FREE SAMPLE

Build Your Own Operating System

A hands-on book, from boot sector to graphical desktop

Noah Parsons

Companion to the NexusOS reference kernel

Sample: Preface + Chapter 1 + Chapter 2.
Full book at leanpub.com/build-your-own-os

About this sample

This is a free sample of *Build Your Own Operating System*, a hands-on book that takes you from a 512-byte BIOS boot sector to a graphical desktop running on real hardware, paired with **NexusOS**, a roughly three-thousand-line x86-64 reference kernel you can clone, build, and modify.

What you have here is the front of the book: the Preface, Chapter 1 (what an operating system is and what we will build), and Chapter 2 (your toolchain and the first booting OS, a 512-byte program that prints `OK` on a virtual machine). Chapter 2 ends at the natural cliffhanger: a working boot sector, and the question of why exactly `[BITS 16]`, `0x7C00`, and `0xAA55` are what they are. Chapter 3 answers that, and the remaining ten chapters build the rest of the kernel on top.

If this front matter pulls you in, the full book is at leanpub.com/build-your-own-os. NexusOS itself is on GitHub at github.com/GuiloScion/NexusOS, MIT-licensed, with the source pinned to the `v1.0-book` tag.

Preface

This book is for the person who has spent years using operating systems and one day caught themselves wondering: *what actually happens between the power button and the cursor blinking on screen?* It is the book I wanted when I started writing one. It takes you from the boot sector to a graphical desktop on real hardware: bootloader, 64-bit kernel, memory management, preemptive scheduler, disk driver, read-only filesystem, framebuffer graphics, compositing window manager. Twelve chapters of working subsystems, with a thirteenth that maps what to build after.

What this book is, and what it is not

This is a brisk tour, not a doorstop. Most of the giants in this space (OSTEP, the xv6 book, the OSDev wiki taken as a whole) give you many hundreds of pages; this gives you about ninety. The trade-off is deliberate. Each chapter introduces one subsystem, shows the actual code that makes it work, names the two or three traps that took the reference implementation a weekend to find on real hardware, and then sets you loose with exercises. If you want the formal treatment of concurrency or virtual memory at the level of a graduate textbook, those books are excellent and free; read them alongside this one. What you have here is a compact, opinionated path from `[BITS 16]` to a draggable terminal window, told by someone who has done it.

This book is also a **companion to a working reference kernel**, not a from-scratch tutorial. The code is **NexusOS** (~3,000 lines of C and assembly); it is inlined where it matters and lives in full on disk. You can clone it, build it, boot it, and modify it. The exercises ask you to *change* the reference (add a syscall to the IDT, swap in a best-fit allocator, drop one task to ring 3) rather than to retype the entire kernel from a blank file. You get a baseline that boots, and the book teaches you to read it, break it productively, and extend it.

How to read this book

Read in order the first time. Each chapter assumes the subsystem from the previous one. After every chapter, build NexusOS, boot it, and notice what is now visibly different from the last run. When a chapter shows a code listing, the line numbers in the caption match what you would see if you opened the file in your editor; treat them as a pointer for when you want to see the function in its surroundings. The exercises at the end of each chapter are the single most important part of the book; doing one or two is worth more than reading three chapters ahead.

You will, by the end, have joined a small club of people who actually know what happens between the power button and the cursor. That is the whole point.

Introduction: what is an OS, and what will we build?

What an operating system *is*

When you press the power button, the CPU starts executing instructions with no help: no files, no windows, no notion of a "program," no `printf`. An **operating system** is the program that takes that bare machine and turns it into something usable. It manages the CPU, the memory, and the devices, and it provides services (*run this program; read this file; show a window*) to everything else.

At its core an OS does five jobs:

- **Boot:** get the CPU from its primitive power-on state into a sane environment where your code can run.
- **Manage memory:** keep track of physical RAM and hand out chunks safely to whoever asks.
- **Manage the CPU:** make multiple things appear to run at once on a chip that can only do one thing at a time.
- **Talk to devices:** keyboard, screen, disk, mouse, network. This is what *drivers* are.
- **Provide services:** a filesystem, a shell, a graphical user interface, eventually a way to run third-party programs.

Each of these is one or two chapters of this book.

What we will build

We will build a small but real OS for the **x86-64** PC. The reference implementation, **NexusOS**, boots on QEMU and on actual hardware, and includes:

- a from-scratch BIOS bootloader (no GRUB, no shims);
- a 64-bit kernel with interrupt handling;
- physical and virtual memory managers, and a heap;
- a preemptive scheduler with mutexes, semaphores, and condition variables;
- a disk driver and a read-only filesystem;
- a framebuffer graphics stack with a compositing window manager and mouse cursor.

The destination is a desktop with draggable windows and a terminal, running on the OS *you* will understand top to bottom:

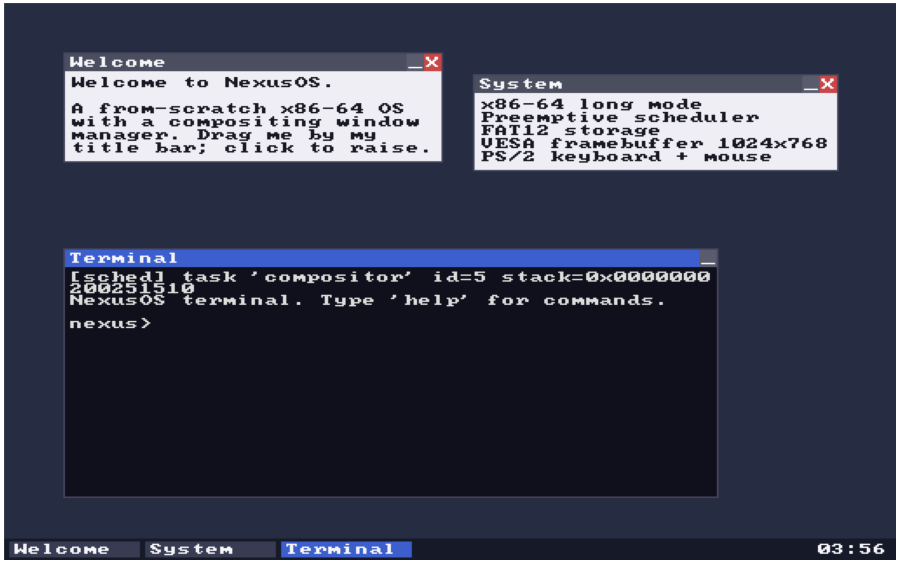

Figure 1.1. The NexusOS desktop after Chapter 11. By the time you reach the window manager chapter, you will recognise every pixel: the desktop background is one gfx_fill, the title bars are rectangles, the cursor is a 19-row ASCII bitmap, and the terminal text is the same character grid the kernel started writing to in Chapter 5.

Why x86-64, why BIOS?

Two pragmatic choices.

x86-64 is the architecture in most desktops and laptops, and QEMU emulates it perfectly. You can develop without risking your real machine,

iterate in under a second, and attach a debugger that single-steps the kernel. When you finally test on metal, you have a working baseline to compare against.

BIOS / legacy boot (rather than UEFI) is dramatically simpler to start with. A few hundred bytes of assembly get you booting. UEFI is a worthy later project; doing it second, after you understand what UEFI is hiding, is much easier than doing it first.

The concepts (booting, memory, scheduling, drivers) transfer to any architecture (ARM, RISC-V). The specific instructions change; the ideas do not. Chapter 13 says a little more about porting.

The mindset

Three things make OS development approachable.

It is incremental. You will always have something that boots. Each chapter adds one capability. If it breaks, you changed one thing, so you know where to look. The instinct to "rewrite from scratch when stuck" rarely pays off here; the instinct to bisect (comment out the most recent change, build, run, narrow it down) almost always does.

The emulator is your friend. QEMU boots your OS in under a second, can show you registers, and can attach a debugger. You will iterate fast. Use it. The first time you try your kernel on real hardware should not also be the first time you have looked closely at any of its output.

Crashes are normal and informative. A triple-fault reboot or a frozen screen is not failure; it is the CPU's only way of telling you that something went very wrong before your handler was installed. We will teach you to read it. By Chapter 6 you will have an exception handler that prints the faulting instruction pointer, the registers, and (for page faults) the address that was being accessed. After that, your kernel will *tell you* what happened.

What you need to know

- **C:** structs, pointers, bit operations. We use plain C, no standard library, no libc.

- **A little assembly.** We explain the x86 as we go. You do not need to be fluent. Chapter 4 is the densest assembly in the book (the mode-switch ritual); by Chapter 8 you will have written and read enough to recognise a context switch when you see one.
- **The command line.** Build with `make`, run with `qemu-system-x86_64`. If you have never used either, Chapter 2 walks through both.

If a term is unfamiliar (*real mode? GDT? page table?*) flip to the Glossary at the back. We define jargon the first time it appears, and the Glossary is there for quick lookup.

A note on the reference code

Every chapter inlines the relevant excerpt of NexusOS so you can read both at once: the prose explains the *idea*, the code shows a *complete working version*. The line numbers in the captions match what you would see if you opened the file in your editor, so when you want to see how a function fits into its surroundings, you know exactly where to look.

The full source tree is small (about three thousand lines of C and assembly) and worth reading end-to-end as you progress. It is structured so that each subsystem lives in its own pair of files (`pic.h` / `pic.c`, `pmm.h` / `pmm.c`, and so on), and the entry points are the function names you will meet here.

You now know what we are building, roughly how the parts fit, and what you need to bring to the keyboard. Next: setting up the tools, and writing the smallest OS that can run.

Exercises

1. **Read a kernel boot log.** On a Linux machine, run `dmesg | head -200`. Even if much of it is unfamiliar, see whether you can spot the high-level shape: detection of CPUs, memory, controllers, then mounting a filesystem. Make a one-paragraph note of three things you recognise. By Chapter 9 you will know what most of those lines mean.

Hint: Look for lines mentioning "ACPI", "PCI", "ata", "EXT4", "Setting up". The kernel is announcing each subsystem coming online, which is exactly the pattern we'll adopt in Chapter 5.

2. **Map the layers.** On paper, draw the stack of an OS as you currently understand it, with the hardware at the bottom and a graphical application at the top. Leave space between layers. As each chapter ends, come back and write the chapter's contribution into the diagram. By Chapter 13 the picture should be complete.

Hint: A useful starting set of layers is *hardware* → *firmware* → *bootloader* → *kernel* → *drivers* → *system calls* → *libraries* → *application*. Don't worry about being precise; this is a thinking exercise.

3. **Predict the failure modes.** Before reading any further, write down three things you suspect could go wrong when an OS starts up on a real, unknown PC. Keep the list. We'll revisit it in Chapter 12, where you'll find that real firmware fails in ways your imagination would not have invented.

Hint: Think about what an OS has to assume about the machine before it has talked to any devices. Those assumptions are the cracks.

Ready? Let's set up your tools and get something booting.

Your toolchain and first boot

Before writing any operating-system code, you need tools to assemble, compile, link, and run it. However, you do **not** need a special cross-compiler for x86-64. Your normal host tools work, as long as you tell them not to assume an operating system. This chapter sets up the toolchain, explains what each tool does, and ends with the smallest possible OS: a 512-byte program that prints `OK` on a virtual machine.

The tools, and what each one does

Tool	Job
`nasm`	Assembler. Turns `.asm` files into machine code.
`gcc`	C compiler, used in *freestanding* mode (no standard library, no startup code).
`ld`	Linker. Places code and data at the right physical addresses.
`objcopy`	Extracts a flat binary from the linked ELF (the form the bootloader can load).
`qemu-system-x86_64`	Emulates a PC. Boots your OS in under a second.
`mtools`	Builds a FAT disk image without root (for the filesystem chapter).
`gdb`	Debugger. Optional but invaluable; QEMU exposes a GDB stub for free.

Installing

- **Linux (Debian / Ubuntu):**

```
sudo apt install nasm gcc binutils make mtools qemu-system-x86 gdb
```

- **macOS:**

```
brew install nasm qemu mtools x86_64-elf-binutils
```

- **Windows:** Use **WSL** (Ubuntu) and follow the Linux instructions.

"Freestanding": the key idea

Normally, when you compile `hello.c` with `gcc`, the resulting program runs *on* an OS. It links against libc, it assumes `main`, it assumes a stack and a heap and stdin and stdout. We have none of that. Our kernel runs on the bare machine; libc would have nowhere to call into.

So we compile **freestanding**: no standard library, no startup files, no assumptions. NexusOS's `Makefile` passes the following flags to `gcc`, and reading them is a quick tour of "what assumptions are we removing?":

```
-ffreestanding              # no hosted environment / standard library
-fno-pic                    # absolute addresses; we control where things live
-fno-stack-protector        # no canary; no libc support function to call
-fno-builtin                # don't replace memcpy/strlen with libc calls
-mno-red-zone               # the SysV red zone is unsafe with interrupts
-mno-mmx -mno-sse -mno-sse2 # don't use vector regs; we haven't set them up
-m64 -std=c11               # 64-bit, ISO C11
```

You do not have to memorise these. Their effect, taken together, is to say: *"there is no operating system here; emit plain 64-bit code that touches nothing it shouldn't."*

How a build comes together

An OS image is built in five steps:

1. **Assemble** the bootloader to a flat binary: `nasm -f bin boot.asm -o boot.bin`.
2. **Compile** each kernel `.c` or `.asm` file to an object file.
3. **Link** the object files with a *linker script* that fixes where the kernel lives in memory. NexusOS puts the kernel at physical `0x10000`. The linker script that does this is short enough to read in one sitting; we'll meet it in Chapter 5.
4. `objcopy -O binary` the linked ELF into a flat binary the bootloader can load verbatim.

5. **Concatenate** bootloader + kernel into one disk image.

NexusOS's `Makefile` automates all of this and auto-discovers any `.c` or `.asm` you drop into the `kernel/` directory. The day-to-day commands are short:

```
make rungui   # GUI in a QEMU window
make run      # headless, serial only
make debug    # QEMU paused (-s -S), with gdb attached
```

The `debug` target is the one you will come to love. It launches QEMU with the CPU frozen at the very first instruction and waits for gdb to connect, so you can single-step your bootloader if you want.

Your first boot: a 512-byte "OK"

The smallest useful OS is a **boot sector**: 512 bytes the BIOS loads from disk and runs. Here is one that prints `OK` to the screen and halts forever. Call it the "hello, world" of bare metal.

Listing 2.1. `boot.asm`*: a complete 512-byte program that the BIOS will boot.*

```
[BITS 16]                       ; the CPU starts in 16-bit "real mode"
[ORG  0x7C00]                   ; the BIOS loads us at this address

    mov ah, 0x0E                ; BIOS teletype: print AL to the screen
    mov al, 'O'
    int 0x10
    mov al, 'K'
    int 0x10

.hang:
    hlt
    jmp .hang

times 510 - ($ - $$) db 0       ; pad to 510 bytes
dw 0xAA55                       ; the boot signature the BIOS looks for
```

Build and run it:

```
nasm -f bin boot.asm -o boot.bin
qemu-system-x86_64 -drive format=raw,file=boot.bin
```

A QEMU window opens, the BIOS flashes briefly, and then you see `OK`. **You have just written and run an operating system's first code on a (virtual) bare machine.** Everything else in this book is adding layers to that program.

Three details to internalise before moving on:

- `[BITS 16]` tells the assembler we are writing 16-bit code, because the CPU starts in 16-bit mode at power-on. We will not leave 16-bit mode for several thousand instructions; Chapter 4 is the journey to 64-bit.
- `[ORG 0x7C00]` tells the assembler our code expects to be loaded at physical address `0x7C00`. This is a hardware-defined convention; the BIOS *will* load us there. If we lied to the assembler about where we live, every absolute address (string pointers, jump targets) would be wrong.
- `dw 0xAA55` at the end is the "boot signature." The BIOS refuses to boot any sector whose last two bytes are not `0x55 0xAA`. Get this wrong and the BIOS skips your sector entirely. (This is also why the BIOS chooses *one* device to boot from: it's looking for that signature.)

If you want to understand *why* `[BITS 16]`, `0x7C00`, and `0xAA55` are what they are, that is exactly the next chapter.

Exercises

1. **Print a longer message.** Modify the boot sector so it prints `HELLO, OS!` instead of `OK`. Be careful: each `int 0x10` only prints one character, so you'll need either a string of `mov al, …; int 0x10` pairs or, much better, a loop. Use the address of a string as a pointer in `si`, `lodsb` to fetch the next byte, test for the terminating zero, and loop until you hit it.

Hint: `lodsb` reads the byte at `[ds:si]` into `al` and increments `si`. The classic loop body is `lodsb; test al, al; jz .done; mov ah, 0x0E; int 0x10; jmp .loop`. Don't forget to declare the string with `db "HELLO, OS!", 0` somewhere after your code.

2. **Crash on purpose.** Replace the `hlt` loop with `mov ax, [0]` and remove every other instruction except the print. Build and run. You should see your message print, then the QEMU window reboot continuously. Congratulations: you have triggered a triple fault. Now restore the `hlt` and notice the difference. The lesson: when there is no interrupt handler, even a tiny mistake reboots the machine silently. Chapter 6 fixes this; for now just observe how informationless the failure is.

Hint: The dereference of `[0]` is fine in *real mode* (it reads the BIOS interrupt vector table), so to actually triple-fault you may need to use a

privileged instruction like `mov cr0, eax` which is invalid before you set things up. Either way, the goal is to see the silent reboot.

3. **Measure the size.** Add `db "X"` lines after the print instructions until `nasm` fails with an "overflow" error. You'll discover how many bytes of code you have to work with after the BIOS print routine. (Spoiler: not many.) This forces the discipline of bootloader writing: real bootloaders chain. The first 512 bytes load a second-stage loader, which loads the kernel.

Hint: The `times 510 - ($ - $$) db 0` directive pads to 510 bytes; if your code exceeds 510 bytes, `$ - $$` overflows the `times` count. You'll get a clear NASM error pointing at that line.

4. **Boot from a real USB stick (optional).** Write `boot.bin` raw to a spare USB stick with `dd if=boot.bin of=/dev/sdX bs=512` on Linux, or *balenaEtcher* with the "Flash from file" option. Boot a PC from it. If you get `OK`, you have shipped your first operating system to hardware. If you get nothing, your firmware is probably UEFI-only; we'll come back to legacy boot in Chapter 12.

Hint: On any PC younger than about 2018, you may need to enable **Legacy / CSM** boot in the BIOS setup and possibly disable Secure Boot. Be patient; this is the same dance Chapter 12 spells out in full.

End of free sample

This is the end of the free sample. You have the booting boot sector from Chapter 2. The rest of the book builds the OS on top of it:

Chapter 3 explains *why* the BIOS hands control to your code at `0x7C00` and what tools it leaves you in real mode. Chapter 4 walks the CPU from 16-bit real mode through 32-bit protected mode into 64-bit long mode, with the actual page-table ritual the kernel uses. Chapter 5 lands you in C with two output channels (VGA text and the COM1 serial port) and a linker script. Chapter 6 builds the IDT, the PIC, the timer, and the keyboard, and turns the kernel's exceptions into readable panic dumps. Chapter 7 is the memory manager: PMM, VMM, heap, all in one self-contained stack. Chapter 8 adds a preemptive scheduler with mutexes and semaphores. Chapter 9 talks to a disk. Chapter 10 draws pixels. Chapter 11 composites windows. Chapter 12 is the hard, honest chapter on real hardware. Chapter 13 maps what to build after.

The full book is at leanpub.com/build-your-own-os. It is roughly one hundred pages of focused practitioner content, thirteen chapters of working subsystems, a glossary, a full-source appendix, and the back-of-the-book references that make OS development tractable.

Thank you for reading the sample. If it landed for you, the most useful thing you can do beyond buying the book is to actually build one of NexusOS's subsystems and send a pull request.