

Understanding The Linux Virtual Memory Manager

Mel Gorman

22nd January 2003

Contents

1	Introduction	8
1.1	Thesis Overview	9
2	Code Management	10
2.1	Managing the Source	10
2.2	Getting Started	16
2.3	Submitting Work	17
3	Describing Physical Memory	19
3.1	Nodes	20
3.2	Zones	22
3.3	Pages	24
4	Page Table Management	29
4.1	Describing the Page Directory	29
4.2	Describing a Page Table Entry	31
4.3	Using Page Table Entries	32
4.4	Translating and Setting Page Table Entries	34
4.5	Allocating and Freeing Page Tables	35
4.6	Initialising Kernel Page Tables	36
5	Boot Memory Allocator	38
5.1	Representing the Boot Map	38
5.2	Initialising the Boot Memory Allocator	39
5.3	Allocating Memory	41
5.4	Freeing Memory	43
5.5	Retiring the Boot Memory Allocator	43
6	Physical Page Allocation	47
6.1	Allocator API	47
6.2	Managing Free Blocks	47
6.3	Allocating Pages	49
6.4	Free Pages	51
6.5	GFP Flags	52
6.6	Avoiding Fragmentation	53

7	Non-Contiguous Memory Allocation	57
7.1	Kernel Address Space	57
7.2	Describing Virtual Memory Areas	58
7.3	Allocating A Non-Contiguous Area	59
7.4	Freeing A Non-Contiguous Area	60
8	Slab Allocator	62
8.1	Caches	64
8.2	Slabs	72
8.3	Objects	79
8.4	Sizes Cache	80
8.5	Per-CPU Object Cache	82
8.6	Slab Allocator Initialisation	85
8.7	Interfacing with the Buddy Allocator	85
9	Process Address Space	88
9.1	Managing the Address Space	88
9.2	Process Address Space Descriptor	90
9.3	Memory Regions	95
9.4	Exception Handling	109
9.5	Page Faulting	110
9.6	Copying To/From Userspace	114
10	High Memory Management	119
11	Page Frame Reclamation	120
11.1	Page Swap Daemon (kswapd)	120
11.2	Page Cache	122
11.3	Shrinking all caches	123
11.4	Page Hash	123
11.5	Inode Queue	123
11.6	Refilling inactive_list	123
11.7	Reclaiming pages from the page cache	124
11.8	Swapping Out Process Pages	126
12	Swap Management	129
12.1	Swap Cache	129
13	Out Of Memory Management	130

List of Figures

2.1	Example Patch	13
3.1	Relationship Between Nodes, Zones and Pages	20
4.1	Linear Address Bit Size Macros	30
4.2	Linear Address Size and Mask Macros	30
4.3	Page Table Layout	32
4.4	Call Graph: paging_init	36
5.1	Call Graph: setup_memory	40
5.2	Call Graph: __alloc_bootmem	41
5.3	Call Graph: mem_init	44
6.1	Free page block management	49
6.2	Call Graph: alloc_pages	50
6.3	Allocating physical pages	50
6.4	Call Graph: __free_pages	51
7.1	Kernel Address Space	58
7.2	VMalloc Address Space	59
7.3	Call Graph: vmalloc	59
7.4	Call Graph: vfree	60
8.1	Layout of the Slab Allocator	63
8.2	Call Graph: kmem_cache_create	70
8.3	Call Graph: kmem_cache_reap	70
8.4	Call Graph: kmem_cache_shrink	72
8.5	Call Graph: __kmem_cache_shrink	72
8.6	Call Graph: kmem_cache_destroy	73
8.7	Page to Cache and Slab Relationship	74
8.8	Slab With Descriptor On-Slab	75
8.9	Slab With Descriptor Off-Slab	76
8.10	Call Graph: kmem_cache_grow	77
8.11	initialised kmem_bufctl_t Array	77
8.12	Call Graph: kmem_slab_destroy	79
8.13	Call Graph: kmem_cache_alloc	80

8.14	Call Graph: kmalloc	82
8.15	Call Graph: kfree	82
9.1	Data Structures related to the Address Space	89
9.2	Memory Region Flags	98
9.3	Call Graph: sys_mmap2	102
9.4	Call Graph: get_unmapped_area	103
9.5	Call Graph: insert_vm_struct	104
9.6	Call Graph: sys_mremap	105
9.7	Call Graph: move_vma	106
9.8	Call Graph: move_page_tables	106
9.9	Call Graph: sys_mlock	107
9.10	Call Graph: do_munmap	108
9.11	Call Graph: do_page_fault	112
9.12	do_page_fault Flow Diagram	116
9.13	Call Graph: handle_mm_fault	117
9.14	Call Graph: do_no_page	117
9.15	Call Graph: do_swap_page	118
9.16	Call Graph: do_wp_page	118
11.1	Call Graph: kswapd	121
11.2	Page Cache LRU List	122
11.3	Call Graph: shrink_caches	124
11.4	Call Graph: swap_out	126
13.1	Call Graph: out_of_memory	130

List of Tables

1.1	Kernel size as an indicator of complexity	8
3.1	Flags Describing Page Status	27
3.2	Macros For Testing, Setting and Clearing Page Status Bits	28
4.1	Page Table Entry Protection and Status Bits	32
5.1	Boot Memory Allocator API for UMA Architectures	45
5.2	Boot Memory Allocator API for NUMA Architectures	46
6.1	Physical Pages Allocation API	48
6.2	Physical Pages Free API	49
6.3	Low Level GFP Flags Affecting Zone Allocation	52
6.4	Low Level GFP Flags Affecting Allocator Behavior	53
6.5	Low Level GFP Flag Combinations For High Level	54
6.6	High Level GFP Flags Affecting Allocator Behavior	55
6.7	Process Flags Affecting Allocator Behavior	56
7.1	Non-Contiguous Memory Allocation API	60
7.2	Non-Contiguous Memory Free API	61
8.1	Slab Allocator API for caches	86
8.2	Internal cache static flags	86
8.3	Cache static flags set by caller	87
8.4	Cache static debug flags	87
8.5	Cache Allocation Flags	87
9.1	System Calls Related to Memory Regions	90
9.2	Functions related to memory region descriptors	94
9.3	Memory Region VMA API	96
9.4	Reasons For Page Faulting	111
9.5	Accessing Process Address Space API	115
11.1	Page Cache API	128

Abstract

The development of Linux is unusual in that it was built more with an emphasis on the practical rather than a theoretical one. While many of the algorithms used in the Virtual Memory (VM) system were designed by theorists, the implementations have diverged from the theory considerably. Instead of following the traditional development cycle of design to implementation, changes are made in reaction to how the system behaved in the “real world” and intuitive decisions by developers.

This has led to a situation where the VM is poorly documented except for a few general overviews in a small number of books or websites and is fully understood only by a small number of core developers. Developers looking for information on how it functions are generally told to read the source. This requires that even a casual observer invest a large amount of time to read the code. The problem is further compounded by the fact that the code only tells the developer what is happening in a very small instance making it difficult to see how the overall system functions which 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. 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 a later VM than this document describes is of interest to the reader as the time needed to understand new changes to the VM is considerably less than what is needed to learn how it works to begin with.

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 grows but unfortunately as a necessary side effect, so does its size and complexity. The table in Figure 1.1 shows the total gzipped 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 a strong metric for complexity but still serves as a small indicator.

Version	Release Date	Tar Size	Size of mm/	Line count
1.0	March 13th, 1992	1.2MiB	96k	3109
1.2.13	February 8th, 1995	2.2MiB	136k	4531
2.0.39	January 9th 2001	7.2MiB	204k	6792
2.2.22	September 16th, 2002	14.0MiB	292k	9554
2.4.20	November 28th, 2002	32.0MiB	520k	15428

Table 1.1: Kernel size as an indicator of complexity

As is the habit of Open Source projects in general, new developers are sometimes told to refer to the source with the polite acronym RTFS¹ when questions are asked or are referred to the kernel newbies mailing list (<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 into 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 the operating system such as ‘*Understanding the Linux Kernel*’[BC00], tend to be an overview of all subsystems without giving specific attention to one topic with the notable exception of device drivers[RC01]. Increasingly, to get a comprehensive view on how the kernel

¹Read The Flaming Source

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 from the papers describing them considerably.

The documentation on the Memory Manager 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. In this thesis a comprehensive guide to the VM as implemented in the late 2.4 Kernels is given. A companion document called “*Code Commentary On The Linux Virtual Memory Manager*”, hereafter referred to as the *companion document*, provides a detailed tour of the code. It is envisioned that with this pair of documents, the time required to have a working understanding of the VM, even later VM’s, will be measured in weeks instead of the estimated 8 months currently required by even an experienced developer.

1.1 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 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. All the call graphs shown in these two documents were generated with it. 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 CVS² or BitKeeper³ is not always an option. With this tool, a simple spec file can be provided specifying 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.

²<http://www.cvshome.org/>

³<http://www.bitmover.com>

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 answer has to be found by each developer on their own.

The advice that is often given to new developers is to read books on operating systems, on Linux specifically, visit the kernel newbies website 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 analyzing 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.

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 <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 is the preferred method of upgrading for bandwidth considerations. Contributions made to the kernel are almost always in the form of patches which is basically the output of *unified diff* generated with 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 principle 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 distributed as a large patch to the mainline. These subtrees generally contain features or cleanups which have not been merged to the mainstream yet or are still being tested. Two notable subtrees is 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 BitKeeper¹, a proprietary version control system that was designed with the Linux Kernel as the principle consideration. BitKeeper allows developers to have their own distributed version of the tree and other users may “pull” changesets, sets of patches 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 wars² with free software developers, email and patches are still considered the only way to generate discussion on code changes. In fact, some patches will simply not be considered for merging unless some discussion on the main mailing list is observed. 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.

¹<http://www.bitmover.com>

²A regular feature of kernel discussions meaning an acrimonious argument often containing insults bordering on the personal type

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 website³. **diff** is used to generate patches and **patch** is used to apply them. While the tools may be used in a wide variety of ways, there is a “preferred” usage.

Patches generated with diff should always be unified diffs and generated from one directory above the kernel source root. A unified diff is considered the easiest context diff to read as it provides what line numbers the block begins at, how long it lasts and then it marks lines with +, - or a 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 examples, 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 new lines added are clearly marked with a + . In a patch, there may be several “hunks” which are marked with a line starting with @@ . Each hunk 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. Broadly speaking, 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
mel@joshua: linux-2.4.20-mel/ $ patch -p1 < ../example.patch
patching file mm/page_alloc.c
mel@joshua: linux-2.4.20-mel/ $
```

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
```

³<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

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. Generally, related 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 header files. 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 does not work well when there is multiple functions of the same name which is the case for architecture code or if a type of variable needs to be identified.

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 snippets shown in this and the companion document were taken from LXR so that the line numbers would be visible.

2.1.3 Analyzing 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 Devera⁴, I developed a tool called **gen-graph**. 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, cdep files 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 project⁵.

In kernel 2.4.20, there were a total of 14593 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 a function as an argument and generates a call graph with the requested function at the top. 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 X 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

⁴<http://luzik.cdi.cz/~devik>

⁵<http://www.graphviz.org>

a function, but not traverse it which is convenient when the function is covered on a separate call graph.

All call graphs shown in this or the companion document are generated with the *gengraph* package which is freely available at <http://www.csn.ul.ie/~mel/projects/gengraph>. 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
```

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 config 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, this is usually 1. The second is the name of the patch stored in the patches directory. The above example has two patches to update the kernel to 2.4.20 before applying 2.4.20-rmap15a.

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* 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* 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 init code and work from there. I do not believe that this is the best approach as the init code 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 organized 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 FAQ and recommended reading.

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.

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 7 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 6 for similar reasons. The fourth system of interest is the creation of VMA's and memory areas for processes discussed in Chapter 9. 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 frame work 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**⁶. 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 "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.

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 list⁷.

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

⁶<http://www.kernel.org/pub/linux/kernel/people/rusty/trivial/>

⁷<http://www.linux-mm.org/maillinglists.shtml>

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.

Chapter 3

Describing Physical Memory

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

With large scale machines, memory may be arranged into banks that incur a different cost to use depending on processor. For example, there might be a bank of memory assigned to each CPU or a bank of memory very suitable for DMA. These banks are said to be at varying distances and exist on architectures referred to as NUMA (Non-Uniform Memory Access) architectures.

In Linux, each bank is called a *node* and is represented by `struct pg_data_t`. Every node in the system is kept on a NULL terminated list called `pgdat_list`. Each node is linked to the next with the field by `pg_data_t→node_next`. For UMA architectures like PC desktops only one static `pg_data_t` structure called `contig_page_data` is used.

Each node is then divided up into a number of blocks called zones which represent ranges within memory. 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 ranges which certain ISA devices require. `ZONE_NORMAL` is memory that can be directly mapped by the kernel in the upper 1GiB of the linear address space. 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. With the x86 the zones are

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

Each physical page frame is represented by a `struct page` and all the structs are kept in `mem_map` array that is stored at `PAGE_OFFSET`, the beginning of the virtual address space the kernel can see, 3GiB on the x86. The pages are stored in this area so that the physical address for a page struct may be easily calculated. Each zone has a pointer within this array called `zone_mem_map`. Note that a `struct page` is not the actual physical page.

The high memory extensions allow the kernel to address up to 64GiB in the-

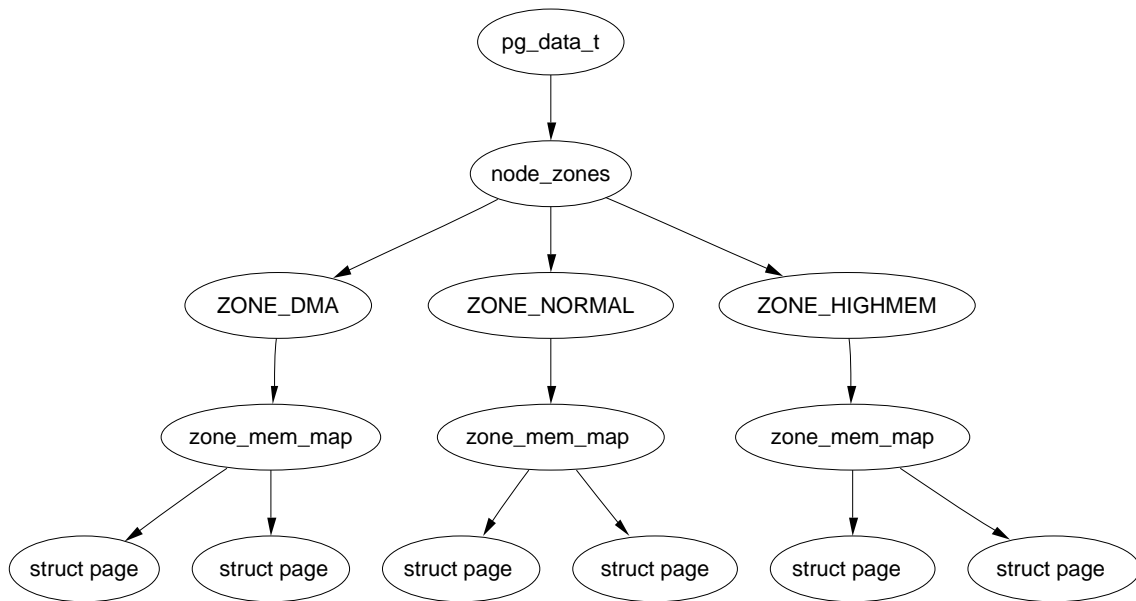


Figure 3.1: Relationship Between Nodes, Zones and Pages

ory but in practice it can't. The struct pages to describe each page frame requires 44 bytes and this uses kernel virtual address space in `ZONE_NORMAL`. That means to describe 1GiB of memory, approximately 11MiB of kernel memory is required. At 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 PTEs require about 16MiB in the worst case. This makes 16GiB about the practical limit for physical memory on an x86.

The relationship between the structs mentioned so far is described in Figure 3.1.

3.1 Nodes

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 or can be explicitly bound, it is likely the memory from the current node will be used.

The struct is declared as follows in `include/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;

```

node_zones The zones for this node, usually 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` does the work when called by `free_area_init_core()`. So a failed allocation 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 a struct page array representing each physical frame in the node.

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

node_start_paddr The starting physical address of the node. This doesn't work really well as an unsigned long as it breaks for ia32 with PAE for example. A more suitable solution would be to record this as a Page Frame Number (pfn). This 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` the the local `mem_map` for this node called `lmem_map`.

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 5.2.2. 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 make the code more readable.

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 *include/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;

```

lock Spinlock to protect the zone

free_pages Total number of free pages in the zone

pages_min, pages_low, pages_high Zone watermarks, described in the next section

need_balance A flag that tells 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

wait_table_shift Defined as the number of bits in a long minus the table size. When the hash is calculated, it will be shifted right this number of bits so that the hash index will be inside the table.

zone_pgdat Points to the parent `pg_data_t`

zone_mem_map The first page in `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 up pages (See Chapter 11). If memory gets too low, the process will free up memory synchronously. The parameters affecting pageout behavior are similar to those by FreeBSD[McK96] and Solaris[JM01].

Each zone has watermarks which help track how much pressure a zone is under. They are `pages_low`, `pages_min` and `pages_high`. 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 is twice the value of `pages_min`. 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.

pages_high is three times the value of **pages_min**. Once reached, **kswapd** is woken, it won't 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*.

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], the structure of this page resembled to some extent to System V[GC94] but like the other families in UNIX, it changed considerably. It is declared as follows in *include/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;
157     atomic_t count;
158     unsigned long flags;
159     struct list_head lru;
160     struct page **pprev_hash;
161     struct buffer_head * buffers;
162
163 #if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)
164     void *virtual;
165 #endif /* CONFIG_HIGHMEM || WANT_PAGE_VIRTUAL */
166 } mem_map_t;

```

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 lists 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 **mmaped**, their inode has an associated **address_space**. This field will point to this address space if the page belongs to the file.

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. This includes if the page is part of the swap cache where the **address_space** is 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 Flags which describe the status of the page. All of them are declared in `include/linux/mm.h` and are listed and described in Table 3.1. There is 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`

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

The struct `page` is a typedef for `mem_map_t` so the struct `page` 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 reference was stored to the zone at `page→zone` which was later considered wasteful. In the most recent kernels, this has been removed and instead the top `ZONE_SHIFT` (8 in the x86) bits of the `page→flags` is used to determine the zone a page belongs to. First a `zone_table` of zones is set up. It is declared in `include/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.

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's 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 <code>mem_init()</code>
PG_laundry	This bit is important only to the page replacement policy. When the VM wants to swap out a page, it will set this bit and call the <code>writepage()</code> function. When scanning, if it encounters a page with this bit and 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's 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 during init until the machine is booted up. 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
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	SetActive	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_laundry	SetPageLaundry	PageLaundry	ClearPageLaundry
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

Chapter 4

Page Table Management

Linux is unusual with how it layers the machine independent/dependent layer[CP99] as while many other operating systems such objects like the **pmap** object in BSD Linux instead always 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 Physical Address Extensions (PAE) mode enabled, only two page table levels are available. The Page Middle Directory (PMD) is defined to be of size 1 and folds back directly onto the Page Global Directory (PGD) which is optimized out at compile time. Unfortunately, for architectures that do not manage their cache or 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].

4.1 Describing the Page Directory

Each process has its own **PGD** (Page Global Directory) which is a physical page frame containing an array of **pgd_t** which is an architecture specific type defined in *include/asm/page.h*. How the page table is loaded is different for each architecture. On the x86, the process page table is loaded by copying the pointer into the **cr3** register which has the side effect of flushing the TLB and in fact is how the function `__flush_tlb()` is implemented in the architecture dependent code.

Each 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 a page frame containing Page Table Entries (PTE) of type **pte_t**, which in turn points to page frames containing 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 tables and finally as an offset within the actual page.

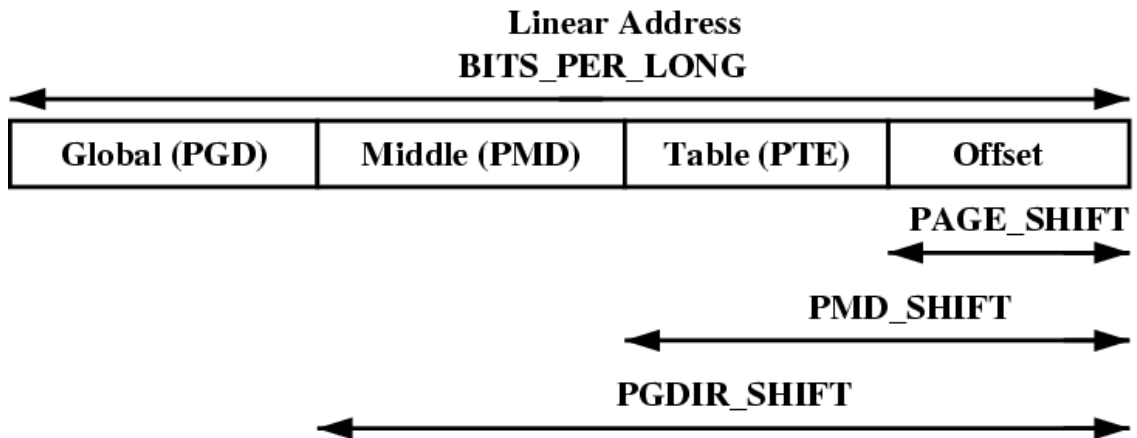


Figure 4.1: Linear Address Bit Size Macros

To help break up the linear address into its component parts, a number of macros are provided in triplets for each level, 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. The MASK values can be AND'd 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. Finally the SIZE macros reveal how many bytes are address by each entry at each level. The relationship between the SIZE and MASK macros is illustrated in Table 4.2.

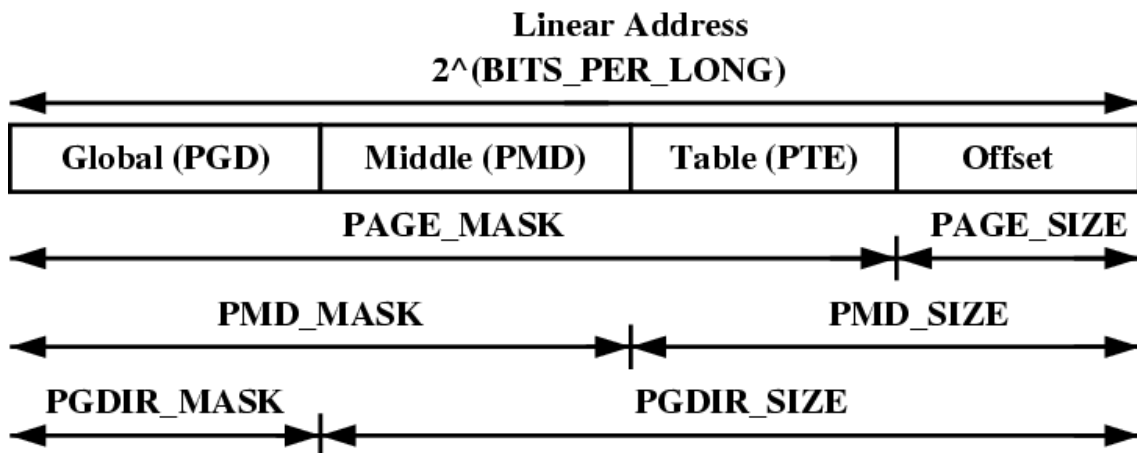


Figure 4.2: Linear Address Size and Mask Macros

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))
```

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 is easily calculated as 2^{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. To determine if an address is page aligned, it is simply AND'd with the **PAGE_MASK** which will yield 0 if it is aligned. To force an address to be page aligned, the **PAGE_ALIGN** function is used.

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 structs **pte_t**, **pmd_t** and **pgt_t** for PTEs, PMDs and PGDs respectively. Even though these are often just unsigned integers, they are defined as structs for two reasons. The first is for type protection so that they will not be used inappropriately. The second is for features like PAE on the x86 where an additional 4 bits may be 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 structs. 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. For this one, the **pte_t** is a 32 bit integer stored within a struct. Each entry in this points to the address of a page frame but all the addresses are guaranteed to be page aligned, therefore there is **PAGE_SHIFT** (12) bits in that 32 bit value that are free for status bits of

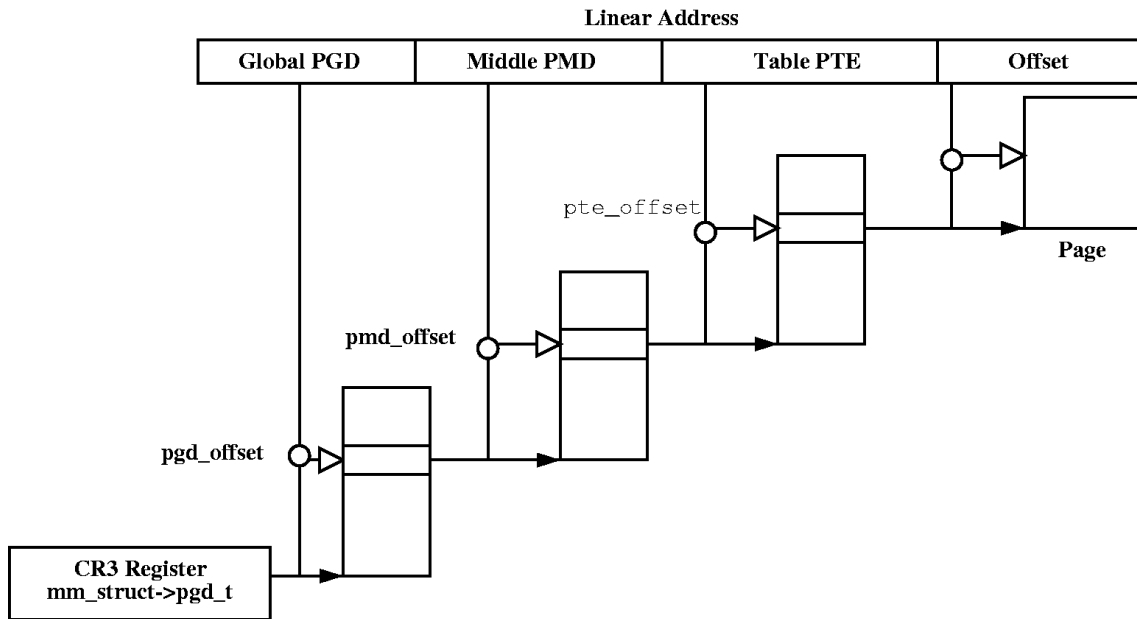


Figure 4.3: Page Table Layout

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 vary between architectures.

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

Table 4.1: Page Table Entry Protection and Status Bits

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 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 is the two most important checks.

There is many parts of the VM which are littered with page table walk code and it is important to recognize it. A very simple example of a page table walk is the function `follow_page()` in `mm/memory.c` which is as follows;

```

405 static struct page * follow_page(struct mm_struct *mm,
                                   unsigned long address, int write)
406 {
407     pgd_t *pgd;
408     pmd_t *pmd;
409     pte_t *ptep, pte;
410
411     pgd = pgd_offset(mm, address);
412     if (pgd_none(*pgd) || pgd_bad(*pgd))
413         goto out;
414
415     pmd = pmd_offset(pgd, address);
416     if (pmd_none(*pmd) || pmd_bad(*pmd))
417         goto out;
418
419     ptep = pte_offset(pmd, address);
420     if (!ptep)
421         goto out;
422
423     pte = *ptep;
424     if (pte_present(pte)) {
425         if (!write ||
426             (pte_write(pte) && pte_dirty(pte)))
427             return pte_page(pte);
428     }
429
430 out:

```

```

431         return 0;
432     }

```

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 page table walk had effectively ended at line 423.

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 to a userspace process.

- The read permissions for an entry is tested with `pte_read`, made readable with `pte_mkread` and protected with `pte_rdprotect`.
- The write permissions are tested with `pte_write`, made writable with `pte_mkwrite` and protected with `pte_wrprotect`.
- The exec permissions are tested with `pte_exec`, made executable with `pte_mkexec` and protected with `pte_exprotect`. It is worth noting that with the x86 architecture, there is no means of setting execute permissions on pages so these three macros act the same way as the read macros.
- The permissions can be modified to a new value with `pte_modify` but its use is almost 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 is only two states 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 and 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 PTE's and the setting the individual entries.

`mk_pte` takes a physical 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 treats the address as a physical address.

`pte_page` returns the `struct page` which corresponds to the PTE entry. `pmd_page` returns the `struct page` containing the set of PTE's.

`set_pte` takes a `pte_t` such as that returned by `mk_pte` and places it within the process's 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 across all of them are the same. For example, not all architectures bother to cache PGD's because the allocation and freeing of them is only during process creation and exit. As these are both very expensive operations, the allocation of another page will not make much of a difference.

PGD's, PMD's and PTE's 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 PMD's and PTE's 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 Section 6). The functions for the three levels of page tables are **get_pgd_slow**, **pmd_alloc_one** and **pte_alloc_one**.

Obviously a large number of pages could end up on these caches and so there is a way of pruning the size of the cache. 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 Initialising 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 which 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 initialise the rest of the page tables.

4.6.1 Bootstrapping

The assembler function `startup_32()` is responsible for enabling the paging unit in *arch/i386/kernel/head.S*. While all the normal kernel code in *vmlinux* is compiled with the base address at `PAGE_OFFSET + 1MiB` but the kernel is actually loaded beginning at the first megabyte (0x00100000) of memory¹. 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`.

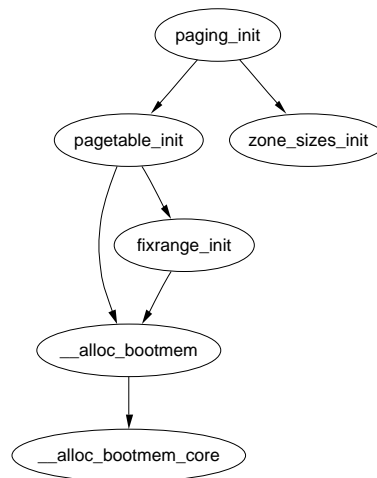


Figure 4.4: Call Graph: `paging_init`

It begins with statically defining an array called `swapper_pg_dir()` which is placed using directives at 0x00101000. It then established page table entries for 2 pages **pg0** and **pg1**. As the 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 be mapping to the correct pages using either physical or virtual addressing.

¹The first megabyte is used by some devices so is skipped

Once this mapping has been established, the paging unit is turned on by setting a bit in the cr0 register and a jmp takes places immediately to ensure the EIP register is correct.

4.6.2 Finalizing

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

For each `pgd_t` used by the kernel, the boot memory allocator 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 enabled 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, `swapper_pg_dir` is loaded again into the cr3 register and the TLB is flushed.

Chapter 5

Boot Memory Allocator

It is impractical to statically initialise all the core kernel memory structures at compile time as there is 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 strong 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 making it very wasteful[WJNB95].

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

5.1 Representing the Boot Map

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

```

node_boot_start is the starting physical address of the represented block

node_low_pfn is the end physical address, in other words, the end of the ZONE_NORMAL this node represents

node_bootmem_map is the location of the bitmap representing allocated or free pages with each bit

last_offset is the offset within the the page of the end of the last allocation. If 0, the page used is full

last_pos 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

5.2 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 `do_init_bootmem()` in the case of the PPC. Regardless the architecture name, the tasks are essentially the same. The parameters it needs to calculate are;

min_low_pfn is the lowest **Page Frame Number (PFN)** that is available in the system.

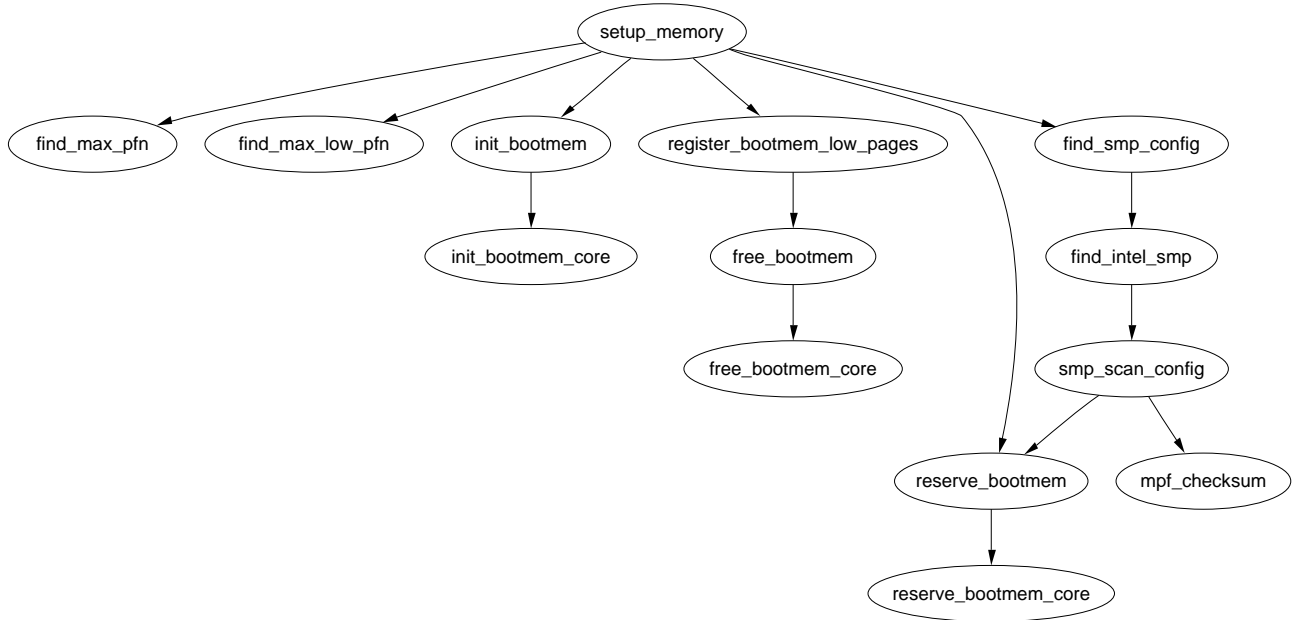
max_low_pfn is the highest PFN that may be addressed by low memory (ZONE_NORMAL)

highstart_pfn is the PFN of the beginning of high memory (ZONE_HIGHMEM)

highend_pfn is the ending PFN of high memory

max_pfn is the last PFN available to the system

5.2.1 Calculating The Size of Zones

Figure 5.1: Call Graph: `setup_memory`

The **Page Frame Number (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, it calls `find_max_pfn()` which reads through the whole e820 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 `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 5.5.

5.2.2 Initialising *bootmem_data*

Once the dimensions of usable physical memory is known, one of two *bootmem* initialise functions is 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 the 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 bytes¹ of the bitmap required is straightforward;

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

It stores the bitmap in the physical address `bootmem_data_t→node_boot_start` and the virtual address pointing to the map is at `bootmem_data_t→node_bootmem_map`. As there is no architecture independent way to detect “holes” in memory, the entire of the bitmap is initialised as 1 to mark 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 all of the e820 map and calling `free_bootmem()` for all usable pages to set the bit to 0 before calling `reserve_bootmem()` to reserve the pages needed by the actual bitmap.

5.3 Allocating Memory

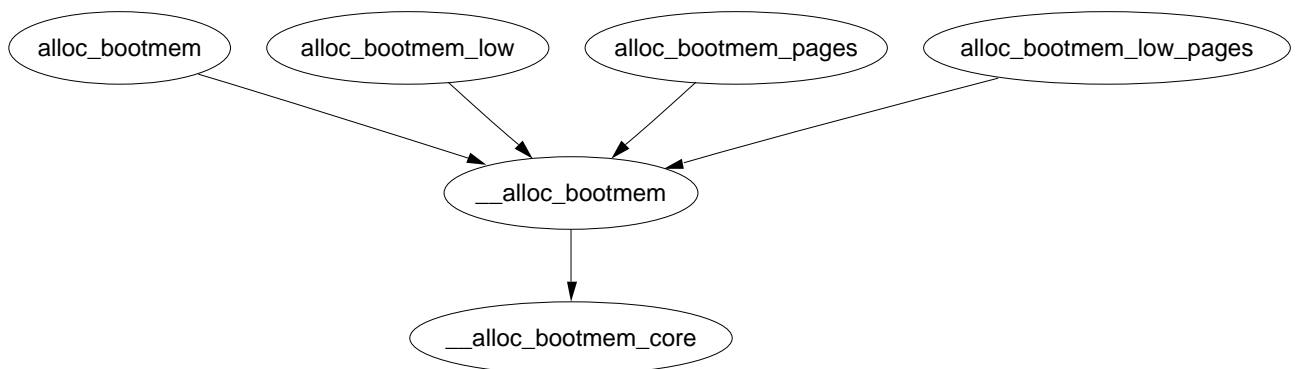


Figure 5.2: Call Graph: `__alloc_bootmem`

The `reserve_bootmem()` function may be used to reserve pages for use by the caller but is very cumbersome to use to general allocations. There is four functions provided for easy allocations on UMA architectures called `alloc_bootmem()`,

¹Hence the division by 8

`alloc_bootmem_low()`, `alloc_bootmem_pages()` and `alloc_bootmem_low_pages()` which are fully described in Table 5.1. All of these macros call `__alloc_bootmem()`, as shown in the call graph in Figure 5.2, with different parameters.

Similar ones exist for NUMA which take the node as an additional parameter as listed in Table 5.2. They are called `alloc_bootmem_node()`, `alloc_bootmem_pages_node()` and `alloc_bootmem_low_pages_node()`. All of these macros call `__alloc_bootmem_node()` with different parameters.

The parameters passed to either `__alloc_bootmem()` or `__alloc_bootmem_node()` are essentially the same. They are

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

size is the size of the requested allocation

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

goal is the preferred starting address to begin allocating from. The “low” functions will start from physical address 0 where 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 the allocations may be merged or not, the `pos` and `offset` fields will be updated to show the last page used for allocating and how much of the last page was used. If the last page was fully used, the offset is 0.

5.4 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 a major a problem as it sounds as the allocations always persist for the lifetime of the system but is still an important restriction for developers during boot time.

5.5 Retiring the Boot Memory Allocator

Late in the system has finished bootstrapping in the function `start_kernel()`, 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 principle 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 NUMA architectures, the equivalent function `free_all_bootmem_node()` is provided. The 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
- Free all pages used for the bitmap and free to 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 page. 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`.

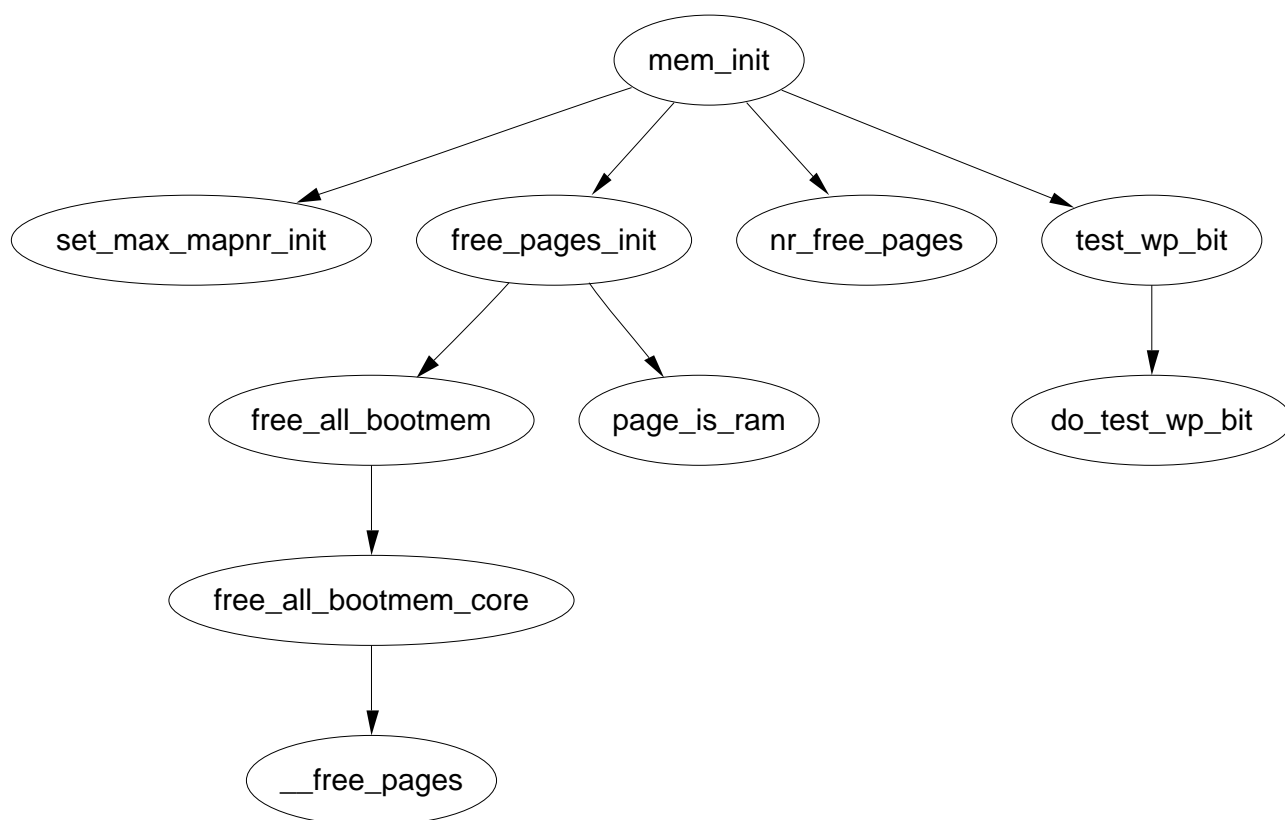


Figure 5.3: Call Graph: mem_init

This function simply 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 bootmem allocator is well and truly retired 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 memory² that is used by bootstrapping code that is no longer required.

²27 pages 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 5.1: 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 5.2: Boot Memory Allocator API for NUMA Architectures

Chapter 6

Physical Page Allocation

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

6.1 Allocator API

Linux provides a quite sizable API for the allocation of page frames. All of them take a *gfp_mask* which determines how the allocator will behave which is discussed in Section 6.5.

As the allocation API functions eventually map onto one function the API's are there so the correct node and zone will be chosen for the allocation. Different users will need different zones such as DMA for certain device drivers or NORMAL for disk buffers. A full list of page allocation API's are listed in Table 6.1.

There is a similar API for the freeing of pages. They are a lot simpler and exist to help with 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 6.2.

6.2 Managing Free Blocks

Pages are divided up into different sized blocks each of which is a power of two number of pages large. An array of **free_area_t** structs is maintained for each order that points to a linked list of blocks of pages that are free as indicated by Figure 6.1. Hence, the 0th element of the array will point to a list of free page blocks of size 2^0 or 1 page, the 1st element will be a list of 2^1 (2) pages up to $2^{MAX_ORDER-1}$ number of pages, the **MAX_ORDER** been 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**.

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

Table 6.1: Physical Pages Allocation API

Each zone has a **free_area_t** struct array called **free_area[`MAX_ORDER`]**. It is declared in *include/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;

```

`free_list` A linked list of free page blocks

`map` A bitmap representing the state of a pair of buddies

Linux saves space by only using one bit to represent each pair of buddies. Each time a buddy is alloc-ed 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. It 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.


```

__free_pages(struct page *page, unsigned int order)
    Free an order number of pages from the given page

__free_page(struct page *page)
    Free a single page

free_page(void *addr)
    Free a page from the given virtual address

```

Table 6.2: Physical Pages Free API

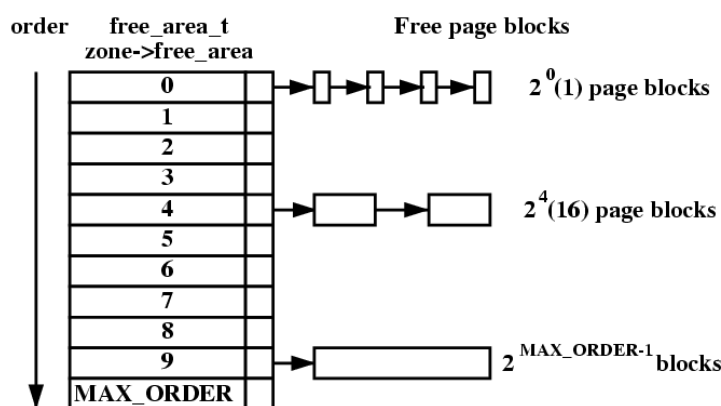


Figure 6.1: Free page block management

6.3 Allocating Pages

For allocation, the buddy system works by rounding requests for a number of pages up to the nearest power of two number of pages which is referred to as the *order* of the allocation. If a free block can not 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 6.3 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 is for which node to use. Linux uses a *node-local* allocation policy which states the memory bank associated with the running CPU is used for allocating pages. Here, the function `_alloc_pages()` is what is important. This function is different depending on whether a UMA (function in `mm/page_alloc.c`) or NUMA (function in `mm/numa.c`) architecture is in use.

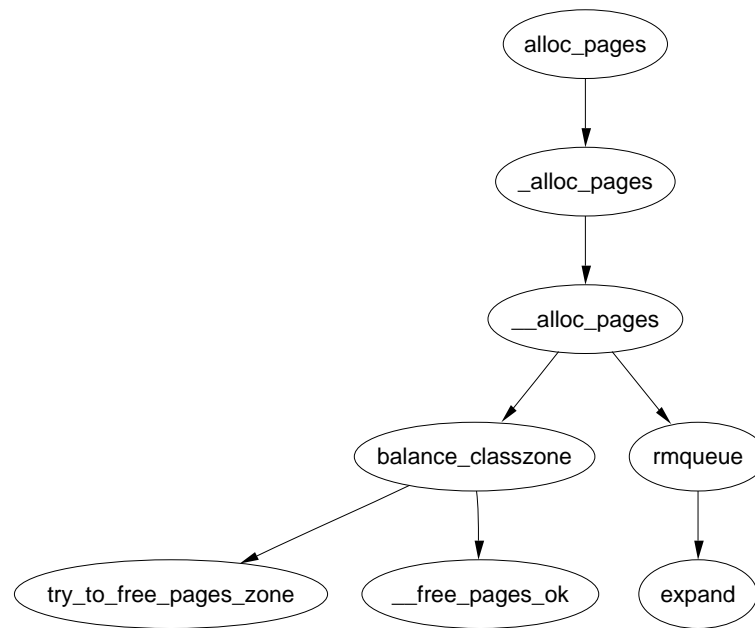


Figure 6.2: Call Graph: alloc_pages

No matter which API is used, they all will use `__alloc_pages()` in `mm/page_alloc.c` for all the real work and it is never called directly, see Figure 6.2 for the call graph. This function selects which zone to allocate from. It starts with the requested zone but will fall back to other zones if absolutely necessary. What zones to fall back on are decided at boot time by the function `build_zonelists()` but generally HIGH-MEM will fall back to NORMAL and that in turn will fall back to 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.

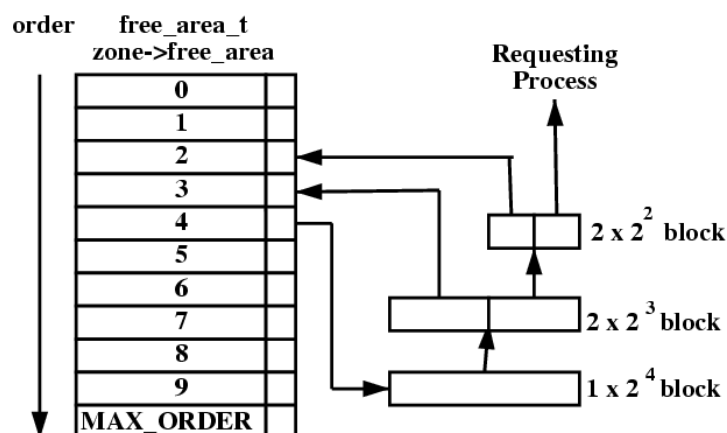


Figure 6.3: Allocating physical pages

The function `rmqueue()` is what allocates the block of pages or splits higher level

blocks if one of the appropriate size is not available.

6.4 Free Pages

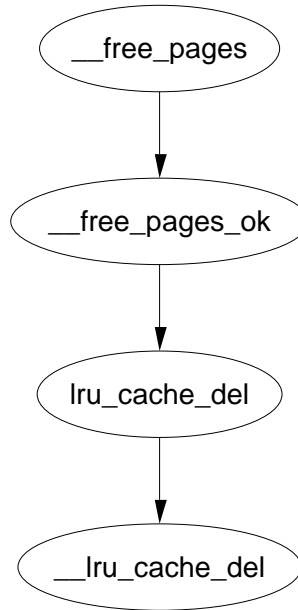


Figure 6.4: Call Graph: `__free_pages`

The principle function for freeing pages is `__free_pages_ok()`. It should not be called directly, instead the function `__free_pages()` should be used which does some simple checks first. See Figure 6.4 for its call graph.

When a buddy is freed, Linux tries to coalesce the buddies together immediately if possible. This is not the best method as the worst case scenario will have many coalescing followed by the immediate splitting of the same blocks[Vah96]. At time of writing, work is taking place on implementing a lazy buddy coalescing scheme[BL89].

To detect if the buddies can be merged or not, Linux checks the bit from the `free_area→map` to determine the state of the buddy. As the buddy has just been freed, it is that at least one is *definitely* free because a buddy has just been freed. If after toggling the bit it is 0, then the buddies can be merged.

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

$$mask = (\sim 0 << k)$$

The mask we are interested in is

$$imask = 1 + \sim mask$$

Linux takes a shortcut in calculating this by noting that

$$imask = -mask = 1 + \sim mask$$

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

6.5 GFP Flags

A persistent concept through the whole VM is the **GFP (Get Free Page)** flags. They determine how the allocator and kswapd may 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 `include/linux/mm.h`.

The first is zone modifiers listed in Table 6.3. 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`.

Flag	Description
<code>__GFP_DMA</code>	Allocate from <code>ZONE_DMA</code> if possible
<code>__GFP_HIGHMEM</code>	Allocate from <code>ZONE_HIGHMEM</code> if possible
<code>GFP_DMA</code>	Alias for <code>__GFP_DMA</code>

Table 6.3: Low Level GFP Flags Affecting Zone Allocation

The next flags are action modifiers listed in Table 6.4. They change the behavior of the VM and what the calling process may do. The low level flags on their own are too primitive to be easily used. It is difficult to know what the correct combinations are for each instance so a few high level combinations are defined and listed in Table 6.5. For clarity the `__GFP_` is removed from the table combinations so, the `__GFP_HIGH` flag will read as `HIGH` below. The combinations to form the high level flags are listed in Table 6.6

To help understand this, take `GFP_ATOMIC` as an example. It has only the `__GFP_HIGH` flag set. This means it is high priority, will use emergency pools (if they exist) but will not sleep, perform IO or access the filesystem. This flag would be used by an interrupt handler for example.

6.5.1 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 `include/linux/sched.h` but only the ones affecting VM behavior are listed in Table 6.7.

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 6.4: Low Level GFP Flags Affecting Allocator Behavior

6.6 Avoiding Fragmentation

One important problem that must be addressed with any allocator is the problem of internal and external fragmentation. External fragmentation is the inability to service a request because the available memory exists only in small blocks. Internal fragmentation is defined as the wasted space where a large block had to be assigned to service a small request. In Linux, external fragmentation is not a serious problem as large requests for contiguous pages are rare and usually **vmalloc** (See Chapter 7) is sufficient to service the request. The lists of free blocks ensure that large blocks do not have to be split unnecessarily.

Internal fragmentation is the single most serious failing of the binary buddy system. 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]. 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	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 6.5: Low Level GFP Flag Combinations For High Level

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 <code>buffer_init()</code> and <code>inode_init()</code>
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. In fact, it appears the flag was introduced specifically to avoid a deadlock in the loopback device.
GFP_NOHIGHIO	This is only used in one place in <code>alloc_bounce_page()</code> during the creating of a bounce buffer for 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 <code>GFP_USER</code> is that this could use emergency pools of pages but that is a no-op on 2.4.x kernels
GFP_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 <code>GFP_USER</code> (low) would fail where as the others would keep trying. Now it has no significance and is not treated any different to <code>GFP_KERNEL</code>
GFP_HIGHUSER	This flag indicates that the allocator should allocate from <code>ZONE_HIGHMEM</code> if possible. It is used when the page is allocated on behalf of a user process
GFP_KSWAPD	More historical significance. In reality this is not treated any different to <code>GFP_KERNEL</code>

Table 6.6: 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 OOM killer. 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. This functions the same as the PF_MEMALLOC flag in 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 <code>try_to_free_pages()</code> itself to indicate that free pages should be reserved for the calling process in <code>__free_pages_ok()</code> instead of returning to the free lists

Table 6.7: Process Flags Affecting Allocator Behavior

Chapter 7

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 virtually memory.

The region to be allocated must be a multiple of the hardware page size and requires altering the kernel page tables and there is a limitation on how much memory can be mapped with **vmalloc()** because only the upper region of memory after **PAGE_OFFSET** is available for the kernel (1GiB on an x86). As a result, it is used sparingly in the core kernel. In 2.4.20, it is only used for storing swap map information and for loading kernel modules into memory.

7.1 Kernel Address Space

The linear virtual address space that is important to the kernel is shown in Figure 7.1. The area up until **PAGE_OFFSET** is reserved for the process and changes with every context switch. In x86, this is defined as 0xC0000000 or 3GiB leaving the upper 1GiB of memory for the kernel.

After the process address space, kernel image is mapped followed by the physical page **mem_map** is stored which is the struct page for each physical page frame in the system. Between the physical memory map and the **vmalloc** address space, there is a gap of space **VMALLOC_OFFSET** in size. On the x86, this gap is 8MiB big and exists to guard against out of bounds errors.

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 in a contiguous virtual address space. In high memory systems, the area extends as far as **PKMAP_BASE** minus the 2 page gap. In that case, the remaining area is used for mapping high memory pages into the kernel virtual address with **kmap** and **kunmap**.

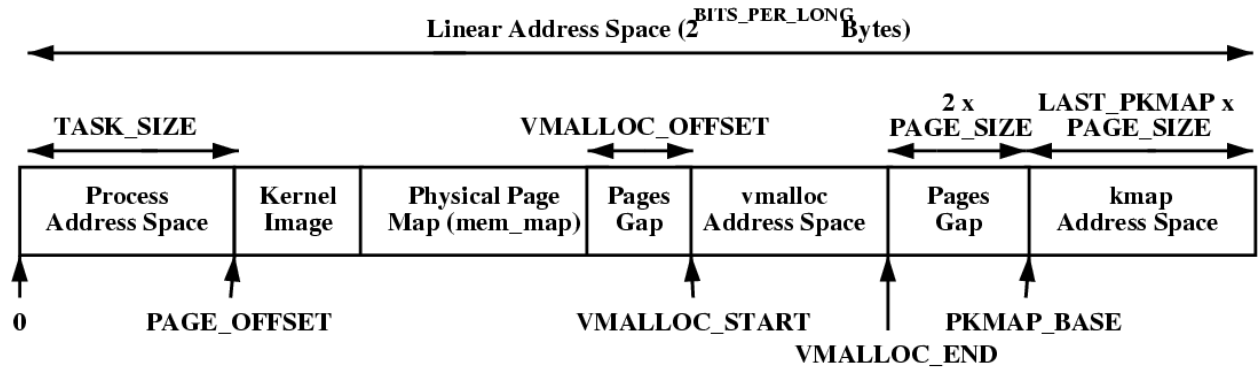


Figure 7.1: Kernel Address Space

7.2 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 `include/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 };

```

flags are set to either `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 is the start of the area

size is its size

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 area ordered by address for easy searches. Each area is separated by at least one page to protect against overruns. This is illustrated by the gaps in 7.2

When the kernel wishes to allocate a new area, the `vm_struct` list is searched literally by the function `get_vm_area()`. Space for the struct is allocated with `kmalloc()`. When the virtual area is used for ioremapping, this function will be called directly to map the requested area.

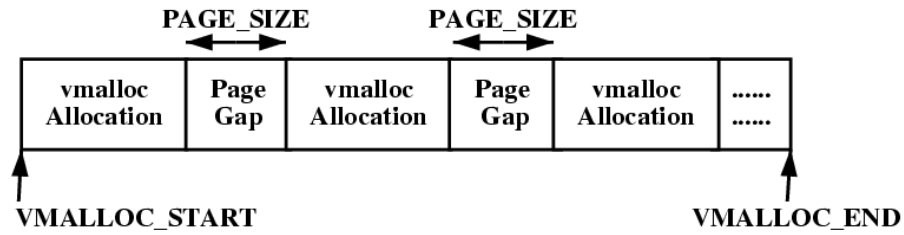


Figure 7.2: VMalloc Address Space

7.3 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 nearest page size. They all return a linear address for the new allocated area.

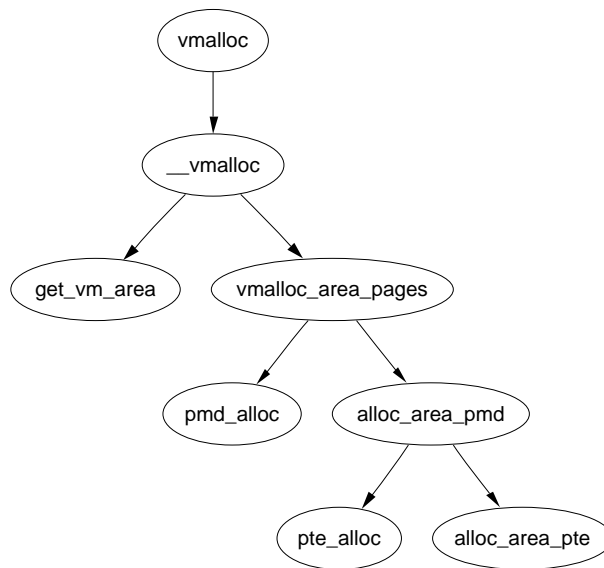


Figure 7.3: Call Graph: vmalloc

As is clear from the call graph shown in Figure 7.3, there is two steps to allocating the area.

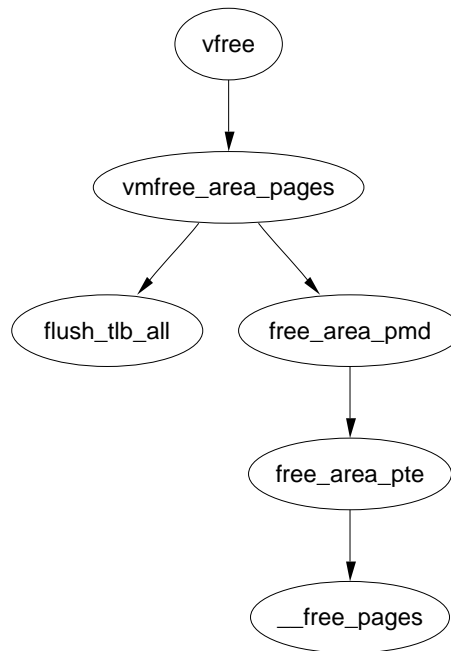
The first step with `get_vm_area()` finds a region large enough to store the request. It searches through a linear linked list of `vm_structs` and returns a new struct describing the allocated region.

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()`. Once allocated there is a special case in the page fault handling code which will allocate the necessary pages as necessary.

<code>vmalloc(unsigned long size)</code>
Allocate a number of pages in vmalloc space that satisfy the requested size
<code>vmalloc_dma(unsigned long size)</code>
Allocate a number of pages from ZONE_DMA
<code>vmalloc_32(unsigned long size)</code>
Allocate memory that is suitable for 32 bit addressing. This ensures it is in ZONE_NORMAL at least which some PCI devices require

Table 7.1: Non-Contiguous Memory Allocation API

7.4 Freeing A Non-Contiguous Area

Figure 7.4: Call Graph: `vfree`

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

The function `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.

<pre>vfree(void *addr)</pre>

<p>Free a region of memory allocated with <code>vmalloc</code>, <code>vmalloc_dma</code> or <code>vmalloc_32</code></p>

Table 7.2: Non-Contiguous Memory Free API

Chapter 8

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[JM01] and 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 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 block of contiguous pages in memory called **slabs** which are carved up into small chunks for the data structures and objects the cache manages. The structure of the allocator as described so far is illustrated in Figure 8.1.

The slab allocator has three principle aims;

- The allocation of small blocks of memory to help eliminate internal fragmentation that would be otherwise caused by the buddy system
- The caching of commonly used objects so that the system does not waste time allocating, 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.

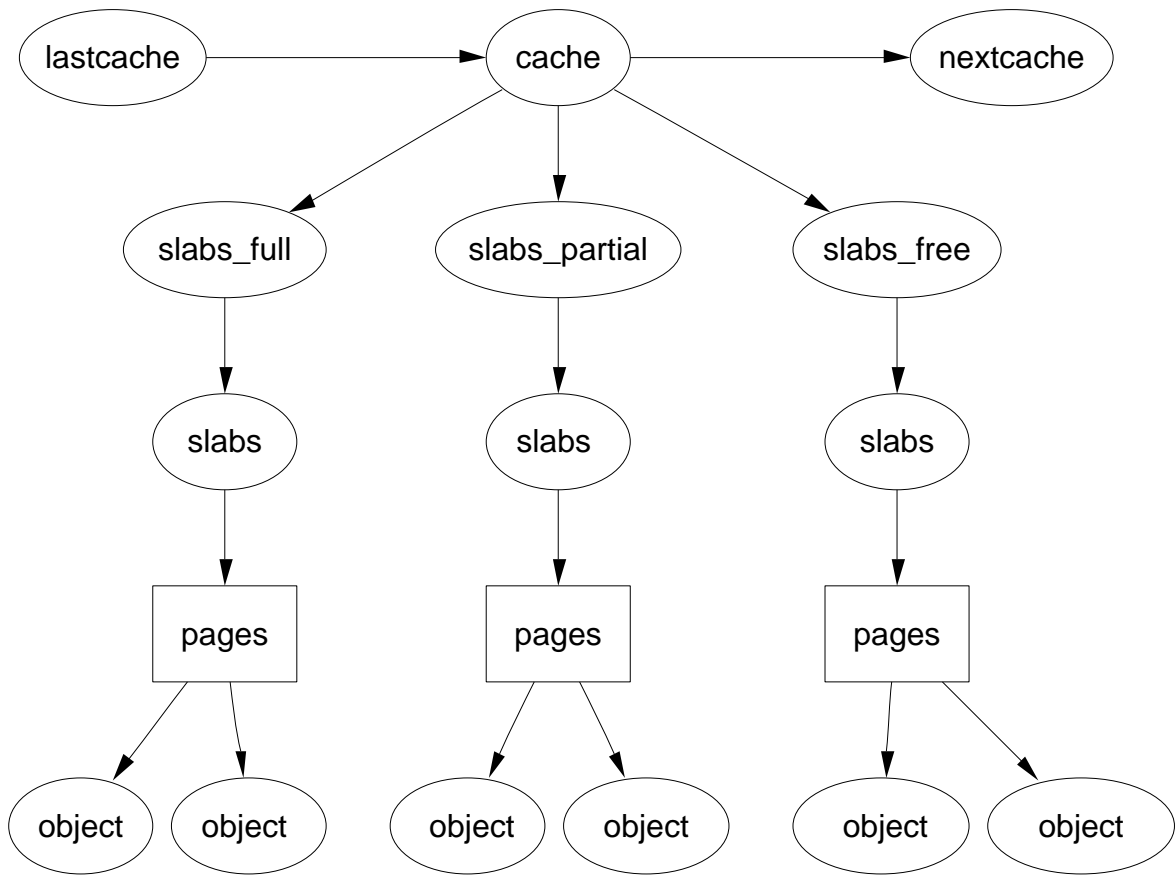


Figure 8.1: Layout of the Slab Allocator

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-X and size-X(DMA) where X is the size of the allocation and a function `kmallocc()` (See Section 8.4.1) 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 8.4.

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 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 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 different lines in the CPU cache helping ensure that objects from the same cache will not

constantly 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 caches[GAV95] or code[HK97] but the scheme used does help improve cache line usage. Cache colouring is further discussed in section 8.1.5. On an SMP system, a further step is taken to help cache utilization where each cache has a small array of objects for each CPU which is discussed further in Section 8.5.

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, 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 was buffer overflowed and reports it. Poisoning an object will fill it with a known pattern 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.

8.1 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. A 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
```

As is obvious, the fields do not have a header to indicate what each column means. Each of them correspond to a field in the `struct kmem_cache_s` structure. The fields listed here are

cache-name A human readable name such as “vm_area_struct”

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

total-objs How many 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 fields will be displayed after a colon. This refers to the per CPU cache described in the last section. The fields are

limit 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 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.

8.1.1 Cache Descriptor

All information describing a cache is stored in a **struct kmem_cache_s** declared in *mm/slab.c*. This is an extremely large struct 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_* are the three lists the slabs are kept on described above

objsize is the size of each object

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

num The number of objects contained in each slab

```

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 is the size of the slab in pages. The slab is $2^{gfporder}$ pages big as required by the buddy allocator

gfpflags is the GFP flags to use when allocating memory. See Section 6.5

colour is the number of different cache lines that can be used. This will be further discussed in Section 8.1.5

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

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

growing 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 are dynamic flags. See Section 8.1.3

ctor is for a complex object which provides a constructor function to be called on each new object. This is a pointer to that function. May be NULL

dtor is the object destructor. May be NULL

failures does not appear to be used anywhere

```

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

```

These are set during cache creation

name is the human readable name of the cache

next is the next cache on the cache chain

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

cpudata is the per-cpu data. Discussed further in Section 8.5

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

These figures are only available if the **CONFIG_SLAB_DEBUG** option is set during compile time. They are all beancounters and not of general interest.

num_active is the number of active objects in the cache

num_allocations is the total number of objects that have been allocated on this cache

high_mark is the highest value **num_active** has been to date

grown is the number of times **kmem_cache_grow()** has been called

reaped refers to the number of times this cache has been reaped

errors is never used

allochit is the number of times an allocation has used the per-cpu cache

allocmiss is the number of times an allocation has missed the per-cpu cache

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

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

8.1.2 Cache Static Flags

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

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

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 8.3.

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.

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.

8.1.3 Cache Dynamic Flags

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

8.1.4 Cache Allocation Flags

The 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 6.5 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, it doesn't.

8.1.5 Cache Colouring

To utilize 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 8.2.7) and what the bytes wasted is. Based on that, two figures are calculated

for the cache descriptor

colour is the number of different offsets that can be used

colour_off is the amount to offset each objects by

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 easiest explained with 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 2.

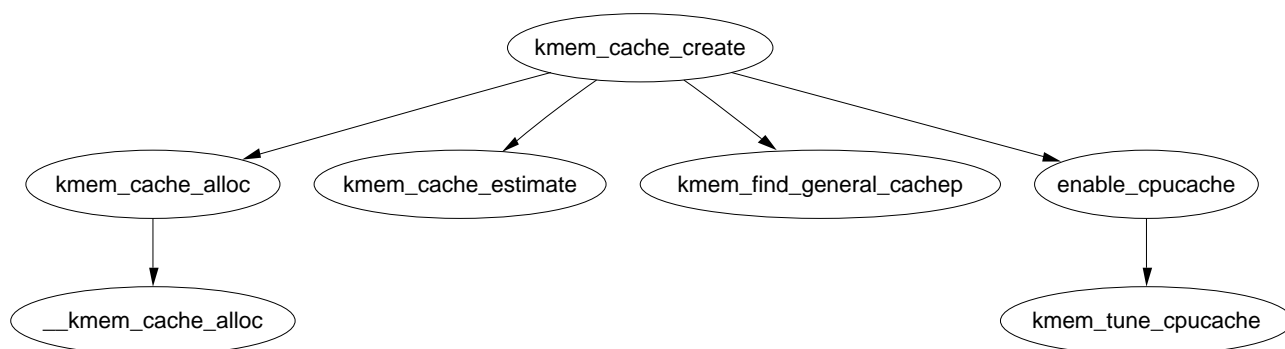
In this scenario, the first slab created will have its objects start at 0. The second will start at 32, the third at 64, the fourth at 96 and the fifth will start back at 0. With this, objects from each of the slabs will not hit the same hardware cache line on the CPU. The value of **colour** is 3 and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

8.1.6 Cache Creation

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

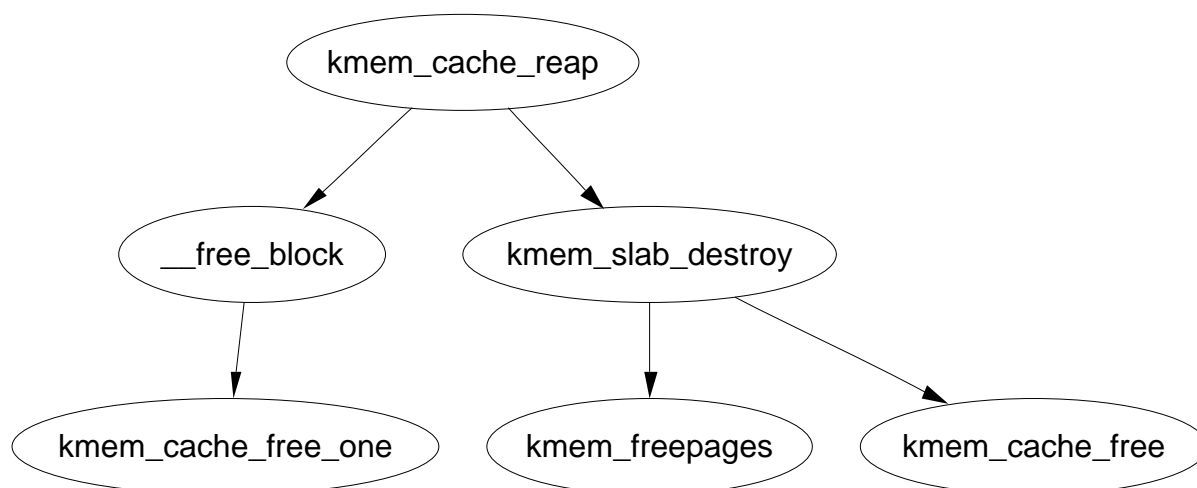
- Perform basic sanity checks for bad usage
- Perform debugging checks if **CONFIG_SLAB_DEBUG** is set
- Allocate a **kmem_cache_t** from the **cache_cache** slab cache
- Align the object size to the word size
- Calculate how many objects will fit on a slab
- Align the slab size to the hardware cache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

Figure 8.2 shows the call graph relevant to the creation of a cache and is more fully described in the code commentary.

Figure 8.2: Call Graph: `kmem_cache_create`

8.1.7 Cache Reaping

When a slab becomes free, 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 is that cache reaping does account what node the memory 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. As the vast majority of the cache memory will be using `ZONE_NORMAL`, the zone consideration is a serious problem.

Figure 8.3: Call Graph: `kmem_cache_reap`

The call graph in Figure 8.3 is deceptively simple as the task of selecting the proper cache to reap is quite long. In case there is many 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 over and over again. For each scanned cache, the reaper does the following

- 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**

8.1.8 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.

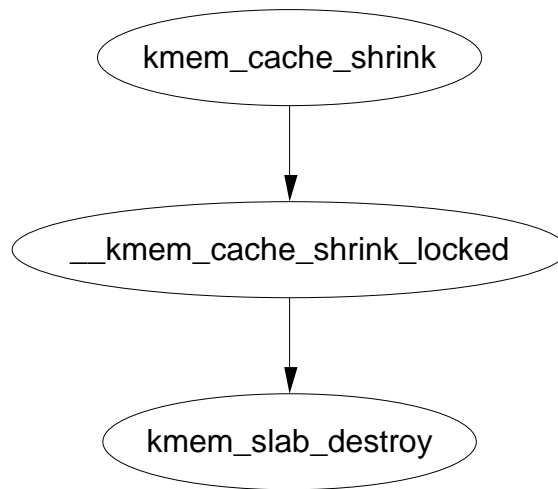
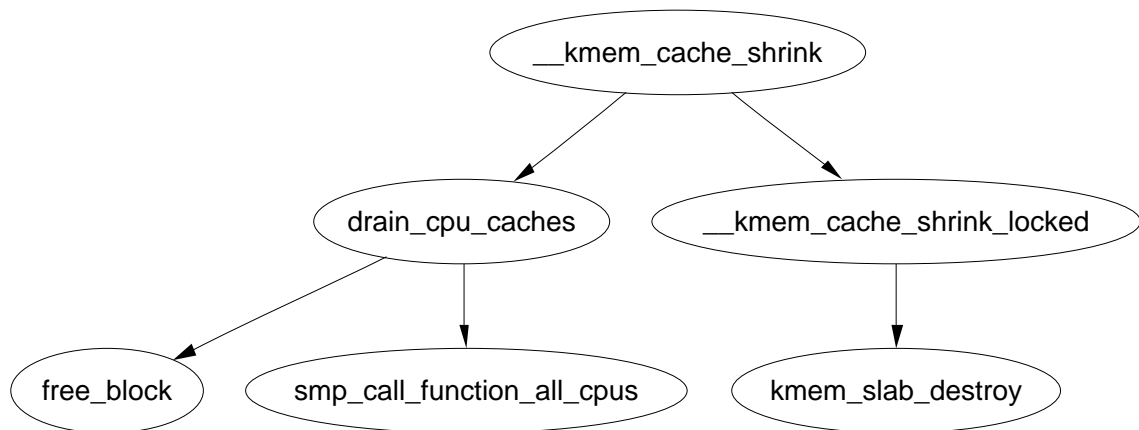
Two varieties of shrink functions exist with confusingly similar names. **kmem_cache_shrink()** removes all slabs from **slabs_free** and returns the number of pages freed as a result. This is the principle function exported for use by the slab allocator users.

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.

8.1.9 Cache Destroying

When a module is unloaded, it is responsible for destroying any cache with the function **kmem_cache_destroy()**. It is important the cache is properly destroyed as two caches of the same 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

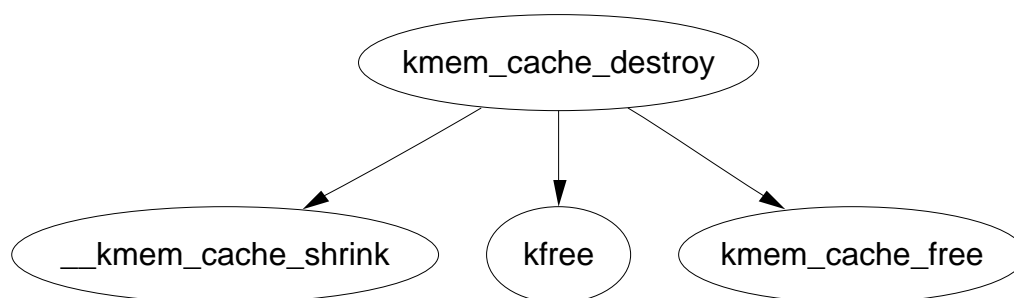
¹Defined statically as 10

Figure 8.4: Call Graph: `kmem_cache_shrink`Figure 8.5: Call Graph: `__kmem_cache_shrink`

- 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`

8.2 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. We begin with the descriptor.

Figure 8.6: Call Graph: `kmem_cache_destroy`

```

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

```

list is the list the slab belongs to. One of `slab_full`, `slab_partial` and `slab_free` from the cache manager

colouroff 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 is the starting address of the first object within the slab

inuse 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 8.2.3

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 `next` and `prev` on the page list to track what cache and slab an object belongs to. To get the descriptors from the page, the macros `GET_PAGE_CACHE()` and `GET_PAGE_SLAB()` are available. This set of relationships is illustrated in Figure 8.7

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.

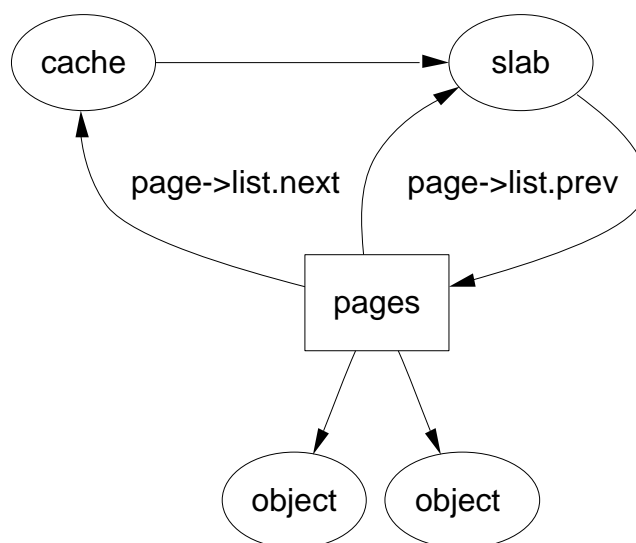


Figure 8.7: Page to Cache and Slab Relationship

8.2.1 Storing the Slab Descriptor

If the objects are larger than a threshold (512 bytes on x86), the **CFGS_OFF_SLAB** is set in the cache flags and the **slab descriptor** or manager is kept off-slab in one of the sizes cache (See Section 8.4) that is large enough to contain the struct is selected 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 there should not be many in the slab.

Alternatively, the slab manger is kept 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 8.8 should help clarify what a slab with the descriptor on-slab looks like and Figure 8.9 illustrates how a cache uses a sizes cache to store the slab descriptor when the descriptor is kept off-slab.

8.2.2 Slab Creation

At this point, we have seen how the cache is created, but on creation, it is an empty cache with empty lists for its `slab_full`, `slab_partial` and `slabs_free`. A cache is grown with the function `kmem_cache_grow()` when no objects are left in the `slabs_partial` list and there is no slabs in `slabs_free`. The tasks it fulfills are

- Perform basic sanity checks to guard against bad usage

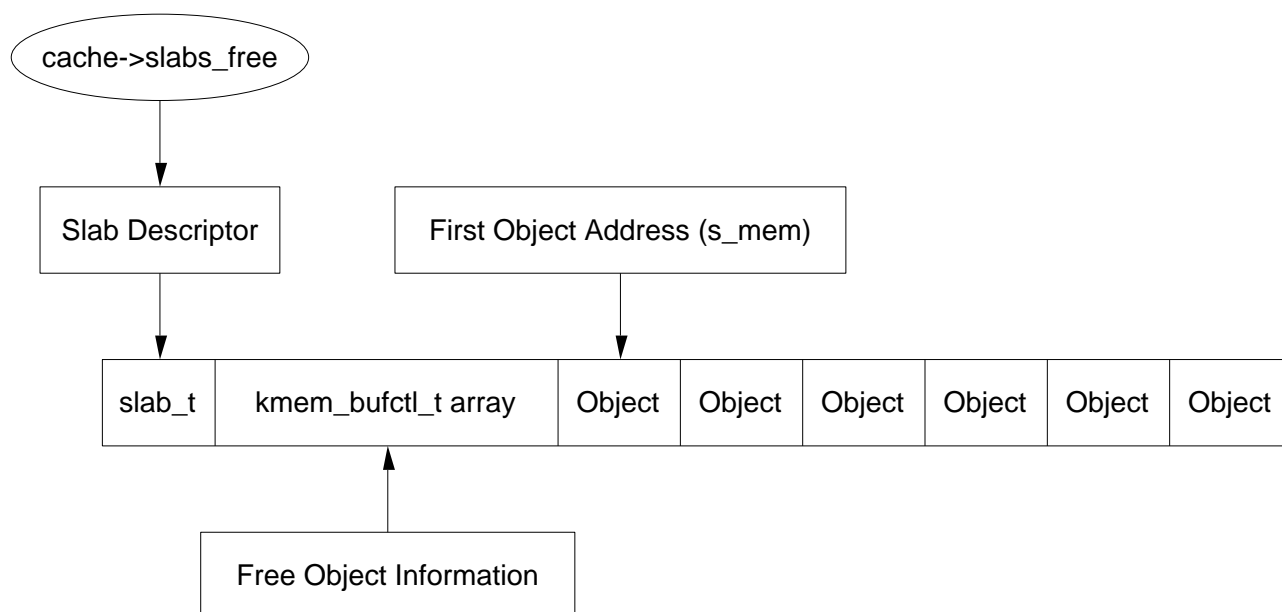


Figure 8.8: Slab With Descriptor On-Slab

- 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 (See Section 8.2)
- Initialise objects in the slab
- Add the slab to the cache

8.2.3 Tracking Free Objects

The slab allocator has to have a quick and simple way of tracking where free objects are on the partially filled slabs. It achieves this 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.

Historically, and according to the paper describing the slab allocator paper [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 the **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;
```

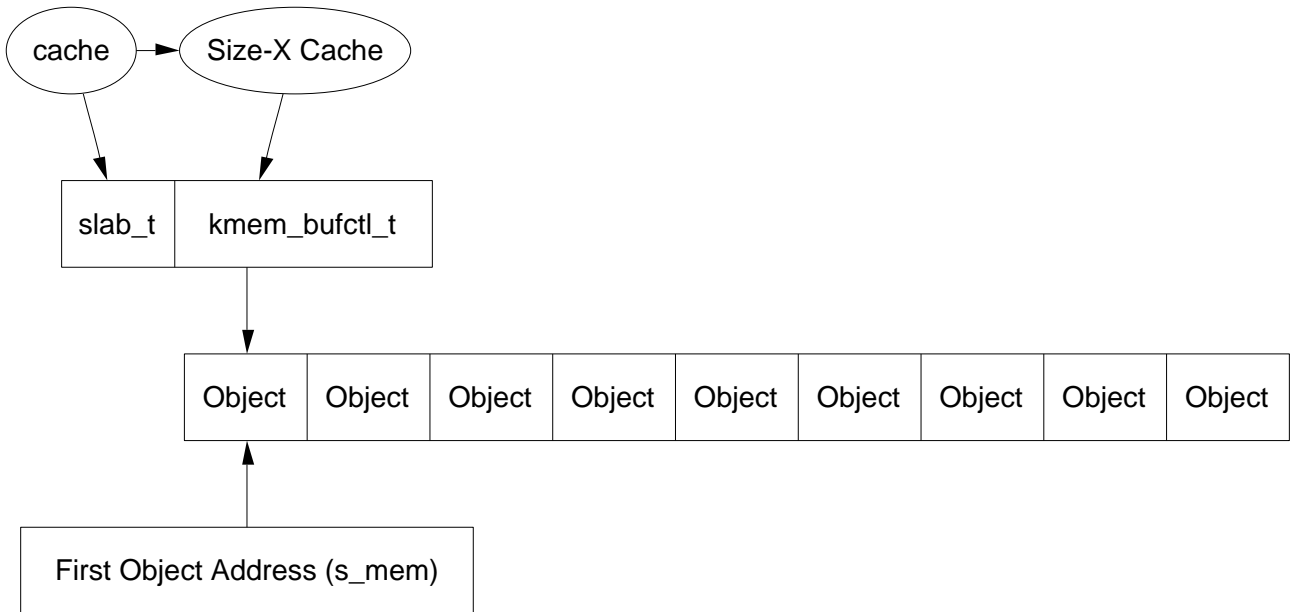


Figure 8.9: Slab With Descriptor Off-Slab

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))

```

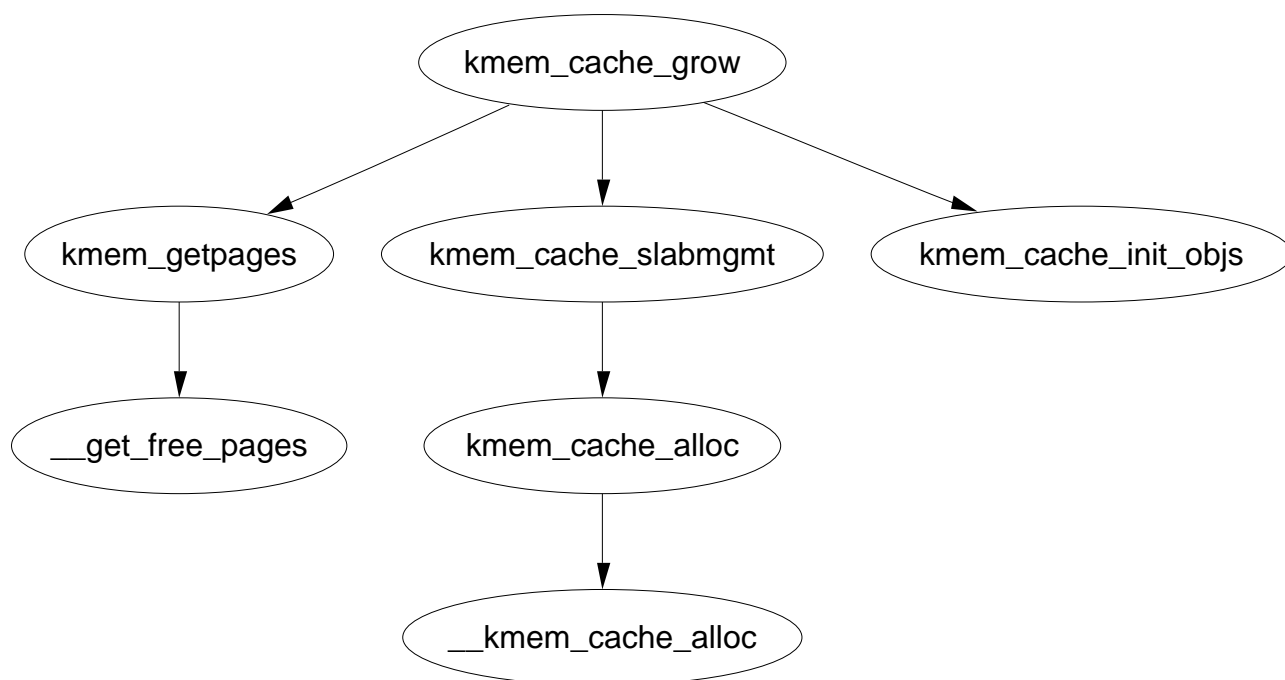
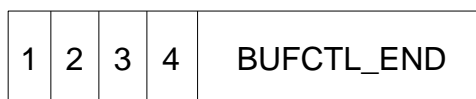
This seemingly cryptic macro is quite simple when broken down. The parameter `slabp` is to the slab manager. The block `((slab_t*)slabp)+1` casts `slabp` to a `slab_t` struct and adds 1 to it. This will give a `slab_t *` pointer to the beginning of the `kmem_bufctl_t` array. `(kmem_bufctl_t *)` recasts 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 give 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.

8.2.4 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 8.11

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

Figure 8.10: Call Graph: `kmem_cache_grow`Figure 8.11: initialised `kmem_bufctl_t` Array

be stored in `kmem_bufctl_t[n]`. Looking at the array above, the next object free after 0 is 1. After 1, there is two and so on. As the array is used, this arrangement will make the array act as a LIFO for free objects.

8.2.5 Finding the Next Free Object

`kmem_cache_alloc()` is the function which allocates an object by calling `kmem_cache_alloc_one()` which does the “real” work and updates 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 to allocate 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.

8.2.6 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 pointer 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 been freed so that it will be the next one allocated.

8.2.7 Calculating the Number of Objects on a Slab

During cache creation, the function `kmem_cache_estimate()` is called to estimate how many objects may be stored on a single slab taking into account whether the slab descriptor must be stored on 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, `PAGE_SIZE << gfp_order`
- Subtract the amount of space required to store the slab descriptor
- Count up the number of objects *i* 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

8.2.8 Slab Destroying

When a cache is been shrunk or destroyed, the slabs will be deleted. As the objects may have destructors, they 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 8.12 is very simple.

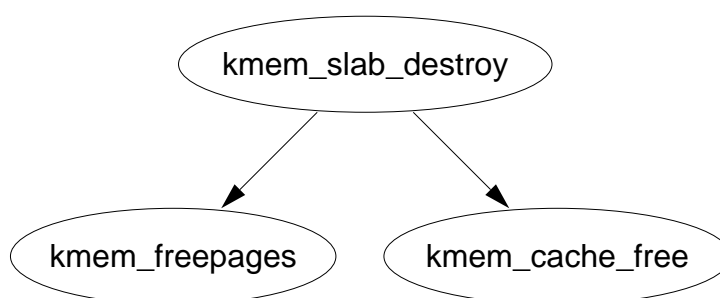


Figure 8.12: Call Graph: `kmem_slab_destroy`

8.3 Objects

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

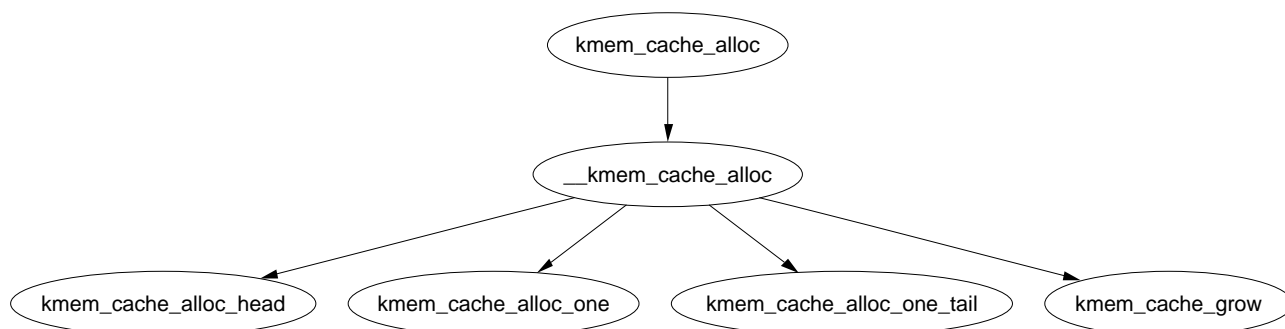
8.3.1 Initialising Objects in a Slab

When a slab is created, all the objects in it put in an initialised state. If a constructor is available, it is called for each object and it is expected when an object is freed, it is left in its initialised state. Conceptually this 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.

8.3.2 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 8.13 shows the basic call graph that is used to allocate an object in the SMP case.

There is 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 is one of `slabs_partial` or `slabs_free`. If there is no slabs in `slabs_free`, the cache is grown (See Section 8.2.2) to create a new slab in `slabs_free`. The final step is to allocate the object from the selected slab.

Figure 8.13: Call Graph: `kmem_cache_alloc`

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 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 8.5 for more information on the per-cpu caches.

8.3.3 Object Freeing

Freeing an object is a relatively simple task and is available via the `kmem_cache_free()` function. Just like `kmem_cache_alloc()`, it behaves differently in the UP and SMP cases. The principle 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.

8.4 Sizes Cache

Linux keeps two sets of caches for small memory allocations which the physical page allocator is unsuitable for. One cache is for use with DMA and the other suitable for normal use. The human readable names for these caches are **size-X cache** and **size-X(DMA) cache** viewable from `/proc/cpuinfo`. 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;
  
```

332 The size of the memory block

333 The cache of blocks for normal memory use

334 The cache of blocks for use with DMA

As there is 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.

8.4.1 **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 8.14 is therefore very simple as all the hard work is in cache allocation.

8.4.2 **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 8.3.3) so the call graph in Figure 8.15 is very simple.

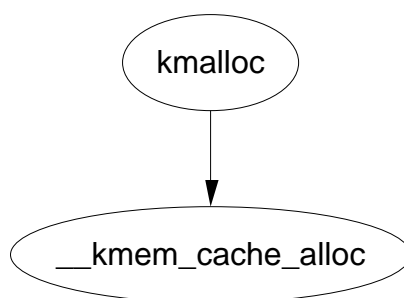


Figure 8.14: Call Graph: kcalloc

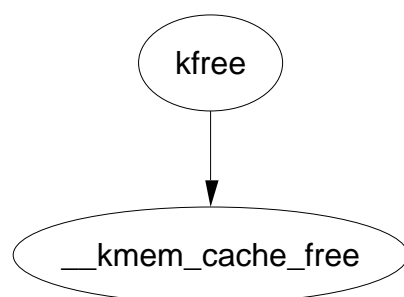


Figure 8.15: Call Graph: kfree

8.5 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, called a **cpucache** for each CPU in the system.

When allocating or freeing objects, they are placed in the cpucache. When there is 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 to 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 taken for every allocation and free which is unnecessarily expensive.

8.5.1 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;
```

avail is the number of free objects available on this cpucache

limit 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.

8.5.2 Adding/Removing Objects from the Per-CPU Cache

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

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

To remove an object

```
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.

8.5.3 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 `cpucache`'s been enabled before it is possible. 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.

8.5.4 Updating Per-CPU Information

When the per-cpu caches have been created or changed, each CPU has to be told about it. It's not sufficient to change all the values in the cache descriptor as that would lead to cache coherency issues and spinlocks would have to be used to protect the `cpucache`'s. 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;
```

The `cachep` is the cache been updated and the array `new` is 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.

8.5.5 Draining a Per-CPU Cache

When a cache is been shrunk, its first step is to drain the `cpucaches` of any objects they might have. This is so 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 is the least of its trouble.

8.6 Slab Allocator Initialisation

Here we will describe 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 };

```

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 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()`.

8.7 Interfacing with the Buddy Allocator

The slab allocator doesn't 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.

<code>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))</code> Creates a new cache and adds it to the cache chain
<code>kmem_cache_reap(int gfp_mask)</code> Scans at most REAP_SCANLEN caches and selects one for reaping all per-cpu objects and free slabs from. Called when memory is tight
<code>kmem_cache_shrink(kmem_cache_t *cachep)</code> 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.
<code>kmem_cache_alloc(kmem_cache_t *cachep, int flags)</code> Allocate a single object from the cache and return it to the caller
<code>kmem_cache_free(kmem_cache_t *cachep, void *objp)</code> Free an object and return it to the cache
<code>kmalloc(size_t size, int flags)</code> Allocate a block of memory from one of the sizes cache
<code>kfree(const void *objp)</code> Free a block of memory allocated with <code>kmalloc</code>
<code>kmem_cache_destroy(kmem_cache_t *cachep)</code> Destroys all objects in all slabs and frees up all associated memory before removing the cache from the chain

Table 8.1: Slab Allocator API for caches

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

Table 8.2: Internal cache static flags

Flag	Description
SLAB_HWCACHE_ALIGN	Align the objects to the L1 CPU cache
SLAB_NO_REAP	Never reap slabs in this cache
SLAB_CACHE_DMA	Use memory from ZONE_DMA

Table 8.3: Cache static flags set by caller

Flag	Description
SLAB_DEBUG_FREE SLAB_DEBUG_INITIAL	Perform expensive checks on free 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 known a pattern for trapping changes made to objects not allocated or initialised

Table 8.4: Cache static debug flags

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

Table 8.5: Cache Allocation Flags

Chapter 9

Process Address Space

The allocation methods discussed till now have dealt exclusively with kernel requests. They are considered high priority, rarely deferred¹ and never swapped out. It is presumed that the kernel is error free and has a good reason for needing the memory. More importantly, the kernel addressing space does not change so no matter what process is running, the virtual address space reserved for the kernel remains the same.

It is very different for processes. Each process has its own linear address space which potentially can change with every context switch. The only exception is when lazy TLB switch is in used which processes such as `init` and kernel threads use.

Allocations on behalf of a user process are considered low priority and are not satisfied immediately. Instead space is reserved in the linear address space and a physical page is only allocated upon access which is signaled by a *page fault*.

The process address is not trusted or presumed to be constant. The kernel is prepared to catch all exception and addressing errors raised from userspace. When the kernel is copying to or from userspace, the functions `copy_to_user()` and `copy_from_user()` are used to read memory rather than accessing the addresses directly. Linux relies on the MMU to raise exceptions when the address is invalid and have the Page Fault Exception handler catch 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.

9.1 Managing the Address Space

From a user perspective, the address space is a flat linear set of addresses which may be used but the kernel's perspective is slightly different. The linear address space is split into two parts, the userspace part which changes with each 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 3GiB on the x86.

¹`vmalloc` being the exception which is only allocated on page fault

This means that 3GiB is available for the process to use the the remaining 1GiB is always mapped by the kernel.

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

Each address space consists of a number of page aligned regions of memory that are in use. They never overlap and represent a set of addresses which contain pages that are related to each other in terms of protection and purpose. These regions are represented by a struct `vm_area_struct` and is 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 in the region may have been never allocation, are present and in use or swapped out to disk.

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.

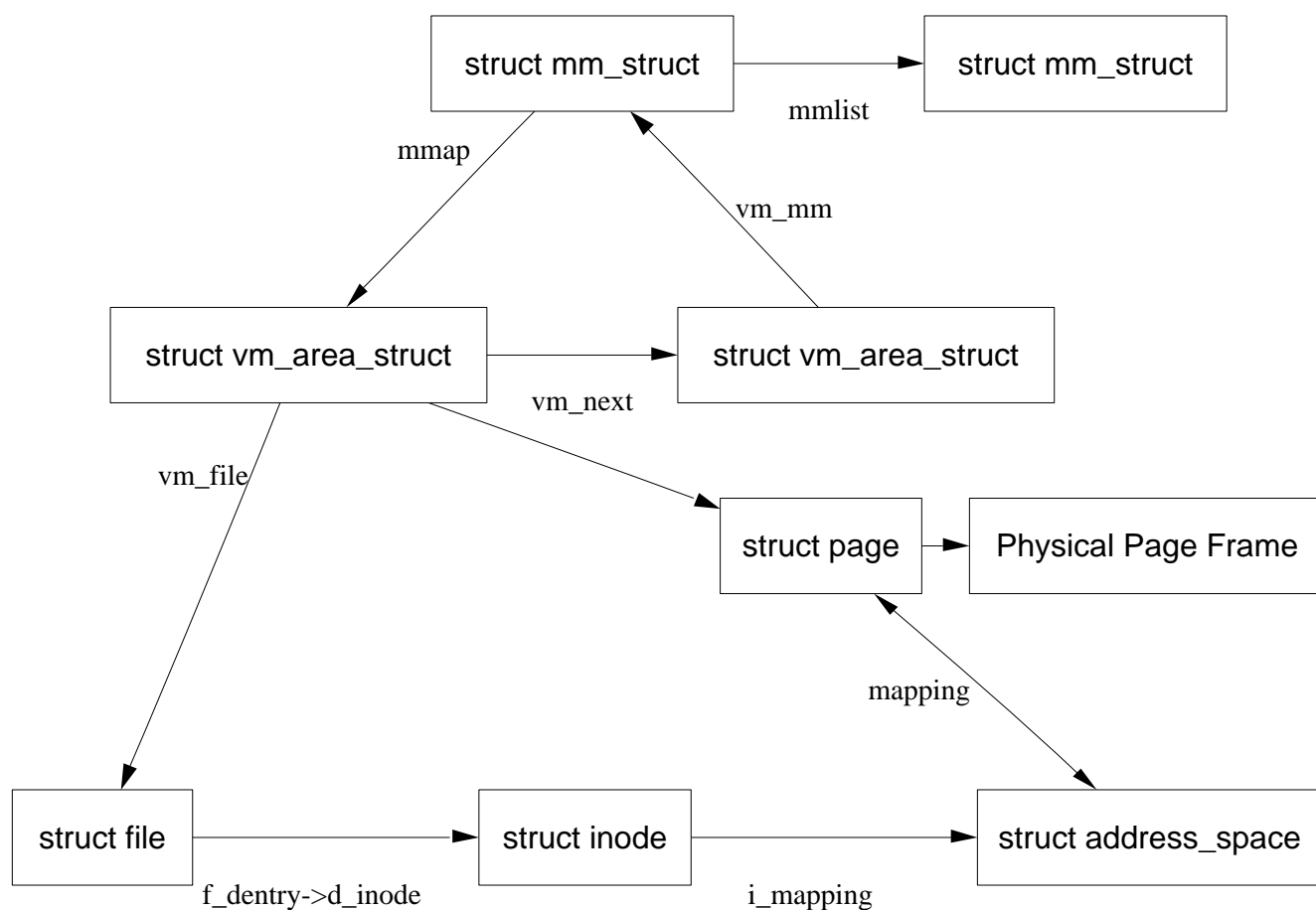


Figure 9.1: Data Structures related to the Address Space

A number of system calls are provided which affect the address space and regions

which are listed in Table 9.1

System Call	Description
<code>fork()</code>	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
<code>clone()</code>	<code>clone()</code> allows a new process to be created that shares parts of its context with its parent and is how threading is implemented in Linux. <code>clone()</code> without the <code>CLONE_VM</code> set will create a new address space which is essentially the same as <code>fork()</code>
<code>mmap()</code>	<code>mmap</code> creates a new region within the process linear address space
<code>mremap()</code>	Remaps or resizes a region of memory. If the virtual address space is not available for the mapping, the region may be moved unless the move is forbidden by the caller.
<code>munmap()</code>	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
<code>shmat()</code>	This attaches a shared memory segment to a process address space
<code>shmdt()</code>	Removes a shared memory segment from an address space
<code>execve()</code>	This loads a new executable file replacing the current address space
<code>exit()</code>	Destroys an address space and all regions

Table 9.1: System Calls Related to Memory Regions

9.2 Process Address Space Descriptor

The process address space is described by the `mm_struct`. Only one exists for each process and is shared between threads. Threads are identified in the task list by having two task list entries with the same `mm_struct` pointer.

Kernel threads have no user space context and so the `task_struct`→`mm` field is NULL. For some tasks such as the boot idle task, it is never setup but for kernel threads, a call to `daemonize()` calls `exit_mm()` to delete it. These tasks use what is called **Lazy TLB** during context switches during `schedule()`. Instead of carrying out an expensive TLB flush by calling `switch_mm()`, these processes borrow the mm of the previous task and place it in `task_struct`→`active_mm`.

A unique `mm_struct()` is not needed for kernel threads as they will never be page faulting or accessing the userspace portion. The only exception is faulting in vmalloc space which is treated as a special case of the page fault handling code. As

flushes are extremely expensive, especially with architectures such as the PPC, the use of Lazy TLB can show large improvements for context switches.

When entering Lazy TLB, the function `enter_lazy_tlb()` is called to ensure that a mm is not shared between processors in SMP machines although on UP machines, the function is a a NULL operation. The second time when lazy TLB is used 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”. The `mm_users` is a reference count of processes accessing the userspace portion of for this mm such as the page tables and file mappings. Threads and the `swap_out` code for instance will increment this count make 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 reference count of the “anonymous users” for the mm 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 of these are kernel threads which use lazy TLB switching and have no `mm_struct` of their own. When this count drops to 0, the `mm_struct` may be 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 their removal.

The `mm_struct` is defined in `include/linux/sched.h` as follows;

```

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
223     unsigned long start_code, end_code, start_data, end_data;
224     unsigned long start_brk, brk, start_stack;
225     unsigned long arg_start, arg_end, env_start, env_end;
226     unsigned long rss, total_vm, locked_vm;
227     unsigned long def_flags;
228     unsigned long cpu_vm_mask;
229     unsigned long swap_address;
230
231     unsigned dumpable:1;
232
233     /* Architecture-specific MM context */
234     mm_context_t context;
235 };
```

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 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 the taker could run for so long, a spinlock is inappropriate. A reader of the list takes this semaphore with `down_read()`. If they need to write, it must be taken with `down_write()` and the `page_table_lock` must be taken as well

page_table_lock This protects most fields on the `mm_struct`. As well as the page tables, It protects the rss count and the vma from modification

mmlist All mm'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, end_brk The start and end address of the heap

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), 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 amount of memory locked with `mlock()` by the process

def_flags It has 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 CPU's in an SMP system. The mask is used with IPI to determine if a processor should execute a particular function or not. This is important during TLB flush for each CPU for example

swap_address Used by the vmscan code to record the last address that was swapped from when swapping out entire processes

dumpable Set by `prctl()`, this flag is important only to `ptrace`

context Architecture specific MMU context

There is a small number of functions for dealing with `mm_structs` which is described in Table 9.2

9.2.1 Allocating a Descriptor

Two functions are provided to allocate a `mm_struct()`. To be slightly confusing, they are essentially the same but with important differences. **allocate_mm** will allocate a `mm_struct` from the slab allocator. **alloc_mm** will allocate from slab and then call the function `mm_init()` to initialise it.

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

Table 9.2: Functions related to memory region descriptors

9.2.2 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 copies of their parent `mm_struct` copied using `copy_mm()` with the process specific fields initialised with `init_mm()`.

9.2.3 Destroying a Descriptor

A new user to an `mm` increments the usage count with a simple call,

```
atomic_int(&mm->mm_users);
```

As long as the count is above 0, the caller is guaranteed that the `mm_struct` will not disappear prematurely. It is decremented with a call to `mmapput()`. If the count reaches zero, all the mapped regions with `exit_mmap()` and the `mm` destroyed with `mm_drop()`.

9.3 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_number/maps`.

The region is represented by a number of different structures illustrated in Figure 9.1. 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.

```

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

```

vm_mm The `mm_struct` this VMA belongs to

Function	Description
<code>find_vma()</code>	Finds the VMA that covers a given address. If the region does not exist, it returns the VMA closest to the requested address
<code>find_vma_prev()</code>	Same as <code>find_vma</code> except it also gives the VMA pointing to the returned VMA. It is not used, with <code>sys_mprotect()</code> being the notable exception, as it is usually <code>find_vma_prepare()</code> that is required
<code>find_vma_prepare()</code>	Same as <code>find_vma</code> 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
<code>find_vma_intersection()</code>	Returns the VMA which intersects a given address range. Useful when checking if a linear address region is in use by any VMA
<code>vma_merge()</code>	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 may it be expanded backwards to cover the address range instead. Regions may be merged if there is no file/device mapping and the permissions match
<code>get_unmapped_area()</code>	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
<code>insert_vm_struct()</code>	Inserts a new VMA into a linear address space

Table 9.3: Memory Region VMA API

vm_start The starting address

vm_end The end address

vm_next All the VMAs in an address space are linked together in an address ordered singly linked list with this field

vm_page_prot The protection flags for all pages in this VMA which are all defined in *include/linux/mm.h*. See Table 9.2 for a full description

vm_rb As well as been 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 to **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 mmap'ed

vm_file The struct file pointer to the file been mapped

vm_raend This is the end address of a readahead window. When a fault occurs, a readahead window will page in a number of pages after the fault address. This field records how far to read ahead

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

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.

9.3.1 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 earlier in Figure 9.1. The struct contains information of relevance to the filesystem such as the number of dirty pages which must be flushed to disk. It is defined as follows in *include/linux/fs.h*

Flags	Description
Protection Flags	
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. Appears 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_SHARED 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 9.2: Memory Region Flags

```

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 };

```

clean_pages A list of clean pages which do not have to be synchronized with the disk

dirty_pages Pages that the process has touched and need to be sync-ed

locked_pages The number 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 `include/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()
389      * call be followed
390      * by a commit_write() call - they must be balanced
391      */
392     int (*prepare_write)(struct file *, struct page *,
393                          unsigned, unsigned);
394     int (*commit_write)(struct file *, struct page *,
395                        unsigned, unsigned);
396     /* Unfortunately this kludge is needed for FIBMAP.
397      * Don't use it */
398     int (*bmap)(struct address_space *, long);
399     int (*flushpage) (struct page *, unsigned long);
400     int (*releasepage) (struct page *, int);
401 #define KERNEL_HAS_O_DIRECT
402     int (*direct_IO)(int, struct inode *, struct kiobuf *,
403                     unsigned long, int);
404 };

```

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 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()`

9.3.2 Creating A Memory Region

The system call `mmap` is provided for creating new memory regions within a process. For the x86, the function is called `sys_mmap2()` and is responsible for performing basic checks before calling `do_mmap_pgoff()` which is the prime function for creating new areas for all architectures.

The two high functions above `do_mmap_pgoff()` are essentially sanity checkers. They ensure the mapping size of page aligned if necessary, clears invalid flags, looks up the `struct file` for the given file descriptor and acquires the `mmap_sem` semaphore.

This `do_mmap_pgoff()` function is very large and broadly speaking it takes the following steps;

- Call the filesystem or device specific `mmap` function
- Sanity check the parameters
- Find a free linear address space large enough for the memory mapping
- 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 it's suitable for the new mapping
- Allocate a `vm_area_struct` from the slab allocator and fill in its entries
- Link in the new VMA
- Update statistics and exit

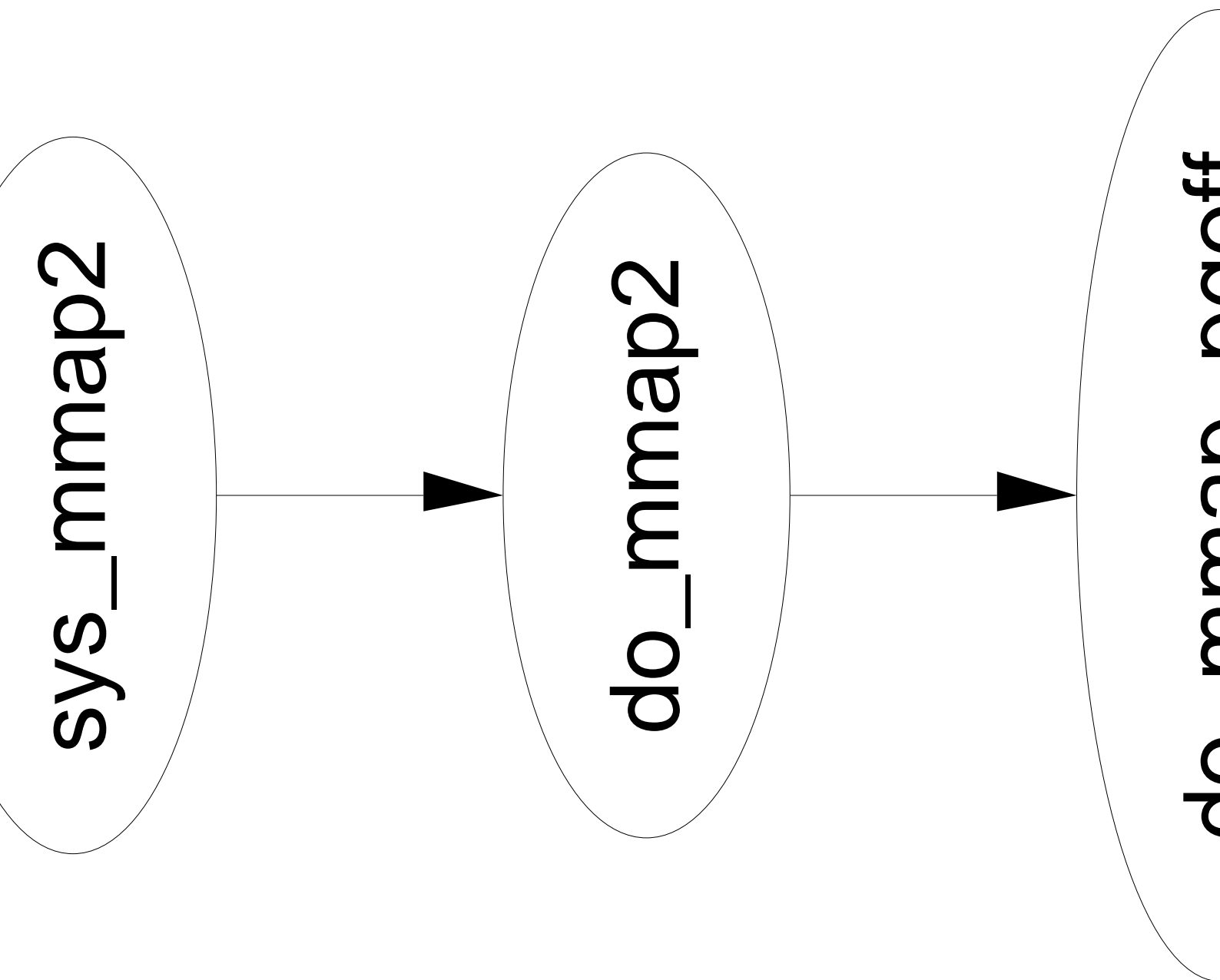
9.3.3 Finding a Mapped Memory Region

A common operation is to find the VMA a particular address belongs to during operations such as a page fault and the function responsible is `find_vma()`.

It first checks the `mmap_cache` field which caches the result of the last call to `find_vma()` as it is quite likely the same region is needed a few times in succession. If it not the desired region, the red-back tree stored in the `mm_rb` field is traversed. It returns the VMA closest to the requested address so it is important callers ensure the returned VMA contains the desired address.

A second function is provided which is functionally similar called `find_vma_prev()`. The only difference is that it also returns the pointer to the VMA preceding the searched for VMA² which is required as the list is a singly listed list. This is used rarely but most notably, it is used when deciding if two VMAs can be merged so that

²This is one of the very rare cases where a singly linked list is used in the kernel

Figure 9.3: Call Graph: `sys_mmap2`

the two VMAs may be easily compared. It is also used while removing a memory region so that the linked lists may be fixed up.

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.

9.3.4 Finding a Free Memory Region

When a new area is to be `mmap`'d, 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()`.

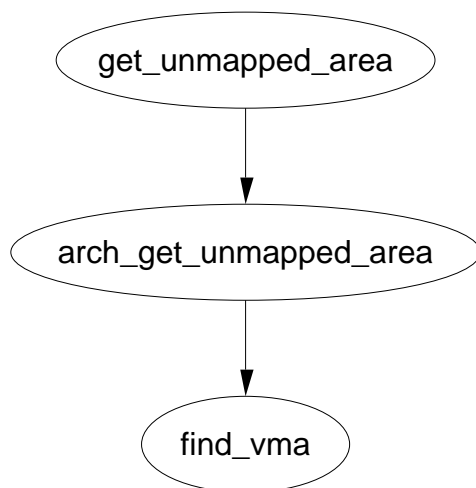


Figure 9.4: Call Graph: `get_unmapped_area`

As the call graph in Figure 9.4 shows, there is not much 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`, 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.

If a device is been mapped, such as a video card, the associated `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 is 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`.

9.3.5 Inserting a memory region

The principle function available for inserting a new memory region is `insert_vm_struct()` whose call graph can be seen in Figure 9.5. It is a very simply function which first called `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.

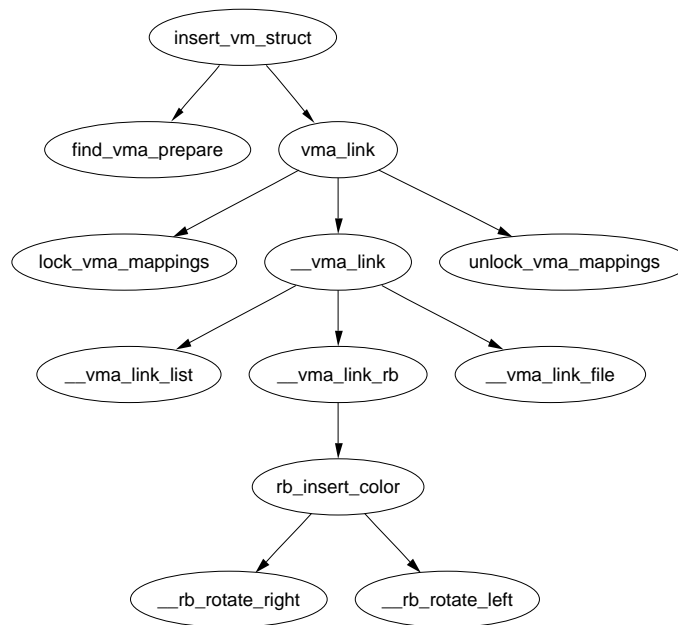


Figure 9.5: Call Graph: `insert_vm_struct`

The function `insert_vm_struct()` is rarely used as it does not increase the `map_count` field. Instead, the function more commonly used is `__insert_vm_struct()` which performs the same tasks except it increases `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'll 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 users 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, each of which has a single function. `__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`), then it will be made the red-black root as well. 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.

9.3.6 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()`.

Only one function exists now that is roughly equivalent `vma_merge()` and its use is quite rare. It is only called during `sys_mmap()` if it is an anonymous region been mapped and during `do_brk()`. The principle difference is that instead of merging two regions together, it will check to see can another region be expanded to cover the new allocation removing the need to create a new region. A region can be expanded if there is no file or device mappings and the permissions of the two areas are the same.

Regions are merged elsewhere albeit no function is explicitly called to perform the merging. The first is during a call to `sys_mprotect()`. During the fixup of areas, the two regions will be merged if the permissions are now the same. The second is during a call to `move_vma()` when it is likely similar regions will be located beside each other.

9.3.7 Remapping and moving a memory region

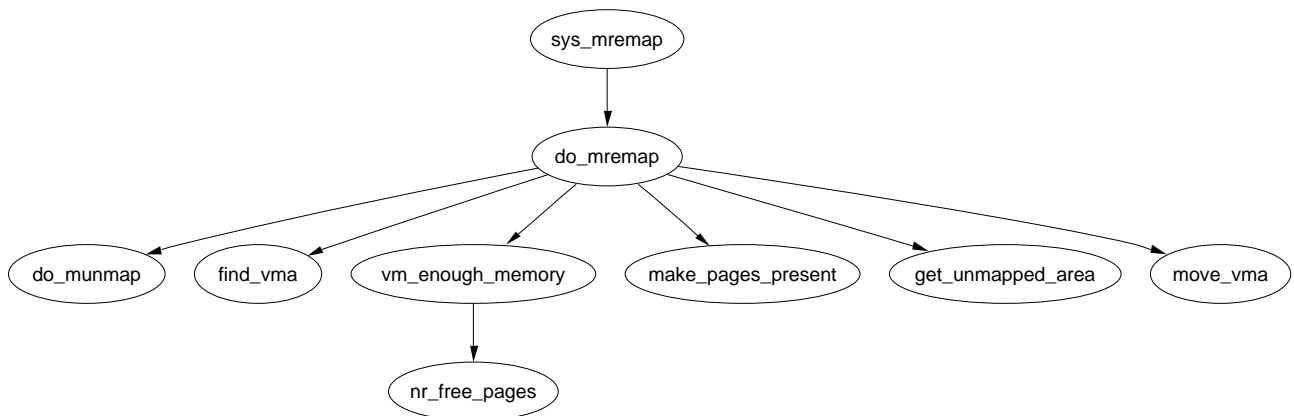
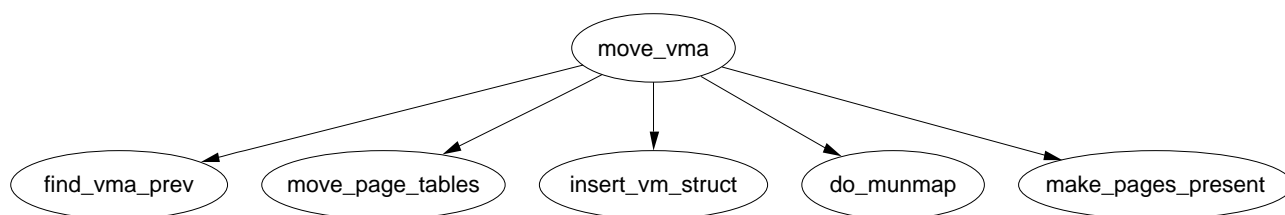


Figure 9.6: Call Graph: `sys_mremap`

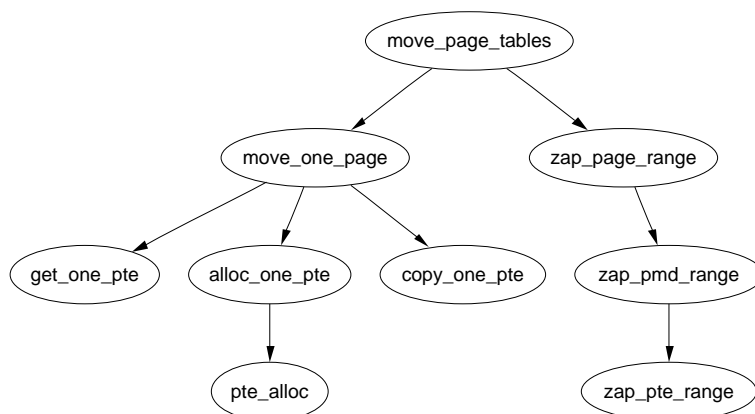
Memory regions may be moved during a call to `sys_mremap()` if the region is growing, would overlap another region and `MREMAP_FIXED` is not specified in the flags. The call graph is illustrated in Figure 9.6.

To move a region, it 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 9.7 for the call graph.

Figure 9.7: Call Graph: `move_vma`

First the function 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.

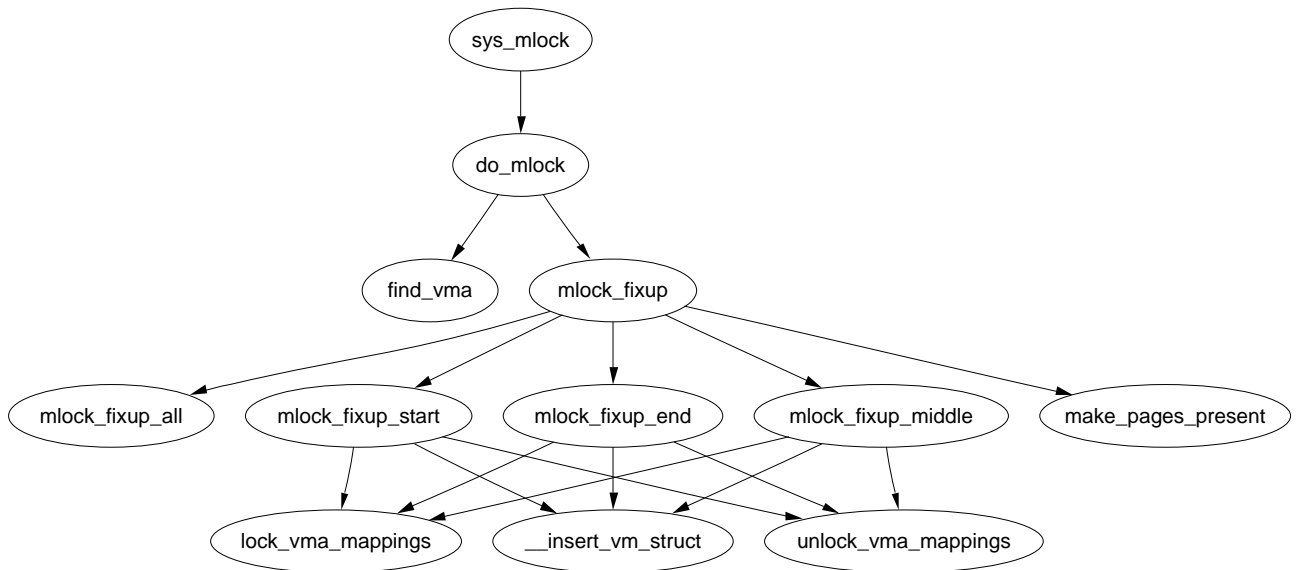
Next `move_page_tables()` is called, see Figure 9.8 for its call graph. This function copies all the page table entries from the old mapping to the new one. While there may be better ways to move the page tables, this method makes error recovery much easier as it is easy to backtrack if an error occurs during the page table move.

Figure 9.8: Call Graph: `move_page_tables`

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. The normal page fault handling code will either swap the pages back in from swap, files or call the device specific `do_nopage()` function.

9.3.8 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 9.9. At a high level, the function is pretty 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 the same functions page fault routines work. A second system call `mlockall()`

Figure 9.9: Call Graph: `sys_mlock`

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 `do_mmap()` to do the real work of finding the affected VMAs and deciding what function is needed to fix up the regions as described later.

There are some limitations. 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 limit is that the process limit `RLIMIT_MLOCK` imposed by the system administrator may not be exceeded. The last limit is that each process can only lock half of physical memory may be locked 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.

9.3.9 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 much checks. They both too rely on `do_mmap()` to do the work of fixing up the regions.

9.3.10 Fixing up regions after locking

When a locking or unlocking occurs, VMA's will be affected in one of four ways which has to 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 a new VMA is 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 VMA's to be allocated.

It is interesting to note that VMAs created as a result of a 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.

9.3.11 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 PTE's related to the region to be unmapped and the third is to fix up the regions if a hole has been generated.

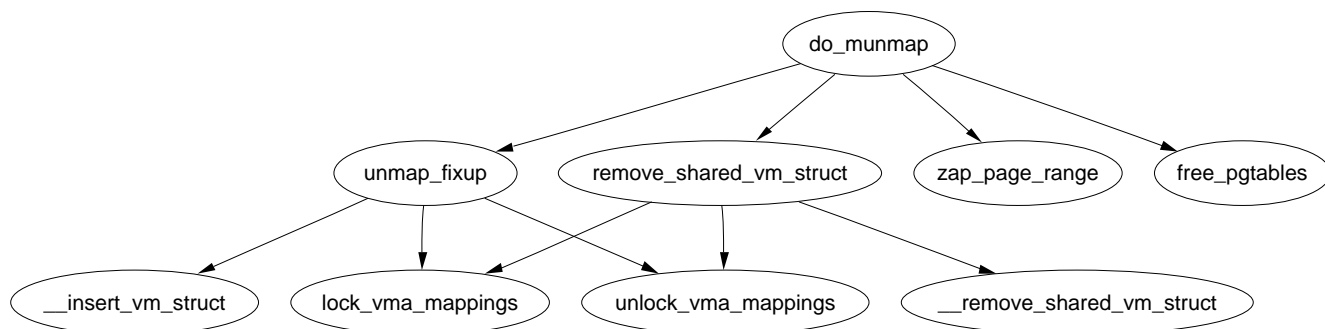


Figure 9.10: Call Graph: `do_munmap`

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 of free is walked through and checks are made 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 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.

9.3.12 Deleting all memory regions

During process exit, it is necessary to unmap all VMAs associated with a mm. 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 companion document.

9.4 Exception Handling

A very important part of VM is how exceptions related to bad kernel address references are caught³ which are not a result of a kernel bug⁴. 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.

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 module's 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 9.6 how this is used to trap bad reads and writes to userspace.

³Many thanks go to Ingo Oeser for clearing up the details of how this is implemented

⁴Of course bad references due to kernel bugs should rightfully cause the system to have a minor fit

9.5 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, writing a read-only page or simple programming error.

Linux, like most operating system, 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 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 $2^{page_cluster}$ is read in by `swpin_readahead()` and placed in the swap cache. Unfortunately there is not much guarantee that the pages placed in swap are related to each other or likely to be used soon.

There is two types of page fault, major and minor faults. Major fault have to have the data fetched from disk else it is referred to as a minor or soft page fault. Linux maintains statistics on these two types of fault with the `task_struct→majflt` and `task_struct→minflt` 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 9.4 which will be discussed in detail later in this section.

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 9.11.

This function is provided with a wealth of information such as the address of the fault, whether the page was simply not found or was a 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 it has and how it should be handled by the architecture independent code. The flow chart which shows broadly speaking what this function does is shown in Figure 9.12. In the figure, points with a colon after it is 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 invokes the out of memory handler.

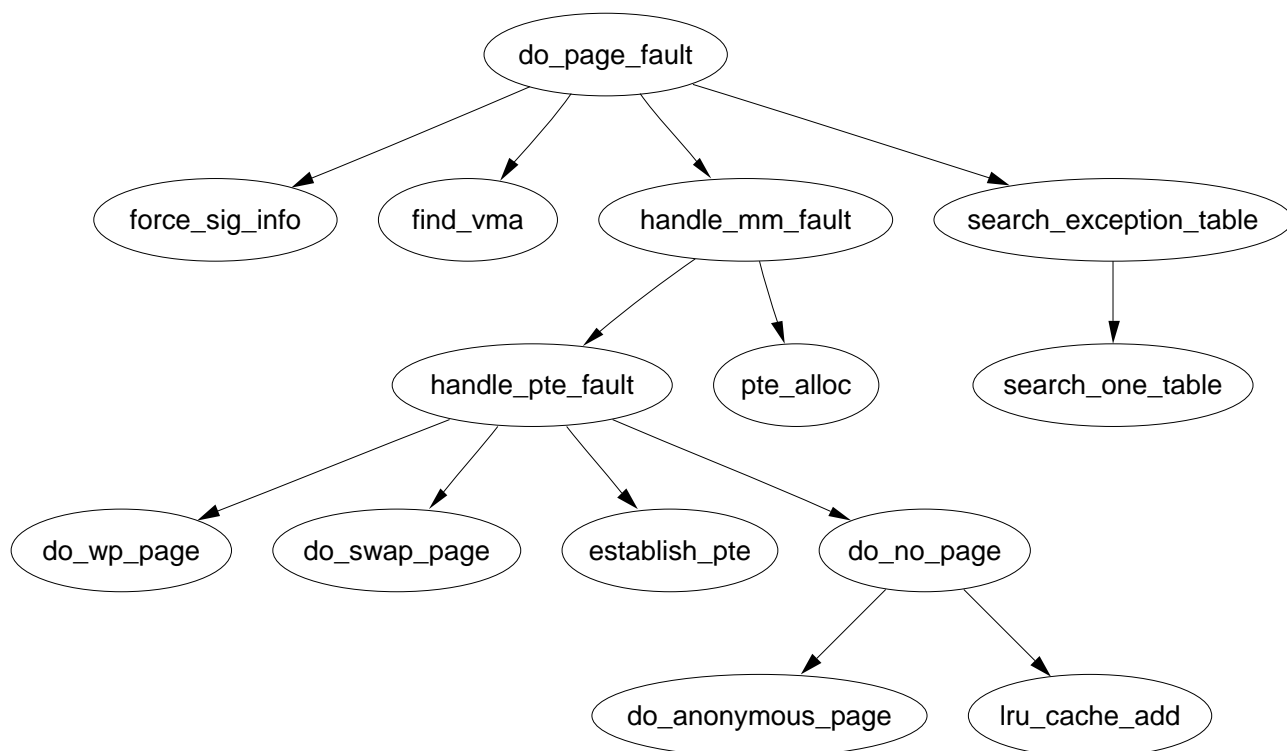
9.5.1 Handling a Page Fault

Once the exception handler has decided it is a normal page fault, `handle_mm_fault()`, whose call graph is shown in Figure 9.13, 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 the call graph will be used. The first checks are made if the PTE is marked not present

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 SEGSEV 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, a page is allocated and placed. 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 9.4: Reasons For Page Faulting

Figure 9.11: Call Graph: `do_page_fault`

as shown by `pte_present()` then `pte_none()` is called. If it returns there is no PTE, `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()` is what is required for **Demand Paging**.

The second option is if the page is been written to. If the PTE is write protected, then `do_wp_page()` is called as the page is a Copy-On-Write (COW) page as the VMA for the region is marked writable even if the individual PTE is not. Otherwise 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.

9.5.2 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 managing VMA has filled in the `vm_ops` struct and has supplied a `nopage()`, 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.

Handling anonymous pages If the struct is not filled in or a `nopage()` function is not supplied, the function `do_anonymous_page()` is called to handle an anonymous access which we will discuss first as it is simpler. 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 zeros⁵ is mapped for the PTE and the PTE is write protected. The PTE is write protected so another page fault will occur if the process writes to the page.

If this is the first write to the page `alloc_page()` is called to allocate a free page (see Chapter 6) and is zero filled by `clear_user_highpage()`. Assuming the page was successfully allocated the **Resident Set Size (rss)** field in the `mark_structmm_struct` will be incremented, `flush_page_to_ram()` is called as it is required when a page is been inserted into a userspace process by some architectures to ensure cache coherency, the page is inserted on the LRU lists so it may be reclaimed later by the swapping code and 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 pages worth 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 should an early COW break 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 isn't 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.

9.5.3 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.

⁵On the x86, it is zeroed out in the function `mem_init()`

A shared page can not be swapped out immediately because there is no way of mapping a struct page to the PTE's 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 PTE's a page is mapped by are linked together in 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's 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 $2^{page_cluster}$ 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.

9.5.4 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 PTE's of the two processes are made read-only so that when a write occurs there will be a page fault. Linux recognizes a COW page because even though the PTE is write protected, the controlling VMA shows the region is writable. It uses the function `do_wp_page()` 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.

9.6 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. Instead, Linux provides an ample API for copying data to and from user safely as shown in Table 9.5.

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()` maps on to one of two functions `__constant_copy_from_user()` or `__generic_copy_from_user()` depending on if the size of the copy is known at compile time or not. If the size is known at compile time, there is different assembler

<code>copy_from_user(void *to, const void *from, unsigned long n)</code>
Copies <code>n</code> bytes from the user address space (<code>from</code>) to the kernel address space (<code>to</code>)
<code>copy_to_user(void *to, const void *from, unsigned long n)</code>
Copies <code>n</code> bytes from the kernel address space (<code>from</code>) to the user address space (<code>to</code>)
<code>get_user(void *to, void *from)</code>
Copies an integer value from userspace (<code>from</code>) to kernel space (<code>to</code>)
<code>put_user(void *from, void *to)</code>
Copies an integer value from kernel space (<code>from</code>) to userspace (<code>to</code>)
<code>strncpy_from_user(char *dst, const char *src, long count)</code>
Copies a null terminated string of at most <code>count</code> bytes long from userspace (<code>src</code>) to kernel space (<code>dst</code>)
<code>strlen_user(const char *s, long n)</code>
Returns the length, upper bound by <code>n</code> , of the userspace string including the terminating NULL

Table 9.5: Accessing Process Address Space API

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 `include/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.

These pairings, as described in Section 9.4, 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.

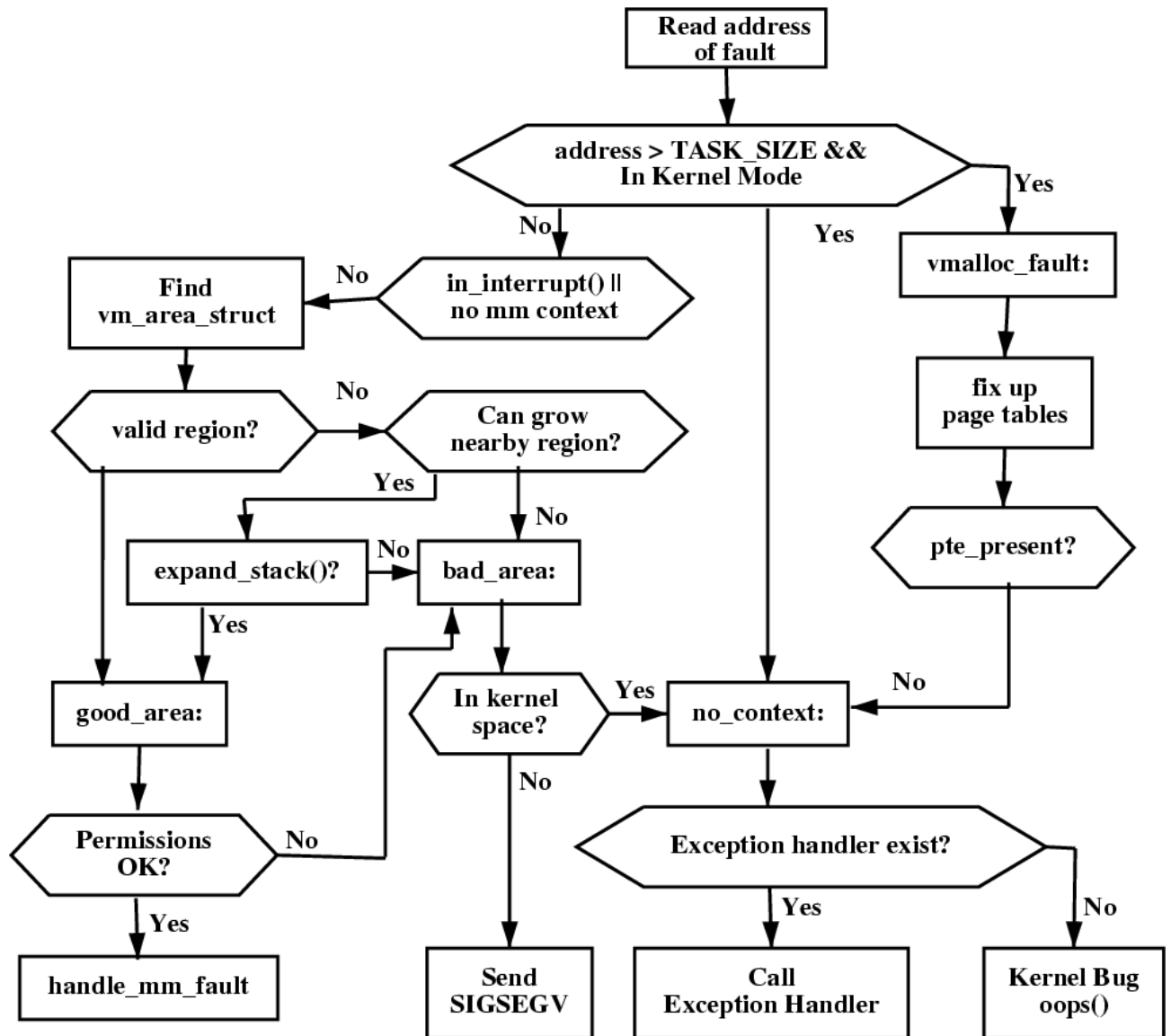
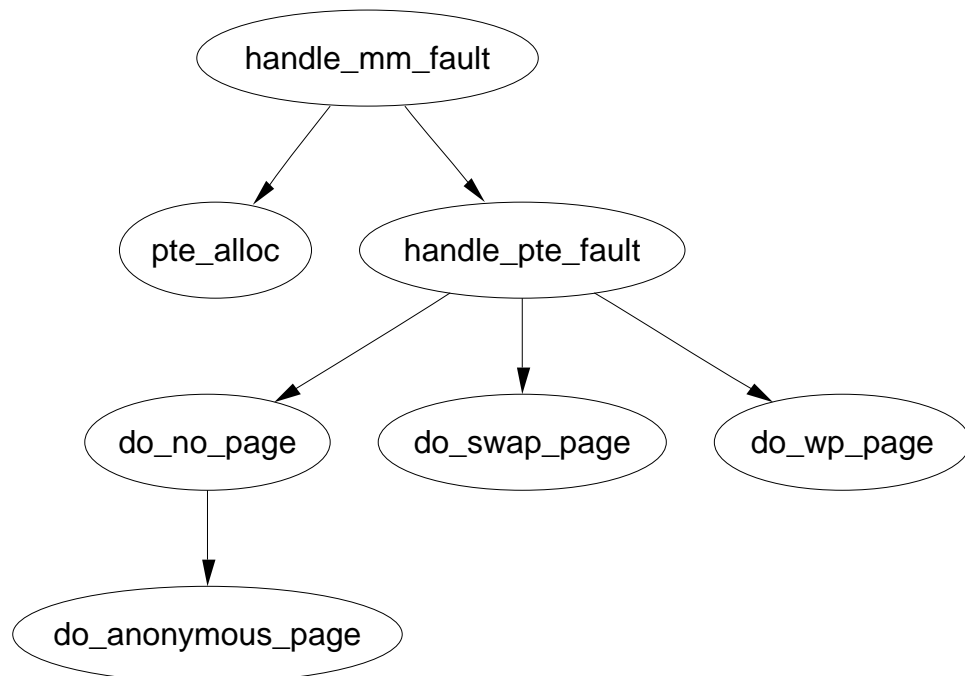
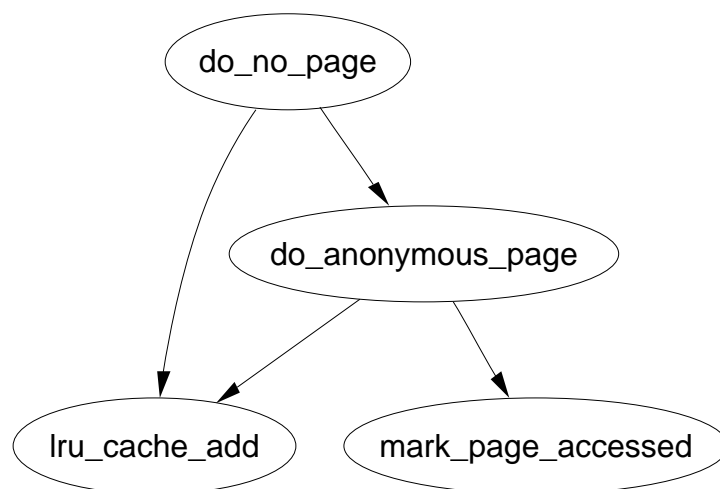
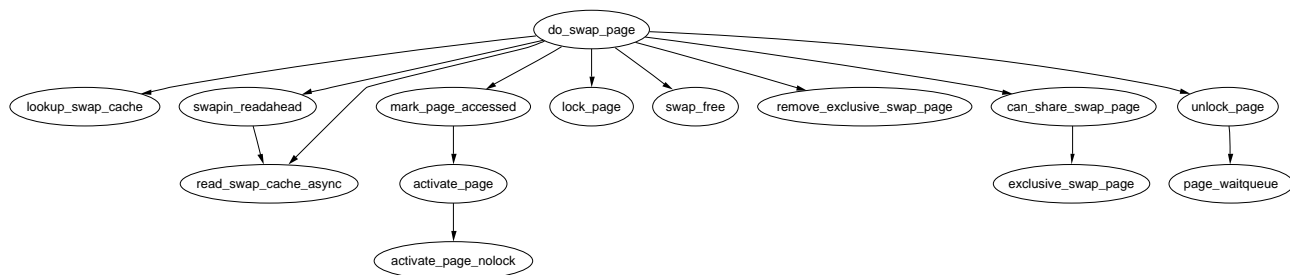
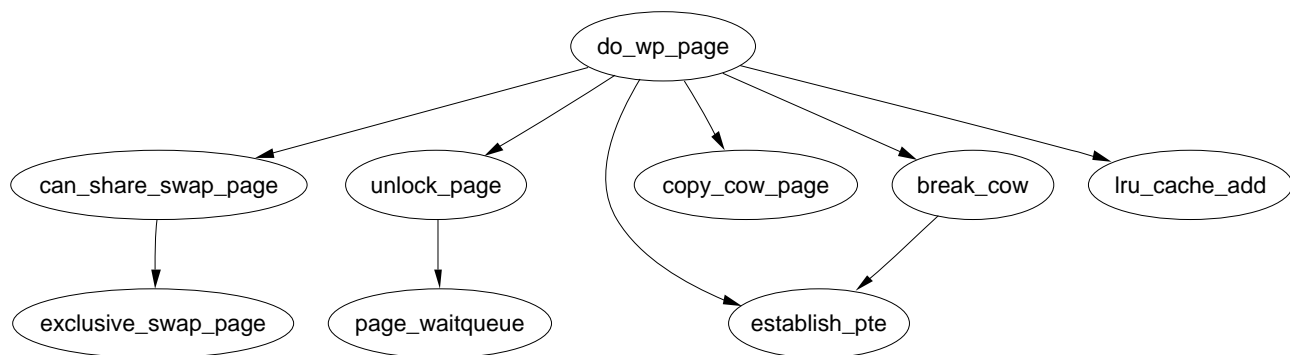


Figure 9.12: do_page_fault Flow Diagram

Figure 9.13: Call Graph: `handle_mm_fault`Figure 9.14: Call Graph: `do_no_page`

Figure 9.15: Call Graph: `do_swap_page`Figure 9.16: Call Graph: `do_wp_page`

Chapter 10

High Memory Management

Chapter 11

Page Frame Reclamation

A running system will eventually use all page frames for purposes like disk buffers, dentries, inode entries or process pages. Linux needs to begin selecting old pages which can be freed and invalidated for new uses before physical memory is exhausted. This section 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.

All pages, except those used by the slab allocator, in use by the system are initially stored on the page cache 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 pages are stored in the page cache but are not easily swappable as there is no way to map page structs to PTE's 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 has 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 type of shared pages are considered to be in the **swap cache**.

11.1 Page Swap 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` that 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).

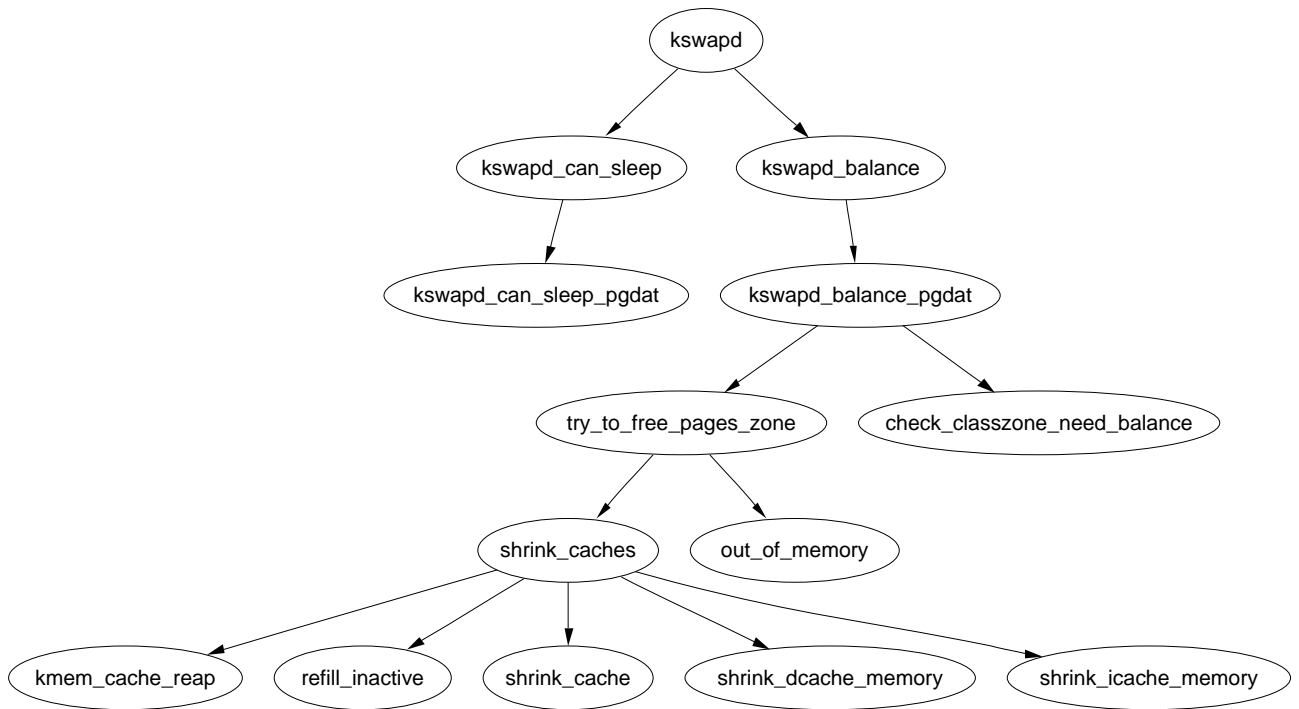


Figure 11.1: Call Graph: kswapd

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[JM01] which is woken up with increasing frequency as there is memory pressure, kswapd keeps freeing pages until the `pages_high` limit 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 `zone_t` struct. 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 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 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`. The objective is for the `active_list` to contain the **working set**[Den70] and the `inactive_list` contain pages that can be reclaimed.

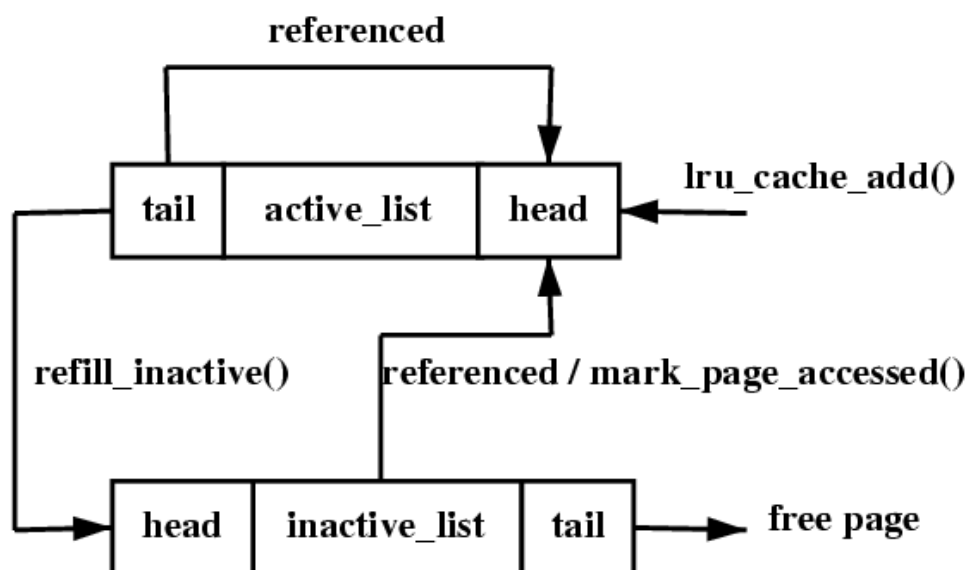


Figure 11.2: Page Cache LRU List

The page cache is generally said to use a **Least Recently Used (LRU)** based replacement algorithm but that is not strictly speaking true as the lists are not strictly maintained in LRU order. They instead resemble a simplified LRU 2Q[JS94] where two lists called **Am** and **A1** are maintained. 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.

The lists described for 2Q presumes **Am** is an LRU list but the list in Linux closer resembles a Clock algorithm[Car84] where the handspread 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`.

11.3 Shrinking all caches

The function responsible for shrinking the various caches is `shrink_caches()`. It 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 it calls `kmem_cache_reap()` (See Section 8.1.7) which selects a slab cache to shrink. If `nr_pages` number of pages are freed, the work is complete and the function returns otherwise it will try to free `nr_pages` from other caches.

If other caches are to be affected, `refill_inactive()` to move pages from the `active_list` to the `inactive_list` discussed in the next subsection.

Next it shrinks the page cache by reclaiming pages at the end of the `inactive_list` with `shrink_cache()`. If there is a large number of pages in the queue that belong to processes, whole processes will be swapped out with `swap_out()`

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.

11.4 Page Hash

11.5 Inode Queue

11.6 Refilling inactive_list

Every time caches are being shrunk by the function `shrink_caches()`, 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 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

$$pages = nr_pages * \frac{nr_active_pages}{2 * (nr_inactive_pages + 1)}$$

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

²Currently statically defined as 6 in `mm/vmscan.c`



Figure 11.3: Call Graph: shrink_caches

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.

11.7 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 should it reschedule itself if it has used up it's quanta 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` is decremented. If it reaches 0, the page tables of processes will be linearly searched and swapped out started by the function `swap_out()`

Page is locked and the PG_laundry 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, `page_cache_release()` is called to decrement the reference count. 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_laundry` bit is set. A reference to the page is taken with `page_cache_get()` so the page will not disappear prematurely and then the provided `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 is too many process mapped pages in the page cache. An anonymous page may have associated buffers if a file was truncated and immediately followed by a page fault.

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.

11.8 Swapping Out Process Pages

When the `max_mapped` number of pages has been found in the page cache, `swap_out()` (See Figure 11.4) is called to start swapping out process pages. Starting from the `mm` pointed to by `swap_mm` and the address `mm→swap_address`, the page tables are searched forward until `nr_pages` have been freed.

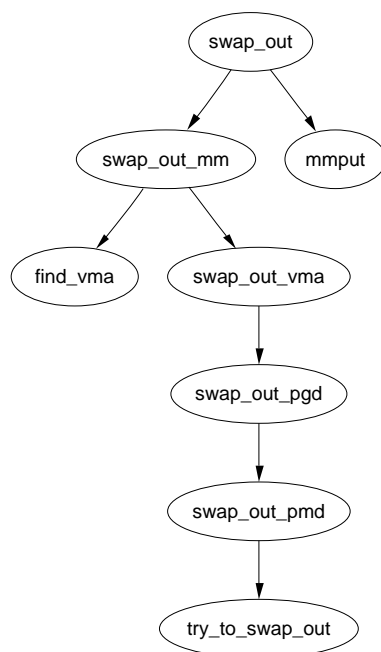


Figure 11.4: Call Graph: `swap_out`

All 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 nothing in comparison to linearly searching all processes for the PTE's 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 so avoid constant looping when no pages are available. Writing out the pages in bulk like this 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` 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` 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 isn't 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.

If this is the first time the page has been swapped, a swap entry is allocated for it with `get_swap_page()` and the page is added to the swap cache. If the page is already part of the swap cache, the reference to it in the current process will be simply dropped, when it reaches 0, the page will be freed. Once in the swap cache, the PTE in the process page tables will be updated with the information needed to get the page from swap again. This is important because it means the PTE's for a process can never be swapped out or discarded.

`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

`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 had not been 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()` which marks the page hot is called and the referenced flag 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)`

Removed a page from the `inactive_list` and placed 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

12.1 Swap Cache

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 it 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.

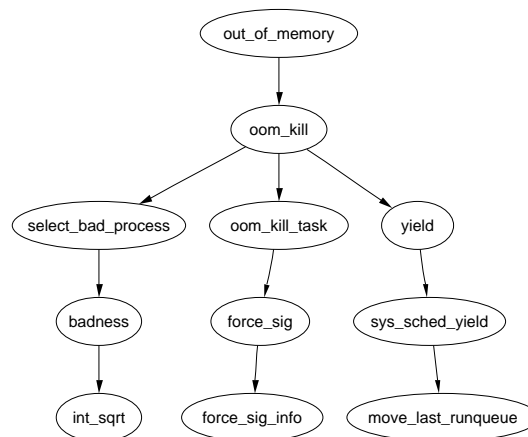


Figure 13.1: Call Graph: `out_of_memory`

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.

13.0.1 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()`;

$$badness_for_task = \frac{total_vm_for_task}{\sqrt{(cpu_time_in_seconds)} * \sqrt[4]{(cpu_time_in_minutes)}}$$

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.

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.

Bibliography

- [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.
- [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.
- [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*, CONFERENCE PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON SUPERCOMPUTING 1995; 9th, 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
. In <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
. In <ftp://ftp.kernel.org/pub/linux/kernel/v2.4/linux-2.4.18.tar.gz>.
- [Hac00] Random Kernel Hacker. Submittingpatches (how to get your change into the linux kernel). *Kernel Source Documentation Tree*, 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.
- [JM01] Richard McDougall Jim Maura. *Solaris Internals*. Rachael Borden, 2001.
- [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.

- [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.
- [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.
- [Rus] Paul Rusty Russell. Unreliable guide to locking. *Kernel Source Documentation Tree*.
- [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

active_list, 120
address_space, 96
alloc_bootmem, 40
 __alloc_bootmem, 41
alloc_bootmem_low, 41
 __alloc_bootmem_core, 41
alloc_bootmem_low_pages, 41
alloc_bootmem_low_pages_node, 41
alloc_bootmem_node, 41
 __alloc_bootmem_node, 41
alloc_bootmem_pages, 41
alloc_bootmem_pages_node, 41
alloc_mm, 92
allocate_mm, 92, 93
arch_get_unmapped_area, 102

Binary Buddy Allocator, 46
Boot Memory Allocator, 37
bootmem_data, 37

cache chain, 61
cache_cache, 83
cache_sizes, 80
cache_sizes_t, 79
cc_data, 82
cc_entry, 82
ccupdate_t, 83
CFGFS_OFF_SLAB, 73, 85
CFLGS_OPTIMIZE, 85
check_pgt_cache, 34
clear_user_highpage, 113
ClearPageActive, 27
ClearPageDirty, 27
ClearPageError, 27
ClearPageLaunder, 27
ClearPageReferenced, 27
ClearPageReserved, 27
ClearPageUptodate, 27

clock_searchp, 70
CONFIG_SLAB_DEBUG, 66
contig_page_data, 18
copy-on-write (COW), 115
copy_from_user, 87
copy_mm, 93
copy_to_user, 87
cpu_vm_mask, 92
cpucache, 81

def_flags, 92
DEF_PRIORITY, 121
Demand Allocation, 112
Demand Fetch, 109
Demand Paging, 112
DFLGS_GROWN, 67
diff, 11
do_anonymous_page, 113
do_ccupdate_local, 83
do_mmap_pgoff, 100
do_munmap, 107
do_no_page, 112
do_page_fault, 109
do_swap_page, 28, 112, 114
do_wp_page, 112, 115

empty_zero_page, 113
enable_all_cpucaches, 83
enable_cpucache, 82
exception_table_entry, 108
 _end, 39
exit_mmap, 93

filemap_nopage, 113
 __ex_table, 108
find_max_low_pfn, 39
find_max_pfn, 39
find_vma, 100

- find_vma_intersection, 102
- find_vma_prepare, 103
- find_vma_prev, 100
- fixrange_init, 36
- flush_page_to_ram, 113
- free_all_bootmem, 42
- free_all_bootmem_node, 42
- free_area_t, 47
- free_bootmem, 42
- free_bootmem_node, 42
- free_initmem, 43
- free_mm, 93
- free_pages_init, 42
- free_pgtables, 107
- g_cpucache_up, 83
- gengraph, 8
- GET_PAGE_CACHE, 72
- GET_PAGE_SLAB, 72
- get_pgd_fast, 34
- get_pgd_slow, 34
- get_unmapped_area, 102
- GFP (Get Free Page), 51
- GFP_ATOMIC, 53, 54
- GFP_DMA, 51
- __GFP_DMA, 51
- GFP_HIGHUSER, 53, 54
- __GFP_FS, 52
- GFP_KERNEL, 53, 54
- __GFP_HIGH, 52
- GFP_KSWAPD, 53, 54
- __GFP_HIGHIO, 52
- GFP_NFS, 53, 54
- __GFP_HIGHMEM, 51
- GFP_NOFS, 53, 54
- __GFP_IO, 52
- GFP_NOHIGHIO, 53, 54
- GFP_NOIO, 53, 54
- GFP_USER, 53, 54
- handle_mm_fault, 111
- __GFP_WAIT, 52
- handle_pte_fault, 112
- highend_pfn, 38
- highstart_pfn, 38
- inactive_list, 120
- INIT_MM, 93
- __init, 43
- init_mm, 93
- __init_begin, 43
- insert_vm_struct, 103
- __init_end, 43
- kfree, 81
- kmalloc, 80
- kmap, 56
- kmem_bufctl_t, 74
- kmem_cache, 83
- kmem_cache_init, 84
- kmem_cache_slabmgmt, 73
- kmem_freepages, 84
- kmem_getpages, 84
- kmem_tune_cpucache, 82
- kswapd, 118
- kswapd_wait, 119
- kunmap, 56
- Lazy TLB, 89
- Least Recently Used (LRU), 120
- locked_vm, 92
- LockPage, 27
- LXR, 13
- MARK_USED, 47
- MAX_DMA_ADDRESS, 41
- max_low_pfn, 38
- max_mapped, 122
- MAX_NR_ZONES, 24
- MAX_ORDER, 46
- max_pfn, 38
- max_scan, 122
- mem_init, 42
- merge_segments, 104
- min_low_pfn, 38
- mk_pte, 33
- mk_pte_phys, 33
- mlock_fixup, 106
- mlock_fixup_all, 107
- mlock_fixup_end, 107
- mlock_fixup_middle, 107
- mlock_fixup_start, 107

- mm_alloc, 93
- mm_count, 90
- mm_drop, 93
- mm_init, 92, 93
- mm_struct, 89
- mm_users, 90
- mmap_sem, 92
- mmlist, 92
- mmput, 93
- move_page_tables, 105
- move_vma, 104
- mremap, 104
- nr_pages, 122
- one_highpage_init, 42
- Page Frame Number (PFN), 37, 38
- Page Frame Number (pfn), 39
- page struct, 23
- PAGE_ALIGN, 30
- page_cluster, 115
- PAGE_OFFSET, 87
- PAGE_SHIFT, 30
- PageActive, 27
- PageChecked, 27
- PageClearSlab, 27
- PageDirty, 27
- PageError, 27
- PageHighMem, 27
- PageLaunder, 27
- PageLocked, 27
- PageLRU, 27
- PageReferenced, 27
- PageReserved, 27
- pages_high, 23
- pages_low, 22
- pages_min, 22
- PageSetSlab, 27
- PageSlab, 27
- PageUptodate, 27
- paging_init, 36
- patch, 11
- pg0, 35
- pg1, 35
- PG_active, 26
- PG_arch_1, 26
- PG_checked, 26
- pg_data_t, 19
- __pgd, 30
- PG_dirty, 26
- PG_error, 26
- PG_highmem, 26
- PG_launder, 26
- PG_locked, 26
- PG_lru, 26
- PG_referenced, 26
- __pgprot, 30
- PG_reserved, 26
- PG_skip, 26
- PG_slab, 26
- PG_unused, 26
- PG_uptodate, 26
- PGD, 28
- pgd_alloc, 34
- pgd_free, 34
- pgd_quicklist, 34
- pgd_t, 28
- pgd_val, 30
- PGDIR_SHIFT, 30
- pglist_data, 19
- pgprot_t, 30
- pgprot_val, 30
- PKMAP_BASE, 56
- pmd_alloc, 34
- __pmd, 30
- pmd_alloc_one, 34
- pmd_alloc_one_fast, 34
- pmd_free, 34
- pmd_page, 33
- pmd_quicklist, 34
- PMD_SHIFT, 30
- pmd_t, 28
- pmd_val, 30
- pte_alloc, 34
- __pte, 30
- pte_alloc_one, 34
- pte_alloc_one_fast, 34
- pte_clear, 33
- pte_dirty, 33
- pte_exec, 33

- pte_exprotect, 33
- pte_free, 34
- pte_mkclean, 33
- pte_mkdirty, 33
- pte_mkexec, 33
- pte_mkread, 33
- pte_mkwrite, 33
- pte_mkyoung, 33
- pte_modify, 33
- pte_old, 33
- pte_page, 33
- pte_quicklist, 34
- pte_rdprotect, 33
- pte_read, 33
- pte_t, 28
- pte_val, 30
- pte_write, 33
- pte_wrprotect, 33
- pte_young, 33
- ptep_get_and_clear, 33
- PTRS_PER_PGD, 30
- PTRS_PER_PMD, 30
- PTRS_PER_PTE, 30
- quicklists, 34
- REAP_SCANLEN, 70
- Resident Set Size (RSS), 92
- Resident Set Size (rss), 113
- Reverse Mapping (rmap), 114
- rss, 92
- search_exception_table, 108
- SET_PAGE_CACHE, 72
- set_pte, 33
- SetPageActive, 27
- SetPageChecked, 27
- SetPageDirty, 27
- SetPageError, 27
- SetPageLaunder, 27
- SetPageReferenced, 27
- SetPageReserved, 27
- SetPageUptodate, 27
- setup_arch, 38
- setup_memory, 38
- size-X cache, 79
- size-X(DMA) cache, 79
- slab descriptor, 73
- SLAB_ATOMIC, 86
- slab_bufctl, 75
- SLAB_CACHE_DMA, 86
- SLAB_DMA, 86
- SLAB_HWCACHE_ALIGN, 86
- SLAB_KERNEL, 86
- SLAB_NFS, 86
- SLAB_NO_REAP, 86
- SLAB_NOFS, 86
- SLAB_NOHIGHIO, 86
- SLAB_NOIO, 86
- SLAB_USER, 86
- slabs, 61
- slabs_free, 64
- slabs_full, 64
- slabs_partial, 64
- startup_32, 35
- struct kmem_cache_s, 64
- struct page, 23
- swap cache, 118
- SWAP_CLUSTER_MAX, 121
- swap_mm, 124
- swap_out_mm, 125
- swap_out_vma, 125
- swpin_readahead, 109, 114
- swapper_pg_dir, 35
- sys_mlock, 105
- sys_mlockall, 105
- sys_mmap2, 100
- sys_mprotect, 104
- sys_mremap, 104
- sys_munlock, 106
- sys_munlockall, 106
- TestClearPageLRU, 27
- TestSetPageLRU, 27
- total_vm, 92
- tq_disk, 119
- Trivial Patch Monkey, 16
- try_to_free_buffers(), 52
- UnlockPage, 27
- unmap_fixup, 107

vm_struct, 57
vma_link, 103
vma_merge, 104
vmalloc, 52, 56
VMALLOC_OFFSET, 56
vmlist_lock, 57

working set, 120

Zone watermarks, 22
ZONE_DMA, 18
ZONE_HIGHMEM, 18
ZONE_NORMAL, 18
zone_t, 21
zone_table, 24
Zones, 21

[Gor02] [Knu68] [Vah96] [McK96] [JM01] [CS98] [BC00] [RC01] [GC94] [Tan01]
[MM87] [BBD⁺98] [JS94] [Bon94] [BA01] [KB85] [JW98] [BL89] [HK97] [GAV95]
[Hacb] [Haca] [CP99] [Ous90] [CH81] [PN77] [Kno65] [WJNB95] [Rus] [Den70]
[Hac00] [Mil00] [Sho75] [Kes91]