MASTERING C FOR KERNEL DEVELOPMENT

JAC HERMOCILLA

OPERATING SYSTEMS

Mastering C for Kernel Development

JAC Hermocilla

2026-07-12 · 37d86e6

Contents

License

A Note on AI-Assisted Production

This book was produced with the assistance of artificial intelligence tools. AI systems were used to help draft, rewrite, edit, and refine the prose, to improve the clarity and consistency of explanations, and to assist with organizing and formatting the manuscript.

All content was reviewed by a human author, who is responsible for the final text, its technical accuracy, and its pedagogical choices. AI tools were used as an aid to writing and editing, not as a replacement for authorship or review.

Because both AI systems and human authors can make mistakes, some errors may remain. Readers are encouraged to verify technical details against the accompanying source code and authoritative references such as the processor manuals, and to report any errors they find so that future revisions can correct them.

Reviewers

This book is better for the generosity of the reviewers who read the manuscript with care, tested its explanations against the source code, and told me plainly where it fell short. Their questions sharpened the prose, caught mistakes I could no longer see, and kept the book honest to the kernel it describes. I am grateful to each of them.

- **Arian J. Jacildo**, *Assistant Professor, UPLB* — a teacher of CMSC 123 (Data Structures), whose feel for how data is organized in memory sharpened the chapters on structures, page tables, and the heap.
- **Fermin Roberto G. Lapitan**, *Assistant Professor, UPLB* — who teaches CMSC 21 (C Programming), the course this book takes as its starting point, and helped keep its C precise and idiomatic.
- **Rozano S. Maniaol**, *IT Consultant* — a mobile-game developer who read the manuscript with a working programmer's eye for performance and the details of low-level code.
- **Rizza DC. Mercado**, *Assistant Professor, UPLB* — whose CMSC 131 (Assembly Language Programming) course made her an ideal reader for the chapters where C meets the machine, from inline assembly to the calling convention.

Any errors that remain are mine alone.

Introduction

Why This Book Exists

Learning the C programming language is relatively easy. Learning to use C to build an operating system is not.

Most introductory C books teach the language as though every program begins with `main()`, prints text using `printf()`, allocates memory with `malloc()`, and runs under Linux or Windows. These books assume an operating system already exists, and they teach C from the perspective of an application programmer.

Kernel development is different. There is no operating system, no standard library, no debugger waiting in the background, and no process manager, file system, or memory allocator. The software you are writing must create those services itself. This changes the way C is used: the language is the same, but the environment is completely different.

This book was written to bridge that gap. It is not another introduction to C, nor is it a textbook on operating system theory. Instead, it is a companion for students who want to understand the C code found in the 64-bit operating system tutorial. Every chapter explains the language features, programming idioms, design patterns, and engineering decisions that appear in the repository. The emphasis is always practical, and whenever possible, concepts are connected directly to code that you will read, modify, and execute.

By the end of this book, opening an unfamiliar kernel source file should no longer feel intimidating. You should be able to understand not only what the code does, but also why it was written that way.

Who This Book Is For

This book is intended primarily for my undergraduate students in `CMSC 125`. But the book can be used by students in

- Computer Science,
- Computer Engineering,
- Software Engineering,

- Computer Systems,
- Operating Systems,
- Embedded Systems,
- Computer Architecture.

It assumes that you already know some of the fundamentals of the C language (taught in `CMSC 21` and `CMSC 123`). Specifically, you should already understand

- variables,
- arithmetic expressions,
- loops,
- conditional statements,
- functions,
- arrays,
- basic pointers.

Check the Appendix for a refresher.

You do **not** need prior experience writing operating systems, and you do **not** need to know assembly language in advance. Whenever assembly appears (taught in `CMSC 131`), it is introduced only to explain how it interacts with C. Likewise, advanced processor concepts (taught in `CMSC 132`) are introduced only when they become necessary for understanding the code. This book teaches enough architecture to make the C code meaningful.

What This Book Is Not

This book is not a complete reference manual for the C language. Many excellent books already serve that purpose, and it does not attempt to describe every feature of ISO C. Instead, it focuses on the subset of the language that matters most for systems programming. You will not find long discussions of

- console applications,
- graphical interfaces,
- file processing,
- standard input,
- desktop programming,
- application frameworks.

Those topics belong to application development. Kernel programming presents different challenges, and this book focuses on those challenges.

Similarly, this is not a complete operating systems textbook. Topics such as scheduling theory, synchronization algorithms, virtual-memory policies, filesystems, networking protocols, and security models deserve books of their own. Whenever these topics appear, they are introduced only to explain the accompanying C code. The emphasis remains on understanding implementation rather than surveying theory.

Learning C in Context

One of the weaknesses of many programming books is that language features are introduced without context. Students learn pointers, then structures, then arrays, then function pointers, and only much later do they encounter realistic software. The result is that the language feels like a collection of disconnected features.

This book takes a different approach: every language feature appears because the operating system needs it. Pointers become meaningful when accessing memory. Structures become meaningful when describing processor registers. Function pointers become meaningful when dispatching interrupt handlers. Bit manipulation becomes meaningful when constructing page-table entries. Inline assembly becomes meaningful when communicating with hardware. The operating system provides the motivation, the language provides the implementation, and each reinforces the other.

Why Operating Systems Are Excellent Teachers

An operating system exercises nearly every important feature of the C language: pointers, structures, memory management, bit operations, arrays, function pointers, inline assembly, separate compilation, linking, macros, and debugging.

Unlike many application programs, kernels cannot hide behind libraries. Every abstraction must eventually be implemented, every memory allocation must be understood, and every byte has a purpose. This makes operating system code an excellent environment for learning careful programming. The software is demanding, but it is also remarkably honest, because nothing important is hidden.

A Different Way of Reading Code

Many beginning programmers read source code one line at a time, but professional programmers rarely do. They begin by asking larger questions. What problem does this module solve? What interface does it expose? Which data structures does it define? What assumptions does it make? How does execution reach this function, and how does control leave it?

These questions create a mental model, and individual statements make sense only after the larger design becomes clear. Throughout this book you will learn to read code this way, because understanding architecture before implementation is one of the defining habits of experienced systems programmers.

Learning Through Experimentation

Reading source code is necessary, but it is not sufficient. Every chapter therefore encourages experimentation. Compile the kernel, boot it, and observe its behavior; change

one line, compile again, and observe what changed. Predict outcomes before running the code, test your predictions, and keep notes.

The computer is not merely a machine that executes programs. It is an experimental laboratory. Programming is an empirical discipline, and understanding grows through observation as much as through reading.

How to Use This Book

Each chapter follows a consistent structure. First, a programming concept is introduced. Next, the concept is connected to examples from the operating system repository. The discussion then explains why the implementation is written that way rather than some other way. Finally, the chapter concludes with practical exercises designed to encourage experimentation rather than memorization.

You should resist the temptation to read the entire book without touching the code. Instead, work alongside the repository: compile every chapter, run every kernel, modify every subsystem, and observe the results. This active approach requires more time, but it also produces deeper understanding.

The Philosophy of This Book

This book follows a simple philosophy: clarity before cleverness, understanding before memorization, experimentation before optimization, and small steps before large leaps.

Systems programming often appears intimidating because many ideas arrive simultaneously — processors, memory, interrupts, assembly language, linkers, debuggers, and build systems. Taken together, they seem overwhelming; taken one idea at a time, they become manageable. This book deliberately proceeds in small, carefully connected steps. Each chapter builds upon the previous one, and nothing is introduced without purpose.

Beyond This Book

Completing this book does not make you an operating system expert. It gives you something more valuable: a foundation. With that foundation you can continue to study

- the Linux kernel,
- BSD operating systems,
- embedded kernels,
- hypervisors,
- device drivers,
- filesystems,
- networking,
- virtualization,
- computer architecture.

More importantly, you will possess the confidence to open unfamiliar source code and begin exploring it systematically. That confidence is one of the defining characteristics of successful systems programmers.

A Final Word

The code in the accompanying tutorial was not written merely to produce a working operating system. It was written to teach. Read it carefully, question every design decision, and experiment freely. Expect mistakes. Break the kernel, then repair it. Measure your progress by what you understand rather than by how quickly you finish.

Learning operating systems is not a race. It is an apprenticeship. The chapters that follow are your guide, the repository is your laboratory, and the computer is your teacher.

Welcome to systems programming.

Chapter 1 - Types and Portability

Speaking the Language of the Processor

> *"Programs manipulate values. Operating systems manipulate representations."*

Learning Objectives

After completing this chapter, you should be able to

- explain why kernels avoid ordinary C integer types,
- understand how integer size depends on the machine architecture,
- distinguish between signed and unsigned arithmetic,
- understand why pointers become 64 bits in long mode,
- recognize the custom type definitions used throughout the codebase,
- explain why incorrect data sizes produce subtle kernel bugs,
- confidently read every type declaration in the repository.

1.1 Why This Is the First Chapter

When students begin learning C, they naturally focus on the visible parts of the language. They learn how to write loops, call functions, declare variables, and manipulate arrays. These are the features that allow programs to solve problems.

Operating systems introduce a different concern. Before asking *what value does this variable contain?*, we must ask *how many bits does this variable occupy?* This sounds like a small detail, but it is not.

Most application programs survive incorrect assumptions about integer sizes. An image editor rarely crashes because an integer occupies eight bytes instead of four, and a compiler rarely fails because a variable is signed instead of unsigned. An operating system has far less room for error. The processor expects page-table entries to occupy

exactly eight bytes, interrupt descriptors to occupy sixteen bytes, and control registers to contain sixty-four bits. Physical addresses are interpreted bit by bit according to the processor manual, so every structure in memory is a contract between software and hardware.

When the compiler stores something differently from what the processor expects, the machine does not attempt to recover. It simply executes the wrong instructions or accesses the wrong memory, and the result is usually a page fault, a general protection fault, or a triple fault.

This is why nearly every kernel begins by defining its own integer types — not because the language is inadequate, but because the hardware demands precision.

1.2 The Problem with Ordinary C Types

One of C's greatest strengths is portability. The same source code can compile on a microcontroller, a laptop, or a supercomputer with few changes. This portability exists because the C standard deliberately avoids fixing the sizes of most integer types. The language promises relationships such as

```
sizeof(char) <= sizeof(short)
sizeof(short) <= sizeof(int)
sizeof(int) <= sizeof(long)
```

Notice what it does **not** promise. It does not promise

```
sizeof(int) == 4
```

nor does it promise

```
sizeof(long) == 8
```

Those decisions belong to the implementation. For application programmers, this flexibility is usually invisible; for kernel programmers, it becomes impossible to ignore.

Suppose you are writing code that constructs a page-table entry. The processor expects exactly sixty-four bits — not approximately sixty-four bits, exactly. If your compiler chooses a different representation, the processor interprets the data differently, and the bug appears not because the compiler failed but because the compiler did exactly what the language allowed. The programmer simply asked the wrong question. Instead of asking for

> "an integer"

the programmer should have asked for

> "an unsigned sixty-four-bit integer."

Kernel code therefore prefers explicit sizes over generic names.

1.3 Defining Our Own Types

Open `common.h`. Near the beginning of the file you will find a familiar collection of definitions.

```
typedef unsigned long long u64int;
typedef          long long s64int;
typedef unsigned int       u32int;
typedef          int       s32int;
typedef unsigned short     u16int;
typedef          short     s16int;
typedef unsigned char      u8int;
typedef          char      s8int;
```

If you have never seen `typedef` before, the syntax may appear strange, but its purpose is simple: a `typedef` creates a new name for an existing type, and nothing more. The compiler does not generate different machine instructions, and the processor does not know the new names exist. These declarations exist entirely for human readers.

Instead of writing

```
unsigned long long physical_address;
```

we write

```
u64int physical_address;
```

The declaration becomes shorter, and more importantly, it communicates intent. Every reader immediately understands two things: the value is unsigned, and the value occupies sixty-four bits. Good code explains itself whenever possible, and type names are one of the easiest ways to accomplish this.

One detail rewards a second look. The signed types are written with the bare keyword — `char`, `short`, `int`, `long long` — not `signed char` and so on. For `short`, `int`, and `long long` this is merely shorthand, since those types are signed by default. Plain `char` is the exception: the standard leaves its signedness to the implementation, so `s8int` is reliably signed only because the compilers used here happen to make `char` signed. It is a small reminder of a theme we take up in §1.6 — that in kernel code, the exact representation of a type is never something to assume.

1.4 Why Not Use `<stdint.h>`?

Students familiar with modern C often ask why kernels define

```
u32int
```

instead of

```
uint32_t
```

from `<stdint.h>`. The answer depends on context. Modern kernels frequently use the standard definitions, while educational kernels often define their own. The original JamesM tutorial predates widespread adoption of C99 in hobby operating system projects, and defining the types locally also avoids depending on any hosted standard library during the earliest stages of bootstrapping. Later, after the kernel becomes more mature, replacing these definitions with the standard fixed-width types is straightforward.

The important idea is not the names. The important idea is that the sizes are explicit.

1.5 Understanding the LP64 Data Model

Moving to a sixty-four-bit processor introduces another misconception. Many beginners believe every variable suddenly doubles in size, but that is not how modern systems work.

Linux, BSD, and macOS follow the LP64 data model. Under LP64,

Type	Size
char	1 byte
short	2 bytes
int	4 bytes
long	8 bytes
pointer	8 bytes

Notice that `int` remains thirty-two bits. Only `long` and pointers become sixty-four bits.

This distinction explains many of the changes between the original 32-bit tutorial and the 64-bit version. Addresses become larger, pointers become larger, and page-table entries become larger, but many ordinary integers do not. Understanding this distinction prevents countless bugs later in the tutorial.

1.6 Signed and Unsigned Arithmetic

Choosing the correct integer size is only part of the story. Choosing whether that integer is signed or unsigned is equally important.

At first glance, the distinction appears straightforward. Signed integers can represent both positive and negative numbers, while unsigned integers represent only non-negative values. Since memory addresses, page numbers, and hardware registers are never negative, it seems obvious that kernels should always use unsigned types. In practice, the situation is more subtle.

The C language performs automatic type conversions whenever signed and unsigned values appear in the same expression. These conversions are well defined by the language standard, but they often surprise programmers. Consider the following code.

```
u32int length = 10;
int offset = -1;

if (offset < length)
    ...
```

Many students expect the comparison to be true because negative one is less than ten. The compiler sees something different. Because `length` is unsigned, C converts `offset` into an unsigned integer before making the comparison, and the value `-1` becomes the largest possible 32-bit unsigned integer. Instead of comparing

```
-1 < 10
```

the program compares

```
4294967295 < 10
```

so the result is false. Nothing is wrong with the compiler; it simply follows the language rules.

These implicit conversions are a frequent source of bugs in systems software, and they become particularly dangerous when calculating memory sizes or validating addresses. A single incorrect conversion can transform a small negative value into a very large positive one, causing the kernel to access memory far beyond its intended range.

As a general guideline, kernel code should avoid mixing signed and unsigned arithmetic unless there is a compelling reason to do so. Keep addresses, sizes, and hardware registers unsigned, and reserve signed integers for quantities that are naturally positive or negative, such as return values that indicate success or failure. Whenever the compiler warns about signed and unsigned comparisons, resist the temptation to ignore the warning, because the compiler is often pointing to a real problem.

1.7 Thinking in Hexadecimal

Soon after reading the source code, you will notice that many numbers begin with the prefix `0x`.

```
0x1000
0xB8000
0xFFFFFFFF80000000
```

These are hexadecimal constants. Programmers often learn hexadecimal notation as an alternative way of writing numbers, but kernel programmers think about it differently. Hexadecimal is the language of binary, because each hexadecimal digit represents exactly four bits.

```
Binary      Hexadecimal

0000            0
0001            1
```

```
0010            2
...
1111            F
```

This relationship makes hexadecimal especially convenient for describing addresses, bit masks, processor registers, and page-table entries. Suppose the processor documentation states that bit 7 enables interrupts. Representing this value in decimal obscures the underlying bit pattern,

```
128
```

whereas the hexadecimal representation is much more informative.

```
0x80
```

Experienced systems programmers can often recognize the bit layout immediately. The same principle applies to memory addresses. An address such as

```
0xB8000
```

is easier to relate to processor documentation than its decimal equivalent.

Throughout this tutorial, train yourself to read hexadecimal numbers as naturally as decimal numbers. They are not mysterious symbols; they are simply a more convenient representation for binary data.

1.8 Addresses Are Data

Most introductory programming courses teach that variables contain values. Kernel programming extends this idea: addresses are values too. A memory address is simply another number stored inside a variable.

Consider the declaration

```
u64int address = 0x100000;
```

Nothing about this variable tells the processor that the value represents memory. To the compiler, it is merely a sixty-four-bit integer. Only when the value is converted into a pointer does it acquire a new meaning.

```
u32int *ptr = (u32int *)address;
```

The bytes stored in `address` remain unchanged. The cast simply tells the compiler how future operations should interpret those bytes.

Learning to separate the numeric value of an address from the memory it refers to is an important milestone, because many programming errors arise when programmers confuse these two concepts. The address is a number, the pointer is an interpretation of that number, and the memory exists independently of both.

1.9 Integer Overflow

Computers perform arithmetic using a fixed number of bits, so eventually every counter reaches its maximum value. Suppose a variable contains

```
u8int value = 255;
```

and we add one.

```
value++;
```

The result is not 256. The result is zero. The arithmetic wraps around because an eight-bit unsigned integer cannot represent any larger value.

Unsigned overflow is well defined in C: it wraps modulo the size of the type. Signed overflow is different. The C standard treats signed overflow as undefined behavior, and the compiler is free to assume that it never happens, enabling aggressive optimizations that may produce unexpected results if the assumption is violated. For this reason, kernel programmers generally avoid relying on signed overflow. When wraparound behavior is required, unsigned arithmetic is usually the safer choice.

1.10 Reading Type Declarations

At first, C declarations appear intimidating. Consider the following example.

```
page_t *get_page(u64int address,
                 int make,
                 page_table_t *pml4);
```

Instead of trying to understand the entire declaration at once, read it one piece at a time. Begin with the function name,

```
get_page
```

which returns

```
page_t *
```

meaning a pointer to a page-table entry. The first parameter is

```
u64int address
```

which tells us that the function receives a sixty-four-bit virtual address. The second parameter determines whether missing page tables should be created, and the final parameter identifies the page-table hierarchy to search.

This systematic approach works for nearly every declaration in the kernel. Resist the urge to decode everything simultaneously, and instead read declarations from the inside out, one component at a time. With practice, even complex function prototypes become easy to understand.

1.11 Walking Through the Paging Code

Open `paging.c` and search for every occurrence of `u64int`. Do not begin by reading the implementation. Instead, study the declarations and ask yourself what each variable represents. Is it a virtual address, a physical address, a page-table entry, or a frame number?

Notice how the choice of type communicates the programmer's intent before you understand the surrounding code. Once you have formed a hypothesis, continue reading until the implementation confirms or contradicts your understanding.

This habit of making predictions before reading the code develops active rather than passive reading, and it is one of the most effective ways to become comfortable with unfamiliar systems software.

1.12 Common Mistakes

Nearly every beginner makes the same mistakes during the first few weeks of kernel development. They assume that `int` always occupies four bytes, they cast pointers to thirty-two-bit integers, they ignore compiler warnings about signed and unsigned comparisons, and they perform arithmetic on memory addresses without considering integer width.

Perhaps the most common mistake is treating hexadecimal numbers as unusual or difficult. In reality, hexadecimal notation is often easier to understand than decimal because it reflects the underlying binary representation directly.

Do not be discouraged by these mistakes. Every experienced systems programmer has made them, and what matters is learning to recognize them early.

1.13 Debugging Type-Related Bugs

Many kernel failures originate from incorrect assumptions about data representation. When debugging such problems, begin by asking simple questions. What is the size of this type? Should this value be signed or unsigned? Is this really a pointer, or is it simply an integer that represents an address?

Printing values in hexadecimal is often more informative than printing them in decimal. Memory addresses, processor registers, and page-table entries become much easier to interpret when viewed in the notation used throughout the processor manuals.

As the tutorial progresses, you will find yourself relying increasingly on careful reasoning rather than trial and error. The goal is not merely to fix bugs but to understand why they occurred.

Practice Exercises

Exercise 1

Open `common.h` and match each of its eight typedefs — `u8int` through `s64int` — to the built-in C type it renames. Then compile a small program that prints `sizeof` for `char`, `short`, `int`, `long`, and a pointer, and check the results against the LP64 table in §1.5. Which types did not grow when the code moved to sixty-four bits?

Exercise 2

Reproduce the comparison from §1.6 — `u32int length = 10;` and `int offset = -1;` — and predict the result of `offset < length` before you compile. Then build with `-Wall -Wextra`, and reconcile the compiler's warning and the actual result with the implicit-conversion rule the section describes.

Exercise 3

Write out the eight bits behind `0x80` and `0xFF`, naming which positions are set in each. Then store `0xFF` in a `u8int`, add one, and predict the result before running it, explaining the wraparound in terms of the bits you just drew.

Exercise 4

Open `paging.c` and find several declarations that use `u64int`. Judging from the name and type alone, decide whether each value is:

- a virtual address,
- a physical address,
- a page-table entry,
- or a frame number.

Then read enough of the surrounding code to confirm or correct each guess, and notice how much the type told you before the logic did.

A Brief Historical Note

The original versions of C were developed at a time when computers varied enormously in word size and architecture. Rather than forcing every implementation to adopt identical integer sizes, the language emphasized portability. This design decision contributed greatly to C's success, because the same language could run on minicomputers, workstations, embedded systems, and eventually personal computers.

Operating systems inherited this flexibility, but they also inherited the responsibility of managing it. Modern kernels therefore define explicit integer types that preserve portability while satisfying the processor's strict hardware requirements. Understanding this historical context explains why the language appears the way it does today. Many features that seem inconvenient were deliberate compromises made to keep C useful across many generations of computers.

Chapter Summary

This chapter introduced one of the central themes of kernel programming: representation matters. Unlike ordinary applications, an operating system cannot treat integers as abstract values. Every type communicates information about the underlying hardware. The width of an integer determines how much memory it can address, the choice between signed and unsigned arithmetic influences comparisons and overflow behavior, and even the notation used to write numbers reflects the way processors organize information internally.

As you continue reading the codebase, pay attention to every type declaration. Ask why the programmer chose `u64int` instead of `u32int`, or `u16int` instead of `int`. Those choices are rarely arbitrary; they are clues to how the hardware works.

In the next chapter, we build on this foundation by examining structures. Individual values are useful, but kernels rarely manipulate isolated numbers. Instead, they organize related values into carefully designed layouts that mirror the processor's own data structures, and understanding those layouts is the next step toward reading and writing kernel code with confidence.

Chapter 2 - Structures, Packing, and Hardware Interaction

Learning Objectives

After completing this chapter, you should be able to

- explain why structures are fundamental to kernel development,
- understand how a C structure represents a layout in memory,
- distinguish between logical organization and physical representation,
- explain compiler alignment and padding,
- understand the purpose of `__attribute__((packed))`,
- interpret hardware-defined structures such as the Interrupt Descriptor Table (IDT),
- explain why the order of fields in a structure is often critical, and
- read and modify kernel structures without introducing subtle memory layout errors.

2.1 Why Structures Matter

The previous chapter focused on individual values. We learned that integers have precise sizes because the processor expects precise representations, and that a thirty-two-bit value and a sixty-four-bit value are not interchangeable, even if both happen to store the same number.

Real operating systems rarely manipulate isolated values. Instead, they manipulate collections of related information. A process consists of many pieces of state: registers, a process identifier, scheduling information, open files, and a memory map. A page table contains hundreds of entries. An interrupt descriptor contains multiple fields that together describe how the processor should respond to an interrupt. Even a simple character displayed on the screen is represented by both the character itself and its display attributes. These collections are represented in C using structures.

At first glance, a structure appears to be nothing more than a convenient way to group related variables. That is certainly true in ordinary application programming, where a

structure describing a student or a bank account exists primarily to organize information in a logical way. Kernel programming introduces another purpose: a structure is often a direct representation of a hardware-defined memory layout.

This distinction changes the way we think about structures. Instead of asking, "What information belongs together?" we ask, "What sequence of bytes does the processor expect to find in memory?" That question guides the design of nearly every important structure in an operating system.

2.2 From Variables to Memory Layouts

Imagine describing an interrupt descriptor without using a structure. You might write

```
u16int base_lo;
u16int selector;
u8int  ist;
u8int  flags;
u16int base_mid;
u32int base_hi;
u32int reserved;
```

These variables clearly belong together, yet nothing in the language expresses that relationship. Every function would have to receive seven separate parameters, every array would require seven independent arrays, and the code would quickly become difficult to understand.

Structures solve this problem by treating related values as a single object.

```
typedef struct {
    u16int base_lo;
    u16int selector;
    u8int  ist;
    u8int  flags;
    u16int base_mid;
    u32int base_hi;
    u32int reserved;
} idt_entry_t;
```

Now the entire interrupt descriptor becomes one object.

```
idt_entry_t idt[256];
```

This declaration creates an array containing 256 interrupt descriptors. The compiler knows exactly where every field resides, and the programmer works with meaningful names instead of calculating byte offsets manually. Good abstractions hide unnecessary complexity, and structures are one of C's simplest and most effective abstractions.

2.3 A Structure Is a Blueprint

Many students think of a structure as a collection of variables. That description is correct, but incomplete. A better mental model is to think of a structure as a blueprint describing how memory should be organized.

Suppose we define

```
typedef struct {
    u32int student_id;
    u16int year;
    u16int age;
} student_t;
```

This definition does not allocate memory. It merely describes how memory should look whenever a `student_t` object is created. Later, when the program declares

```
student_t alice;
```

the compiler reserves enough memory for the entire structure, and each field occupies a fixed position relative to the beginning of the object.

```
Offset

0   student_id
4   year
6   age
```

When we write

```
alice.age = 20;
```

the compiler generates instructions that store the value at offset six. The programmer never needs to calculate addresses manually, because the compiler performs that work automatically.

This idea appears repeatedly throughout the kernel. Every process descriptor, page table, interrupt frame, and scheduler data structure is simply another blueprint describing memory.

2.4 The Compiler and Alignment

Computers access memory more efficiently when data begins at certain addresses. For example, a four-byte integer is usually read faster when it begins at an address divisible by four. To improve performance, compilers automatically align data.

Consider the following structure.

```
typedef struct {
    char c;
```

```
    int x;
} example_t;
```

Most students expect this structure to occupy five bytes: one byte for the character and four bytes for the integer. Most compilers instead produce a structure eight bytes long, because they insert three unused bytes between the fields.

```
Offset

0   c
1   padding
2   padding
3   padding
4   x
```

Those unused bytes are called **padding**. The compiler adds them because aligned accesses are generally faster than unaligned accesses. For ordinary software this optimization is beneficial, but for hardware-defined structures it becomes dangerous. The processor expects a specific layout, and hidden bytes change every subsequent offset, so the hardware begins reading incorrect information.

2.5 Packing Structures

The kernel therefore sometimes instructs the compiler to disable automatic padding. The GCC and Clang compilers provide the attribute

```
__attribute__((packed))
```

When applied to a structure,

```
typedef struct {
    ...
} __attribute__((packed)) idt_entry_t;
```

the compiler stores every field exactly as written — no hidden bytes, no automatic alignment, and no rearrangement. The resulting memory layout matches the programmer's specification exactly.

Whenever you encounter `packed` in kernel code, pause and ask why it is necessary. In most cases the answer is that the structure communicates directly with hardware.

2.6 The Interrupt Descriptor Table

The Interrupt Descriptor Table provides an excellent example. Every interrupt or exception causes the processor to index into this table, and each entry occupies exactly sixteen bytes in 64-bit mode. The Intel Software Developer's Manual defines the layout, and the kernel reproduces that layout in C.

```
typedef struct idt_entry_struct {
    u16int base_lo;
    u16int sel;
    u8int  ist;
    u8int  flags;
    u16int base_mid;
    u32int base_hi;
    u32int always0;
} __attribute__((packed)) idt_entry_t;
```

Notice that the interrupt handler's address does not appear as one sixty-four-bit field. Instead, it is divided into three separate pieces, because the processor expects this format and the kernel follows the specification exactly. The names of the fields mirror the processor documentation, making it easier to compare the source code with the architecture manual.

This is a recurring pattern throughout systems programming. Hardware manuals describe bytes, and kernel programmers translate those descriptions into structures.

2.7 Structures Created by Assembly

Not every structure is filled in by C code. Some originate in assembly language, and the interrupt subsystem provides an excellent example.

When an interrupt occurs, the processor automatically saves part of its state, and the assembly interrupt stub saves the remaining registers before calling the C interrupt handler. The C code receives a pointer.

```
void isr_handler(registers_t *regs)
```

The structure `registers_t` does not describe data created by C. It describes the stack frame produced by assembly instructions, so the order of the fields must match the order of the pushes exactly. If the assembly code executes

```
push rax
push rbx
push rcx
```

the structure must describe those registers in precisely the same order. Changing two adjacent fields causes the C code to interpret one register as another. The compiler sees nothing unusual, and the hardware faithfully executes incorrect code.

Understanding where a structure originates is often just as important as understanding its fields.

2.8 Structures and Arrays

Structures become even more powerful when combined with arrays. The Interrupt Descriptor Table is not one descriptor; it is an array of descriptors.

```
idt_entry_t idt[256];
```

Likewise, page tables contain arrays of page-table entries, process tables contain arrays of process descriptors, and file systems contain arrays of inodes. Instead of manipulating isolated objects, the kernel manages collections of objects that share an identical layout.

Arrays and structures therefore complement one another naturally. The structure defines the format, and the array defines the collection.

2.9 Reading Structures in the Codebase

As you continue reading the repository, develop the habit of studying structures before studying functions. Functions tell you how the kernel behaves; structures tell you what information the kernel considers important.

When you encounter a new structure, ask yourself several questions. What real-world object does it represent? Who creates it, and who modifies it? Does the processor ever read it directly? Must its layout match a hardware specification, or can the compiler safely insert padding?

These questions often reveal more than the implementation itself.

2.10 Common Mistakes

Beginning kernel programmers often assume that field order affects only readability. Hardware disagrees. Changing the order of fields changes their offsets, and changing offsets changes the meaning of every byte stored in memory.

Another common mistake is forgetting that compilers insert padding automatically. Students sometimes compare a processor diagram with a C structure and wonder why the sizes differ; the answer is almost always alignment.

Finally, programmers occasionally apply `packed` to every structure. This is unnecessary and can reduce performance. Most structures benefit from normal compiler alignment, and only structures that communicate directly with hardware require packed layouts.

Practice Exercises

Exercise 1

Draw the memory layout of `idt_entry_t`, labeling every field with its size and byte offset. Confirm that the total comes to exactly sixteen bytes.

Exercise 2

Temporarily remove `__attribute__((packed))`, recompile the kernel, and use `sizeof(idt_entry_t)` to measure the new size. Then explain why the processor would misinterpret the structure.

Exercise 3

Open `interrupt.s` alongside `isr.h` and trace every register the assembly interrupt stub pushes. Verify that each push corresponds to exactly one field in `registers_t`.

Exercise 4

Find three more structures in the repository. For each one, answer:

- Does it describe hardware or software?
- Is padding acceptable?
- Why was each field included?

Historical Perspective

Structures were introduced into C to make programs easier to organize, but operating systems soon adopted them for another reason. Hardware designers publish memory layouts using tables of fields and byte offsets, and kernel developers discovered that C structures provide an almost perfect translation of these tables into source code.

As processor architectures evolved, this relationship became even stronger. Modern operating systems contain hundreds of structures whose layouts are defined not by the programmer but by processor manuals, networking standards, filesystem specifications, and executable file formats. Learning to read these structures is therefore an essential systems programming skill.

Chapter Summary

Structures are far more than convenient containers for variables. In kernel development, they describe memory itself. Every field has a purpose, every offset matters, and every byte may be interpreted by hardware rather than software.

This chapter introduced the idea that a structure is a blueprint for memory. We examined how compilers align data, why padding exists, and why certain structures must be packed to match processor specifications exactly. We also saw that some structures describe objects created by C code, while others describe memory produced by assembly routines or consumed directly by the processor.

As you continue through the codebase, resist the temptation to skim over structure definitions. Read them carefully, because they reveal how the kernel models the machine.

Once you understand the structures, the functions that manipulate them become much easier to follow.

In the next chapter, we move from describing memory to accessing it directly. We will explore pointers, addresses, and the relationship between C variables and the physical memory managed by the operating system.

Chapter 3 - Pointers and Memory

Learning Objectives

After completing this chapter, you should be able to

- explain what a pointer is and why it is fundamental to systems programming,
- distinguish between an object and the address of an object,
- understand how C performs pointer arithmetic,
- explain how arrays and pointers are related,
- understand pointer casting and why kernels frequently cast addresses,
- recognize the role of `void *` and `volatile`,
- explain how pointers are used in the paging subsystem, and
- confidently trace memory accesses throughout the kernel.

3.1 Why Pointers Frighten Beginners

Ask a class of beginning programmers which part of C they find most difficult, and many will answer with a single word: pointers.

The reputation is understandable. Pointer syntax initially looks unfamiliar, the symbols `*`, `&`, and `->` appear in many different contexts, error messages often seem cryptic, and a single incorrect pointer can crash an entire program. Fortunately, pointers are much simpler than their reputation suggests. A pointer is nothing more than a variable whose value happens to be a memory address, and everything else follows from this one idea.

The difficulty comes not from pointers themselves, but from the way they force us to think about programs. Instead of thinking only about values, we must also think about where those values live. That shift in perspective is precisely why pointers are indispensable in operating systems. A kernel does not merely manipulate data; it manages memory itself.

3.2 Memory Is Just Addresses

Before discussing pointers, let us forget about C for a moment and think like the processor. To the processor, memory is simply a long sequence of numbered bytes.

```
Address

0x00000000
0x00000001
0x00000002
...
0x000B8000
...
0x00100000
...
0xFFFFFFFF80000000
```

Each address identifies exactly one byte. Nothing in memory says, “I am an integer,” and nothing says, “I am part of a structure.” Those interpretations exist only because software gives them meaning.

Suppose four consecutive bytes contain

```
78 56 34 12
```

One program might interpret them as the integer

```
0x12345678
```

Another program might interpret the same bytes as four ASCII characters, a debugger may display them as hexadecimal, and a network analyzer may treat them as part of a packet header. The bytes never change; only the interpretation changes. Pointers are the mechanism that tells the compiler how memory should be interpreted.

3.3 Objects and Addresses

Consider a simple variable.

```c
int value = 42;
```

This declaration creates an object somewhere in memory. Suppose the compiler places it at address `0x1000`. We now have two separate pieces of information: the value

```
42
```

and the address

```
0x1000
```

These are not the same thing. The value answers the question

> What is stored?

while the address answers the question

> Where is it stored?

The address operator retrieves the location of the object.

```
int *ptr = &value;
```

Now the pointer contains

```
0x1000
```

while the integer remains

```
42
```

This distinction is one of the most important concepts in C. The pointer does not contain the object; it contains the location of the object.

3.4 Dereferencing a Pointer

If a pointer stores an address, how do we obtain the object stored at that address? The answer is the dereference operator.

```
*ptr
```

The star means

> Go to the address stored in this pointer and access the object located there.

Suppose

```
int value = 42;
int *ptr = &value;
```

Then

```
*ptr = 100;
```

changes the original variable. After this statement,

```
value == 100
```

No copy was made, because both names refer to the same object. This ability to manipulate memory indirectly is one of the defining features of C, and it is also why pointers are so powerful.

3.5 Why Kernels Use Pointers Everywhere

Application programs often hide memory management behind libraries. A programmer calls

```
malloc()
```

and receives usable memory, while the operating system decides where that memory comes from.

Kernel programmers have no operating system beneath them, so the kernel performs the allocation itself — every page of memory, every process, every interrupt stack, and every page table. Everything begins with an address. Consequently, almost every important kernel function receives or returns pointers.

Consider

```
page_t *get_page(
    u64int address,
    int make,
    page_table_t *pml4);
```

The function does not return an entire page-table entry; it returns the address of one. Returning a pointer avoids copying large structures and allows the caller to modify the original object directly.

Once you begin reading the repository, you will notice this pattern everywhere. Pointers are not exceptional; they are the normal way of referring to kernel data.

3.6 Hardware Is Memory Too

One of the first pointer declarations in the repository appears in `monitor.c`.

```
u16int *video_memory = (u16int *)0xB8000;
```

This single line illustrates several important ideas. First, `0xB8000` is a physical memory address, because the VGA hardware maps its text buffer to this location. Second, the kernel converts the integer into a pointer, and the cast tells the compiler

> Treat this address as an array of sixteen-bit values.

Finally, the pointer behaves like any ordinary array.

```
video_memory[0] = 0x0F41;
```

writes the character `'A'` to the upper-left corner of the screen. Nothing magical happens: the processor stores two bytes in memory, and the VGA controller notices the change and updates the display. This is one of the defining characteristics of systems programming — hardware often appears as memory.

3.7 Understanding Pointer Arithmetic

Pointers support arithmetic. Unlike ordinary arithmetic, however, the compiler automatically accounts for the size of the referenced object. Suppose

```
u16int *video_memory;
```

points to

`0xB8000`

Adding one,

```
video_memory++;
```

leads many beginners to expect

`0xB8001`

but the compiler instead produces

`0xB8002`

because a `u16int` occupies two bytes. Likewise,

```
video_memory + 10
```

moves twenty bytes forward. This behavior allows arrays to work naturally, because the compiler performs the multiplication automatically and the programmer simply counts elements instead of bytes.

3.8 Arrays and Pointers

Arrays and pointers are closely related. Given

```
char buffer[128];
```

the expression

```
buffer
```

evaluates to the address of the first element. This relationship explains why

```
buffer[5]
```

is equivalent to

```
*(buffer + 5)
```

The compiler translates array indexing into pointer arithmetic, and understanding this equivalence makes many kernel data structures much easier to read. Page tables are arrays, descriptor tables are arrays, and process tables are arrays, and the kernel accesses nearly all of them through pointers.

3.9 Pointers to Structures

Structures rarely travel by value inside the kernel. Instead, functions receive pointers.

```
page_t *page;
```

Fields are then accessed using the arrow operator.

```
page->present = 1;
page->rw = 1;
```

The arrow operator is simply shorthand, and the compiler interprets it as

```
(*page).present
```

The shorter notation improves readability. Whenever you encounter `->`, remember that the program is following an address before accessing the requested field.

3.10 Casting Addresses

Kernel code frequently converts integers into pointers.

```
(u16int *)0xB8000
```

Students sometimes assume that the cast changes memory. It does not. The cast changes only the compiler's interpretation. Before the cast,

```
0xB8000
```

is simply a number; after the cast, the compiler treats the same bits as the address of sixteen-bit objects.

Casting therefore changes meaning rather than data. Learning this distinction helps explain much of the code found in low-level systems software.

3.11 Generic Pointers

Not every function knows what type of object it will eventually manage. Memory allocators provide an excellent example. The allocator simply reserves bytes; it does not know whether those bytes will later hold a process descriptor, a page table, or a filesystem object. Generic allocators therefore often return

```
void *
```

A `void *` represents an address without specifying the object's type, and the caller later converts the pointer.

```
page_t *page = (page_t *)kmalloc(sizeof(page_t));
```

The allocator remains general, and the caller determines the interpretation.

3.12 The Meaning of `volatile`

Most compiler optimizations assume that memory changes only when the program writes to it. Hardware violates this assumption. Consider a status register that changes whenever a device receives new data. Even if the program never writes to the register, the value may change at any moment.

The keyword

```
volatile
```

tells the compiler never to assume that repeated reads produce identical values. Every access must occur exactly as written. Hardware registers, memory-mapped devices, and certain synchronization variables therefore frequently use `volatile`, because without it the compiler might optimize away operations that are essential for correct hardware behavior.

3.13 Virtual and Physical Addresses

One subtle point deserves attention. Pointers inside the kernel usually represent **virtual** addresses rather than physical addresses.

Early in the tutorial, the paging system creates an identity mapping, so virtual address

```
0xB8000
```

refers to physical address

```
0xB8000
```

The kernel keeps this arrangement for the whole tutorial. It is linked at `0x100000` and continues to execute from identity-mapped low memory, so a pointer to kernel code or data keeps equaling its physical address.

The one place where virtual and physical part ways is the kernel heap. When the heap is created (Chapter 8), it is placed high in the 64-bit address space, at `0xFFFF800000000000`, in a region that is deliberately *not* identity-mapped. A pointer into the heap still holds an address, but that address is now purely virtual: the Memory Management Unit translates it to some physical frame elsewhere in RAM. From the perspective of C, nothing changes — a pointer is a pointer; from the perspective of the hardware, the heap's addresses no longer name the bytes they appear to.

Understanding this distinction prepares us for the paging subsystem (Chapter 7) and the heap that builds upon it (Chapter 8).

3.14 Common Mistakes

Beginning programmers often confuse an object with its address. Others perform arithmetic on bytes instead of typed objects, and many accidentally dereference uninitialized pointers or cast pointers to integers that are too small to hold a sixty-four-bit address.

Another common mistake is assuming that pointer arithmetic operates on bytes. It does not. The compiler scales every operation according to the referenced type.

Finally, many students think pointers are inherently dangerous. Pointers are not dangerous; incorrect addresses are dangerous. A correctly initialized pointer is one of the safest and most expressive tools available in C.

Practice Exercises

Exercise 1

Trace every access to `video_memory` in `monitor.c`, then draw the memory addresses that correspond to the first twenty characters displayed on the screen.

Exercise 2

Locate every function in the repository that returns a pointer. For each one, explain why returning the object itself would be inefficient or incorrect.

Exercise 3

In the paging chapter's `main.c`, attempt to read from several addresses, some mapped and some unmapped. Predict which accesses will succeed before you run the kernel, then compare your predictions with the observed behavior.

Exercise 4

Write a small function that receives a pointer to a structure and modifies one of its fields, then rewrite it to take the structure by value instead. Compare the generated machine code with `objdump`. Which version copies more data?

Historical Perspective

Pointers have been part of C since the language's earliest days because they reflect the way computers actually work. Earlier programming languages often hid memory addresses from the programmer, making systems programming difficult or impossible. Dennis Ritchie designed C with the opposite philosophy: the language exposes addresses directly while providing enough abstraction to keep programs readable. This balance between abstraction and control explains why C remains the dominant language for operating systems more than fifty years after its creation.

Chapter Summary

Pointers are not merely another feature of C. They are the language through which software describes memory. Every operating system relies on them because every operating system ultimately manages addresses rather than abstract values.

In this chapter, we examined the relationship between objects and addresses, learned how pointer arithmetic works, explored arrays and structures through pointers, and saw how hardware itself often appears as memory. We also introduced `void *`, `volatile`, and the distinction between virtual and physical addresses—ideas that will become increasingly important as the kernel grows more sophisticated.

In the next chapter, we leave ordinary memory behind and learn how the kernel communicates directly with the processor using inline assembly. There we discover where the

C language ends, where assembly begins, and how the two cooperate to build a modern operating system.

Chapter 4 - Inline Assembly and Talking to Hardware

Learning Objectives

After completing this chapter, you should be able to

- explain why kernels cannot be written entirely in C,
- understand the role of inline assembly in a modern operating system,
- read simple GNU inline assembly statements,
- understand the purpose of assembly constraints,
- explain why the `volatile` keyword appears in inline assembly,
- understand how C and assembly cooperate during kernel execution, and
- modify simple hardware access routines with confidence.

4.1 Why C Is Not Enough

One of the attractions of C is that it is often described as a "portable assembly language." The description is only partly true. C gives programmers direct access to memory, pointers, and low-level data structures. It maps naturally onto the underlying hardware, and optimizing compilers produce machine code that rivals handwritten assembly for most tasks.

Yet C deliberately avoids exposing certain processor features. The language has no statement for loading the Interrupt Descriptor Table, no operator for enabling interrupts, and no keyword for reading the CR3 control register or invalidating a page-table entry. These omissions are intentional. The C language was designed to be portable across many processor architectures, but instructions such as `lidt`, `lgdt`, `mov %cr3`, and `invlpg` exist only on x86 processors, and including them in the language would make C less portable.

Operating systems, however, are not ordinary applications. A kernel must initialize the processor, configure interrupt handling, manage page tables, switch between processes, and communicate directly with hardware devices, and many of these operations require instructions that exist outside the C language. Whenever you encounter assembly in

this codebase, it is because the programmer has reached one of the boundaries of C. Assembly is not replacing C; it is extending it.

4.2 Where Assembly Appears in the Kernel

Many students imagine that operating systems are written mostly in assembly language. That was true decades ago, but modern kernels are overwhelmingly written in C, and assembly appears only where it provides capabilities unavailable in the language.

In this tutorial, assembly is used in several distinct places. Boot code initializes the processor before C can execute. Interrupt stubs save processor state before transferring control to C. Context-switching routines restore registers when changing processes. Hardware access routines execute instructions such as `inb` and `outb`. Control-register operations configure paging and processor state. Everything else is ordinary C.

This division of responsibility is worth remembering: C implements policy, while assembly performs operations that only the processor can perform.

4.3 Inline Assembly

Assembly can be placed in separate `.s` files, but small hardware operations are often easier to write directly inside a C function. In this codebase, the port routines live in `common.c` and are prototyped in `common.h`. Consider the familiar output routine.

```
void outb(u16int port, u8int value)
{
    asm volatile (
        "outb %1, %0"
        :
        : "dN"(port), "a"(value)
    );
}
```

At first glance, this statement looks intimidating, but every part has a specific purpose. The keyword

```
asm
```

tells the compiler that the following text is assembly language, and the keyword

```
volatile
```

tells the compiler that this instruction has important side effects and must not be removed or reordered during optimization. The string

```
"outb %1, %0"
```

contains the assembly instruction itself, and the remaining fields describe how C variables should be placed into processor registers before the instruction executes.

Although the syntax appears unusual, the underlying idea is simple. The compiler still generates almost the entire function; it merely inserts one assembly instruction at the appropriate point.

4.4 Understanding the `outb` Instruction

To understand why inline assembly exists, we must first understand the hardware operation it performs. The x86 architecture provides two methods of communicating with hardware devices: memory-mapped I/O and port-mapped I/O.

The original IBM PC adopted port-mapped I/O for many peripheral devices. Instead of reading or writing memory addresses, the processor executes special instructions that communicate with numbered ports. The `outb` instruction writes one byte to an I/O port, and conceptually it performs an operation similar to

```
HardwarePort[port] = value;
```

except that the hardware ports do not exist in ordinary memory. Only the processor understands how to perform this operation, and the C language has no equivalent statement, so the instruction must be written in assembly.

4.5 Breaking Down the Syntax

Let us examine the inline assembly statement one component at a time.

```
asm volatile(
    "outb %1, %0"
    :
    :
    "dN"(port),
    "a"(value)
);
```

The first field contains the assembly template.

```
"outb %1, %0"
```

The placeholders `%0` and `%1` refer to operands supplied later in the statement. The next field normally lists output operands, but because `outb` produces no result, the output section is empty. The following field specifies input operands.

```
"dN"(port)
"a"(value)
```

These are called **constraints**. A constraint tells the compiler where a value must be placed before executing the instruction. The `"a"` constraint requests the accumulator register, which for an 8-bit operation means the `AL` register, while the `"d"` constraint requests the `DX` register, which the processor uses for port addresses.

The compiler performs the necessary register allocation automatically. Instead of worrying about saving and restoring registers yourself, you simply describe what the instruction requires. This cooperation between the programmer and compiler is one of the strengths of inline assembly.

4.6 Why `volatile` Matters

Compilers are remarkably good at eliminating unnecessary work. Suppose a program computes

```
x = 5;
x = 5;
```

The compiler notices that the second assignment changes nothing and removes the redundant instruction. This optimization improves performance.

Hardware access is different. Suppose the kernel executes

```
outb(0x20, 0x20);
```

The compiler cannot determine the effect of writing to port `0x20`. Perhaps it acknowledges an interrupt, perhaps it resets a device, perhaps it starts a disk controller — the effect occurs outside the compiler's view. The keyword `volatile` tells the compiler that the assembly instruction has important side effects even if no ordinary variables appear to change. Without `volatile`, aggressive optimization could remove or reorder hardware operations, producing subtle and difficult-to-diagnose bugs.

4.7 Reading from Hardware

Writing to a port is only half the story. Many devices return information, and reading requires another instruction.

```
u8int inb(u16int port)
{
    u8int result;

    asm volatile(
        "inb %1, %0"
        : "=a"(result)
        : "dN"(port)
    );

    return result;
}
```

Notice the difference: this function contains an output operand.

```
"=a"(result)
```

The equals sign tells the compiler that the assembly instruction writes a value into the accumulator register, and after the instruction finishes, the compiler copies the register into the C variable `result`. From the perspective of the calling code, the function behaves like any ordinary C function, and the assembly remains hidden inside the implementation.

4.8 An Alternative: `static inline`

In this tutorial, `outb`, `inb`, and `inw` are ordinary functions. Each is defined once in `common.c`, prototyped in `common.h`, and called like any other function. This keeps the code plain and easy to follow, at the cost of a real function call — a `call` instruction and a `ret` — every time the kernel touches a port.

Many other kernels write these same tiny routines differently, declaring them

```
static inline
```

in a header rather than defining them in a `.c` file. It is worth knowing what those two keywords buy, because you will meet the pattern constantly once you read code beyond this repository.

The keyword `inline` suggests that the compiler replace the call with the function body. Instead of generating

```
call outb
```

the compiler drops the single assembly instruction directly into the caller, eliminating the call overhead. The keyword `static` limits the function's visibility to the current source file, which matters when a routine is defined in a header that many files include: without `static`, every file would export its own conflicting copy. Together the two keywords let a header hand out a hardware routine that costs nothing at the call site. The compiler is free to ignore the `inline` suggestion, but for functions this small it usually complies.

Neither approach changes what the instruction does. The choice is one of packaging — a function in a `.c` file, as here, or an inlined helper in a header — and recognizing both is part of reading kernel code fluently.

4.9 C and Assembly Working Together

It is tempting to think of C and assembly as competing languages, but kernel development teaches a different lesson: each language performs the tasks it handles best. Assembly initializes the processor, while C configures data structures. Assembly saves registers during interrupts, while C decides which interrupt handler should execute. Assembly loads the page-table register, while C constructs the page tables themselves.

Neither language replaces the other; instead, they cooperate continuously. Understanding where one language ends and the other begins is an important step toward reading

operating system code confidently.

4.10 Common Mistakes

Inline assembly is powerful, but it is also unforgiving. One common mistake is forgetting the `volatile` keyword, allowing the compiler to optimize away hardware operations. Another is specifying incorrect constraints, causing values to appear in the wrong registers. Students also sometimes assume that assembly instructions automatically preserve registers, but many instructions modify processor state, and the programmer must tell the compiler which registers or memory locations are affected.

Finally, beginners often attempt to solve problems in assembly that are better expressed in C. Remember the guiding principle of modern kernel development: write as much code as possible in C, and use assembly only when the processor requires it.

Practice Exercises

Exercise 1

Locate every inline assembly statement in the repository. For each one, answer:

- Why can't this operation be written in standard C?
- Which processor instruction does it execute?
- Which subsystem depends on it?

Exercise 2

Write a function that reads the CR3 control register with inline assembly and prints its value in hexadecimal. Explain why that value changes during a context switch.

Exercise 3

Study the implementations of `outb()` and `inb()`, and identify every constraint the compiler is given. Consult the GCC documentation and explain what each constraint means.

Exercise 4

Modify the keyboard driver to mask keyboard interrupts temporarily by writing to the Programmable Interrupt Controller. Observe the effect on keyboard input, then restore normal operation.

Historical Perspective

Early operating systems were written largely in assembly language because compilers generated relatively inefficient code and hardware resources were limited. As compiler technology improved, most kernel code migrated to C. Today, even large production kernels such as Linux contain relatively little assembly compared to their C source.

The assembly that remains performs highly specialized tasks: processor initialization, interrupt entry and exit, context switching, atomic synchronization primitives, and hardware-specific optimizations. The lesson is reassuring. Learning kernel development does not require becoming an assembly-language expert; it requires understanding enough assembly to recognize where the hardware interface begins.

Chapter Summary

This chapter introduced one of the defining characteristics of systems programming: not every processor capability is available through the C language. Whenever the kernel needs to execute privileged instructions, access I/O ports, manipulate control registers, or initialize the processor, it crosses the boundary between C and assembly.

Fortunately, these boundaries are narrow. Modern kernels are written almost entirely in C, and inline assembly serves as a carefully controlled bridge to the underlying hardware. Once you understand how operands, constraints, and the `volatile` keyword work together, even unfamiliar assembly routines become approachable.

In the next chapter, we examine how the kernel responds to events originating outside the currently executing program. We leave the world of ordinary function calls and enter the world of interrupts, exceptions, and hardware-driven control flow, where the processor—not the programmer—decides what code executes next.

Chapter 5 - Interrupts, Function Pointers, and Event-Driven Programming

Learning Objectives

After completing this chapter, you should be able to

- explain why operating systems rely on interrupts,
- distinguish between ordinary function calls and interrupt-driven execution,
- understand function pointers and why kernels use them extensively,
- explain how interrupt handlers are registered and dispatched,
- understand the purpose of the `registers_t` structure,
- trace the complete path from a hardware interrupt to a C function,
- understand why interrupt handlers receive pointers rather than copies of processor state, and
- confidently read and modify the interrupt subsystem in the codebase.

5.1 Programs Normally Execute in Order

Everything you have learned about C so far assumes a simple model of execution. A program begins at one function, it calls another function, that function returns, and execution continues where it left off. The flow of control is entirely predictable.

```
main()
    |
    +--> initialize()
    |
    +--> setup_paging()
    |
    +--> start_scheduler()
```

The programmer decides exactly which function executes next, and this model works

well for ordinary applications. Operating systems cannot rely on it. A keyboard does not wait until the kernel reaches the correct line of code, a timer does not ask permission before generating its next tick, and a network card does not postpone receiving packets because the kernel is busy.

The processor must be able to interrupt whatever it is currently doing and immediately execute code that responds to external events. This ability is called **interrupt-driven execution**, and it is one of the defining characteristics of every modern operating system.

5.2 What Is an Interrupt?

Imagine writing a text editor. While the program is waiting for keyboard input, the processor executes millions of instructions every second, and checking the keyboard continuously would waste enormous amounts of processor time. Instead, the hardware waits. When the user presses a key, the keyboard controller sends a signal to the processor, the processor immediately suspends the current program and begins executing the keyboard interrupt handler, and after the handler finishes, execution resumes exactly where it stopped. From the perspective of the running program, it is almost as though nothing happened.

Interrupts therefore allow hardware devices to attract the processor's attention only when necessary. Instead of constantly asking

> Has something happened?

the processor is told

> Something happened. Go handle it.

This event-driven model makes modern computers efficient.

5.3 Interrupts Versus Function Calls

At first glance, interrupt handlers resemble ordinary functions. Both execute C code, both have parameters, and both eventually return. The similarities end there.

When one function calls another, the caller decides when the call occurs.

```
initialize_keyboard();
```

An interrupt is different, because the processor itself decides when execution changes. The current instruction completes, the processor saves its state, the interrupt descriptor table is consulted, and control transfers to the appropriate handler. The interrupted program has no opportunity to refuse.

Interrupts are therefore **asynchronous**: they occur independently of the currently executing program. Understanding this distinction helps explain why interrupt handlers must be short, efficient, and carefully written.

5.4 From Hardware to C

Let us follow the complete path of a keyboard interrupt.

The keyboard controller detects a key press.

↓

The Programmable Interrupt Controller signals the processor.

↓

The processor identifies the interrupt number.

↓

The Interrupt Descriptor Table provides the address of the interrupt stub.

↓

Assembly code saves processor registers.

↓

The assembly stub calls

```
isr_handler(registers_t *regs);
```

↓

The C code examines the interrupt and dispatches it to the correct handler.

Although many components participate, the overall process follows a simple pattern: hardware generates an event, assembly preserves processor state, and C decides what to do. Each language performs the task it handles best.

5.5 The `registers_t` Structure

The most important parameter received by every interrupt handler is

```
void isr_handler(registers_t *regs)
```

At first glance, this seems like an ordinary structure. It is not. Unlike most structures, `registers_t` does not describe an object created by C. It describes the processor state saved by the interrupt entry code.

When an interrupt occurs, several values already exist inside the processor: the instruction pointer, the code segment, the processor flags, and the current stack pointer. The assembly interrupt stub saves the remaining registers before transferring control to C, and the resulting stack frame is interpreted as a `registers_t` object.

Notice the direction of reasoning. The structure does not determine what appears on the stack; the stack determines what the structure must look like. Changing the order of the fields changes the interpretation of every saved register, so the processor continues

executing correctly while the kernel begins reading incorrect information. This explains why interrupt-related structures rarely change once they are written.

5.6 Why the Handler Receives a Pointer

The handler receives

`registers_t *regs`

instead of

`registers_t regs`

This choice is deliberate. The saved registers already exist on the interrupt stack, and copying the entire structure would consume additional time during every interrupt. More importantly, modifications would affect only the copy.

By passing a pointer, the handler accesses the original stack frame. Suppose a future system call changes the instruction pointer before returning to user mode. Changing

`regs->rip`

modifies the value that the processor restores during `iretq`, with no additional copying necessary.

Passing a pointer is therefore both faster and more flexible. This design pattern appears throughout operating systems, where large structures are almost always manipulated through pointers.

5.7 Function Pointers

Eventually the kernel must decide which handler should respond to a particular interrupt. One possibility would be an enormous `switch` statement.

```
switch(interrupt_number) {
    ...
}
```

Although functional, this approach becomes difficult to maintain. Instead, the kernel stores addresses of functions.

`interrupt_handlers[interrupt_number]`

This introduces one of the most useful features of C. A function has an address, and that address can be stored inside a variable. Such variables are called **function pointers**. Conceptually, a function pointer behaves like any other pointer, except that instead of referring to data, it refers to executable code.

5.8 Calling Through a Function Pointer

Suppose we declare

```
typedef void (*isr_t)(registers_t *);
```

An object of type `isr_t` stores the address of a function that accepts one parameter of type `registers_t *`. For now a handler simply acts and returns nothing; when we reach multitasking in Appendix C, this type gains a return value — a handler will hand back the frame that should run next — but the shape you learn here is the foundation. Registering a handler becomes straightforward.

```
interrupt_handlers[33] = keyboard_handler;
```

Later, the dispatcher executes

```
interrupt_handlers[33](regs);
```

Although this syntax initially appears unusual, it simply means

> Call the function whose address is stored in this array element.

Nothing more. The processor jumps to whatever address the pointer contains, and this flexibility allows the kernel to replace handlers dynamically without changing the dispatcher itself.

5.9 Event Dispatching

The interrupt dispatcher illustrates an important software engineering principle: the dispatcher does not know what every interrupt means. It simply forwards events to the appropriate handler. The keyboard handler understands keyboards, the timer handler understands timers, and the page-fault handler understands memory management.

Separating these responsibilities keeps the kernel modular. As additional devices are added, the dispatcher remains unchanged, and only the handler table grows.

This technique appears repeatedly throughout operating systems. Network stacks, device drivers, system calls, and filesystem operations all rely on some form of dispatch table.

5.10 Interrupt Context

Interrupt handlers execute under unusual circumstances. They interrupt code that was already running, they execute on the current processor state, they must preserve registers correctly, and they often execute with interrupts temporarily disabled. Consequently, interrupt handlers follow different rules than ordinary functions: they should complete quickly, avoid blocking operations, and minimize memory allocation whenever possible.

Many production kernels divide interrupt processing into two stages. The interrupt handler performs only essential work, and more expensive processing occurs later in a safer execution context. Although this educational kernel is much simpler, the underlying philosophy remains the same. Respond quickly, return promptly, and allow normal execution to continue.

5.11 Reading the Interrupt Code

Open `isr.c` and begin with the dispatcher. Ignore the details, and simply identify the major stages.

Receive processor state.

↓

Determine interrupt number.

↓

Call registered handler.

↓

Return.

Once this overall structure becomes clear, examine individual helper functions and notice how each one performs one specific task. Good kernel code often resembles good mathematical proofs: each step is small, each step has a clear purpose, and together they produce complex behavior.

5.12 Common Mistakes

Beginning programmers often forget that interrupts can occur almost anywhere. They assume that handlers execute only after the current function returns, but they do not.

Another common mistake is modifying the order of fields inside `registers_t`. The compiler accepts the change, the processor continues saving registers exactly as before, and the handler silently begins reading incorrect values.

Students also confuse function pointers with ordinary pointers. Remember the difference: data pointers reference objects, while function pointers reference executable code.

Finally, avoid performing lengthy computations inside interrupt handlers. Every additional instruction delays the interrupted program, and efficient handlers improve the responsiveness of the entire system.

Practice Exercises

Exercise 1

Trace a timer interrupt's complete path, from the hardware signal to the moment control returns to the interrupted program. Draw each stage on paper.

Exercise 2

Locate every function pointer declaration in the repository. For each one, identify:

- what type of function it references,
- where the pointer is initialized,
- where it is called.

Exercise 3

Register a custom interrupt handler that prints a message whenever the timer interrupt occurs. Verify that it runs exactly once for every timer tick.

Exercise 4

Modify the dispatcher to count how many times each interrupt has occurred, and display the running counts on the screen every few seconds. Which interrupt appears most frequently, and why?

Historical Perspective

The earliest computers had no interrupts. Programs repeatedly examined hardware devices in a process known as polling, and although simple, polling wasted valuable processor time. The introduction of hardware interrupts transformed computer architecture. Instead of continuously asking devices whether they needed attention, processors could devote their resources to useful computation until hardware signaled that service was required.

Modern operating systems extend this idea even further. Interrupts drive process scheduling, networking, storage, timers, power management, and virtually every interaction between software and hardware. Without interrupts, multitasking operating systems would be impractical.

Chapter Summary

Interrupts change the way we think about program execution. Instead of following a predictable sequence of function calls, the processor can suspend the current program at almost any moment to respond to hardware events. This event-driven model is fundamental to operating systems.

In this chapter, we followed the complete path from a hardware interrupt to a C interrupt handler, examined the `registers_t` structure, learned why interrupt handlers receive

pointers, and introduced function pointers as a mechanism for dispatching events. We also saw how interrupt dispatch tables make the kernel modular and extensible.

In the next chapter, we build on these ideas by exploring dynamic memory management. Instead of reacting to external events, the kernel will begin creating and managing its own memory structures through allocators, heaps, and paging. Those components rely heavily on the pointer and structure concepts introduced in the previous chapters.

Chapter 6 - Dynamic Memory Management

Learning Objectives

After completing this chapter, you should be able to

- explain why kernels need dynamic memory allocation,
- distinguish between static, stack, and heap memory,
- understand the placement allocator used in the early kernel,
- explain how `kmalloc()` works,
- understand why physical addresses are sometimes returned separately,
- follow the flow of memory allocation through the kernel,
- recognize how dynamic allocation supports nearly every kernel subsystem, and
- prepare for the transition from simple allocation to full virtual memory management.

6.1 Why the Kernel Cannot Know Everything in Advance

So far, the kernel has relied mostly on memory that already exists. Global variables are allocated by the linker, local variables are placed on the stack, and the compiler determines their locations before the program ever runs. This approach works only for objects whose sizes and lifetimes are known beforehand.

Operating systems quickly outgrow this model. A process table cannot be completely static because new processes are created while the system is running. A page table cannot be allocated entirely at compile time because new virtual memory mappings appear as programs execute. A filesystem cannot predict how many files users will create.

The kernel therefore needs a way to request memory while it is running. This capability is called **dynamic memory allocation**. Instead of asking the compiler for memory, the kernel becomes its own memory manager.

6.2 Three Places Where Memory Lives

Before discussing the allocator, it helps to understand the major regions of memory inside a program.

The first region is **static memory**. Global variables and static variables are placed here, and their lifetime extends from boot until shutdown.

```
static int timer_ticks;
```

The second region is the **stack**. Every function call creates a stack frame containing local variables, return addresses, and saved registers.

```
void foo(void)
{
    int x;
}
```

The variable `x` disappears automatically when the function returns.

The third region is the **heap**. Unlike the other two, heap memory has no predetermined lifetime. The program requests memory when needed and releases it when it is no longer required.

```
+----------------------+
| Kernel Image         |
+----------------------+
| Static Data          |
+----------------------+
| Heap                 |
|   grows upward       |
+----------------------+
|                      |
|   Free Memory        |
|                      |
+----------------------+
| Stack                |
| grows downward       |
+----------------------+
```

The heap is the most flexible part of memory, and it is also the most complicated to manage.

6.3 The Simplest Allocator

The earliest chapters of the kernel avoid complicated heap management. Instead, they use a **placement allocator**, and the idea is remarkably simple.

Imagine a bookmark placed inside memory. The bookmark always points to the next unused byte. It does not start at zero, nor at the bottom of RAM; it starts at `end`, a symbol

the linker script places at the first byte just past the kernel image, so the allocator can never hand out memory that the kernel itself occupies.

When the kernel requests memory,

```
kmalloc(4096);
```

the allocator returns the current bookmark and then moves the bookmark forward.

```
old placement_address
   |
   v
end -------------------------->

after allocation

end + 0x1000 ------------------>
```

Nothing is ever freed, nothing is reused, and memory simply grows upward. Although primitive, this allocator is ideal during early kernel development because it is easy to understand and almost impossible to break. Complex allocators come later; simplicity comes first.

6.4 Walking Through `kmalloc()`

Open the implementation of `kmalloc()`. Ignore the details at first, and instead identify its major steps.

Receive the requested size.

↓

Align the allocation if necessary.

↓

Remember the current placement address.

↓

Advance the placement address.

↓

Return the previous address.

Notice that no searching occurs — no linked lists, no trees, and no bookkeeping structures. The allocator simply moves forward through memory. This explains why early kernel allocators are extremely fast, because each allocation requires only a handful of instructions. The price is that allocated memory cannot be reclaimed, but for bootstrapping a kernel, this tradeoff is entirely reasonable.

6.5 Alignment Matters

Not every object can begin at an arbitrary address. Suppose the kernel allocates a page table. Each page table occupies one page, a page is exactly 4096 bytes, and the processor expects page tables to begin on page boundaries. An address such as

`0x101234`

is unacceptable, while an address such as

`0x102000`

is correct.

The allocator therefore supports page alignment. Before allocating memory, it rounds the placement address upward until it reaches the next page boundary. Conceptually,

`0x101234`

becomes

`0x102000`

and the unused bytes are skipped. Alignment wastes a small amount of memory, but it greatly simplifies hardware requirements. Throughout systems programming, small amounts of wasted memory are often exchanged for simpler and faster algorithms.

6.6 Returning Physical Addresses

Some versions of the allocator provide a function such as

```
kmalloc_ap(size, &physical_address);
```

At first this interface seems unusual. Why return two addresses? The answer lies in virtual memory. The pointer returned by `kmalloc()` is the address used by the kernel, while the physical address identifies where the memory actually resides in RAM.

While only the placement allocator exists, the two values are identical: everything it hands out comes from the identity-mapped low region, where a virtual address equals its physical address. That changes once the kernel heap is running (Chapter 8). The heap lives high in the address space, at `0xFFFF800000000000`, which is not identity-mapped, so a heap allocation's virtual and physical addresses genuinely differ — and the allocator has to ask the page tables to translate one into the other. The reason to return the physical address at all is that many processor structures, page tables chief among them, must be given physical frame addresses rather than the virtual pointers the kernel normally works with.

This design illustrates an important principle: interfaces are often designed with future expansion in mind.

6.7 Memory Allocation Throughout the Kernel

As the operating system grows, almost every subsystem depends on dynamic allocation. The paging subsystem allocates new page tables, the scheduler allocates process descriptors, the filesystem allocates metadata structures, device drivers allocate buffers, and network stacks allocate packets. Even the heap manager eventually allocates memory for its own bookkeeping information.

This dependency explains why memory management is often considered one of the foundational components of an operating system. Without it, every other subsystem becomes unnecessarily rigid.

6.8 Following an Allocation

Suppose the paging subsystem needs a new page table. The code requests memory,

```
page_table_t *table = kmalloc(...);
```

the allocator returns an address, the paging code initializes the entries, and the processor later reads the completed table. Although several subsystems participate, the sequence remains simple.

```
Paging Code

      |
      v

kmalloc()

      |
      v

Placement Allocator

      |
      v

Returns Memory

      |
      v

Paging Initializes Structure

      |
      v
```

```
Processor Uses Structure
```

Tracing this sequence yourself is an excellent exercise. Follow the pointer from its creation until the processor eventually uses the data. Doing so develops the habit of reading systems code as a sequence of cooperating components rather than isolated functions.

6.9 Why Memory Is Not Freed Yet

Students often notice that the placement allocator never releases memory. Is this a bug? No — it is a design decision.

The early kernel allocates only a small number of permanent objects, such as interrupt tables, page tables, and kernel stacks. These structures exist for the lifetime of the operating system, so freeing them would accomplish little.

A more sophisticated heap manager appears later. At that point, memory blocks gain headers, free lists, coalescing algorithms, and fragmentation management. Learning these ideas is much easier after first understanding the simpler placement allocator, because complexity should arrive gradually.

6.10 Reading the Allocator

When reading `kheap.c`, resist the temptation to examine every statement immediately. Begin with the interface. What functions exist? What arguments do they receive? What values do they return?

Only after understanding the public interface should you examine the implementation. This mirrors professional software development, where most programmers encounter interfaces before implementations. Learning to read code from the outside inward is a valuable habit.

6.11 Common Mistakes

Beginning kernel programmers often assume that `kmalloc()` behaves exactly like the standard C library's `malloc()`. It does not, because the placement allocator never frees memory.

Another common mistake is forgetting page alignment when allocating processor structures. Page tables that begin at arbitrary addresses will not work correctly.

Students also confuse virtual and physical addresses. Remember that a pointer describes how the processor accesses memory, while a physical address describes where memory actually exists. These concepts coincide early in the tutorial but diverge as virtual memory becomes more sophisticated.

Finally, avoid allocating objects on the stack when their lifetime extends beyond the current function. Kernel data structures usually outlive individual function calls, so heap allocation is often the appropriate choice.

Practice Exercises

Exercise 1

Trace every call to `kmalloc()` in the repository. For each allocation, determine:

- what object is being created,
- why it cannot be allocated statically,
- how long the object remains alive.

Exercise 2

Modify the allocator to print the current placement address after every allocation, then run the kernel and watch how memory usage grows during boot. Which subsystem performs the most allocations?

Exercise 3

Force every allocation to begin on a page boundary, and measure the resulting increase in memory consumption. Discuss the tradeoff between alignment and wasted space.

Exercise 4

Draw the kernel memory map after boot, identifying:

- the kernel image,
- the heap,
- free memory,
- page tables,
- the stack.

Then indicate which regions grow upward and which grow downward.

Historical Perspective

Early operating systems often relied on extremely simple allocation strategies because memory was scarce and systems performed relatively few dynamic allocations after boot. As multitasking, networking, graphical interfaces, and virtual memory became commonplace, kernels required increasingly sophisticated allocators capable of managing millions of allocations efficiently.

Modern kernels employ slab allocators, buddy allocators, per-CPU caches, and NUMA-aware allocation strategies. Despite this sophistication, many kernels still begin booting with an allocator remarkably similar to the placement allocator studied in this chapter. The simplest solution is often the best way to bootstrap a more complex one.

Chapter Summary

Dynamic memory allocation allows the kernel to create data structures while the system is running rather than relying solely on compile-time memory. We examined the distinction between static memory, the stack, and the heap, introduced the placement allocator, and followed the flow of a typical allocation from request to use. We also explored alignment, physical addresses, and the reasons why early kernels deliberately avoid freeing memory.

Although the placement allocator is intentionally simple, it establishes the foundation for every memory-management subsystem that follows. Understanding how memory is acquired is the first step toward understanding how memory is mapped, protected, shared, and eventually reclaimed.

In the next chapter, we examine virtual memory and paging. There we will discover how the processor transforms virtual addresses into physical addresses and why this mechanism lies at the heart of every modern operating system.

Chapter 7 - Paging and Virtual Memory

Learning Objectives

After completing this chapter, you should be able to

- explain why modern operating systems use virtual memory,
- distinguish between virtual and physical addresses,
- understand the purpose of paging,
- explain the organization of the four-level page-table hierarchy in x86-64,
- understand how the Memory Management Unit (MMU) translates addresses,
- trace a virtual address through the paging subsystem,
- explain the role of page faults, and
- confidently read the paging code in the kernel.

7.1 The Problem with Physical Memory

Imagine writing a program that always stores its data beginning at physical address

`0x00100000`

Now imagine loading a second program that also expects to begin at exactly the same address. The problem is immediate: both programs want to occupy the same memory, and only one can succeed.

Early computers solved this problem by running only one program at a time. Modern computers do something far more interesting. Each program believes it owns the machine — every process thinks it begins at familiar addresses, every process believes it has its own stack, and every process believes it has its own heap. None of these beliefs are completely true. The operating system creates the illusion, and virtual memory is the mechanism that makes this illusion possible.

7.2 The Illusion of Memory

Suppose two processes both access address

```
0x400000
```

At first this seems impossible. How can two different programs use exactly the same address? The answer is that the address is **virtual**, not physical. The processor never accesses physical memory directly; instead, it performs a translation.

```
Process A

Virtual Address
      |
      v
0x400000
      |
      v
MMU
      |
      v
Physical Address
0x105000
```

```
Process B

Virtual Address
      |
      v
0x400000
      |
      v
MMU
      |
      v
Physical Address
0x2F9000
```

Both programs see the same virtual address, but the processor silently maps each one to different physical memory, and neither process knows the difference. This simple idea lies at the heart of every modern operating system.

7.3 Why Virtual Memory Exists

Virtual memory solves several important problems simultaneously. First, it isolates processes, because one program cannot accidentally overwrite another program's memory when each process has its own address space. Second, it simplifies programming, because applications no longer need to know where they are loaded in physical memory.

Third, it enables efficient memory management: unused pages can be moved to disk, shared libraries can occupy the same physical pages while appearing in multiple processes, and kernel memory can remain mapped regardless of which process is executing.

One mechanism solves many different problems, and this elegance explains why paging became one of the defining innovations in computer architecture.

7.4 Pages Instead of Bytes

Translating every byte individually would require enormous tables. Instead, modern processors divide memory into fixed-size blocks called **pages**. On x86-64, the standard page size is

```
4096 bytes
```

or

```
4 KiB
```

Rather than translating every address separately, the processor translates one page at a time.

```
Virtual Memory

+----------+
|  Page 0  |
+----------+
|  Page 1  |
+----------+
|  Page 2  |
+----------+
```

Each page can be mapped independently. One page may contain executable code, another may contain writable data, and another may not exist at all. This flexibility makes paging remarkably powerful.

7.5 The Four-Level Page Table

The original JamesM tutorial used a two-level page-table hierarchy because it targeted 32-bit processors. Long mode introduces a more sophisticated design, and the x86-64 architecture divides address translation into four stages.

```
Virtual Address
      |
      v
PML4
      |
```

```
      v
PDPT
      |
      v
Page Directory
      |
      v
Page Table
      |
      v
Physical Frame
```

Each level contains 512 entries, and each entry occupies eight bytes. Together these structures allow the processor to describe enormous address spaces while allocating page tables only when needed.

Do not worry about memorizing every level immediately. Think of the hierarchy as a sequence of increasingly detailed maps. Each table answers one question — where should I look next? — and eventually the final table identifies the physical page.

7.6 Reading a Virtual Address

A sixty-four-bit virtual address is not treated as one large number. Instead, the processor divides it into several fields. Conceptually,

```
+------+-------+-------+-------+-------------+
|PML4 | PDPT  |  PD   |  PT   | Page Offset |
+------+-------+-------+-------+-------------+
```

The first portion selects an entry in the PML4, that entry identifies a PDPT, the PDPT selects a Page Directory, the Page Directory selects a Page Table, and the Page Table identifies the physical frame. Finally, the page offset identifies the exact byte within that frame.

Instead of searching one enormous table, the processor performs four small lookups, and this hierarchical organization saves large amounts of memory.

7.7 Walking Through `get_page()`

One of the most important functions in the kernel is

```
page_t *get_page(u64int address,
                 int make,
                 page_table_t *pml4);
```

Rather than reading the implementation immediately, begin by reading the interface. The first parameter is a virtual address, the second indicates whether missing page tables should be created automatically, and the third, `pml4`, is a pointer to the top-level page

table that defines the address space to search. The function returns a pointer to the page-table entry representing the requested address.

Notice what it does **not** return. It does not return the memory itself; it returns the data structure describing that memory. This distinction is subtle but important. The page-table entry contains metadata: whether the page exists, whether it is writable, whether it belongs to user mode, and which physical frame it references. Changing these fields changes how the processor interprets memory.

7.8 Creating Page Tables on Demand

Suppose a program accesses an address that belongs to an unused portion of memory. The corresponding page table may not yet exist. The function

```
get_page(address, 1, pml4);
```

creates the missing tables automatically. This strategy is called **lazy allocation**. Instead of constructing thousands of empty page tables during boot, the kernel creates them only when necessary, and the result is significant memory savings. This principle appears throughout operating systems: allocate resources only when they are needed.

7.9 Page Table Entries

Each page-table entry contains much more than an address. It also contains several control bits, and among the most important are

- Present
- Read/Write
- User/Supervisor
- Frame Number

These bits tell the processor how memory should be treated. A page marked “not present” causes a page fault, a page marked read-only rejects write operations, and a supervisor page becomes inaccessible from user mode. One sixty-four-bit value therefore describes both the location and the permissions of a page, so data and policy coexist inside the same structure.

7.10 The Memory Management Unit

Everything we have discussed so far is merely preparation. The actual translation is performed by specialized hardware called the **Memory Management Unit**, or MMU. Whenever the processor executes an instruction such as

```
*x = 5;
```

the MMU immediately begins translating the virtual address. The compiler is unaware of this process, the C program is unaware of this process, and the translation occurs

entirely in hardware.

From the programmer's perspective, memory appears ordinary; from the processor's perspective, every memory access requires multiple table lookups. This separation of responsibilities is one of the reasons virtual memory works so well: software constructs the page tables, and hardware performs the translation.

7.11 Page Faults

Sometimes the MMU cannot complete the translation. Perhaps the page is not present, perhaps the process lacks permission, or perhaps the address is completely invalid. Rather than guessing what to do, the processor generates an exception. This exception is called a **page fault**.

The processor transfers control to the kernel, and the kernel examines the fault. If the page simply has not been allocated yet, the kernel may create it; if the access is illegal, the kernel terminates the offending process.

Page faults are therefore not always errors. Many operating systems deliberately rely on them as part of normal memory management.

7.12 Reading the Paging Code

The paging subsystem contains many unfamiliar data structures, so do not attempt to understand everything at once. Begin with the high-level flow.

Receive a virtual address.

↓

Locate the appropriate page table.

↓

Create missing tables if requested.

↓

Return the page-table entry.

Once this overall process becomes clear, study the helper functions one by one. Understanding the algorithm first makes the implementation much easier to follow.

7.13 Common Mistakes

Students often confuse virtual addresses with physical addresses. Remember that a pointer almost always contains a virtual address, and the MMU performs the translation automatically.

Another common mistake is assuming that page tables contain memory. They do not. Page tables describe memory.

Students also forget that changing a page-table entry does not immediately affect the processor. The Translation Lookaside Buffer (TLB) may still contain the previous translation, and later chapters introduce instructions such as `invlpg` to invalidate stale entries.

Finally, avoid thinking of paging as merely an implementation detail. Virtual memory shapes the architecture of the entire operating system.

Practice Exercises

Exercise 1

Trace a call to `get_page()`, identifying every page-table level visited before the final entry is returned. Draw the sequence on paper.

Exercise 2

Locate every field inside the `page_t` structure. For each one, explain whether it represents:

- an address,
- a permission,
- processor state,
- or allocation metadata.

Exercise 3

Modify the paging code to print every virtual address it translates during boot. Observe which regions of memory are accessed most frequently.

Exercise 4

Trigger a deliberate page fault by accessing an unmapped address, then trace the resulting exception through the interrupt subsystem. Explain why the processor generated the fault.

Historical Perspective

Early computers accessed physical memory directly. As systems became larger and multitasking became common, this approach proved inadequate. The introduction of paging transformed operating system design by separating the addresses used by software from the addresses used by hardware.

Modern processors have extended this idea with larger address spaces, multiple page sizes, translation caches, nested paging for virtualization, and hardware support for memory protection. Despite these advances, the fundamental idea remains remarkably

simple: programs use virtual addresses, hardware translates them, and the operating system controls the translation.

Chapter Summary

Paging allows every process to believe it owns a private memory space while safely sharing the physical memory of the computer. In this chapter, we introduced virtual memory, examined the four-level page-table hierarchy used by x86-64 processors, followed the translation of a virtual address, and explored the role of the Memory Management Unit. We also learned how page faults arise and why they are an essential part of modern memory management rather than merely a sign of programming errors.

As you continue through the kernel, remember that page tables do not contain data. They describe how data should be interpreted by the processor. This distinction between objects and their descriptions appears repeatedly throughout systems programming.

In the next chapter, we build upon this foundation by building a heap from which dynamic memory is allocated..

Chapter 8 - Building a Heap

Learning Objectives

After completing this chapter, you should be able to

- explain why the placement allocator is insufficient for a long-running operating system,
- understand the purpose of a kernel heap,
- distinguish between allocation and memory management,
- explain how the heap's sorted hole index tracks unused memory,
- understand heap block metadata,
- trace a call to `kmalloc()` through the heap manager,
- understand fragmentation and why it occurs, and
- confidently read and modify the kernel heap implementation.

8.1 Growing Beyond the Placement Allocator

In the previous chapter, we learned how the placement allocator works. It is difficult to imagine a simpler allocator: the kernel remembers one address, every allocation returns that address, the address moves forward, and memory is never reused. For the earliest stages of boot, this approach is almost ideal, because it is fast, reliable, and contains very little code.

Unfortunately, it also contains a serious limitation. Memory never comes back. Suppose the kernel allocates one thousand objects during boot, and later eight hundred of those objects become unnecessary. The placement allocator cannot reuse the freed memory because it has no concept of "free," so the unused memory simply remains lost. Eventually the system exhausts available RAM.

A real operating system cannot function this way. Long-running systems continuously create and destroy objects: processes start and terminate, files open and close, and network packets arrive and disappear. Memory must be recycled, and this need gives rise to the **kernel heap**.

8.2 What Is a Heap?

The word *heap* is sometimes confusing because it has several meanings in computer science. In data structures, a heap is a specialized tree used to implement priority queues. In operating systems, the heap refers to a region of memory used for dynamic allocation. The kernel heap is simply an area where memory blocks can be allocated, released, and reused throughout the lifetime of the operating system, and unlike the placement allocator, the heap remembers which regions are available.

Imagine a bookshelf. The placement allocator behaves like someone who always places the next book on the far right; once a space has been used, it is never used again. A heap manager behaves differently. Whenever a book is removed, the empty space becomes available for another book, and the bookshelf continues serving new requests without growing indefinitely. The heap performs the same task for memory.

8.3 Allocation Is Only Half the Problem

Many beginning programmers think memory allocation consists of answering one question.

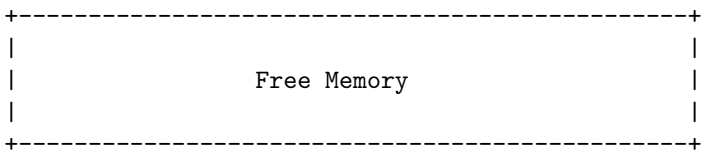
> Where can I find enough free bytes?

That question is only the beginning. The allocator must also answer several others. When should memory be returned? How can free blocks be found efficiently? What happens if a requested block is larger than every available space? Can neighboring free blocks be merged? How should memory remain aligned? How much bookkeeping information is necessary?

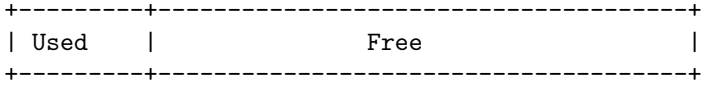
These questions transform allocation from a simple arithmetic exercise into a complete management problem. The kernel heap is therefore not merely a function; it is a subsystem.

8.4 The Idea of Free Blocks

Imagine that the heap initially consists of one large unused region.



The kernel requests 256 bytes, and the allocator divides the region.



Later, another request consumes part of the remaining space.

```
+---------+-----------+--------------------------+
| Used    |   Used    |          Free            |
+---------+-----------+--------------------------+
```

Eventually one block is released.

```
+---------+-----------+--------------------------+
| Used    |   Free    |          Free            |
+---------+-----------+--------------------------+
```

Notice something important. The heap is no longer divided into one large region. Instead, it contains multiple blocks — some allocated, some available — and the allocator must remember which is which.

8.5 Metadata: Memory About Memory

The heap cannot manage blocks unless it stores information describing each one. This information is called **metadata**. In this kernel, every block — whether allocated or free — carries a small **header** at its front and a matching **footer** at its end. The caller's data sits in between.

```
+--------------------+
| Header             |   magic, is_hole, size
+--------------------+
| User Data          |
|                    |
+--------------------+
| Footer             |   magic, pointer back to header
+--------------------+
```

Open `kheap.h` and you will find both structures.

```
typedef struct
{
    u32int magic;    // Error checking and identification.
    u8int  is_hole;  // 1 if this is a hole, 0 if it is an allocated block.
    u64int size;     // Size of the whole block, header and footer included.
} header_t;

typedef struct
{
    u32int magic;
    header_t *header; // Points back to this block's header.
} footer_t;
```

Three ideas deserve a pause. The `magic` number is a fixed value stamped into every header and footer; if the allocator ever reads a block whose magic is wrong, it knows

the heap has been corrupted. The `is_hole` flag records whether the block is free. And the footer's back-pointer is what lets the allocator recover a block's header starting from its *end* — the key to merging neighbors, as §8.9 will show.

When the programmer requests

```
kmalloc(128);
```

the allocator actually reserves more than 128 bytes: the header and footer are overhead the caller never sees. Most programmers never notice this hidden bookkeeping, but heap managers depend on it.

8.6 Tracking the Holes

A free block in this heap is called a **hole**, and the allocator needs a fast way to find one. A classic solution is the **free list**: a linked list that threads every free block together through pointers stored inside the blocks. It is a sound design, and you will meet it in other kernels — but it is *not* what this kernel does.

Instead, the heap keeps a separate **index**: an array of pointers to every hole, held apart from the blocks themselves and kept **sorted by size**, smallest first.

```
index (sorted by size)

+------+------+------+------+
| ptr  | ptr  | ptr  | ptr  |
+--|---+--|---+--|---+--|---+
   v      v      v      v
 128 B  512 B  4 KiB  64 KiB    <- holes, scattered through the heap
```

The index is the `ordered_array_t` declared inside `heap_t`.

```
typedef struct
{
    ordered_array_t index;   // Pointers to every hole, sorted by size.
    u64int start_address;
    u64int end_address;
    u64int max_address;
    // ...
} heap_t;
```

Only *holes* appear in the index; allocated blocks do not. Freeing a block inserts it; handing a hole out removes it. The array is insertion-sorted, so it is in sorted order between every call — and that single guarantee is what makes allocation cheap, as the next section shows.

Keeping the directory of free space separate from the blocks, rather than threading a list through them, has a practical payoff: the index stays compact and ordered, and the allocator never walks through allocated memory to find a hole.

8.7 Finding a Suitable Block

Suppose three free blocks exist.

```
128 bytes

512 bytes

4096 bytes
```

The kernel requests 200 bytes. Which block should be used? Several strategies exist. The **first-fit** algorithm selects the first block large enough to satisfy the request, the **best-fit** algorithm searches for the smallest block capable of holding the allocation, and the **worst-fit** algorithm selects the largest available block.

Each strategy has advantages and disadvantages. This kernel uses **best-fit**, and the sorted index makes it nearly free. Because the holes are already ordered by size, the allocator walks the index from the smallest hole upward and stops at the first one large enough; that first hole large enough is, by construction, the *smallest* one that fits. The function that does this is named, aptly, `find_smallest_hole`. Production kernels often prefer strategies that trade a little fragmentation for speed, but best-fit over a sorted index is both easy to read and easy to reason about.

8.8 Splitting Blocks

Suppose the allocator chooses a 4096-byte block to satisfy a request for only 256 bytes. Discarding the remaining memory would be wasteful, so instead the allocator divides the block.

```
Before

+--------------------------------+
|          4096 Bytes            |
+--------------------------------+

After

+------+-------------------------+
| 256  |         Free            |
+------+-------------------------+
```

The first portion satisfies the request, and the remainder becomes a new, smaller hole that the allocator inserts back into the index. This process is called **splitting**. Nearly every heap manager performs it, because without splitting, memory would disappear surprisingly quickly. One guard is worth noting: if the leftover would be too small to hold even a header and footer, the allocator does not create a stub hole — it simply lets the allocation absorb the extra bytes.

8.9 Merging Blocks

Splitting introduces a new problem. Imagine the following sequence: allocate, allocate, free, free. The heap becomes

```
+------+------+
|Free  |Free  |
+------+------+
```

These blocks are adjacent, and treating them as separate objects wastes opportunities for larger allocations. The allocator therefore combines neighboring free blocks. This process is called **coalescing**.

```
Before
+------+------+
|Free  |Free  |
+------+------+

After

+-------------+
|    Free     |
+-------------+
```

Because every block lies head to tail in memory, the allocator finds a freed block's neighbors by arithmetic rather than by following a list. The footer just *before* this block's header describes the neighbor on the left; the header just *after* this block's footer describes the neighbor on the right. If either neighbor still has an intact `magic` and its `is_hole` flag set, the two are fused into a single larger hole. The code names these steps *unify left* and *unify right*, and they are the reason every block carries a footer pointing back to its own header.

Coalescing reduces fragmentation and helps preserve large contiguous regions of memory. Good heap managers split when allocating and merge when freeing, and the two operations complement one another.

8.10 Fragmentation

Fragmentation occurs whenever free memory becomes divided into many small pieces. Consider the following heap.

```
+----+----+----+----+----+
|Used|Free|Used|Free|Used|
+----+----+----+----+----+
```

The heap contains free memory, yet no large contiguous region exists. A request for a large block may fail even though the total amount of free memory appears sufficient. This phenomenon surprises many students, because memory capacity and usable mem-

ory are not always the same thing. Reducing fragmentation is one of the principal goals of heap design.

8.11 Walking Through `kmalloc()`

Now open `kheap.c`. Instead of reading every line, identify the overall sequence.

Receive a size request.

↓

Scan the hole index for the smallest hole that fits.

↓

If none fits, expand the heap — ask paging for more pages — and try again.

↓

Remove the chosen hole from the index.

↓

Split it if it is larger than needed, returning the leftover as a new hole.

↓

Mark the block allocated, and write its header and footer.

↓

Return the address just past the header.

Notice how much more work occurs compared with the placement allocator. The interface remains almost identical, but the implementation has become substantially more sophisticated. This illustrates an important lesson: simple interfaces often hide complicated implementations.

8.12 Heap and Paging

The heap manager does not create memory from nothing. Eventually it reaches the end of the currently mapped heap, and when that happens, it requests additional pages from the paging subsystem. The relationship therefore looks like this.

```
Program

      |
      v

kmalloc()

      |
```

```
      v

Heap Manager

      |
      v

Paging System

      |
      v

Physical Memory
```

This is where the virtual addresses of Chapter 3 §3.13 become concrete. The heap does not sit in the identity-mapped low memory the kernel executes from; it is placed high in the address space, at `KHEAP_START` (`0xFFFF800000000000`), and the pages that back it are mapped there on demand. A heap pointer is therefore a genuinely virtual address — which is exactly why `kmalloc_ap` must consult the page tables to report the physical frame behind an allocation (Chapter 6 §6.6).

Each layer solves a different problem. The heap manages variable-sized objects, paging manages fixed-sized pages, and physical memory stores the actual bytes. Understanding these layers helps prevent confusion, because each subsystem builds upon the one beneath it.

8.13 Reading `kheap.c`

Do not begin by reading helper functions. Instead, identify the major data structures. Where does the hole index live, and how is it kept sorted? What sits in a header, and what sits in a footer? Which function allocates (`alloc`), which frees (`free`), and which grow and shrink the mapped region (`expand` and `contract`)?

Once these relationships become clear, the implementation becomes much easier to understand. Good programmers rarely read large source files from top to bottom. They first build a mental map of the subsystem, and only then do they study individual functions.

8.14 Common Mistakes

Many beginning programmers assume that heap allocation is instantaneous. It is not, because searching free blocks and updating metadata requires time.

Another common mistake is forgetting that freeing memory does not erase its contents; the block simply becomes available for reuse.

Students also confuse memory leaks with fragmentation. A memory leak occurs when allocated memory is never released, while fragmentation occurs when free memory becomes divided into unusable pieces.

Finally, avoid assuming that all heap managers behave identically. Different operating systems employ different algorithms depending on their performance goals.

Practice Exercises

Exercise 1

Read the implementation of `kmalloc()` and draw a flowchart of every major decision it makes during allocation. Ignore the small details; focus on the overall algorithm.

Exercise 2

Modify the allocator to print the address and size of every allocated block, then boot the kernel and observe how the heap evolves over time.

Exercise 3

Write a small test that repeatedly allocates and frees blocks of different sizes. Predict where fragmentation will occur, then verify your prediction by examining the heap metadata.

Exercise 4

Trace one allocation from `kmalloc()` down through the paging subsystem, identifying where new physical pages are requested. Explain why the heap cannot function without paging.

Historical Perspective

The earliest operating systems often relied on extremely simple allocation strategies because available memory was small and workloads were predictable. As systems became interactive and multitasking matured, kernels required allocators capable of handling thousands or even millions of dynamic allocations while minimizing fragmentation and synchronization overhead.

Modern operating systems employ specialized allocators such as slab allocators, SLUB, SLOB, buddy allocators, and per-CPU caches. Despite these sophisticated designs, every allocator still solves the same fundamental problem explored in this chapter: finding, tracking, splitting, merging, and reusing blocks of memory efficiently. Understanding these simple mechanisms makes advanced allocators much easier to appreciate.

Chapter Summary

The placement allocator introduced in the previous chapter was sufficient for bootstrapping the kernel, but it could never support a long-running operating system. A practical kernel must reuse memory, manage free space, and cope with fragmentation, and these requirements give rise to the kernel heap.

In this chapter, we introduced heap management as a complete subsystem rather than a single allocation function. We explored metadata, the sorted hole index, allocation strategies, block splitting, coalescing, and fragmentation, and we also examined how the heap cooperates with the paging subsystem to obtain additional memory when necessary.

As the operating system continues to grow, memory management becomes increasingly intertwined with every other subsystem. Process management, filesystems, networking, and device drivers all depend upon reliable dynamic allocation.

In the next chapter, we move from managing memory to managing execution. We will study multitasking, context switching, and how the operating system allows multiple processes to share a single processor while each believes it is running alone.

Chapter 9 - Bit Manipulation and Flags

Learning Objectives

After completing this chapter, you should be able to

- understand why kernels manipulate individual bits instead of entire integers,
- read and write hexadecimal values confidently,
- use C bitwise operators correctly,
- explain masks, flags, and bit fields,
- understand how page-table entries and processor registers encode information,
- recognize common bit manipulation idioms used throughout the kernel,
- avoid common mistakes involving shifts and masks, and
- confidently read low-level systems code that manipulates hardware registers.

9.1 Thinking Smaller Than an Integer

Most programming courses teach integers as whole numbers. You add them, subtract them, multiply them, and compare them. Operating systems use integers differently, because a sixty-four-bit value is often not a number at all. Instead, it is a collection of individual bits, and each bit carries its own meaning.

Consider a page-table entry. Part of the value stores the physical frame address, several bits describe permissions, one bit indicates whether the page exists, another determines whether user programs may access it, and another controls caching. Although stored in one integer, these pieces represent completely different ideas. Kernel programmers therefore spend much of their time manipulating bits rather than numbers.

9.2 A Bit Is Information

A bit has only two possible values: zero and one. Despite this simplicity, bits combine to represent remarkably complex information. Eight bits form one byte,

```
01001101
```

sixteen bits form a word, thirty-two bits form a double word, and sixty-four bits form a quad word.

Processors interpret these bits differently depending on context. The same pattern could represent

- an integer,
- a pointer,
- an instruction,
- a floating-point value,
- or a collection of flags.

The bits never change; only their interpretation changes. This idea should already sound familiar, because it is exactly the lesson we learned when studying pointers and memory.

9.3 Why Kernels Love Bits

Suppose we need to describe whether a page

- exists,
- is writable,
- belongs to user mode,
- has been accessed,
- has been modified.

One approach is

```
struct page {
    int present;
    int writable;
    int user;
    int accessed;
    int dirty;
};
```

This works, but it is also wasteful. Five integers occupy twenty bytes on most systems, whereas the processor stores the same information inside five bits.

```
00011001
```

Every bit has a predefined meaning. This representation is compact, and more importantly, it matches the processor documentation. When reading architecture manuals, you will frequently encounter diagrams such as

```
63                    12 11   9 8 7 6 5 4 3 2 1 0
+----------------------+-----+-+-+-+-+-+-+-+-+-+-+
| Physical Frame       | ... |D|A|P|U|W|P|
+----------------------+-----+-+-+-+-+-+-+-+-+-+-+
```

The operating system must manipulate these individual bits directly.

9.4 Binary and Hexadecimal

Kernel programmers rarely write long binary numbers. Instead, they use hexadecimal. Consider

```
11111111
```

Writing binary quickly becomes tedious. Instead,

```
0xFF
```

represents exactly the same bits. Every hexadecimal digit corresponds to four binary bits.

```
Hex    Binary

0      0000
1      0001
2      0010
...
A      1010
F      1111
```

Because one hexadecimal digit equals four bits, it becomes easy to interpret processor registers visually, and with practice, experienced systems programmers often recognize common hexadecimal values immediately.

9.5 The Bitwise Operators

C provides several operators that work directly on bits.

AND

```
a & b
```

A bit remains one only if both operands contain one.

```
1101
1011
----
1001
```

AND is commonly used to clear unwanted bits.

OR

```
a | b
```

A bit becomes one if either operand contains one.

```
1101
1011
----
1111
```

OR is commonly used to enable flags.

XOR

```
a ^ b
```

Bits differ.

```
1101
1011
----
0110
```

XOR is useful for toggling bits and implementing certain algorithms.

NOT

```
~a
```

Every bit is inverted.

```
1010

becomes

0101
```

9.6 Shifting Bits

Sometimes we do not want to change bits; we want to move them. A left shift

```
value << 3
```

moves every bit three positions left, and multiplication by powers of two often emerges naturally. A right shift

```
value >> 2
```

moves bits toward the least significant end.

Kernel programmers constantly use shifts to position values inside hardware-defined fields. Suppose bits 12 through 51 contain a physical address; the kernel shifts the address into position before combining it with control bits.

9.7 Masks

A **mask** isolates specific bits. Suppose we wish to extract only the lowest eight bits.

```
value & 0xFF
```

Every higher bit becomes zero, and only the desired portion remains.

Masks also remove flags. Suppose a page-table entry stores both an address and several permission bits.

```
Address | Flags
```

The kernel masks away the flags.

```
entry & 0x000FFFFFFFFFF000ULL
```

The remaining value represents only the physical frame. Without masks, extracting processor-defined fields would be extremely difficult.

9.8 Setting and Clearing Flags

Flags are usually manipulated through simple idioms. To enable a flag,

```
flags |= FLAG_PRESENT;
```

to clear a flag,

```
flags &= ~FLAG_PRESENT;
```

to toggle a flag,

```
flags ^= FLAG_PRESENT;
```

and to test a flag,

```
if (flags & FLAG_PRESENT)
```

These four patterns appear throughout operating systems, and after enough practice, you begin recognizing them instantly.

9.9 Reading Processor Registers

Many processor registers consist almost entirely of flags. Consider CR0. One bit enables protected mode, another enables paging, and others control caching, write protection, and floating-point behavior.

The kernel rarely modifies the entire register. Instead, it changes one bit while preserving every other setting. For example,

```
cr0 |= (1 << 31);
```

sets the paging bit without disturbing the remaining configuration. This approach appears repeatedly throughout low-level programming.

9.10 Page Table Entries

Open the paging subsystem and notice how page-table entries combine addresses and flags inside one sixty-four-bit value. Conceptually,

```
63                        12 11      0
+-------------------------+----------+
| Physical Frame Address | Flags    |
+-------------------------+----------+
```

The address occupies most of the entry, and the lowest bits describe permissions. When creating a mapping, the kernel first positions the frame address, then enables the desired flags, and finally writes the completed value into the page table. Understanding this sequence makes paging code much easier to read.

9.11 Naming the Bits

Magic numbers quickly become unreadable. Compare

```
entry |= 0x4;
```

with

```
entry |= PAGE_USER;    // one common style
```

The second version communicates intent immediately. There are two widespread ways to give a bit a name. Many kernels define a symbolic constant for each flag — `PAGE_PRESENT`, `PAGE_RW`, `PAGE_USER` — and combine them with the bitwise operators from this chapter. The other approach lets the compiler track the bits for you, and it is the one this tutorial uses.

Open `paging.h` and look at the page-table entry. Rather than a bare `u64int` masked by hand, it is a **bitfield structure**.

```
typedef struct page
{
    u64int present    : 1;   // Page present in memory
    u64int rw         : 1;   // Read-only if clear, readwrite if set
    u64int user       : 1;   // Supervisor level only if clear
    // ...
    u64int frame      : 40;  // Physical frame number
    // ...
} page_t;
```

Each `: 1` tells the compiler that the field occupies exactly one bit, in the order written. Setting a flag then reads like ordinary field assignment,

```
page->present = 1;
page->rw      = 1;
page->user    = 0;
```

and the compiler emits the same shifting and masking you would otherwise write by hand. The layout still has to match the processor's definition bit for bit — that is why the fields appear in a precise order and why the structure is exactly sixty-four bits wide — but the names now live in the type itself.

Both styles share the same goal: a reader should never have to remember that `0x4` means "user." Good kernel programmers rarely memorize numeric flag values; they remember what the flags mean, and they let named bitfields or named constants carry that meaning in the code.

9.12 Reading Bitwise Code

Many students initially find bitwise expressions intimidating. Consider

```
entry = (frame << 12) | PAGE_PRESENT | PAGE_RW;
```

Do not read it from left to right. Instead, break it apart.

Shift the frame address.

↓

Move it into the address field.

↓

Enable the present bit.

↓

Enable write permission.

↓

Store the completed value.

Complex expressions become much easier once each operation is viewed independently.

9.13 Common Mistakes

Beginning programmers often confuse logical operators with bitwise operators.

`&&`

is not the same as

`&`

Likewise,

```
||
```

is not the same as

```
|
```

Another common mistake is forgetting parentheses. Shift operators have lower precedence than many arithmetic operators, so explicit parentheses improve both correctness and readability.

Students also shift signed integers, producing implementation-dependent results. Kernel code almost always uses unsigned integers for bit manipulation.

Finally, avoid writing unexplained hexadecimal constants, because names are easier to read than numbers.

Practice Exercises

Exercise 1

Locate every occurrence of `<<`, `>>`, `&`, `|`, `^`, and `~` in the repository, and for each one explain what information is being manipulated.

Exercise 2

Choose a page-table entry and draw every field. Indicate which bits represent the physical frame and which represent permissions.

Exercise 3

Rewrite several hexadecimal constants using symbolic names, then compare the readability of the resulting code.

Exercise 4

Write four helper functions — `set_flag()`, `clear_flag()`, `toggle_flag()`, and `test_flag()` — implementing each with the appropriate bitwise operator.

Historical Perspective

Bit manipulation has been part of systems programming since the earliest computers. Early processors offered only a handful of registers, making efficient use of every bit essential, and as architectures evolved, hardware designers continued encoding processor state into individual bits because compact representations simplified circuitry and reduced memory requirements.

Today, virtually every hardware specification relies on bit fields. Processor registers, network packet headers, executable file formats, page tables, filesystem metadata, and

communication protocols all describe information one bit at a time. Learning to manipulate bits is therefore not a specialized skill reserved for operating system developers. It is one of the fundamental literacies of computer systems.

Chapter Summary

Modern operating systems treat integers as containers for structured information rather than merely numerical values. We introduced the bitwise operators provided by C, explored hexadecimal notation, learned how masks isolate fields, and examined the common idioms used to set, clear, toggle, and test individual flags. We also saw how page-table entries and processor registers combine addresses and permissions within a single sixty-four-bit value.

As you continue reading the kernel, you will notice that bit manipulation appears almost everywhere. The processor communicates through bits, hardware manuals describe bits, and operating systems ultimately control hardware by arranging bits into precisely the patterns the architecture expects.

In the next chapter, we will connect everything we have learned so far by examining how C and assembly language cooperate. We will study calling conventions, stack layouts, the Application Binary Interface (ABI), and context switching—the machinery that allows high-level C code and low-level processor instructions to work together as a single operating system.

Chapter 10 - Linking C and Assembly

Learning Objectives

After completing this chapter, you should be able to

- explain why operating systems combine C and assembly language,
- understand the purpose of the Application Binary Interface (ABI),
- describe the x86-64 calling convention,
- explain how function calls use the stack,
- understand stack frames and register preservation,
- trace the transition from assembly into C and back again,
- explain how context switching preserves processor state, and
- confidently read code that combines C and assembly.

10.1 Why Kernels Need Two Languages

Throughout this book we have written almost everything in C. C provides structures, functions, pointers, type checking, and readable source code, and these features make it an excellent language for operating systems. Yet there is a problem: the processor does not understand C. The processor understands machine instructions, and assembly language is simply a human-readable representation of those instructions.

Some processor operations cannot be expressed in standard C. Loading the Global Descriptor Table, loading the Interrupt Descriptor Table, reading control registers, changing privilege levels, executing `iretq`, switching page tables, and entering long mode all require assembly language.

A kernel therefore speaks two languages. Most of the operating system is written in C because C is easier for humans to understand, while the small parts that communicate directly with the processor are written in assembly. The two languages cooperate continuously.

10.2 The Invisible Contract

Suppose an assembly routine calls a C function. How does the function know where its arguments are? How does it know which register contains the return value? How does it know which registers may be modified?

The answer is that both languages obey a shared set of rules. These rules form the **Application Binary Interface**, usually abbreviated **ABI**. The ABI is not a programming language; it is a contract. Every compiler follows it, every assembler follows it, and every operating system depends upon it. Without a common convention, independently compiled code could never communicate correctly.

10.3 What Happens During a Function Call?

Consider an ordinary C function.

```
int add(int a, int b)
{
    return a + b;
}
```

When another function calls

```
result = add(3, 5);
```

many things happen automatically. Arguments are placed into registers, the return address is saved, execution jumps to the new function, the function performs its work, the return value is placed into another register, and execution resumes where it left off.

None of these steps are visible in C, because the compiler generates them. Assembly programmers perform the same steps manually, so understanding these hidden operations makes the relationship between C and assembly much clearer.

10.4 The x86-64 Calling Convention

The System V AMD64 ABI specifies where arguments are placed. The first six integer or pointer arguments occupy registers.

```
Argument    Register

1           RDI
2           RSI
3           RDX
4           RCX
5           R8
6           R9
```

Additional arguments are placed on the stack, and the return value appears in

```
RAX
```

Suppose the kernel executes

```
foo(a, b, c);
```

Conceptually, the compiler performs

```
RDI = a
RSI = b
RDX = c
CALL foo
```

The function immediately finds its arguments where the ABI promises they will be. The caller and callee never need to negotiate; both simply obey the rules.

10.5 The Stack Frame

Every function receives its own workspace, and this workspace is called the **stack frame**. Conceptually,

```
High Addresses

+--------------------+
| Previous Frames    |
+--------------------+
| Return Address     |
+--------------------+
| Saved RBP          |
+--------------------+
| Local Variables    |
+--------------------+

Low Addresses
```

When a function begins, it usually saves the previous frame pointer and then reserves space for local variables. When the function finishes, the process reverses, and the stack returns to its previous state. Every function call creates one stack frame, and every return removes one. This organization allows recursive functions and nested calls to work naturally.

10.6 Caller-Saved and Callee-Saved Registers

Registers are valuable resources, and a function cannot simply overwrite every register it uses. Some registers belong temporarily to the caller, while others must be preserved by the callee. The ABI divides responsibility.

For example,

```
Caller-Saved

RAX
RCX
RDX
RSI
RDI
R8-R11
```

These registers may be modified freely, and if the caller wishes to preserve their values, the caller saves them. Other registers belong to the callee.

```
Callee-Saved

RBX
RBP
R12-R15
```

If the function changes these registers, it must restore them before returning. This division minimizes unnecessary work while ensuring correctness. Every compiler depends upon these rules, and assembly code must obey them as well.

10.7 Crossing the Language Boundary

Interrupt handling provides one of the best examples of cooperation between assembly and C. The processor transfers control to an assembly interrupt stub, and the stub saves registers.

```
push rax
push rbx
...
```

Once the processor state has been preserved, the stub calls

```
isr_handler(registers_t *regs);
```

The C function performs the high-level logic: it determines which interrupt occurred and invokes the appropriate handler. Eventually control returns to the assembly stub, the stub restores registers, and finally

```
iretq
```

returns to the interrupted program. Notice how each language performs the task for which it is best suited. Assembly preserves processor state, and C makes decisions.

10.8 Why C Cannot Execute `iretq`

Students sometimes ask why the interrupt handler cannot simply execute

`iretq`

directly from C. The answer is that `iretq` expects a very specific stack layout. The processor restores registers from the interrupt frame, and one missing value causes the return to fail.

Standard C has no concept of this processor-specific stack structure, whereas assembly provides precise control over every instruction and every byte on the stack. For this reason, entering and leaving interrupt handlers always involves assembly.

10.9 Reading Control Registers

Control registers provide another example. Suppose the kernel wishes to read CR3. In assembly,

```
mov rax, cr3
```

retrieves the current page-table base address. Standard C has no equivalent statement, so instead, kernels embed assembly inside C.

```
static inline u64int read_cr3(void)
{
    u64int value;

    asm volatile(
        "mov %%cr3, %0"
        : "=r"(value)
    );

    return value;
}
```

The function appears ordinary to its callers, but internally it performs a processor-specific operation. Two things are worth noting. First, `read_cr3` is an illustration of the technique, not a function you will find in the repository: this kernel writes CR3 to install a page table but has no need to read it back. Second, notice the two assembly dialects. The standalone `mov rax, cr3` is Intel syntax — the form the `.s` files use — while the inline template `"mov %%cr3, %0"` is AT&T syntax, which GCC expects inside C, with the destination written last. The repository mixes both, Intel in its `.s` files and AT&T in its inline assembly, so it is worth being able to read each.

10.10 What Is a Context Switch?

So far we have discussed ordinary function calls. Context switching is much more dramatic. A function call changes execution within one program, while a context switch changes execution between different programs.

Imagine two processes.

```
Process A

Registers
Stack
Instruction Pointer

Process B

Registers
Stack
Instruction Pointer
```

When the scheduler decides to run Process B, the kernel must preserve the complete processor state of Process A, and then it restores the state belonging to Process B. The processor resumes execution as though Process B had never stopped. The illusion is complete: each process believes it owns the processor.

10.11 Saving Processor State

During a context switch, the kernel preserves

- general-purpose registers,
- the stack pointer,
- the instruction pointer,
- processor flags,
- page-table information,
- floating-point state when necessary.

Conceptually,

```
Running Process

        |
        v

Save Registers

        |
        v

Select Next Process

        |
        v

Restore Registers
```

```
        |
        v

Resume Execution
```

Notice how similar this sequence is to interrupt handling. Both involve preserving processor state, but the difference is what happens afterward. Interrupt handlers eventually return to the same process, whereas schedulers often return to an entirely different one.

10.12 Reading Assembly Without Fear

Many students believe they must become expert assembly programmers before reading kernel code. Fortunately, this is unnecessary, because most operating system assembly follows predictable patterns: save registers, load registers, move values, call C functions, and return.

Learn to recognize these recurring patterns rather than memorizing every instruction. Whenever you encounter an unfamiliar assembly routine, ask what information enters, what information leaves, and what processor state changes. These questions reveal the routine's purpose even before every instruction is understood.

10.13 Reading the Repository

Locate the assembly source files. Begin by identifying every function visible to C, then find the corresponding declarations inside the header files, and next determine where each function is called. Only afterward should you examine individual instructions.

Many beginning programmers do the opposite. They start with the assembly itself, and the result is often confusion. Understanding the interface first makes the implementation much easier to follow.

10.14 Common Mistakes

Beginning programmers often assume that C functions have complete control over the processor. They do not. The compiler generates code that obeys the ABI, and assembly must obey the same rules.

Another common mistake is modifying callee-saved registers without restoring them. Programs may appear to work before failing unpredictably.

Students also forget that interrupt handlers and ordinary function calls return differently. Normal functions execute `ret`, while interrupt handlers execute `iretq`, and the instructions are not interchangeable.

Finally, avoid treating assembly language as mysterious. Most kernel assembly consists of short, carefully written routines that establish the environment in which C can safely execute.

Practice Exercises

Exercise 1

Choose an assembly routine in the repository and identify:

- which registers it modifies,
- whether those registers are caller-saved or callee-saved,
- where control transfers into C.

Exercise 2

Compile the kernel and disassemble it with:

```
objdump -d kernel.elf
```

Locate one C function, compare the generated assembly with the original source, and identify the function prologue and epilogue.

Exercise 3

Trace one interrupt's complete execution path, from the processor to the final `iretq`. Identify every transition between assembly and C.

Exercise 4

Draw the stack before and after an ordinary function call, then repeat the exercise for an interrupt. Explain why interrupt handling requires additional saved state.

Historical Perspective

Early operating systems were written almost entirely in assembly language because processors offered little support for high-level languages. As C matured during the 1970s, developers realized that most kernel logic could be expressed more clearly and maintained more easily in C while retaining the performance and control required for systems programming.

This hybrid approach remains the dominant design today. The Linux kernel, BSD family, Windows kernel, and most embedded operating systems are written primarily in C, with carefully isolated assembly routines responsible for bootstrapping, interrupt entry, context switching, processor initialization, and other hardware-specific operations. The boundary between C and assembly is therefore one of the most stable interfaces in operating system design.

Chapter Summary

An operating system cannot be written entirely in C because some processor operations are accessible only through assembly language. At the same time, writing an entire kernel in assembly would make the system unnecessarily difficult to develop and

maintain. Modern kernels therefore combine the strengths of both languages through a well-defined Application Binary Interface.

In this chapter, we examined the x86-64 calling convention, stack frames, register preservation, the transition between assembly and C, and the mechanics of context switching. We also saw how interrupt handlers rely on assembly to preserve processor state before transferring control to C. Understanding these conventions makes assembly routines far less intimidating and reveals how the processor, compiler, and operating system cooperate to execute every function call.

In the next chapter, we step away from processor mechanics and examine how a large kernel is organized. We will study header files, translation units, the build system, the linker, and the modular design practices that allow thousands of source files to become a single operating system.

Chapter 11 - Building Reusable Kernel Code

Learning Objectives

After completing this chapter, you should be able to

- explain why large programs are divided into multiple source files,
- distinguish between header files and implementation files,
- understand declarations, definitions, and external linkage,
- explain the purpose of include guards,
- understand how the C preprocessor participates in compilation,
- follow the compilation and linking process used by the kernel,
- understand the role of the Makefile and linker script, and
- confidently navigate the organization of the operating system source tree.

11.1 Why One File Is Never Enough

The first C programs most students write fit comfortably into a single file. Everything is visible at once, functions call one another directly, and global variables are easy to find. As programs grow, this approach becomes impossible.

Imagine placing the entire operating system into one source file, containing interrupt handling, paging, memory allocation, scheduling, device drivers, filesystems, and networking. Thousands of functions would occupy a single document, finding one routine would become frustrating, and changing one subsystem would risk breaking another.

Large software therefore divides responsibilities into modules. Each module solves one problem, each exposes a small public interface, and everything else remains private. This principle is not unique to operating systems; it appears throughout software engineering. The operating system simply provides a particularly good example.

11.2 Source Files and Header Files

Open almost any directory in the repository, and you will usually find two kinds of files.

`paging.c`

`paging.h`

The `.c` file contains the implementation, while the `.h` file describes the interface. Think of the header as a promise: it tells other source files what services are available, and the implementation explains how those services work.

For example,

```
void initialise_paging(void);
```

appears in the header, while its complete implementation appears elsewhere. Programs that include the header do not need to understand the implementation; they simply call the function. This separation keeps modules independent.

11.3 Declarations Versus Definitions

Beginning programmers often confuse declarations with definitions. A declaration introduces something, while a definition creates it. Consider

```
extern u64int placement_address;
```

This statement tells the compiler that the variable exists. It does **not** allocate storage. Some other source file contains

```
u64int placement_address = 0x100000;
```

This statement defines the variable, so storage is allocated and initialization occurs. Only one definition should exist, but many declarations may exist. Remember the distinction: declarations describe, definitions create.

11.4 Sharing Functions Across Files

Suppose `paging.c` contains

```
void initialise_paging(void)
{
    ...
}
```

The scheduler also needs this function. Instead of copying the implementation, the scheduler includes

```
#include "paging.h"
```

The compiler now knows the function's interface, and during linking, the actual implementation is connected automatically. This design allows many modules to share the same functionality without duplication, and reuse is one of the central goals of modular programming.

11.5 The Role of the Preprocessor

Compilation begins before the compiler even examines the C language. The first stage is the **preprocessor**, and its job is simple: copy included files, expand macros, evaluate conditional compilation directives, and remove comments.

Suppose the compiler encounters

```
#include "common.h"
```

The preprocessor literally inserts the contents of `common.h` into the source file. The compiler never sees the `#include` directive; it sees one large source file produced by the preprocessor. This explains why syntax errors inside header files often appear in unrelated source files, because the compiler processes the expanded result, not the original files.

11.6 Include Guards

Suppose two headers both include

`common.h`

Without protection, the compiler would process the file twice, duplicate definitions would appear, and compilation would fail. Include guards prevent this problem.

```
#ifndef COMMON_H
#define COMMON_H

...

#endif
```

The first inclusion defines the symbol, and subsequent inclusions detect that it already exists, so the header is skipped. Although simple, include guards are among the most important conventions in C programming, and every professional C programmer recognizes them immediately.

11.7 Static and External Functions

Not every function should be visible everywhere. Some helper functions belong only inside one source file. The keyword

`static`

limits visibility.

```
static void helper(void)
{
    ...
}
```

No other source file can call this function. Removing unnecessary visibility improves modularity, and it also allows the compiler to perform additional optimizations.

Functions intended for public use omit `static`. Their declarations appear in header files, and their implementations remain inside the corresponding source file. Good software exposes only what other modules actually need.

11.8 Following the Build Process

Building the kernel involves several distinct stages.

```
Source Files

        |

Preprocessor

        |

Compiler

        |

Assembler

        |

Object Files

        |

Linker

        |

Kernel Image
```

Each source file becomes an object file, object files remain independent, and the linker later combines them into one executable kernel. Understanding this pipeline explains many compiler and linker errors, because a compiler error usually indicates incorrect C

code, while a linker error usually indicates missing or duplicate definitions. Knowing where an error originates often reveals how to fix it.

11.9 The Makefile

Compiling every source file manually would be tedious. The Makefile automates the process, so instead of typing dozens of compiler commands,

```
make
```

executes them automatically. The Makefile specifies

- compiler options,
- assembler options,
- linker options,
- source files,
- output names,
- dependencies.

Whenever one source file changes, only the necessary files are rebuilt, which dramatically reduces compilation time. The Makefile is therefore not merely a convenience; it describes how the entire project is constructed.

11.10 Why Compiler Flags Matter

Open the Makefile and notice options such as

```
-ffreestanding
```

```
-nostdlib
```

```
-mno-red-zone
```

These are not arbitrary; each solves a specific kernel problem. `-ffreestanding` tells the compiler that no hosted operating system exists, `-nostdlib` prevents linking against the standard C library, and `-mno-red-zone` disables a compiler optimization that interferes with interrupt handling. Every compiler option reflects an engineering decision, and understanding these options is part of understanding the operating system itself.

11.11 The Linker Script

The compiler translates C into object code, but the linker decides where everything belongs in memory. The linker script describes this layout. Conceptually,

```
.multiboot
```

```
↓
```

```
.text

↓

.rodata

↓

.data

↓

.bss
```

Each section serves a different purpose. The multiboot header (`.multiboot`) comes first, because the bootloader must find it within the opening bytes of the image; executable instructions belong in `.text`, read-only constants in `.rodata`, initialized global variables in `.data`, and uninitialized variables in `.bss`. The linker combines these sections into the final kernel image. Without the linker script, the operating system would not occupy the correct memory locations.

11.12 Reading the Repository

When opening an unfamiliar directory, begin by identifying

- the header file,
- the implementation file,
- the public interface,
- the major data structures.

Then locate

- initialization functions,
- helper routines,
- exported functions.

Do not read line by line immediately. Build a map first. Experienced programmers spend considerable time understanding software organization before examining implementation details, and the larger the project becomes, the more valuable this habit becomes.

11.13 Common Mistakes

Students often place function definitions inside header files. Unless the function is explicitly intended to be inline, this produces duplicate definitions.

Another common mistake is forgetting to update header files after modifying public interfaces. Source files continue compiling until the linker discovers inconsistencies.

Many beginners also misuse global variables. Whenever possible, expose behavior through functions rather than shared global state.

Finally, avoid reading large projects sequentially. Kernel development is rarely a linear activity, and successful programmers navigate by subsystem rather than by file order.

Practice Exercises

Exercise 1

Choose one subsystem and draw its dependency graph. Which header files does it include, and which other modules depend upon it?

Exercise 2

Locate every `static` function in one source file, and explain why each helper remains private. Would making any of them public improve the design?

Exercise 3

Open the Makefile and identify:

- compiler flags,
- assembler commands,
- linker commands,
- object files.

Then explain the purpose of each stage.

Exercise 4

Open the linker script and locate:

- `.text`
- `.data`
- `.rodata`
- `.bss`

Determine where each section begins in memory, and explain why the order matters.

Historical Perspective

Early software systems often consisted of only a few source files because memory and storage were extremely limited. As operating systems grew to millions of lines of code, modular organization became essential. The C language encouraged this approach through separate compilation, header files, and external linkage, allowing teams of programmers to develop independent subsystems simultaneously.

Modern kernels such as Linux contain tens of thousands of source files organized into carefully designed subsystems. Despite this enormous scale, the organizational principles remain remarkably similar to those used in this tutorial. Source files implement behavior, header files define interfaces, the preprocessor assembles translation units, and the linker combines everything into a single executable image. Understanding these concepts transforms a seemingly overwhelming codebase into a collection of manageable components.

Chapter Summary

Large operating systems cannot be developed as single source files. Instead, they are divided into modules that expose carefully designed interfaces through header files while hiding implementation details inside source files. In this chapter, we distinguished declarations from definitions, examined include guards, explored the role of the C preprocessor, and followed the compilation pipeline from source code to executable kernel.

We also studied the Makefile and linker script, two files that often receive little attention from beginners but are essential to understanding how the operating system is built. By learning to read a codebase as a collection of interacting modules rather than isolated files, you are developing one of the most important skills of a systems programmer.

In the next chapter, we will examine another defining characteristic of kernel development: programming in a **freestanding environment**, where the familiar conveniences of the standard C library no longer exist and the operating system must provide its own implementations of fundamental services such as memory copying, string manipulation, and formatted output.

Chapter 12 - The Freestanding C Environment

Learning Objectives

After completing this chapter, you should be able to

- distinguish between hosted and freestanding C environments,
- explain why kernels cannot use the standard C library,
- understand how the compiler treats freestanding programs,
- read and implement fundamental library functions such as `memcpy()` and `memset()`,
- understand why functions like `printf()` do not exist in early kernels,
- recognize the responsibilities of a runtime environment, and
- confidently write C code that executes without an operating system.

12.1 The First Surprise

One of the first surprises students encounter while writing an operating system is that familiar C functions disappear. Consider this simple program.

```c
#include <stdio.h>

int main(void)
{
    printf("Hello, World!\n");
}
```

On Linux, Windows, or macOS, this program compiles immediately. Now try writing the same program inside your kernel, and the compiler reports an error: `printf()` does not exist. Neither does `malloc()`, neither does `fopen()`, and even `exit()` has no meaning.

At first glance, this seems strange. After all, aren't these part of the C language? The answer is no. They belong to the **standard C library**, not to the language itself, and

this distinction is one of the most important lessons in systems programming.

12.2 Hosted Versus Freestanding

The C standard defines two execution environments. The first is the **hosted environment**, and most programmers spend their careers working entirely within it. A hosted environment provides

- an operating system,
- a standard library,
- program startup code,
- input and output,
- files,
- memory allocation,
- process management.

When you compile an ordinary application, the compiler assumes all of these services already exist.

Kernel development uses the second environment. A **freestanding environment** provides almost none of them. The compiler guarantees only a very small subset of the language, and everything else becomes the programmer's responsibility. Ironically, the operating system itself is the software that eventually provides these missing services, so the kernel cannot depend upon facilities that it has not yet created.

12.3 What the Compiler Knows

Open the kernel's Makefile. One compiler option should immediately attract your attention.

`-ffreestanding`

This single option changes many compiler assumptions. Without it, the compiler expects a hosted environment; with it, the compiler understands that no operating system exists. It stops assuming that standard startup code is available, it avoids certain optimizations that rely on library behavior, and it expects the programmer to provide essential runtime support.

The compiler has not become less capable. It has simply become more honest, because it no longer assumes someone else has solved the difficult problems.

12.4 Where Does Execution Really Begin?

Every C programming textbook introduces the same function.

```
int main(void)
{
```

```
    ...
}
```

In a hosted environment, `main()` is special. Startup code supplied by the toolchain runs first — it sets up the stack, initializes the standard library, and gathers the command-line arguments — and only then calls `main()`. The programmer never sees this machinery; `main()` simply appears to be where the program begins.

A kernel has no such startup code, so nothing calls `main()` on its behalf. After reset the processor begins execution from a hardware-defined address, firmware and the bootloader hand control to the kernel, and the kernel's own assembly runs first. Only after it has prepared the machine — entering long mode and setting up a stack — does it call into C. Conceptually,

```
Power On

↓

Firmware

↓

Bootloader

↓

Assembly Startup   (the symbol `start`, in boot.s)

↓

C Code             (main)
```

The true entry point is not `main()`; it is whatever symbol the linker script names. Open `link.ld` and you will find the line `ENTRY(start)`, and `start` is defined in `boot.s`. This kernel *does* have a function called `main` — its signature is `int main(struct multiboot *mboot_ptr)` — but there is nothing magic about the name. It is an ordinary function that the assembly calls, with `call main`, once the processor is ready, and it receives a pointer to the information the bootloader left behind. Rename it, and nothing breaks as long as `boot.s` calls the new name. The point is not that `main()` is forbidden in a kernel, but that in a freestanding program *you* decide where execution begins.

12.5 Who Provides `memcpy()`?

Suppose the paging subsystem needs to copy memory. The obvious solution seems to be

```
memcpy(destination, source, size);
```

Where does this function come from? In an application, the standard library provides it. Inside the kernel, no such library exists, so the kernel must implement its own version. A simplified implementation might resemble

```
void *memcpy(void *dest,
             const void *src,
             size_t n)
{
    unsigned char *d = dest;
    const unsigned char *s = src;

    while (n--)
        *d++ = *s++;

    return dest;
}
```

Nothing magical occurs; the function copies bytes one at a time. The kernel's own version has a deliberately different signature — `void memcpy(u8int *dest, const u8int *src, u32int len)` — returning nothing and using the fixed-width typedefs of Chapter 1 rather than the standard `void *` and `size_t`. The idea is identical; only the interface is the kernel's own. The important point is not the implementation but ownership. The kernel now owns this function, and it is no longer borrowed from someone else's library.

12.6 Building Your Own Library

As the operating system grows, familiar functions gradually reappear — not because the compiler provides them, but because the kernel implements them. Typical examples include

- `memcpy()`
- `memmove()`
- `memset()`
- `memcmp()`
- `strlen()`
- `strcmp()`
- `strcpy()`

Eventually the kernel possesses its own collection of utility routines. These functions resemble the standard library, but they are not the standard library; they belong to the operating system. This distinction becomes especially important when debugging, because if `memcpy()` contains a bug, there is no vendor library to blame. The bug belongs to your kernel.

12.7 Why `printf()` Is Difficult

Students often wonder why implementing `printf()` receives so much attention. After all, printing text seems straightforward. The difficulty is not formatting numbers; the difficulty is output.

A hosted program writes characters to a terminal supplied by the operating system, but the kernel has no terminal, no console, no file descriptor, and no output stream. Someone must first communicate with the video hardware or serial port, and only after output exists can formatted printing exist.

This explains why early kernels typically begin with functions such as

```
monitor_put()
```

```
monitor_write()
```

These routines write characters directly into video memory, and higher-level formatting functions appear only afterward. Software is built layer by layer.

12.8 Dynamic Memory Without `malloc()`

Applications request memory through

```
malloc()
```

and the operating system fulfills the request. Inside the kernel, no operating system exists, so the kernel must become its own allocator. This explains the progression of previous chapters. First came the placement allocator, then the kernel heap, and eventually these routines become the kernel's equivalent of `malloc()`. Applications depend upon the operating system; the operating system depends upon itself.

12.9 Startup Code

Ordinary applications never initialize the processor, because the operating system has already done that work. Kernel software begins much earlier. Assembly startup code performs tasks such as

- establishing the stack,
- enabling long mode,
- initializing processor registers,
- configuring descriptor tables,
- enabling paging,
- preparing memory management.

Only after these tasks are complete is C allowed to execute safely. This explains why every kernel begins with assembly language rather than C. The language itself is not the limitation; the execution environment is.

12.10 Reading `common.c`

The repository contains implementations of several familiar utility functions. Open `common.c`, and you will recognize many names immediately: `memcpy()`, `memset()`, and `strlen()`.

Do not dismiss these implementations as uninteresting. Each one illustrates how surprisingly little machinery is required to build useful abstractions. Modern libraries contain countless optimizations, but educational kernels prioritize clarity instead, and understanding the simple versions first makes optimized implementations much easier to appreciate later.

12.11 Avoiding Hidden Dependencies

Kernel programmers develop an important habit: they constantly ask

> Where does this function come from?

If the answer is

"The operating system"

they immediately ask another question: which operating system? When building a kernel, there is only one acceptable answer — "My operating system." This mindset prevents accidental dependence upon unavailable facilities. Whenever you add new code, consider whether it relies on services that the kernel has not yet implemented.

12.12 Reading Freestanding Code

Freestanding C often appears simpler than application code. There are fewer libraries, fewer abstractions, and more direct interaction with hardware. At first this simplicity can feel limiting, but eventually it becomes liberating. You know exactly where every function originates, every abstraction can be inspected, and nothing is hidden behind opaque system libraries. This transparency is one of the pleasures of operating system development.

12.13 Common Mistakes

Students frequently include headers such as

```
#include <stdio.h>
```

or

```
#include <stdlib.h>
```

without realizing that these libraries are unavailable.

Another common mistake is assuming that compiler support implies runtime support. The compiler may recognize a function declaration, but that does not mean the function exists.

Many beginners also forget that standard startup code performs considerable initialization before `main()` executes. Kernel programmers must perform these tasks themselves.

Finally, avoid copying application code directly into the kernel. Always examine its dependencies first.

Practice Exercises

Exercise 1

Locate the repository's implementations of:

- `memcpy()`
- `memset()`
- `strlen()`

Explain why each one is necessary.

Exercise 2

Write your own implementation of `strcmp()`, then compare it with the version in the repository. Which is easier to read?

Exercise 3

Trace the complete startup sequence from the assembly entry point, and identify the first C function that executes. Explain why earlier execution cannot occur in C.

Exercise 4

Attempt to compile a kernel module that includes `<stdio.h>`. Record the resulting compiler or linker errors, and explain why they occur.

Historical Perspective

The earliest C compilers targeted operating systems that already existed, making the hosted environment the norm for application programmers. Operating-system developers, however, have always worked in a freestanding environment where the runtime must be constructed from the ground up.

Modern kernels still follow this model. During the earliest stages of boot, Linux, FreeBSD, and other operating systems execute with only a tiny collection of self-contained utility routines. Only after memory management, device drivers, and system initialization become available do more sophisticated runtime facilities appear. Understanding the freestanding environment reveals an important truth about systems

programming: every library, every runtime, and every operating system ultimately rests upon a surprisingly small foundation of carefully written code.

Chapter Summary

Kernel programming takes place in a freestanding C environment where the operating system and standard library do not yet exist. In this chapter, we distinguished hosted and freestanding execution, explored the role of compiler options such as `-ffreestanding`, and examined why kernels must implement their own versions of familiar functions like `memcpy()` and `memset()`. We also saw why `printf()`, `malloc()`, and even `main()` are absent during the earliest stages of boot.

The most important lesson is conceptual rather than technical. The operating system is not a client of the runtime environment—it is the runtime environment. Every service available to future applications must first be designed and implemented by the kernel itself.

In the next chapter, we shift our attention from writing kernel code to reading it. We will develop a systematic approach for exploring the repository, following execution paths, tracing data structures, and understanding unfamiliar source files without becoming overwhelmed by their size.

Chapter 13 - Reading and Writing Kernel Code

Learning Objectives

After completing this chapter, you should be able to

- develop a systematic approach to reading unfamiliar kernel code,
- identify the major subsystems of the repository,
- trace program execution across multiple source files,
- distinguish between interface code and implementation code,
- understand dependencies between kernel modules,
- use debugging and source navigation tools effectively,
- recognize common architectural patterns in operating system code, and
- confidently modify and extend the kernel without becoming overwhelmed.

13.1 Learning to Read Before Learning to Write

Many beginning programmers believe writing code is the difficult part. Experienced programmers know otherwise: reading code is harder. A large operating system contains thousands of lines spread across dozens of files, and no one understands the entire codebase after reading it once.

Professional kernel developers rarely begin by writing new code. They begin by reading. They study interfaces, follow execution paths, and identify responsibilities, and only after understanding the existing design do they make changes. This chapter teaches that process.

13.2 Do Not Read Line by Line

Students often open a source file and begin reading from the first line. This works for a one-page assignment, but it fails for an operating system.

Imagine receiving the blueprints for an entire city. You would not begin by examining every brick. You would first identify

- the roads,
- the buildings,
- the neighborhoods,
- the transportation system.

Only afterward would individual structures make sense. Kernel source code is similar, so always begin with the architecture and leave the implementation details for later.

13.3 Start with the Directory Structure

Before opening any source file, examine the repository itself. Each directory usually represents one subsystem. For example,

```
boot/
common/
drivers/
memory/
interrupts/
tasking/
syscalls/
```

The exact names differ between projects, but the principle remains the same: directories describe responsibilities. Asking

> "Why does this directory exist?"

is often more valuable than asking

> "What does this function do?"

because large software is organized around responsibilities.

13.4 Read the README First

One of the best habits you can develop is surprisingly simple: read the documentation before reading the implementation. Each chapter of the tutorial contains a README explaining

- the purpose of the chapter,
- the new concepts introduced,
- the major files,
- the expected behavior.

Many students skip this step because they want to "look at the real code," but the README *is* part of the real code. Good documentation saves hours of confusion, and professional developers read documentation first whenever it exists.

13.5 Build a Mental Map

When opening a new subsystem, avoid reading individual statements immediately. Instead, answer four questions.

First, **what problem does this subsystem solve?** Is it memory management, interrupt handling, or scheduling?

Second, **what are its public functions?** Look in the header file.

Third, **what data structures does it define?** Read the structure declarations before reading the algorithms.

Fourth, **which modules depend upon it?** Understanding relationships is more valuable than memorizing implementation.

Only after answering these questions should you study the code itself.

13.6 Trace Execution Instead of Reading Files

Programs execute; files do not. Suppose the kernel boots successfully. Rather than reading `main.c` completely, ask, "What happens first?" Execution might resemble

```
Bootloader

↓

Assembly Startup

↓

Kernel Initialization

↓

Descriptor Tables

↓

Interrupts

↓

Paging

↓

Heap
```

```
↓

Scheduler

↓

User Mode
```

This execution path crosses many source files, and following the flow of execution is often easier than studying files independently. Programs are stories, stories have sequences, and you should read them as sequences.

13.7 Learn to Follow Functions

Suppose you encounter

```
initialise_paging();
```

Do not continue reading immediately. Instead, find its declaration, then find its implementation, then identify the functions it calls, and then repeat the process. This gradually reveals the architecture of the subsystem.

Modern editors make this easy. “Go to Definition” is one of the most valuable features in an IDE, so use it constantly.

13.8 Understand Data Before Algorithms

Many algorithms manipulate complex structures. Students often begin with the algorithm, but experienced programmers usually begin with the data. Consider

```
page_t
```

Before reading functions that manipulate page tables, understand what a page-table entry contains. Similarly, understand

```
registers_t
```

before reading interrupt handlers, and understand

```
task_t
```

before reading the scheduler. Algorithms become much easier once the underlying data makes sense.

13.9 Learn the Initialization Sequence

Kernel development differs from application programming because initialization matters enormously. Many components depend upon earlier components: interrupts can-

not work before the IDT exists, paging cannot function before page tables exist, and the scheduler cannot execute processes before memory management exists.

Initialization therefore forms a dependency graph, and understanding this graph often explains why code appears in its current order.

13.10 Reading One Function

Suppose you open a function containing fifty lines. Resist the urge to understand every statement immediately, and instead ask a series of questions. What enters?

`Arguments`

What leaves?

`Return value`

What state changes? Which functions are called? What assumptions does the function make?

These questions reveal the purpose of the function before revealing its details, and purpose should always precede implementation.

13.11 Looking for Patterns

Operating systems contain many recurring patterns: initialization functions, interrupt handlers, allocation routines, lookup functions, dispatcher functions, helper functions, and error handling. Once you recognize these patterns, unfamiliar code begins to feel familiar. You are no longer reading individual functions; you are recognizing designs. Pattern recognition is one of the defining characteristics of experienced programmers.

13.12 Using Your Tools

Modern editors provide excellent navigation features, and you should learn them. Use Go to Definition, Find All References, Symbol Search, Call Hierarchy, and File Search, because these tools reduce hours of manual searching to seconds. Similarly, learn to use command-line tools such as

`grep`

`ctags`

`clangd`

`gdb`

Your editor is not merely a text editor; it is a navigation system for software.

13.13 Reading Code Like a Detective

When debugging unfamiliar code, think like an investigator. Something happened — why? What information entered, what information changed, where did it change, and which function modified it?

Do not guess. Collect evidence, observe variables, read comments, and trace execution. Kernel debugging rewards careful observation far more than clever speculation.

13.14 Making Your First Modification

Eventually you must stop reading, and the best first modification is a small one. Add a diagnostic message, rename a variable, improve a comment, or modify one constant. Then compile, boot, and observe the result.

Confidence grows through many successful small changes rather than one ambitious rewrite, and professional developers work incrementally for exactly this reason.

13.15 Common Mistakes

Students frequently try to understand everything simultaneously, which almost never succeeds. Another common mistake is jumping directly into implementation without understanding the interface.

Many beginners also ignore comments and documentation because they assume “real programmers” read only source code. In reality, professional programmers eagerly read good documentation because it saves time.

Finally, avoid copying code you do not understand. Understanding should always precede modification.

Practice Exercises

Exercise 1

Choose one subsystem and draw its dependency diagram. Which modules call it, and which does it call?

Exercise 2

Starting from `main()`, trace execution until paging becomes active, recording every major function you encounter. Summarize each one in a single sentence.

Exercise 3

Choose one structure and locate every function that accesses it. Determine which functions create it, which modify it, and which destroy it.

Exercise 4

Open an unfamiliar source file. Without reading every line, identify:

- its purpose,
- its public interface,
- its major data structures,
- its dependencies.

Only then read the implementation, and compare your initial understanding with your final one.

Historical Perspective

Large software systems have always presented a challenge to new developers. Early operating systems consisted of only a few thousand lines of code, making complete understanding achievable. Modern kernels contain tens of millions of lines spread across thousands of directories, and no individual developer understands every subsystem in detail.

Professional programmers therefore rely on systematic reading strategies rather than complete memorization. They construct mental models, follow execution paths, study interfaces before implementations, and use powerful navigation tools to explore unfamiliar code efficiently. These habits are as important as knowing the C language itself.

Chapter Summary

Learning to read kernel code is a skill distinct from learning to write it. Throughout this chapter, we developed a systematic approach to understanding large codebases by studying architecture before implementation, tracing execution rather than files, understanding data before algorithms, and relying on interfaces to guide exploration. We also introduced practical navigation techniques using modern development tools and emphasized the value of making small, incremental modifications as a way to build confidence.

By this point in the book, you have acquired most of the C language and systems programming concepts required to understand the tutorial repository. The remaining challenge is no longer syntax but engineering: learning how to explore, debug, and extend a complex software system without becoming lost.

In the next chapter, we focus on one of the defining activities of kernel development—debugging. Unlike application debugging, kernel debugging takes place without the safety net of an operating system. We will learn how to use QEMU, GDB, assertions, panic handlers, logging, and careful reasoning to diagnose problems when the entire machine appears to have stopped working.

Chapter 14 - Debugging a Kernel

Learning Objectives

After completing this chapter, you should be able to

- explain why kernel debugging differs from application debugging,
- use QEMU as a development platform,
- attach GDB to a running kernel,
- understand the purpose of assertions and panic handlers,
- interpret common kernel failures,
- trace bugs across multiple subsystems,
- develop systematic debugging habits, and
- diagnose problems using observation instead of guesswork.

14.1 When Everything Stops

Application programs fail, and operating systems fail differently. When an application crashes, the operating system usually survives: a window closes, an error message appears, and you restart the application.

Kernel failures are different, because the operating system itself is the program that has failed. There is no higher-level software to recover from the error, so the entire machine may freeze, the display may stop updating, the keyboard may become unresponsive, and the processor may repeatedly reboot.

At first this seems discouraging. In reality, debugging kernels is often simpler than debugging large applications. The kernel is smaller, its execution path is more predictable, and most importantly, you control every layer of the software stack.

14.2 Debugging Begins Before the Bug

Many students believe debugging begins after something breaks. Experienced programmers know otherwise: debugging begins while writing the code. Clear function names, small functions, meaningful comments, simple interfaces, and consistent formatting reduce the number of bugs long before the first program executes.

Debugging is not merely a collection of tools; it is a way of writing software. Code that is easy to read is usually easier to debug.

14.3 Reproducing the Problem

Suppose the kernel suddenly reboots. Do not immediately begin changing code. Instead, answer three questions. Can the problem be reproduced? Does it happen every time? What is the last thing that works correctly? These questions narrow the search dramatically.

Imagine the boot sequence.

```
Bootloader

↓

Screen Initialization

↓

Descriptor Tables

↓

Paging

↓

Heap

↓

Scheduler
```

If the screen initializes successfully but paging never completes, the problem almost certainly lies between those two stages. Finding where the system stops is often more valuable than immediately finding why.

14.4 QEMU Is Your Laboratory

Throughout this tutorial, we execute the operating system inside QEMU. QEMU is much more than an emulator; it is a laboratory. You can reboot instantly, pause execution, inspect registers, observe memory, attach debuggers, and experiment freely.

Mistakes become inexpensive. Unlike physical hardware, virtual machines encourage exploration, so never hesitate to break your kernel. A broken kernel often teaches more than a working one.

14.5 The Simplest Debugger: Printing

Students sometimes dismiss printing as primitive, yet professional kernel developers use it constantly. Suppose initialization proceeds through several stages. Add diagnostic messages.

```
monitor_write("Initializing GDT...\n");

monitor_write("Initializing IDT...\n");

monitor_write("Initializing Paging...\n");
```

If the final message never appears, you immediately know where execution stopped. This technique is remarkably effective, because simple observations often solve complicated problems.

14.6 Panic Early

Every kernel eventually encounters situations from which recovery is impossible. Rather than continuing with corrupted state, the kernel deliberately stops. This is the purpose of

```
PANIC()
```

A panic handler should answer three questions. What failed? Where did it fail? Why did execution stop? A useful panic message resembles

```
PANIC

Page Fault

Address: 0xFFFF800000123000

File: paging.c

Line: 247
```

The goal is not merely to stop; the goal is to provide useful information.

14.7 Assertions

Assertions catch programming mistakes before they become mysterious failures. Consider

```
ASSERT(current_task != NULL);
```

If the pointer unexpectedly becomes `NULL`, execution stops immediately. Without the assertion, the kernel may continue until memory becomes corrupted, and the eventual failure may appear hundreds of functions later.

Assertions move failures closer to their true causes, which dramatically simplifies debugging. Good programmers use assertions liberally during development.

14.8 Reading a Crash

Imagine the kernel displays

```
Page Fault

RIP = 0xFFFFFFFF80101234

CR2 = 0x0000000000000000
```

Do not panic. Begin interpreting the information. `RIP` identifies the instruction that caused the fault, while `CR2` identifies the address the processor attempted to access. Together they answer two questions: where was the processor, and what memory did it attempt to use? Many crashes become understandable once these two values are known.

14.9 Using GDB

Printing eventually reaches its limits, and some bugs require interactive inspection. GDB allows exactly that. The typical workflow looks like

```
Compile Kernel

↓

Start QEMU

↓

Attach GDB

↓

Set Breakpoints
```

```
↓

Inspect State

↓

Continue Execution
```

Instead of guessing variable values, you inspect them directly; instead of imagining the call stack, you display it; instead of wondering which function executed, you observe it. Debuggers replace speculation with evidence.

14.10 Breakpoints

Suppose paging initialization appears suspicious. Rather than inserting dozens of print statements, place a breakpoint. Execution stops before the function begins, and you may then inspect

- registers,
- variables,
- pointers,
- memory,
- stack frames.

Continue execution one instruction at a time, observing rather than assuming. One carefully chosen breakpoint often replaces hours of guessing.

14.11 Following the Stack

Every crash has a history, and the stack records that history. Suppose the current function is

```
alloc_frame()
```

The stack may reveal

```
alloc_frame()

↓

get_page()

↓

initialise_paging()
```

↓

`main()`

The deepest function may contain the symptom, while the higher functions often contain the cause. Always inspect the complete call chain, because software rarely fails in isolation.

14.12 Common Kernel Failures

Most early kernel bugs belong to only a few categories.

Null pointer dereferences occur when the code attempts to access address zero.

Invalid page mappings arise when virtual addresses point nowhere.

Stack corruption happens when saved return addresses become damaged.

Incorrect interrupt handlers cause the processor to restore corrupted register values.

Uninitialized variables leave structures containing unpredictable values.

Off-by-one errors cause loops to exceed array boundaries.

Recognizing these recurring patterns dramatically reduces debugging time.

14.13 Binary Search for Bugs

Suppose one hundred functions execute before the kernel crashes. Reading every function would be inefficient. Instead, divide the search. Determine whether execution reaches the middle: if it does, the bug lies later, and if not, it lies earlier. Repeat.

This strategy resembles binary search, because each observation cuts the search space roughly in half. Experienced programmers unconsciously apply this technique every day.

14.14 Debugging Memory Problems

Memory bugs often appear far from their true causes. Suppose

`kmalloc()`

returns an invalid pointer. The crash may occur much later when another subsystem dereferences it, so do not assume the failing function contains the bug.

Instead, trace the pointer backward. Where was it allocated? Who modified it? Was it initialized? Where did it become invalid? Debugging often means tracing data rather than tracing execution.

14.15 Reading Registers

Processor registers tell stories. CR3 reveals the active page tables, RSP identifies the current stack, RIP identifies the current instruction, and RFLAGS reveals processor state.

Students often ignore registers because they appear unfamiliar. In reality, registers frequently contain the most valuable debugging information available, so learn to inspect them regularly.

14.16 One Change at a Time

Beginning programmers frequently make several modifications simultaneously. When the kernel suddenly works, they do not know why, and when it breaks further, they also do not know why.

Professional developers make one change, then compile, boot, observe, and repeat. This discipline may seem slow, but in practice it is remarkably fast, because every experiment produces reliable information.

14.17 Keep a Debugging Journal

Complex bugs rarely disappear immediately. Record what you changed, what you observed, what hypotheses failed, and what evidence supports the next step.

Writing clarifies thinking, and many programmers discover solutions while describing the problem itself. A debugging notebook often becomes as valuable as the debugger.

14.18 Common Mistakes

Students often begin changing code before understanding the failure, which usually creates additional bugs. Another common mistake is assuming the last function executed contains the defect, but the symptom and the cause are frequently different.

Many beginners also ignore compiler warnings, even though warnings deserve attention. Finally, avoid debugging from memory. Observe the system, inspect variables, read registers, and measure, because good debugging relies on evidence rather than intuition.

Practice Exercises

Exercise 1

Insert diagnostic messages throughout the boot sequence, then introduce a deliberate page fault and determine precisely where execution stops.

Exercise 2

Attach GDB to QEMU and set a breakpoint at `initialise_paging()`. Single-step through the function, recording every significant state change.

Exercise 3

Modify one assertion so that it intentionally fails, and observe the panic output. Would the diagnostic information it prints help you identify the bug?

Exercise 4

Introduce a deliberate null pointer dereference, and predict the resulting exception before running the kernel. Compare your prediction with the observed behavior.

Historical Perspective

Early operating system developers often debugged kernels using front-panel switches, serial terminals, or hardware logic analyzers because sophisticated software debuggers did not yet exist. Diagnosing failures required careful reasoning and an intimate understanding of processor architecture.

Modern virtualization has transformed kernel development. Tools such as QEMU and GDB allow developers to pause execution, inspect registers, trace memory accesses, and analyze crashes with unprecedented precision. Yet the underlying principles remain unchanged, because effective debugging depends far more on disciplined observation than on advanced tools.

The best debugger has always been a programmer who asks careful questions and follows the evidence wherever it leads.

Chapter Summary

Kernel debugging differs fundamentally from application debugging because the operating system itself is the software being investigated. Throughout this chapter, we explored practical debugging techniques using diagnostic output, panic handlers, assertions, QEMU, GDB, breakpoints, call stacks, and processor registers. More importantly, we developed a systematic approach to debugging based on observation, controlled experiments, and incremental changes rather than guesswork.

As your kernels become more sophisticated, the number of possible failures will increase, but the debugging process will remain remarkably consistent. Reproduce the problem, narrow the search space, collect evidence, test one hypothesis at a time, and allow the system itself to reveal the answer.

In the next chapter, we bring together everything learned throughout this book by examining the common programming patterns and design idioms that appear repeatedly across operating system code. Once you recognize these patterns, unfamiliar kernel code will become much easier to understand and extend.

Chapter 15 - Thinking Like a Kernel Programmer

Learning Objectives

After completing this chapter, you should be able to

- recognize the recurring programming patterns used throughout the kernel,
- understand why operating system code favors simple abstractions over clever ones,
- identify common C idioms used in systems programming,
- explain the design philosophy behind kernel development,
- distinguish between educational code and production-quality kernel code,
- read unfamiliar kernel subsystems by recognizing patterns instead of memorizing implementations,
- write new kernel components that fit naturally into the existing codebase, and
- continue studying larger operating systems such as Linux with confidence.

15.1 Looking Back

When you began this book, a kernel source file probably looked unfamiliar. There were pointers everywhere, structures seemed enormous, assembly appeared mysterious, and the build system looked intimidating.

Now those pieces have become familiar. You know how interrupts reach C, you know how paging structures are organized, and you understand why the kernel implements its own memory allocator. You have learned the mechanics, and the final step is learning the mindset. Professional programmers do not memorize every function; they recognize patterns, and this chapter is about those patterns.

15.2 Simplicity Wins

Students often expect operating system code to be clever, imagining complicated algorithms and obscure language features. The opposite is usually true, because kernel code values simplicity. Simple code is easier to debug, easier to verify, and able to survive for decades.

Consider the placement allocator introduced earlier.

```
placement_address += size;
```

The algorithm is almost trivial, and it succeeds because it solves exactly one problem. Later allocators become more sophisticated, but the principle remains unchanged: solve today's problem, and avoid solving tomorrow's problem prematurely. Operating systems reward restraint.

15.3 Layers

Every subsystem in the repository builds upon another. Video output appears before formatted printing, memory allocation appears before process creation, interrupt handling appears before multitasking, and user mode appears after virtual memory. Conceptually,

```
Hardware

↓

Assembly

↓

Low-Level C

↓

Memory Management

↓

Interrupt Handling

↓

Scheduling

↓
```

```
User Programs
```

Each layer depends upon the layers beneath it, and none depend upon the layers above. This organization appears in nearly every operating system. Whenever you encounter unfamiliar code, ask, “What layer does this belong to?” The answer immediately limits what the code can reasonably do.

15.4 Small Interfaces

Kernel modules rarely expose dozens of public functions. Instead, they present a small interface. Consider paging, whose public functions might include

```
initialise_paging()

get_page()

alloc_frame()

free_frame()
```

The implementation may occupy hundreds of lines, yet the public interface remains remarkably small. This simplicity is intentional, because small interfaces reduce coupling, reduce mistakes, and simplify maintenance. Whenever you design a new subsystem, ask yourself, “What is the smallest interface that solves the problem?”

15.5 Data First

Throughout the repository, algorithms follow data structures, not the reverse. Before implementing the scheduler, the kernel defines

```
task_t
```

before implementing interrupts, it defines

```
registers_t
```

and before implementing paging, it defines

```
page_t
```

Good kernel programmers begin by designing the data, and functions naturally follow. Poorly designed structures almost always produce complicated algorithms, while well-designed structures often make algorithms surprisingly short.

15.6 Initialization Patterns

Nearly every subsystem follows the same pattern: create structures, initialize hardware, enable functionality, and verify success. For example,

```
Allocate

↓

Initialize

↓

Enable

↓

Use
```

Recognizing this sequence helps you read unfamiliar code quickly. Initialization code often appears only once, and understanding it explains everything that follows.

15.7 Tables Instead of Decisions

Operating systems frequently replace long chains of conditional statements with tables. Consider interrupt handling. Instead of

```
if interrupt == 0

else if interrupt == 1

else if interrupt == 2

...
```

the kernel stores handler addresses in a table, the interrupt number becomes an index, and the processor performs the lookup efficiently. This idea appears repeatedly in system-call tables, interrupt tables, page tables, file-operation tables, and device tables. Whenever values naturally map to indices, kernels often prefer tables over repeated decisions.

15.8 Separate Policy from Mechanism

One of the oldest ideas in operating system design is separating **mechanism** from **policy**. Mechanism answers, “How is something done?” while policy answers, “Which choice should we make?”

Consider the scheduler. Mechanism performs context switching, while policy chooses the next process. These concerns remain separate whenever possible, so changing scheduling algorithms should not require rewriting the context-switching code. This separation makes kernels easier to evolve over time.

15.9 Defensive Programming

Kernel code assumes mistakes will happen: pointers become invalid, memory becomes exhausted, and hardware behaves unexpectedly. Good code checks assumptions through assertions, panic handlers, input validation, and clear error reporting.

Defensive programming is not pessimism; it is professionalism. The kernel occupies the foundation of the software stack, and mistakes at this level affect everything above it.

15.10 Consistency Matters More Than Cleverness

Students sometimes ask whether one implementation is "more elegant." Kernel developers usually ask a different question: "Is it consistent?" Consistent naming, consistent indentation, consistent interfaces, and consistent error handling all reduce cognitive effort, because once programmers learn one subsystem, the others feel familiar.

The repository deliberately follows this philosophy, so study it carefully. It is teaching more than C; it is teaching software engineering.

15.11 Reading Linux

Many students eventually open the Linux kernel and become discouraged, because the code appears overwhelming. Remember what you have learned, and look for familiar patterns: initialization functions, tables, structures, header files, small interfaces, and subsystem boundaries.

These ideas appear in Linux just as they appear here. The difference is scale, but the design philosophy remains surprisingly similar, because large systems are usually collections of small ideas.

15.12 Writing New Code

Suppose you wish to add a new device driver. Do not begin writing code immediately. Instead, ask what subsystem should own this functionality, what interface it should expose, what structures are required, how errors will be reported, and which existing modules will depend upon it.

Good architecture usually produces straightforward implementation, whereas poor architecture rarely improves through additional code.

15.13 Common Mistakes

Beginning kernel programmers often over-engineer simple problems. Others create interfaces that expose every internal detail, some duplicate functionality instead of ex-

tending existing modules, and many focus exclusively on algorithms while ignoring software organization.

The repository demonstrates a different philosophy: simple abstractions, incremental development, clear responsibilities, and readable code. These principles remain valuable regardless of language or operating system.

Practice Exercises

Exercise 1

Choose three subsystems from the repository and identify:

- their public interfaces,
- their private helper functions,
- the data structures they define.

Then compare their organization.

Exercise 2

Locate every initialization function and determine whether they follow the same design pattern. If not, explain why.

Exercise 3

Choose one subsystem and imagine replacing its implementation without changing its public interface. Would other modules require modification, and why or why not?

Exercise 4

Open a source file from the Linux kernel and, ignoring unfamiliar details, identify at least five patterns discussed in this chapter. Summarize how they resemble the educational kernel.

Historical Perspective

Although operating systems have evolved dramatically over the past fifty years, the programming patterns used to build them have changed surprisingly little. Early UNIX kernels, educational systems such as MINIX, and modern kernels such as Linux all rely on modular organization, carefully defined interfaces, explicit data structures, incremental initialization, and disciplined separation of responsibilities.

The greatest difference is not complexity but scale. Educational kernels contain a few thousand lines of code designed to illustrate ideas clearly, while production kernels contain millions of lines designed to support thousands of hardware platforms, filesystems, networking protocols, security mechanisms, and device drivers. Yet an experienced programmer can navigate both, because the underlying design principles remain remarkably consistent.

Learning these patterns is therefore more valuable than memorizing any individual function. Programming languages evolve, processors change, and hardware becomes faster, but good software engineering practices endure.

Chapter Summary

This chapter brought together the major ideas developed throughout the book. We examined the programming patterns that appear repeatedly in operating system code: layered design, small interfaces, data-oriented design, initialization sequences, table-driven dispatch, defensive programming, and the separation of mechanism from policy. More importantly, we explored the habits of experienced kernel developers, who rely on recognition, consistency, and architecture rather than cleverness.

You should now be able to approach the 64-bit operating system tutorial not as a collection of isolated C programs but as a coherent software system. The syntax of the language is only the beginning. Understanding why the code is organized as it is, why abstractions are introduced gradually, and why seemingly simple design decisions matter so much is what transforms a C programmer into a systems programmer.

Where to Go Next

Completing this book means you have crossed an important threshold. You have learned not only the C language features needed for kernel development but also the engineering mindset required to understand a real systems codebase.

From here, several paths naturally open.

Continue through the remaining chapters of the 64-bit operating system tutorial and extend the kernel with additional features.

Study the Linux kernel, beginning with small, self-contained subsystems such as linked lists, kernel memory allocation, or scheduler data structures.

Explore more advanced topics including virtual memory management, filesystems, networking, multiprocessor scheduling, synchronization, and device-driver development.

Most importantly, continue writing code. Read existing implementations, modify them, break them, and repair them. Every successful systems programmer has learned through experimentation.

A kernel is not built by reading alone. It is built one function, one subsystem, and one carefully reasoned decision at a time.

Congratulations. You are now prepared to read, understand, and extend the operating system tutorial—and, with patience and persistence, much larger kernels as well.

Chapter 16 - Learning by Experiment

Learning Objectives

After completing this chapter, you should be able to

- develop a repeatable workflow for studying operating system code,
- learn new kernel concepts through controlled experimentation,
- modify kernel code safely and systematically,
- use observation to validate hypotheses,
- build confidence through incremental changes,
- maintain a laboratory notebook for kernel experiments,
- distinguish between understanding code and merely reading it, and
- prepare yourself for studying production operating systems such as Linux.

16.1 Reading Is Not Enough

By now you understand the C language used throughout the repository. You understand pointers, structures, memory management, interrupts, the build system, and debugging. Yet there remains one important lesson: reading code is only the beginning, and real understanding comes from changing the code.

Every experienced systems programmer learns by experimentation. They ask questions, make predictions, modify the implementation, and observe the result. Programming is an experimental science, and the computer is your laboratory.

16.2 Build, Run, Observe

Every chapter in the repository follows the same cycle. Compile the kernel, boot it, and observe its behavior, and only then should you begin modifying the source. Never edit code you have not successfully executed, because a working baseline gives you confidence and also provides something to return to if your experiments fail.

The workflow is simple.

```
Read

↓

Compile

↓

Boot

↓

Observe

↓

Modify

↓

Compile

↓

Observe Again
```

Do not skip steps, because each one teaches something different.

16.3 Ask One Question at a Time

Suppose you wish to understand paging. Do not rewrite the paging subsystem. Instead, ask one question: what happens if this page is marked read-only? Change one line, compile, and observe. Then ask another question: what happens if the page is not present? Repeat.

Learning accelerates when experiments answer one question at a time. Large changes answer too many questions simultaneously, and the results become difficult to interpret.

16.4 Predict Before You Run

One habit separates experienced programmers from beginners: they predict outcomes before executing code. Suppose you remove an interrupt gate. Ask yourself whether the kernel will boot, whether it will panic, or whether it will triple fault. Write down your prediction, and then test it.

Whether your prediction proves correct or incorrect, you learn something. Programming becomes much more interesting when every experiment begins with a hypothesis.

16.5 Change One Thing

One of the most common mistakes in kernel development is changing many things at once. Imagine modifying the scheduler, the heap, and the interrupt handlers during a single editing session. When the kernel crashes, which modification caused the problem? No one knows.

Professional developers make one logical change, then compile, test, commit, and repeat. Small changes produce understandable results, while large changes produce confusion.

16.6 Keep Notes

Maintain a notebook. For every experiment, record

- what you changed,
- why you changed it,
- what you expected,
- what actually happened,
- what you learned.

Your notes become a personal textbook. Months later you will remember neither every experiment nor every bug, but your notebook will. Many researchers maintain laboratory notebooks for exactly this reason, and kernel programming deserves the same discipline.

16.7 Learn to Break Things

Students often avoid making mistakes, but kernel developers deliberately create them. Break paging, cause a page fault, corrupt an interrupt descriptor, overflow a stack, trigger an assertion, and observe the result.

Failures reveal how systems behave under unusual conditions, and educational kernels are designed to be broken. Do not fear mistakes; fear making the same mistake twice.

16.8 Read the Compiler

Many beginners treat compiler messages as obstacles, whereas experienced programmers treat them as documentation. Read every warning, understand every error, and never ignore diagnostics simply because the kernel still compiles.

Compiler warnings often identify real bugs before the kernel executes. The compiler is your first reviewer, so listen carefully.

16.9 Use Version Control

Every experiment should be reversible, and Git makes this possible. Before beginning a new investigation, commit your current work, and then experiment freely. If everything breaks, restore the previous version.

Version control encourages exploration because mistakes become inexpensive, and professional developers rely on this safety net every day.

16.10 Read Beyond the Repository

The tutorial explains one implementation; it is not the only implementation. When you finish a chapter, read

- the Intel Software Developer's Manual,
- the AMD64 Architecture Programmer's Manual,
- the OSDev Wiki,
- relevant sections of the Linux kernel,
- academic papers describing the subsystem.

Compare different implementations, noticing what changes and what remains the same. This comparison deepens understanding.

16.11 Explain It to Someone Else

One of the fastest ways to discover gaps in your understanding is to teach. Explain page tables to a classmate, describe interrupt handling on a whiteboard, and write a short summary after each chapter. If you cannot explain an idea simply, you probably do not understand it completely. Teaching is a powerful debugging tool for your own thinking.

16.12 Build a Mental Model

Avoid memorizing code, and instead build mental models. Paging, for example, is not merely a collection of functions; it is a translation mechanism. The scheduler is not merely a source file; it is a policy for deciding which process executes next. Interrupt handling is not merely an assembly routine; it is a communication mechanism between hardware and software.

Mental models survive implementation changes. Line numbers do not.

16.13 Think Like a Researcher

Systems programming and systems research share the same habits: observe, measure, hypothesize, experiment, analyze, and repeat. When performance changes, measure it;

when behavior changes, explain it; and when something surprises you, investigate it. Curiosity is one of the most valuable tools a systems programmer possesses.

16.14 Preparing for Linux

Many students ask, “When should I begin reading the Linux kernel?” The answer is sooner than you think, because you already possess the necessary foundation.

Do not attempt to understand Linux completely. Choose one subsystem—memory allocation, linked lists, scheduling, or virtual memory—and read only a few functions, recognizing familiar patterns. The educational kernel has prepared you well. Linux is larger, but its principles are remarkably similar.

16.15 Your Study Workflow

For every chapter of the operating system tutorial, follow the same routine.

1. Read the README completely.
2. Compile the kernel without making changes.
3. Boot the kernel.
4. Observe the expected behavior.
5. Read the major header files.
6. Read the implementation files.
7. Trace one execution path.
8. Make one small modification.
9. Test the modification.
10. Record what you learned.

This process transforms passive reading into active learning.

16.16 Common Mistakes

Students often try to finish chapters quickly, but speed is not the goal—understanding is. Another common mistake is copying code from the Internet without understanding it; the code may work, yet you may learn very little.

Some beginners also avoid reading documentation because they believe the source code tells the whole story, even though good documentation shortens the learning process considerably. Finally, avoid comparing yourself with experienced kernel developers. Every expert once struggled to understand pointers, page tables, and interrupts, and mastery comes through repeated practice.

Practice Exercises

Exercise 1

Choose any completed chapter from the tutorial and, before reading its source code, write down three questions about the implementation. After studying the code, answer each in your own words.

Exercise 2

Modify one constant in the kernel and predict the outcome. Compile, run, and record your observations, then explain any differences between your prediction and the actual behavior.

Exercise 3

Create a laboratory notebook for your kernel experiments, and document one complete debugging session in it, from initial hypothesis to final solution.

Exercise 4

Choose one subsystem from the educational kernel and locate its equivalent in the Linux kernel. Compare their:

- directory organization,
- public interfaces,
- data structures,
- initialization sequence.

Then summarize the similarities and differences.

Historical Perspective

The most influential operating system developers learned through experimentation rather than passive reading. Early UNIX evolved through continuous modification, measurement, and refinement. Educational operating systems such as MINIX were created not only to provide working software but also to encourage students to explore, modify, and understand every subsystem.

Modern open-source development follows the same philosophy. Contributors to projects such as Linux rarely begin by writing entirely new subsystems. They start with small fixes, read existing code, perform controlled experiments, and gradually build an understanding of the architecture. This incremental approach has proven remarkably effective for developing both expertise and reliable software.

Chapter Summary

Throughout this book you have learned the C language, the architecture of a small operating system, and the programming patterns used throughout the repository. This

final chapter focused on **how to continue learning**. We developed a disciplined workflow based on observation, experimentation, prediction, measurement, and reflection. These habits transform a code repository from a collection of files into a laboratory for understanding computer systems.

Perhaps the most important lesson is that mastery comes from interaction, not memorization. Read the code carefully, but do not stop there. Compile it, run it, modify it, break it, and repair it, and record what you discover. Every experiment strengthens your intuition about how computers really work.

Final Thoughts

You now possess a foundation that many programmers never acquire. You understand not only how to write C programs, but also how C interacts with hardware, how an operating system is organized, how processors communicate with software, how memory is managed, how debugging works without an underlying operating system, and how to study a large systems codebase effectively.

The 64-bit kernel tutorial is no longer simply a programming project. It has become a guided exploration of computer systems. Continue through it with curiosity, patience, and discipline. Ask questions, test ideas, and build confidence one subsystem at a time.

The journey from C programmer to systems programmer does not end here. It begins here.

Appendix A - C Syntax Essentials

A Quick Reference for Reading the Tutorial

This appendix collects the C syntax you will actually encounter while reading the tutorial's kernel source. It is not a complete description of the C language, and it does not attempt to teach programming from scratch. Instead, it gathers the essential constructs in one place so that you can look them up quickly. Wherever a topic deserves fuller treatment, the relevant chapter is noted.

A.1 Comments

C provides two comment styles. Both appear throughout the codebase.

```
// A single-line comment continues to the end of the line.

/* A block comment
   may span several lines. */
```

Comments are removed by the preprocessor before compilation and have no effect on the generated machine code.

A.2 Basic Types and the Kernel's Typedefs

Standard C provides a small set of built-in types.

```
char   c;   // a single byte
short  s;   // usually 16 bits
int    i;   // usually 32 bits
long   l;   // 64 bits under the LP64 model
void       // "no type"; also used for generic pointers
```

Because exact sizes matter in kernel code, the tutorial defines its own fixed-width names (see Chapter 1).

```
u8int   b;   // unsigned 8-bit
u16int  w;   // unsigned 16-bit
u32int  d;   // unsigned 32-bit
u64int  q;   // unsigned 64-bit
s32int  n;   // signed 32-bit
```

Prefer these names whenever a value has a hardware-defined width.

A.3 Variables and Constants

A variable is declared with a type and a name, and may be initialized at the same time.

```
u32int count = 0;
u64int address = 0x100000;
```

Numeric constants may be written in decimal or, more commonly in systems code, in hexadecimal using the `0x` prefix (see Chapter 9). A suffix fixes the constant's type; `U` marks it unsigned and `LL` marks it long long.

```
0x1000              // hexadecimal
0xFFFFFFFF80000000ULL   // 64-bit unsigned constant
'A'                 // a character constant (its numeric code)
"hello"             // a string constant (an array of characters)
```

A.4 Operators

The arithmetic, relational, and logical operators behave as in most languages.

Category	Operators
Arithmetic	`+ - * / %`
Relational	`== != < > <= >=`
Logical	`&& || !`

The bitwise and shift operators are used constantly when manipulating flags and hardware registers (see Chapter 9).

Category	Operators
Bitwise	`& | ^ ~`
Shift	`<< >>`

Note the difference between the logical operators (`&&`, `||`) and the bitwise operators (`&`, `|`); they are not interchangeable. Several shorthand and unary operators also appear regularly.

```
x += 1;       // compound assignment (also -=, |=, &=, ^=, <<=, >>=)
x++;          // increment (also x--)
flags & 0x4   // test bits without changing them
(u16int *)p   // a cast: reinterpret the type of an expression
sizeof(x)     // the size in bytes of a type or object
cond ? a : b // the conditional (ternary) operator
```

Two operators are specific to memory. The address-of operator `&` yields the location of an object, and the dereference operator `*` accesses the object a pointer refers to (see Section A.7).

A.5 Control Flow

Conditional execution uses `if` and `else`.

```
if (flags & PAGE_PRESENT) {
    // present
} else {
    // not present
}
```

A `switch` selects among many constant cases; each case usually ends with `break`, and `default` handles the rest.

```
switch (interrupt_number) {
    case 14:  handle_page_fault(); break;
    default:  handle_generic();    break;
}
```

The three loop forms all appear in kernel code.

```
for (int i = 0; i < 256; i++) { ... }

while (n--) { ... }

do { ... } while (0);
```

Inside loops, `continue` skips to the next iteration and `break` exits the loop. A function returns with `return`, optionally yielding a value. An intentional infinite loop, often used to halt the processor, is written as follows.

```
for (;;) { }     // or: while (1) { }
```

A.6 Functions

A function is introduced by a prototype (its interface) and later given a definition (its implementation).

```
void initialise_paging(void);              // prototype

void initialise_paging(void) {             // definition
    ...
}
```

Parameters are listed with their types, and `void` marks a function that takes no parameters or returns nothing. Two qualifiers appear frequently. `static` limits a function to the file in which it is defined, and `inline` suggests the compiler substitute the body directly at the call site. Some kernels combine the two to give a tiny hardware routine a header-defined, call-free form; this tutorial keeps `inb`/`outb` as ordinary functions in `common.c` (see Chapter 4 §4.8).

```
static void helper(void) { ... }
static inline u64int rdtsc(void) { ... }
```

A.7 Pointers

A pointer holds the address of an object (see Chapter 3). It is declared with a `*`.

```
u32int *ptr;              // ptr holds the address of a u32int
u32int  value = 42;

ptr = &value;             // & takes the address of value
*ptr = 100;               // * accesses the object at that address
```

The constant `NULL` represents a pointer that refers to nothing. Pointer arithmetic is scaled by the size of the pointed-to type, so adding one advances by one whole object rather than one byte.

```
u16int *vga = (u16int *)0xB8000;
vga[1] = 0x0F42;          // same as *(vga + 1); advances two bytes
```

A `void *` is a generic pointer that carries an address without a specific type; it is cast to a concrete type before use.

```
void *mem = kmalloc(sizeof(page_t));
page_t *page = (page_t *)mem;
```

A.8 Arrays

An array is a contiguous sequence of objects of the same type, indexed from zero.

```
idt_entry_t idt[256];      // 256 descriptors
idt[0].flags = 0x8E;       // access one element
```

Arrays and pointers are closely related: the array name evaluates to the address of its first element, so `buffer[i]` is equivalent to `*(buffer + i)`.

A.9 Structures

A structure groups related values into one object whose layout in memory is fixed (see Chapter 2). The `typedef struct` form gives the type a convenient name.

```
typedef struct {
    u16int base_lo;
    u16int sel;
    u8int  flags;
} idt_entry_t;
```

Members are reached with `.` on an object and with `->` through a pointer.

```
idt_entry_t  entry;
idt_entry_t *p = &entry;

entry.flags = 0x8E;        // via the object
p->flags    = 0x8E;        // via a pointer (same as (*p).flags)
```

When a structure must match a hardware-defined layout exactly, the compiler is told not to insert padding.

```
typedef struct {
    ...
} __attribute__((packed)) idt_entry_t;
```

A.10 typedef and Function Pointers

`typedef` creates a new name for an existing type; it changes nothing about the generated code and exists only for readability.

```
typedef unsigned long long u64int;
```

The same mechanism names function-pointer types, which the interrupt subsystem uses to store handlers in a table (see Chapter 5).

```
typedef void (*isr_t)(registers_t *);

isr_t handlers[256];
handlers[33] = keyboard_handler;   // store a function's address
handlers[33](regs);                // call through the pointer
```

A.11 Type Qualifiers: const and volatile

`const` marks a value that must not be modified. `volatile` tells the compiler that a value may change on its own — for example, a hardware register — so every access must occur exactly as written and must not be optimized away (see Chapter 3).

```
const u32int limit = 4096;
volatile u32int *status = (volatile u32int *)0xFEE00000;
```

A.12 Storage and Linkage: static and extern

At file scope, `static` keeps a variable or function private to its source file, while `extern` declares that a variable is defined in another file (see Chapter 11).

```
static u32int placement_address;     // private to this file

extern u32int placement_address;     // defined elsewhere
```

Inside a function, `static` gives a local variable a lifetime that spans the whole program rather than a single call.

A.13 The Preprocessor

The preprocessor runs before compilation and performs textual substitution (see Chapter 11). `#include` inserts the contents of another file.

```
#include "common.h"
```

`#define` introduces a named constant or a macro. Function-like macros take arguments, and multi-statement macros are commonly wrapped in a `do { ... } while (0)` so they behave like a single statement.

```
#define PAGE_PRESENT 0x1

#define PANIC(msg) do { panic(msg, __FILE__, __LINE__); } while (0)
```

`PAGE_PRESENT` here is only a syntax example; this tutorial expresses its page-table flags as bitfields rather than mask constants (see Chapter 9 §9.11).

Include guards prevent a header from being processed twice.

```
#ifndef COMMON_H
#define COMMON_H
    ...
#endif
```

A.14 GCC Extensions You Will See

The tutorial targets the GCC and Clang compilers and uses a few extensions beyond standard C. Two appear often: `__attribute__(( ... ))` attaches properties to a declaration, such as `packed` on a structure, and the `asm` statement embeds assembly instructions inside a C function (see Chapters 4 and 10).

```
__attribute__((packed))                 // exact memory layout
__attribute__((noreturn))               // function never returns
asm volatile ("cli");                   // an inline assembly instruction
```

Where to Read More

Topic	Chapter
Types, sizes, and hexadecimal	1, 9
Structures and packing	2
Pointers and memory	3
Inline assembly	4
Function pointers	5
Bitwise operators and flags	9
Linking, `static`, `extern`	10, 11
The preprocessor and headers	11

Appendix B - A Guided Tour of the `03_screen_64` Kernel

Reading, Understanding, Building, and Modifying Your First 64-bit Kernel

The chapters of this book introduce one idea at a time. This appendix does the opposite: it takes a single, complete, working kernel and walks through it end to end, showing where each idea actually lives in real code. The kernel we study is the screen chapter of the 64-bit tutorial, found at

`64-bit/03_screen_64`

It does exactly one thing — it prints `Hello, world!` in the top-left corner of the screen. That modest output hides a surprising amount of machinery, and almost every concept from the book appears somewhere inside it. By the end of this tour you should be able to open the directory, read every file with understanding, build and run the kernel, and change it deliberately rather than by guessing.

Wherever a topic deserves fuller treatment, the relevant chapter is named. Read this appendix with the repository open in one window and the referenced chapters within reach.

Learning Objectives

After completing this guided tour, you should be able to

- navigate the `03_screen_64` directory and explain the purpose of every file,
- connect each file to the chapters of this book that explain it,
- trace execution from the bootloader's handoff all the way to `main()`,
- explain why a "64-bit" kernel begins life running 32-bit code,
- build the kernel and run it under an emulator,
- modify the kernel safely and predict the result before you compile,
- recognize the failure signatures of the most common mistakes.

B.1 What This Kernel Is, and What It Is Not

This kernel is a *port*. It began as the 32-bit screen chapter of James Molloy's well-known tutorial and was rewritten to run in **long mode**, the 64-bit execution mode of the x86-64 processor. The README states the essential fact plainly: almost the entire port lives in one file, `boot.s`. The C code — the screen driver, the entry point, the port helpers — is nearly identical to the 32-bit original.

That single sentence tells you where the difficulty is. The interesting part of this kernel is not *printing text*; printing text is old news. The interesting part is everything that must happen **before** a single line of 64-bit C code can run. Understanding that bootstrap is the real reward of this chapter, and it exercises Chapter 7 (paging), Chapter 9 (bit manipulation), and Chapter 10 (linking C and assembly) all at once.

Keep your expectations calibrated. When you finally see `Hello, world!` appear, the achievement is not the greeting. The achievement is that the processor is now in 64-bit mode, paging is on, a stack exists, and C code is executing — and you understand how it got there.

B.2 First, Map the Territory

Chapter 13 gives one instruction before all others: **start with the directory structure, not with the code**. Follow it now. List the files:

```
03_screen_64/
   README.md      ← read this first
   DISCUSSION.md   ← the concepts behind the port
   src/
       boot.s      ← the entire 64-bit bootstrap
       link.ld     ← where the kernel goes in memory
       Makefile    ← how the pieces become one file
       common.h    ← typedefs and hardware helpers (declarations)
       common.c    ← hardware helpers (definitions)
       main.c      ← the C entry point
       monitor.c   ← the screen driver
       monitor.h   ← the screen driver's interface
```

Nine files, and each has a single responsibility. Chapter 13 §13.4 tells you to read the README before the implementation, and here that advice pays off immediately: the README explains that only four files were rewritten for the port (`boot.s`, `link.ld`, `Makefile`, and two typedefs in `common.h`) and that `main.c`, `monitor.c`, and `common.c` are essentially untouched. That single paragraph saves you from hunting for 64-bit magic in files that contain none.

Before opening anything, build a mental map (Chapter 13 §13.5) by answering four questions:

1. **What problem does this kernel solve?** Reaching 64-bit C and printing to the screen.
2. **What is its public surface?** The three `monitor_*` functions in `monitor.h`.
3. **What data does it define?** The VGA buffer pointer, the cursor position, and three page tables.
4. **What depends on what?** `main.c` depends on `monitor.c`, which depends on `common.c`, all of which depend on `boot.s` having already reached long mode.

Hold that map in your head. Everything below fills it in.

B.3 Trace the Story Before Reading the Files

Chapter 13 §13.6 warns that *programs execute, files do not*, and that following the flow of execution is easier than reading files in isolation. Here is the whole story of this kernel as a sequence, which we will spend the rest of the appendix expanding:

```
GRUB / QEMU hands off
          (32-bit protected mode, paging OFF)

start:  in boot.s        ← 32-bit assembly
          check CPU, build page tables, enable long mode

long_mode_start:         ← first 64-bit instruction
          set up stack, pass multiboot pointer

main()  in main.c        ← ordinary C at last


monitor_clear(), monitor_write("Hello, world!")


hlt (idle forever)
```

Notice that `main()` is not the beginning. This surprises students who have only written hosted programs, and it is exactly the surprise Chapter 12 §12.4 prepares you for: *if there is no operating system, where does execution begin?* The answer here is the label `start` in `boot.s`, named as the entry point by the linker script. We will return to this.

B.4 The Type Foundation: `common.h`

Open `common.h`. It is the smallest file with the widest reach, and it is pure Chapter 1.

```
#ifndef COMMON_H
#define COMMON_H

// These typedefs are written for 64-bit X86 (LP64).
```

```
typedef unsigned long long u64int;
typedef          long long s64int;
typedef unsigned int   u32int;
typedef          int   s32int;
typedef unsigned short u16int;
typedef          short s16int;
typedef unsigned char  u8int;
typedef          char  s8int;

void outb(u16int port, u8int value);
u8int inb(u16int port);
u16int inw(u16int port);

#endif // COMMON_H
```

Everything in this file was explained earlier in the book:

- The fixed-width typedefs — `u8int` through `u64int` — are the subject of **Chapter 1** (see §1.3) and Appendix A §A.2. The kernel defines its own names rather than relying on ordinary `int` because, as §1.2 argued, the hardware demands exact sizes and the C standard refuses to promise them.
- The comment *"written for 64-bit X86 (LP64)"* points at **Chapter 1 §1.5**, the LP64 data model. Under LP64, `int` stays 32 bits but `long` and pointers grow to 64. That is why the port only had to *add* `u64int` and `s64int`: the narrower types were already correct.
- The three lines `void outb(...)`, `u8int inb(...)`, `u16int inw(...)` are **declarations, not definitions** — the distinction drawn in **Chapter 11 §11.3**. They promise these functions exist; `common.c` will deliver them.
- The `#ifndef COMMON_H` / `#define COMMON_H` / `#endif` wrapper is an **include guard**, explained in **Chapter 11 §11.6**. It prevents the typedefs from being defined twice when several files include this header.

This one header is a compact review of Chapters 1 and 11. If any line looks unfamiliar, that is a signal to reread the corresponding section before continuing.

B.5 Talking to Hardware: `common.c`

`common.c` defines the functions that `common.h` promised. Its first three functions are the beating heart of **Chapter 4**.

```
void outb(u16int port, u8int value)
{
    asm volatile ("outb %1, %0" : : "dN" (port), "a" (value));
}

u8int inb(u16int port)
{
```

```
    u8int ret;
    asm volatile("inb %1, %0" : "=a" (ret) : "dN" (port));
    return ret;
}
```

Read these alongside Chapter 4:

- `outb` is the exact example dissected in **Chapter 4 §4.4**. It sends one byte to an I/O port, a thing C alone cannot express, which is why §4.1 argued that "C is not enough."
- The strange punctuation `: : "dN" (port), "a" (value)` is the GCC inline-assembly constraint syntax broken down in **§4.5**. The `"a"` binds a value to the `AL`/`AX` register; `"dN"` allows the port number in `DX` or as an immediate.
- The keyword `volatile` is not decoration. **§4.6** explains that it forbids the compiler from optimizing the instruction away or reordering it, because the effect of talking to hardware is invisible to the compiler's model of the program.
- `inb` reads a byte *back* from a port — the mirror image, covered in **§4.7** — using an output constraint `"=a" (ret)`.

Below these three functions you will find something important for later chapters:

```
void memcpy(u8int *dest, const u8int *src, u32int len)
{
    // TODO: implement this yourself!
}
```

`memcpy`, `memset`, `strcmp`, `strcpy`, and `strcat` are present but **empty**. This is not an oversight; it is **Chapter 12** made visible. §12.5 asks *"Who provides `memcpy()`?"* and answers that in a freestanding environment, *you* do — there is no standard library. This kernel does not yet need them, so they are left as stubs for you to complete (see the exercises). When you build, the compiler will warn that the string functions "reach the end of a non-void function"; that warning is Chapter 12's point arriving on schedule.

B.6 The Screen Driver: `monitor.c`

The screen driver is where Chapters 3, 4, and 9 meet. Open `monitor.c` and read its first two lines of data:

```
// The VGA framebuffer starts at 0xB8000.
u16int *video_memory = (u16int *)0xB8000;
```

This single declaration is a small anthology of the book:

- Treating the constant `0xB8000` as a **pointer** is **Chapter 3 §3.6** ("hardware is memory too") and **§3.10** ("casting addresses"). The cast `(u16int *)0xB8000` tells the compiler: *interpret this integer as the address of an array of 16-bit cells.*
- That the number is written in **hexadecimal** is Chapter 1 §1.7 and Chapter 9 §9.4 — hardware addresses are almost always written in hex because their bit structure matters.

- That an **address is just data** you can assign to a variable is **Chapter 1 §1.8**.

Each screen cell is a 16-bit value: the low byte is the character, the high byte is a colour attribute. Look at how a character is placed:

```
u8int  attributeByte = (backColour << 4) | (foreColour & 0x0F);
u16int attribute = attributeByte << 8;
...
location = video_memory + (cursor_y*80 + cursor_x);
*location = c | attribute;
```

Every operator here has a home in the book:

- `backColour << 4` and `attributeByte << 8` are **bit shifts** — **Chapter 9 §9.6**. Shifting the background colour into the high nibble, then the attribute byte into the high byte, assembles the 16-bit cell piece by piece.
- `foreColour & 0x0F` is a **mask** (**Chapter 9 §9.7**) that keeps only the low four bits.
- `c | attribute` uses **OR to combine** the character and its colour into one word (**Chapter 9 §9.5**), the same idiom §9.8 uses for setting flags.
- `video_memory + (cursor_y*80 + cursor_x)` is **pointer arithmetic** (**Chapter 3 §3.7-3.8**). Because `video_memory` is a `u16int *`, adding one advances by two bytes — one cell — automatically.
- `*location = ...` is a **dereference** (**Chapter 3 §3.4**) that writes directly into video memory. There is no `printf`; the write *is* the output.

The tab-handling line is a lovely bit-trick worth pausing on:

```
cursor_x = (cursor_x+8) & ~(8-1);
```

`~(8-1)` is `~0x07`, a mask with the low three bits cleared (**Chapter 9 §9.5** NOT and §9.7 masks). ANDing with it rounds the cursor down to the nearest multiple of eight — a tab stop — without a single division.

Finally, `move_cursor()` reaches back to Chapter 4:

```
outb(0x3D4, 14);
outb(0x3D5, cursorLocation >> 8);
```

Here the driver uses the `outb` from `common.c` to program the VGA hardware cursor, and `cursorLocation >> 8` (Chapter 9 §9.6) extracts the high byte to send it. Note also that `move_cursor` and `scroll` are declared `static` — **Chapter 11 §11.7** — because they are private helpers no other file should call. This is Chapter 15's "small interfaces" (§15.4) in practice: only three functions are public, and the rest are hidden.

B.7 The C Entry Point: `main.c`

`main.c` is almost anticlimactic, and deliberately so:

```c
#include "monitor.h"

int main(struct multiboot *mboot_ptr)
{
    monitor_clear();
    monitor_write("Hello, world!");
    return 0;
}
```

Two things here connect to the book:

- This `main` is **not** the C `main` you know from hosted programs. Chapter 12 §12.4 explains that in a freestanding kernel, execution begins in the startup assembly, which then *calls* a C function. The name `main` is just a convention; `boot.s` could have called it anything.
- The parameter `struct multiboot *mboot_ptr` is a pointer the bootloader fills with information about the machine. Where does its value come from? From a register, set by the assembly bootstrap according to the calling convention — the subject of **Chapter 10**, which the next section examines. (When you build, GCC warns that `struct multiboot` is declared inside the parameter list; the type is never actually dereferenced here, so it is harmless, but it is a good thread to pull on later.)

`main` calls two functions from the driver and returns. After it returns, control goes back to `boot.s`, which halts the processor.

B.8 The Bootstrap: `boot.s`

This is the file the whole port exists for. It is long, but Chapter 10 §10.12 offers the right attitude: *read assembly without fear*. Read it in the order the processor runs it, and lean on the README and DISCUSSION documents, which narrate it step by step.

The multiboot header and the entry symbol

```
[EXTERN main]
[GLOBAL start]
```

`[EXTERN main]` declares that `main` is defined elsewhere (in `main.c`), and `[GLOBAL start]` exposes `start` so the linker can use it as the entry point. This is the assembly side of **Chapter 10 §10.7** — "crossing the language boundary" — and the `extern`/`global` pairing of **Chapter 11 §11.7**. The multiboot header itself (the three `dd` values) is the handshake that lets a Multiboot loader recognize and load the kernel; it is the startup machinery Chapter 12 §12.9 describes.

Step 0 — Does this CPU support long mode?

```
    mov     eax, 0x80000001
    cpuid
    test    edx, 1 << 29            ; the LM bit
    jz      .no_long_mode
```

`test edx, 1 << 29` isolates a single bit — the long-mode flag — using the shift-and-mask thinking of **Chapter 9 §9.6-9.7**. Checking before acting is **defensive programming** (Chapter 15 §15.9): the README notes that skipping this check turns an unsupported CPU into a silent reboot loop, whereas ten lines of assembly buy you a readable error message instead.

Step 1 — Build the page tables

This is **Chapter 7** made concrete. Long mode *requires* paging (DISCUSSION.md and Chapter 7 §7.1 both stress this), so the kernel must build a page table before it can enter 64-bit mode. First it zeroes three tables:

```
    mov     edi, pml4
    mov     ecx, (4096 * 3) / 4
    xor     eax, eax
    rep     stosd
```

The three tables — `pml4`, `pdpt`, `pd` — are the top three levels of the **four-level page table** from **Chapter 7 §7.5**. Zeroing them by hand rather than trusting the loader is again defensive (Chapter 15 §15.9): a stray "present" bit in an unwritten entry would map a random page and cause an instant fault. Then it fills the page directory:

```
    mov     ecx, 512
    mov     eax, 0x83               ; present | writable | page-size (2 MiB)
    mov     edi, pd
.map_pd:
    mov     [edi], eax
    add     eax, 0x200000           ; next 2 MiB frame
    add     edi, 8                  ; next 64-bit entry
    loop    .map_pd
```

The constant `0x83` is a **page-table entry** with three flag bits set, precisely the construction of **Chapter 9 §9.10**: bit 0 (present), bit 1 (writable), and bit 7 (page-size). Setting bit 7 uses the 2 MiB "large page" shortcut described in DISCUSSION.md, so one entry maps 2 MiB directly and no fourth-level table is needed. Five hundred and twelve entries × 2 MiB = 1 GiB, **identity-mapped** — virtual address `X` maps to physical address `X` (Chapter 3 §3.13 and Chapter 7 §7.2). This is why `monitor.c` did not have to change: `0xB8000` still means the VGA buffer after paging turns on.

Steps 2-5 — Flip the switches

```
    mov     eax, cr4
    or      eax, 1 << 5             ; CR4.PAE
    mov     cr4, eax
    mov     eax, pml4
    mov     cr3, eax
    mov     ecx, 0xC0000080         ; IA32_EFER
    rdmsr
    or      eax, 1 << 8             ; EFER.LME
    wrmsr
    mov     eax, cr0
    or      eax, 1 << 31            ; CR0.PG
    mov     cr0, eax
```

Every line here is *read a register, set one bit with OR, write it back* — the read-modify-write flag pattern of **Chapter 9 §9.8**, applied to the **control registers** of **Chapter 10 §10.9**. `1 << 5`, `1 << 8`, and `1 << 31` are named positions expressed as shifts (Chapter 9 §9.6). The `CR3` load points the MMU (Chapter 7 §7.10) at the page table just built. The `rdmsr`/`wrmsr` pair reads and writes a *model-specific register*, `IA32_EFER`, whose `LME` bit announces the intent to enter long mode.

Step 6 — The far jump into 64-bit code

```
gdt64:
    dq 0                                        ; null descriptor
.code: equ $ - gdt64
    dq (1<<43) | (1<<44) | (1<<47) | (1<<53) ; executable, code/data, present, 64-bit
...
    lgdt    [gdt64.pointer]
    jmp     gdt64.code:long_mode_start
```

The GDT is a small table of 64-bit **descriptors**, and building one with OR-ed bit positions is Chapter 2's structures-as-hardware-layout idea (§2.6, where the IDT was built the same way) combined with Chapter 9's bit flags. The far jump is the one operation Chapter 10 §10.8 singled out as *impossible to write in C*: it reloads the code segment and, because the new descriptor has bit 53 (the "L" bit) set, the very next instruction runs as true 64-bit code.

Step 7 — Call `main` the 64-bit way

```
[BITS 64]
long_mode_start:
    ...
    mov     rsp, stack_top
    xor     rdi, rdi
    mov     edi, [mboot_ptr]        ; struct multiboot *mboot_ptr
```

```
    call    main
```

This is **Chapter 10 §10.4** — the **System V AMD64 calling convention** — in its purest form. Setting up `rsp` establishes the stack (Chapter 10 §10.5). Then the multiboot pointer is placed in `RDI`, because the ABI passes the first integer argument in that register, *not* on the stack the way 32-bit code did. This is exactly the register that becomes `mboot_ptr` back in `main.c` §B.7. When `main` returns, the code falls into a `hlt`/`jmp` loop that idles the processor forever.

The page tables and stack themselves live in `.bss`:

```
section .bss
align 4096
pml4:  resb 4096
pdpt:  resb 4096
pd:    resb 4096
...
stack_bottom:
    resb 16384                      ; 16 KiB kernel stack
stack_top:
```

`.bss` is the zero-initialized region introduced in Chapter 6 §6.2 ("three places where memory lives") and placed by the linker script in the next section. The 4096-byte alignment matters because page tables must sit on page boundaries.

B.9 Assembling the Pieces: the `Makefile`

The Makefile is **Chapter 11 §11.8-11.10** brought to life, with a twist from Chapter 10 and Chapter 12.

```
SOURCES=boot.o main.o monitor.o common.o

CFLAGS=-m64 -nostdlib -nostdinc -fno-builtin -fno-stack-protector \
       -fno-pie -fno-pic -ffreestanding -mno-red-zone \
       -mno-mmx -mno-sse -mno-sse2 -Wall
LDFLAGS=-T link.ld -m elf_x86_64 -z max-page-size=0x1000 -z noexecstack
ASFLAGS=-felf64

link:
    ld $(LDFLAGS) -o kernel64.elf $(SOURCES)
    objcopy -I elf64-x86-64 -O elf32-i386 kernel64.elf kernel
```

Read the flags with Chapter 11 §11.10 ("why compiler flags matter") and Chapter 12 open:

- `-ffreestanding`, `-nostdlib`, and `-nostdinc` declare the **freestanding environment** of **Chapter 12 §12.2-12.3**: no standard library, no standard headers, no assumed `main`. This is why `common.c` had to provide its own `memcpy`.

- `-fno-builtin` and `-fno-stack-protector` remove hidden helpers the compiler would otherwise insert — Chapter 12 §12.11's "avoiding hidden dependencies."
- `-mno-red-zone` and `-mno-sse` are the two kernel-specific flags the README dwells on. They have no 32-bit counterpart and, if omitted, cause bugs that do not appear until interrupts fire. They belong to the same family of subtle, environment-driven decisions Chapter 12 §12.13 warns about.

The two-line `link` rule is the port's cleverest trick. It **links as ELF64** so relocations resolve correctly, then rewrites the container to **ELF32** with `objcopy`, because Multiboot 1 loaders only parse ELF32 headers. The machine code is untouched; only the metadata changes. This is a linking subtlety beyond Chapter 11 §11.8, and the README explains why it works: every address in the kernel is below 4 GiB, so nothing is lost in translation.

The `.s.o` rule at the bottom runs `nasm $(ASFLAGS)`, with `-felf64` producing 64-bit object files (Chapter 10). The C files compile through Make's built-in rule using `CFLAGS`.

B.10 Placing the Kernel in Memory: `link.ld`

The linker script is the subject of **Chapter 11 §11.11**, and this one is short enough to read whole.

```
OUTPUT_FORMAT(elf64-x86-64)
ENTRY(start)

SECTIONS
{
    . = 0x100000;

    .multiboot : { *(.multiboot) }
    .text   ALIGN(4096) : { *(.text) }
    .rodata ALIGN(4096) : { *(.rodata*) }
    .data   ALIGN(4096) : { *(.data) }
    .bss    ALIGN(4096) : { *(COMMON) *(.bss) }
    end = .;
}
```

Three lines are worth connecting to the book:

- `ENTRY(start)` names the label `start` from `boot.s` as the entry point — the answer to Chapter 12 §12.4's question about where execution begins, and the reason `main` is *not* first.
- `. = 0x100000;` sets the load address to 1 MiB. That the kernel lives at a fixed **physical** address, and that identity mapping makes it also its **virtual** address, is Chapter 3 §3.13 and Chapter 7 §7.2. (One megabyte is `0x100000`; the multiboot

header must come first, which is why it is listed before `.text`.)
- The four sections `.text`, `.rodata`, `.data`, `.bss` are the memory regions of Chapter 6 §6.2, and placing them is the linker's job (Chapter 11 §11.11). `.bss` is where the page tables and stack from `boot.s` end up.

If you build and inspect the result, the entry symbol `start` sits at `0x101000` and `main` a little past it, near `0x101000` — a fact worth verifying yourself as an exercise in reading tools (Chapter 13 §13.12).

B.11 Building and Running

Now stop reading and *do*, as Chapter 16 §16.2 insists: build, run, observe. From the `src` directory:

```
cd 03_screen_64/src
make
qemu-system-x86_64 -kernel kernel
```

You should see `Hello, world!` in the top-left corner. Two cautions from the README, both worth internalizing:

- **Use `qemu-system-x86_64`, never `qemu-system-i386`.** The 32-bit emulator has no long mode; you will get a blank screen and no error. The README calls this "the single most common way to waste an afternoon."
- **Expect warnings during the build.** The empty string functions in `common.c` produce "control reaches end of non-void function" — that is Chapter 12's free-standing library, not a real error. Chapter 14 §14.2 reminds you that reading warnings carefully is part of debugging *before* the bug.

When it works, do not move on immediately. Chapter 16 §16.4 asks you to *predict before you run*: before changing anything, write down what you expect. Then change one thing (§16.5) and see whether the machine agrees with you.

B.12 Modifying the Kernel: Guided Experiments

Reading is not enough (Chapter 16 §16.1). The following experiments are ordered from safe to daring. For each, follow the Chapter 16 discipline: predict the outcome, change exactly one thing, rebuild, observe, and keep notes (§16.6). Where an experiment can crash the machine, that is the point — Chapter 16 §16.7 wants you to learn to break things and Chapter 14 stands ready when you do.

1. Change the message. Edit the string in `main.c` and rebuild. This touches only C and cannot break the bootstrap — a gentle confirmation that your build-and-run loop works.

2. Change the colour. In `monitor.c`, alter `foreColour` or `backColour` in `monitor_put` and `monitor_clear`. Predict the new attribute byte using Chapter 9 §9.8 before you compile, then check whether the screen matches your prediction.

3. Implement `monitor_write_hex`. The function is a stub (Chapter 12 §12.6). Fill it in using bit shifts and masks (Chapter 9 §9.6-9.7) to print a `u32int` in hexadecimal, then use it to print the address of `main` and confirm it is near `0x101000`.

4. Implement the string functions. Complete `memcpy`, `memset`, and `strcmp` in `common.c` using pointers and loops (Chapters 3 and 12). Nothing calls them yet, but writing them is the freestanding library of Chapter 12 §12.6.

5. Break the bootstrap on purpose. Comment out the line `mov cr0, eax` that enables paging, rebuild, and watch the machine fail. Then restore it and comment out `lgdt` instead. Each failure has a distinct signature; learning to recognize them now (Chapter 14 §14.12) will save you hours later. Change only one line at a time (Chapter 14 §14.16).

6. Probe the mapping's edge. The kernel identity-maps exactly 1 GiB. From C, dereference a pointer to `0x40000000` (just past the mapped region) and see what happens. This is a page fault (Chapter 7 §7.11) waiting to occur; the next tutorial chapter gives you the tools to *see* it rather than merely reboot.

B.13 When It Breaks: Debugging This Kernel

Because this kernel has no debugger of its own, Chapter 14's techniques are your instruments.

- **QEMU is your laboratory** (§14.4). Run with a monitor to inspect the machine:

  ```
  qemu-system-x86_64 -kernel kernel -monitor stdio
  ```

  Then type `info registers` and read `EFER` and `CR4` (Chapter 14 §14.15). You want `EFER` to show the long-mode bits set and `CR4` bit 5 (PAE) high — direct confirmation that steps 2-5 of the bootstrap succeeded.

- **Printing is the simplest debugger** (§14.5). Once `main` runs, `monitor_write` and your new `monitor_write_hex` become print-debugging tools. Before `main`, you have only the raw VGA writes that `boot.s` uses for its "no long mode" message — a reminder of how much the C environment gives you.

- **Read a crash by its symptoms** (§14.8, §14.12). A blank screen with no message usually means you booted with the 32-bit emulator or the far jump failed. A reboot loop means a triple fault, most often a broken page table. Match the symptom to the step.

- **Change one thing at a time** (§14.16) and **keep a journal** (§14.17). In bootstrap code, where a single wrong bit can be fatal and invisible, this discipline is not optional.

B.14 How the Whole Book Appears in One Kernel

Chapter 15 asks you to step back and see the design, not just the code. This tiny kernel is layered exactly as §15.3 recommends: the assembly bootstrap is the lowest layer, the hardware helpers (`common.c`) sit above it, the screen driver builds on those, and `main` sits on top, ignorant of everything below. Each layer exposes a small interface (§15.4) and hides its mechanism (§15.8) — the driver never knows paging is on, and `main` never knows a driver programs a VGA cursor.

For quick reference, here is where each chapter surfaces in `03_screen_64`:

Chapter	Topic	Where it appears
1	Types and portability	`common.h` typedefs; every `u16int`, `0xB8000`
2	Structures and hardware layout	the GDT descriptors in `boot.s`
3	Pointers and memory	`video_memory`, casting `0xB8000`, dereferencing
4	Inline assembly	`outb`/`inb`/`inw` in `common.c`; cursor programming
6	Where memory lives	`.data`/`.bss` sections; page tables and stack
7	Paging and virtual memory	the identity-mapped page tables in `boot.s`
9	Bit manipulation and flags	`0x83`, attribute bytes, `1 << 29`, masks
10	Linking C and assembly	`extern main`, the far jump, `RDI`, the calling convention
11	Reusable code and the build	headers, include guards, Makefile, linker script
12	Freestanding C	compiler flags, stub library functions, no real `main`
13	Reading kernel code	the reading strategy of §B.2-B.3
14	Debugging	QEMU monitor, deliberate breakage, reading registers
15	Thinking like a kernel programmer	the layered design, small interfaces
16	Learning by experiment	the modifications of §B.12

Only Chapters 5 and 8 — interrupts and the heap — are absent, because this kernel does neither yet. The next tutorial chapters add them, and when you reach that code you will already know how to read it.

Practice Exercises

Exercise 1

Without looking at §B.3, write the execution path of this kernel from GRUB's handoff to the final `hlt`, naming every file and function control passes through. Then verify it against the trace and against `boot.s`.

Exercise 2

Take the line `u16int *video_memory = (u16int *)0xB8000;` and explain each of its three ideas — the hexadecimal address, the cast, and the pointer type — citing the exact section of Chapters 1 and 3 that covers it.

Exercise 3

The bootstrap builds a page-table entry with the constant `0x83`. Write out its eight low bits, label the three that are set, and explain what each flag tells the processor. Then change the identity map to cover only 512 MiB instead of 1 GiB and predict, before building, what will break.

Exercise 4

Explain, using Chapter 10, why the line `mov edi, [mboot_ptr]` in `boot.s` corresponds to the parameter `mboot_ptr` in `main.c`. What would happen if the bootstrap placed the pointer in `RSI` instead of `RDI`?

Exercise 5

Remove `-mno-sse` from the Makefile's `CFLAGS`, rebuild, and describe what happens and why, connecting your explanation to the freestanding environment of Chapter 12. Then restore the flag.

Exercise 6

Implement `monitor_write_hex` and `monitor_write_dec` in `monitor.c`. Use them to print the address of `main` and the value of `cursor_x` after writing a message. Confirm the address is near `0x101000` and explain, using the linker script, why.

Appendix Summary

`03_screen_64` looks like a program that prints a greeting, but it is really a demonstration of everything that must exist before a greeting is possible. Its nine files divide cleanly by responsibility, and each responsibility maps onto chapters you have already read: types and portability in `common.h`, inline assembly in `common.c`, pointers and

bits in `monitor.c`, the freestanding entry point in `main.c`, the paging-and-calling-convention bootstrap in `boot.s`, and the build machinery in the Makefile and linker script.

The habits this appendix asked of you — map the directory before reading files, trace execution rather than lines, connect every construct to the concept behind it, predict before running, and change one thing at a time — are the habits Chapters 13 through 16 describe. Practiced on a kernel this small, they become second nature. Practiced on a kernel this small, they also become transferable: the next tutorial chapter is larger, but you will read it the same way.

You now have a kernel you understand completely. That is a rare and valuable thing. Break it, repair it, extend it, and measure your progress by what you can explain rather than by what compiles.

Appendix C - Task Execution

A Capstone Where Structures, Pointers, Function Pointers, and the Language Boundary Meet

Every chapter so far has taught one idea and pointed at the kernel code that needs it. This appendix does something different. It introduces almost no new C. Instead, it takes a single subsystem — the one that lets a kernel run more than one thread of control — and shows how the ideas you already know combine to build it.

The subsystem is the multitasking code of the 64-bit tutorial, found at

`64-bit/09_multitasking_64`

and its core is a single file, `task.c`, of roughly one hundred and twenty lines. That file gives the kernel a preemptive round-robin scheduler, kernel threads that each own a stack, and a voluntary `task_yield()`. It is short not because it does little, but because it rests on everything the book has already built.

Read this appendix as a capstone. Its purpose is not to teach scheduling theory — Chapter 15 §15.8 already warned that policy is a subject for other books — but to demonstrate a claim this book has made repeatedly: **when the data structure is right, the algorithm nearly disappears.** Nowhere in the tutorial is that claim proved more convincingly than here.

Wherever a topic was covered earlier, the chapter is named. Keep `task.c`, `task.h`, and `isr.h` open beside you.

Learning Objectives

After completing this capstone, you should be able to

- explain what a "thread of control" is in terms of processor state,
- recognize the register frame as a complete, restartable description of execution,
- read the `task_t` structure and the ready queue as a linked list,
- explain why, in this design, a context switch is a single `mov` instruction,
- read the three-line scheduler and say what each line does,

- explain how `create_task` manufactures a thread that has never run,
- describe how cooperative and preemptive switching reach the same code path,
- articulate why the correct data structure deleted hundreds of lines of fragile code.

C.1 Why This Is a Capstone

Look at what the execution code actually uses, and notice that you have seen all of it:

- The register frame is a **structure describing hardware** — Chapter 2, and specifically the `registers_t` type introduced in Chapter 5 §5.5.
- The scheduler receives a **pointer** to that frame so it can act on live state — Chapter 3 §3.9, and the reasoning of Chapter 5 §5.6.
- The scheduler is dispatched through a **function pointer** — Chapter 5 §5.7.
- The ready queue is a **self-referential structure forming a linked list** — the same shape as the free list of Chapter 8 §8.6.
- Each task's stack comes from the **heap allocator** — Chapter 8 §8.11.
- The switch itself lives on the **boundary between C and assembly** — Chapter 10.

No new syntax appears. What appears is fluency: function pointers that return pointers to structures, a frame that is written by assembly and read by C, a linked list walked in interrupt context. This is the book's material used at full strength, which is exactly what a *mastering* text should end on.

The spine of the appendix is a single sentence, and it is worth holding in mind from the start: *understand the frame, and the scheduler writes itself.*

C.2 What "Execution" Means to a Processor

Chapter 10 §10.10 described a context switch abstractly, as preserving Process A and restoring Process B. This appendix makes that concrete, so begin by asking what, precisely, must be preserved.

At any instant, a running thread of control is completely described by a small amount of processor state:

- the general-purpose registers, which hold its working values,
- the stack pointer, which locates its stack,
- the instruction pointer, which says what runs next,
- the flags register, which holds condition bits and the interrupt-enable flag.

That is the whole of it. If you could capture those values, freeze them, and later put them back exactly, the thread would resume as though nothing had happened — it could not tell it had ever stopped. This is the illusion Chapter 10 §10.10 named: each task believes it owns the processor.

The entire multitasking system is built on one observation about that list of state. **You have already seen every item on it saved to memory, together, in one place — by**

the interrupt machinery of Chapter 5.

C.3 The Register Frame Is the Thread

Recall from Chapter 5 §5.4 what happens when an interrupt fires. The processor pushes some state automatically, an assembly stub pushes the rest, and the C handler is handed a pointer to the result. In the 64-bit kernel that result is `registers_t`, now complete for long mode:

```
typedef struct registers
{
    // Pushed by us, in interrupt.s.
    u64int r15, r14, r13, r12, r11, r10, r9, r8;
    u64int rbp, rdi, rsi, rdx, rcx, rbx, rax;

    // Pushed by the ISR stub: interrupt number and error code (or a dummy 0).
    u64int int_no, err_code;

    // Pushed by the processor automatically.
    u64int rip, cs, rflags, userrsp, ss;
} registers_t;
```

Read this structure the way Chapter 13 §13.8 insists — data before algorithms — because everything else follows from it. Compare its fields against the list in §C.2: fifteen general-purpose registers, `rip` (the instruction pointer), `userrsp` (the stack pointer), `rflags` (the flags). Every item that describes a thread of control is here, in one contiguous block of memory.

That is the key realization, and it deserves to be stated plainly:

> A `registers_t` frame is not merely information *about* a thread. It *is* the thread, in the only form the processor cares about. Everything needed to stop a thread and later restart it is already sitting in this structure, on the stack, the moment the handler runs.

Chapter 5 §5.6 explained why the handler receives a *pointer* to the frame rather than a copy: a copy would let the handler read the state but never change what the processor restores on the way out. Under the System V AMD64 ABI (Chapter 10 §10.4), passing this 176-byte structure by value would force the caller to copy it into a fresh stack slot, and the handler would be editing a corpse. The pointer keeps the handler connected to the live frame — the one `iretq` will actually reload. Hold on to that fact; the scheduler depends entirely on it.

C.4 A Task Is Almost Nothing

If the frame already captures a thread, then a "task" needs to store shockingly little. Here is the whole of it, from `task.h`:

```
typedef struct task
{
    u32int id;              // Process ID.
    u64int rsp;             // Saved pointer to this task's registers_t frame.
    u64int kstack;          // Base of this task's kernel stack (for freeing).
    struct task *next;      // The next task in the ready queue.
} task_t;
```

Four fields. Read them with Chapter 2 in hand, because this is a structure used exactly as Chapter 2 §2.3 described — a blueprint for a small record — and with Chapter 8 in hand, because the `next` field makes it a node in a linked list, the same construction as the heap's free list (§8.6).

The most important field is `rsp`. It is not a copy of the task's registers; it is a *pointer to where that task's frame lives* on the task's own kernel stack. When a task is not running, its complete state sits frozen in a `registers_t` on its stack, and `task_t::rsp` remembers the address. The task structure is a bookmark, not a photograph.

Two globals tie the tasks together:

```
volatile task_t *current_task = 0;
volatile task_t *ready_queue  = 0;
```

`ready_queue` is the head of the linked list; `current_task` points at whichever task is running now. Both are `volatile` for the reason Chapter 3 §3.12 gave: they are changed inside an interrupt handler, asynchronously with respect to ordinary code, and the compiler must not cache them in a register across that boundary.

That is the entire data model. A list of small structures, each holding a pointer to a saved frame. Everything from here on is a consequence of getting these two paragraphs right.

C.5 The Context Switch Is a `mov`

Now the payoff. Since a task's state already lives in a frame, and the handler already holds a pointer to a frame, switching tasks cannot require *setting* registers one at a time. It requires only *choosing which frame to restore*. Watch the assembly stub do it, from `interrupt.s`:

```
irq_common_stub:
    PUSH_ALL                ; save the outgoing task's 15 registers
    mov rdi, rsp            ; RDI = pointer to the frame (ABI first argument)
    call irq_handler        ; C decides what runs next; returns a frame in RAX
    mov rsp, rax            ; <-- the entire context switch
    POP_ALL                 ; load the incoming task's 15 registers
    add rsp, 16             ; discard int_no and err_code
    iretq                   ; load its RIP, RSP, RFLAGS, CS, SS atomically
```

Read it as Chapter 10 §10.12 taught — ask what enters, what leaves, what state changes — and the shape is clear. `PUSH_ALL` finishes building the outgoing task's frame on its

stack. `mov rdi, rsp` hands the C handler a pointer to that frame, obeying the calling convention of Chapter 10 §10.4. The handler returns, in `RAX`, a pointer to *some* frame. Then one instruction does the work:

```
mov rsp, rax
```

If `RAX` holds the same frame that came in, the stub unwinds the interrupt normally and the same task resumes — this is the ordinary interrupt return of Chapter 5. But if `RAX` holds a *different* task's frame, on a *different* stack, then `POP_ALL` loads that task's registers and `iretq` loads its instruction pointer, stack pointer, and flags. The processor walks away as that other task. The switch was a single move.

This is why Chapter 10 §10.8 mattered: only `iretq`, not a C `return`, can reload `RIP`, `RSP`, and `RFLAGS` together and atomically. C cannot express the final step; assembly must. But notice how *little* assembly — one `mov` beyond the interrupt handling you already had.

The mechanism reaches into C through a function-pointer type that Chapter 5 §5.7 prepared you for, now sharpened:

```
typedef registers_t *(*isr_t)(registers_t *);
```

Read this declaration carefully, using Appendix A §A.10 if the syntax resists. In Chapter 5 a handler received a frame and returned nothing. Here a handler receives a pointer to a frame *and returns a pointer to a frame*. That return value is not decoration; it is load-bearing. It is the frame the stub moves into `RSP`. A handler that wants to change what resumes — a scheduler, a page-fault handler — now has a way to say so.

C.6 The Scheduler in Three Lines

With the frame and the task list understood, the scheduler is almost anticlimactic. Here it is in full, from `task.c`:

```
registers_t *schedule(registers_t *regs)
{
    if (regs->int_no == IRQ0)
        tick++;

    if (!current_task)
        return regs;

    // Save the frame we were handed. It lives on the outgoing task's own
    // kernel stack, so there is nothing to copy -- we just remember where it is.
    current_task->rsp = (u64int)regs;

    // Round-robin.
    current_task = current_task->next;
    if (!current_task)
```

```
        current_task = ready_queue;

    return (registers_t *)current_task->rsp;
}
```

Strip the timer bookkeeping and the guard, and three lines remain, each doing exactly one job:

- **Save.** `current_task->rsp = (u64int)regs;` records where the outgoing task's frame lives. There is no copying — the frame is already sitting on that task's stack, so the scheduler only writes down its address. This is the bookmark of §C.4 being placed.
- **Select.** `current_task = current_task->next;` advances to the next task in the list, wrapping back to `ready_queue` at the end. That single sentence is the whole of the scheduling *policy*: round-robin, nothing more. (What a real scheduler adds here — priorities, fairness, sleeping — is the subject Chapter 15 §15.8 deliberately leaves to other books.)
- **Return.** `return (registers_t *)current_task->rsp;` hands back the incoming task's frame. The stub's `mov rsp, rax` does the rest.

One line saves, one line selects, one line returns. This is the sentence from §C.1 made real: because the frame was the right data structure, the algorithm collapsed to almost nothing. Chapter 15 §15.5 called this "data first," and Chapter 13 §13.8 called it "understand data before algorithms." Here is the proof that the advice pays.

Note also the pointer casts, `(u64int)regs` and `(registers_t *)current_task->rsp`. These are the address-as-data idea of Chapter 1 §1.8 and the casting of Chapter 3 §3.10: an address is stored as an integer in the task structure and interpreted back as a frame pointer when needed.

C.7 Manufacturing a Thread That Has Never Run

The scheduler can switch to any task whose frame is ready. But a brand-new task has never been interrupted, so no frame exists for it yet. `create_task` builds one by hand — and it is the only place in the kernel that manufactures a frame from nothing, which makes it worth reading as a definition of what a thread *is*.

```
int create_task(void (*entry)(void))
{
    asm volatile("cli");

    task_t *task = (task_t *)kmalloc(sizeof(task_t));
    task->id     = next_pid++;
    task->next   = 0;

    // A fresh kernel stack, page-aligned so we can spot overruns easily.
    task->kstack = kmalloc_a(KERNEL_STACK_SIZE);
```

```
    u64int stack_top_addr = task->kstack + KERNEL_STACK_SIZE;

    // Hand-build the frame that iretq will restore the first time this task runs.
    registers_t *frame = (registers_t *)(stack_top_addr - sizeof(registers_t));
    memset((u8int *)frame, 0, sizeof(registers_t));

    frame->rip     = (u64int)entry;    // where the task begins executing
    frame->cs      = 0x08;             // kernel code segment
    frame->ss      = 0x10;             // kernel data segment
    frame->rflags  = 0x202;            // reserved bit 1, plus IF: interrupts on
    frame->userrsp = stack_top_addr;   // its own stack
    frame->rbp     = 0;                // terminate any backtrace here

    task->rsp = (u64int)frame;

    // Append to the ready queue.
    task_t *tmp = (task_t *)ready_queue;
    while (tmp->next)
        tmp = tmp->next;
    tmp->next = task;

    asm volatile("sti");
    return task->id;
}
```

Nearly every line ties back to an earlier chapter:

- `kmalloc` and `kmalloc_a` allocate the task structure and a page-aligned stack — Chapter 8 §8.11. The stack is aligned so an overflow lands on a page boundary you can detect, a small piece of the defensive thinking of Chapter 15 §15.9.
- `stack_top_addr - sizeof(registers_t)` is pointer arithmetic (Chapter 3 §3.7) placing the frame at the very top of the new stack, because a stack grows downward.
- `memset(...)` zeroes the frame; recall from Chapter 12 §12.6 that in a freestanding kernel you supply `memset` yourself. Zeroing first means every register the task starts with is a defined value.
- The assignments then write only the fields that matter. `rip` is set to `entry`, so the task begins executing at the function you passed in. `cs` and `ss` are the kernel segments from Chapter 2. `rflags = 0x202` sets the interrupt-enable bit so the task is preemptible the instant it starts — one of the flag constructions of Chapter 9 §9.8. `userrsp` points the task at its own stack. `rbp = 0` stops a debugger's backtrace cleanly at the task's origin.
- `entry` itself arrives as `void (*entry)(void)`, a function pointer (Chapter 5 §5.7) — the task's body is passed the same way an interrupt handler is registered.

When the scheduler later selects this task and the stub does `mov rsp, rax; POP_ALL; iretq`, the processor loads these hand-written values and jumps to `entry` with a clean

stack. The task cannot tell that its "resume" is actually its birth. That is the elegance: *starting* a thread and *resuming* a thread are the same operation, because both are just restoring a frame.

The `cli`/`sti` pair around the body disables interrupts while the ready queue is modified. The queue is a linked list being walked by the scheduler in interrupt context; appending to it without protection is a race, and Chapter 15 §15.9's defensive habit applies.

C.8 Turning the Running Kernel Into Task One

There is a chicken-and-egg question hiding here. The scheduler switches *between* tasks, but when the kernel first boots there are no tasks — only the ordinary thread of control running `main`. `initialise_tasking` resolves this by declaring that existing thread to be task one:

```
void initialise_tasking()
{
    asm volatile("cli");

    current_task = ready_queue = (task_t *)kmalloc(sizeof(task_t));
    current_task->id     = next_pid++;
    current_task->rsp    = 0;                  // filled in by the first interrupt
    current_task->kstack = (u64int)&stack_top; // the boot stack from boot.s
    current_task->next   = 0;

    register_interrupt_handler(IRQ0, &schedule);
    register_interrupt_handler(INT_YIELD, &schedule);

    asm volatile("sti");
}
```

Two details are worth pausing on.

First, `current_task->rsp` is left at zero. The running thread has no saved frame yet, because it has not been interrupted. That is fine: the very first timer interrupt will enter the stub, build a frame on the boot stack, and `schedule` will fill in `rsp` with `(u64int)regs` — the save line of §C.6. The kernel's original thread of control quietly acquires a frame the first time it is preempted.

Second, and central to the whole design, the scheduler is installed as an ordinary interrupt handler through a **function pointer**:

```
register_interrupt_handler(IRQ0, &schedule);
```

This is Chapter 5 §5.9 — event dispatching through a table of function pointers — used to plug scheduling into the interrupt system. `IRQ0` is the timer (defined as 32 in `isr.h`), so every timer tick now runs `schedule`, and every tick is an opportunity to switch tasks. Preemption is nothing more than the timer's handler being the scheduler.

C.9 Cooperation and Preemption Share One Path

Preemption happens *to* a task when the timer fires. Sometimes a task wants to give up the processor *voluntarily* — it has nothing to do and would rather let others run. The obvious temptation is to write a second switching routine for the voluntary case. The design refuses that temptation, and the refusal is instructive.

The scheduler already knows how to switch, but it needs an interrupt frame to do it, because the frame *is* the mechanism (§C.5). So the voluntary path simply raises an interrupt on purpose:

```
#define INT_YIELD 0x81

void task_yield()
{
    asm volatile("int %0" :: "i"(INT_YIELD));
}
```

This is inline assembly of the kind Chapter 4 §4.3 introduced, here executing a software interrupt. The `int $0x81` instruction pushes a frame exactly as a hardware interrupt would, lands in the same stub, and reaches the same `schedule` — which was registered against `INT_YIELD` right beside `IRQ0` in §C.8. Cooperation and preemption travel one code path.

Chapter 15 §15.10 valued consistency over cleverness, and this is a clean example. There is exactly one thing in the kernel that knows how to switch tasks. The timer reaches it; `yield` reaches it; a future system call could reach it. Every route funnels through the same three-line scheduler, so there is only one place to get right and only one place to look when something is wrong.

C.10 The Line Count That Proves the Point

It is worth seeing what this design replaced, because the contrast is the whole argument of the appendix.

The original 32-bit tutorial switched tasks from ordinary C code rather than from an interrupt. Standing in the middle of a function with a live stack frame, it had to *manufacture* the state an interrupt would have saved. Doing so required reading the instruction pointer with a `pop`/`jmp` trick, stuffing a magic sentinel value into a register so a resumed task could recognize itself, and a block of hand-written inline assembly to set the stack and instruction pointers. That assembly is famously miscompiled by modern GCC, which allocates one of the C variables into the very register the template uses as scratch. Around two hundred and fifty lines of such machinery — including a routine that walked the kernel stack *rewriting any value that happened to look like a stack pointer* — simply do not exist in the 64-bit redesign.

They do not exist because they were never solving a real problem. They were compensating for a missing data structure. Once the register frame is acknowledged as the

complete, addressable description of a thread, the questions those lines answered — *where do I resume? with what stack?* — are already answered, in the frame, for free. The algorithm did not get cleverer. The data got honest, and the algorithm evaporated.

This is the lesson Chapter 13 §13.8 and Chapter 15 §15.5 stated as advice. Here it is stated as arithmetic: the right structure was worth roughly two hundred and fifty lines and three genuine bugs. Internalize this, and you will reach for the data structure first for the rest of your career.

C.11 Proving It Correct, Not Merely Alive

A demonstration in which two tasks take turns printing their names *looks* like success, but Chapter 14 §14.2 warns that looking right is not being right. A context switch that quietly dropped one callee-saved register, or let two tasks' stacks overlap, would produce output that looked identical while corrupting state invisibly.

The tutorial verifies the switch the way Chapter 16 §16.4 recommends — predict a ground truth, then test against it. It computes a long, register-hungry checksum three times with interrupts disabled, capturing the answer a correct machine must produce. Then it runs the same three checksums inside three preemptible tasks and compares. If any register is lost across a switch, or any two stacks alias, a checksum changes and the mismatch is caught. A stack canary in each computation guards against silent stack corruption. When the preempted results match the uninterrupted truth bit for bit across thousands of switches, the switch is not merely alive; it is correct.

This is the empirical habit the whole book has argued for. Do not trust a subsystem because it produced plausible output once. Construct a test whose failure you would actually notice, and let the machine try to break it.

C.12 Reading This Code in the Right Order

Chapter 13 §13.6 said to trace execution rather than read files top to bottom. Applied here, the story of a single timer tick is the clearest way in:

```
timer fires (IRQ0)


interrupt stub  (interrupt.s)
          PUSH_ALL builds the outgoing frame; RDI points at it

irq_handler  (isr.c)
          looks up the handler for IRQ0 in the function-pointer table

schedule  (task.c)
          save current->rsp; advance to next; return its frame
```

```
back in the stub
          mov rsp, rax   ← now on the incoming task's stack
          POP_ALL; iretq

the next task resumes exactly where it left off
```

Follow that path once with your finger on the code, from `interrupt.s` into `isr.c` into `task.c` and back, and the subsystem stops being a collection of files and becomes a single motion. Then read `create_task` to see how a task joins the cycle, and `initialise_tasking` to see how the cycle begins. Read data first (`registers_t`, then `task_t`), mechanism second (the stub), policy last (the three lines of `schedule`). In that order, nothing is mysterious.

Common Mistakes

The most common misunderstanding is to imagine the context switch as a loop that *copies* registers from one task to another. It copies nothing. Each task's registers live in a frame on that task's own stack; switching only changes which stack the processor is standing on. If you find yourself looking for the code that saves fifteen registers into a task structure, stop — that code does not exist, and its absence is the point.

A second mistake is forgetting that the scheduler runs in interrupt context (Chapter 5 §5.10). Anything it touches — the ready queue above all — can be touched between any two instructions of ordinary code. Modifying that list without disabling interrupts, as `create_task` is careful to do, invites a corruption that appears only under load.

A third is treating `current_task` and `ready_queue` as ordinary variables and wondering why a debug build behaves differently from an optimized one. They are `volatile` for a reason (Chapter 3 §3.12); removing that qualifier lets the compiler cache a stale value across the interrupt that changed it.

Finally, students sometimes try to make a resumed task "start" differently from how it "resumes." There is no difference. `create_task` builds a frame, and the first schedule restores it exactly as any later schedule would. If starting and resuming were different operations, the design would have failed.

Practice Exercises

Exercise 1

Without looking at §C.5, explain in your own words why `mov rsp, rax` performs a context switch. Your explanation should mention where the outgoing task's state is, where the incoming task's state is, and what `iretq` does. Then verify it against `interrupt.s`.

Exercise 2

The scheduler's selection policy is a single line: `current_task = current_task->next;`. Describe, in one sentence each, three different policies you could implement by changing only that line, and identify which field you would have to add to `task_t` for each. Do not implement them; the exercise is to see that policy and mechanism are separable (Chapter 15 §15.8).

Exercise 3

In `create_task`, the frame is placed at `stack_top_addr - sizeof(registers_t)`. Draw the new task's stack and mark where the frame sits. Explain what would go wrong if the frame were placed at `task->kstack` (the *bottom* of the stack) instead.

Exercise 4

`create_task` sets `frame->rflags = 0x202`. Using Chapter 9, identify which bit enables interrupts and explain what would happen to a task created with `rflags = 0x002` instead. Would it ever be preempted? Would it ever yield?

Exercise 5

Add a `switches` counter to `schedule()` that increments on every call, and print it alongside `tick`. Predict, before running, whether it will grow faster than, slower than, or at the same rate as `tick`, and explain your prediction using §C.8 and §C.9. Then run it and reconcile.

Exercise 6

Deliberately break the switch: swap two adjacent `pop` instructions inside `POP_ALL` in `interrupt.s`. Predict what the simple two-task demo will do, then predict what the checksum verification of §C.11 will do. Run both. Explain why one notices the bug and the other does not, connecting your answer to Chapter 14 §14.2.

Appendix Summary

Execution is where the book's threads are tied together. A thread of control is nothing more than processor state, and that state already lives in one place — the `registers_t` frame the interrupt machinery of Chapter 5 builds on every interrupt. A task is a small structure holding a pointer to its saved frame and a link to the next task, a linked list of the same shape you built in Chapter 8. Because the frame is complete, switching tasks is not a matter of setting registers but of choosing which frame to restore, and that choice reduces to one instruction, `mov rsp, rax`, reached through a function pointer that returns a frame. The scheduler that drives it is three lines: save, select, return.

The deeper lesson is the one this appendix exists to deliver. The 64-bit redesign is short — and correct on a modern toolchain where the original is neither — not because its author was cleverer, but because the data structure was chosen honestly. Recognizing the frame as the thread erased hundreds of lines and three real bugs at a stroke. That is the habit worth carrying out of this book: before writing the algorithm, get the data right, and much of the algorithm will turn out to be unnecessary.

You began this book learning what a `u64int` is. You end it watching a linked list of saved frames become the illusion of many programs running at once, built entirely from parts you already understood. That is what mastering C for kernel development looks like.