研究有些根本没注意到该问题的存在,只实行了页粒度的保护;有些注意到了该问题的存在,但解决办法要么技术复杂、性能开销大,要么完全自己构建新的块分配器,实现简单,分配效率低。

(3) 面向 Linux 内核空间的内存分配隔离方法的探究中得出的结论。SLUB 块分配器为每种对象类型创建一个内存缓存,每个内存缓存由多个 slab 组成。每个内存缓存对应一个 kmem_cache 结构,通过为主内核和内核模块新建专用的 kmem_cache 结构数组,可以在内存分配时为二者分配专用的内存缓存,进而解决混合页的问题。在此基础上,本文研究并继续改进了伙伴系统分配器,在完成对 SLUB 分配器和伙伴系统分配器的改进后,本文提出了完整的面向 Linux 内核空间的内存分配隔离方法,并构建了相应的新分配器处理的理论模型,从而实现主内

核和内核模块在内存分配时的隔离。

(4) 验证面向 Linux 内核空间的内存分配隔离方法实现的新分配器处理模型得出的结论。新分配器处理模型可以在保留 Linux 内核原有分配机制的前提下, 根据新增的标识内存分配请求来源的分配标志, 调用新增的主内核和内核模块专用内存缓存完成对应的内存分配操作, 从而实现主内核和内核模块在内存分配时的隔离。

5.2 研究展望

虽然本文在内存分配隔离方面取得一定成果, 但由于本人经验不足以及知识水平有限, 还存在以下几点需要进一步研究:

(1) 本文实现的新分配器处理模型主要目的是解决混合页问题, 并未在此基础上实现完整的页表分配机制, 为主内核和内核模块创建多套独立的页表, 后续应进行这方面的研究。

(2) 本文实现的新分配器处理模型, 是一种理论模型, 它提供的是一种面向 Linux 内核空间的内存分配隔离方法, 在实际使用时, 如何给所有内核模块尤其是 Linux 内核原有内核模块添加分配标志是该模型面临的主要难题。后续可进行这方面的研究。

参考文献

- [1] 金峰. 基于微信小程序的家用物联网系统开发[D].杭州:浙江大学,2019.
- [2] 任欢. 基于无线传感器网络的生物信号测量系统的研究[D].秦皇岛:燕山大学,2009.
- [3] 陈可昕,刘宇涛. 一种基于内存隔离的关键数据保护机制[J]. 计算机与现代化,2016(02):75-81.
- [4] 邓良. 不可信内核环境下的系统安全技术研究[D].南京:南京大学,2016.
- [5] 吕腾飞. 基于 Linux 内核页表构建内核隔离空间的研究及实现[D].南京:南京大学,2017.
- [6] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity[C]. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2005.
- [7] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity[C]. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2014.
- [8] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT[C]. In *29th IEEE Symposium on Security and Privacy, S&P*, 2008.
- [9] X. Xiong and P. Liu. Silver: Fine-grained and transparent protection domain primitives in commodity os kernel[C]. In *Proceeding of the International Symposium on Recent Advances in Intrusion Detection*, 2013:123-122.
- [10] R. Nikolaev and G. Back. VirtuOS: an operating system with kernel virtualization[C]. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013:116-132.
- [11] D. Oliveira, J. Navarro, N. Wetzel, and M. Bucci. Ianus: secure and holistic coexistence with kernel extensions - a immune system-inspired approach[C]. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014:1672-1679.
- [12] D. Evtvushkin and M. Ozsoy. Iso-X:A flexible architecture for hardware-managed isolated execution[C]. In *Proceedings of the Annual International Symposium on Microarchitecture*, 2015:190-202.
- [13] Y. Liu, T. Zhou, and K. Kexin. Thwarting memory disclosure with efficient hypervisor enforced intra-domain isolation[C]. In *Proceedings of the 22th ACM Conference on Computer and Communications Security*. 2015.
- [14] 钱振江,刘永俊,汤力,姚宇峰,黄皓. KCapISO:一种基于 HybridHP 的宏内核操作系统载入模块权限隔离方案[J].计算机学报,2016,39(03):552-561.
- [15] 陈国良,朱艳军. 基于 Linux 的多核实时任务调度算法改进[J].计算机测量与控制,2020,28(11):238-241+246.
- [16] 胡志希,戴新发,徐士伟. 一种可配置的虚拟机内存隔离方法[J].计算机与数字工程,2016,44(08):1548-1552.
- [17] F. Tommaso, J. Patrick, L. Christopher, and S. Ahmad. IMIX: In-Process Memory Isolation Extension[C]. In *Proceeding of the 27th USENIX Security Symposium*, 2018.
- [18] W. Zheng, Y. Wu, X. Wu, and C. Feng. A survey of Intel SGX and its applications[J]. *Frontiers of Computer Science*,2020,15(3).

- [19] M. Choi, B. Jang, and M. Kim. Efficient Security Method Using Mobile Virtualization Technology And Trustzone of ARM[J]. *Journal of Digital Convergence*, 2014, 12(10).
- [20] 陈吉强. 异构内存系统中页面管理的优化设计[D]. 合肥: 中国科学技术大学, 2019.
- [21] 常宸. 基于内核旁路技术的工业控制系统网络审计产品的设计与实现[D]. 北京: 北京邮电大学, 2018.
- [22] 张航. 基于系统调用的文件系统入侵检测的设计与实现[D]. 武汉: 华中科技大学, 2009.
- [23] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and M. Qunaibit. Losing control: On the effectiveness of control-flow integrity under stack attacks[C]. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2015.
- [24] T. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries[C]. In *10th ACM Symposium on Information, Computer and Communications Security, ASIACCS*, 2015.
- [25] 秦杰杰. Redis 数据库在非易失性内存上的交换技术的研究与实现[D]. 重庆: 重庆大学, 2018.
- [26] 王小银, 陈莉君. Linux 内核中内存池的实现及应用[J]. *西安邮电学院学报*, 2011, 16(04): 40-43.
- [27] 张士林. 基于嵌入式 Linux 的串口自定义键盘驱动开发[J]. *信息通信*, 2019(12): 291-292.
- [28] 杨瑞. Linux 操作系统内核分析与研究[J]. *计算机光盘软件与应用*, 2015, 18(01): 163-164.
- [29] Brian Foote. 北京 SPIN 译. 程序设计的模式语言 卷 4[M]. 北京: 清华大学, 2004.
- [30] 朱宗卫. 基于系统时空行为特征的内存功耗优化研究[D]. 合肥: 中国科学技术大学, 2014.
- [31] 胡雨翠. 嵌入式实时系统 ARTs-OS 的动态内存管理研究[D]. 武汉: 华中科技大学, 2010.
- [32] 李颂. 嵌入式设备接入 HPC 系统关键技术研究[D]. 厦门: 厦门大学, 2007.
- [33] 胡志希. Xen 虚拟机内存安全隔离技术研究与设计[D]. 北京: 中国舰船研究院, 2016.
- [34] 张华, 刘晶晶. 多级缓存的内存分配器 MCMalloc 应用研究[J]. *信息化研究*, 2020, 46(01): 70-75.
- [35] 王小强, 刘鹏, 罗军, 罗宏伟, 王之哲. 内核页表隔离对 x86 架构处理器性能影响评测[J]. *电子产品可靠性与环境试验*, 2020, 38(05): 75-79.

图表索引

图 2-1 内存缓存的组成.....	16
图 3-1 SLAB 中内存缓存的数据结构.....	19
图 3-2 SLUB 中内存缓存的数据结构.....	20
图 3-3 SLUB 分配器改进的方法框架.....	22
图 3-4 内存缓存的类型.....	23
图 3-5 伙伴系统分配器改进的方法框架.....	32
图 3-6 完整的内存分配隔离方法理论模型.....	33
图 4-1 create_unique_id()函数新增代码.....	36
图 4-2 主内核的专用 kmem_cache 结构数组.....	37
图 4-3 内核模块的专用 kmem_cache 结构数组.....	37
图 4-4 只指定主内核分配标志的测试模块 1.....	38
图 4-5 测试模块 1 加载前的属性.....	39
图 4-6 测试模块 1 加载后的属性值.....	39
图 4-7 只指定内核模块分配标志的测试模块 2.....	40
图 4-8 测试模块 2 加载前的属性值.....	40
图 4-9 测试模块 2 加载后的属性值.....	40
图 4-10 同时指定两种分配标志的测试模块 3.....	41
表 2-1 传统区域修饰符.....	12
表 2-2 行为修饰符.....	12
表 2-3 类型标志.....	13
表 2-4 类型标志与修饰符的关联关系.....	14
表 4-1 属性值说明.....	39
表 4-2 性能测试表.....	42

作者简历及攻读硕士学位期间取得的研究成果

一、作者简历

姓名：段鑫峰

教育经历：

2014 年 9 月至 2018 年 7 月就读于西北农林科技大学信息工程学院，专业为计算机科学与技术，取得工学学士学位；

2018 年 9 月至今就读于北京交通大学计算机与信息技术学院，专业为软件工程，研究方向为安全操作系统。

工作经历：

2019 年 3 月至今，工作于北京千秋智业图书发行有限公司，担任信息技术研发编辑。

二、参与科研项目

[1] 信息安全认证认可关键技术研究与应用，国家重点研发计划

[2] 基于鲲鹏处理器的操作系统实践教学研究与开发，教育部产学合作协同育人项目（项目号 201902146001）

[3] 基于机器人操作系统的实训项目开发，教育部产学合作协同育人项目（项目号 201901026005）

[4] 基于龙芯处理器的操作系统构建实训项目开发，教育部产学合作协同育人项目（项目号 201702025004）

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作和取得的研究成果，除了文中特别加以标注和致谢之处外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京交通大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

学位论文作者签名：段鑫峰 签字日期：2021年 6 月 1 日

学位论文数据集

表 1.1: 数据集页

关键词*	密级*	中图分类号	UDC	论文资助
系统安全; 内核 隔离; 内存分配; 块分配器; 混合 页	公开			
学位授予单位名称*		学位授予单位代码*	学位类别*	学位级别*
北京交通大学		10004	专业学位	工程硕士
论文题名*		并列题名		论文语种*
面向 Linux 内核空间的内存分配 隔离方法的研究与实现				中文
作者姓名*	段鑫峰		学号*	18140056
培养单位名称*		培养单位代码*	培养单位地址	邮编
北京交通大学		10004	北京市海淀区西直 门外上园村 3 号	100044
专业领域*		研究方向*	学制*	学位授予年*
软件工程		系统安全	两年制	2021
论文提交日期*	2021 年 6 月			
导师姓名*	翟高寿		职称*	副教授
评阅人	答辩委员会主席*		答辩委员会成员	
	胡绍海			
电子版论文提交格式 文本 () 图像 () 视频 () 音频 () 多媒体 () 其他 () 推荐格式: application/msword; application/pdf				
电子版论文出版 (发布) 者		电子版论文出版 (发布) 地		权限声明
论文总页数*	51			
共 33 项, 其中带*为必填数据, 为 21 项。				