Contents

OS

A Custom x86_64 Operating System Kernel

Author: Will **Architecture:** x86_64 (64-bit) **Codebase:** ~10,800 lines (C and Assembly)

Table of Contents

1. Introduction

What is OS?

OS is a hobby operating system kernel built entirely from scratch for the x86_64 architecture. Unlike projects that rely on existing kernels or heavy libraries, this OS implements every component from the ground up: the bootloader handoff, memory allocation, interrupt handling, device drivers, a TCP/IP networking stack, a graphical window manager, and an interactive shell.

The project serves multiple purposes:

1. **Educational**: Demonstrates how operating systems actually work at the lowest level
2. **Practical**: Provides a working system that can boot on real hardware or emulators
3. **Portfolio**: Showcases systems programming skills across multiple domains

Why Build an OS from Scratch?

Modern operating systems like Linux or Windows contain millions of lines of code and decades of accumulated complexity. By building a simpler system from scratch, we can understand the fundamental concepts without getting lost in edge cases and legacy compatibility.

This OS answers questions like: - How does a computer transition from BIOS to running your code? - How does the CPU know what to do when you press a key? - How do network packets actually get from the wire to your application? - How does a window manager decide which window is on top?

Architecture Overview

The system is organized in layers, with each layer building on the one below:

User Space contains the shell, window manager, and applications. These components interact with the kernel through system calls.

Kernel Space contains the core operating system: memory management, interrupt handling, device drivers, the filesystem, and the networking stack. The kernel has direct access to hardware.

Hardware includes the CPU, RAM, disk, network card, and display. The kernel abstracts these into usable interfaces.

Feature Summary

Component	What It Does
Boot	Transitions CPU from 32-bit to 64-bit mode, sets up memory
Memory	Provides dynamic allocation (malloc/free) for the kernel
Interrupts	Handles hardware events (keyboard, mouse, network, disk)
PCI Driver	Discovers hardware devices on the system bus

Component	What It Does
e1000 Driver	Communicates with Intel Gigabit Ethernet adapters
Network Stack	Implements Ethernet, IP, ARP, ICMP, TCP, and UDP
Filesystem	Provides file and directory operations
Graphics	Draws pixels, shapes, and text to the screen
Window Manager	Manages overlapping windows with mouse interaction
Shell	Accepts and executes user commands
Processes	Loads and runs ELF binaries

---

2. Boot Process

The Challenge of Booting

When you press the power button, your computer doesn't immediately start running your operating system. Instead, it goes through several stages:

1. **BIOS/UEFI** initializes hardware and looks for something bootable
2. **Bootloader** (GRUB in our case) loads the kernel into memory
3. **Kernel initialization** sets up the CPU, memory, and devices

The CPU starts in a primitive 16-bit "real mode" dating back to the 1970s. Modern operating systems need 64-bit "long mode" for full access to memory and modern instructions. Getting from one to the other is surprisingly complex.

Multiboot2: Speaking the Bootloader's Language

Rather than writing our own bootloader (a complex project in itself), we use GRUB, which understands the Multiboot2 specification. This is a contract between bootloaders and kernels: if we provide a specific header, GRUB will load us and give us information about the system.

Our Multiboot2 header requests a framebuffer (graphics mode) at 1280x720 with 32 bits per pixel. GRUB sets this up before transferring control to us.

When GRUB jumps to our code, we're in 32-bit protected mode. The CPU can't directly jump to 64-bit mode—we need to:

1. Verify the CPU supports 64-bit mode
2. Set up page tables for virtual memory
3. Enable paging and long mode
4. Load a 64-bit Global Descriptor Table
5. Far jump to 64-bit code

CPU Capability Detection

Before attempting to enter 64-bit mode, we must verify the CPU supports it. Not all x86 processors do—particularly older or embedded ones.

We check three things:

CPUID Support: The CPUID instruction lets us query CPU features. We detect its presence by attempting to flip bit 21 in the EFLAGS register. If we can change this bit, CPUID is available.

Extended CPUID: Long mode information is in the "extended" CPUID functions. We call CPUID with EAX=0x80000000 to check if these extended functions exist.

Long Mode Support: Finally, we call CPUID with EAX=0x80000001 and check bit 29 of EDX. If set, the CPU supports 64-bit long mode.

If any check fails, we halt with an error code displayed on screen.

Page Tables: The Foundation of Virtual Memory

Virtual memory is one of the most important concepts in operating systems. It lets each program think it has its own private memory space, starting at address 0. The CPU's Memory Management Unit (MMU) translates these "virtual" addresses to "physical" addresses in RAM.

x86_64 uses a four-level page table hierarchy:

- **PML4** (Page Map Level 4): The root table, pointed to by the CR3 register
- **PDPT** (Page Directory Pointer Table): Second level
- **PD** (Page Directory): Third level
- **PT** (Page Table): Fourth level, points to actual memory pages

Each level contains 512 entries, and each entry can point to the next level or directly to memory. For simplicity, we use 2MB "huge pages" which skip the final PT level, reducing the number of tables we need.

We set up "identity mapping" where virtual address X maps to physical address X. This is the simplest possible mapping and works well for a kernel that needs direct hardware access. We map the first 4GB of memory, which covers all RAM and memory-mapped hardware devices.

Entering Long Mode

With page tables prepared, we can enable 64-bit mode through a specific sequence:

1. **Load CR3**: Point the CPU to our PML4 table
2. **Enable PAE**: Set bit 5 in CR4 to enable Physical Address Extension
3. **Set LME**: Write to the EFER Model-Specific Register to enable Long Mode Enable
4. **Enable Paging**: Set bit 31 in CR0 to turn on paging

At this point, we're technically in "compatibility mode"—a transitional state. To fully enter 64-bit mode, we must load a 64-bit GDT and perform a far jump to a 64-bit code segment.

The 64-bit GDT

The Global Descriptor Table describes memory segments. In 64-bit mode, segmentation is mostly disabled (everything is flat), but we still need a minimal GDT with at least a code segment. Our 64-bit GDT contains:

- A null descriptor (required, never used)
- A code segment descriptor with the "long mode" bit set

After loading this GDT and jumping to our 64-bit code segment, we're finally running in full 64-bit mode and can call our C kernel.

Kernel Entry Point

The C function `kernel_main()` is where the real OS begins. It initializes subsystems in a careful order:

1. **Interrupts**: Set up the IDT so the CPU knows how to handle hardware events
2. **Input devices**: Enable keyboard and mouse
3. **Disk**: Initialize ATA controller for storage access
4. **Filesystem**: Mount the root filesystem
5. **PCI**: Scan for hardware devices
6. **Network**: Initialize the network card and protocol stack
7. **Processes**: Set up the process table and syscall interface
8. **Graphics**: Initialize the framebuffer and cursor
9. **Window Manager**: Prepare for GUI operations
10. **Shell**: Enter the interactive command loop

The order matters. For example, the filesystem needs the disk driver, and network initialization needs the heap (memory allocation) to already be working.

---

3. Memory Management

Why Dynamic Memory Matters

Programs need to allocate memory at runtime. A web browser doesn't know in advance how many tabs you'll open or how large each page will be. The kernel itself needs dynamic memory for buffers, data structures, and device drivers.

In C, this is typically `malloc()` and `free()`. Since we can't use the C standard library (there's no operating system beneath us to provide it!), we must implement our own allocator.

The Heap: Where Dynamic Memory Lives

Our heap is a contiguous region of memory that starts right after the kernel binary ends. The linker script exports a symbol `_kernel_end` marking this location. We align this to a 4KB page boundary and allocate 32MB for the heap.

The heap is managed as a linked list of blocks. Each block has a header containing:

- **Magic number** (0xDEADBEEF): Used to detect memory corruption
- **Size**: How many bytes of usable space this block contains
- **Used flag**: Whether the block is currently allocated
- **Next/Previous pointers**: Links to adjacent blocks

Allocation Strategy: First Fit with Splitting

When `kmalloc(size)` is called, we walk the block list looking for the first free block that's large enough. This is called "first fit"—simple and reasonably efficient.

If we find a block much larger than needed, we split it into two blocks: one for the allocation and one containing the leftover space. This prevents wasting memory. We only split if there's room for a new header plus at least 16 bytes of usable space.

All allocations are aligned to 8 bytes. This is important because some CPU instructions require aligned memory access, and it simplifies the allocator.

```
void *kmalloc(size_t size) {
    if (size == 0) return NULL;
    if (!heap_start) heap_init();

    // Round up to 8-byte alignment
    size = (size + 7) & ~7;

    // Find first free block that fits
    heap_block_t *block = find_free_block(size);
    if (!block) return NULL;  // Out of memory

    // Split if block is much larger than needed
    split_block(block, size);

    block->used = 1;
    return (void *)((uint8_t *)block + sizeof(heap_block_t));
}
```

Deallocation: Freeing and Coalescing

When `kfree(ptr)` is called, we find the block header (located just before the user data pointer) and mark it as free.

A naive allocator would stop there, but this leads to fragmentation. Imagine allocating and freeing many small blocks—you'd end up with lots of tiny free blocks that can't satisfy larger requests, even if there's enough total free memory.

We solve this with coalescing: after freeing a block, we check if the adjacent blocks are also free. If so, we merge them into one larger block. This keeps the heap from becoming fragmented over time.

```
void kfree(void *ptr) {
    if (!ptr) return;

    // Find the block header
    heap_block_t *block = (heap_block_t *)((uint8_t *)ptr - sizeof(heap_block_t));

    // Validate magic number to detect corruption
    if (block->magic != HEAP_MAGIC) {
        serial_print("kfree: corrupted heap!\n");
        return;
    }
```

```
    // Check for double-free
    if (!block->used) {
        serial_print("kfree: double free!\n");
        return;
    }

    block->used = 0;
    merge_blocks(block);  // Coalesce with neighbors
}
```

Aligned Allocation for Hardware

Some hardware requires memory at specific alignments. DMA descriptors for the network card must be 128-byte aligned. We provide `kmalloc_aligned()` which allocates extra space and returns an address within that space that meets the alignment requirement.

Debugging Memory Issues

Memory bugs are notoriously difficult to find. Our allocator includes several safety features:

- **Magic numbers**: If a block's magic number is wrong during free, the heap is corrupted
- **Double-free detection**: Freeing an already-free block indicates a bug
- **Statistics**: `heap_stats()` reports total, used, and free memory for monitoring

---

4. Interrupt Handling

What Are Interrupts?

When you press a key, how does the CPU know? It can't constantly poll every device—that would waste cycles. Instead, hardware devices send interrupt signals that temporarily pause the current code, run a handler, and resume.

Interrupts are essential for: - **Keyboard/Mouse**: Input events - **Disk**: Completion of read/write operations - **Network**: Packet arrival - **Timer**: Preemptive multitasking (not yet implemented)

The Interrupt Descriptor Table (IDT)

The IDT tells the CPU what code to run for each interrupt. x86_64 supports 256 interrupt vectors:

- **0-31**: CPU exceptions (divide by zero, page fault, etc.)
- **32-47**: Hardware IRQs (after PIC remapping)
- **48-255**: Software interrupts and user-defined

Each IDT entry is 16 bytes containing the handler address (split across multiple fields due to historical reasons), the code segment selector, and attribute flags.

We populate all 256 entries with handler stubs. Most just acknowledge the interrupt and return, but specific handlers (keyboard, mouse, network) do real work.

The PIC: Routing Hardware Interrupts

The 8259 Programmable Interrupt Controller routes hardware interrupt requests (IRQs) to the CPU. There are actually two PICs in a "master/slave" configuration handling IRQs 0-15.

The Remapping Problem: By default, the master PIC maps IRQs 0-7 to interrupts 8-15. This conflicts with CPU exceptions! IRQ 0 (timer) would collide with interrupt 8 (double fault). We must reprogram the PIC to use different interrupt numbers.

We remap: - IRQs 0-7 → Interrupts 0x20-0x27 (master PIC) - IRQs 8-15 → Interrupts 0x28-0x2F (slave PIC)

Now hardware interrupts don't conflict with CPU exceptions.

Interrupt Handler Flow

When an interrupt occurs:

1. CPU saves the current instruction pointer and flags
2. CPU looks up the handler in the IDT
3. CPU jumps to the handler
4. Handler saves registers it will modify
5. Handler does its work (read keyboard buffer, process network packet, etc.)
6. Handler sends "End of Interrupt" (EOI) to the PIC
7. Handler restores registers
8. Handler executes IRETQ to return

The EOI is critical. The PIC won't send more interrupts until it receives EOI. If you forget it, the keyboard stops working after the first keypress!

```
void c_default_handler(int interrupt_num) {
    // Handle hardware interrupts (0x20-0x2F)
    if (interrupt_num >= 0x20 && interrupt_num < 0x30) {
        // If from slave PIC, acknowledge it first
        if (interrupt_num >= 0x28) {
            outb(PIC2_COMMAND, 0x20);  // EOI to slave
        }
        outb(PIC1_COMMAND, 0x20);  // EOI to master
    }
}
```

Specialized Handlers

Different devices need different handling:

Keyboard Handler: Reads the scancode from port 0x60, translates it to ASCII, and adds it to an input buffer the shell reads from.

Mouse Handler: Reads movement deltas and button state from port 0x60 (PS/2 mice share the keyboard controller). Updates cursor position.

Network Handler: Signals that packets are available. The actual packet processing happens in the main loop to avoid spending too long in the interrupt handler.

5. Driver Architecture

The PCI Bus: Discovering Hardware

Modern PCs use the PCI (Peripheral Component Interconnect) bus to connect expansion cards and onboard devices. Before we can talk to the network card, we need to find it.

PCI devices are identified by: - **Bus number** (0-255): Which PCI bus - **Device number** (0-31): Which slot on that bus - **Function number** (0-7): Multi-function devices have multiple functions

Each device has a 256-byte configuration space containing: - Vendor ID and Device ID (who made it, what model) - Class and subclass (what type of device) - Base Address Registers (BARs): Where the device's memory or I/O ports are mapped

We scan all possible bus/device/function combinations. If the vendor ID isn't 0xFFFF (meaning "no device"), we've found something and record its information.

```
void pci_scan(void) {
    for (int bus = 0; bus < 256; bus++) {
        for (int device = 0; device < 32; device++) {
            for (int func = 0; func < 8; func++) {
                uint32_t id = pci_read(bus, device, func, 0);
                uint16_t vendor = id & 0xFFFF;

                if (vendor != 0xFFFF) {
                    // Found a device! Store its info
                    pci_device_t *dev = &devices[count++];
                    dev->vendor_id = vendor;
                    dev->device_id = (id >> 16) & 0xFFFF;
                    // Read BARs, class code, etc.
                }
            }
        }
    }
}
```

The Intel e1000 Network Driver

The e1000 is Intel's Gigabit Ethernet adapter family. QEMU emulates the 82540EM variant, making it perfect for OS development—real hardware behavior with easy debugging.

Finding the Device: We search PCI for vendor 0x8086 (Intel) and device 0x100E (82540EM). The device's BAR0 contains its Memory-Mapped I/O (MMIO) base address.

MMIO: Instead of using I/O ports, the e1000 maps its registers into the physical address space. Reading from address `mmio_base + 0x0000` reads the device control register; writing to `mmio_base + 0x0008` modifies the status register.

MAC Address: Every network card has a unique 48-bit MAC address. The e1000 stores it in EEPROM or the RAL/RAH registers. We read it during initialization for use in Ethernet frames.

Descriptor Rings: The Heart of Network I/O

Network cards use DMA (Direct Memory Access) to transfer packets without CPU involvement. The driver sets up "descriptor rings"—circular buffers of descriptors, each pointing to a packet buffer.

Receive (RX) Ring: We allocate 32 descriptors, each pointing to a 2KB buffer. We tell the hardware where the ring is (RDBAL/RDBAH registers), how long it is (RDLEN), and where we've processed up to (RDH/RDT).

When a packet arrives, the hardware: 1. Finds the next available descriptor (at the tail) 2. DMAs the packet into that descriptor's buffer 3. Sets the descriptor's status to "done" 4. Optionally raises an interrupt

Our driver checks for completed descriptors, copies data out, and advances the tail to give that descriptor back to hardware.

Transmit (TX) Ring: Similar structure, but we fill in the buffers. When we want to send a packet: 1. Copy data into the next TX buffer 2. Set the descriptor's length and command flags (like "this is the last fragment") 3. Advance the tail pointer 4. Hardware DMAs the data to the wire

```
int e1000_send_packet(const void *data, size_t length) {
    uint16_t cur = e1000_dev.tx_cur;
    e1000_tx_desc_t *desc = &e1000_dev.tx_descs[cur];

    // Wait for descriptor to be available
    while (!(desc->status & E1000_TXD_STAT_DD)) { }

    // Copy data to buffer
    memcpy(e1000_dev.tx_buffers[cur], data, length);

    // Configure descriptor
    desc->length = length;
    desc->cmd = E1000_TXD_CMD_EOP | E1000_TXD_CMD_IFCS | E1000_TXD_CMD_RS;
    desc->status = 0;

    // Tell hardware there's a new packet
    e1000_dev.tx_cur = (cur + 1) % NUM_TX_DESC;
    e1000_write(E1000_TDT, e1000_dev.tx_cur);

    return length;
}
```

---

6. Networking Stack

The TCP/IP Model

Network communication is organized in layers. Each layer has a specific job and talks to the layers above and below it:

Layer	Protocol	Purpose
Application	HTTP, DNS	User-facing services
Transport	TCP, UDP	Reliable/unreliable delivery
Network	IP, ICMP	Routing between networks
Link	Ethernet, ARP	Local network delivery
Physical	e1000 driver	Bits on the wire

When you send data, it flows down: your application gives data to TCP, TCP adds its header and gives it to IP, IP adds its header and gives it to Ethernet, Ethernet adds its header and gives it to the driver.

When data arrives, it flows up: the driver gives a frame to Ethernet, Ethernet strips its header and gives the payload to IP, and so on.

Ethernet: The Link Layer

Ethernet frames have a simple structure: - **Destination MAC** (6 bytes): Who should receive this - **Source MAC** (6 bytes): Who sent this - **EtherType** (2 bytes): What protocol is inside (0x0800 = IPv4, 0x0806 = ARP) - **Payload** (46-1500 bytes): The actual data - **CRC** (4 bytes): Error detection (hardware handles this)

We handle two EtherTypes: ARP for address resolution and IPv4 for regular traffic.

ARP: Finding MAC Addresses

IP addresses are logical, assigned by administrators. MAC addresses are physical, burned into hardware. To send an IP packet, we need the destination's MAC address.

ARP (Address Resolution Protocol) asks "who has IP address X?" by broadcasting to all devices on the local network. The owner responds "I have X, my MAC is Y." We cache these responses to avoid asking repeatedly.

```
int arp_request(uint32_t target_ip) {
    // Build ARP request packet
    arp_header_t *arp = ...;
    arp->operation = ARP_REQUEST;
    arp->sender_ip = our_ip;
    arp->sender_mac = our_mac;
    arp->target_ip = target_ip;
    arp->target_mac = 00:00:00:00:00:00;  // We don't know yet

    // Broadcast it (destination MAC = FF:FF:FF:FF:FF:FF)
    return eth_send(broadcast_mac, ETH_TYPE_ARP, arp, sizeof(*arp));
}
```

IP: The Network Layer

IPv4 packets contain: - **Version/Header Length**: Always 0x45 for standard IPv4 - **Total Length**: Size of the entire packet - **TTL**: Decremented by each router, prevents infinite loops

- **Protocol**: What's inside (1=ICMP, 6=TCP, 17=UDP) - **Checksum**: Verifies header integrity - **Source/Destination IP**: Where it's from and going

The checksum is calculated as the one's complement sum of all 16-bit words in the header. If the receiver's calculation doesn't match, the packet is discarded.

Routing Decision: Before sending, we check if the destination is on our local network (by comparing against our netmask). If yes, we ARP for that IP directly. If no, we send to our gateway, which will forward it.

ICMP: Ping and Diagnostics

ICMP (Internet Control Message Protocol) provides network diagnostics. The most famous is ping: send an "echo request," receive an "echo reply." This verifies network connectivity and measures round-trip time.

Our ping implementation: 1. Build an ICMP echo request with a sequence number 2. Send it via IP 3. Start polling for incoming packets 4. If we receive an echo reply with matching ID, it worked 5. If timeout expires, report failure

```
int ping(uint32_t dest_ip, int count) {
    for (int i = 0; i < count; i++) {
        icmp_send_echo_request(dest_ip, 1, i + 1);

        // Wait for reply with timeout
        int received = 0;
        for (int j = 0; j < 30000 && !received; j++) {
            net_process_packet();
            if (icmp_reply_received) received = 1;
        }

        if (received) printf("Reply from %s: seq=%d\n", ip_str, i+1);
        else printf("Request timeout: seq=%d\n", i+1);
    }
}
```

TCP and UDP: Transport Layer

UDP is simple: just source/destination ports and a checksum. It's unreliable (packets can be lost or reordered) but fast. Good for DNS queries or streaming.

TCP is complex: it provides reliable, ordered delivery. It uses sequence numbers to detect loss and reorder packets, acknowledgments to confirm receipt, and window sizes for flow control. Our implementation handles basic TCP but not all edge cases.

---

7. Virtual Filesystem

Abstracting Storage

Different filesystems (FAT, ext4, NTFS) have completely different on-disk formats, but applications don't want to care. They want to `open()`, `read()`, `write()`, and `close()` files regardless of where they're stored.

The Virtual Filesystem (VFS) provides this abstraction. It defines a common interface that all filesystems implement. When you open `/foo/bar.txt`, the VFS:

1. Parses the path into components (`foo`, `bar.txt`)
2. Looks up `foo` in the root directory
3. Looks up `bar.txt` in `foo`
4. Returns a file handle you can read from

VFS Nodes

Each file or directory is represented by a `vfs_node_t` structure containing:

- **Name**: The file's name
- **Flags**: Is it a file or directory? Permissions?
- **Size**: How many bytes of data
- **Operations**: Function pointers for read, write, open, close, etc.

The function pointer approach is key to the abstraction. When you call `vfs_read(node, ...)`, you're actually calling whatever `node->read` points to. For a tmpfs node, that reads from RAM. For a SimpleFS node, that reads from disk. The caller doesn't know or care.

```
typedef struct vfs_node {
    char name[128];
    uint32_t flags;
    uint32_t size;

    // Operations - each filesystem implements these
    int (*read)(struct vfs_node *, uint32_t offset, uint32_t size, uint8_t *buf);
    int (*write)(struct vfs_node *, uint32_t offset, uint32_t size, uint8_t *buf);
    void (*open)(struct vfs_node *, uint32_t flags);
    void (*close)(struct vfs_node *);
    struct vfs_node *(*readdir)(struct vfs_node *, uint32_t index);
    struct vfs_node *(*finddir)(struct vfs_node *, const char *name);
} vfs_node_t;
```

Path Resolution

Given a path like `/home/user/file.txt`, we need to walk the directory tree:

1. Start at the root directory
2. Look up "home" in root's children
3. Look up "user" in home's children
4. Look up "file.txt" in user's children
5. Return the final node

This is `vfs_resolve_path()`. It splits the path on `/` and repeatedly calls `finddir()` to traverse the tree.

Filesystem Implementations

tmpfs: A memory-based filesystem. Files are just allocated buffers. Fast but not persistent—everything is lost on reboot. Useful for temporary files.

SimpleFS: A simple persistent filesystem stored on disk. It has a superblock (metadata), an inode table (file metadata), and data blocks (file contents). More complex but survives reboots.

---

8. Graphics & Window Manager

Framebuffer Basics

Our display is a grid of pixels. At 1280x720, that's 921,600 pixels. Each pixel is 32 bits (8 bits each for red, green, blue, and alpha/padding). The framebuffer is simply this pixel data in memory.

The Multiboot2 information tells us where the framebuffer is mapped and its dimensions. Writing a color value to `framebuffer[y * width + x]` immediately changes that pixel on screen.

Drawing Primitives: Everything complex is built from simple operations:

- `put_pixel(x, y, color)`: Set one pixel
- `draw_line(x0, y0, x1, y1, color)`: Bresenham's line algorithm
- `fill_rect(x, y, w, h, color)`: Fill a rectangular area
- `draw_circle(cx, cy, r, color)`: Midpoint circle algorithm
- `draw_string(text, x, y, color)`: Render text using a bitmap font

Window Manager Architecture

A window manager handles multiple overlapping windows. Each window has:

- **Position and size**: Where it is and how big
- **Title**: Displayed in the title bar
- **Framebuffer**: The window's content (separate from the screen)
- **Z-order**: Which windows are in front of others
- **Flags**: Is it visible? Focused? Movable?

Windows are stored in a list sorted by z-order. When rendering, we draw from back to front so front windows obscure back ones.

Per-Window Framebuffers

Each window has its own framebuffer, separate from the screen. Applications draw to their window's framebuffer. The window manager then composites all window framebuffers to the screen, adding decorations (title bar, borders, close button).

This separation means: - Applications don't need to know about other windows - We can add effects (transparency, shadows) without changing applications - Windows can be partially off-screen without issues

Mouse Handling and Window Dragging

The window manager processes mouse events:

1. **Click in title bar**: Start dragging, or if on close button, close the window
2. **Click in window content**: Focus the window, pass click to application
3. **Click on desktop**: Unfocus all windows

Dragging uses an XOR outline technique: we draw the window outline by XORing pixels (flipping their colors). To erase it, we XOR again, restoring the original. This is fast and doesn't require redrawing the screen.

When the mouse button is released, we move the window to its new position and do a full redraw.

```
void wm_handle_mouse(int32_t x, int32_t y, uint8_t buttons) {
    if (drag_window && (buttons & LEFT_BUTTON)) {
        // Update drag outline
        draw_xor_outline(old_x, old_y, width, height);  // Erase old
        draw_xor_outline(new_x, new_y, width, height);  // Draw new
    }

    if (button_released && drag_window) {
        draw_xor_outline(outline_x, outline_y, w, h);  // Erase outline
        window_move(drag_window, outline_x, outline_y);
        drag_window = NULL;
        force_full_redraw();
    }

    if (button_pressed) {
        window_t *win = get_window_at(x, y);
        if (win && in_title_bar(win, x, y)) {
            if (in_close_button(win, x, y)) {
                window_destroy(win);
            } else {
                start_drag(win, x, y);
            }
        }
    }
}
```

---

9. Shell & User Interface

Command-Line Interface

The shell is the user's primary interface. It displays a prompt, reads commands, executes them, and repeats. This simple loop is the foundation of all command-line interfaces.

Each command is defined by: - **Name**: What the user types - **Description**: Help text - **Function**: What to do when invoked

Commands receive arguments parsed from the input line, just like C's `main(int argc, char **argv)`.

Argument Parsing

When the user types `ping 10.0.2.2`, we need to split this into: - `argv[0]` = "ping" - `argv[1]` = "10.0.2.2" - `argc` = 2

The parser walks through the command buffer, finding words separated by spaces. Each word becomes an argument. Consecutive spaces are skipped.

Built-in Commands

Command	Purpose
`help`	List available commands
`clear`	Clear the screen
`ls`	List directory contents
`cat`	Display file contents
`touch`	Create an empty file
`rm`	Delete a file
`write`	Write text to a file
`ping`	Test network connectivity
`ifconfig`	Show/set network configuration
`pci`	List PCI devices
`wget`	Fetch content from a web server
`view`	Open an image in a window
`heap`	Display memory statistics
`draw`	Draw shapes on screen

The Main Loop

The shell's main loop does more than just process commands. It also:

1. **Polls keyboard**: Check for new input, process characters
2. **Polls mouse**: Update cursor position, handle clicks
3. **Renders windows**: If any window is dirty, redraw it
4. **Processes network**: Check for incoming packets

This cooperative approach works because we're single-threaded. A more sophisticated OS would use interrupts and separate threads, but this is simpler and sufficient for our needs.

---

10. Process Management

Why Processes?

So far, the kernel does everything directly. But we want to run user programs—separate binaries that don't have access to kernel internals. Processes provide:

- **Isolation**: Programs can't interfere with each other

- **Abstraction**: Programs don't need to know about hardware
- **Resource management**: The kernel controls memory, CPU time, etc.

The Process Control Block

Each process is described by a Process Control Block (PCB) containing:

- **PID**: Unique process identifier
- **State**: New, ready, running, blocked, or terminated
- **CPU state**: Saved registers when the process isn't running
- **Memory**: Stack pointer, heap limits, etc.

When switching between processes, we save the current process's registers to its PCB and restore the next process's registers from its PCB. This is called a context switch.

ELF Loading

User programs are compiled to ELF (Executable and Linkable Format), the standard binary format on Linux. An ELF file contains:

- **ELF header**: Magic number, architecture, entry point
- **Program headers**: Describe segments to load into memory
- **Section headers**: Describe sections (code, data, etc.)

To run an ELF: 1. Parse the ELF header to verify it's valid 2. For each loadable segment, allocate memory and copy data 3. Set up a stack for the process 4. Jump to the entry point

System Calls

User programs can't directly access hardware—they run in a restricted CPU mode. To do anything useful (read files, send network packets), they must ask the kernel through system calls.

We use interrupt 0x80 for syscalls. The program puts a syscall number in RAX and arguments in RDI, RSI, RDX, etc., then executes `int 0x80`. The kernel's syscall handler examines RAX to decide what to do, does it, and returns the result in RAX.

---

11. Build System

Cross-Compilation

We can't use the system's normal GCC—that produces Linux executables. We need a cross-compiler targeting x86_64 with no operating system assumptions ("freestanding").

The toolchain includes: - **x86_64-elf-gcc**: Cross-compiler - **x86_64-elf-ld**: Cross-linker - **nasm**: Assembler for x86_64

Key compiler flags: - `-ffreestanding`: Don't assume a hosted environment - `-mno-red-zone`: Disable the red zone (required for interrupt handlers) - `-mno-sse`: Don't use SSE registers (we haven't set them up)

The Linker Script

The linker script controls memory layout:

```
ENTRY(start)
SECTIONS {
    . = 1M;                     /* Start at 1MB (below is reserved) */

    .text : { *(.text) }        /* Code */
    .rodata : { *(.rodata) }    /* Read-only data */
    .data : { *(.data) }        /* Initialized data */
    .bss : { *(.bss) }          /* Uninitialized data */

    _kernel_end = .;            /* Heap starts here */
}
```

The kernel loads at 1MB because addresses below are used by BIOS, video memory, etc. The `_kernel_end` symbol is used by the heap allocator.

Creating the Bootable Image

After linking, we have a raw kernel binary. To make it bootable:

1. Copy kernel.bin into an ISO structure under `/boot/`
2. Create a GRUB configuration file specifying our kernel
3. Run `grub-mkrescue` to build a bootable ISO

The result is `kernel.iso`, which can boot in QEMU or be burned to a CD/USB for real hardware.

Running in QEMU

```
qemu-system-x86_64 \
    -cdrom kernel.iso \
    -m 256M \
    -device e1000,netdev=net0 \
    -netdev user,id=net0
```

This boots our ISO with 256MB RAM and an emulated Intel e1000 network card connected to QEMU's user-mode networking (provides DHCP and internet access through the host).

---

12. Conclusion

What We've Built

OS demonstrates that operating systems aren't magic. They're software following logical rules:

- **The boot process** is a carefully orchestrated transition through CPU modes
- **Memory management** is bookkeeping: tracking which bytes are allocated
- **Interrupts** let hardware get the CPU's attention
- **Drivers** are translators between the kernel's abstractions and hardware's quirks

- **The network stack** is layers of protocols, each adding its own header
- **The filesystem** maps names to data on storage
- **The window manager** composites rectangles and handles input
- **The shell** reads text, parses it, and calls functions

Each component is understandable in isolation. The complexity comes from their interactions, but even that follows rules.

Learning Outcomes

Building this OS teaches:

- **Low-level programming**: Assembly, bit manipulation, hardware registers
- **Memory management**: Allocation strategies, fragmentation, virtual memory
- **Concurrency**: Interrupt handling, reentrancy, race conditions
- **Networking**: Protocol layering, byte ordering, checksums
- **Systems design**: Abstraction, interfaces, modularity

Future Directions

Possible enhancements: - **Preemptive multitasking**: Timer-based context switching - **Virtual memory**: Per-process address spaces, demand paging - **More filesystems**: FAT32, ext2 for USB drives - **USB support**: USB keyboards, mice, storage - **Sound**: Audio output through AC97/HDA - **SMP**: Multiple CPU cores

Final Thoughts

Operating systems are the foundation everything else runs on. Understanding them deeply makes you a better programmer regardless of what you build. Every high-level abstraction eventually becomes system calls, interrupts, and memory accesses.

This project shows that building an OS is achievable. Start small, add features incrementally, and always understand what your code is actually doing. The computer is deterministic—if something doesn't work, there's a reason, and you can find it.

Project Statistics

- **Lines of Code**: ~10,800
- **Languages**: C (~9,800 lines), x86_64 Assembly (~1,000 lines)
- **Major Subsystems**: 10
- **Shell Commands**: 20+
- **Supported Hardware**: x86_64 CPUs, Intel e1000 NICs, PS/2 keyboards/mice, ATA disks

---

OS - A journey into the heart of computing