ChatGPT

Veridian OS Development Plan

1. Initial Hardware and Software Stack

- **Target architecture:** Begin with x86_64 (Intel/AMD desktop & server). Plan for future embedded support (ARM, RISC-V, etc.) – many Rust OS projects already target these (e.g. Redox supports x86, x86_64, ARM64, RISC-V [1]).
- **Host environment:** Use a mainstream Linux development machine (Ubuntu/Fedora) or WSL on Windows. Install Rust toolchains (stable and nightly). Add target triples (e.g. `rustup target add x86_64-unknown-none`) and tools like `cargo-xbuild` or use Cargo's `build-std` feature [2] . Ensure LLVM tools (`rustup component add llvm-tools-preview`). Use Rust-aware IDEs (VSCode with rust-analyzer) and debugging tools (gdb/LLDB for Rust).
- **Build tools:** Use Cargo for build management. Employ OS-specific tools like `bootimage` (to create bootable disk images) and Rust-based bootloaders. For example, the Rust `bootloader` crate provides a custom x86_64 BIOS/UEFI bootloader [3] [4] . The `bootimage` tool automates building and running kernels under QEMU [2] [5] . Containerize builds or use cross-compilation as needed.
- **Bootloader options:** Support both legacy BIOS (SeaBIOS) and UEFI. Options include using GRUB (mature but C-based) or pure-Rust bootloader crates. The Rust OSDev "bootloader" crate is an experimental BIOS/UEFI bootloader (in Rust/ASM) for 64-bit kernels [3] . After kernel linking, use a bootable image with either GRUB or this Rust bootloader.
- **Emulation/testing platforms:** Use QEMU for rapid iteration: QEMU supports x86_64 emulation and can attach disk images, network, and devices. Tools like `cargo bootimage run` can launch QEMU automatically [5] . For multi-boot testing or hardware access, use VirtualBox/VMware with a custom disk image. For lower-level debugging, Bochs or hardware debuggers can be used.
- **Essential libraries and crates:** Include architecture crates (e.g. `x86_64` for paging, CPU registers), memory allocators (`linked_list_allocator` or `buddy_system_allocator` [6]), and hardware interface crates (`acpi` , `pci` parsers). Use `core` / `alloc` (no_std) initially. For I/O, use crates for drivers or write minimal drivers (keyboard, VGA text mode). Rust's strict borrow-checker eliminates many bugs: Redox notes that Rust "almost entirely eliminates... memory leaks, buffer overruns, use after free" – the root of most OS security bugs [7] . Build a minimal C runtime (like Redox's `relibc` in Rust [8]) for userland interoperability.

Phase 1: Microkernel and Core Services

- **Key features:**
- **Rust microkernel:** Kernel only handles CPU setup, context switching, scheduling, low-level interrupt handling and inter-process communication (IPC). All other services (drivers, filesystems) run in user space. This follows microkernel principles (move drivers to userland) for security and modularity [9] [10] .
- **Process and thread management:** Basic kernel tasks: create/destroy processes, threading, user/kernel mode transitions. Simple scheduler (e.g. round-robin or priority) to start.

- **IPC mechanisms:** Provide messaging channels or ports for user processes. For example, a capabilty-based or message-passing model (inspired by seL4/L4). Existing Rust projects like LakeOS use capability-based IPC (kernel objects tracked as capabilities) [11] .
- **Simple filesystem:** Implement a minimal block-based filesystem (e.g. FAT or RedoxFS). Initially, a read-only ramfs or FAT12 to boot the OS, then read-write support. Use Rust crates if available: the `fatfs` crate provides a FAT library [12] and an `ext2` crate exists for ext2/3 support [13] .
- **Essential drivers:** Basic drivers in user space (if microkernel): a console display driver, a timer interrupt driver, and a block storage driver (e.g. AHCI or VirtIO). Kernel provides raw device access via I/O ports and memory-mapped I/O, with safe Rust abstractions where possible.
- **Implementation considerations:** Follow Redox's model of user-space drivers [14] [9] . Use existing Rust OS crates for help: e.g., the `x86_64` crate for page tables, `interrupt` macros for IRQ handling, and `spin` or `parking_lot` for locks. For memory, allocate pages via a frame allocator and use a crate like `linked_list_allocator` [6] for heap. Avoid `unsafe` code except where absolutely needed (low-level hardware init). Hardware spec crates (ACPI, SMBIOS, PCI) are available to parse tables. Keep the kernel minimal: as Redox notes, microkernels have "small code size" and fewer bugs [10] .
- **Development activities:** Start with a "Hello, world" kernel (print to screen) and build up: set up GDT/IDT, enable paging, handle timer IRQs. Develop the scheduler and context-switch in Rust. Then add IPC primitives (e.g., message queues or channels). Progressively move drivers into user processes. Keep feature branches small; continually integrate. Follow Rust best practices (no panics in kernel, consistent style) and use Git for version control. Document kernel interfaces and coding guidelines (as Redox has a "Best practices" guide [9]).
- **Challenges and best practices:** Common pitfalls include interrupt safety (requires `unsafe` code), stack overflows, and boot failures. Use unit tests in host environment wherever possible (e.g. test a Rust module logic with `#[cfg(test)]`). Write comprehensive documentation for IPC APIs. Emphasize code reviews and coding standards to maintain safety (as Redox recommends, "Rusting properly" to avoid panics [15]).
- **Testing strategies:** *Unit tests* for pure-Rust modules (e.g. data structures) using Rust's `#[test]`. For integration testing, build disk images and boot them in QEMU frequently. Redox advises to "rebuild and test boot often" (e.g. `make qemu` each change) to catch regressions [15] . Tools like `bootimage` support running tests under QEMU automatically [16] . Logging or simple test suites can run under QEMU (redirect serial console output). Eventually, consider formal methods: e.g. formally verify critical parts with Rust verification tools (Miri, Kani), or adopt a proven microkernel spec (like seL4's methods) for reference.
- **Potential enhancements:** Design a **modular driver framework** so new drivers (e.g. graphics, USB) can be loaded without changing the kernel. Early on, consider an asynchronous I/O model (e.g. use Rust's `async/await` for driver requests) to experiment with async in kernel/user processes. This paves the way for high-throughput I/O later.

Phase 2: User Space and Basic Utilities

- **Key features:** Build out a minimal userland. Provide a command-line interface (shell) and core utilities (ls, cat, etc.). Implement basic memory management (e.g. malloc for processes, using heap allocators and virtual memory). Develop a networking stack (TCP/IP, sockets). Support common file formats (FAT, ext).
- **System calls and user-space design:** Define a minimal syscall ABI for user/kernel interactions (trap-based interrupts or message passing). Provide syscalls for file I/O, networking, process control, and

IPC. Either follow POSIX semantics (for familiarity) or a custom design (Redox is mostly POSIX-like). Consider implementing a WASM sandbox layer (like Theseus) for running untrusted code securely. Design userland as modular programs launched by the shell; each utility can be a separate binary (Rust crate).

- **Utilities development:** Leverage existing Rust ecosystem: e.g. the **uutils/coreutils** project is a cross-platform Rust rewrite of GNU core utilities. Ubuntu is even planning to use uutils as the system coreutils [17]. Adopting or porting uutils can jumpstart text utilities. Write a simple shell (e.g. based on rushell or similar). For text editing, one could port a small editor (like a Rust `vi` clone) or use existing Rust crates. All user programs should link against a Rust-based C library (`relibc` in Redox [14]) that provides basic runtime (printf, malloc).
- **Networking stack:** Integrate a Rust TCP/IP stack such as **smoltcp** [18] (a no-heap, event-driven stack). Begin with IPv4, DHCP, ARP, and simple protocols (UDP, TCP). Create user-space networking daemons/servers. Use virtio-net drivers in QEMU initially.
- **File format support:** Incorporate libraries for filesystems and image formats. The `fatfs` crate provides FAT support [12]. The `ext2` crate can parse ext2/3 filesystems [13]. For executable formats, support a.out or ELF so user programs can be loaded (e.g. use the ELF loader logic from [bootloader-api]).
- **Development workflows:** Manage user utilities as separate Rust cargo projects. Establish a build pipeline so that building the OS also compiles these crates. Use continuous integration to build and test each utility (unit tests and integration tests in QEMU). Encourage modularity: e.g. a common "std" in userland while the kernel is `no_std`. Publish documentation on syscalls and conventions.
- **Testing:** Write tests for CLI utilities using Rust's test harness and tools like insta for diff testing. For networking, simulate loopback traffic in QEMU or use network emulation. Use fuzzing (cargo-fuzz) on protocols. For performance, benchmark user applications (see Phase 5).
- **Optional features:** Early GUI elements (e.g. a pixel framebuffer driver, basic graphics library). Provide scripting support (embed a language like Lua or WASM for quick apps). Even a basic windowing protocol (ASCII or pixel-based) can be prototyped.

Phase 3: Security and Privilege Separation

- **Key features:** Implement fine-grained access control. Add privileges/sandboxing so processes only access allowed resources (e.g. file permission bits, IPC capabilities, network access flags). Support disk encryption (e.g. encrypt at block level), secure boot (UEFI Secure Boot or a Rust-based bootloader with signature checks [19]), and auditing/logging of security events. Provide a secure random number generator and standard crypto functions for apps. Build in integrity checks for binaries (e.g. signatures).
- **Rust security practices:** Leverage Rust's safety: e.g. all crypto should use vetted RustCrypto libraries (AES, SHA, XChaCha20-Poly1305, etc.). Avoid `unsafe` unless necessary, and minimize its scope. Use code analysis tools: Clippy, Miri, and static analyzers (MirChecker) to catch undefined behavior. Enforce no-panics in critical paths. Use tools like `cargo-audit` to detect vulnerable dependencies. For secure boot, the SentinelBoot project (Rust for RISC-V) shows how to limit memory-unsafety: it "minimize[s] memory safety vulnerabilities" in its threat model [19].
- **Security architecture:** Enforce least privilege everywhere. Consider a **capability-based model** for resources (file handles, IPC ports) similar to seL4 or LakeOS [11]. For example, represent rights as unforgeable tokens. Implement process isolation via paging. Use supervisor mode to restrict direct hardware access (everything else is a protected driver process). For kernel modules or drivers, apply

secure coding and code signing. Enable UEFI Secure Boot or a cryptographic bootloader to verify the kernel before execution.

- **Testing:** Perform penetration testing on the OS. This includes fuzzing syscalls and interfaces (e.g. malformed IPC messages). Use static analysis on Rust code for common pitfalls. Use dynamic analysis like AddressSanitizer (with nightly LLVM) on userland programs. Audit the code manually and via community review. For sandboxing, test by trying to escape from jailed processes. Consider formal methods for key components (e.g. formally verify a simple syscall implementation).
- **Advanced security:** As enhancements, explore Trusted Execution support: e.g. run critical code in Intel SGX enclaves or ARM TrustZone (Rust has SGX frameworks like Fortanix [20], and research like RusTEE [21] for TrustZone). Evaluate adding fine-grained capabilities for devices (like GenodeOS style). For network security, integrate TLS libraries (e.g. Rustls) into userland. Implement a minimal firewall or packet filter.

Phase 4: Package Management and Software Ecosystem

- **Key features:** A secure package manager (e.g. `pkg`) and central repository of applications. Tools to *build/port* third-party software. Possibly support foreign binary formats (Linux/BSD compatibility layers).
- **Package design:** Design a package format with strong integrity guarantees. For example, Redox's `pkgar` format is a signed, atomic archive: it uses encryption and hashing to ensure packages are only installed if verified [22]. Follow similar goals: atomic upgrades, delta updates (hash-based), and relocatable installs. Use cryptographic signatures (e.g. NaCl, BLAKE3) to verify packages before install [22]. Encourage static linking (Rust's default) so that libraries are bundled; Redox notes static linking "improves security" by isolating each program's code [23].
- **Repository and tools:** Set up a central package repository (with public keys for signing). Provide a CLI tool (like `pkg`) to install/update packages. Automate building and testing packages on release (CI builds packages for each commit). Provide templates and build scripts for popular software (e.g. a `Cargo.toml` for Rust apps, a port of a Python app, etc.). Consider interoperability with existing ecosystems (e.g. allow installing Linux software via container or chroot).
- **Ecosystem practices:** Foster community contributions: write clear documentation and examples for creating packages. Use an open contribution model (public Git repo, issue tracker). Host forums/mailing lists for support. Follow a meritocratic governance (similar to Linux Foundation projects) with clear roles (maintainers, committers). Use an open-source license that encourages adoption (e.g. MIT/Apache-2.0 dual license for core, see below). Provide tutorials (e.g. "How to port your app") and code of conduct.
- **Testing:** Automate package building: run every package build in a sandbox (QEMU or container), run test suites. Use tools like `cargo test` for Rust crates, and equivalent for other languages. For compatibility, validate by running userland compatibility tests (e.g. try common Linux commands, benchmark programs). For the package manager itself, fuzz its input (e.g. malformed package indexes).
- **Enhancements:** Over time, add an "app store" front-end, containerized apps (like Flatpak), or compatibility layers (e.g. a Wine-like or Linux ABI). Create developer incentives (bug bounties, recognition programs) to enrich the software ecosystem.

Phase 5: Performance Optimization and Hardware Support

- **Key features:** Scale to multi-core and advanced hardware. Optimize kernel scheduler, memory management (caching, huge pages), and I/O paths. Provide broad driver coverage (network cards, GPUs via open specs, NVMe, USB, etc.).
- **Scheduler and concurrency:** Design a scalable SMP scheduler (e.g. per-core run queues, load balancing). For real-time needs, consider a configurable priority scheduler or deadline-based scheduler. Use Rust concurrency primitives to avoid locks when possible (e.g. lock-free queues). Enable NUMA awareness if supporting many cores.
- **Zero-copy and driver architecture:** Use DMA and memory-mapped I/O to minimize copies (e.g. network receive queues mapping user pages). In a microkernel, design user-space driver APIs to permit zero-copy message passing (pass buffer references instead of copying data). For broad driver support, consider leveraging existing driver frameworks or ports (e.g. a Rust port of Linux's VirtIO drivers for virtualization).
- **Profiling and tuning:** Profile the OS to find bottlenecks. For example, Redox provides a kernel profiling system that samples instructions and generates flamegraphs [24]. Developers can build a `profiling` kernel and use flamegraph tools (like Inferno) to visualize hotspots [24]. This guides optimizations (e.g. hot-path in scheduler or IPC). Use benchmarking tools: simple disk/RAM tests with `dd` (as Redox does [25]) to measure throughput, or run SPEC-like suites. Integrate profiling tools (perf, eBPF tracers) once virtualization permits, or use QEMU's built-in profiling.
  Figure: Example kernel flamegraph showing hot code paths in the scheduler (in red) [24].
- **Benchmarking and compatibility:** Regularly run microbenchmarks (context-switch latency, syscall overhead, filesystem I/O). Test stability under load (heavy networking, many processes). Validate compatibility by running real workloads (e.g. compiling a large Rust project). Compare against baseline systems. Use these results to iteratively optimize (e.g. reduce lock contention, improve cache usage).
- **Advanced optimizations:** For real-time or low-latency tasks, consider a tickless kernel mode or priority inheritance. Enable hardware acceleration: use SIMD (Rust's `std::simd`) for math-heavy tasks, leverage GPU via APIs (OpenCL/Vulkan through crates like `wgpu` for user-space apps, or writing a Mesa driver in Rust). Explore preemptive scheduling and fine-grained timers. If targeting cloud/HPC, support CPU isolation and hardware virtualization features (VT-x, SR-IOV).

Phase 6: GUI and Windowing System

- **Key features:** Develop a full graphical desktop environment. Provide a display server (like X11 or Wayland equivalent), window manager, and compositing. Standard desktop services (file manager, terminal, panel, etc.) all in Rust. Support modern toolkits.
- **Display protocols and toolkit:** Evaluate using **Wayland** protocol (current Linux standard) with Rust libraries (e.g. Smithay, a toolkit for writing Wayland compositors). Alternatively, design a custom protocol (Redox's Orbital uses its own "orbital:" scheme). For toolkits, consider integrating or porting existing Rust GUI frameworks: e.g. Druid, iced, or Slint. Also support common GUI libraries via FFI (GTK-RS, Qt via CXX/FFI).
- **Compositor and window manager:** Implement a compositor that handles input/output: GPU acceleration for rendering windows. For example, Redox's **Orbital** is a Rust display server that manages windows and the `orbital:` URI scheme entirely in user space [26]. Launch Orbital early and run GUI apps as clients. The window manager should support basic features (move/resize windows, double buffering).

- **Desktop integration:** Build or port standard desktop components: terminal emulator, text editor, calculator, file browser, etc. These run as regular user processes. Integrate internationalization and theming support (use a resource system or stylesheet). Implement at least basic accessibility (keyboard focus, font scaling).
- **Testing:** Automate GUI tests: measure rendering latency and framerate. Use tools to simulate user input (mouse/keyboard macros) and check UI state. Test driver compatibility by running on different GPUs/VMs (QEMU with virtio-gpu, or pass-through real GPU). Do user feedback testing for UX. Ensure responsiveness under load (e.g. simultaneous animations).
- **Enhancements:** Add window theming and customization. Provide accessibility options (high-contrast mode, screen-reader hooks via Orbi, etc.). Leverage Rust's safety by using GUI frameworks that support memory safety (e.g. avoid unsafe OpenGL bindings). Eventually support graphics-rich apps (video playback, web browsers).

3. Best Practices for OS Development

- **Agile, iterative development:** Adopt an incremental workflow (sprints or milestones). Break the project into phases (as above) and review after each phase. Commit early and often; maintain continuous integration builds. Use source control with code reviews. Treat OS development much like any critical systems project: write tests, document interfaces, and adjust plans based on feedback. Agile values (working code over docs) apply: focus on delivering a minimal working OS frequently, then iterating [27] .
- **Open-source community:** Host the code on a public platform (e.g. GitHub/GitLab). Use collaborative tools: a dedicated forum or Zulip chat for developers, an issue tracker for bugs/feature requests, and a pull-request/contribution workflow. Encourage community involvement by maintaining clear contribution guidelines and a code of conduct. Governance can be meritocratic (like many FOSS projects) – trusted maintainers merge PRs, steering committee sets direction. Documentation (wiki, tutorials) and responsive support forums help grow the user/developer base. Participate in conferences or publish blog updates (e.g. "This Month in Rust OSDev" [2]) to attract contributors.
- **Licensing considerations:** Choose a permissive license to encourage adoption and contributions. Most Rust projects use MIT or Apache-2.0 (often dual-licensed). For example, Redox is MIT-licensed [8] , and many crates use Apache/MIT so downstream users face no onerous restrictions. Apache 2.0 adds a patent grant (attractive for companies). GPLv3 is stricter (copyleft) and may deter some users; if compatibility with Linux/BSD is desired, GPL or LGPL could be chosen for parts. Clearly document licensing in all repositories. Permissive licenses give downstream users maximal freedom (they "can use, copy, modify" without much restriction [28]), whereas GPL ensures changes stay open. Weigh community goals: a BSD/MIT license usually builds a wider ecosystem, but GPL can protect against proprietary forks.

Sources: Veridian's design draws on existing Rust OS projects and tools. For example, Redox OS is a Unix-like microkernel in Rust [8] [9] ; the "bootloader" and "bootimage" crates demonstrate Rust-based bootloaders and test runners [3] [16] ; and Rust libraries like `fatfs` and `smoltcp` offer filesystem and networking support [12] [18] . Best practices are informed by Redox's documentation on testing and packaging [15] [29] and by research systems like LakeOS [11] (capability microkernel) and SentinelBoot [19] (secure boot in Rust). All cited references are integrated above in context.

---

[1] GitHub - flosse/rust-os-comparison: A comparison of operating systems written in Rust
https://github.com/flosse/rust-os-comparison

[2] [4] This Month in Rust OSDev (July 2020) | Rust OSDev
https://rust-osdev.com/this-month/2020-07/

[3] GitHub - rust-osdev/bootloader: An experimental pure-Rust x86 bootloader
https://github.com/rust-osdev/bootloader

[5] [16] GitHub - rust-osdev/bootimage: Tool to create bootable disk images from a Rust OS kernel.
https://github.com/rust-osdev/bootimage

[6] Allocator Designs | Writing an OS in Rust
https://os.phil-opp.com/allocator-designs/

[7] [9] Why a New OS? - The Redox Operating System
https://doc.redox-os.org/book/why-a-new-os.html

[8] [14] Redox - Your Next(Gen) OS - Redox - Your Next(Gen) OS
https://www.redox-os.org/

[10] Microkernels - The Redox Operating System
https://doc.redox-os.org/book/microkernels.html

[11] GitHub - vincenthouyi/LakeOS: A L4-like micro-kernel OS written in Rust
https://github.com/vincenthouyi/LakeOS

[12] fatfs - Rust
https://docs.rs/fatfs/latest/fatfs/

[13] ext2 - Rust
https://docs.rs/ext2

[15] Testing Practices - The Redox Operating System
https://doc.redox-os.org/book/testing-practices.html

[17] Ubuntu 25.10 plans to swap GNU coreutils for Rust • The Register
https://www.theregister.com/2025/03/19/ubuntu_2510_rust/

[18] GitHub - smoltcp-rs/smoltcp: a smol tcp/ip stack
https://github.com/smoltcp-rs/smoltcp

[19] Developing a cryptographically secure bootloader for RISC-V in Rust
https://www.codethink.co.uk/articles/2024/secure_bootloader/

[20] Intel SGX - Fortanix
https://www.fortanix.com/intel-sgx

[21] RusTEE: Developing Memory-Safe ARM TrustZone Applications
https://dl.acm.org/doi/10.1145/3427228.3427262

[22] [23] [29] Package Management - The Redox Operating System
https://doc.redox-os.org/book/package-management.html

[24] [25] Performance - The Redox Operating System
https://doc.redox-os.org/book/performance.html

26 GUI - The Redox Operating System
https://doc.redox-os.org/book/gui.html

27 Agile software development - Wikipedia
https://en.wikipedia.org/wiki/Agile_software_development

28 MIT License - Wikipedia
https://en.wikipedia.org/wiki/MIT_License