Silo OS

Complete Project Overview

Registry: [HOWL-COMP-3-2026]

Series Path: [HOWL-COMP-1-2026] → [HOWL-COMP-2-2026] → [HOWL-COMP-3-2026] → [HOWL-COMP-4-2026] → [HOWL-COMP-5-2026]

Parent Framework: [HOWL-COMP-1-2026]

DOI: 10.5281/zenodo.18655396

Date: February 2026

Domain: Software Architecture / Systems Engineering / Real-Time Computing

Status: Architectural Blueprint for Independent Implementation

AI Usage Disclosure: Only the top metadata, figures, refs and final copyright sections were edited by the author. All paper content was LLM-generated using Anthropic's Claude 4.5 Sonnet. Claims made by AI are often inaccurate in terms of magnitude, but not correctness and connections. Performance is never simple, but the Silo system is tuned to batch processing arrays of structs that are sequential in memory.

What This Is

Silo OS is a modern operating system being built from scratch for x86-64 computers. It boots using UEFI firmware (the modern replacement for old BIOS) and is written entirely in the Zig programming language version 0.15.1.

The Two-Part System

The project has two separate programs that work together. First is a bootloader - a small UEFI application that runs when the computer starts up. Its job is to load the actual operating system kernel from disk into memory and hand control over to it. The kernel is the main operating system that manages all the hardware and provides services.

Why This Project Exists

This is a learning and research project to understand how operating systems work at the lowest level. Unlike most OS projects that use C or assembly language, this one uses Zig, which is a newer systems programming language that aims to be safer and easier to work with while still giving direct hardware access.

The Major Technical Challenge

The compiler version being used (Zig 0.15.1) has bugs in one of its newer features. When you try to embed raw assembly instructions directly in Zig code, the compiler generates broken machine code. The workaround is to write all assembly in separate files and link them together. This affected about a dozen different parts of the system - interrupt handlers, hardware port access, and thread switching.

What The System Can Do

The kernel initializes in eight stages, setting up different subsystems one at a time. It can talk to the CPU's interrupt system, manage memory, control disk drives (both older SATA and newer NVMe), handle keyboards and mice, display graphics, and most impressively has a complete network stack. This means it can send and receive packets, get IP addresses automatically, resolve domain names, and serve web pages.

Current Progress

The bootloader is fully working. It successfully starts up, communicates with the UEFI firmware to access graphics and the filesystem, and can display status messages. The next step is to have it actually read the kernel file from disk, understand its format (ELF executable), put it in the right place in memory, and jump to it.

The kernel itself has all its code written - interrupt handling, memory management, device drivers for disks and network cards, a TCP/IP network stack, threading system, and basic security features. It just hasn't been loaded and started yet.

Why Version Numbers Matter

Zig changes significantly between versions. Code that works in one version often breaks in another. This project is locked to exactly version 0.15.1 because the build configuration, standard library APIs, and compiler behavior all changed before and after this version.

The Development Environment

Development happens in Windows using WSL2 (Windows Subsystem for Linux). The code is built on the Linux side, and testing uses QEMU, which is a program that simulates a complete computer. This lets you boot and test an operating system without needing to restart your actual computer or use physical hardware.

Testing Procedure

After building, the bootloader and kernel files are copied to a directory structure that mimics a boot drive. QEMU loads a UEFI firmware file (OVMF) and points it at this directory. The virtual machine boots, the firmware finds the bootloader, the bootloader runs and prints status messages. Right now it successfully reports finding the graphics system and filesystem. Soon it will load the kernel.

What Makes This Different

Most hobby operating systems either use legacy BIOS booting (old method) or if they use UEFI, they're written in C or Rust. This project uses modern UEFI exclusively and leverages Zig's features like compile-time execution and integrated build system. The network stack is particularly unusual for a hobby OS - most stop at basic graphics and keyboard input.

The Architecture Philosophy

The system is designed in layers. The bootloader knows only about UEFI services and loading files. The kernel is completely independent - it receives a memory map and framebuffer info from the bootloader but doesn't depend on UEFI at all. Hardware drivers are isolated subsystems. The network code is a proper layered stack - physical layer (Ethernet), network layer (IP), transport layer (TCP/UDP), and application layer (HTTP).

Why Assembly Files Exist

Normally you'd write inline assembly for things like reading hardware ports or handling interrupts. Due to the compiler bug, there are three assembly files: one for port input/output operations, one containing all 48 interrupt handler entry points, and one for switching between threads. These are pure assembly language, assembled separately, and linked into the kernel.

What Still Needs Work

The immediate next step is completing the bootloader's kernel loading routine. It needs to open the kernel file, read its ELF header, parse the segment descriptions, copy each segment to the correct memory address, then exit UEFI services and jump to the kernel's entry point. After that, the kernel needs testing - does each initialization stage work? Can it actually talk to hardware? Does the network stack function?

Long-term Vision

The goal isn't to replace Linux or Windows. It's to have a minimal but functional operating system that demonstrates modern OS concepts - UEFI booting, proper interrupt handling, virtual memory, device drivers, networking, and multitasking. It's an educational project that happens to be sophisticated enough to do real networking.

Key Design Decisions Made

Use UEFI exclusively, no legacy support. Target only x86-64, no 32-bit compatibility. Use Zig's self-hosted compiler backend instead of LLVM. Make the kernel completely freestanding with no dependencies. Implement a real network stack instead of just basic I/O. Use cooperative multitasking instead of preemptive scheduling. Keep graphics simple - just framebuffer rendering, no GPU acceleration.

The File Organization

Source files are organized by subsystem. Core files handle booting and initialization. Hardware drivers each get their own file. Network protocols are separate files that build on each other. There's a security layer for authentication and encryption. Everything compiles together through a single build script that outputs the two executables.

How Another Developer Would Continue

They'd need Zig 0.15.1 installed, QEMU for testing, and the OVMF firmware file. The immediate task is implementing kernel loading in the bootloader. After that, testing each kernel subsystem in isolation. The network stack probably needs the most work since it hasn't been tested at all yet. Long term, adding a filesystem beyond FAT32 and implementing preemptive multitasking would make it more capable.

Silo OS - Project Documentation

Overview

Silo OS is a bare-metal x86_64 operating system written in Zig 0.15.1. It consists of a UEFI bootloader and a freestanding kernel with networking, storage, and threading capabilities.

Critical Constraints

- **Zig Version**: 0.15.1 ONLY (breaking changes between versions)
- **No LLVM**: Uses Zig's self-hosted x86 backend (`kernel.use_llvm = false`)
- **UEFI Only**: No legacy BIOS support
- **Target**: x86_64-freestanding-none for kernel, x86_64-uefi-msvc for bootloader

Architecture

Boot Flow

```
UEFI Firmware → BOOTX64.efi → kernel (ELF) → 8-stage init → main loop
```

Component Structure

1. **Bootloader** (`src/boot.zig`): UEFI application, loads kernel ELF, provides memory map and framebuffer
2. **Kernel** (`src/kernel.zig`): Bare-metal OS with 8-stage initialization

Kernel Subsystems

- **CPU**: IDT, APIC, interrupts, exceptions
- **Memory**: Physical/virtual memory management
- **Storage**: AHCI (SATA), NVMe disk drivers
- **Network**: E1000 NIC, TCP/IP stack (ARP, DHCP, DNS, HTTP, ICMP, UDP)
- **Peripherals**: PS/2 keyboard/mouse, HDA audio, XHCI USB
- **Threading**: Cooperative multitasking
- **Graphics**: Framebuffer, basic rendering
- **Security**: SiloSec authentication/encryption layer

Critical Zig 0.15.1 Issue: Inline Assembly Bug

The Problem

Zig 0.15.1's self-hosted x86 backend has bugs with inline assembly, specifically:

1. **Port I/O**: `inb`/`outb`/`inl`/`outl` instructions fail with "InvalidInstruction" errors
2. **Cross-function labels**: ISR stubs can't jump to shared handlers

The Solution: External Assembly Files

All inline assembly moved to external `.s` files:

src/port_io.s - x86 port I/O:

```
.globl outb, inb, outl, inl

outb:

mov %sil, %al

mov %di, %dx

outb %al, %dx

ret
```

src/isr_stubs.s - Interrupt handlers (48 ISRs):

```
.globl isrDivideByZero, isrPageFault, isrTimer, ...
```

```
isrDivideByZero:

push $0

push $0

jmp isrCommon

isrCommon:

# Save registers, call interruptHandler, restore, iretq
```

src/context_switch.s - Thread context switching:

```
.globl contextSwitch

contextSwitch:

# Save old context (rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8-r15, rip, rfl

# Restore new context

# Jump to new RIP
```

Files That Use External Assembly

All these files declare `extern fn` and link to assembly:

- `timer.zig` - outb/inb
- `apic.zig` - outb/inb
- `ahci.zig` - outl/inl
- `ps2.zig` - outb/inb
- `nvme.zig` - outl/inl
- `pci.zig` - outl/inl
- `idt.zig` - All ISR functions, lidt
- `threading.zig` - contextSwitch

Build System (build.zig)

Key Configuration

```
// Bootloader - UEFI target
```

```
const bootloader = b.addExecutable(.{
.name = "BOOTX64",
.root_module = b.createModule(.{
    .root_source_file = b.path("src/boot.zig"),
    .target = uefi_target, // x86_64-uefi-msvc
    .optimize = .ReleaseSafe,
}),
});
// Kernel - Freestanding target
const kernel = b.addExecutable(.{
.name = "kernel",
.root_module = b.createModule(.{
    .root_source_file = b.path("src/kernel.zig"),
    .target = kernel_target, // x86_64-freestanding-none
    .optimize = .ReleaseSafe,
}),
});
kernel.use_llvm = false;  // CRITICAL: Must use self-hosted backend
kernel.use_lld = false;
kernel.addAssemblyFile(b.path("src/context_switch.s"));
kernel.addAssemblyFile(b.path("src/isr_stubs.s"));
kernel.addAssemblyFile(b.path("src/port_io.s"));
kernel.setLinkerScript(b.path("src/linker.ld"));
```

Build Output

```
build/esp/
├── EFI/
│   ├── BOOT/
│   │   └── BOOTX64.efi  (bootloader)
│   └── SILO/
│       └── kernel        (ELF executable)
```

UEFI Bootloader Details

Entry Point

```
pub fn main() uefi.Status {
const bs = uefi.system_table.boot_services.?;
// Load kernel, setup memory, exit boot services, jump to kernel
return .Success;
}
```

UEFI Protocol Usage (Zig 0.15.1 Specific)

Correct pattern for locateProtocol:

```
const gop = bs.locateProtocol(
std.os.uefi.protocol.GraphicsOutput,  // Protocol TYPE (not &Protocol.guid
null,
) catch |err| {
// Handle error
};
```

NEVER use `&Protocol.guid` - the wrapper handles this internally.

Bootloader Responsibilities

1. Get Graphics Output Protocol (GOP) for framebuffer
2. Get Simple File System Protocol to read kernel
3. Load kernel ELF from `/EFI/SILO/kernel`
4. Parse ELF headers and load segments into memory
5. Get final memory map
6. Exit UEFI boot services
7. Jump to kernel entry point (from ELF `e_entry`)

Kernel Details

Entry Point & Linker Script

The kernel uses `src/linker.ld` which defines:

- Entry point (typically `_start`)
- Memory layout (starts at 1MB: 0x100000)
- Section placement (.text, .data, .bss, .rodata)

8-Stage Initialization (kernel.zig)

```
pub fn kernelMain() noreturn {
initCPU();         // IDT, GDT, interrupts
initMemory();      // Physical/virtual memory
initHardware();    // PCI, APIC, timers
initStorage();     // AHCI, NVMe
initPeripherals(); // PS/2, HDA, XHCI
initNetwork();     // NIC, TCP/IP stack
initThreading();   // Cooperative scheduler
mainLoop();        // Event loop
}
```

Interrupt Handling

IDT Setup (idt.zig):

- 256 IDT entries
- Exceptions (0-31): Division by zero, page fault, general protection, etc.
- IRQs (32-47): Timer, keyboard, mouse, disk, network
- All handlers are in `isr_stubs.s`
- `interruptHandler()` in Zig dispatches to appropriate handlers

Critical Fix:

Changed ISR function type from `callconv(.naked)` to `callconv(.c)` for extern functions:

```
extern fn isrTimer() callconv(.c) void;

extern fn isrKeyboard() callconv(.c) void;

// etc.
```

Testing with QEMU

Requirements

- QEMU for Windows or WSL2
- OVMF UEFI firmware

Running (WSL2)

```
# Install dependencies

sudo apt install qemu-system-x86 ovmf

# Build

cd /mnt/c/Users/Geoff/work/os_zig

zig build

# Create bootable directory structure

mkdir -p ../os_zig_build/esp/EFI/BOOT
```

```
cp build/esp/EFI/BOOT/BOOTX64.efi ../os_zig_build/esp/EFI/BOOT/
cp build/esp/EFI/SILO/kernel ../os_zig_build/esp/EFI/BOOT/
# Run with QEMU
cd ../os_zig_build
qemu-system-x86_64 \
  -bios OVMF.fd \
  -hda fat:rw:esp \
  -m 512M \
  -serial stdio
```

Expected Boot Sequence

```
UEFI Firmware messages...
BdsDxe: loading Boot0002 "UEFI QEMU HARDDISK"...
Silo OS Bootloader
GOP OK
FS OK
[Kernel loads...]
```

Common Pitfalls & Solutions

1. "InvalidInstruction" Errors

Cause: Inline assembly in Zig 0.15.1's self-hosted backend

Solution: Move to external `.s` files

2. "no member named 'main'" in UEFI boot

Cause: UEFI expects `pub fn main() uefi.Status`

Solution: Don't use `pub export fn efi_main()` - use `main()`

3. Page Faults in UEFI Code

Cause: Wrong function signature or calling convention

Solution: Use wrapper functions, check return types are error unions

4. ISR/IDT Linker Errors

Cause: Function name mismatches between Zig and assembly

Solution: Ensure extern declarations match `.globl` names exactly

5. Build Fails with "undefined symbol"

Cause: Assembly file not linked in build.zig

Solution: Add `kernel.addAssemblyFile(b.path("src/filename.s"))`

File Structure

```
src/
├── boot.zig              # UEFI bootloader
├── kernel.zig            # Kernel entry & init
├── linker.ld             # Kernel linker script
├── *.s             # Assembly files (port_io, isr_stubs, context_switch)
├── idt.zig              # Interrupt descriptor table
├── apic.zig              # Advanced PIC
├── timer.zig             # PIT, TSC timing
├── memory.zig            # Memory management
├── threading.zig         # Task switching
├── pci.zig               # PCI bus enumeration
├── ahci.zig              # SATA disk driver
├── nvme.zig              # NVMe disk driver
├── nic.zig               # E1000 network card
├── arp.zig, dhcp.zig, dns.zig, http.zig, icmp.zig, ip.zig, tcp.zig, udp.zig,
├── ps2.zig              # Keyboard/mouse
├── graphics.zig         # Framebuffer
├── hda.zig              # Audio
├── xhci.zig             # USB
├── disk.zig             # Disk abstraction
├── fat32.zig            # FAT32 filesystem
└── silosec.zig          # Security layer
```

Development Workflow

1. Make changes to Zig or assembly files
2. Run `zig build` from project root
3. Copy bootloader and kernel to test directory
4. Run QEMU
5. Check serial output for boot messages
6. Debug with QEMU `-d int,cpu_reset` for crashes

Current Status

- ✅ Bootloader working, loads GOP and filesystem protocols
- ✅ All inline assembly externalized
- ✅ Build system functional
- Next: Load kernel ELF, parse headers, setup memory map
- Then: Exit boot services, jump to kernel
- Future: Complete kernel initialization stages

Key Lessons Learned

1. Zig 0.15.1 self-hosted backend has inline asm bugs - use external assembly
2. UEFI protocols use wrapper functions that take Protocol TYPE, not `&Protocol.guid`

3. Error handling in UEFI is via error unions - use `catch`
4. Assembly files must match extern declarations exactly
5. ISR names must match between `isr_stubs.s` and Zig extern declarations
6. Build configuration matters: `.use_llvm = false` is required

Now reading all KB files to write comprehensive documentation:I'll now write comprehensive documentation for all modules. Given the size, I'll create a structured document covering all files:

```
# Silo OS - Complete Module Documentation

## Table of Contents

1. [Boot System](#boot-system)

2. [CPU & Interrupts](#cpu--interrupts)

3. [Memory Management](#memory-management)

4. [Hardware Discovery](#hardware-discovery)

5. [Storage Subsystem](#storage-subsystem)

6. [Network Stack](#network-stack)

7. [Input/Output](#inputoutput)

8. [Graphics](#graphics)

9. [Audio](#audio)

10. [Threading](#threading)

11. [Security](#security)

12. [Utilities](#utilities)

---

## Boot System

### boot.zig
```

~~~
**Purpose**: UEFI bootloader executable that loads the kernel into memory

**Location**: Compiled to `EFI/BOOT/BOOTX64.efi`

**Entry Point**:

```zig

pub fn main() uefi.Status
~~~

Responsibilities:

1. Initialize UEFI protocols (GraphicsOutput, SimpleFileSystem)
2. Display boot status messages via console output
3. Locate kernel file at `/EFI/SILO/kernel.elf`
4. Parse ELF64 headers and program headers
5. Allocate memory pages for each loadable segment
6. Copy kernel segments to physical addresses specified in ELF
7. Obtain final memory map from UEFI
8. Exit UEFI boot services
9. Jump to kernel entry point

Key Structures:

- `MemoryMap`: UEFI memory descriptor array passed to kernel
- `Framebuffer`: Graphics framebuffer info (base address, width, height, pitch)
- `Elf64_Ehdr`: ELF file header (magic, entry point, program header offset)
- `Elf64_Phdr`: Program header (segment type, physical address, file size, memory size)

Protocol Usage:

```
// Graphics Output Protocol

const gop = bs.locateProtocol(std.os.uefi.protocol.GraphicsOutput, null);

// File System Protocol

const file_protocol = bs.locateProtocol(std.os.uefi.protocol.SimpleFileSys
```

Memory Allocation:

```
// Allocate pages for kernel segment
const pages = bs.allocatePages(
.{ .address = segment_address },
.loader_data,
num_pages
);
```

Current Status: Protocols initialized, file system accessible, kernel loading incomplete

---

CPU & Interrupts

idt.zig

Purpose: Interrupt Descriptor Table setup and interrupt dispatch

Constants:

- 256 IDT entries (0-31: exceptions, 32-47: hardware IRQs, 48-255: reserved)

Data Structures:

```
const IDTEntry = extern struct {
offset_low: u16,     // Handler address bits 0-15
selector: u16,       // Code segment (0x08)
ist: u8,             // Interrupt Stack Table index
type_attr: u8,       // Type and attributes
offset_mid: u16,     // Handler address bits 16-31
offset_high: u32,    // Handler address bits 32-63
reserved: u32,
};
const IDTPointer = extern struct {
```

```
limit: u16,         // Size - 1
base: u64,          // IDT base address
};
```

Functions:

`pub fn init() void`

- Sets up all 256 IDT entries
- Links to external ISR stub functions from `isr_stubs.s`
- Loads IDT using `lidt` instruction
- Enables interrupts with `sti`

`export fn interruptHandler(frame: *InterruptFrame) callconv(.c) void`

- Called by all ISR stubs
- Dispatches to exception handlers (0-31) or IRQ handlers (32-47)
- Sends EOI (End of Interrupt) to APIC

Exception Handlers:

- Exception 0: Divide by zero
- Exception 6: Invalid opcode
- Exception 13: General protection fault
- Exception 14: Page fault (reads CR2 for faulting address)

IRQ Handlers:

- IRQ 0 (vector 32): Timer tick → `threading.tick()`
- IRQ 1 (vector 33): Keyboard → `ps2.handleKeyboardInterrupt()`
- IRQ 11 (vector 43): Network → `net.processNetworkIRQ()`
- IRQ 12 (vector 44): Mouse → `ps2.handleMouseInterrupt()`

External Dependencies:

- `isr_stubs.s`: All 48 ISR entry points
- `apic.zig`: `sendEOI()` for interrupt acknowledgment

isr_stubs.s

Purpose: Assembly language interrupt service routine stubs

Implementation:

Each ISR:

1. Pushes error code (or dummy 0 if CPU doesn't provide one)
2. Pushes interrupt number
3. Jumps to common handler `isrCommon`

Common Handler (`isrCommon`):

1. Saves all general-purpose registers (rax, rbx, rcx, rdx, rsi, rdi, rbp, r8-r15)
2. Calls `interruptHandler(frame)` in C calling convention
3. Restores all registers
4. Pops error code and interrupt number
5. Returns with `iretq` (interrupt return)

ISR List:

- Exceptions: `isrDivideByZero`, `isrDebug`, `isrNMI`, `isrBreakpoint`, `isrOverflow`, `isrBoundRange`, `isrInvalidOpcode`, `isrDeviceNotAvailable`, `isrDoubleFault`, `isrInvalidTSS`, `isrSegmentNotPresent`, `isrStackFault`, `isrGeneralProtection`, `isrPageFault`, `isrFPUException`, `isrAlignmentCheck`, `isrMachineCheck`, `isrSIMDException`
- IRQs: `isrTimer`, `isrKeyboard`, `isrCascade`, `isrCOM2`, `isrCOM1`, `isrLPT2`, `isrFloppy`, `isrLPT1`, `isrRTC`, `isrIRQ9`, `isrIRQ10`, `isrIRQ11`, `isrMouse`, `isrFPU`, `isrPrimaryATA`, `isrSecondaryATA`

apic.zig

Purpose: Advanced Programmable Interrupt Controller management

Global Variables:

```
var lapic_base: u64          // Local APIC memory-mapped address

var ioapic_base: u64         // I/O APIC memory-mapped address
```

Functions:

```
pub fn init() void
```

- Gets Local APIC address from ACPI
- Enables Local APIC in spurious interrupt vector register
- Sets up timer

```
pub fn initIOAPIC(base: u32) void
```

- Initializes I/O APIC for routing hardware IRQs
- Maps I/O APIC registers

```
pub fn routeIRQToCore(irq: u8, vector: u8, core: u8) void
```

- Routes specific IRQ to interrupt vector on target CPU core
- Used to direct all IRQs to core 0

```
pub fn sendEOI() void
```

- Writes to EOI register to signal interrupt completion
- Must be called at end of every interrupt handler

```
pub fn disablePIC() void
```

- Masks all interrupts on legacy 8259 PIC
- Required when using APIC

```
pub fn getAPICID() u8
```

- Returns current CPU's Local APIC ID
- Used for per-core data structures

Register Access:

All APIC registers accessed via memory-mapped I/O at `lapic_base`

port_io.s

Purpose: x86 port I/O instructions (workaround for Zig inline asm bugs)

Functions:

```
.globl outb, inb, outw, inw, outl, inl
```

outb: Write byte to I/O port

- Parameters: `rdi` = port (u16), `rsi` = value (u8)

```
outb:

mov %sil, %al

mov %di, %dx

outb %al, %dx

ret
```

inb: Read byte from I/O port

- Parameters: `rdi` = port (u16)
- Returns: `al` = value (u8)

```
inb:

mov %di, %dx

inb %dx, %al

ret
```

outl: Write dword to I/O port (32-bit)

inl: Read dword from I/O port (32-bit)

Used By: timer.zig, ps2.zig, pci.zig, ahci.zig, nvme.zig

Memory Management

memory.zig

Purpose: Physical memory allocation, virtual memory paging, heap management

Physical Memory Allocator:

```
pub const PhysicalAllocator = struct {

bitmap: []u8,           // 1 bit per 4KB frame

frame_count: usize,     // Total frames

next_free: usize,       // Hint for allocation
pub fn init(
    descriptors: []MemoryDescriptor,
    allocator: std.mem.Allocator
) !PhysicalAllocator
pub fn allocFrame() ?usize           // Returns frame number
pub fn freeFrame(frame: usize) void
pub fn getPhysicalAddress(frame: usize) u64  // frame * 4096
};
```

Virtual Memory (4-Level Paging):

```
pub const VirtualMemory = struct {
pml4: *PageTable,               // Page Map Level 4
phys_alloc: *PhysicalAllocator,
pub fn init(phys_alloc: *PhysicalAllocator) !VirtualMemory
pub fn identityMap(vm: *VirtualMemory, start: u64, size: u64) !void
pub fn mapPage(vm: *VirtualMemory, virt: u64, phys: u64, writable: bool, k
pub fn unmapPage(vm: *VirtualMemory, virt: u64) void
pub fn activate(vm: *VirtualMemory) void  // Load CR3
};
```

Page Table Entry:

```
const PageEntry = packed struct {
present: u1,        // Page present in memory
writable: u1,       // Write permission
user: u1,           // User mode access
writethrough: u1,

cache_disable: u1,

accessed: u1,

dirty: u1,

page_size: u1,

global: u1,

available: u3,

address: u40,       // Physical frame address >> 12

reserved: u11,

nx: u1,             // No execute

};
```

Page Table Structure:

Each table has 512 entries (512 * 8 = 4096 bytes)

- PML4 → PDPT → PD → PT → Physical Frame
- Each entry covers different address ranges:
  - PML4: 512 GB
  - PDPT: 1 GB
  - PD: 2 MB
  - PT: 4 KB

Heap Allocator:

```
pub const HeapAllocator = struct {

vm: *VirtualMemory,

heap_start: usize,      // Virtual address (0x100000000 = 4GB)

heap_end: usize,

heap_pos: usize,
pub fn init(vm: *VirtualMemory, start: usize, size: usize) HeapAllocator

pub fn alloc(heap: *HeapAllocator, size: usize, alignment: usize) ?[]u8

pub fn free(heap: *HeapAllocator, ptr: []u8) void

};
```

Zig Allocator Interface:

```
pub const ZigAllocator = struct {

heap: *HeapAllocator,

pub fn init(heap: *HeapAllocator) std.mem.Allocator

};
```

Memory Layout:

- 0x0 - 0x100000: Reserved (BIOS, kernel low memory)
- 0x100000 - 0xFFFFFFFF: Identity mapped (first 4GB)
- 0x100000000+: Kernel heap (16MB at 4GB mark)
- Framebuffer: Identity mapped if above 4GB

Functions Used by Kernel Init:

```
pub fn allocateStack(vm: *VirtualMemory, size: usize) !usize
```

- Allocates virtual memory for thread stack
- Returns top of stack address

Hardware Discovery

acpi.zig

Purpose: Parse ACPI tables to discover hardware configuration

Key Structures:

```
const RSDP = extern struct {

signature: [8]u8,       // "RSD PTR "
checksum: u8,
oem_id: [6]u8,
revision: u8,
rsdt_address: u32,
// ACPI 2.0+
length: u32,
xsdt_address: u64,
extended_checksum: u8,
reserved: [3]u8,
};
const MADT = extern struct {  // Multiple APIC Description Table
header: SDTHeader,
local_apic_address: u32,
flags: u32,
// Followed by variable-length entries
};
const FADT = extern struct {  // Fixed ACPI Description Table
// Contains PM timer port
pm_tmr_blk: u32,
// ...
};
```

MADT Entry Types:

- Type 0: Local APIC (processor info)
- Type 1: I/O APIC (interrupt controller)

- Type 2: Interrupt Source Override (IRQ remapping)

Extracted Information:

```
pub const ACPIInfo = struct {
cpu_count: u8,              // Number of processors
local_apic_address: u32,    // Local APIC base
ioapic_address: u32,        // I/O APIC base
ioapic_count: u8,
pm_timer_port: u32,         // Power management timer
};
```

Functions:

```
pub fn init() void
```

- Searches for RSDP in BIOS memory areas
- Parses RSDT/XSDT to find MADT and FADT
- Extracts CPU count, APIC addresses, PM timer port

```
pub fn getCPUCount() u8
pub fn getLocalAPICAddress() u32
pub fn getIOAPICAddress() u32
pub fn getPMTimerPort() u32
```

```
pub fn readPMTimer() u32
pub fn pmTimerToMicroseconds(ticks: u32) u64
```

- Reads ACPI power management timer (3.579545 MHz)
- Converts ticks to microseconds

```
pub fn shutdown() void
pub fn reboot() void
```

- System power management (requires DSDT parsing, currently unimplemented)

pci.zig

Purpose: Enumerate PCI devices on the system bus

PCI Configuration Space Access:

```
const CONFIG_ADDRESS = 0xCF8;  // Address port

const CONFIG_DATA = 0xCFC;     // Data port
```

Data Structure:

```
pub const PCIDevice = struct {

bus: u8,

device: u8,

function: u8,

vendor_id: u16,

device_id: u16,

class_code: u8,

subclass: u8,

prog_if: u8,

bar0: u32,      // Base Address Register 0

bar1: u32,

bar2: u32,

bar3: u32,

bar4: u32,

bar5: u32,

interrupt_line: u8,
};
```

Functions:

```
pub fn enumerateDevices(allocator: std.mem.Allocator) ![]PCIDevice
```

- Scans all 256 buses, 32 devices per bus, 8 functions per device
- Reads configuration space for each present device
- Returns array of found devices

```
pub fn readConfig(bus: u8, device: u8, function: u8, offset: u8) u32
```

```
pub fn writeConfig(bus: u8, device: u8, function: u8, offset: u8, value: u
```

- Low-level PCI configuration space access

Device Classes Used:

- Class 0x01: Mass Storage (AHCI: subclass 0x06, NVMe: subclass 0x08)
- Class 0x02: Network Controller (E1000: vendor 0x8086)
- Class 0x04: Multimedia (HDA: subclass 0x03)
- Class 0x0C: Serial Bus (USB XHCI: subclass 0x03, prog_if 0x30)

timer.zig

Purpose: High-resolution timing using TSC and ACPI PM timer

Timing Sources:

1. **TSC** (Time Stamp Counter): CPU cycle counter, very fast
2. **ACPI PM Timer**: 3.579545 MHz timer, always available

Calibration:

```
pub fn init() void
```

- Attempts TSC calibration using ACPI PM timer
- Falls back to PM timer if TSC unavailable
- Stores TSC frequency for microsecond conversion

Functions:

```
pub fn busyWait(microseconds: u32) void
```

- Busy-wait loop using TSC or PM timer
- Does NOT yield to scheduler

```
pub fn sleep(ms: u32) void
```

- Calls `busyWait(ms * 1000)`

```
pub fn getTimeMicroseconds() u64

pub fn getTimeMilliseconds() u64

pub fn getTimeSeconds() f64
```

- Returns elapsed time since init

```
pub const ProfileTimer = struct {

start: u64,

pub fn begin() ProfileTimer

pub fn end(self: ProfileTimer) u64          // Returns cycles

pub fn endMicroseconds(self: ProfileTimer) u64

pub fn endMilliseconds(self: ProfileTimer) u64

};
```

Usage Example:

```
const prof = timer.ProfileTimer.begin();

// ... code to measure ...

const us = prof.endMicroseconds();
```

Storage Subsystem

disk.zig

Purpose: Abstract disk interface for filesystem layer

Structure:

```
pub const DiskInterface = struct {

read_fn: *const fn (*anyopaque, u64, u32, []u8) anyerror!void,

write_fn: *const fn (*anyopaque, u64, u32, []const u8) anyerror!void,

context: *anyopaque,

};
```

Usage: Wraps either AHCI or NVMe controller for FAT32 filesystem

ahci.zig

Purpose: SATA disk controller driver (Advanced Host Controller Interface)

HBA Registers (memory-mapped):

```
const HBA_CAP = 0x00;        // Capabilities

const HBA_GHC = 0x04;        // Global Host Control

const HBA_IS = 0x08;         // Interrupt Status

const HBA_PI = 0x0C;         // Ports Implemented
```

Port Registers (offset 0x100 + port * 0x80):

```
const PORT_CMD = 0x18;       // Command and Status

const PORT_IS = 0x10;        // Interrupt Status

const PORT_CI = 0x38;        // Command Issue

const PORT_SSTS = 0x28;      // SATA Status
```

FIS Types (Frame Information Structure):

```
const FIS_TYPE_REG_H2D = 0x27;       // Register - Host to Device

const FIS_TYPE_DMA_SETUP = 0x41;     // DMA Setup
```

Structures:

```
pub const AHCIController = struct {
```

```
hba_base: [*]volatile u32,      // Memory-mapped registers

ports: [32]?AHCIPort,           // Up to 32 SATA ports

pub fn init(devices: []PCIDevice, allocator: std.mem.Allocator) !AHCIContr

pub fn detectPort(ctrl: *AHCIController, port_num: u8) bool

};

pub const AHCIPort = struct {

port_base: [*]volatile u32,

clb: [*]CommandHeader,          // Command List Base

cmd_tables: [32]*CommandTable,

pub fn read(port: *AHCIPort, lba: u64, count: u16, buffer: []u8) !void

pub fn write(port: *AHCIPort, lba: u64, count: u16, buffer: []const u8) !v

};
```

Command Execution:

1. Find free command slot (0-31)
2. Setup command header (FIS length, PRDT count)
3. Setup PRDT (Physical Region Descriptor Table) with buffer address
4. Build command FIS (READ_DMA_EX or WRITE_DMA_EX, LBA, sector count)
5. Issue command by setting bit in PORT_CI
6. Poll until completion (PORT_CI bit clears)
7. Clear interrupt status

Wrapper Functions:

```
pub fn readWrapper(context: *anyopaque, lba: u64, count: u32, buffer: []u8

pub fn writeWrapper(context: *anyopaque, lba: u64, count: u32, buffer: []c
```

nvme.zig

Purpose: NVMe (Non-Volatile Memory Express) SSD driver

Controller Registers (memory-mapped BAR0):

```
const CAP = 0x00;       // Controller Capabilities (64-bit)

const VS = 0x08;        // Version

const CC = 0x14;        // Controller Configuration

const CSTS = 0x1C;      // Controller Status

const AQA = 0x24;       // Admin Queue Attributes

const ASQ = 0x28;       // Admin Submission Queue (64-bit)

const ACQ = 0x30;       // Admin Completion Queue (64-bit)
```

Queue Structure:

- Admin Queue: Commands to controller (identify, create queues)
- I/O Submission Queue: Read/write commands
- I/O Completion Queue: Command results

Structures:

```
pub const NVMeController = struct {

bar0: [*]volatile u32,          // Register base

admin_sq: []NVMeSubmissionQueueEntry,

admin_cq: []NVMeCompletionQueueEntry,

io_sq: []NVMeSubmissionQueueEntry,

io_cq: []NVMeCompletionQueueEntry,

pub fn init(devices: []PCIDevice, allocator: std.mem.Allocator) !NVMeContr

pub fn read(ctrl: *NVMeController, lba: u64, count: u16, buffer: []u8) !vo

pub fn write(ctrl: *NVMeController, lba: u64, count: u16, buffer: []const
pub fn flush(ctrl: *NVMeController) !void
```

```
};
```

Initialization Steps:

1. Find NVMe device in PCI devices (class 0x01, subclass 0x08)
2. Enable bus mastering and memory space in PCI command register
3. Map BAR0 to access controller registers
4. Reset controller (CC.EN = 0)
5. Allocate and setup Admin queues
6. Enable controller (CC.EN = 1)
7. Identify controller and namespace
8. Create I/O submission/completion queues
9. Ready for I/O commands

Command Execution:

1. Build submission queue entry (opcode, namespace, LBA, data pointer)
2. Write to submission queue
3. Ring doorbell (write queue tail to register)
4. Poll completion queue
5. Check status in completion entry
6. Update completion queue head

Wrapper Functions:

```
pub fn readWrapper(context: *anyopaque, lba: u64, count: u32, buffer: []u8

pub fn writeWrapper(context: *anyopaque, lba: u64, count: u32, buffer: []c
```

fat32.zig

Purpose: FAT32 filesystem implementation

Structures:

```
const BPB = extern struct {  // BIOS Parameter Block

jump_boot: [3]u8,
oem_name: [8]u8,

bytes_per_sector: u16,

sectors_per_cluster: u8,

reserved_sectors: u16,

fat_count: u8,

root_entry_count: u16,

total_sectors_16: u16,

media_type: u8,

fat_size_16: u16,

sectors_per_track: u16,

head_count: u16,

hidden_sectors: u32,

total_sectors_32: u32,

// FAT32 specific

fat_size_32: u32,

ext_flags: u16,

fs_version: u16,

root_cluster: u32,

// ...

};

const DirectoryEntry = extern struct {

name: [11]u8,               // 8.3 format

attributes: u8,
reserved: u8,
creation_time_tenths: u8,
creation_time: u16,
creation_date: u16,
access_date: u16,
first_cluster_high: u16,
write_time: u16,
write_date: u16,
first_cluster_low: u16,
file_size: u32,
};
```

FAT32 Object:

```
pub const FAT32 = struct {
disk: DiskInterface,
bytes_per_sector: u32,
sectors_per_cluster: u32,
reserved_sectors: u32,
fat_count: u32,
fat_size: u32,
root_cluster: u32,
data_start_sector: u32,
pub fn init(disk: DiskInterface, allocator: std.mem.Allocator) !FAT32
pub fn readFile(fs: *FAT32, path: []const u8, buffer: []u8) !usize
pub fn writeFile(fs: *FAT32, path: []const u8, data: []const u8) !void

pub fn listDirectory(fs: *FAT32, path: []const u8, allocator: std.mem.Allo

};
```

Functions:

```
fn readSector(fs: *FAT32, lba: u64, buffer: []u8) !void

fn readCluster(fs: *FAT32, cluster: u32, buffer: []u8) !void

fn getNextCluster(fs: *FAT32, cluster: u32) !u32

fn findFile(fs: *FAT32, path: []const u8) !DirectoryEntry
```

Cluster Chain Walking:

1. Start at file's first cluster
2. Read data from cluster
3. Look up next cluster in FAT
4. Continue until end marker (0x0FFFFFFF)

Network Stack

nic.zig

Purpose: Intel E1000 network card driver

Registers (memory-mapped):

```
const CTRL = 0x0000;        // Device Control

const STATUS = 0x0008;      // Device Status

const EECD = 0x0010;        // EEPROM Control

const CTRL_EXT = 0x0018;    // Extended Control

const RDBAL = 0x2800;       // RX Descriptor Base Low

const RDBAH = 0x2804;       // RX Descriptor Base High

const RDLEN = 0x2808;       // RX Descriptor Length
const RDH = 0x2810;          // RX Descriptor Head

const RDT = 0x2818;          // RX Descriptor Tail

const TDBAL = 0x3800;        // TX Descriptor Base Low

// ...
```

Structures:

```
const RxDescriptor = extern struct {

buffer: u64,        // Physical address of receive buffer

length: u16,

checksum: u16,

status: u8,

errors: u8,

special: u16,

};

const TxDescriptor = extern struct {

buffer: u64,        // Physical address of transmit buffer

length: u16,

cso: u8,

cmd: u8,

status: u8,

css: u8,

special: u16,

};

pub const EthernetFrame = struct {
```

```
dst_mac: [6]u8,

src_mac: [6]u8,

ethertype: u16,     // 0x0800 = IPv4, 0x0806 = ARP

payload: []const u8,

payload_len: usize,

};
```

NIC Object:

```
pub const NIC = struct {

mmio_base: [*]volatile u32,

mac: [6]u8,

rx_descriptors: [32]RxDescriptor,

tx_descriptors: [32]TxDescriptor,

rx_buffers: [32][2048]u8,

tx_buffers: [32][2048]u8,

rx_tail: u32,

tx_tail: u32,

pub fn init(devices: []PCIDevice, allocator: std.mem.Allocator) !NIC

pub fn transmit(nic: *NIC, frame: *const EthernetFrame) !void

pub fn receive(nic: *NIC) ?EthernetFrame

};
```

Initialization:

1. Find E1000 in PCI devices (vendor 0x8086, device 0x100E or similar)
2. Enable bus mastering in PCI command register
3. Map BAR0 for register access

4. Read MAC address from EEPROM
5. Allocate RX/TX descriptor rings
6. Setup descriptor base addresses in registers
7. Enable receiver and transmitter
8. Configure interrupt throttling

Transmit Flow:

1. Get next TX descriptor
2. Copy frame data to TX buffer
3. Setup descriptor (buffer address, length, EOP flag)
4. Advance TX tail register
5. Hardware sends packet

Receive Flow:

1. Hardware writes to RX descriptor ring
2. Poll RX descriptors for DD (Descriptor Done) bit
3. Copy packet from RX buffer
4. Advance RX tail to reuse descriptor

net.zig

Purpose: Network stack coordinator and packet dispatcher

Global State:

```
var network_config: NetConfig = undefined;

var network_interface: ?*nic.NIC = null;

var rx_queue: RxQueue = undefined;

pub const NetConfig = struct {

ip_address: u32,         // Network byte order

subnet_mask: u32,

gateway: u32,

dns_server: u32,
mac_address: [6]u8,

};
```

Packet Queue:

```
pub const RxQueue = struct {

packets: [64]PacketBuffer,

head: u32,

tail: u32,

lock: threading.SpinLock,

pub fn enqueue(queue: *RxQueue, packet: *const PacketBuffer) bool

pub fn dequeue(queue: *RxQueue) ?PacketBuffer

};
```

Functions:

```
pub fn init(config: NetConfig, pci_devices: []PCIDevice, allocator: std.me
```

- Initializes NIC
- Initializes all protocol layers (ARP, IP, UDP, TCP, ICMP, DHCP, DNS)

```
pub fn processNetworkIRQ() void
```

- Called from IRQ handler on core 0
- Receives packets from NIC
- Dispatches to protocol layers based on EtherType

```
pub fn sendEthernet(dst_mac: [6]u8, ethertype: u16, payload: []const u8) !

pub fn sendIP(dst_ip: u32, protocol: u8, payload: []const u8) !void
```

- Send packet functions
- `sendIP` performs ARP resolution before sending

```
pub fn setConfig(new_config: NetConfig) void
```

- Thread-safe configuration update (used by DHCP)

arp.zig

Purpose: Address Resolution Protocol - maps IP to MAC addresses

Cache:

```
const ARPEntry = struct {

ip: u32,

mac: [6]u8,

timestamp: u64,

valid: bool,

};

var arp_cache: [256]ARPEntry = undefined;
```

Packet Structure:

```
const ARPPacket = extern struct {

htype: u16,         // Hardware type (1 = Ethernet)

ptype: u16,         // Protocol type (0x0800 = IPv4)

hlen: u8,           // Hardware address length (6)

plen: u8,           // Protocol address length (4)

operation: u16,     // 1 = request, 2 = reply

sender_mac: [6]u8,

sender_ip: u32,

target_mac: [6]u8,

target_ip: u32,

};
```

Functions:

```
pub fn init(our_mac: [6]u8, our_ip: u32) void

pub fn resolve(ip: u32) ?[6]u8         // Returns cached MAC or null

pub fn sendRequest(target_ip: u32) !void

pub fn processPacket(data: []const u8) void

pub fn tick() void                     // Age out old entries
```

Operation:

1. Request: Broadcast "Who has IP X.X.X.X?"
2. Reply: Unicast "I have IP X.X.X.X, my MAC is XX:XX:XX:XX:XX:XX"
3. Cache entries expire after timeout
4. Gratuitous ARP on IP address change

ip.zig

Purpose: Internet Protocol (IPv4) packet handling

Header Structure:

```
pub const IPv4Header = extern struct {

version_ihl: u8,        // Version (4) | Header Length

tos: u8,                // Type of Service

total_length: u16,      // Total packet length

identification: u16,    // Fragment ID

flags_offset: u16,      // Flags | Fragment Offset

ttl: u8,                // Time to Live (64)

protocol: u8,           // 1=ICMP, 6=TCP, 17=UDP

checksum: u16,

src_ip: u32,
dst_ip: u32,
```

```
};
```

Functions:

```
pub fn init(ip: u32) void

pub fn processPacket(data: []const u8) void

pub fn buildPacket(dst_ip: u32, protocol: u8, payload: []const u8, buffer:
```

Packet Processing:

1. Verify version = 4
2. Verify checksum
3. Check destination IP (our IP or broadcast)
4. Dispatch to protocol handler based on `protocol` field:
    - 1 → `icmp.processPacket()`
    - 6 → `tcp.processPacket()`
    - 17 → `udp.processPacket()`

Checksum Calculation:

```
fn calculateChecksum(data: []const u8) u16

fn verifyChecksum(data: []const u8) bool
```

- One's complement sum of 16-bit words

Utility:

```
pub fn ipToString(ip: u32, buffer: []u8) ![]const u8

pub fn stringToIP(str: []const u8) !u32
```

icmp.zig

Purpose: Internet Control Message Protocol (ping, error messages)

Message Types:

```
const ECHO_REPLY = 0;
const ECHO_REQUEST = 8;

const DEST_UNREACHABLE = 3;

const TIME_EXCEEDED = 11;
```

Packet Structure:

```
const ICMPHeader = extern struct {

msg_type: u8,

code: u8,

checksum: u16,

identifier: u16,

sequence: u16,

};
```

Functions:

```
pub fn init() void

pub fn sendEchoRequest(dst_ip: u32, identifier: u16, sequence: u16) !void

pub fn processPacket(src_ip: u32, data: []const u8) void
```

Echo Request (Ping):

1. Build ICMP header with type=8 (Echo Request)
2. Calculate checksum
3. Send via IP layer
4. Receive Echo Reply (type=0)

udp.zig

Purpose: User Datagram Protocol - connectionless transport

Header Structure:

```
const UDPHeader = extern struct {
```

```
src_port: u16,

dst_port: u16,

length: u16,        // Header + data

checksum: u16,      // Optional for IPv4

};
```

Socket Management:

```
const MAX_SOCKETS = 256;

const UDPSocket = struct {

local_port: u16,

remote_ip: u32,

remote_port: u16,

bound: bool,

rx_queue: [16]UDPPacket,

rx_head: u8,

rx_tail: u8,

rx_lock: threading.SpinLock,

};

var socket_registry: [MAX_SOCKETS]?UDPSocket = undefined;
```

Functions:

```
pub fn init() void

pub fn createSocket() !u16              // Returns socket ID

pub fn bind(socket_id: u16, port: u16) !void

pub fn connect(socket_id: u16, remote_ip: u32, remote_port: u16) !void
pub fn send(socket_id: u16, dst_ip: u32, dst_port: u16, data: []const u8)

pub fn recv(socket_id: u16, buffer: []u8) !?struct { src_ip: u32, src_port

pub fn close(socket_id: u16) void

pub fn processPacket(src_ip: u32, data: []const u8) void
```

Packet Routing:

- Port 53: DNS responses → `dns.processResponse()`
- Port 68: DHCP responses → `dhcp.processPacket()`
- Other: Check socket registry, queue packet

tcp.zig

Purpose: Transmission Control Protocol - reliable stream transport

Header Structure:

```
const TCPHeader = extern struct {

src_port: u16,

dst_port: u16,

seq_num: u32,

ack_num: u32,

data_offset_flags: u16,     // Offset (4 bits) | Flags (12 bits)

window_size: u16,

checksum: u16,

urgent_pointer: u16,

};
```

TCP Flags:

```
const FLAG_FIN = 0x01;

const FLAG_SYN = 0x02;
const FLAG_RST = 0x04;

const FLAG_PSH = 0x08;

const FLAG_ACK = 0x10;

const FLAG_URG = 0x20;
```

Connection States:

```
pub const TCPState = enum {

Closed,

Listen,

SynSent,

SynReceived,

Established,

FinWait1,

FinWait2,

CloseWait,

Closing,

LastAck,

TimeWait,

};
```

Socket Structure:

```
const TCPSocket = struct {

local_port: u16,

remote_ip: u32,

remote_port: u16,
state: TCPState,

seq_num: u32,           // Our sequence number

ack_num: u32,           // Expected next byte from peer

window_size: u16,

rx_buffer: [8192]u8,

rx_len: usize,

tx_buffer: [8192]u8,

tx_len: usize,

lock: threading.SpinLock,

};
```

Functions:

```
pub fn init() void

pub fn createSocket() !u16

pub fn bind(socket_id: u16, port: u16) !void

pub fn listen(socket_id: u16, port: u16) !void

pub fn connect(socket_id: u16, remote_ip: u32, remote_port: u16) !void

pub fn send(socket_id: u16, data: []const u8) !void

pub fn recv(socket_id: u16, buffer: []u8) !usize

pub fn close(socket_id: u16) !void

pub fn getState(socket_id: u16) ?TCPState

pub fn processPacket(src_ip: u32, data: []const u8) void

pub fn tick() void                     // Retransmission timer
```

Three-Way Handshake (Client):

1. Send SYN

2. Receive SYN-ACK
3. Send ACK
4. State = Established

Connection Termination:

1. Send FIN
2. Receive FIN-ACK
3. Send ACK
4. State = Closed

dhcp.zig

Purpose: Dynamic Host Configuration Protocol - automatic IP configuration

Message Types:

```
const DHCP_DISCOVER = 1;     // Client broadcasts "I need an IP"
const DHCP_OFFER = 2;        // Server offers IP address
const DHCP_REQUEST = 3;      // Client requests offered IP
const DHCP_ACK = 5;          // Server confirms lease
const DHCP_NAK = 6;          // Server denies request
const DHCP_RELEASE = 7;      // Client releases IP
```

Packet Structure:

```
pub const DHCPPacket = extern struct {
op: u8,                 // 1 = request, 2 = reply
htype: u8,              // 1 = Ethernet
hlen: u8,               // 6 bytes
hops: u8,
xid: u32,               // Transaction ID
secs: u16,
flags: u16,             // 0x8000 = broadcast

ciaddr: u32,            // Client IP

yiaddr: u32,            // Your (offered) IP

siaddr: u32,            // Server IP

giaddr: u32,            // Gateway IP

chaddr: [16]u8,         // Client MAC

sname: [64]u8,

file: [128]u8,

magic: u32,             // 0x63825363

options: [312]u8,

};
```

DHCP State Machine:

```
pub const DHCPState = enum {

Disabled,

Init,

Selecting,      // Sent DISCOVER, waiting for OFFER

Requesting,     // Sent REQUEST, waiting for ACK

Bound,          // Lease acquired

Renewing,

Rebinding,

};
```

Lease Information:

```
pub const DHCPLease = struct {
```

```
ip_address: u32,
subnet_mask: u32,
gateway: u32,
dns_server: u32,
server_id: u32,
lease_time: u32,        // Seconds
renewal_time: u32,
rebind_time: u32,
lease_start: u64,       // Timestamp
};
```

Functions:

```
pub fn init() void
pub fn enable(mac: [6]u8) !void
pub fn disable() !void
pub fn isEnabled() bool
pub fn getState() DHCPState
pub fn getLease() ?DHCPLease
pub fn renew() !void
pub fn processPacket(data: []const u8, src_ip: u32, src_port: u16) void
pub fn tick() void                  // Handle timeouts, renewals
```

DORA Process:

1. **Discover**: Broadcast to 255.255.255.255:67
2. **Offer**: Server responds with available IP
3. **Request**: Client requests offered IP

4. **Ack**: Server confirms, provides lease time

Lease Renewal:

- At 50% of lease time: Try to renew with same server
- At 87.5% of lease time: Rebind with broadcast
- At 100%: Release and start over

dns.zig

Purpose: Domain Name System - resolve domain names to IP addresses

Constants:

```
pub const GOOGLE_DNS_PRIMARY = 0x08080808;       // 8.8.8.8

pub const GOOGLE_DNS_SECONDARY = 0x08080404;     // 8.8.4.4
```

Packet Structure:

```
const DNSHeader = extern struct {

id: u16,

flags: u16,

qdcount: u16,       // Question count

ancount: u16,       // Answer count

nscount: u16,       // Authority count

arcount: u16,       // Additional count

};
```

Query Cache:

```
const DNSQuery = struct {

id: u16,

hostname: [256]u8,

hostname_len: usize,
result_ip: u32,
completed: bool,
timestamp: u64,
};
var query_cache: [16]DNSQuery = undefined;
```

Functions:

```
pub fn init(dns_server: u32) void
pub fn resolve(hostname: []const u8) !u16       // Returns query ID
pub fn resolveBlocking(hostname: []const u8, timeout_ms: u32) !u32
pub fn getResult(query_id: u16) ?u32
pub fn processResponse(data: []const u8, src_ip: u32) void
pub fn tick() void                               // Expire old queries
```

Query Process:

1. Create DNS query packet with question section
2. Send to DNS server via UDP port 53
3. Receive response
4. Parse answer section for A record (IPv4 address)
5. Cache result

Name Encoding:

Domain names encoded as length-prefixed labels:

`www.example.com` → `\x03www\x07example\x03com\x00`

http.zig

Purpose: HTTP/1.1 client implementation

URL Parsing:

```
const URL = struct {
```

```
scheme: []const u8,      // "http"

host: []const u8,        // "example.com"

port: u16,               // 80

path: []const u8,        // "/index.html"

};
```

Response Structure:

```
pub const HTTPResponse = struct {

status_code: u16,

headers: std.StringHashMap([]const u8),

body: []u8,

allocator: std.mem.Allocator,

pub fn deinit(response: *HTTPResponse) void

};
```

Functions:

```
pub fn simpleGet(url_string: []const u8, allocator: std.mem.Allocator) !HT
```

Request Flow:

1. Parse URL
2. DNS resolve hostname to IP
3. Create TCP socket
4. Connect to IP:port
5. Wait for connection established
6. Send HTTP request:

   ```
   GET /path HTTP/1.1\r\n

   Host: hostname\r\n

   User-Agent: SiloOS/0.1\r\n
   ```

```
Accept: */*\r\n
Connection: close\r\n
\r\n
```

7. Receive response
8. Parse status line and headers
9. Extract body
10. Close connection

Input/Output

ps2.zig

Purpose: PS/2 keyboard and mouse controller driver

Ports:

```
const DATA_PORT = 0x60;
const STATUS_PORT = 0x64;
const COMMAND_PORT = 0x64;
```

Controller State:

```
pub const PS2Controller = struct {
keyboard_buffer: [16]u8,
keyboard_head: u8,
keyboard_tail: u8,
mouse_x: i32,
mouse_y: i32,
mouse_buttons: u8,
key_states: [256]bool,      // Scancode state
};

var controller: ?PS2Controller = null;
```

Scancode Mapping:

```
pub const SCANCODE_ESC = 0x01;

pub const SCANCODE_SPACE = 0x39;

pub const SCANCODE_ENTER = 0x1C;

pub const SCANCODE_TAB = 0x0F;

pub const SCANCODE_P = 0x19;

pub const SCANCODE_D = 0x20;

// ... etc
```

Functions:

```
pub fn init() !void

pub fn getController() ?*PS2Controller

pub fn handleKeyboardInterrupt() void

pub fn handleMouseInterrupt() void

pub fn isKeyPressed(scancode: u8) bool

pub fn getMousePosition() struct { x: i32, y: i32 }

pub fn getMouseButtons() u8
```

Keyboard Handling:

1. IRQ 1 triggers
2. Read scancode from DATA_PORT
3. Update key state (0x80 bit = release)
4. Buffer scancode

Mouse Handling:

1. IRQ 12 triggers

2. Read 3-byte packet:
    - Byte 0: Button flags and sign bits
    - Byte 1: X movement
    - Byte 2: Y movement
3. Update position with bounds checking

Initialization:

1. Disable both PS/2 ports
2. Flush output buffer
3. Set controller configuration
4. Enable ports
5. Enable interrupts

Graphics

graphics.zig

Purpose: 2D framebuffer rendering and 3D software rasterization

2D Framebuffer:

```
pub const Framebuffer = struct {

pixels: [*]u32,          // ARGB format

width: u32,

height: u32,

pitch: u32,              // Pixels per scanline

};
```

2D Drawing Functions:

```
pub fn clear(fb: *Framebuffer, color: u32) void

pub fn setPixel(fb: *Framebuffer, x: i32, y: i32, color: u32) void

pub fn drawLine(fb: *Framebuffer, x0: i32, y0: i32, x1: i32, y1: i32, colo
pub fn drawRect(fb: *Framebuffer, x: i32, y: i32, width: i32, height: i32,

pub fn drawCircle(fb: *Framebuffer, cx: i32, cy: i32, radius: i32, color:

pub fn drawString(fb: *Framebuffer, x: i32, y: i32, text: []const u8, colo
```

3D Structures:

```
pub const Vec3 = struct { x: f32, y: f32, z: f32 };

pub const Vec2 = struct { x: f32, y: f32 };

pub const Mat4 = struct {

m: [16]f32,              // Column-major order

pub fn identity() Mat4

pub fn multiply(a: Mat4, b: Mat4) Mat4

pub fn perspective(fov: f32, aspect: f32, near: f32, far: f32) Mat4

};

pub const Vertex = struct {

pos: Vec3,

uv: Vec2,

color: u32,

};

pub const Mesh = struct {

vertices: []const Vertex,

indices: []const u32,       // Triangle indices

};
```

3D Context:

```
pub const GraphicsContext3D = struct {
```

```
framebuffer: *Framebuffer,
zbuffer: []f32,             // Depth buffer
view_matrix: Mat4,
projection_matrix: Mat4,
pub fn init(fb: *Framebuffer, allocator: std.mem.Allocator) !GraphicsConte
pub fn clearZBuffer(ctx: *GraphicsContext3D) void
};
```

3D Rendering Functions:

```
pub fn drawMesh(ctx: *GraphicsContext3D, mesh: *const Mesh, model: Mat4) v
```

- Transforms vertices by model * view * projection matrices
- Performs perspective division (w component)
- Clips to viewport
- Rasterizes triangles with Z-buffering
- Interpolates vertex colors

Triangle Rasterization:

1. Sort vertices by Y coordinate
2. Compute edge equations
3. Scanline fill with barycentric interpolation
4. Depth test against Z-buffer
5. Write pixel if depth test passes

Audio

hda.zig

Purpose: Intel High Definition Audio controller driver

Registers (memory-mapped):

```
const GCAP = 0x00;          // Global Capabilities
const GCTL = 0x08;          // Global Control

const WAKEEN = 0x0C;        // Wake Enable

const STATESTS = 0x0E;      // State Change Status

const INTCTL = 0x20;        // Interrupt Control

const INTSTS = 0x24;        // Interrupt Status

const CORBLBASE = 0x40;     // CORB Lower Base Address

const RIRBLBASE = 0x50;     // RIRB Lower Base Address
```

Audio System:

```
pub const AudioSystem = struct {

mmio_base: [*]volatile u32,

output_stream: AudioStream,

buffer: []i16,              // PCM samples

buffer_size: usize,

sample_rate: u32,           // 48000 Hz

pub fn init(devices: []PCIDevice, allocator: std.mem.Allocator) !AudioSyst

pub fn playPCM(sys: *AudioSystem, samples: []const i16) !void

pub fn stop(sys: *AudioSystem) void

};
```

Stream Descriptor:

```
const StreamDescriptor = extern struct {

ctl_sts: u32,

lpib: u32,              // Link Position in Buffer

cbl: u32,               // Cyclic Buffer Length
lvi: u16,                // Last Valid Index

fifod: u16,

fmt: u16,                // Stream format

bdpl: u32,               // Buffer Descriptor Pointer Low

bdpu: u32,               // Buffer Descriptor Pointer Upper

};
```

Buffer Descriptor:

```
const BufferDescriptor = extern struct {

address: u64,            // Physical address of buffer

length: u32,             // Buffer length in bytes

ioc: u32,                // Interrupt on Completion

};
```

Initialization:

1. Find HDA device (class 0x04, subclass 0x03)
2. Reset controller
3. Allocate CORB/RIRB (Command/Response ring buffers)
4. Enumerate codecs
5. Configure output stream
6. Setup buffer descriptors
7. Set stream format (48kHz, 16-bit stereo)
8. Start stream

audio_player.zig

Purpose: Multi-channel PCM audio playback system

PCM Sound:

```
pub const PCMSound = struct {
```

```
samples: []const i16,       // Interleaved stereo L,R,L,R
sample_rate: u32,           // 48000
channels: u8,               // 1=mono, 2=stereo
loop: bool,
};
```

Audio Player:

```
pub const AudioPlayer = struct {
hda_device: hda.HDADevice,
channels: [16]SoundChannel,     // 16 simultaneous sounds
mix_buffer: []i16,
sample_rate: u32,
buffer_half_size: u32,
pub fn init(devices: []PCIDevice, allocator: std.mem.Allocator) !AudioPlay
pub fn play(player: *AudioPlayer, sound: *const PCMSound, channel: u8) !vo
pub fn stop(player: *AudioPlayer, channel: u8) void
pub fn setVolume(player: *AudioPlayer, channel: u8, volume: f32) void
pub fn update(player: *AudioPlayer) void
};
```

Mixing:

1. Clear mix buffer
2. For each active channel:
    - Read samples from position
    - Multiply by volume
    - Add to mix buffer (clamp to prevent overflow)
3. Send mixed buffer to HDA

4. Advance channel positions

Threading

threading.zig

Purpose: Cooperative multitasking scheduler

Thread Context:

```
pub const ThreadContext = struct {

rax: u64, rbx: u64, rcx: u64, rdx: u64,

rsi: u64, rdi: u64, rbp: u64, rsp: u64,

r8: u64, r9: u64, r10: u64, r11: u64,

r12: u64, r13: u64, r14: u64, r15: u64,

rip: u64,           // Instruction pointer

rflags: u64,        // CPU flags

};
```

Thread Control Block:

```
pub const Thread = struct {

id: u32,

state: ThreadState,

context: ThreadContext,

stack_base: usize,

stack_size: usize,

affinity: u8,           // CPU core affinity

priority: u8,

entry_point: *const fn (*anyopaque) void,
arg: *anyopaque,

pub fn init(...) Thread

};

pub const ThreadState = enum {

Ready,

Running,

Waiting,

Terminated,

};
```

Per-CPU Scheduler:

```
pub const CPUScheduler = struct {

cpu_id: u8,

current_thread: ?*Thread,

ready_queue: [256]?*Thread,     // Ring buffer

queue_head: u8,

queue_tail: u8,

idle_thread: Thread,

lock: SpinLock,

pub fn init(cpu_id: u8, vm: *VirtualMemory, allocator: std.mem.Allocator)

pub fn addThread(sched: *CPUScheduler, thread: *Thread) void

pub fn schedule(sched: *CPUScheduler) ?*Thread

pub fn yield(sched: *CPUScheduler) void

};
```

Global Functions:

```
pub fn initScheduler(cpu_id: u8, vm: *VirtualMemory, allocator: std.mem.Al

pub fn createThread(entry: fn, arg: *anyopaque, cpu_id: u8, ...) !*Thread

pub fn yield() void

pub fn sleep(ms: u32) void

pub fn tick() void              // Called by timer interrupt
```

Context Switch (in `context_switch.s`):

1. Save current thread context (all registers, RIP, RFLAGS)
2. Load new thread context
3. Jump to new RIP

Synchronization Primitives:

```
pub const SpinLock = struct {

locked: u32,

pub fn init() SpinLock

pub fn acquire(lock: *SpinLock) void

pub fn release(lock: *SpinLock) void

};

pub const Barrier = struct {

count: u32,

current: u32,

generation: u32,

pub fn init(count: u32) Barrier

pub fn wait(barrier: *Barrier) void

};
```

Work Distribution:

```
pub const WorkBatch = struct {

workload_type: WorkloadType,

input_data_ptr: ?[*]u8,

output_data_ptr: ?[*]u8,

item_count: u32,

item_stride: u32,

batch_id: u32,

};

pub fn distributeWork(dist: *const WorkDistribution, vm: *VirtualMemory, a
```

- Divides work into batches
- Creates worker threads on different cores
- Each thread processes subset of data

context_switch.s

Purpose: Assembly routine for thread context switching

Function Signature:

```
.globl contextSwitch

contextSwitch:      # void contextSwitch(ThreadContext *old, ThreadContext
```

Operation:

1. Save old context:
    - Save all general-purpose registers to `old` struct
    - Save return address as RIP
    - Save RFLAGS
2. Restore new context:
    - Load all registers from `new` struct
    - Load RFLAGS

    - Load stack pointer

3. Jump to new RIP (continues execution of new thread)

Register Layout:

```
old context pointer in %rdi

new context pointer in %rsi
```

Security

silosec.zig

Purpose: Authentication and encryption layer

Key Structures:

```
pub const AuthToken = struct {

user_id: u32,

permissions: u64,       // Bitfield

expiry: u64,            // Timestamp

signature: [32]u8,      // HMAC-SHA256

};

pub const EncryptionContext = struct {

key: [32]u8,            // AES-256 key

iv: [16]u8,             // Initialization vector

};
```

Functions:

```
pub fn generateAuthToken(user_id: u32, permissions: u64, duration: u64) !A

pub fn verifyAuthToken(token: *const AuthToken) bool

pub fn encryptData(ctx: *EncryptionContext, plaintext: []const u8, ciphert
pub fn decryptData(ctx: *EncryptionContext, ciphertext: []const u8, plaint
```

```
pub fn hashPassword(password: []const u8, salt: []const u8, hash: []u8) !v
```

Hashing:

- Uses SHA-256 for password hashing
- PBKDF2 key derivation

Encryption:

- AES-256 in CBC mode
- HMAC-SHA256 for message authentication

Utilities

demo.zig

Purpose: Demo functions for testing different subsystems

Functions:

```
pub fn demo3D(rotation: *f32) void
```

- Renders rotating cube using 3D graphics context
- Updates rotation angle each frame

```
pub fn demoNetwork(udp_sock: *?u16, tcp_sock: *?u16) void
```

- Displays network status (IP, MAC, gateway)
- Creates UDP and TCP test sockets
- Shows packet statistics

```
pub fn demoDHCP() void
```

- Shows DHCP client state
- Displays lease information (IP, subnet, gateway, DNS)
- Shows lease timing

disk_sqlite.zig

Purpose: SQLite-style database on disk (planned feature)

Status: Stub implementation, not currently functional

Intended Functionality:

- B-tree index structures
- Page-based storage
- Transaction log
- SQL query parsing

Kernel Entry Point

kernel.zig

Purpose: Main kernel entry and initialization coordinator

Global State:

```
pub var phys_allocator: memory.PhysicalAllocator = undefined;

pub var virtual_memory: memory.VirtualMemory = undefined;

pub var heap_allocator: memory.HeapAllocator = undefined;

pub var kernel_allocator: std.mem.Allocator = undefined;

pub var frame_buffer: KernelFramebuffer = undefined;

pub var pci_devices: []pci.PCIDevice = undefined;

pub var disk_controller: union(enum) {

ahci: *ahci.AHCIController,

nvme: *nvme.NVMeController,

none,

} = .none;

pub var filesystem: ?*fat32.FAT32 = null;
pub var audio_system: ?*hda.AudioSystem = null;

pub var input_manager: ?*xhci.InputManager = null;

pub var graphics_2d: graphics.Framebuffer = undefined;

pub var graphics_3d: ?graphics.GraphicsContext3D = null;

var network_enabled: bool = false;
```

Entry Point:

```
export fn kernelMain(

memory_map_ptr: *MemoryMap,

framebuffer_ptr: *KernelFramebuffer,

) callconv(.c) noreturn
```

Initialization Stages:

Stage 1 - CPU Core:

```
fn initCore() void
```

- Setup IDT
- Enable interrupts

Stage 2 - Memory:

```
fn initMemory(memory_map_ptr: *MemoryMap) !void
```

- Initialize physical allocator from UEFI memory map
- Setup virtual memory (4-level paging)
- Identity map first 4GB + framebuffer
- Create kernel heap at 4GB
- Activate new page tables

Stage 3 - Hardware Discovery:

```
fn initHardware() !void
```

- Parse ACPI tables
- Initialize Local APIC

- Disable legacy PIC
- Initialize I/O APIC
- Initialize timer
- Enumerate PCI devices
- Route IRQs to core 0:
  - IRQ 0 (timer) → vector 32
  - IRQ 1 (keyboard) → vector 33
  - IRQ 11 (network) → vector 43
  - IRQ 12 (mouse) → vector 44

Stage 4 - Storage:

```
fn initStorage() !void
```

- Try NVMe first
- Fallback to AHCI if NVMe unavailable
- Initialize FAT32 filesystem

Stage 5 - Peripherals:

```
fn initPeripherals() !void
```

- Initialize PS/2 controller
- Initialize USB (XHCI) if available
- Initialize HDA audio if available
- Setup 2D graphics framebuffer
- Initialize 3D graphics context

Stage 6 - Network:

```
fn initNetwork() !void
```

- Create initial network config
- Initialize NIC
- Initialize all protocol layers
- Start DHCP if enabled

Stage 7 - Threading:

```
fn initThreading() !void
```

- Initialize scheduler for each CPU core

Stage 8 - Main Loop:

```
fn runMainLoop() noreturn
```

- Clear screen
- Initialize test mode state
- Enter infinite loop:
  1. Poll PS/2 for input
  2. Handle mode switching (TAB key)
  3. Handle network tests (P key for ping)
  4. Render current mode (2D/3D/Network/DHCP)
  5. Update protocol timers (ARP, TCP, DHCP, DNS)
  6. Yield to scheduler

Test Modes:

- Mode 0: 2D graphics test (color, mouse cursor)
- Mode 1: 3D rendering (rotating cube)
- Mode 2: Network test (UDP/TCP sockets, stats)
- Mode 3: DHCP status display

Panic Handler:

```
fn panic(msg: []const u8) noreturn
```

- Fills screen with red
- Halts all CPUs

Linker Script

linker.ld

Purpose: Define kernel memory layout

```
OUTPUT_FORMAT(elf64-x86-64)
```

```
ENTRY(kernelMain)

SECTIONS {

. = 1M;          /* Start at 1MB */

.text : {

    *(.text)     /* Code */

}

.rodata : {

    *(.rodata)  /* Read-only data */

}

.data : {

    *(.data)     /* Initialized data */

}

.bss : {

    *(.bss)      /* Uninitialized data */

}

}
```

System Flow Overview

Boot Sequence

1. **UEFI Firmware** loads `BOOTX64.efi`
2. **Bootloader**:
    - Initializes GOP and filesystem protocols
    - Loads kernel ELF from disk
    - Parses ELF headers
    - Allocates memory for segments

    - Copies segments to physical addresses
    - Obtains memory map
    - Exits boot services
    - Jumps to kernel entry point
3. **Kernel**: 8-stage initialization
4. **Main Loop**: Event handling and rendering

Interrupt Flow

```
Hardware → CPU → IDT → ISR stub (isr_stubs.s) → interruptHandler (idt.zig) → Sp
```

Example - Keyboard:

```
Keyboard pressed → IRQ 1 → Vector 33 → isrKeyboard → interruptHandler → ps2.han
```

Example - Network:

```
Packet received → IRQ 11 → Vector 43 → isrIRQ11 → interruptHandler → net.proces
```

Network Packet Flow

Receive:

```
NIC → RX descriptor ring → net.processNetworkIRQ → net.processPacket

  ↓

EtherType 0x0806 → arp.processPacket

EtherType 0x0800 → ip.processPacket

  ↓

Protocol 1 → icmp.processPacket

Protocol 6 → tcp.processPacket

Protocol 17 → udp.processPacket

  ↓

Port-based routing → Socket queue or protocol handler (DHCP, DNS)
```

Transmit:

```
Application → udp.send/tcp.send
  ↓
ip.buildPacket
  ↓
arp.resolve (get MAC for IP)
  ↓
net.sendEthernet
  ↓
nic.transmit → TX descriptor ring → Hardware
```

Memory Allocation Flow

```
Application requests memory
  ↓
kernel_allocator (ZigAllocator)
  ↓
heap_allocator.alloc
  ↓
virtual_memory.mapPage
  ↓
phys_allocator.allocFrame
  ↓
Physical RAM
```

Thread Scheduling Flow

```
Timer interrupt (IRQ 0)

  ↓

threading.tick()

  ↓

scheduler.yield()

  ↓

scheduler.schedule() → picks next thread

  ↓

contextSwitch(old_context, new_context)

  ↓

New thread runs
```

Module Dependencies

```
kernel.zig

 ├─ boot.zig (external - provides memory map and framebuffer)

 ├─ idt.zig

 │   ├─ isr_stubs.s

 │   ├─ apic.zig (sendEOI)

 │   └─ threading.zig (tick)

 ├─ memory.zig

 │   └─ [uses UEFI memory map]

 ├─ acpi.zig
```

```
│   └─ port_io.s (inb/outb for PM timer)
├─ apic.zig
│   ├─ acpi.zig (get APIC addresses)
│   └─ port_io.s (legacy PIC disable)
├─ pci.zig
│   └─ port_io.s (CONFIG_ADDRESS/CONFIG_DATA)
├─ timer.zig
│   ├─ acpi.zig (PM timer)
│   └─ port_io.s
├─ threading.zig
│   ├─ context_switch.s
│   ├─ apic.zig (getAPICID)
│   └─ memory.zig (allocateStack)
├─ ps2.zig
│   └─ port_io.s
├─ ahci.zig
│   ├─ pci.zig
│   └─ port_io.s
├─ nvme.zig
│   ├─ pci.zig
│   └─ port_io.s
├─ fat32.zig
│   └─ disk.zig (interface to AHCI/NVMe)
```

```
├─ xhci.zig
│   └─ pci.zig
├─ hda.zig
│   └─ pci.zig
├─ graphics.zig
│   └─ [uses framebuffer]
├─ net.zig
│   ├─ nic.zig
│   ├─ arp.zig
│   ├─ ip.zig
│   ├─ udp.zig
│   ├─ tcp.zig
│   ├─ icmp.zig
│   ├─ dhcp.zig
│   └─ dns.zig
├─ nic.zig
│   └─ pci.zig
├─ arp.zig
├─ ip.zig
│   ├─ icmp.zig
│   ├─ tcp.zig
│   └─ udp.zig
├─ udp.zig
```

```
│   ├─ dhcp.zig
│   ├─ dns.zig
│   └─ net.zig (sendIP)
├─ tcp.zig
│   └─ net.zig (sendIP)
├─ dhcp.zig
│   ├─ net.zig (setConfig)
│   └─ udp.zig
├─ dns.zig
│   └─ udp.zig
├─ http.zig
│   ├─ tcp.zig
│   └─ dns.zig
├─ silosec.zig
└─ demo.zig
└─ [all subsystems for testing]
```

Critical Implementation Notes

Assembly Workaround

Due to Zig 0.15.1 inline assembly bugs, three assembly files required:

- **port_io.s**: I/O port operations
- **isr_stubs.s**: Interrupt handlers
- **context_switch.s**: Thread switching

All Zig files using these declare `extern fn` functions.

Calling Conventions

- ISR functions: `callconv(.c)` (changed from `.naked`)
- Context switch: System V AMD64 ABI (rdi, rsi for first two args)
- Kernel entry: `callconv(.c)`

Memory Safety

- No virtual memory initially (identity mapped)
- Page tables activated in Stage 2
- Heap at 4GB to avoid conflicts
- All allocations through kernel_allocator

Interrupt Routing

All hardware IRQs routed to core 0 to avoid synchronization issues:

- Timer, keyboard, mouse, network all on core 0
- Work distribution creates threads on other cores

Network Thread Safety

- Single network thread processes all packets
- Socket registries protected by spinlocks
- DHCP state updates require net.setConfig() lock

This completes the comprehensive documentation of all Silo OS modules.

References

[**HOWL-COMP-3-2026**] Silo OS. Github: https://github.com/ghowland/cks/tree/main/papers/COMP/HOWL-COMP-3-2026

[**HOWL-COMP-1-2026**] Implementing a Tall-Infra Data-Only Execution System . Github: https://github.com/ghowland/cks/tree/main/papers/COMP/HOWL-COMP-1-2026

[**HOWL-COMP-2-2026**] Tall-Infra, Data-Only Development. Github: https://github.com/ghowland/cks/tree/main/papers/ COMP-2-2026

[**HOWL-COMP-4-2026**] Geometric Security. Github: https://github.com/ghowland/cks/tree/main/papers/COMP/HOWL COMP-4-2026

[**HOWL-COMP-5-2026**] Partial Geometric Security. Github: https://github.com/ghowland/cks/tree/main/papers/COMP/
COMP-5-2026