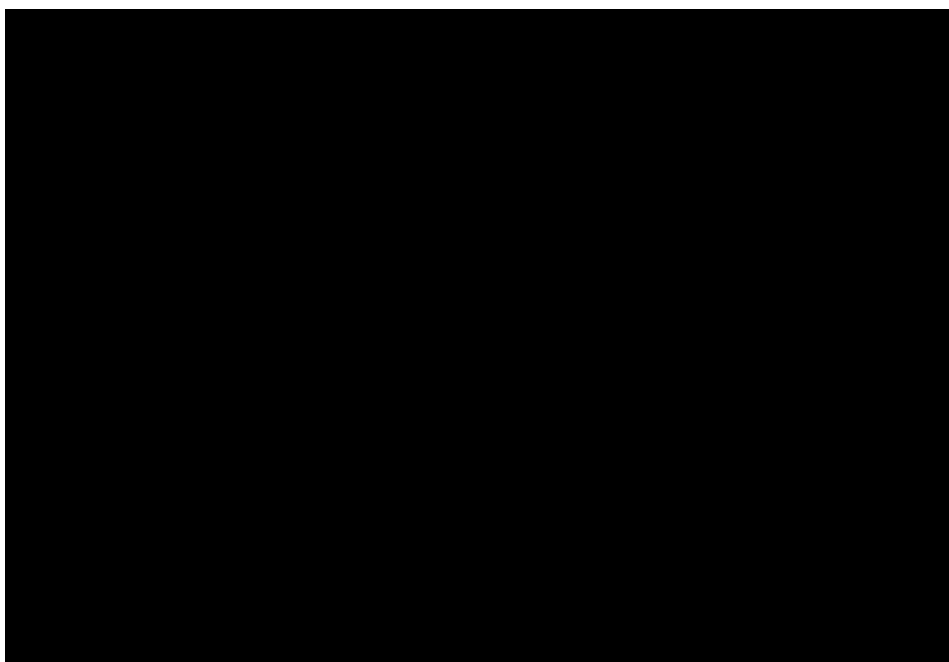


Understanding the Linux® Virtual Memory Manager



Contents

PREFACE	xiii
1 INTRODUCTION	1
1.1 Getting Started	2
1.2 Managing the Source	4
1.3 Browsing the Code	9
1.4 Reading the Code	11
1.5 Submitting Patches	12
2 DESCRIBING PHYSICAL MEMORY	15
2.1 Nodes	16
2.2 Zones	18
2.3 Zone Initialization	23
2.4 Initializing mem_map	24
2.5 Pages	24
2.6 Mapping Pages to Zones	29
2.7 High Memory	29
2.8 What's New in 2.6	30
3 PAGE TABLE MANAGEMENT	33
3.1 Describing the Page Directory	33
3.2 Describing a Page Table Entry	36
3.3 Using Page Table Entries	37
3.4 Translating and Setting Page Table Entries	39
3.5 Allocating and Freeing Page Tables	39
3.6 Kernel Page Tables	40
3.7 Mapping Addresses to a struct page	42
3.8 Translation Lookaside Buffer (TLB)	43
	vii

3.9	Level 1 CPU Cache Management	44
3.10	What's New in 2.6	47
4	PROCESS ADDRESS SPACE	53
4.1	Linear Address Space	53
4.2	Managing the Address Space	55
4.3	Process Address Space Descriptor	57
4.4	Memory Regions	61
4.5	Exception Handling	79
4.6	Page Faulting	80
4.7	Copying to/from Userspace	87
4.8	What's New in 2.6	90
5	BOOT MEMORY ALLOCATOR	95
5.1	Representing the Boot Map	96
5.2	Initializing the Boot Memory Allocator	98
5.3	Initializing bootmem_data	98
5.4	Allocating Memory	99
5.5	Freeing Memory	100
5.6	Retiring the Boot Memory Allocator	101
5.7	What's New in 2.6	102
6	PHYSICAL PAGE ALLOCATION	105
6.1	Managing Free Blocks	105
6.2	Allocating Pages	106
6.3	Free Pages	109
6.4	Get Free Page (GFP) Flags	110
6.5	Process Flags	111
6.6	Avoiding Fragmentation	112
6.7	What's New in 2.6	113
7	NONCONTIGUOUS MEMORY ALLOCATION	117
7.1	Describing Virtual Memory Areas	117
7.2	Allocating a Noncontiguous Area	118
7.3	Freeing a Noncontiguous Area	120
7.4	What's New in 2.6	121
8	SLAB ALLOCATOR	123
8.1	Caches	125
8.2	Slabs	137

8.3	Objects	144
8.4	Sizes Cache	146
8.5	Per-CPU Object Cache	148
8.6	Slab Allocator Initialization	150
8.7	Interfacing With the Buddy Allocator	151
8.8	What's New in 2.6	151
9	HIGH MEMORY MANAGEMENT	153
9.1	Managing the PKMap Address Space	153
9.2	Mapping High Memory Pages	154
9.3	Unmapping Pages	156
9.4	Mapping High Memory Pages Atomically	156
9.5	Bounce Buffers	157
9.6	Emergency Pools	159
9.7	What's New in 2.6	160
10	PAGE FRAME RECLAMATION	163
10.1	Page Replacement Policy	164
10.2	Page Cache	165
10.3	LRU Lists	169
10.4	Shrinking All Caches	173
10.5	Swapping Out Process Pages	173
10.6	Pageout Daemon (kswapd)	175
10.7	What's New in 2.6	177
11	SWAP MANAGEMENT	179
11.1	Describing the Swap Area	180
11.2	Mapping Page Table Entries to Swap Entries	183
11.3	Allocating a Swap Slot	184
11.4	Swap Cache	185
11.5	Reading Pages From Backing Storage	189
11.6	Writing Pages to Backing Storage	189
11.7	Reading/Writing Swap Area Blocks	192
11.8	Activating a Swap Area	192
11.9	Deactivating a Swap Area	193
11.10	What's New in 2.6	194
12	SHARED MEMORY VIRTUAL FILESYSTEM	195
12.1	Initializing the Virtual Filesystem	196
12.2	Using shmem Functions	197

12.3	Creating Files in tmpfs	199
12.4	Page Faulting Within a Virtual File	201
12.5	File Operations in tmpfs	203
12.6	Inode Operations in tmpfs	203
12.7	Setting Up Shared Regions	204
12.8	System V IPC	204
12.9	What's New in 2.6	207
13	OUT OF MEMORY MANAGEMENT	209
13.1	Checking Available Memory	209
13.2	Determining OOM Status	210
13.3	Selecting a Process	211
13.4	Killing the Selected Process	211
13.5	Is That It?	211
13.6	What's New in 2.6	211
14	THE FINAL WORD	213
 CODE COMMENTARY		
A	INTRODUCTION	217
B	DESCRIBING PHYSICAL MEMORY	219
B.1	Initializing Zones	220
B.2	Page Operations	234
C	PAGE TABLE MANAGEMENT	239
C.1	Page Table Initialization	240
C.2	Page Table Walking	248
D	PROCESS ADDRESS SPACE	251
D.1	Process Memory Descriptors	254
D.2	Creating Memory Regions	261
D.3	Searching Memory Regions	309
D.4	Locking and Unlocking Memory Regions	315
D.5	Page Faulting	328
D.6	Page-Related Disk I/O	355
E	BOOT MEMORY ALLOCATOR	395
E.1	Initializing the Boot Memory Allocator	396

E.2	Allocating Memory	399
E.3	Freeing Memory	409
E.4	Retiring the Boot Memory Allocator	411
F	PHYSICAL PAGE ALLOCATION	419
F.1	Allocating Pages	420
F.2	Allocation Helper Functions	433
F.3	Free Pages	435
F.4	Free Helper Functions	440
G	NONCONTIGUOUS MEMORY ALLOCATION	441
G.1	Allocating a Noncontiguous Area	442
G.2	Freeing a Noncontiguous Area	452
H	SLAB ALLOCATOR	457
H.1	Cache Manipulation	459
H.2	Slabs	479
H.3	Objects	486
H.4	Sizes Cache	501
H.5	Per-CPU Object Cache	504
H.6	Slab Allocator Initialization	511
H.7	Interfacing with the Buddy Allocator	512
I	HIGH MEMORY MANAGEMENT	513
I.1	Mapping High Memory Pages	514
I.2	Mapping High Memory Pages Atomically	519
I.3	Unmapping Pages	521
I.4	Unmapping High Memory Pages Atomically	523
I.5	Bounce Buffers	524
I.6	Emergency Pools	532
J	PAGE FRAME RECLAMATION	535
J.1	Page Cache Operations	537
J.2	LRU List Operations	547
J.3	Reclaiming inactive list	552
J.4	Reclaiming Pages From the LRU Lists	554
J.5	Shrinking All Caches	562
J.6	Swapping Out Process Pages	566
J.7	Page Swap Daemon	577

K	SWAP MANAGEMENT	583
K.1	Scanning for Free Entries	585
K.2	Swap Cache	590
K.3	Swap Area I/O	597
K.4	Activating a Swap Area	607
K.5	Deactivating a Swap Area	619
L	SHARED MEMORY VIRTUAL FILESYSTEM	633
L.1	Initializing shmfs	635
L.2	Creating Files in tmpfs	641
L.3	File Operations in tmpfs	645
L.4	Inode Operations in tmpfs	659
L.5	Page Faulting Within a Virtual File	668
L.6	Swap Space Interaction	679
L.7	Setting Up Shared Regions	686
L.8	System V IPC	689
M	OUT OF MEMORY MANAGEMENT	697
M.1	Determining Available Memory	698
M.2	Detecting and Recovering From OOM	700
	REFERENCES	707
	CODE COMMENTARY INDEX	711
	INDEX	717
	ABOUT THE AUTHOR	729

Preface

Linux is developed with a stronger practical emphasis than a theoretical one. When new algorithms or changes to existing implementations are suggested, it is common to request code to match the argument. Many of the algorithms used in the Virtual Memory (VM) system were designed by theorists, but the implementations have now diverged considerably from the theory. In part, Linux does follow the traditional development cycle of design to implementation, but changes made in reaction to how the system behaved in the “real world” and intuitive decisions by developers are more common.

This means that the VM performs well in practice. However, very little VM documentation is available except for a few incomplete overviews on a small number of Web sites, except the Web site containing an earlier draft of this book, of course! This lack of documentation has led to the situation where the VM is fully understood only by a small number of core developers. New developers looking for information on how VM functions are generally told to read the source. Little or no information is available on the theoretical basis for the implementation. This requires that even a casual observer invest a large amount of time reading the code and studying the field of Memory Management.

This book gives a detailed tour of the Linux VM as implemented in 2.4.22 and gives a solid introduction of what to expect in 2.6. As well as discussing the implementation, the theory that Linux VM is based on will also be introduced. This is not intended to be a memory management theory book, but understanding why the VM is implemented in a particular fashion is often much simpler if the underlying basis is known in advance.

To complement the description, the appendices include a detailed code commentary on a significant percentage of the VM. This should drastically reduce the amount of time a developer or researcher needs to invest in understanding what is happening inside the Linux VM because VM implementations tend to follow similar code patterns even between major versions. This means that, with a solid understanding of the 2.4 VM, the later 2.5 development VMs and the 2.6 final release will be decipherable in a number of weeks.

The Intended Audience

Anyone interested in how the VM, a core kernel subsystem, works will find answers to many of their questions in this book. The VM, more than any other subsystem, affects the overall performance of the operating system. The VM is also one of the most poorly understood and badly documented subsystems in Linux, partially because there is, quite literally, so much of it. It is very difficult to isolate and understand individual parts of the code without first having a strong conceptual model of the whole VM, so this book intends to give a detailed description of what to expect before going to the source.

This material should be of prime interest to new developers who want to adapt the VM to their needs and to readers who simply would like to know how the VM works. It also will benefit other subsystem developers who want to get the most from the VM when they interact with it and operating systems researchers looking for details on how memory management is implemented in a modern operating system. For others, who just want to learn more about a subsystem that is the focus of so much discussion, they will find an easy-to-read description of the VM functionality that covers all the details without the need to plow through source code.

However, it is assumed that the reader has read at least one general operating system book or one general Linux kernel-orientated book and has a general knowledge of C before tackling this book. Although every effort is made to make the material approachable, some prior knowledge of general operating systems is assumed.

Book Overview

In Chapter 1, we go into detail on how the source code may be managed and deciphered. Three tools are introduced that are used for analysis, easy browsing and management of code. The main tools are the *Linux Cross Referencing (LXR)* tool, which allows source code to be browsed as a Web page, and *CodeViz*, which was developed while researching this book, for generating call graphs. The last tool, *PatchSet*, is for managing kernels and the application of patches. Applying patches manually can be time consuming, and using version control software, such as Concurrent Versions Systems (CVS) (<http://www.cvshome.org/>) or BitKeeper (<http://www.bitmover.com>), is not always an option. With PatchSet, a simple specification determines what source to use, what patches to apply and what kernel configuration to use.

In the subsequent chapters, each part of the Linux VM implementation is discussed in detail, such as how memory is described in an architecture-independent manner, how processes manage their memory, how the specific allocators work and so on. Each chapter will refer to other sources that describe the behavior of Linux, as well as covering in depth the implementation, the functions used and their call graphs so that the reader will have a clear view of how the code is structured. The end of each chapter has a *What's New* section, which introduces what to expect in the 2.6 VM.

The appendices are a code commentary of a significant percentage of the VM. They give a line-by-line description of some of the more complex aspects of the VM. The style of the VM tends to be reasonably consistent, even between major releases of the kernel, so an in-depth understanding of the 2.4 VM will be an invaluable aid to understanding the 2.6 kernel when it is released.

What's New in 2.6

At the time of writing, 2.6.0-test4 has just been released, so 2.6.0-final is due any month now. Fortunately, the 2.6 VM, in most ways, is still quite recognizable in comparison with 2.4. However, 2.6 has some new material and concepts, and it would be a pity to ignore them. Therefore the book has the What's New in 2.6 sections. To some extent, these sections presume you have read the rest of the book, so only glance at them during the first reading. If you decide to start reading 2.5 and 2.6 VM code, the basic description of what to expect from the What's New sections should greatly aid your understanding. The sections based on the 2.6.0-test4 kernel should not change significantly before 2.6. Because they are still subject to change, though, you should treat the What's New sections as guidelines rather than definite facts.

Companion CD

A companion CD is included with this book, and it is highly recommended the reader become familiar with it, especially as you progress more through the book and are using the code commentary. It is recommended that the CD is used with a GNU/Linux system, but it is not required.

The text of the book is contained on the CD in HTML, PDF and plain text formats so the reader can perform basic text searches if the index does not have the desired information. If you are reading the first edition of the book, you may notice small differences between the CD version and the paper version due to printing deadlines, but the differences are minor.

Almost all the tools used to research the book's material are contained on the CD. Each of the tools may be installed on virtually any GNU/Linux installation, references are included to available documentation and the project home sites, so you can check for further updates.

With many GNU/Linux installations, there is the additional bonus of being able to run a Web server directly from the CD. The server has been tested with Red Hat 7.3 and Debian Woody but should work with any distribution. The small Web site it provides at <http://localhost:10080> offers a number of useful features:

- A searchable index for functions that have a code commentary available. If a function is searched for that does not have a commentary, the browser will be automatically redirected to LXR.
- A Web browsable copy of the Linux 2.4.22 source. This allows code to be browsed and identified to be searched for.

A live version of CodeViz, the tool used to generate call graphs for the book, is available. If you feel that the book's graphs are lacking some detail you want, generate them yourself.

The VMRegress, CodeViz and PatchSet packages, which are discussed in Chapter 1, are available in /cdrom/software. gcc-3.0.4 is also provided because it is required for building CodeViz.

Mount the CD on /cdrom as follows:

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

The Web server is Apache 1.3.27 (<http://www.apache.org/>) and has been built and configured to run with its root as /cdrom/. If your distribution normally uses another directory, you will need to use this one instead. To start it, run the script /cdrom/start_server. If no errors occur, the output should look like:

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

The URL to access is <http://localhost:10080/>

When the server starts successfully, point your browser to <http://localhost:10080> to avail of the CD's Web services. To shut down the server, run the script /cdrom/stop_server, and the CD may then be unmounted.

Typographic Conventions

The conventions used in this document are simple. New concepts that are introduced, as well as URLs, are in *italicized* font. Binaries and package names are in **bold**. Structures, **enum** names, compile time defines and variables are in a constant-width font. At times, when talking about a **enum** in a structure, both the structure and **enum** name will be included as `page → list`, for example. File names are in a constant-width font, but include `%s` have angle brackets around them like `<linux/mm.h>` and may be found in the `include/` directory of the kernel source.

Acknowledgments

The compilation of this book was not a trivial task. This book was researched and developed in the open, and I would be remiss not to mention some of the people who helped me at various intervals. If there is anyone I missed, I apologize now.

First, I would like to thank John O'Gorman, who tragically passed away while the material for this book was being researched. His experience and guidance largely inspired the format and quality of this book.

Second, I would like to thank Mark L. Taub from Prentice Hall PTR for giving me the opportunity to publish this book. It has been a rewarding experience and

made trawling through all the code worthwhile. Massive thanks go to my reviewers, who provided clear and detailed feedback long after I thought I had ~~finished~~ writing. Finally, on the publisher's front, I would like to thank Bruce Perens for allowing me to publish in the Bruce Perens Open Source Series (<http://www.perens.com/Books>).

With the technical research, a number of people provided invaluable insight. Abhishek Nayani was a source of encouragement and enthusiasm early in the research. Ingo Oeser kindly provided invaluable assistance early on with a detailed explanation of how data is copied from userspace to kernel space, and he included some valuable historical context. He also kindly offered to help me if I felt I ever got lost in the twisty maze of kernel code. Scott Kaplan made numerous corrections to a number of systems from noncontiguous memory allocation to page replacement policy. Jonathon Corbet provided the most detailed account of the history of kernel development with the kernel page he writes for *Linux Weekly News*. Zack Brown, the chief behind Kernel Traffic, is the sole reason I did not drown in kernel-related mail. IBM, as part of the Equinox Project, provided an xSeries 350, which was invaluable for running my own test kernels on machines larger than those I previously had access to. Late in the game, Jeffrey Haran found the few remaining technical corrections and more of the ever-present grammar errors. Most importantly, I'm grateful for his enlightenment on some PPC issues. Finally, Patrick Healy was crucial to ensuring that this book was consistent and approachable to people who are familiar with, but not experts on, Linux or memory management.

A number of people helped with smaller technical issues and general inconsistencies where material was not covered in sufficient depth. They are Muli Ben-Yehuda, Parag Sharma, Matthew Dobson, Roger Luethi, Brian Lowe and Scott Crosby. All of them sent corrections and queries on different parts of the document, which ensured that too much prior knowledge was not assumed.

Carl Spalletta sent a number of queries and corrections to every aspect of the book in its earlier online form. Steve Greenland sent a large number of grammar corrections. Philipp Marek went above and beyond being helpful by sending more than 90 separate corrections and queries on various aspects. Long after I thought I was ~~finished~~, Aris Sotiropoulos sent a large number of small corrections and suggestions. The last person, whose name I cannot remember, but is an editor for a magazine, sent me more than 140 corrections to an early version. You know who you are. Thanks.

Eleven people sent a few corrections. Though small, they were still missed by several of my own checks. They are Marek Januszewski, Amit Shah, Adrian Stanciu, Andy Isaacson, Jean Francois Martinez, Glen Kaukola, Wolfgang Oertl, Michael Babcock, Kirk True, Chuck Luciano and David Wilson.

On the development of VMRegress, nine people helped me keep it together. Danny Faught and Paul Larson both sent me a number of bug reports and helped ensure that VMRegress worked with a variety of different kernels. Cli White, from the OSDL labs, ensured that VMRegress would have a wider application than my own test box. Dave Olien, also associated with the OSDL labs, was responsible for updating VMRegress to work with 2.5.64 and later kernels. Albert Cahalan sent all the information I needed to make VMRegress function against later proc utilities. Finally, Andrew Morton, Rik van Riel and Scott Kaplan all provided insight on the

direction the tool should be developed to be both valid and useful.

The last long list are people who sent me encouragement and thanks at various intervals. They are Martin Bligh, Paul Rolland, Mohamed Ghouse, Samuel Chessman, Ersin Er, Mark Hoy, Michael Martin, Martin Gallwey, Ravi Parimi, Daniel Codt, Adnan Shah, Xiong Quanren, Dave Airlie, Der Herr Hofrat, Ida Hallgren, Manu Anand, Eugene Teo, Diego Calleja and Ed Cashin. Thanks. The encouragement was heartening.

In conclusion, I would like to thank a few people without whom I would not have completed this book. Thanks to my parents, who kept me going long after I should have been earning enough money to support myself. Thanks to my girlfriend, Karen, who patiently listened to rants, tech babble and angsting over the book and made sure I was the person with the best toys. Kudos to friends who dragged me away from the computer periodically and kept me relatively sane, including Daren, who is cooking me dinner as I write this. Finally, thanks to the thousands of hackers who have contributed to GNU, the Linux kernel and other Free Software projects over the years, without whom I would not have an excellent system to write about. It was an inspiration to see such dedication when I ~~first~~ started programming on my own PC six years ago, after ~~finally~~ figuring out that Linux was not an application that Windows used for reading email.

Introduction

Linux is a relatively new operating system that has begun to enjoy a lot of attention from the business, academic and free software worlds. As the operating system matures, its feature set, capabilities and performance grow, but so, out of necessity does its size and complexity. Table 1.1 shows the size of the kernel source code in bytes and lines of code of the `mm/` part of the kernel tree. This size does not include the machine-dependent code or any of the buffer management code and does not even pretend to be an accurate metric for complexity, but it still serves as a small indicator.

Version	Release Date	Total Size	Size of <code>mm/</code>	Line Count
1.0	March 13, 1992	5.9MiB	96KiB	3,109
1.2.13	February 8, 1995	11MiB	136KiB	4,531
2.0.39	January 9, 2001	35MiB	204KiB	6,792
2.2.22	September 16, 2002	93MiB	292KiB	9,554
2.4.22	August 25, 2003	181MiB	436KiB	15,724
2.6.0-test4	August 22, 2003	261MiB	604KiB	21,714

Table 1.1. Kernel Size as an Indicator of Complexity

Out of habit, open source developers tell new developers with questions to refer directly to the source with the polite acronym RTFS¹, or refer them to the kernel newbies mailing list (<http://www.kernelnewbies.org>). With the Linux VM manager, this used to be a suitable response because the time required to understand the VM could be measured in weeks. Moreover, the books available devoted enough time to the memory management chapters to make the relatively small amount of code easy to navigate.

The books that describe the operating system such as *Understanding the Linux Kernel* [BC00] [BC03] tend to cover the entire kernel rather than one topic with the notable exception of device drivers [RC01]. These books, particularly *Understanding the Linux Kernel*, provide invaluable insight into kernel internals, but they miss the details that are specific to the VM and not of general interest. But the book you are holding details why `ZONE_NORMAL` is exactly 896MiB and exactly how per-cpu caches

¹Read The Flaming Source. It doesn't really stand for Flaming, but children could be reading.

are implemented. Other aspects of the VM, such as the boot memory allocator and the VM system, which are not of general kernel interest, are also covered in this book.

Increasingly, to get a comprehensive view on how the kernel functions, one is required to read through the source code line by line. This book tackles the VM specifically so that this investment of time to understand the kernel functions will be measured in weeks and not months. The details that are missed by the main part of the book are caught by the code commentary.

In this chapter, there will be an informal introduction to the basics of acquiring information on an open source project and some methods for managing, browsing and comprehending the code. If you do not intend to be reading the actual source, you may skip to Chapter 2.

1.1 Getting Started

One of the largest initial obstacles to understanding code is deciding where to start and how to easily manage, browse and get an overview of the overall code structure. If requested on mailing lists, people will provide some suggestions on how to proceed, but a comprehensive methodology is rarely offered aside from suggestions to keep reading the source until it makes sense. The following sections introduce some useful rules of thumb for open source code comprehension and specific guidelines for how the rules may be applied to the kernel.

1.1.1 Configuration and Building

With any open source project, the first step is to download the source and read the installation documentation. By convention, the source will have a README or INSTALL file at the top level of the source tree [FF02]. In fact, some automated build tools such as `automake` require the `install` file to exist. These files contain instructions for configuring and installing the package or give a reference to where more information may be found. Linux is no exception because it includes a README that describes how the kernel may be configured and built.

The second step is to build the software. In earlier days, the requirement for many projects was to edit the Makefile by hand, but this is rarely the case now. Free software usually uses at least `autoconf`² to automate testing of the build environment and `automake`³ to simplify the creation of Makefiles, so building is often as simple as:

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

Some older projects, such as the Linux kernel, use their own configuration tools, and some large projects such as the Apache Web server have numerous configuration options, but usually the `configure` script is the starting point. In the case of the

²<http://www.gnu.org/software/autoconf/>

³<http://www.gnu.org/software/automake/>

kernel, the configuration is handled by the Makefiles and supporting tools. The simplest means of configuration is to:

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

This asks a long series of questions on what type of kernel should be built. After all the questions have been answered, compiling the kernel is simply:

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

A comprehensive guide on configuring and compiling a kernel is available with the Kernel HOWTO⁴ and will not be covered in detail with this book. For now, we will presume you have one fully built kernel, and it is time to begin figuring out how the new kernel actually works.

1.1.2 Sources of Information

Open source projects will usually have a home page, especially because free project hosting sites such as <http://www.sourceforge.net> are available. The home site will contain links to available documentation and instructions on how to join the mailing list, if one is available. Some sort of documentation always exists, even if it is as minimal as a simple README file, so read whatever is available. If the project is old and reasonably large, the Web site will probably feature a *Frequently Asked Questions (FAQ)* page.

Next, join the development mailing list and lurk, which means to subscribe to a mailing list and read it without posting. Mailing lists are the preferred form of developer communication followed by, to a lesser extent, *Internet Relay Chat (IRC)* and online newsgroups, commonly referred to as *UseNet*. Because mailing lists often contain discussions on implementation details, it is important to read at least the previous months archives to get a feel for the developer community and current activity. The mailing list archives should be the first place to search if you have a question or query on the implementation that is not covered by available documentation. If you have a question to ask the developers, take time to research the questions and ask it the *Right Way* [RM01]. Although people will answer previous questions, you will not help your credibility by constantly asking questions that were answered a week previously or are clearly documented.

Now, how does all this apply to Linux? First, the documentation. A README is at the top of the source tree, and a wealth of information is available in the Documentation/ directory. A number of books on UNIX design [Vah96], Linux specifically [BC00] and of course this book are available to explain what to expect in the code.

One of the best online sources of information available on kernel development is the Kernel Page in the weekly edition of *Linux Weekly News* (<http://www.lwn.net>). This page also reports on a wide range of Linux-related topics and is worth a regular read. The kernel does not have a home Web site as such, but the closest equivalent is <http://www.kernelnewbies.org>, which is a vast

⁴<http://www.tldp.org/HOWTO/Kernel-HOWTO/index.html>

source of information on the kernel that is invaluable to new and experienced people alike.

An FAQ is available for the *Linux Kernel Mailing List (LKML)* at <http://www.tux.org/lkml/> that covers questions ranging from the kernel development process to how to join the list itself. The list is archived at many sites, but a common choice to reference is <http://marc.theaimsgroup.com/?l=linux-kernel>. Be aware that the mailing list is a very high volume list that can be a very daunting read, but a weekly summary is provided by the *Kernel Trap* site at <http://kt.zork.net/kernel-trap/c/>.

The sites and sources mentioned so far contain general kernel information, but memory management-specific sources are available too. A Linux-MM Web site at <http://www.linux-mm.org> contains links to memory management-specific documentation and a linux-mm mailing list. The list is relatively light in comparison to the main list and is archived at <http://mail.nl.linux.org/linux-mm/>.

The last site to consult is the *Kernel Trap* site at <http://www.kerneltrap.org>. The site contains many useful articles on kernels in general. It is not specific to Linux, but it does contain many Linux-related articles and interviews with kernel developers.

As is clear, a vast amount of information is available that may be consulted before resorting to the code. With enough experience, it will eventually be faster to consult the source directly, but, when getting started, check other sources of information first.

1.2 Managing the Source

The mainline or stock kernel is principally distributed as a compressed tape archive (.tar.bz) that is available from your nearest kernel source repository. In Ireland, case, it is <ftp://ftp.ie.kernel.org/>. The stock kernel is always considered to be the one released by the tree maintainer. For example, at time of writing, the stock kernels for 2.2.x are those released by Alan Cox⁵, for 2.4.x by Marcelo Tosatti and for 2.5.x by Linus Torvalds. At each release, the full tar file is available as well as a smaller *patch*, which contains the differences between the two releases. Patching is the preferred method of upgrading because of bandwidth considerations. Contributions made to the kernel are almost always in the form of patches, which are unified diffs generated by the GNU tool `diff`.

Why patches Sending patches to the mailing list initially sounds clumsy, but it is remarkably efficient in the kernel development environment. The principal advantage of patches is that it is much easier to read what changes have been made than to compare two full versions of a file side by side. A developer familiar with the code can easily see what impact the changes will have and if it should be merged. In addition, it is very easy to quote the email that includes the patch and request more information about it.

⁵Last minute update, Alan just announced he was going on sabbatical and will no longer maintain the 2.2.x tree. There is no maintainer at the moment.

Subtrees At various intervals, individual influential developers may have their own version of the kernel distributed as a large patch to the main tree. These subtrees generally contain features or cleanups that have not been merged to the mainstream yet or are still being tested. Two notable subtrees are the *-rmap* tree maintained by Rik Van Riel, a long-time influential VM developer, and the *-mm* tree maintained by Andrew Morton, the current maintainer of the stock development VM. The *-rmap* tree contains a large set of features that, for various reasons, are not available in the mainline. It is heavily influenced by the FreeBSD VM and has a number of significant differences from the stock VM. The *-mm* tree is quite different from *-rmap* in that it is a testing tree with patches that are being tested before merging into the stock kernel.

BitKeeper In more recent times, some developers have started using a source code control system called BitKeeper (<http://www.bitmover.com>), a proprietary version control system that was designed with Linux as the principal consideration. BitKeeper allows developers to have their own distributed version of the tree, and other users may pull sets of patches called *changesets* from each others' trees. This distributed nature is a very important distinction from traditional version control software that depends on a central server.

BitKeeper allows comments to be associated with each patch, and these are displayed as part of the release information for each kernel. For Linux, this means that the email that originally submitted the patch is preserved, making the progress of kernel development and the meaning of different patches a lot more transparent. On release, a list of the patch titles from each developer is announced, as well as a detailed list of all patches included.

Because BitKeeper is a proprietary product, email and patches are still considered the only method for generating discussion on code changes. In fact, some patches will not be considered for acceptance unless some discussion occurs *not* on the main mailing list because code quality is considered to be directly related to the amount of peer review [Ray02]. Because the BitKeeper maintained source tree is exported in formats accessible to open source tools like CVS, patches are still the preferred means of discussion. This means that developers are not required to use BitKeeper for making contributions to the kernel, but the tool is still something that developers should be aware of.

1.2.1 Diff and Patch

The two tools for creating and applying patches are *diff* and *patch*, both of which are GNU utilities available from the GNU website⁶. *diff* is used to generate patches, and *patch* is used to apply them. Although the tools have numerous options, there is a preferred usage.

Patches generated with *diff* should always be *unified diff*, include the C function that the change affects and be generated from one directory above the kernel source root. A *unified diff* includes more information than just the differences between two lines. It begins with a two-line header with the names and creation date of the

⁶<http://www.gnu.org>

two files that diff is comparing. After that, the file will consist of one or more hunks. The beginning of each hunk is marked with a line beginning with @@, which includes the starting line in the source code and how many lines there are before and after the hunk is applied. The hunk includes context lines that show lines above and below the changes to aid a human reader. Each line begins with a +, - or blank. If the mark is +, the line is added. If it is a -, the line is removed, and a blank is to leave the line alone because it is there just to provide context. The reasoning behind generating from one directory above the kernel root is that it is easy to see quickly what version the patch has been applied against. It also makes the scripting of applying patches easier if each patch is generated the same way.

Let us take, for example, a very simple change that has been made to mm/page_alloc.c, which adds a small piece of commentary. The patch is generated as follows. Note that this command should be all on one line minus the backslashes.

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

This generates a unified context diff (-u switch) between two files and places the patch in example.patch as shown in Figure 1.1. It also displays the name of the affected C function.

From this patch, it is clear even at a casual glance which files are affected (page_alloc.c) and which line it starts at (76), and the new lines added are clearly marked with a +. In a patch, there may be several hunks that are marked with a line starting with @@. Each hunk will be treated separately during patch application.

Broadly speaking, patches come in two varieties: plain text such as the previous one that is sent to the mailing list and compressed patches that are compressed with either gzip (.gz extension) or bzip2 (.bz2 extension). It is usually safe to assume that patches were generated one directory above the root of the kernel source tree. This means that, although the patch is generated one directory above, it may be applied with the option -p1 while the current directory is the kernel source tree root.

Broadly speaking, this means a plain text patch to a clean tree can be easily applied as follows:

```
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/ $
```

To apply a compressed patch, it is a simple extension to just decompress the patch to standard out (stdout) first.

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

```

--- 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)
{

```

Figure 1.1. Example Patch

If a hunk can be applied, but the line numbers are different, the hunk number and the number of lines that need to be offset will be output. These are generally safe warnings and may be ignored. If there are slight differences in the context, the hunk will be applied, and the level of fuzziness will be printed, which should be double-checked. If a hunk fails to apply, it will be saved to `filename.c.rej`, and the original file will be saved to `filename.c.orig` and have to be applied manually.

1.2.2 Basic Source Management With PatchSet

The untarring of sources, management of patches and building of kernels is initially interesting, but quickly palls. To cut down on the tedium of patch management, a simple tool was developed while writing this book called **PatchSet**, which is designed to easily manage the kernel source and patches and to eliminate

a large amount of the tedium. It is fully documented and freely available from <http://www.csn.ul.ie/~mel/projects/patchset/> and on the companion CD.

Downloading Downloading kernels and patches in itself is quite tedious, and scripts are provided to make the task simpler. First, the configuration file `etc/patchset.conf` should be edited, and the `KERNEL_MIRROR` parameter should be updated for your local <http://www.kernel.org/> mirror. After that is done, use the script `download` to download patches and kernel sources. A simple use of the script is as follows:

```
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
```

After the relevant sources or patches have been downloaded, it is time to configure a kernel build.

Configuring Builds Files called `set configuration` files are used to specify what kernel source tar to use, what patches to apply, what kernel configuration (generated by `make config`) to use and what the resulting kernel is to be called. A sample specification to build kernel `2.4.20-rmap15f` is:

```
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
```

This first line says to unpack a source tree starting with `linux-2.4.18.tar.gz`. The second line specifies that the kernel will be called `2.4.20-rmap15f`. `2.4.20` was selected for this example because `rmap` patches against a later stable release were not available at the time of writing. To check for updated `rmap` patches, see <http://surriel.com/patches/>. The third line specifies which kernel `.config` to use for compiling the kernel. Each line after that has two parts. The first part says what patch depth to use, that is, what number to use with the `-p` switch to patch. As discussed earlier in Section 1.2.1, this is usually 1 for applying patches while in the source directory. The second is the name of the patch stored in the patches directory. The previous example will apply two patches to update the kernel from `2.4.18` to `2.4.20` before building the `2.4.20-rmap15f` kernel tree.

If the kernel configuration required is very simple, use the `createset` script to generate a set file for you. It simply takes a kernel version as a parameter and

guesses how to build it based on available sources and patches.

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

Building a Kernel The package comes with three scripts. The first script, called `make-kernel.sh`, will unpack the kernel to the `kernels/` directory and build it if requested. If the target distribution is Debian, it can also create Debian packages for easy installation by specifying the `-d` switch. The second script, called `make-gengraph.sh`, will unpack the kernel, but, instead of building an installable kernel, it will generate the files required to use `CodeViz`, discussed in the next section, for creating call graphs. The last, called `make-lxr.sh`, will install a kernel for use with LXR.

Generating Diffs Ultimately, you will need to see the difference between files in two trees or generate a diff of changes you have made yourself. Three small scripts are provided to make this task easier. The first is `setclean`, which sets the source tree to compare from. The second is `setworking` to set the path of the kernel tree you are comparing against or working on. The third is `diff-tree`, which will generate diffs against files or directories in the two trees. To generate the diff shown in Figure 1.1, the following would have worked:

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

The generated diff is a unified diff with the C function context included and complies with the recommended use of diff. Two additional scripts are available that are very useful when tracking changes between two trees. They are `diff-struct` and `diff-func`. These are for printing out the differences between individual structures and functions. When used with diff, the `-f` switch must be used to record what source file the structure or function is declared in, but it is only needed the first time.

1.3 Browsing the Code

When code is small and manageable, browsing through the code is not particularly difficult because operations are clustered together in the same file, and there is not much coupling between modules. The kernel, unfortunately, does not always exhibit this behavior. Functions of interest may be spread across multiple files or contained as inline functions in headers. To complicate matters, files of interest may be buried beneath architecture-specific directories, which makes tracking them down time consuming.

One solution for easy code browsing is `ctags` (<http://ctags.sourceforge.net/>), which generates tag files from a set of source files. These tags can be used to jump to the C file and line where the identifier is declared with editors such as `Vi` and `Emacs`. In the event there are multiple instances of the same tag, such as with multiple functions with the same name, the correct one may be selected from a list. This method works best when editing the code because it allows very fast navigation through the code to be confined to one terminal window.

A more friendly browsing method is available with the *LXR* tool hosted at <http://lxr.linux.no/>. This tool provides the ability to represent source code as browsable Web pages. Identifiers such as global variables, macros and functions become hyperlinks. When clicked, the location where the identifier is defined is displayed along with every line and line referencing the definition. This makes code navigation very convenient and is almost essential when reading the code for the first time.

The tool is very simple to install, and a browsable version of the kernel 2.4.22 source is available on the CD included with this book. All code extracts throughout the book are based on the output of LXR so that the line numbers would be clearly visible in excerpts.

1.3.1 Analyzing Code Flow

Because separate modules share code across multiple C files, it can be difficult to see what functions are affected by a given code path without tracing through all the code manually. For a large or deep code path, this can be extremely time consuming to answer what should be a simple question.

One simple, but effective, tool to use is *CodeViz*, which is a call graph generator and is included with the CD. It uses a modified compiler for either C or C++ to collect information necessary to generate the graph. The tool is hosted at <http://www.csn.ul.ie/~mel/projects/codeviz/>.

During compilation with the modified compiler, files with a .cdep extension are generated for each C file. This .cdep file contains all function declarations and calls made in the C file. These files are distilled with a program called *genfull* to generate a full call graph of the entire source code, which can be rendered with *dot*, part of the *GraphViz* project hosted at <http://www.graphviz.org/>.

In the kernel compiled for the computer this book was written on, a total of 40,165 entries were in the *full.graph* file generated by *genfull*. This call graph is essentially useless on its own because of its size, so a second tool is provided called *gengraph*. This program, at basic usage, takes the name of one or more functions as an argument and generates a postscript file with the call graph of the requested function as the root node. The postscript file may be viewed with *ghostview* or *gv*.

The generated graphs can be to an unnecessary depth or show functions that the user is not interested in, so there are three limiting options to graph generation. The first is limit by depth where functions that are greater than *N* levels deep in a call chain are ignored. The second is to totally ignore a function so that it will not appear on the call graph or any of the functions it calls. The last is to display a function, but not traverse it, which is convenient when the function is covered on a separate call graph or is a known API with an implementation that is not currently of interest.

All call graphs shown in these documents are generated with the *CodeViz* tool because it is often much easier to understand a subsystem at a glance when a call graph is available. The tool has been tested with a number of other open source projects based on C and has a wider application than just the kernel.

1.3.2 Simple Graph Generation

If both PatchSet and CodeViz are installed, the `%g` call graph in this book shown in Figure 3.4 can be generated and viewed with the following set of commands. For brevity, the output of the commands is omitted:

```
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 Reading the Code

When new developers or researchers ask how to start reading the code, experienced developers often recommend starting with the initialization code and working from there. This may not be the best approach for everyone because initialization is quite architecture dependent and requires detailed hardware knowledge to decipher it. It also gives very little information on how a subsystem like the VM works. It is during the late stages of initialization that memory is set up in the way the running system sees it.

The best starting point to understand the VM is this book and the code commentary. It describes a VM that is reasonably comprehensive without being overly complicated. Later VMs are more complex, but are essentially extensions of the one described here.

For when the code has to be approached afresh with a later VM, it is always best to start in an isolated region that has the minimum number of dependencies. In the case of the VM, the best starting point is the *Out Of Memory (OOM)* manager in `mm/oom_kill.c`. It is a very gentle introduction to one corner of the VM where a process is selected to be killed in the event that memory in the system is low. Because this function touches so many different aspects of the VM, it is covered last in this book. The second subsystem to then examine is the noncontiguous memory allocator located in `mm/vmalloc.c` and discussed in Chapter 7 because it is reasonably contained within one file. The third system should be the physical page allocator located in `mm/page_alloc.c` and discussed in Chapter 6 for similar reasons. The fourth system of interest is the creation of Virtual Memory Addresses (VMAs) and memory areas for processes discussed in Chapter 4. Between these systems, they have the bulk of the code patterns that are prevalent throughout the rest of the kernel code, which makes the deciphering of more complex systems such as the page replacement policy or the buffer Input/Output (I/O) much easier to comprehend.

The second recommendation that is given by experienced developers is to benchmark and test the VM. Many benchmark programs are available, but commonly used ones are ConTest(<http://members.optusnet.com.au/ckolivas/contest/>), SPEC(<http://www.specbench.org/>), lmbench(<http://www.bitmover.com/lmbench/>)

and `dbench`(<http://freshmeat.net/projects/dbench/>). For many purposes, these benchmarks will %the requirements.

Unfortunately, it is difficult to test just the VM accurately and benchmarking it is frequently based on timing a task such as a kernel compile. A tool called `VM Regress` is available at <http://www.csn.ul.ie/~mel/projects/vmregress/> that lays the foundation required to build a fully %aged testing, regression and benchmarking tool for the VM. `VM Regress` uses a combination of kernel modules and userspace tools to test small parts of the VM in a reproducible manner and has one benchmark for testing the page replacement policy using a large reference string. It is intended as a framework for the development of a testing utility and has a number of Perl libraries and helper kernel modules to do much of the work. However, it is still in the early stages of development, so use it with care.

1.5 Submitting Patches

Two %s, `SubmittingPatches` and `CodingStyle`, are in the `Documentation/` directory that cover the important basics. However, very little documentation describes how to get patches merged. This section will give a brief introduction on how, broadly speaking, patches are managed.

First and foremost, the coding style of the kernel needs to be adhered to because having a style inconsistent with the main kernel will be a barrier to getting merged regardless of the technical merit. After a patch has been developed, the %st problem is to decide where to send it. Kernel development has a definite, if nonapparent, hierarchy of who handles patches and how to get them submitted. As an example, we %take the case of 2.5.x development.

The %st check to make is if the patch is very small or trivial. If it is, post it to the main kernel mailing list. If no bad reaction occurs, it can be fed to what is called the *Trivial Patch Monkey*⁷. The trivial patch monkey is exactly what it sounds like. It takes small patches and feeds them en masse to the correct people. This is best suited for documentation, commentary or one-liner patches.

Patches are managed through what could be loosely called a set of rings with Linus in the very middle having the %al say on what gets accepted into the main tree. Linus, with rare exceptions, accepts patches only from who he refers to as his %utenants, %o group of around 10 people who he trusts to %eed him correct code. An example lieutenant is Andrew Morton, the VM maintainer at time of writing. Any change to the VM has to be accepted by Andrew before it will get to Linus. These people are generally maintainers of a particular system, but sometimes will %eed him patches from another subsystem if they feel it is important enough.

Each of the lieutenants are active developers on different subsystems. Just like Linus, they have a small set of developers they trust to be knowledgeable about the patch they are sending, but will also pick up patches that affect their subsystem more readily. Depending on the subsystem, the list of people they trust will be heavily influenced by the list of maintainers in the `MAINTAINERS` %o. The second major area of influence will be from the subsystem-specific mailing list if there is

⁷<http://www.kernel.org/pub/linux/kernel/people/rusty/trivial/>

one. The VM does not have a list of maintainers, but it does have a mailing list⁸.

The maintainers and lieutenants are crucial to the acceptance of patches. Linus, broadly speaking, does not appear to want to be convinced with argument alone on the merit for a significant patch, but prefers to hear it from one of his lieutenants, which is understandable considering the volume of patches that exist.

In summary, a new patch should be emailed to the subsystem mailing list and cc'd to the main list to generate discussion. If no reaction occurs, it should be sent to the maintainer for that area of code if there is one and to the lieutenant if there is not. After it has been picked up by a maintainer or lieutenant, chances are it will be merged. The important key is that patches and ideas must be released early and often so developers have a chance to look at them while they are still manageable. There are notable cases where massive patches merged with the main tree because there were long periods of silence with little or no discussion. A recent example of this is the Linux Kernel Crash Dump project, which still has not been merged into the mainstream because there has not been enough favorable feedback from lieutenants or strong support from vendors.

⁸<http://www.linux-mm.org/maillinglists.shtml>

Describing Physical Memory

Linux is available for a wide range of architectures, so an architecture-independent way of describing memory is needed. This chapter describes the structures used to keep account of memory banks, pages and `pgs` that affect VM behavior.

The ~~first~~ principal concept prevalent in the VM is *Non Uniform Memory Access (NUMA)*. With large-scale machines, memory may be arranged into banks that incur a different cost to access depending on their distance from the processor. For example, a bank of memory might be assigned to each CPU, or a bank of memory very suitable for Direct Memory Access (DMA) near device cards might be assigned.

Each bank is called a *node*, and the concept is represented under Linux by a struct `pglist_data` even if the architecture is Uniform Memory Access (UMA). This struct is always referenced by its typedef `pg_data_t`. Every node in the system is kept on a NULL terminated list called `pgdat_list`, and each node is linked to the next with the ~~field~~ `pg_data_t → node_next`. For UMA architectures like PC desktops, only one static `pg_data_t` structure called `contig_page_data` is used. Nodes are discussed further in Section 2.1.

Each node is divided into a number of blocks called *zones*, which represent ranges within memory. Zones should not be confused with zone-based allocators because they are unrelated. A zone is described by a struct `zone_struct`, typedef'd to `zone_t`, and each one is of type `ZONE_DMA`, `ZONE_NORMAL` or `ZONE_HIGHMEM`. Each zone type is suitable for a different type of use. `ZONE_DMA` is memory in the lower physical memory ranges that certain Industry Standard Architecture (ISA) devices require. Memory within `ZONE_NORMAL` is directly mapped by the kernel into the upper region of the linear address space, which is discussed further in Section 4.1. `ZONE_HIGHMEM` is the remaining available memory in the system and is not directly mapped by the kernel.

With the x86, the zones are the following:

<code>ZONE_DMA</code>	First 16MiB of memory
<code>ZONE_NORMAL</code>	16MiB - 896MiB
<code>ZONE_HIGHMEM</code>	896 MiB - End

Many kernel operations can only take place using `ZONE_NORMAL`, so it is the most performance-critical zone. Zones are discussed further in Section 2.2. The system's memory is comprised of fixed-size chunks called *page frames*. Each physical page frame is represented by a struct `page`, and all the structs are kept in a global `mem_map` array, which is usually stored at the beginning of `ZONE_NORMAL` or just after


```

138     unsigned long node_size;
139     int node_id;
140     struct pglist_data *node_next;
141 } pg_data_t;

```

We now briefly describe each of these fields:

node_zones The zones for this node are ZONE_HIGHMEM, ZONE_NORMAL, ZONE_DMA.

node_zonelist This is the order of zones that allocations are preferred from. `build_zonelist()` in `mm/page_alloc.c` sets up the order when called by `free_area_init_core()`. A failed allocation in ZONE_HIGHMEM may fall back to ZONE_NORMAL or back to ZONE_DMA.

nr_zones This is the number of zones in this node between one and three. Not all nodes will have three. A CPU bank may not have ZONE_DMA, for example.

node_mem_map This is the `page` array that represents each physical frame in the node. It will be placed somewhere within the global `mem_map` array.

valid_addr_bitmap This is a bitmap that describes holes in the memory node that no memory exists for. In reality, this is only used by the Sparc and Sparc64 architectures and is ignored by all others.

bdata This is only of interest to the boot memory allocator discussed in Chapter 5.

node_start_paddr This is the starting physical address of the node. An unsigned long does not work optimally because it breaks for ia32 with *Physical Address Extension (PAE)* and for some PowerPC variants such as the PPC440GP. PAE is discussed further in Section 2.7. A more suitable solution would be to record this as a *Page Frame Number (PFN)*. A PFN is simply an index within physical memory that is counted in page-sized units. PFN for a physical address could be trivially derived as `(page_phys_addr >> PAGE_SHIFT)`.

node_start_mapnr This gives the page offset within the global `mem_map`. It is calculated in `free_area_init_core()` by calculating the number of pages between `mem_map` and the local `mem_map` for this node called `lmem_map`.

node_size This is the total number of pages in this zone.

node_id This is the *Node ID (NID)* of the node and starts at 0.

node_next Pointer to next node in a NULL terminated list.

All nodes in the system are maintained on a list called `pgdat_list`. The nodes are placed on this list as they are initialized by the `init_bootmem_core()` function, which is described later in Section 5.3. Up until late 2.4 kernels (> 2.4.18), blocks of code that traversed the list looked something like the following:

```

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

```

In more recent kernels, a macro `for_each_pgdat()`, which is trivially defined as a loop, is provided to improve code readability.

2.2 Zones

Each zone is described by a struct `zone_struct`. `zone_structs` keep track of information like page usage statistics, free area information and locks. They are declared as follows in `<linux/mmzone.h>`:

```

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;

```

This is a brief explanation of each field in the struct.

lock Spinlock protects the zone from concurrent accesses.

free_pages The total number of free pages in the zone.

pages_min, pages_low and pages_high These are zone watermarks that are described in the next section.

need_balance This tells the pageout `kswapd` to balance the zone. A zone is said to need balance when the number of available pages reaches one of the *zone watermarks*. Watermarks are discussed in the next section.

free_area These are free area bitmaps used by the buddy allocator.

wait_table This is a hash table of wait queues of processes waiting on a page to be freed. This is of importance to `wait_on_page()` and `unlock_page()`. Although processes could all wait on one queue, this would cause all waiting processes to race for pages still locked when woken up. A large group of processes contending for a shared resource like this is sometimes called a thundering herd. Wait tables are discussed further in Section 2.2.3.

wait_table_size This is the number of queues in the hash table, which is a power of 2.

wait_table_shift This is defined as the number of bits in a long minus the binary logarithm of the table size above.

zone_pgdat This points to the parent `pg_data_t`.

zone_mem_map This is the first page in the global `mem_map` that this zone refers to.

zone_start_paddr This uses the same principle as `node_start_paddr`.

zone_start_mapnr This uses the same principle as `node_start_mapnr`.

name This is the string name of the zone: `Normal` or `HighMem`.

size This is the size of the zone in pages.

2.2.1 Zone Watermarks

When available memory in the system is low, the pageout daemon `kswapd` is woken up to start freeing pages (see Chapter 10). If the pressure is high, the process will free up memory synchronously, sometimes referred to as the *direct-reclaim* path. The parameters affecting pageout behavior are similar to those used by FreeBSD [McK96] and Solaris [MM01].

Each zone has three watermarks called `pages_low`, `pages_min` and `pages_high`, which help track how much pressure a zone is under. The relationship between them is illustrated in Figure 2.2. The number of pages for `pages_min` is calculated in the function `free_area_init_core()` during memory init and is based on a ratio to the size of the zone in pages. It is calculated initially as `ZoneSizeInPages/128`. The lowest value it will be is 20 pages (80K on a x86), and the highest possible value is 255 pages (1MiB on a x86).

At each watermark a different action is taken to address the memory shortage.

pages_low When the `pages_low` number of free pages is reached, `kswapd` is woken up by the buddy allocator to start freeing pages. This is equivalent to when `lotsfree` is reached in Solaris and `freemin` in FreeBSD. The value is twice the value of `pages_min` by default.

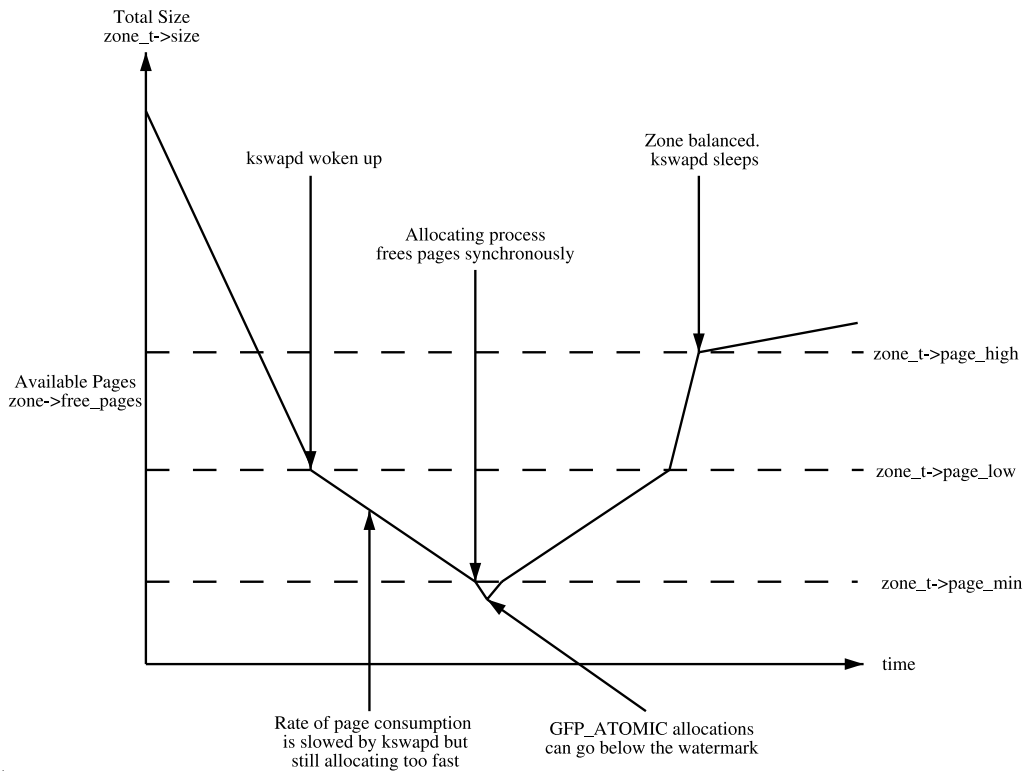


Figure 2.2. Zone Watermarks

pages_min When `pages_min` is reached, the allocator will do the `kswapd` work in a synchronous fashion, sometimes referred to as the *direct-reclaim* path. Solaris does not have a real equivalent, but the closest is the `desfree` or `minfree`, which determine how often the pageout scanner is woken up.

pages_high After `kswapd` has been woken to start freeing pages, it will not consider the zone to be balanced when `pages_high` pages are free. After the watermark has been reached, `kswapd` will go back to sleep. In Solaris, this is called `lotsfree`, and, in BSD, it is called `free_target`. The default for `pages_high` is three times the value of `pages_min`.

Whatever the pageout parameters are called in each operating system, the meaning is the same. It helps determine how hard the pageout daemon or processes work to free up pages.

2.2.2 Calculating the Size of Zones

The size of each zone is calculated during `setup_memory()`, shown in Figure 2.3.

The PFN is an offset, counted in pages, within the physical memory map. The first PFN usable by the system, `min_low_pfn`, is located at the beginning of the

the last page after `_end`, which is the end of the loaded kernel image. The value is stored as a global scope variable in `mm/bootmem.c` for use with the boot memory allocator.

How the last page frame in the system, `max_pfn`, is calculated is quite architecture specific. In the x86 case, the function `find_max_pfn()` reads through the whole `e820` map for the highest page frame. The value is also stored as a global scope variable in `mm/bootmem.c`. The `e820` is a table provided by the BIOS describing what physical memory is available, reserved or nonexistent.

The value of `max_low_pfn` is calculated on the x86 with `find_max_low_pfn()`, and it marks the end of `ZONE_NORMAL`. This is the physical memory directly accessible by the kernel and is related to the kernel/userspace split in the linear address space marked by `PAGE_OFFSET`. The value, with the others, is stored in `mm/bootmem.c`. In low memory machines, the `max_pfn` will be the same as the `max_low_pfn`.

With the three variables `min_low_pfn`, `max_low_pfn` and `max_pfn`, it is straightforward to calculate the start and end of high memory and place them as global scope variables in `arch/i386/mm/init.c` as `highstart_pfn` and `highend_pfn`. The values are used later to initialize the high memory pages for the physical page allocator, as we will see in Section 5.6.

2.2.3 Zone Wait Queue Table

When I/O is being performed on a page, such as during page-in or page-out, the I/O is locked to prevent accessing it with inconsistent data. Processes that want to use it have to join a wait queue before the I/O can be accessed by calling `wait_on_page()`. When the I/O is completed, the page will be unlocked with `UnlockPage()`, and any process waiting on the queue will be woken up. Each page could have a wait queue, but it would be very expensive in terms of memory to have so many separate queues. Instead, the wait queue is stored in the `zone_t`. The basic process is shown in Figure 2.4.

It is possible to have just one wait queue in the zone, but that would mean that all processes waiting on any page in a zone would be woken up when one was unlocked. This would cause a serious *thundering herd* problem. Instead, a hash table of wait queues is stored in `zone_t→wait_table`. In the event of a hash collision, processes may still be woken unnecessarily, but collisions are not expected to occur frequently.

The table is allocated during `free_area_init_core()`. The size of the table is calculated by `wait_table_size()` and is stored in `zone_t→wait_table_size`. The maximum size it will be is 4,096 wait queues. For smaller tables, the size of the table is the minimum power of 2 required to store `NoPages / PAGE_PER_WAITQUEUE` number of queues, where `NoPages` is the number of pages in the zone and `PAGE_PER_WAITQUEUE` is defined to be 256. In other words, the size of the table is calculated as the integer component of the following equation:

$$\text{wait\_table\_size} = \log_2\left(\frac{\text{NoPages} - 1}{\text{PAGE\_PER\_WAITQUEUE} - 1}\right) + 1$$

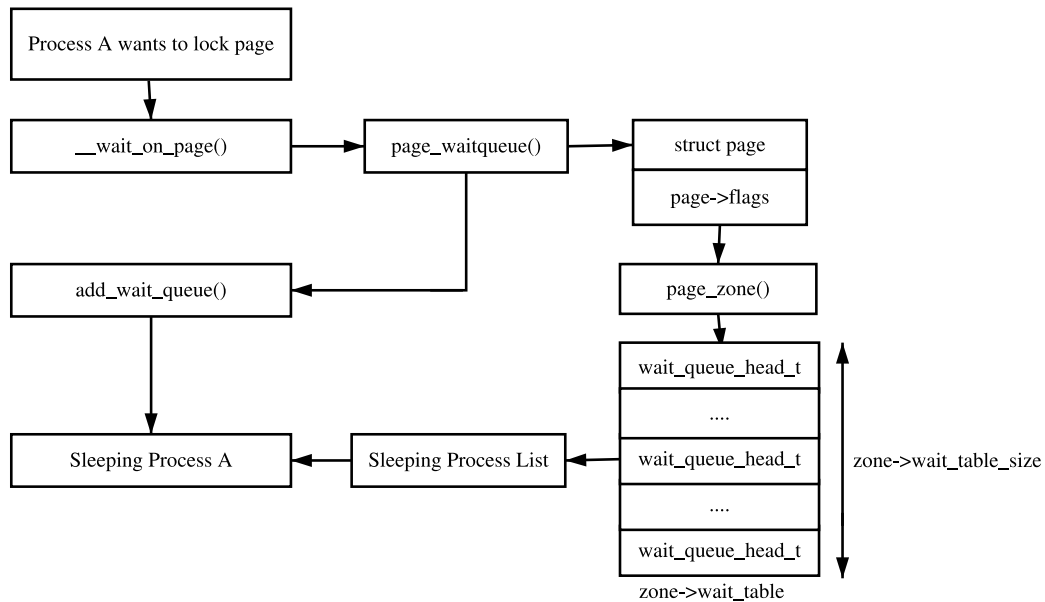


Figure 2.4. Sleeping on a Locked Page

The `zone_t->wait_table_shift` is calculated as the number of bits a page address must be shifted right to return an index within the table. The function `page_waitqueue()` is responsible for returning which wait queue to use for a page in a zone. It uses a simple multiplicative hashing algorithm based on the virtual address of the struct page being hashed.

`page_waitqueue` works by simply multiplying the address by `GOLDEN_RATIO_PRIME` and shifting the result `zone_t->wait_table_shift` bits right to index the result within the hash table. `GOLDEN_RATIO_PRIME[Lev00]` is the largest prime that is closest to the *golden ratio*[Knu68] of the largest integer that may be represented by the architecture.

2.3 Zone Initialization

The zones are initialized after the kernel page tables have been fully set up by `paging_init()`. Page table initialization is covered in Section 3.6. Predictably, each architecture performs this task differently, but the objective is always the same: to determine what parameters to send to either `free_area_init()` for UMA architectures or `free_area_init_node()` for NUMA. The only parameter required for UMA is `zones_size`. The full list of parameters follows:

`nid` is the NodeID that is the logical identity of the node whose zones are being initialized.

`pgdat` is the node's `pg_data_t` that is being initialized. In UMA, this will simply be `contig_page_data`.

`pmap` is set later by `free_area_init_core()` to point to the beginning of the local `lmem_map` array allocated for the node. In NUMA, this is ignored because NUMA treats `mem_map` as a virtual array starting at `PAGE_OFFSET`. In UMA, this pointer is the global `mem_map` variable, which is now `mem_map`, and gets initialized in UMA.

`zones_sizes` is an array containing the size of each zone in pages.

`zone_start_paddr` is the starting physical address for the `zone`.

`zone_holes` is an array containing the total size of memory holes in the zones.

The core function `free_area_init_core()` is responsible for filling in each `zone_t` with the relevant information and the allocation of the `mem_map` array for the node. Information on what pages are free for the zones is not determined at this point. That information is not known until the boot memory allocator is being retired, which will be discussed in Chapter 5.

2.4 Initializing mem_map

The `mem_map` area is created during system startup in one of two fashions. On NUMA systems, the global `mem_map` is treated as a virtual array starting at `PAGE_OFFSET`. `free_area_init_node()` is called for each active node in the system, which allocates the portion of this array for the node being initialized. On UMA systems, `free_area_init()` uses `contig_page_data` as the node and the global `mem_map` as the local `mem_map` for this node. The call graph for both functions is shown in Figure 2.5.

The core function `free_area_init_core()` allocates a local `lmem_map` for the node being initialized. The memory for the array is allocated from the boot memory allocator with `alloc_bootmem_node()` (see Chapter 5). With UMA architectures, this newly allocated memory becomes the global `mem_map`, but it is slightly different for NUMA.

NUMA architectures allocate the memory for `lmem_map` within their own memory node. The global `mem_map` never gets explicitly allocated, but instead is set to `PAGE_OFFSET` where it is treated as a virtual array. The address of the local map is stored in `pg_data_t → node_mem_map`, which exists somewhere within the virtual `mem_map`. For each zone that exists in the node, the address within the virtual `mem_map` for the zone is stored in `zone_t → zone_mem_map`. All the rest of the code then treats `mem_map` as a real array because only valid regions within it will be used by nodes.

2.5 Pages

Every physical page frame in the system has an associated struct `page` that is used to keep track of its status. In the 2.2 kernel [BC00], this structure resembled

to the `%`. If the page is anonymous and `mapping` is set, the `address_space` is `swapper_space`, which manages the swap address space.

index This `%d` has two uses, and the state of the page determines what it means. If the page is part of a `%` mapping, it is the offset within the `%`. If the page is part of the swap cache, this will be the offset within the `address_space` for the swap address space (`swapper_space`). Second, if a block of pages is being freed for a particular process, the order (power of two number of pages being freed) of the block being freed is stored in `index`. This is set in the function `--free-pages-ok()`.

next_hash Pages that are part of a `%` mapping are hashed on the inode and offset set. This `%d` links pages together that share the same hash bucket.

count This is the reference count to the page. If it drops to zero, it may be freed. If it is any greater, it is in use by one or more processes or is in use by the kernel like when waiting for I/O.

flags These are `%s` that describe the status of the page. All of them are declared in `<linux/mm.h>` and are listed in Table 2.1. A number of macros defined for testing, clearing and setting the bits are all listed in Table 2.2. The only really interesting `%s` is `SetPageUptodate()`, which calls an architecture-specific function, `arch_set_page_uptodate()`, if it is defined before setting the bit.

lru For the page replacement policy, pages that may be swapped out will exist on either the `active_list` or the `inactive_list` declared in `page_alloc.c`. This is the list head for these Least Recently Used (LRU) lists. These two lists are discussed in detail in Chapter 10.

pprev_hash This complement to `next_hash` is so that the hash can work as a doubly linked list.

buffers If a page has buffers for a block device associated with it, this `%d` is used to keep track of the `buffer_head`. An anonymous page mapped by a process may also have an associated `buffer_head` if it is backed by a swap `%`. This is necessary because the page has to be synced with backing storage in block-sized chunks defined by the underlying `%` system.

virtual Normally only pages from `ZONE_NORMAL` are directly mapped by the kernel. To address pages in `ZONE_HIGHMEM`, `kmap()` is used to map the page for the kernel, which is described further in Chapter 9. Only a fixed number of pages may be mapped. When a page is mapped, this is its virtual address.

The type `mem_map_t` is a typedef for `struct page`, so it can be easily referred to within the `mem_map` array.

Bit Name	Description
PG_active	This bit is set if a page is on the <code>active_list</code> LRU and cleared when it is removed. It marks a page as being hot.
PG_arch_1	Quoting directly from the code, <code>PG_arch_1</code> is an architecture-specific page state bit. The generic code guarantees that this bit is cleared for a page when it is entered into the page cache. This allows an architecture to defer the flushing of the D-Cache (See Section 3.9) until the page is mapped by a process.
PG_checked	This is only used by the Ext2 filesystem.
PG_dirty	This indicates if a page needs to be flushed to disk. When a page is written to that is backed by disk, it is not flushed immediately. This bit is needed to ensure a dirty page is not freed before it is written out.
PG_error	If an error occurs during disk I/O, this bit is set.
PG_fs_1	This bit is reserved for a filesystem to use for its own purposes. Currently, only NFS uses it to indicate if a page is in sync with the remote server.
PG_highmem	Pages in high memory cannot be mapped permanently by the kernel. Pages that are in high memory are tagged with this bit during <code>mem_init()</code> .
PG_laundry	This bit is important only to the page replacement policy. When the VM wants to swap out a page, it will set this bit and call the <code>writepage()</code> function. When scanning, if it encounters a page with this bit and <code>PG_locked</code> set, it will wait for the I/O to complete.
PG_locked	This bit is set when the page must be locked in memory for disk I/O. When I/O starts, this bit is set and released when it completes.
PG_lru	If a page is on either the <code>active_list</code> or the <code>inactive_list</code> , this bit will be set.
PG_referenced	If a page is mapped and it is referenced through the mapping, index hash table, this bit is set. It is used during page replacement for moving the page around the LRU lists.
PG_reserved	This is set for pages that can never be swapped out. It is set by the boot memory allocator (See Chapter 5) for pages allocated during system startup. Later it is used to tag empty pages or ones that do not even exist.
PG_slab	This will tag a page as being used by the slab allocator.
PG_skip	This was used by some Sparc architectures to skip over parts of the address space but is no longer used. In 2.6, it is totally removed.
PG_unused	This bit is literally unused.
PG_uptodate	When a page is read from disk without error, this bit will be set.

Table 2.1. Flags Describing Page Status

Bit Name	Set	Test	Clear
PG_active	SetActive()	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()

Table 2.2. Macros for Testing, Setting and Clearing page → flags Status Bits

2.6 Mapping Pages to Zones

Up until as recently as kernel 2.4.18, a `struct page` stored a reference to its zone with `page→zone`, which was later considered wasteful, because even such a small pointer consumes a lot of memory when thousands of `struct page`s exist. In more recent kernels, the `zone` field has been removed and instead the top `ZONE_SHIFT` (8 in the x86) bits of the `page→flags` are used to determine the zone that a page belongs to. First, a `zone_table` of zones is set up. It is declared in `mm/page_alloc.c` as:

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

`MAX_NR_ZONES` is the maximum number of zones that can be in a node, i.e., three. `MAX_NR_NODES` is the maximum number of nodes that may exist. The function `EXPORT_SYMBOL()` makes `zone_table` accessible to loadable modules. This table is treated like a multidimensional array. During `free_area_init_core()`, all the pages in a node are initialized. First, it sets the value for the table

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

Where `nid` is the node ID, `j` is the zone index and `zone` is the `zone_t` struct. For each page, the function `set_page_zone()` is called as:

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

The parameter `page` is the page for which the zone is being set. Therefore, clearly the index in the `zone_table` is stored in the page.

2.7 High Memory

Because the address space usable by the kernel (`ZONE_NORMAL`) is limited in size, the kernel has support for the concept of high memory. Two thresholds of high memory exist on 32-bit x86 systems, one at 4GiB and a second at 64GiB. The 4GiB limit is related to the amount of memory that may be addressed by a 32-bit physical address. To access memory between the range of 1GiB and 4GiB, the kernel temporarily maps pages from high memory into `ZONE_NORMAL` with `kmap()`. This is discussed further in Chapter 9.

The second limit at 64GiB is related to *PAE*, which is an Intel invention to allow more RAM to be used with 32-bit systems. It makes four extra bits available for the addressing of memory, allowing up to 2^{36} bytes (64GiB) of memory to be addressed.

PAE allows a processor to address up to 64GiB in theory but, in practice, processes in Linux still cannot access that much RAM because, the virtual address space is still only 4GiB. This has led to some disappointment from users who have tried to `malloc()` all their RAM with one process.

Second, PAE does not allow the kernel itself to have this much RAM available. The `struct page` used to describe each page frame still requires 44 bytes,

and this uses kernel virtual address space in `ZONE_NORMAL`. That means that to describe 1GiB of memory, approximately 11MiB of kernel memory is required. Thus, with 16GiB, 176MiB of memory is consumed, putting significant pressure on `ZONE_NORMAL`. This does not sound too bad until other structures are taken into account that use `ZONE_NORMAL`. Even very small structures, such as *Page Table Entries (PTEs)*, require about 16MiB in the worst case. This makes 16GiB about the practical limit for available physical memory of Linux on an x86. If more memory needs to be accessed, the advice given is simple and straightforward. Buy a 64-bit machine.

2.8 What's New in 2.6

Nodes At first glance, there have not been many changes made to how memory is described, but the seemingly minor changes are wide reaching. The node descriptor `pg_data_t` has a few new fields that are as follows:

`node_start_pfn` replaces the `node_start_paddr` field. The only difference is that the new field is a PFN instead of a physical address. This was changed because PAE architectures can address more memory than 32 bits can address, so nodes starting over 4GiB would be unreachable with the old field.

`kswapd_wait` is a new wait queue for `kswapd`. In 2.4, there was a global wait queue for the page swapper daemon. In 2.6, there is one `kswapdN` for each node where `N` is the node identifier and each `kswapd` has its own wait queue with this field.

The `node_size` field has been removed and replaced instead with two fields. The change was introduced to recognize the fact that nodes may have holes in them where no physical memory is backing the address.

`node_present_pages` is the total number of physical pages that are present in the node.

`node_spanned_pages` is the total area that is addressed by the node, including any holes that may exist.

Zones Even at first glance, zones look very different. They are no longer called `zone_t`, but instead are referred to as simply `struct zone`. The second major difference is the LRU lists. As we see in Chapter 10, kernel 2.4 has a global list of pages that determine the order pages are freed or paged out. These lists are now stored in the `struct zone`. The relevant fields are the following:

`lru_lock` is the spinlock for the LRU lists in this zone. In 2.4, this is a global lock called `pagemap_lru_lock`.

`active_list` is the active list for this zone. This list is the same as described in Chapter 10 except it is now per-zone instead of global.

`inactive_list` is the inactive list for this zone. In 2.4, it is global.

`remove_counter` is the number of pages to remove from the `active_list` in one pass and only of interest during page replacement.

`nr_active` is the number of pages on the `active_list`.

`nr_inactive` is the number of pages on the `inactive_list`.

`all_unreclaimable` is set to 1 if the pageout daemon scans through all the pages in the zone twice and still fails to free enough pages.

`pages_scanned` is the number of pages scanned since the last bulk amount of pages has been reclaimed. In 2.6, lists of pages are freed at once rather than freeing pages individually, which is what 2.4 does.

`pressure` measures the scanning intensity for this zone. It is a decaying average that affects how hard a page scanner will work to reclaim pages.

Three other fields are new, but they are related to the dimensions of the zone. They are the following:

`zone_start_pfn` is the starting PFN of the zone. It replaces the `zone_start_paddr` and `zone_start_mapnr` fields in 2.4.

`spanned_pages` is the number of pages this zone spans, including holes in memory that exist with some architectures.

`present_pages` is the number of real pages that exist in the zone. For many architectures, this will be the same value as `spanned_pages`.

The next addition is `struct per_cpu_pageset`, which is used to maintain lists of pages for each CPU to reduce spinlock contention. The `zone->pageset` field is an `NR_CPU`-sized array of `struct per_cpu_pageset` where `NR_CPU` is the compiled upper limit of number of CPUs in the system. The per-cpu struct is discussed further at the end of the section.

The last addition to `struct zone` is the inclusion of padding of zeros in the struct. Development of the 2.6 VM recognized that some spinlocks are very heavily contended and are frequently acquired. Because it is known that some locks are almost always acquired in pairs, an effort should be made to ensure they use different cache lines, which is a common cache programming trick [Sea00]. This padding in the struct `zone` is marked with the `ZONE_PADDING()` macro and is used to ensure the `zone->lock`, `zone->lru.lock` and `zone->pageset` fields use different cache lines.

Pages The first noticeable change is that the ordering of fields has been changed so that related items are likely to be in the same cache line. The fields are essentially the same except for two additions. The first is a new union used to create a PTE chain. PTE chains are related to page table management, so will be discussed at the end of Chapter 3. The second addition is the `page->private` field, which contains private information specific to the mapping. For example, the field is used

to store a pointer to a `buffer_head` if the page is a buffer page. This means that the `page→buffers` field has also been removed. The last important change is that `page→virtual` is no longer necessary for high memory support and will only exist if the architecture specifically requests it. How high memory pages are supported is discussed further in Chapter 9.

Per-CPU Page Lists In 2.4, only one subsystem actively tries to maintain per-cpu lists for any object, and that is the Slab Allocator, which is discussed in Chapter 8. In 2.6, the concept is much more widespread, and there is a formalized concept of hot and cold pages.

The struct `per_cpu_pageset`, declared in `<linux/mmzone.h>`, has one field, which is an array with two elements of type `per_cpu_pages`. The zeroth element of this array is for hot pages, and the first element is for cold pages where hot and cold determines how active the page is currently in the cache. When it is known for a fact that the pages are not to be referenced soon, such as with I/O readahead, they will be allocated as cold pages.

The struct `per_cpu_pages` maintains a count of the number of pages currently in the list, a high and low watermark that determines when the set should be refilled or pages freed in bulk, a variable that determines how many pages should be allocated in one block and, finally, the actual list head of pages.

To build upon the per-cpu page lists, there is also a per-cpu page accounting mechanism. A struct `page_state` holds a number of accounting variables, such as the `pgalloc` field, which tracks the number of pages allocated to this CPU, and `pswpin`, which tracks the number of swap reads. The struct is heavily commented in `<linux/page-flags.h>`. A single function `mod_page_state()` is provided for updating fields in the `page_state` for the running CPU, and three helper macros are provided and are called `inc_page_state()`, `dec_page_state()` and `sub_page_state()`.

Page Table Management

Linux layers the machine independent/dependent layer in an unusual manner in comparison to other operating systems [CP99]. Other operating systems have objects that manage the underlying physical pages, such as the `pmap` object in BSD. Linux instead maintains the concept of a three-level page table in the architecture-independent code even if the underlying architecture does not support it. Although this is conceptually easy to understand, it also means that the distinction between different types of pages is very blurry, and page types are identified by their flags or what lists they exist on rather than the objects they belong to.

Architectures that manage their *Memory Management Unit (MMU)* differently are expected to emulate the three-level page tables. For example, on the x86 without PAE enabled, only two page table levels are available. The *Page Middle Directory (PMD)* is defined to be of size 1 and folds back directly onto the *Page Global Directory (PGD)*, which is optimized out at compile time. Unfortunately, for architectures that do not manage their cache or *Translation Lookaside Buffer (TLB)* automatically, hooks that are architecture dependent have to be explicitly left in the code for when the TLB and CPU caches need to be altered and flushed, even if they are null operations on some architectures like the x86. These hooks are discussed further in Section 3.8.

This chapter will begin by describing how the page table is arranged and what types are used to describe the three separate levels of the page table. Next is how a virtual address is broken up into its component parts for navigating the table. After this is covered, I discuss the lowest level entry, the *PTE*, and what bits are used by the hardware. After that, the macros used for navigating a page table and setting and checking attributes will be discussed before talking about how the page table is populated and how pages are allocated and freed for the use with page tables. The initialization stage is then discussed, which shows how the page tables are initialized during boot strapping. Finally, I cover how the TLB and CPU caches are utilized.

3.1 Describing the Page Directory

Each process is a pointer (`mm_struct → pgd`) to its own *PGD* which is a physical page frame. This frame contains an array of type `pgd_t`, which is an architecture-specific type defined in `<asm/page.h>`. The page tables are loaded differently

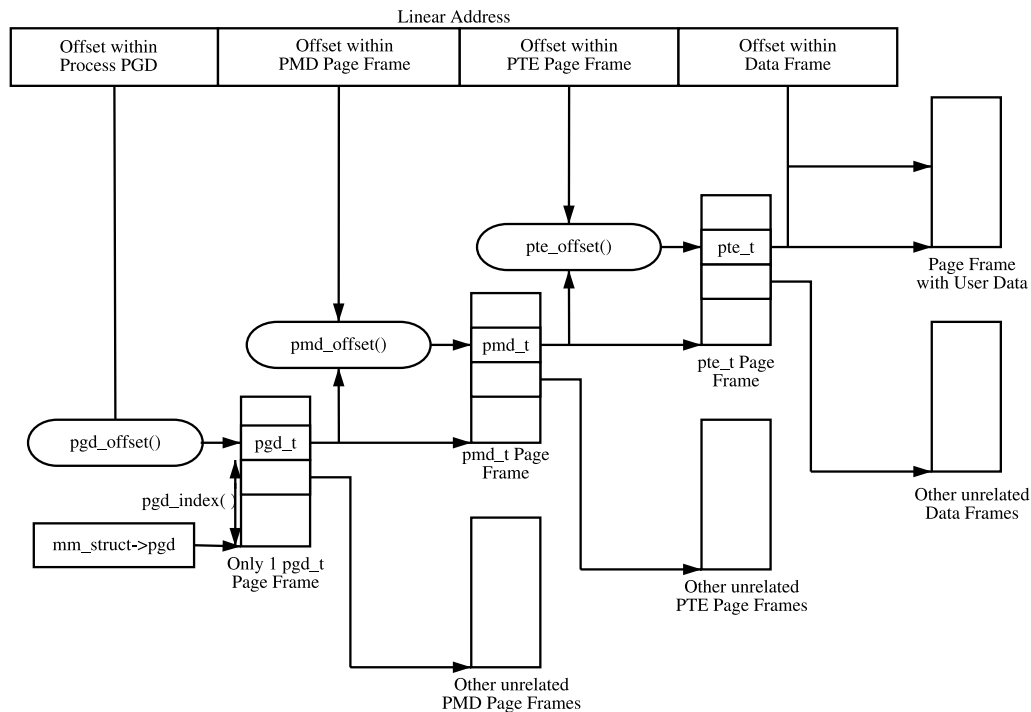


Figure 3.1. Page Table Layout

depending on the architecture. On the x86, the process page table is loaded by copying `mm_struct->pgd` into the `cr3` register, which has the side effect of flushing the TLB. In fact, this is how the function `__flush_tlb()` is implemented in the architecture-dependent code.

Each active entry in the PGD table points to a page frame containing an array of *PMD* entries of type `pmd_t`, which in turn points to page frames containing *PTEs* of type `pte_t`, which finally point to page frames containing the actual user data. In the event that the page has been swapped out to backing storage, the swap entry is stored in the PTE and used by `do_swap_page()` during page fault to find the swap entry containing the page data. The page table layout is illustrated in Figure 3.1.

Any given linear address may be broken up into parts to yield offsets within these three page table levels and an offset within the actual page. To help break up the linear address into its component parts, a number of macros are provided in triplets for each page table level, namely a *SHIFT*, a *SIZE* and a *MASK* macro. The *SHIFT* macros specify the length in bits that are mapped by each level of the page tables as illustrated in Figure 3.2.

The *MASK* values can be ANDed with a linear address to mask out all the upper bits and are frequently used to determine if a linear address is aligned to a given level within the page table. The *SIZE* macros reveal how many bytes are addressed

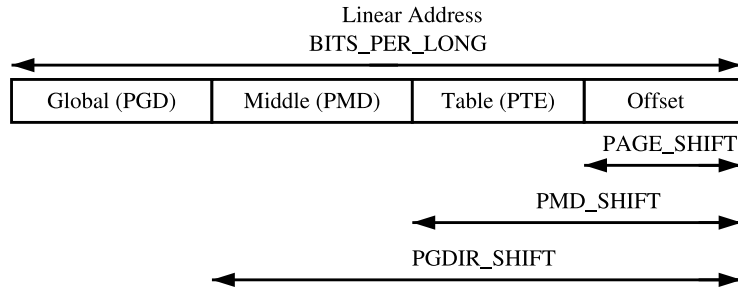


Figure 3.2. Linear Address Bit Size Macros

by each entry at each level. The relationship between the SIZE and MASK macros is illustrated in Figure 3.3.

For the calculation of each of the triplets, only SHIFT is important because the other two are calculated based on it. For example, the three macros for page level on the x86 are:

```

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

```

PAGE_SHIFT is the length in bits of the offset part of the linear address space, which is 12 bits on the x86. The size of a page is easily calculated as $2^{\text{PAGE\_SHIFT}}$ which is the equivalent of the previous code. Finally, the mask is calculated as the negation of the bits that make up the $\text{PAGE\_SIZE} - 1$. If a page needs to be aligned on a page boundary, PAGE_ALIGN() is used. This macro adds $\text{PAGE\_SIZE} - 1$ to the address before simply ANDing it with the PAGE_MASK to zero out the page offset bits.

PMD_SHIFT is the number of bits in the linear address that are mapped by the second-level part of the table. The PMD_SIZE and PMD_MASK are calculated in a similar way to the page-level macros.

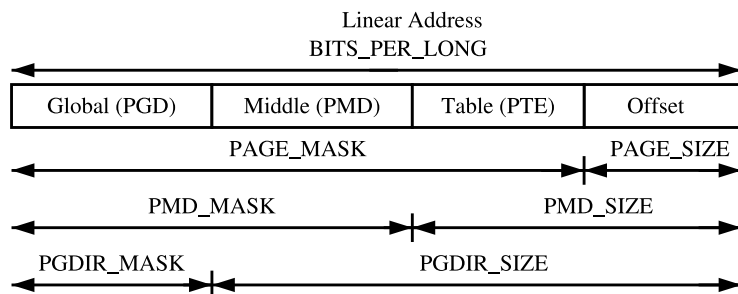


Figure 3.3. Linear Address Size and Mask Macros

PGDIR_SHIFT is the number of bits that are mapped by the top, or ~~the~~ level, of the page table. The PGDIR_SIZE and PGDIR_MASK are calculated in the same manner.

The last three macros of importance are the PTRS_PER_X, which determines the number of entries in each level of the page table. PTRS_PER_PGD is the number of pointers in the PGD, which is 1,024 on an x86 without PAE. PTRS_PER_PMD is for the PMD, which is one on the x86 without PAE, and PTRS_PER_PTE is for the lowest level, which is 1,024 on the x86.

3.2 Describing a Page Table Entry

As mentioned, each entry is described by the structs `pte_t`, `pmd_t` and `pgd_t` for PTEs, PMDs and PGDs respectively. Even though these are often just unsigned integers, they are defined as structs for two reasons. The ~~first~~ is for type protection so that they will not be used inappropriately. The second is for features like PAE on the x86 where an additional 4 bits is used for addressing more than 4GiB of memory. To store the protection bits, `pgprot_t` is defined, which holds the relevant ~~bits~~ and is usually stored in the lower bits of a page table entry.

For type casting, four macros are provided in `asm/page.h`, which takes the previous types and returns the relevant part of the structs. They are `pte_val()`, `pmd_val()`, `pgd_val()` and `pgprot_val()`. To reverse the type casting, four more macros are provided: `_pte()`, `_pmd()`, `_pgd()` and `_pgprot()`.

Where exactly the protection bits are stored is architecture dependent. For illustration purposes, we will examine the case of an x86 architecture without PAE enabled, but the same principles apply across architectures. On an x86 without PAE, the `pte_t` is simply a 32-bit integer within a struct. Each `pte_t` points to an address of a page frame, and all the addresses pointed to are guaranteed to be page aligned. Therefore, there are PAGE_SHIFT (12) bits in that 32-bit value that are free for status bits of the page table entry. A number of the protection and status bits are listed in Table 3.1, but what bits exist and what they mean varies between architectures.

Bit	Function
<code>_PAGE_PRESENT</code>	Page is resident in memory and not swapped out.
<code>_PAGE_PROTNONE</code>	Page is resident, but not accessible.
<code>_PAGE_RW</code>	Set if the page may be written to
<code>_PAGE_USER</code>	Set if the page is accessible from userspace
<code>_PAGE_DIRTY</code>	Set if the page is written to
<code>_PAGE_ACCESSED</code>	Set if the page is accessed

Table 3.1. Page Table Entry Protection and Status Bits

These bits are self-explanatory except for the `_PAGE_PROTNONE`, which I will discuss further. On the x86 with Pentium III and higher, this bit is called the *Page*

Attribute Table (PAT) while earlier architectures such as the Pentium II had this bit reserved. The PAT bit is used to indicate the size of the page that the PTE is referencing. In a PGD entry, this same bit is instead called the *Page Size Extension (PSE)* bit, so obviously these bits are meant to be used in conjunction.

Because Linux does not use the PSE bit for user pages, the PAT bit is free in the PTE for other purposes. There is a requirement for having a page resident in memory, but inaccessible to the user space process, such as when a region is protected with `mprotect()` with the `PROT_NONE` flag. When the region is to be protected, the `_PAGE_PRESENT` bit is cleared, and the `_PAGE_PROTNONE` bit is set. The macro `pte_present()` checks if either of these bits are set, so the kernel itself knows the PTE is present. It is just inaccessible to *userspace*, which is a subtle, but important, point. Because the hardware bit `_PAGE_PRESENT` is clear, a page fault will occur if the page is accessed so that Linux can enforce the protection while still knowing the page is resident if it needs to swap it out or the process exits.

3.3 Using Page Table Entries

Macros are defined in `<asm/pgtable.h>`, which is important for the navigation and examination of page table entries. To navigate the page directories, three macros are provided that break up a linear address space into its component parts. `pgd_offset()` takes an address and the `mm_struct` for the process and returns the PGD entry that covers the requested address. `pmd_offset()` takes a PGD entry and an address and returns the relevant PMD. `pte_offset()` takes a PMD and returns the relevant PTE. The remainder of the linear address provided is the offset within the page. The relationship between these macros is illustrated in Figure 3.1.

The second round of macros determine if the page table entries are present or may be used.

`pte_none()`, `pmd_none()` and `pgd_none()` return 1 if the corresponding entry does not exist.

`pte_present()`, `pmd_present()` and `pgd_present()` return 1 if the corresponding page table entries have the PRESENT bit set.

`pte_clear()`, `pmd_clear()` and `pgd_clear()` will clear the corresponding page table entry.

`pmd_bad()` and `pgd_bad()` are used to check entries when passed as input parameters to functions that may change the value of the entries. Whether they return 1 varies between the few architectures that define these macros. However, for those that actually define it, making sure the page entry is marked as present and accessed are the two most important checks.

Many parts of the VM are littered with page table walk code, and it is important to recognize it. A very simple example of a page table walk is the function `follow_page()` in `mm/memory.c`. The following is an excerpt from that function. The parts unrelated to the page table walk are omitted.

```

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;

```

It simply uses the three offset macros to navigate the page tables and the `_none()` and `_bad()` macros to make sure it is looking at a valid page table.

The third set of macros examine and set the permissions of an entry. The permissions determine what a userspace process can and cannot do with a particular page. For example, the kernel page table entries are never readable by a userspace process.

~~%~~ The read permissions for an entry are tested with `pte_read()`, set with `pte_mkrread()` and cleared with `pte_rdpprotect()`.

~~%~~ The write permissions are tested with `pte_write()`, set with `pte_mkwrite()` and cleared with `pte_wrprotect()`.

~~%~~ The execute permissions are tested with `pte_exec()`, set with `pte_mkexec()` and cleared with `pte_exprotect()`. It is worth noting that, with the x86 architecture, there is no means of setting execute permissions on pages, so these three macros act the same way as the read macros.

~~%~~ The permissions can be modified to a new value with `pte_modify()`, but its use is almost nonexistent. It is only used in the function `change_pte_range()` in `mm/mprotect.c`.

The fourth set of macros examine and set the state of an entry. There are only two bits that are important in Linux, the dirty bit and the accessed bit. To check these bits, the macros `pte_dirty()` and `pte_young()` are used. To set the bits, the macros `pte_mkdirty()` and `pte_mkyoung()` are used. To clear them, the macros `pte_mkclean()` and `pte_old()` are available.

3.4 Translating and Setting Page Table Entries

This set of functions and macros deal with the mapping of addresses and pages to PTEs and the setting of the individual entries.

The macro `mk_pte()` takes a `struct page` and protection bits and combines them together to form the `pte_t` that needs to be inserted into the page table. A similar macro `mk_pte_phys()` exists, which takes a physical page address as a parameter.

The macro `pte_page()` returns the `struct page`, which corresponds to the PTE entry. `pmd_page()` returns the `struct page` containing the set of PTEs.

The macro `set_pte()` takes a `pte_t` such as that returned by `mk_pte()` and places it within the process's page table. `pte_clear()` is the reverse operation. An additional function is provided called `ptep_get_and_clear()`, which clears an entry from the process page table and returns the `pte_t`. This is important when some modification needs to be made to either the PTE protection or the `struct page` itself.

3.5 Allocating and Freeing Page Tables

The last set of functions deal with the allocation and freeing of page tables. Page tables, as stated, are physical pages containing an array of entries, and the allocation and freeing of physical pages is a relatively expensive operation, both in terms of time and the fact that interrupts are disabled during page allocation. The allocation and deletion of page tables, at any of the three levels, is a very frequent operation, so it is important the operation is as quick as possible.

Hence the pages used for the page tables are cached in a number of different lists called *quicklists*. Each architecture implements these caches differently, but the principles used are the same. For example, not all architectures cache PGDs because the allocation and freeing of them only happens during process creation and exit. Because both of these are very expensive operations, the allocation of another page is negligible.

PGDs, PMDs and PTEs have two sets of functions each for the allocation and freeing of page tables. The allocation functions are `pgd_alloc()`, `pmd_alloc()` and `pte_alloc()`, respectively, and the free functions are, predictably enough, called `pgd_free()`, `pmd_free()` and `pte_free()`.

Broadly speaking, the three implement caching with the use of three caches called `pgd_quicklist`, `pmd_quicklist` and `pte_quicklist`. Architectures implement these three lists in different ways, but one method is through the use of a Last In, First Out (LIFO) type structure. Ordinarily, a page table entry contains pointers to other pages containing page tables or data. While cached, the `next` element of the list is used to point to the next free page table. During allocation, one page is popped off the list, and, during free, one is placed as the new head of the list. A count is kept of how many pages are used in the cache.

The quick allocation function from the `pgd_quicklist` is not externally defined outside of the architecture, although `get_pgd_fast()` is a common choice for the

function name. The cached allocation function for PMDs and PTEs are publicly defined as `pmd_alloc_one_fast()` and `pte_alloc_one_fast()`.

If a page is not available from the cache, a page will be allocated using the physical page allocator (see Chapter 6). The functions for the three levels of page tables are `get_pgd_slow()`, `pmd_alloc_one()` and `pte_alloc_one()`.

Obviously, a large number of pages may exist on these caches, so a mechanism is in place for pruning them. Each time the caches grow or shrink, a counter is incremented or decremented, and it has a high and low watermark. `check_pgt_cache()` is called in two places to check these watermarks. When the high watermark is reached, entries from the cache will be freed until the cache size returns to the low watermark. The function is called after `clear_page_tables()` when a large number of page tables are potentially reached and is also called by the system idle task.

3.6 Kernel Page Tables

When the system ~~is~~ starts, paging is not enabled because page tables do not magically initialize themselves. Each architecture implements this differently so only the x86 case will be discussed. The page table initialization is divided into two phases. The bootstrap phase sets up page tables for just 8MiB so that the paging unit can be enabled. The second phase initializes the rest of the page tables. We discuss both of these phases in the following sections.

3.6.1 Bootstrapping

The assembler function `startup_32()` is responsible for enabling the paging unit in `arch/i386/kernel/head.S`. While all normal kernel code in `vmLinux` is compiled with the base address at `PAGE_OFFSET + 1MiB`, the kernel is actually loaded beginning at the ~~1st~~ megabyte (`0x00100000`) of memory. The ~~1st~~ megabyte is used by some devices for communication with the BIOS and is skipped. The bootstrap code in this ~~file~~ treats 1MiB as its base address by subtracting `_PAGE_OFFSET` from any address until the paging unit is enabled. Therefore before the paging unit is enabled, a page table mapping has to be established that translates the 8MiB of physical memory to the virtual address `PAGE_OFFSET`.

Initialization begins at compile time with statically defining an array called `swapper_pg_dir`, which is placed using linker directives at `0x00101000`. It then establishes page table entries for two pages, `pg0` and `pg1`. If the processor supports the *Page Size Extension (PSE)* bit, it will be set so that pages that will be translated are 4MiB pages, not 4KiB as is the normal case. The ~~1st~~ pointers to `pg0` and `pg1` are placed to cover the region 1-9MiB; the second pointers to `pg0` and `pg1` are placed at `PAGE_OFFSET+1MiB`. This means that, when paging is enabled, they will map to the correct pages using either physical or virtual addressing for just the kernel image. The rest of the kernel page tables will be initialized by `paging_init()`.

After this mapping has been established, the paging unit is turned on by setting a bit in the `cr0` register, and a jump takes places immediately to ensure the *Instruction Pointer (EIP register)* is correct.

3.7 Mapping Addresses to a struct page

There is a requirement for Linux to have a fast method of mapping virtual addresses to physical addresses and for mapping struct pages to their physical address. Linux achieves this by knowing where, in both virtual and physical memory, the global `mem_map` array is because the global array has pointers to all struct pages representing physical memory in the system. All architectures achieve this with very similar mechanisms, but, for illustration purposes, we will only examine the x86 carefully. This section will ~~not~~ discuss how physical addresses are mapped to kernel virtual addresses and then what this means to the `mem_map` array.

3.7.1 Mapping Physical to Virtual Kernel Addresses

As we saw in Section 3.6, Linux sets up a direct mapping from the physical address 0 to the virtual address `PAGE_OFFSET` at 3GiB on the x86. This means that any virtual address can be translated to the physical address by simply subtracting `PAGE_OFFSET`, which is essentially what the function `virt_to_phys()` with the macro `__pa()` does:

```
/* 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 }
```

Obviously, the reverse operation involves simply adding `PAGE_OFFSET`, which is carried out by the function `phys_to_virt()` with the macro `__va()`. Next we see how this helps the mapping of struct pages to physical addresses.

There is one exception where `virt_to_phys()` cannot be used to convert virtual addresses to physical ones.¹ Specifically, on the PPC and ARM architectures, `virt_to_phys()` cannot be used to convert addresses that have been returned by the function `consistent_alloc()`. `consistent_alloc()` is used on PPC and ARM architectures to return memory from non-cached for use with DMA.

3.7.2 Mapping struct pages to Physical Addresses

As we saw in Section 3.6.1, the kernel image is located at the physical address 1MiB, which of course translates to the virtual address `PAGE_OFFSET + 0x00100000`, and a virtual region totaling about 8MiB is reserved for the image, which is the region that can be addressed by two PGDs. This would imply that the ~~the~~ available memory to use is located at `0xC0800000`, but that is not the case. Linux tries to reserve the ~~the~~ 16MiB of memory for `ZONE_DMA`, so the ~~the~~ virtual area used

¹This tricky issue was pointed out to me by Je rey Haran.

for kernel allocations is actually `0xC1000000`. This is where the global `mem_map` is usually located. `ZONE_DMA` will still get used, but only when absolutely necessary.

Physical addresses are translated to `struct pages` by treating them as an index into the `mem_map` array. Shifting physical address `PAGE_SHIFT` bits to the right will treat them as a *Page Frame Number (PFN)* from physical address 0, which is *also* an index within the `mem_map` array. This is exactly what the macro `virt_to_page()` does, which is declared as follows in `<asm-i386/page.h>`:

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

The macro `virt_to_page()` takes the virtual address `kaddr`, converts it to the physical address with `_pa()`, converts it into an array index by bit shifting `PAGE_SHIFT` bits right and indexing into the `mem_map` by simply adding them together. No macro is available for converting `struct pages` to physical addresses, but, at this stage, you should see how it could be calculated.

3.8 Translation Lookaside Buffer (TLB)

Initially, when the processor needs to map a virtual address to a physical address, it must traverse the full page directory searching for the PTE of interest. This would normally imply that each assembly instruction that references memory actually requires several separate memory references for the page table traversal [Tan01]. To avoid this considerable overhead, architectures take advantage of the fact that most processes exhibit a locality of reference, or, in other words, large numbers of memory references tend to be for a small number of pages. They take advantage of this reference locality by providing a *Translation Lookaside Buffer (TLB)*, which is a small associative memory that caches virtual to physical page table resolutions.

Linux assumes that most architectures support some type of TLB, although the architecture-independent code does not care how it works. Instead, architecture-dependent hooks are dispersed throughout the VM code at points where it is known that some hardware with a TLB would need to perform a TLB-related operation. For example, when the page tables have been updated, such as after a page fault has completed, the processor may need to update the TLB for that virtual address mapping.

Not all architectures require these type of operations, but, because some do, the hooks have to exist. If the architecture does not require the operation to be performed, the function for that TLB operation will be a null operation that is optimized out at compile time.

A quite large list of TLB API hooks, most of which are declared in `<asm/pgtable.h>`, are listed in Tables 3.2 and 3.3, and the APIs are quite well documented in the kernel source by `Documentation/cachetlb.txt` [Mil00]. It is possible to have just one TLB flush function, but, because both TLB flushes and TLB reloads are *very* expensive operations, unnecessary TLB flushes should be avoided if at all possible. For example, when context switching, Linux will avoid loading new page tables using *Lazy TLB Flushing*, discussed further in Section 4.3.

```
void flush_tlb_all(void)
```

This flushes the entire TLB on all processors running in the system, which makes it the most expensive TLB flush operation. After it completes, all modifications to the page tables will be visible globally. This is required after the kernel page tables, which are global in nature, have been modified, such as after `vfree()` (see Chapter 7) completes or after the PKMap is flushed (see Chapter 9).

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

This flushes all TLB entries related to the userspace portion (i.e., below `PAGE_OFFSET`) for the requested mm context. In some architectures, such as MIPS, this will need to be performed for all processors, but usually it is confined to the local processor. This is only called when an operation has been performed that affects the entire address space, such as after all the address mapping has been duplicated with `dup_mmap()` for fork or after all memory mappings have been deleted with `exit_mmap()`.

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

As the name indicates, this flushes all entries within the requested user space range for the mm context. This is used after a new region has been moved or changed as during `mremap()`, which moves regions, or `mprotect()`, which changes the permissions. The function is also indirectly used during unmapping a region with `munmap()`, which calls `tlb_finish_mmu()`, which tries to use `flush_tlb_range()` intelligently. This API is provided for architectures that can remove ranges of TLB entries quickly rather than iterating with `flush_tlb_page()`.

Table 3.2. Translation Lookaside Buffer Flush API

3.9 Level 1 CPU Cache Management

Because Linux manages the CPU cache in a very similar fashion to the TLB, this section covers how Linux uses and manages the CPU cache. CPU caches, like TLB caches, take advantage of the fact that programs tend to exhibit a locality of reference [Sea00] [CS98]. To avoid having to fetch data from main memory for each reference, the CPU will instead cache very small amounts of data in the CPU cache. Frequently, there are two levels called the Level 1 and Level 2 CPU caches. The Level 2 CPU caches are larger, but slower than the L1 cache, but Linux only concerns itself with the Level 1 or L1 cache.

CPU caches are organized into *lines*. Each line is typically quite small, usually 32 bytes, and each line is aligned to its boundary size. In other words, a cache line of 32 bytes will be aligned on a 32-byte address. With Linux, the size of the line is `L1_CACHE_BYTES`, which is defined by each architecture.

How addresses are mapped to cache lines vary between architectures, but the mappings come under three headings, *direct mapping*, *associative mapping* and *set*

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

Predictably, this API is responsible for flushing a single page from the TLB. The two most common uses of it are for flushing the TLB after a page has been faulted in or has been paged out.

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

This API is called when the page tables are being torn down and freed. Some platforms cache the lowest level of the page table, i.e., the actual page frame storing entries, which needs to be flushed when the pages are being deleted. This is called when a region is being unmapped and the page directory entries are being reclaimed.

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

This API is only called after a page fault completes. It tells the architecture-dependent code that a new translation now exists at pte for the virtual address addr. Each architecture decides how this information should be used. For example, Sparc64 uses the information to decide if the local CPU needs to flush its data cache or does it need to send an Inter Processor Interrupt (IPI) to a remote processor.

Table 3.3. Translation Lookaside Buffer Flush API (cont.)

associative mapping. Direct mapping is the simplest approach where each block of memory maps to only one possible cache line. With associative mapping, any block of memory can map to any cache line. Set associative mapping is a hybrid approach where any block of memory can map to any line, but only within a subset of the available lines. Regardless of the mapping scheme, they each have one thing in common. Addresses that are close together and aligned to the cache size are likely to use different lines. Hence Linux employs simple tricks to try and maximize cache use:

- Frequently accessed structure fields are at the start of the structure to increase the chance that only one line is needed to address the common fields.
- Unrelated items in a structure should try to be at least cache-size bytes in part to avoid false sharing between CPUs.
- Objects in the general caches, such as the mm_struct cache, are aligned to the L1 CPU cache to avoid false sharing.

If the CPU references an address that is not in the cache, a *cache miss* occurs, and the data is fetched from main memory. The cost of cache misses is quite high because a reference to a cache can typically be performed in less than 10ns where a reference to main memory typically will cost between 100ns and 200ns. The basic objective is then to have as many cache hits and as few cache misses as possible.

Just as some architectures do not automatically manage their TLBs, some do not automatically manage their CPU caches. The hooks are placed in locations where the virtual to physical mapping changes, such as during a page table update. The CPU cache flushes should always take place ~~not~~ because some CPUs require a virtual to physical mapping to exist when the virtual address is being ~~flushed~~ from the cache. The three operations that require proper ordering are important and are listed in Table 3.4.

Flushing Full MM	Flushing Range	Flushing Page
flush_cache_mm() Change all page tables flush_tlb_mm()	flush_cache_range() Change page table range flush_tlb_range()	flush_cache_page() Change single PTE flush_tlb_page()

Table 3.4. Cache and TLB Flush Ordering

The API used for ~~flushing~~ the caches is declared in `<asm/pgtable.h>` and is listed in Table 3.5. In many respects, it is very similar to the TLB ~~flushing~~ API.

<pre>void flush_cache_all(void)</pre> This ~~flushes~~ the entire CPU cache system, which makes it the most severe ~~flush~~ operation to use. It is used when changes to the kernel page tables, which are global in nature, are to be performed. <pre>void flush_cache_mm(struct mm_struct mm)</pre> This ~~flushes~~ all entries related to the address space. On completion, no cache lines will be associated with mm. <pre>void flush_cache_range(struct mm_struct *mm, unsigned long start, unsigned long end)</pre> This ~~flushes~~ lines related to a range of addresses in the address space. Like its TLB equivalent, it is provided in case the architecture has an efficient way of ~~flushing~~ ranges instead of ~~flushing~~ each individual page. <pre>void flush_cache_page(struct vm_area_struct *vma, unsigned long vmaddr)</pre> This is for ~~flushing~~ a single-page-sized region. The VMA is supplied because the mm_struct is easily accessible through <code>vma->vm_mm</code>. Additionally, by testing for the VM_EXEC ~~flag~~, the architecture will know if the region is executable for caches that separate the instructions and data caches. VMAs are described further in Chapter 4.

Table 3.5. CPU Cache Flush API

It does not end there, though. A second set of interfaces is required to avoid virtual aliasing problems. The problem is that some CPUs select lines based on the virtual address, which means that one physical address can exist on multiple

lines leading to cache coherency problems. Architectures with this problem may try and ensure that shared mappings will only use addresses as a stop-gap measure. However, a proper API to address this problem is also supplied, which is listed in Table 3.6.

<pre>void flush_page_to_ram(unsigned long address)</pre> This is a deprecated API that should no longer be used and, in fact, will be removed totally for 2.6. It is covered here for completeness and because it is still used. The function is called when a new physical page is about to be placed in the address space of a process. It is required to avoid writes from kernel space being invisible to userspace after the mapping occurs.
<pre>void flush_dcache_page(struct page *page)</pre> This function is called when the kernel writes to or copies from a page cache page because these are likely to be mapped by multiple processes.
<pre>void flush_icache_range(unsigned long address, unsigned long endaddr)</pre> This is called when the kernel stores information in addresses that is likely to be executed, such as when a kernel module has been loaded.
<pre>void flush_icache_user_range(struct vm_area_struct *vma, struct page *page, unsigned long addr, int len)</pre> This is similar to <code>flush_icache_range()</code> except it is called when a userspace range is affected. Currently, this is only used for <code>ptrace()</code> (used when debugging) when the address space is being accessed by <code>access_process_vm()</code>.
<pre>void flush_icache_page(struct vm_area_struct *vma, struct page *page)</pre> This is called when a page-cache page is about to be mapped. It is up to the architecture to use the VMA flags to determine whether the I-Cache or D-Cache should be flushed.

Table 3.6. CPU D-Cache and I-Cache Flush API

3.10 What's New in 2.6

Most of the mechanics for page table management are essentially the same for 2.6, but the changes that have been introduced are quite wide reaching and the implementations are in depth.

MMU-less Architecture Support A new file has been introduced called `mm/nommu.c`. This source file contains replacement code for functions that assume the existence of a MMU, like `mmap()` for example. This is to support architectures, usually microcontrollers, that have no MMU. Much of the work in this area was developed by the uClinux Project (www.uclinux.org).

Reverse Mapping The most significant and important change to page table management is the introduction of *Reverse Mapping (rmap)*. Referring to it as *rmap* is deliberate because it is the common use of the acronym and should not be confused with the *-rmap* tree developed by Rik van Riel, which has many more alterations to the stock VM than just the reverse mapping.

In a single sentence, *rmap* grants the ability to locate all PTEs that map a particular page given just the `struct page`. In 2.4, the only way to find all PTEs that mapped a shared page, such as a memory mapped shared library, is to linearly search all page tables belonging to all processes. This is far too expensive, and Linux tries to avoid the problem by using the swap cache (see Section 11.4). This means that, with many shared pages, Linux may have to swap out entire processes regardless of the page age and usage patterns. 2.6 instead has a *PTE chain* associated with every `struct page`, which may be traversed to remove a page from all page tables that reference it. This way, pages in the LRU can be swapped out in an intelligent manner without resorting to swapping entire processes.

As might be imagined by the reader, the implementation of this simple concept is a little involved. The first step in understanding the implementation is the union `pte` that is a field in `struct page`. This union has two fields, a pointer to a `struct pte_chain` called `chain` and a `pte_addr_t` called `direct`. The union is an optimization whereby `direct` is used to save memory if there is only one PTE mapping the entry. Otherwise, a chain is used. The type `pte_addr_t` varies between architectures, but, whatever its type, it can be used to locate a PTE, so we will treat it as a `pte_t` for simplicity.

The `struct pte_chain` is a little more complex. The struct itself is very simple, but it is *compact* with overloaded fields, and a lot of development effort has been spent on making it small and efficient. Fortunately, this does not make it indecipherable.

First, it is the responsibility of the slab allocator to allocate and manage `struct pte_chains` because it is this type of task that the slab allocator is best at. Each `struct pte_chain` can hold up to `NRPTE` pointers to PTE structures. After that many PTEs have been added, a `struct pte_chain` is allocated and added to the chain.

The `struct pte_chain` has two fields. The first is `unsigned long next_and_idx`, which has two purposes. When `next_and_idx` is ANDed with `NRPTE`, it returns the number of PTEs currently in this `struct pte_chain` and indicates where the next free slot is. When `next_and_idx` is ANDed with the negation of `NRPTE` (i.e., `~NRPTE`), a pointer to the next `struct pte_chain` in the chain is returned². This is basically how a PTE chain is implemented.

To give you a taste of the *rmap* intricacies, I'll give an example of what happens when a new PTE needs to map a page. The basic process is to have the caller allocate a new `pte_chain` with `pte_chain_alloc()`. This allocated chain is passed with the `struct page` and the PTE to `page_add_rmap()`. If the existing PTE chain associated with the page has slots available, it will be used, and the `pte_chain`

²I told you it was compact.

allocated by the caller is returned. If no slots were available, the allocated `pte_chain` will be added to the chain, and `NULL` returned.

There is a quite substantial API associated with `rmap` for tasks such as creating chains and adding and removing PTEs to a chain, but a full listing is beyond the scope of this section. Fortunately, the API is confined to `mm/rmap.c`, and the functions are heavily commented so that their purpose is clear.

There are two main benefits, both related to pageout, with the introduction of reverse mapping. The first is with the set up and tear down of page tables. As will be seen in Section 11.4, pages being paged out are placed in a swap cache, and information is written into the PTE that is necessary to find the page again. This can lead to multiple minor faults because pages are put into the swap cache and then faulted again by a process. With `rmap`, the setup and removal of PTEs is atomic. The second major benefit is when pages need to be paged out, finding all PTEs referencing the pages is a simple operation, but impractical with 2.4, hence the swap cache.

Reverse mapping is not without its cost, though. The first, and obvious one, is the additional space requirements for the PTE chains. Arguably, the second is a CPU cost associated with reverse mapping, but it has not been proved to be significant. What is important to note, though, is that reverse mapping is only a benefit when pageouts are frequent. If the machine's workload does not result in much pageout or memory is ample, reverse mapping is all cost with little or no benefit. At the time of writing, the merits and downsides to `rmap` are still the subject of a number of discussions.

Object-Based Reverse Mapping The reverse mapping required for each page can have very expensive space requirements. To compound the problem, many of the reverse mapped pages in a VMA will be essentially identical. One way of addressing this is to reverse map based on the VMAs rather than individual pages. That is, instead of having a reverse mapping for each page, all the VMAs that map a particular page would be traversed and unmap the page from each. Note that objects in this case refer to the VMAs, not an object in the object-orientated sense of the word³. At the time of writing, this feature has not been merged yet and was last seen in kernel 2.5.68-mm1, but a strong incentive exists to have it available if the problems with it can be resolved. For the very curious, the patch for just `/dev` backed `objrmap` at this release is available⁴, but it is only for the very very curious reader.

Two tasks require all PTEs that map a page to be traversed. The first task is `page_referenced()`, which checks all PTEs that map a page to see if the page has been referenced recently. The second task is when a page needs to be unmapped from all processes with `try_to_unmap()`. To complicate matters further, two types of mappings must be reverse mapped, those that are backed by a `/dev` or device and those that are anonymous. In both cases, the basic objective is to traverse all

³Don't blame me, I didn't name it. In fact, the original patch for this feature came with the comment from Dave. Crappy name.

⁴<http://ftp.kernel.org/pub/linux/kernel/people/akpm/patches/2.5/2.5.68/2.5.68-mm2/experimental>

VMAs that map a particular page and then walk the page table for that VMA to get the PTE. The only difference is how it is implemented. The case where it is backed by some sort of `Page` is the easiest case and was implemented `Page` so `Page` deal with it `Page`. For the purposes of illustrating the implementation, `Page` discuss how `page_referenced()` is implemented.

`page_referenced()` calls `page_referenced_obj()`, which is the top-level function for finding all PTEs within VMAs that map the page. As the page is mapped for a `Page` or device, `page->mapping` contains a pointer to a valid `address_space`. The `address_space` has two linked lists that contain all VMAs that use the mapping with the `address_space->i_mmap` and `address_space->i_mmap_shared` `Lists`. For every VMA that is on these linked lists, `page_referenced_obj_one()` is called with the VMA and the page as parameters. The function `page_referenced_obj_one()` `Page` checks if the page is in an address managed by this VMA and, if so, traverses the page tables of the `mm_struct` using the VMA (`vma->vm_mm`) until it `Page` the PTE mapping the page for that `mm_struct`.

Anonymous page tracking is a lot trickier and was implemented in a number of stages. It only made a very brief appearance and was removed again in 2.5.65-mm4 because it conflicted with a number of other changes. The `Page` stage in the implementation was to use `page->mapping` and `page->index` `Lists` to track `mm_struct` and address pairs. These `Lists` previously had been used to store a pointer to `swapper_space` and a pointer to the `swp_entry_t` (See Chapter 11). Exactly how it is addressed is beyond the scope of this section, but the summary is that `swp_entry_t` is stored in `page->private`.

`try_to_unmap_obj()` works in a similar fashion, but, obviously, all the PTEs that reference a page with this method can do so without needing to reverse map the individual pages. A serious search complexity problem prevents it from being merged. The scenario that describes the problem is as follows.

Take a case where 100 processes have 100 VMAs mapping a single `Page`. To unmap a *single* page in this case with object-based reverse mapping would require 10,000 VMAs to be searched, most of which are totally unnecessary. With page-based reverse mapping, only 100 `pte_chain` slots need to be examined, one for each process. An optimization was introduced to order VMAs in the `address_space` by virtual address, but the search for a single page is still far too expensive for object-based reverse mapping to be merged.

PTEs in High Memory In 2.4, page table entries exist in `ZONE_NORMAL` because the kernel needs to be able to address them directly during a page table walk. This was acceptable until it was found that, with high memory machines, `ZONE_NORMAL` was being consumed by the third-level page table PTEs. The obvious answer is to move PTEs to high memory, which is exactly what 2.6 does.

As we will see in Chapter 9, addressing information in high memory is far from free, so moving PTEs to high memory is a compile-time configuration option. In short, the problem is that the kernel must map pages from high memory into the lower address space before it can be used but a very limited number of slots are available for these mappings, which introduces a troublesome bottleneck. However, for applications with a large number of PTEs, there is little other option. At the

time of writing, a proposal has been made for having a User Kernel Virtual Area (UKVA), which would be a region in kernel space private to each process, but it is unclear if it will be merged for 2.6 or not.

To take the possibility of high memory mapping into account, the macro `pte_offset()` from 2.4 has been replaced with `pte_offset_map()` in 2.6. If PTEs are in low memory, this will behave the same as `pte_offset()` and return the address of the PTE. If the PTE is in high memory, it will ~~not~~ be mapped into low memory with `kmap_atomic()`, so it can be used by the kernel. This PTE must be unmapped as quickly as possible with `pte_unmap()`.

In programming terms, this means that page table walk code looks slightly different. In particular, to ~~find~~ the PTE for a given address, the code now reads as (taken from `mm/memory.c`):

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

Additionally, the PTE allocation API has changed. Instead of `pte_alloc()`, there is now a `pte_alloc_kernel()` for use with kernel PTE mappings and `pte_alloc_map()` for userspace mapping. The principal difference between them is that `pte_alloc_kernel()` will never use high memory for the PTE.

In memory management terms, the overhead of having to map the PTE from high memory should not be ignored. Only one PTE at a time may be mapped per CPU, although a second may be mapped with `pte_offset_map_nested()`. This introduces a penalty when all PTEs need to be examined, such as during `zap_page_range()` when all PTEs in a given range need to be unmapped.

At the time of writing, a patch has been submitted that places PMDs in high memory using essentially the same mechanism and API changes. It is likely that it will be merged.

Huge TLB Filesystem Most modern architectures support more than one page size. For example, on many x86 architectures, there is an option to use 4KiB pages or 4MiB pages. Traditionally, Linux only used large pages for mapping the actual kernel image and nowhere else. Because TLB slots are a scarce resource, it is desirable to be able to take advantage of the large pages, especially on machines with large amounts of physical memory.

In 2.6, Linux allows processes to use huge pages, the size of which is determined by `HugeTLB_SIZE`. The number of available huge pages is determined by the system administrator by using the `/proc/sys/vm/nr_hugepages` proc interface, which ultimately uses the function `set_hugetlb_mem_size()`. Because the success of the allocation depends on the availability of physically contiguous memory, the allocation should be made during system startup.

The root of the implementation is a *Huge TLB Filesystem* (*hugetlbfs*), which is a pseudofs implemented in `fs/hugetlbfs/inode.c`. Basically,

each `%o` in this `%s` system is backed by a huge page. During initialization, `init_hugetlbfs_fs()` registers the `%s` system and mounts it as an internal `%s` system with `kern_mount()`.

There are two ways that huge pages may be accessed by a process. The `%s` is by using `shmget()` to set up a shared region backed by huge pages, and the second is the call `mmap()` on a `%o` opened in the huge page `%s` system.

When a shared memory region should be backed by huge pages, the process should call `shmget()` and pass `SHM_HUGETLB` as one of the `%s`s. This results in `hugetlb_zero_setup()` being called, which creates a new `%o` in the root of the internal `hugetlbfs`. A `%o` is created in the root of the internal `%s` system. The name of the `%o` is determined by an atomic counter called `hugetlbfs_counter`, which is incremented every time a shared region is set up.

To create a `%o` backed by huge pages, a `%s` system of type `hugetlbfs` must `%s` be mounted by the system administrator. Instructions on how to perform this task are detailed in `Documentation/vm/hugetlbpage.txt`. After the `%s` system is mounted, `%s` can be created as normal with the system call `open()`. When `mmap()` is called on the open `%o`, the `file_operations` struct `hugetlbfs_file_operations` ensures that `hugetlbfs_file_mmap()` is called to set up the region properly.

Huge TLB pages have their own function for the management of page tables, address space operations and `%s` system operations. The names of the functions for page table management can all be seen in `<linux/hugetlb.h>`, and they are named very similar to their normal page equivalents. The implementation of the `hugetlbfs` functions are located near their normal page equivalents, so are easy to `%d`.

Cache Flush Management The changes here are minimal. The API function `flush_page_to_ram()` has been totally removed, and a new API `flush_dcache_range()` has been introduced.

Process Address Space

One of the principal advantages of virtual memory is that each process has its own virtual address space, which is mapped to physical memory by the operating system. In this chapter I discuss the process address space and how Linux manages it.

The kernel treats the userspace portion of the address space very differently from the kernel portion. For example, allocations for the kernel are satisfied immediately and are visible globally no matter what process is on the CPU. `vmalloc()` is an exception because a minor page fault will occur to sync the process page tables with the reference page tables, but the page will still be allocated immediately upon request. With a process, space is simply reserved in the linear address space by pointing a page table entry to a read-only globally visible page filled with zeros. On writing, a page fault is triggered, which results in a new page being allocated, filled with zeros, placed in the page table entry and marked writable. The new page is filled with zeros so that it will appear exactly the same as the global zero-filled page.

The userspace portion is not trusted or presumed to be constant. After each context switch, the userspace portion of the linear address space can potentially change except when a *Lazy TLB* switch is used as discussed later in Section 4.3. As a result of this, the kernel must be prepared to catch all exceptions and to address errors raised from the userspace. This is discussed in Section 4.5.

This chapter begins with how the linear address space is broken up and what the purpose of each section is. I then cover the structures maintained to describe each process, how they are allocated, initialized and then destroyed. Next, I cover how individual regions within the process space are created and all the various functions associated with them. That will bring us to exception handling related to the process address space, page faulting and the various cases that occur to satisfy a page fault. Finally, I cover how the kernel safely copies information to and from userspace.

4.1 Linear Address Space

From a user perspective, the address space is a *flat* linear address space, but, predictably, the kernel's perspective is very different. The address space is split into two parts: the userspace part, which potentially changes with each full context switch, and the kernel address space, which remains constant. The location of the

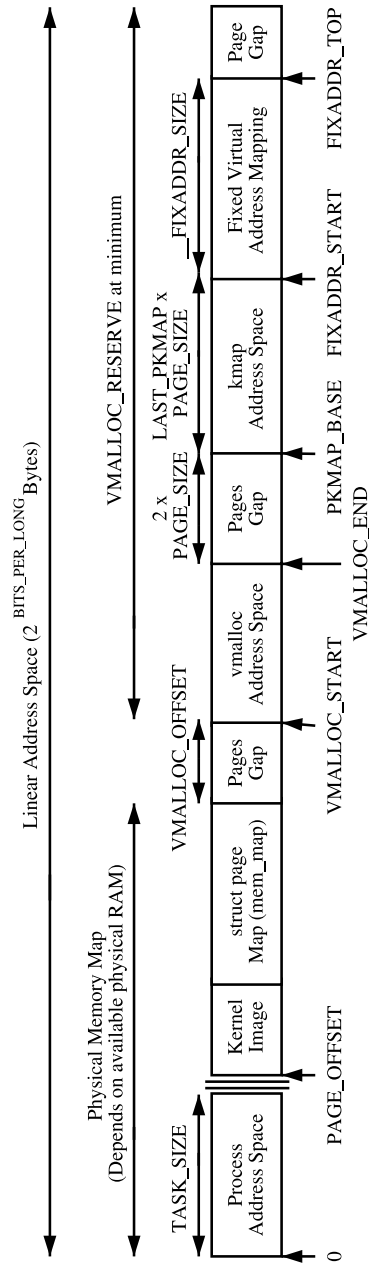


Figure 4.1. Kernel Address Space

split is determined by the value of `PAGE_OFFSET`, which is at `0xC0000000` on the x86. This means that 3GiB is available for the process to use while the remaining 1GiB is always mapped by the kernel. The linear virtual address space as the kernel sees it is illustrated in Figure 4.1.

To load the kernel image to run, 8MiB (the amount of memory addressed by two PGDs) is reserved at `PAGE_OFFSET`. The 8MiB is simply a reasonable amount of space to reserve for the purposes of loading the kernel image. The kernel image is placed in this reserved space during kernel page table initialization as discussed in Section 3.6.1. Somewhere shortly after the image, the `mem_map` for UMA architectures, as discussed in Chapter 2, is stored. The location of the array is usually at the 16MiB mark to avoid using `ZONE_DMA`, but not always. With NUMA architectures, portions of the virtual `mem_map` will be scattered throughout this region. Where they are actually located is architecture dependent.

The region between `PAGE_OFFSET` and `VMALLOC_START - VMALLOC_OFFSET` is the physical memory map, and the size of the region depends on the amount of available RAM. As we saw in Section 3.6, page table entries exist to map physical memory to the virtual address range beginning at `PAGE_OFFSET`. Between the physical memory map and the `vmalloc` address space, there is a gap of space `VMALLOC_OFFSET` in size, which on the x86 is 8MiB, to guard against out-of-bounds errors. For illustration, on a x86 with 32MiB of RAM, `VMALLOC_START` will be located at `PAGE_OFFSET + 0x02000000 + 0x00800000`.

In low memory systems, the remaining amount of the virtual address space, minus a 2 page gap, is used by `vmalloc()` for representing noncontiguous memory allocations in a contiguous virtual address space. In high-memory systems, the `vmalloc` area extends as far as `PKMAP_BASE` minus the two-page gap, and two extra regions are introduced. The first, which begins at `PKMAP_BASE`, is an area reserved for the mapping of high memory pages into low memory with `kmap()` as discussed in Chapter 9. The second is for fixed virtual address mappings that extend from `FIXADDR_START` to `FIXADDR_TOP`. Fixed virtual addresses are needed for subsystems that need to know the virtual address at compile time such as the *APIC*. `FIXADDR_TOP` is statically defined to be `0xFFFFE000` on the x86 which is one page before the end of the virtual address space. The size of the fixed mapping region is calculated at compile time in `__FIXADDR_SIZE` and used to index back from `FIXADDR_TOP` to give the start of the region `FIXADDR_START`.

The region required for `vmalloc()`, `kmap()` and the fixed virtual address mapping is what limits the size of `ZONE_NORMAL`. As the running kernel needs these functions, a region of at least `VMALLOC_RESERVE` will be reserved at the top of the address space. `VMALLOC_RESERVE` is architecture specific but on the x86, it is defined as 128MiB. This is why `ZONE_NORMAL` is generally referred to being only 896MiB in size; it is the 1GiB of the upper portion of the linear address space minus the minimum 128MiB that is reserved for the `vmalloc` region.

4.2 Managing the Address Space

The address space usable by the process is managed by a high level `mm_struct` which is roughly analogous to the `vm_space` struct in BSD [McK96].

Each address space consists of a number of page-aligned regions of memory that are in use. They never overlap and represent a set of addresses which contain pages that are related to each other in terms of protection and purpose. These regions are represented by a `struct vm_area_struct` and are roughly analogous to the `vm_map_entry` struct in BSD. For clarity, a region may represent the process heap for use with `malloc()`, a memory mapped file such as a shared library or a block of anonymous memory allocated with `mmap()`. The pages for this region may still have to be allocated, be active and resident or have been paged out.

If a region is backed by a file, its `vm_file` field will be set. By traversing `vm_file` → `f_dentry` → `d_inode` → `i_mapping`, the associated address space for the region may be obtained. The address space has all the system specific information required to perform page-based operations on disk.

The relationship between the different address space related structures is illustrated in Figure 4.2. A number of system calls are provided which affect the address space and regions. These are listed in Table 4.1.

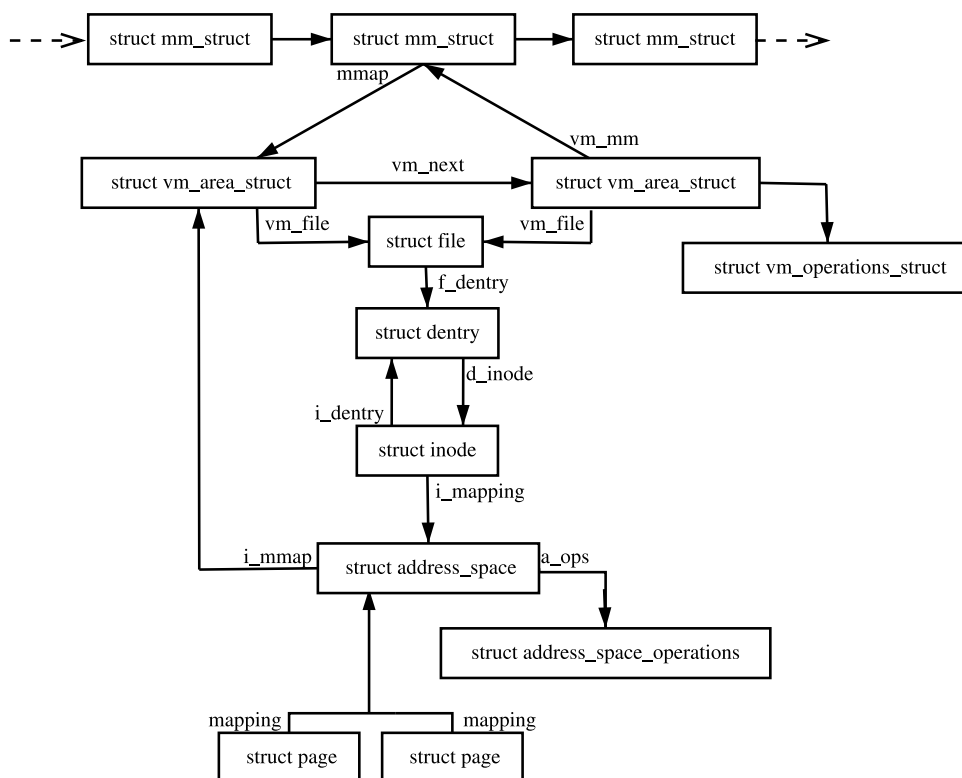


Figure 4.2. Data Structures Related to the Address Space

System Call	Description
<code>fork()</code>	Creates a new process with a new address space. All the pages are marked Copy-On-Write (COW) and are shared between the two processes until a page fault occurs. Once a write-fault occurs, a copy is made of the COW page for the faulting process. This is sometimes referred to as breaking a COW page.
<code>clone()</code>	<code>clone()</code> allows a new process to be created that shares parts of its context with its parent and is how threading is implemented in Linux. <code>clone()</code> without the <code>CLONE_VM</code> set will create a new address space, which is essentially the same as <code>fork()</code> .
<code>mmap()</code>	<code>mmap()</code> creates a new region within the process linear address space.
<code>mremap()</code>	Remaps or resizes a region of memory. If the virtual address space is not available for the mapping, the region may be moved unless the move is forbidden by the caller.
<code>munmap()</code>	Destroys part or all of a region. If the region being unmapped is in the middle of an existing region, the existing region is split into two separate regions.
<code>shmat()</code>	Attaches a shared memory segment to a process address space.
<code>shmdt()</code>	Removes a shared memory segment from an address space.
<code>execve()</code>	Loads a new executable file and replaces the current address space.
<code>exit()</code>	Destroys an address space and all regions.

Table 4.1. System Calls Related to Memory Regions

4.3 Process Address Space Descriptor

The process address space is described by the `mm_struct` struct, meaning that only one exists for each process and is shared between userspace threads. In fact, threads are identified in the task list by finding all `task_struct`s that have pointers to the same `mm_struct`.

A unique `mm_struct` is not needed for kernel threads because they will never page fault or access the userspace portion. The only exception is page faulting within the `vmalloc` space. The page fault handling code treats this as a special case and updates the current page table with information in the master page table. Because an `mm_struct` is not needed for kernel threads, the `task_struct->mm` field for kernel threads is always `NULL`. For some tasks, such as the boot idle task, the `mm_struct` is never set up, but, for kernel threads, a call to `daemonize()` will call `exit_mm()` to decrement the usage counter.

Because TLB flushes are extremely expensive, especially with architectures such as the PowerPC (PPC), a technique called *lazy TLB* is employed, which avoids unnecessary TLB flushes by processes that do not access the userspace page tables because the kernel portion of the address space is always visible. The call to `switch_mm()`, which results in a TLB flush, is avoided by borrowing the `mm_struct` used by the previous task and placing it in `task_struct->active_mm`. This technique has made large improvements to context switch times.

When entering lazy TLB, the function `enter_lazy_tlb()` is called to ensure that a `mm_struct` is not shared between processors in Symmetric Multiprocessing (SMP) machines, making it a NULL operation on UP machines. The second-time use of lazy TLB is during process exit when `start_lazy_tlb()` is used briefly while the process is waiting to be reaped by the parent.

The struct has two reference counts called `mm_users` and `mm_count` for two types of users. `mm_users` is a reference count of processes accessing the userspace portion of this `mm_struct`, such as the page tables and `mm` mappings. Threads and the `swap_out()` code, for instance, will increment this count and make sure an `mm_struct` is not destroyed early. When it drops to 0, `exit_mmap()` will delete all mappings and tear down the page tables before decrementing the `mm_count`.

`mm_count` is a reference count of the anonymous users for the `mm_struct` initialized at 1 for the real user. An anonymous user is one that does not necessarily care about the userspace portion and is just borrowing the `mm_struct`. Example users are kernel threads that use lazy TLB switching. When this count drops to 0, the `mm_struct` can be safely destroyed. Both reference counts exist because anonymous users need the `mm_struct` to exist even if the userspace mappings get destroyed and there is no point delaying the teardown of the page tables.

The `mm_struct` is defined in `<linux/sched.h>` as follows:

```

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;
221
222     unsigned long start_code, end_code, start_data, end_data;
223     unsigned long start_brk, brk, start_stack;
224     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 };

```


The meaning of each of the fields in this sizeable struct is as follows:

mmap The head of a linked list of all VMA regions in the address space.

mm_rb The VMAs are arranged in a linked list and in a red-black tree for fast lookups. This is the root of the tree.

mmap_cache The VMA found during the last call to `find_vma()` is stored in this field on the assumption that the area will be used again soon.

pgd The PGD for this process.

mm_users A reference count of users accessing the userspace portion of the address space as explained at the beginning of the section.

mm_count A reference count of the anonymous users for the `mm_struct` that starts at 1 for the real user as explained at the beginning of this section.

map_count Number of VMAs in use.

mmap_sem This is a long-lived lock that protects the VMA list for readers and writers. Because users of this lock require it for a long time and may need to sleep, a spinlock is inappropriate. A reader of the list takes this semaphore with `down_read()`. If they need to write, it is taken with `down_write()`, and the `page_table_lock` spinlock is later acquired while the VMA linked lists are being updated.

page_table_lock This protects most fields on the `mm_struct`. As well as the page tables, it protects the Resident Set Size (RSS) (see `rss`) count and the VMA from modification.

mmlist All `mm_structs` are linked together by this field.

start_code, end_code The start and end address of the code section.

start_data, end_data The start and end address of the data section.

start_brk, brk The start and end address of the heap.

start_stack Predictably enough, the start of the stack region.

arg_start, arg_end The start and end address of command-line arguments.

env_start, env_end The start and end address of environment variables.

rss RSS is the number of resident pages for this process. It should be noted that the global zero page is not accounted for by RSS.

total_vm The total memory space occupied by all VMA regions in the process.

locked_vm The number of resident pages locked in memory.

def_flags Only one possible value, `VM_LOCKED`. It is used to determine if all future mappings are locked by default.

cpu_vm_mask A bitmask representing all possible CPUs in an SMP system. The mask is used by an *InterProcessor Interrupt (IPI)* to determine if a processor should execute a particular function or not. This is important during TLB flush for each CPU.

swap_address Used by the pageout daemon to record the last address that was swapped from when swapping out entire processes.

dumpable Set by `prctl()`. This is important only when tracing a process.

context Architecture-specific MMU context.

There are a small number of functions for dealing with `mm_struct`s. They are described in Table 4.2.

Function	Description
<code>mm_init()</code>	Initializes an <code>mm_struct</code> by setting starting values for each field, allocating a PGD, initializing spinlocks, etc.
<code>allocate_mm()</code>	Allocates an <code>mm_struct()</code> from the slab allocator
<code>mm_alloc()</code>	Allocates an <code>mm_struct</code> using <code>allocate_mm()</code> and calls <code>mm_init()</code> to initialize it
<code>exit_mmap()</code>	Walks through an <code>mm_struct</code> and unmaps all VMAs associated with it
<code>copy_mm()</code>	Makes an exact copy of the current tasks <code>mm_struct</code> needs for a new task. This is only used during fork.
<code>free_mm()</code>	Returns the <code>mm_struct</code> to the slab allocator

Table 4.2. Functions Related to Memory Region Descriptors

4.3.1 Allocating a Descriptor

Two functions are provided to allocate an `mm_struct`. To be slightly confusing, they are essentially the same, but with small important differences. `allocate_mm()` is just a preprocessor macro that allocates an `mm_struct` from the *slab allocator* (see Chapter 8). `mm_alloc()` allocates from slab and then calls `mm_init()` to initialize it.

4.3.2 Initializing a Descriptor

The first `mm_struct` in the system that is initialized is called `init_mm`. All subsequent `mm_struct`s are copies of a parent `mm_struct`. That means that `init_mm` has to be statically initialized at compile time. This static initialization is performed by the macro `INIT_MM()`.

```

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 }

```

After it is established, new `mm_struct`s are created using their parent `mm_struct` as a template. The function responsible for the copy operation is `copy_mm()`, and it uses `init_mm()` to initialize process-specific `mm_struct`s.

4.3.3 Destroying a Descriptor

While a new user increments the usage count with `atomic_inc(&mm->mm_users)`, it is decremented with a call to `mmaput()`. If the `mm_users` count reaches zero, all the mapped regions are destroyed with `exit_mmap()`, and the page tables are destroyed because there are no longer any users of the userspace portions. The `mm_count` count is decremented with `mmdrop()` because all the users of the page tables and VMAs are counted as one `mm_struct` user. When `mm_count` reaches zero, the `mm_struct` will be destroyed.

4.4 Memory Regions

The full address space of a process is rarely used. Only sparse regions are. Each region is represented by a `vm_area_struct`, which never overlaps and represents a set of addresses with the same protection and purpose. Examples of a region include a read-only shared library loaded into the address space or the process heap. A full list of mapped regions that a process has may be viewed using the `proc` interface at `/proc/PID/maps` where `PID` is the process ID of the process that is to be examined.

The region may have a number of different structures associated with it as illustrated in Figure 4.2. At the top, there is the `vm_area_struct`, which, on its own, is enough to represent anonymous memory.

If the region is backed by a file, the `struct file` is available through the `vm_file` field, which has a pointer to the `struct inode`. The `inode` is used to get the `struct address_space`, which has all the private information about the file, including a set of pointers to system functions that perform the file-specific operations, such as reading and writing pages to disk.

The struct `vm_area_struct` is declared as follows in `<linux/mm.h>`:

```

44 struct vm_area_struct {
45     struct mm_struct * vm_mm;
46     unsigned long vm_start;
47     unsigned long vm_end;
48
49     /* linked list of VM areas per task, sorted by address */
50     struct vm_area_struct *vm_next;
51
52     pgprot_t vm_page_prot;
53     unsigned long vm_flags;
54
55     rb_node_t vm_rb;
56
57     struct vm_area_struct *vm_next_share;
58     struct vm_area_struct **vm_pprev_share;
59
60     /* Function pointers to deal with this struct. */
61     struct vm_operations_struct * vm_ops;
62
63     /* Information about our backing store: */
64     unsigned long vm_pgoff;
65     struct file * vm_file;
66     unsigned long vm_raend;
67     void * vm_private_data;
68 };

```

Here is a brief description of the fields.

vm_mm The `mm_struct` this VMA belongs to.

vm_start The starting address of the region.

vm_end The end address of the region.

vm_next All the VMAs in an address space are linked together in an address-ordered singly linked list by this field. It is interesting to note that the VMA list is one of the very rare cases where a singly linked list is used in the kernel.

vm_page_prot The protection flags that are set for each PTE in this VMA. The different bits are described in Table 3.1.

vm_flags A set of flags describing the protections and properties of the VMA. They are all defined in `<linux/mm.h>` and are described in Table 4.3.

vm_rb As well as being in a linked list, all the VMAs are stored on a *red-black tree* for fast lookups. This is important for page fault handling when finding the correct region quickly is important, especially for a large number of mapped regions.

Protection Flags	
Flags	Description
VM_READ	Pages may be read.
VM_WRITE	Pages may be written.
VM_EXEC	Pages may be executed.
VM_SHARED	Pages may be shared.
VM_DONTCOPY	VMA will not be copied on fork.
VM_DONTEXPAND	Prevents a region from being resized. Flag is unused.
mmap Related Flags	
VM_MAYREAD	Allows the VM_READ flag to be set.
VM_MAYWRITE	Allows the VM_WRITE flag to be set.
VM_MAYEXEC	Allows the VM_EXEC flag to be set.
VM_MAYSHARE	Allows the VM_SHARED flag to be set.
VM_GROWSDOWN	Shared segment (probably stack) may grow down.
VM_GROWSUP	Shared segment (probably heap) may grow up.
VM_SHM	Pages are used by shared SHM memory segment.
VM_DENYWRITE	What MAP_DENYWRITE for mmap() translates to. It is now unused.
VM_EXECUTABLE	What MAP_EXECUTABLE for mmap() translates to. It is now unused.
VM_STACK_FLAGS	Flags used by setup_arg_flags() to set up the stack.
Locking Flags	
VM_LOCKED	If set, the pages will not be swapped out. It is set by mlock().
VM_IO	Signals that the area is an mmaped region for I/O to a device. It will also prevent the region from being core dumped.
VM_RESERVED	Do not swap out this region. It is used by device drivers.
madvise() Flags	
VM_SEQ_READ	A hint that pages will be accessed sequentially.
VM_RAND_READ	A hint stating that read-ahead in the region is useless.

Table 4.3. Memory Region Flags

- vm_next_share** Links together shared VMA regions based on `vm_mappings` (such as shared libraries).
- vm_pprev_share** The complement of `vm_next_share`.
- vm_ops** The `vm_ops` field contains functions pointers for `open()`, `close()` and `nopage()`. These are needed for syncing with information from the disk.
- vm_pgo** The page aligned offset within a `vm` that is memory mapped.
- vm_file** The struct file pointer to the file being mapped.
- vm_raend** The end address of a read-ahead window. When a fault occurs, a number of additional pages after the desired page will be paged in. This determines how many additional pages are faulted in.

`vm_private_data` Used by some device drivers to store private information and is not of concern to the memory manager.

All the regions are linked together on a linked list ordered by address through the `vm_next` field. When searching for a free area, it is a simple matter of traversing the list, but a frequent operation is to search for the VMA for a particular address, such as during page faulting, for example. In this case, the red-black tree is traversed because it has $O(\log N)$ search time on average. The tree is ordered so that lower addresses than the current node are on the left leaf and higher addresses are on the right.

4.4.1 Memory Region Operations

There are three operations which a VMA may support called `open()`, `close()` and `nopage()`. VMA supports these with a `vm_operations_struct` in the VMA called `vma->vm_ops`. The struct contains three function pointers and is declared as follows in `<linux/mm.h>`:

```
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,
                             unsigned long address,
                             int unused);
137 };
```

The `open()` and `close()` functions are called every time a region is created or deleted. These functions are only used by a small number of devices, one for system and System V shared regions, which need to perform additional operations when regions are opened or closed. For example, the System V `open()` callback will increment the number of VMAs using a shared segment (`shp->shm_nattch`).

The main operation of interest is the `nopage()` callback. This callback is used during a page-fault by `do_no_page()`. The callback is responsible for locating the page in the page cache or allocating a page and populating it with the required data before returning it.

Most files that are mapped will use a generic `vm_operations_struct()` called `generic_file_vm_ops`. It registers only a `nopage()` function called `filemap_nopage()`. This `nopage()` function will either locate the page in the page cache or read the information from disk. The struct is declared as follows in `mm/filemap.c`:

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

4.4.2 File/Device-Backed Memory Regions

In the event the region is backed by a file, the `vm_file` leads to an associated `address_space` as shown in Figure 4.2. The struct contains information of relevance

to the `%s` system such as the number of dirty pages that must be flushed to disk. It is declared as follows in `<linux/fs.h>`:

```

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 };

```

A brief description of each field is as follows:

`clean_pages` is a list of clean pages that need no synchronization with backing storage.

`dirty_pages` is a list of dirty pages that need synchronization with backing storage.

`locked_pages` is a list of pages that are locked in memory.

`nrpages` is the number of resident pages in use by the address space.

`a_ops` is a struct of function for manipulating the `%s` system. Each `%s` system provides its own `address_space_operations`, although they sometimes use generic functions.

`host` is the host inode the `%s` belongs to.

`i_mmap` is a list of private mappings using this `address_space`.

`i_mmap_shared` is a list of VMAs that share mappings in this `address_space`.

`i_shared_lock` is a spinlock to protect this structure.

`gfp_mask` is the mask to use when calling `__alloc_pages()` for new pages.

Periodically, the memory manager will need to flush information to disk. The memory manager does not know and does not care how information is written to disk, so the `a_ops` struct is used to call the relevant functions. It is declared as follows in `<linux/fs.h>`:

```

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      * Don't 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 };

```

These fields are all function pointers and are described in the following:

writepage Writes a page to disk. The offset within the page to write to is stored within the page struct. It is up to the system-specific code to find the block. See `buffer.c:block_write_full_page()`.

readpage Reads a page from disk. See `buffer.c:block_read_full_page()`.

sync_page Syncs a dirty page with disk. See `buffer.c:block_sync_page()`.

prepare_write This is called before data is copied from userspace into a page that will be written to disk. With a journaled system, this ensures the system log is up to date. With normal systems, it makes sure the needed buffer pages are allocated. See `buffer.c:block_prepare_write()`.

commit_write After the data has been copied from userspace, this function is called to commit the information to disk. See `buffer.c:block_commit_write()`.

bmap Maps a block so that raw I/O can be performed. It is mainly of concern to system-specific code, although it is also used when swapping out pages that are backed by a swap file instead of a swap partition.

- flushpage** Makes sure there is no I/O pending on a page before releasing it. See `buffer.c:discard_bh_page()`.
- releasepage** Tries to flush all the buffers associated with a page before freeing the page itself. See `try_to_free_buffers()`.
- direct_I/O** This function is used when performing direct I/O to an inode. The `#define` exists so that external modules can determine at compile time if the function is available because it was only introduced in 2.4.21.
- direct_file_I/O** Used to perform direct I/O with a `struct file`. Again, the `#define` exists for external modules because this API was only introduced in 2.4.22.
- removepage** An optional callback that is used when a page is removed from the page cache in `remove_page_from_inode_queue()`.

4.4.3 Creating a Memory Region

The system call `mmap()` is provided for creating new memory regions within a process. For the x86, the function calls `sys_mmap2()`, which calls `do_mmap2()`, directly with the same parameters. `do_mmap2()` is responsible for acquiring the parameters needed by `do_mmap_pgoff()`, which is the principal function for creating new areas for all architectures.

`do_mmap2()` first clears the `MAP_DENYWRITE` and `MAP_EXECUTABLE` bits from the flags parameter because they are ignored by Linux, which is confirmed by the `mmap()` manual page. If a file is being mapped, `do_mmap2()`, shown in Figure 4.3, will look up the `struct file` based on the file descriptor passed as a parameter and will acquire the `mm_struct → mmap_sem` semaphore before calling `do_mmap_pgoff()`.

`do_mmap_pgoff()` begins by performing some basic sanity checks. It first checks that the appropriate system or device functions are available if a file or device is being mapped. It then ensures the size of the mapping is page aligned and that it does not attempt to create a mapping in the kernel portion of the address space. It then makes sure the size of the mapping does not overflow the range of `pgoff` and finally that the process does not have too many mapped regions already.

This rest of the function is large, but, broadly speaking, it takes the following steps:

1. Sanity check the parameters.
2. Find a free linear address space large enough for the memory mapping. If a system or device specific `get_unmapped_area()` function is provided, it will be used. Otherwise, `arch_get_unmapped_area()` is called.
3. Calculate the VM flags and check them against the process access permissions.
4. If an old area exists where the mapping is to take place, fix it up so that it is suitable for the new mapping.
5. Allocate a `vm_area_struct` from the slab allocator and fill in its entries.

6. Link in the new VMA.
7. Call the `mmap` system or device-specific `mmap` function.
8. Update statistics and exit.

4.4.4 Finding a Mapped Memory Region

A common operation is to find the VMA that a particular address belongs to, such as during operations like page faulting, and the function responsible for this is `find_vma()`. The function `find_vma()` and other API functions affecting memory regions are listed in Table 4.4.

It first checks the `mmap_cache`, which caches the result of the last call to `find_vma()` because it is quite likely the same region will be needed a few times in succession. If it is not the desired region, the red-black tree stored in the `mm_rb` is traversed. If the desired address is not contained within any VMA, the function will return the VMA *closest* to the requested address, so it is important callers double-check to ensure the returned VMA contains the desired address.

A second function called `find_vma_prev()` is provided, which is functionally the same as `find_vma()` except that it also returns a pointer to the VMA preceding the desired VMA, which is required as the list is a singly linked list. `find_vma_prev()` is rarely used, but notably it is used when two VMAs are being compared to determine if they may be merged. It is also used when removing a memory region so that the singly linked list may be updated.

The last function of note for searching VMAs is `find_vma_intersection()`, which is used to find a VMA that overlaps a given address range. The most notable use of this is during a call to `do_brk()` when a region is growing up. It is important to ensure that the growing region will not overlap an old region.

4.4.5 Finding a Free Memory Region

When a new area is to be memory mapped, a free region has to be found that is large enough to contain the new mapping. The function responsible for finding a free area is `get_unmapped_area()`.

As the call graph in Figure 4.4 indicates, little work is involved with finding an unmapped area. The function is passed a number of parameters. A `struct file` is passed that represents the `file` or device to be mapped as well as `pgoff`, which is the offset within the `file` that is being mapped. The requested address for the mapping is passed as well as its length. The last parameter is the protection flags for the area.

If a device is being mapped, such as a video card, the associated `file_operations` `get_unmapped_area()` is used. This is because devices or `files` may have additional requirements for mapping that generic code cannot be aware of, such as the address having to be aligned to a particular virtual address.

If there are no special requirements, the architecture-specific function `arch_get_unmapped_area()` is called. Not all architectures provide their own function. For those that don't a generic version is provided in `mm/mmap.c`.

<pre>struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr)</pre> Finds the VMA that covers a given address. If the region does not exist, it returns the VMA closest to the requested address.
<pre>struct vm_area_struct * find_vma_prev(struct mm_struct * mm, unsigned long addr, struct vm_area_struct **pprev)</pre> The same as <code>find_vma()</code> except that it also gives the VMA pointing to the returned VMA. It is not often used, with <code>sys_mprotect()</code> being the notable exception, because usually <code>find_vma_prepare()</code> is required.
<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> The same as <code>find_vma()</code> except that it will also ~~add~~ the preceeding VMA in the linked list as well as the red-black tree nodes needed to perform an insertion into the tree.
<pre>struct vm_area_struct * find_vma_intersection(struct mm_struct * mm, unsigned long start_addr, unsigned long end_addr)</pre> Returns the VMA that intersects a given address range. It is useful when checking if a linear address region is in use by any VMA.
<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> Attempts to expand the supplied VMA to cover a new address range. If the VMA cannot be expanded forward, the next VMA is checked to see if it may be expanded backward to cover the address range instead. Regions may be merged if there is no ~~no~~ device mapping and the permissions match.
<pre>unsigned long get_unmapped_area(struct file *file, unsigned long addr, unsigned long len, unsigned long pgoff, unsigned long flags)</pre> Returns the address of a free region of memory large enough to cover the requested size of memory. It is used principally when a new VMA is to be created.
<pre>void insert_vm_struct(struct mm_struct *, struct vm_area_struct *)</pre> Inserts a new VMA into a linear address space.

Table 4.4. Memory Region VMA API

4.4.6 Inserting a Memory Region

The principal function for inserting a new memory region is `insert_vm_struct()` that has the call graph seen in Figure 4.5. It is a very simple function that ~~get~~

Two varieties of linking functions are provided, `vma_link()` and `__vma_link()`. `vma_link()` is intended for use when no locks are held. It will acquire all the necessary locks, including locking the `%s` if the VMA is a `%s` mapping, before calling `__vma_link()`, which places the VMA in the relevant lists.

Many functions do not use the `insert_vm_struct()` functions, but instead prefer to call `find_vma_prepare()` themselves, followed by a later `vma_link()` to avoid having to traverse the tree multiple times.

The linking in `__vma_link()` consists of three stages that are contained in three separate functions. `__vma_link_list()` inserts the VMA into the linear, singly linked list. If it is the `%s` mapping in the address space (i.e., `prev` is `NULL`), it will become the red-black tree root node. The second stage is linking the node into the red-black tree with `__vma_link_rb()`. The `%s` stage is `%s`ing up the `%s`share mapping with `__vma_link_file()`, which basically inserts the VMA into the linked list of VMAs using the `vm_pprev_share` and `vm_next_share` `%s`s.

4.4.7 Merging Contiguous Regions

Linux used to have a function called `merge_segments()` [Hac02] that was responsible for merging adjacent regions of memory together if the `%s` and permissions matched. The objective was to remove the number of VMAs required, especially because many operations resulted in a number of mappings being created, such as calls to `sys_mprotect()`. This was an expensive operation because it could result in large portions of the mappings been traversed and was later removed as applications, especially those with many mappings, spent a long time in `merge_segments()`.

The equivalent function that exists now is called `vma_merge()`, and it is only used in two places. The `%s` is user in `sys_mmap()`, which calls it if an anonymous region is being mapped because anonymous regions are frequently mergeable. The second time is during `do_brk()`, which is expanding one region into a newly allocated one where the two regions should be merged. Rather than merging two regions, the function `vma_merge()` checks if an existing region may be expanded to satisfy the new allocation, which negates the need to create a new region. A region may be expanded if there are no `%s` or device mappings and the permissions of the two areas are the same.

Regions are merged elsewhere, although no function is explicitly called to perform the merging. The `%s` is during a call to `sys_mprotect()` during the `%s` of areas where the two regions will be merged if the two sets of permissions are the same after the permissions in the affected region change. The second is during a call to `move_vma()` when it is likely that similar regions will be located beside each other.

4.4.8 Remapping and Moving a Memory Region

`mremap()` is a system call provided to grow or shrink an existing memory mapping. This is implemented by the function `sys_mremap()`, which may move a memory region if it is growing or it would overlap another region and if `MREMAP_FIXED` is not specified in the `%s`. The call graph is illustrated in Figure 4.6.

If a region is to be moved, `do_mremap()` `%s` calls `get_unmapped_area()` to `%s` a region large enough to contain the new resized mapping and then calls `move_vma()`

There are some limitations to what memory may be locked. The address range must be page aligned because VMAs are page aligned. This is addressed by simply rounding the range up to the nearest page-aligned range. The second proviso is that the process limit `RLIMIT_MLOCK` imposed by the system administrator may not be exceeded. The last proviso is that each process may only lock half of physical memory at a time. This is a bit nonfunctional because there is nothing to stop a process forking a number of times and each child locking a portion, but, because only root processes are allowed to lock pages, it does not make much difference. It is safe to presume that a root process is trusted and knows what it is doing. If it does not, the system administrator with the resulting broken system probably deserves it and gets to keep both parts of it.

4.4.10 Unlocking the Region

The system calls `munlock()` and `munlockall()` to provide the corollary for the locking functions and mapping to `sys_munlock()` and `sys_munlockall()`, respectively. The functions are much simpler than the locking functions because they do not have to make numerous checks. They both rely on the same `do_mmap()` function to ~~map~~ the regions.

4.4.11 Fixing Up Regions After Locking

When locking or unlocking, VMAs will be affected in one of four ways, each of which must be ~~fixed~~ up by `mlock_fixup()`. The locking may affect the whole VMA, in which case `mlock_fixup_all()` is called. The second condition, handled by `mlock_fixup_start()`, is where the start of the region is locked, requiring that a new VMA be allocated to map the new area. The third condition, handled by `mlock_fixup_end()`, is predictably enough where the end of the region is locked. Finally, `mlock_fixup_middle()` handles the case where the middle of a region is mapped requiring two new VMAs to be allocated.

It is interesting to note that VMAs created as a result of locking are never merged, even when unlocked. It is presumed that processes that lock regions will need to lock the same regions over and over again, and it is not worth the processor power to constantly merge and split regions.

4.4.12 Deleting a Memory Region

The function responsible for deleting memory regions, or parts thereof, is `do_munmap()`, which is shown in Figure 4.10. It is a relatively simple operation in comparison with the other memory region-related operations and is basically divided up into three parts. The ~~first~~ is to ~~map~~ the red-black tree for the region that is about to be unmapped. The second is to release the pages and PTEs related to the region to be unmapped, and the third is to ~~map~~ the regions if a hole has been generated.

To ensure the red-black tree is ordered correctly, all VMAs to be affected by the unmap are placed on a linked list called `free` and then deleted from the red-black tree with `rb_erase()`. The regions, if they still exist, will be added with their new

addresses later during the `zap`.

Next the linked-list VMAs on `free` are walked through and checked to ensure it is not a partial unmapping. Even if a region is just to be partially unmapped, `remove_shared_vm_struct()` is still called to remove the shared `zap` mapping. Again, if this is a partial unmapping, it will be recreated during `zap`. `zap_page_range()` is called to remove all the pages associated with the region about to be unmapped before `unmap_fixup()` is called to handle partial unmappings.

Last, `free_pgtables()` is called to try and free up all the page table entries associated with the unmapped region. It is important to note that the page table entry freeing is not exhaustive. It will only unmap full PGD directories and their entries, so, for example, if only half a PGD was used for the mapping, no page table entries will be freed. This is because a `zap` grain freed of page table entries would be too expensive to free up data structures that are both small and likely to be used again.

4.4.13 Deleting All Memory Regions

During process exit, it is necessary to unmap all VMAs associated with an `mm_struct`. The function responsible is `exit_mmap()`. It is a very simple function that `zap` the CPU cache before walking through the linked list of VMAs, unmapping each of them in turn and freeing up the associated pages before `zap` the TLB and deleting the page table entries. It is covered in detail in the Code Commentary.

4.5 Exception Handling

A very important part of VM is how kernel address space exceptions, which are not bugs, are caught.¹ This section does *not* cover the exceptions that are raised with errors such as divide by zero. I am only concerned with the exception raised as the result of a page fault. There are two situations where a bad reference may occur. The first is where a process sends an invalid pointer to the kernel by a system call, which the kernel must be able to safely trap because the only check made initially is that the address is below `PAGE_OFFSET`. The second is where the kernel uses `copy_from_user()` or `copy_to_user()` to read or write data from userspace.

At compile time, the linker creates an exception table in the `__ex_table` section of the kernel code segment, which starts at `__start__ex_table` and ends at `__stop__ex_table`. Each entry is of type `exception_table_entry`, which is a pair consisting of an execution point and a `zap` routine. When an exception occurs that the page fault handler cannot manage, it calls `search_exception_table()` to see if a `zap` routine has been provided for an error at the faulting instruction. If module support is compiled, each module's exception table will also be searched.

¹Many thanks go to Ingo Oeser for clearing up the details of how this is implemented.

If the address of the current exception is found in the table, the corresponding location of the trap code is returned and executed. We will see in Section 4.7 how this is used to trap bad reads and writes to userspace.

4.6 Page Faulting

Pages in the process linear address space are not necessarily resident in memory. For example, allocations made on behalf of a process are not satisfied immediately because the space is just reserved within the `vm_area_struct`. Other examples of nonresident pages include the page having been swapped out to backing storage or writing a read-only page.

Linux, like most operating systems, has a *Demand Fetch* policy as its fetch policy for dealing with pages that are not resident. This states that the page is only fetched from backing storage when the hardware raises a page fault exception, which the operating system traps and allocates a page. The characteristics of backing storage imply that some sort of page prefetching policy would result in less page faults [MM87], but Linux is fairly primitive in this respect. When a page is paged in from swap space, a number of pages after it, up to $2^{\text{page\_cluster}}$ are read in by `swpin_readahead()` and placed in the swap cache. Unfortunately, there is only a chance that pages likely to be used soon will be adjacent in the swap area, which makes it a poor prepaging policy. Linux would likely benefit from a prepaging policy that adapts to program behavior [KMC02].

There are two types of page fault, major and minor faults. Major page faults occur when data has to be read from disk, which is an expensive operation, or the fault is referred to as a minor, or soft, page fault. Linux maintains statistics on the number of these types of page faults with the `task_struct` `→majflt` and `task_struct` `→minflt` fields, respectively.

The page fault handler in Linux is expected to recognize and act on a number of different types of page faults listed in Table 4.5, which will be discussed in detail later in this chapter.

Each architecture registers an architecture-specific function for the handling of page faults. Although the name of this function is arbitrary, a common choice is `do_page_fault()`, for which the call graph for the x86 is shown in Figure 4.11.

This function is provided with a wealth of information such as the address of the fault, whether the page was simply not found or was a protection error, whether it was a read or write fault and whether it is a fault from user or kernel space. It is responsible for determining which type of fault has occurred and how it should be handled by the architecture-independent code. The flow chart, in Figure 4.12, shows broadly speaking what this function does. In the figure, identifiers with a colon after them correspond to the label as shown in the code.

`handle_mm_fault()` is the architecture-independent, top-level function for faulting in a page from backing storage, performing Copy-On-Write (COW), and so on. If it returns 1, it was a minor fault, 2 was a major fault, 0 sends a SIGBUS error and any other value invokes the out of memory handler.

Exception	Type	Action
Region valid, but page not allocated	Minor	Allocate a page frame from the physical page allocator.
Region not valid but is beside an expandable region like the stack	Minor	Expand the region and allocate a page.
Page swapped out, but present in swap cache	Minor	Re-establish the page in the process page tables and drop a reference to the swap cache.
Page swapped out to backing storage	Major	Find where the page with information is stored in the PTE and read it from disk.
Page write when marked read-only	Minor	If the page is a COW page, make a copy of it, mark it writable and assign it to the process. If it is in fact a bad write, send a SIGSEGV signal.
Region is invalid or process has no permissions to access	Error	Send a SIGSEGV signal to the process.
Fault occurred in the kernel portion address space	Minor	If the fault occurred in the vmalloc area of the address space, the current process page tables are updated against the master page table held by <code>init_mm</code> . This is the only valid kernel page fault that may occur.
Fault occurred in the userspace region while in kernel mode	Error	If a fault occurs, it means a kernel system did not copy from userspace properly and caused a page fault. This is a kernel bug that is treated quite severely.

Table 4.5. Reasons for Page Faulting

4.6.1 Handling a Page Fault

After the exception handler has decided the fault is a valid page fault in a valid memory region, the architecture-independent function `handle_mm_fault()`, which has its call graph shown in Figure 4.13, takes over. It allocates the required page table entries if they do not already exist and calls `handle_pte_fault()`.

Based on the properties of the PTE, one of the handler functions shown in Figure 4.13 will be used. The first stage of the decision is to check if the PTE is marked not present or if it has been allocated with, which is checked by `pte_present()` and `pte_none()`. If no PTE has been allocated (`pte_none()` returned true), `do_no_page()` is called, which handles *Demand Allocation*. Otherwise, it is a page that has been swapped out to disk and `do_swap_page()` performs *Demand Paging*.

and first time write. Because it is an anonymous page, the first read is an easy case because no data exists. In this case, the systemwide `empty_zero_page`, which is just a page of zeros, is mapped for the PTE, and the PTE is write protected. The write protection is set so that another page fault will occur if the process writes to the page. On the x86, the global zero-filled page is zeroed out in the function `mem_init()`.

If this is the first write to the page, `alloc_page()` is called to allocate a free page (see Chapter 6) and is zero-filled by `clear_user_highpage()`. Assuming the page was successfully allocated, the RSS field in the `mm_struct` will be incremented; `flush_page_to_ram()` is called as required when a page has been inserted into a userspace process by some architectures to ensure cache coherency. The page is then inserted on the LRU lists so that it may be reclaimed later by the page reclaiming code. Finally the page table entries for the process are updated for the new mapping.

Handling File/Device-Backed Pages If backed by a file or device, a `nopage()` function will be provided within the `VMA`'s `vm_operations_struct`. In the file-backed case, the function `filemap_nopage()` is frequently the `nopage()` function for allocating a page and reading a page-sized amount of data from disk. Pages backed by a virtual file, such as those provided by *shmfs*, will use the function `shmem_nopage()` (See Chapter 12). Each device driver provides a different `nopage()`. Their internals are unimportant to us here as long as it returns a valid `struct page` to use.

On return of the page, a check is made to ensure a page was successfully allocated and appropriate errors were returned if not. A check is then made to see if an early COW break should take place. An early COW break will take place if the fault is a write to the page and the `VM_SHARED` flag is not included in the managing VMA. An early break is a case of allocating a new page and copying the data across before reducing the reference count to the page returned by the `nopage()` function.

In either case, a check is then made with `pte_none()` to ensure a PTE is not already in the page table that is about to be used. It is possible with SMP that two faults would occur for the same page at close to the same time, and because the spinlocks are not held for the full duration of the fault, this check has to be made at the last instant. If there has been no race, the PTE is assigned, statistics are updated and the architecture hooks for cache coherency are called.

4.6.3 Demand Paging

When a page is swapped out to backing storage, the function `do_swap_page()`, shown in Figure 4.15, is responsible for reading the page back in, with the exception of virtual files, which are covered in Section 12. The information needed to find it is stored within the PTE itself. The information within the PTE is enough to find the page in swap. Because pages may be shared between multiple processes, they cannot always be swapped out immediately. Instead, when a page is swapped out, it is placed within the swap cache.

A shared page cannot be swapped out immediately because there is no way of mapping a struct page to the PTEs of each process it is shared between. Searching the page tables of all processes is simply far too expensive. It is worth noting that the late 2.5.x kernels and 2.4.x with a custom patch have what is called *Reverse Mapping (RMAP)*, which is discussed at the end of the chapter.

With the swap cache existing, it is possible that, when a fault occurs, it still exists in the swap cache. If it is, the reference count to the page is simply increased, and it is placed within the process page tables again and registers as a minor page fault.

If the page exists only on disk, `swpin_readahead()` is called, which reads in the requested page and a number of pages after it. The number of pages read in is determined by the variable `page_cluster` defined in `mm/swap.c`. On low memory machines with less than 16MiB of RAM, it is initialized as 2 or 3. The number of pages read in is $2^{\text{page\_cluster}}$ unless a bad or empty swap entry is encountered. This works on the premise that a seek is the most expensive operation in time, so after the seek has completed, the succeeding pages should also be read in.

4.6.4 COW Pages

Once upon a time, the full parent address space was duplicated for a child when a process forked. This was an extremely expensive operation because it is possible a significant percentage of the process would have to be swapped in from backing storage. To avoid this considerable overhead, a technique called COW is employed.

During a fork, the PTEs of the two processes are made read-only so that, when a write occurs, there will be a page fault. Linux recognizes a COW page because, even though the PTE is write protected, the controlling VMA shows the region is writable. It uses the function `do_wp_page()`, shown in Figure 4.16, to handle it by making a copy of the page and assigning it to the writing process. If necessary, a new swap slot will be reserved for the page. With this method, only the page table entries have to be copied during a fork.

4.7 Copying to/from Userspace

It is not safe to access memory in the process address space directly because there is no way to quickly check if the page addressed is resident or not. Linux relies on the MMU to raise exceptions when the address is invalid and have the Page Fault Exception handler catch the exception and fix it up. In the x86 case, an assembler is provided by the `__copy_user()` to trap exceptions where the address is totally useless. The location of the trap code is found when the function `search_exception_table()` is called. Linux provides an ample API (mainly macros) for copying data to and from the user address space safely as shown in Table 4.6.

<pre>unsigned long copy_from_user(void *to, const void *from, unsigned long n)</pre>	Copies n bytes from the user address(from) to the kernel address space(to).
<pre>unsigned long copy_to_user(void *to, const void *from, unsigned long n)</pre>	Copies n bytes from the kernel address(from) to the user address space(to).
<pre>void copy_user_page(void *to, void *from, unsigned long address)</pre>	Copies data to an anonymous or COW page in userspace. Ports are responsible for avoiding D-cache aliases. It can do this by using a kernel virtual address that would use the same cache lines as the virtual address.
<pre>void clear_user_page(void *page, unsigned long address)</pre>	Similar to copy_user_page(), except it is for zeroing a page.
<pre>void get_user(void *to, void *from)</pre>	Copies an integer value from userspace (from) to kernel space (to).
<pre>void put_user(void *from, void *to)</pre>	Copies an integer value from kernel space (from) to userspace (to).
<pre>long strncpy_from_user(char *dst, const char *src, long count)</pre>	Copies a null terminated string of at most count bytes long from userspace (src) to kernel space (dst).
<pre>long strlen_user(const char *s, long n)</pre>	Returns the length, upper bound by n, of the userspace string including the terminating NULL.
<pre>int access_ok(int type, unsigned long addr, unsigned long size)</pre>	Returns nonzero if the userspace block of memory is valid and zero otherwise.

Table 4.6. Accessing Process Address Space API

All the macros map on to assembler functions, which all follow similar patterns of implementation. For illustration purposes, we will just trace how `copy_from_user()` is implemented on the x86.

If the size of the copy is known at compile time, `copy_from_user()` calls `__constant_copy_from_user()`, or `__generic_copy_from_user()` is used. If the size is known, there are different assembler optimizations to copy data in 1, 2 or 4 byte strides. Otherwise, the distinction between the two copy functions is not important.

The generic copy function eventually calls the function `__copy_user_zeroing()` in `<asm-i386/uaccess.h>`, which has three important parts. The first part is the assembler for the actual copying of size number of bytes from userspace. If any page is not resident, a page fault will occur, and, if the address is valid, it will get swapped in as normal. The second part is trap code, and the third part is the

`__ex_table` mapping the instructions from the `__start` part to the `__stop` code in the second part.

These pairings, as described in Section 4.5, copy the location of the copy instructions and the location of the `__stop` code to the kernel exception handle table by the linker. If an invalid address is read, the function `do_page_fault()` will fall through, call `search_exception_table()`, find the Enhanced Instruction Pointer (EIP) where the faulty read took place and jump to the `__stop` code, which copies zeros into the remaining kernel space, resets up registers and returns. In this manner, the kernel can safely access userspace with no expensive checks and let the MMU hardware handle the exceptions.

All the other functions that access userspace follow a similar pattern.

4.8 What's New in 2.6

Linear Address Space The linear address space remains essentially the same as 2.4 with no modifications that cannot be easily recognized. The main change is the addition of a new page usable from userspace that has been entered into the kernel address virtual mappings. On the x86, this page is located at `0xFFFFF000` and called the *vsyscall page*. Code is located at this page, which provides the optimal method for entering kernel space from userspace. A userspace program now should use `call 0xFFFFF000` instead of the traditional `int 0x80` when entering kernel space.

struct mm_struct This struct has not changed significantly. The main change is the addition of a `free_area_cache` field, which is initialized as `TASK_UNMAPPED_BASE`. This field is used to remember where the free hole is in the linear address space to improve search times. A small number of fields have been added at the end of the struct, which are related to core dumping and are beyond the scope of this book.

struct vm_area_struct This struct also has not changed significantly. The main difference is that the `vm_next_share` and `vm_pprev_share` have been replaced with a proper linked list with a new field called `shared`. The `vm_ra_end` has been removed altogether because `vm_readahead` is implemented very differently in 2.6. Readahead is mainly managed by a `struct file_ra_state` struct stored in `struct file->f_ra`. How readahead is implemented is described in a lot of detail in `mm/readahead.c`.

struct address_space The main change is relatively minor. The `gfp_mask` field has been replaced with a `flags` field where the `__GFP_BITS_SHIFT` bits are used as the `gfp_mask` and accessed with `mapping_gfp_mask()`. The remaining bits are used to store the status of asynchronous I/O. The two flags that may be set are `AS_EIO` to indicate an I/O error and `AS_ENOSPC` to indicate the filesystem ran out of space during an asynchronous write.

This struct has a number of significant additions, mainly related to the page cache and `vm_readahead`. Because the fields are quite unique, we'll introduce them in detail:

page_tree This is a radix tree of all pages in the page cache for this mapping indexed by the block that the data is located on the physical disk. In 2.4, searching the page cache involved traversing a linked list. In 2.6, it is a

radix tree lookup, which considerably reduces search times. The radix tree is implemented in `lib/radix-tree.c`.

page_lock This is a spinlock that protects `page_tree`.

io_pages When dirty pages are to be written out, they are added to this list before `do_writepages()` is called. As explained in the previous comment, `mpage_writepages()` in `fs/mpage.c`, pages to be written out are placed on this list to avoid deadlocking by locking a page that is already locked for I/O.

dirtyed_when This `md` records, in `ji` es, the ~~last~~ time an inode was dirtied. This `md` determines where the inode is located on the `super_block→s_dirty` list. This prevents a frequently dirtied inode from remaining at the top of the list and starving writeout on other inodes.

backing_dev_info This `md` records readahead-related information. The struct is declared in `include/linux/backing-dev.h` with comments explaining the `mds`.

private_list This is a private list available to the `address_space`. If the helper functions `mark_buffer_dirty_inode()` and `sync_mapping_buffers()` are used, this list links `buffer_heads` through the `buffer_head→b_assoc_buffers` `md`.

private_lock This spinlock is available for the `address_space`. The use of this lock is very convoluted, but some of the uses are explained in the long ChangeLog for 2.5.17 (lwn.net/2002/0523/a/2.5.17.php3). It is mainly related to protecting lists in other mappings that share buffers in this mapping. The lock would not protect this `private_list`, but it would protect the `private_list` of another `address_space` sharing buffers with this mapping.

assoc_mapping This is the `address_space` that backs buffers contained in this mapping's `private_list`.

truncate_count This is incremented when a region is being truncated by the function `invalidate_mmap_range()`. The counter is examined during page fault by `do_no_page()` to ensure that a page is not faulted that was just invalidated.

struct address_space_operations Most of the changes to this struct initially look quite simple, but are actually quite involved. The changed `mds` are the following:

writepage The `writepage()` callback has been changed to take an additional parameter `struct writeback_control`. This struct is responsible for recording information about the writeback, such as if it is congested or not or if the writer is the page allocator for direct reclaim or `kupdated` and contains a handle to the backing `backing_dev_info` to control readahead.

- writepages** This moves all pages from `dirty_pages` to `io_pages` before writing them all out.
- set_page_dirty** This is an `address_space`-specific method of dirtying a page. This is mainly used by the backing storage `address_space_operations` and for anonymous shared pages where there are no buffers associated with the page to be dirtied.
- readpages** This is used when reading in pages so that readahead can be accurately controlled.
- bmap** This has been changed to deal with disk sectors rather than unsigned longs for devices larger than 2^{32} bytes.
- invalidatepage** This is a renaming change. `block_flushpage()` and the callback `flushpage()` have been renamed to `block_invalidatepage()` and `invalidatepage()`.
- direct_I/O** This has been changed to use the new I/O mechanisms in 2.6. The new mechanisms are beyond the scope of this book.

Memory Regions The operation of `mmap()` has two important changes. The first is that it is possible for security modules to register a callback. This callback is called `security_file_mmap()`, which looks up a `security_ops` struct for the relevant function. By default, this will be a NULL operation.

The second is that much stricter address space accounting code is in place. `vm_area_structs` that are to be accounted will have the `VM_ACCOUNT` flag set, which will be all userspace mappings. When userspace regions are created or destroyed, the functions `vm_acct_memory()` and `vm_unacct_memory()` update the variable `vm_committed_space`. This gives the kernel a much better view of how much memory has been committed to userspace.

4GiB/4GiB User/Kernel Split One limitation that exists for the 2.4.x kernels is that the kernel has only 1GiB of virtual address space available, which is visible to all processes. At time of writing, a patch has been developed by Ingo Molnar² which allows the kernel to optionally have its own full 4GiB address space. The patches are available from <http://redhat.com/~mingo/4g-patches/> and are included in the -mm test trees, but it is unclear if it will be merged into the mainstream.

This feature is intended for 32-bit systems that have very large amounts (>16GiB) of RAM. The traditional 3/1 split adequately supports up to 1GiB of RAM. After that, high-memory support allows larger amounts to be supported by temporarily mapping high-memory pages. However, with more RAM, this forms a significant bottleneck. For example, as the amount of physical RAM approached the 60GiB range, almost all the low memory is consumed by `mem_map`. By giving the kernel its own 4GiB virtual address space, it is much easier to support the memory. The serious penalty, though, is that there is a per-syscall TLB flush, which heavily impacts performance.

²See lwn.net/Articles/39283/ for the patch announcement of the patch.

With the patch, only a small 16MiB region of memory is shared between userspace and kernelspace, and this is used to store the Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT), Task State Segments (TSS), Local Descriptor Table (LDT), vsyscall page and the kernel stack. The code for doing the actual switch between the page tables is then contained in the trampoline code for entering/exiting kernelspace. There are a few changes made to the core, such as the removal of direct pointers for accessing userspace buffers, but, by and large, the core kernel is unaffected by this patch.

Nonlinear VMA Population In 2.4, a VMA backed by a file is populated in a linear fashion. This can be optionally changed in 2.6 with the introduction of the `MAP_POPULATE` flag to `mmap()` and the new system call `remap_file_pages()`, which are implemented by `sys_remap_file_pages()`. This system call allows arbitrary pages in an existing VMA to be remapped to an arbitrary location on the backing file by manipulating the page tables.

On page-out, the nonlinear address for the file is encoded within the PTE so that it can be installed again correctly on page fault. How it is encoded is architecture specific, so two macros are defined, `pgoff_to_pte()` and `pte_to_pgoff()`, for the task.

This feature is largely of benefit to applications with a large number of mappings, such as database servers and virtualizing applications such as emulators. It was introduced for a number of reasons. First, VMAs are per-process and can have considerable space requirements, especially for applications with a large number of mappings. Second, the search `get_unmapped_area()` uses for finding a free area in the virtual address space is a linear search, which is very expensive for large numbers of mappings. Third, nonlinear mappings will prefault most of the pages into memory whereas normal mappings may cause a major fault for each page. This can be avoided though, by using the new `MAP_POPULATE` flag with `mmap()` or by using `mlock()`. The last reason is to avoid sparse mappings, which, at worst case, would require one VMA for every page mapped.

However, this feature is not without some serious drawbacks. The first is that the system calls, `truncate()` and `mincore()`, are broken with respect to nonlinear mappings. Both system calls depend on `vm_area_struct->vm_pgoff`, which is meaningless for nonlinear mappings. If a file mapped by a nonlinear mapping is truncated, the pages that exist within the VMA will still remain. It has been proposed that the proper solution is to leave the pages in memory, but make them anonymous. At the time of writing, no solution has been implemented.

The second major drawback is TLB invalidations. Each remapped page will require that the MMU be told the remapping took place with `flush_icache_page()`, but the more important penalty is with the call to `flush_tlb_page()`. Some processors are able to invalidate just the TLB entries related to the page, but other processors implement this by flushing the entire TLB. If remappings are frequent, the performance will degrade due to increased TLB misses and the overhead of constantly entering kernel space. In some ways, these penalties are the worst because the impact is heavily processor dependent.

It is currently unclear what the future of this feature, if it remains, will be. At

the time of writing, there are still ongoing arguments on how the issues with the feature will be ~~fixed~~, but it is likely that nonlinear mappings are going to be treated very differently from normal mappings with respect to pageout, truncation and the reverse mapping of pages. Because the main user of this feature is likely to be databases, this special treatment is not likely to be a problem.

Page Faulting The changes to the page faulting routines are more cosmetic than anything else, other than the necessary changes to support reverse mapping and PTEs in high memory. The main cosmetic change is that the page faulting routines return self-explanatory compile time ~~definitions~~ rather than magic numbers. The possible return values for `handle_mm_fault()` are `VM_FAULT_MINOR`, `VM_FAULT_MAJOR`, `VM_FAULT_SIGBUS` and `VM_FAULT_OOM`.

Boot Memory Allocator

It is impractical to statically initialize all the core kernel memory structures at compile time because there are simply far too many permutations of hardware configurations. To set up even the basic structures, though, requires memory because even the physical page allocator, discussed in the next chapter, needs to allocate memory to initialize itself. But how can the physical page allocator allocate memory to initialize itself?

To address this, a specialized allocator called the *Boot Memory Allocator* is used. It is based on the most basic of allocators, a *First Fit* allocator, which uses a bitmap to represent memory [Tan01] instead of linked lists of free blocks. If a bit is 1, the page is allocated, and if the bit is 0, it is unallocated. To satisfy allocations of sizes smaller than a page, the allocator records the *Page Frame Number (PFN)* of the last allocation and the 0 set the allocation ended at. Subsequent small allocations are merged together and stored on the same page.

The reader may ask why this allocator is not used for the running system. One compelling reason is that, although the *First Fit* allocator does not suffer badly from fragmentation [JW98], memory frequently has to be linearly searched to satisfy an allocation. Because this is examining bitmaps, it gets very expensive, especially because the *First Fit* algorithm tends to leave many small free blocks at the beginning of physical memory that still get scanned for large allocations, thus making the process very wasteful [WJNB95].

There are two very similar but distinct APIs for the allocator. One is for UMA architectures listed in Table 5.1, and the other is for NUMA listed in Table 5.2. The principal difference is that the NUMA API must be supplied with the node affected by the operation, but, because the callers of these APIs exist in the architecture-dependent layer, it is not a significant problem.

This chapter begins with a description of the structure that the allocator uses to describe the physical memory available for each node. I then illustrate how the limits of physical memory and the sizes of each zone are discovered before talking about how the information is used to initialize the boot memory allocator structures. The allocation and free routines are then discussed before fully talking about how the boot memory allocator is retired.

<pre>unsigned long init_bootmem(unsigned long start, unsigned long page)</pre> Initializes the memory between 0 and the PFN page. The beginning of usable memory is at the PFN start.
<pre>void reserve_bootmem(unsigned long addr, unsigned long size)</pre> Marks the pages between the address addr and addr+size reserved. Requests to partially reserve a page will result in the full page being reserved.
<pre>void free_bootmem(unsigned long addr, unsigned long size)</pre> Marks the pages between the address addr and addr+size as free.
<pre>void * alloc_bootmem(unsigned long size)</pre> Allocates size number of bytes from ZONE_NORMAL. The allocation will be aligned to the L1 hardware cache to get the maximum benefit from the hardware cache.
<pre>void * alloc_bootmem_low(unsigned long size)</pre> Allocates size number of bytes from ZONE_DMA. The allocation will be aligned to the L1 hardware cache.
<pre>void * alloc_bootmem_pages(unsigned long size)</pre> Allocates size number of bytes from ZONE_NORMAL aligned on a page size so that full pages will be returned to the caller.
<pre>void * alloc_bootmem_low_pages(unsigned long size)</pre> Allocates size number of bytes from ZONE_DMA aligned on a page size so that full pages will be returned to the caller.
<pre>unsigned long bootmem_bootmap_pages(unsigned long pages)</pre> Calculates the number of pages required to store a bitmap representing the allocation state of pages number of pages.
<pre>unsigned long free_all_bootmem()</pre> Used at the boot allocator end of life. It cycles through all pages in the bitmap. For each one that is free, the <code>bits</code> are cleared, and the page is freed to the physical page allocator (see next chapter) so that the runtime allocator can set up its free lists.

Table 5.1. Boot Memory Allocator API for UMA Architectures

5.1 Representing the Boot Map

A `bootmem_data` struct exists for each node of memory in the system. It contains the information needed for the boot memory allocator to allocate memory for a node, such as the bitmap representing allocated pages and where the memory is

```

unsigned long init_bootmem_node(pg_data_t *pgdat, unsigned long
freepfn, unsigned long startpfn, unsigned long endpfn)
    For use with NUMA architectures. It initializes the memory between PFNs
startpfn and endpfn with the %d usable PFN at freepfn. After it is initial-
ized, the pgdat node is inserted into the pgdat_list.

void reserve_bootmem_node(pg_data_t *pgdat, unsigned long physaddr,
unsigned long size)
    Marks the pages between the address addr and addr+size on the speci%d
node pgdat reserved. Requests to partially reserve a page will result in the full
page being reserved.

void free_bootmem_node(pg_data_t *pgdat, unsigned long physaddr,
unsigned long size)
    Marks the pages between the address addr and addr+size on the speci%d
node pgdat free.

void * alloc_bootmem_node(pg_data_t *pgdat, unsigned long size)
    Allocates size number of bytes from ZONE_NORMAL on the speci%d node
pgdat. The allocation will be aligned to the L1 hardware cache to get the
maximum bene%dfrom the hardware cache.

void * alloc_bootmem_pages_node(pg_data_t *pgdat, unsigned long
size)
    Allocates size number of bytes from ZONE_DMA on the speci%d node pgdat
aligned on a page size so that full pages will be returned to the caller.

void * alloc_bootmem_low_pages_node(pg_data_t *pgdat, unsigned long
size)
    Allocates size number of bytes from ZONE_DMA on the speci%d node pgdat
aligned on a page size so that full pages will be returned to the caller.

unsigned long free_all_bootmem_node(pg_data_t *pgdat)
    Used at the boot allocator end of life. It cycles through all pages in the bitmap
for the speci%d node. For each one that is free, the page %ds are cleared, and
the page is freed to the physical page allocator (see next chapter) so that the
runtime allocator can set up its free lists.

```

Table 5.2. Boot Memory Allocator API for NUMA Architectures

located. It is declared as follows in `<linux/bootmem.h>`:

```

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;

```

The fields of this struct are as follows:

- node_boot_start** This is the starting physical address of the represented block.
- node_low_pfn** This is the end physical address, in other words, the end of the ZONE_NORMAL this node represents.
- node_bootmem_map** This is the location of the bitmap representing allocated or free pages with each bit.
- last_offset** This is the offset within the the page of the end of the last allocation. If 0, the page used is full.
- last_pos** This is the the PFN of the page used with the last allocation. By using this with the **last_offset** field, a test can be made to see if allocations can be merged with the page used for the last allocation rather than using up a full new page.

5.2 Initializing the Boot Memory Allocator

Each architecture is required to supply a `setup_arch()` function, which, among other tasks, is responsible for acquiring the necessary parameters to initialize the boot memory allocator.

Each architecture has its own function to get the necessary parameters. On the x86, it is called `setup_memory()` as discussed in Section 2.2.2, but, on other architectures such as MIPS or Sparc, it is called `bootmem_init()` or, in the case of the PPC, `do_init_bootmem()`. Regardless of the architecture, the tasks are essentially the same. The parameters it calculates are the following:

- min_low_pfn** This is the lowest PFN that is available in the system.
- max_low_pfn** This is the highest PFN that may be addressed by low memory (ZONE_NORMAL).
- highstart_pfn** This is the PFN of the beginning of high memory (ZONE_HIGHMEM).
- highend_pfn** This is the last PFN in high memory.
- max_pfn** Finally, this is the last PFN available to the system.

5.3 Initializing bootmem_data

After the limits of usable physical memory are discovered by `setup_memory()`, one of two boot memory initialization functions is selected and provided with the start and end PFN for the node to be initialized. `init_bootmem()`, which initializes

parameter as listed in Table 5.2. They are called `alloc_bootmem_node()`, `alloc_bootmem_pages_node()` and `alloc_bootmem_low_pages_node()`. All of these macros call `__alloc_bootmem_node()` with different parameters.

The parameters to `__alloc_bootmem()` and `__alloc_bootmem_node()` are essentially the same. They are the following:

pgdat This is the node to allocate from. It is omitted in the UMA case because it is assumed to be `contig_page_data`.

size This is the size in bytes of the requested allocation.

align This is the number of bytes that the request should be aligned to. For small allocations, they are aligned to `SMP_CACHE_BYTES`, which, on the x86, will align to the L1 hardware cache.

goal This is the preferred starting address to begin allocating from. The low functions will start from physical address 0 whereas the others will begin from `MAX_DMA_ADDRESS`, which is the maximum address DMA transfers may be made from on this architecture.

The core function for all the allocation APIs is `__alloc_bootmem_core()`. It is a large function, but with simple steps that can be broken down. The function linearly scans memory starting from the `goal` address for a block of memory large enough to satisfy the allocation. With the API, this address will either be 0 for DMA-friendly allocations or `MAX_DMA_ADDRESS` otherwise.

The clever part, and the main bulk of the function, deals with deciding if this new allocation can be merged with the previous one. It may be merged if the following conditions hold:

- ❌ The page used for the previous allocation (`bootmem_data->pos`) is adjacent to the page found for this allocation.

- ❌ The previous page has some free space in it (`bootmem_data->offset != 0`).

- ❌ The alignment is less than `PAGE_SIZE`.

Regardless of whether the allocations may be merged or not, the `pos` and `offset` fields will be updated to show the last page used for allocating and how much of the last page was used. If the last page was fully used, the `offset` is 0.

5.5 Freeing Memory

In contrast to the allocation functions, only two free functions are provided, which are `free_bootmem()` for UMA and `free_bootmem_node()` for NUMA. They both call `free_bootmem_core()`, and the only difference is that a `pgdat` is supplied with NUMA.

The core function is relatively simple in comparison to the rest of the allocator. For each *full* page affected by the free, the corresponding bit in the bitmap is set to 0. If it already was 0, `BUG()` is called to show a double-free occurred. `BUG()`

parameters. The core function is simple in principle and performs the following tasks:

- For all unallocated pages known to the allocator for this node, it does the following:
 - Clear the `PG_reserved` in its struct page.
 - Set the count to 1.
 - Call `__free_pages()` so that the buddy allocator (discussed in the next chapter) can build its free lists.
- Free all pages used for the bitmap and give them to the buddy allocator.

At this stage, the buddy allocator now has control of all the pages in low memory, which leaves only the high memory pages. After `free_all_bootmem()` returns, it counts the number of reserved pages for accounting purposes. The remainder of the `free_pages_init()` function is responsible for the high memory pages. However, at this point, it should be clear how the global `mem_map` array is allocated and initialized and how the pages are given to the main allocator. The basic flow used to initialize pages in low memory in a single node system is shown in Figure 5.3.

After `free_all_bootmem()` returns, all the pages in `ZONE_NORMAL` have been given to the buddy allocator. To initialize the high memory pages, `free_pages_init()` calls `one_highpage_init()` for every page between `highstart_pfn` and `highend_pfn`. `one_highpage_init()` simply clears the `PG_reserved`, sets the `PG_highmem`, sets the count to 1 and calls `__free_pages()` to release it to the buddy allocator in the same manner `free_all_bootmem_core()` did.

At this point, the boot memory allocator is no longer required, and the buddy allocator is the main physical page allocator for the system. An interesting feature to note is that not only is the data for the boot allocator removed, but also all code that was used to bootstrap the system. All initialization functions that are required only during system start-up are marked `__init`, such as the following:

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

All of these functions are placed together in the `.init` section by the linker. On the x86, the function `free_initmem()` walks through all pages from `__init_begin` to `__init_end` and frees up the pages to the buddy allocator. With this method, Linux can free up a considerable amount of memory that is used by bootstrapping code that is no longer required. For example, 27 pages were freed while booting the kernel running on the machine this document was composed on.

5.7 What's New in 2.6

The boot memory allocator has not changed significantly since 2.4 and is mainly concerned with optimizations and some minor NUMA-related modifications. The

Not optimization is the addition of a `last_success` field to the `bootmem_data_t` struct. As the name suggests, it keeps track of the location of the last successful allocation to reduce search times. If an address is freed before `last_success`, it will be changed to the freed location.

The second optimization is also related to the linear search. When searching for a free page, 2.4 tests every bit, which is expensive. 2.6 instead tests if a block of `BITS_PER_LONG` is all ones. If it is not, it will test each of the bits individually in that block. To help the linear search, nodes are ordered in order of their physical addresses by `init_bootmem()`.

The last change is related to NUMA and contiguous architectures. Contiguous architectures now define their own `init_bootmem()` function and any architecture can optionally define their own `reserve_bootmem()` function.

Physical Page Allocation

This chapter describes how physical pages are managed and allocated in Linux. The principal algorithm used is the *Binary Buddy Allocator*, devised by Knowlton [Kno65] and further described by Knuth [Knu68]. The binary buddy allocator has been shown to be extremely fast in comparison to other allocators [KB85].

This is an allocation scheme that combines a normal power-of-two allocator with free buffer coalescing [Vah96], and the basic concept behind it is quite simple. Memory is broken up into large blocks of pages where each block is a power of two number of pages. If a block of the desired size is not available, a large block is broken up in half, and the two blocks are *buddies* to each other. One half is used for the allocation, and the other is free. The blocks are continuously halved as necessary until a block of the desired size is available. When a block is later freed, the buddy is examined, and the two are coalesced if it is free.

This chapter will begin with describing how Linux remembers what blocks of memory are free. After that the methods for allocating and freeing pages will be discussed in detail. The subsequent section will cover the ~~ways~~ that affect the allocator behavior and ~~finally~~ the problem of fragmentation and how the allocator handles it.

6.1 Managing Free Blocks

As stated, the allocator maintains blocks of free pages where each block is a power of two number of pages. The exponent for the power of two-sized block is referred to as the *order*. An array of `free_area_t` structs are maintained for each order that points to a linked list of blocks of pages that are free as indicated by Figure 6.1.

Hence, the 0th element of the array will point to a list of free page blocks of size 2^0 or 1 page, the 1st element will be a list of 2^1 (2) pages up to $2^{\text{MAX\_ORDER}}$ number of pages, where the `MAX_ORDER` is currently defined as 10. This eliminates the chance that a larger block will be split to satisfy a request where a smaller block would have succeeded. The page blocks are maintained on a linear linked list using `page → list`.

Each zone has a `free_area_t` struct array called `free_area[MAX_ORDER]`. It is declared in `<linux/mm.h>` as follows:

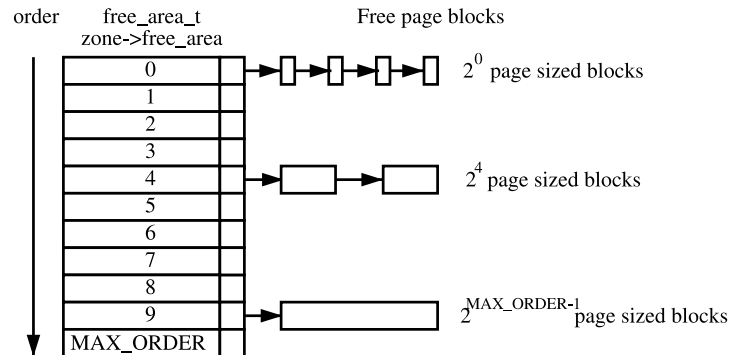


Figure 6.1. Free Page Block Management

```

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

```

The fields in this struct are as follows:

free_list A linked list of free page blocks

map A bitmap representing the state of a pair of buddies

Linux saves memory by only using one bit instead of two to represent each pair of buddies. Each time a buddy is allocated or freed, the bit representing the pair of buddies is toggled so that the bit is zero if the pair of pages are both free or both full and 1 if only one buddy is in use. To toggle the correct bit, the macro `MARK_USED()` in `page_alloc.c` is used, which is declared as follows:

```

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

```

`index` is the index of the page within the global `mem_map` array. By shifting it right by `1+order` bits, the bit within the map representing the pair of buddies is revealed.

6.2 Allocating Pages

Linux provides a quite sizable API for the allocation of page frames. All of them take a `gfp_mask` as a parameter, which is a set of flags that determine how the allocator will behave. The flags are discussed in Section 6.4.

As shown in Figure 6.2, the allocation API functions all use the core function `__alloc_pages()`, but the APIs exist so that the correct node and zone will be chosen. Different users will require different zones, such as `ZONE_DMA` for certain device drivers or `ZONE_NORMAL` for disk buffers, and callers should not have to be

<code>struct page * alloc_page(unsigned int gfp_mask)</code>	Allocates a single page and returns a struct address.
<code>struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)</code>	Allocates 2^{order} number of pages and returns a struct page.
<code>unsigned long get_free_page(unsigned int gfp_mask)</code>	Allocates a single page, zeros it, and returns a virtual address.
<code>unsigned long __get_free_page(unsigned int gfp_mask)</code>	Allocates a single page and returns a virtual address.
<code>unsigned long __get_free_pages(unsigned int gfp_mask, unsigned int order)</code>	Allocates 2^{order} number of pages and returns a virtual address.
<code>struct page * __get_dma_pages(unsigned int gfp_mask, unsigned int order)</code>	Allocates 2^{order} number of pages from the DMA zone and returns a struct page.

Table 6.1. Physical Pages Allocation API

`build_zonelists()`, but generally `ZONE_HIGHMEM` will fall back to `ZONE_NORMAL` and that in turn will fall back to `ZONE_DMA`. If number of free pages reaches the `pages_low` watermark, it will wake `kswapd` to begin freeing up pages from zones, and, if memory is extremely tight, the caller will do the work of `kswapd` itself.

After the zone has ~~fully~~ been decided on, the function `rmqueue()` is called to allocate the block of pages or split higher level blocks if one of the appropriate size is not available.

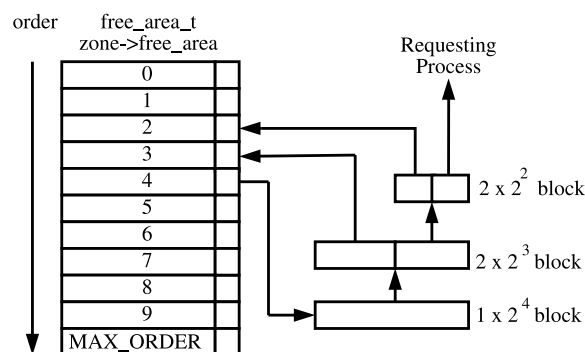


Figure 6.3. Allocating Physical Pages

because, if the bit is 0, it means both buddies are either both free or both allocated. If both are free, they may be merged.

Calculating the address of the buddy is a well known concept [Knu68]. Because the allocations are always in blocks of size 2^k , the address of the block, or at least its offset within `zone_mem_map`, will also be a power of 2^k . The end result is that there will always be at least k number of zeros to the right of the address. To get the address of the buddy, the k th bit from the right is examined. If it is 0, the buddy will have this bit flipped. To get this bit, Linux creates a `mask`, which is calculated as

$$\text{mask} = (1 \ll k)$$

The mask we are interested in is

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

Linux takes a shortcut in calculating this by noting that

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

After the buddy is merged, it is removed from the free list, and the newly coalesced pair moves to the next higher order to see if it may also be merged.

6.4 Get Free Page (GFP) Flags

A persistent concept through the whole VM is the *Get Free Page (GFP)* flags. These flags determine how the allocator and `kswapd` will behave for the allocation and freeing of pages. For example, an interrupt handler may not sleep, so it will *not* have the `__GFP_WAIT` flag set because this flag indicates the caller may sleep. There are three sets of GFP flags, which are all defined in `<linux/mm.h>`.

The first of the three is the set of zone modifiers listed in Table 6.3. These flags indicate that the caller must try to allocate from a particular zone. `ZONE_NORMAL` does not have a zone modifier. This is because the zone modifier 0 is used as an offset within an array, and 0 implicitly means allocate from `ZONE_NORMAL`.

Flag	Description
<code>__GFP_DMA</code>	Allocate from <code>ZONE_DMA</code> if possible.
<code>__GFP_HIGHMEM</code>	Allocate from <code>ZONE_HIGHMEM</code> if possible.
<code>GFP_DMA</code>	Act as alias for <code>__GFP_DMA</code> .

Table 6.3. Low-Level GFP Flags Affecting Zone Allocation

The next flags are action modifiers listed in Table 6.4. They change the behavior of the VM and what the calling process may do. The low-level flags on their own are too primitive to be easily used.

Flag	Description
<code>__GFP_WAIT</code>	Indicates that the caller is not high priority and can sleep or reschedule.
<code>__GFP_HIGH</code>	Used by a high priority or kernel process. Kernel 2.2.x used it to determine if a process could access emergency pools of memory. In 2.4.x kernels, it does not appear to be used.
<code>__GFP_IO</code>	Indicates that the caller can perform low-level I/O. In 2.4.x, the main effect this has is determining if <code>try_to_free_buffers()</code> can flush buffers. It is used by at least one journaled filesystem.
<code>__GFP_HIGHIO</code>	Determines that I/O can be performed on pages mapped in high memory. It is only used in <code>try_to_free_buffers()</code> .
<code>__GFP_FS</code>	Indicates if the caller can make calls to the filesystem layer. This is used when the caller is filesystem related, the buffer cache, for instance, and wants to avoid recursively calling itself.

Table 6.4. Low-Level GFP Flags Affecting Allocator Behavior

It is difficult to know what the correct combinations are for each instance, so a few high-level combinations are defined and listed in Table 6.5. For clarity the `__GFP_` is removed from the table combinations, so the `__GFP_HIGH` flag will read as `HIGH` in the table. The combinations to form the high-level flags are listed in Table 6.6. To help understand this, take `GFP_ATOMIC` as an example. It has only the `__GFP_HIGH` flag set. This means it is high priority, will use emergency pools (if they exist), but it will not sleep, perform I/O, or access the filesystem. This flag would be used by an interrupt handler, for example.

Flag	Low-Level Flag Combination
<code>GFP_ATOMIC</code>	<code>HIGH</code>
<code>GFP_NOIO</code>	<code>HIGH & WAIT</code>
<code>GFP_NOHIGHIO</code>	<code>HIGH & WAIT & NO</code>
<code>GFP_NOFS</code>	<code>HIGH & WAIT & NO & HIGHIO</code>
<code>GFP_KERNEL</code>	<code>HIGH & WAIT & NO & HIGHIO & FS</code>
<code>GFP_NFS</code>	<code>HIGH & WAIT & NO & HIGHIO & FS</code>
<code>GFP_USER</code>	<code>WAIT & NO & HIGHIO & FS</code>
<code>GFP_HIGHUSER</code>	<code>WAIT & NO & HIGHIO & FS & HIGHMEM</code>
<code>GFP_KSWAPD</code>	<code>WAIT & NO & HIGHIO & FS</code>

Table 6.5. Low-Level GFP Flag Combinations for High-Level Use

6.5 Process Flags

A process may also set flags in the `task_struct`, which affects allocator behavior. The full list of process flags is defined in `<linux/sched.h>`, but only the ones affecting VM behavior are listed in Table 6.7.

Flag	Description
GFP_ATOMIC	This flag is used whenever the caller cannot sleep and must be serviced if at all possible. Any interrupt handler that requires memory must use this flag to avoid sleeping or performing I/O. Many subsystems during init will use this system, such as <code>buffer_init()</code> and <code>inode_init()</code> .
GFP_NOIO	This is used by callers who are already performing an I/O-related function. For example, when the loopback device is trying to get a page for a buffer head, it uses this flag to make sure it will not perform some action that would result in more I/O. In fact, it appears the flag was introduced specifically to avoid a deadlock in the loopback device.
GFP_NOHIGHIO	This is only used in one place in <code>alloc_bounce_page()</code> during the creating of a bounce buffer for I/O in high memory.
GFP_NOFS	This is only used by the buffer cache and filesystems to make sure they do not recursively call themselves by accident.
GFP_KERNEL	This is the most liberal of the combined flags. It indicates that the caller is free to do whatever it pleases. Strictly speaking the difference between this flag and <code>GFP_USER</code> is that this could use emergency pools of pages, but that is a no-op on 2.4.x kernels.
GFP_USER	This is another flag of historical significance. In the 2.2.x series, an allocation was given a LOW, MEDIUM or HIGH priority. If memory was tight, a request with <code>GFP_USER</code> (low) would fail whereas the others would keep trying. Now it has no significance and is not treated any differently to <code>GFP_KERNEL</code> .
GFP_HIGHUSER	This flag indicates that the allocator should allocate from <code>ZONE_HIGHMEM</code> if possible. It is used when the page is allocated on behalf of a user process.
GFP_NFS	This flag is defunct. In the 2.0.x series, this flag determined what the reserved page size was. Normally, 20 free pages were reserved. If this flag was set, only 10 would be reserved. Now it is not treated differently anywhere.
GFP_KSWAPD	This has more historical significance. In reality, this is not treated any differently to <code>GFP_KERNEL</code> .

Table 6.6. High-Level GFP Flags Affecting Allocator Behavior

6.6 Avoiding Fragmentation

One important problem that must be addressed with any allocator is the problem of internal and external fragmentation. External fragmentation is the inability to service a request because the available memory exists only in small blocks. Internal fragmentation is defined as the wasted space where a large block had to be assigned to service a small request. In Linux, external fragmentation is not a serious problem because large requests for contiguous pages are rare, and usually `vmalloc()` (see

Flag	Description
PF_MEMALLOC	This flag sets the process as a memory allocator. <code>kswapd</code> sets this flag, and it is set for any process that is about to be killed by the <i>OOM</i> killer, which is discussed in Chapter 13. It tells the buddy allocator to ignore zone watermarks and assigns the pages if at all possible.
PF_MEMDIE	This is set by the <i>OOM</i> killer and functions the same as the <code>PF_MEMALLOC</code> flag by telling the page allocator to give pages if at all possible because the process is about to die.
PF_FREE_PAGES	This is set when the buddy allocator calls <code>try_to_free_pages()</code> itself to indicate that free pages should be reserved for the calling process in <code>__free_pages_ok()</code> instead of returning to the free lists.

Table 6.7. Process Flags Affecting Allocator Behavior

Chapter 7) is sufficient to service the request. The lists of free blocks ensure that large blocks do not have to be split unnecessarily.

Internal fragmentation is the single-most serious failing of the binary buddy system. Although fragmentation is expected to be in the region of 28 percent [WJNB95], it has been shown that it can be in the region of 60 percent, in comparison to just 1 percent with the *best-fit* allocator [JW98]. It has also been shown that using variations of the buddy system will not help the situation significantly [PN77]. To address this problem, Linux uses a *slab allocator* [Bon94] to carve up pages into small blocks of memory for allocation [Tan01], which is discussed further in Chapter 8. With this combination of allocators, the kernel can ensure that the amount of memory wasted due to internal fragmentation is kept to a minimum.

6.7 What's New in 2.6

Allocating Pages The most noticeable difference seems cosmetic at first. The function `alloc_pages()` is now a macro and is defined in `<linux/gfp.h>` instead of a function defined in `<linux/mm.h>`. The new layout is still very recognizable, and the main difference is a subtle, but important one. In 2.4, specific code was dedicated to selecting the correct node to allocate from based on the running CPU, but 2.6 removes this distinction between NUMA and UMA architectures.

In 2.6, the function `alloc_pages()` calls `numa_node_id()` to return the logical ID of the node associated with the current running CPU. This NID is passed to `_alloc_pages()`, which calls `NODE_DATA()` with the NID as a parameter. On UMA architectures, this will unconditionally result in `contig_page_data` being returned, but NUMA architectures instead set up an array that `NODE_DATA()` uses NID as an offset into. In other words, architectures are responsible for setting up a CPU ID to NUMA memory node mapping. This is effectively still a node-local allocation policy as is used in 2.4, but it is a lot more clearly defined.

Per-CPU Page Lists The most important addition to the page allocation is the addition of the per-cpu lists, ~~not~~ discussed in Section 2.8.

In 2.4, a page allocation requires an interrupt-safe spinlock to be held while the allocation takes place. In 2.6, pages are allocated from a `struct per_cpu_pageset` by `buffered_rmqueue()`. If the low watermark (`per_cpu_pageset → low`) has not been reached, the pages will be allocated from the pageset with no requirement for a spinlock to be held. After the low watermark is reached, a large number of pages will be allocated in bulk with the interrupt-safe spinlock held, added to the per-cpu list and then one returned to the caller.

Higher order allocations, which are relatively rare, still require the interrupt-safe spinlock to be held, and there will be no delay in the splits or coalescing. With 0 order allocations, splits will be delayed until the low watermark is reached in the per-cpu set, and coalescing will be delayed until the high watermark is reached.

However, strictly speaking, this is not a lazy buddy algorithm [BL89]. Although pagesets introduce a merging delay for order-0 allocations, it is a side effect rather than an intended feature, and no method is available to drain the pagesets and merge the buddies. In other words, despite the per-cpu and new accounting code that bulks up the amount of code in `mm/page_alloc.c`, the core of the buddy algorithm remains the same as it was in 2.4.

The implication of this change is straightforward; the number of times the spinlock protecting the buddy lists must be acquired is reduced. Higher order allocations are relatively rare in Linux, so the optimization is for the common case. This change will be noticeable on a large number of CPU machines, but will make little difference to single CPUs. There are a few issues with pagesets, but they are not recognized as a serious problem. The ~~first~~ issue is that high-order allocations may fail if the pagesets hold order-0 pages that would normally be merged into higher order contiguous blocks. The second is that an order-0 allocation may fail if memory is low, the current CPU pageset is empty and other CPUs' pagesets are full because no mechanism exists for reclaiming pages from remote pagesets. The last potential problem is that buddies of newly freed pages could exist in other pagesets, leading to possible fragmentation problems.

Freeing Pages Two new API functions have been introduced for the freeing of pages called `free_hot_page()` and `free_cold_page()`. Predictably, they determine if the freed pages are placed on the hot or cold lists in the per-cpu pagesets. However, although the `free_cold_page()` is exported and available for use, it is actually never called.

Order-0 page frees from `__free_pages()` and frees resulting from page cache releases by `__page_cache_release()` are placed on the hot list whereas higher order allocations are freed immediately with `__free_pages_ok()`. Order-0 are usually related to userspace and are the most common type of allocation and free. By keeping them local to the CPU, lock contention will be reduced because most allocations will also be of order-0.

Eventually, lists of pages must be passed to `free_pages_bulk()`, or the pageset lists would hold all free pages. This `free_pages_bulk()` function takes a list of page block allocations, the order of each block and the count number of blocks to free

from the list. There are two principal cases where this is used. The first is higher order frees passed to `__free_pages_ok()`. In this case, the page block is placed on a linked list of the specified order and a count of 1. The second case is where the high watermark is reached in the pageset for the running CPU. In this case, the pageset is passed with an order of 0 and a count of `pageset->batch`.

After the core function `__free_pages_bulk()` is reached, the mechanisms for freeing pages is very similar to the buddy lists in 2.4.

GFP Flags There are still only three zones, so the zone modifiers remain the same. However, three new GFP flags have been added that affect how hard the VM will work, or not work, to satisfy a request. The flags are the following:

- `__GFP_NOFAIL` This flag is used by a caller to indicate that the allocation should never fail and that the allocator should keep trying to allocate indefinitely.
- `__GFP_REPEAT` This flag is used by a caller to indicate that the request should try to repeat the allocation if it fails. In the current implementation, it behaves the same as `__GFP_NOFAIL`, but later the decision might be made to fail after a while.
- `__GFP_NORETRY` This flag is almost the opposite of `__GFP_NOFAIL`. It indicates that, if the allocation fails, it should just return immediately.

At time of writing, these flags are not heavily used, but they have just been introduced and are likely to be used more over time. The `__GFP_REPEAT` flag, in particular, is likely to be heavily used because blocks of code which implement this flag's behavior exist throughout the kernel.

The next GFP flag that has been introduced is an allocation modifier called `__GFP_COLD`, which is used to ensure that cold pages are allocated from the per-cpu lists. From the perspective of the VM, the only user of this flag is the function `page_cache_alloc_cold()`, which is mainly used during I/O readahead. Usually, page allocations will be taken from the hot pages list.

The last new flag is `__GFP_NO_GROW`. This is an internal flag used only by the slab allocator (discussed in Chapter 8), which aliases the flag to `SLAB_NO_GROW`. It is used to indicate when new slabs should never be allocated for a particular cache. In reality, the GFP flag has just been introduced to complement the old `SLAB_NO_GROW` flag, which is currently unused in the main kernel.

Noncontiguous Memory Allocation

It is preferable when dealing with large amounts of memory to use physically contiguous pages in memory both for cache-related and memory-access-latency reasons. Unfortunately, due to external fragmentation problems with the buddy allocator, this is not always possible. Linux provides a mechanism through `vmalloc()` where noncontiguous physical memory can be used that is contiguous in virtual memory.

An area is reserved in the virtual address space between `VMALLOC_START` and `VMALLOC_END`. The location of `VMALLOC_START` depends on the amount of available physical memory, but the region will always be at least `VMALLOC_RESERVE` in size, which on the x86 is 128MiB. The exact size of the region is discussed in Section 4.1.

The page tables in this region are adjusted as necessary to point to physical pages, which are allocated with the normal physical page allocator. This means that allocation must be a multiple of the hardware page size. Because allocations require altering the kernel page tables, there is a limitation on how much memory can be mapped with `vmalloc()` because only the virtual addresses space between `VMALLOC_START` and `VMALLOC_END` is available. As a result, `vmalloc()` is used sparingly in the core kernel. In 2.4.22, it is only used for storing the swap map information (see Chapter 11) and for loading kernel modules into memory.

This small chapter begins with a description of how the kernel tracks which areas in the `vmalloc` address space are used and how regions are allocated and freed.

7.1 Describing Virtual Memory Areas

The `vmalloc` address space is managed with a resource map allocator [Vah96]. The `struct vm_struct` is responsible for storing the base,size pairs. It is defined in `<linux/vmalloc.h>` as the following:

```
14 struct vm_struct {
15     unsigned long flags;
16     void * addr;
17     unsigned long size;
18     struct vm_struct * next;
19 };
```

A fully-paged VMA could have been used but it contains extra information that does not apply to vmalloc areas and would be wasteful. Here is a brief description of the fields in this small struct.

flags These set either to VM_ALLOC, in the case of use with vmalloc(), or VM_IOREMAP, when ioremap() is used to map high memory into the kernel virtual address space.

addr This is the starting address of the memory block.

size This is, predictably enough, the size in bytes.

next This is a pointer to the next vm_struct. They are ordered by address, and the list is protected by the vmList_lock lock.

As is clear, the areas are linked together by the next field and are ordered by address for simple searches. Each area is separated by at least one page to protect against overruns. This is illustrated by the gaps in Figure 7.1.

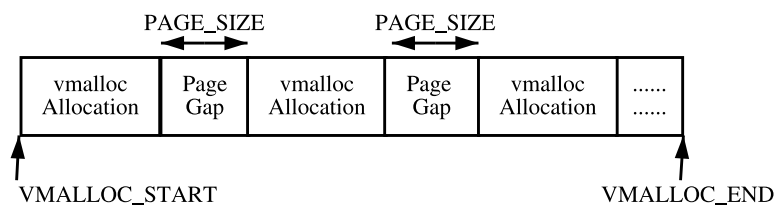


Figure 7.1. vmalloc Address Space

When the kernel wants to allocate a new area, the vm_struct list is searched linearly by the function get_vm_area(). Space for the struct is allocated with kmalloc(). When the virtual area is used for remapping an area for I/O (commonly referred to as *ioremapping*), this function will be called directly to map the requested area.

7.2 Allocating a Noncontiguous Area

The functions vmalloc(), vmalloc_dma() and vmalloc_32() are provided to allocate a memory area that is contiguous in virtual address space, as described in Table 7.1. They all take a single parameter size, which is rounded up to the next page alignment. They all return a linear address for the new allocated area.

As is clear from the call graph shown in Figure 7.2, there are two steps to allocating the area. The first step taken by get_vm_area() is to find a region large enough to store the request. It searches through a linear linked list of vm_structs and returns a new struct describing the allocated region.

The second step is to allocate the necessary PGD entries with vmalloc_area_pages(), PMD entries with alloc_area_pmd() and PTE entries with alloc_area_pte() before finally allocating the page with alloc_page().

Slab Allocator

In this chapter, the general-purpose allocator is described. It is a slab allocator that is very similar in many respects to the general kernel allocator used in Solaris [MM01]. Linux's implementation is heavily based on the ~~the~~ slab allocator paper by Bonwick [Bon94] with many improvements that bear a close resemblance to those described in his later paper [BA01]. I begin with a quick overview of the allocator, followed by a description of the different structures used before giving an in-depth tour of each task the allocator is responsible for.

The basic idea behind the slab allocator is to have caches of commonly used objects kept in an initialized state available for use by the kernel. Without an object-based allocator, the kernel will spend much of its time allocating, initializing and freeing the same object. The slab allocator aims to cache the freed object so that the basic structure is preserved between uses [Bon94].

The slab allocator consists of a variable number of caches that are linked together on a doubly linked circular list called a *cache chain*. A cache, in the context of the slab allocator, is a manager for a number of objects of a particular type, like the `mm_struct` or `fs_cache` cache, and is managed by a `struct kmem_cache_s` discussed in detail later. The caches are linked by the next ~~id~~ in the cache struct.

Each cache maintains blocks of contiguous pages in memory called *slabs* that are carved up into small chunks for the data structures and objects that the cache manages. The relationship between these different structures is illustrated in Figure 8.1.

The slab allocator has three principle aims:

- The allocation of small blocks of memory to help eliminate internal fragmentation that would be otherwise caused by the buddy system.
- The caching of commonly used objects so that the system does not waste time allocating, initializing and destroying objects. Benchmarks on Solaris showed excellent speed improvements for allocations with the slab allocator in use [Bon94].
- Better use of the hardware cache by aligning objects to the L1 or L2 caches.

To help eliminate internal fragmentation normally caused by a binary buddy allocator, two sets of caches of small memory buffers ranging from 2^5 (32) bytes to 2^{17} (131,072) bytes are maintained. One cache set is suitable for use with DMA


```

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))
    Creates a new cache and adds it to the cache chain.

int kmem_cache_reap(int gfp_mask)
    Scans at most REAP_SCANLEN caches and selects one for reaping all per-cpu
objects and free slabs from. It is called when memory is tight.

int kmem_cache_shrink(kmem_cache_t *cachep)
    This function will delete all per-cpu objects associated with a cache and delete
all slabs in the slabs_free list. It returns the number of pages freed.

void * kmem_cache_alloc(kmem_cache_t *cachep, int flags)
    Allocates a single object from the cache and returns it to the caller.

void kmem_cache_free(kmem_cache_t *cachep, void *objp)
    Frees an object and returns it to the cache.

void * kcalloc(size_t size, int flags)
    Allocates a block of memory from one of the sizes cache.

void kfree(const void *objp)
    Frees a block of memory allocated with kcalloc.

int kmem_cache_destroy(kmem_cache_t * cachep)
    Destroys all objects in all slabs and frees up all associated memory before
removing the cache from the chain.

```

Table 8.1. Slab Allocator API for Caches

```

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

```

Each of the column ~~ids~~ corresponds to a ~~id~~ in the struct `kmem_cache_s` structure. The columns listed in the previous excerpt are the following:

cache-name A human-readable name such as `%p_bind_bucket%`

num-active-objs Number of objects that are in use

total-objs How many objects are available in total including unused

obj-size The size of each object, typically quite small

num-active-slabs Number of slabs containing objects that are active

total-slabs How many slabs in total exist

num-pages-per-slab The pages required to create one slab, typically 1

If SMP is enabled like in the example excerpt, two more columns will be displayed after a colon. They refer to the per-CPU cache described in Section 8.5. The columns are the following:

limit This is the number of free objects the pool can have before half of it is given to the global free pool.

batchcount This is the number of objects allocated for the processor in a block when no objects are free.

To speed allocation and freeing of objects and slabs, they are arranged into three lists: `slabs_full`, `slabs_partial` and `slabs_free`. `slabs_full` has all of its objects in use. `slabs_partial` has free objects in it, so is a prime candidate for allocation of objects. `slabs_free` has no allocated objects, so is a prime candidate for slab destruction.

8.1.1 Cache Descriptor

All information describing a cache is stored in a `struct kmem_cache_s` declared in `mm/slab.c`. This is an extremely large struct, so it will be described in parts.

```

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 }
```

Most of these fields are of interest when allocating or freeing objects.

slabs_* These are the three lists where the slabs are stored as described in the previous section.

objsize This is the size of each object packed into the slab.

%gs These flags determine how parts of the allocator will behave when dealing with the cache. See Section 8.1.2.

num This is the number of objects contained in each slab.

spinlock This is a spinlock protecting the structure from concurrent accesses.

batchcount This is the number of objects that will be allocated in batch for the per-cpu caches as described in the previous section.

```

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
```

This block deals with fields of interest when allocating or freeing slabs from the cache.

gfporder This indicates the size of the slab in pages. Each slab consumes 2^{gfporder} pages because these are the allocation sizes that the buddy allocator provides.

%gs The GFP flags used when calling the buddy allocator to allocate pages are stored here. See Section 6.4 for a full list.

colour Each slab stores objects in different cache lines if possible. Cache coloring will be further discussed in Section 8.1.5.

colour_o This is the byte alignment to keep slabs at. For example, slabs for the size-X caches are aligned on the L1 cache.

colour_next This is the next colour line to use. This value wraps back to 0 when it reaches colour;

growing This `%s` is set to indicate if the cache is growing or not. If it is, it is much less likely that this cache will be selected to reap free slabs under memory pressure.

dyns These are the dynamic `%s`s that change during the cache lifetime. See Section 8.1.3.

ctor A complex object has the option of providing a constructor function to be called to initialize each new object. This is a pointer to that function and may be NULL.

dtor This is the complementing object destructor and may be NULL.

failures This `%d` is not used anywhere in the code other than being initialized to 0.

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

These are set during cache creation.

name This is the human-readable name of the cache.

next This is the next cache on the cache chain.

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

cpudata This is the per-cpu data and is discussed further in Section 8.5.

```
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 };
```

These `%s`s are only available if the `CONFIG_SLAB_DEBUG` option is set during compile time. They are all bean counters and not of general interest. The statistics

for `/proc/slabinfo` are calculated when the proc entry is read by another process by examining every slab used by each cache rather than relying on these fields to be available.

`num_active` The current number of active objects in the cache is stored here.

`num_allocations` A running total of the number of objects that have been allocated on this cache is stored in this field.

`high_mark` This is the highest value `num_active` has had to date.

`grown` This is the number of times `kmem_cache_grow()` has been called.

`reaped` The number of times this cache has been reaped is kept here.

`errors` This field is never used.

`allochit` This is the total number of times an allocation has used the per-cpu cache.

`allocmiss` To complement `allochit`, this is the number of times an allocation has missed the per-cpu cache.

`freehit` This is the number of times a free was placed on a per-cpu cache.

`freemiss` This is the number of times an object was freed and placed on the global pool.

8.1.2 Cache Static Flags

A number of flags are set at cache creation time that remain the same for the lifetime of the cache. They affect how the slab is structured and how objects are stored within it. All the flags are stored in a bitmask in the `flags` field of the cache descriptor. The full list of possible flags that may be used are declared in `<linux/slab.h>`.

There are three principle sets. The first set is internal flags, which are set only by the slab allocator and are listed in Table 8.2. The only relevant flag in the set is the `CFGFS_OFF_SLAB` flag, which determines where the slab descriptor is stored.

Flag	Description
<code>CFGFS_OFF_SLAB</code>	Indicates that the slab managers for this cache are kept off-slab. This is discussed further in Section 8.2.1.
<code>CFLGS_OPTIMIZE</code>	This flag is only set and never used.

Table 8.2. Internal Cache Static Flags

The second set is set by the cache creator, and these flags determine how the allocator treats the slab and how objects are stored. They are listed in Table 8.3.

Flag	Description
SLAB_HWCACHE_ALIGN	Aligns the objects to the L1 CPU cache.
SLAB_MUST_HWCACHE_ALIGN	Forces alignment to the L1 CPU cache even if it is very wasteful or slab debugging is enabled.
SLAB_NO_REAP	Never reap slabs in this cache.
SLAB_CACHE_DMA	Allocates slabs with memory from ZONE_DMA.

Table 8.3. Cache Static Flags Set by Caller

The last flags are only available if the compile option `CONFIG_SLAB_DEBUG` is set; they are listed in Table 8.4. They determine what additional checks will be made to slabs and objects and are primarily of interest only when new caches are being developed.

Flag	Description
SLAB_DEBUG_FREE	Perform expensive checks on free
SLAB_DEBUG_INITIAL	On free, call the constructor as a verification to ensure the object is still initialized correctly
SLAB_RED_ZONE	This places a marker at either end of objects to trap overflows
SLAB_POISON	Poison objects with a known pattern for trapping changes made to objects not allocated or initialized

Table 8.4. Cache Static Debug Flags

To prevent callers from using the wrong flags, a `CREATE_MASK` is defined in `mm/slab.c` that consists of all the allowable flags. When a cache is being created, the requested flags are compared against the `CREATE_MASK` and reported as a bug if invalid flags are used.

8.1.3 Cache Dynamic Flags

The `dflags` field has only one flag, `DFLGS_GROWN`, but it is important. The flag is set during `kmem_cache_grow()` so that `kmem_cache_reap()` will be unlikely to choose the cache for reaping. When the function does reap a cache with this flag set, it skips the cache and removes the flag.

8.1.4 Cache Allocation Flags

These flags, listed in Table 8.5, correspond to the GFP page allocation options for allocating pages for slabs. Callers sometimes call with either `SLAB_*` or `GFP_*` flags, but they really should use only `SLAB_*` flags. They correspond directly to the flags described in Section 6.4 so will not be discussed in detail here. It is presumed that the existence of these flags is for clarity and in case the slab allocator needs to

behave differently in response to a particular flag. However, in reality, there is no difference.

Flag	Description
SLAB_ATOMIC	Equivalent to GFP_ATOMIC
SLAB_DMA	Equivalent to GFP_DMA
SLAB_KERNEL	Equivalent to GFP_KERNEL
SLAB_NFS	Equivalent to GFP_NFS
SLAB_NOFS	Equivalent to GFP_NOFS
SLAB_NOHIGHIO	Equivalent to GFP_NOHIGHIO
SLAB_NOIO	Equivalent to GFP_NOIO
SLAB_USER	Equivalent to GFP_USER

Table 8.5. Cache Allocation Flags

A very small number of flags, listed in Table 8.6, may be passed to constructor and destructor functions.

Flag	Description
SLAB_CTOR_CONSTRUCTOR	Set if the function is being called as a constructor for caches that use the same function as a constructor and a destructor.
SLAB_CTOR_ATOMIC	Indicates that the constructor may not sleep.
SLAB_CTOR_VERIFY	Indicates that the constructor should just verify that the object is initialized correctly.

Table 8.6. Cache Constructor Flags

8.1.5 Cache Coloring

To use the hardware cache better, the slab allocator will offset objects in different slabs by different amounts depending on the amount of space left over in the slab. The offset is in units of `BYTES_PER_WORD` unless `SLAB_HWCACHE_ALIGN` is set, in which case it is aligned to blocks of `L1_CACHE_BYTES` for alignment to the L1 hardware cache.

During cache creation, how many objects can fit on a slab (see Section 8.2.7) and how many bytes would be wasted are calculated. Based on wastage, two figures are calculated for the cache descriptor:

`colour` This is the number of different offsets that can be used.

`colour_offset` This is the multiple to offset each object in the slab.

With the objects of set, they will use different lines on the associative hardware cache. Therefore, objects from slabs are less likely to overwrite each other in memory.

The result of this is best explained by an example. Let us say that `s_mem` (the address of the first object) on the slab is 0 for convenience, that 100 bytes are wasted on the slab and alignment is to be at 32 bytes to the L1 Hardware Cache on a Pentium II.

In this scenario, the first slab created will have its objects start at 0. The second will start at 32, the third at 64, and the fourth at 96, and the fifth will start back at 0. With this, objects from each of the slabs will not hit the same hardware cache line on the CPU. The value of `colour` is 3 and `colour_off` is 32.

8.1.6 Cache Creation

The function `kmem_cache_create()` is responsible for creating new caches and adding them to the cache chain. The tasks that are taken to create a cache are the following:

- % Perform basic sanity checks for bad usage.
- % Perform debugging checks if `CONFIG_SLAB_DEBUG` is set.
- % Allocate a `kmem_cache_t` from the `cache_cache` slab cache.
- % Align the object size to the word size.
- % Calculate how many objects will fit on a slab.
- % Align the object size to the hardware cache.
- % Calculate color offsets.
- % Initialize remaining bits in the cache descriptor.
- % Add the new cache to the cache chain.

Figure 8.3 shows the call graph relevant to the creation of a cache; each function is fully described in the Code Commentary.

8.1.7 Cache Reaping

When a slab is freed, it is placed on the `slabs_free` list for future use. Caches do not automatically shrink themselves, so, when `kswapd` notices that memory is tight, it calls `kmem_cache_reap()` to free some memory. This function is responsible for selecting a cache that will be required to shrink its memory usage. It is worth noting that cache reaping does not take into account what memory node or zone is under pressure. This means that, with a NUMA or high memory machine, it is possible the kernel will spend a lot of time freeing memory from regions that are under no memory pressure, but this is not a problem for architectures like the x86, which has only one bank of memory.

The reader will note that, given the slab manager or objects within the slab, there does not appear to be an obvious way to determine what slab or cache they belong to. This is addressed by using the `list` field in the `struct page` that makes up the cache. `SET_PAGE_CACHE()` and `SET_PAGE_SLAB()` use the `next` and `prev` fields on the `page`→`list` to track what cache and slab an object belongs to. To get the descriptors from the page, the macros, `GET_PAGE_CACHE()` and `GET_PAGE_SLAB()`, are available. This set of relationships is illustrated in Figure 8.8.

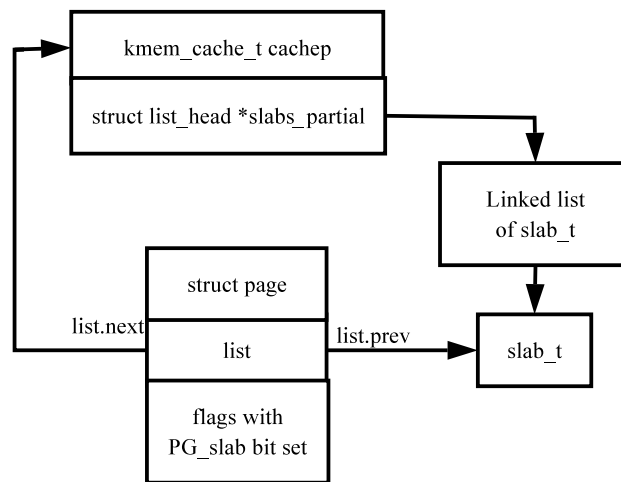


Figure 8.8. Page to Cache and Slab Relationship

The last issue is where the slab management struct is kept. Slab managers are kept either on-(`CFLGS_OFF_SLAB` set in the static flags) or off-slab. Where they are placed are determined by the size of the object during cache creation. In Figure 8.8, the `struct slab_t` could be stored at the beginning of the page frame although the diagram implies the `struct slab_t` is separate from the page frame.

8.2.1 Storing the Slab Descriptor

If the objects are larger than a threshold (512 bytes on x86), `CFLGS_OFF_SLAB` is set in the cache flags, and the *slab descriptor* is kept off-slab in one of the sizes cache (see Section 8.4). The selected sizes cache is large enough to contain the `struct slab_t`, and `kmem_cache_slabmgmt()` allocates from it as necessary. This limits the number of objects that can be stored on the slab because there is limited space for the buffers. However, that is unimportant because the objects are large, so there should not be many stored in a single slab.

Alternatively, the slab manager is reserved at the beginning of the slab. When stored on-slab, enough space is kept at the beginning of the slab to store both the `slab_t` and the `kmem_bufctl_t`, which is an array of unsigned integers. The array is responsible for tracking the index of the next free object that is available for use,

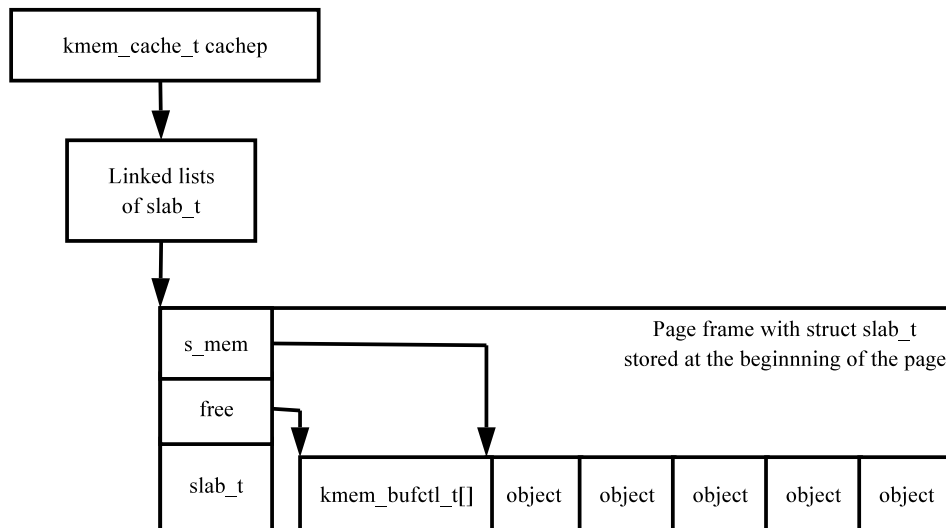


Figure 8.9. Slab With Descriptor On-Slab

which is discussed further in Section 8.2.3. The actual objects are stored after the `kmem_bufctl_t` array.

Figure 8.9 should help clarify what a slab with the descriptor on-slab looks like, and Figure 8.10 illustrates how a cache uses a sizes cache to store the slab descriptor when the descriptor is kept on-slab.

8.2.2 Slab Creation

At this point, we have seen how the cache is created, but, on creation, it is an empty cache with empty lists for its `slab_full`, `slab_partial` and `slabs_free`. New slabs are allocated to a cache by calling the function `kmem_cache_grow()` whose call graph is shown in Figure 8.11. This is frequently called *cache growing* and occurs when no objects are left in the `slabs_partial` list and when there are no slabs in `slabs_free`. The tasks it fulfills are the following:

- ❏ Perform basic sanity checks to guard against bad usage.
- ❏ Calculate color or set for objects in this slab.
- ❏ Allocate memory for the slab and acquire a slab descriptor.
- ❏ Link the pages used for the slab to the slab and cache descriptors described in Section 8.2.
- ❏ Initialize objects in the slab.
- ❏ Add the slab to the cache.

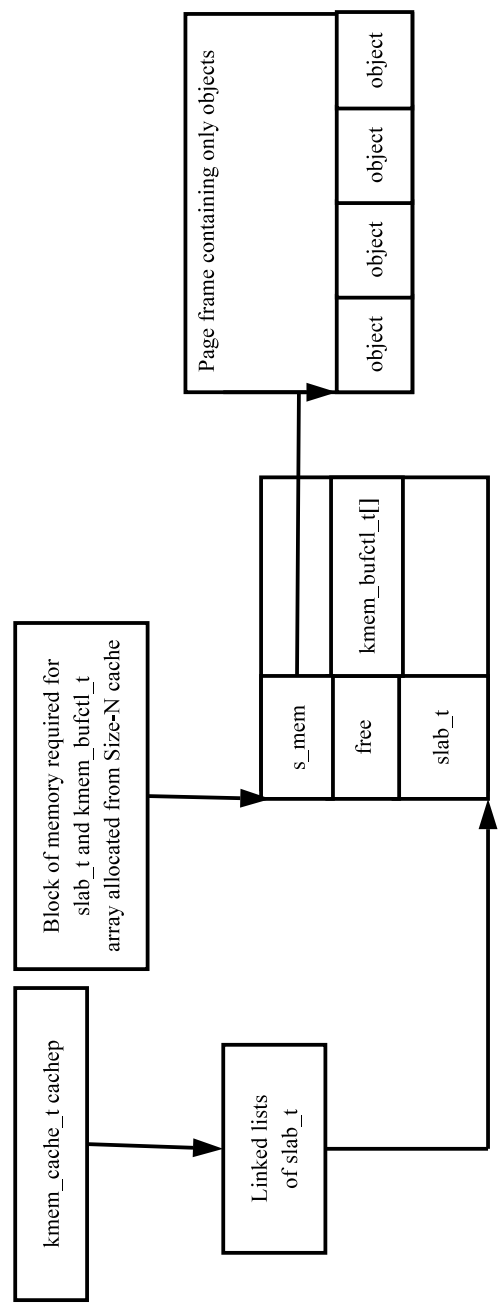


Figure 8.10. Slab With Descriptor O -Slab


```

1451     unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;
1452
1453     slab_bufctl(slabp)[objnr] = slabp->free;
1454     slabp->free = objnr;

```

The pointer `objp` is the object about to be freed, and `objnr` is its index. `kmem_bufctl_t[objnr]` is updated to point to the current value of `slabp->free`, effectively placing the object pointed to by `free` on the pseudolinked list. `slabp->free` is updated to the object being freed so that it will be the next one allocated.

8.2.7 Calculating the Number of Objects on a Slab

During cache creation, the function `kmem_cache_estimate()` is called to calculate how many objects may be stored on a single slab, which takes into account whether the slab descriptor must be stored on-slab or off-slab and the size of each `kmem_bufctl_t` needed to track if an object is free or not. It returns the number of objects that may be stored and how many bytes are wasted. The number of wasted bytes is important if cache coloring is to be used.

The calculation is quite basic and takes the following steps:

- ❧ Initialize `wastage` to be the total size of the slab, i.e., `PAGE_SIZEgfp-order`.
- ❧ Subtract the amount of space required to store the slab descriptor.
- ❧ Count up the number of objects that may be stored. Include the size of the `kmem_bufctl_t` if the slab descriptor is stored on the slab. Keep increasing the size of `i` until the slab is `mod`.
- ❧ Return the number of objects and bytes wasted.

8.2.8 Slab Destroying

When a cache is being shrunk or destroyed, the slabs will be deleted. Because the objects may have destructors, these must be called, so the tasks of this function are the following:

- ❧ If available, call the destructor for every object in the slab.
- ❧ If debugging is enabled, check the red marking and poison pattern.
- ❧ Free the pages the slab uses.

The call graph in Figure 8.13 is very simple.


```
180 #define cc_data(cachep) \
181     ((cachep)->cpudata[smp_processor_id()])
```

This will take a given cache descriptor (`cachep`) and return a pointer from the `cpucache` array (`cpudata`). The index needed is the ID of the current processor, `smp_processor_id()`.

Pointers to objects on the `cpucache` are placed immediately after the `cpucache_t` struct. This is very similar to how objects are stored after a slab descriptor.

8.5.2 Adding/Removing Objects From the Per-CPU Cache

To prevent fragmentation, objects are always added or removed from the end of the array. To add an object (`obj`) to the CPU cache (`cc`), the following block of code is used:

```
cc_entry(cc)[cc->avail++] = obj;
```

To remove an object, this block of code is used:

```
obj = cc_entry(cc)[--cc->avail];
```

There is a helper macro called `cc_entry()`, which gives a pointer to the `%i` object in the `cpucache`. It is defined as:

```
178 #define cc_entry(cpucache) \
179     ((void **)(((cpucache_t*)(cpucache))+1))
```

This takes a pointer to a `cpucache` and increments the value by the size of the `cpucache_t` descriptor and gives the `%i` object in the cache.

8.5.3 Enabling Per-CPU Caches

When a cache is created, its CPU cache has to be enabled, and memory has to be allocated for it using `kmalloc()`. The function `enable_cpucache()` is responsible for deciding what size to make the cache and for calling `kmem_tune_cpucache()` to allocate memory for it.

Obviously, a CPU cache cannot exist until after the various sizes caches have been enabled, so a global variable `g_cpucache_up` is used to prevent CPU caches from being enabled prematurely. The function `enable_all_cpucaches()` cycles through all caches in the cache chain and enables their `cpucache`.

After the CPU cache has been set up, it can be accessed without locking because a CPU will never access the wrong `cpucache`, so it is guaranteed safe access to it.

8.5.4 Updating Per-CPU Information

When the per-cpu caches have been created or changed, each CPU is signalled by an IPI. It is not sufficient to change all the values in the cache descriptor because that would lead to cache coherency issues and spinlocks would have to be used to protect the CPU caches. Instead a `ccupdate_t` struct is populated with all the

information that each CPU needs, and each CPU swaps the new data with the old information in the cache descriptor. The struct for storing the new cpucache information is defined as follows:

```

868 typedef struct ccupdate_struct_s
869 {
870     kmem_cache_t *cachep;
871     cpucache_t *new[NR_CPUS];
872 } ccupdate_struct_t;

```

`cachep` is the cache being updated, and `new` is the array of the cpucache descriptors for each CPU on the system. The function `smp_function_all_cpus()` is used to get each CPU to call the `do_ccupdate_local()` function, which swaps the information from `ccupdate_struct_t` with the information in the cache descriptor.

After the information has been swapped, the old data can be deleted.

8.5.5 Draining a Per-CPU Cache

When a cache is being shrunk, its last step is to drain the cpucaches of any objects they might have by calling `drain_cpu_caches()`. This is so that the slab allocator will have a clearer view of what slabs can be freed or not. This is important because, if just one object in a slab is placed in a per-cpu cache, that whole slab cannot be freed. If the system is tight on memory, saving a few milliseconds on allocations has a low priority.

8.6 Slab Allocator Initialization

Here I describe how the slab allocator initializes itself. When the slab allocator creates a new cache, it allocates the `kmem_cache_t` from the `cache_cache` or `kmem_cache` cache. This is an obvious chicken and egg problem, so the `cache_cache` has to be statically initialized as:

```

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 };

```

This code statically initialized the `kmem_cache_t` struct as follows:

358-360 This initializes the three lists as empty lists.

361 The size of each object is the size of a cache descriptor.

362 The creation and deleting of caches is extremely rare, so do not ever consider it for reaping.

363 This initializes the spinlock unlocked.

364 This aligns the objects to the L1 cache.

365 This records the human-readable name.

This code statically defines all the fields that can be calculated at compile time. To initialize the rest of the struct, `kmem_cache_init()` is called from `start_kernel()`.

8.7 Interfacing With the Buddy Allocator

The slab allocator does not come with pages attached; it must ask the physical page allocator for its pages. Two APIs are provided for this task called `kmem_getpages()` and `kmem_freepages()`. They are basically wrappers around the buddy allocators API so that slab pages will be taken into account for allocations. For allocations, the default pages are taken from `cache->gfpflags`, and the order is taken from `cache->gfporder` where `cache` is the cache requesting the pages. When freeing the pages, `PageClearSlab()` will be called for every page being freed before calling `free_pages()`.

8.8 What's New in 2.6

The most obvious change is that the version of the `/proc/slabinfo` format has changed from 1.1 to 2.0 and is a lot friendlier to read. The most helpful change is that the fields now have a header negating the need to memorize what each column means.

The principal algorithms and ideas remain the same. There are no major algorithm shakeups, but the implementation is quite different. Particularly, there is a greater emphasis on the use of per-cpu objects and the avoidance of locking. Second, a lot more debugging code is mixed in, so keep an eye out for `#ifdef DEBUG` blocks of code because they can be ignored when reading the code. Last, some changes are purely cosmetic with function name changes, but very similar behavior. For example, `kmem_cache_estimate()` is now called `cache_estimate()` even though they are identical in every other respect.

Cache descriptor The changes to the `kmem_cache_s` are minimal. First, the elements are reordered to have commonly used elements, such as the per-cpu related data, at the beginning of the struct (see Section 3.9 for the reasoning). Second, the slab lists (e.g. `slabs_full`) and statistics related to them have been moved to a separate struct `kmem_list3`. Comments and the unusual use of macros indicate that there is a plan to make the structure per-node.

Cache Static Flags The `flags` in 2.4 still exist, and their use is the same. `CFLGS_OPTIMIZE` no longer exists, but its use in 2.4 was nonexistent. Two new `flags` have been introduced, which are the following:

SLAB_STORE_USER This is a debugging-only `flag` for recording the function that freed an object. If the object is used after it was freed, the poison bytes will not match, and a kernel error message will be displayed. Because the last function to use the object is known, it can simplify debugging.

SLAB_RECLAIM_ACCOUNT This `flag` is set for caches with objects that are easily reclaimable, such as inode caches. A counter is maintained in a variable called `slab_reclaim_pages` to record how many pages are used in slabs allocated to these caches. This counter is later used in `vm_enough_memory()` to help determine if the system is truly out of memory.

Cache Reaping This is one of the most interesting changes made to the slab allocator. `kmem_cache_reap()` no longer exists because it is very indiscriminate in how it shrinks caches when the cache user could have made a far superior selection. Users of caches can now register a shrink cache callback with `set_shrinker()` for the intelligent aging and shrinking of slabs. This simple function populates a struct `shrinker` with a pointer to the callback and a `weight`, which indicates how difficult it is to recreate an object before placing it in a linked list called `shrinker_list`.

During page reclaim, the function `shrink_slab()` is called, which steps through the full `shrinker_list` and calls each shrinker callback twice. The first call passes 0 as a parameter, which indicates that the callback should return how many pages it expects it could free if it was called properly. A basic heuristic is applied to determine if it is worth the cost of using the callback. If it is, it is called a second time with a parameter indicating how many objects to free.

How this mechanism accounts for the number of pages is a little tricky. Each task struct has a `field` called `reclaim_state`. When the slab allocator frees pages, this `field` is updated with the number of pages that is freed. Before calling `shrink_slab()`, this `field` is set to 0 and then read again after `shrink_cache` returns to determine how many pages were freed.

Other changes The rest of the changes are essentially cosmetic. For example, the slab descriptor is now called struct `slab` instead of `slab_t`, which is consistent with the general trend of moving away from typedefs. Per-cpu caches remain essentially the same except the structs and APIs have new names. The same type of points applies to most of the 2.6 slab allocator implementation.

High Memory Management

The kernel may only directly address memory for which it has set up a page table entry. In the most common case, the user/kernel address space split of 3GiB/1GiB implies that, at best, only 896MiB of memory may be directly accessed at any given time on a 32-bit machine as explained in Section 4.1. On 64-bit hardware, this is not really an issue because there is more than enough virtual address space. It is highly unlikely there will be machines running 2.4 kernels with more than terabytes of RAM.

Many high end 32-bit machines have more than 1GiB of memory, and the inconveniently located memory cannot be simply ignored. The solution Linux uses is to temporarily map pages from high memory into the lower page tables. This will be discussed in Section 9.2.

High memory and I/O have a related problem that must be addressed because not all devices are able to address high memory or all the memory available to the CPU. This may be the case if the CPU has PAE extensions enabled, the device is limited to addresses the size of a signed 32-bit integer (2GiB) or a 32-bit device is being used on a 64-bit architecture. Asking the device to write to memory will fail at best and possibly disrupt the kernel at worst. The solution to this problem is to use a *bounce buffer*, and this will be discussed in Section 9.5.

This chapter begins with a brief description of how the *Persistent Kernel Map (PKMap)* address space is managed before talking about how pages are mapped and unmapped from high memory. The subsequent section will deal with the case where the mapping must be atomic before discussing bounce buffers in depth. Finally, we will talk about how emergency pools are used for when memory is very tight.

9.1 Managing the PKMap Address Space

Space is reserved at the top of the kernel page tables from PKMAP_BASE to FIXADDR_START for a PKMap. The size of the space reserved varies slightly. On the x86, PKMAP_BASE is at 0xFE000000, and the address of FIXADDR_START is a compile time constant that varies with configure options, but that is typically only a few pages located near the end of the linear address space. This means that there is slightly below 32MiB of page table space for mapping pages from high memory into usable space.

For mapping pages, a single page set of PTEs is stored at the beginning of the PKMap area to allow 1,024 high pages to be mapped into low memory for short periods with the function `kmap()` and to be unmapped with `kunmap()`. The pool seems very small, but the page is only mapped by `kmap()` for a *very* short time. Comments in the code indicate that there was a plan to allocate contiguous page table entries to expand this area, but it has remained just that, comments in the code, so a large portion of the PKMap is unused.

The page table entry for use with `kmap()` is called `pkmap_page_table`, which is located at `PKMAP_BASE` and which is set up during system initialization. On the x86, this takes place at the end of the `pagetable_init()` function. The pages for the PGD and PMD entries are allocated by the boot memory allocator to ensure they exist.

The current state of the page table entries is managed by a simple array called `pkmap_count`, which has `LAST_PKMAP` entries in it. On an x86 system without PAE, this is 1,024, and, with PAE, it is 512. More accurately, albeit not expressed in code, the `LAST_PKMAP` variable is equivalent to `PTRS_PER_PTE`.

Each element is not exactly a reference count, but it is very close. If the entry is 0, the page is free and has not been used since the last TLB ~~flush~~. If it is 1, the slot is unused, but a page is still mapped there waiting for a TLB ~~flush~~. Flashes are delayed until every slot has been used at least once because a global ~~flush~~ is required for all CPUs when the global page tables are modified and is extremely expensive. Any higher value is a reference count of n-1 users of the page.

9.2 Mapping High Memory Pages

The API for mapping pages from high memory is described in Table 9.1. The main function for mapping a page is `kmap()`, whose call graph is shown in Figure 9.1. For users that do not want to block, `kmap_nonblock()` is available, and interrupt users have `kmap_atomic()`. The `kmap` pool is quite small, so it is important that users of `kmap()` call `kunmap()` as quickly as possible because the pressure on this small window grows incrementally worse as the size of high memory grows in comparison to low memory.

The `kmap()` function itself is fairly simple. It ~~yet~~ checks to make sure an interrupt is not calling this function (because it may sleep) and calls `out_of_line_bug()` if true. An interrupt handler calling `BUG()` would panic the system, so `out_of_line_bug()` prints out bug information and exits cleanly. The second check is that the page is below `highmem_start_page` because pages below this mark are already visible and do not need to be mapped.

It then checks if the page is already in low memory and simply returns the address if it is. This way, users that need `kmap()` may use it unconditionally knowing that, if it is already a low memory page, the function is still safe. If it is a high page to be mapped, `kmap_high()` is called to begin the real work.

The `kmap_high()` function begins with checking the `page → virtual` ~~id~~, which is set if the page is already mapped. If it is NULL, `map_new_virtual()` provides a mapping for the page.

After a mapping has been created, the corresponding entry in the `pkmap_count` array is incremented, and the virtual address in low memory is returned.

9.3 Unmapping Pages

The API for unmapping pages from high memory is described in Table 9.2. The `kunmap()` function, like its complement, performs two checks. The first is an identical check to `kmap()` for usage from interrupt context. The second is that the page is below `highmem_start_page`. If it is, the page already exists in low memory and needs no further handling. After it is established that it is a page to be unmapped, `kunmap_high()` is called to perform the unmapping.

```
void kunmap(struct page *page)
```

This unmaps a struct page from low memory and frees up the page table entry mapping it.

```
void kunmap_atomic(void *kvaddr, enum km_type type)
```

This unmaps a page that was mapped atomically.

Table 9.2. High Memory Unmapping API

The `kunmap_high()` is simple in principle. It decrements the corresponding element for this page in `pkmap_count`. If it reaches 1 (remember this means no more users but a TLB flush is required), any process waiting on the `pkmap_map_wait` is woken up because a slot is now available. The page is not unmapped from the page tables then because that would require a TLB flush. It is delayed until `flush_all_zero_pkmops()` is called.

9.4 Mapping High Memory Pages Atomically

The use of `kmap_atomic()` is discouraged, but slots are reserved for each CPU for when they are necessary, such as when bounce buffers are used by devices from interrupt. There are a varying number of different requirements an architecture has for atomic high memory mapping, which are enumerated by `km_type`. The total number of uses is `KM_TYPE_NR`. On the x86, there are a total of six different uses for atomic kmaps.

`KM_TYPE_NR` entries per processor are reserved at boot time for atomic mapping at the location `FIX_KMAP_BEGIN` and ending at `FIX_KMAP_END`. Obviously, a user of an atomic kmap may not sleep or exit before calling `kunmap_atomic()` because the next process on the processor may try to use the same entry and fail.

The function `kmap_atomic()` has the very simple task of mapping the requested page to the slot set aside in the page tables for the requested type of operation and processor. The function `kunmap_atomic()`, whose call graph is shown in Figure 9.2,

function is `page_pool_free()`. The memory pools replace the emergency pool code that exists in 2.4.

To allocate or free objects from the memory pool, the memory pool API functions `mempool_alloc()` and `mempool_free()` are provided. Memory pools are destroyed with `mempool_destroy()`.

Mapping High Memory Pages In 2.4, the `page->virtual` was used to store the address of the page within the `pkmap_count` array. Due to the number of `struct page`s that exist in a high memory system, this is a very large penalty to pay for the relatively small number of pages that need to be mapped into `ZONE_NORMAL`. 2.6 still has this `pkmap_count` array, but it is managed very differently.

In 2.6, a hash table called `page_address_htable` is created. This table is hashed based on the address of the `struct page`, and the list is used to locate `struct page_address_slot`. This struct has two fields of interest, a `struct page` and a virtual address. When the kernel needs to find the virtual address used by a mapped page, it is located by traversing through this hash bucket. How the page is actually mapped into lower memory is essentially the same as 2.4 except now `page->virtual` is no longer required.

Performing I/O The last major change is that the `struct bio` is now used instead of the `struct buffer_head` when performing I/O. How `bio` structures work is beyond the scope of this book. However, the principle reason that `bio` structures were introduced is so that I/O could be performed in blocks of whatever size the underlying device supports. In 2.4, all I/O had to be broken up into page-sized chunks regardless of the transfer rate of the underlying device.

Page Frame Reclamation

A running system will eventually use all available page frames for purposes like disk buffers, dentries, inode entries, process pages and so on. Linux needs to select old pages that can be freed and invalidated for new uses before physical memory is exhausted. This chapter focuses exclusively on how Linux implements its page replacement policy and how different types of pages are invalidated.

The methods Linux uses to select pages are rather empirical in nature and the theory behind the approach is based on different ideas. It has been shown to work well in practice, and adjustments are made based on user feedback and benchmarks. The basics of the page replacement policy is the first item of discussion in this chapter.

The second topic of discussion is the *page cache*. All data that is read from disk is stored in the page cache to reduce the amount of disk I/O that must be performed. Strictly speaking, this is not directly related to page frame reclamation, but the LRU lists and page cache are closely related. The relevant section will focus on how pages are added to the page cache and quickly located.

This will bring us to the third topic, the LRU lists. With the exception of the slab allocator, all pages in use by the system are stored on LRU lists and linked together by `page->lru` so that they can be easily scanned for replacement. The slab pages are not stored on the LRU lists because it is considerably more difficult to age a page based on the objects used by the slab. The section focuses on how pages move through the LRU lists before they are reclaimed.

From there, I cover how pages belonging to other caches, such as the dcache and the slab allocator, are reclaimed before talking about how process-mapped pages are removed. Process-mapped pages are not easily swappable because there is no way to map `struct pages` to PTEs except to search every page table, which is far too expensive. If the page cache has a large number of process-mapped pages in it, process page tables will be walked, and pages will be swapped out by `swap_out()` until enough pages have been freed, but `swap_out()` will still have trouble with shared pages. If a page is shared, a swap entry is allocated, the PTE is filled with the necessary information to find the page in swap again and the reference count is decremented. Only when the count reaches zero will the page be freed. Pages like this are considered to be in the *swap cache*.

Finally, this chapter will cover the page replacement daemon `kswapd`, how it is implemented and what its responsibilities are.

10.1 Page Replacement Policy

During discussions the page replacement policy is frequently said to be a *LRU*-based algorithm, but this is not strictly speaking true because the lists are not strictly maintained in LRU order. The LRU in Linux consists of two lists called the *active_list* and the *inactive_list*. The objective is for the *active_list* to contain the *working set* [Den70] of all processes and the *inactive_list* to contain reclaim candidates. Because all reclaimable pages are contained in just two lists and pages belonging to any process may be reclaimed, rather than just those belonging to a faulting process, the replacement policy is a global one.

The lists resemble a simplified LRU 2Q [JS94] where two lists called *A_m* and *A₁* are maintained. With LRU 2Q, pages when first allocated are placed on a First In, First Out (FIFO) queue called *A₁*. If they are referenced while on that queue, they are placed in a normal LRU managed list called *A_m*. This is roughly analogous to using `lru_cache_add()` to place pages on a queue called *inactive_list* (*A₁*) and using `mark_page_accessed()` to get moved to the *active_list* (*A_m*). The algorithm describes how the size of the two lists have to be tuned, but Linux takes a simpler approach by using `refill_inactive()` to move pages from the bottom of *active_list* to *inactive_list* to keep *active_list* about two-thirds the size of the total page cache. Figure 10.1 illustrates how the two lists are structured, how pages are added and how pages move between the lists with `refill_inactive()`.

The lists described for 2Q presumes *A_m* is an LRU list, but the list in Linux closer resembles a clock algorithm [Car84] where the handspread is the size of the

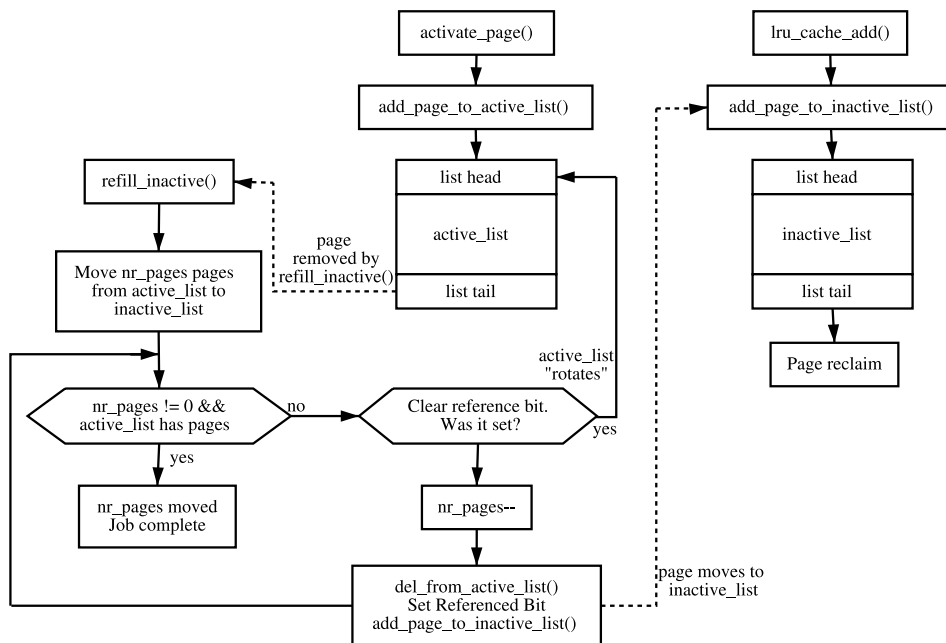


Figure 10.1. Page Cache LRU Lists

active list. When pages reach the bottom of the list, the referenced page is checked. If it is set, it is moved back to the top of the list, and the next page is checked. If it is cleared, it is moved to the `inactive_list`.

The Move-To-Front heuristic means that the lists behave in an LRU-like manner, but there are too many differences between the Linux replacement policy and LRU to consider it a stack algorithm [MM87]. Even if we ignore the problem of analyzing multiprogrammed systems [CD80] and the fact the memory size for each process is not fixed, the policy does not satisfy the *inclusion property* because the location of pages in the lists depend heavily upon the size of the lists as opposed to the time of last reference. The list priority is also not ordered because that would require list updates with every reference. As a final nail in the stack algorithm coffin, the lists are almost ignored when paging out from processes because pageout decisions are related to their location in the virtual address space of the process rather than the location within the page lists.

In summary, the algorithm does exhibit LRU-like behavior, and it has been shown by benchmarks to perform well in practice. There are only two cases where the algorithm is likely to behave really badly. The first is if the candidates for reclamation are principally anonymous pages. In this case, Linux will keep examining a large number of pages before linearly scanning process page tables searching for pages to reclaim, but this situation is fortunately rare.

The second situation is where there is a single process with many backed resident pages in the `inactive_list` that are being written to frequently. Processes and `kswapd` may go into a loop of constantly laundering these pages and placing them at the top of the `inactive_list` without freeing anything. In this case, few pages are moved from the `active_list` to `inactive_list` because the ratio between the two lists, sizes remains do not change significantly.

10.2 Page Cache

The page cache is a set of data structures that contain pages that are backed by regular files, block devices or swap. There are basically four types of pages that exist in the cache:

- One is pages that were faulted in as a result of reading a memory mapped file.
- Blocks read from a block device or filesystem are packed into special pages called buffer pages. The number of blocks that may be cached depends on the size of the block and the page size of the architecture.
- Anonymous pages exist in a special aspect of the page cache called the swap cache when slots are allocated in the backing storage for page-out, which is discussed further in Chapter 11.
- Pages belonging to shared memory regions are treated in a similar fashion to anonymous pages. The only difference is that shared pages are added to the swap cache and space reserved in backing storage immediately after the first write to the page.

The principal reason for the existence of this cache is to eliminate unnecessary disk reads. Pages read from disk are stored in a *page hash* table, which is hashed on the `struct address_space`, and the offset, which is always searched before the disk is accessed. An API is provided that is responsible for manipulating the page cache, which is listed in Table 10.1.

<pre>void add_to_page_cache(struct page * page, struct address_space * mapping, unsigned long offset)</pre> This adds a page to the LRU with <code>lru_cache_add()</code> in addition to adding it to the inode queue and page hash tables.
<pre>void add_to_page_cache_unique(struct page * page, struct address_space *mapping, unsigned long offset, struct page **hash)</pre> This is similar to <code>add_to_page_cache()</code> except it checks that the page is not already in the page cache. This is required when the caller does not hold the <code>pagecache_lock</code> spinlock.
<pre>void remove_inode_page(struct page *page)</pre> This function removes a page from the inode and hash queues with <code>remove_page_from_inode_queue()</code> and <code>remove_page_from_hash_queue()</code>, effectively removing the page from the page cache
<pre>struct page * page_cache_alloc(struct address_space *x)</pre> This is a wrapper around <code>alloc_pages()</code> that uses <code>x->gfp_mask</code> as the GFP mask.
<pre>void page_cache_get(struct page *page)</pre> This increases the reference count to a page already in the page cache.
<pre>int page_cache_read(struct file * file, unsigned long offset)</pre> This function adds a page corresponding to an offset with a file if it is not already there. If necessary, the page will be read from disk using an <code>address_space_operations->readpage</code> function.
<pre>void page_cache_release(struct page *page)</pre> This is an alias for <code>__free_page()</code>. The reference count is decremented, and, if it drops to 0, the page will be freed

Table 10.1. Page Cache API

10.2.1 Page Cache Hash Table

There is a requirement that pages in the page cache be quickly located. To facilitate this, pages are inserted into a table `page_hash_table`, and the fields `page->next_hash` and `page->pprev_hash` are used to handle collisions.

The table is declared as follows in `mm/filemap.c`:

```
45 atomic_t page_cache_size = ATOMIC_INIT(0);
46 unsigned int page_hash_bits;
47 struct page **page_hash_table;
```

The table is allocated during system initialization by `page_cache_init()`, which takes the number of physical pages in the system as a parameter. The desired size of the table (`htable_size`) is enough to hold pointers to every `struct page` in the system and is calculated by:

$$\text{htable\_size} = \text{num\_physpages} \cdot \text{sizeof}(\text{struct page})$$

To allocate a table, the system begins with an `order` allocation large enough to contain the entire table. It calculates this value by starting at 0 and incrementing it until $2^{\text{order}} > \text{htable\_size}$. This may be roughly expressed as the integer component of the following simple equation:

$$\text{order} = \log_2((\text{htable\_size} + 1))$$

An attempt is made to allocate this order of pages with `__get_free_pages()`. If the allocation fails, lower orders will be tried, and, if no allocation is satisfied, the system panics.

The value of `page_hash_bits` is based on the size of the table for use with the hashing function `_page_hashfn()`. The value is calculated by successive divides by two, but, in real terms, this is equivalent to:

$$\text{page\_hash\_bits} = \log_2 \left(\frac{\text{PAGE\_SIZE} \cdot 2^{\text{order}}}{\text{sizeof}(\text{struct page})} \right)$$

This makes the table a power-of-two hash table, which negates the need to use a modulus, which is a common choice for hashing functions.

10.2.2 Inode Queue

The *inode queue* is part of the `struct address_space` introduced in Section 4.4.2. The struct contains three lists. `clean_pages` is a list of clean pages associated with the inode; `dirty_pages` have been written to since the list sync to disk; and `locked_pages` are those currently locked. These three lists in combination are considered to be the inode queue for a given mapping, and the `page→list` field is used to link pages on it. Pages are added to the inode queue with `add_page_to_inode_queue()`, which places pages on the `clean_pages` lists and removes them with `remove_page_from_inode_queue()`.

10.2.3 Adding Pages to the Page Cache

Pages read from a `%s` or block device are generally added to the page cache to avoid further disk I/O. Most `%s` systems use the high-level function `generic_file_read()`

as their `file_operations→read()`, shown in Figure 10.2. The shared memory system, which is covered in Chapter 12, is one noteworthy exception, but, in general, systems perform their operations through the page cache. For the purposes of this section, we illustrate how `generic_file_read()` operates and how it adds pages to the page cache.

For normal I/O¹, `generic_file_read()` begins with a few basic checks before calling `do_generic_file_read()`. This searches the page cache by calling `__find_page_nolock()` with the `pagecache_lock` held to see if the page already exists in it. If it does not, a new page is allocated with `page_cache_alloc()`, which is a simple wrapper around `alloc_pages()` and is added to the page cache with `__add_to_page_cache()`. After a page frame is present in the page cache, `generic_file_readahead()` is called, which uses `page_cache_read()` to read the page from disk. It reads the page using `mapping→a_ops→readpage()`, where `mapping` is the address space managing the page. `readpage()` is the system-specific function used to read a page on disk.

Anonymous pages are added to the swap cache when they are unmapped from a process, which will be discussed further in Section 11.4. Until an attempt is made to swap them out, they have no address space acting as a mapping or any offset within a page, which leaves nothing to hash them into the page cache with. Note that these pages still exist on the LRU lists, however. Once in the swap cache, the only real difference between anonymous pages and swapped pages is that anonymous pages will use `swapper_space` as their `struct address_space`.

Shared memory pages are added during one of two cases. The first is during `shmem_getpage_locked()`, which is called when a page has to be either fetched from swap or allocated because it is the first reference. The second is when the swapout code calls `shmem_unuse()`. This occurs when a swap area is being deactivated and a page, backed by swap space, is found that does not appear to belong to any process. The inodes related to shared memory are exhaustively searched until the correct page is found. In both cases, the page is added with `add_to_page_cache()`, shown in Figure 10.3.

10.3 LRU Lists

As stated in Section 10.1, the LRU lists consist of two lists called `active_list` and `inactive_list`. They are declared in `mm/page_alloc.c` and are protected by the `pagemap_lru_lock` spinlock. They, broadly speaking, store the hot and cold pages respectively, or, in other words, the `active_list` contains all the working sets in the system, and `inactive_list` contains reclaim candidates. The API that deals with the LRU lists is listed in Table 10.2.

10.3.1 Refilling inactive list

When caches are being shrunk, pages are moved from the `active_list` to the `inactive_list` by the function `refill_inactive()`. It takes as a parameter the

¹Direct I/O is handled differently with `generic_file_direct_IO()`.

<pre>void lru_cache_add(struct page * page)</pre> Adds a cold page to the <code>inactive_list</code>. It will be moved to <code>active_list</code> with a call to <code>mark_page_accessed()</code> if the page is known to be hot, such as when a page is faulted in.
<pre>void lru_cache_del(struct page *page)</pre> Removes a page from the LRU lists by calling either <code>del_page_from_active_list()</code> or <code>del_page_from_inactive_list()</code>, whichever is appropriate.
<pre>void mark_page_accessed(struct page *page)</pre> Marks that the page has been accessed. If it was not recently referenced (in the <code>inactive_list</code> and <code>PG_referenced</code> ~~is~~ not set), the <code>referenced</code> ~~is~~ is set. If it is referenced a second time, <code>activate_page()</code> is called, which marks the page hot, and the <code>referenced</code> ~~is~~ is cleared.
<pre>void activate_page(struct page * page)</pre> Removes a page from the <code>inactive_list</code> and places it on <code>active_list</code>. It is very rarely called directly because the caller has to know the page is on <code>inactive_list</code>. <code>mark_page_accessed()</code> should be used instead.

Table 10.2. LRU List API

where `nr_inactive_pages` is the number of pages in the `inactive_list`. This means that, at lowest priority 6, at most one-sixth of the pages in the `inactive_list` will be scanned, and, at highest priority, all of them will be.

The second parameter is `max_mapped`, which determines how many process pages are allowed to exist in the page cache before whole processes will be swapped out. This is calculated as the minimum of either one-tenth of `max_scan` or

$$\text{max\_mapped} = \text{nr\_pages} \cdot 2^{(10 - \text{priority})}$$

In other words, at lowest priority, the maximum number of mapped pages allowed is either one-tenth of `max_scan` or 16 times the number of pages to swap out (`nr_pages`), whichever is the lower number. At high priority, it is either one-tenth of `max_scan` or 512 times the number of pages to swap out.

From there, the function is basically a very large for-loop that scans at most `max_scan` pages to free up `nr_pages` pages from the end of the `inactive_list` or until the `inactive_list` is empty. After each page, it checks to see whether it should reschedule itself so that the swapper does not monopolize the CPU.

For each type of page found on the list, it makes a different decision on what to do. The different page types and actions taken are handled in this order:

1. *Page is mapped by a process.* This jumps to the `page_mapped` label, which we will meet again in a later case. The `max_mapped` count is decremented. If it

reaches 0, the page tables of processes will be linearly searched and swapped out by the function `swap_out()`.

2. *Page is locked, and the PG_launder bit is set.* The page is locked for I/O, so it could be skipped over. However, if the `PG_launder` bit is set, it means that this is the second time that the page has been found locked, so it is better to wait until the I/O completes and get rid of it. A reference to the page is taken with `page_cache_get()` so that the page will not be freed prematurely, and `wait_on_page()` is called, which sleeps until the I/O is complete. After it is completed, the reference count is decremented with `page_cache_release()`. When the count reaches zero, the page will be reclaimed.
3. *Page is dirty, is unmapped by all processes, has no buffers and belongs to a device or no mapping.* Because the page belongs to a `%o` or device mapping, it has a valid `writepage()` function available through `page→mapping→a_ops→writepage`. The `PG_dirty` bit is cleared, and the `PG_launder` bit is set because it is about to start I/O. A reference is taken for the page with `page_cache_get()` before calling the `writepage()` function to synchronize the page with the backing `%o` before dropping the reference with `page_cache_release()`. Be aware that this case will also synchronize anonymous pages that are part of the swap cache with the backing storage because swap cache pages use `swapper_space` as a `page→mapping`. The page remains on the LRU. When it is found again, it will be simply freed if the I/O has completed, and the page will be reclaimed. If the I/O has not completed, the kernel will wait for the I/O to complete as described in the previous case.
4. *Page has buffers associated with data on disk.* A reference is taken to the page, and an attempt is made to free the pages with `try_to_release_page()`. If it succeeds and is an anonymous page (no `page→mapping`), the page is removed from the LRU, and `page_cache_release()` is called to decrement the usage count. There is only one case where an anonymous page has associated buffers and that is when it is backed by a swap `%o` because the page needs to be written out in block-sized chunk. If, on the other hand, it is backed by a `%o` or device, the reference is simply dropped, and the page will be freed as usual when the count reaches 0.
5. *Page is anonymous and is mapped by more than one process.* The LRU is unlocked, and the page is unlocked before dropping into the same `page_mapped` label that was encountered in the `%o` case. In other words, the `max_mapped` count is decremented, and `swap_out` is called when, or if, it reaches 0.
6. *Page has no process referencing it.* This is the `%o` case that is fallen into rather than explicitly checked for. If the page is in the swap cache, it is removed from it because the page is now synchronized with the backing storage and has no process referencing it. If it was part of a `%o`, it is removed from the inode queue, deleted from the page cache and freed.

10.4 Shrinking All Caches

The function responsible for shrinking the various caches is `shrink_caches()`, which takes a few simple steps to free up some memory (see Figure 10.4). The maximum number of pages that will be written to disk in any given pass is `nr_pages`, which is initialized by `try_to_free_pages_zone()` to be `SWAP_CLUSTER_MAX`. The limitation is there so that, if `kswapd` schedules a large number of pages to be written to disk, it will sleep occasionally to allow the I/O to take place. As pages are freed, `nr_pages` is decremented to keep count.

The amount of work that will be performed also depends on the priority initialized by `try_to_free_pages_zone()` to be `DEF_PRIORITY`. For each pass that does not free up enough pages, the priority is decremented for the highest priority of 1.

The function `shrink_caches()` calls `kmem_cache_reap()` (see Section 8.1.7), which selects a slab cache to shrink. If `nr_pages` number of pages are freed, the work is complete, and the function returns. Otherwise, it will try to free `nr_pages` from other caches.

If other caches are to be affected, `refill_inactive()` will move pages from the `active_list` to the `inactive_list` before shrinking the page cache by reclaiming pages at the end of the `inactive_list` with `shrink_cache()`.

Finally, it shrinks three special caches, the *dcache* (`shrink_dcache_memory()`), the *icache* (`shrink_icache_memory()`) and the *dqcache* (`shrink_dqcache_memory()`). These objects are quite small in themselves, but a cascading effect allows a lot more pages to be freed in the form of buffer and disk caches.

10.5 Swapping Out Process Pages

When `max_mapped` pages have been found in the page cache, `swap_out()`, shown in Figure 10.5, is called to start swapping out process pages. Starting from the `mm_struct` pointed to by `swap_mm` and the address `mm->swap_address`, the page tables are searched forward until `nr_pages` have been freed.

All process-mapped pages are examined regardless of where they are in the lists or when they were last referenced, but pages that are part of the `active_list` or have been recently referenced will be skipped over. The examination of hot pages is a bit costly, but insignificant in comparison to linearly searching all processes for the PTEs that reference a particular struct page.

After it has been decided to swap out pages from a process, an attempt will be made to swap out at least `SWAP_CLUSTER_MAX` number of pages, and the full list of `mm_structs` will only be examined once to avoid constant looping when no pages are available. Writing out the pages in bulk increases the chance that pages close together in the process address space will be written out to adjacent slots on disk.

The marker `swap_mm` is initialized to point to `init_mm`, and the `swap_address` is initialized to 0 the first time it is used. A task has been fully searched when the `swap_address` is equal to `TASK_SIZE`. After a task has been selected to swap pages from, the reference count to the `mm_struct` is incremented so that it will not be freed early, and `swap_out_mm()` is called with the selected `mm_struct` as a parameter.

pages when memory is running low. Historically, `kswapd` used to wake up every 10 seconds, but now it is only woken by the physical page allocator when the `pages_low` number of free pages in a zone is reached (see Section 2.2.1).

It is this daemon that performs most of the tasks needed to maintain the page cache correctly, shrink slab caches and swap out processes if necessary. Unlike swapout daemons such, as Solaris [MM01], which are woken up with increasing frequency because there is memory pressure, `kswapd` keeps freeing pages until the `pages_high` watermark is reached. Under extreme memory pressure, processes will do the work of `kswapd` synchronously by calling `balance_classzone()`, which calls `try_to_free_pages_zone()`. As shown in Figure 10.6, it is at `try_to_free_pages_zone()` where the physical page allocator synchronously performs the same task as `kswapd` when the zone is under heavy pressure.

When `kswapd` is woken up, it performs the following:

- It calls `kswapd_can_sleep()`, which cycles through all zones checking the `need_balance` bit in the `struct zone_t`. If any of them are set, it can not sleep.

- If it cannot sleep, it is removed from the `kswapd_wait` wait queue.

- It calls the functions `kswapd_balance()`, which cycles through all zones. It will free pages in a zone with `try_to_free_pages_zone()` if `need_balance` is set and will keep freeing until the `pages_high` watermark is reached.

- The task queue for `tq_disk` is run so that pages queued will be written out.

- It adds `kswapd` back to the `kswapd_wait` queue and goes back to the `get` step.

10.7 What's New in 2.6

kswapd As stated in Section 2.8, there is now a `kswapd` for every memory node in the system. These daemons are still started from `kswapd()`, and they all execute the same code, except their work is now confined to their local node. The main changes to the implementation of `kswapd` are related to the `kswapd-per-node` change.

The basic operation of `kswapd` remains the same. Once woken, it calls `balance_pgdat()` for the `pgdat` it is responsible for. `balance_pgdat()` has two modes of operation. When called with `nr_pages == 0`, it will continually try to free pages from each zone in the local `pgdat` until `pages_high` is reached. When `nr_pages` is specified, it will try and free either `nr_pages` or `MAX_CLUSTER_MAX * 8`, whichever is the smaller number of pages.

Balancing Zones The two main functions called by `balance_pgdat()` to free pages are `shrink_slab()` and `shrink_zone()`. `shrink_slab()` was from Section 8.8 so will not be repeated here. The function `shrink_zone()` is called to free a number of pages based on how urgent it is to free pages. This function behaves very similar to how 2.4 works. `refill_inactive_zone()` will move a number of pages from

zone→active_list to zone→inactive_list. Remember from Section 2.8 that LRU lists are now per-zone and not global as they are in 2.4. shrink_cache() is called to remove pages from the LRU and, reclaim pages.

Pageout Pressure In 2.4, the pageout priority determined how many pages would be scanned. In 2.6, there is a decaying average that is updated by zone_adj_pressure(). This adjusts the zone→pressure field to indicate how many pages should be scanned for replacement. When more pages are required, this will be pushed up toward the highest value of DEF_PRIORITY << 10 and then decays over time. The value of this average affects how many pages will be scanned in a zone for replacement. The objective is to have page replacement start working and slow gracefully rather than act in a bursty nature.

Manipulating LRU Lists In 2.4, a spinlock would be acquired when removing pages from the LRU list. This made the lock very heavily contended, so, to relieve contention, operations involving the LRU lists take place using struct pagevec structures. This allows pages to be added or removed from the LRU lists in batches of up to PAGEVEC_SIZE numbers of pages.

To illustrate, when refill_inactive_zone() and shrink_cache() are removing pages, they acquire the zone→lru_lock lock, remove large blocks of pages and store them on a temporary list. After the list of pages to remove is assembled, shrink_list() is called to perform the actual freeing of pages, which can now perform most of its task without needing the zone→lru_lock spinlock.

When adding the pages back, a new page vector struct is initialized with pagevec_init(). Pages are added to the vector with pagevec_add() and then committed to being placed on the LRU list in bulk with pagevec_release().

A sizable API is associated with pagevec structs that can be seen in <linux/pagevec.h> with most of the implementation in mm/swap.c.

Swap Management

Just as Linux uses free memory for purposes such as buffering data from disk, there eventually is a need to free up private or anonymous pages used by a process. These pages, unlike those backed by a file on disk, cannot be simply discarded to be read in later. Instead they have to be carefully copied to *backing storage*, sometimes called the *swap area*. This chapter details how Linux uses and manages its backing storage.

Strictly speaking, Linux does not swap because *swapping* refers to copying an entire process address space to disk and *paging* to copying out individual pages. Linux actually implements paging as modern hardware supports it, but traditionally has called it swapping in discussions and documentation. To be consistent with the Linux usage of the word, we, too, will refer to it as swapping.

There are two principal reasons that the existence of swap space is desirable. First, it expands the amount of memory that a process may use. Virtual memory and swap space allows a large process to run even if the process is only partially resident. Because old pages may be swapped out, the amount of memory addressed may easily exceed RAM because demand paging will ensure the pages are reloaded if necessary.

The casual reader¹ may think that, with a sufficient amount of memory, swap is unnecessary, but this brings me to the second reason. A significant number of the pages referenced by a process early in its life may only be used for initialization and then never used again. It is better to swap out those pages and create more disk buffers than leave them resident and unused.

Swap is not without its drawbacks, and the most important one is the most obvious one. Disk is slow, very very slow. If processes are frequently addressing a large amount of memory, no amount of swap or expensive high-performance disks will make it run within a reasonable time, only more RAM will help. This is why it is very important that the correct page be swapped out as discussed in Chapter 10, but also that related pages be stored close together in the swap space so they are likely to be swapped in at the same time while reading ahead. I start with how Linux describes a swap area.

This chapter begins with describing the structures Linux maintains about each active swap area in the system and how the swap area information is organized on disk. I cover how Linux remembers how to find pages in the swap after they

¹Not to mention the absent reader.

have been paged out and how swap slots are allocated. After that the *swap cache* is discussed, which is important for shared pages. At that point, there is enough information to begin understanding how swap areas are activated and deactivated, how pages are paged in and paged out and finally how the swap area is read and written to.

11.1 Describing the Swap Area

Each active swap area, be it a *%* or partition, has a struct `swap_info_struct` describing the area. All the structs in the running system are stored in a statically declared array called `swap_info`, which holds `MAX_SWAPFILES`, which is statically defined as 32, entries. This means that at most 32 swap areas can exist on a running system. The `swap_info_struct` is declared as follows in `<linux/swap.h>`:

```

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 };

```

Here is a small description of each of the fields in this quite sizable struct.

flags This is a bit field with two possible values. `SWP_USED` is set if the swap area is currently active. `SWP_WRITEOK` is defined as 3, the two lowest significant bits, *including* the `SWP_USED` bit. The flags are set to `SWP_WRITEOK` when Linux is ready to write to the area because it must be active to be written to.

swap_device The device corresponding to the partition used for this swap area is stored here. If the swap area is a *%*, this is NULL.

sdev_lock As with many structs in Linux, this one has to be protected, too. `sdev_lock` is a spinlock protecting the struct, principally the `swap_map`. It is locked and unlocked with `swap_device_lock()` and `swap_device_unlock()`.

swap_file This is the dentry for the actual special *%* that is mounted as a swap area. This could be the dentry for a *%* in the `/dev/` directory, for example,

in the case that a partition is mounted. This field is needed to identify the correct `swap_info_struct` when deactivating a swap area.

vfs_mount This is the `vfs_mount` object corresponding to where the device or `%s` for this swap area is stored.

swap_map This is a large array with one entry for every swap entry, or page-sized slot in the area. An entry is a reference count of the number of users of this page slot. The swap cache counts as one user, and every PTE that has been paged out to the slot counts as a user. If it is equal to `SWAP_MAP_MAX`, the slot is allocated permanently. If equal to `SWAP_MAP_BAD`, the slot will never be used.

lowest_bit This is the lowest possible free slot available in the swap area and is used to start from when linearly scanning to reduce the search space. It is known that there are definitely no free slots below this mark.

highest_bit This is the highest possible free slot available in this swap area. Similar to `lowest_bit`, there are definitely no free slots above this mark.

cluster_next This is the offset of the next cluster of blocks to use. The swap area tries to have pages allocated in cluster blocks to increase the chance related pages will be stored together.

cluster_nr This the number of pages left to allocate in this cluster.

prio Each swap area has a priority, which is stored in this field. Areas are arranged in order of priority and determine how likely the area is to be used. By default the priorities are arranged in order of activation, but the system administrator may also specify it using the `-p %d` when using `swapon`.

pages Because some slots on the swap `%s` may be unusable, this field stores the number of usable pages in the swap area. This differs from `max` in that slots marked `SWAP_MAP_BAD` are not counted.

max This is the total number of slots in this swap area.

next This is the index in the `swap_info` array of the next swap area in the system.

The areas, though stored in an array, are also kept in a pseudolist called `swap_list`, which is a very simple type declared as follows in `<linux/swap.h>`:

```
153 struct swap_list_t {
154     int head;    /* head of priority-ordered swapfile list */
155     int next;    /* swapfile to be used next */
156 };
```

The field `swap_list_t→head` is the swap area of the highest priority swap area in use, and `swap_list_t→next` is the next swap area that should be used. This is

so areas may be arranged in order of priority when searching for a suitable area, but still may be looked up quickly in the array when necessary.

Each swap area is divided up into a number of page-sized slots on disk, which means that each slot is 4,096 bytes on the x86, for example. The first slot is always reserved because it contains information about the swap area that should not be overwritten. The first 1 KiB of the swap area is used to store a disk label for the partition that can be picked up by userspace tools. The remaining space is used for information about the swap area, which is read when the swap area is created with the system program `mkswap`. The information is used to form a union `swap_header`, which is declared as follows in `<linux/swap.h>`:

```

25 union swap_header {
26     struct
27     {
28         char reserved[PAGE_SIZE - 10];
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 };

```

A description of each of the fields follows:

magic The `magic` part of the union is used just for identifying the magic string.

The string exists to make sure there is no chance a partition that is not a swap area will be used and to decide what version of swap area is to be used. If the string is `SWAP-SPACE`, it is version 1 of the swap format. If it is `SWAPSPACE2`, it is version 2. The large reserved array is just so that the magic string will be read from the end of the page.

bootbits This is the reserved area containing information about the partition, such as the disk label.

version This is the version of the swap area layout.

last_page This is the last usable page in the area.

nr_badpages The known number of bad pages that exist in the swap area are stored in this field.

padding A disk section is usually about 512 bytes in size. The three fields `version`, `last_page` and `nr_badpages` make up 12 bytes, and the padding fields make up the remaining 500 bytes to cover one sector.

badpages The remainder of the page is used to store the indices of up to `MAX_SWAP_BADPAGES` number of bad page slots. These slots are filled in by the `mkswap` system program if the `-c` switch is specified to check the area.

`MAX_SWAP_BADPAGES` is a compile time constant that varies if the struct changes, but it is 637 entries in its current form as given by the simple equation.

$$\text{MAX\_SWAP\_BADPAGES} = \frac{\text{PAGE\_SIZE} \times 1,024 \times 512 \times 10}{\text{sizeof}(\text{long})}$$

Where 1,024 is the size of the bootblock, 512 is the size of the padding and 10 is the size of the magic string identifying the format of the swap file.

11.2 Mapping Page Table Entries to Swap Entries

When a page is swapped out, Linux uses the corresponding PTE to store enough information to locate the page on disk again. Obviously, a PTE is not large enough in itself to store precisely where on disk the page is located, but it is more than enough to store an index into the `swap_info` array and an `o` set within the `swap_map`. This is precisely what Linux does.

Each PTE, regardless of architecture, is large enough to store a `swp_entry_t`, which is declared as follows in `<linux/shmem_fs.h>`:

```
16 typedef struct {
17     unsigned long val;
18 } swp_entry_t;
```

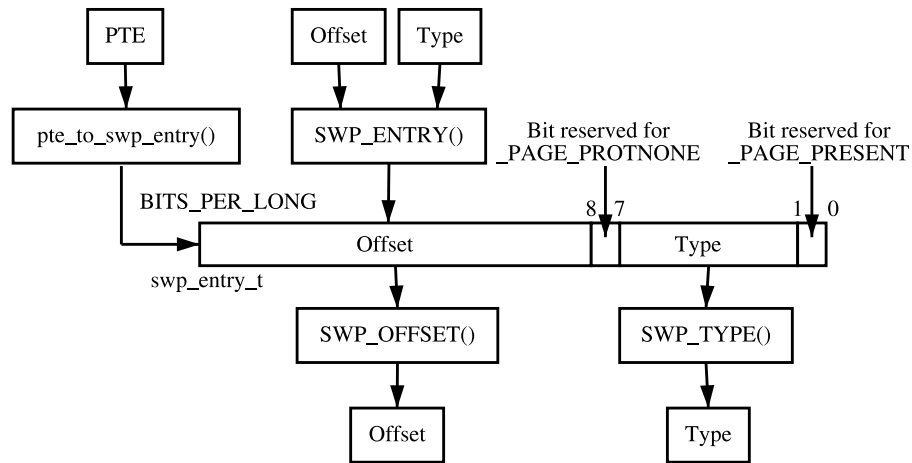
Two macros are provided for the translation of PTEs to swap entries and vice versa. They are `pte_to_swp_entry()` and `swp_entry_to_pte()`, respectively.

Each architecture has to be able to determine if a PTE is present or swapped out. For illustration, I show how this is implemented on the x86. In the `swp_entry_t`, two bits are always kept free. On the x86, Bit 0 is reserved for the `_PAGE_PRESENT` flag, and Bit 7 is reserved for `_PAGE_PROTNONE`. The requirement for both bits is explained in Section 3.2. Bits 1 through 6 are for the *type*, which is the index within the `swap_info` array and are returned by the `SWP_TYPE()` macro.

Bits 8 through 31 are used to store the `o` set within the `swap_map` from the `swp_entry_t`. On the x86, this means 24 bits are available, which limits the size of the swap area to 64GiB. The macro `SWP_OFFSET()` is used to extract the `o` set.

To encode a `type` and `o` set into a `swp_entry_t`, the macro `SWP_ENTRY()` is available, which simply performs the relevant bit-shifting operations. The relationship between all these macros is illustrated in Figure 11.1.

The six bits for `type` should allow up to 64 swap areas to exist in a 32-bit architecture instead of the `MAX_SWAPFILES` restriction of 32. The restriction is due to the consumption of the `vmalloc` address space. If a swap area is the maximum possible size, 32MiB is required for the `swap_map` ($2^{24} \times \text{sizeof}(\text{short})$); remember that each page uses one short for the reference count. For just `MAX_SWAPFILES`

Figure 11.1. Storing Swap Entry Information in `swp_entry_t`

maximum number of swap areas to exist, 1GiB of virtual malloc space is required, which is simply impossible because of the user/kernel linear address space split.

This would imply that supporting 64 swap areas is not worth the additional complexity, but there are cases where a large number of swap areas would be desirable even if the overall swap available does not increase. Some modern machines² have many separate disks, which, between them, can create a large number of separate block devices. In this case, it is desirable to create a large number of small swap areas that are evenly distributed across all disks. This would allow a high degree of parallelism in the page swapping behavior, which is important for swap-intensive applications.

11.3 Allocating a Swap Slot

All page-sized slots are tracked by the array `swap_info_struct → swap_map`, which is of type `unsigned short`. Each entry is a reference count of the number of users of the slot, which happens in the case of a shared page and is 0 when free. If the entry is `SWAP_MAP_MAX`, the page is permanently reserved for that slot. It is unlikely, if not impossible, for this condition to occur, but it exists to ensure the reference count does not overflow. If the entry is `SWAP_MAP_BAD`, the slot is unusable.

The task of finding and allocating a swap entry is divided into two major tasks. The first is performed by the high-level function `get_swap_page()`, shown in Figure 11.2. Starting with `swap_list → next`, it searches swap areas for a suitable slot. After a slot has been found, it records what the next swap area to be used will be and returns the allocated entry.

²A Sun E450 could have in the region of 20 disks in it, for example.


```
swp_entry_t get_swap_page()
```

This function allocates a slot in a `swap_map` by searching active swap areas. This is covered in greater detail in Section 11.3, but included here because it is principally used in conjunction with the swap cache.

```
int add_to_swap_cache(struct page *page, swp_entry_t entry)
```

This function adds a page to the swap cache. It ~~first~~ checks if it already exists by calling `swap_duplicate()`, and, if not, it adds it to the swap cache using the normal page cache interface function `add_to_page_cache_unique()`.

```
struct page * lookup_swap_cache(swp_entry_t entry)
```

This searches the swap cache and returns the `struct page` corresponding to the supplied entry. It works by searching the normal page cache based on `swapper_space` and the `swap_map` o set.

```
int swap_duplicate(swp_entry_t entry)
```

This function verifies a swap entry is valid and, if so, increments its swap map count.

```
void swap_free(swp_entry_t entry)
```

The complement function to `swap_duplicate()`. It decrements the relevant counter in the `swap_map`. When the count reaches zero, the slot is effectively free.

Table 11.1. Swap Cache API

Anonymous pages are not part of the swap cache *until* an attempt is made to swap them out. The variable `swapper_space` is declared as follows in `swap_state.c`:

```
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 };
```

A page is identified as being part of the swap cache after the `page->mapping` ~~id~~ has been set to `swapper_space`, which is tested by the `PageSwapCache()` macro. Linux uses the exact same code for keeping pages between swap and memory in sync as it uses for keeping ~~backed~~ pages and memory in sync. They both share the page cache code, but the differences are just in the functions used.

The address space for backing storage, `swapper_space`, uses `swap_ops` for its `address_space->a_ops`. The `page->index` ~~id~~ is then used to store the `swp_entry_t` structure instead of a ~~zero~~ o set, which is its normal purpose. The `address_space_operations` struct `swap_aops` is declared as follows in

swap_state.c:

```

34 static struct address_space_operations swap_aops = {
35     writepage: swap_writepage,
36     sync_page: block_sync_page,
37 };

```

When a page is being added to the swap cache, a slot is allocated with `get_swap_page()`, added to the page cache with `add_to_swap_cache()` and then marked dirty. When the page is next laundered, it will actually be written to backing storage on disk as the normal page cache would operate. This process is illustrated in Figure 11.4.

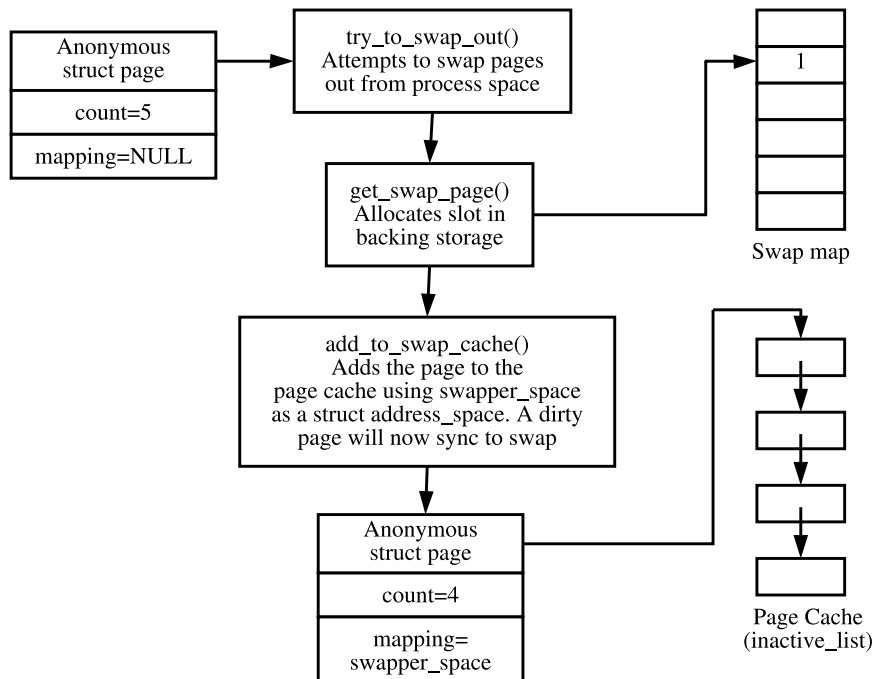


Figure 11.4. Adding a Page to the Swap Cache

Subsequent swapping of the page from shared PTEs results in a call to `swap_duplicate()`, which simply increments the reference to the slot in the `swap_map`. If the PTE is marked dirty by the hardware as a result of a write, the bit is cleared, and the `struct page` is marked dirty with `set_page_dirty()` so that the on-disk copy will be synced before the page is dropped. This ensures that, until all references to the page have been dropped, a check will be made to ensure the data on disk matches the data in the page frame.

When the reference count to the page fully reaches 0, the page is eligible to be dropped from the page cache, and the swap map count will have the count of the number of PTEs the on-disk slot belongs to so that the slot will not be freed

prematurely. It is laundered and finally dropped with the same LRU aging and logic described in Chapter 10.

If, on the other hand, a page fault occurs for a page that is swapped out, the logic in `do_swap_page()` will check to see if the page exists in the swap cache by calling `lookup_swap_cache()`. If it does, the PTE is updated to point to the page frame, the page reference count is incremented and the swap slot is decremented with `swap_free()`.

11.5 Reading Pages From Backing Storage

The principal function used when reading in pages is `read_swap_cache_async()`, which is mainly called during page faulting (see Figure 11.5). The function begins searching the swap cache with `find_get_page()`. Normally, swap cache searches are performed by `lookup_swap_cache()`, but that function updates statistics on the number of searches performed. Because the cache may need to be searched multiple times, `find_get_page()` is used instead.

The page can already exist in the swap cache if another process has the same page mapped or if multiple processes are faulting on the same page at the same time. If the page does not exist in the swap cache, one must be allocated and filled with data from backing storage.

After the page is allocated with `alloc_page()`, it is added to the swap cache with `add_to_swap_cache()` because swap cache operations may only be performed on pages in the swap cache. If the page cannot be added to the swap cache, the swap cache will be searched again to make sure another process has not put the data in the swap cache already.

To read information from backing storage, `rw_swap_page()` is called, which is discussed in Section 11.7. After the function completes, `page_cache_release()` is called to drop the reference to the page taken by `find_get_page()`.

11.6 Writing Pages to Backing Storage

When any page is being written to disk, the `address_space→a_ops` is consulted to find the appropriate write-out function. In the case of backing storage, the `address_space` is `swapper_space`, and the swap operations are contained in `swap_aops`. The struct `swap_aops` registers `swap_writepage()` because of its write-out function (see Figure 11.6).

The function `swap_writepage()` behaves differently depending on whether the writing process is the last user of the swap cache page or not. It knows this by calling `remove_exclusive_swap_page()`, which checks if there are any other processes using the page. This is a simple case of examining the page count with the `pagecache_lock` held. If no other process is mapping the page, it is removed from the swap cache and freed.

If `remove_exclusive_swap_page()` removed the page from the swap cache and freed it, `swap_writepage()` will unlock the page because it is no longer in use.

If it still exists in the swap cache, `rw_swap_page()` is called to write the data to the backing storage.

11.7 Reading/Writing Swap Area Blocks

The top-level function for reading and writing to the swap area is `rw_swap_page()`. This function ensures that all operations are performed through the swap cache to prevent lost updates. `rw_swap_page_base()` is the core function that performs the real work.

It begins by checking if the operation is a read. If it is, it clears the `uptodate` flag with `ClearPageUptodate()` because the page is obviously not up to date if I/O is required to write it with data. This flag will be set again if the page is successfully read from disk. It then calls `get_swaphandle_info()` to acquire the device for the swap partition of the inode for the swap page. These are required by the block layer, which will be performing the actual I/O.

The core function can work with either swap partition or file because it uses the block layer function `brw_page()` to perform the actual disk I/O. If the swap area is a file, `bmap()` is used to build a local array with a list of all blocks in the filesystem that contain the page data. Remember that filesystems may have their own method of storing pages and disk, and it is not as simple as the swap partition where information may be written directly to disk. If the backing storage is a partition, only one page-sized block requires I/O, and, because no filesystem is involved, `bmap()` is unnecessary.

After it is known what blocks must be read or written, a normal block I/O operation takes place with `brw_page()`. All I/O that is performed is asynchronous, so the function returns quickly. After the I/O is complete, the block layer will unlock the page, and any waiting process will wake up.

11.8 Activating a Swap Area

Now that you know what swap areas are, how they are represented and how pages are tracked, it is time to see how they all tie together to activate an area. Activating an area is conceptually quite simple: Open the file, load the header information from disk, populate a `swap_info_struct` and add it to the swap list.

The function responsible for the activation of a swap area is `sys_swapon()`, and it takes two parameters, the path to the special file for the swap area and a set of flags. While swap is being activated, the *Big Kernel Lock (BKL)* is held, which prevents any application from entering kernel space while this operation is being performed. The function is quite large, but can be broken down into the following simple steps:

1. Find a free `swap_info_struct` in the `swap_info` array and initialize it with default values.
2. Call `user_path_walk()`, which traverses the directory tree for the supplied specialfile and populates a `namidata` structure with the available data on the file, such as the dentry and the filesystem information for where it is stored (`vfsmount`).

3. Populate `swap_info_struct` fields pertaining to the dimensions of the swap area and how to find it. If the swap area is a partition, the block size will be configured to the `PAGE_SIZE` before calculating the size. If it is a file, the information is obtained directly from the inode.
4. Ensure the area is not already activated. If not, allocate a page from memory and read the first page-sized slot from the swap area. This page contains information such as the number of good slots and how to populate the `swap_info_struct → swap_map` with the bad entries.
5. Allocate memory with `vmalloc()` for `swap_info_struct → swap_map` and initialize each entry with 0 for good slots and `SWAP_MAP_BAD` otherwise. Ideally, the header information will be a version 2 format because version 1 was limited to swap areas of just under 128MiB for architectures with 4KiB page sizes like the x86.³
6. After ensuring the information indicated in the header matches the actual swap area, fill in the remaining information in the `swap_info_struct`, such as the maximum number of pages and the available good pages. Update the global statistics for `nr_swap_pages` and `total_swap_pages`.
7. The swap area is now fully active and initialized, so it is inserted into the swap list in the correct position based on priority of the newly activated area.

At the end of the function, the BKL is released, and the system now has a new swap area available for paging to.

11.9 Deactivating a Swap Area

In comparison to activating a swap area, deactivation is incredibly expensive. The principal problem is that the area cannot be simply removed. Every page that is swapped out must now be swapped back in again. Just as there is no quick way of mapping a struct page to every PTE that references it, there is no quick way to map a swap entry to a PTE either. This requires that all process page tables be traversed to find PTEs that reference the swap area to be deactivated and swap them in. This, of course, means that swap deactivation will fail if the physical memory is not available.

The function responsible for deactivating an area is, predictably enough, called `sys_swapoff()`. This function is mainly concerned with updating the `swap_info_struct`. The major task of paging in each paged-out page is the responsibility of `try_to_unuse()`, which is *extremely* expensive. For each slot used in the `swap_map`, the page tables for processes have to be traversed searching for it. In the worst case, all page tables belonging to all `mm_structs` may have to be traversed. Therefore, the tasks taken for deactivating an area are the following, broadly speaking:

1. Call `user_path_walk()` to acquire the information about the special file to be deactivated and then take the BKL.

³See the Code Commentary for the comprehensive reason for this.

2. Remove the `swap_info_struct` from the swap list and update the global statistics on the number of swap pages available (`nr_swap_pages`) and the total number of swap entries (`total_swap_pages`). After this is acquired, the BKL can be released again.
3. Call `try_to_unuse()`, which will page in all pages from the swap area to be deactivated. This function loops through the swap map using `find_next_to_unuse()` to locate the next used swap slot. For each used slot it finds, it performs the following:
 - ❖ Call `read_swap_cache_async()` to allocate a page for the slot saved on disk. Ideally, it exists in the swap cache already, but the page allocator will be called if it is not.
 - ❖ Wait on the page to be fully paged in and lock it. Once locked, call `unuse_process()` for every process that has a PTE referencing the page. This function traverses the page table searching for the relevant PTE and then updates it to point to the struct page. If the page is a shared memory page with no remaining reference, `shmem_unuse()` is called instead.
 - ❖ Free all slots that were permanently mapped. It is believed that slots will never become permanently reserved, so the risk is taken.
 - ❖ Delete the page from the swap cache to prevent `try_to_swap_out()` from referencing a page in the event it still somehow has a reference in swap from map.
4. If there was not enough available memory to page in all the entries, the swap area is reinserted back into the running system because it cannot be simply dropped. If it succeeded, the `swap_info_struct` is placed into an uninitialized state, and the `swap_map` memory is freed with `vfree()`

11.10 What's New in 2.6

The most important addition to the struct `swap_info_struct` is the addition of a linked list called `extent_list` and a cache field called `curr_swap_extent` for the implementation of extents.

Extents, which are represented by a struct `swap_extent`, map a contiguous range of pages in the swap area into a contiguous range of disk blocks. These extents are set up at swapon time by the function `setup_swap_extents()`. For block devices, there will only be one swap extent, and it will not improve performance, but the extent is set up so that swap areas backed by block devices or regular files can be treated the same.

It can make a large difference with swap files, which will have multiple extents representing ranges of pages clustered together in blocks. When searching for the page at a particular offset, the extent list will be traversed. To improve search times, the last extent that was searched will be cached in `swap_extent → curr_swap_extent`.

Shared Memory Virtual Filesystem

Sharing a region of memory backed by a file or device is simply a case of calling `mmap()` with the `MAP_SHARED` flag. However, there are two important cases where an anonymous region needs to be shared between processes. The first is when `mmap()` with `MAP_SHARED` is used without backing. These regions will be shared between a parent and child process after a `fork()` is executed. The second is when a region is explicitly setting them up with `shmget()` and is attached to the virtual address space with `shmat()`.

When pages within a VMA are backed by a file on disk, the interface used is straightforward. To read a page during a page fault, the required `no_page()` function is found in `vm_area_struct->vm_ops`. To write a page to backing storage, the appropriate `write_page()` function is found in the `address_space_operations` using `inode->i_mapping->a_ops` or alternatively using `page->mapping->a_ops`. When normal file operations are taking place, such as `mmap()`, `read()` and `write()`, the struct `file_operations` with the appropriate functions is found using `inode->i_fop` and so on. These relationships were illustrated in Figure 4.2.

This is a very clean interface that is conceptually easy to understand, but it does not help anonymous pages because there is no backing. To keep this nice interface, Linux creates an artificial backing for anonymous pages using a RAM-based filesystem where each VMA is backed by a file in this system. Every inode in the filesystem is placed on a linked list called `shmem_inodes` so that it may always be easily located. This allows the same file-based interface to be used without treating anonymous pages as a special case.

The filesystem comes in two variations called *shm* and *tmpfs*. They both share core functionality and mainly differ in what they are used for. *shm* is for use by the kernel for creating backings for anonymous pages and for backing regions created by `shmget()`. This filesystem is mounted by `kern_mount()` so that it is mounted internally and not visible to users. *tmpfs* is a temporary filesystem that may be optionally mounted on `/tmp/` to have a fast RAM-based temporary filesystem. A secondary use for *tmpfs* is to mount it on `/dev/shm/`. Processes that `mmap()` in the *tmpfs* filesystem will be able to share information between them as an alternative to System V Inter-Process Communication (IPC) mechanisms. Regardless of the type of use, *tmpfs* must be explicitly mounted by the system administrator.

The fields are the following:

`lock` is a spinlock protecting the inode information from concurrent accesses.

`next_index` is an index of the last page being used in the `%s`. This will be different from `inode->i_size` while a `%s` is being truncated.

`i_direct` is a direct block containing the `%s` SHMEM_NR_DIRECT swap vectors in use by the `%s`. See Section 12.4.1.

`i_indirect` is a pointer to the `%s` indirect block. See Section 12.4.1.

`swapped` is a count of the number of pages belonging to the `%s` that are currently swapped out.

`%s` is currently only used to remember if the `%s` belongs to a shared region set up by `shmget()`. It is set by specifying `SHM_LOCK` with `shmctl()` and unlocked by specifying `SHM_UNLOCK`.

`list` is a list of all inodes used by the `%s` system.

`inode` is a pointer to the parent inode.

12.2 Using shmem Functions

Different structs contain pointers for shmem specific functions. In all cases, `tmpfs` and `shm` share the same structs.

For faulting in pages and writing them to backing storage, two structs called `shmem_aops` and `shmem_vm_ops` of type `struct address_space_operations` and `struct vm_operations_struct`, respectively, are declared.

The address space operations struct `shmem_aops` contains pointers to a small number of functions of which the most important one is `shmem_writepage()`, which is called when a page is moved from the page cache to the swap cache. `shmem_removepage()` is called when a page is removed from the page cache so that the block can be reclaimed. `shmem_readpage()` is not used by `tmpfs`, but is provided so that the `sendfile()` system call may be used with `tmpfs` `%s`. `shmem_prepare_write()` and `shmem_commit_write()` are also unused, but are provided so that `tmpfs` can be used with the loopback device. `shmem_aops` is declared as follows in `mm/shmem.c`:

```
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 };
```

Anonymous VMAs use `shmem_vm_ops` as the `vm_operations_struct` so that `shmem_nopage()` is called when a new page is being faulted in. It is declared as follows:

```
1426 static struct vm_operations_struct shmem_vm_ops = {
1427     nopage: shmem_nopage,
1428 };
```

To perform operations on `%s` and inodes, two structs, `file_operations` and `inode_operations`, are required. The `file_operations`, called `shmem_file_operations`, provides functions that implement `mmap()`, `read()`, `write()` and `fsync()`. It is declared as follows:

```
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 };
```

Three sets of `inode_operations`, are provided. The `%s` is `shmem_inode_operations`, which is used for `%s` inodes. The second, called `shmem_dir_inode_operations`, is for directories. The last pair, called `shmem_symlink_inline_operations` and `shmem_symlink_inode_operations`, is for use with symbolic links.

The two `%s` operations supported are `truncate()` and `setattr()`, which are stored in a struct `inode_operations` called `shmem_inode_operations`. `shmem_truncate()` is used to truncate a `%s`. `shmem_notify_change()` is called when the `%s` attributes change. This allows, among other things, for a `%s` to be grown with `truncate()` and to use the global zero page as the data page. `shmem_inode_operations` is declared as follows:

```
1519 static struct inode_operations shmem_inode_operations = {
1520     truncate:      shmem_truncate,
1521     setattr:       shmem_notify_change,
1522 };
```

The directory `inode_operations` provides functions such as `create()`, `link()` and `mkdir()`. They are declared as follows:

```
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 };

```

The last pair of operations are for use with symlinks. They are declared as follows:

```

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 };

```

The difference between the two `readlink()` and `follow_link()` functions is related to where the link information is stored. A symlink inode does not require the private inode information `struct shmem_inode_information`. If the length of the symbolic link name is smaller than this struct, the space in the inode is used to store the name, and `shmem_symlink_inline_operations` becomes the inode operations struct. Otherwise, a page is allocated with `shmem_getpage()`, the symbolic link is copied to it and `shmem_symlink_inode_operations` is used. The second struct includes a `truncate()` function so that the page will be reclaimed when the file is deleted.

These various structs ensure that the shmem equivalent of inode-related operations will be used when regions are backed by virtual pages. When they are used, the majority of the VM sees no difference between pages backed by a real page and ones backed by virtual pages.

12.3 Creating Files in tmpfs

Because `tmpfs` is mounted as a proper filesystem that is visible to the user, it must support directory inode operations such as `open()`, `mkdir()` and `link()`. Pointers to functions that implement these for `tmpfs` are provided in `shmem_dir_inode_operations`, which is shown in Section 12.2.

The implementations of most of these functions are quite small, and, at some level, they are all interconnected as can be seen from Figure 12.2. All of them share the same basic principle of performing some work with inodes in the virtual filesystem, and the majority of the inode fields are filled in by `shmem_get_inode()`.

implemented with a few simple steps:

1. Record the current page → mapping and information about the inode.
2. Allocate a free slot in the backing storage with `get_swap_page()`.
3. Allocate a `swp_entry_t` with `shmem_swp_entry()`.
4. Remove the page from the page cache.
5. Add the page to the swap cache. If it fails, free the swap slot, add back to the page cache and try again.

12.5 File Operations in tmpfs

Four operations, `mmap()`, `read()`, `write()` and `fsync()`, are supported with virtual files. Pointers to the functions are stored in `shmem_file_operations`, which was shown in Section 12.2.

Little is unusual in the implementation of these operations, and they are covered in detail in the Code Commentary. The `mmap()` operation is implemented by `shmem_mmap()`, and it simply updates the VMA that is managing the mapped region. `read()`, implemented by `shmem_read()`, performs the operation of copying bytes from the virtual file to a userspace buffer, faulting in pages as necessary. `write()`, implemented by `shmem_write()`, is essentially the same. The `fsync()` operation is implemented by `shmem_file_sync()`, but is essentially a NULL operation because it performs no task and simply returns 0 for success. Because the files only exist in RAM, they do not need to be synchronized with any disk.

12.6 Inode Operations in tmpfs

The most complex operation that is supported for inodes is truncation and involves four distinct stages. The first, in `shmem_truncate()`, will truncate a partial page at the end of the file and continually calls `shmem_truncate_indirect()` until the file is truncated to the proper size. Each call to `shmem_truncate_indirect()` will only process one indirect block at each pass, which is why it may need to be called multiple times.

The second stage, in `shmem_truncate_indirect()`, understands both doubly and triply indirect blocks. It finds the next indirect block that needs to be truncated. This indirect block, which is passed to the third stage, will contain pointers to pages, which in turn contain swap vectors.

The third stage in `shmem_truncate_direct()` works with pages that contain swap vectors. It selects a range that needs to be truncated and passes the range to the last stage `shmem_swp_free()`. The last stage frees entries with `free_swap_and_cache()`, which frees both the swap entry and the page containing data.

The linking and unlinking of files is very simple because most of the work is performed by the filesystem layer. To link a file, the directory inode size is incremented,

the `ctime` and `mtime` of the affected inodes is updated and the number of links to the inode being linked to is incremented. A reference to the new dentry is then taken with `dget()` before instantiating the new dentry with `d_instantiate()`. Unlinking updates the same inode statistics before decrementing the reference to the dentry with `dput()`. `dput()` will also call `iput()`, which will clear up the inode when its reference count hits zero.

Creating a directory will use `shmem_mkdir()` to perform the task. It simply uses `shmem_mknod()` with the `S_IFDIR` flag before incrementing the parent directory inode's `i_nlink` counter. The function `shmem_rmdir()` will delete a directory by first ensuring it is empty with `shmem_empty()`. If it is, the function then decrements the parent directory inode's `i_nlink` count and calls `shmem_unlink()` to remove the requested directory.

12.7 Setting Up Shared Regions

A shared region is backed by a `Page` created in `shm`. There are two cases where a new `Page` will be created: during the setup of a shared region with `shmget()` and when an anonymous region is set up with `mmap()` with the `MAP_SHARED` flag. Both functions use the core function `shmem_file_setup()` to create a `Page`.

Because the `Page` system is internal, the names of the `Page`s created do not have to be unique because the `Page`s are always located by inode, not name. Therefore, `shmem_zero_setup()` (see Figure 12.5) always says to create a `Page` called `dev/zero`, which is how it shows up in the `Page` `/proc/pid/maps`. Files created by `shmget()` are called `SYSVNN` where the `NN` is the key that is passed as a parameter to `shmget()`.

The core function `shmem_file_setup()` simply creates a new dentry and inode, `Page`s in the relevant `Page`s and instantiates them.

12.8 System V IPC

The full internals of the IPC implementation are beyond the scope of this book. This section will focus just on the implementations of `shmget()` and `shmat()` and how they are affected by the VM. The system call `shmget()` is implemented by `sys_shmget()`, shown in Figure 12.6. It performs basic checks to the parameters and sets up the IPC-related data structures. To create the segment, it calls `newseg()`. This is the function that creates the `Page` in `shmfs` with `shmem_file_setup()` as discussed in the previous section.

The system call `shmat()` is implemented by `sys_shmat()`. There is little remarkable about the function. It acquires the appropriate descriptor and makes sure all the parameters are valid before calling `do_mmap()` to map the shared region into the process address space. Only two points of note are in the function.

The first is that it is responsible for ensuring that VMAs will not overlap if the caller specifies the address. The second is that the `shp→shm_nattach` counter is maintained by a `vm_operations_struct()` called `shm_vm_ops`. It registers `open()` and `close()` callbacks called `shm_open()` and `shm_close()`, respectively. The

`shm_close()` callback is also responsible for destroyed shared regions if the `SHM_DEST` flag is specified and the `shm_nattch` counter reaches zero.

12.9 What's New in 2.6

The core concept and functionality of the `%s` system remains the same, and the changes are either optimizations or extensions to the `%s` system's functionality. If the reader understands the 2.4 implementation well, the 2.6 implementation will not present much trouble.¹

A new field has been added to the `shmem_inode_info` called `allocated`. The `allocated` field stores how many data pages are allocated to the `%s`, which had to be calculated on the `%s` in 2.4 based on `inode->i_blocks`. It both saves a few clock cycles on a common operation as well as makes the code a bit more readable.

The flags field now uses the `VM_ACCOUNT` flag as well as the `VM_LOCKED` flag. The `VM_ACCOUNT`, always set, means that the VM will carefully account for the amount of memory used to make sure that allocations will not fail.

Extensions to the `%s` operations are the ability to seek with the system call `llseek()`, implemented by `generic_file_llseek()`, and to use `sendfile()` with virtual `%s`, implemented by `shmem_file_sendfile()`. An extension has been added to the VMA operations to allow nonlinear mappings, implemented by `shmem_populate()`.

The last major change is that the `%s` system is responsible for the allocation and destruction of its own inodes, which are two new callbacks in `struct super_operations`. It is simply implemented by the creation of a slab cache called `shmem_inode_cache`. A constructor function `init_once()` is registered for the slab allocator to use for initializing each new inode.

¹I find that saying how hard could it possibly be always helps.

Out of Memory Management

The last aspect of the VM I am going to discuss is the Out Of Memory (OOM) manager. This intentionally is a very short chapter because it has one simple task: check if there is enough available memory to satisfy, verify that the system is truly out of memory and, if so, select a process to kill. This is a controversial part of the VM and it has been suggested that it be removed on many occasions. Regardless of whether it exists in the latest kernel, it still is a useful system to examine because it touches a number of other subsystems.

13.1 Checking Available Memory

For certain operations, such as expanding the heap with `brk()` or remapping an address space with `mremap()`, the system will check if there is enough available memory to satisfy a request. Note that this is separate to the `out_of_memory()` path that is covered in the next section. This path is used to avoid the system being in a state of OOM if at all possible.

When checking available memory, the number of required pages is passed as a parameter to `vm_enough_memory()`. Unless the system administrator has specified that the system should overcommit memory, the amount of available memory will be checked. To determine how many pages are potentially available, Linux sums up the following bits of data:

`Total page cache` because page cache is easily reclaimed.

`Total free pages` because they are already available.

`Total free swap pages` because userspace pages may be paged out.

`Total pages managed by swapper_space` However, this double-counts the free swap pages. This is balanced by the fact that slots are sometimes reserved, but not used.

`Total pages used by the dentry cache` because they are easily reclaimed.

`Total pages used by the inode cache` because they are easily reclaimed.

If the total number of pages added here is sufficient for the request, `vm_enough_memory()` returns true to the caller. If false is returned, the caller knows that the memory is not available and usually decides to return `-ENOMEM` to userspace.

13.3 Selecting a Process

The function `select_bad_process()` is responsible for choosing a process to kill. It decides by stepping through each running task and calculating how suitable it is for killing with the function `badness()`. The badness is calculated as follows. The square roots are integer approximations calculated with `int_sqrt()`:

$$\text{badness\_for\_task} = \frac{\text{total\_vm\_for\_task}}{(\text{cpu\_time\_in\_seconds})^4 (\text{cpu\_time\_in\_minutes})}$$

This has been chosen to select a process that is using a large amount of memory, but is not that long lived. Processes that have been running a long time are unlikely to be the cause of memory shortage, so this calculation is likely to select a process that uses a lot of memory, but has not been running long. If the process is a root process or has `CAP_SYS_ADMIN` capabilities, the points are divided by four because it is assumed that root privilege processes are well behaved. Similarly, if it has `CAP_SYS_RAWIO` capabilities (access to raw devices) privileges, the points are further divided by four because it is undesirable to kill a process that has direct access to hardware.

13.4 Killing the Selected Process

After a task is selected, the list is walked again, and each process that shares the same `mm_struct` as the selected process (i.e., they are threads) is sent a signal. If the process has `CAP_SYS_RAWIO` capabilities, a `SIGTERM` is sent to give the process a chance of exiting cleanly. Otherwise, a `SIGKILL` is sent.

13.5 Is That It?

Yes, that is it. OOM management touches a lot of subsystems, but, otherwise, there is not much to it.

13.6 What's New in 2.6

The majority of OOM management remains essentially the same for 2.6 except for the introduction of VM-accounted objects. These are VMAs that are tagged with the `VM_ACCOUNT` flag, as mentioned in Section 4.8. Additional checks will be made to ensure there is memory available when performing operations on VMAs with this flag set. The principal incentive for this complexity is to avoid the need of an OOM killer.

Some regions that always have the `VM_ACCOUNT` flag set are the process stack, the process heap, regions `mmap()`ed with `MAP_SHARED`, private regions that are writable and regions that set up `shmget()`. In other words, most userspace mappings have the `VM_ACCOUNT` flag set.

Linux accounts for the amount of memory that is committed to these VMAs with `vm_acct_memory()`, which increments a variable called `committed_space`. When the

VMA is freed, the committed space is decremented with `vm_unacct_memory()`. This is a fairly simple mechanism, but it allows Linux to remember how much memory it has already committed to userspace when deciding if it should commit more.

The checks are performed by calling `security_vm_enough_memory()`, which introduces another new feature. A feature is available in 2.6 that allows security-related kernel modules to override certain kernel functions. The full list of hooks available is stored in a struct `security_operations` called `security_ops`. There are a number of dummy, or default, functions that may be used, which are all listed in `security/dummy.c`, but the majority do nothing except return. If no security modules are loaded, the `security_operations` struct used is called `dummy_security_ops`, which uses all the default functions.

By default, `security_vm_enough_memory()` calls `dummy_vm_enough_memory()`, which is declared in `security/dummy.c` and is very similar to 2.4's `vm_enough_memory()` function. The new version adds the following pieces of information together to determine available memory:

Total page cache because page cache is easily reclaimed.

Total free pages because they are already available.

Total free swap pages because userspace pages may be paged out.

Slab pages with `SLAB_RECLAIM_ACCOUNT` set because they are easily reclaimed.

These pages, minus a 3 percent reserve for root processes, is the total amount of memory that is available for the request. If the memory is available, it makes a check to ensure the total amount of committed memory does not exceed the allowed threshold. The allowed threshold is $\text{TotalRam} * (\text{OverCommitRatio}/100) + \text{TotalSwapPage}$, where `OverCommitRatio` is set by the system administrator. If the total amount of committed space is not too high, 1 will be returned so that the allocation can proceed.

The Final Word

Make no mistake, memory management is a large, complex and time-consuming field to research and difficult to apply to practical implementations. Because it is very difficult to model how systems behave in real multiprogrammed systems [CD80], developers often rely on intuition to guide them, and examination of virtual memory algorithms depends on simulations of specific workloads. Simulations are necessary because modeling how scheduling, paging behavior and multiple processes interact presents a considerable challenge. Page replacement policies, a field that has been the focus of considerable amounts of research, is a good example because it is only ever shown to work well for specific workloads. The problem of adjusting algorithms and policies to different workloads is addressed by having administrators tune systems as much as by research and algorithms.

The Linux kernel is also large, complex and fully understood by a relatively small core group of people. Its development is the result of contributions of thousands of programmers with a varying range of specialties, backgrounds and spare time. The best implementations are developed based on the all-important foundation that theory provides. Contributors built upon this framework with changes based on real-world observations.

It has been asserted on the Linux Memory Management mailing list that the VM is poorly documented and difficult to pick up because the implementation is a nightmare to follow¹ and the lack of documentation on practical VMs is not just confined to Linux. Matt Dillon, one of the principal developers of the FreeBSD VM² and considered a VM guru³, stated in an interview³ that documentation can be hard to come by. One of the principal difficulties with deciphering the implementation is the fact that the developer must have a background in memory management theory to see why implementation decisions were made because a pure understanding of the code is insufficient for any purpose other than microoptimizations.

This book attempted to bridge the gap between memory management theory and the practical implementation in Linux and to tie both fields together in a single place. It tried to describe what life is like in Linux as a memory manager in a

¹<http://mail.nl.linux.org/linux-mm/2002-05/msg00035.html>

²His past involvement with the Linux VM is evident from <http://mail.nl.linux.org/linux-mm/2000-05/msg00419.html>.

³<http://kerneltrap.com/node.php?id=8>

manner that was relatively independent of hardware architecture considerations. I hope after reading this and progressing onto the code commentary that you, the reader, feels a lot more comfortable with tackling the VM subsystem. As a ~~final~~ parting shot, Figure 14.1 broadly illustrates how the subsystems I discussed in detail interact with each other.

On a ~~final~~ personal note, I hope that this book encourages other people to produce similar works for other areas of the kernel. I know I ~~do~~ buy them!

B.1 Initializing Zones

Contents

B.1 Initializing Zones	220
B.1.1 Function: <code>setup_memory()</code>	220
B.1.2 Function: <code>zone_sizes_init()</code>	223
B.1.3 Function: <code>free_area_init()</code>	224
B.1.4 Function: <code>free_area_init_node()</code>	224
B.1.5 Function: <code>free_area_init_core()</code>	226
B.1.6 Function: <code>build_zonelists()</code>	232

B.1.1 Function: `setup_memory()` (*arch/i386/kernel/setup.c*)

The call graph for this function is shown in Figure 2.3. This function gets the necessary information to give to the boot memory allocator to initialize itself. It is broken up into a number of different tasks.

- Find the start and ending PFN for low memory (`min_low_pfn`, `max_low_pfn`), the start and end PFN for high memory (`highstart_pfn`, `highend_pfn`) and the PFN for the last page in the system (`max_pfn`).

- Initialize the `bootmem_data` structure and declare which pages may be used by the boot memory allocator.

- Mark all pages usable by the system as free, and then reserve the pages used by the bitmap representing the pages.

- Reserve pages used by the SMP console or the `initrd` image if one exists.

```

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     }

```


1020 `register_bootmem_low_pages()` reads the e820 map and calls `free_bootmem()` (See Section E.3.1) for all usable pages in the running system. This marks the pages as reserved during initialization as free.

1028-1029 Reserve the pages that are being used to store the bitmap representing the pages.

1035 Reserves page 0 because it is often a special page used by the BIOS.

1043 Reserves an extra page that is required by the trampoline code. The trampoline code deals with how userspace enters kernel space.

1045-1050 If sleep support is added, reserve memory is required for it. This is only of interest to laptops interested in suspending and is beyond the scope of this book.

```

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
1067                 "initrd extends beyond end of memory "
1068                 "(0x%08lx > 0x%08lx)\ndisabling initrd\n",
1069                 INITRD_START + INITRD_SIZE,
1070                 max_low_pfn << PAGE_SHIFT);
1071             initrd_start = 0;
1072         }
1073     }
1074 #endif
1075     return max_low_pfn;
1076 }

```

1055 This function reserves memory that stores configuration information about the SMP setup.

B.1.3 Function: `free_area_init()` (*mm/page_alloc.c*)

This is the architecture-independent function for setting up a UMA architecture. It simply calls the core function passing the static `contig_page_data` as the node. NUMA architectures will use `free_area_init_node()` instead.

```
838 void __init free_area_init(unsigned long *zones_size)
839 {
840     free_area_init_core(0, &contig_page_data, &mem_map,
                        zones_size, 0, 0, 0);
841 }
```

838 The parameters passed to `free_area_init_core()` are the following:

`0` is the Node Identifier (NID) for the node, which is 0.

`&contig_page_data` is the static global `pg_data_t`.

`&mem_map` is the global `mem_map` used for tracking struct pages. The function `free_area_init_core()` will allocate memory for this array.

`zones_size` is the array of zone sizes used by `zone_sizes_init()`.

This zero is the starting physical address.

The second zero is an array of memory hole sizes that does not apply to UMA architectures.

The last 0 is a pointer to a local `mem_map` for this node that is used by NUMA architectures.

B.1.4 Function: `free_area_init_node()` (*mm/numa.c*)

This function has two versions. The first is almost identical to `free_area_init()` except that it uses a different starting physical address. This function is also for architectures that have only one node (so they use `contig_page_data`), but their physical address is not at 0.

This version of the function, called after the pagetable initialization, is for initialization of each `pgdat` in the system. The callers have the option of allocating their own local portion of the `mem_map` and passing it in as a parameter if they want to optimize its location for the architecture. If they choose not to, it will be allocated later by `free_area_init_core()`.

```
61 void __init free_area_init_node(int nid,
    pg_data_t *pgdat, struct page *pmap,
62     unsigned long *zones_size, unsigned long zone_start_paddr,
63     unsigned long *zhole_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;
```


B.1.5 Function: `free_area_init_core()` (*mm/page_alloc.c*)

This function is responsible for initializing all zones and allocating their local `lmem_map` within a node. In UMA architectures, this function is called in a way that will initialize the global `mem_map` array. In NUMA architectures, the array is treated as a virtual array that is sparsely populated.

```

684 void __init free_area_init_core(int nid,
    pg_data_t *pgdat, struct page **gmap,
685     unsigned long *zones_size, unsigned long zone_start_paddr,
686     unsigned long *zholes_size, struct page *lmem_map)
687 {
688     unsigned long i, j;
689     unsigned long map_size;
690     unsigned long totalpages, offset, realtotalpages;
691     const unsigned long zone_required_alignment =
        1UL << (MAX_ORDER-1);
692
693     if (zone_start_paddr & ~PAGE_MASK)
694         BUG();
695
696     totalpages = 0;
697     for (i = 0; i < MAX_NR_ZONES; i++) {
698         unsigned long size = zones_size[i];
699         totalpages += size;
700     }
701     realtotalpages = totalpages;
702     if (zholes_size)
703         for (i = 0; i < MAX_NR_ZONES; i++)
704             realtotalpages -= zholes_size[i];
705
706     printk("On node %d totalpages: %lu\n", nid, realtotalpages);

```

This block is mainly responsible for calculating the size of each zone.

691 The zone must be aligned against the maximum-sized block that can be allocated by the buddy allocator for bitwise operations to work.

693-694 It is a bug if the physical address is not page aligned.

696 Initializes the `totalpages` count for this node to 0.

697-700 Calculates the total size of the node by iterating through `zone_sizes`.

701-704 Calculates the real amount of memory by subtracting the size of the holes in `zholes_size`.

706 Prints an informational message for the user on how much memory is available in this node.


```

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;

```

This block starts a loop that initializes every `zone_t` within the node. The initialization starts with the setting of the simpler fields that values already exist for.

728 Loops through all zones in the node.

733 Records a pointer to this zone in the `zone_table`. See Section 2.6.

734-736 Calculates the real size of the zone based on the full size in `zones_size` minus the size of the holes in `zholes_size`.

738 Prints an informational message saying how many pages are in this zone.

739 Records the size of the zone.

740 `zone_names` is the string name of the zone for printing purposes.

741-744 Initializes some other fields for the zone such as its parent `pgdat`.

745-746 If the zone has no memory, this continues to the next zone because nothing further is required.

```

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

```


765-766 The `zone_balance_min` ratios are 20 for all zones, which means that `pages_min` will never be below 20.

767-768 Similarly, the `zone_balance_max` ratios are all 255, so `pages_min` will never be over 255.

769 `pages_min` is set to `mask`.

770 `pages_low` is twice the number of pages as `pages_min`.

771 `pages_high` is three times the number of pages as `pages_min`.

773 Records where the `%s` struct `page` for this zone is located within `mem_map`.

774 Records the index within `mem_map` that this zone begins at.

775 Records the starting physical address.

777-778 Ensures that the zone is correctly aligned for use with the buddy allocator. Otherwise, the bitwise operations used for the buddy allocator will break.

```

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 Initially, all pages in the zone are marked as reserved because there is no way to know which ones are in use by the boot memory allocator. When the boot memory allocator is retiring in `free_all_bootmem()`, the unused pages will have their `PG_reserved` bit cleared.

786 Gets the page for this `o` set.

787 The zone the page belongs to is encoded with the page `%s`. See Section 2.6.

788 Sets the count to 0 because no one is using it.

834 Loops back to move to the next zone.

835 Builds the zone fallback lists for this node with `build_zonelists()`.

B.1.6 Function: `build_zonelists()` (*mm/page_alloc.c*)

This function builds the list of fallback zones for each zone in the requested node. This is for when an allocation cannot be satisfied and another zone is consulted. When this consultation is finished, allocations from `ZONE_HIGHMEM` will fall back to `ZONE_NORMAL`. Allocations from `ZONE_NORMAL` will fall back to `ZONE_DMA`, which in turn has nothing to fall back on.

```

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             /*
611              * fallthrough:
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)

```


B.2 Page Operations

Contents

B.2 Page Operations	234
B.2.1 Locking Pages	234
B.2.1.1 Function: lock_page()	234
B.2.1.2 Function: __lock_page()	234
B.2.1.3 Function: sync_page()	235
B.2.2 Unlocking Pages	236
B.2.2.1 Function: unlock_page()	236
B.2.3 Waiting on Pages	236
B.2.3.1 Function: wait_on_page()	236
B.2.3.2 Function: __wait_on_page()	237

B.2.1 Locking Pages

B.2.1.1 Function: lock_page() (*mm/vmap.c*)

This function tries to lock a page. If the page cannot be locked, it will cause the process to sleep until the page is available.

```

921 void lock_page(struct page *page)
922 {
923     if (TryLockPage(page))
924         __lock_page(page);
925 }
```

923 TryLockPage() is just a wrapper around test_and_set_bit() for the PG_locked bit in page→flags. If the bit was previously clear, the function returns immediately because the page is now locked.

924 Otherwise, __lock_page() is called (See Section B.2.1.2) to put the process to sleep.

B.2.1.2 Function: __lock_page() (*mm/vmap.c*)

This is called after a TryLockPage() failed. It will locate the waitqueue for this page and sleep on it until the lock can be acquired.

```

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)) {
```


B.2.2 Unlocking Pages

B.2.2.1 Function: `unlock_page()` (*mm/bmap.c*)

This function unlocks a page and wakes up any processes that may be waiting on it.

```

874 void unlock_page(struct page *page)
875 {
876     wait_queue_head_t *waitqueue = page_waitqueue(page);
877     ClearPageLaunder(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()` is the implementation of the hash algorithm, which determines which wait queue this page belongs to in the table zone → `wait_table`.

877 Clears the launder bit because I/O has now completed on the page.

878 This is a memory block operation that must be called before performing bit operations that may be seen by multiple processors.

879-880 Clears the `PG_locked` bit. It is a `BUG()` if the bit was already cleared.

881 Completes the SMP memory block operation.

889-890 If there are processes waiting on the page queue for this page, this wakes them.

B.2.3 Waiting on Pages

B.2.3.1 Function: `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 }
```


C.1 Page Table Initialization

Contents

C.1 Page Table Initialization	240
C.1.1 Function: <code>paging_init()</code>	240
C.1.2 Function: <code>pagetable_init()</code>	241
C.1.3 Function: <code>fixrange_init()</code>	245
C.1.4 Function: <code>kmap_init()</code>	246

C.1.1 Function: `paging_init()` (*arch/i386/mm/init.c*)

This is the top-level function called from `setup_arch()`. When this function returns, the page tables have been fully set up. Be aware that this is all x86 specific.

```

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     #ifdef CONFIG_HIGHMEM
365         kmap_init();
366     #endif
367     zone_sizes_init();
368 }

```

353 `pagetable_init()` is responsible for setting up a static page table using `swapper_pg_dir` as the PGD.

355 Loads the initialized `swapper_pg_dir` into the CR3 register so that the CPU will be able to use it.

362-363 If PAE is enabled, this sets the appropriate bit in the CR4 register.

366 Flushes all TLBs, including the global kernel ones.

369 `kmap_init()` initializes the region of pagetables reserved for use with `kmap()`.

371 `zone_sizes_init()` (See Section B.1.2) records the size of each of the zones before calling `free_area_init()` (See Section B.1.3) to initialize each zone.


```

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();

```

This loop begins setting up valid PMD entries to point to. In the PAE case, pages are allocated with `alloc_bootmem_low_pages()`, and the PGD is set appropriately. Without PAE, there is no middle directory, so it is just folded back onto the PGD to preserve the illusion of a three-level pagetable.

227 `i` is already initialized to the beginning of the kernel portion of the linear address space, so this keeps looping until the last `pgd_t` at `PTRS_PER_PGD` is reached.

228 Calculates the virtual address for this PGD.

229-230 If the end of `ZONE_NORMAL` is reached, this exits the loop because further pagetable entries are not needed.

231-234 If PAE is enabled, this allocates a page for the PMD and inserts the page into the pagetable with `set_pgd()`.

235 If PAE is not available, just set `pmd` to the current `pgd_t`. This is the `fold` back-trick for emulating three-level pagetables.

237-238 This is a sanity check to make sure the PMD is valid.

```

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);

```


260-265 For each `pte_t`, calculate the virtual address currently being examined and create a PTE that points to the appropriate physical page frame.

266 The PTEs have been initialized, so set the PMD to point to the page containing them.

267-268 Makes sure that the entry was established correctly.

```

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 }
```

At this point, pagetable entries have been set up that reference all parts of `ZONE_NORMAL`. The remaining regions needed are those for `fix` mappings and those needed for mapping high memory pages with `kmap()`.

277 The `fix` address space is considered to start at `FIXADDR_TOP` and to `fix` earlier in the address space. `__fix_to_virt()` takes an index as a parameter and returns the index `fix` pageframe backward (starting from `FIXADDR_TOP`)


```

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 Sets the starting virtual address (vadd) to the requested starting address provided as the parameter.

176 Gets the index within the PGD corresponding to vaddr.

177 Gets the index within the PMD corresponding to vaddr.

178 Gets the starting pgd_t.

180 Keeps cycling until end is reached. When pagetable_init() passes in 0, this loop will continue until the end of the PGD.

182-187 In the case of PAE, this allocates a page for the PMD if one has not already been allocated.

190 Without PAE, there is no PMD, so this treats the pgd_t as the pmd_t.

192-200 For each entry in the PMD, this allocates a page for the pte_t entries and sets it within the pagetables. Note that vaddr is incremented in PMD-sized strides.

C.1.4 Function: kmap_init() (*arch/i386/mm/init.c*)

This function only exists if CONFIG_HIGHMEM is set during compile time. It is responsible for caching where the beginning of the kmap region is, the PTE referencing it and the protection for the page tables. This means the PGD will not have to be checked every time kmap() is used.

```

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)

```


C.2 Page Table Walking

Contents

C.2 Page Table Walking	248
C.2.1 Function: follow_page()	248

C.2.1 Function: follow_page() (*mm/memory.c*)

This function returns the struct page used by the PTE at address in mm's pagetables.

```

405 static struct page * follow_page(struct mm_struct *mm,
                                   unsigned long address,
                                   int write)
406 {
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;
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 The parameters are the mm with the pagetables that are about to be walked, the address that has the struct page of interest and write, which indicates if the page is about to be written to.

411 Gets the PGD for the address and makes sure it is present and valid.

415-417 Gets the PMD for the address and makes sure it is present and valid.

D.2.4.3	Function: <code>move_vma()</code>	284
D.2.4.4	Function: <code>make_pages_present()</code>	288
D.2.4.5	Function: <code>get_user_pages()</code>	289
D.2.4.6	Function: <code>move_page_tables()</code>	292
D.2.4.7	Function: <code>move_one_page()</code>	293
D.2.4.8	Function: <code>get_one_pte()</code>	294
D.2.4.9	Function: <code>alloc_one_pte()</code>	295
D.2.4.10	Function: <code>copy_one_pte()</code>	295
D.2.5	Deleting a Memory Region	296
D.2.5.1	Function: <code>do_munmap()</code>	296
D.2.5.2	Function: <code>unmap_fixup()</code>	300
D.2.6	Deleting All Memory Regions	304
D.2.6.1	Function: <code>exit_mmap()</code>	304
D.2.6.2	Function: <code>clear_page_tables()</code>	306
D.2.6.3	Function: <code>free_one_pgd()</code>	306
D.2.6.4	Function: <code>free_one_pmd()</code>	307
D.3	Searching Memory Regions	309
D.3.1	Finding a Mapped Memory Region	309
D.3.1.1	Function: <code>find_vma()</code>	309
D.3.1.2	Function: <code>find_vma_prev()</code>	310
D.3.1.3	Function: <code>find_vma_intersection()</code>	312
D.3.2	Finding a Free Memory Region	312
D.3.2.1	Function: <code>get_unmapped_area()</code>	312
D.3.2.2	Function: <code>arch_get_unmapped_area()</code>	313
D.4	Locking and Unlocking Memory Regions	315
D.4.1	Locking a Memory Region	315
D.4.1.1	Function: <code>sys_mlock()</code>	315
D.4.1.2	Function: <code>sys_mlockall()</code>	316
D.4.1.3	Function: <code>do_mlockall()</code>	318
D.4.1.4	Function: <code>do_mlock()</code>	319
D.4.2	Unlocking the Region	321
D.4.2.1	Function: <code>sys_munlock()</code>	321
D.4.2.2	Function: <code>sys_munlockall()</code>	321
D.4.3	Fixing Up Regions After Locking/Unlocking	322
D.4.3.1	Function: <code>mlock_fixup()</code>	322
D.4.3.2	Function: <code>mlock_fixup_all()</code>	323
D.4.3.3	Function: <code>mlock_fixup_start()</code>	324
D.4.3.4	Function: <code>mlock_fixup_end()</code>	325
D.4.3.5	Function: <code>mlock_fixup_middle()</code>	326
D.5	Page Faulting	328
D.5.1	x86 Page Fault Handler	328
D.5.1.1	Function: <code>do_page_fault()</code>	328
D.5.2	Expanding the Stack	337

D.1 Process Memory Descriptors

Contents

D.1 Process Memory Descriptors	254
D.1.1 Initializing a Descriptor	254
D.1.2 Copying a Descriptor	254
D.1.2.1 Function: <code>copy_mm()</code>	254
D.1.2.2 Function: <code>mm_init()</code>	257
D.1.3 Allocating a Descriptor	257
D.1.3.1 Function: <code>allocate_mm()</code>	257
D.1.3.2 Function: <code>mm_alloc()</code>	258
D.1.4 Destroying a Descriptor	258
D.1.4.1 Function: <code>mmput()</code>	258
D.1.4.2 Function: <code>mmdrop()</code>	259
D.1.4.3 Function: <code>__mmdrop()</code>	260

This section covers the functions used to allocate, initialize, copy and destroy memory descriptors.

D.1.1 Initializing a Descriptor

The initial `mm_struct` in the system is called `init_mm` and is statically initialized at compile time using the macro `INIT_MM()`.

```

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 }
```

After it is established, new `mm_struct`s are copies of their parent `mm_struct` and are copied using `copy_mm()` with the process-specific fields initialized with `init_mm()`.

D.1.2 Copying a Descriptor

D.1.2.1 Function: `copy_mm()` (*kernel/fork.c*)

This function makes a copy of the `mm_struct` for the given task. This is only called from `do_fork()` after a new process has been created and needs its own `mm_struct`.

```

315 static int copy_mm(unsigned long clone_flags,
                      struct task_struct * tsk)
316 {
```



```

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 Allocates a new mm.

348-350 Copies the parent mm and initializes the process-specific mm with `init_mm()`.

352-353 Initializes the MMU context for architectures that do not automatically manage their MMU.

355-357 Calls `dup_mmap()`, which is responsible for copying all the VMA regions in use by the parent process.

```

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 `dup_mmap()` returns 0 on success. If it failed, the label `free_pt` will call `mmput()`, which decrements the use count of the mm.

365 Copies the LDT for the new process based on the parent process.

368-370 Sets the new mm, `active_mm`, and return success.

D.1.3.2 Function: `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 Allocates an `mm_struct` from the slab allocator.

254 Zeroes out all contents of the struct.

255 Performs basic initialization.

D.1.4 Destroying a Descriptor

A new user to an `mm` increments the usage count with a simple call:

```
atomic_inc(&mm->mm_users);
```

It is decremented with a call to `mmaput()`. If the `mm_users` count reaches zero, all the mapped regions are deleted with `exit_mmap()`, and the pagetables are destroyed because there are no longer any users of the userspace portions. The `mm_count` count is decremented with `mmdrop()` because all the users of the pagetables and VMAs are counted as one `mm_struct` user. When `mm_count` reaches zero, the `mm_struct` will be destroyed.

D.1.4.1 Function: `mmaput()` (*kernel/fork.c*)

```

276 void mmaput(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,
                                   struct mm_struct, mmlist);
282         list_del(&mm->mmlist);
283         mmlist_nr--;
284         spin_unlock(&mmlist_lock);
285         exit_mmap(mm);
286         mmdrop(mm);
287     }
288 }

```


D.1.4.3 Function: `__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 Makes sure the `init_mm` is not destroyed.

268 Deletes the PGD entry.

269 Deletes the LDT (Local Descriptor Table).

270 Calls `kmem_cache_free()` for the `mm`, freeing it with the slab allocator.


```

        unsigned long addr,
558        unsigned long len, unsigned long prot,
559        unsigned long flag, unsigned long offset)
560 {
561     unsigned long ret = -EINVAL;
562     if ((offset + PAGE_ALIGN(len)) < offset)
563         goto out;
564     if (!(offset & ~PAGE_MASK))
565         ret = do_mmap_pgoff(file, addr, len, prot, flag,
                           offset >> PAGE_SHIFT);
566 out:
567     return ret;
568 }

```

561 By default, this returns -EINVAL.

562-563 Makes sure that the size of the region will not overflow the total size of the address space.

564-565 Page aligns the offset and calls `do_mmap_pgoff()` to map the region.

D.2.1.2 Function: `do_mmap_pgoff()` (*mm/mmap.c*)

This function is very large, so it is broken up into a number of sections. Broadly speaking the sections are the following:

%Sanity check the parameters.

%Find a free linear address space large enough for the memory mapping. If a %system or device-specific %`get_unmapped_area()` function is provided, it will be used. Otherwise, `arch_get_unmapped_area()` is called.

%Calculate the VM %s and check them against the %access permissions.

%If an old area exists where the mapping is to take place, %it so it is suitable for the new mapping.

%Allocate a `vm_area_struct` from the slab allocator and %in its entries.

%Link in the new VMA.

%Call the %system or device-specific %`mmap()` function.

%Update statistics and exit.

```

393 unsigned long do_mmap_pgoff(struct file * file,
        unsigned long addr,
        unsigned long len, unsigned long prot,
394        unsigned long flags, unsigned long pgoff)
395 {
396     struct mm_struct * mm = current->mm;

```


419-420 Only `max_map_count` number of mappings are allowed. By default, this value is `DEFAULT_MAX_MAP_COUNT` or 65,536 mappings.

```

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 After basic sanity checks, this function will call the device- or `%specific%` `get_unmapped_area()` function. If a device-specific one is unavailable, `arch_get_unmapped_area()` is called. This function is discussed in Section D.3.2.2.

```

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
434                 | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
435
436     /* mlock MCL_FUTURE? */
437     if (vm_flags & VM_LOCKED) {
438         unsigned long locked = mm->locked_vm << PAGE_SHIFT;
439         locked += len;
440         if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
441             return -EAGAIN;
442     }

```

433 `calc_vm_flags()` translates the `prot` and `flags` from userspace and translates them to their VM equivalents.

436-440 Checks if it has been requested that all future mappings be locked in memory. If yes, it makes sure the process isn't locking more memory than it is allowed to. If it is, it returns `-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 don't allow writing to

```



```

479         break;
480     }
481 }

```

471-481 If the ~~file~~ is mapped for anonymous use, this ~~sets~~ ~~sets~~ up the ~~flags~~ if the requested mapping is MAP_PRIVATE to make sure the ~~flags~~ are consistent.

```

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()(See Section D.2.2.2) steps through the RB tree for the VMA corresponding to a given address.

486-488 If a VMA was found and it is part of the new mmaping, this removes the old mapping because the new one will cover both.

```

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))
507             goto out;

```

493-495 Ensures the new mapping will not exceed the total VM that a process is allowed to have. It is unclear why this check is not made earlier.

498-501 If the caller does not specifically request that free space is not checked with MAP_NORESERVE and it is a private mapping, this ensures enough memory is available to satisfy the mapping under current conditions.

531-536 This `%p` is cleared by the system call `mmap()`, but is still cleared for kernel modules that call this function directly. Historically, `-ETXTBUSY` was returned to the calling process if the underlying `%p` was written to.

537 Fills in the `vm_file` field.

538 Increments the `%p` usage count.

539 Calls the `%p` system or device-specific `mmap()` function. In many `%p` system cases, this will call `generic_file_mmap()` (See Section D.6.2.1).

540-541 If an error is called, this goes to `unmap_and_free_vma` to clean up and return the error.

```
542     } else if (flags & MAP_SHARED) {
543         error = shmem_zero_setup(vma);
544         if (error)
545             goto free_vma;
546     }
547
```

543 If this is an anonymous shared mapping, the region is created and set up by `shmem_zero_setup()` (See Section L.7.1). Anonymous shared pages are backed by a virtual tmpfs `%p` system so that they can be synchronized properly with swap. The writeback function is `shmem_writepage()` (See Section L.6.1).

```
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         */

```


598-600 This goto is used if the mapping failed immediately after the `vm_area_struct` is created. It is freed back to the slab allocator before the error is returned.

D.2.2 Inserting a Memory Region

The call graph for `insert_vm_struct()` is shown in Figure 4.5.

D.2.2.1 Function: `__insert_vm_struct()` (*mm/mmap.c*)

This is the top-level function for inserting a new vma into an address space. There is a second function like it called simply `insert_vm_struct()` that is not described in detail here because the only difference is the one line of code increasing the `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,
                             &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 The arguments are the `mm_struct` that represents the linear address space and the `vm_area_struct` that is to be inserted.

1179 `find_vma_prepare()` (See Section D.2.2.2) locates where the new VMA can be inserted. It will be inserted between `prev` and `__vma`, and the required nodes for the red-black tree are also returned.

1180-1181 This is a check to make sure the returned VMA is invalid. It is virtually impossible for this condition to occur without manually inserting bogus VMAs into the address space.

1182 This function does the actual work of linking the VMA struct into the linear linked list and the red-black tree.

1183 Increases the `map_count` to show a new mapping has been added. This line is not present in `insert_vm_struct()`.

1184 `validate_mm()` is a debugging macro for red-black trees. If `DEBUG_MM_RB` is set, the linear list of VMAs and the tree will be traversed to make sure it is


```

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 The function arguments are described previously.

253-255 Initializes the search.

263-272 This is a similar tree walk to what was described for `find_vma()`. The only real difference is the nodes last traversed are remembered with the `__rb_link` and `__rb_parent` variables.

275-276 Gets the back linking VMA through the red-black tree.

279 Returns the forward linking VMA.

D.2.2.3 Function: `vma_link()` (*mm/mmap.c*)

This is the top-level function for linking a VMA into the proper lists. It is responsible for acquiring the necessary locks to make a safe insertion.

```

337 static inline void vma_link(struct mm_struct * mm,
                             struct vm_area_struct * vma,
                             struct vm_area_struct * prev,
                             rb_node_t ** rb_link, rb_node_t * rb_parent)
338 {
339 {
340     lock_vma_mappings(vma);
341     spin_lock(&mm->page_table_lock);
342     __vma_link(mm, vma, prev, rb_link, rb_parent);
343     spin_unlock(&mm->page_table_lock);
344     unlock_vma_mappings(vma);
345
346     mm->map_count++;
347     validate_mm(mm);
348 }

```

337 `mm` is the address space that the VMA is to be inserted into. `prev` is the backward-linked VMA for the linear-linked-list of VMAs. `rb_link` and `rb_parent` are the nodes required to make the rb insertion.

340 This function acquires the spinlock that protects the `address_space` representing the ~~6~~ that is memory mapped.

341 Acquires the pagetable lock, which protects the whole `mm_struct`.

342 Inserts the VMA.

343 Frees the lock protecting the `mm_struct`.


```

292         else
293             vma->vm_next = NULL;
294     }
295 }

```

285 If prev is not null, the VMA is simply inserted into the list.

289 If not, this is the first mapping, and the first element of the list has to be stored in the mm_struct.

290 The VMA is stored as the parent node.

D.2.2.6 Function: __vma_link_rb() (mm/mmap.c)

The principal workings of this function are stored within <linux/rbtree.h> and will not be discussed in detail in this book.

```

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 {
302     rb_link_node(&vma->vm_rb, rb_parent, rb_link);
303     rb_insert_color(&vma->vm_rb, &mm->mm_rb);
304 }

```

D.2.2.7 Function: __vma_link_file() (mm/mmap.c)

This function links the VMA into a linked list of shared mappings.

```

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 inode's share list */
322         if((vma->vm_next_share = *head) != NULL)

```


~~%vm_flags~~ The permission ~~%s~~ of the region to be merged

353 This is the lock to the mm.

354-357 If prev is not passed it, it is taken to mean that the VMA being tested for merging is in front of the region from addr to end. The entry for that VMA is extracted from the rb_parent.

```

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 &&
365             can_vma_merge(next, vm_flags)) {
366             prev->vm_end = next->vm_end;
367             __vma_unlink(mm, next, prev);
368             spin_unlock(lock);
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 Checks to see if the region pointed to by prev may be expanded to cover the current region.

D.2.4 Remapping and Moving a Memory Region

D.2.4.1 Function: `sys_mremap()` (*mm/mremap.c*)

The call graph for this function is shown in Figure 4.6. This is the system service call to remap a memory region.

```

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 The parameters are the same as those described in the `mremap()` man page.

353 Acquires the mm semaphore.

354 `do_mremap()` (See Section D.2.4.2) is the top-level function for remapping a region.

355 Releases the mm semaphore.

356 Returns the status of the remapping.

D.2.4.2 Function: `do_mremap()` (*mm/mremap.c*)

This function does most of the actual work required to remap, resize and move a memory region. It is quite long, but can be broken up into distinct parts, which will be dealt with separately here. The tasks are, broadly speaking, the following:

- Check usage flags and page align lengths.

- Handle the condition where `MAP_FIXED` has set and the region has been moved to a new location.

- If a region is shrinking, allow it to happen unconditionally.

- If the region is growing or moving, perform a number of checks in advance to make sure the move is allowed and safe.

- Handle the case where the region has been expanded and cannot be moved.

- Finally, handle the case where the region has to be resized and moved.


```

245     /* Check if the location we're 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 }

```

This block handles the condition where the region location is `%d` and must be fully moved. It ensures the area being moved to is safe and definitely unmapped.

236 `MREMAP_FIXED` is the `%d` that indicates the location is `%d`.

237-238 The specified `new_addr` must be page-aligned.

239-240 If `MREMAP_FIXED` is specified, the `MAYMOVE` `%d` must be used as well.

242-243 Makes sure the resized region does not exceed `TASK_SIZE`.

248-249 Just as the comments indicate, the two regions being used for the move may not overlap.

254 Unmaps the region that is about to be used. It is presumed the caller ensures that the region is not in use for anything important.

```

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 At this point, the address of the resized region is the return value.

262 If the old length is larger than the new length, the region is shrinking.

263 Unmaps the unused region.

264-265 If the region is not to be moved, either because `MREMAP_FIXED` is not used or the new address matches the old address, goto out, which will return the address.

```

271     ret = -EFAULT;
272     vma = find_vma(current->mm, addr);
273     if (!vma || vma->vm_start > addr)
274         goto out;

```


294-297 Ensures that there is enough memory to satisfy the request after the resizing with `vm_enough_memory()` (See Section M.1.1).

```

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     }

```

This block handles the case where the region is being expanded and cannot be moved.

302 If it is the full region that is being remapped and ...

303 The region is definitely not being moved and ...

304 The region is being expanded and cannot be moved, then ...

305 Sets the maximum address that can be used to `TASK_SIZE`, which is 3GiB on an x86.

306-307 If there is another region, this sets the max address to be the start of the next region.

309-322 Only allows the expansion if the newly sized region does not overlap with the next VMA.

310 Calculates the number of extra pages that will be required.

311 Locks the mm spinlock.

312 Expands the VMA.

D.2.4.3 Function: `move_vma()` (*mm/mremap.c*)

The call graph for this function is shown in Figure 4.7. This function is responsible for moving all the pagetable entries from one VMA to another region. If necessary, a new VMA will be allocated for the region being moved to. Just like the previous function, it is very long, but may be broken up into the following distinct parts.

- %Function preamble % Finds the VMA preceding the area about to be moved to and the VMA in front of the region to be mapped.
- %Handles the case where the new location is between two existing VMAs. It determines if the preceding region can be expanded forward or the next region expanded backward to cover the new mapped region.
- %Handles the case where the new location is going to be the last VMA on the list. It determines if the preceding region can be expanded forward.
- %If a region could not be expanded, it allocates a new VMA from the slab allocator.
- %Calls `move_page_tables()`, if % in the new VMA details if a new one was allocated, and updates statistics before returning.

```

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 The parameters are the following:

- %vma The VMA that the address being moved belongs to
- %addr The starting address of the moving region
- %old_len The old length of the region to move
- %new_len The new length of the region moved
- %new_addr The new address to relocate to

134 Finds the VMA preceding the address being moved indicated by `prev` and returns the region after the new mapping as `next`.

```

135     if (next) {
136         if (prev && prev->vm_end == new_addr &&

```



```

145 If it can, this locks the mm.

146 Expands the VMA further to cover the next VMA.

147 There is now an extra VMA, so this unlinks it.

148 Unlocks the mm.

150 There is one less mapping now, so this updates the map_count.

151 Frees the memory used by the memory mapping.

153 If the prev region could not be expanded forward, this checks if the region
    pointed to be next may be expanded backward to cover the new mapping
    instead.

155 If it can, this locks the mm.

156 Expands the mapping backward.

157 Unlocks the mm.

158 The VMA representing the new mapping is now 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 &&
164             !(vma->vm_flags & VM_SHARED)) {
165         spin_lock(&mm->page_table_lock);
166         prev->vm_end = new_addr + new_len;
167         spin_unlock(&mm->page_table_lock);
168         new_vma = prev;
169     }

```

This block is for the case where the newly mapped region is the last VMA (next is NULL), so a check is made to see if the preceding region can be expanded.

```

161 Gets the previously mapped region.

162-163 Checks if the regions may be mapped.

164 Locks the mm.

165 Expands the preceding region to cover the new mapping.

166 Locks the mm.

167 The VMA representing the new mapping is now prev.

```



```

206         kmem_cache_free(vm_area_cache, new_vma);
207     out:
208         return -ENOMEM;
209 }

```

179 `move_page_tables()`(See Section D.2.4.6) is responsible for copying all the pagetable entries. It returns 0 on success.

182-193 If a new VMA was allocated, this ~~is~~ in all the relevant details, including the ~~is~~ device entries, and inserts it into the various VMA linked lists with `insert_vm_struct()`(See Section D.2.2.1).

194 Unmaps the old region because it is no longer required.

197 Updates the `total_vm` size for this process. The size of the old region is not important because it is handled within `do_munmap()`.

198-202 If the VMA has the `VM_LOCKED` ~~is~~, all the pages within the region are made present with `mark_pages_present()`.

203 Returns the address of the new region.

205-206 This is the error path. If a VMA was allocated, it deletes it.

208 Returns an out of memory error.

D.2.4.4 Function: `make_pages_present()` (*mm/memory.c*)

This function makes all pages between `addr` and `end` present. It assumes that the two addresses are within the one VMA.

```

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 Finds the VMA with `find_vma()`(See Section D.3.1.1) that contains the starting address.

~~No~~ ~~vmas~~ is similar to the `pages` array. If supplied, it will be ~~bad~~ with VMAs that were affected by the faults.

464 Sets the required ~~Yes~~ to `VM_WRITE` and `VM_MAYWRITE` ~~Yes~~ if the parameter `write` is set to 1. Otherwise, it uses the read equivalents.

465 If force is specified, this only requires the `MAY` ~~Yes~~.

```

468     do {
469         struct vm_area_struct * vma;
470
471         vma = find_extend_vma(mm, start);
472
473         if ( !vma ||
              (pages && vma->vm_flags & VM_IO) ||
              !(flags & vma->vm_flags) )
474             return i ? : -EFAULT;
475
476         spin_lock(&mm->page_table_lock);
477         do {
478             struct page *map;
479             while (!(map = follow_page(mm, start, write))) {
480                 spin_unlock(&mm->page_table_lock);
481                 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]);

```


508 Increments `i`, which is a counter for the number of pages present in the requested region.

509 Increments start in a page-sized stride.

510 Decrements the number of pages that must be faulted in.

511 Keeps moving through the VMAs until the requested pages have been faulted in.

512 Releases the pagetable spinlock.

515 Returns the number of pages known to be present in the region.

```

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 This will only be reached if a struct `page` is found that represents a non-existent page frame.

523-524 If one is found, it releases references to all pages stored in the `pages` array.

525-526 Returns `-EFAULT`.

D.2.4.6 Function: `move_page_tables()` (*mm/mremap.c*)

The call graph for this function is shown in Figure 4.8. This function is responsible for copying all the pagetable entries from the region pointed to `old_addr` to `new_addr`. It works by literally copying pagetable entries one at a time. When it is finished, it deletes all the entries from the old area. This is not the most efficient way to perform the operation, but it is very easy to error recover.

```

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     flush_cache_range(mm, old_addr, old_addr + len);
```



```

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 Acquires the mm lock.

83 Calls `get_one_pte()` (See Section D.2.4.8), which walks the pagetables to get the correct PTE.

84-85 If the PTE exists, this allocates a PTE for the destination and copies the PTEs with `copy_one_pte()` (See Section D.2.4.10).

86 Releases the lock.

87 Returns whatever `copy_one_pte()` returned. It will only return an error if `alloc_one_pte()` (See Section D.2.4.9) failed on line 85.

D.2.4.8 Function: `get_one_pte()` (*mm/mremap.c*)

This is a very simple pagetable walk.

```

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)) {

```



```

61 {
62     int error = 0;
63     pte_t pte;
64
65     if (!pte_none(*src)) {
66         pte = ptep_get_and_clear(src);
67         if (!dst) {
68             /* No dest? We must put it back. */
69             dst = src;
70             error++;
71         }
72         set_pte(dst, pte);
73     }
74     return error;
75 }

```

65 If the source PTE does not exist, this just returns 0 to say the copy was successful.

66 Gets the PTE and removes it from its old location.

67-71 If the dst does not exist, it means the call to `alloc_one_pte()` failed, and the copy operation has failed and must be aborted.

72 Moves the PTE to its new location.

74 Returns an error if one occurred.

D.2.5 Deleting a Memory Region

D.2.5.1 Function: `do_munmap()` (*mm/mmap.c*)

The call graph for this function is shown in Figure 4.10. This function is responsible for unmapping a region. If necessary, the unmapping can span multiple VMAs, and it can partially unmap one if necessary. Hence, the full unmapping operation is divided into two major operations. This function is responsible for finding what VMAs are affected, and `unmap_fixup()` is responsible for fixing up the remaining VMAs.

This function is divided up in a number of small sections that will be dealt with in turn. They are, broadly speaking, the following:

- Function as a preamble, and find the VMA to start working from.
- Take all VMAs affected by the unmapping out of the mm and place them on a linked list headed by the variable `free`.
- Cycle through the list headed by `free`, unmap all the pages in the region to be unmapped and call `unmap_fixup()` to fix up the mappings.
- Validate the mm and free memory associated with the unmapping.

948-950 The ~~948~~ part of the check sees if the VMA is just being partially unmapped. If it is, another VMA will be created later to deal with a region being broken into, so the `map_count` has to be checked to make sure it is not too large.

956-958 In case a new mapping is required, it is allocated now because later it will be much more difficult to back out in event of an error.

```

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);

```

This section takes all the VMAs affected by the unmapping and places them on a separate linked list headed by a variable called `free`. This makes the ~~948~~ `up` of the regions much easier.

960 `npp` becomes the next VMA in the list during the for loop that follows. To initialize it, it is either the current VMA (`mpnt`), or it becomes the ~~948~~ next VMA in the list.

961 `free` is the head of a linked list of VMAs that are affected by the unmapping.

962 Locks the mm.

963 Cycles through the list until the start of the current VMA is past the end of the region to be unmapped.

964 `npp` becomes the next VMA in the list.

965-966 Removes the current VMA from the linear linked list within the mm and places it on a linked list headed by `free`. The current `mpnt` becomes the head of the free linked list.

967 Deletes `mpnt` from the red-black tree.

969 Removes the cached result in case the last looked-up result is one of the regions to be unmapped.

970 Frees the mm.

987 Calculates the size of the region to be unmapped in this pass.

989-993 If the VM_DENYWRITE flag is specified, a hole will be created by this unmapping, and a page is mapped. Then, the `i_writecounts` are decremented. When this flag is negative, it counts how many users there are protecting this page from being opened for writing.

994 Removes the page mapping. If the page is still partially mapped, it will be acquired again during `unmap_fixup()` (See Section D.2.5.2).

995 Reduces the map count.

997 Removes all pages within this region.

1002 Calls `unmap_fixup()` (See Section D.2.5.2) to fix up the regions after this one is deleted.

1003-1004 Increments the writecount to the page because the region has been unmapped. If it was just partially unmapped, this call will simply balance out the decrement at line 987.

```

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_pgtables(mm, prev, addr, addr+len);
1013
1014     return 0;
1015 }
```

1006 `validate_mm()` is a debugging function. If enabled, it will ensure the VMA tree for this mm is still valid.

1009-1010 If extra VMA was not required, this deletes it.

1012 Frees all the pagetables that were used for the unmapped region.

1014 Returns success.

D.2.5.2 Function: `unmap_fixup()` (*mm/mmap.c*)

This function fixes up the regions after a block has been unmapped. It is passed a list of VMAs that are affected by the unmapping, the region and length to be unmapped and a spare VMA that may be required to fix up the region if a whole is created. This function handles four principle cases: the unmapping of a region, partial unmapping from the start to somewhere in the middle, partial unmapping from somewhere in the middle to the end and creation of a hole in the middle of the region. Each case will be taken in turn.

802-803 If a ~~%~~ or device is mapped, this calls `fput()`, which decrements the usage count and releases it if the count falls to 0.

804 Frees the memory for the VMA back to the slab allocator.

805 Returns the extra VMA because it was unused.

```

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     }

```

This block handles the case where the middle of the region to the end is been unmapped.

814 Truncates the VMA back to `addr`. At this point, the pages for the region have already freed, and the pagetable entries will be freed later, so no further work is required.

815 If a ~~%~~ or device is being mapped, the lock protecting shared access to it is taken in the function `lock_vma_mappings()`.

816 Locks the mm. Later in the function, the remaining VMA will be reinserted into the mm.

```

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 {

```

This block handles the case where the VMA is been unmapped from the start to some part in the middle.

818 Increases the `o` set within the ~~%~~ or device mapped by the number of pages this unmapping represents.

820 Moves the start of the VMA to the end of the region being unmapped.

821-822 Locks the ~~%~~ or device and mm as previously described.


```

852
853     __insert_vm_struct(mm, area);
854     spin_unlock(&mm->page_table_lock);
855     unlock_vma_mappings(area);
856     return extra;
857 }

```

853 Reinserts the VMA into the mm.

854 Unlocks the pagetables.

855 Unlocks the spinlock to the shared mapping.

856 Returns the extra VMA if it was not used and NULL if it was.

D.2.6 Deleting All Memory Regions

D.2.6.1 Function: `exit_mmap()` (*mm/mmap.c*)

This function simply steps through all VMAs associated with the supplied mm and unmaps them.

```

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);

```


1160 Flushes the TLB for this whole mm because it is about to be unmapped.

1163-1164 If the map_count is positive, it means the map count was not accounted for properly, so this calls BUG() to mark it.

1166 Clears the pagetables associated with this region with clear_page_tables() (See Section D.2.6.2).

D.2.6.2 Function: clear_page_tables() (*mm/memory.c*)

This is the top-level function used to unmap all PTEs and free pages within a region. It is used when pagetables need to be torn down, such as when the process exits or a region is unmapped.

```
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 Gets the PGD for the mm being unmapped.

150 Locks the pagetables.

151-155 Steps through all PGDs in the requested range. For each PGD found, this calls free_one_pgd() (See Section D.2.6.3).

156 Unlocks the pagetables.

159 Checks the cache of available PGD structures. If there are too many PGDs in the PGD quicklist, some of them will be reclaimed.

D.2.6.3 Function: free_one_pgd() (*mm/memory.c*)

This function tears down one PGD. For each PMD in this PGD, free_one_pmd() will be called.

```
109 static inline void free_one_pgd(pgd_t * dir)
110 {
```



```
106     pte_free(pte);  
107 }
```

97-98 If no PMD exists here, this returns.

99-103 If the PMD is bad, this ~~was~~ gives the error and returns.

104 Gets the ~~the~~ PTE in the PMD.

105 Clears the PMD from the pagetable.

106 Frees the PTE page to the PTE quicklist cache with `pte_free()`. Later, `check_pgt_cache()` will be called, and, if the cache has too many PTE pages in it, they will be reclaimed.


```

687         }
688         if (vma)
689             mm->mmap_cache = vma;
690     }
691 }
692 return vma;
693 }

```

661 The two parameters are the top-level `mm_struct` that is to be searched and the address the caller is interested in.

663 Defaults to returning `NULL` for address not found.

665 Makes sure the caller does not try to search a bogus `mm`.

668 `mmap_cache` has the result of the last call to `find_vma()`. This has a chance of not having to search at all through the red-black tree.

669 If it is a valid VMA that is being examined, this checks to see if the address being searched is contained within it. If it is, the VMA was the `mmap_cache` one, so it can be returned. Otherwise, the tree is searched.

670-674 Starts at the root of the tree.

675-687 This block is the tree walk.

678 The macro, as the name suggests, returns the VMA that this tree node points to.

680 Checks if the next node is traversed by the left or right leaf.

682 If the current VMA is what is required, this exits the while loop.

689 If the VMA is valid, this sets the `mmap_cache` for the next call to `find_vma()`.

692 Returns the VMA that contains the address or, as a side effect of the tree walk, returns the VMA that is closest to the requested address.

D.3.1.2 Function: `find_vma_prev()` (*mm/mmap.c*)

```

696 struct vm_area_struct * find_vma_prev(struct mm_struct * mm,
                                         unsigned long addr,
697                                         struct vm_area_struct **pprev)
698 {
699     if (mm) {
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;

```


731-732 Extracts the VMA from the red-black tree node.

733-734 A debugging check. If this is the previous node, its next ~~node~~ should point to the VMA being returned. If it is not, it is a bug.

D.3.1.3 Function: `find_vma_intersection()` (*include/linux/mm.h*)

```
673 static inline struct vm_area_struct * find_vma_intersection(
        struct mm_struct * mm,
        unsigned long start_addr, unsigned long end_addr)
674 {
675     struct vm_area_struct * vma = find_vma(mm, start_addr);
676
677     if (vma && end_addr <= vma->vm_start)
678         vma = NULL;
679     return vma;
680 }
```

675 Returns the VMA closest to the starting address.

677 If a VMA is returned and the end address is still less than the beginning of the returned VMA, the VMA does not intersect.

679 Returns the VMA if it does intersect.

D.3.2 Finding a Free Memory Region

D.3.2.1 Function: `get_unmapped_area()` (*mm/mmap.c*)

The call graph for this function is shown in Figure 4.4.

```
644 unsigned long get_unmapped_area(struct file *file,
        unsigned long addr,
        unsigned long len,
        unsigned long pgoff,
        unsigned long flags)
645 {
646     if (flags & MAP_FIXED) {
647         if (addr > TASK_SIZE - len)
648             return -ENOMEM;
649         if (addr & ~PAGE_MASK)
650             return -EINVAL;
651         return addr;
652     }
653
654     if (file && file->f_op && file->f_op->get_unmapped_area)
655         return file->f_op->get_unmapped_area(file, addr,
            len, pgoff, flags);
656 }
```



```

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      addr = vma->vm_end;
638  }
639 }
640 #else
641 extern unsigned long arch_get_unmapped_area(struct file *,
        unsigned long, unsigned long,
        unsigned long, unsigned long);
642 #endif

```

614 If this is not defined, it means that the architecture does not provide its own `arch_get_unmapped_area()`, so this one is used instead.

615 The parameters are the same as those for `get_unmapped_area()` (See Section D.3.2.1).

619-620 A sanity check to make sure the required map length is not too long.

622-628 If an address is provided, this uses it for the mapping.

623 Makes sure the address is page aligned.

624 `find_vma()` (See Section D.3.1.1) will return the region closest to the requested address.

625-627 Makes sure the mapping will not overlap with another region. If it does not, it returns it because it is safe to use. Otherwise, it gets ignored.

629 `TASK_UNMAPPED_BASE` is the starting point for searching for a free region to use.

631-638 Starting from `TASK_UNMAPPED_BASE`, this linearly searches the VMAs until a large enough region between them is found to store the new mapping. This is essentially a ~~for~~ ~~while~~ search.

641 If an external function is provided, it still needs to be declared here.


```

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 Takes the semaphore. We are likely to sleep during this, so a spinlock cannot be used.

202 Rounds the length up to the page boundary.

203 Rounds the start address down to the page boundary.

205 Calculates how many pages will be locked.

206 Calculates how many pages will be locked in total by this process.

208-209 Calculates what the limit is to the number of locked pages.

212-213 Does not allow the process to lock more than it should.

217-218 Does not allow the process to map more than half of physical memory.

220 Calls `do_mlock()` (See Section D.4.1.4), which starts the real work by finding the VMA closest to the area to lock before calling `mlock_fixup()` (See Section D.4.3.1).

222 Frees the semaphore.

223 Returns the error or success code from `do_mlock()`.

D.4.1.2 Function: `sys_mlockall()` (*mm/mlock.c*)

This is the system call `mlockall()`, which attempts to lock all pages in the calling process in memory. If `MCL_CURRENT` is specified, all current pages will be locked. If `MCL_FUTURE` is specified, all future mappings will be locked. The flags may be or-ed together. This function makes sure that the flags and process limits are ok before calling `do_mlockall()`.

```

266 asmlinkage long sys_mlockall(int flags)
267 {
268     unsigned long lock_limit;

```


D.4.1.3 Function: `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,
                             newflags);
260         if (error)
261             break;
262     }
263     return error;
264 }

```

244-245 The calling process must be either root or have `CAP_IPC_LOCK` capabilities.

248-250 The `MCL_FUTURE` flag says that all future pages should be locked, so, if set, the `def_flags` for VMAs should be `VM_LOCKED`.

253-262 Cycles through all VMAs.

256 Sets the `VM_LOCKED` flag in the current VMA flags.

257-258 If the `MCL_CURRENT` flag has not been set requesting that all current pages be locked, then this clears the `VM_LOCKED` flag. The logic is arranged like this so that the unlock code can use this same function, just with no flags.

259 Calls `mlock_fixup()` (See Section D.4.3.1), which will adjust the regions to match the locking as necessary.

260-261 If a nonzero value is returned at any point, this stops locking. It is interesting to note that VMAs already locked will not be unlocked.


```

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 }

```

This block walks through the VMAs affected by this locking and calls `mlock_fixup()` for each of them.

166-192 Cycles through as many VMAs as necessary to lock the pages.

171 Sets the `VM_LOCKED` flag on the VMA.

172-173 If this is an unlock, it removes the flag.

175-177 If this VMA is the last VMA to be affected by the unlocking, this calls `mlock_fixup()` with the end address for the locking and exits.

180-190 This is whole VMA that needs to be locked. To lock it, the end of this VMA is passed as a parameter to `mlock_fixup()` (See Section D.4.3.1) instead of the end of the actual locking.

180 `tmp` is the end of the mapping on this VMA.

181 `next` is the next VMA that will be affected by the locking.

182 Calls `mlock_fixup()` (See Section D.4.3.1) for this VMA.

183-184 If an error occurs, this backs out. Note that the VMAs already locked are not fixed up right.

185 The next start address is the start of the next VMA.


```

298     ret = do_mlockall(0);
299     up_write(&current->mm->mmap_sem);
300     return ret;
301 }

```

297 Acquires the semaphore protecting the `mm_struct`.

298 Calls `do_mlockall()` (See Section D.4.1.3) with 0 as `flags`, which will remove the `VM_LOCKED` from all VMAs.

299 Releases the semaphore.

300 Returns the error or success code.

D.4.3 Fixing Up Regions After Locking/Unlocking

D.4.3.1 Function: `mlock_fixup()` (*mm/mlock.c*)

This function identifies four separate types of locking that must be addressed. The first is where the full VMA is to be locked, and it calls `mlock_fixup_all()`. The second is where only the beginning portion of the VMA is affected, which is handled by `mlock_fixup_start()`. The third is the locking of a region at the end, which is handled by `mlock_fixup_end()`, and the last is locking a region in the middle of the VMA with `mlock_fixup_middle()`.

```

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     if (!retval) {
138         /* keep track of amount of locked VM */
139         pages = (end - start) >> PAGE_SHIFT;
140         if (newflags & VM_LOCKED) {

```


D.4.3.3 Function: `mlock_fixup_start()` (*mm/mlock.c*)

This is slightly more complicated. A new VMA is required to represent the affected region. The start of the old VMA is moved forward.

```

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 Allocates a VMA from the slab allocator for the affected region.

31-34 Copies in the necessary information.

35-36 If the VMA has a `%o` or device mapping, `get_file()` will increment the reference count.

37-38 If an `open()` function is provided, this calls it.

39 Updates the `o` set within the `%o` or device mapping for the old VMA to be the end of the locked region.

40 `lock_vma_mappings()` will lock any `%s` if this VMA is a shared region.

41-44 Locks the parent `mm_struct`, updates its start to be the end of the affected region, inserts the new VMA into the processes linked lists (See Section D.2.2.1) and releases the lock.

45 Unlocks the `%o` mappings with `unlock_vma_mappings()`.

71 Unlocks the mappings with `unlock_vma_mappings()`.

72 Returns success.

D.4.3.5 Function: `mlock_fixup_middle()` (*mm/mlock.c*)

This is similar to the previous two `fixup` functions except that two new regions are required to `fixup` the mapping.

```

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;
97     if (vma->vm_file)
98         atomic_add(2, &vma->vm_file->f_count);
99
100     if (vma->vm_ops && vma->vm_ops->open) {
101         vma->vm_ops->open(left);
102         vma->vm_ops->open(right);
103     }
104     vma->vm_raend = 0;
105     vma->vm_pgoff += (start - vma->vm_start) >> PAGE_SHIFT;
106     lock_vma_mappings(vma);
107     spin_lock(&vma->vm_mm->page_table_lock);
108     vma->vm_start = start;
109     vma->vm_end = end;
110     vma->vm_flags = newflags;
111     __insert_vm_struct(current->mm, left);
112     __insert_vm_struct(current->mm, right);

```


D.5 Page Faulting

Contents

D.5 Page Faulting	328
D.5.1 x86 Page Fault Handler	328
D.5.1.1 Function: <code>do_page_fault()</code>	328
D.5.2 Expanding the Stack	337
D.5.2.1 Function: <code>expand_stack()</code>	337
D.5.3 Architecture-Independent Page Fault Handler	338
D.5.3.1 Function: <code>handle_mm_fault()</code>	339
D.5.3.2 Function: <code>handle_pte_fault()</code>	340
D.5.4 Demand Allocation	341
D.5.4.1 Function: <code>do_no_page()</code>	341
D.5.4.2 Function: <code>do_anonymous_page()</code>	344
D.5.5 Demand Paging	346
D.5.5.1 Function: <code>do_swap_page()</code>	346
D.5.5.2 Function: <code>can_share_swap_page()</code>	350
D.5.5.3 Function: <code>exclusive_swap_page()</code>	351
D.5.6 Copy On Write (COW) Pages	352
D.5.6.1 Function: <code>do_wp_page()</code>	352

This section deals with the page fault handler. It begins with the architecture-specific function for the x86 and then moves to the architecture-independent layer. The architecture-specific functions all have the same responsibilities.

D.5.1 x86 Page Fault Handler

D.5.1.1 Function: `do_page_fault()` (*arch/i386/mm/fault.c*)

The call graph for this function is shown in Figure 4.11. This function is the x86 architecture-dependent function for the handling of page fault exception handlers. Each architecture registers its own, but all of them have similar responsibilities.

```

140 asmlinkage void do_page_fault(struct pt_regs *regs,
                               unsigned long error_code)
141 {
142     struct task_struct *tsk;
143     struct mm_struct *mm;
144     struct vm_area_struct * vma;
145     unsigned long address;
146     unsigned long page;
147     unsigned long fixup;
148     int write;
149     siginfo_t info;
150
151     /* get the address */
152     __asm__("movl %%cr2,%0":"=r" (address));
153
154     /* It's safe to allow irq's after cr2 has been saved */
155     if (regs->eflags & X86_EFLAGS_IF)

```



```

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;

```

If the fault is in userspace, this block finds the VMA for the faulting address and determines if it is a good area, a bad area or if the fault occurred near a region that can be expanded, such as the stack.

186 Takes the long-lived mm semaphore.

188 Finds the VMA that is responsible or is closest to the faulting address.

189-190 If a VMA does not exist at all, goto bad_area.

191-192 If the start of the region is before the address, it means this VMA is the correct VMA for the fault, so goto good_area, which will check the permissions.

193-194 For the region that is closest, this checks if it can grow down (VM_GROWSDOWN). If it does, it means the stack can probably be expanded. If not, goto bad_area.

195-204 Checks to make sure it is not an access below the stack. If the error_code is 4, it means it is running in userspace.

205-206 The stack is the only region with VM_GROWSDOWN set, so, if we reach here, the stack is expanded with expand_stack()(See Section D.5.2.1). If it fails, goto 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 */

```



```

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;

```

At this point, an attempt is going to be made to handle the fault gracefully with `handle_mm_fault()`.

239 Calls `handle_mm_fault()` with the relevant information about the fault. This is the architecture-independent part of the handler.

240-242 A return of 1 means it was a minor fault. Updates statistics.

243-245 A return of 2 means it was a major fault. Update statistics

246-247 A return of 0 means some I/O error happened during the fault, so it goes to the `do_sigbus` handler.

248-249 Any other return means memory could not be allocated for the fault, so we are out of memory. In reality, this does not happen because another function `out_of_memory()` is invoked in `mm/oom_kill.c` before this could happen, which is a function that is a lot more graceful about who it kills.

260 Releases the lock to the mm.

261 Returns because the fault has been successfully handled.

```

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;

```



```

300     regs->eip = fixup;
301     return;
302 }

```

299-302 Searches the exception table with `search_exception_table()` to see if this exception be handled, and, if so, it calls the proper exception handler after returning. This is really important during `copy_from_user()` and `copy_to_user()` when an exception handler is installed to trap reads and writes to invalid regions in userspace without having to make expensive checks. It means that a small `fixup` block of code can be called rather than falling through to the next block, which causes an oops.

```

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");
314     else
315         printk(KERN_ALERT "Unable to handle kernel paging
316                     request");
317     printk(" at virtual address %08lx\n", address);
318     printk(" printing eip:\n");
319     printk("%08lx\n", regs->eip);
320     asm("movl %%cr3,%%0":"=r" (page));
321     page = ((unsigned long *) __va(page))[address >> 22];
322     printk(KERN_ALERT "*pde = %08lx\n", page);
323     if (page & 1) {
324         page &= PAGE_MASK;
325         address &= 0x003ff000;
326         page = ((unsigned long *)
327                 __va(page))[address >> PAGE_SHIFT];
328         printk(KERN_ALERT "*pte = %08lx\n", page);
329     }
330     die("Oops", regs, error_code);
331     bust_spinlocks(0);
332     do_exit(SIGKILL);

```

This is the `no_context` handler. Some bad exception occurred, which is going to end up in the process being terminated in all likelihood. Otherwise, the kernel faulted when it ~~definitely~~ should have, and an oops report is generated.

309-329 Otherwise, the kernel faulted when it really should not have, and it is a kernel bug. This block generates an oops report.


```

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 Frees the mm lock.

353-359 Fills in information to show a SIGBUS occurred at the faulting address so that a debugger can trap it later.

360 Sends the signal.

363-364 If in kernel mode, this tries and handles the exception during no_context.

365 If it is in userspace, this just returns, and the process will die in due course.

```

367 vmalloc_fault:
368     {
369         int offset = __pgd_offset(address);
370         pgd_t *pgd, *pgd_k;
371         pmd_t *pmd, *pmd_k;
372         pte_t *pte_k;
373
374         asm("movl %%cr3,%0":"=r" (pgd));
375         pgd = offset + (pgd_t *)__va(pgd);
376         pgd_k = init_mm.pgd + offset;
377
378         if (!pgd_present(*pgd_k))
379             goto no_context;
380         set_pgd(pgd, *pgd_k);
381
382         pmd = pmd_offset(pgd, address);
383         pmd_k = pmd_offset(pgd_k, address);
384         if (!pmd_present(*pmd_k))
385             goto no_context;
386         set_pmd(pmd, *pmd_k);
387
388         pte_k = pte_offset(pmd_k, address);
389         if (!pte_present(*pte_k))
390             goto no_context;

```



```

648     */
649     address &= PAGE_MASK;
650     spin_lock(&vma->vm_mm->page_table_lock);
651     grow = (vma->vm_start - address) >> PAGE_SHIFT;
652     if (vma->vm_end - address >
        current->rlim[RLIMIT_STACK].rlim_cur ||
653     ((vma->vm_mm->total_vm + grow) << PAGE_SHIFT) >
        current->rlim[RLIMIT_AS].rlim_cur) {
654         spin_unlock(&vma->vm_mm->page_table_lock);
655         return -ENOMEM;
656     }
657     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 Rounds the address down to the nearest page boundary.

650 Locks the pagetables spinlock.

651 Calculates how many pages the stack needs to grow by.

652 Checks to make sure that the size of the stack does not exceed the process limits.

653 Checks to make sure that the size of the address space will not exceed process limits after the stack is grown.

654-655 If either of the limits are reached, this returns -ENOMEM, which will cause the faulting process to segfault.

657-658 Grows the VMA down.

659 Updates the amount of address space used by the process.

660-661 If the region is locked, this updates the number of locked pages used by the process.

662-663 Unlocks the process pagetables and returns success.

D.5.3 Architecture-Independent Page Fault Handler

This is the top-level pair of functions for the architecture-independent page fault handler.

1380 If the pmd has been successfully allocated, then...

1381 Allocates a PTE for this address if one does not already exist.

1382-1383 Handles the page fault with `handle_pte_fault()` (See Section D.5.3.2) and returns the status code.

1385 Failure path and unlocks the `mm_struct`.

1386 Returns -1, which will be interpreted as an out of memory condition. This is correct because this line is only reached if a PMD or PTE could not be allocated.

D.5.3.2 Function: `handle_pte_fault()` (*mm/memory.c*)

This function decides what type of fault this is and which function should handle it. `do_no_page()` is called if this is the first time a page is to be allocated. `do_swap_page()` handles the case where the page was swapped out to disk with the exception of pages swapped out from tmpfs. `do_wp_page()` breaks COW pages. If none of them are appropriate, the PTE entry is simply updated. If it was written to, it is marked dirty, and it is marked accessed to show it is a young page.

```

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);
1347         return do_swap_page(mm, vma, address, pte, entry,
1348                             write_access);
1349     }
1350     if (write_access) {
1351         if (!pte_write(entry))
1352             return do_wp_page(mm, vma, address, pte, entry);
1353         entry = pte_mkdirty(entry);
1354     }

```


~~%~~ Add the page to the pagetable entries and call the appropriate architecture-dependent hooks.

```

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);
1254     spin_unlock(&mm->page_table_lock);
1255     new_page = vma->vm_ops->nopage(vma, address & PAGE_MASK, 0);
1256     if (new_page == NULL) /* no page was available -- SIGBUS */
1257         return 0;
1258     if (new_page == NOPAGE_OOM)
1259         return -1;
1260

```

1245 The parameters supplied are the same as those for `handle_pte_fault()`.

1251-1252 If no `vm_ops` is supplied or no `nopage()` function is supplied, then this calls `do_anonymous_page()` (See Section D.5.4.2) to allocate a page and return it.

1253 Otherwise, this frees the pagetable lock because the `nopage()` function cannot be called with spinlocks held.

1255 Calls the supplied `nopage` function. In the case of ~~%~~ systems, this is frequently `filemap_nopage()` (See Section D.6.4.1), but will be different for each device driver.

1257-1258 If `NULL` is returned, it means some error occurred in the `nopage` function, such as an I/O error while reading from disk. In this case, 0 is returned which results in a `SIGBUS` being sent to the faulting process.

1259-1260 If `NOPAGE_OOM` is returned, the physical page allocator failed to allocate a page, and -1 is returned, which will forcibly kill the process.

```

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         }

```


- 1277 Locks the pagetables again because the allocations have finished and the pagetables are about to be updated.
- 1289 Checks if there is still no PTE in the entry we are about to use. If two faults hit here at the same time, it is possible another processor has already completed the page fault and that this one should be backed out.
- 1290-1297 If there is no PTE entered, this completes the fault.
- 1290 Increases the RSS count because the process is now using another page. A check really should be made here to make sure it isn't the global zero page because the RSS count could be misleading.
- 1291 As the page is about to be mapped to the process space, it is possible for some architectures that write to the page in kernel space will not be visible to the process. `flush_page_to_ram()` ensures the CPU cache will be coherent.
- 1292 `flush_icache_page()` is similar in principle except it ensures the icache and dcaches are coherent.
- 1293 Creates a `pte_t` with the appropriate permissions.
- 1294-1295 If this is a write, then this makes sure the PTE has write permissions.
- 1296 Places the new PTE in the process pagetables.
- 1297-1302 If the PTE is already valid, the page acquired from the `nopage()` function must be released.
- 1299 Decrements the reference count to the page. If it drops to 0, it will be freed.
- 1300-1301 Releases the `mm_struct` lock and returns 1 to signal this is a minor page fault because no major work had to be done for this fault because it was all done by the winner of the race.
- 1305 Updates the MMU cache for architectures that require it.
- 1306-1307 Releases the `mm_struct` lock and returns 2 to signal this is a major page fault.

D.5.4.2 Function: `do_anonymous_page()` (*mm/memory.c*)

This function allocates a new page for a process accessing a page for the first time. If it is a read access, a systemwide page containing only zeros is mapped into the process. If it is write, a zero-filled page is allocated and placed within the pagetables.

```
1190 static int do_anonymous_page(struct mm_struct * mm,
                               struct vm_area_struct * vma,
                               pte_t *page_table, int write_access,
                               unsigned long addr)
```


- 1195 For read accesses, this simply maps the systemwide `empty_zero_page`, which the `ZERO_PAGE()` macro returns with the given permissions. The page is write protected so that a write to the page will result in a page fault.
- 1198-1220 If this is a write fault, it allocates a new page and zero-~~ies~~^{fills} it.
- 1202 Unlocks the `mm_struct` so the allocation of a new page could sleep.
- 1204 Allocates a new page.
- 1205 If a page could not be allocated, this returns -1 to handle the OOM situation.
- 1207 Zero-~~ies~~^{fills} the page.
- 1209 Reacquires the lock because the pagetables are to be updated.
- 1215 Updates the RSS for the process. Note that the RSS is not updated if it is the global zero page being mapped as is the case with the read-only fault at line 1195.
- 1216 Ensures the cache is coherent.
- 1217 Marks the PTE writable and dirty because it has been written to.
- 1218 Adds the page to the LRU list so that it may be reclaimed by the swapper later.
- 1219 Marks the page accessed, which ensures the page is marked hot and on the top of the active list.
- 1222 Fixes the PTE in the pagetables for this process.
- 1225 Updates the MMU cache if the architecture needs it.
- 1226 Frees the pagetable lock.
- 1227 Returns as a minor fault. Even though it is possible the page allocator spent time writing out pages, data did not have to be read from disk to ~~be~~^{fill} this page.

D.5.5 Demand Paging

D.5.5.1 Function: `do_swap_page()` (*mm/memory.c*)

The call graph for this function is shown in Figure 4.15. This function handles the case where a page has been swapped out. A swapped-out page may exist in the swap cache if it is shared between a number of processes or recently swapped in during readahead. This function is broken up into three parts:

- ~~1~~¹ Search for the page in swap cache.
- ~~2~~² If it does not exist, call `swpin_readahead()` to read in the page.

page_cluster variable in mm/swap.c, which is initialized to 2 on machines with less than 16MiB of memory and 3 otherwise. $2^{\text{page\_cluster}}$ pages are read in after the requested page unless a bad or empty page entry is encountered.

1130 read_swap_cache_async() (See Section K.3.1.1) will look up the requested page and read it from disk if necessary.

1131-1141 If the page does not exist, there was another fault that swapped in this page and removed it from the cache while spinlocks were dropped.

1137 Locks the mm_struct.

1138 Compares the two PTEs. If they do not match, -1 is returned to signal an I/O error. If not, 1 is returned to mark a minor page fault because a disk access was not required for this particular page.

1139-1140 Frees the mm_struct and returns the status.

1144 The disk had to be accessed to mark that this is a major page fault.

```

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
```


1181-1182 Unlocks the `mm_struct` and returns whether it was a minor or major page fault.

D.5.5.2 Function: `can_share_swap_page()` (*mm/swap.c*)

This function determines if the swap cache entry for this page may be used or not. It may be used if there is no other references to it. Most of the work is performed by `exclusive_swap_page()`, but this function ~~not~~ makes a few basic checks to avoid having to acquire too many locks.

```

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         retval = 1;
279     }
280     return retval;
281 }
```

263-264 This function is called from the fault path, and the page must be locked.

265 Switch is based on the number of references.

266-268 If the count is 3, but there are no buffers associated with it, there is more than one process using the page. Buffers may be associated for just one process if the page is backed by a swap ~~file~~ instead of a partition.

270-273 If the count is only two, but it is not a member of the swap cache, then it has no slot that may be shared, so it returns false. Otherwise, it performs a full check with `exclusive_swap_page()` (See Section D.5.5.3).

276-277 If the page is reserved, it is the global `ZERO_PAGE`, so it cannot be shared. Otherwise, this page is ~~definitely~~ the only one.

D.5.6 Copy On Write (COW) Pages

D.5.6.1 Function: `do_wp_page()` (*mm/memory.c*)

The call graph for this function is shown in Figure 4.16. This function handles the case where a user tries to write to a private page shared among processes, such as what happens after `fork()`. Basically what happens is a page is allocated, the contents are copied to the new page and the shared count is decremented in the old page.

```

948 static int do_wp_page(struct mm_struct *mm,
                        struct vm_area_struct * vma,
949                        unsigned long address, pte_t *page_table, pte_t pte)
950 {
951     struct page *old_page, *new_page;
952
953     old_page = pte_page(pte);
954     if (!VALID_PAGE(old_page))
955         goto bad_wp_page;
956
```

948-950 The parameters are the same as those supplied to `handle_pte_fault()`.

953-955 Gets a reference to the current page in the PTE and makes sure it is valid.

```

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,
                           pte_mkyoung(pte_mkdirty(pte_mkwrite(pte))));
963             spin_unlock(&mm->page_table_lock);
964             return 1;          /* Minor fault */
965         }
966     }
```

957 First tries to lock the page. If 0 is returned, it means the page was previously unlocked.

958 If we managed to lock it, this calls `can_share_swap_page()` (See Section D.5.5.2) to see if we are the exclusive user of the swap slot for this page. If we are, it means that we are the last process to break COW and that we can simply use this page rather than allocating a new one.

960-965 If we are the only users of the swap slot, it means we are the only user of this page and are the last process to break COW. Therefore, the PTE is simply re-established, and we return a minor fault.

for in the RSS. If this was a normal page, the process would be using the same number of physical frames after the fault as it was before, but, against the zero page, it will be using a new frame, so `rss++` reflects the use of a new page.

986 `break_cow()` is responsible for calling the architecture hooks to ensure the CPU cache and TLBs are up to date and then establishes the new page into the PTE. It ~~it~~ calls `flush_page_to_ram()`, which must be called when a struct `page` is about to be placed in userspace. Next is `flush_cache_page()`, which ~~flushes~~ the page from the CPU cache. Last is `establish_pte()`, which establishes the new page into the PTE.

987 Adds the page to the LRU lists.

992 Releases the spinlock.

993-994 Drops the references to the pages.

995 Returns a minor fault.

```

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 This is a false COW break, which will only happen with a buggy kernel. It prints out an informational message and returns.

1001-1003 The page allocation failed, so this releases the reference to the old page and returns -1.


```

1696 {
1697     ssize_t retval;
1698
1699     if ((ssize_t) count < 0)
1700         return -EINVAL;
1701
1702     if (filp->f_flags & O_DIRECT)
1703         goto o_direct;
1704
1705     retval = -EFAULT;
1706     if (access_ok(VERIFY_WRITE, buf, count)) {
1707         retval = 0;
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;

```

This block is concerned with normal ~~%o~~ I/O.

1702-1703 If this is direct I/O, it jumps to the `o_direct` label.

1706 If the access permissions to write to a userspace page are ok, then this proceeds.

1709 If the count is 0, there is no I/O to perform.

1712-1715 Populates a `read_descriptor_t` structure, which will be used by `file_read_actor()` (See Section L.3.2.3).

1716 Performs the ~~%o~~ read.

1718 Extracts the number of bytes written from the read descriptor struct.

1719-1720 If an error occurred, this extracts what the error was.

D.6.1.2 Function: `do_generic_file_read()` (*mm/~~fs~~map.c*)

This is the core part of the generic ~~%~~ read operation. It is responsible for allocating a page if it doesn't already exist in the page cache. If it does, it must make sure the page is up to date, and it is responsible for making sure that the appropriate readahead window is set.

```

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 Gets the maximum readahead window size for this block device.

1360 Calculates the page index, which holds the current % position pointer.

1361 Calculates the offset within the page that holds the current % position
    pointer.

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 ||
         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 {

```



```

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 This loop goes through each of the pages necessary to satisfy the read request.

1406 Calculates where the end of the `%o` is in pages.

1408-1409 If the current index is beyond the end, then this breaks out because we are trying to read beyond the end of the `%o`.

1410-1417 Calculates `nr` to be the number of bytes remaining to be read in the current page. The block takes into account that this might be the last page used by the `%o` and where the current `%o` position is within the page.

1422-1425 Searches for the page in the page cache.

1426-1427 If the page is not in the page cache, goto `no_cached_page` where it will be allocated.

```

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);

```

In this block, the page was found in the page cache.

1429 Takes a reference to the page in the page cache so it does not get freed prematurely.

1432-1433 If the page is not up to date, goto `page_not_up_to_date` to update the page with information on the disk.

1434 Performs `%o` readahead with `generic_file_readahead()` (See Section D.6.1.3).


```

1461 Updates the current o set within the %o
1462 Moves to the next page if necessary.
1463 Updates the o set within the page we are currently reading. Remember that
    we could have just crossed into the next page in the %o
1465 Releases our reference to this page.
1466-1468 If there is still data to be read, this loops again to read the next page.
    Otherwise, it breaks because the read operation is complete.

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     }

```

In this block, the page being read was not up to date with information on the disk. `generic_file_readahead()` is called to update the current page and readahead because I/O is required anyway.

- 1474 Calls `generic_file_readahead()` (See Section D.6.1.3) to sync the current page and readahead if necessary.
- 1476-1477 If the page is now up to date, `goto page_ok` to start copying the bytes to userspace.

1508 Otherwise, an error occurred, so this sets -EIO to be returned to userspace.

1512-1514 An I/O error occurred, so this records it and releases the reference to the current page. This error will be picked up from the `read_descriptor_t` struct by `generic_file_read()` (See Section D.6.1.1).

```

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 }
```

In this block, the page does not exist in the page cache, so it allocates one and adds it.

1523-1539 If a cache page has not already been allocated, then allocate one and


```

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 Gets the index to start from based on the supplied page.

1230 Gets the maximum-sized readahead for this block device.

1232 Gets the index, in pages, of the end of the file to.

1234 Gets the end of the readahead window from the struct file.

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 ||
            index >= raend ||
            index + filp->f_rawin < raend) {
1247             raend = index;
1248             if (raend < end_index)
1249                 max_ahead = filp->f_ramax;
1250             filp->f_rawin = 0;
1251             filp->f_ralen = 1;
1252             if (!max_ahead) {
1253                 filp->f_raend = index + filp->f_ralen;
1254                 filp->f_rawin += filp->f_ralen;
1255             }
1256         }
1257     }

```

This block has encountered a page that is locked, so it must decide whether to temporarily disable readahead.

1245 If the current page is locked for I/O, then check if the current page is within the last readahead window. If it is, there is no point trying to readahead again. If it is not or readahead has not been performed previously, update the readahead context.


```

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;
1297     }

```

This block performs the actual readahead by calling `page_cache_read()` for each of the pages in the readahead window. Note here how `ahead` is incremented for each page that is readahead.

```

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 }

```


2258-2259 Makes sure an `a_ops → readpage()` function exists.

2260 Updates the access time for the inode.

2261 Uses `generic_file_vm_ops` for the `%o` operations. The generic VM operations structure, defined in `mm/filemap.c`, only supplies `filemap_nopage()` (See Section D.6.4.1) as its `nopage()` function. No other callback is defined.

D.6.3 Generic File Truncation

This section covers the path where a `%o` is being truncated. The actual system call `truncate()` is implemented by `sys_truncate()` in `fs/open.c`. By the time the top-level function in the VM is called (`vmtruncate()`), the dentry information for the `%o` has been updated, and the inode's semaphore has been acquired.

D.6.3.1 Function: `vmtruncate()` (`mm/memory.c`)

This is the top-level VM function responsible for truncating a `%o`. When it completes, all pagetable entries mapping pages that have been truncated have been unmapped and reclaimed if possible.

```

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;

```


- 1066-1071 If the `%s` is being expanded, this makes sure that the process limits for maximum `%s` size are not being exceeded and that the hosting `%s` system is able to support the new `%s` size.
- 1072 If the limits are `%s`, this updates the inodes size and falls through to call the `%s` system-specific `truncate` function, which will `%s` the expanded `%s` size with zeros.
- 1075-1079 If the `%s` system provides a `truncate()` function, then this locks the kernel, calls it and unlocks the kernel again. Filesystems do not acquire the proper locks to prevent races between `%s` truncation and `%s` expansion due to writing or faulting so the big kernel lock is needed.
- 1080 Returns success.
- 1082-1084 If the `%s` size grows too big, this sends the `SIGXFSZ` signal to the calling process and returns `-EFBIG`.

D.6.3.2 Function: `vmtruncate_list()` (*mm/memory.c*)

This function cycles through all VMAs in an `address_spaces` list and calls `zap_page_range()` for the range of addresses that map a `%s` that is being truncated.

```

1006 static void vmtruncate_list(struct vm_area_struct *mpnt,
                               unsigned long pgoff)
1007 {
1008     do {
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;

```



```

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 Calculates the start and end address for zapping.

367 Calculates the PGD (dir) that contains the starting address.

376-377 Makes sure the start address is not after the end address.

378 Acquires the spinlock protecting the page tables. This is a very longheld lock and would normally be considered a bad idea, but the comment prior to the block explains why it is ok in this case.

379 Flushes the CPU cache for this range.

380 tlb_gather_mmu() records the MM that is being altered. Later, tlb_remove_page() will be called to unmap the PTE that stores the PTEs in a struct free_pte_ctx until the zapping is finished. This is to avoid having to constantly flush the TLB as PTEs are freed.

382-386 For each PMD affected by the zapping, this calls zap_pmd_range() until the end address has been reached. Note that tlb is passed as well for tlb_remove_page() to use later.

389 tlb_finish_mmu() frees all the PTEs that were unmapped by tlb_remove_page() and then flushes the TLBs. Doing the flushing this way avoids a storm of TLB flushing that would be otherwise required for each PTE unmapped.

D.6.3.5 Function: zap_pte_range() (*mm/memory.c*)

This function calls `tlb_remove_page()` for each PTE in the requested pmd within the requested address range.

```

294 static inline int zap_pte_range(mmu_gather_t *tlb, pmd_t * pmd,
                                unsigned long address,
                                unsigned long size)
295 {
296     unsigned long offset;
297     pte_t * ptep;
298     int freed = 0;
299
300     if (pmd_none(*pmd))
301         return 0;
302     if (pmd_bad(*pmd)) {
303         pmd_ERROR(*pmd);
304         pmd_clear(pmd);
305         return 0;
306     }
307     ptep = pte_offset(pmd, address);
308     offset = address & ~PMD_MASK;
309     if (offset + size > PMD_SIZE)
310         size = PMD_SIZE - offset;
311     size &= PAGE_MASK;
312     for (offset=0; offset < size; ptep++, offset += PAGE_SIZE) {
313         pte_t pte = *ptep;
314         if (pte_none(pte))
315             continue;
316         if (pte_present(pte)) {
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 If the PMD does not exist, this returns.

302-306 If the PMD is bad, it logs the error and returns.

307 Gets the starting PTE offset.

- 330 Calculates `partial` as an offset within the last page if it is being partially truncated.
- 333 Locks the page cache.
- 334 This will loop until none of the calls to `truncate_list_pages()` returns that a page was found that should have been reclaimed.
- 335 Uses `truncate_list_pages()` (See Section D.6.3.7) to truncate all pages in the `clean_pages` list.
- 336 Similarly, truncates pages in the `dirty_pages` list.
- 337 Similarly, truncates pages in the `locked_pages` list.
- 340 Unlocks the page cache.

D.6.3.7 Function: `truncate_list_pages()` (*mm/mmap.c*)

This function searches the requested list (`head`), which is part of an `address_space`. If pages are found after `start`, they will be truncated.

```

259 static int truncate_list_pages(struct list_head *head,
                                unsigned long start,
                                unsigned *partial)
260 {
261     struct list_head *curr;
262     struct page *page;
263     int unlocked = 0;
264
265     restart:
266     curr = head->prev;
267     while (curr != head) {
268         unsigned long offset;
269
270         page = list_entry(curr, struct page, list);
271         offset = page->index;
272
273         /* Is one of the pages to truncate? */
274         if ((offset >= start) ||
            (*partial && (offset + 1) == start)) {
275             int failed;
276
277             page_cache_get(page);
278             failed = TryLockPage(page);
279
280             list_del(head);
281             if (!failed)
282                 /* Restart after this page */

```


- 284-286 If not, it adds it back where it will be found again immediately. Later in the function, `wait_on_page()` is called until the page is unlocked.
- 288 Releases the pagecache lock.
- 299 Sets `locked` to 1 to indicate a page was found that had to be truncated. This will force `truncate_inode_pages()` to call this function again to make sure there are no pages left behind. This looks like an oversight and was intended to have the functions recalled only if a *locked* page was found. However the way it is implemented means that it will be called whether the page was locked or not.
- 291-299 If we locked the page, this truncates it.
- 292-294 If the page is to be partially truncated, this calls `truncate_partial_page()` (See Section D.6.3.10) with the `o` set within the page where the truncation begins (partial).
- 296 If not, it calls `truncate_complete_page()` (See Section D.6.3.8) to truncate the whole page.
- 298 Unlocks the page.
- 300 If the page locking failed, this calls `wait_on_page()` to wait until the page can be locked.
- 302 Releases the reference to the page. If there are no more mappings for the page, it will be reclaimed.
- 304-307 Checks if the process should call `schedule()` before continuing. This is to prevent a truncating process from hogging the CPU.
- 309 Reacquires the spinlock and restarts the scanning for pages to reclaim.
- 312 The current page should not be reclaimed, so this moves to the next page.
- 314 Returns 1 if a page was found in the list that had to be truncated.

D.6.3.8 Function: `truncate_complete_page()` (*mm/bmap.c*)

This function truncates a full page, frees associated resources and reclaims the page.

```

239 static void truncate_complete_page(struct page *page)
240 {
241     /* Leave it on the LRU if it gets converted into
       * anonymous buffers */
242     if (!page->buffers || do_flushpage(page, 0))
243         lru_cache_del(page);
244
245     /*
246     * We remove the page from the page cache _after_ we have

```



```

233 {
234     memclear_highpage_flush(page, partial, PAGE_CACHE_SIZE-partial);
235     if (page->buffers)
236         do_flushpage(page, partial);
237 }

```

234 `memclear_highpage_flush()` ~~is~~ an address range with zeros. In this case, it will zero from `partial` to the end of the page.

235-236 If the page has any associated buffers, this ~~is~~ flushes any buffers containing data in the truncated region.

D.6.4 Reading Pages for the Page Cache

D.6.4.1 Function: `filemap_nopage()` (*mm/~~filemap~~.c*)

This is the generic `nopage()` function used by many VMAs. This loops around itself with a large number of `goto`s which can be difficult to trace, but there is nothing novel here. It is principally responsible for fetching the faulting page from either the pagecache or reading it from disk. If appropriate, it will also perform ~~the~~ readahead.

```

1994 struct page * filemap_nopage(struct vm_area_struct * area,
                                unsigned long address,
                                int unused)
1995 {
1996     int error;
1997     struct file *file = area->vm_file;
1998     struct address_space *mapping =
1999         file->f_dentry->d_inode->i_mapping;
2000     struct inode *inode = mapping->host;
2001     struct page *page, **hash;
2002     unsigned long size, pgoff, endoff;
2003     pgoff = ((address - area->vm_start) >> PAGE_CACHE_SHIFT) +
2004         area->vm_pgoff;
2005     endoff = ((area->vm_end - area->vm_start) >> PAGE_CACHE_SHIFT) +
2006         area->vm_pgoff;
2007 }

```

This block acquires the `struct file`, `address_space` and `inode`, which are important for this page fault. It then acquires the starting `o` set within the ~~the~~ needed for this fault and the `o` set that corresponds to the end of this VMA. The `o` set is the end of the VMA instead of the end of the page in case ~~the~~ readahead is performed.

1997-1999 Acquires the `struct file`, `address_space` and `inode` required for this fault.

2032-2033 If the page is not up to date, goto `page_not_uptodate` where the page will either be declared invalid or the data in the page will be updated.

2035 `success:`

```
2036     /*
2037     * Try read-ahead for sequential areas.
2038     */
2039     if (VM_SequentialReadHint(area))
2040         nopage_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 If this mapping specifies the `VM_SEQ_READ` hint, the pages of the current fault will be prefaulted with `nopage_sequential_readahead()`.

2046 Marks the faulted-in page as accessed, so it will be moved to the `active_list`.

2047 As the page is about to be installed into a process page table, this calls `flush_page_to_ram()` so that recent stores by the kernel to the page will definitely be visible to userspace.

2048 Returns the faulted-in page.

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, we're 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.

```



```

2099         goto success;
2100     }

```

In this block, the page was found, but it was not up to date so the reasons for the page not being up to date are checked. If it looks ok, the appropriate `readpage()` function is called to resync the page.

2081 Locks the page for I/O.

2084-2088 If the page was removed from the mapping (possible because of a %o truncation) and is now anonymous, then goto `retry_all`, which will try and fault in the page again.

2090-2094 Checks again for the `Uptodate` %o in case the page was updated just before we locked the page for I/O.

2096 Calls the `address_space → readpage()` function to schedule the data to be read from disk.

2097 Waits for the I/O to complete and if it is now up to date, goto `success` to return the page. If the `readpage()` function failed, it falls through to the error recovery path.

```

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))

```



```

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 Acquires the `address_space` mapping managing the `%s`.

705 The page cache is a hash table, and `page_hash()` returns the `%s` page in the bucket for this mapping and offset.

708-709 Searches the page cache with `__find_page_nolock()` (See Section J.1.4.3). This basically will traverse the list starting at hash to see if the requested page can be found.

711-712 If the page is already in the page cache, this returns.

714 Allocates a new page for insertion into the page cache. `page_cache_alloc()` will allocate a page from the buddy allocator using GFP mask information contained in mapping.

718 Inserts the page into the page cache with `add_to_page_cache_unique()` (See Section J.1.1.2). This function is used because a second check needs to be made to make sure the page was not inserted into the page cache while the `pagecache_lock` spinlock was not acquired.

719 If the allocated page was inserted into the page cache, it needs to be populated with data, so the `readpage()` function for the mapping is called. This schedules the I/O to take place, and the page will be unlocked when the I/O completes.

720 The path in `add_to_page_cache_unique()` (See Section J.1.1.2) takes an extra reference to the page being added to the page cache, which is dropped here. The page will not be freed.

727 If another process added the page to the page cache, it is released here by `page_cache_release()` because there will be no users of the page.

D.6.5 File Readahead for `nopage()`

D.6.5.1 Function: `nopage_sequential_readahead()` (*mm/filemap.c*)

This function is only called by `filemap_nopage()` when the `VM_SEQ_READ` flag has been specified in the VMA. When half of the current readahead window has been faulted in, the next readahead window is scheduled for I/O, and pages from the previous window are freed.


```

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 >
          (vma->vm_pgoff + ra_window + ra_window)) {
1973          unsigned long window = ra_window << PAGE_SHIFT;
1974
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 If the fault has occurred halfway through the readahead window, this schedules the next readahead window to be read in from disk and frees the pages for the first half of the current window because they are presumably not required any more.

1954-1955 Calculates the start and end of the next readahead window because we are about to schedule it for I/O.

1957 If the end of the readahead window is after the end of the VMA, this sets end to the end of the VMA.

1959-1960 If we are at the end of the mapping, this just returns because there is no more readahead to perform.

1962-1967 Schedules the next readahead window to be paged in by calling `read_cluster_nonblocking()` (See Section D.6.5.2).

1968 Calls `run_task_queue()` to start the I/O.

1972-1978 Recycles the pages in the previous readahead window with `filemap_sync()` as they are no longer required.

1980 Updates where the end of the readahead window is.

D.6.5.2 Function: `read_cluster_nonblocking()` (*mm/vmap.c*)

This function schedules the next readahead window to be paged in.

```

737 static int read_cluster_nonblocking(struct file * file,
                                     unsigned long offset,
738     unsigned long filesize)
739 {
740     unsigned long pages = CLUSTER_PAGES;

```



```

1107         break;
1108     page_cache_release(new_page);
1109 }
1110 return;
1111 }

```

1102 `valid_swaphandles()` is what determines how many pages should be swapped in. It will stop at the first empty entry or when `CLUSTER_PAGES` is reached.

1103-1109 Swaps in the pages.

1105 Attempts to swap the page into the swap cache with `read_swap_cache_async()` (See Section K.3.1.1).

1106-1107 If the page could not be paged in, this breaks and returns.

1108 Drops the reference to the page that `read_swap_cache_async()` takes.

1110 Returns.

D.6.6.2 Function: `valid_swaphandles()` (*mm/swap.c*)

This function determines how many pages should be readahead from swap starting from offset. It will readahead to the next unused swap slot, but, at most, it will return `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         /* Don't read-ahead past the end of the swap area */
1254         if (toff >= swapdev->max)
1255             break;
1256         /* Don't read in free or bad pages */
1257         if (!swapdev->swap_map[toff])
1258             break;
1259         if (swapdev->swap_map[toff] == SWAP_MAP_BAD)

```


E.1 Initializing the Boot Memory Allocator

Contents

E.1 Initializing the Boot Memory Allocator	396
E.1.1 Function: <code>init_bootmem()</code>	396
E.1.2 Function: <code>init_bootmem_node()</code>	396
E.1.3 Function: <code>init_bootmem_core()</code>	397

The functions in this section are responsible for bootstrapping the boot memory allocator. It starts with the architecture-specific function `setup_memory()` (See Section B.1.1), but all architectures cover the same basic tasks in the architecture-specific function before calling the architecture-independent function `init_bootmem()`.

E.1.1 Function: `init_bootmem()` (*mm/bootmem.c*)

This is called by UMA architectures to initialize their boot memory allocator structures.

```

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 Confusingly, the `pages` parameter is actually the end PFN of the memory addressable by this node, not the number of pages as the name implies.

306 Sets the max PFN addressable by this node in case the architecture-dependent code did not.

307 Sets the min PFN addressable by this node in case the architecture-dependent code did not.

308 Calls `init_bootmem_core()` (See Section E.1.3), which does the real work of initializing the `bootmem_data`.

E.1.2 Function: `init_bootmem_node()` (*mm/bootmem.c*)

This is called by NUMA architectures to initialize boot memory allocator data for a given node.

```

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 }
```


- 55 Rounds the mapsize up to the closest word boundary.
- 56 Converts the mapstart to a virtual address and stores it in `bdata→node_bootmem_map`.
- 57 Converts the starting PFN to a physical address and stores it on `node_boot_start`.
- 58 Stores the end PFN of `ZONE_NORMAL` in `node_low_pfn`.
- 64 Fills the full map with 1s that mark all pages as allocated. It is up to the architecture-dependent code to mark the usable pages.


```

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 The `sidx` is the starting index to serve pages from. The value is obtained by subtracting the starting address from the requested address and dividing by the size of a page.

82 A similar calculation is made for the ending index `eidx` except that the allocation is rounded up to the nearest page. This means that requests to partially reserve a page will result in the full page being reserved.

84 `end` is the last PFN that is affected by this reservation.

86 Checks that a nonzero value has been given.

88-89 Checks that the starting index is not before the start of the node.

90-91 Checks that the end index is not before the start of the node.

92-93 Checks that the starting index is not after the end index.

94-95 Checks that the starting address is not beyond the memory that this bootmem node represents.


```

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;

```

This is the function preamble, which makes sure the parameters are sane.

144 The parameters are the following:

~~%b~~ `bdata` is the bootmem for the struct being allocated from.

~~%s~~ `size` is the size of the requested allocation.

~~%o~~ `align` is the desired alignment for the allocation. It must be a power of 2.

~~%g~~ `goal` is the preferred address to allocate above if possible.

151 Calculates the ending bit index `eidx`, which returns the highest page index that may be used for the allocation.

154 Calls `BUG()` if a request size of 0 is specified.

156-157 If the alignment is not a power of 2, this calls `BUG()`.

159 The default `offset` for alignments is 0.

160 If an alignment has been specified and...

161 The requested alignment is the same alignment as the start of the node, this calculates the `offset` set to use.

162 The `offset` set to use is the requested alignment masked against the lower bits of the starting address. In reality, this `offset` will likely be identical to `align` for the prevalent values of `align`.

```

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;

```


This block scans through memory looking for a large enough block to satisfy this request.

180 If the allocation could not be satisfied starting from `goal`, this label is jumped to so that the map will be rescanned.

181-194 Starting from `preferred`, this scans linearly searching for a free block large enough to satisfy the request. It walks the address space in `incr` steps to satisfy alignments greater than one page. If the alignment is less than a page, `incr` will just be 1.

183-184 Tests the bit. If it is already 1, it is not free, so it moves to the next page.

185-190 Scans the next `areasize` number of pages and sees if they are also free. It fails if the end of the addressable space is reached (`eidx`) or one of the pages is already in use.

191-192 A free block is found, so this records the `start` and jumps to the found block.

195-198 The allocation failed, so it starts again from the beginning.

199 If that also failed, it returns `NULL`, which will result in a kernel panic.

```

200 found:
201     if (start >= eidx)
202         BUG();
203
204     if (align <= PAGE_SIZE
205         && bdata->last_offset && bdata->last_pos+1 == start) {
206         offset = (bdata->last_offset+align-1) & ~(align-1);
207         if (offset > PAGE_SIZE)
208             BUG();
209         remaining_size = PAGE_SIZE-offset;
210         if (size < remaining_size) {
211             areasize = 0;
212             // last_pos unchanged
213             bdata->last_offset = offset+size;
214             ret = phys_to_virt(bdata->last_pos*PAGE_SIZE + offset +
215                             bdata->node_boot_start);
216         } else {
217             remaining_size = size - remaining_size;
218             areasize = (remaining_size+PAGE_SIZE-1)/PAGE_SIZE;
219             ret = phys_to_virt(bdata->last_pos*PAGE_SIZE +
220                             offset +
221                             bdata->node_boot_start);
222         }
223         bdata->last_pos = start+areasize-1;

```


226 The last page used is the `start` page plus the number of additional pages required to satisfy this allocation `areasize`.

227 The end of the allocation has already been calculated.

229 If the `o` set is at the end of the page, this makes it 0.

231 No merging took place, so this records the last page used to satisfy this allocation.

232 Records how much of the last page was used.

233 Records the starting virtual address of the allocation.

```
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 }
```

This block marks the pages allocated as 1 in the bitmap and zeros-out the contents of the pages.

238-240 Cycles through all pages used for this allocation and sets the bit to 1 in the bitmap. If any of them are already 1, a double allocation took place, so it calls `BUG()`.

241 Zero-~~es~~^{es} the pages.

242 Returns the address of the allocation.


```

113     unsigned long end = (addr + size)/PAGE_SIZE;
114
115     if (!size) BUG();
116     if (end > bdata->node_low_pfn)
117         BUG();
118
119     /*
120      * Round up the beginning of the address.
121      */
122     start = (addr + PAGE_SIZE-1) / PAGE_SIZE;
123     sidx = start - (bdata->node_boot_start/PAGE_SIZE);
124
125     for (i = sidx; i < eidx; i++) {
126         if (!test_and_clear_bit(i, bdata->node_bootmem_map))
127             BUG();
128     }
129 }

```

112 Calculates the end index affected as `eidx`.

113 The end address is the end of the affected area rounded down to the nearest page if it is not already page aligned.

115 If a size of 0 is freed, this calls BUG.

116-117 If the end PFN is after the memory addressable by this node, this calls BUG.

122 Rounds the starting address up to the nearest page if it is not already page aligned.

123 Calculates the starting index to free.

125-127 For all full pages that are freed by this action, this clears the bit in the boot bitmap. If it is already 0, it is a double free or is memory that was never used, so it calls BUG.

514 This function records that the PFN high memory starts in `mem_map` (`highmem_start_page`), the maximum number of pages in the system (`max_mapnr` and `num_physpages`) and finally the maximum number of pages that may be mapped by the kernel (`num_mappedpages`).

516 `high_memory` is the virtual address where high memory begins.

519 Zeros-out the systemwide zero page.

520

521 `reservedpages = free_pages_init();`

522

521 Calls `free_pages_init()` (See Section E.4.2), which tells the boot memory allocator to retire itself as well as initializing all pages in high memory for use with the buddy allocator.

523 `codesize = (unsigned long) &_etext - (unsigned long) &_text;`

524 `datasize = (unsigned long) &_edata - (unsigned long) &_etext;`

525 `initssize = (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 `initssize >> 10,`

534 `(unsigned long) (totalhigh_pages << (PAGE_SHIFT-10))`

535 `);`

This block prints out an informational message.

523 Calculates the size of the code segment, data segment and memory used by initialization code and data (all functions marked `__init` will be in this section).

527-535 Prints out a nice message on the availability of memory and the amount of memory consumed by the kernel.

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`

486 There is a bug in the Pentium Pros that prevents certain pages in high memory from being used. The function `ppro_with_ram_bug()` checks for its existence.

489 Calls `free_all_bootmem()` to retire the boot memory allocator.

491-498 Cycles through all of memory and counts the number of reserved pages that were left over by the boot memory allocator.

500-501 For each page in high memory, this calls `one_highpage_init()` (See Section E.4.3). This function clears the `PG_reserved` bit, sets the `PG_high` bit, sets the count to 1, calls `__free_pages()` to give the page to the buddy allocator and increments the `totalhigh_pages` count. Pages that kill buggy Pentium Pros are skipped.

E.4.3 Function: `one_highpage_init()` (*arch/i386/mm/init.c*)

This function initializes the information for one page in high memory and checks to make sure that the page will not trigger a bug with some Pentium Pros. It only exists if `CONFIG_HIGHMEM` is specified at compile time.

```

449 #ifdef CONFIG_HIGHMEM
450 void __init one_highpage_init(struct page *page, int pfn,
                               int bad_ppro)
451 {
452     if (!page_is_ram(pfn)) {
453         SetPageReserved(page);
454         return;
455     }
456
457     if (bad_ppro && page_kills_ppro(pfn)) {
458         SetPageReserved(page);
459         return;
460     }
461
462     ClearPageReserved(page);
463     set_bit(PG_highmem, &page->flags);
464     atomic_set(&page->count, 1);
465     __free_page(page);
466     totalhigh_pages++;
467 }
468 #endif /* CONFIG_HIGHMEM */

```

452-455 If a page does not exist at the PFN, this marks the struct page as reserved, so it will not be used.

457-460 If the running CPU is susceptible to the Pentium Pro bug and this page is a page that would cause a crash (`page_kills_ppro()` performs the check), this marks the page as reserved so that it will never be allocated.


```

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 If no map is available, it means that this node has already been freed and that something woeful is wrong with the architecture-dependent code, so it calls BUG().

254 Keeps running count of the number of pages given to the buddy allocator.

255 idx is the last index that is addressable by this node.

256-263 Cycles through all pages addressable by this node.

257 If the page is marked free, then...

258 Increases the running count of pages given to the buddy allocator.

259 Clears the PG_reserved flag.

260 Sets the count to 1 so that the buddy allocator will think this is the last user of the page and place it in its free lists.

261 Calls the buddy allocator free function so that the page will be added to its free lists.

264 total will give the total number of pages given over by this function.

```

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++;

```


F.1 Allocating Pages

Contents

F.1 Allocating Pages	420
F.1.1 Function: <code>alloc_pages()</code>	420
F.1.2 Function: <code>_alloc_pages()</code>	420
F.1.3 Function: <code>__alloc_pages()</code>	421
F.1.4 Function: <code>rmqueue()</code>	425
F.1.5 Function: <code>expand()</code>	427
F.1.6 Function: <code>balance_classzone()</code>	428

F.1.1 Function: `alloc_pages()` (*include/linux/mm.h*)

The call graph for this function is shown in Figure 6.2. It is declared as follows:

```

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 The `gfp_mask` (Get Free Pages) ~~is~~ tells the allocator how it may behave. For example, if `GFP_WAIT` is not set, the allocator will not block and instead will return `NULL` if memory is tight. The order is the power of two number of pages to allocate.

444-445 A simple debugging check optimized away at compile time.

446 This function is described next.

F.1.2 Function: `_alloc_pages()` (*mm/page_alloc.c*)

The function `_alloc_pages()` comes in two varieties. The ~~first~~ is designed to only work with UMA architectures such as the x86 and is in `mm/page_alloc.c`. It only refers to the static node `contig_page_data`. The second is in `mm/numa.c` and is a simple extension. It uses a node-local allocation policy, which means that memory will be allocated from the bank closest to the processor. For the purposes of this book, only the `mm/page_alloc.c` version will be examined, but developers on NUMA architectures should read `_alloc_pages()` and `_alloc_pages_pgdat()` as well in `mm/numa.c`.

```

244 #ifndef CONFIG_DISCONTIGMEM
245 struct page *_alloc_pages(unsigned int gfp_mask,
                           unsigned int order)
246 {
247     return __alloc_pages(gfp_mask, order,
```



```

348             return page;
349         }
350     }

```

334 Sets the zone to be the preferred zone to allocate from.

335 The preferred zone is recorded as the classzone. If one of the pages_low watermarks is reached later, the classzone is marked as needing balance.

336-337 An unnecessary sanity check. build_zonelists() would need to be seriously broken for this to happen.

338-350 This style of block appears a number of times in this function. It reads as `for (z = classzone; z <= zone; z++)` and see if the allocation can be satisfied without violating watermarks. The pages_low for each fallback zone is added together. This is deliberate to reduce the probability that a fallback zone will be used.

340 z is the zone currently been examined. The zone variable is moved to the next fallback zone.

341-342 If this is the last zone in the fallback list, break.

344 Increments the number of pages to be allocated by the watermark for easy comparisons. This happens for each zone in the fallback zones. Although this appears at first to be a bug, this behavior is actually intended to reduce the probability that a fallback zone is used.

345-349 Allocates the page block if it can be assigned without reaching the pages_min watermark. rmqueue() (See Section F.1.4) is responsible for removing the block of pages from the zone.

347-348 If the pages could be allocated, this returns a pointer to them.

```

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))

```


379-391 These two ~~Yes~~ are only set by the OOM killer. Because the process is trying to kill itself cleanly, this allocates the pages if at all possible because it is known they will be freed very soon.

```

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     /* Don't 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 If the calling process cannot sleep, this returns NULL because the only way to allocate the pages from here involves sleeping.

397 `balance_classzone()` (See Section F.1.6) performs the work of `kswapd` in a synchronous fashion. The principal difference is that, instead of freeing the memory into a global pool, it is kept for the process using the `current->local_pages` linked list.

398-399 If a page block of the right order has been freed, this returns it. Just because this is NULL does not mean an allocation will fail because it could be a higher order of pages that was released.


```

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 The parameters are the zone to allocate from and what order of pages are required.

201 Because the `free_area` is an array of linked lists, the order may be used as an an index within the array.

207 Acquires the zone lock.

208-238 This while block is responsible for finding what order of pages we will need to allocate from. If a free block is not at the order we are interested in, this checks the higher blocks until a suitable one is found.

209 `head` is the list of free page blocks for this order.

210 `curr` is the `%d` block of pages.

212-235 If a free page block is at this order, this allocates it.

215 The page is set to be a pointer to the `%d` page in the free block.

216-217 A sanity check that checks to make sure this page belongs to this zone and is within the `zone_mem_map`. It is unclear how this could possibly happen without severe bugs in the allocator itself that would place blocks in the wrong zones.

218 Because the block is going to be allocated, this removes it from the free list.

219 `index` treats the `zone_mem_map` as an array of pages so that `index` will be the `o` set within the array.


```

193     if (BAD_RANGE(zone, page))
194         BUG();
195     return page;
196 }

```

177 The parameters are the following:

- `zone` is where the allocation is coming from.
- `page` is the ~~low~~ page of the block being split.
- `index` is the index of page within `mem_map`.
- `low` is the order of pages needed for the allocation.
- `high` is the order of pages that is being split for the allocation.
- `area` is the `free_area_t` representing the high order block of pages.

180 `size` is the number of pages in the block that is to be split.

182-192 Keeps splitting until a block of the needed page order is found.

183-184 A sanity check that checks to make sure this page belongs to this zone and is within the `zone_mem_map`.

185 `area` is now the next `free_area_t` representing the lower order of page blocks.

186 `high` is the next order of page blocks to be split.

187 The size of the block being split is now half as big.

188 Of the pair of buddies, the one lower in the `mem_map` is added to the free list for the lower order.

189 Toggles the bit representing the pair of buddies.

190 `index` is now the index of the second buddy of the newly created pair.

191 `page` now points to the second buddy of the newly created pair.

193-194 A sanity check.

195 The blocks have been successfully split, so this returns the page.

F.1.6 Function: `balance_classzone()` (*mm/page_alloc.c*)

This function is part of the direct-reclaim path. Allocators that can sleep will call this function to start performing the work of `kswapd` in a synchronous fashion. Because the process is performing the work itself, the pages it frees of the desired order are reserved in a linked list in `current → local_pages`, and the number of page blocks in the list is stored in `current → nr_local_pages`. Note that page blocks are not the same as number of pages. A page block could be of any order.


```

276
277     if (likely(__freed)) {
278         /* pick from the last inserted so we're likely */
279         entry = local_pages->next;
280         do {
281             tmp = list_entry(entry, struct page, list);
282             if (tmp->index == order &&
                memclass(page_zone(tmp), classzone)) {
283                 list_del(entry);
284                 current->nr_local_pages--;
285                 set_page_count(tmp, 1);
286                 page = tmp;
287
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     }

```

Presuming that pages exist in the `local_pages` list, this function will cycle through the list looking for a page block belonging to the desired zone and order.

270 Only enter this block if pages are stored in the local list.

275 Starts at the beginning of the list.

277 If pages were freed with `try_to_free_pages_zone()`, then...

279 The last one inserted is chosen ~~not~~ because it is likely to be cache hot, and it is desirable to use pages that have been recently referenced.

280-305 Cycles through the pages in the list until we find one of the desired order and zone.

- 314-315 If the count of page blocks reaches zero and pages are still in the list, it means that the accounting is seriously broken somewhere or that someone added pages to the `local_pages` list manually, so it calls `BUG()`.
- 317 Sets the number of page blocks to 0 because they have all been freed.
- 320 Updates the `freed` parameter to tell the caller how many pages were freed in total.
- 321 Returns the page block of the requested order and zone. If the freeing failed, this will be returning `NULL`.

F.2.4 Function: `__get_dma_pages()` (*include/linux/mm.h*)

This is of principal interest to device drivers. It will return memory from ZONE_DMA suitable for use with DMA devices. It is declared as follows:

```
457 #define __get_dma_pages(gfp_mask, order) \
458     __get_free_pages((gfp_mask) | GFP_DMA, (order))
```

458 The `gfp_mask` is or-ed with `GFP_DMA` to tell the allocator to allocate from ZONE_DMA.

F.2.5 Function: `get_zeroed_page()` (*mm/page_alloc.c*)

This function will allocate one page and then zeros out the contents of it. It is declared as follows:

```
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` are the flags that affect allocator behavior.

442 `alloc_pages()` does the work of allocating the page block. See Section F.1.1.

444 `page_address()` returns the physical address of the page.

445 `clear_page()` will zero the contents of a page with zero.

446 Returns the address of the zeroed page.


```

99         if (page->buffers)
100             BUG();
101         if (page->mapping)
102             BUG();
103         if (!VALID_PAGE(page))
104             BUG();
105         if (PageLocked(page))
106             BUG();
107         if (PageActive(page))
108             BUG();
109         page->flags &= ~((1<<PG_referenced) | (1<<PG_dirty));

```

82 The parameters are the beginning of the page block to free and what order number of pages are to be freed.

93-97 A dirty page on the LRU will still have the LRU bit set when pinned for I/O. On I/O completion, it is freed, so it must now be removed from the LRU list.

99-108 Sanity checks.

109 The ~~Yes~~ showing a page has been referenced and is dirty have to be cleared because the page is now free and not in use.

```

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 If this ~~Yes~~ is set, the pages freed are to be kept for the process doing the freeing. This is set by `balance_classzone()` (See Section F.1.6) during page allocation if the caller is freeing the pages itself rather than waiting for `kswapd` to do the work.

115 The zone that the page belongs to is encoded within the page ~~Yes~~. The `page_zone()` macro returns the zone.


```

156         page_idx &= mask;
157     }

```

126 The zone is about to be altered, so this takes out the lock. The lock is an interrupt-safe lock because it is possible for interrupt handlers to allocate a page in this path.

128 Another side effect of the calculation of `mask` is that `-mask` is the number of pages that are to be freed.

130-157 The allocator will keep trying to coalesce blocks together until it either cannot merge or reaches the highest order that can be merged. `mask` will be adjusted for each order block that is merged. When the highest order that can be merged is reached, this while loop will evaluate to 0 and exit.

133-134 If by some miracle, `mask` is corrupt, this check will make sure the `free_area` array will not not be read beyond the end.

135 Toggles the bit representing this pair of buddies. If the bit was previously zero, both buddies were in use. Because this buddy is being freed, one is still in use and cannot be merged.

145-146 The calculation of the two addresses is discussed in Chapter 6.

147-150 A sanity check to make sure the pages are within the correct `zone_mem_map` and actually belong to this zone.

152 The buddy has been freed, so it removes it from any list it was part of.

153-156 Prepares to examine the higher order buddy for merging.

153 Moves the mask one bit to the left for order 2^{k+1} .

154 `area` is a pointer within an array, so `area++` moves to the next index.

155 The index in the bitmap of the higher order.

156 The page index within the `zone_mem_map` for the buddy to merge.

```

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

```


F.4 Free Helper Functions

Contents

F.4 Free Helper Functions	440
F.4.1 Function: <code>free_pages()</code>	440
F.4.2 Function: <code>__free_page()</code>	440
F.4.3 Function: <code>free_page()</code>	440

These functions are very similar to the page allocation helper functions in that they do not really work themselves and depend on the `__free_pages()` function to perform the actual free.

F.4.1 Function: `free_pages()` (*mm/page_alloc.c*)

This function takes an address instead of a page as a parameter to free. It is declared as follows:

```
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 The function is discussed in Section F.3.1. The macro `virt_to_page()` returns the struct `page` for the `addr`.

F.4.2 Function: `__free_page()` (*include/linux/mm.h*)

This trivial macro just calls the function `__free_pages()` (See Section F.3.1) with an order 0 for one page. It is declared as follows:

```
472 #define __free_page(page) __free_pages((page), 0)
```

F.4.3 Function: `free_page()` (*include/linux/mm.h*)

This trivial macro just calls the function `free_pages()`. The essential difference between this macro and `__free_page()` is that this function takes a virtual address as a parameter and `__free_page()` takes a struct `page`. It is declared as follows:

```
472 #define free_page(addr) free_pages((addr),0)
```


G.1 Allocating a Noncontiguous Area

Contents

G.1 Allocating a Noncontiguous Area	442
G.1.1 Function: <code>vmalloc()</code>	442
G.1.2 Function: <code>__vmalloc()</code>	442
G.1.3 Function: <code>get_vm_area()</code>	443
G.1.4 Function: <code>vmalloc_area_pages()</code>	445
G.1.5 Function: <code>__vmalloc_area_pages()</code>	446
G.1.6 Function: <code>alloc_area_pmd()</code>	447
G.1.7 Function: <code>alloc_area_pte()</code>	448
G.1.8 Function: <code>vmap()</code>	450

G.1.1 Function: `vmalloc()` (*include/linux/vmalloc.h*)

The call graph for this function is shown in Figure 7.2. The following macros differ only by the GFP_ flags (See Section 6.4) used. The size parameter is page aligned by `__vmalloc()` (See Section G.1.2).

```

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 }

```

37 The flags indicate to use either `ZONE_NORMAL` or `ZONE_HIGHMEM` as necessary.

46 The flag indicates to only allocate from `ZONE_DMA`.

55 Only physical pages from `ZONE_NORMAL` will be allocated.

G.1.2 Function: `__vmalloc()` (*mm/vmalloc.c*)

This function has three tasks. It page aligns the size request, asks `get_vm_area()` to find an area for the request and uses `vmalloc_area_pages()` to allocate the PTEs for the pages.

```

261 void * __vmalloc (unsigned long size, int gfp_mask, pgprot_t prot)
262 {
263     void * addr;
264     struct vm_struct *area;

```



```

198     struct vm_struct **p, *tmp, *area;
199
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 The parameters are the size of the requested region, which should be a multiple of the page size and the area flags, either VM_ALLOC or VM_IOREMAP.

200-202 Allocates space for the vm_struct description struct.

204 Pads the request so a page gap is between areas. This is to guard against overwrites.

G.1.5 Function: `__vmalloc_area_pages()` (*mm/vmalloc.c*)

This is the beginning of a standard pagetable walk function. This top-level function will step through all PGDs within an address range. For each PGD, it will call `pmd_alloc()` to allocate a PMD directory and call `alloc_area_pmd()` for the directory.

```

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
180         address = (address + PGDIR_SIZE) & PGDIR_MASK;
181         dir++;
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 The parameters are the following:

`address` is the starting address to allocate PMDs for.

`size` is the size of the region.

`gfp_mask` is the GFP flags for `alloc_pages()` (See Section F.1.1).


```

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 The parameters are the following:

`pmd` is the PMD that needs the allocations.

`address` is the starting address to start from.

`size` is the size of the region within the PMD to allocate for.

`gfp_mask` is the GFP_ flags for `alloc_pages()` (See Section F.1.1).

`prot` is the protection to give the PTE entry.

`pages` is an optional array of pages to use instead of allocating each page individually.

138 Aligns the starting address to the PGD.

139-141 Calculates the end to be the end of the allocation or the end of the PGD, whichever occurs first.

142-151 For every PMD within the given address range, this allocates a PTE directory and calls `alloc_area_pte()` (See Section G.1.7).

143 Allocates the PTE directory.

146-147 Calls `alloc_area_pte()`, which will allocate the physical pages if an array of pages is not already supplied with `pages`.

149 `address` becomes the base address of the next PMD entry.

150 Moves `pmd` to the next PMD entry.

152 Returns success.

G.1.7 Function: `alloc_area_pte()` (*mm/vmalloc.c*)

This is the last stage of the pagetable walk. For every PTE in the given PTE directory and address range, a page will be allocated and associated with the 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;
```


- 123-124 Returns failure if physical pages are not available.
- 125 Sets the page with the desired protection bits (prot) into the PTE.
- 126 address becomes the address of the next PTE.
- 127 Moves to the next PTE.
- 129 Returns success.

G.1.8 Function: `vmap()` (*mm/vmalloc.c*)

This function allows a caller-supplied array of pages to be inserted into the vmalloc address space. This is unused in 2.4.22, and I suspect it is an accidental backport from 2.6.x where it is used by the sound subsystem core.

```

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 The parameters are the following:

- `pages` is the caller-supplied array of pages to insert.
- `count` is the number of pages in the array.
- `flags` is the flags to use for the `vm_struct`.
- `prot` is the protection bits to set the PTE with.

286 Calculates the size in bytes of the region to create based on the size of the array.

G.2 Freeing a Noncontiguous Area

Contents

G.2 Freeing a Noncontiguous Area	452
G.2.1 Function: <code>vfree()</code>	452
G.2.2 Function: <code>vmfree_area_pages()</code>	453
G.2.3 Function: <code>free_area_pmd()</code>	454
G.2.4 Function: <code>free_area_pte()</code>	455

G.2.1 Function: `vfree()` (*mm/vmalloc.c*)

The call graph for this function is shown in Figure 7.4. This is the top-level function responsible for freeing a noncontiguous area of memory. It performs basic sanity checks before finding the `vm_struct` for the requested `addr`. Once found, it calls `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         return;
247     }
248     write_lock(&vmlist_lock);
249     for (p = &vmlist ; (tmp = *p) ; p = &tmp->next) {
250         if (tmp->addr == addr) {
251             *p = tmp->next;
252             vmfree_area_pages(VMALLOC_VMADDR(tmp->addr),
253                             tmp->size);
254             write_unlock(&vmlist_lock);
255             kfree(tmp);
256             return;
257         }
258     }
259     write_unlock(&vmlist_lock);
260     printk(KERN_ERR
261         "Trying to vfree() nonexistent vm area (%p)\n", addr);
262 }
```

237 The parameter is the address returned by `get_vm_area()` (See Section G.1.3) to either `vmalloc()` or `ioremap()`.

241-243 Ignores NULL addresses.

89 address becomes the starting address of the next PGD.

90 Moves to the next PGD.

92 Flushes the TLB because the pagetables have now changed.

G.2.3 Function: `free_area_pmd()` (*mm/vmalloc.c*)

This is the second stage of the pagetable walk. For every PMD in this directory, it calls `free_area_pte()` to free up the pages and PTEs.

```

56 static inline void free_area_pmd(pgd_t * dir,
                                   unsigned long address,
                                   unsigned long size)
57 {
58     pmd_t * pmd;
59     unsigned long end;
60
61     if (pgd_none(*dir))
62         return;
63     if (pgd_bad(*dir)) {
64         pgd_ERROR(*dir);
65         pgd_clear(dir);
66         return;
67     }
68     pmd = pmd_offset(dir, address);
69     address &= ~PGDIR_MASK;
70     end = address + size;
71     if (end > PGDIR_SIZE)
72         end = PGDIR_SIZE;
73     do {
74         free_area_pte(pmd, address, end - address);
75         address = (address + PMD_SIZE) & PMD_MASK;
76         pmd++;
77     } while (address < end);
78 }

```

56 The parameters are the PGD being stepped through, the starting address and the length of the region.

61-62 If there is no PGD, this returns. This can occur after `vfree()` (See Section G.2.1) is called during a failed allocation.

63-67 A PGD can be bad if the entry is not present, it is marked read-only or it is marked accessed or dirty.

68 Gets the ~~68~~ PMD for the address range.

69 Makes the address PGD aligned.


```

    "whee.. Swapped out page in kernel page table\n");
53     } while (address < end);
54 }

```

22 The parameters are the PMD that PTEs are being freed from, the starting address and the size of the region to free.

27-28 The PMD could be absent if this region is from a failed `vmalloc()`.

29-33 A PMD can be bad if it is not in main memory, it is read only or it is marked dirty or accessed.

34 `pte` is the ~~%t~~ PTE in the address range.

35 Aligns the address to the PMD.

36-38 The end is either the end of the requested region or the end of the PMD, whichever occurs ~~%t~~.

38-53 Steps through all PTEs, performs checks and frees the PTE with its associated page.

41 `ptep_get_and_clear()` will remove a PTE from a pagetable and return it to the caller.

42 `address` will be the base address of the next PTE.

43 Moves to the next PTE.

44 If there was no PTE, this simply continues.

46-51 If the page is present, this performs basic checks and then frees it.

47 `pte_page()` uses the global `mem_map` to ~~%d~~ the `struct page` for the PTE.

48-49 Makes sure the page is a valid page and that it is not reserved before calling `__free_page()` to free the physical page.

50 Continues to the next PTE.

52 If this line is reached, a PTE within the kernel address space was somehow swapped out. Kernel memory is not swappable, so this is a critical error.

H.3.2.6	Function: <code>kmem_cache_alloc_one_tail()</code>	492
H.3.2.7	Function: <code>kmem_cache_alloc_batch()</code>	494
H.3.3	Object Freeing	495
H.3.3.1	Function: <code>kmem_cache_free()</code>	495
H.3.3.2	Function: <code>__kmem_cache_free (UP Case)()</code>	496
H.3.3.3	Function: <code>__kmem_cache_free (SMP Case)()</code>	496
H.3.3.4	Function: <code>kmem_cache_free_one()</code>	497
H.3.3.5	Function: <code>free_block()</code>	499
H.3.3.6	Function: <code>__free_block()</code>	500
H.4	Sizes Cache	501
H.4.1	Initializing the Sizes Cache	501
H.4.1.1	Function: <code>kmem_cache_sizes_init()</code>	501
H.4.2	<code>kmalloc()</code>	502
H.4.2.1	Function: <code>kmalloc()</code>	502
H.4.3	<code>kfree()</code>	503
H.4.3.1	Function: <code>kfree()</code>	503
H.5	Per-CPU Object Cache	504
H.5.1	Enabling Per-CPU Caches	504
H.5.1.1	Function: <code>enable_all_cpucaches()</code>	504
H.5.1.2	Function: <code>enable_cpucache()</code>	505
H.5.1.3	Function: <code>kmem_tune_cpucache()</code>	506
H.5.2	Updating Per-CPU Information	508
H.5.2.1	Function: <code>smp_call_function_all_cpus()</code>	508
H.5.2.2	Function: <code>do_ccupdate_local()</code>	509
H.5.3	Draining a Per-CPU Cache	509
H.5.3.1	Function: <code>drain_cpu_caches()</code>	509
H.6	Slab Allocator Initialization	511
H.6.1.1	Function: <code>kmem_cache_init()</code>	511
H.7	Interfacing with the Buddy Allocator	512
H.7.1.1	Function: <code>kmem_getpages()</code>	512
H.7.1.2	Function: <code>kmem_freepages()</code>	512

```

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

```

This block performs basic sanity checks for bad usage.

622 The parameters of the function are the following:

- `%name` The human readable name of the cache
- `%size` The size of an object
- `%o set` Used to specify a specific alignment for objects in the cache, but usually left as 0
- `%flags` Static cache `%s`
- `%ctor` A constructor function to call for each object during slab creation
- `%dtor` The corresponding destructor function. The destructor function is expected to leave an object in an initialized state.

633-640 These are all serious usage bugs that prevent the cache from even attempting to create.

634 This is used if the human-readable name is greater than the maximum size for a cache name (CACHE_NAMELEN).

635 An interrupt handler cannot create a cache because access to interrupt-safe spinlocks and semaphores are needed.

636 The object size must be at least a word in size. The slab allocator is not suitable for objects with size measured in individual bytes.

637 The largest possible slab that can be created is $2^{\text{MAX\_OBJ\_ORDER}}$ number of pages, which provides 32 pages.

638 A destructor cannot be used if a constructor is available.

639 The `o set` cannot be before the slab or beyond the boundary of the `%s` page.

640 Calls `BUG()` to exit.

Allocates a `kmem_cache_t` from the `cache_cache` slab cache.

673 Allocates a cache descriptor object from the `cache_cache` with `kmem_cache_alloc()` (See Section H.3.2.1).

674-675 If out of memory, goto `oops`, which handles the OOM situation.

676 Zero-~~the~~ the object to prevent surprises with uninitialized data.

```

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) {
694         flags &= ~SLAB_HWCACHE_ALIGN;
695         size += 2*BYTES_PER_WORD;
696     }
697 #endif
698     align = BYTES_PER_WORD;
699     if (flags & SLAB_HWCACHE_ALIGN)
700         align = L1_CACHE_BYTES;
701
703     if (size >= (PAGE_SIZE>>3))
708         flags |= CFLGS_OFF_SLAB;
709
710     if (flags & SLAB_HWCACHE_ALIGN) {
714         while (size < align/2)
715             align /= 2;
716         size = (size+align-1)&(~(align-1));
717     }

```

Aligns the object size to some word-sized boundary.

682 If the size is not aligned to the size of a word, then...

683-684 Increases the object by the size of a word and then masks out the lower bits. This will effectively round the object size up to the next word boundary.

685 Prints out an informational message for debugging purposes.

688-697 If debugging is enabled, the alignments have to change slightly.

694 Do not bother trying to align things to the hardware cache if the slab will be red-zoned. The red-zoning of the object is going to be set by moving the object one word away from the cache boundary.


```

751 next:
752     cachep->gfporder++;
753 } while (1);
754
755 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 }

```

Calculates how many objects will fit on a slab and adjusts the slab size as necessary.

727-728 `kmem_cache_estimate()` (See Section H.1.2.1) calculates the number of objects that can fit on a slab at the current gfp order and what the amount of leftover bytes will be.

729-730 The `break_flag` is set if the number of objects fitting on the slab exceeds the number that can be kept when only slab slab descriptors are used.

731-732 The order number of pages used must not exceed `MAX_GFP_ORDER` (5).

733-734 If even one object did not fit, goto `next`, which will increase the gfporder used for the cache.

735 If the slab descriptor is kept in `cache`, but the number of objects exceeds the number that can be tracked with `bufctl` in `slab`, then ...

737 Reduces the order number of pages used.

738 Sets the `break_flag` so that the loop will exit.

739 Calculates the new wastage figures.

746-747 The `slab_break_gfp_order` is the order to not exceed unless 0 objects fit on the slab. This check ensures the order is not exceeded.

749-759 A rough check for internal fragmentation. If the wastage as a fraction of the total size of the cache is less than one-eighth, it is acceptable.

752 If the fragmentation is too high, this increases the gfp order and recalculates the number of objects that can be stored and the wastage.

755 If, after adjustments, objects still do not fit in the cache, it cannot be created.

757-758 Frees the cache descriptor and sets the pointer to NULL.

758 Goto `opps`, which simply returns the NULL pointer.


```

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 =
796             kmem_find_general_cachep(slab_size, 0);
797     cachep->ctor = ctor;
798     cachep->dtor = dtor;
799     strcpy(cachep->name, name);
800
801 #ifdef CONFIG_SMP
802     if (g_cpubcache_up)
803         enable_cpubcache(cachep);
804 #endif

```

This block initializes remaining fields in the cache descriptor.

781-782 For caches with slabs of only one page, the `CFLGS_OPTIMIZE` flag is set. In reality, it makes no difference because the flag is unused.

784 Sets the cache static flags.

785 Zeroes out the `gfpflags`. This is a defunct operation, as `memset()` is used to clear these flags after the cache descriptor is allocated.

786-787 If the slab is for DMA use, this sets the `GFP_DMA` flag so that the buddy allocator will use `ZONE_DMA`.

788 Initializes the spinlock for accessing the cache.

789 Copies in the object size, which now takes hardware cache alignment if necessary.

790-792 Initializes the slab lists.

794-795 If the descriptor is kept on-slab, this allocates a slab manager and places it for use in `slabp_cache` (See Section H.2.1.2).

796-797 Sets the pointers to the constructor and destructor functions.

799 Copies in the human-readable name.

802-803 If per-CPU caches are enabled, this creates a set for this cache (See Section 8.5).

```

806     down(&cache_chain_sem);
807     {
808         struct list_head *p;
809

```



```

393     size_t extra = 0;
394     size_t base = 0;
395
396     if (!(flags & CFLGS_OFF_SLAB)) {
397         base = sizeof(slab_t);
398         extra = sizeof(kmem_bufctl_t);
399     }
400     i = 0;
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 The parameters of the function are as follows:

- ~~%gfporder~~ The 2^{gfporder} number of pages to allocate for each slab
- ~~%size~~ The size of each object
- ~~%flags~~ The cache ~~%s~~
- ~~%left_over~~ The number of bytes left over in the slab, which is returned to caller
- ~~%num~~ The number of objects that will ~~%in~~ a slab, which is returned to caller

392 wastage is decremented through the function. It starts with the maximum possible amount of wastage.

393 extra is the number of bytes needed to store kmem_bufctl_t.

394 base is where usable memory in the slab starts.

396 If the slab descriptor is kept on cache, the base begins at the end of the slab_t struct and the number of bytes needed to store the bufctl is the size of kmem_bufctl_t.

400 i becomes the number of objects that the slab can hold.

401-402 Counts up the number of objects that the cache can store. i*size is the size of the object itself. L1_CACHE_ALIGN(base+i*extra) is slightly trickier.

H.1.4 Cache Destroying

When a module is unloaded, it is responsible for destroying any cache it has created. As during module loading, it is ensured two caches do not have the same name. Core kernel code often does not destroy its caches because their existence persists for the life of the system. The steps taken to destroy a cache are the following:

1. Delete the cache from the cache chain.
2. Shrink the cache to delete all slabs (See Section 8.1.8).
3. Free any per-CPU caches (`kfree()`).
4. Delete the cache descriptor from the `cache_cache` (See Section 8.3.3).

H.1.4.1 Function: `kmem_cache_destroy()` (*mm/slab.c*)

The call graph for this function is shown in Figure 8.7.

```

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     }
1020     #ifdef CONFIG_SMP
1021     {
1022         int i;
1023         for (i = 0; i < NR_CPUS; i++)
1024             kfree(cachep->cpudata[i]);
1025     }
1026 #endif

```



```

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 The only parameter is the GFP flag. The only check made is against the `__GFP_WAIT` flag. As the only caller, `kswapd`, can sleep. This parameter is virtually worthless.

1748-1749 Can the caller sleep? If yes, then this acquires the semaphore.

1751-1752 If not, this tries and acquires the semaphore. If it is not available, this returns.

1754 `REAP_SCANLEN` (10) is the number of caches to examine.

1758 Sets `searchp` to be the last cache that was examined at the last reap.

```

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);
1779             cc->avail = 0;
1780         }
1781     }
1782 #endif

```



```

1801 Calculates the number of pages that all the slabs hold.
1802-1803 If the objects have constructors, this reduces the page count by one-
      %h to make it less likely to be selected for reaping.
1804-1805 If the slabs consist of more than one page, this reduces the page count
      by one-%h. This is because high-order pages are hard to acquire.
1806 If this is the best candidate found for reaping so far, this checks if it is perfect
      for reaping.
1807-1809 Records the new maximums.
1808 best_len is recorded so that it is easy to know how many slabs are half of
      the slabs in the free list.
1810 If this cache is perfect for reaping, then...
1811 Updates clock_searchp.
1812 Goto perfect where half the slabs will be freed.
1816 This label is reached if it was found that the cache was growing after acquiring
      the lock.
1817 Releases the cache descriptor lock.
1818 Moves to the next entry in the cache chain.
1820 Scans while REAP_SCANLEN has not been reached and while we have not cycled
      around the whole cache chain.

1822     clock_searchp = searchp;
1823
1824     if (!best_cachep)
1826         goto out;
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);

```


1851 Frees the cache descriptor and enables interrupts.

1852 Destroys the slab (See Section 8.2.8).

1851 Reacquires the cache descriptor spinlock and disables interrupts.

1855 Frees the cache descriptor and enables interrupts.

1856 ret is the number of pages that were freed.

1858-1859 Frees the cache semaphore and returns the number of pages freed.


```

1130     BUG();
1131
1132     ctor_flags = SLAB_CTOR_CONSTRUCTOR;
1133     local_flags = (flags & SLAB_LEVEL_MASK);
1134     if (local_flags == SLAB_ATOMIC)
1135         ctor_flags |= SLAB_CTOR_ATOMIC;

```

This performs basic sanity checks to guard against bad usage. The checks are made here rather than `kmem_cache_alloc()` to protect the speed-critical path. There is no point in checking the `%s` every time an object needs to be allocated.

1118-1119 Makes sure only allowable `%s` are used for allocation.

1120-1121 Do not grow the cache if this is set. In reality, it is never set.

1129-1130 If this is called within interrupt context, make sure the `ATOMIC %s` is set, so we do not sleep when `kmem_getpages()` (See Section H.7.1.1) is called.

1132 This `%s` tells the constructor it is to init the object.

1133 The `local_flags` are just those relevant to the page allocator.

1134-1139 If the `SLAB_ATOMIC %s` is set, the constructor needs to know about it in case it wants to make new allocations.

```

1142     spin_lock_irqsave(&cachep->spinlock, save_flags);
1143
1144     offset = cachep->colour_next;
1145     cachep->colour_next++;
1146     if (cachep->colour_next >= cachep->colour)
1147         cachep->colour_next = 0;
1148     offset *= cachep->colour_off;
1149     cachep->dflags |= DFLGS_GROWN;
1150
1151     cachep->growing++;
1152     spin_unlock_irqrestore(&cachep->spinlock, save_flags);

```

Calculates color o set for objects in this slab.

1142 Acquires an interrupt-safe lock for accessing the cache descriptor.

1145 Gets the o set for objects in this slab.

1146 Moves to the next color o set.

1147-1148 If `colour` has been reached, no more o sets are available, so this resets `colour_next` to 0.

1149 `colour_off` is the size of each o set, so `offset * colour_off` will give how many bytes to o set the objects to.


```

1179 Moves to the next page for this slab to be linked.
1182     kmem_cache_init_objs(cachep, slabp, ctor_flags);
1182     Initializes all objects (See Section 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;

```

Adds the slab to the cache.

1184 Acquires the cache descriptor spinlock in an interrupt-safe fashion.

1185 Decreases the growing count.

1188 Adds the slab to the end of the `slabs_free` list.

1189 If `STATS` is set, this increases the `cachep->grown` and `STATS_INC_GROWN()`.

1190 Sets failures to 0. This `fail` is never used elsewhere.

1192 Unlocks the spinlock in an interrupt-safe fashion.

1193 Returns success.

```

1194 opps1:
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 }

```

This block is for error handling.

1194-1195 `opps1` is reached if the pages for the slab were allocated. They must be freed.

1197 Acquires the spinlock for accessing the cache descriptor.

1198 Reduces the growing count.

1199 Releases the spinlock.

1300 Returns failure.

H.3 Objects

Contents

H.3 Objects	486
H.3.1 Initializing Objects in a Slab	486
H.3.1.1 Function: <code>kmem_cache_init_objs()</code>	486
H.3.2 Object Allocation	488
H.3.2.1 Function: <code>kmem_cache_alloc()</code>	488
H.3.2.2 Function: <code>__kmem_cache_alloc (UP Case)()</code>	488
H.3.2.3 Function: <code>__kmem_cache_alloc (SMP Case)()</code>	489
H.3.2.4 Function: <code>kmem_cache_alloc_head()</code>	491
H.3.2.5 Function: <code>kmem_cache_alloc_one()</code>	492
H.3.2.6 Function: <code>kmem_cache_alloc_one_tail()</code>	492
H.3.2.7 Function: <code>kmem_cache_alloc_batch()</code>	494
H.3.3 Object Freeing	495
H.3.3.1 Function: <code>kmem_cache_free()</code>	495
H.3.3.2 Function: <code>__kmem_cache_free (UP Case)()</code>	496
H.3.3.3 Function: <code>__kmem_cache_free (SMP Case)()</code>	496
H.3.3.4 Function: <code>kmem_cache_free_one()</code>	497
H.3.3.5 Function: <code>free_block()</code>	499
H.3.3.6 Function: <code>__free_block()</code>	500

This section will cover how objects are managed. At this point, most of the real hard work has been completed by either the cache or slab managers.

H.3.1 Initializing Objects in a Slab

H.3.1.1 Function: `kmem_cache_init_objs()` (*mm/slab.c*)

The vast part of this function is involved with debugging, so I start with the function without the debugging and explain that in detail before handling the debugging part. The two sections that are debugging are marked in the code excerpt that follows as Part 1 and Part 2.

```

1058 static inline void kmem_cache_init_objs (kmem_cache_t * cachep,
1059     slab_t * slabb, unsigned long ctor_flags)
1060 {
1061     int i;
1062
1063     for (i = 0; i < cachep->num; i++) {
1064         void* objp = slabb->s_mem+cachep->objsize*i;
1065-1072         /* Debugging Part 1 */
1066
1067
1068
1069         if (cachep->ctor)
1070             cachep->ctor(objp, cachep, ctor_flags);
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081-1094         /* Debugging Part 2 */
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095         slab_bufctl(slabb)[i] = i+1;

```



```

1081 #if DEBUG
1082     if (cachep->flags & SLAB_RED_ZONE)
1083         objp -= BYTES_PER_WORD;
1084     if (cachep->flags & SLAB_POISON)
1085         kmem_poison_obj(cachep, objp);
1086     if (cachep->flags & SLAB_RED_ZONE) {
1087         if (*((unsigned long*)(objp)) != RED_MAGIC1)
1088             BUG();
1089         if (*((unsigned long*)(objp + cachep->objsize -
1090             BYTES_PER_WORD)) != RED_MAGIC1)
1091             BUG();
1092     }
1093 }
1094 #endif

```

This is the debugging block that takes place after the constructor, if it exists, has been called.

1082-1083 The objp pointer was increased by the size of the red marker in the previous debugging block, so it moves it back again.

1084-1086 If there was no constructor, this poisons the object with a known pattern that can be examined later to trap uninitialized writes.

1088 Checks to make sure the red marker at the beginning of the object was preserved to trap writes before the object.

1090-1091 Checks to make sure writes did not take place past the end of the object.

H.3.2 Object Allocation

H.3.2.1 Function: kmem_cache_alloc() (*mm/slab.c*)

The call graph for this function is shown in Figure 8.14. This trivial function simply calls `__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 Function: __kmem_cache_alloc (UP Case)() (*mm/slab.c*)

This will take the parts of the function specific to the UP case. The SMP case will be dealt with in the next section.

```

1338 static inline void * __kmem_cache_alloc (kmem_cache_t *cachep,
                                           int flags)
1339 {
1340     unsigned long save_flags;

```



```

1342
1343     kmem_cache_alloc_head(cachep, flags);
1344 try_again:
1345     local_irq_save(save_flags);
1346     {
1347         cpucache_t *cc = cc_data(cachep);
1348
1349         if (cc) {
1350             if (cc->avail) {
1351                 STATS_INC_ALLOCHIT(cachep);
1352                 objp = cc_entry(cc)[--cc->avail];
1353             } else {
1354                 STATS_INC_ALLOCMISS(cachep);
1355                 objp =
1356                     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);
1367     return objp;
1368 alloc_new_slab:
1369     spin_unlock(&cachep->spinlock);
1370 alloc_new_slab_nolock:
1371     local_irq_restore(save_flags);
1372     if (kmem_cache_grow(cachep, flags))
1373         goto try_again;
1374     return NULL;
1375 }

```

1338-1347 The same as the UP case.

1349 Obtains the per-CPU data for this CPU.

1350-1360 If a per-CPU cache is available, then

1351 If an object is available, then

1352 Updates statistics for this cache if enabled.

1353 Gets an object and updates the available count.

1354 If not, an object is not available, so

1256-1259 If the slab is full, this removes it from the `slabs_partial` list and places it on `slabs_full`.

1260-1274 Debugging code.

1275 Without debugging, the object is returned to the caller.

1261-1263 If the object was poisoned with a known pattern, this checks it to guard against uninitialized access.

1266-1267 If red-zoning was enabled, this checks the marker at the beginning of the object and confirms it is safe. It changes the red marker to check for writes before the object later.

1269-1271 Checks the marker at the end of the object and changes it to check for writes after the object later.

1272 Updates the object pointer to point to after the red marker.

1275 Returns the object.

H.3.2.7 Function: `kmem_cache_alloc_batch()` (*mm/slab.c*)

This function allocates a batch of objects to a CPU cache of objects. It is only used in the SMP case. In many ways, it is very similar to `kmem_cache_alloc_one()` (See Section H.3.2.5).

```

1305 void* kmem_cache_alloc_batch(kmem_cache_t* cachep,
                                cpucache_t* cc, int flags)
1306 {
1307     int batchcount = cachep->batchcount;
1308
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     }

```



```

1586     __kmem_cache_free(cachep, objp);
1587     local_irq_restore(flags);
1588 }

```

1576 The parameter is the cache that the object is being freed from and the object itself.

1579-1583 If debugging is enabled, the page will ~~not~~ be checked with `CHECK_PAGE()` to make sure it is a slab page. Second, the page list will be examined to make sure it belongs to this cache (See Figure 8.8).

1585 Interrupts are disabled to protect the path.

1586 `__kmem_cache_free()` (See Section H.3.3.2) will free the object to the per-CPU cache for the SMP case and to the global pool in the normal case.

1587 Re-enables interrupts.

H.3.3.2 Function: `__kmem_cache_free (UP Case)()` (*mm/slab.c*)

This covers what the function looks like in the UP case. Clearly, it simply releases the object to the slab.

```

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 Function: `__kmem_cache_free (SMP Case)()` (*mm/slab.c*)

This case is slightly more interesting. In this case, the object is released to the per-CPU cache if it is available.

```

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;

```



```

1434
1435     if (cachep->flags & SLAB_RED_ZONE) {
1436         objp -= BYTES_PER_WORD;
1437         if (xchg((unsigned long *)objp, RED_MAGIC1) !=
1438             RED_MAGIC2)
1439             BUG();
1440         if (xchg((unsigned long *) (objp+cachep->objsize -
1441             BYTES_PER_WORD), RED_MAGIC1) !=
1442             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
1458     {
1459         int inuse = slabp->inuse;
1460         if (unlikely(!--slabp->inuse)) {
1461             /* Was partial or full, now empty. */
1462             list_del(&slabp->list);
1463             list_add(&slabp->list, &cachep->slabs_free);
1464         } else if (unlikely(inuse == cachep->num)) {
1465             /* Was full. */
1466             list_del(&slabp->list);
1467             list_add(&slabp->list, &cachep->slabs_part
1468
1469         }
1470     }
1471 }

```

1418 Makes sure the page is a slab page.

1425 Gets the slab descriptor for the page.

1427-1449 Debugging material. It is discussed at end of the section.

1451 Calculates the index for the object being freed.

1454 Because this object is now free, it updates the bufctl to reflect that.

H.3.3.6 Function: `__free_block()` (*mm/slab.c*)

This function is responsible for freeing each of the objects in the per-CPU array `objpp`.

```
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 The parameters are the `cachep` the objects belong to, the list of objects (`objpp`) and the number of objects to free (`len`).

1477 Loops `len` number of times.

1478 Frees an object from the array.

- 438 Gets a pointer to the `cache_sizes` array.
- 439 The human-readable name of the cache. It should be sized `CACHE_NAMELEN`, which is defined to be 20 bytes long.
- 444-445 `slab_break_gfp_order` determines how many pages a slab may use unless 0 objects fit into the slab. It is statically initialized to `BREAK_GFP_ORDER_LO` (1). This check sees if more than 32MiB of memory is available, and, if it is, it allows `BREAK_GFP_ORDER_HI` number of pages to be used because internal fragmentation is more acceptable when more memory is available.
- 446-470 Creates two caches for each size of memory allocation needed.
- 452 Stores the human-readable cache name in `name`.
- 453-454 Creates the cache, aligned to the L1 cache.
- 460-463 Calculates the `o_slab bufctl` limit, which determines the number of objects that can be stored in a cache when the slab descriptor is kept `o_slab` cache.
- 464 The human-readable name for the cache for DMA use.
- 465-466 Creates the cache, aligned to the L1 cache and suitable for the DMA user.
- 467 If the cache failed to allocate, it is a bug. If memory is unavailable this early, the machine will not boot.
- 469 Moves to the next element in the `cache_sizes` array.
- 470 The array is terminated with a 0 as the last element.

H.4.2 `kmalloc()`

H.4.2.1 Function: `kmalloc()` (*mm/slab.c*)

The call graph for this function is shown in Figure 8.16.

```

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);
1565     }
1566     return NULL;
1567 }
```



```

1705     else
1706         limit = 252;
1707
1708     err = kmem_tune_cpucache(cachep, limit, limit/2);
1709     if (err)
1710         printk(KERN_ERR
1711             "enable_cpucache failed for %s, error %d.\n",
1712             cachep->name, -err);
1712 }

```

1699-1700 If an object is larger than a page, return to avoid creating a per-CPU cache for this object type because per-CPU caches are too expensive.

1701-1702 If an object is larger than 1KiB, this keeps the cpucache lower than 3MiB in size. The limit is set to 124 objects to take the size of the cpucache descriptors into account.

1703-1704 For smaller objects, this just makes sure the cache does not go above 3MiB in size.

1708 Allocates the memory for the cpucache.

1710-1711 Prints out an error message if the allocation failed.

H.5.1.3 Function: `kmem_tune_cpucache()` (*mm/slab.c*)

This function is responsible for allocating memory for the cpucaches. For each CPU on the system, `kmallocc` gives a block of memory large enough for one cpucache and ~~the~~ a `ccupdate_struct_t` struct. The function `smp_call_function_all_cpus()` then calls `do_ccupdate_local()`, which swaps the new information with the old information in the cache descriptor.

```

1639 static int kmem_tune_cpucache (kmem_cache_t* cachep,
1640                                int limit, int batchcount)
1641 {
1642     ccupdate_struct_t new;
1643     int i;
1644
1645     /*
1646      * These are admin-provided, so we are more graceful.
1647      */
1648     if (limit < 0)
1649         return -EINVAL;
1650     if (batchcount < 0)
1651         return -EINVAL;
1652     if (batchcount > limit)
1653         return -EINVAL;
1654     if (limit != 0 && !batchcount)

```


- 1647 The number of objects in the cache cannot be negative.
- 1649 A negative number of objects cannot be allocated.
- 1651 A batch of objects greater than the limit cannot be allocated.
- 1653 A batchcount must be provided if the limit is positive.
- 1656 Zero-~~fills~~ the update struct.
- 1657 If a limit is provided, this allocates memory for the cpucache.
- 1658-1668 For every CPU, this allocates a cpucache.
- 1661 The amount of memory needed is limit number of pointers and the size of the cpucache descriptor.
- 1663 If out of memory, this cleans up and exits.
- 1665-1666 Fills in the ~~ids~~ for the cpucache descriptor.
- 1667 Fills in the information for ccupdate_update_t struct.
- 1670 Tells the ccupdate_update_t struct what cache is being updated.
- 1671-1673 Acquires an interrupt-safe lock to the cache descriptor and sets its batchcount.
- 1675 Gets each CPU to update its cpucache information for itself. This swaps the old cpucaches in the cache descriptor with the new ones in new using do_ccupdate_local() (See Section H.5.2.2).
- 1677-1685 After smp_call_function_all_cpus() (See Section H.5.2.1), the old cpucaches are in new. This block of code cycles through them all, frees any objects in them and deletes the old cpucache.
- 1686 Returns success.
- 1688 In the event there is no memory, this deletes all cpucaches that have been allocated up until this point and returns failure.

H.5.2 Updating Per-CPU Information

H.5.2.1 Function: smp_call_function_all_cpus() (*mm/slab.c*)

This calls the function func() for all CPUs. In the context of the slab allocator, the function is do_ccupdate_local(), and the argument is 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);

```



```

889
890     memset(&new.new, 0, sizeof(new.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 Blanks the update structure because it is going to be clearing all data.

892 Sets `new.cachep` to `cachep` so that `smp_call_function_all_cpus()` knows what cache it is affecting.

894 Acquires the cache descriptor semaphore.

895 `do_ccupdate_local()` (See Section H.5.2.2) swaps the `cpucache_t` information in the cache descriptor with the ones in `new` so they can be altered here.

897-905 For each CPU in the system,...

898 Gets the `cpucache` descriptor for this CPU.

899 If the structure does not exist for some reason or no objects are available in it, this moves to the next CPU.

901 Disables interrupts on this processor. It is possible an allocation from an interrupt handler elsewhere would try to access the per-CPU cache.

902 Frees the block of objects with `free_block()` (See Section H.3.3.5).

903 Re-enables interrupts.

904 Shows that no objects are available.

906 The information for each CPU has been updated, so this calls `do_ccupdate_local()` (See Section H.5.2.2) for each CPU to put the information back into the cache descriptor.

907 Releases the semaphore for the cache chain.

H.7 Interfacing with the Buddy Allocator

Contents

H.7 Interfacing with the Buddy Allocator	512
H.7.1.1 Function: <code>kmem_getpages()</code>	512
H.7.1.2 Function: <code>kmem_freepages()</code>	512

H.7.1.1 Function: `kmem_getpages()` (*mm/slab.c*)

This allocates pages for the slab allocator.

```

486 static inline void * kmem_getpages (kmem_cache_t *cachep,
                                     unsigned long flags)
487 {
488     void *addr;
495     flags |= cachep->gfpflags;
496     addr = (void*) __get_free_pages(flags, cachep->gfporder);
503     return addr;
504 }
```

495 Whatever `flags` were requested for the allocation, this appends the cache `flags` to it. The only `flag` it may append is `ZONE_DMA` if the cache requires DMA memory.

496 Allocates from the buddy allocator with `__get_free_pages()` (See Section F.2.3).

503 Returns the pages or NULL if it failed.

H.7.1.2 Function: `kmem_freepages()` (*mm/slab.c*)

This frees pages for the slab allocator. Before it calls the buddy allocator API, it will remove the `PG_slab` bit from the page `flags`.

```

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
517     while (i--) {
518         PageClearSlab(page);
519         page++;
520     }
521     free_pages((unsigned long)addr, cachep->gfporder);
522 }
```

509 Retrieves the order used for the original allocation.

510 Gets the struct page for the address.

517-520 Clears the `PG_slab` bit on each page.

521 Frees the pages to the buddy allocator with `free_pages()` (See Section F.4.1).

I.1 Mapping High Memory Pages

Contents

I.1 Mapping High Memory Pages	514
I.1.1 Function: <code>kmap()</code>	514
I.1.2 Function: <code>kmap_nonblock()</code>	514
I.1.3 Function: <code>__kmap()</code>	514
I.1.4 Function: <code>kmap_high()</code>	515
I.1.5 Function: <code>map_new_virtual()</code>	515
I.1.6 Function: <code>flush_all_zero_pkmaps()</code>	518

I.1.1 Function: `kmap()` (*include/asm-i386/highmem.c*)

This API is used by callers willing to block.

```
62 #define kmap(page) __kmap(page, 0)
```

62 The core function `__kmap()` is called with the second parameter indicating that the caller is willing to block.

I.1.2 Function: `kmap_nonblock()` (*include/asm-i386/highmem.c*)

```
63 #define kmap_nonblock(page) __kmap(page, 1)
```

63 The core function `__kmap()` is called with the second parameter indicating that the caller is not willing to block.

I.1.3 Function: `__kmap()` (*include/asm-i386/highmem.h*)

The call graph for this function is shown in Figure 9.1.

```
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 This function may not be used from interrupt because it may sleep. Instead of `BUG()`, `out_of_line_bug()` calls `do_exit()` and returns an error code. `BUG()` is not used because `BUG()` kills the process with extreme prejudice, which would result in the fabled `%die`, killing interrupt handler! kernel panic.

69-70 If the page is already in low memory, this returns a direct mapping.

71 Calls `kmap_high()` (See Section I.1.4) for the beginning of the architecture-independent work.


```

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 Starts scanning at the last possible slot.

88-122 This loop keeps scanning and waiting until a slot becomes free. This allows the possibility of an infinite loop for some processes if they were unlucky.

89 last_pkmap_nr is the last pkmap that was scanned. To prevent searching over the same pages, this value is recorded so that the list is searched circularly. When it reaches LAST_PKMAP, it wraps around to 0.

90-93 When last_pkmap_nr wraps around, this calls flush_all_zero_pkmaps() (See Section I.1.6), which will set all entries from 1 to 0 in the pkmap_count array before flushing the TLB. The count is set back to LAST_PKMAP to restart scanning.

94-95 If this element is 0, a usable slot has been found for the page.

96-97 Moves to the next index to scan.

99-100 The next block of code is going to sleep while waiting for a slot to be free. If the caller requested that the function not block, it returns now.

```

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 */

```


127 Sets the virtual ~~id~~ for the page.

129 Returns the virtual address.

I.1.6 Function: `flush_all_zero_pkmaps()` (*mm/highmem.c*)

This function cycles through the `pkmap_count` array and sets all entries from 1 to 0 before ~~flushing~~ the TLB.

```

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;
63     }
64     flush_tlb_all();
65 }

```

46 As the global pagetables are about to change, the CPU caches of all processors have to be ~~flushed~~.

48-76 Cycles through the entire `pkmap_count` array.

57-58 If the element is not 1, this moves to the next element.

59 Sets from 1 to 0.

62-63 Makes sure the PTE is not somehow mapped.

72-73 Unmaps the page from the PTE and clears the PTE.

75 Updates the virtual ~~id~~ as the page is unmapped.

77 Flushes the TLB.

94-95 If the page is in low memory, this returns a direct mapping.

97 type gives which slot to use. `KM_TYPE_NR * smp_processor_id()` gives the set of slots reserved for this processor.

98 Gets the virtual address.

100-101 For debugging code. In reality, a PTE will always exist.

103 Sets the PTE into the reserved slot.

104 Flushes the TLB for this slot.

106 Returns the virtual address.


```

191
192     /* do wake-up, if needed, race-free outside of the spin lock */
193     if (need_wakeup)
194         wake_up(&pkmap_map_wait);
195 }

```

163 Acquires `kmap_lock`, protecting the virtual `Ma` and the `pkmap_count` array.

164 Gets the virtual page.

165-166 If the virtual `Ma` is not set, it is a double unmapping or unmapping of a nonmapped page, so it uses `BUG()`.

167 Gets the index within the `pkmap_count` array.

173 By default, a wakeup call to processes calling `kmap()` is not needed.

174 Checks the value of the index after decrement.

175-176 Falling to 0 is a bug because the TLB needs to be flushed to make 0 a valid entry.

177-188 If it has dropped to 1 (the entry is now free, but needs a TLB flush), this checks to see if anyone is sleeping on the `pkmap_map_wait` queue. If necessary, the queue will be woken up after the spinlock is freed.

190 Frees `kmap_lock`.

193-194 If waiters are on the queue and a slot has been freed, this wakes them up.

I.5 Bounce Buffers

Contents

I.5 Bounce Buffers	524
I.5.1 Creating Bounce Buffers	524
I.5.1.1 Function: <code>create_bounce()</code>	524
I.5.1.2 Function: <code>alloc_bounce_bh()</code>	526
I.5.1.3 Function: <code>alloc_bounce_page()</code>	527
I.5.2 Copying Using Bounce Buffers	528
I.5.2.1 Function: <code>bounce_end_io_write()</code>	528
I.5.2.2 Function: <code>bounce_end_io_read()</code>	528
I.5.2.3 Function: <code>copy_from_high_bh()</code>	529
I.5.2.4 Function: <code>copy_to_high_bh_irq()</code>	529
I.5.2.5 Function: <code>bounce_end_io()</code>	530

I.5.1 Creating Bounce Buffers

I.5.1.1 Function: `create_bounce()` (*mm/highmem.c*)

The call graph for this function is shown in Figure 9.3. It is a high-level function for the creation of bounce buffers. It is broken into two major parts, the allocation of the necessary resources and the copying of data from the template.

```

405 struct buffer_head * create_bounce(int rw,
                                     struct buffer_head * bh_orig)
406 {
407     struct page *page;
408     struct buffer_head *bh;
409
410     if (!PageHighMem(bh_orig->b_page))
411         return bh_orig;
412
413     bh = alloc_bounce_bh();
420     page = alloc_bounce_page();
421
422     set_bh_page(bh, page, 0);
423

```

405 The parameters of the function are the following:

`rw` is set to 1 if this is a write buffer.

`bh_orig` is the template buffer head to copy from.

410-411 If the template buffer head is already in low memory, this simply returns it.

413 Allocates a buffer head from the slab allocator or from the emergency pool if it fails.

420 Allocates a page from the buddy allocator or the emergency pool if it fails.

446-447 Copies the remaining information from the template `buffer_head`.

452 Returns the new bounce buffer.

I.5.1.2 Function: `alloc_bounce_bh()` (*mm/highmem.c*)

This function ~~not~~ tries to allocate a `buffer_head` from the slab allocator, and, if that fails, an emergency pool will be used.

```
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_bdflush();
```

374 Tries to allocate a new `buffer_head` from the slab allocator. Note how the request is made to *not* use I/O operations that involve high I/O to avoid recursion.

375-376 If the allocation was successful, this returns.

381 If it was not, this wakes up `bdflush` to launder pages.

```
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);
392         list_del(tmp->next);
393         nr_emergency_bhs--;
394     }
395     spin_unlock_irq(&emergency_lock);
396     if (bh)
397         return bh;
398     /* we need to wait I/O completion */
399     run_task_queue(&tq_disk);
400
401     yield();
402     goto repeat_alloc;
403 }
```

The allocation from the slab failed, so this allocates from the emergency pool.


```

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 Gets the end of the emergency buffer head list.

352 Acquires the lock protecting the pools.

353-357 If the pool is not empty, this takes a page from the list and decrements the number of available `nr_emergency_pages`.

358 Releases the lock.

359-360 If the allocation was successful, this returns it.

363 Runs the I/O task queue to try and replenish the emergency pool.

365 Yields the processor.

366 Attempts to allocate from the emergency pools again.

I.5.2 Copying Using Bounce Buffers

I.5.2.1 Function: `bounce_end_io_write()` (*mm/highmem.c*)

This function is called when a bounce buffer used for writing to a device completes I/O. Because the buffer is copied *from* high memory and to the device, there is nothing left to do except reclaim the resources.

```

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 Function: `bounce_end_io_read()` (*mm/highmem.c*)

This is called when data has been read from the device and needs to be copied to high memory. It is called from interrupt, so it has to be more careful.

```

324 static void bounce_end_io_read (struct buffer_head *bh,
                                int uptodate)
325 {
326     struct buffer_head *bh_orig =
        (struct buffer_head *) (bh->b_private);

```



```

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 Saves the flags and disables interrupts.

238 Maps the high-memory page into low memory.

239 Copies the data.

240 Unmaps the page.

241 Restores the interrupt flags.

I.5.2.5 Function: bounce_end_io() (*mm/highmem.c*)

This reclaims the resources used by the bounce buffers. If emergency pools are depleted, the resources are added to it.

```

244 static inline void bounce_end_io (struct buffer_head *bh,
                                   int uptodate)
245 {
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) {

```


I.6 Emergency Pools

Contents

I.6 Emergency Pools	532
I.6.1 Function: <code>init_emergency_pool()</code>	532

Only one function is of relevance to the emergency pools, and that is the `init` function. It is called during system startup, and then the code is deleted because it is never needed again.

I.6.1 Function: `init_emergency_pool()` (*mm/highmem.c*)

This creates a pool for emergency pages and for emergency buffer_heads.

```

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 If no high memory is available, this does not bother.

291 Acquires the lock protecting the pools.

292-300 Allocates `POOL_SIZE` pages from the buddy allocator and adds them to a linked list. It keeps a count of the number of pages in the pool with `nr_emergency_pages`.

```

301     while (nr_emergency_bhs < POOL_SIZE) {
302         struct buffer_head * bh =
303             kmem_cache_alloc(bh_cachep, SLAB_ATOMIC);
304         if (!bh) {
305             printk("couldn't refill highmem emergency bhs");
306             break;
307         }

```

J.2.2.3	Function: del_page_from_active_list()	549
J.2.2.4	Function: del_page_from_inactive_list()	549
J.2.3	Activating Pages	550
J.2.3.1	Function: mark_page_accessed()	550
J.2.3.2	Function: activate_lock()	550
J.2.3.3	Function: activate_page_nolock()	550
J.3	Refilling inactive_list	552
J.3.1	Function: refill_inactive()	552
J.4	Reclaiming Pages From the LRU Lists	554
J.4.1	Function: shrink_cache()	554
J.5	Shrinking All Caches	562
J.5.1	Function: shrink_caches()	562
J.5.2	Function: try_to_free_pages()	563
J.5.3	Function: try_to_free_pages_zone()	564
J.6	Swapping Out Process Pages	566
J.6.1	Function: swap_out()	566
J.6.2	Function: swap_out_mm()	568
J.6.3	Function: swap_out_vma()	569
J.6.4	Function: swap_out_pgdat()	570
J.6.5	Function: swap_out_pmd()	571
J.6.6	Function: try_to_swap_out()	573
J.7	Page Swap Daemon	577
J.7.1	Initializing kswapd	577
J.7.1.1	Function: kswapd_init()	577
J.7.2	kswapd Daemon	577
J.7.2.1	Function: kswapd()	577
J.7.2.2	Function: kswapd_can_sleep()	579
J.7.2.3	Function: kswapd_can_sleep_pgdat()	579
J.7.2.4	Function: kswapd_balance()	580
J.7.2.5	Function: kswapd_balance_pgdat()	580

671 Releases the lock protecting the hash and inode queue.

672 Adds the page to the page cache. `page_hash()` hashes into the page hash table based on the mapping and the offset within the page. If a page is returned, there was a collision, and the colliding pages are chained with the `page→next_hash` and `page→pprev_hash` fields.

J.1.1.2 Function: `add_to_page_cache_unique()` (*mm/page_map.c*)

In many respects, this function is very similar to `add_to_page_cache()`. The principal difference is that this function will check the page cache with the `pagecache_lock` spinlock held before adding the page to the cache. It is for callers that may race with another process for inserting a page in the cache, such as `add_to_swap_cache()` (See Section 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         err = 0;
689     }
690
691     spin_unlock(&pagecache_lock);
692     if (!err)
693         lru_cache_add(page);
694     return err;
695 }
```

682 Acquires the `pagecache_lock` for examining the cache.

683 Checks if the page already exists in the cache with `__find_page_nolock()` (See Section J.1.4.3).

686-689 If the page does not exist in the cache, this adds it with `__add_to_page_cache()` (See Section J.1.1.3).

691 Releases the `pagecache_lock`.

692-693 If the page did not already exist in the page cache, it adds it to the LRU lists with `lru_cache_add()` (See Section J.2.1.1).


```

88
89     mapping->numpages++;
90     list_add(&page->list, head);
91     page->mapping = mapping;
92 }

```

87 When this function is called, the page is clean, so `mapping->clean_pages` is the list of interest.

89 Increments the number of pages that belong to this mapping.

90 Adds the page to the clean list.

91 Sets the `page->mapping` field.

J.1.1.5 Function: `add_page_to_hash_queue()` (*mm/bmap.c*)

This adds `page` to the top of the hash bucket headed by `p`. Bear in mind that `p` is an element of the array `page_hash_table`.

```

71 static void add_page_to_hash_queue(struct page * page,
                                   struct page **p)
72 {
73     struct page *next = *p;
74
75     *p = page;
76     page->next_hash = next;
77     page->pprev_hash = p;
78     if (next)
79         next->pprev_hash = &page->next_hash;
80     if (page->buffers)
81         PAGE_BUG(page);
82     atomic_inc(&page_cache_size);
83 }

```

73 Records the current head of the hash bucket in `next`.

75 Updates the head of the hash bucket to be `page`.

76 Points `page->next_hash` to the old head of the hash bucket.

77 Points `page->pprev_hash` to point to the array element in `page_hash_table`.

78-79 This will point the `pprev_hash` field to the head of the hash bucket, completing the insertion of the `page` into the linked list.

80-81 Checks that the page entered has no associated buffers.

82 Increments `page_cache_size`, which is the size of the page cache.


```

100     list_del(&page->list);
101     page->mapping = NULL;
102     wmb();
103     mapping->nr_pages--;
104 }

```

96 Gets the associated `address_space` for this page.

98-99 Calls the ~~%system-specific%~~ `removepage()` function if one is available.

100 Deletes the page from whatever list it belongs to in the mapping, such as the `clean_pages` list in most cases or the `dirty_pages` in rarer cases.

101 Sets the `page->mapping` to `NULL` because it is no longer backed by any `address_space`.

103 Decrements the number of pages in the mapping.

J.1.2.4 Function: `remove_page_from_hash_queue()` (*mm/~~6~~map.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 Gets the next page after the page being removed.

110 Gets the `pprev` page before the page being removed. When the function completes, `pprev` will be linked to `next`.

112 If this is not the end of the list, this updates `next->pprev_hash` to point to `pprev`.

114 Similarly, this points `pprev` forward to `next`. `page` is now unlinked.

116 Decrements the size of the pagecache.

J.1.3 Acquiring/Releasing Page Cache Pages

J.1.3.1 Function: `page_cache_get()` (*include/linux/pagemap.h*)

```

31 #define page_cache_get(x)      get_page(x)

```

31 A simple call `get_page()`, which uses `atomic_inc()` to increment the page reference count.

J.1.4.3 Function: `__find_page_nolock()` (*mm/~~pg~~map.c*)

This function traverses the hash collision list looking for the page specified by the `address_space` and `offset`.

```

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 Begins by examining the ~~first~~ page in the list.

450-451 If the page is NULL, the right one could not be found, so it returns NULL.

452 If the `address_space` does not match, this moves to the next page on the collision list.

454 If the offset matches, this returns it or moves on.

448 Moves to the next page on the hash list.

459 Returns the found page or NULL if not.

J.1.4.4 Function: `find_lock_page()` (*include/linux/pagemap.h*)

This is the top-level function for searching the pagecache for a page and having it returned in a locked state.

```

84 #define find_lock_page(mapping, index) \
85     __find_lock_page(mapping, index, page_hash(mapping, index))

```

85 Calls the core function `__find_lock_page()` after looking up what hash bucket this page is using with `page_hash()`.


```
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 Uses `_find_page_nolock()` (See Section J.1.4.3) to locate the page in the pagecache.

983-984 If the page was found, this takes a reference to it.

985 Tries and locks the page with `TryLockPage()`. This macro is just a wrapper around `test_and_set_bit()`, which attempts to set the `PG_locked` bit in the `page->flags`.

986-988 If the lock failed, this releases the `pagecache_lock` spinlock and calls `lock_page()` (See Section B.2.1.1) to lock the page. It is likely this function will sleep until the page lock is acquired. When the page is locked, it acquires the `pagecache_lock` spinlock again.

991 If the mapping and index no longer match, it means that this page was reclaimed while we were asleep. The page is unlocked, and the reference dropped before searching the pagecache again.

998 Returns the page in a locked state or `NULL` if it was not in the pagecache.


```

180     DEBUG_LRU_PAGE(page);           \
181     SetPageActive(page);             \
182     list_add(&(page)->lru, &active_list); \
183     nr_active_pages++;               \
184 } while (0)

```

180 The `DEBUG_LRU_PAGE()` macro will call `BUG()` if the page is already on the LRU list or is marked active.

181 Updates the ~~yes~~ `PageActive` of the page to show it is active.

182 Adds the page to the `active_list`.

183 Updates the count of the number of pages in the `active_list`.

J.2.1.3 Function: `add_page_to_inactive_list()` (*include/linux/swap.h*)

This adds the page to the `inactive_list`.

```

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 The `DEBUG_LRU_PAGE()` macro will call `BUG()` if the page is already on the LRU list or is marked active.

189 Adds the page to the `inactive_list`.

190 Updates the count of the number of inactive pages on the list.

J.2.2 Deleting Pages From the LRU Lists

J.2.2.1 Function: `lru_cache_del()` (*mm/swap.c*)

This acquires the lock protecting the LRU lists before calling `__lru_cache_del()`.

```

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 Acquires the LRU lock.

93 `__lru_cache_del()` does the real work of removing the page from the LRU lists.

94 Releases the LRU lock.

J.2.3 Activating Pages

J.2.3.1 Function: mark_page_accessed() (*mm/pmap.c*)

This marks that a page has been referenced. If the page is already on the active_list or the referenced flag is clear, the referenced flag will be set. If it is in the inactive_list and the referenced flag has been set, activate_page() will be called to move the page to the top of the active_list.

```

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 If the page is on the inactive_list (!PageActive()) and has been referenced recently (PageReferenced()), activate_page() is called to move it to the active_list.

1338 Otherwise, it marks the page as being referenced.

J.2.3.2 Function: activate_lock() (*mm/swap.c*)

This acquires the LRU lock before calling activate_page_nolock(), which moves the page from the inactive_list to the 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 Acquires the LRU lock.

50 Calls the main work function.

51 Releases the LRU lock.

J.2.3.3 Function: activate_page_nolock() (*mm/swap.c*)

This moves the page from the inactive_list to the 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 }
```


J.3 Refilling inactive_list

Contents

J.3 Refilling inactive_list	552
J.3.1 Function: refill_inactive()	552

This section covers how pages are moved from the active lists to the inactive lists.

J.3.1 Function: refill_inactive() (*mm/vmscan.c*)

This moves `nr_pages` from the `active_list` to the `inactive_list`. The parameter `nr_pages` is calculated by `shrink_caches()` and is a number that tries to keep the active list two-thirds the size of the pagecache.

```

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 Acquires the lock protecting the LRU list.

538 Takes the last entry in the `active_list`.

539-555 Keeps moving pages until `nr_pages` are moved or the `active_list` is empty.

542 Gets the `struct page` for this entry.

J.4 Reclaiming Pages From the LRU Lists

Contents

J.4 Reclaiming Pages From the LRU Lists	554
J.4.1 Function: shrink_cache()	554

This section covers how a page is reclaimed after it has been selected for pageout.

J.4.1 Function: 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 The parameters are as follows:

- ~~%~~**nr_pages** The number of pages to swap out
- ~~%~~**classzone** The zone we are interested in swapping pages out for. Pages not belonging to this zone are skipped.
- ~~%~~**gfp_mask** The gfp mask determining what actions may be taken, such as if ~~%~~system operations may be performed
- ~~%~~**priority** The priority of the function, which starts at DEF_PRIORITY (6) and decreases to the highest priority of 1

341 The maximum number of pages to scan is the number of pages in the active_list divided by the priority. At lowest priority, one-sixth of the list may be scanned. At highest priority, the full list may be scanned.

342 The maximum amount of process-mapped pages allowed is either one-tenth of the max_scan value or nr_pages $2^{10 - \text{priority}}$. If this number of pages is found, whole processes will be swapped out.

344 Locks the LRU list.

345 Keeps scanning until max_scan pages have been scanned or the inactive_list is empty.

```

346     struct page * page;
347
348     if (unlikely(current->need_resched)) {

```


361-362 Moves the page to the top of the `inactive_list` so that, if the page is not freed, we can just continue knowing that it will be simply examined later.

368-369 If the page count has already reached 0, this skips over it. In `__free_pages()`, the page count is dropped with `put_page_testzero()` before `__free_pages_ok()` is called to free it. This leaves a window where a page with a zero count is left on the LRU before it is freed. A special case to trap this is at the beginning of `__free_pages_ok()`.

371-372 Skips over this page if it belongs to a zone we are not currently interested in.

375-376 If the page is mapped by a process, goto `page_mapped` where the `max_mapped` is decremented and the next page is examined. If `max_mapped` reaches 0, process pages will be swapped out.

```

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         }

```

In this block, a page is locked, and the launder bit is set. In this case, it is the second time this page has been found dirty. The ~~1st~~ time it was scheduled for I/O and placed back on the list. This time we wait until the I/O is complete and then try to free the page.

382-383 If we could not lock the page, the `PG_launder` bit is set, and the GFP ~~1st~~ ~~as~~ allow the caller to perform FS operations. Then...

384 Takes a reference to the page so that it does not disappear while we sleep.

385 Frees the LRU lock.

386 Waits until the I/O is complete.

387 Releases the reference to the page. If it reaches 0, the page will be freed.

388 Reacquires the LRU lock.

390 Moves to the next page.

```

392
393         if (PageDirty(page) &&
            is_page_cache_freeable(page) &&
            page->mapping) {

```



```

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                     spin_lock(&pagemap_lru_lock);
446                 }
447             } else {
448                 /* failed to drop the buffers so stop here */
449                 UnlockPage(page);
450                 page_cache_release(page);
451
452                 spin_lock(&pagemap_lru_lock);
453                 continue;
454             }
455         }
456     }

```

Page has buffers associated with it that must be freed.

425 Releases the LRU lock because we may sleep.

428 Takes a reference to the page.

430 Calls `try_to_release_page()`, which will attempt to release the buffers associated with the page. It returns 1 if it succeeds.

431-447 This is a case where an anonymous page that was in the swap cache has now had its buffers cleared and removed. Because it was on the swap cache, it was placed on the LRU by `add_to_swap_cache()`, so it removes it now from the LRU and drops the reference to the page. In `swap_writepage()`, it calls `remove_exclusive_swap_page()`, which will delete the page from the swap


```

493     if (PageDirty(page)) {
494         spin_unlock(&pagecache_lock);
495         UnlockPage(page);
496         continue;
497     }

```

493-497 The page has no references, but could have been dirtied by the last process to free it if the dirty bit was set in the PTE. It is left in the pagecache and will get laundered later. After it has been cleaned, it can be safely deleted.

```

498
499     /* point of no return */
500     if (likely(!PageSwapCache(page))) {
501         __remove_inode_page(page);
502         spin_unlock(&pagecache_lock);
503     } else {
504         swp_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 If the page does not belong to the swapcache, it is part of the inode queue so it is removed.

504-508 Removes it from the swapcache because there are no more references to it.

511 Deletes it from the pagecache.

512 Unlocks the page.

515 Frees the page.

517-518 Decrements nr_page and moves to the next page if it is not 0.

519 If it reaches 0, the work of the function is complete.

J.5 Shrinking All Caches

Contents

J.5 Shrinking All Caches	562
J.5.1 Function: shrink_caches()	562
J.5.2 Function: try_to_free_pages()	563
J.5.3 Function: try_to_free_pages_zone()	564

J.5.1 Function: shrink_caches() (*mm/vmscan.c*)

The call graph for this function is shown in Figure 10.4.

```

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 *
572            nr_active_pages / ((nr_inactive_pages + 1) * 2);
573     refill_inactive(ratio);
574
575     nr_pages = shrink_cache(nr_pages, classzone, gfp_mask, priority);
576     if (nr_pages <= 0)
577         return 0;
578
579     shrink_dcache_memory(priority, gfp_mask);
580     shrink_icache_memory(priority, gfp_mask);
581 #ifdef CONFIG_QUOTA
582     shrink_dqcache_memory(DEF_PRIORITY, gfp_mask);
583 #endif
584     return nr_pages;
585 }

```

560 The parameters are as follows:

- `classzone` is the zone that pages should be freed from.
- `priority` determines how much work will be done to free pages.
- `gfp_mask` determines what sort of actions may be taken.
- `nr_pages` is the number of pages remaining to be freed.


```

622     current->flags |= pf_free_pages;
623     return error;
624 }

```

614-615 This clears the PF_FREE_PAGES ~~Yes~~ if it is set so that pages freed by the process will be returned to the global pool rather than reserved for the process itself.

617-620 Cycles through all nodes and calls try_to_free_pages() for the preferred zone in each node.

618 This function is only called with GFP_NOIO as a parameter. When ANDed with GFP_ZONEMASK, it will always result in 0.

622-623 Restores the process ~~Yes~~ and returns the result.

J.5.3 Function: try_to_free_pages_zone() (*mm/vmscan.c*)

This tries to free SWAP_CLUSTER_MAX pages from the requested zone. As well as being used by kswapd, this function is the entry for the buddy allocator's direct-reclaim path.

```

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      * Mhwahahaha! This is the part I really like. Giggle.
602      */
603     out_of_memory();
604     return 0;
605 }

```

589 Starts with the lowest priority. This is statically ~~Yes~~ defined to be 6.

590 Tries and frees SWAP_CLUSTER_MAX pages. This is statically ~~Yes~~ defined to be 32.

592 pf_gfp_mask() checks the PF_NOIO ~~Yes~~ in the current process ~~Yes~~. If no I/O can be performed, it ensures no incompatible ~~Yes~~ are in the GFP mask.

J.6 Swapping Out Process Pages

Contents

J.6 Swapping Out Process Pages	566
J.6.1 Function: <code>swap_out()</code>	566
J.6.2 Function: <code>swap_out_mm()</code>	568
J.6.3 Function: <code>swap_out_vma()</code>	569
J.6.4 Function: <code>swap_out_pgd()</code>	570
J.6.5 Function: <code>swap_out_pmd()</code>	571
J.6.6 Function: <code>try_to_swap_out()</code>	573

This section covers the path where too many process-mapped pages have been found in the LRU lists. This path will start scanning whole processes and reclaiming the mapped pages.

J.6.1 Function: `swap_out()` (*mm/vmscan.c*)

The call graph for this function is shown in Figure 10.5. This function linearly searches through every process's pagetables trying to swap out `SWAP_CLUSTER_MAX` number of pages. The process it starts with is the `swap_mm`, and the starting address is `mm->swap_address`.

```

296 static int swap_out(unsigned int priority, unsigned int gfp_mask,
    zone_t * classzone)
297 {
298     int counter, nr_pages = SWAP_CLUSTER_MAX;
299     struct mm_struct *mm;
300
301     counter = mmlist_nr;
302     do {
303         if (unlikely(current->need_resched)) {
304             __set_current_state(TASK_RUNNING);
305             schedule();
306         }
307
308         spin_lock(&mmlist_lock);
309         mm = swap_mm;
310         while (mm->swap_address == TASK_SIZE || mm == &init_mm) {
311             mm->swap_address = 0;
312             mm = list_entry(mm->mmlist.next,
                struct mm_struct, mmlist);
313             if (mm == swap_mm)
314                 goto empty;
315             swap_mm = mm;
316         }
317
318         /* Make sure the mm doesn't disappear
319            when we drop the lock.. */
319         atomic_inc(&mm->mm_users);

```


328 If we failed on this pass, this increases the priority so more processes will be scanned.

330 Returns failure.

J.6.2 Function: `swap_out_mm()` (*mm/vmscan.c*)

This walks through each VMA and calls `swap_out_mm()` for each one.

```

256 static inline int swap_out_mm(struct mm_struct * mm, int count,
                               int * mmcounter, zone_t * classzone)
257 {
258     unsigned long address;
259     struct vm_area_struct* vma;
260
265     spin_lock(&mm->page_table_lock);
266     address = mm->swap_address;
267     if (address == TASK_SIZE || swap_mm != mm) {
268         /* We raced: don't count this mm but try again */
269         ++*mmcounter;
270         goto out_unlock;
271     }
272     vma = find_vma(mm, address);
273     if (vma) {
274         if (address < vma->vm_start)
275             address = vma->vm_start;
276
277         for (;;) {
278             count = swap_out_vma(mm, vma, address,
                                   count, classzone);
279             vma = vma->vm_next;
280             if (!vma)
281                 break;
282             if (!count)
283                 goto out_unlock;
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 Acquires the pagetable lock for this mm.

233-234 Skips over this VMA if the VM_RESERVED flag is set. This is used by some device drivers, such as the Small Computer System Interface (SCSI) generic driver.

236 Gets the starting PGD for the address.

238 Marks where the end is and uses BUG() if the starting address is somehow past the end.

240 Cycles through PGDs until the end address is reached.

241 Calls swap_out_pgd() (See Section J.6.4) to keep count of how many more pages need to be freed.

242-243 If enough pages have been freed, this breaks and returns.

244-245 Moves to the next PGD and moves the address to the next PGD-aligned address.

247 Returns the remaining number of pages to be freed.

J.6.4 Function: swap_out_pgd() (*mm/vmscan.c*)

This steps through all PMDs in the supplied PGD and calls swap_out_pmd().

```

197 static inline int swap_out_pgd(struct mm_struct * mm,
                                struct vm_area_struct * vma, pgd_t *dir,
                                unsigned long address, unsigned long end,
                                int count, zone_t * classzone)
198 {
199     pmd_t * pmd;
200     unsigned long pgd_end;
201
202     if (pgd_none(*dir))
203         return count;
204     if (pgd_bad(*dir)) {
205         pgd_ERROR(*dir);
206         pgd_clear(dir);
207         return count;
208     }
209
210     pmd = pmd_offset(dir, address);
211
212     pgd_end = (address + PGDIR_SIZE) & PGDIR_MASK;
213     if (pgd_end && (end > pgd_end))
214         end = pgd_end;
215
216     do {
217         count = swap_out_pmd(mm, vma, pmd,
```



```

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
186                 if (!count) {
187                     address += PAGE_SIZE;
188                     break;
189                 }
190             }
191             address += PAGE_SIZE;
192             pte++;
193         } while (address && (address < end));
194     }

```

163-164 Returns if there is no PMD.

165-169 If the PMD is bad, this ~~yes~~ it as such and returns.

171 Gets the starting PTE.

173-175 Calculates the end to be the end of the PMD or the end of the VMA, whichever is closer.

177-191 Cycles through each PTE.

178 Makes sure the PTE is marked present.

179 Gets the struct page for this PTE.

181 If it is a valid page and it is not reserved, then ...

182 Calls try_to_swap_out().

183-186 If enough pages have been swapped out, this moves the address to the next page and breaks to return.

189-190 Moves to the next page and PTE.

192 Updates the swap_address to show where we last ~~yes~~ished o .

193 Returns the number of pages remaining to be freed.

63-64 If the page belongs to a zone we are not interested in, do not swap it out.

66-67 If the page is already locked for I/O, this skips it.

```

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 Calls the architecture hook to flush this page from all CPUs.

75 Gets the PTE from the pagetables and clears it.

76 Calls the architecture hook to flush the TLB.

78-79 If the PTE was marked dirty, this marks the struct page dirty so that it will be laundered correctly.

```

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;
97             page_cache_release(page);
98             return freeable;
99         }

```

Handles the case where the page is already in the swap cache.

86 Enters this block only if the page is already in the swap cache. Note that it can also be entered by calling goto to the set_swap_pte and drop_pte labels.

87-88 Fills in the index value for the swap entry. swap_duplicate() verifies the swap identity is valid and increases the counter in the swap_map if it is.

90 Fills the PTE with information needed to get the page from swap.

92 Updates RSS to show one less page is being mapped by the process.

93 Unlocks the page.


```
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 Allocates a swap entry for this page.

135-136 If one could not be allocated, it breaks out where the PTE and page will be reattached to the process pagetables.

141 Adds the page to the swap cache.

142 Marks the page as up to date in memory.

143 Marks the page dirty so that it will be written out to swap soon.

144 Goto `set_swap_pte`, which will update the PTE with information needed to get the page from swap later.

147 If the add to swap cache failed, it means that the page was placed in the swap cache already by a readahead, so it drops the work done here.

152 Reattaches the PTE to the page tables.

153 Unlocks the page.

154 Returns that no page was freed.


```

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 Calls `daemonize()`, which will make this a kernel thread, remove the mm context, close all ~~86~~ and reparent the process.

726 Sets the name of the process.

727 Ignores all signals.

741 By setting this ~~88~~, the physical page allocator will always try to satisfy requests for pages. Because this process will always be trying to free pages, it is worth satisfying requests.

746-764 Endlessly loops.

747-748 This adds `kswapd` to the wait queue in preparation to sleep.

750 The memory block function (`mb()`) ensures that all reads and writes that occurred before this line will be visible to all CPUs.

751 `kswapd_can_sleep()` (See Section J.7.2.2) cycles through all nodes and zones checking the `need_balance` ~~90~~. If any of them are set to 1, `kswapd` cannot sleep.

752 By calling `schedule()`, `kswapd` will now sleep until woken again by the physical page allocator in `_alloc_pages()` (See Section F.1.3).

754-755 After it is woken up, `kswapd` is removed from the wait queue because it is now running.

686 The `node_zones` ~~is~~ is an array of all available zones, so adding `i` gives the index.

687-688 If the zone does not need balance, this continues.

689 0 is returned if any zone needs balance, indicating `kswapd` cannot sleep.

692 Returns indicating `kswapd` can sleep if the for loop completes.

J.7.2.4 Function: `kswapd_balance()` (*mm/vmscan.c*)

This continuously cycles through each `pgdat` until none require balancing.

```
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 Cycles through all `pgdats` until none of them report that they need balancing.

675 For each `pgdat`, this calls `kswapd_balance_pgdat()` to check if the node requires balancing. If any node required balancing, `need_more_balance` will be set to 1.

J.7.2.5 Function: `kswapd_balance_pgdat()` (*mm/vmscan.c*)

This function will check if a node requires balance by examining each of the nodes in it. If any zone requires balancing, `try_to_free_pages_zone()` will be called.

```
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)
```


K.5	Deactivating a Swap Area	619
K.5.1	Function: <code>sys_swapoff()</code>	619
K.5.2	Function: <code>try_to_unuse()</code>	623
K.5.3	Function: <code>unuse_process()</code>	627
K.5.4	Function: <code>unuse_vma()</code>	628
K.5.5	Function: <code>unuse_pgd()</code>	629
K.5.6	Function: <code>unuse_pmd()</code>	630
K.5.7	Function: <code>unuse_pte()</code>	631

```

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 Locks the list of swap areas.

108 Gets the next swap area that is to be used for allocating from. This list will be ordered depending on the priority of the swap areas.

109-110 If there are no swap areas, this returns NULL.

111-112 If the accounting says no swap slots are available, this returns NULL.

114-141 Cycles through all swap areas.

115 Gets the current swap info struct from the `swap_info` array.

116 If this swap area is available for writing to and is active, then...

117 Locks the swap area.

118 Calls `scan_swap_map()` (See Section K.1.2), which searches the requested swap map for a free slot.

119 Unlocks the swap device.

120-130 If a slot was free, then...

121 Encodes an identifier for the entry with `SWP_ENTRY()`.

122 Records the next swap area to use.

123-126 If the next area is the end of the list or the priority of the next swap area does not match the current one, this moves back to the head.

126-128 Otherwise, it moves to the next area.

129 Goto out.

132 Moves to the next swap area.


```

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     }

```

At this stage, SWAPFILE_CLUSTER pages have been allocated sequentially, so this finds the next free block of SWAPFILE_CLUSTER pages.

56 Reinitializes the count of sequential pages to allocate to SWAPFILE_CLUSTER.

59 Starts searching at the lowest known free slot.

61 If the offset plus the cluster size is less than the known last free slot, this examines all the pages to see if this is a large free block.

64 Scans from offset to offset + SWAPFILE_CLUSTER.

65-69 If this slot is used, this starts searching again for a free slot, beginning after this known allocated one.

73 A large cluster was found, so this uses it.

```

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;

```

This unusual for loop extract starts scanning for a free page starting from lowest_bit.

77-78 If the slot is in use, this moves to the next one.

K.2 Swap Cache

Contents

K.2 Swap Cache	590
K.2.1 Adding Pages to the Swap Cache	590
K.2.1.1 Function: <code>add_to_swap_cache()</code>	590
K.2.1.2 Function: <code>swap_duplicate()</code>	591
K.2.2 Deleting Pages From the Swap Cache	592
K.2.2.1 Function: <code>swap_free()</code>	592
K.2.2.2 Function: <code>swap_entry_free()</code>	593
K.2.3 Acquiring/Releasing Swap Cache Pages	594
K.2.3.1 Function: <code>swap_info_get()</code>	594
K.2.3.2 Function: <code>swap_info_put()</code>	595
K.2.4 Searching the Swap Cache	596
K.2.4.1 Function: <code>lookup_swap_cache()</code>	596

K.2.1 Adding Pages to the Swap Cache

K.2.1.1 Function: `add_to_swap_cache()` (*mm/swap_state.c*)

The call graph for this function is shown in Figure 11.3. This function wraps around the normal page cache handler. It ~~first~~ checks if the page is already in the swap cache with `swap_duplicate()`, and, if it is not, it calls `add_to_page_cache_unique()` instead.

```

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 A check is made with `PageSwapCache()` before this function is called to make sure the page is not already in the swap cache. This check here ensures


```

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 The parameter is the swap entry to increase the `swap_map` count for.

1167-1169 Gets the `o` set within the `swap_info` for the `swap_info_struct` containing this entry. If it is greater than the number of swap areas, goto `bad_file`.

1170-1171 Gets the relevant `swap_info_struct` and gets the `o` set within its `swap_map`.

1173 Locks the swap device.

1174 Makes a quick sanity check to ensure the `o` set is within the `swap_map` and that the slot indicated has a positive count. A 0 count would mean that the slot is not free, and this is a bogus `swp_entry_t`.

1175-1177 If the count is not `SWAP_MAP_MAX`, this increments it and returns 1 for success.

1178-1183 If not, the count would overflow, so this sets it to `SWAP_MAP_MAX` and reserves the slot permanently. In reality, this condition is virtually impossible.

1185-1187 Unlocks the swap device and returns.

1190-1191 If a bad device was used, this prints out the error message and returns failure.

K.2.2 Deleting Pages From the Swap Cache

K.2.2.1 Function: `swap_free()` (`mm/swap.c`)

This decrements the corresponding `swap_map` entry for the `swp_entry_t`.

K.2.4 Searching the Swap Cache

K.2.4.1 Function: `lookup_swap_cache()` (*mm/swap_state.c*)

This is a top-level function for finding a page in the swap cache.

```

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()` (See Section J.1.4.1) is the principal function for returning the struct page. It uses the normal page hashing and cache functions for quickly finding it.

172 Increases the statistic recording the number of times a page was searched for in the cache.

173-174 If one was found, this increments the successful find count.

175 Returns the struct page or NULL if it did not exist.


```

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 Loops in case `add_to_swap_cache()` fails to add a page to the swap cache.

196 First searches the swap cache with `find_get_page()` (See Section J.1.4.1) to see if the page is already available. Ordinarily, `lookup_swap_cache()` (See Section K.2.4.1) would be called, but it updates statistics (such as the number of cache searches), so `find_get_page()` (See Section J.1.4.1) is called directly.

203-207 If the page is not in the swap cache and we have not allocated one yet, this allocates one with `alloc_page()`.

218 Adds the newly allocated page to the swap cache with `add_to_swap_cache()` (See Section K.2.1.1).

223 Schedules the data to be read with `rw_swap_page()` (See Section K.3.3.1). The page will be returned locked and will be unlocked when I/O completes.

224 Returns the new page.

226 Loops until `add_to_swap_cache()` succeeds or another process successfully inserts the page into the swap cache.

228-229 This is either the error path, or another process added the page to the swap cache for us. If a new page was allocated, this frees it with `page_cache_release()` (See Section J.1.3.2).

230 Returns either the page found in the swap cache or an error.


```

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 This operation should only be made with the page locked.

295-296 If the page is not in the swap cache, then there is nothing to do.

297-298 If there are other users of the page, then it cannot be reclaimed, so it returns.

300 The `swp_entry_t` for the page is stored in `page->index` as explained in Section 2.5.

301 Gets the `swap_info_struct` with `swap_info_get()` (See Section K.2.3.1).

307 If the only user of the swap slot is the swap cache itself (i.e, no process is mapping it), this deletes this page from the swap cache to free the slot. Later, the swap slot usage count will be decremented because the swap cache is no longer using it.

310 If the current user is the only user of this page, it is safe to remove from the swap cache. If another process is sharing it, it must remain here.

311 Deletes from the swap cache.

341 Drops the reference taken to the swap info struct at line 337.

343-352 If the page was located, then we try to reclaim it.

344 Takes a reference to the page so that it will not be freed prematurely.

346-349 The page is deleted from the swap cache if no processes are mapping the page or if the swap area is more than 50 percent full (checked by `vm_swap_full()`).

350 Unlocks the page again.

351 Drops the local reference to the page taken at line 344.

K.3.3 Block I/O

K.3.3.1 Function: `rw_swap_page()` (*mm/page_io.c*)

This is the main function used for reading data from backing storage into a page or writing data from a page to backing storage. Which operation it performs depends on the `rw` parameter `rw`. It is basically a wrapper function around the core function `rw_swap_page_base()`. This simply enforces that the operations are only performed on pages in the swap cache.

```

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` indicates whether a read or write is taking place.

89 Gets the `swp_entry_t` from the index `idx`.

91-92 If the page is not locked for I/O, it is a bug.

93-94 If the page is not in the swap cache, it is a bug.

95 Calls the core function `rw_swap_page_base()`. If it returns failure, the page is unlocked with `UnlockPage()` so that it can be freed.


```

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()` (See Section K.3.3.3) returns either the `kdev_t` or `struct inode` that represents the swap area, whichever is appropriate.

52-55 If the storage area is a partition, then there is only one block to be written, which is the size of a page. Hence, `zones` only has one entry, which is the `o` set within the partition to be written, and the `block_size` is `PAGE_SIZE`.

56 If not, it is a swap ~~%s~~, so each of the blocks in the ~~%s~~ that make up the page has to be mapped with `bmap()` before calling `brw_page()`.

58-59 Calculates what the starting block is.

61 The size of individual block is stored in the superblock information for the ~~%s~~ system the ~~%s~~ resides on.

62-66 Calls `bmap()` for every block that makes up the full page. Each block is stored in the `zones` array for passing to `brw_page()`. If any block fails to be mapped, 0 is returned.

67 Records how many blocks make up the page in `zones_used`.

68 Records which device is being written to.

74 Calls `brw_page()` from the block layer to schedule the I/O to occur. This function returns immediately because the I/O is asynchronous. When the I/O is completed, a callback function (`end_buffer_io_async()`) is called, which unlocks the page. Any process waiting on the page will be woken up at that point.

- 1203 Extracts which area within `swap_info` this entry belongs to.
- 1204-1206 If the index is for an area that does not exist, this prints out an information message and returns. `Bad_file` is a static array declared near the top of `mm/swapfile.c` that says `Bad swap %o entry.%o`
- 1209 Gets the `swap_info_struct` from `swap_info`.
- 1210 Extracts the `o` set within the swap area for this entry.
- 1211-1214 Makes sure the `o` set is not after the end of the `%o`. It prints out the message in `Bad_offset` if it is.
- 1215-1218 If the `o` set is currently not being used, it means that entry is a stale entry, so it prints out the error message in `Unused_offset`.
- 1219-1222 If the swap area is currently not active, this prints out the error message in `Unused_file`.
- 1224 If the swap area is a device, this returns the `kdev_t` in `swap_info_struct→swap_device`.
- 1226-1227 If it is a swap `%o`, this returns the `struct inode`, which is available through `swap_info_struct→swap_file→d_inode`.
- 1229 If not, there is no swap `%o` or device for this entry, so it prints out the error message and returns.


```

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 The two parameters are the path to the swap area and the ~~%s~~ for activation.

872-873 The activating process must have the CAP_SYS_ADMIN capability or be the superuser to activate a swap area.

874 Acquires the Big Kernel Lock (BKL).

875 Locks the list of swap areas.

876 Gets the ~~%s~~ swap area in the swap_info array.

```

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;

```


908 Fills in the `swap_file` with the returned dentry.

909 Similarly, `%s` in the `swap_vfsmnt`.

910 Records the inode of the special `%s`.

911 Now the default error is `-EINVAL`, indicating that the special `%s` was found but it was not a block device or a regular `%s`.

```

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,
928                           BDEV_SWAP);
929         devfs_put_ops(de); /* Decrement module use count
930                             * now we're safe */
931
932         if (error)
933             goto bad_swap_2;
934         set_blocksize(dev, PAGE_SIZE);
935         error = -ENODEV;
936         if (!dev || (blk_size[MAJOR(dev)] &&
937                     !blk_size[MAJOR(dev)][MINOR(dev)]))
938             goto bad_swap;
939         swapfilesize = 0;
940         if (blk_size[MAJOR(dev)])
941             swapfilesize = blk_size[MAJOR(dev)][MINOR(dev)]
942                 >> (PAGE_SHIFT - 10);
943     } else if (S_ISREG(swap_inode->i_mode))
944         swapfilesize = swap_inode->i_size >> PAGE_SHIFT;
945     else
946         goto bad_swap;

```

If a partition, this code confirms the block device before calculating the size of the area, or it obtains it from the inode for the `%s`.

913 Checks if the special `%s` is a block device.


```

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),
963         (char *) swap_header);
964
965     if (!memcmp("SWAP-SPACE", swap_header->magic.magic,10))
966         swap_header_version = 1;
967     else if (!memcmp("SWAPSPACE2", swap_header->magic.magic,10))
968         swap_header_version = 2;
969     else {
970         printk("Unable to find swap-space signature\n");
971         error = -EINVAL;
972         goto bad_swap;
973     }

```

945 The next check makes sure the area is not already active. If it is, the error -EBUSY will be returned.

946-962 Reads through the while swap_info struct and ensures the area to be activated if not already active.

954-959 Allocates a page for reading the swap area information from disk.

961 The function lock_page() locks a page and makes sure it is synced with the disk if it is not backed. In this case, it will just mark the page as locked, which is required for the rw_swap_page_nolock() function.

962 Reads the ~~8~~ page slot in the swap area into swap_header.

964-972 Checks the version based on the swap area information and sets swap_header_version variable with it. If the swap area could not be identified, it returns -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;

```


1001 Exits the switch statement.

```

1003     case 2:
1004         if (swap_header->info.version != 1) {
1005             printk(KERN_WARNING
1006                 "Unable to handle swap header version %d\n",
1007                 swap_header->info.version);
1008             error = -EINVAL;
1009             goto bad_swap;
1010         }
1011
1012         p->lowest_bit = 1;
1013         maxpages = SWP_OFFSET(SWP_ENTRY(0, ~0UL)) - 1;
1014         if (maxpages > swap_header->info.last_page)
1015             maxpages = swap_header->info.last_page;
1016         p->highest_bit = maxpages - 1;
1017
1018         error = -EINVAL;
1019         if (swap_header->info.nr_badpages > MAX_SWAP_BADPAGES)
1020             goto bad_swap;
1021
1022         if (!(p->swap_map = vmalloc(maxpages * sizeof(short)))) {
1023             error = -ENOMEM;
1024             goto bad_swap;
1025         }
1026
1027         error = 0;
1028         memset(p->swap_map, 0, maxpages * sizeof(short));
1029         for (i=0; i<swap_header->info.nr_badpages; i++) {
1030             int page = swap_header->info.badpages[i];
1031             if (page <= 0 ||
1032                 page >= swap_header->info.last_page)
1033                 error = -EINVAL;
1034             else
1035                 p->swap_map[page] = SWAP_MAP_BAD;
1036         }
1037         nr_good_pages = swap_header->info.last_page -
1038             swap_header->info.nr_badpages -
1039             1 /* header page */;
1040         if (error)
1041             goto bad_swap;
1042     }

```

This block reads the header information when the **%6** format is version 2.

1006-1012 Makes absolutely sure we can handle this swap **%6** format and returns **-EINVAL** if we cannot. Remember that, with this version, the **swap_header** struct is placed nicely on disk.

1052-1056 If no good pages were available, this returns an error.

1057 Makes sure the ~~8~~ page in the map containing the swap header information is not used. If it was, the header information would be overwritten the ~~8~~ time this area was used.

1058-1059 Locks the swap list and the swap device.

1060-1062 Fills in the remaining ~~4~~ slots in the `swap_info_struct`.

1063-1064 Updates global statistics for the number of available swap pages (`nr_swap_pages`) and the total number of swap pages (`total_swap_pages`).

1065-1066 Prints an informational message about the swap activation.

```

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 Inserts the new swap area into the correct slot in the swap list based on priority.

1082 Unlocks the swap device.

1083 Unlocks the swap list.

1084-1085 Returns success.

```

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;

```



```
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 Calculates how much memory the system has in megabytes.

105 In low memory systems, this sets `page_cluster` to 2, which means that, at most, four pages will be paged in from disk during readahead.

108 If not, readahead will be eight pages.

728-729 Only the superuser or a process with CAP_SYS_ADMIN capabilities may deactivate an area.

731-732 Acquires information about the special `%s` representing the swap area with `user_path_walk()`. Goto out if an error occurred.

```

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;

```

Acquires the BKL, ~~finds~~ the `swap_info_struct` for the area to be deactivated and removes it from the swap list.

735 Acquires the BKL.

737 Locks the swap list.

738-745 Traverses the swap list and ~~finds~~ the `swap_info_struct` for the requested area. It uses the `dentry` to identify the area.

747-750 If the struct could not be found, this returns.

781 Marks the area as safe to write to again.

782-783 Unlocks the swap list and returns.

```

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 }
```

This block is used if the swap area was successfully deactivated to close the block device and mark the `swap_info_struct` free.

785-786 Closes the block device.

787 Releases the path information.

789-790 Acquires the swap list and swap device lock.

791-799 Resets the fields in `swap_info_struct` to default values.

800-801 Releases the swap list and swap device.

801 Frees the memory used for the `swap_map`.

806 Releases the BKL.

807 Releases the path information in the event we reached here by the error path.

809 Returns success or failure.


```

584         atomic_inc(&init_mm.mm_users);
585     }

```

556 This is the beginning of the major loop in this function. Starting from the beginning of the `swap_map`, it searches for the next entry to be freed with `find_next_to_unuse()` until all swap map entries have been paged in.

562-564 Gets the `swp_entry_t` and calls `read_swap_cache_async()` (See Section K.3.1.1) to ~~get~~ the page in the swap cache or to have a new page allocated for reading in from the disk.

565-576 If we failed to get the page, it means the slot has already been freed independently by another process or thread (process could be exiting elsewhere) or we are out of memory. If independently freed, we continue to the next map, or we return `-ENOMEM`.

581 Checks to make sure this mm is not exiting. If it is, it decrements its count and goes back to `init_mm`.

```

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 = shmem_unuse(entry, page);
609         else
610             unuse_process(start_mm, entry, page);
611     }

```

595 Waits on the page to complete I/O. After it returns, we know for a fact the page exists in memory with the same information as that on disk.

618 Locks the mm list.

619-632 Keeps searching until all mm_structs have been found. Do not traverse the full list more than once.

621 Gets the mm_struct for this list entry.

623-627 Calls shmem_unuse()(See Section L.6.2) if the mm is init_mm because that indicates that is a page from the virtual ~~fs~~ system. If not, it calls unuse_process() (See Section K.5.3) to traverse the current process's pagetables searching for the swap entry. If found, the entry will be freed, and the page reinstantiated in the PTE.

630-633 Records if we need to start searching mm_structs starting from init_mm again.

```

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 If the swap map entry is permanently mapped, we have to hope that all processes have their PTEs updated to point to the page and, in reality, that the swap map entry is free. In reality, it is highly unlikely a slot would be permanently reserved in the ~~fs~~ place.

654-661 Locks the list and swap device, sets the swap map entry to 1, unlocks them again and records that a reset overflow occurred.

```

683         if ((*swap_map > 1) && PageDirty(page) &&
684             PageSwapCache(page)) {
685             rw_swap_page(WRITE, page);
686             lock_page(page);
687         }
687         if (PageSwapCache(page)) {
688             if (shmem)
689                 swap_duplicate(entry);
690             else
691                 delete_from_swap_cache(page);
692         }

```

683-686 In the very rare event a reference still exists to the page, this writes the page back to disk so, at least if another process really has a reference to it, it will copy the page back in from disk correctly.


```

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's 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 Locks the process pagetables.

463 Moves through every VMA managed by this mm. Remember that one page frame could be mapped in multiple locations.

462 Gets the PGD managing the beginning of this VMA.

465 Calls `unuse_vma()` (See Section K.5.4) to search the VMA for the page.

467-468 The full mm has been searched, so this unlocks the process pagetables and returns.

K.5.4 Function: `unuse_vma()` (*mm/swap.c*)

This function searches the requested VMA for pagetable entries mapping the page and using the given swap entry. It calls `unuse_pgd()` for every PGD that this VMA maps.

```

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 }

```



```

436     } while (address && (address < end));
437 }

```

416-417 If there is no PGD here, this returns.

418-422 If the PGD is bad, this sets the appropriate error, clears the PGD and returns. There are very few architectures where this condition can occur.

423 Gets the address of the ~~8th~~ PMD in this PGD.

424 Calculates ~~offset~~ as the offset within the PGD the address is for. Remember that on the ~~8th~~ time this function is called, it might be searching a partial PGD.

425 Aligns the address to the PGD.

426 Calculates the end address of the search.

427-428 If the end is beyond this PGD, this sets the end just to the end of this PGD.

429-430 If the starting address is after the end address, something is very seriously wrong.

431-436 Steps through the PGD in PMD_SIZE-sized strides and calls `unuse_pmd()` (See Section K.5.6) for every PMD in this PGD.

K.5.6 Function: `unuse_pmd()` (*mm/swap.c*)

This function searches the requested PMD for pagetable entries mapping the page and using the given swap entry. It calls `unuse_pte()` for every PTE this PMD maps.

```

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     swp_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;

```


372-373 If there is no PTE or it is already present (meaning there is no way this entry is mapped here), this returns.

374 Otherwise, we have found the entry we are looking for, so it takes a reference to the page because a new PTE is about to map it.

375 Updates the PTE to map page.

376 Frees the swap entry.

377 Increments the RSS count for this process.

L.4.2	Linking	664
L.4.2.1	Function: <code>shmem_link()</code>	664
L.4.3	Unlinking	665
L.4.3.1	Function: <code>shmem_unlink()</code>	665
L.4.4	Making Directories	665
L.4.4.1	Function: <code>shmem_mkdir()</code>	665
L.4.5	Removing Directories	666
L.4.5.1	Function: <code>shmem_rmdir()</code>	666
L.4.5.2	Function: <code>shmem_empty()</code>	666
L.4.5.3	Function: <code>shmem_positive()</code>	667
L.5	Page Faulting Within a Virtual File	668
L.5.1	Reading Pages During Page Fault	668
L.5.1.1	Function: <code>shmem_nopage()</code>	668
L.5.1.2	Function: <code>shmem_getpage()</code>	669
L.5.2	Locating Swapped Pages	676
L.5.2.1	Function: <code>shmem_alloc_entry()</code>	676
L.5.2.2	Function: <code>shmem_swp_entry()</code>	676
L.6	Swap Space Interaction	679
L.6.1	Function: <code>shmem_writepage()</code>	679
L.6.2	Function: <code>shmem_unuse()</code>	681
L.6.3	Function: <code>shmem_unuse_inode()</code>	682
L.6.4	Function: <code>shmem_find_swp()</code>	685
L.7	Setting Up Shared Regions	686
L.7.1	Function: <code>shmem_zero_setup()</code>	686
L.7.2	Function: <code>shmem_file_setup()</code>	686
L.8	System V IPC	689
L.8.1	Creating a SYSV Shared Region	689
L.8.1.1	Function: <code>sys_shmget()</code>	689
L.8.1.2	Function: <code>newseg()</code>	690
L.8.2	Attaching a SYSV Shared Region	692
L.8.2.1	Function: <code>sys_shmat()</code>	692

```

1585         shmem_set_size(SHMEM_SB(shm_mnt->mnt_sb),
                        ULONG_MAX, ULONG_MAX);
1586         return 0;
1587
1588 out1:
1589 #ifdef CONFIG_TMPFS
1590         unregister_filesystem(&shmem_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 The `shm` filesystem is only mountable if `CONFIG_TMPFS` is defined at compile time. Even if it is not specified, a `tmpfs` will still be set up for anonymous shared memory resulting from a `fork()`.

1553 `DECLARE_FSTYPE()`, declared in `<linux/fs.h>`, declares `tmpfs_fs_type` as type `struct file_system_type` and `%s` in four `%ds`. `tmpfs` is its human-readable name. `shmem_read_super()` is the function that is used to read the superblock for the `%s` system (a detailed description of superblocks and how they pertain to `%s` systems is beyond the scope of this book). `FS_LITTER` is a `%s` that indicates the `%s` system tree should be maintained in the dcache. Finally, the macro sets the module owner of the `%s` system to be the module loading the `%s` system.

1560 `__init` places this function in the init section. This means that, after the kernel has finished bootstrapping, the code for the function will be removed.

1564-1568 Registers the `%s` system `tmpfs_fs_type`, which was declared in line 1433. If it fails, goto `out3` where the appropriate error will be returned.

1569-1574 If `tmpfs` is specified at configure time, this registers the `shmem` system. If it fails, goto `out2` where `tmpfs_fs_type` will be unregistered before returning the error.

1575 If `/dev/` is being managed by the device `%s` system (`devfs`), this creates a new `shm` directory. If the kernel does not use `devfs`, the system administrator must manually create the directory.

1577 `kern_mount()` mounts a `%s` system internally. In other words, the `%s` system is mounted and active, but it is not visible to the user anywhere in the VFS. The mount point is `shm_mnt`, which is local to the `shmem.c` `%s` and of type `struct vfsmount`. This variable is needed for searching the `%s` system and for unmounting it later.


```

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 The parameters are the following:

- %sb is the super_block to populate.
- %data contains the mount arguments.
- %silent is unused in this function.

1457-1459 Sets the default mode, uid and gid. These may be overridden with the parameters passed as mount options.

1460 Each super_block is allowed to have a %system-specific struct that is contained within a union called super_block->u. The macro SHMEM_SB() returns the struct shmem_sb_info contained within this union.

1467 si_meminfo() populates struct sysinfo with total memory, available memory and usage statistics. The function is defined in arch/i386/mm/init.c and is architecture dependent.

1468 By default, this only allows the %system to consume half of total available physical memory.

1471-1472 If tmpfs is available, this parses the mount options and allows them to override the defaults.

1475 Acquires the lock protecting sbinfo, which is the struct shmem_sb_info in the super_block.

1483 Populates the sb and sbinfo %ds.

- 868 Calculates the number of `blocks` currently in use by the `%s` system. On initial mount, this is unimportant, but, if the `%s` system is being remounted, the function must ensure that the new `%s` system is not too small.
- 869 Calculates the number of `inodes` currently in use.
- 871-872 If the remounted `%s` system would have too few blocks to store the current information, goto out to return `-EINVAL`.
- 873-874 Similarly, makes sure there are enough available inodes or returns `-EINVAL`.
- 875 It is safe to mount the `%s` system, so this sets `error` to 0 indicating that this operation will be successful.
- 876-877 Sets the maximum number of blocks and number of available blocks in the `%s` system's `superblock` `info` struct.
- 878-879 Sets the maximum and available number of inodes.
- 881 Unlocks the `%s` system's `superblock` `info` struct.
- 882 Returns 0 if successful or `-EINVAL` if not.

- 1144 If the inode was successfully created, this updates the directory statistics and instantiates the new `%s`.
- 1145 Updates the size of the directory.
- 1146 Updates the `ctime` and `mtime` `%s`.
- 1147 Instantiates the inode.
- 1148 Takes a reference to the dentry so that it will be pinned and not accidentally reclaimed during pageout. Unlike normal `%s`, there is no automatic way of recreating dentries after they are deleted.
- 1149 Indicates the call ended successfully.
- 1151 Returns success or `-ENOSPC` on error.

L.2.3 Function: `shmem_get_inode()` (*mm/shmem.c*)

```

809 struct inode *shmem_get_inode(struct super_block *sb,
                                int mode,
                                int dev)
810 {
811     struct inode *inode;
812     struct shmem_inode_info *info;
813     struct shmem_sb_info *sbinfo = SHMEM_SB(sb);
814
815     spin_lock(&sbinfo->stat_lock);
816     if (!sbinfo->free_inodes) {
817         spin_unlock(&sbinfo->stat_lock);
818         return NULL;
819     }
820     sbinfo->free_inodes--;
821     spin_unlock(&sbinfo->stat_lock);
822
823     inode = new_inode(sb);

```

This preamble section is responsible for updating the free inode count and allocating an inode with `new_inode()`.

- 815 Acquires the `sbinfo` spinlock because it is about to be updated.
- 816-819 Makes sure there are free inodes, and if not, it returns `NULL`.
- 820-821 Updates the free inode count and frees the lock.
- 823 `new_inode()` is part of the `%s` system layer and declared in `<linux/fs.h>`. Exactly how it works is beyond the scope of this document, but the summary is simple. It allocates an inode from the slab allocator, zeros most `%s` and populates `inode->i_sb`, `inode->i_dev` and `inode->i_blkbits` based on information in the super block.

835-836 Initializes the inodes semaphore and spinlock.

836-856 Determines how to ~~to~~ the remaining ~~ids~~ based on the mode ~~ids~~ passed in.

838 In this case, a special inode is being created. Specifically, this is while the ~~ids~~ system is being mounted and the root inode is being created.

840-846 Creates an inode for a regular ~~ids~~. The main point to note here is that the `inode->i_op` and `inode->i_fop` ~~ids~~ are set to `shmem_inode_operations`, and `shmem_file_operations`, respectively.

847-852 Creates an inode for a new directory. The `i_nlink` and `i_size` ~~ids~~ are updated to show the increased number of ~~ids~~ and the size of the directory. The main point to note here is that the `inode->i_op` and `inode->i_fop` ~~ids~~ are set to `shmem_dir_inode_operations` and `dcach_dir_ops`, respectively.

854-855 If linking a ~~ids~~, this does nothing for now because it is handled by the parent function `shmem_link()`.

858 Returns the new inode or NULL if it could not be created.

L.3.2 Reading Files

L.3.2.1 Function: `shmem_file_read()` (*mm/shmem.c*)

This is the top-level function called for `read()`ing a tmpfs %o.

```

1088 static ssize_t shmem_file_read(struct file *filp, char *buf,
                                size_t count, loff_t *ppos)
1089 {
1090     read_descriptor_t desc;
1091
1092     if ((ssize_t) count < 0)
1093         return -EINVAL;
1094     if (!access_ok(VERIFY_WRITE, buf, count))
1095         return -EFAULT;
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 The parameters are the following:

- %filp is a pointer to the struct file being read.
- %buf is the buffer that should be read.
- %count is the number of bytes that should be read.
- %ppos is the current position.

1092-1093 count cannot be negative.

1094-1095 `access_ok()` ensures that it is safe to write count number of bytes to the userspace buffer. If it can't, -EFAULT will be returned.

1099-1102 Initializes a `read_descriptor_t` struct, which will eventually be passed to `file_read_actor()` (See Section L.3.2.3).

1104 Calls `do_shmem_file_read()` to start performing the actual read.

1105-1106 Returns the number of bytes that were written to the userspace buffer.

1107 If none were written, it returns the error.


```

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
1063     ret = file_read_actor(desc, page, offset, nr);
1064     offset += ret;
1065     index += offset >> PAGE_CACHE_SHIFT;
1066     offset &= ~PAGE_CACHE_MASK;
1067
1068     page_cache_release(page);
1069     if (ret != nr || !desc->count)
1070         break;
1071 }
1072
1073 *ppos = ((loff_t) index << PAGE_CACHE_SHIFT) + offset;
1074 filp->f_reada = 1;
1075 UPDATE_ATIME(inode);
1076 }

```

1005-1006 Retrieves the inode and mapping using the struct file.

1009 index is the page index within the `%p` that contains the data.

1010 offset is the offset within the page that is currently being read.

1012-1081 Loops until the requested number of bytes has been read. `nr` is the number of bytes that are still to be read within the current page. `desc->count` starts as the number of bytes to read and is decremented by `file_read_actor()` (See Section L.3.2.3).

1016-1018 `end_index` is the index of the last page in the `%p`. It breaks when the end of the `%p` is reached.

1019-1023 When the last page is reached, this sets `nr` to be the number of bytes to be read within this page. If the `%p` pointer is after `nr`, this breaks because there is no more data to be read. This could happen after the `%p` was truncated.

1025-1030 `shmem_getpage()` (See Section L.5.1.2) will locate the requested page in the page cache, swap cache or page it in. If an error occurs, this records it in `desc->error` and returns.


```

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 The parameters are the following:

- ~~%o~~desc is a structure containing information about the read, including the buffer and the total number of bytes that are to be read from this ~~%o~~.
- ~~%o~~page is the page containing ~~%o~~ data that is to be copied to userspace.
- ~~%oo~~ offset is the offset within the page that is being copied.
- ~~%o~~size is the number of bytes to be read from page.

1672 count is now the number of bytes that are to be read from the ~~%o~~.

1674-1675 Makes sure to not read more bytes than are requested.

1677 Maps the page into low memory with kmap(). See Section I.1.1.

1678 Copies the data from the kernel page to the userspace buffer.

1679 Unmaps the page. See Section I.3.1.

1681-1684 If all the bytes were not copied, it must be because the buffer was not accessible. This updates size so that desc->count will reflect how many bytes are still to be copied by the read. -EFAULT will be returned to the process performing the read.

1685-1687 Updates the desc struct to show the current status of the read.

1688 Returns the number of bytes that were written to the userspace buffer.

947 Clears the SUID bit if it is set.

948 Updates the inodes ctime and 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 don't 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         }
```


L.3.4 Symbolic Linking

L.3.4.1 Function: `shmem_symlink()` (*mm/shmem.c*)

This function is responsible for creating a symbolic link `symname` and for deciding where to store the information. The name of the link will be stored in the inode if the name is small enough and in a pageframe otherwise.

```

1272 static int shmem_symlink(struct inode * dir,
                           struct dentry *dentry,
                           const char * symname)
1273 {
1274     int error;
1275     int len;
1276     struct inode *inode;
1277     struct page *page = NULL;
1278     char *kaddr;
1279     struct shmem_inode_info *info;
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;

```

This block performs basic sanity checks and creates a new inode for the symbolic link.

1272 The parameter `symname` is the name of the link to create.

1281 Calculates the length (`len`) of the link.

1282-1283 If the name is larger than a page, this returns `-ENAMETOOLONG`.

1285-1287 Allocates a new inode. Returns `-ENOSPC` if it fails.

1289 Gets the private information struct.

1290 The size of the inode is the length of the link.

```

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 {

```



```

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 Increments the size of the directory as a new inode has been added.
BOGO_DIRENT_SIZE is just a pseudosize of inodes so that ls output looks nice.

1312 Updates the i_ctime and i_mtime.

1313-1314 Instantiates the inode.

1315 Returns success.

L.3.4.2 Function: 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 The link name is contained within the inode, so it passes it as a parameter to the VFS layer with `vfs_readlink()`.

L.3.4.3 Function: 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 The link name is contained within the inode, so it passes it as a parameter to the VFS layer with `vfs_followlink()`.

L.3.4.4 Function: shmem_readlink() (*mm/shmem.c*)

```

1328 static int shmem_readlink(struct dentry *dentry,
                            char *buffer, int buflen)
1329 {
1330     struct page *page = NULL;
1331     int res = shmem_getpage(dentry->d_inode, 0, &page, SGP_READ);
1332     if (res)

```


L.3.5 Synchronizing

L.3.5.1 Function: `shmem_sync_file()` (*mm/shmem.c*)

This function simply returns 0 because the file exists only in memory and does not need to be synchronized with a file on disk.

```
1446 static int shmem_sync_file(struct file * file,
                             struct dentry *dentry,
                             int datasync)
1447 {
1448     return 0;
1449 }
```



```

370     sbinfo->free_blocks += freed;
371     inode->i_blocks -= freed*BLOCKS_PER_PAGE;
372     spin_unlock(&sbinfo->stat_lock);
373 }

```

353 Gets the private ~~%s~~ system information for this inode with `SHMEM_I()`.

354 Gets the superblock private information.

358 Updates the ctime and mtime for the inode.

359 Gets the index of the page that is the new end of the ~~%s~~. The old size is stored in `info->next_index`.

360-361 If the ~~%s~~ is being expanded, this just returns because the global zero page will be used to represent the expanded region.

363 Acquires the private `info` spinlock.

364-365 Continually calls `shmem_truncate_indirect()` until the ~~%s~~ is truncated to the desired size.

366 It is a bug if the `shmem_info_info` struct indicates that more pages are swapped out than there are pages in the ~~%s~~.

367 Releases the private `info` spinlock.

369 Acquires the superblock private `info` spinlock.

370 Updates the number of free blocks available.

371 Updates the number of blocks being used by this inode.

372 Releases the superblock private `info` spinlock.

L.4.1.2 Function: `shmem_truncate_indirect()` (*mm/shmem.c*)

This function locates the last doubly-indirect block in the inode and calls `shmem_truncate_direct()` to truncate it.

```

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;

```


baseidx is SHMEM_NR_DIRECT, which is the page index at the beginning of info->i_indirect. At this point, len is the number of pages in the %s, so the number of direct blocks is subtracted to leave the remaining number of pages.

330-339 If not, this is a triply-indexed block, so the next level must be traversed before the base, baseidx and len are calculated.

341-344 If the %s is going to be bigger after the truncation, this updates next_index to the new end of the %s and makes start the beginning of the indirect block.

344-347 If the %s is being made smaller, this moves the current end of the %s to the beginning of this indirect block that is about to be truncated.

348 If there is a block at base, this calls shmem_truncate_direct() to truncate pages in it.

L.4.1.3 Function: shmem_truncate_direct() (*mm/shmem.c*)

This function is responsible for cycling through an indirect block and calling shmem_free_swap for each page that contains swap vectors that are to be truncated.

```

264 static inline unsigned long
265 shmem_truncate_direct(struct shmem_inode_info *info,
                        swp_entry_t ***dir,
                        unsigned long start, unsigned long len)
266 {
267     swp_entry_t **last, **ptr;
268     unsigned long off, freed_swap, freed = 0;
269
270     last = *dir + (len + ENTRIES_PER_PAGE - 1) / ENTRIES_PER_PAGE;
271     off = start % ENTRIES_PER_PAGE;
272
273     for (ptr = *dir + start/ENTRIES_PER_PAGE;
          ptr < last;
          ptr++, off = 0) {
274         if (!*ptr)
275             continue;
276
277         if (info->swapped) {
278             freed_swap = shmem_free_swap(*ptr + off,
279                                         *ptr + ENTRIES_PER_PAGE);
280             info->swapped -= freed_swap;
281             freed += freed_swap;
282         }
283
284         if (!off) {
285             freed++;
286             free_page((unsigned long) *ptr);

```


245-251 Loops through each of the swap entries to be freed.

246-250 If a swap entry exists, this frees it with `free_swap_and_cache()` and sets the swap entry to 0. It increments the number of pages freed.

252 Returns the total number of pages freed.

L.4.2 Linking

L.4.2.1 Function: `shmem_link()` (*mm/shmem.c*)

This function creates a hard link with `dentry` to `old_dentry`.

```
1172 static int shmem_link(struct dentry *old_dentry,
                        struct inode *dir,
                        struct dentry *dentry)
1173 {
1174     struct inode *inode = old_dentry->d_inode;
1175
1176     if (S_ISDIR(inode->i_mode))
1177         return -EPERM;
1178
1179     dir->i_size += BOGO_DIRENT_SIZE;
1180     inode->i_ctime = dir->i_ctime = dir->i_mtime = CURRENT_TIME;
1181     inode->i_nlink++;
1182     atomic_inc(&inode->i_count);
1183     dget(dentry);
1184     d_instantiate(dentry, inode);
1185     return 0;
1186 }
```

1174 Gets the inode corresponding to `old_dentry`.

1176-1177 If it is linking to a directory, this returns `-EPERM`. Strictly speaking, root should be allowed to hard-link directories, although it is not recommended because of the possibility of creating a loop within the ~~%s~~ system that utilities like ~~%d~~ get lost in. `tmpfs` simply does not allow the hard-linking of directories.

1179 Increments the size of the directory with the new link.

1180 Updates the directories `mtime` and `ctime` and updates the inode `ctime`.

1181 Increments the number of links leading to `inode`.

1183 Gets an extra reference to the new `dentry` with `dget()`.

1184 Instantiates the new `dentry`.

1185 Returns success.

L.4.5 Removing Directories

L.4.5.1 Function: `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 Checks to see if the directory is empty with `shmem_empty()` (See Section L.4.5.2). If it is not, it returns `-ENOTEMPTY`.

1237 Decrements the parent directory's `i_nlink` field.

1238 Returns the result of `shmem_unlink()` (See Section L.4.3.1), which should delete the directory.

L.4.5.2 Function: `shmem_empty()` (*mm/shmem.c*)

This function checks to see if a directory is empty or not.

```

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;
1220 }

```

1205 The `dcache_lock` protects many things, but it mainly protects `dcache` lookups, which is what will be required for this function, so this acquires it.

L.5 Page Faulting Within a Virtual File

Contents

L.5 Page Faulting Within a Virtual File	668
L.5.1 Reading Pages During Page Fault	668
L.5.1.1 Function: <code>shmem_nopage()</code>	668
L.5.1.2 Function: <code>shmem_getpage()</code>	669
L.5.2 Locating Swapped Pages	676
L.5.2.1 Function: <code>shmem_alloc_entry()</code>	676
L.5.2.2 Function: <code>shmem_swp_entry()</code>	677

L.5.1 Reading Pages During Page Fault

L.5.1.1 Function: `shmem_nopage()` (*mm/shmem.c*)

This is the top-level `nopage()` function that is called by `do_no_page()` when faulting in a page. This is called regardless of the fault being the ~~1st~~ fault or if it is being faulted in from backing storage.

```

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 The two parameters of relevance are the VMA the fault occurred in and the faulting address.

765 Records the inode that the fault occurred in.

770-772 Calculates the `idx` as the offset in counts of `PAGE_SIZE` within the virtual
~~860~~


```

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 The parameters are the following:

`%inode` is the inode that the fault is occurring in.

`%idx` is the index of the page within the `%i` that is being faulted.

`%pagep` if NULL will become the faulted page if successful. If a valid page is passed in, this function will make sure it is up to date.

`%sgp` indicates what type of access this is, which determines how a page will be located and returned.

586 `SHMEM_I()` returns the `shmem_inode_info` contained with the `%i` system-specific information within the superblock information.

594-595 Makes sure the index is not beyond the end of the `%i`.

605-606 If no page was passed in with the `pagep` parameter, then this tries and locates the page and locks it with `find_lock_page()` (See Section J.1.4.4).

607-608 If the page was found and is up to date, goto done because this function has nothing more to do.

610 Locks the inode private information struct.

611 Searches for the swap entry for this `idx` with `shmem_swp_alloc()`. If one did not previously exist, it will be allocated.

612-616 If an error occurred, this releases the spinlock and returns the error.

```

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             if (!swappage) {
627                 spin_lock(&info->lock);
628                 entry = shmem_swp_alloc(info, idx, sgp);
629                 if (IS_ERR(entry))
630                     error = PTR_ERR(entry);
631                 else if (entry->val == swap.val)

```



```

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     }

```

At this point, the page exists in the swap cache.

658 Deletes the page from the swap cache so that we can attempt to add it to the pagecache.

659-670 If the caller supplied a page with the pagep parameter, this updates pagep with the data in swappage.

671-680 If not, this tries and adds swappage to the pagecache. Note that info->swapped is updated, and the page is marked up to date before the swap entry is freed, with swap_free().

681-689 If we failed to add the page to the page cache, this adds it back to the swap cache with add_to_swap_cache(). The page is marked up to date before being unlocked and goto repeat to try again.


```

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 }

```

If not, a page that is not in the page cache is being written to. It will need to be allocated.

702 Gets the superblock info with `SHMEM_SB()`.

703 Acquires the superblock info spinlock.

704-709 If no free blocks are left in the ~~file~~ system, this releases the spinlocks, sets the return error to `-ENOSPC` and goto failed.

710 Decrements the number of available blocks.

711 Increments the block usage count for the inode.

712 Releases the superblock private information spinlock.

714-715 If a page was not supplied by `pagep`, this allocates a page and swap entry for the new page.

715 Releases the info spinlock because `page_cache_alloc()` may sleep.

716 Allocates a new page.

717-721 If the allocation failed, this frees the block with `shmem_free_block()` and sets the return error to `-ENOMEM` before goto failed.

723 Reacquires the info spinlock.

724 `shmem_swp_entry()` locates a swap entry for the page. If one does not already exist (which is likely for this page), one will be allocated and returned.

L.5.2 Locating Swapped Pages

L.5.2.1 Function: `shmem_alloc_entry()` (*mm/shmem.c*)

This function is a top-level function that returns the swap entry corresponding to a particular page index within a `%i`. If the swap entry does not exist, one will be allocated.

```

183 static inline swp_entry_t * shmem_alloc_entry (
                                struct shmem_inode_info *info,
                                unsigned long index)
184 {
185     unsigned long page = 0;
186     swp_entry_t * res;
187
188     if (index >= SHMEM_MAX_INDEX)
189         return ERR_PTR(-EFBIG);
190
191     if (info->next_index <= index)
192         info->next_index = index + 1;
193
194     while ((res = shmem_swp_entry(info, index, page)) ==
            ERR_PTR(-ENOMEM)) {
195         page = get_zeroed_page(GFP_USER);
196         if (!page)
197             break;
198     }
199     return res;
200 }
```

188-189 `SHMEM_MAX_INDEX` is calculated at compile time, and it indicates the largest possible virtual `%i` in pages. If the `var` is greater than the maximum possible sized `%i`, this returns `-EFBIG`.

191-192 `next_index` records the index of the page at the end of the `%i` inode `→i.size` alone is insufficient because the `next_index` `%i` is needed for `%i` truncation.

194-198 Calls `shmem_swp_entry()` to locate the `swp_entry_t` for the requested index. While searching, `shmem_swp_entry()` may need a number of pages. If it does, it returns `-ENOMEM`, which indicates that `get_zeroed_page()` should be called before trying again.

199 Returns the `swp_entry_t`.

L.5.2.2 Function: `shmem_swp_entry()` (*mm/shmem.c*)

This function uses information within the inode to locate the `swp_entry_t` for a given index. The inode itself is able to store `SHMEM_NR_DIRECT` swap vectors. After that, indirect blocks are used.

- 132-133 If the `index` is below `SHMEM_NR_DIRECT`, then the swap vector is contained within the direct block, so this returns it.
- 134-140 If a page does not exist at this indirect block, this installs the page that was passed in with the `page` parameter and returns `NULL`. This tells the caller to allocate a new page and calls the function again.
- 142 Treats the indirect blocks as starting from index 0.
- 143 `ENTRIES_PER_PAGE` is the number of swap vectors contained within each page in the indirect block. `offset` is now the index of the desired swap vector within the indirect block page when it is found.
- 144 `index` is now the directory number within the indirect block list that must be found.
- 145 Gets a pointer to the ~~8th~~ indirect block we are interested in.
- 147-159 If the required directory (`index`) is greater than `ENTRIES_PER_PAGE/2`, then it is a triple-indirect block, so the next block must be traversed.
- 148 Pointers to the next set of directory blocks are in the second half of the current block, so this calculates `index` as an offset within the second half of the current block.
- 149 Calculates `dir` as a pointer to the next directory block.
- 150 `index` is now a pointer within `dir` to a page containing the swap vectors we are interested in.
- 151-156 If `dir` has not been allocated, this installs the page supplied with the `page` parameter and returns `NULL` so that the caller will allocate a new page and call the function again.
- 158 `dir` is now the base of the page of swap vectors containing the one we are interested in.
- 161 Moves `dir` forward to the entry we want.
- 162-167 If an entry does not exist, this installs the page supplied as a parameter if available. If not, it returns `NULL` so that one will be allocated, and the function will be called again.
- 168 Returns the found swap vector.


```

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

```

This block is responsible for allocating a swap slot from the backing storage and a `swp_entry_t` within the inode.

541-543 Locates a free swap slot with `get_swap_page()` (See Section K.1.1). If fails, it calls `fail_writepage()`.

545 Locks the inode information.

547 Gets a free `swp_entry_t` from the %s system-specific private inode information with `shmem_swp_entry()`.

```

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 }

```

This block moves from the pagecache to the swap cache and updates statistics.


```

512     }
513 }
514 spin_unlock(&shmem_iloc);
515 return found;
516 }

```

503 Acquires the `shmem_iloc` spinlock, protecting the inode list.

504 Cycles through each entry in the `shmem_inodes` list searching for the inode holding the requested entry and page.

509 Moves the inode to the top of the list. In the event that we are reclaiming many pages, the next search will find the inode of interest at the top of the list.

510 Indicates that the page was found.

511 This page and entry have been found to break out of the loop.

514 Releases the `shmem_iloc` spinlock.

515 Returns if the page was found or not by `shmem_unuse_inode()`.

L.6.3 Function: `shmem_unuse_inode()` (*mm/shmem.c*)

This function searches the inode information in `info` to determine if the entry and page belong to it. If they do, the entry will be cleared, and the page will be removed from the swap cache and moved to the pagecache instead.

```

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

```


448-449 If `offset` is beyond the end of the direct block, this sets it to the end of the direct block for the moment.

450 Uses `shmem_find_swap()` (See Section L.6.4) to search the direct block for the entry.

451-452 If the entry was in the direct block, `goto found`. Otherwise, we have to search the indirect blocks.

454-465 Searches each of the indirect blocks for the entry.

456 `shmem_swap_entry()` (See Section L.5.2.2) returns the swap vector at the current `idx` within the inode. As `idx` is incremented in `ENTRIES_PER_PAGE`-sized strides, this will return the beginning of the next indirect block being searched.

457-458 If an error occurred, the indirect block does not exist, so it continues, which probably will exit the loop.

459 Calculates how many pages are left in the end of the ~~%o~~ to see if we only have to search a partially ~~%o~~ indirect block.

460-461 If `offset` is greater than the size of an indirect block, this sets `offset` to `ENTRIES_PER_PAGE`, so this full indirect block will be searched by `shmem_find_swap()`.

462 Searches the entire of the current indirect block for entry with `shmem_find_swap()` (See Section L.6.4).

463-467 If the entry was found, `goto found`. Otherwise, the next indirect block will be searched. If the entry is never found, the `info` struct will be unlocked, and 0 will be returned, indicating that this inode did not contain the entry and page.

468 The entry was found, so perform the necessary tasks to free it with `swap_free()`.

470 Moves `idx` to the location of the swap vector within the block.

471-472 Gets the inode and mapping.

473 Deletes the page from the swap cache.

476-477 Checks if the inode is currently being deleted or truncated by examining `inode->i_state`. If it is, this sets `limit` to the index of the last page in the adjusted ~~%o~~ size.

479-482 If the page is not being truncated or deleted, this adds it to the pagecache with `add_to_page_cache_unique()`. If successful, this clears the swap entry and decrement `info->swapped`.

483-484 If not, this adds the page back to the swap cache where it will be reclaimed later.

L.7 Setting Up Shared Regions

Contents

L.7 Setting up Shared Regions	686
L.7.1 Function: <code>shmem_zero_setup()</code>	686
L.7.2 Function: <code>shmem_file_setup()</code>	686

L.7.1 Function: `shmem_zero_setup()` (*mm/shmem.c*)

This function is called to set up a VMA that is a shared region backed by anonymous pages. The call graph that shows this function is in Figure 12.5. This occurs when `mmap()` creates an anonymous region with the `MAP_SHARED` flag.

```

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 Calculates the size.

1669 Calls `shmem_file_setup()` (See Section L.7.2) to create a file called `dev/zero` and of the calculated size. We will see in the functions code commentary why the name does not have to be unique.

1673-1674 If a file already exists for this virtual area, this calls `fput()` to drop its reference.

1675 Records the new file pointer.

1676 Sets the `vm_ops` so that `shmem_nopage()` (See Section L.5.1.1) will be called when a page needs to be faulted in for this VMA.

L.7.2 Function: `shmem_file_setup()` (*mm/shmem.c*)

This function is called to create a new file in `shmfs`, the internal shared memory filesystem. Because the filesystem is internal, the supplied `name` does not have to be unique within each directory. Hence, every file that is created by an anonymous region with `shmem_zero_setup()` will be called `dev/zero`, and regions created with `shmget()` will be called `/SYSVNNI`, where `NN` is the key that is passed as the `key` argument to `shmget()`.


```

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 is assigned to be the dnode representing the root of shmfs.

1629 Allocates a new dentry with `d_alloc()`.

1630-1631 Returns `-ENOMEM` if one could not be allocated.

1634 Gets an empty struct `file` from the `file` table. If one could not be found, `-ENFILE` will be returned, indicating a `file` table overflow.

1639-1641 Creates a new inode, which is a regular `%o` (`S_IFREG`) and globally readable, writable and executable. If it fails, it returns `-ENOSPC`, indicating no space is left in the `%s` system.

1643 `d_instantiate()` `%s` in the inode information for a dentry. It is defined in `fs/dcache.c`.

1644-1649 Fills in the remaining inode and `%o` information.

1650 Returns the newly created struct `file`.

1653 The error path when an inode could not be created. `put_filp()` `%o` free up the struct `file` entry in the `file` table.

1655 `dput()` will drop the reference to the dentry, which destroys it.

1656 Returns the error code.

- 235-236 If `IPC_PRIVATE` is specified, most of the flags are ignored, and the region is created with `newseg()`. This flag is intended to provide exclusive access to a shared region, but Linux does not guarantee exclusive access.
- 237 If not, this searches to see if the key already exists with `ipc_findkey()`.
- 238-239 If it does not and `IPC_CREAT` was not specified, then this returns `-ENOENT`.
- 241 If not, this creates a new region with `newseg()`.
- 242-243 If the region already exists and the process requested a new region that did not previously exist to be created, this returns `-EEXIST`.
- 244-255 If not, we are accessing an existing region, so it locks it, makes sure we have the required permissions, builds a segment identifier with `shm_buildid()` and unlocks the region again. The segment identifier will be returned back to userspace.
- 256 Releases the semaphore protecting IDs.
- 257 Returns either the error or the segment identifier.

L.8.1.2 Function: `newseg()` (*ipc/shm.c*)

This function creates a new shared segment.

```

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);

```

This block allocates the segment descriptor.

- 182 Calculates the number of pages the region will occupy.

206-213 Fills in `%s` in the segment descriptor.

214 Builds a segment identifier, which is what is returned to the caller of `shmget()`.

215-217 Sets the `%s` pointers and `%s` operations structure.

218 Updates `shm_tot` to the total number of pages used by shared segments.

220 Returns the identifier.

L.8.2 Attaching a SYSV Shared Region

L.8.2.1 Function: `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;
```


614-615 Makes sure the descriptor exists.

616-620 Makes sure the ID matches the descriptor.

621-624 Makes sure the caller has the correct permissions.

625 Gets a pointer to the struct file, which do_mmap() requires.

626 Gets the size of the shared region, so do_mmap() knows what size of VMA to create.

627 Temporarily increments shm_nattach(), which normally indicates how many VMAs are using the segment. This is to prevent the segment being freed prematurely. The real counter will be incremented by shm_open(), which is the open() callback used by the vm_operations_struct used for shared regions.

628 Releases the descriptor.

```

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);

```

This block is where do_mmap() will be called to attach the region to the calling process.

630 Acquires the semaphore protecting the mm_struct.

632-634 If an address was specified, calls find_vma_intersection() (See Section D.3.1.3) to ensure no VMA overlaps the region we are trying to use.

639-641 Makes sure there is at least a four-page gap between the end of the shared region and the stack.

644 Calls do_mmap() (See Section D.2.1.1), which will allocate the VMA and map it into the process address space.

M.1 Determining Available Memory

Contents

M.1 Determining Available Memory	698
M.1.1 Function: <code>vm_enough_memory()</code>	698

M.1.1 Function: `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)
59         return 1;
60
61     /* The page cache contains buffer pages these days.. */
62     free = atomic_read(&page_cache_size);
63     free += nr_free_pages();
64     free += nr_swap_pages;
65
66     /*
67      * This double-counts: the nrpages are both in the page-cache
68      * and in the swapper space. At the same time, this compensates
69      * for the swap-space over-allocation (ie "nr_swap_pages" being
70      * too small.
71      */
72     free += swapper_space.nrpages;
73
74     /*
75      * The code below doesn't account for free space in the inode
76      * and dentry slab cache, slab cache fragmentation, inodes and
77      * dentries which will become freeable under VM load, etc.
78      * Lets just hope all these (complex) factors balance out...
79      */
80     free += (dentry_stat.nr_unused * sizeof(struct dentry)) >> PAGE_SHIFT;
81     free += (inodes_stat.nr_unused * sizeof(struct inode)) >> PAGE_SHIFT;
82
83     return free > pages;
84 }

```

68-69 If the system administrator has specified through the `proc` interface that overcommit is allowed, this returns immediately saying that the memory is available.

72 Starts the free pages count with the size of the pagecache because these pages may be easily reclaimed.

M.2 Detecting and Recovering From OOM

Contents

M.2 Detecting and Recovering From OOM	700
M.2.1 Function: out_of_memory()	700
M.2.2 Function: oom_kill()	701
M.2.3 Function: select_bad_process()	702
M.2.4 Function: badness()	703
M.2.5 Function: oom_kill_task()	704

M.2.1 Function: 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
207     if (nr_swap_pages > 0)
208         return;
209
210     now = jiffies;
211     since = now - last;
212     last = now;
213
214     last = now;
215     if (since > 5*HZ)
216         goto reset;
217
218     since = now - first;
219     if (since < HZ)
220         return;
221
222     if (++count < 10)
223         return;
224
225     since = now - lastkill;
226     if (since < HZ*5)
227         return;
228
229     lastkill = now;
230     oom_kill();
231
232 reset:
233     first = now;
234     count = 0;
235 }

```



```

194         */
195         yield();
196         return;
197 }

```

176 Acquires the read-only semaphore to the task list.

177 Calls `select_bad_process()` (See Section M.2.3) to find a suitable process to kill.

180-181 If one could not be found, this panics the system because otherwise the system will deadlock. In this case, it is better to deadlock and have a developer solve the bug than have a mysterious hang.

184-187 Cycles through the task list and calls `oom_kill_task()` (See Section M.2.5) for the selected process and all its threads. Remember that threads will all share the same `mm_struct`.

188 Releases the semaphore.

195 Calls `yield()` to allow the signals to be delivered and the processes to die. The comments indicate that `kswapd` will be the sleeper, but it is possible that a process in the direct-reclaim path will be executing this function, too.

M.2.3 Function: `select_bad_process()` (*mm/oom_kill.c*)

This function is responsible for cycling through the entire task list and returning the process that scored highest with the `badness()` function.

```

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 Cycles through all tasks in the task list.

128 If the process is the system idle task, this skips over it.

- 65-66 If the process has already been marked by the OOM killer as exiting, this returns 0 because there is no point trying to kill it multiple times.
- 71 The total VM used by the process is the base starting point.
- 79-80 `cpu_time` is calculated as the total runtime of the process in seconds. `run_time` is the total runtime of the process in minutes. Comments indicate that there is no basis for this other than it works well in practice.
- 82 Divides the points by the integer square root of `cpu_time`.
- 83 Divides the points by the cube root of `run_time`.
- 89-90 If the process has been niced to be of lower priority, double its points because it is likely to be an unimportant process.
- 96-98 On the other hand, if the process has superuser privileges or has the `CAP_SYS_ADMIN` capability, it is likely to be a system process, so it divides the points by 4.
- 106-107 If the process has direct access to hardware, then this divides the process by 4. Killing these processes forcibly could potentially leave hardware in an inconsistent state. For example, forcibly killing X is never a good idea.
- 112 Returns the score.

M.2.5 Function: `oom_kill_task()` (*mm/oom_kill.c*)

This function is responsible for sending the appropriate kill signals to the selected task.

```

144 void oom_kill_task(struct task_struct *p)
145 {
146     printk(KERN_ERR "Out of Memory: Killed process %d (%s).\n",
147                p->pid, p->comm);
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 }
```

References

- [BA01] Je 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 157–164, 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] Je 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 30-31, 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.
- [MM87] Rodney R. Oldehoeft, Maekawa Mamoru and Arthur E. Oldehoeft. *Operating Systems, Advanced Concepts*. Benjamin/Cummings Publishing, 1987.
- [MM01] Richard McDougall and Jim Mauro. *Solaris Internals*. Menlo Park, CA: Sun Microsystems Press, 2001.
- [PN77] James L. Peterson and Theodore A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421-431, 1977.
- [Ray02] Eric S. Raymond. *The Cathedral and the Bazaar (Revised Edition)*. O'Reilly, 2002.
- [RC01] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers, 2nd Edition*. Sebastopol, CA: O'Reilly, 2001.
- [RM01] Eric S. Raymond and Rick Moen. *How to Ask Questions the Smart Way*.
www.catb.org/esr/faqs/smart-questions.html, 2001.
- [Sea00] Chris B. Sears. The elements of cache programming style. In *Proceedings of the 4th Annual Showcase and Conference*, pages 283-298, Berkeley, CA, October 2000. The USENIX Association.
- [Tan01] Andrew S. Tanenbaum. *Modern Operating Systems, 2nd Edition*. Upper Saddle River, NJ: Prentice-Hall, 2001.
- [Vah96] Uresh Vahalia. *UNIX Internals*. Upper Saddle River, NJ: Prentice-Hall, 1996.
- [WJNB95] P. R. Wilson, M. S. Johnstone, M. Neely and D. Boles. Dynamic storage allocation: A survey and critical review. *Lecture Notes in Computer Science*, 986, 1995.

Code Commentary Index

activate_lock(), 550
activate_page_nolock(), 550
add_page_to_active_list(), 547
add_page_to_hash_queue(), 540
add_page_to_inactive_list(), 548
add_page_to_inode_queue(), 539
add_to_page_cache_unique(), 538
add_to_page_cache(), 537
__add_to_page_cache(), 539
add_to_swap_cache(), 590
allocate_mm(), 257
alloc_area_pmd(), 447
alloc_area_pte(), 448
__alloc_bootmem_core(), 403
__alloc_bootmem_node(), 402
alloc_bootmem_node(), 402
__alloc_bootmem(), 401
alloc_bootmem(), 401
alloc_bounce_bh(), 526
alloc_bounce_page(), 527
alloc_one_pte(), 295
_alloc_pages(), 420
alloc_pages(), 420
__alloc_pages(), 421
alloc_page(), 433
arch_get_unmapped_area(), 264, 313

badness(), 703
balance_classzone(), 428
block_flushpage(), 381
block_read_full_page(), 363
bounce_end_io_read(), 528
bounce_end_io_write(), 528
bounce_end_io(), 530

break_cow(), 354
BREAK_GFP_ORDER_HI, 502
BREAK_GFP_ORDER_LO, 502
brw_page(), 603
build_zonelists(), 232

CACHE_NAMELEN, 502
calc_vm_flags(), 264
can_share_swap_page(), 350
can_vma_merge(), 277
CHECK_PAGE(), 496
clear_page_tables(), 306
clear_user_highpage(), 353
CLUSTER_OFFSET(), 389, 391
CLUSTER_PAGES, 389, 391
contig_page_data, 420
copy_from_high_bh(), 529
copy_mm(), 254
copy_one_pte(), 295
copy_to_high_bh_irq(), 529
copy_user_highpage(), 353
create_bounce(), 524

DECLARE_FSTYPE(), 636
DEFAULT_MAX_MAP_COUNT, 264
del_page_from_active_list(), 549
del_page_from_inactive_list(),
549
do_anonymous_page(), 344
do_ccupdate_local(), 509
do_flushpage(), 381
do_generic_file_read(), 358
do_mlockall(), 318
do_mlock(), 319
do_mmap_pgoff(), 262

do_mmap(), 261
 do_mremap(), 278
 do_munmap(), 296
 do_no_page(), 341
 do_page_fault(), 328
 do_shmem_file_read(), 647
 do_swap_page(), 346
 do_wp_page(), 352
 drain_cpu_caches(), 509

 enable_all_cpucaches(), 504
 enable_cpucache(), 505
 _end, 221
 end_buffer_io_async(), 604
 __end_of_fixed_addresses, 245
 exclusive_swap_page(), 351
 exit_mmap(), 258, 304
 expand_stack(), 330, 337
 expand(), 425, 427

 fail_writepage(), 679
 filemap_nopage(), 382
 file_read_actor(), 649
 __find_get_page(), 543
 find_get_page(), 543
 __find_lock_page_helper(), 545
 find_lock_page(), 544
 __find_lock_page(), 545
 find_max_low_pfn(), 221
 find_max_pfn(), 221
 __find_page_nolock(), 544
 find_vma_intersection(), 312
 find_vma_prepare(), 271
 find_vma_prev(), 310
 find_vma(), 309
 FIXADDR_TOP, 244
 fixrange_init(), 245
 __fix_to_virt(), 244
 flush_all_zero_pkmappings(), 518
 follow_page(), 248
 free_all_bootmem_core(), 415
 free_all_bootmem(), 415
 free_area_init_core(), 226
 free_area_init_node(), 224
 free_area_init(), 224
 free_area_pmd(), 454

 free_area_pte(), 455
 free_block(), 499
 __free_block(), 500
 free_bootmem_core(), 409
 free_bootmem(), 409
 free_one_pgd(), 306
 free_one_pmd(), 307
 free_pages_init(), 413
 __free_pages_ok(), 435
 __free_pages(), 435
 free_pages(), 435, 440
 free_page(), 440
 __free_page(), 440
 free_swap_and_cache(), 601

 generic_file_mmap(), 369
 generic_file_readahead(), 365
 generic_file_read(), 355
 generic_file_vm_ops, 370
 __get_dma_pages(), 434
 __get_free_pages(), 433
 __get_free_page(), 433
 get_max_readahead(), 389
 get_one_pte(), 294
 get_swaphandle_info(), 605
 get_swap_page(), 585
 get_unmapped_area(), 312
 get_user_pages(), 289
 get_vm_area(), 443
 get_zeroed_page(), 434
 gfp_mask, 420

 handle_mm_fault(), 339
 handle_pte_fault(), 340
 highstart_pfn, 221

 __init, 412
 init_bootmem_core(), 397
 init_bootmem_node(), 396
 init_bootmem(), 396
 init_emergency_pool(), 532
 init_mm, 254
 INIT_MM(), 254
 init_tmpfs(), 635
 insert_vm_struct(), 270
 __insert_vm_struct(), 270

ipc_findkey(), 690

_KERNPG_TABLE, 243

kfree(), 503

kmalloc(), 502

kmap_atomic(), 519

kmap_get_fixmap_pte(), 247

kmap_high(), 515

kmap_init(), 240, 246

kmap_nonblock(), 514

kmap_prot, 247

kmap_pte, 247

kmap_vstart, 247

__kmap(), 514

kmap(), 514

__kmem_cache_alloc (SMP Case)(), 489

__kmem_cache_alloc (UP Case)(), 488

kmem_cache_alloc_batch(), 494

kmem_cache_alloc_head(), 491

kmem_cache_alloc_one_tail(), 492

kmem_cache_alloc_one(), 492

kmem_cache_alloc(), 488

kmem_cache_create(), 459

kmem_cache_destroy(), 472

kmem_cache_estimate(), 467

__kmem_cache_free (SMP Case)(), 496

__kmem_cache_free (UP Case)(), 496

kmem_cache_free_one(), 497

kmem_cache_free(), 495

kmem_cache_grow(), 481

kmem_cache_init_objs(), 486

kmem_cache_init(), 511

kmem_cache_reap(), 473

__kmem_cache_shrink_locked(), 471

kmem_cache_shrink(), 469

__kmem_cache_shrink(), 470

kmem_cache_sizes_init(), 501

kmem_cache_slabmgmt(), 479

kmem_find_general_cachep(), 480

kmem_freepages(), 512

kmem_getpages(), 512

kmem_slab_destroy(), 485

kmem_tune_cpucache(), 506

km_type, 519

kswapd_balance_pgdat(), 580

kswapd_balance(), 580

kswapd_can_sleep_pgdat(), 579

kswapd_can_sleep(), 579

kswapd_init(), 577

kswapd(), 577

kunmap_atomic(), 523

kunmap_high(), 521

kunmap(), 521

__lock_page(), 234

lock_page(), 234, 612

lookup_swap_cache(), 596

lru_cache_add(), 547

lru_cache_del(), 548

__lru_cache_del(), 549

make_pages_present(), 288

MAP_ALIGN(), 227

map_new_virtual(), 515

mark_page_accessed(), 550

max_low_pfn, 221

max_map_count, 264

max_pfn, 221

memclear_highpage_flush(), 382

mem_init(), 411

mlock_fixup_all(), 323

mlock_fixup_end(), 325

mlock_fixup_middle(), 326

mlock_fixup_start(), 324

mlock_fixup(), 322

mmdrop(), 258, 259

__mmdrop(), 260

mmput(), 258

mm_alloc(), 257, 258

mm_init(), 257

move_one_page(), 293

move_page_tables(), 292

move_vma(), 284

newseg(), 690

nopage_sequential_readahead(), 388

one_highpage_init(), 414

oom_kill_task(), 704
 oom_kill(), 701
 out_of_memory(), 700

 pagetable_init(), 240, 241
 page_cache_get(), 542
 page_cache_read(), 387
 page_cache_release(), 543
 page_cache_size, 540
 page_cluster, 618
 page_hash_table, 540
 page_hash(), 388, 538
 page_waitqueue(), 235, 237
 paging_init(), 240
 PFN_UP(), 221
 ppro_with_ram_bug(), 414
 precheck_file_write(), 651
 pswpin, 603
 pswpout, 603
 ptep_get_and_clear(), 456
 put_page_testzero(), 435

 readpage(), 363
 read_cluster_nonblocking(), 390
 read_swap_cache_async(), 597
 refill_inactive(), 552
 register_bootmem_low_pages(), 222
 remove_exclusive_swap_page(), 599
 _remove_inode_page(), 541
 remove_inode_page(), 541, 681
 remove_page_from_hash_queue(),
 542
 remove_page_from_inode_queue(),
 541
 reserve_bootmem_core(), 399
 reserve_bootmem_node(), 399
 reserve_bootmem(), 399
 rmqueue(), 425
 run_task_queue(), 390
 rw_swap_page_base(), 603
 rw_swap_page(), 602

 scan_swap_map(), 587
 search_exception_table(), 334
 select_bad_process(), 702
 setup_memory(), 220

 SET_PAGE_CACHE(), 483
 SET_PAGE_SLAB(), 483
 SGP_WRITE, 653
 shmem_alloc_entry(), 676
 shmem_create(), 641
 shmem_empty(), 666
 shmem_file_read(), 646
 shmem_file_setup(), 686
 shmem_file_write(), 651
 shmem_find_swp(), 685
 shmem_follow_link_inline(), 656
 shmem_free_swp(), 663
 shmem_getpage(), 669
 shmem_get_inode(), 642
 shmem_ilock, 655
 SHMEM_I(), 660, 670
 shmem_link(), 664
 SHMEM_MAX_INDEX, 676
 shmem_mkdir(), 665
 shmem_mknod(), 641
 shmem_mmap(), 645
 shmem_nopage(), 668
 SHMEM_NR_DIRECT, 676
 shmem_positive(), 667
 shmem_readlink_inline(), 656
 shmem_readlink(), 656
 shmem_read_super(), 637
 shmem_rmdir(), 666
 SHMEM_SB(), 638
 shmem_set_size(), 639
 shmem_swp_entry(), 674, 676
 shmem_symlink(), 654
 shmem_sync_file(), 658
 shmem_truncate_direct(), 662
 shmem_truncate_indirect(), 660
 shmem_truncate(), 659
 shmem_unlink(), 665
 shmem_unuse_inode(), 682
 shmem_unuse(), 681
 shmem_writepage(), 679
 shmem_zero_setup(), 686
 SHMLBA, 693
 shm_lock(), 693
 SHM_RND, 693
 shm_tot, 692
 shrink_caches(), 562

shrink_cache(), 554
slab_break_gfp_order, 502
smp_call_function_all_cpus(), 508
start_pfn, 221
STATS_INC_GROWN(), 484
swapon_readahead(), 391
swapper_pg_dir, 240
swap_duplicate(), 591
swap_entry_free(), 593
swap_free(), 592
swap_info_get(), 594
swap_info_put(), 595
swap_out_mm(), 568
swap_out_pgd(), 570
swap_out_pmd(), 571
swap_out_vma(), 569
swap_out(), 566
swap_setup(), 617
swap_writepage(), 599
SWP_ENTRY(), 586
sync_page(), 235
sys_mlockall(), 316
sys_mlock(), 315
sys_mremap(), 278
sys_munlockall(), 321
sys_munlock(), 321
sys_shmat(), 692
sys_shmget(), 689
sys_swapoff(), 619
sys_swapon(), 607

tlb_finish_mmu(), 374
tlb_gather_mmu(), 374
tlb_remove_page(), 374, 376
totalhigh_pages, 415
truncate_complete_page(), 380
truncate_inode_pages(), 377
truncate_list_pages(), 378
truncate_partial_page(), 381
truncate(), 370
TryLockPage(), 234, 546

try_to_free_pages(), 563
try_to_free_pages_zone(), 564
try_to_swap_out(), 573
try_to_unuse(), 623

unlock_page(), 236
unmap_fixup(), 300
unuse_pgd(), 629
unuse_pmd(), 630
unuse_process(), 627
unuse_pte(), 631
unuse_vma(), 628

valid_swaphandles(), 392
vfree(), 452
vmalloc_area_pages(), 445
__vmalloc_area_pages(), 446
__vmalloc(), 442
vmalloc(), 442
vmap(), 450
__vma_link_file(), 274
__vma_link_list(), 273
__vma_link_rb(), 274
vma_link(), 272
__vma_link(), 273
vma_merge(), 275
vmfree_area_pages(), 453
vmtruncate_list(), 372
vmtruncate(), 370, 659
vm_enough_memory(), 698
vm_reend, 389
VM_SEQ_READ, 384
vm_swap_full(), 602

wait_on_page(), 236
___wait_on_page(), 237

zap_page_range(), 373
zap_pmd_range(), 375
zap_pte_range(), 376
zone_sizes_init(), 223

Index

3GiB/1GiB Split, 55, 153

896MiB limit of ZONE_NORMAL, 55

Accessing userspace, 87

access_ok(), 89

activate_page(), 171

active_list, 26, 164, 169

Address mapping to pages, 42

Address space management, 55

Address space regions, 61

Address space, 53

address_space, 25, 56, 64

add_to_page_cache_unique(), 166

add_to_page_cache(), 166

add_to_swap_cache(), 186, 188

allocate_mm(), 60

__alloc_bootmem_core(), 100

alloc_bootmem_low_pages_node(), 97, 100

alloc_bootmem_low_pages(), 96, 99

alloc_bootmem_low(), 96, 99

__alloc_bootmem_node(), 100

alloc_bootmem_node(), 97, 100

alloc_bootmem_pages_node(), 97, 100

alloc_bootmem_pages(), 96, 99

alloc_bootmem(), 96, 99

__alloc_bootmem(), 99

alloc_bounce_bh(), 158

alloc_bounce_page(), 158

alloc_pages(), 108

alloc_page(), 108

Anonymous pages, 84

Anonymous pages, backing, 195

arch_get_unmapped_area(), 69

arch_set_page_uptodate(), 26

Associative mapping, 44

AS_EIO, 90

AS_ENOSPC, 90

autoconf, 2

automake, 2

Backing storage, 179

Benchmarking kernels, 11

Big Kernel Lock (BKL), 192

Binary Buddy Allocator, 105

BitKeeper, 5

bmap(), 192

Boot allocator initialization, 98

Boot map representation, 96

Boot memory allocator, 95

bootmem_bootmap_pages(), 96

bootmem_data, 96

Bootstrap finalization, 101

Bounce buffers, 153, 157

bounce_end_io_write(), 157

bounce_end_io(), 159

Browsing code, 9

brw_page(), 192

Buddies, 105

Buddy coalescing, 105, 109

buffer_head, 157

BUG(), 100

Cache chain, 123

Cache coloring, 124

Cache creation (slab), 133

Caches (slab allocator), 125

cache_cache, 150

- cache_sizes, 146
- cache_sizes_t, 146
- Call graph generation, 11
- ccupdate_t, 149
- cc_data(), 148
- cc_entry(), 149
- CFGS_OFF_SLAB, 130, 138
- CFLGS_OPTIMIZE, 130
- check_pgt_cache(), 40
- ClearPageActive(), 28
- ClearPageDirty(), 28
- ClearPageError(), 28
- ClearPageLaunder(), 28
- ClearPageReferenced(), 28
- ClearPageReserved(), 28
- ClearPageUptodate(), 28
- clear_user_highpage(), 85
- clear_user_page(), 89
- clock_searchp, 135
- CLUSTER, 185
- Code comprehension, 11
- CodeViz, 10
- Coloring, 124
- committed_space, 211
- CONFIG_SLAB_DEBUG, 125, 129
- ConTest, 11
- contig_page_data, 15
- Copy On Write (COW), 87, 331
- Copying to/from userspace, 87
- copy_from_high_bh(), 159
- copy_from_user(), 89
- copy_mm(), 60, 61
- copy_to_user(), 89
- copy_user_page(), 89
- CPU cache hit, 45
- CPU cache line, 44
- CPU cache management, 44
- CPU cache miss, 45
- cpucache, 148
- cpu_vm_mask, 60
- create_bounce(), 158
- CREATE_MASK, 131
- Creating ~~86~~ in tmpfs, 199
- Creating VMAs, 67
- ctags, 9
- dbench, 12
- Deactivating swap area, 193
- def_flags, 60
- DEF_PRIORITY, 173
- Deleting a VMA, 79
- Demand allocation, 82, 84
- Demand fetch, 80
- Demand paging, 82, 85
- Device backed regions, 64
- DFLGS_GROWN, 131
- di , 4, 5
- di func, 9
- di struct, 9
- Direct mapping, 44
- Direct-reclaim, 19, 20
- do_anonymous_page(), 84
- do_ccupdate_local(), 150
- do_mmap2(), 67
- do_mmap_pgoff(), 67
- do_no_page(), 82, 84
- do_page_fault(), 80
- do_swap_page(), 34, 82, 85
- do_wp_page(), 83, 87
- dummy_security_ops, 212
- Eliminating fragmentation (internal), 123
- empty_zero_page, 85
- enable_all_cpucaches(), 149
- enable_cpucache(), 149
- _end, 22
- Exception handling, 79
- Exception table, 334
- exception_table_entry, 79
- exit_mmap(), 60, 61
- Expanding the stack, 82
- EXPORT_SYMBOL(), 29
- External fragmentation, 112
- _ex_table, 79
- File backed regions, 64
- File/device-backed pages, 85
- filemap_nopage(), 64, 85
- Filesystem, shared memory, 195
- file_operations, 198

- Finalizing memory bootstrapping, 101
- find_max_low_pfn(), 22
- find_max_pfn(), 22
- __find_page_nolock(), 169
- find_vma_intersection(), 69, 70
- find_vma_prepare(), 70, 71
- find_vma_prev(), 69, 70
- find_vma(), 69, 70
- First Fit, 95
- __FIXADDR_SIZE, 55
- FIXADDR_START, 55, 153
- FIXADDR_TOP, 55
- fixrange_init(), 41
- FIX_KMAP_BEGIN, 41, 156
- FIX_KMAP_END, 41, 156
- flush_cache_all(), 46
- flush_cache_mm(), 46
- flush_cache_page(), 46
- flush_cache_range(), 46
- flush_dcache_page(), 47
- flush_icache_page(), 47
- flush_icache_range(), 47
- flush_icache_user_range(), 47
- flush_page_to_ram(), 47, 85
- flush_tlb_all(), 44
- flush_tlb_mm(), 44
- flush_tlb_page(), 45
- flush_tlb_pgtables(), 45
- flush_tlb_range(), 44
- for_each_pgdat(), 18
- Fragmentation elimination (external), 117
- Fragmentation, 112
- Free lists, 105
- free_all_bootmem_core(), 101
- free_all_bootmem_node(), 97, 101
- free_all_bootmem(), 96, 101
- free_area_init_node(), 23, 24
- free_area_init(), 24
- free_area_t, 105
- free_bootmem_node(), 97, 100
- free_bootmem(), 96, 100
- free_initmem(), 102
- free_mm(), 60
- free_pages_init(), 101
- __free_pages(), 109
- __free_page(), 109
- free_page(), 109
- free_pgtables(), 79
- free_swap_and_cache(), 203
- Frequently Asked Questions (FAQ), 3
- generic_file_vm_ops, 64
- Get Free Page (GFP) ~~Yes~~, 110
- __get_dma_pages(), 108
- __get_free_pages(), 108
- get_free_page(), 108
- __get_free_page(), 108
- GET_PAGE_CACHE(), 138
- GET_PAGE_SLAB(), 138
- get_pgdat_fast(), 39
- get_pgdat_slow(), 40
- get_swaphandle_info(), 192
- get_swap_page(), 184, 186
- get_unmapped_area(), 69, 70
- get_user(), 89
- get_vm_area(), 118
- GFP ~~Yes~~, 110
- GFP_ATOMIC, 111, 112
- GFP_DMA, 110
- __GFP_DMA, 110
- __GFP_FS, 111
- __GFP_HIGH, 111
- __GFP_HIGHIO, 111
- __GFP_HIGHMEM, 110
- GFP_HIGHUSER, 111, 112
- __GFP_IO, 111
- GFP_KERNEL, 111, 112
- GFP_KSWAPD, 111, 112
- gfp_mask, 106
- GFP_NFS, 111, 112
- __GFP_NOFAIL, 115
- GFP_NOFS, 111, 112
- GFP_NOHIGHIO, 111, 112
- GFP_NOIO, 111, 112
- __GFP_NORETRY, 115
- __GFP_REPEAT, 115
- GFP_USER, 111, 112
- __GFP_WAIT, 111
- Global zero page, 53
- Golden ratio, 23

- GOLDEN_RATIO_PRIME, 23
- GraphViz, 10
- g_cpucache_up, 149
- handle_mm_fault(), 80
- handle_pte_fault(), 82
- High memory atomic mappings, 156
- High memory I/O, 157
- High memory mapping, 154
- High memory, 29, 153
- highend_pfn, 98
- highstart_pfn, 98
- Huge TLB Filesystem (hugetlbfs), 51
- inactive_list, 26, 164, 169
- Inclusion property, 165
- __init, 102
- Initializing buddy allocator, 102
- Initializing kmem_bufctl_t, 142
- Initializing mm_struct, 60
- Initializing objects, 144
- Initializing page hash table, 167
- Initializing page tables, 40
- Initializing shared regions, 204
- Initializing swap areas, 192
- Initializing virtual filesystem, 196
- Initializing boot memory allocator, 98
- Initializing slab allocator, 150
- __init_begin, 102
- init_bootmem_core(), 17
- init_bootmem_node(), 97
- init_bootmem(), 96
- init_emergency_pool(), 159
- __init_end, 102
- init_mm, 60
- INIT_MM(), 60
- Inode queue, 167
- inode_operations, 198
- Inserting a memory region, 70
- insert_vm_struct(), 70
- Internal fragmentation, 112
- Internet Relay Chat (IRC), 3
- InterProcessor Interrupt (IPI), 60
- IPC, 204
- Kernel address space, 53
- Kernel benchmarking, 11
- Kernel configuration, 2
- Kernel documentation, 3
- Kernel image location, 40
- Kernel patching, 4
- Kernel subtrees, 5
- Kernel Trace, 4
- Kernel Trap, 4
- kern_mount(), 195
- kfree(), 126, 147
- kmalloc(), 126, 147
- kmap_atomic(), 41, 154, 156
- kmap_high(), 154
- kmap_nonblock(), 154, 155
- kmap(), 55, 154, 155
- kmem_bufctl_t, 138, 141
- kmem_bufctl_t types, 138
- kmem_cache, 150
- kmem_cache_alloc(), 126, 142
- kmem_cache_create(), 126
- kmem_cache_destroy(), 126
- kmem_cache_free(), 126
- kmem_cache_init(), 151
- kmem_cache_reap(), 126
- kmem_cache_shrink(), 126
- kmem_cache_slabmgmt(), 138
- kmem_freepages(), 151
- kmem_getpages(), 151
- kmem_tune_cpucache(), 149
- km_type, 156
- KM_TYPE_NR, 156
- kswapd, 19, 108, 163
- kswapd_balance(), 177
- kswapd_can_sleep(), 177
- kswapd_init(), 175
- kswapd_wait, 177
- kswapd(), 175
- kunmap_atomic(), 156
- kunmap_high(), 156
- kunmap(), 154, 156
- LAST_PKMAP, 154
- last_pkmap_nr, 155
- Lazy buddy, 114
- Lazy TLB, 57
- Level 1 CPU, 44
- Linear address macros, 34

- Linear address space, 53
- Linux Kernel Mailing List (LKML), 4
- Linux Kernel Newbies, 3
- Linux Weekly News* (LWN), 3
- Linux-MM Web site, 4
- lmbench, 11
- local_pages, 424
- locked_vm, 59
- Locking regions, 75
- LockPage(), 28
- lookup_swap_cache(), 186
- LRU 2Q, 164
- LRU list page reclaim, 170
- LRU lists, 163, 169
- LRU rotation, 170
- LRU, 164
- lru_cache_add(), 171
- lru_cache_del(), 171
- Major page faults, 80
- mapping_gfp_mask(), 90
- map_new_virtual(), 154
- MAP_POPULATE, 93
- mark_page_accessed(), 171
- MARK_USED(), 106
- MAX_DMA_ADDRESS, 100
- max_low_pfn, 98
- max_mapped, 170
- MAX_NR_ZONES, 29
- MAX_ORDER, 105
- max_pfn, 98
- max_scan, 170
- MAX_SWAPFILES, 180
- MAX_SWAP_BADPAGES, 183
- Memory Management Unit (MMU), 33
- Memory pools, 160
- Memory pressure, 19
- Memory regions, 61
- mem_init(), 101
- mem_map, 15
- mem_map initialization, 24
- mem_map_t, 26
- merge_segments(), 72
- Minor page faults, 80
- min_low_pfn, 98
- mkswap, 182
- mk_pte_phys(), 39
- mk_pte(), 39
- mlockall(), 75
- mlock_fixup_all(), 76
- mlock_fixup_end(), 76
- mlock_fixup_middle(), 76
- mlock_fixup_start(), 76
- mlock_fixup(), 76
- mlock(), 75
- mmap_sem, 59
- mmdrop(), 61
- mmlist, 59
- mmput(), 61
- MMU, 33
- mm_alloc(), 60
- mm_count, 58
- mm_init(), 60
- mm_struct, 55, 57
- mm_users, 58
- Move-To-Front heuristic, 165
- move_page_tables(), 75
- move_vma(), 72
- munmap(), 76
- newseg(), 204
- Node ID (NID), 17
- Node structure, 16
- Node-local allocation, 16, 107
- Nodes, 15
- Non Uniform Memory Access (NUMA), 15
- NRPTE, 48
- nr_pages, 170
- NUMA, 15
- Object allocation, 123, 144
- Object coloring, 124
- Object freeing, 144
- Object initialization, 144
- Objects, 144
- one_highpage_init(), 102
- OOM detection, 210
- OOM killing, 211
- OOM management, 209
- OOM prevention, 209

- oom_kill(), 210
- Order allocation, 105
- out_of_memory(), 210
- Page allocation, 105
- Page cache, 163, 165, 167
- Page color, 124
- Page directory describing, 33
- Page fault exception handler, 87
- Page faulting (shmem), 201
- Page faults, 80
- Page flags, 26, 27
- Page fragmentation, 112
- Page Frame Number (PFN), 17, 95
- Page Global Directory (PGD), 33
- Page hash table, 166
- Page hash, 166
- Page index, 26
- Page lists, 25
- Page Middle Directory (PMD), 33
- Page reclamation, 163
- Page replacement policy, 164
- Page Size Extension (PSE), 40
- Page structure, 24
- Page table initialization, 40
- Page table layout, 34
- Page table management, 33
- Page table protection, 36
- Page to zone mapping, 29
- Page wait queues, 22
- PageActive(), 28
- PageChecked(), 28
- PageClearSlab(), 28
- PageDirty(), 28
- PageError(), 28
- PageHighMem(), 28
- PageLaunder(), 28
- PageLocked(), 28
- PageLRU(), 28
- Pageout daemon, 175
- Pageout of process pages, 173
- PageReferenced(), 28
- PageReserved(), 28
- PageSetSlab(), 28
- PageSlab(), 28
- PageSwapCache(), 186
- pages_high, 20
- pages_low, 19
- pages_min, 20
- pagetable_init(), 41
- PageUptodate(), 28
- pagevec, 178
- _PAGE_ACCESSED, 36
- PAGE_ALIGN(), 35
- page_cache_alloc(), 166
- page_cache_get(), 166
- page_cache_init(), 167
- page_cache_read(), 166, 169
- page_cache_release(), 166
- page_cluster, 87
- _PAGE_DIRTY, 36
- _page_hashfn(), 167
- page_hash_bits, 167
- page_hash_table, 166
- PAGE_OFFSET, 55
- PAGE_PER_WAITQUEUE, 22
- _PAGE_PRESENT, 36
- _PAGE_PROTNONE, 36
- _PAGE_RW, 36
- PAGE_SHIFT, 35
- page_state, 32
- _PAGE_USER, 36
- page_waitqueue(), 23
- Paging out, 175
- Paging, 179
- paging_init(), 41
- Patch generation, 5
- Patch submission, 12
- Patch usage, 4
- patch, 5
- PatchSet, 7
- __pa(), 42
- Per-CPU cache, 148
- Persistent Kernel Map (PKMap), 153
- per_cpu_pages, 32
- per_cpu_pageset, 31
- PFN, 17
- PF_FREE_PAGES, 113
- PF_MEMALLOC, 113
- PF_MEMDIE, 113
- pg0, 40
- pg1, 40

PGD, 33
pgdat_list, 15, 17
PGDIR_SHIFT, 36
pgd_alloc(), 39
pgd_free(), 39
pgd_offset(), 37
pgd_quicklist, 39
pgd_t, 33
pgd_val(), 36
__pgd(), 36
pglist_data, 15, 16
pgprot_t, 36
pgprot_val(), 36
__pgprot(), 36
PG_active, 27
PG_arch_1, 27
PG_checked, 27
pg_data_t, 15, 16
PG_dirty, 27
PG_error, 27
PG_fs_1, 27
PG_highmem, 27
PG_launder, 27
PG_locked, 27
PG_lru, 27
PG_referenced, 27
PG_reserved, 27
PG_skip, 27
PG_slab, 27
PG_unused, 27
PG_uptodate, 27
Physical to virtual address mapping, 42
phys_to_virt(), 42
PKMap address space, 153
PKMAP_BASE, 55, 153
pkmap_count, 154
pkmap_map_wait, 155
pkmap_page_table, 154
pmap, 24
PMD, 33
pmd_alloc_one_fast(), 40
pmd_alloc_one(), 40
pmd_alloc(), 39
pmd_free(), 39
pmd_offset(), 37
pmd_page(), 39
pmd_quicklist, 39
PMD_SHIFT, 35
pmd_t, 34
pmd_val(), 36
__pmd(), 36
Process address space, 53
Process descriptor allocation, 60
Process ~~ids~~, 111
Process killing, 211
Process pageout, 173
Process space descriptor, 57
PTE allocation, 39
PTE chains, 31, 48
PTE freeing, 39
PTE instantiation, 39
PTE macros, 37
PTE protection bits, 36
PTE to swap entry mapping, 183
ptep_get_and_clear(), 39
PTEs in high memory, 50
pte_alloc_one_fast(), 40
pte_alloc_one(), 40
pte_alloc(), 39
pte_clear(), 39
pte_dirty(), 38
pte_exec(), 38
pte_exprotect(), 38
pte_free(), 39
pte_mkclean(), 38
pte_mkdirty(), 38
pte_mkexec(), 38
pte_mkread(), 38
pte_mkwrite(), 38
pte_mkyoung(), 38
pte_modify(), 38
pte_offset_map(), 51
pte_offset(), 37
pte_old(), 38
pte_page(), 39
pte_quicklist, 39
pte_rdprotect(), 38
pte_read(), 38
pte_t, 34
pte_to_swp_entry(), 183
pte_val(), 36

- pte_write(), 38
- pte_wrprotect(), 38
- pte_young(), 38
- __pte(), 36
- PTRS_PER_PGD, 36
- PTRS_PER_PMD, 36
- PTRS_PER_PTE, 36
- put_user(), 89
- Quicklists, 39
- RAM-based ~~page~~ system, 195
- read_swap_cache_async(), 189
- REAP_SCANLEN, 135
- Retiring inactive_list, 169
- refill_inactive(), 164, 169
- remap_file_pages(), 93
- remove_exclusive_swap_page(), 189
- remove_inode_page(), 166
- remove_page_from_hash_queue(), 166
- remove_page_from_inode_queue(), 166
- reserve_bootmem_node(), 97
- reserve_bootmem(), 96
- Retiring boot memory, 101
- Reverse Mapping (rmap), 48
- Reverse Mapping (RMAP), 87
- Reverse mapping objects, 49
- Reverse mapping pages, 48
- rss, 59
- rw_swap_page_base(), 192
- rw_swap_page(), 189, 192
- scan_swap_map(), 185
- search_exception_table(), 79
- security_operations, 212
- security_ops, 212
- security_vm_enough_memory(), 212
- Set associative mapping, 45
- SetPageActive(), 28
- SetPageChecked(), 28
- SetPageDirty(), 28
- SetPageError(), 28
- SetPageLaunder(), 28
- SetPageReferenced(), 28
- SetPageReserved(), 28
- SetPageUptodate(), 26, 28
- setup_arch(), 98
- setup_arg_flags(), 63
- setup_memory(), 98
- SET_PAGE_CACHE(), 138
- SET_PAGE_SLAB(), 138
- set_page_zone(), 29
- set_pte(), 39
- set_shrinker(), 152
- Shared regions, 204
- shm, 195
- shmat(), 204
- Shmem ~~page~~ creation, 199
- Shmem functions, 197
- shmem_commit_write(), 197
- shmem_dir_inode_operations, 198
- shmem_file_operations, 198
- shmem_getpage(), 199
- shmem_inodes, 195
- shmem_inode_cache, 207
- shmem_inode_info, 196
- shmem_inode_operations, 198
- SHMEM_I(), 196
- SHMEM_MAX_INDEX, 202
- shmem_prepare_write(), 197
- shmem_readpage(), 197
- shmem_symlink_inline_operations, 198
- shmem_symlink_inode_operations, 198
- shmem_writepage(), 197
- shmget(), 204
- SHM_DEST, 207
- SHM_HUGETLB, 52
- SHM_LOCK, 197
- SHM_UNLOCK, 197
- shm_vm_ops, 204
- shrink_caches(), 173
- shrink_cache(), 170
- Size-N cache, 146
- Size-N (DMA) cache, 146
- Slab allocator, 123
- Slab cache allocation ~~page~~s, 131
- Slab cache chain, 123
- Slab cache coloring, 132
- Slab cache creation, 133

- Slab cache deletion, 136
- Slab cache dynamic ~~Yes~~, 131
- Slab cache reap, 133
- Slab cache shrinking, 135
- Slab cache static ~~Yes~~, 130
- Slab cache, 123
- Slab caches, per-CPU, 148
- Slab creation, 139
- Slab debugging, 125
- Slab deletion, 143
- Slab descriptors, 138
- Slab free object tracking, 141
- Slab initialization, 150
- Slab object allocation, 144
- Slab objects, 144
- slabinfo, 125
- Slabs, 123, 137
- Slabs, buddy interaction, 151
- Slabs, ~~Adding~~ free objects, 142
- Slabs, number of objects, 143
- slabs_free, 127
- slabs_full, 127
- slabs_partial, 127
- SLAB_ATOMIC, 132
- slab_bufctl(), 141
- SLAB_CACHE_DMA, 131
- SLAB_CTOR_ATOMIC, 132
- SLAB_CTOR_CONSTRUCTOR, 132
- SLAB_CTOR_VERIFY, 132
- SLAB_DEBUG_FREE, 131
- SLAB_DEBUG_INITIAL, 131
- SLAB_DMA, 132
- SLAB_HWCACHE_ALIGN, 131
- SLAB_KERNEL, 132
- SLAB_MUST_HWCACHE_ALIGN, 131
- SLAB_NFS, 132
- SLAB_NOFS, 132
- SLAB_NOHIGHIO, 132
- SLAB_NOIO, 132
- SLAB_NO_REAP, 131
- SLAB_POISON, 131
- SLAB_RED_ZONE, 131
- SLAB_USER, 132
- Small allocation caches, 123, 146
- Sourceforge.net, 3
- SPEC, 11
- Stack algorithm, 165
- Stack expansion, 82
- startup_32(), 40
- strlen_user(), 89
- strncpy_from_user(), 89
- struct kmem_cache_s, 127
- Swap area, 179
- Swap area, describing, 180
- Swap area deactivating, 193
- Swap area initializing, 192
- Swap cache, 163, 185
- Swap entries, 183
- Swap management, 179
- Swap reading, 189, 192
- Swap writing, 189
- SWAPFILE_, 185
- swpin_readahead(), 80, 87
- swapper_pg_dir, 40
- Swapping, 179
- SWAP_CLUSTER_MAX, 173
- swap_duplicate(), 186, 188
- swap_free(), 186
- swap_header, 182
- swap_info, 180, 183
- swap_info_struct, 180
- swap_list, 181
- SWAP_MAP_BAD, 181
- SWAP_MAP_MAX, 181
- swap_mm, 173
- swap_ops, 186
- swap_out_mm(), 173
- swap_out_vma(), 175
- swap_out(), 163, 173
- swp_entry_t, 183
- swp_entry_to_pte(), 183
- SWP_ENTRY(), 183
- SWP_OFFSET(), 183
- SWP_TYPE(), 183
- SWP_USED, 180
- SWP_WRITEOK, 180
- System V IPC, 204
- sys_mmap2(), 67
- sys_mprotect(), 72
- sys_mremap(), 72
- sys_munlockall(), 76
- sys_munlock(), 76

- sys_swapoff(), 193
- sys_swapon(), 192
- TestClearPageLRU(), 28
- TestSetPageLRU(), 28
- Thread identification, 57
- Thundering herd, 22
- TLB API, 44
- TLB flushing, 57
- TLB, 33, 43
- tmpfs, 195
- total_vm, 59
- tq_disk, 177
- Translation Lookaside Buffer (TLB), 33, 43
- Trivial Patch Monkey, 12
- try_to_free_buffers(), 111
- try_to_swap_out(), 175
- try_to_unuse(), 193
- Understanding the Linux Kernel*, 1
- Unigine, 4
- UnlockPage(), 22, 28
- unmap_fixup(), 79
- update_mmu_cache(), 45
- UseNet, 3
- Userspace accessing, 87
- __va(), 42
- vfree(), 120
- Virtual Memory Area, 61
- virt_to_page(), 43
- VM Regress, 12
- VMA creation, 67
- VMA deletion, 79
- VMA insertion, 70
- VMA locking, 75
- VMA merging, 72
- VMA operations, 64
- VMA remapping, 72
- VMA searching, 69
- VMA unlocking, 76
- VMA, 61
- vmalloc address space, 118
- vmalloc areas, 117
- vmalloc_32(), 118, 119
- vmalloc_dma(), 118, 119
- VMALLOC_END, 117
- VMALLOC_OFFSET, 55
- VMALLOC_RESERVE, 55
- VMALLOC_START, 117
- vmalloc(), 112, 117, 119
- vmap(), 121
- __vma_link(), 71
- vma_link(), 72
- vma_merge(), 70, 72
- vmlist_lock, 118
- vm space, 55
- VM_ACCOUNT, 92, 211
- vm_acct_memory(), 211
- VM_ALLOC, 118
- vm_area_struct, 56, 61
- VM_DENYWRITE, 63
- vm_enough_memory(), 209
- VM_EXECUTABLE, 63
- VM_GROWSDOWN, 63
- VM_GROWSUP, 63
- VM_IO, 63
- VM_IOREMAP, 118
- VM_LOCKED, 63
- VM_MAYEXEC, 63
- VM_MAYREAD, 63
- VM_MAYSHARE, 63
- VM_MAYWRITE, 63
- vm_operations_struct, 64
- VM_RAND_READ, 63
- VM_RESERVED, 63
- VM_SEQ_READ, 63
- VM_SHM, 63
- VM_STACK_FLAGS, 63
- vm_struct, 117
- vm_unacct_memory(), 212
- vsyscall page, 90
- Waiting on pages, 22
- wait_on_page(), 22
- wait_table_size(), 22
- Working set, 164
- Zero page, 53
- Zone balance, 20
- Zone dimensions, 15

Zone fallbacks, 17	ZONE_HIGHMEM, 15
Zone initialization, 23	zone_holes, 24
Zone modifiers, 110	ZONE_NORMAL, 15
Zone pressure, 19	ZONE_NORMAL at 896MiB, 55
Zone size calculation, 20	ZONE_PADDING(), 31
Zone structure, 18	zone_sizes_init(), 41
Zone watermarks, 19	zone_start_paddr, 24
Zones, 15	zone_struct, 15, 18
zones_sizes, 24	zone_t, 15
ZONE_DMA, 15	zone_table, 29

About the Author



MEL GORMAN earned B.Sc. and M.Sc. degrees in Computer Science from the University of Limerick, Ireland. He has worked on a wide variety of applications ranging from web development to real-time display systems for cameras. His toughest projects have been those he figured could not possibly be as hard as they were made out to be, like the Linux VM. He has also worked as a system administrator, mainly with Linux but also with Solaris and Windows. Currently Mel is a Java applications developer for IBM in Santry, Dublin.

Mel developed most of his skills through a healthy combination of college experience, exposure to the UL Computer Society and real-world work. It was the Computer Society that introduced him to Linux and convinced him that it was useful for more than a nice email address. The society developed his interest in Open Source software and particularly in the Linux kernel. He is eternally grateful to UL for providing the platform, the people and the time to work on the VM for two years.

When not pushing buttons, Mel enjoys spending time with his girlfriend, Karen, playing the guitar (badly), reading anything he can get his hands on, hanging out with friends and family (who, for some reason, avoid conversations about the VM), developing plans of dubious merit (sometimes implementing them with novel results) and getting into a kayak any time Andy convinces him it is a good idea. He is still deciding whether to continue writing about Linux or get back to programming with it, which was his original intention.

About the CD-ROM

The CD-ROM that accompanies *Understanding the Linux Virtual Memory Manager* contains software and tools that aid code comprehension of the kernel. The CD-ROM includes the following:

- An installation of Apache that is usable with any GNU/Linux system. Mount the CD with the command `mount /dev/cdrom /cdrom -o exec`; start the server with the command `/cdrom/start_server`.
- A web-browsable copy of the Linux 2.4.22 and 2.6.0-test4 kernels. This allows code to be browsed and identified to be searched for.
- A searchable index for functions that have a code commentary available. If a function is searched for that does not have a commentary, the browser will be automatically redirected to a web-browsable version of the source.
- A live version of CodeViz, the tool used to generate call graphs for the book. The generated graphs may be clicked upon to bring the browser to that function in the source code.
- The VMRegress, CodeViz and PatchSet packages. gcc-3.0.4 is also provided because it is required for building CodeViz.

The CD-ROM can be used on the GNU/Linux or Windows 95/98/NT/2000/Me/XP platforms. The web server and site require that a GNU/Linux system is used and has been tested with Debian Woody and Red Hat 7.3. It is strongly recommended that you use a GNU/Linux system to access the CD.

License Agreement

Use of the software accompanying *Understanding the Linux Virtual Manager* is subject to the terms of the License Agreement and Limited Warranty found on the previous pages.

Technical Support

Prentice Hall does not offer technical support for any of the programs on the CD-ROM. However, if the CD-ROM is damaged, you may obtain a replacement copy by sending an email that describes the problem to: disc.exchange@prenhall.com