The Linux Memory Manager

Lorenzo Stoakes

Contents

Preface

Linux is the most successful operating system of all time – running seamlessly on billions of devices worldwide including almost *all* of the internet's infrastructure, nearly all supercomputers and a dominant market share of mobile devices. Its success has been nothing short of astonishing.

Underlying all of this are core subsystems which provide the foundation on top of which the rest of the kernel is built such as the scheduler, the virtual file system, networking and arguably the most fundamental of all - the memory manager.

This book dives into the Linux kernel's memory management subsystem in substantial detail, describing its core algorithms, how memory is allocated and managed, how 'memory pressure' is handled (when the system runs low on memory), how it interacts with I/O via the page cache and much else.

This is not intended to be a practical how-to guide on diagnosing issues with memory, nor is it meant to be in any way exhaustive. No author, no matter their longevity, could dream of covering a subsystem in its entirety. It does however try to describe in as much detail as practically possible the *core* fundamentals of how things work, the concepts, data structures and algorithms used to manage memory.

Having said that, I have endeavoured to provide a considerable amount of practical information, with examples, references to procfs and sysfs data sources for diagnosis, a break down of dmesg output when the out of memory killer kicks in, knobs to twiddle to adjust memory management behaviour and much else which should hopefully render it a useful reference for practical use.

While the book is not exhaustive, I have tried hard to describe as much as I practically can with references to the ultimate authority on how things work – no, not Linus, the kernel source code. Where insufficient detail is provided you can always refer back to this, the ultimate reference.

The aim of this book is to help the reader understand how memory is managed in Linux in as much detail as possible and to provide a stepping-off point for further investigation. The aim of writing it was to do the same for me.

This book is aimed at both those simply curious about how this stuff works (in finest tradition of hacker culture) and Linux professionals wishing to gain a deeper understanding of how their operating system handles memory.

The fascination that drives my interest in this area is how simultaneously simple and incredibly complicated this part of the Linux kernel is. Something so seemingly straightforward hides beneath it a great deal of engineering effort replete with trade-offs and many, many moving parts. Much like mechanical watches I get a special thrill from knowing there is so much going on to provide something quite so fundamental.

1 Introduction

The Linux operating system (often referred to as the GNU/Linux operating system as it uses a great many components sourced from the GNU project) consists of a multitude of ordinary 'userland' programs which provide basic functionality, but ultimately what makes Linux, Linux is the *kernel*.

The kernel is the part of the operating system which is 'in charge' and in a sense is really the only thing your computer is running. It schedules programs, manages shared resources, handles errors, abstracts hardware and sits in the background as essentially the aether in which everything else resides.

One of the shared resources an operating system must manage is its memory. Modern systems have gigabytes of Random Access Memory (RAM) which must be shared between processes, drivers and internal kernel data structures. The memory management 'subsystem' (simply a subdivision of the kernel) is what does this, and what this book describes.

The book is based on the most recent of the kernel at the start of writing – **Linux 5.17** – and all code snippets and references are relative to this version. Additionally, while I try to keep things as architecture-independent as possible, in order to be able to be as specific as I can I will in some cases need to focus on one architecture in particular. For reasons of both ubiquity (on the desktop and server) and convenience, my focus will be the **x86-64** architecture.

2 Working with the kernel

Basics on getting the kernel source code, getting qemu setup, where mm files live, etc. Maybe too basic?

3 Fundamentals

3.1 Virtual memory

In the good old days of the 8-bit microcomputers programs ran one at a time and each one could access any part of the memory. Memory ranges were subdivided into ROM (Read-Only Memory), RAM (Random-Access Memory), video memory, device memory and various system-defined ranges (for example the Spectrum 128[1]) and each program simply had to know where they could and could not write to.

This poses problems – even if you run one program at a time a bug might result in you overwriting critical system state causing unexpected behaviour and most likely a crash, necessitating a system restart. If you run more than one program at once then you enter a world of pain – each program will need to somehow be able to determine what parts of memory it can and cannot access, while simultaneously being able to trample all over the data of both the system and any other program. A single bug and you kill the system.

What is termed *fragmentation* is also a huge problem in this scenario – every block of memory allocated by the operating system must sit in the 'gaps' left by all programs. If a program asks for more memory than a gap provides the request must be refused.

As memory is not mediated by the operating system but accessed directly by programs a solution to this issue must necessarily be implemented in hardware. The solution is *virtual memory* – a mechanism where the operating system is able to instruct a *Memory Management Unit (MMU)* to map 'virtual' addresses to 'physical' ones.

3.2 Page tables

Keeping a track of memory raises a practical issue – if we need metadata to keep track of things such as whether a certain part of memory is allocated or not, then we can't keep track of things at a byte granularity or we'd end up in the absurd situation where each byte of memory requires (at least) a byte to describe it.

We therefore have to divide memory up into blocks. These blocks are called *pages*.

4 Allocators

Covers basic design principles of an allocator with simple examples of a userland implementation and how this interacts with the operating system. ‘From malloc to mmap’.

5 Physical memory

Covers memory discovery (e820), zones, NUMA nodes, most importantly the buddy allocator, memory order sizes (maybe reference how OOM can occur because larger order sizes aren't available), `struct page`, `struct folio`, and whatever else is required to describe how the fundamental physical memory resource is allocated.

6 Virtual memory in the kernel

Covers the process address space, the physical memory mapping, page fault handling, the slab allocator, `kmalloc()` and `kfree()` and whatever else is required to describe virtual memory in the kernel.

7 Virtual memory in user space

Covers VMAs, CoW, fork, **mmap** and **brk** syscalls, and whatever else is required to describe virtual memory for user space processes.

8 cgroups and containers

Cover cgroups, the fundamental building block of containerisation as well as other means and approaches for constraining and controlling memory usage.

9 The memory programming interface

A practical how-to guide as to procfs interfaces, sysfs interfaces, `madvise()`, understanding the sysrq/oom dmesg output, perhaps eBPF, system knobs to twiddle – a general practical course as to how to work with the linux memory manager.

10 Shared memory

A detailed, practical and specific overview of how to share memory in linux as this is such a useful interface and therefore worthy of its own chapter.

11 A process autopsy

Work through a process actually starting, doing some things and exiting and examining what is happening under the covers.

12 Moving memory

Covers pinned pages vs. unpinned, why and how memory moves around. Make it clear that there's more going on here than you might expect.

13 Memory pressure and reclaim

Covers memory pressure, compaction, swapping, measuring memory pressure, handling memory pressure, how it fits with demand paging design decisions in Linux.

14 Memory-mapped I/O

General coverage of memory-mapped file I/O, how it works, trade-offs. Leads in neatly to the page cache.

15 The page cache

Describes the Linux kernel file system caching mechanism i.e. the page cache. Links in again with swapping.

16 Huge pages

Covers huge pages both via hugetlb, transparent huge pages, tmpfs and shmem.

17 The Out-Of-Memory killer

Covers how the OOM killer operates, how to change its behaviour, etc.

18 Device memory

Covers memory-mapped device registers, PCI, `ioremap`, and other topics relevant to driver development.

19 Early memory management

Covers memblock and how the kernel bootstraps into its memory manager + early memory that is discarded.

20 Userfaultfd

Covers userland page fault handling via **userfaultfd**.

References

[1] Sinclair Research. *ZX Spectrum +3 manual, chapter 8, part 24*. https://worldofspectrum.org/ZXSpectrum128+3Manual/chapter8pt24.html. [Online; accessed 7-March-2022]. 2022.

Index