

Bruce Perens 开源系列

深入理解Linux[®] 虚拟内存管理

Understanding the Linux[®] Virtual
Memory Manager

[爱尔兰] Mel Gorman 著
白 洛 李俊宝 刘森林 译



CHINA-PUB.COM
 北京航空航天大学出版社

译者序

这是我心仪已久的一本书,相信很多读者也会有同样的感受。

Linux 在国内的普及还不成熟,而关于内核方面的中文资料却又都偏向大而全,细节部分描述得相当不够。这些书籍往往由于篇幅问题而仅仅选择性地讲解内核源代码,不少读者无法真正吃透内核的实际实现方式,存在看不下去或不能理解等问题。还有的书籍代码注释较为粗略,省去了很多内核实现细节,初、中级水平的读者阅读起来也有困难。而另有一部分内核书籍由于时代久远,很多系统调用方面,特别是与硬件相关的代码使读者产生不少疑问,对于中、高级内核开发人员以及在较新内核版本下进行开发的人员就略显不足了。

一次较为偶然的机会我在网络上发现这一至宝,在它还是草稿版的时候就得到 LinuxCN 网站的关注。但由于种种原因一直没有和大家见面。后来痛定思痛,觉得这样一本好书不能为中国众多 Linux 爱好者学习实为可惜,终于克服种种困难翻译该书完整正式版,希望能给大家带来一定的帮助。

在众多 Linux 内核分析的书籍中,本书的特点非常鲜明:

(1) 本书问世以前,基本上没有一套关于 Linux 虚拟内存 (Virtual Memory, VM) 的完整文档。文档的匮乏直接导致 VM 只能被很少一部分内核开发人员所完全理解。极少或根本没有任何信息对这种实现的理论基础进行描述。

(2) 本书试图弥补内存管理理论和在 Linux 中具体实现之间的差距,

并且将两个部分结合起来,是当今最能够充分理解 Linux VM 的运行机制以及将理论和实现联系在一起的书籍。本书不是以讲述理论为目标,而是先讲述理论,再结合 Linux 的实现深入剖析其细节。

(3) 本书以相对独立于硬件的角度阐述 Linux VM。

(4) 函数调用图和系统架构图一目了然,思路清晰。读者可以快速地掌握系统总体架构。

(5) 在附录里面包含了对 VM 的详细注释。读者能够更简单地弄清楚 VM 的实现过程,对 VM 系统的运行机制更加得心应手。

(6) 在剖析 2.4 内核的基础上还分析 2.6 内核中 VM 的新特性。紧跟世界 Linux 内核发展进度。

本书内容涵盖物理内存、页表管理、进程地址空间、引导内存分配器、物理页面分配、非连续内存分配、Slab 分配器、高端内存管理、页面帧回收、交换管理、共享内存虚拟文件系统等部分。

本书的翻译工作主要由白洛、李俊奎、刘森林完成。参与少量翻译工作的还有黄静端、张申、贺虎、查志勇、吕志华、夏玉、倪晓雷、vbar、newface、hs_guanqi。黄忠霖教授完成了本书的审校工作。

翻译过程中,我的家人、朋友和同学给了我莫大的支持和鼓励。感谢我的家人、朋友和同学。感谢黄忠霖教授的谆谆指导和辛勤工作。感谢 LinuxCN 的热心网友们。感谢所有关心和热爱 Linux 的人们。

由于本人才疏学浅,翻译中不免有少量错误和遗漏,望各位读者批评指正。共同学习,共同进步。

本书读者主要针对高等院校师生、研究机构科研人员、Linux 内核开发人员、应用开发人员及系统管理人员,以及任何对 VM 或内核子系统运行机制感兴趣的爱好者。

不要错过这本书,你会发现它有多神奇。

白 洛
华中科大喻家山

前 言

Linux 的发展侧重于实践而非纸上谈兵。当建议在已有的应用当中使用一种新算法或变更时,人们通常要求编码与理论一致。虚拟内存系统中使用的许多算法都是由理论学家提出的,但实现它们的方法基本上已经和原来的理论相脱离。狭义上,Linux 确实遵循了传统意义上从设计到实现的开发周期,但是随着系统在真实世界中的运转而进行的变更以及开发者直觉性的决定是很普遍的事情。

这意味着 VM 在实践中表现良好。然而,除了少量网站上未完成的部分框架和这些网站上早期的草稿以外,基本上没有一套完好无缺的 VM 文档。文档的匮乏直接导致 VM 只能被很少一部分内核开发人员所完全理解。新的开发人员在了解 VM 的运行机制时总是被告之去阅读源码。极少或根本没有任何信息对这种实现的理论基础进行描述。这就导致一个只想了解概要的人不得不投入大量的时间阅读源码和学习内存管理。

这本书讲述了 Linux 内核 2.4.22 下 VM 的实现细节,同时给出了内核 2.6 下 VM 变更情况的简介。在讨论实现的同时,也会详细介绍 Linux VM 的理论基础。这本书并不是为成为一本内存管理的书而设计的,而是为了在了解理论基础的情况下,理解 VM 为什么要以如此独特而简单的方式实现。

为完善讲述内容,在附录里面包含了对 VM 中重要部分的代码注释。这极大减少了开发者或研究人员在理解 Linux VM 的工作方式时所需投入

的时间,这是因为即使 Linux 的版本不同,其 VM 也是按照类似的代码组织方式实现的。也就是说,只要我们深入理解内核 2.4 的 VM,其后的 2.5 开发版以及 2.6 的最终发布版中的 VM,我们也能在几周之内掌握清楚。

针对的读者

任何对 VM 以及内核子系统的运行机制感兴趣的读者,在本书中都将找到满意的答案。比起其他的子系统,VM 对操作系统的整体性能具有重要的影响。VM 也是 Linux 中最难理解、文档最糟糕的一个子系统,部分原因就在于其相关文档资料的欠缺。如果事先对整个 VM 没有一个清晰的概念模型,要想分离出并理解 VM 的代码相当困难。本书就是为了给那些即将阅读源码的读者一个细节性的描述。

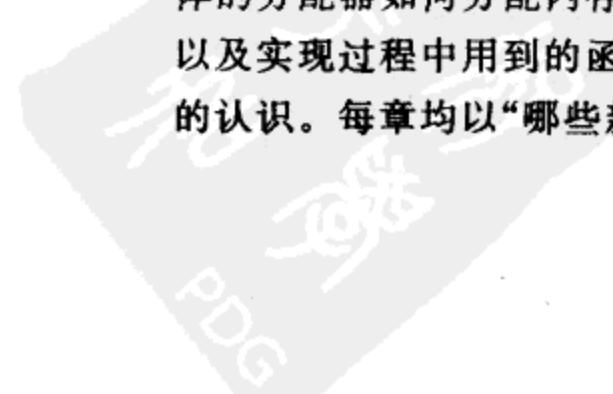
本书对那些需要改进 VM 以满足自身需要的开发人员和需要降低理解 VM 难度的读者来说非常有益。同时,对于需要与 VM 打交道的子系统开发人员以及想清晰理解现代操作系统内存管理实现方式的研究人员来说,也是获益匪浅。甚至是对一些只想更多了解该子系统的其他人员而言,他们无需阅读源码就能轻易地了解 VM 中涵盖所有细节的函数功能。

尽管如此,读者在翻阅本书之前,至少应当阅读一本有关通用操作系统方面,或是介绍通用 Linux 内核的书,并且掌握 C 语言的一般知识。尽管作者竭尽全力以使本书通俗易懂,但前提是读者必须提前了解通用操作系统的相关知识。

本书概要

在第 1 章,详细介绍如何分析、浏览以及管理源代码。本书介绍了 3 种工具,其中的主要工具是 Linux 交叉引用器(LXR),我们可以使用它来以网页或 CodeViz 的形式阅读源码,其中 CodeViz 是在研究本书的同时为产生调用图而开发的。最后一个工具就是 PatchSet,它用来管理内核和应用程序补丁。手工使用补丁比较耗时,而且使用诸如 CVS(并发版本控制系统)(<http://www.cvshome.org/>)或者 BitKeeper(<http://www.bitmover.com>)等类似的版本控制软件也不是一个好的选择。通过 PatchSet,一个简单的规范文件就解决了使用哪些源码、哪些补丁以及内核如何配置等问题。

在其后的几个章节,我们将详细讨论 Linux 中 VM 实现的各个部分,比如,如何以与体系结构无关的方式描述内存,处理器怎样管理内存以及具体的分配器如何分配内存等问题。每一章都涉及描述 Linux 行为的源码,以及实现过程中用到的函数和调用图,以使读者对代码的组织结构有清晰的认识。每章均以“哪些新特性”作为最后一节结束,用于介绍 VM 在 2.6



版本内核中的新特性。

附录包含了 VM 中大部分代码的注释,对其中一些比较复杂的部分也进行了逐句解释。即使是在两个不同的内核发布版本间,VM 的风格也是基本一致的,因而深入理解内核 2.4 中的 VM 对于理解即将发布的内核 2.6 具有重要意义。

2.6 中有哪些新特性

在编写本书时,2.6.0-test4 已经发布,因而 2.6.0 的最终版本将指日可待。幸好 2.6 中的 VM 与 2.4 比较起来,大部分都相似。但是 2.6 中依然有些新的方法和概念,忽略它们是非常可惜的。因此本书在“2.6 中有哪些新特性”这一节中将分别介绍。从一定程度上表述,阅读这些章节的前提是你已经阅读了本书的其他几个部分,因而如果你是第一次阅读本书,可以仅浏览这些章节。“2.6 中有哪些新特性”中的内容对于理解 2.5 和 2.6 的 VM 代码具有很强的辅助作用。这些节基于 2.6.0-test4 的内核版本,在 2.6 正式版本发布前,不会有太大的改变。但是,它们依然容易发生改动,所以可以将这些节的内容看作指导方针,却不能当成既定的事实。

附带光盘

原版书附带有一张光盘^{*}。强烈推荐使用光盘以方便读者更加熟悉本书,尤其是对于想通过本书获得长足进步或者需要查看附录中代码注释的读者。推荐在 GNU/Linux 系统中使用 CD,但并不要求。

原版书中的内容以 HTML、PDF 以及文本方式存放在光盘中,以方便读者进行基本的文本查询,尤其是在目录中没有读者需要的信息时。如果你所阅读的是原版书的第 1 版,根据打印截止期,你会发现光盘版和打印版有少许细微差别。

光盘几乎包含了研究本书材料过程中使用到的所有工具。其中每个工具都可以安装在任何一个装有 GNU/Linux 的系统中。光盘同时也包含了有用的文档和项目主站的参考信息,以方便读者日后的更新。

对于大多数 GNU/Linux 系统,我们可以从光盘直接运行 Web 服务器。该服务器已经经过 RedHat7.3 和 Debian Woody 的测试,但是也应当可以在任意的发行版上面测试通过 <http://localhost:10080> 网址提供了许多有用的特性:

- 为具有代码注释的函数提供查询索引。如果要查找一个没有注释

^{*} 读者可以登录 www.buaapress.com.cn 的下载中心,下载原版书中光盘的内容。

的函数,浏览器将自动重定向到 LXR。

- 提供 2.4.22 源码的网页形式可浏览副本。这将允许我们查看代码并验证所查询的代码。
- 提供一个 CodeViz 的运行版本。使用该工具可以为本书生成函数调用图。如果你觉得本书所提供的调用图不够详细,你可以自己使用该工具重新生成它。
- 在第 1 章中提到的 VMRegress,CodeViz 以及 PatchSet 工具包可以在/cdrom/software 中找到。同时还提供 gcc-3.0.4 来编译 CodeViz。

用如下命令将光盘挂载到/cdrom:

```
root@joshua:~$ mount/dev/cdrom/cdrom-o exec
```

我们使用 Apache 1.3.27(<http://www.apache.org/>)作为 Web 服务器。该服务器在/cdrom/根目录下已经编译配置并启动。如果你的发行版通常使用其他目录,你需要切换至并使用该目录。使用/cdrom/start_server 启动它。如果不出错,其输出应当如下:

```
mel@joshua:~$ /cdrom/start_server
Starting CodeViz Server: done
Starting Apache Server: done
```

访问网址:<http://localhost:10080/>

成功启动服务器后,点击浏览器指向 <http://localhost:10080/> 以使用光盘所提供的 Web 服务。运行/cdrom/stop_server 脚本就可关闭服务器,然后就可卸载 CD 了。

排版风格

本书的排版风格比较简单。通常,在谈及结构中的字段时,结构名和字段名都将被包含在 page-list 中。文件名用统一字体表示,但是 include 文件则应当在文件名外添加尖括号如:<linux/mm.h>。这些文件可以在内核源码的 include/ 目录下找到。

致 谢

完成本书是一件非常艰辛的工作。它在得到以下众多朋友的支持和帮助下研究和完成。如有遗漏,在此报以最真诚的歉意。

首先,感谢 John O'Gorman 先生(很遗憾,在本书编辑过程中已辞世),他的经历和指导使本书的格式和质量得到了极大的提高。

其次,也感谢 Prentice Hall PTR 的 Mark L. Taub 帮助我们出版本书。

这是一段有回报的和值得研究所有代码的经历。在本书完成之时,同时得到了许多学者清晰可靠的反馈意见,在此表示感谢。最后,也感谢 Bruce Perens 允许我们在其开源站点上(<http://www.perens.com/Books>)出版此书。

在研究技术的过程中,许许多多的人都给出了他们的真知灼见。Abhishek Nayani 在我们研究初期就表现出极大的热情。Ingo Oeser 对数据如何从用户空间复制内核空间作出了详细的阐述,并且还搜集了大量有价值的相关资料。同时,在我对内核代码感到迷茫和失落时,他也真诚地激励和帮助了我。Scott Kaplan 对系统中从非连续内存的分配到页面置换等许多问题做了进一步的校正。Jonathon Corbet 则提供了大量 Linux 内核发展历史的资料,该资料发布在 Linux Weekly News 的内核版。Zack Brown, Kernel Traffic 的首席专家,他是使我没有在内核邮件列表中迷失方向的唯一原因。作为 Equinox 工程的一部分,IBM 公司为我们提供了一款 xSeries350,使我们在更大型的机器上运行测试我们自己的内核。在完稿后期,Jeffrey Haran 发现了少许遗留的技术错误以及许多以前存在的语法问题。最后,也是至关重要的,Patrick Healy 确保了本书始终对那些仅熟悉 Linux 或内存管理而非专家的读者而言,通俗易懂,容易掌握。

有许多热心的人士帮助我们解决一些小的技术问题和一些本书涉及的相关材料不一致问题。他们有 Muli Ben-Yehuada, Parag Sharma, Matthew Dobson, Roger Luethi, Brian Lowe 和 Scott Crosby。他们中许多人对本书的不同部分提出了一些校正和质疑意见,这些是我开始并不了解的知识。

Carl Spalletta 在线对本书的各个方面都提出了质疑和校正。Steve Greenland 校正了许多语法问题。Philipp Marek 前后给我们提出了 90 多次的校正和质疑意见,涉及本书的方方面面。在我原以为完成了本书的很长一段时间后,Aris Sotiropoulos 依然提出了许多小型的修改建议。最后一位,我不记得他的名字,但知道他是一个杂志社的编辑,对我的早期版本,提出了多达 140 多份的修改意见,在此表示感谢。

还有 11 个人也做了少量的修改。尽管少,但由于我的疏忽有些仍然忘记了,敬请见谅。他们是 Marek Januszewski, Amit Shah, Adrian Stanciu, Andy Isaacson, Jean Francois Martinez, Glen Kaukola, Wolfgang Oertl, Michael Babcock, Kirk True, Chuck Luciano 和 David Wilson。

在 VMRegress 的开发过程中,有 9 人至始至终地帮助了我。Danny Faught 和 Paul Larson 为我作出了许多错误报告,确保了 VMRegress 在不同的内核下能正确执行。来自 OSDL 实验室的 Cliff White 为 VMRegress

提供了许多应用背景,比我自己测试案例更加广泛。OSDL 的另一位成员 Dave Olien 主要负责更新 VMRegress 使之能在 2.5.64 版本及其以后的内核版本上运行。Albert Cahalan 给我送来了所有的信息,依靠这些信息我能够在最新的 proc 工具下构建 VMRegress 函数。最后,Andrew Morton, Rik van Riel 和 Scott Kaplan 在确定需要开发哪些有效实用工具的方向上都提出了卓越的见解。

最后,也诚挚感谢以下始终给予我帮助和鼓励的人。他们是 Martin Bligh, Paul Rolland, Mohamed Ghouse, Samuel Chessman, Ersin Er, Mark Hoy, Michael Martin, Martin Gallwey, Ravi Parimi, Daniel Codt, Adnan Shafi, Xiong Quanren, Dave Airlie, Der Herr Hofrat, Ida Hallgren, Manu Anand, Eugene Teo, Diego Calleja 和 Ed Cashin 等。

总之,我要感谢以上许许多多的人,没有他们的共同努力也就无法完成本书。在此,感谢我的父母给予我经济上的支持;也感谢我的女友 Karen,在我为本书焦虑、烦躁而喋喋不休时,是她一直陪伴着我,并坚信我一定能成功。还有 Kudos 定期地让我远离电脑去锻炼身体,Daren 也时刻关注我的饮食。另外,感谢几年来,成千上万的程序员们对 GNU, Linux 内核以及其他自由软件项目的努力,正是因为他们的努力,使我 6 年前在自己的 PC 上有了第一次编程练习,并随之坚信 Linux 不再只是 Windows 用来阅读邮件的应用工具之后,获取了巨大的灵感,找到了研究对象,衷心地感谢大家。



目 录

第 1 章 简 介

| | | |
|-----|------|----|
| 1.1 | 开始启程 | 2 |
| 1.2 | 管理源码 | 4 |
| 1.3 | 浏览代码 | 9 |
| 1.4 | 阅读代码 | 11 |
| 1.5 | 提交补丁 | 12 |

第 2 章 描述物理内存

| | | |
|-----|-------------|----|
| 2.1 | 节 点 | 14 |
| 2.2 | 管理区 | 16 |
| 2.3 | 管理区初始化 | 21 |
| 2.4 | 初始化 mem_map | 21 |
| 2.5 | 页 面 | 22 |
| 2.6 | 页面映射到管理区 | 26 |
| 2.7 | 高端内存 | 26 |
| 2.8 | 2.6 中有哪些新特性 | 27 |

第 3 章 页表管理

| | | |
|------|-----------------------|----|
| 3.1 | 描述页目录 | 30 |
| 3.2 | 描述页表项 | 33 |
| 3.3 | 页表项的使用 | 34 |
| 3.4 | 页表项的转换和设置 | 36 |
| 3.5 | 页表的分配和释放 | 36 |
| 3.6 | 内核页表 | 37 |
| 3.7 | 地址和 struct page 之间的映射 | 39 |
| 3.8 | 转换后援缓冲区(TLB) | 40 |
| 3.9 | 一级 CPU 高速缓存管理 | 41 |
| 3.10 | 2.6 中有哪些新特性 | 44 |

第 4 章 进程地址空间

| | | |
|-----|--------|----|
| 4.1 | 线性地址空间 | 50 |
|-----|--------|----|

| | |
|---------------------|----|
| 4.2 地址空间的管理 | 52 |
| 4.3 进程地址空间描述符 | 53 |
| 4.4 内存区域 | 57 |
| 4.5 异常处理 | 72 |
| 4.6 缺页中断 | 74 |
| 4.7 复制到用户空间/从用户空间复制 | 81 |
| 4.8 2.6 中有哪些新特性 | 82 |

第 5 章 引导内存分配器

| | |
|----------------------|----|
| 5.1 表示引导内存映射 | 88 |
| 5.2 初始化引导内存分配器 | 89 |
| 5.3 初始化 bootmem_data | 89 |
| 5.4 分配内存 | 90 |
| 5.5 释放内存 | 91 |
| 5.6 销毁引导内存分配器 | 91 |
| 5.7 2.6 中有哪些新特性 | 93 |

第 6 章 物理页面分析

| | |
|--------------------|-----|
| 6.1 管理空闲块 | 94 |
| 6.2 分配页面 | 96 |
| 6.3 释放页面 | 98 |
| 6.4 获得空闲页面(GFP)标志位 | 99 |
| 6.5 进程标志位 | 101 |
| 6.6 防止碎片 | 102 |
| 6.7 2.6 中有哪些新特性 | 102 |

第 7 章 非连续内存分配

| | |
|-----------------|-----|
| 7.1 描述虚拟内存区 | 105 |
| 7.2 分配非连续区域 | 106 |
| 7.3 释放非连续内存 | 108 |
| 7.4 2.6 中有哪些新特性 | 109 |

第 8 章 Slab 分配器

| | |
|-----------|-----|
| 8.1 高速缓存 | 113 |
| 8.2 Slabs | 124 |
| 8.3 对象 | 131 |

| | |
|--------------------|-----|
| 8.4 指定大小的高速缓存 | 133 |
| 8.5 per-CPU 对象高速缓存 | 135 |
| 8.6 初始化 slab 分配器 | 137 |
| 8.7 伙伴分配器接口 | 138 |
| 8.8 2.6 中有哪些新特性 | 138 |

第 9 章 高端内存管理

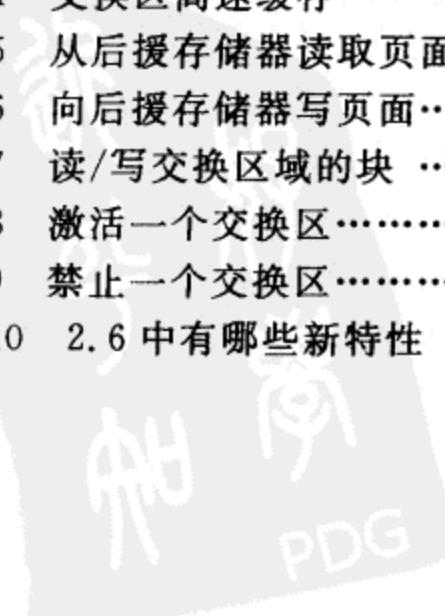
| | |
|-------------------|-----|
| 9.1 管理 PKMap 地址空间 | 140 |
| 9.2 映射高端内存页面 | 141 |
| 9.3 解除页面映射 | 143 |
| 9.4 原子性的映射高端内存页面 | 143 |
| 9.5 弹性缓冲区 | 144 |
| 9.6 紧急池 | 146 |
| 9.7 2.6 中有哪些新特性 | 147 |

第 10 章 页面帧回收

| | |
|-----------------------|-----|
| 10.1 页面替换策略 | 150 |
| 10.2 页面高速缓存 | 151 |
| 10.3 LRU 链表 | 156 |
| 10.4 收缩所有的高速缓存 | 159 |
| 10.5 换出进程页面 | 161 |
| 10.6 页面换出守护程序(kswapd) | 162 |
| 10.7 2.6 中有哪些新特性 | 162 |

第 11 章 交换管理

| | |
|-------------------|-----|
| 11.1 描述交换区 | 166 |
| 11.2 映射页表项到交换项 | 169 |
| 11.3 分配一个交换槽 | 170 |
| 11.4 交换区高速缓存 | 171 |
| 11.5 从后援存储器读取页面 | 174 |
| 11.6 向后援存储器写页面 | 174 |
| 11.7 读/写交换区域的块 | 175 |
| 11.8 激活一个交换区 | 177 |
| 11.9 禁止一个交换区 | 178 |
| 11.10 2.6 中有哪些新特性 | 179 |



第 12 章 共享内存虚拟文件系统

| | | |
|------|----------------|-----|
| 12.1 | 初始化虚拟文件系统 | 181 |
| 12.2 | 使用 shmem 函数 | 182 |
| 12.3 | 在 tmpfs 中创建文件 | 184 |
| 12.4 | 虚拟文件中的缺页中断 | 186 |
| 12.5 | tmps 中的文件操作 | 188 |
| 12.6 | tmpfs 中的索引节点操作 | 188 |
| 12.7 | 建立共享区 | 189 |
| 12.8 | System V IPC | 189 |
| 12.9 | 2.6 中有哪些新特性 | 192 |

第 13 章 内存溢出管理

| | | |
|------|-------------|-----|
| 13.1 | 检查可用内存 | 193 |
| 13.2 | 确定 OOM 状态 | 194 |
| 13.3 | 选择进程 | 194 |
| 13.4 | 杀死选定的进程 | 195 |
| 13.5 | 是这样吗？ | 195 |
| 13.6 | 2.6 中有哪些新特性 | 195 |

第 14 章 结束语**附录 A 介绍****附录 B 描述物理内存**

| | | |
|-----|--------|-----|
| B.1 | 初始化管理区 | 201 |
| B.2 | 页面操作 | 214 |

附录 C 页表管理

| | | |
|-----|-------|-----|
| C.1 | 初始化页表 | 219 |
| C.2 | 遍历页表 | 227 |

附录 D 进程地址空间

| | | |
|-----|---------|-----|
| D.1 | 进程内存描述符 | 232 |
| D.2 | 创建内存区域 | 239 |
| D.3 | 查找内存区域 | 285 |

| | |
|------------------|-----|
| D. 4 对内存区域上锁和解锁 | 291 |
| D. 5 缺页中断 | 304 |
| D. 6 页面相关的磁盘 I/O | 330 |

附录 E 启动内存分配

| | |
|-----------------|-----|
| E. 1 初始化引导内存分配器 | 370 |
| E. 2 分配内存 | 372 |
| E. 3 释放内存 | 381 |
| E. 4 释放引导内存分配器 | 383 |

附录 F 物理页面分配

| | |
|-------------|-----|
| F. 1 分配页面 | 391 |
| F. 2 分配辅助函数 | 402 |
| F. 3 释放页面 | 404 |
| F. 4 释放辅助函数 | 409 |

附录 G 不连续内存分配

| | |
|-----------------|-----|
| G. 1 分配一块非连续的区域 | 411 |
| G. 2 释放一块非连续区域 | 420 |

附录 H Slab 分配器

| | |
|---------------------|-----|
| H. 1 高速缓存控制 | 427 |
| H. 2 Slabs | 446 |
| H. 3 对象 | 452 |
| H. 4 指定大小的高速缓存 | 466 |
| H. 5 Per-CPU 对象高速缓存 | 469 |
| H. 6 初始化 Slab 分配器 | 476 |
| H. 7 与伙伴分配器的接口 | 477 |

附录 I 高端内存管理

| | |
|-------------------|-----|
| I. 1 映射高端内存页面 | 479 |
| I. 2 自动映射高端内存页面 | 484 |
| I. 3 解除页面映射 | 485 |
| I. 4 自动解除高端内存页面映射 | 487 |
| I. 5 弹性缓冲区 | 488 |
| I. 6 紧急池 | 495 |

附录 J 页面帧回收

| | | |
|------|-------------------|-----|
| J. 1 | 页面高速缓存操作 | 500 |
| J. 2 | LRU 链表操作 | 510 |
| J. 3 | 重填充 inactive_list | 514 |
| J. 4 | 从 LRU 链表回收页面 | 515 |
| J. 5 | 收缩所有高速缓存 | 522 |
| J. 6 | 换出进程页面 | 525 |
| J. 7 | 页面交换守护程序 | 536 |

附录 K 交换管理

| | | |
|------|---------|-----|
| K. 1 | 查找空闲项 | 543 |
| K. 2 | 交换高速缓存 | 548 |
| K. 3 | 交换区 I/O | 555 |
| K. 4 | 激活一个交换区 | 564 |
| K. 5 | 禁止一个交换区 | 575 |

附录 L 共享内存虚拟文件系统

| | | |
|------|----------------|-----|
| L. 1 | 初始化 shmfs | 591 |
| L. 2 | 在 tmpfs 中创建文件 | 596 |
| L. 3 | tmpfs 中的文件操作 | 600 |
| L. 4 | tmpfs 中的索引节点操作 | 613 |
| L. 5 | 虚拟文件中的缺页中断 | 622 |
| L. 6 | 交换空间交互 | 633 |
| L. 7 | 建立共享区 | 639 |
| L. 8 | System V IPC | 642 |

附录 M 内存溢出管理

| | | |
|------|--------------|-----|
| M. 1 | 确定可用内存 | 650 |
| M. 2 | 检查 OOM 并从中恢复 | 652 |

参考文献

第 1 章

简介

Linux 是相对比较新的操作系统,逐渐为商业界、学术界和自由软件世界所关注。作为一个成熟的操作系统,它的特征、功能、性能日趋完善,但与此同时它的复杂度也在不断增加。表 1.1 中所列的是内核源代码和子目录 mm/ 的大小和行数。这还不包括机器相关代码和缓冲区管理代码,虽然不能准确评价其复杂度,但仍然具有一定的指导意义。

表 1.1 内核规模复杂度

| 版本 | 发布日期 | 总体大小/MB | mm 目录大小/KB | 代码行数 |
|-------------|-------------|---------|------------|--------|
| 1.0 | 3月13日,1992年 | 5.9 | 96 | 3 109 |
| 1.2.13 | 2月8日,1995年 | 11 | 136 | 4 531 |
| 2.0.39 | 1月9日,2001年 | 35 | 204 | 6 792 |
| 2.2.22 | 9月16日,2002年 | 93 | 292 | 9 554 |
| 2.4.22 | 8月25日,2003年 | 181 | 436 | 15 724 |
| 2.6.0-test4 | 8月22日,2003年 | 261 | 604 | 21 714 |

对于开放源码项目,人们通常告知新手直接从源代码或针对新手的邮件列表中(<http://www.kernelnewbies.org>)获取答案。本书对于新手非常合适,可以帮助新手在数周内理解 VM。同时,本书花了大量的章节在内存管理部分的源码上,从而使浏览源代码比较容易,浏览源代码已不再是问题。

有些讲述操作系统的书籍,如 *Understanding the Linux Kernel* [BC00][BC03],往往涵盖内核的方方面面,除了设备驱动以外却没有明显单独讲述某一块的。这些书籍,尤其是 *Understanding the Linux Kernel*,虽然对内核提出了非常有价值的见解,但是它们除研究内核普遍关心的内容外并没有提到 VM 的独特细节。从本书中却可以了解到为什么 ZONE_NOR-

MAL 必须为 896 MiB, 以及 Per-cpu 中的高速缓存如何运作的细节。在 VM 的另一块, 诸如引导内存分配器和 VM 文件系统等这些研究内核时一般不去关心的细节, 本书中同样有所涉及。

为了能充分理解内核的运行机制, 研究人员必须一行一行地阅读源代码。本书明确地只针对 VM, 因此投入理解内核机制的时间将从数月锐减到数周。书中主要部分省去的细节在代码注释部分也会有详细描述。

在这一章中, 我们将介绍如何从开放源码项目获取信息的基础知识, 以及如何管理、浏览和理解代码的方法。如果你并不打算阅读实际的代码, 可以直接跳到第 2 章。

1.1 开始启程

在开始理解内核代码时, 其中一个最大的障碍就是从哪里开始, 以及如何较容易地管理和浏览代码, 如何获得对代码结构的总体把握。如果你在邮件列表上咨询的话, 人们会提出一些如何研究代码的建议, 但是更适合个人的方法则必须由每个开发者自己去研究。接下来的部分将介绍一些理解开源代码的有用规则, 以及具体指导如何针对内核利用这些规则。

1.1.1 配置和建立

对于任何开源项目而言, 首先要做的就是下载源码并阅读安装说明文档。一般情况下, 代码的根目录里都附带有一个 README 或者 INSTALL 文件[FF02]。实际上, 一些自动化构建工具, 比如 automake 都是要求存在安装文件中的。这些文件包括如何配置和安装软件包, 或者告诉你在什么地方可以找到相关的参考信息。Linux 也不例外, 因为它也包括了一个 README 文件, 用于描述如何配置和安装内核。

第二步就是构建软件。在早期, 对于许多项目来说必备的条件便是编辑 makefile 文件, 但是现在已经很少再采用这样的方式。通常情况下, 自由软件至少会使用 autoconf 来自动测试构建环境, 并使用 automake 简化 makefile 的创建, 因此构建过程就像下面一样简单:

```
mel@joshua: project $ ./configure && make
```

一些久远的项目, 如 Linux 内核, 采用它自带的配置工具, 而另一些大型的项目如 ApacheWeb 服务器则具有大量的配置选项, 但是这些配置脚本通常还只是后继步骤的开始。对于内核而言, 配置工作由 makefile 文件和相关支撑工具完成。最简单的配置工作就像下面一样:

```
mel@joshua: linux-2.4.22 $ make config
```

接着一连串的问题可能会接踵而来：如我们应该建立什么类型的内核呢？在诸多此类的问题得到解答后，编译内核将是十分简单的：

```
mel@joshua: linux-2.4.22 $ make bzImage && make modules
```

关于配置以及编译内核的详细向导，可以参阅内核 HOWTO 文档，在本书中不会讨论这些细节。现在，假定你有一个完整构建好了的内核，我们将开始指导你这个新内核是如何运作的。

1.1.2 信息来源

开源项目一般都有一个主页，之所以如此特别是因为自由项目集中在一些站点中如 <http://www.sourceforge.net>。主页上包括可用文档的链接以及指导你如何加入一些可用的邮件列表。文档或多或少是存在的，即便它小型化到只是一个简单的 README 文件，但至少它可以拿来读。如果项目很久远了或相当大，则网站上将会有相应的常见问题列表(FAQ)页面。

接下来，加入开发的邮件列表并沉浸于其中。这意味着你可以订阅邮件列表并阅读它，但并不一定要发表任何信息。邮件列表是开发者们相互交流的首选方式，其次是网络即时聊天(IRC)以及在线新闻组，或者 UseNet。邮件列表一般包括许多实现细节的讨论，阅读这些曾经留下的档案是非常重要的，至少可以了解开发者社区以及目前正在进行的活动。在需要对实现细节有疑问而在当前文档里查询不到实现细节时，邮件列表的档案应当是你首先要搜索的地方。如果你有问题要询问开发人员，请花点时间先搜索相关的问题然后以“适当的方式”[RM01]提问。虽然人们会回答“显而易见”的问题，但持续地问一周前就被提过的问题或者已经在文档中写得很清楚的问题，对于你个人而言没有任何帮助。

现在，如何在 Linux 中实施上述所说的方案呢？首当其冲的便是文档。在代码树最上层有一个 README 文件，还有在 Documentation/路径下详细的可用信息。一系列关于 UNIX 设计的书[Vah96]，特别是那些针对 Linux 的书[BC00]，对代码均有详尽的解释。

内核开发中关于源码信息还存在于诸多优秀在线资源中，其中之一便是周刊性质的 Linux Weekly News(<http://www.lwn.net>)中的“内核版块”。这个版块同样也报道一大部分和 Linux 相关的话题，很值得进行一般性的阅读。虽然内核方面并没有类似于这样的网站，但有一个近似的站点 <http://www.kernelnewbies.org>，不论是对于新手还是老手，其中都有许多非常好的关于内核的信息来源。

Linux Kernel Mailing List (LKML) 的 FAQ 可以在 <http://www.tux.org/lkml/> 找到，它覆盖了从内核发展历程到如何加入其邮件列表的方方面面。邮件列表在许多站点都有存档，但是一般会选择如下镜像 <http://marc.theaimsgroup.com/?l=linux-kernel>。请注意邮

件列表是非常大量的列表,去阅读它是很让人生畏的,在 Kernel Traffic 站点 <http://kt.zork.net/kernel-traffic/> 提供有一周概要。

目前为止所提到的站点和来源,包括的都是关于内核的一般性信息,然而专门关于内存管理的代码也是有的。一个 Linux-MM 站点 <http://www.linux-mm.org> 就包括了专门与内存管理相关的文档和 linux-mm 邮件列表。这个列表与主流列表比起来要次要一些,可以在 <http://mail.nl.linux.org/linux-mm/> 找到其存档。

最后要提到的站点便是 Kernel Trap,网址为 <http://www.kerneltrap.org>。这个站点包括许多有用的关于内核的一般性文章。虽然它并不是专门针对 Linux 的,但是也包括了许多 Linux 的相关文章和内核开发者的交互信息。

很明显,在看源码以前大量的可用信息可供参考。在具备了足够的经验以后,再直接来参考源码将是非常快的,所以,在开始阶段,请首先查看其他的信息来源。

1.2 管理源码

内核的正式版本和分支版本主要以压缩文档的格式(.tar.bz)发布,你可以从离你地域最近的内核源码库中得到。如在爱尔兰,镜像位于 [ftp://ftp.ie.kernel.org/](http://ftp.ie.kernel.org/)。内核的分支版本通常可以看作是代码维护树中的一个节点。例如,在写代码时,2.2.x 的分支版本由 Alan Cox 发布,2.4.x 的分支版本由 Marcelo Tosatti 发布,而 2.5.x 的分支版本则由 Linus Torvalds 发布。在每个发布版本中,都会含有完整的 tar 文件和一个包含新旧发布版本之间差异的小补丁。鉴于带宽问题,打补丁将是升级的首选方法。对内核的改进一直以补丁的方式实现,统一采用 GNU 工具 diff 进行 diff 操作。

为什么要打补丁

发送补丁到邮件列表中初听起来很笨拙,但对于内核开发环境却是非常有效的。打补丁的主要优点在于很容易从其中了解到做了哪些修改,而不是逐个地比较两个完整版本中的文件。对代码比较熟悉的开发者将很容易看到所做的修改会产生什么影响,并决定它是否该被合并到源码中。除此之外,我们也很容易查看包含补丁的邮件,以获得更多的相关信息。

子 树

随着时间的推进,个别有影响力的开发者可能拥有他们自己版本的内核,这些版本可以作为主流内核源码树的一个大补丁。这些子树通常包含有还未合并到主流中或者仍处在测试阶段的特征或部分。其中有两个著名的子树,一个就是长时间影响 VM 的由 Rik Van Riel 维护的-rmap tree,另外一个就是由目前维护 stock VM 的 Andrew Morton 所维护的-mm tree。-rmap tree 有很多特色,但由于种种原因而未能并入主流。-rmap tree 受到 FreeBSD VM 很

大的影响,但与 stock VM 有很多的差别。-mm tree 与-rmap tree 也有很大的差异,因为它是一棵带有补丁的测试树,这些补丁在并入分支内核前都会得到测试。

BitKeeper

最近,一些开发者已经开始使用一个叫做 BitKeeper 的源码控制系统(<http://www.bitmover.com>),它是 Linux 首选的一种所有权版本控制系统。BitKeeper 允许开发者拥有他们自己的分布式版本,其他用户则可以从其他的树中“取出”一系列称为变更集的补丁。传统的版本控制软件都依赖于中心服务器,所以这个分布式特征正是与传统的版本控制软件的一个非常重要的区别。

BitKeeper 允许给每个补丁附加注释,而这些注释会作为每个内核发布信息的一部分显示出来。对于 Linux,就意味着补丁保存有最初提交给补丁的 email 或者来自树的信息,这样内核的开发进程就会更透明。一旦发布,来自每个开发者的补丁摘要会作为清单列出,而且还有详细的补丁说明。

由于 BitKeeper 是一个具有版权的产品,所以电子邮件和补丁仍然被认为是在代码变更时进行讨论的有效方法。事实上,除非首次出现在主要邮件列表上有所讨论,否则一些补丁并不会被接受,因为代码的质量被认为与 peer review[Ray02]的数量是直接挂钩的。由于 BitKeeper 代码维护树的导出格式可以用开源软件如 CVS 来访问,使得打补丁依旧成为升级的较佳方式。这也意味着开发者们没有必要非得采用 BitKeeper 才能为内核做贡献,但是这个工具也应该是开发者们需要考虑的东西。

1.2.1 Diff 和 Patch

diff 和 patch 分别是用来创建补丁和使用补丁的工具,也是可从 GNU 站点获得的 GNU 实用工具。diff 用来创建补丁,而 patch 则用来使用已经创建好的补丁。虽然这个工具有许多的选项,但一般使用的是一些“常用选项”。

利用 diff 生成的补丁总是统一的格式,其中有变更所涉及的 C 函数,列表放置在内核源码根目录的上一层目录中。一个统一的 diff 文件除了说明不同之处的几行外,还包括有更多的信息。开始是 diff 比较的两个文件名和创建日期。后面的部分由一个或多个“大块”组成。每个块的第一行由@@标记开始,接着是其在源码中起始行数,以及该块在源码中被应用时的前后行数。这个块包括了所作变更前后的行数,以此帮助人们阅读。每行起始于+、-或者空白。如果是+,表示该行是新加的。如果是一,则该行是被删除了的,而一个空白代表将该行保持不变。放到内核源码根目录的上一层目录的原因是可以让开发者容易而快速地看到 patch 被用到了哪个版本上,并且每个 patch 都用相同的方式生成,可使运行补丁的脚本更为简单。

让我们先来看一个简单的例子,它只对 mm/page_alloc.c 做一点变更,加了少量的注释。补丁生成的过程如下。注意该命令应当只在一行上,末尾使用了反斜线。

```
mel@joshua: kernels/ $ diff -up \
    linux-2.4.22-clean/mm/page_alloc.c \
    linux-2.4.22-mel/mm/page_alloc.c > example.patch
```

这将在两个文件之间生成统一格式的 diff(-u 开关)文件,并将补丁写入 example.patch 文件,如例 1 所示。它也显示了被修改的 C 函数名。

例 1

```
--- linux-2.4.22-clean/mm/page_alloc.c Thu Sep 4 03:53:15 2003
+++ linux-2.4.22-mel/mm/page_alloc.c Thu Sep 3 03:54:07 2003
@@ -76,8 +76,23 @@
     * triggers coalescing into a block of larger size.
     *
     * -- wli
+
+ * There is a brief explanation of how a buddy algorithm works at
+ * http://www.memorymanagement.org/articles/alloc.html . A better
+ * idea is to read the explanation from a book like UNIX Internals
+ * by Uresh Vahalia
+
+ */
+/**/
+
+ * __free_pages_ok - Returns pages to the buddy allocator
+ * @page: The first page of the block to be freed
+ * @order: 2^order number of pages are freed
+
+ * This function returns the pages allocated by __alloc_pages and
+ * tries to merge buddies if possible. Do not call directly, use
+ * free_pages()
+ */
+
static void FASTCALL(__free_pages_ok (struct page * page, unsigned
int order));
static void __free_pages_ok (struct page * page, unsigned int order)
{
```

在这个补丁中,即使是粗略地看也可以发现文件 pag_alloc.c 中被修改的地方,即从 76 行

开始,新增加的行有+标记。在一个补丁文件当中,有若干个起始于标记@@的“块”。每个块在应用补丁时会被独立处理。

一般地,补丁分为两类:即如前述的发送到邮件列表中的纯文本,以及压缩为 gzip(.gz 扩展)或者 bzip2(.bz2 扩展)的补丁文件。通常假定补丁来自内核源码树根目录的上一级。这意味着,即便补丁生成在某个目录的上一级,也可以在当前目录通过选项-p1 为内核源码树根目录提交补丁。

一般地,在一棵干净的树上打纯文本补丁非常简单,过程如下:

```
mel@joshua: kernels/ $ cd linux-2.4.22-clean/
mel@joshua: linux-2.4.22-clean/ $ patch -p1 < ../example.patch
patching file mm/page_alloc.c
mel@joshua: linux-2.4.22-clean/ $
```

为了提交压缩的补丁,只需要简单地多做一点事情就可以完成,先解压补丁到标准输出终端(stdout):

```
mel@joshua: linux-2.4.22-mel/ $ gzip -dc ../example.patch.gz | patch -p1
```

当应用某个块时,如果行号不同,将会输出块号和需要偏移的行的数量。这些一般是安全的警告并且可以忽略掉。如果在上下文中差别很细微,该块将被应用,并打印其混淆的程度,以表示其应该做双重的检查。如果应用某块时失败了,它将保存在 filename.c.rej,原始文件将保存为 filename.c.orig,这时必须通过手工进行提交。

1.2.2 通过 PatchSet 进行基本源码管理

对未压缩的源代码,一开始补丁的管理以及内核的构建会很有趣,但很快你会觉得枯燥无味。为了减少你对补丁管理的厌倦,还在写这本书的时候,一种称为 PatchSet 的工具被开发了出来,它可以很容易地管理内核源码及补丁,并能减少大量的枯燥工作。可以从 <http://www.csn.ul.ie/~mel/projects/patchset/> 或压缩光盘里获得其完整的资料。

下 载

下载内核及补丁本身是很有趣的,所提供的脚本也使得工作很简单。首先,应该编辑配置文件 etc/patchset.conf,参数 KERNEL_MIRROR 应改为 <http://www.kernel.org/> 的本地镜像。在做完以上工作后,使用 download 脚本开始下载补丁及内核源码。使用的脚本很简单:

```
mel@joshua: patchset/ $ download 2.4.18
# Will download the 2.4.18 kernel source
mel@joshua: patchset/ $ download -p 2.4.19
```

```
# Will download a patch for 2.4.19
mel@joshua: patchset/ $ download -p -b 2.4.20
# Will download a bzip2 patch for 2.4.20
```

在相关的源码或补丁下载完以后,就开始配置如何建立内核了。

配置构建

设置配置文件要指定使用哪个内核源源码压缩包、应用哪个补丁、构建内核时使用什么配置及目标内核如何命名。下面是一个构建内核 2.4.20-rmap15f 时配置文件的例子:

```
linux-2.4.18.tar.gz
2.4.20-rmap15f
config_generic

1 patch-2.4.19.gz
1 patch-2.4.20.bz2
1 2.4.20-rmap15f
```

第 1 行表示从 linux-2.4.18.tar.gz 开始解压源码树,第 2 行表示新内核的名字是 2.4.20-rmap15f。之所以选择 2.4.20 作为例子,是因为在写本书时 rmap 补丁和最新的稳定版有冲突不可用。要查看是否有升级的 rmap 补丁,请到 <http://surriel.com/patches/>。第 3 行指定编译时用到的配置文件。接下来的每行有两部分,第 1 部分表示使用的补丁深度,数字用于打补丁的-p 开关,就像 1.1.1 小节中所讨论的,在源码目录里打补丁时通常是 1。第 2 部分是补丁目录中补丁的名称。上述例子中,在构建内核树 2.4.20-rmap15f 之前,将提交 2 个补丁,将内核 2.4.18 更新至 2.4.20。

如果内核配置文件很简单,可以使用 createset 脚本来产生一个集文件。它采用内核版本作为参数,并推测如何在可用的源码及补丁上构建内核:

```
mel@joshua: patchset/ $ createset 2.4.20
```

构建内核

包中一般有 3 个脚本文件。第 1 个脚本,称为 make-kernel.sh,用于解压内核到 kernels/ 目录,并根据需求构建内核。如果发行版是 Debian,使用-d 开关可以创建 Debian 包并能简化安装。第 2 个脚本,称为 make-gengraph.sh,用于解压内核,但是不同于构建一个可安装的内核,它将生成一个需要使用 CodeViz 的文件用来创建调用图,这个在下个章节将会涉及。第 3 个,称为 make-lxr.sh,它使用 LXR 完成内核的安装工作。

生成 Diff 文件

最终,你需要查看在两棵树中的文件的不同之处,或者产生一个“diff”记录你自己所做的

变更。有3个脚本可以使该工作变得简单。第1个是 `setclean` 脚本,用于设置需要比较的源码树。第2个是 `setworking` 脚本,用于设置你用来比较或用来操作的源码树的路径。第3个是 `difftree` 脚本,用于在两棵树之间生成相比较的文件或目录的 `diff` 文件。例1中演示了如何生成 `diff` 文件,操作如下:

```
mel@joshua: patchset/ $ setclean linux-2.4.22-clean
mel@joshua: patchset/ $ setworking linux-2.4.22-mel
mel@joshua: patchset/ $ difftree mm/page_alloc.c
```

`diff` 文件通过建议使用的 `diff` 操作,生成的一个关于 C 函数上下文的统一文件。当跟踪两棵树直接的变更时,有两个附加的脚本非常有用,它们是 `diffstruct` 和 `difffunc`。它们用来打印两个单独的结构或函数的不同之处。第一次使用时,必须使用 `-f` 开关以记录结构或函数在哪个源码文件中被定义,但这仅仅是第一次使用时才有必要。

1.3 浏览代码

当代码很小并且容易管理时,浏览代码并不是太难的事情,因为操作都集中在同一文件里,并且模块之间耦合度也不高。不幸的是,内核并不总是这样的情况。涉及的函数可能分布在多个文件里,或者写成包含在头文件中的内联函数。更复杂的是,涉及的文件可能隐藏在体系结构相关的目录下,这时要跟踪它们将很费时间。

早期解决代码浏览问题的方法是用 `ctags`(<http://ctags.sourceforge.net/>)从一系列源文件生成标签文件。这些标签用于转向 C 文件中的行,就像 `Vi` 和 `Emacs` 中一样。如果同一个标签有多个实例,比如多个函数采用同一个函数名,就从列表里选出正确的一个。在编写代码的时候这个方法是最有效的,因为它允许在一个终端窗口中快速地浏览代码。

一个更有效的浏览方法是利用工具 `LXR`,可以从 <http://lxr.linux.no/> 中获得。这个工具具有将源代码组织为可浏览的网页的功能。标识符如全局变量、宏和函数都变成了超链接。点击某个链接时,定义标识符的位置以及参考该定义的文件及行都会显示出来。这使得浏览代码变得非常方便,在第一次阅读代码时很有必要。

该工具安装简单,在随书的 CD 中包括一个可浏览内核版本 2.4.22 的版本。书中所涉及的代码都是通过 `LXR` 工具解析出来的,所以在摘录中可以很清楚地看清行号。

1.3.1 分析代码流

由于各模块共享的代码分散分布在多个 C 文件中,所以人工跟踪给定代码路径上被调用

的函数特别困难,只能浏览所有的代码。对于广而深的代码路径,这个本应很简单的问题将会花费掉很多的时间。

有一个小巧且高效的工具 CodeViz,用来生成调用图,在随书 CD 中可以获得。它使用一个经修改过的 C 或 C++ 编译器收集信息,并生成图。从 <http://www.csn.ul.ie/~mel/projects/codeviz/> 中可以获得。

当这个经过修改的编译器编译时,会为每个 C 文件生成一个扩展名为.cdep 的文件,这个.cdep 文件包含了所有在 C 文件中的函数定义及函数调用。这些文件通过一个叫 genfull 的程序提炼,生成一张所有代码的调用关系图,通过 dot 转化,它是项目 GraphViz project2.6 的一部分,可以在 <http://www.graphviz.org> 中获得。

还在写本书时,针对特定计算机编译内核,由 genfull 生成的 full.graph 文件中,总共有 40 165 个项。这个调用图本身是没什么用的,因为它实在太大了,因而需要第二个称为 gen-graph 的工具。这个工具的基本用法,是以一个或多个函数名为参数,生成以给定函数为根节点的调用图。生成的文件可以用 ghostview 或 gv 工具来查看。

由于用户在生成图时并不是都关心深度或者需要的函数,所以在生成调用图时有 3 个选项。第 1 个是深度限制,超过指定的深度的调用链将被忽略。第 2 个是忽略指定的函数,包括它所调用的。第 3 个是只显示一个函数,但如果它能很方便地转换为单独的调用图或者它是一个目前暂时不用的已知 API,就不截断它。

这些文档中出现的所有调用图都是由 CodeViz 工具生成的,所以在第一次浏览时,有一个调用图可用,那么理解一个子系统就是相当简单的了。这个工具不仅仅用于内核分析,它已经经过了大量的基于 C 的开源项目考验,有很广阔的应用前景。

1.3.2 生成图的简单过程

如果已经安装了 PatchSet 和 CodeViz,那么生成和查看本书中如图 3.4 所示的调用图需要使用以下一系列命令。为简单起见,先忽略掉输出的命令:

```
mel@joshua: patchset $ download 2.4.22
mel@joshua: patchset $ createset 2.4.22
mel@joshua: patchset $ make-gengraph.sh 2.4.22
mel@joshua: patchset $ cd kernels/linux-2.4.22
mel@joshua: linux-2.4.22 $ gengraph -t -s "alloc_bootmem_low_pages \
zone_sizes_init" -f paging_init
mel@joshua: linux-2.4.22 $ gv paging_init.ps
```

1.4 阅读代码

当新手问到怎样开始阅读源代码时,有经验的开发者通常会建议他们从初始化代码处开始。我并不认为这对任何人都是最好的方法,因为初始化是与体系结构相关的,需要有详实的硬件知识才能理解。而且它也没有说明像 VM 这样的子系统是如何工作的,只有在初始化后期建立运行系统的内存时才看得见。

理解 VM 最好的起始点就是本书,包括代码注释。它相当全面地描述了 VM,但又不是过分复杂。VM 在后期开发中变得更为复杂,但是都是这里所讲述内容的必要拓展。

当需要结合后继的 VM 重新考察源代码时,通常最好是从独立性强、与其他部分耦合最少的模块开始。对于 VM,最好的切入点就是 `mm/oom_kill.c` 里的内存溢出(OOM)管理器。它对 VM 的那部分描述得非常明晰,当系统内存很低时,VM 如何选择杀死一个进程。因为这个过程涉及了 VM 的许多方面,在本书的最后部分会涉及。第 2 个是 `mm/vmalloc.c` 里的非连续内存分配器,将在第 7 章中讨论,因为它几乎只在一个文件里。第 3 个是 `mm/page_alloc.c` 里的物理页面分配器,将在第 6 章中讨论,也是几乎只在一个文件里。第 4 个是在第 4 章中讨论过的为进程创建虚拟内存地址(VMA)和内存区域的过程。在这些系统间,含有大多数在内核其他地方较为普遍的代码风格,这有助于理解其他复杂的系统,例如页面替换策略和缓冲区 I/O。

有经验的开发者的第二个建议,就是对 VM 进行基准测试。有许多的基准测试程序可用,但通常会使用的一些是 ConTest(<http://members.optusnet.com.au/ckolivas/contest/>), SPEC(<http://www.specbench.org/>), lmbench(<http://www.bitmover.com/lmbench/>) 和 dbench(<http://freshmeat.net/projects/dbench/>)。对多种目的,这些基准测试程序已经够用了。

不幸的是,要准确地测试 VM 是件困难的事,而且标准测试通常是基于对任务计时的情况的,比如内核编译的过程。有一个称为 VMRegress 的工具,可以在 <http://www.csn.ie/~mel/projects/vmregress/> 中获得,建立了一个完全从起始,衰退以及标准测试的工具,满足了测试 VM 的最基本需要。VM Regress 结合了内核模块及用户空间的工具,以可复写的方式测试 VM 的小区域,并且以一个很大的参考字符串来对页面替换策略进行基准测试。它可以用做开发测试实用程序的框架,它还有许多 Perl 函数库能成为开发内核模块的帮手。然而,该工具还处于开发的早期阶段,所以请小心使用。

1.5 提交补丁

在内核源代码子目录 Documentation/中有两个文件 (SubmittingPatches 和 Coding-Style), 涉及一些重要的基础知识。但是很少有文档描述如何合并补丁。因此这部分将简要介绍如何管理补丁。

首先, 也是最重要的, 内核的代码风格必须一致。如果代码风格和主流内核源码的风格不一致的话, 无论采取何种技术都会给代码合并带来麻烦。一旦开发好了一个补丁, 首要的问题是确定把它送到哪儿。内核开发中对由谁处理补丁以及如何提交补丁有明确的规定。作为一个例子, 我们以 2.5.x 开发为例。

首先检查确定补丁是否够小或细微。如果是, 就提交到内核主流邮件列表。如果没有副作用, 就会被反馈至 Trivial Patch Monkey。Trivial Patch Monkey 的行为正如其名。它会将这些小补丁汇集起来发送给适当的人群。这种机制对代码中的文档、注释和 1 行的补丁最合适。

补丁以一系列松散的环的形式组织管理, Linus 在环的中央, 对于是否接受补丁到主内核树中有最终的决定权。Linus 通常只接受他的助手们提交的补丁, 大概有 10 多人, Linus 非常信任经他们修正过的正确代码。比如 Andrew Morton 就是他的助手, 在写本书时由他负责维护 VM。所有关于 VM 的变更在提交给 Linus 之前必经过 Andrew Morton 之手。他们通常各自维护一个专门的子系统, 但某些时候, 如果他们认为非常重要的话, 也会为另一个子系统提交补丁给 Linus。

每个助手在各个不同子系统上都是活跃的开发者。和 Linus 一样, 他们也有一部分他们信任的开发者, 对发送的补丁见解深厚, 他们同样乐于接收有益于子系统的补丁。他们信任的那些对子系统有重要影响的开发者将在文件 MAINTAINERS 中列出。次要一点的开发者将列于各子系统邮件列表中, VM 没有维护者的列表, 但有邮件列表。

维护者和助手对于是否接受补丁至关重要。一般而言, Linus 不会就一个单独的重要补丁进行辩论, 他更倾向于听取他的助手们对于大体积补丁的深刻理解。

总的说来, 一个新补丁应当首先发送到子系统邮件列表, 而后才被发送到主列表上经过广泛的讨论。直至没有回复时, 才会发送到相应的维护者手中, 如果没有维护者就发送到助手那里。维护者或助手收到后, 就很有可能合并补丁。其中关键的一点就是补丁和想法要尽早发布, 这样, 还处在管理期间的开发者们就有机会看到它们。值得注意的是, 有的情况下存在大量的补丁难以合并进主流树, 针对它们很少有讨论或者甚至没有讨论。最近的一个例子就是项目 Linux Kernel Crash Dump, 到现在仍然没有合并到主流树中, 因为没有从助手那里得到很好的反馈, 也没有供应商表示有力的支持。

第 2 章

描述物理内存

Linux 适用于广泛的体系结构,因此需要用一种与体系结构无关的方式来描述内存。本章描述了用于记录影响 VM 行为的内存簇、页面和标志位的结构。

在 VM 中首要的普遍概念就是非一致内存访问(NUMA)。对大型机器而言,内存会分成许多簇,依据簇与处理器“距离”的不同,访问不同的簇会有不同的代价。比如,可能把内存的一个簇指派给每个处理器,或者某个簇和设备卡很近,很适合内存直接访问(DMA),那么就指派给该设备。

每个簇都被认为是一个节点,在 Linux 中的 `struct pg_data_t` 体现了这一概念,既便在一致内存访问(UMA)体系结构中亦是如此。该结构通常用 `pg_data_t` 来引用。系统中的每个节点链接到一个以 `NULL` 结尾的 `pgdat_list` 链表中,而其中的每个节点利用 `pg_data_tnode_next` 字段链接到下一个节点。对于像 PC 这种采用 UMA 结构的机器,只使用了一个称作 `contig_page_data` 的静态 `pg_data_t` 结构。在 2.1 节中会对节点作进一步的讨论。

在内存中,每个节点被分成很多的称为管理区(zone)的块,用于表示内存中的某个范围。不要混淆管理区和基于管理区的分配器,因为两者是完全不相关的。一个管理区由一个 `struct zone_struct` 描述,并被定义为 `zone_t`,且每个管理区的类型都是 `ZONE_DMA`,`ZONE_NORMAL` 或者 `ZONE_HIGHMEM` 中的一种。不同的管理区类型适合不同类型的用途。`ZONE_DMA` 指低端范围的物理内存,某些工业标准体系结构(ISA)设备需要用到它。`ZONE_NORMAL` 部分的内存由内核直接映射到线性地址空间的较高部分,在 4.1 节会进一步讨论。`ZONE_HIGHMEM` 是系统中预留的可用内存空间,不被内核直接映射。

对于 x86 机器,管理区的示例如下:

`ZONE_DMA` 内存的首部 16 MB;

`ZONE_NORMAL` 16 MB~896 MB;

`ZONE_HIGHMEM` 896 MB~末尾。

许多内核操作只有通过 ZONE_NORMAL 才能完成,因此 ZONE_NORMAL 是影响系统性能最为重要的管理区。这些管理区会在 2.2 节讨论。系统的内存划分成大小确定的许多块,这些块也称为页面帧。每个物理页面帧由一个 struct page 描述,所有的结构都存储在一个全局 mem_map 数组中,该数组通常存放在 ZONE_NORMAL 的首部,或者就在小内存系统中为装入内核映象而预留的区域之后。2.4 节讨论了 struct page 的细节问题,在 3.7 节将讨论全局 mem_map 数组的细节问题。在图 2.1 中解释了所有这些结构之间的基本关系。

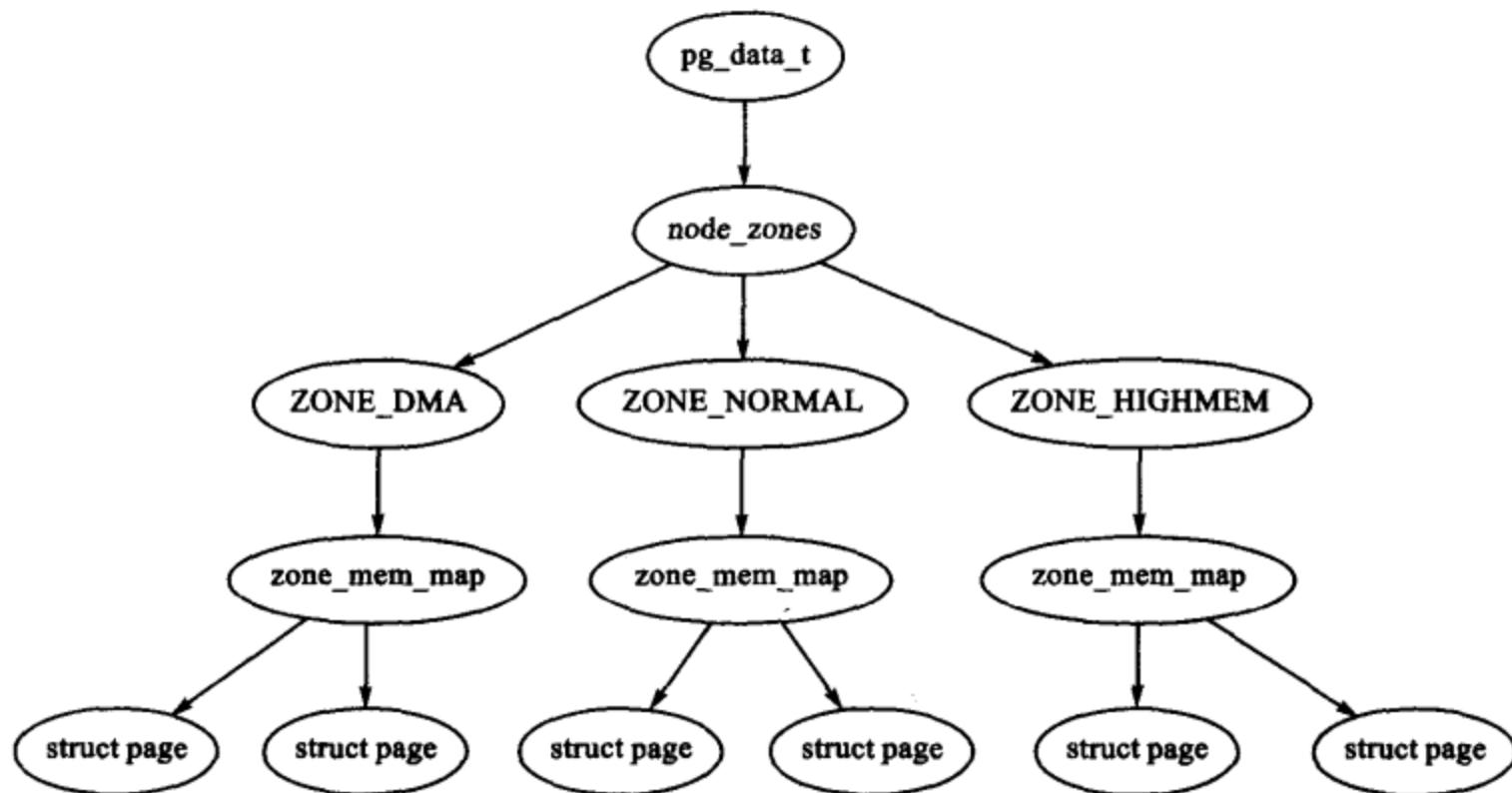


图 2.1 节点、管理区和页面的关系

由于能够被内核直接访问的内存空间(ZONE_NORMAL)大小有限,所以 Linux 提出高端内存的概念,它将会在 2.7 节中进一步讨论。这一章讨论在引入高端内存管理以前,节点、管理区和页面如何工作。

2.1 节 点

正如前面所提到的,内存中的每个节点都由 pg_data_t 描述,而 pg_data_t 由 struct pglist_data 定义而来。在分配一个页面时,Linux 采用节点局部分配的策略,从最靠近运行中的 CPU 的节点分配内存。由于进程往往是在同一个 CPU 上运行,因此从当前节点得到的内存很可能被用到。结构体在<linux/mmzone.h>文件中声明如下所示:

```
129 typedef struct pglist_data {
```

```

130 zone_t node_zones[MAX_NR_ZONES];
131 zonelist_t node_zonelists[GFP_ZONEMASK + 1];
132 int nr_zones;
133 struct page * node_mem_map;
134 unsigned long * valid_addr_bitmap;
135 struct bootmem_data * bdata;
136 unsigned long node_start_paddr;
137 unsigned long node_start_mapnr;
138 unsigned long node_size;
139 int node_id;
140 struct pglist_data * node_next;
141 } pg_data_t;

```

现在我们简要地介绍每个字段。

node_zones: 该节点所在管理区为 ZONE_HIGHMEM, ZONE_NORMAL 和 ZONE_DMA。

node_zonelists: 它按分配时的管理区顺序排列。在调用 free_area_init_core() 时, 通过 mm/page_alloc.c 文件中的 build_zonelists() 建立顺序。如果在 ZONE_HIGHMEM 中分配失败, 就有可能还原成 ZONE_NORMAL 或 ZONE_DMA。

nr_zones: 表示该节点中的管理区数目, 在 1 到 3 之间。并不是所有的节点都有 3 个管理区。例如, 一个 CPU 簇就可能没有 ZONE_DMA。

node_mem_map: 指 struct page 数组中的第一个页面, 代表该节点中的每个物理帧。它被放置在全局 mem_map 数组中。

valid_addr_bitmap: 一张描述内存节点中“空洞”的位图, 因为并没有实际的内存空间存在。事实上, 它只在 Sparc 和 Sparc64 体系结构中使用, 而在任何其他体系结构中可以忽略掉这种情况。

bdata: 指向内存引导程序, 在第 5 章中有介绍。

node_start_paddr: 节点的起始物理地址。无符号长整型并不是最佳选择, 因为它会在 ia32 上被物理地址拓展 (PAE) 以及一些 PowerPC 上的变量如 PPC440GP 拆散。PAE 在第 2.7 节有讨论。一种更好的解决方法是用页面帧号 (PFN) 记录该节点的起始物理地址。一个 PFN 仅是一个简单的物理内存索引, 以页面大小为基础的单位计算。物理地址的 PFN 一般定义为 (page_phys_addr >> PAGE_SHIFT)。

node_start_mapnr: 它指出该节点在全局 mem_map 中的页面偏移。在 free_area_init_core() 中, 通过计算 mem_map 与该节点的局部 mem_map 中称为 lmem_map 之间的页面数, 从而得到页面偏移。

node_size: 这个管理区中的页面总数。

node_id: 节点的 ID 号 (NID), 从 0 开始。

node_next: 指向下一个节点, 该链表以 NULL 结束。

所有节点都由一个称为 pgdat_list 的链表维护。这些节点都放在该链表中, 均由函数 init_bootmem_core() 初始化节点, 在 5.3 节会描述该函数。截至最新的 2.4 内核 (>2.4.18), 对该链表操作的代码段基本上如下所示:

```
pg_data_t * pgdat;
pgdat = pgdat_list;
do {
    /* do something with pgdata_t */
    ...
} while ((pgdat = pgdat->node_next));
```

在最新的内核版本中, 有一个宏 for_each_pgdat(), 它一般定义成一个 for 循环, 以提高代码的可读性。

2.2 管理区

每个管理区由一个 struct zone_t 描述。zone_structs 用于跟踪诸如页面使用情况统计数, 空闲区域信息和锁信息等。在<linux/mmzone.h>中它的声明如下所示:

```
37 typedef struct zone_struct {
41     spinlock_t lock;
42     unsigned long free_pages;
43     unsigned long pages_min, pages_low, pages_high;
44     int need_balance;
45
49     free_area_t free_area[MAX_ORDER];
50
76     wait_queue_head_t * wait_table;
77     unsigned long wait_table_size;
78     unsigned long wait_table_shift;
79
83     struct pglist_data * zone_pgdat;
84     struct page * zone_mem_map;
85     unsigned long zone_start_paddr;
86     unsigned long zone_start_mapnr;
87
91     char * name;
```

```

92 unsigned long size;
93 } zone_t;

```

下面是对该结构的每个字段的简要解释。

lock: 并行访问时保护该管理区的自旋锁。

free_pages: 该管理区中空闲页面的总数。

pages_min, **pages_low**, **pages_high**: 这些都是管理区极值, 在下一节中会讲到这些极值。

need_balance: 该标志位通知页面换出 kswapd 平衡该管理区。当可用页面的数量达到管理区极值的某一个值时, 就需要平衡该管理区了。极值会在下一节讨论。

free_area: 空闲区域位图, 由伙伴分配器使用。

wait_table: 等待队列的哈希表, 该等待队列由等待页面释放的进程组成。这对 `wait_on_page()` 和 `unlock_page()` 非常重要。虽然所有的进程都可以在一个队列中等待, 但这可能会导致所有等待进程在被唤醒后, 都去竞争依旧被锁的页面。大量的进程像这样去尝试竞争一个共享资源, 有时被称为惊群效应。等待队列表会在 2.2.3 小节作进一步的讨论。

wait_table_size: 该哈希表的大小, 它是 2 的幂。

wait_table_shift: 定义为一个 long 型所对应的位数减去上述表大小的二进制对数。

zone_pgdat: 指向父 `pg_data_t`。

zone_mem_map: 涉及的管理区在全局 `mem_map` 中的第一页。

zone_start_paddr: 同 `node_start_paddr`。

zone_start_mapnr: 同 `node_start_mapnr`。

name: 该管理区的字符串名字, “DMA”, “Normal” 或者“HighMem”。

size: 该管理区的大小, 以页面数计算。

2.2.1 管理区极值

当系统中的可用内存很少时, 守护程序 kswapd 被唤醒开始释放页面(见第 10 章)。如果内存压力很大, 进程会同步地释放内存, 有时候这种情况被引用为 direct-reclaim 路径。影响页面换出行为的参数与 FreeBSD[McK96] 和 Solaris[MM01] 中所用的参数类似。

每个管理区都有三个极值, 分别称为 `pages_low`, `pages_min` 和 `pages_high`, 这些极值用于跟踪一个管理区承受了多大的压力。它们之间的关系如图 2.2 所示。`pages_min` 的页面数量在内存初始化阶段由函数 `free_area_init_core()` 计算出来, 并且是基于页面的管理区大小的一个比率。计算值初始化为 `ZoneSizeInPages/128`。它所能取的最小值是 20 页(在 x86 上是 80 KB), 而可能的最大值是 255 页(在 x86 上是 1 MB)。

每个极值在表示内存不足时的行为都互不相同。

pages_low: 在空闲页面数达到 `pages_low` 时, 伙伴分配器就会唤醒 kswapd 释放页面。

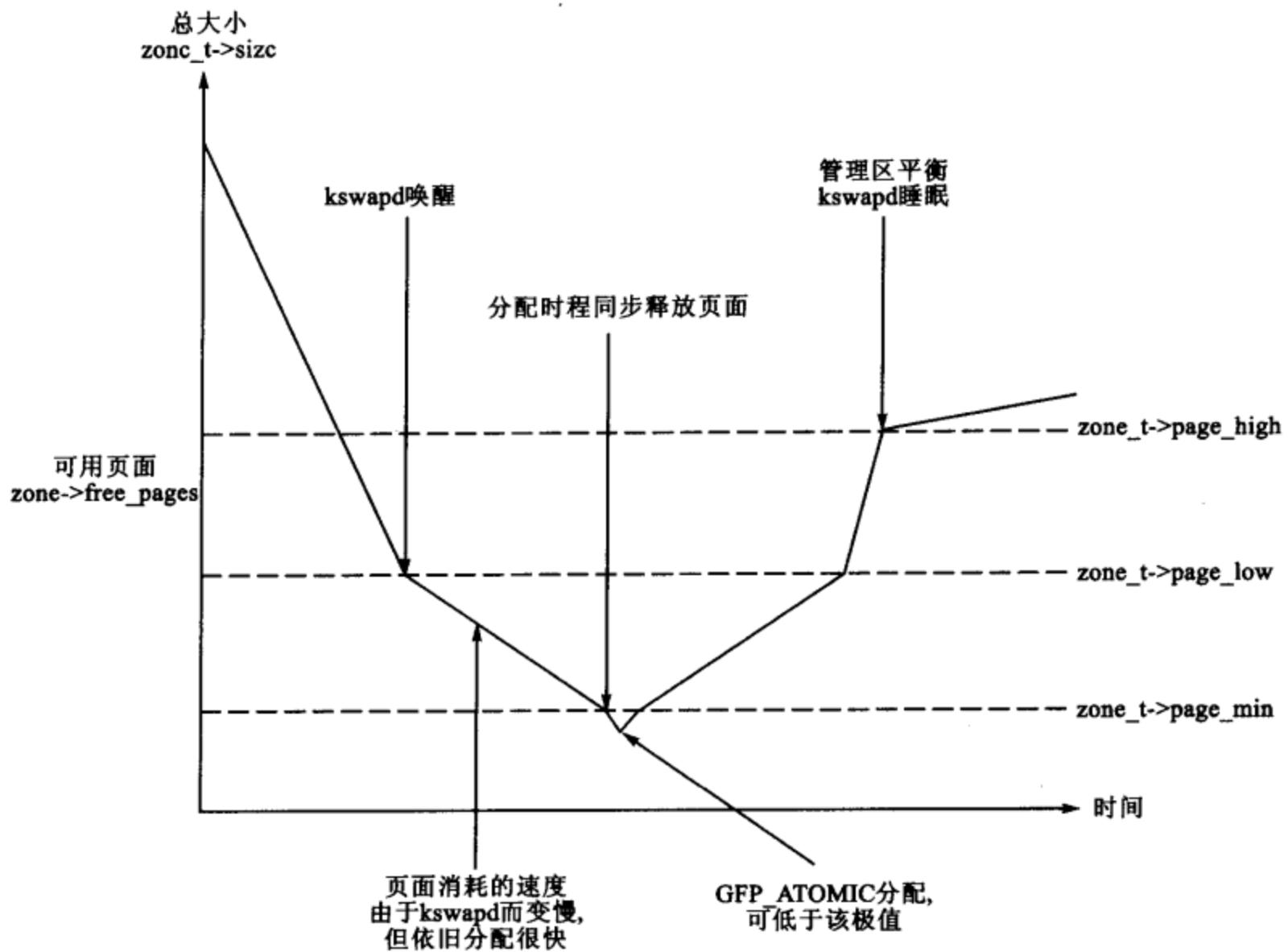


图 2.2 管理区极值

与之对应的是 Solaris 中的 lotsfree 和 FreeBSD 中的 freemin。pages_low 的默认值是 pages_min 的两倍。

pages_min：当达到 pages_min 时，分配器会以同步方式启动 kswapd，有时候这种情况被引用为 direct-reclaim 路径。在 Solaris 中没有与之等效的参数，最接近的是 desfree 或 min-free，这两个参数决定了页面换出扫描程序被唤醒的频率。

pages_high：kswapd 被唤醒并开始释放页面后，在 pages_high 个页面被释放以前，是不会认为该管理区已经“平衡”的。当达到这个极值后，kswapd 就再次睡眠。在 Solaris 中，它被称为 lotsfree，在 BSD 中，它被称为 free_target。pages_high 的默认值是 pages_min 的三倍。

在任何操作系统中，无论调用什么样的页面换出参数，它们的含义都是相同的。它们都用于决定页面换出守护程序或页面换出进程释放页面的频繁程度。

2.2.2 计算管理区大小

每个管理区的大小在 `setup_memory()` 中计算出来,如图 2.3 所示。

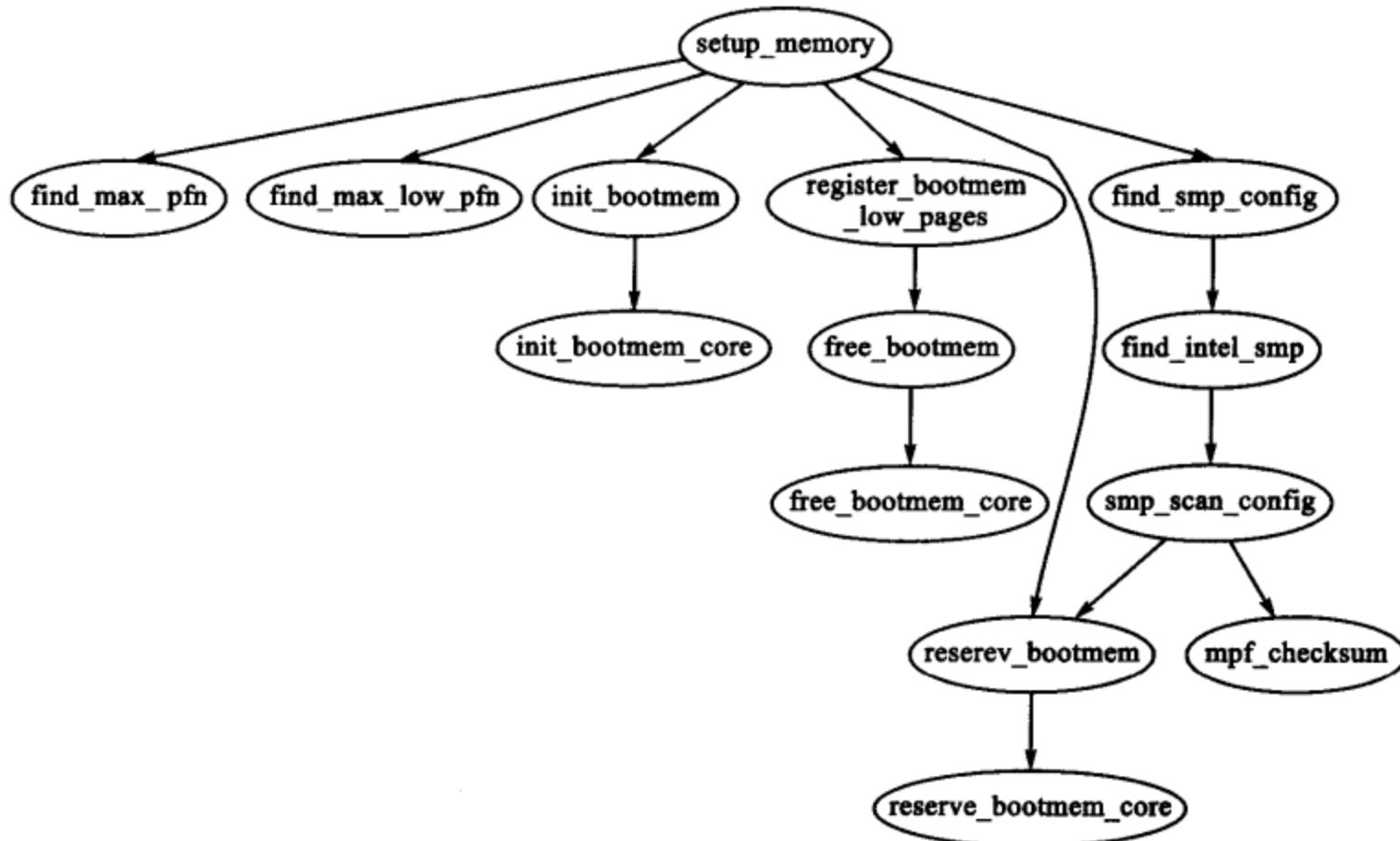


图 2.3 调用图: `setup_memory`

PFN 物理内存映射,以页面计算的偏移量。系统中第一个可用的 PFN(`min_low_pfn`)分配在被导人的内核映象末尾`_end`后的第一个页面位置。其值作为一个文件范围变量存储在 `mm/bootmem.c` 文件中,与引导内存分配器配套使用。

系统中的最后一个页面帧的 `max_pfn` 如何计算完全与体系结构相关。在 x86 的情况下,通过函数 `find_max_pfn()` 计算出 `ZONE_NORMAL` 管理区的结束位置值 `max_low_pfn`。这一块管理区是可以被内核直接访问的物理内存,并通过 `PAGE_OFFSET` 标记了内核/用户空间之间划分的线性地址空间。这个值与其他的一些值,都存储在 `mm/bootmem.c` 文件中。在内存很少的机器上, `max_pfn` 的值等于 `max_low_pfn`。

通过 `min_low_pfn`, `max_low_pfn` 和 `max_pfn` 这三个变量,系统可以直接计算出高端内存的起始位置和结束位置,表示文件范围的变量 `highstart_pfn` 和 `highend_pfn` 存储在 `arch/i286/mm/init.c` 文件中。这些值接着被物理页面分配器用来初始化高端内存页面,见 5.6 节。

2.2.3 管理区等待队列

当页面需要进行 I/O 操作时,比如页面换入或页面换出,I/O 必须被锁住以防止访问不一致的数据。使用这些页面的进程必须在 I/O 能访问前,通过调用 `wait_on_page()` 被添加到一个等待队列中。当 I/O 完成后,页面通过 `UnlockPage()` 解锁,然后等待队列上的每个进程都将被唤醒。理论上每个页面都应有一个等待队列,但是系统这样会花费大量的内存存放如此多分散的队列。Linux 的解决办法是将等待队列存储在 `zone_t` 中。基本进程如图 2.4 所示。

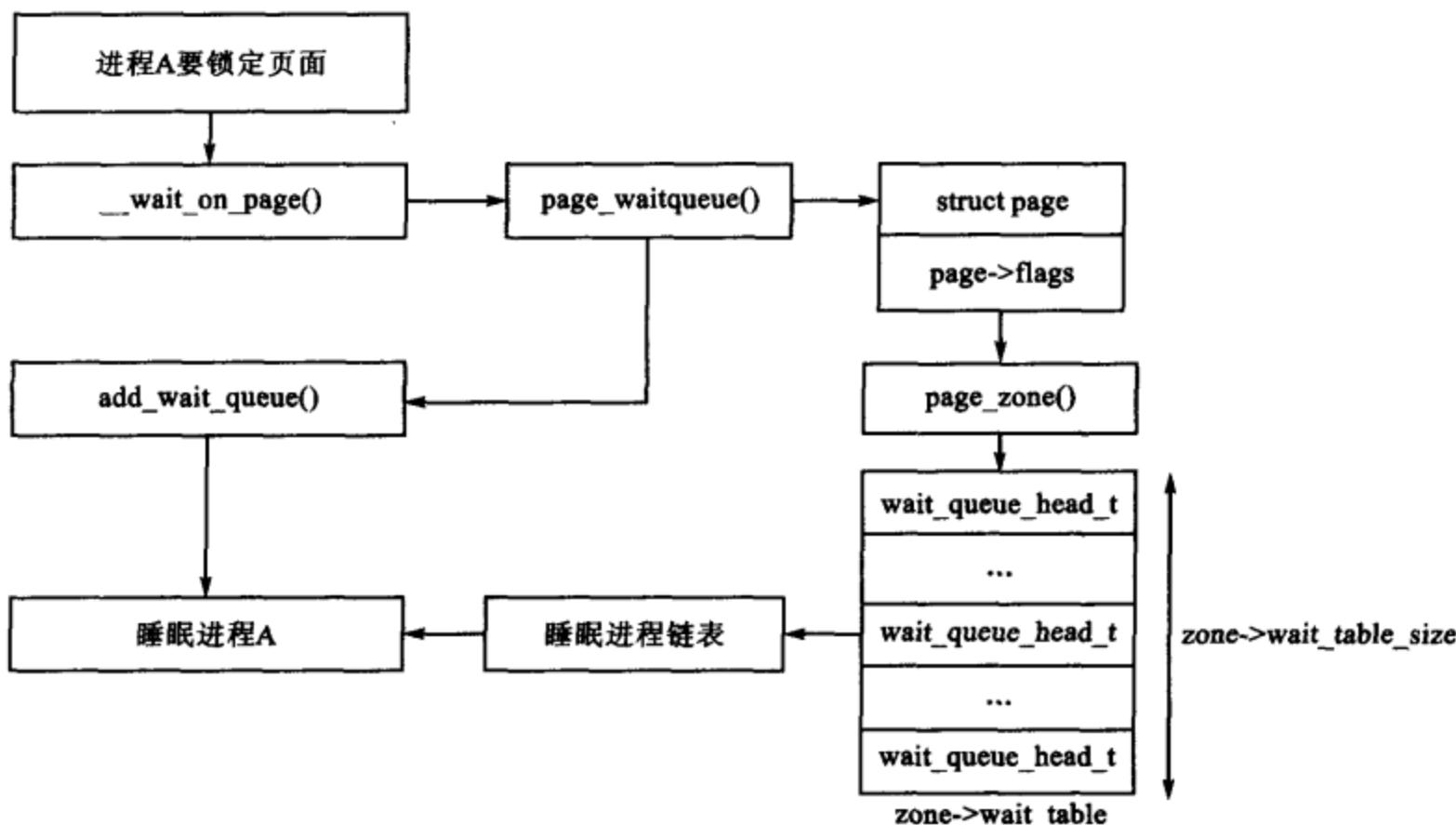


图 2.4 被锁住页面的睡眠图示

在管理区中只有一个等待队列是有可能的,但是这意味着等待该管理区中任何一个页面的所有进程在页面解锁时都将被唤醒。这会引起惊群效应的问题。Linux 的解决办法是将等待队列的哈希表存储在 `zone_t`→`wait_table` 中。在发生哈希冲突时,虽然进程也有可能会被无缘无故地唤醒,但冲突不会再发生得如此频繁了。

该表在 `free_area_init_core()` 时就被初始化了。它的大小通过 `wait_table_size()` 计算,并存储在 `zone_t`→`wait_table_size` 中。等待队列最多有 4 096 个。而小一点的队列的大小是 `NoPages/PAGE_PER_WAITQUEUE` 个队列数和 2 的幂次方的最小值,其中 `NoPages` 是该管理区中的页面数, `PAGE_PER_WAITQUEUE` 被定义为 256,即表的大小由如下等式计算出的整数部分表示:

$$\text{wait_table_size} = \log_2 \left(\frac{\text{NoPages} \times 2}{\text{PAGE_PER_WAITQUEUE}} - 1 \right)$$

`zone_t->wait_table_shift` 字段通过将一个页面地址的比特数右移以返回其在表中的索引而计算出来。函数 `page_waitqueue()` 用于返回某管理区中一个页面对应的等待队列。它一般采用基于已经被哈希的 `struct page` 的虚拟地址的乘积哈希算法。

`page_waitqueue` 一般需要用 `GOLDEN_RATIO_PRIME` 乘以该地址，并将结果 `zone_t->wait_table_shift` 的比特数右移以得到其在哈希表中的索引结果。`GOLDEN_RATIO_PRIME` [Lev00]是在系统中最接近所能表达的最大整数的 golden ratio[Knu68]的最大素数。

2.3 管理区初始化

管理区的初始化在内核页表通过函数 `paging_init()` 完全建立起来以后进行。页面表的初始化在 3.6 节会涉及。可以肯定地说，不同的系统执行这个任务虽然不一样，但它们的目标都是相同的：应当传递什么样的参数给 UMA 结构中的 `free_area_init()` 或者 NUMA 结构中的 `free_area_init_node()`。UMA 惟一需要得到的参数是 `zones_size`。参数的完整列表如下。

`nid`: 被初始化管理区中节点的逻辑标识符, `NodeID`。

`pgdat`: 节点中被初始化的 `pa_data_t`。在 UMA 结构里为 `contig_page_data`。

`pmap`: 被 `free_area_init_core()` 函数用于设置指向分配给节点的局部 `lmem_map` 数组的指针 UMA 结构中，它往往被忽略掉，因为 NUMA 将 `mem_map` 处理为起始于 `PAGE_OFFSET` 的虚拟数组。而在 UMA 中，该指针指向全局 `mem_map` 变量，目前还有 `mem_map`，都在 UMA 中被初始化。

`zones_sizes`: 一个包含每个管理区大小的数组，管理区大小以页面为单位计算。

`zone_start_paddr`: 第一个管理区的起始物理地址。

`zone_holes`: 一个包含管理区中所有内存空洞大小的数组。

核心函数 `free_area_init_core()` 用于向每个 `zone_t` 填充相关的信息，并为节点分配 `mem_map` 数组。释放管理区中的哪些页面的信息在这里并不考虑。这些信息直到引导内存分配器使用完之后才可能知道，这将在第 5 章进行讨论。

2.4 初始化 `mem_map`

`mem_map` 区域在系统启动时会被创建成下列两种方式中的某一种。在 NUMA 系统中，全局 `mem_map` 被处理为一个起始于 `PAGE_OFFSET` 的虚拟数组。`free_area_init_node()` 函数

数在系统中被每一个活动节点所调用,在节点被初始化的时候分配数组的一部分。而在 UMA 系统中,free_area_init()使用 contig_page_data 作为节点,并将全局 mem_map 作为该节点的局部 mem_map。在图 2.5 中显示了这两个函数的调用图。

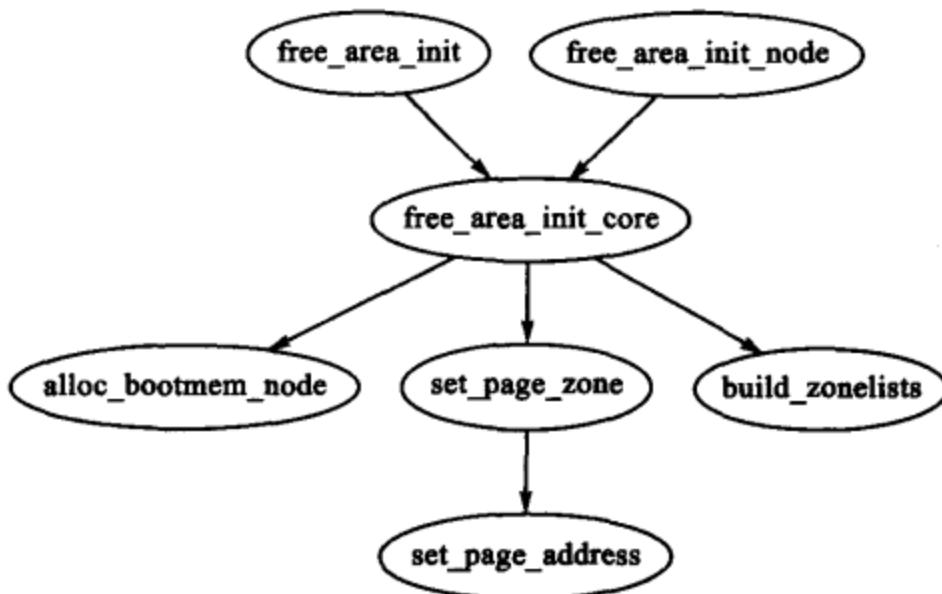


图 2.5 调用图:free_area_init()

核心函数 free_area_init_core() 为已经初始化过的节点分配局部 lmem_map。而该数组的内存通过引导内存分配器中的 alloc_bootmem_node() (见第 5 章) 分配得到。在 UMA 结构中,新分配的内存变成了全局的 mem_map,但是这和 NUMA 中的还是稍有不同的。

NUMA 结构中,分配给 lmem_map 的内存存在它们自己的内存节点中。全局 mem_map 从未被明确地分配过,取而代之的是被处理成起始于 PAGE_OFFSET 的虚拟数组。局部映射的地址存储在 pg_data_t→node_mem_map 中,也存在于虚拟 mem_map 中。对节点中的每个管理区而言,虚拟 mem_map 中表示管理区的地址存储在 zone_t→zone_mem_map 中。余下的节点都将 mem_map 作为真实的数组,因为其中只有有效的管理区会被节点所使用。

2.5 页 面

系统中的每个物理页面都有一个相关联的 struct page 用以记录该页面的状态。在内核 2.2 版本[BC00]中,该结构类似它在 System V 中的等价物,就像 unix 中的其他分支一样,该结构经常变动。它在 linux/mm.h 中声明如下:

```

152 typedef struct page {
153     struct list_head list;
154     struct address_space * mapping;
155     unsigned long index;
  
```

```

156 struct page * next_hash;
158 atomic_t count;
159 unsigned long flags;
161 struct list_head lru;
163 struct page **pprev_hash;
164 struct buffer_head * buffers;
175
176 #if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)
177 void * virtual;
179 #endif /* CONFIG_HIGMEM || WANT_PAGE_VIRTUAL */
180 } mem_map_t;

```

下面是对该结构中的各个字段的简要介绍。

list: 页面可能属于多个列表,此字段用作该列表的首部。例如,映射中的页面将属于 address_space 所记录的 3 个循环链表中的一个。这 3 个链表是 clean_pages, dirty_pages 以及 locked_pages。在 slab 分配器中,该字段存储有指向管理页面的 slab 和高速缓存结构的指针。它也用于链接空闲页面的块。

mapping: 如果文件或设备已经映射到内存,它们的索引节点会有一个相关联的 address_space。如果这个页面属于这个文件,则该字段会指向这个 address_space。如果页面是匿名的,且设置了 mapping,则 address_space 就是交换地址空间的 swapper_space。

index: 这个字段有两个用途,它的意义与该页面的状态有关。如果页面是文件映射的一部分,它就是页面在文件中的偏移。如果页面是交换高速缓存的一部分,它就是在交换地址空间中(swapper_space)address_space 的偏移量。此外,如果包含页面的块被释放以提供给一个特殊的进程,那么被释放的块的顺序(被释放页面的 2 的幂)存放在 index 中。这在函数_free_pages_ok() 中设置。

next_hash: 属于一个文件映射并被散列到索引节点及偏移中的页面。该字段将共享相同的哈希桶的页面链接在一起。

count: 页面被引用的数目。如果 count 减到 0, 它就会被释放。当页面被多个进程使用到,或者被内核用到的时候, count 就会增大。

flags: 这些标志位用于描述页面的状态。所有这些标志位在<linux/mm.h>中声明,并在表 2.1 中列出。有许多已定义的宏用于测试、清空和设置这些标志位,已在表 2.2 中列出。其中最为有用的标志位是 SetPageUptodate(), 如果在设置该位之前已经定义,那么它会调用体系结构相关的函数 arch_set_page_uptodate()。

lru: 根据页面替换策略,可能被交换出内存的页面要么会存放于 page_alloc.c 中所声明的 active_list 中,要么存放于 inactive_list 中。这是最近最少使用(LRU)链表的链表首部。这两个链表的细节将在第 10 章讨论。

pprev_hash: 是对 next_hash 的补充,使得哈希链表可以以双向链表工作。

buffers: 如果一个页面有相关的块设备缓冲区,该字段就用于跟踪 buffer_head。如果匿名页面有一个后援交换文件,那么由进程映射的该匿名页面也有一个相关的 buffer_head。这个缓冲区是必不可少的,因为页面必须与后援存储器中的文件系统定义的块同步。

virtual: 通常情况下,只有来自 ZONE_NORMAL 的页面才由内核直接映射。为了定位 ZONE_HIGHMEM 中的页面,kmap()用于为内核映射页面,这些在第 9 章有进一步的讨论,但只有一定数量的页面会被映射到。当某个页面被映射时,这就是它的虚拟地址。

类型 mem_map_t 是对 struct page 的类型定义,因此在 mem_map 数组中可以很容易就引用它。

表 2.1 描述页面状态的标志位

| 位名 | 描述 |
|------------|--|
| PG_active | 当位于 LRU active_list 链表上的页面该位被设置,并在页面移除时清除该位。它标记了页面是否处于活动状态 |
| PG_arch_1 | 直接从代码中引用,PG_arch_1 是一个体系结构相关的页面状态位。一般的代码保证了在第一次进入页面高速缓存时,该位被清除。这使得体系结构可以延迟到页面被某个进程映射后,才进行 D-Cache(见 3.9 节)刷盘 |
| PG_checked | 仅由 Ext2 文件系统使用 |
| PG_dirty | 该位显示了页面是否需要被刷新到磁盘上去。当磁盘上对应的某个页面进行了写操作后,并不是马上刷新到磁盘上去。该位保证了一个脏页面在写出以前不会被释放掉 |
| PG_error | 在磁盘 I/O 发生错误时,该位被设置 |
| PG_fs_1 | 该位被文件系统所保留,以作他用。目前,只有 NFS 用它表示一个页面是否和远程服务器同步 |
| PG_highmem | 高端内存的页面不可能长久地被内核所映射。高端内存的页面在 mem_init() 时通过该位来标记 |
| PG_lander | 该位只对页面替换策略很重要。在 VM 要换出页面时,该位将被设置,并调用 writepage() 函数。在扫描过程中,若遇到一个设置了该位以及 PG_locked 位的页面,则需要等待 I/O 完成 |
| PG_locked | 在页面必须在磁盘 I/O 所在内存时,该位被设置。在 I/O 启动时,该位被设置,并在 I/O 被完成时释放掉 |
| PG_lru | 当页面存在于 active_list 或 inactive_list 其中之一时该位被设置 |

续表 2.1

| 位名 | 描述 |
|---------------|--|
| PG_referenced | 当页面被映射了，并被映射的索引哈希表引用时，该位被设置。在 LRU 链表上移动页面做页面替换时使用该位 |
| PG_reserved | 该位标志了禁止换出的页面。该位在系统初始化时，由引导内存分配器（见第 5 章）设置。接下来该位用于标志空页面或者不存在的页面 |
| PG_slab | 该位标志了被 slab 分配器使用的页面 |
| PG_skip | 该位被某些 Sparc 体系结构用于跳过部分地址空间，但是现在不再被使用了。在 2.6 中，该位被完全废除了 |
| PG_unused | 该位表示没有被使用 |
| PG_uptodate | 当一个页面从磁盘无错误地读出时，该位被设置 |

表 2.2 用于测试、设置和清除 page→flags 的宏

| 位名 | 设置 | 测试 | 清除 |
|---------------|---------------------|------------------|-----------------------|
| PG_active | SetPageActive() | PageActive() | ClearPageActive() |
| PG_arch_1 | None | None | None |
| PG_checked | SetPageChecked() | PageChecked() | None |
| PG_dirty | SetPageDirty() | PageDirty() | ClearPageDirty() |
| PG_error | SetPageError() | PageError() | ClearPageError() |
| PG_highmem | None | PageHighMem() | None |
| PG_launder | SetPageLaunder() | PageLaunder() | ClearPageLaunder() |
| PG_locked | LockPage() | PageLocked() | UnlockPage() |
| PG_lru | TestSetPageLRU() | PageLRU() | TestClearPageLRU() |
| PG_referenced | SetPageReferenced() | PageReferenced() | ClearPageReferenced() |
| PG_reserved | SetPageReserved() | PageReserved() | ClearPageReserved() |
| PG_skip | None | None | None |
| PG_slab | PageSetSlab() | PageSlab() | PageClearSlab() |
| PG_unused | None | None | None |
| PG_uptodate | SetPageUptodate() | PageUptodate() | ClearPageUptodate() |

2.6 页面映射到管理区

在最近的 2.4.18 版本的内核中, struct page 存储有一个指向对应管理区的指针 `page→zone`。该指针在后来被认为是一种浪费, 因为如果有成千上万的这样的 struct page 存在, 那么即使是很小的指针也会消耗大量的内存空间。在更新后的内核版本中, 已经删除了该 zone 字段, 取而代之的是 `page→flags` 的最高 `ZONE_SHIFT`(在 x86 下是 8 位)位, 该 `ZONE_SHIFT` 位记录该页面所属的管理区。首先, 建立管理区的 `zone_table`。在 `linux/page_alloc.c` 中它的声明如下:

```
33 zone_t * zone_table[MAX_NR_ZONES * MAX_NR_NODES];
34 EXPORT_SYMBOL(zone_table);
```

`MAX_NR_ZONES` 是一个节点中所能容纳的管理区的最大数, 如 3 个。

`MAX_NR_NODES` 是可以存在的节点的最大数。函数 `EXPORT_SYMBOL()` 使得 `zone_table` 可以被载入模块访问。该表处理起来就像一个多维数组。在函数 `free_area_init_core()` 中, 一个节点中的所有页面都会初始化。首先它设置该表的值

```
733     zone_table[nid * MAX_NR_ZONES + j] = zone;
```

其中, `nid` 是节点 ID, `j` 是管理区索引号, `zone` 是结构 `zone_t`。对每个页面, 函数 `set_page_zone()` 的调用方式如下所示:

```
788     set_page_zone(page, nid * MAX_NR_ZONES + j);
```

其中, 参数 `page` 是管理区被设置了的页面。因此, `zone_table` 中的索引被显式地存储在页面中。

2.7 高端内存

由于内核(`ZONE_NORMAL`)中可用地址空间是有限的, 所以 Linux 内核已经支持了高端内存的概念。高端内存的两个阈值都存在于 32 位 x86 系统中, 分别是 4 GB 和 64 GB。4 GB 的限制与 32 位物理地址定位的内存容量有关。为了访问 1 GB 和 4 GB 之间的内存, 内核通过 `kmap()` 将高端内存的页面临时映射成 `ZONE_NORMAL`。这一点会在第 9 章深入讨论。

第二个 64 GB 的限制与物理地址扩展(PAE)有关, 物理地址扩展是 Intel 发明的用于允

许 32 位系统使用 RAM 的。它使用了附加的 4 位来定位内存中的地址,实现了 2^{36} 字节(64 GB)内存的定位。

在理论上,物理地址扩展(PAE)允许处理器寻址到 64 GB,但实际上,Linux 中的进程仍然不能访问如此大的 RAM,因为虚拟地址空间仍然是 4 GB。这就导致一些试图用 `malloc()` 函数分配所有 RAM 的用户感到失望。

其次,地址扩展空间(PAE)不允许内核自身拥有大量可用的 RAM。用于描述页面的 `struct page` 仍然需要 44 字节,并且用到了 `ZONE_NORMAL` 中的内核虚拟地址空间。这意味着,为了描述 1 GB 的内存,需要大约 11 MB 的内核内存。同理,为了描述 16 GB 的内存则需要耗费 176 MB 的内存,这将对 `ZONE_NORMAL` 产生非常大的压力。在不考虑其他的数据结构而使用 `ZONE_NORMAL` 时,情况看起来还不太糟糕。在最坏的情况下,甚至像页面表项(PTE)之类的很小的数据结构也需要大约 16 MB。所以在 x86 机器上的 Linux 的可用物理内存实际被限制为 16 GB。如果确实需要访问更多的内存,我的建议一般是直接去买一台 64 位机器。

2.8 2.6 中有哪些新特性

节点

大致看去,内存如何被描述好像没有多大的变化似的,但是表面上细微的变化实际上涉及的面却相当广。节点描述器 `pg_data_t` 增加了如下新的字段。

`node_start_pfn`: 替换了 `node_start_paddr` 字段。惟一的一个差异便是新的字段是一个 PFN,而不是一个物理地址。之所以变更是因为 PAE 体系结构可以访问比 32 位更多的内存,因此大于 4 GB 的节点用原来的字段是访问不到的。

`kswapd_wait`: 为 `kswapd` 新添加的等待队列链表。在 2.4 里面,页面交换守护程序有一个全局等待队列。而在 2.6 里面,每个节点都有一个 `kswapdN`,其中 N 是节点的标识符,而每个 `kswapd` 都有其自己的等待队列对应该字段。

`node_size`: 字段被移除了,取而代之的是两个新字段。之所以如此变更是因为认识到节点中会有空洞的存在,空洞是指地址后面其实没有真正存在的物理内存。

`node_present_pages`: 节点中所有物理页面的总数。

`node_spanned_pages`: 通过节点访问的所有区域,包括任何可能存在的空洞。

管理区

初始看来,两个版本中的管理区也是有很大差异的。它们不再被称为 `zone_t`,取而代之被简单地引用为 `struct zone`。第二个主要的不同是 LRU 链表。正如我们在第 10 章中看到的,

2.4 的内核有一个全局的页面链表,决定了被释放页面或进行换出页面的顺序。这些链表目前被存储在 `struct zone` 中。相关的字段如下所示。

`lru_lock`: 该管理区中 LRU 链表的自旋锁。在 2.4 里面,它是个被称为 `pagemap_lru_lock` 的全局锁。

`active_list`: 该管理区中的活动链表。这个链表和第 10 章中描述的一样,但它现在不再是全局的,而是每个管理区一个了。

`inactive_list`: 该管理区中的非活动链表。在 2.4 中,它是全局的。

`refill_counter`: 是从 `active_list` 链表上一次性移除的页面的数量,而且只有在页面替换时才考虑它。

`nr_active active_list`: 链表上的页面数量。

`nr_inactive inactive_list`: 链表上的页面数量。

`all_unreclaimable`: 当页面换出守护程序第二次扫描整个管理区里的所有页面时,依旧无法释放掉足够的页面,该字段置为 1。

`pages_scanned`: 自最后一次大量页面被回收以来,被扫描过的页面数量。在 2.6 里面,页面被一次性释放掉,而不是单独地释放某个页面,在 2.4 里面采取的是后者。

`pressure`: 权衡该管理区的扫描粒度。它是个衰退的均值,影响页面扫描器回收页面时的工作强度。

其他 3 个字段是新加进去的,但它们和管理区的尺度是有关系的。如下所示。

`zone_start_pfn`: 管理区中 PFN 的起始位置。它取代了 2.4 中的 `zone_start_paddr` 和 `zone_start_mapnr`。

`spanned_pages`: 该管理区范围内页面的数量,包括某些体系结构中存在的内存空洞。

`present_pages`: 管理区中实际存在的页面的数量。对一些体系结构而言,其值和 `spanned_pages` 是一样的。

另外一个新加的是 `struct per_cpu_pageset`,用于维护每个 CPU 上的一系列页面,以减少自旋锁的争夺。`zone->pageset` 字段是一个关于 `struct per_cpu_pageset` 的 `NR_CPU` 大小的数组,其中 `NR_CPU` 是系统中可以编译的 CPU 数量上限。而 `per-cpu` 结构会在本节的最后部分作更进一步的讨论。

最后一个新加入 `struct zone` 的便是结构中零填充。在 2.6 内核 VM 的开发过程中,逐步认识到一些自旋锁会竞争得非常厉害,很难被获取。因为大家都知道有些锁总是成对地被获取,同时又必须保证它们使用不同的高速缓冲行,这是一种很普遍的缓冲编程技巧[Sea00]。该填充管理区在 `struct zone` 中由 `ZONE_PADDING()` 宏标记,并被用于保证 `zone->lock`, `zone->lru_lock`, 以及 `zone->pageset` 字段使用不同的高速缓冲行。

页 面

最值得注意的变更就是该字段的顺序被改变了,因此相关联的项目看起来都使用了同一个

高速缓冲行。该字段除了新加了两个特性以外,本质上还是一样的。第一个特性就是采用了一个新的联合来创建 PTE 链。PTE 和页表管理相关联的会在第 3 章的结尾处进行讨论。另外一个特性便是添加了 `page->private` 字段,它包括了映射中详实的私有信息。比如,当页面是一个缓冲区页面时,该字段被用于存储一个指向 `buffer_head` 的指针。这意味着 `page->buffers` 字段也被移除掉了。最后一个重要的变更是 `page->virtual` 对高端内存的支持不再必要,只有在特定的体系结构需要时才会考虑它的存在。如何支持高端内存将在第 9 章进一步讨论。

Per-CPU 上的页面链表

在 2.4 里面,只有一个子系统会积极地尝试为任何对象维护 per-cpu 上的链表,而这个子系统就是 slab 分配器,在第 8 章会进行讨论。在 2.6 里面,这个概念则更为普遍一些,存在一个关于活动页面和不活动页面的正式概念。

在 `<linux/mmzone.h>` 文件中声明的 `struct per_cpu_pageset` 具有一个字段,该字段是一个具有两个 `per_cpu_pages` 类型的元素的数组。该数组中第 0 位的元素是为活动页面预留的,而另一个元素则是为非活动页面预留的,其中活动页面和非活动页面决定了高速缓存中的页面的活跃状态。如果知道了页面不会马上被引用,比如 I/O 预读中,那么它们就会被置为非活动页面。

`struct per_cpu_pages` 维护了链表中目前已有一系列页面,高极值和低极值决定了何时填充该集合或者释放一批页面,变量决定了一个块当中应当分配多少个页面,并最后决定在页面前的实际链表中分配多少个页面。

为建立每个 CPU 的链表,需要有一个计算每个 CPU 上有多少页面的方法。`struct page_state` 具有一系列计算的变量,比如 `pgalloc` 字段,用于跟踪分配给当前 CPU 的页面数量;而 `pswpin` 字段,用于跟踪读进交换的页面数量。该结构在 `<linux/page-flags.h>` 文件中作了详细的注解。函数 `mod_page_state()` 用于为运行中的 CPU 更新 `page_state` 字段,另外其还提供了辅助的三个宏,分别调用 `inc_page_state()`,`dec_page_state()` 和 `sub_page_state()`。

第 3 章

页表管理

Linux 不同于其他的操作系统[CP99]，它把计算机分成独立层/依赖层两层。其他的操作系统如 BSD 中使用了一种叫做 pmap 的对象来管理底层的物理页面，而 Linux 则采用了一种与具体的体系结构无关代码的三层页表机制来完成内存管理，即使底层的体系结构并不支持这个概念。虽然这个概念很容易理解，但是它同样意味着两种不同类型的页面之间的区别是很模糊的，这种方式将不同页面通过它们自身的标志位或它们所属的页表来进行区分，而非通过它们所属的对象。

那些对内存管理单元(MMU)支持不尽相同的结构可用于模拟三层页表。例如，在不支持 PAE 的 x86 机器上，两层页表就已足够。中间页目录(PMD)被定义成大小 1，它在系统初始化时直接映射为全局页目录(PGD)，并在编译时间以外的时段进行优化。不幸的是，对于那些不会自动管理高速缓存和转换后援缓冲区(TLB)的结构，当 TLB 和 CPU 高速缓存需要改变或刷新时，即便这种调用就像是在某些结构如 x86 里的空操作一样，也必须在代码里对体系结构相关的钩子进行显式的说明。这些钩子将在 3.8 节进行更一步的讨论。

本章开始将介绍如何安排这些页表，以及采用怎样的类型来分别描述页表的三个层次。然后我们将介绍虚拟地址如何被分成不同的部分以对应不同的页表。接着，我们将介绍最底层的项：PTE，以及硬件使用它其中的哪些位。然后，在我们讨论如何建立页表以及如何分配、释放页面，接着我们将会介绍一些相关的宏，这些宏有浏览页表以及设置和检查相关属性的功能。接下来的部分讨论初始化的阶段，以及介绍在系统启动过程中如何初始化页表。最后，将讨论如何利用 TLB 和 CPU 高速缓存。

3.1 描述页目录

每个进程都有一个指向其自己 PGD 的指针(`mm_struct->pgd`)，它其实就是一个物理页

面帧。该帧包括了一个 pgd_t 类型的数组,但 pgd_t 因不同的体系结构也有所不同,它的定义在<asm/page.h>文件中。不同的结构中载入页表的方式也有所不同。在 x86 结构中,进程页表的载入是通过把 mm_struct->pgd 复制到 cr3 寄存器完成,这种方式对 TLB 的刷新有副作用。事实上,这种方式体现了在不同的结构中 flush_tlb() 的实现情况。

PGD 表中每个有效的项都指向一个页面帧,此页面帧包含着一个 pmd_t 类型的 PMD 项数组,每一个 pmd_t 又指向另外的页面帧,这些页面帧由很多个 pte_t 类型的 PTE 构成,而 pte_t 最终指向包含真正用户数据的页面。当页面被交换到后援设备时,存储在 PTE 里的将是交换项,这个交换项在系统发生页面错误时在调用 do_swap_page 时作为查找页面数据的依据。页表的布局如图 3.1 所示。

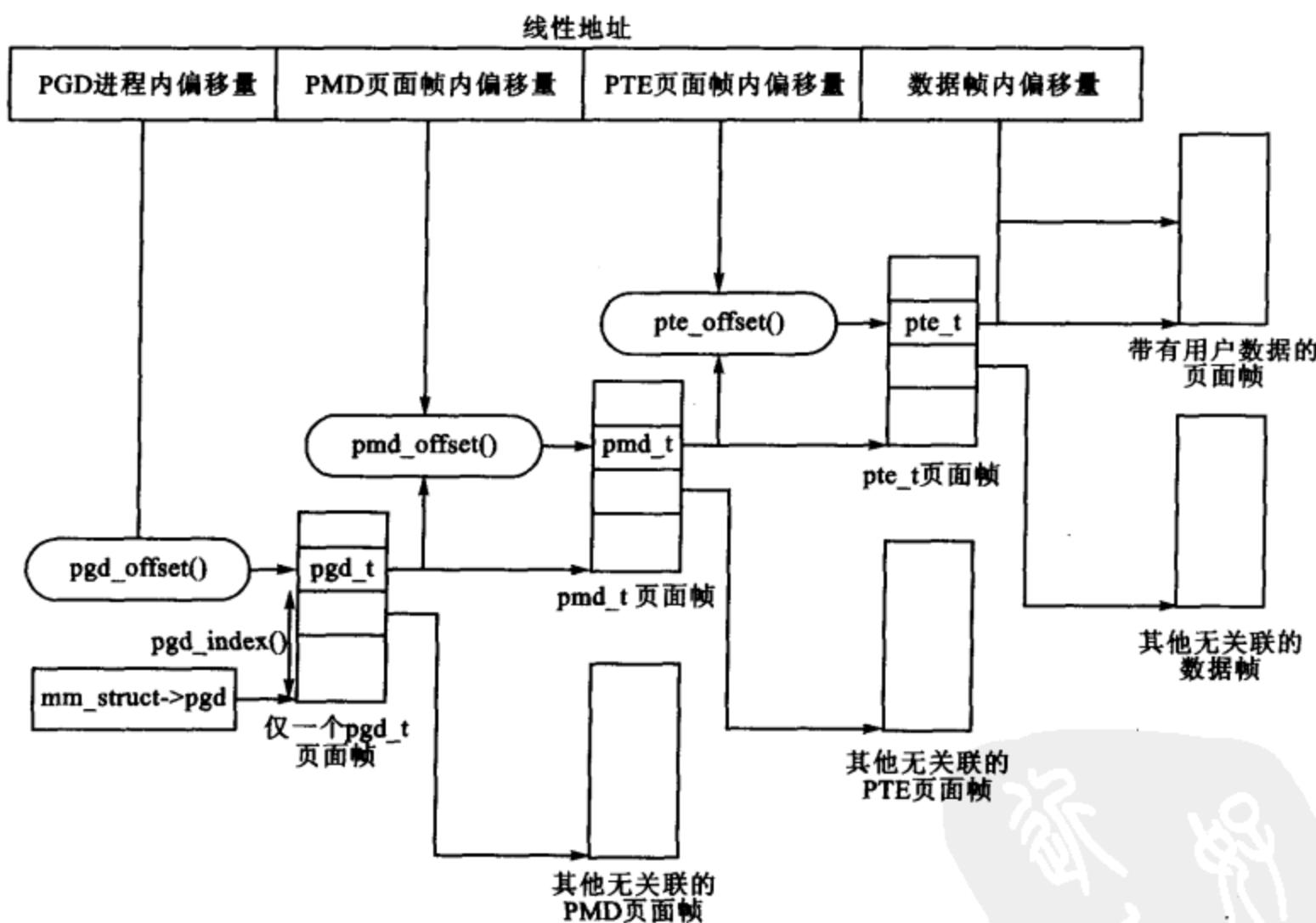


图 3.1 页表布局

为了能在这三个不同的页表层里产生不同的偏移量以及在实际的页内产生偏移量,任何一个给定的线性地址将会被划分成几个部分。为了有助于将线性地址划分成几个部分,每个页表层均提供了 3 个宏来完成此工作,它们是 SHIFT,SIZE 和 MASK 宏。SHIFT 宏主要用于指定在页面每层映射的长度,以位为单位计算,如图 3.2 所示。

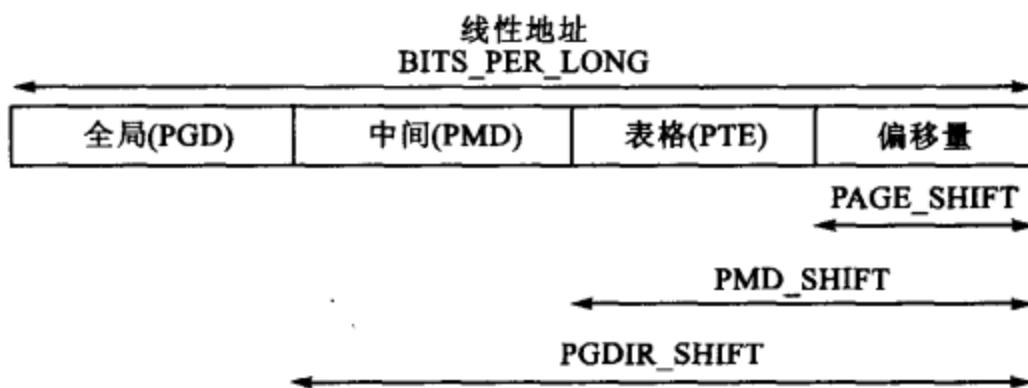


图 3.2 线性地址位大小的宏

MASK 宏用于和线性地址作“与”操作以找到所属各个页表层的位，这种功能经常被用于判断线性地址是否与某一页表层对齐。SIZE 宏用于某一页表层里的每一项所代表的空间大小(以 B 为单位)。SIZE 和 MASK 之间的关系如图 3.3 所示。

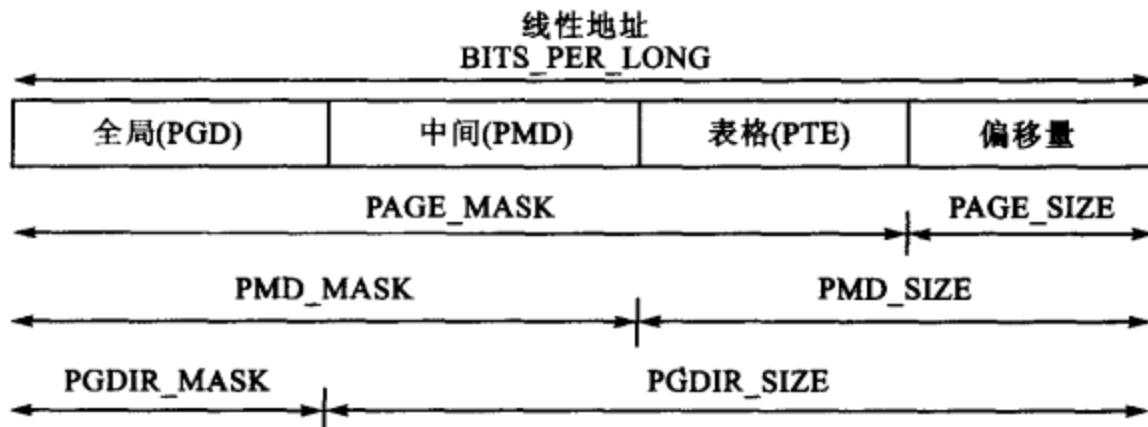


图 3.3 线性地址大小和掩码宏

在这 3 个宏中，SHIFT 是最重要的，因为其他两个都是以它为基础计算出来的。例如，在 x86 中三个宏分别定义为：

```

5 #define PAGE_SHIFT 12
6 #define PAGE_SIZE (1UL << PAGE_SHIFT)
7 #define PAGE_MASK (~(PAGE_SIZE-1))

```

PAGE_SHIFT 是在线性地址里以位为单位的偏移量长度，其值在 x86 中是 12 位。页面的大小很容易通过前述代码的等价式 2^{PAGE_SHIFT} 计算得到。最后，mask 的值也是通过 PAGE_SIZE-1 位的取反操作获得。如果一个页面需要与一个页面的大小对齐，系统将会调用 PAGE_ALIGN()。这个宏的功能是把 PAGE_SIZE-1 加到这个地址上，然后同 PAGE_MASK 进行与操作。

PMD_SHIFT 表示在线性地址中第二级页表所映射的位。PMD_SIZE 和 PMD_MASK 的计算方式也类似于页级宏。

PGDIR_SHIFT 表示在线性地址中第一级页表所在的位。PGDIR_SIZE 和 PGDIR_MASK 的计算类似于如上所述。

最后 3 个重要宏是 PTRS_PER_x, 它们决定了页表每一层所包含的项个数。PTRS_PER_PGD 是 PGD 所包含的指针数目, 在不支持 PAE 的 x86 中它的值是 1 024。PTRS_PER_PMD 是 PMD 所包含的指针数目, 在不支持 PAE 的 x86 中它的值是 1, 最后是处于最底层的 PTRS_PER_PTE, 在 x86 中它的值是 1 024。

3.2 描述页表项

如前所述, 三层页表中的每一个项 PTE, PMD, PGD 分别由 pte_t, pmd_t, pgd_t 描述。它们实际上都是无符号的整型数据, 之所以定义成结构, 是出于两个原因。第一是为了起到类型保护的作用, 以使得它们不会被滥用。第二是为了满足某些特性, 如在支持 PAE 的 x86 中, 将有额外的 4 位用于对大于 4 GB 内存的寻址。为了存储用于保护的位, 内核中使用 pgprot_t 定义了一些相关的标志位, 它们一般被放在页表项的低位。

出于类型转换的考虑, 内核在 `asm/page.h` 文件中定义了 4 个宏, 这些宏管理先前讨论的类型并返回数据结构中相关的部分。它们是 `pte_val()`, `pmd_val()`, `pgd_val()` 和 `pgprot_val()`。为了能反向转换, 内核又提供了另外 4 个宏 `_pte()`, `_pmd()`, `_pgd()` 和 `_pgprot()`。

那些保护位因结构的不同而不同。为了比较, 我们将考察不支持 PAE 的 x86 结构的这种情况下使用相同的规则。在不支持 PAE 的 x86 中, `pte_t` 只是一个 32 位的整型指针。因为每个 `pte_t` 指向一个页面帧的地址, 而所有的被指向的地址都确定是页对齐的。因此在 32 位值里面有 `PAGE_SHIFT(12)` 个位空闲出来用于描述这个页表项的状态位。表 3.1 中列出了一些保护标志位及状态标志位, 注意这些标志位是否存在以及它们所代表的含义因不同的结构而有所不同。

表 3.1 页表项保护位及状态位

| 位名 | 功能 |
|-----------------------------|------------------|
| <code>_PAGE_PRESENT</code> | 页面常驻内存, 不进行换出操作 |
| <code>_PAGE PROTNONE</code> | 页面常驻内存, 但不可访问 |
| <code>_PAGE_RW</code> | 页面可能被写入时设置该位 |
| <code>_PAGE_USER</code> | 页面可以被用户空间访问时设置该位 |
| <code>_PAGE_DIRTY</code> | 页面被写入时设置该位 |
| <code>_PAGE_ACCESSED</code> | 页面被访问时设置该位 |

除了 `_PAGE_PROTNONE` 以外这些位从字面意思就可以理解, `_PAGE_PROTNONE` 将在后面讨论。在 x86 Pentium III 或更高频的 CPU 中, 这个位被称作页属性表(PAT), 在早期的结构中如 Pentium II 保留了该位。PAT 位用于指示这个 PTE 所指向的页面的大小。在 PGD 的一个项中, 这个同样的位被叫作页面大小拓展(PSE)位, 很明显这些位是起连接作用的。

由于 Linux 没有因为用户页面而使用这个 PSE 位, 所以这个 PAT 位在 PTE 中空闲出来以用于其他的用途。事实上是可能存在这样一种情况的, 即有个页面常驻内存并且它不允许被用户空间的进程访问, 例如通过设置 `PROT_NONE` 标志位及 `mprotect()` 函数保护的内存区间。当这个区间受到保护时, `_PAGE_PRESENT` 位将被清 0 而 `_PAGE_PROTNONE` 位将被置位。内核通过宏 `pte_present()` 检查这些位是否被置位, 从而得知这个 PTE 的存在。它只是不能被用户空间所访问, 这一点很微妙但很重要。在 `_PAGE_PRESENT` 已被清 0 的情况下, 访问该页面就会发生页面错误, 因此在已知的常驻内存需要被换出时, 或者进程退出时还有页面常驻内存, Linux 可以增强这种保护措施。

3.3 页表项的使用

定义在 `<asm/pgtable.h>` 文件中的一些宏, 对于页表项的定位及检查是很重要的。为了定位一个页目录, 这里提供了 3 个宏, 用以把线性地址分成了 3 个不同的部分。`pgd_offset()` 通过对一个线性地址和 `mm_struct` 结构的操作覆盖所需地址的 PGD 项。`pmd_offset()` 通过对一个 PGD 项和一个线性地址的操作返回相应的 PMD。`pte_offset()` 通过对一个 PMD 的操作返回相应的 PTE。线性地址剩下的部分就是在此页面内的偏移量部分。这些字段之间的关系如图 3.1 所示。

第 2 组宏决定了一个页表项是否存在或者是否正在使用中。

- `pte_none()`, `pmd_none()` 和 `pgd_none()` 在相应的项不存在时返回 1。
- `pte_present()`, `pmd_present()` 和 `pgd_present()` 在相应的页表项的 `PRESENT` 位被置位时返回 1。
- `pte_clear()`, `pmd_clear()` 和 `pgd_clear()` 将相应的页表项清空。
- `pmd_bad()` 和 `pgd_bad()` 用于在页表项作为函数的参数传递时并且这个函数有可能改变这个表项的值时, 对页表项进行检查。它们是否返回 1 需要看某些体系结构是如何定义这些宏。即便是很明确地定义了这些宏, 也应确保页面项被设置为存在并被访问, 这里需要被重点检查的两个方面。

VM 中涉及页表流程的代码通常很零散, 但是读懂它们却很重要。一个页表流程代码的简单例子便是 `mm/memory.c` 文件中的 `follow_page()`。以下就是这个函数的部分摘抄。与

页表流程无关的代码被略去。

```

407 pgd_t * pgd;
408 pmd_t * pmd;
409 pte_t * ptep, pte;
410
411 pgd = pgd_offset(mm, address);
412 if (pgd_none(*pgd) || pgd_bad(*pgd))
413     goto out;
414
415 pmd = pmd_offset(pgd, address);
416 if (pmd_none(*pmd) || pmd_bad(*pmd))
417     goto out;
418
419 ptep = pte_offset(pmd, address);
420 if (!ptep)
421     goto out;
422
423 pte = *ptep;

```

使用这 3 个 offset 宏可以很容易地浏览整个页表, 它用_none()和_bad()宏检查所浏览的是否是合法页表。

第 3 组宏用于检查和设定项权限。这些权限决定了哪些用户空间的进程以及进程是否有权限对一特定页面进行存取访问。例如, 内核页表项是不能被用户空间的进程读取的。

- 项的读权限通过 pte_read()检查, 通过 pte_mkread()设置, 通过 pte_rdprotect()清除。
- 项的写权限通过 pte_write()检查, 通过 pte_mkwrite()设置, 通过 pte_wrprotect()清除。
- 执行权限通过 pte_exec()检查, 通过 pte_mkexec()设置, 通过 pte_exprotect()清除。在 x86 中执行权限是没有什么用处的, 没有对页面的执行权限设置可言, 因此这 3 个宏只是和读取宏做了同样的事情。
- 可以通过 pte_modify()修改权限值, 但几乎是不用它的。只在 mm/mprotect.c 文件里的 pte_range()函数中用到。

第四组宏用于检查和设定一个项的状态。只有两个位在 Linux 中很重要: 脏数据位和被访问位, 它们分别由 pte_dirty()和 pte_youn()宏进行检查工作。通过 pte_mkdirty()和 pte_mkyoung()完成设置工作。通过 pte_mkclean()和 pte_old()完成清除工作。

3.4 页表项的转换和设置

下面的函数和宏用于映射地址和页面到 PTE，并设置个别的项。

宏 `mk_pte()` 用于把一个 `struct page` 和一些保护位合成一个 `pte_t`，以便插入到页表当中。另一个宏 `mk_pte_phys()` 也具有类似的功能，它将一个物理页面的地址作为参数。

宏 `pte_page()` 返回一个与 PTE 项相对应的 `struct page`。`pmd_page()` 则返回包含页表项集的 `struct page`。

宏 `set_pte()` 把诸如从 `mk_pte` 返回的一个 `pte_t` 写到进程的页表里。`pte_clear()` 则是反向操作。此外还有一个函数 `ptep_get_and_clear()` 用于清空进程页表的一个项并返回 `pte_t`。不论是对 PTE 的保护还是 `struct page` 本身，一旦需要修改它们时这个工作则是很重要的。

3.5 页表的分配和释放

最后一组函数用于对页表进行分配和释放。如前所述，页表是一些包含一个项数组的物理页面。而分配和释放物理页面的工作，相对而言代价很高，这不仅体现在时间上，还体现在页面分配时是关中断的。无论在 3 级的哪一级，分配已经被删除页表的操作都是非常频繁的，所以要求这些操作尽可能地快很重要。

于是，这些物理页面被缓存在许多被称作快速队列的不同队列里。不同的结构实现它们虽都有所不同，但原理相同。例如，并不是所有的系统都会缓存 PGD，因为对它们的分配和释放只发生在进程创建和退出的时候。因为这些操作花费的代价较大，其他页面的分配就显得微不足道了。

PGD, PMD 和 PTE 有两组不同的函数分别用作分配页表和释放页表。分配的函数有 `pgd_alloc()`, `pmd_alloc()` 和 `pte_alloc()`，相对应的释放函数有 `pgd_free()`, `pmd_free()` 和 `pte_free()`。

具体而言，存在三种不同的用作缓冲的高速缓存，分别称为 `pgd_quicklist`, `pmd_quicklist` 以及 `pte_quicklist`。不同的结构实现它们的方式虽都有所不同，但都使用了一种叫做后入先出 (LIFO) 的结构。一般而言，一个页表项包含着很多指向其他包括页表或数据的页面的指针。当队列被缓存时，队列里的第一个元素将指向下一个空闲的页表。在分配时，这个队列里最后进入的元素将被弹出来，而在释放时，一个元素将被放入这个队列中作为新的队列首部。使用一个计数器对这个高速缓存中所包含的页面数量进行计数。

虽然可以选用 `get_pgd_fast()` 作为 `pgd_quicklist` 上快速分配函数的名称，但 Linux 并未

独立于体系结构对它进行显式的定义。PMD 和 PTE 的缓存分配函数明确地定义为 pmd_alloc_one_fast() 和 pte_alloc_one_fast()。

如果高速缓存中没有多余的页面,页面的分配将通过物理页面分配器(见第6章)完成。分别对应3级的函数为get_pgd_slow(),pmd_alloc_one()以及pte_alloc_one()。

显然,在这些高速缓存中可能有大量的页面存在,所以应当有一种机制来管理高速缓存的空间。每当高速缓存增大或收缩时,就通过一个计数器增大或减小来计数,并且该计数器有最大值和最小值。在两个地方中可以调用check_pgt_cache()检查这些极值。当计数器达到最大值时,系统就会释放一些高速缓存里的项,直到它重新到达最小值。在调用clear_page_tables()后,在可能有大量的页表到达时,系统就会调用check_pgt_cache(),这个函数同时也可能被系统的空闲任务所调用。

3.6 内核页表

不同的体系结构对页表机制的初始化都不尽相同。在系统启动的时候,页表机制不会初始化其自身。x86的页表初始化可分成两个部分:首先,在系统初始化的时候,为了能开启页表机制,它会建立代表着8MB空间的两张页表;其次,在后续初始化过程剩下的内存空间。

3.6.1 引导初始化

在arch/i386/kernel/head.S的汇编程序中的函数startup_32()主要用于开启页面单元。由于所有在vmlinuz中的普通内核代码都编译成以PAGE_OFFSET+1MB为起始地址,实际上系统将内核装载到以第一个1MB(0x00100000)为起始地址的物理空间中。第一个1MB的地址常在一些设备用作和BIOS进行通讯的地方自行跳过。该文件中的引导初始化代码总是把虚拟地址减去__PAGE_OFFSET,从而能获得以1M为起始地址的物理地址。所以在开启换页单元以前,必须首先建立相应的页表映射,从而将8MB的物理空间转换为虚拟地址PAGE_OFFSET。

初始化工作在编译过程中开始进行,它先静态地定义一个称为swapper_pg_dir的数组,使用链接器指示在地址0x00101000。然后分别为两个页面pg0和pg1创建页表项。如果处理器支持页面大小拓展(PSE)位,那么该位将被设置,使得调动的页面大小是4MB,而不是通常的4KB。第一组指向pg0和pg1的指针被放在能覆盖1~9MB内存空间的位置;而第二组则被放置在PAGE_OFFSET+1MB的位置。这样一旦开启换页,在上述页表及页表项指针建立后系统可以保证,在内核映象中不论是采用物理地址还是采用虚拟地址,页面之间的映射关系都是正确的。

映射建立后,系统通过对 cr0 寄存器某一位置位开启换页单元,接着通过一个跳跃指令保证指令指针(EIP 寄存器)的正确性。

3.6.2 收尾工作

用于完成页表收尾工作的对应函数是 `paging_init()`。`x86` 上该函数的调用图如图 3.4 所示。

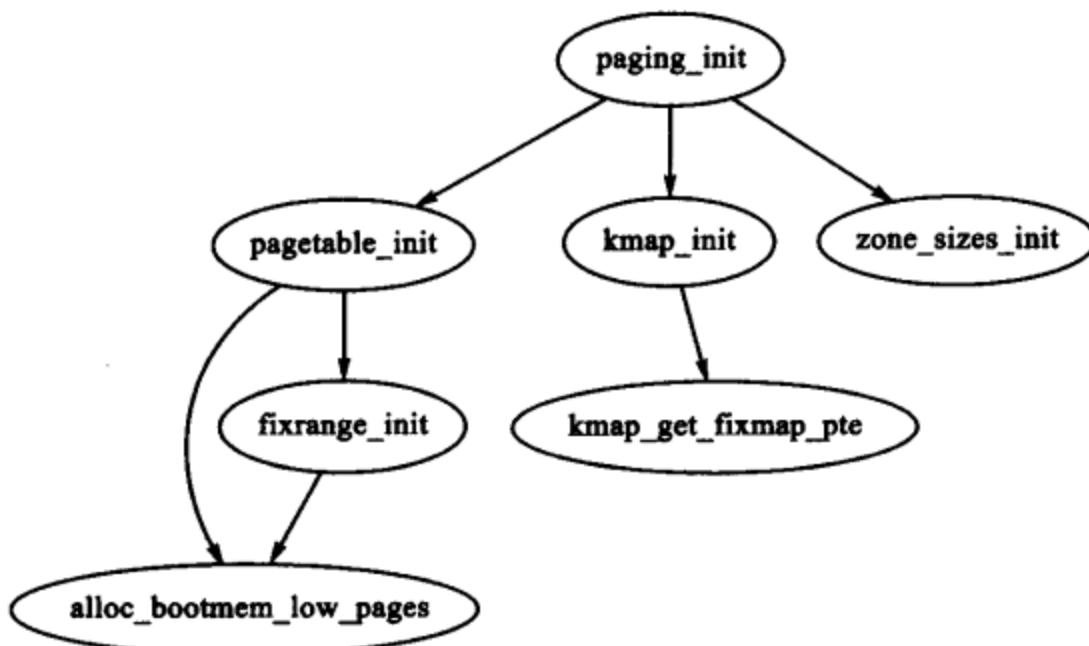


图 3.4 调用图: `paging_init()`

系统首先调用 `pagetable_init()` 初始化对应于 `ZONE_DMA` 和 `ZONE_NORMAL` 的所有物理内存所必要的页表。注意在 `ZONE_HIGHMEM` 中的高端内存不能被直接引用,对其的映射都是临时建立起来的。对于内核所使用的每一个 `pgd_t`, 系统会调用引导内存分配器(见第 5 章)来分配一个页面给 PGD。若可以使用 4 MB 的 TLB 项替换 4 KB, 则 PSE 位将被设置。如果系统不支持 PSE 位, 那么系统会为每个 `pmd_t` 分配一个针对 PTE 的页面。若 CPU 支持 PGE 标志位, 系统也会设置该标志以使得页表项是全局性的, 以对所有进程可见。

接下来, `pagetable_init()` 函数调用 `fixrange_init()` 建立固定大小地址空间, 以映射从虚拟地址空间的尾部即起始于 `FIXADDR_START` 的空间。映射用于局部高级编程中断控制器(APIC), 以及在 `FIX_KMAP_BEGIN` 和 `FIX_KMAP_END` 之间 `kmap_atomic()` 的原子映射。最后, 函数调用 `fixrange_init()` 初始化高端内存映射中 `kmap()` 函数所需要的页表项。

在 `pagetable_init()` 函数返回后, 内核空间的页表则完成了初始化, 此时静态 PGD(`swapper_pg_dir`)被载入 `CR3` 寄存器中, 换页单元可以使用静态表。

`paging_init()`接下来的工作是调用 `kmap_init()` 初始化带有 `PAGE_KERNEL` 标志位的每个 PTE。最终的工作是调用 `zone_sizes_init()`, 用于初始化所有被使用的管理区结构。

3.7 地址和 struct page 之间的映射

对 Linux 而言,必须有一种快速的方法把虚拟地址映射到物理地址或把 struct page 映射到它们的物理地址。Linux 为实现该机制,在虚拟和物理内存使用了一个 mem_map 的全局数组,因为这个数组包含着指向系统物理内存所有 struct page 的指针。所有的结构都采用了非常相似的机制,为了表述简单,在这里我们只详细讨论 x86 结构中的情况。本节我们首先讨论物理地址如何被映射到内核虚拟地址,以及又是如何利用 mem_map 数组的。

3.7.1 物理和虚拟内核地址之间的映射

正如在 3.6 节看到的,在 x86 中 Linux 把从 0 开始的物理地址直接映射成从 PAGE_OFFSET 即 3 GB 开始的虚拟地址。这意味着在 x86 上,可以简单地将任意一个虚拟地址减去 PAGE_OFFSET 而获得其物理地址,就像函数 virt_to_phys() 和宏 __pa() 所做的那样:

```
/* from <asm-i386/page.h> */
132 #define __pa(x) ((unsigned long)(x)-PAGE_OFFSET)
/* from <asm-i386/io.h> */
76 static inline unsigned long virt_to_phys(volatile void * address)
77 {
78     return __pa(address);
79 }
```

很明显,逆操作只需简单加上 PAGE_OFFSET 即可,它通过 phys_to_virt() 和宏 __va() 完成。接下来我们将看到内核如何利用这些功能将 struct pages 映射成物理地址。

有一个例外,就是 virt_to_phys() 不能用于将虚拟地址转换成物理地址。尤其是在 PPC 和 ARM 的结构中,virt_to_phys 不能转换由 consistent_alloc() 函数返回的地址。在 PPC 和 ARM 结构中,使用 consistent_alloc() 函数从无缓冲的 DMA 返回内存。

3.7.2 struct page 和物理地址间的映射

正如在 3.6.1 所看到的一样,系统将内核映象装载到 1 MB 物理地址起始位置,当然,这个物理地址就是虚拟地址 PAGE_OFFSET + 0x00100000。此外,物理内存为内核映象预留了 8 MB 的虚拟空间,这个空间可以被 2 个 PGD 所访问到。这好像意味着第一个有效的内存空间应在 0xC0800000 开始的地方,但事实并非如此。Linux 还为 ZONE_DMA 预留了 16 MB

的内存空间,所以真正能被内核分配使用的内存起始位置应在 0xC1000000,这个位置即为全局量 `mem_map` 所在的位置。虽然 Linux 还使用 `ZONE_DMA`,但是只会在非常有必要的情况下使用。

通过把物理地址作为 `mem_map` 里的一个下标,从而将其转换成对应的 `struct pages`。通过把物理地址位右移 `PAGE_SHIFT` 位,从而将右移后的物理地址作为从物理地址 0 开始的页面帧号(PFN),它同样也是 `mem_map` 数组的一个下标。正如宏 `virt_to_page()` 所做的那样,其声明在 `<asm-i386/pae.h>` 中如下:

```
# define virt_to_page(kaddr) (mem_map + (_pa(kaddr) >> PAGE_SHIFT))
```

宏 `virt_to_page()` 通过 `_pa()` 把虚拟地址 `kaddr` 转换成物理地址,然后再通过右移 `PAGE_SHIFT` 位转换成 `mem_map` 数组的一个下标,接着通过简单的加法操作就可以在 `mem_map` 中查找它们。Linux 中不存在将页面转换成物理地址的宏,但你应该知道如何计算。

3.8 转换后援缓冲区(TLB)

最初,当处理器需要映射一个虚拟地址到一个物理地址时,需要遍历整个页面目录以搜索相关的 PTE。通常这表现为每个引用内存的汇编指令实际上需要多个页表截断的相分离的内存引用[Tan01]。为了避免这种情况的过度出现,许多体系结构中都利用了这样一个事实,就是大多数的进程都是采用局部引用,或者,换句话说,少量的页面却使用了大量的内存引用。它们提供一个转换后援缓冲区(TLB)来利用这种引用的局部性原理,这个高速缓存是一个联合内存,用来缓存虚拟到物理页表转换的中间结果。

Linux 假设大多数的体系结构都是支持 TLB 的,即便独立于体系结构的代码并不关心它如何工作。相反,与体系结构相关的钩子都分散在 VM 的代码中,大家知道,一些带有 TLB 的硬件是需要提供一个 TLB 相关的操作的。例如,在页表更新后,诸如在一个页面错误发生时,处理器可能需要更新 TLB 以满足虚拟地址的映射要求。

不是所有的体系结构都需要这种类型的操作,但是,因为有些体系结构是需要的,所以 Linux 中就需要存在钩子。如果某个体系结构并不需要诸如此类的操作,那么在这个体系结构中完成 TLB 操作的函数就是一个空函数,这在编译时就进行过优化。

大部分关于 TLB 的 API 钩子列表都在 `<asm/pgtable.h>` 中声明,如表 3.2 和表 3.3 所列,在内核源码的 `Documentation/cachetlb.txt`[Mil00]文件已写明了这些 API。在某些情况下可能仅只有一个 TLB 刷新操作,但由于 TLB 刷新操作和 TLB 填充工作都是开销非常大的操作,所以应尽可能避免不必要的 TLB 刷新操作。例如,切换上下文时,Linux 会使用延迟 TLB 刷新以避免载入新的页表,这将在 4.3 节作进一步的讨论。

表 3.2 转换后缓缓冲区刷新 API(cont.)

```
void flush_tlb_all(void)
```

该函数刷新系统中所有处理器上的所有 TLB, 所以它是开销最大的 TLB 刷新操作。在它完成以后, 对应页表的变更在全局都可见。修改全局的内核页表后, 如完成 vfree() (见第 7 章) 后或者刷新 PKMap 以后 (见第 9 章), 有必要调用该函数操作

```
void flush_tlb_mm(struct mm_struct * mm)
```

该函数刷新与用户空间部分 (例如, 低于 PAGE_OFFSET) 所需要的 mm 上下文相关的所有 TLB 项。在某些体系结构中, 诸如 MIPS, 这个操作需要在所有的处理器上进行, 但是通常它都限制在局部的处理器上。该函数只有在某个操作会影响到整个地址空间时才被调用, 如调用 dup_mmap() 复制所有的地址映射完成后, 或者调用函数 exit_mmap() 删除所有的内存映射后

```
void flush_tlb_range(struct mm_struct * mm, unsigned long start, unsigned long end)
```

正如函数名所表示的, 该函数刷新所有在用户空间范围内对应 mm 上下文的项。它在移动或变更新区域时使用, 其中 mremap() 用于移动区域, mprotect() 用于变更权限。在 munmap() 调用 tlb_finish_mmu() 解除映射时也间接地调用这个函数, 其中 tlb_finish_mmu() 巧妙地调用了 flush_tlb_range() 函数。系统中所提供的 API 函数能够快速移除 TLB 项的范围, 而不是重复性地调用 flush_tlb_page() 函数

3.9 一级 CPU 高速缓存管理

由于 Linux 管理 CPU 高速缓存的策略与 TLB 非常近似, 因此本小节将讲述 Linux 如何使用和管理 CPU 高速缓存。CPU 高速缓存, 和 TLB 高速缓存一样, 也利用了程序中常见的隐藏局部性引用 [Sea00][CS98]。为了避免因每次引用而要从主存中取得数据, CPU 采用了缓存一小部分数据在 CPU 高速缓存中的方式。一般而言, 存在一二级两级 CPU 高速缓存。二级高速缓存要大一些, 但与一级高速缓存相比速度上慢一些, 在这里我们只考虑 Linux 的一级高速缓存, 即 L1 高速缓存。

CPU 的高速缓存是线性排列的。其中的每行的空间都很小, 一般为 32 B, 各行都与其边界对齐。或者说, 一个 32 B 的高速缓存行与 32 B 的地址对齐。在 Linux 中, 行大小为 L1_CACHE_BYTES, 这是按每个结构的不同来定义的。

虽然地址映射高速缓存行的方式随体系结构不同而可能不同, 但映射一般都是如下三个操作中的一种: 直接映射、关联映射和关联集映射。直接映射是最简单的方式, 每个内存块只与惟一一个可能的高速缓存行相映射。关联映射中, 任意的内存块可以与任意的高速缓存行相对应。关联集映射是一种混合的方式, 任意的内存块可以与任意的高速缓存行相映射, 但只能在一个可用行的子集里面映射。如果不考虑这些映射模式, 它们都有一个相同的地方, 即相

邻的地址与和高速缓存大小相对齐的地址一般会使用不同的行。因此 Linux 采用了如下简单技巧来尝试加大使用高速缓存的力度：

- 频繁访问的结构字段放在结构的头部,以此增加当只需要一个行来访问一般字段时的高速缓存使用率;
- 一个结构中不相关的项的大小尽量以高速缓存大小为准,这样可以避免在共享 CPU 之间出现错误;
- 在普通高速缓存中的对象,比如 mm_struct 高速缓存,一般与 CPU 的 L1 高速缓存对齐,以此避免共享中的错误。

如果 CPU 引用的地址不在高速缓存中,则发生了一次高速缓存未命中,然后 Linux 会将数据从主存中取到高速缓存中。高速缓存未命中时的代价是非常高的,因为访问一次高速缓存一般少于 10 ns,而访问主存一次的时间开销一般在 100 ns 到 200 ns 之间。所以 Linux 基本的目标就是要尽可能多地高速缓存命中以及尽可能少出现高速缓存未命中的情况。

表 3.3 转换后援缓冲区刷新 API

| |
|---|
| void flush_tlb_page(struct vm_area_struct *vma, unsigned long addr) |
|---|

可以断定,这个 API 用于从 TLB 中刷新单个页面。最普遍的两个作用是在发生缺页中断时或换出错误页面时刷新 TLB

| |
|--|
| void flush_tlb_ptables(struct mm_struct *mm, unsigned long start, unsigned long end) |
|--|

该 API 在页表销毁并释放时被调用。某些平台的高速缓存会保存最低级别的页表,如当前的页面帧存储项应在删除页面后刷新。该函数在解除一个区域的映射以及回收页面目录项时调用

| |
|--|
| void update_mmu_cache(struct vm_area_struct *vma, unsigned long addr, pte_t pte) |
|--|

该 API 只在页面错误处理完成时才被调用。在 pte 中存在一个新的针对虚拟地址的转换与体系结构相关的代码。每一个体系结构会根据这些信息作出不同的使用决定。例如, Sparc64 就使用该信息来决定是否需要刷新本地 CPU 的数据高速缓存,或是否需要发送一个内部处理器中断信号(IPI)给远程处理器

由于一部分体系结构并不会自动地管理它们自己的 TLB,而另有一部分体系结构并不自动管理它们自己的 CPU 高速缓存。所有的钩子函数都分布在虚拟到物理映射变更的地方,例如在页表更新时。应当首先进行 CPU 高速缓存的刷新,因为虚拟地址在某些 CPU 的高速缓存中刷新时需要存在从虚拟到物理的映射。这 3 个操作有固定的调用顺序,如表 3.4 中所列。

表 3.4 高速缓存和 TLB 刷新顺序

| 刷新所有的 MM | 刷新范围 | 刷新页面 |
|------------------|---------------------|--------------------|
| flush_cache_mm() | flush_cache_range() | flush_cache_page() |
| 变更所有的页表 | 变更页表的范围 | 变更单个的 PTE |
| flush_tlb_mm() | flush_tlb_range() | flush_tlb_page() |

用于刷新高速缓存的 API 在<asm/pgtable.h>文件中声明,如表 3.5 所列。在某种意义上,它和 TLB 刷新 API 很相似。

表 3.5 CPU 高速缓存刷新 API

| | |
|---|---|
| void flush_cache_all(void) | 该函数刷新整个 CPU 高速缓存系统,是刷新中最为剧烈的操作。该函数在内核页表发生变化时使用,它在系统中是全局的 |
| void flush_cache_mm(struct mm_struct mm) | 该函数刷新所有和地址空间相关联的项。完成后,将不再有高速缓存行和 mm 相关联 |
| void flush_cache_range(struct mm_struct * mm, unsigned long start, unsigned long end) | 该函数刷新和地址空间中与某个范围内的地址相关联的高速缓存行。与其 TLB 类似,也在体系结构具有更有效的刷新范围的方式时提供该方式,而不是刷新每个单独的页面 |
| void flush_cache_page(struct vm_area_struct * vma, unsigned long vmaddr) | 该函数刷新一个单页面大小的区域。因为通过 vma->vm_mm 更容易访问 mm_struct,所以这里提供了 VMA 参数。另外,通过测试 VM_EXEC 标志位,体系结构将可以知道该区域是否可以执行区分指令高速缓存和数据高速缓存的操作。VMA 会在第 4 章有更多的介绍 |

然而到这里还没有结束。还需要有第 2 个接口的集合来避免虚拟混淆现象的问题。之所以会出现这样的问题是因为某些 CPU 基于虚拟地址来选择高速缓存行,这就意味着同一个物理地址可以存在于多个高速缓存行上,从而导致了高速缓存的一致性问题。拥有这个问题的体系结构需要尝试和保证共享的映射仅只使用地址作为一个 stop-gap 方法。即使如此,Linux 还是提供了适当解决该问题的 API,如表 3.6 所列。

表 3.6 CPU D-Cache 和 I-Cache 刷新 API

| | |
|---|--|
| void flush_page_to_ram(unsigned long address) | 这是一个不值得推荐的 API,应当不再被使用了,事实上,它将在 2.6 中被完全删除掉。在这里之所以要谈及它是为了完整性,因为目前的版本还在使用它。该函数在一个新的物理页面替换进程的地址空间时调用。需要避免的是映射发生后,页面从内核空间不可见地写入到用户空间中 |
| void flush_dcache_page(struct page * page) | 该函数在内核写入一个页面高速缓存或者从一个页面高速缓存复制时调用,因为它们可能被多个进程所映射 |
| void flush_icache_range(unsigned long address, unsigned long endaddr) | 该函数在内核在可能被执行的地址中存储信息时调用,比如载入一个内核模块时 |

续表 3.6

```
void flush_icache_user_range(struct vm_area_struct *vma, struct page *page, unsigned long addr, int len)
除了它是在一个用户空间的范围受影响时才被调用外,该函数与 flush_icache_range()类似。目前,它只在访问进程 vm()访问地址空间时,用于 ptrace()(调试时使用)
```

```
void flush_icache_page(struct vm_area_struct *vma, struct page *page)
该函数在映射一个对应于高速缓存的页面调用。它是与体系结构相关的,使用 VMA 标志位决定是否应该刷新 I-Cache 或者 D-Cache
```

3.10 2.6 中有哪些新特性

大多数页面管理的机制在 2.6 当中本质上都是一样的,但是需要提到的变更了的地方范围还是非常广泛的,而对其实现的介绍也很深入。

MMU-less 体系结构的支持

在这里需要提到一个新的文件,称为 mm/nommu.c。这个源文件包括替换函数的代码,用于假设存在 MMU 体系结构,例如 mmap() 代码。这是为了支持体系结构,因为通常微控制器并不具有 MMU。这一领域的大量工作主要由 uCLinux 工程(www.uclinux.org)开发。

反向映射

页表管理中最为显著也是最为重要的变更就是引入了反向映射(rmap)。称其为“rmap”也是经过深思熟虑了的,因为这里采用了普遍适用的取首字母的方式,它也不会混淆 Rik 先驱 Riel 所开发的-rmap 树,-rmap 树对于 VM 的主干有很大的改造,而不仅仅是反向映射。

用一句话来概括,rmap 是在映射一个只提供了 struct page 的独特页面时具有了定位所有 PTE 的能力。在 2.4 中,发现所有 PTE 的唯一方法就是映射一个共享页面,比如一个内存映射的共享库,然后线性地搜索属于所有进程的所有页表。这样做的开销很大,而 Linux 尝试使用交换高速缓存来避免这个问题(见 11.4 节)。这就意味着,由于大量共享页面的存在,Linux 可能需要换出整个进程而不考虑页面的生命期和使用模式。在 2.6 中取而代之的是为每一个 struct page 关联了一个 PTE 链,这样就可以在所有的引用某个页面的页表中来回移动这个页面。就这样,系统可以使用更聪明的方式换出 LRU 中的页面而不需要采取交换整个进程的方式。

读者可能可以想像得到,实现这样一个简单概念其实是很棘手的事情。理解该实现的第一步是了解 union pte 已经作为了 struct page 的一个字段。这个联合包括两个字段,一个是指向 struct pte_chain 的称为 chain 的指针,而另一个是 pte_addr_t 类型的称为 direct 的字段。

该联合在仅有一个 PTE 映射项时用 `direct` 保存内存而进行了优化。否则,就使用一个链。`pte_addr_t` 类型随体系结构而变化,但是,不管它是什么类型,它都可以用于定位一个 PTE,因此我们可以简单地像 `pte_t` 一样看待它。

而 `struct pte_chain` 就有一些复杂了。该结构本身很简单,但是它和其他很多的字段相关,Linux 为了使其精小而高效做了很多的工作。幸运的是,这并不会导致其难以理解。

首先,slab 分配器负责分配和管理 `struct pte_chains`,因为这种类型的工作是 slab 分配器最擅长处理的。每一个 `struct pte_chain` 都可以通过 NRPTE 指针访问 PTE 结构。在这之后 Linux 将会填充多个 PTE,接着将会分配和添加一个 `struct pte_chain` 到链当中。

`struct pte_chain` 具有两个字段。第 1 个字段是 `unsigned long next_and_idx`,它有两个用法。在 `next_and_idx` 和 NRPTE 进行与操作后,它将返回 `struct pte_chain` 中目前 PTE 的数量并指出下一个空槽的位置。当 `next_and_idx` 与 NRPTE 的反(即 \sim NRPTE)进行与操作后,它将返回一个指向该链中下一个 `struct pte_chain` 的指针。这基本上实现了 PTE 链的工作。

为了让你感受一下 `rmap` 的复杂程度,我将给出一个例子来描述在需要映射一个新的 PTE 到一个页面时都发生了些什么。基本的流程是系统让调用者通过 `pte_chain_alloc()` 分配一个新的 `pte_chain`。该分配的链和 `struct page` 及 PTE 作为 3 个参数传递给 `page_add_rmap()`。如果与页面相关联的 PTE 链有槽可用,那么就使用该链,然后返回由调用者分配的 `pte_chain`。如果没有槽可用,那么将分配的 `pte_chain` 添加到 PTE 链中,并返回 `NULL`。

`rmap` 完成工作所用到的 API 非常多,比如创建链,在链上添加和移除 PTE 等,要列出其全部的 API 超出了本节的范围。幸运的是,这些 API 都定义在文件 `mm/rmap.c` 中,而且函数的注释非常详细,它们的用法也非常明晰。

介绍反向映射主要有两个好处,它们都和页面换出相关。第 1 个好处体现为建立和销毁页表上。正如将在 11.4 节中所看到的,页面换出时都处在一个交换高速缓存中,而信息都写入到 PTE 中以便再次找到该页面。这样操作可能会导致很多的小错误发生,因为页面被放入交换高速缓存然后会被某个进程再次错误处理。而通过 `rmap`,建立和移除 PTE 将具有原子性。第 2 个主要的好处是在页面需要换出时,找到引用页面的所有的 PTE 是很容易的,但这在 2.4 中却很不实际,因此在 2.4 中使用了交换高速缓存。

即便如此,反向映射也是有些开销的。首先,最为明显的就是 PTE 链需要额外的空间。很显然,其次,是反向映射需要 CPU 开销,但是还没有什么迹象表明这种开销很大。还有很重要的一点,就是反向映射只在页面换出很频繁时才有很大的好处。如果机器的负载不会导致换出大量的页面或者内存足够大时, `rmap` 的优点和缺点就只是在讨论层面上的。

基于对象的反向映射

为每个页面都进行反向映射可能需要相当大的空间开销。为了解决这样的问题,在

VMA 中许多反向映射的页面实际上都是相同的。其中的一个方法就是在 VMA 上反向映射,而不是针对单个的页面反向映射。即不采用为每个页面反向映射的方式,而是对某一个特定页面的所有 VMA 中来回映射,并解除为每个页面所作的映射。请注意这里针对 VMA 而提到的对象,不是面向对象领域的对象概念。在写这本书的时候,这种特征还没有合并进去,见到它是在内核 2.5.68-mm1 中,但是只要存在问题,就会具备非常强烈的动机来解决它。非常有趣的是,只针对 file/device 后援 objrmap 已经发布有补丁可用,但这只针对那些对这个非常感兴趣的读者。

让所有的 PTE 实现来回映射一个页面有两步工作要做。第 1 步工作就是调用 `page_referenced()` 函数检查映射于某个页面的所有的 PTE,以确定该页当前是否被引用。第 2 步工作就在一个页面需要从所有的进程中解除映射时调用 `try_to_unmap()`。后面的内容更为复杂,因为系统必须有两种类型的映射反向,它们是有文件或设备后援的映射以及那些匿名映射。在这两种情况下,Linux 基本的目标就是遍历映射某个特定页面的 VMA,以及遍历 VMA 的页表从而取得 PTE。惟一的差异在于它的实现方式。在某些文件后援时最为简单,因此本书将先阐述这种情况。先讨论 `page_referenced()` 如何实现。

`page_referenced()` 函数调用 `page_referenced_obj()`,后者是查找映射于指定页面的 VMA 上的所有 PTE 的函数中最上层的函数。在该页面由文件或设备映射后,`page->mapping` 包括了一个指向有效 `address_space` 的指针。其中 `address_space` 中包括两个链接列表用于映射 `address_space->i_mmap` 和 `address_space->i_mmap_shared` 字段的所有 VMA。对于该链接列表上的每一个 VMA,系统将 VMA 和页面作为参数传递给 `page_referenced_obj_one()`。函数 `page_referenced_obj_one()` 首先检查该页面是否在由该 VMA 管理的地址中,如果是,则使用 `VMA(vma->vm_mm)` 遍历页表中的 `mm_struct`,直到找到映射到该页面 `mm_struct` 的 PTE 为止。

对匿名页面的跟踪要复杂一些,它的实现分为多个步骤。这部分只出现了很少一段时间,而且已经在 2.5.65-mm4 中被移除掉,因为它和其他的变更相冲突。实现它的第一步就是使用 `page->mapping` 和 `page->index` 字段跟踪成对的 `mm_struct` 和 `address`。这些字段在以前用于存储指向 `swapper_space` 的指针和指向 `swp_entry_t` 的指针(见第 11 章)。至于如何准确定位它,已经超出了本节的讨论范围,简单而言,就是将 `swp_entry_t` 存储到 `page->private` 中。

`try_to_unmap_obj()` 的操作方式类似。虽然,在不需要为单个的页面进行反向映射时使用这种方法也可以引用同一个页面的所有 PTE,但一个严重的搜索复杂度问题却使合并不可能。具体的问题描述见下文。

考虑 100 个进程映射了 100 个 VMA 到一个文件的情况。在这种情况下通过基于对象的反向映射来解除单个的页面的映射,可能需要搜索 10 000 个 VMA,而搜索中的大部分工作都是不必要的。而基于页面的反向映射,只需要检查 100 个 `pte_chain` 槽,每一个槽对应一个进程。优化工作就是在 `address_space` 中以虚拟地址对 VMA 进行排序,但是搜索单个的页面在

向基于对象的反向映射合并时依旧开销很大。

高端内存中的 PTE

在 2.4 里,页表项在 ZONE_NORMAL 中是因为内核在遍历页表时需要直接定位它们。初看起来是可以接受的,但是后来却出现了下面这种情况:采用高端内存的机器上,ZONE_NORMAL 已经被第三级页表的 PTE 使用。解决这个问题的方法很明显,就是需要将 PTE 移动到高端内存中,在 2.6 中就是这么实现的。

正如我们在第 9 章里看到的一样,计算在高端内存中的地址信息是很不方便的,所以移动 PTE 到高端内存中是一个很耗编译时间的配置选项。简单而言,可以使用低地址空间而又同时用于映射的槽的数量非常有限,所以内核必须从高端内存映射页面到低地址空间中去,这里的瓶颈也正是问题的所在。尽管如此,为了对大量的 PTE 进行处理,Linux 在这里还没有其他的可选方案。在写本书的时候有一个提议,即建立一个用户内核虚拟区域(UKVA),它可以成为每个进程在内核中的私有空间,目前还不清楚这个方案是否会被合并到 2.6 中。

考虑到高端内存映射,2.4 中的宏 pte_offset()被 2.6 中的宏 pte_offset_map()代替。如果 PTE 在低端内存中,则 pte_offset_map()的行为和 pte_offset()一样,返回 PTE 的地址。如果 PTE 在高端内存中,则首先使用 kmap_atomic()映射 PTE 到低端内存中,这时它是可以被内核所使用的。其中必须用 pte_unmap()尽可能快地解除对 PTE 的映射。

从编程的角度来看,这就意味着遍历页表的代码看起来不一样。尤其是从给定的地址找到 PTE 的代码,现在这些代码如下所示(从 mm/memory.c 文件得到):

```

640 ptep = pte_offset_map(pmd, address);
641 if (!ptep)
642     goto out;
643
644 pte = *ptep;
645 pte_unmap(ptep);

```

另外,PTE 的分配 API 有了变更。为了取代 pte_alloc(),目前使用 pte_alloc_kernel()进行内核 PTE 映射,使用 pte_alloc_map()进行用户空间的映射。它们之间主要的差异就是 pte_alloc_kernel()不会因为 PTE 而使用高端内存。

在内存管理方面,从高端内存映射过量的 PTE 是不容忽略的情况。在某一个时刻只允许有一个 PTE 可以映射单个 CPU,即便是 pte_offset_map_nested()只需要一秒钟的时间。这说明了在检查所有的 PTE 时需要额外的开销,如在 zap_page_range()过程中,在给定了所有的 PTE 范围时就需要解除映射。

在写这本书的时候,有一个补丁已经被提交以替换高端内存中的 PMD,本质上它使用了同样的机制和 API 变更。它很有可能会被合并到新版本中。

大型 TLB 文件系统

大多数流行的体系结构都支持大于一个页面的大小。例如,在许多 x86 体系结构上,有这样一个选项来选择是使用 4 KB 的页面还是使用 4 MB 的页面。一般而言,Linux 只使用大型的页面来映射实际的内核映象或映射任何位置。由于 TLB 槽是很稀有的资源,所以 Linux 尽可能地利用了大页面的优点,尤其是在机器有大量内存的情况下。

在 2.6 中,Linux 允许进程使用大型页面,其大小由 HPAGE_SIZE 决定。而大型页面的可用数量由系统管理员通过使用/proc/sys/vm/nr_hugepages proc 接口设定,它最终使用了函数 set_hugetlb_mem_size()。由于是否成功分配取决于可用的在物理上邻近的内存,所以应该在系统启动时就进行分配工作。

2.6 中实现的最后部分是一个大型 TLB 文件系统(hugetlbfs),它是在文件 fs/hugetlbfs/inode.c 中的一个伪文件系统。一般地,该文件系统中的每一个文件都有一个后援的大型页面。在初始化过程中,init_hugetlbfs_fs()注册该文件系统,并通过 kern_mount()把它作为一个内部的文件系统挂载到系统中。

有两种方法可以实现某个进程访问大型页面。第 1 种方法就是使用 shmget()建立一个具有大型页面作后援的共享区域,而第 2 种方法在这个大型页面的文件系统中打开一个文件时调用 mmap()。

当一个共享的内存区域需要由大型页面作为其后援时,进程就应当调用 shmget()并传递 SHM_HUGETLB 作为一个标志位。它的结果在 hugetlb_zero_setup()中被调用,它在内部 hugetlbfs 的根部创建一个新的文件,即在内部文件系统的根部创建一个文件。而该文件的命名取决于一个称为 hugetlbfs_counter 的原子计数器,它在每当有一个共享区域被建立时就增一。

为创建一个由大型页面作后援的文件,一个 hugetlbfs 类型的文件系统必须首先由系统管理员挂载到系统当中。完成该工作的指令细节在文件 documentation/vm/hugetlbpage.txt 中。挂载该文件系统后,就可以通过系统调用 open()来像对待一般文件那样进行创建文件。在打开文件时 mmap()被调用,file_operations 结构 hugetlbfs_file_operations 会保证在正确建立区域时调用 hugetlbfs_file_mmap()。

大型 TLB 页面由其自身函数来完成页表的管理、地址空间的操作和文件系统的操作。页表管理的函数的命名都可以在文件<linux/hugetlb.h>中找到,而它们都和它们普通页面的命名非常相似。hugetlbfs 函数的实现都分布在它们普通页面的附近,这样便于查找。

高速缓存刷新管理

2.6 中所作的变更很小。只是移除了 API 函数 flush_page_to_ram(),而引入了一个新的 API 函数 flush_dcache_range()。

第 4 章

进程地址空间

虚拟内存的一个主要好处就是可以让每个进程都有属于自己的虚拟地址空间,这种空间可以通过操作系统映射到物理内存。本章我们将讨论进程地址空间以及 Linux 如何对其进行管理。

内核在处理用户地址空间和内核空间上有着很大的不同。例如,内核空间的分配不管此时 CPU 上运行何种进程,都可以很快得到满足,它对整个系统来说是全局性的。但 `vmalloc()` 除外,因为一个微小的错误都会使进程页表和引用页表发生一次同步操作,但一旦请求页面,系统依旧可以很快地分配页面。对于进程,它通过一个页表项指针指向一个只读的全局全零页面,以实现在进程的线性地址空间中保留空间。一旦进程对该页面进行写操作,就会发生缺页中断,这时系统会分配一个新的全零页面,并由一个页表项指定,且标记为可写。新的全零页面看起来和原来的全局全零页面完全一样。

用户空间并非一成不变。每一次上下文切换后,除了将在 4.3 节提到的 TLB 切换以外,用户空间的内容都可能发生变化。也正因为如此,内核必须能够捕获用户空间中的异常并定位用户空间中的错误。这将在 5.5 节中讨论。

本章我们首先讨论线性地址空间的构成及其各个部分的用途。然后讨论一些与进程相关的结构,以及它们的配置、初始化和释放。接着将介绍如何创建进程地址空间中的私有区域以及与之相关的函数。同时涉及与进程地址空间相关的异常处理、缺页中断等。最后我们将介绍内核与用户空间之间如何相互正确地拷贝数据。

4.1 线性地址空间

从用户的角度来看,地址空间是一个平坦的线性地址空间,但从内核的角度来看却大不一样。地址空间分为两个部分:一个是随上下文切换而改变的用户空间部分,一个是保持不变的内核空间部分。两者的分界点由 PAGE_OFFSET 决定,在 x86 中它的值是 0xC0000000。这意味着有 3 GB 的空间可供用户使用,与此同时,内核可以映射剩余的 1 GB 空间。内核角度的线性虚拟地址空间如图 4.1 所示。

系统为了载入内核映象,需要保留从 PAGE_OFFSET 开始的 8 MB(两个 PGD 定位的内存大小)空间,这 8 MB 只是为载入内核而保留的合适空间。如 3.6.1 小节所述,内核映象在内核页表初始化时被放置到此保留的 8 MB 空间内,紧随其后的是供 UMA 体系结构使用的 mem_map 数组,这已经在第 2 章中讨论过。该数组通常位于标记为 16 MB 的位置,但为避免用到 ZONE_DMA,也不是经常这样。对于 NUMA 体系结构,虚拟 mem_map 各部分分散在该区域内,各部分所在具体位置由不同的体系结构决定。

从 PAGE_OFFSET 到 VMALLOC_START - VMALLOC_OFFSET 是物理内存映射的部分。这个区域的大小由可用 RAM 的大小决定。正如我们将在 4.6 节所看到的,它通过页表项把物理内存映射到 PAGE_OFFSET 开始的虚拟地址。为防止边界错误,在物理内存映射和 vmalloc 地址空间之间存在一个大小为 VMALLOC_OFFSET 的空隙。在 x86 上,这个空间大小为 8 MB。例如,在一个 RAM 大小为 32 MB 的 x86 系统上,VMALLOC_START 等于 PAGE_OFFSET + 0x02000000 + 0x00800000。

在小内存的系统中,为了能使 vmalloc() 在一个连续的虚拟地址空间里表示一个非连续的内存分配情况,余下的虚拟地址空间减去 2 个页面空隙的大小将全部用于 vmalloc()。而在大内存系统中,vmalloc 的区域则扩大到 PKMAP_BASE 减去 2 个页面空隙的大小,此外还引入 2 个区域。第 1 个是从 PKMAP_BASE 开始的区域,这部分保留给 kmap() 使用,而 kmap() 的作用是把高端内存页面映射到低端内存,如第 9 章所述。第 2 个区域是从 FIXADDR_START 至 FIXADDR_TOP 的固定虚拟地址映射区域,这个区域供在编译时需要知道虚拟地址的子系统使用,例如高级可编程的中断控制器(APIC)。FIXADDR_TOP 在 x86 中静态地定义为 0xFFFFE000,这个位置在虚拟地址空间结束的前一页上。固定映射区域的大小通过在编译时的 __FIXADDR_SIZE 变量计算,再从 FIXADDR_TOP 向后索引 __FIXADDR_SIZE 大小,从而标识 FIXADDR_START 区域的起始地址。

vmalloc(),kmap() 以及固定映射区域所需的区域大小限制了 ZONE_NORMAL 的大小。由于运行中的内核需要这些函数,所以在地址空间的顶端至少需要保留 VMALLOC_RESERVED 大小的区域。VMALLOC_RESERVED 在每个体系结构中都有所不同,在 x86 中它是

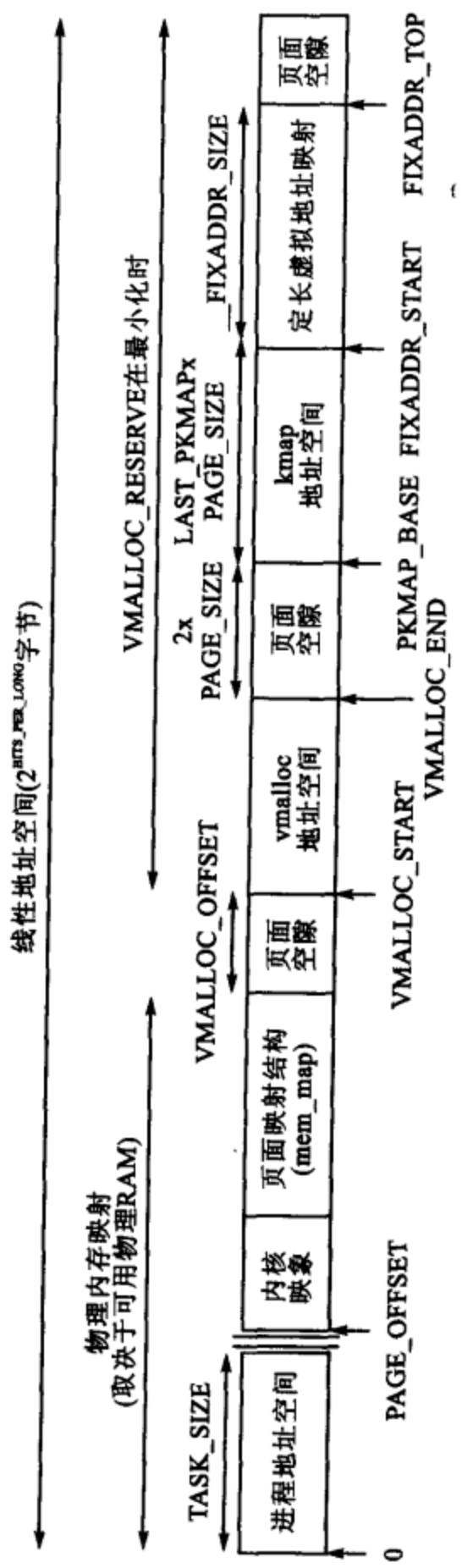


图 4.1 内核地址空间

128 MB。这正是 ZONE_NORMAL 大小通常只有 896 MB 的原因。vmalloc 区域由线性地址空间上端 1 GB 空间大小减去保留的 128 MB 区域所得。

4.2 地址空间的管理

进程可使用的地址空间由 `mm_struct` 管理, 它类似于 BSD 中的 `vmspace` 结构 [McK96]。

每个进程地址空间中都包含许多使用中的页面对齐的内存区域。它们不会相互重叠, 而且表示了一个地址的集合, 这个集合包含那些出于保护或其他目的而相互关联的页面。这些区域由 `struct vm_area_struct` 管理, 它们类似于 BSD 中的 `vm_map_entry` 结构。具体而言, 一个区域可能表示 `malloc()` 所使用的进程堆, 或是一个内存映射文件(例如共享库), 又或是一块由 `mmap()` 分配的匿名内存区域。这些区域中的页面可能还未被分配, 或已分配, 或常驻内存中又或已被交换出去。

如果一个区域是一个文件的映象, 那么它的 `vm_file` 字段将被设置。通过查看 `vm_file` → `f_dentry` → `d_inode` → `i_mapping` 可以获得这段区域所代表的地址空间内容。这个地址空间包含所有与文件系统相关的特定信息, 这些信息都是为了实现在磁盘上进行基于页面的操作。

图 4.2 中图示了各种地址空间相关结构之间的关联。表 4.1 则列举了许多影响地址空间和区域的系统调用。

表 4.1 内存区域相关的系统调用

| 系统调用 | 描述 |
|-----------------------|---|
| <code>fork()</code> | 创建一个具有新地址空间的进程。所有的页面都标记上写时复制(COW), 并且在页面中断以前一直由两个进程所共享。一旦发生写错误, 则为此错误进程复制 COW 页面。有时又被称为使一个 COW 页面失效 |
| <code>clone()</code> | <code>clone()</code> 允许创建一个共享上下文给父进程的新进程, 这在 Linux 中就是线程的实现。如果没有设置 <code>CLONE_VM</code> 位, 则 <code>clone()</code> 将创建一个新的地址空间, 其作用和 <code>fork()</code> 一样 |
| <code>mmap()</code> | <code>mmap()</code> 在进程线性地址空间中创建一个新区域 |
| <code>mremap()</code> | 重映射一个内存区域或者重设其大小。如果虚拟地址空间对映射来讲不可用, 那么移走该区域, 除非该移动操作被调用者所禁止。 |
| <code>munmap()</code> | 销毁部分或所有的区域。如果已经解除映射的区域位于已存在区域的中间, 则该存在的区域就分裂成两个单独的区域 |
| <code>shmat()</code> | 关联共享内存段到进程地址空间 |
| <code>shmdt()</code> | 从地址空间移除共享内存段 |
| <code>execve()</code> | 载入一个新的可执行文件, 并替换掉当前的地址空间 |
| <code>exit()</code> | 销毁一个地址空间和所有的区域 |

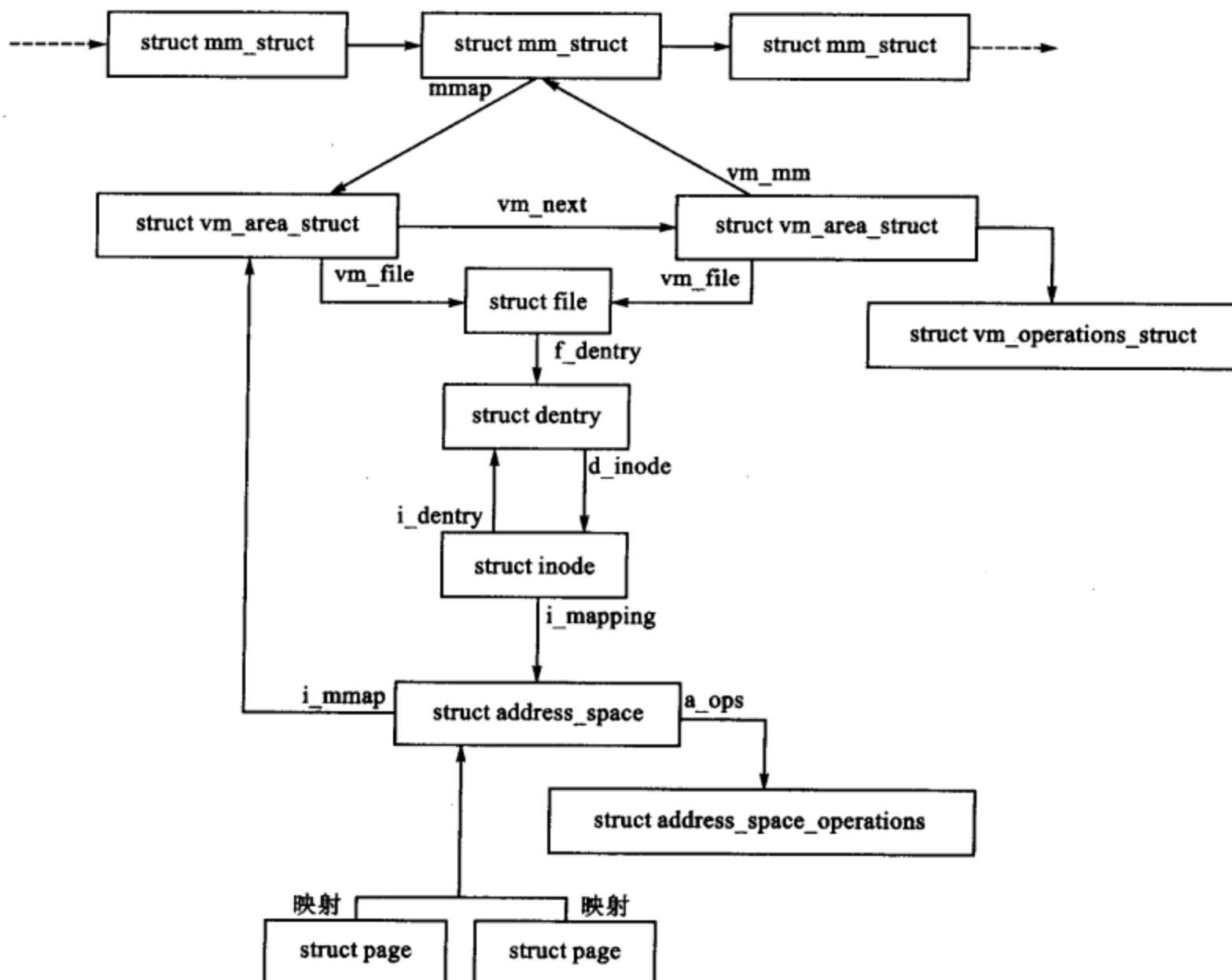


图 4.2 地址空间相关的数据结构

4.3 进程地址空间描述符

进程地址空间由 `mm_struct` 结构描述, 这意味着一个进程只有一个 `mm_struct` 结构, 且该结构在进程用户空间中由多个线程共享。事实上, 线程正是通过任务链表里的任务是否指向同一个 `mm_struct` 来判定的。

内核线程不需要 `mm_struct`, 因为它们永远不会发生缺页中断或访问用户空间。惟一的例外是 `vmalloc` 空间的缺页中断。缺页中断的处理代码认为该例外是一种特殊情况, 并借助主页表中的信息更新当前页表。由于内核线程不需要 `mm_struct`, 故 `task_struct->mm` 字段总为 `NULL`。对某些任务如引导空闲任务, `mm_struct` 永远不会被设置, 但对于内核线程而

言,调用 `daemonize()`也会调用 `exit_mm()`以减少对它的使用计数。

由于 TLB 的刷新需要很大的开销,特别是像在 PPC 这样的体系结构中,由于地址空间的内核部分对所有进程可见,那些未访问用户空间的进程所做的 TLB 刷新操作就是无效的,而 Linux 采用了一种叫“延迟 TLB”的技术避免了这种刷新操作。Linux 通过借用前个任务的 `mm_struct`,并放入 `task_struct->active_mm` 中,避免了调用 `switch_mm()`刷新 TLB。这种技术在上下文切换次数上取得了很大的进步。

进入延迟 TLB 时,在对称多处理机(Symmetric Multiprocessing, SMP)上,系统会调用 `enter_lazy_tlb()`函数以确保 `mm_struct` 不会被 SMP 的处理器所共享。而在 UP 机器上这是一个空操作。第二次用到延迟 TLB 是在进程退出时,系统会在该进程等待被父进程回收时,调用 `start_lazy_tlb()`函数。

该结构有两个引用计数,分别是 `mm_users` 和 `mm_count`。`mm_struct` 描述存取这个 `mm_struct` 用户空间的进程数,存取的内容包括页表、文件的映象等。例如,线程以及 `swap_out()` 代码会增加这个计数以确保 `mm_struct` 不会被过早地释放。当这个计数值减为 0 时, `exit_mmmap()`会删除所有的映象并释放页表,然后减小 `mm_count` 值。

`mm_count` 是对 `mm_struct` 匿名用户的计数。初始化为 1 则表示该结构的真实用户。匿名用户不用关心用户空间的内容,它们只是借用 `mm_struct` 的用户。例子用户是使用延迟 TLB 转换的核心线程。当这个计数减为 0 时,就可安全释放掉 `mm_struct`。存在两种计数是因为匿名用户需要 `mm_struct`,即便 `mm_struct` 中的用户映象已经被释放。但它不会延迟页表的释放操作。

<linux/sched.h> 对 `mm_struct` 的定义如下:

```

206 struct mm_struct {
207     struct vm_area_struct * mmap;
208     rb_root_t mm_rb;
209     struct vm_area_struct * mmap_cache;
210     pgd_t * pgd;
211     atomic_t mm_users;
212     atomic_t mm_count;
213     int map_count;
214     struct rw_semaphore mmap_sem;
215     spinlock_t page_table_lock;
216
217     struct list_head mmlist;
218
219     unsigned long start_code, end_code, start_data, end_data;
220     unsigned long start_brk, brk, start_stack;
221     unsigned long arg_start, arg_end, env_start, env_end;

```

```

225 unsigned long rss, total_vm, locked_vm;
226 unsigned long def_flags;
227 unsigned long cpu_vm_mask;
228 unsigned long swap_address;
229
230 unsigned dumpable:1;
231
232 /* Architecture-specific MM context */
233 mm_context_t context;
234 };

```

该结构中各个字段的含义如下。

mmap VMA: 地址空间中所有 VMA 的链表首部。

mm_rb VMA: VMA 都排列在一个链表中,且存放在一个红黑树中以加快查找速度。该字段表示树的根部。

mmap_cache: 最后一次通过 `find_vma()` 找到的 VMA 存放处,前提假设是该区可能会被再次用到。

pgd: 全局目录表的起始地址。

mm_users: 访问用户空间部分的用户计数值,本节已经介绍过。

mm_count: 匿名用户计数值。访问 `mm_struct` 的匿名用户技术值。本节已经介绍过。数值 1 针对真实用户。

map_count: 正被使用中的 vma 的数量。

mmap_sem: 这是一个读写保护锁并长期有效。因为用户需要这种锁作长时间的操作或者可能进入睡眠,所以不能用自旋锁。一个读操作通过 `down_read()` 来获得这个信号量。如果需要进行写操作,则通过 `down_write()` 获得该信号量,并在 VMA 链表更新后,获得 `page_table_lock` 锁。

page_table_lock: 该锁用于保护 `mm_struct` 中大部分字段,与页表类似,它防止驻留集大小(Resident Set Size, RSS) (见 `rss`) 计数和 VMA 被修改。

mmplist: 所有的 `mm_struct` 结构通过它链接在一起。

start_code, end_code: 代码段的起始地址和结束地址。

start_data, end_data: 数据段的起始地址和结束地址。

start_brk, brk: 堆的起始和结束地址。

start_stack: 栈的起始和结束地址。

arg_start, arg_end: 命令行参数的起始地址和结束地址。

env_start, env_end: 环境变量区域的起始和结束地址。

rss: 驻留集的大小是该进程常驻内存的页面数,注意,全局零页面不包括在

RSS 计数之内。

total_vm: 进程中所有 vma 区域的内存空间总和。

locked_vm: 内存中被锁住的常驻页面数。

def_flags: 只有一种可能值, VM_LOCKED。它用于指定在默认情况下将来所有的映射是上锁还是未锁。

cpu_vm_mask: 代表 SMP 系统中所有 CPU 的掩码值, 内部处理器中断(IPI)用这个值来判定一个处理器是否应执行一个特殊的函数。这对于每一个 CPU 的 TLB 刷新很重要。

swap_address: 当换出整个进程时, 页换出进程记录最后一次被换出的地址。

dumpable: 由 prctl()设置, 只有在跟踪一个进程时, 这个字段才有用。

context: 跟体系结构相关的 MMU 上下文。

对 mm_struct 结构进行操作的函数描述如表 4.2 所列。

表 4.2 内存区域描述器相关的函数

| 函数 | 描述 |
|---------------|--|
| mm_init() | 初始化一个 mm 结构, 设置每一个字段的初始值, 分配一个 PGD, 初始化自旋锁等 |
| allocate_mm() | 从 slab 分配器中分配一个 mm_struct() |
| mm_alloc() | 用 allocate_mm()分配一个 mm 结构, 并调用 mm_init()初始化该结构 |
| exit_mmap() | 遍历 mm_struct 结构, 并解除所有与其相关联的 VMA 映射 |
| copy_mm() | 为新的任务复制一个其所需 mm_struct 的完美副本。这在 fork 过程中用到 |
| free_mm() | 返回 mm 结构给 slab 分配器 |

4.3.1 分配一个描述符

系统有两个函数用于分配 mm_struct 结构。它们本质上相同, 但有一个重要的区别。Allocate_mm()只是一个预处理宏, 它从 slab allocator(见第 8 章)中分配一个 mm_struct。而 mm_alloc()从 slab 中分配, 然后调用 mm_init()函数对其进行初始化。

4.3.2 初始化一个描述符

系统中第一个 mm_struct 通过 init_mm()初始化。因为后继的子 mm_struct 都通过复制进行设置, 所以第 1 个 mm_struct 在编译时静态设置, 通过宏 INIT_MM()完成设置。

```
238 #define INIT_MM(name) \
239 { \
```

```

240 mm_rb: RB_ROOT, \
241 pgd: swapper_pg_dir, \
242 mm_users: ATOMIC_INIT(2), \
243 mm_count: ATOMIC_INIT(1), \
244 mmap_sem: __RWSEM_INITIALIZER(name.mmap_sem), \
245 page_table_lock: SPIN_LOCK_UNLOCKED, \
246 mm_list: LIST_HEAD_INIT(name.mm_list), \
247 }

```

第1个mm_struct创建后,系统将该mm_struct作为一个模板来创建新的mm_struct。copy_mm()函数完成复制操作,它调用init_mm()初始化与具体进程相关的字段。

4.3.3 销毁一个描述符

新的用户通过atomic_inc(&mm->mm_users)增加使用计数,同时通过mmput()减少该计数。如果mm_users变成0,所有的映射区域通过exit_mmap()释放,同时释放页表,因为已经没有用户使用这个用户空间。mm_count之所以通过mmdrop()减1,是因为所有页表和VMA的使用者都被看成是一个mm_struct用户。在mm_count变成0时,mm_struct会被释放。

4.4 内存区域

进程的地址空间很少能全部用满,一般都只是用到了其中一些分离的区域。区域由vm_area_struct来表示。区域之间不会交叉,它们各自代表一个有着相同属性和用途的地址集合。如一个被装载到进程堆空间的只读共享库就包含在这样的一个区域。一个进程所有已被映射的区域都可以在/proc/PID/maps里看到,其中PID是该进程的进程号。

一个区域可能有许多与它相关的结构,如图4.2所示。在图的顶端,有一个vm_area_struct结构,它足以用来表示一个匿名区域。

如果一个文件被映射到内存,则可以通过vm_file字段得到struct file。vm_file字段有一个指针指向struct inode,索引节点用于找到struct address_space,而struct address_space中包含与文件有关的所有信息,包括一系列指向与文件系统相关操作函数的指针,如读磁盘页面和写磁盘页面的操作。

vm_struct结构在<linux/mm.h>中声明如下:

```

44 struct vm_area_struct {
45     struct mm_struct * vm_mm;
46     unsigned long vm_start;

```

```
47 unsigned long vm_end;
49
50 /* linked list of VM areas per task, sorted by address */
51 struct vm_area_struct * vm_next;
52
53 pgprot_t vm_page_prot;
54 unsigned long vm_flags;
55
56 rb_node_t vm_rb;
57
63 struct vm_area_struct * vm_next_share;
64 struct vm_area_struct ** vm_pprev_share;
65
66 /* Function pointers to deal with this struct. */
67 struct vm_operations_struct * vm_ops;
68
69 /* Information about our backing store: */
70 unsigned long vm_pgoff;
72 struct file * vm_file;
73 unsigned long vm_raend;
74 void * vm_private_data;
75 };
```

下面是对该结构中字段的简要解释。

vm_mm: 这个 VMA 所属的 mm_struct。

vm_start: 这个区域的起始地址。

vm_end: 这个区间的结束地址。

vm_next: 在一个地址空间中的所有 VMA 都按地址空间次序通过该字段简单地链接在一起。可以很有趣地发现该 VMA 链表在内核中所使用的单个链表中是非常少见的。

vm_page_prot: 这个 VMA 对应的每个 PTE 里的保护标志位。其不同的位在表 3.1 中有描述。

vm_flags: 这个 VMA 的保护标志位和属性标志位。它们都定义在<linux/mm.h>中，描述见表 4.3。

vm_rb: 同链表一样，所有的 VMA 都存储在一个红黑树上以加快查找速度。这对于在发生缺页时能快速找到正确的区域非常重要，尤其是大量的映射区域。

表 4.3 内存区域标志位

| 保护标志位 | |
|----------------|--|
| 标志位 | 描述 |
| VM_READ | 页面可能被读取 |
| VM_WRITE | 页面可能被写入 |
| VM_EXEC | 页面可能被执行 |
| VM_SHARED | 页面可能被共享 |
| VM_DONTCOPY | VMA 不能在 fork 时被复制 |
| VM_DONTEXPAND | 防止一个区域被重新设置大小。标志位没有被使用过 |
| mmap 相关的标志位 | |
| VM_MAYREAD | 允许设置 VM_READ 标志位 |
| VM_MAYWRITE | 允许设置 VM_WRITE 标志位 |
| VM_MAYEXEC | 允许设置 VM_EXEC 标志位 |
| VM_MAYSHARE | 允许设置 VM_SHARE 标志位 |
| VM_GROWSDOWN | 共享段（可能是栈）可能减小 |
| VM_GROWSUP | 共享段（可能是堆）可能增大 |
| VM_SHM | 页面被共享的 SHM 内存段所使用 |
| VM_DENYWRITE | MAP_DENYWRITE 经 mmap() 转换得到。目前不再使用 |
| VM_EXECUTABLE | MAP_EXECUTABLE 经 mmap() 转换得到。目前不再使用 |
| VM_STACK_FLAGS | setup_arg_flags() 所使用的标志位, 用于设置栈 |
| 上锁标志位 | |
| VM_LOCKED | 如果设置了该位, 那么页面将不会被换出。由 mlock() 进行设置 |
| VM_IO | 发出信号以表明该区域是一个对设备进行 I/O 的内存映射区域。它同样防止了区域出现内核崩溃的情况 |
| VM_RESERVED | 不要换出该区域。其由设备驱动所使用 |
| madvise() 标志位 | |
| VM_SEQ_READ | 表明页面将被顺序访问 |
| VM_RAND_READ | 表明该区域的预读不可用 |

vm_next_share: 这个指针把由文件映射而来的 VMA 共享区域链接在一起(如共享库)。

vm_pprev_share: vm_next_share 的辅助指针。

vm_ops: 包含指向与磁盘作同步操作时所需函数的指针。vm_ops 字段包含有指向 open(), close() 和 nopage() 的函数指针。

vm_pgoff: 在已被映射文件里对齐页面的偏移。

vm_file: 指向被映射的文件的指针。

vm_raend: 预读窗口的结束地址。在发生错误时,一些额外的页面将被收回,这个值决定了这些额外页面的个数。

vm_private_data: 一些设备驱动私有数据的存储,与内存管理器无关。

所有的区域按地址排序由指针 vm_next 链接成一个链表。寻找一个空闲的区间时,只需要遍历这个链表即可。但若在发生缺页中断时搜索 VMA 以找到一个指定区域,则是一个频繁操作。正因为如此,才有了红黑树,因为它平均只需要 $O(\log N)$ 的遍历时间。红黑树通过排序使得左结点的地址小于当前结点的地址,而当前结点的地址又小于右结点的地址。

4.4.1 内存区域的操作

VMA 提供三个操作函数,分别是 open(), close() 和 nopage()。VMA 通过类型 vm_operations_struct 的 vma->vm_ops 提供上述几个操作函数。该结构包含三个函数指针,它在<linux/mm.h> 中的声明如下所示:

```
133 struct vm_operations_struct {
134     void (*open)(struct vm_area_struct * area);
135     void (*close)(struct vm_area_struct * area);
136     struct page * (*nopage)(struct vm_area_struct * area,
137                             unsigned long address,
138                             int unused);
139 };
```

每当创建或者删除一个区域时系统会调用 open() 和 close() 函数。只有一小部分设备使用这些操作函数,如文件系统和 system v 的共享区域。在打开或关闭区域时,需要执行额外的操作。例如,system V 中 open() 回调函数会递增使用共享段的 VMA 的数量。

我们关心的主要操作函数是 nopage() 回调函数,在发生缺页中断时 do_no_page() 会使用该回调函数。该回调函数负责定位该页面在高速缓存中的位置,或者分配一个新页面并填充请求的数据,然后返回该页面。

大多数被映射的文件会用到名为 generic_file_vm_ops 的 vm_operations_struct()。它只注册一个函数名为 filemap_nopage() 的 nopage() 函数。该 nopage() 函数或者定位该页面在页面高速缓存中的位置,或者从磁盘上读取数据。该结构在 mm/filemap.c 中声明如下:

```
2243 static struct vm_operations_struct generic_file_vm_ops = {
2244     .nopage = filemap_nopage,
2245 };
```

4.4.2 有后援文件/设备的内存区域

如表 4.2 所列,在有后援文件的区域中,vm_file 引出了相关的 address_space。address_space 结构包含一些与文件系统相关的信息,如必须写回到磁盘的脏页面数目。该结构在 <linux/fs.h> 中声明如下:

```

406 struct address_space {
407     struct list_head clean_pages;
408     struct list_head dirty_pages;
409     struct list_head locked_pages;
410     unsigned long nrpages;
411     struct address_space_operations * a_ops;
412     struct inode * host;
413     struct vm_area_struct * i_mmap;
414     struct vm_area_struct * i_mmap_shared;
415     spinlock_t i_shared_lock;
416     int gfp_mask;
417 };

```

各字段简要描述如下。

clean_pages: 不需要后援存储器同步的干净页面链表。

dirty_pages: 需要后援存储器同步的脏页面链表。

locked_pages: 在内存中被锁住的页面链表。

nrpages: 在地址空间中正被使用且常驻内存的页面数。

a_ops: 是一个操纵文件系统的函数结构。每一个文件系统都提供其自身的 address_space_operations, 即便在某些时候它们使用普通的函数。

host: 这个文件的索引节点。

i_mmap: 使用 address_space 的私有映射链表。

i_mmap_shared: 该地址空间中共享映射的 VMA 链表。

i_shared_lock: 保护此结构的自旋锁。

gfp_mask: 调用 __alloc_pages() 所要用到的掩码。

内存管理器需要定期将信息写回磁盘。但是内存管理器并不知道也不关心信息如何写回到磁盘, 因此需要 a_ops 结构来调用相关的函数。它在 <linux/fs.h> 中的声明如下所示:

```

385 struct address_space_operations {
386     int (* writepage)(struct page * );
387     int (* readpage)(struct file * , struct page * );

```

```

388 int (* sync_page)(struct page *);
389 /*
390 * ext3 requires that a successful prepare_write() call be
391 * followed by a commit_write() call - they must be balanced
392 */
393 int (* prepare_write)(struct file *, struct page *,
394                      unsigned, unsigned);
394 int (* commit_write)(struct file *, struct page *,
395                      unsigned, unsigned);
395 /* Unfortunately this kludge is needed for FIBMAP.
396 * Dont use it */
396 int (* bmap)(struct address_space *, long);
397 int (* flushpage)(struct page *, unsigned long);
398 int (* releasepage)(struct page *, int);
399 #define KERNEL_HAS_O_DIRECT
400 int (* direct_IO)(int, struct inode *, struct kiobuf *,
401                    unsigned long, int);
401 #define KERNEL_HAS_DIRECT_FILEIO
402 int (* direct_fileIO)(int, struct file *, struct kiobuf *,
403                        unsigned long, int);
403 void (* removepage)(struct page *);
404 };

```

该结构中的字段都是函数指针，它们描述如下。

writepage: 把一个页面写到磁盘。写到文件里的偏移量保存在页结构中。寻找磁盘块的工作则由具体文件系统的代码完成，参考 `buffer.c`; `block_write_full_page()`。

readpage: 从磁盘读一个页面。参考 `buffer.c`; `block_read_full_page()`。

sync_page: 同步一个脏页面到磁盘。参考 `buffer.c`; `blobk_sync_page()`。

prepare_write: 在复制用户空间数据到将要写回磁盘的页面之前，调用此函数。对于一个日志文件系统，该操作保证了文件系统日志是最新的。而对于一般的文件系统而言，它也确保了分配所需要的缓冲区页面。参考 `buffer.c`; `block_prepare_write()`。

commit_write: 在从用户空间复制数据之后，会调用该函数将数据提交给磁盘。参考 `buffer.c`; `block_commit_write()`。

bmap: 映射一个磁盘块使得裸设备 I/O 操作能够执行。虽然它在后援为一个交换文件而非交换分区时也用于换出页面，但它还是主要与具体的文件系统有关。

flushpage: 该函数确保在释放一个页面之前已不存在等待该页面的 I/O 操作。参考 `buffer.c`; `discard_bh_page()`。

releasepage: 该函数在释放掉页面之前将刷新所有与这个页面有关的缓冲区。参考 `try_`

to_free_buffers()。

direct_I/O: 该函数在对一个索引节点执行直接 I/O 时使用。由于 #define 的存在,因此外部模块可以决定在编译时间该函数是否可用,因为它仅仅在 2.4.21 中被引入。

direct_fileI/O: 用于对 struct file 进行直接 I/O。并且, #define 也由于外部模块而存在,因为该 API 只在 2.4.22 中被引入。

removepage: 一个候选回调函数,当页面从页面高速缓存中由 remove_page_from_inode_queue() 移除时使用。

4.4.3 创建内存区域

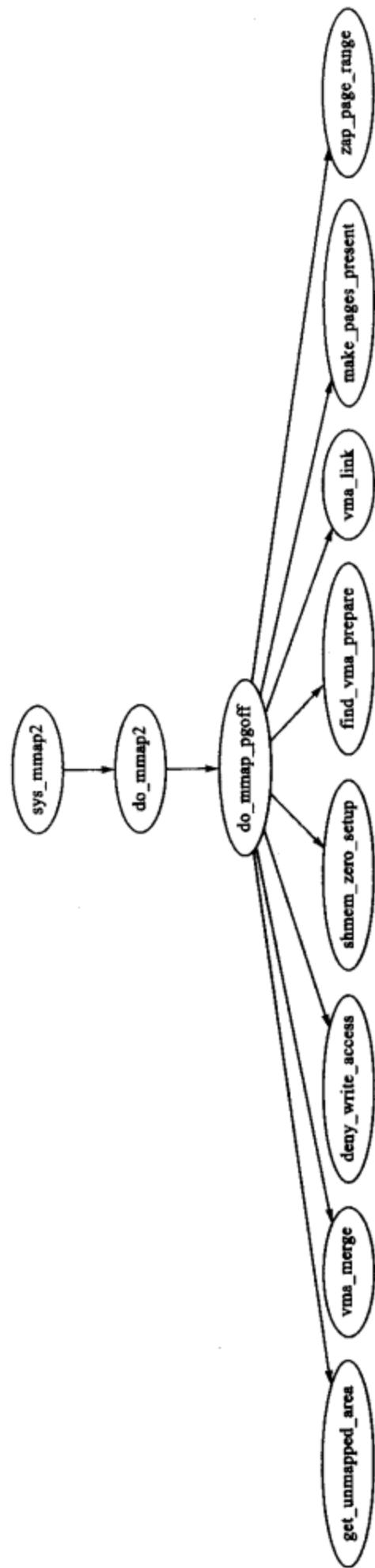
系统调用 mmap() 为一个进程创建新的内存区域。x86 中, mmap() 会调用 sysy_mmap2(), 而 sysy_mmap2() 进一步调用 do_mmap2(), 三个函数都使用相同的参数。do_mmap2() 负责获得 do_mmap_pgoff() 所需要的参数。而 do_mmap_pgoff() 才是所有体系结构中创建新区域的主要函数。

do_mmap2() 首先清空 flags 参数中的 MAP_DENYWRITE 和 MAP_EXECUTABLE 位, 因为 Linux 用不到这两个标志位, 这一点在 mmap() 的操作手册里有说明。如果映射一个文件, do_mmap2() 将通过文件描述符查找到相应的 struct file, 并在调用 do_mmap_pgoff() 前获得 mm_struct->mmap_sem 信号量。

do_mmap_pgoff() 首先做一些合法检查。它首先检查在文件或设备被映射时, 相应的文件系统和设备的操作函数是否有效。然后, 它检查需要映射的大小是否与页面对齐, 并且保证不会在内核空间创建映射。最后, 它必须保证映射的大小不会超过 pgoff 的范围以及这个进程没有过多的映射区域。

这个函数的余下部分比较大, 大致有以下几个步骤:

- 参数的合法检查。
- 找出内存映射所需的空闲线性地址空间。如果系统提供了基于文件系统和设备的 get_unmapped_area() 函数, 那么会调用它, 否则将使用 arch_get_unmapped_area() 函数。
- 获得 VM 标志位, 并根据文件存取权限对它们进行检查。
- 如果在映射的地方有旧区域存在, 系统会修正它, 以便新的映射能用这一部分的区域。
- 从 slab 分配器里分配一个 vm_area_struct, 并填充它的各个字段。
- 把新的 VMA 链接到链表中。
- 调用与文件系统或设备相关的 mmap() 函数。
- 更新数据并返回。

图 4.3 调用图: `sys_mmap2()`

4.4.4 查找已映射内存区域

一个常用的操作是查找给定地址所属的 VMA, 如处理缺页中断时, 函数 `find_vma()` 就负责这个操作, 与线性区有关的 API 函数如表 4.4 所列。

表 4.4 内存区域 VMA 的 API

| |
|---|
| <pre>struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr)</pre> <p>查找涉及给定地址的 VMA。如果该区域不存在, 将返回离请求地址最近的 VMA</p> |
| <pre>struct vm_area_struct * find_vma_prev(struct mm_struct * mm, unsigned long addr, struct vm_area_struct ** pprev)</pre> <p>和 <code>find_vma()</code> 一样, 但除此以外它还为 VMA 指出返回的 VMA。系统不常使用它, 但 <code>sys_mprotect()</code> 是个例外, 因为它常常需要调用函数 <code>find_vma_prepare()</code></p> |
| <pre>struct vm_area_struct * find_vma_prepare(struct mm_struct * mm, unsigned long addr, struct vm_area_struct ** pprev, rb_node_t ** rb_link, rb_node_t ** rb_parent)</pre> <p>和 <code>find_vma()</code> 一样, 但除此以外它还在链表中查找运行中的 VMA, 正如红黑树的节点需要表现出插入树这一行为一样</p> |
| <pre>struct vm_area_struct * find_vma_intersection(struct mm_struct * mm, unsigned long start_addr, unsigned long end_addr)</pre> <p>返回与给定地址范围相交的 VMA。这在检查一个线性地址区域是否被任一个 VMA 所使用时非常有用</p> |
| <pre>int vma_merge(struct mm_struct * mm, struct vm_area_struct * prev, rb_node_t * rb_parent, unsigned long addr, unsigned long end, unsigned long vm_flags)</pre> <p>尝试拓展一个补给的 VMA 以覆盖新的地址范围。如果 VMA 不能向前拓展, 那么替换的方法是检查接下来的 VMA 是否可以向后拓展以涵盖该地址范围。区域可能会在没有任何文件/设备映射和许可匹配的情况下合并掉</p> |
| <pre>unsigned long get_unmapped_area(struct file * file, unsigned long addr, unsigned long len, unsigned long pgoff, unsigned long flags)</pre> <p>返回足以覆盖所请求内存大小的空闲内存区域地址。该函数主要在一个新的 VMA 被创建时使用</p> |
| <pre>void insert_vm_struct(struct mm_struct *, struct vm_area_struct *)</pre> <p>插入一个新的 VMA 到线性地址空间中</p> |

`find_vma()` 首先检查 `mmap_cache` 所代表的 VMA, `mmap_cache` 是最后一次调用 `find_vma()` 时返回的结果。如果不是目标 VMA, 系统将遍历 `mm_rb` 存放的红黑树。如果给定的地址不包含在任何一个 VMA 中, 函数将返回离给定地址最近的 VMA, 所以调用 `find_vma()` 后一定要检查返回的 VMA 是否包含给定的地址。

第 2 个函数是 `find_vma_prev()`, 和 `find_vma()` 有些相似, 不过它返回指向目标 VMA 的前一个 VMA 的指针。这个函数一般很少用, 但在判断两个 VMA 能否合并时要用到, 而且删

除一个内存区域时也要用到。

最后一个函数是 `find_vma_intersection()`, 用于查找与给定地址范围相交的 VMA。该函数主要在扩展一个内存区域且调用 `do_brk()` 过程中用到, 其中要确保扩展的区域不会与一个原有的区域相交。

4.4.5 查找空闲内存区域

映射内存时, 先得获取足够大的空闲区域, `get_unmapped_area()` 就用于查找一个空闲区域。

调用图如图 4.4 所示, 获取空闲区域并不复杂。

`get_unmapped_area()` 有很多参数: `struct file` 表示映射的文件或设备; `pgoff` 表示文件的偏移量; `address` 表示请求区域的起始地址; `length` 表示请求区域的长度; `flags` 表示此区域的保护标志位。

如果映射的是设备, 比如视频卡, 就还要使用 `f_op->get_unmapped_area()`。这是因为设备或文件有额外的操作要求, 而通用代码并不能完成额外的要求, 比如映射的地址必须对齐到一个特殊的虚拟地址。

如果没有特殊的要求, 系统将调用体系结构相关的函数 `arch_get_unmapped_area()`。并不是所有的体系结构都提供自己的函数, 这时将调用 `mm/mmap.c` 中提供的通用版本函数。

4.4.6 插入内存区域

插入一个新区域的主要函数是 `insert_vm_struct()`, 调用图如图 4.5 所示。该函数很简单, 首先调用 `find_vma_prepare()` 找到将要插入到两个 VMA 间的新区域, 以及新区域在红黑树中的正确节点。然后调用 `vma_link()` 将新区域链接到 VMA 链表。

Linux 很少用到函数 `insert_vm_start()`, 因为它不增加 `mm_struct` 里字段 `map_count` 的值。而函数 `_insert_vm_struct` 功能和它相似, 却经常用到, 它增加 `map_count` 的值。

有两个不同的链接函数: `vma_link()` 和 `_vma_link()`。`vma_link()` 用于没有加锁的情况, 在调用 `_vma_link()` 之前先获取所需要的锁。如果是文件映射, 还应该锁住文件, 然后 `_vma_link()` 将该 VMA 添加到相关的链表中。

值得注意的是很多函数不调用 `insert_vm_struct()`, 而是自己调用 `find_vma_prepare()`, 然后调用 `vma_link()` 以避免多次遍历 VMA 树。

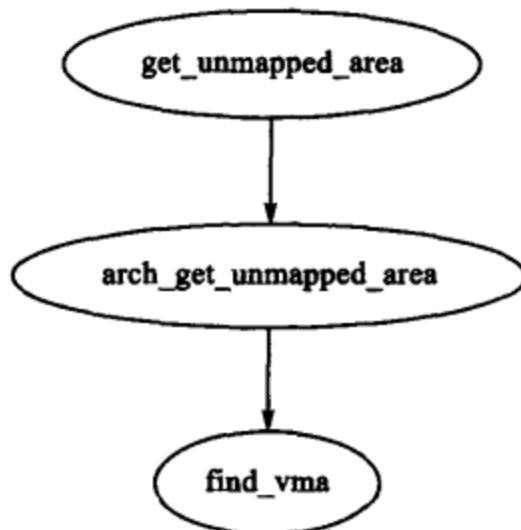
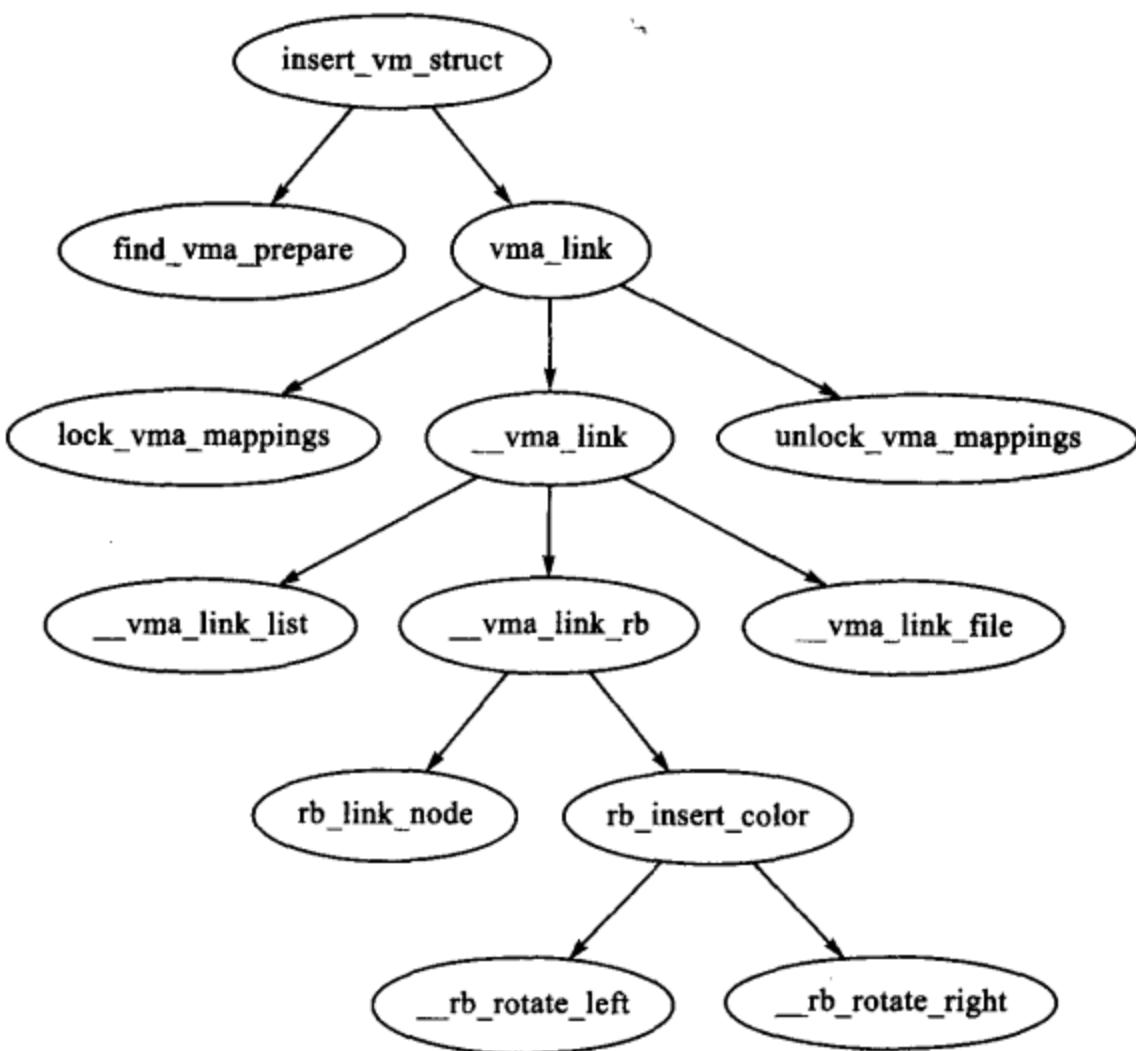


图 4.4 调用图: `get_unmapped_area()`

图 4.5 调用图: `insert_vm_struct()`

`__vma_link()` 分 3 步，每步都是一个单独的函数。第 1 步是 `__vma_link_list()`，它把新区域插入到单向 VMA 线性链表中，如果是第一个节点（比如前驱为 `NULL`），那么它将成为红黑树的根节点。第 2 步是 `__vma_link_rb()`，它把新区域插入到 VMA 红黑树中。第 3 步是 `__vma_link_file()`，用于处理文件共享映射，它通过字段 `vm_pprev_share` 和 `vm_next_share` 把 VMA 插入到 VMA 的链表中。

4.4.7 合并邻接区域

如果文件和权限都匹配，Linux 一般使用函数 `merge_segments()` [Hac02] 合并相邻的内存区域。其目的是减少 VMA 的数量，因为大量的操作会导致创建大量的映射，如系统调用 `sys_mprotect()`。该合并操作的开销很大，它会遍历大部分的映射，接着被删掉，特别是存在大量的映射时，`merge_segments()` 将会花费很长时间。

目前与上述函数等价的函数是 `vma_merge()`，它仅在 2 个地方用到。第 1 个是在 `sys_mmap()` 中，当映射匿名区域时会调用它，因为匿名区域通常可以合并。第 2 个是在 `do_brk()`

中,给区域扩展一个新分配的区域后,应该将这 2 个区域合并,这时就调用 `vma_merge()`。`vma_merge()`不仅合并两个线性区,还检查现有的区域能否安全地扩展到新区域,从而无需创建一个新区域。在没有文件或设备映射且两个区域的权限相同时,一个区域才能扩展。

其他地方也可能合并区域,只不过没有显式地调用函数。第 1 个就是在修正区域过程中系统调用 `sys_mprotect()` 时,如果两个区域的权限一致,就要确定它们是否要合并。第 2 个是在 `move_vma()` 中,它很可能把相似的区域移到一起。

4.4.8 重映射和移动内存区域

系统调用 `mremap()` 用于扩展或收缩现有区域,由函数 `sys_mremap()` 实现。在一个区域扩展时有可能移动该区域,或者在一个区域将要与其他区域相交并且没有指定标志位 `MREMAP_FIXED` 时,也可能移动该区域。调用图如图 4.6 所示。

移动一个区域时, `do_mremap()` 将首先调用 `get_unmapped_area()`, 找到一个足以容纳扩展后的映射空闲区域, 然后调用 `move_vma()` 把旧 VMA 移到新位置。`move_vma()` 调用图如图 4.7 所示。

`move_vma()` 首先检查新区域能否和相邻的 VMA 合并。如果不能合并,将创建一个新 VMA,每次分配一个 PTE。然后调用 `move_page_tables()` (调用图如图 4.8 所示), 将旧映射中所有页表项都复制过来。尽管可能有更好的移动页表的方法,但是采用这种方法使得错误恢复更直观,因为相对而言,回溯更直接。

页面的内容并没有被复制,而是调用 `zap_page_range()` 将旧映射中的所有页都交换出去或者删除,通常缺页中断处理代码会将辅存、文件中的页交换回至内存,或调用设备相关函数 `do_nopage()`。

4.4.9 对内存区域上锁

Linux 可以通过系统调用 `mlock()` 锁住给定地址范围的内存, `mlock()` 由 `sys_lock()` 实现, 调用图如图 4.9 所示。`sys_lock()` 作为高级函数,很简单,它为需要上锁的地址范围创建 VMA,并设置 `VM_LOCKED` 标志位,然后调用 `make_pages_present()` 函数以保证所有的页都在内存中。第 2 个系统调用是 `mlockall()`,由 `sys_mlockall()` 实现。`sys_mlockall()` 与 `sys_mlock()` 所做的工作一样,仅做了很少的扩展,除了影响调用进程中的每个 VMA。二者都依赖于核心函数 `do_mlock()`,该函数处理真正的工作,包括查找受影响的 VMA 和决定用来修整区域的函数,这在后面会有描述。

对于要上锁的内存还有一些限制:因为 VMA 是页对齐的,待锁定的地址范围也必须是页对齐的,只需要简单地向上取整即可保证。第 2 个限制是不能超过系统管理员设置的进程限

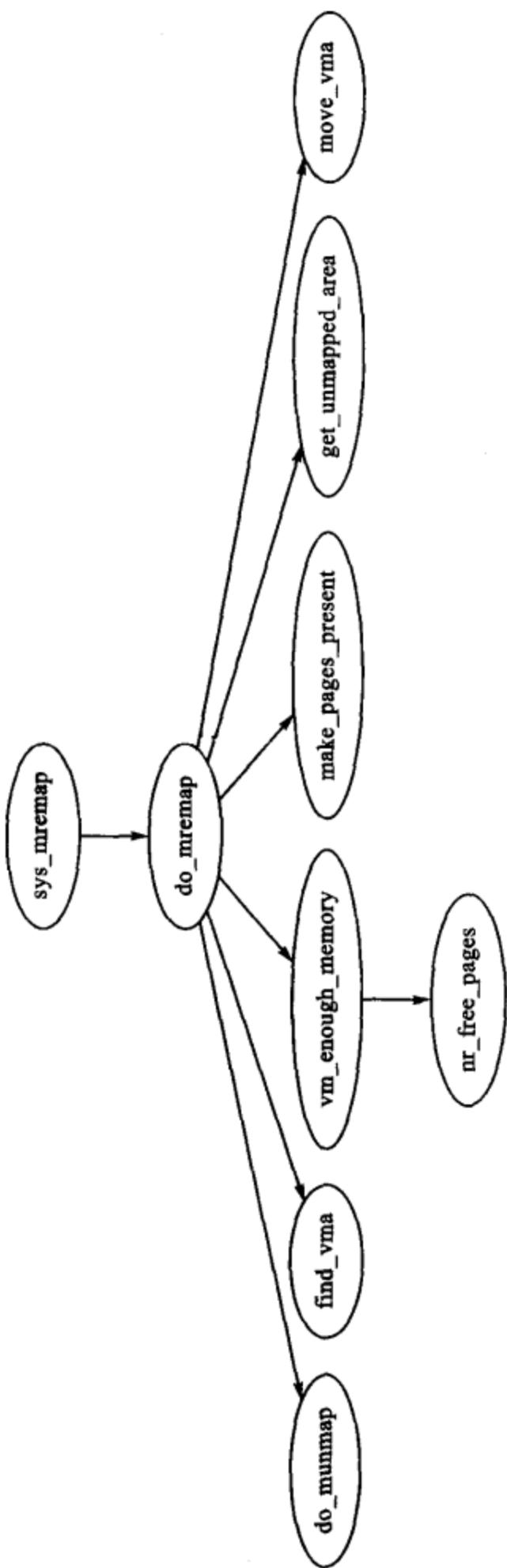


图 4.6 调用图：sys_mremap()

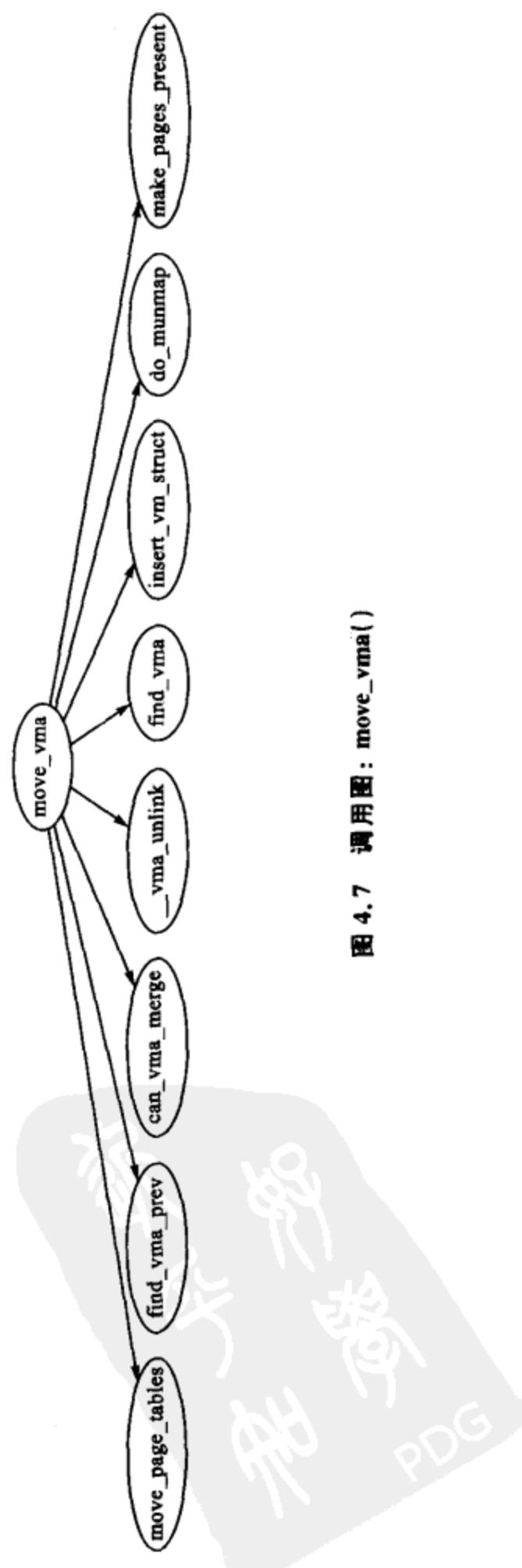


图 4.7 调用图：move_vma()

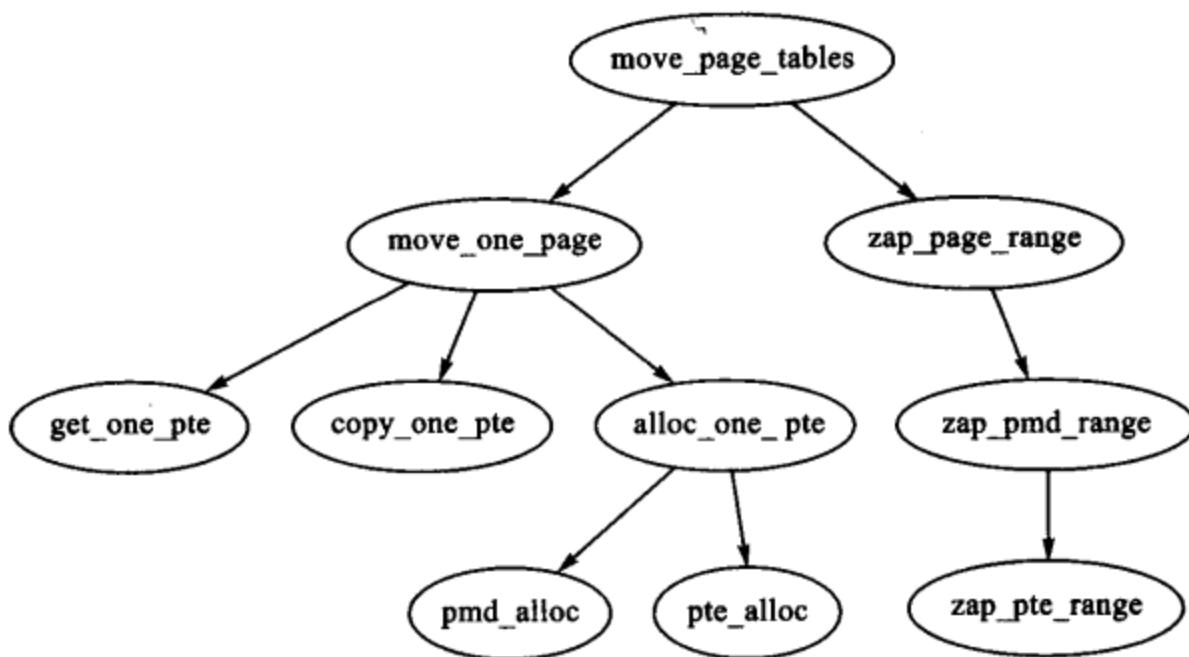


图 4.8 调用图：move_page_tables()

制 RLIMIT_MLOCK。最后一个限制是每个进程每次只能锁住一半的物理内存,这不太起作用,因为它无法阻止一个进程创建大量的子进程以及每个子进程都锁住一部分,这与仅允许根进程锁住页面也没有多大区别。可以有把握地假定根进程是可信任的和可控制的,否则的话,系统崩溃时系统管理员无法获取合适的有价值的保留信息。

4.4.10 对区域解锁

系统调用 munlock() 和 munlockall() 分别是相应的解锁函数,它们分别由 sys_munlock() 和 sys_munlockall() 实现。它们比上锁函数简单得多,因为不必做过多的检查,它们都依赖于同一个函数 do_mmap() 来修整区域。

4.4.11 上锁后修整区域

在上锁或解锁时,VMA 将受到 4 个方面的影响,每个必须由 mlock_fixup() 修正。上锁可能影响到所有的 VMA 时,系统就要调用 mlock_fixup_all() 进行修正。第 2 个条件是被锁住区域地址的起始值,由 mlock_fixup_start() 处理,Linux 需要分配一个新的 VMA 来映射新区域。第 3 个条件是被锁住区域地址的结束值,由 mlock_fixup_end() 处理。最后,mlock_fixup_middle() 处理映射区域的中间部分,此时需要分配 2 个新的 VMA。

值得注意的是创建上锁的 VMA 时从不合并,即使该区域被解锁也不能合并。一般而言,已经锁住某个区域的进程没必要再次锁住同一区域,因为经常开销处理器计算能力来合并和分裂区域是不值得的。

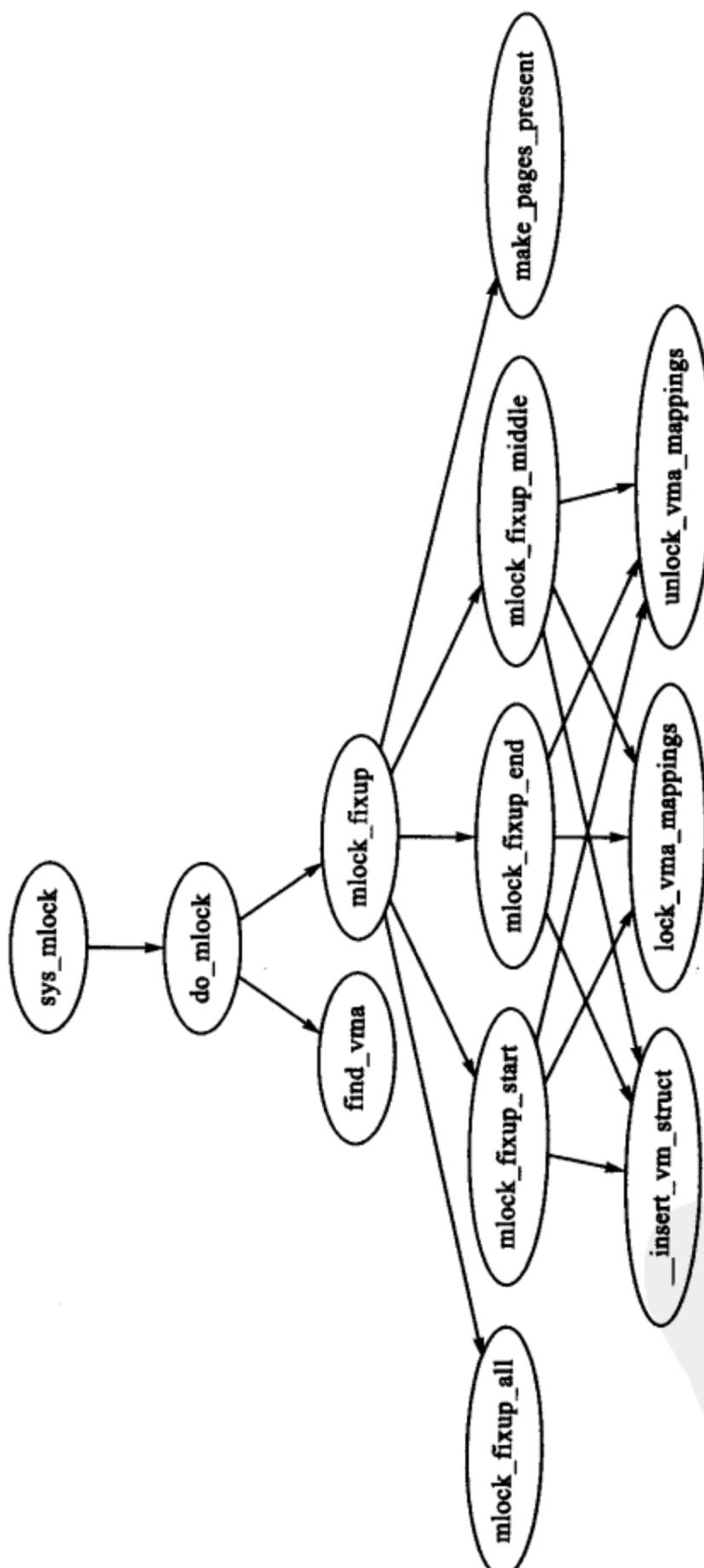


图 4.9 调用图：sys_mlock()

4.4.12 删除内存区域

do_munmap()负责删除区域。它相对于其他的区域操作函数,比较简单,它基本上可分为3个部分。第1部分是修整红黑树,第2部分是释放和对应区域相关的页面和页表项,第3部分是如果生成了空洞就修整区域。

为保证红黑树已排好序,所有要删除的VMA都添加到称为free的链表,然后利用rb_erase()从红黑树中删除。如果区域还存在,那么在后来的修整中将以它们的新地址添加到系统中。

接下来遍历free所指向的VMA链表,即使删除线性区的一部分,系统也会调用remove_shared_vm_struct()把共享文件映射删掉。再次说明,如果仅是部分删除,在修整中也会重新创建。zap_page_range()删掉所有与区域相关的页面,而部分删除则调用unmap_fixup处理。

最后调用free_ptables()释放所有和对应区域相关的页表项。这里注意到页表项并没有彻底释放完很重要。它反而释放所有的PGD和页目录项,因此,如果仅有半的PGD用于映射,则不需要释放页表项。这是因为释放少量的页表项和数据结构代价很大,而且这些结构很小,另外它们还可能会再次被使用。

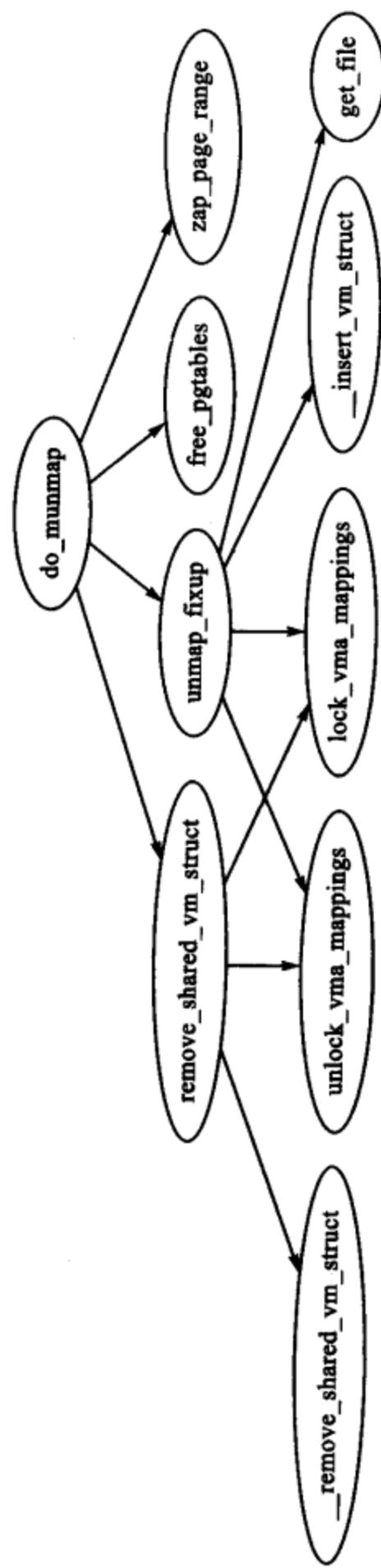
4.4.13 删除所有的内存区域

进程退出时,必须删除与其mm_struct相关联的所有VMA,由函数exit_mmap()负责操作。这是个非常简单的函数,在遍历VMA链表前将刷新CPU高速缓存,依次删除每一个VMA并释放相关的页面,然后刷新TLB和删除页表项,这个过程在代码注释中有详细描述。

4.5 异常处理

VM中很重要的一个部分就是如何捕获内核地址空间异常,这并不是内核的bug。这部分不讨论如何处理诸如除数为零的异常错误,仅关注由于页面中断而产生的异常。有两种情况会发生错误的引用。第1种情况是进程通过系统调用向内核传递了一个无效的指针,内核必须能够安全地陷入,因为开始只检查地址是否低于PAGE_OFFSET。第2种情况是内核使用copy_from_user()或copy_to_user()读写用户空间的数据。

编译时,连接器将在代码段中的__ex_table处创建异常表,异常表开始于__start_ex_table,结束于__stop_ex_table。每个表项的类型是exception_table_entry,由可执行点和修整子程序二者组成。在产生异常时,缺页中断处理程序不能处理,它调用search_exception_table()查看是否为引起中断的指令提供了修整子程序,若系统支持模块,还要搜索每个模块的异常表。

图 4.10 调用图: `do_unmap()`

如果在表中找到了当前异常的地址,将返回对应的修整子程序的位置并执行该子程序。在 4.7 节中将看到如何捕获读写用户地址空间数据的异常。

4.6 缺页中断

进程线性地址空间里的页面不必常驻内存。例如,进程的分配请求并被立即满足,空间仅仅保留为满足 `vm_area_struct` 的空间。其他非常驻内存页面的例子有,页面可能被交换到后援存储器,还有就是写一个只读页面。

和大多操作系统一样,Linux 采用请求调页技术来解决非常驻页面的问题。在操作系统捕捉到由硬件发出的缺页中断异常时,它才会从后援存储器中调入请求的页面。由后援存储器的特征可知,采取页面预取技术可以减少缺页中断[MM87],但是 Linux 在这方面相当原始。在将一个页面读入交换区时,`swapin_readahead()`会预取该页面后多达 $2^{\text{page_cluster}}$ 的页面,并放置在交换区中。不幸的是,很快要用到的页面只有一次机会邻近交换区,从而导致预约式换页技术很无效。Linux 将采用适合应用程序的预约式换页策略[KMC02]。

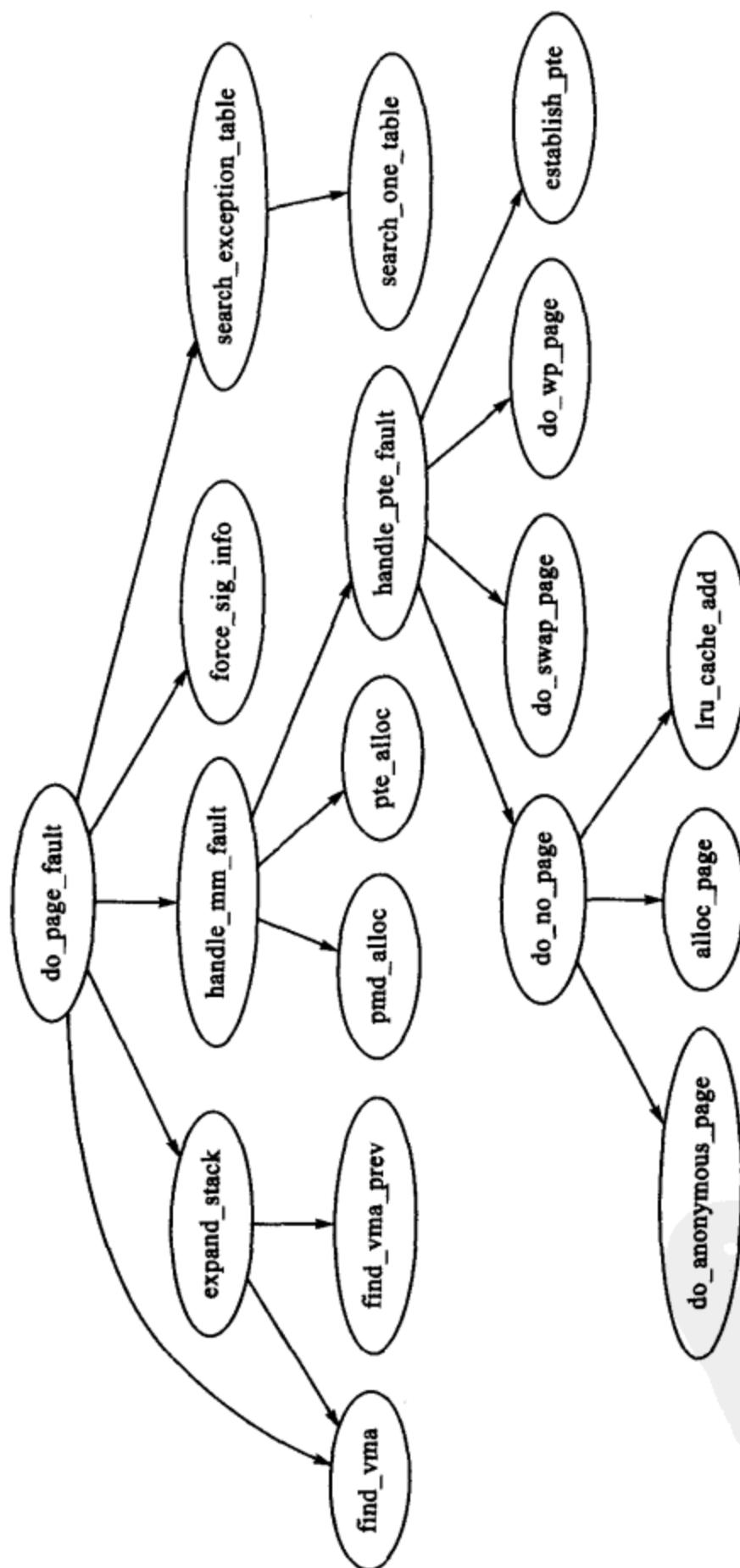
有两种类型的缺页中断,分别是主缺页中断和次缺页中断。当要费时地从磁盘中读取数据时,就会产生主缺页中断,其他的就是次缺页中断,或者是轻微的缺页中断。Linux 通过字段 `task_struct->maj_fault` 和 `task_struct->min_fault` 来统计各自的数目。

Linux 中能处理的页面异常类型如表 4.5 所列,本章后面将会详细讨论。

表 4.5 缺页中断的发生原因

| 异常 | 类型 | 动作 |
|-----------------|----|--|
| 线性区有效但页面没有分配 | 次要 | 通过物理页面分配器分配一个页面帧 |
| 线性区无效但是可扩展,例如堆栈 | 次要 | 扩展线性区并分配一页 |
| 页面被交换但是在交换高速缓存中 | 次要 | 从交换高速缓存中删除并分配给进程 |
| 页面被交换至后援存储器 | 主要 | 通过 PTE 中的信息查找页面并从磁盘读到内存中 |
| 写只读页面 | 次要 | 如果是 COW 页,就复制一页,标志为可写的并分配给进程,如果是写异常,就发送 SIGSEGV 信号 |
| 线性区无效或者没有访问权限 | 错误 | 给进程发送 SIGSEGV 信号 |
| 异常发生在内核地址空间 | 次要 | 如果异常发生在 <code>vmalloc</code> ,就更新当前进程页表,相对于 <code>init_mm</code> 的主内核页表。这是惟一的有效发生内核页面异常的情况 |
| 异常发生在内核模态用户空间 | 错误 | 如果发生异常,表明内核不能从用户空间正确地复制数据,这是非常严重的内核 Bug |

每种体系结构都会注册一个与体系结构相关的处理缺页中断的函数,函数名是任意的,通常是 `do_page_fault()`,x86 中调用图如图 4.11 所示。

图 4.11 调用图: `do_page_fault()`

这个函数提供了大量的信息,例如发生异常的地址,是由于页面没找到还是页面保护错误,是读异常还是写异常,来自于用户空间还是内核空间。它负责确定异常的类型及异常如何被体系结构无关的代码处理。流程图如图 4.12 所示,其中全面地说明了这个函数。在图中,其后带有括号的标识符与代码中的标识符是对应的。

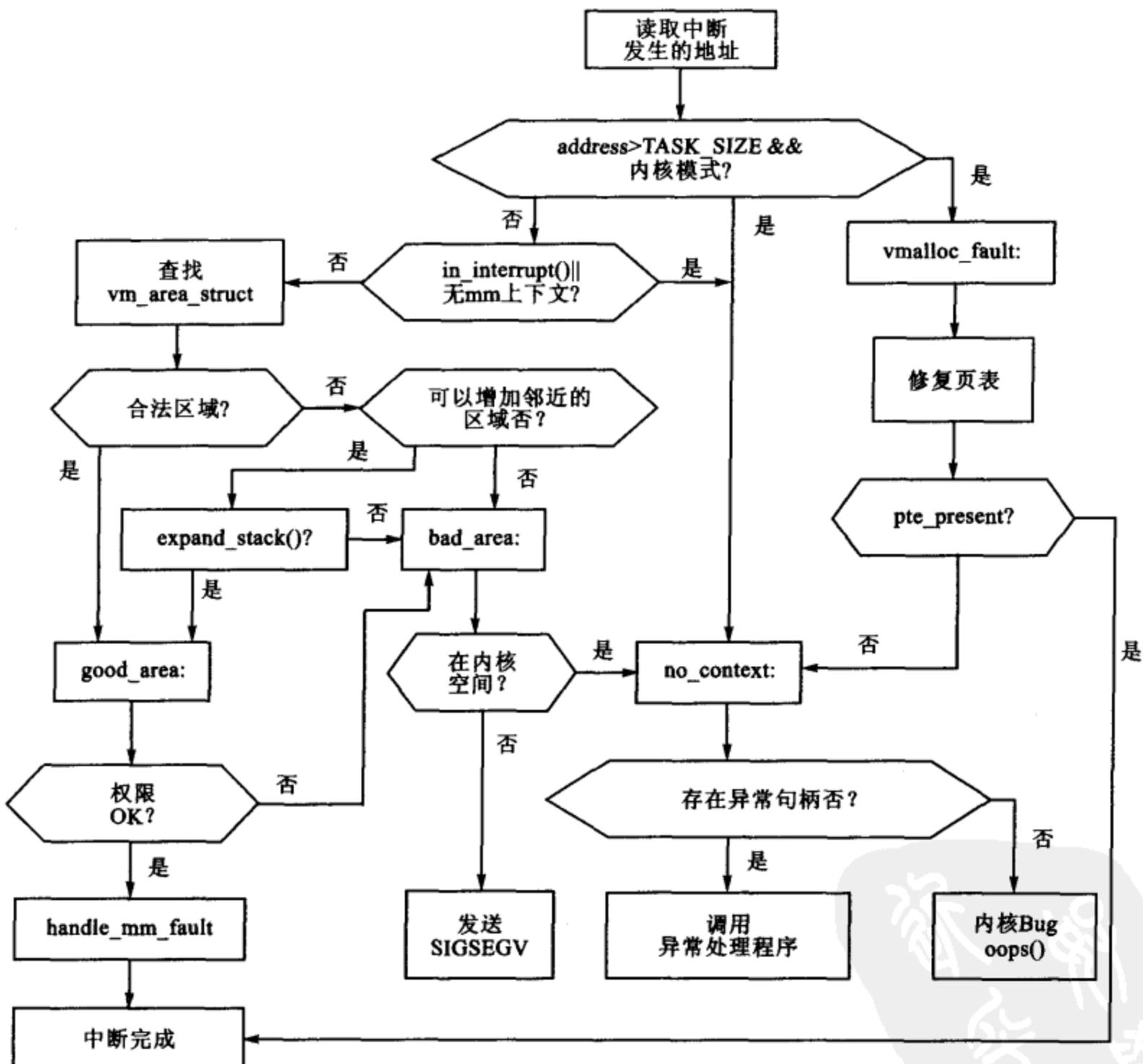


图 4.12 do_page_fault() 流程图

handle_mm_fault() 是与体系结构无关的顶层函数,负责处理后援存储器中的缺页中断,执行写时复制(COW)等。如果返回 1,就是次缺页中断,2 表示主缺页中断,0 表示发送 SIGBUS 错误,其他值就激活内存溢出处理子程序。

4.6.1 处理缺页中断

一旦异常处理程序确定异常是有效内存区域中的有效缺页中断,将会调用与体系结构无关的函数 handle_mm_fault(),调用图如图 4.13 所示。如果请求的页表项不存在,就分配请求的页表项,并调用 handle_pte_fault()。

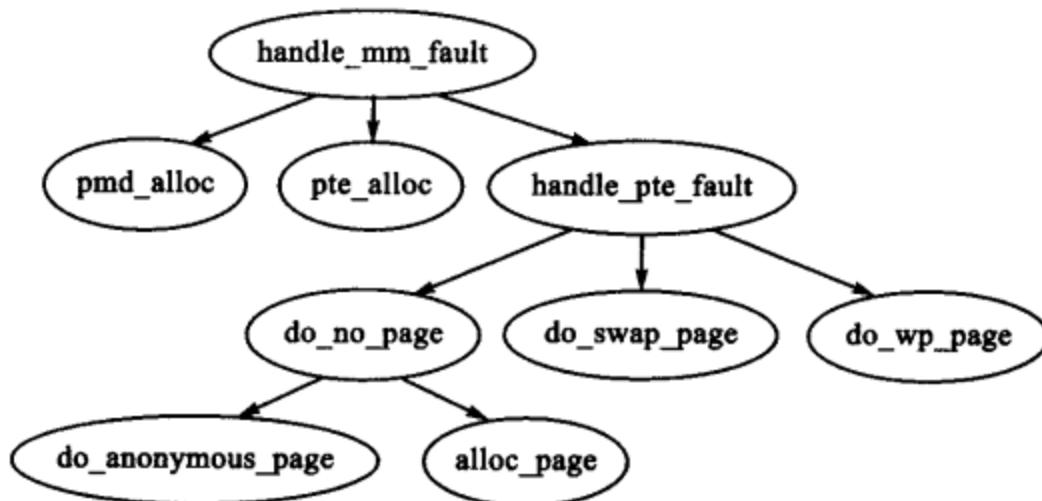


图 4.13 调用图: handle_mm_fault()

图 4.13 中所示的是用到的一个基于 PTE 属性的函数。第 1 步调用 pte_present() 检查 PTE 的标志位,确定其是否在内存中,然后调用 pte_none() 检查 PTE 是否分配。如果 PTE 还没有分配的话(pte_none() 返回 true),将调用 do_no_page() 处理请求页面的分配,否则说明该页面已经被交换到了磁盘,于是调用 do_swap_page() 处理请求换页。有一个意外情况就是在换出的页面属于虚拟文件时将由 do_no_page() 来处理。这种特殊情况将在 12.4 节讨论。

第 2 步确定是否写页面。如果 PTE 写保护,就调用 do_wp_page(),因为这个页面是写时复制(COW)页面。写时复制页面是指多个进程共享一个页面(通常是父子进程),直到其中一个进程对该页面进行写操作时才为其分配并复制一个单独的页面。写时复制页面的识别方法是,页面所在区域的 VMA 标志位为可写,但是相应的 PTE 却不是可写的。如果不是写时复制页面,通常将其标志为脏,因为它已经被写过了。

第 3 步确定页面是否已经读取以及是否在内存中,但还会发生异常。这是因为在某些体系结构中没有 3 级页表,在这种情况下建立 PTE 并标志为新即可。

4.6.2 请求页面分配

在进程第一次访问页面时,首先要分配页面,一般由函数 do_no_page() 填充数据。如果父 VMA 的 vm_ops 提供了 nopage() 函数,则调用它填充数据。这对视频卡之类的内存映射

设备非常重要,因为它们需要分配特殊的页面来支持数据访问,或者必须从辅存中读取数据的内存映射文件。我们先讨论产生缺页异常的页面是匿名页面这种最简单的情况。

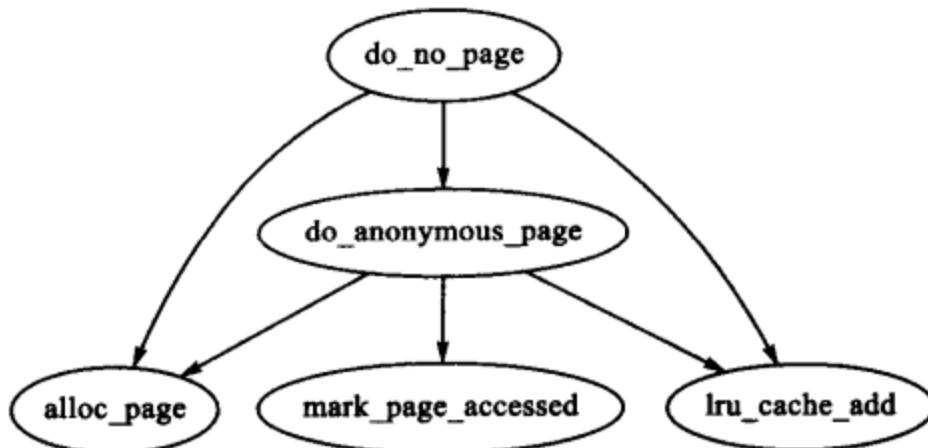


图 4.14 调用图: do_no_page()

处理匿名页面

如果 `vm_area_struct->vm_ops` 字段没有被填充或者没有提供 `nopage()` 函数, 则调用 `do_anonymous_page()` 处理匿名访问。只有两种处理方式, 第一次读和第一次写。由于是匿名页面, 第一次读很简单, 因为不存在数据, 所以系统一般使用映射 PTE 的全零页 `empty_zero_page`, 并且 PTE 是写保护的, 所以如果进程要写页面就发生另一个缺页中断。在 x86 中, 函数 `mem_init()` 负责把全局零页面归零。

如果是第一次写页面, 就调用 `alloc_page()` 分配一个由 `clear_user_highpage()` 用零填充的空闲页(见第 7 章)。假定成功地分配了这样一个页面, `mm_struct` 中的 Resident Set Size (RSS) 字段将递增; 在一些体系结构中, 为保证高速缓存的一致性, 当一个页面插入进程空间时要调用 `flush_page_to_ram()`。然后页面插入到 LRU 链表中, 以后就可以被页面回收代码处理。最后需要更新进程的页表项来反映新映射。

处理文件/设备映射页

如果被文件或设备映射, VMA 中的 `vm_operation_struct` 将提供 `nopage()` 函数。如果是文件映射, 函数 `filemap_nopage()` 将替代 `nopage()` 分配一个页面并从磁盘中读取一个页面大小的数据。如果页面由虚文件映射而来, 就使用函数 `shmem_nopage()`(见第 12 章)。每种设备驱动程序将提供不同的 `nopage()` 函数, 内部如何实现对我们来说并不重要, 只要知道该函数返回一个可用的 `struct page` 即可。

在返回页面时, 要先做检查以确定分配是否成功, 如果不成功就返回相应的错误。然后检查提前 COW 失效是否发生。如果是向页面写, 而在受管 VMA 中没有包括 `VM_SHARED` 标志, 就会发生提前 COW 失效。提前 COW 失效是指分配一个新页面, 在减少 `nopage()` 返回页面的引用计数前就将数据交叉地复制过来。

无论哪一种情况,都将利用 pte_none()进行检查,确保将被使用的 PTE 不在页表中。在 SMP 中可能会发生这样的情况,两个异常几乎在同一个页面同时发生,而在整个异常期间自旋锁没有完全被异常获得,这时必须立即做这种检查。如果没有竞争条件,就给 PTE 赋值,更新统计数据,并为高速缓存的一致性调用体系结构相关的钩子函数。

4.6.3 请求换页

在将页面交换至后援存储器后,函数 do_swap_page()负责将页面读入内存,该函数的调用图如图 4.15 所示。通过 PTE 的信息就足够查找到交换的页面。由于页面可为多个进程所共享,所以一般不能立即交换出去。将页面交换出去时,页面先放到交换高速缓存中。

共享页面不能立即交换出去,因为无法将 struct page 映射至每个进程共享的 PTE 中。而查找所有进程的页表过于浪费时间。值得注意的是 2.5.x 的后期版本和定制的 2.4.x 的补丁中包含有反向映射(RMAP),通过 RMAP,页面所映射的 PTE 组成了一个链表,可以方便地反向查找。

由于存在交换高速缓存,就有可能在发生缺页中断时,该页面仍然在交换高速缓存中,这时只需要简单地增加页面计数,然后把它放到进程页表中并计数次缺页中断发生的次数。

如果页面仅存在于磁盘中,Linux 将调用 swapin_readahead()读取它以及它后续的若干页面。读取的页面数量由 mm/swap.c 中定义的变量 page_cluster 决定。对于内存小于 16 MB 的机器,这个值初始化为 2 或 3。除非碰到坏或空的交换表项,通常读取的数量是 $2^{\text{page_cluster}}$ 。寻道操作很费时,所以一旦查找成功,应该将后续的页面一起读入内存。

4.6.4 写时复制(COW)页

以前在创建一个进程时,把父进程地址空间完全复制给子进程。这是非常费时的操作,因为可能有很大一部分进程将要从后援存储器交换入内存。为了避免大量的开销,Linux 采用了写时复制(COW)技术。

在创建的过程中,将两个进程的 PTE 设置为只读,当要写时就会导致缺页中断。Linux 能够识别 COW 页,即 PTE 是写保护的,但是 VMA 所在的区域却是可写的。Linux 利用函数 do_wp_page()将生成的复制页赋值给写进程。如有必要,将为页面保留一个新的交换插槽。采用这种技术,在创建子进程时,只需复制页表项。do_wp_page()函数的调用图如图 4.16 所示。

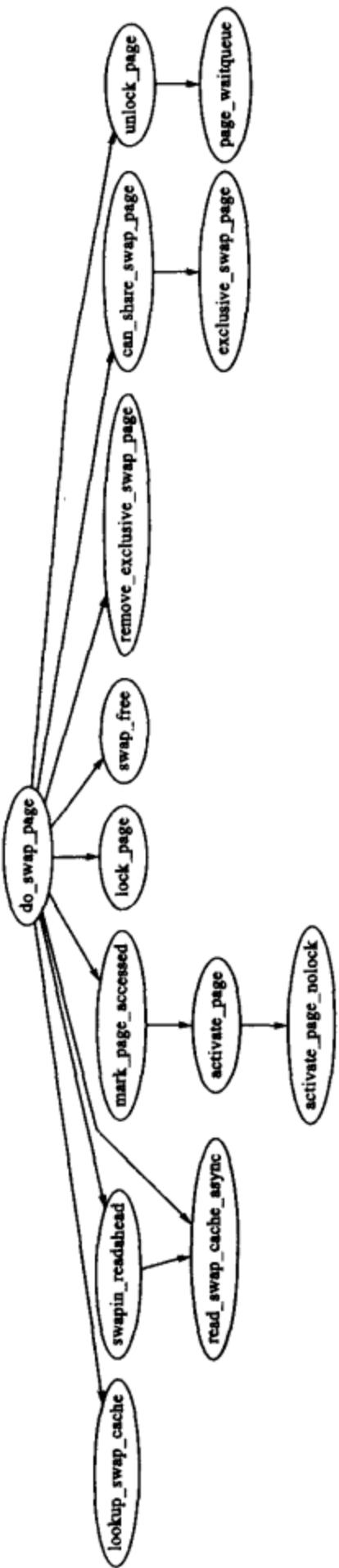


图 4.15 调用图：do_swap_page()

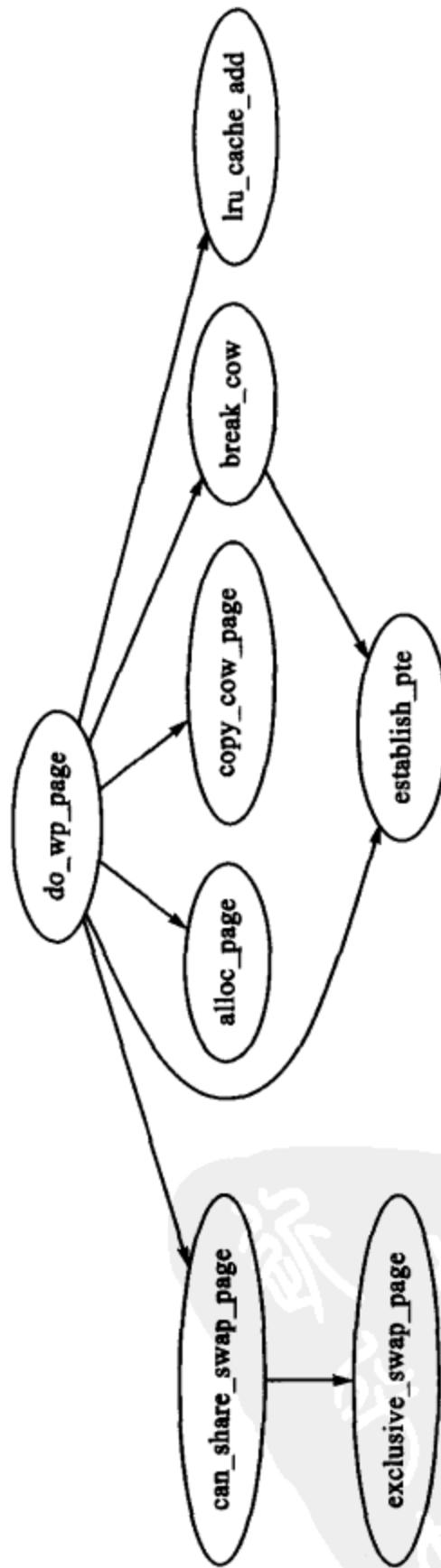


图 4.16 调用图：do_wp_page()

4.7 复制到用户空间/从用户空间复制

直接访问进程地址空间中的内存是不安全的,因为无法快速检测所访问的页面是否常驻内存。在 Linux 中,当地址无效时,MMU 会抛出缺页中断,缺页中断处理子程序捕获并处理该异常。在 x86 中,由 `__copy_user()` 提供的汇编器捕获无效地址的异常,调用 `search_exception_table()` 确定修整代码的位置。Linux 提供了大量的 API(主要是宏)用于将数据安全地复制到用户空间以及从用户空间复制数据,如表 4.6 所列。

表 4.6 访问进程地址空间 API

| |
|--|
| unsigned long <code>copy_from_user(void * to, const void * from, unsigned long n)</code> 从用户地址复制 <i>n</i> 个字节到内核地址空间 |
| unsigned long <code>copy_to_user(void * to, const void * from, unsigned long n)</code> 从内核地址复制 <i>n</i> 个字节到用户地址空间 |
| void <code>copy_user_page(void * to, void * from, unsigned long address)</code> 复制数据到用户空间的一个匿名页面或 COW 页面。端口应该避免 D-cache 别名。可以通过使用一个内核虚拟地址来达到这一目的,因为这样可以使用同一个高速缓存行作为虚拟地址 |
| void <code>clear_user_page(void * page, unsigned long address)</code> 类似 <code>copy_user_page()</code> ,但是它是为清零一个页面而设的 |
| void <code>get_user(void * to, void * from)</code> 从用户空间复制一个整型值到内核空间 |
| void <code>put_user(void * from, void * to)</code> 从内核空间复制一个整型值到用户空间 |
| long <code>strncpy_from_user(char * dst, const char * src, long count)</code> 从用户空间复制一个在最长的字节末尾的 NULL 终结字符串到内核空间 |
| long <code>strlen_user(const char * s, long n)</code> 返回包含终结符 NULL 在内的用户空间字符串的长度,最大值为 <i>n</i> |
| int <code>access_ok(int type, unsigned long addr, unsigned long size)</code> 如果用户空间的内存块合法或者为 0 时返回非 0 |

所有的宏实现都很相似,所以我们仅讨论 x86 上 `copy_from_user()` 如何实现。

`copy_from_user()` 不是调用 `__constant_copy_from_user()`,就是调用 `_generic_copy_from_user()`,这取决于编译时要复制的数据量是否确定。如果在编译时知道要复制的字节大小,

对于复制 1、2、4 个字节均有不同的优化,否则,这两个复制函数没什么很大的不同。

通用的复制函数最终调用<asm-i386/uaccess.h>里的__copy_user_zeroing(),它有 3 个重要的部分。第 1 部分汇编器,它计算实际从用户空间中复制的字节数,如果页面不在内存中就发出缺页中断,地址有效时就进行平常的换入操作。第 2 部分是修整代码。第 3 部分是__ex_table,它把第 1 部分中的指令映射到第 2 部分中的修整代码。

如同 4.5 节中描述的,执行点和修整代码子程序通过连接器复制到内核异常处理表中。如果读到无效地址,将执行函数 do_page_fault(),进而调用 search_exception_table() 查找 EIP,读取异常发生的地方,并跳到修整代码处,修整代码将 0 复制至保留的内核空间,修整寄存器并返回。以这种方式内核可以很小的检查代价安全地访问用户空间,并使 MMU 处理异常。

所有其他访问用户空间的函数都遵循这样一种类似的模式。

4.8 2.6 中有哪些新特性

线性地址空间

线性地址空间基本上保持了与 2.4 版本中相同的内容,几乎没有什么可以容易识别的变更。主要的变化是在用户空间增加一个新的页面以映射到固定的虚拟地址。在 x86 上,该页面位于 0xFFFFF000 处称为 vsyscall 页。该页中的代码提供了从用户空间进入内核空间最理想的方法。一个用户空间的程序现在可以通过调用 0xFFFFF000 来代替传统的 int 0x80 中断进入内核空间。

struct mm_struct 这个结构没有很重大的变化。首先的变化是结构中新增加了一个 free_area_cache 字段,该字段初始化为 TASK_UNMAPPED_BASE。这个字段用于标记第一个空洞在线性地址空间中的位置,以改善搜索的时间开销。在该结构的尾部新增加了少量的字段,这与内核转储有关,但已经超出本书的范围。

struct vm_area_struct 这个结构也没有很重大的变化。主要的区别是 vm_next_share 和 vm_pprev_share 两个字段中的特定链表被称为 shared 的新字段所替换。vm_raend 字段已被彻底地删除,因为文件预读在 2.6 内核中实现起来非常难。预读主要由储存在 struct file→f_ra 中的一个 struct file_ra_state 管理。如何实现预读的许多细节在 mm/readahead.c 文件中有描述。

struct address_space 第一种变化相比之下是较次要的。gfp_mask 字段被 flags 字段取代,该标志字段首部__GFP_BITS_SHIFT 个位用作 gfp_mask,并且由 mapping_gfp_mask() 函数访问。余下的多个位用于存储异步 I/O 的状态。这两个标志位可能被设置为 AS_EIO,以表明是一个 I/O 错误,或者被设置为 AS_ENOSPC,以表示在异步写期间文件系统空间已耗尽。

该结构增加了较多的东西,它们主要与页面高速缓存和预读有关。由于这些字段十分独

特,所以我们将会详细地介绍它们。

`page_tree`: 这个字段是通过索引映射的页面高速缓存所有页面的基树,所有的数据都位于物理磁盘的块上。在 2.4 内核中,搜索页面高速缓存需要遍历整个链表。而在 2.6 内核中,它是一种基树查找,可以减少相当多的搜索时间。这个基树的实现在 `lib/radix-tree.c` 文件中。

`page_lock`: 这个是保护页树的自旋锁。

`io_pages`: 在写出脏页以及调用 `do_writepages()` 函数前,这些脏页被添加到这个链表中。正如前面注释所解释的一样, `mpage_writepages()` 函数在 `fs/mpage.c` 文件中,被写出的页放在这个链表中,以避免因为锁住一个已经被 I/O 锁住的页面而造成死锁。

`dirtied_when`: 这个字段记录一个索引节点第一次变脏瞬间的时间点。这个字段决定索引节点在 `super_block->s_dirty` 链表上的位置。这样就防止了经常变脏的索引节点仍然逗留在链表的首部而导致在其他一些索引节点上出现不能写出而饿死(starving)的情况。

`backing_dev_info`: 这个字段记录预读相关的信息。该结构在 `include/linux/backing-dev.h` 中有声明,而且还有注释来解释这些字段的作用。

`private_list`: 这是一个可用的针对 `address_space` 的私有链表。如果已经使用了辅助函数 `mark_buffer_dirty_inode()` 和 `sync_mapping_buffers()`,则该链表就通过 `buffer_head->b_assoc_buffers` 字段链接到缓冲区的首部。

`private_lock`: 这是一个可用的针对 `address_space` 的自旋锁。该锁的使用情况虽然是令人费解的,但是在针对 2.5.17 (lwn.net/2002/0523/a/2.5.17.php3) 版本的冗长变更日志中解释了它的一部分使用情况。它主要保护在这个映射中共享缓冲区的其他映射中的相关信息。该锁虽然不保护 `private_list` 这个字段,但是它保护该映射中其他 `address_space` 共享缓冲区的 `private_list` 字段。

`assoc_mapping`: 映射 `private_list` 链表包含的 `address_space` 中的后援缓冲区。

`truncate_count`: 这是在一个区域由函数 `invalidate_mmap_range()` 收缩时的一个增量。当发生缺页中断时通过函数 `do_no_page()` 检查该计数器,以确保在一个页面没有错误时该计数器无效。

`struct address_space_operations`: 大多数关于该结构的变更初看起来,似乎相当简单,但实际上非常棘手。以下是作了变更的字段。

`writepage`: 函数 `writepage()` 的回调函数已经改变,另外增加了一个参数 `struct writeback_control`。这个结构负责记录一些回写信息,如是否阻塞了或者页面分配器对于写操作是否是直接回收的或者 `kupdated` 的,并且它包含一个备份 `backing_dev_info` 的句柄以控制预读。

`writepages`: 这个字段在把所有的页面写出之前,将所有的页从 `dirty_pages` 转移到 `io_pages` 中。

`set_page_dirty`: 这是一个与 `address_space` 相关的设置某页为脏的方法。主要是供后援存储器的 `address_space_operations` 使用,以及那些与脏页相关联却没有缓冲区的匿名共享页

面所使用。

readpages: 这个字段用于页面读取,使预读能够得到正确的控制。

bmap: 这个字段已经变更为处理磁盘扇区而不是设备的大于 2^{32} 个字节的无符号长整型。

invalidatepage: 这是一种更名的变化。函数 `block_flushpage()` 和回调函数 `flushpage()` 已经被重命名为 `block_invalidatepage()` 和 `invalidatepage()`。

direct I/O: 已经改变成用 2.6 版本中新的 I/O 机制。新的机制已经超出了本书的范围。

内存区域

函数 `mmap()` 的操作有两个重要的改变。首先安全模块的回调是可能的了。该回调函数调用 `security_file_mmap()`, 用于访问 `security_ops` 结构相关的函数。在缺省情况下, 它是一个空操作。其次是较严格的地址空间计数的代码添加了进来。被计数的 `vm_area_structs` 结构都会设置 `VM_ACCOUNT` 标志位, 这些都是在用户空间的映射。当用户空间区域被创建或者被销毁时, 函数 `vm_acct_memory()` 和 `vm_unacct_memory()` 会更新变量 `vm_committed_space`。这给了内核一个更直观的视图, 即有多少内存被提交到用户空间中去了。

4 GB/4 GB 用户/内核分离

2.4.x 内核存在的一种限制, 即内核仅仅有 1 GB 可供使用的虚拟地址空间, 并且对所有进程是可见的。在写该书的时候, Ingo Molnar 开发了一种补丁技术允许内核可以有选择地拥有完全属于自己的 4 GB 地址空间。<http://redhat.com/~mingo/4g-patches/> 中提供了该补丁, 包含在-mm 测试树中, 但是并不清楚它是否将被合并到主流内核中。

这个特性旨在使 32 位系统拥有超大(>16 GB)的 RAM。传统的 3/1 分裂完全支持 1 GB 的 RAM。在这之后, 允许更大数量的内存通过临时映射高端内存页面。然而, 由于使用了更多的 RAM, 却形成一个较大的瓶颈。如当物理 RAM 的数量接近 60 GB 的范围时, 几乎所有的低端内存都用于 `mem_map`。而通过让内核拥有自己 4 GB 的虚拟地址空间, 就会比较容易地支持如此巨大的内存。尽管这已经非常严重, 而每次的系统调用所产生的 TLB 刷新则会更严重地影响到性能。

有了这个补丁, 在用户空间和内核空间之间被共享的仅仅是 16 MB 的内存, 而这被用来存储全局描述表(GDT)、中断描述表(IDT)、任务状态段(TSS)、局部描述表(LDT)、`vsyscall` 页面和内核堆栈。在页表之间作实际切换工作的代码随后被包含在进入/退出内核空间的弹性代码中。内核中也作了一些变更, 诸如删除了直接访问用户空间缓冲区的指针, 但是, 大体上, 内核的核心部分没有受到这个补丁的影响。

非线性 VMA 生成形式

在 2.4 内核中, 具有后援文件的 VMA 以一种线性的方式生成。在 2.6 中可以有选择地改变, 通过引入 `MAP_POPULATE` 标志位给 `mmap()` 函数和新的系统调用函数 `remap_file_`

pages(),它们都由 sys_remap_file_pages()函数实现。该系统调用通过操作页表,允许将任意一个已经在虚拟内存区域中的页面重新映射到后援文件的任意位置中去。

在页面换出时,文件的非线性地址用 PTE 进行了编码,这样在缺页中断时它就能被正确地重新安装。由于它的实现如何被编码是体系结构相关的问题,为了达到这个目的,定义了两个宏 pgoff_to_pte()和 pte_to_pgoff()。

这个特性在一个具有大量映射的应用中有很多好处,例如数据库服务器和虚拟应用如模拟器。之所以介绍这个特性有很多原因。第一,VMA 是针对单一进程的,并且有非常大的空间需求,尤其是在有大量映射的应用中。第二,搜索函数 get_unmapped_area()为了找到空闲的虚拟地址空间采用线性搜索,该方法会由于大量的映射而开销甚大。第三,非线性映射将预先导致大多数的页错误存在于内存中,反之,一般的映射会导致每一个页面都有一个主要错误。但这是可以避免的,只要使用 mmap()函数中新的 MAP_POPULATE 标志位或者使用 mlock()函数就可以达到。最后一个原因是为了避免稀疏映射,在最坏的情况下,每一个文件页面的映射都将需要一个 VMA。

然而,这个特性有一些严重的缺点。首先是,系统调用函数 truncate()和 mincore()破坏了非线性映射。这两个系统调用都依赖于 vm_area_struct→vm_pgoff,对非线性来说是毫无意义的映射。如果一个非线性映射的文件被截断,这些被截断的页还会继续在 VMA 之内保留下来。已有人建议,正确的解决方案是将这些页仍然留在内存中,但要使这些页面匿名。在写这本书的时候,还没有解决方案实现。

第二种主要的弊端是 TLB 无效。每一个重新映射的页面都会要求通过函数 flush_icache_page()通知 MMU 发生了重映射,但是更重要的是调用 flush_tlb_page()函数带来了副作用。一些处理器会使得关联到这种页面的 TLB 项失效,但是也有其他的处理器通过刷新整个 TLB 来实现这一工作。如果重映射比较频繁,由于 TLB 未命中次数的增加和过度的不断进入内核空间,而使得性能降低。从某些方面来讲,这些副作用是最为恶劣的,因为这些影响和处理器联系紧密。

现在仍然不清楚这个特性将来会怎样,如果保留,也就这么用吧。

在写这篇文章时,依旧正在争论如何修正这个特性,但是非线性映射很可能得到不同的处理,以区别于普通映射中的页面换出,截断和反向映射处理。由于这个特性的主要使用者很可能是数据库,那么这一特殊的处理就有可能不是问题了。

页面错误

页面错误规则的改变和其他变化比较起来更具随意性,不像支持高端内存的反向映射和 PTE 的变化那么有必要。主要的变化是缺页中断规则中返回的是自解释的编译时间定义而不是魔数(magic number)。函数 handle_mm_fault()可能的返回值有 VM_FAULT_MINOR, VM_FAULT_MAJOR, VM_FAULT_SIGBUS 和 VM_FAULT_OOM。

第 5 章

引导内存分配器

由于硬件配置多种多样,所以在编译时就静态初始化所有的内核存储结构是不现实的。下一章将讨论到物理页面的分配,即使是确定基本数据结构也需要分配一定的内存空间来完成其自身的初始化过程。但物理页面分配器如何分配内存去完成自身的初始化呢?

为了说明这一点,我们使用一种特殊的分配器——引导内存分配器(boot memory allocator)。它基于大多数分配器的原理,以位图代替原来的空闲块链表结构来表示存储空间[Tan01]。若位图中某一位为 1,表示该页面已经被分配;否则表示为被占有。该分配机制通过记录上一次分配的页面帧号(PFN)以及结束时的偏移量来实现分配大小小于一页的空间。连续的小的空闲空间将被合并存储在同一页上。

读者也许会问,当系统运行时为何不使用这种分配机制呢?其中的一个关键原因在于:首次适应分配机制虽然不会受到碎片的严重影响[JW98],但是它每次都需要通过对内存进行线性搜索来实现分配。若它检查的是位图,其成本将是相当高的。尤其是首次适应算法容易在内存的起始端留下许多小的空闲碎片,在需要分配较大的空间时,检查位图这一过程就显得代价很高[WJNB95]。

在该分配机制中,存在着两种相似但又不同的 API。一种是如表 5.1 所列的 UMA 结构,另一种是如表 5.2 所列的 NUMA 结构。两者主要区别在于:NUMA API 必须附带对节点的操作,但由于 API 函数的调用者来自于与体系结构相关的层中,所以这不是一个很大的问题。

本章首先描述内存分配机制的结构,该结构用于记录每一个节点的可用物理内存。然后举例阐述如何设定物理内存的界定和每一个管理区的大小。接下来讨论如何使用这些信息初始化引导内存分配机制的结构。在解决了引导内存分配机制不再被使用后,我们开始研究其中的一些分配算法和函数。

表 5.1 针对 UMA 体系结构的引导内存分配器 API

| | |
|---|--|
| unsigned long init_bootmem(unsigned long start, unsigned long page) | 初始化内存为 0 到 PFN 页面之间的值。可用内存的起始位置在 PFN 的起始处 |
| void reserve_bootmem(unsigned long addr, unsigned long size) | 标记页面在地址 addr 和预留的 addr+size 之间的请求部分,保留页面的某个部分将导致整个页面都被保留 |
| void free_bootmem(unsigned long addr, unsigned long size) | 标记在地址 addr 和 addr+size 之间的页面空闲 |
| void * alloc_bootmem(unsigned long size) | 从 ZONE_NORMAL 分配 size 数量的字节。该分配的地方将对齐 L1 硬件高速缓存以最充分地利用硬件高速缓存 |
| void * alloc_bootmem_low(unsigned long size) | 从 ZONE_DMA 分配 size 数量的字节。该分配的地方将对齐 L1 硬件高速缓存 |
| void * alloc_bootmem_pages(unsigned long size) | 从 ZONE_NORMAL 分配 size 数量的字节。之所以对齐为一个页面的大小是为了让所有的页面可以返回给调用者 |
| void * alloc_bootmem_low_pages(unsigned long size) | 从 ZONE_DMA 分配 size 数量的字节。之所以对齐为一个页面的大小是为了让所有的页面可以返回给调用者 |
| unsigned long bootmem_bootmap_pages(unsigned long pages) | 计算用于存储一个位图所需要的页面数量,以表明页面数量的分配状态 |
| unsigned long free_all_bootmem() | 该函数在引导分配器生命周期结束时使用。它遍历位图中的所有页面。对于每一个空闲的页面,其标志位将被清除,且页面被释放到物理页面分配器中(见下一章),因此运行时分配器可以建立自己的空闲链表 |

表 5.2 针对 NUMA 体系结构的引导内存分配器 API

| | |
|---|--|
| unsigned long init_bootmem_node(pg_data_t * pgdat, unsigned long freepfn, unsigned long startpfn, unsigned long endpfn) | 用于 NUMA 体系结构。它使用在 freepfn 中第一个可用的 PFN 来初始化在 PFN startpfn 和 endpfn 之间的内存。在初始化后,pgdat 节点将插入 pgdat_list 中 |
| void reserve_bootmem_node(pg_data_t * pgdat, unsigned long physaddr, unsigned long size) | 标记特定节点 pgdat 上地址 addr 和 addr+size 之间的页面保留。请求部分保留页面的某个部分将导致这个页面被保留 |
| void free_bootmem_node(pg_data_t * pgdat, unsigned long physaddr, unsigned long size) | 标记特定节点 pgdat 上地址 addr 和 addr+size 之间的页面空闲 |

续表 5.2

| | |
|--|---|
| void * alloc_bootmem_node(pg_data_t * pgdat, unsigned long size) | 在特定节点 pgdat 上从 ZONE_NORMAL 分配 size 数量的字节。这个分配的地方将对齐 L1 硬件高速缓存, 以最充分地利用硬件高速缓存 |
| void * alloc_bootmem_pages_node(pg_data_t * pgdat, unsigned long size) | 在特定节点 pgdat 上从 ZONE_DMA 分配 size 数量的字节, 以使所有的页面可以返回给调用者 |
| void * alloc_bootmem_low_pages_node(pg_data_t * pgdat, unsigned long size) | 在特定节点 pgdat 上从 ZONE_DMA 分配和对齐一个页面大小的 size 数量的字节, 以使所有的页面可以返回给调用者 |
| unsigned long free_all_bootmem_node(pg_data_t * pgdat) | 该函数在引导分配器生命周期结束时使用。它遍历位图中的所有页面。对于每一个空闲的页面, 其标志位将被清除, 且页面被释放到物理页面分配器中(见下一章), 因此运行时分配器可以建立自己的空闲链表 |

5.1 表示引导内存映射

系统内存中的每一个节点都存在一个 bootmem_data 结构。它含有引导内存分配器给节点分配内存时所需的信息, 如表示已分配页面的位图以及分配地点等信息。它在<linux/bootmem.h>文件中定义如下:

```

25 typedef struct bootmem_data {
26     unsigned long node_boot_start;
27     unsigned long node_low_pfn;
28     void * node_bootmem_map;
29     unsigned long last_offset;
30     unsigned long last_pos;
31 } bootmem_data_t;

```

该结构各个字段如下。

node_boot_start: 表示块的起始物理地址。

node_low_pfn: 表示块的结束地址, 或就是该节点表示的 ZONE_NORMAL 的结束。

node_bootmem_map: 以位表示的已分配和空闲页面的位图的地址。

last_offset: 最后一次分配所在页面的偏移。如果为 0, 则表示该页全部使用。

last_pos: 最后一次分配时的页面帧数。使用 last_offset 字段, 我们可以检测在内存分配时, 是否可以合并上次分配所使用的页, 而不用重新分配新页。

5.2 初始化引导内存分配器

每一种体系结构中都提供了一个 `setup_arch()` 函数, 用于获取初始化引导内存分配器时所必须的参数信息。

各种体系结构都有其函数来获取这些信息。在 x86 体系结构中为 `setup_memory()`, 而在其他体系结构中, 如在 MIPS 或 Sparc 中为 `bootmem_init()`, PPC 中为 `do_init_bootmem()`。除体系结构不同外各任务本质上是相同的。参数信息如下。

`min_low_pfn`: 系统中可用的最小 PFN。

`max_low_pfn`: 以低端内存区域表示的最大 PFN。

`highstart_pfn`: 高端内存区域的起始 PFN。

`highend_pfn`: 高端内存区域的最后一个 PFN。

`max_pfn`: 表示系统中可用的最大 PFN。

5.3 初始化 bootmem_data

一旦 `setup_memory()` 确定了可用物理页面的界限, 系统将从两个引导内存的初始化函数中选择一个, 并以待初始化的节点的起始和终止 PFN 作为调用参数。在 UMA 结构中, `init_bootmem()` 用于初始化 `contig_page_data`, 而在 NUMA, `init_bootmem_node()` 则初始化一个具体的节点。这两个函数主要通过调用 `init_bootmem_core()` 来完成实际工作。

内核函数首先要把 `pgdat_data_t` 插入到 `pgdat_list` 链表中, 因为这个节点在函数末尾很快就会用到。然后它记录下该节点的起始和结束地址(该节点与 `bootmem_data_t` 有关)并且分配一个位图来表示页面的分配情况。位图所需的大小以字节计算, 计算公式如下:

$$\text{mapsize} = \frac{(\text{end_pfn} - \text{start_pfn}) + 7}{8}$$

该位图存放于由 `bootmem_data_t` \rightarrow `node_boot_start` 指向的物理地址处, 而其虚拟地址的映射由 `bootmem_data_t` \rightarrow `node_bootmem_map` 指定。由于不存在与结构无关的方式来检测内存中的空洞, 整个位图就被初始化为 1 来有效地标志所有已分配的页。将可用页面的位设置为 0 的工作则由与结构相关的代码完成, 但在实际中只有 Spare 结构使用到这个位图。在 x86 结构中, `register_bootmem_low_pages()` 通过检测 e820 映射图, 并在每一个可用页面上调用 `free_bootmem()` 函数, 将其位设为 1, 然后再调用 `reserve_bootmem()` 为保存实际位图所需的页面预留空间。

5.4 分配内存

`reserve_bootmem()` 函数用于保存调用者所需的页面, 但对于一般的页面分配而言它是相当麻烦的。在 UMA 结构中, 有四个简单的分配函数: `alloc_bootmem()`, `alloc_bootmem_low()`, `alloc_bootmem_pages()` 和 `alloc_bootmem_low_pages()`。对它们的详细描述如表 5.1 所列。这些函数都以不同的参数调用 `__alloc_bootmem()`, 如图 5.1 所示:

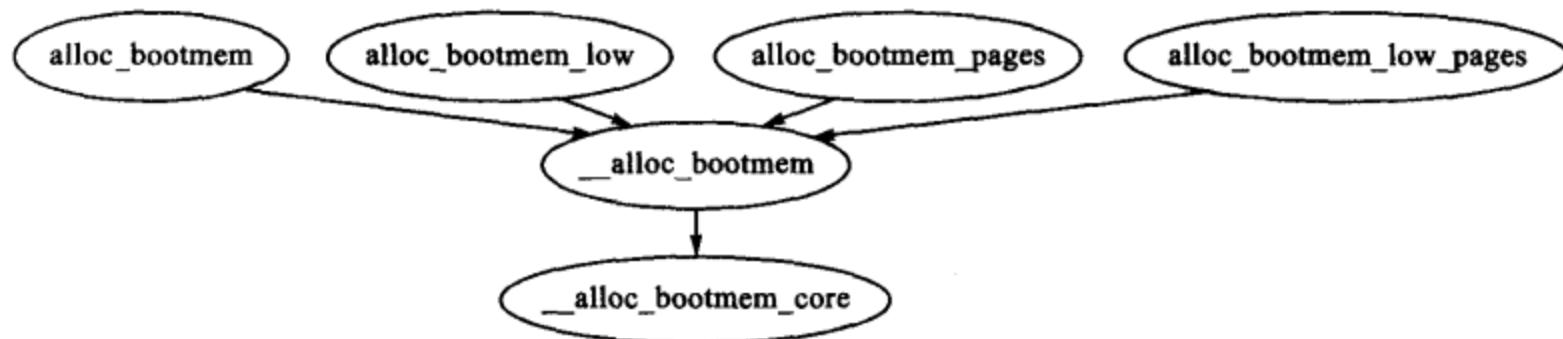


图 5.1 调用图: `alloc_bootmem()`

在 NUMA 结构中, 同样存在几个相似的函数, `alloc_bootmem_node()`, `alloc_bootmem_pages_node()` 和 `alloc_bootmem_low_pages_node()`, 只不过它们多了一个节点作为参数。同样, 它们都调用 `__alloc_bootmem_node()`, 只是参数不同。

无论是 `__alloc_bootmem()` 还是 `__alloc_bootmem_node()`, 本质上它们的参数相同。

`pdgat`: 要分配的节点。在 UMA 结构中, 它被省掉了, 其默认值是 `contig_page_data`。

`size`: 所需要分配的空间大小;

`align`: 要求对齐的字节数。如果分配空间比较小, 就以 `SMP_CACHE_BYTES` 对齐, 在 x86 结构中, 它是硬件一级高速缓存的对齐方式。

`goal`: 最佳分配的起始地址。底层函数一般从物理地址 0 开始, 其他函数则开始于 `MAX_DMA_ADDRESS`, 它是这种结构中 DMA 传输模式的最大地址。

`__alloc_bootmem_core()` 是所有 API 分配函数的核心。它是一个非常大的函数, 因为它拥有许多可能出错的小步骤。该函数从 `goal` 地址开始, 在线性范围内扫描一个足够大的内存空间以满足分配要求。对于 API 来说, 这个地址或者是 0(适合 DMA 的分配方式), 或者就是 `MAX_DMA_ADDRESS`。

该函数最巧妙、最主要的一部分在于判断新的分配空间是否能与上一个合并这一问题。满足下列条件则可合并:

- 上次分配的页与此次分配(`bootmem_date`→`pos`)所找到的页相邻;
- 上一页有一定的空闲空间, 即 `bootmem_dataoffset` != 0;

- 对齐小于 PAGE_SIZE。

不管分配是否能够合并,我们都必须更新 pos 和 offset 两个字段来表示分配的最后一页,以及该页使用了多少。如果该页完全使用,则 offset 为 0;

5.5 释放内存

与分配的函数不同,Linux 只提供了释放的函数,即用于 UMA 的 free_bootmem(),和用于 NUMA 的 free_bootmem_node()。两者都调用 free_bootmem_core(),只是 NUMA 中的不提供参数 pgdat。

相比分配器的其他函数而言,核心函数较为简单。对于受释放影响的每个完整页面的相应位被设为 0。如果原来就是 0,则调用 BUG()提示发生重复释放错误。BUG()用于由于内核故障而产生的不能消除的错误。它终止正在运行中的程序,使内核陷入循环,并打印出栈内信息和调试信息供开发者调试。

对释放函数的一个重要限制是只有完整的页面才可释放,它不会记录何时一个页面被部分分配。所以当一个页面要部分释放时,整个页面都将保留。其实这并非是一个大问题,因为分配器在整个系统周期中都驻留内存,但启动时间内,它却是对开发者的一个重要限制。

5.6 销毁引导内存分配器

启动过程的末期,系统调用函数 start_kernel(),这个函数知道此时可以安全地移除启动分配器和与之相关的所有数据结构。每种体系都要求提供 mem_init()函数,该函数负责消除启动内存分配器和与之相关的结构。

这个函数的功能相当简单。它负责计算高、低端内存的维度并向用户显示出信息消息。若有需要,还将运行硬盘的最终初始化。在 x86 平台,有关 VM 的主要函数是 free_pages_init()。

这个函数首先使引导内存分配器调用函数回收自身,UMA 结构中是调用 free_all_bootmem()函数,NUMA 中是调用 free_all_bootmem_node()函数。这两个函数都调用内核函数 free_all_bootmem_core(),但是使用的参数不同。free_all_bootmem_core()函数原理很简单,它执行如下任务。

- 对于这个节点上分配器可以识别的所有未被分配的页面:
 - 将它们结构页面上的 PG_reserved 标志位清 0;
 - 将计数器设置为 1;
 - 调用 __free_pages()以使伙伴分配器(下章将讨论)能建立释放列表。

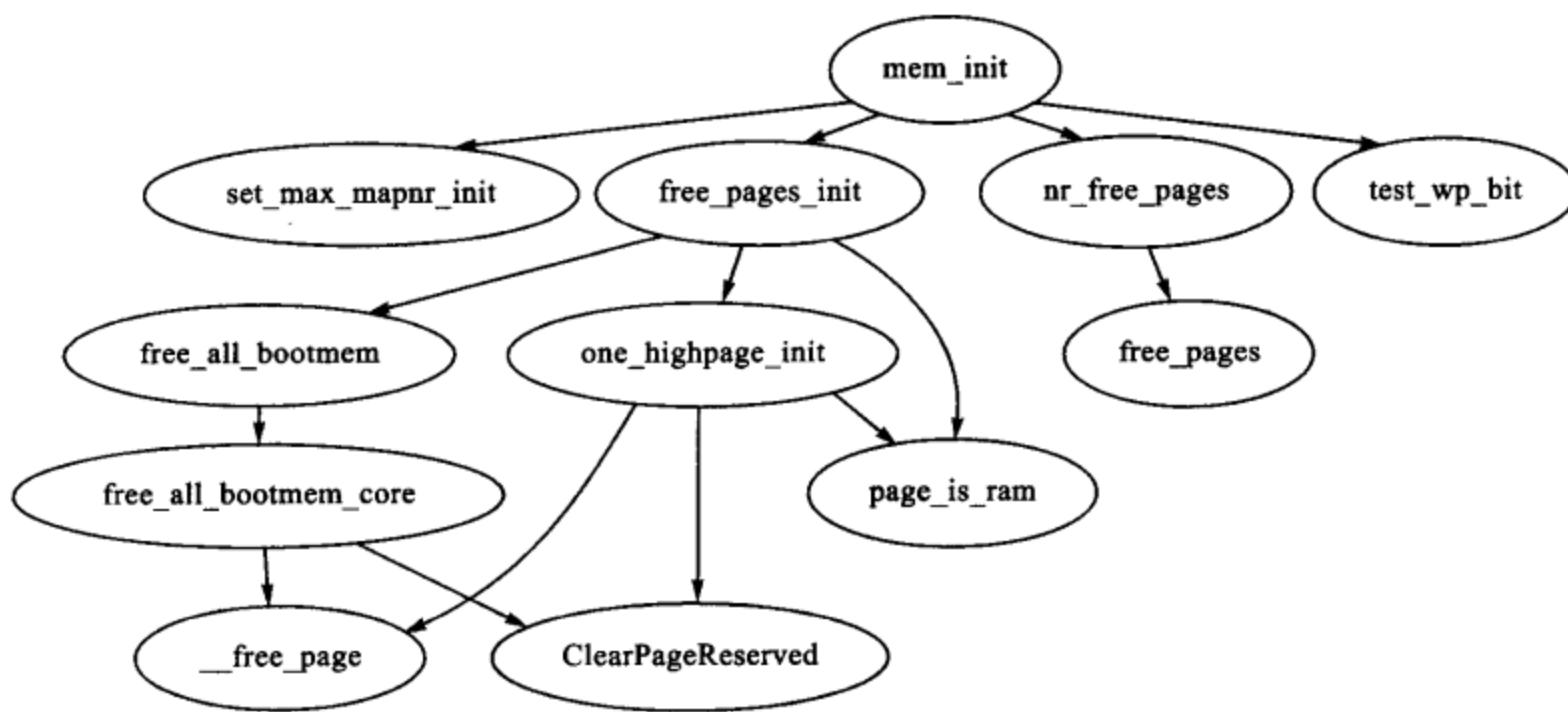


图 5.2 调用图: mem_init()

- 释放位图使用的所有页面，并将这些页面交给伙伴分配器。

在这个阶段，伙伴分配器控制了所有在低端内存下的页面。free_all_bootmem()返回后首先清算保留页面的数量。高端内存页面由 free_pages_init()负责处理。但此时需要理解的是如何分配和初始化整个 mem_map 序列，以及主分配器如何获取页面。图 5.3 显示了单节点系统中初始化低端内存页面的基本流程。free_all_bootmem()返回后，ZONE-NORMAL 中的所有页面都为伙伴分配器所有。为了初始化高端内存页面，free_pages_init()对 highstart_pfn 和 highend_pfn 之间的每一个页面都调用了函数 one_highpage_init()。此函数简单将 PG_reserved 标志位清 0，设置 PG_highmem 标志位，设置计数器为 1 并调用 _free_pages() 将自己释放到伙伴分配器中。这与 free_all_bootmem_core() 操作一样。

此时不再需要引导内存分配器，伙伴分配器成为系统的主要物理页面分配器。值得注意的是，不仅启动分配器的数据被移除，它所有用于启动系统的代码也被移除了。所有用于启动系统的初始函数声明都被标明为_init，如下所示：

```
321 unsigned long __init free_all_bootmem (void)
```

连接器将这些函数都放在 init 区。x86 平台上 free_initmem() 函数可以释放 __init_begin 到 __init_end 的所有页面给伙伴分配器。通过这种方法，Linux 能释放数量很大的启动代码所使用的内存，已不再需要启动代码。

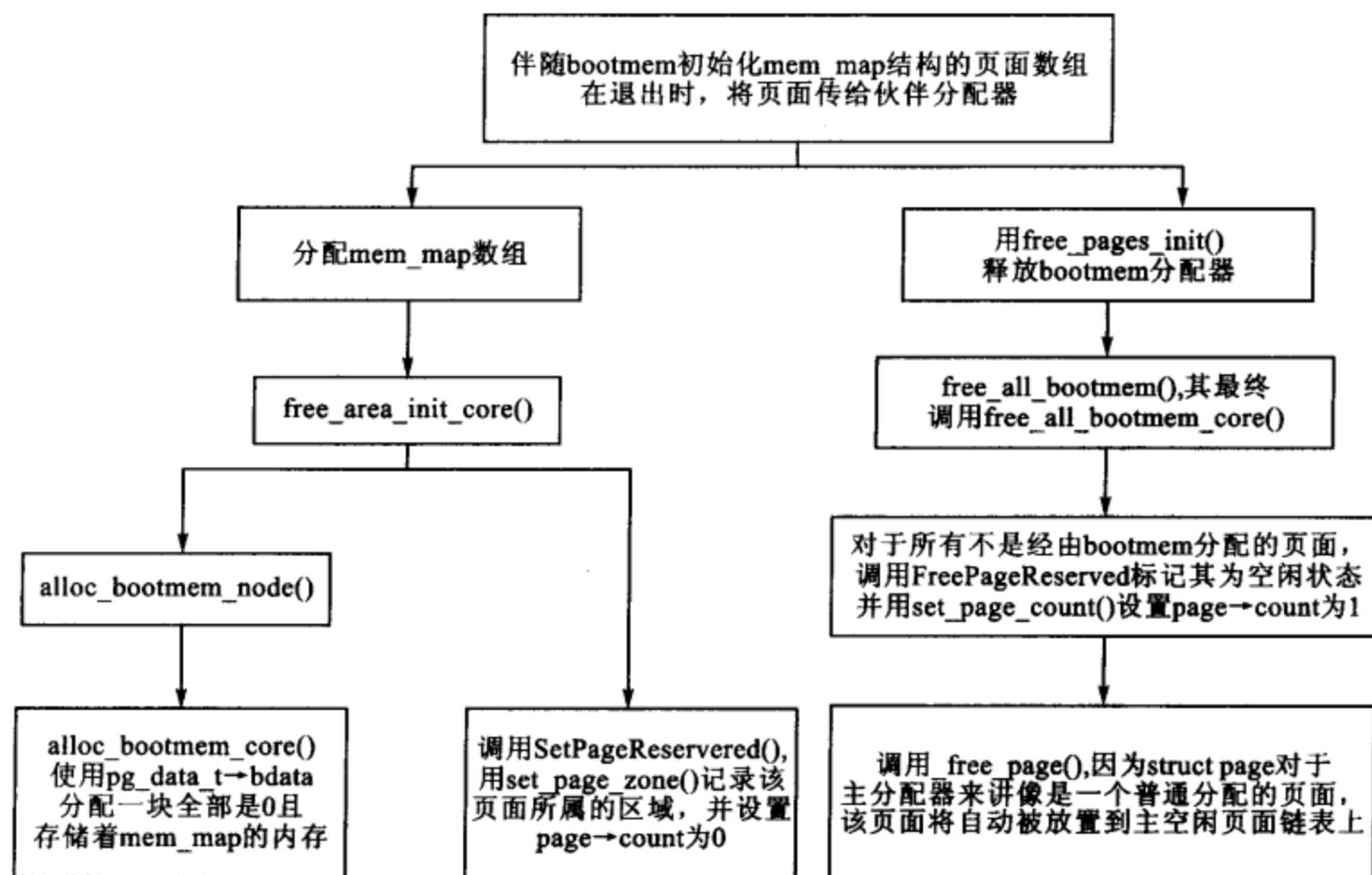


图 5.3 初始化 mem_map 和主物理页面分配器

5.7 2.6 中有哪些新特性

引导内存分配器自 2.4 内核就没有什么重要的变化，主要是涉及一些优化和一些次要的有关 NUMA 体系结构的修改。第 1 个优化是在 bootmem_data_t 结构中增加了 last_success 字段。正如名字所表达的意思，该字段记录最近的一次成功分配的位置以减少搜索次数。如果在 last_success 之前的一个地址释放，则该地址将会被改成空闲的区域。

第 2 个优化也和线性搜索有关。在搜索一个空闲页面时，2.4 内核将测试每个位，这样的开销是很大的。2.6 内核中使用测试一个长度为 BITS_PER_LONG 的块是否都是 1 来取代原来的操作。如果不是，就测试该块中单独的每个比特位。为加快线性搜索，通过函数 init_bootmem() 以节点的物理地址为序将它们排列起来。

最后的一个变更与 NUMA 体系结构及相似的体系结构相关。相似的体系结构现在定义了自己的 init_bootmem() 函数，并且任一个体系结构都可以有选择地定义它们自己的 reserve_bootmem() 函数。

第 6 章

物理页面分配

本章描述了 Linux 如何管理和分配物理内存页面。这里用的主要算法是二进制伙伴分配器,此算法由 Knowlton 设计,后来 Knuth[Knu68]又进行了更深刻的描述。与其他分配器相比,这个算法显示出了超快的速度[KB85]。

这是一个结合了 2 的方幂个分配器与空闲缓冲区合并技术的分配方案[Vah96],其基本概念非常简单。内存被分成了含有很多页面的大块,每一块都是 2 个页面大小的方幂。如果找不到想要的块,一个大块会被分成两部分,这两部分彼此就成了伙伴。其中一半被用来分配而另一半空闲。这些块会继续被二分直至产生一个所需大小的块。当一个块被最终释放时,其伙伴将被检测出来,如果它空闲则合并两者。

本章首先描述 Linux 如何记住哪些内存块是空闲的;之后将详细讨论分配空闲页面的方法;接下来说明影响分配器行为的众多标志位;最后讨论内存碎片的问题以及分配器将如何处理。

6.1 管理空闲块

如上所述,分配器维护空闲页面所组成的块,这里每一块都是 2 的方幂个页面。方幂的指数被称为阶。如图 6.1 所示,内核对于每一个阶都维护结构数组 `free_area_t`,用于指向一个空闲页面块的链表。

所以,数组的第 0 个元素将会指向具有 2^0 或 1 个页面大小的块的链表,第一个元素将会是一个具有 2^1 (2)个页面大小的块的链表,直到 $2^{\text{MAX_ORDER}-1}$ 个页面大小的块,`MAX_ORDER` 的值一般为 10。这消除了一个大页面被分开来满足一个小页面块即能满足的要求的可能性。页面块通过 `page->list` 这个线性链表来维护。

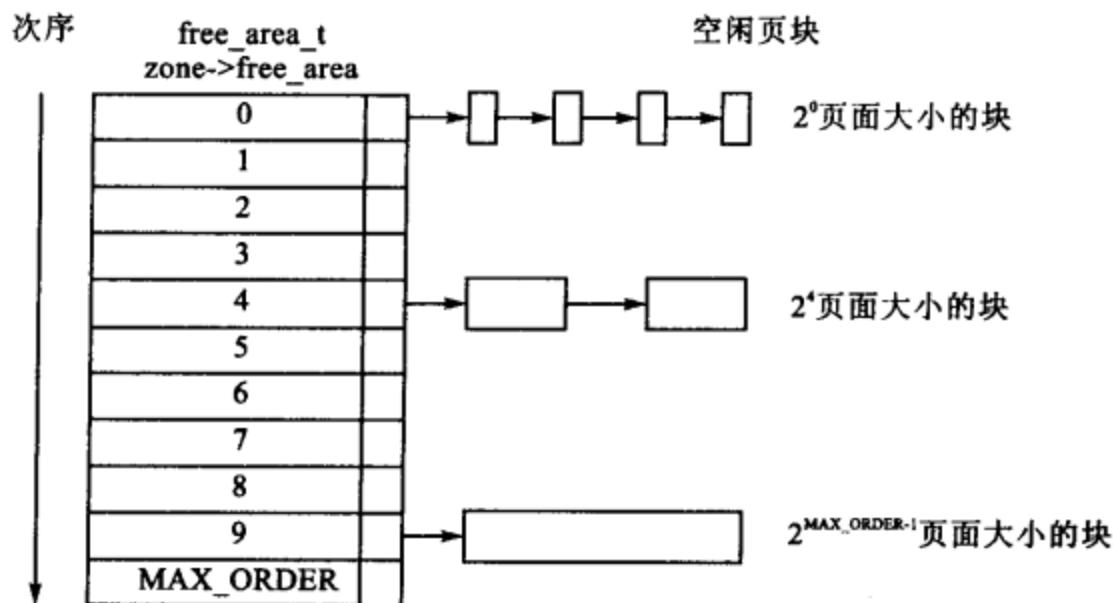


图 6.1 空闲页块管理

每一个管理区都有一个 `free_area_t` 结构数组, 即 `free_area[MAX_ORDER]`, 它在 `<linux/mm.h>` 中的定义如下所示:

```

22 typedef struct free_area_struct {
23     struct list_head free_list;
24     unsigned long * map;
25 } free_area_t;

```

此结构的字段如下:

`free_list` 空闲页面块的链表;

`map` 表示一对伙伴状态的位图。

Linux 通过使用一位而非两位来表示每一对伙伴从而节省内存。每当一个伙伴被分配出去或者被释放, 这对伙伴的位就会被切换。因此, 如果这对页面块都是空闲的或者都在使用中, 这个位就是 0; 如果仅仅有一个在使用则这个位就是 1。为了切换到正确的位, 我们使用文件 `page_alloc.c` 中的宏 `MARK_USED()`, 对它的解释如下:

```

164 #define MARK_USED(index, order, area) \
165 __change_bit((index) >> (1 + (order)), (area)->map)

```

`index` 就是全局的 `mem_map` 数组中的页面下标。将它向右移动 $1+order$ 位就可以得到代表伙伴对的映射中的位。

6.2 分配页面

Linux 提供了相当多的 API 来分配页面帧。所有的 API 都带有参数 gfp_mask, 这个参数决定分配器如何进行分配的一系列标志位。这些标志位将在 6.4 节中讨论。

如图 6.2 所示, 用作分配的 API 函数都使用核心函数 `__alloc_pages()`, 但这些 API 都独立存在, 以便选择正确的节点和管理区。不同的用户将会选择不同的管理区, 例如一些驱动程序使用 `ZONE_DMA`, 而一些磁盘缓冲区使用 `ZONE_NORMAL`, 并且调用者不需要知道使用了哪些节点。所有页面分配的 API 列表如表 6.1 所列。

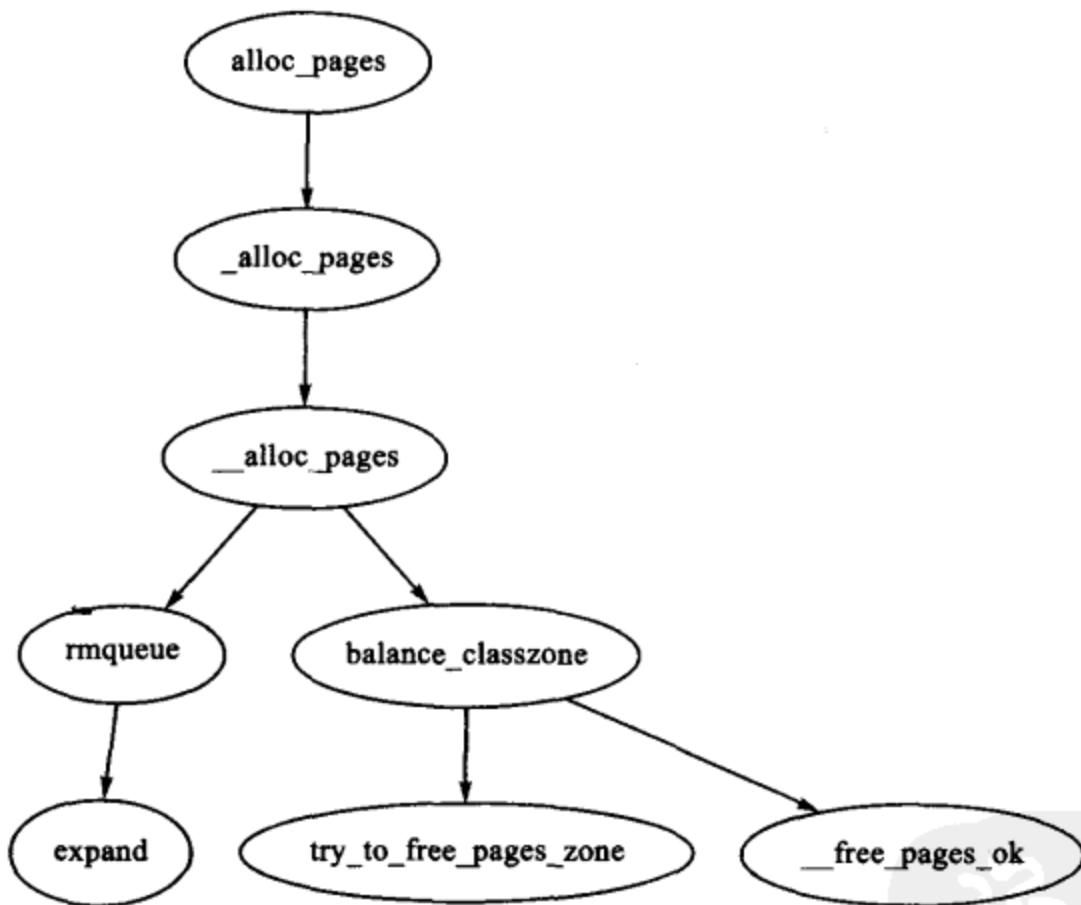


图 6.2 调用图: `alloc_pages()`

分配通常依据一个特定的次进行, 0 代表需要一个单独的页面。如果空闲块不能满足所需的层, 则一个高次的块会被分成两个伙伴, 一个用于分配, 另一个放入低次的空闲链表中。图 6.3 展示了一个块在何处被分开以及伙伴如何被加入到空闲链表直至找到一个适合进程的块。

当这个块被最终释放时, 其伙伴将会被检查。如果两个都是空闲块, 它们就会合并为一个次数较高的块并被放入高次链表。在此链表中伙伴被检测。如果伙伴并不空闲, 则被释放的页面块就会加入当前次数的空闲链表。在这些链表的操作中, 当一个进程处于不一致状态时, 必须禁止中断, 以防其他中断处理操作链表。这些措施通过使用中断安全的自旋锁来实现。

表 6.1 物理页面分配 API

| | |
|---|---|
| struct page * alloc_page(unsigned int gfp_mask) | 分配一个页面并返回一个结构地址 |
| struct page * alloc_pages(unsigned int gfp_mask, unsigned int order) | 分配 2^{order} 数量的页面并返回一个页面结构 |
| unsigned long get_free_page(unsigned int gfp_mask) | 分配一个页面,对其赋 0 值,并返回一个虚拟地址 |
| unsigned long __get_free_page(unsigned int gfp_mask) | 分配一个页面并返回一个虚拟地址 |
| unsigned long __get_free_pages(unsigned int gfp_mask, unsigned int order) | 分配 2^{order} 数量的页面并返回一个虚拟地址 |
| struct page * __get_dma_pages(unsigned int gfp_mask, unsigned int order) | 从 DMA 管理区分配 2^{order} 数量的页面并返回一个页面结构 |

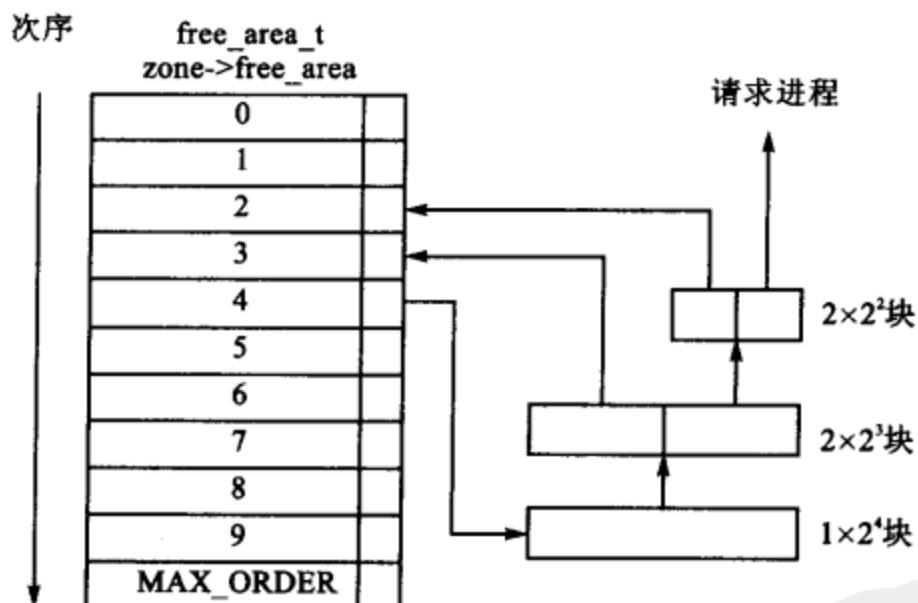


图 6.3 分配物理页面

第二步决定使用哪个内存节点以及哪个 `pg_data_t`。Linux 使用了本地节点的分配策略,这是为了使用和运行与页面分配进程的 CPU 相关联的内存库。此处函数 `_alloc_pages()` 非常重要,因为它根据内核是建立在 UMA(`mm/page_alloc.c` 中提供的功能)上还是建立在 NUMA(`mm/numa.c` 提供的功能)上而有所不同。

无论使用哪一个 API, `mm/page_alloc.c` 中的函数 `__alloc_pages()` 都是分配器的核心。这个函数从不被直接调用,它会检查选定的管理区,看这个管理区可用的页面数量上是否适合分配。如果这个管理区不适合,分配器将会退回到其他管理区。退回的管理区的次在启动时由函数 `build_zonelists()` 决定。但通常是从 `ZONE_HIGHMEM` 退回到 `ZONE_NORMAL`,

从 ZONE_NORMAL 再退回到 ZONE_DMA。如果空闲页面的数量达到 pages_low 的要求, 系统激活 kswapd 开始从管理区中释放页面。如果内存空间极度紧张, 调用者将自己完成 kswapd 的工作。

一旦最终选定了管理区, 系统会调用函数 rmqueue() 分配页面块, 或在没有合适大小的情况下切分高次的块。

6.3 释放页面

用于释放页面的 API 比较简单, 它们可以帮助记忆将要释放的页面块的次。伙伴分配器的一个缺点是调用者必须记住最初分配的大小。释放页面的 API 列表如表 6.2 所列。

表 6.2 物理页面释放 API

| | |
|---|-----------------------|
| void __free_pages(struct page * page, unsigned int order) | 从给定页面中释放次序为 order 的页面 |
| void __free_page(struct page * page) | 释放单个页面 |
| void free_page(void * addr) | 从给定虚拟地址空间删除一个页面 |

用于释放页面的主要函数是 __free_pages_ok(), 它不能被直接调用, 而是提供了函数 __free_pages(), 它会首先进行一些简单的检查, 如图 6.4 所示。

在释放伙伴时, Linux 会试着尽可能快地合并它们。但这并不是最优的做法, 因为在最坏的情况下, 分开块后即会有许多次的合并 [Vah96]。

为了探测伙伴们是不是可以合并, Linux 在 free_areamap 中检查与受影响的伙伴相对应的位。由于一个伙伴刚刚被此函数释放, 很显然它知道至少有一个伙伴是空闲的。如果切换以后位图中的位是 0, 那么另外一个伙伴也肯定是空闲的, 因为如果这个位是 0, 就意味着两个伙伴或者同为空闲或者同被分配。如果两个都是空闲的, 系统可以合并它们。

计算这个伙伴地址有一个著名的方法 [Knu68]。由于分配是以 2^k 个块进行的, 块的地址, 或者说至少它在 zone_mem_

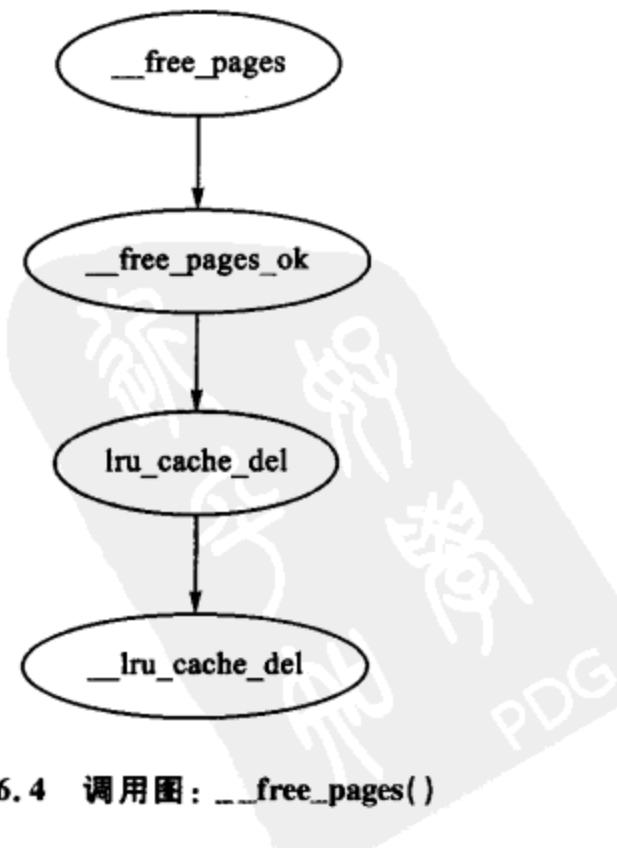


图 6.4 调用图: __free_pages()

map 的起始地址将会是 2^k 的整次幂。最终的结论是总会有至少 k 个数字的 0 在地址的右边。为了得到伙伴的地址,从右边数的第 k 位检测。如果为 0,伙伴将会翻转自己的位。为了得到这个位的值,Linux 引入了一个掩码,其计算如下:

$$\text{mask} = (\sim 0 \ll \ll k)$$

我们感兴趣的掩码是:

$$\text{imask} = 1 + \sim \text{mask}$$

Linux 在计算此掩码时采用了这个技巧:

$$\text{imask} = -\text{mask} = 1 + \sim \text{mask}$$

合并伙伴后,它便从空闲链表中被移除,并且这个新的合并对会被移到下一个高次链表中以确定是否可以再次合并。

6.4 获得空闲页面(GFP)标志位

获得空闲页面(GFP)标志位是贯穿整个虚拟内存的永恒概念。这个标志位决定分配器以及 kswapd 如何分配和释放页面。例如,一个中断处理程序也许不需要睡眠,所以它不需要设定 `__GFP_WAIT` 标志位,设置这个标志位表明调用者可以睡眠。存在 3 组 GFP 标志位,在 `linux/mm.h` 中有说明。

这 3 组中的第 1 组是列于表 6.3 中的管理区修饰符。这些标志位意味着调用者必须尽可能在一个特定管理区中进行分配。读者需要注意的是,这里没有一个适用于 `ZONE_NORMAL` 的管理区修饰符,因为管理区修饰符标志位在一个数组里是被用作开端的,0 则暗示在 `ZONE_NORMAL` 中进行分配。

表 6.3 影响管理区分配的低级 GFP 标志位

| 标志位 | 描述 |
|----------------------------|--|
| <code>__GFP_DMA</code> | 如果可能,从 <code>ZONE_DMA</code> 处开始分配 |
| <code>__GFP_HIGHMEM</code> | 如果可能,从 <code>ZONE_HIGHMEM</code> 处开始分配 |
| <code>GFP_DMA</code> | <code>__GFP_DMA</code> 的别名 |

下一个标志位是列于表 6.4 中的动作修饰符。它们改变了虚拟内存以及调用进程的运作。低级的标志位由于过于简单而不易使用。

表 6.4 影响分配器行为的低级 GFP 标志位

| 标志位 | 描述 |
|--------------|--|
| __GFP_WAIT | 表示调用者并非高优先级,但可以睡眠或重新调度 |
| __GFP_HIGH | 适用于高优先级或内核进程。内核 2.2.x 用它来决定一个进程是否可以使用内存的紧急情况池。在 2.4.x 内核中,这个标志位看起来并没有被使用 |
| __GFP_IO | 表示调用者可以进行低级的 I/O。在 2.4.x 中,这个标志位的主要作用是决定 <code>try_to_free_buffers()</code> 能不能清空缓冲区。这个标志位至少被一种日志文件系统所使用 |
| __GFP_HIGHIO | 决定 I/O 可以在映射到高端内存的页面上运行。这个标志位仅仅用在函数 <code>try_to_free_buffers()</code> 当中 |
| __GFP_FS | 表明调用者是否可以调用文件系统层。在调用者与文件系统相关时使用。例如缓冲区高速缓存,并且调用者想避免递归调用自己 |

要知道每一个实例的正确组合是非常困难的,所以 Linux 中给出了一些高级组合的定义,列于表 6.5 中。具体地,-GFP 从列表组合中被移除,所以-GFP-HIGH 标志位在表中将被称作 HIGH。表 6.6 给出了形成高级标志位的组合。为了帮助理解,我们以 GFP_ATOMIC 为例子说明。它仅仅设置了 __GFP_HIGH 标志位,这意味着它具有高优先级,并使用紧急情况池(如果存在),但不会睡眠,也不会执行 I/O 或访问文件系统。比如这个标志位将会在中断处理程序中使用。

表 6.5 低级 GFP 标志位和高级标志位的组合

| 标志位 | 低级的标志位组合 |
|--------------|---------------------------|
| GFP_ATOMIC | HIGH |
| GFP_NOIO | HIGH-WAIT |
| GFP_NOHIGHIO | HIGH-WAIT-IO |
| GFP_NOFS | HIGH-WAIT-IO-HIGHIO |
| GFP_KERNEL | HIGH-WAIT-IO-HIGHIO-FS |
| GFP_NFS | HIGH-WAIT-IO-HIGHIO-FS |
| GFP_USER | WAIT-IO-HIGHIO-FS |
| GFP_HIGHUSER | WAIT-IO-HIGHIO-FS-HIGHMEM |
| GFP_KSWAPD | WAIT-IO-HIGHIO-FS |

一个进程也许也会在 `task_struct` 中设置一些影响分配器动作的标志位。`linux/shed.h` 对所有进程标志位都有定义,表 6.7 只列举了影响虚拟内存的进程标志位。

表 6.6 影响分配器行为的高级 GFP 标志位

| 标志位 | 描述 |
|--------------|--|
| GFP_ATOMIC | 只有调用者不能睡眠并且必须在所有可能的情况下得到服务时使用该标志位。任何请求内存的中断处理程序必须使用该标志位以避免睡眠或者有 I/O 操作。许多子系统在初始化时将使用该系统,如 buffer_init() 和 inode_init() |
| GFP_NOIO | 该标志位为已经执行 I/O 相关操作的调用者所使用。例如,在回环设备尝试为一个缓冲区首部取得一个页面时,它就使用该标志位来保证其不会执行引起更多 I/O 操作的行为。实际上,似乎专门引入该标志位被以避免在回环设备上产生死锁 |
| GFP_NOHIGHIO | 该标志位只在一个地方使用,就是在为高端内存的 I/O 创建弹性缓冲区时使用 |
| GFP_NOFS | 该标志位只被缓冲区高速缓存和文件系统所使用,以保证它们不会出现偶尔递归调用自己的情况 |
| GFP_KERNEL | 这是在组合标志位中最为宽松的一个。它表明了调用者可以做它任意想做的事情。严格意义上,在该标志位和 GFP_USER 之间的区别就是它可以使用页面紧急池,但是它在 2.4.x 的内核中没什么用处 |
| GFP_USER | 这是另一个历史意义上的标志位。在 2.2.x 的内核系列中,每一个分配都给予一个 LOW, MEDIUM 或者 HIGH 的优先级。如果内存很紧张,那么一个带有 GFP_USER(低)标志位的请求会失败,然而其他的请求会继续尝试。目前它不再那么重要,Linux 如 GFP_KERNEL 一样看待 |
| GFP_HIGHUSER | 该标志位表明了分配器应该尽可能地从 ZONE_HIGHMEM 处开始分配。当分配的页面用于用户进程时使用该标志位 |
| GFP_NFS | 该标志位已经不再使用。在 2.0.x 内核系列中,该标志位决定保留的页面的大小。一般而言,会保留 20 个空闲页面。如果设置有该标志位,那么就只能保留 5 个空闲页面。目前其处理方式已没有什么不同 |
| GFP_KSWAPD | 该标志位也只具有历史意义。事实上,Linux 对待它和对待 GFP_KERNEL 没有什么两样 |

6.5 进程标志位

一个必须提出的分配器普遍存在的一个重要问题是内外部的碎片问题。外部碎片是由于可用内存全部是小块而不能满足要求。内部碎片是由于一个大的块必须被分开来响应一些小的请求而浪费的空间。在 Linux 中,外部碎片不是一个非常严重的问题,因为对大块连续页面的请求是非常少见的,通常 vmalloc() 就可以满足这些请求。空闲块的链表确保大的块在不必要的
情况下不被分开。

表 6.7 影响分配器行为的进程标志位

| 标志位 | 描述 |
|---------------|---|
| PF_MEMALLOC | 它标识进程为一个内存分配器。kswapd 设置该标志位,不论它被设置给哪些进程,都由 OOM 销毁程序完成销毁,这将在第 13 章讨论。它告诉伙伴分配器在各种的情况下都可以忽略管理区极值并分配页面 |
| PF_MEMDIE | 该位由 OOM 销毁程序完成设置,其功能与 PF_MEMALLOC 相同,也是告诉页面分配器在任何可能的情况下都分配页面,否则将杀死进程 |
| PF_FREE_PAGES | 该位在伙伴分配器调用 <code>try_to_free_pages()</code> 时设置,用于表明空闲页面应该在 <code>__free_page_ok()</code> 中的调用进程所保留,而不是返回空闲链表 |

6.6 防止碎片

内部碎片是二进制伙伴系统所特有的一个非常严重的问题。尽管预计碎片只存在于 28% 的空间中[WJNB95],但与首先适应分配器的仅 1% 相比,碎片已达到了 60% 的面积。即使使用多种伙伴系统也无法显著改善这种状况[PN77]。为了解决这个问题,Linux 使用了一个 slab 分配器[Bon94]将页面块切割为小的内存块进行分配[Tan01],这将在第 9 章中详细讨论。有了这种分配器的组合,内核可以使由内部分片导致的内存消耗量保持在最低限度。

6.7 2.6 中有哪些新特性

分配页面

尤其值得注意的不同之处似乎带有表面性。以前定义在<linux/gfp.h>中的函数 `alloc_pages()` 现在成了一个定义在<linux/mm.h>中的宏。新的布局仍然容易识别,主要的变化很微小,却很重要。2.4 内核通过编写特定的代码来完成基于运行中的 CPU 选择正确的节点进行分配,但在 2.6 内核中消除了 NUMA 和 UMA 这两种体系结构之间的差别。

在 2.6 内核中,函数 `alloc_pages()` 调用 `numa_node_id()` 返回一个当前正在运行的 CPU 相关节点的逻辑 ID。系统把该 NID 传递给 `to_alloc_pages()` 函数,作为该函数调用 `NODE_`

DATA()时的参数。在 UMA 结构中,将无条件地返回结果给 `contig_page_data`,在 NUMA 结构中取而代之的是建立了一个数组,其中存储了函数 `NODE_DATA()` 以 NID 作为的偏移量。或者说,体系结构负责为 NUMA 内存节点映射建立一个 CPU 的 ID 号。这在 2.4 内核中的节点局部分配策略下依然非常有效,而现在 2.6 中它的定义更加清晰。

Per-CPU 的页面链表

前面在 2.8 节中已经讨论过,页面分配中最重要的改变便是为 Per-CPU 增加的链表。

在 2.4 内核中,分配页面时,需要加上一个中断-安全的自旋锁。在 2.6 内核中,内存页面都由函数 `buffered_rmqueue()` 从 `struct per_cpu_pageset` 中分配。如果还没有达到内存下限 (`per_cpu_pageset->low`),则在不需要加自旋锁的情况下页面的分配从页面集中分配。在到达内存下限后,系统将会成批地分配大量的页面并加上中断-安全自旋锁,然后添加到 Per-CPU 的链表中,最后返回一个链表给调用函数。

通常更高次的分配是比较少见的,当然它也需要加上中断-安全自旋锁,这样分离和合并伙伴时才没有时间延迟。在 0 次的分配,分离时会有延迟,直到在 Per-CPU 集中到达内存下限为止,而合并时的延迟则会一直持续到达内存上限。

然而,严格意义上,这不是一个延迟伙伴关系算法[BL89]。尽管页面集中为 0 次分配引入了一个合并延迟,但它看起来更像是一个副作用而不是一个特意设计的特征,并且不存在什么有效的方法释放页面集以及合并伙伴。或者说,尽管 Per-CPU 的代码和新加入的代码占了 `mm/page_alloc.c` 文件中的大量代码,核心的伙伴关系算法却仍然和 2.4 内核中一样。

这个变化的意义是直截了当的;它减少了为保护伙伴链表的必须加锁的次数。在 Linux 中,高次的分配相对来说较少,因此优化适用于普通的情况。这个变化在有多个 CPU 的机器上更容易看出来,对于单个的 CPU 基本上没有什么差异。当然,页面集也有少量的问题,但它们都并不严重。

第 1 个问题是在页面集中如果有一些 0 次页以正常状态被合并到邻近的高次块中,则系统在进行高次分配时就有可能失败。第 2 个是如果内存很少,而当前 CPU 页面集为空并且其他 CPU 的页集都满时,由于不存在用作从远程页集中回收页面的机制,0 次页分配就有可能失败。最后一个潜在的问题是,伙伴关系中一些新的空闲页可能在其他的页面集中,这可能导致碎片问题。

空闲页面

前面已经介绍过两个新的释放页面的 API 函数 `free_hot_page()` 和 `free_cold_page()`。就是这两个函数决定了是否把 Per-CPU 页面集中的页面放在活动或者非活动链表中。然而,尽管设计了 `free_cold_page()` 函数,但实际上它从没有被调用过。

函数 `_free_pages()` 所释放的 0 次页面和函数 `_page_cache_release()` 所释放的页面高速缓存中的空闲页面都存放在活动链表中;反之,高次分配的页面都立即由函数 `_free_pages_`

ok()释放。0 次页面通常和用户空间相关,它的分配和释放是最普通的类型。由于大多数分配都是在 0 次上的,所以通过保持页面为 CPU 本地页面,将可以减少锁冲突。

最终,系统必须把页面链表传递给 free_pages_bulk() 函数,否则页面集链表将会占用所有的空闲页。函数 free_pages_bulk() 将获得一个链表,其中包括页面块分配、每一个块的次以及从该链表释放的块的计数。这里调用有两种情况:第 1 种情况是高次的页面被释放后传递给 __free_pages_ok() 函数。这时,页面块被放置在一个指定的次序链表中并被计数 1。第 2 种情况是,在运行的 CPU 中,页面集到达了内存上限。在这种情况下,页面集被赋予次 0 且计数为 pageset→batch。

在内核函数 __free_pages_bulk() 开始运行后,释放页面的机制与 2.4 中的伙伴链表非常类似。

GFP 标志位

仍然只有 3 个管理区,因此管理区修饰符仍然相同。然而,增加了 3 个新的 GFP 标志位,它们将影响 VM 的响应请求时工作力度,或者不工作。这些标志位如下:

__GFP_NOFAIL

这个标志位被调用函数用于表明分配从不失败,以及分配器应该保持分配的不确定性。

__GFP_REPEAT

这个标志位被调用函数用于表明被一个请求在分配失败后应当尝试再次分配。在目前的实现中,它与 __GFP_NOFAIL 标志位的行为相同,以后它会被规定为在一小段时间后失败。

__GFP_NORETRY

这个标志位几乎与 __GFP_NOFAIL 标志位相反。它表明,如果分配失败,应该立即返回。

在写本书的时候,这些标志位由于它们刚引入,还没有被大量地使用,随着时间的流逝,可能会被使用得越来越多。特别是 __GFP_REPEAT 标志位,将可能被广泛地使用,因为实现这个标志位行为的代码块贯穿于整个内核。

下面介绍的另外一个 GFP 标志位是一个称为 __GFP_COLD 的分配修改器,用于保证非活动页面从 Per-CPU 的链表中分配出来。以 VM 的观点来看,只有函数 page_cache_alloc_cold() 使用这个标志位,而该函数主要在 I/O 预读中使用。通常,页面分配从活动页面链表中取得页面。

最后一个新标志位是 __GFP_NO_GROW。这个标志位是 slab 分配器(在第 8 章中讨论)所使用的内部标志位,它的别名是 SLAB_NO_GROW。它用于表明新的 slab 任何时候都不应该从特定的高速缓存中分配。实际上,系统引入 GFP 标志位以补充旧的 SLAB_NO_GROW 标志位,后者目前在主流内核中并不被使用。

第 7 章

非连续内存分配

在处理高速缓存相关和内存存取所需大量内存的问题时,内存中使用连续的物理页应该是可取的,但由于伙伴系统的外部分页问题,这种做法又经常行不通。所以 Linux 使用一种叫 `vmalloc()` 的机制,在这种机制中,非连续的物理内存在虚存中是连续的。

Linux 中虚拟地址空间在 `VMALLOC_START` 和 `VMALLOC_END` 之间保留了一块区域, `VMALLOC_START` 的位置取决于可以访问的物理内存大小,该存储区域的大小至少是 `VMALLOC_RESERVE`, x86 上它的大小为 127 MB。5.1 节中已经讨论过该存储区域的精确的大小。

该存储区域的页表可以按照请求修改以指向物理页指示器分配的物理页,这就意味着分配的大小必须是硬件页面大小的整数倍。由于分配内存需要改变内核页表,而且仅可用在 `VMALLOC_START` 和 `VMALLOC_RESERVE` 之间的虚拟地址空间,所以在内存映射到 `vmalloc()` 时,内存的数量是有限制的。正是由于这个原因,页表仅保留在核心内核中使用。在 2.4.22 中,页表仅用于存储映射信息(见第 11 章)和把内核模块装载到内存。

本章首先描述内核如何跟踪使用 `vmalloc` 地址空间中区域,接着是如何分配和释放存储区域。

7.1 描述虚拟内存区

`vmalloc` 地址空间由一个资源映射分配器管理[Vah06], `struct vm_struct` 负责存储地址/大小键对,在 `linux/malloc.h` 中 `vm_struct` 定义为:

```
14 struct vm_struct {  
15     unsigned long flags;
```

```

16 void * addr;
17 unsigned long size;
17 struct vm_struct * next;
19 };

```

Linux 中的 VMA 的字段不止这么一些,它还包括其他的不属于 vmalloc 领域的信息,该结构体中的字段可以如下简要描述。

flags: 使用 vmalloc() 时设为 VM_ALLOC, 使用 ioremap 来完成高端内存地址到内核虚拟空间映射时设为 VM_IOREMAP。

addr: 内存块的起始地址。

size: 正如其名,它是以字节计的内存块大小。

next: 是一个指向下一个 vm_struct 的指针,所有的 vm_struct 以地址为次序,且该链表由 vmlist_lock 锁保护。

很显然,内存区域由 next 字段链到一起,并且为了查找简单,它们以地址为次序。为了防止溢出,每个区域至少由一个页面隔离开。如图 7.1 中的空隙所示。

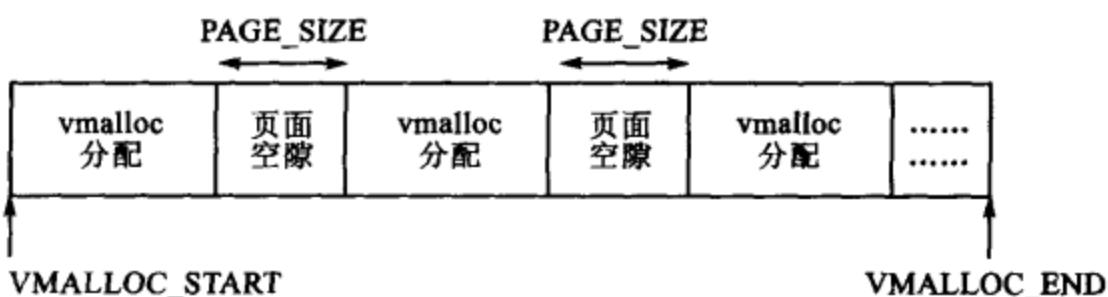


图 7.1 vmalloc 地址空间

当内核要分配一块新的内存时,函数 get_vm_area() 线性查找 vm_struct 链表,然后由函数 kmalloc() 分配结构体所需的空间。为了重新映射一块内存以完成 I/O,需要使用虚拟区域(习惯上称之为 ioremapping),系统将直接调用该函数完成请求区域的映射。

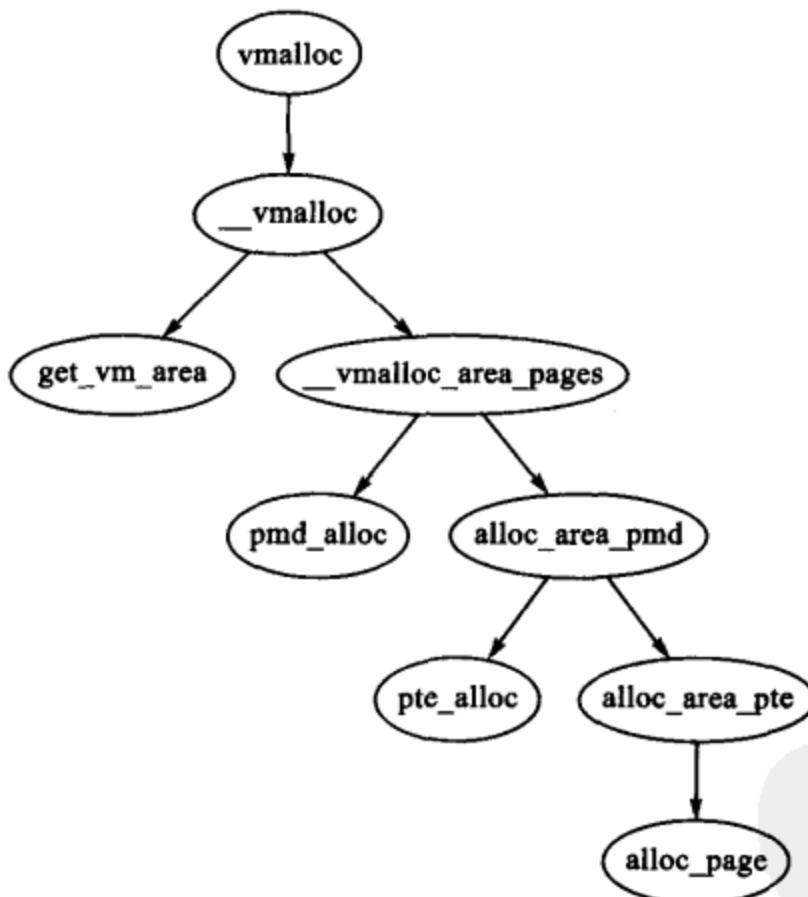
7.2 分配非连续区域

如表 7.1 所列,Linux 提供了函数 vmalloc(), vmalloc_dma() 和 vmalloc_32() 以在连续的虚拟地址空间分配内存。它们都只有一个参数 size,它的值是下一页的边距向上取整。这几个函数都返回新分配区域的线性地址。

表 7.1 非连续内存分配的 API

| |
|---|
| void * vmalloc(unsigned long size) |
| 在 vmalloc 空间分配请求大小的页面 |
| void * vmalloc_dma(unsigned long size) |
| 从 ZONE_DMA 分配页面 |
| void * vmalloc_32(unsigned long size) |
| 分配符合 32 位寻址的内存。函数保证物理页面帧在 32 位设备需要的 ZONE_NORMAL 中 |

图 7.2 中的调用图很清楚地表明,在分配内存时有两个步骤。第 1 步使用 `get_vm_area()` 找到满足请求大小的区域, `get_vm_area()` 查找 `vm_struct` 的线性链表, 然后返回一个描述该区域的新结构体。

图 7.2 调用图: `vmalloc()`

第 2 步首先使用 `vmalloc_area_pages()` 分配所需的 PGD 记录, 然后使用 `alloc_area_pmd()` 分配 PMD 记录, 接着使用 `alloc_area_pte()` 分配 PTE 记录, 最后使用 `alloc_page()` 分配页面。

`vmalloc()` 更新的页表并不属于当前进程, 属于当前进程的是在 `init_mm`→`pgd` 中的引用页表。这意味着当进程访问 `vmalloc` 区域时将发生缺页中断异常, 因为它的页表并没有指向正确的区域。在这个缺页中断处理代码中有个特殊的地方, 那就是中断处理代码知道在 `vmalloc` 区域有个缺页中断, 它利用主页表的信息更新当前进程的页表。使用 `vmalloc()` 处理

伙伴分配器以及缺页中断的方法如图 7.3 所示。

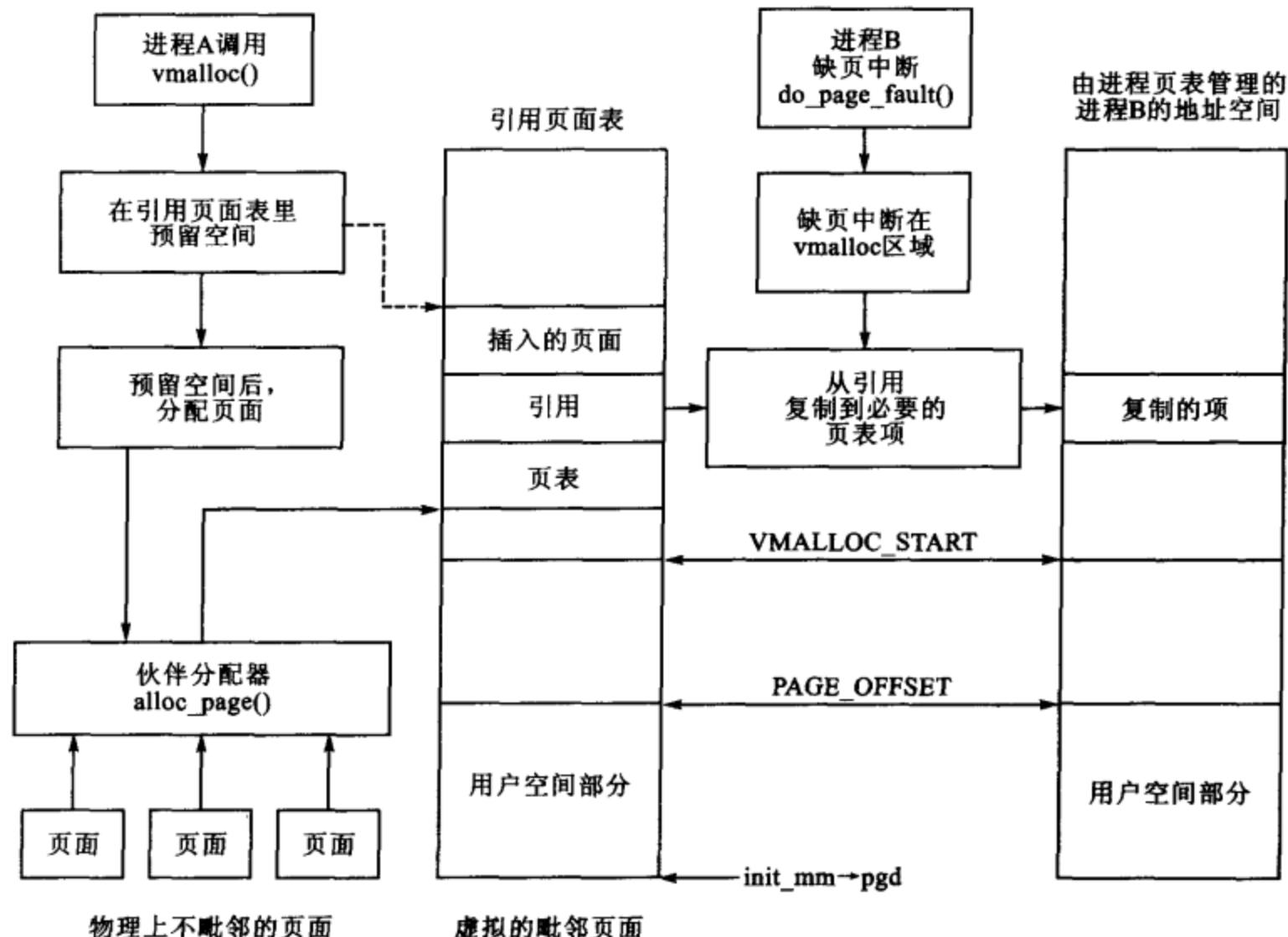


图 7.3 `vmalloc()`, `alloc_page()` 和缺页中断的关系

7.3 释放非连续内存

函数 `vfree()`(如表 7.2 所列)负责释放一块虚拟内存区域, 它首先线性查找 `vm_struct` 链表, 在找到需要释放的区域后, 就在其 `s` 上调用 `vmfree_area_pages()`。如图 7.4 所示。

表 7.2 非连续内存释放 API

```
void vfree(void * addr)
    释放由 vmalloc(), vmalloc_dma() 或 vmalloc_32() 分配的内存
```

`vmfree_area_pages()` 与 `vmalloc_area_pages()` 正好相反, 它遍历页表, 并清除该区域的页表记录和相应的页面。

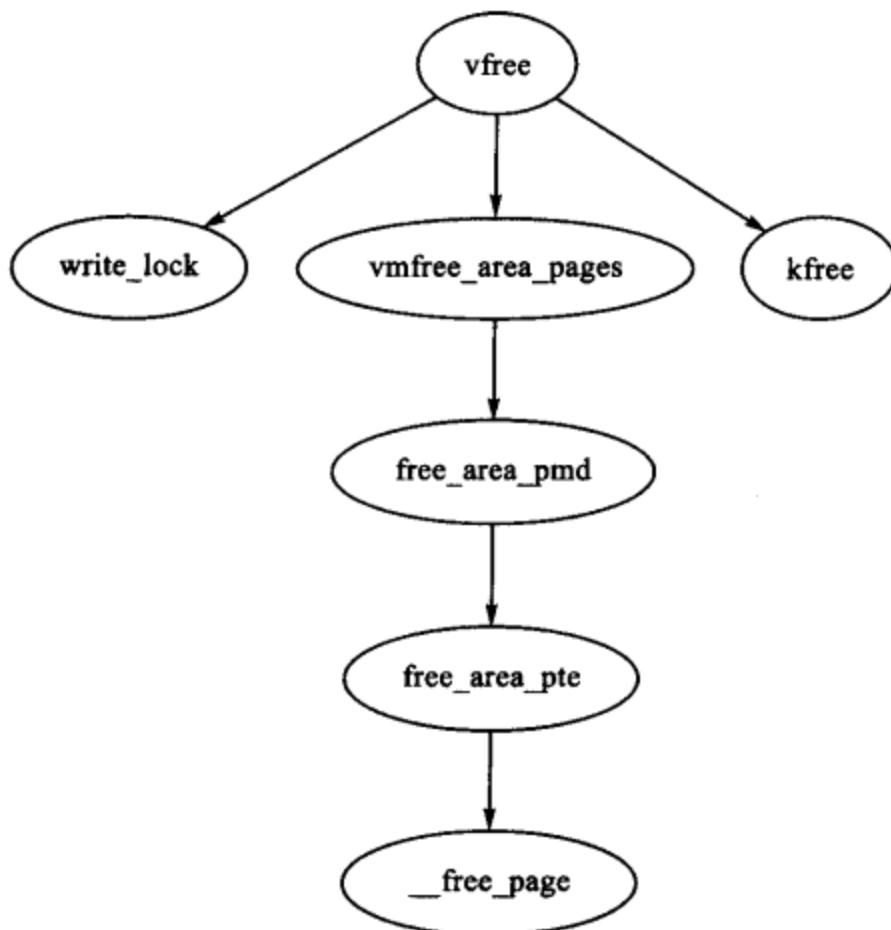


图 7.4 调用图: vfree()

7.4 2.6 中有哪些新特性

2.6 中的非连续内存分配基本与 2.4 中的保持不变。主要的区别是分配页面的内部 API 有一些细微区别。在 2.4 中, vmalloc_area_pages()负责遍历页表, 调用 alloc_area_pte()分配 PTE 以及页表。在 2.6 中, 所有的页表都由 __vmalloc()预先分配, 然后存储在一个数组中并传递给函数 map_vm_area(), 由它向内核页表中插入页面。

API 函数 get_vm_area()也有细微的改变。在调用它时, 它和以前一样遍历整个 vmalloc 虚拟地址空间, 寻找一块空闲区域。但是, 调用者也可以直接调用 __get_vm_area()并指明范围, 就可以只遍历 vmalloc 地址空间的一部分。这仅用于高级 RISC 机器(ARM) 装载模块。

最后一个显著的改变是引入了一个新的接口 vmap(), 它负责向 vmalloc 地址空间插入页面数组。vmap()仅用于声音子系统的内核。这个接口向后兼容到 2.4.22, 但那时根本没有用到过它。它的引入仅仅是偶然性的向后兼容, 或者是为了减轻那些需要 vmap()的特定供应商补丁的应用负担。

第 8 章

Slab 分配器

这一章将介绍一种通用的分配器: slab 分配器。它与 Solaris 系统[MM01]上的通用内核分配器在许多方面都很相似。Linux 中 slab 的实现主要是基于 Bonwick[Bon94]的第 1 篇 slab 分配器的论文,并进一步做了大量的改进,这在他最近的论文中[BA01]有所描述。这一章我们先对 slab 分配器做一个快速的浏览,然后描述所用到的数据结构,并深入讨论 slab 分配器的每一项功能。

slab 分配器的基本思想是:将内核中经常使用的对象放到高速缓存中,并且由系统保持为初始的可利用状态。如果没有基于对象的分配器,内核将花费更多的时间分配、初始化以及释放一个对象。slab 分配器的目的是缓存这些空闲的对象,保留基本结构,从而能够重复使用它们[Bon94]。

slab 分配器由多个高速缓存组成,它们通过双向循环链表连接在一起,称为高速缓存链表。在 slab 分配器中,高速缓存可以管理各种不同类型的对象,如 `mm_struct` 或者 `fs_cache`,它们由 `struct kmem_cache_s` 管理,在后面我们将详细讨论这个结构。高速缓存链表是通过字段 `next` 链接在一起的。

每一个高速缓存包括若干 slab, slab 由连续的页面帧组成,它们被划分成许多小的块以存放由高速缓存所管理的数据结构和对象。不同数据结构之间的关系如图 8.1 所示。

slab 分配器有三个基本目标:

- 减少伙伴系统分配小块内存时所产生的内部碎片。
- 把经常使用的对象缓存起来,减少分配、初始化以及释放对象的时间开销。在 Solaris 上的基准测试表明,使用 slab 分配器后速度有了很大的提高[Bon94]。
- 调整对象以更好地使用 L1 和 L2 硬件高速缓存。

为减少二进制伙伴分配器所产生的内部碎片,系统需要维护两个由小内存缓冲区所构成的高速缓存集,它们的大小从 2^5 (32)字节到 2^{17} (131 072)字节。一个高速缓存集适用于使用

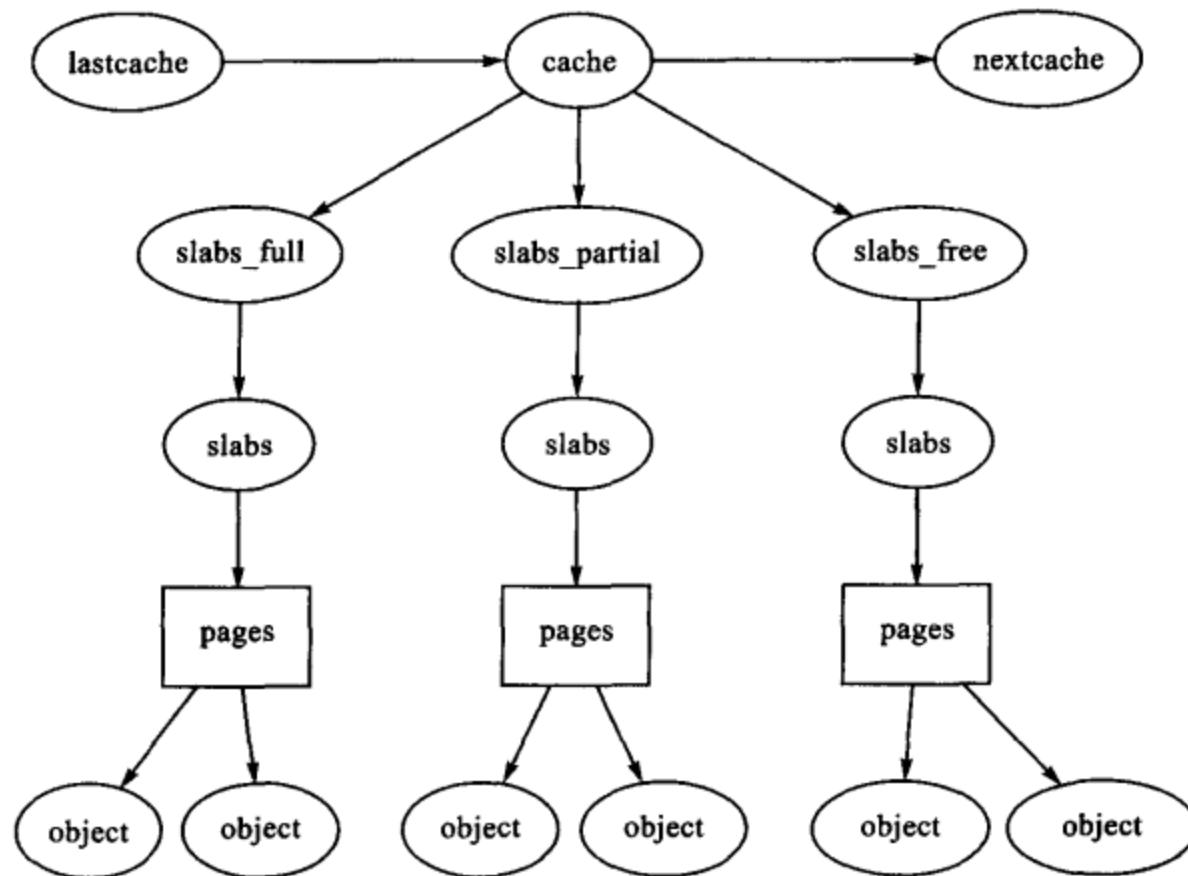


图 8.1 slab 分配器布局

DMA 的设备。这些高速缓存叫作 size- N 和 size- N (DMA), N 是分配器的尺寸, 而函数 `kmalloc()`(见 8.4.1 小节)负责分配这些高速缓存。这样就解决了低级页分配器最大的问题。我们将在 8.4 节详细讨论指定大小的高速缓存。

slab 分配器的第 2 个任务是缓存经常使用的对象。初始化内核中许多数据结构所需要的时间与分配空间所花的时间相当, 甚至超过了分配空间所花时间。在创建一个新的 slab 时, 一些对象被存放在里面, 并且由可用的构造器初始化它们。而在对象释放后, 系统将保持它们为初始化时的状态, 所以再次分配对象的速度很快。

slab 分配器的第 3 个任务是充分利用硬件高速缓存。若对象放入 slab 还有剩余的空间, 它们就用于为 slab 着色。slab 着色方案试图让不同 slab 中的对象使用不同的硬件高速缓存行。通过将对象放置在 slab 中不同的位移起始处, 对象将很可能利用 CPU 中不同的硬件高速缓存行, 因此来自同一个 slab 中的对象不会彼此刷新高速缓存。这样, 空间可能会因为着色而浪费掉。如图 8.2 所示, 从伙伴分配器分配的页面如何存储因对齐 CPU L1 硬件高速缓存而着色的对象。

Linux 显然不会试图去给基于物理地址分配的页面帧着色[Kes91], 也不会将对象放在某一特定位置, 如数据段[GAV95]、代码段[HK97]中, 但是 slab 着色方案都可以提高硬件高速缓存行的利用率。在 8.1.5 小节中将深入讨论高速缓存着色。在 SMP 系统中, 有另外的方案

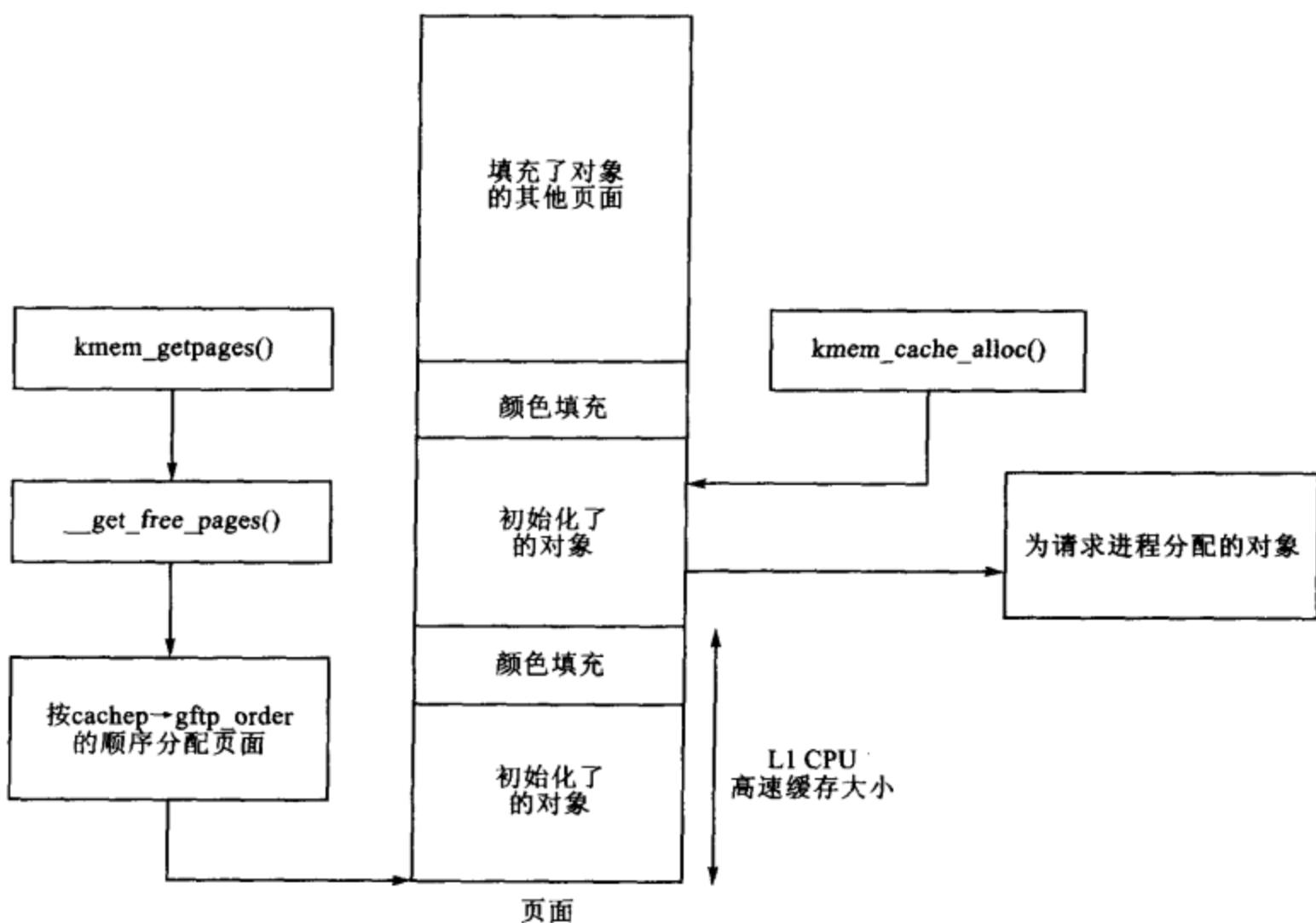


图 8.2 包含对齐 CPU L1 硬件高速缓存对象的页面帧

来利用高速缓存,即每一个高速缓存都有一个小的对象数组,而这些对象就是为每一个 CPU 所保留的。这些内容将会在 8.5 节中进一步讨论。

在编译的时候,如果设置了选项 CONFIG_SLAB_DEBUG, slab 分配器会提供额外的 slab 调试选项。其中提供的两个调试特征称为红色区域和对象感染。在红色区域中,每一个对象的末尾有一个标志位。如果这个标志位被打乱,分配器就会知道发生缓冲区溢出的对象的位置并且向系统报告。创建和释放 slab 时,感染一个对象会将预先定义好的位模(在 mm/slab.c 中定义为 0x5A)填入其中。分配时会检查这个位模,如果被改变,分配器就知道这个对象在之前已经使用过,并把它标记出来。

分配器中小而强大的 API 如表 8.1 所列。

表 8.1 slab 分配器中与高速缓存相关的 API

| | |
|--|--|
| kmem_cache_t * kmem_cache_create(const char * name, size_t size, size_t offset, unsigned long flags, void (* ctor)(void *, kmem_cache_t *, unsigned long), void (* dtor)(void *, kmem_cache_t *, unsigned long)) | 创建高速缓存并插入 cache_chain |
| int kmem_cache_reap(int gfp mask) | 最多扫描 REAP_SCANLEN 个高速缓存, 并选择一个, 回收所有的 per-cpu 对象, 并释放其中的 slab。当内存紧张时调用 |
| int kmem_cache_shrink(kmem_cache_t * cachep) | 这个函数删除所有的和高速缓存相关的 per-cpu 对象, 并删除空闲 slab 链表中的所有 slab。返回释放的页数目 |
| void * kmem_cache_alloc(kmem_cache_t * cachep, int flags) | 从高速缓存中分配一个对象, 返回给调用者 |
| void kmem_cache_free(kmem_cache_t * cachep, void * objp) | 释放一个对象并返回给高速缓存 |
| void * kmalloc(size_t size, int flags) | 从一个指定大小的高速缓存中分配一块内存 |
| void kfree(const void * objp) | 释放由 kmalloc 分配的内存 |
| int kmem_cache_destroy(kmem_cache_t * cachep) | 从链表中删除高速缓存之前, 销毁所有 slab 中的所有对象, 并释放相关的内存 |

8.1 高速缓存

每种类型的对象所对应的高速缓存都将被缓存起来。执行命令 `cat /proc/slabinfo` 就可以得到运行系统中完整的可用高速缓存列表。这个文件包含高速缓存的基本信息。文件摘录如下：

| slabinfo | -version: 1.1 (SMP) | | | | | | | | |
|-----------------|---------------------|------|-----|-----|-----|---|---|-----|-----|
| kmem_cache | 80 | 80 | 248 | 5 | 5 | 1 | : | 252 | 126 |
| urb_priv | 0 | 0 | 64 | 0 | 0 | 1 | : | 252 | 126 |
| tcp_bind_bucket | 15 | 226 | 32 | 2 | 2 | 1 | : | 252 | 126 |
| inode_cache | 5714 | 5992 | 512 | 856 | 856 | 1 | : | 124 | 62 |

| | | | | | | | | | |
|----------------|------|------|-----|-----|-----|---|---|-----|-----|
| dentry_cache | 5160 | 5160 | 128 | 172 | 172 | 1 | : | 252 | 126 |
| mm_struct | 240 | 240 | 160 | 10 | 10 | 1 | : | 252 | 126 |
| vm_area_struct | 3911 | 4480 | 96 | 112 | 112 | 1 | : | 252 | 126 |
| size-64(DMA) | 0 | 0 | 64 | 0 | 0 | 1 | : | 252 | 126 |
| size-64 | 432 | 1357 | 64 | 23 | 23 | 1 | : | 252 | 126 |
| size-32(DMA) | 17 | 113 | 32 | 1 | 1 | 1 | : | 252 | 126 |
| size-32 | 850 | 2712 | 32 | 24 | 24 | 1 | : | 252 | 126 |

其中每一列对应于 struct kmem_cache_s 结构中的一个字段,每列含意如下。

cache-name: 便于识别的名称,例如“tcp_bind_bucket”。

num-active-objs: 正在使用的对象数目。

total-objs: 可用的对象总数,包括未使用的。

obj-size: 对象的大小,一般都很小。

num-active-slabs: slab 中活动的对象数目。

total-slabs: 存在的 slab 总数。

num-pages-per-slab: slab 占用的页面帧数,一般是 1。

在 SMP 系统中,上面的摘录将会在分号后多显示两列或者更多。它们涉及 8.5 节中所述的 per-CPU 高速缓存。这些列如下。

limit: 缓冲池中一半空间作为全局空闲池时,其所能容纳的空闲对象数目。

batchcount: 在没有空闲对象时,在一个块里为处理器分配的对象数目。

为加快分配、释放对象的和 slab 自身的速度,slab 被组织成 3 个链表:slabs_full, slabs_partial 以及 slabs_free。slabs_full 中所有对象都在使用中,而 slabs_partial 中有空闲对象,所以分配对象时可以考虑它。slabs_free 中没有已分配的对象,因此它主要用于 slab 销毁。

8.1.1 高速缓存描述符

所有描述高速缓存的信息都存储在 mm/slab.c 中声明的 struct kmem_cache_s 中。它是一个非常大的结构,这里仅描述其中一部分。

```

190 struct kmem_cache_s {
193 struct list_head slabs_full;
194 struct list_head slabs_partial;
195 struct list_head slabs_free;

```

```

196 unsigned int objsize;
197 unsigned int flags;
198 unsigned int num;
199 spinlock_t spinlock;
200 #ifdef CONFIG_SMP
201 unsigned int batchcount;
202#endif
203

```

下面这些字段在分配和释放对象时很重要。

slabs_*：在前一节中介绍的三个 slab 链表。

objsize：slab 中每一个对象的大小。

flags：一组标志位，决定分配器应该如何处理高速缓存，见 8.1.2 小节。

num：每一个 slab 中包含的对象个数。

spinlock：并发控制锁，避免并发访问高速缓存描述符。

batchcount：在前面的章节中描述的为 per-cpu 高速缓存批量分配的对象数目。

```

206 unsigned int gfporder;
209 unsigned int gfpflags;
210
211 size_t colour;
212 unsigned int colour_off;
213 unsigned int colour_next;
214 kmem_cache_t * slabp_cache;
215 unsigned int growing;
216 unsigned int dflags;
217
219 void (* ctor)(void *, kmem_cache_t *, unsigned long);
222 void (* dtor)(void *, kmem_cache_t *, unsigned long);
223
224 unsigned long failures;
225

```

从高速缓存中分配和释放 slab 时将用到这些字段。

gfporder：slab 以页为单位的大小，每个 slab 占用 $2^{gfporder}$ 个连续页面帧，其中分配的大小由伙伴系统提供。

gfpflags：调用伙伴系统分配页面帧时用到的一组 GFP 标志位，完整列表见 7.4 节。

colour：尽可能将 slab 中的对象存储在不同的硬件高速缓存行中。高速缓存着色将在

8.1.5 小节中进一步讨论。

colour_off: 对齐 slab 中的字节。例如, size-X 高速缓存对齐 L1 硬件高速缓存。

colour_next: 这是下一个将使用的着色行, 其值超过 colour 时, 它重新从 0 开始。

growing: 这个标志位用于指示高速缓存是否增长。如果设置了该标志位, 就不太可能在处于内存压力的情况下选中该高速缓存以回收空闲 slab。

dflags: 动态标志位, 在高速缓存的生命期里动态变化, 见 8.1.3 小节。

ctor: 为复杂对象提供的构造函数, 用以初始化每一个新的对象。这是一个函数指针, 可能为空(NULL)。

dtor: 对象的析构函数指针, 也可能为空。

failures: 仅仅初始化为 0, 其他的地方都没有用到。

```
227 char name[CACHE_NAMELEN];
228 struct list_head next;
```

这两个字段在创建高速缓存时设置。

name: 便于识别的高速缓存名称。

next: 指向高速缓存链表中的下一个高速缓存。

```
229 # ifdef CONFIG_SMP
231 cpucache_t * cpudata[NR_CPUS];
232 # endif
```

cpudata: per-cpu 数据, 在 8.5 中进一步讨论。

```
233 # if STATS
234 unsigned long num_active;
235 unsigned long num_allocations;
236 unsigned long high_mark;
237 unsigned long grown;
238 unsigned long reaped;
239 unsigned long errors;
240 # ifdef CONFIG_SMP
241 atomic_t allochit;
242 atomic_t allocmiss;
243 atomic_t freehit;
244 atomic_t freemiss;
245 # endif
246 # endif
247 };
```

在编译时设置 CONFIG_SLAB_DEBUG 选项时,这些数字才有效。它们都是不重要的计数器,一般不必关心。读取/proc/slabinfo 里的统计信息时,与其依赖这些字段是否有效,还不如通过另一个进程检查每个高速缓存里的每个 slab 使用情况。

num_active: 当前在高速缓存中活动的对象数目。

num_allocations: 已经分配的对象数目。

high_mark: num_active 的上限。

grown: kmem_cache_grow() 的调用次数。

reaped: 该高速缓存被回收的次数。

errors: 这个字段从未使用过。

allochit: 分配器使用 per-cpu 高速缓存的次数。

allocmiss: 对 allochit 的补充,是分配器未命中 per-cpu 高速缓存的次数。

freehit: 在 per-cpu 高速缓存中空闲的次数。

freemiss: 对象释放后被置于全局池中的次数。

8.1.2 高速缓存静态标志位

一些标志位在高速缓存初始化时就被设置了,并且在生命周期内一直不变。这些标志位影响到 slab 的结构如何组织以及对象如何存储在 slab 中。这些标志位组成一个位掩码,存储在高速缓存描述符的字段 flag 中。所有的标志位都在 linux/slab.h 中声明。

有 3 组标志位,第 1 组是内部标志位,仅由 slab 分配器使用,如表 8.2 所列。CFGs_OFF_SLAB 标志位决定 slab 描述符存储的位置。

表 8.2 高速缓存静态内部标志位

| 标志位 | 描述 |
|----------------|---|
| CFGs_OFF_SLAB | 表示 slab 描述符存储在 slab 之外,在 8.2.1 小节将进一步讨论 |
| CFLGS_OPTIMIZE | 仅设置,但从未使用 |

第 2 组在创建高速缓存时设置,这些标志位决定分配器如何处理 slab,如何存储对象。列表如 8.3 所列。

表 8.3 由调用者设置的高速缓存静态标志位

| 标志位 | 描述 |
|------------------------|--------------------------------------|
| SLAB_HWCACHE_ALIGN | 对象对齐 CPU L1 硬件高速缓存 |
| SLAB_MUST_HWCACHEALIGN | 强制对齐 CPU L1 硬件高速缓存, 即便很浪费或者系统允许了调试状态 |
| SLAB_NO_REAP | 从不释放这个高速缓存里的 slab |
| SLAB_CACHE_DMA | 从内存区 ZONE_DMA 中分配 slab |

第 3 组在编译选项 CONFIG_SLAB_DEBUG 设置后才有效, 列表如 8.4 所列。它们决定了对 slab 和对象做哪些附加的检查, 主要与新建的高速缓存相关。

表 8.4 高速缓存静态调试标志位

| 标志位 | 描述 |
|--------------------|--|
| SLAB_DEBUG_FREE | 空闲时执行费时的检测 |
| SLAB_DEBUG_INITIAL | 空闲时调用构造器验证, 确保正确初始化对象 |
| SLAB_RED_ZONE | 给任意对象的末尾放置一个标志位, 跟踪溢出 |
| SLAB_POISON | 为了跟踪对没有进行分配或者进行初始化的对象所作的变更, 用已知的模式感染对象 |

为了防止调用错误的标志位, mm/slab.c 中定义了 CREATE_MASK, 它由所有允许的标志位所组成。在创建一个高速缓存时, 请求的标志位会和 CREATE_MASK 作比较, 如果使用了无效的标志位, 系统将会报告一个错误。

8.1.3 高速缓存动态标志位

虽然字段 dflag 中只有一个标志位 DFLGS_GROWN, 但它却非常重要。该标志位在 kmem_cache_grow() 中设置, 这样 kmem_cache_reap() 就不会选择该高速缓存进行回收。kmem_cache_reap() 将跳过设置了该标志位的高速缓存, 并清除该标志位。

8.1.4 高速缓存分配标志位

这些标志位, 与为 slab 分配页面帧的 GFP 页面标志位选项相对应, 列表如 8.5 所列。调用者既可以使用 SLAB_* 标志位, 也可以使用 GFP_* 标志位, 但实际上应该仅仅使用 SLAB_* 标志位。它们和 6.4 节中所述的标志位直接对应, 所以在这里不再详细讨论。假设

现在也有可能允许其他类型的帧唤醒系统。有少量设备可以根据在模块加载期间设置的一个参数开启或关闭 WOL 功能（参见 *drivers/net/3c59x.c* 为例）。*ethtool* 工具允许管理员配置哪几种帧可以唤醒系统，其中一种选择是 ARP 包，如第二十八章的“网络唤醒事件”一节所述。*net-utils* 套件中有一个命令 *ether-wake*，可用于产生 WOL Ethernet 帧。

只要一个开启 WOL 功能的设备识别一个帧，其类型为可唤醒系统，就会产生一个电源管理通知信息，去做相应的工作。

有关电源管理的更多细节，可参考第八章的“与电源管理之间的交互”一节。

PCI NIC 驱动程序注册范例

我们用 *drivers/net/e100.c* 中的 Intel PRO/100 Ethernet 驱动程序来说明驱动程序的注册：

```
#define INTEL_8255X_ETHERNET_DEVICE(device_id, ich) (\n    PCI_VENDOR_ID_INTEL, device_id, PCI_ANY_ID, PCI_ANY_ID, \n    PCI_CLASS_NETWORK_ETHERNET << 8, 0xFFFF00, ich )\nstatic struct pci_device_id e100_id_table[] = {\n    INTEL_8255X_ETHERNET_DEVICE(0x1029, 0),\n    INTEL_8255X_ETHERNET_DEVICE(0x1030, 0),\n    ...\n}
```

由“PCI NIC 设备驱动程序的注册”一节可知，PCI NIC 设备驱动程序会把一个 *pci_device_id* 结构向量注册给内核，该结构向量列出了其所能处理的那些设备。例如，*e100_id_table* 就是 *e100.c* 驱动程序使用的结构。注意：

- 第一个字段（相当于此结构定义中的 *vendor*）为固定值 *PCI_VENDOR_ID_INTEL*，它为指定给 Intel 初始化的开发商 ID（注 3）。
- 第三字段和第四字段（*subvendor* 和 *subdevice*）通常初始化为通用值 *PCI_ANY_ID*，因为前两个字段（*vendor* 和 *device*）足以识别那些设备。
- 许多设备都会对那张设备表使用宏 *__devinitdata*，用来标记初始化数据，不过 *e100_id_table* 没有。第七章将说明这个宏有什么作用。

此模块由 *module_init* 宏指定的 *e100_init_module* 初始化（注 4）。当此函数在引导期间或者模块加载期间由内核执行时，就会调用“PCI NIC 设备驱动程序的注册”一节

注 3：更新的列表可在 <http://pciids.sourceforge.net> 中找到。

注 4：有关模块初始化代码的更多细节，参见第七章。

最好是通过一个例子来解释。为方便起见,在 slab 中设置 `s_mem`(第 1 个对象的地址)为 0,且 slab 中共有 100 B 被浪费掉,另外奔 II 的硬件高速缓存 L1 是 32 B。

在这个例子中,第 1 个创建的 slab 使得它的对象从 0 开始。第 2 个从 32 开始,第 3 个从 64 开始,第 4 个从 96 开始,第 5 个又从 0 开始。这样,从不同 slab 中来的对象就不会命中同一硬件高速缓存行了。`colour` 和 `colour_off` 的值分别是 3 和 32。

8.1.6 创建高速缓存

函数 `kmem_cache_create()` 用于创建高速缓存并把它们添加到高速缓存链表中去。创建高速缓存有以下几步:

- 为了防止错误的调用,执行基本的检查。
- 如果设置了 `CONFIG_SLAB_DEBUG`,则执行调试检查。
- 从 `cache_cache` slab 高速缓存中分配一个 `kmem_cache_t` 结构。
- 将对象的大小调整为字对齐。
- 计算有多少对象可以放入一个 slab 中。
- 调整对象大小以适应硬件高速缓存。
- 计算着色偏移量。
- 初始化高速缓存描述符中其他字段。
- 将这个新的高速缓存添加到高速缓存链表中。

图 8.3 是创建高速缓存相关的函数调用图,每一个函数在代码注释部分都有完整描述。

8.1.7 回收高速缓存

在一个 slab 空闲时,系统将其放到 `slab_free` 链表中,以备将来使用。高速缓存不会自动收缩它们自己,因此当 `kswapd` 发现内存紧张时,就调用 `kmem_cache_reap()` 释放一些内存。这个函数负责选择一个需要收缩它自己内存使用的高速缓存。值得注意的是,回收高速缓存不会考虑哪些内存节点或者哪些管理区处于压力下。这意味着,在 NUMA 或者有高端内存的机器上,内核可能会在没有内存压力的区域花大量的时间释放内存资源,但是在诸如 x86 体系结构的机器上不会出现这样的问题,因为它只有一个内存区。

图 8.4 中调用图虽然简单但其实是个假象,因为选择一个适当的高速缓存进行回收其实是一个非常漫长的过程。在有大量高速缓存的系统中,每次调用却只检查 `REAP_SCANLEN`(目前定义为 10)个高速缓存。最后一个被检查的高速缓存保存在变量 `clock_searchp` 中,以防止重复检查同一个高速缓存。对于每个已经检查过的高速缓存,回收器做如下事情:

- 检查 `SLAB_NO_REAP` 标志位,如果设置了就跳过。

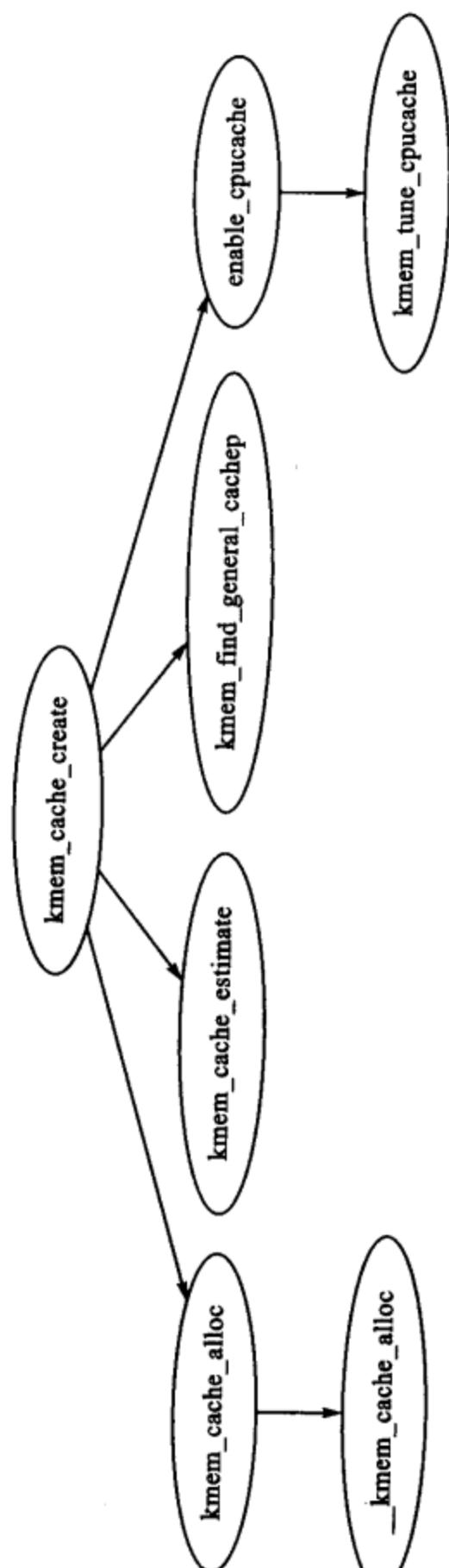


图8.3 调用图：kmem_cache_create()

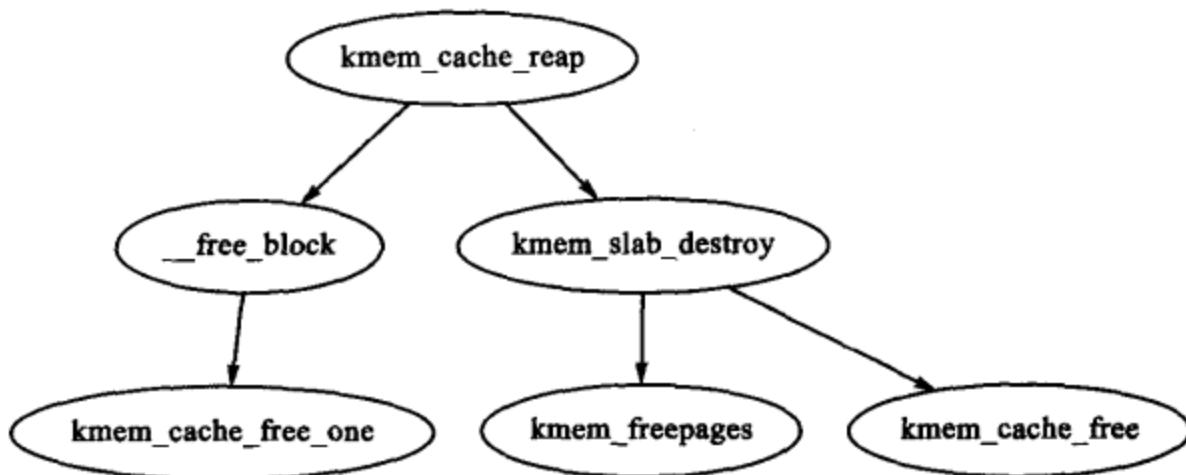


图 8.4 调用图: kmem_cache_reap()

- 如果高速缓存正在增长,也跳过。
- 如果高速缓存最近已经增长了,或正在增长,就设置标志位 DFLGS_GROW。如果该标志位已经被设置了,则跳过 slab,但同时要清除这个标志位,以便作为下一次回收的候选对象。
- 统计 slabs_free 链表中空闲 slab 的数量,并且计算在变量 pages 这么多个页面中将释放多少页面。
- 如果高速缓存有构造器,或者有大型的 slab,则调整 pages 以减少该高速缓存被选中的机会。
- 如果待释放的页面数量超过 REAP_PERFECT,则释放 slabs_free 中一半的 slab。
- 否则,扫描剩下的高速缓存,并选择可释放最多页数的高速缓存,以释放 slabs_free 中它一半的 slab。

8.1.8 收缩高速缓存

在选中一个高速缓存以收缩它自己时,步骤既简单又直接:

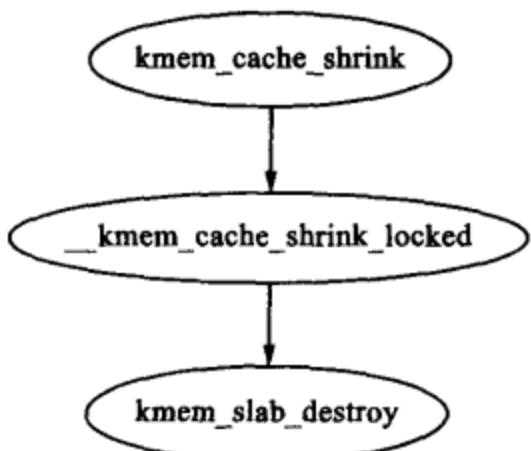


图 8.5 调用图: kmem_cache_shrink()

- 删除 per-CPU 高速缓存中所有的对象。
- 删除 slabs_free 中所有的 slab,除非设置了增长标志位。

如果不是如此的微妙,Linux 其实什么也不是。

这里提供了两个易混淆的有相似名称却互不相同的收缩函数。如图 8.5 所示,kmem_cache_shrink() 删除 slabs_free 链表中所有的 slab 并返回已释放页面的数量。它是提供给 slab 分配器用户所使用的主要函数。

第2个函数`__kmem_cache_shrink()`如图8.6所示,它释放`slabs_free`链表中所有的slab,并且核实`slab_partial`链表和`slab_full`链表是否为空。该函数仅用于内部,在销毁高速缓存的过程中显得非常重要,它不管要释放多少页面,只需高速缓存为空。

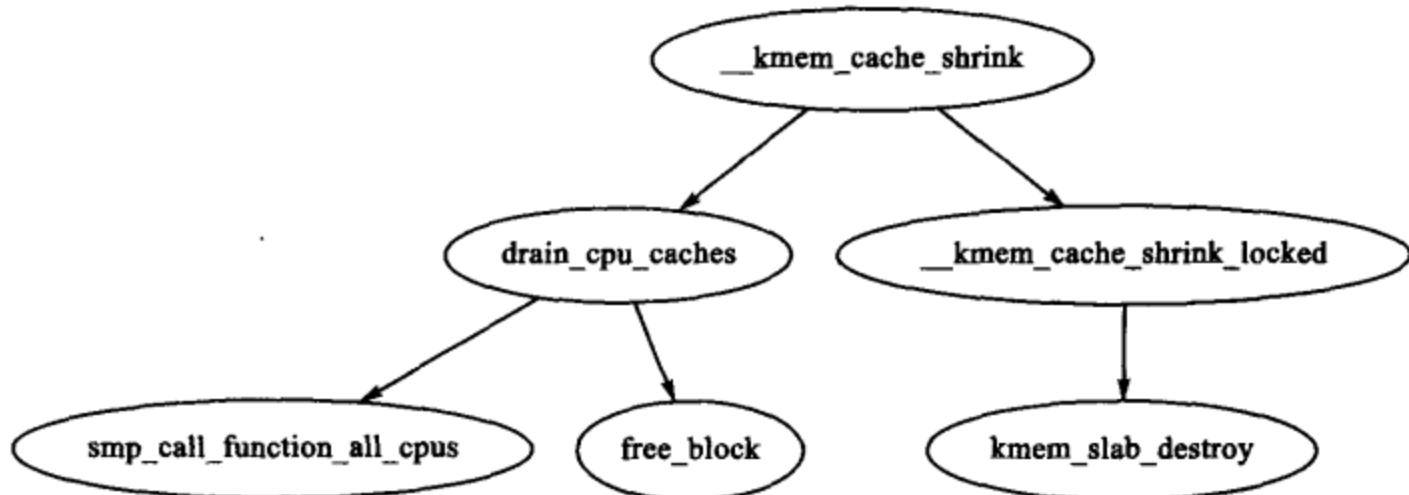


图8.6 调用图: `__kmem_cache_shrink()`

8.1.9 销毁高速缓存

在卸载一个模块时,Linux需要由函数`kmem_cache_destroy()`销毁所有与之相关的高速缓存,如图8.7所示。是否适当地销毁高速缓存显得很重要,因为不允许有两个相同名称的高速缓存同时存在。核心内核代码一般不用关心去销毁它自己的高速缓存,因为这些高速缓存在整个系统生存期中一直存在。销毁一个高速缓存的步骤如下:

- 从高速缓存链表中删除该高速缓存。
- 收缩高速缓存,删除其中所有的slab。
- 释放任意的per-CPU高速缓存(`kfree()`)。
- 从`cache_cache`结构中删除高速缓存描述符。

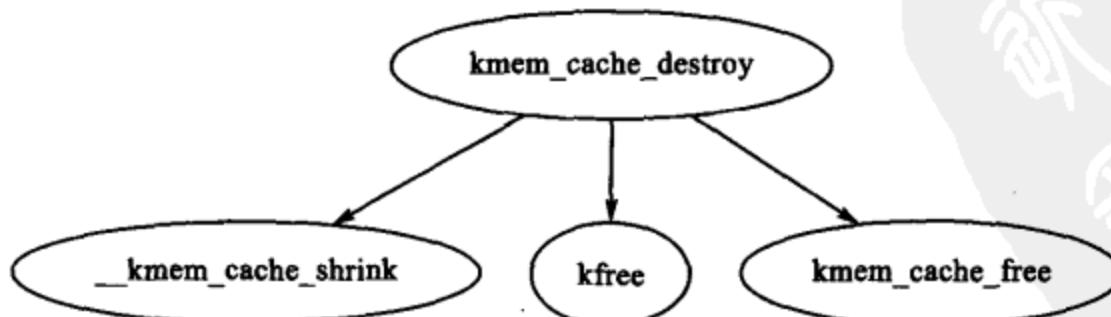


图8.7 调用图: `kmem_cache_destroy()`

8.2 Slabs

这一节描述如何构造以及如何管理 slab。它的结构描述虽然比高速缓存描述符要简单得多,但是如何组织 slab 却更为复杂。它的声明如下:

```
typedef struct slab_s {
    struct list_head list;
    unsigned long colouroff;
    void * s_mem;
    unsigned int inuse;
    kmem_bufctl_t free;
} slab_t;
```

这个简单结构的各字段如下。

list: 该 slab 所属的链接列表,可能是高速缓存管理器中 slab_full, slab_partial 或者 slab_free 其中的一个。

colouroff: slab 中的相对于第一个对象地址的着色偏移。第一个对象的地址是: s_mem + colouroff。

s_mem: slab 中第一个对象的起始地址。

inuse: slab 中活动的对象数目。

free: 这是一个 bufctls 数组,用于存储空闲对象的位置。更多的细节见 8.2.3 小节。

读者应该注意到,对于 slab 中给定的 slab 管理器或者对象,没有明显的方法判断它应属于哪一个 slab 或者高速缓存。这里通过 struct page 里的 list 字段来整合高速缓存。SET_PAGE_CACHE() 和 SET_PAGE_SLAB() 函数使用 page→list 中的 next 和 prev 字段来跟踪对象属于哪一个高速缓存或 slab。也可以通过宏 GET_PAGE_CACHE() 和 GET_PAGE_SLAB() 从页面中得到高速缓存描述符和 slab 描述符。这些关系如图 8.8 所示。

最后讨论保存 slab 描述符的位置。slab 描述符既可保存在 slab 中(CFLGS_OFF_SLAB 在静态标志位中设置)也可保存在 slab 外。具体保存的位置由创建高速缓存时对象的大小决定。但是 struct slab_t 可以存储在页面帧的首部,尽管图 8.8 中显示的 struct slab_t 和页面帧是分开的。

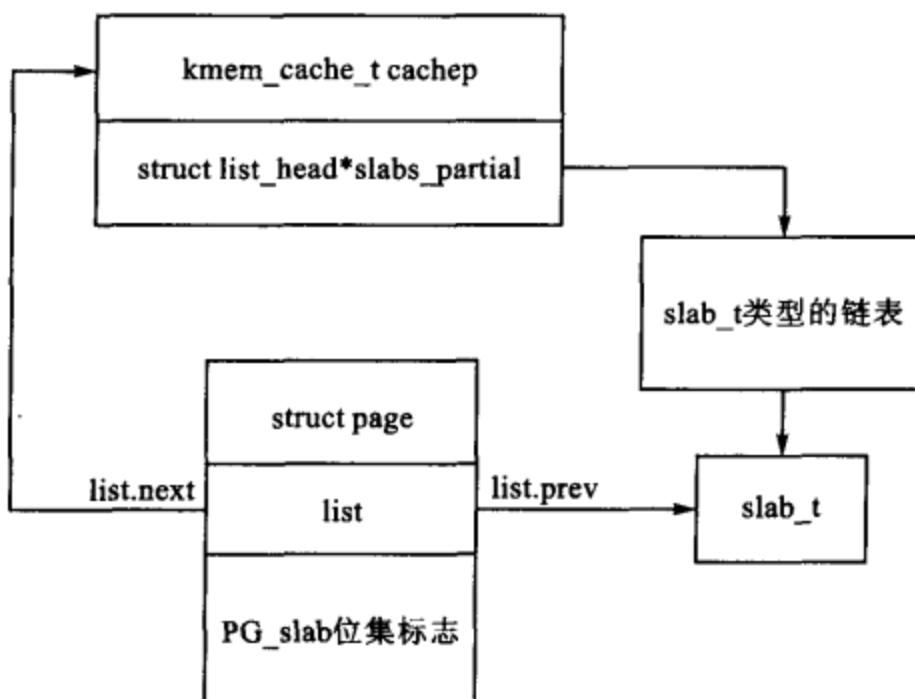


图 8.8 页面与高速缓存、slab 之间的关系

8.2.1 存储 slab 描述符

如果对象大于阈值(在 x86 架构中是 512 B),就需要设置高速缓存中的标志位 CFGS_OFF_SLAB,slab 描述符保存在 slab 之外的某个指定大小的高速缓存中(见 8.4 节)。所选择的指定大小的高速缓存要足够大,能够容纳 struct slab_t,在需要的时候,函数 kmem_cache_slabmgmt()从中进行分配。这将限制存储在 slab 中的对象数目,因为只有有限的空间分配给 bufctls 数组。然而当对象很大时,这就不重要了,因此没有必要将很多对象存储在单个 slab 中。

或者说,可以将 slab 管理器保存在 slab 的起始处。当保存在 slab 中时,在起始处要留有足够的空间以存储 slab_t 和无符号整形数组 kmem_bufctl_t。这个数组用于跟踪下一个可用空闲对象的索引,这将在 8.2.3 小节作进一步讨论。而实际的对象保存在 kmem_bufctl_t 数组之后。

图 8.9 可以帮助我们清楚地理解描述符在 slab 内部时的 slab 结构。图 8.10 描述了 slab 描述符保存在外部时,高速缓存如何用指定大小的高速缓存来存储 slab 描述符。

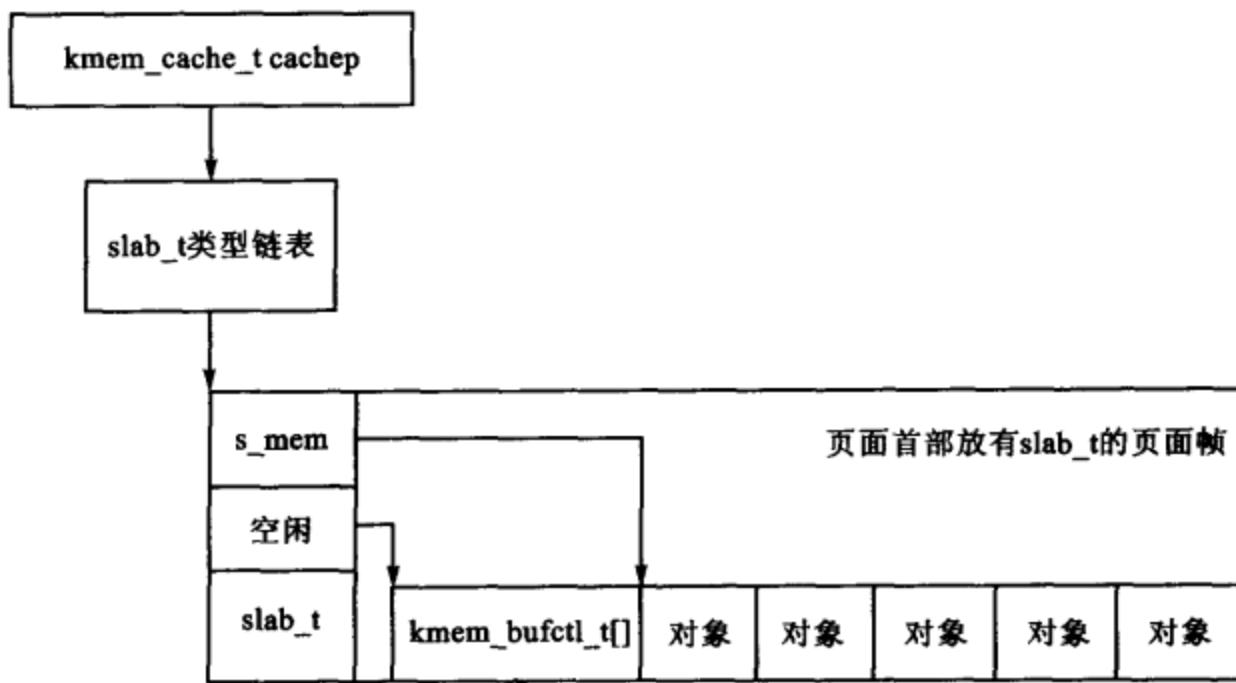


图 8.9 描述符存储在内部的 slab

8.2.2 创建 slab

在这里,我们已经看到怎样创建一个高速缓存,但在创建时,它是一个空的高速缓存,而且链表 `slab_full`, `slab_partial` 和 `slabs_free` 也都是空的。通过调用函数 `kmem_cache_grow()` 给高速缓存分配新的 slab,其调用图如图 8.11 所示。通常称之为“高速缓存增长”,当 `slab_partial` 链表中没有对象或者 `slabs_free` 链表中没有 slab 时调用该函数。所有的任务如下:

- 为了防止错误的调用,执行基本的全面检查。
- 计算 slab 中对象的着色偏移量。
- 为 slab 分配内存,并获取 slab 描述符。
- 把用于 slab 的页面链接起来放置到 slab 和高速缓存描述符中,见 8.2 节。
- 初始化 slab 中的对象。
- 将这个 slab 添加到高速缓存中。

8.2.3 跟踪空闲对象

slab 分配器有一个快速简单的方法来跟踪在部分填充的 slab 中的空闲对象。通过使用与每个 slab 管理器相关联的称为 `kmem_bufctl_t` 的无符号整形数组来达到这个目的。很明显,slab 管理器需要知道它的空闲对象在哪。

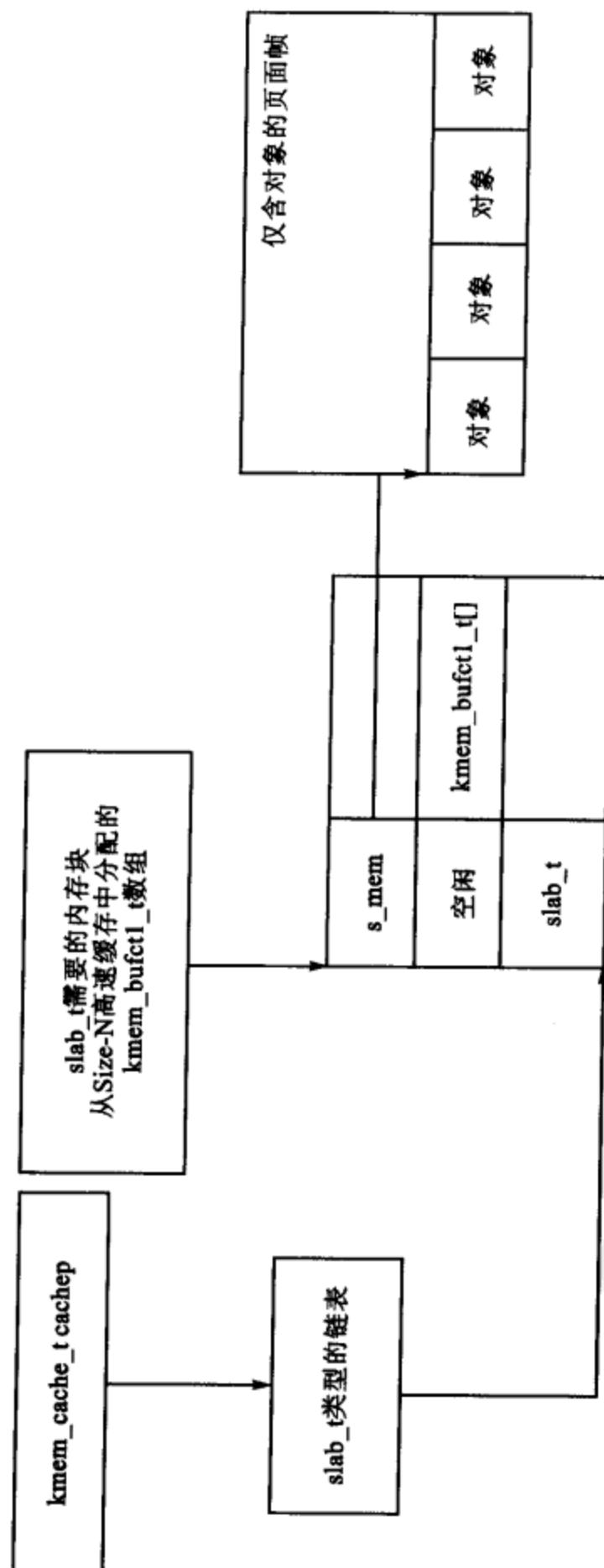


图 8.10 描述符存储在外部的 slab

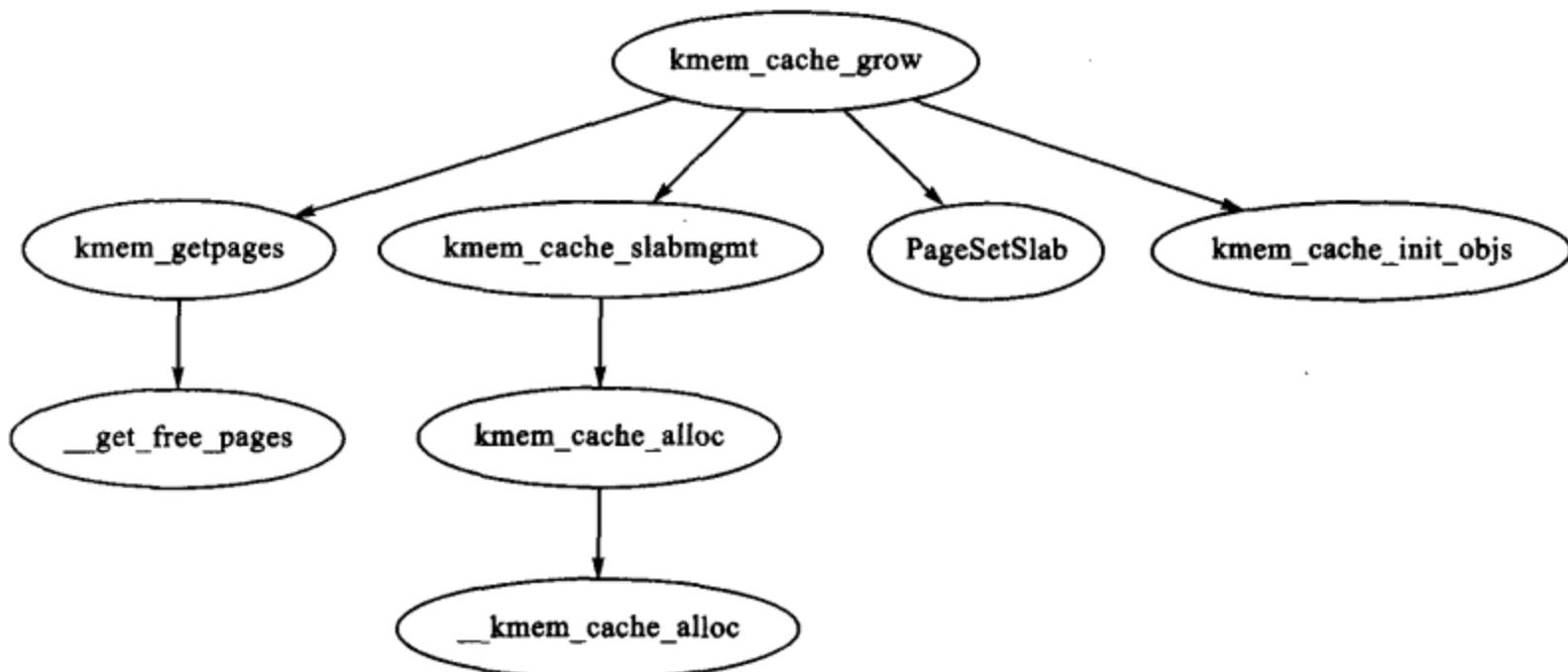


图 8.11 调用图: kmem cache grow()

参照以往描述 slab 分配器的文章[Bon94]，kmem_bufctl_t 是一个对象链表。在 Linux2.2.x 内核中，这个结构是个具有 3 项的联合体：一个指向下一个空闲对象的指针，一个指向 slab 管理器的指针，以及一个指向对象本身的指针。在联合中具体是哪一个字段取决于对象的状态。

现在，对象属于哪一个 slab 和高速缓存是由 struct page 决定的，而 kmem_bufctl_t 是一个简单的对象索引整型数组。数组中元素的个数与 slab 中对象的个数相同。

```
141 typedef unsigned int kmem_bufctl_t;
```

因为数组保存在 slab 描述符之后，并且没有指针直接指向第一个元素，所以需要提供辅助宏 slab_bufctl()。

```
163 #define slab_bufctl(slabp) \
164 ((kmem_bufctl_t *)(((slab_t *)slabp) + 1))
```

这个看起来很神秘的宏展开后却相当简单。参数 slabp 是一个指向 slab 管理器的指针。表达式((slab_t *)slabp)+1 将结构 slabp 转换成 slab_t 结构，然后加 1。这个操作实际上返回了指向数组 kmem_bufctl_t 起始地址的指针。(kmem_bufctl_t *) 又将这个指针转换为所需的类型。这个代码块的返回值是 slab_bufctl(slabp)[i]。或者说是，“通过指向 slab 描述符的指针，由宏 slab_bufctl() 进行偏移得到数组 kmem_bufctl_t 的起始地址，然后返回该数组中第 i 个元素”。

在 slab 里下一个空闲对象的索引存储在 slab_t→free 中，删除了为跟踪空闲对象而需要的链表。分配或者释放对象时，这个指针基于数组 kmem_bufctl_t 中的信息更新。

8.2.4 初始化 kmem_bufctl_t 数组

当高速缓存增长时, slab 中的对象和数组 kmem_bufctl_t 将被初始化。数组被填充每一个对象的索引, 从 1 开始, 并以标记 BUFCTL_END 结束。例如具有 5 个对象的 slab, 数组中的元素如图 8.12 所示。

数值 0 存储在 slab_t→free 中, 是因为第 0 个对象表示第一个被使用的空闲对象。对于给定的第 n 个对象, 其下一个空闲对象索引将被存储在 kmem_bufctl_t[n] 中。看着上面的数组, 0 之后的下一个空闲对象是 1。而 1 之后是 2, 依此类推。正如数组所使用的, 这样的排列使数组看起来更像一个空闲对象的后进先出队列(LIFO)。

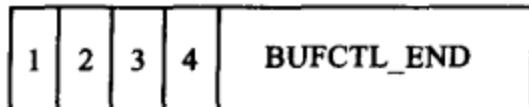


图 8.12 初始化数组 kmem_bufctl_t

8.2.5 查找下一个空闲对象

当分配一个对象时, kmem_cache_alloc() 调用 kmem_cache_alloc_one_tail() 执行更新数组 kmem_bufctl_t() 的实际动作。字段 slab_t→free 具有第一个空闲对象的索引。而下一个空闲对象的索引在 kmem_bufctl_t[slab_t→free] 中。它在代码中如下:

```
1253 objp = slabp->s_mem + slabp->free * cachep->objsize;
1254 slabp->free = slab_bufctl(slabp)[slabp->free];
```

字段 Slabp→s_mem 是指向 slab 中第一个对象的指针。slabp→free 是将被分配的对象的索引, 而且必须和一个对象的大小相乘。

下一个空闲对象的索引存储在 kmem_bufctl_t[slabp→free] 中。由于没有指针直接指向数组, 因此使用宏 slab_bufctl() 作为辅助手段。请注意数组 kmem_bufctl_t 在分配过程中并没有改变, 但是没有被分配的元素不能被访问。例如, 分配 2 个元素后, 数组 kmem_bufctl_t 中的索引 0 和 1 并没有被任何其他的元素所指向。

8.2.6 更新 kmem_bufctl_t

只有在函数 kmem_cache_free_one() 中释放一个对象时才需要更新数组 kmem_bufctl_t。数组更新的代码如下:

```
1451 unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;
1452
```

```

1453 slab_bufctl(slabp)[objnr] = slabp->free;
1454 slabp->free = objnr;

```

指针 `objp` 指向将要被释放的对象, 而 `objnr` 是其索引。`kmem_bufctl_t[objnr]` 被更新为指向 `slabp->free` 的当前值, 通过字段 `free`, 有效地替换了虚拟链表上的对象。`slabp->free` 被更新为即将被释放的对象, 因此它也会是下一个被分配的对象。

8.2.7 计算 slab 中对象的数量

创建高速缓存时, 系统通过调用 `kmem_cache_estimate()` 函数计算在单个的 slab 上可以存放多少个对象。这要考慮 slab 描述符是存储在 slab 之内还是之外, 并且跟踪每一个 `kmem_bufctl_t` 的大小, 无论对象是否空闲。该函数返回可能存储的对象个数以及浪费的字节数。若使用高速缓存着色, 则浪费的字节数就非常可观。

基本的计算步骤如下:

- 初始化 `wastage` 为整个 slab 的大小, 例如 `PAGE_SIZEgfp_order`。
- 减去 slab 描述符所占用的空间。
- 计算可能存储的对象个数。如果 slab 描述符存储在 slab 之内, 还要考虑 `kmem_bufctl_t` 的大小。持续增加 i 的大小直至 slab 被填充。
- 返回对象的个数和浪费的字节数。

8.2.8 销毁 slab

在收缩、销毁高速缓存时, 系统将删除其中的 slab。由于对象可能有析构函数, 如果这样它们就必须被调用, 因此该功能的任务如下:

- 如果析构函数可用的话, 调用 slab 中所有对象的析构函数。
- 如果允许调试, 检查红色标志位和感染模式。
- 释放 slab 使用的页面。

函数调用图如图 8.13 所示, 非常简单。

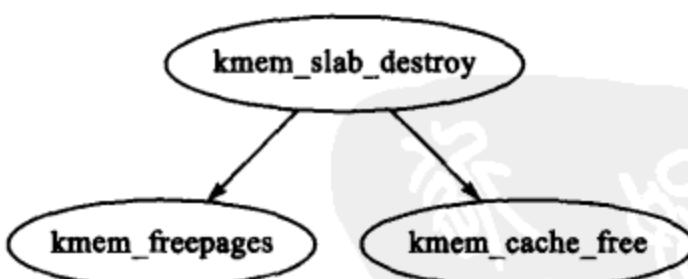


图 8.13 调用图: `kmem_slab_destroy()`

8.3 对象

这一节主要内容是如何管理对象。大多数真正需要做的工作已经被高速缓存或 slab 管理器完成。

8.3.1 初始化 slab 中的对象

当创建一个 slab 时,所有的对象都处于初始化状态。如果构造函数可用,它将为每一个对象所调用,而一旦对象被释放,就认为它处于初始化状态。从概念上讲,这个初始化过程很简单。遍历所有的对象,调用构造函数,并为它初始化 kmem_bufctl。函数 kmem_cache_init_objs()负责初始化对象。

8.3.2 分配对象

函数 kmem_cache_alloc()负责分配一个对象给调用者,这在 UP 和 SMP 系统中稍微有所不同。在 SMP 系统中分配对象时的基本函数调用图如图 8.14 所示。

有 4 个基本步骤:第 1 步(kmem_cache_alloc_head())进行基本的检查,确保允许分配。第 2 步选择从哪一个 slab 链表中分配对象。slabs_partial 或者 slabs_free 中的任一个都有可能。如果在 slabs_free 中没有,就增大高速缓存空间(见 8.2.2 小节)并在 slabs_free 中创建一个新的 slab。最后一步从选中的 slab 中分配对象。

在 SMP 系统中,还有一步,在分配对象前,系统会检查 per-CPU 高速缓存并查找一个可用的高速缓存,如果有的话,则使用它。如果没有,系统在会成批地分配对象的 batchcount 数量并置于它们自己的 per-CPU 高速缓存中。关于 per-CPU 高速缓存更多的信息见 8.5 节。

8.3.3 释放对象

函数 kmem_cache_free()的调用图如图 8.15 所示,它用于释放对象,这是一个相对简单的任务。跟 kmem_cache_alloc()函数一样,它在 UP 以及 SMP 中表现不一样。在这两个系统中主要的区别是,在 UP 中,对象直接返回给 slab;而在 SMP 中,对象却返回给 per-CPU 高速缓存。但在这两种情况下,如果对象有可用的析构函数的话,则函数都会被调用。析构函数负责把对象状态返回到初始化状态。

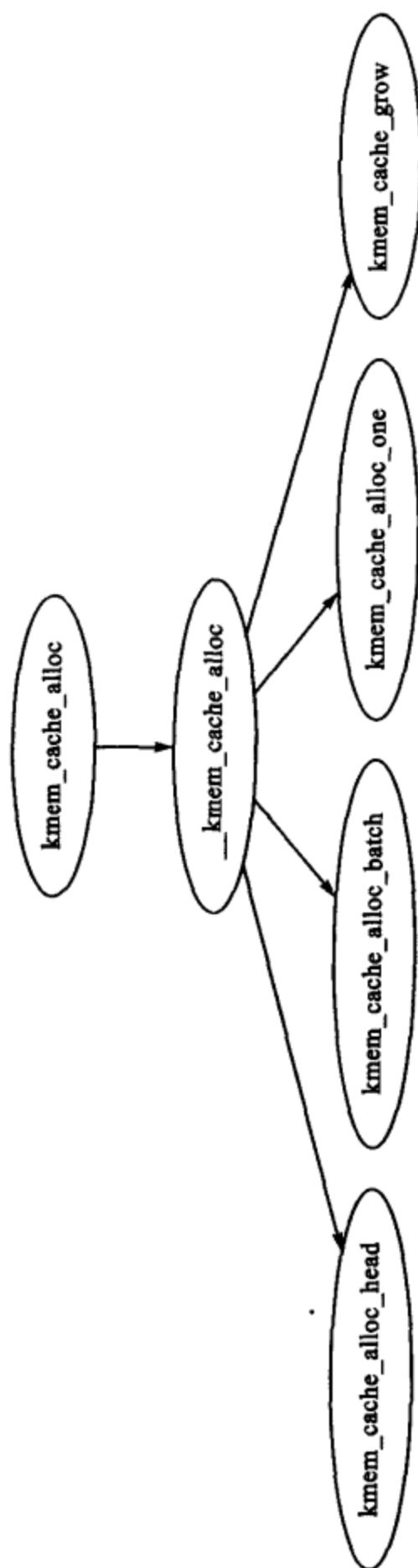


图 8.14 调用图：kmem_cache_alloc()

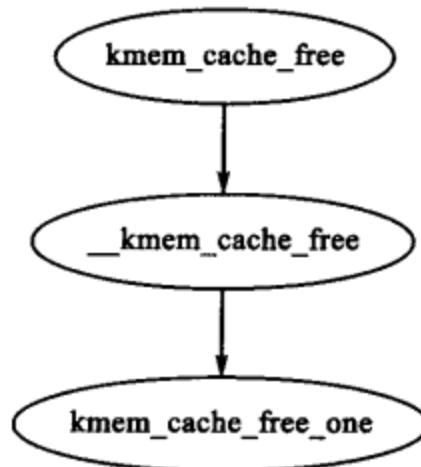


图 8.15 调用图: kmem_cache_free()

8.4 指定大小的高速缓存

对于小块内存分配,由于物理页面分配器不再适用,所以 Linux 保留有两套高速缓存系统。一套高速缓存由 DMA 使用,而另一套在通常情况下使用。人们称呼它们为 size-N 高速缓存和 size-N(DMA) 高速缓存,可以通过/proc/slabinfo 看到。每种指定大小的高速缓存的信息都存放在一个 struct cache_sizes 中,它们也具有 cache_sizes_t 的类型,在 mm/slab.c 中定义如下:

```

331 typedef struct cache_sizes {
332     size_t cs_size;
333     kmem_cache_t * cs_cachep;
334     kmem_cache_t * cs_dmacachep;
335 } cache_sizes_t;
  
```

该结构中各字段的含义描述如下。

cs_size: 内存块的大小。

cs_cachep: 常规内存使用的高速缓存块。

cs_dmacachep: DMA 使用的高速缓存块。

由于这些高速缓存的数量是有限的,所以在编译时期系统会初始化一个静态数组 cache_sizes。它的值在 4 KB 机器上以 32 B 作为起始,如果页面更大,它将会从 64 开始。

```

337 static cache_sizes_t cache_sizes[] = {
338 #if PAGE_SIZE == 4096
339 { 32, NULL, NULL },
340#endif
  
```

```

341 { 64, NULL, NULL},
342 { 128, NULL, NULL},
343 { 256, NULL, NULL},
344 { 512, NULL, NULL},
345 { 1024, NULL, NULL},
346 { 2048, NULL, NULL},
347 { 4096, NULL, NULL},
348 { 8192, NULL, NULL},
349 { 16384, NULL, NULL},
350 { 32768, NULL, NULL},
351 { 65536, NULL, NULL},
352 {131072, NULL, NULL},
353 { 0, NULL, NULL}

```

很显然,这是一个以 0 结尾的数组,它从 2^5 到 2^{17} 由 2 的指数个缓冲区组成。当系统启动时,这个数组就必须被初始化,然后用于描述每一种指定大小的高速缓存。

8.4.1 kmalloc()

由于存在指定大小的高速缓存,slab 分配器可以提供一个新的分配函数 kcalloc(),每当需要较少内存缓冲区时就调用该函数。当系统接收到一个这样的请求时,系统会选择适当的指定大小的缓冲区,并且从它当中分配一个对象。因为大部分困难的工作在分配高速缓存时已经完成,所以如图 8.16 所示的调用过程非常简单。

8.4.2 kfree()

既然有一个 kcalloc() 函数用来分配小块内存对象,那么就有一个 kfree() 用来释放小块内存对象。和 kcalloc() 函数一样,实际的工作在对象释放的时候完成(见 8.3.3 小节),所以如图 8.17 所示的函数调用图也很简单。

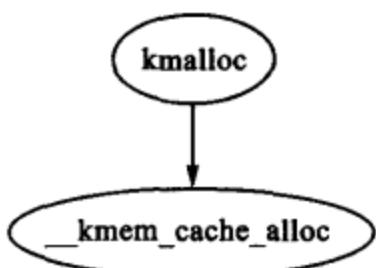


图 8.16 调用图: kcalloc()

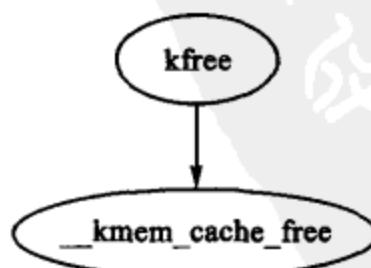


图 8.17 调用图: kfree()

8.5 per-CPU 对象高速缓存

slab 分配器致力于的一个任务就是提升硬件和高速缓存的使用效率。一般来讲,高性能计算[CS98]的一个目标就是尽可能长时间地使用同一个 CPU 上的数据。Linux 通过 per-CPU 尝试将对象保留在同一个 CPU 高速缓存中来实现这个任务, per-CPU 高速缓存在系统中可以简单地称为每个 CPU 的 cpucache。

当对象被分配或者被释放时,它们都被放置在 cpucache 中。在没有空闲对象时,系统就将一批对象放置到这个池中。在池变得过于庞大时,系统就移除其中一半的对象,放到全局高速缓存中。这样,就可以尽可能长时间地使用同一个 CPU 上的硬件高速缓存。

这种方法的第二个好处是在访问 CPU 池时不需要获得自旋锁,这是因为我们已经保证不会有另外的 CPU 访问这些局部数据。这点很重要,如果没有高速缓存,那么就必须在每次分配和释放时都要获得自旋锁,这是不必要的开销。

8.5.1 描述 per-CPU 对象高速缓存

每一个高速缓存描述符都有一个指针指向一个 CPU 高速缓存数组,它在高速缓存中描述如下:

```
231 cpucache_t * cpudata[NR_CPUS];
```

该结构非常简单:

```
173 typedef struct cpucache_s {
174     unsigned int avail;
175     unsigned int limit;
176 } cpucache_t;
```

各字段如下。

avail: 在 cpucache 中可用的空闲对象的数量。

limit: 能够存在的空闲对象的总数量。

对于一个给定的高速缓存和处理器,系统为 cpucache 提供了一个 cc_data() 宏作为辅助,它的定义如下:

```
180 #define cc_data(cachep) \
181 ((cachep)->cpudata[smp_processor_id()])
```

该宏需要一个给定的高速缓存描述符(cachep)作为输入参数,然后从 cpucache 数组(cpu-data)中返回一个指针。所需的索引是当前处理器的 ID, 即 smp_processor_id()。

很快系统就会在 cpucache_t 结构后面存储 cpucache 中的对象。这与在 slab 描述符后面存储对象类似。

8.5.2 在 per-CPU 高速缓存中添加/移除对象

为防止碎片的产生,通常是在数组的末尾进行添加和删除对象的操作。以下代码块用于添加对象(obj)到 CPU 高速缓存(cc)中:

```
cc_entry(cc)[cc->avail ++] = obj;
```

以下代码块用于移除对象:

```
obj = cc_entry(cc)[--cc->avail];
```

辅助宏 cc_entry()给出了 CPU 高速缓存中指向第一个对象的指针,它的定义如下:

```
178 #define cc_entry(cpucache) \
179 ((void **)(((cpucache_t *)cpucache) + 1))
```

它用一个指针指向 cpucache,并根据 cpucache_t 描述符的大小增加其值,从而得到高速缓存中第一个对象。

8.5.3 启用 per-CPU 高速缓存

在创建一个高速缓存时,就必须启动 CPU 高速缓存,并调用 kmalloc()函数为之分配内存。函数 enable_cpucache()负责决定 per-CPU 高速缓存的大小,并调用 kmem_tune_cpucache()函数为该高速缓存分配空间。

显然,各种指定大小的高速缓存被启动后,CPU 高速缓存就不能再存在了,因此,系统使用全局变量 g_cpucache_up 来防止 CPU 高速缓存被过早地启用。函数 enable_all_cpucaches()遍历高速缓存链表中所有的高速缓存,并逐个启动各自的 cpucache。

一旦启用了 CPU 高速缓存,不需要加锁就可以直接访问了,因为 CPU 决不会访问到错误的 cpucache,所以可以保证访问的安全性。

8.5.4 更新 per-CPU 高速缓存的信息

per-CPU 高速缓存被创建或被修改后,每一个 CPU 通过 IPI 得知该情况。这时若没有改变

高速缓存描述符中所有的值,就可能导致高速缓存的信息不一致,因此就必须用加锁的方式来保护CPU高速缓存中的信息。所以,Linux中提供了一个ccupdate_t结构,它里面包含了每一个CPU所需要的信息,并且每一个CPU在这个高速缓存描述符中通过旧信息交换新数据。用于存储新cpucache信息的结构定义如下:

```
868 typedef struct ccupdate_struct_s
869 {
870 kmem_cache_t *cachep;
871 cpucache_t *new[NR_CPUS];
872 } ccupdate_struct_t;
```

其中的cachep是已经更新过的高速缓存,new是系统中每个CPU的cpucache描述符的数组。函数smp_function_all_cpus()取得每个CPU,然后调用do_ccupdate_local()函数将ccupdate_struct_t中的信息与高速缓存描述符中的信息进行交换。

一旦这些信息被交换后,旧的数据就可以删除了。

8.5.5 清理 per-CPU 高速缓存

收缩一个高速缓存时,第一步就是调用drain_cpu_caches()来清理对象可能具有的cpucache。在这里,slab分配器必须清楚地知道可以释放哪些slab,这一点很重要,因为即便在per-CPU高速缓存的slab中只有一个对象,这整个slab也不能被释放。在系统内存紧张时,没有必要在分配操作上节约几毫秒的时间。

8.6 初始化 slab 分配器

这一节描述slab分配器如何初始化它自己。在slab分配器创建一个新的高速缓存时,它就从cache_cache或者kmem_cache高速缓存中分配kmem_cache_t结构。这显然是一个是有鸡还是先有蛋的问题,所以必须静态地初始化cache_cache,如下:

```
357 static kmem_cache_t cache_cache = {
358 slabs_full: LIST_HEAD_INIT(cache_cache.slabs_full),
359 slabs_partial: LIST_HEAD_INIT(cache_cache.slabs_partial),
360 slabs_free: LIST_HEAD_INIT(cache_cache.slabs_free),
361 objsize: sizeof(kmem_cache_t),
362 flags: SLAB_NO_REAP,
363 spinlock: SPIN_LOCK_UNLOCKED,
```

```

364 colour_off: L1_CACHE_BYTES,
365 name: "kmem_cache",
366 };

```

静态初始化 `kmem_cache_t` 结构的代码如下。

358~360 初始化这 3 个链表成空链表。

361 每个对象的大小是一个高速缓存描述器的大小。

362 高速缓存的创建和删除是非常少见的,因此并不用考虑它的回收。

363 初始化自旋锁为开锁状态。

364 对象和 L1 高速缓存对齐。

365 记录可读性好的命名。

编译时,系统会计算所有这些静态定义字段的代码。为了初始化结构的其余部分,函数 `start_kernel()` 会调用 `kmem_cache_init()`。

8.7 伙伴分配器接口

slab 分配器并不拥有页面,它必须通过物理页面分配器为其分配页面。为此,系统提供了两个 API 函数 `kmem_getpages()` 和 `kmem_freepages()`。这两个函数基本上都是对伙伴分配器的 API 封装,因此在分配时要考虑 slab 标志位。分配时,缺省页面从 `cachep->gfpflags` 得到,而其顺序从 `cachep->gforder` 得到,其中 `cachep` 是请求页面的高速缓存。在释放页面时, `PageClearSlab()` 会在调用 `free_pages()` 之前由于每个即将释放的页面而被调用。

8.8 2.6 中有哪些新特性

最明显的改变是/`proc/slabinfo` 格式的版本由 1.1 升级到 2.0,这样更具可读性。最有帮助的改变是现在字段都有了一个首部,省去了记忆每栏意思的必要。

主要的算法思想以及概念与原来一样。虽然主要的算法没有变,但是算法的实现却很不同。尤其是,2.6 中特别强调使用 per-CPU 对象以避免上锁操作。其二,2.6 中包含了大量的调试代码,所以在阅读代码时就需要略过那些`#ifdef DEBUG` 的部分。最后,对一些函数名做了字面意义上的修改,其实它们的行为依旧没变。例如, `kmem_cache_estimate()` 现在称为 `cache_estimate()`,其实它们除了名字什么都是一样的。

高速缓存描述符

对 `kmem_cache_s` 的改变很少。首先,在这个结构的起始位置记录常用的元素,如 per-

CPU 相关的数据(原因见 3.9 节)。其次,slab 链表(如 `slabs_full`)以及与其相关的统计数据都被转移到了另一个单独的 `struct kmem_list3` 中。注释以及不常用的宏表明将有计划使该结构对应于单个的节点。

高速缓存静态标志位

在 2.4 中的这些标志位仍然存在,他们的使用方法也是相同的。`CFLAGS_OPTIMIZE` 不再存在了,它在 2.4 中就没有使用过。2.6 中还引入了两个新标志位。

`SLAB_STORE_USER`: 这是只用于调试的标志位,它记录释放对象的函数。如果对象在释放后被使用,那么感染标志位的字节就不会匹配,系统将显示一个内核错误消息。由于知道最后一个使用对象的函数,所以调试更轻松。

`SLAB_RECLAIM_ACCOUNT`: 这个标志位由具有易回收对象的高速缓存所使用,如索引节点高速缓存。系统在一个称为 `slab_reclaim_pages` 的变量中设置了一个计数器,用于记录 slab 在这些高速缓存中分配了多少页面。这个计数器在后面用于 `vm_enough_memory()`,来帮助决定系统是否真的内存溢出。

回收高速缓存

这是对 slab 分配器最有意思的改变。不再存在 `kmem_cache_reap()` 是因为在用户面对更为长远的选择时,该函数不加选择地决定如何缩小高速缓存。高速缓存的用户现在可以使用 `set_shrinker()` 注册一个收缩高速缓存的回调函数来实现智能计数和收缩 slab。这个简单的函数生成一个 `struct shrinker`,在这个结构中有一个指向回调函数的指针和一个搜寻权重,这个权重表明了在把对象放入称为 `shrinker_list` 的链表之前重建对象的困难程度。

在页面回收时,系统调用函数 `shrink_slab()`,它遍历整个 `shrinker_ist`,并调用每个收缩回调函数两次。第 1 次调用传一个 0 作为参数,它表明如果函数被适当调用了,回调函数应该返回它希望自己能够释放的页面数量。系统对各回调函数的代价作了基本的试探以决定是否值得使用该回调函数。如果值得,它将第 2 次作为参数被调用,以表明有多少对象被释放。

计算释放页面数量的机制是有技巧的。每个任务结构中有个称为 `reclaim_state` 的字段。在 slab 分配器释放页面时,就用被释放的页面数量更新该字段。在调用 `shrink_slab()` 之前,该字段设置为 0,在 `shrink_cache` 返回以决定释放多少页面之后,系统将再次读取该字段。

其他的改变

其他的改变实际上都是表面上的。例如,slab 描述器现在称为 `struct slab` 而不是 `slab_t`,这和目前逐步不使用 `typedefs` 的趋势相一致。per-CPU 高速缓存除了结构以及 API 有新的命名以外基本保持不变。相同类型的变化在大多数 2.6 内核的 slab 分配器中都实现了。

第 9 章

高端内存管理

内核只有在设置了页表项后才能直接定址内存。在大多数情况下,用户/内核地址空间分别分成 3 GB/1 GB。这就意味着,正如在 4.1 节中解释的那样,在一台 32 位的机器上至多只有 896 MB 的内存可以直接被访问。在一台 64 位的机器上,因为有足够的虚拟地址空间,所以这实际上不是个问题。现在还不可能有运行 2.4 内核的机器具有 TB 级的 RAM。

现在已经有很多高端的 32 位机器拥有 1 GB 的内存,因此,就不能简单地忽略不方便定址的那部分内存。Linux 使用的方法是暂时把高端内存映射为较低的页表。9.2 节中将会讨论。

高端内存与 I/O 相关的一个问题必须提到,即并不是所有的设备都可以访问高端内存或者对 CPU 可用的所有内存。这可能是这样一种情况,如果 CPU 开启了 PAE 拓展,那么设备就被限制在只能访问有符号 32 位整数的空间(2 GB)或者是 64 位架构中所使用的 32 位设备。控制设备写内存,最好的情形是内存失效,而最坏的情形可能使内核崩溃。解决这个问题的方法是使用弹性缓冲区,这个将在 9.5 节中讨论。

这一章首先简要地描述如何管理持久内核映射(PKMap)地址空间,然后讨论页面如何映射到高端内存以及如何解除映射。接下来的小节将先讨论哪个地方的映射必须是原子性的,然后深入讨论弹性缓冲区。最后我们将谈到在内存非常紧张时,如何使用紧急池。

9.1 管理 PKMap 地址空间

在内核页表顶部,从 PKMAP_BASE 到 FIXADDR_START 的空间保留给 PKMap。这部分保留的空间大小变化很细微。在 x86 上,PKMAP_BASE 位于 0xFE000000,而 FIXADDR_START 的地址是一个编译时常量,它只随着配置选项而变化,但一般是只有一少部分页面被分配在线性地址空间的尾部附近。这就意味着把页面从高端内存映射到可用空间的页表

空间略小于 32 MB。

为了映射页面,单个页面集的 PTE 被存放在 PKMap 区域的起始部分,从而允许 1 024 个页面在短期内通过函数 `kmap()` 映射到低端内存,接着由 `kunmap()` 解除映射。这个池看起来虽然很小,但同时 `kmap()` 映射页面的时间却也非常短。代码的注解表明了有计划进行分配连续的页表项以扩充这部分区域,但现在保留的仍只是代码注释,所以大部分 PKMap 的部分没有被用到。

`kmap()` 调用中用到的页表项称为 `pkmap_page_table`,它位于 `PKMAP_BASE`,并且在系统初始化时就建立。在 x86 上,它是在 `pagetable_init()` 函数结束部分进行的。含有 PGD 和 PMD 项的页面由引导内存分配器分配,以保证它们的存在性。

当前页表项的状态由一个称为 `pkmap_count` 的简单数组管理,它具有 `LAST_PKMAP` 项。在没有 PAE 的 x86 系统中,它的值是 1 024,而在具有 PAE 的系统中,它的值是 512。更准确地说,虽然在代码中没有体现出来, `LAST_PKMAP` 变量的值等于 `PTRS_PER_PTE`。

虽然每个元素不是准确的引用计数,但是那也很接近引用计数。如果该项是 0,则该页空闲,而且自上次 TLB 刷新后就没被使用过。如果是 1,则该槽没有被使用,但是仍有页面映射该槽,等待一次 TLB 刷新。因为当全局的页表修改时需要对所有的 CPU 进行刷新,而这个操作的开销是相当大的,所以每个槽都被使用至少一次以后才会进行刷新。任意更高的数值都是对该页面 $n-1$ 个用户的引用计数。

9.2 映射高端内存页面

表 9.1 中描述了从高端内存映射页面的 API。主要用于映射页面的函数是 `kmap()`,其调用图见图 9.1。如果用户不想阻塞,那么可以使用 `kmap_nonblock()`,而中断用户可以使用 `kmap_atomic()`。`kmap` 池相当小,所以 `kmap()` 调用者尽可能快地调用 `kunmap()` 就显得很重要,因为与低端内存相比,这个小窗口的压力随着高端内存的增大而变得更为严重。

`kmap()` 函数本身非常简单。它首先检查一下以保证中断没有调用这个函数(因为它可能睡眠),然后在确认的情况下调用 `out_of_line_bug()`。由于调用 `BUG()` 中断处理程序有可能使系统瘫痪,所以 `out_of_line_bug()` 输出 bug 信息并干净地退出。接着要检查的是页面是不是在 `highmem_start_page` 以下,因为低于该标记位的页面已经是可见的了,而且不需要映射。

接着要检查的是该页面是否已经在低端内存,如果是就只返回该页面的地址。这样,如果该页面正处于低端内存,`kmap()` 的调用者将会无条件地知道这一情况,所以这个函数总是安全的。如果要映射的页面是高端内存页面,那就调用 `kmap_high()` 来完成实际的工作。

表 9.1 高端内存映射/解除映射 API

| | |
|---|---|
| void * kmap(struct page * page) | 从高端内存中取一个页面结构,并将它映射到低端内存。返回映射的虚拟地址 |
| void * kmap_nonblock(struct page * page) | 与 kmap() 功能相同,但如果没可用的槽,它就会返回 NULL 而不会阻塞。这与 kmap_atomic() 不同,因为 kmap_atomic() 使用特别保留的槽 |
| void * kmap_atomic(struct page * page, enum km_type type) | 在映射中保留有用于中断的原子槽(见 9.4 节)。注意不要使用它们,因为这个函数的调用者可能不会睡眠或调度。在某种原因下,这个函数从高端内存原子性的映射一个页面 |

kmap_high() 函数一开始就检查 page→virtual 字段,该字段在页面已经被映射时设置。如果该字段为 NULL,那么由 map_new_virtual() 提供页面的一个映射。

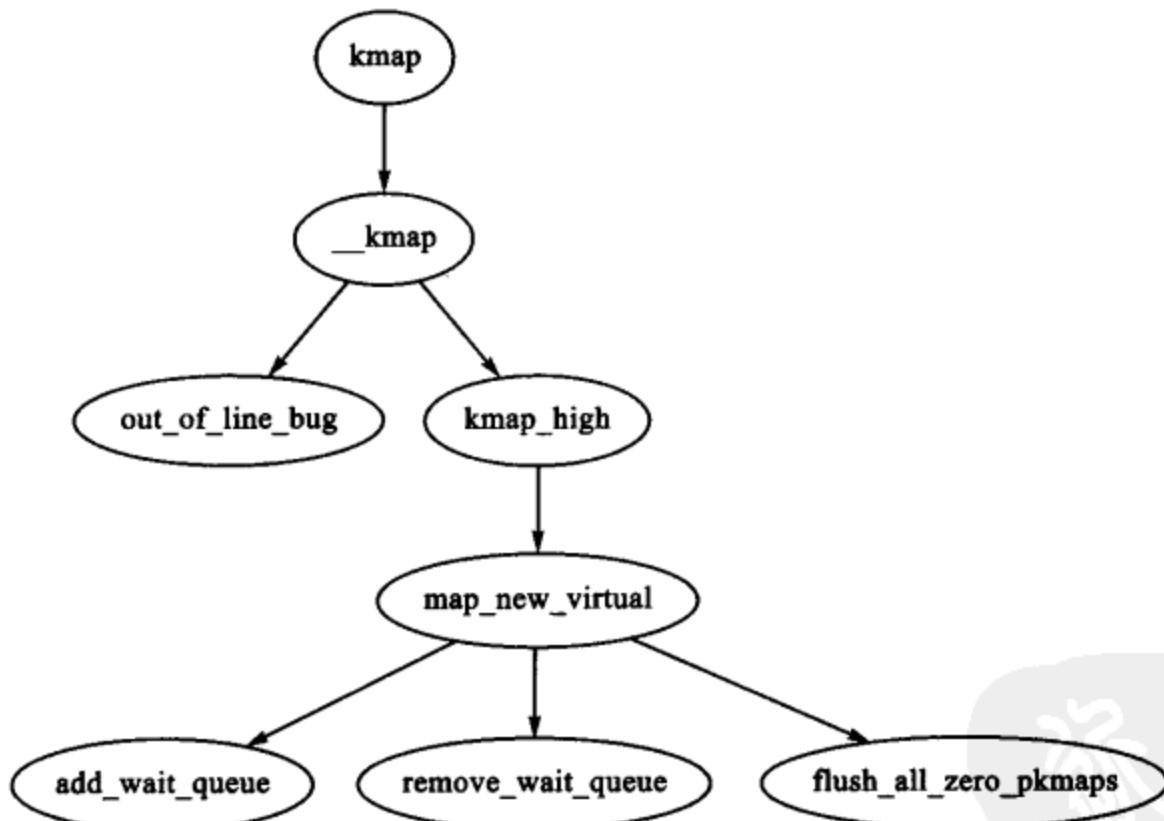


图 9.1 调用图: kmap()

map_new_virtual() 通过简单的线性扫描 pkmap_count 来创建新的虚拟映射。为了避免在几次 kmap() 调用间隔之间重复搜索同一个区域,因此扫描不是在 0 处开始的,而是开始于 last_pkmap_nr 处。当 last_pkmap_nr 折返到 0, 系统就会调用 flush_all_zero_pkmaps() 将所有项从 1 设为 0, 然后刷新 TLB。

如果在某次扫描完成后,依旧找不到某项,则该进程就会在 pkmap_map_wait 队列上睡

眠,直至它被下一次 `kunmap()` 唤醒。

映射创建之后, `pkmap_count` 数组中的相应项就会增 1, 然后返回低端内存的虚拟地址。

9.3 解除页面映射

解除高端内存页面映射的 API 如表 9.2 所列。`kunmap()` 函数就像它的辅助物一样, 执行两次检查。第 1 次做与 `kmap()` 相似的检查, 用于检查中断上下文。第 2 次检查页面是否处于 `highmem_start_page` 以下。如果是, 则该页面已经在低端内存中, 不需要做进一步的处理。如果确定该页面要被解除映射, 就调用 `kunmap_high()` 进行解除映射的操作。

表 9.2 解除高端内存页面映射的 API

| | |
|--|-------------------------------|
| <code>void kunmap(struct page * page)</code> | 从低端内存中解除一个页表结构的映射, 并释放映射它的页表项 |
| <code>void kunmap_atomic(void * kvaddr, enum km_type type)</code> | 解除原子性映射的页面 |

`kunmap_high()` 的原理很简单。它把 `pkmap_count` 中该页面的相应元素减 1。如果减到了 1(记住这意味着没有更多的用户了, 需要一次 TLB 刷新), 所有在 `pkmap_map_wait` 队列上等待的进程都会被唤醒, 因为现在有一个槽可用。但这时并不把该页面从页表上解除映射, 因为解除映射需要一次 TLB 刷新。它会一直延迟到 `flush_all_zero_pkmaps()` 被调用。

9.4 原子性的映射高端内存页面

虽然并不推荐使用 `kmap_atomic()`, 但是页面槽都会留给每个 CPU 以备需要, 比如弹性缓冲区, 在中断时供设备使用。一种结构是需要很多不同数量的原子性映射页面操作的, 这种数量由 `km_type` 枚举。可用的总数是 `KM_TYPE_NR` 标志位。在 x86 上, 共有 6 种不同的原子 `kmap` 使用方法。

在引导时, 在地址 `FIX_KMAP_BEGIN` 和 `FOX_KMAP_END` 之间, 系统会为每个处理器保留 `KM_TYPE_NR` 个项。显然在调用 `kunmap_atomic()` 之前, 一个 `kamp` 原子操作的调用者可能不会睡眠或退出, 这是因为同一处理器上的下一个进程可能尝试使用同一个项, 但却会失败。

在所请求类型的操作和处理机的页表中, 把请求页面映射到槽时, 函数 `kmap_atomic()` 的任务很简单。函数 `kunmap_atomic()` 的调用图如图 9.2 所示, 该函数很有意思, 因为它仅仅是

在启用调试项时调用 `ptr_clear()` 清理 PTE。一般认为不需要解除原子页面映射,也不需要作 TLB 刷新操作,因为下一次 `kmap_atomic()` 调用会替换掉这些原子页面。

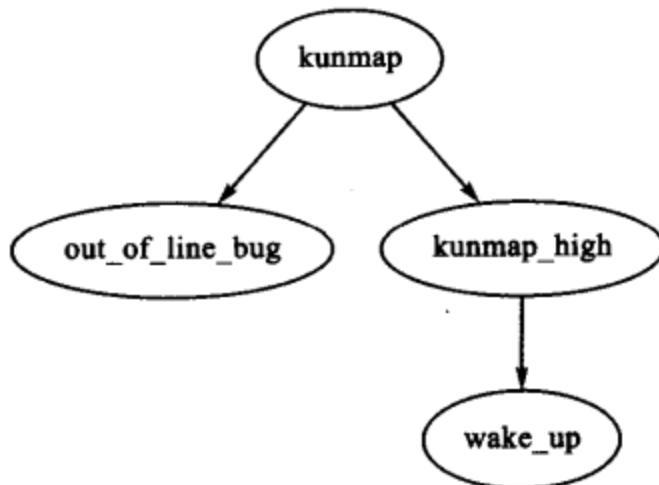


图 9.2 调用图: `kunmap()`

9.5 弹性缓冲区

对那些不能访问所有 CPU 可见的所有内存的设备而言,弹性缓冲区是必需的。一个很明显的例子是不能寻址和 CPU 一样多的位数的设备,如在 64 位结构上的 32 位设备,或最近的允许 PAE 的 Intel 处理器。

它的基本的思想很简单。弹性缓冲区驻留在足够低端的内存中,从而使设备从里面复制数据以及向里面写数据。然后它会把相应的用户页面复制到高端内存。这种附加的复制虽然是不合理的,但它又是不可或缺的。系统首先在低端内存中分配页面供 DMA 和设备的缓冲区页面。然后在 I/O 完成时由内核把它们复制到高端内存的缓冲区中,所以弹性缓冲区扮演着一个中介的角色。这种复制操作有一些负担,因为它至少涉及复制整个页面,但是与换出低端内存中的页面操作比较起来,这些负担又是无关紧要的。

9.5.1 磁盘缓冲

磁盘块,一般为 1 KB 的大小,并且都装配成页面由 slab 分配器分配的 `struct buffer_head` 管理。缓冲区首部的用户具有一个回调函数选项。该回调函数在 `buffer_head` 中注册为 `buffer_head->b_end_io()`,它在 I/O 完成时调用。这就是弹性缓冲区用于完成把数据复制出弹性缓冲区的机制。注册的回调函数叫 `bounce_end_io_write()`。

缓冲区首部的其他特征以及磁盘块层如何使用它们已经超出了本书的范围,它们更多的是 I/O 层所关心的。

9.5.2 创建弹性缓冲区

如图 9.3 所示, 创建弹性缓冲区是一件简单的事, 它从函数 `create_bounce()` 开始。它的原理很简单, 使用一个所提供的缓冲区首部作为模板来创建一块新的缓冲区。该函数有两个参数, 它们是 `read/write` 参数(`rw`)和所使用的缓冲区首部的模板(`bh_orig`)。

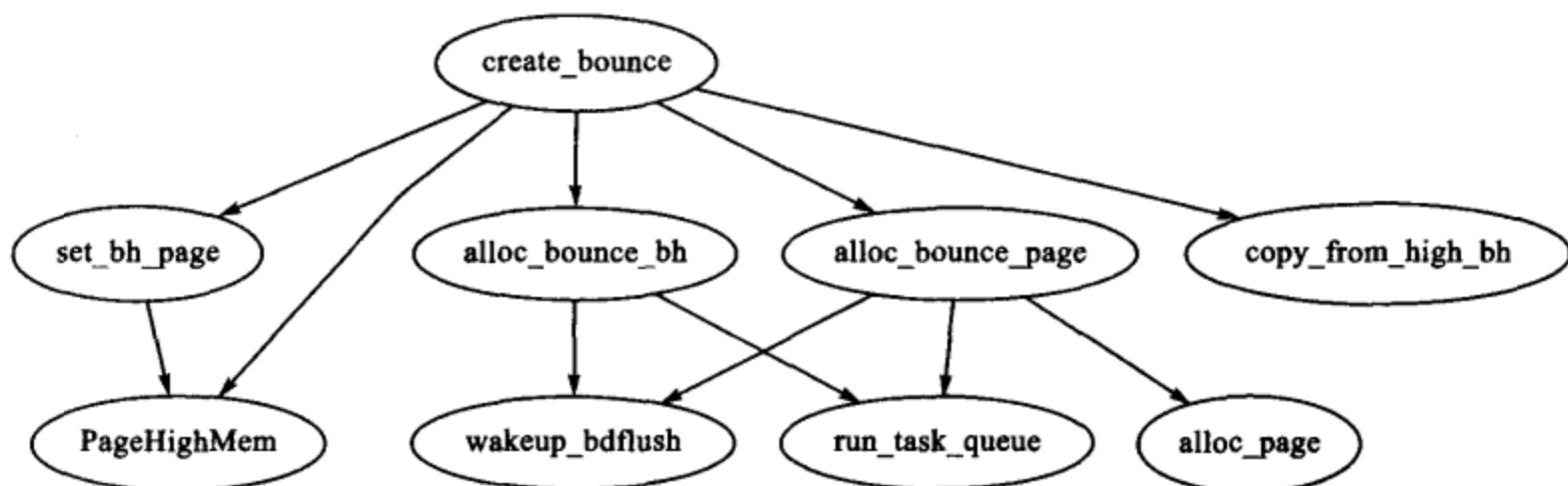


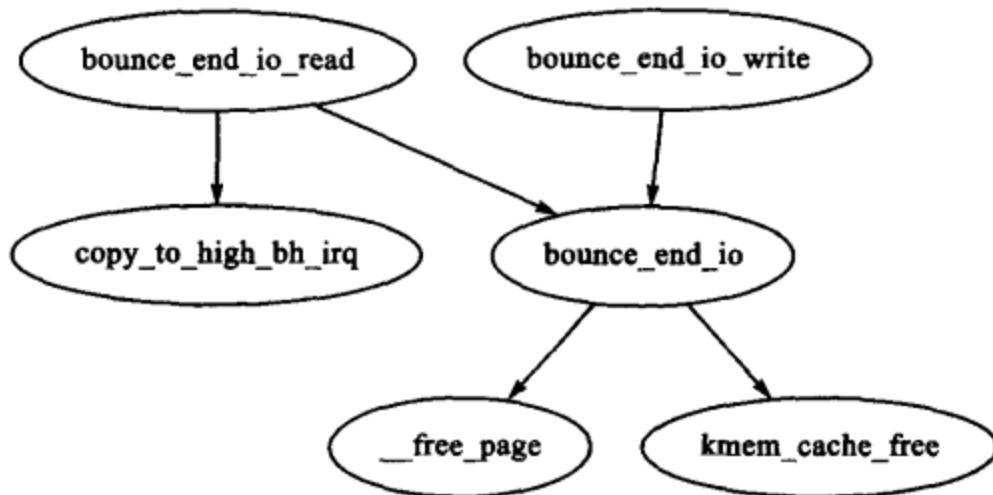
图 9.3 调用图: `create_bounce()`

函数 `alloc_bounce_page()` 为缓冲区本身分配页面, 它是一个 `alloc_page()` 的包装器, 但是有一个重要的例外情形。如果该分配操作没有成功, 就会有一个页面的紧急池以及缓冲区首部返回给弹性缓冲区。这将在 9.6 讨论。

可以肯定, 该缓冲区的首部将由 `alloc_bounce_bh()` 分配, 该函数原理上与 `alloc_bounce_page()` 类似, 它调用 slab 分配器得到一个 `buffer_head`, 如果无法分配时将使用缓冲池。另外, 唤醒 `bdflush` 以开始将脏缓冲区刷出到磁盘, 这样就可以立即释放缓冲区。

一旦分配了页面和 `buffer_head`, 信息就会从模板 `buffer_head` 被复制到新的 `buffer_head` 中。由于这部分操作可能用到 `kmap_atomic()`, 弹性缓冲区的创建仅在设置了 IRQ 安全锁 `io_request_lock` 后才开始。I/O 完成的回调过程或者改变为 `bounce_end_io_write()` 或者改变为 `bounce_end_io_read()`, 这取决于这是一个读或写缓冲区, 因此数据将会在高端内存来回复制。

在分配方面要注意的最重要的方面是 GFP 标志位并没有涉及高端内存的 I/O 操作。这一点很重要, 因为弹性缓冲区是用于高端内存的 I/O 操作的。如果分配器试图进行高端内存 I/O, 它将被递归调用并最终失败。

图 9.4 调用图: `bounce_end_io_read/write()`

9.5.3 通过弹性缓冲区复制数据

通过弹性缓冲区复制的数据随着它是读还是写缓冲区而不同。如果该缓冲区用于写数据到设备,那么该缓冲区存储的是在 `copy_from_high_bh()` 创建弹性缓冲区时从高端内存读出的数据,回调函数 `bounce_end_io_write()` 将在设备准备好接收数据时完成 I/O 操作。

如果该缓冲区用于从设备读数据,则在设备准备好前是没有数据转移的。当设备准备好后,设备的中断处理程序将调用回调函数 `bounce_end_io_read()`,由它调用 `copy_to_high_bh_irq()` 把数据复制到高端内存。

在这两种情形下一旦 I/O 完成,模板的回调函数 `buffer_head()` 调用后,缓冲区首部和页面都有可能被 `bounce_end_io()` 回收。如果紧急池没有满,则资源会被加入到池中,否则它们就会被释并放回各自的分配器。

9.6 紧急池

为了快速使用弹性缓冲区,系统提供了 `buffer_head` 的两个紧急池以及相应的页面。由于高端内存的缓冲区不能等到低端内存可用时才释放,如果对分配器而言内存过于紧张,这时分配器无法完成请求的 I/O 操作,就会出现这种情况。这会导致操作失败,也可能会阻止这些操作释放它们的内存。

紧急池由 `init_emergency_pool()` 初始化,每个池包含 `POOL_SIZE` 项。页面通过一个 `pagelist` 字段相互链接在一个以 `emergency_pages` 为头部的链表上。图 9.5 说明了在紧急池中如何存储页面和如何在需要时获得页面。

buffer_head 与之非常相似,因为它们也通过 `buffer_head->inode_buffers` 链接到一个以 `emergency_bhs` 为头部的链表上。页面和缓冲区链表中剩余的项数分别由两个计数器 `nr_emergency_pages` 和 `nr_emergency_bhs` 记录,而且这两个列表由一个叫 `emergency_lock` 的自旋锁保护。

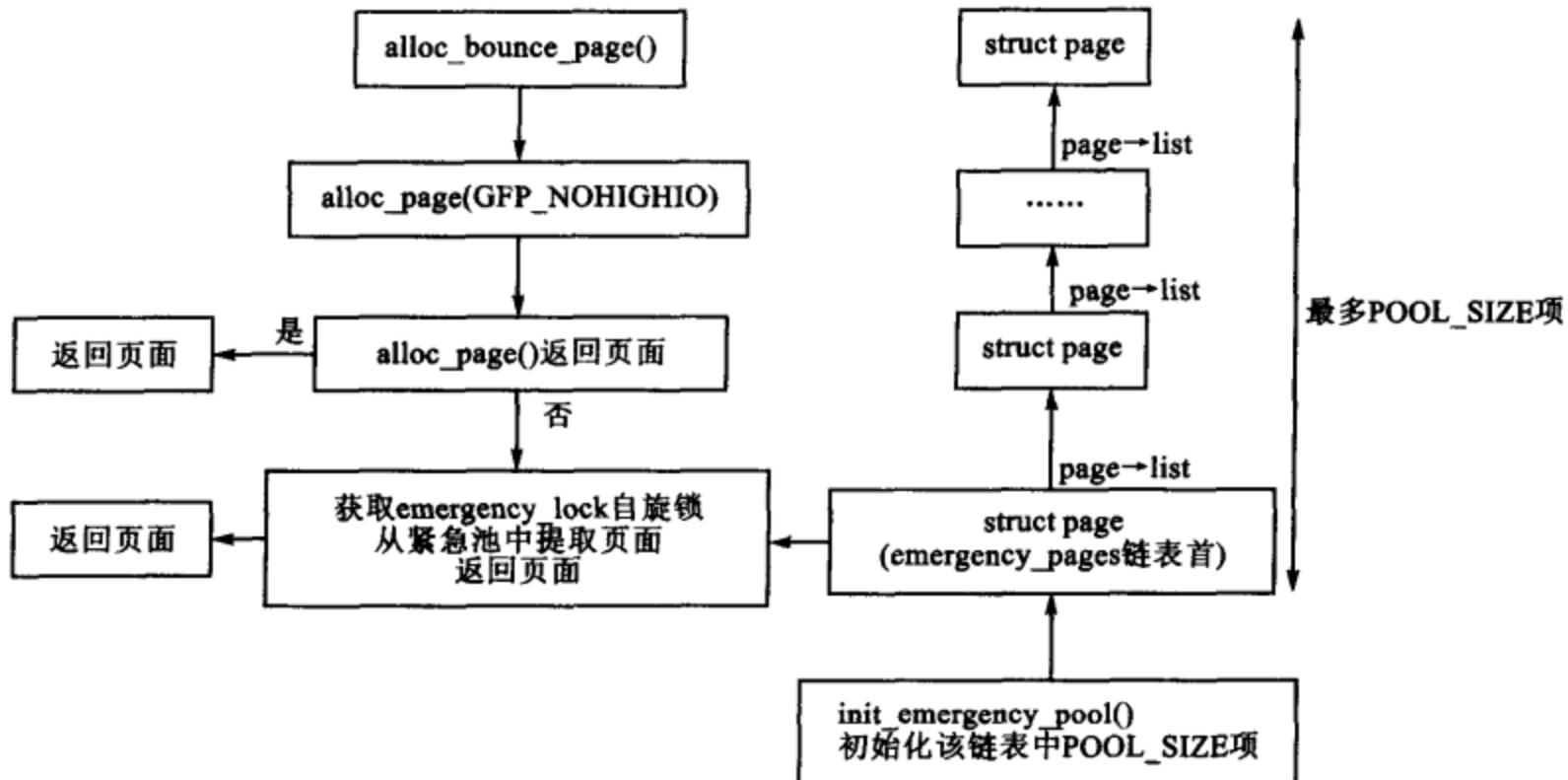


图 9.5 从紧急池中获得页面

9.7 2.6 中有哪些新特性

内存池

在 2.4 中,高端内存管理器是惟一维护紧急池中页面的子系统。2.6 中实现了内存池,它是当内存紧张时需要保留的对象的一般概念。这里的对象可以是任何对象,如除高端内存管理器中的页面外,更多的是由 slab 分配器管理的一些对象。系统使用 `mempool_create()` 来初始化内存池,它需要有如下参数:需要保留对象的最小值(`min_nr`),一个对象类型的分配函数(`alloc_fn()`),一个对象类型的释放函数(`free_fn()`),以及一些用于分配和释放函数的可选的私有数据。

内存池 API 提供两类分配和释放函数,分别叫做 `mempool_alloc_slab()` 和 `mempool_free_slab()`。在使用这两类函数时,私有数据就是待分配和释放对象的 slab 高速缓存。

在高端内存管理器中,创建有两个页面池,其中一个页面池用于普通的页面池,另外一个是

用于 ISA 设备。ISA 设备必须从 ZONE_DMA 中分配,其分配函数是 `page_pool_alloc()`,所传递的私有数据参数表明 GFP 标志位是否使用,而释放函数是 `page_pool_free()`。这样内存池就代替了 2.4 中紧急池的那部分代码。

在从紧急池中分配或释放对象时,系统提供了内存池 API 函数 `mempool_alloc()` 和 `mempool_free()` 函数。另外系统调用 `mempool_destroy()` 释放内存池。

映射高端内存页面

在 2.4 中,字段 `page->virtual` 用于存储 `pkmap_count` 数组中的页面地址。由于在高端内存系统中结构体 `pages` 的数量很多,所以相对而言就有很多较小的页面需要映射到 ZONE_NORMAL 中,2.6 中也有这样一个 `pkmap_count` 数组,但是如何管理它就与 2.4 有很大不同。

在 2.6 中创建了一个 `page_address_htable` 的哈希表,这个表基于结构体 `page` 的地址做哈希操作,另外使用了一个列表来定址结构体 `page_address_slot`。我们感兴趣的是 `struct page` 中的两个字段,一个是 `struct page`,另一个是虚拟地址。当内核需要一个映射页面的虚拟地址时,系统就遍历这个哈希桶。至于页面如何映射到低端内存,这实际上与 2.4 中相似,但 2.6 中不再需要 `page->virtual`。

进行 I/O

最后一个改变是在进行 I/O 时,最主要的是使用了 `struct bio` 来代替结构体 `buffer_head`。`bio` 结构的工作原理已经超出了本书的范围。需要说明的是,这里引入 `bio` 结构的主要原因是无论底层的设备是否支持,都可以进行以块为单位的 I/O 操作。在 2.4 中,无论底层设备的传输速率如何,所有的 I/O 操作都被分解为页面大小。

第 10 章

页面帧回收

由于磁盘缓冲区、目录表项、索引节点项、进程页面以及其他的原因,运行中的系统最终会使用完所有可用的页面帧。因此,Linux 需要在物理内存耗尽前选择旧的页面进行释放,以使这些页面失效而待重新使用。这一章主要集中讨论 Linux 如何实现页面替换策略以及如何让不同类型的页面失效。

本质上,Linux 选择页面的方法在一定程度上是凭经验的,而其背后的理论基础基于多种不同的策略。在实际过程中这些方法运行良好,并且它们根据用户的反馈和基准测试在不断地进行调整。在这一章中,首先要讨论的话题是页面替换策略的基础。

第 2 个要讨论的话题是页面高速缓存。所有从磁盘读出来的数据都存储在页面高速缓存中,以此来减少磁盘 I/O 的次数。严格意义上,这和页面帧回收并没有什么直接的联系,但是 LRU 链表和页面高速缓存却是密不可分的。相关的章节会集中讨论页面如何添加到页面高速缓存中以及如何被快速定位。

接下来是第 3 个话题,LRU 链表。除了 slab 分配器,系统中所有正在使用的页面都存放在页面高速缓存中,并由 page→lru 链接在一起,所以很容易就可以扫描并替换它们。slab 页面并没有存放在页面高速缓存当中,因为人们认为基于被 slab 所使用的对象来对页面计数是很困难的。

在这个部分,会谈及页面如何依附于其他高速缓存。在谈到进程映射如何被移除之前,诸如 dcache 和 slab 分配器这样的高速缓存应当被回收。进程映射页面并不容易进行交换,这是因为除了采用查找每个页表这种手段以外没有其他的方法能把结构页面映射为 PTE,而且查找每个页表的代价也是非常大的。如果页面高速缓存中存在大量的进程映射页面,系统将会遍历进程页表,然后通过 swap_out() 函数交换出页面直至有足够的页面空闲,然而 swap_out() 函数依旧会因为共享页的缘故而带来问题。如果一个页面是共享的,同时一个交换项已经被分配,PTE 就会填写所需的信息以便在交换分区里重新找到该页并将其引用计数减 1。只有

当引用计数减到零时,这个页面才被替换出去。诸如此类的共享页面都会在交换高速缓存部分涉及到。

最后,这个章节同样会涉及页面替换守护程序 `kswapd`,并讨论它的实现和职责所在。

10.1 页面替换策略

在讨论页面替换策略时讲的最多的便是基于最近很少使用的算法(LRU),但严格意义上,这并不是正确的,因为这里的链表并不是严格地按照 LRU 顺序来保持的。Linux 中的 LRU 链包含两个链,分别是 `active_list` 和 `inactive_list`。`active_list` 上的对象包含所有进程的工作集[Den70],而 `inactive_list` 则包含需要回收的候选对象。由于所有可回收的页面都仅存于这两个链表中,并且任何进程的页面都可被回收,而不仅仅局限于那些出错的进程,因此替换策略的概念是全局的。

这两个列表很像一个简化了的 LRU 2Q[JS94],其中分别维护着两个叫做 `Am` 和 `Ai` 的链表。在 LRU 2Q 中,第一次分配的页面被放置在一个叫做 `Al` 的先进先出(FIFO)队列中。如果被引用的页面同时也在那个队列中,它们就会被放置到一个叫做 `Am` 的普通 LRU 链表中。这与使用 `lru_cache_add()` 函数把页面放到一个叫 `inactive_list`(`Al`)的队列中及使用 `mark_page_accessed()` 函数把页面移到 `active_list` 中(`Am`)的方式有点相似。这个算法描述了两个链表的大小是如何相互变化的,但是 Linux 采用了一种更加简单的方法:使用 `refill_inactive()` 函数把页面从 `active_list` 尾部移到 `inactive_list`,以确保 `active_list` 约占总页面高速缓存大小的 $2/3$ 。图 10.1 图解了如何构造这两个列表,并阐明了如何添加页面以及如何通过 `refill_inactive()` 函数在两个链表之间移动。

2Q 所描述的链表假定 `Am` 是一个 LRU 链表,而 Linux 中的链表更像是一个时钟算法[Car84],其中时针是 `active_list` 的大小。当页面到达链表的底部时,就会检查引用的标志位。如果设置了引用的标志位,则该页面会被移回到链表的顶部,然后检查下一个页面。如果没有设置,则该页面会被移到 `inactive_list`。

这种前向试探的方法意味这些链表的行为像 LRU 一样,但是 Linux 替换策略和 LRU 还是有很多不同之处的,一般认为后者是栈式算法[MM87]。即使我们忽略掉这是在分析多道程序系统的问题,同时也不考虑每个进程所占内存大小不固定的情况[CD80],这种策略也不满足内含特性,因为链表中页面的位置主要取决于链表的大小,这和最后一次被引用时是相反的。因为需要链表按每次的引用来进行更新,所以链表的优先级也没有排序。在整个栈算法框架中,当从进程中换出页面时,链表的关键地位几乎被忽略掉了,它的换出取决于进程在虚拟地址空间的位置,而不是页面在链表中的位置。

总而言之,这种算法的确表现得像 LRU 一样,而且基准测试中已经表明它在实际使用过

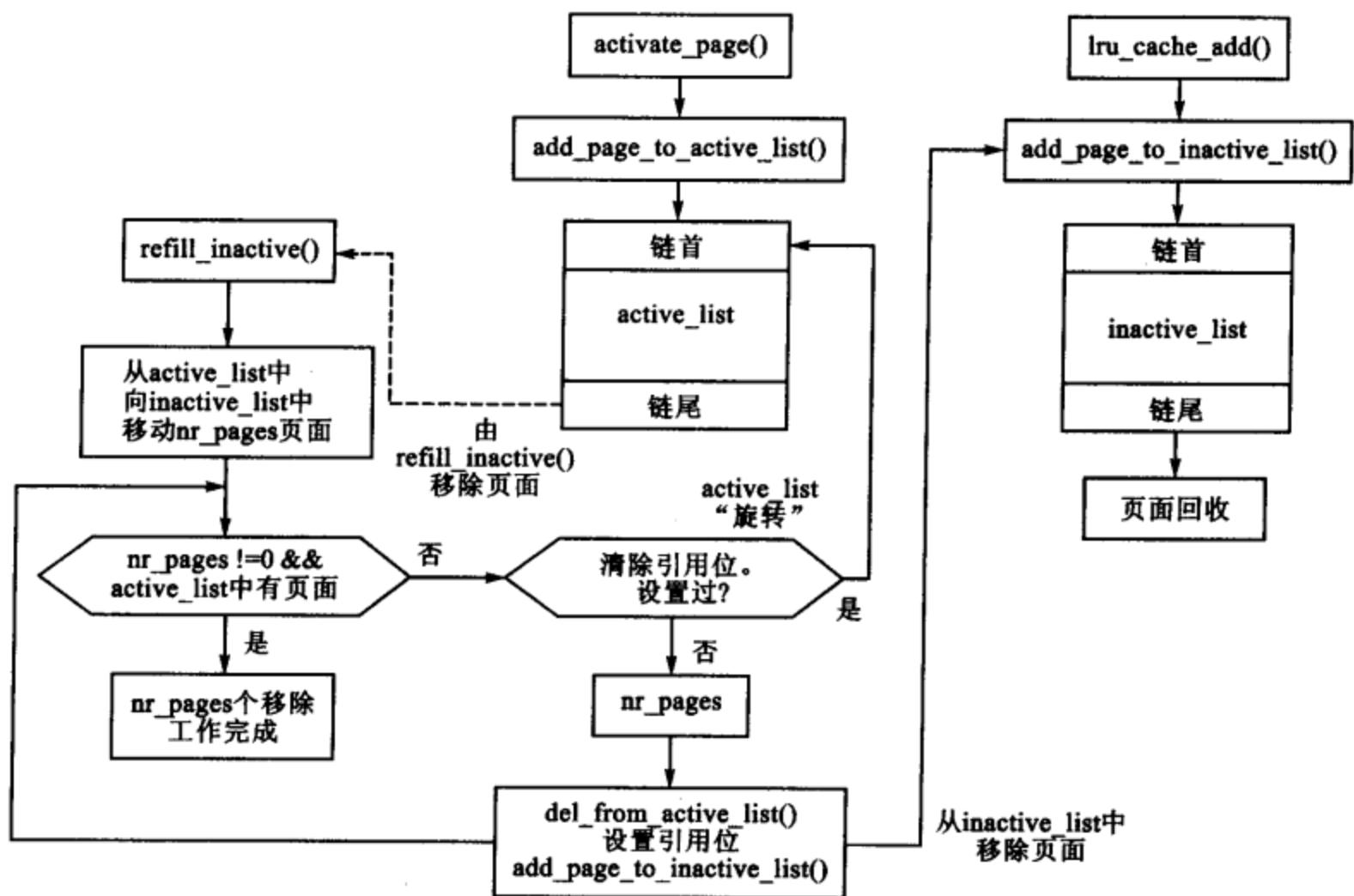


图 10.1 页面高速缓存 LRU 列表

程中运行良好。只有 2 种情况下这种算法才可能表现得很差。第 1 情况是如果待回收的页面主要是匿名页面,在这种情况下,Linux 会连续检查大量的页面,而这将在线性扫描进程页表以搜索待回收的页面之前完成。幸运的是,这种情况极为少见。

第 2 种情况是在某个单进程中,系统频繁地写在 inactive_list 上的许多对应于文件的常驻页面中。进程和 kswapd 可能会进入一个循环过程,持续地调换这些页面并把它们放到 inactive_list 的首部,却没有释放任何东西。在这种情况下,由于这两个链表的大小没有明显的改变,几乎没有页面会从 active_list 移到 inactive_list 中。

10.2 页面高速缓存

页面高速缓存是一个包含页面的数据结构集,这些页面对应于普通文件、块设备或交换分区。在高速缓存中,基本上存在四种类型的页面:

- 读内存映象文件时所产生的页面碎片。

- 从块设备上读出的块,或从文件系统中读出的块,它们都被封装到一个成为缓冲区页面的特殊页面中。块的数量随块大小和系统中页面大小而变化。
- 在第 11 章中将讨论匿名页面,这些页面存放于一种称为交换高速缓存的特殊页面高速缓存中,交换高速缓存在后援存储器中分配槽的时候用于换出页面。
- 属于共享内存区的页面在处理方式上和匿名页面类似。它们之间惟一的区别是在第一次写到页面后,共享页面才被添加到交换高速缓存中,并马上在后援存储设备中保留一份空间。

高速缓存存在的主要原因是减少了不必要的读磁盘操作。从磁盘读出的页面被存储在一个基于 address_space 结构和偏移地址的页面哈希表里,在磁盘访问前都要先查找这个表。把页面放入这个高速缓存有两个原因:第一个原因是减少了不必要的读磁盘操作。从磁盘读出的页面是存放在一个基于 address_space 结构和偏移地址的页面哈希表中的,第二个原因是由于页面高速缓存中具有队列,这就为磁盘替换算法选择丢弃或换出页面打下了基础。这里提供了一个负责操作页面高速缓存的 API 函数,如表 10.1 所列。

表 10.1 页面高速缓存 API

| |
|---|
| <pre>void add_to_page_cache(struct page * page, struct address_space * mapping, unsigned long offset)</pre> <p>添加页面到 LRU 中,同时通过 lru_cache_add() 将其添加到索引节点队列和页面哈希表</p> |
| <pre>void add_to_page_cache_unique(struct page * page, struct address_space * mapping, unsigned long offset, struct page ** hash)</pre> <p>同 add_to_page_cache(),除了要检查页面是否已经在页面高速缓存中。这在调用者不持有 pagecache_lock 自旋锁时使用</p> |
| <pre>void remove_inode_page(struct page * page)</pre> <p>这个函数从索引节点和哈希队列移除一个页面,同时调用了 remove_page_from_inode_queue() 和 remove_page_ from_hash_queue() 函数,相应地从页面高速缓存中移除了该页面</p> |
| <pre>struct page * page_cache_alloc(struct address_space * x)</pre> <p>这个函数是对 alloc_pages() 函数的封装,用 x->gfp_mask 作为 GFP 的掩码</p> |
| <pre>void page_cache_get(struct page * page)</pre> <p>这个函数用于增加已经在页面高速缓存中的页面的引用计数</p> |
| <pre>int page_cache_read(struct file * file, unsigned long offset)</pre> <p>如果页面不在高速缓存中,这个函数将添加对应于某文件相应偏移量的页面。如果有必要的话,页面会通过 address_space_operations->readpage 函数从磁盘读到高速缓存中</p> |
| <pre>void page_cache_release(struct page * page)</pre> <p>这个函数是 __free_page() 函数的别名。引用计数减小,而且如果减少到 0,释放掉页面</p> |

10.2.1 页面高速缓存哈希表

系统要求被快速地定位在高速缓存中的页面。为了简化这一点,页面都被插入了一个叫 page_hash_table 的表中,而 page→next_hash 和 page→pprev_hash 则用于处理冲突。

该表在 mm/filemap.c 中声明如下:

```
45 atomic_t page_cache_size = ATOMIC_INIT(0);
46 unsigned int page_hash_bits;
47 struct page ** page_hash_table;
```

在系统初始化时,函数 page_cache_init() 将系统中物理页面的数量作为一个参数,并分配该表。该表所申请的大小(htable_size)足够保存系统中每个结构页面的指针,它的计算公式如下:

$$\text{htable_size} = \text{num_physpages} * \text{sizeof}(\text{struct page} *)$$

为了分配这样一个表,系统首先分配一块足够大的顺序空间来存放整个表。它从 0 开始计算并逐步增大该值,直到 $2^{\text{order}} > \text{htable_size}$ 。这大致可以用下列简单方程的整数部分来粗略地表示:

$$\text{order} = \log_2((\text{htable_size} * 2) - 1)$$

系统尝试使用 _get_free_pages() 来分配这些顺序页面。如果分配失败,则尝试分配序号较小的页面,如果同样没有可以分配的页面,则系统瘫痪。

page_hash_bits 的值是基于表的大小的,并被哈希函数 _page_hashfn() 所使用。这个值通过连续除以 2 得到,如果是在实数域,它等于

$$\text{page_hash_bits} = \log_2 \left\lceil \frac{\text{PAGE_SIZE} * 2^{\text{order}}}{\text{sizeof}(\text{struct page} *)} \right\rceil$$

这样就使得该表是一个 2 的幂次哈希表,从而避免了在哈希函数中经常使用的取模运算。

10.2.2 索引节点队列

索引节点队列是 address_space 结构中的一部分,这在 4.4.2 小节中已有介绍。该结构包含三个链表:clean_pages 是一个由干净页面组成的链表,这些页面均与索引节点相关;dirty_pages 是一个由自从链表与磁盘同步以来已经被写过的页面所组成的链表;locked_pages 则是由那些正处于锁定状态的页面所组成的。这三种链表合起来被认为就是给定的某种映射的索引节点队列,并且 page→list 字段用来链接队列上的页面。页面通过 add_page_to_inode_queue() 函数添加到索引节点队列中的 clean_pages 链表上,并通过 remove_page_from_inode_queue() 函数完成移出操作。

10.2.3 向页面高速缓存添加页面

从文件或块设备读取页面时,通常将页面添加到页面高速缓存中以避免更多的磁盘 I/O。大多数的文件系统都使用较高层次的函数 `generic_file_read()` 来完成它们的 `file_operations->read()` 操作,如图 10.2 所示。在第 12 章中涉及的共享内存文件系统,是一个值得关注的例外。一般情况下,文件系统都是通过页面高速缓存来进行它们的操作的。在这一节中,我们介绍 `generic_file_read()` 如何完成操作,以及如何把页面添加到页面高速缓存中的。

对于普通 I/O, `generic_file_read()` 函数在调用 `do_generic_file_read()` 函数之前会进行少量的基本检查工作。通过调用 `_find_page_nolock()` 函数并上锁 `pagecache_lock` 来搜索页面高速缓存,以查看该页面是否已经存在于高速缓存中。如果不在,一个新的页面将通过 `page_cache_alloc()` 函数分配,其过程一般通过 `alloc_pages()` 函数封装后再通过 `_add_to_page_cache()` 函数添加到页面高速缓存中。在页面高速缓存中形成页面帧后,再通过调用 `generic_file_readahead()` 函数(实质上是调用 `page_cache_read()` 函数)从磁盘读取页面。它使用 `mapping->a_ops->readpage()` 这样的方法来读取页面,其中 `mapping` 指的是管理文件的 `address_space`。`readpage()` 函数是用来从磁盘读取页面的文件系统专有函数。

若没有进程与匿名页映射,系统会将匿名页添加到交换高速缓存中,这将在 11.4 节有更进一步的讨论。持续地尝试将它们换出时,由于它们没有可用来映射的 `address_space` 或者文件中的任何偏移量,这样将无法把它们添加到页面高速缓存的哈希表中。即便如此,也要注意到这些页面依旧存在于 LRU 链表中。一旦进入交换高速缓存中,匿名页面和有文件后援的页面之间的惟一实际差异便是匿名页面采用 `swapper_space` 作为其 `struct address_space`。

共享内存页面在下面两种情况下会被添加到页面高速缓存中。第 1 种情况是在调用 `shmem_getpage_locked()` 函数时,不论它是从交换分区中取出或者是在其中分配页面,其原因就是该页面是第 1 次被引用。第 2 种情况是当换出代码调用 `shmem_unuse()` 函数时。当一个交换分区被禁止时以及一个有后援交换区的页面被发现不属于任何进程时就会出现第 2 种情况。与共享内存相关的索引节点都会被穷尽搜索,直至找到正确的页面。在这两种情况下, `add_to_page_cache()` 函数将会把该页面添加到页面高速缓存中,如图 10.3 所示。

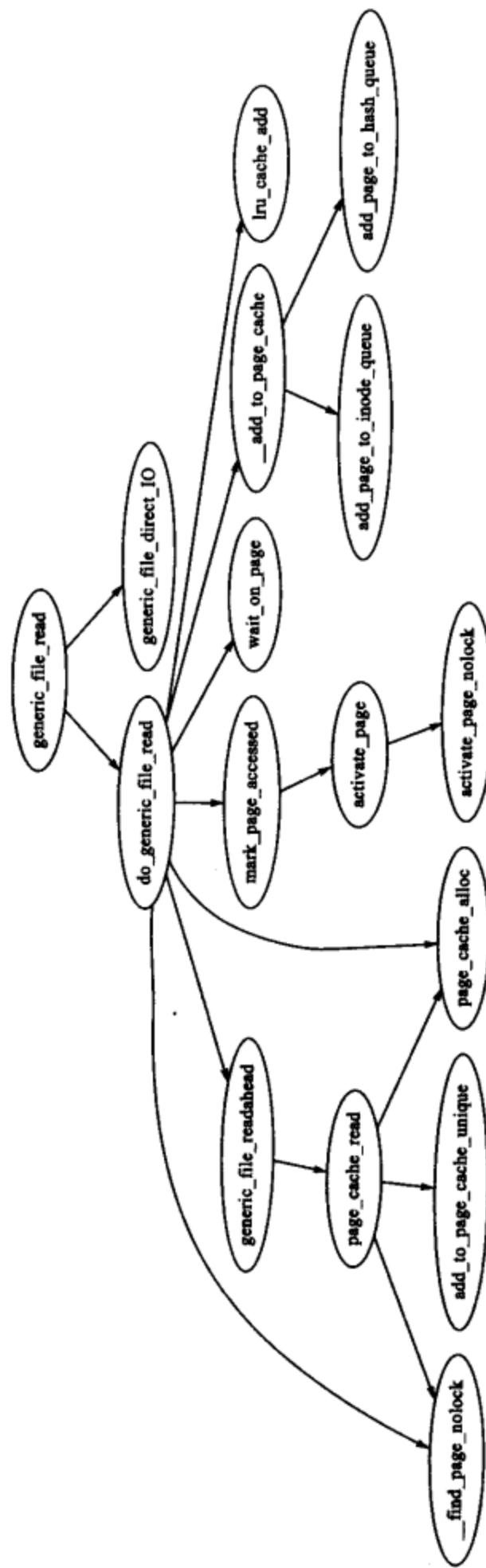


图 10.2 调用图：generic_file_read()

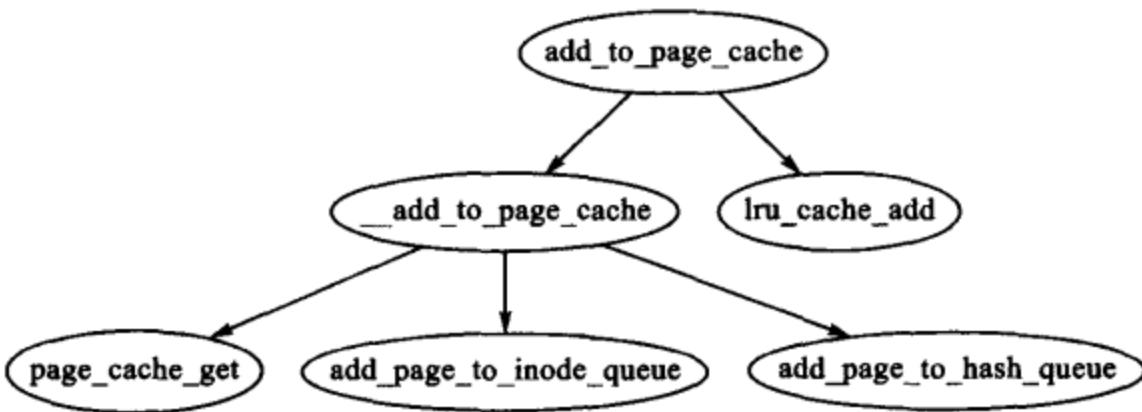


图 10.3 调用图: add_to_page_cache()

10.3 LRU 链表

正如 10.1 节中所谈到的, LRU 链表包含两个链表, 一个称为 `active_list`, 而另一个称为 `inactive_list`。它们在 `mm/page_alloc.c` 中声明, 并由 `pagemap_lru_lock` 自旋锁保护。一般地, 它们分别存储着经常使用和不经常使用的页面, 或者说, `active_list` 包含系统中所有的工作集, 而 `inactive_list` 则包含回收候选集。在表 10.2 中列出了操作 LRU 链表的 API 函数。

10.3.1 重新填充 `inactive_list`

在高速缓存收缩时, `refill_inactive()` 函数会把页面从 `active_list` 移到 `inactive_list` 中。它有一个参数表示移动页面的数量, 该参数通过在 `shrink_caches()` 函数中求比率得到, 而这个比率依赖于 `nr_pages`, 即 `active_list` 中的页面数量, 还有 `inactive_list` 中的页面数量。需要移动的页面数量计算公式为:

$$\text{pages} = \text{nr_pages} * \frac{\text{nr_active_pages}}{2 * (\text{nr_inactive_pages} + 1)}$$

这样可以保证 `active_list` 大约是 `inactive_list` 大小的 $2/3$, 要移动的页面数量是由一个基于我们需要换出的页面数量(`nr_pages`)的比率决定的。

页面从 `active_list` 的尾部取出。如果设置了 `PG_referenced` 标志位, 则该标志位要被清除, 然后将页面放置到 `active_list` 的首部, 因为它最近被使用过且依旧活跃。这种情况有时被称为链表旋转。如果没有设置该标志位, 则该页面被移到 `inactive_list`, `PG_referenced` 标志位会被设置, 这样使得该页面在需要时能很快地添加到 `active_list` 中。

10.3.2 从 LRU 链表回收页面

shrink_cache() 函数是替换算法的一部分, 它把页面从 inactive_list 中取出, 并决定它们应该如何被换出。nr_pages 和 priority 用于决定多少工作需要做的初始参数。nr_pages 初始值为 SWAP_CLUSTER_MAX, 目前在 mm/vmscan.c 中定义为 32。而 priority 的初始值为 DEF_PRIORITY, 目前在 mm/vmscan.c 中定义为 6。

max_scan 和 max_mapped 这两个参数决定了该函数要做多少工作, 并根据 priority 调整。每当没有足够的空闲页面却又调用了 shrink_caches() 函数时, priority 会逐渐减小直至达到最高优先级 1。

表 10.2 LRU 链表 API

| | |
|---|--|
| void lru_cache_add(struct page * page) | 添加一个未激活页面到 inactive_list 中。当页面处于激活状态时, 如当一个页面已经是碎片时, 调用 mark_page_accessed() 函数将该页面移到 active_list |
| void lru_cache_del(struct page * page) | 既可以调用 del_page_from_active_list() 函数, 也可以调用 del_page_from_inactive_list() 函数, 将页面从 LRU 链表移除 |
| void mark_page_accessed(struct page * page) | 标记该页面已经被访问过。如果它不是最近被引用过(在 inactive_list 链表里, 而且 PG_referenced 标准位未被设置), 引用标准位将被设置。如果被第二次引用, 将调用 active_page() 函数, 用于标记该页面是激活状态, 且其引用标准位被清除 |
| void activate_page(struct page * page) | 从 inactive_list 移动一个页面到 active_list 中。该函数很少被直接调用, 因为调用者必须知道该页面是在 inactive_list 上。取而代之的是使用 mark_page_accessed() 函数 |

变量 max_scan 表示该函数扫描的最大页面数, 它可简单计算如下

$$\text{max_scan} = \frac{\text{nr_inactive_pages}}{\text{priority}}$$

其中 nr_inactive_pages 表示 inactive_list 中的页面数量。这就意味着在最小优先级 6 时, inactive_list 的页面中最多只有 1/6 会被扫描到, 而在最大优先级时, inactive_list 中所有页面都会被扫描到。

第二个参数是 max_mapped, 它决定了在所有进程换出前, 允许多少进程页面存在于页面高速缓存中。它可以在 max_scan 的 1/10 或者以下计算公式

$$\text{max_mapped} = \text{nr_pages} * 2^{(10 - \text{priority})}$$

中取较小的那个值来计算。

或者说,在最低优先级时,映射页面被允许使用的最大数量是 max_scan 的 1/10 或者是待替换页面数(nr_pages)的 16 倍中那个较小的数值。而在高优先级时,它是 max_scan 的 1/10 或是待换出页面的 512 倍。

从上面可以看出,该函数基本上是一个很庞大的 for 循环,它会从 inactive_list 的尾部开始扫描至多 max_scan 个页面,来释放掉 nr_pages 个页面,或直至 inactive_list 为空。在扫描每个页面时,它会检查是否需要重新调度它自己,以保证换出页面不会一直独占 CPU。

对于链表中每一种类型的页面,函数会作出不同的处理决定。这些不同的页面类型以及对应的处理方式如下:

1. 由进程映射的页面

这会跳转到 page_mapped 标志位,稍后还会再谈到。max_mapped 计数减 1。如果 max_mapped 减到 0,进程的页表会被线性搜索并且使用 swap_out() 函数完成换出。

2. 被锁定,并且 PG_launder 位也被设置了的页面

这样的页面由于在进行 I/O 而被锁定,因此应当可以直接跳过。然而,如果 PG_launder 位同时被设置了,则意味着这是第二次发现该页面被锁定,因此等 I/O 完成后再去除该页面会更好一些。该页面的引用可以通过 page_cache_get() 函数得到,以保证该页面不会被过早地释放,然后调用 wait_on_page() 函数进行睡眠,直至 I/O 完成。在 I/O 完成后,通过 page_cache_release() 函数将引用计数减 1。当引用计数减到 0 时,页面将会被回收。

3. 脏页面

脏页面没有被任何进程映射,没有缓冲区,属于某个设备或文件映射。由于该页面属于某个文件或设备映射,它就有一个可用的有效函数 writepage(),通过 page→mapping→a_ops→writepage 得到。在开始 I/O 的时候,清除 PG_dirty 位,并设置 PG_launder 位。调用 writepage() 函数同步页面及其后援文件前,系统要调用 page_cache_get() 函数得到该页面的引用计数,而后调用 page_cache_release() 函数清除该引用。请注意,这种情况也会引起交换高速缓存中有后援存储的异步页面的同步操作,因为交换高速缓存页面使用 swapper_space 作为 page→mapping。该页面依旧在 LRU 链上。当其再度被发现时,如果 I/O 已经完成了,很容易就可以释放掉它,然后页面将被回收。如果 I/O 还没有完成,正如前面所述,内核会等待 I/O 完成。

4. 具有缓冲区与磁盘数据相关联的页面

用一个引用指针指向该页面,并使用 try_to_release_page() 函数来尝试释放该页面。如果成功并且它是一个匿名页(无 page→maping),则该页面从 LRU 链上移除,并调用 page_cache_released() 函数来减少引用计数。只有一种情况匿名页面会和高速缓存区相关联,那就

是当其有后援的交换文件时,因为页面必须以块对齐的方式写出。在另一方面,如果它有后援的文件或设备,系统就可以在其计数达到 0 时,只需要简单地释放掉其引用指针,便释放掉了该页面。

5. 被多个进程映射的匿名页

第一种情况里被计数的同一个 `page_mapped` 标志位,在被释放掉以前,解锁 LRU 链,同时解锁该页面。或者说, `max_mapped` 计数将减少,在其达到 0 时,调用 `swap_out` 函数。

6. 没有被任何进程引用的页面

这是最后一种情况,但不是非得明确提出的。如果该页在交换高速缓存中,它将从交换高速缓存中移除,因为它现在正在和其后援存储器进行同步,而且没有任何的进程引用它。如果它是一个文件中的某部分,则它会从索引节点队列中移除,然后会从页面高速缓存中删除,并被释放掉。

10.4 收缩所有的高速缓存

负责收缩各种高速缓存的函数是 `shrink_caches()`,它使用很简单的步骤释放一些内存(参考图 10.4)。任意一次允许写回磁盘的页面数量最大值是 `nr_pages`,它由 `try_to_free_pages_zone()` 函数初始化为 `SWAP_CLUSTER_MAX`。有了这样的限制后,如果 `kswapd` 调度大量的页面写回到磁盘,它会偶尔睡眠一下以允许 I/O 开始。随着页面的释放, `nr_pages` 会相应地减小。

待做的工作量也取决于 `priority`,它由 `try_to_free_pages_zone()` 函数初始化为 `DEF_PRIORITY`。如果每次释放的页面不够数量, `priority` 会减小到最高优先级 1。

该函数会首先调用 `kmem_cache_reap()` 函数(见 8.1.7 小节)来选择一个待减小的 slab 高速缓存。如果释放的页面的数量达到 `nr_pages`,就表示工作完成了,然后函数返回。否则它会试着从其他高速缓存中释放页面以达到 `nr_pages` 个。

如果没有涉及到其他的高速缓存,在调用 `shrink_cache()` 函数来从 `inactive_list` 尾部回收页面以减小页面高速缓存以前, `refill_inactive()` 函数会将页面从 `active_list` 移到 `inactive_list`。

最后,它会减小三种特殊的高速缓存: `dcache(shrink_dcache_memory())` 高速缓存, `icache(shrink_icache_memory())` 高速缓存以及 `dqcache(shrink_dqcache_memory())` 高速缓存。这些对象本身都很小,但是级联效应会使大量的页面以缓冲区和磁盘高速缓存的形式释放。

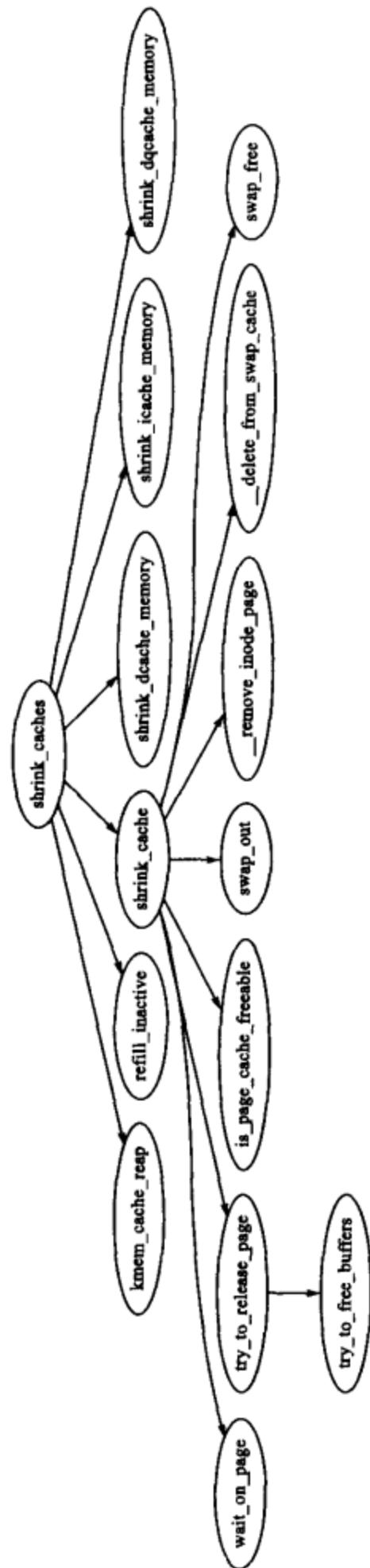


图 10.4 调用图: shrink caches()

10.5 换出进程页面

如图 10.5 所示,当在页面高速缓存中找到了 max_mapped 个页面时,swap_out()就会被调用以启动换出进程页面的操作。系统从 swap_mm 指向的 mm_struct 和 mm→swap_address 地址开始向前搜索页表,直到释放 nr_pages 个页面。

除了 active_list 部分的页面和最近被引用的页面会被跳过外,进程映射的页面不管处于链表中的哪个位置,也不管什么时候被最后引用过,都会被检查。检查激活状态页面的开销是不小的,但是与为了获得引用某一结构页面的 PTE 而线性查找所有进程的开销比起来,这种开销又是微不足道的。

一旦决定了要换出某个进程的页面,就会试图换出至少 SWAP_CLUSTER 个页面,并且 mm_struct 的整个链表仅会被检查一遍,这样避免了在没有页面时无休止地循环。大块写出页面增加了在进程地址空间近邻的页面写到相邻磁盘槽的几率。

标志位 swap_mm 被初始化以指向 init_mm,而 swap_address 在首次使用时被初始化为 0。当 swap_address 等于 TASK_SIZE 时,说明某个进程已经被完全地搜索了一遍。一旦选择从某个进程换出页面,它的 mm_struct 的引用指针增加 1,使得它不会过早地被释放。然后系统会以选定的 mm_struct 作为参数来调用 swap_out_mm()。这个函数遍历该进程持有的每个 VMA,并且在其上调用 swap_out_vma()。这样就避免了必须遍历整个很稀疏的页表的情形。swap_out_pgd()和 swap_out_pmd()遍历给定 VMA 的页表直到最后在实际页面以及 PTE 上调用 try_to_swap_out()。

函数 try_to_swap_out()首先检查以确保该页面或者不是 active_list 的一部分,或者该页面最近被引用过了,或者是我们并不关心的某个管理区的一部分。一旦确定了这是一个待换出的页面,该页面便会从该进程的页表中移出。接着检查刚移除的 PTE 以确定它是否是脏页面。如果它是脏页面,则会更新 struct page 标志位来反映这一点,以使它与后援存储设备同步。如果该页面已经在交换高速缓存中,就更新它的 RSS 位,并去掉对它的引用。否则,就把该页面加入到换出高速缓存中。在后援存储设备上分配空间和换出页面的过程将在第 11 章进一步讨论。

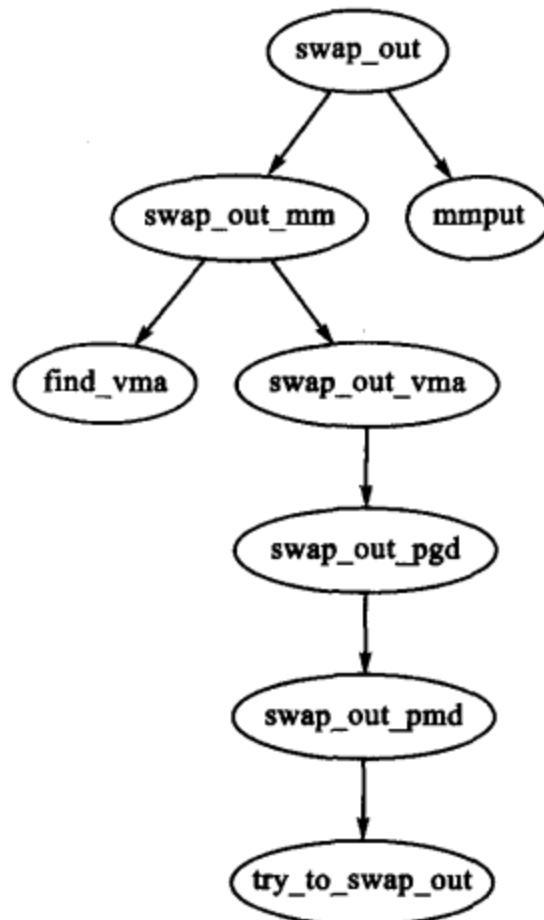


图 10.5 调用图: swap_out()

10.6 页面换出守护程序(kswapd)

在系统启动时,一个 kswapd 的内核线程从 kswapd_init()中启动。大多数情形下它都处于睡眠状态,一旦运行,它就不断地执行在 mm/vmscan.c 中的 kswapd()函数。这个守护程序负责在内存大小剩下很少时回收页面。从历史经验上来看,kswapd 经常是每 10 s 被唤醒一次。现在它由物理页面分配器在管理区中空闲页的 pages_low 大小达到了某个值时唤醒(见 2.2.1 小节)。

正是这个守护程序完成了大多数诸如保持页面高速缓存正确,收缩 slab 高速缓存并在需要的时候替换出进程的任务。与 Solaris[MM01]中的替换守护程序不同,kswpad 不是随着内存压力的增大而增加被唤醒的频率,而是不断地在释放页面直到空闲页的 pages_high 大小达到阙值。在极端的内存压力下,所有的进程会通过调用 balance_classzone(),然后 balance_classzone()调用 try_to_free_pages_zone(),来同步地完成 kswap 的工作。在图 10.6 中,当物理页面分配器在和 kswapd 同步相同的任务量,分配处于重负的管理区时也会调用 try_to_free_pages_zone()。

当 kswapd 被唤醒时,它执行下列步骤:

调用 kswapd_can_sleep(),它遍历所有的管理区,检查在结构 zone_t 中的 need_balance 字段。如果有 need_balance 被设置,则它不能睡眠。

如果不能睡眠,则会被移出 kswapd_wait 等待队列。

调用 kswapd_balance(),它遍历所有的管理区。如果 need_balance 字段被设置,它将使用 try_to_free_pages_zone()来释放管理区内的页面,直到达到 pages_high 阈值。

由于 tq_disk 的任务队列的运行,所以页面队列都将清除。

将 kswapd 重新加入 kswapd_wait 队列并且返回第一步。

10.7 2.6 中有哪些新特性

守护进程 kswapd

如同已经在 2.8 节中说过的一样,现在系统中每一个内存节点都拥有一个 kswapd。这些守护进程现在仍然由 kswapd()启动,它们执行一样的代码,但是现在它们都只限于在局部的节点上执行。2.6 中 kswapd 的改变主要与 kswapd-per-node 的改变有关。

kswapd 的基本操作没有变。一旦该守护进程被唤醒,它就会在其监管的 pgdat 上调用

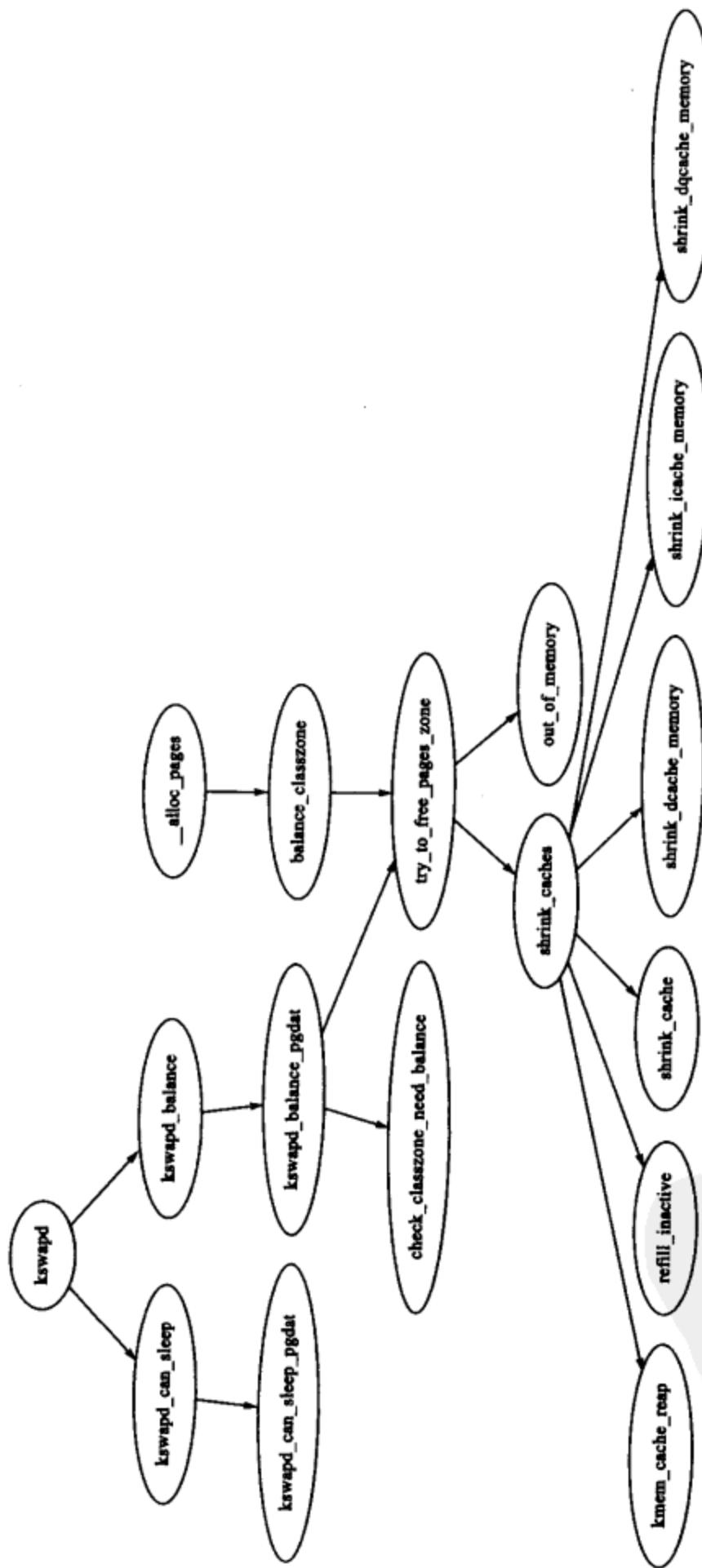


图 10.6 调用图：kswapd()

balance_pgdat()。balance_pgdat()有两种操作模式。如果以 `nr_pages == 0` 的条件调用,则它就会不断地试着释放局部 pgdat 中各个管理区的页面,直到达到了 `pages_high` 的值。如果以某个 `nr_pages` 的条件调用,则它就会不断地试着释放 `nr_pages` 和 `MAX_CLUSTER_MAX * 8` 中那个较小值的页面。

平衡管理区

balance_pgdat()释放页面时调用的两个主要函数是 `shrink_slab()` 和 `shrink_zone()`。`shrink_slab()` 在 8.8 节中已经讨论过了,不再重复。balance_pgdat()调用函数 `shrink_zone()` 来释放一定数量的页面,所要释放页面的确切数量取决于释放页面的紧急程度。该函数与 2.4 中的几乎一样。`refill_inactive_zone()` 用于把一定数量的页面从 `zone->active_list` 移到 `zone->inactive_list` 中。注意在 2.8 节中说过,2.6 中的 LRU 链表是每个区域都有的,与 2.4 中 LRU 链表是全局的情况不同。系统调用 `shrink_cache()` 移出 LRU 中的页面并且回收。

页面换出压力

2.4 中页面换出操作的优先级由扫描的页面数量决定。2.6 中由 `zone_adj_pressure()` 调整 `zone->pressure` 字段指示为替换而需要扫描的页面数量,而这个调整通常是逐渐减弱的。当需要更多的页面时,`zone->pressure` 的值就会达到最高值 `DEF_PRIORITY << 10`,然后随着时间慢慢降低。通常这个值影响着为替换而需要在管理区中扫描的页面数量。这样做的目的是启动页面替换然后逐渐降低压力,而不是短时间突然进行大量替换。

控制 LRU 链表

2.4 中当需要从 LRU 链表中移出页面的时候就需要获得一个自旋锁,这样就会引起很激烈的加锁竞争。所以,为了缓解这种竞争程度,2.6 中使用了 `struct pagevec` 结构来处理涉及到 LRU 链表的操作,允许在 LRU 链表中批量加入或移出多达 `PAGEVEC_SIZE` 个页面。

具体情形是这样的,当 `refill_inactive_zone()` 或 `shrink_cache()` 要移出页面时,它们先获得 `zone->lru_lock` 锁,然后移出一部分页面,并把它们存储在一个临时链表中。系统只在要移出的页面链表组织好以后,才会调用 `shrink_list()` 进行实际的页面释放工作,这个过程中大部分工作不需要获得 `zone->lru_lock` 锁就可以完成。

当要加入页面时,系统就会调用 `pagevec_init()` 来初始化一个新的页面向量结构体,然后调用 `pagevec_add()` 将所有要加入的页面加入到这个向量中,接着调用 `pagevec_release()` 成批地加入到 LRU 链表中。

相当一部分与 `pagevec` 结构体有关的 API 声明可以在 `<linux/pagevec.h>` 中找到,在 `mm/swap.c` 中有它大部分的实现。

第 11 章

交换管理

由于 Linux 使用空闲内存作硬盘数据的缓冲,所以就有必要释放进程所占用的私有页面或匿名页面。这些页面与普通磁盘文件中的页面不同,不能简单地将它们先丢弃,将来再读取。取而代之的是 Linux 把这些页面复制到后援存储器,有时也称为交换区。本章详细介绍 Linux 如何使用和管理后援存储器。

严格意义上,Linux 并不进行交换操作,字面意义上的“交换”通常是指复制整个进程地址空间到磁盘,而“换页”指的是复制出单独的页面。实际上在当前硬件的支持下,Linux 实现的是换页,也就是传统意义上讨论的文档中所谓的交换。为了与 Linux 中对这个词的使用惯例一致,这里也称之为交换。

存在交换区有两个主要原因。首先,交换区扩充了内存以供进程使用。虚拟内存和交换区的存在使得一个大型的进程即使部分常驻内存也可以运行。由于可能换出旧的页面,以及请求换页要确保页面在必要时能重新载入,所以被访问的内存可能很容易就超过 RAM 的大小。

某些读者可能会认为只要内存容量足够大,就不需要交换区。这就引出了我们需要交换区的第二个原因。一个进程在开始运行时所引用的大量页可能只是为了初始化,在后来的运行期间这些页面将不再使用。把这些页面交换出去可以空闲出更多的磁盘缓冲区,这样做比把它们常驻内存却不使用更能提高系统效率。

这并不是指交换区没有缺点。最重要也是最明显的缺点是访问磁盘的操作非常耗时。若没有交换区和高性能磁盘,那些频繁访问大量内存的进程就只可能在较大内存的支持下才能在合理的时间内完成。所以,正如我们在第 10 章所讨论的一样,如何选择正确的页面交换出去是非常重要的。同样,这也是为什么相关的页面应该存放在交换区毗邻位置的原因,因为这样进程在预读的时候能同时把相关页面换入内存。下面我们从 Linux 如何描述一个交换区开始。

本章首先讲述 Linux 所维护的每个活动交换区的结构以及如何在磁盘上组织交换区信息。然后是 Linux 如何在页面换出后的交换区定位该页,以及如何分配交换槽。接下来讨论交换高速缓存,它对共享页面非常重要。最后本章可以让你了解到如何激活和禁止交换区,内

存的页面如何换出到交换区又如何换入到内存,以及如何读写交换区。

11.1 描述交换区

每个活动的交换区,无论是一个文件或分区,都由一个 `swap_info_struct` 结构描述。系统中该结构都存储在一个静态声明的 `swap_info` 数组中,它有 `MAX_SWAPFILES`(一般被定义为 32)个元素项。这意味着运行系统中最多存在 32 个交换区。`swap_info_struct` 结构在<`linux/swap.h`>中声明如下:

```

64 struct swap_info_struct {
65     unsigned int flags;
66     kdev_t swap_device;
67     spinlock_t sdev_lock;
68     struct dentry * swap_file;
69     struct vfsmount * swap_vfsmnt;
70     unsigned short * swap_map;
71     unsigned int lowest_bit;
72     unsigned int highest_bit;
73     unsigned int cluster_next;
74     unsigned int cluster_nr;
75     int prio;
76     int pages;
77     unsigned long max;
78     int next;
79 };

```

下面我们简要解释这个比较大的结构的每个字段。

`flags`: 有两个可取的值。`SWP_USED` 表示此 swap 区域处于激活状态。`SWP_WRITEOK` 被定义为 3,最低两位包含了 `SWP_USED`。标志位为 `SWP_WRITEOK` 时,表示 Linux 准备对此交换区进行写操作,因为写之前必须先激活交换区。

`swap_device`: 该交换区所占用的磁盘设备信息,在交换区为普通文件时,其值为 `NULL`。

`sdev_lock`: 和 Linux 中其他结构一样,`swap_info_struct` 也需要保护。`sdev_lock` 就是一个用来保护此结构的自旋锁,它主要是保护 `swap_map`。它通过 `swap_device_lock()` 和 `swap_device_unlock()` 进行上锁和解锁。

`swap_file`: 实际的特殊文件作为交换区被挂载到系统的 `dentry`。如果交换区是一个分区的话,则 `swap_file` 是 `/dev` 目录下一个 `dentry`。在禁止一个交换区时,此字段用于确认正确的

swap_info_struct。

vfs_mount: 这是设备或文件作为交换区存储位置相对应的 vfs_mode 对象。

swap_map: 这是一个非常大的数组,其中一个项对应每个交换项,或者对应每个交换区中页面大小的槽。一个项作为页槽用户数量的引用计数。交换缓存以单个用户进行计数,这里一个 PTE 被换出到槽中时算一个用户。如果计数为 SWAP_MAP_MAX,将永久地分配该槽。而如果其值为 SWAP_MAP_BAD,将不再使用该槽。

lowest_bit: 交换区中最低的可用空闲槽,它一般在线性扫描减少搜索空间时开始。由此可知不可能有空闲槽低于该标记值。

highest_bit: 交换区中最高的可用空闲槽。如同 lowest_bit,高于该标记值不可能有空闲槽。

cluster_next: 下一个被使用的簇块偏移量。交换区通过在簇块中分配页面,使得相关页能有机会存储在一起。

cluster_nr: 簇内剩余的供分配的页面数量。

prio: 每个交换区都有一个优先级,就存储在该字段里面。交换区按照优先级顺序进行组织,并决定如何使用它们。缺省状态下,系统按照活跃程度的顺序来分配优先级,当然系统管理员也可以使用 swapon-f 来指定优先级。

pages: 由于可能不能使用交换文件中某些页面,此字段用于记录交换区中可用的页面数量。该值不同于 max,因为标记为 SWAP_MAP_BAD 的槽并没有计数。

max: 交换区中槽的总数。

next: swap_info 数组中用于指向系统中下一个交换区的下标。

虽然这个交换区存放在一个数组中,但它同时也存放在一个称为 swap_list 的伪链表中,它是一个很简单的类型,在<linux/swap.h>中声明如下:

```
153 struct swap_list_t {
154     int head; /* head of priority-ordered swapfile list */
155     int next; /* swapfile to be used next */
156 };
```

swap_list_t→head 字段使用具有最高优先级的交换区。swap_list_t→next 是下一个将被使用的交换区。尽管搜索一个合适的交换区要按照优先级顺序,但在必要的情况下,在数组中查询速度还是很快。

每个交换区在磁盘上都划分出大量页面大小的槽。例如在 X86 机上,每个页面大小为 4 096 个字节。第一个槽由于存放有交换区的基本信息,故被保留且不可写。该交换区的第一个 1 KB 用于存放分区的磁盘标签,为用户空间的工具提供信息。剩下的空间用于存放交换区的其他信息,在系统程序 mkswap 创建交换区时,这些剩余的交换区将被填充。交换区的信息由一个 union swap_header 表示,在<linux/swap.h>中定义如下:

```

25 union swap_header {
26 struct
27 {
28 char reserved[PAGE_SIZE - 10]; /* 交换文件的可用页位图 */
29 char magic[10];
30 } magic;
31 struct
32 {
33 char bootbits[1024];
34 unsigned int version;
35 unsigned int last_page;
36 unsigned int nr_badpages;
37 unsigned int padding[125];
38 unsigned int badpages[1];
39 } info;
40 };

```

各字段描述如下。

magic: 联合结构的 **magic** 字段, 仅用于鉴别 **magic** 字符串。这个字符串的存在使得分区必定有一个可使用的交换区, 并用于决定交换区的版本。如果字符串为 SWAP-SPACE, 则交换文件格式版本为 1, 如果为 SWAPSPACE2, 则版本为 2。由于数组保留得很大, 所以一般从页的末端开始读取该 **magic** 字符串。

bootbits: 保留区, 用于存放分区信息, 如磁盘标签。

version: 交换区的版本号。

last_page: 交换区中最后一个可用页面。

nr_badpages: 交换区中已知的坏页面数都存储在这个字段中。

padding: 通常一个磁盘扇区为 512 B。 **version**, **last_page** 和 **nr_badpages** 这 3 个字段占用了 12 B, **padding** 字段用于填充扇区剩下的 500 B。

badpages: 页面的剩下部分用于存放至多 **MAX_SWAP_BADPAGES** 个坏页槽。在系统程序 **mkswap** 打开 -c 开关检查交换区时, 填充这些槽。

MAX_SWAP_BADPAGES 是一个编译时常数, 它随着结构的改变而变化, 但是根据当前的结构, 由下面这个简单的公式可以知道它为 637。

$$\text{MAX_SWAP_BADPAGES} = \frac{\text{PAGE_SIZE} - 1024 - 512 - 10}{\text{sizeof}(\text{long})}$$

其中, 1 024 是 **bootlock** 的大小, 512 是 **padding** 的大小, 10 是 **magic** 字符串的大小, 其中 **magic** 字符串用于鉴别交换文件的格式。

11.2 映射页表项到交换项

在一个页面换出时, Linux 使用相应的页表项 PTE(page table entry)来存放足够用于再次在磁盘上定位该页的信息。很显然, PTE 本身并不能够大到精确保存交换页面位置的信息,但它只要存放交换槽在 swap_info 数组的下标以及在 swap_map 中的偏移量就够了。这正是 Linux 的处理方式。

每个 PTE,无论其结构如何,都必须大到能够存放一个 swap_entry_t 变量。swap_entry_t 在 <linux/shmem_fs.h> 定义如下:

```
16 typedef struct {
17     unsigned long val;
18 } swap_entry_t;
```

linux 分别提供了宏 pte_to_swap_entry() 和 swap_entry_to_pte() 来分别转换 PTE 到 swap_info 数组元素的映射和 swap_info 数组元素到 PTE 的映射。

在不同体系结构下 Linux 都必须可以确定 PTE 在内存还是已经换出。为了说明,我以 x86 为例。在 x86 中,swap_entry_t 中的第 0 位预留给_PAGE_PRESENT 标志位,第 7 位预留给_PAGE_PROTNONE 标志位,需要这两位的原因已在 3.2 节解释过。1~6 位标识 swap_info 数组中下标的类型,它由宏 SWP_TYPE() 返回。

8~31 位则代表交换文件内的页号,也就是在 swap_map 中的偏移量。在 x86 中,这意味着交换文件的页号占用 24 位,它限制了交换区大小为 64 GB。宏 SWP_OFFSET() 用于分离偏移。

为了将类型和其相应的偏移编号存放到 swap_entry_t 中,Linux 使用了宏 SWP_ENTRY()。这些宏之间的关系如图 11.1 所示。

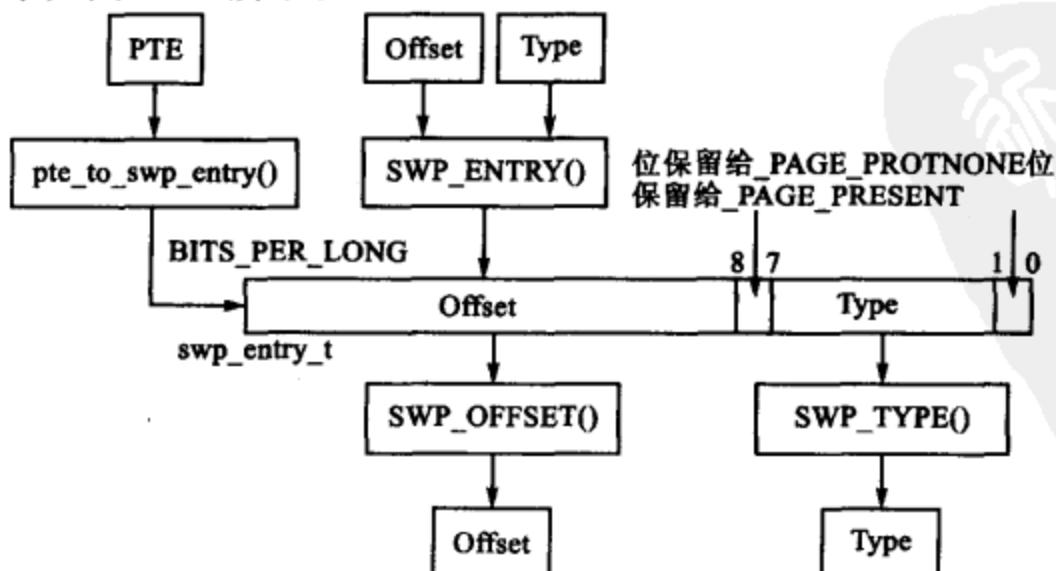


图 11.1 swap_entry_t 中存放的交换项信息

标识类型的那 6 位必须允许 32 位系统中存在 64 个交换区,而不是 MAX_SWAPFILES 的中限制的 32 个交换区。MAX_SWAPFILES 的限制是由于 vmalloc 地址空间有消耗。如果交换区具有可能的最大尺寸,则 swap_map 需要 32 MB($2^{24} * \text{sizeof}(\text{short})$)空间,不要忘记每个页面都有一个短整型的引用数。在 MAX_SWAPFILES 个最大尺寸的交换文件存在时,就要 1 GB 虚拟 malloc 空间,而这样分割用户内核线性地址空间几乎是不可能的。

这意味着不值得这样为支持 64 个交换区而增加系统处理复杂度,但是在某些情况下,即使没有增加整个有效交换区,存在大量的小交换区也可以提高系统执行效率。一些现代化的机器已经拥有许多独立的磁盘,可以在这些磁盘间创建大量的独立块设备。在这种情况下,创建大量分布在磁盘间的小交换区,可以大大加强页调度过程的并行度,这对那些交换密集的应用很重要。

11.3 分配一个交换槽

所有指定大小的槽都由数组 swap_info_struct → swap_map 跟踪,它的元素类型为 unsigned short。在共享页且 0 页空闲的情形下,数组中每项都是一个槽使用者的引用计数。如果项值为 SWAP_MAP_MAX,则对应的页将永久保留给这个槽。尽管元素值不可能为 SWAP_MAP_MAX,但还是这样做,主要是为防止引用数溢出。如果项值为 SWAP_MAP_BAD,则对应的页将不能再使用。



图 11.2 调用图: get_swap_page()

寻找和分配一个交换项的任务分为两个步骤。如图 11.2 所示,首先调用高层函数 get_swap_page()。从当前交换文件索引 swap_list_next 处开始在分配区域中寻找可分配的交换槽。找到后,记录下一个待用的交换区,并返回分配的表项。

scan_swap_map() 函数负责查找交换图。原理上,这是很简单的一步,因为它线性查找空闲槽并返回,可以肯定的是,它的实现还要彻底一点。

Linux 将磁盘中的 SWAPFILE_CLUSTER 个页分配到一个簇中。它在交换区中顺序分配 SWAPFILE_CLUSTER 个页面,然后在 swap_info_struct cluster_nr 记录簇中已分配的页面数,在 swap_info_struct cluster_next 中记录交换文件当前偏移量。在分配好一个连续的块后,系统寻找一个空闲的尺寸为 SWAPFILE_CLUSTER 的表项,供下一个簇使用。如果找到了一个足够大的块,系统将会把它作为另一个簇大小的序列。

如果交换区中找不到足够大的空闲簇,系统将从 swap_info_struct lowest_bit 位置进行首次空闲搜索。这样做的目的是使得在同一时间交换出去的页位置靠近,当然前提是同一时间

交换出去的页是相关的。这个前提初看上去很奇怪,但若考虑到页替换算法在线性扫描交换出去页的地址空间要使用很大的交换区,就很合理了。如果不需要扫描大量的空闲块并使用它们,则扫描就会退化成首次空闲搜索,速度也不需提高。但是若需要,退出中的进程可能释放很大的块的槽。

11.4 交换区高速缓存

前面讲过,Linux 无法快速完成页面结构到每个引用它的 PTE 的映射,所以,由多个进程共享的页面不能简单地换出。这就导致如果不同步磁盘数据,就没有条件判断某个 PTE 引用的页面是否因为其他进程的换出而得到了更新,因此会丢失更新。

为了解决这个问题,共享页在后援存储器中保留一个槽作为交换高速缓存的一部分。交换高速缓存有一些与其有关的 API,如表 11.1 所列。交换高速缓存是一个纯概念性的东西,因为它只是页面高速缓存的特殊形式。页面高速缓存页面与交换高速缓存页面的一个区别是交换高速缓存总是使用 `swapper_space` 作为 `page→mapping` 的地址空间。另外一个区别如图 11.3 所示,交换高速缓存中的页面通过 `add_to_swap_cache()` 加到交换高速缓存中,而不是使用 `add_to_page_cache()`。

匿名页只有在交换出去时才是交换高速缓存的一部分。同时这意味着,系统在首次写属于共享内存区的页时才把它们加入到交换高速缓存中。变量 `swapper_space` 在 `swap_state.c` 中如下声明:

```
39 struct address_space swapper_space = {
40 LIST_HEAD_INIT(swapper_space.clean_pages),
41 LIST_HEAD_INIT(swapper_space.dirty_pages),
42 LIST_HEAD_INIT(swapper_space.locked_pages),
43 0,
44 &swap_aops,
45 };
```

表 11.1 交换高速缓存 API

`swp_entry_t get_swap_page()`

此函数在活动交换区中查找 `swap_map`,然后在其中分配一个槽。它在 11.3 节中有详细描述,包含在这里是因为它主要用于交换 cache 的连接

`int add_to_swap_cache(struct page * page, swp_entry_t entry)`

这个函数向交换 cache 中加入一个页面。首先它调用 `swap_duplicate()` 检查页面是否存在,如果没有,它就调用普通的页面 cache 接口函数 `add_to_page_cache_unique()` 来向交换 cache 中加入该页面

续表 11.1

| | |
|---|--|
| struct page * lookup_swap_cache(swp_entry_t entry) | 搜索交换 cache 并返回所供表项的页面结构。它的方法是利用 swapper_cahe 和 swap_map 偏移搜索普通页面 cache |
| int swap_duplicate(swp_entry_t entry) | 该函数验证交换表项是否有效,如果不是,它就增加交换图计数 |
| void swap_free(swp_entry_t entry) | 这是一个 swap_duplicate() 的辅助函数,它将 swap_map 的相应计数器减 1。当计数减到 0 时,就释放页面槽 |

一个页面在 page mapping 设置为 swapper_space 时才成为交换高速缓存的一部分。swapper_space 是 address_space 类型,管理交换文件。宏 PageSwapCache() 测试 page mapping 是否为 swapper_space。Linux 中同步交换区和内存间页面的代码相同,因为它们都使用后援文件来实现同步。它们共享页面高速缓存代码,之间的区别仅是使用到的函数不同。

作为后援存储地址空间的 swapper_space 使用 swap_ops 作为它的 address_space→a_ops。在一般意义上,系统使用 page→index 字段而不是文件偏移来存储 swp_entry_t 结构。类型为 address_space_operations 的结构体 swap_aops 在 swap_state.c 中如下声明:

```
swap state.c:
34 static struct address_space_operations swap_aops = {
35     writepage: swap_writepage,
36     sync_page: block_sync_page,
37 };
```

当一个页面加到交换高速缓存中后,系统调用 get_swap_page() 来分配一个可用的交换页目录项,然后使用 add_to_swap_cache() 将它加入到页高速缓存中,接着把它标志为脏数据。脏页清洗程序处理该页时,就把它写入到磁盘中。这个过程如图 11.4 所示。

接下来共享 PTE 的页交换将调用 swap_duplicate(), 它仅将 swap_map 相应页的引用计数加 1。如果 PTE 由于被写过而被硬件标志为“脏数据”,则其对应的交换页目录项位将被清除,且页结构由 set_page_dirty 标志为“脏页”,这时系统在页被删除之前将进行磁盘复制同步。在删除对页的所有引用之前,系统会检查磁盘数据是否和页数据一致。

在页面的引用计数最后到 0 时,该页面就可以从页面高速缓存中删除,但交换映射计数将等于磁盘槽所属的 PTE 的计数,所以该槽将不会过早地被释放掉。而是由同一个 LRU 清除并在最后销毁,这种逻辑已在第 10 章中有描述。

另一方面,如果系统在已交换到交换区的页上产生页中断,则 do_swap_page() 函数将调用 lookup_swap_cache() 检查页是否在交换高速缓存中。如果在,则更新 PTE 指向该页,将页引用计数加 1,然后调用 swap_free() 释放交换槽。

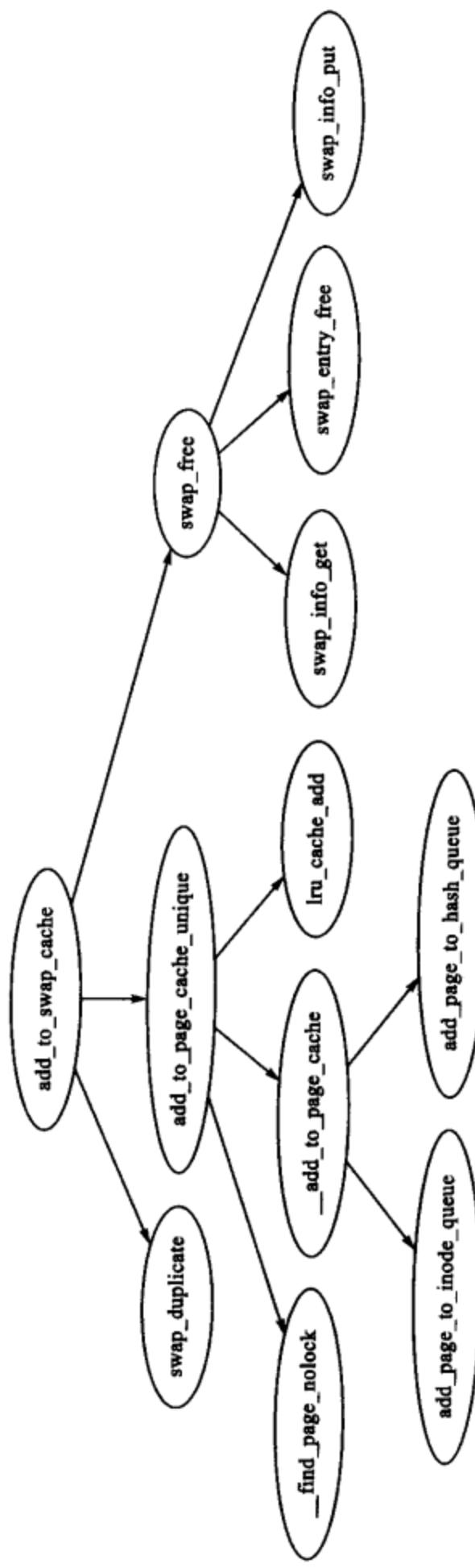


图 11.3 调用图：add_to_swap_cache()

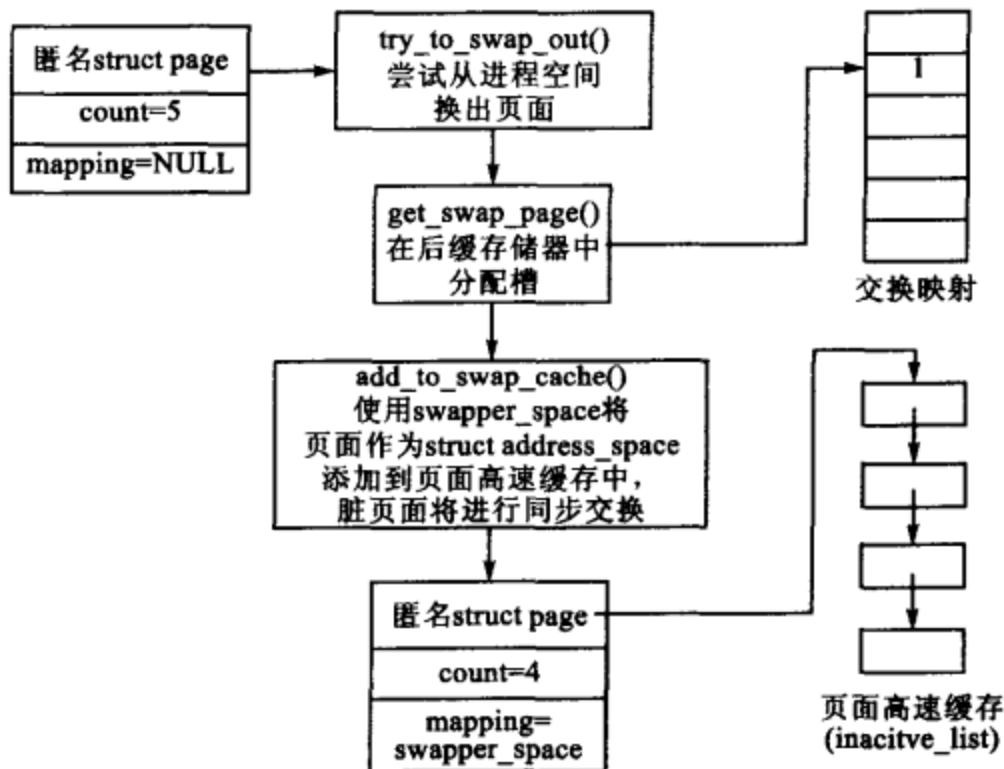


图 11.4 添加一个页面到交换高速缓存

11.5 从后援存储器读取页面

读页面时的主要函数是 `read_swap_cache_async()`，它主要在发生缺页中断时调用，如图 11.5 所示。此函数首先调用 `find_get_page()` 寻找交换高速缓存。通常情况下，交换高速缓存通过 `lookup_swap_cache()` 函数完成查找，但该函数在更新执行的查询时也会更新统计的数量。由于需要多次查询高速缓存，所以 Linux 使用了 `find_get_page()`。

如果其他进程有页面的映射，则该页面已经在交换区中或者多个进程同时都在同一页上发生缺页中断。如果交换高速缓存中不存在页面，系统就会从后援存储器中分配一页并填入数据。

由于在交换高速缓存中的操作都是对页面的操作，在 `alloc_page()` 分配页面后，就由 `add_to_swap_cache()` 将其加入到交换高速缓存中。如果不能把页面加入到交换高速缓存中，系统就会再次查找一遍交换高速缓存，保证其他进程不会向交换高速缓存中加入数据。

为了从后援存储器中读入信息，它会调用 `rw_swap_page()`，这将在 11.7 节讨论。在这个函数调用结束后，系统调用 `page_cache_release()` 释放 `find_get_page()` 找到的页面。

11.6 向后援存储器写页面

在向磁盘写页面时，系统就使用 `address_space->a_ops` 找到相应的写出函数。在使用后

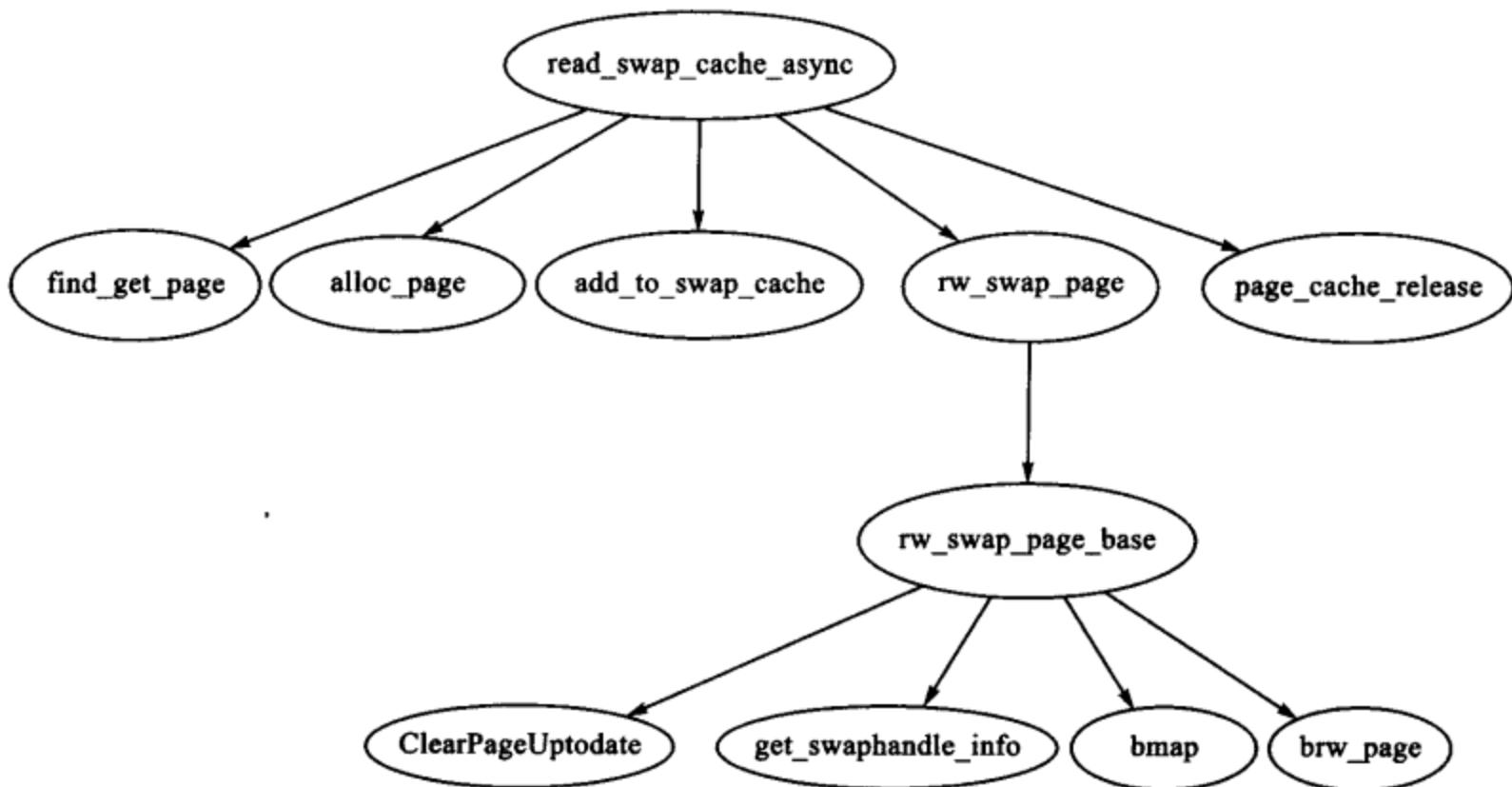


图 11.5 调用图：`read_swap_cache_async()`

授存储时，`address_space` 即为 `swapper_space`，`swap_aops` 中就包含有交换操作。由于 `swap_aops` 的写出函数的缘故，所以由 `swap_aops` 注册 `swap_writepage()` 函数，如图 11.6 所示。

函数 `swap_writepage()` 因为写进程是否是交换高速缓存页面的最后使用者的不同而表现为不同的操作，它通过调用 `remove_exclusive_swap_page()` 确定这一点。函数 `remove_exclusive_swap_page()` 通过检查被操作页获取的 `pagecache_lock` 的数量知道页面的引用计数，从而得知是否存在其他进程也在使用被操作页。如果没有，该页将从交换高速缓存中移走并释放。

在 `remove_exclusive_swap_page()` 将页面从交换高速缓存中移走并释放后，因为已经不再使用该页，所以 `swap_writepage()` 将释放页锁，唤醒所有等待该锁的进程。如果页面还在交换高速缓存中，则调用 `rw_swap_page()` 将页的内容写到备份空间中。

11.7 读/写交换区域的块

读写交换区的高层函数是 `rw_swap_page()`。此函数确保所有操作在交换高速缓存中完成以避免丢失更新。`rw_swap_page_base()` 是完成实际工作的核心函数。

它首先检查操作是否为读。如果是，则调用 `ClearPageUptodate()` 清除 `uptodate` 标志位，因为 I/O 请求写入数据的页面显然不是过时的页面。此标志位会在页成功后从磁盘读入时再次置位。然后调用 `get_swaphandle_info()` 获得文件索引节点的交换分区的设备。这些是

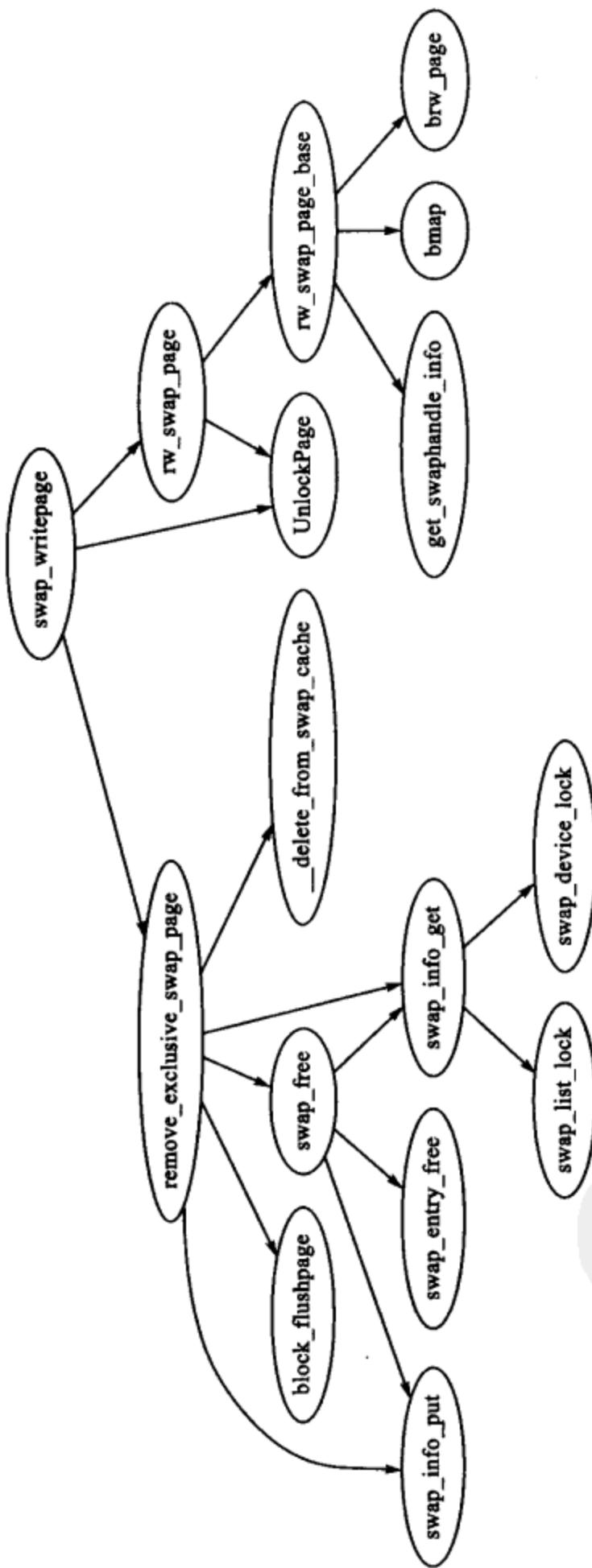


图 11.6 调用图: swap_writepage()

在块一级所需要的,而在这一级上有实际 I/O 操作。

核心函数既可以在交换分区运行,也可以在文件一级运行。因为它使用了块一级的函数 `brw_page()` 完成实际的磁盘 I/O。如果交换区是一个文件,就调用 `bmap()` 把操作页所在的文件系统所有块组成的列表填充为一个本地数组。请注意文件系统可能有自己存储文件和磁盘的方法,可能和磁盘分区的直接将信息写入到磁盘的方法不一致。如果后援存取区是一个分区,由于没有涉及到文件,就不需要 `bmap()`,这时只需要一次页面大小块的 I/O 操作。

在确定需要读入或写入一块时,系统使用 `brw_page()` 执行平常的块 I/O 操作。由于所有的 I/O 都是异步进行的,所以所有函数很快就可以返回。I/O 操作完成后,在块一层就会解锁页面,系统将唤醒所有处于等待状态的进程。

11.8 激活一个交换区

现在你已经知道什么是交换区,它是怎样组织以及页是怎样记录的,现在我们来看看它们是如何链接在一起以激活一个区域的。激活一个交换区的操作从概念上说很简单:打开一个文件,从磁盘获得交换首部的信息,填写 `swap_info`,将它加入到交换列表中。

函数 `sys_swapon()` 负责激活交换区。它有两个参数,交换区对应文件的路径和一组标志位。当系统激活交换区时,大内核锁(BKL, Big Kernel Lock)启动,阻止任何程序在该函数执行时进入到核心空间。该函数比较大,但可以分成下面几个简单的步骤:

- 在 `swap_info` 数组中找到一个空闲项,对其进行缺省初始化。
- 调用 `user_path_walk()`,该函数遍历提供的文件路径目录树,用文件的有效数据填充 `namidata` 结构,例如存放在 `vfsmount` 中的目录项和文件系统的信息。
- 填充涉及到交换区的大小和如何找到交换区的 `swap_info` 结构。如果交换区是一个分区,则块大小在计算大小之前为 `PAGE_SIZE`。如果是一个文件,信息直接从索引节点处获得。
- 确定空间是否未被激活。如果未激活,则从内存中分配一页并读取交换区第一页信息。这页包含很多信息,如可用可用槽,怎样用坏项填充 `swap_info` 中的 `swap_map`。
- 系统调用 `vmalloc()` 为 `swap_info_struct->swap_map` 分配内存,并将每个可用槽的项初始化为 0,不可用槽对应的项初始化为 `SWAP_MAP_BAD`。理想情况下,首部信息的文件格式为版本 2,因为版本 1 限制交换区在页大小为 4 KB 的系统结构下,小于 128 MB。如 x86。
- 验证头节点信息与实际的交换区匹配,在这之后,在 `swap_info_struct` 中填充类似页面最大数量和可用页面数等其余的信息,修改全局统计参数 `nr_swap_pages` 和 `total_swap_pages`。

- 至此,交换区已激活并初始化,在交换区的逻辑列表中插入代表此交换区的新元素,仍遵循优先级排序的顺序。

在函数结束时,释放 BKL,此时系统拥有一个新的用于换页的交换区。

11.9 禁止一个交换区

与激活一个交换区比,禁止交换区的代价非常高。这主要是因为交换区不能被随便地移去,每个交换出去的页必须再次交换回来。正是由于没有快速的方法把 `struct page` 映射到每个引用它的 PTE 上,所以也没有快速的方法映射交换项到 PTE。这要求遍历所有的进程页表以找到需要禁止的交换区的相关 PTE,然后将其交换回去。当然,这意味着如果物理内存被禁止,此操作将失败。

可以肯定,函数 `sys_swapoff()` 负责禁止交换区。该函数主要是更新 `swap_info_struct`。`try_to_unuse()` 负责将每个交换出去的页交换回来,但它执行的代价非常高。在 `swap_map` 中所使用的每一个槽,都必须遍历进程的页表来搜索它。在极端情况下,所有属于 `mm_structs` 的页表都需要被遍历。因此,一般地,禁止一个交换区的任务如下:

- 调用 `user_path_walk()` 获得需禁止的交换文件信息,并启动 BKL 将相应的 `swap_info_struct` 从交换列表中移走。并更新全局变量 `nr_swap_pages`(有效交换页)和 `total_swap_pages`(交换项总数)。完成后,释放 BKL。
- 从交换列表中释放 `swap_info_struct`,修改全局统计参数 `nr_swap_pages`(可用交换页面数)和 `total_swap_pages`(交换项总数),成功后,可以再次释放 BKL。
- 调用 `try_to_unuse()`,将需要禁止的交换区域中所有页交换回去。该函数遍历交换映射表并调用 `find_next_to_unuse()` 定位下个已用的交换页。对于找到的每个已用的页,执行下面的步骤:

① 调用 `read_swap_cache_async()` 为磁盘上分配保存该页的空间。理想情况下,空间应该已在交换高速缓存里分配好,若没有,将调用页分配器进行分配。

② 等待所有的页被交换回去,并对其加锁。加锁后,为每一个具有引用该页的 PTE 的进程调用 `unuse_process()`。该函数遍历页表查找相关的 PTE,然后更新它指向 `page`。如果页面是一个共享内存页,且没有其他的引用,则调用 `shmem_unuse()` 释放所有被永久映射的页。

③ 释放那些永久性映射的存储槽,有人认为存储槽不会被永久保留,但是这种风险仍然存在。

④ 如果在意外情况下交换映射表仍存在页的引用,则从交换高速缓存中删除此页以防止 `try_to_swap_out()` 引用该页。

- 如果没有足够的内存将所有的项交换回去, 则不能简单地删除交换区, 而是将其重新插入到系统中。如果成功地将所有项交换回去, 则 swap_info_struct 处于未定义状态, swap_map 的空间将由 vfree() 释放。

11.10 2.6 中有哪些新特性

为实现扩展区, 在 struct swap_info_struct 中改变的最重要的部分是增加了一个名为 extent_list 的链表, 以及一个叫做 curr_swap_extent 高速缓存字段。

扩展区由 struct swap_extent 表示, 它把交换区的一串连续的页面映射为磁盘上的一片磁盘块。这些扩展区由函数 setup_swap_extents() 在交换时启用。对一个块设备只启用一个扩展区, 这不是为了提高性能, 而是为了使系统一致对待块设备或普通文件所使用的交换区。

它们与交换文件有很大的不同, 交换文件会启用多个扩展区, 表示在块中连续的多个页面。当查找处于某个偏移值的文件时, 系统会遍历一遍扩展区列表。为减少搜索时间, 最后一次搜索的扩展区将会缓存在 swap_extent->curr_swap_extent。



第 12 章

共享内存虚拟文件系统

共享一个有后援文件或后援设备的内存区域仅需要调用 `mmap()` 设置 `MAP_SHARED` 标志位。但是,进程间共享一片匿名区域时存在两种重要情形:其一是在 `mmap()` 设置 `MAP_SHARED` 时没有后援文件,这片区域将在 `fork()` 执行之后由父、子进程共享;其二是该区域明确地利用 `shmget()` 设置,并由 `shmat()` 附加到虚拟地址空间中。

当 VMA 中的页面有磁盘上的后援文件时,所使用的接口很简单。系统在缺页时为了读入一页会调用 `vm_area_struct->vm_ops` 中的 `nopage()` 函数。要写一页到后援存储设备时,就使用 `inode->i_mapping->a_ops` 或 `page->mapping->a_ops` 在 `address_space_operations` 中寻找到相应的 `writepage()` 函数。当进行普通文件操作如 `mmap()`, `read()` 和 `write()` 时,系统则使用 `inode->i_fop` 等找到 `struct file_operations` 及其相应的函数。这些关系如图 4.2 所示。

这是一个相当清晰的接口,概念上也很容易理解。但它却无法处理匿名页的情况,因为匿名页没有后援文件。所以,为了保留这个清晰的接口, Linux 引入了一个基于 RAM 文件系统的人造文件,用于后援匿名页面,其中每个 VMA 都由这个文件系统中的一个文件作为后援。此文件系统的每个索引节点都被放置在 `shmem_inodes` 链表中,这样就很容易确定每个索引节点。通过引入这个概念, Linux 允许使用相同的基于文件的接口,而不是对匿名页作特殊处理。

这种文件系统有两个变种: `shm` 和 `tmpfs`。它们的核心功能相同,主要不同的地方是用途。内核利用 `shm` 为匿名页创建后援文件并由 `shmget()` 生成后援区域。该文件系统由 `kern_mount()` 加载,所以它在内部加载,并不为用户所见。`tmpfs` 是一个临时文件系统,它可以有选择地加载到 `/tmp/` 以获得一个基于 RAM 的临时文件系统。`tmpfs` 的另一个用途是被加载到 `/dev/shm/`。这时,在 `tmpfs` 文件系统中, `mmap()` 文件的进程之间可以共享信息,这是代替 System V 进程间通信(IPC)机制的一种方法。但是无论哪一种使用情形, `tmpfs` 必须显式地由系统管理员加载。

本章首先描述如何实现虚拟文件系统。接着讨论如何建立及销毁共享区域，最后说明怎样使用工具实现 System V IPC 机制。

12.1 初始化虚拟文件系统

如图 12.1 所示,在系统启动或载入模块时,由函数 `init_tmpfs()` 初始化虚拟文件系统。`init_tmpfs()` 注册两个文件系统 `tmpfs` 和 `shm`,并调用 `kern_mount()` 加载作为内部文件系统的 `shm`。然后计算文件系统中块和索引节点的最大数量。作为注册过程的一部分,它还调用 `shmem_read_super()` 作为一个回调函数来生成 `super_block` 结构,这个结构中保存有更多关于文件系统的信息,如保证块大小等于页面大小。

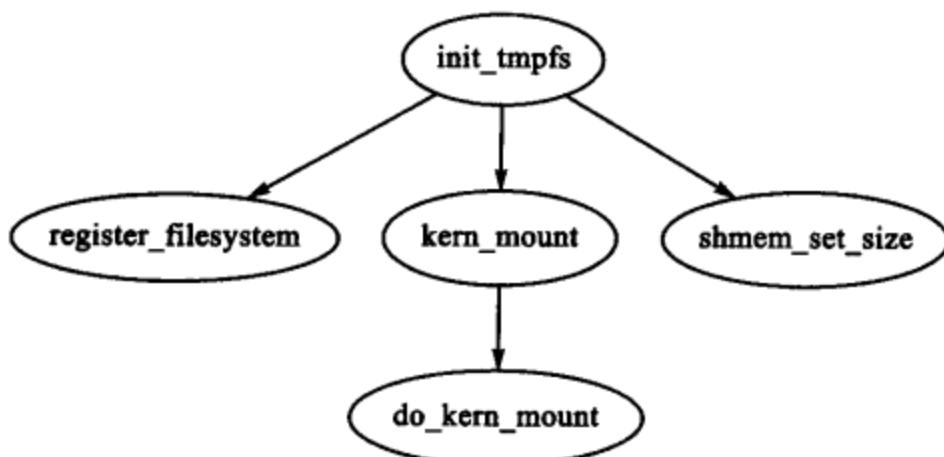


图 12.1 调用图: init_tmpfs()

文件系统创建的每一个索引节点都附带有 `shmem_inode_info` 结构，这个结构含有文件系统的私有信息。`SHMEM_I()` 函数以索引节点为参数，返回指针到此类型的结构中。`<linux/shmem_fs.h>` 将其如下定义：

```
20 struct shmem_inode_info {  
21     spinlock_t          lock;  
22     unsigned long        next_index;  
23     swp_entry_t          i_direct[SHMEM_NR_DIRECT];  
24     void                ** i_indirect;  
25     unsigned long        swapped;  
26     unsigned long        flags;  
27     struct list_head     list;  
28     struct inode          * inode;  
29 };
```

其中各字段含义如下。

lock: 为了保护索引节点信息不被并发访问的自旋锁。

next_index: 文件最后一页的索引。在文件被截断时它与 `inode->i_size` 是不同的。

`i_direct`: 一个直接块, 它存放有文件中用到的第一个 `SHMEM_NR_DIRECT` 交换向量。具体见 12.4.1 小节。

`i_indirect`: 指向首个间接块的指针。具体见 12.4.1 小节。

`swapped`: 记录文件当前换出的页面数量的计数。

`flags`: 目前仅用于记住文件是否属于 `shmget()` 建立的共享区域。它通过 `shmctl()` 指定 `SHM_LOCK` 锁来设置, 指定 `SHM_UNLOCK` 来解锁。

`list`: 文件系统中所有索引节点的列表。

`inode`: 父索引节点的指针。

12.2 使用 shmem 函数

包含 `shmem` 特定函数指针的结构有多种, 所有结构中 `tmpfs` 和 `shm` 都共享相同的结构。

Linux 中声明了 `hmem_aops` 和 `shmem_vm_ops`, 它们分别具有 `struct address_space_operations` 和 `struct vm_operations_struct`, 分别对应于换页中的缺页处理和写页面到后援存储区。

地址空间操作结构 `struct shmem_aops` 包含有指向一些函数的指针, 其中最重要的的是 `shmem_writepage()`, 它在从高速缓存移动页面到交换高速缓存时被调用。`shmem_removepage()` 在从页面高速缓存中移除页面时被调用, 这时系统可以回收存储块。`tmpfs` 不使用 `shmem_readpage()`, 但 Linux 还是提供了这个函数, 这样 `tmpfs` 文件上的系统调用 `sendfile()` 才可能被使用。同样 `tmpfs` 也没有使用 `shmem_prepare_write()` 和 `shmem_commit_write()`, 但 Linux 提供它们从而 `tmpfs` 可以被用作回环设备。`mm/shmem.c` 中, `shmem_aops` 有如下声明:

```

1500 static struct address_space_operations shmem_aops = {
1501     removepage: shmem_removepage,
1502     writepage: shmem_writepage,
1503 # ifdef CONFIG_TMPFS
1504     readpage: shmem_readpage,
1505     prepare_write: shmem_prepare_write,
1506     commit_write: shmem_commit_write,
1507 # endif
1508 };

```

匿名 VMA 使用 shmem_vm_ops 作为 vm_operations_struct, 所以在中断陷入时系统调用 shmem_nopage()。它如下声明:

```
1426 static struct vm_operations_struct shmem_vm_ops = {
1427     nopage: shmem_nopage,
1428 };
```

在文件和索引节点上完成操作时需要有两个结构: file_operations 和 inode_operations。file_operations 由 shmem_file_operations 调用, 它提供函数实现 mmap(), read(), write() 和 fsync()。它如下声明:

```
1510 static struct file_operations shmem_file_operations = {
1511     mmap: shmem_mmap,
1512     #ifdef CONFIG_TMPFS
1513     read: shmem_file_read,
1514     write: shmem_file_write,
1515     fsync: shmem_sync_file,
1516     #endif
1517 };
```

Linux 提供了 3 种 inode_operations: 第 1 种是 shmem_inode_operations, 它在文件索引节点中使用; 第 2 种是 shmem_dir_inode_operations, 它针对目录; 第 3 种是 shmem_symlink_inline_operations 和 shmem_symlink_inode_operations 的组合, 它针对符号链接。

在名为 shmem_inode_operations 的 inode_operations 结构中存放有 Linux 支持的两种文件操作, 即 truncate() 和 setattr()。shmem_truncate() 用于截断文件。shmem_notify_change() 在文件属性改变时被调用, 从而使文件中的其他部分通过 truncate() 得以增长并使用全局零页面作为数据页面。shmem_inode_operations 如下声明:

```
1519 static struct inode_operations shmem_inode_operations = {
1520     truncate: shmem_truncate,
1521     setattr: shmem_notify_change,
1522 };
```

目录 inode_operations 提供了诸如 create(), link() 和 mkdir() 的函数, 它们如下声明:

```
1524 static struct inode_operations shmem_dir_inode_operations = {
1525     #ifdef CONFIG_TMPFS
1526     create: shmem_create,
1527     lookup: shmem_lookup,
1528     link: shmem_link,
1529     unlink: shmem_unlink,
```

```

1530 symlink: shmem_symlink,
1531 mkdir: shmem_mkdir,
1532 rmdir: shmem_rmdir,
1533 mknod: shmem_mknod,
1534 rename: shmem_rename,
1535 #endif
1536 };

```

最后一对操作用于符号链接。它们如下声明：

```

1354 static struct inode_operations shmem_symlink_inline_operations = {
1355 readlink: shmem_readlink_inline,
1356 follow_link: shmem_follow_link_inline,
1357 };
1358
1359 static struct inode_operations shmem_symlink_inode_operations = {
1360 truncate: shmem_truncate,
1361 readlink: shmem_readlink,
1362 follow_link: shmem_follow_link,
1363 };

```

函数 `readlink()` 和 `follow_link()` 之间的差异与链接信息存放的位置有关。符号链接索引节点不需要私有索引节点信息结构 `shmem_inode_information`。如果符号链接名的长度小于这个结构，则索引节点中的空间用于存放名字，而 `shmem_symlink_inline_operations` 会变成索引节点操作结构。否则，`shmem_getpage()` 开辟一个页面，把符号链接复制到页面中，索引节点操作结构定义为 `shmem_symlink_inode_operations`。第 2 个结构中包括 `truncate()` 函数，可以在删除文件时回收页面。

这些不同的结构保证了与索引节点相关操作等价的 `shmem` 会在共享区域有后援虚拟文件时被用到。当用到它们时，VM 的主要部分中不会出现由实际文件作后援的页面和由虚拟文件作后援的页面之间的差异。

12.3 在 tmpfs 中创建文件

由于 `tmpfs` 作为一种用户可见的文件系统加载，因此它必须支持目录索引节点操作，如 `open()`，`mkdir()` 和 `link()`。实现 `tmpfs` 的函数指针由 `shmem_dir_inode_operations` 提供，详见 12.2 节。

这些函数中大部分的实现都不完全，在某种层次上，它们之间的交互如图 12.2 所示。它

们实现虚拟文件系统中的索引节点相关操作的基本原理都是相同的，并且大部分索引节点字段都由 `shmem_get_inode()` 填充。

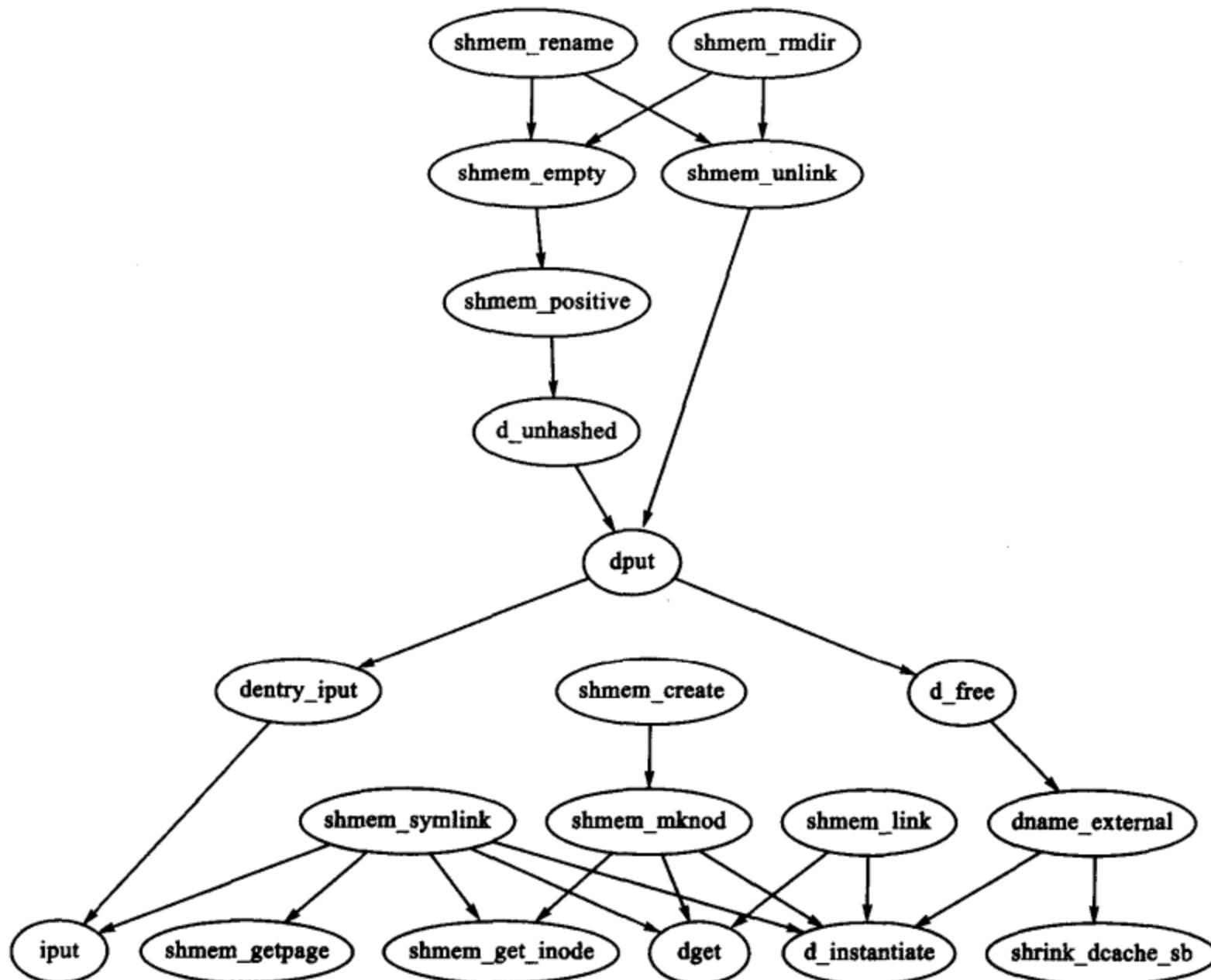


图 12.2 调用图: `shmem_create()`

在创建一个新文件时，调用的顶层函数是 `shmem_create()`，它调用 `shmem_mknod()`，设置 `S_IFREG` 标志位，从而创建一个普通文件。`shmem_mknod()` 仅是 `shmem_get_inode()` 的一个包装器，可以确定，它创建一个新的索引节点并填充结构字段。主要要填充的 3 个字段是 `inode->i_mapping->a_ops`，`inode->i_op` 和 `inode->i_fop` 字段。在创建索引节点之后，`shmem_mknod()` 会更新目录索引节点大小并统计参数 `mtime`，然后实例化这个新的索引节点。

即使各文件系统的功能本质上相同，shm 中文件的创建也各不相同。如何创建这些文件将在 12.7 中详细讨论。

12.4 虚拟文件中的缺页中断

发生缺页中断时,如果 `vma->vm_ops->nopage` 操作存在, `do_no_page()` 将会调用该操作。在虚拟文件系统的情形下,这意味着函数 `shmem_nopage()` 以及它的调用图中函数(如图 12.3 所示)将在缺页时被调用。

在这种情形下的核心函数是 `shmem_getpage()`,它负责分配新页或者在交换区中找到该页。这种对错误类型的重载是不寻常的,因为 `do_swap_page()` 一般负责使用 PTE 中的编址信息来定位那些已经移到交换高速缓存或后备存储区的页面。在这种情况下,有后援虚拟文件的页面在被移到交换高速缓存中时其 PTE 是被设为 0 的,而索引节点的私有文件系统数据储存了直接或间接的关于块的信息,这些信息将被用来定位页面。这个操作在很多方面都与普通缺页中断处理类似。

12.4.1 定位交换页面

当一个页面被换出后, `swp_entry_t` 就会包含再次定位该页面所需的信息。这些信息存放在索引节点中作为文件系统特定的私有信息,而不是使用 PTE 来完成这个工作。

在发生缺页时,系统会调用 `shmem_alloc_entry()` 来定位交换项。它的基本任务是进行基本的检查并保证 `shmem_inode_info->next_index` 始终指向虚拟文件末端的页面索引。它的主要任务是调用 `shmem_swp_entry()` 得到索引节点信息,然后查找索引节点信息中的交换向量,并且在需要时分配新页来存放交换向量。

第一个 `SHMEM_NR_DIRECT` 项存放在 `inode->i_direct` 中。这意味着对 x86 而言,小于 $64\text{ KB}(\text{SHMEM_NR_DIRECT} * \text{PAGE_SIZE})$ 的文件将不需要使用间接块。更大的文件则必须使用从 `inode->i_indirect` 指向位置开始的间接块。

刚开始间接块(`inode->i_indirect`)分为两半。一半包含指向二次间接块的指针,另一半则包含指向三次间接块的指针。二次间接块由包含交换向量(`swp_entry_t`)的页面组成。三次间接块包含由交换向量填充的页面指针。图 12.4 中显示了间接块不同层次之间的关系。这种关系意味着在一个虚拟文件(`SHMEM_MAX_INDEX`)中页面的最大数量(在 `mm/shmem.c` 中定义)为:

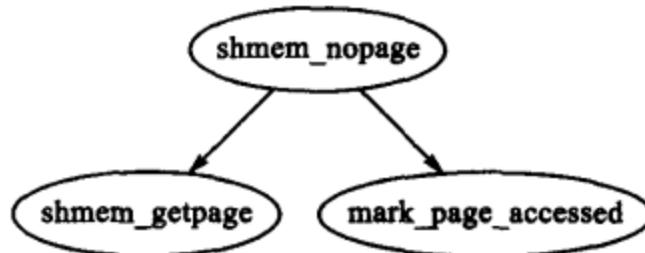


图 12.3 调用图: `shmem_nopage()`

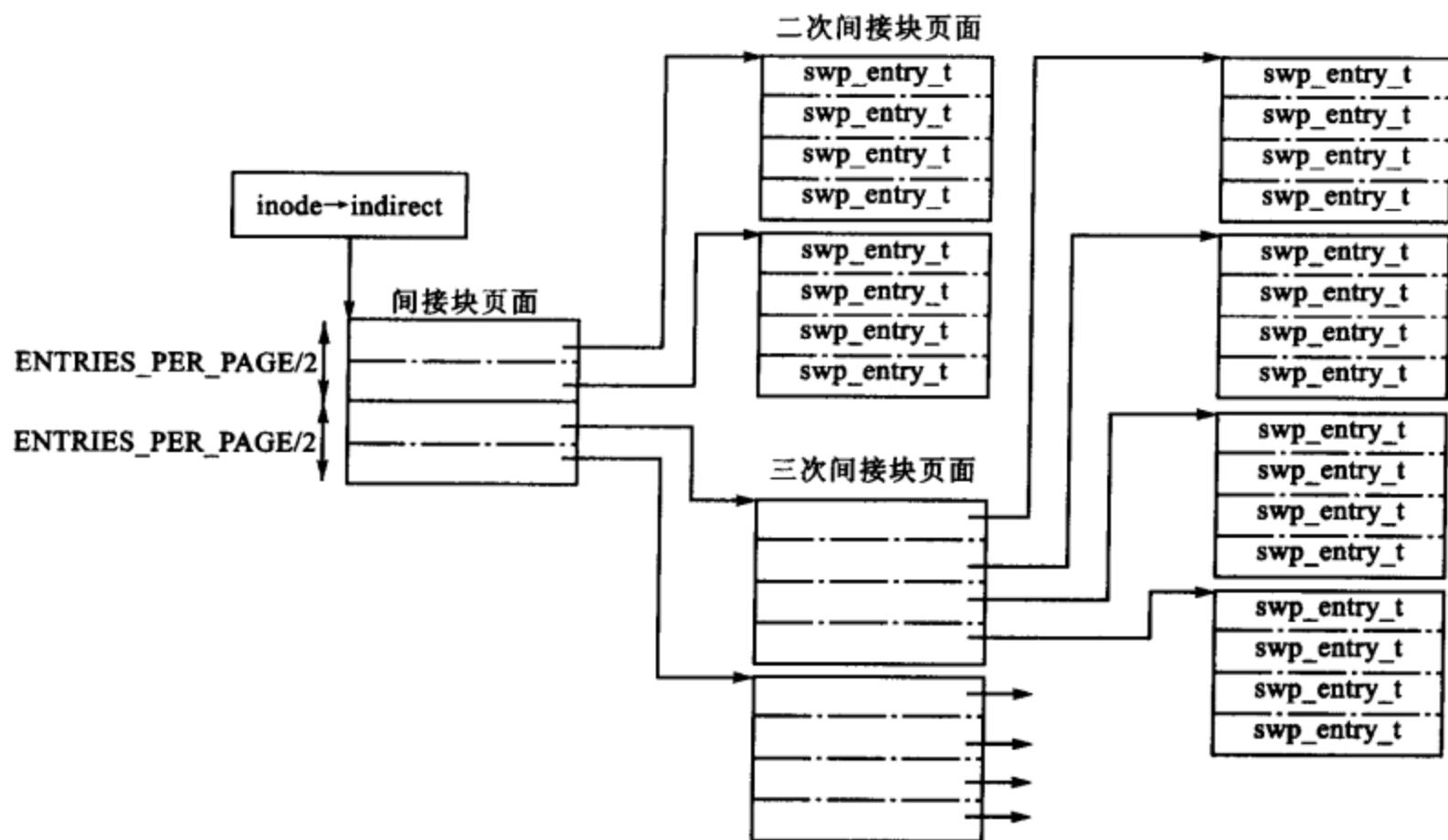


图 12.4

```
44 #define SHMEM_MAX_INDEX (SHMEM_NR_DIRECT + (ENTRIES_PER_PAGE/2) * (ENTRIES_PER_PAGE + 1))
```

12.4.2 向交换区写页面

在虚拟文件系统中,由 `address_space_operations` 的注册函数 `shmem_writepage()` 向交换区写页面。这个函数负责把页面从页面高速缓存移到交换高速缓存。过程包括以下几步:

- 记录当前 `page->mapping` 和索引节点相关的信息。
- 调用 `get_swap_page()` 在后援存储区中分配一空闲槽。
- 调用函数 `shmem_swp_entry()` 分配 `swp_entry_t`。
- 从页面高速缓存中移除该页。
- 将该页加入到交换高速缓存中。如果有失败,则释放交换槽,将页面加入到页面高速缓存中并再次进行这步操作。

12.5 tmps 中的文件操作

虚拟文件支持四种操作, `mmap()`, `read()`, `write()` 和 `fsync()`。这些函数的指针都存放在 `shmem_file_operations` 中, 在 12.2 节中已有叙述。

这些操作的实现都很普通, 在代码注释部分也会有详细阐述。`mmap()` 操作由 `shmem_mmap()` 实现, 它仅是更新负责那片映射区域的 VMA。`read()` 操作由 `shmem_read()` 实现, 它从虚拟文件中读数据到用户空间的缓冲区中, 并在需要时产生缺页中断。`write()` 由 `shmem_write()` 实现, 本质上与 `read()` 类似。`fsync()` 操作由 `shmem_file_sync()` 实现, 但它实际上是一个 NULL 操作, 因为它不做任何事, 仅返回 0 表示成功。由于这些文件只存在于 RAM 中, 所以它们不需要与磁盘同步。

12.6 tmpfs 中的索引节点操作

支持索引节点的最复杂的操作是截断, 它涉及 4 个不同的阶段。第 1 个阶段是在 `shmem_truncate()` 中, 它会截断文件末尾的部分页面, 并不停地调用 `shmem_truncate_indirect()`, 直到文件被截断到合适大小。每一次调用 `shmem_truncate_indirect()` 只能对一个间接块进行处理, 这也是它可能需要多次调用的原因。

第 2 个阶段, 是在 `shmem_truncate_indirect()` 中, 处理二次和三次间接块。它找到下一个需要被截断的间接块。这个将被传到第 3 个阶段的直接块将包含含有交换向量的页面指针。

第 3 个阶段, 是在 `shmem_truncate_direct()` 中处理包含交换向量的页面。它选出一个需要截断的范围并且将这个范围传到最后一个阶段 `shmem_swp_free()`。最后一个阶段调用 `free_swap_and_cache()` 来释放表项, 包括释放交换区表项以及包含数据的页面。

文件的链接和断开操作很简单, 因为大部分的工作由文件系统层完成。要链接一个文件, 需要增大目录索引节点的大小, 更新相应索引节点的 `ctime` 和 `mtime` 参数, 以及增加将要链接的索引节点的链接节点数量。然后调用 `dget()` 指向新表项, 接着调用 `d_instantiate()` 来实例化新表项。断开操作同样会更新相应索引节点的参数, 再减少将要链接的索引节点的链接节点数量, 然后调用 `dput()` 来减少引用。`dput()` 也会调用 `iput()` 在引用计数到 0 时清除该索引节点。

创建目录的操作将通过调用 `shmem_mkdir()` 来完成。它调用 `shmem_mknod()` 设置 `S_IFDIR` 标志位, 然后增加父目录索引节点的 `i_nlink` 计数。函数 `shmem_rmdir()` 在调用 `shmem_empty()` 清空目录之后再删除此目录。如果目录是空的, `shmem_rmdir()` 会将父目录

索引节点的 `i_nlink` 计数减 1, 然后调用 `shmem_unlink()` 来移除目录。

12.7 建立共享区

共享区由 `shm` 创建的文件作后援。存在创建文件的两种情形: 一种是在 `shmget()` 建立共享区时, 另一种是在 `mmap()` 设立 `MAP_SHARED` 标志位来启动匿名共享区时。这两个函数都使用核心函数 `shmem_file_setup()` 创建文件。

由于文件系统是在内部的, 所以所要创建的文件的名字不需要惟一(文件总是以索引节点来定位的, 不是名字)。因此, `shmem_zero_setup()`(如图 12.5 所示)据说总是创建一个名为 `dev/zero` 的文件, 在文件 `/proc/pid/maps` 中它就是这样出现的。`shmget()` 创建的文件叫做 `SYSVNN`, 其中 `NN` 是传递给 `shmget()` 的一个参数。

核心函数 `shmem_file_setup()` 仅创建一个目录项和索引节点, 然后填充相关的字段并实例化。

12.8 System V IPC

IPC 实现的内部机制超出了本书的范围。这一节仅集中讨论 `shmget()` 和 `shmat()` 的实现以及它们如何受到 VM 的影响。如图 12.6 所示, 系统调用 `shmget()` 由 `sys_shmget()` 实现。它进行基本的参数检查, 然后建立 IPC 相关的数据结构。为了创建段, 它调用 `newset()`, 就是由这个函数在 `shmfds` 中调用前面节中讨论过的 `shmem_file_setup()` 来创建文件。

系统调用 `shmat()` 由 `sys_shmat()` 实现。对这个函数没有什么要注意的地方。它获得适当的描述符, 并且保证所有的参数都有效, 然后调用 `do_mmap()` 把共享区映射到进程地址空间。这个函数中仅有两点需要注意。

首先, 如果调用者指明了地址, 必须保证 VMA 不会被覆盖。其次 `shp->shm_nattch` 计数由一个类型为 `shm_vm_ops` 的 `vm_operations_struct()` 来维护, 它分别注册了 `open()` 和 `close()` 的回调函数 `shm_open()` 和 `shm_close()`。回调函数 `shm_close()` 还负责在指明 `SHM_DEST` 标志位且 `shm_nattch` 计数到 0 时销毁共享区。

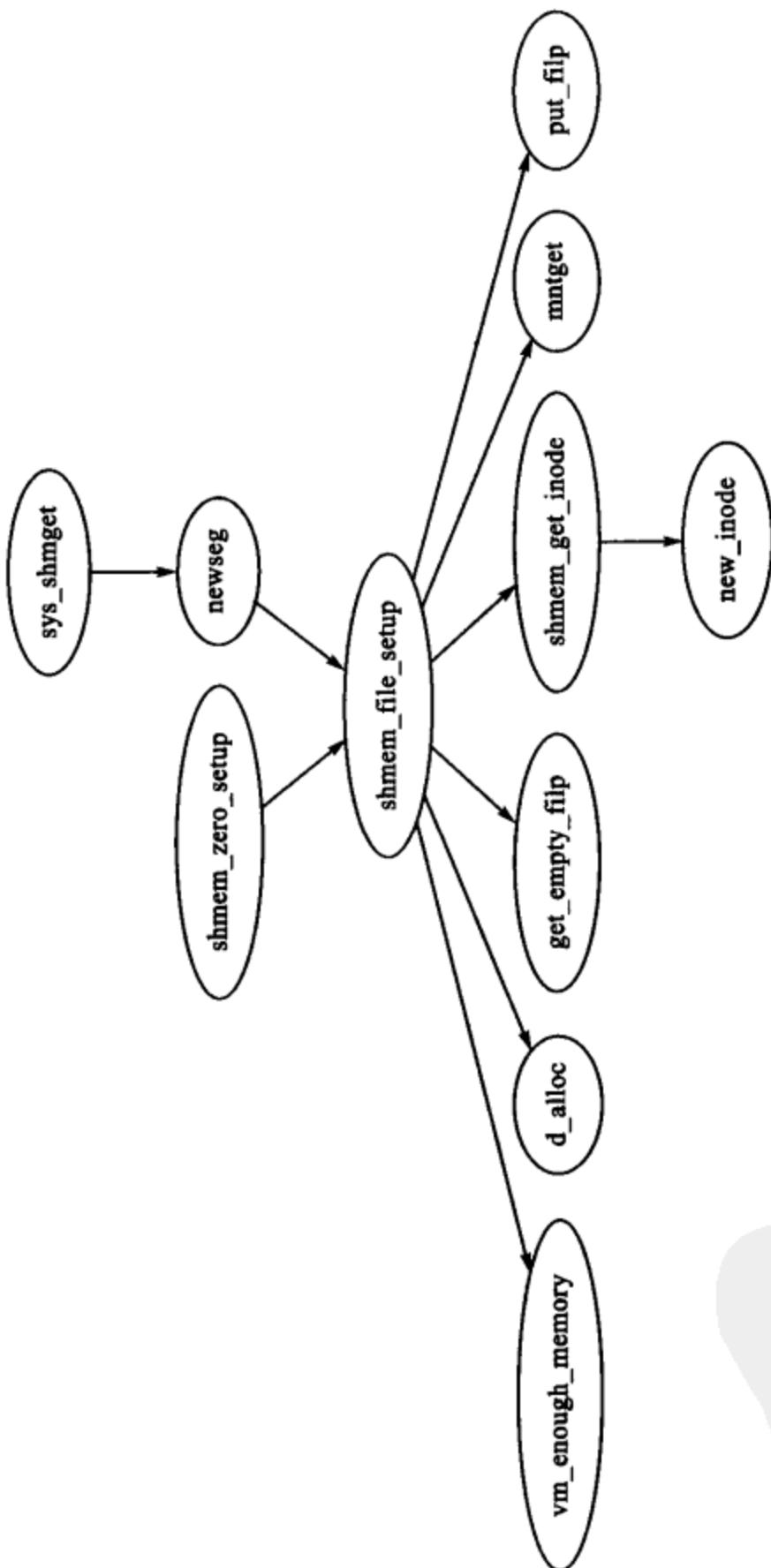


图 12.5 调用图: shmem_zero_setup()

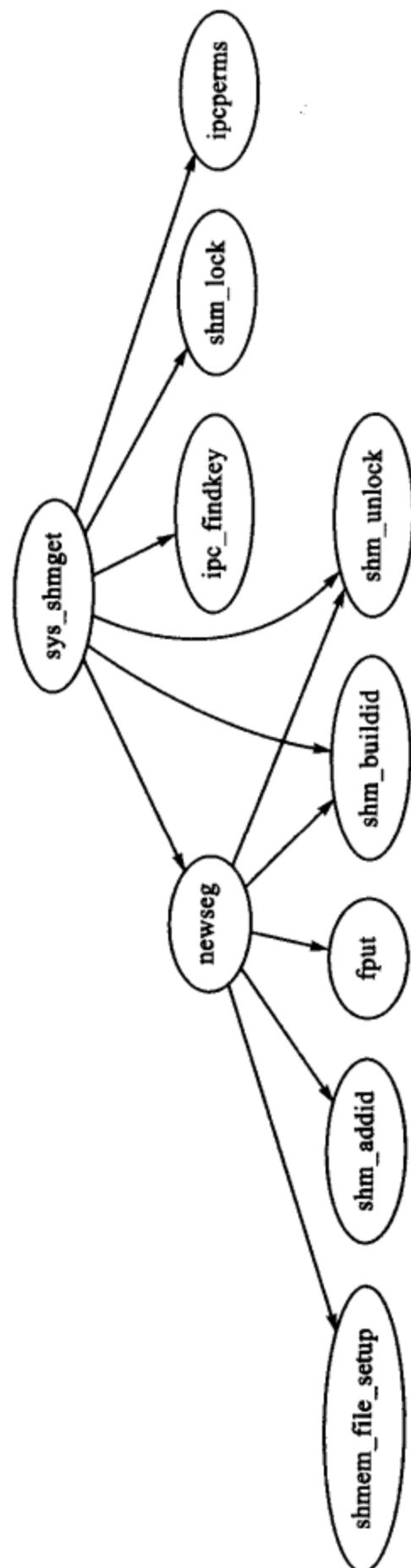


图 12.6 调用图：`shmem_zero_setup()`

12.9 2.6 中有哪些新特性

在 2.6 中,文件系统的核心概念和功能保持不变,改变的部分仅是优化或者对文件系统功能的扩展。如果读者能够很好地理解 2.4 中的实现,那么对 2.6 中的实现的理解就不会构成很大的障碍。

在 `shmem_inode_info` 中增加了一个叫做 `alloced` 的字段。它存储了分配给这个文件的页数,在 2.4 中这是需要通过 `inode->i_blocks` 的值在运行时计算的。同时它保存了一些共同操作的时钟周期,从而使代码更具可读性。

2.4 中的 `flags` 字段现在使用 `VM_ACCOUNT` 标志位以及 `VM_LOCKED` 标志位。系统总是要设置 `VM_ACCOUNT`,表明 VM 将仔细地计算所需的内存从而确保分配不会失败。

对文件操作的扩展能通过系统调用 `_llseek()` 来定位内容, `_llseek` 由 `generic_file_llseek()` 实现。还有虚拟文件上调用 `sendfile()`,它由 `shmem_file_sendfile()` 实现。还有 2.6 中允许非线性映射的对 VMA 操作的扩展,它由 `shmem_populate()` 实现。

最后一个主要的改变是文件系统负责分配和回收它自己的索引节点,这涉及到在结构 `super_operations` 中的两个回调函数。它们通过调用 `shmem_inode_cache` 来创建一个 slab 高速缓存实现。在这个 slab 分配器上注册一个 `init_once()` 构造函数来初始化每个新索引节点。



第 13 章

内存溢出管理

现在要讨论的有关 VM 的最后一部分是内存溢出 (OOM, Out Of Memory) 管理。这一章写得比较短, 因为检查系统是否有足够的内存、是否真的溢出以及在内存溢出时杀死一个进程等都是比较简单任务。这部分内容在 VM 中颇具争议, 所以很多场合中都会删除这部分。但我认为无论它在以后的内核中是否会出现, 它都是一个需要考虑的重要子系统, 因为它涉及到一系列其他的子系统。

13.1 检查可用内存

对于某些操作, 如扩展堆的 `brk()` 和重映射地址空间的 `mremap()`, 系统都会检查是否有足够的空间来满足请求。注意这与下一节中的 `out_of_memory()` 路径是有差异的。`out_of_memory()` 路径用于在可能的时候避免系统处于 OOM 状态。

在进行可用内存检查时, 系统将所请求的页面数作为参数传递给 `vm_enough_memory()`, 然后检查是否可以满足该请求。除非系统管理员指定了系统可以多用内存, 否则系统都将进行可用内存的检查。为了确定系统所有可用的页面数量, Linux 把以下数据相加。

页面高速缓存容量: 因为页面高速缓存很容易就可以回收。

空闲页面数: 因为它们已经处于可用状态。

空闲交换页面数: 因为可以换出用户空间页面。

`swapper_space` 管理的页面数量: 其对空闲交换页面双倍计数。但由于槽要保留起来, 而且不曾用到, 因此该方式是比较妥当的。

`dentry` 高速缓存使用的页面数: 因为很容易回收它们。

`inode` 高速缓存使用的页面数: 因为也很容易回收它们。

如果加起来的页面数对请求页面数来说已经足够,则 `vm_enough_memory()`会返回 `true` 给调用者。如果调用者获得了 `false` 的返回值,则它知道系统当前空闲页面不足,这时它就会决定是否向用户空间返回-ENOMEM 错误。

13.2 确定 OOM 状态

在机器内存不足时,系统就会回收旧的页面帧(见第 10 章),但是在这个过程中可能会发现,系统即使是以最高优先级扫描都无法释放足够的页面来满足请求。如果系统不能够释放页面,就会调用 `out_of_memory()`,告知系统发生内存溢出,这时需要杀死某个进程。函数调用图如图 13.1 所示。

不幸的是,系统可能并没有发生内存溢出,而只是等待 I/O 操作完成,或者将页面换出到后备存储器。这很糟糕,不是因为系统没有内存,而是因为函数被不必要地调用,它将导致进程可能会被不必要地杀掉。所以在决定杀死某个进程之前,系统会做如下检查:

- 有足够的换出空间留下吗(`nr_swap_pages > 0`)? 如果是,不是 OOM;
- 自从上次失败已经大于 5 s 了? 如果是,不是 OOM;
- 在上一秒中失败过? 如果不是,不是 OOM;
- 如果在至少上 5 s 中没有 10 个错误,不是 OOM;
- 在上 5 s 中有进程被杀死? 如果是,不是 OOM。

仅当上面的检查都通过时,系统才会调用 `oom_kill()`选择一个进程杀死。

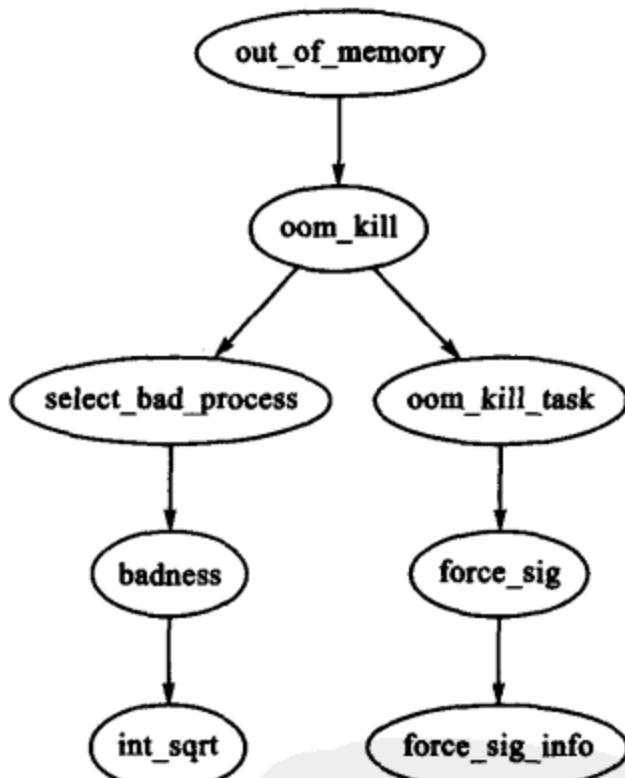


图 13.1 调用图: `out_of_memory()`

13.3 选择进程

函数 `select_bad_process()` 负责选择一个进程杀死。它逐个检查各个正在运行的进程,并调用函数 `badness()`计算各个进程是否合适被杀死。`badness()`的计算方法如下,注意方根是

使用 `int_sqrt()` 计算出来的整型估计数值。

$$\text{badness_for_task} = \frac{\text{total_vm_for_task}}{\sqrt{(\text{cpu_time_in_seconds})} * \sqrt[4]{(\text{cpu_time_in_minutes})}}$$

它所选择的是那个使用了最大量的内存空间而没有生存很久的进程,因为那些已经生存了很久的进程不像是导致内存不足的原因,所以这个计算选择一个使用了大量的内存而没有生存很久的进程。如果该进程是一个根进程或者具有 `CAP_SYS_ADMIN` 能力,数值就会除以 4,因为具有根特权的进程被认为是表现良好的。类似地,如果该进程具有 `CAP_SYS_RAWIO`(访问源设备的能力),数值会再除以 4,因为杀死一个可以访问硬件的进程是不值得的。

13.4 杀死选定的进程

一旦选定一个进程,系统会再一次遍历该列表,发信号给所有与该进程共享相同的 `mm_struct` 的进程(或它们都是线程)。如果该进程具有 `CAP_SYS_RAWIO` 能力,系统将会发出一个 `SIGTERM` 信号,给它一个完全结束的机会,否则就会发出一个 `SIGKILL` 信号。

13.5 是这样吗?

是这样的。OOM 管理涉及到很多其他的子系统,但是又没有那么多。

13.6 2.6 中有哪些新特性

除引入了 VM 占用对象以外,2.6 内核中的 OOM 管理几乎保持不变。在 4.8 节中曾经提到过 `VM_ACCOUNT`,在这里用许多 VMA 表示。系统在这些标识空间中对 VMA 进行操作时,就进行额外的检查以确保有足够的内存。这种复杂操作背后的动机是要争取避免 OOM 终止进程。

含有 `VM_ACCOUNT` 标志位的区域有:进程栈,进程堆,`mmap()` 使用 `MAP_SHARED` 映射的区域,可写的私有区域,以及建立 `shmget()` 的区域,或者说,大部分用户空间区域都含有 `VM_ACCOUNT` 标志位。

Linux 所占内存中的 VMA 数量由 `vm_acct_memory()` 确定, `vm_acct_memory()` 会增加变量 `committed_space` 的大小。释放 VMA 时, `vm_unacct_memory()` 会减少 `committed_`

space 的大小。这是一个相当简单的机制,不过这个机制要求在决定是否分配用户空间时 Linux 必须记住多少内存已经用于用户空间。

系统通过调用 `security_vm_enough_memory()` 实现这项检查, `security_vm_enough_memory()` 引入了另外的新特性。2.6 中用到的一个新特性是与安全相关的内核模块重载某些内核函数。类型为 `security_opt` 的结构体 `security_operations` 就存放有可用的钩子函数的列表。`security/dummy.c` 中存放了可能用到的大量的哑元和缺省函数,但大部分只是返回。如果没有载入安全模块,则使用 `dummy_security_ops`, 它是一个使用所有缺省函数的 `security_operations` 结构。

缺省情况下, `security_vm_enough_memory()` 调用 `security/dummy.c` 中声明的 `dummy_vm_enough_memory()`, 它与 2.4 中的 `vm_enough_memory()` 函数非常相似。新的版本中加入下列的信息一起来决定可用的内存。

页面高速缓存的大小: 因为页面高速缓存很容易就可以回收。

空闲页面的总数: 因为它们已经处于可用状态。

空闲交换页面的总数: 因为用户空间的页面可能会被换出。

设置了 `SLAB_RECLAIM_ACCOUNT` 的 slab 页面: 因为可以很容易回收它们。

这些页面中除减去 3% 保留在 ROOT 进程的那部分外就是当前请求的可用内存总数。如果有足够的内存,系统就检查一次以保证分配的内存不会超过允许的阈值 $\text{TotalRam} * (\text{OverCommitRatio}/100) + \text{TotalSwapPage}$, 其中 `OverCommitRatio` 由系统管理员设定。如果分配的空间总量不是太大,就会返回 1 从而开始分配。

第 14 章

结束语

毫无疑问,内存管理是一个既庞大又复杂的部分,研究起来很费时,并且很难在实际中应用和实现。这是因为在实时、多进程环境中建模系统行为比较困难 [CD80],在这种环境下开发者不得不依靠直觉来引导,而且理论上验证虚拟内存算法的效果往往取决于在特定负载下对它的模拟。由于调度、换页操作以及多进程交互建模所带来的巨大挑战,所以模拟方法也是需要的。换页策略这一研究热点就是个极好的例子,因为它仅在特定负载下才能显示出良好的运行结果。在不同负载下,调整算法和策略中的问题已经在对系统(系统由管理员调整)的大量研究以及算法中有过阐述。

Linux 内核也是一个既庞大又复杂的系统,它仅为相对较少的一部分人所完全理解。它的开发是成千上万具有不同专业、背景以及业余时间的程序员所贡献的结晶。最初的实现是在理论这个重要基础上发展的。贡献者们在这个框架基础上随着现实世界的变化进行开发。

经常有人在 Linux 内存管理邮件列表中因为虚存部分糟糕的文档及理解的困难而说“要遵循这个实现简直是个噩梦”,而实际上缺乏虚存文档的不仅仅是 Linux。Matt Dillon (FreeBSD VM 的主要开发者)就曾经在一次访问中谈到,文档是“很难跟上的”,他说到实现该系统的一个主要困难是要求开发者必须具备内存管理理论的背景知识,只有当开发者具备这些知识时他才会知道为什么要如此实现,而单纯地理解代码只可能做非常细微的优化工作。

本书试图弥补内存管理理论和在 Linux 中具体实现之间的差距,并且将理论和实现联系在一起。它不是对内核的泛泛而谈,而是以相对独立于硬件的角度来阐述 Linux VM。我希望读者在读完这些以及后面的代码注释后,对 VM 系统的运行机制更加得心应手。作为结束部分,图 14.1 大体上阐述了本书所讨论的 VM 子系统如何交互的细节。

作为个人愿望,我希望本书能够推动其他内核子系统中类似文档的产生。我保证我会买的!

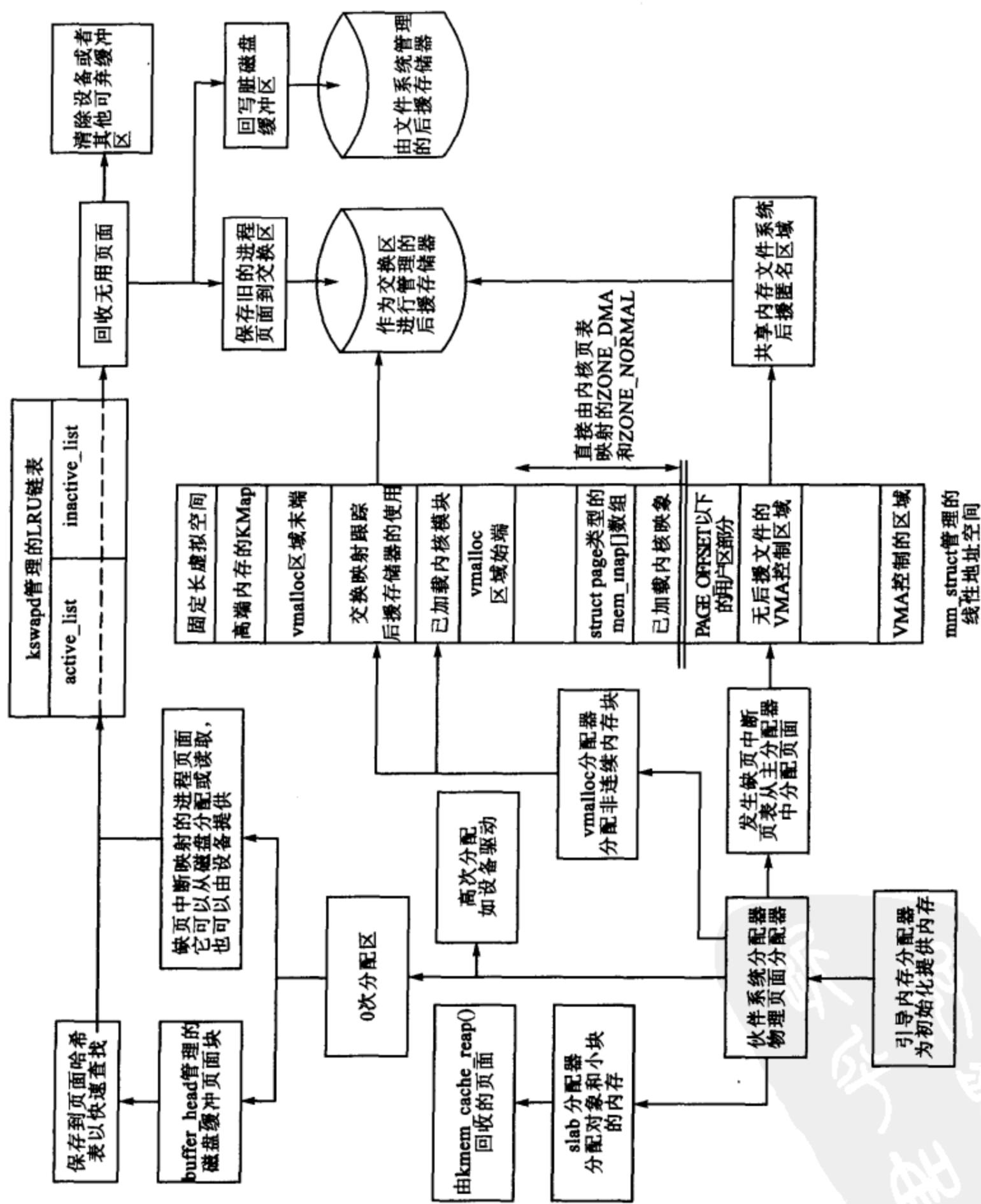


图 14.1 VM 子系统交互的预览图

附录 A

介绍

欢迎来到本书的代码注释部分。如果读到这里,你将找到大量详细的代码。我在代码注释部分假定你已经阅读过本书主体部分的相关章节,所以,如果你是刚开始阅读,可能你到错了地方。

每个附录节的顺序和结构都与本书的顺序和结构相对应。在注释中,函数的显示顺序与调用图中的引用顺序也相同。每个附录和子章节前面都有一个很小的内容列表,帮助指导大家浏览注释。虽然这里的代码注释并没有覆盖 100% 的代码,但是在 VM 中的主要代码样本都可以在这里找到。如果没有注释你感兴趣的代码,则请查看我对类似函数的解释。

为了表述方便,我对一些代码已经做过少许修改,但实际中的代码并没有变。我推荐你在阅读代码注释时使用本书的配套 CD,尤其是使用 LXR 来浏览源代码,这样无论是否存在注释,你都可以获得阅读代码时的良好感觉。

祝你好运!



附录 B

描述物理内存

目 录

| | |
|-----------------------------------|-----|
| B. 1 初始化管理区 | 208 |
| B. 1. 1 函数: setup_memory() | 208 |
| B. 1. 2 函数: zone_sizes_init() | 211 |
| B. 1. 3 函数: free_area_init() | 212 |
| B. 1. 4 函数: free_area_init_node() | 213 |
| B. 1. 5 函数: free_area_init_core() | 214 |
| B. 1. 6 函数: build_zonelists() | 220 |
| B. 2 页面操作 | 234 |
| B. 2. 1 锁住页面 | 234 |
| B. 2. 1. 1 函数: lock_page() | 234 |
| B. 2. 1. 2 函数: __lock_page() | 234 |
| B. 2. 1. 3 函数: sync_page() | 235 |
| B. 2. 2 解锁页面 | 236 |
| B. 2. 2. 1 函数: unlock_page() | 236 |
| B. 2. 3 等待页面 | 236 |
| B. 2. 3. 1 函数: wait_on_page() | 236 |
| B. 2. 3. 2 函数: __wait_on_page() | 237 |

B.1 初始化管理区

B.1.1 函数: setup_memory() (arch/i386/kernel/setup.c)

这个函数的调用图如图 2.3 所示。它为引导内存分配器初始化自身进行所需信息的获取。它可以分成几个不同的任务。

- 找到低端内存的 PFN 的起点和终点(min_low_pfn, max_low_pfn), 找到高端内存的 PFN 的起点和终点(highstart_pfn, highend_pfn), 以及找到系统中最后一页的 PFN。
- 初始化 bootmem_date 结构以及声明可能被引导内存分配器用到的页面。
- 标记所有系统可用的页面为空闲, 然后为那些表示页面的位图保留页面。
- 在 SMP 配置或 initrd 镜像存在时, 为它们保留页面。

```

991 static unsigned long __init setup_memory(void)
992 {
993     unsigned long bootmap_size, start_pfn, max_low_pfn;
994
995     /*
996     * partially used pages are not usable - thus
997     * we are rounding upwards:
998     */
999     start_pfn = PFN_UP(__pa(&_end));
1000
1001     find_max_pfn();
1002
1003     max_low_pfn = find_max_low_pfn();
1004
1005 #ifdef CONFIG_HIGHMEM
1006     highstart_pfn = highend_pfn = max_pfn;
1007     if (max_pfn > max_low_pfn) {
1008         highstart_pfn = max_low_pfn;
1009     }
1010     PRINTK(kern_notice "%ldMB HIGHMEM available.\n",
1011            pages_to_mb(highend_pfn-highstart_pfn));
1012 #endif
1013     printk(KERN_NOTICE "%ldMB LOWMEM available.\n",
1014            pages_to_mb(max_low_pfn));

```

999 PFN_UP()将物理地址向上取整到下一页,返回页帧号。由于_end 是已载入内核镜像的底端地址,所以 start_pfn 现在是可能被用到的第一块物理页面帧的偏移。

1001 find_max_pfn()遍历 e820 图,查找最高的可用 PFN。

1003 find_max_low_pfn()在 ZONE_NORMAL 中找到可寻址的最高页面帧。

1005~1011 如果高端内存可用,则从高端内存区域的 0 位置开始。如果内存 max_low_pfn 后,则把高端内存区的起始位置(highstart_pfn)定在那里,而其结束位置定在 max_pfn,然后打印可用高端内存的提示消息。

1013~1014 打印低端内存量的提示消息。

```

1018 bootmap_size = init_bootmem(start_pfn, max_low_pfn);
1019
1020 register_bootmem_low_pages(max_low_pfn);
1021
1028 reserve_bootmem(HIGH_MEMORY, (PFN_PHYS(start_pfn) +
1029 bootmap_size + PAGE_SIZE-1) - (HIGH_MEMORY));
1030
1035 reserve_bootmem(0, PAGE_SIZE);
1036
1037 # ifdef CONFIG_SMP
1043 reserve_bootmem(PAGE_SIZE, PAGE_SIZE);
1044 # endif
1045 # ifdef CONFIG_ACPI_SLEEP
1046 /*
1047 * Reserve low memory region for sleep support.
1048 */
1049 acpi_reserve_bootmem();
1050 # endif

```

1018 init_bootmem()(见 E.1.1 小节)为 config_page_data 节点初始化 bootmem_data 结构。它设置节点的物理内存起始点和终点,分配一张位图来表示这些页面,并将所有的页面设置为初始时保留。

1020 register_bootmem_low_pages()读入 e820 图,然后为运行时系统中的所有可用页面调用 free_bootmem()(见 E.3.1 小节)。这将标记页面在初始化时为空闲。

1028~1029 为那些表示页面的位图保留页面。

1035 保留 0 号页面,因为 0 号页面是 BIOS 用到的一个特殊页面。

1043 保留额外的页面为跳板代码用。跳板代码处理用户空间如何进入内核空间。

1045~1050 如果加入了睡眠机制,就需要为它保留内存。这仅为那些有挂起功能的手提电脑所用到。它已经超过本书的范围。

```

1051 # ifdef CONFIG_X86_LOCAL_APIC
1052 /*
1053 * Find and reserve possible boot-time SMP configuration;
1054 */
1055 find_smp_config();
1056 #endif
1057 # ifdef CONFIG_BLK_DEV_INITRD
1058 if (LOADER_TYPE && INITRD_START) {
1059 if (INITRD_START + INITRD_SIZE <=
    (max_low_pfn << PAGE_SHIFT)) {
1060 reserve_bootmem(INITRD_START, INITRD_SIZE);
1061 initrd_start =
1062 INITRD_START? INITRD_START + PAGE_OFFSET : 0;
1063 initrd_end = initrd_start + INITRD_SIZE;
1064 }
1065 else {
1066 printk(KERN_ERR
    "initrd extends beyond end of memory "
1067 "(0x%08lx > 0x%08lx)\n disabling initrd\n",
1068 INITRD_START + INITRD_SIZE,
1069 max_low_pfn << PAGE_SHIFT);
1070 initrd_start = 0;
1071 }
1072 }
1073 #endif
1074
1075 return max_low_pfn;
1076 }

```

1055 这个函数为存储关于启动 SMP 的信息而保留内存。

1057~1073 如果 initrd 可用,那么保留存放它的镜像的内存。initrd 提供一个小型文件系统镜像,用于启动系统。

1075 返回 ZONE_NORMAL 中可寻址内存上限。

B.1.2 函数: zone_sizes_init() (arch/i386/mm/init.c)

这是一个用于初始化各管理区的高层函数。PFN 中管理区大小在 setup_memory()过程中发现(见 B.1.1 小节)。这个函数填充一个管理区大小的数组,并把它传给 free_area_init()。

```

323 static void __init zone_sizes_init(void)
324 {
325     unsigned long zones_size[MAX_NR_ZONES] = {0, 0, 0};
326     unsigned int max_dma, high, low;
327
328     max_dma = virt_to_phys((char *)MAX_DMA_ADDRESS) >> PAGE_SHIFT;
329     low = max_low_pfn;
330     high = highend_pfn;
331
332     if (low < max_dma)
333         zones_size[ZONE_DMA] = low;
334     else {
335         zones_size[ZONE_DMA] = max_dma;
336         zones_size[ZONE_NORMAL] = low - max_dma;
337 #ifdef CONFIG_HIGHMEM
338         zones_size[ZONE_HIGHMEM] = high - low;
339 #endif
340     }
341     free_area_init(zones_size);
342 }

```

325 初始化大小为 0。

328 计算最大可能 DMA 寻址的 PFN。它使可能存放于 ZONE_DMA 中最大页面数加倍。

329 max_low_pfn 是 ZONE_NORMAL 可用的最高 PFN。

330 highest_pfn 是 ZONE_HIGHMEM 可用的最高 PFN。

332~333 如果在 ZONE_NORMAL 中的最高 PFN 低于 MAX_DMA_ADDRESS，则仅把 ZONE_DMA 的大小赋值给它，其他管理区仍然为 0。

335 设置 ZONE_DMA 中的页面数。

336 ZONE_NORMAL 的大小等于 max_low_pfn 减去 ZONE_DMA 中的页面数。

338 ZONE_HIGHMEM 的大小是可能的最高 PFN 减去 ZONE_NORMAL 中可能的最高 PFN(max_low_pfn)。

B.1.3 函数: free_area_init() (mm/page_alloc.c)

这是一个与体系结构无关的函数，用于设置 UMA 体系结构。它仅是调用核心函数，传入静态 contig_page_data 作为节点。不同的是，NUMA 体系结构将使用 free_area_node()。

```

838 void __init free_area_init(unsigned long * zones_size)
839 {
840     free_area_init_core(0, &contig_page_data, &mem_map,
841                         zones_size, 0, 0, 0);
841 }

```

838 传递给 free_area_init_core() 的参数如下：

- 0 是该节点的节点标识(NID), 这里为 0。
- contig_page_data 是静态全局 pg_data_t。
- mem_map 用于跟踪 struct page 的全局 mem_map。函数 free_area_init_core() 将为这个数组分配内存。
- zones_sizes 是由 zone_sizes_init() 填充的管理区大小的数组。
- 0, 这个 0 是物理地址的起始点。
- 0, 第 2 个 0 是内存空洞大小的数组, 但不用于 UMA 体系结构。
- 0, 最后一个 0 是一个指向该节点局部 mem_map 的一个指针, 用于 NUMA 体系结构。

B.1.4 函数: free_area_init_node() (mm/numa.c)

这个函数有 2 个版本。第 1 个版本除了使用了一个不同的物理地址起点外, 几乎与 free_area_init() 一样。这个函数还用于仅有一个节点的体系结构(所以它们使用 contig_page_data), 但是它们的物理地址不是 0。

在页表初始化后调用的版本用于初始化系统中的每个 pgdat。如果希望在特定的体系结构中调优它们的位置, 调用者可以有选择地分配它们自己的 mem_map 部分并将它们作为参数传递给这个函数。如果选择不, 则 mem_map 部分就会在后面由 free_area_init_core() 来分配。

```

61 void __init free_area_init_node(int nid,
62                                 pg_data_t * pgdat, struct page * pmap,
63                                 unsigned long * zones_size, unsigned long zone_start_paddr,
64                                 unsigned long * zholes_size)
64 {
65     int i, size = 0;
66     struct page * discard;
67
68     if (mem_map == (mem_map_t *)NULL)
69         mem_map = (mem_map_t *)PAGE_OFFSET;

```

```

70
71 free_area_init_core(nid, pgdat, &discard, zones_size,
72 zone_start_paddr,
73 zholes_size, pmap);
74 pgdat->node_id = nid;
75
75 /*
76 * Get space for the valid bitmap.
77 */
78 for (i = 0; i < MAX_NR_ZONES; i++)
79 size += zones_size[i];
80 size = LONG_ALIGN((size + 7) >> 3);
81 pgdat->valid_addr_bitmap =
82     (unsigned long *)alloc_bootmem_node(pgdat, size);
83 memset(pgdat->valid_addr_bitmap, 0, size);
83 }

```

61 这个函数的参数如下：

- nid 是传入的 pgdat 的 NID。
- pgdat 是将被初始化的节点。
- pmap 是指向将要用到的节点的 mem_map 部分的指针, 它经常传入 NULL, 并在以后分配空间。
- zones_size 是该节点中容量为管理区大小的一个数组。
- zone_start_paddr 是节点的物理地址起点。
- zholes_size 是容量为每个管理区中空洞大小的一个数组。

68~69 如果设置全局的 mem_map, 就将其设置在线性地址空间中内核部分的起点。请记住, 在 NUMA 中, mem_map 是一个分块的虚拟数组, 数组由每个节点中的局部映射填充。

71 调用 free_area_init_core()。注意, discard 作为第 3 个参数传入, 这是因为 NUMA 中并不需要设置全局的 mem_map。

73 记录 pgdat 的 NID。

78~79 计算 NID 的总大小。

80 重新计算位数, 以满足这样大小的每个字节都有一位。

81 分配一张位图, 表示节点中存在的有效管理区。实际上, 这条语句仅用于 Sparc 体系结构, 所以对其他体系结构而言是浪费内存。

82 开始时, 所有的区域都无效。有效区域由后面所述 Sparc 中的 mem_init() 函数标记。其他体系结构仅是忽略掉这张位图。

B.1.5 函数: free_area_init_core() (mm/page_alloc.c)

这个函数负责初始化所有的区域,并在节点中分配它们的局部 lmem_map。在 UMA 体系结构中,调用这个函数将初始化全局 mem_map 数组。在 NUMA 体系结构中,这个数组被看作是一个稀疏分布的虚拟数组。

```

684 void __init free_area_init_core(int nid,
685     pg_data_t * pgdat, struct page ** gmap,
686     unsigned long * zones_size, unsigned long zone_start_paddr,
687     unsigned long * zholes_size, struct page * lmem_map)
688 {
689     unsigned long i, j;
690     unsigned long map_size;
691     unsigned long totalpages, offset, realtotalpages;
692     const unsigned long zone_required_alignment =
693         1UL << (MAX_ORDER-1);
694
695     if (zone_start_paddr & ~PAGE_MASK)
696         BUG();
697
698     totalpages = 0;
699     for (i = 0; i < MAX_NR_ZONES; i++) {
700         unsigned long size = zones_size[i];
701         totalpages += size;
702     }
703     realtotalpages = totalpages;
704     if (zholes_size)
705         for (i = 0; i < MAX_NR_ZONES; i++)
706             realtotalpages -= zholes_size[i];
707
708     printk("On node %d totalpages: %lu\n", nid, realtotalpages);

```

该块主要负责计算每个区域的大小。

691 这个区域必须邻近由伙伴分配器分配的最大大小的块,从而进行位级操作。

693~694 如果物理地址不是按页面排列的,就是一个 BUG()。

696 为这个节点初始化 totalpages 为 0。

697~700 通过遍历 zone_sizes 来计算节点的总大小。

701~704 通过减去 zholes_size 的空洞大小来计算实际的内存量。

706 打印提示信息告知用户这个节点可用的内存量。

```

708 /*
709 * Some architectures (with lots of mem and discontinuous memory
710 * maps) have to search for a good mem_map area:
711 * For discontigmem, the conceptual mem map array starts from
712 * PAGE_OFFSET, we need to align the actual array onto a mem map
713 * boundary, so that MAP_NR works.
714 */
715 map_size = (totalpages + 1) * sizeof(struct page);
716 if (lmem_map == (struct page *)0) {
717 lmem_map = (struct page *) alloc_bootmem_node(pgdat, map_size);
718 lmem_map = (struct page *) (PAGE_OFFSET +
719 MAP_ALIGN((unsigned long)lmem_map - PAGE_OFFSET));
720 }
721 *gmap = pgdat->node_mem_map = lmem_map;
722 pgdat->node_size = totalpages;
723 pgdat->node_start_paddr = zone_start_paddr;
724 pgdat->node_start_mapnr = (lmem_map - mem_map);
725 pgdat->nr_zones = 0;
726
727 offset = lmem_map - mem_map;

```

这一块在需要时分配局部 lmem_map, 设置 gmap 位。在 UMA 体系结构中, gmap 实际上就是 mem_map, 所以这就是为它分配的内存。

715 计算数组所需的内存量。它等于页面总数量乘以 struct page 的大小。

716 如果映射图还没有分配, 就在这里分配。

717 从引导内存分配器中分配一块内存。

718 MAP_ALIGN()将在一个 struct page 大小范围内排列数组, 从而计算在 mem_map 中基于物理地址 MAP_NR()宏的内部偏移。

721 设置 gmap 和 pgdat->node_mem_map 变量以分配 lmem_map。在 UMA 体系结构中, 仅设置 mem_map。

722 记录节点大小。

723 记录起始物理地址。

724 记录节点所占 mem_map 中的偏移。

725 初始化管理区计数为 0, 这将在函数的后面设置。

727 offset 现在是 lmem_map 开始的局部部分中 mem_map 的偏移。

```
728 for (j = 0; j < MAX_NR_ZONES; j++) {
```

```

729 zone_t * zone = pgdat->node_zones + j;
730 unsigned long mask;
731 unsigned long size, realsize;
732
733 zone_table[nid * MAX_NR_ZONES + j] = zone;
734 realsize = size = zones_size[j];
735 if (zholes_size)
736 realsize -= zholes_size[j];
737
738 printk("zone( % lu): % lu pages.\n", j, size);
739 zone->size = size;
740 zone->name = zone_names[j];
741 zone->lock = SPIN_LOCK_UNLOCKED;
742 zone->zone_pgdat = pgdat;
743 zone->free_pages = 0;
744 zone->need_balance = 0;
745 if (! size)
746 continue;

```

从这块管理区开始循环, 初始化节点中的每个 zone_t。该初始化过程开始时设置已存在基本字段的值。

728 遍历节点中所有管理区。

733 记录在 zone_table 中指向该管理区的指针, 见 2.6 节。

734~736 计算管理区的实际大小。它基于 zones_sizes 的总大小减去 zholes_size 的空洞大小。

738 打印提示信息告知在这个管理区中的页面数。

739 记录管理区的大小。

740 zone_names 是管理区的大小, 这里是为了打印的需要。

741~744 初始化管理区的其他字段, 如它的父 pgdat。

745~746 如果管理区没有内存, 就转到下一个管理区, 因为不需要其他操作了。

```

752 zone->wait_table_size = wait_table_size(size);
753 zone->wait_table_shift =
754 BITS_PER_LONG - wait_table_bits(zone->wait_table_size);
755 zone->wait_table = (wait_queue_head_t *)
756 alloc_bootmem_node(pgdat, zone->wait_table_size
757 * sizeof(wait_queue_head_t));
758
759 for(i = 0; i < zone->wait_table_size; ++ i)

```

760 init_waitqueue_head(zone->wait_table + i);

这一块初始化管理区的等待队列。等待该管理区中页面的进程将会使用这个哈希表选择一个队列等待。这意味着在页面解锁时并不需要唤醒所有的等待进程,仅需要唤醒其中的一个子集。

752 wait_table_size() 计算所用哈希表的大小。它基于管理区的页面数和队列数与页面数之间的特定比率完成计算。该哈希表不会大于 4 KB。

753~754 计算哈希算法的因子。

755 分配可以容纳 zone->wait_table_size 项的 wait_queue_head_t 表。

759~760 初始化所有的等待队列。

```

762 pgdat->nr_zones = j + 1;
763
764 mask = (realsize / zone_balance_ratio[j]);
765 if (mask < zone_balance_min[j])
766 mask = zone_balance_min[j];
767 else if (mask > zone_balance_max[j])
768 mask = zone_balance_max[j];
769 zone->pages_min = mask;
770 zone->pages_low = mask * 2;
771 zone->pages_high = mask * 3;
772
773 zone->zone_mem_map = mem_map + offset;
774 zone->zone_start_mapnr = offset;
775 zone->zone_start_paddr = zone_start_paddr;
776
777 if ((zone_start_paddr >> PAGE_SHIFT) &
    (zone_required_alignment-1))
778 printk("BUG: wrong zone alignment, it will crash\n");
779

```

这一块计算管理区极值并记录管理区地址。这个极值记为管理区大小的比率。

762 首先,若激活了一个新的管理区,更新节点中的管理区数量。

764 计算掩码(将用于 page_min 极值)为管理区的大小除以管理区的平衡因子。所有管理区的平衡因子在 mm/page_alloc.c 的首部声明为 128。

765~766 所有管理区的 zone_balance_min 比率都为 20,这意味着 page_min 将不低于 20。

767~768 类似地,所有的 zone_balance_max 都为 255,所以 pages_min 将不会超过 255。

769 pages_min 设为 mask。

770 pages_low 是 pages_min 页面数量的 2 倍。

771 pages_high 是 pages_min 页面数量的 3 倍。

773 记录管理区的第 1 个 struct page 在 mem_map 中的地址。

774 记录 mem_map 中管理区起点的索引。

775 记录起始的物理地址。

777~778 利用伙伴分配器保证管理区已经正确的排列可用。否则,伙伴分配器用到的位级操作就会失败。

```

780 /*
781 * Initially all pages are reserved - free ones are freed
782 * up by free_all_bootmem() once the early boot process is
783 * done. Non-atomic initialization, single-pass.
784 */
785 for (i = 0; i < size; i++) {
786 struct page *page = mem_map + offset + i;
787 set_page_zone(page, nid * MAX_NR_ZONES + j);
788 set_page_count(page, 0);
789 SetPageReserved(page);
790 INIT_LIST_HEAD(&page->list);
791 if (j != ZONE_HIGHMEM)
792 set_page_address(page, __va(zone_start_paddr));
793 zone_start_paddr += PAGE_SIZE;
794 }
795

```

785~794 初始时,管理区中所有的页面都标记为保留,因为没有办法知道引导内存分配器使用的是哪些页面。当引导内存分配器在 free_all_bootmem() 中收回时,未使用页面中的 PG_reserved 会被清除。

786 获取页面的偏移。

787 页面所在的管理区由页面标志编码,见 2.6 节。

788 设置计数为 0,因为管理区未被使用。

789 设置保留标志,然后,若页面不再被使用,引导内存分配器将清除该位。

790 初始化页面链表头。

791~792 如果页面可用且页面在低端内存,设置 page->virtual 字段。

793 将 zone_start_paddr 增加一个页面大小,该参数将用于记录下一个管理区的起点。

```

796 offset += size;
797 for (i = 0; ; i++) {

```

```

798 unsigned long bitmap_size;
799
800 INIT_LIST_HEAD(&zone->free_area[i].free_list);
801 if (i == MAX_ORDER-1) {
802 zone->free_area[i].map = NULL;
803 break;
804 }
805
829 bitmap_size = (size-1) >> (i + 4);
830 bitmap_size = LONG_ALIGN(bitmap_size + 1);
831 zone->free_area[i].map =
832 (unsigned long *) alloc_bootmem_node(pgdat,
                                         bitmap_size);
833 }
834 }
835 build_zonelists(pgdat);
836 }

```

这一块初始化管理区的空闲链表,分配伙伴分配器在记录页面伙伴状态时的位图。

797 从 0 循环到 MAX_ORDER-1。

800 初始化当前次序为 i 的 free_list 链表。

801~804 如果处于最后,设置空闲区域映射为 NULL,表示空闲链表的末端。

829 计算 bitmap_size 为容纳整个位图所需的字节数。位图中每一位表示一个有 2^i 数量页面的伙伴对。

830 利用 LONG_ALIGN()转化容量值为一个长整型,因为所有的位操作都在长整型上进行。

831~832 分配映射用到的内存。

834 循环到下一个管理区。

835 利用 build_zonelists()来构造节点的管理区回退链表。

B. 1.6 函数: build_zonelists() (mm/page_alloc.c)

这个函数在所请求的节点中为每个管理区构造回退管理区。这是为了在不能满足一个分配时可以考察下一个管理区而设立的。在考察结束时,分配将从 ZONE_HIGHMEM 回退到 ZONE_NORMAL,在分配从 ZONE_NORMAL 回退到 ZONE_DMA 后就不再回退。

```

589 static inline void build_zonelists(pg_data_t * pgdat)
590 {

```

```
591 int i, j, k;
592
593 for (i = 0; i <= GFP_ZONEMASK; i++) {
594     zonelist_t *zonelist;
595     zone_t *zone;
596
597     zonelist = pgdat->node_zonelists + i;
598     memset(zonelist, 0, sizeof(*zonelist));
599
600     j = 0;
601     k = ZONE_NORMAL;
602     if (i & __GFP_HIGHMEM)
603         k = ZONE_HIGHMEM;
604     if (i & __GFP_DMA)
605         k = ZONE_DMA;
606
607     switch (k) {
608     default:
609         BUG();
610     /* fallthrough:
611     */
612     }
613     case ZONE_HIGHMEM:
614         zone = pgdat->node_zones + ZONE_HIGHMEM;
615         if (zone->size) {
616             #ifndef CONFIG_HIGHMEM
617             BUG();
618             #endif
619             zonelist->zones[j++] = zone;
620         }
621     case ZONE_NORMAL:
622         zone = pgdat->node_zones + ZONE_NORMAL;
623         if (zone->size)
624             zonelist->zones[j++] = zone;
625     case ZONE_DMA:
626         zone = pgdat->node_zones + ZONE_DMA;
627         if (zone->size)
628             zonelist->zones[j++] = zone;
629     }
```

```

630 zonelist->zones[j ++ ] = NULL;
631 }
632 }

```

593 遍历最大可能数量的管理区。

597 获取管理区的 zonelist 并归 0。

600 与 ZONE_DMA 对应, j 从 0 开始。

601~605 设置 k 为当前检查过的管理区类型。

614 获取 ZONE_HIGHMEM。

615~620 如果管理区有内存, ZONE_HIGHMEM 就是从高端内存分配的相应管理区。如果 ZONE_HIGHMEM 没有内存, ZONE_NORMAL 将在下一次变成相应的管理区, 这是因为 j 并没有在空管理区时加 1。

621~624 设置下一个相应的管理区中分配的为 ZONE_NORMAL, 同样, 如果管理区没有内存就不使用它。

626~628 设置最后的回退管理区为 ZONE_DMA。这个检查也对拥有内存的 ZONE_DMA 进行。在 NUMA 体系结构中, 不是所有的节点都有 ZONE_DMA。

B. 2 页面操作

B. 2. 1 锁住页面

B. 2. 1. 1 函数: `lock_page()` (`mm/filemap.c`)

这个函数试着锁住一个页面。若页面无法被锁住, 就会导致进程睡眠直到页面可用。

```

921 void lock_page(struct page * page)
922 {
923 if (TryLockPage(page))
924 __lock_page(page);
925 }

```

923 TryLockPage() 仅是 `page->flags` 中 `PG_locked` 的 `test_and_set_bit()` 包装器。如果该位以前已经被清除, 函数将立即返回, 因为页面已经被锁住。

924 否则, `__lock_page()` (见 B. 2. 1. 2) 被调用以使进程睡眠。

B. 2. 1. 2 函数: `__lock_page()` (`mm/filemap.c`)

这个函数在 TryLockPage() 失败后调用, 它将进入这个页面的一个等待队列, 并在其中睡

眠直到可以获得锁。

```

897 static void __lock_page(struct page * page)
898 {
899 wait_queue_head_t * waitqueue = page_waitqueue(page);
900 struct task_struct * tsk = current;
901 DECLARE_WAITQUEUE(wait, tsk);
902
903 add_wait_queue_exclusive(waitqueue, &wait);
904 for (;;) {
905 set_task_state(tsk, TASK_UNINTERRUPTIBLE);
906 if (PageLocked(page)) {
907 sync_page(page);
908 schedule();
909 }
910 if (! TryLockPage(page))
911 break;
912 }
913 __set_task_state(tsk, TASK_RUNNING);
914 remove_wait_queue(waitqueue, &wait);
915 }

```

899 page_waitqueue()是哈希算法的实现,决定该页面属于 zone→wait_table 表中的哪个队列。

900~901 为进程初始化等待队列。

903 将进程加入到由 page_waitqueue()返回的等待队列中。

904~912 在这里循环直到获得锁。

905 设置进程状态为不可中断睡眠。在调用 schedule()时,进程将会睡眠,并且不会再醒过来直至队列被显式地唤醒。

906 如果页面仍被锁定,将调用 sync_page()函数来调度页面同步后援存储器。然后调用 schedule()睡眠直到队列被唤醒,就像在页面上 I/O 完成时的情况一样。

910~911 试着再次锁住页面。如果成功,则跳出循环,否则再次在队列中睡眠。

913~914 现在获得锁,所以设置进程状态为 TASK_RUNNING,并从等待队列中移除进程。函数现在返回获得的锁。

B. 2. 1. 3 函数: sync_page() (mm/filemap.c)

调用文件系统特定的 sync_page()来同步页面和它的后援存储器。

```
140 static inline int sync_page(struct page * page)
```

```

141 {
142 struct address_space * mapping = page->mapping;
143
144 if (mapping && mapping->a_ops && mapping->a_ops->sync_page)
145 return mapping->a_ops->sync_page(page);
146 return 0;
147 }

```

142 如果退出则获取页面的 address_spacer。

144~145 如果存在后援,并有相关的 address_space_operations 来提供一个 sync_page() 函数,则在这里调用。

B.2.2 解锁页面

B.2.2.1 函数: unlock_page() (mm/filemap.c)

这个函数解锁页面并唤醒在其上等待的所有进程。

```

874 void unlock_page(struct page * page)
875 {
876 wait_queue_head_t * waitqueue = page_waitqueue(page);
877 ClearPageLander(page);
878 smp_mb__before_clear_bit();
879 if (! test_and_clear_bit(PG_locked, &(page)->flags))
880 BUG();
881 smp_mb__after_clear_bit();
882
883 /*
884 * Although the default semantics of wake_up() are
885 * to wake all, here the specific function is used
886 * to make it even more explicit that a number of
887 * pages are being waited on here.
888 */
889 if (waitqueue_active(waitqueue))
890 wake_up_all(waitqueue);
891 }

```

876 page_waitqueue()是哈希算法的实现,决定该页面属于 zone->wait_table 表中的哪个队列。

877 清除清洗位,因为现在该页面上的 I/O 已经完成。

878 这是一个必须在进行多处理器可见的位操作前必须调用的内存块操作。

879~880 清除 PG_locked 位。如果位已经被清除就是一个 BUG()。

881 完成 SMP 内存块操作。

889~890 如果有进程在该页面的等待队列中等待，则唤醒它们。

B.2.3 在页面上等待

B.2.3.1 函数：wait_on_page() (include/linux/pagemap.h)

```
94 static inline void wait_on_page(struct page * page)
95 {
96 if (PageLocked(page))
97 __wait_on_page(page);
98 }
```

96~97 如果页面正被锁住，将调用 __wait_on_page() 睡眠直到被解锁。

B.2.3.2 函数：__wait_on_page() (mm/filemap.c)

在用 PageLocked() 决定哪个页面被锁住后调用这个函数。调用的进程可能睡眠直到 page 被解锁。

```
849 void __wait_on_page(struct page * page)
850 {
851 wait_queue_head_t * waitqueue = page_waitqueue(page);
852 struct task_struct * tsk = current;
853 DECLARE_WAITQUEUE(wait, tsk);
854
855 add_wait_queue(waitqueue, &wait);
856 do {
857 set_task_state(tsk, TASK_UNINTERRUPTIBLE);
858 if (!PageLocked(page))
859 break;
860 sync_page(page);
861 schedule();
862 } while (PageLocked(page));
863 __set_task_state(tsk, TASK_RUNNING);
864 remove_wait_queue(waitqueue, &wait);
865 }
```

851 `page_waitqueue()` 是哈希算法的实现, 决定该页面属于 `zone->wait_table` 表中的哪个队列。

852~853 为当前进程初始化等待队列。

855 将进程添加到由 `page_waitqueue()` 返回的等待队列中。

857 将进程状态设置为不可中断睡眠。在调用 `schedule()` 后, 进程将睡眠。

858~859 检查确定页面自上次检查后没有被解锁。

860 调用 `sync_page`(见 B. 2. 1. 3) 来调用文件系统特定的函数来同步页面和它的后援存储器。

861 调用 `schedule()` 睡眠。在页面解锁时进程将被唤醒。

862 检查页面是否仍然被锁定。请记住多个页面可能使用这个等待队列, 而且还可能有处于睡眠的进程想锁住这个页面。

863~864 页面已经被解锁了。设置进程为 `TASK_RUNNING` 状态, 并将其从等待队列中移除。



附录 C

页表管理

目 录

| | |
|------------------------------|-----|
| C. 1 初始化页表 | 229 |
| C. 1. 1 函数: paging_init() | 229 |
| C. 1. 2 函数: pagetable_init() | 230 |
| C. 1. 3 函数: fixrange_init() | 234 |
| C. 1. 4 函数: kmap_init() | 236 |
| C. 2 遍历页表 | 237 |
| C. 2. 1 函数: follow_page() | 237 |

C. 1 初始化页表

C. 1. 1 函数: paging_init() (arch/i386/mm/init.c)

这是 setup_arch() 调用的高层函数。当这个函数返回时, 页面已经完全建立完成。注意这里都是与 x86 相关的。

```
351 void __init paging_init(void)
352 {
353     pagetable_init();
354
355     load_cr3(swapper_pg_dir);
```

```

356
357 # if CONFIG_X86_PAE
358 if (cpu_has_pae)
359     set_in_cr4(X86_CR4_PAE);
360 #endif
361
362 __flush_tlb_all();
363
364 #endif
365
366 #ifdef CONFIG_HIGHMEM
367     kmap_init();
368 #endif
369     zone_sizes_init();
370 }

```

353 pagetable_init() 负责利用 swapper_pg_dir 设立一个静态页表作为 PGD。

355 将初始化后的 swapper_pg_dir 载入 CR3 寄存器, 这样 CPU 可以使用它。

362~363 如果 PAE 可用, 则在 CR4 寄存器中设置相应的位。

366 清洗所有(包括在全局内核中)的 TLB。

369 kmap_init() 调用 kmap() 初始化保留的页表区域。

371 zone_sizes_init()(见 B. 1. 2 小节)记录每个管理区的大小, 然后调用 free_area_init()(见 B. 1. 3 小节)来初始化各个管理区。

C. 1. 2 函数: pagetable_init() (arch/i386/mm/init.c)

这个函数负责静态初始化一个从静态定义的称为 swapper_pg_dir 的 PGD 开始的页表。不管怎样, PTE 将可以指向在 ZONE_NORMAL 中的每个页面帧。

```

205 static void __init pagetable_init (void)
206 {
207     unsigned long vaddr, end;
208     pgd_t * pgd, * pgd_base;
209     int i, j, k;
210     pmd_t * pmd;
211     pte_t * pte, * pte_base;
212
213     /*
214     * This can be zero as well - no problem, in that case we exit
215     * the loops anyway due to the PTRS_PER_* conditions.
216     */

```

```

217 end = (unsigned long) __va(max_low_pfn * PAGE_SIZE);
218
219 pgd_base = swapper_pg_dir;
220 #if CONFIG_X86_PAE
221 for (i = 0; i < PTRS_PER_PGD; i++)
222     set_pgd(pgd_base + i, __pgd(1 + __pa(empty_zero_page)));
223 #endif
224 i = __pgd_offset(PAGE_OFFSET);
225 pgd = pgd_base + i;

```

这里初始化 PGD 的第一块。它把每个表项指向全局 0 页面。需要引用的在 ZONE_NORMAL 中可用内存的表项将在后面分配。

217 变量 end 标志在 ZONE_NORMAL 中物理内存的末端。

219 pgd_base 设立指向静态声明的 PGD 起始位置。

220~223 如果 PAE 可用,仅将每个表项设为 0(实际上,将每个表项指向全局 0 页面)就不够了,因为每个 pgd_t 是一个结构。所以,set_pgd 必须在每个 pgd_t 上调用以使每个表项都指向全局 0 页面。

224 i 初始化为 PGD 中的偏移,与 PAGE_OFFSET 相对应。或者说,这个函数将仅初始化线性地址空间的内核部分。可以不用关心这个用户空间部分。

225 pgd 初始化为 pgd_t,对应于线性地址空间中内核部分的起点。

```

227 for ( ; i < PTRS_PER_PGD; pgd++, i++) {
228     vaddr = i * PGDIR_SIZE;
229     if (end && (vaddr >= end))
230         break;
231 #if CONFIG_X86_PAE
232     pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
233     set_pgd(pgd, __pgd(__pa(pmd) + 0x1));
234 #else
235     pmd = (pmd_t *) pgd;
236 #endif
237     if (pmd != pmd_offset(pgd, 0))
238         BUG();

```

这个循环开始指向有效 PMD 表项。在 PAE 的情形下,页面由 alloc_bootmem_low_pages() 分配,然后设立相应的 PGD。没有 PAG 时,就没有中间目录,所以就折返到 PGD 以保留三层页表的假象。

227 i 已经初始化为线性地址空间的内核部分起始位置,所以这里一直循环到最后 PTRS_PER_PGD 处的 pgd_t。

228 计算这个 PGD 的虚拟地址。

229~230 如果到达 ZONE_NORMAL 的末端, 则跳出循环, 因为不再需要另外的页表项。

231~234 如果 PAE 可用, 则为 PMD 分配一个页面, 并利用 set_pgd() 将页面插入到页表中。

235 如果 PAE 不可用, 仅设置 pmd 指向当前 pgd_t。这就是模拟三层页表的“折返”策略。

237~238 这是一个有效性检查, 以保证 PMD 有效。

```

239 for (j = 0; j < PTRS_PER_PMD; pmd++, j++) {
240     vaddr = i * PGDIR_SIZE + j * PMD_SIZE;
241     if (end && (vaddr >= end))
242         break;
243     if (cpu_has_pse) {
244         unsigned long __pe;
245
246         set_in_cr4(X86_CR4_PSE);
247         boot_cpu_data.wp_works_ok = 1;
248         __pe = _KERNPG_TABLE + _PAGE_PSE + __pa(vaddr);
249         /* Make it "global" too if supported */
250         if (cpu_has_pge) {
251             set_in_cr4(X86_CR4_PGE);
252             __pe += _PAGE_GLOBAL;
253         }
254         set_pmd(pmd, __pmd(__pe));
255     continue;
256     }
257
258     pte_base = pte =
259         (pte_t *) alloc_bootmem_low_pages(PAGE_SIZE);

```

这一块初始化 PMD 中的每个表项。这个循环仅在 PAE 可用时进行。请记住, 没有 PAE 时 PTRS_PER_PMD 是 1。

240 计算这个 PMD 的虚拟地址。

241~242 如果到达 ZONE_NORMAL 的末端, 就完成一次循环。

243~248 如果 CPU 支持 PSE, 使用大 TLB 表项。这意味着, 对内核页面而言, 一个 TLB 项将映射 4 MB 而不是平常的 4 KB, 将不再需要三层 PTE。

248 `__pe` 设置为内核页表的标志(`_KERNPG_TABLE`), 以及表明这是一个映射 4 MB(`_PAGE_PSE`)的标志, 然后利用`__pa()`表明这块虚拟地址的物理地址。这意味着 4 MB 的物理地址不由页表映射。

250~253 如果 CPU 支持 PGE, 则为页表项设置它。它标记表项为全局的, 并为所有进程可见。

254~255 由于 PSE 的缘故, 所以不需要三层页表。现在利用`set_pmd()`来设置 PMD, 并继续到下一个 PMD。

258 相反, 如果 PSE 不被支持, 需要 PTE 的时候, 为它们分配一个页面。

```

260 for (k = 0; k < PTRS_PER_PTE; pte++, k++) {
261     vaddr = i * PGDIR_SIZE + j * PMD_SIZE + k * PAGE_SIZE;
262     if (end && (vaddr >= end))
263         break;
264     *pte = mk_pte_phys(__pa(vaddr), PAGE_KERNEL);
265 }
266 set_pmd(pmd, __pmd(_KERNPG_TABLE + __pa(pte_base)));
267 if (pte_base != pte_offset(pmd, 0))
268     BUG();
269
270 }
271 }
```

这一块初始化 PTE。

260~265 对每个 `pte_t`, 计算当前被检查的虚拟地址, 创建一个 PTE 来指向相应的物理页面帧。

266 PTE 已经被初始化, 所以设置 PMD 来指向包含它们的页面。

267~268 保证表项已经正确建立。

```

273 /*
274 * Fixed mappings, only the page table structure has to be
275 * created - mappings will be set by set_fixmap():
276 */
277 vaddr = __fix_to_virt(__end_of_fixed_addresses - 1) & PMD_MASK;
278 fixrange_init(vaddr, 0, pgd_base);
279
280 #if CONFIG_HIGHMEM
281 /*
282 * Permanent kmaps:
283 */
```



```

284 vaddr = PKMAP_BASE;
285 fixrange_init(vaddr, vaddr + PAGE_SIZE * LAST_PKMAP, pgd_base);
286
287 pgd = swapper_pg_dir + __pgd_offset(vaddr);
288 pmd = pmd_offset(pgd, vaddr);
289 pte = pte_offset(pmd, vaddr);
290 pkmap_page_table = pte;
291 #endif
292
293 #if CONFIG_X86_PAE
294 /*
295 * Add low memory identity-mappings - SMP needs it when
296 * starting up on an AP from real-mode. In the non-PAE
297 * case we already have these mappings through head.S.
298 * All user-space mappings are explicitly cleared after
299 * SMP startup.
300 */
301 pgd_base[0] = pgd_base[USER_PTRS_PER_PGD];
302 #endif
303 }

```

在这点上,已经设立页面表项来引用 ZONE_NORMAL 的所有部分。需要的其他区域是那些固定映射以及那些需要利用 kmap() 映射高端内存的区域。

277 固定地址空间被认为从 FIXADDR_TOP 开始,并在地址空间前面结束。`__fix_to_virt()` 将一个下标作为参数,并返回在固定虚拟地址空间中的第 index 个下标后续页面帧(从 FIXADDR_TOP 开始)。`__end_of_fixed_addresses` 是上一个由固定虚拟地址空间用到的下标。或者说,这一行返回的应该是固定虚拟地址空间起始位置的 PMD 虚拟地址。

278 这个函数传递 0 作为 `fixrange_init()` 的结束,它开始于 `vaddr`,并创建有效 PGD 和 PMD 直到虚拟地址空间的末端,对这些地址而言不需要 PTE。

280~291 利用 kmap() 来设立页表。

287~290 利用 kmap() 获取对应于待用到的区域首址的 PTE。

301 这里设立一个临时标记来映射虚拟地址 0 和物理地址 0。

C.1.3 函数: `fixrange_init()` (`arch/i386/mm/init.c`)

这个函数为固定虚拟地址映射创建有效的 PGD 和 PMD。

```
167 static void __init fixrange_init (unsigned long start,
```

```

        unsigned long end,
        pgd_t * pgd_base)

168 {
169 pgd_t * pgd;
170 pmd_t * pmd;
171 pte_t * pte;
172 int i, j;
173 unsigned long vaddr;
174
175 vaddr = start;
176 i = __pgd_offset(vaddr);
177 j = __pmd_offset(vaddr);
178 pgd = pgd_base + i;
179
180 for ( ; (i < PTRS_PER_PGD) && (vaddr != end); pgd++, i++) {
181 #if CONFIG_X86_PAE
182 if (pgd_none(*pgd)) {
183 pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
184 set_pgd(pgd, __pgd(__pa(pmd) + 0x1));
185 if (pmd != pmd_offset(pgd, 0))
186 printk("PAE BUG #02! \n");
187 }
188 pmd = pmd_offset(pgd, vaddr);
189 #else
190 pmd = (pmd_t *) pgd;
191 #endif
192 for ( ; (j < PTRS_PER_PMD) && (vaddr != end); pmd++, j++) {
193 if (pmd_none(*pmd)) {
194 pte = (pte_t *) alloc_bootmem_low_pages(PAGE_SIZE);
195 set_pmd(pmd, __pmd(__KERNPG_TABLE + __pa(pte)));
196 if (pte != pte_offset(pmd, 0))
197 BUG();
198 }
199 vaddr += PMD_SIZE;
200 }
201 j = 0;
202 }
203 }

```

175 设置虚拟地址(vaddr)作为请求起始地址的一个参数。

- 176 获取对应于 vaddr 的 PGD 内部索引。
- 177 获取对应于 vaddr 的 PMD 内部索引。
- 178 获取 pgd_t 的起点。
- 180 一直循环直到到达 end。当 pagetable_init() 传入 0 时, 将继续循环直到 PGD 的末端。
- 182~187 在有 PAE 时, 若没有为 PMD 分配页面, 这里就为 PMD 分配一个页面。
- 190 没有 PAE 时, 也没有 PMD, 所以这里把 pgd_t 看作 pmd_t。
- 192~200 对 PMD 中的每个表项, 这里将为 pte_t 表项分配一个页面, 并在页表中设置。注意 vaddr 是以 PMD 大小作为步增加的。

C.1.4 函数: kmap_init() (arch/i386/mm/init.c)

这个函数仅存在于如果在编译时设置了 CONFIG_HIGHMEM 的情况下。它负责获取 kmap 区域的首址, 引用它的 PTE 以及保护页表。这意味着在使用 kmap() 时不一定都需要检查 PGD。

```

74 # if CONFIG_HIGHMEM
75 pte_t * kmap_pte;
76 pgprot_t kmap_prot;
77
78 # define kmap_get_fixmap_pte(vaddr) \
79 pte_offset(pmd_offset(pgd_offset_k(vaddr)), (vaddr)), (vaddr))
80
81 void __init kmap_init(void)
82 {
83     unsigned long kmap_vstart;
84
85     /* cache the first kmap pte */
86     kmap_vstart = __fix_to_virt(FIX_KMAP_BEGIN);
87     kmap_pte = kmap_get_fixmap_pte(kmap_vstart);
88
89     kmap_prot = PAGE_KERNEL;
90 }
91 #endif /* CONFIG_HIGHMEM */

```

78~79 由于 fixrange_init() 已经设立了有效的 PGD 和 PMD, 所以就不需要再一次检查它们, 这样 kmap_get_fixmap_pte() 可以快速遍历页表。

86 缓存 kmap_vstart 中 kmap 区域的虚拟地址。

87 缓存 PTE 作为 kmap_pte 中 kmap 区域的首址。

89 利用 kmap_prot 缓存页表表项的保护项。

C. 2 遍历页表

C. 2. 1 函数: follow_page() (mm/memory.c)

这个函数返回在 mm 页表中 address 处 PTE 用到的 struct page。

```

405 static struct page * follow_page(struct mm_struct * mm,
406                                     unsigned long address,
407                                     int write)
408
409 pte_t * ptep, pte;
410
411 pgd = pgd_offset(mm, address);
412 if (pgd_none(* pgd) || pgd_bad(* pgd))
413     goto out;
414
415 pmd = pmd_offset(pgd, address);
416 if (pmd_none(* pmd) || pmd_bad(* pmd))
417     goto out;
418
419 ptep = pte_offset(pmd, address);
420 if (! ptep)
421     goto out;
422
423 pte = * ptep;
424 if (pte_present(pte)) {
425     if (! write ||
426         (pte_write(pte) && pte_dirty(pte)))
427         return pte_page(pte);
428 }
429
430 out:
431 return 0;

```

432 }

405 这个函数是 mm 中待遍历的页表。address 与 struct page 和 write 有关, 它表明该页是否将被写。

411 获取 address 处的 PGD 并保证它存在且有效。

415~417 获取 address 处的 PMD 并保证它存在且有效。

419 获取 address 处的 PTE 并保证它存在且有效。

424 如果 PTE 当前存在, 则可以返回一些东西。

425~426 如果调用者指定了将发生写操作, 则检查以保证 PTE 有写权限。如果是这样, 将 PTE 设为脏。

427 如果 PTE 存在并有权限, 则返回由 PTE 映射的 struct page。

431 返回 0 表明该 address 没有相应的 struct page。



附录 D

进程地址空间

目 录

| | |
|------------------------------------|-----|
| D. 1 进程内存描述符 | 243 |
| D. 1. 1 初始化一个描述符 | 243 |
| D. 1. 2 复制一个描述符 | 244 |
| D. 1. 2. 1 函数:copy_mm() | 244 |
| D. 1. 2. 2 函数:mm_init() | 246 |
| D. 1. 3 分配一个描述符 | 247 |
| D. 1. 3. 1 函数:allocate_mm() | 247 |
| D. 1. 3. 2 函数:mm_alloc() | 258 |
| D. 1. 4 销毁一个描述符 | 247 |
| D. 1. 4. 1 函数:mmpput() | 248 |
| D. 1. 4. 2 函数:mmdrop() | 249 |
| D. 1. 4. 3 函数:__mmdrop() | 250 |
| D. 2 创建内存区域 | 251 |
| D. 2. 1 创建一个内存区域 | 251 |
| D. 2. 1. 1 函数:do_mmap() | 251 |
| D. 2. 1. 2 函数:do_mmap_pgoff() | 251 |
| D. 2. 2 插入一个内存区域 | 260 |
| D. 2. 2. 1 函数:__insert_vm_struct() | 260 |
| D. 2. 2. 2 函数:find_vma_prepare() | 261 |
| D. 2. 2. 3 函数:vma_link() | 262 |

| | |
|---------------------------------------|-----|
| D. 2. 2. 4 函数:__vma_link() | 263 |
| D. 2. 2. 5 函数:__vma_link_list() | 263 |
| D. 2. 2. 6 函数:__vma_link_rb() | 264 |
| D. 2. 2. 7 函数:__vma_link_file() | 264 |
| D. 2. 3 合并相邻区域 | 265 |
| D. 2. 3. 1 函数:vma_merge() | 265 |
| D. 2. 3. 2 函数:can_vma_merget() | 267 |
| D. 2. 4 重映射并移动一个内存区域 | 268 |
| D. 2. 4. 1 函数:sys_mremap() | 268 |
| D. 2. 4. 2 函数:do_mremap() | 268 |
| D. 2. 4. 3 函数:move_vma() | 284 |
| D. 2. 4. 4 函数:make_pages_present() | 288 |
| D. 2. 4. 5 函数:get_user_pages() | 289 |
| D. 2. 4. 6 函数:move_page_tables() | 292 |
| D. 2. 4. 7 函数:move_one_page() | 293 |
| D. 2. 4. 8 函数:get_one_pte() | 294 |
| D. 2. 4. 9 函数:alloc_one_pte() | 295 |
| D. 2. 4. 10 函数:copy_one_pte() | 295 |
| D. 2. 5 删除一个内存区域 | 296 |
| D. 2. 5. 1 函数:do_munmap() | 296 |
| D. 2. 5. 2 函数:unmap_fixup() | 300 |
| D. 2. 6 删除所有的内存区域 | 304 |
| D. 2. 6. 1 函数:exit_mmap() | 304 |
| D. 2. 6. 2 函数:clear_page_tables() | 306 |
| D. 2. 6. 3 函数:free_one_pgd() | 306 |
| D. 2. 6. 4 函数:free_one_pmd() | 307 |
| D. 3 查询内存区域 | 309 |
| D. 3. 1 找到一个映射内存区域 | 309 |
| D. 3. 1. 1 函数:find_vma() | 309 |
| D. 3. 1. 2 函数:find_vma_prev() | 310 |
| D. 3. 1. 3 函数:find_vma_intersection() | 312 |
| D. 3. 2 找到一个空袭那内存区域 | 312 |
| D. 3. 2. 1 函数:get_unmapped_area() | 312 |

| | |
|--|-----|
| D. 3. 2. 2 函数:arch_get_unmapped_area() | 313 |
| D. 4 对内存区域上锁和解锁 | 315 |
| D. 4. 1 对内存区域上锁 | 315 |
| D. 4. 1. 1 函数:sys_mlock() | 315 |
| D. 4. 1. 2 函数:sys_mlockall() | 316 |
| D. 4. 1. 3 函数:do_mlockall() | 318 |
| D. 4. 1. 4 函数:do_mlock() | 319 |
| D. 4. 2 对区域解锁 | 321 |
| D. 4. 2. 1 函数:sys_munlock() | 321 |
| D. 4. 2. 2 函数:sys_munlockall() | 321 |
| D. 4. 3 上锁/解锁后修整区域 | 322 |
| D. 4. 3. 1 函数:mlock_fixup() | 322 |
| D. 4. 3. 2 函数:mlock_fixup_all() | 323 |
| D. 4. 3. 3 函数:mlock_fixup_start() | 324 |
| D. 4. 3. 4 函数:mlock_fixup_end() | 325 |
| D. 4. 3. 5 函数:mlock_fixup_middle() | 326 |
| D. 5 缺页中断 | 328 |
| D. 5. 1 x86 缺页中断处理程序 | 328 |
| D. 5. 1. 1 函数:do_page_fault() | 328 |
| D. 5. 2 扩展栈 | 337 |
| D. 5. 2. 1 函数:expand_stack() | 337 |
| D. 5. 3 独立体系结构的页面中断处理程序 | 338 |
| D. 5. 3. 1 函数:handle_mm_fault() | 339 |
| D. 5. 3. 2 函数:handle_pte_fault() | 340 |
| D. 5. 4 请求分配 | 341 |
| D. 5. 4. 1 函数:do_no_page() | 341 |
| D. 5. 4. 2 函数:do_anonymous_page() | 344 |
| D. 5. 5 请求分页 | 346 |
| D. 5. 5. 1 函数:do_swap_page() | 346 |
| D. 5. 5. 2 函数:can_share_swap_page() | 350 |
| D. 5. 5. 3 函数:exclusive_swap_page() | 351 |
| D. 5. 6 写时复制(COW)页面 | 352 |
| D. 5. 6. 1 函数:do_wp_page() | 352 |

| | |
|---|-----|
| D. 6 页面相关的磁盘 I/O | 355 |
| D. 6. 1 一般文件读 | 355 |
| D. 6. 1. 1 函数:generic_file_read() | 355 |
| D. 6. 1. 2 函数:do_generic_file_read() | 358 |
| D. 6. 1. 3 函数:generic_file_readahead() | 365 |
| D. 6. 2 一般文件 mmap() | 369 |
| D. 6. 2. 1 函数:generic_file_mmap() | 369 |
| D. 6. 3 一般文件截断 | 370 |
| D. 6. 3. 1 函数:vmtruncate() | 370 |
| D. 6. 3. 2 函数:vmtruncate_list() | 372 |
| D. 6. 3. 3 函数:zap_page_range() | 373 |
| D. 6. 3. 4 函数:zap_pmd_range() | 375 |
| D. 6. 3. 5 函数:zap_pte_range() | 376 |
| D. 6. 3. 6 函数:truncate_inode_pages() | 377 |
| D. 6. 3. 7 函数:truncate_list_pages() | 378 |
| D. 6. 3. 8 函数:truncate_complete_page() | 380 |
| D. 6. 3. 9 函数:do_flushpage() | 381 |
| D. 6. 3. 10 函数:truncate_partial_page() | 381 |
| D. 6. 4 为页面高速缓存读入页面 | 382 |
| D. 6. 4. 1 函数:filemap_nopage() | 382 |
| D. 6. 4. 2 函数:page_cache_read() | 387 |
| D. 6. 5 为 nopage() 预读文件 | 388 |
| D. 6. 5. 1 函数:nopage_sequential_readahead() | 388 |
| D. 6. 5. 2 函数:read_cluster_nonblocking() | 390 |
| D. 6. 6 缓冲区相关的预读 | 391 |
| D. 6. 6. 1 函数:swabin_readahead() | 391 |
| D. 6. 6. 2 函数:valid_swaphandles() | 392 |

D. 1 进程内存描述符

这部分涵盖用于分配、初始化、复制和销毁内容描述符的函数。

D.1.1 初始化一个描述符

系统中的 mm_struct 开始称为 init_mm, 它在编译时由宏 INIT_MM() 静态初始化。

```
238 #define INIT_MM(name) \
239 { \
240     mm_rb: RB_ROOT, \
241     pgd: swapper_pg_dir, \
242     mm_users: ATOMIC_INIT(2), \
243     mm_count: ATOMIC_INIT(1), \
244     mmap_sem: __RWSEM_INITIALIZER(name.mmap_sem), \
245     page_table_lock: SPIN_LOCK_UNLOCKED, \
246     mmlist: LIST_HEAD_INIT(name.mmlist), \
247 }
```

新 mm_struct 在建立以后, 是它们父 mm_struct 的备份, 并且它们由 copy_mm() 以 init_mm() 初始化的字段来复制。

D.1.2 复制一个描述符

D.1.2.1 函数: copy_mm() (kernel/fork.c)

这个函数为给定的进程复制一份 mm_struct。它仅在创建一个新进程后且需要它自己的 mm_struct 时由 do_fork() 调用。

```
315 static int copy_mm(unsigned long clone_flags,
316                      struct task_struct * tsk)
317 {
318     struct mm_struct * mm, * oldmm;
319     int retval;
320
321     tsk->min_flt = tsk->maj_flt = 0;
322     tsk->cmin_flt = tsk->cmaj_flt = 0;
323     tsk->nswap = tsk->cnswap = 0;
324
325     tsk->mm = NULL;
326     tsk->active_mm = NULL;
327     /*
```



```

328 * Are we cloning a kernel thread?
329 * We need to steal an active VM for that..
330 */
331
332 oldmm = current->mm;
333 if (!oldmm)
334 return 0;
335
336 if (clone_flags & CLONE_VM) {
337 atomic_inc(&oldmm->mm_users);
338 mm = oldmm;
339 goto good_mm;
340 }

```

这一块重置没有被子 mm_struct 继承的字段并找到一个复制源 mm 的字段。

315 这些参数是为克隆而传入的标志位和那些复制 mm_struct 的进程。

320~325 初始化与内存管理相关的 task_struct 字段。

332 借用当前运行进程的 mm 来复制。

333 一个没有 mm 的内核线程,所以它可以立即返回。

336~341 如果设置了 CLONE_VM 标志位,子进程将与父进程共享 mm。像 pthreads 这样的用户需要这样做。mm_users 字段加 1,以使 mm 不会过早销毁。good_mm 标记设置 tsk->mm 和 tsk->active_mm,并返回成功。

```

342 retval = -ENOMEM;
343 mm = allocate_mm();
344 if (!mm)
345 goto fail_nomem;
346
347 /* Copy the current MM stuff.. */
348 memcpy(mm, oldmm, sizeof(*mm));
349 if (!mm_init(mm))
350 goto fail_nomem;
351
352 if (init_new_context(tsk,mm))
353 goto free_pt;
354
355 down_write(&oldmm->mmap_sem);
356 retval = dup_mmap(mm);
357 up_write(&oldmm->mmap_sem);
358

```

343 分配新的 mm。

348~350 复制父 mm，并利用 init_mm() 来初始化特定进程的 mm 字段。

352~353 为那些无法自动管理其 MMU 的体系结构初始化 MMU 上下文。

355~357 调用 dup_mmap()，它负责复制所有 VMA 父进程用到的区域。

```

359 if (retval)
360 goto free_pt;
361
362 /*
363 * child gets a private LDT (if there was an LDT in the parent)
364 */
365 copy_segments(tsk, mm);
366
367 good_mm:
368 tsk->mm = mm;
369 tsk->active_mm = mm;
370 return 0;
371
372 free_pt:
373 mmput(mm);
374 fail_nomem:
375 return retval;
376 }
```

359 一旦成功，dup_mmap() 返回 0。如果失败，标记 free_pt 将调用 mmput()，它将 mm 的使用计数减 1。

365 基于父进程为新进程复制 LDT。

368~370 设置新 mm, active_mm 并返回成功。

D. 1. 2. 2 函数: mm_init() (kernel/fork.c)

这个函数初始化特定进程的 mm 字段。

```

230 static struct mm_struct * mm_init(struct mm_struct * mm)
231 {
232 atomic_set(&mm->mm_users, 1);
233 atomic_set(&mm->mm_count, 1);
234 init_rwsem(&mm->mmap_sem);
235 mm->page_table_lock = SPIN_LOCK_UNLOCKED;
236 mm->pgd = pgd_alloc(mm);
237 mm->def_flags = 0;
```



```

238 if (mm->pgd)
239 return mm;
240 free_mm(mm);
241 return NULL;
242 }

232 设置用户数为 1。
233 设置 mm 的引用计数为 1。
234 初始化保护 VMA 链表的信号量。
235 初始化保护写访问的自旋锁。
236 为该结构分配新的 PGD。
237 缺省时,进程所用的页面未在内存中上锁。
238 如果 PGD 存在,这里将返回初始化后的结构。
240 如果初始化失败,这里删除 mm_struct 并返回。

```

D.1.3 分配一个描述符

提供了两个函数来分配一个 mm_struct。虽然有点容易混淆,但它们实际上都是一样的。allocate_mm()将从 slab 分配器中分配一个 mm_struct。mm_alloc()将分配结构,然后调用 mm_init()来初始化。

D.1.3.1 函数:allocate_mm() (kernel/fork.c)

```

227 #define allocate_mm() (kmem_cache_alloc(mm_cachep, SLAB_KERNEL))

227 从 slab 分配器分配一个 mm_struct。

```

D.1.3.2 函数:mm_alloc() (kernel/fork.c)

```

248 struct mm_struct * mm_alloc(void)
249 {
250 struct mm_struct * mm;
251
252 mm = allocate_mm();
253 if (mm) {
254 memset(mm, 0, sizeof(*mm));
255 return mm_init(mm);
256 }
257 return NULL;

```



258 }
 252 从 slab 分配器分配一个 mm_struct。
 254 将结构的所有字段归 0。
 255 进行基本的初始化。

D.1.4 销毁一个描述符

mm 的一个新用户利用如下调用将使用计数加 1:

atomic_inc(&mm->mm_users);

利用 mmput() 将使用计数减 1。如果 mm_usrs 计数减到 0, 将调用 exit_mmap() 将所有的已映射区域删除, 而且所有的页表也将会销毁, 因为已经没有任何使用用户空间部分的使用者。mm_count 计数由 mmdrop() 减 1, 因为页表和 VMA 的使用者都记为一个 mm_struct 使用者。在 mm_count 减到 0 时, mm_struct 将会被销毁。

D.1.4.1 函数: mmput() (kernel/fork.c)

```
276 void mmput(struct mm_struct * mm)
277 {
278 if (atomic_dec_and_lock(&mm->mm_users, &mmlist_lock)) {
279 extern struct mm_struct * swap_mm;
280 if (swap_mm == mm)
281 swap_mm = list_entry(mm->mmlist.next,
282 struct mm_struct, mmlist);
283 list_del(&mm->mmlist);
284 mmlist_nr--;
285 spin_unlock(&mmlist_lock);
286 exit_mmap(mm);
287 }
288 }
```

278 在获取 mmlist_lock 锁时, 原子性地将 mm_users 字段减 1。若计数减到 0 则返回该锁。

279~286 如果使用计数减到 0, 需要将 mm 和相关的结构移除。

279~281 swap_mm 时由 vmscan 代码换出最后的 mm。如果当前进程是最后换出的 mm, 这里将移到链表的下一个表项。

282 从链表中移除这个 mm。

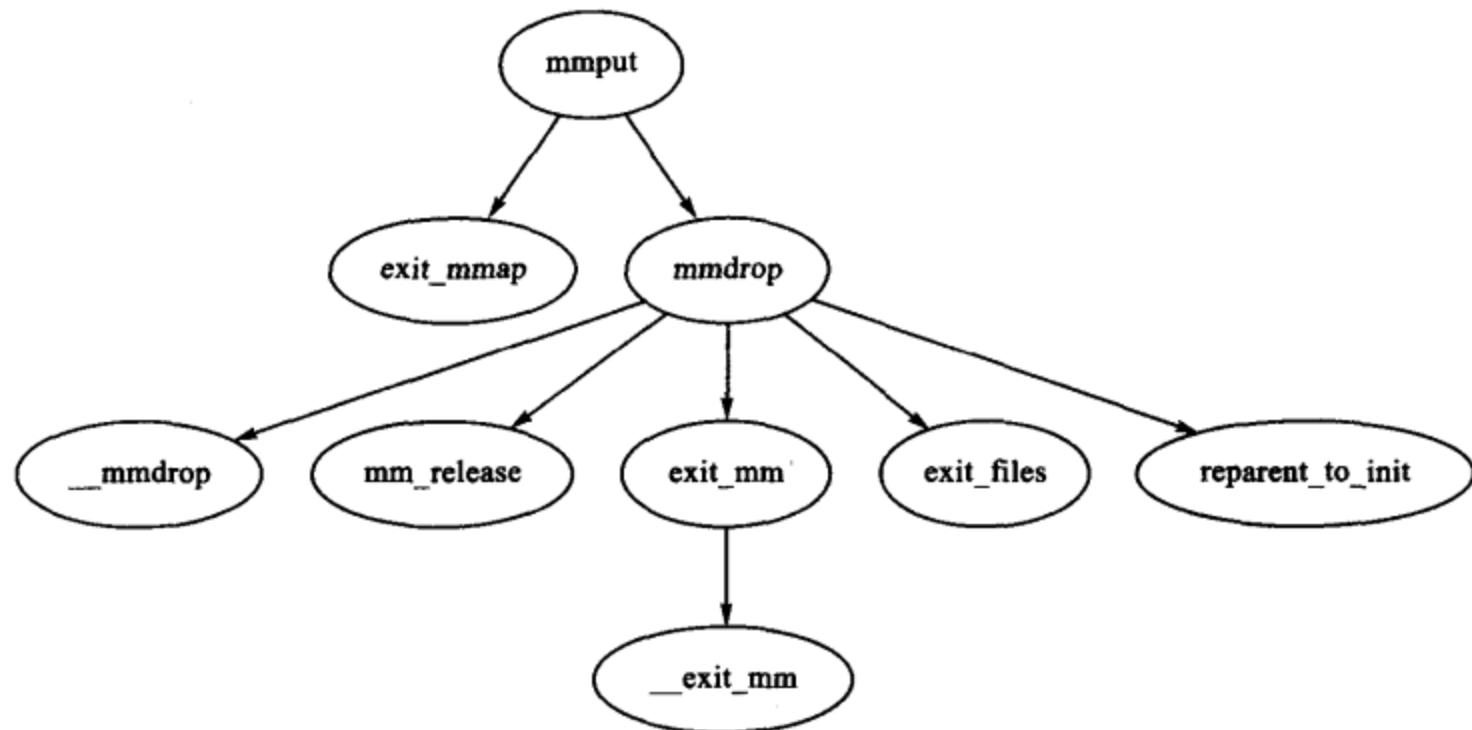


图 D.1 调用图:mmpput()

283~284 减少列表中的 mm 计数,并释放 mmlist 锁。

285 移除所有的相关映射。

286 删除 mm。

D.1.4.2 函数: mmdrop() (include/linux/sched.h)

```

765 static inline void mmdrop(struct mm_struct * mm)
766 {
767     if (atomic_dec_and_test(&mm->mm_count))
768         __mmdrop(mm);
769 }
  
```

767 原子性地将引用计数减 1。如果 mm 由延迟 tlb 交换进程所用,引用计数将比较高。

768 如果引用计数减为 0,这里将调用 __mmdrop()。

D.1.4.3 函数: __mmdrop() (kernel/fork.c)

```

265 inline void __mmdrop(struct mm_struct * mm)
266 {
267     BUG_ON(mm == &init_mm);
268     pgd_free(mm->pgd);
269     destroy_context(mm);
270     free_mm(mm);
271 }
  
```

- 267 保证 init_mm 没有被销毁。
 268 删除 PGD 表项。
 269 删除 LDT(局部描述表)。
 270 在 mm 上调用 kmem_cache_free(), 利用 slab 分配器将其释放。

D.2 创建内存区域

这一大部分讨论创建、删除和处理内存区域。

D.2.1 创建一个内存区域

创建内存区域的主要调用图如图 4.3 所示。

D.2.1.1 函数: do_mmap() (include/linux/mm.y)

这是一个对 do_mmap_pgoff() 的简单封装函数, 它完成如下大部分工作。

```
557 static inline unsigned long do_mmap(struct file * file,
558                                     unsigned long addr,
559                                     unsigned long len, unsigned long prot,
560                                     unsigned long flag, unsigned long offset)
561 {
562     unsigned long ret = -EINVAL;
563     if ((offset + PAGE_ALIGN(len)) < offset)
564         goto out;
565     if (!(offset & ~PAGE_MASK))
566         ret = do_mmap_pgoff(file, addr, len, prot, flag,
567                             offset >> PAGE_SHIFT);
568 }
```

561 缺省时, 这里返回-EINVAL。

562~563 保证区域的大小不会超过地址空间的总大小。

564~565 按页面排列 offset, 并且调用 do_mmap_pgoff() 来映射该区域。

D.2.1.2 函数: do_mmap_pgoff() (mm/mmap.c)

这个函数非常大, 所以把它分成几个部分。粗略说来, 有如下部分:

- 参数的有效检查。
- 找到一块足够大的线性地址空间来完成内存映射。如果提供了文件系统或设备特定的 `get_unmapped_area()`，就使用这个函数。否则，调用 `arch_get_unmapped_area()`。
- 计算 VM 标志位，检查它们是否违反了文件访问权限。
- 如果在要映射的地方存在一个旧的区域，修整它从而使它适合新的映射。
- 从 slab 分配器中分配 `vm_area_struct` 并填充表项。
- 链接到新的 VMA。
- 调用文件系统或者设备相关的 `mmap()` 函数。
- 更新统计并退出。

```

393 unsigned long do_mmap_pgoff(struct file * file,
394                               unsigned long addr,
395                               unsigned long len, unsigned long prot,
396                               unsigned long flags, unsigned long pgoff)
397 {
398     struct mm_struct * mm = current->mm;
399     struct vm_area_struct * vma, * prev;
400     unsigned int vm_flags;
401     int correct_wcount = 0;
402     int error;
403     rb_node_t ** rb_link, * rb_parent;
404
405     if (file && (! file->f_op || ! file->f_op->mmap))
406         return -ENODEV;
407
408     if (! len)
409         return addr;
410
411     len = PAGE_ALIGN(len);
412
413     if (len > TASK_SIZE || len == 0)
414         return -EINVAL;
415
416     if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
417         return -EINVAL;
418
419     if (mm->map_count > max_map_count)

```

420 return -ENOMEM;
421

393 与 mmap 系统调用参数有直接相关的参数如下：

- file 是若有后援文件映射的 mmap 的文件结构。
- addr 是映射的请求地址。
- len 是要 mmap 的字节数。
- prot 是该区域的权限。
- flags 是映射的标志位。
- pgoff 是开始 mmap 的文件偏移。

403~404 如果映射了一个文件或设备,这里确定提供了一个文件系统或者设备相关的 mmap 函数。对多数文件系统而言,这里将调用 generic_file_mmap() (见 D. 6. 2. 1)。

406~407 保证长度为 0 的 mmap() 不被请求。

409 保证映射限于地址空间的用户空间部分。在 x86 中,内核空间开始于 PAGE_OFFSET(3 GB)。

415~416 保证映射不会超过最大可能的文件大小。

419~420 仅允许 max_map_count 数量个映射。缺省时,这个数值为 DEFAULT_MAX_MAP_COUNT 或者 65 536 个映射。

422 /* Obtain the address to map to. we verify (or select) it and
423 * ensure that it represents a valid section of the address space.
424 */
425 addr = get_unmapped_area(file, addr, len, pgoff, flags);
426 if (addr & ~PAGE_MASK)
427 return addr;
428

425 在基本的有效性检查后,这个函数将调用设备或文件相关的 get_unmapped_area() 函数。如果没有设备相关的函数,则调用 arch_get_unmapped_area()。这个函数在 D. 3. 2. 2 进行描述。

429 /* Do simple checking here so the lower-level routines won't
430 * have to. we assume access permissions have been handled by
431 * the open of the memory object, so we don't do any here.
432 */
433 vm_flags = calc_vm_flags(prot, flags) | mm->def_flags
| VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
434
435 /* mlock MCL_FUTURE? */

```

436 if (vm_flags & VM_LOCKED) {
437     unsigned long locked = mm->locked_vm << PAGE_SHIFT;
438     locked += len;
439     if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
440         return -EAGAIN;
441 }
442

```

433 calc_vm_flags() 将 prot 和 flags 从用户空间转换为 VM 的相应标志位。

436~440 检查将来所有的映射是否都在内存中被上锁。如果是，这里保证进程不会上锁比允许它上锁还要多的内存。如果多上锁，就返回-EAGAIN。

```

443 if (file) {
444     switch (flags & MAP_TYPE) {
445         case MAP_SHARED:
446             if ((prot & PROT_WRITE) &&
447                 !(file->f_mode & FMODE_WRITE))
448                 return -EACCES;
449
450             /* Make sure we dont allow writing to
451              an append-only file.. */
452             if (IS_APPEND(file->f_dentry->d_inode) &&
453                 (file->f_mode & FMODE_WRITE))
454                 return -EACCES;
455
456             /* make sure there are no mandatory
457              locks on the file. */
458             if (locks_verify_locked(file->f_dentry->d_inode))
459                 return -EAGAIN;
460
461             /* fall through */
462         case MAP_PRIVATE:
463             if (!(file->f_mode & FMODE_READ))
464                 return -EACCES;
465             break;
466

```



```
467 default;
468 return -EINVAL;
469 }
```

443~469 如果一个文件有内存映射,这里将检查文件的访问权限。

446~447 如果请求写访问,这里保证文件已经以写方式打开。

450~451 相似地,如果文件以追加方式打开,这里保证它不能被写入。但这里并不检查 prot,因为 prot 字段仅应用于映射那些需要检查的已打开文件。

453 如果文件是强制上锁的,这里返回-EAGAIN 从而调用者将试着用第 2 种类型。

457~459 修整标志位以与文件标志位一致。

463~464 保证文件可以在 mmaping 前可以读。

```
470 } else {
471     vm_flags |= VM_SHARED | VM_MAYSHARE;
472     switch (flags & MAP_TYPE) {
473     default:
474         return -EINVAL;
475     case MAP_PRIVATE:
476         vm_flags &= ~(VM_SHARED | VM_MAYSHARE);
477         /* fall through */
478     case MAP_SHARED:
479         break;
480     }
481 }
```

471~481 如果文件映射用于匿名使用,如果请求映射是 MAP_PRIVATE,这里将修整标志位以使标志位一致。

```
483 /* Clear old maps */
484 munmap_back;
485 vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
486 if (vma && vma->vm_start < addr + len) {
487     if (do_munmap(mm, addr, len))
488         return -ENOMEM;
489     goto munmap_back;
490 }
```

485 find_vma_prepare()(见 D.2.2.2) 遍历对应于某个给定地址 VMA 的 RB 树。

486~488 如果找到一个 VMA,并且是新映射的一部分,这里就移除旧的映射,因为新的映射将涵盖这两部分。

```

491
492 /* Check against address space limit. */
493 if ((mm->total_vm << PAGE_SHIFT) + len
494 > current->rlim[RLIMIT_AS].rlim_cur)
495 return -ENOMEM;
496
497 /* Private writable mapping? Check memory availability.. */
498 if ((vm_flags & (VM_SHARED | VM_WRITE)) == VM_WRITE &&
499 ! (flags & MAP_NORESERVE) &&
500 ! vm_enough_memory(len >> PAGE_SHIFT))
501 return -ENOMEM;
502
503 /* Can we just expand an old anonymous mapping? */
504 if (! file && ! (vm_flags & VM_SHARED) && rb_parent)
505 if (vma_merge(mm, prev, rb_parent,
506 addr, addr + len, vm_flags))
506 goto out;
507

```

493~495 保证新映射将不会超过允许一个进程拥有的总 VM, 现在还不清楚为什么不在前面进行这项检查。

498~501 如果调用者没有特别请求那些没有利用 MAP_NORESERVE 检查的空闲空间, 并且这是一个私有映射, 就在这里保证当前条件下有足够的内存来满足映射。

504~506 如果两个相邻的内存映射是匿名的, 并且可被视为一个, 这里将扩展旧的映射而不是创建一个新映射。

```

508 /* Determine the object being mapped and call the appropriate
509 * specific mapper. the address has already been validated,
510 * but not unmapped, but the maps are removed from the list.
511 */
512 vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
513 if (! vma)
514 return -ENOMEM;
515
516 vma->vm_mm = mm;
517 vma->vm_start = addr;
518 vma->vm_end = addr + len;
519 vma->vm_flags = vm_flags;
520 vma->vm_page_prot = protection_map[vm_flags & 0x0f];
521 vma->vm_ops = NULL;

```

```

522 vma->vm_pgoff = pgoff;
523 vma->vm_file = NULL;
524 vma->vm_private_data = NULL;
525 vma->vm_raend = 0;

```

512 从 slab 分配器中分配一个 vm_area_struct。

516~525 填充基本的 vm_area_struct 字段。

```

527 if (file) {
528     error = -EINVAL;
529     if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
530         goto free_vma;
531     if (vm_flags & VM_DENYWRITE) {
532         error = deny_write_access(file);
533         if (error)
534             goto free_vma;
535         correct_wcount = 1;
536     }
537     vma->vm_file = file;
538     get_file(file);
539     error = file->f_op->mmap(file, vma);
540     if (error)
541         goto unmap_and_free_vma;

```

527~541 如果这个文件被映射,填充文件相关的字段。

529~530 这两个是一个文件映射无效的标志位,所以这里释放 vm_area_struct 并返回。

531~536 这个标志位由系统调用 mmap() 清除,也由那些直接调用这个函数的内核模块清除。以前,如果写了底层的函数就会返回一个-ETXTBUSY 为调用进程。

537 填充 vm_file 字段。

538 将文件使用计数加 1。

539 调用文件系统或设备相关的 mmap() 函数。在许多文件系统中,这里将调用 generic_file_mmap()(见 D. 6. 2. 1)。

540~541 如果调用了一个错误,这里将转到 unmap_and_free_vma 清除并返回错误。

```

542 } else if (flags & MAP_SHARED) {
543     error = shmem_zero_setup(vma);
544     if (error)
545         goto free_vma;
546 }
547

```

543 如果这是一个匿名共享映射,则区域由 shmem_zero_setup()来创建和建造(见 L. 7.1 小节)。匿名共享也由一个虚拟 tmpfs 文件系统后援,所以它们可以很好地在交换区同步。其中的写回函数是 shmem_writepage()(见 L. 6.1 小节)。

```
548 /* Can addr have changed??  
549 *  
550 * Answer: Yes, several device drivers can do it in their  
551 * f_op->mmap method. -DaveM  
552 */  
553 if (addr != vma->vm_start) {  
554 /*  
555 * It is a bit too late to pretend changing the virtual  
556 * area of the mapping, we just corrupted userspace  
557 * in the do_munmap, so FIXME (not in 2.4 to avoid  
558 * breaking the driver API).  
559 */  
560 struct vm_area_struct * stale_vma;  
561 /* Since addr changed, we rely on the mmap op to prevent  
562 * collisions with existing vmas and just use  
563 * find_vma_prepare to update the tree pointers.  
564 */  
565 addr = vma->vm_start;  
566 stale_vma = find_vma_prepare(mm, addr, &prev,  
567 &rb_link, &rb_parent);  
568 /*  
569 * Make sure the lowlevel driver did its job right.  
570 */  
571 if (unlikely(stale_vma && stale_vma->vm_start <  
      vma->vm_end)) {  
572 printk(KERN_ERR "buggy mmap operation: [< %p>]\n",  
573 file ? file->f_op->mmap : NULL);  
574 BUG();  
575 }  
576 }  
577  
578 vma_link(mm, vma, prev, rb_link, rb_parent);  
579 if (correct_wcount)  
580 atomic_inc(&file->f_dentry->d_inode->i_writecount);  
581
```

553~576 如果地址发生改变,则意味着设备相关的 mmap 设备把 VMA 地址移到其他地方。利用函数 find_vma_prepare()(见 D. 2. 2. 2)来找到 VMA 移到的位置。

578 连接新的 vm_area_struct。

579~580 更新文件写计数。

```

582 out:
583 mm->total_vm += len >> PAGE_SHIFT;
584 if (vm_flags & VM_LOCKED) {
585 mm->locked_vm += len >> PAGE_SHIFT;
586 make_pages_present(addr, addr + len);
587 }
588 return addr;
589
590 unmap_and_free_vma:
591 if (correct_wcount)
592 atomic_inc(&file->f_dentry->d_inode->i_writecount);
593 vma->vm_file = NULL;
594 fput(file);
595
596 /* Undo any partial mapping done by a device driver. */
597 zap_page_range(mm,
598                 vma->vm_start,
599                 vma->vm_end - vma->vm_start);
598 free_vma:
599 kmem_cache_free(vm_area_cachep, vma);
600 return error;
601 }
```

583~588 更新进程 mm_struct 的统计计数并返回新地址。

590~597 如果文件在失败前已经部分地映射,则将到达这里。在这里将更新写统计计数,然后所有的用户页面都由 zap_page_range()移除。

598~600 这里的 goto 用于在 vm_area_struct 创建后映射失败的情况下。在返回错误前它退回到 slab 分配器。

D. 2. 2 插入一个内存区域

insert_vm_struct()的调用图如图 4.5 所示。

D.2.2.1 函数: __insert_vm_struct() (mm/mmap.c)

这是将一个新 VMA 插入地址空间的顶层函数。另外有个类似的函数称为 insert_vm_struct(), 在这里不详细描述, 因为它们惟一的区别就是有一行代码增加了 map_count。

```

1174 void __insert_vm_struct(struct mm_struct * mm,
                           struct vm_area_struct * vma)
1175 {
1176     struct vm_area_struct * __vma, * prev;
1177     rb_node_t ** rb_link, * rb_parent;
1178
1179     __vma = find_vma_prepare(mm, vma->vm_start, &prev,
1180                             &rb_link, &rb_parent);
1180     if (__vma && __vma->vm_start < vma->vm_end)
1181         BUG();
1182     __vma_link(mm, vma, prev, rb_link, rb_parent);
1183     mm->map_count++;
1184     validate_mm(mm);
1185 }
```

1174 这里的参数表示线性地址空间的 mm_struct 和待插入的 vm_area_struct。

1179 find_vma_prepare() (见 D.2.2.2) 定位新 VMA 可以被插入的位置。它将被插入在 prev 和 __vma 之间, 并且也返回红黑树的所需节点。

1180~1181 这里检查保证所返回的 VMA 是无效的。实际上, 如果没有手动地向地址空间插入伪 VMA, 则这个条件不可能发生。

1182 这个函数完成把 VMA 结构链接到线性链表以及红黑树中的实际工作。

1183 增加 map_count 来显示增加了一个新的映射。这一行没有在 insert_vm_struct() 中提供。

1184 validate_mm() 是一个调试红黑树的宏。如果设置了 DEBUG_MM_RB 标志位, 将遍历 VMA 的线性链表和红黑树以保证它有效。由于这个遍历函数是一个递归函数, 所以很重要的是它必须在真正需要的时候才调用, 因为大量的映射将导致栈溢出。如果没有设立这个标志位, validate_mm() 不会做任何事。

D.2.2.2 函数: find_vma_prepare() (mm/mmap.c)

这个函数负责在给定地址找到一个合适的位置来插入 VMA。它通过实际的返回和函数参数返回一系列的信息。待插入 VMA 前面的节点也被返回。pprev 是前一个节点, 这里需要它是因为链表是个单链表。rb_link 和 rb_parent 分别是父节点和叶节点, VMA 插入两者之间。

```
246 static struct vm_area_struct * find_vma_prepare(
247     struct mm_struct * mm,
248     unsigned long addr,
249     struct vm_area_struct ** pprev,
250     rb_node_t *** rb_link,
251     rb_node_t ** rb_parent)
252 {
253     struct vm_area_struct * vma;
254     rb_node_t ** __rb_link, * __rb_parent, * rb_prev;
255     vma = NULL;
256
257     while (* __rb_link) {
258         struct vm_area_struct * vma_tmp;
259
260         __rb_parent = * __rb_link;
261         vma_tmp = rb_entry(__rb_parent,
262                           struct vm_area_struct, vm_rb);
263
264         if (vma_tmp->vm_end > addr) {
265             if (vma_tmp->vm_start <= addr)
266                 return vma;
267             __rb_link = &__rb_parent->rb_left;
268         } else {
269             rb_prev = __rb_parent;
270             __rb_link = &__rb_parent->rb_right;
271         }
272     }
273
274     * pprev = NULL;
275     if (rb_prev)
276         * pprev = rb_entry(rb_prev, struct vm_area_struct, vm_rb);
277     * rb_link = __rb_link;
278     * rb_parent = __rb_parent;
279     return vma;
280 }
```

246 函数的参数在前面已经说明。

253~255 初始化查找。

263~272 这是与在 `find_vma()` 中相似的一棵遍历树。惟一的实际区别在于最后一个被访问的节点由 `__rb_link` 和 `__rb_parent` 变量记录。

275~276 通过红黑树获取后继 VMA。

279 返回前继 VMA。

D. 2.2.3 函数: `vma_link()` (`mm/mmap.c`)

这是一个把 VMA 链接到相应链表中的高层函数。它负责获取所需的锁来保证插入的安全性。

```

337 static inline void vma_link(struct mm_struct * mm,
338                             struct vm_area_struct * vma,
339                             struct vm_area_struct * prev,
340                             rb_node_t ** rb_link, rb_node_t * rb_parent)
341 {
342     lock_vma_mappings(vma);
343     spin_lock(&mm->page_table_lock);
344     __vma_link(mm, vma, prev, rb_link, rb_parent);
345     spin_unlock(&mm->page_table_lock);
346     unlock_vma_mappings(vma);
347     validate_mm(mm);
348 }
```

337 `mm` 是 VMA 要插入的地址空间。`prev` 是 VMA 线性链表中的后接 VMA。`rb_link` 和 `rb_parent` 是需要完成 rb 插入的节点。

340 这个函数获取自旋锁来保护表示内存映射文件的 `address_space`。

341 获取保护整个 `mm_struct` 的页表锁。

342 插入 VMA。

343 释放保护 `mm_struct` 的锁。

345 解锁文件的 `address_space`。

346 增加这个 `mm` 中的映射数量。

347 如果设置了 `DEBUG_MM_RB` 标志位, 将检查 RB 树和链表以保证它们有效。

D. 2.2.4 函数: `__vma_link()` (`mm/mmap.c`)

这里仅是调用 3 个负责把 VMA 链接到 3 个链表中的辅助函数, 这些链表把 VMA 链到

一起。

```

329 static void __vma_link(struct mm_struct * mm,
  struct vm_area_struct * vma,
  struct vm_area_struct * prev,
330 rb_node_t ** rb_link, rb_node_t * rb_parent)
331 {
332 __vma_link_list(mm, vma, prev, rb_parent);
333 __vma_link_rb(mm, vma, rb_link, rb_parent);
334 __vma_link_file(vma);
335 }

```

332 通过 `vm_next` 字段将 VMA 链接到该 `mm` 中 VMA 的一个线性链表中。

333 把 VMA 链接到 `mm` 中 VMA 的红黑树中, 红黑树的根节点在 `vm_rb` 字段中。

334 链接 VMA 到共享映射 VMA 链接中。通过这个函数以及 `vm_next_share` 和 `vm_pprev_share` 字段, 内存映射的文件跨越多个 `mm` 而链接到一起。

D.2.2.5 函数: __vma_link_list() (mm/mmap.c)

```

282 static inline void __vma_link_list(struct mm_struct * mm,
  struct vm_area_struct * vma,
  struct vm_area_struct * prev,
283 rb_node_t * rb_parent)
284 {
285 if (prev) {
286 vma->vm_next = prev->vm_next;
287 prev->vm_next = vma;
288 } else {
289 mm->mmap = vma;
290 if (rb_parent)
291 vma->vm_next = rb_entry(rb_parent,
  struct vm_area_struct,
  vm_rb);
292 else
293 vma->vm_next = NULL;
294 }
295 }

```

285 如果 `prev` 为 `NULL`, 就仅把 VMA 插入到链表中。

289 如果不为 `NULL`, 则这里是第 1 次映射, 而且链表的第一个元素存放在 `mm_struct` 中。

290 VMA 存放为一个父节点。

D.2.2.6 函数: __vma_link_rb() (mm/mmap.c)

这个函数的主要工作存放于<linux/rbtree.c>中,本书将不会详细讨论。

```
297 static inline void __vma_link_rb(struct mm_struct * mm,
298                                     struct vm_area_struct * vma,
299                                     rb_node_t ** rb_link,
300                                     rb_node_t * rb_parent)
301 {
300 rb_link_node(&vma->vm_rb, rb_parent, rb_link);
301 rb_insert_color(&vma->vm_rb, &mm->mm_rb);
302 }
```

D.2.2.7 函数: __vma_link_file() (mm/mmap.c)

这个函数将 VMA 链接到一个共享文件映射的链表中。

```
304 static inline void __vma_link_file(struct vm_area_struct * vma)
305 {
306     struct file * file;
307
308     file = vma->vm_file;
309     if (file) {
310         struct inode * inode = file->f_dentry->d_inode;
311         struct address_space * mapping = inode->i_mapping;
312         struct vm_area_struct ** head;
313
314         if (vma->vm_flags & VM_DENYWRITE)
315             atomic_dec(&inode->i_writecount);
316
317         head = &mapping->i_mmap;
318         if (vma->vm_flags & VM_SHARED)
319             head = &mapping->i_mmap_shared;
320
321         /* insert vma into inodes share list */
322         if ((vma->vm_next_share = * head) != NULL)
323             (* head)->vm_pprev_share = &vma->vm_next_share;
324         * head = vma;
325         vma->vm_pprev_share = head;
326     }
```

327 }

309 检查 VMA 是否有一个共享文件映射。如果没有，则函数不再做任何事。

310~312 从 VMA 中摘取与映射有关的信息。

314~315 如果即使在具有写权限时该映射也不可写，将 `i_writecount` 字段减 1。该字段的负值表示文件是内存映射的，可能不可写。所以试着以写方式打开文件将失败。

317~319 检查以保证这是一个共享映射。

322~325 将 VMA 插入到一个共享映射链表中。

D.2.3 合并相邻区域

D.2.3.1 函数: `vma_merge()` (`mm/mmap.c`)

这个函数检查由 `prev` 指向的区域可能向前扩展到从 `addr` 到 `end` 之间的区域，而不是分配一个新的 VMA。如果无法向前扩展，就检查 VMA 头，看它是否可以向后扩展。

```
350 static int vma_merge(struct mm_struct * mm,
351                     struct vm_area_struct * prev,
352                     rb_node_t * rb_parent,
353                     unsigned long addr, unsigned long end,
354                     unsigned long vm_flags)
355
356 {
357     spinlock_t * lock = &mm->page_table_lock;
358     if (!prev) {
359         prev = rb_entry(rb_parent, struct vm_area_struct, vm_rb);
360         goto merge_next;
361     }
362 }
```

350 参数如下：

- `mm` 是 VMA 所属的 `mm`。
- `prev` 是我们所感兴趣地址前的 VMA。
- `rb_parent` 是由 `find_vma_prepare()` 返回的父 RB 节点。
- `addr` 是待合并区域的起始地址。
- `end` 是待合并区域的结束地址。
- `vm_flags` 是待合并区域的权限标志位。

353 这是 `mm` 的锁。

354~357 如果没有传递 `prev`，则意味着正测试且待合并的 VMA 位于 `addr` 到 `end` 区域之前。VMA 的那个表项从 `rb_parent` 前部摘取。

```
358 if (prev->vm_end == addr && can_vma_merge(prev, vm_flags)) {  
359     struct vm_area_struct * next;  
360  
361     spin_lock(lock);  
362     prev->vm_end = end;  
363     next = prev->vm_next;  
364     if (next && prev->vm_end == next->vm_start &&  
         can_vma_merge(next, vm_flags)) {  
365         prev->vm_end = next->vm_end;  
366         __vma_unlink(mm, next, prev);  
367         spin_unlock(lock);  
368  
369         mm->map_count--;  
370         kmem_cache_free(vm_area_cachep, next);  
371         return 1;  
372     }  
373     spin_unlock(lock);  
374     return 1;  
375 }  
376  
377     prev = prev->vm_next;  
378     if (prev) {  
379         merge_next;  
380         if (! can_vma_merge(prev, vm_flags))  
381             return 0;  
382         if (end == prev->vm_start) {  
383             spin_lock(lock);  
384             prev->vm_start = addr;  
385             spin_unlock(lock);  
386             return 1;  
387         }  
388     }  
389  
390     return 0;  
391 }
```

358~375 检查 prev 指向的区域是否可以扩展到覆盖当前区域。

358 函数 can_vma_merge() 检查那些带有 vm_flag 标志位的权限以及那些没有文件映射 VMA(例如,它是匿名的)的 prev 的权限。如果为真,prev 的区域可以扩展。

361 锁定 mm。
 362 把 VMA 区域的末端(vm_end)扩展到新的映射末端(end)。
 363 next 是现在新扩展后的 VMA 前面的 VMA。
 364 检查扩展后的区域是否可以与前面的 VMA 合并。
 365 如果可以,这里持续扩展区域以覆盖下一个 VMA。
 366 由于已经合并了一个 VMA,有一个区域就变成了无效的,可以链接出去。
 367 没有进一步调整 mm 结构,所以释放锁。
 369 少了一个映射的区域,map_count 减 1。
 370 删除描述合并了的 VMA 的结构。
 371 返回成功。
 377 如果到了这一行,则意味着 prev 指向的区域不能向前扩展,所以检查前面的区域是否可以向后扩展。
 382~388 除了不是用调整的 vm_end 来覆盖 end,,还可以使用 vm_start 扩展来覆盖 addr,这里与前面一块的思想相同。

D. 2. 3. 2 函数: can_vma_merge() (include/linux/mm.h)

这个小函数检查给定 VMA 的权限是否符合 vm_flags 的权限。

```
582 static inline int can_vma_merge(struct vm_area_struct * vma,
583                                  unsigned long vm_flags)
584 {
585     if (! vma->vm_file && vma->vm_flags == vm_flags)
586         return 1;
587     else
588         return 0;
589 }
```

584 一目了然。如果没有文件或设备映射(如是匿名的)以及两块区域的 VMA 标志位匹配,则返回 true。

D. 2. 4 重映射并移动一个内存区域

D. 2. 4. 1 函数: sys_mremap() (mm/mremap.c)

这个函数的调用图如图 4. 6 所示。这是一个重映射内存区域的系统服务调用。

```
347 asmlinkage unsigned long sys_mremap(unsigned long addr,
348                                         unsigned long old_len, unsigned long new_len,
```

```

349 unsigned long flags, unsigned long new_addr)
350 {
351 unsigned long ret;
352
353 down_write(&current->mm->mmap_sem);
354 ret = do_mremap(addr, old_len, new_len, flags, new_addr);
355 up_write(&current->mm->mmap_sem);
356 return ret;
357 }

```

347~349 参数与在 mremap() 帮助页中描述的一样。

353 获得 mm 信号量。

354 do_mremap()(见 D. 2. 4. 2) 是重映射一块区域的高层函数。

355 释放 mm 信号量。

356 返回重映射状态。

D. 2. 4. 2 函数: do_mremap() (mm/mremap.c)

这个函数完成重映射,修改尺寸以及移动一块内存区域的实际工作。它相当长,但也可以分成几个独立的部分,在这里分别处理。粗略地说,主要的任务如下:

- 检查使用标志位以及页面排列长度。
- 处理 MAP_FIXED 的设置和移到新地方的区域位置。
- 如果是收缩一个区域,则允许它无条件发生。
- 如果增大区域或者移动区域,提前进行一系列检查来保证允许移动以及移动的安全性。
- 处理已经扩展了区域但是不能移动的情况。
- 最后,处理区域需要改变大小和移动的情况。

```

219 unsigned long do_mremap(unsigned long addr,
220 unsigned long old_len, unsigned long new_len,
221 unsigned long flags, unsigned long new_addr)
222 {
223 struct vm_area_struct * vma;
224 unsigned long ret = -EINVAL;
225
226 if (flags & ~(MREMAP_FIXED | MREMAP_MAYMOVE))
227 goto out;
228
229 if (addr & ~PAGE_MASK)

```



```

230 goto out;
231
232 old_len = PAGE_ALIGN(old_len);
233 new_len = PAGE_ALIGN(new_len);
234

```

219 函数的参数如下：

- `addr` 是旧的起始地址。
- `old_len` 是旧的区域长度。
- `new_len` 是新的区域长度。
- `flags` 是传入的是一个可选标志位。如果设置了 `MREMAP_MAYMOVE`, 则意味着如果当前空间没有足够的线性地址空间时, 那么允许移动区域。如果设置了 `MREMAP_FIXED`, 则意味着整个区域将移动到指定的 `new_addr` 并具有新长度。从 `new_addr` 到 `new_addr+new_len` 的区域将通过 `do_munmap()` 来解除映射。
- `new_addr` 是如果区域移动后新区域的地址。

224 这里缺省返回-EINVAL 报告无效的参数。

226~227 保证没有使用那两个允许使用的标志位之外的标志位。

229~230 传入的地址必须与页对齐。

232~233 传入与页对齐的区域长度。

```

236 if (flags & MREMAP_FIXED) {
237 if (new_addr & ~PAGE_MASK)
238 goto out;
239 if (!(flags & MREMAP_MAYMOVE))
240 goto out;
241
242 if (new_len > TASK_SIZE || new_addr > TASK_SIZE - new_len)
243 goto out;
244
245 /* Check if the location were moving into overlaps the
246 * old location at all, and fail if it does.
247 */
248 if ((new_addr <= addr) && (new_addr + new_len) > addr)
249 goto out;
250
251 if ((addr <= new_addr) && (addr + old_len) > new_addr)
252 goto out;
253

```



```
254 do_munmap(current->mm, new_addr, new_len);
255 }
```

这一块处理区域地址固定并且必须全部移动的情况。它保证将移动的区域是安全的且明确被解除映射的。

236 MREMAP_FIXED 是表明地址为固定的标志位。

237~238 指定的 new_addr 必须是页对齐的。

239~240 如果指定了 MREMAP_FIXED, 也必须使用 MAYMOVE 标志位。

242~243 保证可变大小的区域不能超过 TASK_SIZE。

248~249 如注释, 所使用的两个区域不能重叠。

254 对将要使用的区域解除映射。假定调用者确定这块区域没有被其他重要进程使用。

```
261 ret = addr;
262 if (old_len >= new_len) {
263 do_munmap(current->mm, addr + new_len, old_len - new_len);
264 if (!(flags & MREMAP_FIXED) || (new_addr == addr))
265 goto out;
266 }
```

261 这里, 可变大小区域的地址是返回值。

262 如果旧的长度超过新的长度, 将缩小区域。

263 取消那些没有使用区域的映射。

264~265 如果不移动区域, 或者由于 MREMAP_FIXED 没有使用或者新地址与旧地址不匹配, 则跳到 out, 在那里返回地址。

```
271 ret = -EFAULT;
272 vma = find_vma(current->mm, addr);
273 if (!vma || vma->vm_start > addr)
274 goto out;
275 /* We can't remap across vm area boundaries */
276 if (old_len > vma->vm_end - addr)
277 goto out;
278 if (vma->vm_flags & VM_DONTEXPAND) {
279 if (new_len > old_len)
280 goto out;
281 }
282 if (vma->vm_flags & VM_LOCKED) {
283 unsigned long locked = current->mm->locked_vm << PAGE_SHIFT;
284 locked += new_len - old_len;
285 ret = -EAGAIN;
```



```

286 if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
287 goto out;
288 }
289 ret = -ENOMEM;
290 if ((current->mm->total_vm << PAGE_SHIFT) + (new_len - old_len)
291 > current->rlim[RLIMIT_AS].rlim_cur)
292 goto out;
293 /* Private writable mapping? Check memory availability.. */
294 if ((vma->vm_flags & (VM_SHARED | VM_WRITE)) == VM_WRITE &&
295 ! (flags & MAP_NORESERVE) &&
296 ! vm_enough_memory((new_len - old_len) >> PAGE_SHIFT))
297 goto out;

```

这一块进行一系列的检查来保证增大或者移动区域是安全的。

271 在这里,缺省的动作是返回-EFAULT, 这将导致一个段错误,因为正使用的内存区域是无效的。

272 找到所请求地址的 VMA。

273 如果返回的 VMA 和这个地址不匹配,则返回一个无效地址作为异常。

276~277 如果传入的 old_len 超过了 VMA 的长度,则意味着使用者尝试重映射多个区域,而这是不允许的。

278~281 如果 VMA 已经被显式地标记为不可改变大小,这里将报异常。

282~283 如果该 VMA 的页面必须在内存中上锁,这里重新计算将在内存中锁定的页面数量。如果这样的页面已经超过资源的极限集,这里将返回 EAGAIN,向调用者表明这片区域已经锁定,并且无法改变大小。

289 这里返回的缺省值用于表明没有足够的内存。

290~292 保证用户不会超过使用它们所允许分配的内存。

294~297 保证在利用 vm_enough_memory()(见 M. 1.1 小节)重新划定大小后有足够的内存来满足请求。

```

302 if (old_len == vma->vm_end - addr &&
303 ! ((flags & MREMAP_FIXED) && (addr != new_addr)) &&
304 (old_len != new_len || ! (flags & MREMAP_MAYMOVE))) {
305 unsigned long max_addr = TASK_SIZE;
306 if (vma->vm_next)
307 max_addr = vma->vm_next->vm_start;
308 /* can we just expand the current mapping? */
309 if (max_addr - addr >= new_len) {
310 int pages = (new_len - old_len) >> PAGE_SHIFT;

```

```

311 spin_lock(&vma->vm_mm->page_table_lock);
312 vma->vm_end = addr + new_len;
313 spin_unlock(&vma->vm_mm->page_table_lock);
314 current->mm->total_vm += pages;
315 if (vma->vm_flags & VM_LOCKED) {
316 current->mm->locked_vm += pages;
317 make_pages_present(addr + old_len,
318 addr + new_len);
319 }
320 ret = addr;
321 goto out;
322 }
323 }

```

这一块处理区域正拓展却无法移动的情况。

302 如果它是要重映射的满区域且……

303 区域明确没有被移动且……

304 正扩展的区域无法移动，则……

305 设置 TASK_SIZE 可用的最大地址，在 x86 上为 3 GB。

306~307 如果有另外一个区域，这里设置下一块区域的最大地址。

309~322 只有新划分的区域与下一个 VMA 没有重叠时才允许扩展。

310 计算所需的额外页面数。

311 上锁 mm 自旋锁。

312 扩展 VMA。

313 释放 mm 自旋锁。

314 更新 mm 统计。

315~319 如果该区域的页面在内存中锁定，这里使它们保持原样。

320~321 返回重新划分大小的区域地址。

```

329 ret = -ENOMEM;
330 if (flags & MREMAP_MAYMOVE) {
331 if (!(flags & MREMAP_FIXED)) {
332 unsigned long map_flags = 0;
333 if (vma->vm_flags & VM_SHARED)
334 map_flags |= MAP_SHARED;
335
336 new_addr = get_unmapped_area(vma->vm_file, 0,
new_len, vma->vm_pgoff, map_flags);

```

```

337 ret = new_addr;
338 if (new_addr & ~PAGE_MASK)
339     goto out;
340 }
341 ret = move_vma(vma, addr, old_len, new_len, new_addr);
342 }
343 out:
344 return ret;
345 }

```

为了扩展区域,分配一个新区域,将旧的区域移到其中。

329 缺省动作是返回信息,提示无内存可用。

330 检查以保证允许移动区域。

331 如果没有指定 MREMAP_FIXED,这里意味着没有提供新地址,所以必须找到一个。

333~334 保留 MAP_SHARED 选项。

336 找到一片足够大小的用于扩展且未映射的内存区域。

337 返回值是新区域的地址。

338~339 对不是页对齐的返回地址,将需要 get_unmapped_area()进行打散操作。这里将可能发生错误百出的设备驱动程序没有正确实现 get_unmapped_area()的情况。

341 调用 move_vma()来移动区域。

343~344 如果成功则返回地址,否则返回错误码。

D.2.4.3 函数: move_vma() (mm/mremap.c)

这个函数的调用图如图 4.7 所示。这个函数负责从一个 VMA 移动所有的页表项到另一片区域。如果需要的话,将为这块移动的区域分配一块新 VMA。就像前面的函数一样,它也是很长的,但我们可以把它分成如下不同的部分:

- 函数的前面部分找到在待移动区域前面的 VMA,以及待映射区域前面的 VMA。
- 处理新地址在两块现存 VMA 之间的情况。它决定了前面的区域是否可以向前扩展到前一个区域或者后面的区域可以向后扩展来覆盖一块新的映射区域。
- 处理新地址是在链表中最后那个 VMA 的情况。它决定前面的区域是否可以向前扩展。
- 如果一块区域不可以扩展,则从 slab 分配器中分配一块新的 VMA。
- 调用 move_page_tables(),如果分配了一块新的 VMA 则填充新 VMA 内容,更新统计数量然后返回。

```

125 static inline unsigned long move_vma(struct vm_area_struct * vma,
126 unsigned long addr, unsigned long old_len, unsigned long

```

```

127 new_len, unsigned long new_addr)
128 {
129 struct mm_struct * mm = vma->vm_mm;
130 struct vm_area_struct * new_vma, * next, * prev;
131 int allocated_vma;
132
133 new_vma = NULL;
134 next = find_vma_prev(mm, new_addr, &prev);

```

125~127 参数如下：

- vma 是移动地址所属的 VMA。
- addr 是移动区域的首地址。
- old_len 是待移动区域的旧长度。
- new_len 是待移动区域的新长度。
- new_addr 是重定位的新地址。

134 找到将移动地址前面的 VMA, 它由 prev 指向, 在新映射后的区域由 next 指向, 返回 next。

```

135 if (next) {
136 if (prev && prev->vm_end == new_addr &&
137 can_vma_merge(prev, vma->vm_flags) &&
    !vma->vm_file && !(vma->vm_flags & VM_SHARED)) {
138 spin_lock(&mm->page_table_lock);
139 prev->vm_end = new_addr + new_len;
140 spin_unlock(&mm->page_table_lock);
141 new_vma = prev;
142 if (next != prev->vm_next)
143 BUG();
144 if (prev->vm_end == next->vm_start &&
    can_vma_merge(next, prev->vm_flags)) {
145 spin_lock(&mm->page_table_lock);
146 prev->vm_end = next->vm_end;
147 __vma_unlink(mm, next, prev);
148 spin_unlock(&mm->page_table_lock);
149
150 mm->map_count--;
151 kmem_cache_free(vm_area_cachep, next);
152 }
153 } else if (next->vm_start == new_addr + new_len &&

```

```

154 can_vma_merge(next, vma->vm_flags) &&
155   ! vma->vm_file && !(vma->vm_flags & VM_SHARED)) {
156   spin_lock(&mm->page_table_lock);
157   next->vm_start = new_addr;
158   new_vma = next;
159 }
160 } else {

```

在这一块,新地址在两个存在的 VMA 之间。检查前面的区域是否可以扩展来覆盖整个映射,然后检查它是否也可以覆盖下一个 VMA。如果它不能扩展,就检查下一个区域是否可以向后扩展。

136~137 如果前面的区域与待映射的地址邻接,并且可能合并,则进入这一块,试着扩展区域。

138 上锁 mm。

139 扩展前面的区域来覆盖整个新地址。

140 解锁 mm。

141 新 VMA 现在是刚才扩展区域前面的 VMA。

142~143 保证 VMA 链表是完整的。有时一块逻辑遭到损坏的设备驱动将可能导致这种情况发生。

144 检查这片区域是否可以向前扩展来包含下一个区域。

145 如果可以,则锁定 mm。

146 进一步扩展 VMA 来覆盖下一个 VMA。

147 有一个额外的 VMA,所以从链表中移除它。

148 解锁 mm。

149 已经减少了一个映射,所以更新 map_count。

150 释放由内存映射使用的内存。

151 如果 prev 区域不能向前扩展,检查 next 指向的区域是否可以向后扩展来覆盖新的映射。

152 如果可以,则锁定 mm。

153 向后扩展映射。

154 解锁 mm。

155 现在表示新映射 VMA 的是 next。

```

161 prev = find_vma(mm, new_addr-1);
162 if (prev && prev->vm_end == new_addr &&
163 can_vma_merge(prev, vma->vm_flags) && ! vma->vm_file &&

```

```

161 ! (vma->vm_flags & VM_SHARED)) {
162 spin_lock(&mm->page_table_lock);
163 prev->vm_end = new_addr + new_len;
164 spin_unlock(&mm->page_table_lock);
165 new_vma = prev;
166 }
167
168 }
169 }

```

这一块用于新映射区域是最后一块 VMA (next 为 NULL) 的情况, 检查前面的区域是否可以扩展。

161 获取前面映射的区域。

162~163 检查区域是否可以被映射。

164 上锁 mm。

165 扩展前面的区域来覆盖新的映射。

166 解锁 mm。

167 现在表示新映射 VMA 的是 prev。

```

170
171 allocated_vma = 0;
172 if (! new_vma) {
173 new_vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
174 if (! new_vma)
175 goto out;
176 allocated_vma = 1;
177 }
178

```

171 设置标志位以表明没有分配新 VMA。

172 如果一个 VMA 还没有扩展来覆盖新的映射区, 则……

173 从一个 slab 分配器来分配新的 VMA。

174~175 如果不能分配, 跳转到 out, 在那里返回失败。

176 设置标志位以表明已经分配新 VMA。

```

179 if (! move_page_tables(current->mm, new_addr, addr, old_len)) {
180 unsigned long vm_locked = vma->vm_flags & VM_LOCKED;
181
182 if (allocated_vma) {
183 * new_vma = * vma;
184 new_vma->vm_start = new_addr;
185 new_vma->vm_end = new_addr + new_len;

```

```

186 new_vma->vm_pgoff +=  

    (addr-vma->vm_start) >> PAGE_SHIFT;  

187 new_vma->vm_raend = 0;  

188 if (new_vma->vm_file)  

189     get_file(new_vma->vm_file);  

190 if (new_vma->vm_ops && new_vma->vm_ops->open)  

191     new_vma->vm_ops->open(new_vma);  

192 insert_vm_struct(current->mm, new_vma);  

193 }  

194 do_munmap(current->mm, addr, old_len);  

197 current->mm->total_vm += new_len >> PAGE_SHIFT;  

198 if (new_vma->vm_flags & VM_LOCKED) {  

199     current->mm->locked_vm += new_len >> PAGE_SHIFT;  

200     make_pages_present(new_vma->vm_start,  

201     new_vma->vm_end);  

202 }  

203 return new_addr;  

204 }  

205 if (allocated_vma)  

206     kmem_cache_free(vm_area_cachep, new_vma);  

207 out:  

208 return -ENOMEM;  

209 }

```

179 move_page_tables(见 D. 2. 4. 6)负责复制所有的页表项,返回 0 表示成功。

182~193 如果分配了一个新的 VMA,这里填充所有相关的内容细节,包括文件/设备表项,并调用 insert_vm_struct()(见 D. 2. 2. 1)来将其插入到各种 VMA 链表中。

194 取消旧区域的映射,因为不再需要它们了。

197 更新该进程的 total_vm 大小。旧的区域大小无关紧要,因为这里是在 do_munmap() 中进行的。

198~202 如果 VMA 有 VM_LOCKED 标志,在一片区域中的所有页面都利用 make_pages_present() 来设置为当前。

203 返回新区域的地址。

205~206 这是一个错误路径,如果分配了一块 VMA,则删除它。

208 返回内存溢出错误。

D. 2. 4. 4 函数: make_pages_present() (mm/memory.c)

这个函数使所有在 addr 和 end 之间的页面变为当前。这里假定这两个地址处于同一个

VMA 中。

```

1460 int make_pages_present(unsigned long addr, unsigned long end)
1461 {
1462     int ret, len, write;
1463     struct vm_area_struct * vma;
1464
1465     vma = find_vma(current->mm, addr);
1466     write = (vma->vm_flags & VM_WRITE) != 0;
1467     if (addr >= end)
1468         BUG();
1469     if (end > vma->vm_end)
1470         BUG();
1471     len = (end + PAGE_SIZE-1)/PAGE_SIZE-addr/PAGE_SIZE;
1472     ret = get_user_pages(current, current->mm, addr,
1473     len, write, 0, NULL, NULL);
1474     return ret == len ? 0 : -1;
1475 }
```

1465 利用 find_vma() (见 D. 3.1.1) 找到包含首地址的 VMA。

1466 记录写访问在 write 中是否允许。

1467~1468 如果起始地址在结束地址之后, 则产生 BUG()。

1469~1470 如果范围超过一个 VMA, 则这是一个 bug。

1471 计算陷入的区域长度。

1472 在请求区域所有页面中调用 get_user_pages() 陷入。这里返回陷入的页面数。

D. 2.4.5 函数: get_user_pages() (mm/memory.c)

这个函数用于陷入用户页面, 并可能用于属于其他进程的页面, 例如, 在 ptrace() 要用到。

```

454 int get_user_pages(struct task_struct * tsk, struct mm_struct * mm,
455                     unsigned long start,
456                     int len, int write, int force, struct page
457                     ** pages, struct vm_area_struct ** vmas)
458 {
459     int i;
460     unsigned int flags;
461     /* Require read or write permissions.
462     * If 'force' is set, we only require the "MAY" flags.
```

```

463 */
464 flags = write ? (VM_WRITE | VM_MAYWRITE) : (VM_READ | VM_MAYREAD);
465 flags &= force ? (VM_MAYREAD | VM_MAYWRITE) : (VM_READ | VM_WRITE);
466 i = 0;
467

```

454 参数如下：

- tsk 是陷入页面的进程。
- mm 是管理待陷入地址空间的 mm_struct。
- start 是开始陷入的位置。
- len 是以页计数的陷入区域长度。
- write 是表明是否可以写待陷入页面。
- force 是表明即使区域有 VM_MAYREAD 和 VM_MAYWRITE 标志，页面也应该陷入。
- pages 是一个结构页面数组，它可能为 NULL。如果指定了这个数组，则它可以填充待陷入的 struct pages。
- vmas 是与 pages 数组类似，如果指定了这个数组，它可以填充受到陷入影响的 VMA。

464 如果 write 设置为 1，则设置所需的标志为 VM_WRITE 和 VM_MAYWRITE 标志，否则利用相应的读标志。

465 如果指定了 force 值，这里仅需要 MAY 标志。

```

468 do {
469     struct vm_area_struct * vma;
470
471     vma = find_extend_vma(mm, start);
472
473     if (! vma ||
474         (pages && vma->vm_flags & VM_IO) ||
475         !(flags & vma->vm_flags))
476         return i ? : -EFAULT;
477
478     spin_lock(&mm->page_table_lock);
479     do {
480         struct page * map;
481         while (! (map = follow_page(mm, start, write)))
482             spin_unlock(&mm->page_table_lock);
483         switch (handle_mm_fault(mm, vma, start, write)) {

```

```

482 case 1:
483 tsk->min_flt++;
484 break;
485 case 2:
486 tsk->maj_flt++;
487 break;
488 case 0:
489 if (i) return i;
490 return -EFAULT;
491 default:
492 if (i) return i;
493 return -ENOMEM;
494 }
495 spin_lock(&mm->page_table_lock);
496 }
497 if (pages) {
498 pages[i] = get_page_map(map);
499 /* FIXME: call the correct function,
500 * depending on the type of the found page
501 */
502 if (!pages[i])
503 goto bad_page;
504 page_cache_get(pages[i]);
505 }
506 if (vmas)
507 vmas[i] = vma;
508 i++;
509 start += PAGE_SIZE;
510 len--;
511 } while(len && start < vma->vm_end);
512 spin_unlock(&mm->page_table_lock);
513 } while(len);
514 out:
515 return i;

```

468~513 外层循环将遍历受陷入影响的 VMA。

471 找到 start 当前值影响的 VMA。这个参数在第 PAGE_SIZE 步加 1。

473 如果这个地址上的 VMA 不存在,或者调用者已经为一块 I/O 映射的区域指定了一个 struct page (因此不是由后援内存支持的),或者 VMA 没有所需的标志位,则返

回-EFAULT。

476 上锁页表自旋锁。

479~496 follow_page(见 C. 2.1 小节)遍历页表并返回表示映射在 start 处页面帧的 struct page。这里的循环仅在没有 PTE 时才进入,并且将一直循环直到找到一个拥有页表自旋锁的 PTE。

480 解锁页表自旋锁,因为 handle_mm_fault()将要睡眠。

481 如果没有页面,这里调用 handle_mm_fault()(见 D. 5.3.1)来陷入。

482~487 更新 task_struct 统计并且表明是否发生了主/次异常。

488~490 如果异常地址无效,在这里返回-EFAULT。

491~493 如果系统内存不足,返回-ENOMEM。

495 重新上锁页表。这个循环将检查一遍以保证页面都实际存在。

497~505 如果调用者请求,则获取受到函数影响的 struct pages 的 pages 数组。每个结构都通过 page_cache_get()来引用这个数组。

506~507 相似地,这里记录受影响的 VMA。

508 把 i 加 1, i 是在请求区域的页面数量。

509 在页面划分的那一步把 strat 加 1。

510 将必须陷入的页面数减 1。

511 不断遍历 VMA 直到请求页被陷入。

512 释放页表自旋锁。

515 返回在区域中存在的页面数。

516

```
517 /*
518 * We found an invalid page in the VMA. Release all we have
519 * so far and fail.
520 */
```

521 bad_page:

```
522 spin_unlock(&mm->page_table_lock);
523 while (i--)
524 page_cache_release(pages[i]);
525 i = -EFAULT;
526 goto out;
527 }
```

521 只有找到了一个表明不存在页面帧的 struct page 才可能到这里。

523~524 如果找到了这样一个 struct page,则释放存放在 pages 数组中的所有页面。

525~526 返回-EFAULT。

D. 2. 4. 6 函数: move_page_tables() (mm/mremap.c)

这个函数的调用图如图 4.8 所示。这个函数负责从 old_addr 指向的区域把所有的页表项复制到 new_addr 所指向的区域。它只是每次复制一个表项。在这个函数完成时, 返回所有在旧区域的表项。还没有更有效的方式来完成这个操作, 但这里却很容易从错误中恢复。

```

90 static int move_page_tables(struct mm_struct * mm,
91 unsigned long new_addr, unsigned long old_addr,
92     unsigned long len)
93 {
94     unsigned long offset = len;
95
96     flush_cache_range(mm, old_addr, old_addr + len);
97
98     while (offset) {
99         offset -= PAGE_SIZE;
100        if (move_one_page(mm, old_addr + offset, new_addr +
101             offset))
102            goto oops_we_failed;
103    }
104
105    flush_tlb_range(mm, old_addr, old_addr + len);
106
107    return 0;
108
109
110    oops_we_failed:
111    flush_cache_range(mm, new_addr, new_addr + len);
112    while ((offset += PAGE_SIZE) < len)
113        move_one_page(mm, new_addr + offset, old_addr + offset);
114    zap_page_range(mm, new_addr, len);
115
116    return -1;
117 }
```

90 参数分别为进程的 mm、新地址、旧地址以及要移动表项的区域长度。

95 flush_cache_range() 将刷新在这个区域的所有 CPU 高速缓存。必须首先调用这个函数, 因为在某些体系结构中, 比如著名的 Sparc 中, 在刷新 TLB 前需要存在一个虚拟到物理地址的映射。

102~106 遍历该区域中的每个页面, 并调用 move_one_pte() (见 D. 2. 4. 7) 来移除 PTE。这里将转换大量遍历的页表并完成得更好, 但这是个很少使用的操作。

107 刷新旧区域的 TLB。

108 返回成功。

118~120 这一块将所有的 PTE 移回去。在这里并不需要 flush_tlb_range(), 因为这块区域还没有使用过, 所以 TLB 表项应该还不存在。

121 销毁那些多分配的页面。

122 返回失败。

D. 2.4.7 函数: move_one_page() (mm/mremap.c)

这个函数负责在调用 get_one_pte() 找到正确的 PTE 和调用 copy_one_pte() 来复制 PTE 之前获取一个自旋锁。

```

77 static int move_one_page(struct mm_struct *mm,
    unsigned long old_addr, unsigned long new_addr)
78 {
79     int error = 0;
80     pte_t * src;
81
82     spin_lock(&mm->page_table_lock);
83     src = get_one_pte(mm, old_addr);
84     if (src)
85         error = copy_one_pte(mm, src, alloc_one_pte(mm, new_addr));
86     spin_unlock(&mm->page_table_lock);
87     return error;
88 }
```

82 获取 mm 锁。

83 调用 get_one_pte() (见 D. 2.4.8), 它遍历页表来获得正确的 PTE。

84~85 如果存在 PTE, 则在这里为复制目的地分配一个 PTE, 并调用 copy_one_pte() (见 D. 2.4.10) 复制到 PTE。

86 释放 mm 锁。

87 返回 copy_one_pte() 的返回值。如果在 85 行的 alloc_one_pte() (见 D. 2.4.9) 失败就返回错误。

D. 2.4.8 函数: get_one_pte() (mm/mremap.c)

这是一个非常简单的遍历页表函数。

```

18 static inline pte_t * get_one_pte(struct mm_struct *mm,
    unsigned long addr)
19 {
20     pgd_t * pgd;
21     pmd_t * pmd;
```

```

22 pte_t * pte = NULL;
23
24 pgd = pgd_offset(mm, addr);
25 if (pgd_none(*pgd))
26     goto end;
27 if (pgd_bad(*pgd)) {
28     pgd_ERROR(*pgd);
29     pgd_clear(pgd);
30     goto end;
31 }
32
33 pmd = pmd_offset(pgd, addr);
34 if (pmd_none(*pmd))
35     goto end;
36 if (pmd_bad(*pmd)) {
37     pmd_ERROR(*pmd);
38     pmd_clear(pmd);
39     goto end;
40 }
41
42 pte = pte_offset(pmd, addr);
43 if (pte_none(*pte))
44     pte = NULL;
45 end;
46 return pte;
47 }

```

24 获取该地址的 PGD。

25~26 如果不存在 PGD，则返回 NULL，因为也不会存在 PTE。

27~31 如果 PGD 被损坏,这里标记在该区域发生了错误,并清除 PGD 的内容,然后返回 NULL。

33~40 与 PGD 的处理一样, 返回正确的 PMD。

42 获取 PTE,如果 PTE 存在,则可以返回。

D. 2.4.9 函数: alloc_one_pte() (mm/mremap.c)

这个函数在需要的情况下在区域中分配一个 PTE。

```
49 static inline pte_t * alloc_one_pte(struct mm_struct * mm,
           unsigned long addr)
50 {
```

```

51 pmd_t * pmd;
52 pte_t * pte = NULL;
53
54 pmd = pmd_alloc(mm, pgd_offset(mm, addr), addr);
55 if (pmd)
56 pte = pte_alloc(mm, pmd, addr);
57 return pte;
58 }

```

54 如果不存在 PMD, 这里分配一个。

55~56 如果存在 PMD, 这里分配一个 PTE 项。然后检查在函数 `copy_one_pte()` 中的操作是否成功。

D. 2.4.10 函数: `copy_one_pte()` (`mm/mremap.c`)

这个函数将一个 PTE 的内容复制到另外一个 PTE。

```

60 static inline int copy_one_pte(struct mm_struct *mm,
61                               pte_t * src, pte_t * dst)
62 {
63     int error = 0;
64     pte_t pte;
65
66     if (!pte_none(*src)) {
67         pte = ptep_get_and_clear(src);
68     }
69     if (!dst) {
70         /* No dest? We must put it back. */
71         dst = src;
72         error++;
73     }
74     set_pte(dst, pte);
75 }

```

65 如果源 PTE 不存在, 这里仅返回 0, 提示复制成功。

66 获取 PTE 并将其从旧地址移除。

67~71 如果 dst 不存在, 则意味着对 `alloc_one_pte()` 的调用失败, 复制也将失败, 必须中止操作。

72 把 PTE 移到新地址。

74 如果出错则返回错误信息。

D. 2.5 删除内存区域

D. 2.5.1 函数: do_munmap() (mm/mmap.c)

这个函数的调用图如图 4.10 所示。这个函数负责取消一个区域的映射。如果需要，这个解除映射操作将会产生多个 VMA。而且，如果需要，它可以只取消一部分。所以，完整的解除映射操作分为两个主要操作。这个函数负责找到受影响的 VMA，而 `unmap_fixup()` 负责修整其余的 VMA。

这个函数分为许多小的部分，我们分别处理。粗略而言，这些小部分如下：

- 前序函数,找到开始操作的 VMA。
 - 把受解除映射操作影响的 VMA 移除出 mm,并把它们放到一个由变量 free 为头的链表中。
 - 循环遍历以 free 为头的链表,取消在区域中待解除映射的页面映射。并调用 unmap_fixup()来修整映射。
 - 验证与解除映射操作有关的 mm 和空闲内存。

```
924 int do_munmap(struct mm_struct * mm, unsigned long addr,
925                 size_t len)
926 {
927     struct vm_area_struct * mpnt, * prev, ** npp, * free, * extra;
928     if ((addr & ~PAGE_MASK) || addr > TASK_SIZE ||
929         len > TASK_SIZE-addr)
930     return -EINVAL;
931     if ((len = PAGE_ALIGN(len)) == 0)
932     return -EINVAL;
933
934     mpnt = find_vma_prev(mm, addr, &prev);
935     if (! mpnt)
936     return 0;
937     /* we have addr < mpnt->vm_end */
938
939     if (mpnt->vm_start >= addr + len)
940     return 0;
941
942     if ((mpnt->vm_start < addr && mpnt->vm_end > addr + len)
```

```

949 && mm->map_count >= max_map_count)
950 return -ENOMEM;
951
956 extra = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
957 if (! extra)
958 return -ENOMEM;

```

924 参数如下：

- mm 是进行 unmap 操作的进程的 mm。
- addr 是 unmap 的区域的起始地址。
- len 是区域长度。

928~929 保证地址都页对齐，并且待解除映射的区域不在内核虚拟地址空间。

931~932 保证要解除映射的区域是页对齐的。

939 找到包含起始地址的 VMA 以及它前面的 VMA，这样在后面可以很容易地从链表中移除 VMA。

941~941 如果没有返回 mpnt，则意味着地址必须是前面最近使用过的 VMA，所以，地址空间还没有使用过，返回就可以了。

944~945 如果返回的 VMA 从你试着解除映射的区域开始，这个区域也没有使用过，则只返回。

948~950 第 1 部分检查 VMA 是否被部分解除映射。如果是，将在后面创建另外一个 VMA 来处理被拆开的区域，所以必须检查 map_count 以保证这个计数不会过大。

956~958 一旦获得了新的映射，现在就开始分配。因为如果将来分配的话，在发生错误时很难恢复过来。

```

960 npp = (prev ? &prev->vm_next : &mm->mmap);
961 free = NULL;
962 spin_lock(&mm->page_table_lock);
963 for ( ; mpnt && mpnt->vm_start < addr + len; mpnt = * npp) {
964 * npp = mpnt->vm_next;
965 mpnt->vm_next = free;
966 free = mpnt;
967 rb_erase(&mpnt->vm_rb, &mm->mm_rb);
968 }
969 mm->mmap_cache = NULL; /* Kill the cache. */
970 spin_unlock(&mm->page_table_lock);

```

这部分把受解除映射操作影响的 VMA 从 mm 移除，并把它们放到一个由变量 free 为头的链表中。这样可以使修整这片区域更简单。

960 npp 在循环中变成链表中的下一个 VMA。初始化时,它或者是当前的 VMA(mpnt)或者变成了链表的第一个 VMA。

961 free 是受到解除映射操作的 VMA 的链表头节点。

962 上锁 mm。

963 循环遍历链表,直到当前 VMA 是过去待解除映射区域的末端。

964 npp 变成了链表的下一个 VMA。

965~966 将当前 VMA 从 mm 中的线性链表中移除,并把它放到一个由 free 为头的链表中。当前 mpnt 变成空闲链表的头节点。

967 从红黑树中删除 mpnt。

969 移除缓存的结果,因为以前查询的结果是待解除映射的区域。

970 释放 mm。

971

```

972 /* Ok - we have the memory areas we should free on the
973 * 'free' list, so release them, and unmap the page range..
974 * If one of the segments is only being partially unmapped,
975 * it will put new vm_area_struct(s) into the address space.
976 * In that case we have to be careful with VM_DENYWRITE.
977 */
978 while ((mpnt = free) != NULL) {
979     unsigned long st, end, size;
980     struct file *file = NULL;
981
982     free = free->vm_next;
983
984     st = addr < mpnt->vm_start ? mpnt->vm_start : addr;
985     end = addr + len;
986     end = end > mpnt->vm_end ? mpnt->vm_end : end;
987     size = end - st;
988
989     if (mpnt->vm_flags & VM_DENYWRITE &&
990         (st != mpnt->vm_start || end != mpnt->vm_end) &&
991         (file = mpnt->vm_file) != NULL) {
992         atomic_dec(&file->f_dentry->d_inode->i_writecount);
993     }
994     remove_shared_vm_struct(mpnt);
995     mm->map_count--;
996

```



```

997 zap_page_range(mm, st, size);
998
999 /*
1000 * Fix the mapping, and free the old area
1001 * if it wasn't reused.
1001 */
1002 extra = unmap_fixup(mm, mpnt, st, size, extra);
1003 if (file)
1004 atomic_inc(&file->f_dentry->d_inode->i_writecount);
1005 }

```

978 遍历链表直到没有 VMA 剩下。

982 将 free 指向链表中的下一个元素, 剩下 mpnt 作为将要被移除的元素头。

984 st 是待解除映射区域的起点。如果 addr 在 VMA 的起始位置, 起点就是 mpnt->vm_struct, 否则, 起点就是指定的地址。

985~986 以相似的方式计算要映射的区域末端。

987 计算在这一遍中要解除映射的区域大小。

989~993 如果指定了 VM_DENYWRITE 标志, 这个解除映射操作将创建一个空洞, 而且将映射一个文件。然后, 将 i_writecounts 减 1。在这个字段为负时, 它记录着保护这个文件没有以写方式打开的用户数。

994 移除文件映射。如果文件还是被部分地映射, 就在 unmap_fixup() (见 D. 2.5.2) 时再次获得。

995 减少 map 计数。

997 移除该区域中的所有页面。

1002 在删除后调用 unmap_fixup(见 D. 2.5.2) 来修整区域。

1003~1004 将文件的写计数减 1, 因为该区域已经被解除映射。如果它仅是被部分地解除映射, 这个调用将权衡在行 987 而减 1。

```

1006 validate_mm(mm);
1007
1008 /* Release the extra vma struct if it wasn't used */
1009 if (extra)
1010 kmem_cache_free(vm_area_cachep, extra);
1011
1012 free_ptables(mm, prev, addr, addr + len);
1013
1014 return 0;
1015 }

```

1006 validate_mm()是一个调试函数。如果它可用,则它会保证该 mm 的 VMA 树也有效。

1009~1010 如果不需要额外的 VMA,则在这里删除它。

1012 释放所有在解除映射区域的页表。

1014 返回成功。

D. 2.5.2 函数: unmap_fixup() (mm/mmap.c)

这个函数在取消一块映射后修整区域。它传入受到解除映射操作的 VMA 链表,解除映射的区域和长度以及一个空的 VMA。如果创建了整个区域,这个 VMA 可能在修整区域时用到。这个函数主要处理四种情况:解除映射的区域;从开始到中间某个地方的局部解除映射的区域;从中间的某个地方到末尾的局部解除映射的区域;在区域中间创建一个空洞。我们分别处理这四种情况。

```
787 static struct vm_area_struct * unmap_fixup(struct mm_struct * mm,
788 struct vm_area_struct * area, unsigned long addr, size_t len,
789 struct vm_area_struct * extra)
790 {
791 struct vm_area_struct * mpnt;
792 unsigned long end = addr + len;
793
794 area->vm_mm->total_vm -= len >> PAGE_SHIFT;
795 if (area->vm_flags & VM_LOCKED)
796 area->vm_mm->locked_vm -= len >> PAGE_SHIFT;
797
```

这是函数的前面部分。

787 函数的参数如下:

- mm 是解除了映射区域所属的 mm。
- area 是受到解除映射操作影响的 VMA 的链表头。
- addr 是解除映射区域的首地址。
- len 是解除映射区域的长度。
- extra 是在区域中间建立空洞时传入的空 VMA。

792 计算被解除映射区域的末地址。

794 减少进程用到的页面计数。

795~796 如果页面在内存中被锁住,这里将减少锁住页面的计数。

```
798 /* Unmapping the whole area. */
799 if (addr == area->vm_start && end == area->vm_end) {
```

```

800 if (area->vm_ops && area->vm_ops->close)
801 area->vm_ops->close(area);
802 if (area->vm_file)
803 fput(area->vm_file);
804 kmem_cache_free(vm_area_cachep, area);
805 return extra;
806 }

```

这是第一个,也是最简单的情况,这里整个区域都被解除映射。

799 如果 `addr` 是 VMA 的首部,且 `end` 是 VMA 的尾部,整个区域都被解除映射。这里很有趣,因为,如果解除映射的是一个生成区域,就有可能 `end` 已经超出了 VMA 末端,但是这个 VMA 仍然全部被解除了映射。

800~801 如果 VMA 提供了结束操作,则在这里调用。

802~803 如果映射了一个文件或设备,则在这里调用 `fput()`,它决定使用计数,并在引用计数减为 0 时释放映射。

804 将 VMA 释放回 slab 分配器。

805 返回额外的 VMA,因为没有使用过它们。

```

809 if (end == area->vm_end) {
810 /*
811 * here area isn't visible to the semaphore-less readers
812 * so we don't need to update it under the spinlock.
813 */
814 area->vm_end = addr;
815 lock_vma_mappings(area);
816 spin_lock(&mm->page_table_lock);
817 }

```

这一块处理在区域中间到末尾部分被映射的情况。

814 将 VMA 截断为 `addr`。在这里,区域的页面已经被释放了,后面将会释放页表表项,所以不再需要做额外的工作。

815 如果一个文件/设备已经映射,它就由函数 `lock_vma_mappings()` 来上锁保护共享访问。

816 上锁 `mm`。在函数后面,VMA 的其他部分将被插入到 `mm` 中。

```

817 else if (addr == area->vm_start) {
818 area->vm_pgoff += (end - area->vm_start) >> PAGE_SHIFT;
819 /* same locking considerations of the above case */
820 area->vm_start = end;

```

```

821 lock_vma_mappings(area);
822 spin_lock(&mm->page_table_lock);
823 } else {

```

这一块处理在区域开始到中间的文件/设备被映射的情况。

818 增加文件/设备的偏移,这些偏移由这次解除映射的页面数表示。

820 将 VMA 的首部移到被解除映射区域的末尾。

821~822 上锁前面描述的文件/设备和 mm。

```

823 } else {
825 /* Add end mapping - leave beginning for below */
826 mpnt = extra;
827 extra = NULL;
828
829 mpnt->vm_mm = area->vm_mm;
830 mpnt->vm_start = end;
831 mpnt->vm_end = area->vm_end;
832 mpnt->vm_page_prot = area->vm_page_prot;
833 mpnt->vm_flags = area->vm_flags;
834 mpnt->vm_raend = 0;
835 mpnt->vm_ops = area->vm_ops;
836 mpnt->vm_pgoff = area->vm_pgoff +
    ((end - area->vm_start) >> PAGE_SHIFT);
837 mpnt->vm_file = area->vm_file;
838 mpnt->vm_private_data = area->vm_private_data;
839 if (mpnt->vm_file)
840     get_file(mpnt->vm_file);
841 if (mpnt->vm_ops && mpnt->vm_ops->open)
842     mpnt->vm_ops->open(mpnt);
843 area->vm_end = addr; /* Truncate area */
844
845 /* Because mpnt->vm_file == area->vm_file this locks
846 * things correctly.
847 */
848 lock_vma_mappings(area);
849 spin_lock(&mm->page_table_lock);
850 __insert_vm_struct(mm, mpnt);
851 }

```

这一块处理在部分解除映射的区域中创建空洞的情况。在这种情形下,需要一个额外的

VMA 来创建一个从解除映射的区域末尾到旧 VMA 末尾的新映射。

826~827 保存额外的 VMA，并使 VMA 为 NULL，这样调用函数知道 VMA 已经在使用且不能释放这个 VMA。

828~838 复制所有的 VMA 信息。

839 如果一个文件/设备已经被映射，这里通过 get_file() 获得一个引用。

841~842 如果提供了一个打开函数，则在这里调用它。

843 截断 VMA 从而使其在将被解除映射的区域首部截止。

848~849 与前两种情况相同，锁住文件和 mm。

850 将额外的 VMA 插入到 mm 中。

852

```
853 __insert_vm_struct(mm, area);
854 spin_unlock(&mm->page_table_lock);
855 unlock_vma_mappings(area);
856 return extra;
857 }
```

853 重新插入 VMA 到 mm 中。

854 解锁链表。

855 解锁锁住共享映射的自旋锁。

856 如果额外 VMA 没有再使用，则返回。如果为 NULL 则返回 NULL。

D.2.6 删除所有的内存区域。

D.2.6.1 函数：exit_mmap() (mm/mmap.c)

这个函数仅遍历与给定 mm 相关的所有 VMA，并取消对它们的映射。

```
1127 void exit_mmap(struct mm_struct * mm)
1128 {
1129     struct vm_area_struct * mpnt;
1130
1131     release_segments(mm);
1132     spin_lock(&mm->page_table_lock);
1133     mpnt = mm->mmap;
1134     mm->mmap = mm->mmap_cache = NULL;
1135     mm->mm_rb = RB_ROOT;
1136     mm->rss = 0;
1137     spin_unlock(&mm->page_table_lock);
```

```

1138 mm->total_vm = 0;
1139 mm->locked_vm = 0;
1140
1141 flush_cache_mm(mm);
1142 while (mpnt) {
1143     struct vm_area_struct * next = mpnt->vm_next;
1144     unsigned long start = mpnt->vm_start;
1145     unsigned long end = mpnt->vm_end;
1146     unsigned long size = end - start;
1147
1148     if (mpnt->vm_ops) {
1149         if (mpnt->vm_ops->close)
1150             mpnt->vm_ops->close(mpnt);
1151     }
1152     mm->map_count--;
1153     remove_shared_vm_struct(mpnt);
1154     zap_page_range(mm, start, size);
1155     if (mpnt->vm_file)
1156         fput(mpnt->vm_file);
1157     kmem_cache_free(vm_area_cachep, mpnt);
1158     mpnt = next;
1159 }
1160 flush_tlb_mm(mm);
1161
1162 /* This is just debugging */
1163 if (mm->map_count)
1164     BUG();
1165
1166 clear_page_tables(mm, FIRST_USER_PGD_NR, USER_PTRS_PER_PGD);
1167 }

```

1131 如果体系结构支持段并且进程使用了内存段, `realase_segment()` 将释放在进程局部描述表(LDT)上与进程相关的内存段。一些应用,如著名的 WINE,就使用了这个特性。

1132 上锁 mm。

1133 mpnt 变成链表上的第一个 VMA。

1134 从 mm 中释放 VMA 相关的信息,这样就可以解锁 mm。

1137 解锁 mm。

1138~1139 清除 mm 统计信息。

1141 刷新 CPU 地址范围。

- 1142~1159 遍历每个与 mm 相关的 VMA。
 1143 记录下一个要清除的 VMA, 这个可以被删除。
 1144~1146 记录待删除区域的起始、末尾和大小。
 1148~1151 如果有一个与这个 VMA 有关的结束操作, 则在这里调用它。
 1152 减少 map 计数。
 1153 从共享映射链表中清除文件/设备映射。
 1154 释放与该区域有关的所有页面。
 1155~1156 如果这个区域中有一个文件/设备映射, 则在这里释放它。
 1157 释放 VMA 结构。
 1158 转到下一个 VMA。
 1160 刷新整个 mm 的 TLB, 因为它将被解除映射。
 1163~1164 如果 map_count 是正的, 则意味着 map 计数没有正确计数, 在这里调用 BUG() 来进行标记。
 1166 调用 clear_page_tables() (见 D. 2.6.2) 来清除与这片区域相关的页表。

D. 2.6.2 函数: clear_page_tables() (mm/memory.c)

这是一个用于取消所有 PTE 映射和释放区域中页面的高层函数。它在需要销毁页表时使用, 如在进程退回或区域被解除映射时。

```

146 void clear_page_tables(struct mm_struct * mm,
                           unsigned long first, int nr)
147 {
148 pgd_t * page_dir = mm->pgd;
149
150 spin_lock(&mm->page_table_lock);
151 page_dir += first;
152 do {
153 free_one_pgd(page_dir);
154 page_dir++;
155 } while (--nr);
156 spin_unlock(&mm->page_table_lock);
157
158 /* keep the pagetable cache within bounds */
159 check_pgt_cache();
160 }
```

148 获取被解除映射 mm 的 PGD。

150 上锁页表。

151~155 在指定范围内遍历所有 PGD。对找到的每一个 PGD，这里调用 free_one_pgd()（见 D. 2. 6. 3）。

156 解锁页表。

159 检测可用 PGD 结构的高速缓存。如果在 PGD 快表中有太多的 PGD，这里将回收一些。

D. 2. 6. 3 函数：free_one_pgd() (mm/memory.c)

整个函数销毁一个 PGD。对 PGD 中每一个 PMD，调用 free_one_pmd()。

```

109 static inline void free_one_pgd(pgd_t * dir)
110 {
111     int j;
112     pmd_t * pmd;
113
114     if (pgd_none(*dir))
115         return;
116     if (pgd_bad(*dir)) {
117         pgd_ERROR(*dir);
118         pgd_clear(dir);
119         return;
120     }
121     pmd = pmd_offset(dir, 0);
122     pgd_clear(dir);
123     for (j = 0; j < PTRS_PER_PMD; j++) {
124         prefetchw(pmd + j + (PREFETCH_STRIDE/16));
125         free_one_pmd(pmd + j);
126     }
127     pmd_free(pmd);
128 }
```

114~115 如果在这里不存在 PGD，则返回。

116~120 如果 PGD 已经被损坏，设置错误标志并返回。

121 获取 PGD 中的第一个 PMD。

122 清除 PGD 表项。

123~126 对 PGD 中每个 PMD，在这里调用 free_one_pmd()（见 D. 2. 6. 4）。

127 将 PMD 页释放到 PMD 快表中，然后，调用 check_pgt_cache()。如果在高速缓存中有太多的 PMD 页面，它们将被回收。

D. 2. 6. 4 函数: free_one_pmd() (mm/memory.c)

```
93 static inline void free_one_pmd(pmd_t * dir)
94 {
95     pte_t * pte;
96
97     if (pmd_none(* dir))
98         return;
99     if (pmd_bad(* dir)) {
100         pmd_ERROR(* dir);
101         pmd_clear(dir);
102         return;
103     }
104     pte = pte_offset(dir, 0);
105     pmd_clear(dir);
106     pte_free(pte);
107 }
```

97~98 如果在这里不存在 PMD，则返回。

99~103 如果 PMD 已经被损坏,则设置错误标志并返回。

104 获得 PMD 中第一个 PTE。

105 从页表中清除 PMD。

106 调用 `pte_free()` 将 PTE 页面释放到 PTE 快表高速缓存。然后，调用 `check_pgt_cache()`，如果在高速缓存中有太多的 PTE 页面，则回收它们。

D.3 查找内存区域

在这部分的函数在虚拟地址空间查找已映射区域和空闲区域。

D.3.1 查找已映射内存区域

D. 3.1.1 函数: find_vma() (mm/mmap.c)

```

663 struct vm_area_struct * vma = NULL;
664
665 if (mm) {
666 /* Check the cache first. */
667 /* (Cache hit rate is typically around 35 %. ) */
668 vma = mm->mmap_cache;
669 if (! (vma && vma->vm_end > addr &&
670 vma->vm_start <= addr)) {
671 rb_node_t * rb_node;
672 rb_node = mm->mm_rb.rb_node;
673 vma = NULL;
674
675 while (rb_node) {
676 struct vm_area_struct * vma_tmp;
677
678 vma_tmp = rb_entry(rb_node,
679                     struct vm_area_struct, vm_rb);
680 if (vma_tmp->vm_end > addr) {
681 vma = vma_tmp;
682 if (vma_tmp->vm_start <= addr)
683 break;
684 rb_node = rb_node->rb_left;
685 } else
686 rb_node = rb_node->rb_right;
687 }
688 if (vma)
689 mm->mmap_cache = vma;
690 }
691 }
692 return vma;
693 }

```

661 其中两个参数是高层待查找的 mm_struct 以及调用者感兴趣的地址。

663 地址没有找到时,缺省返回 NULL。

665 保证调用者不会尝试找无效的 mm。

668 mmap_cache 具有上次调用 find_vma() 的结果。这里可能不需要遍历整个红黑树。

669 如果这是被检查的有效 VMA,这里检查被查找的地址是否包含在其中。如果是,

VMA 就是 mmap_cache 的那个地址, 所以可以返回它, 否则搜索这棵树。

670~674 从树根开始。

675~687 这部分遍历树。

678 这个宏正如其名, 返回这个树节点指向的 VMA。

680 检查下一个要遍历的节点是在左树还是右树叶子。

682 如果当前 VMA 就是所请求的 VMA, 这里退出 while 循环。

689 如果 VMA 有效, 这里设置 mmap_cache, 这样可以找到 find_vma() 的下一个调用。

692 返回包含地址的 VMA, 作为遍历树的副作用, 返回与所请求地址最接近的 VMA。

D.3.1.2 函数: find_vma_prev() (mm/mmap.c)

```

696 struct vm_area_struct * find_vma_prev(struct mm_struct * mm,
697                                         unsigned long addr,
698                                         struct vm_area_struct ** pprev)
699 {
700 /* Go through the RB tree quickly. */
701 struct vm_area_struct * vma;
702 rb_node_t * rb_node, * rb_last_right, * rb_prev;
703
704 rb_node = mm->mm_rb.rb_node;
705 rb_last_right = rb_prev = NULL;
706 vma = NULL;
707
708 while (rb_node) {
709 struct vm_area_struct * vma_tmp;
710
711 vma_tmp = rb_entry(rb_node,
712                     struct vm_area_struct, vm_rb);
713 if (vma_tmp->vm_end > addr) {
714 vma = vma_tmp;
715 rb_prev = rb_last_right;
716 if (vma_tmp->vm_start <= addr)
717 break;
718 rb_node = rb_node->rb_left;
719 } else {
720 rb_last_right = rb_node;
721 rb_node = rb_node->rb_right;

```

```

722 !
723 }
724 if (vma) {
725 if (vma->vm_rb.rb_left) {
726 rb_prev = vma->vm_rb.rb_left;
727 while (rb_prev->rb_right)
728 rb_prev = rb_prev->rb_right;
729 }
730 *pprev = NULL;
731 if (rb_prev)
732 *pprev = rb_entry(rb_prev, struct
733 vm_area_struct, vm_rb);
734 if ((rb_prev ? (*pprev)->vm_next : mm->mmmap) !=
735 vma)
736 BUG();
737 }
738 *pprev = NULL;
739 return NULL;
740 }

```

696~723 实际上这与所描述的 `find_vma()` 函数相同。仅有的区别是要记住上次访问的节点,因为这里表示请求 VMA 前面的 VMA。

725~729 如果 VMA 有一个左节点,则意味着必须遍历它。首先是左叶子,然后沿着每个右叶子直至找到树底。

731~732 从红黑树节点中分离 VMA。

733~734 一个调试用的检查。如果这里是前面的节点,它的下一个字段将指向返回的 VMA。如果没有,则是一个 bug。

D.3.1.3 函数: `find_vma_intersection()` (include/linux/mm.h)

```

673 static inline struct vm_area_struct * find_vma_intersection(
674     struct mm_struct * mm,
675     unsigned long start_addr, unsigned long end_addr)
676 {
677     struct vm_area_struct * vma = find_vma(mm, start_addr);
678     if (vma && end_addr <= vma->vm_start)
679         vma = NULL;

```

```
679 return vma;
680 }
```

675 返回与起始地址最近的 VMA。

677 如果返回了一个 VMA 并且结束地址还小于返回 VMA 的起始地址，则 VMA 没有交叉。

679 如果没有交叉则返回这个 VMA。

D.3.2 查找空闲内存区域

D.3.2.1 函数：get_unmapped_area() (mm/mmap.c)

这个函数的调用图如图 4.4 所示。

```
644 unsigned long get_unmapped_area(struct file * file,
645         unsigned long addr,
646         unsigned long len,
647         unsigned long pgoff,
648         unsigned long flags)
649 {
650     if (flags & MAP_FIXED) {
651         if (addr > TASK_SIZE - len)
652             return -ENOMEM;
653     }
654     if (file && file->f_op && file->f_op->get_unmapped_area)
655         return file->f_op->get_unmapped_area(file, addr,
656                                         len, pgoff, flags);
657     return arch_get_unmapped_area(file, addr, len, pgoff, flags);
658 }
```

644 传入的参数如下：

- file 是映射的文件或设备。
- addr 是要映射到的地址。
- len 是映射的长度。

- pgoff 是在被映射文件中的偏移。

- flags 是保护标志位。

646~652 有效性检查。如果请求了放置映射在特定的位置,这里保证不会溢出地址空间,并且是页对齐的。

654 如果 struct file 提供了一个 get_unmapped_area() 函数,这里使用它。

657 使用 arch_get_unmapped_area() (见 D. 3. 2. 2 节)作为 get_unmapped_area() 函数的匿名版本。

D. 3. 2. 2 函数: arch_get_unmapped_area() (mm/mmap.c)

通过定义 HAVE_ARCH_UNMAPPED_AREA, 体系结构可以选择性地指定这个函数。如果体系结构不指定,就使用这个版本。

```

614 #ifndef HAVE_ARCH_UNMAPPED_AREA
615 static inline unsigned long arch_get_unmapped_area(
616     struct file *filp,
617     unsigned long addr, unsigned long len,
618     unsigned long pgoff, unsigned long flags)
619 {
620     struct vm_area_struct *vma;
621
622     if (addr) {
623         addr = PAGE_ALIGN(addr);
624         vma = find_vma(current->mm, addr);
625         if (TASK_SIZE - len >= addr &&
626             (! vma || addr + len <= vma->vm_start))
627             return addr;
628     }
629     addr = PAGE_ALIGN(TASK_UNMAPPED_BASE);
630
631     for (vma = find_vma(current->mm, addr); ; vma = vma->vm_next) {
632         /* At this point: (! vma || addr < vma->vm_end). */
633         if (TASK_SIZE - len < addr)
634             return -ENOMEM;
635         if (! vma || addr + len <= vma->vm_start)
636             return addr;
637     }
638     addr = vma->vm_end;

```

```

638 }
639 }
640 #else
641 extern unsigned long arch_get_unmapped_area(struct file * ,
642           unsigned long, unsigned long,
643           unsigned long, unsigned long);
642#endif

```

614 如果这里没有定义,则意味着体系结构没有提供它自己的 `arch_get_unmapped_area()`, 所以使用这个函数。

615 这里的参数与 `get_unmapped_area()`(见 D. 3. 2. 1)相同。

619~620 有效性检查保证所需的映射长度不会太长。

622~628 如果提供了一个地址,这里使用它作为映射地址。

623 保证地址是页对齐的。

624 `find_wma()`(见 D. 3. 1. 1)将返回与所请求地址最近的区域。

625~627 保证这里的映射不会与另外一个区域重叠。如果没有重叠,就返回它,因为可以安全地使用它。否则,忽略它。

629 `TASK_UNMAPPED_BASE` 是查询要使用的空闲区域的起点。

631-638 从 `TASK_UNMAPPED_BASE` 开始,线性查找 VMA 直到在其中找到足够大的区域来存放新映射。这实际上是首次适应查询。

641 如果提供了一个外部函数,还需要在这里声明。

D. 4 对内存区域上锁和解锁

这部分包含与上锁和解锁一块区域相关的函数。它们最繁杂的部分是在操作发生时需要怎样修整区域。

D. 4. 1 对内存区域上锁

D. 4. 1. 1 函数: `sys_mlock()` (`mm/mlock.c`)

这个函数的调用图如图 4. 9 所示。这就是用于上锁物理内存中一片内存区域的系统调用 `mlock()`。这个函数仅检查以保证没有超过进程和用户数极限,并且要上锁的这个区域是页对齐的。

```
195 asmlinkage long sys_mlock(unsigned long start, size_t len)
```

```

196 {
197     unsigned long locked;
198     unsigned long lock_limit;
199     int error = -ENOMEM;
200
201     down_write(&current->mm->mmap_sem);
202     len = PAGE_ALIGN(len + (start & ~PAGE_MASK));
203     start &= PAGE_MASK;
204
205     locked = len >> PAGE_SHIFT;
206     locked += current->mm->locked_vm;
207
208     lock_limit = current->rlim[RLIMIT_MEMLOCK].rlim_cur;
209     lock_limit >>= PAGE_SHIFT;
210
211     /* check against resource limits */
212     if (locked > lock_limit)
213         goto out;
214
215     /* we may lock at most half of physical memory... */
216     /* (this check is pretty bogus, but doesn't hurt) */
217     if (locked > num_physpages/2)
218         goto out;
219
220     error = do_mlock(start, len, 1);
221 out:
222     up_write(&current->mm->mmap_sem);
223     return error;
224 }

```

201 获得信号量。因为可能在这里睡眠，所以不可以使用自旋锁。

202 将长度向上取整到页面边界。

203 将地址向下取整到页面边界。

205 计算要上锁多少页面。

206 计算在这个进程中要上锁的页面总数。

208~209 计算要锁住的页面数极限。

212~213 不允许进程锁住过多的页面。

217~218 不允许进程映射多于物理内存半数的页面。

220 调用 do_mlock(见 D. 4. 1. 4)，它首先找到与要上锁区域最近的 VMA，然后调用

mlock_fixup()(见 D. 4. 3. 1)。

222 释放信号量。

223 从 do_mlock()返回成功或错误代码。

D. 4. 1. 2 函数: sys_mlockall() (mm/mlock.c)

这是系统调用 mlockall(), 它试着锁定在内存中调用进程的所有页面。如果指定了 MCL_CURRENT,所有的当前页面都会被上锁。如果指定了 MCL_FUTURE,所有的映射都会被锁定。这些标志位可以以或操作连接。这个函数保证标志位以及进程限制正确,然后调用 do_mlockall()。

```

266 asmlinkage long sys_mlockall(int flags)
267 {
268     unsigned long lock_limit;
269     int ret = -EINVAL;
270
271     down_write(&current->mm->mmap_sem);
272     if (!flags || (flags & ~(MCL_CURRENT | MCL_FUTURE)))
273         goto out;
274
275     lock_limit = current->rlim[RLIMIT_MEMLOCK].rlim_cur;
276     lock_limit >>= PAGE_SHIFT;
277
278     ret = -ENOMEM;
279     if (current->mm->total_vm > lock_limit)
280         goto out;
281
282     /* we may lock at most half of physical memory... */
283     /* (this check is pretty bogus, but doesn't hurt) */
284     if (current->mm->total_vm > num_physpages/2)
285         goto out;
286
287     ret = do_mlockall(flags);
288 out:
289     up_write(&current->mm->mmap_sem);
290     return ret;
291 }
```

269 缺省时,这里返回-EINVAL 表明无效的参数。

271 获取当前 mm_struct 信号量。

272~273 保证指定了一些有效的标志。如果没有,则跳转到 out 来解锁信号量并且返回 -EINVAL。

275~276 检查进程限制,确定可以上锁多少页面。

278 从这里开始,缺省的错误是-ENOMEM。

279~280 如果锁定的大小超过了设置的限定,则返回 out。

284~285 不允许进程锁定一半以上的物理内存。这是一个无效的检查,因为四个进程每个上锁 1/4 的物理内存时,将超过这个限制。但还是接受这个检查,因为仅有根进程允许锁定内存,所以不太可能发生这样的错误。

287 调用核心函数 do_mlockcall()(见 D. 4. 1. 3)。

289~290 解锁信号量并返回。

D. 4. 1. 3 函数: do_mlockall() (mm/mlock.c)

```

238 static int do_mlockall(int flags)
239 {
240     int error;
241     unsigned int def_flags;
242     struct vm_area_struct * vma;
243
244     if (! capable(CAP_IPC_LOCK))
245         return -EPERM;
246
247     def_flags = 0;
248     if (flags & MCL_FUTURE)
249         def_flags = VM_LOCKED;
250     current->mm->def_flags = def_flags;
251
252     error = 0;
253     for (vma = current->mm->mmap; vma; vma = vma->vm_next) {
254         unsigned int newflags;
255
256         newflags = vma->vm_flags | VM_LOCKED;
257         if (! (flags & MCL_CURRENT))
258             newflags &= ~VM_LOCKED;
259         error = mlock_fixup(vma, vma->vm_start, vma->vm_end,
260                             newflags);
261         if (error)
262             break;
263     }

```

```
263 return error;
264 }
```

244~245 调用进程必须是两种情况之一,是根进程或者具有 CAP_IPC_LOCK 能力。

248~250 MCL_FUTURE 标志位意思是将来的所有的页面都必须锁定,所以,如果设置了该位,VMA 的 def_flags 必须为 VM_LOCKED。

253~262 遍历所有的 VMA。

256 在当前的 VMA 标志中设置 VM_LOCKED 标志位。

257~258 如果没有设置 MCL_CURRENT 标志位而请求给所有当前页面上锁,则这里将清除 VM_LOCKED 标志位。在这里的逻辑是这样的:不需要上锁的代码可以使用这部分功能,只是没有标志位。

259 调用 mlock_fixup() (见 D. 4. 3. 1),它将调整区域来匹配上锁。

260~261 任何时候如果返回了非 0 值,就停止上锁。注意已经上锁的 VMA 不会被解锁。

263 返回成功或者错误代码。

D. 4. 1. 4 函数: do_mlock() (mm/mlock.c)

这个函数负责根据参数值来决定上锁或解锁一块区域。我们把它分成两个部分,第 1 部分是保证区域是页对齐的(虽然只有该函数的调用者做相同的事),然后找到将要调整的 VMA。第 2 部分接着设置合适的标志位,然后对每个受锁定影响的 VMA 调用 mlock_fixup()。

```
148 static int do_mlock(unsigned long start, size_t len, int on)
149 {
150     unsigned long nstart, end, tmp;
151     struct vm_area_struct * vma, * next;
152     int error;
153
154     if (on && ! capable(CAP_IPC_LOCK))
155         return -EPERM;
156     len = PAGE_ALIGN(len);
157     end = start + len;
158     if (end < start)
159         return -EINVAL;
160     if (end == start)
161         return 0;
162     vma = find_vma(current->mm, start);
163     if (! vma || vma->vm_start > start)
164         return -ENOMEM;
```

这一块对请求按页对齐并找到 VMA。

154 仅有根进程可以上锁页面。

156 将长度按页对齐。这是一个冗余操作,因为在父函数中长度已经是页对齐的。

157~159 计算上锁的区域末端,并保证这是一个有效的区域。如果不是,则返回-EINVAL。

160~161 如果上锁的区域大小为 0,则在这里返回。

162 找到那些受该上锁操作影响的 VMA。

163~164 如果该地址范围的 VMA 不存在,则返回-ENOMEM。

```

166 for (nstart = start ; ; ) {
167     unsigned int newflags;
168
169
171     newflags = vma->vm_flags | VM_LOCKED;
172     if (! on)
173         newflags &= ~VM_LOCKED;
174
175     if (vma->vm_end >= end) {
176         error = mlock_fixup(vma, nstart, end, newflags);
177         break;
178     }
179
180     tmp = vma->vm_end;
181     next = vma->vm_next;
182     error = mlock_fixup(vma, nstart, tmp, newflags);
183     if (error)
184         break;
185     nstart = tmp;
186     vma = next;
187     if (! vma || vma->vm_start != nstart) {
188         error = -ENOMEM;
189         break;
190     }
191 }
192 return error;
193 }
```

这一块遍历由这个上锁操作影响的 VMA 并对每个 VMA 调用 mlock_fixup()。

166~192 遍历所需 VMA 以上锁页面。

171 设置在 VMA 上的 VM_LOCKED 标志位。

172~173 如果有一个解锁,则移除这个标志位。

175~177 如果这个 VMA 是最后一个受到解锁影响的 VMA,则在这里调用 `mlock_fixup()` 并传入上锁的末地址,然后返回。

180~190 这是整个需要上锁的 VMA。为了上锁,将这个 VMA 的末端作为一个参数传入 `mlock_fixup()`(见 D. 4. 3. 1),而不是实际上锁的末尾。

180 `tmp` 是在这个 VMA 中映射的末端。

181 `next` 是受到上锁影响的下一个 VMA。

182 在这个 VMA 上调用 `mlock_fixup`(见 D. 4. 3. 1)。

183~184 如果发生错误,则在这里跳出循环,请记住 VMA 已经被锁定但是还没有修整好。

185 下一个起始地址是下一个 VMA 的起点。

186 移到下一个 VMA。

187~190 如果没有 VMA,在这里返回-ENOMEM。下一个条件是,在 `mlock_fixup()` 实现中断或有重叠的 VMA 时,区域将会很分散。

192 返回错误或成功代码。

D. 4. 2 对区域解锁

D. 4. 2. 1 函数: `sys_munlock()` (`mm/mlock.c`)

在这里对请求按页对齐,然后调用 `do_mlock()`,它开始修整区域的工作。

```
226 asmlinkage long sys_munlock(unsigned long start, size_t len)
227 {
228     int ret;
229
230     down_write(&current->mm->mmap_sem);
231     len = PAGE_ALIGN(len + (start & ~PAGE_MASK));
232     start &= PAGE_MASK;
233     ret = do_mlock(start, len, 0);
234     up_write(&current->mm->mmap_sem);
235     return ret;
236 }
```

230 获取保护 `mm_struct` 的信号量。

231 将区域长度向上取整为最接近的页面边界。

232 将区域的起点向下取整为最接近的页面边界。

233 调用 `do_mlock()`(见 D. 4. 1. 4), 将 0 作为第 3 个参数传入来解锁这片区域。

234 释放信号量。

235 返回成功或者失败代码。

D. 4. 2. 2 函数: `sys_munlockall()` (`mm/mlock.c`)

这个函数很小。如果 `mlockall()` 的标志位为 0, 其意思是不需要当前页面处于当前状态, 页不需要锁定其他的映射, 这也意味着 `VM_LOCKED` 标志位将从所有的 VMA 中移除。

```
293 asmlinkage long sys_munlockall(void)
294 {
295     int ret;
296
297     down_write(&current->mm->mmap_sem);
298     ret = do_mlockall(0);
299     up_write(&current->mm->mmap_sem);
300     return ret;
301 }
```

297 获取保护 `mm_struct` 的信号量。

298 调用 `do_mlockall()`(见 D. 4. 1. 3), 传入 0 作为标志位, 这将把 `VM_LOCKED` 从所有的 VMA 中移除。

299 释放信号量。

300 返回错误或者成功代码。

D. 4. 3 上锁/解锁后修整区域

D. 4. 3. 1 函数: `mlock_fixup()` (`mm/mlock.c`)

必须说明的是, 这个函数识别 4 种不同类型的锁。第 1 种是锁定整个 VMA 时, 调用 `mlock_fixup_all()`。第 2 种是仅有 VMA 的首部受影响, 它由 `mlock_fixup_start()` 处理。第 3 种是锁定末尾的区域, 这由 `mlock_fixup_end()` 处理。第 4 种是利用 `mlock_fixup_middle()` 来锁定中间部分的区域。

```
117 static int mlock_fixup(struct vm_area_struct * vma,
118 unsigned long start, unsigned long end, unsigned int newflags)
119 {
120     int pages, retval;
121
122     if (newflags == vma->vm_flags)
```

```

123 return 0;
124
125 if (start == vma->vm_start) {
126 if (end == vma->vm_end)
127 retval = mlock_fixup_all(vma, newflags);
128 else
129 retval = mlock_fixup_start(vma, end, newflags);
130 } else {
131 if (end == vma->vm_end)
132 retval = mlock_fixup_end(vma, start, newflags);
133 else
134 retval = mlock_fixup_middle(vma, start,
135         end, newflags);
136 }
137 /* keep track of amount of locked VM */
138 pages = (end - start) >> PAGE_SHIFT;
139 if (newflags & VM_LOCKED) {
140 pages = -pages;
141 make_pages_present(start, end);
142 }
143 vma->vm_mm->locked_vm -= pages;
144 }
145 return retval;
146 }

```

122~123 如果没有发生改变则直接返回。

125 如果上锁的起点是 VMA 的开始部分,则意味着或者是整个区域被锁定,或者仅是首部被锁定。

126~127 如果是整个 VMA 被锁定,这里就调用 mlock_fixup_all()(见 D. 4. 3. 2)。

128~129 如果被锁定的 VMA 中开始部分与锁定的开始部分相匹配,则这里调用 mlock_fixup_start()(见 D. 4. 3. 3)。

130 这意味着或者在末尾的区域要被锁定,或者在中间的区域要被锁定。

131~132 如果锁定的末端与 VMA 的末端相匹配,这里调用 mlock_fixup_middle()(见 D. 4. 3. 5)。

136~144 在这里,修整函数成功时返回 0。如果成功修整该区域且该区域已经被标记为锁定,就在这里调用 make_pages_present()做一些基本的检查,然后调用 get_user_pages()。get_user_pages()与缺页中断处理程序一样陷入所有页面。

D. 4.3.2 函数: `mlock_fixup_all()` (`mm/mlock.c`)

```

15 static inline int mlock_fixup_all(struct vm_area_struct * vma,
16                                int newflags)
17 {
18     spin_lock(&vma->vm_mm->page_table_lock);
19     vma->vm_flags = newflags;
20     spin_unlock(&vma->vm_mm->page_table_lock);
21     return 0;
22 }
```

17~19 这个函数很小, 它利用自旋锁锁住 VMA, 设置新标志位, 释放锁并返回成功。

D. 4.3.3 函数: `mlock_fixup_start()` (`mm/mlock.c`)

这个有点复杂, 它需要一个新的 VMA 来表示受影响的区域。旧 VMA 的起始位置移到前面。

```

23 static inline int mlock_fixup_start(struct vm_area_struct * vma,
24                                    unsigned long end, int newflags)
25 {
26     struct vm_area_struct * n;
27
28     n = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
29     if (!n)
30         return -EAGAIN;
31     *n = *vma;
32     n->vm_end = end;
33     n->vm_flags = newflags;
34     n->vm_raend = 0;
35     if (n->vm_file)
36         get_file(n->vm_file);
37     if (n->vm_ops && n->vm_ops->open)
38         n->vm_ops->open(n);
39     vma->vm_pgoff += (end - vma->vm_start) >> PAGE_SHIFT;
40     lock_vma_mappings(vma);
41     spin_lock(&vma->vm_mm->page_table_lock);
42     vma->vm_start = end;
43     __insert_vm_struct(current->mm, n);
44     spin_unlock(&vma->vm_mm->page_table_lock);
45     unlock_vma_mappings(vma);
```

```
46 return 0;
47 }
```

28 从 slab 分配器中为受影响的区域分配一个 VMA。

31~34 复制所需的信息。

35~36 如果 VMA 有一个文件或设备映射, get_file() 将增加引用计数。

37~38 如果提供了一个 open() 函数, 在这里调用它。

39 更新将在上锁区域末端的旧 VMA 的文件偏移或设备映射偏移。

40 若该 VMA 是一个共享区域, lock_vma_mappings() 将上锁任何文件。

41~44 上锁父 mm_struct, 更新其起点为受影响区域的末端, 将一个新的 VMA 加入进程链表(见 D. 2. 2. 1)并释放锁。

45 利用 unlock_vma_mappings() 解锁文件映射。

46 返回成功。

D. 4. 3. 4 函数: mlock_fixup_end() (mm/mlock.c)

除了受影响区域是在 VMA 末端之外, 这个函数实际上与 mlock_fixup_startu() 相同。

```
49 static inline int mlock_fixup_end(struct vm_area_struct * vma,
50 unsigned long start, int newflags)
51 {
52 struct vm_area_struct * n;
53
54 n = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
55 if (!n)
56 return -EAGAIN;
57 *n = *vma;
58 n->vm_start = start;
59 n->vm_pgoff += (n->vm_start - vma->vm_start) >> PAGE_SHIFT;
60 n->vm_flags = newflags;
61 n->vm_raend = 0;
62 if (n->vm_file)
63 get_file(n->vm_file);
64 if (n->vm_ops && n->vm_ops->open)
65 n->vm_ops->open(n);
66 lock_vma_mappings(vma);
67 spin_lock(&vma->vm_mm->page_table_lock);
68 vma->vm_end = start;
69 __insert_vm_struct(current->mm, n);
70 spin_unlock(&vma->vm_mm->page_table_lock);
```

```

71 unlock_vma_mappings(vma);
72 return 0;
73 }

```

54 从 slab 分配器上为受影响区域分配一个 VMA。

57~61 向文件或设备映射中复制所需信息并更新其中的偏移。

62~63 如果 VMA 有一个文件或设备映射, get_file() 将引用计数加 1。

64~65 如果提供了一个 open() 函数, 这里将调用它。

66 如果该 VMA 是一个共享区域, lock_vma_mappings() 将上锁任何文件。

67~70 上锁父 mm_struct, 更新其起点为受影响区域的末端, 将一个新的 VMA 加入进程链表(见 D. 2. 2. 1 节)并释放锁。

71 利用 unlock_vma_mappings() 解锁文件映射。

72 返回成功。

D. 4. 3. 5 函数: mlock_fixup_middle() (mm/unlock.c)

除了需要两个新区域来修整映射外, 这个函数与前面两个修整函数类似。

```

75 static inline int mlock_fixup_middle(struct vm_area_struct * vma,
76 unsigned long start, unsigned long end, int newflags)
77 {
78 struct vm_area_struct * left, * right;
79
80 left = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
81 if (! left)
82 return -EAGAIN;
83 right = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
84 if (! right) {
85 kmem_cache_free(vm_area_cachep, left);
86 return -EAGAIN;
87 }
88 * left = * vma;
89 * right = * vma;
90 left->vm_end = start;
91 right->vm_start = end;
92 right->vm_pgoff += (right->vm_start - left->vm_start) >>
93 PAGE_SHIFT;
94 vma->vm_flags = newflags;
95 left->vm_raend = 0;
96 right->vm_raend = 0;

```

```

96 if (vma->vm_file)
97 atomic_add(2, &vma->vm_file->f_count);
98
99 if (vma->vm_ops && vma->vm_ops->open) {
100 vma->vm_ops->open(left);
101 vma->vm_ops->open(right);
102 }
103 vma->vm_raend = 0;
104 vma->vm_pgoff += (start - vma->vm_start) >> PAGE_SHIFT;
105 lock_vma_mappings(vma);
106 spin_lock(&vma->vm_mm->page_table_lock);
107 vma->vm_start = start;
108 vma->vm_end = end;
109 vma->vm_flags = newflags;
110 __insert_vm_struct(current->mm, left);
111 __insert_vm_struct(current->mm, right);
112 spin_unlock(&vma->vm_mm->page_table_lock);
113 unlock_vma_mappings(vma);
114 return 0;
115 }

```

80~87 从 slab 分配器中分配两块新的 VMA。

88~89 将信息从旧 VMA 中复制到新的 VMA 中。

90 左区域的末端是受影响区域的始端。

91 右区域的始端是受影响区域的末端。

92 更新文件偏移。

93 旧 VMA 现在是受影响区域, 所以这里更新它的标志位。

94~95 将预读窗口设为 0, 以保证不属于它们区域的页面不会突然预读。

96~97 如果有文件/设备映射, 则增加它们的引用计数。

99~192 对两个新映射调用 open() 函数。

103~104 取消预读窗口, 更新将处于上锁区域始端文件中的偏移。

105 上锁共享文件/设备映射。

106~112 上锁父 mm_struct, 更新 VMA 并将其插入到这两个新区域到进程链表中, 然后再次释放锁。

113 解锁共享区域。

114 返回成功。

D.5 缺页中断

这一部分关于缺页中断处理程序。首先是 x86 体系结构相关的函数,然后转移到体系结构无关层。特定体系结构的函数都具有相同的任务。

D.5.1 x86 缺页中断处理程序

D.5.1.1 函数: `do_page_fault()` (`arch/i386/mm/fault.c`)

该函数的调用图如图 4.11 所示。它是处理缺页中断异常的 x86 体系结构相关函数。每种体系结构都只注册它自己,但它们的任务却都相似。

```

140 asmlinkage void do_page_fault(struct pt_regs *regs,
141                               unsigned long error_code)
142 {
143     struct task_struct *tsk;
144     struct mm_struct *mm;
145     struct vm_area_struct * vma;
146     unsigned long address;
147     unsigned long page;
148     unsigned long fixup;
149     int write;
150     siginfo_t info;
151
152     /* get the address */
153     __asm__("movl %cr2,%0": = r" (address));
154
155     /* It's safe to allow irqs after cr2 has been saved */
156     if (regs->eflags & X86_EFLAGS_IF)
157         local_irq_enable();
158
159     tsk = current;

```

这是函数的开始部分,它获取异常地址并开中断。

140 参数如下:

- `reg` 是在异常时包含所有寄存器的结构。

- error_code 表明发生异常的类型。

152 如注释所示,cr2 寄存器保留有异常地址。

155~156 如果异常是从中断中来,这里开中断。

158 设置当前的进程。

```

173 if (address >= TASK_SIZE && ! (error_code & 5))
174 goto vmalloc_fault;
175
176 mm = tsk->mm;
177 info.si_code = SEGV_MAPERR;
178
183 if (in_interrupt() || ! mm)
184 goto no_context;
185

```

这一块检查例外异常、核心异常、陷入中断和无内存上下文异常。

173 如果异常地址超过 TASK_SIZE,则它在内核地址空间。如果错误码是 5,则意味着错误发生在核态且不是一个保护性错误,所以这里处理 vmalloc 异常。

176 记录一个工作中的 mm。

183 如果这是一个中断且没有内存上下文(如在一个内核线程中),则没有办法来安全地处理这个异常,所以转到 no_context。

```

186 down_read(&mm->mmap_sem);
187
188 vma = find_vma(mm, address);
189 if (! vma)
190 goto bad_area;
191 if (vma->vm_start <= address)
192 goto good_area;
193 if (!(vma->vm_flags & VM_GROWSDOWN))
194 goto bad_area;
195 if (error_code & 4) {
196 /*
197 * accessing the stack below %esp is always a bug.
198 * The "+ 32" is there due to some instructions (like
199 * pusha) doing post-decrement on the stack and that
200 * doesn't show up until later..
201 */
202 if (address + 32 < regs->esp)
203 goto bad_area;

```

```

204 }
205 if (expand_stack(vma, address))
206 goto bad_area;

```

如果异常发生在用户空间,这一块找到异常地址的 VMA,以确定它是一块好区域,还是一块坏区域,或者异常是否发生在一块可扩展区域附近,如栈。

186 获取长生命期的 mm 信号量。

188 找到在异常地址的或与异常地址最近的 VMA。

189~190 如果根本就不存在 VMA,则转到 bad_area。

191~192 如果区域的始端在该地址前面,则意味着这个 VMA 就是发生异常的 VMA,所以转到 good_area,在那里检查权限。

193~194 对最邻近的区域,这里检查它是否可以向下延伸(VM_GROWDOWN)。如果可以,则意味着栈可以扩展。如果不可以,则转到 bad_area。

205~206 栈是惟一设置了 VM_GROWDOWN 的区域。所以,如果我们到达这里,将调用 expand_stack() 来扩展栈(见 D.5.2.1)。如果失败,则转到 bad_area。

```

211 good_area;
212 info.si_code = SEGV_ACCERR;
213 write = 0;
214 switch (error_code & 3) {
215 default: /* 3: write, present */
216 #ifdef TEST_VERIFY_AREA
217 if (regs->cs == KERNEL_CS)
218 printk("WP fault at %08lx\n", regs->eip);
219 #endif
220 /* fall through */
221 case 2: /* write, not present */
222 if (!(vma->vm_flags & VM_WRITE))
223 goto bad_area;
224 write++;
225 break;
226 case 1: /* read, present */
227 goto bad_area;
228 case 0: /* read, not present */
229 if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
230 goto bad_area;
231 }

```

这一块是在好的区域中处理异常的部分。这里需要检查权限以防止这是一个保护性

异常。

212 缺省情况下,这里返回一个错误。

214 检查错误码的 0 位和 1 位。0 位为 0 表示页面不存在,为 1 表示这是一个如向只读区域写一样的保护性异常。1 位为 0 表示这是一个读异常,为 1 表示这是一个写异常。

215 如果错误码为 3,而 0 位和 1 位都为 1,则为一个写保护异常。

221 位 1 为 1,则为一个写异常。

222~223 如果该区域不能写入,则是一个转到 bad_area 的坏的写。如果该区域可以写入,则这是一个标记为写时复制(COW)的页面。

224 发生写的标志位。

226~227 这是一个读操作,页面也存在。由于该异常的发生没有理由解释,则这里肯定是其他的异常,比如被 0 除,处理时转到 bad_area。

228~230 在缺失页上进行读操作。在这里保证该页面可读,或可对页面进行 exec 处理。如果不是这样,则转到 bad_area。在这里对 exec 进行检查是因为 x86 不能对页面进行 exec 保护,而是使用了读保护标志位。这也正是两者都必须进行检查的原因。

```

233 survive:
239 switch (handle_mm_fault(mm, vma, address, write)) {
240 case 1:
241     tsk->min_flt++;
242     break;
243 case 2:
244     tsk->maj_flt++;
245     break;
246 case 0:
247     goto do_sigbus;
248 default:
249     goto out_of_memory;
250 }
251
252 /*
253 * Did it hit the DOS screen memory VA from vm86 mode?
254 */
255 if (regs->eflags & VM_MASK) {
256     unsigned long bit = (address - 0xA0000) >> PAGE_SHIFT;
257     if (bit < 32)
258         tsk->thread.screen_bitmap |= 1 << bit;
259 }
260 up_read(&mm->mmap_sem);

```

261 return;

在这里,试着调用 handle_mm_fault()来恰当地处理异常。

239 调用 handle_mm_fault(),传入与异常有关的信息。这是处理程序独立于体系结构的部分。

240~242 返回 1 表示这是一个次异常,更新统计信息。

243~245 返回 2 表示这是一个主异常,更新统计信息。

246~247 返回 0 表示在异常时发生了一些 I/O 错误,所以转到 do_sigbus 处理程序。

248~249 其他的返回值表示不能为该异常分配内存,即发生了内存溢出。实际上,这几乎不会发生,因为 mm/com_kill.c 中涉及的 out_of_memory()在其他函数中可能已经发生, out_of_memory()是一个处理这种情况更为恰当的函数。

260 释放 mm 上的锁。

261 因为异常都成功地处理,则在这里返回。

```

267 bad_area:
268 up_read(&mm->mmap_sem);
269
270 /* User mode accesses just cause a SIGSEGV */
271 if (error_code & 4) {
272 tsk->thread.cr2 = address;
273 tsk->thread.error_code = error_code;
274 tsk->thread.trap_no = 14;
275 info.si_signo = SIGSEGV;
276 info.si_errno = 0;
277 /* info.si_code has been set above */
278 info.si_addr = (void *)address;
279 force_sig_info(SIGSEGV, &info, tsk);
280 return;
281 }
282
283 /*
284 * Pentium F0 OF C7 C8 bug workaround.
285 */
286 if (boot_cpu_data.f00f_bug) {
287 unsigned long nr;
288
289 nr = (address - idt) >> 3;
290
291 if (nr == 6) {

```

```

292 do_invalid_op(regs, 0);
293 return;
294 }
295 }

```

这是一个坏区域处理程序,坏区域是指比如使用了没有 `vm_area_struct` 管理的内存区域等。如果该异常不是一个用户进程异常或者 `foofbug`,则转到 `no_context` 标记。

271 错误码为 4 表示是用户空间,所以这是一个简单的情形,发送 `SIGSEGV` 来杀死进程。

272~274 设置关于所发生事件的线程信息,这些信息可以在后面由调试器来读。

275 对发送了一个 `SIGSEGV` 信号进行记录。

276 清除错误码,因为 `SIGSEGV` 已经足够用来解释这个错误。

278 记录地址。

279 发送 `SIGSEGB` 信号。这个进程将退出,并打印所有的相关信息。

280 因为异常都成功地处理,这里返回。

286~295 在奔腾第 1 版中有一个 bug 是 `foofbug`,它会导致处理器不断进行缺页中断。它用于在运行 Linux 系统中进行局部 Dos 攻击。这个 bug 在发布的几个小时后就被发现,并且打上了布丁。现在它只会导致进程的无害中止,而不是系统重启。

```

296
297 no_context:
298 /* Are we prepared to handle this kernel fault? */
299 if ((fixup = search_exception_table(regs->eip)) != 0) {
300 regs->eip = fixup;
301 return;
302 }

```

299~302 调用 `search_exception_table()` 查找异常表,来确定该异常已经被处理过。如果处理过,则在返回后调用一个合适的异常处理程序。这在 `copy_from_user()` 和 `copy_to_user()` 时非常重要,在这时会设置一个异常处理程序来监控读和写用户空间中的无效区域,而不需要进行多次代价巨大的检查。这意味着可以调用一个小部分的修整代码,而不是退回到会产生 `oops` 的下一块。

```

304 /*
305 * Oops. The kernel tried to access some bad page. We'll have to
306 * terminate things with extreme prejudice.
307 */
308
309 bust_spinlocks(1);

```

```

310
311 if (address < PAGE_SIZE)
312 printk (KERN_ALERT "Unable to handle kernel NULL pointer
313 dereference");
313 else
314 printk (KERN_ALERT "Unable to handle kernel paging
315 request");
315 printk(" at virtual address %08lx\n",address);
316 printk(" printing eip:\n");
317 printk("%08lx\n", regs->eip);
318 asm("movl %cr3,%0":=r" (page));
319 page = ((unsigned long *) __va(page))[address >> 22];
320 printk(KERN_ALERT "* pde = %08lx\n", page);
321 if (page & 1) {
322 page &= PAGE_MASK;
323 address &= 0x003ff000;
324 page = ((unsigned long *) __va(page))[address >> PAGE_SHIFT];
325 printk(KERN_ALERT "* pte = %08lx\n", page);
326 }
327 die("Oops", regs, error_code);
328 bust_spinlocks(0);
329 do_exit(SIGKILL);

```

这是一个 no_context 处理程序。在发生一些破坏性异常时,它们会使进程中止。否则,在确定要发生内核异常时,生成一个 oops 报告,然后开始内核陷入。

309~329 否则,在不应该发生内核异常时发生内核陷入,这是一个内核 bug,这一块产生一个 oops 报告。

309 强制性的释放自旋锁,这可能阻止信息显示到屏幕上。

311~312 如果地址< PAGE_SIZE,则意味着使用了一个 null 指针,Linux 中故意不分配 0 页,这样可以捕获此种类型的异常,这是一个常见的程序错误。

313~314 否则,则是一些破坏性的内核错误,如一个驱动程序尝试不正确地访问用户空间。

315~320 打印关于异常的信息。

321~326 打印关于缺页的信息。

327 死亡,并产生 oops 报告,这些可以在后面用于跟踪堆栈,这样开发人员可以更加清晰地看到在哪里或是什么时候发生了异常。

329 强制性地杀死异常进程。

```

335 out_of_memory:
336 if (tsk->pid == 1) {
337     yield();
338     goto survive;
339 }
340 up_read(&mm->mmap_sem);
341 printk("VM: killing process %s\n", tsk->comm);
342 if (error_code & 4)
343     do_exit(SIGKILL);
344 goto no_context;

```

这一块处理内存溢出。除非是 init, 这里经常以结束异常进程为收尾。

336~339 如果进程是 init, 则折返到 survive, 在那里试着恰当地处理异常。init 应该永远都不会被杀死。

340 释放 mm 信号量。

341 打印帮助信息“You are Dead”。

342 如果这是来自用户空间的, 则这里仅杀死该进程。

344 如果来自内核空间, 则转到 no_context 处理程序, 在那里将可能导致一个内核 oops。

```

345
346 do_sigbus:
347 up_read(&mm->mmap_sem);
348
353 tsk->thread.cr2 = address;
354 tsk->thread.error_code = error_code;
355 tsk->thread.trap_no = 14;
356 info.si_signo = SIGBUS;
357 info.si_errno = 0;
358 info.si_code = BUS adrerr;
359 info.si_addr = (void *)address;
360 force_sig_info(SIGBUS, &info, tsk);
361
362 /* Kernel mode? Handle exceptions or die */
363 if (!(error_code & 4))
364     goto no_context;
365 return;

```

347 释放 mm 锁。

353~359 填充信息, 显示在异常地址处发生了一个 SIGBUS, 这样在后面调试器可以跟踪到这里。

360 发送信号。

363~364 如果处于内核模式,则这里试着在 no_context 时处理该异常。

365 如果是用户空间,则这里仅返回,所有的进程都按预定时的情况全部被杀死。

```

367 vmalloc_fault:
368 {
376 int offset = __pgd_offset(address);
377 pgd_t * pgd, * pgd_k;
378 pmd_t * pmd, * pmd_k;
379 pte_t * pte_k;
380
381 asm("movl % %cr3, %0": = r" (pgd));
382 pgd = offset + (pgd_t *)__va(pgd);
383 pgd_k = init_mm.pgd + offset;
384
385 if (! pgd_present(* pgd_k))
386 goto no_context;
387 set_pgd(pgd, * pgd_k);
388
389 pmd = pmd_offset(pgd, address);
390 pmd_k = pmd_offset(pgd_k, address);
391 if (! pmd_present(* pmd_k))
392 goto no_context;
393 set_pmd(pmd, * pmd_k);
394
395 pte_k = pte_offset(pmd_k, address);
396 if (! pte_present(* pte_k))
397 goto no_context;
398 return;
399 }
400 }
```

这是 vmalloc 异常处理程序。当页面映射到 vmalloc 空间时,则仅更新引用页表。对该区域中的各进程引用,将陷入一个异常,进程页表也会在这里与引用页表进行同步操作。

376 获得在 PGD 中的偏移。

381 从 cr3 中复制进程的 PGD 地址到 pgd。

382 从进程 PGD 中计算 pgd 指针。

383 计算内核引用 PGD。

385~386 如果 pgd 项对内核页表无效,则转入 no_context。

386 从内核引用页表中复制一份页表项, 设置进程页表中的页表项。

389~393 对 PMD 进行同样的操作。从内核引用页表中复制一份页表项, 设置进程页表中的页表项。

395 检查 PTE。

396~397 如果 PTE 不存在, 则意味着即使在内核引用页表中该页也无效, 所以转到 no_context 进行处理。这里可能是一个内核 bug, 或者是一个对无用内核空间中随机部分的引用。

398 返回已经更新过的进程页表, 并与内核页表进行同步操作。

D.5.2 扩展栈

D.5.2.1 函数: `expand_stack()` (`include/linux/mm.h`)

这是一个依赖于体系结构的缺页异常处理程序调用函数。VMA 就是一个可以扩大来覆盖该地址的例子。

```

640 static inline int expand_stack(struct vm_area_struct * vma,
641                           unsigned long address)
642 {
643     unsigned long grow;
644
645     /* vma->vm_start/vm_end cannot change under us because
646     * the caller is required
647     * to hold the mmap_sem in write mode. We need to get the
648     * spinlock only before relocating the vma range ourself.
649     */
650     address &= PAGE_MASK;
651     spin_lock(&vma->vm_mm->page_table_lock);
652     grow = (vma->vm_start - address) >> PAGE_SHIFT;
653     if (vma->vm_end - address >
654         current->rlim[RLIMIT_STACK].rlim_cur ||
655         ((vma->vm_mm->total_vm + grow) << PAGE_SHIFT) >
656         current->rlim[RLIMIT_AS].rlim_cur) {
657         spin_unlock(&vma->vm_mm->page_table_lock);
658         return -ENOMEM;
659     }
660     vma->vm_start = address;

```

```

658 vma->vm_pgoff -= grow;
659 vma->vm_mm->total_vm += grow;
660 if (vma->vm_flags & VM_LOCKED)
661 vma->vm_mm->locked_vm += grow;
662 spin_unlock(&vma->vm_mm->page_table_lock);
663 return 0;
664 }

```

649 将地址向下取整到页面边界。

650 上锁页表自旋锁。

651 计算栈需要扩展多大的页面。

652 检查以保证栈的大小不会超过进程限度。

653 检查以保证地址空间的大小在栈扩展之后不会超过进程限度。

654~655 如果达到了任何一个限度, 则这里返回-ENOMEM, 它将导致一个段错误的异常进程。

657~658 向下扩大 VMA。

659 更新进程用到的地址空间量。

660~661 如果锁定了一片区域, 则更新该进程用到的锁定页面数量。

662~663 更新进程页表, 返回成功。

D.5.3 独立体系结构的页面中断处理程序

这是独立体系结构的页面中断处理程序的顶层函数对。

D.5.3.1 函数: handle_mm_fault() (mm/memory.c)

这个函数的调用图如图 4.13 所示。这个函数为待分配的新 PTE 分配所需的 PMD 和 PTE。它先获取必要的锁来保护页表, 然后调用 handle_pte_fault() 陷入页面本身。

```

1364 int handle_mm_fault(struct mm_struct * mm,
1365     struct vm_area_struct * vma,
1366     unsigned long address, int write_access)
1367 {
1368     pgd_t * pgd;
1369     pmd_t * pmd;
1370     current->state = TASK_RUNNING;
1371     pgd = pgd_offset(mm, address);
1372

```

```

1373 /*
1374 * We need the page table lock to synchronize with kswapd
1375 * and the SMP-safe atomic PTE updates.
1376 */
1377 spin_lock(&mm->page_table_lock);
1378 pmd = pmd_alloc(mm, pgd, address);
1379
1380 if (pmd) {
1381 pte_t * pte = pte_alloc(mm, pmd, address);
1382 if (pte)
1383 return handle_pte_fault(mm, vma, address,
1384                         write_access, pte);
1384 }
1385 spin_unlock(&mm->page_table_lock);
1386 return -1;
1387 }

```

1364 该函数的参数如下：

- mm 是异常进程的 mm_struct。
- vma 是管理异常发生区域的 vm_area_struct。
- address 是异常地址。
- write_access 如果是一个写异常则为 1。

1370 设置该进程的当前状态。

1371 获取顶层页表中的 pgd 项。

1377 由于页表将会被改变,所以锁住 mm_struct。

1378 如果 pmd_t 不存在,pmd_alloc()将分配一个。

1380 如果已经成功分配了 pmd,则……

1381 如果 PTE 不存在,则为该地址分配一个。

1382~1383 调用 handle_pte_fault()(见 D. 5.3.2)来处理缺页中断,返回状态码。

1385 失败路径,解锁 mm_struct。

1386 返回-1,这将被解释为一个内存溢出异常。这里是正确的,因为要到达这条线,当且仅当不能分配 PMD 或 PTE。

D. 5.3.2 函数: handle_pte_fault() (mm/memory.c)

这个函数确定发生的异常类型,以及调用的处理函数。如果这是第一次分配页面,则调用 do_no_page(),do_swap_page()处理在页面从 tmpfs 中意外地换出到磁盘的情形。do_wp_page()分割 COW 页面。如果它们都不合适,则简单地更新 PTE 表项。如果是写入,则标记

为脏,如果是初期页面,则标记为已访问。

```

1331 static inline int handle_pte_fault(struct mm_struct * mm,
1332 struct vm_area_struct * vma, unsigned long address,
1333 int write_access, pte_t * pte)
1334 {
1335     pte_t entry;
1336
1337     entry = * pte;
1338     if (! pte_present(entry)) {
1339         /*
1340         * If it truly wasn't present, we know that kswapd
1341         * and the PTE updates will not touch it later. So
1342         * drop the lock.
1343         */
1344     if (pte_none(entry))
1345         return do_no_page(mm, vma, address,
1346                           write_access, pte);
1346     return do_swap_page(mm, vma, address, pte, entry,
1347                         write_access);
1347 }
1348
1349     if (write_access) {
1350         if (! pte_write(entry))
1351             return do_wp_page(mm, vma, address, pte, entry);
1352
1353     entry = pte_mkdirty(entry);
1354 }
1355     entry = pte_mkyoung(entry);
1356     establish_pte(vma, address, pte, entry);
1357     spin_unlock(&mm->page_table_lock);
1358     return 1;
1359 }
```

1331 这个函数的参数与 handle_mm_fault() 的参数基本相同,除了这个函数包括的异常 PTE。

1337 记录 PTE。

1338 处理存在 PTE 的情形。

1344 如果 PTE 从来就没有被填充,则这里利用 do_no_page() (见 D. 5. 4. 1) 来处理 PTE 的分配。

1346 如果页面已经被交换到后援存储器,这里利用 do_swap_page()(见 D.5.5.1)处理。

1349~1354 处理页面已经被写入的情形。

1350~1351 如果 PTE 被标记为只写,则它是一个 COW 页面,所以这里利用 do_wp_page()(见 D.5.6.1)处理。

1353 否则,这里仅标记页面为脏。

1355 标记页面为已访问。

1356 establish_pte()复制 PTE,然后更新 TLB 和 MMU 高速缓存。这里并不复制新的 PTE,但是某些体系结构需要更新 TLB 和 MMU。

1357 解锁 mm_struct 并返回一个发生了的小异常。

D.5.4 请求分配

D.5.4.1 函数: do_no_page() (mm/memory.c)

这个函数的调用图如图 4.14 所示。这个函数在页面第一次被引用时调用,这样它可以在需要时分配和填充数据。如果它是一个由 VMA 中的 vm_ops 或缺少 nopage()函数所决定的匿名页,则调用 do_anonymous_page()。否则,调用替代的 nopage()函数来分配页面,然后插入到这里的页表中。函数的功能如下:

- 检查是否可用 do_anonymous_page()。如果可以则调用它,返回它分配的页面。如果不可以调用,则调用替代的 nopage()函数,并保证它能够成功地分配一页。
- 如果需要则尽早使 COW 失效。
- 将页面加入到页表项,然后调用适合的体系结构相关的钩子函数。

```

1245 static int do_no_page(struct mm_struct * mm,
1246                         struct vm_area_struct * vma,
1247                         unsigned long address, int write_access, pte_t * page_table)
1248 {
1249     struct page * new_page;
1250     pte_t entry;
1251     if (!vma->vm_ops || !vma->vm_ops->nopage)
1252         return do_anonymous_page(mm, vma, page_table,
1253                               write_access, address);
1253     spin_unlock(&mm->page_table_lock);
1254
1255     new_page = vma->vm_ops->nopage(vma, address & PAGE_MASK, 0);
1256

```

```

1257 if (new_page == NULL) /* no page was available — SIGBUS */
1258 return 0;
1259 if (new_page == NOPAGE_OOM)
1260 return -1;

```

1245 这里提供的参数与 handle_pte_fault() 的相同。

1251~1252 如果没有提供 vm_ops, 或者没有提供 nopage() 函数, 则这里调用 do_anonymous_page() (见 D. 5.4.2) 来分配一个页面并返回该页面。

1253 否则, 这里释放页表锁, 因为 nopage() 在上锁自旋锁的时候不能被调用。

1255 调用替代的 nopage() 函数, 在文件系统的情形下, 这里频繁地调用 filemap_nopage() (见 D. 6.4.1), 但每个不同的设备驱动器不同。

1257~1258 如果返回了 NULL, 则意味着在 nopage() 函数中发生了某些错误, 如在读磁盘的时候发生了一个 I/O 错误。在这种情形下, 返回 0, 它将导致发送一个 SIGBUS 到中断处理进程。

1259~1260 如果返回了 NOPAGE_OOM, 则物理页面分配器分配页面失败, 返回 -1, 它将强制性地杀死该进程。

```

1265 if (write_access && !(vma->vm_flags & VM_SHARED)) {
1266 struct page * page = alloc_page(GFP_HIGHUSER);
1267 if (!page) {
1268 page_cache_release(new_page);
1269 return -1;
1270 }
1271 copy_user_highpage(page, new_page, address);
1272 page_cache_release(new_page);
1273 lru_cache_add(page);
1274 new_page = page;
1275 }

```

这一块在适当的时候尽早使 COW 失效。如果发生的异常是写异常, 且区域没有利用 VM_SHARED 共享, 则要使 COW 失效。如果在这种情况下没有使 COW 失效, 则一旦返回就立即发生第二次异常。

1265 检查 COW 是否需要立即失效。

1266 如果是, 则这里为进程分配一个新页。

1267~1270 如果不能分配页面, 则这里将利用函数 nopage() 返回的页面引用计数减 1, 并返回 -1 报告内存不足。

1271 否则, 复制页面内容。

1272 将返回页面的引用计数减 1, 该页面可能还被其他进程使用。

1273 将新页加入到 LRU 链表中,这样可以在后面由 kswapd 回收。

```

1277 spin_lock(&mm->page_table_lock);
1288 /* Only go through if we didn't race with anybody else... */
1289 if (pte_none(*page_table)) {
1290     ++mm->rss;
1291     flush_page_to_ram(new_page);
1292     flush_icache_page(vma, new_page);
1293     entry = mk_pte(new_page, vma->vm_page_prot);
1294     if (write_access)
1295         entry = pte_mkwrite(pte_mkdirty(entry));
1296     set_pte(page_table, entry);
1297 } else {
1298     /* One of our sibling threads was faster, back out. */
1299     page_cache_release(new_page);
1300     spin_unlock(&mm->page_table_lock);
1301     return 1;
1302 }
1303
1304 /* no need to invalidate: a not-present page shouldn't
 * be cached
 */
1305 update_mmu_cache(vma, address, entry);
1306 spin_unlock(&mm->page_table_lock);
1307 return 2; /* Major fault */
1308 }
```

1277 再一次上锁页表,因为已经完成了分配,页表即将被更新。

1289 检查在我们将使用的表项中是否仍然没有 PTE。如果在这里同时有两个异常,则可能是另外一个处理器已经完成了缺页中断处理,所以这个应该退出。

1290~1297 如果没有 PTE 进入,这里就完成异常处理。

1290 将 RSS 计数加 1,因为进程现在正在使用另外一个页面。在这里确实需要一次检查来保证这不是一个全局 0 页面,否则 RSS 计数将会不正确。

1291 因为页面将被映射到进程空间,所以对某些体系结构而言可能需要将页面写到那些不为进程所见的内核空间页面中。flush_page_to_ram()将保证 CPU 高速缓存是连贯的。

1292 flush_icache_page()原理上是相似的,除了它保证 icache 和 dcache 是连贯的。

1293 以合适的权限来创建 pte_t。

1294~1295 如果这是一个写,则保证 PTE 拥有写权限。

1296 将新 PTE 放入到进程页表中。

1297~1302 如果已经填充了 PTE, 则从函数 `nopage()` 获取的页面将被释放。

1299 将页面引用计数减 1, 如果减到 0, 则释放页面。

1300~1301 释放 `mm_struct` 锁, 并返回 1 来表明这是一个次缺页中断, 因为不需要对这种异常进行大的处理, 大部分工作已经在竞争胜方中完成。

1305 更新体系结构所需的 MMU 高速缓存。

1306~1307 释放 `mm_struct` 锁, 并返回 2 来表明这是一个主缺页中断。

D.5.4.2 函数: `do_anonymous_page()` (`mm/memory.c`)

这个函数为第一次访问页面的进程分配一个新页面。如果是读访问, 则将一个系统范围内全 0 页面映射到进程。如果是写访问, 则分配一个填充 0 的页面, 然后放入进程页表。

```

1190 static int do_anonymous_page(struct mm_struct * mm,
1191     struct vm_area_struct * vma,
1192     pte_t * page_table, int write_access,
1193     unsigned long addr)
1194 {
1195     pte_t entry;
1196
1197     /* Read-only mapping of ZERO_PAGE. */
1198     entry = pte_wrprotect(mk_pte(ZERO_PAGE(addr),
1199         vma->vm_page_prot));
1200
1201     /* ...except if it's a write access */
1202     if (write_access) {
1203         struct page * page;
1204
1205         page = alloc_page(GFP_HIGHUSER);
1206         if (!page)
1207             goto no_mem;
1208
1209         spin_lock(&mm->page_table_lock);
1210         if (!pte_none(*page_table)) {
1211             page_cache_release(page);
1212             spin_unlock(&mm->page_table_lock);
1213         }
1214     }
1215
1216     return 1;

```

```

1214 }
1215 mm->rss++;
1216 flush_page_to_ram(page);
1217 entry = pte_mkwrite(
    pte_mkdirty(mk_pte(page, vma->vm_page_prot)));
1218 lru_cache_add(page);
1219 mark_page_accessed(page);
1220 }
1221
1222 set_pte(page_table, entry);
1223
1224 /* No need to invalidate - it was non-present before */
1225 update_mmu_cache(vma, addr, entry);
1226 spin_unlock(&mm->page_table_lock);
1227 return 1; /* Minor fault */
1228
1229 no_mem:
1230 return -1;
1231 }

```

1190 参数与传递给 handle_pte_fault() (见 D.5.3.2) 的参数相同。

1195 对读访问, 这里只是简单地映射系统范围的 empty_zero_page, 它由给定权限的 ZERO_PAGE() 宏返回。由于页面写保护, 所以如果对这个页面进行写操作将导致一个缺页中断。

1198~1220 如果是写异常, 则分配一个新页并用 0 填充。

1202 解锁 mm_struct, 这样分配新页的进程将睡眠。

1204 分配一个新页。

1205 如果不能分配一页, 则这里返回 -1 来处理 OOM 的情形。

1207 用 0 填充页面。

1209 重新获得锁, 因为页表将被更新。

1215 更新进程的 RSS。注意如果 RSS 由全局 0 页面映射, 则没有被更新。因为在 1195 行已经发生了只读异常。

1216 保证高速缓存连贯。

1217 标记 PTE 为可写和脏的, 因为它已经被写入。

1218 将页面加入到 LRU 链表, 这样它可能由换出进程回收。

1219 标记页面为已访问, 这样保证页面标记为活跃, 并在 active 链表的首部。

1222 为该进程调整页表中的 PTE。

1225 如果体系结构需要则更新 MMU 高速缓存。

1226 释放页表锁。

1227 返回次缺页中断。即使页面分配器可能已经开始写出页面,但是仍然没有从磁盘读入数据来填充这个页面。

D.5.5 请求分页

D.5.5.1 函数: `do_swap_page()` (`mm/memory.c`)

这个函数的调用图如图 4.15 所示。这个函数处理页面已经被换出的情形。一个已经被换出的页面在预读时如果是被多个进程所共享或者是刚刚换出则可能存在于交换高速缓存中。这个函数分成 3 个部分:

- 在交换高速缓存中找到该页面。
- 如果不存在,则调用 `swapin_readahead()` 来读入页面。
- 将页面插入到进程页表中。

```

1117 static int do_swap_page(struct mm_struct * mm,
1118 struct vm_area_struct * vma, unsigned long address,
1119 pte_t * page_table, pte_t orig_pte, int write_access)
1120 {
1121 struct page * page;
1122 swap_entry_t entry = pte_to_swap_entry(orig_pte);
1123 pte_t pte;
1124 int ret = 1;
1125 .
1126 spin_unlock(&mm->page_table_lock);
1127 page = lookup_swap_cache(entry);

```

这一块是函数的前面部分。它检查在交换高速缓存中的页面。

1117~1119 参数与 `handle_pte_fault()` (见 D.5.3.2) 的一样。

1122 从 PTE 中获取交换项信息。

1126 释放 `mm_struct` 自旋锁。

1127 在交换高速缓存中查找该页面。

```

1128 if (! page) {
1129 swapin_readahead(entry);
1130 page = read_swap_cache_async(entry);
1131 if (! page) {

```

```

1136 int retval;
1137 spin_lock(&mm->page_table_lock);
1138 retval = pte_same(*page_table, orig_pte) ? -1 : 1;
1139 spin_unlock(&mm->page_table_lock);
1140 return retval;
1141 }
1142
1143 /* Had to read the page from swap area; Major fault */
1144 ret = 2;
1145 }

```

如果页面不在交换高速缓存中,则这一块利用 swapin_readhead() 从后援存储器中读入该页, swapin_readhead() 读入请求页以及其后的一些页。完成后, read_swap_cache_async() 将返回该页。

1128~1145 如果页面不在交换高速缓存中则执行这一块。

1129 swapin_readahead() (见 D. 6. 6. 1) 读入请求页, 以及其后的一些页。读入页数由 mm/swap.c 中 page_cluster 变量决定, 这个变量在小于 16 MB 内存时初始化为 2, 否则初始化为 3。除非已经读到错误的或者空的页表项, 这一块在读取请求页后还将读取 $2^{\text{page_cluster}}$ 个页面。

1130 read_swap_cache_async() (见 K. 3. 1. 1) 将查找请求页并在需要时将其从磁盘读入交换高速缓存。

1131~1141 如果页面不存在, 则换入的该页上将有异常, 并在释放自旋锁时将其从高速缓存中移除。

1137 上锁 mm_struct。

1138 将两个 PTE 进行比较。如果它们不匹配, 就返回 -1 表明一个 I/O 错误。如果匹配, 则返回 1 表明一个次缺页中断, 因为在这个特殊页上不需要一次磁盘访问。

1139~1140 释放 mm_struct 并返回状态位。

1144 磁盘必须被访问以标记这是一个主缺页中断。

```

1147 mark_page_accessed(page);
1148
1149 lock_page(page);
1150
1151 /*
1152 * Back out if somebody else faulted in this pte while we
1153 * released the page table lock.
1154 */
1155 spin_lock(&mm->page_table_lock);

```

```

1156 if (!pte_same(*page_table, orig_pte)) {
1157     spin_unlock(&mm->page_table_lock);
1158     unlock_page(page);
1159     page_cache_release(page);
1160     return 1;
1161 }
1162
1163 /* The page isn't present yet, go ahead with the fault. */
1164
1165 swap_free(entry);
1166 if (vm_swap_full())
1167     remove_exclusive_swap_page(page);
1168
1169 mm->rss++;
1170 pte = mk_pte(page, vma->vm_page_prot);
1171 if (write_access && can_share_swap_page(page))
1172     pte = pte_mkdirty(pte_mkwrite(pte));
1173 unlock_page(page);
1174
1175 flush_page_to_ram(page);
1176 flush_icache_page(vma, page);
1177 set_pte(page_table, pte);
1178
1179 /* No need to invalidate - it was non-present before */
1180 update_mmu_cache(vma, address, pte);
1181 spin_unlock(&mm->page_table_lock);
1182 return ret;
1183 }

```

这一块将页面放到进程页表中。

1147 `mark_page_accessed()`(见 J. 2. 3. 1)将标记页面为活跃,这样它将被移到活跃 LRU 链表的顶端。

1149 上锁页面,这里的副作用是等待换入页面 I/O 完成。

1155~1161 如果其他人在我们之前已经陷入页面,则撤销对页面的引用,释放锁,返回一个次中断。

1165 函数 `swap_free()`(见 K. 2. 2. 1)减少对交换区表项的引用,如果减到 0,则它实际上被释放了。

1166~1167 在页面已经交换出去以避免每次都必须搜索一个空闲槽之后,在交换空间的

页槽依旧为页面保留着。如果交换空间已满，则打破保留，释放该槽给另外一个页面。

1169 即将使用这个页面，所以这里增加 mm_struct 的 RSS 计数。

1170 标记该页面的 PTE。

1171 如果页面已经被写入，而且没有被多个进程共享，这里将标记该页面为脏，这样它可以与后援存储器和其他进程的交换缓冲区保持同步。

1173 解锁页面。

1175 因为页面将被映射到进程空间，所以对某些体系结构而言就可能将页面写入到进程不可见的内核空间。flush_page_to_ram()保证高速缓存是连贯的。

1176 flush_icache_page()原理上相似，此外它还要保证 icache 和 dcache 连贯。

1177 设置进程页表中的 PTE。

1180 如果体系结构需要则更新 MMU 高速缓存。

1181~1182 解锁 mm_struct，并返回它是否是一个主或次缺页中断。

D.5.5.2 函数：can_share_swap_page() (mm/swapfile.c)

这个函数决定该页面的交换高速缓存表项是否可用。如果没有其他的引用则它可用。大部分的工作都由 exclusive_swap_page()来完成，但是这个函数首先进行一些基本的检查以避免必须获取很多的锁。

```

259 int can_share_swap_page(struct page * page)
260 {
261     int retval = 0;
262
263     if (!PageLocked(page))
264         BUG();
265     switch (page_count(page)) {
266     case 3:
267         if (!page->buffers)
268             break;
269         /* Fallthrough */
270     case 2:
271         if (!PageSwapCache(page))
272             break;
273         retval = exclusive_swap_page(page);
274         break;
275     case 1:
276         if (PageReserved(page))
277             break;
278     }
279     retval = 1;

```

```

279 }
280 return retval;
281 }

```

263~264 这个函数从异常路径中调用,所以页面必须上锁。

265 按照引用次数进行分支选择。

266~268 如果计数为 3,但没有与之相关的缓冲区,则有多于一个的进程正在使用它。如果页面由一个交换文件而不是一个分区后援支持,则缓冲区可能仅与一个进程相关联。

270~273 如果计数仅为 2,但是它不是一个交换高速缓存的页面,则它没有槽可供共享,所以它返回 false。否则,它利用 exclusive_swap_page()(见 D. 5.5.3)进行完全检查。

276~277 如果页面是保留页面,则是一个全局 ZERO_PAGE,不能被共享。否则,可以确定这个页面就是仅有的这个 ZERO_PAGE。

D. 5.5.3 函数: exclusive_swap_page() (mm/swapfile.c)

这个函数检查进程是否是上锁交换页面的惟一用户。

```

229 static int exclusive_swap_page(struct page * page)
230 {
231     int retval = 0;
232     struct swap_info_struct * p;
233     swap_entry_t entry;
234
235     entry.val = page->index;
236     p = swap_info_get(entry);
237     if (p) {
238         /* Is the only swap cache user the cache itself? */
239         if (p->swap_map[SWP_OFFSET(entry)] == 1) {
240             /* Recheck the page count with the pagecache
241             * lock held.. */
242             spin_lock(&pagecache_lock);
243             if (page_count(page) - !page->buffers == 2)
244                 retval = 1;
245             spin_unlock(&pagecache_lock);
246         }
247     }
248     return retval;
249 }

```

231 缺省情况下,这里返回 false。

235 如 2.5 节所述, 页面的 `swp_entry_t` 存放于 `page->index` 中。

236 利用 `swap_info_get()`(见 K.2.3.1)来获取 `swap_info_struct`。

237~247 如果存在槽, 则这里检查我们是否是专门用户, 如果是则返回 `true`。

239 检查这个槽是否仅被高速缓存自身使用。如果是, 则需要利用获得的 `pagecache_lock` 再次检查页面计数。

242~243 如果存在缓冲区, 则 `page->buffers` 将赋值为 1。这样可以有效地检查该进程是否是页面的惟一用户。如果是 `retval` 设为 1, 则返回 `true`。

246 释放由 `swap_info_get()`(见 K.2.3.1)持有的槽引用。

D.5.6 写时复制(COW)页面

D.5.6.1 函数: `do_wp_page()` (`mm/memory.c`)

这个函数的调用图如图 4.16 所示。函数处理当一个用户试着向一个由多进程共享的私有页写的情形, 如在发生 `fork()` 后。基本操作是分配页面, 复制内容到新页, 旧页的共享计数减 1。

```
948 static int do_wp_page(struct mm_struct * mm,
949                         struct vm_area_struct * vma,
950                         unsigned long address, pte_t * page_table, pte_t pte)
951 {
952     struct page * old_page, * new_page;
953     old_page = pte_page(pte);
954     if (! VALID_PAGE(old_page))
955         goto bad_wp_page;
956 }
```

948~950 参数与 `handle_pte_fault()` 的相同。

954~955 在 PTE 中获取当前页面的引用, 并保证它有效。

```
957     if (! TryLockPage(old_page)) {
958         int reuse = can_share_swap_page(old_page);
959         unlock_page(old_page);
960         if (reuse) {
961             flush_cache_page(vma, address);
962             establish_pte(vma, address, page_table,
963                         pte_mkyoung(pte_mkdirty(pte_mkdirty(pte))));
964             spin_unlock(&mm->page_table_lock);
965         }
966     }
967 }
```

```

964 return 1; /* Minor fault */
965 }
966 }

```

957 首先试着锁住页面。如果返回 0，则意味着页面在前面已经被释放锁。

958 如果我们想上锁，这里调用 `can_share_swap_page()`（见 D.5.5.2）以确定我们是否是该页面交换槽的专门用户。如果是，则意味着我们是最后一个使 COW 失效的进程，这样我们可以简单地使用这一页，而不用再次分配一个新页。

960~965 如果我们是交换槽的惟一用户，则意味着我们是该页的惟一用户和最后一个使 COW 失效的进程。因此，可以重新构造 PTE，而我们返回一个次异常。

```

968 /*
969 * Ok, we need to copy. Oh, well..
970 */
971 page_cache_get(old_page);
972 spin_unlock(&mm->page_table_lock);
973
974 new_page = alloc_page(GFP_HIGHUSER);
975 if (!new_page)
976 goto no_mem;
977 copy_cow_page(old_page, new_page, address);
978

```

971 我们需要复制该页，所以首先获取对旧页的引用，在我们还没有完成之前，它不会消失。

972 因为我们将调用 `alloc_page()`（见 F.2.1 小节），而这可能睡眠，所以释放自旋锁。

974~976 分配页面，并保证成功返回一页。

977 猜测这个函数的操作是很容易的。如果页面打散为全局 0 页面，则使用 `clear_user_highpage()` 将其内容清 0。否则，`copy_user_highpage()` 复制实际的内容。

```

982 spin_lock(&mm->page_table_lock);
983 if (pte_same(*page_table, pte)) {
984 if (PageReserved(old_page))
985 ++mm->rss;
986 break_cow(vma, new_page, address, page_table);
987 lru_cache_add(new_page);
988
989 /* Free the old page.. */
990 new_page = old_page;
991 }

```

```

992 spin_unlock(&mm->page_table_lock);
993 page_cache_release(new_page);
994 page_cache_release(old_page);
995 return 1; /* Minor fault */

```

982 由于 alloc_page() (见 F. 2.1 小节) 释放页表锁, 这样重新获取页表锁。

983 保证在此期间 PTE 没有改变, 如果在释放自旋锁时发生了另外一个异常则它可能发生改变。

984~985 当 PageReserved() 为真时更新 RSS。这只会发生在如果陷入的页面是全局 ZERO_PAGE, 而且没有在 RSS 中计数的情况下。如果这是一个普通页面, 进程将在异常发生之后使用相同数量的物理帧。在以前, 除了 0 页面, 它将使用一个新页面帧, 所以 rss++ 反映新页面的使用情况。

986 break_cow() 负责调用体系结构的钩子函数来保证 CPU 高速缓存和 TLB 得到更新, 然后在 PTE 中建立一个新页面。它首先调用 flush_page_to_ram(), 这个函数在一个 struct page 将被放到用户空间时必须调用。下一步是调用 flush_cache_page() 清理来自于 CPU 高速缓存的页面。最后调用 sestablish_pte(), 它在 PTE 中建立一个新页面。

987 将页面加入到 LRU 链表中。

992 释放自旋锁。

993~994 撤销页面引用。

995 返回一个次异常。

```

996
997 bad_wp_page:
998 spin_unlock(&mm->page_table_lock);
999 printk("do_wp_page: bogus page at address %08lx (page 0x%lx)\n",
          address, (unsigned long)old_page);
1000 return -1;
1001 no_mem:
1002 page_cache_release(old_page);
1003 return -1;
1004 }

```

997~1000 这是一个错误的 COW 失效, 它仅在轻型内核中使用。这里打印一条提示信息, 然后返回。

1001~1003 这次页面分配失败, 所以这里释放对旧页面的引用并返回 -1。

D.6 页面相关的磁盘 I/O

D.6.1 一般文件读

这里更多的是 I/O 管理而不是 VM 的领域,但是由于它通过页面高速缓存完成操作,所以在这里我们简要讨论一番。虽然 generic_file_write() 在本书中并不讨论,但是实际上它与读操作一样。因此,如果你理解了读是如何发生的,那么写操作的理解也不会对你造成困难。

D.6.1.1 函数: generic_file_read() (mm/filemap.c)

这是一个用于任何文件系统从页面高速缓存读入页面的一般文件读函数。对普通 I/O 而言,它负责利用 do_generic_file_read() 和 file_read_actor() 来建立一个 read_descriptor_t。对直接 I/O 而言,这个函数是 generic_file_direct_I/O() 的封装。

```
1695 ssize_t generic_file_read(struct file * filp,
1696     char * buf, size_t count,
1697     loff_t * ppos)
1698 {
1699     ssize_t retval;
1700
1701     if ((ssize_t) count < 0)
1702         return -EINVAL;
1703
1704     if (filp->f_flags & O_DIRECT)
1705         goto o_direct;
1706
1707     retval = -EFAULT;
1708
1709     if (count) {
1710         read_descriptor_t desc;
1711
1712         desc.written = 0;
1713         desc.count = count;
1714         desc.buf = buf;
1715         desc.error = 0;
```

```
1716 do_generic_file_read(filp, ppos, &desc,
1717                         file_read_actor);
1718
1719         retval = desc.written;
1720         if (!retval)
1721             retval = desc.error;
1722         }
1723     out:
1724     return retval;
```

这一块有关普通文件 I/O。

1702~1703 如果这是一个直接 I/O，则跳转到 o_direct 标号。

1706 如果对用户空间页面具有写权限，则继续。

1709 如果 count 为 0, 则不进行 I/O。

1712~1715 组装一个 `read_descriptor_t` 结构, `file_read_actor()`(见 L. 3. 2. 3)要用到。

1716 进行文件读。

1718 从读描述符结构中提取写的字节数。

1719~1720 如果发生错误,则这里提取这种错误。

1724 返回读取的字节数或者发生的错误。

1725

1726 *q* direct.

1727

1728 loff t pos = * rpos, size:

1729 struct address_space * mapping =

filp->f_dentry->d_inode->i_mapping;

1730 struct inode * inode = mapping->host;

1731

1732 retval = 0;

1733 if (! count)

1734 goto out: /* skip atime */

1735 down read(&inode->i_a

1736 down(&inode->i_sem);

1737 size = inode->i

```
1738 if (pos < size) {
```

(READ, filp, 1)

```

1741 *ppos = pos + retval;
1742 }
1743 UPDATE_ATIME(filp->f_dentry->d_inode);
1744 goto out;
1745 }
1746 }

```

这一块有关直接 I/O。它主要负责提取 generic_file_direct_IO() 的参数。

1729 获取这个 struct file 用到的 address_space。

1733~1734 如果不再请求 I/O, 这里跳转到 out 以避免更新 inode 的访问次数。

1737 获取文件大小。

1738~1739 如果当前位置在文件结束之前, 则读是安全的, 并调用 generic_file_direct_IO()。

1740~1741 如果成功读, 则这里更新读指针在文件中的当前位置。

1743 更新访问次数。

1744 转到 out, 在那里返回 retval。

D. 6.1.2 函数: do_generic_file_read() (mm/filemap.c)

这是一般文件读操作的核心部分。它负责页面不在页面高速缓存中时分配页面。如果存在页面时则保证页面是最新的, 然后保证设置了合适大小的预读窗口。

```

1349 void do_generic_file_read(struct file * filp,
                               loff_t * ppos,
                               read_descriptor_t * desc,
                               read_actor_t actor)
1350 {
1351 struct address_space * mapping =
    filp->f_dentry->d_inode->i_mapping;
1352 struct inode * inode = mapping->host;
1353 unsigned long index, offset;
1354 struct page * cached_page;
1355 int reada_ok;
1356 int error;
1357 int max_readahead = get_max_readahead(inode);
1358
1359 cached_page = NULL;
1360 index = *ppos >> PAGE_CACHE_SHIFT;
1361 offset = *ppos & ~PAGE_CACHE_MASK;
1362

```

1357 获取该块设备的最大预读窗口大小。

1360 计算页面索引,其中包含有当前文件位置指针。

1361 计算持有当前文件位置指针的页面内偏移。

```

1363 /*
1364 * If the current position is outside the previous read-ahead
1365 * window, we reset the current read-ahead context and set read
1366 * ahead max to zero (will be set to just needed value later),
1367 * otherwise, we assume that the file accesses are sequential
1368 * enough to continue read-ahead.
1369 */
1370 if (index > filp->f_raend ||
1371     index + filp->f_rawin < filp->f_raend) {
1371     reada_ok = 0;
1372     filp->f_raend = 0;
1373     filp->f_ralen = 0;
1374     filp->f_ramax = 0;
1375     filp->f_rawin = 0;
1376 } else {
1377     reada_ok = 1;
1378 }
1379 /*
1380 * Adjust the current value of read-ahead max.
1381 * If the read operation stay in the first half page, force no
1382 * readahead. Otherwise try to increase read ahead max just
1383 * enough to do the read request.
1384 * Then, at least MIN_READAHEAD if read ahead is ok,
1385 * and at most MAX_READAHEAD in all cases.
1386 */
1386 if (!index && offset + desc->count <= (PAGE_CACHE_SIZE >> 1)) {
1387     filp->f_ramax = 0;
1388 } else {
1389     unsigned long needed;
1390
1391     needed = ((offset + desc->count) >> PAGE_CACHE_SHIFT) + 1;
1392
1393     if (filp->f_ramax < needed)
1394         filp->f_ramax = needed;
1395
1396     if (reada_ok && filp->f_ramax < vm_min_readahead)
1397         filp->f_ramax = vm_min_readahead;

```

```

1398 if (filp->f_ramax > max_readahead)
1399 filp->f_ramax = max_readahead;
1400 }

```

1370~1378 如注释所言,如果当前文件位置在当前预读窗口以外,则需要重置预读窗口。在这里重置为0,然后在需要的时候由 generic_file_readahead()(见 D. 6.1.3)调整。

1386~1400 如注释所言,如果在当前页的后半部分,则稍微地调整预读窗口。

```

1402 for (;;) {
1403     struct page * page, * * hash;
1404     unsigned long end_index, nr, ret;
1405
1406     end_index = inode->i_size >> PAGE_CACHE_SHIFT;
1407
1408     if (index > end_index)
1409         break;
1410     nr = PAGE_CACHE_SIZE;
1411     if (index == end_index) {
1412         nr = inode->i_size & ~PAGE_CACHE_MASK;
1413         if (nr <= offset)
1414             break;
1415     }
1416
1417     nr = nr - offset;
1418
1419     /*
1420      * Try to find the data in the page cache..
1421      */
1422     hash = page_hash(mapping, index);
1423
1424     spin_lock(&pagecache_lock);
1425     page = __find_page_nolock(mapping, index, *hash);
1426     if (!page)
1427         goto no_cached_page;

```

1402 这里的循环遍历每个需要满足读请求的页面。

1406 计算文件的末端在哪个页面。

1408~1409 如果当前索引已经超出末尾,则从这里退出,因为我们正试着读超过文件末尾的地方。

1410~1417 计算 nr 为在当前页还需要读的字节数。这一块考虑可能用到的文件最后一

页,以及页中当前文件位置。

1422~1425 在页面高速缓存中搜索这一页。

1426~1427 如果页面不在页面高速缓存中,则跳转到 no_cached_page,在那里分配一个页面。

```
1428 found_page:
1429 page_cache_get(page);
1430 spin_unlock(&pagecache_lock);
1431
1432 if (!Page_Uptodate(page))
1433 goto page_not_up_to_date;
1434 generic_file_readahead(reada_ok, filp, inode, page);
```

这一块,在页面高速缓存中已经找到该页面。

1429 引用页面高速缓存中的这一页,以使它不会过早地被释放。

1432~1433 如果页面不是最新的,则转到 page_not_up_to_date,在那里利用磁盘中的信息更新页面。

1434 利用 generic_file_readahead()(见 D. 6. 1. 3)完成预读文件。

```
1435 page_ok:
1436 /* If users can be writing to this page using arbitrary
1437 * virtual addresses, take care about potential aliasing
1438 * before reading the page on the kernel side.
1439 */
1440 if (mapping->i_mmap_shared != NULL)
1441 flush_dcache_page(page);
1442
1443 /*
1444 * Mark the page accessed if we read the
1445 * beginning or we just did an lseek.
1446 */
1447 if (!offset || !filp->f_reada)
1448 mark_page_accessed(page);
1449
1450 /*
1451 * Ok, we have the page, and it's up-to-date, so
1452 * now we can copy it to user space...
1453 *
1454 * The actor routine returns how many bytes were actually
1455 * used.. NOTE! This may not be the same as how much of a
```

```

1456 * user buffer we filled up (we may be padding etc), so we
1457 * can only update "pos" here (the actor routine has to
1458 * update the user buffer pointers and the remaining count).
1459 */
1460 ret = actor(desc, page, offset, nr);
1461 offset += ret;
1462 index += offset >> PAGE_CACHE_SHIFT;
1463 offset &= ~PAGE_CACHE_MASK;
1464
1465 page_cache_release(page);
1466 if (ret == nr && desc->count)
1467 continue;
1468 break;

```

在这一块,页面已经在页面高速缓存中,且已经准备好由文件读操作函数读。

1440~1441 由于其他用户可能在写这个页面,所以调用 flush_dcache_page()来保证所有的改变都可见。

1447~1448 由于刚才已经访问过该页面,所以这里调用 mark_page_accessed()(见 J. 2. 3. 1)来将其移到 active_list 中。

1460 调用操作函数。在这种情况下,操作函数是 file_read_actor()(见 L. 2. 3. 2),它负责从页面复制字节到用户空间。

1461 更新文件中的当前偏移。

1462 如果有必要,则移动到下一个页面。

1463 更新我们当前正在读的页面中的偏移。记住我们刚才已经进入了文件的下一个页面。

1465 释放我们对该页的引用。

1466~1468 如果还有数据读,这里再次循环以读下一个页面。否则,跳出,因为读操作已经完成。

```

1470 /*
1471 * Ok, the page was not immediately readable, so let's try to
1472 * read ahead while we're at it..
1473 */
1473 page_not_up_to_date;
1474 generic_file_readahead(reada_ok, filp, inode, page);
1475
1476 if (Page_Uptodate(page))
1477 goto page_ok;

```

```

1478
1479 /* Get exclusive access to the page ... */
1480 lock_page(page);
1481
1482 /* Did it get unhashed before we got the lock? */
1483 if (!page->mapping) {
1484 UnlockPage(page);
1485 page_cache_release(page);
1486 continue;
1487 }
1488
1489 /* Did somebody else fill it already? */
1490 if (Page_Uptodate(page)) {
1491 UnlockPage(page);
1492 goto page_ok;
1493 }

```

在这一块中,被读的页面并不是磁盘上最新的数据信息,调用 generic_file_readahead()更新当前页面和预读指针,因为这些都是 I/O 需要的。

1474 调用 generic_file_readahead()(见 D. 6.1.3),在需要的时候同步当前页面和预读指针。

1476~1477 如果页面现在是最新的,则转到 page_ok,在那里开始复制字节到用户空间。

1480 否则,预读指针上发生了改变,这里对页面上锁以进行排他访问。

1483~1487 如果在还未获得自旋锁的时候页面已经从页面高速缓存中移除,则这里释放对页面的引用,重新开始。在第二轮中,将分配页面并重新插入到页面高速缓存中。

1490~1493 如果某个人在我们没有上锁的页面上更新了页面,则这里再次解锁,然后转到 page_ok,开始读字节到用户空间。

```

1495 readpage;
1496 /* ... and start the actual read. The read will
   * unlock the page. */
1497 error = mapping->a_ops->readpage(filp, page);
1498
1499 if (!error) {
1500 if (Page_Uptodate(page))
1501 goto page_ok;
1502
1503 /* Again, try some read-ahead while waiting for
   * the page to finish. */
1504 generic_file_readahead(reada_ok, filp, inode, page);

```



```

1505 wait_on_page(page);
1506 if (Page_Uptodate(page))
1507 goto page_ok;
1508 error = -EIO;
1509 }
1510
1511 /* UHHUH! A synchronous read error occurred. Report it */
1512 desc->error = error;
1513 page_cache_release(page);
1514 break;

```

在这一块,预读与由 readpage()函数提供的 address_space 页面读同步。

1497 调用 address_space 文件系统特定的 readpage()函数。在多数情形下,这里最后将调用在 fs/buffer.c()中声明的 block_read_full_page()函数。

1499~1501 如果没有发生错误,页面也是最新的,则转到 page_ok 开始向用户空间读字节。

1504 否则,调度某个预读,因为我们被强制性地等待 I/O。

1505~1507 等待请求页上的 I/O 完成。如果成功完成,则转到 page_ok。

1508 否则,发生了错误,所以这里设置-EIO 返回到用户空间。

1512~1514 发生了 I/O 错误,记录错误,然后释放对当前页面的引用。这个错误将由 generic_file_read()(见 D. 6. 1. 1)从 read_descriptor_t 结构中获取。

```

1516 no_cached_page:
1517 /*
1518 * Ok, it wasn't cached, so we need to create a new
1519 * page..
1520 *
1521 * We get here with the page cache lock held.
1522 */
1523 if (!cached_page) {
1524 spin_unlock(&pagecache_lock);
1525 cached_page = page_cache_alloc(mapping);
1526 if (!cached_page) {
1527 desc->error = -ENOMEM;
1528 break;
1529 }
1530
1531 /*
1532 * Somebody may have added the page while we

```



```

1533 * dropped the page cache lock. Check for that.
1534 */
1535 spin_lock(&pagecache_lock);
1536 page = __find_page_nolock(mapping, index, *hash);
1537 if (page)
1538 goto found_page;
1539 }
1540
1541 /*
1542 * Ok, add the new page to the hash-queues...
1543 */
1544 page = cached_page;
1545 __add_to_page_cache(page, mapping, index, hash);
1546 spin_unlock(&pagecache_lock);
1547 lru_cache_add(page);
1548 cached_page = NULL;
1549
1550 goto readpage;
1551 }

```

在这一块,页面不在页面高速缓存中,所以这里分配一个页面并加入到页面高速缓存中。

1523~1539 如果还没有分配一个高速缓存页面,则这里分配一个,并保证在我们睡眠时其他人没有将其插入到页面高速缓存中。

1524 释放 pagecache_lock,因为 page_cache_alloc()可能睡眠。

1525~1529 分配一页,如果分配失败则设置-ENOMEM 为返回值。

1535~1536 再次获得 pagecache_lock,在页面高速缓存中搜索以保证在释放锁的时候其他进程没有将页面插入到页面高速缓存中。

1537 如果其他进程已经添加一个合适页面到页面高速缓存中,则这里跳转到 found_page,因为我们刚才分配的那一页已经不再需要了。

1544~1545 否则,在这里将我们刚才分配的那一页加入到页面高速缓存中。

1547 将页面加入到 LRU 链表中。

1548 设置 cached_page 为 NULL,因为它现在正在使用中。

1550 跳转到 readpage 以调用将从磁盘读的页面。

```

1552
1553 * ppos = ((loff_t) index << PAGE_CACHE_SHIFT) + offset;
1554 filp->f_reada = 1;
1555 if (cached_page)

```

```

1556 page_cache_release(cached_page);
1557 UPDATE_ATIME(inode);
1558 }

```

1553 更新我们在文件中的位置。

1555~1556 如果分配的一页对页面高速缓存已经多余,则不需要搜索,直接在这里释放该页面。

1557 更新文件的访问次数。

D. 6.1.3 函数: generic_file_readahead() (mm/filemap.c)

这个函数完成一般文件预读。预读部分是代码中很少见的着重注释的地方。推荐读者去读在 mm/filemap.c 中标记为“预读上下文”的注释。

```

1222 static void generic_file_readahead(int reada_ok,
1223 struct file * filp, struct inode * inode,
1224 struct page * page)
1225 {
1226     unsigned long end_index;
1227     unsigned long index = page->index;
1228     unsigned long max_ahead, ahead;
1229     unsigned long raend;
1230     int max_readahead = get_max_readahead(inode);
1231
1232     end_index = inode->i_size >> PAGE_CACHE_SHIFT;
1233
1234     raend = filp->f_raend;
1235     max_ahead = 0;

```

1227 以提供的 page 为基础获取开始的索引。

1230 获取这个块设备的最大预读指针。

1232 获取在文件末端的页面索引。

1234 获取 struct file 的预读窗口末端。

```

1236
1237 /*
1238 * The current page is locked.
1239 * If the current position is inside the previous read IO request,
1240 * do not try to reread previously read ahead pages.
1241 * Otherwise decide or not to read ahead some pages synchronously.
1242 * If we are not going to read ahead, set the read ahead context

```



```

1243 * for this page only.
1244 */
1245 if (PageLocked(page)) {
1246 if (!filp->f_ralen ||
1247     index >= raend ||
1248     index + filp->f_rawin < raend) {
1249 raend = index;
1250 if (raend < end_index)
1251     max_ahead = filp->f_ramax;
1252 filp->f_rawin = 0;
1253 filp->f_ralen = 1;
1254 if (!max_ahead) {
1255     filp->f_raend = index + filp->f_ralen;
1256     filp->f_rawin += filp->f_ralen;
1257 }

```

这一块碰到已经上锁的一页。所以这里必须确定是否暂时停止预读。

1245 如果当前页面因为 I/O 而被上锁，则这里检查当前页面是否在最后一个预读窗口，如果是，则没必要再次预读。如果不是，或者前面还没有进行预读，则这里更新预读上下文。

1246 首先检查预读在前面是否已经进行过。然后检查当前上锁的页面是否在上一次预读完成的地址之后。第 3 步检查当前上锁的页面是否在当前预读窗口中。

1247 更新预读窗口的末端。

1248~1249 如果预读窗口的末端不在文件末端之后，则这里设置 max_head 为预读的最大量，这个将在 struct file(filp->f_ramax) 中用到。

1250~1255 设置预读仅在当前页面发生，有效地停止预读。

```

1258 /*
1259 * The current page is not locked.
1260 * If we were reading ahead and,
1261 * if the current max read ahead size is not zero and,
1262 * if the current position is inside the last read-ahead IO
1263 * request, it is the moment to try to read ahead asynchronously.
1264 * We will later force unplug device in order to force
1265 * asynchronous read IO.
1266 else if (reada_ok && filp->f_ramax && raend >= 1 &&
1267 index <= raend && index + filp->f_ralen >= raend) {

```

```

1268 /*
1269 * Add ONE page to max_ahead in order to try to have about the
1270 * same IO maxsize as synchronous read-ahead
1271 * (MAX_READAHEAD + 1) * PAGE_CACHE_SIZE.
1272 * Compute the position of the last page we have tried to read
1273 * in order to begin to read ahead just at the next page.
1274 */
1275 if (raend < end_index)
1276 max_ahead = filp->f_ramax + 1;
1277
1278 if (max_ahead) {
1279 filp->f_rawin = filp->f_ralen;
1280 filp->f_ralen = 0;
1281 reada_ok = 2;
1282 }
1283 }

```

这是在代码注释方面很少见的情形:使代码尽可能得清晰。基本上,它的意思是说如果当前页面没有因为 I/O 而上锁,则稍微地扩大预读窗口,然后记住预读进展顺利。

```

1284 /*
1285 * Try to read ahead pages.
1286 * We hope that ll_rw_blk() plug/unplug, coalescence, requests
1287 * sort and the scheduler, will work enough for us to avoid too
1288 * * bad actuals IO requests.
1289 */
1290 ahead = 0;
1291 while (ahead < max_ahead) {
1292 ahead++;
1293 if ((raend + ahead) >= end_index)
1294 break;
1295 if (page_cache_read(filp, raend + ahead) < 0)
1296 break;

```

这一块通过对先前读窗口中的每个页面调用 page_cache_read()进行实际的预读。注意这里 ahead 是如何在每个预读页面上增加的。

```

1297 /*
1298 * If we tried to read ahead some pages,

```

```

1299 * If we tried to read ahead asynchronously,
1300 * Try to force unplug of the device in order to start an
1301 * asynchronous read IO request.
1302 * Update the read-ahead context.
1303 * Store the length of the current read-ahead window.
1304 * Double the current max read ahead size.
1305 * That heuristic avoid to do some large IO for files that are
1306 * not really accessed sequentially.
1307 */
1308 if (ahead) {
1309     filp->f_ralen += ahead;
1310     filp->f_rawin += filp->f_ralen;
1311     filp->f_raend = raend + ahead + 1;
1312
1313     filp->f_ramax += filp->f_ramax;
1314
1315     if (filp->f_ramax > max_readahead)
1316         filp->f_ramax = max_readahead;
1317
1318 #ifdef PROFILE_READAHEAD
1319     profile_readahead((reada_ok == 2), filp);
1320 #endif
1321 }
1322
1323 return;
1324 }

```

如果预读成功，则这里更新在 struct file 中的预读字段来标识进度。基本上是预读上下文在增长，但是如果发现预读效率低下，可以由 do_generic_file_readahead() 重置。

1309 更新在这一遍中预读的页面数 f_ralen。

1310 更新预读窗口的大小。

1311 标记预读的末端。

1313 将当前的预读最大大小扩大一倍。

1315~1316 不使预读的最大大小超过块设备中的最大预读大小。

D.6.2 一般文件 mmap()

D.6.2.1 函数: generic_file_mmap() (mm/filemap.c)

这是由许多 struct files 用作 struct file_operations 的一般 mmap() 函数。它主要负责存在合适的 address_space 函数，并设置 VMA 的使用操作。

```

2249 int generic_file_mmap(struct file * file,
                           struct vm_area_struct * vma)
2250 {
2251 struct address_space * mapping =
    file->f_dentry->d_inode->i_mapping;
2252 struct inode * inode = mapping->host;
2253
2254 if ((vma->vm_flags & VM_SHARED) &&
    (vma->vm_flags & VM_MAYWRITE)) {
2255 if (! mapping->a_ops->writepage)
2256 return -EINVAL;
2257 }
2258 if (! mapping->a_ops->readpage)
2259 return -ENOEXEC;
2260 UPDATE_ATIME(inode);
2261 vma->vm_ops = &generic_file_vm_ops;
2262 return 0;
2263 }
```

2251 获取管理被映射文件的 address_space。

2252 获取该 address_space 的 struct inode。

2254~2257 如果 VMA 可以共享和可写，则这里保证存在一个 a_ops->writepage() 函数。如果没有，则返回-EINVAL。

2258~2259 保证存在一个 a_ops->readpage() 函数。

2260 更新 inode 的访问次数。

2261 使用 generic_file_vm_ops 进行文件操作。在 mm/filemap.c 中定义的一般 VM 操作结构，仅提供 filemap_nopage() (见 D.6.4.1) 作为它的 nopage() 函数。这里没有定义其他的回调函数。

D.6.3 一般文件截断

这一节有关截断文件的路径。实际的系统调用 truncate() 由 fs/open.c 中的 sys_truncate()

来实现。在 VM 中的高层函数(vmtruncate())调用时,文件的目录项信息已经被更新,而且还获得了 inode 的信号量。

D. 6.3.1 函数: vmtruncate() (mm/memory.c)

这是一个负责截断文件的 VM 高层函数。当它完成时,所有映射页面的页表项都被截断,解除映射并在可能时回收。

```

1042 int vmtruncate(struct inode * inode, loff_t offset)
1043 {
1044     unsigned long pgoff;
1045     struct address_space * mapping = inode->i_mapping;
1046     unsigned long limit;
1047
1048     if (inode->i_size < offset)
1049         goto do_expand;
1050     inode->i_size = offset;
1051     spin_lock(&mapping->i_shared_lock);
1052     if (!mapping->i_mmap && !mapping->i_mmap_shared)
1053         goto out_unlock;
1054
1055     pgoff = (offset + PAGE_CACHE_SIZE - 1) >> PAGE_CACHE_SHIFT;
1056     if (mapping->i_mmap != NULL)
1057         vmtruncate_list(mapping->i_mmap, pgoff);
1058     if (mapping->i_mmap_shared != NULL)
1059         vmtruncate_list(mapping->i_mmap_shared, pgoff);
1060
1061     out_unlock:
1062     spin_unlock(&mapping->i_shared_lock);
1063     truncate_inode_pages(mapping, offset);
1064     goto out_truncate;
1065
1066     do_expand:
1067     limit = current->rlim[RLIMIT_FSIZE].rlim_cur;
1068     if (limit != RLIM_INFINITY && offset > limit)
1069         goto out_sig;
1070     if (offset > inode->i_sb->s_maxbytes)
1071         goto out;
1072     inode->i_size = offset;
1073
1074     out_truncate:

```

```

1075 if (inode->i_op && inode->i_op->truncate) {
1076     lock_kernel();
1077     inode->i_op->truncate(inode);
1078     unlock_kernel();
1079 }
1080 return 0;
1081 out_sig:
1082 send_sig(SIGXFSZ, current, 0);
1083 out:
1084 return -EFBIG;
1085 }

```

1042 传入的参数是被截断的 inode, 标记文件新末端的 offset。文件以前的长度存放在 inode->i_size。

1045 获取 inode 的 address_space。

1048~1049 如果新文件大小大于旧的大小, 则跳转到 do_expand, 在扩大文件前进行进程极限检查。

1050 在这里收缩文件, 所以更新 inode->i_size 来匹配这种收缩。

1051 上锁自旋锁, 保护使用该 inode 的两个 VMA 链表。

1052~1053 如果没有 VMA 映射到该 inode, 则跳转到 out_unlock。在那里文件使用的页面将被 truncate_inode_pages() (见 D. 6. 3. 6) 回收。

1055 计算作为文件偏移的 pgoff, 它是截断开始的页面。

1056~1057 调用 vmtruncate_list() (见 D. 6. 3. 2) 从所有私有映射中截断页面。

1058~1059 从所有的共享映射中截断页面。

1062 解锁保护 VMA 链表的自旋锁。

1063 如果在文件的页面高速缓存中存在页面, 则调用 truncate_inode_pages() (见 D. 6. 3. 6) 回收页面。

1064 跳转到 out_truncate 来调用特定文件系统的 truncate() 函数, 这样磁盘上使用的块将被释放。

1066~1071 如果扩展了文件, 这里保证将不会超过最大文件大小的进程限制, 而且宿主文件系统可以支持这种新的文件大小。

1072 如果符合限制, 则更新 inode 大小, 并回退来调用特定文件系统的截断函数, 用 0 填充扩展的文件大小。

1075~1079 如果文件系统提供一个 truncate() 函数, 则上锁内核, 调用它, 并再次释放内核锁。没有获取合适锁的文件系统将在文件截断和文件扩展之间阻止这种竞争, 因为在写或陷入前必须需要一个大内核锁。

1080 返回成功。

1082~1084 如果文件大小变得过大,则给调用进程发送 SIGXFSZ 信号并能够返回-EFBIG。

D. 6.3.2 函数: vmtruncate_list() (mm/memory.c)

这个函数遍历在一个 address_spaces 链表中的所有 VMA,并在映射正被截断文件的地址范围内调用 zap_page_range()。

```

1006 static void vmtruncate_list(struct vm_area_struct * mpnt,
1007                           unsigned long pgoff)
1008 {
1009     struct mm_struct * mm = mpnt->vm_mm;
1010     unsigned long start = mpnt->vm_start;
1011     unsigned long end = mpnt->vm_end;
1012     unsigned long len = end - start;
1013     unsigned long diff;
1014
1015     /* mapping wholly truncated? */
1016     if (mpnt->vm_pgoff >= pgoff) {
1017         zap_page_range(mm, start, len);
1018         continue;
1019     }
1020
1021     /* mapping wholly unaffected? */
1022     len = len >> PAGE_SHIFT;
1023     diff = pgoff - mpnt->vm_pgoff;
1024     if (diff >= len)
1025         continue;
1026
1027     /* Ok, partially affected.. */
1028     start += diff << PAGE_SHIFT;
1029     len = (len - diff) << PAGE_SHIFT;
1030     zap_page_range(mm, start, len);
1031 } while ((mpnt = mpnt->vm_next_share) != NULL);
1032 }
```

1008~1031 遍历链表中的所有 VMA。

1009 获取该 VMA 持有的 mm_struct。

1010~1012 计算 VMA 起点、终点和长度。

1016~1019 如果截断了整个 VMA, 这里调用函数 zap_page_page() (见 D. 6. 3. 3), 以整个 VMA 的起点和长度为参数。

1022 计算以页面为单位的 VMA 长度。

1023~1025 检查 VMA 映射的任何区域是否可以被截断。如果这个 VMA 不受影响则转到下一个 VMA。

1028~1029 如果 VMA 只是部分被截断, 则计算要截断区域的起点和以页面为单位的长度。

1030 调用 zap_page_range() (见 D. 6. 3. 3) 来取消受影响区域的映射。

D. 6. 3. 3 函数: zap_page_range() (mm/memory.c)

这个函数是遍历页表的顶层函数, 它从一个 mm_struct 中取消对特定区域中用户空间的映射。

```

360 void zap_page_range(struct mm_struct * mm,
361                      unsigned long address, unsigned long size)
362 {
363     mmu_gather_t * tlb;
364     pgd_t * dir;
365     unsigned long start = address, end = address + size;
366     int freed = 0;
367
368     dir = pgd_offset(mm, address);
369     /*
370      * This is a long-lived spinlock. That's fine.
371      * There's no contention, because the page table
372      * lock only protects against kswapd anyway, and
373      * even if kswapd happened to be looking at this
374      * process we _want_ it to get stuck.
375     */
376     if (address >= end)
377         BUG();
378     spin_lock(&mm->page_table_lock);
379     flush_cache_range(mm, address, end);
380     tlb = tlb_gather_mmu(mm);
381
382     do {
383         freed += zap_pmd_range(tlb, dir, address, end - address);
384         address = (address + PGDIR_SIZE) & PGDIR_MASK;

```

```

385 dir++;
386 } while (address && (address < end));
387
388 /* this will flush any remaining tlb entries */
389 tlb_finish_mmu(tlb, start, end);
390
391 /*
392 * Update rss for the mm_struct (not necessarily current->mm)
393 * Notice that rss is an unsigned long.
394 */
395 if (mm->rss > freed)
396 mm->rss -= freed;
397 else
398 mm->rss = 0;
399 spin_unlock(&mm->page_table_lock);
400 }

```

364 计算销毁的 start 和 end 地址。

367 计算包含起始 address 的 PGD(dir)。

376~377 保证起始地址不在末尾地址之后。

378 获取保护页表的自旋锁。这是一个长生命期的锁,一般认为这不是一个好方法,但是在这块前面的注释解释了在这种情形下的可行性。

379 清理这片区域的 CPU 高速缓存。

380 tlb_gather_mmu()记录了正在被改变的 MM。最后将调用 tlb_remove_page()来取消 PTE 映射,并将 PTE 存放在一个 struct free_pte_ctx 中直到销毁过程结束。这里是为了避免必须不断地因为释放 PTE 而要刷新 TLB。

382~386 对由于销毁而受影响的每个 PMD,这里调用 zap_pmd_range()直到到达末端地址。注意这里也传入了 tlb 参数,在后面 tlb_remove_page()将用到。

389 tlb_finish_mmu()释放所有由 tlb_remove_page()解除映射的 PTE,然后刷新 TLB。这样刷新是为了避免 TLB 刷新风暴,否则需要解除映射的每个 PTE。

395~398 更新 RSS 计数。

399 释放页表锁。

D. 6.3.4 函数:zap_pmd_range() (mm/memory.c)

这个函数没有加注释。它遍历在请求范围内受影响的 PMD,然后在每个前面调用 zap_pte_range()。

```
331 static inline int zap_pmd_range(mmu_gather_t * tlb, pgd_t * dir,
```

```

    unsigned long address,
    unsigned long size)

332 {
333 pmd_t * pmd;
334 unsigned long end;
335 int freed;
336
337 if (pgd_none(*dir))
338 return 0;
339 if (pgd_bad(*dir)) {
340 pgd_ERROR(*dir);
341 pgd_clear(dir);
342 return 0;
343 }
344 pmd = pmd_offset(dir, address);
345 end = address + size;
346 if (end > ((address + PGDIR_SIZE) & PGDIR_MASK))
347 end = ((address + PGDIR_SIZE) & PGDIR_MASK);
348 freed = 0;
349 do {
350 freed += zap_pte_range(tlb, pmd, address, end - address);
351 address = (address + PMD_SIZE) & PMD_MASK;
352 pmd++;
353 } while (address < end);
354 return freed;
355 }

```

337~338 如果不存在 PGD, 这里返回。

339~343 如果 PGD 损坏, 则表明是错误并返回。

344 获取起始 pmd。

345~347 计算销毁的 end 地址。如果它已经超过了这个 PGD 的末端, 则设置 end 为 PGD 的末端。

349~353 遍历该 PGD 中所有 PMD, 对每个 PMD, 这里调用 zap_pte_range() (见 D. 6. 3. 5) 来取消对 PTE 的映射。

354 返回释放的页面数。

D. 6. 3. 5 函数:zap_pte_range() (mm/memory.c)

这个函数对请求地址范围中的 pmd 中每个 PTE, 调用 tlb_remove_page()。

```

294 static inline int zap_pte_range(mmu_gather_t * tlb, pmd_t * pmd,
295                                 unsigned long address,
296                                 unsigned long size)
297 {
298     unsigned long offset;
299     pte_t * ptep;
300     int freed = 0;
301
302     if (pmd_none(*pmd))
303         return 0;
304     if (pmd_bad(*pmd)) {
305         pmd_ERROR(*pmd);
306         pmd_clear(pmd);
307         return 0;
308     }
309     offset = address & ~PMD_MASK;
310     if (offset + size > PMD_SIZE)
311         size = PMD_SIZE - offset;
312     size &= PAGE_MASK;
313     for (offset = 0; offset < size; ptep++, offset += PAGE_SIZE) {
314         pte_t pte = *ptep;
315         if (pte_none(pte))
316             continue;
317         struct page * page = pte_page(pte);
318         if (VALID_PAGE(page) && !PageReserved(page))
319             freed++;
320         /* This will eventually call __free_pte on the pte. */
321         tlb_remove_page(tlb, ptep, address + offset);
322     } else {
323         free_swap_and_cache(pte_to_swp_entry(pte));
324         pte_clear(ptep);
325     }
326 }
327
328     return freed;
329 }

```

300~301 如果不存在 PGD, 这里返回。

- 302~306 如果 PGD 损坏,则表明是错误并返回。
- 307 获取起始 PTE 偏移。
- 308 将偏移对齐于 PMD 边界。
- 309 如果要解除映射的区域大小是以前的 PMD 边界,在这里指定大小,这样仅这个 PMD 会受到影响。
- 311 将 size 对齐于页面边界。
- 312~326 遍历区域中所有 PTE。
- 314~315 如果不存在 PTE,这里继续下一个。
- 316~322 如果存在 PTE,这里调用 tlb_remove_page() 来取消页面映射。如果页面可回收,则增加 freed 计数。
- 322~325 如果 PTE 正在使用中,而页面被换出或者在交换高速缓存中,这里利用 free_swap_and_cache()(见 K.3.2.3) 释放交换槽和页面。如果在高速缓存中的页面没有在这里计数,它可能被回收,但这不是最重要的。
- 328 返回释放的页面数。

D.6.3.6 函数: truncate_inode_pages() (mm/filemap.c)

这是一个负责截断所有来自 mapping 中 lstart 之后出现的页面高速缓存中所有页面的高层函数。

```

327 void truncate_inode_pages(struct address_space * mapping,
                           loff_t lstart)
328 {
329     unsigned long start = (lstart + PAGE_CACHE_SIZE - 1) >>
                           PAGE_CACHE_SHIFT;
330     unsigned partial = lstart & (PAGE_CACHE_SIZE - 1);
331     int unlocked;
332
333     spin_lock(&pagecache_lock);
334     do {
335         unlocked = truncate_list_pages(&mapping->clean_pages,
336                                         start, &partial);
336         unlocked |= truncate_list_pages(&mapping->dirty_pages,
337                                         start, &partial);
337         unlocked |= truncate_list_pages(&mapping->locked_pages,
338                                         start, &partial);
338     } while (unlocked);
339     /* Traversed all three lists without dropping the lock */
340     spin_unlock(&pagecache_lock);

```

341 }

329 计算作为页面索引的截断的 start 位置。

330 如果是部分截断则作为计算最后一页中偏移的 partial。

333 上锁页面高速缓存。

334 这里一直循环直到不再调用 truncate_list_pages(), 它的返回表明找到的任何页都已经回收。

335 使用 truncate_list_pages()(见 D. 6. 3. 7) 截断在 clean_pages 链表中的所有页面。

336 相似地, 截断在 dirty_pages 链表中的页面。

337 相似地, 截断在 locked_pages 链表中的页面。

340 解锁页面高速缓存。

D. 6. 3. 7 函数: truncate_list_pages() (mm/filemap.c)

这个函数找到请求链表(head), 它是 address_space 的一部分。如果在 start 之后找到页面, 它们都将被截断。

```

259 static int truncate_list_pages(struct list_head * head,
260                                unsigned long start,
261                                unsigned * partial)
262
263 struct list_head * curr;
264 struct page * page;
265 int unlocked = 0;
266
267 restart:
268 curr = head->prev;
269 while (curr != head) {
270     unsigned long offset;
271     page = list_entry(curr, struct page, list);
272     offset = page->index;
273     /* Is one of the pages to truncate? */
274     if ((offset >= start) ||
275         (*partial && (offset + 1) == start)) {
276         int failed;
277         page_cache_get(page);
278         failed = TryLockPage(page);

```

```

279
280 list_del(head);
281 if (! failed)
282 /* Restart after this page */
283 list_add_tail(head, curr);
284 else
285 /* Restart on this page */
286 list_add(head, curr);
287
288 spin_unlock(&pagecache_lock);
289 unlocked = 1;
290
291 if (! failed) {
292 if (*partial && (offset + 1) == start) {
293 truncate_partial_page(page, *partial);
294 *partial = 0;
295 } else
296 truncate_complete_page(page);
297
298 UnlockPage(page);
299 } else
300 wait_on_page(page);
301
302 page_cache_release(page);
303
304 if (current->need_resched) {
305 __set_current_state(TASK_RUNNING);
306 schedule();
307 }
308
309 spin_lock(&pagecache_lock);
310 goto restart;
311 }
312 curr = curr->prev;
313 }
314 return unlocked;
315 }

```

266~267 记录链表的开始并循环, 直到扫描了整个链表。

270~271 获取该表项的页面, 以及它所表示的文件中的 offset。

274 如果当前的页面是在 start 之后,或者是部分截断的页面,则截断该页面并移到下一个页面。

277~278 引用页面并试着对它上锁。

280 从链表中移除页面。

281~283 如果我们上锁页面,则这里将其添加回链表,在那里它将在下次循环时被跳过。

284~286 如果不是这样,则这里将其加回去,这样它就可以被立即发现。在这个函数后面,将调用 `wait_on_page()` 直到页面解锁。

288 释放页面高速缓存锁。

289 设置上锁为 1 表明已经找到了截断错误的该页。这里将强制使用 `truncate_inode_pages()` 再次调用这个函数以保证后面不再有页面。这里看似有点多余,实际上是想在找到一个上锁页后重新调用这个函数。但是它以这种实现的方式表明无论页面是否上锁它都会被调用。

291~299 如果我们对页面加了锁,这里截断它。

292~294 如果页面是部分地被截断,这里调用 `truncate_partial_page()`(见 D. 6. 3. 10),以截断开始页面中的偏移为参数。

296 如果不是部分截断,则这里调用 `truncate_complete_page()`(见 D. 6. 3. 8)来截断整个页面。

298 解锁页面。

300 如果页面上锁失败,这里调用 `wait_on_page()` 等待直到页面可以上锁。

302 释放对页面的引用。如果没有对页面的引用,则这里回收该页面。

304~307 检查进程是否应该在继续前调用 `schedule()`。这是为了避免在运行 CPU 时中断一个截断进程。

309 重新获得自旋锁并重新扫描页面以回收。

312 当前的页面应该不能回收,所以这里移到下一个页面。

314 如果在列表中找到一个待截断的页面,则这里返回 1。

D. 6. 3. 8 函数: `truncate_complete_page()` (`mm/filemap.c`)

这个函数截断一整个页面,释放相关的资源并回收页面。

```
239 static void truncate_complete_page(struct page * page)
240 {
241 /* Leave it on the LRU if it gets converted into
242 * anonymous buffers */
243 if (!page->buffers || do_flushpage(page, 0))
244 lru_cache_del(page);
```

```
245 /*  
246 * We remove the page from the page cache after we have  
247 * destroyed all buffer-cache references to it. Otherwise  
248 * some other process might think this inode page is not in  
249 * the page cache and creates a buffer-cache alias to it  
250 * causing all sorts of fun problems ...  
251 */  
252 ClearPageDirty(page);  
253 ClearPageUptodate(page);  
254 remove_inode_page(page);  
255 page_cache_release(page);  
256 }
```

242 如果页面拥有缓冲,则这里调用 `do_flushpage()`(见 D. 6. 3. 9)来清理页面相关的所有缓冲。下一行注释清楚地描述了这个问题。

243 从 LRU 中删除页面。

252~253 清除脏的和更新页面的标志。

254 调用 `remove_inode_page()`(见 J. 1. 2. 1)来从页面高速缓存中删除页面。

255 撤销页面引用。在 `truncate_list_pages()` 释放它的私有引用之后，该页将在后面被回收。

D. 6.3.9 函数: `do_flushpage()` (mm/filemap.c)

这个函数负责清理页面相关的所有缓冲。

```
223 static int do_flushpage(struct page * page, unsigned long offset)
224 {
225     int (* flushpage) (struct page * , unsigned long);
226     flushpage = page->mapping->a_ops->flushpage;
227     if (flushpage)
228         return (* flushpage)(page, offset);
229     return block_flushpage(page, offset);
230 }
```

226~228 如果 page->mapping 提供 flushpage() 函数, 这里调用它。

229 如果没有,则这里调用 `block_filepage()`,它是一个清理页面相关缓冲的一般函数。

D. 6.3.10 函数: truncate_partial_page() (mm/filemap.c)

这个函数利用不再使用的高位字节清 0 来部分截断一个页面，然后清理所有的相关缓冲。

```
234 memclear_highpage_flush(page, partial, PAGE_CACHE_SIZE-partial);  
235 if (page->buffers)  
236 do_flushpage(page, partial);  
237 }
```

234 memclear_highpage_flush()利用 0 填充地址范围。在这种情形下,它将从 partial 到
页面末端清 0。

235~236 如果页面拥有任何的相关缓冲，则这里清理包含在截断后区域的数据缓冲。

D. 6.4 从页面高速缓存中读入页面

D. 6.4.1 函数: filemap_nopage() (mm/filemap.c)

这是许多 VMA 使用的一般 `nopage()` 函数。这里利用大量的 `goto` 来循环，所以跟踪起来比较困难，但在这里并没有什么新颖的地方。它主要负责从页面高速缓存中提取错页面或者从磁盘中读入错页面。如果需要，它还可以进行预读。

这一块获取 struct file, address_space 和 inode, 这对缺页中断很重要。然后获取陷入需要的文件中起始偏移, 以及该 VMA 末端相应的偏移。偏移是 VMA 的末端, 而不是在预读文件时的页面末端。

1997~1999 获取陷入所需的 struct file, address_space 和 inode。

2003 计算 pgoff, 它是对于陷入起点的文件中的偏移。

2004 计算对应于 VMA 末端的文件中的偏移。

```

2006 retry_all:
2007 /*
2008 * An external ptracer can access pages that normally aren't
2009 * accessible..
2010 */
2011 size = (inode->i_size + PAGE_CACHE_SIZE - 1) >> PAGE_CACHE_SHIFT;
2012 if ((pgoff >= size) && (area->vm_mm == current->mm))
2013 return NULL;
2014
2015 /* The "size" of the file, as far as mmap is concerned, isn't
2016 bigger than the mapping */
2017 if (size > endoff)
2018 size = endoff;
2019 /*
2020 * Do we have something in the page cache already?
2021 */
2022 hash = page_hash(mapping, pgoff);
2023 retry_find:
2024 page = __find_get_page(mapping, pgoff, hash);
2025 if (!page)
2026 goto no_cached_page;
2027
2028 /*
2029 * Ok, found a page in the page cache, now we need to check
2030 * that it's up-to-date.
2031 */
2032 if (!Page_Uptodate(page))
2033 goto page_not_uptodate;

```

2011 计算文件的大小(以页数计)。

2012 如果陷入 pgoff 已经超出了文件末端, 则这里不是一个跟踪进程, 所以这里返回 NULL。

2016~2017 如果 VMA 映射超出了文件末端, 这里设置文件大小为映射的大小。

2022~2024 在页面高速缓存中查找该页。

2025~2026 如果不存在, 则跳转到 no_cached_page, 在那里调用 page_cache_read() 从后

援存储器中读入该页。

2032~2033 如果页面不是最新,则跳转到 page_not_uptodate,在那里页面或者被声明为无效,或者更新页面中的数据。

```

2035 success;
2036 /*
2037 * Try read-ahead for sequential areas.
2038 */
2039 if (VM_SequentialReadHint(area))
2040 nopal_sequential_readahead(area, pgoff, size);
2041
2042 /*
2043 * Found the page and have a reference on it, need to check
2044 * sharing and possibly copy it over to another page..
2045 */
2046 mark_page_accessed(page);
2047 flush_page_to_ram(page);
2048 return page;
2049

```

2039~2040 如果 VM_SEQ_READ 指明了映射,则利用 nopal_sequential_readahead() 对当前陷入的页面进行预陷入。

2046 将陷入页面标记为已访问,该页面将被移到 active_list。

2047 因为页面将被添加到进程页表中,所以这里调用 flush_page_to_ram(),这样内存对页面的最近存储将肯定可以为用户空间所见。

2048 返回陷入页面。

```

2050 no_cached_page;
2051 /*
2052 * If the requested offset is within our file, try to read
2053 * a whole cluster of pages at once.
2054 *
2055 * Otherwise, were off the end of a privately mapped file,
2056 * so we need to map a zero page.
2057 */
2058 if ((pgoff < size) && ! VM_RandomReadHint(area))
2059 error = read_cluster_nonblocking(file, pgoff, size);
2060 else
2061 error = page_cache_read(file, pgoff);
2062

```

```

2063 /*
2064 * The page we want has now been added to the page cache.
2065 * In the unlikely event that someone removed it in the
2066 * meantime, we'll just come back here and read it again.
2067 */
2068 if (error >= 0)
2069 goto retry_find;
2070
2071 /*
2072 * An error return from page_cache_read can result if the
2073 * system is low on memory, or a problem occurs while trying
2074 * to schedule I/O.
2075 */
2076 if (error == -ENOMEM)
2077 return NOPAGE_OOM;
2078 return NULL;

```

2058~2059 如果还没有到达文件的末端以及还未指明随机读,则这里调用 read_cluster_nonblocking()在这个陷入页面附近的几个页面中进行预陷入。

2061 如果没有,且文件是被随机访问的,则这里调用 page_cache_read()(见 D. 6. 4. 2)来读入刚陷入的页面。

2068~2069 如果没有错误发生,则转到 1958 行的 retry_find,在那里检查以保证返回前页面在页面高速缓存中。

2076~2077 如果由于内存溢出发生错误,这里返回,异常处理程序将相应进行处理。

2078 如果不是这个错误,这里返回 NULL 表明陷入一个不存在的页面,发送 SIGBUS 给陷入页面。

```

2080 page_not_upToDate;
2081 lock_page(page);
2082
2083 /* Did it get unhashed while we waited for it? */
2084 if (!page->mapping) {
2085     UnlockPage(page);
2086     page_cache_release(page);
2087     goto retry_all;
2088 }
2089
2090 /* Did somebody else get it up-to-date? */
2091 if (Page_Uptodate(page)) {

```



```

2092 UnlockPage(page);
2093 goto success;
2094 }
2095
2096 if (! mapping->a_ops->readpage(file, page)) {
2097 wait_on_page(page);
2098 if (Page_Uptodate(page))
2099 goto success;
2100 }

```

在这一块,找到的页面不是最新的,所以检查页面是否被更新过。如果通过,则调用合适的 readpage() 来重新同步页面。

2081 为 I/O 锁定页面。

2084~2088 如果页面从映射中移除(可能是因为文件截断),现在处于匿名状态,则这里转到 retry_all,在那里试着再次陷入页面。

2090~2094 再次检查 Uptodate 标志以防我们为 I/O 而锁定页面时页面已经改变。

2096 调用函数 address_space->readpage() 来从磁盘调度数据读。

2097 等待 I/O 完成,如果现在是最新的,则转到 success 返回页面。如果函数 readpage() 调用失败,则后退到错误恢复路径。

```

2101
2102 /*
2103 * Umm, take care of errors if the page isn't up-to-date.
2104 * Try to re-read it _once_. We do this synchronously,
2105 * because there really aren't any performance issues here
2106 * and we need to check for errors.
2107 */
2108 lock_page(page);
2109
2110 /* Somebody truncated the page on us? */
2111 if (! page->mapping) {
2112 UnlockPage(page);
2113 page_cache_release(page);
2114 goto retry_all;
2115 }
2116
2117 /* Somebody else successfully read it in? */
2118 if (Page_Uptodate(page)) {
2119 UnlockPage(page);

```

```

2120 goto success;
2121 }
2122 ClearPageError(page);
2123 if (! mapping->a_ops->readpage(file, page)) {
2124 wait_on_page(page);
2125 if (Page_Uptodate(page))
2126 goto success;
2127 }
2128
2129 /* Things didn't work out. Return zero to tell the
2130 * mm layer so, possibly freeing the page cache page first.
2131 */
2132
2133 page_cache_release(page);
2134 return NULL;
2135 }

```

在这条路径上,由于 I/O 错误,页面不是最新的。这里第 2 次试着读页面数据,如果失败则返回。

2110~2127 这与前一块几乎相同。惟一的区别是调用 ClearPageError() 来清除由前面 I/O 引起的错误。

2133 如果还是失败,则这里撤销对页面的引用,因为该页面已经没有用了。

2134 因为陷入失败所以返回 NULL。

D. 6.4.2 函数: page_cache_read() (mm/filemap.c)

如果页面不在页面高速缓存中,则这个函数将与 file 中 offset 相关的页面加入到页面高速缓存中。

```

702 static int page_cache_read(struct file * file,
                           unsigned long offset)
703 {
704 struct address_space * mapping =
    file->f_dentry->d_inode->i_mapping;
705 struct page ** hash = page_hash(mapping, offset);
706 struct page * page;
707
708 spin_lock(&pagecache_lock);
709 page = __find_page_nolock(mapping, offset, * hash);
710 spin_unlock(&pagecache_lock);

```



```

711 if (page)
712 return 0;
713
714 page = page_cache_alloc(mapping);
715 if (!page)
716 return -ENOMEM;
717
718 if (!add_to_page_cache_unique(page, mapping, offset, hash)) {
719 int error = mapping->a_ops->readpage(file, page);
720 page_cache_release(page);
721 return error;
722 }
723 /*
724 * We arrive here in the unlikely event that someone
725 * raced with us and added our page to the cache first.
726 */
727 page_cache_release(page);
728 return 0;
729 }

```

704 获取管理文件映射的 address_space。

705 页面高速缓存是一个哈希表,page_hash()返回的是相应 mapping 和 offset 桶中第 1 个页面。

708~709 利用 __find_page_nolock()(见 J. 1. 4. 3)来搜索页面高速缓存。这里基本上是从 hash 开始遍历链表以确定是否可以找到请求页。

711~712 如果页面已经在页面高速缓存中,则这里返回。

714 分配一页并将其插入到页面高速缓存中,page_cache_alloc()使用 mapping 中含的 GFP 信息从伙伴分配器中分配一个页面。

718 利用 add_to_page_cache_unique()(见 J. 1. 1. 2)将页面插入到页面高速缓存中。使用这个函数是因为需要再次检查,以确定在没有获得 pagecache_lock 自旋锁时页面没有被插入到页面高速缓存中。

719 如果分配的页面没有插入到页面高速缓存中,则需要用数据填充。所以调用该 mapping 的 readpage()函数。这里发生 I/O 调度,在 I/O 完成后将解锁页面。

720 add_to_page_cache_unique()(见 J. 1. 1. 2)中对加入页面高速缓存中页面的额外引用将在这里撤销,页面将不会被释放。

727 如果其他进程向页面高速缓存加入页面,则在这里由 page_cache_release()释放,因为现在没有该页面的用户。

D. 6.5 为 `nopage()` 进行预读文件

D. 6.5.1 函数: `nopage_sequential_readahead()` (`mm/filemap.c`)

这个函数仅在 VMA 中已经指定了 `VM_SEQ_READ` 标志时才由 `filemap_nopage()` 调用。当半个当前预读窗口已经陷入时, 将为 I/O 调度下一个预读窗口, 前面一个窗口中的页面将被释放。

```

1936 static void nopage_sequential_readahead(
1937     struct vm_area_struct * vma,
1938     unsigned long pgoff, unsigned long filesize)
1939     unsigned long ra_window;
1940
1941     ra_window = get_max_readahead(vma->vm_file->f_dentry->d_inode);
1942     ra_window = CLUSTER_OFFSET(ra_window + CLUSTER_PAGES - 1);
1943
1944 /* vm_raend is zero if we havent read ahead
1945 * in this area yet. */
1946 if (vma->vm_raend == 0)
1947     vma->vm_raend = vma->vm_pgoff + ra_window;

```

1941 `get_max_readahead()` 将为在特定 `inode` 中驻留的设备块返回先前读窗口的最大大小。

1942 `CLUSTER_PAGES` 是批量换入换出的页面数。宏 `CLUSTER_OFFSET()` 将预读窗口与集边界对齐。

1945~1946 如果还没有发生预读, 这里设置预读窗口 (`vm_reend`)。

```

1948 /*
1949 * If we've just faulted the page half-way through our window,
1950 * then schedule reads for the next window, and release the
1951 * pages in the previous window.
1952 */
1953 if ((pgoff + (ra_window >> 1)) == vma->vm_raend) {
1954     unsigned long start = vma->vm_pgoff + vma->vm_raend;
1955     unsigned long end = start + ra_window;
1956
1957     if (end > ((vma->vm_end >> PAGE_SHIFT) + vma->vm_pgoff))

```

```

1958 end = (vma->vm_end >> PAGE_SHIFT) + vma->vm_pgoff;
1959 if (start > end)
1960 return;
1961
1962 while ((start < end) && (start < filesize)) {
1963 if (read_cluster_nonblocking(vma->vm_file,
1964 start, filesize) < 0)
1965 break;
1966 start += CLUSTER_PAGES;
1967 }
1968 run_task_queue(&tq_disk);
1969
1970 /* if we're far enough past the beginning of this area,
1971 recycle pages that are in the previous window. */
1972 if (vma->vm_raend >
1973 (vma->vm_pgoff + ra_window + ra_window)) {
1974 unsigned long window = ra_window << PAGE_SHIFT;
1975 end = vma->vm_start + (vma->vm_raend << PAGE_SHIFT);
1976 end -= window + window;
1977 filemap_sync(vma, end - window, window, MS_INVALIDATE);
1978 }
1979
1980 vma->vm_raend += ra_window;
1981 }
1982
1983 return;
1984 }

```

1953 如果在预读窗口的过程中发生异常,这里调度下一个预读窗口从磁盘读入数据。并释放当前窗口的前半部分页面,因为不再需要它们。

1954~1955 计算下一个预读窗口的 start 和 end,因为我们要利用它们开始 I/O。

1957 如果预读窗口的末端在 VMA 的末端之后,这里设置 end 为 VMA 的末端。

1959~1960 如果我们在映射的末端,这里仅返回,因为不再需要进行预读。

1962~1967 调用 read_cluster_nonblocking()(见 D. 6. 5. 2)调度下一个预读窗口读入页面。

1968 调用 run_task_queue()开始 I/O。

1972~1978 利用 filemap_sync()来处理以前预读窗口的页面,因为不再需要它们了。

1980 更新预读窗口的末端地址。

D. 6.5.2 函数: `read_cluster_nonblocking()` (`mm/filemap.c`)

这个函数调度下一个读入页面的预读窗口。

```

737 static int read_cluster_nonblocking(struct file * file,
738                                     unsigned long offset,
739                                     unsigned long filesize)
740 {
741     unsigned long pages = CLUSTER_PAGES;
742     offset = CLUSTER_OFFSET(offset);
743     while ((pages-- > 0) && (offset < filesize)) {
744         int error = page_cache_read(file, offset);
745         if (error < 0)
746             return error;
747         offset++;
748     }
749
750     return 0;
751 }
```

740 `CLUSTER_PAGES` 在低端内存系统中是四个页面, 在高端内存系统中则是八个页面。这意味着, 在有充分内存的 x86 系统中, 在一块中读入 32 KB。

742 `CLUSTER_OFFSET()` 将偏移对齐于块大小边界。

743~748 对块中的每一页, 调用 `page_cache_read()` (见 D. 6.4.2) 将整块读入页面高速缓存。

745~746 如果在预读时发生错误, 则这里返回错误。

750 返回成功。

D. 6.6 交换相关的预读

D. 6.6.1 函数: `swapin_readahead()` (`mm/memory.c`)

这个函数在当前表项陷入一系列的页面。它在 `CLUSTER_PAGES` 个页面换入后或者找到一个未使用的交换表项后停止。

```

1093 void swapin_readahead(swp_entry_t entry)
1094 {
1095     int i, num;
```

```

1096 struct page * new_page;
1097 unsigned long offset;
1098
1099 /*
1100 * Get the number of handles we should do readahead io to.
1101 */
1102 num = valid_swaphandles(entry, &offset);
1103 for (i = 0; i < num; offset++, i++) {
1104 /* Ok, do the async read-ahead now */
1105 new_page =
1106     read_swap_cache_async(SWP_ENTRY(SWP_TYPE(entry)),
1107                           offset));
1108 if (! new_page)
1109     break;
1110 page_cache_release(new_page);
1111 }

```

1102 `valid_swaphandles()` 是决定页面换入数目的函数。它在到达第 1 个空表项或者 `CLUSTER_PAGES` 时停止。

1103~1109 换入页面。

1105 试着利用 `read_swap_cache_async()`(见 K.3.1.1)将页面换入到交换高速缓存。

1106~1107 如果页面不能被换入,这里跳出并返回。

1108 撤销对发生 `read_swap_cache_async()` 页面的引用。

1110 返回。

D.6.6.2 函数: `valid_swaphandles()` (`mm/swapfile.c`)

这个函数决定从交换区起点 `offset` 开始要预读多少页面。它将预读到下一个未使用的交换槽,但是至多返回 `CLUSTER_PAGES`。

```

1238 int valid_swaphandles(swp_entry_t entry, unsigned long * offset)
1239 {
1240     int ret = 0, i = 1 << page_cluster;
1241     unsigned long toff;
1242     struct swap_info_struct * swapdev = SWP_TYPE(entry) + swap_info;
1243
1244     if (! page_cluster) /* no readahead */
1245         return 0;

```

```

1246 toff = (SWP_OFFSET(entry) >> page_cluster) << page_cluster;
1247 if (! toff) /* first page is swap header */
1248 toff ++, i --;
1249 * offset = toff;
1250
1251 swap_device_lock(swapdev);
1252 do {
1253 /* Dont read-ahead past the end of the swap area */
1254 if (toff >= swapdev->max)
1255 break;
1256 /* Dont read in free or bad pages */
1257 if (! swapdev->swap_map[toff])
1258 break;
1259 if (swapdev->swap_map[toff] == SWAP_MAP_BAD)
1260 break;
1261 toff ++ ;
1262 ret ++ ;
1263 } while ( -- i);
1264 swap_device_unlock(swapdev);
1265 return ret;
1266 }

```

1240 *i* 设置为 CLUSTER_PAGE, 与这里的位移操作相对应。

1242 获取包含这个 entry 的 swap_info_struct。

1244~1245 如果禁止了预读, 这里返回。

1246 计算 toff 为 entry 向下取整到最近的 CLUSTER_PAGES 大小边界。

1247~1248 如果 toff 为 0, 则将其移到 1, 因为这是第 1 个包含交换区新信息的页面。

1251 上锁我们将扫描的交换设备。

1252~1263 *i* 初始化为 CLUSTER_PAGES, 最多循环 *i* 次。

1254~1255 如果到达交换区的末端, 这是能够进行预读最远的地方。

1257~1258 如果到达一个没有使用的表项, 这里仅返回, 因为它是我们将预读最远的地方。

1259~1260 相似地, 如果发现了一个坏表项, 这里也返回。

1261 移到下一个槽。

1262 将预读页面数加 1。

1264 解锁交换设备。

1265 返回应该预读的页面数。

附录 E

启动内存分配

目 录

| | |
|---------------------------------------|-----|
| E. 1 初始化引导内存分配器 | 396 |
| E. 1. 1 函数: init_bootmem() | 396 |
| E. 1. 2 函数: init_bootmem_node() | 396 |
| E. 1. 3 函数: init_bootmem_core() | 397 |
| E. 2 分配内存 | 399 |
| E. 2. 1 保留大块区域的内存 | 399 |
| E. 2. 1. 1 函数: reserve_bootmem() | 399 |
| E. 2. 1. 2 函数: reserve_bootmem_node() | 399 |
| E. 2. 1. 3 函数: reserve_bootmem_core() | 399 |
| E. 2. 2 在启动时分配内存 | 401 |
| E. 2. 2. 1 函数: alloc_bootmem() | 401 |
| E. 2. 2. 2 函数: __alloc_bootmem() | 401 |
| E. 2. 2. 3 函数: alloc_bootmem_node() | 402 |
| E. 2. 2. 4 函数: __alloc_bootmem_node() | 402 |
| E. 2. 2. 5 函数: __alloc_bootmem_code() | 403 |
| E. 3 释放内存 | 409 |
| E. 3. 1 函数: free_bootmem() | 409 |
| E. 3. 2 函数: free_bootmem_core() | 409 |
| E. 4 释放引导内存分配器 | 411 |
| E. 4. 1 函数: mem_init() | 411 |
| E. 4. 2 函数: free_pages_init() | 413 |
| E. 4. 3 函数: one_highpage_init() | 414 |
| E. 4. 4 函数: free_all_bootmem() | 415 |
| E. 4. 5 函数: free_all_bootmem_core() | 415 |

E. 1 初始化引导内存分配器

本节中的函数负责引导内存分配器的自举启动。首先是特定于体系结构的函数 `setup_memory()`(见 B. 1.1 小节),但是在调用独立于体系结构的函数 `init_bootmem()`后,所有的体系结构在特定体系结构的函数中都包含相同的基本工作。

E. 1. 1 函数: `init_bootmem()` (`mm/bootmem.c`)

这个函数由 UMA 体系结构调用,用于初始化它们的引导内存分配器结构。

```
304 unsigned long __init init_bootmem (unsigned long start,
                                         unsigned long pages)
305 {
306     max_low_pfn = pages;
307     min_low_pfn = start;
308     return(init_bootmem_core(&contig_page_data, start, 0, pages));
309 }
```

304 这是容易混淆的地方。参数 `pages` 实际上是该节点可寻址内存的 PFN 末端,而不是按名字的意思:页面数。

306 如果没有依赖于体系结构的代码,则设置该节点的可寻址最大 PFN。

307 如果没有依赖于体系结构的代码,则设置该节点的可寻址最小 PFN。

308 调用 `init_bootmem_core()`(见 E. 1.3 小节),在那里完成初始化 `bootmem_data` 的实际工作。

E. 1. 2 函数: `init_bootmem_node()` (`mm/bootmem.c`)

这个函数由 NUMA 体系结构调用,用于初始化特定节点的引导内存分配器数据。

```
284 unsigned long __init init_bootmem_node (pg_data_t * pgdat,
                                         unsigned long freepfn,
                                         unsigned long startpfn,
                                         unsigned long endpfn)
285 {
286     return(init_bootmem_core(pgdat, freepfn, startpfn, endpfn));
287 }
```

286 仅直接调用 `init_bootmem_core()` (见 E. 1.3 小节)。

E. 1.3 函数: `init_bootmem_core()` (`mm/bootmem.c`)

这里初始化对应的 `struct bootmem_data_t`, 并将该节点插入到 `pgdat_list` 节点链表中。

```

46 static unsigned long __init init_bootmem_core (
47     pg_data_t * pgdat,
48     unsigned long mapstart,
49     unsigned long start,
50     unsigned long end)
51 {
52     bootmem_data_t * bdata = pgdat->bdata;
53     unsigned long mapsize = ((end - start) + 7)/8;
54
55     pgdat->node_next = pgdat_list;
56     pgdat_list = pgdat;
57
58     mapsize = (mapsize + (sizeof(long) - 1UL)) &
59             ~(sizeof(long) - 1UL);
60
61     bdata->node_bootmem_map = phys_to_virt(mapstart << PAGE_SHIFT);
62     bdata->node_boot_start = (start << PAGE_SHIFT);
63     bdata->node_low_pfn = end;
64
65     /* Initially all pages are reserved - setup_arch() has to
66      * register free RAM areas explicitly.
67      */
68     memset(bdata->node_bootmem_map, 0xff, mapsize);
69
70     return mapsize;
71 }
```

46 参数如下:

- `pgdat` 是要初始化的节点描述符。
- `mapstart` 是可用内存的起点。
- `start` 是该节点的开始 PFN。
- `end` 是该节点的末尾 PFN。

50 每一页都需要一位来标识, 所以所需映射图的大小是该节点中页面数向上取整到最近

的 8 的倍数后除以 8 所得到的字节数。

52~53 考虑到该节点马上会被初始化,所以将其插入到全局 pgdat_list 中。

55 将图大小向上取整到最近的字边界。

56 将 mapstart 转换为一个虚拟地址,并将其存入 bdata→node_boomem_map。

57 将 PFN 的起点转换为一个物理地址,并将其存入 node_boot_start 中。

58 在 node_low_pfn 中存放处于 ZONE_NORMAL 状态的 PFN 末端。

64 用 1 填充整个图,标记所有的页面都已分配。由那些特定的体系结构代码来标记可用页面。

E. 2 分配内存

E. 2. 1 保留大块区域的内存

E. 2. 1. 1 函数: `reserve_bootmem()` (`mm/bootmem.c`)

```
311 void __init reserve_bootmem (unsigned long addr, unsigned long size)
312 {
313     reserve_bootmem_core(contig_page_data.bdata, addr, size);
314 }
```

313 仅调用 `reserve_bootmem_core`(见 E. 2. 1. 3)。由于这是 NUMA 体系结构,所以从中分配的节点是静态 `contig_page_data` 节点。

E. 2. 1. 2 函数: `reserve_bootmem_node()` (`mm/bootmem.c`)

```
289 void __init reserve_bootmem_node (pg_data_t * pgdat,
290                                     unsigned long physaddr,
291                                     unsigned long size)
292 {
293     reserve_bootmem_core(pgdat->bdata, physaddr, size);
294 }
```

291 仅调用 `reserve_bootmem_core`(见 E. 2. 1. 3),传入参数为所请求节点的启动内存数据。

E. 2. 1. 3 函数: `reserve_bootmem_core()` (`mm/bootmem.c`)

```
74 static void __init reserve_bootmem_core(bootmem_data_t * bdata,
```

```

        unsigned long addr,
        unsigned long size)

75 {
76 unsigned long i;
77 /*
78 * round up, partially reserved pages are considered
79 * fully reserved.
80 */
81 unsigned long sidx = (addr - bdata->node_boot_start)/PAGE_SIZE;
82 unsigned long eidx = (addr + size - bdata->node_boot_start +
83 PAGE_SIZE-1)/PAGE_SIZE;
84 unsigned long end = (addr + size + PAGE_SIZE-1)/PAGE_SIZE;
85
86 if (! size) BUG();
87
88 if (sidx < 0)
89 BUG();
90 if (eidx < 0)
91 BUG();
92 if (sidx >= eidx)
93 BUG();
94 if ((addr >> PAGE_SHIFT) >= bdata->node_low_pfn)
95 BUG();
96 if (end > bdata->node_low_pfn)
97 BUG();
98 for (i = sidx; i < eidx; i++)
99 if (test_and_set_bit(i, bdata->node_bootmem_map))
100 printk("hm, page %08lx reserved twice.\n",
101 i * PAGE_SIZE);
101 }

```

81 sidx 是服务页的起始索引。它的值是从请求地址中减去起始地址并除以页大小得到的。

82 末尾索引 eidx 的计算与 sidx 类似,但它的分配是向上取整到最近的页面。这意味着对保留一页中部分请求将导致整个页都被保留。

84 end 是受本次保留影响的最后 PFN。

86 检查是否给定了一个非零值。

88~89 检查起始索引不在节点起点之前。

90~91 检查末尾索引不在节点末端之后。

92~93 检查起始索引不在末尾索引之后。

94~95 检查起始地址没有超出该启动内存节点所表示的内存范围。

96~97 检查末尾地址没有超出该启动内存节点所表示的内存范围。

88~100 从 `sidx` 开始, 到 `eidx` 结束, 这里测试和设置启动内存分配图中表示页面已经分配的位。如果该位已经设置为 1, 则打印一条消息: 该位被设置了两次。

E. 2.2 在启动时分配内存

E. 2.2.1 函数: `alloc_bootmem()` (`mm/bootmem.c`)

这些宏的调用图如图 5.1 所示。

```
38 #define alloc_bootmem(x) \
39 __alloc_bootmem((x), SMP_CACHE_BYTES, __pa(MAX_DMA_ADDRESS)) \
40 #define alloc_bootmem_low(x) \
41 __alloc_bootmem((x), SMP_CACHE_BYTES, 0) \
42 #define alloc_bootmem_pages(x) \
43 __alloc_bootmem((x), PAGE_SIZE, __pa(MAX_DMA_ADDRESS)) \
44 #define alloc_bootmem_low_pages(x) \
45 __alloc_bootmem((x), PAGE_SIZE, 0)
```

39 `alloc_bootmem()` 将 L1 硬件高速缓存页对齐, 并在 DMA 中可用的最大地址后开始查找一页。

40 `alloc_bootmem_low()` 将 L1 硬件高速缓存页对齐, 并从 0 开始查找。

42 `alloc_bootmem_pages()` 将已分配位对齐到一页大小, 这样满页将从 DMA 可用的最大地址开始分配。

44 `alloc_bootmem_pages()` 将已分配位对齐到一页大小, 这样满页将从物理地址 0 开始分配。

E. 2.2.2 函数: `__alloc_bootmem()` (`mm/bootmem.c`)

```
326 void * __init __alloc_bootmem (unsigned long size,
                                  unsigned long align, unsigned long goal)
327 {
328 pg_data_t * pgdat;
329 void * ptr;
330
331 for_each_pgdat(pgdat)
332 if ((ptr = __alloc_bootmem_core(pgdat->bdata, size,
```

```

333 align, goal)))
334 return(ptr);
335
336 /*
337 * Whoops, we cannot satisfy the allocation request.
338 */
339 printk(KERN_ALERT "bootmem alloc of %lu bytes failed! \n", size);
340 panic("Out of memory");
341 return NULL;
342 }

```

326 参数如下：

- size 是请求分配的大小。
- align 是指定的对齐方式, 它必须是 2 的幂。目前, 一般设置为 SMP_CACHE_BYTES 或 PAGE_SIZE。
- goal 是开始查询的起始地址。

331~334 遍历所有的可用节点, 并试着轮流从各个节点开始分配。在 UMA 中, 就从 contig_page_data 节点开始分配。

339~340 如果分配失败, 系统将不能启动, 故系统瘫痪。

E. 2. 2. 3 函数: alloc_bootmem_node() (mm/bootmem.c)

```

53 #define alloc_bootmem_node(pgdat, x) \
54 __alloc_bootmem_node((pgdat), (x), SMP_CACHE_BYTES,
55     __pa(MAX_DMA_ADDRESS))
55 #define alloc_bootmem_pages_node(pgdat, x) \
56 __alloc_bootmem_node((pgdat), (x), PAGE_SIZE,
57     __pa(MAX_DMA_ADDRESS))
57 #define alloc_bootmem_low_pages_node(pgdat, x) \
58 __alloc_bootmem_node((pgdat), (x), PAGE_SIZE, 0)

```

53~54 alloc_bootmem_node() 将从请求节点处开始分配, 对齐 L1 硬件高速缓存, 并从 ZONE_NORMAL 处开始找到一页 (如, 在 ZONE_DMA 末端, 其为 MAX_DMA_ADDRESS)。

55~56 alloc_bootmem_pages() 将从请求节点处开始分配, 分配页对齐到一页大小, 所以全部页面将从 ZONE_NORMAL 中开始分配。

57~58 alloc_bootmem_low_pages() 将从请求节点处开始分配, 分配页对齐到一页大小, 所以全部页面将从物理地址 0 处开始分配, 这里将使用 ZONE_DMA。

E. 2. 2. 4 函数: __alloc_bootmem_node() (mm/bootmem.c)

```

344 void * __init __alloc_bootmem_node(pg_data_t * pgdat,
345                                     unsigned long size,
346                                     unsigned long align,
347                                     unsigned long goal)
348 {
349     void * ptr;
350
351     ptr = __alloc_bootmem_core(pgdat->bdata, size, align, goal);
352     if (ptr)
353         return (ptr);
354
355     /* Whoops, we cannot satisfy the allocation request.
356     */
357     printk(KERN_ALERT "bootmem alloc of %lu bytes failed! \n", size);
358     panic("Out of memory");
359     return NULL;
360 }
```

344 这里的参数与 __alloc_bootmem_node() (见 E. 2. 2. 4) 处相同, 但是它所要分配的节点是特定的。

348 调用核心函数 __alloc_bootmem_core(见 E. 2. 2. 5) 来完成分配。

349~350 如果成功则返回一个指针。

355~356 否则, 若在这里都没有分配内存, 系统将不能启动, 所以打印一条消息, 表示系统瘫痪。

E. 2. 2. 5 函数: __alloc_bootmem_core() (mm/bootmem.c)

这是从一个带有引导内存分配器的特定节点中分配内存的核心函数。它非常大, 所以将其分解成如下步骤:

- 函数开始处保证所有的参数都正确。
- 以 goal 参数为基础计算开始扫描的起始地址。
- 检查本次分配是否可以使用上次分配的页面以节省内存。
- 在位图中标记已分配页为 1, 并将页中内容清 0。

```

144 static void * __init __alloc_bootmem_core (bootmem_data_t * bdata,
145                                         unsigned long size, unsigned long align, unsigned long goal)
146 {
```

```

147 unsigned long i, start = 0;
148 void * ret;
149 unsigned long offset, remaining_size;
150 unsigned long areasize, preferred, incr;
151 unsigned long eidx = bdata->node_low_pfn -
152 (bdata->node_boot_start >> PAGE_SHIFT);
153
154 if (! size) BUG();
155
156 if (align & (align-1))
157 BUG();
158
159 offset = 0;
160 if (align &&
161 (bdata->node_boot_start & (align - 1UL)) != 0)
162 offset = (align - (bdata->node_boot_start &
163 (align - 1UL)));
163 offset >>= PAGE_SHIFT

```

这是函数的前面部分,它保证参数有效。

144 参数如下:

- bdata 是要分配结构体的启动内存。
- size 是请求分配的大小。
- align 是分配的对齐方式,它必须是 2 的幂。
- goal 是上面要分配的最可能的合适大小。

151 计算末尾位索引 eidx,它返回可能用于分配的最高页面索引。

154 如果指定了 0 则调用 BUG()。

156~157 如果对齐不是 2 的幂,则调用 BUG()。

159 对齐的缺省偏移是 0。

160 如果指定了对齐方式则……

161 请求的对齐方式与开始节点的对齐方式相同,这里计算偏移。

162 要使用的偏移与起始地址低位标记的请求对齐。实际上,这里的 offset 与 align 的一般值 align 相似。

```

169 if (goal && (goal >= bdata->node_boot_start) &&
170 ((goal >> PAGE_SHIFT) < bdata->node_low_pfn)) {
171 preferred = goal - bdata->node_boot_start;
172 } else

```

```

173 preferred = 0;
174
175 preferred = ((preferred + align - 1) & ~ (align - 1))
    >> PAGE_SHIFT;
176 preferred += offset;
177 areasize = (size + PAGE_SIZE - 1) / PAGE_SIZE;
178 incr = align >> PAGE_SHIFT ? : 1;

```

这一块计算开始的 PFN 从 goal 参数为基础的地址处开始扫描。

169 如果指定了 goal, 且 goal 在该节点的开始地址之后, goal 小于该节点可寻址 PFN, 则……

170 开始的适当偏移是 goal 减去该节点可寻址的内存起点。

173 如果不是这样, 则适当偏移为 0。

175~176 考虑偏移, 调整适当偏移的大小, 这样该地址将可以正确对齐。

177 本次分配影响到的页面数存放在 areasize 中。

178 incr 是要跳过的页面数, 如果多于一页, 则它们满足对齐请求。

```

179
180 restart_scan:
181 for (i = preferred; i < eidx; i += incr) {
182     unsigned long j;
183     if (test_bit(i, bdata->node_bootmem_map))
184         continue;
185     for (j = i + 1; j < i + areasize; ++j) {
186         if (j >= eidx)
187             goto fail_block;
188         if (test_bit(j, bdata->node_bootmem_map))
189             goto fail_block;
190     }
191     start = i;
192     goto found;
193 fail_block:;
194 }
195 if (preferred) {
196     preferred = offset;
197     goto restart_scan;
198 }
199 return NULL;

```

这一块扫描内存, 寻找一块足够大的块来满足请求。

180 如果本次分配不能满足从 goal 开始,则跳到这里的标志,这样将重新扫描该映射图。

181~194 从 preferred 开始,这里线性扫描一块足够大的块来满足请求。这里以 incr 为步长来遍历整个地址空间以满足大于一页的对齐。如果对齐小于一页,incr 为 1。

185~190 扫描下一个 areasize 大小的页面来确定它是否也可以被释放。如果到达可寻址空间的末端(eidx)或者其中的一页已经在使用,则失败。

191~192 找到一个空闲块,所以这里记录 start,并跳转到找到的块。

195~198 分配失败,所以从开始处重新开始。

199 如果再次失败,则返回 NULL,这将导致系统瘫痪。

```

200 found:
201 if (start >= eidx)
202 BUG();
203
209 if (align <= PAGE_SIZE
210 && bdata->last_offset && bdata->last_pos + 1 == start) {
211 offset = (bdata->last_offset + align-1) & ~(align-1);
212 if (offset > PAGE_SIZE)
213 BUG();
214 remaining_size = PAGE_SIZE-offset;
215 if (size < remaining_size) {
216 areasize = 0;
217 // last_pos unchanged
218 bdata->last_offset = offset + size;
219 ret = phys_to_virt(bdata->last_pos * PAGE_SIZE + offset +
220 bdata->node_boot_start);
221 } else {
222 remaining_size = size - remaining_size;
223 areasize = (remaining_size+PAGE_SIZE-1)/PAGE_SIZE;
224 ret = phys_to_virt(bdata->last_pos * PAGE_SIZE +
225 offset +
bdata->node_boot_start);
226 bdata->last_pos = start + areasize-1;
227 bdata->last_offset = remaining_size;
228 }
229 bdata->last_offset &= ~PAGE_MASK;
230 } else {
231 bdata->last_pos = start + areasize - 1;
232 bdata->last_offset = size & ~PAGE_MASK;
233 ret = phys_to_virt(start * PAGE_SIZE +

```

```

bdata->node_boot_start);
234 }

```

这一块测试以确定本次分配可以与前一次分配合并。

201~202 检查分配的起点不会在可寻址内存之后。刚才已经检查过,所以这里是多余的。

209~230 如果对齐小于 PAGE_SIZE,前面的页面在其中有空间(last_offset != 0),而且前面使用的页与本次分配的页相邻,则试着与前一次分配合并。

231~234 如果没有,这里记录本次分配用到的页面和偏移,以便于下次分配时的合并。

211 更新用于对 align 请求正确分页的偏移。

212~213 如果偏移现在超过了页面边界,则调用 BUG()。这个条件需要使用一次非常糟糕的对齐选择。由于一般使用的对齐仅是一个 PAGE_SIZE 的因子,所以不可能在平常使用。

214 remaining_size 是以前用过的页面中处于空闲的空间。

215~221 如果在旧的页面中有足够的空间剩余,这里使用旧的页面,然后更新 bootmem_data 结构来反映这一点。

221~228 如果不是这样,则这里计算除了这一页外还需要多少页面,并更新 bootmem_data。

216 这次分配中用到的页面数现在为 0。

218 更新 last_offset 为本次分配的末端。

219 计算返回成功分配的虚拟地址。

222 remaining_size 是上一次用来满足分配的页面空间。

223 计算还需要多少页面来满足分配请求。

224 计算分配开始的地址。

226 使用到的最后一页是 start 页面加上满足这次分配的额外页面数量 areasize。

227 已经计算过本次分配的末端。

229 如果偏移在页尾,则标记为 0。

231 不发生合并,所以这里记录用到的满足本次分配的最后一页。

232 记录用到的最后一页。

233 记录分配的起始虚拟地址。

```

238 for (i = start; i < start + areasize; i++)
239 if (test_and_set_bit(i, bdata->node_bootmem_map))
240 BUG();
241 memset(ret, 0, size);
242 return ret;

```

243 }

这一块在位图中标记分配页为 1，并将其内容清 0。

238~240 遍历本次分配用到的所有页面，在位图中设置 1。如果它们中已经有 1，则发生了一次重复分配，所以调用 BUG()。

241 将页面用 0 填充。

242 返回分配的地址。

E.3 释放内存

E.3.1 函数：free_bootmem() (mm/bootmem.c)

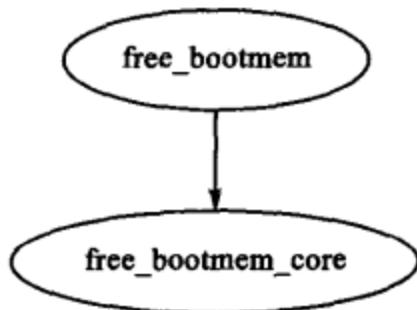


图 E.1 调用图：free_bootmem()

```

294 void __init free_bootmem_node (pg_data_t * pgdat,
                                  unsigned long physaddr, unsigned long size)
295 {
296     return(free_bootmem_core(pgdat->bdata, physaddr, size));
297 }
316 void __init free_bootmem (unsigned long addr, unsigned long size)
317 {
318     return(free_bootmem_core(contig_page_data.bdata, addr, size));
319 }
  
```

296 调用核心函数，以被请求节点的相应启动内存数据为参数。

318 调用核心函数，以 contig_page_data 的启动内存数据为参数。

E.3.2 函数: free_bootmem_core() (mm/bootmem.c)

```

103 static void __init free_bootmem_core(bootmem_data_t * bdata,
104                                     unsigned long addr,
105                                     unsigned long size)
106 {
107     unsigned long i;
108     unsigned long start;
109     unsigned long sidx;
110     unsigned long eidx = (addr + size -
111                           bdata->node_boot_start)/PAGE_SIZE;
112     unsigned long end = (addr + size)/PAGE_SIZE;
113
114     if (!size) BUG();
115     if (end > bdata->node_low_pfn)
116         BUG();
117
118     /*
119      * Round up the beginning of the address.
120      */
121     start = (addr + PAGE_SIZE-1) / PAGE_SIZE;
122     sidx = start - (bdata->node_boot_start/PAGE_SIZE);
123
124     for (i = sidx; i < eidx; i++) {
125         if (!test_and_clear_bit(i, bdata->node_bootmem_map))
126             BUG();
127     }
128 }
129 }
```

112 计算受影响的末端索引 eidx。

113 如果末端地址不是页对齐,则它为受影响区域的末端向下取整到最近页面。

115 如果释放了大小为 0 的页面,则调用 BUG()。

116~117 如果末端 PFN 在该节点可寻址内存之后,则这里调用 BUG()。

122 如果起始地址不是页对齐的,则将其向上取整到最近页面。

123 计算要释放的起始索引。

125~127 释放全部满页面,这里清理在启动位图中的位。如果已经为 0,则表示是一次重复释放或内存从未使用,这里调用 BUG()。

E. 4 释放引导内存分配器

在系统启动后,引导内存分配器就不再需要了,这些函数负责销毁不需要的引导内存分配器结构,并将其余的页面传入到普通的物理页面分配器中。

E. 4. 1 函数: mem_init() (arch/i386/mm/init.c)

这个函数的调用图如图 5.2 所示。引导内存分配器的这个函数的重要部分是它调用 free_pages_init()(见 E. 4. 2 小节)。这个函数分成如下几部分:

- 函数前面部分为高端内存地址设置在全局 mem_map 中的 PFN,并将系统范围的 0 页面清零。
- 调用 free_pages_init()(见 E. 4. 2 小节)。
- 打印系统中可用内存的提示信息。
- 如果配置项可用,则检查 CPU 是否支持 PAE,并测试 CPU 中的 WP 位。这很重要,如果没有 WP 位,就必须调用 verify_write()对内核到用户空间的每一次写检查。这仅应用于像 386 一样的老处理器。
- 填充 swapper_pg_dir 的 PGD 用户空间部分的表项,其中有内核页表。0 页面映射到所有的表项。

```

507 void __init mem_init(void)
508 {
509     int codesize, reservedpages, datasize, initsize;
510
511     if (!mem_map)
512         BUG();
513
514     set_max_mapnr_init();
515
516     high_memory = (void *) __va(max_low_pfn * PAGE_SIZE);
517
518     /* clear the zero-page */
519     memset(empty_zero_page, 0, PAGE_SIZE);

```

514 这个函数记录从 mem_map(highmem_start_page)处高端内存开始的 PFN,系统中最大的页面数(max_mapnr 和 num_physpages),以及最后是可能被内核映射的最大页面数

(num_mappedpages)。

516 high_memory 是高端内存开始处的虚拟地址。

519 将系统范围内的 0 页面清 0。

520

521 reservedpages = free_pages_init();

522

521 调用 free_pages_init() (见 E. 4. 2 小节), 在那里告知引导内存分配器释放它自身以及初始化高端内存的所有页面, 以用于伙伴分配器。

```
523 codesize = (unsigned long) &_etext - (unsigned long) &_text;
524 datasize = (unsigned long) &_edata - (unsigned long) &_etext;
525 initsize = (unsigned long) &__init_end - (unsigned long)
               &__init_begin;
526
527 printk (KERN_INFO "Memory: %luk / %luk available (%dk kernel code,
               %dk reserved, %dk data, %dk init, %ldk highmem)\n",
528 (unsigned long) nr_free_pages() << (PAGE_SHIFT-10),
529 max_mapnr << (PAGE_SHIFT-10),
530 codesize >> 10,
531 reservedpages << (PAGE_SHIFT-10),
532 datasize >> 10,
533 initsize >> 10,
534 (unsigned long) (totalhigh_pages << (PAGE_SHIFT-10))
535 );
```

这一块打印一条提示信息。

523 计算用于初始化代码和数据的代码段、数据段和内存大小。(所有标识为 __init 的函数都将在这一部分)。

527~535 打印一条整洁的信息, 告知可用内存和由内核消耗的内存量。

536

```
537 # if CONFIG_X86_PAE
538 if (!cpu_has_pae)
539 panic("cannot execute a PAE-enabled kernel on a PAE-less
       CPU!");
540 #endif
541 if (boot_cpu_data.wp_works_ok < 0)
542 test_wp_bit();
543
```

538~539 如果 PAE 可用,但是处理器不支持,则这里瘫痪。

541~542 测试 WP 位可用。

```
550 #ifndef CONFIG_SMP
551 zap_low_mappings();
552#endif
553
554 }
```

551 遍历 swapper_pg_dir 的用户空间部分用到的每个 PGD,并将 0 页面与之映射。

E. 4.2 函数: free_pages_init() (arch/i386/mm/init.c)

这个函数有 3 个重要的功能:调用 free_all_bootmem() (见 E. 4.4 小节),销毁引导内存分配器,以及释放伙伴分配器的所有高端内存。

```
481 static int __init free_pages_init(void)
482 {
483     extern int ppro_with_ram_bug(void);
484     int bad_ppro, reservedpages, pfn;
485
486     bad_ppro = ppro_with_ram_bug();
487
488     /* this will put all low memory onto the freelists */
489     totalram_pages += free_all_bootmem();
490
491     reservedpages = 0;
492     for (pfn = 0; pfn < max_low_pfn; pfn++) {
493         /*
494          * Only count reserved RAM pages
495         */
496         if (page_is_ram(pfn) && PageReserved(mem_map + pfn))
497             reservedpages++;
498     }
499 #ifdef CONFIG_HIGHMEM
500     for (pfn = highend_pfn-1; pfn >= highstart_pfn; pfn--)
501         one_highpage_init((struct page *) (mem_map + pfn), pfn,
502                           bad_ppro);
502     totalram_pages += totalhigh_pages;
503 #endif
```

```
504 return reservedpages;
505 }
```

486 在奔腾 Pro 版本中有一个 bug, 阻止高端内存中的某些页被使用。函数 ppro_with_ram_bug() 检查是否存在这个 bug。

489 调用 free_all_bootmem() 来销毁引导内存分配器。

491~498 遍历所有的内存, 计数保留给引导内存分配器的页面数。

500~501 对高端内存中的每一页, 这里调用 one_highpage_init() (见 E. 4.3 小节)。这个函数清除 PG_reserved 位, 设置 PG_high 位, 设置计数为 1, 调用 __free_page() 来给伙伴分配器分配页面, 增加 totalhigh_pages 计数。杀死有 bug 的奔腾 Pro 的页面将被跳过。

E. 4.3 函数: one_highpage_init() (arch/i386/mm/init.c)

这个函数初始化高端内存中的页面信息, 并检查以保证页面不会在某些奔腾 Pro 上报 bug。它仅在编译时指定了 CONFIG_HIGHMEM 的情况下存在。

```
449 #ifdef CONFIG_HIGHMEM
450 void __init one_highpage_init(struct page * page, int pfn,
451                           int bad_ppro)
452 {
453     if (!page_is_ram(pfn)) {
454         SetPageReserved(page);
455         return;
456     }
457     if (bad_ppro && page_kills_ppro(pfn)) {
458         SetPageReserved(page);
459         return;
460     }
461     ClearPageReserved(page);
462     set_bit(PG_highmem, &page->flags);
463     atomic_set(&page->count, 1);
464     __free_page(page);
465     totalhigh_pages++;
466 }
467 #endif /* CONFIG_HIGHMEM */
```

452~455 如果在 PFN 处不存在页面, 则这里标记 struct page 为保留的, 所以不会使用该

页面。

457~460 如果当前运行的 CPU 是有奔腾 Pro bug 的,而这个页面会导致崩溃(page_kill_ppro()进行这项检查),这里就标记页面为保留的,它也不会被分配。

462 从这里开始,就会使用高端内存的页面,所以这里首先清除保留位,然后将它们分配给伙伴分配器。

463 设置 PG_highmem 位表示它是一个高端内存页面。

464 初始化页面的使用计数为 1,它由伙伴分配器设为 0。

465 利用__free_page() (见 F. 4.2 小节)来释放页面,这样伙伴分配器会将高端内存页面加到它的空闲链表中。

466 将可用的高内存页面总数(totalhigh_pages)加 1。

E. 4.4 函数: free_all_bootmem() (mm/bootmem.c)

```
299 unsigned long __init free_all_bootmem_node (pg_data_t * pgdat)
300 {
301     return(free_all_bootmem_core(pgdat));
302 }
321 unsigned long __init free_all_bootmem (void)
322 {
323     return(free_all_bootmem_core(&contig_page_data));
324 }
```

299~302 对 NUMA 而言,这里仅是简单地调用以特定 pgdat 为参数的核心函数。

321~324 对 UMA 而言,这里调用仅以节点 contig_page_data 为参数的核心函数。

E. 4.5 函数: free_all_bootmem_core() (mm/bootmem.c)

这是销毁引导内存分配器的核心函数。它分为如下两个主要部分:

- 对该节点已知的未分配页面,它完成如下步骤:
 - 清除结构页面中的 PG_reserved 标志。
 - 置计数为 1。
 - 调用__free_pages(),这样伙伴分配器可以构建它的空闲链表。
- 释放用于位图的页面,并将其释放给伙伴分配器。

```
245 static unsigned long __init free_all_bootmem_core(pg_data_t * pgdat)
246 {
```

```

247 struct page * page = pgdat->node_mem_map;
248 bootmem_data_t * bdata = pgdat->bdata;
249 unsigned long i, count, total = 0;
250 unsigned long idx;
251
252 if (! bdata->node_bootmem_map) BUG();
253
254 count = 0;
255 idx = bdata->node_low_pfn -
    (bdata->node_boot_start >> PAGE_SHIFT);
256 for (i = 0; i < idx; i++, page++) {
257 if (! test_bit(i, bdata->node_bootmem_map)) {
258 count++;
259 ClearPageReserved(page);
260 set_page_count(page, 1);
261 __free_page(page);
262 }
263 }
264 total += count;

```

252 如果没有映射图,则意味着这个节点已经被释放了,且肯定在依赖于体系结构的代码中出现了错误,所以这里调用 BUG()。

254 将页面数的运行数传给伙伴分配器。

255 idx 是该节点最后可寻址的索引。

256~263 遍历该节点可寻址的所有页面。

257 如果该页标记为空闲,则……

258 将传给伙伴分配器页面数的运行数加 1。

259 清除 PG_reserved 标志。

260 设置计数为 1,这样伙伴分配器将考虑这是页面的最后一个用户,并将其放入到空闲链表中。

261 调用伙伴分配器的释放函数,这样页面将被加入到空闲链表中。

264 total 将设为由此函数传递的页面总数。

```

270 page = virt_to_page(bdata->node_bootmem_map);
271 count = 0;
272 for (i = 0;
    i < ((bdata->node_low_pfn - (bdata->node_boot_start >> PAGE_SHIFT)
        )/8 + PAGE_SIZE-1)/PAGE_SIZE;
    i++, page++) {

```

```
273 count ++ ;  
274 ClearPageReserved(page);  
275 set_page_count(page, 1);  
276 __free_page(page);  
277 }  
278 total += count;  
279 bdata->node_bootmem_map = NULL;  
280  
281 return total;  
282 }
```

这一块释放分配器位图并返回。

270 获得在启动内存映射图顶端的 struct page。

271 位图释放的页面数。

272~277 对该位图使用的所有页面,这里与前面的代码一样,将其释放给伙伴分配器。

279 设启动内存映射图为 NULL,以阻止其意外地被第 2 次释放。

281 返回该函数释放的页面总数,或者说,返回加入到伙伴分配器空闲链表的页面总数。

附录 F

物理页面分配

目 录

| | |
|--------------------------------|-----|
| F. 1 分配页面 | 420 |
| F. 1. 1 函数:alloc_pages() | 420 |
| F. 1. 2 函数:_alloc_pages() | 420 |
| F. 1. 3 函数:__alloc_pages() | 421 |
| F. 1. 4 函数:rmqueue() | 425 |
| F. 1. 5 函数:expand() | 427 |
| F. 1. 6 函数:balance_classzone() | 428 |
| F. 2 分配辅助函数 | 433 |
| F. 2. 1 函数:alloc_page() | 433 |
| F. 2. 2 函数:__get_free_page() | 433 |
| F. 2. 3 函数:_get_free_pages() | 433 |
| F. 2. 4 函数:__get_dma_pages() | 434 |
| F. 2. 5 函数:get_zeroed_page() | 434 |
| F. 3 释放页面 | 435 |
| F. 3. 1 函数:__free_pages() | 435 |
| F. 3. 2 函数:_free_pages_ok() | 435 |
| F. 4 释放辅助函数 | 440 |
| F. 4. 1 函数 free_pages() | 440 |
| F. 4. 2 函数:__free_page() | 440 |
| F. 4. 3 函数:free_page() | 440 |

F.1 分配页面

F.1.1 函数: alloc_pages() (include/linux/mm.h)

本函数的调用图如图 6.2 所示。它的声明如下：

```
439 static inline struct page * alloc_pages(unsigned int gfp_mask,
                                         unsigned int order)
440 {
444 if (order >= MAX_ORDER)
445 return NULL;
446 return _alloc_pages(gfp_mask, order);
447 }
```

439 gfp_mask(Get Free Pages) 标志表明分配器如何操作。如,如果没有设置 GFP_WAIT,分配器将不会阻塞,而是在内存紧张时返回 NULL,次是分配页面数的 2 的幂。

444~445 在编译时的调试检查。

446 函数的描述如下。

F.1.2 函数: _alloc_pages() (mm/page_alloc.c)

这个函数_alloc_pages()有两个变种。第 1 个是设计为仅 UMA 体系结构如 x86 可用,它在 mm/page_alloc.c 中。仅指静态节点 contig_page_data。第 2 个是在 mm/numa.c 中,它是一个简单的扩展。它使用一个局部分配节点的策略,这意味着仅从靠近处理器的空闲区分配内存。对本书而言,我们仅考察 mm/page_alloc.c 中的那个。但是 NUMA 体系结构上的开发者应该阅读 mm/numa.c 中的_alloc_pages()和_alloc_pages_pgdat()。

```
244 #ifndef CONFIG_DISCONTIGMEM
245 struct page * _alloc_pages(unsigned int gfp_mask,
                           unsigned int order)
246 {
247 return __alloc_pages(gfp_mask, order,
248 contig_page_data.node_zonelists + (gfp_mask & GFP_ZONEMASK));
249 }
250#endif
```

244 UMA 体系结构中的 ifndef 象 x86 中一样。NUMA 体系结构使用 mm/numa.c 中的

`_alloc_pages()` 函数, 它在分配时采取局部部分配节点的策略。

245 `gfp_mask` 标志位告知分配器如何操作。`order`(次)是待分配页面数的 2 的幂。

247 `node_zonelists` 是由分配回退管理区组成的一个数组。它在 `build_zonelists()`(见 B.1.6 小节)中初始化。`gfp_mask` 的低 16 位表明哪个管理区适合分配。应用位掩码 `gfp_mask & GFP_ZONEMASK` 将给出 `node_zonelists` 中我们要分配的索引。

F.1.3 函数: `__alloc_pages()` (`mm/page_alloc.c`)

在这个阶段, 我们到达了描述为“伙伴分配器的中心地带”, 即 `__alloc_pages()` 函数。它负责遍历回退管理区, 然后选择一个合适的进行分配。如果内存紧张, 则采用一些步骤来解决这个问题。它将唤醒 `kswapd`, 并在需要的时候, 自动完成 `kswapd` 的工作。

```

327 struct page * __alloc_pages(unsigned int gfp_mask,
328                           unsigned int order,
329                           zonelist_t * zonelist)
330 {
331     unsigned long min;
332     zone_t ** zone, * classzone;
333     struct page * page;
334     int freed;
335
336     zone = zonelist->zones;
337     classzone = * zone;
338     if (classzone == NULL)
339         return NULL;
340     min = 1UL << order;
341     for (;;) {
342         zone_t * z = * (zone++);
343         if (! z)
344             break;
345         min += z->pages_low;
346         if (z->free_pages > min) {
347             page = rmqueue(z, order);
348             if (page)
349                 return page;
350     }

```

334 设置该管理区为要从中分配的合适管理区。

335 合适的管理区标记为 classzone。如果在后面达到一个页面的极值，则 classzone 标记为需要平衡。

336~337 不需要的有效性检查。build_zonelists()需要被严重地打散。

338~350 这一块的风格似乎在这个函数中出现多次。它读作“遍历在该回退链表中的所有管理区，看是否有一个分配器可以在不超过极值的情况下满足条件”。每个回退管理区的 pages_low 加到一起。这里特意减少使用一个回退管理区的概率。

340 z 是当前检查的管理区。zone 变量移到下一个回退管理区。

341~342 如果这是回退链表的最后一个管理区，则退出循环。

344 为方便比较，将该极值分配器的页面数加 1，这一步在每个回退管理区的每个区域中都进行。虽然这看起来像是一个 bug，但它实际上是想减少使用回退管理区的概率。

345~349 如果页面块分配可以不超过 pages_min 极值，就分配一块页面块。rmqueue()（见 F.1.4 小节）负责从管理区中重新移动该页面块。

347~348 如果可以分配该页，则这里返回一个指向它们的指针。

```

352 classzone->need_balance = 1;
353 mb();
354 if (waitqueue_active(&kswapd_wait))
355 wake_up_interruptible(&kswapd_wait);
356
357 zone = zonelist->zones;
358 min = 1UL << order;
359 for (;;) {
360     unsigned long local_min;
361     zone_t *z = * (zone++);
362     if (!z)
363         break;
364
365     local_min = z->pages_min;
366     if (!(gfp_mask & __GFP_WAIT))
367         local_min >>= 2;
368     min += local_min;
369     if (z->free_pages > min) {
370         page = rmqueue(z, order);
371         if (page)
372             return page;
373     }
374 }
```

375

352 标记合适的管理区为需要平衡。这个标志位将在后面由 kswapd 读取。

353 这是一个内存界限。它保证所有的 CPU 都可以看到在这行代码之前的所有变化。这很重要,因为 kswapd 除了在内存分配器上运行外还可以在不同的处理器上运行。

354~355 如果 kswapd 处于睡眠状态,则唤醒它。

357~358 以第一个合适的管理区和 min 值再次开始。

360~374 遍历所有的管理区。这一次,如果不超过 page_min 的极限值则分配一页。

365 local_min 表明该管理区可以拥有的空闲页数量。

366~367 如果该进程不能等待或者重调度(_GFP_WAIT 清空),在这里允许管理区进一步增加比普通极值还要大的内存压力。

376 /* here we're in the low on memory slow path */

377

378 rebalance:

379 if (current->flags & (PF_MEMALLOC | PF_MEMDIE)) {

380 zone = zonelist->zones;

381 for (;;) {

382 zone_t * z = * (zone++);

383 if (! z)

384 break;

385

386 page = rmqueue(z, order);

387 if (page)

388 return page;

389 }

390 return NULL;

391 }

378 在试着同步空闲页面后返回这个标号。从这一行起,就达到了内存路径的低端,在这种情形下,进程很可能会长时间睡眠。

379~391 OOM 管理程序只会设置两个标志。由于进程试着完整地杀死它自身,在可能的情况下,将分配页面,因为知道马上就可以释放这些页面。

393 /* Atomic allocations - we can't balance anything */

394 if (! (gfp_mask & __GFP_WAIT))

395 return NULL;

396

397 page = balance_classzone(classzone, gfp_mask, order, &freed);

398 if (page)

```

399 return page;
400
401 zone = zonelist->zones;
402 min = 1UL << order;
403 for (;;) {
404     zone_t *z = * (zone++);
405     if (!z)
406         break;
407
408     min += z->pages_min;
409     if (z->free_pages > min) {
410         page = rmqueue(z, order);
411         if (page)
412             return page;
413     }
414 }
415
416 /* Dont let big-order allocations loop */
417 if (order > 3)
418     return NULL;
419
420 /* Yield for kswapd, and try again */
421 yield();
422 goto rebalance;
423 }

```

394~395 如果调用进程无法睡眠,这里返回 NULL,因为分配页面的惟一途径是调用睡眠。

397 balance_classzone()(见 F.1.6 小节)同步进行 kswapd 的工作。主要的区别在于,它不是将内存释放到一个全局的池中,而是保证进程始终使用 current->local_pages 链表。

398~399 如果已经按顺序释放了一个页块,这里就返回该页块。由于高次页面可能被释放,所以这里 NULL 并不意味着分配失败。

403~414 这里与前面块相似。如果可以不超过 pages_min 极值则可以分配一页块。

417~418 要满足一次分配一块如 2^4 那样数量的页面比较困难。如果现在还没有满足,那么最好直接返回 NULL。

421 阻塞进程,以给 kswapd 工作的机会。

433 试着再次平衡和分配管理区。

F. 1.4 函数: rmqueue() (mm/page_alloc.c)

这个函数从__alloc_pages()处调用。它负责找到一块足够大的用于分配的内存块。如果没有满足请求的内存块,它就试着寻找更高次的可能被分割开的两个伙伴块。实际的分割由函数expand()(见 F. 1.5 小节)完成。

```

198 static FASTCALL(struct page * rmqueue(zone_t * zone,
                                         unsigned int order));
199 static struct page * rmqueue(zone_t * zone, unsigned int order)
200 {
201     free_area_t * area = zone->free_area + order;
202     unsigned int curr_order = order;
203     struct list_head * head, * curr;
204     unsigned long flags;
205     struct page * page;
206
207     spin_lock_irqsave(&zone->lock, flags);
208     do {
209         head = &area->free_list;
210         curr = head->next;
211
212         if (curr != head) {
213             unsigned int index;
214
215             page = list_entry(curr, struct page, list);
216             if (BAD_RANGE(zone, page))
217                 BUG();
218             list_del(curr);
219             index = page - zone->zone_mem_map;
220             if (curr_order != MAX_ORDER-1)
221                 MARK_USED(index, curr_order, area);
222             zone->free_pages -= 1UL << order;
223
224             page = expand(zone, page, index, order,
225                           curr_order, area);
225             spin_unlock_irqrestore(&zone->lock, flags);
226
227             set_page_count(page, 1);

```

```

228 if (BAD_RANGE(zone, page))
229 BUG();
230 if (PageLRU(page))
231 BUG();
232 if (PageActive(page))
233 BUG();
234 return page;
235 }
236 curr_order++;
237 area++;
238 } while (curr_order < MAX_ORDER);
239 spin_unlock_irqrestore(&zone->lock, flags);
240
241 return NULL;
242 }

```

199 参数是要从中分配的区域和页面所需的次。

201 由于 free_area 是顺序链表的一个数组, 次可以用作数组中的下标。

207 获取一个管理区锁。

208~238 这个 while 块负责找到我们需要分配的页面次。如果空闲块处于我们感兴趣的次中, 这里就检查更高的块直到找到更加适合的块。

209 head 是该次空闲页块的链表。

210 curr 是页面的第一块。

212~235 如果空闲页块处于这一次, 这里就开始分配。

215 该页设置为指向空闲块的第一个页面的指针。

216~217 有效性检查保证该页属于这个管理区以及处于 zone_mem_map 中。现在还不清楚是否可能发生在分配器自身将块放到错误的管理区中而不是严重 bug 的情况。

218 由于该块将被分配, 这里就从空闲链表中移除。

219 index 将 zone_mem_map 看做一个由页面组成的数组, 下标就是该数组中的偏移。

220~221 表示伙伴对的位置位。MARX_USED() 是一个计算置位位的宏。

222 更新该管理区统计。1UL << order 是要分配的页面数。

224 expand() (见 F. 1.5 小节) 是负责分割高次页面块的函数。

225 不再需要进一步更新管理区, 所以这里释放锁。

227 表明页面正在使用中。

228~233 进行有效性检查。

234 由于已经成功分配页面块, 所以返回它。

236~237 如果没有释放正确次的页面块, 这里就将转移到高次页面块, 看在那里可以找

到什么。

239 不再需要进一步更新管理区,所以这里释放锁。

241 没有所请求页面块或者更高次页面块可用,所以这里返回失败。

F. 1.5 函数: expand() (mm/page_allo.c)

这个函数将页面块分割成更高次，直到所需次的页面块可用。

```
177 static inline struct page * expand (zone_t * zone,
178                                     struct page * page,
179                                     unsigned long index,
180                                     int low,
181                                     int high,
182                                     free_area_t * area)
183 {
184     unsigned long size = 1 << high;
185
186     while (high > low) {
187         if (BAD_RANGE(zone, page))
188             BUG();
189         area--;
190         high--;
191         size >>= 1;
192         list_add(&(page)->list, &(area)->free_list);
193         MARK_USED(index, high, area);
194         index += size;
195         page += size;
196     }
197
198     if (BAD_RANGE(zone, page))
199         BUG();
200
201     return page;
202 }
```

177 参数如下：

- zone 是从中分配的地方。
 - page 是要待分割块的一个页面。
 - index 是在 mem_map 中的页面索引。
 - low 是需要分配的页面次。
 - high 是分配时要分割的页面次。

- area 是代表高次页面块的 free_area_t。

180 size 是待分割的页面数量。

182~192 不断分割, 直到找到了所需页面次的一块。

183~184 有效性检查保证该页面属于该管理区, 且在 zone_mem_map 中。

185 area 是现在表示低次页面块的下一个 free_area_t。

186 high 是待分割页面块的下一次。

187 分割块的大小是原来大小的一半。

188 在伙伴对中, mem_map 中较低的那个加入到低次的空闲链表中。

189 表示伙伴对的位置位。

190 index 是现在新创建伙伴对的第 2 个伙伴索引。

191 page 现在指向新创建伙伴对的第 2 个伙伴。

193~194 有效性检查。

195 已经成功分割一块。所以返回页面。

F. 1.6 函数: balance_classzone() (mm/page_alloc.c)

这个函数是直接回收路径的一部分。可以睡眠的分配器将调用这个函数同步完成 kswapd 的工作。由于这个进程现在亲自完成工作, 所以它释放的特定次的页面保留在一个 current→local_pages 链表中, 链表中的页面块数量保存在 current→nr_local_pages 中。注意页面块与页面数量不同, 页面块可以是任意次。

```

253 static struct page * balance_classzone(zone_t * classzone,
                                         unsigned int gfp_mask,
                                         unsigned int order,
                                         int * freed)
254 {
255     struct page * page = NULL;
256     int __freed = 0;
257
258     if (!(gfp_mask & __GFP_WAIT))
259         goto out;
260     if (in_interrupt())
261         BUG();
262
263     current->allocation_order = order;
264     current->flags |= PF_MEMALLOC | PF_FREE_PAGES;
265

```

```

266 __freed = try_to_free_pages_zone(classzone, gfp_mask);
267
268 current->flags &= ~(PF_MEMALLOC | PF_FREE_PAGES);
269

```

258~259 如果调用者不允许睡眠,这里转到 out 退出该函数。如果发生睡眠,这个函数将必须直接调用,或者 __alloc_pages() 需要被故意中断。

260~261 这个函数可能不会被突然使用。另外,发生这种条件必须引入故意的损坏。

263 记录在 current->allocation_order 的分配大小。虽然它可能用于将特定次的页面加入到 local_pages 链表中,但实际上这没有用。链表中页面的次存放在 page->index 中。

264 设置释放函数的标志以将页面加入 local_list。

266 利用 try_to_free_pages_zone() (见 J. 5.3 小节)直接从特定管理区中释放页面。这也是 kswapd 与直接回收路径交互的地方。

268 再一次清除标志位,这样释放函数不会继续将页面加入到 local_pages 链表中。

```

270 if (current->nr_local_pages) {
271 struct list_head * entry, * local_pages;
272 struct page * tmp;
273 int nr_pages;
274
275 local_pages = &current->local_pages;
276
277 if (likely(__freed)) {
278 /* pick from the last inserted so were lifo */
279 entry = local_pages->next;
280 do {
281 tmp = list_entry(entry, struct page, list);
282 if (tmp->index == order &&
283     memclass(page_zone(tmp), classzone)) {
284 list_del(entry);
285 current->nr_local_pages--;
286 set_page_count(tmp, 1);
287 page = tmp;
288 if (page->buffers)
289 BUG();
290 if (page->mapping)
291 BUG();
292 if (!VALID_PAGE(page))

```



```

293 BUG();
294 if (PageLocked(page))
295 BUG();
296 if (PageLRU(page))
297 BUG();
298 if (PageActive(page))
299 BUG();
300 if (PageDirty(page))
301 BUG();
302
303 break;
304 }
305 } while ((entry = entry->next) != local_pages);
306 }

```

假设页面在 local_pages 链表中,这个函数将遍历链表查找属于特定管理区和次的页面块。

270 如果页面存放在局部链表中则仅进入这一块。

275 从链表头开始。

277 如果利用 try_to_free_pages_zone() 释放了页面,则……

279 插入的最后一页选择为第 1 页,因为这可能是一次高速缓存命中,而且一般都使用最近引用过的页面。

280~305 遍历链表中所有页面,直到我们找到合适的管理区和次。

281 从链表项中获取页面。

282 页面块的次存放在 page->index 中,所以这里检查该次是否与请求的次相符以及它是否属于正确的管理区。虽然链表中的页面还不太可能来自于其他管理区,但是如果调用 swap_out() 将页面直接从进程页表中释放,这种情况还是有可能发生的。

283 这是一个处于正确次和管理区的页面,所以从链表中移除它。

284 将链表的页面块数量减 1。

285 设置页面计数为 1,因为它将被释放。

286 设置 page 因为它可能返回。需要 tmp 在下一块中释放局部链表中的剩余页。

288~391 进行在 __free_pages_ok() 中相同的检查以确保释放该页的安全。

305 如果当前页面不在特定的次和管理区中则移到链表中的下一页。

```

308 nr_pages = current->nr_local_pages;
309 /* free in reverse order so that the global
   * order will be lifo */
310 while ((entry = local_pages->prev) != local_pages) {
311 list_del(entry);

```

```

312 tmp = list_entry(entry, struct page, list);
313 __free_pages_ok(tmp, tmp->index);
314 if (!nr_pages--)
315     BUG();
316 }
317 current->nr_local_pages = 0;
318 }
319 out:
320 *freed = __freed;
321 return page;
322 }

```

这一块释放链表中的剩余页面。

308 获取那些待释放的页面块数量。

310 循环直到 local_pages 链表变为空。

311 从链表中移除该页面块。

312 获取 struct page 对应的表项。

313 利用 __free_pages_ok() (见 F. 3.2 小节) 释放页面。

314~315 如果页面块的引用计数达到 0, 而页面还在链表中, 则意味着计数在某个地方被严重地打乱, 或者某个人手动将页面加入到 local_pages 链表中, 所以这里调用 BUG()。

317 设置页面块数量为 0, 因为它们都将被释放。

320 更新 freed 参数告知调用者总共释放了多少页面。

321 返回请求次和管理区的页面块。如果释放失败, 这里将返回 NULL。

F. 2 分配辅助函数

这一节包含各种伙伴分配器用于分配页面的辅助函数和宏。它们几乎不作“实际”的工作, 仅为了方便程序员而设。

F. 2. 1 函数: alloc_page() (include/linux/mm.h)

这个小宏仅调用 alloc_pages(), 返回一个 0 次页面。它的声明如下:

```
449 #define alloc_page(gfp_mask) alloc_pages(gfp_mask, 0)
```

F. 2.2 函数: __get_free_page() (include/linux/mm.h)

这个小宏调用 __get_free_pages(), 返回一个 0 次页面, 它的声明如下:

```
454 #define __get_free_page(gfp_mask) \
455 __get_free_pages((gfp_mask),0)
```

F. 2.3 函数: __get_free_pages() (include/page_alloc.c)

这个函数是为那些不想操心页面而仅返回它们可用地址的调用者准备的。它的声明如下:

```
428 unsigned long __get_free_pages(unsigned int gfp_mask,
429                                unsigned int order)
430 {
431     struct page * page;
432
433     page = alloc_pages(gfp_mask, order);
434     if (! page)
435         return 0;
436 }
```

431 alloc_pages() 完成分配页面块的工作, 见 F. 1.1 小节。

433~434 保证页面有效。

435 page_address() 返回页面的物理地址。

F. 2.4 函数: __get_dma_pages() (include/linux/mm.h)

这是设备驱动器主要关心的函数。它从 DMA 设备中返回适合使用 ZONE_DMA 中的内存。它的声明如下:

```
457 #define __get_dma_pages(gfp_mask, order) \
458 __get_free_pages((gfp_mask) | GFP_DMA,(order))
```

458 gfp_mask 与 GFP_DMA 进行或操作, 告知分配器从 ZONE_DMA 中分配。

F. 2.5 函数: get_zeroed_page() (mm/page_alloc.c)

这个函数分配一页,然后将其内容清零,它的声明如下:

```
438 unsigned long get_zeroed_page(unsigned int gfp_mask)
439 {
440 struct page * page;
441
442 page = alloc_pages(gfp_mask, 0);
443 if (page) {
444 void * address = page_address(page);
445 clear_page(address);
446 return (unsigned long) address;
447 }
448 return 0;
449 }
```

438 gfp_mask 是影响分配器行为的标志位。

442 alloc_pages()完成分配器页面块的工作,见 F. 1.1 小节。

444 page_address()返回页面的物理地址。

445 clear_page()将页面内容清零。

446 返回清零后的页面。

F. 3 释放页面

F. 3.1 函数: __free_pages() (mm/page_alloc.c)

这个函数的调用图如图 6.4 所示。很容易误解的是,alloc_pages()对应的函数并不是 free_pages(),而是__free_pages()。free_pages()是一个以地址为参数的辅助函数。它将在后一节讨论。

```
451 void __free_pages(struct page * page, unsigned int order)
452 {
453 if (!PageReserved(page) && put_page_testzero(page))
454 __free_pages_ok(page, order);
455 }
```

451 参数是我们将释放的 page 和块处于的次。

453 有效性检查。PageReserved()表示页面由引导内存分配器保留。Put_page_testzero()仅是一个对 atomic_dec_and_test()的宏封装。它将使用计数减 1, 保证它为 0。

454 调用函数来完成所有的复杂工作。

F.3.2 函数: __free_pages_ok() (mm/page_alloc.c)

这个函数将完成实际的释放页面工作, 并在可能的情况下合并伙伴。

```

81 static void FASTCALL(__free_pages_ok (struct page * page,
82                                     unsigned int order));
83 {
84     unsigned long index, page_idx, mask, flags;
85     free_area_t * area;
86     struct page * base;
87     zone_t * zone;
88
89     if (PageLRU(page)) {
90         if (unlikely(in_interrupt()))
91             BUG();
92         lru_cache_del(page);
93     }
94
95     if (page->buffers)
96         BUG();
97     if (page->mapping)
98         BUG();
99     if (! VALID_PAGE(page))
100         BUG();
101    if (PageLocked(page))
102        BUG();
103    if (PageActive(page))
104        BUG();
105    if (PageReserved(page))
106        Put_page_testzero(page);
107    page->flags &= ~((1<<PG_referenced) | (1<<PG_dirty));

```

82 参数是待释放页面块的开始和待释放页面的次数。

93~97 在标志 I/O 时, LRU 中的脏页面将仍然设置有 LRU 位。一旦 I/O 完成, 它就会被释放, 所以现在必须从 LRU 链表中移除。

99~108 有效性检查。

109 由于页面现在空闲,没有在使用中,这个标志位表示页面已经被引用,而且是必须被清洗的脏页面。

```

110
111 if (current->flags & PF_FREE_PAGES)
112 goto local_freelist;
113 back_local_freelist;
114
115 zone = page_zone(page);
116
117 mask = (~0UL) << order;
118 base = zone->zone_mem_map;
119 page_idx = page - base;
120 if (page_idx & ~mask)
121 BUG();
122 index = page_idx >> (1 + order);
123
124 area = zone->free_area + order;
125

```

111~112 如果设置了该标志,这些已经释放了的页面将保存在释放它们的进程中。如果调用者是它自己在释放页面,而不是等待 kswapd 来释放,在分配页面时这里调用 balance_classzone()(见 F. 1.6 小节)。

115 页面所属管理区用页面标志位编码。宏 page_zone()返回该管理区。

117 有关掩码计算的讨论在随书附带的文档中。它基本上与伙伴系统的地址计算有关。

118 base 是这个 zone_mem_map 的起始端。对伙伴计算而言,它与地址 0 有关,这样地址就是 2 的幂。

119 page_idx 视 zone_mem_map 为一个由页面组成的数组,这是映射图中的页索引。

120~121 如果索引不是 2 的幂,则肯定是某个地方出现严重错误,伙伴的计算将不会进行。

122 index 是 free_area→map 的位索引。

124 area 是存储空闲链表和映射图的区域,其中映射图是释放页面的有序块。

```

126 spin_lock_irqsave(&zone->lock, flags);
127
128 zone->free_pages -= mask;
129
130 while (mask + (1 << (MAX_ORDER-1))) {

```

```

131 struct page * buddy1, * buddy2;
132
133 if (area >= zone->free_area + MAX_ORDER)
134 BUG();
135 if (! __test_and_change_bit(index, area->map))
136 /*
137 * the buddy page is still allocated.
138 */
139 break;
140 /*
141 * Move the buddy up one level.
142 * This code is taking advantage of the identity:
143 * -mask = 1 + ~mask
144 */
145 buddy1 = base + (page_idx ^ -mask);
146 buddy2 = base + page_idx;
147 if (BAD_RANGE(zone, buddy1))
148 BUG();
149 if (BAD_RANGE(zone, buddy2))
150 BUG();
151
152 list_del(&buddy1->list);
153 mask <<= 1;
154 area++;
155 index >>= 1;
156 page_idx &= mask;
157 }

```

126 管理区将改变,所以这里上锁。由于中断处理程序可能在这个路径上分配页面,所以这个锁是一个中断安全的锁。

128 mask 计算的另一个副作用是-mask 是待释放的页面数。

130~157 分配器将不断地试着合并块直到不能再合并,或者到达了可以合并的最高次。对合并的每一次序块,mask 都将调整。当到达了可以合并的最高次的时候,while 循环将为 0 并退出。

133~134 如果发生什么意外,mask 被损坏,这个检查将保证 free_area 不会超过末端读。

135 表示伙伴对的位置位。如果以前该位是 0,则两个伙伴都在使用中。因此,在这个伙伴释放后,另外一个正在使用中,所以不能合并。

145~146 这两个地址的计算在第 6 章讨论。

147~150 有效性检查保证页面在正确的 zone_mem_map 中,而且实际上属于这个管理区。

152 伙伴已经被释放,所以这里将其从包含它的链表中移除。

153~156 准备检查待合并的高次伙伴。

153 将掩码左移 1 位到次 2^{k+1} 。

154 area 是一个数组内指针,所以 area++ 移到下一个下标。

155 高次位图的索引。

156 待合并 zone_mem_map 中的页面索引。

```

158 list_add(&(base + page_idx)->list, &area->free_list);
159
160 spin_unlock_irqrestore(&zone->lock, flags);
161 return;
162
163 local_freelist:
164 if (current->nr_local_pages)
165 goto back_local_freelist;
166 if (in_interrupt())
167 goto back_local_freelist;
168
169 list_add(&page->list, &current->local_pages);
170 page->index = order;
171 current->nr_local_pages++;
172 }
```

158 由于尽可能多地合并已经完成,而且释放了一个新页面块,所以这里将其加入到该次的 free_list 中。

160~161 对管理区的改变已经完成,所以这里释放锁并返回。

163 这是在页面没有释放到主页面池时的代码路径,它将页面保留给释放的进程。

164~165 如果进程已经有保留页面,则这里不允许再保留页面,所以返回。这里很不寻常,因为 balance_classzone() 假设多于一个页面块可能从该链表上返回。这很有可能过虑了,但是如果释放的第一个页面是同一次的,而 balance_classzone() 请求管理区,则这里仍然可以工作。

166~167 一个中断没有进程上下文,所以它必须象平常一样释放。现在还不明白这里的中断如何结束。这里的检查似乎是假的,而且不可能为真的。

169 将页面块加入到链表中处理 local_pages。

170 记录分配的次数,从而方便后面的释放操作。

171 将 nr_local_pages 使用计数加 1。

F. 4 释放辅助函数

这些函数与页面分配的辅助函数非常相似, 因为它们也不完成“实际”的工作, 它们依赖于 __free_pages() 函数来完成实际的释放。

F. 4. 1 函数: free_pages() (mm/page_alloc.c)

这个函数以一个地址, 而不是以一个页面作为参数来进行释放操作。它的声明如下:

```
457 void free_pages(unsigned long addr, unsigned int order)
458 {
459 if (addr != 0)
460 __free_pages(virt_to_page(addr), order);
461 }
```

460 这个函数在 F. 3. 1 中讨论。宏 virt_to_page() 返回 addr 的 struct page。

F. 4. 2 函数: __free_page() (include/linux/mm.h)

这个小宏仅调用函数 __free_pages() (见 F. 3. 1 节), 参数为 0 次和一个页面。它的声明如下:

```
472 #define __free_page(page) __free_pages((page), 0)
```

F. 4. 3 函数: free_page() (include/linux/mm.h)

这个小宏仅调用 free_pages()。这个宏与 __free_page() 的主要区别在于这个函数以一个虚拟地址为参数, 而 __free_page() 以一个 struct page 为参数, 它的声明如下:

```
472 #define free_page(addr) free_pages((addr), 0)
```

附录 G

不连续内存分配

目 录

| | |
|-----------------------------------|-----|
| G. 1 分配一块不连续区域 | 442 |
| G. 1. 1 函数:vmalloc() | 442 |
| G. 1. 2 函数:__vmalloc() | 442 |
| G. 1. 3 函数:get_vm_area() | 443 |
| G. 1. 4 函数:vmalloc_area_pages() | 445 |
| G. 1. 5 函数:__vmalloc_area_pages() | 446 |
| G. 1. 6 函数:alloc_area_pmd() | 447 |
| G. 1. 7 函数:alloc_area_pte() | 448 |
| G. 1. 8 函数:vmap() | 450 |
| G. 2 释放一块不连续区域 | 452 |
| G. 2. 1 函数:vfree() | 452 |
| G. 2. 2 函数:vmfree_area_pages() | 453 |
| G. 2. 3 函数:free_area_pmd() | 454 |
| G. 2. 4 函数:free_area_pte() | 455 |

G. 1 分配一块非连续的区域

G. 1. 1 函数: `vmalloc()` (`include/linux/vmalloc.h`)

这个函数的调用图如图 7.2 所示。下面宏之间的差别仅在于使用的 GFP_ 标志位(见 6.4 节)不同。`size` 参数是由 `_vmalloc()` 对齐的分页(见 G. 1. 2 小节)。

```

37 static inline void * vmalloc (unsigned long size)
38 {
39     return __vmalloc(size, GFP_KERNEL | __GFP_HIGHMEM, PAGE_KERNEL);
40 }
41
42 static inline void * vmalloc_dma (unsigned long size)
43 {
44     return __vmalloc(size, GFP_KERNEL|GFP_DMA, PAGE_KERNEL);
45 }
46
47 static inline void * vmalloc_32(unsigned long size)
48 {
49     return __vmalloc(size, GFP_KERNEL, PAGE_KERNEL);
50 }
51
52 static inline void * vmalloc_nopage(unsigned long size)
53 {
54     return __vmalloc(size, GFP_KERNEL, PAGE_KERNEL);
55 }
```

37 这个标志位表明在需要时使用 ZONE_NORMAL 或 ZONE_HIGHMEM。

46 这个标志位表明仅从 ZONE_DMA 中分配。

55 仅 ZONE_NORMAL 中的物理页面会被分配。

G. 1. 2 函数: `__vmalloc()` (`mm/vmalloc.c`)

这个函数有 3 个任务。将请求大小转化为页面数, 调用 `get_vm_area()` 找到该请求的一个区域, 使用 `vmalloc_area_pages()` 分配页面的 PTE。

```

261 void * __vmalloc (unsigned long size, int gfp_mask, pgprot_t prot)
262 {
263     void * addr;
264     struct vm_struct * area;
265
266     size = PAGE_ALIGN(size);
```

```

267 if (! size || (size >> PAGE_SHIFT) > num_physpages)
268 return NULL;
269 area = get_vm_area(size, VM_ALLOC);
270 if (! area)
271 return NULL;
272 addr = area->addr;
273 if (_vmalloc_area_pages(VMALLOC_VMADDR(addr), size, gfp_mask,
274 prot, NULL)) {
275 vfree(addr);
276 return NULL;
277 }
278 return addr;
279 }

```

261 参数是要分配的大小,分配时使用的 GFP_标志位和对 PTE 采用何种保护手段。

266 将 size 与页面大小对齐。

267 有效性检查,保证大小不是 0,请求的大小不会大于已请求的物理页面数。

269 利用 get_vm_area()(见 G.1.3 小节)找到存放分配虚拟地址空间中的一块区域。

272 addr 字段已经由 get_vm_area()填充。

273 利用 __vmalloc_area_pages()分配所需的 PTE 表项。如果失败,则返回一个非 0 值-ENOMEM。

275~276 如果分配失败,则这里释放所有的 PTE、页面和区域描述符。

278 返回已分配的区域地址。

G.1.3 函数: get_vm_area() (mm/vmalloc.c)

为了给 vm_struct 分配一块区域,使用 kmalloc()请求 slab 分配器提供所需的内存。然后线性查找 vm_struct 链表,找到一块满足请求的足够大的区域,且在区域尾部包括一个页面填充。

```

195 struct vm_struct * get_vm_area(unsigned long size,
196                                     unsigned long flags)
197 {
198     unsigned long addr, next;
199     struct vm_struct ** p, * tmp, * area;
200     area = (struct vm_struct *) kmalloc(sizeof(*area), GFP_KERNEL);
201     if (! area)

```

```

202 return NULL;
203
204 size += PAGE_SIZE;
205 if(! size) {
206 kfree (area);
207 return NULL;
208 }
209
210 addr = VMALLOC_START;
211 write_lock(&vmlist_lock);
212 for (p = &vmlist; (tmp = *p) ; p = &tmp->next) {
213 if ((size + addr) < addr)
214 goto out;
215 if (size + addr <= (unsigned long) tmp->addr)
216 break;
217 next = tmp->size + (unsigned long) tmp->addr;
218 if (next > addr)
219 addr = next;
220 if (addr > VMALLOC_END-size)
221 goto out;
222 }
223 area->flags = flags;
224 area->addr = (void *)addr;
225 area->size = size;
226 area->next = *p;
227 *p = area;
228 write_unlock(&vmlist_lock);
229 return area;
230
231 out:
232 write_unlock(&vmlist_lock);
233 kfree(area);
234 return NULL;
235 }

```

195 参数是必须是页面大小的倍数的请求区域的大小、区域标志位、VM_ALLOC 或 VM_IOTEMAP。

200~202 允许为 vm_struct 描述符结构分配空间。

204 填充请求,在区域间留出一页空隙,这里是保护防止覆写。

205~206 保证 size 在溢出填充时不会为 0, 如果中间出了什么错, 这里将释放刚才分配的 area, 并返回 NULL。

210 在 vmalloc 地址空间的首部开始搜索。

211 上锁链表。

212~222 遍历链表, 寻找一块对请求足够大的区域。

213~214 检查保证没有到达可寻址区域范围的末端。

215~216 如果所请求的区域不能在当前地址和下一个地址之间取出, 查找结束。

217 保证地址不会超过 vmalloc 地址空间的末端。

223~225 复制区域信息。

226~227 将新区域链入链表中。

228~229 解锁链表并返回。

231 如果不能满足请求则到达这个标记。

232 解锁链表。

223~234 释放区域描述符使用的内存并返回。

G.1.4 函数: `vmalloc_area_pages()` (`mm/vmalloc.c`)

这个函数只是对 `_vmalloc_area_pages()` 的封装。这个函数为了与旧的内核兼容才存在。改变名字是为了反映新函数 `_vmalloc_area_pages()` 可以使用一个页面数组来插入到页表中。

```
189 int vmalloc_area_pages(unsigned long address, unsigned long size,
190 int gfp_mask, pgprot_t prot)
191 {
192 return __vmalloc_area_pages(address, size, gfp_mask,
193 prot, NULL);
193 }
```

192 以通用的参数调用 `_vmalloc_area_pages()`。 `pages` 数组传递为 NULL, 因为后面将在需要时分配页面。

G.1.5 函数: `_vmalloc_area_pages()` (`mm/vmalloc.c`)

这是标准的页面遍历函数的开始部分。这个顶层函数将遍历在一定地址范围内的所有 PGD。对每个 PGD, 它将调用 `pmd_alloc()` 来分配一个 PMD 目录, 然后调用 `alloc_area_pmd()` 分配目录。

```
155 static inline int __vmalloc_area_pages (unsigned long address,
```

```

156 unsigned long size,
157 int gfp_mask,
158 pgprot_t prot,
159 struct page *** pages)
160 {
161 pgd_t * dir;
162 unsigned long end = address + size;
163 int ret;
164
165 dir = pgd_offset_k(address);
166 spin_lock(&init_mm.page_table_lock);
167 do {
168 pmd_t * pmd;
169
170 pmd = pmd_alloc(&init_mm, dir, address);
171 ret = -ENOMEM;
172 if (!pmd)
173 break;
174
175 ret = -ENOMEM;
176 if (alloc_area_pmd(pmd, address, end - address,
177 gfp_mask, prot, pages))
178 break;
179 address = (address + PGDIR_SIZE) & PGDIR_MASK;
180 dir++;
181
182 ret = 0;
183 } while (address && (address < end));
184 spin_unlock(&init_mm.page_table_lock);
185 flush_cache_all();
186 return ret;
187 }

```

155 参数如下：

- address 是 PMD 分配的起始地址。
- size 是区域的大小。
- gfp_mask 是 alloc_pages() (见 F. 1.1 小节) 的所有 GFP_ 标志位。
- prot 是对 PTE 表项的保护方式。

- `pages` 是一个用于插入的页面数组, 它不是一次性调用 `alloc_area_pte()` 分配的。只有 `vmap()` 接口使用数组传递。

162 末尾地址是起始地址加上大小。

165 获取起始地址的 PGD 表项。

166 上锁内核引用页表。

167~183 对地址范围内的每个 PGD, 这里分配一个 PMD 目录, 然后调 `alloc_area_pmd()` (见 G. 1.6 小节)。

170 分配一个 PMD 目录。

176 调用 `alloc_area_pmd()` (见 G. 1.6 节), 它将为 PMD 中每个 PTE 槽分配一个 PTE。

179 `address` 变成下一个 PGD 表项的基地址。

180 将 `dir` 移至下一个 PGD 表项。

184 释放内核页表锁。

185 `flush_cache_all()` 将清理所有的 CPU 高速缓存。这是必要的, 因为内核页表已经改变了。

186 返回成功。

G. 1.6 函数: `alloc_area_pmd()` (`mm/vmalloc.c`)

这是在地址范围内为分配 PTE 表项而进行的标准页表遍历的第 2 个阶段。对 PGD 中给定地址的每个 PMD, `pte_alloc()` 将创建 PTE 目录, 然后调用 `alloc_area_pte()` 分配物理页面。

```

132 static inline int alloc_area_pmd(pmd_t * pmd, unsigned long
133 address, unsigned long size, int gfp_mask,
134 pgprot_t prot, struct page *** pages)
135 {
136     unsigned long end;
137
138     address &= ~PGDIR_MASK;
139     end = address + size;
140     if (end > PGDIR_SIZE)
141         end = PGDIR_SIZE;
142     do {
143         pte_t * pte = pte_alloc(&init_mm, pmd, address);
144         if (!pte)
145             return -ENOMEM;
146         if (alloc_area_pte(pte, address, end - address,
147 gfp_mask, prot, pages))

```

```

148 return -ENOMEM;
149 address = (address + PMD_SIZE) & PMD_MASK;
150 pmd++;
151 } while (address < end);
152 return 0;
152 }

```

132 参数如下：

- pmd 是需要分配的 PMD。
- address 是开始的起始地址。
- size 是为 PMD 分配的区域大小。
- gfp+mask 是给 alloc_pages()(见 F. 1.1 小节)的 GFP_flag。
- prot 是对 PTE 表项的保护方式。
- pages 是一个可选的页面数组,它用于替代单独地一次一页的分配。

138 将 PGD 的起始地址页对齐。

139~141 计算分配的末端,或者 PGD 的末端,无论哪个先出现。

142~151 对给定地址范围内的每个 PMD,这里分配一个 PTE 目录,然后调用 alloc_area_pte()(见 G. 1.7 小节)。

143 分配 PTE 目录。

146~147 调用 alloc_area_pte(),如果页面数组还没有提供 pages,将分配物理页。

149 address 变成下一个 PMD 表项的基地址。

150 将 pmd 移到下一个 PMD 表项。

152 返回成功。

G. 1.7 函数: alloc_area_pte() (mm/vmalloc.c)

这是页表遍历的最后一个阶段。对给定 PTE 目录中和地址范围的各个 PTE,分配一页以及相关的 PTE。

```

95 static inline int alloc_area_pte (pte_t * pte, unsigned long address,
96 unsigned long size, int gfp_mask,
97 pgprot_t prot, struct page *** pages)
98 {
99 unsigned long end;
100
101 address &= ~PMD_MASK;
102 end = address + size;

```

```

103 if (end > PMD_SIZE)
104 end = PMD_SIZE;
105 do {
106     struct page * page;
107
108     if (! pages) {
109         spin_unlock(&init_mm.page_table_lock);
110         page = alloc_page(gfp_mask);
111         spin_lock(&init_mm.page_table_lock);
112     } else {
113         page = (* pages);
114         (* pages)++;
115
116     /* Add a reference to the page so we can free later */
117     if (page)
118         atomic_inc(&page->count);
119
120 }
121 if (! pte_none(* pte))
122     printk(KERN_ERR "alloc_area_pte: page already exists\n");
123 if (! page)
124     return -ENOMEM;
125 set_pte(pte, mk_pte(page, prot));
126 address += PAGE_SIZE;
127 pte++;
128 } while (address < end);
129 return 0;
130 }

```

101 将地址与 PMD 目录对齐。

103~104 末尾地址是请求的末尾或者目录的末尾,无论哪个先出现。

105~128 遍历该页面中的每个 PTE。如果提供了 pages 数组,则使用从它里面的页面来构建表格。否则,分别分配每个。

108~111 如果没有提供一个 pages 数组,则在这里解锁内核引用页表,利用 alloc_page() 分配一个页面,然后重新获取自旋锁。

112~120 如果不是这样,则从数组中取出一页,由于它将被插入到引用页表中,所以增加它的引用计数。

121~122 如果 PTE 正在使用中,则意味着在某个地方 vmalloc 区域中的区域重叠了。

- 123~124 如果没有物理页面可用则返回失败。
- 125 在 PTE 中设置 page 的相应保护位(prot)。
- 126 address 变成下一个 PTE 的地址。
- 127 移到下一个 PTE。
- 128 返回成功。

G. 1.8 函数: vmap() (mm/vmalloc.c)

这个函数允许由调用者提供的 pages 数组插入到 vmalloc 地址空间,这个在 2.4.22 中没有使用,我怀疑是从 2.6.x 起,linux 为了兼容某种声音子系统内核而设立的。

```

281 void * vmap(struct page ** pages, int count,
282 unsigned long flags, pgprot_t prot)
283 {
284 void * addr;
285 struct vm_struct * area;
286 unsigned long size = count << PAGE_SHIFT;
287
288 if (! size || size > (max_mapnr << PAGE_SHIFT))
289 return NULL;
290 area = get_vm_area(size, flags);
291 if (! area)
292 return NULL;
293 }
294 addr = area->addr;
295 if (_vmalloc_area_pages(VMALLOC_VMADDR(addr), size, 0,
296 prot, &pages)) {
297 vfree(addr);
298 return NULL;
299 }
300 return addr;
301 }
```

281 参数如下:

- pages 是调用者提供的待插入的页面数组。
- count 是数组中的页面数。
- flags 是用于 vm_struct 的标志位。
- prot 是设置 PTE 的保护位。

286 在数组的基础上计算要创建的区域的大小(以字节计)。

288~289 保证区域大小不会超过限制。

290~293 使用 `get_vm_area()` 来找到一个足够大的映射区域。如果没有找到, 则返回 `NULL`。

294 获取区域的虚拟地址。

295 利用 `__vmalloc_area_pages()` 向页表中插入数组。

297 如果插入失败, 则这里释放区域, 并返回 `NULL`。

298 返回新映射区域的虚拟地址。

G.2 释放一块非连续区域

G.2.1 函数: `vfree()` (`mm/vmalloc.c`)

这个函数的调用图如图 7.4 所示。这是一个负责释放非连续内存区域的高层函数。它进行简单的有效性检查, 然后找到请求 `addr` 的 `vm_struct`, 一旦找到, 则调用 `vmfree_area_pages()`。

```

237 void vfree(void * addr)
238 {
239     struct vm_struct ** p, * tmp;
240
241     if (!addr)
242         return;
243     if ((PAGE_SIZE-1) & (unsigned long)addr) {
244         printk (KERN_ERR
245                 "Trying to vfree() bad address (%p)\n", addr);
246     }
247     write_lock(&vmlist_lock);
248     for (p = &vmlist ; (tmp = *p) ; p = &tmp->next) {
249         if (tmp->addr == addr) {
250             *p = tmp->next;
251             vmfree_area_pages(VMALLOC_VMADDR(tmp->addr),
252                               tmp->size);
252             write_unlock(&vmlist_lock);
253             kfree(tmp);

```

```

254 return;
255 }
256 }
257 write_unlock(&vmlist_lock);
258 printk (KERN_ERR
           "Trying to vfree() nonexistent vm area (%p)\n", addr);
259 }

```

237 参数为 get_vm_area() (见 G. 1.3 小节) 返回给 vmalloc() 或 ioremap() 的地址。

241~243 忽略 NULL 地址。

243~246 检查以确定地址是否页对齐, 同时也是对地址是否有效的一次快速检查。

247 获取 vmlist 的写锁。

248 遍历 vmlist, 查找 addr 的正确的 vm_struct。

249 如果这是一个正确的地址, 则……

250 从 vmlist 链表中移除这个区域。

251 释放所有与地址范围有关的页面。

252 释放 vmlist 锁。

253 释放用于 vm_struct 的内存并返回。

257~258 没有找到 vm_struct。这里释放锁, 然后答应有关失败释放的信息。

G. 2.2 函数: vmfree_area_pages() (mm/vmalloc.c)

这是遍历页表中与某个地址范围有关的所有页面和 PTE 的第一阶段。它负责遍历相关的 PGD 以及刷新 TLB。

```

80 void vmfree_area_pages(unsigned long address, unsigned long size)
81 {
82 pgd_t * dir;
83 unsigned long end = address + size;
84
85 dir = pgd_offset_k(address);
86 flush_cache_all();
87 do {
88 free_area_pmd(dir, address, end - address);
89 address = (address + PGDIR_SIZE) & PGDIR_MASK,
90 dir++;
91 } while (address && (address < end));
92 flush_tlb_all();

```

93 }

80 参数是起始的 address 和区域的 size。

82 地址空间末端是起始地址加上大小。

85 获取地址范围的第一个 PGD。

86 刷新高速缓存 CPU, 这样就不会在将被删除的页面上发生高速缓存命中。在许多体系结构中(包括 x86), 这是一个空操作。

87 调用 free_area_pmd()(见 G. 2.3 小节)完成遍历页表的第二阶段。

89 address 变成下一个 PGD 的开始地址。

90 移到下一个 PGD。

92 由于页表现在已经改变了, 所以刷新 TLB。

G. 2.3 函数: free_area_pmd() (mm/vmalloc.c)

这是页表遍历的第 2 个阶段。对这个目录中的每个 PMD, 它调用 free_area_pte()来释放页面和 PTE。

```

56 static inline void free_area_pmd(pgd_t * dir,
57                                 unsigned long address,
58                                 unsigned long size)
59 {
60     pmd_t * pmd;
61     unsigned long end;
62
63     if (pgd_none(*dir))
64         return;
65     if (pgd_bad(*dir)) {
66         pgd_ERROR(*dir);
67         pgd_clear(dir);
68         return;
69     }
70     pmd = pmd_offset(dir, address);
71     address &= ~PGDIR_MASK;
72     end = address + size;
73     if (end > PGDIR_SIZE)
74         end = PGDIR_SIZE;
75     do {
76         free_area_pte(pmd, address, end - address);
77     }
78 }
```

```

75 address = (address + PMD_SIZE) & PMD_MASK;
76 pmd++;
77 } while (address < end);
78 }

```

56 参数是进入的 PGD、起始地址和区域长度。

61~62 如果没有 PGD，则返回。在分配失败过程中调用 vfree()（见 G.2.1 小节）后这可能发生。

63~67 如果没有表项，则 PGD 可能已损坏，这里将其标记为只读或者访问过或者脏的。

68 获取地址范围的第一个 PMD。

69 将地址与 PGD 对齐。

70~72 end 或者是待释放空间的末端，或者是这个 PGD 的末端，无论哪个首先出现。

73 对每个 PMD，这里调用 free_area_pte()（见 G.2.4 小节）释放 PTE 表项。

75 address 是下一个 PMD 的基地址。

76 移到下一个 PMD。

G.2.4 函数：free_area_pte() (mm/vmalloc.c)

这是页表遍历的最后一个阶段。对地址范围内 PMD 的每个 PTE，它将释放 PTE 和相关的页面。

```

22 static inline void free_area_pte(pmd_t * pmd, unsigned long address,
                                  unsigned long size)
23 {
24     pte_t * pte;
25     unsigned long end;
26
27     if (pmd_none(*pmd))
28         return;
29     if (pmd_bad(*pmd)) {
30         pmd_ERROR(*pmd);
31         pmd_clear(pmd);
32         return;
33     }
34     pte = pte_offset(pmd, address);
35     address &= ~PMD_MASK;
36     end = address + size;
37     if (end > PMD_SIZE)

```

```

38 end = PMD_SIZE;
39 do {
40 pte_t page;
41 page = ptep_get_and_clear(pte);
42 address += PAGE_SIZE;
43 pte++;
44 if (pte_none(page))
45 continue;
46 if (pte_present(page)) {
47 struct page *ptpage = pte_page(page);
48 if (VALID_PAGE(ptpage) &&
        (!PageReserved(ptpage)))
49 __free_page(ptpage);
50 continue;
51 }
52 printk(KERN_CRIT
        "Whee.. Swapped out page in kernel page table\n");
53 } while (address < end);
54 }

```

22 参数是将从 PTE 中释放的 PMD、待释放区域的起始地址和大小。

27~28 如果区域是从一个失败的 vmaloc() 而来，则 PMD 将不存在。

29~33 如果 PMD 不在主存，则它被损坏了，这里标识为只读或者脏的或者访问过的。

34 pte 是地址范围的第一个 PTE。

35 将地址与 PMD 对齐。

38~53 遍历所有的 PTE，进行检查，然后释放相关页面的 PTE。

41 ptep_get_and_clear() 将从页表中移除一个 PTE，然后返回该 PTE 给调用者。

42 address 是下一个 PTE 的基地址。

43 移到下一个 PTE。

44 如果没有 PTE，则继续。

46~51 如果页面存在，则这里进行基本的检查，然后释放该页面。

47 pte_page 使用全局 mem_map 来找到 PTE 的 struct page。

48~49 保证页面是一个有效页面，且没有被保留，然后调用 __free_page() 释放物理页面。

50 继续下一个 PTE。

52 如果到达这一行，则在某个时候换出了内核地址空间中的 PTE，内核内存是不可换出的，所以这是一个很严重的错误。

附录 H

Slab 分配器

目 录

| | |
|---|-----|
| H. 1 高速缓存控制 | 459 |
| H. 1. 1 创建高速缓存 | 459 |
| H. 1. 1. 1 函数: kmem_cache_create() | 459 |
| H. 1. 2 计算 slab 上的对象数 | 467 |
| H. 1. 2. 1 函数: kmem_cache_estimate() | 467 |
| H. 1. 3 收缩高速缓存 | 469 |
| H. 1. 3. 1 函数: kmem_cache_shrink() | 469 |
| H. 1. 3. 2 函数: __kmem_cache_shrink() | 470 |
| H. 1. 3. 3 函数: __kmem_cache_shrink_locked() | 471 |
| H. 1. 4 销毁高速缓存 | 472 |
| H. 1. 4. 1 函数: kmem_cache_destroy() | 472 |
| H. 1. 5 回收高速缓存 | 473 |
| H. 1. 5. 1 函数: kmem_cache_reap() | 473 |
| H. 2 Slabs | 479 |
| H. 2. 1 存储 Slab 描述符 | 479 |
| H. 2. 1. 1 函数: kmem_cache_slabmgmt() | 479 |
| H. 2. 1. 2 函数: kmem_find_general_cachep() | 480 |
| H. 2. 2 创建 Slab | 481 |
| H. 2. 2. 1 函数: kmem_cache_grow() | 481 |
| H. 2. 3 销毁 Slab | 485 |

| | |
|---|-----|
| H. 2. 3. 1 函数: kmem_slab_destroy() | 485 |
| H. 3 对象 | 486 |
| H. 3. 1 初始化 Slab 中的对象 | 486 |
| H. 3. 1. 1 函数: kmem_cache_init_objs() | 486 |
| H. 3. 2 分配对象 | 488 |
| H. 3. 2. 1 函数: kmem_cache_alloc() | 488 |
| H. 3. 2. 2 函数: __kmem_cache_alloc(UP Case)() | 488 |
| H. 3. 2. 3 函数: __kmem_cache_alloc(SMP Case)() | 489 |
| H. 3. 2. 4 函数: kmem_cache_alloc_head() | 491 |
| H. 3. 2. 5 函数: kmem_cache_alloc_one() | 492 |
| H. 3. 2. 6 函数: kmem_cache_alloc_one_tail() | 492 |
| H. 3. 2. 7 函数: kmem_cache_alloc_batch() | 494 |
| H. 3. 3 释放对象 | 495 |
| H. 3. 3. 1 函数: kmem_cache_free() | 495 |
| H. 3. 3. 2 函数: __kmem_cache_free(UP Case)() | 496 |
| H. 3. 3. 3 函数: __kmem_cache_free(SMP Case)() | 496 |
| H. 3. 3. 4 函数 kmem_cache_free_one() | 497 |
| H. 3. 3. 5 函数 free_block() | 499 |
| H. 3. 3. 6 函数: __free_block() | 500 |
| H. 4 指定大小的高速缓存 | 501 |
| H. 4. 1 初始化指定大小的高速缓存 | 501 |
| H. 4. 1. 1 函数: kmem_cache_sizes_init() | 501 |
| H. 4. 2 kmalloc() | 502 |
| H. 4. 2. 1 函数: kmalloc() | 502 |
| H. 4. 3 kfree() | 503 |
| H. 4. 3. 1 函数: kfree() | 503 |
| H. 5 Per-CPU 对象高速缓存 | 504 |
| H. 5. 1 启动 Per-CPU 高速缓存 | 504 |
| H. 5. 1. 1 函数: enable_all_cpucaches() | 504 |
| H. 5. 1. 2 函数: enable_cpucache() | 505 |
| H. 5. 1. 3 函数: kmem_tune_cpucache() | 506 |
| H. 5. 2 更新 Per-CPU 信息 | 508 |
| H. 5. 2. 1 函数: smp_call_function_all_cpus() | 509 |

| | |
|-----------------------------------|-----|
| H. 5. 2. 2 函数:do_ccupdate_local() | 509 |
| H. 5. 3 清理 Per-CPU 高速缓存 | 509 |
| H. 5. 3. 1 函数:drain_cpu_caches() | 509 |
| H. 6 初始化 Slab 分配器 | 511 |
| H. 6. 1. 1 函数:kmem_cache_init() | 511 |
| H. 7 与伙伴分配器的接口 | 512 |
| H. 7. 1. 1 函数:kmem_getpages() | 512 |
| H. 7. 1. 2 函数:kmem_freepages() | 512 |

H. 1 高速缓存控制

H. 1. 1 创建高速缓存

H. 1. 1. 1 函数: kmem_cache_create() (mm/slab.c)

这个函数的调用图如图 8.3 所示。这个函数负责创建一个新的高速缓存,然后根据大小进行批量处理。这个批量处理大约包括如下操作:

- 进行基本的有效性检查以防错误使用。
- 如果设置有 CONFIG_SLAB_DEBUG 则进行调试检查。
- 从 cache_cache slab 高速缓存中分配一个 kmem_cache_t。
- 将对象大小对齐为字大小。
- 计算在 slab 中有多少合适的对象。
- 将 slab 大小对齐为硬件高速缓存。
- 计算着色偏移。
- 初始化在高速缓存描述符中的其余字段。
- 将新高速缓存加入到高速缓存链。

```

621 kmem_cache_t *
622 kmem_cache_create (const char * name, size_t size,
623 size_t offset, unsigned long flags,
624 void (* ctor)(void * , kmem_cache_t * , unsigned long),
625 void (* dtor)(void * , kmem_cache_t * , unsigned long))
626 const char * func_nm = KERN_ERR "kmem_create: ";

```

```

627 size_t left_over, align, slab_size;
628 kmem_cache_t * cachep = NULL;
629
633 if ((!name) ||
634 ((strlen(name) >= CACHE_NAMELEN - 1)) ||
635 in_interrupt() ||
636 (size < BYTES_PER_WORD) ||
637 (size > (1 << MAX_OBJ_ORDER) * PAGE_SIZE) ||
638 (dtor && !ctor) ||
639 (offset < 0 || offset > size))
640 BUG();
641

```

这一块进行基本的有效性检查防止错误使用。

622 这个函数的参数如下：

- name 是该高速缓存的可读名。
- size 是一个对象的大小。
- offset 是用于指定高速缓存中对象的边界,但一般设为 0。
- flags 是静态高速缓存标志位。
- ctor 是在 slab 创建时为每个对象调用的构造函数。
- dtor 是相应的销毁函数。销毁函数可以使一个对象回到初始态。

633~640 在创建高速缓存时有各种 bug 的使用方法。

634 这里用于可读名大于最大的高速缓存名(CACHE_MAXELEN)的情况。

635 中断处理程序不能创建高速缓存,因为需要访问中断安全的自旋锁以及信号量。

635 对象大小必须至少是一个字大小。slab 分配器不适合于以单个字节为度量的对象。

637 最大可能的 slab 利用 32 个页面可以创建 $2^{MAX_OBJ_ORDER}$ 个页面数。

638 如果没有构造函数可用,也不能使用销毁函数。

639 偏移不能在 slab 前,也不能超出第 1 个页面的边界。

640 调用 BUG()退出。

```

642 #if DEBUG
643 if ((flags & SLAB_DEBUG_INITIAL) && !ctor) {
645     printk("%sNo con, but init state check
646     requested - %s\n", func_nm, name);
646 flags &= ~SLAB_DEBUG_INITIAL;
647 }
648
649 if ((flags & SLAB_POISON) && ctor) {

```



```

651 printk(" % sPoisoning requested, but con given - % s\n",
           func_nm, name);
652 flags &= ~SLAB_POISON;
653 }
654 #if FORCED_DEBUG
655 if ((size < (PAGE_SIZE)>>3)) &&
    !(flags & SLAB_MUST_HWCACHE_ALIGN))
656 flags |= SLAB_RED_ZONE;
657 if (!ctor)
658 flags |= SLAB_POISON;
659 #endif
660 #endif
670 BUG_ON(flags & ~CREATE_MASK);

```

如果设置了 CONFIG_SLAB_CONFIG, 则这一块进行调试检查。

643~646 SLAB_DEBUG_INIIAL 需要构造函数对对象进行检查以保证它们处于初始状态, 所以, 必须退出一个构造函数。如果没有退出, 则清除该标志。

649~653 slab 可能会被某种已知模式感染, 以保证某个对象在分配之前没有被使用, 但是一个构造函数可能会破坏这种模式, 它会错误地报告一个 bug。如果存在这样一个构造函数, 则这里如果设置了 SLAB_POISON 就移除该标志位。

655~660 仅有很少一部分对象是调试的红色区域。很大的红色区域对象将导致严重的碎片。

661~662 如果没有构造函数, 这里设置感染位。

670 调用所有可以调用的 kmem_cache_create() (见 H. 1.1.1) 来设置 CRATE_MASK 以及所有允许设置的标志位。这里阻止调用者在没有这些标志位的时候使用调试标志位, 如果使用则调用 BUG()。

```

673 cachep =
    (kmem_cache_t *) kmem_cache_alloc(&cache_cache,
    SLAB_KERNEL);
674 if (!cachep)
675     goto oops;
676 memset(cachep, 0, sizeof(kmem_cache_t));

```

673 利用 kmem_cache_alloc() (见 H. 3.2.1) 从 cache_cache 中分配一个高速缓存描述符对象。

674~675 如果内存不足, 则转到 oops, 在那里处理 OOM。

676 用 0 填充对象以防止意外地使用未初始化的数据。

```

682 if (size & (BYTES_PER_WORD - 1)) {
683 size += (BYTES_PER_WORD - 1);
684 size &= ~(BYTES_PER_WORD - 1);
685 printk (" % sForcing size word alignment
- % s\n", func_nm, name);
686 }
687
688 # if DEBUG
689 if (flags & SLAB_RED_ZONE) {
690 flags &= ~SLAB_HWCACHE_ALIGN;
691 size += 2 * BYTES_PER_WORD;
692 }
693 #endif
694 align = BYTES_PER_WORD;
695 if (flags & SLAB_HWCACHE_ALIGN)
696 align = L1_CACHE_BYTES;
697
700 if (size >= (PAGE_SIZE >> 3))
701 flags |= CFLGS_OFF_SLAB;
702
703 if (size < align/2)
704 align /= 2;
705 size = (size + align - 1) & (~ (align - 1));
706
707 }

```

将对象大小对齐于某个字大小边界。

682 如果大小没有与某个字大小边界对齐,则……

683~684 增加一个对象的字大小,然后屏蔽低位。这能有效地将对象大小向上取整到下一个字边界。

685 为了调试打印一个提示信息。

688~697 如果调试可用,则必须稍微改变一下对齐方式。

694 如果 slab 将红色分区的话,就不需要尝试将硬件高速缓存中的东西对齐了。对象的红色分区是从高速缓存边界处移动一个字的偏移。

695 对象的大小增加两个 BYTES_PER_WORD,以在对象的两端标记红色区域。

698 初始化对齐方式为一个字边界。如果调用者请求的是一个 CPU 高速缓存对齐方式,则在这里改变。

699~700 如果有请求,则这里将对象对齐到 L1 CPU 高速缓存。

703 如果对象比较大,在这里存储 slab 描述符 off-slab,这将更好地允许打包对象到 slab 中。

710 如果请求硬件高速缓存对齐,则对象的大小将与硬件高速缓存对齐。

714~715 如果对象符合对齐,则试着将对象打包到高速缓存中的一行。对于那些带有大 L1 高速缓存字节的系统(如 Alpha 和奔腾 4),这很重要。align 将被调整为对齐硬件高速缓存边界的最小值。对大的 L1 高速缓存行,两个或者更多的小对象将可以填入到一行中。例如,从 32 位高速缓存的两个对象将在奔腾 4 中填入一个高速缓存行。

716 将高速缓存大小向上取整为硬件高速缓存边界。

```

724 do {
725     unsigned int break_flag = 0;
726     cal_wastage:
727     kmem_cache_estimate(cachep->gfporder,
728         size, flags,
729         &left_over,
730         &cachep->num);
731     if (break_flag)
732         break;
733     if (!cachep->num)
734         goto next;
735     if (flags & CFLGS_OFF_SLAB &&
736         cachep->num > offslab_limit) {
737         cachep->gfporder--;
738         break_flag++;
739         goto cal_wastage;
740     }
741
742     if (cachep->gfporder >= slab_break_gfp_order)
743         break;
744
745     if ((left_over * 8) <= (PAGE_SIZE << cachep->gfporder))
746         break;
747     next:
748     cachep->gfporder++;
749 } while (1);
750
751 if (!cachep->num) {

```

```

756 printk("kmem_cache_create: couldn't
        create cache %s.\n", name);
757 kmem_cache_free(&cache_cache, cachep);
758 cachep = NULL;
759 goto opps;
760 }

```

计算 slab 中将填入多少个对象,并在需要时调整 slab 大小。

727~728 kmem_cache_estimate()(见 H.1.2.1)计算以 gfp 次序填入 slab 的对象数,并计算剩余的字节数。

729~730 在使用了 offslab slab 描述符后,如果填入 slab 的对象数超过了 slab 存放的对象数,则设置 break_flag。

731~732 页面使用的次数不能超过 MAX_GFP_ORDER(5)。

733~734 如果对象不能填入,则跳转到 next,在那里将增加高速缓存使用的 gfporder。

735 如果在高速缓存中没有 slab 描述符,但对象数超过了 bufctl 的 off-slab 所确定的数目,则……

737 降低使用页面的次数。

738 设置 break_flag 标志位,这样将退出循环。

739 计算新的消耗数。

746~747 除非填充了 0 号对象,slab_break_gfp_order 将是不超过 slab 的次数。这里检查以保证不会超过次数。

749~759 对外部碎片的粗略检查。如果高速缓存的消耗量小于八分之一,则可以接受。

752 如果碎片过多,则这里增加 gfp 次,并重新计算存储的对象数目以及消耗量。

755 在调整后如果对象还是不能填入高速缓存,则不能创建该对象。

757~758 释放高速缓存描述符并设指针指向 NULL。

758 跳转到 oops,在那里仅返回 NULL 指针。

```

761 slab_size = L1_CACHE_ALIGN(
        cachep->num * sizeof(kmem_bufctl_t) +
        sizeof(slab_t));
762
763 if (flags & CFLGS_OFF_SLAB && left_over >= slab_size) {
764     flags &= ~CFLGS_OFF_SLAB;
765     left_over -= slab_size;
770 }

```

这一块将 slab 大小对齐于硬件高速缓存。

761 slab_size 是 slab 描述符的总大小,不是 slab 自身的大小。它是一个固定的 slab_t 结

构,大小为对象数乘以 bufctl 的结果。

767~769 如果有足够的空间留给 slab 描述符,则这里指定为放置 off-slab 描述符,这里移除标志位并更新 left_over 字节量。这样做会影响到高速缓存着色,但是对 off-slab 描述符而言,这不是问题。

```
773 offset += (align - 1);
774 offset &= ~(align - 1);
775 if (! offset)
776 offset = L1_CACHE_BYTES;
777 cachep->colour_off = offset;
778 cachep->colour = left_over/offset;
```

计算着色偏移。

773~774 offset 是请求调用者页面中的偏移。这里保证所请求偏移在使用中的高速缓存的正确对齐方式中。

775~776 如果由于某个原因偏移为 0,则这里设置它与 CPU 高速缓存对齐。

777 这个偏移用于在不同的高速缓存行中存入对象。对创建的每个 slab,将给予不同的颜色偏移。

778 可以使用的不同偏移数。

```
781 if (! cachep->gfporder && ! (flags & CFLGS_OFF_SLAB))
782 flags |= CFLGS_OPTIMIZE;
783
784 cachep->flags = flags;
785 cachep->gfpflags = 0;
786 if (flags & SLAB_CACHE_DMA)
787 cachep->gfpflags |= GFP_DMA;
788 spin_lock_init(&cachep->spinlock);
789 cachep->objsize = size;
790 INIT_LIST_HEAD(&cachep->slabs_full);
791 INIT_LIST_HEAD(&cachep->slabs_partial);
792 INIT_LIST_HEAD(&cachep->slabs_free);
793
794 if (flags & CFLGS_OFF_SLAB)
795 cachep->slabp_cache =
    kmem_find_general_cachep(slab_size,0);
796 cachep->ctor = ctor;
797 cachep->dtor = dtor;
799 strcpy(cachep->name, name);
```

```

800
801 # ifdef CONFIG_SMP
802 if (g_cputcache_up)
803 enable_cputcache(cachep);
804 # endif

```

这一块初始化高速缓存的其他字段。

781~782 对只有一个页面的 slab 高速缓存, 设置 CFLAGS_OPTIMIZE 标志位, 这实际上没有什么效果, 因为并没有用到该标志位。

784 设置高速缓存静态标志位。

785 将 gfpflags 清 0。这是个无效操作, 因为可以使用 memset() 在分配了高速缓存描述符后清除这些标志位。

786~787 如果 slab 用于 DMA, 则这里设置 GFP_DMA 标志位, 这样伙伴分配器将使用 ZONE_DMA。

788 为访问高速缓存初始化自旋锁。

789 复制对象大小, 如果需要, 则按硬件高速缓存大小对齐。

790~792 初始化 slab 链表。

794~795 如果描述符是 off-slab 的, 这里分配和放置一个 slab 管理器以在 slabp_cache (见 H.2.1.2) 使用。

796~797 设置指向构造函数和销毁函数的指针。

799 复制可读名。

802~803 如果 per-CPU 可用, 这里为该高速缓存创建一个集(见 8.5 节)。

```

806 down(&cache_chain_sem);
807 {
808 struct list_head * p;
809
810 list_for_each(p, &cache_chain) {
811 kmem_cache_t * pc = list_entry(p,
812 kmem_cache_t, next);
813
814 if (! strcmp(pc->name, name))
815 BUG();
816 }
817 }
818
822 list_add(&cachep->next, &cache_chain);
823 up(&cache_chain_sem);

```



```

824 opps;
825 return cachep;
826 }

```

这一块将新的高速缓存加入到高速缓存链表中。

806 获取对高速缓存链表同步访问的信号量。

810~816 检查在高速缓存链表中的每个高速缓存，并保证没有其他的高速缓存具有相同的名字。如果有相同的名字，则意味着创建了具有相同类型的两个高速缓存，这是一个严重的 bug。

811 从链表中获取高速缓存。

814~815 比较名字，如果相同则调用 BUG()。值的注意的是并不删除新的高速缓存，这个错误是开发时由于不规范编程而造成的，是一个常见的问题。

822 将高速缓存链到链表中。

823 释放高速缓存链表信号量。

825 返回新的高速缓存指针。

H.1.2 计算 slab 上的对象数量

H.1.2.1 函数：kmem_cache_estimate() (mm/slab.c)

在创建高速缓存时，可以存放多少个对象以及需要耗费多少空间是确定的。下面的函数计算有多少个对象可以存储，并且考虑到 slab 和 bufctls 必须以 on-slab 方式存放。

```

388 static void kmem_cache_estimate (unsigned long gfporder,
389 size_t size,
390 int flags, size_t * left_over, unsigned int * num)
391 {
392 int i;
393 size_t wastage = PAGE_SIZE<<gfporder;
394 size_t extra = 0;
395 size_t base = 0;
396
397 if (! (flags & CFLGS_OFF_SLAB)) {
398 base = sizeof(slab_t);
399 extra = sizeof(kmem_bufctl_t);
400 }
401 while (i * size + L1_CACHE_ALIGN(base + i * extra) <= wastage)

```

```

402 i++;
403 if (i > 0)
404 i--;
405
406 if (i > SLAB_LIMIT)
407 i = SLAB_LIMIT;
408
409 *num = i;
410 wastage -= i * size;
411 wastage -= L1_CACHE_ALIGN(base + i * extra);
412 *left_over = wastage;
413 }

```

388 这个函数的参数如下：

- `gforder` 为每个 slab 分配 $2^{gforder}$ 个页面。
- `size` 是每个对象的大小。
- `flags` 是高速缓存标志位。
- `left_over` 是 slab 中剩余的字节数, 由调用者返回。
- `num` 是填入 slab 的对象数, 由调用者返回。

392 `wastage` 是一个递减函数。它从可能的最大消耗量开始。

393 `extra` 是需要存储 `kmem_bufctl_t` 的字节数。

394 `base` 是在 slab 开始处的可用内存地址。

396 如果 slab 描述符保存在高速缓存中, 基地址开始于 `slab_t` 结构的末端, 存放 `bufctl` 所需的字节数是 `kmem_bufctl_t` 的大小。

400 `i` 变成 slab 可以拥有的对象数。

401~402 将高速缓存可以容纳的对象数加起来。`i * size` 是对象自身的大小。`L1_CACHE_ALIGN(base + i * extra)` 有一点灵活性。这是一个计算需要存储在 slab 中每个对象所需 `kmem_bufctl_t` 的内存量函数。由于它在 slab 的起点, 所以它与 L1 高速缓存对齐, 这样 slab 中的第 1 个对象将与硬件高速缓存对齐。`i * extra` 将计算为容纳该对象 `kmem_bufctl_t` 所需的空间。由于消耗量以 slab 的大小计算, 所以在这里用于负载。

403~404 由于前面循环计数直到 slab 溢出, 所以对象的数量记为 `i-1`。

406~407 `SLAB_LIMIT` 是一个 slab 可存储对象的最大绝对值。它定义为 `0xffffFFF`, 因为这是 `kmem_bufctl_t()` 可以容纳的最大无符号整数中最大的数值。

409 `num` 现在是 slab 可以容纳的对象数。

410 减去消耗对象所占据的空间。

411 减去由 `kmem_bufctl_t` 所占据的空间。

412 现在消耗量计算为 slab 中剩下的空间。

H.1.3 收缩高速缓存

`kmem_cache_shrink()` 的调用图如图 8.5 所示。提供两套收缩函数。`kmem_cache_shrink()` 从 `slabs_free` 中移除所有 slab，并返回释放的页面数作为结果。`__kmem_cache_shrink()` 从 `slabs_free` 中释放所有的 slab，然后验证 `slabs_partial` 和 `slabs_free` 是否为空。这在销毁高速缓存时比较重要，在那时它不关心释放的页面数，而只关心高速缓存是否为空。

H.1.3.1 函数：`kmem_cache_shrink()` (`mm/slab.c`)

这个函数进行基本的调试检查，然后获取高速缓存描述符，接着释放 slab。在这时，它也用于调用 `drain_cpu_caches()` 来释放放在 per-CPU 高速缓存中的对象。很奇怪的是，由于对象可以在 per-CPU 高速缓存中分配，slab 可能不能被释放，这里却移除，并不使用。

```

966 int kmem_cache_shrink(kmem_cache_t *cachep)
967 {
968     int ret;
969
970     if (!cachep || in_interrupt() ||
971         !is_chained_kmem_cache(cachep))
971     BUG();
972
973     spin_lock_irq(&cachep->spinlock);
974     ret = __kmem_cache_shrink_locked(cachep);
975     spin_unlock_irq(&cachep->spinlock);
976
977     return ret << cachep->gfporder;
978 }
```

966 参数是被收缩的高速缓存。

970 进行如下检查：

- 高速缓存不为 NULL。
- 调用者不是一个异常。
- 高速缓存在高速缓存链表中，且不是一个坏指针。

973 获取高速缓存描述符锁并关中断。

974 收缩高速缓存区。

975 释放高速缓存锁并开中断。

976 返回释放的页面数，但是并没有考虑消耗 CPU 时释放的对象。

H.1.3.2 函数: `__kmem_cache_shrink()` (mm/slab.c)

这个函数与 `kmem_cache_shrink()` 相似,除了它在高速缓存为空时返回。这在销毁高速缓存时比较重要,在那时,释放的内存量并不很重要,重要的是安全地删除高速缓存,且不泄漏内存。

```

945 static int __kmem_cache_shrink(kmem_cache_t *cachep)
946 {
947     int ret;
948
949     drain_cpu_caches(cachep);
950
951     spin_lock_irq(&cachep->spinlock);
952     __kmem_cache_shrink_locked(cachep);
953     ret = !list_empty(&cachep->slabs_full) ||
954     !list_empty(&cachep->slabs_partial);
955     spin_unlock_irq(&cachep->spinlock);
956     return ret;
957 }
```

949 从 per-CPU 对象高速缓存中移除所有对象。

951 获取高速缓存描述符锁并关中断。

952 释放 `slabs_free` 链表中的所有 slab。

953~954 检查 `slabs_partial` 和 `slabs_full` 链表是否为空。

955 释放高速缓存描述符锁并重新打开中断。

956 如果释放了高速缓存中所有 slab,则返回。

H.1.3.3 函数: `_kmem_cache_shrink_locked()` (mm/slab.c)

这里完成释放 slab 的实际工作。它将不断地销毁 slab 直到设置了增长标志位,表明高速缓存正在使用或者直到 `slabs_free` 中没有更多的 slab。

```

917 static int __kmem_cache_shrink_locked(kmem_cache_t *cachep)
918 {
919     slab_t *slabp;
920     int ret = 0;
921
923     while (!cachep->growing) {
924         struct list_head *p;
925
926         p = cachep->slabs_free.prev;
```

```

927 if (p == &cachep->slabs_free)
928 break;
929
930 slabp = list_entry(cachep->slabs_free.prev,
931                      slab_t, list);
932 #if DEBUG
933 if (slabp->inuse)
934     BUG();
935 #endif
936 list_del(&slabp->list);
937
938 kmem_slab_destroy(cachep, slabp);
939 ret++;
940 spin_lock_irq(&cachep->spinlock);
941 }
942 return ret;
943 }

```

923 当高速缓存不再增长时,这里释放 slab。

926~930 获取 slabs_free 链表中最后一个 slab。

932~933 如果调试可用,则这里保证它目前不在使用中。如果不可用,则它首先就不会在 slabs_free 链表中。

935 从链表中移除 slab。

937 重新打开中断。调用这个函数时关闭了中断,所以需要尽快地释放中断。

938 利用 kmem_slab_destroy()(见 H. 2. 3. 1)删除 slab。

939 记录释放的 slab 数。

940 获取高速缓存描述符锁并关中断。

H. 1. 4 销毁高速缓存

当卸载某个模块时,如果创建了高速缓存,则它负责销毁高速缓存。由于在载入一个模块时已经保证没有两个相同名字的高速缓存。内核代码经常不销毁它自己的高速缓存,因为它们在整个系统生命周期中都一直存在。销毁一个高速缓存的步骤如下:

- 从高速缓存链表中删除该高速缓存。
- 收缩高速缓存,删除所有的 slab(见 8. 1. 8 小节)。
- 释放所有的 per-CPU 高速缓存(kfree())。

- 从 cache_cache 中删除高速缓存描述符(见 8.3.3 小节)。

H.1.4.1 函数: kmem_cache_destroy() (mm/slab.c)

这个函数的调用图如图 8.7 所示。

```

997 int kmem_cache_destroy (kmem_cache_t * cachep)
998 {
999 if (! cachep || in_interrupt() || cachep->growing)
1000 BUG();
1001
1002 /* Find the cache in the chain of caches. */
1003 down(&cache_chain_sem);
1004 /* the chain is never empty, cache_cache is never destroyed */
1005 if (clock_searchp == cachep)
1006 clock_searchp = list_entry(cachep->next.next,
1007 kmem_cache_t, next);
1008 list_del(&cachep->next);
1009 up(&cache_chain_sem);
1010
1011 if (_kmem_cache_shrink(cachep)) {
1012 printk(KERN_ERR
1013         "kmem_cache_destroy: Can't free all objects %p\n",
1014 cachep);
1015 down(&cache_chain_sem);
1016 list_add(&cachep->next,&cache_chain);
1017 up(&cache_chain_sem);
1018 return 1;
1019 #ifdef CONFIG_SMP
1020 {
1021 int i;
1022 for (i = 0; i < NR_CPUS; i++)
1023 kfree(cachep->cpudata[i]);
1024 }
1025 #endif
1026 kmem_cache_free(&cache_cache, cachep);
1027
1028 return 0;
1029 }
```

999~1000 有效性检查。保证 cachep 不为空, 没有中断正尝试使用这个函数, 以及高速缓存没有标记为增长, 表明它正在使用中。

1003 获取访问高速缓存链表的信号量。

1005~1007 从高速缓存链表中获取一个链表项。

1008 从高速缓存链表中删除该高速缓存。

1009 释放高速缓存链表信号量。

1011 利用 `_kmem_cache_shrink()` 收缩高速缓存, 释放所有的 slab(见 H. 1.3.2)。

1012~1017 如果在高速缓存中还存在 slab, 则这个收缩函数返回真。如果它们是无法销毁的高速缓存, 则这里将其加入到高速缓存链表中, 并报告错误。

1022~1023 如果 SMP 可用, 则利用 `kfree()`(见 H. 4.3.1)删除 per-CPU 数据结构。

1026 利用 `kmem_cache_free()`(见 H. 3.3.1)从 `cache_cache` 中删除高速缓存描述符。

H. 1.5 回收高速缓存

H. 1.5.1 函数: `kmem_cache_reap()` (`mm/slab.c`)

这个函数的调用图如图 8.4 所示。由于这个函数比较大, 所以将其分成几个独立的小节。第 1 部分是一个简单的函数开始部分, 第 2 部分选择要回收的高速缓冲, 第 3 部分是释放 slab, 基本的任务在 8.1.7 小节中描述。

```

1738 int kmem_cache_reap (int gfp_mask)
1739 {
1740 slab_t * slabp;
1741 kmem_cache_t * searchp;
1742 kmem_cache_t * best_cachep;
1743 unsigned int best_pages;
1744 unsigned int best_len;
1745 unsigned int scan;
1746 int ret = 0;
1747
1748 if (gfp_mask & __GFP_WAIT)
1749 down(&cache_chain_sem);
1750 else
1751 if (down_trylock(&cache_chain_sem))
1752 return 0;
1753
1754 scan = REAP_SCANLEN;

```

```

1755 best_len = 0;
1756 best_pages = 0;
1757 best_cachep = NULL;
1758 searchp = clock_searchp;

```

1738 惟一的一个参数是 GFP 标志位。惟一做的检查是对 __GFP_WAIT 标志位进行检查。作为惟一的调用者,kswapd 可以睡眠。这个参数实际上没有用。

1748~1749 调用者可以睡眠吗? 如果可以, 则在这里获取信号量。

1751~1752 如果不可以睡眠, 则这里试着获取信号量。如果没有信号量可用, 则返回。

1754 REAP_SCANLEN(10)是要检查的高速缓存数目。

1758 设置 searchp 为上次回收时检查过的最后一个高速缓存。

```

1759 do {
1760     unsigned int pages;
1761     struct list_head * p;
1762     unsigned int full_free;
1763
1764     if (searchp->flags & SLAB_NO_REAP)
1765         goto next;
1766     spin_lock_irq(&searchp->spinlock);
1767     if (searchp->growing)
1768         goto next_unlock;
1769     if (searchp->dflags & DFLGS_GROWN) {
1770         searchp->dflags &= ~DFLGS_GROWN;
1771         goto next_unlock;
1772     }
1773
1774 #ifdef CONFIG_SMP
1775 {
1776     cpucache_t * cc = cc_data(searchp);
1777     if (cc && cc->avail) {
1778         __free_block(searchp, cc_entry(cc),
1779                      cc->avail);
1780     }
1781 }
1782 #endif
1783
1784 full_free = 0;
1785 p = searchp->slabs_free.next;
1786 while (p != &searchp->slabs_free) {

```

```

1787 slabp = list_entry(p, slab_t, list);
1788 #if DEBUG
1789 if (slabp->inuse)
1790 BUG();
1791 #endif
1792 full_free++;
1793 p = p->next;
1794 }
1795
1801 pages = full_free * (1<<searchp->gfporder);
1802 if (searchp->ctor)
1803 pages = (pages * 4 + 1)/5;
1804 if (searchp->gfporder)
1805 pages = (pages * 4 + 1)/5;
1806 if (pages > best_pages) {
1807 best_cachep = searchp;
1808 best_len = full_free;
1809 best_pages = pages;
1810 if (pages >= REAP_PERFECT) {
1811 clock_searchp =
    list_entry(searchp->next.next,
1812 kmem_cache_t, next);
1813 goto perfect;
1814 }
1815 }
1816 next_unlock;
1817 spin_unlock_irq(&searchp->spinlock);
1818 next:
1819 searchp =
    list_entry(searchp->next.next, kmem_cache_t, next);
1820 } while (--scan && searchp != clock_searchp);

```

这一块考察 REAP_SCANLEN 数量个高速缓存,并选择一个来释放。

1767 获取一个对高速缓存描述符的中断安全锁。

1768~1769 如果高速缓存在增长,这里跳过它。

1770~1773 如果高速缓存最近增长过,则这里跳过它并清除标志位。

1775~1781 释放 per-CPU 对象到全局池中。

1786~1794 计算在 slab_free 链表中的 slab 数量。

1801 计算所有 slab 持有的页面数量。

1802~1803 如果对象有构造函数,则这里将其页面计数减为 1/5,这样就不太可能选择它来回收。

1804~1805 如果 slab 由多于一页组成,这里减少页面计数到 1/5。这是因为难以获取高次的页面。

1806 如果这是目前找到的最好的回收候选高速缓存,这里检查它是否适合回收。

1807~1809 记录新的最大量。

1808 记录 best_len,这样很容易知道空闲链表中 slab 一半的 slab 数量。

1810 如果这个高速缓存适合回收,则……

1811 更新 clock_searchp。

1812 转到 perfect 在那里释放一半的 slab。

1816 如果发现在获取锁后高速缓存正在增长,则到达这个标号。

1817 释放高速缓存描述符锁。

1818 移到下一个高速缓存链表项。

1820 在达到 REAP_SCANLEN 之前,且在还没有循环遍历整个高速缓存链表之前一直扫描。

```

1822 clock_searchp = searchp;
1823
1824 if (!best_cachep)
1825     goto out;
1826
1827
1828 spin_lock_irq(&best_cachep->spinlock);
1829 perfect:
1830 /* free only 50 % of the free slabs */
1831 best_len = (best_len + 1)/2;
1832 for (scan = 0; scan < best_len; scan++) {
1833     struct list_head *p;
1834
1835     if (best_cachep->growing)
1836         break;
1837     p = best_cachep->slabs_free.prev;
1838     if (p == &best_cachep->slabs_free)
1839         break;
1840     slabp = list_entry(p, slab_t, list);
1841     #if DEBUG
1842     if (slabp->inuse)
1843         BUG();
1844     #endif

```

```

1845 list_del(&slabp->list);
1846 STATS_INC_REAPED(best_cachep);
1847
1848 /* Safe to drop the lock. The slab is no longer
1849 * lined to the cache.
1850 */
1851 spin_unlock_irq(&best_cachep->spinlock);
1852 kmem_slab_destroy(best_cachep, slabp);
1853 spin_lock_irq(&best_cachep->spinlock);
1854 }
1855 spin_unlock_irq(&best_cachep->spinlock);
1856 ret = scan * (1 << best_cachep->gfporder);
1857 out:
1858 up(&cache_chain_sem);
1859 return ret;
1860 }

```

这一块从选择的高速缓存中释放一半的 slab。

1822 更新 clock_searchp 以准备下一次高速缓存回收。

1824~1826 如果没有找到一个高速缓存, 则转到 out 以释放高速缓存链表并退出。

1828 获取高速缓存链表自旋锁, 并关闭中断。cachep 描述符必须持有一个中断安全的锁, 因为一些高速缓存可能在中断上下文中使用。slab 分配器无法区分中断安全和中断不安全的高速缓存。

1831 调整 best_len 为要释放的 slab 数量。

1832~1854 释放 best_len 个 slab。

1835~1847 如果高速缓存正在增长, 则在这里退出。

1837 从链表中获得一个 slab。

1838~1839 如果在链表中没有一个 slab, 则在这里退出。

1840 获取 slab 指针。

1842~1843 如果调试可用, 则这里保证没有活动的对象在 slab 中。

1845 从 slabs_free 链表中移除 slab。

1846 如果可用则更新统计计数。

1851 释放高速缓存并开中断。

1852 释放 slab(见 8.2.8 小节)。

1851 重新获取高速缓存描述符的自旋锁并关中断。

1855 释放高速缓存描述符并开中断。

1856 ret 是已经释放的页面数。

1858~1859 释放高速缓存信号量并返回释放的页面数。

H. 2 Slabs

H. 2. 1 存储 Slab 描述符

H. 2. 1. 1 函数: `kmem_cache_slabmgmt()` (`mm/slab.c`)

这个函数或者为不在高速缓存中的 slab 描述符分配空间,或者在 slab 开始的地方为描述符和 bufctls 保留足够的空间。

```

1032 static inline slab_t * kmem_cache_slabmgmt (
    kmem_cache_t * cachep,
1033 void * objp,
    int colour_off,
    int local_flags)
1034 {
1035 slab_t * slabp;
1036
1037 if (OFF_SLAB(cachep)) {
1039 slabp = kmem_cache_alloc(cachep->slabp_cache,
    local_flags);
1040 if (! slabp)
1041 return NULL;
1042 } else {
1047 slabp = objp + colour_off;
1048 colour_off += L1_CACHE_ALIGN(cachep->num *
1049 sizeof(kmem_bufctl_t) +
sizeof(slab_t));
1050 }
1051 slabp->inuse = 0;
1052 slabp->colouroff = colour_off;
1053 slabp->s_mem = objp + colour_off;
1054
1055 return slabp;
1056 }
```

1032 这个函数的参数如下:

- cachep 是 slab 要分配的高速缓存。
- objp 是当调用该函数时指向 slab 的开始处。
- colour_off 是这个 slab 的颜色偏移。
- local_flags 是高速缓存的标志位。

1037~1042 如果 slab 描述符并不存放在高速缓存中, 则……

1039 从指定大小的高速缓存中分配内存。在创建高速缓存时, slabp_cache 设置为分配内存指定大小的高速缓存。

1040 如果分配失败, 这里返回。

1042~1050 在 slab 的起点保留空间。

1047 slab 的地址将是 slab 的起点(objp)加上颜色偏移。

1048 colour_off 计算为放置第一个对象的偏移位置。该地址对齐于 L1 高速缓存, cachep->num * sizeof(kmem_bufctl_t) 是容纳 slab 中各对象的 bufctls 所需的空间大小, sizeof(slab_t) 是 slab 描述符的大小。这里在 slab 起点已经有效地保留了空间。

1051 在 slab 中使用的对象数为 0。

1052 更新 colouroff 以放置新对象。

1053 第 1 个对象的地址计算为 slab 的开始地址加上偏移。

H.2.1.2 函数:kmem_find_general_cachep() (mm/slab.c)

如果 slab 描述符不在 slab 中保存, 则这个函数在创建高速缓存时被调用, 它将找到合适大小的高速缓存供使用, 并将其存放在 slabp_cache 的高速缓存描述符中。

```

1620 kmem_cache_t * kmem_find_general_cachep (size_t size,
1621     int gfpflags)
1621 {
1622     cache_sizes_t * csizep = cache_sizes;
1623
1624     for ( ; csizep->cs_size; csizep++) {
1625         if (size > csizep->cs_size)
1626             continue;
1627         break;
1628     }
1629     return (gfpflags & GFP_DMA) ? csizep->cs_dmacachep :
1630         csizep->cs_cachep;
1631 }
1632 }
```

1620 size 是 slab 描述符的大小, gfpflags 总是为 0, 因为 DMA 内存不需要 slab 描述符。

1628~1632 从最小的大小开始, 这里不断增加大小, 直至找到一个足够大的缓冲区来存

放 slab 描述符的高速缓存。

1633 返回一个普通的或 DMA 大小的高速缓存,这取决于传入的 gfpflags 标志位。实际上,仅传回 cs_cachep。

H.2.2 创建 slab

H.2.2.1 函数: kmem_cache_grow() (mm/slab.c)

这个函数的调用图如图 8.11 所示。这个函数的基本步骤如下:

- 进行基本的有效性检查以防止错误使用。
- 计算该 slab 中对象的颜色偏移。
- 为 slab 分配内存,并获取 slab 描述符。
- 将 slab 使用的页面链接到 slab 和高速缓存描述符。
- 初始化 slab 中的对象。
- 将 slab 加入高速缓存。

```
1105 static int kmem_cache_grow (kmem_cache_t * cachep, int flags)
1106 {
1107 slab_t * slabp;
1108 struct page * page;
1109 void * objp;
1110 size_t offset;
1111 unsigned int i, local_flags;
1112 unsigned long ctor_flags;
1113 unsigned long save_flags;
```

这是一些基本的声明。这个函数的参数如下:

- cachep 是要分配新 slab 的高速缓存。
- flags 创建 slab 的标志位。

```
1118 if (flags & ~(SLAB_DMA|SLAB_LEVEL_MASK|SLAB_NO_GROW))
1119 BUG();
1120 if (flags & SLAB_NO_GROW)
1121 return 0;
1122
1123 if (in_interrupt() &&
     (flags & SLAB_LEVEL_MASK) != SLAB_ATOMIC)
1124 BUG();
1125
```



```

1132 ctor_flags = SLABCTOR_CONSTRUCTOR;
1133 local_flags = (flags & SLAB_LEVEL_MASK);
1134 if (local_flags == SLAB_ATOMIC)
1139 ctor_flags |= SLABCTOR_ATOMIC;

```

这里进行基本的有效性检查以防止错误使用。在这里进行检查,而不是利用 kmem_cache_alloc()来保护速度优先的路径。这里不需要在每次分配对象时都检查这些标志位。

1118~1119 保证仅使用允许的标志位来进行分配。

1120~1121 如果设置了这些标志位,则不增长高速缓存,实际上,从来不设置这些标志位。

1129~1130 如果在中断上下文中调用,保证设置了 ATOMIC 标志位。这样我们在调用 kmem_getpages()(见 H. 7.1.1)时不会睡眠。

1132 这个标志位告知构造函数初始化对象。

1133 local_flags 仅与页面分配器相关。

1134~1139 如果设置了 SLAB_ATOMIC 标志位,则构造函数在创建一次新分配时就需要知道它。

```

1142 spin_lock_irqsave(&cachep->spinlock, save_flags);
1143
1145 offset = cachep->colour_next;
1146 cachep->colour_next++;
1147 if (cachep->colour_next >= cachep->colour)
1148 cachep->colour_next = 0;
1149 offset *= cachep->colour_off;
1150 cachep->dflags |= DFLGS_GROWN;
1151
1152 cachep->growing++;
1153 spin_unlock_irqrestore(&cachep->spinlock, save_flags);

```

计算该 slab 中对象的颜色偏移。

1142 获得访问高速缓存描述符的中断安全锁。

1145 获得该 slab 中的对象偏移。

1146 移到下一个颜色偏移。

1147~1148 如果达到 colour,就没有更多的偏移,这里将 colour_next 重置为 0。

1149 colour_off 是每个偏移的大小。所以 offset * colour_off 表明对象偏移的字节数。

1150 将高速缓存标记为增长,这样 kmem_cache_reap()(见 H. 1.5.1)将忽略该高速缓存。

1152 增加增长该高速缓存的调用者计数。

1153 释放自旋锁并重新打开中断。

```
1165 if (! (objp = kmem_getpages(cachep, flags)))
1166     goto failed;
1167
1169 if (! (slabp = kmem_cache_slabmgmt(cachep,
1170           objp, offset,
1171           local_flags)))
1160     goto opps1;
```

为 slab 分配内存并获取一个 slab 描述符。

1165~1166 利用 kmem_getpages() (见 H. 7.1.1) 从页面分配器中为 slab 分配页面。

1169 利用 kmem_cache_slabmgmt() (见 H. 2.1.1) 获取一个 slab 描述符。

```
1173 i = 1 << cachep->gfporder;
1174 page = virt_to_page(objp);
1175 do {
1176     SET_PAGE_CACHE(page, cachep);
1177     SET_PAGE_SIZE(page, slabp);
1178     PageSetSlab(page);
1179     page++;
1180 } while (--i);
```

将 slab 使用的页面链接到 slab 和高速缓存描述符。

1173 *i* 是 slab 使用的页面数。每个页面都必须链接到 slab 和高速缓存描述符。

1174 *objp* 是一个指向 slab 起点的指针。宏 *virt_to_page()* 将赋予 struct page 该地址值。

1175~1180 将每个页面链接到 slab 字段以及高速缓存描述符。

1176 *SET_PAGE_CACHE()* 使用 *page->link_next* 字段将页面链接到高速缓存描述符。

1177 *SET_PAGE_SIZE()* 使用 *page->link_prev* 字段将页面链接到高速缓存描述符。

1178 设置 PG_slab 页面字段。整个 PG_flags 在表 2.1 中列出。

1179 移到 slab 中下一个页面进行链接。

```
1182 kmem_cache_init_objs(cachep, slabp, ctor_flags);
```

1182 初始化对象 (见 H. 3.1.1)。

```
1184 spin_lock_irqsave(&cachep->spinlock, save_flags);
```

```
1185 cachep->growing--;
```

```
1186
```

```
1188 list_add_tail(&slabp->list, &cachep->slabs_free);
```

```

1189 STATS_INC_GROWN(cachep);
1190 cachep->failures = 0;
1191
1192 spin_unlock_irqrestore(&cachep->spinlock, save_flags);
1193 return 1;

```

将 slab 加入到高速缓存中。

1184 以一种中断安全的方式获取高速缓存描述符自旋锁。

1185 将增长计数减 1。

1188 将 slab 加入到 slabs_free 链表的末尾。

1189 如果设置了 STATS, 这里增加 cachep->grown 字段 STATS_INC_GROWN()。

1190 设置失败为 0, 这个字段在其他地方没有用到。

1192 以一种中断安全的方式释放高速缓存描述符自旋锁。

1193 返回成功。

```

1194 oops1:
1195 kmem_freepages(cachep, objp);
1196 failed:
1197 spin_lock_irqsave(&cachep->spinlock, save_flags);
1198 cachep->growing--;
1199 spin_unlock_irqrestore(&cachep->spinlock, save_flags);
1300 return 0;
1301 }

```

这一块进行错误处理:

1194~1195 如果为 slab 分配了页面, 则转到 oops1。必须在这里释放这些页面。

1197 为访问高速缓存描述符获取自旋锁。

1198 将增长计数减 1。

1199 释放自旋锁。

1300 返回假。

H. 2.3 销毁 Slab

H. 2.3.1 函数: kmem_slab_destroy() (mm/slab.c)

这个函数的调用图如图 8.13 所示。为了便于阅读, 调试部分已经从这个函数中略去, 它们几乎与分配对象时的调试部分相同。如何标记以及检查感染模式见 H. 3.1.1。

```

555 static void kmem_slab_destroy (kmem_cache_t * cachep, slab_t * slabp)
556 {
557 if (cachep->dtor
561 ) {
562 int i;
563 for (i = 0; i < cachep->num; i++) {
564 void * objp = slabp->s_mem + cachep->objsize * i;
565-574 DEBUG: Check red zone markers
575 if (cachep->dtor)
576 (cachep->dtor)(objp, cachep, 0);
577-584 DEBUG: Check poison pattern
585 }
586 }
587
588 kmem_freepages(cachep, slabp->s_mem-slabp->colouroff);
589 if (OFF_SLAB(cachep))
590 kmem_cache_free(cachep->slabp_cache, slabp);
591 }

```

557~586 如果有销毁函数可用,这里将对 slab 中每个对象调用它。

563~585 遍历 slab 中每个对象。

564 计算要销毁对象的地址。

575~576 调用销毁函数。

588 释放 slab 中使用的页面。

589 如果 slab 描述符不在 slab 中,则这里使用它所用到的内存。

H. 3 对象

这一节讨论如何管理对象。在这个地方,大部分的工作已经由它们的高速缓存管理器或者 slab 管理器完成。

H. 3. 1 初始化 Slab 中的对象

H. 3. 1. 1 函数: kmem_cache_init_objs() (mm/slab.c)

这个函数大部分都与调试有关,所以从没有调试的部分开始,并在处理调试部分之前先进

行详细的解释。在代码中标记的调试部分分别标记为 Part1 和 Part2。

```

1058 static inline void kmem_cache_init_objs (kmem_cache_t * cachep,
1059 slab_t * slabp, unsigned long ctor_flags)
1060 {
1061 int i;
1062
1063 for (i = 0; i < cachep->num; i++) {
1064 void * objp = slabp->s_mem + cachep->objsize * i;
1065-1072 /* Debugging Part 1 */
1079 if (cachep->ctor)
1080 cachep->ctor(objp, cachep, ctor_flags);
1081-1094 /* Debugging Part 2 */
1095 slab_bufctl(slabp)[i] = i + 1;
1096 }
1097 slab_bufctl(slabp)[i - 1] = BUFCTL_END;
1098 slabp->free = 0;
1099 }

```

这个函数的参数如下：

- cachep 是对象初始化的高速缓存。
- slabp 是存放对象的 slab。
- ctor_flags 是构造函数需要的标志, 以标识这是否是一个原子性的分配。

1063 初始化 cache->num 个对象。

1064 slab 中的基地址是 s_mem。待分配的对象地址是 $i *$ (单个对象的大小)。

1079~1090 如果构造函数可用, 则调用它。

1095 宏 slab_bufctl() 将 slabp 转换为 slab_t slab 描述符, 并向其中加入一个。这里将一个指针指向 slab 描述符的末尾, 然后将其转换为 kmem_bufctl_t, 并有效地给出 bufctl 数组的起始。

1098 在 bufctl 数组中第 1 个空闲对象的下标是 0。

这里讨论初始化对象的核心。下一步, 将讨论调试的第 1 部分。

```

1065 #if DEBUG
1066 if (cachep->flags & SLAB_RED_ZONE) {
1067 * ((unsigned long *) (objp)) = RED_MAGIC1;
1068 * ((unsigned long *) (objp + cachep->objsize -
1069 BYTES_PER_WORD)) = RED_MAGIC1;
1070 objp += BYTES_PER_WORD;
1071 }

```

1072 #endif

1066 如果高速缓存是红色分区的,这里放置一个标记在对象的任一个末端。

1067 放置一个标记在对象的起点。

1068 放置一个标记在对象的末端。记住对象的大小的同时考虑了在红色分区可用时的红色标记大小。

1070 为了方便在这一调试部分后调用构造函数,这里将 objp 指针加上红色标记的大小。

```
1081 #if DEBUG
1082 if (cachep->flags & SLAB_RED_ZONE)
1083 objp -= BYTES_PER_WORD;
1084 if (cachep->flags & SLAB_POISON)
1086 kmem_poison_obj(cachep, objp);
1087 if (cachep->flags & SLAB_RED_ZONE) {
1088 if (*((unsigned long *) (objp)) != RED_MAGIC1)
1089 BUG();
1090 if (*((unsigned long *) (objp + cachep->objsize -
1091 BYTES_PER_WORD)) != RED_MAGIC1)
1092 BUG();
1093 }
1094 #endif
```

这是在构造函数存在并被调用发生后的调试块。

1082~1083 objp 指针以先前调试块的红色标记的大小向后移,所以这里将其移回去。

1084~1086 如果没有构造函数,这里以一种已知模式来感染对象,这样可以在后面跟踪未初始化写时发现这个对象。

1088 检查以保证在对象开始部分的红色标志保留在对象之前的跟踪写。

1090~1091 检查以保证跟踪写没有在超过对象末端的地址发生。

H.3.2 分配对象

H.3.2.1 函数: kmem_cache_alloc() (mm/slab.c)

这个函数的调用图如图 8.14 所示。这个小函数仅调用 __kmem_cache_alloc()。

```
1529 void * kmem_cache_alloc (kmem_cache_t * cachep, int flags)
1531 {
1532 return __kmem_cache_alloc(cachep, flags);
1533 }
```

H.3.2.2 函数: __kmem_cache_alloc(UP Case)() (mm/slab.c)

这里处理 UP 情形。SMP 情形将在下一节处理。

```

1338 static inline void * __kmem_cache_alloc (kmem_cache_t * cachep,
                                             int flags)
1339 {
1340     unsigned long save_flags;
1341     void * objp;
1342
1343     kmem_cache_alloc_head(cachep, flags);
1344     try_again:
1345     local_irq_save(save_flags);
1346     objp = kmem_cache_alloc_one(cachep);
1347     local_irq_restore(save_flags);
1348     return objp;
1349     alloc_new_slab:
1350     local_irq_restore(save_flags);
1351     if (kmem_cache_grow(cachep, flags))
1352         goto try_again;
1353     return NULL;
1354 }
```

1338 参数是要从中分配的高速缓存以及指定的分配标志位。

1343 这个函数保证使用了 DMA 标志位的恰当组合。

1345 关中断并保存标志位。这个函数用于中断,所以这里是在 UP 情形下提供同步的惟一方式。

1367 kmem_cache_alloc_one()(见 H.3.2.5)从链表中分配一个对象并返回这个对象。如果没有释放对象,则这个宏(注意它不是一个函数)将转到该函数末尾处的 alloc_new_slab。

1369~1370 重新开中断并返回。

1376 在这个标记处,slabs_partial 中没有释放对象,slabs_free 也为空,则需要一个新的 slab。

1377 分配一个新的 slab(见 8.2.2 小节)。

1381 因为有一个新的 slab 可用,所以这里再次尝试。

1382 没有分配一个 slab,所以这里返回失败。

H.3.2.3 函数: __kmem_cache_alloc(SMP Case)() (mm/slab.c)

下面就是 SMP 情形下的函数:

```
1338 static inline void * __kmem_cache_alloc (kmem_cache_t * cachep,
```

```

        int flags)
1339 {
1340 unsigned long save_flags;
1341 void * objp;
1342
1343 kmem_cache_alloc_head(cachep, flags);
1344 try_again:
1345 local_irq_save(save_flags);
1347 {
1348 cpucache_t * cc = cc_data(cachep);
1349
1350 if (cc) {
1351 if (cc->avail) {
1352 STATS_INC_ALLOCHIT(cachep);
1353 objp = cc_entry(cc)[--cc->avail];
1354 } else {
1355 STATS_INC_ALLOCMISS(cachep);
1356 objp =
            kmem_cache_alloc_batch(cachep, cc, flags);
1357 if (!objp)
1358 goto alloc_new_slab_nolock;
1359 }
1360 } else {
1361 spin_lock(&cachep->spinlock);
1362 objp = kmem_cache_alloc_one(cachep);
1363 spin_unlock(&cachep->spinlock);
1364 }
1365 }
1366 local_irq_restore(save_flags);
1370 return objp;
1371 alloc_new_slab:
1373 spin_unlock(&cachep->spinlock);
1374 alloc_new_slab_nolock:
1375 local_irq_restore(save_flags);
1377 if (kmem_cache_grow(cachep, flags))
1381 goto try_again;
1382 return NULL;
1383 }

```

1338~1347 与 UP 情形相同。

1349 从这个 CPU 中获取 per-CPU 数据。

1350~1360 如果 per-CPU 可用, 则……

1351 如果一个对象可用, 则……

1352 如果允许则更新高速缓存统计计数。

1353 获取一个对象并更新 avail 数值。

1354 如果不是这样, 即没有对象可用, 则……

1355 如果激活则更新这个高速缓存的统计。

1356 分配 batchcount 个对象, 除最后一个以外将它们都放置到 per-CPU 中, 返回最后一个给 objp。

1357~1358 分配失败, 所以跳转到 alloc_new_slab_nolock 来增长高速缓存并分配一个新的 slab。

1360~1364 如果没有可用的 per-CPU 高速缓存, 则这里释放高速缓存自旋锁并像 UP 情形一样分配一个对象。例如, 在初始化 cache_cache 时就会出现这种情形。

1363 成功指定对象, 所以释放高速缓存自旋锁。

1366~1370 重新开中断, 返回已分配的对象。

1371~1373 如果 kmem_cache_alloc_one() 未能分配一个对象, 则跳转到这里, 这个时候还持有自旋锁, 所以必须在这里释放。

H.3.2.4 函数: kmem_cache_alloc_head() (mm/slab.c)

这个简单函数保证从 slab 中分配时用到的 GFP 标记位和 slab 正确关联。如果该高速缓存用作 DMA, 则这个函数将保证调用者不会突然请求普通内存, 反之亦然。

```
1231 static inline void kmem_cache_alloc_head(kmem_cache_t *cachep,
                                             int flags)
1232 {
1233     if (flags & SLAB_DMA) {
1234         if (! (cachep->gfpflags & GFP_DMA))
1235             BUG();
1236     } else {
1237         if (cachep->gfpflags & GFP_DMA)
1238             BUG();
1239     }
1240 }
```

1231 参数是我们将从中分配的高速缓存, 分配时需要的标志位。

1233 如果调用者请求用于 DMA 的内存, 则……

1234 这个高速缓存没有使用 DMA 内存, 则调用 BUG()。

1237 如果不是这样,如果调用者没有请求 DMA 内存,且该高速缓存用于 DMA,则这里调用 BUG()。

H.3.2.5 函数: kmem_cache_alloc_one() (mm/slab.c)

这是一个预处理宏。奇怪的是在这里没有把它写成一个内联函数,而是写成了一个预处理宏,在__kmem_cache_alloc() (见 H.3.2.2 节)中优化了 goto。

```

1283 #define kmem_cache_alloc_one(cachep) \
1284 ({ \
1285     struct list_head * slabs_partial, * entry; \
1286     slab_t * slabp; \
1287 \
1288     slabs_partial = &(cachep)->slabs_partial; \
1289     entry = slabs_partial->next; \
1290     if (unlikely(entry == slabs_partial)) { \
1291         struct list_head * slabs_free; \
1292         slabs_free = &(cachep)->slabs_free; \
1293         entry = slabs_free->next; \
1294         if (unlikely(entry == slabs_free)) \
1295             goto alloc_new_slab; \
1296         list_del(entry); \
1297         list_add(entry, slabs_partial); \
1298     } \
1299 \
1300     slabp = list_entry(entry, slab_t, list); \
1301     kmem_cache_alloc_one_tail(cachep, slabp); \
1302 })

```

1288~1289 从 slabs_partial 链表中获得第一个 slab。

1290~1298 如果从这个链表中不能获得 slab,从这里开始执行这一块。

1291~1293 从 slabs_free 链表中获得第 1 个 slab。

1294~1295 如果 slabs_free 中没有 slab,则跳转到 alloc_new_slab(),这里跳转的标号在__kmem_cache_alloc()中,在那里把高速缓存增大一个 slab。

1296~1297 如果不是这样,则从空闲链表中移除 slab,并将其放到 slabs_partial 链表中,因为将有一个对象从那里移除。

1300 从链表上获取 slab。

1301 从 slab 中分配一个对象。

H.3.2.6 函数: kmem_cache_alloc_one_tail() (mm/slab.c)

这个函数负责从 slab 中分配一个对象。这里大部分都是调试代码。

```

1242 static inline void * kmem_cache_alloc_one_tail (
    kmem_cache_t * cachep,
1243 slab_t * slabp)
1244 {
1245 void * objp;
1246
1247 STATS_INC_ALLOCED(cachep);
1248 STATS_INC_ACTIVE(cachep);
1249 STATS_SET_HIGH(cachep);
1250
1252 slabp->inuse++;
1253 objp = slabp->s_mem + slabp->free * cachep->objsize;
1254 slabp->free = slab_bufctl(slabp)[slabp->free];
1255
1256 if (unlikely(slabp->free == BUFCTL_END)) {
1257 list_del(&slabp->list);
1258 list_add(&slabp->list, &cachep->slabs_full);
1259 }
1260 #if DEBUG
1261 if (cachep->flags & SLAB_POISON)
1262 if (kmem_check_poison_obj(cachep, objp))
1263 BUG();
1264 if (cachep->flags & SLAB_RED_ZONE) {
1265 if (xchg((unsigned long *)objp, RED_MAGIC2) !=
1266 RED_MAGIC1)
1267 BUG();
1268 if (xchg((unsigned long *) (objp + cachep->objsize -
1269 BYTES_PER_WORD), RED_MAGIC2) != RED_MAGIC1)
1270 BUG();
1271 BUG();
1272 objp += BYTES_PER_WORD;
1273 }
1274 #endif
1275 return objp;
1276 }

```

1242 参数是高速缓存和要从中分配的 slab。

1247~1249 如果 stats 可用，则这里设置三个统计计数。ALLOCED 是已经分配的对象总数。ACTIVE 是高速缓存中活跃对象的总数。HIGH 是某个时间上活跃对象的最大数。

1252 inuse 是该 slab 上的活动对象数。

1253 获取指向一个空闲对象的指针。s_mem 是指向 slab 上第 1 个对象的指针。free 是 slab 中空闲对象的索引。index * 对象大小是 slab 中的偏移。

1254 更新空闲指针为下一个空闲对象的索引。

1256~1259 如果 slab 是满的,则这里从 slabs_partial 链表中将它移除并放到 slab_full 链表中。

1260~1274 调试代码。

1275 没有调试,则返回对象给调用者。

1261~1263 如果对象由已知模式感染,则这里检查它是否是一次未初始化访问。

1266~1267 如果红色分区可用,则这里检查对象开始处的标记并保证它是安全的。在后面改变该红色标记以检查对象后面的写操作。

1269~1271 检查对象末端的标记并在后面改变该红色标记,以检查对象后面的写操作。

1272 更新对象指针以指向红色标记后面的指针。

1275 返回对象。

H. 3.2.7 函数: kmem_cache_alloc_batch() (mm/slab.c)

这个函数分配一批对象给一个对象的 CPU 高速缓存。它仅在 SMP 情形下使用。在许多方面,它和 kmem_cache_alloc_one()(见 H. 3.2.5)类似。

```

1305 void * kmem_cache_alloc_batch(kmem_cache_t * cachep,
1306                                 cpcache_t * cc, int flags)
1307 {
1308     int batchcount = cachep->batchcount;
1309     spin_lock(&cachep->spinlock);
1310     while (batchcount --) {
1311         struct list_head * slabs_partial, * entry;
1312         slab_t * slabp;
1313         /* Get slab alloc is to come from. */
1314         slabs_partial = &(cachep)->slabs_partial;
1315         entry = slabs_partial->next;
1316         if (unlikely(entry == slabs_partial)) {
1317             struct list_head * slabs_free;
1318             slabs_free = &(cachep)->slabs_free;
1319             entry = slabs_free->next;
1320             if (unlikely(entry == slabs_free))
1321                 break;
1322             list_del(entry);
1323             list_add(entry, slabs_partial);
1324     }

```

```

1325
1326 slabp = list_entry(entry, slab_t, list);
1327 cc_entry(cc)[cc->avail ++] =
1328 kmem_cache_alloc_one_tail(cachep, slabp);
1329 }
1330 spin_unlock(&cachep->spinlock);
1331
1332 if (cc->avail)
1333 return cc_entry(cc)[ -- cc->avail];
1334 return NULL;
1335 }

```

1305 参数是要从中分配的高速缓存,要填充的 per-CPU 高速缓存以及分配标志位。

1307 batchcount 是待分配的对象数目。

1309 获取访问高速缓存描述符的自旋锁。

1310~1329 循环 batchcount 次数。

1311~1324 这个例子与 kmem_cache_alloc_one()(见 H. 3. 2. 5)相同,它从 slabs_partial 或 slabs_free 中选择一个 slab 来分配。如果没有,则退出循环。

1326~1327 调用 kmem_cache_alloc_one_tail()(见 H. 3. 2. 6)并将其放入 per-CPU 高速缓存。

1330 释放高速缓存描述符锁。

1332~1333 从这批分配的对象中取出一个并返回它。

1334 如果没有分配对象,这里直接返回。__kmem_cache_alloc()(见 H. 3. 2. 3)将通过一个 slab 增大高速缓存并再次尝试。

H. 3. 3 释放对象

H. 3. 3. 1 函数: kmem_cache_free() (mm/slab.c)

这个函数的调用图如图 8. 15 所示。

```

1576 void kmem_cache_free (kmem_cache_t * cachep, void * objp)
1577 {
1578     unsigned long flags;
1579     #if DEBUG
1580     CHECK_PAGE(virt_to_page(objp));
1581     if (cachep != GET_PAGE_CACHE(virt_to_page(objp)))
1582         BUG();
1583     #endif

```

```

1584
1585 local_irq_save(flags);
1586 __kmem_cache_free(cachep, objp);
1587 local_irq_restore(flags);
1588 }

```

1576 参数是对象释放的高速缓存,以及对象本身。

1579~1583 如果调试可用,则首先利用 CHECK_PAGE() 进行检查以确定它是一个 slab 页面。然后检查一个页面链表以保证它属于这个高速缓存(如图 8.8 所示)。

1585 关闭中断来保护该路径。

1586 SMP 情形下, __kmem_cache_free() (见 H. 3. 3. 2) 释放对象到 per-CPU 高速缓存中。普通情形下,释放到全局池中。

1587 重新打开中断。

H. 3. 3. 2 函数: __kmem_cache_free(UP Case)() (mm/slab.c)

这种情形比较有趣。在这种情形下,如果有对象则释放到 per-CPU 高速缓存。

```

1493 static inline void __kmem_cache_free (kmem_cache_t * cachep,
                                         void * objp)
1494 {
1517 kmem_cache_free_one(cachep, objp);
1519 }

```

H. 3. 3. 3 函数: __kmem_cache_free(SMP Case)() (mm/slab.c)

这种情形比较有趣。在这种情形下,如果有对象则释放到 per-CPU 高速缓存。

```

1493 static inline void __kmem_cache_free (kmem_cache_t * cachep,
                                         void * objp)
1494 {
1496 cpucache_t * cc = cc_data(cachep);
1497
1498 CHECK_PAGE(virt_to_page(objp));
1499 if (cc) {
1500 int batchcount;
1501 if (cc->avail < cc->limit) {
1502 STATS_INC_FREEHIT(cachep);
1503 cc_entry(cc)[cc->avail ++ ] = objp;
1504 return;
1505 }
1506 STATS_INC_FREEMISS(cachep);

```

```

1507 batchcount = cachep->batchcount;
1508 cc->avail -= batchcount;
1509 free_block(cachep,
1510 &cc_entry(cc)[cc->avail],batchcount);
1511 cc_entry(cc)[cc->avail ++ ] = objp;
1512 return;
1513 } else {
1514 free_block(cachep, &objp, 1);
1515 }
1519 }

```

1496 获得这个 per-CPU 高速缓存的数据(见 8.5.1 小节)。

1498 保证该页面是一个 slab 页面。

1499~1513 如果 per-CPU 高速缓存可用,则这里试着使用它。这里并不是一直都可用。例如,在销毁高速缓存的时候,per-CPU 高速缓存已经不存在了。

1501~1505 如果可用的 per-CPU 高速缓存数目在极限值以下,这里将对象加入到空闲链表中并返回。

1506 如果可用,则更新统计计数。

1507 池已经溢出,所以将 batchcount 个对象释放到全局池中。

1508 更新可用(avail)对象的数目。

1509~1510 释放一块对象到全局池。

1511 释放请求的对象并将其放到 per-CPU 池中。

1513 如果 per-CPU 高速缓存不可用,则这里释放对象到全局池中。

H.3.3.4 函数: kmem_cache_free_one() (mm/slab.c)

```

1414 static inline void kmem_cache_free_one(kmem_cache_t * cachep,
                                         void * objp)
1415 {
1416 slab_t * slabp;
1417
1418 CHECK_PAGE(virt_to_page(objp));
1425 slabp = GET_PAGE_SLAB(virt_to_page(objp));
1426
1427 #if DEBUG
1428 if (cachep->flags & SLAB_DEBUG_INITIAL)
1433 cachep->ctor (objp, cachep,
                     SLABCTOR_CONSTRUCTOR|SLABCTOR_VERIFY);
1434
1435 if (cachep->flags & SLAB_RED_ZONE) {

```

```

1436 objp -= BYTES_PER_WORD;
1437 if (xchg((unsigned long *)objp, RED_MAGIC1) !=
    RED_MAGIC2)
1438 BUG();
1440 if (xchg((unsigned long *) (objp + cachep->objsize -
1441 BYTES_PER_WORD), RED_MAGIC1) !=
    RED_MAGIC2)
1443 BUG();
1444 }
1445 if (cachep->flags & SLAB_POISON)
1446 kmem_poison_obj(cachep, objp);
1447 if (kmem_extra_free_checks(cachep, slabp, objp))
1448 return;
1449 #endif
1450 {
1451 unsigned int objnr = (objp - slabp->s_mem)/cachep->objsize;
1452
1453 slab_bufctl(slabp)[objnr] = slabp->free;
1454 slabp->free = objnr;
1455 }
1456 STATS_DEC_ACTIVE(cachep);
1457
1459 {
1460 int inuse = slabp->inuse;
1461 if (unlikely(! -- slabp->inuse)) {
1462 /* Was partial or full, now empty. */
1463 list_del(&slabp->list);
1464 list_add(&slabp->list, &cachep->slabs_free);
1465 } else if (unlikely(inuse == cachep->num)) {
1466 /* Was full. */
1467 list_del(&slabp->list);
1468 list_add(&slabp->list, &cachep->slabs_part
1469 )
1470 }
1471 }

```

1418 保证页面是一个 slab 页面。

1425 获取页面的 slab 描述符。

1427~1449 调试部分。在这一节的末尾讨论。

1451 计算被释放对象的下标。

1454 由于这个对象已经被释放,所以更新 bufctl 来反映这一点。

1456 如果统计计数可用,则这里不激活在 slab 中的活跃对象。

1461~1464 如果 inuse 到达 0,则释放 slab 并将其移到 slabs_free 链表中。

1465~1468 如果使用的数量等于 slab 中的对象数量,则 slab 已满,所以将该 slab 移到 slabs_full 链表中。

1471 函数末尾。

1428~1433 如果设置了 SLAB_DEBUG_INITIAL 标志位,则调用构造函数来验证该对象处于初始化的状态。

1435~1444 检查对象两端的红色标记。这将检查超出对象边界写和重复释放。

1445~1446 以一种已知模式来感染释放的对象。

1447~1448 这个函数确定该对象是这个 slab 和高速缓存的一部分。它将检查空闲链表(bufctl)以保证这不是一次重复释放。

H. 3.3.5 函数: `free_block()` (`mm/slab.c`)

这个函数仅用于 SMP 情形中 per-CPU 高速缓存变得过满时。它用于批量释放对象。

```
1481 static void free_block (kmem_cache_t * cachep, void ** objpp,
                           int len)
1482 {
1483     spin_lock(&cachep->spinlock);
1484     __free_block(cachep, objpp, len);
1485     spin_unlock(&cachep->spinlock);
1486 }
```

1481 参数如下:

- cachep 是从中释放对象的高速缓存。
- objpp 指向待释放的第一个对象的指针。
- len 是待释放对象的数量。

1484 获取对高速缓存描述符的锁。

1484 `__free_block()`(见 H. 3.3.6)来进行释放每个页面的实际工作。

1485 释放锁。

H. 3.3.6 函数: `__free_block()` (`mm/slab.c`)

这个函数负责释放 per-CPU 数组 objpp 中的每个对象。

```
1474 static inline void __free_block (kmem_cache_t * cachep,
1475     void ** objpp, int len)
```

```
1476 {  
1477     for ( ; len > 0; len-- , objpp++ )  
1478         kmem_cache_free_one(cachep, *objpp);  
1479 }
```

1474 参数是对象所属的 cache, 对象链表(objpp)和待释放对象的数目(len)。

1477 循环 len 次。

1478 从数组中释放对象。

H.4 指定大小的高速缓存

H. 4.1 初始化指定大小的高速缓存

H. 4. 1. 1 函数: kmem_cache_sizes_init() (mm/slab.c)

这个函数负责为适应普通或 DMA 内存的小内存缓冲区创建高速缓存对。

```
436 void __init kmem_cache_sizes_init(void)
437 {
438     cache_sizes_t * sizes = cache_sizes;
439     char name[20];
440
441     if (num_physpages > (32 << 20) >> PAGE_SHIFT)
442         slab_break_gfp_order = BREAK_GFP_ORDER_HI;
443
444     do {
445         sprintf(name, sizeof(name), "size- %Zd",
446                 sizes->cs_size);
447         if (! (sizes->cs_cachep =
448             kmem_cache_create(name, sizes->cs_size,
449                               0, SLAB_HWCACHE_ALIGN, NULL, NULL))) {
450             BUG();
451         }
452
453         if (! (OFF_SLAB(sizes->cs_cachep))) {
454             offslab_limit = sizes->cs_size - sizeof(slab_t);
455             offslab_limit /= 2;
456         }
457
458         if (offslab_limit >= 1024) {
459             offslab_limit = 1024;
460         }
461     } while (offslab_limit < 1024);
462
463 }
```

```

        sizes->cs_size);
465 sizes->cs_dmacachep = kmem_cache_create(name,
        sizes->cs_size, 0,
466 SLAB_CACHE_DMA|SLAB_HWCACHE_ALIGN,
        NULL, NULL);
467 if (! sizes->cs_dmacachep)
468 BUG();
469 sizes++;
470 } while (sizes->cs_size);
471 }

```

438 获取指向 cache_sizes 数组的指针。

439 高速缓存的可读名。它必须是指定大小的 CACHE_NAMELEN, 定义为 20 个字节长。

444~445 除非 0 个对象填入 slab, 否则 slab_break_gfp_order 决定了一个 slab 可以使用的页面数。它静态地初始化为 BREAK_GFP_ORDER_LO(1)。这里检查以确定是否有多于 32 MB 的内存, 如果有, 则允许使用 BREAK_GFP_ORDER_HI 个页面, 这是因为有更多内存可用时可以接受更多的内部碎片。

446~470 为需要的每个内存分配大小以创建两个高速缓存。

452 将可读的高速缓存名字存放于 name 中。

453~454 创建高速缓存, 与 L1 高速缓存对齐。

460~463 计算 off-slab 的 bufctl 限制, 当 slab 描述符不在高速缓存中时它决定了高速缓存可以存放的对象数目。

464 用于 DMA 的高速缓存可读名。

465~466 创建对齐于 L1 高速缓存以及适合用于 DMA 的高速缓存。

467 如果分配高速缓存失败, 则这是一个 bug, 如果在这之前没有可用的内存, 则系统不会启动。

469 移到 cache_sizes 数组中的下一个元素。

470 数组以最后一个元素 0 结尾。

H. 4.2 kmalloc()

H. 4.2.1 函数: kmalloc() (mm/slab.c)

这个函数的调用图如图 8.16 所示。

```

1555 void * kmalloc (size_t size, int flags)
1556 {

```

```

1557 cache_sizes_t * csizep = cache_sizes;
1558
1559 for (; csizep->cs_size; csizep++) {
1560 if (size > csizep->cs_size)
1561 continue;
1562 return __kmem_cache_alloc(flags & GFP_DMA ?
1563 csizep->cs_dmacachep :
1564 csizep->cs_cachep, flags);
1564 }
1565 return NULL;
1566 }

```

1557 `cache_sizes` 是每个指定大小的高速缓存数组(见 8.4 节)。

1559~1564 从最小的高速缓存开始,这里检查每个高速缓存,直至找到了满足请求的足够大的高速缓存。

1562 如果分配用于 DMA,则这里从 `cs_dmacachep` 中分配一个对象,否则从 `cs_cachep` 中分配一个对象。

1565 如果没有一个足够大的指定大小的高速缓存或者无法分配一个对象,则这里返回失败。

H.4.3 kfree()

H.4.3.1 函数: kfree() (mm/slab.c)

这个函数的调用图如图 8.17 所示。值得注意的是这个函数与带调试标志的 `kmem_cache_free()`(见 H.3.3.1)几乎一样。

```

1597 void kfree (const void * objp)
1598 {
1599 kmem_cache_t * c;
1600 unsigned long flags;
1601
1602 if (! objp)
1603 return;
1604 local_irq_save(flags);
1605 CHECK_PAGE(virt_to_page(objp));
1606 c = GET_PAGE_CACHE(virt_to_page(objp));
1607 __kmem_cache_free(c, (void *)objp);
1608 local_irq_restore(flags);
1609 }

```

1602 如果指针为 NULL 则返回。如果一个调用者使用 kmalloc() 并在捕捉异常路径上直接调用了 kfree() 就有可能发生这种情形。

1604 关闭中断。

1605 保证这个对象页面是 slab 中的页面。

1606 获取这个指针所属的高速缓存(见 8.2 节)。

1607 释放内存对象。

1608 重新打开中断。

H.5 Per-CPU 对象高速缓存

per-CPU 对象的结构以及如何向 per-CPU 中加入或移除对象,在 8.5.1 小节和 8.5.2 小节有详细讨论。

H.5.1 启动 Per-CPU 高速缓存

H.5.1.1 函数: enable_all_cpucaches() (mm/slab.c)

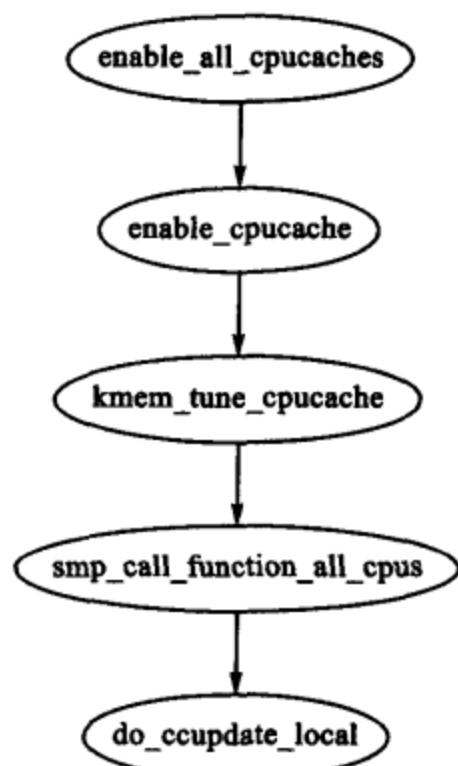


图 H.1 调用图: enable_all_cpucaches()

这个函数上锁高速缓存链表,并启动每个高速缓存的 cpucache。这在启动 cache_cache

和指定大小的高速缓存之后很重要。

```

1714 static void enable_all_cpucaches (void)
1715 {
1716     struct list_head * p;
1717
1718     down(&cache_chain_sem);
1719
1720     p = &cache_cache.next;
1721     do {
1722         kmem_cache_t * cachep = list_entry(p, kmem_cache_t, next);
1723
1724         enable_cpucache(cachep);
1725         p = cachep->next.next;
1726     } while (p != &cache_cache.next);
1727
1728     up(&cache_chain_sem);
1729 }
```

1718 获取高速缓存链表的信号量。

1719 获取链表中第 1 个高速缓存。

1721~1726 遍历整个高速缓存链表。

1722 从链表中获取一个高速缓存,这里的代码将跳过链表中的第 1 个高速缓存。但由于 cache_cache 很少使用,所以它不需要一个 cpucache。

1724 启动 cpucache。

1725 移到链表中的下一个高速缓存。

1726 释放高速缓存链表信号量。

H.5.1.2 函数: enable_cpucache() (mm/slab.c)

这个函数基于高速缓存中对象的大小计算 cpucache 大小,然后调用 kmem_tune_cpucache()来进行实际的分配工作。

```

1693 static void enable_cpucache (kmem_cache_t * cachep)
1694 {
1695     int err;
1696     int limit;
1697
1699     if (cachep->objsize > PAGE_SIZE)
1700         return;
1701     if (cachep->objsize > 1024)
```

```

1702 limit = 60;
1703 else if (cachep->objsize > 256)
1704 limit = 124;
1705 else
1706 limit = 252;
1707
1708 err = kmem_tune_cpucache(cachep, limit, limit/2);
1709 if (err)
1710 printk(KERN_ERR
           "enable_cpucache failed for %s, error %d.\n",
1711 cachep->name, -err);
1712 }

```

1699~1700 如果一个对象大于一页,则返回以避免为该对象创建一个 per-CPU 高速缓存,因为 per-CPU 高速缓存的代价比较高。

1701~1702 如果一个对象大于 1 KB,则这里将 cpucache 保持为小于 3 MB 的大小。这里对 124 个对象的限制是考虑到 cpucache 描述符的大小。

1703~1704 对更小的对象,这里仅保证高速缓存不会超过 3 MB。

1708 为 cpucache 分配内存。

1710~1711 如果分配失败,则打印一条错误信息。

H.5.1.3 函数: kmem_tune_cpucache() (mm/slab.c)

这个函数负责为 cpucache 分配内存。对系统中的每个 CPU,kmalloc 给每个 cpucache 分配一块足够大的内存并填充 ccupdate_struct_t 结构。函数 swap_call_function_all_cpus() 然后调用 do_ccupdate_local(), 它将高速缓存描述符中的旧信息用新信息替换。

```

1639 static int kmem_tune_cpucache (kmem_cache_t * cachep,
1640                                int limit, int batchcount)
1641 ccupdate_struct_t new;
1642 int i;
1643
1644 /*
1645 * These are admin-provided, so we are more graceful.
1646 */
1647 if (limit < 0)
1648 return -EINVAL;
1649 if (batchcount < 0)
1650 return -EINVAL;

```

```
1651 if (batchcount > limit)
1652 return -EINVAL;
1653 if (limit != 0 && !batchcount)
1654 return -EINVAL;
1655
1656 memset(&new.new, 0, sizeof(new.new));
1657 if (limit) {
1658 for (i = 0; i < smp_num_cpus; i++) {
1659 cpucache_t * ccnew;
1660
1661 ccnew = kmalloc(sizeof(void *) * limit +
1662 sizeof(cpucache_t),
1663 GFP_KERNEL);
1664 if (!ccnew)
1665 goto oom;
1666 ccnew->limit = limit;
1667 ccnew->avail = 0;
1668 new.new[cpu_logical_map(i)] = ccnew;
1669 }
1670 new.cachep = cachep;
1671 spin_lock_irq(&cachep->spinlock);
1672 cachep->batchcount = batchcount;
1673 spin_unlock_irq(&cachep->spinlock);
1674
1675 smp_call_function_all_cpus(do_ccupdate_local, (void *)&new);
1676
1677 for (i = 0; i < smp_num_cpus; i++) {
1678 cpucache_t * ccold = new.new[cpu_logical_map(i)];
1679 if (!ccold)
1680 continue;
1681 local_irq_disable();
1682 free_block(cachep, cc_entry(ccold), ccold->avail);
1683 local_irq_enable();
1684 kfree(ccold);
1685 }
1686 return 0;
1687 oom:
1688 for (i--; i >= 0; i--)
```

```

1689 kfree(new.new[cpu_logical_map(i)]);
1690 return -ENOMEM;
1691 }

```

1639 这个函数的参数如下：

- cachep 是分配给 cpucache 的高速缓存。
- limit 可以存在于 cpucache 中的对象总数。
- batchcount 是在 cpucache 为空时一次分配的对象数。

1647 在高速缓存中的对象数目不能为负数。

1649 不能分配负数个对象。

1651 不能一次分配大于限制数量的对象。

1653 如果限制为整数，则必须提供 batchcount。

1656 用 0 填充来更新结构。

1657 如果提供了限制，则这里为 cpucache 分配内存。

1658~1668 对每个 CPU，这里分配一个 cpucache。

1661 所需的内存是 limit 个指针和 cpucache 描述符的大小。

1663 如果内存不足则在这里清理并退出。

1665~1666 填充 cpucache 描述符的字段。

1667 填充 ccupdate_update_t 结构的信息。

1670 告知 ccupdate_update_t 结构正在更新哪个高速缓存。

1671~1673 获取对高速缓存描述符的中断安全锁并设置它的 batchcount。

1675 获取 CPU 以更新它自身的 cpucache 信息。do_ccupdate_local() (见 H. 5.2.2) 用 new 中的新 cpucache 替换高速缓存描述符中的旧 cpucache。

1677~1685 在 smp_call_function_all_cpus() (见 H. 5.2.1) 后，旧的 cpucache 在 new 中。这一块代码遍历它们，释放它们中所有对象并删除旧的 cpucache。

1686 返回成功。

1688 内存不足时，这里删除所有已经分配的 cpucache，并返回失败。

H. 5.2 更新 Per-CPU 信息

H. 5.2.1 函数：smp_call_function_all_cpus() (mm/slab.c)

这个函数调用所有 CPU 的 func() 函数。在 slab 分配器上下文中，func() 为 do_ccupdate_local()，参数是 ccupdate_struct_t。

```

859 static void smp_call_function_all_cpus(void (*func)(void *arg),

```

```

void * arg)
860 {
861 local_irq_disable();
862 func(arg);
863 local_irq_enable();
864
865 if (smp_call_function(func, arg, 1, 1))
866 BUG();
867 }

```

861~863 关闭局部中断并调用该 CPU 的这个函数。

865 对其他 CPU, 这里调用函数 smp_call_function(), 它是一个特定体系结构的函数, 将不在这里做进一步讨论。

H.5.2.2 函数: do_ccupdate_local() (mm/slab.c)

这个函数用这个 CPU 中的 info 信息替换高速缓存描述符中的 cpucache 信息。

```

874 static void do_ccupdate_local(void * info)
875 {
876 ccupdate_struct_t * new = (ccupdate_struct_t *)info;
877 cpucache_t * old = cc_data(new->cachep);
878
879 cc_data(new->cachep) = new->new[smp_processor_id()];
880 new->new[smp_processor_id()] = old;
881 }

```

876 info 是一个指向 ccupdate_struct_t 的指针, 它将被传入 smp_call_function_all_cpus()(见 H.5.2.1)。

877 ccupdate_struct_t 部分是一个指向这个 cpucache 所属高速缓存的指针。cc_data()返回这个处理器的 cpucache_t。

879 将新的 cpucache 放入高速缓存描述符中。cc_data()返回指向这个 CPU 的 cpucache 指针。

880 将 new 中的指针替换为旧的 cpucache, 这样后面可以用例如 smp_call_function_call_cpus(), kmem_tune_cpucache() 来删除它。

H.5.3 清理 Per-CPU 高速缓存

调用这个函数来清理 per-CPU 高速缓存中的所有对象。当需要释放 slab 来收缩高速缓存的时候就调用它。如果对象处于 per-CPU 高速缓存, 即使没有使用也可以说该 slab 可以被释放。

H. 5.3.1 函数: drain_cpu_caches() (mm/slab.c)

```

885 static void drain_cpu_caches(kmem_cache_t * cachep)
886 {
887     ccupdate_struct_t new;
888     int i;
889
890     memset(&new, new, sizeof(new));
891
892     new.cachep = cachep;
893
894     down(&cache_chain_sem);
895     smp_call_function_all_cpus(do_ccupdate_local, (void *) &new);
896
897     for (i = 0; i < smp_num_cpus; i++) {
898         cpucache_t * ccold = new.new[cpu_logical_map(i)];
899         if (!ccold || (ccold->avail == 0))
900             continue;
901         local_irq_disable();
902         free_block(cachep, cc_entry(ccold), ccold->avail);
903         local_irq_enable();
904         ccold->avail = 0;
905     }
906     smp_call_function_all_cpus(do_ccupdate_local, (void *) &new);
907     up(&cache_chain_sem);
908 }

```

890 将更新结构清空,因为马上将清空所有的数据。

892 设置 new.cachep 指向 cachep,这样 smp_call_function_all_cpus()知道哪个高速缓存受影响。

894 获取高速缓存描述符信号量。

895 do_ccupdate_local()(见 H.5.2.2)利用 new 中的 cpucache_t 信息来替换高速缓存描述符中的 cpucache_t 信息,这样它们在这里可以改变。

897~905 对系统中的每个 CPU,.....

898 获取该 CPU 中的 cpucache 描述符。

899 如果由于某个原因这个结构不存在,或其中不存在对象,则移到下一个 CPU。

901 在这个处理器上关闭中断。有可能在其他地方中断处理程序的分配将试着访问 per-CPU 高速缓存。

- 902 利用 free_block() (见 H. 3.3.5) 释放对象块。
 903 重新打开中断。
 904 显示没有对象可用。
 906 每个 CPU 的信息已经更新, 所以这里对每个 CPU 调用 do_ccupdate_local() (见 H. 5.2.2) 来将信息重新放入到高速缓存描述符中。
 907 释放高速缓存链表中的信号量。

H.6 初始化 Slab 分配器

H.6.1.1 函数: kmem_cache_init() (mm/slab.c)

这个函数将做如下工作:

- 初始化高速缓存链表。
- 初始化访问高速缓存链表的 mutex。
- 计算 cache_cache 颜色。

```
416 void __init kmem_cache_init(void)
417 {
418     size_t left_over;
419
420     init_MUTEX(&cache_chain_sem);
421     INIT_LIST_HEAD(&cache_chain);
422
423     kmem_cache_estimate(0, cache_cache.objsize, 0,
424     &left_over, &cache_cache.num);
425     if (!cache_cache.num)
426         BUG();
427
428     cache_cache.colour = left_over/cache_cache.colour_off;
429     cache_cache.colour_next = 0;
430 }
```

- 420 初始化访问高速缓存链表的 semaphore 信号量。
 421 初始化高速缓存链表。
 423 kmem_cache_estimate() (见 H.1.2.1) 计算对象的数量以及消耗的字节数。
 425 如果 kmem_cache_t 不能存放在页面中, 肯定是某个地方出现了很严重的错误。
 428 colour 是 L1 高速缓存对齐时可用的不同高速缓存行数目。
 429 colour_next 表明要使用的下个高速缓存行, 它从 0 开始。

H.7 与伙伴分配器的接口

H.7.1.1 函数: kmem_getpages() (mm/slab.c)

这里为 slab 分配器分配页面。

```

486 static inline void * kmem_getpages (kmem_cache_t * cachep,
487     unsigned long flags)
488 {
489     void * addr;
490     flags |= cachep->gfpflags;
491     addr = (void *) __get_free_pages(flags, cachep->gfporder);
492     return addr;
493 }
```

495 分配所请求的无论是何种标志,都取决于高速缓存的标志位。如果需要 DMA 内存,则惟一可以追加的标志位是 ZONE_DMA。

496 利用 __get_free_pages() (见 F.2.3 小节)从伙伴分配器中分配。

503 返回页面,如果分配失败则返回 NULL。

H.7.1.2 函数: kmem_freepages() (mm/slab.c)

这里为 slab 分配器释放页面。在调用伙伴分配器 API 之前,它将从页面标志位中移除 PG_slab 位。

```

507 static inline void kmem_freepages (kmem_cache_t * cachep, void * addr)
508 {
509     unsigned long i = (1<<cachep->gfporder);
510     struct page * page = virt_to_page(addr);
511
512     while (i--) {
513         PageClearSlab(page);
514         page++;
515     }
516     free_pages((unsigned long)addr, cachep->gfporder);
517 }
```

509 检索用于原始分配的次。

510 获取该地址的结构页面。

517~520 清除每一页中的 PG_slab 位。

521 利用 free_pages() (见 F.4.1 小节)从伙伴分配器中释放页面。

附录 I

高端内存管理

目 录

| | |
|------------------------------------|-----|
| I. 1 映射高端内存页面 | 514 |
| I. 1. 1 函数:kmap() | 514 |
| I. 1. 2 函数:kmap_nonblock() | 514 |
| I. 1. 3 函数:_kmap() | 514 |
| I. 1. 4 函数:kmap_high() | 515 |
| I. 1. 5 函数:map_new_virtual() | 515 |
| I. 1. 6 函数:flush_all_zero_pkmaps() | 518 |
| I. 2 自动映射高端内存页面 | 519 |
| I. 2. 1 函数:kmap_atomic() | 519 |
| I. 3 解除页面映射 | 521 |
| I. 3. 1 函数:kunmap() | 521 |
| I. 3. 2 函数:kunmap_high() | 521 |
| I. 4 自动解除高端内存页面映射 | 523 |
| I. 4. 1 函数:kunmap_atomic() | 523 |
| I. 5 弹性缓冲区 | 524 |
| I. 5. 1 创建弹性缓冲区 | 524 |
| I. 5. 1. 1 函数:create_bounce() | 524 |
| I. 5. 1. 2 函数:alloc_bounce_bh() | 526 |
| I. 5. 1. 3 函数:alloc_bounce_page() | 527 |
| I. 5. 2 用弹性缓冲区复制 | 528 |

| | |
|-----------------------------------|-----|
| I. 5.2.1 函数:bounce_end_io_write() | 528 |
| I. 5.2.2 函数:bounce_end_io_read() | 528 |
| I. 5.2.3 函数:copy_from_high_bh() | 529 |
| I. 5.2.4 函数:copy_to_high_bh_irq() | 529 |
| I. 5.2.5 函数:bounce_end_io() | 530 |
| I. 6 紧急池 | 532 |
| I. 6.1 函数:init_emergency_pool() | 532 |

I. 1 映射高端内存页面

I. 1.1 函数: kmap() (include/asm-i386/highmem. c)

这个 API 用于进行阻塞的调用者。

62 #define kmap(page) __kmap(page, 0)

62 核心函数__kmap()的第 2 个参数表示调用者将进行阻塞。

I. 1.2 函数: kmap_nonblock() (include/asm-i386/highmem. c)

63 #define kmap_nonblock(page) __kmap(page, 1)

63 核心函数__kmap()的第 2 个参数表示调用者将不进行阻塞。

I. 1.3 函数: __kmap() (include/asm-i386/highmem. h)

调用图如图 9.1 所示。

```

65 static inline void * kmap(struct page * page, int nonblocking)
66 {
67 if (in_interrupt())
68 out_of_line_bug();
69 if (page < highmem_start_page)
70 return page_address(page);
71 return kmap_high(page);

```

72 }

67~68 这个函数不会在中断时调用,因为它已睡眠。`out_of_line_bug()`调用 `do_exit` 并返回错误码来取代 `BUG()`。之所以不调用 `BUG()` 是因为 `BUG()` 用极端方式杀掉进程,这将引起如寓言“*Aiee, 杀掉中断句柄!*”般的内核瘫痪。

69~70 如果页面已在低端内存中,则返回一个直接映射。

71 调用 `kmap_high()`(见 I. 1.4 小节)完成与体系结构无关的工作。

I. 1.4 函数: `kmap_high()` (`mm/highmem.c`)

```

132 void * kmap_high(struct page * page, int nonblocking)
133 {
134     unsigned long vaddr;
135
142     spin_lock(&kmap_lock);
143     vaddr = (unsigned long) page->virtual;
144     if (! vaddr) {
145         vaddr = map_new_virtual(page, nonblocking);
146     if (! vaddr)
147         goto out;
148 }
149     pkmap_count[PKMAP_NR(vaddr)]++;
150     if (pkmap_count[PKMAP_NR(vaddr)] < 2)
151         BUG();
152     out:
153     spin_unlock(&kmap_lock);
154     return (void *) vaddr;
155 }
```

142 `kmap_lock` 保护页面的 `virtual` 字段和 `pkmap_count` 数组。

143 获取页面的虚拟地址。

144~148 如果尚未映射,则调用 `map_new_virtual()`,进行页面映射并返回其虚拟地址。如果失败,则跳转到 `out` 处,释放自旋锁并返回 `NULL`。

149 增加页面映射的引用计数。

150~151 如果计数现在小于 2,则这是个严重的 bug。在实际运行中,严重的系统问题可能会引起这个 bug。

153 释放 `kmap_lock`。

I. 1.5 函数: map_new_virtual() (mm/highmem.c)

这个函数分为 3 个主要部分: 查找一个空闲槽; 如果没有可用的槽则在队列上等待; 然后映射该页面。

```

80 static inline unsigned long map_new_virtual(struct page * page)
81 {
82     unsigned long vaddr;
83     int count;
84
85     start:
86     count = LAST_PKMAP;
87     /* Find an empty entry */
88     for (;;) {
89         last_pkmap_nr = (last_pkmap_nr + 1) & LAST_PKMAP_MASK;
90         if (!last_pkmap_nr) {
91             flush_all_zero_pkmaps();
92             count = LAST_PKMAP;
93         }
94         if (!pkmap_count[last_pkmap_nr])
95             break; /* Found a usable entry */
96         if (--count)
97             continue;
98
99         if (nonblocking)
100            return 0;

```

86 从最后一个可能的槽开始扫描。

88~122 持续扫描并等待直至有一个槽空闲。这里可能导致某些进程进入死循环。

89 last_pkmap_nr 是扫描中最后一个 pkmap。为阻止搜索同一个页面, 记录该值就可以进行循环搜索。如果达到 LAST_PKMAP, 则折返为 0。

90~93 在 last_pkmap_nr 折返为 0 时, 调用 flush_all_zero_pkmaps() (见 I.1.6 小节), 在刷新 TLB 之前设置 pkmap_count 数组中所有的项从 1 改为 0。接着重新设置 LAST_PKMAP 表示再次开始扫描。

94~95 如果元素为 0, 则表示为页面找到了一个可用槽。

96~97 转到下一个下标, 开始扫描。

99~100 下一块代码将睡眠, 等待一个槽变空闲。如果调用者要求不阻塞该函数, 则直接

返回。

```

105 {
106 DECLARE_WAITQUEUE(wait, current);
107
108 current->state = TASK_UNINTERRUPTIBLE;
109 add_wait_queue(&pkmap_map_wait, &wait);
110 spin_unlock(&kmap_lock);
111 schedule();
112 remove_wait_queue(&pkmap_map_wait, &wait);
113 spin_lock(&kmap_lock);
114
115 /* Somebody else might have mapped it while we
   slept */
116 if (page->virtual)
117 return (unsigned long) page->virtual;
118
119 /* Re-start */
120 goto start;
121 }
122 }
```

如果在扫描完所有页面一遍后仍没有可用槽，则在 pkmap_map_wait 队列上睡眠直至在解除映射后被唤醒。

106 声明该等待队列。

108 设置队列可中断，因为是在内核空间中睡眠。

109 添加自己到 pkmap_map_wait 队列。

110 释放 kmap_lock 自旋锁。

111 调用 schedule()，使自己开始睡眠。在解除映射后如果有槽空闲，则自己被唤醒。

112 从等待队列移除自己。

113 重新获取 kmap 锁。

116~117 如果在睡眠时有其他的事务映射了该页面，则仅仅返回地址，并由 kmap_high() 增加引用计数。

120 重新开始扫描。

```

123 vaddr = PKMAP_ADDR(last_pkmap_nr);
124 set_pte(&(pkmap_page_table[last_pkmap_nr]), mk_pte(page,
   kmap_prot));
125
```

```

126 pkmap_count[last_pkmap_nr] = 1;
127 page->virtual = (void *) vaddr;
128
129 return vaddr;
130 }

```

这块代码在找到一个槽时进行处理,用于映射页面。

123 获取被找到槽的虚拟地址。

124 确保 PTE 与页面对应且已获得必需的保护,将其放置在页表中被找到槽的地方。

126 初始化 pkmap_count 数组的值为 1。该计数在父函数中增加,如果这是第一次在该函数处出现,可以保证这是第一次映射。

127 设置页面的虚拟字段。

129 返回虚拟地址。

I. 1.6 函数: flush_all_zero_pkmaps() (mm/highmem.c)

这个函数循环遍历 pkmap_count 数组并在刷新 TLB 之前设置所有的项从 1 变为 0。

```

42 static void flush_all_zero_pkmaps(void)
43 {
44     int i;
45
46     flush_cache_all();
47
48     for (i = 0; i < LAST_PKMAP; i++) {
49         struct page *page;
50
51         if (pkmap_count[i] != 1)
52             continue;
53         pkmap_count[i] = 0;
54
55         /* sanity check */
56         if (pte_none(pkmap_page_table[i]))
57             BUG();
58
59         page = pte_page(pkmap_page_table[i]);
60         pte_clear(&pkmap_page_table[i]);
61
62         page->virtual = NULL;

```

```
76 }
77 flush_tlb_all();
78 }
```

46 由于全局页表将变化,因此必须刷新所有处理器的 CPU 高速缓存。

48~76 循环遍历整个 pkmap_count 数组。

57~58 如果元素不为 1,则移到下一个元素。

59 从 1 设置为 0。

62~63 确保 PTE 没有被映射过。

72~73 从 PTE 解除对页面的映射,并清除 PTE。

75 更新虚拟字段为页面未被映射。

77 刷新 TLB。

I. 2 自动映射高端内存页面

下面是 x86 中 km_type 枚举类型的例子。这里列举了自动调用 kmap 的不同中断。由于 KM_TYPE_NR 是最后一个元素,所以它用作所有元素的计数器。

```
4 enum km_type {
5 KM_BOUNCE_READ,
6 KM_SKB_SUNRPC_DATA,
7 KM_SKB_DATA_SOFTIRQ,
8 KM_USER0,
9 KM_USER1,
10 KM_BH_IRQ,
11 KM_TYPE_NR
12 };
```

I. 2. 1 函数: kmap_atomic() (include/asm-i386/highmem.h)

这是 kmap()的原子版本。请注意,不论在什么时候都不持有自旋锁或不睡眠。之所以不需要自旋锁是因为每个处理器都有它自己的保留空间。

```
89 static inline void * kmap_atomic(struct page * page,
  enum km_type type)
90 {
91 enum fixed_addresses idx;
```

```

92 unsigned long vaddr;
93
94 if (page < highmem_start_page)
95 return page_address(page);
96
97 idx = type + KM_TYPE_NR * smp_processor_id();
98 vaddr = __fix_to_virt(FIX_KMAP_BEGIN + idx);
99 #if HIGHMEM_DEBUG
100 if (!pte_none(* (kmap_pte+idx)))
101 out_of_line_bug();
102 #endif
103 set_pte(kmap_pte+idx, mk_pte(page, kmap_prot));
104 __flush_tlb_one(vaddr);
105
106 return (void *) vaddr;
107 }

```

89 这里的参数为映射的页面和处理所需的类型。一个槽的使用对应一个处理器。

94~95 如果页面在低端内存，则返回直接映射。

97 type 给出了使用哪个槽。而 KM_TYPE_NR * smp_processor_id() 则给出了为处理器保留的槽的集合。

98 获取虚拟地址。

100~101 用于调试的代码。在实际运行中，PTE 将退出。

103 为保留的槽设置 PTE。

104 为槽刷新 TLB。

106 返回虚拟地址。

I.3 解除页面映射

I.3.1 函数：kunmap() (include/asm-i386/highmem.h)

```

74 static inline void kunmap(struct page * page)
75 {
76 if (in_interrupt())
77 out_of_line_bug();
78 if (page < highmem_start_page)

```

```

79 return;
80 kunmap_high(page);
81 }

```

76~77 `kunmap()` 不能在中断中进行调用,因此自然退出。

78~79 如果页面已在低端内存中,则不需要进行解除映射的操作。

80 调用与体系结构无关的函数 `kunmap_high()`。

I. 3. 2 函数: `kunmap_high()` (mm/highmem.c)

这是与体系结构无关的 `kunmap()` 操作部分。

```

157 void kunmap_high(struct page * page)
158 {
159     unsigned long vaddr;
160     unsigned long nr;
161     int need_wakeup;
162
163     spin_lock(&kmap_lock);
164     vaddr = (unsigned long) page->virtual;
165     if (! vaddr)
166         BUG();
167     nr = PKMAP_NR(vaddr);
168
169     need_wakeup = 0;
170     switch ( -- pkmap_count[nr] ) {
171     case 0:
172         BUG();
173     case 1:
174         need_wakeup = waitqueue_active(&pkmap_map_wait);
175     }
176     spin_unlock(&kmap_lock);
177
178 /* do wake-up, if needed, race-free outside of the spin lock */
179     if (need_wakeup)
180         wake_up(&pkmap_map_wait);
181 }

```

163 获取 `kmap` 锁,用来保护虚拟字段和 `pkmap_count` 数组。

164 获取虚拟页面。

165~166 如果没有设置虚拟字段,那这是两次解除映射的操作或是对没有映射页面的解除映射操作,则调用 BUG()。

167 获取在 pkmap_count 数组中的下标。

173 在缺省情况下,并不需要唤醒进程来调用 kmap()。

174 在减小下标后检查其值。

175~176 如果减为 0,那这是个 bug,因为需要刷新 TLB 以使 0 成为合法项。

177~188 如果减小到 1(项虽然已释放,但还需要刷新 TLB),则检查是否有谁在 pkmap_map_wait_queue 队列上睡眠。如果有必要,则在释放自旋锁后唤醒队列。

190 释放 kmap_lock。

193~194 如果还有等待者在队列上而槽已被释放,则唤醒它们。

I. 4 自动解除高端内存页面映射

I. 4. 1 函数: **kunmap_atomic()** (include/asm-i386/highmem.h)

整个函数是调试代码。其原因是页面只在此处自动映射,那么在解除映射以前只会在很小的范围且较短的时间内被使用。保留页面在那里是安全的,因为在解除映射之后不会再引用它们,而对同一个槽的另一个映射会简单地替换它。

```

109 static inline void kunmap_atomic(void * kvaddr, enum km_type type)
110 {
111 #if HIGHMEM_DEBUG
112     unsigned long vaddr = (unsigned long) kvaddr & PAGE_MASK;
113     enum fixed_addresses idx = type + KM_TYPE_NR * smp_processor_id();
114
115     if (vaddr < FIXADDR_START) // FIXME
116         return;
117
118     if (vaddr != __fix_to_virt(FIX_KMAP_BEGIN + idx))
119         out_of_line_bug();
120
121     /*
122     * force other mappings to Oops if they'll try to access
123     * this pte without first remap it
124     */
125     pte_clear(kmap_pte_idx);

```

```

126 __flush_tlb_one(vaddr);
127 #endif
128 }

```

112 获取虚拟地址并确保它和单个页边界对齐。

115~116 如果提供的地址不在定长区域内,则返回。

118~119 如果地址与使用类型及处理器不对应,则声明它。

125~126 现在解除页面映射,如果再次引用它,则会产生 Oops。

I.5 弹性缓冲区

I.5.1 创建弹性缓冲区

I.5.1.1 函数: `create_bounce()` (`mm/highmem.c`)

函数调用图如图 9.3 所示。它是创建弹性缓冲区的高层函数。它由两部分组成,即分配必要的资源和从模板复制数据。

```

405 struct buffer_head * create_bounce(int rw,
406                                     struct buffer_head * bh_orig)
407 {
408     struct page * page;
409     struct buffer_head * bh;
410
411     if (!PageHighMem(bh_orig->b_page))
412         return bh_orig;
413
414     bh = alloc_bounce_bh();
415     page = alloc_bounce_page();
416
417     set_bh_page(bh, page, 0);
418
419     return bh;

```

405 函数的参数如下所示:

- `rw` 如果是写缓冲区,则设置为 1。
- `bh_orig` 是模板缓冲区头,是复制数据源。

410~411 如果模板缓冲区头已在低端内存中,则简单地返回它。

413 从 slab 分配器分别缓冲区头,如果失败则从紧急池分配。

420 从伙伴分配器分配页面,如果失败则从紧急池分配。

422 将分配的页面与分配的缓冲区头关联起来。

```

424 bh->b_next = NULL;
425 bh->b_blocknr = bh_orig->b_blocknr;
426 bh->b_size = bh_orig->b_size;
427 bh->b_list = -1;
428 bh->b_dev = bh_orig->b_dev;
429 bh->b_count = bh_orig->b_count;
430 bh->b_rdev = bh_orig->b_rdev;
431 bh->b_state = bh_orig->b_state;
432 #ifdef HIGHMEM_DEBUG
433 bh->b_flushtime = jiffies;
434 bh->b_next_free = NULL;
435 bh->b_prev_free = NULL;
436 /* bh->b_this_page */
437 bh->b_reqnext = NULL;
438 bh->b_pprev = NULL;
439#endif
440 /* bh->b_page */
441 if (rw == WRITE) {
442 bh->b_end_io = bounce_end_io_write;
443 copy_from_high_bh(bh, bh_orig);
444 } else
445 bh->b_end_io = bounce_end_io_read;
446 bh->b_private = (void *)bh_orig;
447 bh->b_rsector = bh_orig->b_rsector;
448 #ifdef HIGHMEM_DEBUG
449 memset(&bh->b_wait, -1, sizeof(bh->b_wait));
450#endif
451
452 return bh;
453 }
```

这块产生新的缓冲区头。

431 逐个复制必要的信息,除了 b_list 字段,因为该缓冲区并不与该链表上的其他部分直接连接。

433~438 仅用于调试的信息。

441~444 如果这个缓冲区将被写入,则用于结束 I/O 的回调函数是 bounce_end_io_write()

(见 I.5.2.1 节), 它在设备接收所有信息后调用。由于数据在高端内存中, 则从 `copy_from_high_bh()`(见 I.5.2.3) 复制“下来”。

437~438 如果等待设备写数据到缓冲区中, 则调用 `bounce_end_io_read()`(见 I.5.2.2) 回调函数。

446~447 从模板缓冲区头复制剩下的信息。

452 返回新弹性缓冲区。

I.5.1.2 函数: `alloc_bounce_bh()` (`mm/highmem.c`)

这个函数首先尝试从 slab 分配器分配一个 `buffer_head`, 如果失败, 则使用紧急池。

```
369 struct buffer_head * alloc_bounce_bh (void)
370 {
371 struct list_head * tmp;
372 struct buffer_head * bh;
373
374 bh = kmem_cache_alloc(bh_cachep, SLAB_NOHIGHIO);
375 if (bh)
376 return bh;
380
381 wakeup_bdfflush();
```

374 尝试从 slab 分配器分配一个新 `buffer_head`。请注意, 如何产生不使用 I/O 操作的请求以避免递归, 包括高级 I/O。

375~376 如果分配成功, 则返回。

381 如果失败, 则唤醒 `bdfflush` 清洗页面。

```
383 repeat_alloc:
387 tmp = &emergency_bhs;
388 spin_lock_irq(&emergency_lock);
389 if (! list_empty(tmp)) {
390 bh = list_entry(tmp->next, struct buffer_head,
391 b_inode_buffers);
391 list_del(tmp->next);
392 nr_emergency_bhs -- ;
393 }
394 spin_unlock_irq(&emergency_lock);
395 if (bh)
396 return bh;
397
398 /* we need to wait I/O completion */
```

```

399 run_task_queue(&tq_disk);
400
401 yield();
402 goto repeat_alloc;
403 }

```

由于从 slab 分配失败,因此从紧急池中分配。

387 取得紧急池头链表的尾部。

388 获取保护池的锁。

389~393 如果池不空,则从链表取得一个缓冲区并减小 nr_emergency_bhs 计数器。

394 释放锁。

395~396 如果分配成功,则返回它。

399 如果失败,则表示内存不足,那么补充池的惟一方法便是完成高端内存 I/O 操作。就这样,请求从 tq_disk 开始,然后写数据到磁盘,接着释放进程中适当的页面。

401 出让处理器。

402 再次尝试从紧急池分配。

I. 5. 1. 3 函数: alloc_bounce_page() (mm/highmem.c)

这个函数本质上与 alloc_bounce_bh()一样。它首先尝试从伙伴分配器分配页面,如果失败则从紧急池中分配。

```

333 struct page * alloc_bounce_page (void)
334 {
335 struct list_head * tmp;
336 struct page * page;
337
338 page = alloc_page(GFP_NOHIGHIO);
339 if (page)
340 return page;
341
345 wakeup_bdfflush();

```

338~340 从伙伴分配器进行分配,如果成功则返回页面。

345 唤醒 bdfflush 清洗页面。

```

347 repeat_alloc:
351 tmp = &emergency_pages;
352 spin_lock_irq(&emergency_lock);
353 if (! list_empty(tmp)) {
354 page = list_entry(tmp->next, struct page, list);

```

```

355 list_del(tmp->next);
356 nr_emergency_pages -- ;
357 }
358 spin_unlock_irq(&emergency_lock);
359 if (page)
360 return page;
361
362 /* we need to wait I/O completion */
363 run_task_queue(&tq_disk);
364
365 yield();
366 goto repeat_alloc;
367 }

```

351 取得紧急池缓冲区头链表尾部。

352 获取保护池的锁。

353~357 如果池不空,则从链表取得页面并减小可用 nr_emergency_pages 的计数。

358 释放锁。

359~360 如果分配成功,则返回它。

363 运行 I/O 任务队列,尝试并补充紧急池。

365 让出处理器。

366 再次尝试从紧急池中分配。

I.5.2 用弹性缓冲区复制

I.5.2.1 函数: `bounce_end_io_write()` (`mm/highmem.c`)

在弹性缓冲区用于向设备写数据而完成 I/O 操作时,调用该函数。由于缓冲区用于从高端内存复制数据到设备,除了回收资源没有其他更多的任务待处理。

```

319 static void bounce_end_io_write (struct buffer_head * bh,
  int uptodate)
320 {
321 bounce_end_io(bh, uptodate);
322 }

```

I.5.2.2 函数: `bounce_end_io_read()` (`mm/highmem.c`)

在数据从设备读出且待复制到高端内存时,调用该函数。由于在中断中调用,所以要更加

谨慎。

```

324 static void bounce_end_io_read (struct buffer_head * bh,
325     int uptodate)
326 {
327     struct buffer_head * bh_orig =
328         (struct buffer_head *) (bh->b_private);
329     if (uptodate)
330         copy_to_high_bh_irq(bh_orig, bh);
331     bounce_end_io(bh, uptodate);

```

328~329 调用 copy_to_high_bh_irq()(见 I. 5. 2. 4)从弹性缓冲区复制数据并移至高端内存。
330 回收资源。

I. 5. 2. 3 函数: copy_from_high_bh() (mm/highmem.c)

这个函数用于从高端内存 buffer_head 复制数据到弹性缓冲区。

```

215 static inline void copy_from_high_bh (struct buffer_head * to,
216     struct buffer_head * from)
217 {
218     struct page * p_from;
219     char * vfrom;
220
221     p_from = from->b_page;
222
223     vfrom = kmap_atomic(p_from, KM_USER0);
224     memcpy(to->b_data, vfrom + bh_offset(from), to->b_size);
225     kunmap_atomic(vfrom, KM_USER0);
226 }

```

223 映射高端内存页面到低端内存。该执行路径由 IRQ 安全锁 io_request_lock 保护, 这样可以安全地调用 kmap_atomic()(见 I. 2. 1 小节)。

224 复制数据。

225 解除页面映射。

I. 5. 2. 4 函数: copy_to_high_bh_irq() (mm/highmem.c)

在设备完成写数据到弹性缓冲区后, 从中断中调用该函数。这个函数复制数据到高端内存。

```

228 static inline void copy_to_high_bh_irq (struct buffer_head * to,
229 struct buffer_head * from)
230 {
231     struct page * p_to;
232     char * vto;
233     unsigned long flags;
234
235     p_to = to->b_page;
236     __save_flags(flags);
237     __cli();
238     vto = kmap_atomic(p_to, KM_BOUNCE_READ);
239     memcpy(vto + bh_offset(to), from->b_data, to->b_size);
240     kunmap_atomic(vto, KM_BOUNCE_READ);
241     __restore_flags(flags);
242 }

```

236~237 保存标志位并禁止中断。

238 映射高端内存页面到低端内存。

239 复制数据。

240 解除页面映射。

241 恢复中断标志位。

I. 5.2.5 函数: bounce_end_io() (mm/highmem.c)

这个函数回收由弹性缓冲区使用的资源。如果紧急池耗尽，则添加资源给它。

```

244 static inline void bounce_end_io (struct buffer_head * bh,
245                                     int uptodate)
246     struct page * page;
247     struct buffer_head * bh_orig =
248         (struct buffer_head *) (bh->b_private);
249     unsigned long flags;
250     bh_orig->b_end_io(bh_orig, uptodate);
251
252     page = bh->b_page;
253
254     spin_lock_irqsave(&emergency_lock, flags);
255     if (nr_emergency_pages >= POOL_SIZE)
256         __free_page(page);

```

```

257 else {
258 /*
259 * We are abusing page->list to manage
260 * the highmem emergency pool:
261 */
262 list_add(&page->list, &emergency_pages);
263 nr_emergency_pages++;
264 }
265
266 if (nr_emergency_bhs >= POOL_SIZE) {
267 #ifdef HIGHMEM_DEBUG
268 /* Dont clobber the constructed slab cache */
269 init_waitqueue_head(&bh->b_wait);
270#endif
271 kmem_cache_free(bh_cachep, bh);
272 } else {
273 /*
274 * Ditto in the bh case, here we abuse b_inode_buffers:
275 */
276 list_add(&bh->b_inode_buffers, &emergency_bhs);
277 nr_emergency_bhs++;
278 }
279 spin_unlock_irqrestore(&emergency_lock, flags);
280 }

```

250 为原始 buffer_head 调用 I/O 完成的回调函数。

252 获取指向待释放的缓冲区页面的指针。

254 获取紧急池的锁。

255~256 如果页面池已满,则仅返回页面至伙伴分配器。

257~264 否则,添加页面到紧急池。

266~272 如果 buffer_head 池已满,则仅返回它至 slab 分配器。

272~278 否则,添加该 buffer_head 到紧急池。

279 释放锁。

I.6 紧急池

只有一个函数与紧急池相关,那就是其初始化函数。在系统启动时调用它,接着就删除它

的代码,因为不再需要它。

I. 6.1 函数: init_emergency_pool() (mm/highmem.c)

这个函数为紧急页面和紧急缓冲区头创建一个池。

```

282 static __init int init_emergency_pool(void)
283 {
284     struct sysinfo i;
285     si_meminfo(&i);
286     si_swapinfo(&i);
287
288     if (!i.totalhigh)
289         return 0;
290
291     spin_lock_irq(&emergency_lock);
292     while (nr_emergency_pages < POOL_SIZE) {
293         struct page * page = alloc_page(GFP_ATOMIC);
294         if (!page) {
295             printk("couldn't refill highmem emergency pages");
296             break;
297         }
298         list_add(&page->list, &emergency_pages);
299         nr_emergency_pages++;
300     }

```

288~289 如果没有可用的高端内存,则返回。

291 获取保护紧急池的锁。

292~300 从伙伴分配器分配 POOL_SIZE 个页面并添加它们到链接表中。然后用 nr_emergency_pages 记录紧急池中页面的数量。

```

301     while (nr_emergency_bhs < POOL_SIZE) {
302         struct buffer_head * bh =
303             kmalloc_cache_alloc(bh_cachep, SLAB_ATOMIC);
304         if (!bh) {
305             printk("couldn't refill highmem emergency bhs");
306             break;
307         }
308         list_add(&bh->b_inode_buffers, &emergency_bhs);
309         nr_emergency_bhs++;

```

```
309 }
310 spin_unlock_irq(&emergency_lock);
311 printk("allocated %d pages and %d bhs reserved for the
312 highmem bounces\n",
313 nr_emergency_pages, nr_emergency_bhs);
314
315 }
```

301~309 从 slab 分配器分配 POOL_SIZE 个页面并添加它们到由 b_inode_buffers 链接的链表中。而且以 nr_emergency_bhs 记录池中缓冲区头的数量。

310 释放用于保护池的锁。

314 返回成功。

附录 J

页面帧回收

目 录

| | |
|---|-----|
| J. 1 页面高速缓存操作 | 537 |
| J. 1. 1 向页面高速缓存添加页面 | 537 |
| J. 1. 1. 1 函数: add_to_page_cache() | 537 |
| J. 1. 1. 2 函数: add_to_page_cache_unique() | 538 |
| J. 1. 1. 3 函数: __add_to_page_cache() | 539 |
| J. 1. 1. 4 函数: add_page_to_inode_queue() | 539 |
| J. 1. 1. 5 函数: add_page_to_hash_queue() | 540 |
| J. 1. 2 从页面高速缓存删除页面 | 541 |
| J. 1. 2. 1 函数: remove_inode_page() | 541 |
| J. 1. 2. 2 函数: __remove_inode_page() | 541 |
| J. 1. 2. 3 函数: remove_page_from_inode_queue() | 541 |
| J. 1. 2. 4 函数: remove_page_from_hash_queue() | 542 |
| J. 1. 3 获取/释放页面高速缓存的页面 | 542 |
| J. 1. 3. 1 函数: page_cache_get() | 542 |
| J. 1. 3. 2 函数: page_cache_release() | 543 |
| J. 1. 4 搜索页面高速缓存 | 543 |
| J. 1. 4. 1 函数: find_get_page() | 543 |
| J. 1. 4. 2 函数: __find_get_page() | 543 |
| J. 1. 4. 3 函数: __find_page_nolock() | 544 |
| J. 1. 4. 4 函数: find_lock_page() | 544 |

| | |
|---|-----|
| J. 1. 4. 5 函数:__find_lock_page() | 545 |
| J. 1. 4. 6 函数:__find_lock_page_helper() | 545 |
| J. 2 LRU 链表操作 | 547 |
| J. 2. 1 向 LRU 链表添加页面 | 547 |
| J. 2. 1. 1 函数:lru_cache_add() | 547 |
| J. 2. 1. 2 函数:add_page_to_active_list() | 547 |
| J. 2. 1. 3 函数:add_page_to_inactive_list() | 548 |
| J. 2. 2 从 LRU 链表删除页面 | 548 |
| J. 2. 2. 1 函数:lru_cache_del() | 548 |
| J. 2. 2. 2 函数:__lru_cache_del() | 549 |
| J. 2. 2. 3 函数:del_page_from_active_list() | 549 |
| J. 2. 2. 4 函数:del_page_from_inactive_list() | 549 |
| J. 2. 3 激活页面 | 550 |
| J. 2. 3. 1 函数:mark_page_accessed() | 550 |
| J. 2. 3. 2 函数:activate_lock() | 550 |
| J. 2. 3. 3 函数:activate_page_nolock() | 550 |
| J. 3 重填充 inactive_list | 552 |
| J. 3. 1 函数:refill_inactive() | 552 |
| J. 4 从 LRU 链表回收页面 | 554 |
| J. 4. 1 函数:shrink_cache() | 554 |
| J. 5 收缩所有高速缓存 | 562 |
| J. 5. 1 函数:shrink_caches() | 562 |
| J. 5. 2 函数:try_to_free_pages() | 563 |
| J. 5. 3 函数:try_to_free_pages_zone() | 564 |
| J. 6 换出进程页面 | 566 |
| J. 6. 1 函数:swap_out() | 566 |
| J. 6. 2 函数:swap_out_mm() | 568 |
| J. 6. 3 函数:swap_out_vma() | 569 |
| J. 6. 4 函数:swap_out_pgd() | 570 |
| J. 6. 5 函数:swap_out_pmd() | 571 |
| J. 6. 6 函数:try_to_swap_out() | 573 |
| J. 7 页面交换守护程序 | 577 |
| J. 7. 1 初始化 kswapd | 577 |

| | |
|--------------------------------------|-----|
| J. 7. 1. 1 函数:kswapd_init() | 577 |
| J. 7. 2 kswapd 守护程序 | 577 |
| J. 7. 2. 1 函数:kswapd() | 577 |
| J. 7. 2. 2 函数:kswapd_can_sleep() | 579 |
| J. 7. 2. 3 函数:kswapd_can_sleep_pgd() | 579 |
| J. 7. 2. 4 函数:kswapd_balance() | 580 |
| J. 7. 2. 5 函数:kswapd_balance_pgd() | 580 |

J. 1 页面高速缓存操作

这节涉及如何在页面高速缓存和 LRU 链表中移动和添加页面,这两个操作紧密结合在一起。

J. 1. 1 向页面高速缓存添加页面

J. 1. 1. 1 函数: add_to_page_cache() (mm/filemap.c)

这个函数在调用 __add_to_page_cache() 以前获取保护页面高速缓存的锁,其中 __add_to_page_cache() 用于添加页面到页面 hash 表和索引节点队列,这样可以快速找到页面所属的文件。

```

667 void add_to_page_cache(struct page * page,
                           struct address_space * mapping,
                           unsigned long offset)

668 {
669     spin_lock(&pagecache_lock);
670     __add_to_page_cache(page, mapping,
671                         offset, page_hash(mapping, offset));
672     spin_unlock(&pagecache_lock);
673 }

```

669 获取保护页面 hash 表和索引节点队列的锁。

670 调用函数执行真正的工作。

671 释放保护页面 hash 表和索引节点队列的锁。

672 向页面高速缓存添加页面。page_hash()基于映射和文件中的偏移量对页面 hash 表进行 hash 查找。如果返回了页面，则会产生冲突，冲突的页面都通过 page→next_hash 和 page→pprev_hash 字段链接起来。

J. 1. 1. 2 函数：add_to_page_cache_unique() (mm/filemap.c)

在许多方面，这个函数和 add_to_page_cache()都非常相似。它们主要的区别是这个函数在添加页面至高速缓存以前，持有 pagecache_lock 自旋锁对页面高速缓存进行检查。在这种情况下，调用者会与另一个进程竞争向高速缓存添加页面，比如 add_to_swap_cache()（见 K. 2. 1. 1）。

```

675 int add_to_page_cache_unique(struct page * page,
676 struct address_space * mapping, unsigned long offset,
677 struct page ** hash)
678 {
679     int err;
680     struct page * alias;
681
682     spin_lock(&pagecache_lock);
683     alias = __find_page_nolock(mapping, offset, * hash);
684
685     err = 1;
686     if (! alias) {
687         __add_to_page_cache(page, mapping, offset, hash);
688     } else {
689         err = 0;
690     }
691     spin_unlock(&pagecache_lock);
692     if (! err)
693         lru_cache_add(page);
694     return err;
695 }
```

682 获取检查高速缓存的 pagecache_lock。

683 用 __find_page_nolock()（见 J. 1. 4. 3）检查页面是否已在高速缓存中。

686~689 如果页面不在高速缓存中，则用 __add_to_page_cache()（见 J. 1. 1. 3）添加页面。

691 释放 pagecache_lock。

692~693 如果页面不在页面高速缓存中，则调用 lru_cache_add()（见 J. 2. 1. 1）将它添加到 LRU 链表中。

694 如果该调用添加页面至高速缓存则返回 0, 而如果页面已在高速缓存中则返回 1。

J. 1. 1. 3 函数: `__add_to_page_cache()` (`mm/filemap.c`)

这个函数清除所有的页面标志位, 锁住页面, 增加页面的引用计数, 并添加页面至索引节点和哈希队列。

```

653 static inline void __add_to_page_cache(struct page * page,
654 struct address_space * mapping, unsigned long offset,
655 struct page ** hash)
656 {
657     unsigned long flags;
658
659     flags = page->flags & ~(1 << PG_uptodate |
660                           1 << PG_error | 1 << PG_dirty |
661                           1 << PG_referenced | 1 << PG_arch_1 |
662                           1 << PG_checked);
663     page->flags = flags | (1 << PG_locked);
664     page_cache_get(page);
665     page->index = offset;
666     add_page_to_inode_queue(mapping, page);
667     add_page_to_hash_queue(page, hash);
668 }
```

659 清除所有的页面标志位。

660 锁住页面。

661 如果页面被提前释放了, 则引用该页面。

662 更新索引, 这样就知道页面在文件中的偏移位置。

663 调用 `add_page_to_inode_queue()`(见 J. 1. 1. 4)向索引节点队列添加页面。这个操作用 `page->list` 链接页面到 `address_space` 中的 `clean_pages` 链表, 并使 `page->mapping` 指向同一个 `address_space`。

664 调用 `add_page_to_hash_queue()`(见 J. 1. 1. 5)添加页面至页面哈希队列。经 `hash` 处理后的页面由父函数的 `page_hash()` 返回。页面哈希队列使得页面高速缓存中的页面不需要在索引队列中进行线性查找。

J. 1. 1. 4 函数: `add_page_to_inode_queue()` (`mm/filemap.c`)

```

85 static inline void add_page_to_inode_queue(
86     struct address_space * mapping, struct page * page)
87 }
```

```

87 struct list_head * head = &mapping->clean_pages;
88
89 mapping->nrpages++;
90 list_add(&page->list, head);
91 page->mapping = mapping;
92 }

```

87 在调用这个函数时,清除页面,这样所需的链表就是 `mapping->clean`。

89 增加该映射的页面数量。

90 向 `clean` 链表添加页面。

91 设置 `page->mapping` 字段。

J. 1. 1. 5 函数: `add_page_to_hash_queue()` (`mm/filemap.c`)

这个函数向由 `p` 所指向的 hash 桶首部添加 `page`。请记住 `p` 是 `page_hash_table` 数组的一个元素。

```

71 static void add_page_to_hash_queue(struct page * page,
72                                     struct page ** p)
73 {
74     struct page * next = * p;
75     * p = page;
76     page->next_hash = next;
77     page->pnext_hash = p;
78     if (next)
79         next->pnext_hash = &page->next_hash;
80     if (page->buffers)
81         PAGE_BUG(page);
82     atomic_inc(&page_cache_size);
83 }

```

73 用 `next` 记录当前 hash 桶的首部。

75 更新 hash 桶首部为 `page`。

76 使 `page->next_hash` 指向 hash 桶原来的首部。

77 使 `page->pnext_hash` 指向 `page_hash_table` 数组中的元素。

78~79 这将使 `pnext_hash` 字段指向 hash 桶的首部,完成页面至链表的插入工作。

80~81 检查插入的页面没有相关联的缓冲区。

82 增加页面高速缓存的大小 `page_cache_size`。

J. 1. 2 从页面高速缓存删除页面

J. 1. 2. 1 函数: `remove_inode_page()` (`mm/filemap.c`)

```

130 void remove_inode_page(struct page * page)
131 {
132 if (!PageLocked(page))
133 PAGE_BUG(page);
134
135 spin_lock(&pagecache_lock);
136 __remove_inode_page(page);
137 spin_unlock(&pagecache_lock);
138 }

```

132~133 如果页面没有上锁,这是个 bug。

135 获取保护页面高速缓存的锁。

136 在持有页面高速缓存锁的情况下, `__remove_inode_page()`(见 J. 1. 2. 2)是位于高层的函数。

137 释放页面高速缓存锁。

J. 1. 2. 2 函数: `__remove_inode_page()` (`mm/filemap.c`)

这是从页面高速缓存移除页面的高层函数,调用者必须持有 `pagecache_lock` 自旋锁。而不持有该锁的调用者应当调用 `remove_inode_page()`。

```

124 void __remove_inode_page(struct page * page)
125 {
126 remove_page_from_inode_queue(page);
127 remove_page_from_hash_queue(page);
128

```

126 `remove_page_from_inode_queue()`(见 J. 1. 2. 3)从 `address_space` 的 `page→mapping` 移除页面。

127 `remove_page_from_hash_queue()`从 `page_hash_table` 的 `hash` 表移除页面。

J. 1. 2. 3 函数: `remove_page_from_inode_queue()` (`mm/filemap.c`)

```

94 static inline void remove_page_from_inode_queue(struct page * page)
95 {
96 struct address_space * mapping = page->mapping;

```

```

97
98 if (mapping->a_ops->removepage)
99 mapping->a_ops->removepage(page);
100 list_del(&page->list);
101 page->mapping = NULL;
102 wmb();
103 mapping->nr_pages--;
104 }

```

96 获取该 page 相关的 address_space。

98~99 如果文件系统相关的 removepage() 函数可用，则调用它。

100 删除页面，不论它在 mapping 中属于任何一个链表，比如大多数情况下的 clean_page 链表或较少情况下的 dirty_page 链表。

101 设置 page->mapping 为 NULL，因为它已没有任何后援的 address_space。

103 减少 mapping 中的页面数量。

J. 1. 2. 4 函数：remove_page_from_hash_queue() (mm/filemap.c)

```

107 static inline void remove_page_from_hash_queue(struct page * page)
108 {
109 struct page * next = page->next_hash;
110 struct page ** pprev = page->pprev_hash;
111
112 if (next)
113 next->pprev_hash = pprev;
114 *pprev = next;
115 page->pprev_hash = NULL;
116 atomic_dec(&page_cache_size);
117 }

```

109 获取被移除 page 的下一个页面。

110 获取被移除 page 的前一个 pprev 页面。在函数完成时，pprev 将链接下一个。

112 如果不是链表尾部，则更新 next->pprev_hash 指向 pprev。

114 同理，使 pprev 向前指向 next。此时 page 已经解除链接。

116 减小页面高速缓存的大小。

J. 1.3 获取/释放页面高速缓存的页面

J. 1.3.1 函数: `page_cache_get()` (`include/linux/pagemap.h`)

31 `#define page_cache_get(x) get_page(x)`

31 简单地调用 `get_page()`, 它用 `atomic_inc()`增加页面引用计数。

J. 1.3.2 函数: `page_cache_release()` (`include/linux/pagemap.h`)

32 `#define page_cache_release(x) __free_page(x)`

32 调用 `free_page()`来减小页面计数。如果计数减小到 0, 则释放页面。

J. 1.4 搜索页面高速缓存

J. 1.4.1 函数: `find_get_page()` (`include/linux/pagemap.h`)

这是在页面高速缓存中查找页面的位于高层的宏。它简单地查找页面哈希。

75 `#define find_get_page(mapping, index) \`
76 `__find_get_page(mapping, index, page_hash(mapping, index))`

76 `page_hash()`基于 `address_space` 和偏移量, 并在 `page_hash_table` 中定位项。

J. 1.4.2 函数: `__find_get_page()` (`mm/filemap.c`)

这个函数用于根据 `page_hash_table` 中作为起始点的给定项查找页面结构。

```
931 struct page * __find_get_page(struct address_space * mapping,  
932 unsigned long offset, struct page ** hash)  
933 {  
934 struct page * page;  
935  
936 /*  
937 * We scan the hash list read-only. Addition to and removal from  
938 * the hash-list needs a held write-lock.  
939 */  
940 spin_lock(&pagecache_lock);  
941 page = __find_page_nolock(mapping, offset, * hash);  
942 if (page)
```



```

943 page_cache_get(page);
944 spin_unlock(&pagecache_lock);
945 return page;
946 }

```

940 获取页面高速缓存只读锁。

941 调用页面高速缓存遍历函数,前提假设是已持有锁。

942~943 如果找到页面,则用 page_cache_get() (见 J. 1.3.1) 获取对页面的引用,这样就不会提前释放页面。

944 释放页面高速缓存锁。

945 返回页面,或在没有找到页面的情况下返回 NULL。

J. 1.4.3 函数: __find_page_nolock() (mm/filemap.c)

这个函数遍历哈希冲突链表,找到特定 address_space 和 offset 的页面。

```

443 static inline struct page * __find_page_nolock(
        struct address_space * mapping,
        unsigned long offset,
        struct page * page)
444 {
445     goto inside;
446
447     for (;;) {
448         page = page->next_hash;
449     inside:
450         if (!page)
451             goto not_found;
452         if (page->mapping != mapping)
453             continue;
454         if (page->index == offset)
455             break;
456     }
457
458 not_found:
459     return page;
460 }

```

445 从检查链表的第一个页面开始。

450~451 如果页面为 NULL,则表示没有找到正确的页面,那么返回 NULL。

452 如果 address_space 不匹配,则转到冲突链表中的下一个页面。

454 如果 offset 匹配, 则返回它, 否则继续移动。

448 转到哈希链表中下一个页面。

459 返回找到的页面, 否则返回 NULL。

J.1.4.4 函数: `find_lock_page()` (`include/linux/pagemap.h`)

这是用于在页面高速缓存中查找页面并在上锁状态下返回页面的高层函数。

```
84 #define find_lock_page(mapping, index) \
85 __find_lock_page(mapping, index, page_hash(mapping, index))
```

85 在用 `page_hash()` 获取页面所在的 hash 桶以后, 调用内核函数 `__find_lock_page()`。

J.1.4.5 函数: `__find_lock_page()` (`mm/filemap.c`)

在调用内核函数 `__find_lock_page_helper()` 定位并锁住页面以后, 这个函数获取 `pagecache_lock` 自旋锁。

```
1005 struct page * __find_lock_page (struct address_space * mapping,
1006 unsigned long offset, struct page ** hash)
1007 {
1008 struct page * page;
1009
1010 spin_lock(&pagecache_lock);
1011 page = __find_lock_page_helper(mapping, offset, * hash);
1012 spin_unlock(&pagecache_lock);
1013 return page;
1014 }
```

1010 获取 `pagecache_lock` 自旋锁。

1011 调用 `__find_lock_page_helper()` 搜索页面高速缓存, 并在找到页面后锁住页面。

1012 释放 `pagecache_lock` 自旋锁。

1013 如果找到页面, 则返回上锁状态的页面。如果没有找到, 则返回 NULL。

J.1.4.6 函数: `__find_lock_page_helper()` (`mm/filemap.c`)

这个函数调用 `__find_page_nolock()` 在页面高速缓存中定位页面。如果找到, 则锁住页面并返回给调用者。

```
972 static struct page * __find_lock_page_helper(
973         struct address_space * mapping,
974         unsigned long offset, struct page * hash)
975 struct page * page;
```

```

976
977 /*
978 * We scan the hash list read-only. Addition to and removal
979 * from the hash-list needs a held write-lock.
980 */
981 repeat:
982 page = __find_page_nolock(mapping, offset, hash);
983 if (page) {
984 page_cache_get(page);
985 if (TryLockPage(page)) {
986 spin_unlock(&pagecache_lock);
987 lock_page(page);
988 spin_lock(&pagecache_lock);
989
990 /* Has the page been re-allocated while we slept? */
991 if (page->mapping != mapping || page->index != offset) {
992 UnlockPage(page);
993 page_cache_release(page);
994 goto repeat;
995 }
996 }
997 }
998 return page;
999 }

```

982 调用 __find_page_nolock() (见 J. 1. 4. 3) 在页面高速缓存中定位页面。

983~984 如果找到页面, 则引用它。

985 用 TryLockPage() 尝试并锁住页面。这个宏是对 test_and_set_bit() 的封装, 后者用于尝试对 page->flags 中的 PG_locked 位进行设置。

986~988 如果上锁失败, 则释放 pagecache_lock 自旋锁并调用 lock_page() (见 B. 2. 1. 1) 锁住页面。这个函数会一直睡眠直至获取页面锁。在页面上锁后, 它再次获取 pagecache_lock 自旋锁。

991 如果 mapping 和 index 都不再匹配, 这意味着在睡眠的时候页面已经回收。在再次搜索页面高速缓存以前, 为页面解锁, 并释放对页面的引用。

998 返回上锁状态的页面, 如果在页面高速缓存中没有找到页面则返回 NULL。

J. 2 LRU 链表操作

J. 2. 1 向 LRU 链表添加页面

J. 2. 1. 1 函数: lru_cache_add() (mm/swap.c)

这个函数向 LRU inactive_list 添加页面。

```
58 void lru_cache_add(struct page * page)
59 {
60 if (!PageLRU(page)) {
61 spin_lock(&pagemap_lru_lock);
62 if (!TestSetPageLRU(page))
63 add_page_to_inactive_list(page);
64 spin_unlock(&pagemap_lru_lock);
65 }
66 }
```

60 如果页面还不是 LRU 链表中的一部分,则添加它。

61 获取 LRU 锁。

62~63 测试并设置 LRU 位。如果已经清除,则调用 add_page_to_inactive_list()。

64 释放 LRU 锁。

J. 2. 1. 2 函数: add_page_to_active_list() (include/linux/swap.h)

这个函数向 active_list 添加页面。

```
178 #define add_page_to_active_list(page) \
179 do { \
180 DEBUG_LRU_PAGE(page); \
181 SetPageActive(page); \
182 list_add(&(page)->lru, &active_list); \
183 nr_active_pages ++; \
184 } while (0)
```

180 如果页面已经在 LRU 链表中或被标记为活动,DEBUG_LRU_PAGE()宏将调用 BUG()。

181 更新页面的标志位,表示它为活动的。

182 向 active_list 添加页面。

183 更新 active_list 中的页面计数。

J. 2. 1. 3 函数: add_page_to_inactive_list() (include/linux/swap.h)

向 inactive_list 添加页面。

```
186 #define add_page_to_inactive_list(page) \
187 do { \
188     DEBUG_LRU_PAGE(page); \
189     list_add(&(page)->lru, &inactive_list); \
190     nr_inactive_pages++; \
191 } while (0)
```

188 如果页面已经在 LRU 链表中或被标记为活动, 则 DEBUG_LRU_PAGE() 宏将调用 BUG()。

189 向 inactive_list 添加页面。

190 更新链表中非活动页面的计数。

J. 2. 2 从 LRU 链表删除页面

J. 2. 2. 1 函数: lru_cache_del() (mm/swap.c)

这个函数在调用 __lru_cache_del() 前获取保护 LRU 链表的锁。

```
90 void lru_cache_del(struct page * page)
91 {
92     spin_lock(&pagemap_lru_lock);
93     __lru_cache_del(page);
94     spin_unlock(&pagemap_lru_lock);
95 }
```

92 获取 LRU 锁。

93 __lru_cache_del() 执行从 LRU 链表移除页面的真正工作。

94 释放 LRU 锁。

J. 2. 2. 2 函数: __lru_cache_del() (mm/swap.c)

这个函数用于选择从 LRU 链表移除页面的函数。

```
75 void __lru_cache_del(struct page * page)
76 {
77     if (TestClearPageLRU(page)) {
78         if (PageActive(page)) {
```

```

79 del_page_from_active_list(page);
80 } else {
81 del_page_from_inactive_list(page);
82 }
83 }
84 }

```

77 测试并清除表示页面在 LRU 链表中的标志位。

78~82 如果页面在 LRU 链表中，则选择合适的移除函数。

78~79 如果是活动页面，则调用 `del_page_from_active_list()`，否则调用 `del_page_from_inactive_list()` 从非活动链表删除页面。

J.2.2.3 函数：`del_page_from_active_list()` (include/linux/swap.h)

这个函数用于从 `active_list` 删除页面。

```

193 #define del_page_from_active_list(page) \
194 do { \
195 list_del(&(page)->lru); \
196 ClearPageActive(page); \
197 nr_active_pages--; \
198 } while (0)

```

195 从链表删除页面。

196 清除标志位，表示它是 `active_list` 的一部分。该标志位表示它是已由 `__lru_cache_del()` 清除的 LRU 链表的一部分。

197 更新 `active_list` 中页面的计数。

J.2.2.4 函数：`del_page_from_inactive_list()` (include/linux/swap.h)

```

200 #define del_page_from_inactive_list(page) \
201 do { \
202 list_del(&(page)->lru); \
203 nr_inactive_pages--; \
204 } while (0)

```

202 从 LRU 链表移除页面。

203 更新 `inactive_list` 页面计数。

J. 2.3 激活页面

J. 2.3.1 函数: `mark_page_accessed()` (`mm/filemap.c`)

这个函数标记页面已被引用过。如果页面依旧在 `active_list` 中或已清除引用标志位, 则设置引用标志位。如果页面在 `inactive_list` 中且已设置引用标志位, 则调用 `activate_page()` 移动页面至 `active_list` 首部。

```
1332 void mark_page_accessed(struct page * page)
1333 {
1334 if (! PageActive(page) && PageReferenced(page)) {
1335 activate_page(page);
1336 ClearPageReferenced(page);
1337 } else
1338 SetPageReferenced(page);
1339 }
```

1334~1337 如果页面在 `inactive_list` 中(`! PageActive()`)且最近被引用了(`PageReferenced()`), 则调用 `activate_page()` 移动页面至 `active_list`。

1338 否则, 标记页面为被引用。

J. 2.3.2 函数: `activate_lock()` (`mm/swap.c`)

在调用 `activate_page_nolock()` 之前获取 LRU 锁, 其中 `activate_page_nolock()` 用于从 `inactive_list` 移动页面至 `active_list`。

```
47 void activate_page(struct page * page)
48 {
49 spin_lock(&pagemap_lru_lock);
50 activate_page_nolock(page);
51 spin_unlock(&pagemap_lru_lock);
52 }
```

49 获取 LRU 锁。

50 调用主要的工作函数。

51 释放 LRU 锁。

J. 2.3.3 函数: `activate_page_nolock()` (`mm/swap.c`)

这个函数从 `inactive_list` 将页面移至 `active_list`。

```
39 static inline void activate_page_nolock(struct page * page)
```

```

40 {
41 if (PageLRU(page) && ! PageActive(page)) {
42 del_page_from_inactive_list(page);
43 add_page_to_active_list(page);
44 }
45 }

```

41 确保页面在 LRU 链表中而不在 active_list 中。

42~43 从 inactive_list 删 除页面并添加到 active_list 中。

J. 3 重填充 inactive_list

这节涵盖了有多少页面从活动链表移向非活动链表。

J. 3. 1 函数: refill_inactive() (mm/vmscan.c)

这个函数从 active_list 移动 nr_pages 个页面到 inactive_list。参数 nr_pages 由 shrink_caches() 进行计算, 它用于保持活动链表的大小为页面高速缓存大小的三分之二。

```

533 static void refill_inactive(int nr_pages)
534 {
535 struct list_head * entry;
536
537 spin_lock(&pagemap_lru_lock);
538 entry = active_list.prev;
539 while (nr_pages && entry != &active_list) {
540 struct page * page;
541
542 page = list_entry(entry, struct page, lru);
543 entry = entry->prev;
544 if (PageTestandClearReferenced(page)) {
545 list_del(&page->lru);
546 list_add(&page->lru, &active_list);
547 continue;
548 }
549
550 nr_pages -- ;
551

```



```

552 del_page_from_active_list(page);
553 add_page_to_inactive_list(page);
554 SetPageReferenced(page);
555 }
556 spin_unlock(&pagemap_lru_lock);
557 }

```

537 获取保护 LRU 链表的锁。

538 取 active_list 中最后一项。

539~555 持续移动页面直至达到 nr_pages 个页面或 active_list 已为空。

542 获取项的 struct page。

544~548 测试并清除引用标志位。如果页面已被引用，则移回至 active_list 的首部。

550~553 从 active_list 移动页面至 inactive_list。

554 标记页面被引用，这样在再次引用的情况下，将把它移回 active_list 而不需要第 2 次引用。

556 释放保护 LRU 链表的锁。

J. 4 从 LRU 链表回收页面

这节涵盖了页面在被选定进行换出操作时如何被回收。

J. 4. 1 函数：shrink_cache() (mm/vmscan.c)

```

338 static int shrink_cache(int nr_pages, zone_t * classzone,
                           unsigned int gfp_mask, int priority)
339 {
340 struct list_head * entry;
341 int max_scan = nr_inactive_pages / priority;
342 int max_mapped = min((nr_pages << (10 - priority)),
                       max_scan / 10);
343
344 spin_lock(&pagemap_lru_lock);
345 while ( --max_scan >= 0 &&
           (entry = inactive_list.prev) != &inactive_list) {

```

338 参数如下所示：

- nr_pages 是换出页面的数量
- classzone 是我们所关心的页面换出管理区。不属于该管理区的页面均跳过。
- gfp_mask 决定采取何种操作,比如可能会执行文件系统操作。
- priority 是函数的优先级,起始于 DEF_PRIORITY(6),逐渐减小到最高优先级 1。

341 扫描的最大页面数量是 active_list 中按优先级区分的页面数量。在最低优先级,将扫描链表的六分之一。在最高优先级,将扫描整个链表。

342 进程映射的页面所允许的最大数量不是最大扫描值的十分之一,就是 $nr_pages * 2^{10 - priority}$ 。如果找到如此数量的页面,则整个进程将换出。

344 锁住 LRU 链表。

345 持续扫描直至扫描了 max_scan 个页面或 inactive_list 已为空。

```

346 struct page * page;
347
348 if (unlikely(current->need_resched)) {
349     spin_unlock(&pagemap_lru_lock);
350     __set_current_state(TASK_RUNNING);
351     schedule();
352     spin_lock(&pagemap_lru_lock);
353     continue;
354 }
355

```

348~354 如果配额用完则重新调度。

349 即将睡眠,则释放 LRU 锁。

350 表示依旧在运行中。

351 调用 schedule(),这样其他的进程可以通过上下文切换进来。

352 重新获取 LRU 锁。

353 重新循环并再次获取 inactive_list 的项。在睡眠期间,另一个进程有可能改变链表上的项,这也是为什么另一个项必须持有自旋锁的原因。

```

356 page = list_entry(entry, struct page, lru);
357
358 BUG_ON(!PageLRU(page));
359 BUG_ON(PageActive(page));
360
361 list_del(entry);
362 list_add(entry, &inactive_list);
363

```

```

364 /*
365 * Zero page counts can happen because we unlink the pages
366 * _after_ decrementing the usage count..
367 */
368 if (unlikely(! page_count(page)))
369 continue;
370
371 if (! memclass(page_zone(page), classzone))
372 continue;
373
374 /* Racy check to avoid trylocking when not worthwhile */
375 if (! page->buffers && (page_count(page) != 1 || ! page->mapping))
376 goto page_mapped;

```

356 在 LRU 链表中获取项的 struct page。

358~359 如果页面属于 active_list 或目前标记为活动的,这是个 bug。

361~362 移动页面至 inactive_list 首部,在执行这样的操作后,如果页面没有释放,也同样可以知道在不久会简单地检查它。

368~369 如果页面计数已经达到 0,则跳过它。在 __free_pages() 中,在调用 __free_pages_ok() 释放页面以前,页面计数将由 put_page_testzero() 减小。这也即页面在释放以前其在 LRU 链表中的计数为 0。一种特殊情形是在 __free_pages_ok() 开始时捕获它。

371~372 如果页面属于我们并不关心的管理区,则跳过该页面。

375~376 如果页面由进程映射,则跳转到 page_mapped,减小 max_mapped 并检查下一个页面。如果 max_mapped 达到 0,则换出进程页面。

```

382 if (unlikely(TryLockPage(page))) {
383 if (PageLaunder(page) && (gfp_mask & __GFP_FS)) {
384 page_cache_get(page);
385 spin_unlock(&pagemap_lru_lock);
386 wait_on_page(page);
387 page_cache_release(page);
388 spin_lock(&pagemap_lru_lock);
389 }
390 continue;
391 }

```

在这块,页面上锁,且设置清洗位。在这种情况下,这是页面第二次被发现为脏页面。第一次发生在 I/O 调度并将页面放回链表时。在这里将等待 I/O 完成然后释放页面。

382~383 如果不能锁住页面,则设置 PG_launder 位,而 GFP 标志位允许调用者执行 FS

操作。接着……

- 384 引用该页面,这样在睡眠时它不会消失。
- 385 释放 LRU 锁。
- 386 等待直至 I/O 完成。
- 387 释放页面的引用。如果达到 0,则释放页面。
- 388 重新获取 LRU 锁。
- 390 转到下一个页面。

392

```

393 if (PageDirty(page) &&
394     is_page_cache_freeable(page) &&
395     page->mapping) {
396     /* It is not critical here to write it only if
397     * the page is unmapped because any direct writer
398     * like O_DIRECT would set the PG_dirty bitflag
399     * on the physical page after having successfully
400     * pinned it and after the I/O to the page is finished,
401     * so the direct writes to the page cannot get lost.
402     */
403     int (*writepage)(struct page *);
404     writepage = page->mapping->a_ops->writepage;
405     if ((gfp_mask & __GFP_FS) && writepage) {
406         ClearPageDirty(page);
407         SetPageLander(page);
408         page_cache_get(page);
409         spin_unlock(&pagemap_lru_lock);
410
411         writepage(page);
412         page_cache_release(page);
413
414         spin_lock(&pagemap_lru_lock);
415         continue;
416     }
417 }
```

这块代码在页面为脏,不被任何进程所映射,没有缓冲区,且有后援文件或后援设备时进行操作。页面将被清除且在 I/O 完成时由前一块代码进行回收。

393 PageDirty()检查 PG_dirty 位。在页面不被任何进程映射且没有缓冲区时, is_page_cache_freeable()将返回真。

404 为映射或设备获取指向必需的 writepage()的指针。

405~416 这块代码只在 writepage()可用且 GFP 标志位允许文件操作的情况下执行。

406~407 清除脏位并标记页面已被清洗了。

408 引用该页面,这样页面不会被异常释放。

409 为 LRU 链表解锁。

411 调用文件系统相关的 writepage()函数,它来自属于 page->mapping 的 address_space_operations。

412 释放页面的引用。

414~415 获取 LRU 链表的锁并转到下一个页面。

```

424 if (page->buffers) {
425     spin_unlock(&pagemap_lru_lock);
426
427     /* avoid to free a locked page */
428     page_cache_get(page);
429
430     if (try_to_release_page(page, gfp_mask)) {
431         if (!page->mapping) {
432             spin_lock(&pagemap_lru_lock);
433             UnlockPage(page);
434             __lru_cache_del(page);
435
436             /* effectively free the page here */
437             page_cache_release(page);
438
439             if (nr_pages)
440                 continue;
441             break;
442         } else {
443             page_cache_release(page);
444
445             if (--nr_pages)
446                 continue;
447             break;
448         }
449     }
450     spin_lock(&pagemap_lru_lock);
451 }
452 } else {
453     /* failed to drop the buffers so stop here */
454     UnlockPage(page);

```

```

461 page_cache_release(page);
462
463 spin_lock(&pagemap_lru_lock);
464 continue;
465 }
466 }

```

必须释放与页面相关联的缓冲区。

425 由于已睡眠,因此释放 LRU 锁。

428 引用该页面。

430 调用 `try_to_release_page()` 来尝试释放与页面相关联的缓冲区。如果成功则返回 1。

431~447 这种情况是交换高速缓存中的匿名页面已清除并移除它的缓冲区。由于它在交换高速缓存中,且由 `add_to_swap_cache()` 放置到 LRU 链表中,因此现在从 LRU 链表移除它并释放对该页面的引用。在 `swap_writepage()` 中,将调用 `remove_exclusive_swap_page()` 在没有更多的进程映射该页面时从交换高速缓存删除页面。如果存在后援交换文件,则这块代码将在缓冲区写出后释放该页面。

438~440 获取 LRU 链的锁,为页面解锁,从页面高速缓存删除并释放它。

445~446 更新 `nr_pages`,表示页面已被释放,并转到下一个页面。

447 如果 `nr_pages` 减小到 0,则表示工作完成,跳出循环。

449~456 如果页面有相关联的映射,则释放页面的引用并重新获取 LRU 锁。接下来会执行更多的操作,如在 499 行的从页面高速缓存移除页面。

459~464 如果无法释放缓冲区,则为页面解锁,释放页面的引用,重新获取 LRU 锁并转到下一个页面。

```

468 spin_lock(&pagecache_lock);
469
470 /*
471 * this is the non-racy check for busy page.
472 */
473 if (!page->mapping || !is_page_cache_freeable(page)) {
474 spin_unlock(&pagecache_lock);
475 UnlockPage(page);
476 page_mapped:
477 if (--max_mapped >= 0)
478 continue;
479
484 spin_unlock(&pagemap_lru_lock);
485 swap_out(priority, gfp_mask, classzone);

```

```
486 return nr_pages;
487 }
```

468 从这里开始,将检查交换高速缓存中的页面,此时必须持有 pagecache_lock 保护锁。

473~487 没有缓冲区的匿名页由进程所映射。

474~475 释放页面高速缓存锁和页面。

477~478 减小 max_mapped。如果没有达到 0,则转到下一个页面。

484~485 在页面高速缓存中找到过多的映射页面。释放 LRU 锁,并调用 swap_out() 换出整个进程。

```
493 if (PageDirty(page)) {
494     spin_unlock(&pagecache_lock);
495     UnlockPage(page);
496     continue;
497 }
```

493~497 虽然页面没有被引用,但也有可能在 PTE 中的脏位被设置的情况下最后一个进程释放它时使页面为脏。它依旧在页面高速缓存中,并在后期进行清洗。在被清除以后,可以安全地删除页面。

```
498
499 /* point of no return */
500 if (likely(!PageSwapCache(page))) {
501     __remove_inode_page(page);
502     spin_unlock(&pagecache_lock);
503 } else {
504     swap_entry_t swap;
505     swap.val = page->index;
506     __delete_from_swap_cache(page);
507     spin_unlock(&pagecache_lock);
508     swap_free(swap);
509 }
510
511 __lru_cache_del(page);
512 UnlockPage(page);
513
514 /* effectively free the page here */
515 page_cache_release(page);
516
517 if ( --nr_pages)
```



```
518 continue;
519 break;
520 }
```

500~503 如果页面不属于交换高速缓存,而是索引节点队列的一部分,则移除它。

504~508 由于页面不再有引用,因此从交换高速缓存移除它。

511 从页面高速缓存移除它。

512 为页面解锁。

515 释放页面。

517~518 减小 nr_page,如果没有达到 0,则转到下一个页面。

519 如果达到 0,则表示函数的工作已完成。

```
521 spin_unlock(&pagemap_lru_lock);
522
523 return nr_pages;
524 }
```

521~524 退出函数。释放 LRU 锁并返回待释放的页面数。

J. 5 收缩所有高速缓存

J. 5. 1 函数: `shrink_caches()` (`mm/vmscan.c`)

函数调用图如图 10. 4 所示。

```
560 static int shrink_caches(zone_t * classzone, int priority,
                           unsigned int gfp_mask, int nr_pages)
561 {
562     int chunk_size = nr_pages;
563     unsigned long ratio;
564
565     nr_pages -= kmem_cache_reap(gfp_mask);
566     if (nr_pages <= 0)
567         return 0;
568
569     nr_pages = chunk_size;
570     /* try to keep the active list 2/3 of the size of the cache */
571     ratio = (unsigned long) nr_pages *
```



```

nr_active_pages / ((nr_inactive_pages + 1) * 2);
572 refill_inactive(ratio);
573
574 nr_pages = shrink_cache(nr_pages, classzone, gfp_mask, priority);
575 if (nr_pages <= 0)
576 return 0;
577
578 shrink_dcache_memory(priority, gfp_mask);
579 shrink_icache_memory(priority, gfp_mask);
580 #ifdef CONFIG_QUOTA
581 shrink_dqcache_memory(DEF_PRIORITY, gfp_mask);
582#endif
583
584 return nr_pages;
585 }

```

560 参数如下所示：

- classzone 是被释放页面的管理区来源。
- priority 决定释放页面的工作量。
- gfp_mask 决定执行何种操作。
- nr_pages 是待释放的页面数量。

565~567 请求 slab 分配器用 kmem_cache_reap() (见 H. 1. 5. 1) 释放一部分页面。如果释放了足够多的页面，则函数返回。否则，将从其他的高速缓存释放 nr_pages 个页面。

571~572 通过调用 refill_inactive() (见 J. 3. 1 小节) 从 active_list 移动页面至 inactive_list。移动的页面数量取决于需要被释放的页面数量，且 active_list 的大小必须为页面高速缓存大小的三分之二。

574~575 收缩页面高速缓存。如果释放了足够的页面，则返回。

578~582 收缩 dcache, icache 和 dqcache。它们本身虽然都是很小的对象，但合起来的效应可以释放掉大量的磁盘缓冲区。

584 返回待释放的页面数量。

J. 5. 2 函数：try_to_free_pages() (mm/vmscan.c)

这个函数循环遍历所有的 pgdat，并尝试为每个 pgdat 平衡最佳的分配管理区 (通常为 ZONE_NORMAL)。这个函数只在 buffer.c:free_more_memory() 这一个地方进行调用，即在缓冲区管理器创建新缓冲区失败或增长一个已有缓冲区时调用。它将 GFP_NOIO 作为 gfp_mask 来调用 free_pages()。

第 1 个管理区的 `pg_data_t->node_zonelists` 的结果为释放掉了页面,那么缓冲区便可以增长。该数组是从管理区进行分配的最佳顺序,一般从 `ZONE_NORMAL` 开始,它是缓冲区管理器所需的。在 NUMA 体系结构中,如果内存簇用于 I/O 设备,那么一部分节点会将 `ZONE_DMA` 作为最佳的管理区。UML 也只会使用这个管理区。由于缓冲区管理区限制在它所使用的管理区中,那么就不能平衡其他的管理区。

```

607 int try_to_free_pages(unsigned int gfp_mask)
608 {
609 pg_data_t * pgdat;
610 zonelist_t * zonelist;
611 unsigned long pf_free_pages;
612 int error = 0;
613
614 pf_free_pages = current->flags & PF_FREE_PAGES;
615 current->flags &= ~PF_FREE_PAGES;
616
617 for_each_pgdat(pgdat) {
618 zonelist = pgdat->node_zonelists +
619 (gfp_mask & GFP_ZONEMASK);
620 error |= try_to_free_pages_zone(
621 zonelist->zones[0], gfp_mask);
622 }
623
624 current->flags |= pf_free_pages;
625 return error;
626 }
```

614~615 如果设置了 `PF_FREE_PAGES` 则清除它,这样由进程释放的页面将返回给全局池,而不是保留在进程本身。

617~620 循环遍历所有的节点并为每个节点的最佳管理区调用 `free_pages()`。

618 这个函数只在 `GFP_NOIO` 为参数时进行调用。在与 `GFP_ZONEMASK` 进行与操作后,通常返回 0。

622~623 存储进程标志位并返回结果。

J. 5.3 函数: `try_to_free_pages_zone()` (`mm/vmscan.c`)

这个函数从被请求的管理区释放 `SWAP_CLUSTER_MAX` 个页面。这个函数由 `kswapd` 所使用,且为伙伴分配器直接回收路径上的项。

```

587 int try_to_free_pages_zone(zone_t * classzone,
                               unsigned int gfp_mask)
588 {
589 int priority = DEF_PRIORITY;
590 int nr_pages = SWAP_CLUSTER_MAX;
591
592 gfp_mask = pf_gfp_mask(gfp_mask);
593 do {
594 nr_pages = shrink_caches(classzone, priority,
                           gfp_mask, nr_pages);
595 if (nr_pages <= 0)
596 return 1;
597 } while ( -- priority);
598
599 /*
600 * Hmm.. Cache shrink failed - time to kill something?
601 * Mhwahahahaha! This is the part I really like. Giggle.
602 */
603 out_of_memory();
604 return 0;
605 }

```

589 从最低优先级开始。最低优先级一般定义为 6。

590 尝试并释放 SWAP_CLUSTER_MAX 个页面。通常定义为 32。

592 pf_gfp_mask() 在当前进程标志位中检查 PF_NOIO 标志位。如果没有 I/O 执行，则要确保在 GFP 掩码中没有不兼容的标志位。

593~597 从最低优先级开始并在每次执行后增加，这里调用 shrink_caches() 直至释放了 nr_pages 个页面。

595~596 如果释放了足够多的页面，则返回，表示工作完成。

603 如果在最高优先级也无法释放足够多的页面（最坏的情况是扫描整个 inactive_list），则检查是否发生了内存溢出。如果是，则选择一个进程销毁。

604 返回，表示释放足够多的页面时操作失败。

J. 6 换出进程页面

这节涵盖了在 LRU 链表中找到过量的进程映射页面的路径。这条路径将扫描整个进程

并回收映射的页面。

J. 6.1 函数: swap_out() (mm/vmscan.c)

调用图如图 10.5 所示。这个函数线性搜索每个进程的页表，并尝试换出 SWAP_CLUSTER_MAX 个页面。进程起始于 swap_mm，而起始地址为 mm->swap_address。

```

296 static int swap_out (unsigned int priority, unsigned int gfp_mask,
297                      zone_t * classzone)
298 {
299     int counter, nr_pages = SWAP_CLUSTER_MAX;
300     struct mm_struct * mm;
301     counter = mmlist_nr;
302     do {
303         if (unlikely(current->need_resched)) {
304             __set_current_state(TASK_RUNNING);
305             schedule();
306         }
307         spin_lock(&mmlist_lock);
308         mm = swap_mm;
309         while (mm->swap_address == TASK_SIZE || mm == &init_mm) {
310             mm->swap_address = 0;
311             mm = list_entry(mm->mmlist.next,
312                             struct mm_struct, mmlist);
313             if (mm == swap_mm)
314                 goto empty;
315             swap_mm = mm;
316         }
317         /* Make sure the mm doesn't disappear
318          when we drop the lock.. */
319         atomic_inc(&mm->mm_users);
320         spin_unlock(&mmlist_lock);
321
322         nr_pages = swap_out_mm(mm, nr_pages, &counter, classzone);
323
324         mmput(mm);

```

```

325
326 if (! nr_pages)
327 return 1;
328 } while ( -- counter >= 0);
329
330 return 0;
331
332 empty:
333 spin_unlock(&mm_list_lock);
334 return 0;
335 }

```

301 设置计数器,这样只会扫描进程链表一次。

303~306 如果使用了配额阻止进程贪婪占据 CPU,则重新调度。

308 获取保护 mm 链表的锁。

309 从 swap_mm 开始。有意思的是这里并不检查以确保它是否合法。由于最后一次扫描并收缩 slab 持有的 mm_struct 在高速缓存的时候已回收,带有 mm 的进程退出时已经使得指针完全不合法,这有可能,虽然不一定。之所以缺少 bug 报告,是因为在实际过程中很少回收 slab 且极难触发。

310~316 如果 swap_address 已达到 TASK_SIZE,或 mm 是 init_mm,则转到下一个进程。

311 从进程空间的起始部分开始。

312 获取该进程的 mm。

313~314 如果相同,则不检查任何运行中的进程。

315 记录这次的 swap_mm 供下次执行所用。

319 增加引用计数,这样在扫描过程中 mm 不会被释放。

320 释放 mm 锁。

322 用 swap_out_mm()(见 J.6.2 小节)开始扫描 mm。

324 释放对 mm 的引用。

326~327 如果释放了所请求数量的页面,则返回成功。

328 如果这次执行失败,则增加优先级,这样可以扫描更多的进程。

330 返回失败。

J.6.2 函数: swap_out_mm() (mm/vmscan.c)

遍历每个 VMA 并在每个 VMA 上调用 swap_out_mm()。

```

256 static inline int swap_out_mm(struct mm_struct * mm, int count,
257                                int * mmcounter, zone_t * classzone)
258 {
259     unsigned long address;
260     struct vm_area_struct * vma;
261
262     spin_lock(&mm->page_table_lock);
263     address = mm->swap_address;
264     if (address == TASK_SIZE || swap_mm != mm) {
265         /* We raced; dont count this mm but try again */
266         ++ * mmcounter;
267         goto out_unlock;
268     }
269
270     vma = find_vma(mm, address);
271     if (vma) {
272         if (address < vma->vm_start)
273             address = vma->vm_start;
274
275         for (;;) {
276             count = swap_out_vma(mm, vma, address,
277                                 count, classzone);
278             vma = vma->vm_next;
279             if (! vma)
280                 break;
281             if (! count)
282                 goto out_unlock;
283         }
284         address = vma->vm_start;
285     }
286
287     /* Indicate that we reached the end of address space */
288     mm->swap_address = TASK_SIZE;
289
290     out_unlock;
291     spin_unlock(&mm->page_table_lock);
292     return count;
293 }

```

265 为 mm 获取页表锁。

266 从 swap_address 中的地址开始。

267~271 如果地址为 TASK_SIZE, 这意味着已有一个线程在竞争并扫描该进程。它增

加 mmcounter,这样 swap_out_mm()知道去尝试另一个进程。

272 找到该地址的 VMA。

273 假设找到了 VMA,则……

274~275 从 VMA 的起始位置开始。

277~285 扫描它,而在接下来的每个 VMA 上都调用 swap_out_vma()(见 J. 6.3 小节)。如果释放了必需数量(count)的页面,则结束扫描并返回。

288 在扫描完最后一个 VMA 后,设置 swap_address 为 TASK_SIZE,这样在下次执行时 swap_out_mm()可以跳过该进程。

J. 6.3 函数: swap_out_vma() (mm/vmscan.c)

这个函数遍历 VMA,并对其中每个 PGD 调用 swap_out_pgd()。

```

227 static inline int swap_out_vma (struct mm_struct * mm,
228                                struct vm_area_struct * vma,
229                                unsigned long address, int count,
230                                zone_t * classzone)
231 {
232     pgd_t * pgdir;
233     unsigned long end;
234
235     /* Dont swap out areas which are reserved */
236     if (vma->vm_flags & VM_RESERVED)
237         return count;
238
239     pgdir = pgd_offset(mm, address);
240
241     end = vma->vm_end;
242     BUG_ON(address >= end);
243     do {
244         count = swap_out_pgd(mm, vma, pgdir,
245                               address, end, count, classzone);
246         if (! count)
247             break;
248         address = (address + PGDIR_SIZE) & PGDIR_MASK;
249     } while (address && (address < end));
250
251     return count;

```

248 }

233~234 如果设置了 VM_RESERVED 则跳过该 VMA。这用于某些设备驱动,比如小型计算机系统接口(SCSI)的一般驱动。

236 获取该地址的起始 PGD。

238 标记尾部在什么地方,如果起始地址在尾部之后,则调用 BUG()。

240 循环遍历 PGD 直至达到尾部地址。

241 调用 swap_out_pgd()(见 J. 6.4 小节)来计算待释放页面的数量。

242~243 如果已释放足够的页面,则跳出并返回。

244~245 转到下一个 PGD,并移动地址到与 PGD 对齐的下一个地址。

247 返回待释放页面的数量。

J. 6.4 函数: swap_out_pgd() (mm/vmscan.c)

这个函数遍历系统提供的 PGD 中所有 PMD,并调用 swap_out_pmd()。

```

197 static inline int swap_out_pgd (struct mm_struct * mm,
198                         struct vm_area_struct * vma, pgd_t * dir,
199                         unsigned long address, unsigned long end,
200                         int count, zone_t * classzone)
201 {
202     pmd_t * pmd;
203     unsigned long pgd_end;
204
205     if (pgd_none(*dir))
206         return count;
207     if (pgd_bad(*dir)) {
208         pgd_ERROR(*dir);
209         pgd_clear(dir);
210         return count;
211     }
212     pmd = pmd_offset(dir, address);
213     pgd_end = (address + PGDIR_SIZE) & PGDIR_MASK;
214     if (pgd_end && (end > pgd_end))
215         end = pgd_end;

```

```

216 do {
217     count = swap_out_pmd(mm, vma, pmd,
218                           address, end, count, classzone);
219     if (!count)
220         break;
221     address = (address + PMD_SIZE) & PMD_MASK;
222 } while (address && (address < end));
223 return count;
224 }

```

202~203 如果没有 PGD, 则返回。

204~208 如果 PGD 已损坏, 则标记它已损坏并返回。

210 获取起始 PMD。

212~214 计算尾部地址为该 PGD 的尾部地址或为被扫描的 VMA 的尾部地址, 因为它们很接近。

216~222 对 PGD 中每个 PMD, 调用 swap_out_pmd() (见 J. 6.5 小节)。如果已释放足够多的页面, 则跳出并返回。

223 返回待释放页面的数量。

J. 6.5 函数: swap_out_pmd() (mm/vmscan.c)

对 PMD 中每个 PTE, 调用 try_to_swap_out()。在执行完成后, 更新 mm->swap_address 以表示完成的地点, 这样就阻止了在接下来的扫描中检查同一个页面。

```

158 static inline int swap_out_pmd(struct mm_struct * mm,
159                                 struct vm_area_struct * vma, pmd_t * dir,
160                                 unsigned long address, unsigned long end,
161                                 int count, zone_t * classzone)
162 {
163     pte_t * pte;
164     unsigned long pmd_end;
165     if (pmd_none(*dir))
166         return count;
167     if (pmd_bad(*dir)) {
168         pmd_ERROR(*dir);
169         pmd_clear(dir);

```

```

168 return count;
169 }
170
171 pte = pte_offset(dir, address);
172
173 pmd_end = (address + PMD_SIZE) & PMD_MASK;
174 if (end > pmd_end)
175 end = pmd_end;
176
177 do {
178 if (pte_present(*pte)) {
179 struct page *page = pte_page(*pte);
180
181 if (VALID_PAGE(page) && !PageReserved(page)) {
182 count -= try_to_swap_out(mm, vma,
183 address, pte,
184 page, classzone);
185 if (!count)
186 break;
187 }
188 }
189 address += PAGE_SIZE;
190 pte++;
191 } while (address && (address < end));
192 mm->swap_address = address;
193 return count;
194 }

```

163~164 如果没有 PMD 则返回。

165~169 如果 PMD 已损坏，则标记它已损坏并返回。

171 获取起始 PTE。

173~175 计算尾部地址为 PMD 的尾部地址或 VMA 的尾部地址，因为它们很接近。

177~191 循环遍历每个 PTE。

178 确保 PTE 现在都被标记了。

179 获取 PTE 的 struct page。

181 如果是合法页面且没有保留，则……

182 调用 try_to_swap_out()。

183~186 如果已换出足够多的页面，则移动地址到下一个页面，接着跳出并返回。

189~190 转到下一个页面和 PTE。

192 更新 swap_address 以表示结束的地址。

193 返回待释放页面的数量。

J. 6.6 函數: try_to_swap_out() (mm/vmscan.c)

这个函数尝试从进程换出页面。它是个规模较大的函数，因此需要一部分一部分地处理。一般情况下，流程如下所示：

- 确保该页面是需要被换出的(函数导言)。
 - 从页表移除页面和 PTE。
 - 处理页面已在交换高速缓存中的情况。
 - 处理页面为脏或它有相关缓冲区的情况。
 - 处理页面被添加到交换高速缓存的情况。

```

64 return 0;
65
66 if (TryLockPage(page))
67 return 0;

```

53~56 如果页面已上锁(如 I/O 操作中)或 PTE 显示页面最近被访问过,则清除引用位并调用 `mark_page_accessed()`(见 J. 2. 3. 1)来使 `struct page` 反映页面生命周期。为表示它没有被换出,将返回 0。

59~60 如果页面在 `active_list` 中,则不换出它。

63~64 如果页面属于我们不感兴趣的管理区,则不换出它。

66~67 如果页面已因 I/O 而上锁,则跳过它。

```

74 flush_cache_page(vma, address);
75 pte = ptep_get_and_clear(page_table);
76 flush_tlb_page(vma, address);
77
78 if (pte_dirty(pte))
79 set_page_dirty(page);
80

```

74 调用体系结构的钩子程序从所有的 CPU 刷新页面。

75 从页表获取 PTE 并清除它。

76 调用体系结构的钩子程序刷新 TLB。

78~79 如果 PTE 标记为脏,则标记 `struct page` 也为脏,这样才会执行正确的清洗操作。

```

86 if (PageSwapCache(page)) {
87 entry.val = page->index;
88 swap_duplicate(entry);
89 set_swap_pte;
90 set_pte(page_table, swp_entry_to_pte(entry));
91 drop_pte;
92 mm->rss--;
93 UnlockPage(page);
94 {
95 int freeable =
96     page_count(page) - !! page->buffers <= 2;
96 page_cache_release(page);
97 return freeable;
98 }
99 }

```

处理页面已在交换高速缓存中的情况。

86 只有在页面已在交换高速缓存中时进入该代码块。请注意也可以通过调用跳转到 set_swap_pte 和 drop_pte 标记处而进入该代码块。

87~88 为交换项填充索引值。swap_duplicate()验证交换区标识符是否合法并在它合法的情况下增加 swap_map 中的计数。

90 用从交换区获取页面的信息填充 PTE。

92 更新 RSS 以表示由进程映射的页面减少一个。

93 为页面解锁。

95 如果计数目前为 2 或更少,且没有缓冲区,则表示页面可以被释放。如果计数高于 2,则表示它由其他的进程所映射或有文件后援,且“用户”为页面高速缓存。

96 减小引用计数,如果达到 0 则释放页面。请注意,如果页面有文件后援,即便是进程没有映射它,其计数也不会为 0。接下来,shrink_cache()(见 J. 4.1 小节)会从页面高速缓存回收该页面。

97 如果页面被释放则返回。

```
115 if (page->mapping)
116 goto drop_pte;
117 if (!PageDirty(page))
118 goto drop_pte;
124 if (page->buffers)
125 goto preserve;
```

115~116 如果页面有相关联的映射,则从页表销毁它。在没有进程映射它的情况下,shrink_cache()将从页面高速缓存回收该页面。

117~118 如果页面已清除,则可以安全地销毁它。

124~125 如果由于缺页中断的截取操作使页面有相关联的缓冲区,则重新使页面和 PTE 附属于页表,因为在这里也无法处理它。

```
126
127 /*
128 * This is a dirty, swappable page. First of all,
129 * get a suitable swap entry for it, and make sure
130 * we have the swap cache set up to associate the
131 * page with that swap entry.
132 */
133 for (;;) {
134     entry = get_swap_page();
135     if (!entry.val)
```

```

136 break;
137 /* Add it to the swap cache and mark it dirty
138 * (adding to the page cache will clear the dirty
139 * and uptodate bits, so we need to do it again)
140 */
141 if (add_to_swap_cache(page, entry) == 0) {
142 SetPageUptodate(page);
143 set_page_dirty(page);
144 goto set_swap_pte;
145 }
146 /* Raced with "speculative" read_swap_cache_async */
147 swap_free(entry);
148 }
149
150 /* No swap space left */
151 preserve;
152 set_pte(page_table, pte);
153 UnlockPage(page);
154 return 0;
155 }

```

134 为页面分配交换项。

135~136 如果无法分配,则跳出,重新使 PTE 和页面附属于进程页表。

141 添加页面到交换高速缓存。

142 标记内存中的页面为最新。

143 标记页面为脏,接下来便可以将它写出到交换区。

144 跳转到 set_swap_pte,用从交换区获取页面的信息更新 PTE。

147 如果向交换高速缓存添加页面失败,这意味着页面已由预读操作提前放置到交换高速缓存中,则销毁当前操作。

152 重新使 PTE 附属于页表。

153 为页面解锁。

154 如果没有释放任何页面则返回。

J.7 页面交换守护程序

这节描述 kswapd 守护程序主循环的细节问题,kswapd 守护程序在内存很少的情况下被

唤醒。主函数涵盖了 kswapd 是否可以睡眠和如何决定平衡哪些节点。

J. 7.1 初始化 kswapd

J. 7.1.1 函数: kswapd_init() (mm/vmscan.c)

这是启动 kswapd 内核线程的函数。

```
767 static int __init kswapd_init(void)
768 {
769     printk("Starting kswapd\n");
770     swap_setup();
771     kernel_thread(kswapd, NULL, CLONE_FS
772                   | CLONE_FILES
773                   | CLONE_SIGNAL);
772     return 0;
773 }
```

770 在基于物理内存大小的情况下,swap_setup()(见 K. 4.2 小节)建立从后援存储器进行预读取页面的数量。

771 启动 kswapd 内核线程。

J. 7.2 kswapd 守护程序

J. 7.2.1 函数: kswapd() (mm/vmscan.c)

这是 kswapd 内核线程的主函数。

```
720 int kswapd(void * unused)
721 {
722     struct task_struct * tsk = current;
723     DECLARE_WAITQUEUE(wait, tsk);
724
725     daemonize();
726     strcpy(tsk->comm, "kswapd");
727     sigfillset(&tsk->blocked);
728
741     tsk->flags |= PF_MEMALLOC;
742
746     for (;;) {
```



```

747 __set_current_state(TASK_INTERRUPTIBLE);
748 add_wait_queue(&kswapd_wait, &wait);
749
750 mb();
751 if (kswapd_can_sleep())
752 schedule();
753
754 __set_current_state(TASK_RUNNING);
755 remove_wait_queue(&kswapd_wait, &wait);
756
762 kswapd_balance();
763 run_task_queue(&tq_disk);
764 }
765 }

```

725 调用 `daemonize()` 创建该内核线程, 接着移除 mm 上下文, 关闭所有文件并显示父进程。

726 设置进程的名字。

727 忽略所有信号。

741 通过设置该标志位, 物理页面分配器将一直尝试满足对页面的请求。由于该进程将一直尝试释放页面, 那么满足对页面的请求是值得的。

746~764 不断循环。

747~748 添加 kswapd 到等待队列, 准备睡眠。

750 内存块函数(`mb()`)确保在该行为所有 CPU 可见之前进行所有的读和写操作。

751 `kswapd_can_sleep()`(见 J. 7.2.2)循环遍历所有检查必需平衡字段的节点和管理区。如果任意一个设置为 1, 则 kswapd 不能睡眠。

752 通过调用 `schedule()`, kswapd 将开始睡眠直至被 `__alloc_pages()`(见 F.1.3 小节)中物理页面分配器唤醒。

754~755 kswapd 在被唤醒之后, 将从等待队列移除, 因为它现在已开始运行。

762 `kswapd_balance()`(见 J. 7.2.4)循环遍历所有的管理区并为每个待平衡的管理区调用 `try_to_free_pages_zone()`(见 J. 5.3 小节)。

763 运行 I/O 任务队列, 开始向磁盘写数据。

J. 7.2.2 函数: `kswapd_can_sleep()` (`mm/vmscan.c`)

这个函数简单地遍历所有 pgdat, 并为每个 pgdat 调用 `kswapd_can_sleep_pgdat()`。

```

695 static int kswapd_can_sleep(void)
696 {
697 pg_data_t * pgdat;

```

```

698
699 for_each_pgdat(pgdat) {
700 if (! kswapd_can_sleep_pgdat(pgdat))
701 return 0;
702 }
703
704 return 1;
705 }

```

699~702 for_each_pgdat()正如其名字一样地工作。它循环遍历所有可用 pgdat 并在这种情况下对每个 pgdat 调用 kswapd_can_sleep_pgdat()(见 J. 7.2.3)。在 x86 上,只有一个 pgdat。

J. 7.2.3 函数: kswapd_can_sleep_pgdat() (mm/vmscan.c)

这个函数循环遍历所有的管理区以确保没有管理区需要平衡。在管理区中的空闲页面数量达到 pages_low 极值时,由 __alloc_pages()设置 zone->need_balance 标志位。

```

680 static int kswapd_can_sleep_pgdat(pg_data_t * pgdat)
681 {
682 zone_t * zone;
683 int i;
684
685 for (i = pgdat->nr_zones-1; i >= 0; i--) {
686 zone = pgdat->node_zones + i;
687 if (! zone->need_balance)
688 continue;
689 return 0;
690 }
691
692 return 1;
693 }

```

685~689 遍历所有管理区的简单循环。

686 node_zones 字段是所有可用管理区的数组,则增加 i 就可以得到下标。

687~688 如果不需要平衡管理区,则继续。

689 如果需要平衡某管理区,则返回 0,以表示 kswapd 不能睡眠。

692 如果循环完成,则返回以表示 kswapd 可以睡眠。

J. 7.2.4 函数: kswapd_balance() (mm/vmscan.c)

这个函数循环遍历每个 pgdat 直至不需要平衡任何一个。

```
667 static void kswapd_balance(void)
```

```

668 {
669 int need_more_balance;
670 pg_data_t * pgdat;
671
672 do {
673 need_more_balance = 0;
674
675 for_each_pgdat(pgdat)
676 need_more_balance |= kswapd_balance_pgdat(pgdat);
677 } while (need_more_balance);
678 }

```

672~677 循环遍历所有的 pgdat 直至不需要平衡任何一个。

675 对每个 pgdat, 调用 kswapd_balance_pgdat() 检查是否需要平衡节点。如果有任一个节点需要平衡, 则设置 need_more_balance 为 1。

J.7.2.5 函数: kswapd_balance_pgdat() (mm/vmscan.c)

这个函数通过检查每个节点以确定是否需要平衡某节点。如果有任一个管理区需要平衡, 则调用 try_to_free_pages_zone()。

```

641 static int kswapd_balance_pgdat(pg_data_t * pgdat)
642 {
643 int need_more_balance = 0, i;
644 zone_t * zone;
645
646 for (i = pgdat->nr_zones-1; i >= 0; i--) {
647 zone = pgdat->node_zones + i;
648 if (unlikely(current->need_resched))
649 schedule();
650 if (! zone->need_balance)
651 continue;
652 if (! try_to_free_pages_zone(zone, GFP_KSWAPD)) {
653 zone->need_balance = 0;
654 __set_current_state(TASK_INTERRUPTIBLE);
655 schedule_timeout(HZ);
656 continue;
657 }
658 if (check_classzone_need_balance(zone))
659 need_more_balance = 1;
660 else

```



```
661 zone->need_balance = 0;  
662 }  
663  
664 return need_more_balance;  
665 }
```

646~662 循环遍历每个管理区并在需要平衡的情况下调用 `try_to_free_pages_zone()`(见 J. 5.3 小节)。

647 `node_zones` 是一个数组,而 i 是其中的下标。

648~649 如果阻止 `kswapd` 贪婪占用 CPU 的配额超时,则调用 `schedule()`。

650~651 如果该管理区需要平衡,则转到下一个管理区。

652~657 如果函数返回 0,这意味着调用了 `out_of_memory()` 函数,因为无法释放足够量的页面。`kswapd` 睡眠 1 分钟,给系统机会来回收被杀死进程的页面并执行 I/O。而管理区被标记为已平衡,则 `kswapd` 将忽略该管理区直至分配器函数 `alloc_pages()` 被再次调用。

658~661 如果成功,将调用 `check_classzone_need_balance()` 检查以后是否还需要平衡管理区。

664 如果以后还需要平衡某个管理区,则返回 1。

附录 K

交换管理

目 录

| | |
|--|-----|
| K. 1 查找空闲项 | 585 |
| K. 1. 1 函数: get_swap_page() | 585 |
| K. 1. 2 函数: scan_swap_map() | 587 |
| K. 2 交换高速缓存 | 590 |
| K. 2. 1 向交换高速缓存添加页面 | 590 |
| K. 2. 1. 1 函数: add_to_swap_cache() | 590 |
| K. 2. 1. 2 函数: swap_duplicate() | 591 |
| K. 2. 2 从交换高速缓存删除页面 | 592 |
| K. 2. 2. 1 函数: swap_free() | 592 |
| K. 2. 2. 2 函数: swap_entry_free() | 593 |
| K. 2. 3 获取/释放交换高速缓存页面 | 594 |
| K. 2. 3. 1 函数: swap_info_get() | 594 |
| K. 2. 3. 2 函数: swap_info_put() | 595 |
| K. 2. 4 搜索交换高速缓存 | 596 |
| K. 2. 4. 1 函数: lookup_swap_cache() | 596 |
| K. 3 交换区 I/O | 597 |
| K. 3. 1 读取后援存储器 | 597 |
| K. 3. 1. 1 函数: read_swap_cache_async() | 597 |
| K. 3. 2 写入后援存储器 | 599 |
| K. 3. 2. 1 函数: swap_writepage() | 599 |

| | |
|--|-----|
| K. 3. 2. 2 函数:remove_exclusive_swap_page() | 599 |
| K. 3. 2. 3 函数:free_swap_and_cache() | 601 |
| K. 3. 3 块 I/O | 602 |
| K. 3. 3. 1 函数:rw_swap_page() | 602 |
| K. 3. 3. 2 函数:rw_swap_page_base() | 603 |
| K. 3. 3. 3 函数:get_swaphandle_info() | 605 |
| K. 4 激活一个交换区 | 607 |
| K. 4. 1 函数:sys_swapon() | 607 |
| K. 4. 2 函数:swap_setup() | 617 |
| K. 5 禁止一个交换区 | 619 |
| K. 5. 1 函数:sys_swapoff() | 619 |
| K. 5. 2 函数:try_to_unuse() | 623 |
| K. 5. 3 函数:unuse_process() | 627 |
| K. 5. 4 函数:unuse_vma() | 628 |
| K. 5. 5 函数:unuse_pgd() | 629 |
| K. 5. 6 函数:unuse_pmd() | 630 |
| K. 5. 7 函数:unuse_pte() | 631 |

K. 1 查找空闲项

K. 1. 1 函数: get_swap_page() (mm/swapfile.c)

这个函数的调用图见图 11.2。它是搜索交换区的高层函数, 用于找到空闲交换槽并返回 `swp_entry_t`。

```

99 swp_entry_t get_swap_page(void)
100 {
101 struct swap_info_struct * p;
102 unsigned long offset;
103 swp_entry_t entry;
104 int type, wrapped = 0;
105
106 entry.val = 0; /* Out of memory */

```

```
107 swap_list_lock();
108 type = swap_list.next;
109 if (type < 0)
110 goto out;
111 if (nr_swap_pages <= 0)
112 goto out;
113
114 while (1) {
115     p = &swap_info[type];
116     if ((p->flags & SWP_WRITEOK) == SWP_WRITEOK) {
117         swap_device_lock(p);
118         offset = scan_swap_map(p);
119         swap_device_unlock(p);
120         if (offset) {
121             entry = SWP_ENTRY(type, offset);
122             type = swap_info[type].next;
123             if (type < 0 || 
124                 p->prio != swap_info[type].prio) {
125                 swap_list.next = swap_list.head;
126             } else {
127                 swap_list.next = type;
128             }
129             goto out;
130         }
131     }
132     type = p->next;
133     if (!wrapped) {
134         if (type < 0 || p->prio != swap_info[type].prio) {
135             type = swap_list.head;
136             wrapped = 1;
137         }
138     } else
139     if (type < 0)
140         goto out; /* out of swap space */
141 }
142 out:
143 swap_list_unlock();
144 return entry;
145 }
```

107 锁住交换区链表。

108 获取下一个交换区用于分配。这个链表将按交换区的优先级排序。

109~110 如果没有交换区,则返回 NULL。

111~112 如果计数器表明没有槽可用,则返回 NULL。

114~141 循环遍历所有的交换区。

115 从 swap_info 数组获取当前交换信息结构。

116 如果交换区对于写操作可用且为活动的,则……

117 锁住交换区。

118 调用 scan_swap_map()(见 K.1.2 小节)搜索请求的交换映射得到空闲槽。

119 解锁交换设备。

120~130 如果槽空闲,则……

121 用 SWP_ENTRY()为项设置一个标识符编码。

122 记录下一个使用的交换区。

123~126 如果下一个区域是链表的尾部或下一个交换区的优先级和当前的不匹配,则移回到头部。

126~128 否则,转到下一个区域。

129 跳转到 out。

132 转到下一个交换区。

133~138 检查封装。如果达到交换区的链表尾部,则设置 wrapped 为 1。

139~140 如果没有可用的交换区,则跳转到 out。

142 退出函数。

143 解锁交换区链表。

144 如果找到一个项则返回它,否则返回 NULL。

K.1.2 函数: scan_swap_map() (mm/swapfile.c)

这个函数用于在交换区中分配 SWAPFILE_CLUSTER 个顺序页面。当分配完如此数量的页面时,在另一个大小为 SWAPFILE_CLUSTER 的块中搜索空闲槽。如果搜索失败,则恢复到分配第 1 个空闲槽。这种簇的尝试是为了确保槽在 SWAPFILE_CLUSTER 大小的块中分配和释放。

```
36 static inline int scan_swap_map(struct swap_info_struct * si)
37 {
38     unsigned long offset;
```

```

47 if (si->cluster_nr) {
48 while (si->cluster_next <= si->highest_bit) {
49 offset = si->cluster_next++;
50 if (si->swap_map[offset])
51 continue;
52 si->cluster_nr--;
53 goto got_page;
54 }
55 }

```

这块顺序分配 SWAPFILE_CLUSTER 个页面。cluster_nr 初始化为 SWAPFILE_CLUSTER，并在每次分配后减少。

47 如果 cluster_nr 还为正数，则分配下一个可用的顺序槽。

48 如果当前所使用的偏移量 (cluster_next) 小于最高位的已知空闲槽 (highest_bit)，则……

49 记录偏移量并更新下一个空闲槽的下一个簇。

50~51 如果槽还没有真正空闲，则转到下一个。

52 如果找到槽，则减小 cluster_nr 字段的值。

53 跳转到 out。

```

56 si->cluster_nr = SWAPFILE_CLUSTER;
57
58 /* try to find an empty (even not aligned) cluster. */
59 offset = si->lowest_bit;
60 check_next_cluster;
61 if (offset + SWAPFILE_CLUSTER-1 <= si->highest_bit)
62 {
63 int nr;
64 for (nr = offset; nr < offset + SWAPFILE_CLUSTER; nr++)
65 if (si->swap_map[nr])
66 {
67 offset = nr + 1;
68 goto check_next_cluster;
69 }
70 /* We found a completely empty cluster, so start
71 * using it.
72 */
73 goto got_page;

```

74 }

至此,已经顺序分配了 SWAPFILE_CLUSTER 个页面,因此接着查找下一个具有 SWAPFILE_CLUSTER 个页面的空闲块。

56 重新初始化顺序分配的页面计数为 SWAPFILE_CLUSTER。

59 从最低位已知的空闲槽开始搜索。

61 如果偏移量乘以簇大小后的结果小于已知的最后一个空闲槽,则检查所有的页面以鉴别这是否是个大空闲块。

64 从 offset 搜索到 offset + SWAPFILE_CLUSTER。

65~69 如果该槽已经被用,则从这个已知的槽开始搜索另一个空闲槽,

73 由于发现了一个大簇,则跳转。

```
75 /* No luck, so now go finegrained as usual. -Andrea */
76 for (offset = si->lowest_bit; offset <= si->highest_bit ;
77     offset++) {
78     if (si->swap_map[offset])
79         continue;
80     si->lowest_bit = offset + 1;
```

这是个不常使用的循环,用于从 lowest_bit 开始搜索空闲页面。

77~78 如果槽已经使用,则转到下一个。

79 将已知的 lowest_bit 中空闲槽更新为接下来的一个。

```
80 got_page:
81     if (offset == si->lowest_bit)
82         si->lowest_bit++;
83     if (offset == si->highest_bit)
84         si->highest_bit--;
85     if (si->lowest_bit > si->highest_bit) {
86         si->lowest_bit = si->max;
87         si->highest_bit = 0;
88     }
89     si->swap_map[offset] = 1;
90     nr_swap_pages--;
91     si->cluster_next = offset + 1;
92     return offset;
93 }
94     si->lowest_bit = si->max;
95     si->highest_bit = 0;
96     return 0;
```

97 }

如果找到槽,则作一些内部处理并返回。

81~82 如果偏移量为已知的最低位空闲槽(lower_bit),则增加它。

83~84 如果偏移量为最高位的已知空闲槽,则减小它。

85~88 如果最低位和最高位标志位相比较,则没有必要搜索更多的交换区,因为这些标志位表示了最低和最高的已知空闲槽。它用于设置低位槽为可能的最高槽并设置高位标志位为0,以便于在后期搜索时进行删节。这可以在下一次槽被释放时进行修整。

89 设置槽的引用计数。

90 更新可用交换页面的计数(nr_swap_pages)。

91 设置与槽相毗邻的簇,下一次搜索从此处开始。

92 返回空闲槽。

94~96 如果没有可用的空闲槽,则标记区域不可搜索并返回0。

K.2 交换高速缓存

K.2.1 向交换高速缓存添加页面

K.2.1.1 函数: add_to_swap_cache() (mm/swap_state.c)

这个函数的调用图如图 11.3 所示。这个函数封装了正常页面的高速缓存句柄。它首先调用 swap_duplicate() 检查页面是否已在交换高速缓存中存在,如果没有,则调用 add_to_page_cache_unique()。

```

70 int add_to_swap_cache(struct page *page, swp_entry_t entry)
71 {
72     if (page->mapping)
73         BUG();
74     if (!swap_duplicate(entry)) {
75         INC_CACHE_INFO(noent_race);
76         return -ENOENT;
77     }
78     if (add_to_page_cache_unique(page, &swapper_space, entry.val,
79         page_hash(&swapper_space, entry.val)) != 0) {
80         swap_free(entry);
81         INC_CACHE_INFO(exist_race);

```



```

82 return -EEXIST;
83 }
84 if (! PageLocked(page))
85 BUG();
86 if (! PageSwapCache(page))
87 BUG();
88 INC_CACHE_INFO(add_total);
89 return 0;
90 }

```

72~73 在调用这个函数以前调用 PageSwapCache() 进行检查, 以确保页面在交换高速缓存中还不存在。之所以做这个检查是为了确保该页面不存在其他的映射, 以防调用者由于疏忽而没有做相应的检查工作。

74~77 调用 swap_duplicate() (见 K. 2.1.2) 尝试并增加该项的计数。如果槽已经在交换映射中, 则增加用于记录向交换高速缓存中添加页面的竞争项的统计数量并返回-ENOENT。

78 用 add_to_page_cache_unique() (见 J. 1.1.2) 尝试并添加页面至页面高速缓存。这个函数和 add_to_page_cache() (见 J. 1.1.1) 近似, 但它还调用 __find_page_nolock() 在页面高速缓存中搜索复制项。在这里管理地址空间为交换空间。在这种情况下, “文件中的偏移量”即为交换映射中的偏移量, 因此 entry. val 以及页面都是基于地址空间进行 hash 查找并根据交换映射中的偏移量查找。

80~83 如果已经在页面高速缓存中, 则进行竞争操作, 增加用于记录向交换高速缓存插入已存在页面的统计数量并返回 EEXIST。

84~85 如果页面因 I/O 而上锁, 这是个 bug。

86~87 如果目前不在交换高速缓存中, 则表示发生了严重错误。

88 增加永远记录交换高速缓存中页面的统计数量。

89 返回成功。

K. 2.1.2 函数: swap_duplicate() (mm/swapfile.c)

这个函数核实交换项是否合法, 如果合法则增加交换映射计数。

```

1161 int swap_duplicate(swp_entry_t entry)
1162 {
1163 struct swap_info_struct * p;
1164 unsigned long offset, type;
1165 int result = 0;
1166
1167 type = SWP_TYPE(entry);
1168 if (type >= nr_swapfiles)

```

```

1169 goto bad_file;
1170 p = type + swap_info;
1171 offset = SWP_OFFSET(entry);
1172
1173 swap_device_lock(p);
1174 if (offset < p->max && p->swap_map[offset]) {
1175 if (p->swap_map[offset] < SWAP_MAP_MAX - 1) {
1176 p->swap_map[offset]++;
1177 result = 1;
1178 } else if (p->swap_map[offset] <= SWAP_MAP_MAX) {
1179 if (swap_overflow++ < 5)
1180 printk(KERN_WARNING "swap_dup: swap entry
           overflow\n");
1181 p->swap_map[offset] = SWAP_MAP_MAX;
1182 result = 1;
1183 }
1184 }
1185 swap_device_unlock(p);
1186 out:
1187 return result;
1188
1189 bad_file:
1190 printk(KERN_ERR "swap_dup: %s %08lx\n", Bad_file, entry.val);
1191 goto out;
1192 }

```

1161 参数用于交换项增加交换映射计数。

1167~1169 在 swap_info 中为包含项的 swap_info_struct 获取偏移量。如果大于交换区，则跳转到 bad_file。

1170~1171 获取相关的 swap_info_struct 并在其 swap_map 中获取偏移量。

1173 锁住交换设备。

1174 做一次快速的全面检查以确保偏移量在 swap_map 中且槽具有正计数。0 计数可能表示槽不是空闲的，且这是个伪 swp_entry_t。

1175~1177 如果计数不是 SWAP_MAP_MAX，则增加它并返回 1 表示成功。

1178~1183 如果不是，则计数可能溢出了，因此设置它为 SWAP_MAP_MAX，并永久保留该槽。实际上，该条件是不可能成立的。

1185~1187 解锁交换设备并返回。

1190~1191 如果使用了坏设备，则打印错误信息并返回失败。

K. 2.2 从交换高速缓存删除页面

K. 2.2.1 函数: swap_free() (mm/swapfile.c)

减少 swp_entry_t 相关的 swap_map 项。

```

214 void swap_free(swp_entry_t entry)
215 {
216     struct swap_info_struct * p;
217
218     p = swap_info_get(entry);
219     if (p) {
220         swap_entry_free(p, SWP_OFFSET(entry));
221         swap_info_put(p);
222     }
223 }
```

218 swap_info_get()(见 K. 2.3.1) 获取正确的 swap_info_struct, 并执行一系列的调试检查, 以确保区域合法及交换映射项合法。如果都满足, 则锁住交换设备。

219~222 如果合法, 则用 swap_entry_free()(见 K. 2.2.2) 减小相关的交换映射项, 并调用 swap_info_put()(见 K. 2.3.2) 释放设备。

K. 2.2.2 函数: swap_entry_free() (mm/swapfile.c)

```

192 static int swap_entry_free(struct swap_info_struct * p,
193                             unsigned long offset)
194 {
195     int count = p->swap_map[offset];
196
197     if (count < SWAP_MAP_MAX) {
198         count--;
199         p->swap_map[offset] = count;
200         if (!count) {
201             if (offset < p->lowest_bit)
202                 p->lowest_bit = offset;
203             if (offset > p->highest_bit)
204                 p->highest_bit = offset;
205             nr_swap_pages++;
206         }
207     }
208 }
```

```
207 return count;
208 }
```

194 获取当前计数。

196 如果计数表明槽不是永久保留的,则……

197~198 减小计数并存储到交换映射中。

199 如果计数达到 0,则表示槽已空闲,因此更新部分信息。

200~201 如果被释放的槽低于 lowest_bit,则更新 lowest_bit,因为它表示已知的最低空闲槽。

202~203 同样,如果新释放的槽高于 highest_bit,则更新 highest_bit。

204 增加用于表明空闲交换槽数量的计数。

207 返回当前计数。

K.2.3 获取/释放交换高速缓存页面

K.2.3.1 函数: swap_info_get() (mm/swapfile.c)

这个函数为给定的项查找 swap_info_struct,并执行一些基本检查,接着锁住设备。

```
147 static struct swap_info_struct * swap_info_get(swp_entry_t entry)
148 {
149 struct swap_info_struct * p;
150 unsigned long offset, type;
151
152 if (! entry.val)
153 goto out;
154 type = SWP_TYPE(entry);
155 if (type >= nr_swapfiles)
156 goto bad_nofile;
157 p = & swap_info[type];
158 if (!(p->flags & SWP_USED))
159 goto bad_device;
160 offset = SWP_OFFSET(entry);
161 if (offset >= p->max)
162 goto bad_offset;
163 if (! p->swap_map[offset])
164 goto bad_free;
165 swap_list_lock();
166 if (p->prio > swap_info[swap_list.next].prio)
```

```

167 swap_list.next = type;
168 swap_device_lock(p);
169 return p;
170
171 bad_free:
172 printk(KERN_ERR "swap_free: %s %08lx\n", Unused_offset,
        entry.val);
173 goto out;
174 bad_offset:
175 printk(KERN_ERR "swap_free: %s %08lx\n", Bad_offset,
        entry.val);
176 goto out;
177 bad_device:
178 printk(KERN_ERR "swap_free: %s %08lx\n", Unused_file,
        entry.val);
179 goto out;
180 bad_nofile:
181 printk(KERN_ERR "swap_free: %s %08lx\n", Bad_file,
        entry.val);
182 out:
183 return NULL;
184 }

```

152~153 如果提供的项为 NULL, 则返回。

154 在 swap_info 数组中获取偏移量。

155~156 确保这是个合法区域。

157 获取区域的地址。

158~159 如果区域不是活动的, 则打印坏设备错误信息并返回。

160 获取交换映射中的偏移量。

161~162 确保偏移量不在映射的尾部之后。

163~164 确保槽正在使用中。

165 锁住交换区链表。

166~167 如果区域的优先级高于下一个区域的优先级, 则确保当前区域在使用中。

168~169 锁住交换设备并返回交换区描述符。

K. 2.3.2 函数: swap_info_put() (mm/swapfile.c)

这个函数用于为区域和链表解锁。

```
186 static void swap_info_put(struct swap_info_struct * p)
```

```

187 {
188 swap_device_unlock(p);
189 swap_list_unlock();
190 }

```

188 为设备解锁。

189 为交换区链表解锁。

K.2.4 搜索交换高速缓存

K.2.4.1 函数: `lookup_swap_cache()` (`mm/swap_state.c`)

这是在交换高速缓存中查找页面且位于最高层的函数。

```

161 struct page * lookup_swap_cache(swp_entry_t entry)
162 {
163 struct page * found;
164
165 found = find_get_page(&swapper_space, entry.val);
166 /*
167 * Unsafe to assert PageSwapCache and mapping on page found;
168 * if SMP nothing prevents swapoff from deleting this page from
169 * the swap cache at this moment. find_lock_page would prevent
170 * that, but no need to change; we _have_ got the right page.
171 */
172 INC_CACHE_INFO(find_total);
173 if (found)
174 INC_CACHE_INFO(find_success);
175 return found;
176 }

```

165 `find_get_page()`(见 J.1.4.1)是用于返回 `struct page` 的主要函数。它使用常规页面 hash 和高速缓存函数进行快速查找。

172 增加用于记录高速缓存中页面被搜索次数的统计数量。

173~174 如果找到一个,则增加成功找到的计数。

175 返回 `struct page` 或在不存在的情况下返回 `NULL`。

K.3 交换区 I/O

K.3.1 读取后援存储器

K.3.1.1 函数: `read_swap_cache_async()` (`mm/swap_state.c`)

这个函数将返回从交换高速缓存请求的页面。如果不存在,则分配一个页面并放置到交换高速缓存中。而数据会在接下来的调度中调用 `rw_swap_page()` 从磁盘读出。

```

184 struct page * read_swap_cache_async(swp_entry_t entry)
185 {
186     struct page * found_page, * new_page = NULL;
187     int err;
188
189     do {
190         found_page = find_get_page(&swapper_space, entry.val);
191         if (found_page)
192             break;
193
194         /* Get a new page to read into from swap.
195          */
196         if (! new_page) {
197             new_page = alloc_page(GFP_HIGHUSER);
198             if (! new_page)
199                 break; /* Out of memory */
200
201             /* Associate the page with swap entry in the swap cache.
202              * May fail (-ENOENT) if swap entry has been freed since
203              * our caller observed it. May fail (-EEXIST) if there
204              * is already a page associated with this entry in the
205              * swap cache; added by a racing read_swap_cache_async,
206              * or by try_to_swap_out (or shmem_writepage) re-using
207              * the just freed swap entry for an existing page.
208
209             */
210
211             /* May fail (-ENOENT) if swap entry has been freed since
212             * our caller observed it. May fail (-EEXIST) if there
213             * is already a page associated with this entry in the
214             * swap cache; added by a racing read_swap_cache_async,
215             * or by try_to_swap_out (or shmem_writepage) re-using
216             * the just freed swap entry for an existing page.
217         */

```

```

218 err = add_to_swap_cache(new_page, entry);
219 if (!err) {
220 /*
221 * Initiate read into locked page and return.
222 */
223 rw_swap_page(READ, new_page);
224 return new_page;
225 }
226 } while (err != -ENOENT);
227
228 if (new_page)
229 page_cache_release(new_page);
230 return found_page;
231 }

```

189 如果 `add_to_swap_cache()` 向交换高速缓存添加失败，则一直循环。

196 首先用 `find_get_page()` (见 J. 1.4.1) 在交换高速缓存中搜索是否有可用页面。一般情况下，会先调用 `lookup_swap_cache()` (见 K. 2.4.1)，但这会更新数据 (比如搜索高速缓存的次数)，因此直接调用 `find_get_page()` (见 J. 1.4.1)。

203~207 如果页面不在交换高速缓存中且我们尚未分配一个页面，则用 `alloc_page()` 分配一个。

218 调用 `add_to_swap_cache()` (见 K. 2.1.1) 向交换高速缓存添加这个新分配的页面。

223 调用 `rw_swap_page()` (见 K. 3.3.1) 读取数据。页面返回时将上锁并在 I/O 完成时解锁。

224 返回新页面。

226 循环直到 `add_to_swap_cache()` 成功或者另一个进程成功地向交换高速缓存插入了页面。

228~229 这要么是错误处理的路径，要么是另一个进程向交换高速缓存添加了页面。如果新分配了页面，则调用 `page_cache_release()` (见 J. 1.3.2) 释放它。

230 返回在交换高速缓存中查找到的页面或者错误。

K.3.2 写入后援存储器

K.3.2.1 函数: `swap_writepage()` (`mm/swap_state.c`)

这个函数在 `swap_aops` 中注册，用于写出页面。其功能很简单。首先，它调用 `remove_exclusive_swap_page()` 尝试并释放页面。如果页面被释放，则因为没有 I/O 等待在该页面上，在返回页面之前被解锁。否则，调用 `rw_swap_page()` 同步页面到后援存储器。

```

24 static int swap_writepage(struct page * page)
25 {
26 if (remove_exclusive_swap_page(page)) {
27 UnlockPage(page);
28 return 0;
29 }
30 rw_swap_page(WRITE, page);
31 return 0;
32 }

```

26~29 `remove_exclusive_swap_page()`(见 K. 3. 2. 2)将从交换高速缓存中回收页面。如果页面被回收,则在返回它之前解锁。

30 否则,页面依旧在交换高速缓存中,因此调用 `rw_swap_page()`(见 K. 3. 3. 1)同步页面到后援存储器。

K. 3. 2. 2 函数: `remove_exclusive_swap_page()` (mm/swapfile.c)

这个函数将在另一个进程共享页面时起作用。如果可能,将从交换高速缓存中移除并释放页面。在移除页面之后, `swap_free()`将减小以表明交换高速缓存不再使用该槽。而计数器反映了 PTE 的数量,PTE 包含了针对该槽的 `swp_entry_t`。

```

287 int remove_exclusive_swap_page(struct page * page)
288 {
289 int retval;
290 struct swap_info_struct * p;
291 swp_entry_t entry;
292
293 if (! PageLocked(page))
294 BUG();
295 if (! PageSwapCache(page))
296 return 0;
297 if (page_count(page) - !! page->buffers != 2) /* 2: us + cache */
298 return 0;
299
300 entry.val = page->index;
301 p = swap_info_get(entry);
302 if (! p)
303 return 0;
304
305 /* Is the only swap cache user the cache itself? */
306 retval = 0;

```

```

307 if (p->swap_map[SWP_OFFSET(entry)] == 1) {
308 /* Recheck the page count with the pagecache lock held.. */
309 spin_lock(&pagecache_lock);
310 if (page_count(page) - !! page->buffers == 2) {
311 __delete_from_swap_cache(page);
312 SetPageDirty(page);
313 retval = 1;
314 }
315 spin_unlock(&pagecache_lock);
316 }
317 swap_info_put(p);
318
319 if (retval) {
320 block_flushpage(page, 0);
321 swap_free(entry);
322 page_cache_release(page);
323 }
324
325 return retval;
326 }

```

293~294 该操作只能在页面被锁住时使用。

295~296 如果页面不在交换高速缓存中，则不作任何操作。

297~298 如果有其他使用该页面的用户，则不能回收该页面，因此返回。

300 页面的 swp_entry_t 存储在 page->index 中，细节见 2.5 节。

301 用 swap_info_get() (见 K. 2.3.1) 获取 swap_info_struct。

307 如果交换槽的惟一使用者是交换高速缓存本身 (例如，没有进程映射它)，则从交换高速缓存删除页面并释放槽。接着，减小交换槽的使用计数，因为不再使用该交换高速缓存。

310 如果当前用户是该页面的惟一使用者，则可以安全地从交换高速缓存移除页面。如果另一个进程共享它，则必须保持原状。

311 从交换高速缓存删除页面。

313 设置 retval 为 1，这样调用者可以知道页面已被释放且 swap_free() (见 K. 2.2.1) 将被调用以减少交换映射中的使用计数。

317 释放 swap_info_get() (见 K. 2.3.1) 对交换槽的引用。

320 槽已被释放，则调用 block_flushpage()，这样所有的 I/O 都将完成且任何与页面相关的缓冲区都将被释放。

321 用 swap_free() 释放交换槽。

322 释放对页面的引用。

K.3.2.3 函数: free_swap_and_cache() (mm/swapfile.c)

这个函数从交换高速缓存释放一个项并尝试回收页面。请注意该函数只用于交换高速缓存。

```

332 void free_swap_and_cache(swp_entry_t entry)
333 {
334     struct swap_info_struct * p;
335     struct page * page = NULL;
336
337     p = swap_info_get(entry);
338     if (p) {
339         if (swap_entry_free(p, SWP_OFFSET(entry)) == 1)
340             page = find_trylock_page(&swapper_space, entry.val);
341         swap_info_put(p);
342     }
343     if (page) {
344         page_cache_get(page);
345         /* Only cache user (+ us), or swap space full? Free it! */
346         if (page_count(page) - !!page->buffers == 2 ||
347             vm_swap_full()) {
348             delete_from_swap_cache(page);
349             SetPageDirty(page);
350             UnlockPage(page);
351             page_cache_release(page);
352         }
353     }

```

337 获取被请求 entry 的 swap_info 结构。

338~342 假设交换区信息结构存在,则调用 swap_entry_free() 释放交换项。接着用 find_trylock_page() 在交换高速缓存中定位项的页面。请注意在这里页面已经上锁。

341 解除在 337 行对交换信息结构的引用。

343~352 如果找到页面,则尝试回收它。

344 引用页面以防止它被提前释放。

346~349 如果没有进程映射页面或者交换区超过 50% 满(用 vm_swap_full() 检查),则从交换高速缓存删除页面。

350 再次对页面解锁。

351 解除在 344 行对页面的引用。

K.3.3 块 I/O

K.3.3.1 函数: `rw_swap_page()` (`mm/page_io.c`)

这个是用于从后援存储器读取数据至页面或从页面写入数据至后援存储器的主函数。使用什么样的操作取决于第 1 个参数 `rw`。这基本上是对内核函数 `rw_swap_page_base()` 的函数封装。它只是简单地要求对交换缓存中的页面进行操作。

```

85 void rw_swap_page(int rw, struct page *page)
86 {
87     swp_entry_t entry;
88
89     entry.val = page->index;
90
91     if (!PageLocked(page))
92         PAGE_BUG(page);
93     if (!PageSwapCache(page))
94         PAGE_BUG(page);
95     if (!rw_swap_page_base(rw, entry, page))
96         UnlockPage(page);
97 }
```

85 `rw` 表明是读操作还是写操作。

89 从 `index` 字段获取 `swp_entry_t`。

91~92 如果页面没有因 I/O 而上锁, 这是个 bug。

93~94 如果页面不在交换缓存中, 这是个 bug。

95 调用内核函数 `rw_swap_page_base()`。如果返回失败, 则用 `UnlockPage()` 为页面解锁, 这样可以释放它。

K.3.3.2 函数: `rw_swap_page_base()` (`mm/page_io.c`)

这个是用于读或写数据至后援存储器的内核函数。不论是写入一个分区还是一个文件, 块层的函数 `brw_page()` 都用于执行真正的 I/O。这个函数为块层建立必要的缓冲区信息以完成该工作。`brw_page()` 执行异步 I/O, 这样系统将返回被锁住的页面, 该页面在 I/O 完成时解锁。

```

36 static int rw_swap_page_base(int rw, swp_entry_t entry,
                           struct page *page)
```

```

37 {
38 unsigned long offset;
39 int zones[PAGE_SIZE/512];
40 int zones_used;
41 kdev_t dev = 0;
42 int block_size;
43 struct inode * swapf = 0;
44
45 if (rw == READ) {
46 ClearPageUptodate(page);
47 kstat.pswpin++;
48 } else
49 kstat.pswpout++;
50

```

36 参数如下所示：

- rw 表明操作是读还是写。
- entry 用于在后援存储器中定位数据的交换项。
- page 是被读取或被写入的页面。

39 zones 是块层 brw_page() 所需要的参数。它包括一个写入块数的数组。这在后援存储器是一个文件而非一个分区时尤其显得重要。

45~47 如果页面不是从磁盘读取的，则清除 Uptodate 标志位，因为从磁盘上读取的信息很明显不是最新的。在这里增加页面换入(pswpin)的统计数据。

49 如果不是，则只更新页面换出(pswpout)的统计数据。

```

51 get_swaphandle_info(entry, &offset, &dev, &swapf);
52 if (dev) {
53 zones[0] = offset;
54 zones_used = 1;
55 block_size = PAGE_SIZE;
56 } else if (swapf) {
57 int i, j;
58 unsigned int block =
59 offset << (PAGE_SHIFT - swapf->i_sb->s_blocksize_bits);
60
61 block_size = swapf->i_sb->s_blocksize;
62 for (i = 0, j = 0; j < PAGE_SIZE; i++, j += block_size)
63 if (! (zones[i] = bmap(swapf, block++))) {
64 printk("rw_swap_page: bad swap file\n");

```

```

65 return 0;
66 }
67 zones_used = i;
68 dev = swapf->i_dev;
69 } else {
70 return 0;
71 }
72
73 /* block_size == PAGE_SIZE/zones_used */
74 brw_page(rw, page, dev, zones, block_size);
75 return 1;
76 }

```

51 `get_swaphandle_info()`(见 K. 3.3.3)要么返回 `kdev_t`, 要么返回表示交换区的 `struct inode`。

52~55 如果存储区是一个分区, 则只写入一个页面大小的块。因此, `zones` 只有一个项, 即被写入分区得偏移量, 而块大小为 `PAGE_SIZE`。

56 如果不是, 则这是个交换文件, 因此在文件中组成页面那些块必须在调用 `brw_page()` 以前调用 `bmap()` 进行映射。

58~59 计算块的起始位置。

61 单个块的大小存储在文件常驻的文件系统的超级块信息中。

62~66 为每个块调用 `bmap()` 以建立全页面。每个块都存储在管理区数组中供传递给 `brw_page()`。如果任意一个块映射失败, 则返回 0。

67 记录在管理区中构成页面的块的使用量。

68 记录哪个设备被写入。

74 从块层调用 `brw_page()` 执行 I/O 调度。由于 I/O 异步, 因此这个函数立即返回。在 I/O 完成时, 调用回调函数(`end_buffer_io_async()`), 为页面解锁。任意等待该页面的进程在此时都可以被唤醒。

75 返回成功。

K. 3.3.3 函数: `get_swaphandle_info()` (`mm/swapfile.c`)

这个函数要么返回 `kdev_t`, 要么返回管理项所属交换区的 `struct inode`。

```

1197 void get_swaphandle_info(swp_entry_t entry, unsigned long * offset,
1198 kdev_t * dev, struct inode ** swapf)
1199 {
1200 unsigned long type;
1201 struct swap_info_struct * p;

```

```

1202
1203 type = SWP_TYPE(entry);
1204 if (type >= nr_swapfiles) {
1205 printk(KERN_ERR "rw_swap_page: %s %08lx\n", Bad_file,
1206           entry.val);
1207 }
1208
1209 p = &swap_info[type];
1210 *offset = SWP_OFFSET(entry);
1211 if (*offset >= p->max && *offset != 0) {
1212 printk(KERN_ERR "rw_swap_page: %s %08lx\n", Bad_offset,
1213           entry.val);
1214 }
1215 if (p->swap_map && ! p->swap_map[*offset]) {
1216 printk(KERN_ERR "rw_swap_page: %s %08lx\n", Unused_offset,
1217           entry.val);
1218 }
1219 if (!(p->flags & SWP_USED)) {
1220 printk(KERN_ERR "rw_swap_page: %s %08lx\n", Unused_file,
1221           entry.val);
1222 }
1223
1224 if (p->swap_device) {
1225 *dev = p->swap_device;
1226 } else if (p->swap_file) {
1227 *swapf = p->swap_file->d_inode;
1228 } else {
1229 printk(KERN_ERR "rw_swap_page: no swap file or device\n");
1230 }
1231 return;
1232 }

```

1203 获取该项在 swap_info 中所属的区域。

1204~1206 如果索引指向一个不存在的区域，则打印信息并返回。坏文件在 mm/swapfile.c 文件首部附近以静态数组进行声明，打印为“Bad swap file entry.”。

1209 从 swap_info 获取 swap_info_struct。

1210 在交换区中为 entry 获取偏移量。

1211~1214 确保偏移量不超过文件尾部。如果超过则在 Bad_offset 打印信息。

1215~1218 如果偏移量当前没有被使用,这意味着 entry 是已失效的项,则在 Unused_offset 打印错误信息。

1219~1222 如果交换区当前不活动,则在 Unused_file 打印错误信息。

1224 如果交换区是一个设备,则返回 swap_info_struct→swap_device 中的 kdev。

1226~1227 如果这是个交换文件,则返回 struct inode,它通过 swap_info_struct→swap_file→d_inode 可用。

1229 如果不是,则该项没有交换文件或交换设备,因此打印错误信息并返回。

K. 4 激活一个交换区

K. 4. 1 函数: sys_swapon() (mm/swapfile.c)

这个大型函数用于激活交换空间。一般地,它的工作如下所示:

- 在 swap_info 数组中查找一个空闲的 swap_info_struct,并用缺省值初始化它。
- 调用 user_path_walk(),遍历目录树查找提供的 specialfile 并利用文件上的可用数据产生一个 namidata 结构,比如 dentry 和存储它的文件系统信息(vfsmount)。
- 产生 swap_info_struct 字段与交换区维度相对应,并知道如何查找它。如果交换区是一个分区,则在计算块大小以前置块大小为 PAGE_SIZE。如果交换区是一个文件,则直接从 inode 获取信息。
- 确保交换区尚未被激活。如果没有,则从内存分配一个页面并从交换区读取第 1 个页大小的槽。这个页面包括的信息有好槽的数量,以及如何用坏项产生 swap_info_struct→swap_map。
- 系统调用 vmalloc() 为 swap_info_struct→swap_map 分配内存,并将每个可用槽的项初始化为 0,不可用槽对应的项初始化为 SWAP_MAP_BAD。理想情况下,首部信息的文件格式为版本 2,因为版本 1 限制交换区在页大小为 4 KB 的系统结构下,小于 128 MB。如 x86。
- 验证头节点信息与实际的交换区匹配,在这之后,在 swap_info_struct 中填充类似页面最大数量和可用页面数等其余的信息,修改全局统计参数 nr_swap_pages 和 total_swap_pages。
- 至此,交换区已激活并初始化,在交换区的逻辑列表中插入代表此交换区的新元素,仍

遵循按优先级排序的顺序。

```

855 asmlinkage long sys_swapon(const char * specialfile,
                                int swap_flags)
856 {
857 struct swap_info_struct * p;
858 struct nameidata nd;
859 struct inode * swap_inode;
860 unsigned int type;
861 int i, j, prev;
862 int error;
863 static int least_priority = 0;
864 union swap_header * swap_header = 0;
865 int swap_header_version;
866 int nr_good_pages = 0;
867 unsigned long maxpages = 1;
868 int swapfilesize;
869 struct block_device * bdev = NULL;
870 unsigned short * swap_map;
871
872 if (! capable(CAP_SYS_ADMIN))
873 return -EPERM;
874 lock_kernel();
875 swap_list_lock();
876 p = swap_info;

```

855 这两个参数是交换区的路径和用于激活的标志位。

872~873 激活的过程必须具备 CAP_SYS_ADMIN 的权限或者为超级用户,这样才能激活交换区。

874 获取大内核锁(BKL)。

875 锁住交换区链表。

876 获取 swap_info 数组中第 1 个交换区。

```

877 for (type = 0 ; type < nr_swapfiles ; type ++ ,p ++ )
878 if (! (p->flags & SWP_USED))
879 break;
880 error = -EPERM;
881 if (type >= MAX_SWAPFILES) {
882 swap_list_unlock();
883 goto out;

```



```

884 }
885 if (type >= nr_swapfiles)
886 nr_swapfiles = type + 1;
887 p->flags = SWP_USED;
888 p->swap_file = NULL;
889 p->swap_vfsmnt = NULL;
890 p->swap_device = 0;
891 p->swap_map = NULL;
892 p->lowest_bit = 0;
893 p->highest_bit = 0;
894 p->cluster_nr = 0;
895 p->sdev_lock = SPIN_LOCK_UNLOCKED;
896 p->next = -1;
897 if (swap_flags & SWAP_FLAG_PREFER) {
898 p->prio =
899 (swap_flags & SWAP_FLAG_PRIO_MASK) >> SWAP_FLAG_PRIO_SHIFT;
900 } else {
901 p->prio = --least_priority;
902 }
903 swap_list_unlock();

```

查找空闲的 swap_info_struct，并用缺省值初始化它。

877~879 循环遍历 swap_info，直至找到一个未使用的结构。

880 在缺省情况下，返回的错误是拒绝权限，它表明调用者没有合适的权限或使用的交换区已经过量。

881 如果没有空闲结构，且已经激活 MAX_SWAPFILE 个交换区，则为交换链表解锁并返回。

885~886 如果被选择的交换区位于最后一个可知的交换区(nr_swapfiles)之后，则更新 nr_swapfiles。

887 设置标志位以表明该区域在使用中。

888-896 初始化字段为缺省值。

897~902 如果调用者指定了优先级，则使用它或设置它为最低优先级并减小它。这样，交换区就可以优先被激活。

903 释放交换链表的锁。

```

904 error = user_path_walk(specialfile, &nd);
905 if (error)
906 goto bad_swap_2;

```

```

907
908 p->swap_file = nd.dentry;
909 p->swap_vfsmnt = nd.mnt;
910 swap_inode = nd.dentry->d_inode;
911 error = -EINVAL;
912

```

这块遍历 VFS 并获取特定文件的相关信息。

904 user_path_walk()遍历目录结构,获取用于描述 specialfile 的 nameidata 结构。

905~906 如果失败,则返回失败。

908 用返回的 dentry 填充 swap_file 字段。

909 同理,填充 swap_vfsmnt。

910 记录特定文件的索引节点。

911 此时缺省错误为-EINVAL,这表明找到了特定文件,但它不是块设备或常规文件。

```

913 if (S_ISBLK(swap_inode->i_mode)) {
914 kdev_t dev = swap_inode->i_rdev;
915 struct block_device_operations * bdops;
916 devfs_handle_t de;
917
918 p->swap_device = dev;
919 set_blocksize(dev, PAGE_SIZE);
920
921 bd_acquire(swap_inode);
922 bdev = swap_inode->i_bdev;
923 de = devfs_get_handle_from_inode(swap_inode);
924 bdops = devfs_get_ops(de);
925 if (bdops) bdev->bd_op = bdops;
926
927 error = blkdev_get(bdev, FMODE_READ|FMODE_WRITE, 0,
BDEV_SWAP);
928 devfs_put_ops(de); /* Decrement module use count
* now we're safe */
929 if (error)
930 goto bad_swap_2;
931 set_blocksize(dev, PAGE_SIZE);
932 error = -ENODEV;
933 if (! dev || (blk_size[MAJOR(dev)] &&
934 ! blk_size[MAJOR(dev)][MINOR(dev)]))

```



```

935 goto bad_swap;
936 swapfilesize = 0;
937 if (blk_size[MAJOR(dev)])
938 swapfilesize = blk_size[MAJOR(dev)][MINOR(dev)]
939 >> (PAGE_SHIFT - 10);
940 } else if (S_ISREG(swap_inode->i_mode))
941 swapfilesize = swap_inode->i_size >> PAGE_SHIFT;
942 else
943 goto bad_swap;

```

分区中,在计算交换区大小之前配置块设备的代码,或从文件的索引节点获取。

913 检查特定文件是否是块设备。

914~939 这块代码段处理交换区为分区的情况。

914 记录指向设备结构的指针,该设备结构用于块设备。

918 存储指向设备结构的指针,该设备结构用于描述在块 I/O 操作中所需的特定文件。

919 设置设备上的块大小为 PAGE_SIZE,因为我们所关心的是块大小为页大小的交换区。

921 bd_acquire() 函数增加块设备的使用计数。

922 获取指向 block_device 结构的指针,该结构是设备文件的描述符,需要打开它。

923 如果允许则获取 devfs 句柄。devfs 已经超出本书讨论的范围。

924~925 增加设备项的使用计数。

927 打开块设备的读/写模式,并设置 BDEV_SWAP 标志位,它采用的是枚举类型,但在调用 do_open() 的情况下忽略它。

928 减小 devfs 项的使用计数。

929~930 打开文件时若发生错误,则返回失败。

931 再次设置块大小。

932 在这之后,缺省的错误表示没有找到设备。

933~935 确保返回的设备 ok。

937~939 用块设备中页大小的块数量来计算交换文件的大小,用 blk_size 表示。计算交换区的大小是为了确保交换区的信息完整。

941 如果交换区是常规文件,则直接从索引节点获取大小并计算有多少页大小的块。

943 如果文件不是块设备或常规文件,则返回错误。

```

945 error = -EBUSY;
946 for (i = 0 ; i < nr_swapfiles ; i++) {
947 struct swap_info_struct *q = &swap_info[i];
948 if (i == type || !q->swap_file)

```

```

949 continue;
950 if (swap_inode->i_mapping ==
      q->swap_file->d_inode->i_mapping)
951 goto bad_swap;
952 }
953
954 swap_header = (void *) __get_free_page(GFP_USER);
955 if (! swap_header) {
956     printk("Unable to start swapping: out of memory :-)\n");
957     error = -ENOMEM;
958     goto bad_swap;
959 }
960
961 lock_page(virt_to_page(swap_header));
962 rw_swap_page_nolock(READ, SWP_ENTRY(type,0),
                      (char *) swap_header);
963
964 if (! memcmp("SWAP-SPACE", swap_header->magic.magic,10))
965     swap_header_version = 1;
966 else if (! memcmp("SWAPSPACE2", swap_header->magic.magic,10))
967     swap_header_version = 2;
968 else {
969     printk("Unable to find swap-space signature\n");
970     error = -EINVAL;
971     goto bad_swap;
972 }

```

945 接着检查以确保该交换区尚未被激活。如果是，则返回-EBUSY 错误。

946~962 读取整个 swap_info 结构，并确保该区域在未激活的情况下被激活。

954~959 分配页面以从磁盘读取交换区信息。

961 函数 lock_page() 锁住页面并确保在有文件后援的情况下页面与磁盘同步。在这种情况下，只需要标记该页面已上锁，这是 rw_swap_page_nolock() 所需的。

962 读取交换区中第 1 个页槽到 swap_header。

964~672 检查基于交换区信息的版本，并设置 swap_header_version 可用。如果不能识别交换区，则返回-EINVAL。

```

974 switch (swap_header_version) {
975 case 1:
976     memset(((char *) swap_header) + PAGE_SIZE-10, 0, 10);

```

```

977 j = 0;
978 p->lowest_bit = 0;
979 p->highest_bit = 0;
980 for (i = 1; i < 8 * PAGE_SIZE; i++) {
981 if (test_bit(i, (char *) swap_header)) {
982 if (! p->lowest_bit)
983 p->lowest_bit = i;
984 p->highest_bit = i;
985 maxpages = i + 1;
986 j++;
987 }
988 }
989 nr_good_pages = j;
990 p->swap_map = vmalloc(maxpages * sizeof(short));
991 if (! p->swap_map) {
992 error = -ENOMEM;
993 goto bad_swap;
994 }
995 for (i = 1; i < maxpages; i++) {
996 if (test_bit(i, (char *) swap_header))
997 p->swap_map[i] = 0;
998 else
999 p->swap_map[i] = SWAP_MAP_BAD;
1000 }
1001 break;
1002

```

这块在交换区版本为 1 的情况下,读取用于产生交换映射的信息。

976 对魔数字符串(magic string)赋值 0,用于确定交换区的版本。

978~979 初始化 swap_info_struct 结构中的字段为 0。

980~988 具有 8 * PAGE_SIZE 个项的位图存储在交换区中。整个页面,减掉 10 位得到魔数字符串,用于描述交换映射并限制交换区大小在 128 MB 以内。如果设置位为 1,表示磁盘上有一个槽可用。这将计算有多少可用槽,供交换映射使用。

981 测试槽的位是否已设置。

982~983 如果 lowest_bit 字段还没有设置,则设置它为槽。在大多数情况下,lowest_bit 将初始化为 1。

984 一旦找到新槽,则更新 highest_bit。

985 计算页面数量。

986 *j* 是交换区中好页面的计数。
 990 调用 `vmalloc()` 为 `swap_map` 分配内存。
 991~994 如果不能分配内存, 则返回 `ENOMEM`。
 995~1000 对没有槽检查是否为“好”槽。如果是, 则初始化槽计数为 0, 或设置为 `SWAP_MAP_BAD`, 这样就不会使用它。
 1001 退出 `switch` 语句。

```

1003 case 2:
1006 if (swap_header->info.version != 1) {
1007     printk(KERN_WARNING
1008 "Unable to handle swap header version %d\n",
1009     swap_header->info.version);
1010     error = -EINVAL;
1011     goto bad_swap;
1012 }
1013
1014 p->lowest_bit = 1;
1015 maxpages = SWP_OFFSET(SWP_ENTRY(0, ~0UL)) - 1;
1016 if (maxpages > swap_header->info.last_page)
1017     maxpages = swap_header->info.last_page;
1018 p->highest_bit = maxpages - 1;
1019
1020 error = -EINVAL;
1021 if (swap_header->info.nr_badpages > MAX_SWAP_BADPAGES)
1022     goto bad_swap;
1023
1025 if (!(p->swap_map = vmalloc(maxpages * sizeof(short)))) {
1026     error = -ENOMEM;
1027     goto bad_swap;
1028 }
1029
1030 error = 0;
1031 memset(p->swap_map, 0, maxpages * sizeof(short));
1032 for (i = 0; i < swap_header->info.nr_badpages; i++) {
1033     int page = swap_header->info.badpages[i];
1034     if (page <= 0 ||
1035         page >= swap_header->info.last_page)
1036         error = -EINVAL;
1037     else
1038         swap_header->info.nr_goodpages++;
1039 }
1040
1041 if (error)
1042     goto bad_swap;
1043
1044 swap_header->info.nr_goodpages = maxpages - error;
1045
1046 if (error)
1047     goto bad_swap;
1048
1049 if (error)
1050     goto bad_swap;
1051
1052 if (error)
1053     goto bad_swap;
1054
1055 if (error)
1056     goto bad_swap;
1057
1058 if (error)
1059     goto bad_swap;
1060
1061 if (error)
1062     goto bad_swap;
1063
1064 if (error)
1065     goto bad_swap;
1066
1067 if (error)
1068     goto bad_swap;
1069
1070 if (error)
1071     goto bad_swap;
1072
1073 if (error)
1074     goto bad_swap;
1075
1076 if (error)
1077     goto bad_swap;
1078
1079 if (error)
1080     goto bad_swap;
1081
1082 if (error)
1083     goto bad_swap;
1084
1085 if (error)
1086     goto bad_swap;
1087
1088 if (error)
1089     goto bad_swap;
1090
1091 if (error)
1092     goto bad_swap;
1093
1094 if (error)
1095     goto bad_swap;
1096
1097 if (error)
1098     goto bad_swap;
1099
1100 if (error)
1101     goto bad_swap;
1102
1103 if (error)
1104     goto bad_swap;
1105
1106 if (error)
1107     goto bad_swap;
1108
1109 if (error)
1110     goto bad_swap;
1111
1112 if (error)
1113     goto bad_swap;
1114
1115 if (error)
1116     goto bad_swap;
1117
1118 if (error)
1119     goto bad_swap;
1120
1121 if (error)
1122     goto bad_swap;
1123
1124 if (error)
1125     goto bad_swap;
1126
1127 if (error)
1128     goto bad_swap;
1129
1130 if (error)
1131     goto bad_swap;
1132
1133 if (error)
1134     goto bad_swap;
1135
1136 if (error)
1137     goto bad_swap;
1138
1139 if (error)
1140     goto bad_swap;
1141
1142 if (error)
1143     goto bad_swap;
1144
1145 if (error)
1146     goto bad_swap;
1147
1148 if (error)
1149     goto bad_swap;
1150
1151 if (error)
1152     goto bad_swap;
1153
1154 if (error)
1155     goto bad_swap;
1156
1157 if (error)
1158     goto bad_swap;
1159
1160 if (error)
1161     goto bad_swap;
1162
1163 if (error)
1164     goto bad_swap;
1165
1166 if (error)
1167     goto bad_swap;
1168
1169 if (error)
1170     goto bad_swap;
1171
1172 if (error)
1173     goto bad_swap;
1174
1175 if (error)
1176     goto bad_swap;
1177
1178 if (error)
1179     goto bad_swap;
1180
1181 if (error)
1182     goto bad_swap;
1183
1184 if (error)
1185     goto bad_swap;
1186
1187 if (error)
1188     goto bad_swap;
1189
1190 if (error)
1191     goto bad_swap;
1192
1193 if (error)
1194     goto bad_swap;
1195
1196 if (error)
1197     goto bad_swap;
1198
1199 if (error)
1200     goto bad_swap;
1201
1202 if (error)
1203     goto bad_swap;
1204
1205 if (error)
1206     goto bad_swap;
1207
1208 if (error)
1209     goto bad_swap;
1210
1211 if (error)
1212     goto bad_swap;
1213
1214 if (error)
1215     goto bad_swap;
1216
1217 if (error)
1218     goto bad_swap;
1219
1220 if (error)
1221     goto bad_swap;
1222
1223 if (error)
1224     goto bad_swap;
1225
1226 if (error)
1227     goto bad_swap;
1228
1229 if (error)
1230     goto bad_swap;
1231
1232 if (error)
1233     goto bad_swap;
1234
1235 if (error)
1236     goto bad_swap;
1237
1238 if (error)
1239     goto bad_swap;
1240
1241 if (error)
1242     goto bad_swap;
1243
1244 if (error)
1245     goto bad_swap;
1246
1247 if (error)
1248     goto bad_swap;
1249
1250 if (error)
1251     goto bad_swap;
1252
1253 if (error)
1254     goto bad_swap;
1255
1256 if (error)
1257     goto bad_swap;
1258
1259 if (error)
1260     goto bad_swap;
1261
1262 if (error)
1263     goto bad_swap;
1264
1265 if (error)
1266     goto bad_swap;
1267
1268 if (error)
1269     goto bad_swap;
1270
1271 if (error)
1272     goto bad_swap;
1273
1274 if (error)
1275     goto bad_swap;
1276
1277 if (error)
1278     goto bad_swap;
1279
1280 if (error)
1281     goto bad_swap;
1282
1283 if (error)
1284     goto bad_swap;
1285
1286 if (error)
1287     goto bad_swap;
1288
1289 if (error)
1290     goto bad_swap;
1291
1292 if (error)
1293     goto bad_swap;
1294
1295 if (error)
1296     goto bad_swap;
1297
1298 if (error)
1299     goto bad_swap;
1300
1301 if (error)
1302     goto bad_swap;
1303
1304 if (error)
1305     goto bad_swap;
1306
1307 if (error)
1308     goto bad_swap;
1309
1310 if (error)
1311     goto bad_swap;
1312
1313 if (error)
1314     goto bad_swap;
1315
1316 if (error)
1317     goto bad_swap;
1318
1319 if (error)
1320     goto bad_swap;
1321
1322 if (error)
1323     goto bad_swap;
1324
1325 if (error)
1326     goto bad_swap;
1327
1328 if (error)
1329     goto bad_swap;
1330
1331 if (error)
1332     goto bad_swap;
1333
1334 if (error)
1335     goto bad_swap;
1336
1337 if (error)
1338     goto bad_swap;
1339
1340 if (error)
1341     goto bad_swap;
1342
1343 if (error)
1344     goto bad_swap;
1345
1346 if (error)
1347     goto bad_swap;
1348
1349 if (error)
1350     goto bad_swap;
1351
1352 if (error)
1353     goto bad_swap;
1354
1355 if (error)
1356     goto bad_swap;
1357
1358 if (error)
1359     goto bad_swap;
1360
1361 if (error)
1362     goto bad_swap;
1363
1364 if (error)
1365     goto bad_swap;
1366
1367 if (error)
1368     goto bad_swap;
1369
1370 if (error)
1371     goto bad_swap;
1372
1373 if (error)
1374     goto bad_swap;
1375
1376 if (error)
1377     goto bad_swap;
1378
1379 if (error)
1380     goto bad_swap;
1381
1382 if (error)
1383     goto bad_swap;
1384
1385 if (error)
1386     goto bad_swap;
1387
1388 if (error)
1389     goto bad_swap;
1390
1391 if (error)
1392     goto bad_swap;
1393
1394 if (error)
1395     goto bad_swap;
1396
1397 if (error)
1398     goto bad_swap;
1399
1400 if (error)
1401     goto bad_swap;
1402
1403 if (error)
1404     goto bad_swap;
1405
1406 if (error)
1407     goto bad_swap;
1408
1409 if (error)
1410     goto bad_swap;
1411
1412 if (error)
1413     goto bad_swap;
1414
1415 if (error)
1416     goto bad_swap;
1417
1418 if (error)
1419     goto bad_swap;
1420
1421 if (error)
1422     goto bad_swap;
1423
1424 if (error)
1425     goto bad_swap;
1426
1427 if (error)
1428     goto bad_swap;
1429
1430 if (error)
1431     goto bad_swap;
1432
1433 if (error)
1434     goto bad_swap;
1435
1436 if (error)
1437     goto bad_swap;
1438
1439 if (error)
1440     goto bad_swap;
1441
1442 if (error)
1443     goto bad_swap;
1444
1445 if (error)
1446     goto bad_swap;
1447
1448 if (error)
1449     goto bad_swap;
1450
1451 if (error)
1452     goto bad_swap;
1453
1454 if (error)
1455     goto bad_swap;
1456
1457 if (error)
1458     goto bad_swap;
1459
1460 if (error)
1461     goto bad_swap;
1462
1463 if (error)
1464     goto bad_swap;
1465
1466 if (error)
1467     goto bad_swap;
1468
1469 if (error)
1470     goto bad_swap;
1471
1472 if (error)
1473     goto bad_swap;
1474
1475 if (error)
1476     goto bad_swap;
1477
1478 if (error)
1479     goto bad_swap;
1480
1481 if (error)
1482     goto bad_swap;
1483
1484 if (error)
1485     goto bad_swap;
1486
1487 if (error)
1488     goto bad_swap;
1489
1490 if (error)
1491     goto bad_swap;
1492
1493 if (error)
1494     goto bad_swap;
1495
1496 if (error)
1497     goto bad_swap;
1498
1499 if (error)
1500     goto bad_swap;
1501
1502 if (error)
1503     goto bad_swap;
1504
1505 if (error)
1506     goto bad_swap;
1507
1508 if (error)
1509     goto bad_swap;
1510
1511 if (error)
1512     goto bad_swap;
1513
1514 if (error)
1515     goto bad_swap;
1516
1517 if (error)
1518     goto bad_swap;
1519
1520 if (error)
1521     goto bad_swap;
1522
1523 if (error)
1524     goto bad_swap;
1525
1526 if (error)
1527     goto bad_swap;
1528
1529 if (error)
1530     goto bad_swap;
1531
1532 if (error)
1533     goto bad_swap;
1534
1535 if (error)
1536     goto bad_swap;
1537
1538 if (error)
1539     goto bad_swap;
1540
1541 if (error)
1542     goto bad_swap;
1543
1544 if (error)
1545     goto bad_swap;
1546
1547 if (error)
1548     goto bad_swap;
1549
1550 if (error)
1551     goto bad_swap;
1552
1553 if (error)
1554     goto bad_swap;
1555
1556 if (error)
1557     goto bad_swap;
1558
1559 if (error)
1560     goto bad_swap;
1561
1562 if (error)
1563     goto bad_swap;
1564
1565 if (error)
1566     goto bad_swap;
1567
1568 if (error)
1569     goto bad_swap;
1570
1571 if (error)
1572     goto bad_swap;
1573
1574 if (error)
1575     goto bad_swap;
1576
1577 if (error)
1578     goto bad_swap;
1579
1580 if (error)
1581     goto bad_swap;
1582
1583 if (error)
1584     goto bad_swap;
1585
1586 if (error)
1587     goto bad_swap;
1588
1589 if (error)
1590     goto bad_swap;
1591
1592 if (error)
1593     goto bad_swap;
1594
1595 if (error)
1596     goto bad_swap;
1597
1598 if (error)
1599     goto bad_swap;
1600
1601 if (error)
1602     goto bad_swap;
1603
1604 if (error)
1605     goto bad_swap;
1606
1607 if (error)
1608     goto bad_swap;
1609
1610 if (error)
1611     goto bad_swap;
1612
1613 if (error)
1614     goto bad_swap;
1615
1616 if (error)
1617     goto bad_swap;
1618
1619 if (error)
1620     goto bad_swap;
1621
1622 if (error)
1623     goto bad_swap;
1624
1625 if (error)
1626     goto bad_swap;
1627
1628 if (error)
1629     goto bad_swap;
1630
1631 if (error)
1632     goto bad_swap;
1633
1634 if (error)
1635     goto bad_swap;
1636
1637 if (error)
1638     goto bad_swap;
1639
1640 if (error)
1641     goto bad_swap;
1642
1643 if (error)
1644     goto bad_swap;
1645
1646 if (error)
1647     goto bad_swap;
1648
1649 if (error)
1650     goto bad_swap;
1651
1652 if (error)
1653     goto bad_swap;
1654
1655 if (error)
1656     goto bad_swap;
1657
1658 if (error)
1659     goto bad_swap;
1660
1661 if (error)
1662     goto bad_swap;
1663
1664 if (error)
1665     goto bad_swap;
1666
1667 if (error)
1668     goto bad_swap;
1669
1670 if (error)
1671     goto bad_swap;
1672
1673 if (error)
1674     goto bad_swap;
1675
1676 if (error)
1677     goto bad_swap;
1678
1679 if (error)
1680     goto bad_swap;
1681
1682 if (error)
1683     goto bad_swap;
1684
1685 if (error)
1686     goto bad_swap;
1687
1688 if (error)
1689     goto bad_swap;
1690
1691 if (error)
1692     goto bad_swap;
1693
1694 if (error)
1695     goto bad_swap;
1696
1697 if (error)
1698     goto bad_swap;
1699
1700 if (error)
1701     goto bad_swap;
1702
1703 if (error)
1704     goto bad_swap;
1705
1706 if (error)
1707     goto bad_swap;
1708
1709 if (error)
1710     goto bad_swap;
1711
1712 if (error)
1713     goto bad_swap;
1714
1715 if (error)
1716     goto bad_swap;
1717
1718 if (error)
1719     goto bad_swap;
1720
1721 if (error)
1722     goto bad_swap;
1723
1724 if (error)
1725     goto bad_swap;
1726
1727 if (error)
1728     goto bad_swap;
1729
1730 if (error)
1731     goto bad_swap;
1732
1733 if (error)
1734     goto bad_swap;
1735
1736 if (error)
1737     goto bad_swap;
1738
1739 if (error)
1740     goto bad_swap;
1741
1742 if (error)
1743     goto bad_swap;
1744
1745 if (error)
1746     goto bad_swap;
1747
1748 if (error)
1749     goto bad_swap;
1750
1751 if (error)
1752     goto bad_swap;
1753
1754 if (error)
1755     goto bad_swap;
1756
1757 if (error)
1758     goto bad_swap;
1759
1760 if (error)
1761     goto bad_swap;
1762
1763 if (error)
1764     goto bad_swap;
1765
1766 if (error)
1767     goto bad_swap;
1768
1769 if (error)
1770     goto bad_swap;
1771
1772 if (error)
1773     goto bad_swap;
1774
1775 if (error)
1776     goto bad_swap;
1777
1778 if (error)
1779     goto bad_swap;
1780
1781 if (error)
1782     goto bad_swap;
1783
1784 if (error)
1785     goto bad_swap;
1786
1787 if (error)
1788     goto bad_swap;
1789
1790 if (error)
1791     goto bad_swap;
1792
1793 if (error)
1794     goto bad_swap;
1795
1796 if (error)
1797     goto bad_swap;
1798
1799 if (error)
1800     goto bad_swap;
1801
1802 if (error)
1803     goto bad_swap;
1804
1805 if (error)
1806     goto bad_swap;
1807
1808 if (error)
1809     goto bad_swap;
1810
1811 if (error)
1812     goto bad_swap;
1813
1814 if (error)
1815     goto bad_swap;
1816
1817 if (error)
1818     goto bad_swap;
1819
1820 if (error)
1821     goto bad_swap;
1822
1823 if (error)
1824     goto bad_swap;
1825
1826 if (error)
1827     goto bad_swap;
1828
1829 if (error)
1830     goto bad_swap;
1831
1832 if (error)
1833     goto bad_swap;
1834
1835 if (error)
1836     goto bad_swap;
1837
1838 if (error)
1839     goto bad_swap;
1840
1841 if (error)
1842     goto bad_swap;
1843
1844 if (error)
1845     goto bad_swap;
1846
1847 if (error)
1848     goto bad_swap;
1849
1850 if (error)
1851     goto bad_swap;
1852
1853 if (error)
1854     goto bad_swap;
1855
1856 if (error)
1857     goto bad_swap;
1858
1859 if (error)
1860     goto bad_swap;
1861
1862 if (error)
1863     goto bad_swap;
1864
1865 if (error)
1866     goto bad_swap;
1867
1868 if (error)
1869     goto bad_swap;
1870
1871 if (error)
1872     goto bad_swap;
1873
1874 if (error)
1875     goto bad_swap;
1876
1877 if (error)
1878     goto bad_swap;
1879
1880 if (error)
1881     goto bad_swap;
1882
1883 if (error)
1884     goto bad_swap;
1885
1886 if (error)
1887     goto bad_swap;
1888
1889 if (error)
1890     goto bad_swap;
1891
1892 if (error)
1893     goto bad_swap;
1894
1895 if (error)
1896     goto bad_swap;
1897
1898 if (error)
1899     goto bad_swap;
1900
1901 if (error)
1902     goto bad_swap;
1903
1904 if (error)
1905     goto bad_swap;
1906
1907 if (error)
1908     goto bad_swap;
1909
1910 if (error)
1911     goto bad_swap;
1912
1913 if (error)
1914     goto bad_swap;
1915
1916 if (error)
1917     goto bad_swap;
1918
1919 if (error)
1920     goto bad_swap;
1921
1922 if (error)
1923     goto bad_swap;
1924
1925 if (error)
1926     goto bad_swap;
1927
1928 if (error)
1929     goto bad_swap;
1930
1931 if (error)
1932     goto bad_swap;
1933
1934 if (error)
1935     goto bad_swap;
1936
1937 if (error)
1938     goto bad_swap;
1939
1940 if (error)
1941     goto bad_swap;
1942
1943 if (error)
1944     goto bad_swap;
1945
1946 if (error)
1947     goto bad_swap;
1948
1949 if (error)
1950     goto bad_swap;
1951
1952 if (error)
1953     goto bad_swap;
1954
1955 if (error)
1956     goto bad_swap;
1957
1958 if (error)
1959     goto bad_swap;
1960
1961 if (error)
1962     goto bad_swap;
1963
1964 if (error)
1965     goto bad_swap;
1966
1967 if (error)
1968     goto bad_swap;
1969
1970 if (error)
1971     goto bad_swap;
1972
1973 if (error)
1974     goto bad_swap;
1975
1976 if (error)
1977     goto bad_swap;
1978
1979 if (error)
1980     goto bad_swap;
1981
1982 if (error)
1983     goto bad_swap;
1984
1985 if (error)
1986     goto bad_swap;
1987
1988 if (error)
1989     goto bad_swap;
1990
1991 if (error)
1992     goto bad_swap;
1993
1994 if (error)
1995     goto bad_swap;
1996
1997 if (error)
1998     goto bad_swap;
1999
2000 if (error)
2001     goto bad_swap;
2002
2003 if (error)
2004     goto bad_swap;
2005
2006 if (error)
2007     goto bad_swap;
2008
2009 if (error)
2010     goto bad_swap;
2011
2012 if (error)
2013     goto bad_swap;
2014
2015 if (error)
2016     goto bad_swap;
2017
2018 if (error)
2019     goto bad_swap;
2020
2021 if (error)
2022     goto bad_swap;
2023
2024 if (error)
2025     goto bad_swap;
2026
2027 if (error)
2028     goto bad_swap;
2029
2030 if (error)
2031     goto bad_swap;
2032
2033 if (error)
2034     goto bad_swap;
2035
2036 if (error)
2037     goto bad_swap;
2038
2039 if (error)
2040     goto bad_swap;
2041
2042 if (error)
2043     goto bad_swap;
2044
2045 if (error)
2046     goto bad_swap;
2047
2048 if (error)
2049     goto bad_swap;
2050
2051 if (error)
2052     goto bad_swap;
2053
2054 if (error)
2055     goto bad_swap;
2056
2057 if (error)
2058     goto bad_swap;
2059
2060 if (error)
2061     goto bad_swap;
2062
2063 if (error)
2064     goto bad_swap;
2065
2066 if (error)
2067     goto bad_swap;
2068
2069 if (error)
2070     goto bad_swap;
2071
2072 if (error)
2073     goto bad_swap;
2074
2075 if (error)
2076     goto bad_swap;
2077
2078 if (error)
2079     goto bad_swap;
2080
2081 if (error)
2082     goto bad_swap;
2083
2084 if (error)
2085     goto bad_swap;
2086
2087 if (error)
2088     goto bad_swap;
2089
2090 if (error)
2091     goto bad_swap;
2092
2093 if (error)
2094     goto bad_swap;
2095
2096 if (error)
2097     goto bad_swap;
2098
2099 if (error)
2100     goto bad_swap;
2101
2102 if (error)
2103     goto bad_swap;
2104
2105 if (error)
2106     goto bad_swap;
2107
2108 if (error)
2109     goto bad_swap;
2110
2111 if (error)
2112     goto bad_swap;
2113
2114 if (error)
2115     goto bad_swap;
2116
2117 if (error)
2118     goto bad_swap;
2119
2120 if (error)
2121     goto bad_swap;
2122
2123 if (error)
2124     goto bad_swap;
2125
2126 if (error)
2127     goto bad_swap;
2128
2129 if (error)
2130     goto bad_swap;
2131
2132 if (error)
2133     goto bad_swap;
2134
2135 if (error)
2136     goto bad_swap;
2137
2138 if (error)
2139     goto bad_swap;
2140
2141 if (error)
2142     goto bad_swap;
2143
2144 if (error)
2145     goto bad_swap;
2146
2147 if (error)
2148     goto bad_swap;
2149
2150 if (error)
2151     goto bad_swap;
2152
2153 if (error)
2154     goto bad_swap;
2155
2156 if (error)
2157     goto bad_swap;
2158
2159 if (error)
2160     goto bad_swap;
2161
2162 if (error)
2163     goto bad_swap;
2164
2165 if (error)
2166     goto bad_swap;
2167
2168 if (error)
2169     goto bad_swap;
2170
2171 if (error)
2172     goto bad_swap;
2173
2174 if (error)
2175     goto bad_swap;
2176
2177 if (error)
2178     goto bad_swap;
2179
2180 if (error)
2181     goto bad_swap;
2182
2183 if (error)
2184     goto bad_swap;
2185
2186 if (error)
2187     goto bad_swap;
2188
2189 if (error)
2190     goto bad_swap;
2191
2192 if (error)
2193     goto bad_swap;
2194
2195 if (error)
2196     goto bad_swap;
2197
2198 if (error)
2199     goto bad_swap;
2200
2201 if (error)
2202     goto bad_swap;
2203
220
```

```

1037 p->swap_map[page] = SWAP_MAP_BAD;
1038 }
1039 nr_good_pages = swap_header->info.last_page -
1040 swap_header->info.nr_badpages -
1041 1 /* header page */;
1042 if (error)
1043 goto bad_swap;
1044 }

```

在文件格式为版本 2 时,这块读取头信息。

1006~1012 完全确保我们可用处理交换文件的格式,如果不能则返回-EINVAL。请记住,保存版本信息的 swap_header 结构很好地放置在磁盘上。

1014 初始化 lowest_bit 为已知的最低可用槽。

1015~1017 计算 maxpages 初试值为交换映射最可能大的大小,然后根据磁盘上的信息设置它的大小。这确保了交换映射数组不会偶然超载。

1018 初始化 hightest_bit。

1020~1022 确保存在的坏页数量不超过 MAX_SWAP_BADPAGES。

1025~1028 调用 vmalloc() 为交换映射分配内存。

1031 初始化整个交换映射为 0,表示所有的槽都可用。

1032~1038 使用从磁盘加载的信息。设置每个不可用槽为 SWAP_MAP_BAD。

1039~1041 计算可用好页面的数量。

1042-1043 如果发生错误则返回。

```

1045
1046 if (swapfilesize && maxpages > swapfilesize) {
1047     printk(KERN_WARNING
1048         "Swap area shorter than signature indicates\n");
1049     error = -EINVAL;
1050     goto bad_swap;
1051 }
1052 if (!nr_good_pages) {
1053     printk(KERN_WARNING "Empty swap-file\n");
1054     error = -EINVAL;
1055     goto bad_swap;
1056 }
1057 p->swap_map[0] = SWAP_MAP_BAD;
1058 swap_list_lock();
1059 swap_device_lock(p);

```

```

1060 p->max = maxpages;
1061 p->flags = SWP_WRITEOK;
1062 p->pages = nr_good_pages;
1063 nr_swap_pages += nr_good_pages;
1064 total_swap_pages += nr_good_pages;
1065 printk(KERN_INFO "Adding Swap:
                         %dk swap-space (priority %d)\n",
1066 nr_good_pages<<(PAGE_SHIFT-10), p->prio);

```

1046~1051 确保从磁盘加载的信息和实际的交换区维度匹配。如果不匹配，则打印警告信息并返回错误。

1052~1056 如果没有可用的好页，则返回错误。

1057 确保包含交换头信息的映射中第 1 个页面没有使用。如果使用了，则头信息在该交换区第一次使用时会被覆写。

1058~1059 锁住交换链表和交换设备。

1060~1062 填充 swap_info_struct 中剩下的字段。

1063~1064 更新全局可用交换页面的统计数据(nr_swap_pages)和交换页面的总数(total_swap_pages)。

1065~1066 打印交换区激活的信息。

```

1068 /* insert swap space into swap_list: */
1069 prev = -1;
1070 for (i = swap_list.head; i >= 0; i = swap_info[i].next) {
1071 if (p->prio >= swap_info[i].prio) {
1072 break;
1073 }
1074 prev = i;
1075 }
1076 p->next = i;
1077 if (prev < 0) {
1078 swap_list.head = swap_list.next = p - swap_info;
1079 } else {
1080 swap_info[prev].next = p - swap_info;
1081 }
1082 swap_device_unlock(p);
1083 swap_list_unlock();
1084 error = 0;
1085 goto out;

```

1070~1080 根据优先级在交换链表的对应槽中插入新交换区。

1082 为交换设备解锁。

1083 为交换链表解锁。

1084~1085 返回成功。

```

1086 bad_swap:
1087 if (bdev)
1088 blkdev_put(bdev, BDEV_SWAP);
1089 bad_swap_2:
1090 swap_list_lock();
1091 swap_map = p->swap_map;
1092 nd.mnt = p->swap_vfsmnt;
1093 nd.dentry = p->swap_file;
1094 p->swap_device = 0;
1095 p->swap_file = NULL;
1096 p->swap_vfsmnt = NULL;
1097 p->swap_map = NULL;
1098 p->flags = 0;
1099 if (!(swap_flags & SWAP_FLAG_PREFER))
1100 ++least_priority;
1101 swap_list_unlock();
1102 if (swap_map)
1103 vfree(swap_map);
1104 path_release(&nd);
1105 out:
1106 if (swap_header)
1107 free_page((long) swap_header);
1108 unlock_kernel();
1109 return error;
1110 }
```

1087~1088 解除对块设备的引用。

1090~1104 这是为交换链表解锁的错误处理路径,设置 swap_info 中的槽为未使用。如果没有指定,则设置分配给交换映射的内存为已释放。

1104 解除对特定文件的引用。

1106~1107 释放包含交换头信息的页面,因为不再使用它。

1108 释放大内核锁。

1109 返回错误值或成功值。

K.4.2 函数: swap_setup() (mm/swap.c)

这个函数在 kswapd 初始化时进行调用, 用于设置 page_cluster 的大小。这个 page_cluster 变量决定了从文件和后援存储器进行数据换页操作时预读页面的数量。

```

100 void __init swap_setup(void)
101 {
102     unsigned long megs = num_physpages >> (20 - PAGE_SHIFT);
103
104     /* Use a smaller cluster for small-memory machines */
105     if (megs < 16)
106         page_cluster = 2;
107     else
108         page_cluster = 3;
109     /*
110     * Right now other parts of the system means that we
111     * _really_ don't want to cluster much more
112     */
113 }
```

102 计算系统中的内存有多少兆字节。

105 在低内存系统中, 设置 page_cluster 为 2。这意味着, 在预读情况下最多只有 4 个页面从磁盘读到内存。

108 如果不是, 预读将取 8 个页面。

K.5 禁止一个交换区

K.5.1 函数: sys_swapoff() (mm/swapfile.c)

这个函数主要用于更新 swap_info_struct 和交换链表。向交换区换入所有页面的主要任务由 try_to_unuse() 完成。这个函数的功能如下所示:

- 调用 user_path_walk() 获得需禁止的交换文件信息, 并启动 BKL。
- 从交换列表中释放 swap_info_struct, 修改全局统计参数 nr_swap_pages(可用交换页面数) 和 total_swap_pages(交换项总数), 成功后, 可以再次释放 BKL。
- 调用 try_to_unuse(), 将需要禁止的交换区域中所有页交换回去。

- 如果没有足够的内存将所有的项交换回去，则不能简单地删除交换区，而是将其重新插入到系统中。如果成功地将所有项交换回去，则 swap_info_struct 处于未定义状态，swap_map 的空间将由 vfree() 释放。

```

720 asmlinkage long sys_swapoff(const char * specialfile)
721 {
722 struct swap_info_struct * p = NULL;
723 unsigned short * swap_map;
724 struct nameidata nd;
725 int i, type, prev;
726 int err;
727
728 if (! capable(CAP_SYS_ADMIN))
729 return -EPERM;
730
731 err = user_path_walk(specialfile, &nd);
732 if (err)
733 goto out;
734

```

728~729 只有超级用户或具有 CAP_SYS_ADMIN 的进程可用禁止交换区。

731~732 用 user_path_walk() 获取用于表示交换区的特定文件的信息。如果发生错误则跳转到 out。

```

735 lock_kernel();
736 prev = -1;
737 swap_list_lock();
738 for (type = swap_list.head; type >= 0;
           type = swap_info[type].next) {
739 p = swap_info + type;
740 if ((p->flags & SWP_WRITEOK) == SWP_WRITEOK) {
741 if (p->swap_file == nd.dentry)
742 break;
743 }
744 prev = type;
745 }
746 err = -EINVAL;
747 if (type < 0) {
748 swap_list_unlock();
749 goto out_dput;

```



```

750 }
751
752 if (prev < 0) {
753 swap_list.head = p->next;
754 } else {
755 swap_info[prev].next = p->next;
756 }
757 if (type == swap_list.next) {
758 /* just pick something that's safe... */
759 swap_list.next = swap_list.head;
760 }
761 nr_swap_pages -= p->pages;
762 total_swap_pages -= p->pages;
763 p->flags = SWP_USED;

```

获取 BKL, 查找被禁止交换区的 swap_info_struct, 并从交换链表上移除它。

735 获取 BKL。

737 锁住交换链表。

738~745 遍历交换链表并查找被请求区域的 swap_info_struct。调用 dentry 识别区域。

747~750 如果找不到相应的结构, 则返回。

752~760 从交换链表移除, 确保被移除的不是首部。

761 更新空闲交换槽的总数。

762 更新已存在交换槽的总数。

763 标记区域为活动的, 但不能写入。

```

764 swap_list_unlock();
765 unlock_kernel();
766 err = try_to_unuse(type);

```

764 为交换链表解锁。

765 释放 BKL。

766 从交换区换入所有页面。

```

767 lock_kernel();
768 if (err) {
769 /* re-insert swap space back into swap_list */
770 swap_list_lock();
771 for (prev = -1, i = swap_list.head;
    i >= 0;
    prev = i, i = swap_info[i].next)

```



```

772 if (p->prio >= swap_info[i].prio)
773 break;
774 p->next = i;
775 if (prev < 0)
776 swap_list.head = swap_list.next = p - swap_info;
777 else
778 swap_info[prev].next = p - swap_info;
779 nr_swap_pages += p->pages;
780 total_swap_pages += p->pages;
781 p->flags = SWP_WRITEOK;
782 swap_list_unlock();
783 goto out_dput;
784 }

```

这块获取 BKL。如果换入所有页面失败，则重新将区域插入到交换链表中。

767 获取 BKL。

770 锁住交换链表。

771~778 重新将区域插入到交换链表中。插入的位置取决于交换区的优先级。

779~780 更新全局统计数据。

781 再次标记该区域可以安全写。

782~783 为交换链表解锁并返回。

```

785 if (p->swap_device)
786 blkdev_put(p->swap_file->d_inode->i_bdev, BDEV_SWAP);
787 path_release(&nd);
788
789 swap_list_lock();
790 swap_device_lock(p);
791 nd.mnt = p->swap_vfsmnt;
792 nd.dentry = p->swap_file;
793 p->swap_vfsmnt = NULL;
794 p->swap_file = NULL;
795 p->swap_device = 0;
796 p->max = 0;
797 swap_map = p->swap_map;
798 p->swap_map = NULL;
799 p->flags = 0;
800 swap_device_unlock(p);
801 swap_list_unlock();

```

```

802 vfree(swap_map);
803 err = 0;
804
805 out_dput;
806 unlock_kernel();
807 path_release(&nd);
808 out;
809 return err;
810 }

```

这块代码在交换区被成功禁止以关闭块设备并标记 swap_info_struct 为空闲时使用。

785~786 关闭块设备。

787 释放路径信息。

789~790 获取交换链表的锁和交换设备的锁。

791~799 重置 swap_info_struct 中的字段为缺省值。

800~801 释放交换链表和交换设备。

801 释放用于交换映射的内存。

806 释放 BKL。

807 在错误路径上执行到这个地方,则释放路径信息。

809 返回成功或失败。

K. 5.2 函数: try_to_unuse() (mm/swapfile.c)

这个函数在源代码中注释得很详实,虽然有些地方是推测的或有些地方并不准确。注释在这里省略以求简洁。

```

513 static int try_to_unuse(unsigned int type)
514 {
515     struct swap_info_struct * si = &swap_info[type];
516     struct mm_struct * start_mm;
517     unsigned short * swap_map;
518     unsigned short swcount;
519     struct page * page;
520     swp_entry_t entry;
521     int i = 0;
522     int retval = 0;
523     int reset_overflow = 0;
525

```

```

540 start_mm = &init_mm;
541 atomic_inc(&init_mm.mm_users);
542

```

540~541 页面中起始 mm_struct 是 init_mm。即使是在特定的结构并不出现以阻止函数剩余部分写操作的情况下,该计数也会减小。

```

556 while ((i = find_next_to_unuse(si, i))) {
557 /*
558 * Get a page for the entry, using the existing swap
559 * cache page if there is one. Otherwise, get a clean
560 * page and read the swap into it.
561 */
562 swap_map = &si->swap_map[i];
563 entry = SWP_ENTRY(type, i);
564 page = read_swap_cache_async(entry);
565 if (!page) {
566 if (!*swap_map)
567 continue;
568 retval = -ENOMEM;
569 break;
570 }
571
572 /*
573 * Dont hold on to start_mm if it looks like exiting.
574 */
575 if (atomic_read(&start_mm->mm_users) == 1) {
576 mmput(start_mm);
577 start_mm = &init_mm;
578 atomic_inc(&init_mm.mm_users);
579 }

```

556 这是函数中主循环的开始。从交换映射的起始部分开始,用 find_next_to_unuse() 搜索下一个待释放的项,直至所有的交换映射都换入。

562~564 获取 swp_entry_t 并调用 read_swap_cache_async() (见 K.3.1.1) 查找交换高速缓存中的页面,或者新分配一个页面供存储从磁盘读取的数据。

565~576 如果获取页面失败,则意味着该槽已由另一个进程或线程(其进程可能在任意某处)单独释放掉,或内存溢出了。如果是单独释放掉的,则继续下一个映射,否则返回-ENOMEM。

581 检查以确保该 mm 并不存在。如果是,则减小它的计数并回退进行 mm 初始化。

```

587 /*
588 * Wait for and lock page. When do_swap_page races with
589 * try_to_unuse, do_swap_page can handle the fault much
590 * faster than try_to_unuse can locate the entry. This
591 * apparently redundant "wait_on_page" lets try_to_unuse
592 * defer to do_swap_page in such a case - in some tests,
593 * do_swap_page and try_to_unuse repeatedly compete.
594 */
595 wait_on_page(page);
596 lock_page(page);
597
598 /*
599 * Remove all references to entry, without blocking.
600 * Whenever we reach init_mm, there's no address space
601 * to search, but use it as a reminder to search shmem.
602 */
603 shmem = 0;
604 swcount = *swap_map;
605 if (swcount > 1) {
606 flush_page_to_ram(page);
607 if (start_mm == &init_mm)
608 shmem_unuse(entry, page);
609 else
610 unuse_process(start_mm, entry, page);
611 }

```

595 等待页面完成 I/O。在它返回以后,内存中页面的信息和磁盘上的信息就是相同的了。

596 锁住页面。

604 获取交换映射引用计数。

605 如果计数为正,则……

606 在页面准备插入进程页表时,必须从 D-cache 释放它,或者内核对页面的修改对进程“不可见”。

607~608 如果使用 init_mm,则调用 shmem_unuse()(见 L. 6.2 小节)释放在使用中的任意共享内存区域中的页面。

610 如果不是,则更新当前 mm 中的 PTE,它引用了该页面。

```

612 if (*swap_map > 1) {
613 int set_start_mm = (*swap_map >= swcount);
614 struct list_head *p = &start_mm->mmlist;

```

```

615 struct mm_struct * new_start_mm = start_mm;
616 struct mm_struct * mm;
617
618 spin_lock(&mmlist_lock);
619 while (*swap_map > 1 &&
620 (p = p->next) != &start_mm->mmlist) {
621 mm = list_entry(p, struct mm_struct,
622                  mmlist);
623 swcount = *swap_map;
624 if (mm == &init_mm) {
625 set_start_mm = 1;
626 spin_unlock(&mmlist_lock);
627 shmem = shmem_unuse(entry, page);
628 spin_lock(&mmlist_lock);
629 } else
630 unuse_process(mm, entry, page);
631 if (set_start_mm && *swap_map < swcount) {
632 new_start_mm = mm;
633 set_start_mm = 0;
634 }
635 atomic_inc(&new_start_mm->mm_users);
636 spin_unlock(&mmlist_lock);
637 mmput(start_mm);
638 start_mm = new_start_mm;
639 }

```

612~637 如果项还存在,则开始遍历所有的 mm_struct 以查找该页面的引用,并更新相关的 PTE。

618 锁住 mm 链表。

619~632 继续搜索直至找到所有的 mm_struct。不要遍历整个链表超过一次。

621 获取链表项的 mm_struct。

623~627 如果 mm 是 init_mm 则调用 shmem_unuse()(见 L. 6.2 小节),因为这表示页面来自虚拟文件系统。如果不是,则调用 unuse_process()(见 K. 5.3 小节)遍历当前进程的页表以搜索交换项。如果找到,则释放该项,并重新实例化 PTE 中的页面。

630~633 如果需要从 init_mm 重新搜索 mm_struct,则记录下来。

```

654 if (*swap_map == SWAP_MAP_MAX) {
655 swap_list_lock();

```

```

656 swap_device_lock(si);
657 nr_swap_pages++;
658 *swap_map = 1;
659 swap_device_unlock(si);
660 swap_list_unlock();
661 reset_overflow = 1;
662 }

```

654 如果交换映射项是永久映射的,则需要所有的进程更新它们的 PTE 指向页面,而实际上,交换映射项是空闲的。在实际情况下,一个槽在第 1 个位置永久映射基本上不太可能。

654~661 锁住链表和交换设备,设置交换映射项为 1,然后再次对它们解锁并记录溢出发生的重设置。

```

683 if ((*swap_map > 1) && PageDirty(page) &&
PageSwapCache(page)) {
684 rw_swap_page(WRITE, page);
685 lock_page(page);
686 }
687 if (PageSwapCache(page)) {
688 if (shmem)
689 swap_duplicate(entry);
690 else
691 delete_from_swap_cache(page);
692 }

```

683~686 在依旧存在页面引用的这种少数情况下,将页面写回磁盘,这样至少在另一个进程引用它的情况下,可以将页面从磁盘正确地复制回去。

687~689 如果页面在交换高速缓存中且属于共享内存文件系统,则调用 swap_duplicate() 产生对页面的引用,这样便可以在后期调用 shmem_unuse() 再次尝试并移除它。

691 如果不是,对于一般页面而言,则仅仅从交换高速缓存删除它们。

```

699 SetPageDirty(page);
700 UnlockPage(page);
701 page_cache_release(page);

```

699 标记页面为脏,这样换出代码将保留页面。如果需要再次移除它,换出代码能正确地将它写到新交换区。

700 为页面解锁。

701 释放对页面高速缓存中页面的引用。

```
708 if (current->need_resched)
```

```

714 schedule();
715 }
716
717 mmput(start_mm);
718 if (reset_overflow) {
719 printk(KERN_WARNING "swapoff: cleared swap entry
    overflow\n");
720 swap_overflow = 0;
721 }
722 return retval;
723 }

```

708~709 如果有必要,则调用 schedule(),这样交换区的禁止不会占用整个 CPU。

717 释放对 mm 的引用。

718~721 如果必须移除永久映射的页面,则打印警告信息,这样在后期有错误发生这种极少情况下,可以参考发生了什么错误。

717 返回成功或失败。

K.5.3 函数: unuse_process() (mm/swapfile.c)

这个函数开始遍历页表,用于从 mm 管理的进程页表中移除被请求的页面和项。这只在交换区被禁止时使用,因此它即便是开销很大的操作,却也是很少使用的操作。这一组函数应当能快速组织成标准的页表遍历操作。

```

454 static void unuse_process(struct mm_struct * mm,
455 swp_entry_t entry, struct page * page)
456 {
457 struct vm_area_struct * vma;
458
459 /*
460 * Go through process' page directory.
461 */
462 spin_lock(&mm->page_table_lock);
463 for (vma = mm->mmap; vma; vma = vma->vm_next) {
464 pgd_t * pgd = pgd_offset(mm, vma->vm_start);
465 unuse_vma(vma, pgd, entry, page);
466 }
467 spin_unlock(&mm->page_table_lock);
468 return;

```



469 }

462 锁住进程页表。

463 遍历每个由该 mm 管理的 VMA。请记得单个页面帧可以映射到多个地方。

462 获取 PGD, 它用于管理 VMA 起始处。

465 调用 unuse_vma() (见 K. 5.4 小节) 搜索页面的 VMA。

467~468 整个 mm 已经搜索, 则为进程页表解锁并返回。

K. 5.4 函数: unuse_vma() (mm/swapfile.c)

这个函数搜索映射页面并使用给定交换项的页表项所请求的 VMA。它为 VMA 所映射的每个 PGD 调用 unuse_pgd()。

```

440 static void unuse_vma(struct vm_area_struct * vma, pgd_t * pgdir,
441 swp_entry_t entry, struct page * page)
442 {
443 unsigned long start = vma->vm_start, end = vma->vm_end;
444
445 if (start >= end)
446 BUG();
447 do {
448 unuse_pgd(vma, pgdir, start, end - start, entry, page);
449 start = (start + PGDIR_SIZE) & PGDIR_MASK;
450 pgdir++;
451 } while (start && (start < end));
452 }
```

443 获取 VMA 的 start 和 end 的虚拟地址。

445~446 检查 start 位置不在 end 之后。如果发生这样的情况, 在内核中会引起相当严重的后果。

447~451 以 PGDIR_SIZE 为间距遍历 VMA 直至达到 VMA 的尾部。这有效地遍历了映射 VMA 的每个 PGD。

448 调用 unuse_pgd() (见 K. 5.5 小节) 遍历解除页面映射的 PGD。

449 移动虚拟地址的起始位置为下一个 PGD 的首部。

450 移动 pgdir 到 VMA 中下一个 PGD。

K. 5.5 函数: unuse_pgd() (mm/swapfile.c)

这个函数搜索被请求的 PGD 以得到映射该页面及使用给定交换项的页表项。它对 PGD

映射中每个 PMD 调用 unuse_pmd()。

```

409 static inline void unuse_pgd(struct vm_area_struct * vma, pgd_t * dir,
410 unsigned long address, unsigned long size,
411 swp_entry_t entry, struct page * page)
412 {
413     pmd_t * pmd;
414     unsigned long offset, end;
415
416     if (pgd_none(*dir))
417         return;
418     if (pgd_bad(*dir)) {
419         pgd_ERROR(*dir);
420         pgd_clear(dir);
421         return;
422     }
423     pmd = pmd_offset(dir, address);
424     offset = address & PGDIR_MASK;
425     address &= ~PGDIR_MASK;
426     end = address + size;
427     if (end > PGDIR_SIZE)
428         end = PGDIR_SIZE;
429     if (address >= end)
430         BUG();
431     do {
432         unuse_pmd(vma, pmd, address, end - address, offset, entry,
433         page);
434         address = (address + PMD_SIZE) & PMD_MASK;
435         pmd++;
436     } while (address && (address < end));
437 }

```

416~417 如果不存在 PGD，则返回。

418~422 如果 PGD 已经损坏，则设置对应的错误，清除 PGD 并返回。很少有体系结构会发生这样的情况。

423 获取 PGD 中第 1 个 PMD 的地址。

424 计算偏移量的大小为地址在 PGD 中的偏移量。请记得在第一次调用该函数的时候，有可能只搜索一部分的 PGD。

425 对齐地址到 PGD。

426 计算搜索的结束地址。

427~428 如果尾部超过 PGD，则设置尾部为 PGD 的尾部。

429~430 如果起始地址在结束地址之后，则表示发生了严重错误。

431~436 以 PMD_SIZE 为间距遍历 PGD，并为 PGD 中每个 PMD 调用 unuse_pmd()（见 K.5.6 小节）。

K.5.6 函数：unuse_pmd() (mm/swapfile.c)

这个函数搜索被请求的 PGD 以得到映射该页面及使用给定交换项的页表项。它对 PMD 映射中每个 PTE 调用 unuse_pmd()。

```

381 static inline void unuse_pmd(struct vm_area_struct * vma, pmd_t * dir,
382 unsigned long address, unsigned long size, unsigned long offset,
383 swap_entry_t entry, struct page * page)
384 {
385     pte_t * pte;
386     unsigned long end;
387
388     if (pmd_none(*dir))
389         return;
390     if (pmd_bad(*dir)) {
391         pmd_ERROR(*dir);
392         pmd_clear(dir);
393         return;
394     }
395     pte = pte_offset(dir, address);
396     offset += address & PMD_MASK;
397     address &= ~PMD_MASK;
398     end = address + size;
399     if (end > PMD_SIZE)
400         end = PMD_SIZE;
401     do {
402         unuse_pte(vma, offset + address - vma->vm_start, pte, entry, page);
403         address += PAGE_SIZE;
404         pte++;
405     } while (address && (address < end));
406 }
```

388~389 如果不存在 PMD 则返回。

390~394 设置对应的错误并在 PMD 损坏的情况下清除它。很少有体系结构会发生这样的情况。

395 计算地址的起始 PTE。

396 设置偏移量的大小为 PMD 中自起始位置的偏移量。

397 对齐地址到 PMD。

398~400 计算结束地址。如果超过了 PMD 的尾部,则设置它的地址为该 PMD 的尾部地址。

401~405 以 PAGE_SIZE 为间距遍历 PMD,并为每个 PTE 调用 unuse_pte()(见 K.5.7 小节)。

K.5.7 函数: unuse_pte() (mm/swapfile.c)

这个函数检查在 dir 的 PTE 和我们搜索的 entry 是否相匹配。如果是,则释放交换项,而用于表示 PTE 的页面引用将更新为映射它。

```

365 static inline void unuse_pte(struct vm_area_struct * vma,
366                           unsigned long address,
367                           pte_t * dir, swp_entry_t entry, struct page * page)
368 {
369     pte_t pte = *dir;
370
371     if (likely(pte_to_swp_entry(pte).val != entry.val))
372         return;
373     if (unlikely(pte_none(pte) || pte_present(pte)))
374         return;
375     get_page(page);
376     set_pte(dir, pte_mkold(mk_pte(page, vma->vm_page_prot)));
377     swap_free(entry);
378 }
```

370~371 如果 entry 和 PTE 不匹配,则返回。

372~373 如果不存在 PTE 或 PTE 已经呈现(意思是 entry 不可能映射这里),则返回。

374 否则,表示已经找到要查找的项,那么对页面进行引用,因为一个新的 PTE 将映射它。

375 更新 PTE 映射到页面。

376 释放交换项。

377 增加进程的 RSS 计数。

附录 L

共享内存虚拟文件系统

目 录

| | |
|--|-----|
| L. 1 初始化 shmfs | 635 |
| L. 1. 1 函数: init_tmpfs() | 635 |
| L. 1. 2 函数: shmem_read_super() | 637 |
| L. 1. 3 函数: shmem_set_size() | 639 |
| L. 2 在 tmpfs 中创建文件 | 641 |
| L. 2. 1 函数: shmem_create() | 641 |
| L. 2. 2 函数: shmem_mknod() | 641 |
| L. 2. 3 函数: shmem_get_inode() | 642 |
| L. 3 tmpfs 中的文件操作 | 645 |
| L. 3. 1 内存映射 | 645 |
| L. 3. 1. 1 函数: shmem_mmap() | 645 |
| L. 3. 2 读取文件 | 646 |
| L. 3. 2. 1 函数: shmem_file_read() | 646 |
| L. 3. 2. 2 函数: do_shmem_file_read() | 647 |
| L. 3. 2. 3 函数: file_read_actor() | 649 |
| L. 3. 3 写入文件 | 651 |
| L. 3. 3. 1 函数: shmem_file_write() | 651 |
| L. 3. 4 符号链接 | 654 |
| L. 3. 4. 1 函数: shmem_symlink() | 654 |
| L. 3. 4. 2 函数: shmem_readlink_inline() | 656 |

| | |
|--|-----|
| L. 3.4.3 函数:shmem_follow_link_inline() | 656 |
| L. 3.4.4 函数:shmem_readlink() | 656 |
| L. 3.5 同步文件 | 658 |
| L. 3.5.1 函数:shmem_sync_file() | 658 |
| L. 4 tmpfs 中的索引节点操作 | 659 |
| L. 4.1 截取 | 659 |
| L. 4.1.1 函数:shmem_truncate() | 659 |
| L. 4.1.2 函数:shmem_truncate_indirect() | 660 |
| L. 4.1.3 函数:shmem_truncate_direct() | 662 |
| L. 4.1.4 函数:shmem_free_swp() | 663 |
| L. 4.2 链接 | 664 |
| L. 4.2.1 函数:shmem_link() | 664 |
| L. 4.3 解除链接 | 665 |
| L. 4.3.1 函数:shmem_unlink() | 665 |
| L. 4.4 创建目录 | 665 |
| L. 4.4.1 函数:shmem_mkdir() | 665 |
| L. 4.5 移除目录 | 666 |
| L. 4.5.1 函数:shmem_rmdir() | 666 |
| L. 4.5.2 函数:shmem_empty() | 666 |
| L. 4.5.3 函数:shmem_positive() | 667 |
| L. 5 虚拟文件中的缺页中断 | 668 |
| L. 5.1 缺页中断时读取页面 | 668 |
| L. 5.1.1 函数:shmem_nopage() | 668 |
| L. 5.1.2 函数:shmem_getpage() | 669 |
| L. 5.2 定位交换页面 | 676 |
| L. 5.2.1 函数:shmem_alloc_entry() | 676 |
| L. 5.2.2 函数:shmem_swp_entry() | 676 |
| L. 6 交换空间交互 | 679 |
| L. 6.1 函数:shmem_writepage() | 679 |
| L. 6.2 函数:shmem_unuse() | 681 |
| L. 6.3 函数:shmem_unuse_inode() | 682 |
| L. 6.4 函数:shmem_find_swp() | 685 |
| L. 7 建立共享区 | 686 |

| | |
|------------------------------|-----|
| L. 7.1 函数:shmem_zero_setup() | 686 |
| L. 7.2 函数:shmem_file_setup() | 686 |
| L. 8 System V IPC | 689 |
| L. 8.1 创建一个 SYSV 共享区 | 689 |
| L. 8.1.1 函数:sys_shmget() | 689 |
| L. 8.1.2 函数:newseg() | 690 |
| L. 8.2 附属一个 SYSV 共享区 | 692 |
| L. 8.2.1 函数:sys_shmat() | 692 |

L. 1 初始化 shmemfs

L. 1.1 函数: init_tmpfs() (mm/shmem.c)

这个函数用于注册和挂载 tmpfs 及 shmemfs 文件系统。

```

1551 #ifdef CONFIG_TMPFS
1553 static DECLARE_FSTYPE(shmem_fs_type, "shm",
1554                         shmem_read_super, FS_LITTER);
1554 static DECLARE_FSTYPE(tmpfs_fs_type, "tmpfs",
1555                         shmem_read_super, FS_LITTER);
1555 #else
1556 static DECLARE_FSTYPE(tmpfs_fs_type, "tmpfs",
1557                         shmem_read_super, FS_LITTER|FS_NOMOUNT);
1557#endif

1560 static int __init init_tmpfs(void)
1561 {
1562     int error;
1563
1564     error = register_filesystem(&tmpfs_fs_type);
1565     if (error) {
1566         printk(KERN_ERR "Could not register tmpfs\n");
1567         goto out3;
1568     }
1569 #ifdef CONFIG_TMPFS

```

```

1570 error = register_filesystem(&shmfs_fs_type);
1571 if (error) {
1572     printk(KERN_ERR "Could not register shm fs\n");
1573     goto out2;
1574 }
1575 devfs_mk_dir(NULL, "shm", NULL);
1576 #endif
1577 shm_mnt = kern_mount(&tmpfs_fs_type);
1578 if (IS_ERR(shm_mnt)) {
1579     error = PTR_ERR(shm_mnt);
1580     printk(KERN_ERR "Could not kern_mount tmpfs\n");
1581     goto out1;
1582 }
1583
1584 /* The internal instance should not do size checking */
1585 shmfs_set_size(SHMEM_SB(shm_mnt->mnt_sb),
1586                 ULONG_MAX, ULONG_MAX);
1586 return 0;
1587
1588 out1:
1589 #ifdef CONFIG_TMPFS
1590 unregister_filesystem(&shmfs_fs_type);
1591 out2:
1592 #endif
1593 unregister_filesystem(&tmpfs_fs_type);
1594 out3:
1595 shm_mnt = ERR_PTR(error);
1596 return error;
1597 }
1598 module_init(init_tmpfs)

```

1551 shm 文件系统只有在编译时定义 CONFIG_TMPFS 的情况下可以挂载。即使没有指定,也会因 fork() 而为匿名共享内存建立 tmpfs。

1553 声明在文件<linux/fs.h>中的 DECLARE_FSTYPE(), 声明了 tmpfs_fs_type 是 struct file_system_type, 并填充了 4 个字段。“tmpfs”是它的可读名字。shmfs_read_super() 函数用于为文件系统读取超级块(超级块的细节以及它们如何匹配文件系统已经超出本书范畴)。FS_LITTER 是一个标志位, 用于表明应该在 dcache 中维护文件系统树。最后, 这个宏设置文件系统的模块所有者成为载入文件系统的模块。

1560 __init 放置该函数在初始化部分。这意味着, 内核启动完毕后, 这个函数的代码将

被移除。

1564~1568 注册文件系统为 tmpfs_fs_type 类型,这已在第 1433 行进行了声明。如果失败,跳转到 out3,返回适当的错误。

1569~1574 如果在配置时指定了 tmpfs,那么注册 shmem 文件系统。如果失败,跳转到 out2,解除 tmpfs_fs_type 的注册并返回错误。

1575 如果由设备文件系统(devfs)管理/dev/,则创建一个新的 shm 目录。如果内核没有使用 devfs,则系统管理员必须手工创建该目录。

1577 kern_mount()在内部挂载一个文件系统。换言之,该文件系统被挂载并被激活,但在 VFS 中不被任何用户可见。其挂载点是 shm_mnt,位于 shmem.c 文件中,其类型是 struct vfsmount。在后期需要搜索文件系统并卸载这个变量。

1578~1582 确保正确地卸载文件系统,但是如果失败,则跳转到 out1,解除文件系统的注册,并返回错误。

1585 函数 shmem_set_size()(见 L. 1.3 小节)用于设置文件系统中创建的块数及索引节点数的最大值。

1598 在这种情况下,module_init()表明了在载入模块上应当调用 init_shmem_fs()以及如何直接编译进内核,在系统启动时调用这个函数。

L. 1.2 函数: shmem_read_super() (mm/shmem.c)

这是文件系统提供的回调函数,用于读取超级块。对于普通的文件系统,这可以从磁盘读取细节信息,但是,由于这个文件系统基于 RAM,相反它产生一个 struct super_block。

```

1452 static struct super_block * shmem_read_super(struct super_block * sb,
                                                 void * data, int silent)
1453 {
1454     struct inode * inode;
1455     struct dentry * root;
1456     unsigned long blocks, inodes;
1457     int mode = S_IRWXUGO | S_ISVTX;
1458     uid_t uid = current->fsuid;
1459     gid_t gid = current->fsgid;
1460     struct shmem_sb_info * sbinfo = SHMEM_SB(sb);
1461     struct sysinfo si;
1462
1463     /*
1464     * Per default we only allow half of the physical ram per
1465     * tmpfs instance

```

```

1466 */
1467 si_meminfo(&si);
1468 blocks = inodes = si.totalram / 2;
1469
1470 #ifdef CONFIG_TMPFS
1471 if (shmem_parse_options(data, &mode, &uid, &gid, &blocks, &inodes))
1472 return NULL;
1473 #endif
1474
1475 spin_lock_init(&sbinfo->stat_lock);
1476 sbinfo->max_blocks = blocks;
1477 sbinfo->free_blocks = blocks;
1478 sbinfo->max_inodes = inodes;
1479 sbinfo->free_inodes = inodes;
1480 sb->s_maxbytes = SHMEM_MAX_BYTES;
1481 sb->s_blocksize = PAGE_CACHE_SIZE;
1482 sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
1483 sb->s_magic = TMPFS_MAGIC;
1484 sb->s_op = &shmem_ops;
1485 inode = shmem_get_inode(sb, S_IFDIR | mode, 0);
1486 if (!inode)
1487 return NULL;
1488
1489 inode->i_uid = uid;
1490 inode->i_gid = gid;
1491 root = d_alloc_root(inode);
1492 if (!root) {
1493 iput(inode);
1494 return NULL;
1495 }
1496 sb->s_root = root;
1497 return sb;
1498 }

```

1471 参数如下所示：

- sb 是产生的超级块。
- data 包括了一些参数。
- silent 在这个函数中未使用。

1457~1459 设置缺省模式,uid 和 gid。这些可能覆盖挂载选项中的参数。

1460 每个 super_block 都允许具有一个特定文件系统的结构,该结构包括一个称为 super_block->u 的联合结构。宏 SHMEM_SB()返回联合结构中所包含的 struct shmem_sb_info。

1467 si_meminfo()产生 struct sysinfo,包括了全部内存,可用内存和已用内存的统计数据。这个函数在 arch/i386/mm/init.c 文件中进行了定义,它是架构相关的。

1468 缺省情况下,只允许文件系统消耗物理内存的一半。

1471~1472 如果 tmpfs 可用,这将解析挂载选项,并允许覆盖缺省值。

1475 获取锁保护的 sbinfo,它是 super_block 中的 struct shmem_sb_info。

1483 产生 sb 和 sbinfo 字段。

1484 shmem_ops 是超级块结构的函数指针,用于重新挂载文件系统和删除索引节点。

1485~1487 这个块分配特定的索引节点,用于表示文件系统的根节点。

1489~1490 设置新文件系统的根部 uid 和.gid。

1496 设置根索引节点至 super_block 中。

1497 返回产生的超级块。

L. 1.3 函数: shmem_set_size() (mm/shmem.c)

这个函数更新文件系统中可用块和索引节点的数量。在文件系统挂载和卸载时进行设置。

```

861 static int shmem_set_size(struct shmem_sb_info * info,
862                           unsigned long max_blocks,
863                           unsigned long max_inodes)
864 {
865     int error;
866     unsigned long blocks, inodes;
867     spin_lock(&info->stat_lock);
868     blocks = info->max_blocks - info->free_blocks;
869     inodes = info->max_inodes - info->free_inodes;
870     error = -EINVAL;
871     if (max_blocks < blocks)
872         goto out;
873     if (max_inodes < inodes)
874         goto out;
875     error = 0;
876     info->max_blocks = max_blocks;

```

```

877 info->free_blocks = max_blocks - blocks;
878 info->max_inodes = max_inodes;
879 info->free_inodes = max_inodes - inodes;
880 out:
881 spin_unlock(&info->stat_lock);
882 return error;
883 }

```

861 这些参数描述了文件系统超级块的信息,块的最大数量(max_blocks)和索引节点的最大数量(max_inodes)。

867 锁定超级块信息自旋锁。

868 计算文件系统中当前使用的块数。在初始挂载时,这并不重要,然而,如果重新挂载文件系统,该函数必须保证新的文件系统不会太小。

869 计算当前使用的索引节点数。

871~872 如果重新挂载的文件系统没有足够的块存放当前信息,则跳转到 out,并返回-EINVAL。

873~874 同样地,确保是否有足够的索引节点,否则返回-EINVAL。

875 可以安全地挂载文件系统,因此这里设置 error 为 0 表示操作成功。

876~877 设置文件系统超级块信息结构的最大块数和可用块数。

878~879 设置最大索引节点数和可用节点数。

881 为文件系统超级块信息结构解锁。

882 成功则返回 0,否则返回-EINVAL。

L. 2 在 tmpfs 中创建文件

L. 2. 1 函数: shmem_create() (mm/shmem.c)

这是创建新文件时位于最顶层的函数。

```

1164 static int shmem_create(struct inode * dir,
                           struct dentry * dentry,
                           int mode)
1165 {
1166     return shmem_mknod(dir, dentry, mode | S_IFREG, 0);
1167 }

```

1164 参数如下所示:

- dir 是新文件创建时的目录索引节点。
- entry 是新文件创建时的目录节点。
- mode 是传递给开放系统调用的标志位。

1166 调用 shmem_mknod()(见 L. 2.2 小节), 并添加 S_IFREG 标志位到模式标志位, 以此创建一个常规文件。

L. 2.2 函数: shmem_mknod() (mm/shmem.c)

```

1139 static int shmem_mknod(struct inode * dir,
  struct dentry * dentry,
  int mode, int dev)

1140 {
1141 struct inode * inode = shmem_get_inode(dir->i_sb, mode, dev);
1142 int error = -ENOSPC;
1143
1144 if (inode) {
1145 dir->i_size += BOGO_DIRENT_SIZE;
1146 dir->i_ctime = dir->i_mtime = CURRENT_TIME;
1147 d_instantiate(dentry, inode);
1148 dget(dentry); /* Extra count - pin the dentry in core */
1149 error = 0;
1150 }
1151 return error;
1152 }
```

1141 调用 shmem_get_inode()(见 L. 2.3 小节)创建一个新的索引节点。

1144 如果成功创建索引节点, 则更新目录统计数据并实例化新文件。

1145 更新目录的大小。

1146 更新 ctime 和 mtime 字段。

1147 实例化索引节点。

1148 对目录项进行引用, 以阻止在页面换出时意外地回收了目录项。

1149 表明调用成功结束。

1151 返回成功, 否则返回-ENOSPC。

L. 2.3 函数: shmem_get_inode() (mm/shmem.c)

```

809 struct inode * shmem_get_inode(struct super_block * sb,
810                                int mode,
811                                int dev)
812
813 struct inode * inode;
814 struct shmem_inode_info * info;
815 struct shmem_sb_info * sbinfo = SHMEM_SB(sb);
816
817 spin_lock(&sbinfo->stat_lock);
818 if (! sbinfo->free_inodes) {
819     spin_unlock(&sbinfo->stat_lock);
820     return NULL;
821 }
822
823 sbinfo->free_inodes--;
824 spin_unlock(&sbinfo->stat_lock);
825
826 inode = new_inode(sb);

```

这个函数用于更新空闲索引节点数,用 new_inode()分配一个索引节点。

815 获取 sbinfo 自旋锁,因为它即将被更新。

816~819 确保有空闲的索引节点,如果没有,则返回 NULL。

820~821 更新空闲索引节点计数并释放锁。

823 new_inode()处于文件系统层面,并在<linux/fs.h>中声明。它如何运作的详细情况不在本文档讨论范围内,但其概要内容很简单。它从 slab 分配器中分配一个索引节点,并将各字段赋予 0,并根据超级块中的信息产生 inode->i_sb,inode->i_dev 和 inode->i_blkbits。

```

824 if (inode) {
825     inode->i_mode = mode;
826     inode->i_uid = current->fsuid;
827     inode->i_gid = current->fsgid;
828     inode->i_blksize = PAGE_CACHE_SIZE;
829     inode->i_blocks = 0;
830     inode->i_rdev = NODEV;
831     inode->i_mapping->a_ops = &shmem_aops;
832     inode->i_atime = inode->i_mtime
833                           = inode->i_ctime

```

```

        = CURRENT_TIME;

833 info = SHMEM_I(inode);
834 info->inode = inode;
835 spin_lock_init(&info->lock);
836 switch (mode & S_IFMT) {
837 default:
838     init_special_inode(inode, mode, dev);
839     break;
840 case S_IFREG:
841     inode->i_op = &shmem_inode_operations;
842     inode->i_fop = &shmem_file_operations;
843     spin_lock(&shmem_ilock);
844     list_add_tail(&info->list, &shmem_inodes);
845     spin_unlock(&shmem_ilock);
846     break;
847 case S_IFDIR:
848     inode->i_nlink++;
849     /* Some things misbehave if size == 0 on a directory */
850     inode->i_size = 2 * BOGO_DIRENT_SIZE;
851     inode->i_op = &shmem_dir_inode_operations;
852     inode->i_fop = &dcache_dir_ops;
853     break;
854 case S_IFLNK:
855     break;
856 }
857 }
858 return inode;
859 }

```

824~858 如果创建成功则填充索引节点各字段。

825~830 填充基本的索引节点信息。

831 设置 address_space_operations 使用 shmem_aops, 后者建立函数 shmem_writepage() (见 L. 6.1 小节) 用于 address_space 的页面回写回调函数。

832~834 填充更多的基本信息。

835~836 初始化索引节点的 semaphore 信号量和自旋锁。

836~856 确定如何根据传入的模式信息填充剩余的字段。

838 在这种情况下, 创建特定的索引节点。尤其是在挂载文件系统时和创建根索引节点时。

840~846 为常规文件创建索引节点。这里主要考虑的问题是设置 `inode->i_op` 和 `inode->i_fop` 字段分别为 `shmem_inode_operations` 和 `shmem_file_operations`。

847~852 为新目录创建索引节点。更新 `i_nlink` 和 `i_size` 字段以显示增加的文件数量和目录大小。这里主要考虑的问题是设置 `inode->i_op` 和 `inode->i_fop` 字段分别为 `shmem_dir_inode_operations` 和 `dcach_dir_ops`。

854~855 如果链接了文件, 由于它由父函数 `shmem_link()` 操作, 因此它现在什么也不是。

858 返回新索引节点, 如果没有创建则返回 `NULL`。

L. 3 tmpfs 中的文件操作

L. 3. 1 内存映射

用于映射虚拟文件到内存。惟一要做的改变是更新 VMA 的 `vm_operations_struct` 字段使用异常的等价 `shmemfs`。

L. 3. 1. 1 函数: `shmem_mmap()` (mm/shmem.c)

```

796 static int shmem_mmap(struct file * file, struct vm_area_struct * vma)
797 {
798     struct vm_operations_struct * ops;
799     struct inode * inode = file->f_dentry->d_inode;
800
801     ops = &shmem_vm_ops;
802     if (!S_ISREG(inode->i_mode))
803         return -EACCES;
804     UPDATE_ATIME(inode);
805     vma->vm_ops = ops;
806     return 0;
807 }
```

801 操作目前为虚拟文件系统所使用的 `vm_operations_struct`。

802 确保索引节点映射的是常规文件。如果不是, 则返回-EACCESS。

804 更新索引节点的 `atime`, 显示它是否已经被访问。

805 更新 `vma->vm_ops`, 这样 `shmem_nopage()` (见 L. 5. 1. 1) 可以用于处理映射中的缺页中断。

L. 3.2 读取文件

L. 3.2.1 函数: shmem_file_read() (mm/shmem.c)

这是读取 tmpfs 文件时所调用的位于最顶层的函数。

```

1088 static ssize_t shmem_file_read(struct file *filp, char *buf,
1089                                size_t count, loff_t *ppos)
1090 {
1091     read_descriptor_t desc;
1092
1093     if ((ssize_t) count < 0)
1094         return -EINVAL;
1095
1096     if (!count)
1097         return 0;
1098
1099     desc.written = 0;
1100    desc.count = count;
1101    desc.buf = buf;
1102    desc.error = 0;
1103
1104    do_shmem_file_read(filp, ppos, &desc);
1105    if (desc.written)
1106        return desc.written;
1107    return desc.error;
1108 }

```

1088 参数如下所示：

- filp 是指向被读取文件的指针。
- buf 是应当填充的缓冲区。
- count 是应当读取的字节数。
- ppos 是当前位置。

1092~1093 计数不可能为负数。

1094~1095 access_ok() 确保安全地写 count 数量的字节到用户空间缓冲区。如果不能，则返回-EFAULT。

1099~1102 初始化 read_descriptor_t 结构，该结构最终传递给 file_read_actor()

(见L. 3.2.3)。

1104 调用 do_shmem_file_read()开始执行实际的读操作。

1105~1106 返回写到用户空间缓冲区的字节数。

1107 如果没有写任何东西,而返回错误。

L. 3.2.2 函数: do_shmem_file_read() (mm/shmem.c)

这个函数通过 shmem_getpage()找回读取文件所需的页面数,并调用 file_read_actor()复制数据到用户空间。

```

1003 static void do_shmem_file_read(struct file * filp,
1004                                   loff_t * ppos,
1005                                   read_descriptor_t * desc)
1006
1007 struct inode * inode = filp->f_dentry->d_inode;
1008 struct address_space * mapping = inode->i_mapping;
1009 unsigned long index, offset;
1010
1011 index = * ppos >> PAGE_CACHE_SHIFT;
1012 offset = * ppos & ~PAGE_CACHE_MASK;
1013
1014 for (;;) {
1015     struct page * page = NULL;
1016     unsigned long end_index, nr, ret;
1017     if (index > end_index)
1018         break;
1019     if (index == end_index) {
1020         nr = inode->i_size & ~PAGE_CACHE_MASK;
1021         if (nr <= offset)
1022             break;
1023     }
1024
1025     desc->error = shmem_getpage(inode, index, &page, SGP_READ);
1026     if (desc->error) {
1027         if (desc->error == -EINVAL)
1028             desc->error = 0;
1029         break;
1030     }

```

```

1031
1036 nr = PAGE_CACHE_SIZE;
1037 end_index = inode->i_size >> PAGE_CACHE_SHIFT;
1038 if (index == end_index) {
1039 nr = inode->i_size & ~PAGE_CACHE_MASK;
1040 if (nr <= offset) {
1041 page_cache_release(page);
1042 break;
1043 }
1044 }
1045 nr -= offset;
1046
1047 if (page != ZERO_PAGE(0)) {
1053 if (mapping->i_mmap_shared != NULL)
1054 flush_dcache_page(page);
1055 /*
1056 * Mark the page accessed if we read the
1057 * beginning or we just did an lseek.
1058 */
1059 if (! offset || ! filp->f_reada)
1060 mark_page_accessed(page);
1061 }
1062
1073 ret = file_read_actor(desc, page, offset, nr);
1074 offset += ret;
1075 index += offset >> PAGE_CACHE_SHIFT;
1076 offset &= ~PAGE_CACHE_MASK;
1077
1078 page_cache_release(page);
1079 if (ret != nr || ! desc->count)
1080 break;
1081 }
1082
1083 * ppos = ((loff_t) index << PAGE_CACHE_SHIFT) + offset;
1084 filp->f_reada = 1;
1085 UPDATE_ATIME(inode);
1086 }

```

1005~1006 找回 inode 和使用 struct file 的 mapping。

1009 index 是文件中包含数据的页面的索引。

1010 offset 是在当前被读取页面中的偏移量。

1012~1081 循环直到读取完请求的字节数。nr 是当前页面中还需要读取的字节数。desc->count 初始为需要读取的字节数，并由 file_read_actor() (见 L. 3.2.3) 减小。

1016~1018 end_index 是文件中最后页面的索引。当到达文件尾部时停止。

1019~1023 当到达最后一个页面时，设置 nr 为当前页面中还需要读取的字节数。如果文件指针在 nr 后面，则停止，因为没有更多的数据供读取。这有可能发生在文件被截断时的情况。

1025~1030 shmem_getpage() (见 L. 5.1.2) 查找被请求页在页面高速缓存，交换缓存中的位置。如果错误发生，则记录错误到 desc->error 并返回。

1036 nr 是页面中必须读取的字节数，因此初始化为一个页面的大小，这样就可以读取整个页面。

1037 初始化 end_index，它是文件中最后一个页面的索引。

1038~1044 如果是文件的最后一个页面，则更新 nr 为页面的字节数。如果 nr 当前在文件尾部之后（可能发生在文件截取的情况），则释放页面的引用（由 shmem_getpage() 使用），并退出循环。

1045 更新需要读取的字节数。请记得 offset 是在页面中当前的文件读取处。

1047~1061 如果读取的页面不是全局零页面，则需要注意调用 flush_dcache_page() 所引起的别名混淆隐患。如果是第一次读取的页面或者仅仅发生 lseek() (f_reada 是 0)，则用 mark_page_accessed() 标记页面被访问过。

1073 调用 file_read_actor() (见 L. 3.2.3) 复制数据到用户空间。它返回复制的字节数，并更新用户缓冲指针及剩下的计数。

1074 更新页面中读取的偏移量。

1075 如果可能，移动索引至下一个页面。

1076 确保 offset 是页面中的偏移量。

1078 释放被复制页面的引用。该引用由 shmem_getpage() 使用。

1079~1080 如果已经读取完请求的字节数，则返回。

1083 更新文件指针。

1084 允许文件预读。

1085 更新 inode 的访问计数，因为它已经被读取。

L. 3.2.3 函数: file_read_actor() (mm/filemap.c)

这个函数用于从页面复制数据到用户空间缓冲区。它最终由多个函数所调用，包括 generic_file_read() 和 shmem_file_read()。

1669 int file_read_actor(read_descriptor_t * desc,

```

    struct page * page,
    unsigned long offset,
    unsigned long size)

1670 {
1671 char * kaddr;
1672 unsigned long left, count = desc->count;
1673
1674 if (size > count)
1675 size = count;
1676
1677 kaddr = kmap(page);
1678 left = __copy_to_user(desc->buf, kaddr + offset, size);
1679 kunmap(page);
1680
1681 if (left) {
1682 size -= left;
1683 desc->error = -EFAULT;
1684 }
1685 desc->count = count - size;
1686 desc->written += size;
1687 desc->buf += size;
1688 return size;
1689 }

```

1669 参数如下所示：

- desc 是个包含读信息的结构,包括缓冲区和从文件读取的总字节数。
- page 是复制到用户空间且包含文件数据的页面。
- offset 是页面中复制的偏移量。
- size 是从页面读取的字节数。

1672 count 现在是从文件读取的字节数。

1674~1675 确保读取的字节数不会比请求的多。

1677 调用 kmap()映射页面到低端内存。见 I.1.1 小节。

1678 从内核页面复制数据到用户空间缓冲区。

1679 解除页面映射。见 I.3.1 小节。

1681~1684 如果所有的字节都没有被复制,那肯定是缓冲区不能访问。这更新 size,因此 desc->count 可以反映读操作中还有多少字节待复制。返回-EFAULT 到执行读操作的进程。

1685~1687 更新 desc 结构,以表明当前的读状态。

1688 返回写入用户空间缓冲区的字节数。

L. 3.3 写入文件

L. 3.3.1 函数: shmem_file_write() (mm/shmem.c)

```

925 shmem_file_write(struct file * file, const char * buf,
926             size_t count, loff_t * ppos)
927 {
928     struct inode * inode = file->f_dentry->d_inode;
929     loff_t pos;
930     unsigned long written;
931     int err;
932
933     if ((ssize_t) count < 0)
934         return -EINVAL;
935
936     if (! access_ok(VERIFY_READ, buf, count))
937         return -EFAULT;
938
939     down(&inode->i_sem);
940
941     pos = * ppos;
942     written = 0;
943
944     err = precheck_file_write(file, inode, &count, &pos);
945     if (err || ! count)
946         goto out;
947
948     remove_suid(inode);
949     inode->i_ctime = inode->i_mtime = CURRENT_TIME;

```

这块是函数导引。

927 获取描述哪些文件被写入的索引节点。

932~933 如果用户尝试写负的字节数,则返回-EINVAL。

935~936 如果用户空间缓冲区不可访问,则返回-EINVAL。

938 获取保护索引节点的 semaphore 信号量。

940 记录写入开始的位置。

941 初始化写入字节为 0。

943 precheck_file_write() 执行一系列的检查以确保写操作的正确进行。检查包括以 append 模式打开文件时更新 pos 为文件尾部，并保证系统不会超过进程的限定。

944~945 如果不能执行写操作，则跳转到 out。

947 如果设置了 SUID 位则清除它。

948 更新索引节点的 ctime 和 mtime。

```

950 do {
951     struct page * page = NULL;
952     unsigned long bytes, index, offset;
953     char * kaddr;
954     int left;
955
956     offset = (pos & (PAGE_CACHE_SIZE -1)); /* Within page */
957     index = pos >> PAGE_CACHE_SHIFT;
958     bytes = PAGE_CACHE_SIZE - offset;
959     if (bytes > count)
960         bytes = count;
961
962     /*
963     * We dont hold page lock across copy from user -
964     * what would it guard against? - so no deadlock here.
965     */
966
967     err = shmem_getpage(inode, index, &page, SGP_WRITE);
968     if (err)
969         break;
970
971     kaddr = kmap(page);
972     left = __copy_from_user(kaddr + offset, buf, bytes);
973     kunmap(page);
974
975     written += bytes;
976     count -= bytes;
977     pos += bytes;
978     buf += bytes;
979     if (pos > inode->i_size)
980         inode->i_size = pos;
981

```



```

982 flush_dcache_page(page);
983 SetPageDirty(page);
984 SetPageReferenced(page);
985 page_cache_release(page);
986
987 if (left) {
988 pos -= left;
989 written -= left;
990 err = -EFAULT;
991 break;
992 }
993 } while (count);
994
995 *ppos = pos;
996 if (written)
997 err = written;
998 out:
999 up(&inode->i_sem);
1000 return err;
1001 }

```

950~993 循环直至所有的请求都进行了写操作。

956 设置 offset 为当前被写页面内的偏移量。

957 index 是当前被写文件中页面的索引。

958 bytes 是当前页面中还需要写入的字节数。

959~960 如果 bytes 表明需要写入比请求(count)更多的字节数, 则设置 bytes 为 count。

967~969 定位要写入的页面。SGP_WRITE 标志位表明了如果某个页面还不存在则需要分配它。如果无法找到某页面或分配页面, 则跳出循环。

971~973 在再次解除页面映射以前, 映射被写入且从用户空间缓冲区复制数据的页面。

975 更新写入的字节数。

976 更新还需要写入的字节数。

977 更新在文件中的位置。

978 更新用户空间缓冲区中的指针。

979~980 如果文件变大, 则更新 inode->i_size。

982 刷新 dcache 以避免别名混淆隐患。

983~984 设置该页面为脏且被引用过。

985 释放 shmem_getpage()对页面使用的引用。

987~992 如果所有请求的字节都没有从用户空间缓存区读出,则更新写操作统计数据和文件中的位置及缓冲区中的位置。

995 更新文件指针。

996~997 如果所有的字节都没有写入,则设置错误返回变量。

999 释放索引节点 semaphore 信号量。

1000 返回成功或者返回还需要写入的字节数。

L. 3.4 符号链接

L. 3.4.1 函数: shmem_symlink() (mm/shmem.c)

这个函数用于创建符号链接 symname 并决定在什么地方存储该信息。如果链接的名字足够小则存储在索引节点中,否则存储在页面帧当中。

```

1272 static int shmem_symlink(struct inode * dir,
1273                         struct dentry * dentry,
1274                         const char * symname)
1275 {
1276     int error;
1277     struct inode * inode;
1278     struct page * page = NULL;
1279     char * kaddr;
1280
1281     len = strlen(symname) + 1;
1282     if (len > PAGE_CACHE_SIZE)
1283         return -ENAMETOOLONG;
1284
1285     inode = shmem_get_inode(dir->i_sb, S_IFLNK|S_IRWXUGO, 0);
1286     if (! inode)
1287         return -ENOSPC;
1288
1289     info = SHMEM_I(inode);
1290     inode->i_size = len-1;

```

这块执行基本的全部检查,并为符号链接创建一个新的索引节点。

1272 参数 symname 是所创建链接的名字。

1281 计算链接的长度(len)。

1282~1283 如果名字比一个页面长,则返回-ENAMETOOLONG。

1285~1287 分配一个新的 inode。分配失败则返回-ENOSPC。

1289 获取私有信息结构。

1290 索引节点的大小即链接的长度。

```

1291 if (len <= sizeof(struct shmem_inode_info)) {
1292 /* do it inline */
1293 memcpy(info, symname, len);
1294 inode->i_op = &shmem_symlink_inline_operations;
1295 } else {
1296 error = shmem_getpage(inode, 0, &page, SGP_WRITE);
1297 if (error) {
1298 iput(inode);
1299 return error;
1300 }
1301 inode->i_op = &shmem_symlink_inode_operations;
1302 spin_lock(&shmem_ilock);
1303 list_add_tail(&info->list, &shmem_inodes);
1304 spin_unlock(&shmem_ilock);
1305 kaddr = kmap(page);
1306 memcpy(kaddr, symname, len);
1307 kunmap(page);
1308 SetPageDirty(page);
1309 page_cache_release(page);
1310 }

```

这块用于存储链接信息。

1291~1295 如果名字的长度小于 shmem_inode_info 所使用的空间,则复制名字到保留给私有结构的空间中。

1294 设置 inode->i_op 为 shmem_symlink_inline 操作,这样可以知道链接名字在索引节点中。

1296 用 shmem_getpage_locked 分配页面。

1297~1300 如果发生错误,则解除索引节点的引用并返回错误。

1301 使用 shmem_symlink_inode_operations,这样可以知道链接信息被包括在页面中。

1302 shmem_ilock 是一个全局锁,用于保护索引节点的全局链表,这个链表是由私有的信息结构 info->list 字段链接起来的。

1303 添加新的索引节点到全局链表。

1304 释放 shmem_ilock。

- 1305 映射页面。
- 1306 复制链接信息。
- 1307 解除页面映射。
- 1308 设置页面为脏。
- 1309 释放对页面的引用。

```

1311 dir->i_size += BOGO_DIRENT_SIZE;
1312 dir->i_ctime = dir->i_mtime = CURRENT_TIME;
1313 d_instantiate(dentry, inode);
1314 dget(dentry);
1315 return 0;
1316 }
```

1311 增加目录的大小,因为添加了新的索引节点。BOGO_DIRENT_SIZE 是索引节点的伪大小,这样 !s 操作看起来更好一些。

- 1312 更新 i_ctime 和 i_mtime。
- 1313~1314 实例化索引节点。
- 1315 返回成功。

L. 3.4.2 函数: shmem_readlink_inline() (mm/shmem.c)

```

1318 static int shmem_readlink_inline(struct dentry *dentry,
                                     char *buffer, int buflen)
1319 {
1320 return vfs_readlink(dentry, buffer, buflen,
                      (const char *)SHMEM_I(dentry->d_inode));
1321 }
```

1320 链接名字被包括在索引节点中,这样可以将它作为 vfs_readlink() 的参数传递给 VFS 层。

L. 3.4.3 函数: shmem_follow_link_inline() (mm/shmem.c)

```

1323 static int shmem_follow_link_inline(struct dentry *dentry,
                                         struct nameidata *nd)
1324 {
1325 return vfs_follow_link(nd,
                          (const char *)SHMEM_I(dentry->d_inode));
1326 }
```

1325 链接名字被包括在索引节点中,这样可以将它作为 vfs_followlink() 的参数传递给

VFS 层。

L. 3.4.4 函数: shmem_readlink() (mm/shmem.c)

```

1328 static int shmem_readlink(struct dentry * dentry,
1329                             char * buffer, int buflen)
1330 {
1331     struct page * page = NULL;
1332     int res = shmem_getpage(dentry->d_inode, 0, &page, SGP_READ);
1333     if (res)
1334         return res;
1335     res = vfs_readlink(dentry, buffer, buflen, kmap(page));
1336     kunmap(page);
1337     page_cache_release(page);
1338     return res;
1339 }
```

1331 链接名字被包括在和 symlink 关联的页面中, 它可以调用 shmem_getpage() (见 L. 5.1.2) 获取指针。

1332~1333 如果发生错误, 则返回 NULL。

1334 调用 kmap() (见 I. 1.1 小节) 映射页面, 并将它作为参数传递给 vfs_readlink()。该链接位于页面首部。

1335 解除页面映射。

1336 标记页面被访问过。

1337 解除调用 shmem_getpage() 对页面的引用。

1338 返回该链接。

```

1231 static int shmem_follow_link(struct dentry * dentry,
1232                               struct nameidata * nd)
1233 {
1234     struct page * page;
1235     int res = shmem_getpage(dentry->d_inode, 0, &page);
1236     if (res)
1237         return res;
1238     res = vfs_follow_link(nd, kmap(page));
1239     kunmap(page);
1240     page_cache_release(page);
1241     return res;
```

1242 }

1234 由于链接名字在页面中,因此可以调用 shmem_getpage()获取页面。

1235~1236 如果发生错误,则返回错误。

1238 映射页面并作为指针传递给 vfs_follow_link()。

1239 解除页面映射。

1240 解除对页面的引用。

1241 返回成功。

L. 3.5 同步文件

L. 3.5.1 函数: shmem_sync_file() (mm/shmem.c)

这个函数简单返回 0,因为文件只存在于内存而不需要同步文件到磁盘。

```
1446 static int shmem_sync_file(struct file * file,
    struct dentry * dentry,
    int datasync)
1447 {
1448     return 0;
1449 }
```

L. 4 tmpfs 中的索引节点操作

L. 4.1 截取

L. 4.1.1 函数: shmem_truncate() (mm/shmem.c)

当调用这个函数的时候,inode->i_size 被 vmtruncate()设置为新大小。这个函数用于创建或移除页面,以此设置文件的大小。

```
351 static void shmem_truncate(struct inode * inode)
352 {
353     struct shmem_inode_info * info = SHMEM_I(inode);
354     struct shmem_sb_info * sbinfo = SHMEM_SB(inode->i_sb);
355     unsigned long freed = 0;
356     unsigned long index;
357 }
```

```

358 inode->i_ctime = inode->i_mtime = CURRENT_TIME;
359 index = (inode->i_size + PAGE_CACHE_SIZE - 1)
            >> PAGE_CACHE_SHIFT;
360 if (index >= info->next_index)
361     return;
362
363 spin_lock(&info->lock);
364 while (index < info->next_index)
365     freed += shmem_truncate_indirect(info, index);
366 BUG_ON(info->swapped > info->next_index);
367 spin_unlock(&info->lock);
368
369 spin_lock(&sbinfo->stat_lock);
370 sbinfo->free_blocks += freed;
371 inode->i_blocks -= freed * BLOCKS_PER_PAGE;
372 spin_unlock(&sbinfo->stat_lock);
373 }

```

353 用 SHMEM_I() 获取索引节点的私有文件系统信息。

354 获取超级块私有信息。

358 更新索引节点的 ctime 和 mtime。

359 获取文件新的尾部页面的索引。原来的大小存储在 info->next_index 中。

360~361 如果文件扩大了, 只需要返回, 因为全局 0 页面将用于表示扩大的区域。

363 获取私有 info 自旋锁。

364~365 继续调用 shmem_truncate_indirect() 直至文件被截取为需要的大小。

366 如果 shmem_info_info 结构表示的换出页面多于文件中的页面, 这是个 bug。

367 释放私有 info 自旋锁。

369 获取超级块私有 info 自旋锁。

370 更新可用空闲块的数量。

371 更新索引节点所使用的块数量。

372 释放超级块私有 info 自旋锁。

L. 4.1.2 函数: shmem_truncate_indirect() (mm/shmem.c)

这个函数定位索引节点中最后两次间接访问的块, 并调用 shmem_truncate_direct() 截取它。

```

308 static inline unsigned long
309 shmem_truncate_indirect(struct shmem_inode_info *info,

```

```
        unsigned long index)

310 {
311     swp_entry_t *** base;
312     unsigned long baseidx, start;
313     unsigned long len = info->next_index;
314     unsigned long freed;
315
316     if (len <= SHMEM_NR_DIRECT) {
317         info->next_index = index;
318         if (! info->swapped)
319             return 0;
320         freed = shmem_free_swp(info->i_direct + index,
321                                info->i_direct + len);
322         info->swapped -= freed;
323         return freed;
324     }
325
326     if (len <= ENTRIES_PER_PAGEPAGE/2 + SHMEM_NR_DIRECT) {
327         len -= SHMEM_NR_DIRECT;
328         base = (swp_entry_t *** ) &info->i_indirect;
329         baseidx = SHMEM_NR_DIRECT;
330     } else {
331         len -= ENTRIES_PER_PAGEPAGE/2 + SHMEM_NR_DIRECT;
332         BUG_ON(len > ENTRIES_PER_PAGEPAGE * ENTRIES_PER_PAGE/2);
333         baseidx = len - 1;
334         baseidx -= baseidx % ENTRIES_PER_PAGEPAGE;
335         base = (swp_entry_t *** ) info->i_indirect +
336                 ENTRIES_PER_PAGE/2 + baseidx/ENTRIES_PER_PAGEPAGE;
337         len -= baseidx;
338         baseidx += ENTRIES_PER_PAGEPAGE/2 + SHMEM_NR_DIRECT;
339     }
340
341     if (index > baseidx) {
342         info->next_index = index;
343         start = index - baseidx;
344     } else {
345         info->next_index = baseidx;
346         start = 0;
347     }

```

```
348 return *base? shmem_truncate_direct(info, base, start, len): 0;
349 }
```

313 len 是当前文件中被第 2 次使用的最后页面。

316~324 如果文件很小且所有的项都存储在直接块信息中, 则调用 shmem_free_swp() 并传递 info→i_direct 中第一个交换项和项数目给这个函数, 供截取之用。

326~339 被截取的页面位于间接块的某处。这个代码部分用于计算 3 个变量: base, baseidx 和 len。base 是页面的首部, 该页面包括将被截取交换项的指针。baseidx 是被使用间接块中第一个项的页索引, 而 len 是这里将被截取的项数目。

326~330 计算两次间接块的变量。接着设置 base 到 info→i_indirect 首部的交换项。而 info→i_indirect 首部的页面索引 baseidx 为 SHMEM_NR_DIRECT。在这里, len 为文件中页面的数量, 因此减去直接块的数量就是剩下页面的数量。

330~339 如果不是这样, 那么这是一个 3 级索引块, 因此必须在计算 base, baseidx 和 len 以前遍历下一级。

341~344 如果在截取后文件变大, 则更新 next_index 为文件新尾部, 并设置 start 为间接块的首部。

344~347 如果在截取后文件变小, 则移动当前文件尾部至将被截取的间接块的首部。

348 如果在 base 处有块, 则调用 shmem_truncate_direct() 截取其中的页面。

L. 4.1.3 函数: shmem_truncate_direct() (mm/shmem.c)

这个函数用于循环遍历间接块, 并对包含被截取交换向量的每个页面调用 shmem_free_swp。

```
264 static inline unsigned long
265 shmem_truncate_direct(struct shmem_inode_info *info,
266                         swp_entry_t ***dir,
267                         unsigned long start, unsigned long len)
268 {
269     swp_entry_t **last, **ptr;
270     unsigned long off, freed_swp, freed = 0;
271     off = start % ENTRIES_PER_PAGE;
272
273     for (ptr = *dir + start/ENTRIES_PER_PAGE;
274          ptr < last;
275          ptr++, off = 0) {
276         if (! *ptr)
277             continue;
```



```

276
277 if (info->swapped) {
278 freed_swp = shmem_free_swp(*ptr + off,
279 *ptr + ENTRIES_PER_PAGE);
280 info->swapped -= freed_swp;
281 freed += freed_swp;
282 }
283
284 if (! off) {
285 freed++;
286 free_page((unsigned long) *ptr);
287 *ptr = 0;
288 }
289 }
290
291 if (! start) {
292 freed++;
293 free_page((unsigned long) *dir);
294 *dir = 0;
295 }
296 return freed;
297 }

```

270 last 是被截取间接块中最后一个页面。

271 如果截取是部分截取而非全页截取, 则 off 是页面中的截取偏移量。

273~289 从 dir 的 startth 块开始截取, 直至最后一块。

274~275 如果这里没有页面, 则继续下一个。

277~282 如果 info 结构表明交换出属于这个索引节点的页面, 则调用 shmem_free_swp() 释放与该页面相关联的交换槽。如果释放了一个, 那么更新 infoswapped 并增加空闲页面的计数。

284~288 如果这不是部分截取, 则释放该页面。

291~295 如果整个间接块现在已释放, 则回收该页面。

296 返回被释放页面的数量。

L. 4. 1. 4 函数: shmem_free_swp() (mm/shmem.c)

这个函数释放起始项位于 dir 的交换项的 count。

```

240 static int shmem_free_swp(swp_entry_t *dir, swp_entry_t *edir)
241 {

```

```

242 swp_entry_t * ptr;
243 int freed = 0;
244
245 for (ptr = dir; ptr < edir; ptr++) {
246 if (ptr->val) {
247 free_swap_and_cache(*ptr);
248 *ptr = (swp_entry_t){0};
249 freed++;
250 }
251 }
252 return freed;
254 }

```

245~251 循环释放每个交换项。

246~250 如果存在交换项，则用 free_swap_and_cache() 释放它并设置交换项为 0。它增加被释放页面的数量。

252 返回被释放页面的总数量。

L. 4.2 链接

L. 4.2.1 函数: shmem_link() (mm/shmem.c)

这个函数创建一个从 dentry 到 old_dentry 的硬链接。

```

1172 static int shmem_link(struct dentry * old_dentry,
1173                      struct inode * dir,
1174                      struct dentry * dentry)
1175 {
1176 struct inode * inode = old_dentry->d_inode;
1177
1178 if (S_ISDIR(inode->i_mode))
1179 return -EPERM;
1180
1181 dir->i_size += BOGO_DIRENT_SIZE;
1182 inode->i_ctime = dir->i_ctime = dir->i_mtime = CURRENT_TIME;
1183 inode->i_nlink++;
1184 atomic_inc(&inode->i_count);
1185 dget(dentry);
1186 d_instantiate(dentry, inode);

```



```
1185 return 0;
1186 }
```

1174 获取与 old_dentry 对应的索引节点。

1176~1177 如果它链接到一个目录,则返回-EPERM。严格意义上,根目录应当允许硬链接目录,虽然并不推荐这么处理,但是在文件系统中创建循环时利用诸如 find 的操作时可能会找不到路径。所以 tmpfs 简单地不允许目录硬链接。

1179 为新链接增加目录的大小。

1180 更新目录的 mtime 和 ctime,并更新索引节点的 ctime。

1181 增加指向索引节点的链接数量。

1183 调用 dget()获取新 dentry 的额外引用。

1184 实例化新目录项。

1185 返回成功。

L. 4.3 解除链接

L. 4.3.1 函数: shmem_unlink() (mm/shmem.c)

```
1221 static int shmem_unlink(struct inode * dir,
                           struct dentry * dentry)
1222 {
1223 struct inode * inode = dentry->d_inode;
1224
1225 dir->i_size -= BOGO_DIRENT_SIZE;
1226 inode->i_ctime = dir->i_ctime = dir->i_mtime = CURRENT_TIME;
1227 inode->i_nlink--;
1228 dput(dentry);
1229 return 0;
1230 }
```

1223 获取解除链接的 dentry 的索引节点。

1225 更新目录索引节点的大小。

1226 更新 ctime 和 mtime 变量。

1227 减小索引节点的链接数量。

1228 调用 dput()减小 dentry 的引用计数。在引用计数为 0 时,这个函数也调用 iput()清除索引节点。

L. 4.4 创建目录

L. 4.4.1 函数: shmem_mkdir() (mm/shmem.c)

```

1154 static int shmem_mkdir(struct inode *dir,
1155     struct dentry *dentry,
1156     int mode)
1157 {
1158     int error;
1159
1160     if ((error = shmem_mknod(dir, dentry, mode | S_IFDIR, 0)))
1161         return error;
1162     dir->i_nlink++;
1163
1164     return 0;
1165 }
```

1158 调用 shmem_mknod()(见 L. 2.2 小节)创建一个特定文件。通过指定 S_IFDIR 标志位来创建一个目录。

1160 增加父目录的 i_nlink 字段。

L. 4.5 移除目录

L. 4.5.1 函数: shmem_rmdir() (mm/shmem.c)

```

1232 static int shmem_rmdir(struct inode *dir, struct dentry *dentry)
1233 {
1234     if (!shmem_empty(dentry))
1235         return -ENOTEMPTY;
1236
1237     dir->i_nlink--;
1238     return shmem_unlink(dir, dentry);
1239 }
```

1234~1235 用 shmem_empty() 检查目录是否为空。如果不是，则返回-ENOTEMPTY。

1237 减小父目录的 i_nlink 字段。

1238 返回 shmem_unlink()(见 L. 4.3.1)的结果，结果一般是删除该目录。

L. 4.5.2 函数 shmem_empty() (mm/shmem.c)

这个函数检查目录是否为空。

```

1201 static int shmem_empty(struct dentry * dentry)
1202 {
1203     struct list_head * list;
1204
1205     spin_lock(&dcache_lock);
1206     list = dentry->d_subdirs.next;
1207
1208     while (list != &dentry->d_subdirs) {
1209         struct dentry * de = list_entry(list,
1210                                         struct dentry, d_child);
1211
1212         if (shmem_positive(de)) {
1213             spin_unlock(&dcache_lock);
1214             return 0;
1215         }
1216         list = list->next;
1217     }
1218     spin_unlock(&dcache_lock);
1219     return 1;

```

1205 dcache_lock 虽然保护多个事物,但主要保护 dcache 的查找,因为这是该函数必须做的工作。

1208 循环遍历子目录链表并查找活动目录项,而子目录链表包含了所有的子目录项。如果找到,则表明目录不为空。

1209 获取子目录项。

1211 如果目录项有一个相关联的合法索引节点且目前通过 hash 查找到,则由 shmem_positive()(见 L. 4.5.3)返回。因为如果 hash 查找到,则意味着目录项是活动的,且目录不为空。

1212~1213 如果目录不为空,则释放自旋锁并返回。

1215 转到下一个子目录项。

1217~1218 目录为空。则释放自旋锁并返回。

L. 4.5.3 函数: shmem_positive() (mm/shmem.c)

```
1188 static inline int shmem_positive(struct dentry * dentry)
```

```

1189 {
1190 return dentry->d_inode && ! d_unhashed(dentry);
1191 }

```

1190 如果目录项有一个相关联的合法索引节点且目前通过 hash 查找到，则返回真。

L. 5 虚拟文件中的缺页中断

L. 5. 1 缺页中断时读取页面

L. 5. 1. 1 函数: shmem_nopage() (mm/shmem.c)

这是位于最高层的 nopage() 函数，在发生缺页中断时由 do_no_page() 调用。不论该异常是第一次异常还是由后援存储器引起的异常都将调用该函数。

```

763 struct page * shmem_nopage(struct vm_area_struct * vma,
                               unsigned long address,
                               int unused)
764 {
765 struct inode * inode = vma->vm_file->f_dentry->d_inode;
766 struct page * page = NULL;
767 unsigned long idx;
768 int error;
769
770 idx = (address - vma->vm_start) >> PAGE_SHIFT;
771 idx += vma->vm_pgoff;
772 idx >>= PAGE_CACHE_SHIFT - PAGE_SHIFT;
773
774 error = shmem_getpage(inode, idx, &page, SGP_CACHE);
775 if (error)
776 return (error == -ENOMEM) ? NOPAGE_OOM : NOPAGE_SIGBUS;
777
778 mark_page_accessed(page);
779 flush_page_to_ram(page);
780 return page;
781 }

```

763 这两个参数表示异常发生的 VMA 结构和异常发生的地址。

765 记录发生异常的索引节点。

770~772 在虚拟文件中将 idx 作为偏移量以 PAGE_SIZE 为单位进行计算。

772 考虑到页面高速缓存中的项大小可能与单个页面的大小不一致，则调整它。但是在这里没有什么不同。

774~775 shmem_getpage() (见 L. 5.1.2) 用于定位在 idx 的页面。

775~776 如果发生错误，则决定是返回一个 OOM 错误还是返回一个无效异常地址错误。

778 标记页面被访问过，这样将移动它到 LRU 链表首部。

779 flush_page_to_ram() 用于避免 dcache 别名混淆隐患。

780 返回发生异常的页面。

L. 5.1.2 函数: shmem_getpage() (mm/shmem.c)

```

583 static int shmem_getpage(struct inode * inode,
584                         unsigned long idx,
585                         struct page ** pagep,
586                         enum sgp_type sgp)
587 {
588     struct address_space * mapping = inode->i_mapping;
589     struct shmem_inode_info * info = SHMEM_I(inode);
590     struct shmem_sb_info * sbinfo;
591     struct page * filepage = * pagep;
592     struct page * swappage;
593     struct swp_entry_t * entry;
594     int error = 0;
595
596     if (idx >= SHMEM_MAX_INDEX)
597         return -EFBIG;
598
599     /* Normally, filepage is NULL on entry, and either found
600      * uptodate immediately, or allocated and zeroed, or read
601      * in under swappage, which is then assigned to filepage.
602      * But shmem_readpage and shmem_prepare_write pass in a locked
603      * filepage, which may be found not uptodate by other callers
604      * too, and may need to be copied from the swappage read in.
605
606     repeat:
607     if (! filepage)
608         filepage = find_lock_page(mapping, idx);

```

```

607 if (filepage && Page_Uptodate(filepage))
608     goto done;
609
610 spin_lock(&info->lock);
611 entry = shmem_swp_alloc(info, idx, sgp);
612 if (IS_ERR(entry)) {
613     spin_unlock(&info->lock);
614     error = PTR_ERR(entry);
615     goto failed;
616 }
617 swap = *entry;

```

583 参数如下所示：

- inode 是异常发生的索引节点。
- idx 是异常发生的文件中页面的索引。
- pagep (如果 NULL) 将成为异常页面(如果成功)。如果传入一个合法的页面, 这个函数要确保它是最新的。
- sgp 表明这是什么类型的访问, 即决定如何定位页面且如何返回。

586 SHMEM_I() 返回超级块信息中具有特定文件系统信息的 shmem_inode_info。

594~595 确保索引不会超过文件尾部。

605~606 如果 pagep 参数没有传入页面, 则尝试定位页面并用 find_lock_page() 锁住页面。

607~608 如果页面被找到且为最新的, 则跳转到 done, 因为这个函数不需要再做其他更多的工作。

610 锁住索引私有信息结构。

611 调用 shmem_swp_alloc() 搜索交换项得到 idx。如果它开始不存在, 则分配它。

612~616 如果发生错误, 则释放自旋锁并返回错误。

```

619 if (swap.val) {
620     /* Look it up and read it in.. */
621     swappage = lookup_swap_cache(swap);
622     if (!swappage) {
623         spin_unlock(&info->lock);
624         swapin_readahead(swap);
625         swappage = read_swap_cache_async(swap);
626     }
627     if (!swappage) {
628         spin_lock(&info->lock);
629         entry = shmem_swp_alloc(info, idx, sgp);

```



```

629 if (IS_ERR(entry))
630 error = PTR_ERR(entry);
631 else if (entry->val == swap.val)
632 error = -ENOMEM;
633 spin_unlock(&info->lock);
634 if (error)
635 goto failed;
636 goto repeat;
637 }
638 wait_on_page(swappage);
639 page_cache_release(swappage);
640 goto repeat;
641 }
642
643 /* We have to do this with page locked to prevent races */
644 if (TryLockPage(swappage)) {
645 spin_unlock(&info->lock);
646 wait_on_page(swappage);
647 page_cache_release(swappage);
648 goto repeat;
649 }
650 if (!Page_Uptodate(swappage)) {
651 spin_unlock(&info->lock);
652 UnlockPage(swappage);
653 page_cache_release(swappage);
654 error = -EIO;
655 goto failed;
656 }

```

在这块,页面有一个合法的交换项。首先在交换高速缓存中搜索页面,如果不存在,则从后援存储器读取页面。

619~690 这些代码用于处理合法交换项存在的地方。

612 调用 `lookup_swap_cache()`(见 K. 2. 4. 1)在交换高速缓存中搜索 `swappage`。

622~641 如果页面不在交换高速缓存中,则用 `read_swap_cache_async()`从后援存储器读取页面。在 638 行, `wait_on_page()`一直被调用直至 I/O 完成。待 I/O 完成后,释放页面引用,并跳转到 `repeat` 标签处获取自旋锁并重新尝试。

644~649 尝试并锁住页面。如果失败,则一直等待直至页面可以上锁,并跳转到 `repeat` 再次尝试。

650~656 如果页面不是最新的, I/O 也由于某种原因失败, 则返回错误。

```

658 delete_from_swap_cache(swappage);
659 if (filepage) {
660     entry->val = 0;
661     info->swapped--;
662     spin_unlock(&info->lock);
663     flush_page_to_ram(swappage);
664     copy_highpage(filepage, swappage);
665     UnlockPage(swappage);
666     page_cache_release(swappage);
667     flush_dcache_page(filepage);
668     SetPageUptodate(filepage);
669     SetPageDirty(filepage);
670     swap_free(swap);
671 } else if (add_to_page_cache_unique(swappage,
672     mapping, idx, page_hash(mapping, idx)) == 0) {
673     entry->val = 0;
674     info->swapped--;
675     spin_unlock(&info->lock);
676     filepage = swappage;
677     SetPageUptodate(filepage);
678     SetPageDirty(filepage);
679     swap_free(swap);
680 } else {
681     if (add_to_swap_cache(swappage, swap) != 0)
682         BUG();
683     spin_unlock(&info->lock);
684     SetPageUptodate(swappage);
685     SetPageDirty(swappage);
686     UnlockPage(swappage);
687     page_cache_release(swappage);
688     goto repeat;
689 }

```

这里所讨论的是页面在交换高速缓存中的情况。

658 从交换高速缓存删除页面, 这样可以尝试将它添加到页面高速缓存中。

659~670 如果调用者用 pagep 参数提供页面, 则用数据更新 swappage 的 pagep。

671~680 如果没有, 则尝试并添加 swappage 到页面高速缓存中。请注意 info->swapped 已经更新, 标记页面为最新, 然后调用 swap_free() 释放交换缓存。

681~689 如果添加页面到页面高速缓存失败,则用 add_to_swap_cache()添加它到交换缓存。标记页面为最新,接着为页面解锁并跳转到 repeat 再次尝试。

```

690 } else if (sgp == SGP_READ && ! filepage) {
691 filepage = find_get_page(mapping, idx);
692 if (filepage &&
693 (! Page_Uptodate(filepage) || TryLockPage(filepage))) {
694 spin_unlock(&info->lock);
695 wait_on_page(filepage);
696 page_cache_release(filepage);
697 filepage = NULL;
698 goto repeat;
699 }
700 spin_unlock(&info->lock);

```

在这块,没有合法的交换项对应 idx。如果页面已经读取且 pagep 为 NULL,则在页面高速缓存中定位页面。

691 调用 find_get_page()(见 J.1.4.1)在页面高速缓存中查找页面。

692~699 如果找到页面,但不是最新的或者无法上锁,则释放自旋锁并一直等待直至页面解锁。然后跳转到 repeat 获取自旋锁并再次尝试。

700 释放自旋锁。

```

701 } else {
702 sbinfo = SHMEM_SB(inode->i_sb);
703 spin_lock(&sbinfo->stat_lock);
704 if (sbinfo->free_blocks == 0) {
705 spin_unlock(&sbinfo->stat_lock);
706 spin_unlock(&info->lock);
707 error = -ENOSPC;
708 goto failed;
709 }
710 sbinfo->free_blocks--;
711 inode->i_blocks += BLOCKS_PER_PAGE;
712 spin_unlock(&sbinfo->stat_lock);
713
714 if (! filepage) {
715 spin_unlock(&info->lock);
716 filepage = page_cache_alloc(mapping);
717 if (! filepage) {
718 shmem_free_block(inode);

```



```

719 error = -ENOMEM;
720 goto failed;
721 }
722
723 spin_lock(&info->lock);
724 entry = shmem_swp_alloc(info, idx, sgp);
725 if (IS_ERR(entry))
726 error = PTR_ERR(entry);
727 if (error || entry->val ||
728 add_to_page_cache_unique(filepage,
729 mapping, idx, page_hash(mapping, idx)) != 0) {
730 spin_unlock(&info->lock);
731 page_cache_release(filepage);
732 shmem_free_block(inode);
733 filepage = NULL;
734 if (error)
735 goto failed;
736 goto repeat;
737 }
738 }
739
740 spin_unlock(&info->lock);
741 clear_highpage(filepage);
742 flush_dcache_page(filepage);
743 SetPageUptodate(filepage);
744 }

```

如果没有，则写入一个不在页面高速缓存的页面。当然，需要定位它。

702 用 SHMEM_SB() 获取超级块信息。

703 获取超级块信息自旋锁。

704~709 如果文件系统中没有空闲块，则释放自旋锁，然后设置返回错误为-ENOSPC 并跳转到 failed。

710 减小可用块的数量。

711 增加 inode 的块使用计数。

712 释放超级块私有信息自旋锁。

714~715 如果 pagep 没有提供页面，则分配新页面并为新页面分配交换项。

715 释放信息自旋锁，因为 page_cache_alloc() 可能已经睡眠。

716 分配新页面。

717~721 如果分配失败,则调用 `shmem_free_block()` 释放块并设置返回错误为 `-ENOMEM`,然后跳转 `failed`。

723 重新获取信息自旋锁。

724 `shmem_swp_entry()` 为页面定位交换项。如果不存在交换项(或许是对应该页的),则分配交换项并返回。

725~726 如果没有找到交换项或分配交换项,则进行设置并返回错误。

728~729 如果没有错误发生,则添加页面到页面高速缓存。

730~732 如果没有页面被添加到页面高速缓存中(例如,由于进程在竞争中,当释放自旋锁的同时有另一个进程插入页面),则释放新页面的引用并释放块。

734~735 如果错误发生,则跳转到 `failed` 报告错误。

736 否则,跳转到 `repeat` 在页面高速缓存中再次搜索指定的页面。

740 释放信息自旋锁。

741 对新页面填充 0。

742 刷新 `dcache` 以避免 CPU 的 `dcache` 别名混淆隐患。

743 标记页面为最新。

745 `done`:

```
746 if (! *pagep) {  
747 if (filepage) {  
748 UnlockPage(filepage);  
749 *pagep = filepage;  
750 } else
```

```
751 *pagep = ZERO_PAGE(0);  
752 }  
753 return 0;  
754
```

755 `failed`:

```
756 if (*pagep != filepage) {  
757 UnlockPage(filepage);  
758 page_cache_release(filepage);  
759 }  
760 return error;  
761 }
```

746~752 如果 `pagep` 没有传入页面,则决定返回什么。如果分配的页面用于写操作,则解锁并返回 `filepage`。否则,调用者为读操作,则返回全局 0 填充页面。

753 返回成功。

755 这是操作失败的执行路径。

756 如果页面由该函数分配并存储在 filepage 中, 则为页面解锁并释放对它的引用。

760 返回错误代码。

L. 5.2 定位交换页面

L. 5.2.1 函数: shmem_alloc_entry() (mm/shmem.c)

这个函数是位于最高层的函数, 它返回与文件中特定页面索引相对应的交换项。如果交换项不存在, 则分配一个。

```

183 static inline swp_entry_t * shmem_alloc_entry(
184         struct shmem_inode_info * info,
185         unsigned long index)
186 {
187     unsigned long page = 0;
188     swp_entry_t * res;
189
190     if (index >= SHMEM_MAX_INDEX)
191         return ERR_PTR(-EFBIG);
192
193     if (info->next_index <= index)
194         info->next_index = index + 1;
195
196     while ((res = shmem_swp_entry(info, index, page)) ==
197             ERR_PTR(-ENOMEM)) {
198         page = get_zeroed_page(GFP_USER);
199         if (!page)
200             break;
201     }
202
203     return res;
204 }
```

188~189 SHMEM_MAX_INDEX 在编译时进行计算, 它表明了页面中可能存在的最大虚拟文件。如果 var 比可能存在的最大文件还要大, 则返回-EFBIG。

191~192 next_index 记录文件尾部页面的索引。只是 inode->i_size 是不够的, 因为文件截取中需要 next_index 字段。

194~198 调用 shmem_swp_entry() 为请求的 index 定位 swp_entry_t。在搜索过程中, shmem_swp_entry() 可能需要一定数量的页面。如果它需要, 则返回-ENOMEM, 这表明在再次尝试以前应当调用 shmem_swp_entry()。

199 返回 swp_entry_t。

L. 5.2.2 函数: shmem_swp_entry() (mm/shmem.c)

这个函数用 inode 中的信息为给定的 index 定位 swp_entry_t。inode 本身可以存储 SHMEM_NR_DIRECT 个交换向量。在这之后, 使用间接块。

```

127 static swp_entry_t * shmem_swp_entry (struct shmem_inode_info * info,
                                         unsigned long index,
                                         unsigned long page)

128 {
129     unsigned long offset;
130     void ** dir;
131
132     if (index < SHMEM_NR_DIRECT)
133         return info->i_direct + index;
134     if (!info->i_indirect) {
135         if (page) {
136             info->i_indirect = (void **) * page;
137             * page = 0;
138         }
139         return NULL;
140     }
141
142     index -= SHMEM_NR_DIRECT;
143     offset = index % ENTRIES_PER_PAGE;
144     index /= ENTRIES_PER_PAGE;
145     dir = info->i_indirect;
146
147     if (index >= ENTRIES_PER_PAGE/2) {
148         index -= ENTRIES_PER_PAGE/2;
149         dir += ENTRIES_PER_PAGE/2 + index/ENTRIES_PER_PAGE;
150         index %= ENTRIES_PER_PAGE;
151         if (!* dir) {
152             if (page) {
153                 * dir = (void *) * page;
154                 * page = 0;
155             }
156             return NULL;
157         }
158         dir = ((void **) * dir);

```

```

159 }
160
161 dir += index;
162 if (! *dir) {
163 if (! page || ! *page)
164 return NULL;
165 *dir = (void *) *page;
166 *page = 0;
167 }
168 return (swp_entry_t *) *dir + offset;
169 }

```

132~133 如果 index 小于 SHMEM_NR_DIRECT，则交换向量是包含在直接块当中的，因此返回它。

134~140 如果页面不在间接块当中，则通过传入的页面参数安装页面并返回 NULL。它告诉调用者分配一个新页面并再次调用函数。

142 使间接块起始于索引 0。

143 ENTRIES_PER_PAGE 是间接块中每个页面包含的交换向量数。offset 目前是间接块页中指定交换向量被找到时的索引。

144 index 目前是间接块链表中必须被找到的目录数。

145 获取系统所需的第 1 个间接块的指针。

147~159 如果请求的目录(index)大于 ENTRIES_PER_PAGE/2，则这是个 3 次间接块，因此必须穿过下一块。

148 指向下一个目录块集合的指针都存储在当前块的后半部分，因此在当前块的后半部分中将 index 作为偏移量进行计算。

149 将 dir 作为指向下一个目录块的指针进行计算。

150 index 目前是 dir 中的指针，指向包含我们感兴趣的交换向量的页面。

151~156 如果尚未分配 dir，则通过页面参数安装页面并返回 NULL，这样调用者可以分配一个新页面并再次调用函数。

158 dir 目前是包括我们感兴趣的交换向量的页面基址了。

161 向前移动 dir 至我们所需的项。

162~167 如果不存在项，则在参数可用的情况下将页面作为参数安装。如果不是这样，则返回 NULL，这样分配一个新的页面，并再次调用函数。

168 返回找到的交换向量。

L. 6 交换空间交互

L. 6. 1 函数: shmem_writepage() (mm/shmem.c)

这个函数用于从页面高速缓存移动页面至交换高速缓存。

```

522 static int shmem_writepage(struct page * page)
523 {
524     struct shmem_inode_info * info;
525     swp_entry_t * entry, swap;
526     struct address_space * mapping;
527     unsigned long index;
528     struct inode * inode;
529
530     BUG_ON(! PageLocked(page));
531     if (! PageLander(page))
532         return fail_writepage(page);
533
534     mapping = page->mapping;
535     index = page->index;
536     inode = mapping->host;
537     info = SHMEM_I(inode);
538     if (info->flags & VM_LOCKED)
539         return fail_writepage(page);

```

这块是函数导引,用于确保操作的可行性。

522 参数表示移动至交换高速缓存的页面。

530 如果页面已经由于 I/O 而上锁,这是个 bug。

531~532 如果没有设置清除位,则调用 fail_writepage()。fail_writepage()由内存中的文件系统用于标记脏页面并重新激活它,这样页面回收器就不会重复尝试写同一个页面。

534~537 记录接下来会作为函数参数的变量。

538~539 如果索引节点文件系统信息被锁住,则返回错误。

```

540     getswap;
541     swap = get_swap_page();
542     if (! swap.val)
543         return fail_writepage(page);

```

```

544
545 spin_lock(&info->lock);
546 BUG_ON(index >= info->next_index);
547 entry = shmem_swp_entry(info, index, NULL);
548 BUG_ON(! entry);
549 BUG_ON(entry->val);
550

```

这块用于从后援存储器分配交换槽和分配索引节点中的 swp_entry_t。

541~543 调用 get_swap_page() (见 K.1.1 小节) 定位空闲交换槽。如果失败, 则调用 fail_writepage()。

545 锁住索引节点信息。

547 用 shmem_swp_entry() 从特定文件系统私有索引节点信息中获取空闲 swp_entry_t。

```

551 /* Remove it from the page cache */
552 remove_inode_page(page);
553 page_cache_release(page);
554
555 /* Add it to the swap cache */
556 if (add_to_swap_cache(page, swap) != 0) {
557 /*
558 * Raced with "speculative" read_swap_cache_async.
559 * Add page back to page cache, unref swap, try again.
560 */
561 add_to_page_cache_locked(page, mapping, index);
562 spin_unlock(&info->lock);
563 swap_free(swap);
564 goto getswap;
565 }
566
567 * entry = swap;
568 info->swapped++;
569 spin_unlock(&info->lock);
570 SetPageUptodate(page);
571 set_page_dirty(page);
572 UnlockPage(page);
573 return 0;
574 }

```

这块从页面高速缓存移动页面到交换高速缓存并更新数据。

552 `remove_inode_page()`(见 J. 1. 2. 1)从索引节点移除页面并在页面链表中 hash 查找它。

553 `page_cache_release()`释放由 `writepage()`操作对页面的局部引用。

556 添加页面到交换高速缓存。在返回之后, `page->mapping` 就可以成为 `swapper_space`。

561 操作失败,则添加页面到页面高速缓存。

562 解锁私有信息。

563-564 释放交换槽并再次尝试。

567 至此,页面已经成为交换缓存的一部分。那么更新索引节点信息指向后援存储器的交换槽。

568 增加交换区中属于索引节点的页面计数。

569 释放私有索引信息。

570~571 移动页面至地址空间脏页面链表,这样可以写回后援存储器。

573 返回成功。

L. 6. 2 函数: `shmem_unuse()` (`mm/shmem.c`)

这个函数用于在 `shmem_inodes` 链表中搜索包含所请求项信息和页面信息的索引节点。这是个开销很大的操作,但仅仅在交换区未激活的情况下使用,因此这不是个大问题。在返回时,将释放交换项,并从交换高速缓存移动页面至页面高速缓存。

```

498 int shmem_unuse(swp_entry_t entry, struct page *page)
499 {
500     struct list_head *p;
501     struct shmem_inode_info * nfo;
502
503     spin_lock(&shmem_ilock);
504     list_for_each(p, &shmem_inodes) {
505         info = list_entry(p, struct shmem_inode_info, list);
506
507         if (info->swapped && shmem_unuse_inode(info, entry, page)) {
508             /* move head to start search for next from here */
509             list_move_tail(&shmem_inodes, &info->list);
510             found = 1;
511             break;
512     }
513 }
```

```

514 spin_unlock(&shmem_ilock);
515 return found;
516 }

```

503 获取 shmem_ilock 自旋锁, 保护索引节点链表。

504 循环遍历 shmem_inodes 链表中每个项, 以此搜索持有所请求项和页面的索引节点。

509 移动索引节点至链表首部。在这种情况下, 接着回收部分页面, 然后在链表首部搜索系统所需的索引节点。

510 表明找到了页面。

511 找到页面和项, 则跳出循环。

514 释放 shmem_ilock 自旋锁。

515 如果找到页面或者没有由 shmem_unuse_inode() 找到, 则返回。

L. 6.3 函数: shmem_unuse_inode() (mm/shmem.c)

这个函数在 info 中搜索索引节点信息, 以此判断项和页面是否属于它。如果它们是, 则清除项, 并从交换缓存移除页面, 将其添加到页面高速缓存中。

```

436 static int shmem_unuse_inode(struct shmem_inode_info * info,
                                swp_entry_t entry,
                                struct page * page)
437 {
438     struct inode * inode;
439     struct address_space * mapping;
440     swp_entry_t * ptr;
441     unsigned long idx;
442     int offset;
443
444     idx = 0;
445     ptr = info->i_direct;
446     spin_lock(&info->lock);
447     offset = info->next_index;
448     if (offset > SHMEM_NR_DIRECT)
449         offset = SHMEM_NR_DIRECT;
450     offset = shmem_find_swp(entry, ptr, ptr + offset);
451     if (offset >= 0)
452         goto found;
453

```

```
454 for (idx = SHMEM_NR_DIRECT; idx < info->next_index;
455 idx += ENTRIES_PER_PAGE) {
456     ptr = shmem_swp_entry(info, idx, NULL);
457     if (!ptr)
458         continue;
459     offset = info->next_index - idx;
460     if (offset > ENTRIES_PER_PAGE)
461         offset = ENTRIES_PER_PAGE;
462     offset = shmem_find_swp(entry, ptr, ptr + offset);
463     if (offset >= 0)
464         goto found;
465 }
466 spin_unlock(&info->lock);
467 return 0;
468 found:
469     idx += offset;
470     inode = info->inode;
471     mapping = inode->i_mapping;
472     delete_from_swap_cache(page);
473
474
475 /* Racing against delete or truncate?
476  * Must leave out of page cache */
477     limit = (inode->i_state & I_FREEING)? 0:
478             (inode->i_size + PAGE_CACHE_SIZE - 1) >> PAGE_CACHE_SHIFT;
479
480     if (idx >= limit || add_to_page_cache_unique(page,
481         mapping, idx, page_hash(mapping, idx)) == 0) {
482         ptr[offset].val = 0;
483     } else if (add_to_swap_cache(page, entry) != 0)
484         BUG();
485     spin_unlock(&info->lock);
486     SetPageUptodate(page);
487
488     /* Decrement swap count even when the entry is left behind;
489      * try_to_unuse will skip over mms, then reincrement count.
490      */
491     swap_free(entry);
492
493 return 1;
494 }
```

493 }

445 初始化 ptr 起始位置在直接块的首部,以便搜索索引节点。

446 锁住索引节点私有信息。

447 初始化 offset 为文件中最后一个页面的索引。

448~449 如果 offset 超过了直接块的尾部,则设置它的值为直接块的尾部。

450 调用 `shmem_fine_swap()`(见 L. 6.4 小节)搜索项对应的直接块。

451~452 如果项在直接块中,则跳转到 found。否则,需要搜索间接块。

454~465 搜索每个间接块以查找项。

456 `shmem_swp_entry()`(见 L. 5.2.2)返回索引节点中在当前 idx 的交换向量。由于 idx 是以 `ENTRIES_PER_PAGE` 为单位进行增加的,则返回下一个搜索的间接块的首部。

457~458 如果发生错误,则表明不存在间接块,那么继续循环直至正常退出。

459 计算文件中还剩下多少页面,这样可以决定是否只需要搜索部分填充的间接块。

460~461 如果 offset 大于一个间接块的大小,则设置 offset 为 `ENTRIES_PER_PAGE`,这样 `shmem_find_swp()` 可以搜索整个间接块。

462 调用 `shmem_find_swp()` 搜索当前整个间接块以查找项。

463~467 如果找到项,则跳转到 found。否则搜索下一个间接块。如果一直没有找到项,则解锁 info 结构,并返回 0,以表明该索引节点不包括项和页面。

468 找到了项,则调用 `swap_free()` 执行必要的释放工作。

470 移动 idx 到块中的交换向量处。

471~472 获取索引节点和映射。

473 从交换高速缓存删除页面。

476~477 检查索引节点是否删除掉或检查 `inode->i_state` 判断是否被截取了。如果是,则在调整了大小的文件中设置最后一个页面的索引的范围。

479~482 如果页面没有被截取和被删除,则用 `add_to_page_cache_unique()` 添加页面到页面高速缓存中。如果成功,则清除交换高速缓存并减小 `info->swapped`。

483~484 如果不是这样,则添加页面至交换高速缓存,这样它在随后会被回收。

485 释放 info 自旋锁。

486 标记页面为最新。

491 减小交换区计数。

492 返回成功。

L. 6.4 函数: `shmem_find_swp()` (`mm/shmem.c`)

这个函数在 ptr 和 eptr 这两个指针之间搜索间接块以查找被请求的项。请注意这两个指

针必须在同一个间接块中。

```

425 static inline int shmem_find_swp(swp_entry_t entry,
426                                     swp_entry_t *dir,
427                                     swp_entry_t *edir)
428 {
429     swp_entry_t *ptr;
430
431     for (ptr = dir; ptr < edir; ptr++) {
432         if (ptr->val == entry.val)
433             return ptr - dir;
434     }

```

429 在 dir 和 edir 两个指针之间循环。

430 如果当前 ptr 项和被请求的项相匹配, 则从 dir 返回偏移量。由于 shmem_unuse_inode() 是这个函数的惟一使用者, 因此将返回间接块中的偏移量。

433 返回的值表示没有找到项。

L. 7 建立共享区

L. 7.1 函数: shmem_zero_setup() (mm/shmem.c)

这个函数用于建立一个 VMA, 该 VMA 是由匿名页面后援的共享区域。该函数的调用图如图 12.5 所示。在 mmap() 用 MAP_SHARED 标志位创建匿名页面时调用该函数。

```

1664 int shmem_zero_setup(struct vm_area_struct *vma)
1665 {
1666     struct file *file;
1667     loff_t size = vma->vm_end - vma->vm_start;
1668
1669     file = shmem_file_setup("dev/zero", size);
1670     if (IS_ERR(file))
1671         return PTR_ERR(file);
1672
1673     if (vma->vm_file)
1674         fput(vma->vm_file);

```

```

1675 vma->vm_file = file;
1676 vma->vm_ops = &shmem_vm_ops;
1677 return 0;
1678 }

```

1667 计算大小。

1669 调用 shmem_file_setup() (见 L. 7.2 小节) 创建称为 dev/zero 且指定大小的文件。可以从代码注释看出名字为什么没必要惟一。

1673~1674 如果已经存在文件对应该虚拟区域, 则调用 fput() 释放其引用。

1675 记录新文件指针。

1676 设置 vm_ops, 这样在 VMA 中发生缺页时将调用 shmem_nopage() (见 L. 5.1.1)。

L. 7.2 函数: shmem_file_setup() (mm/shmem.c)

这个函数用于在内部文件系统 shmemfs 中创建新文件。由于该文件系统是内部的, 所以名字没有必要在每个目录中惟一。因此, 由 shmem_zero_setup() 在匿名区域创建的每个文件都称为“dev/zero”, 而由 shmget() 创建的区域称为“SYSVNN”, 其中 NN 是个键, 它作为第 1 个参数传递给 shmget()。

```

1607 struct file * shmem_file_setup(char * name, loff_t size)
1608 {
1609     int error;
1610     struct file * file;
1611     struct inode * inode;
1612     struct dentry * dentry, * root;
1613     struct qstr this;
1614     int vm_enough_memory(long pages);
1615
1616     if (IS_ERR(shm_mnt))
1617         return (void *)shm_mnt;
1618
1619     if (size > SHMEM_MAX_BYTES)
1620         return ERR_PTR(-EINVAL);
1621
1622     if (! vm_enough_memory(VM_ACCT(size)))
1623         return ERR_PTR(-ENOMEM);
1624
1625     this.name = name;

```

```
1626 this.len = strlen(name);
1627 this.hash = 0; /* will go */
```

1607 参数是要创建的文件名和指定的大小。

1614 vm_enough_memory()(见 M.1.1 小节)检查以确保有足够的内存供映射。

1616~1617 如果挂载点有错误,则返回错误。

1619~1620 不要创建大于 SHMEM_MAX_BYTES 的文件,SHMEM_MAX_BYTES 在 mm/shmem.c 的文件开始处计算过。

1622~1623 确保有足够的内存供映射。

1625~1627 产生 struct qstr,它是用于目录节点的字符串类型。

```
1628 root = shm_mnt->mnt_root;
1629 dentry = d_alloc(root, &this);
1630 if (!dentry)
1631     return ERR_PTR(-ENOMEM);
1632
1633 error = -ENFILE;
1634 file = get_empty_filp();
1635 if (!file)
1636     goto put_dentry;
1637
1638 error = -ENOSPC;
1639 inode = shmem_get_inode(root->d_sb, S_IFREG | S_IRWXUGO, 0);
1640 if (!inode)
1641     goto close_file;
1642
1643 d_instantiate(dentry, inode);
1644 inode->i_size = size;
1645 inode->i_nlink = 0; /* It is unlinked */
1646 file->f_vfsmnt = mntget(shm_mnt);
1647 file->f_dentry = dentry;
1648 file->f_op = &shmem_file_operations;
1649 file->f_mode = FMODE_WRITE | FMODE_READ;
1650 return file;
1651
1652 close_file;
1653 put_filp(file);
1654 put_dentry;
1655 dput(dentry);
```

```
1656 return ERR_PTR(error);
1657 }
```

1628 root 用于表示 shmfs 根目录的目录节点。

1629 调用 d_alloc() 分配新目录项。

1630~1631 如果不能分配则返回-ENOMEM。

1634 从文件表中获取空 struct file。如果一个也找不到，则返回-ENFILE，表示文件表溢出。

1639~1641 创建新索引节点，它是一个常规文件(S_IFREG)且全局可读、可写、可执行。如果失败，则返回-ENOSPC，表示文件系统中没有空间了。

1643 d_instantiate() 为目录项填充索引节点信息。在 fs/dcache.c 中有它的定义。

1644~1649 填充剩下的索引节点和文件信息。

1650 返回新创建的 struct file。

1653 索引节点不能创建情况下的错误处理路径。put_filp() 填充文件表中的 struct file 项。

1655 dput() 将释放目录项的引用，且销毁它。

1656 返回错误代码。

L. 8 System V IPC

L. 8. 1 创建一个 SYSV 共享区

L. 8. 1. 1 函数: sys_shmget() (ipc/shm.c)

```
229 asmlinkage long sys_shmget (key_t key, size_t size, int shmflg)
230 {
231 struct shmid_kernel * shp;
232 int err, id = 0;
233
234 down(&shm_ids.sem);
235 if (key == IPC_PRIVATE) {
236 err = newseg(key, shmflg, size);
237 } else if ((id = ipc_findkey(&shm_ids, key)) == -1) {
238 if (!(shmflg & IPC_CREAT))
239 err = -ENOENT;
```



```

240 else
241 err = newseg(key, shmflg, size);
242 } else if ((shmflg & IPC_CREAT) && (shmflg & IPC_EXCL)) {
243 err = -EEXIST;
244 } else {
245 shp = shm_lock(id);
246 if(shp == NULL)
247 BUG();
248 if (shp->shm_segsz < size)
249 err = -EINVAL;
250 else if (ipcperms(&shp->shm_perm, shmflg))
251 err = -EACCES;
252 else
253 err = shm_buildid(id, shp->shm_perm.seq);
254 shm_unlock(id);
255 }
256 up(&shm_ids.sem);
257 return err;
258 }

```

234 获取保护共享内存 ID 的 semaphore 信号量。

235~236 如果指定了 IPC_PRIVATE，则忽略大部分的标志位，并调用 newseg() 创建区域。该标志位用于提供惟一访问共享区的方式，然后 Linux 并不保证惟一访问方式。

237 如果不是，则调用 ipc_findkey() 搜索查看 key 是否已存在。

238~239 如果不是且没有指定 IPC_CREAT，则返回-ENOENT。

241 如果不是，则调用 newseg() 创建一个新的区域。

242~243 如果区域已存在且进程请求先前未创建的新区域，则返回-EEXIST。

244~255 如果不是，则表示访问一个已有的区域，则锁住它，并确保具有必要的权限，然后调用 shm_buildid() 建立段标识符，接着再次解锁该区域。段标识符将返回到用户空间。

256 释放保护 ID 的 semaphore 信号量。

257 返回错误或者时段标识符。

L. 8.1.2 函数: newseg() (ipc/shm.c)

这个函数创建新共享区。

```

178 static int newseg (key_t key, int shmflg, size_t size)
179 {
180 int error;
181 struct shmid_kernel * shp;

```

```

182 int numpages = (size + PAGE_SIZE -1) >> PAGE_SHIFT;
183 struct file * file;
184 char name[13];
185 int id;
186
187 if (size < SHMMIN || size > shm_ctlmax)
188 return -EINVAL;
189
190 if (shm_tot + numpages >= shm_ctlall)
191 return -ENOSPC;
192
193 shp = (struct shmid_kernel *) kmalloc (sizeof (* shp), GFP_USER);
194 if (! shp)
195 return -ENOMEM;
196 sprintf (name, "SYSV%08x", key);

```

这块分配段描述符。

182 计算区域占用的页面数。

187~188 确保区域大小不超过范围界限。

190~191 确保段所需要的页面数不超过范围界限。

193 调用 kmalloc()(见 H. 4. 2. 1)分配描述符。

196 打印 shmfs 中创建的文件名。名字为 SYSVNN, 其中 NN 是区域的 key 描述符。

```

197 file = shmem_file_setup(name, size);
198 error = PTR_ERR(file);
199 if (IS_ERR(file))
200 goto no_file;
201
202 error = -ENOSPC;
203 id = shm_addid(shp);
204 if(id == -1)
205 goto no_id;
206 shp->shm_perm.key = key;
207 shp->shm_flags = (shmflg & S_IRWXUGO);
208 shp->shm_cprid = current->pid;
209 shp->shm_lprid = 0;
210 shp->shm_atim = shp->shm_dtim = 0;
211 shp->shm_ctim = CURRENT_TIME;
212 shp->shm_segsz = size;
213 shp->shm_nattch = 0;

```

```

214 shp->id = shm_buildid(id, shp->shm_perm.seq);
215 shp->shm_file = file;
216 file->f_dentry->d_inode->i_ino = shp->id;
217 file->f_op = &shm_file_operations;
218 shm_tot += numpages;
219 shm_unlock(id);
220 return shp->id;
221
222 no_id:
223 fput(file);
224 no_file:
225 kfree(shp);
226 return error;
227 }

```

197 用 `shmem_file_setup()`(见 L. 7.2 小节)在 `shmfs` 中创建新文件。

198~200 确保文件创建过程中没有错误发生。

202 缺省情况下,返回的错误表示没有可用的共享内存或请求的大小过于偏大。

206~213 填充段描述符的各个字段。

214 建立段描述符,用于返回给调用者 `shmget()`。

215~217 设置文件指针和文件操作数据结构。

218 更新 `shm_tot` 为共享段所使用的页面总数。

220 返回描述符。

L. 8.2 附属一个 SYSV 共享区

L. 8.2.1 函数: `sys_shmat()` (`ipc/shm.c`)

```

568 asmlinkage long sys_shmat(int shmid, char *shmaddr,
                               int shmflg, ulong *raddr)
569 {
570 struct shmid_kernel *shp;
571 unsigned long addr;
572 unsigned long size;
573 struct file * file;
574 int err;
575 unsigned long flags;
576 unsigned long prot;

```

```

577 unsigned long o_flags;
578 int acc_mode;
579 void * user_addr;
580
581 if (shmid < 0)
582 return -EINVAL;
583
584 if ((addr = (ulong)shmaddr)) {
585 if (addr & (SHMLBA-1)) {
586 if (shmflg & SHM_RND)
587 addr &= ~(SHMLBA-1); /* round down */
588 else
589 return -EINVAL;
590 }
591 flags = MAP_SHARED | MAP_FIXED;
592 } else {
593 if ((shmflg & SHM_REMAP))
594 return -EINVAL;
595
596 flags = MAP_SHARED;
597 }
598
599 if (shmflg & SHM_RDONLY) {
600 prot = PROT_READ;
601 o_flags = O_RDONLY;
602 acc_mode = S_IRUGO;
603 } else {
604 prot = PROT_READ | PROT_WRITE;
605 o_flags = O_RDWR;
606 acc_mode = S_IRUGO | S_IWUGO;
607 }

```

这部分确保 shmat() 的参数均合法。

581~582 不允许出现负标识符,如果出现则返回-EINVAL。

584~591 如果调用者提供地址,则确保它是 ok 的。

585 SHMLBA 是段边界地址的倍数。在 Linux 中,它一般是 PAGE_SIZE。如果地址不是和地址对齐的,则检查调用者是否指定了 SHM_RND,因为它允许改变地址。如果指定了,则四舍五入地址到最近的页面边界。否则,返回-EINVAL。

591 设置 VMA 所使用的标志位,这样可以创建定长(MAP_FIXED)的共享区域(MAP_

SHARED)。

593~596 如果没有提供地址,则确保指定了 SHM_REMAP 且只使用 VMA 的 MAP_SHARED 标志位。这意味着 do_mmap()(见 D.2.1.1)将找到附属于共享区的合适地址。

```

613 shp = shm_lock(shmid);
614 if(shp == NULL)
615     return -EINVAL;
616 err = shm_checkid(shp, shmid);
617 if (err) {
618     shm_unlock(shmid);
619     return err;
620 }
621 if (ipcperms(&shp->shm_perm, acc_mode)) {
622     shm_unlock(shmid);
623     return -EACCES;
624 }
625 file = shp->shm_file;
626 size = file->f_dentry->d_inode->i_size;
627 shp->shm_nattach++;
628 shm_unlock(shmid);

```

这块确保 IPC 权限合法。

613 shm_lock()锁住和 shmid 对应的描述符并返回指向描述符的指针。

614~615 确保描述符存在。

616~620 确保 ID 和描述符相匹配。

621~624 确保调用者具有正确的权限。

625 获取 do_mmap()所需要的指向 struct file 的指针。

626 获取共享区大小,这样 do_mmap()就知道创建多大的 VMA。

627 临时增加 shm_nattach(),它一般用于表示使用该段的 VMA 数。因为这样可以阻止段被提前释放。真正的计数器由 shm_open()进行增加,shm_open 是共享区域中 vm_operations_struct 所使用的 open()的回调函数。

628 释放描述符。

```

630 down_write(&current->mm->mmap_sem);
631 if (addr && !(shmflg & SHM_REMAP)) {
632     user_addr = ERR_PTR(-EINVAL);
633     if (find_vma_intersection(current->mm, addr, addr + size))
634         goto invalid;
635     /*

```

```

636 * If shm segment goes below stack, make sure there is some
637 * space left for the stack to grow (at least 4 pages).
638 */
639 if (addr < current->mm->start_stack &&
640 addr > current->mm->start_stack - size - PAGE_SIZE * 5)
641 goto invalid;
642 }
643
644 user_addr = (void *) do_mmap (file, addr, size, prot, flags, 0);

```

这块作用是调用 do_mmap() 将区域附属到调用进程。

630 获取保护 mm_struct 的 semaphore 信号量。

632~634 如果指定了地址, 则调用 find_vma_intersection() (见 D. 3.1.3) 确保没有 VMA 与我们尝试使用的区域相重叠。

639~641 确保在共享区尾部和栈之间至少有 4 页大小的空隙。

644 调用 do_mmap() (见 D. 2.1.1) 分配 VMA 并映射它到进程地址空间。

```

646 invalid:
647 up_write(&current->mm->mmap_sem);
648
649 down (&shm_ids.sem);
650 if(! (shp = shm_lock(shmid)))
651 BUG();
652 shp->shm_nattch--;
653 if(shp->shm_nattch == 0 &&
654 shp->shm_flags & SHM_DEST)
655 shm_destroy (shp);
656 else
657 shm_unlock(shmid);
658 up (&shm_ids.sem);
659
660 *raddr = (unsigned long) user_addr;
661 err = 0;
662 if (IS_ERR(user_addr))
663 err = PTR_ERR(user_addr);
664 return err;
665
666 }

```

647 释放 mm_struct 的 semaphore 信号量。

649 释放区域 ID 的 semaphore 信号量。

650~651 锁住段描述符。

652 减小临时 shm_nattch 计数器。它将会由 vm_ops→open 的回调函数适当地增加。

653~655 如果用户减少到 0 且指定了 SHM_DEST 标志位, 则销毁区域, 因为已经不需要它了。

657 否则, 给段解锁。

660 设置返回给调用者的地址。

661~663 如果发生错误, 则设置返回给调用者的错误。

664 返回。

附录 M

内存溢出管理

目 录

| | |
|---------------------------------|-----|
| M. 1 确定可用内存 | 698 |
| M. 1. 1 函数:vm_enough_memory() | 698 |
| M. 2 检查 OOM 并从中恢复 | 700 |
| M. 2. 1 函数:out_of_memory() | 700 |
| M. 2. 2 函数:oom_kill() | 701 |
| M. 2. 3 函数:select_bad_process() | 702 |
| M. 2. 4 函数:badness() | 703 |
| M. 2. 5 函数:oom_kill_task() | 704 |

M. 1 确定可用内存

M. 1. 1 函数: vm_enough_memory() (mm/mmap.c)

```
53 int vm_enough_memory(long pages)
54 {
55     unsigned long free;
56
57     /* Sometimes we want to use more memory than we have. */
58     if (sysctl_overcommit_memory)
```

```

69 return 1;
70
71 /* The page cache contains buffer pages these days.. */
72 free = atomic_read(&page_cache_size);
73 free += nr_free_pages();
74 free += nr_swap_pages;
75
76 /*
77  * This double-counts: the nrpages are both in the page-cache
78  * and in the swapper space. At the same time, this compensates
79  * for the swap-space over-allocation (ie "nr_swap_pages" being
80  * too small.
81  */
82 free += swapper_space.nrpages;
83
84 /*
85  * The code below doesn't account for free space in the inode
86  * and dentry slab cache, slab cache fragmentation, inodes and
87  * dentries which will become freeable under VM load, etc.
88  * Lets just hope all these (complex) factors balance out...
89  */
90 free += (dentry_stat.nr_unused * sizeof(struct dentry)) >> PAGE_SHIFT;
91 free += (inodes_stat.nr_unused * sizeof(struct inode)) >> PAGE_SHIFT;
92
93 return free > pages;
94 }

```

68~69 如果系统管理员已经指定允许使用过量的 proc 接口, 它将立即返回并告知系统内存可用。

72 由于页面会被回收, 那么以页面高速缓存的大小对空闲页面开始计数。

73 增加系统空闲页面的总数量。

74 增加系统可用交换槽的总数量。

82 增加交换空间中页面的数量。这将对交换区中的空闲槽双倍计数, 但是由于某些槽要为页面保留起来, 而且这些槽目前也不曾用到, 因此该处理方式是比较妥当的。

90 增加目录项高速缓存中未被使用页面的数量。

91 增加索引节点高速缓存中未使用页面的数量。

93 如果空闲页面的数量比请求的数量要多, 则返回。

M. 2 检查 OOM 并从中恢复

M. 2. 1 函数: out_of_memory() (mm/oom_kill.c)

```
202 void out_of_memory(void)
203 {
204     static unsigned long first, last, count, lastkill;
205     unsigned long now, since;
206
210     if (nr_swap_pages > 0)
211         return;
212
213     now = jiffies;
214     since = now - last;
215     last = now;
216
221     last = now;
222     if (since > 5 * HZ)
223         goto reset;
224
229     since = now - first;
230     if (since < HZ)
231         return;
232
237     if ( ++count < 10)
238         return;
239
245     since = now - lastkill;
246     if (since < HZ * 5)
247         return;
248
252     lastkill = now;
253     oom_kill();
254
255     reset:
256     first = now;
```

```
257 count = 0;
258 }
```

210~211 如果有可用的交换槽,则系统没有 OOM。

213~215 以 jiffies 记录当前时间,并确定函数从被调用时计时经历了多长时间。

222~223 如果从函数最后一次被调用开始计时已经超过 5 s,将重启计时器并退出函数。

229~231 如果从函数最后一次被调用开始计时已经超过 1 s,则退出函数。这可能是在进行 I/O 操作,会马上完成。

237~238 如果在最后一个短间隔当中函数没有被调用达到 10 次,则系统不进行 OOM。

245~247 如果在最后 5 s 内进程被杀死,由于死亡进程会释放内存,则退出函数。

253 Ok, 系统真的是 OOM, 因此调用 oom_kill() (见 M. 2.2 小节)选择杀死哪个进程。

M. 2.2 函数: oom_kill() (mm/oom_kill.c)

该函数首先调用 select_bad_process() 查找一个合适的进程进行杀死操作。一旦查找到, 则遍历任务链表, 系统调用 oom_kill_task() 处理被选择的进程以及它所有的线程。

```
172 static void oom_kill(void)
173 {
174     struct task_struct * p, * q;
175
176     read_lock(&tasklist_lock);
177     p = select_bad_process();
178
179     /* Found nothing?!?! Either we hang forever, or we panic. */
180     if (p == NULL)
181         panic("Out of memory and no killable processes...\n");
182
183     /* kill all processes that share the ->mm (i.e. all threads) */
184     for_each_task(q) {
185         if (q->mm == p->mm)
186             oom_kill_task(q);
187     }
188     read_unlock(&tasklist_lock);
189
190     /*
191      * Make kswapd go out of the way, so "p" has a good chance of
192      * killing itself before someone else gets the chance to ask
193  }
```

```

193 * for more memory.
194 */
195 yield();
196 return;
197 }

```

176 获得任务链表的只读 semaphore 信号量。

177 调用 select_bad_process()(见 M. 2.3 小节)查找一个合适的进程供杀掉。

180~181 如果没有找到合适的进程,将导致系统瘫痪,否则会造成系统的死锁。在这种情况下,死锁或许更好一些,然后让开发人员解决这个 bug,这总比放任不管要好。

184~187 循环遍历任务链表,并为被选中的进程及其所有子线程调用 oom_kill_task()(见 M. 2.5 小节)。请记得所有的线程共享同一个 mm_struct。

188 释放 semaphore 信号量。

195 调用 yield()来允许发送信号并使进程死掉。注释部分表明了 kswapd 将睡眠,但在直接回收路径上的进程也可能执行这个函数。

M. 2.3 函数: select_bad_process() (mm/oom_kill.c)

这个函数用于循环遍历整个任务链表,并返回经 badness()函数记录的最大值的进程。

```

121 static struct task_struct * select_bad_process(void)
122 {
123 int maxpoints = 0;
124 struct task_struct * p = NULL;
125 struct task_struct * chosen = NULL;
126
127 for_each_task(p) {
128 if (p->pid) {
129 int points = badness(p);
130 if (points > maxpoints) {
131 chosen = p;
132 maxpoints = points;
133 }
134 }
135 }
136 return chosen;
137 }

```

127 循环遍历任务链表中的所有任务。

- 128 如果是系统的空闲任务进程,则跳过它。
 129 调用 badness()(见 M. 2. 4 小节)记录该进程。
 130~133 如果这是到目前为止最高的分值,则记录它。
 136 返回由 badness()记录最高值的 task_struct。

M. 2. 4 函数 badness() (mm/oom_kill.c)

它用于计算分值来决定杀掉哪个进程。在第 13 章详细解释了这种记分机制。

```

58 static int badness(struct task_struct * p)
59 {
60 int points, cpu_time, run_time;
61
62 if (! p->mm)
63 return 0;
64
65 if (p->flags & PF_MEMDIE)
66 return 0;
67
71 points = p->mm->total_vm;
72
79 cpu_time = (p->times.tms_utime + p->times.tms_stime)
79 >> (SHIFT_HZ + 3);
80 run_time = (jiffies - p->start_time) >> (SHIFT_HZ + 10);
81
82 points /= int_sqrt(cpu_time);
83 points /= int_sqrt(int_sqrt(run_time));
84
89 if (p->nice > 0)
90 points *= 2;
91
96 if (cap_t(p->cap_effective) & CAP_TO_MASK(CAP_SYS_ADMIN) ||
97 p->uid == 0 || p->euid == 0)
98 points /= 4;
99
106 if (cap_t(p->cap_effective) & CAP_TO_MASK(CAP_SYS_RAWIO))
107 points /= 4;
108 #ifdef DEBUG
109 printk(KERN_DEBUG "OOMkill: task %d (%s) got %d points\n",

```

```

110 p->pid, p->comm, points);
111 #endif
112 return points;
113 }

```

62~63 如果没有 mm, 则返回 0, 因为这是个内核线程。

65~66 如果进程已经由 OOM 销毁程序标记为退出状态, 则返回 0, 因为没有道理重复杀掉同一个进程。

71 进程所使用的全部 VM 是基本起始分值。

79~80 以秒为单位计算进程的整个运行时间得到 cpu_time。run_time 是以分钟为单位计算的进程的整个运行时间。注释表明这么做并没有什么理论基础, 只是在实际运行中工作良好。

82 用 cpu_time 的平方整除分值。

83 用 run_time 的立方整除分值。

89~90 如果进程的 nice 值处于低优先级, 则增加分值为原来的两倍, 因为这个进程看起来并不重要。

96~98 另一方面, 如果进程具有超级用户特权, 或者具有 CAP_SYS_ADMIN 权限, 那么它看起来是一个系统进程, 则将分值除以 4。

106~107 如果进程已经直接访问硬件, 则进程除以 4。强行杀掉这些进程会潜在地使硬件处于不一致状态。例如, 强行杀掉 X 进程可不是一个好主意。

112 返回分值。

M. 2.5 函数: oom_kill_task() (mm/oom_kill.c)

这个函数用于发送合适的杀死信号至被选择的任务。

```

144 void oom_kill_task(struct task_struct * p)
145 {
146 printk(KERN_ERR "Out of Memory: Killed process %d (%s).\n",
p->pid, p->comm);
147
148 /*
149 * We give our sacrificial lamb high priority and access to
150 * all the memory it needs. That way it should be able to
151 * exit() and clear out its resources quickly...
152 */
153 p->counter = 5 * HZ;

```

```
154 p->flags |= PF_MEMALLOC | PF_MEMDIE;  
155  
156 /* This process has hardware access, be more careful. */  
157 if (cap_t(p->cap_effective) & CAP_TO_MASK(CAP_SYS_RAWIO)) {  
158 force_sig(SIGTERM, p);  
159 } else {  
160 force_sig(SIGKILL, p);  
161 }  
162 }
```

146 打印被杀掉进程的信息。

153 这给予将死进程在 CPU 上更多的时间,这样它可以快速地杀死自己。

154 如果进程在清除它自己时需要更多的页面,这些标志位告诉分配器以合适的方式处理这个进程。

157~158 如果进程可以直接访问硬件,就发送一个 SIGTERM 信号,以使它干净退出。

160 否则,发送一个 SIGKILL 信号强行杀死进程。

参考文献

- [BA01] Jeff Bonwick and Jonathan Adams. Magazines and vmem: Extending the slab allocator to many CPUs and arbitrary resources. In *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX-01)* , pages 15-34, Berkeley, CA, June 25-30 2001. The USENIX Association.
- [BC00] D. (Daniele) Bovet and Marco Cesati. *Understanding the Linux Kernel* . Cambridge, MA: O'Reilly, 2000.
- [BC03] D. (Daniele) Bovet and Marco Cesati. *Understanding the Linux Kernel 2nd Edition*. Cambridge, MA: O'Reilly, 2003.
- [BL89] R. Barkley and T. Lee. A lazy buddy system bounded by two coalescing delays. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* . ACM Press, 1989.
- [Bon94] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer* , pages 87-98, 1994.
- [Car84] Rickard W. Carr. *Virtual Memory Management* . UMI Research Press, 1984.
- [CD80] E. G. Coffman and P. J. Denning. *Operating Systems Theory* . Englewood Cliffs, NJ: Prentice-Hall, 1980.
- [CP99] Charles D. Cranor and Gurudatta M. Parulkar. The UVM virtual memory system. In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX-99)* , pages 117-130, Berkeley, CA, 1999. USENIX Association.
- [CS98] Kevin Dowd and Charles Severance. *High Performance Computing, 2nd Edition* . Newton, MA: O'Reilly, 1998.
- [Den70] Peter J. Denning. Virtual memory. *ACM Computing Surveys (CSUR)* , 2(3):153-189, 1970.
- [FF02] Joseph Feller and Brian Fitzgerald. *Understanding Open Source Software Development* . Pearson Education Ltd. , 2002.
- [GAV95] A. Gonzalez, C. Aliagas and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In ACM, *Conference Proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain, July 3-7, 1995* , pages 338-347, New York, 1995. ACM Press.
- [GC94] Berny Goodheart and James Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4, an Open Systems Design* . Prentice-Hall, 1994.

- [Hac] Various Kernel Hackers. *Kernel 2.4.18 Source Code* . <ftp://ftp.kernel.org/pub/linux/kernel/v2.4/linux-2.4.18.tar.gz>, February 25, 2002.
- [Hac00] Random Kernel Hacker. How to get your change into the linux kernel. *Kernel Source Documentation Tree (SubmittingPatches)* , 2000.
- [Hac02] Various Kernel Hackers. *Kernel 2.2.22 Source Code* . <ftp://ftp.kernel.org/pub/linux/kernel/v2.2/linux-2.2.22.tar.gz>, 2002.
- [HK97] Amir H. Hashemi and David R. Kaeli. Efficient procedure mapping using cache line coloring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)* , 32(5) of *ACM SIGPLAN Notices* , pages 171-182, New York, June 15-18 1997. ACM Press.
- [JS94] Theodore Johnson and Dennis Shasha. 2q: a low overhead high performance buffer management replacement algorithm. In *Proceedings of the Twentieth International Conference on Very Large Databases* , pages 439-450, Santiago, Chile, 1994.
- [JW98] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: solved? In *Proceedings of the First International Symposium on Memory Management* . ACM Press, 1998.
- [KB85] David G. Korn and Kiem-Phong Bo. In search of a better malloc. In *Proceedings of the Summer 1985 USENIX Conference* , pages 489-506, Portland, OR, 1985.
- [Kes91] Richard E. Kessler. Analysis of multi-megabyte secondary CPU cache memories. *Technical Report CS-TR-1991-1032* , University of Wisconsin, Madison, July 1991.
- [KMC02] Scott Kaplan, Lyle McGeoch and Megan Cole. Adaptive caching for demand prepaging. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management* , ACM SIGPLAN Notices, pages 114-126, Berlin, Germany, June 2002. ACM Press.
- [Kno65] Kenneth C. Knowlton. A fast storage allocator. *Communications of the ACM* , 8(10):623-624, 1965.
- [Knu68] D. Knuth. *The Art of Computer Programming, Fundamental Algorithms, Volume 1* . Reading, MA: Addison-Wesley, 1968.
- [Lev00] Check Lever. *Linux Kernel Hash Table Behavior: Analysis and Improvements* . <http://www.citi.umich.edu/techreports/reports/citi-tr-00-1.pdf>, 2000.
- [McK96] Marshall Kirk McKusick. *The Design and Implementation of the 4.4BSD Operating System* . Addison-Wesley, 1996.
- [Mil00] David S. Miller. *Cache and TLB Flushing Under Linux* . Kernel Source Documentation Tree, 2000.