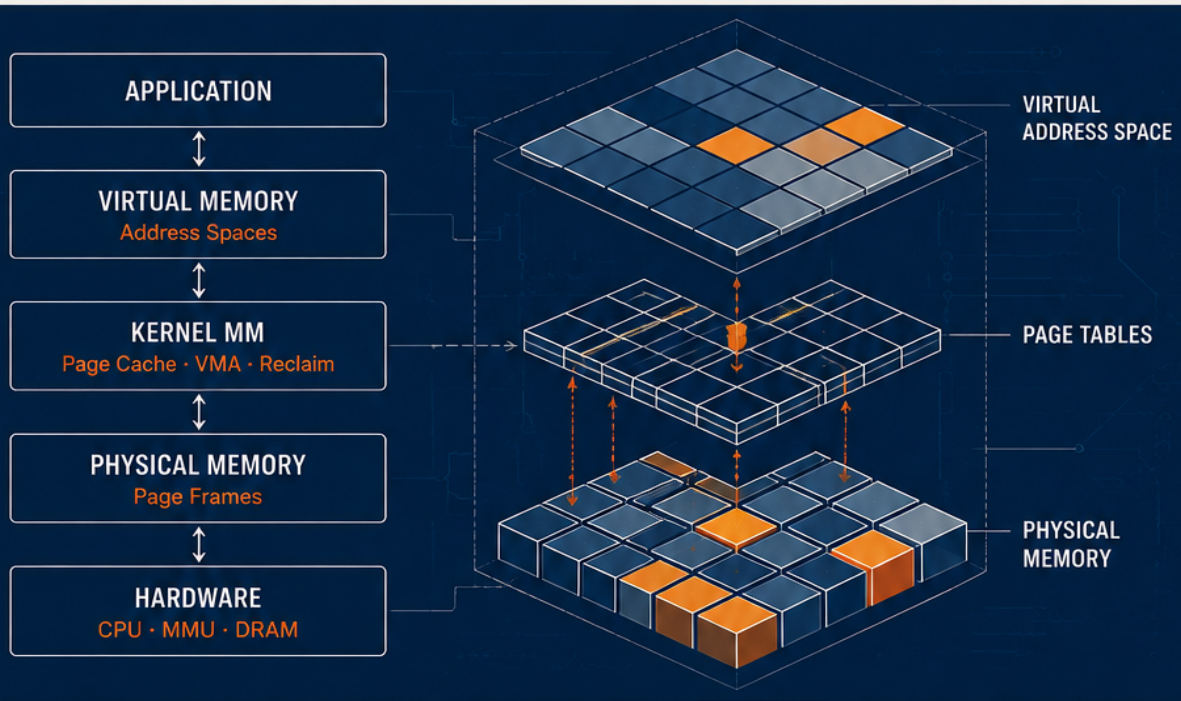
#LINUX MEMORY MANAGEMENT

FROM FUNDAMENTALS TO KERNEL INTERNALS

A Complete Guide to Virtual Memory, Allocators, Paging, and Performance in the Modern Linux Kernel

- VIRTUAL MEMORY
- PAGING & MMU
- KERNEL ALLOCATORS
- NUMA SYSTEMS
- CGROUPS & ISOLATION
- PERFORMANCE & DEBUGGING



- REAL KERNEL SOURCE EXAMPLES
- HANDS-ON EXERCISES
- PROFILE, DEBUG & OPTIMIZE
- HARDEN MEMORY IN PRODUCTION

DANIEL WHITAKER

Linux Memory Management: From Fundamentals to Kernel Internals

A Complete Guide to Virtual Memory, Allocators, Paging, and Performance in the Modern Linux Kernel

Steve Publications

This book is available at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinternals

This version was published on 2026-08-01

Contents

A Complete Guide to Virtual Memory, Allocators, Paging, and Performance in the Modern Linux Kernel*

This book takes you on a systematic journey through every layer of Linux memory management. You will learn how virtual memory works from silicon up to application code, how the kernel allocates and reclaims physical pages, how allocators optimize for speed and fragmentation, how NUMA systems distribute memory across nodes, how containers isolate resources with cgroups, and how to profile, debug, and harden memory behavior in production. Every concept is explained with clear reasoning, concrete examples from real kernel source code, and hands-on exercises you can run on your own system. The target audience is systems programmers, kernel developers, performance engineers, and advanced software practitioners who need deep understanding of how Linux manages memory.

Introduction

A server is running a database cluster under heavy load when suddenly one node becomes unresponsive. The logs show nothing unusual until you check dmesg and find the unmistakable message: Out of memory: Kill process 12345 (postgres) score 987 or sacrifice child. Another instance, a web application serving thousands of requests per second, exhibits intermittent latency spikes that correlate with pages being swapped in and out. A third case involves a real-time system where high-order memory allocations occasionally fail, triggering cascading errors through the driver stack.

These are not hypothetical scenarios. They are everyday problems for engineers working with Linux systems, and all of them trace back to memory management decisions made deep inside the kernel. Understanding those decisions is not optional if you want to build reliable, performant software on Linux.

This book exists because memory management is one of the most complex subsystems in any operating system, and Linux is no exception. The kernel must juggle competing demands: providing each process with its own isolated address space while sharing physical RAM efficiently, allocating memory fast enough not to become a bottleneck, keeping frequently used data close at hand, gracefully degrading when pressure mounts, and doing all of this across dozens or hundreds of CPUs without locking up the system. It must also cope with hardware quirks, device constraints, security requirements, and workloads that range from embedded sensors to hyperscale database clusters.

The promise of this book is straightforward. By the end, you will understand how Linux memory management works from the ground up, why it works that way, and how to apply that knowledge in practice. You will know what happens when a process calls malloc, how the kernel decides which pages to evict, why your allocation sometimes blocks, what causes TLB shootdowns and how they hurt performance, how NUMA affects memory placement, how cgroups enforce limits, and how to diagnose memory problems using the tools available in modern kernels.

The central thesis running through these chapters is that Linux memory management is not a collection of disconnected mechanisms but a coherent,

layered system whose design decisions emerge from fundamental constraints of hardware, performance, and scalability. Virtual memory exists because programs cannot safely share physical addresses directly and need address spaces larger than RAM. Paging and page tables exist because mapping every virtual address eagerly would be wasteful and slow. Allocators sit on top of pages because applications rarely need exactly one page at a time. Reclaim algorithms exist because memory is finite. Huge pages exist because TLB misses are expensive. Each layer solves a specific problem while introducing new ones that the next layer must handle.

The book is organized to build understanding incrementally. We begin with foundational concepts: what virtual memory is, why it matters, and how hardware supports it. We then move into address translation, page tables, and how Linux organizes virtual address space for processes and the kernel. Physical memory management follows, with detailed treatment of the buddy allocator, zones, NUMA nodes, and GFP flags. Kernel allocators come next: slab, slub, and kmalloc, along with vmalloc for large virtually contiguous regions. User-space allocation strategies are covered alongside kernel mechanisms because they interact closely through system calls.

The middle chapters address demand paging, page faults, copy-on-write, the page cache, and swapping. These are where theory meets practice most directly: every file read, every fork, every mapped region depends on these mechanisms. We then cover reclaim algorithms, compaction, huge pages, and NUMA-aware allocation in depth. The later chapters deal with memory cgroups and container isolation, out-of-memory handling, synchronization and concurrency in the memory subsystem, performance profiling and debugging tools, security implications and mitigations, and architecture-specific implementations across x86-64, ARM64, and RISC-V.

Each chapter is self-contained enough to be read independently but contributes to a cumulative picture. Code examples are complete and tested against recent kernels using standard toolchains. Where kernel source is discussed, file paths and function names correspond to the 6.x series, with notes about differences in earlier versions where relevant. You will find practical experiments throughout: commands to inspect your system's memory state, scripts to measure allocator behavior, and procedures for building custom kernels and running them under QEMU for safe experimentation.

A few conventions are worth noting. Inline citations appear as bracketed numbers like [1] and reference the bibliography at the end of the book. Kernel

source paths use the convention mm/page_alloc.c rather than full repository URLs. Architecture-specific terminology follows official documentation: x86-64 for Intel and AMD 64-bit processors, ARM64 or AArch64 for ARM, and RISC-V for that architecture family. When discussing performance, numbers are presented with context about the test environment because absolute values depend heavily on hardware and configuration.

The book assumes you are comfortable with C programming, basic operating system concepts, and using the command line. It does not assume prior kernel development experience. If you have debugged a memory leak, tuned swap settings, or wondered why your process got killed by the OOM killer, this book will give you the background to understand what was happening and how to prevent it next time.

Let us begin at the beginning with the fundamental problem that virtual memory was designed to solve.

Chapter 1: Foundations of Memory Abstraction

The Memory Problem: Why Programs Cannot Share Physical Memory Directly

Imagine a computer system where every program runs directly on physical memory addresses. Program A starts at address 0x00000000 and occupies the first megabyte. Program B must start at 0x10000000 or higher to avoid colliding with A. If the operating system wants to run both simultaneously, it needs to know in advance how much memory each one requires and load them into non-overlapping regions. This approach, called static memory allocation, was used in early batch-processing systems and works fine when you have a small number of programs whose sizes are known beforehand.

The moment you introduce dynamic behavior, the problems multiply. What if Program A grows at runtime by allocating more data? It may overwrite Program B. What if Program A finishes and the system wants to load Program C into its freed space, but Program C is larger than A was originally estimated to be? Now the system must either compact all loaded programs together, leaving one large free block, or accept fragmentation where memory is available in pieces too small for any waiting program.

Worse still, there is no isolation. If Program A contains a bug that writes past the end of an array, it can corrupt Program B's data or even overwrite operating system code. Malicious programs gain immediate access to everything in memory: passwords, cryptographic keys, other users' files. There is no protection.

Virtual memory was invented to solve these problems. The core idea is simple: each program sees its own address space, entirely separate from every other program and from the kernel. Program A can start at address 0x00000000 in its view of the world, and so can Program B, because those addresses are virtual. The hardware and operating system translate virtual ad-

dresses into physical addresses behind the scenes, ensuring that each process accesses only the memory allocated to it.

This abstraction delivers four critical benefits:

1. **Isolation**: A bug or attack in one process cannot directly corrupt another process's memory. The MMU enforces protection bits on every page, and violations trigger exceptions that the kernel can handle.
2. **Flexibility**: Programs no longer need to know their physical location. They use virtual addresses consistently, and the operating system decides where their pages actually live in RAM. Relocation becomes automatic rather than a burden placed on programmers or linkers.
3. **Overcommitment**: The total virtual address space across all processes can exceed physical RAM. Not every page is used at once, and some pages can be backed by disk storage rather than resident memory. This allows systems to run more work than would fit in RAM if everything were loaded eagerly.
4. **Simplified memory management**: Instead of tracking arbitrary-sized blocks of memory, the kernel manages fixed-size pages. Allocation, deallocation, and relocation all become operations on uniform units rather than irregular chunks.

The first production system with true virtual memory was the Ferranti Atlas at the University of Manchester in 1962 [1]. It introduced demand paging, where pages are loaded from secondary storage only when actually accessed rather than all at once. This concept became standard across Unix systems and eventually landed in Intel processors with the 80386 in 1985, which is the processor Linux was originally developed on.

Virtual Memory as an Abstraction Layer

Virtual memory is fundamentally a translation layer between what software sees and what hardware provides. Every memory access by a process uses a virtual address. The memory management unit, or MMU, translates that virtual address into a physical address before the actual read or write occurs. If the translation fails because the page is not in RAM, the MMU raises an exception and the kernel handles it by loading the page from disk or allocating a new one.

This abstraction has profound consequences. Consider what happens when you call malloc(4096) in a C program. The process does not immediately consume four kilobytes of physical RAM. Instead, the kernel extends the process's heap region in its virtual address space and marks those pages as allocated but not yet populated with data. Physical memory is only consumed when the program actually writes to those pages, triggering a page fault that causes the kernel to allocate a real page and map it into the virtual address.

This lazy allocation strategy, called demand paging, is one of the most important performance optimizations in modern operating systems. Programs frequently reserve large virtual address spaces but use only a fraction of them. A web server might allocate gigabytes for connection buffers but never fill all of them simultaneously. Loading every page eagerly would waste RAM and slow down startup dramatically.

The abstraction also enables features that would be impossible with direct physical addressing. Copy-on-write fork, where a child process initially shares all pages with its parent and only duplicates them when either side writes, depends entirely on virtual memory. Without it, every fork call would need to copy the entire address space, making process creation prohibitively expensive.

Shared libraries work similarly. Multiple processes can map the same physical pages containing libc or other shared code into their own virtual address spaces at different virtual addresses. Each process sees its own copy of the library in its own address range, but only one set of physical pages is needed.

Processes, Address Spaces, and Isolation

In Linux, each process has its own address space described by a structure called mm_struct. This structure contains the page directory root that defines how virtual addresses map to physical pages, a list of virtual memory areas describing contiguous regions with specific attributes, and various accounting fields tracking memory usage.

A virtual memory area, or VMA, is a contiguous range of virtual addresses that share the same protection and behavior. A typical process has several VMAs: one for executable code loaded from the binary, one for read-only data like string constants, one for writable global variables, one for the heap

that grows upward as malloc allocates memory, one for the stack that grows downward, and potentially many more for mapped files or shared memory regions.

When a process is created via fork, Linux creates a new mm_struct and duplicates the parent's VMAs. However, none of the physical pages are copied yet. Instead, every page table entry is marked read-only in both parent and child. If either process tries to write to a shared page, the MMU raises a page fault. The kernel's page fault handler detects this as a copy-on-write event, allocates a new page, copies the data, and maps the new page into the writer's address space with write permission. The other process continues using the original page.

This mechanism makes fork extremely fast regardless of process size. A process with gigabytes of memory can be forked in microseconds because no data is actually copied. Only the metadata describing the address space is duplicated, and physical pages are shared until someone modifies them.

Isolation between processes is enforced by the hardware. Each process's page tables define exactly which virtual addresses are valid and what operations are permitted on each page. If a process attempts to access an unmapped address, the MMU raises a page fault that the kernel translates into a segmentation fault signal. If a process tries to write to a read-only page, it gets a similar fault. If a process somehow obtains another process's virtual address and tries to use it, the translation will fail because that virtual address is not mapped in its own page tables.

The kernel maintains strict separation between user-space and kernel-space addresses. On x86-64, the upper half of the 48-bit virtual address space is reserved for kernel use. User processes cannot access those addresses because they are not mapped in their page tables, and the CPU's protection mechanisms prevent accidental or malicious access. This separation means a buggy user program cannot crash the kernel by dereferencing a bad pointer, and it prevents unprivileged code from reading sensitive kernel data structures.

The Role of the Operating System in Memory Management

The MMU provides the hardware mechanism for translation and protection, but the operating system is responsible for managing the mapping. Linux's

memory management subsystem handles several distinct responsibilities:

Address space management: Creating and destroying page tables when processes are born and die, setting up VMAs for code, data, heap, stack, and mapped regions, and maintaining the data structures that describe each process's virtual memory layout.

Physical memory allocation: Deciding which physical pages to assign to which virtual addresses, tracking which pages are free and which are in use, and ensuring that allocations respect hardware constraints like DMA requirements.

Page replacement: When physical memory is full and a new page is needed, deciding which existing page to evict. This decision balances recency of access, whether the page is backed by a file that can be reloaded, and whether the page has been modified and needs to be written back first.

I/O coordination: Coordinating with filesystems and block devices to load pages from disk on demand and flush dirty pages back to storage. This includes readahead strategies that anticipate future accesses and prefetch data proactively.

Memory accounting: Tracking how much memory each process, cgroup, or subsystem is using for monitoring, enforcement of limits, and debugging.

These responsibilities are implemented across dozens of source files in the kernel's mm directory. The code is complex because it must handle every edge case: allocation failures under extreme pressure, race conditions between concurrent accesses, NUMA-aware placement on multi-socket systems, huge pages for performance-critical workloads, and security requirements that prevent information leaks through page reuse.

Hardware Support: MMUs and Protection

The memory management unit is the hardware component that makes virtual memory possible. It sits between the CPU core and the memory controller, intercepting every memory access and performing translation before the actual read or write.

Translation works by consulting page tables, which are data structures in RAM that map virtual addresses to physical addresses. A typical x86-64 system uses four levels of page tables, each level indexed by a portion of the virtual address. The CPU walks through these levels, following pointers from one table

to the next, until it reaches a final entry that contains the physical page frame number.

To avoid walking page tables on every access, CPUs include a translation lookaside buffer, or TLB, which caches recent translations. When the MMU needs to translate an address, it first checks the TLB. If the translation is cached, the access proceeds immediately. If not, a TLB miss occurs, the CPU walks the page tables in RAM, loads the translation into the TLB, and then retries the original access.

The MMU also enforces protection bits on every page. Each page table entry contains flags specifying whether the page is present in RAM, whether it can be read, written, or executed, and whether it is accessible from user mode or only from kernel mode. If a process violates any of these constraints, the MMU raises an exception and transfers control to the kernel's fault handler.

Modern MMUs support additional features that Linux exploits. Huge pages allow mapping 2 megabyte or even 1 gigabyte regions with a single page table entry, reducing TLB pressure for large memory workloads. Write-protect bits enable copy-on-write semantics. Access and dirty bits track whether a page has been read or written since the last check, supporting page replacement algorithms. Address space identifiers allow the TLB to distinguish entries from different processes, avoiding expensive flushes on context switches.

The Linux Memory Management Philosophy

Linux memory management embodies several design principles that have guided its evolution over decades:

Simplicity where possible, complexity where necessary: The basic model is straightforward: pages are the unit of management, page tables map virtual to physical, and faults trigger allocation. Complexity is introduced only when it solves a real problem. Huge pages exist because TLB misses hurt performance on large workloads. NUMA-aware allocation exists because memory latency matters on multi-socket servers. Each complication has a measurable justification.

Aggressive caching: Linux caches aggressively. File data is cached in memory after the first read, executable code is shared across processes, and anonymous pages are kept as long as possible before being swapped out. The

assumption is that RAM should be used productively rather than left idle, and that reclaiming memory when needed is cheaper than allocating it eagerly.

Graceful degradation: When memory runs low, Linux does not immediately panic. It reclaims pages from caches, evicts cold data, swaps anonymous pages to disk, and compacts fragmented regions to make room for large allocations. Only when all of these mechanisms fail does the OOM killer terminate processes. This layered response allows systems to survive temporary spikes in demand without catastrophic failure.

Concurrency and scalability: Memory management must work efficiently on systems with hundreds of CPUs. Lock contention is minimized through per-CPU caches, fine-grained locking, and lockless read paths using RCU. NUMA-aware allocation ensures that each CPU preferentially uses local memory, reducing cross-node traffic.

Pragmatism over purity: Linux memory management includes many features that are not theoretically elegant but work well in practice. The buddy allocator has known fragmentation problems but is simple and fast. The page replacement policy is an approximation of LRU rather than optimal, but it performs well on typical workloads. These choices reflect a bias toward practical effectiveness.

Understanding these principles helps explain why Linux memory management looks the way it does. Every data structure, every algorithm, every configuration knob exists to serve one or more of these goals. As we dive into the details in subsequent chapters, keep these principles in mind as context for the design decisions you encounter.

In the next chapter, we will examine the hardware mechanisms that make all of this possible: how MMUs translate addresses, how page tables are structured, and how different architectures implement virtual memory. This foundation is essential for understanding everything that follows.

Chapter 2: Hardware Foundations and Address Translation

Memory-Mapped I/O and Physical Addressing

The Memory Management Unit Architecture

Page Tables: Structure, Levels, and Walks

Translation Lookaside Buffers and TLB Shootdowns

x86-64 Paging: Four-Level and Five-Level Page Tables

ARM64 and RISC-V Virtual Memory Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Memory Types, Caching Attributes, and Access Permissions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Walking a Page Table by Hand: A Practical Example

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Chapter 3: Linux Address Spaces and the Virtual Memory Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

The Process Address Space in Linux

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

User-Space Layout: Code, Data, Heap, Stack, and Mapped Regions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Kernel Virtual Address Space and Direct Mapping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Architecture-Specific Layouts: x86-64, ARM64, and RISC-V

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Address Space Layout Randomization and Security

The mm_struct and vma_struct: Core Data Structures

Memory Region Management with mmap, brk, and mremap

Practical Experiment: Inspecting Your Process's Memory Map

Chapter 4: Physical Memory Management and the Buddy Allocator

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Tracking Physical Pages: The page Struct and Memmap

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Memory Zones and Node Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

The Buddy Allocator Algorithm and Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Free Page Lists and Order Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Allocation Paths: GFP Flags and Context Sensitivity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Fragmentation Analysis and Mitigation Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

NUMA-Aware Allocation and Policy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Source Walkthrough: mm/page_alloc.c

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Chapter 5: Kernel Memory Allocators: Slab, Slub, and SLOB

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Why Kernel Needs Specialized Allocators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

The Slab Allocator Design and History

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Slub: Simplification and Performance Improvements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

SLOB: Minimalist Allocation for Embedded Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Per-CPU Caches and Lockless Fast Paths

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

kmalloc, kcalloc, and the Kernel Allocation API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Cache Colocation, False Sharing, and NUMA Effects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Building a Custom Kernel Allocator Module

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Chapter 6: User-Space Memory Allocation and vmalloc

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

The malloc Contract and POSIX Requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

glibc ptmalloc: Arenas, Bins, and Thread Safety

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Alternative Allocators: jemalloc and tcmalloc

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

mmap vs brk: Strategies for Large and Small Allocations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

vmalloc and the Kernel's Virtual Memory Allocator

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

vmap, ioremap, and Special Mapping Regions

Performance Comparison: Benchmarking User-Space Allocators

Practical Experiment: Tracing malloc Behavior with eBPF

Chapter 7: Demand Paging, Page Faults, and Copy-on-Write

The Lazy Loading Philosophy: Demand Paging

Page Fault Types and Handling Flow

Major and Minor Faults: Performance Implications

File Backed Pages and the Page Cache Introduction

Copy-on-Write Semantics and Implementation

Fork, Exec, and Memory Efficiency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Shared Memory Regions and mmap SHARED vs PRIVATE

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Source Walkthrough: The Page Fault Handler in mm/memory.c

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Chapter 8: The Page Cache and Buffer Head Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Why Caching File Data in RAM Transforms Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Page Cache Data Structures: Radix Trees and XArrays

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Page State Machine and Lifecycle Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Readahead Algorithms and Prefetching Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Dirty Pages, Writeback, and Flush Daemons

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Buffer Heads, Extents, and Historical Evolution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Interaction with Filesystems and Block Devices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Practical Experiment: Measuring Page Cache Hit Rates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Chapter 9: Swapping, Reclaim, and Page Replacement Policies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

When RAM Runs Out: The Reclaim Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Active and Inactive Page Lists

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Page Replacement Policies and LRU Approximation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Swap Spaces, Swap Devices, and File-Based Swapping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

The Swap-In and Swap-Out Pathways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

zswap, z3fold, and Compressed Caching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Thrashing Detection and Prevention

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Practical Experiment: Tuning vm.swappiness and Observing Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Chapter 10: Memory Compaction, Defragmentation, and Huge Pages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Fragmentation in Long-Running Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

The Compaction Algorithm: Migration and Scanning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Direct Reclaim vs Kswapd vs Compact Daemons

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Huge Pages and TLB Efficiency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Transparent Huge Pages: Automatic Promotion and Demotion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

hugetlbfs and Preallocated Huge Page Pools

Performance Impact: Benchmarks and Trade-offs

Practical Experiment: Enabling and Measuring THP Effects

Chapter 11: NUMA, Memory Policy, and Scalability

The NUMA Problem: Distance Matters

Node, Zone, and Memory Hierarchy in Linux

Memory Policies: Default, Interleave, Bind, and Prefer

Task Migration and Memory Placement Coordination

Per-NUMA Node Allocators and Caches

Scalability Challenges on Large Systems

Performance Measurement and NUMA-Aware Programming

Chapter 12: Memory Cgroups, Controllers, and Resource Limits

Containerization and the Need for Memory Isolation

Cgroup Hierarchy and Memory Controller Architecture

Memory Limits, Soft Limits, and Thresholds

Accounting: Tracking Usage Per-Cgroup

Reclaim Under Constraints: Hierarchical Pressure

Swap Accounting and Memory+Swap Limits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

OOM Behavior Within Cgroups

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Practical Experiment: Setting Up Memory-Limited Containers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Chapter 13: Out-of-Memory Handling, Panic, and Recovery

When Reclaim Fails: The Last Resort

The OOM Killer Algorithm and Victim Selection

Scoring Factors and Heuristics

OOM Score Adjustment and Task Protection

System-wide vs Cgroup-local OOM Events

Kernel Panic Conditions and Memory Exhaustion Scenarios

Recovery Strategies and Production Best Practices

Real-World Case Studies: OOM Incidents and Post-Mortems

Chapter 14: Synchronization, Concurrency, and Locking in Memory Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Concurrency Challenges in a Shared Allocator

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Locking Hierarchy and Deadlock Prevention

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Page Table Locks and RCU Protection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Zone Locks, PGDAT Locks, and Fine-Grained Contention Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Per-CPU Data Structures and Lockless Fast Paths

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Reference Counting and RCU in Page Lifecycle Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Memory Barriers, Ordering, and Cache Coherency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Scalability Analysis: Contention on Large Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Chapter 15: Performance Optimization, Profiling, and Debugging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Memory Performance Metrics That Matter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Using /proc and /sys for Runtime Inspection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Perf Tools: Flame Graphs and Memory Profiling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

eBPF and BCC: Custom Tracing Instruments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

ftrace and Kernel Function Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Kernel Module Development for Memory Experiments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Crash Dump Analysis with crash and kdump

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Practical Experiment: Building a Complete Debugging Workflow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Chapter 16: Security Implications, Exploits, and Mitigations

Memory Corruption and Kernel Exploitation

Use-After-Free and Slab-Based Attacks

Information Leaks Through Page Reuse

ASLR, KASLR, and Address Randomization

Stack Protection and Canary Values

SMAP, SMEP, and Supervisor Mode Protections

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

MDS, Spectre, Meltdown, and TLB Side Channels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Kernel Hardening: CONFIG Options and Best Practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Chapter 17: Architecture-Specific Implementations and Source Walkthroughs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Architecture Abstraction Layers in the Kernel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

x86-64 Implementation Details and Optimizations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

ARM64 Memory Management and Attribute Indirection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

RISC-V Sv39, Sv48, and Sv57 Paging Modes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

TLB Shootdown Differences Across Architectures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

IOMMU and DMA Mapping Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Navigating the Kernel Source Tree: A Practical Guide

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Building and Booting a Custom Kernel for Experimentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte

Conclusion: Synthesis and Future Directions

Key Design Principles

Emerging Trends

Final Perspective

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/linuxmemorymanagementfromfundamentalstokernelinte