# Preface

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

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

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

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

## The Intended Audience

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

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

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

## Book Overview

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

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

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

## What's New in 2.6

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

## Companion CD

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

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

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

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

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

- A live version of CodeViz, the tool used to generate call graphs for the book, is available. If you feel that the book's graphs are lacking some detail you want, generate them yourself.
- The **VMRegress**, **CodeViz** and **PatchSet** packages, which are discussed in Chapter 1, are available in `/cdrom/software`. **gcc-3.0.4** is also provided because it is required for building **CodeViz**.

Mount the CD on `/cdrom` as follows:

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

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

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

The URL to access is http://localhost:10080/
```

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

## Typographic Conventions

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

## Acknowledgments

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

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

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

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

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

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

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

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

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

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

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

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

CHAPTER 1

# Introduction

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

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

**Table 1.1.** Kernel Size as an Indicator of Complexity

Out of habit, open source developers tell new developers with questions to refer directly to the source with the "polite" acronym RTFS[1], or refer them to the kernel newbies mailing list (*http://www.kernelnewbies.org*). With the Linux VM manager, this used to be a suitable response because the time required to understand the VM could be measured in weeks. Moreover, the books available devoted enough time to the memory management chapters to make the relatively small amount of code easy to navigate.

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

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

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

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

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

## 1.1 Getting Started

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

### 1.1.1 Configuration and Building

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

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

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

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

[2] *http://www.gnu.org/software/autoconf/*
[3] *http://www.gnu.org/software/automake/*

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

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

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

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

A comprehensive guide on configuring and compiling a kernel is available with the Kernel HOWTO[4] and will not be covered in detail with this book. For now, we will presume you have one fully built kernel, and it is time to begin figuring out how the new kernel actually works.

### 1.1.2 Sources of Information

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

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

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

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

[4] *http://www.tldp.org/HOWTO/Kernel-HOWTO/index.html*

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

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

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

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

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

## 1.2 Managing the Source

The mainline or stock kernel is principally distributed as a compressed tape archive (.tar.bz) file that is available from your nearest kernel source repository. In Ireland's case, it is *ftp://ftp.ie.kernel.org/*. The stock kernel is always considered to be the one released by the tree maintainer. For example, at time of writing, the stock kernels for 2.2.x are those released by Alan Cox[5], for 2.4.x by Marcelo Tosatti and for 2.5.x by Linus Torvalds. At each release, the full tar file is available as well as a smaller *patch*, which contains the differences between the two releases. Patching is the preferred method of upgrading because of bandwidth considerations. Contributions made to the kernel are almost always in the form of patches, which are *unified diffs* generated by the GNU tool **diff**.

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

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

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

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

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

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

### 1.2.1 Diff and Patch

The two tools for creating and applying patches are **diff** and **patch**, both of which are GNU utilities available from the GNU website[6]. **diff** is used to generate patches, and **patch** is used to apply them. Although the tools have numerous options, there is a "preferred usage."

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

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

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

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

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

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

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

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

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

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

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

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

```
--- linux-2.4.22-clean/mm/page_alloc.c Thu Sep  4 03:53:15 2003
+++ linux-2.4.22-mel/mm/page_alloc.c Thu Sep  3 03:54:07 2003
@@ -76,8 +76,23 @@
  * triggers coalescing into a block of larger size.
  *
  * -- wli
+ *
+ *  There is a brief explanation of how a buddy algorithm works at
+ *  http://www.memorymanagement.org/articles/alloc.html . A better
+ *  idea is to read the explanation from a book like UNIX Internals
+ *  by Uresh Vahalia
+ *
  */

+/**
+ *
+ * __free_pages_ok - Returns pages to the buddy allocator
+ * @page: The first page of the block to be freed
+ * @order: 2^order number of pages are freed
+ *
+ * This function returns the pages allocated by __alloc_pages and
+ * tries to merge buddies if possible. Do not call directly, use
+ * free_pages()
+ **/
 static void FASTCALL(__free_pages_ok (struct page *page, unsigned
 int order));
 static void __free_pages_ok (struct page *page, unsigned int order)
 {
```

**Figure 1.1.** Example Patch

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

### 1.2.2 Basic Source Management With PatchSet

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 1.3 Browsing the Code

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

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

A more friendly browsing method is available with the *LXR* tool hosted at *http://lxr.linux.no/*. This tool provides the ability to represent source code as browsable Web pages. Identifiers such as global variables, macros and functions become hyperlinks. When clicked, the location where the identifier is defined is displayed along with every 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 simple to install, and a browsable version of the kernel `2.4.22` source is available on the CD included with this book. All code extracts throughout the book are based on the output of LXR so that the line numbers would be clearly visible in excerpts.

### 1.3.1 Analyzing Code Flow

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

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

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

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

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

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

### 1.3.2 Simple Graph Generation

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

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

## 1.4 Reading the Code

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

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

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

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

and **dbench**(*http://freshmeat.net/projects/dbench/*). For many purposes, these benchmarks will fit the requirements.

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

## 1.5 Submitting Patches

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

First and foremost, the coding style of the kernel needs to be adhered to because having a style inconsistent with the main kernel will be a barrier to getting merged regardless of the technical merit. After a patch has been developed, the first problem is to decide where to send it. Kernel development has a definite, if nonapparent, hierarchy of who handles patches and how to get them submitted. As an example, we'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 no bad reaction occurs, it can be fed to what is called the *Trivial Patch Monkey*[7]. The trivial patch monkey is exactly what it sounds like. It takes small patches and feeds them en masse to the correct people. This is best suited for documentation, commentary or one-liner patches.

Patches are managed through what could be loosely called a set of rings with Linus in the very middle having the 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 enough.

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

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

one. The VM does not have a list of maintainers, but it does have a mailing list[8].

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

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

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

CHAPTER 2

# Describing Physical Memory

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

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

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

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

With the x86, the zones are the following:

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

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

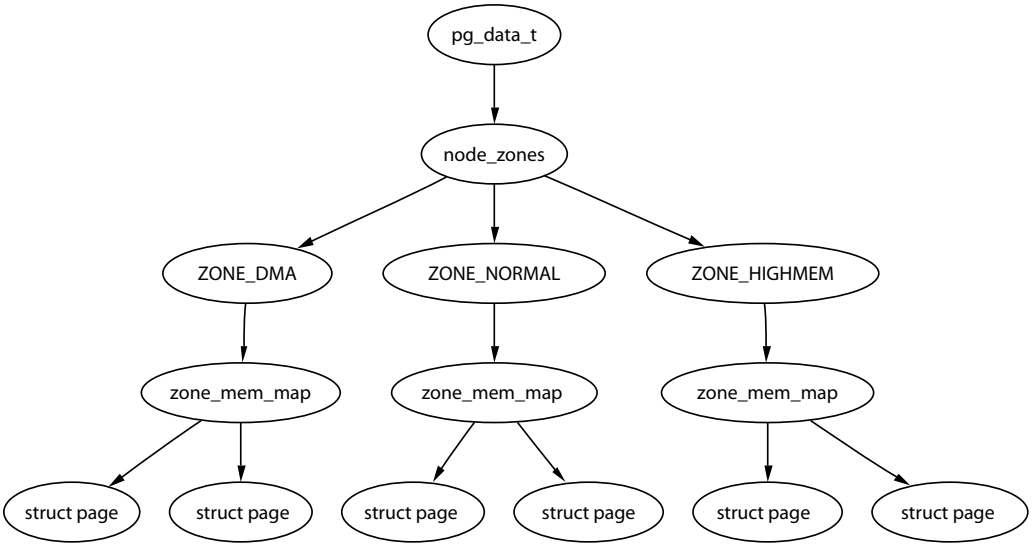


**Figure 2.1.** Relationship Between Nodes, Zones and Pages

the area reserved for the loaded kernel image in low memory machines. Section 2.4 discusses `struct pages` in detail, and Section 3.7 discusses the global `mem_map` array in detail. The basic relationship between all these structs is illustrated in Figure 2.1.

Because the amount of memory directly accessible by the kernel (`ZONE_NORMAL`) is limited in size, Linux supports the concept of *high memory*, which is discussed further in Section 2.7. This chapter discusses how nodes, zones and pages are represented before introducing high memory management.

## 2.1 Nodes

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

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

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

We now briefly describe each of these fields:

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

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

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

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

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

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

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

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

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

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

**node_next** Pointer to next node in a NULL terminated list.

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

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

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

## 2.2 Zones

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

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

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

**lock** Spinlock protects the zone from concurrent accesses.

**free_pages** The total number of free pages in the zone.

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

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

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

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

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

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

**zone_pgdat** This points to the parent `pg_data_t`.

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

**zone_start_paddr** This uses the same principle as `node_start_paddr`.

**zone_start_mapnr** This uses the same principle as `node_start_mapnr`.

**name** This is the string name of the zone: "DMA", "Normal" or "HighMem".

**size** This is the size of the zone in pages.

### 2.2.1 Zone Watermarks

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

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

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

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

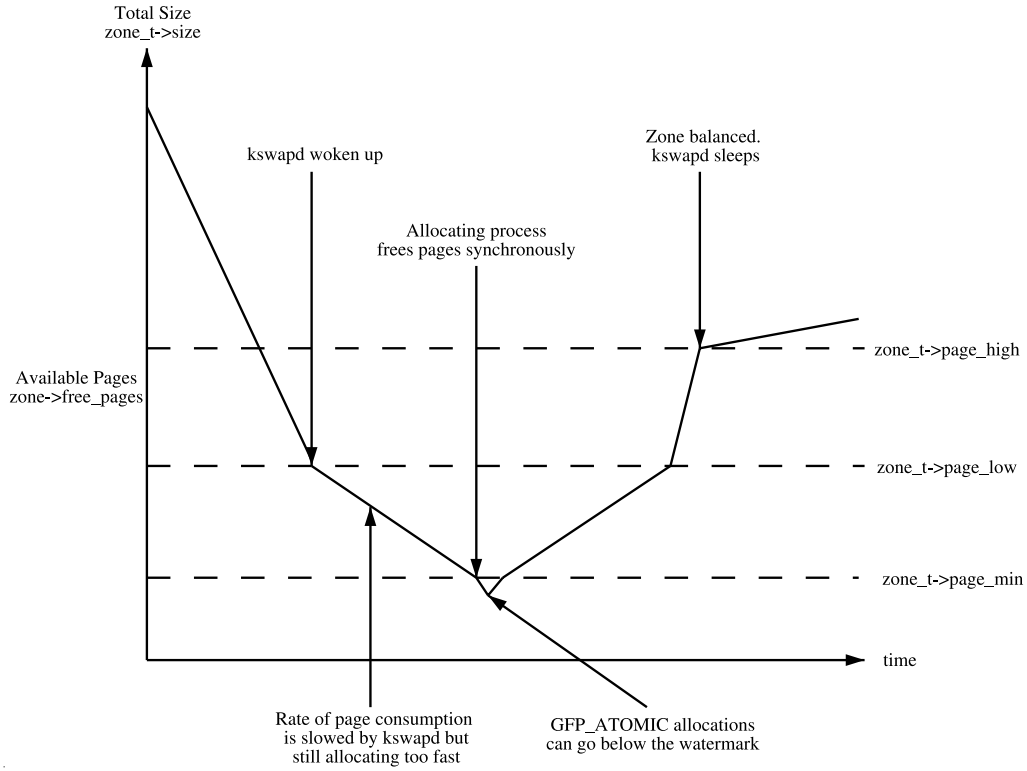


**Figure 2.2.** Zone Watermarks

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

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

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

### 2.2.2 Calculating the Size of Zones

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

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

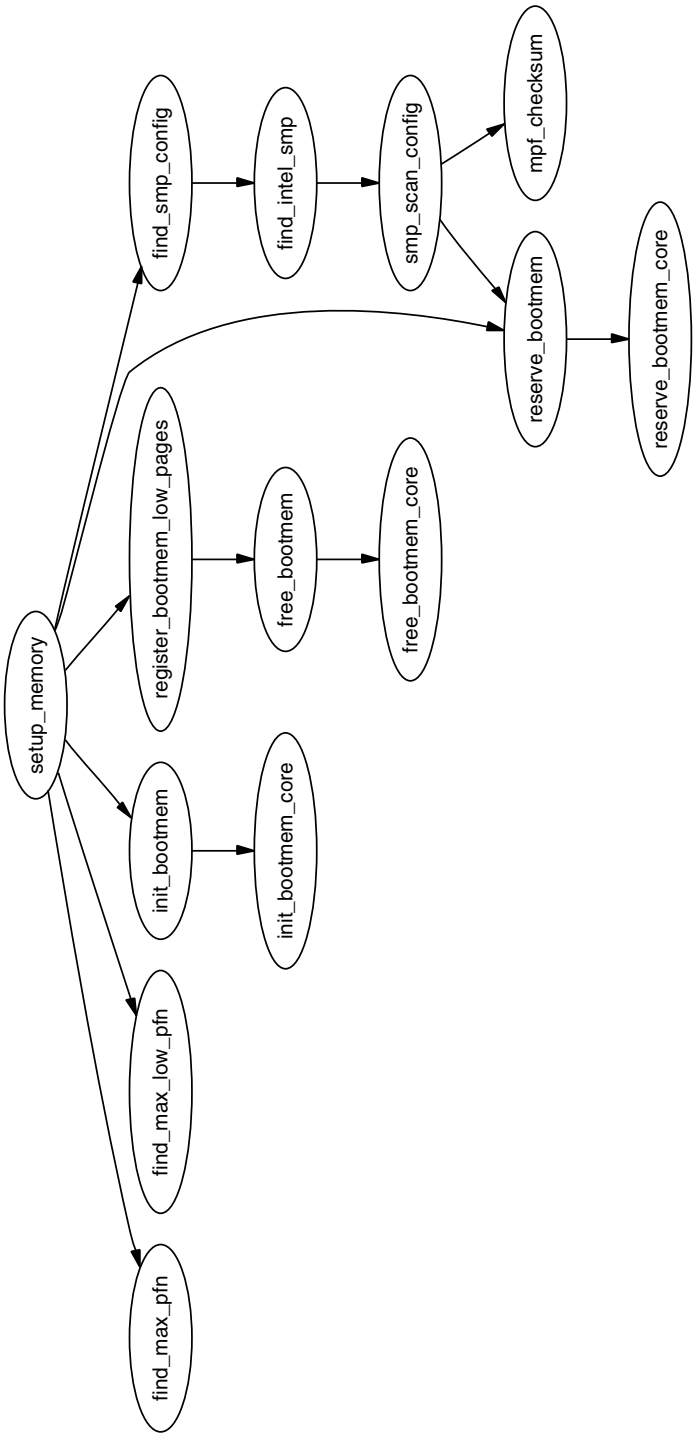


**Figure 2.3.** Call Graph: `setup_memory()`

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

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

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

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

### 2.2.3 Zone Wait Queue Table

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

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

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

$$\text{wait\_table\_size} = \log_2\left(\frac{\text{NoPages} * 2}{\text{PAGE\_PER\_WAITQUEUE}} - 1\right)$$

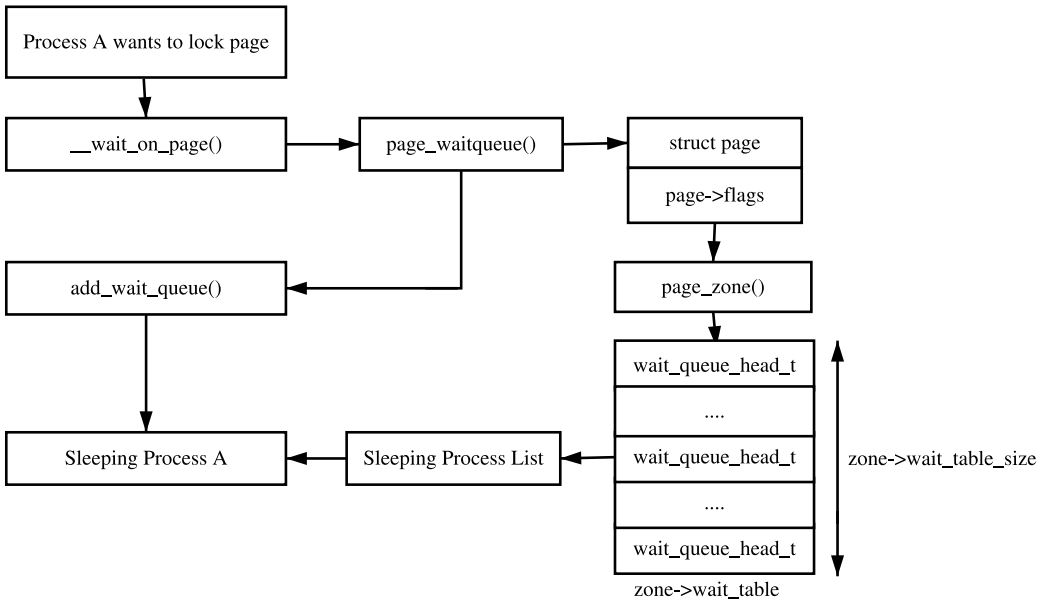


**Figure 2.4.** Sleeping on a Locked Page

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

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

## 2.3 Zone Initialization

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

**nid** is the NodeID that is the logical identifier of the node whose zones are being initialized.

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

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

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

**zone_start_paddr** is the starting physical address for the first zone.

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

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

## 2.4 Initializing mem_map

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

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

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

## 2.5 Pages

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

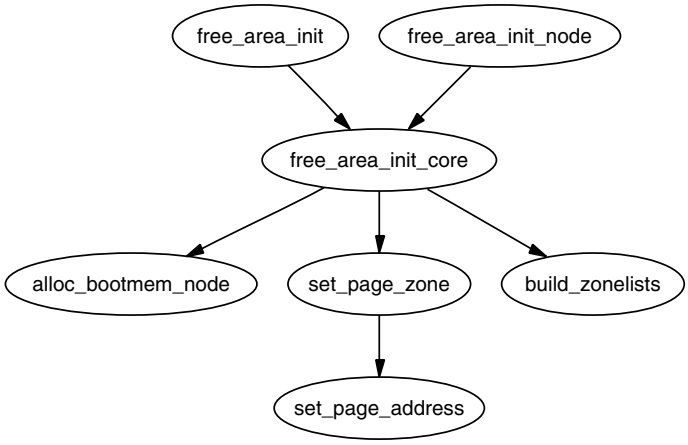


**Figure 2.5.** Call Graph: `free_area_init()`

its equivalent in System V [GC94], but like the other UNIX variants, the structure changed considerably. It is declared as follows in <`linux/mm.h`>:

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

Here is a brief description of each of the fields:

**list** Pages may belong to many lists, and this field is used as the list head. For example, pages in a mapping will be in one of three circular linked links kept by the `address_space`. These are `clean_pages`, `dirty_pages` and `locked_pages`. In the slab allocator, this field is used to store pointers to the slab and cache structures managing the page when it has been allocated by the slab allocator. It is also used to link blocks of free pages together.

**mapping** When files or devices are memory mapped, their inode has an associated `address_space`. This field will point to this address space if the page belongs

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

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

**next_hash** Pages that are part of a file mapping are hashed on the inode and offset. This field links pages together that share the same hash bucket.

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

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

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

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

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

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

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

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

**Table 2.1.** Flags Describing Page Status

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

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

## 2.6 Mapping Pages to Zones

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

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

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

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

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

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

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

## 2.7 High Memory

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

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

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

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

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

## 2.8 What's New in 2.6

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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