# Understanding The Linux Virtual Memory Manager

Mel Gorman

14th July 2003

# Contents

# List of Figures

# List of Tables

# Abstract

Linux is developed with a strong practical emphasis more than a theoretical one. When new algorithms are suggested or existing implementations questioned, 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 from the theory considerably. In part, Linux does follow the traditional development cycle of design to implementation but it is more common for changes to be made in reaction to how the system behaved in the "real-world" and intuitive decisions by developers. These intuitive changes can be a hindrance as they are rarely backed by controlled, repeatable experiments. Consequently, some design choices have been made without a strong foundation.

This has led to a situation where the VM is poorly documented except for a small number of web sites with incomplete coverage. The existing books on Linux are comprehensive but they try to cover the entire kernel and sometimes leave out the details of the VM. This leads to the VM being fully understood by only a small number of core developers. Developers looking for information on how it functions are generally told to read the source and 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 to read the code and study the field of Memory Management. The problem is further compounded by the fact that the code comments, if they even exist, only indicate what is happening in a very small instance. This makes difficult to see how the overall system functions as is roughly analogous to using a microscope to identify a piece of furniture.

As Linux gains in popularity, in the business as well as the academic world, more developers are expressing an interest in developing Linux to suit their needs and the lack of detailed documentation is a significant barrier to entry for a new developer or researcher who wishes to study the VM.

The objective of this thesis is to document fully how the 2.4.20 VM works including its structure, the algorithms used, the implementations thereof and the Linux-specific features. Combined with the companion document "Code Commentary on the Linux Virtual Memory Manager" the documents act as a detailed tour of the code explaining almost line by line how the VM operates and where applicable, explains the theoretical basis for the implementation. It will also describe how to approach reading through the kernel source including tools aimed at making the code easier to read, browse and understand.

It is envisioned that this will drastically reduce the amount of time a developer or researcher needs to invest to understand what is happening inside the Linux VM. This applies even if the VM of interest is a later version as the time needed to understand changes and extensions is considerably less than the time required to learn the fundamentals of the Linux VM.

# Acknowledgments

The compilation of this document was not a trivial task and I am glad I did not know how much work was ahead of me when it was started two years ago. It would be remiss of me not to mention some of the people who helped me at various intervals. If there is anyone I missed, I apologise now. My only excuse is that I lost a significant percentage of my mail due to hardware failure and these acknowledgments are based on the emails I recovered.

First I would like to thank Dr. John O'Gorman, my first supervisor, who tragically passed away during the course of research. His patience and guidance is what ensured this thesis exists. May he Rest In Peace and personally I hope he would have been proud of this.

With the technical research, a number of people provided invaluable insight. Abhishek Nayani, who started the Linux Kernel Documentation Project, was a source of encouragement and enthusiasm for the project. It was through discussions with him and working on LKDP that I gained strong motivation for the work and deciphered some of the early subsystems, particularly the buddy allocator. Ingo Oeser kindly offered assistance should I became seriously stuck and provided a detailed explanation how data is copied from userspace with some valuable historical context. Scott Kaplan made numerous corrections to a number of systems from non-contiguous memory allocation, to swap management to page replacement policy. His criticisms and insight on both the thesis and VM Regress prevented me entering a number of blind alleys. Jonathon Corbet provides the most detailed account of the history of the kernel development through the years with the kernel page he writes for Linux Weekly News. Carlsberg doesn't do kernel documentation but if they did, they would pay Jonathon Corbet as LWN is possibly the best source for explaining kernel features on the Internet and worth every cent of the subscription fee. Zack Brown, the chief behind Kernel Traffic is the sole reason I did not drown in kernel related mail. Finally, I would like to thank my current supervisor Dr. Patrick Healy who kindly took over after John died. He was crucial to ensuring that this document is accurate, consistent and valuable to people who are familiar, but not experts, with the Linux Kernel or memory management. Without him, there are a number of sections that would be a lot more opaque.

A number of people helped with smaller technical issues and general inconsistencies where material was not covered in sufficient depth. I feel that there is a much larger list of people but I lost their mails so if you are one of the missing people, sorry. The people I do have are Muli Ben-Yehuda, Parag Sharma, Matthew Dob-

son, Roger Luethi, Brian Lowe and Scott Crosby. All of them sent corrections and queries on differnet parts of the document which ensured I did not assume too much prior knowledge of either the kernel or memory management.

Four people helped me polish the document and suffered through my grammar and spelling mistakes to make it readable for the rest of the world. Carl Spalletta sent a number of queries and corrections to every aspect of the thesis. Steve Greenland sent a large number of grammar corrections. Philipp Marek went above and beyond being helpful sending over 90 separate corrections and queries on various aspects. The last person, whose name I cannot remember but is an editor for a magazine sent me over 140 corrections against an early version to the document. You know who you are, thanks.

Three people sent a few corrections, though small, 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, Aris Sotiropoulos, Michael Babcock, Kirk True and David Wilson.

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

The next long list are people who sent me encouragement, praise 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 Shafi, 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. I would like to thank my parents who kept me in the University long after I should have been earning enough money to support myself. I would like to thank my girlfriend Karen, who patiently listened to numerous rants, tech babble, angsting over the project and made sure I was the postgrad with the best toys. Kudos to my 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. The last people I would like to thank are the thousands of hackers that have contributed to GNU, the Linux kernel and other Free Software projects over the years who without I would not have an excellent system to work and research with. It was an inspiration to me to see such dedication when I first started programming on my own PC 6 years ago after finally figuring out that Linux was not an application for Windows used for reading email.

# Chapter 1

# Introduction

Linux is a relatively new operating system that has begun to enjoy a lot of attention from the business and academic worlds. As the operating system matures, its feature set, capabilities and performance grow but so does it's size and complexity. The table in Figure 1.1 shows the size of the kernel source code and size in bytes and lines of code of the `mm/` part of the kernel tree. This 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 still serves as a small indicator.

| Version | Release Date | Total Size | Size of mm/ | Line count |
|---|---|---|---|---|
| 1.0 | March 13th, 1992 | 5.9MiB | 96KiB | 3109 |
| 1.2.13 | February 8th, 1995 | 11MiB | 136KiB | 4531 |
| 2.0.39 | January 9th 2001 | 35MiB | 204KiB | 6792 |
| 2.2.22 | September 16th, 2002 | 93MiB | 292KiB | 9554 |
| 2.4.20 | November 28th, 2002 | 167MiB | 520KiB | 15428 |

Table 1.1: Kernel size as an indicator of complexity

As is the habit of open source project developers in general, new developers asking questions are often told to find their answer directly from the source or are advised to ask on the mailing list for beginner developers (*http://www.kernelnewbies.org*). With the Linux Virtual Memory (VM) manager, this was a suitable response for earlier kernels as the time required to understand the VM could be measured in weeks. The books available on the operating system devoted enough time to the memory management chapters to make the relatively small amount of code easy to navigate. This is no longer the case.

The books that describe Linux's internals [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 which are specific to the VM and not of general interest.

Increasingly, to get a comprehensive view on how the kernel functions, the developer or researcher is required to read through the source code line by line which

requires a large investment of time. This is especially true as the implementations of several VM algorithms diverge considerably from the papers that describe them.

In this thesis, a comprehensive guide to the VM as implemented in the 2.4.20 kernel is presented. In addition to an introduction to the theoretical background and verbal description of the implementation, a companion document called *Code Commentary On The Linux Virtual Memory Manager*, hereafter referred to as the *companion document*, provides a line-by-line tour of the code. It is envisioned that with this pair of documents, the time required to have a clear understanding of the VM, even later VMs, will be measured in weeks instead of the estimated 8 months currently required by even an experienced developer.

The VM-specific documentation that exists today is relatively poor. It is not an area of the kernel that many wish to get involved in for a variety of reasons ranging from the amount of code involved, to the complexity of the subject of memory management to the difficulty of debugging the kernel with an unstable VM.

## 1.1 General Kernel Literature

The second edition of *Understanding the Linux Kernel* was published in February 2003. It covers kernel 2.4.18 which contains a VM very similar to 2.4.20 that is discussed in this thesis. As it provides excellent coverage of the kernel, it deserves further discussion as a basis of comparison to this thesis.

First, the book tries to address the entire kernel and, while comprehensive, it misses VM details which are not of general interest. For example, it is detailed here why `ZONE_NORMAL` is exactly 896MiB and exactly how per-cpu caches are implemented. Other aspects of the VM, such as the boot memory allocator, which are not of general kernel interest, are addressed by this thesis. In this thesis, the kernel is discussed entirely from the perspective of the VM alone and includes many subtle details that are missed by other literature.

Secondly, this thesis discusses the theory as well as the implementation so that a researcher can follow the origin of the idea rather than possible mistaking the implementation as being unique to Linux. By understanding the underlying idea behind an implementation, the reader can build a conceptual model of what to expect making the deciphering of the code much simpler.

Finally, this thesis includes a line by line code commentary to cover even the smallest details of the VM. These type of minutiae are not covered in general books as the reader would be overwhelmed with detail. Even a reader with a very strong conceptual model of the VM may encounter difficulties when examining the actual code which is a hurdle that this thesis helps them to overcome. Previously, researchers were required to read the source to find out many of these details which is why specialised research such as this thesis are needed. With a clear and complete understanding of this thesis, later VMs can be analysed and understood in a matter of days.

## 1.2 Thesis Overview

In Chapter 2, I will go into detail on how the code may be managed and deciphered. Three tools will be introduced that are used for the analysis, easy browsing and management of code. The first is a tool called *Linux Cross Referencing (LXR)* which allows source code to be browsed as a web page with identifiers and functions highlighted as hyperlinks to allow easy browsing. The second is a tool called **gengraph** which was developed for this project and is used to generate call graphs starting from a particular function with the ability to limit the depth and what functions are displayed. The last is a simple tool for managing kernels and the application of patches. Applying patches manually can be time consuming and the use of version control software such as CVS[1] or BitKeeper[2] is not always an option. With this tool, a simple file specifies what source to use, what patches to apply and what kernel configuration to use.

In the subsequent chapters, each part of the implementation of the Linux VM will be 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 will refer to the papers that describe closest the behavior of Linux as well as covering in depth the implementation, the functions used and their call graphs so the reader will have a clear view of how the code is structured. For a detailed examination of the code, the reader is encouraged to consult the companion document.

## 1.3 Typographic Conventions

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

## 1.4 About this Document

This document is available in PDF, HTML and plain text formats at *http://www.csn.ul.ie/~mel/projects/vm*. The date on the title page will indicate when it was last updated. If you have questions, comments or suggestions, email Mel Gorman <*mel@csn.ul.ie*>.

[1] *http://www.cvshome.org/*
[2] *http://www.bitmover.com*

## 1.5 Companion CD

A companion CD is available and should be included with this thesis if it is provided by University of Limerick. At time of writing, it is not publicly available but when it is, it will be available for download at *http://www.csn.ul.ie/~mel/projects/vm/*. The CD is designed to be used under Linux and mounted on /cdrom with the command

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

The mount point and options are only important if you wish to start the web server that is installed on the CD. Please note that the default options normally used for mounting CDs may not allow the server to start. The CD has three important components:

- A web server started is available which is started by **/cdrom/start_server**. After starting it, the URL to access is *http://localhost:10080*. It has been tested with Red Hat 7.3 and Debian Woody;
- The "Code Commentary" companion document and this thesis are available from the `/cdrom/docs/` directory in HTML, PDF and plain text formats;
- The **VM Regress**, **gengraph** and **patchset** packages which are discussed in Chapter 2 are available in `/cdrom/software`. **gcc-3.0.1** is also provided as it is required for building **gengraph**.

### 1.5.1 Companion CD Web Server

An unmodified copy of **Apache 1.3.27** (*http://www.apache.org/*) has been built and configured to run from the CD which must be mounted on `/cdrom/`. To start it, run the script **/cdrom/start_server**. If there are no errors, the output should look like:

```
mel@joshua:~$ /cdrom/start_server
Starting Apache Server: done
The URL to access is http://localhost:10080/
```

The URL supplied is a small web site which allows easy browsing of the CD. The most noteworthy feature of the web site is a local running copy of the **LXR** (see Section 2.1.2) which allows the Linux source code to be browsed as a web page with hyperlinks for functions and identifiers. It greatly simplifies source code browsing.

To shutdown the server, execute the script **/cdrom/stop_server** and the CD may be unmounted.

### 1.5.2 Code Commentary Companion Document

The companion document is a considerably sized document. As opposed to including it as a large appendix, it is available from the companion CD in PDF, HTML and plain text formats in the `/cdrom/docs` directory and links are on the companion CDs web site. It is also available at *http://www.csn.ul.ie/~mel/projects/vm/*.

# Chapter 2

# Code Management

One of the largest initial obstacles to understanding the 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 has to be developed by each developer on their own.

The advice that is frequently offered to new developers is to read books on general operating systems, on Linux specifically, visit the kernel newbies website[1] and then read the code, benchmark the kernel and write a few documents. There is a recommended reading list provided on the website but there is no set of recommended tools for analysing and breaking down the code and, while reading the code from beginning to end is admirable, it is hardly the most efficient method of understanding the kernel.

Hence, this section is devoted to describing what tools were used during the course of researching this document to make understanding and managing the code easier and to aid researchers and developers in deciphering the kernel. It begins with a guide to how developers manage their source with patches, revision tools and how developers sometimes develop their own branch which include their own set of modifications to the main development tree. We then introduce **diff** and **patch** in more detail, how to easily browse the code and analyse the flow. We then talk about how to approach the understanding of the VM and how to submit work.

## 2.1 Managing the Source

The mainline or stock kernel is principally distributed as a compressed tape archive (.tar) file available from the nearest kernel source mirror. In Ireland's case, the mirror is located at *ftp://ftp.ie.kernel.org*. The stock kernel is always the one considered to be released by the tree maintainer. For example, at time of writing, the stock kernels for 2.2.x are those released by Alan Cox, 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

[1] *http://www.kernelnewbies.org*

is the preferred method of upgrading for bandwidth considerations. Contributions made to the kernel are almost always in the form of patches which is a *unified diff* generated by the GNU tool **diff**.

**Why patches** This method of sending patches to be merged to the mailing list initially sounds clumsy but it is remarkable efficient in the kernel development environment. The principal advantage of patches is that it is very easy to show what changes have been made rather than sending the full file and viewing both versions side by side. A developer familiar with the code being patched can easily see what impact the changes will have and if they should be merged. In addition, it is very easy to quote the email from the patch and request more information about particular parts of it. There are scripts available that allow emails to be piped to a script which strips away the mail and keeps the patch available.

**Subtrees** At various intervals, individual influential developers may have their own version of the kernel which they distribute as a large patch against the mainline kernel. These subtrees generally contain features or cleanups which 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 VM. The -rmap tree is a large set of features that for various reasons never got merged into the mainline. It is heavily influenced by the FreeBSD VM and has a number of significant differences to the stock VM. The -mm tree is quite different to -rmap in that it is a testing tree with patches that are waiting to be tested before merging into the stock kernel. Much of what exists in the mm tree eventually gets merged.

**BitKeeper** In more recent times, some developers have started using a source code control system called BitKeeper[2], 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 which depends on a central server.

BitKeeper allows comments to be associated with each patch which may be displayed as a list as part of the release information for each kernel. For Linux, this means that patches preserve the email that originally submitted the patch or the information pulled from the tree so that the progress of kernel development is a lot more transparent. On release, a summary of the patch titles from each developer is displayed as a list and a detailed patch summary is also available.

As BitKeeper is a proprietary product, which has sparked any number of flame

[2] *http://www.bitmover.com*

wars[3] with free software developers, 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 there is first some discussion on the main mailing list. In open source software, code quality is considered to be directly related to the amount of peer review. As a number of CVS and plain patch portals are available to the BitKeeper tree and patches are still the preferred means of discussion, it means that at no point is a developer required to have BitKeeper to make contributions to the kernel but the tool is still something that developers should be aware of.

### 2.1.1 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 website[4]. **diff** is used to generate patches and **patch** is used to apply them. While the tools have numerous options, there is a "preferred usage".

Patches generated with **diff** should always be *unified diff* and generated from one directory above the kernel source root. A unified diff include more information that just the differences between two lines. It begins with a two line header with the names and creation date of the two files that **diff** is comparing. After that, the "diff" 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 is before and after the hunk is applied. The hunk includes "context" lines which 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 a `-`, the line is removed and a blank is to leave the line alone as 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 and it makes the scripting of applying patches easier if each patch is generated the same way.

Let us take for example, a very simple change 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 one one line minus the backslashes.

```
mel@joshua: kernels/ $ diff -u                        \
                linux-2.4.20-clean/mm/page_alloc.c \
                linux-2.4.20-mel/mm/page_alloc.c   > example.patch
```

This generates a unified context diff (-u switch) between the two files and places the patch in example.patch as shown in Figure 2.1.1.

From this patch, it is clear even at a casual glance what files are affected (`page_alloc.c`), what line it starts at (76) and the block was 8 lines before the

[3] A regular feature of kernel discussions meaning an acrimonious argument often containing insults bordering on the personal type.

[4] *http://www.gnu.org*

```
--- linux-2.4.20-clean/mm/page_alloc.c Thu Nov 28 23:53:15 2002
+++ linux-2.4.20-mel/mm/page_alloc.c Tue Dec  3 22:54:07 2002
@@ -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 2.1: Example Patch

changes and 23 after them. The new lines are clearly marked with a +. If a patch consists of multiple hunks, each will be treated separately during patch application.

Patches broadly speaking come in two varieties, plain text such as the one above which are sent to the mailing list and a compressed form with **gzip** (.gz extension) or **bzip2** (.bz2 extension). It can be generally assumed that patches are taken from one level above the kernel root so can be applied with the option `-p1`. This option means that the patch is generated with the current working directory being one above the Linux source directory and the patch is applied while in the source directory. Broadly speaking, this means a plain text patch to a clean tree can be easily applied as follows

```
mel@joshua: kernels/ $ cd linux-2.4.20-clean/
mel@joshua: linux-2.4.20-clean/ $ patch -p1 < ../example.patch
patching file mm/page_alloc.c
mel@joshua: linux-2.4.20-clean/ $
```

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

```
mel@joshua: linux-2.4.20-mel/ $ gzip -dc ../example.patch.gz | patch -p1
```

If a hunk can be applied but the line numbers are different, the hunk number and the number of lines needed to offset will be output. These are generally safe warnings and may be ignored. If there are slight differences in the context, it 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.

### 2.1.2 Browsing the Code

When code is small and manageable, it is not particularly difficult to browse through the code as 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 making tracking them down time consuming.

An early solution to the problem of easy code browsing was **ctags** which could generate tag files from a set of source files. These tags could be used to jump to the C file and line where the function existed with editors such as **Vi** and **Emacs**. This method can become cumbersome if there is many functions with the same name. With Linux, this is the case for functions declared in the architecture dependant code.

A more comprehensive solution is available with the *Linux Cross-Referencing (LXR)* tool available from *http://lxr.linux.no*. The tool provides the ability to represent source code as browsable web pages. Global identifiers such as global variables, macros and functions become hyperlinks. When clicked, the location where it is defined is displayed along with every file 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 easily installed as the documentation is very clear. For the research of this document, it was deployed at *http://monocle.csis.ul.ie* which was used to mirror recent development branches. All code extracts shown in this and the companion document were taken from LXR so that the line numbers would be visible.

### 2.1.3 Analysing Code Flow

As 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.

Based partially on the work of Martin Devera[5], I developed a tool called **gengraph**. The tool can be used to generate call graphs from any given C code that has been compiled with a patched version of **gcc**.

During compilation with the patched compiler, files with a .cdep extension are generated for each C file which lists all functions and macros that are contained in other C files as well as any function call that is made. 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[6].

In kernel 2.4.20, there were a total of 28626 entries 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 just the name of one or more functions as an argument and generates a call graph with the requested function as the root node. This can result in unnecessary depth to the graph or graph functions that the user is not interested in, therefore 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 it will not appear on the call graph or any of the functions they call. The last is to display a function, but not traverse it which is convenient when the function is covered on a separate call graph.

All call graphs shown in these documents are generated with the **gengraph** package available at *http://www.csn.ul.ie/~mel/projects/gengraph* as it is often much easier to understand a subsystem at first glance when a call graph is available. It has been tested with a number of other open source projects based on C and has wider application than just the kernel.

### 2.1.4 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 tool was developed called **patchset** designed for the management of kernel sources.

It uses files called *set configurations* to specify what kernel source tar to use, what patches to apply, what configuration to use for the build and what the resulting kernel is to be called. A sample specification file to build kernel 2.4.20-rmap15a is;

```
linux-2.4.18.tar.gz
2.4.20-rmap15a
config_joshua

1 patch-2.4.19.gz
1 patch-2.4.20.gz
1 2.4.20-rmap15a
```

[5] *http://luxik.cdi.cz/~devik*
[6] *http://www.graphviz.org*

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-rmap15a** and the third line specifies which kernel configuration file to use for building the kernel. Each line after that has two parts. The first part says what patch depth to use i.e. what number to use with the -p switch to patch. As discussed earlier in Section 2.1.1, 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 above example will apply two patches to update the kernel from 2.4.18 to 2.4.20 before building the **2.4.20-rmap15a** kernel tree.

The package comes with three scripts. The first **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. The second **make-gengraph.sh** will unpack the kernel but instead of building an installable kernel, it will generate the files required to use **gengraph** for creating call graphs. The last `make-lxr.sh` will install the kernel to the LXR root and update the versions so that the new kernel will be displayed on the web page.

With the three scripts, a large amount of the tedium involved with managing kernel patches is eliminated. The tool is fully documented and freely available from *http://www.csn.ul.ie/~mel/projects/patchset*.

## 2.2 Getting Started

When a new developer or researcher asks how to begin reading the code, they are often recommended to start with the initialisation code and work from there. I do not believe that this is the best approach as initialisation is quite architecture dependent and requires a detailed hardware knowledge to decipher it. It also does not give much information about how a subsystem like the VM works as it is only in the late stages of initialisation that memory is set up in the way the running system sees it.

The best starting point for kernel documentation is first and foremost the `Documentation/` tree. It is very loosely organised but contains much Linux specific information that will be unavailable elsewhere. The second visiting point is the Kernel Newbies website at *http://www.kernelnewbies.org* which is a site dedicated to people starting kernel development and includes a *Frequently Asked Questions (FAQ)* section and a recommended reading list.

The best starting point to understanding the VM, I believe, is now this document and the companion 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 rather than totally new so understanding the 2.4.20 VM is an important starting point.

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. The second subsystem to then examine is the non-contiguous memory allocator located in `mm/vmalloc.c` and discussed in Chapter 8 as it is reasonably contained within one file. The third system should be physical page allocator located in `mm/page_alloc.c` and discussed in Chapter 7 for similar reasons. The fourth system of interest is the creation of VMAs and memory areas for processes discussed in Chapter 5. Between these systems, they have the bulk of the code patterns that are prevalent throughout the rest of the kernel code making the deciphering of more complex systems such as the page replacement policy or the buffer IO much easier to comprehend.

The second recommendation that is given by experienced developers is to benchmark and test but unfortunately the VM is difficult to test accurately and benchmarking is just a shade above vague handwaving at timing figures. A tool called **VM Regress** was developed during the course of research and is available at *http://www.csn.ul.ie/~mel/vmregress* that lays the foundation required to build a fully fledged testing, regression and benchmarking tool for the VM. It 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 but is in the early stages of development at time of writing.

## 2.3 Submitting Work

A quite comprehensive set of documents on the submission of patches is available in the `Documentation/` part of the kernel source tree and it is important to read. There are two files `SubmittingPatches` and `CodingStyle` which cover the important basics but there seems to be very little documentation describing how to go about getting patches merged. Hence, 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 as having a style inconsistent with the main kernel will be a barrier to getting merged regardless of the technical merit. Once a patch has been developed, the first problem is to decide where to send it. Kernel development has a definite, if non-apparent, hierarchy of who handles patches and how to get them submitted. As an example, we'll take the case of 2.5.x development.

The first check to make is if the patch is very small or trivial. If it is, post it to the main kernel mailing list. If there is no bad reaction, 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 final say on what gets accepted into the main tree. Linus, with rare exceptions, accepts patches only from who he refers to as his

[7] *http://www.kernel.org/pub/linux/kernel/people/rusty/trivial/*

"lieutenants", a group of around 10 people who he trusts to "feed" 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 "feed" 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 which 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` file. The second major area of influence will be from the subsystem specific mailing list if there is one. The VM does not have a list of maintainers but it does have a mailing list[8].

The maintainers and lieutenants are crucial to the acceptance of patches. Linus, broadly speaking, does not appear to wish 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 exists.

In summary, a new patch should be emailed to the subsystem mailing list cc'd to the main list to generate discussion. If there is no reaction, it should be sent to the maintainer for that area of code if there is one and to the lieutenant if there is not. Once 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 it while it is still manageable. There are notable cases where massive patches had difficult getting merged because there were long periods of silence with little or no discussions. A recent example of this is the Linux Kernel Crash Dump project which still has not been merged into the main stream because there has not been favorable from lieutenants or strong support from vendors.

[8] *http://www.linux-mm.org/mailinglists.shtml*

# Chapter 3

# Describing Physical Memory

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

The first principle 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 the "distance" from the processor. For example, there might be a bank of memory assigned to each CPU or a bank of memory very suitable for DMA near device cards.

Each bank is called a *node* and the concept is represented under Linux by a `struct pg_data_t` even if the architecture is UMA. 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 will be discussed further in Section 3.1.

Each node is divided up into a number of blocks called *zones* which represent ranges within memory. Zones should not be confused with zone based allocators as they are unrelated. A zone is described by a `struct zone_t` and each one is one of *ZONE_DMA*, *ZONE_NORMAL* or *ZONE_HIGHMEM*. Each is suitable a different type of usage. `ZONE_DMA` is memory in the lower physical memory ranges which certain ISA devices require. Memory within `ZONE_NORMAL` be directly mapped by the kernel in the upper region of the linear address space which is discussed further in Section 5.1. With the x86 the zones are:

| | |
|---|---|
| `ZONE_DMA` | First 16MiB of memory |
| `ZONE_NORMAL` | 16MiB - 896MiB |
| `ZONE_HIGHMEM` | 896 MiB - End |

It is important to note that many kernel operations can only take place using `ZONE_NORMAL` so it is the most performance critical zone. `ZONE_HIGHMEM` is the rest of memory. Zones are discussed further in Section 3.2.

Each physical page frame is represented by a `struct page` and all the structures are kept in a global `mem_map` array which is usually stored at the beginning of `ZONE_NORMAL` or just after the area reserved for the loaded kernel image in low

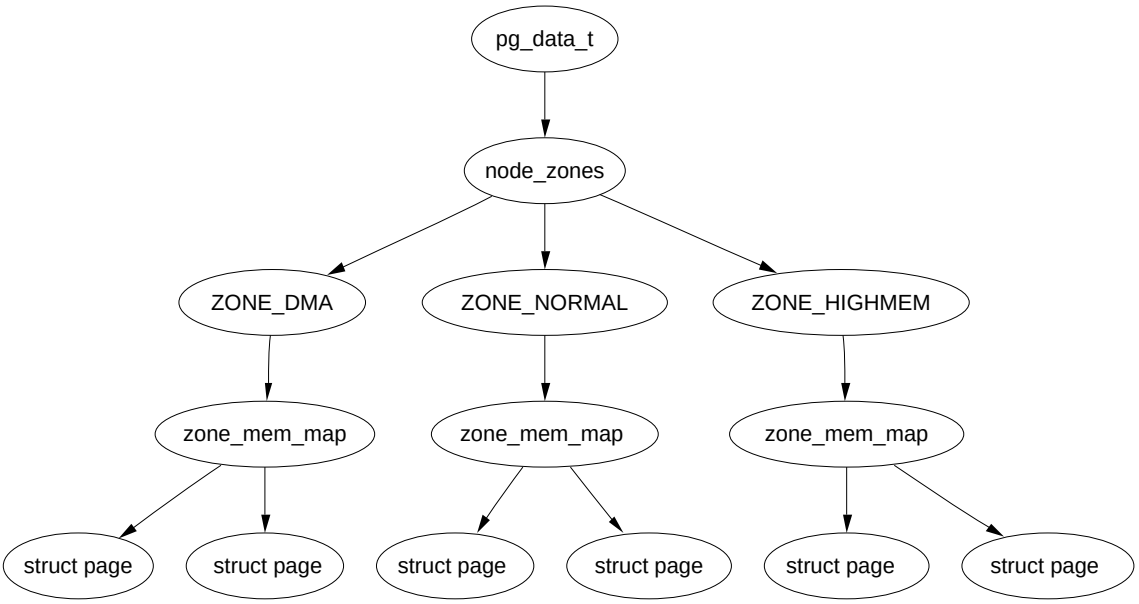


Figure 3.1: Relationship Between Nodes, Zones and Pages

memory machines. `struct page`s are discussed in detail in Section 3.3 and the global `mem_map` array is discussed in detail in Section 4.7. The basic relationship between all these structures is illustrated in Figure 3.1.

As the amount of memory directly accessible by the kernel (`ZONE_NORMAL`) is limited in size, Linux supports the concept of *High Memory* which is discussed in detail in Chapter 10. This chapter will discuss how nodes, zones and pages are represented before introducing high memory management.

## 3.1 Nodes

As we have mentioned, each node in memory is described by a `pg_data_t` struct. When allocating a page, Linux uses a *node-local allocation policy* to allocate memory from the node closest to the running CPU. As processes tend to run on the same CPU, it is likely the memory from the current node will be used. The struct is declared as follows in <`linux/mmzone.h`>:

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

We now briefly describe each of these fields:

**node_zones** The zones for this node, `ZONE_HIGHMEM`, `ZONE_NORMAL`, `ZONE_DMA`;

**node_zonelists** This is the order of zones that allocations are preferred from. `build_zonelists()` in `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** Number of zones in this node, between 1 and 3. Not all nodes will have three. A CPU bank may not have `ZONE_DMA` for example;

**node_mem_map** This is the first page of the `struct page` array representing each physical frame in the node. It will be placed somewhere within the global `mem_map` array;

**valid_addr_bitmap** A bitmap which describes "holes" in the memory node that no memory exists for;

**bdata** This is only of interest to the boot memory allocator discussed in Chapter 6;

**node_start_paddr** The starting physical address of the node. An unsigned long does not work optimally as it breaks for ia32 with *Physical Address Extension (PAE)*[1] for example. A more suitable solution would be to record this as a *Page Frame Number (PFN)* which could be trivially defined 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`;

[1] PAE is discussed further in Section 3.4.

**node_size** The total number of pages in this zone;

**node_id** The ID of the node, 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 initialised by the `init_bootmem_core()` function, described later in Section 6.2.2. Up until late 2.4 kernels (> 2.4.18), blocks of code that traversed the list looked something like:

```
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 for loop, is provided to improve code readability.

## 3.2 Zones

Zones are described by a `struct zone_t`. It keeps track of information like page usage statistics, free area information and locks. It is 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 to protect the zone;

**free_pages** Total number of free pages in the zone;

**pages_min, pages_low, pages_high** These are zone watermarks which are described in the next section;

**need_balance** This flag that tells the pageout **kswapd** to balance the zone;

**free_area** Free area bitmaps used by the buddy allocator;

**wait_table** 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()`. While processes could all wait on one queue, this would cause a "thundering herd" of processes to race for pages still locked when woken up;

**wait_table_size** Size of the hash table which is a power of 2;

**wait_table_shift** Defined as the number of bits in a long minus the binary logarithm of the table size above;

**zone_pgdat** Points to the parent `pg_data_t`;

**zone_mem_map** The first page in the global `mem_map` this zone refers to;

**zone_start_paddr** Same principle as `node_start_paddr`;

**zone_start_mapnr** Same principle as `node_start_mapnr`;

**name** The string name of the zone, "DMA", "Normal" or "HighMem"

**size** The size of the zone in pages.

### 3.2.1 Zone Watermarks

When available memory in the system is low, the pageout daemon **kswapd** is woken up to start freeing pages (see Chapter 11). If the pressure is high, the process will free up memory synchronously which is sometimes referred to as the direct reclaim path. The parameters affecting pageout behavior are similar to those 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 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).

**pages_min** When `pages_min` is reached, the allocator will do the **kswapd** work in a synchronous fashion. There is no real equivalent in Solaris but the closest is the `desfree` or `minfree` which determine how often the pageout scanner is woken up;

**pages_low** When `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;

**pages_high** Once reached, **kswapd** is woken, it will not consider the zone to be "balanced" until `pages_high` pages are free. 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.

## 3.3 Pages

Every physical page frame in the system has an associated `struct page` which is used to keep track of its status. In the 2.2 kernel [BC00], this structure resembled it's equivilent in System V [GC94] but like the other families in UNIX, the structure changed considerably. It is declared as follows in <`linux/mm.h`>:

```
152 typedef struct page {
153         struct list_head list;
154         struct address_space *mapping;
155         unsigned long index;
156         struct page *next_hash;
158         atomic_t count;
159         unsigned long flags;
161         struct list_head lru;
163         struct page **pprev_hash;
164         struct buffer_head * buffers;
175
176 #if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)
177         void *virtual;
179 #endif /* CONFIG_HIGMEM || WANT_PAGE_VIRTUAL */
180 } mem_map_t;
```

Here is a brief description of each of the fields:

**list** Pages may belong to many lists and this field is used as the list head. For example, pages in a mapping will be in one of three circular linked links

kept by the address_space. These are clean_pages, dirty_pages and locked_pages. In the slab allocator, this field is used to store pointers to the slab and cache the page belongs to. It is also used to link blocks of free pages together;

**mapping** When files or devices are memory mapped[2], their inode has an associated address_space. This field will point to this address space if the page belongs to the file. If the page is anonymous and mapping is set, the address_space is swapper_space which manages the swap address space;

**index** This field has two uses and it depends on the state of the page what it means. If the page is part of a file mapping, it is the offset within the file. 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). Secondly, 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 file mapping are hashed on the inode and offset. This field links pages together that share the same hash bucket;

**count** The reference count to the page. If it drops to 0, it may be freed. Any greater and it is in use by one or more processes or is in use by the kernel like when waiting for IO;

**flags** These are flags which describe the status of the page. All of them are declared in <linux/mm.h> and are listed in Table 3.1. There are a number of macros defined for testing, clearing and setting the bits which are all listed in Table 3.2;

**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 LRU lists;

**pprev_hash** The complement to next_hash;

**buffers** If a page has buffers for a block device associated with it, this field 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 file. This is necessary as the page has to be synced with backing storage in block sized chunks defined by the underlying filesystem;

**virtual** Normally only pages from ZONE_NORMAL is 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 10. There are only a fixed number of pages that may be mapped. When it is mapped, this is its virtual address;

[2]Frequently abbreviated to *mmaped* during kernel discussions.

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

### 3.3.1 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, as even such a small pointer consumes a lot of memory when thousands of struct pages 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 a page belongs to. First a zone_table of zones is set up. It is declared in <linux/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. 3. MAX_NR_NODES is the maximum number of nodes that may exist. This table is treated like a multi-dimensional array. During free_area_init_core(), all the pages in a node are initialised. First it sets the value for the table

```
734                 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);
```

page is the page to be set. So, clearly the index in the zone_table is stored in the page.

## 3.4 High Memory

As the addresses 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. This is discussed further in Chapter 10.

The second limit at 64GiB is related to *Physical Address Extension (PAE)* which is an Intel invention to allow more RAM to be used with 32 bit systems. It makes 4 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 as 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.

| Bit name | Description |
|---|---|
| PG_active | This bit is set if a page is on the active_list LRU and cleared when it is removed. It marks a page as being hot |
| PG_arch_1 | Quoting directly from the code: PG_arch_1 is an architecture specific page state bit. The generic code guarantees that this bit is cleared for a page when it first is entered into the page cache |
| PG_checked | 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_highmem | Pages in high memory cannot be mapped permanently by the kernel. Pages that are in high memory are flagged with this bit during mem_init() |
| PG_launder | 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 writepage() function. When scanning, if it encounters a page with this bit and PG_locked 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 active_list or the inactive_list, 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 6 for pages allocated during system startup. Later it is used to flag empty pages or ones that do not even exist |
| PG_slab | This will flag a page as being used by the slab allocator |
| PG_skip | Used by some architectures so skip over parts of the address space with no backing physical memory |
| PG_unused | This bit is literally unused |
| PG_uptodate | When a page is read from disk without error, this bit will be set. |

Table 3.1: Flags Describing Page Status

| Bit name | Set | Test | Clear |
|---|---|---|---|
| PG_active | SetPageActive() | PageActive() | ClearPageActive() |
| PG_arch_1 | n/a | n/a | n/a |
| PG_checked | SetPageChecked() | PageChecked() | n/a |
| PG_dirty | SetPageDirty() | PageDirty() | ClearPageDirty() |
| PG_error | SetPageError() | PageError() | ClearPageError() |
| PG_highmem | n/a | PageHighMem() | n/a |
| 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 | n/a | n/a | n/a |
| PG_slab | PageSetSlab() | PageSlab() | PageClearSlab() |
| PG_unused | n/a | n/a | n/a |
| PG_uptodate | SetPageUptodate() | PageUptodate() | ClearPageUptodate() |

Table 3.2: Macros For Testing, Setting and Clearing Page Status Bits

Secondly, 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 which 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 Linux on an x86. If more memory needs to be accessed, the advice given is simple and straightforward, buy a 64 bit machine.

# Chapter 4

# 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 which manage the underlying physical pages such as the `pmap` object in BSD but 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. While this is relatively 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 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 optimised out at compile time. Unfortunately, for architectures that do not manage their cache or *Translation Lookaside Buffer (TLB)* automatically, hooks for machine 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. Fortunately, the functions and how they have to be used is very well documented in the `cachetlb.txt` file in the kernel documentation tree [Mil00].

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 followed by how a virtual address is broken up into its component parts for navigating the table. Once covered, it will be discussed how the lowest level entry, the *Page Table Entry (PTE)* and what bits are used by the hardware. After that, the macros used for navigating a page table, 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. Finally, it will be discussed how the page tables are initialised during boot strapping.

## 4.1 Describing the Page Directory

Each process has its own *Page Global Directory (PGD)* which is a physical page frame containing an array of `pgd_t`, an architecture specific type defined in <`asm/page.h`>. The page tables are loaded differently on each architecture. On the x86, the process page table is loaded by copying the pointer to the 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 *Page Middle Directory (PMD)* entries of type `pmd_t` which in turn points to page frames containing *Page Table Entries (PTE)* of type `pte_t`, which finally points to page frames containing the actual user data. In the event 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.

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 specifies the length in bits that are mapped by each level of the page tables as illustrated in Figure 4.1.

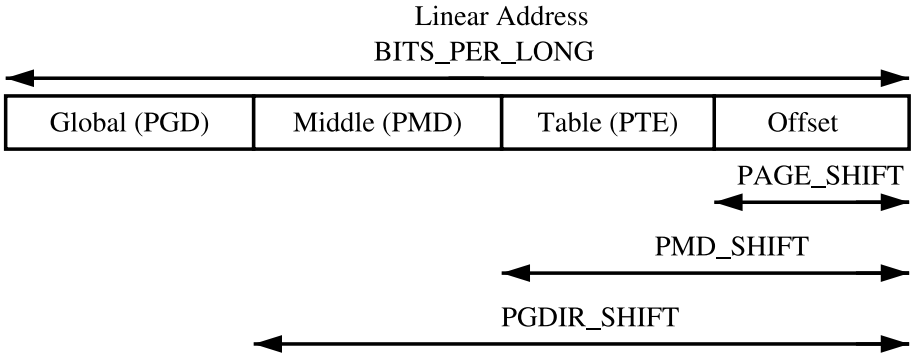


Figure 4.1: Linear Address Bit Size Macros

The `MASK` values can be ANDd with a linear address to mask out all the upper bits and is 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 by each entry at each level. The relationship between the `SIZE` and `MASK` macros is illustrated in Figure 4.2.

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

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

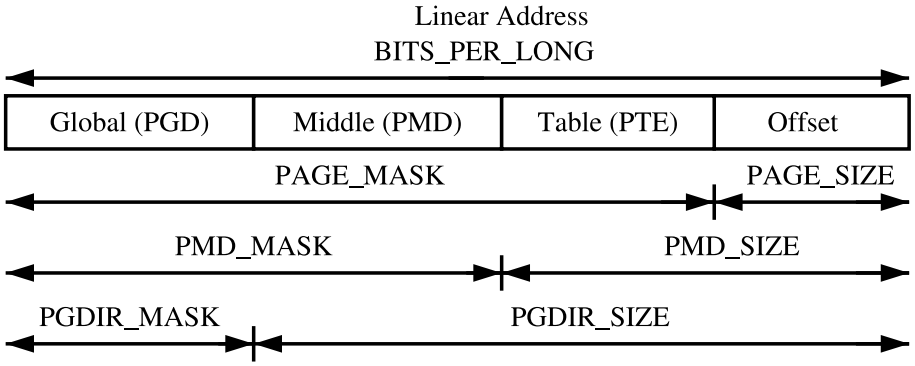


Figure 4.2: Linear Address Size and Mask Macros

`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 code above. Finally the mask is calculated as the negation of the bits which make up the `PAGE_SIZE - 1`. If a page needs to be aligned on a page boundary, `PAGE_ALIGN()` is used. This macro adds `PAGE_SIZE - 1` is added to the address before simply ANDing it with the `PAGE_MASK`.

`PMD_SHIFT` is the number of bits in the linear address which 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.

`PGDIR_SHIFT` is the number of bits which are mapped by the top, or first level, of the page table. The `PGDIR_SIZE` and `PGDIR_MASK` are calculated in the same manner as above.

The last three macros of importance are the `PTRS_PER_x` which determine the number of entries in each level of the page table. `PTRS_PER_PGD` is the number of pointers in the PGD, 1024 on an x86 without PAE. `PTRS_PER_PMD` is for the PMD, 1 on the x86 without PAE and `PTRS_PER_PTE` is for the lowest level, 1024 on the x86.

## 4.2 Describing a Page Table Entry

As mentioned, each entry is described by the structures `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 structures 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 flags and is usually stored in the lower bits of a page table entry.

For type casting, 4 macros are provided in `asm/page.h`, which takes the above types and returns the relevant part of the structures. They are `pte_val()`, `pmd_val()`, `pgd_val()` and `pgprot_val()`. To reverse the type casting, 4 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 with no 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 4.1 but what bits exist and what they mean varies between architectures.

| Bit | Function |
|---|---|
| `_PAGE_PRESENT` | Page is resident in memory and not swapped out |
| `_PAGE_PROTNONE` | Page is resident but not accessable |
| `_PAGE_RW` | Set if the page may be written to |
| `_PAGE_USER` | Set if the page is accessible from user space |
| `_PAGE_DIRTY` | Set if the page is written to |
| `_PAGE_ACCESSED` | Set if the page is accessed |

Table 4.1: Page Table Entry Protection and Status Bits

These bits are self-explanatory except for the `_PAGE_PROTNONE` which we will discuss further. On the x86 with Pentium III and higher, this bit is called the *Page Attribute Table (PAT)*[1] and is used to indicate the size of the page the PTE is referencing. In a PGD entry, this same bit is the PSE bit so obviously these bits are meant to be used in conjunction.

As Linux does not use the PSE bit, 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 userspace 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 and so the kernel itself knows the PTE is present, just inaccessible to *userspace* which is a subtle, but important point. As the hardware bit `_PAGE_PRESENT` is clear, a page fault will occur if the page is accessed so Linux can enforce the protection while still knowing the page is resident if it needs to swap it out or the process exits.

## 4.3 Using Page Table Entries

Macros are defined in `asm/pgtable.h` which are important for the navigation and examination of page table entries. To navigate the page directories, three macros are provided which 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

[1] With earlier architectures such as the Pentium II, this bit was simply reserved.

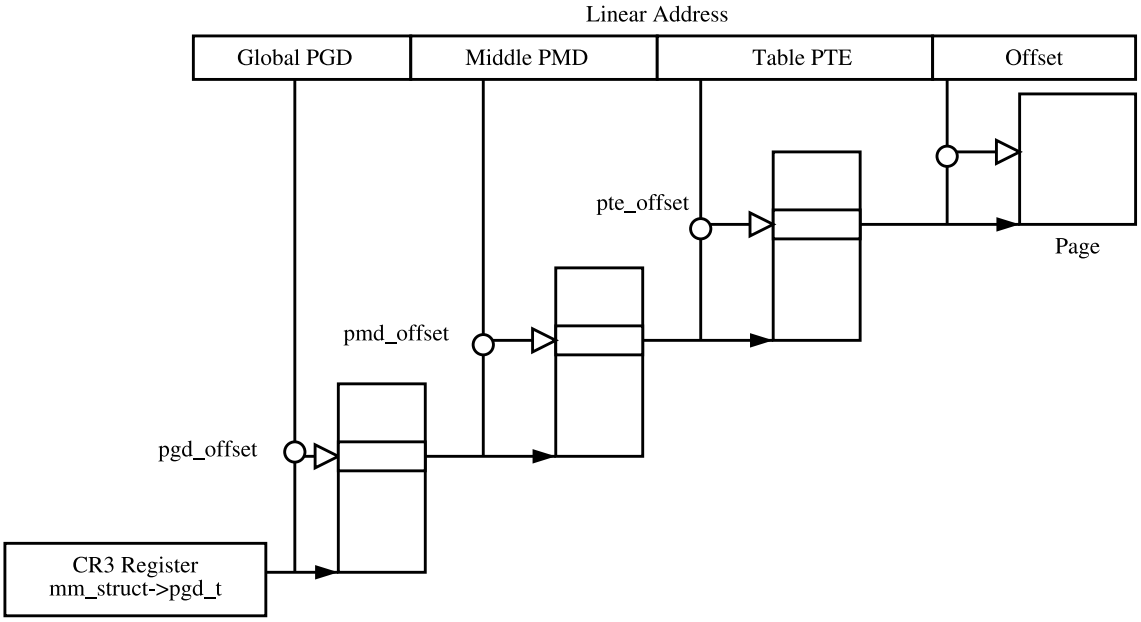


Figure 4.3: Page Table Layout

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 fields is illustrated in Figure 4.3

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 it returns 1 varies between the few architectures that define these macros but for those that actually define it, making sure the page entry is marked as present and accessed are the two most important checks.

There are many parts of the VM which are littered with page table walk code and it is important to recognise 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_mkread()` and cleared with `pte_rdprotect()`;
- 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 nothing 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 non-existent. 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()` macros 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.

## 4.4 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.

`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.

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

`set_pte()` takes a `pte_t` such as that returned by `mk_pte()` and places it within the processes page tables. `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.

## 4.5 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. As 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 LIFO type structure. Ordinarily, a page table entry contains points to other pages containing page tables or data. While cached, the first 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 7). 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 and so there is a mechanism 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.

## 4.6 Kernel Page Tables

When the system first starts, paging is not enabled as page tables do not magically initialise themselves. Each architecture implements this differently so only the x86 case will be discussed. The page table initialisation is divided into two phase. The bootstrap phase sets up page tables for just 8MiB so the paging unit can be enabled. The second phase initialises the rest of the page tables. We discuss both of these phases below.

### 4.6.1 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 vmlinuz is compiled with the base address at PAGE_OFFSET + 1MiB, the kernel is actually loaded beginning at the first megabyte (0x00100000) of memory[2]. 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 so before the paging unit is enabled, a page table mapping has to be established which translates the 8MiB of physical memory at the beginning of physical memory to the correct place after PAGE_OFFSET.

Initialisation begins with statically defining at compile time an array called swapper_pg_dir which is placed using linker directives at 0x00101000. It then establishes page table entries for 2 pages, pg0 and pg1. As the *Page Size Extension (PSE)* bit is set in the cr4 register, pages translated are 4MiB pages, not 4KiB as is the normal case. The first pointers to pg0 and pg1 are placed to cover the region 1-9MiB and 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.

Once 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.

[2]The first megabyte is used by some devices for communication with the BIOS and is skipped.

### 4.6.2 Finalising

The function responsible for finalising the page tables is called `paging_init()`. The call graph for this function on the x86 can be seen on Figure 4.4.

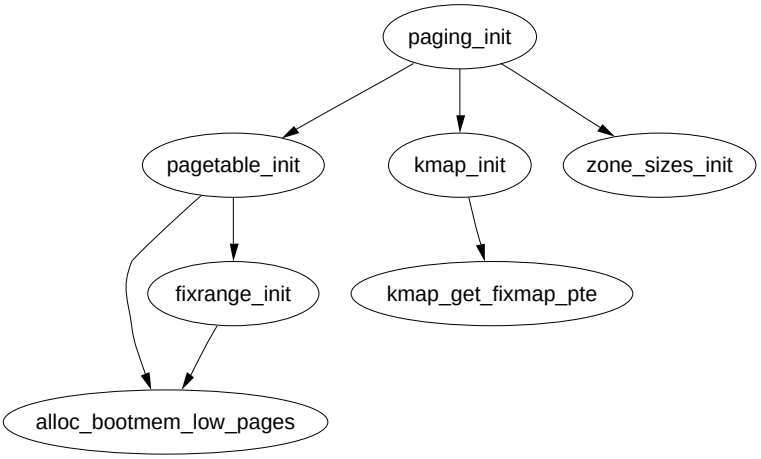


Figure 4.4: Call Graph: `paging_init()`

For each `pgd_t` used by the kernel, the boot memory allocator (see Chapter 6) is called to allocate a page for the PMD. Similarly, a page will be allocated for each `pmd_t` allocator. If the CPU has the PSE flag available, it will be set to enable extended paging. This means that each page table entry in the kernel paging tables will be 4MiB instead of 4KiB. If the CPU supports the PGE flag, it also will be set so that the page table entry will be global. Lastly, the page tables from `PKMAP_BASE` are set up with the function `fixrange_init()`. Once the page table has been fully setup, the statically defined page table at `swapper_pg_dir` is loaded again into the `cr3` register and the TLB is flushed.

## 4.7 Mapping addresses to `struct pages`

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 as the global array has pointers to all struct pages representing physical memory in the system. All architectures achieve this with similar mechanisms but for illustration purposes, we will only examine the x86 carefully. This section will first discuss how physical addresses are mapped to kernel virtual addresses and then what this means to the `mem_map` array.

### 4.7.1 Mapping Physical to Virtual Kernel Addresses

As we saw in Section 4.6, 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.

### 4.7.2 Mapping struct pages to Physical Addresses

As we saw in Section 4.6.1, 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 first available memory to use is located at 0xC0800000 but that is not the case. Linux tries to reserve the first 16MiB of memory for ZONE_DMA. This means the first virtual area used for kernel allocations is 0xC1000000 which the global mem_map is usually located. ZONE_DMA will be 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 a physical address PAGE_SHIFT bits to the right will treat it as a 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))
```

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 it right PAGE_SHIFT bits 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, it should be obvious to see how it could be calculated.

### 4.7.3 Initialising mem_map

The mem_map area is created during system startup in one of two fashions. On NUMA systems, the function free_area_init_node() is called for each active node in the system and on UMA systems, free_area_init() is used. Both use the core function free_area_init_core() to perform the actual task of allocating memory

for the mem_map portions and initialising the zones. Predictably, UMA calls the core function directly with contig_page_data and the global mem_map as parameters.

The core function free_area_init_core() allocates a local lmem_map for the node being initialised. The memory for the array is allocated from the boot memory allocator with alloc_bootmem_node() (see Chapter 6). 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 as only valid regions within it will be used by nodes.

# Chapter 5

# 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 we will discuss the process address space and how Linux manages it.

The kernel treats the userspace portion of the address space very differently to the kernel portion. For example, allocations for the kernel are satisfied immediately[1] and are visible globally no matter what process is on the CPU. 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[2], placed in the page table entry and marked writable.

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 Translation Lookaside Buffer (TLB)* switch is used as discussed later in Section 5.3. As a result of this, the kernel must be prepared to catch all exception and addressing errors raised from userspace. This is discussed in Section 5.5.

This chapter begins with how the linear address space is broken up and what the purpose of each section is. We then cover the structures maintained to describe each process, how they are allocated, initialised and then destroyed. Next, we will 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, we will cover how the kernel safely copies information to and from userspace.

[1] `vmalloc()` is partially an exception as a minor page fault may occur to update the process page tables, but the page will still be allocated immediately upon request.

[2] It is filled with zeros so that the new page will appear exactly the same as the global zero filled page to userspace

## 5.1 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 linear 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 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 5.1. The area up to `PAGE_OFFSET` is reserved for userspace and potentially changes with every context switch. In x86, this is defined as `0xC0000000` or at the 3GiB mark leaving the upper 1GiB of address space for the kernel.

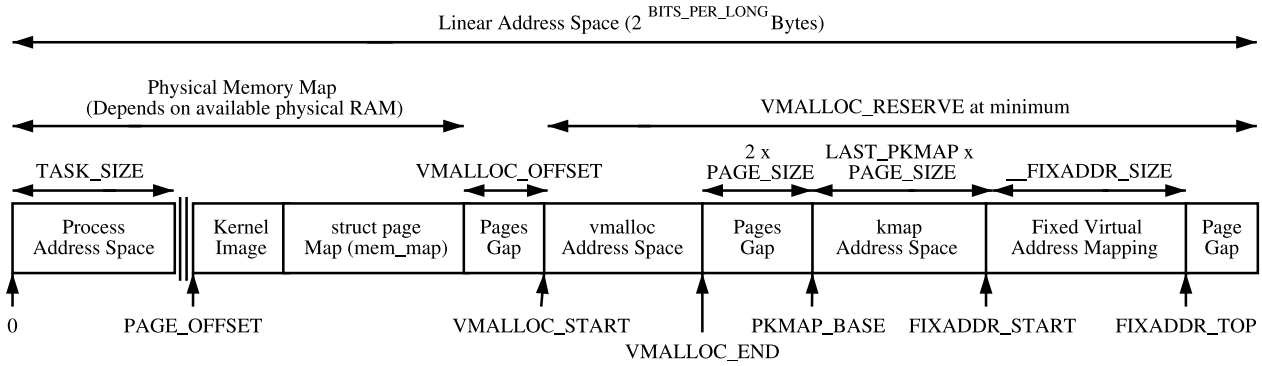


Figure 5.1: Kernel Address Space

8MiB (the amount of memory addressed by two PGDs[3]) is reserved at `PAGE_OFFSET` for loading the kernel image to run. It is placed here during kernel page tables initialisation as discussed in Section 4.6.1. Somewhere shortly after the image[4], the `mem_map` for UMA architectures, as discussed in Chapter 3, is stored. With NUMA architectures, portions of the virtual `mem_map` will be scattered throughout this region and 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 4.6, 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 non-contiguous memory

[3] 8MiB is simply a reasonable amount of space to reserve for the purposes of loading the kernel image

[4] Usually at the 16MiB mark to keep memory reserved for `ZONE_DMA`.

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 10. The second is for fixed virtual address mappings which extends 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 *Advanced Programmable Interrupt Controller (APIC)*[5]. 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 potion of the linear address space minus the minimum 128MiB that is reserved for the vmalloc region.

## 5.2 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 vmspace 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 filesystem specific information required to perform page-based operations on disk.

A number of system calls are provided which affect the address space and regions. These are listed in Table 5.1

[5]Further discussion on the APIC is beyond the scope of this document.

| System Call | Description |
|---|---|
| `fork()` | Creates a new process with a new address space. All the pages are marked COW and are shared between the two processes until a page fault occurs to make private copies |
| `clone()` | `clone()` allows a new process to be created that shares parts of its context with its parent and is how threading is implemented in Linux. `clone()` without the `CLONE_VM` set will create a new address space which is essentially the same as `fork()` |
| `mmap()` | mmap creates a new region within the process linear address space |
| `mremap()` | 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. |
| `munmap()` | This destroys part or all of a region. If the region been unmapped is in the middle of an existing region, the existing region is split into two separate regions |
| `shmat()` | This attaches a shared memory segment to a process address space |
| `shmdt()` | Removes a shared memory segment from an address space |
| `execve()` | This loads a new executable file replacing the current address space |
| `exit()` | Destroys an address space and all regions |

Table 5.1: System Calls Related to Memory Regions

## 5.3 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 threads. In fact, threads are identified in the task list by finding all `task_struct`s which have pointers to the same `mm_struct`.

A unique `mm_struct` is not needed for kernel threads as they will never page fault or access the userspace portion[6]. This results in the `task_struct`→`mm` field for kernel threads always being NULL. For some tasks such as the boot idle task, the `mm_struct` is never setup but for kernel threads, a call to `daemonize()` will call `exit_mm()` to decrement the usage counter.

As *Translation Lookaside Buffer (TLB)* flushes are extremely expensive, especially with architectures such as the PPC, a technique called *lazy TLB* is employed which avoids unnecessary TLB flushes by processes which do not access the user-

[6]The only exception is faulting in vmalloc space for updating the current page tables against the master page table which is treated as a special case by the page fault handling code.

space page tables[7]. 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 switches times.

When entering lazy TLB, the function `enter_lazy_tlb()` is called to ensure that a `mm_struct` is not shared between processors in 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 for this `mm_struct`, such as the page tables and file mappings. Threads and the `swap_out()` code for instance will increment this count making sure a `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` initialised 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 which 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:

[7]Remember that the kernel portion of the address space is always visible.

```
210 struct mm_struct {
211         struct vm_area_struct * mmap;
212         rb_root_t mm_rb;
213         struct vm_area_struct * mmap_cache;
214         pgd_t * pgd;
215         atomic_t mm_users;
216         atomic_t mm_count;
217         int map_count;
218         struct rw_semaphore mmap_sem;
219         spinlock_t page_table_lock;
220
221         struct list_head mmlist;
222
226         unsigned long start_code, end_code, start_data, end_data;
227         unsigned long start_brk, brk, start_stack;
228         unsigned long arg_start, arg_end, env_start, env_end;
229         unsigned long rss, total_vm, locked_vm;
230         unsigned long def_flags;
231         unsigned long cpu_vm_mask;
232         unsigned long swap_address;
233
234         unsigned dumpable:1;
235
236         /* Architecture-specific MM context */
237         mm_context_t context;
238 };
```

The meaning of each of the field 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 Page Global Directory 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` starting 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 which protects the VMA list for readers and writers. As 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 RSS (see below) count and the VMA from modification;

**mmlist** All `mm_struct`s are linked together via 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** *Resident Set Size (RSS)* is the number of resident pages for this process;

**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 or not;

**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 flag 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 5.2.

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

Table 5.2: Functions related to memory region descriptors

### 5.3.1 Allocating a Descriptor

Two functions are provided to allocate a mm_struct. To be slightly confusing, they are essentially the same but with small important differences. allocate_mm() is just a preprocessor macro which allocates a mm_struct from the *slab allocator* (see Chapter 9). mm_alloc() allocates from slab and then calls mm_init() to initialise it.

### 5.3.2 Initialising a Descriptor

The initial mm_struct in the system is called init_mm() and is statically initialised at compile time using the macro INIT_MM().

```
242 #define INIT_MM(name) \
243 {                                                       \
244         mm_rb:          RB_ROOT,                        \
245         pgd:            swapper_pg_dir,                 \
246         mm_users:       ATOMIC_INIT(2),                 \
247         mm_count:       ATOMIC_INIT(1),                 \
248         mmap_sem:       __RWSEM_INITIALIZER(name.mmap_sem), \
249         page_table_lock: SPIN_LOCK_UNLOCKED,            \
250         mmlist:         LIST_HEAD_INIT(name.mmlist),    \
251 }
```

Once it is established, new mm_structs 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 initialise process specific fields.

### 5.3.3 Destroying a Descriptor

While a new user increments the usage count with atomic_inc(&mm->mm_users), it is decremented with a call to mmput(). If the mm_users count reaches zero, all

the mapped regions are destroyed with `exit_mmap()` and the page tables destroyed as there is no longer any users of the userspace portions. The `mm_count` count is decremented with `mmdrop()` as 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.

## 5.4 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 overlap and represent 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 a process has may be viewed via 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 5.2. At the top, there is the `vm_area_struct` which on its own is enough to represent anonymous memory.

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

**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 via this field;

**vm_page_prot** The protection flags that are set for each PTE in this VMA. The different bits are described in Table 4.1;

**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 5.3

**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;

**vm_next_share** Shared VMA regions based on file mappings (such as shared libraries) linked together with this field;

**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_pgoff** This is the page aligned offset within a file that is memory mapped;

**vm_file** The struct file pointer to the file being mapped;

**vm_raend** This is 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 field determines how many additional pages are faulted in;

**vm_private_data** Used by some device drivers to store private information. Not of concern to the memory manager.

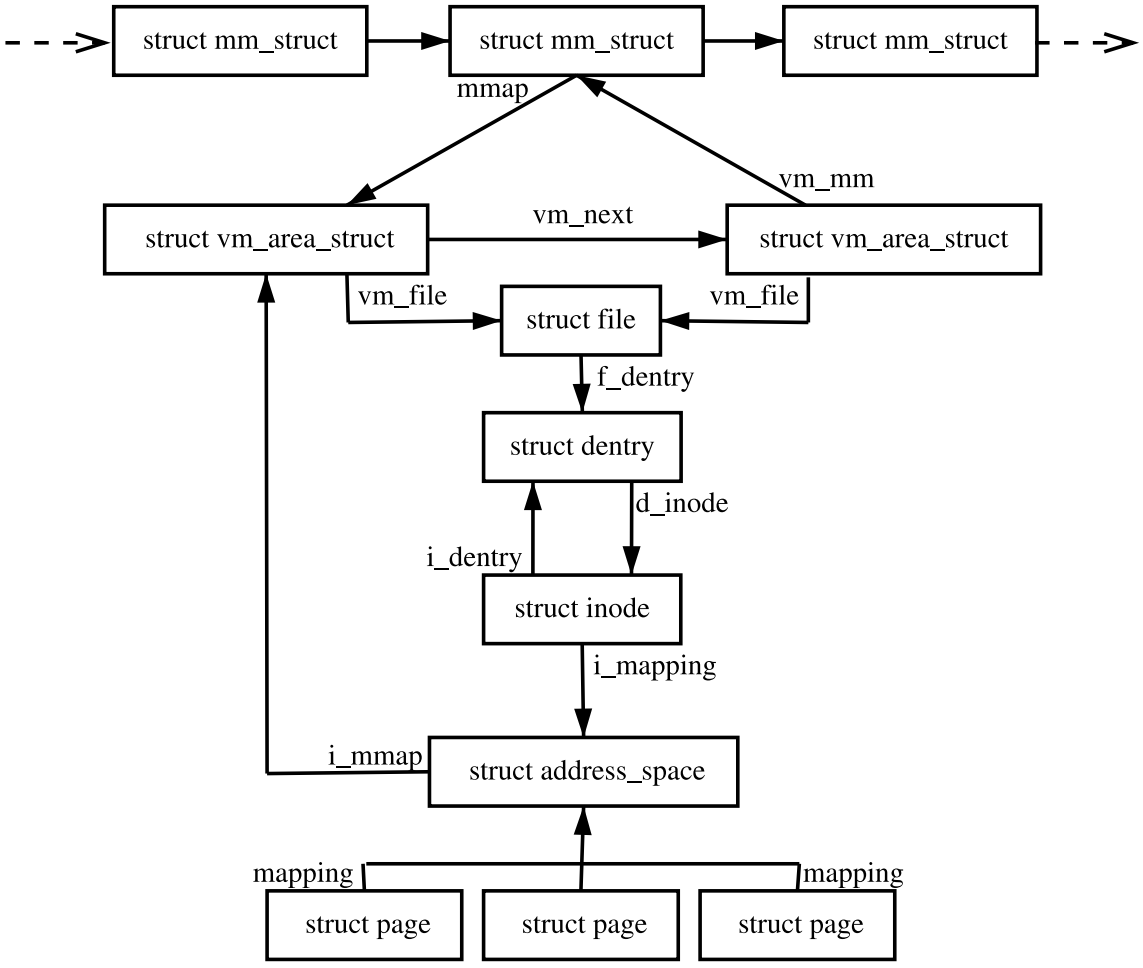


Figure 5.2: Data Structures related to the Address Space

| **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 being resized. Flag is unused |
| **mmap Related Flags** | |
| VM_MAYREAD | Allow the VM_READ flag to be set |
| VM_MAYWRITE | Allow the VM_WRITE flag to be set |
| VM_MAYEXEC | Allow the VM_EXEC flag to be set |
| VM_MAYSHARE | Allow the VM_SHARE flag to be set |
| VM_GROWSDOWN | Shared segment (probably stack) is allowed to grow down |
| VM_GROWSUP | Shared segment (probably heap) is allowed to grow up |
| VM_SHM | Pages are used by shared SHM memory segment |
| VM_DENYWRITE | What MAP_DENYWRITE during mmap translates to. Now unused |
| VM_EXECUTABLE | What MAP_EXECUTABLE during mmap translates to. Now unused |
| **Locking Flags** | |
| VM_LOCKED | If set, the pages will not be swapped out. Set by mlock() |
| VM_IO | Signals that the area is a mmaped region for IO to a device. It will also prevent the region being core dumped |
| VM_RESERVED | Do not swap out this region, used by device drivers |
| **madvise() Flags** | |
| VM_SEQ_READ | A hint stating that pages will be accessed sequentially |
| VM_RAND_READ | A hint stating that readahead in the region is useless |

Figure 5.3: Memory Region Flags

All the regions are linked together on a linked list ordered by address via 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 as 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.

### 5.4.1 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 5.2. The struct contains information of relevance to the filesystem such as the number of dirty pages which must be flushed to disk. It is declared as follows in <`linux/fs.h`>:

```
401 struct address_space {
402         struct list_head        clean_pages;
403         struct list_head        dirty_pages;
404         struct list_head        locked_pages;
405         unsigned long           nrpages;
406         struct address_space_operations *a_ops;
407         struct inode            *host;
408         struct vm_area_struct   *i_mmap;
409         struct vm_area_struct   *i_mmap_shared;
410         spinlock_t              i_shared_lock;
411         int                     gfp_mask;
412 };
```

A brief description of each field is as follows:

**clean_pages** A list of clean pages which do not have to be synchronised with the disk;

**dirty_pages** Pages that the process has touched and need to by sync-ed with the backing storage;

**locked_pages** The list of pages locked in memory;

**nrpages** Number of resident pages in use by the address space;

**a_ops** A struct of function pointers within the filesystem;

**host** The host inode the file belongs to;

**i_mmap** A pointer to the VMA the address space is part of;

**i_mmap_shared** A pointer to the next VMA which shares this address space;

**i_shared_lock** A spinlock to protect this structure;

**gfp_mask** 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 doesn't know and doesn't care how information is written to disk, so the `a_ops` struct is used to call the relevant functions. It is defined as follows in <`linux/fs.h`>:

```
383 struct address_space_operations {
384         int (*writepage)(struct page *);
385         int (*readpage)(struct file *, struct page *);
386         int (*sync_page)(struct page *);
387         /*
388          * ext3 requires that a successful prepare_write()
             * call be followed
389          * by a commit_write() call - they must be balanced
390          */
391         int (*prepare_write)(struct file *, struct page *,
                                unsigned, unsigned);
392         int (*commit_write)(struct file *, struct page *,
                                unsigned, unsigned);
393         /* Unfortunately this kludge is needed for FIBMAP.
             * Don't use it */
394         int (*bmap)(struct address_space *, long);
395         int (*flushpage) (struct page *, unsigned long);
396         int (*releasepage) (struct page *, int);
397 #define KERNEL_HAS_O_DIRECT
398         int (*direct_IO)(int, struct inode *, struct kiobuf *,
                            unsigned long, int);
399 };
```

These fields are all function pointers which are described as follows;

**writepage** Write a page to disk. The offset within the file to write to is stored within the page struct. It is up to the filesystem specific code to find the block. See `buffer.c:block_write_full_page()`;

**readpage** Read a page from disk. See `buffer.c:block_read_full_page()`;

**sync_page** Sync 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 filesystem, this ensures the filesystem log is up to date. With normal filesystems, 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 IO can be performed. Only of concern to the filesystem specific code;

**flushpage** This makes sure there is no IO pending on a page before releasing it. See `buffer.c:discard_bh_page()`;

**releasepage** This tries to flush all the buffers associated with a page before freeing the page itself. See `try_to_free_buffers()`.

| Function | Description |
|---|---|
| `find_vma()` | Finds the VMA that covers a given address. If the region does not exist, it returns the VMA closest to the requested address |
| `find_vma_prev()` | Same as `find_vma()` except it also also gives the VMA pointing to the returned VMA. It is not often used, with `sys_mprotect()` being the notable exception, as it is usually `find_vma_prepare()` that is required |
| `find_vma_prepare()` | Same as `find_vma()` except that it will return the VMA pointing to the returned VMA as well as the red-black tree nodes needed to perform an insertion into the tree |
| `find_vma_intersection()` | Returns the VMA which intersects a given address range. Useful when checking if a linear address region is in use by any VMA |
| `vma_merge()` | Attempts to expand the supplied VMA to cover a new address range. If the VMA can not be expanded forwards, the next VMA is checked to see if it may be expanded backwards to cover the address range instead. Regions may be merged if there is no file/device mapping and the permissions match |
| `get_unmapped_area()` | Returns the address of a free region of memory large enough to cover the requested size of memory. Used principally when a new VMA is to be created |
| `insert_vm_struct()` | Inserts a new VMA into a linear address space |

Table 5.3: Memory Region VMA API

### 5.4.2 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 principle function for creating new areas for all architectures.

`do_mmap2()` first clears the `MAP_DENYWRITE` and `MAP_EXECUTABLE` bits from the `flags` parameter as they are ignored by Linux, which is confirmed by the `mmap()` manual page. If a file is being mapped, `do_mmap2()` will look up the `struct file` based on the file descriptor passed as a parameter and 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 the appropriate filesystem 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:

- Sanity check the parameters;
- Find a free linear address space large enough for the memory mapping. If a filesystem or device specific `get_unmapped_area()` function is provided, it will be used otherwise `arch_get_unmapped_area()` is called;
- Calculate the VM flags and check them against the file access permissions;
- If an old area exists where the mapping is to take place, fix it up so that it is suitable for the new mapping;
- Allocate a `vm_area_struct` from the slab allocator and fill in its entries;
- Link in the new VMA;
- Call the filesystem or device specific mmap function;
- Update statistics and exit.

### 5.4.3 Finding a Mapped Memory Region

A common operation is to find the VMA 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 5.3.

It first checks the `mmap_cache` field which caches the result of the last call to

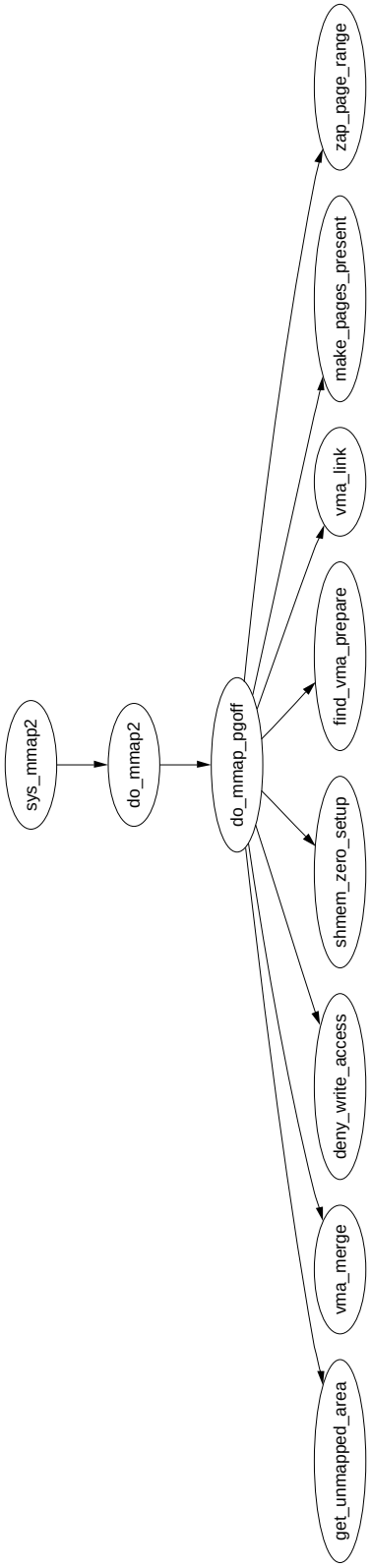


Figure 5.4: Call Graph: `sys_mmap2()`

`find_vma()` as 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` field 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[8] which is required as the list is a singly linked list. This 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 which 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.

### 5.4.4 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 5.5 indicates, there is little work involved with finding an unmapped area. The function is passed a number of parameters. A `struct file` is passed representing the file or device to be mapped as well as `pgoff` which is the offset within the file that is been mapped. The requested `address` for the mapping is passed as well as its `length`. The last parameter is the protection `flags` for the area.

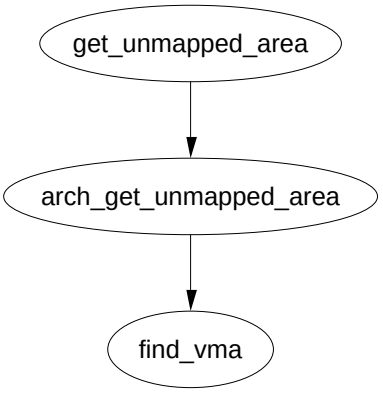


Figure 5.5: Call Graph: `get_unmapped_area()`

If a device is being mapped, such as a video card, the associated

[8]The VMA list is one of the very rare cases where a singly linked list is used in the kernel.

`f_op→get_unmapped_area()` is used. This is because devices or files may have additional requirements for mapping that generic code can not 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, there is a generic function provided in `mm/mmap.c`.

### 5.4.5 Inserting a memory region

The principal function for inserting a new memory region is `insert_vm_struct()` whose call graph can be seen in Figure 5.6. It is a very simple function which first calls `find_vma_prepare()` to find the appropriate VMAs the new region is to be inserted between and the correct nodes within the red-black tree. It then calls `__vma_link()` to do the work of linking in the new VMA.

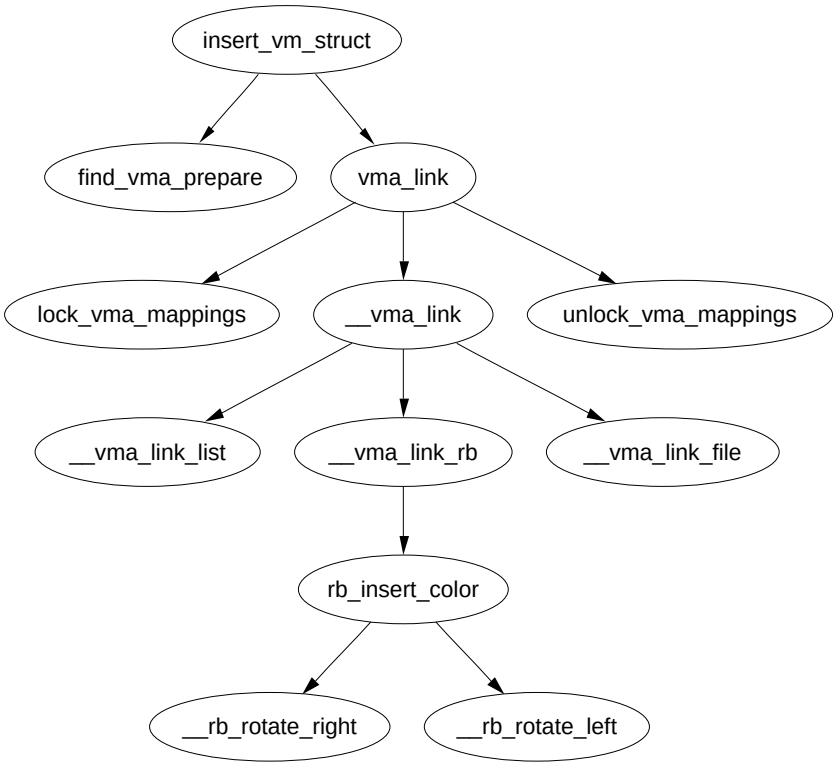


Figure 5.6: Call Graph: `insert_vm_struct()`

The function `insert_vm_struct()` is rarely used as it does not increase the `map_count` field. Instead, the function commonly used is `__insert_vm_struct()` which performs the same tasks except that it increments `map_count`.

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 file if the VMA is a file mapping before calling `__vma_link()` which places the VMA in the relevant lists.

It is important to note that 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 which are contained in three separate functions. `__vma_link_list()` inserts the VMA into the linear, singly linked list. If it is the first 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 final stage is fixing up the file share mapping with `__vma_link_file()` which basically inserts the VMA into the linked list of VMAs via the `vm_pprev_share()` and `vm_next_share()` fields.

### 5.4.6 Merging contiguous regions

Linux used to have a function called `merge_segments()` [Haca] which was responsible for merging adjacent regions of memory together if the file and permissions matched. The objective was to remove the number of VMAs required, especially as many operations resulted in a number of mappings been created such as calls to `sys_mprotect()`. This was an expensive operation as 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 which exists now is called `vma_merge()` and it is only used in two places. The first is user is `sys_mmap()` which calls it if an anonymous region is being mapped, as anonymous regions are frequently mergable. 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 negating the need to create a new region. A region may be expanded if there are no file 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 first is during a call to `sys_mprotect()` during the fixup 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.

### 5.4.7 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 `MREMAP_FIXED` is not specified in the flags. The call graph is illustrated in Figure 5.7.

If a region is to be moved, `do_mremap()` first calls `get_unmapped_area()` to find a region large enough to contain the new resized mapping and then calls `move_vma()` to move the old VMA to the new location. See Figure 5.8 for the call graph to `move_vma()`.

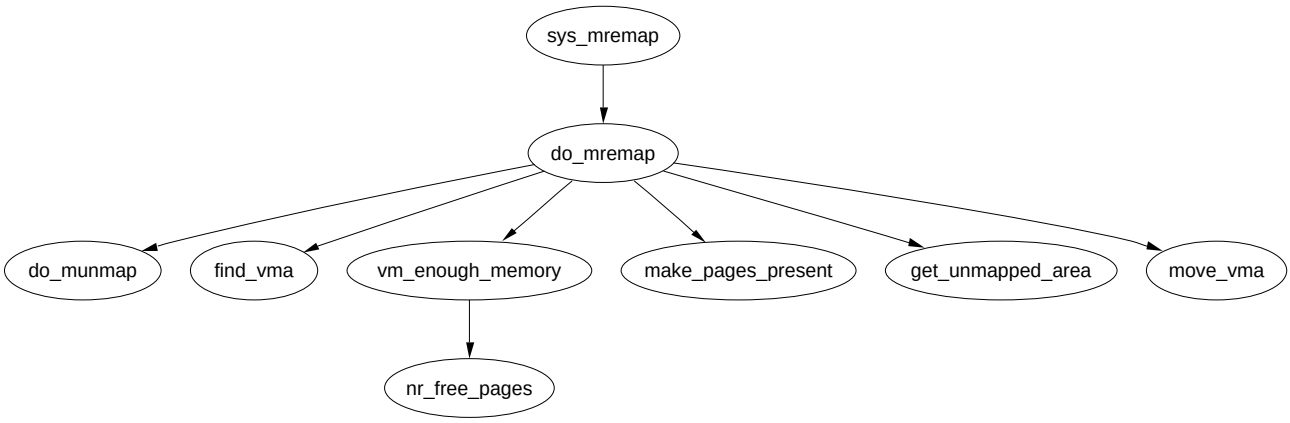


Figure 5.7: Call Graph: `sys_mremap()`

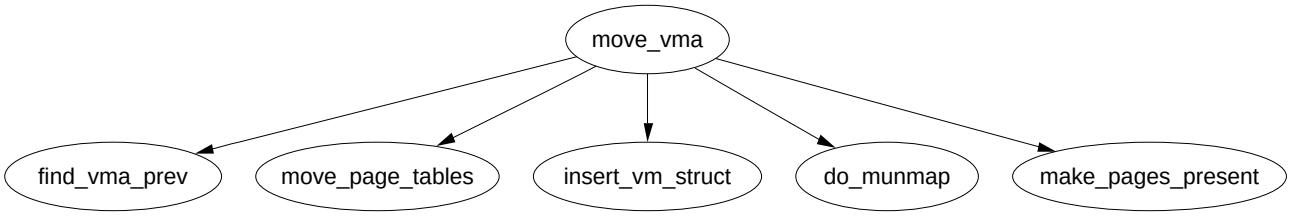


Figure 5.8: Call Graph: `move_vma()`

First `move_vma()` checks if the new location may be merged with the VMAs adjacent to the new location. If they can not be merged, a new VMA is allocated literally one PTE at a time. Next `move_page_tables()` is called(see Figure 5.9 for its call graph) which copies all the page table entries from the old mapping to the new one. While there may be better ways to move the page tables, this method makes error recovery trivial as backtracking is relatively straight forward.

The contents of the pages are not copied. Instead, `zap_page_range()` is called to swap out or remove all the pages from the old mapping and the normal page fault handling code will swap the pages back in from backing storage or from files or will call the device specific `do_nopage()` function.

### 5.4.8 Locking a Memory Region

Linux can lock pages from an address range into memory via the system call `mlock()` which is implemented by `sys_mlock()` whose call graph is shown in Figure 5.10. At a high level, the function is simple; it creates a VMA for the address range to be locked, sets the `VM_LOCKED` flag on it and forces all the pages to be present with `make_pages_present()`. A second system call `mlockall()` which maps to `sys_mlockall()` is also provided which is a simple extension to do the same work as `sys_mlock()` except for every VMA on the calling process. Both functions rely on the core function `do_mlock()` to perform the real work of finding the affected

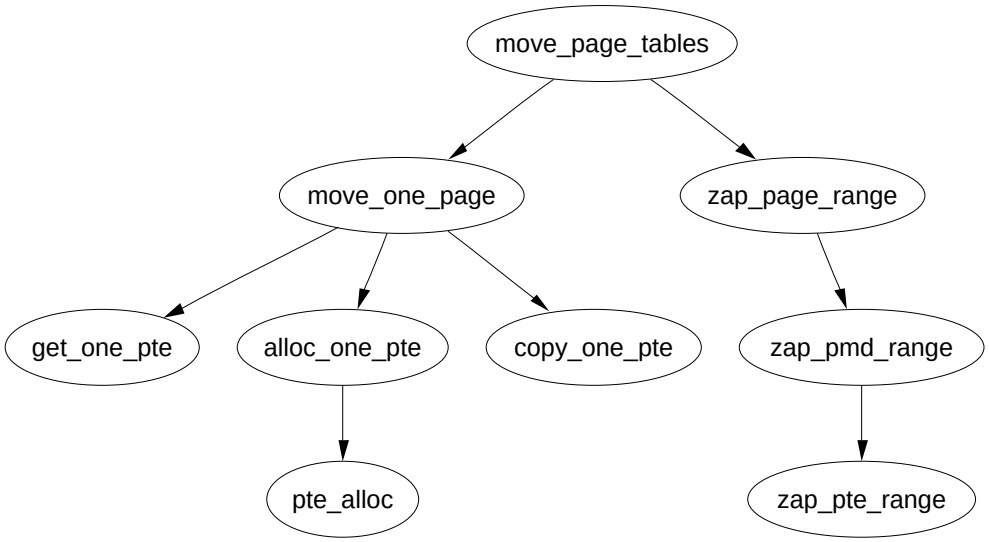


Figure 5.9: Call Graph: `move_page_tables()`

VMAs and deciding what function is needed to fix up the regions as described later.

There are some limitations to what memory may be locked. The address range must be page aligned as 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 non-functional as there is nothing to stop a process forking a number of times and each child locking a portion but as 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.

### 5.4.9 Unlocking the region

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

### 5.4.10 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

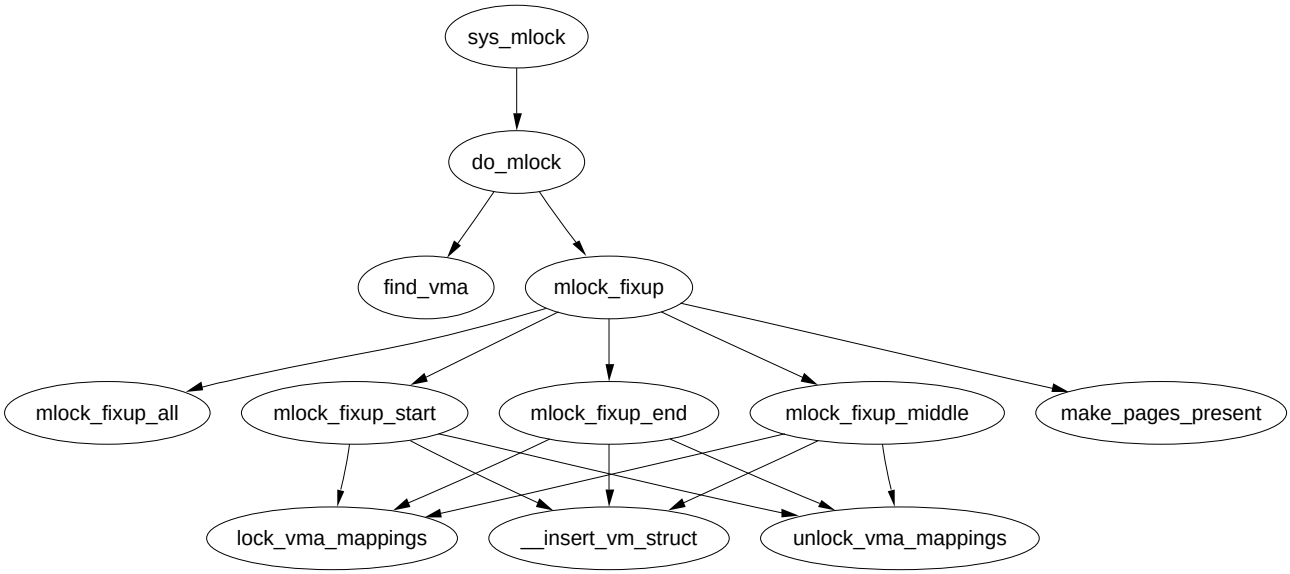


Figure 5.10: Call Graph: `sys_mlock()`

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 which 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.

### 5.4.11 Deleting a memory region

The function responsible for deleting memory regions or parts thereof is `do_munmap()`. It is a relatively simple operation in comparison to the other memory region related operations and is basically divided up into three parts. The first is to fix up 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 fix up 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 fixup.

Next the linked list VMAs on `free` is 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 file mapping. Again, if this is a partial unmapping, it will be recreated during fixup. `zap_page_range()` is called to remove all the pages associated with the region about

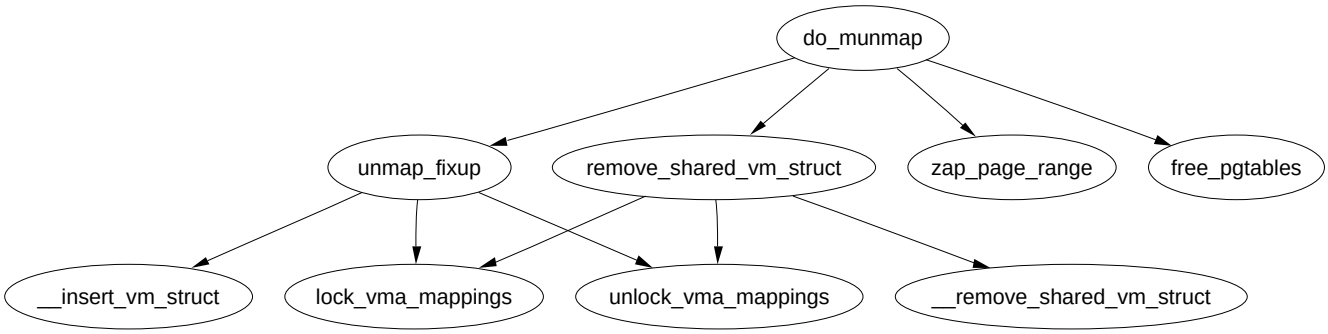


Figure 5.11: Call Graph: `do_munmap()`

to be unmapped before `unmap_fixup()` is called to handle partial unmappings.

Lastly `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 finer grained freeing of page table entries would be too expensive to free up data structures that are both small and likely to be used again.

### 5.4.12 Deleting all memory regions

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

## 5.5 Exception Handling

A very important part of VM is how exceptions related to bad kernel address references are caught[9] which are not a result of a kernel bug[10]. This section does *not* cover the exceptions that are raised with errors such as divide by zero, we are 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 via a system call which the kernel must be able to safely trap as 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.

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

[10]Of course bad references due to kernel bugs should rightfully cause the system to have a minor fit.

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 fixup routine. When an exception occurs that the page fault handler cannot manage, it calls `search_exception_table()` to see if a fixup routine has been provided for an error at the faulting instruction. If module support is compiled, each modules exception table will also be searched.

If the address of the current exception is found in the table, the corresponding location of the fixup code is returned and executed. We will see in Section 5.7 how this is used to trap bad reads and writes to userspace.

## 5.6 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 as the space is just reserved with the `vm_area_struct`. Other examples of non-resident 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 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 `swapin_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 making 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, else 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→maj_flt` and `task_struct→min_flt` fields respectively.

The page fault handler in Linux is expected to recognise and act on a number of different types of page faults listed in Table 5.4 which will be discussed in detail later in this chapter.

Each architecture registers an architecture-specific function for the handling of page faults. While the name of this function is arbitrary, a common choice is `do_page_fault()` whose call graph for the x86 is shown in Figure 5.12.

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

| 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 | Remove the page from the swap cache and allocate it to the process |
| Page swapped out to backing storage | Major | Find where the page with information 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 `SEGSEGV` 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 `init_mm`. 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 which is treated quite severely. |

Table 5.4: Reasons For Page Faulting

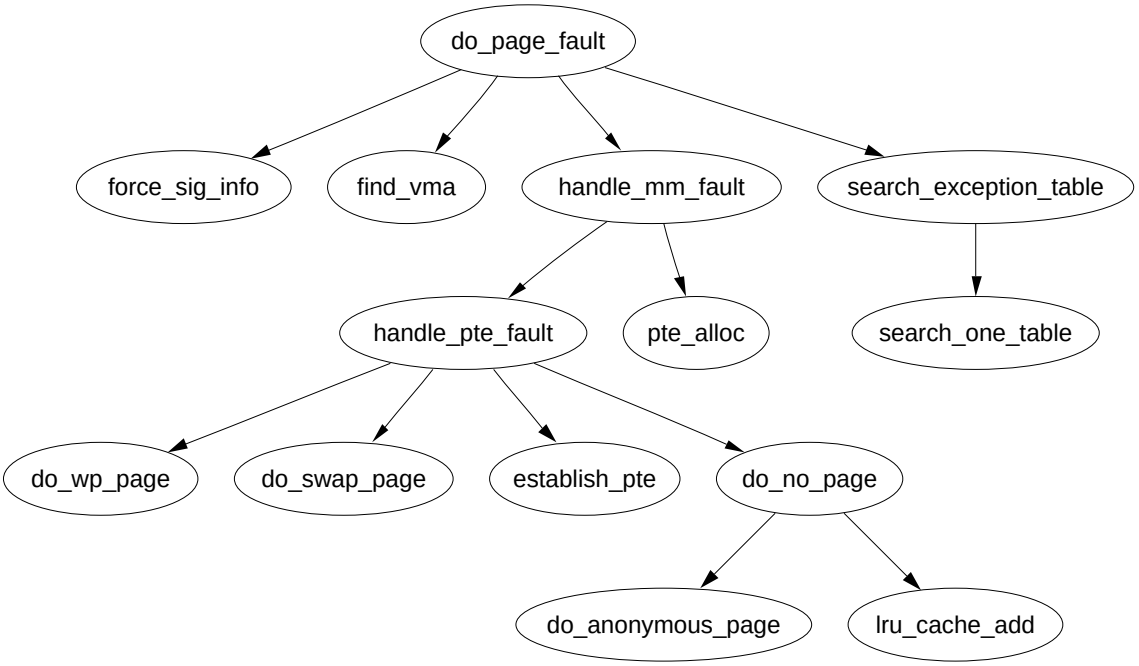


Figure 5.12: Call Graph: `do_page_fault()`

be handled by the architecture-independent code. The flow chart, in Figure 5.17, shows broadly speaking what this function does. In the figure, identifiers with a colon after them corresponds 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 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.

### 5.6.1 Handling a Page Fault

Once the exception handler has decided the fault is a valid page fault in a valid memory region, the architecture-independent function `handle_mm_fault()`, whose call graph is shown in Figure 5.13, 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 5.13 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*.

The second option is if the page is being written to. If the PTE is write protected, then `do_wp_page()` is called as the page is a Copy-On-Write (COW) page. A COW page is one which is shared between multiple processes(usually a parent and child)

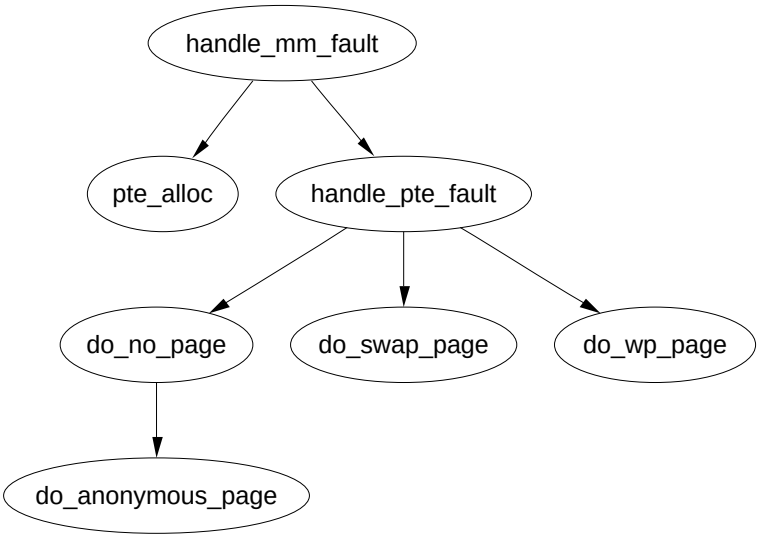


Figure 5.13: Call Graph: `handle_mm_fault()`

until a write occurs after which a private copy is made for the writing process. A COW page is recognised because the VMA for the region is marked writable even though the individual PTE is not. If it is not a COW page, the page is simply marked dirty as it has been written to.

The last option is if the page has been read and is present but a fault still occurred. This can occur with some architectures that do not have a three level page table. In this case, the PTE is simply established and marked young.

### 5.6.2 Demand Allocation

When a process accesses a page for the very first time, the page has to be allocated and possibly filled with data by the `do_no_page()` function. If the parent VMA provided a `vm_ops` struct with a `nopage()` function, it is called. This is of importance to a memory mapped device such as a video card which needs to allocate the page and supply data on access or to a mapped file which must retrieve its data from backing storage. We will first discuss the case where the faulting page is anonymous as this is the simpliest case.

**Handling anonymous pages** If `vm_area_struct→vm_ops` field is not filled or a `nopage()` function is not supplied, the function `do_anonymous_page()` is called to handle an anonymous access. There are only two cases to handle, first time read and first time write. As it is an anonymous page, the first read is an easy case as no data exists so the system-wide `empty_zero_page` which is just a page of zeros[11] is mapped for the PTE and the PTE is write protected. The PTE is write protected so that another page fault will occur if the process writes to the page.

[11] On the x86, it is zerod out in the function `mem_init()`.

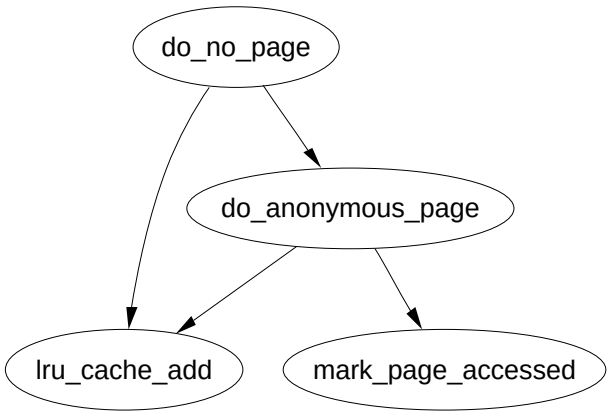


Figure 5.14: Call Graph: `do_no_page()`

If this is the first write to the page `alloc_page()` is called to allocate a free page (see Chapter 7) and is zero filled by `clear_user_highpage()`. Assuming the page was successfully allocated, the *Resident Set Size (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 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. In the file backed case the function `filemap_nopage()` is the `nopage()` function for allocating a page and reading a page-sized amount of data from disk. Each device driver provides a different `nopage()` whose 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 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 there is not a PTE 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 as 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 updated and the architecture hooks for cache coherency called.

### 5.6.3 Demand Paging

When a page is swapped out to backing storage, the function `do_swap_page()` is responsible for reading the page back in. 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. As pages may be shared between multiple processes, they can not always be swapped out immediately. Instead, when a page is swapped out, it is placed within the swap cache.

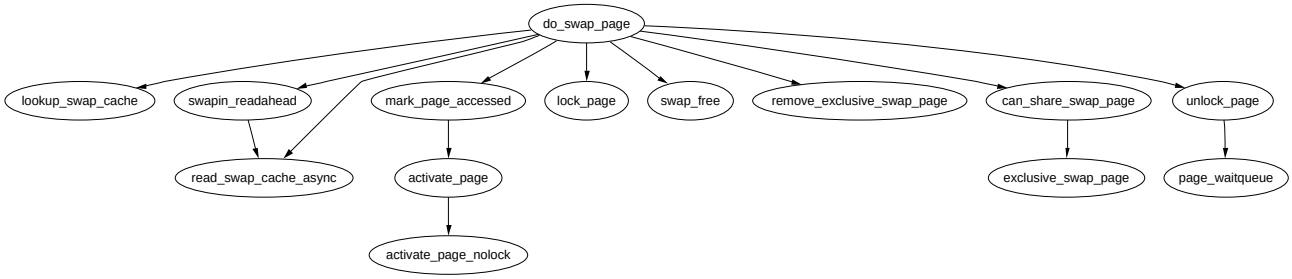


Figure 5.15: Call Graph: `do_swap_page()`

A shared page can not 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)*. With RMAP, the PTEs a page is mapped by are linked together by a chain so they can be reverse looked up.

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 `swapin_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 initialised as 2 or 3 otherwise. 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 once the seek has completed, the succeeding pages should also be read in.

### 5.6.4 Copy On Write (COW) Pages

Traditionally when a process forked, the parent address space was copied to duplicate it for the child. This was an extremely expensive operation as 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 *Copy-On-Write (COW)* is employed.

During fork, the PTEs of the two processes are made read-only so that when a write occurs there will be a page fault. Linux recognises a COW page because

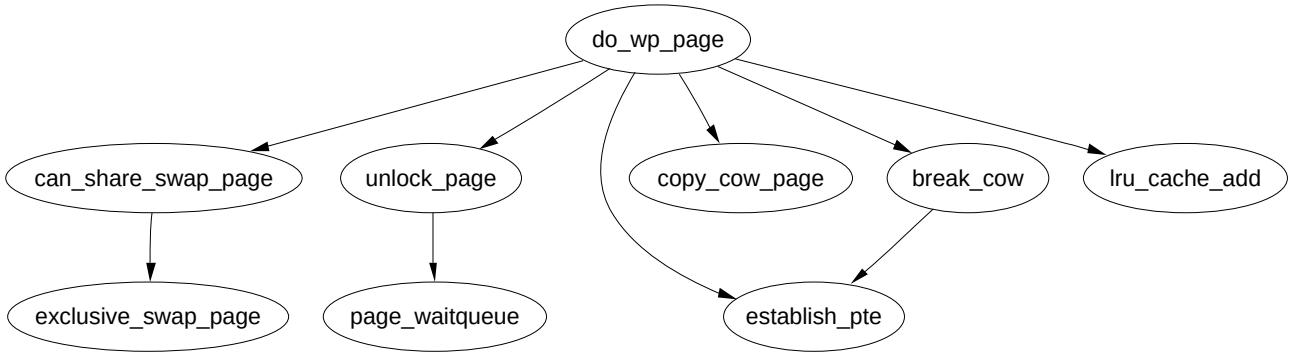


Figure 5.16: Call Graph: `do_wp_page()`

even though the PTE is write protected, the controlling VMA shows the region is writable. It uses the function `do_wp_page()` 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.

## 5.7 Copying To/From Userspace

It is not safe to access memory in the process address space directly as 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, assembler is provided by the `__copy_user()` to trap exceptions where the address is totally useless. The location of the fixup 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 5.5.

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

`copy_from_user()` calls either `__constant_copy_from_user()` or `__generic_copy_from_us` depending on whether the size of the copy is known at compile time or not. If the size is known at compile time, there are different assembler optimisations 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 "fixup" code and the third part is the `__ex_table` mapping the instructions from the first part to the fixup code in the second part.

`copy_from_user(void *to, const void *from, unsigned long n)`
Copies n bytes from the user address space (`from`) to the kernel address space (`to`)

`copy_to_user(void *to, const void *from, unsigned long n)`
Copies n bytes from the kernel address space (`from`) to the user address space (`to`)

`get_user(void *to, void *from)`
Copies an integer value from userspace (`from`) to kernel space (`to`)

`put_user(void *from, void *to)`
Copies an integer value from kernel space (`from`) to userspace (`to`)

`strncpy_from_user(char *dst, const char *src, long count)`
Copies a null terminated string of at most `count` bytes long from userspace (`src`) to kernel space (`dst`)

`strlen_user(const char *s, long n)`
Returns the length, upper bound by `n`, of the userspace string including the terminating NULL

Table 5.5: Accessing Process Address Space API

These pairings, as described in Section 5.5, copy the location of the copy instructions and the location of the fixup code 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()` and find the EIP where the faulty read took place and jump to the fixup code which copies zeros into the remaining kernel space, fixes up registers and returns. In this manner, the kernel can safely access userspace with no expensive checks and letting the MMU hardware handle the exceptions.

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

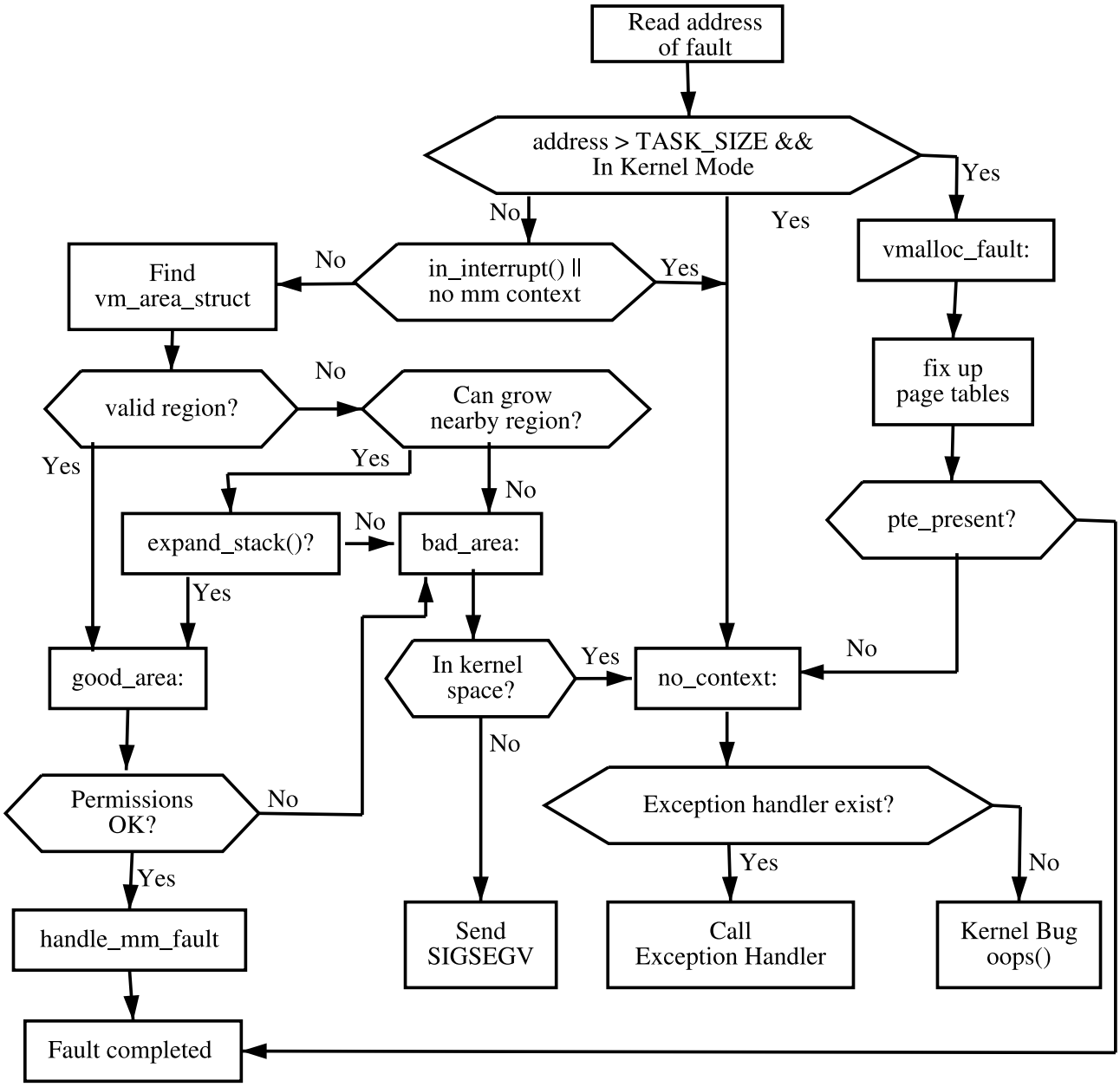


Figure 5.17: do_page_fault Flow Diagram

# Chapter 6

# Boot Memory Allocator

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

To address this, a specialised 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 0 if unallocated. To satisfy allocations of sizes smaller than a page, the allocator records the *Page Frame Number (PFN)* of the last allocation and the offset 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 linearly searched to satisfy an allocation. As this is examining bitmaps, it gets very expensive, especially as the first fit algorithm tends to leave many small free blocks at the beginning of physical memory which 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 6.1 and the other is for NUMA, listed in Table 6.2. The principle difference is that the NUMA API must be supplied with the node affected by the operation but as the callers of these APIs exist in the architecture dependant layer, it is not a significant problem.

This chapter will begin with a description of the structure the allocator uses to describe the physical memory available for each node. We will 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 initialised the boot memory allocator structures. The allocation and free routines will then be discussed before finally talking about how the boot memory allocator is retired.

## 6.1 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 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. 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.

## 6.2 Initialising 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 initialise the boot memory allocator.

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

**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.

### 6.2.1 Calculating The Size of Zones

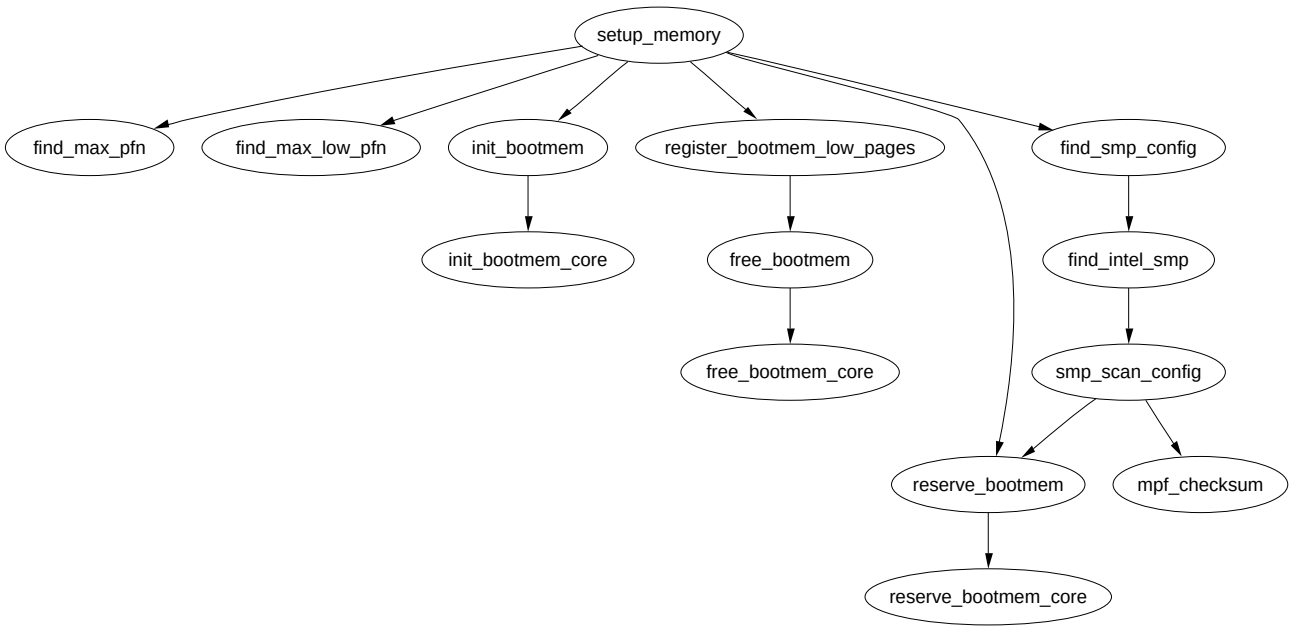


Figure 6.1: Call Graph: setup_memory()

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 first page after _end which is the end of the loaded kernel image. The value is stored as a file 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*[1] map for the highest page frame. The value is also stored as a file scope variable in mm/bootmem.c.

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

[1] e820 is a table provided by the BIOS describing what physical memory is available, reserved or non-existent.

mm/bootmem.c. Note that 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 file scope variables in arch/i386/init.c as highstart_pfn and highend_pfn. The values are used later to initialise the high memory pages for the physical page allocator as we will see in Section 6.5.

### 6.2.2 Initialising bootmem_data

Once the limits of usable physical memory are known, one of two boot memory initialisation functions are selected and provided with the start and end PFN for the node to be initialised. init_bootmem(), which initialises contig_page_data, is used by UMA architectures, while init_bootmem_node() is for NUMA to initialise a specified node. Both function are trivial and rely on init_bootmem_core() to do the real work.

The first task of the core function is to insert this pgdat_data_t into the pgdat_list as at the end of this function, the node is ready for use. It then records the starting and end address for this node in its associated bootmem_data_t and allocates the bitmap representing page allocations. The size in bytes[2] of the bitmap required is straightforward:

$$\text{mapsize} = \frac{(\text{end\_pfn} - \text{start\_pfn}) + 7}{8}$$

The bitmap in stored at the physical address pointed to by bootmem_data_t→node_boot_start and the virtual address to the map is placed in bootmem_data_t→node_bootmem_map. As there is no architecture independent way to detect "holes" in memory, the entire bitmap is initialised to 1, effectively marking all pages allocated. It is up to the architecture dependent code to set the bits of usable pages to 0. In the case of the x86, the function register_bootmem_low_pages() reads through the e820 map and calls free_bootmem() for each usable page to set the bit to 0 before using reserve_bootmem() to reserve the pages needed by the actual bitmap.

## 6.3 Allocating Memory

The reserve_bootmem() function may be used to reserve pages for use by the caller but is very cumbersome to use for general allocations. There are four functions provided for easy allocations on UMA architectures called alloc_bootmem(), alloc_bootmem_low(), alloc_bootmem_pages() and alloc_bootmem_low_pages() which are fully described in Table 6.1. All of these macros call __alloc_bootmem() with different parameters. See the call graph in Figure 6.2.

[2]Hence the division by 8.

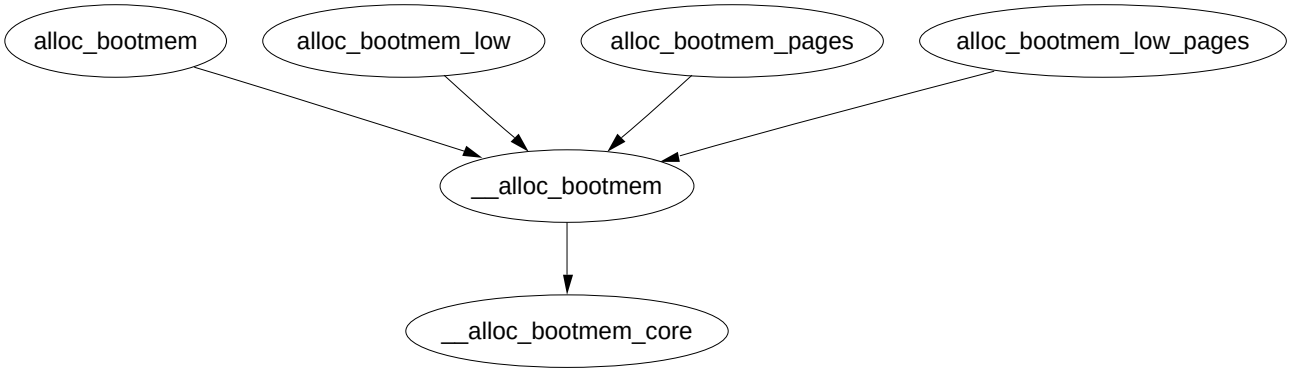


Figure 6.2: Call Graph: `__alloc_bootmem()`

Similar functions exist for NUMA which take the node as an additional parameter, as listed in Table 6.2. 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 either `__alloc_bootmem()` and `__alloc_bootmem_node()` are essentially the same. They are

**pgdat** This is the node to allocate from. It is omitted in the UMA case as 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 where as 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.

## 6.4 Freeing Memory

In contrast to the allocation functions, only two free function are provided which are `free_bootmem()` for UMA and `free_bootmem_node()` for NUMA. They both call `free_bootmem_core()` with the only difference being 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 signal a double-free.

An important restriction with the free functions is that only full pages may be freed. It is never recorded when a page is partially allocated so if only partially freed, the full page remains reserved. This is not as major a problem as it appears as the allocations always persist for the lifetime of the system; However, it is still an important restriction for developers during boot time.

## 6.5 Retiring the Boot Memory Allocator

Late in the bootstrapping process, the function `start_kernel()` is called which knows it is safe to remove the boot allocator and all its associated data structures. Each architecture is required to provide a function `mem_init()` that is responsible for destroying the boot memory allocator and its associated structures.

The purpose of the function is quite simple. It is responsible for calculating the dimensions of low and high memory and printing out an informational message to the user as well as performing final initialisations of the hardware if necessary. On the x86, the principal function of concern for the VM is the `free_pages_init()`.

This function first tells the boot memory allocator to retire itself by calling `free_all_bootmem()` for UMA architectures or `free_all_bootmem_node()` for NUMA. Both call the core function `free_all_bootmem_core()` with different parameters. The core function is simple in principle and performs the following tasks:

- For all unallocated pages known to the allocator for this node;
  - Clear the `PG_reserved` flag in its struct page;
  - Set the count to 1;
  - Call `__free_pages()` so that the buddy allocator (discussed next chapter) can build its free lists.

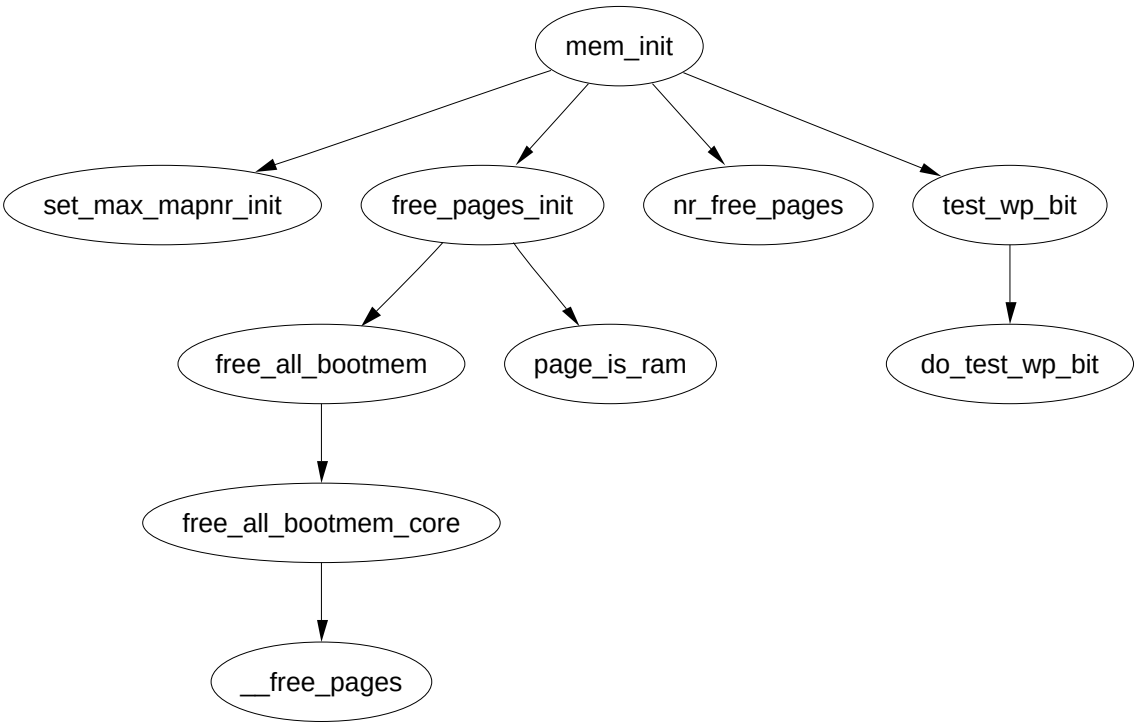


Figure 6.3: Call Graph: `mem_init()`

- 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. The remainder of the `free_pages_init()` function is responsible for those. After `free_all_bootmem()` returns, it first counts the number of reserved pages for accounting purposes and then calls the function `one_highpage_init()` for every page between `highstart_pfn` and `highend_pfn`.

This function simple clears the `PG_reserved` flag, sets the `PG_highmem` flag, 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 the code. All the init function declarations used for bootstrapping the system 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[3] that is used by bootstrapping code that is no longer required.

[3] 27 pages were freed while booting the kernel running on the machine this document is composed on.

`init_bootmem(unsigned long start, unsigned long page)`
This initialises the memory between 0 and the PFN `page`. The beginning of usable memory is at the PFN `start`

`reserve_bootmem(unsigned long addr, unsigned long size)`
Mark the pages between the address `addr` and `addr+size` reserved. Requests to partially reserve a page will result in the full page being reserved

`free_bootmem(unsigned long addr, unsigned long size)`
Mark the pages between the address `addr` and `addr+size` free

`alloc_bootmem(unsigned long size)`
Allocate `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

`alloc_bootmem_low(unsigned long size)`
Allocate `size` number of bytes from `ZONE_DMA`. The allocation will be aligned to the L1 hardware cache

`alloc_bootmem_pages(unsigned long size)`
Allocate `size` number of bytes from `ZONE_NORMAL` aligned on a page size so that full pages will be returned to the caller

`alloc_bootmem_low_pages(unsigned long size)`
Allocate `size` number of bytes from `ZONE_NORMAL` aligned on a page size so that full pages will be returned to the caller

`bootmem_bootmap_pages(unsigned long pages)`
Calculate the number of pages required to store a bitmap representing the allocation state of `pages` number of pages

`free_all_bootmem()`
Used at the boot allocator end of life. It cycles through all pages in the bitmap. For each one that is free, the flags are cleared and the page is freed to the physical page allocator (See next chapter) so the runtime allocator can set up its free lists

Table 6.1: Boot Memory Allocator API for UMA Architectures

`init_bootmem_node(pg_data_t *pgdat, unsigned long freepfn, unsigned long startpfn, unsigned long endpfn)`
For use with NUMA architectures. It initialise the memory between PFNs `startpfn` and `endpfn` with the first usable PFN at `freepfn`. Once initialised, the `pgdat` node is inserted into the `pgdat_list`

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

`free_bootmem_node(pg_data_t *pgdat, unsigned long physaddr, unsigned long size)`
Mark the pages between the address `addr` and `addr+size` on the specified node `pgdat` free

`alloc_bootmem_node(pg_data_t *pgdat, unsigned long size)`
Allocate `size` number of bytes from `ZONE_NORMAL` on the specified node `pgdat`. The allocation will be aligned to the L1 hardware cache to get the maximum benefit from the hardware cache

`alloc_bootmem_pages_node(pg_data_t *pgdat, unsigned long size)`
Allocate `size` number of bytes from `ZONE_NORMAL` on the specified node `pgdat` aligned on a page size so that full pages will be returned to the caller

`alloc_bootmem_low_pages_node(pg_data_t *pgdat, unsigned long size)`
Allocate `size` number of bytes from `ZONE_NORMAL` on the specified node `pgdat` aligned on a page size so that full pages will be returned to the caller

`bootmem_bootmap_pages(unsigned long pages)`
Same function as used for the UMA case. Calculate the number of pages required to store a bitmap representing the allocation state of `pages` number of pages

`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 specified node. For each one that is free, the page flags are cleared and the page is freed to the physical page allocator (See next chapter) so the runtime allocator can set up its free lists

Table 6.2: Boot Memory Allocator API for NUMA Architectures

# Chapter 7

# 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]. It is has been shown to be extremely fast in comparison to other allocators [KB85].

This is an allocation scheme which 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 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 details. The subsequent section will cover the flags which affect the allocator behavior and finally the problem of fragmentation and how the allocator handles it will be covered.

## 7.1 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` structures are maintained for each order that points to a linked list of blocks of pages that are free as indicated by Figure 7.1.

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 $2^{\text{MAX\_ORDER}-1}$ 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 sufficed. The page blocks are maintained on a linear linked list via `page→list`.

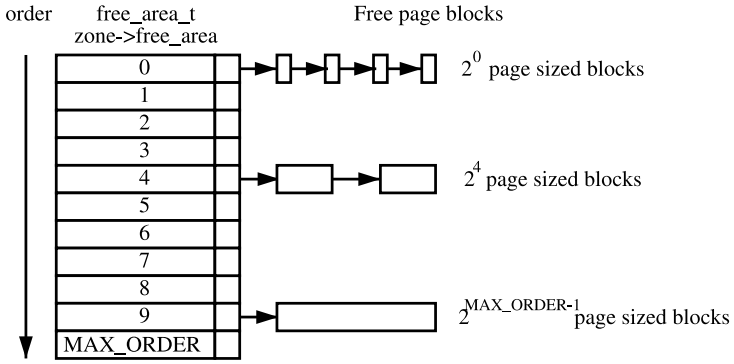


Figure 7.1: Free page block management

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

```
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 simply:

**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 map representing the pair of buddies is revealed.

## 7.2 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 7.4.

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 aware of what node is being used. A full list of page allocation APIs are listed in Table 7.1.

| |
|---|
| `alloc_page(unsigned int gfp_mask)`<br>Allocate a single page and return a struct address |
| `alloc_pages(unsigned int gfp_mask, unsigned int order)`<br>Allocate $2^{order}$ number of pages and returns a struct page |
| `get_free_page(unsigned int gfp_mask)`<br>Allocate a single page, zero it and return a virtual address |
| `__get_free_page(unsigned int gfp_mask)`<br>Allocate a single page and return a virtual address |
| `__get_free_pages(unsigned int gfp_mask, unsigned int order)`<br>Allocate $2^{order}$ number of pages and return a virtual address |
| `__get_dma_pages(unsigned int gfp_mask, unsigned int order)`<br>Allocate $2^{order}$ number of pages from the DMA zone and return a struct page |

Table 7.1: Physical Pages Allocation API

Allocations are always for a specified order, 0 in the case where a single page is required. If a free block cannot be found of the requested order, a higher order block is split into two buddies. One is allocated and the other is placed on the free list for the lower order. Figure 7.2 shows where a $2^4$ block is split and how the buddies are added to the free lists until a block for the process is available.

When the block is later freed, the buddy will be checked. If both are free, they are merged to form a higher order block and placed on the higher free list where its buddy is checked and so on. If the buddy is not free, the freed block is added to the free list at the current order. During these list manipulations, interrupts have to be disabled to prevent an interrupt handler manipulating the lists while a process has them in an inconsistent state. This is achieved by using an interrupt safe spinlock.

The second decision to make is which memory node or `pg_data_t` to use. Linux uses a *node-local* allocation policy which aims to use the memory bank associated with the CPU running the page allocating process. Here, the function `_alloc_pages()` is what is important as this function is different depending on whether the kernel is built for a UMA (function in `mm/page_alloc.c`) or NUMA (function in `mm/numa.c`) machine.

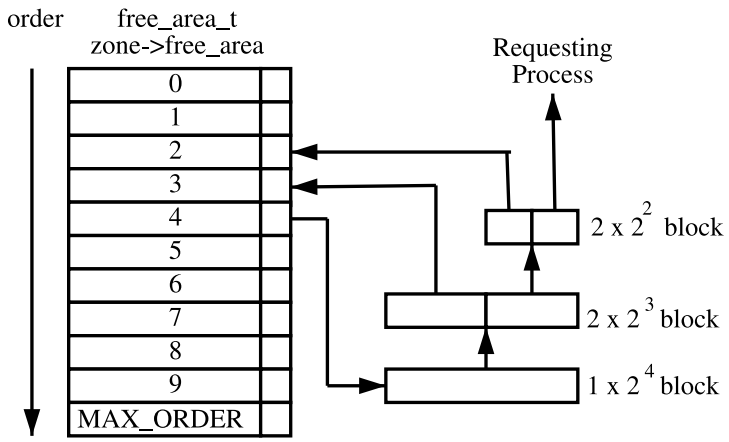


Figure 7.2: Allocating physical pages

Regardless of which API is used, `__alloc_pages()` in `mm/page_alloc.c` is the heart of the allocator. This function, which is never called directly, examines the selected zone and checks if it is suitable to allocate from based on the number of available pages. If the zone is not suitable, the allocator may fall back to other zones. The order of zones to fall back on are decided at boot time by the function `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.

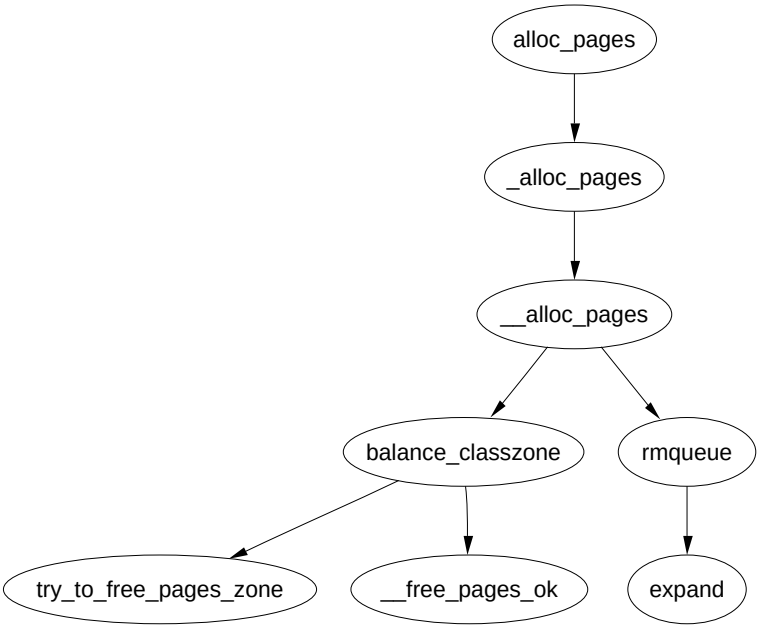


Figure 7.3: Call Graph: `alloc_pages()`

Once the zone has finally 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.

## 7.3 Free Pages

The API for the freeing of pages is a lot simpler and exists to help remember the order of the block to free. One disadvantage of a buddy allocator is that the caller has to remember the size of the original allocation. The API for freeing is listed in Table 7.2.

| |
|---|
| `__free_pages(struct page *page, unsigned int order)`<br>Free an order number of pages from the given page<br><br>`__free_page(struct page *page)`<br>Free a single page<br><br>`free_page(void *addr)`<br>Free a page from the given virtual address |

Table 7.2: Physical Pages Free API

The principal function for freeing pages is `__free_pages_ok()` and it should not be called directly. Instead the function `__free_pages()` is provided which performs simple checks first as indicated in Figure 7.4.

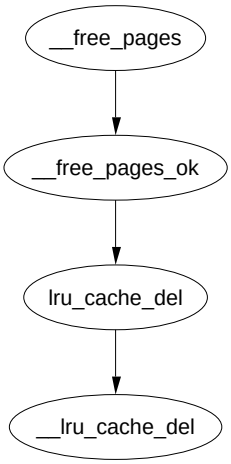


Figure 7.4: Call Graph: `__free_pages()`

When a buddy is freed, Linux tries to coalesce the buddies together immediately if possible. This is not optimal as the worst case scenario will have many coalitions followed by the immediate splitting of the same blocks [Vah96] although it is worth

noting later development kernels have implemented a lazy buddy system [BL89] which delays the coalescing of buddies until it is necessary.

To detect if the buddies can be merged or not, Linux checks the bit corresponding to the affected pair of buddies in `free_area→map`. As one buddy has just been freed by this function, it is obviously known that at least one buddy is free. If the bit in the map is 0 after toggling, we know that the other buddy must also be free 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 is a well known concept [Knu68]. As 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 *kth* bit from the right is examined. If it is 0, then the buddy will have this bit flipped. To get this bit, Linux creates a `mask` which is calculated as

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

The mask we are interested in is

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

Linux takes a shortcut in calculating this by noting that

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

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

## 7.4 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 as this flag indicates the caller may sleep. There are three sets of GFP flags, all defined in <`linux/mm.h`>.

The first of the three is the set of zone modifiers listed in Table 7.3. These flags indicate that the caller must try to allocate from a particular zone. The reader will note there is not a zone modifier for `ZONE_NORMAL`. This is because the zone modifier flag is used as an offset within an array and 0 implicitly means allocate from `ZONE_NORMAL`.

The next flags are action modifiers listed in Table 7.4. 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. 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 7.5. For clarity the `__GFP_` is removed from the table combinations so, the

| Flag | Description |
|---|---|
| __GFP_DMA | Allocate from ZONE_DMA if possible |
| __GFP_HIGHMEM | Allocate from ZONE_HIGHMEM if possible |
| GFP_DMA | Alias for __GFP_DMA |

Table 7.3: Low Level GFP Flags Affecting Zone Allocation

__GFP_HIGH flag will read as HIGH below. The combinations to form the high level flags are listed in Table 7.6

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 will not sleep, perform IO or access the filesystem. This flag would be used by an interrupt handler for example.

| Flag | Description |
|---|---|
| __GFP_WAIT | Indicates that the caller is not high priority and can sleep or reschedule |
| __GFP_HIGH | 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 |
| __GFP_IO | Indicates that the caller can perform low level IO. In 2.4.x, the main affect this has is determining if try_to_free_buffers() can flush buffers or not. It is used by at least one journaled filesystem |
| __GFP_HIGHIO | Determines that IO can be performed on pages mapped in high memory. Only used in try_to_free_buffers() |
| __GFP_FS | 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 7.4: Low Level GFP Flags Affecting Allocator Behavior

### 7.4.1 Process Flags

A process may also set flags in the task_struct which affects allocator behavior. The full list of process flags are defined in <linux/sched.h> but only the ones affecting VM behavior are listed in Table 7.7.

| High Level Flag | Low Level Flag Combination |
|---|---|
| GFP_ATOMIC | HIGH |
| GFP_NOIO | HIGH \| WAIT |
| GFP_NOHIGHIO | HIGH \| WAIT \| IO |
| GFP_NOFS | HIGH \| WAIT \| IO \| HIGHIO |
| GFP_KERNEL | HIGH \| WAIT \| IO \| HIGHIO \| FS |
| GFP_NFS | HIGH \| WAIT \| IO \| HIGHIO \| FS |
| GFP_USER | WAIT \| IO \| HIGHIO \| FS |
| GFP_HIGHUSER | WAIT \| IO \| HIGHIO \| FS \| HIGHMEM |
| GFP_KSWAPD | WAIT \| IO \| HIGHIO \| FS |

Table 7.5: Low Level GFP Flag Combinations For High Level

## 7.5 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 as large requests for contiguous pages are rare and usually `vmalloc()` (see Chapter 8) 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. While fragmentation is expected to be in the region of 28% [WJNB95], it has been shown that it can be in the region of 60%, in comparison to just 1% with the first 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 9. With this combination of allocators, the kernel can ensure that the amount of memory wasted due to internal fragmentation is kept to a minimum.

| High Level 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 IO. Many subsystems during init will use this system such as buffer_init() and inode_init() |
| GFP_NOIO | This is used by callers who are already performing an IO related function. For example, when the loop back 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 IO. If 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 alloc_bounce_page() during the creating of a bounce buffer for IO 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 | 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 GFP_USER is that this could use emergency pools of pages but that is a no-op on 2.4.x kernels |
| 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 5 would be reserved. Now it is not treated differently anywhere |
| GFP_USER | 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 GFP_USER (low) would fail where as the others would keep trying. Now it has no significance and is not treated any different to GFP_KERNEL |
| GFP_HIGHUSER | This flag indicates that the allocator should allocate from ZONE_HIGHMEM if possible. It is used when the page is allocated on behalf of a user process |
| GFP_KSWAPD | More historical significance. In reality this is not treated any different to GFP_KERNEL |

Table 7.6: High Level GFP Flags Affecting Allocator Behavior

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

Table 7.7: Process Flags Affecting Allocator Behavior

# Chapter 8

# Non-Contiguous Memory Allocation

It is preferable when dealing with large amounts of memory to use physically contiguous physical pages in memory both for cache related and memory access latency issues. Unfortunately, due to external fragmentation problems with the buddy allocator, this is not always possible. Linux provides a mechanism via `vmalloc()` where non-contiguous physically 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 5.1.

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. As allocations require altering the kernel page tables, there is a limitation on how much memory can be mapped with `vmalloc()` as only the virtual addresses space between `VMALLOC_START` and `VMALLOC_END` is available. As a result, it is used sparingly in the core kernel. In 2.4.20, it is only used for storing the swap map information (see Chapter 12) 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.

## 8.1 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:

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

```
19 };
```

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** 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 via 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 8.1

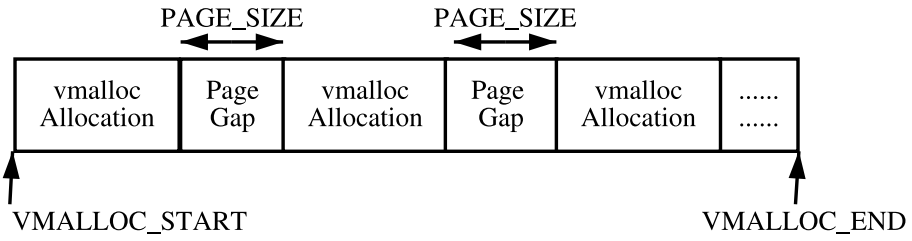


Figure 8.1: vmalloc Address Space

When the kernel wishes 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 IO (commonly referred to as *ioremapping*), this function will be called directly to map the requested area.

## 8.2 Allocating A Non-Contiguous Area

The functions `vmalloc()`, `vmalloc_dma()` and `vmalloc_32()` are provided to allocate a memory area that is contiguous in virtual address space. 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 8.2, there are two steps to allocating the area.

The first step with `get_vm_area()` finds 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.

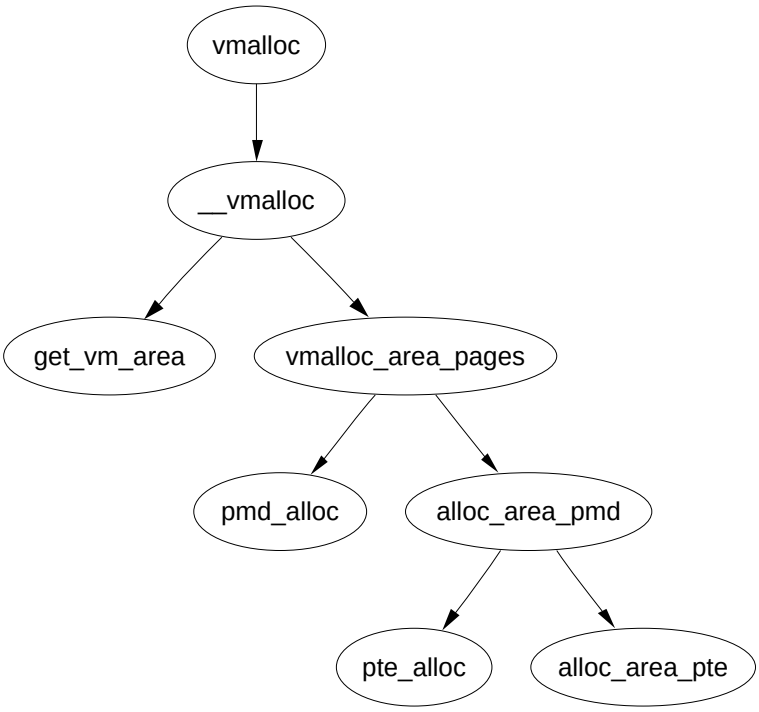


Figure 8.2: Call Graph: `vmalloc()`

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()`.

The page table updated by `vmalloc()` is not the current process but the master page table referenced by `init_mm`→`pgd`. This means that a process accessing the vmalloc area will cause a page fault exception as its page tables are not pointing to the correct area. There is a special case in the page fault handling code which knows that the fault occured in the vmalloc area and updates the current process page tables using information from the master page table.

## 8.3 Freeing A Non-Contiguous Area

The function `vfree()` is responsible for freeing a virtual area. It linearly searches the list of `vm_struct`s looking for the desired region and then calls `vmfree_area_pages()` on the region of memory to be freed.

`vmfree_area_pages()` is the exact opposite of `vmalloc_area_pages()`. It walks the page tables freeing up the page table entries and associated pages for the region.

| |
|---|
| `vmalloc(unsigned long size)`<br>Allocate a number of pages in vmalloc space that satisfy the requested size<br><br>`vmalloc_dma(unsigned long size)`<br>Allocate a number of pages from `ZONE_DMA`<br><br>`vmalloc_32(unsigned long size)`<br>Allocate memory that is suitable for 32 bit addressing. This ensures that the physical page frames are in `ZONE_NORMAL` which 32 bit devices will require |

Table 8.1: Non-Contiguous Memory Allocation API

| |
|---|
| `vfree(void *addr)`<br>Free a region of memory allocated with `vmalloc()`, `vmalloc_dma()` or `vmalloc_32()` |

Table 8.2: Non-Contiguous Memory Free API

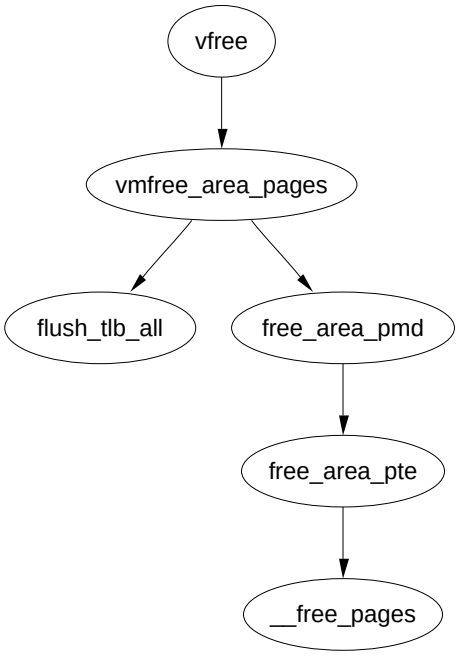


Figure 8.3: Call Graph: `vfree()`

# Chapter 9

# Slab Allocator

In this chapter, the general purpose allocator is described. It is a slab allocator which is very similar in many respects to the general kernel allocator used in Solaris [MM01]. It is heavily based on the first slab allocator paper by Bonwick [Bon94] with many improvements that bear a close resemblance to those described in his later paper [BA01]. We will 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 initialised state available for use by the kernel. Without an object based allocator, the kernel will spend much of its time allocating, initialising and freeing the same object. The slab allocator aims to 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 via the `next` field in the cache struct.

Each cache maintains blocks of contiguous pages in memory called *slabs* which are carved up into small chunks for the data structures and objects the cache manages. The relationship between these different structures is illustrated in Figure 9.1.

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, initialising and destroying objects. Benchmarks on Solaris showed excellent speed improvements for allocations with the slab allocator in use [Bon94];
- The better utilisation of hardware cache by aligning objects to the L1 or L2 caches.

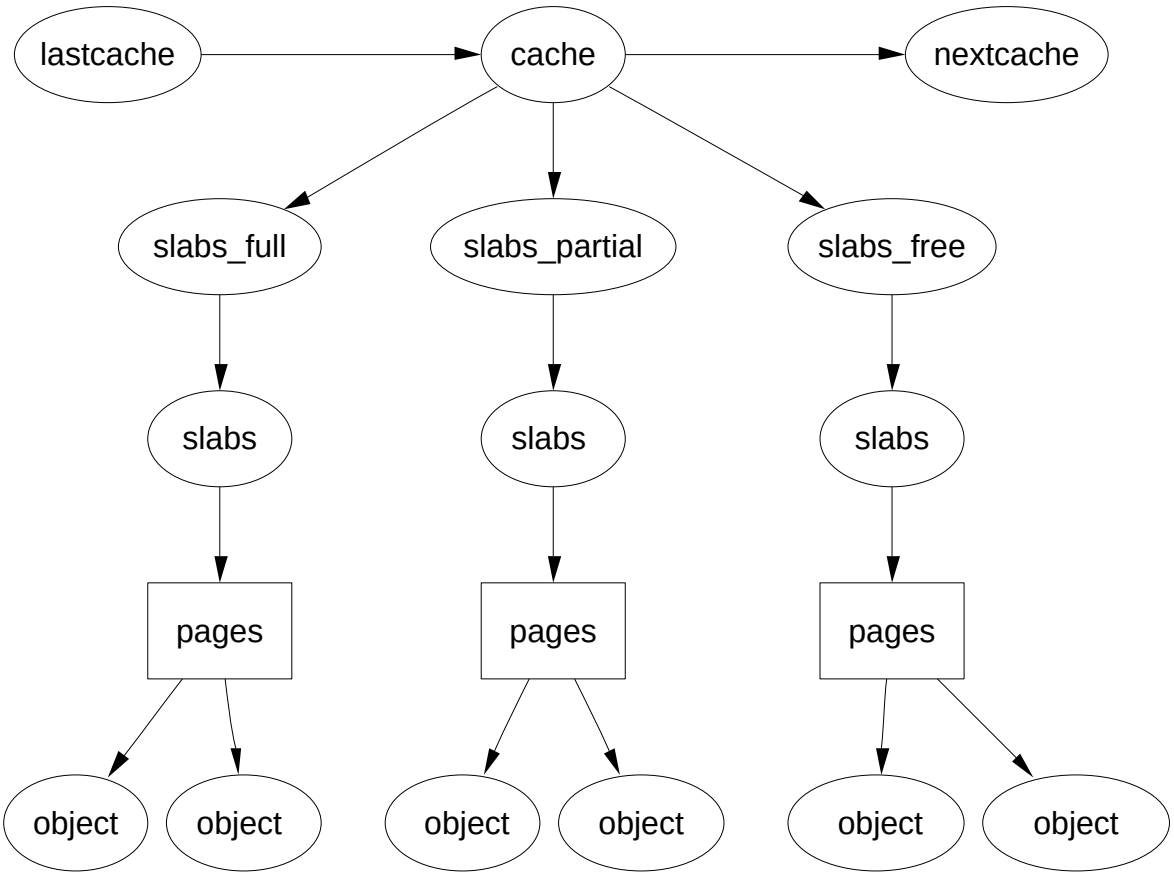


Figure 9.1: Layout of the Slab Allocator

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 $2^{17}$ (131072) bytes are maintained. One cache set is suitable for use with DMA devices. These caches are called size-N and size-N(DMA) where N is the size of the allocation, and a function `kmalloc()` (see Section 9.4.1) is provided for allocating them. With this, the single greatest problem with the low level page allocator is addressed. The sizes caches are discussed in further detail in Section 9.4.

The second task of the slab allocator is to maintain caches of commonly used objects. For many structures used in the kernel, the time needed to initialise an object is comparable to, or exceeds, the cost of allocating space for it. When a new slab is created, a number of objects are packed into it and initialised using a constructor if available. When an object is freed, it is left in its initialised state so that object allocation will be quick.

The final task of the slab allocator is hardware cache utilization. If there is space left over after objects are packed into a slab, the remaining space is used to *color* the slab. By giving objects in different slabs different offsets, they will be assigned to different CPU cache lines helping ensure that objects from the same

cache will be unlikely to flush each other. With this, space that would otherwise be wasted fulfills a new function. Linux does not attempt to color pages [Kes91], or order where objects are placed such as those described for data [GAV95] or code segments [HK97] but the scheme used does help improve cache line usage. Cache colouring is further discussed in Section 9.1.5. On an SMP system, a further step is taken to help cache utilization where each cache has a small array of objects reserved for each CPU. This is discussed further in Section 9.5.

The slab allocator provides the additional option of slab debugging if the option is set at compile time with `CONFIG_SLAB_DEBUG`. Two debugging features are providing called *red zoning* and *object poisoning*. With red zoning, a marker is placed at either end of the object. If this mark is disturbed, the allocator knows the object where a buffer overflow occured and reports it. Poisoning an object will fill it with a predefined bit pattern(defined `0x5A` in `mm/slab.c`) at slab creation and after a free. At allocation, this pattern is examined and if it is changed, the allocator knows that the object was used before it was allocated and flags it.

The small, but powerful, API which the allocator exports is listed in Table 9.5.

## 9.1 Caches

One cache exists for each type of object that is to be cached. For a full list of caches available on a running system, run `cat /proc/slabinfo` . This file gives some basic information on the caches. An excerpt from the output of this file looks like:

```
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 fields correspond to a field in the `struct kmem_cache_s` structure. The columns listed in the excerpt above are:

**cache-name** A human readable name such as “tcp_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 9.5. The columns are:

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

**batchcount** 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 its objects in use. `slabs_partial` has free objects in it and so is a prime candidate for allocation of objects. `slabs_free` has no allocated objects and so is a prime candidate for slab destruction.

### 9.1.1 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 and so 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;

**flags** These flags determine how parts of the allocator will behave when dealing with the cache. See Section 9.1.2;

**num** This is the number of objects contained in each slab;

**spinlock** A spinlock protecting the structure from concurrent accessses;

**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 $2^{\text{gfporder}}$ pages as these are the allocation sizes the buddy allocator provides;

**gfpflags** The GFP flags used when calling the buddy allocator to allocate pages are stored here. See Section 7.4 for a full list;

**colour** Each slab stores objects in different cache lines if possible. Cache colouring will be further discussed in Section 9.1.5;

**colour_off** 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 flag is set to indicate if the cache is growing or not. If it is, it is much less likely this cache will be selected to reap free slabs under memory pressure;

**dflags** These are the dynamic flags which change during the cache lifetime. See Section 9.1.3;

**ctor** A complex object has the option of providing a constructor function to be called to initialise 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 field is not referred to anywhere in the code.

```
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 9.5.

```
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 figures are only available if the `CONFIG_SLAB_DEBUG` option is set during compile time. They are all beancounters 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.

### 9.1.2 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 is listed in Table 9.1. The only relevant flag is the `CFGS_OFF_SLAB` flag which determines where the slab descriptor is stored.

| Flag | Description |
|---|---|
| `CFGS_OFF_SLAB` | Indicates that the slab managers for this cache are kept off-slab. This is discussed further in Section 9.2.1 |
| `CFLGS_OPTIMIZE` | This flag is only ever set and never used |

Table 9.1: Internal cache static flags

The second set are set by the cache creator and they determine how the allocator treats the slab and how objects are stored. They are listed in Table 9.2.

The last flags are only available if the compile option `CONFIG_SLAB_DEBUG` is set. 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_HWCACHE_ALIGN | Align the objects to the L1 CPU cache |
| SLAB_MUST_HWCACHE_ALIGN | Force alignment to the L1 CPU cache even if very wasteful or debugging |
| SLAB_NO_REAP | Never reap slabs in this cache |
| SLAB_CACHE_DMA | Use memory from ZONE_DMA |

Table 9.2: Cache static flags set by caller

| Flag | Description |
|---|---|
| SLAB_DEBUG_FREE | Perform expensive checks on free |
| SLAB_DEBUG_INITIAL | After an object is freed, the constructor is called with a flag set that tells it to check to make sure it is initialised 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 initialised |

Table 9.3: Cache static debug flags

To prevent callers using the wrong flags a CREATE_MASK is defined in mm/slab.c consisting 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.

### 9.1.3 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 find a cache with this flag set, it skips the cache and removes the flag.

### 9.1.4 Cache Allocation Flags

These flags correspond to the GFP page flag 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 7.4 so will not be discussed in detail here. It is presumed the existence of these flags are for clarity and in case the slab allocator needed to behave differently in response to a particular flag but 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 9.4: Cache Allocation Flags

### 9.1.5 Cache Colouring

To utilise 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, it is calculated how many objects can fit on a slab (see Section 9.2.7) and how many bytes would be wasted. Based on wastage, two figures are calculated for the cache descriptor

**colour** This is the number of different offsets that can be used;

**colour_off** This is the multiple to offset each objects by in the slab.

With the objects offset, 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, 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.

### 9.1.6 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

- 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 slab size to the hardware cache;
- Calculate colour offsets ;
- Initialise remaining fields in cache descriptor;
- Add the new cache to the cache chain.

Figure 9.2 shows the call graph relevant to the creation of a cache; each function is fully described in the Code Commentary.

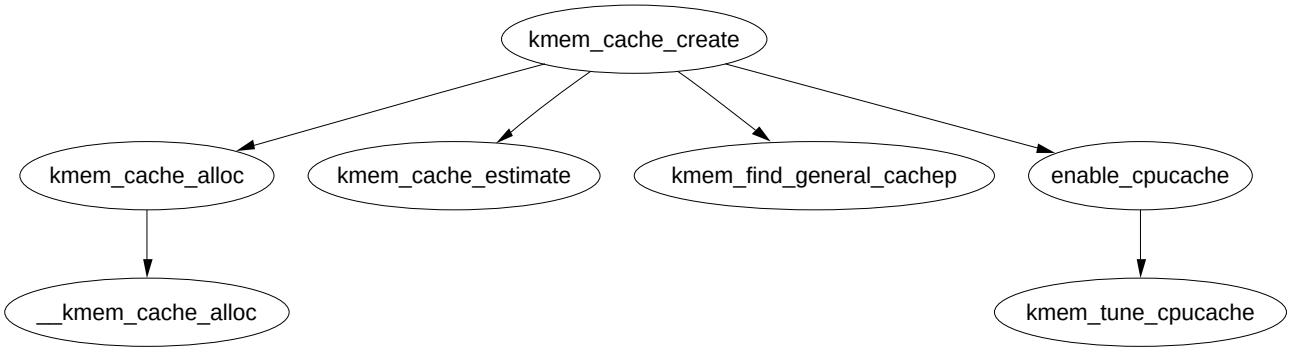


Figure 9.2: Call Graph: `kmem_cache_create()`

### 9.1.7 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 call graph in Figure 9.3 is deceptively simple as the task of selecting the proper cache to reap is quite long. In the event that there are numerous caches in the system, only `REAP_SCANLEN`[1] caches are examined in each call. The last cache to be scanned is stored in the variable `clock_searchp` so as not to examine the same caches repeatedly. For each scanned cache, the reaper does the following

[1] `REAP_SCANLEN` is statically defined as 10.

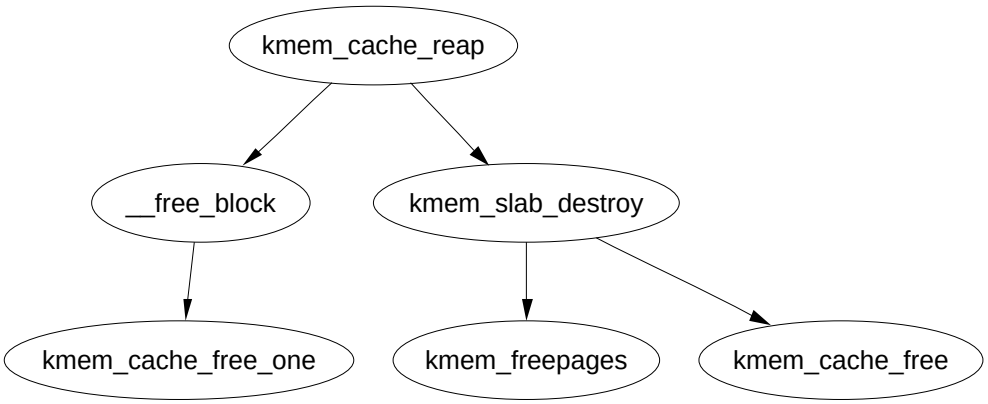


Figure 9.3: Call Graph: `kmem_cache_reap()`

- Check flags for `SLAB_NO_REAP` and skip if set;
- If the cache is growing, skip it;
- if the cache has grown recently (`DFLGS_GROWN` is set in `dflags`), skip it but clear the flag so it will be reaped the next time;
- Count the number of free slabs in `slabs_free` and calculate how many pages that would free in the variable `pages`;
- If the cache has constructors or large slabs, adjust `pages` to make it less likely for the cache to be selected;
- If the number of pages that would be freed exceeds `REAP_PERFECT`, free half of the slabs in `slabs_free`;
- Otherwise scan the rest of the caches and select the one that would free the most pages for freeing half of its slabs in `slabs_free`.

### 9.1.8 Cache Shrinking

When a cache is selected to shrink itself, the steps it takes are simple and brutal

- Delete all objects in the per CPU caches;
- Delete all slabs from `slabs_free` unless the growing flag gets set.

Linux is nothing, if not subtle.

Two varieties of shrink functions are provided with confusingly similar names. `kmem_cache_shrink()` removes all slabs from `slabs_free` and returns the number of pages freed as a result. This is the principal function exported for use by the slab allocator users.

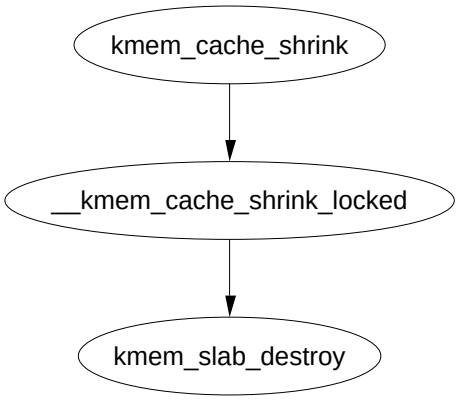


Figure 9.4: Call Graph: `kmem_cache_shrink()`

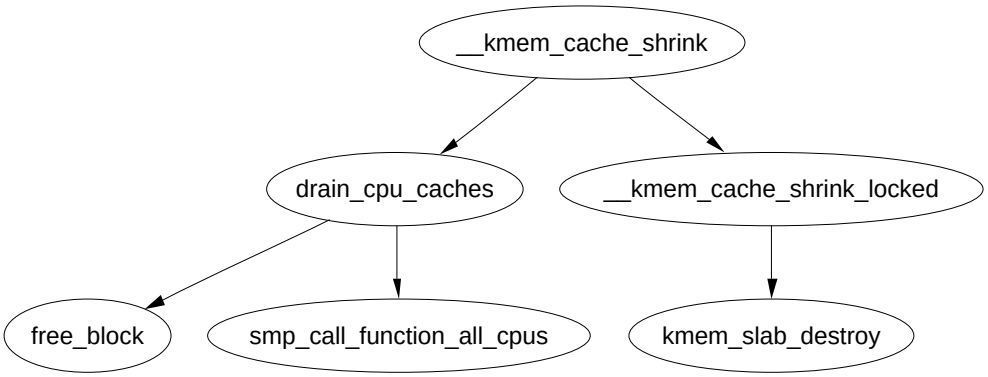


Figure 9.5: Call Graph: `__kmem_cache_shrink()`

The second function `__kmem_cache_shrink()` frees all slabs from `slabs_free` and then verifies that `slabs_partial` and `slabs_full` are empty. This is for internal use only and is important during cache destruction when it doesn't matter how many pages are freed, just that the cache is empty.

### 9.1.9 Cache Destroying

When a module is unloaded, it is responsible for destroying any cache with the function `kmem_cache_destroy()`. It is important that the cache is properly destroyed as two caches of the same human-readable name are not allowed to exist. Core kernel code often does not bother to destroy its caches as their existence persists for the life of the system. The steps taken to destroy a cache are

- Delete the cache from the cache chain;
- Shrink the cache to delete all slabs;
- Free any per CPU caches (`kfree()`);

- Delete the cache descriptor from the `cache_cache`.

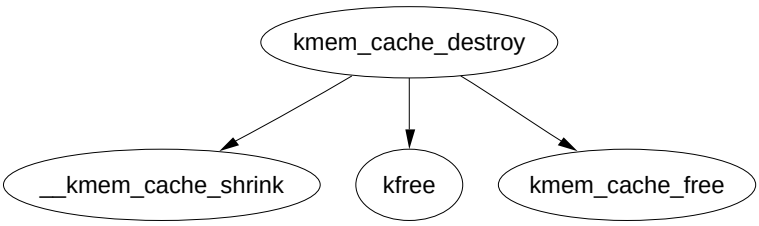


Figure 9.6: Call Graph: `kmem_cache_destroy()`

## 9.2 Slabs

This section will describe how a slab is structured and managed. The struct which describes it is much simpler than the cache descriptor, but how the slab is arranged is considerably more complex. It is declared as follows:

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

The fields in this simple struct are as follows:

**list** This is the linked list the slab belongs to. This will be one of `slab_full`, `slab_partial` or `slab_free` from the cache manager;

**colouroff** This is the colour offset from the base address of the first object within the slab. The address of the first object is `s_mem + colouroff`;

**s_mem** This gives the starting address of the first object within the slab;

**inuse** This gives the number of active objects in the slab;

**free** This is an array of `bufctl`s used for storing locations of free objects. See Section 9.2.3 for further details.

The reader will note that given the slab manager or an object 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`

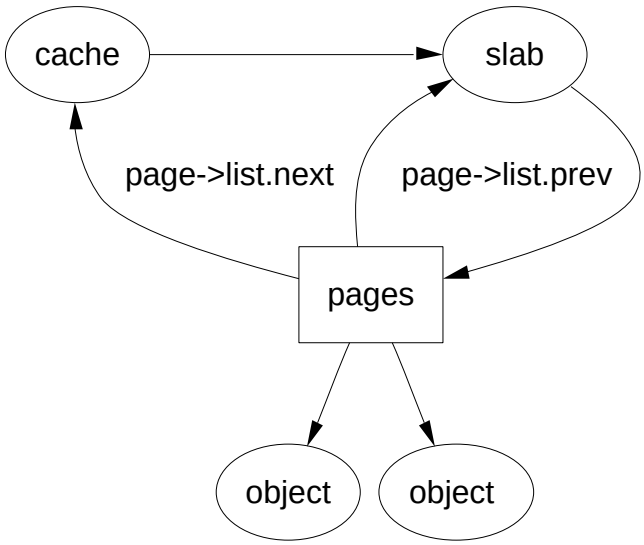


Figure 9.7: Page to Cache and Slab Relationship

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 9.7

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.

### 9.2.1 Storing the Slab Descriptor

If the objects are larger than a threshold (512 bytes on x86), `CFGS_OFF_SLAB` is set in the cache flags and the *slab descriptor* is kept off-slab in one of the sizes cache (see Section 9.4). 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 `bufctl`s but that is unimportant as the objects are large and so there should not be many stored in a single slab.

Alternatively, the slab manger 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` array. The array is responsible for tracking where the next free object is stored and is discussed later in the chapter. The objects are stored after the `kmem_bufctl_t` array.

Figure 9.8 should help clarify what a slab with the descriptor on-slab looks like and Figure 9.9 illustrates how a cache uses a sizes cache to store the slab descriptor when the descriptor is kept off-slab.

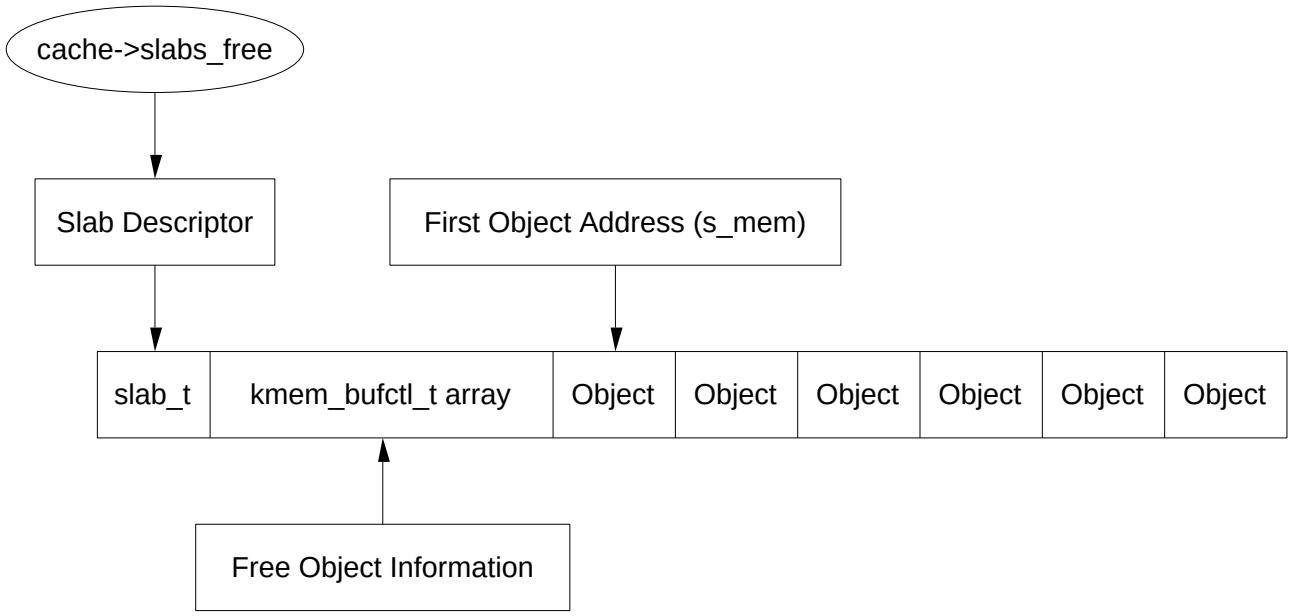


Figure 9.8: Slab With Descriptor On-Slab

### 9.2.2 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()`. This is frequently called "cache growing" and occurs when no objects are left in the `slabs_partial` list and there are no slabs in `slabs_free`. The tasks it fulfills are

- Perform basic sanity checks to guard against bad usage;
- Calculate colour offset for objects in this slab;
- Allocate memory for slab and acquire a slab descriptor;
- Link the pages used for the slab to the slab and cache descriptors described in Section 9.2;
- Initialise objects in the slab;
- Add the slab to the cache.

### 9.2.3 Tracking Free Objects

The slab allocator has got to have a quick and simple means of tracking where free objects are on the partially filled slabs. It achieves this by using a `kmem_bufctl_t` array that is associated with each slab manager as obviously it is up to the slab manager to know where its free objects are.

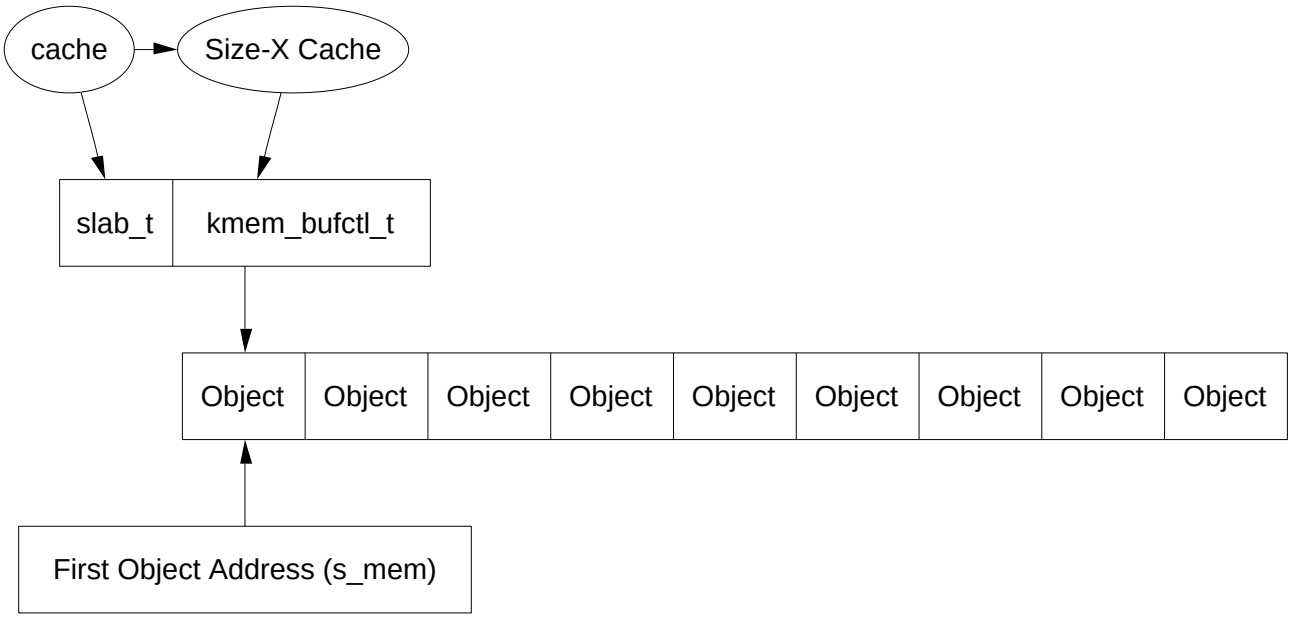


Figure 9.9: Slab With Descriptor Off-Slab

Historically, and according to the paper describing the slab allocator [Bon94], `kmem_bufctl_t` was a linked list of objects. In Linux 2.2.x, this struct was a union of three items, a pointer to the next free object, a pointer to the slab manager and a pointer to the object. Which it was depended on the state of the object.

Today, the slab and cache an object belongs to is determined by the `struct page` and `kmem_bufctl_t` is simply an integer array of object indices. The number of elements in the array is the same as the number of objects on the slab.

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

As the array is kept after the slab descriptor and there is no pointer to the first element directly, a helper macro `slab_bufctl()` is provided.

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

This seemingly cryptic macro is quite simple when broken down. The parameter `slabp` is a pointer to the slab manager. The block `((slab_t*)slabp)+1` casts `slabp` to a `slab_t` struct and adds 1 to it. This will give a `slab_t *` pointer to the beginning of the `kmem_bufctl_t` array. `(kmem_bufctl_t *)` recasts that pointer back to the required type. The results in blocks of code that contain `slab_bufctl(slabp)[i]`. Translated, that says "take a pointer to a slab descriptor, offset it with `slab_bufctl()` to the beginning of the `kmem_bufctl_t` array and return the i*th* element of the array".

The index to the next free object in the slab is stored in `slab_t→free` eliminating the need for a linked list to track free objects. When objects are allocated or freed, this pointer is updated based on information in the `kmem_bufctl_t` array.

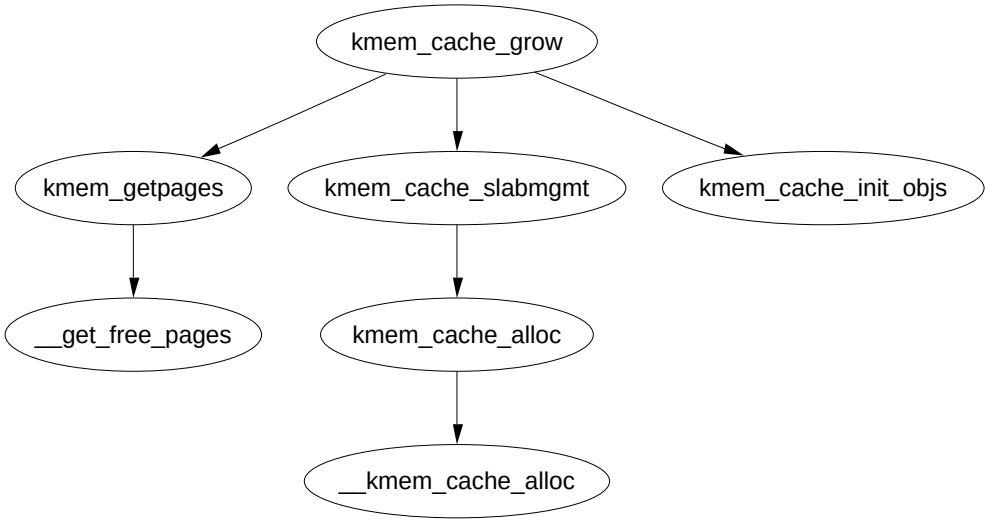


Figure 9.10: Call Graph: `kmem_cache_grow()`

### 9.2.4 Initialising the `kmem_bufctl_t` Array

When a cache is grown, all the objects and the `kmem_bufctl_t` array on the slab are initialised. The array is filled with the index of each object beginning with 1 and ending with the marker `BUFCTL_END`. For a slab with 5 objects, the elements of the array would look like Figure 9.11.

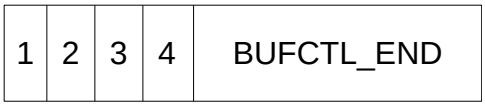


Figure 9.11: Initialised `kmem_bufctl_t` Array

The value 0 is stored in `slab_t→free` as the $0th$ object is the first free object to be used. The idea is that for a given object $n$, the index of the next free object will be stored in `kmem_bufctl_t[n]`. Looking at the array above, the next object free after 0 is 1. After 1, there are two and so on. As the array is used, this arrangement will make the array act as a LIFO for free objects.

### 9.2.5 Finding the Next Free Object

`kmem_cache_alloc()` will call `kmem_cache_alloc_one_tail()` when allocating an object to perform the "real" work of updating the `kmem_bufctl_t()` array.

`slab_t→free` has the index of the first free object. The index of the next free object is at `kmem_bufctl_t[slab_t→free]`. In code terms, this looks like

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

slabp→s_mem is the index of the first object on the slab. slabp→free is the index of the object to allocate and it has to be multiplied by the size of an object.

The index of the next free object is stored at kmem_bufctl_t[slabp→free]. There is no pointer directly to the array hence the helper macro slab_bufctl() is used. Note that the kmem_bufctl_t array is not changed during allocations but that the elements that are unallocated are unreachable. For example, after two allocations, index 0 and 1 of the kmem_bufctl_t array are not pointed to by any other element.

### 9.2.6 Updating kmem_bufctl_t

The kmem_bufctl_t list is only updated when an object is freed in the function kmem_cache_free_one(). The array is updated with this block of code:

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

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 pseudo linked list. slabp→free is updated to the object being freed so that it will be the next one allocated.

### 9.2.7 Calculating the Number of Objects on a Slab

During cache creation, the function kmem_cache_estimate() is called to estimate how many objects may be stored on a single slab taking 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 colouring is to be used.

The calculation is quite basic and takes the following steps

- Initialise wastage to be the total size of the slab, $\text{PAGE\_SIZE}^{\text{gfp\_order}}$;
- 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 filled;
- Return the number of objects and bytes wasted.

### 9.2.8 Slab Destroying

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

- 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 at Figure 9.12 is very simple.

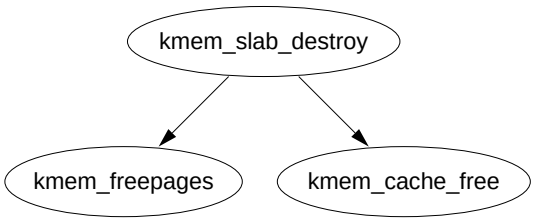


Figure 9.12: Call Graph: `kmem_slab_destroy()`

## 9.3 Objects

This section will cover how objects are managed. At this point, most of the really hard work has been completed by either the cache or slab managers.

### 9.3.1 Initialising Objects in a Slab

When a slab is created, all the objects in it are put in an initialised state. If a constructor is available, it is called for each object and it is expected that objects are left in an initialised state upon free. Conceptually the initialisation is very simple, cycle through all objects and call the constructor and initialise the `kmem_bufctl` for it. The function `kmem_cache_init_objs()` is responsible for initialising the objects.

### 9.3.2 Object Allocation

The function `kmem_cache_alloc()` is responsible for allocating one object to the caller which behaves slightly different in the UP and SMP cases. Figure 9.13 shows the basic call graph that is used to allocate an object in the SMP case.

There are four basic steps. The first step (`kmem_cache_alloc_head()`) covers basic checking to make sure the allocation is allowable. The second step is to select which slabs list to allocate from. This will be one of `slabs_partial` or `slabs_free`. If there are no slabs in `slabs_free`, the cache is grown (see Section 9.2.2) to create

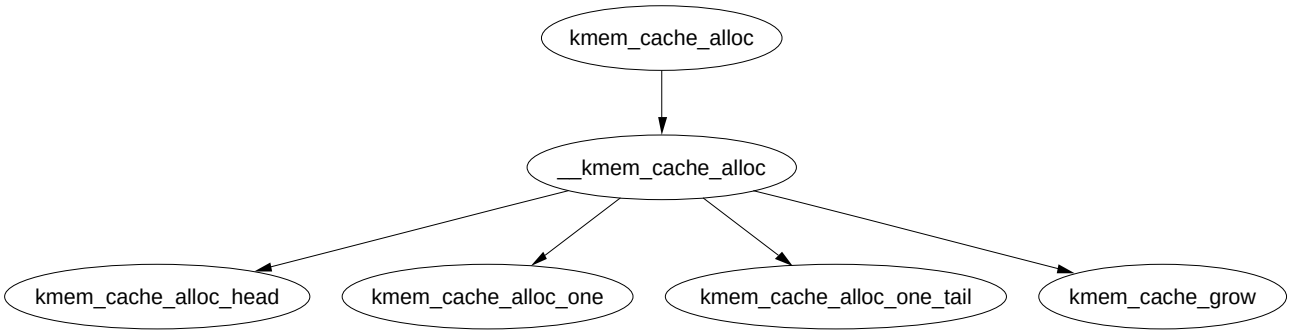


Figure 9.13: Call Graph: `kmem_cache_alloc()`

a new slab in `slabs_free`. The final step is to allocate the object from the selected slab.

The SMP case takes one further step. Before allocating one object, it will check to see if there is one available from the per-CPU cache and will use it if there is. If there is not, it will allocate `batchcount` number of objects in bulk and place them in its per-cpu cache. See Section 9.5 for more information on the per-cpu caches.

### 9.3.3 Object Freeing

`kmem_cache_free()` is used to free objects and it has a relatively simple task. Just like `kmem_cache_alloc()`, it behaves differently in the UP and SMP cases. The principal difference between the two cases is that in the UP case, the object is returned directly to the slab but with the SMP case, the object is returned to the per-cpu cache. In both cases, the destructor for the object will be called if one is available. The destructor is responsible for returning the object to the initialised state.

## 9.4 Sizes Cache

Linux keeps two sets of caches for small memory allocations for which the physical page allocator is unsuitable. One cache is for use with DMA and the other suitable for normal use. The human readable names for these caches are *size-N cache* and *size-N(DMA) cache* which are viewable from `/proc/slabinfo`. Information for each sized cache is stored in a `cache_sizes_t` struct defined in `mm/slab.c`

```
331 typedef struct cache_sizes {
332         size_t           cs_size;
333         kmem_cache_t    *cs_cachep;
334         kmem_cache_t    *cs_dmacachep;
335 } cache_sizes_t;
```

The fields in this struct are described as follows:

**cs_size** The size of the memory block;

**cs_cachep** The cache of blocks for normal memory use;

**cs_dmacachep** The cache of blocks for use with DMA.

As there are a limited number of these caches that exist, a static array called `cache_sizes` is initialised at compile time beginning with 32 bytes on a 4KiB machine and 64 for greater page sizes.

```
337 static cache_sizes_t cache_sizes[] = {
338 #if PAGE_SIZE == 4096
339         {    32,        NULL, NULL},
340 #endif
341         {    64,        NULL, NULL},
342         {   128,        NULL, NULL},
343         {   256,        NULL, NULL},
344         {   512,        NULL, NULL},
345         {  1024,        NULL, NULL},
346         {  2048,        NULL, NULL},
347         {  4096,        NULL, NULL},
348         {  8192,        NULL, NULL},
349         { 16384,        NULL, NULL},
350         { 32768,        NULL, NULL},
351         { 65536,        NULL, NULL},
352         {131072,        NULL, NULL},
353         {     0,        NULL, NULL}
```

As is obvious, this is a static array that is zero terminated consisting of buffers of succeeding powers of 2 from $2^5$ to $2^{17}$ . An array now exists that describes each sized cache which must be initialised with caches at system startup.

### 9.4.1 kmalloc

With the existence of the sizes cache, the slab allocator is able to offer a new allocator function, `kmalloc()` for use when small memory buffers are required. When a request is received, the appropriate sizes cache is selected and an object assigned from it. The call graph on Figure 9.14 is therefore very simple as all the hard work is in cache allocation.

### 9.4.2 kfree

Just as there is a `kmalloc()` function to allocate small memory objects for use, there is a `kfree()` for freeing it. As with `kmalloc()`, the real work takes place during object freeing (See Section 9.3.3) so the call graph in Figure 9.15 is very simple.

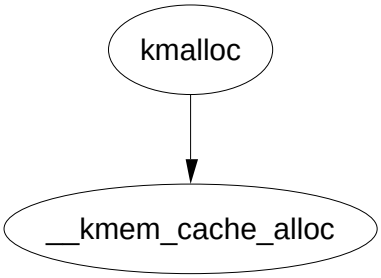


Figure 9.14: Call Graph: `kmalloc()`

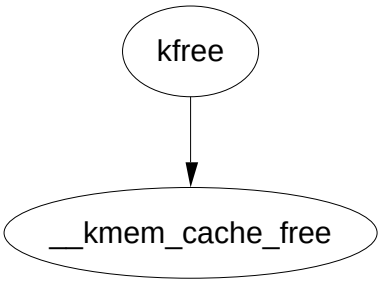


Figure 9.15: Call Graph: `kfree()`

## 9.5 Per-CPU Object Cache

One of the tasks the slab allocator is dedicated to is improved hardware cache utilization. An aim of high performance computing [CS98] in general is to use data on the same CPU for as long as possible. Linux achieves this by trying to keep objects in the same CPU cache with a Per-CPU object cache, simply called a *cpucache* for each CPU in the system.

When allocating or freeing objects, they are placed in the cpucache. When there are no objects free, a `batch` of objects is placed into the pool. When the pool gets too large, half of them are removed and placed in the global cache. This way the hardware cache will be used for as long as possible on the same CPU.

The second major benefit of this method is that spinlocks do not have to be held when accessing the CPU pool as we are guaranteed another CPU won't access the local data. This is important because without the caches, the spinlock would have to be acquired for every allocation and free which is unnecessarily expensive.

### 9.5.1 Describing the Per-CPU Object Cache

Each cache descriptor has a pointer to an array of cpucaches, described in the cache descriptor as

```
231         cpucache_t              *cpudata[NR_CPUS];
```

This structure is very simple

```
173 typedef struct cpucache_s {
174         unsigned int avail;
175         unsigned int limit;
176 } cpucache_t;
```

The fields are as follows:

**avail** This is the number of free objects available on this cpucache;

**limit** This is the total number of free objects that can exist.

A helper macro `cc_data()` is provided to give the cpucache for a given cache and processor. It is defined as

```
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.

### 9.5.2 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

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

`cc_entry()` is a helper macro which gives a pointer to the first 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, increments the value by the size of the `cpucache_t` descriptor giving the first object in the cache.

### 9.5.3 Enabling Per-CPU Caches

When a cache is created, its CPU cache has to be enabled and memory allocated for it using `kmalloc()`. The function `enable_cpucache()` is responsible for deciding what size to make the cache and 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 being enabled prematurely. The function `enable_all_cpucaches()` cycles through all caches in the cache chain and enables their cpucache.

Once the CPU cache has been setup, it can be accessed without locking as a CPU will never access the wrong cpucache so it is guaranteed safe access to it.

### 9.5.4 Updating Per-CPU Information

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

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

### 9.5.5 Draining a Per-CPU Cache

When a cache is being shrunk, its first step is to drain the cpucaches of any objects they might have. 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.

## 9.6 Slab Allocator Initialisation

Here we will describe how the slab allocator initialises 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 initialised 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 initialised the `kmem_cache_t` struct as follows:

**358-360** Initialise 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 consider it for reaping ever;

**363** Initialise the spinlock unlocked;

**364** Align the objects to the L1 cache;

**365** Record the human readable name.

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

## 9.7 Interfacing with the Buddy Allocator

The slab allocator does not come with pages attached, it must ask the physical page allocator for its pages. For this two interfaces are provided, `kmem_getpages()` and `kmem_freepages()`. They are basically wrappers around the buddy allocators API so that slab flags will be taken into account for allocations.

`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

`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. Called when memory is tight

`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.

`kmem_cache_alloc(kmem_cache_t *cachep, int flags)`
Allocate a single object from the cache and return it to the caller

`kmem_cache_free(kmem_cache_t *cachep, void *objp)`
Free an object and return it to the cache

`kmalloc(size_t size, int flags)`
Allocate a block of memory from one of the sizes cache

`kfree(const void *objp)`
Free a block of memory allocated with kmalloc

`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 9.5: Slab Allocator API for caches

# Chapter 10

# 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 32bit machine[1] as explained in Section 5.1.

There are many high end 32 bit machines that 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 10.2.

High memory and IO has a related problem which must be addressed as not all devices are able to address high memory or all the memory available to the CPU in the case of PAE. Indeed some are limited to addresses the size of a signed 32 bit integer or 2GiB. 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 10.4.

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.

## 10.1 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 is typically only a few pages. This means that there is slightly below 32MiB of page table space for mapping pages from high memory into usable space.

[1] On 64 bit hardware, this is not really an issue as 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.

For mapping pages, a single page set of PTEs is stored at the beginning of the PKMap area to allow 1024 high pages to be mapped into low memory for short periods with the function `kmap()` and 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 set up during system initialisation[2]. 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 called `pkmap_count` which has `LAST_PKMAP` entries in it. On an x86 system without PAE, this is 1024 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. Flushes are delayed until every slot has been used at least once as 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.

## 10.2 Mapping High Memory Pages

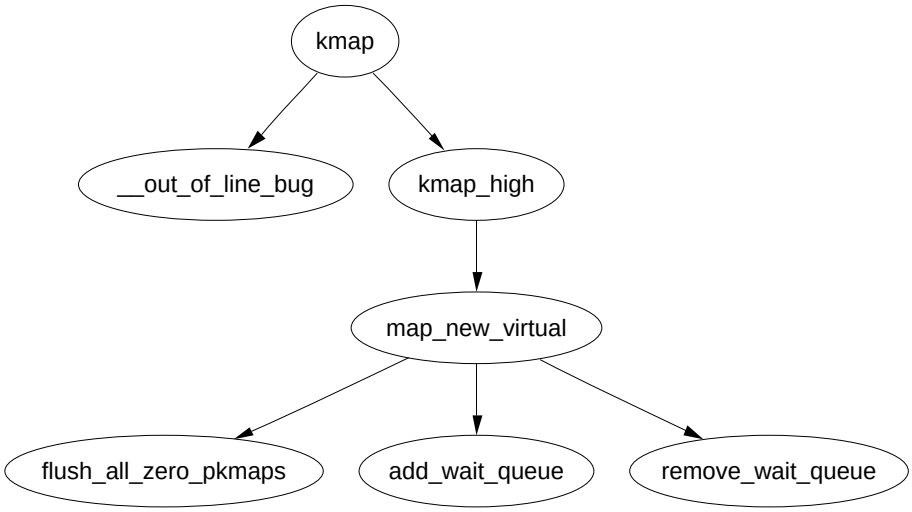


Figure 10.1: Call Graph: `kmap()`

The kmap pool is quite small so it is important that users of `kmap()` call

[2]On the x86, this takes place at the end of the `pagetable_init()` function.

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 API for mapping pages from high memory is described in Table 10.1.

| |
|---|
| `kmap(struct page *page)`<br>Takes a struct page from high memory and maps it into low memory. The address returned is the virtual address of the mapping |
| `kunmap(struct page *page)`<br>Unmaps a struct page from low memory and frees up the page table entry mapping it |
| `kmap_atomic(struct page *page, enum km_type type)`<br>There are slots maintained in the map for atomic use by interrupts (see Section 10.3). Their use is heavily discouraged and callers of this function may not sleep or schedule. This function will map a page from high memory atomically for a specific purpose |
| `kunmap_atomic(void *kvaddr, enum km_type type)`<br>Unmap a page that was mapped atomically |

Table 10.1: High Memory Mapping/Unmapping API

The kmap() function itself is fairly simple. It first checks to make sure an interrupt is not calling this function(as 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.

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 field which is set if the page is already mapped. If it is NULL, map_new_virtual() provides a mapping for the page.

Creating a new virtual mapping with map_new_virtual() is a simple case of linearly scanning pkmap_count. The scan starts at last_pkmap_nr instead of 0 to prevent searching over the same areas repeatedly between kmap()s. When last_pkmap_nr wraps around to 0, flush_all_zero_pkmaps() is called to set all entries from 1 to 0 before flushing the TLB.

If, after another scan, an entry is still not found, the process sleeps on the pkmap_map_wait wait queue until it is woken up after the next kunmap().

Once a mapping has been created, the corresponding entry in the pkmap_count array is incremented and the virtual address in low memory returned.

### 10.2.1 Unmapping Pages

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. Once established that it is a page to be unmapped, `kunmap_high()` is called to perform the unmapping.

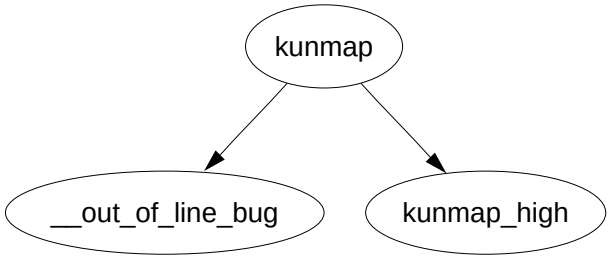


Figure 10.2: Call Graph: `kunmap()`

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 as a slot is now available. The page is not unmapped from the page tables then as that would require a TLB flush. It is delayed until `flush_all_zero_pkmaps()` is called.

## 10.3 Mapping High Memory Pages Atomically

The use of `kmap_atomic()` is heavily 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`[3].

`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()` as 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()` is interesting as it will only clear the PTE with `pte_clear()` if debugging is enabled. It is considered unnecessary to bother unmapping atomic pages as the next call to `kmap_atomic()` will simply replace it making TLB flushes unnecessary.

[3]There are a total of six different uses for atomic kmaps on the x86.

## 10.4 Bounce Buffers

Bounce buffers are required for devices that cannot access the full range of memory available to the CPU. An obvious example of this is when a device does not address with as many bits as the CPU, such as 32 bit devices on 64 bit architectures or recent Intel processors with PAE enabled.

The basic concept is very simple. A bounce buffer resides in memory low enough for a device to copy from and write data to. It is then copied to the desired user page in high memory. This additional copy is undesirable, but unavoidable. Pages are allocated in low memory which are used as buffer pages for DMA to and from the device. This is then copied by the kernel to the buffer page in high memory when IO completes so the bounce buffer acts as a type of bridge. There is significant overhead to this operation as at the very least it involves copying a full page but it is insignificant in comparison to swapping out pages in low memory.

### 10.4.1 Disk Buffering

Blocks, typically around 1KiB are packed into pages and managed by a `struct buffer_head` allocated by the slab allocator. A user of buffer heads has the option of having a callback function registered in the `buffer_head` as `b_end_io()` called when IO completes. It is this mechanism that bounce buffers uses to have data copied out of the bounce buffers. The callback registered is the function `bounce_end_io_write()`.

Any other feature of buffer heads or how they are used by the block layer is beyond the scope of this document and more the concern of the IO layer.

### 10.4.2 Creating Bounce Buffers

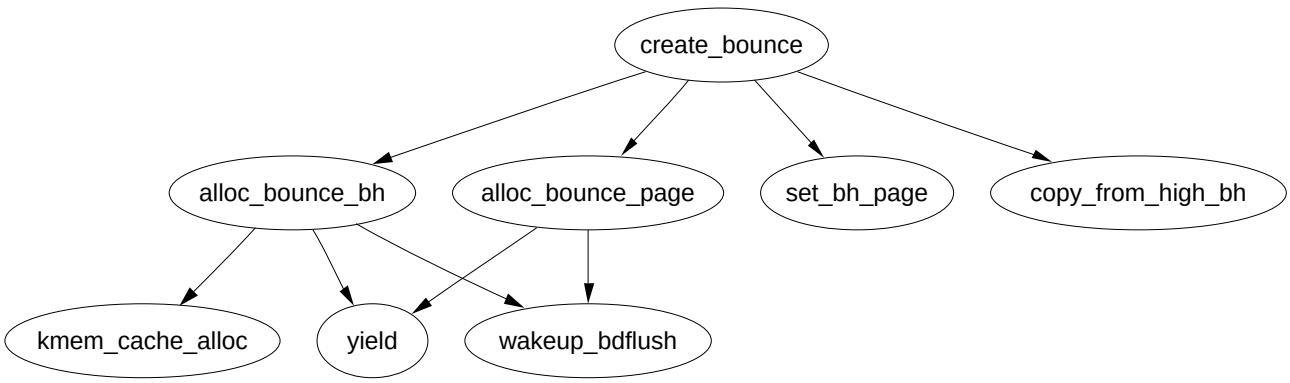


Figure 10.3: Call Graph: `create_bounce()`

The creation of a bounce buffer is a simple affair which is started by the `create_bounce()` function. The principle is very simple, create a new buffer using

a provided buffer head as a template. The function takes two parameters which are a read/write parameter (`rw`) and the template buffer head to use (`bh_orig`).

A page is allocated for the buffer itself with the function `alloc_bounce_page()` which is a wrapper around `alloc_page()` with one important addition. If the allocation is unsuccessful, there is an emergency pool of pages and buffer heads available for bounce buffers. This is discussed further in Section 10.5.

The buffer head is, predictably enough, allocated with `alloc_bounce_bh()` which, similar in principle to `alloc_bounce_page()`, calls the slab allocator for a `buffer_head` and uses the emergency pool if one cannot be allocated. Additionally, **bdflush** is woken up to start flushing dirty buffers out to disk so that buffers are more likely to be freed soon.

Once the page and `buffer_head` have been allocated, information is copied from the template `buffer_head` into the new one. Since part of this operation may use `kmap_atomic()`, bounce buffers are only created with the IRQ safe `io_request_lock` held. The IO completion callbacks are changed to be either `bounce_end_io_write()` or `bounce_end_io_read()` depending on whether this is a read or write buffer so the data will be copied to and from high memory.

The most important aspect of the allocations to note is that the GFP flags specify that no IO operations involving high memory may be used[4]. This is important as bounce buffers are used for IO operations with high memory. If the allocator tries to perform high memory IO, it will recurse and eventually crash.

### 10.4.3 Copying via bounce buffers

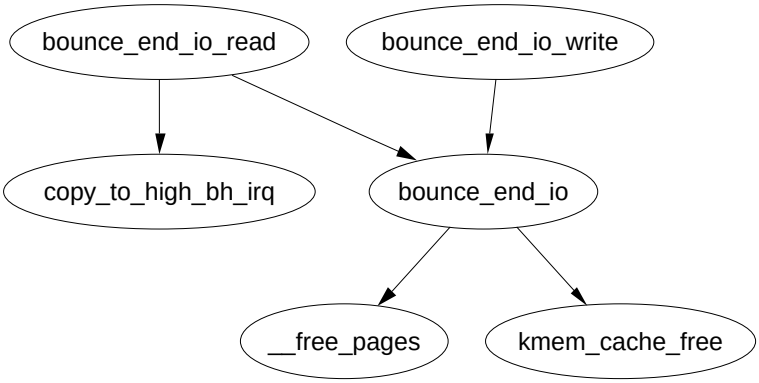


Figure 10.4: Call Graph: `bounce_end_io_read/write()`

Data is copied via the bounce buffer differently depending on whether it is a read or write buffer. If the buffer is for writes to the device, the buffer is populated with the data from high memory during bounce buffer creation with the function

[4]This is specified with `SLAB_NOHIGHIO` to the slab allocator and `GFP_NOHIGHIO` to the buddy allocator.

copy_from_high_bh(). The callback function bounce_end_io_write() will complete the IO later when the device is ready for the data.

If the buffer is for reading from the device, no data transfer may take place until the device is ready. When it is, the interrupt handler for the device calls the callback function bounce_end_io_read() which copies the data to high memory with copy_to_high_bh_irq().

In either case the buffer head and page may be reclaimed by bounce_end_io() once the IO has completed and the IO completion function for the template buffer_head() is called. If the emergency pools are not full, the resources are added to the pools otherwise they are freed back to the respective allocators.

## 10.5 Emergency Pools

Two emergency pools of buffer_heads and pages are maintained for the express use by bounce buffers. If memory is too tight for allocations, failing to complete IO requests is going to compound the situation as buffers from high memory cannot be freed until low memory is available. This leads to processes halting, thus preventing the possibility of them freeing up their own memory.

The pools are initialised by init_emergency_pool() to contain POOL_SIZE[5] entries each. The pages are linked via the page→list field on a list headed by emergency_pages. Figure 10.5 illustrates how pages are stored on emergency pools and acquired when necessary.

buffer_heads are very similar as they linked via the buffer_head→inode_buffers on a list headed by emergency_bhs. The number of entries left on the pages and buffer lists are recorded by two counters nr_emergency_pages and nr_emergency_bhs respectively and the two lists are protected by the emergency_lock spinlock.

.

[5]Currently defined as 32.

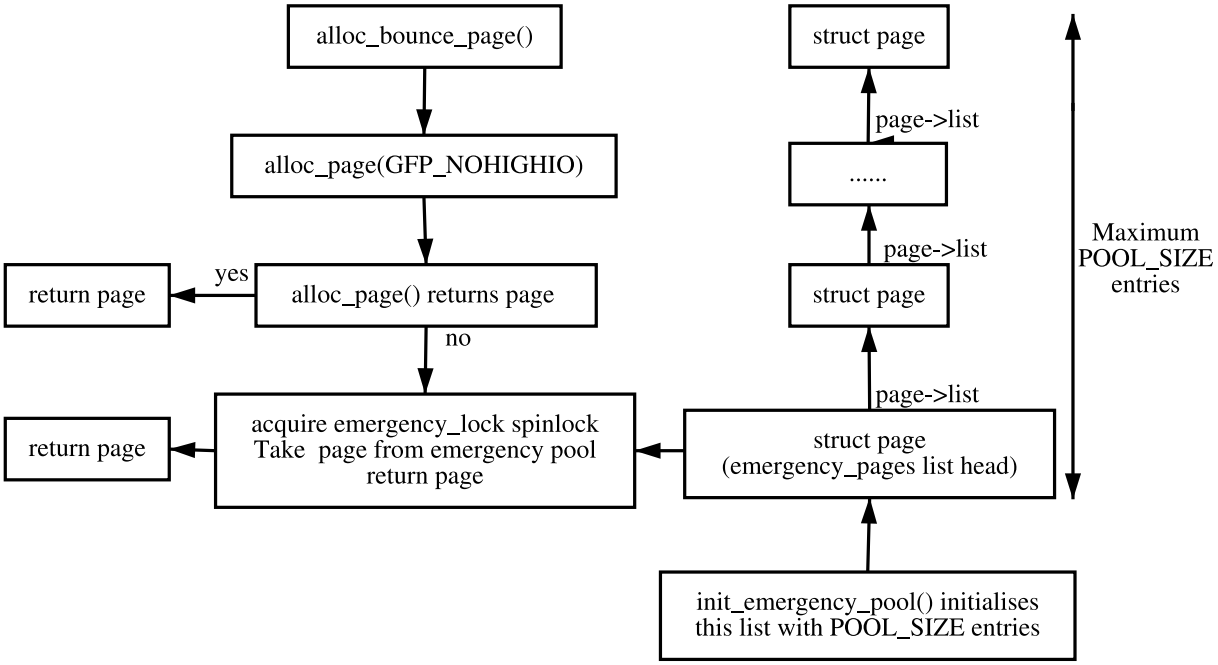


Figure 10.5: Acquiring Pages from Emergency Pools

# Chapter 11

# Page Frame Reclamation

A running system will eventually use all page frames for purposes like disk buffers, *dentries*, inode entries and process pages and so Linux needs to select old pages which can be freed and invalidated for new uses before physical memory is exhausted. This chapter will focus exclusively on how Linux implements its page replacement policy and how different types of pages are invalidated.

The methods Linux uses to select pages is rather empirical in nature and the theory behind the approach is based on multiple different ideas. It has been shown to work well in practice and adjustments are made based on user feedback and benchmarks.

With the exception of the slab allocator, all pages in use by the system are stored on the page cache and linked together via the `page→lru` so they can be easily scanned for replacement. The slab pages are not stored within the page cache as it is considerably more difficult to age a page based on the objects used by the slab.

Process mapped pages are not easily swappable as there is no way to map `struct page`s to PTEs except to search every page table which is far too expensive. If the page cache has a large number of process pages in it, process page tables will be walked and pages swapped out by `swap_out()` until enough pages have been freed but this will still have trouble with shared pages. If a page is shared, a swap entry is allocated, the PTE filled with the necessary information to find the page again and the reference count decremented. Only when the count reaches zero will the page be actually swapped out. These types of shared pages are considered to be in the *swap cache*.

This chapter begins with the pageout daemon **kswapd** and what its task is. From there we introduce the page replacement policy that Linux implements before an in-depth discussion on the *Page Cache* which is the core structure which determines how pages are removed from memory. Finally we cover how pages mapped by processes, which have to be treated specially, are swapped out.

## 11.1 Pageout Daemon (kswapd)

At system start, a kernel thread called **kswapd** is started from `kswapd_init()` which continuously executes the function `kswapd()` in `mm/vmscan.c` which usually sleeps. This daemon is responsible for reclaiming 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 3.2.1).

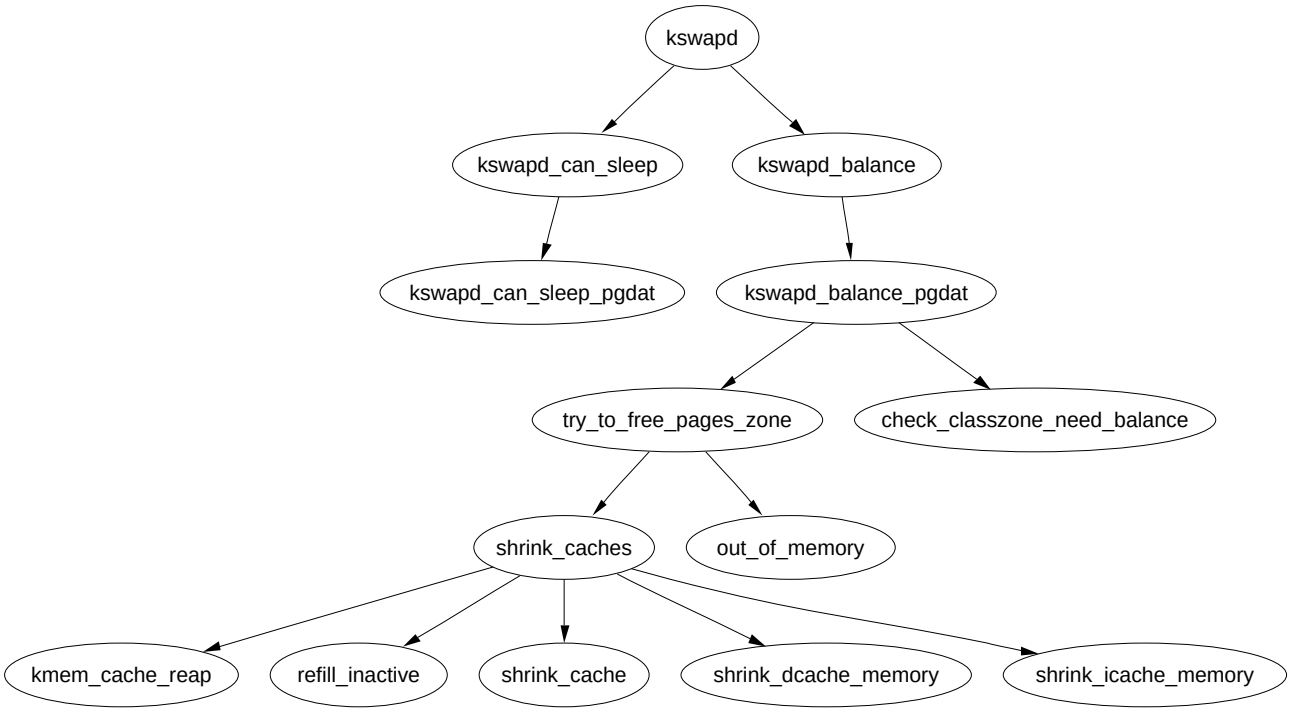


Figure 11.1: Call Graph: `kswapd()`

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 is woken up with increasing frequency as 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()`. The physical page allocator will also call `try_to_free_pages_zone()` when the zone it is allocating from is under heavy pressure.

When **kswapd** is woken up, it performs the following:

- Calls `kswapd_can_sleep()` which cycles through all zones checking the `need_balance` field 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;
- `kswapd_balance()` is called 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;
- Add **kswapd** back to the `kswapd_wait` queue and go back to the first step.

## 11.2 Page Cache

The page cache is a list of pages that are backed by regular files, block devices or swap. There are basically four types of pages that exist in the cache:

- 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 fit depends on the size of the block and the page size of the architecture;
- Anonymous pages first enter the page cache with no backing storage but are allocated slots in the backing storage when the kernel needs to swap them out, discussed furhter in Chapter 12;
- Pages belonging to shared memory regions which 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.

Pages exist in this cache for two reasons. The first is to eliminate unnecessary disk reads. Pages read from disk are stored in a *page hash* table hashed on the `struct address_space` and the offset. This table is always searched before the disk is accessed. The second reason is that the page cache forms the queue as a basis for the page replacement algorithm to select which page to discard or pageout.

The cache collectively consists of two lists defined in `mm/page_alloc.c` called `active_list` and `inactive_list` which broadly speaking store the "hot" and "cold" pages respectively. The lists are protected by the `pagemap_lru_lock`. An API is provided that is responsible for manipulating the page cache which is listed in Table 11.1.

### 11.2.1 Page Cache Hash Table

As stated, 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 initialisation 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

$$\text{htable\_size} = \text{num\_physpages} * \text{sizeof(struct page} *)$$

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.

$$\text{order} = \log_2((\text{htable\_size} * 2) - 1))$$

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:

$$\text{page\_hash\_bits} = \log_2 \left| \frac{\text{PAGE\_SIZE} * 2^{\text{order}}}{\text{sizeof(struct page} *)} \right|$$

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.

### 11.2.2 Inode Queue

The *inode queue* is part of the `struct address_space` introduced in Section 5.4.1. The struct contains three lists: `clean_pages` is a list of clean pages associated with the inode; `dirty_pages` which have been written to since the list sync to disk; and `locked_pages` which 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 removed with `remove_page_from_inode_queue()`.

## 11.3 Manipulating the Page Cache

This section begins with how pages are added to the page cache. It will then cover how pages are moved from the `active_list` to the `inactive_list`. Lastly we will cover how pages are reclaimed from the page cache.

### 11.3.1 Adding Pages to the Page Cache

Pages which are read from a file or block device are added to the page cache by calling `__add_to_page_cache()` during `generic_file_read()`.

All filesystems use the high level function `generic_file_read()` so that operations will take place through the page cache. It calls `do_generic_file_read()` which first checks if the page exists in the page cache. If it does not, the information is read from disk and added to the cache with `__add_to_page_cache()`.

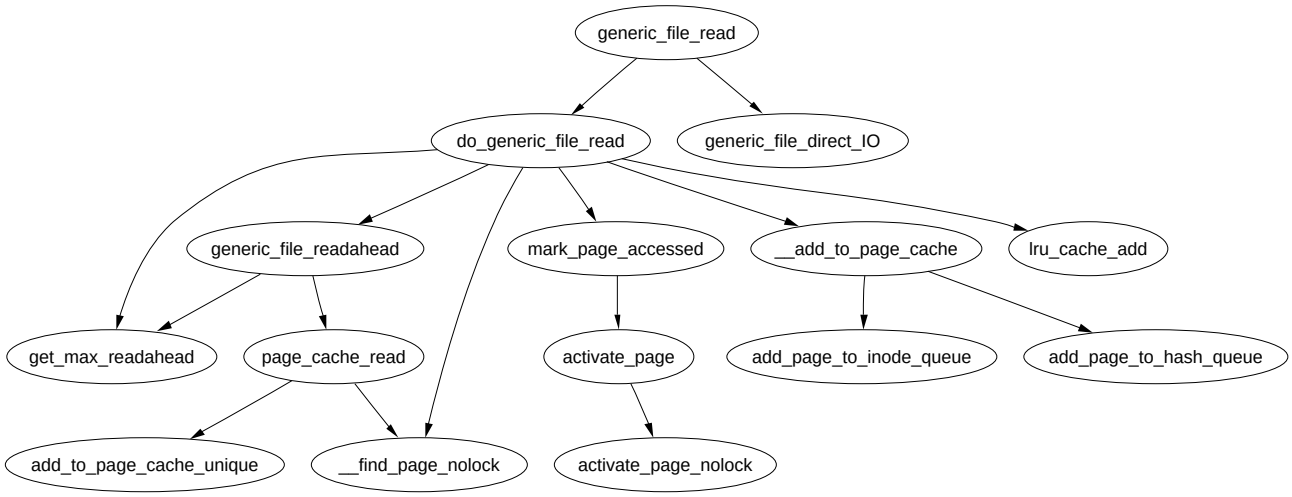


Figure 11.2: Call Graph: `generic_file_read()`

Anonymous pages are added to the page cache the first time they are about to be swapped out and will be discussed further in Section 12.4. The only real difference between anonymous pages and file backed pages as far as the page cache is concerned is that anonymous pages will use `swapper_space` as the `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 as 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()`.

### 11.3.2 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 number of pages to move, which is calculated in `shrink_caches()` as a ratio depending on `nr_pages`, the number of pages in `active_list` and the number of pages in `inactive_list`. The number of pages to move is calculated as

$$\text{pages} = \text{nr\_pages} * \frac{\text{nr\_active\_pages}}{2 * (\text{nr\_inactive\_pages} + 1)}$$

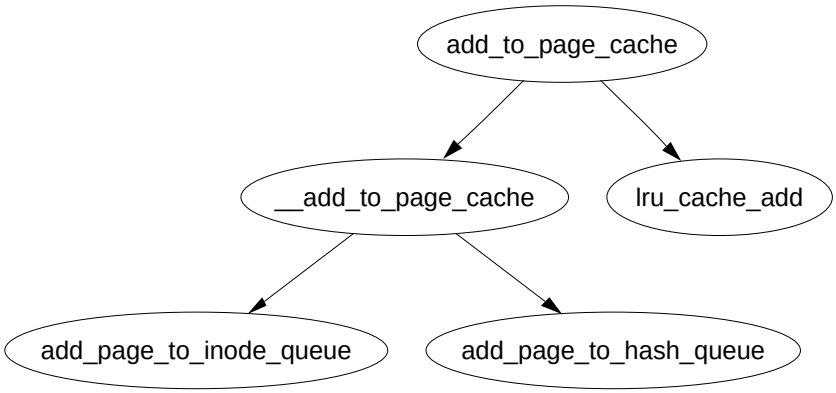


Figure 11.3: Call Graph: `add_to_page_cache()`

This keeps the `active_list` about two thirds the size of the `inactive_list` and the number of pages to move is determined as a ratio based on how many pages we desire to swap out (`nr_pages`).

Pages are taken from the end of the `active_list`. If the `PG_referenced` flag is set, it is cleared and the page is put back at top of the `active_list` as it has been recently used and is still "hot". If the flag is cleared, it is moved to the `inactive_list` and the `PG_referenced` flag set so that it will be quickly promoted to the `active_list` if necessary.

### 11.3.3 Reclaiming Pages from the Page Cache

The function `shrink_cache()` is the part of the replacement algorithm which takes pages from the `inactive_list` and decides how they should be swapped out. The two starting parameters which determine how much work will be performed are `nr_pages` and `priority`. `nr_pages` starts out as `SWAP_CLUSTER_MAX` and priority starts as `DEF_PRIORITY`.

Two parameters, `max_scan` and `max_mapped` determine how much work the function will do and are affected by the `priority`. Each time the function `shrink_caches()` is called without enough pages being freed, the priority will be decreased until the highest priority 1 is reached.

`max_scan` is the maximum number of pages will be scanned by this function and is simply calculated as

$$\text{max\_scan} = \frac{\text{nr\_inactive\_pages}}{\text{priority}}$$

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

$$\text{max\_mapped} = \text{nr\_pages} * 2^{(10-\text{priority})}$$

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 which 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 monopolise the CPU.

For each type of page found on the list, it makes a different decision on what to do. The page types and actions are as follows:

*Page is mapped by a process.* 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()`

*Page is locked and the PG_launder bit is set.* A reference to the page is taken with `page_cache_get()` so that the page will not disappear and `wait_on_page()` is called which sleeps until the IO is complete. Once it is completed, the reference count is decremented with `page_cache_release()`. When the count reaches zero, it is freed.

*Page is dirty, is unmapped by all processes, has no buffers and belongs to a device or file mapping.* The `PG_dirty` bit is cleared and the `PG_launder` bit is set. A reference to the page is taken with `page_cache_get()` so the page will not disappear prematurely and then the `writepage()` function provided by the mapping is called to clean the page. The last case will pick up this page during the next pass and wait for the IO to complete if necessary.

*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, the page can be freed. If it is backed by a file or device, the reference is simply dropped and the page will be freed later. However it is unclear how a page could have both associated buffers and a file mapping.

*Page is anonymous belonging to a process and has no associated buffers.* The LRU is unlocked and the page is unlocked. The `max_mapped` count is decremented. If it reaches zero, then `swap_out()` is called to start swapping out entire processes as there are too many process mapped pages in the page cache. An anonymous page may have associated buffers if it is backed by a swap file. In this case, the page is treated as a buffer page and normal block IO syncs the page with the backing storage.

*Page has no references to it.* If the page is in the swap cache, it is deleted from it as it is now stored in the swap area. If it is part of a file, it is removed from the inode queue. The page is then deleted from the page cache and freed.

## 11.4 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. The maximum number of pages that will be written to disk in any given pass is `nr_pages` which is initialised by `try_to_free_pages_zone()` to be `SWAP_CLUSTER_MAX`[1]. The limitation is there so that if **kswapd** schedules a large number of pages to be swapped to disk, it will sleep occasionally to allow the IO 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` initialised by `try_to_free_pages_zone()` to be `DEF_PRIORITY`[2]. For each pass that does not free up enough pages, the priority is decremented for the highest priority been 1.

The function first calls `kmem_cache_reap()` (see Section 9.1.7) 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.

## 11.5 Swapping Out Process Pages

When `max_mapped` pages have been found in the page cache, `swap_out()` 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 which 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`.

Once it has been decided to swap out pages from a process, an attempt will be made to swap out at least `SWAP_CLUSTER` number of pages and the full list of `mm_struct`s 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.

`swap_mm` is initialised to point to `init_mm` and the `swap_address` is initialised to 0 the first time it is used. A task has been fully searched when the `swap_address`

[1] Currently statically defined as 32 in `mm/vmscan.c`.

[2] `DEF_PRIORITY` is currently statically defined as 6 in `mm/vmscan.c`.

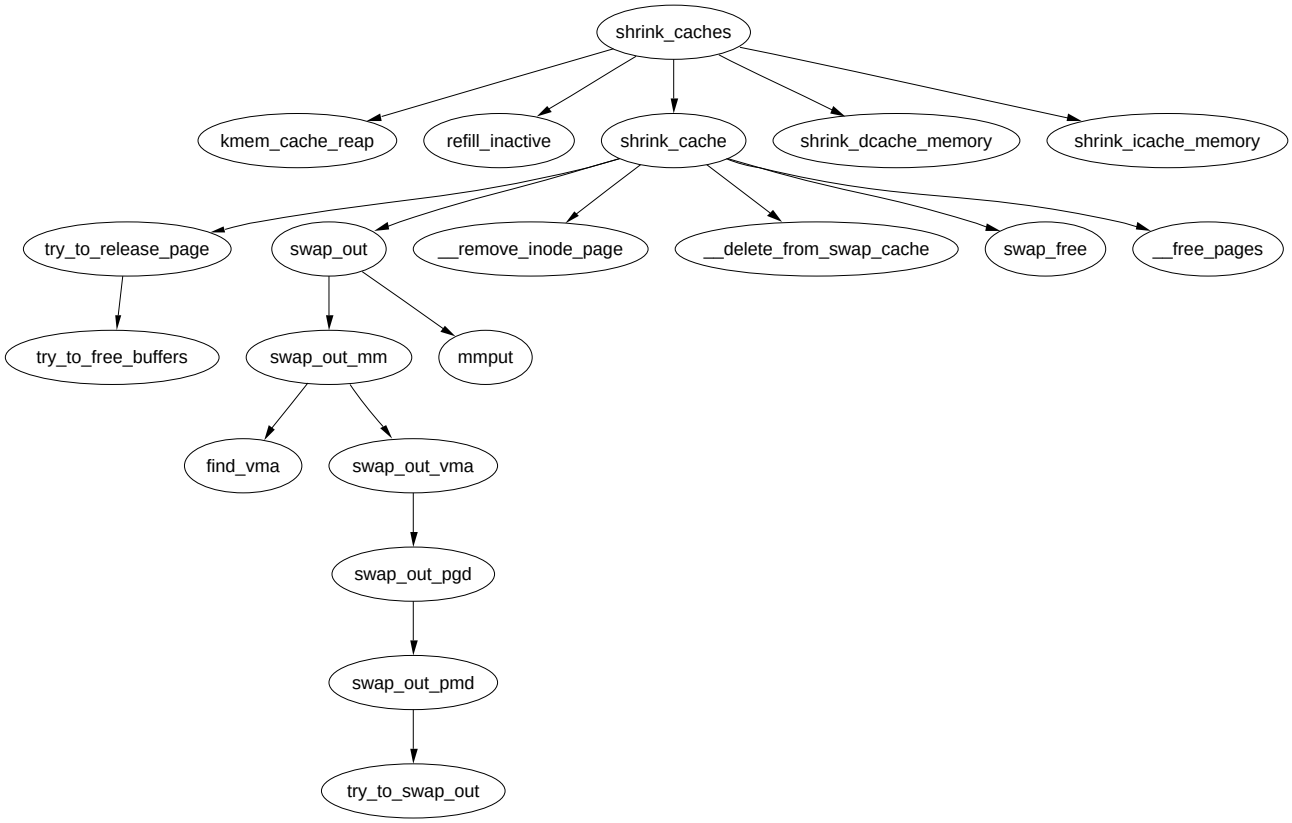


Figure 11.4: Call Graph: `shrink_caches()`

is equal to `TASK_SIZE`. Once 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. This function walks each VMA the process holds and calls `swap_out_vma()` for it. This is to avoid having to walk the entire page table which will be largely sparse. `swap_out_pgd()` and `swap_out_pmd()` walk the page tables for given VMA until finally `try_to_swap_out()` is called on the actual page and PTE.

`try_to_swap_out()` first checks to make sure the page is not part of the `active_list`, been recently referenced or part of a zone that we are not interested in. Once it has been established this is a page to be swapped out, it is removed from the page tables of the process and further work is performed. It is at this point the PTE is checked to see if it is dirty. If it is, the `struct page` flags will be updated to reflect that so that it will get laundered. Pages with buffers are not handled further as they can not be swapped out to backing storage so the PTE for the process is simply established again and the page will be flushed later.

The process of allocating space in the backing storage and swapping pages out is discussed further in Chapter 12.

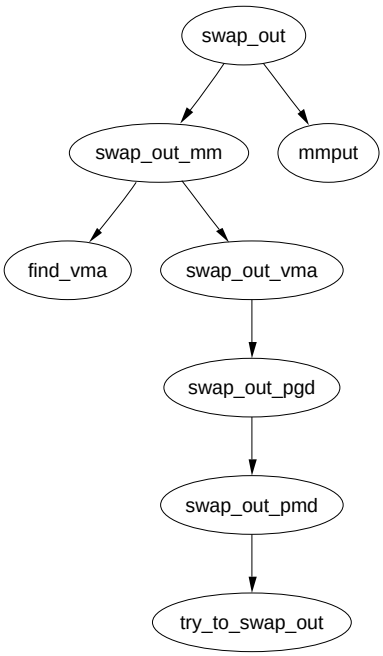


Figure 11.5: Call Graph: `swap_out()`

## 11.6 Page Replacement Policy

During discussions the page replacement policy is frequently said to be a *Least Recently Used (LRU)*-based algorithm but this is not strictly speaking true as the lists are not strictly maintained in LRU order. The objective is for the `active_list` to contain the *working set* [Den70] of all processes and the `inactive_list`. As all reclaimable pages are contained in just two lists and all pages may be selected to reclaimed rather than just the faulting process, the replacement policy is a global one.

The lists resemble a simplified LRU 2Q [JS94] where two lists called `Am` and `A1` are maintained. With LRU 2Q, pages when first allocated are placed on a FIFO queue called `A1`. If they are referenced while on that queue, they are placed in a normal LRU managed list called `Am`. 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` (Am). 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 11.6 illustrates how the two lists are structured.

The lists described for 2Q presumes Am is an LRU list but the list in Linux closer resembles a Clock algorithm [Car84] where the hand-spread is the size of the active list. When pages reach the bottom of the list, the referenced flag is checked, if it is set, it is moved back to the top of the list and the next page checked. If it is cleared, it is moved to the `inactive_list`.

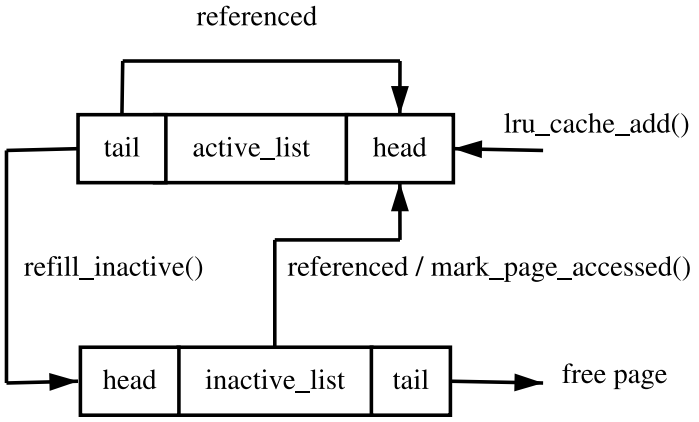


Figure 11.6: Page Cache LRU List

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 analysing multi-programmed systems [CD80] and the fact the memory size for each process is not fixed , the policy does not satisfy the *inclusion property* as the location of pages in the lists depend heavily upon the size of the lists as opposed to the time of last reference. Neither is the list priority ordered as 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 as 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 file 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` as the ratio between the two lists sizes remains not change significantly.

`add_to_page_cache(struct page * page, struct address_space * mapping, unsigned long offset)`
Adds a page to the page cache with `lru_cache_add()` in addition to adding it to the inode queue and page hash tables. Important for pages backed by files on disk.

`remove_inode_page(struct page *page)`
This function removes a page from both the inode queue with `remove_page_from_inode_queue()` and from the hash queues with `remove_page_from_hash_queue()`. This effectively removes the page from the page cache.

`lru_cache_add(struct page * page)`
Add a cold page to the `inactive_list`. Will be followed by `mark_page_accessed()` if known to be a hot page, such as when a page is faulted in.

`lru_cache_del(struct page *page)`
Removes a page from the page cache by calling either `del_page_from_active_list()` or `del_page_from_inactive_list()`, whichever is appropriate.

`mark_page_accessed(struct page *page)`
Mark that the page has been accessed. If it was not recently referenced (in the `inactive_list` and `PG_referenced` flag not set), the referenced flag is set. If it is referenced a second time, `activate_page()` is called, which marks the page hot, and the referenced flag is cleared

`page_cache_get(struct page *page)`
Increases the reference count to a page already in the page cache

`page_cache_release(struct page *page)`
An alias for `__free_page()`. The reference count is decremented and if it drops to 0, the page will be freed

`activate_page(struct page * page)`
Removes a page from the `inactive_list` and places it on `active_list`. It is very rarely called directly as the caller has to know the page is on the inactive list. `mark_page_accessed()` should be used instead

Table 11.1: Page Cache API

# Chapter 12

# 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 as such as "swapping" refers to coping an entire process address space to disk and "paging" to copying out portions or 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 principle reasons that the existence of swap space is desirable. First, it expands the amount of memory a process may use. Virtual memory and swap space allows a large process to run even if the process is only partially resident. As "old" pages may be swapped out, the amount of memory addressed may easily exceed RAM as demand paging will ensure the pages are reloaded if necessary.

The casual reader[1] may think that with a sufficient amount of memory, swap is unnecessary but this brings us to the second reason. A significant number of the pages referenced by a process early in its life may only be used for initialisation and then never used again. It is better to swap out those pages and create more disk buffers than leave them resident and unused.

It is important to note that 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 11, 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. We will start with how Linux describes a swap area.

This chapter begins with describing the structures Linux maintains about each

[1]Not to mention the affluent reader.

active swap area in the system and how the swap area information is organised on disk. We then cover how Linux remembers how to find pages in the swap after they 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.

## 12.1 Describing the Swap Area

Each active swap area, be it a file or partition, has a struct `swap_info_struct` describing the area. All the structures 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 is set to `SWP_WRITEOK` when Linux is ready to write to the area as 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 file, this is NULL;

**sdev_lock** As with many structures 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 file that is mounted as a swap area. This could be the `dentry` for a file in the `/dev/` directory for example in the case a partition is mounted. This field is needed to identify the correct `swap_info_struct` when deactiating a swap area;

**vfs_mount** This is the `vfs_mount` object corresponding to where the device or file 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. 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` flag when using **swapon**;

**pages** As some slots on the swap file 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 pseudo list called `swap_list` which is a very simple type declared as follows in <`linux/swap.h`>:

```
154 struct swap_list_t {
155         int head;    /* head of priority-ordered swapfile list */
156         int next;    /* swapfile to be used next */
157 };
```

The `head` is the swap area of the highest priority swap area in use and the `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 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 4096 bytes on the x86 for example. The first slot is always reserved as 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 filled when the swap area is created with the system program **mkswap**. The information is used to fill in 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 is. If the string is "SWAP-SPACE", it is version 1 of the swap file 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` fills 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 which varies if the struct changes but it is 637 entries in its current form as given by the simple equation;

$$\text{MAX\_SWAP\_BADPAGES} = \frac{\text{PAGE\_SIZE} - 1024 - 512 - 10}{\text{sizeof(long)}}$$

Where 1024 is the size of the bootblock, 512 is the size of the padding and 10 is the size of the magic string identifing the format of the swap file.

## 12.2 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 offset within the `swap_map` and 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.

In the `swp_entry_t`, two bits are always kept free which are used by Linux to determine if a PTE is present or swapped out. 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 4.2.

Bits 1-6 are for the *type* which is the index within the `swap_info` array and are returned by the `SWP_TYPE()` macro.

Bits 8-31 are used are to store the *offset* within the `swap_map` from the `swp_entry_t`. On the x86, this means 24 bits are available, "limiting" the size of the swap area to 64GiB. The macro `SWP_OFFSET()` is used to extract the offset.

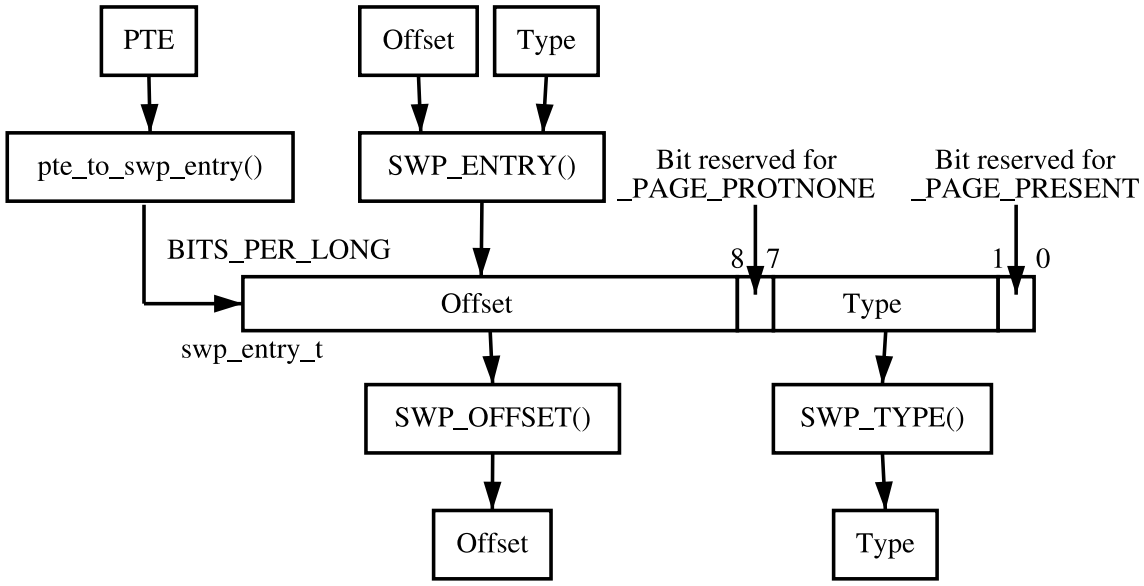


Figure 12.1: Storing Swap Entry Information in `swp_entry_t`

To encode a type and offset 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 12.1.

It should be noted that 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 probably due to the consumption of the vmalloc address space. If a swap area is the maximum possible size then 32MiB is required for the `swap_map` ($2^{24} * \text{sizeof(short)}$); remember that each page uses one short for the reference count. For just `MAX_SWAPFILES` 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 supporting 64 swap areas is not worth the additional complexity but there is cases where a large number of swap areas would be desirable even if the overall swap available does not increase. Some modern machines[2] 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 which 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.

## 12.3 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

[2] A Sun E450 could have in the region of 20 disks in it for example.

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.

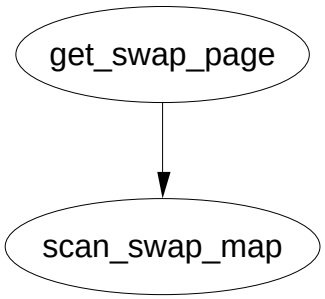


Figure 12.2: Call Graph: get_swap_page()

The task of finding and allocating a swap entry is divided into two major tasks. The first performed by the high level function get_swap_page(). Starting with swap_list→next, it searches swap areas for a suitable slot. Once a slot has been found, it records what the next swap area to be used will be and returns the allocated entry.

The task of searching the map is the responsibility of scan_swap_map(). In principle, it is very simple as it linearly scan the array for a free slot and return. Predictably, the implementation is a bit more thorough.

Linux attempts to organise pages into *clusters* on disk of size SWAPFILE_CLUSTER. It allocates SWAPFILE_CLUSTER number of pages sequentially in swap keeping count of the number of sequentially allocated pages in swap_info_struct→cluster_nr and records the current offset in swap_info_struct→cluster_next. Once a sequential block has been allocated, it searches for a block of free entries of size SWAPFILE_CLUSTER. If a block large enough can be found, it will be used as another cluster sized sequence.

If no free clusters large enough can be found in the swap area, a simple first-free search starting from swap_info_struct→lowest_bit is performed. The aim is to have pages swapped out at the same time close together on the premise that pages swapped out together are related. This premise, which seems strange at first glance, is quite solid when it is considered that the page replacement algorithm will use swap space most when linearly scanning the process address space swapping out pages. Without scanning for large free blocks and using them, it is likely that the scanning would degenerate to first-free searches and never improve. With it, processes exiting are likely to free up large blocks of slots.

## 12.4 Swap Cache

Pages that are shared between many processes can not be easily swapped out because, as mentioned, there is no quick way to map a `struct page` to every PTE that references it. This leads to the race condition where a page is present for one PTE and swapped out for another gets updated without being synced to disk thereby losing the update.

To address this problem, shared pages that have a reserved slot in backing storage are considered to be part of the *swap cache*. The swap cache is purely conceptual as there is no simple way to quickly traverse all the pages on it and there is no dedicated list but pages that exist on the page cache that have a slot reserved in backing storage are members of it. This means that anonymous pages, by default, are not part of the swap cache *until* an attempt is made to swap them out. It also means that by default, pages that belong to a shared memory region are added to the swap cache when they are first written to.

A page is identified as being part of the swap cache once the `page→mapping` field has been set to `swapper_space` which is the `address_space` struct managing the swap area. This condition is tested with the `PageSwapCache()` macro. Linux uses the exact same logic for keeping pages between swap and memory in sync as it uses for keeping pages belonging to files and memory coherent. The principal difference is that instead of using an `struct address_space` tied to a filesystem, `swapper_space` is associated which has registered functions for writing to swap space. The second difference is that instead of using `page`index to mark an offset within a file, it is used to store the `swp_entry_t` structure.

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 12.3 and the call graph is shown in Figure 12.4.

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 finally 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 11.

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 incremented and the swap slot decremented with

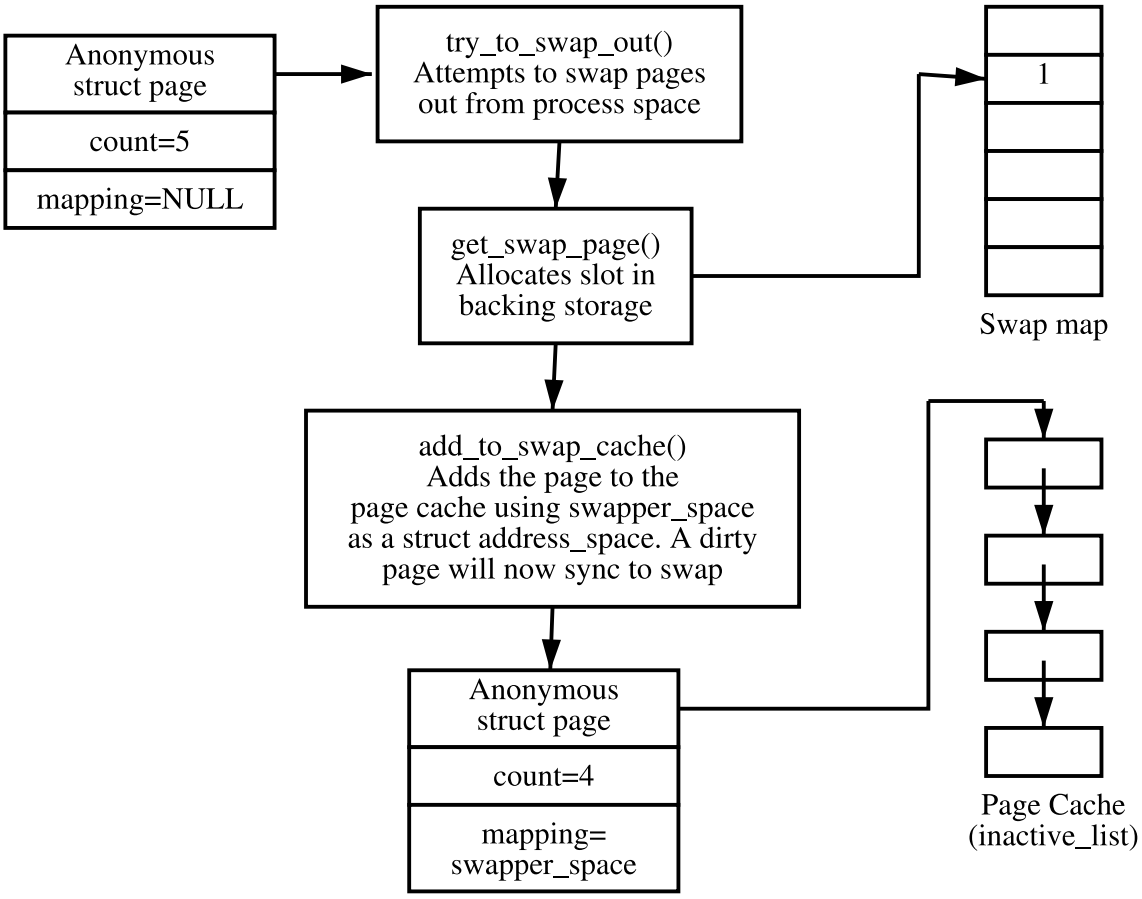


Figure 12.3: Adding a Page to the Swap Cache

`swap_free()`.

## 12.5 Activating a Swap Area

As it has now been covered 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 been activated, the *Big Kernel Lock (BKL)* is held which prevents any application entering kernel space while this operation is been performed. The function is quite large but can be broken down into the following simple steps;

- Find a free `swap_info_struct` in the `swap_info` array an initialise it with default values
- Call `user_path_walk()` which traverses the directory tree for the supplied `specialfile` and populates a `namidata` structure with the available data on

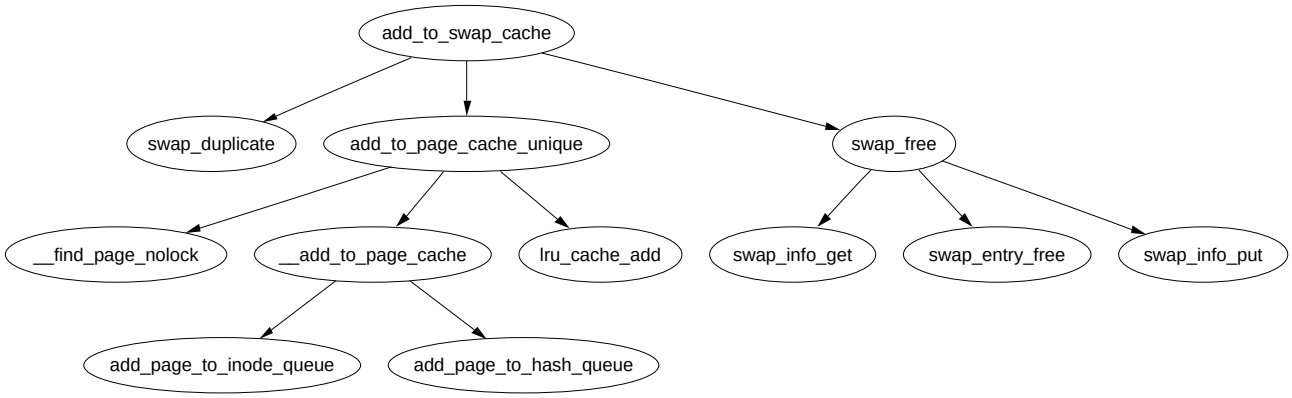


Figure 12.4: Call Graph: `add_to_swap_cache()`

the file, such as the `dentry` and the filesystem information for where it is stored (`vfsmount`)

- 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`
- 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
- Allocate memory with `vmalloc()` for `swap_info_struct→swap_map` and initialise each entry with 0 for good slots and `SWAP_MAP_BAD` otherwise. Ideally the header information will be a version 2 file format as version 1 was limited to swap areas of just under 128MiB for architectures with 4KiB page sizes like the x86[3]
- 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`
- The swap area is now fully active and initialised and 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.

[3]See the Code Commentary for the comprehensive reason for this.

| |
|---|
| `get_swap_page()`<br>This function allocates a slot in a `swap_map` by searching active swap areas. This is covered in greater detail in Section 12.3 but included here as it is principally used in conjunction with the swap cache |
| `add_to_swap_cache(struct page *page, swp_entry_t entry)`<br>This function adds a page to the swap cache. It first checks if it already exists by calling `swap_duplicate()` and if not, is adds it to the swap cache via the normal page cache interface function `add_to_page_cache_unique()` |
| `lookup_swap_cache(swp_entry_t entry)`<br>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` offset |
| `swap_duplicate(swp_entry_t entry)`<br>This function verifies a swap entry is valid and if so, increments its swap map count |
| `swap_free(swp_entry_t entry)`<br>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 12.1: Swap Cache API

## 12.6 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 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 which 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 broadly speaking;

- Call `user_path_walk()` to acquire the information about the special file to be deactivated and then take the BKL
- 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`. Once this is acquired, the BKL can be released again
- 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()` referencing a page in the event it still somehow has a reference in swap map
- If there was not enough available memory to page in all the entries, the swap area is reinserted back into the running system as it cannot be simply dropped. If it succeeded, the `swap_info_struct` is placed into an uninitialised state and the `swap_map` memory freed with `vfree()`

## 12.7 Swapping In Pages

The principal function used when reading in pages is `read_swap_cache_async()` which is called during page faulting for instance. This function is called as it first searches the swap cache with `find_get_page()` and returns it if it does. If it does not already exist, a new page is allocated with `alloc_page()`, it is added to the swap cache with `add_to_swap_cache()` and finally the IO is started with `rw_swap_page()` with flags to start the read operation which is covered in detail later.

## 12.8 Swapping Out Pages

Pages are written out to disk when the pages in the swap cache are laundered. To launder a page, the `address_space→a_ops` is consulted to find the appropriate write-out function. In the case of swap, the `address_space` is `swapper_space` and the swap operations are contained in `swap_aops`. The registered write-out function is `swap_writepage()`.

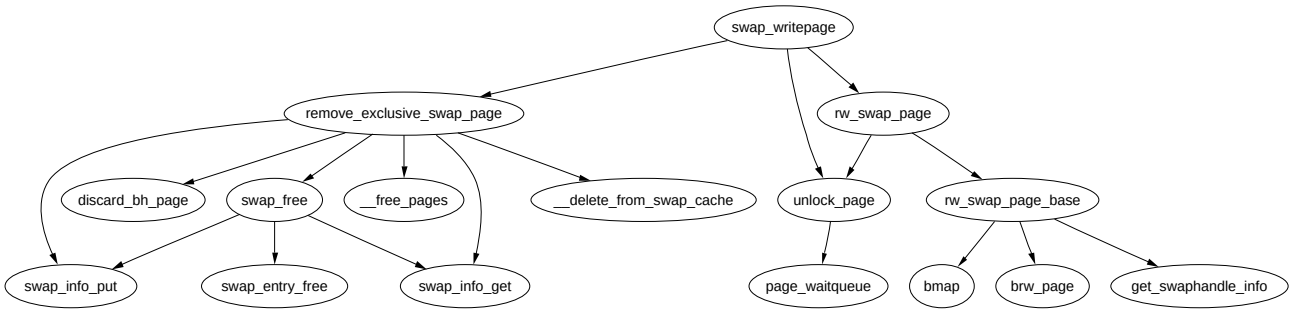


Figure 12.5: Call Graph: `sys_writepage()`

`swap_writepage()` makes a decision based on what is returned by the function `remove_exclusive_swap_page()`. `remove_exclusive_swap_page()` checks if there is any other processes referencing the requested page page in the swap cache by examining the page count with the `pagecache_lock` held. If no other process does, the page is removed from the swap cache and freed.

If the page was removed from the swap cache, 1 is returned to `swap_writepage()` which will unlock the page waking any process that was waiting on it. If the page still exists in the swap cache, 0 is returned and `rw_swap_page()` is called to write the contents of the page out to backing storage.

## 12.9 Reading/Writing the Swap Area

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 which performs the real work.

It begins by checking if the operation is a read. If it is, it clears the uptodate flag with `ClearPageUptodate()`. 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 file. These are needed before block IO operations may be performed.

If the swap area is a file, `bmap()` is used to fill a local array with a list of all blocks in the filesystem which contain the page being operated on. Remember that filesystems may have their own method of storing files and disk and it is not as simple as the swap partition where information may be written directly to disk.

Once that is complete, a normal block IO operation takes place with `brw_page()`. The function of block IO is beyond the scope of this document.

# Chapter 13

# Out Of Memory Management

When the machine is low on memory, old page frames will be reclaimed (see Chapter 11) but during the process is may find it was unable to free enough pages to satisfy a request even when scanning at highest priority. If it does fail to free page frames, `out_of_memory()` is called to see if the system is out of memory and needs to kill a process.

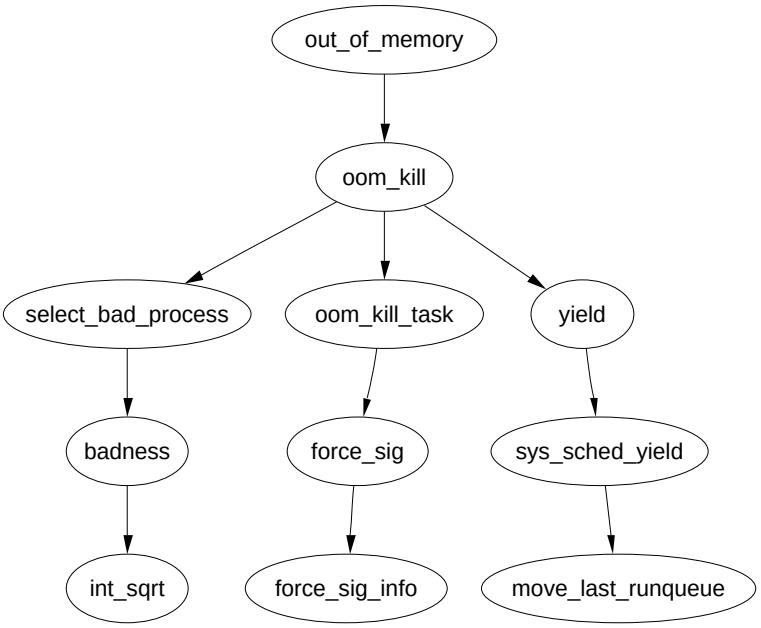


Figure 13.1: Call Graph: `out_of_memory()`

Unfortunately, it is possible that the system is not out memory and simply needs to wait for IO to complete or for pages to be swapped to backing storage so before deciding to kill a process, it goes through the following checklist.

- Is there enough swap space left (`nr_swap_pages` $> 0$) ? If yes, not OOM
- Has it been more than 5 seconds since the last failure? If yes, not OOM

- Have we failed within the last second? If no, not OOM
- If there hasn't been 10 failures at least in the last 5 seconds, we're not OOM
- Has a process been killed within the last 5 seconds? If yes, not OOM

It is only if the above tests are passed that `oom_kill()` is called to select a process to kill.

## 13.1 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, note that the square roots are integer approximations calculated with `int_sqrt()`;

$$\text{badness\_for\_task} = \frac{\text{total\_vm\_for\_task}}{\sqrt{(\text{cpu\_time\_in\_seconds})} * \sqrt[4]{(\text{cpu\_time\_in\_minutes})}}$$

This has been chosen to select a process that is using a large amount of memory but is not that long lived. Processes which 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 as 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 4 as it is undesirable to kill a process that has direct access to hardware.

## 13.2 Killing the Selected Process

Once 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.

# Chapter 14

# Conclusion

The field of memory management is large, complex, time consuming to research and difficult to apply to practical implementations. This is partially related to the difficulty of modeling how systems behave in real multi-programmed systems [CD80] which has resulted in theoretical examination of virtual memory algorithms often depending on simulations of specific workloads. Simulations are necessary as 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 as it is only ever shown to work well for specified 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.

Linux 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 first implementations are developed based on the all-important foundation that theory provides. Contributors built upon this framework with changes based on real world observations but unfortunately the best available documentation [BC00] of the final implementation tries to summarise the entire kernel without giving specific focus to any area.

It has been often asserted on the Linux Memory Management mailing list that the VM is poorly documented and difficult to pick up as "the implementation is a nightmare to follow"[1] and the lack of documentation on practical VMs is not just confined to Linux. Matt Dillon, one of the principal developers of the FreeBSD VM[2] and considered a "VM Guru" stated in an interview[3] that documentation can be "hard to come by". One of the principal difficulties with deciphering the implementation is the fact the developer must have a background in memory management theory to see why implementation decisions were made as a pure understanding of the code is insufficient for any purpose other than micro-optimisations.

[1] *http://mail.nl.linux.org/linux-mm/2002-05/msg00035.html*

[2] His 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*

This thesis attempts to bridge the gap between memory management theory and the practical implementation in Linux and tie both fields together into a single body of work. I believe this thesis is the most comprehensive documentation of the Linux VM to date and is a rare attempt to bind memory management theory and practice together. Rather than providing a general view of the kernel, it has been presented from the perspective of the VM in a manner that is relatively independent of hardware architecture considerations.

A future direction for this research includes the documentation of the, as yet unreleased, version 2.6 VM and the development of VM Regress as a regression, benchmarking and analysis tool for which a framework is still being developed. With this document as a starting point, it is envisioned that the version 2.6 VM can be documented as a series of addenda documents describing the differences between version 2.4.20 and version 2.6 without the necessity of presenting the entire VM as this document has done. On a personal note, I hope that this document encourages other researches to produce similar documents for other subsystems in the kernel.

# Bibliography


[AST97] Andrew S. Woodhull Andrew S. Tanenbaum. *Operating Systems Design and Implementation, 2nd Edition*. Prentice-Hall, 1997.

[BA01] Jeff Bonwick and Jonathan Adams. Magazines and vmem: Extending the slab allocator to many CPUs and arbitrary resources. In *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX-01)*, pages 15–34, Berkeley, CA, June 25–30 2001. The USENIX Association.

[BBD+98] Michael Beck, Harold Bohme, Mirko Dzladzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux Kernel Internals*. Addison-Wesley, 1998.

[BC00] D. (Daniele) Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly, 2000.

[BC03] D. (Daniele) Bovet and Marco Cesati. *Understanding the Linux kernel (2nd Edition)*. O'Reilly, 2003.

[BL89] R. Barkley and T. Lee. A lazy buddy system bounded by two coalescing delays. In *Proceedings of the twelfth ACM symposium on Operating Systems principles*. ACM Press, 1989.

[Bon94] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[Car84] Rickard W. Carr. *Virtual Memory Management*. UMI Research Press, 1984.

[CD80] E. G. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice-Hall Inc., 1980.

[CH81] R. W. Carr and J. L. Hennessy. WSClock - A simple and effective algorithm for virtual memory management. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 87–95, Pacific Grove, CA, December 1981. Association for Computing Machinery.

[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 Charles Severance. *High Performance Computing, 2nd Edition*. O'Reilly, 1998.

[Den70] Peter J. Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[GAV95] A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In ACM, editor, *Conference proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain, July 3–7, 1995*, pages 338–347, New York, NY 10036, USA, 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.

[Gor02] Mel Gorman. *Code Commentry on the Linux Virtual Memory Manager*. Unpublished, 2002.

[Haca] Various Kernel Hackers. *Kernel 2.2.22 Source Code*. ftp://ftp.kernel.org/pub/linux/kernel/v2.2/linux-2.2.22.tar.gz.

[Hacb] Various Kernel Hackers. *Kernel 2.4.18 Source Code*. ftp://ftp.kernel.org/pub/linux/kernel/v2.4/linux-2.4.18.tar.gz.

[Hac00] Random Kernel Hacker. How to get your change into the linux kernel. *Kernel Source Documentation Tree (SubmittingPatches)*, 2000.

[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)*, volume 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, 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, 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.* Addison-Wesley, Reading, Mass., 1968.

[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, Arthur E. Oldehoeft. *Operating Systems, Advanced Concepts.* Benjamin/Cummings Publishing, 1987.

[MM01] Richard McDougall and Jim Maura. *Solaris Internals.* Rachael Borden, 2001.

[Ous90] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Usenix 1990 Summer Conference*, pages 247–256, jun 1990.

[PN77] James L. Peterson and Theodore A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[RC01] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers, 2nd Edition.* O'Reilly, 2001.

[Rus00] Paul Rusty Russell. Unreliable guide to locking. *Kernel Source Documentation Tree*, 2000.

[Sho75] John E. Shore. On the external storage fragmentation produced by first-fit and best-fit allocation strategies. *Communications of the ACM*, 18(8):433–440, 1975.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems, 2nd Edition.* Prentice-Hall, 2001.

[Vah96] Uresh Vahalia. *UNIX Internals.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 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.

# Index