

MATH UNIKERNEL

MANUAL & DOCUMENTATION

Table of Contents

Project Plan	4
Website Documentation	4
OS Build	4
Prerequisites	4
Building the OS	4
OS Usage	5
Running in QEMU	5
Running on Hardware	6
VSCoDe Build and Install	6
Installing the Extension	6
VSCoDe Usage	7
Developing a Workload	7
Streaming a Workload Manually	8
OS Components	8
gtd.c (Global Descriptor Table)	8
Overview	8
System Integration & Initialization	8
Application Example	9
Macros	9
Data Structures	9
Functions	10
Overview	13
System Integration & Initialization	13
Application Example	14
pci.c (Peripheral Component Interconnect)	20
Overview	20
System Integration & Initialization	20
Application Example	21
Macros	21
Functions	21
Internal Helpers	22
network.c (Network Layer)	23
Overview	23
System Integration & Initialization	23
Application Example	23
Macros	24
Functions	24
loader.c (Payload Loader)	26
Overview	26

System Integration & Initialization	26
Application Example	27
Macros	27
Functions	27

Project Plan

The project plan that drove the development of the project is found separately, linked below:

<https://drive.google.com/file/d/190UzmKzdxWjfvG0Znb-MgbkVv2vBNyGM/view?usp=sharing>

Website Documentation

Project website:

<https://math-unikernel-documentation.readthedocs.io/en/latest/>

OS Build

This section details the prerequisites and steps for building the Math Unikernel OS.

Prerequisites

Tool	Purpose	Install
x86_64-elf-gcc / x86_64-elf-ld	Cross-compiler for bare-metal x86-64	Arch: pacman -S x86_64-elf-gcc
xorriso	Builds the bootable ISO	Arch: pacman -S xorriso · Ubuntu: apt install xorriso
qemu-system-x86_64	Emulated testing	Arch: pacman -S qemu-system-x86 · Ubuntu: apt install qemu-system-x86
Python 3 + Scapy	Workload streaming	pip install scapy (requires root or CAP_NET_RAW)
VSCode 1.115.0+	Extension host	code.visualstudio.com
Node.js 18 LTS+	Build extension from source (optional)	nodejs.org

Building the OS

To compile the kernel and build the bootable ISO:

Shell

```
cd math_unikernel_os
```

```
# 1. Compile the kernel
make
# Produces: kernel.elf

# 2. Build the bootable ISO
./build-iso.sh
# Produces: math_unikernel.iso
```

A convenience script is also available that runs all three main steps (compile, build ISO, and run in QEMU) at once:

```
Shell
./run_all.sh
```

OS Usage

This section describes how to run the built OS in an emulator or on physical hardware.

Running in QEMU

To run the OS in the QEMU emulator for testing:

```
Shell
cd math_unikernel_os
./qemu.sh
```

QEMU is configured with no display window (serial → stdout), 512 MB RAM, and one RTL8139 NIC on the user network (NAT). The kernel will stay in the POLLING state until a valid workload is received. Press Ctrl+C to stop.

Expected serial output after boot:

```
None
[OK] Limine handshake
[OK] GDT initialized
[OK] IDT initialized
[OK] PMM initialized
[OK] VMM initialized
[OK] Display initialized
[OK] Serial driver initialized
```

```
[OK] Network controller found
[OK] Timer calibrated
[POLLING] Waiting for workload...
```

Note: Emulated NIC drivers for QEMU are deprecated. The OS loop will not run to completion when emulated.

Running on Hardware

1. **Flash the ISO to a USB drive** (destructive, double-check the device path):

```
Shell
sudo ./flash_drive.sh /dev/sdX
```

2. **Boot the target machine** from the USB drive. The kernel prints its initialization log over the serial port (COM1, 115200 8N1). Connect a serial cable or USB-to-serial adapter from the host to the target to see output.
3. **Note the MAC address** printed on boot (e.g., [OK] Network controller found – MAC: de:ad:be:ef:00:01). Set this value as `mathUnikernel.targetMac` in the VSCode extension settings (or pass it directly to `send.py`).
4. **Connect the host and target** on the same Ethernet segment (direct cable or unmanaged switch). Set `mathUnikernel.interface` to the host NIC facing the target.

The kernel will stay in the POLLING state indefinitely until a workload is received.

VSCode Build and Install

The Math Unikernel VSCode extension provides a sidebar panel ("MU Controls") with **Compile** and **Stream** commands to streamline the workload development loop.

Installing the Extension

Option A — Install from pre-built out/ directory

Use the following commands to copy the extension into your VSCode extensions directory:

- **Linux / macOS:**

```
Shell
cp -r math_unikernel_vscode_extension ~/.vscode/extensions/mu-0.0.1
```

- **Windows (PowerShell):**

Shell

```
Copy-Item -Recurse math_unikernel_vscode_extension  
"$env:USERPROFILE\.vscode\extensions\mu-0.0.1"
```

After copying the files:

1. Restart VSCode.
2. Open **Extensions** (Ctrl+Shift+X) and search for math-unikernel-VSCode to confirm installation.
3. Configure via **File** → **Preferences** → **Settings** (Ctrl+,), search Math Unikernel:
 - a. mathUnikernel.targetMac — MAC address shown by the kernel on boot.
 - b. mathUnikernel.interface — your host NIC (e.g., eno1, eth0).

Option B — Build from TypeScript source

To build the extension from source:

Shell

```
cd math_unikernel_vscode_extension  
npm install  
npm run compile # writes output to out/
```

Then follow Option A to install. To develop with hot-reload, open the folder in VSCode and press **F5** (uses .vscode/launch.json).

VSCode Usage

This section describes how to develop and stream a workload using the VSCode extension.

Developing a Workload

1. Open your workload project folder in VSCode. The folder must contain workload.c and an sdk/ subdirectory (kernel_api.h, linker_script.ld, Makefile). Copy sdk/ from math_unikernel_vscode_extension/sdk/.
2. The **MU Controls** panel appears at the bottom of the Explorer sidebar with **Compile** and **Stream** buttons.
3. Write your workload using the MAIN macro as the entry point.
4. Click **Compile** (or Ctrl+Shift+P → MU: Compile). This runs the SDK Makefile and produces workload.bin.
5. With the unikernel running, click **Stream** (or MU: Stream). The extension calls send.py to transmit workload.bin to the configured MAC over the configured interface.
6. Observe output in the QEMU terminal or serial console.

Streaming a Workload Manually

The streaming protocol involves a handshake frame with magic bytes 0x474F2121 ("GO!!"), followed by an 8-byte little-endian integer specifying the payload byte count, and then the binary is chunked into 1486-byte payloads sent as raw Ethernet frames with EtherType 0x88B5.

To stream manually without the VSCode extension:

Shell

```
sudo python3 math_unikernel_vscode_extension/python/send.py \
workload.bin <target_mac> <interface>
```

Example:

```
sudo python3 send.py workload.bin de:ad:be:ef:00:01 eno1
```

Root privileges (or CAP_NET_RAW) are required for Scapy to send raw Ethernet frames.

OS Components

gtd.c (Global Descriptor Table)

Overview

The Global Descriptor Table (GDT) is a core x86 data structure that defines the memory segments used during program execution. In 64-bit long mode, the legacy concepts of memory "segmentation" (base and limits) are largely ignored by the CPU, but a valid GDT is still strictly required to define the execution privileges (ring levels) and segment types.

This module sets up a foundational 64-bit flat memory model for the unikernel. It initializes three required segments: a Null descriptor, a Ring 0 Kernel Code segment, and a Ring 0 Kernel Data segment. It also provides the underlying assembly routine to load the table pointer into the CPU and perform a far jump to flush the legacy segment registers.

System Integration & Initialization

Within the unikernel's boot sequence, the GDT is initialized in `kernel_main()` immediately following the Limine bootloader handshakes and the legacy PIC remapping (`pic_remap(32, 40)`).

It serves as the primary hardware initialization step, strictly preceding the setup of the Interrupt Descriptor Table (IDT), Physical Memory Manager (PMM), and Virtual Memory Manager (VMM). The kernel tracks the successful initialization of the GDT using a bitwise state flag (`init_status |= gdt_init()`). If the GDT fails to initialize, the kernel will print a failure log and immediately halt the CPU via `hcf()`. Hardware interrupts are safely enabled only after the GDT and all other boot structures are fully validated.

Application Example

The following example demonstrates how the GDT is invoked during the kernel's early boot sequence.

```
#include "headers/kernel_api.h"
#include "headers/gdt.h"

void kernel_main(void) {
    uint16_t init_status = 0;

    // ... Limine handshakes and PIC remapping ...

    // Initialize the GDT and track its success state
    init_status |= gdt_init();

    // Proceed with further architectural setup
    init_status |= idt_init();
    init_status |= pmm_init(memmap_request.response);

    // Validate GDT initialization
    if(init_status & GDT_INIT_SUCCESS) {
        PRINTS("READY: GDT has been initialized and loaded.\n");
    } else {
        PRINTS("ERROR: GDT initialization failed.\n");
        hcf(); // Halt CPU on fatal error
    }
}
```

Direct API References:

Macros

- `GDT_INIT_SUCCESS`: Evaluates to 2 ($1 \ll 1$). Returned upon successful loading and flushing of the GDT.

Data Structures

- `typedef uint64_t descriptor`
 - Represents a single, raw 64-bit entry within the Global Descriptor Table.
- `struct gdt_ptr`
 - The highly specific, packed pointer structure required by the `lgdt` CPU instruction to locate and load the table.
 - Fields:

- `uint16_t` limit: The total size of the GDT in bytes, minus 1.
 - `uint64_t` base: The 64-bit linear memory address pointing to the first entry of the GDT array.
- `struct gdt_context`
 - A state container that holds both the active descriptor array and its corresponding CPU pointer.
 - Fields:
 - `descriptor entries[3]`: The array of GDT entries (Null, Code, Data).
 - `struct gdt_ptr` pointer: The pointer structure referencing the entries array.

Functions

- `uint8_t gdt_init()`
 - Initializes the Global Descriptor Table context. It populates the Null, Kernel Code (0x9a access, 0xa flags), and Kernel Data (0x92 access, 0x0 flags) descriptors using specific access bytes and flags required for x86-64 long mode. It calculates the table limit, sets the base pointer, and internally calls `load_gdt()` to activate it.
 - Returns: `GDT_INIT_SUCCESS` upon completion.
- `descriptor create_descriptor(uint8_t access_byte, uint8_t flags)`
 - Constructs a 64-bit GDT entry. Because 64-bit mode enforces a flat memory model, the legacy base and limit fields are bypassed. This helper function shifts the `access_byte` (to bit 40) and flags (to bit 52) into their correct architectural positions within the 64-bit integer.
 - Parameters:
 - `access_byte`: The byte determining ring level (DPL), executable status, and segment type.
 - `flags`: The nibble determining 64-bit/32-bit sizing and granularity.
 - Returns: A fully formatted 64-bit descriptor.
- `void load_gdt(struct gdt_ptr* ptr)`
 - Note: This function is implemented in assembly (`gdt_load.S`).
 - Executes the `lgdt` instruction to load the new table pointer into the CPU. It then manually updates the data segment registers (`ds`, `es`, `fs`, `gs`, `ss`) with the Data Segment selector (0x10) and performs a far return (`lretq`) pushing the Code Segment selector (0x08) to flush and reload the CS register.
 - Parameters:
 - `ptr`: A pointer to the packed `gdt_ptr` structure containing the table's limit and base address.

idt.c (Interrupt Descriptor Table)

Overview

The Interrupt Descriptor Table (IDT) is an x86 data structure used by the CPU to determine the correct response to exceptions (like division by zero or page faults) and hardware interrupts (like

keyboard input or system timers). In 64-bit long mode, the CPU requires 16-byte (128-bit) IDT entries to accommodate 64-bit memory addresses.

This module initializes the IDT with the first 48 vectors (32 CPU exceptions and 16 hardware IRQs mapped from the legacy PIC). It provides a central C-level dispatcher that catches all interrupts, halting the system and dumping registers upon a fatal exception, or dynamically routing hardware IRQs to registered driver callbacks.

System Integration & Initialization

The IDT is initialized within `kernel_main()` immediately after the Global Descriptor Table (GDT) is successfully loaded. This strict ordering is required because every IDT entry must reference a valid GDT Code Segment selector (in this kernel, 0x08).

The kernel tracks the successful initialization of the IDT using the `init_status` bitwise flag. If it fails, the system prints an error log and halts. Importantly, the CPU's maskable hardware interrupts are kept disabled during this setup; `__asm__ volatile("sti");` is only executed at the very end of the `kernel_main()` boot sequence once all core systems (PMM, VMM, Display) are fully online.

Application Example

The IDT is initialized once during boot, but the API allows other kernel modules (like a keyboard driver) to register their own IRQ handlers later.

```
#include "headers/idt.h"

// 1. Initialization in kernel_main()
void kernel_main(void) {
    // GDT must be initialized first!
    uint16_t init_status = gdt_init();

    // Initialize the IDT and load the pointer
    init_status |= idt_init();

    // ... complete rest of boot sequence ...

    // Enable CPU interrupts
    __asm__ volatile("sti");
}

// 2. Registering a driver handler (e.g., in a keyboard driver)
void keyboard_driver_init(void) {
```

```
// Register the keyboard handler to IRQ 1 (Vector 33)
register_irq_handler(1, keyboard_callback);
}
```

Direct API References

Macros

- `IDT_INIT_SUCCESS`: Evaluates to 4 ($1 < 2$). Returned upon the successful loading of the IDT pointer into the CPU.

Data Structures

- `struct idt_entry`
 - Represents a packed, 16-byte descriptor for a single interrupt vector in 64-bit mode.
 - Fields:
 - `isr_low`, `isr_mid`, `isr_high`: The split components of the 64-bit virtual address pointing to the interrupt service routine (ISR).
 - `kernel_cs`: The GDT segment selector (hardcoded to 0x08 for the Kernel Code Segment).
 - `attributes`: The flags determining gate type (Interrupt or Trap) and execution privileges (DPL).
 - `ist`: The Interrupt Stack Table offset (set to 0).
- `struct idt_ptr`
 - The specific, packed pointer structure required by the `lidt` CPU instruction.
 - Fields:
 - `limit`: The size of the active IDT array in bytes, minus 1.
 - `base`: The 64-bit linear memory address of the first `idt_entry`.
- `struct interrupt_frame`
 - Defines the exact layout of the CPU registers as they are pushed onto the stack during an interrupt. This is used to read the state of the machine at the exact moment an exception occurred.
 - Fields: Includes general-purpose registers (r15 through rax), the interrupt vector number, the hardware error code, and the automatic hardware-pushed state (rip, cs, rflags, rsp, ss).
- `struct idt_context`
 - The global state container holding the active 256-entry array, the `lidt` pointer, and a statistical counter for total interrupts fired.

Functions

- `uint8_t idt_init()`
 - Zeroes out the entire 256-entry IDT array, then populates the first 48 vectors. It wires these entries to the assembly routines found in the `isr_stub_table`, applies full kernel authority flags (0x8e), calculates the table limit, and invokes `load_idt()`.
 - Returns: `IDT_INIT_SUCCESS`.

- `void register_irq_handler(uint8_t irq, void* handler)`
 - Registers a custom function pointer to be executed when a specific hardware IRQ fires.
 - Parameters:
 - `irq`: The hardware IRQ number (0-15). (Note: IRQ 0 corresponds to CPU vector 32).
 - `handler`: A pointer to the driver's handler function.
- `void interrupt_dispatcher(struct interrupt_frame* frame)`
 - The central C-level interrupt router called by the assembly stubs.
 - Behavior:
 - Vectors 0-31 (Exceptions): Triggers a "MATH UNIKERNEL PANIC". Prints the faulting instruction pointer (RIP), the error code, and a complete dump of all general-purpose registers before halting the CPU via `hcf()` to prevent corruption.
 - Vectors 32-47 (Hardware IRQs): Calculates the relative IRQ number, invokes the custom function pointer in `irq_handlers` (if one was registered), and explicitly sends an End of Interrupt (EOI) signal to the programmable interrupt controller (`pic_send_eoi`) so it can unmask future interrupts.
- `void load_idt(struct idt_ptr* ptr)`
 - (Implemented in assembly). Executes the `lidt` instruction to inform the CPU of the newly constructed interrupt table's location and size.

*** `idt.c` and `gdt.c` both are involved in `isr.S`

isr.S

Overview

The ISR assembly file (`isr.S`) provides the critical low-level bridge between the hardware CPU interrupts and the higher-level C kernel. When an exception or hardware IRQ occurs, the CPU suspends current execution and jumps to an address defined in the IDT.

Because C functions expect a specific calling convention (System V AMD64 ABI), the kernel cannot jump directly into C code. This module safely catches the hardware jump, saves a complete snapshot of the CPU's current register state to the stack, standardizes the stack frame (accounting for varying hardware error codes), and carefully hands execution over to the C-level `interrupt_dispatcher` before restoring the system and resuming the original math workload.

System Integration & Initialization

This file does not have an initialization function of its own; rather, it exposes a global array of pointers (`isr_stub_table`). During the unikernel boot sequence, `idt_init()` iterates through this array and wires each of the first 48 IDT entries to its corresponding `isr_stub_[vector]`.

When an interrupt fires, the execution flow is strictly: Hardware Trigger $\rightarrow$ IDT Entry $\rightarrow$ `isr_stub_[vector]` $\rightarrow$ `isr_common` $\rightarrow$ `interrupt_dispatcher` (C code). After the C code returns, execution falls back to `isr_common` to perform the hardware return (`iretq`).

Application Example

While this file is purely assembly, its structures are directly imported and utilized by the IDT module during kernel setup.

```
#include "headers/idt.h"

// The external array defined in the .data section of isr.S
extern void* isr_stub_table[];

uint8_t idt_init() {
    // Wire the generated assembly stubs into the IDT descriptors
    for(uint8_t vector = 0; vector < 48; vector++) {
        idt_set_descriptor(vector, (uint64_t)isr_stub_table[vector], 0x8e);
    }

    // ... load the IDT ...
}
```

Global Tables

- `void* isr_stub_table[]`
 - An array of 48 memory pointers, exposed globally to the linker. Each index corresponds to an interrupt vector (0-47) and points to the specific memory address of that vector's executable stub (`isr_stub_0`, `isr_stub_1`, etc.).

Assembly Macros

- `isr_no_err` vector
 - Generates an entry stub for CPU exceptions and hardware IRQs that do not push a hardware error code. It pushes a dummy error code (0) to ensure the stack frame aligns perfectly with the C `interrupt_frame` structure, pushes the vector number, and jumps to `isr_common`.
- `isr_err` vector
 - Generates an entry stub for specific CPU exceptions (like Page Faults or Double Faults) where the hardware automatically pushes an error code. It simply pushes the vector number and jumps to `isr_common`.

Execution Routines

- `void load_idt(struct idt_ptr* ptr)`
 - Executes the `lidt (%rdi)` instruction. In the System V AMD64 ABI, the first argument of a C function call is placed in the `%rdi` register, which this routine directly passes to the CPU to load the new IDT.

- `isr_common`
 - The universal bridge handling the context switch between the math workload and the kernel interrupt dispatcher.
 - Behavior:
 1. State Preservation: Pushes all 15 general-purpose registers (`%rax` through `%r15`) onto the stack to freeze the current program's state.
 2. C Dispatch: Copies the current stack pointer (`%rsp`) into the first argument register (`%rdi`). This allows the C function `interrupt_dispatcher(struct interrupt_frame* frame)` to read the stack exactly as it was laid out.
 3. Restoration: Once the C router finishes, it pops all 15 registers off the stack in reverse order, cleans up the 16 bytes used by the vector number and error code, and executes `iretq` to cleanly resume the interrupted unikernel execution.
- `isr_stub_[0-47]`
 - The 48 dynamically generated labels that act as the actual entry points for the IDT.

kernel.c and kernel_api.h

Overview

The `kernel.c` module serves as the primary entry point and central orchestrator for the Math Unikernel. It is responsible for accepting the hardware state handed over by the Limine bootloader, initializing all core CPU and memory subsystems, and transitioning the system into an infinite state machine loop designed to ingest, execute, and return mathematical workloads.

Coupled with `kernel_api.h`, this module establishes a fixed-address Application Binary Interface (ABI). This allows dynamically downloaded external payloads to invoke internal kernel functions (like SIMD-accelerated matrix multiplication or huge page allocations) without requiring a traditional operating system system-call (`syscall`) interface.

System Integration & Boot Sequence

The execution strictly begins at `kernel_main()` after Limine sets up the initial 64-bit environment.

The boot sequence follows a strict dependency chain:

1. Bootloader Handshake: Validates memory maps, kernel base addresses, and framebuffers provided by Limine.
2. Hardware Initialization: Remaps the legacy PIC, then initializes the GDT, IDT, Physical Memory Manager (PMM), and Virtual Memory Manager (VMM).
3. Subsystem Bring-up: Enables CPU SIMD instructions (AVX/SSE) for math acceleration, configures the display, initializes serial drivers for I/O, and scans the PCI bus for the network controller.
4. API Mapping: Maps the `kernel_api_t` structure to a strict memory address (`0x10000000`) and populates it with kernel function pointers.

5. Hardware Interrupts: Executes `__asm__ volatile("sti");` to allow the network/serial drivers to start catching incoming data.

The Unikernel State Machine

Once initialized, the kernel drops into an infinite do-while loop representing the unikernel's lifecycle:

- POLLING: The kernel waits to receive a "magic number" over the I/O interface. It then polls for the expected byte size of the incoming workload, allocates the necessary 2MB huge pages via the VMM, and downloads the raw executable payload directly into memory.
- EXECUTING: The kernel casts the downloaded memory block into a function pointer (`void (*workload)()`) and directly executes the untrusted payload in Ring 0. The payload utilizes the `kernel_api_t` to perform its math operations.
- EXTRACTING: After the payload returns, the kernel prepares to extract the results (stored in the API's output buffer) and resets the state back to POLLING.

Direct API References

Macros

- `KERNEL_API_ADDRESS`: Hardcoded to `0x10000000`. The fixed virtual memory address where the kernel exposes its function pointers to the external payloads.

Data Structures

- `kernel_api_t`
 - The central interface struct that grants the external mathematical payload access to kernel-level subroutines.
 - Function Pointers:
 - `alloc_huge_page`: Grants the payload the ability to allocate 2MB continuous pages.
 - `dot_product`: A pointer to the kernel's hardware-accelerated dot product function.
 - `matrix_multiply`: A pointer to the kernel's hardware-accelerated matrix multiplication function.
 - Fields:
 - `void* output_buffer`: A pointer configured by the payload to indicate where its final computed results are stored.
 - `uint64_t output_size`: The byte size of the payload's final results.

Functions

- `void kernel_main(void)`
 - The main C entry point called by the Limine bootloader. It orchestrates the entire hardware initialization sequence, maps the `kernel_api_t`, and traps the CPU in the Polling/Executing/Extracting state machine loop.

pmm.c (Physical Memory Manager)

Overview

The Physical Memory Manager (PMM) is responsible for keeping track of the actual, raw hardware RAM available to the system. It uses a bitmap data structure to represent the entire physical memory space, where each bit corresponds to a single 4KB frame of memory. If a bit is 1, the frame is in use; if it is 0, the frame is free.

The PMM does not deal with virtual addresses or CPU protections; its sole job is to parse the hardware memory map provided by the bootloader, reserve spaces that are occupied by hardware (like the bootloader or the kernel itself), and hand out contiguous blocks of free RAM when requested by higher-level systems.

System Integration & Initialization

The PMM is initialized in the main kernel boot sequence immediately after the GDT and IDT. It relies heavily on the `limine_memmap_response` passed directly from the bootloader to understand the hardware layout.

During `pmm_init()`, the kernel scans the memory map to find the highest usable physical address and calculates how large the bitmap needs to be. It then searches for the first available block of memory large enough to hold this bitmap and places it there. By default, the entire bitmap is initialized to 0xFF (fully used/locked), and then the PMM explicitly iterates through the usable sections of the memory map to "free" the valid frames.

API Reference

Macros

- `PMM_INIT_SUCCESS`: Evaluates to 8 ($1 < 3$). Returned upon successful initialization of the physical bitmap.

Data Structures

- `struct pmm_context`
 - The global state container for the physical memory subsystem.
 - Fields:
 - `uint8_t* bitmap`: A pointer to the dynamically placed bitmap array.
 - `uint64_t bitmap_size`: The total size of the bitmap in bytes.
 - `uint64_t total_frames`: The total number of 4KB physical frames tracked by the system.
 - `uint64_t free_frames`: A running counter of currently available frames.
 -

Functions

- `uint8_t pmm_init(struct limine_memmap_response* response)`
 - Parses the bootloader's memory map to locate available RAM, sizes and places the bitmap structure, and marks all usable and bootloader-reclaimable memory as free.
 - Returns: `PMM_INIT_SUCCESS`.
- `uint64_t pmm_alloc()`

- Scans the bitmap using a "first fit" algorithm to find the first available single 4KB frame. It flips the corresponding bit to claim it.
 - Returns: The 64-bit physical address of the allocated 4KB frame (or 0 if out of memory).
- `uint64_t pmm_alloc_2mb()`
 - Scans the bitmap to find 64 contiguous free 4KB frames (which equates to exactly 2MB of memory) aligned to a 2MB boundary.
 - Returns: The physical address of the 2MB block.
- `void pmm_free(uint64_t phys_addr) / void pmm_free_2mb(uint64_t phys_addr)`
 - Converts the given physical address into a frame index and flips the corresponding bit(s) in the bitmap back to 0, freeing the memory for future use. Ensures the address is properly aligned to 4KB (0xfff) or 2MB (0x200000), respectively.

vmm.c (Virtual Memory Manager)

Overview

The Virtual Memory Manager (VMM) creates the illusion of a clean, continuous memory space for software to run in, isolating the CPU from the fragmented reality of physical RAM. It implements standard x86-64 4-level paging (PML4, PDPT, PD, PT) to translate 64-bit virtual addresses into the physical addresses managed by the PMM.

In this unikernel, the VMM is crucial because it implements the Higher Half Direct Map (HHDM) architecture and supports 2MB "Huge Pages". Huge pages drastically reduce Translation Lookaside Buffer (TLB) misses, providing a massive performance boost for the mathematically intensive, data-heavy workloads this system executes.

System Integration & Initialization

The VMM is initialized immediately after the PMM, as it requires the PMM to allocate physical frames for its page tables.

During `vmm_init()`, the kernel constructs the root PML4 table. It identically maps the lowest 4MB of memory (for legacy hardware access), the kernel and its modules, and maps all usable memory into the Higher Half using the HHDM offset provided by Limine. Finally, it writes the physical address of the PML4 table into the CPU's CR3 control register, activating the new paging structure.

Application Example

The `vmm_alloc_huge_page` function is directly exposed via the `kernel_api_t` interface, allowing external payloads to allocate contiguous 2MB blocks of memory for their math operations.

```

#include "headers/kernel_api.h"

// Example of a payload using the API to get memory
void math_workload() {
    kernel_api_t* api = (kernel_api_t*)KERNEL_API_ADDRESS;

    // Allocate 4 MB of memory (two 2MB huge pages) with Read/Write flags
    float* massive_matrix = api->alloc_huge_page(2, VMM_PRESENT |
VMM_WRITEABLE);

    // ... perform calculations ...
}

```

API Reference

Macros

- Flags: Standard paging attributes such as VMM_PRESENT (1<<0), VMM_WRITEABLE (1<<1), and VMM_HUGE (1<<7) used to dictate the access rights of a given page.

Data Structures

- typedef uint64_t pt_entry_t
 - A 64-bit page table entry containing the physical address of the next table or memory frame, combined with access flags.
- page_table_t
 - An array of 512 pt_entry_ts, strictly aligned to a 4KB boundary, representing a single level in the paging hierarchy.
- struct vmm_context
 - Holds the root of the paging hierarchy.
 - Fields:
 - page_table_t* pml4_virt: The virtual address of the top-level PML4 table.
 - uintptr_t pml4_phys: The physical address of the PML4 table, designed to be loaded into the CR3 register.

Functions

- uint8_t vmm_init(struct limine_kernel_address_response* kernel_addr_response, struct limine_memmap_response* memmap_response)
 - Sets up the root page tables, maps the kernel code, maps the framebuffer, and implements the Higher Half Direct Map (HHDM) before loading the CR3 register.
 - Returns: VMM_INIT_SUCCESS.
- void vmm_map_virt_to_phys(uint64_t virt_addr, uint64_t phys_addr, uint64_t flags)

- Traverses the 4-level paging hierarchy (allocating new intermediate tables via the PMM if they don't exist) to link a specific 4KB virtual address to a physical frame, applying the requested access flags. Uses `invlpg` to flush the CPU cache for that address.
- `void* vmm_alloc_huge_page(uint64_t num_pages, uint64_t flags)`
 - The premier memory allocation function for the unikernel. It requests properly aligned 2MB blocks from the PMM and maps them directly into the Page Directory (bypassing the Page Table level entirely) using the `VMM_HUGE` flag. It zeroes out the memory to prevent data leaks and increments the global virtual address tracker.
 - Returns: A `void*` pointer to the newly allocated block of virtual memory.

pci.c (Peripheral Component Interconnect)

Overview

The Peripheral Component Interconnect (PCI) bus is the standard hardware interface through which the CPU discovers and communicates with expansion devices such as network controllers, storage adapters, and graphics cards. Each device on the bus is uniquely identified by a combination of bus number, slot number, and function number, and exposes a standardized 256-byte configuration space that the CPU can read to determine the device's vendor, class, and memory-mapped or port-mapped I/O addresses.

This module performs a full enumeration of the PCI bus to locate a supported network controller. Once found, it handles the complete bring-up sequence: waking the device from a low-power sleep state (D3) if necessary, enabling bus mastering and memory-mapped I/O access in the device's command register, resolving the hardware IRQ line, and dispatching initialization to the appropriate NIC driver (Intel I219-LM or Realtek RTL8139) based on the device's vendor ID.

System Integration & Initialization

The PCI bus scan is the final step of the kernel's hardware initialization sequence, called in `kernel_main()` after the display and serial drivers are online. It is positioned last among the boot-time initializations because it is the only subsystem that allocates physical and virtual memory at runtime (via the NIC drivers it dispatches to), requiring a fully operational PMM and VMM. It also registers interrupt handlers, requiring the IDT to already be loaded.

`pci_scan_bus()` returns `PCI_INIT_SUCCESS` if a supported network controller is found and its driver is successfully dispatched, or 0 if no compatible device is detected. The kernel tracks this result in the shared `init_status` bitfield. A missing network controller is a fatal error; the kernel will

print NETWORK_CONTROLLER_MISSING and halt via hcf(), as the entire workload delivery pipeline depends on the NIC being operational.

Application Example

The following example demonstrates how pci_scan_bus() is called in the boot sequence and how its result is validated.

```
#include "headers/pci.h"
#include "headers/kernel_logs.h"

void kernel_main(void) {
    uint16_t init_status = 0;

    // ... display and serial init ...

    // Scan the PCI bus for a supported NIC and initialize it
    init_status |= pci_scan_bus();

    // Validate – a missing NIC is fatal
    if (init_status & PCI_INIT_SUCCESS) {
        PRINTS(NETWORK_CONTROLLER_FOUND);
    } else {
        PRINTS(NETWORK_CONTROLLER_MISSING);
        hcf();
    }
}
```

Direct API References:

Macros

- PCI_INIT_SUCCESS: Evaluates to 256 (1<<8). Returned by pci_scan_bus() when a supported network controller has been found and its driver successfully dispatched.

Functions

- uint32_t pci_read_dword(uint8_t bus, uint8_t slot, uint8_t func, uint8_t offset)
 - The fundamental PCI configuration space accessor. It constructs a 32-bit address from the bus, slot, function, and register offset values and writes it to the PCI CONFIG_ADDRESS I/O port (0xCF8). It then reads the 32-bit result back from the PCI CONFIG_DATA port (0xCFC). All higher-level PCI operations in this module are built on top of this function.
 - Parameters:
 - bus: The PCI bus number (0–255).
 - slot: The device slot number (0–31).

- func: The function number within the device (0–7).
 - offset: The byte offset into the device's configuration space (must be 4-byte aligned; the two lowest bits are masked off).
- Returns: The raw 32-bit value from the requested configuration register. A vendor field of 0xFFFF indicates no device is present at the given bus/slot/function combination.
- uint16_t pci_scan_bus()
 - Iterates over all 256 buses, 32 slots, and 8 functions of the PCI address space. For each present device, it reads the class code (offset 0x08) and filters for devices matching base class 0x02 (Network Controller) and sub-class 0x00 (Ethernet). When a match is found, it performs the full bring-up sequence: calling pci_wake_device() to exit D3, enabling bus mastering and MMIO in the command register, resolving the IRQ line, and dispatching to either i219_init() or rtl8139_init() based on the vendor ID. The scan halts and returns immediately upon finding the first supported NIC.
 - Returns: PCI_INIT_SUCCESS if a supported NIC was found and initialized, or 0 if the full bus was exhausted without finding a compatible device.

Internal Helpers

- void pci_write_dword(uint8_t bus, uint8_t slot, uint8_t func, uint8_t offset, uint32_t data)
 - The write counterpart to pci_read_dword(). Constructs the same 32-bit address and writes the provided data value to the CONFIG_DATA port. Used internally by pci_scan_bus() to enable bus mastering and MMIO in the device command register, and by pci_wake_device() to transition a device out of a low-power state. Not exposed in pci.h.
- void pci_wake_device(uint16_t bus, uint8_t slot, uint8_t func)
 - Walks the device's PCI capabilities linked list (starting from offset 0x34) searching for the Power Management capability (ID 0x01). If found, it reads the Power Management Control/Status Register (PMCSR) and checks the two lowest bits for the current power state. If the device is not already in D0 (fully awake), it clears those bits to force a transition to D0 and inserts a software delay of approximately 10ms to allow the hardware to stabilize. If the device reports no capabilities list, the function returns immediately. Not exposed in pci.h.

network.c (Network Layer)

Overview

The network module sits directly above the NIC drivers in the software stack. Its role is to decouple the rest of the kernel from any specific hardware implementation: the NIC drivers call into this layer when a frame arrives, and this layer handles all Ethernet-level filtering, header parsing, and data routing. The NIC driver in use at runtime is itself registered into this layer via

function pointers, meaning the loader, kernel state machine, and all other subsystems interact exclusively with this module and have no direct knowledge of whether an RTL8139 or an Intel I219-LM is physically present.

The module implements a two-phase receive protocol built around the custom EtherType 0x88B5. All frames carrying a different EtherType are silently discarded. The first accepted frame is treated as a header frame, containing the 4-byte magic number and the 8-byte payload size; its payload is buffered internally for the loader to consume byte-by-byte via `read_ethernet()`. Every subsequent frame after the header is confirmed is treated as a data frame and its payload is written directly and contiguously into a caller-supplied destination buffer, typically a VMM-allocated huge page.

System Integration & Initialization

This module has no dedicated initialization function. It is wired up implicitly during the PCI bring-up sequence: after `pci_scan_bus()` identifies and initializes the NIC hardware, it immediately calls `network_set_poll_fn()` and `network_set_send_fn()` to register the driver's specific polling and transmit routines into the network layer. From that point on, all upper layers interact exclusively with the network API.

At the start of every POLLING cycle in the kernel state machine, `kernel_main()` calls `reset_header()` to clear all internal header state from the previous transaction before handing control to the loader. The loader then calls `unlock()` and `poll_payload_size()`, both of which drive the network layer forward by repeatedly calling `read_ethernet()` until the full magic number and payload size have been consumed from the header buffer. Once the header is fully consumed, the module automatically transitions to direct-write mode for all subsequent data frames.

Application Example

The following example demonstrates the complete data flow through the network layer for a single workload transaction, from NIC driver registration through payload delivery.

```
#include "headers/network.h"
#include "nic_drivers/rtl8139.h"

// Step 1 – pci.c wires the NIC driver into the network layer
rtl8139_init(bar0, irq_line);
network_set_poll_fn(rtl8139_poll);

// Step 2 – kernel.c resets header state at the top of each POLLING cycle
reset_header();

// Step 3 – loader.c consumes the header frame byte-by-byte
unlock();           // reads 4 bytes: magic number
```

```
poll_payload_size()); // reads 8 bytes: payload size (little-endian)

// Step 4 – loader.c sets the destination buffer and waits for all data
frames
network_set_dest(payload_mem, payload_byte_size);
while (network_bytes_received() < payload_byte_size) {
    __asm__ volatile("hlt"); // sleep until next NIC interrupt
}
```

Direct API References:

Macros

- **ETHERTYPE_CUSTOM**: Defined as 0x88B5. The IEEE-reserved EtherType used to identify frames belonging to this system. Any Ethernet frame arriving at the NIC that does not carry this EtherType in bytes 12–13 of the header is silently discarded by `network_receive_frame()` and never reaches the rest of the kernel.

Functions

- **void network_set_poll_fn(void (*fn)())**
 - Registers the active NIC driver's polling function. Once set, this function pointer is called by `read_ethernet()` whenever it needs to spin-wait for the next byte to arrive and no data is currently available. Called once by `pci_scan_bus()` immediately after the NIC driver is initialized.
 - Parameters:
 - `fn`: A function pointer to the NIC driver's poll routine (e.g. `rtl8139_poll` or `i219_poll_rx`).
- **void network_set_send_fn(void (*fn)(uint8_t* data, uint16_t length))**
 - Registers the active NIC driver's frame transmit function. This decouples the result extraction path from any specific hardware implementation, allowing a future EXTRACTING stage to call a single unified send function without knowing which NIC is installed. Currently registered by `pci_scan_bus()` but the outbound transmission path in the EXTRACTING state is reserved for future use.
 - Parameters:
 - `fn`: A function pointer to the NIC driver's send routine, accepting a data buffer pointer and its length in bytes.
- **void network_set_dest(uint8_t* dest, uint64_t expected_bytes)**
 - Configures the destination buffer for incoming payload data. After this call, all data frames passing the EtherType filter are written sequentially into `dest` until the total number of bytes written reaches `expected_bytes`. The internal received byte counter is also reset to zero. Called by `poll_payload()` in `loader.c` after the huge page has been allocated and the payload size is known.
 - Parameters:
 - `dest`: A pointer to the destination memory buffer, typically a VMM-allocated huge page.

- `expected_bytes`: The exact number of payload bytes the network layer should write before considering the transfer complete.
- `void network_receive_frame(uint8_t* data, uint16_t length)`
 - The primary entry point called by NIC driver interrupt handlers when a frame has been received. It first validates the frame is large enough to contain a full Ethernet header (14 bytes) and that the `EtherType` field matches `ETHERTYPE_CUSTOM` (0x88B5); frames failing either check are silently dropped. The remaining logic branches on the internal `hdr_done` flag: if the header has not yet been consumed, the frame's payload is copied into the internal header buffer and `read_ethernet()` is signalled; if the header is done, the payload bytes are written directly and sequentially into the destination buffer registered by `network_set_dest()`.
 - Parameters:
 - `data`: A pointer to the start of the raw Ethernet frame, including the 14-byte Ethernet header.
 - `length`: The total length of the frame in bytes, including the Ethernet header.
- `uint64_t network_bytes_received()`
 - Returns the running count of payload bytes written to the destination buffer since the last call to `network_set_dest()`. Used by `poll_payload()` in `loader.c` as the condition of its blocking wait loop to determine when the full workload has been received.
 - Returns: The total number of payload bytes received into the current destination buffer.
- `uint8_t read_ethernet()`
 - A blocking, byte-oriented reader used exclusively by `loader.c` to consume the header frame one byte at a time. While the header has not yet been fully consumed (`hdr_done` is 0), it spins—calling the registered NIC poll function and halting the CPU between iterations—until a byte is available in the internal header buffer, then returns it. The transition point occurs after the 12th byte is returned: at that point `hdr_done` is set to 1, signalling to `network_receive_frame()` that all subsequent frames should be routed directly to the destination buffer rather than the header buffer.
 - Returns: The next byte from the header buffer.
- `void reset_header()`
 - Clears all internal header state in preparation for a new transaction. Resets the `hdr_done` flag to 0, the header buffer length and read position to 0, and the `rx_ready` signal to 0. Must be called by `kernel_main()` at the top of every POLLING cycle before the loader begins consuming header bytes, to ensure that stale state from a previous transaction does not corrupt the parsing of the next incoming workload.

loader.c (Payload Loader)

Overview

The loader module is responsible for the complete ingestion of an incoming workload from the network. It defines the communication protocol that the sender (`send_data_hardware.py`) and the kernel must mutually agree on, and provides three sequential functions that together drive the POLLING phase of the kernel state machine to completion: authenticating the incoming connection with a magic number handshake, determining the exact byte size of the payload, and then blocking until that payload has been fully received into memory.

All three functions are built entirely on top of the network layer's `read_ethernet()` and `network_set_dest()` APIs. The loader itself has no direct knowledge of the underlying NIC hardware or Ethernet framing; it operates purely at the byte stream level, consuming bytes one at a time from the network layer's internal header buffer before handing the larger data transfer over to direct DMA-style writes into the destination huge page.

System Integration & Initialization

The loader has no initialization function of its own. Its three functions are called directly and sequentially by `kernel_main()` within the POLLING case of the state machine, immediately after `reset_header()` has cleared the network layer's internal state. The strict call order is `unlock()` first, then `poll_payload_size()`, then `poll_payload()`. Calling them out of order would result in malformed reads from the byte stream, as each function consumes a fixed and expected number of bytes from the header buffer.

The protocol implemented by this module is a precise mirror of the transmission sequence in `send_data_hardware.py`. The sender transmits a single header frame containing a 4-byte big-endian magic number (0x474F2121, the ASCII string "GO!!") followed immediately by the 8-byte little-endian payload size. All subsequent frames carry raw payload bytes. The kernel side uses `unlock()` to validate the magic number, `poll_payload_size()` to read the size, and `poll_payload()` to receive the raw data — the two sides are tightly coupled and any mismatch in byte order or framing will cause a silent protocol failure.

Application Example

The following example shows how the three loader functions are called within the POLLING case of the kernel state machine.

```
#include "headers/loader.h"
#include "headers/network.h"

case POLLING:
    // Clear network layer header state from the previous transaction
    reset_header();
```

```

// Step 1: Block until the magic number "GO!!" is received
unlock();

// Step 2: Read the 8-byte little-endian payload size
uint64_t payload_byte_count = poll_payload_size();

// Step 3: Allocate huge pages and block until all bytes arrive
uint8_t* payload_mem = vmm_alloc_huge_page(num_pages, VMM_PRESENT |
VMM_WRITEABLE);
poll_payload(payload_mem, payload_byte_count);

state = EXECUTING;
break;

```

Direct API References:

Macros

- **MAGIC_NUMBER:** Defined as 0x474F2121, the ASCII encoding of “GO!!”. This is the 4-byte big-endian handshake value that the sender must transmit as the first four bytes of the header frame. `unlock()` blocks indefinitely until this exact sequence has been assembled from the incoming byte stream. This value must match identically between the kernel and the Python sender script.

Functions

- **uint32_t unlock()**
 - Performs the magic number handshake. It calls `read_ethernet()` in a tight loop, accumulating bytes into a 32-bit integer by shifting left 8 bits and ORing in each new byte. The loop continues until the accumulated value equals **MAGIC_NUMBER** (0x474F2121). Because the magic number is transmitted big-endian and assembled by shifting, the first byte received becomes the most significant byte of the result. This function will block indefinitely if the sender never transmits the correct sequence, making it the gating condition for the entire POLLING cycle.
 - Returns: The validated magic number (0x474F2121) upon a successful handshake.
- **uint64_t poll_payload_size()**
 - Reads the next 8 bytes from the network layer byte stream and assembles them into a 64-bit unsigned integer representing the total byte size of the incoming payload. The bytes are read least-significant-first (little-endian): each byte is cast to `uint64_t`, shifted left by `i*8` bits, and ORed into the result. This matches the `struct.pack("<Q", ...)` format used by the Python sender. The returned value is used by `kernel_main()` to calculate the number of 2MB huge pages to allocate before calling `poll_payload()`.

- Returns: The 64-bit little-endian payload size in bytes.
- `void poll_payload(uint8_t* payload_mem_addr, uint64_t payload_byte_size)`
 - Orchestrates the bulk data transfer phase of the POLLING cycle. It calls `network_set_dest()` to register the allocated huge page and expected byte count with the network layer, then enters a blocking hlt loop, sleeping the CPU between NIC interrupts until `network_bytes_received()` reports that all expected bytes have arrived. Because the network layer writes incoming data frame payloads directly into the destination buffer as interrupts fire, this function has no copying or looping of its own; it serves purely as the synchronization barrier that prevents the state machine from advancing to EXECUTING before the payload is fully resident in memory.
 - Parameters:
 - `payload_mem_addr`: A pointer to the VMM-allocated destination buffer where the incoming payload will be written. Must be large enough to hold `payload_byte_size` bytes.
 - `payload_byte_size`: The exact number of bytes to wait for, as returned by `poll_payload_size()`. The function returns as soon as this threshold is reached.