

Project Plan: MATH UNIKERNEL

High-Performance 64-bit Bare-Metal Execution Environment

Project Plan: MATH UNIKERNEL	1
1. Project Vision	3
2. Technical Goals	3
3. Team Roles & Ownership	4
4. Development Timeline (Jan – April)	5
Phase I: The "Zero-to-Hello" Sprint (Weeks 1–3)	5
Phase II: The Memory Foundation (Weeks 4–7)	5
Phase III: The Safety & Acceleration Sprint (Weeks 8–11)	6
Phase IV: Optimization & Final Delivery (Weeks 12–15)	7
5. Tooling & Environment	8
6. Risk Management	8
7. System Diagrams	9
8. Resources and Documentations	11
9. Future Work	12

1. Project Vision

MATH UNIKERNEL is a specialized, 64-bit bare-metal operating system designed to eliminate "System Noise" (the jitter caused by background tasks, interrupts, and schedulers in general-purpose OSs). By creating a deterministic, single-address-space environment, we provide 100% of CPU cycles to hardware-accelerated mathematical workloads.

2. Technical Goals

- **Boot Protocol:** Utilize the Limine Boot Protocol to enter 64-bit Long Mode.
 - *Explanation:* We use Limine to skip the complex "Assembly Purgatory" of BIOS/UEFI handoffs. It handles the transition to 64-bit mode and provides a memory map, allowing us to focus on the kernel logic immediately.
- **Memory Architecture:** Implement a 4-level paging system (PML4, PDPT, PD, PT) with Identity Mapping.
 - *Explanation:* This establishes the "Floor Plan" of the system. Identity mapping ensures that virtual addresses match physical ones, simplifying debugging while allowing us to use hardware-level memory protection.
- **Stability:** Build an Interrupt Descriptor Table (IDT) to handle hardware exceptions without Triple Faults.
 - *Explanation:* The IDT is our "Safety Net." Without it, any minor CPU error (like dividing by zero) causes an instant hardware reset. This table tells the CPU exactly which function to run when an error occurs.
- **Acceleration:** Activate Advanced Vector Extension and Streaming SIMD Extension (AVX/SSE) vector units and implement SIMD-accelerated math kernels.
 - *Explanation:* Modern CPUs have "hidden" math engines (Single Instruction, Multiple Data) that are disabled by default to save power. We will manually toggle these on to perform calculations on multiple data points simultaneously.
- ~~**I/O:** Headless execution with raw data streaming via the Serial Port (UART).~~
 - ~~*Explanation:* Since we have no GUI or window manager, we communicate with the outside world via the Serial Port. This ensures that I/O overhead is minimal and predictable.~~

3. Team Roles & Ownership

- **Systems Architect (Core & Stability):** Responsible for Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT), and Exception Handling.
 - *Explanation:* This role ensures the "Foundation" is solid. They write the code that keeps the CPU running and manages the transitions between different privilege levels.
- **Memory Surveyor (Paging & Allocation):** Responsible for the 4-level Paging system and Physical Frame Allocator.
 - *Explanation:* This role owns the "Real Estate." They decide how the 16GB of RAM is divided up and write the logic that translates code addresses into physical hardware locations.
- **Math Engineer (Acceleration & I/O):** Responsible for AVX activation, SIMD implementation, and the Serial Driver.
 - *Explanation:* This role owns the "Engine." They focus on the actual performance of the calculations and ensure that the results are streamed out of the system efficiently.

4. Development Timeline (Jan – April)

Phase I: The "Zero-to-Hello" Sprint (Weeks 1–3)

Goal: Prove the bootloader-to-kernel handoff.

- **Week 1: Setup x86_64-elf-gcc cross-compiler environment and Makefile.**
 - *Explanation:* We must build a compiler that doesn't assume a standard library (like libc) exists. This ensures our binary is "freestanding" and ready for bare metal.
- **Week 2: Implement the `_start` assembly trampoline and Limine request headers.**
 - *Explanation:* This is the very first code the CPU runs. It sets up the Stack Pointer (RSP) so we can finally jump from Assembly into high-level C code.
- **Week 3: Write the basic UART/Serial driver.**
 - *Explanation:* This is our "Heartbeat." By writing characters to port 0x3F8, we gain the ability to use `serial_printf` for debugging everything that follows.
- **Milestone 1: "Hello World" printed to the serial console in QEMU.**
 - *Explanation:* This proves the entire toolchain works. If we can print "Hello," we have successfully hijacked the hardware.

Phase II: The Memory Foundation (Weeks 4–7)

Goal: Establish the 16GB memory floor plan.

- **Week 4: Parse Limine memory maps and build the Physical Frame Allocator.**
 - *Explanation:* We need to know which parts of our 16GB are actual RAM and which are reserved for hardware. The allocator tracks "Free" vs "Used" 4KB chunks.
- **Week 5: Code the 4-level paging structures (PML4 to PT).**
 - *Explanation:* We build the actual arrays in memory that the CPU's Memory Management Unit (MMU) will walk to find data.
- **Week 6: Implement Identity Mapping for the kernel and data regions.**
 - *Explanation:* We map the first few gigabytes so that Address X in code is Address X in RAM, preventing the kernel from "crashing into a wall" when paging is turned on.

- **Week 7: Enable paging by loading the CR3 register.**
 - *Explanation:* Moving the PML4 address into the CR3 register is the "Big Bang" moment where the CPU starts using our custom memory map.
- **Milestone 2: Kernel successfully executes code within a virtual address space.**
 - *Explanation:* This confirms our memory math is correct. We now have a structured environment to load massive datasets.

Phase III: The Safety & Acceleration Sprint (Weeks 8–11)

Goal: Hardware stability and SIMD activation.

- **Week 8: Build the Interrupt Descriptor Table (IDT).**
 - *Explanation:* We create a 256-entry "Phone Book" that tells the CPU where to go when hardware events occur.
- **Week 9: Write ISRs (Interrupt Service Routines) for Page Faults and Divide-by-Zero.**
 - *Explanation:* These functions catch errors. Instead of the laptop rebooting, it will now tell us "Hey, you tried to access unmapped memory at this address."
- **Week 10: Toggle CR4/CRO bits to enable SSE and AVX units.**
 - *Explanation:* We flip physical bits in the CPU's control registers to power up the vector math hardware.
- **Week 11: Implement the first SIMD-accelerated math function (C Intrinsics).**
 - *Explanation:* We write our first actual "Workload"—math that processes 4 or 8 numbers at a time using one single CPU instruction.
- **Milestone 3: CPU catches exceptions and runs vector instructions without crashing.**
 - *Explanation:* This proves the system is "Professional Grade." It can handle errors gracefully and perform high-speed math.

Phase IV: Optimization & Final Delivery (Weeks 12–15)

Goal: Benchmarking and Laptop deployment.

- **Week 12: Implement Huge Page (2MB) support at the Page Directory (PD) level.**
 - *Explanation:* By using larger pages, we reduce the number of "map lookups" the CPU has to do, further decreasing system noise and latency.
- **Week 13: Limine Framebuffer integration (trivial task)**
 - *Explanation:* Deprecate logging to stdout (primarily used in QEMU). Conduct all logging on the screen using Limine's provided framebuffer.
- **Week 13: Ethernet I/O drivers**
 - *Explanation:* Deprecate serial printers to CPU ports (i.e. COM1) in favor of Ethernet connection.
 - Scan the PCI bus to locate the Network Interface Card (NIC) and map its registers into the kernel's address space.
 - Configure Direct Memory Access (DMA) "Receive Rings" in RAM. This allows the NIC to stream the workload directly into physical memory without CPU intervention, preserving deterministic timing.
- **Week 14: Conduct "Unikernel vs. Linux" performance benchmarks.**
 - *Explanation:* This is the "Science" phase. We run the same math on Linux and our OS to prove that ours is faster and more deterministic.
- **Week 15: Final presentation and Bare-Metal laptop demonstration.**
 - *Explanation:* The final test. We boot the kernel on a real physical laptop from a USB drive to show it's a real Operating System.
- **Milestone 4: Final Project Delivery (April).**
 - *Explanation:* Successful completion and handover of all code and performance data.

5. Tooling & Environment

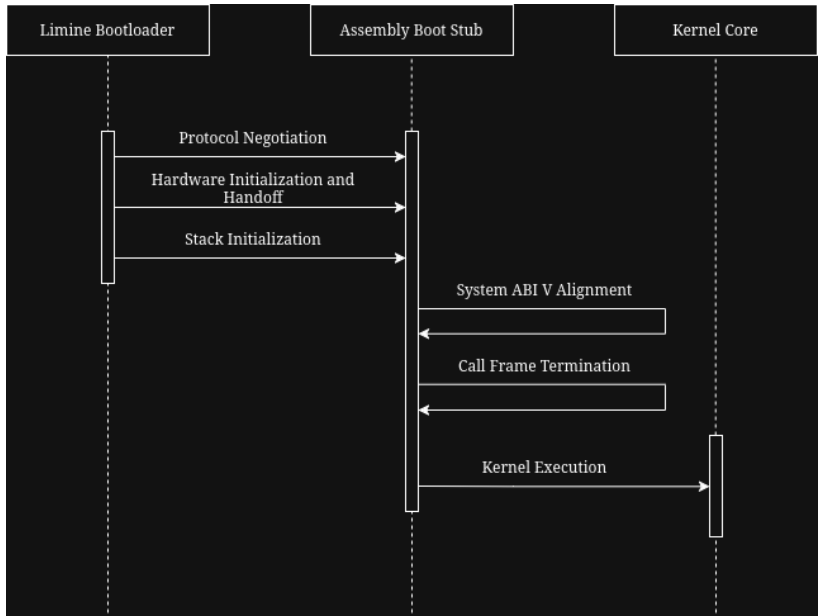
- **Compiler:** x86_64-elf-gcc (Freestanding).
- **Emulator:** QEMU (with -d int,cpu for register-level debugging).
- **Bootloader:** Limine (binary and config files).
- **Debugger:** GDB connected via QEMU's GDB stub.

6. Risk Management

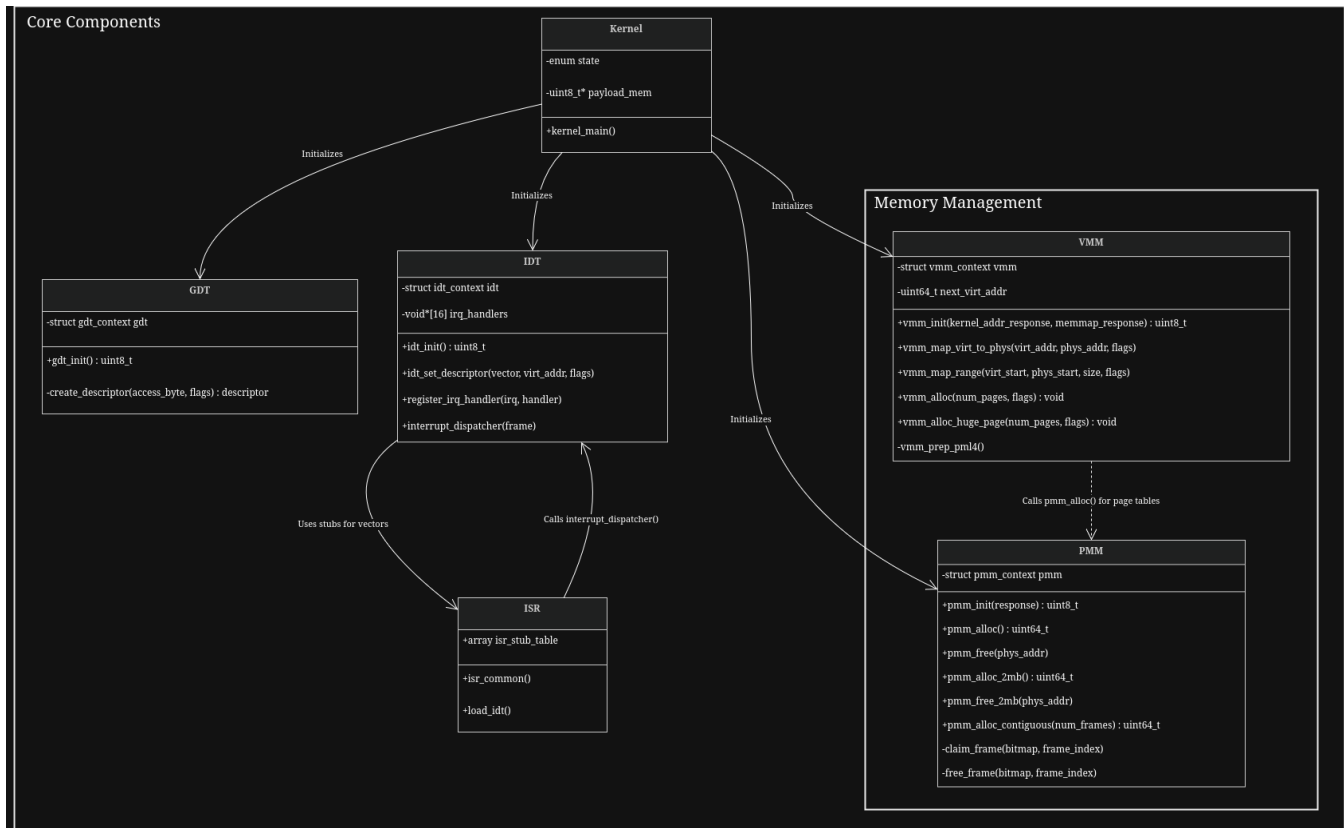
- **The Triple Fault:** Mitigated by implementing the IDT early in Phase III.
- **Headless Blindness:** Mitigated by prioritizing the Serial Driver in Phase I.
- **Hardware Variance:** Mitigated by strictly adhering to the Limine Memory Map and avoiding hard-coded physical addresses.

7. System Diagrams

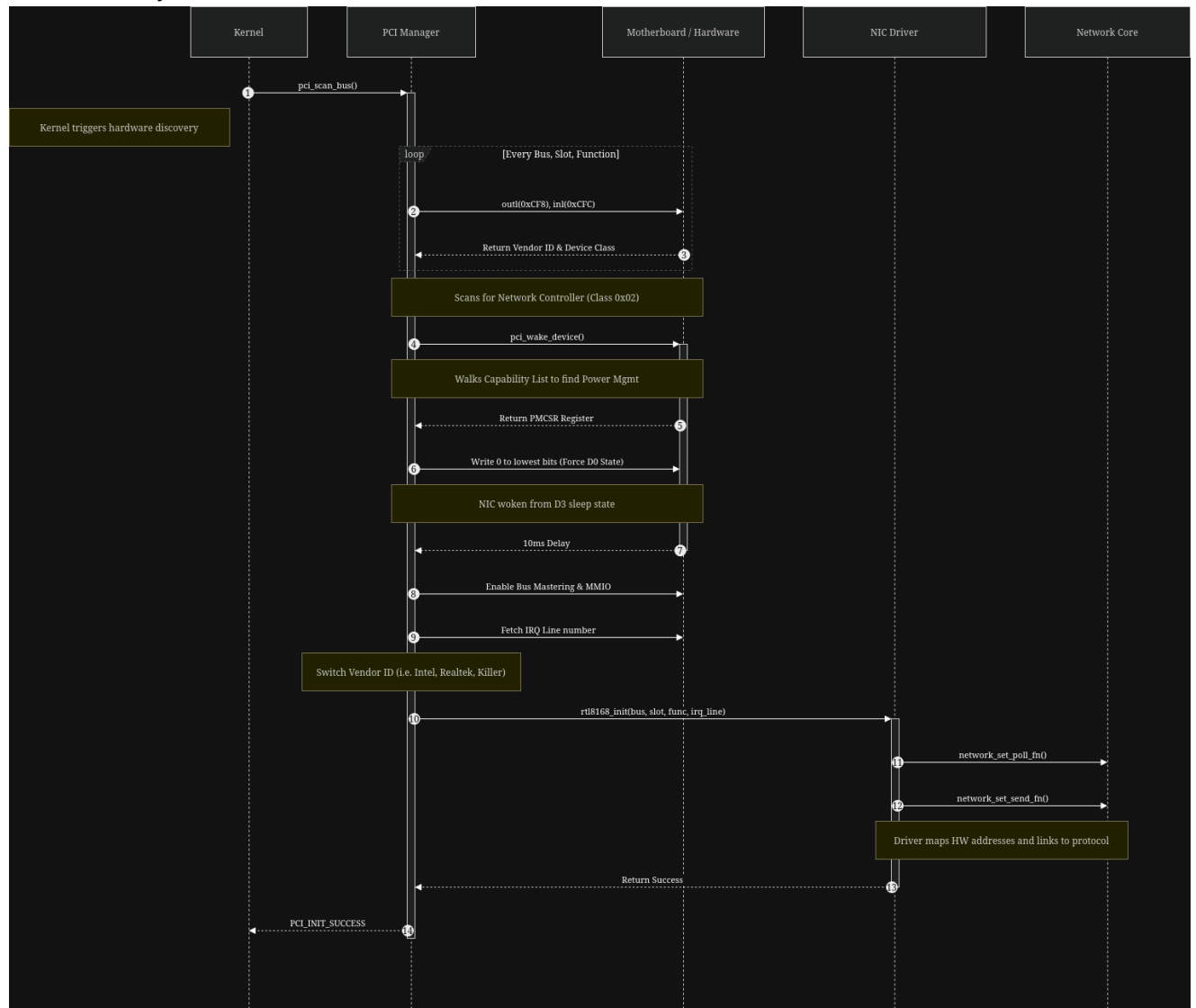
Boot Sequence



Core Components



NIC discovery and driver initialization



Network TX and RX



8. Resources and Documentations

- [OSDev Wiki](#) will be the diving board to start off the project. It is not necessary to read *all* of it. The most useful pages will be the ones about the focus points off the project
- [4-Level Paging](#)
- [Limine Boot Protocol](#)

9. Future Work

- **TX completion on the HP Victus** — the RTL8168 driver is partially written and the network architecture is in place. Resolving the MSI interrupt setup would complete the full RX/TX pipeline and enable real-time result streaming back to the sender, making the system genuinely bidirectional.
- **Multi-core workload dispatch** — bring up additional CPU cores via the APIC and allow workloads to be distributed across them. This would make the Amdahl's Law argument much stronger and enable parallel GEMM tiles to run on separate cores simultaneously.
- **Expanded math library** — add FFT, convolution, and softmax to the kernel API. These are the remaining primitive operations underlying neural network inference, and adding them would make Math Unikernel a more complete bare metal ML inference substrate.
- **Hypervisor guest mode** — add VMX support so Math Unikernel can run as a Type 1 hypervisor guest. This would allow it to coexist with a host OS on the same machine, making deployment more practical without sacrificing the bare metal performance model.
- **RISC-V port** — port the kernel to a RISC-V target such as the SiFive Unmatched or a QEMU RISC-V machine. The architecture is clean enough to support a second ISA, and RISC-V is increasingly relevant in edge compute and ML accelerator hardware.