ZeroOS: A Universal Modular Library OS for zkVMs

Guangxian Zou[1], Isaac Zhang[1], Ryan Zarick[1], Kelvin Wong[1], Thomas Kim[1], Daniel L.-K. Wong[1], Saeid Yazdinejad[1], and Dan Boneh[2]

[1]*LayerZero Labs*
[2]*Stanford University*

Abstract

zkVMs promise general-purpose verifiable computation through ISA-level compatibility with modern programs and toolchains. However, compatibility extends further than just the ISA; modern programs often cannot run or even compile without an operating system and `libc`. zkVMs attempt to address this by maintaining forks of language-specific runtimes and statically linking them into applications to create self-contained unikernels, but this ad-hoc approach leads to version hell and burdens verifiable applications (vApps) with an unnecessarily large trusted computing base. We solve this problem with ZeroOS, a modular library operating system (libOS) for vApp unikernels; vApp developers can use off-the-shelf toolchains to compile and link only the exact subset of the Linux ABI their vApp needs. Any zkVM team can easily leverage the ZeroOS ecosystem by writing a ZeroOS bootloader for their platform, resulting in a reduced maintainence burden and unifying the entire zkVM ecosystem with consolidated development and audit resources. ZeroOS is free and open-sourced at https://github.com/LayerZero-Labs/ZeroOS.

1 Introduction

Zero-knowledge Virtual Machines (zkVMs) cryptographically prove the correctness of execution of an application (called a *guest*) against a given input with regard to the application semantics. Guest proving on zkVM is commonly paired with commitment and verification of the proof on a byzantine-fault tolerant ledger to form a *vApp* [14].

While zkVMs promise to prove any application compiled for an ISA (e.g., RISC-V), reality falls short: ISA-level compatibility does not imply software compatibility. Current zkVM execution environments are much closer to a bare-metal, no-OS environment that is incompatible with modern applications that rely on a rich package ecosystem.

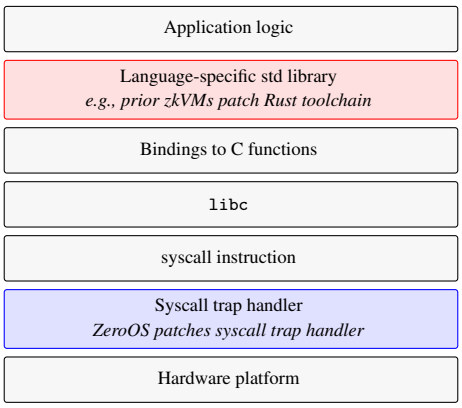


Figure 1: **ZeroOS versus status quo.** Prior zkVM toolchains patch each language's standard library (red), whereas ZeroOS patches the syscall trap handler (blue), which is a generic and language-agnostic solution.

To bridge this gap, developers currently maintain forked, platform and language-specific toolchains to statically compile guests into self-contained *unikernels* [6]. This approach requires each zkVM team to maintain a separate toolchain for each supported language; as a result, vApp developers must accept the risk of bugs in an unnecessarily large *trusted computing base* [8] of modified standard library code, and each zkVM team is burdened with costly audits for their platform-specific forked toolchains.

We solve this problem with ZeroOS, a universal modular library OS (libOS) for zkVM guest unikernels. ZeroOS shifts the integration point from scattered language-specific forked runtimes to a language-agnostic syscall shim that implements a simplified subset of the Linux syscall ABI. Developers can use off-the-shelf toolchains to link unmodified applications against ZeroOS, effectively eliminating "version hell" and unlocking universal compatibility between all applications

and all zkVMs. To illustrate the difference: custom toolchains directly compile `malloc` calls to jump to a custom `malloc` handler, whereas ZeroOS traps the syscall instruction and dispatches it to the `sys_malloc` handler according to the ISA semantics (e.g., `ecall`, `SVC`, `syscall`).

Our architecture is a Pareto improvement over current monolithic or bare-metal approaches. vApp developers can build their application on top of POSIX [13] interfaces, and existing applications can be recompiled to vApps without modification. Each subsystem in ZeroOS (e.g., memory allocation, I/O) is modularized, allowing developers to choose the exact set of features they need to efficiently run their vApp. Additionally, a single subsystem can have multiple implementations in different modules, so applications can choose between highly efficient barebones implementations and more feature-rich implementations by just swapping the module imports. This "pay-for-what-you-use" model has two major benefits: (1) it gives developers full control over their exposure to software bugs in ZeroOS, and (2) it allows developers to minimize execution trace length by unplugging unused logic (e.g., I/O).

In practice, ZeroOS improves the security of ***all*** vApps and ***all*** zkVMs by consolidating ecosystem resources on a single shared codebase. ZeroOS allows the industry to pool resources by standardizing a platform-agnostic OS layer: a single audit, security patch, or performance optimization to ZeroOS benefits ***all*** zkVMs and vApps simultaneously. This replaces the fragmented landscape of ad-hoc patched toolchains with a unified, robust foundation for the entire growing ecosystem of over twenty zkVM projects [1].

2 Background

2.1 ISA, operating system (OS), and standard library

Modern software stacks are layered. At the bottom, the instruction set architecture (ISA) defines the semantics of a set of instructions (e.g., RISC-V) that make up the interface between hardware and software. Above that, an operating system (e.g., Linux) manages and exposes hardware resources to userspace applications through a system call (syscall) interface. The system C library (`libc`) provides the fundamental userspace runtime and the bindings to these syscalls. Language-specific standard libraries (e.g., Rust `std`) then build higher-level APIs for files, threads, and networking on top.

Importantly, there is a distinction between the POSIX-like syscall API exposed by libc to user applications and the OS-level ABI called by libc. For example, `mmap` is a POSIX syscall that internally emits a syscall instruction (e.g., `ECALL`) to invoke the OS-level `sys_mmap`. However, `mmap` does not always invoke `sys_mmap`–for efficiency, small allocations are often managed by libc entirely in userspace after initially provisioning a larger block of memory from the OS.

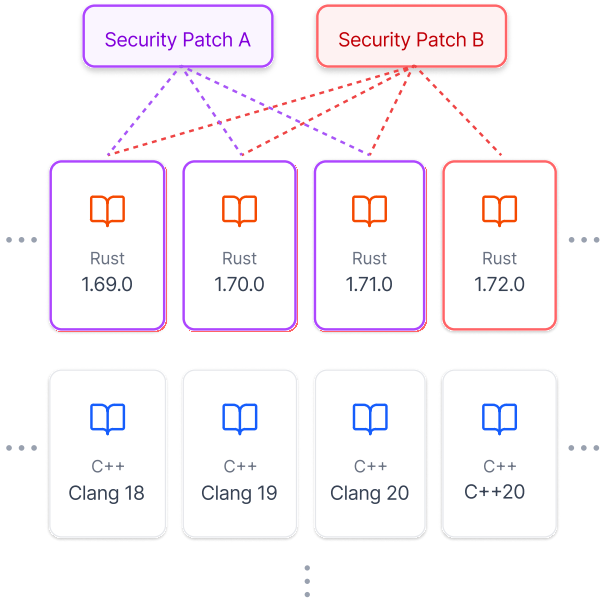


Figure 2: **Version hell.** Using a forked toolchain, each zkVM may need to maintain many different versions across multiple languages. Security patches also need to be rapidly backported to older versions to support customers who have not yet upgraded, creating a near-intractable long-term operational burden.

2.2 zkVM determinism and traces

In this paper, we use the term *guest* for all logic that runs atop the zkVM: this includes the *application* (high-level business logic), its language runtime, the C standard library, the ZeroOS bootloader, and the ZeroOS kernel.

A *prover* runs the guest in a zkVM, which executes the guest bytecode against some ISA specification and produces a succinct proof that this execution is correct. During execution, the prover records a complete trace of guest execution steps, including register updates, memory accesses, and I/O. The prover uses this trace as the witness to the zkVM's proof system, and a verifier checks the resulting proof without re-running the guest. Therefore, the prover cost is tied to the length of the guest trace, and any non-deterministic influence from the host—such as time or randomness—must pass through an explicit interface and be modeled in the statement being proved. As a result, current zkVM designs require guests to be fully deterministic, with a well-specified boundary between guest and host.

A challenge of developing zkVM guests (including ZeroOS) is that all guest logic must be fully *constrained*–a malicious prover must not be capable of influencing the guest trace in a way that violates any guest-defined safety invariants. This requirement affects ZeroOS in several subtle ways, precluding features with stateful side effects, and requiring inputs such as entropy and time to be deterministically constrained.

2.3 Library OS and Unikernels

A library operating system (libOS) [2] implements operating-system functionality as a set of libraries linked into the application's address space, rather than as a separate monolithic kernel. Unikernel [6] systems such as Unikraft [3] statically link an application against a libOS into a standalone binary that runs directly on a hypervisor or bare metal. Unikernels allow developers to select only the OS components needed by their application to improve predictability and efficiency.

Well-designed operating systems, including libOS and unikernels, are agnostic to guest application language and platform. Linkage against the operating system interface is done after compilation from the high level language into machine code; guest applications written in any language that can be compiled to machine code can be linked against ZeroOS, which is a unique advantage of ZeroOS over forked toolchains. While it is possible to make a platform-agnostic forked toolchain, in reality most forked toolchains contain platform-specific optimizations; ZeroOS is compiled alongside the application code using an off-the-shelf toolchain, making it inherently platform-agnostic.

2.4 Custom toolchains

Having set out the execution model and the libOS/unikernel patterns, we describe how zkVMs are typically engineered today. Rather than handling normally emitted syscall instructions at the syscall shim layer, existing zkVM stacks modify language-specific toolchains and standard libraries to replace syscall instructions with a direct invocation of the syscall logic itself. We refer to these modifications collectively as *custom toolchains*.

Ecosystem Fragmentation The reliance on custom toolchains creates two fundamental problems for the zkVM ecosystem: operational fragmentation and security fragility. First, this approach creates *language-specific development silos*. Each zkVM team is forced to maintain a separate fork of the toolchain and standard library for every language they support. This results in "Version Hell" (Figure 2), where zkVM toolchains perpetually suffer from *upstream drift*, lagging behind official releases. Developers are frequently blocked from using modern language features or security fixes because the zkVM-specific fork has not yet forward-ported the necessary patches from the upstream compiler.

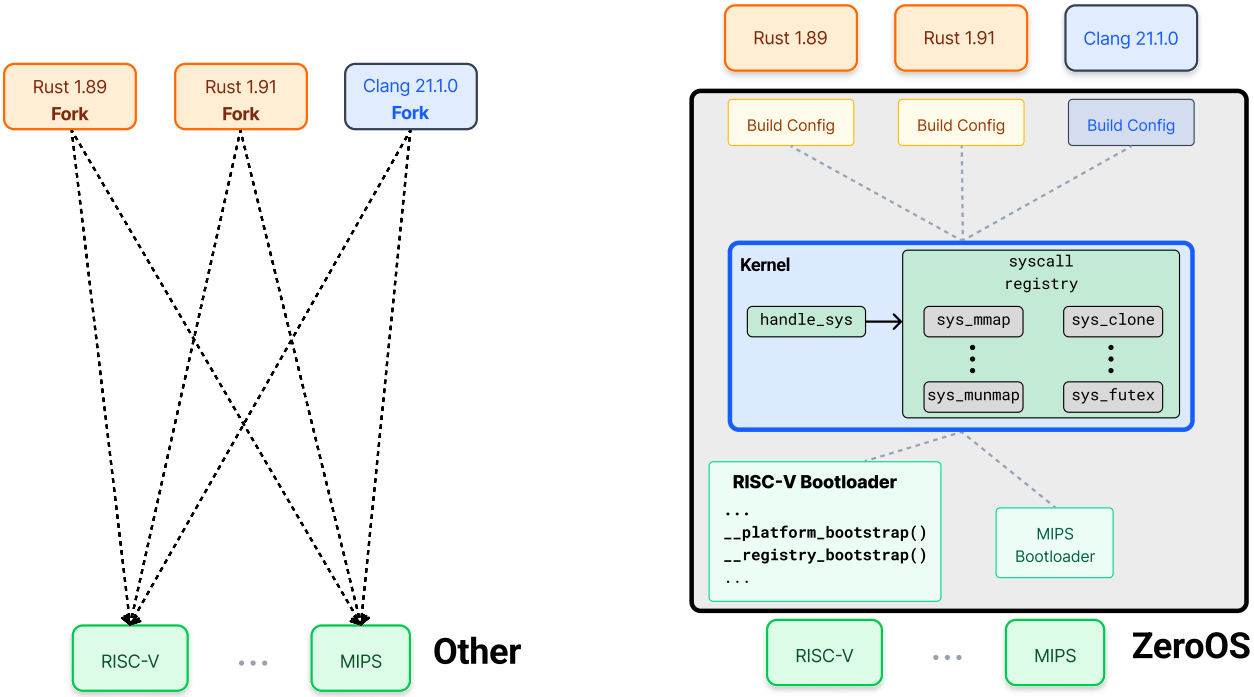
Expanded Trusted Computing Base: Modifying the standard library fragments the security landscape. Every patch applied to a language runtime moves logic from the verified, community-audited upstream codebase into a custom, unaudited fork. A bug in a custom memory allocator patch or a modified thread shim can introduce vulnerabilities that are unique to that specific zkVM implementation. By relying on ad-hoc patches, the industry currently scatters its security resources across dozens of divergent forks.

The slow security response worsens the situation: When a security advisory is published for Rust, developers cannot use the fix until it is ported to the necessary patched fork(s), significantly widening the vulnerability window for vApps. We believe the more secure and scalable approach is to *minimize* the trusted computing base and push all modifications lower in the stack (into the kernel) where interfaces are simplest and most stable.

3 Design

ZeroOS is a library OS that enables compilation of traditional applications into verifiable applications (vApps) [14] packaged as unikernels. Unlike existing approaches which rely on custom language-specific toolchains, ZeroOS performs this transparently at the syscall shim layer *without modifying libc*. As a result, the ZeroOS kernel is language and platform agnostic, and can be compiled using an unmodified off-the-shelf toolchain (note that some toolchains may have to be rebuilt for the target platform (e.g., `riscv64ima-unknown-linux-musl`) without modification to their source code). ZeroOS-specific changes to the application are limited to build scripts within each toolchain (e.g., `musl-toolchain.sh`) rather than the toolchain itself. This design minimizes the trusted computing base and eliminates toolchain fragmentation; additionally, the majority of nontrivial logic is isolated to the highly modular kernel, which enables rapid response to security vulnerabilities. Figure 4 illustrates how developers using ZeroOS can link their zkVM-unaware application logic to ZeroOS to transform off-the-shelf applications into vApps.

Figure 3b illustrates the low-level architecture of ZeroOS with three main components: a small ***build config*** for each toolchain to statically link the application against ZeroOS, a ***kernel*** that defines the syscall trap handler and syscall handlers, and a ***bootloader*** which performs all necessary architecture-specific bootstrap-



(a) **Status quo:** many patched forks of language-specific toolchains and standard libraries compile directly to machine code.

(b) **ZeroOS:** modular kernel compatible with all toolchains and all platforms via build configuration and platform-specific bootloader respectively.

Figure 3: ZeroOS minimizes platform-specific and language-specific code, improving security and simplifying maintenance.

ping steps. This design contrasts with the currently ubiquitous approach of simply maintaining a separate zkVM-specific fork of each toolchain, which as we described in Section 2.4, is impractical due to the operational burden of maintaining many versions across many languages.

3.1 Build Config

The build config acts as the *universal compatibility* bridge between the *unmodified* language-specific toolchain (e.g., preprocessor, compiler, linker) and ZeroOS kernel. ZeroOS allows developers to simply configure their existing toolchain to statically link their application against the ZeroOS bootloader and kernel to build a unikernel. By isolating ZeroOS-specific changes to build scripts rather than modifying the compiler source code, we eliminate "version hell" and ensure that applications remain portable and standard-compliant. Note that applications explicitly written for bare-metal (e.g., Rust `no_std`) do not need ZeroOS or any ZeroOS-specific build config.

3.2 Bootloader

The bootloader handles three main responsibilities: (1) initialize platform components (e.g., CPU, memory, devices), (2) load and initialize the system (kernel, libc), then (3) transfer control to the application. Any zkVM platform can easily support ZeroOS by implementing `__platform_bootstrap` for their specific platform (Figure 4). Many zkVMs implement custom inlines and syscall instruction handling, which makes it impractical to include generic or default implementations of `__platform_bootstrap` in ZeroOS.

However, this design constraint isolates all platform-specific logic to this single function; all other components of ZeroOS and the ZeroOS bootloader are platform-agnostic. This minimal integration surface area allows the ecosystem to consolidate development and security auditing resources on the shared, generic ZeroOS kernel, rather than fracturing effort across completely custom runtimes.

3.2.1 I/O

ZeroOS supports peripherals, such as block devices and human interface devices using memory-mapped I/O

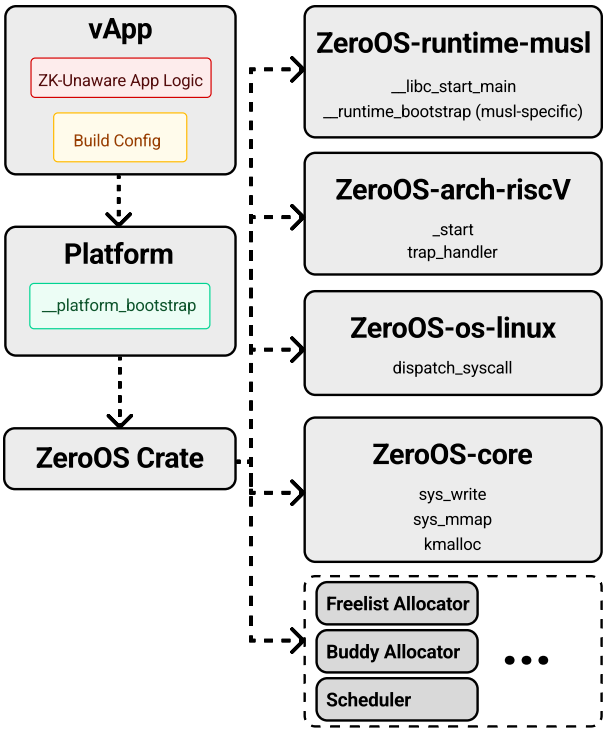


Figure 4: The ZeroOS package is a collection of packages, including libc (ZeroOS-runtime-musl), bootloader (ZeroOS-arch-riscV), syscall ABI (ZeroOS-os-linux), syscall wrappers (ZeroOS-core), and various implementations of OS primitives (e.g., freelist allocator) in separate modular sub-packages. Each zkVM platform must implement `__platform_bootstrap` which is called by bootloader `_start`. vApps automatically inherit this entire stack when they import the platform (zkVM) package into their project.

(MMIO). On top of this low-level I/O layer, it is fairly straightforward to write device drivers for (logical) devices exposed by the zkVM platform.

For brevity, we only provide an overview of how MMIO is implemented in ZeroOS–port-mapped I/O is very similar, just using port-mapped I/O instructions rather than manipulating memory.

ZeroOS supports MMIO through the interface exposed by the zkVM platform - on a real hardware platform this would be a combination of the MMU and I/O device drivers, but on a zkVM platform this extends to more ad-hoc logical interfaces (Host-Target Interface, HTIF). Physical devices typically also require hardware interrupts (device to host) to handle asynchrony between the CPU and the device controller, but this is not strictly necessary in zkVM. Optionally, syscalls such as `ioctl` are not part of the initial release of ZeroOS, but can be added as a module to facilitate device management by the application.

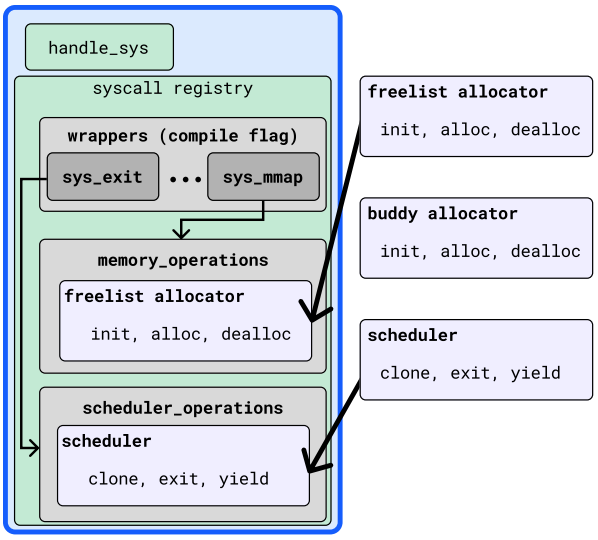


Figure 5: The ZeroOS kernel includes the trap handler (`handle_sys`), wrapper interfaces for each syscall, and a set of imported syscall ops. The syscall ops primitives concretely implement all of the necessary low-level operations invoked by the abstract wrappers.

The platform-specific `__platform_bootstrap` function bootstraps all devices in the zkVM platform; this function is necessarily different for each zkVM, and is the only zkVM-specific code that each zkVM project must maintain to support ZeroOS. Once the devices are bootstrapped (control registers and/or buffers mapped into memory), they can be manipulated by device drivers or syscall handlers through the memory subsystem.

Note that for more complex devices, the zkVM must implement (deterministic) hardware interrupt handling between the device and the CPU. Both the device and CPU are logical components of the zkVM platform, so while this would be complicated to implement it is not impossible in principle.

3.3 Kernel

The ZeroOS kernel design is illustrated in Figure 5: it includes the trap handler (`handle_sys`), abstract implementations of each syscall (`wrappers`) that call into a set of primitives stored in the operation registries (`memory_operations` and `scheduler_operations`). The wrappers can be enabled and disabled via compile flags, so vApp developers can minimize the bytecode size of their ZeroOS unikernels.

Syscall wrappers define the high level logic of each syscall, calling into the operations registries to use low level primitives. One example of a wrapper is `sys_mmap`, which will call `memory_operations.alloc`; the vApp developer can freely change the imported kernel mem-

ory allocator implementation (e.g., freelist vs buddy) to match their precise needs. Wrappers are configured at compile time, and the operation registries are populated by the bootloader during system initialization.

Different applications may require different feature sets; for example, an application which does not use I/O may choose to exclude I/O-related syscalls to minimize compiled code size. Compared to monolithic kernels, this modular design has superior security and efficiency in the context of vApps: it allows developers to "unplug" unused modules to strictly minimize the trusted computing base and execution trace length, achieving the efficiency of bare-metal optimization with the compatibility of a full OS.

Implementing features at the kernel level as opposed to the standard library level significantly reduces the chances of breaking changes. For example, even in 2001 when Linux added 64-bit support to `mmap`, a 64-bit `mmap2` was added on top of the existing 32-bit `mmap` support with no breaking change to the ABI [5]; this shows that kernel ABIs remain stable for decades, making this layer of the stack practical and desirable for integration. This stability drives *ecosystem consolidation*: instead of fragmenting security resources across version-specific toolchain patches, the industry can consolidate auditing efforts on a universal kernel where one fix immediately benefits all applications and zkVMs.

4 Implementation

Current vApps are primarily limited by prover cost, which is a direct function of the trace length. This has led to the majority of zkVM and vApp developers simplifying their unikernels to minimize trace length. As such, our initial release of ZeroOS is built with this goal in mind–ZeroOS supports *single-process*, unified memory space unikernels with no simultaneous parallelism. This initial implementation is highly modular, and all modules can easily be swapped out for more feature-rich modules as they are contributed to the codebase. In Section 5, we discuss how additional features such as parallelism, virtual memory, and virtual filesystem (VFS) could be implemented in ZeroOS for future zkVMs.

To strike a good balance between minimizing the performance footprint of ZeroOS while still maintaining a familiar and full-featured interface for developers, we chose to target `musl` [7] (unmodified) by implementing the Linux ABI/syscall interface in ZeroOS. The implementation is highly modular and extensible, and supports OS + runtime composition (e.g., `linux-musl`, `linux-gnu`, `bsd`), and serves as an open platform for the community to contribute additional OS ABIs and runtimes.

The end-to-end flow from boot to allocating memory in `sys_mmap` is outlined in Figure 6. In this section, we go into detail about the implementation of each step.

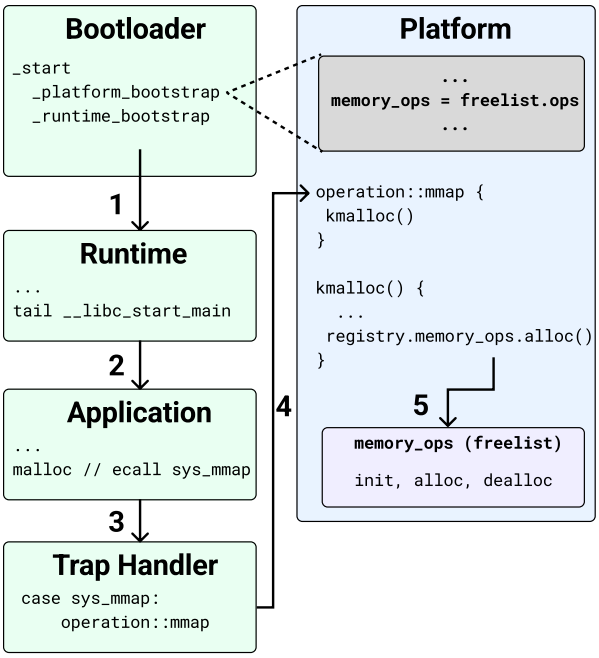


Figure 6: End-to-end execution flow to bootstrap a guest unikernel and call `sys_mmap`. First, the bootloader bootstraps the platform, populating the platform `memory_ops` with the freelist implementation. In call (1), the bootloader bootstraps and invokes the libc runtime (`__libc_start_main`), which performs some bookkeeping before handing execution to the application in call (2). The application executes a syscall instruction (`ecall`) which traps into the ZeroOS trap handler in call (3). The trap handler calls the platform wrappers in call (4), which eventually calls into the `memory_ops` alloc function in call (5).

4.1 Bootloader

The bootloader implements the following flow in ZeroOS: `_start` → `__platform_bootstrap` → `__runtime_bootstrap` → `__libc_musl_main` → application `main`. The full boot sequence is included in Appendix A.

Step 0: Hardware Reset - the zkVM is assumed to be in an untainted state at initialization

Step 1: Kernel `_start` sets up the global pointer and stack, then jumps to `__bootstrap` (implemented in Rust). This step is composed of two subcalls, first to `__platform_bootstrap` which initializes any I/O de-

vices and syscall shim (in RISC-V, by writing the address of the trap handler into the mtvec register).

Note that the syscall shim initialization is specific to the ISA of the zkVM, which often diverges from the RISC-V specification. For example, a RISC-V zkVM may use a fixed address for the syscall trap handler rather than storing the address in `mtvec`; in this case, `__platform_bootstrap` may be mostly empty. The source code corresponding to step 1 is included in Appendix B.

Step 2: The second subcall of `__bootstrap` is `__runtime_bootstrap` (Appendix C), which builds the `musl` stack (Appendix D), runs pre-init/constructor hooks to prepare the stack for the application, then calls into `__libc_start_main`.

Step 3: `musl __libc_start_main` invokes the application `main` and handles termination/cleanup.

Step 4: Application `main` runs the developer's high-level application logic.

4.2 Kernel

4.2.1 Memory Management

Memory model The initial release of ZeroOS implements a single unified address space with no virtual memory or paging. This unified address space is shared between the ZeroOS kernel and a single application (i.e., no process isolation).

Memory management model ZeroOS uses a standard two-layer memory management hierarchy, with a *userspace* allocator that handles guest memory allocations and a *kernel* allocator that helps the userspace allocator expand or shrink its memory pools.

The userspace allocator is implemented in libc and the Rust standard library. Libc includes `malloc`, `free`, and `realloc`, and the Rust standard library implements `__rust_alloc` and `__rust_dealloc`.

The kernel allocator implements `mmap` and `munmap` on pages (e.g., 4 KiB) of global physical memory (i.e., the region of memory described by `__heap_start` and `__heap_end`).

The top section of Table 1 describes the various memory-related syscalls used by libc for memory management, along with a discussion of which syscalls are required vs optional.

When designing the ZeroOS kernel heap allocator, we evaluated several designs with respect to the key concerns of zkVM: auditability, determinism, predictability, feature-sufficiency, and runtime complexity. In particular, `sys_munmap` support is required for compatibility with most modern implementations of `libc`, as allocations larger than `DEFAULT_MMAP_THRESHOLD_MIN` (128 KiB in `musl`) bypass the user-space allocator entirely and use `sys_mmap` and `sys_munmap` directly.

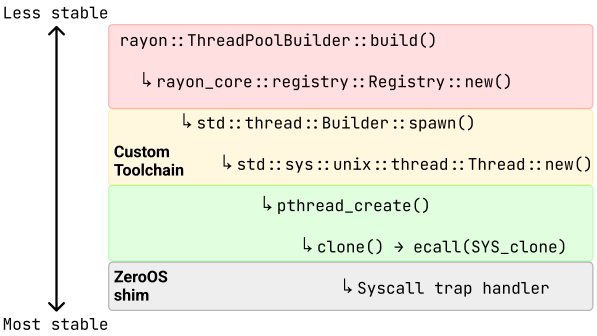


Figure 7: ZeroOS is at the bottom of the callstack, right above the bare-metal, ensuring the highest stability to minimize "version hell". The current status quo of using custom forked Rust `std` incurs higher risk of breaking changes between versions.

We choose the coalescing free list (e.g., `linked_list_allocator::LockedHeap`) for the first ZeroOS kernel allocator, as it strikes a good balance with our key concerns (Table 2). Alternatively, the bump allocator is simpler with no fragmentation, but does not support reclamation. We believe a buddy allocator is a viable alternative if future workloads demand stricter bounds on fragmentation, but the additional code and runtime complexity makes this a long-term target.

Note that applications will always be more efficient if they manage memory in userspace, so the expectation is that skilled developers will minimize calls to `sys_mmap` to reduce trace length and consequently prover cost.

4.2.2 Thread management

Modern threading libraries (e.g., rayon [9]) are typically built on threading primitives exposed by `libc` which in turn build on syscalls such as `sys_futex`. In ZeroOS, we fully implement five main threading-related syscalls intended to enable basic multithreading: `sys_clone`, `sys_futex`, `sys_sched_yield`, `sys_exit`, and `sys_exit_group`. These five syscalls provide the bare minimum functionality to implement a non-preemptive multithreading environment for a single-application unikernel. Additionally, we stub three thread management syscalls (`sys_set_tid_address`, `sys_rt_sigaction`, and `sys_rt_sigprocmask`) to allow threading libraries to compile properly. Other thread management syscalls such as `gettid`, `sched_{set,get}*`, and multitasking syscalls such as

Syscall	Description	Implementation
	Memory management	
sys_mmap	Kernel heap allocator locates a free range and returns the base address to the caller.	Yes
sys_munmap	Returns a previously mmap'ed memory region to the kernel.	Yes
sys_brk	Adjusts the program break of the caller; superseded by mmap.	Stub (-ENOMEM)
sys_mremap	Resizes an existing mmap'ed region.	Stub (-ENOSYS)
sys_mprotect	Changes mmap region protection flags; a no-op in our single-tenant unikernel.	Stub (no-op)
	Thread management	
sys_clone	Fundamental primitive for creating new threads; allocates a TCB and sets up the thread for execution.	Yes
sys_futex (FUTEX_WAIT)	Checks the futex word, records the thread as blocked on the futex, and yields to the next thread.	Yes
sys_futex (FUTEX_WAKE)	Moves all threads blocked on a given futex to the ready queue and returns the number woken.	Yes
sys_sched_yield	Allows a thread to yield its time slice, pushing the caller onto the end of the ready queue.	Yes
sys_exit	Terminates the calling thread, deallocates its TCB, and removes it from the ready queue.	Yes
sys_exit_group	Shuts down the process group; in a ZeroOS unikernel this terminates the whole system.	Yes
sys_set_tid_address	Sets the Thread ID (TID) pointer for the calling thread.	Stub (no-op)
sys_rt_sigaction	Change the handler for a given signal. ZeroOS does not support signals, so this syscall is stubbed.	Stub (no-op)
sys_rt_sigprocmask	Manipulate the signal mask for a given process. ZeroOS does not support signals, so this syscall is stubbed.	Stub (no-op)
	I/O	
sys_write	Writes to a file descriptor. In ZeroOS, the implementation is dependent on the devices initialized during bootstrapping.	stdout / stderr
sys_fstat	Return information about a given file descriptor. ZeroOS supports fstat for all platform devices.	stdout / stderr
sys_writev	Write multiple buffers.	stdout / stderr
sys_prlimit64	Get or set resource limits. In our single-tenant system, resource limits are superfluous.	Stub (no-op)
sys_getrandom	Return a random value. Real entropy requires platform support, ZeroOS currently returns a non-random value.	Return dummy value
sys_clock_gettime	Return current system time. Real time requires platform support, ZeroOS defaults to returning the zero time.	Return zero time
sys_ioctl	Manage I/O devices. ZeroOS does not currently support specialized I/O devices, only memory-mapped stdout and stderr via platform-defined interfaces.	Stub (-ENOTTY)

Table 1: ZeroOS syscall interface.

Allocator	Deallocation	Simplicity	Fragmentation	Small objects	Runtime complexity
Bump allocator	No	Simplest	N/A	Poor	Best
Slab allocator	Yes	Excellent	Minimal	Excellent	Excellent
`kmalloc`-style (SLAB/SLUB)	Yes	Complex	Minimal	Excellent	Excellent
Coalescing free list	Yes	Excellent	Vulnerable to adversarial workloads	Moderate	Good
Buddy allocator	Yes	Complex	Reasonable	Good	Good

Table 2: **Kernel allocator design space.** Simpler allocators are easier to audit but may suffer from fragmentation or limited feature support; ZeroOS uses a coalescing free list as a balanced choice.

Field name	Description
`saved_regs`	Thread's program counter, stack pointer, and general-purpose registers.
`thread_state`	Thread's state enum: `Ready`, `Blocked`, or `Exited`.
`tid`	Thread ID (returned by `sys_clone` and `gettid`).

Table 3: **ZeroOS TCB fields.** Minimal per-thread state needed for scheduling and context switching.

`fork` and `exec` may be implemented in future versions of ZeroOS as the need arises. Certain thread management syscalls are not implemented such as `sys_tgkill` as ZeroOS does not use Unix signals for control flow.

The middle section of Table 1 outlines the threading-related syscalls in the ZeroOS design.

Threading and multitasking in a traditional kernel is often quite complex and heavyweight. For example, the thread control block (TCB) (i.e., `task_struct`) tracks memory mappings, file descriptors, user/group identities, signal handlers, scheduling policies, and more; the definition of `task_struct` in the Linux kernel contains over 300 fields [4], most of which are unnecessary for zkVM guests. ZeroOS assumes a single process, a single address space, and no application-level parallelism in its current implementation. All threads share the same memory and file system context, and there is no notion of per-process credentials or per-process address spaces. This allows us to use a radically simplified TCB that contains only what is needed for scheduling and context switching, shown in Table 3.

These five syscalls and minimal TCB enable ZeroOS to support the `std::thread` API and `rayon`'s work-stealing scheduler without compromising on our primary concerns of auditability, determinism, predictability, feature-sufficiency, and runtime complexity. An example of the resulting callstack of an invocation to `rayon::ThreadPoolBuilder::build()` is shown in Figure 7. Custom toolchains often inject their logic in the shallower, less stable interfaces within the callstack, whereas ZeroOS uses unmodified toolchains and code until the bottom of the callstack.

4.3 I/O

ZeroOS supports rudimentary I/O in the form of stdout and stderr via memory-mapped I/O. Both stdout and stderr are initialized during platform bootstrapping, and are permanently mapped to file descriptors 1 and 2 respectively. The bottom section of Table 1 lists the I/O-related syscall interface of ZeroOS; `sys_write`, `sys_fstat` and `sys_writev` are implemented to handle stdout and stderr. Additional syscalls such as `sys_prlimit64`, `sys_getrandom`, `sys_clock_gettime`, and `sys_ioctl` are stubbed to resolve compilation issues, but are not functionally necessary for current vApps.

ZeroOS interfaces with stdout and stderr via the *Host-Target Interface* (HTIF) defined by the platform and initialized during platform bootstrapping. In the current implementation of ZeroOS, we support an HTIF that exposes stdout and stderr as reserved regions of memory where debug output from the program is written. The HTIF defines the interface by which ZeroOS target interacts with the zkVM host, and the responsibility of actually passing this information to the end user (i.e. developer) is outside the scope of ZeroOS.

Note that I/O in the current version of ZeroOS (specifically the HTIF we implement for the RISC-V Spike [10] platform) is intended to be used strictly as a debugging aid, as it is not strictly deterministic and therefore may complicate guest security analysis.

4.4 Build Configuration

The build configuration is responsible for linking the application, libraries, and ZeroOS kernel into a single executable using a specialized build pipeline.

1. Toolchain preparation: Build static `libc.a` using any `musl`-based RISC-V toolchain

2. Target specification: Specify a custom RISC-V target json that links the guest application against `libstd` and `musl` using the Linux ABI.

3. Static linking: The linker resolves symbols so that the Rust standard library calls into `musl`, and `musl` syscall wrappers properly trap into the ZeroOS kernel syscall shim.

This process outputs a unikernel, which is a self-contained ELF that can run on bare-metal in zkVM while retaining the functionality of an application running on Linux.

5 Discussion and Future Work

The ZeroOS implementation presented in this paper establishes the core primitives for a verifiable library OS. While the initial release targets single-process unikernels, the modular architecture effectively crowdsources the operating system's evolution.

Prior work by Tsai et al. [11] has suggested that supporting 100% of applications in a typical Debian distribution requires implementing 272 system calls, but only 81 for the top 10% most popular applications (and 40 for the top 1%). Additionally, prior work by Unikraft has shown that as much as 60% of syscalls can be faked/stubbed without affecting traditional workloads [12].

In this section, we discuss how ZeroOS facilitates the community-driven implementation of these additional features. Once contributed, these modules become instantly available to every platform in the ecosystem.

5.1 Virtual Memory and Paging

Supporting virtual memory in ZeroOS is feasible within the existing shim architecture, although it significantly increases the code and runtime complexity of the kernel. Many syscalls must be implemented or modified, including but not limited to `mmap`, `munmap`, `mlock`, `mprotect`, `mincore`, etc. In particular, many functions such as `mmap` must maintain page tables in a way that is compatible with the ISA-defined MMU, and other small maintenance tasks such as TLB invalidation.

Virtual memory can be implemented in ZeroOS much in the same way it would be implemented in any other operating system, provided the zkVM exposes the necessary platform-level primitives such as the MMU.

5.2 Multitasking and IPC

Multitasking follows the established modular pattern. The ZeroOS design already establishes the foundation with non-preemptive multithreading primitives based on `sys_clone` and `sys_futex`; full multi*tasking* would extend this foundation using `sys_fork`, `sys_wait4`, and `sys_execve`.

Inter-Process Communication (IPC) can be implemented via message queues (`sys_mq_*`, etc.), shared memory (`sys_shm_*`, etc.), pipes (`sys_pipe`, etc.), or network. We discuss how the VFS can be implemented in Section 5.4, which enables IPC via pipes and network interfaces. Message queues and shared memory can be implemented efficiently over a memory buffer that is either managed by the message queue syscall interface or directly exposed as shared memory, preserving the kernel's low footprint.

5.3 Signaling, time, and randomness

Signaling is currently not supported in ZeroOS (e.g., for preemptive scheduling) because current zkVMs do not implement deterministic hardware signals; this is a difficult problem to solve as signals are intended to be non-deterministic, so injecting determinism would constitute a divergence from the ISA. However, assuming a zkVM supports deterministic signaling primitives, ZeroOS can build upon those deterministic signaling primitives and invariants to implement all signal-related syscalls.

Time and randomness face similar constraints. While the syscalls are trivially implementable given deterministic interfaces, the overriding concern is protecting the vApp from a malicious prover. All inputs must be fully constrained to prevent the prover from manipulating the execution trace. As a simplified example, ZeroOS can enforce that time and random numbers are deterministically derived from public inputs, thereby removing any degree of freedom for malicious manipulation.

5.4 Virtual Filesystem and Network

The virtual filesystem (VFS) interface can be implemented by simply maintaining an `fdtable` that corresponds to memory-mapped I/O devices or an in-memory backing storage (e.g., ramdisk).

Given I/O and VFS support, the sockets interface for network support is quite straightforward. Network devices would be initialized as part of the platform, managed via MMIO or PMIO, and exposed via VFS. Crucially, adhering to the Pareto Principle, these heavy subsystems remain strictly optional within the ZeroOS architecture. A vApp that requires networking can link the module, while a pure computation vApp can exclude it

entirely, ensuring that the "pay-for-what-you-use" efficiency is preserved even as the OS grows.

6 Conclusion

ZeroOS is a necessary maturation of the zkVM software stack. We eliminate the “version hell” of custom toolchains and enable universal language and platform compatibility by moving system-level logic from fragile language runtimes into a language-agnostic Library OS. ZeroOS enables compilation of existing applications into vApps using off-the-shelf toolchains with a simple configuration change, and any zkVM can join the ZeroOS ecosystem by implementing a minimal platform bootstrapper.

Adopting the modular unikernel model delivers a Pareto improvement over the status quo. It allows developers to construct guests that possess the high-level compatibility of a full OS while retaining the ability to “unplug” modules to strictly minimize the trusted computing base.

Our current implementation prioritizes single-process unikernels with non-preemptive multithreading to minimize prover cost, and the ZeroOS architecture lays a robust foundation for future expansion. In our future work, we plan to explore complex features such as simultaneous multitasking, virtual memory, and VFS support through ZeroOS modules to support a wider variety of workloads without unnecessarily expanding the trusted computing base for simpler applications. Additionally, we hope that as zkVM platforms support a richer set of I/O-related primitives, we can shift from limited HTIF-defined I/O interfaces to more general support for MMU and device drivers in ZeroOS.

Finally, ZeroOS achieves higher practical security by consolidating ecosystem resources. ZeroOS provides a single platform to consolidate the fragmented landscape of ad-hoc patches with a unified operating system, and shifts the security model from a function of individual team resources to a function of the collective ecosystem. This unification simplifies development, hardens security, and accelerates the adoption of verifiable applications.

References

[1] AWESOME ZKVM AUTHORS. awesome zkVM. https://github.com/rkdud007/awesome-zkvm?tab=readme-ov-file#projects.

[2] ENGLER, D. R., KAASHOEK, M. F., AND O’TOOLE, J. Exokernel: an operating system architecture for application-level resource management. *SIGOPS Oper. Syst. Rev. 29*, 5 (Dec. 1995), 251–266.

[3] KUENZER, S., BĂDOIU, V.-A., LEFEUVRE, H., SANTHANAM, S., JUNG, A., GAIN, G., SOLDANI, C., LUPU, C., TEODORESCU, Ş., RĂDUCANU, C., ET AL. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (2021), pp. 376–394.

[4] LINUX KERNEL COMMUNITY. Linux Kernel `task_struct` Definition. https://github.com/torvalds/linux/blob/7d0a66e4bb9081d75c82ec4957c50034cb0ea449/include/linux/sched.h#L819. Accessed: 2025-12-01.

[5] LINUX MAN-PAGES PROJECT. mmap(2) — Linux manual page. https://man7.org/linux/man-pages/man2/mmap.2.html. Accessed: 2025-12-01.

[6] MADHAVAPEDDY, A., MORTIER, R., ROTSOS, C., SCOTT, D., SINGH, B., GAZAGNAIRE, T., SMITH, S., HAND, S., AND CROWCROFT, J. Unikernels: library operating systems for the cloud. *SIGARCH Comput. Archit. News 41*, 1 (Mar. 2013), 461–472.

[7] MUSL LIBC PROJECT. musl libc. https://musl.libc.org/. Accessed: 2025-12-01.

[8] NIBALDI, G. Specification of a trusted computing base (TCB). Tech. rep., MITRE Corporation, 1979.

[9] RAYON MAINTAINERS. Crate rayon. https://docs.rs/rayon/latest/rayon/index.html. Accessed: 2025-12-01.

[10] RISC-V INTERNATIONAL. Spike RISC-V ISA Simulator. https://github.com/riscv-software-src/riscv-isa-sim. Accessed: 2025-12-01.

[11] TSAI, C.-C., JAIN, B., ABDUL, N. A., AND PORTER, D. E. A study of modern Linux API usage and compatibility: what to support when you’re supporting. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys ’16, Association for Computing Machinery.

[12] UNIKRAFT PROJECT. Compatibility. https://unikraft.org/docs/concepts/compatibility.

[13] WALLI, S. R. The POSIX family of standards. *StandardView 3*, 1 (1995), 11–17.

[14] ZHANG, I., KULKARNI, K., LI, T., WONG, D., KIM, T., GUIBAS, J., ROY, U., PELLEGRINO, B., AND ZARICK, R. vApps: Verifiable Applications at Internet Scale. *arXiv preprint arXiv:2504.14809* (2025).

A ZeroOS boot sequence

```
Hardware reset
    ↪ `_start` (RISC-V, .text.boot)
        ↪ `__bootstrap`
            ↪ `__platform_bootstrap`
                ↪ register trap-handler
            ↪ `__runtime_bootstrap`
                ↪ `build_musl_stack(buffer_top, ehdr_start, PROGRAM_NAME)`
                    ↪ construct stack: argc / argv / envp / auxv
                ↪ `__libc_start_main(main, argc, argv, _init, _fini, NULL)`
                    ↪ `__init_libc(envp, argv[0])`
                        ↪ parse `auxv`, set `libc.page_size`, `__hwcap`, etc.
                    ↪ `__init_tls(aux)`
                    ↪ `__init_ssp((void*)aux[AT_RANDOM])`
                    ↪ `libc_start_main_stage2(main, argc, argv)`
                        ↪ `__libc_start_init()`
                        ↪ `__main_entry(argc, argv)`
                            ↪ user main()
```

B _start

```rust
/// Boot entry point - shared across all libc implementations.
///
/// Sets up GP and SP, then tail-calls __bootstrap for the rest of initialization.
#[unsafe(naked)]
#[link_section = ".text.boot"]
#[no_mangle]
pub unsafe extern "C" fn _start() -> ! {
    naked_asm!(
        // Initialize global pointer first (RISC-V ABI requirement)
        ".weak __global_pointer$",
        ".hidden __global_pointer$",
        ".option push",
        ".option norelax",
        "   lla     gp, __global_pointer$",
        ".option pop",

        // Initialize stack pointer
        ".weak __stack_top",
        ".hidden __stack_top",
        "   lla     sp, __stack_top",
        "   andi    sp, sp, -16",

        // Tail call to bootstrap coordinator
        "   tail    {bootstrap}",

        bootstrap = sym __bootstrap,
    )
}

/// Bootstrap coordinator - orchestrates the initialization sequence.
/// 1. __platform_bootstrap - platform hardware setup
```

```rust
/// 2. __runtime_bootstrap - runtime environment setup (libc stack or no-std)
#[unsafe(naked)]
#[no_mangle]
pub unsafe extern "C" fn __bootstrap() -> ! {
    naked_asm!(
        // 1. Platform initialization (trap handler, etc.)
        "   call    {platform_bootstrap}",

        // 2. Runtime environment initialization (never returns)
        "   tail    {runtime_bootstrap}",

        // Safety: If main() returns, halt forever.
        // User should call exit() to terminate properly.
        // Using inline asm instead of `loop {}` or `spin_loop()` because:
        // - `loop {}` may be optimized away as undefined behavior
        // - `spin_loop()` generates `pause` only with Zihintpause extension,
        //   otherwise no instruction on RISC-V
        // - `j .` guarantees a single-instruction infinite loop
        "   j       .",

        platform_bootstrap = sym __platform_bootstrap,
        runtime_bootstrap = sym __runtime_bootstrap,
    )
}
```

C __runtime_bootstrap

```rust
#[unsafe(naked)]
#[no_mangle]
pub unsafe extern "C" fn __runtime_bootstrap() -> ! {
    naked_asm!(
        // Build musl stack in buffer (returns buffer_sp in a0)
        "   call    {build_impl}",

        // Snapshot current platform SP after helper returns
        "   mv      t4, sp",

        // Calculate buffer_top, size, and target_sp
        // t0 = buffer_top, t1 = buffer_bytes, t2 = size
        "   la      t0, {buffer}",
        "   li      t1, {buffer_bytes}", // t1 = buffer size in bytes
        "   add     t0, t0, t1",         // t0 = buffer_top
        "   sub     t2, t0, a0",         // t2 = size (buffer_top - buffer_sp)

        // Calculate target_sp (where to copy the stack)
        // t3 = target_sp (current_sp - size)
        "   sub     t3, t4, t2",         // t3 = target_sp
        "   mv      t5, t3",             // t5 = saved target_sp for final SP

        // Copy loop: copy from buffer to target location
        // t6 = src (buffer_sp), t3 = dst (current position), t2 = remaining bytes
        "   mv      t6, a0",             // t6 = src (buffer_sp)
        "1:",
        "   beqz    t2, 2f",             // if size == 0, done
```

```
        "   ld      t1, 0(t6)",          // load 8 bytes from buffer
        "   sd      t1, 0(t3)",          // store 8 bytes to target
        "   addi    t6, t6, 8",          // src += 8
        "   addi    t3, t3, 8",          // dst += 8
        "   addi    t2, t2, -8",         // size -= 8
        "   j       1b",                 // loop
        "2:",

        // Switch to the new musl stack (use saved target_sp)
        "   mv      sp, t5",             // sp = target_sp (start of copied stack)

        // Now SP points to musl stack with layout:
        //   SP+0: argc
        //   SP+8: argv[0]
        //   SP+16: argv[1]
        //   ...

        // Load argc and argv from musl stack
        "   lw      a1, 0(sp)",          // a1 = argc
        "   addi    a2, sp, 8",          // a2 = argv

        // Load function pointers for __libc_start_main
        "   la      a0, {main}",         // a0 = main function
        "   la      a3, {init}",         // a3 = _init
        "   la      a4, {fini}",         // a4 = _fini
        "   li      a5, 0",              // a5 = NULL (ldso_dummy)

        // Tail call __libc_start_main (never returns)
        "   tail    {libc_start_main}",

        build_impl = sym build_musl_in_buffer,
        buffer = sym MUSL_BUILD_BUFFER,
        buffer_bytes = const MUSL_BUFFER_BYTES,
        main = sym __main_entry,
        init = sym _init,
        fini = sym _fini,
        libc_start_main = sym __libc_start_main,
    )
}
```

D build_musl_stack

`build_musl_stack` pushes the required `auxv` entries onto the stack. Note that ZeroOS must set `AT_PHNUM` (program header count) to 0 to ensure `musl` skips reading the ELF headers (which may not be accessible) and instead use the provided defaults.

```
pub unsafe fn build_musl_stack(
    _initial_sp: usize,
    _ehdr_start: usize, // Ignored - ELF header not accessible in bare-metal
    program_name: &'static [u8],
) -> usize {
    let mut ds = DownwardStack::<usize>::new(_initial_sp);

    // Zero-auxv approach: Tell musl that program headers are not available.
```

```
// Musl will skip reading PT_TLS from the ELF header when AT_PHNUM=0.
let at_phdr = 0; // No program headers available
let at_phent = 0; // No program header entry size
let at_phnum = 0; // No program header count

// Prepare auxiliary vector entries
let auxv_entries = [
    (AT_PHDR, at_phdr),
    (AT_PHENT, at_phent),
    (AT_PHNUM, at_phnum),
    (AT_PAGESZ, 4096),
    (AT_CLKTCK, 100),
    (AT_HWCAP, 0),
    (AT_UID, 0),
    (AT_EUID, 0),
    (AT_GID, 0),
    (AT_EGID, 0),
    (AT_SECURE, 0),
    (AT_RANDOM, 0), // Will be replaced with actual pointer below
    (AT_NULL, 0),
];

unsafe {
    // Generate 16 bytes for AT_RANDOM (Linux kernel standard)
    // Musl's __init_ssp uses first sizeof(uintptr_t) bytes for stack canary
    // Use multiple entropy sources: sp and a constant
    let entropy = [_initial_sp as u64, 0xdeadbeef_cafebabe_u64];
    let (random_low, random_high) = generate_random_bytes(&entropy);

    // Push 16 bytes in architecture-appropriate chunks
    //
    // Memory layout after pushes (stack grows downward, lower addresses at bottom):
    //
    // 32-bit architectures (4-byte usize):
    //   sp+12: random_high[63:32]  (4 bytes)
    //   sp+8:  random_high[31:0]   (4 bytes)
    //   sp+4:  random_low[63:32]   (4 bytes)
    //   sp+0:  random_low[31:0]    (4 bytes) ← at_random_ptr, musl reads 4 bytes
    //
    // 64-bit architectures (8-byte usize):
    //   sp+8:  random_high[63:0]   (8 bytes)
    //   sp+0:  random_low[63:0]    (8 bytes) ← at_random_ptr, musl reads 8 bytes
    //
    #[cfg(target_pointer_width = "32")]
    {
        ds.push((random_high >> 32) as usize); // High 32 bits of random_high
        ds.push(random_high as usize); // Low 32 bits of random_high
        ds.push((random_low >> 32) as usize); // High 32 bits of random_low
        ds.push(random_low as usize); // Low 32 bits of random_low (musl reads from here)
    }
    #[cfg(target_pointer_width = "64")]
    {
        ds.push(random_high as usize);
        ds.push(random_low as usize); // Musl reads from here
```

```
        }

        let at_random_ptr = ds.sp();

        // Push auxiliary vector (in reverse order since stack grows downward)
        for &(key, val) in auxv_entries.iter().rev() {
            let eff_val = if key == AT_RANDOM { at_random_ptr } else { val };
            ds.push(eff_val);
            ds.push(key as usize);
        }

        // Push envp (empty, NULL-terminated)
        ds.push(0);

        // Push argv (program name + NULL terminator)
        ds.push(0);
        ds.push(program_name.as_ptr() as usize);

        // Push argc
        ds.push(1);
    }

    ds.sp()
}
```