University of
St Andrews

Stardust Oxide

Ferdia McKeogh

Supervised by Prof. Alan Dearle and Dr. Tom Spink

Abstract

Stardust Oxide is a unikernel, single-address-space Xen-paravirtualised library operating system written in the Rust language. It supports dynamic memory allocation, `async`/`await` syntax for cooperative multitasking, virtualised networking with a TCP/IP stack and a performant console driver. There are type- and memory-safe interfaces to Xen functionality such as shared memory, the XenStore and the XenBus. It boots in less than 3 milliseconds with a compressed image size of 61KB (including the networking stack). The project is structured as three libraries: one containing automatically-generated Rust bindings to the C Xen API, one with ergonomic Rust interfaces to the bindings and then the Stardust Oxide package. This permits other operating system developers to re-use these individually as needed. The software also forms an alternative reference Xen guest implementation to MiniOS and Stardust while benefiting from Rust's expressive type system, polymorphism and safety guarantees.

`https://github.com/StardustOS/stardust-oxide`

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science

1 Acknowledgements

I would like to thank Profesor Alan Dearle for his continual support and guidance throughout this project, Doctor Tom Spink for his insightful advice and debugging assistance, and Doctor Ward Jaradat for his helpful contributions.

I would also like to thank my parents for their love and support throughout my time at university.

Finally, I will thank my friends and flatmates, without whom I would have undoubtedly completed this work far sooner.

Contents

2 Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 9,819 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

3 Ethics

No personal data was collected or processed nor were there participants in any studies in the course of this research and as such there were no ethical considerations required. The completed ethics evaluation may be found in Appendix 12.1.

4 Terminology

4.1 Terms

Operating System: "software that controls the operation of a computer and directs the processing of programs" - *Mirriam-Webster*[27].

Kernel: Component of an operating system which manages hardware resources and their interaction with applications.

Unikernel: Approach to operating system design wherein the application software is compiled alongside the operating system into a single image, where application and kernel code both execute in the same address space.

Hypervisor: Software which emulates virtual machines; the hardware running the hypervisor is known as the *host* and the virtual machines being emulated are known as *guests*.

Paravirtualisation: Virtualisation that utilises software interfaces rather than by emulating a full hardware environment to run a guest operating system in.

4.2 Technologies

Rust: Synonymous with "the Rust programming language" but not with the *Rust Foundation* (the organisation holding Rust trademarks and managing finances) or the *Rust Language Team* (the organisation of developers working on the Rust compiler and making design decisions).

Xen: Synonymous with "the Xen hypervisor".

5 Introduction

Unikernels[26] generally offer greater performance and smaller attack surfaces than monolithic kernels at the cost of flexibility and suitability for deploying multiple applications on a single host. The aim of this project was to develop a unikernel for the purposes of running distributed WebAssembly applications and provide an improved implementation of the MiniOS[9] reference operating system for the Xen[3][7] hypervisor through the use of the Rust language[31]. The three top level goals of the project were implementing *multitasking*, *networking*, and *memory management*, all of which have been achieved.

A study on vulnerabilities in cryptography libraries found that while 27% of vulnerabilities were cryptography-related issues, 37% were memory unsafety[5] (the remaining were classified as input validation, information exposure and numeric errors). The Chromium project reports an even higher figure; that 70% of high severity security bugs were related to memory safety[36]. It follows therefore that a significant portion of issues could be prevented if the software is written in a memory safe language. This forms the motivation for using the Rust[31] language for systems software. Rust has been successful when used in a variety of systems applications (particularly those that are high-performance and required high reliability) such as in garbage collectors[23]. Rust was used for this project with the aims of evaluating the benefits of type and memory safety in operating system development and whether the security benefits of a unikernel are compounded with the use of a memory-safe language.

While not all Stardust[16] and MiniOS functionality has been implemented in Stardust Oxide, useful applications can be written and in that respect the project was successful in achieving the initial goals.

5.1 Multitasking

The original project plan described several options for preemptive scheduling in order to implement multitasking. A cooperative multitasking approach was not originally considered however it was significantly easier (as the implementation complexity is deferred to the language). One of the primary drawbacks of cooperative multitasking is that malicious tasks could simply never yield control back after being scheduled and prevent other tasks from executing (known as “starvation”). However unikernels typically assume that the application

(and therefore any and all of its tasks) is trusted by nature of being compiled into the deployed image, and do not typically allow the installation and execution of new software at runtime. For this reason cooperative multitasking was found to be suitable to use in this project.

"Async/await" is a common pattern in asynchronous programming, usually where `async` is a keyword applied to a function or block, and `await` is a keyword to execute an `async`-marked function, asynchronously. Asynchronous execution is commonly implemented with a green threading model using lightweight tasks that can yield when performing IO. Since 2019 Rust has supported the `async/await` syntax for concurrent programming[2]. Four of the five most widely used Rust web frameworks support `async/await`[19]. Typically `async/await` applications are deployed to Linux hosts where the `async` runtime executes futures using a threadpool. Building the `async` runtime directly into the operating system removes the need to implement a preemptive threading system thereby reducing the surface for bugs and security vulnerabilities. A Rust executor was successfully implemented as well as a `Delay` future for that blocks for the specified duration when awaited.

5.2 Networking

The paravirtualised network driver made available to guest operating systems by the Xen hypervisor has two "halves": a back-end running on the hypervisor host and a front-end running in the guest operating system. Each half communicates using ring buffers within shared memory (pages mapped into the address spaces of both operating systems). Packets are placed in pages then the ring buffer updated to indicate they are ready to be transmitted or received.

`smoltcp`[33] is a Rust TCP/IP stack. It defines an interface for a physical layer device; therefore adding support only requires a thin "shim" between the Xen frontend driver and `smoltcp`. From there any applications using Stardust Oxide have access to TCP, UDP, ICMP and raw sockets.

5.3 Memory Management

On startup Xen provides information on the number and location of page frames of memory to guest operating systems, along with a pointer to the base of where the page table should be placed. This information is used to build the page table; the result of which

is a contiguous sequence of mapped virtual pages which the memory allocator may use. The page table infrastructure has been totally re-implemented in Rust; one of the major improvements in design is representing different memory addresses and numbers with different types. In the original C implementations `unsigned long` is used to represent virtual addresses, machine addresses, frame numbers, etc, using a series of `A_to_B` conversion functions. In Stardust Oxide these are each their own type and conversions are implemented using the `From` trait. This eliminates whole classes of bugs; it's impossible to convert between types that cannot be converted, pass in the wrong type to a function or return the wrong value. For example there is a function for mapping a new page table frame. The Rust signature is `pub unsafe fn new_frame(l4_table: *mut PageEntry, pt_pfn: PageFrameNumber, prev_l_mfn: MachineFrameNumber, offset: isize, level: usize)`. In C the signature is `static void new_pt_frame(unsigned long pt_mfn_for_pfn, unsigned long prev_l_mfn, unsigned long offset, unsigned long level)`. The frame to be mapped has the type `PageFrameNumber` so unlike the C version will give a compile time error if a machine frame number was passed in.

When the project was started it was unclear whether an existing Rust memory allocator would be usable due to the level of integration with Xen memory management however due to the separation of the page table infrastructure several "drop-in" memory allocators are compatible with the current implementation. A linked list-based allocator[30] was used for most of the development but was replaced with a buddy system allocator[18][6]. Future work could involve porting Jemalloc[12] or other state-of-the-art allocator to Stardust Oxide.

6 Context survey

6.1 Xen

Xen[3] is an open source hypervisor, originally developed at the University of Cambridge Computer Laboratory but now developed by the Linux Foundation. It is a type 1 hypervisor; it runs on the real hardware and presents an emulated environment (virtual machines) for guest operating systems. This is in contrast to type 2 hypervisors which are programs running in an operating system.

Xen offers two virtualisation modes: paravirtualisation (PV) and hardware virtual machine (HVM). PV requires explicit operating system support by presenting a software interface to guest virtual machines. HVM by contrast provides a bare metal hardware environment such that guest operating systems can run unmodified and unaware of the virtualisation. PV was the first Xen mode, introduced in 2003[32] and required no hardware support. However both Intel and AMD introduced virtualisation-specific instruction extensions which add an additional "theoretical" ring below ring 0, where the hypervisor resides, allowing guest operating systems to execute in ring 0 where they were designed to run.

Originally, PV offered greater performance as HVM required an expensive trap for each privileged instruction executed by the guest and PV software I/O drivers were more efficient than emulating hardware devices for HVM. However more recent improvements in hardware support for virtualisation and a "hybrid" approach has yielded the best performance. This combines paravirtualised drivers for I/O with hardware-accelerated HVM in a mode known as "PVHVM"[39]. The relative performance of various combinations of Xen guest modes can be seen in Figure 2 and the years in which support for various Xen modes were added are shown in Figure 1.

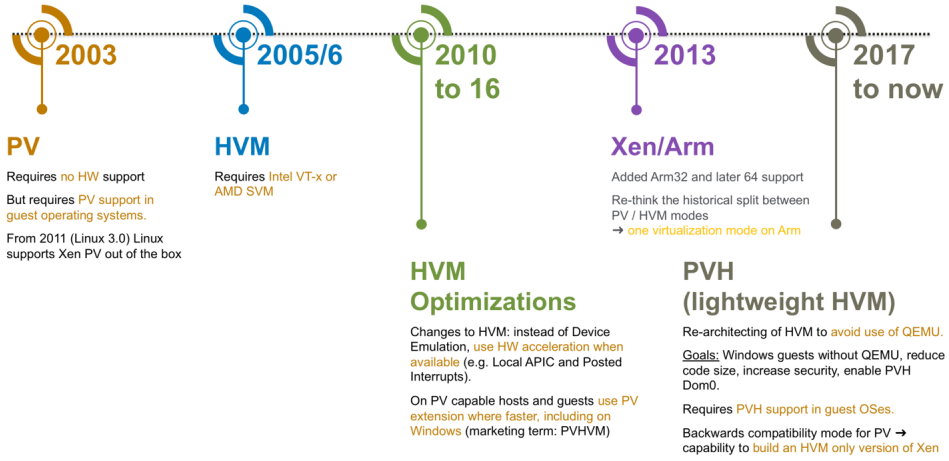


Figure 1: Diagram of supported Xen over time from the Xen wiki[39].

Poor Performance
Scope for Improvement
Optimal Performance

PV = Paravirtualized
VS = Software Virtualized (QEMU)
VH = Hardware Virtualized
HA = Hardware Accelerated

x86 Shortcut	Mode	With	Disk and Network	Interrupts & Timers	Boot Path	Privileged Instructions, Page Tables	QEMU Used
HVM / Fully Virtualized	HVM		VS	VS[1]	VS	VH	Yes
HVM + PV drivers	HVM	PV Drivers Installed	PV	VS[1]	VS	VH	Yes
PVHVM	HVM	PVHVM Capable Guest	PV	PV[2]	VS	VH	Yes
PVH	PVH	PVH Capable Guest	PV	HA[3]	PV[4]	VH	No
PV	PV		PV	PV	PV[5]	PV	No
ARM							
N/A	N/A		PV	VH	PV[6]	VH	No

Figure 2: Table of relative performance of Xen modes from the Xen wiki[39].

6.2 Unikernels

MiniOS[9] is the Xen first-party reference unikernel guest operating system, containing minimal implementations for most Xen functionality. However substantial portions of the codebase are close to a decade old and the most recent commit was in 2016. There is a general lack of documentation, especially for a codebase intended to be used as a reference implementation. Furthermore the quality of software itself is poor: there is not a consistent style in naming, there is heavy use of macros which obfuscates the underlying logic of many functions and there are many deep interdependencies between modules. Writing clear boundaries with well-defined interfaces is difficult in C, however it results in unnecessary complexity for example when the implementation for a free list, a ring buffer and the network front end are all combined. The MirageOS[25] unikernel from the Xen project is based on MiniOS and intended for OCaml applications, with over 1,000 library packages available.

The Stardust[16] operating system was created for distributed microservices[17] and lambda functions[15], used as a platform for research and forms the main reference when implementing Stardust Oxide. Stardust supports multithreading with a preemptive scheduler (on multiple cores), block and network devices, and runs a Java interpreter.

Include OS[8] has a remarkably simple setup requiring only an `#include` statement to compile a C++ program into a unikernel image. However there has not been recent development on the repository and the project does not have a stable API, making it unlikely to see widespread adoption.

Unikraft[20] is another unikernel with substantial commerical adoption. Its design has demonstrated very impressive performance benchmarks showing a 166% improvement over Linux[20]. Crucially POSIX compatibility (with over 160 syscalls implemented so far) and the use of the `musl` libc[28] allows Unikraft to run a very wide range of applications. The developer experience is also improved by providing a text user interface configurator application for customising a Unikraft image, instead of doing so with manual modifications to Makefiles or build scripts like with other unikernels.

6.3 Rust

Rust is a general-purpose, strongly- and statically-typed compiled language with the key features of type, memory and thread safety. Instead of using a garbage collector or reference

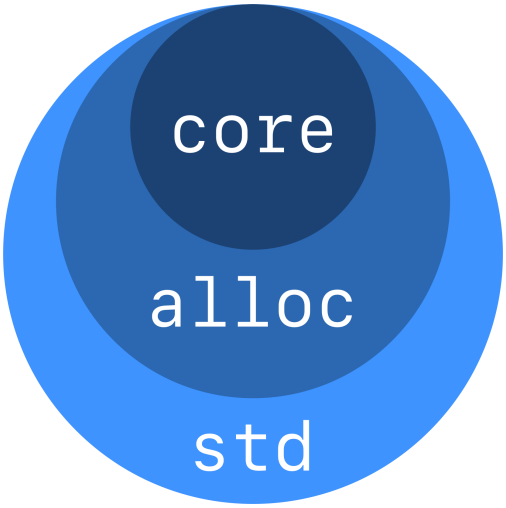


Figure 3: Diagram showing the three constituent libraries of the Rust standard library.

counting, which require a language runtime and incur a performance penalty, Rust uses a compile time "borrow-checking" ownership model to validate references. The invariant upheld is that there may only exist either many immutable references to a value *or* a single mutable reference. The syntax for this is `&Foo` for an immutable reference to an instance of `Foo` and `&mut Foo` for a mutable reference. This prevents data races and ensures thread safety. Rust has been the "most loved" programming language in the StackOverflow Developer Survey every year since it's 1.0 release[34]. `cargo` is the package manager for Rust. It allows Rust libraries or binaries (known as "crates") to declare dependencies, standardises the build process, and automatically invokes the Rust compiler with the correct parameters.

The Rust language can also be viewed as two languages: Safe Rust and Unsafe Rust. Unsafe Rust enforces all the same memory safety guarantees as Safe Rust, however it allows for the dereferencing of raw pointers, accessing and modifying mutable static variables, and calling unsafe functions or methods. Using the `unsafe` keyword is a contract with the Rust compiler where the developer has the responsibility to uphold all the invariants in situations where Safe Rust is insufficient.

The Rust Core Library (`libcore`) is platform agnostic with no dependencies and defines Rust intrinsics and primitives, providing no heap allocation, concurrency or I/O. The Rust Standard Library (`libstd`) is dependant upon the `alloc` library which provides smart pointers

and heap-allocated collections structures, both of which depend on `libcore`. This relationship is shown in Figure 3. When writing a hosted application, such as to run on Linux, the Rust Standard Library is used to provide the developer with a set of abstractions for interacting with the system and is enabled by default. However not all software environments are hosted, for example when writing microcontroller firmware the software is executed directly on the hardware with no intermediary layers like an operating system. In these "bare-metal" environments the `#![no_std]` attribute in Rust indicates a package will be linked against `libcore` rather than `libstd`. `alloc` may be used as a dependency (if a global allocator is provided) enabling heap-allocated collections in bare-metal applications.

Functions in Rust may be related to a particular type. The two forms are associated functions, which are functions defined on a type, and methods, which are functions defined on a specific instance of a type (and can be called using dot-suffix syntax). Both are defined inside an `impl` block. For example, `new` is a common associated method name for a function which creates a new instance of a type. For example, `Foo::new() -> Foo` is the signature of an associated function which returns an instance of `Foo`. Methods can take an instance of a type by value or by reference. For example, `fn increment(bar: &mut Bar)` would be used as `bar.increment()`. However inside an `impl` block the type can be referenced using `Self`. So both functions could be more cleanly written as `new() -> Self` and `fn increment(bar: &mut Self)`. However for methods, `self` can be used as shorthand so `increment` can become `fn increment(&mut self)` and the `self` variable used in the body of the function.

Rust uses "traits" as the basis for parametric polymorphism and are similar to "interfaces" in other languages. Traits formally describe shared behaviour as functions, constants, and other traits that must be implemented by implementors. Generic parameters in functions and generic fields in structs and enums can be constrained with a sequence of traits they must implement. For example the function signature `fn foo<T: Bar>(a: T)` describes a function which takes a single argument `a` which has type `T` where `T` can be any type which implements the `Bar` trait. The Rust compiler will perform monomorphisation here, duplicating the function for each unique concrete type it is used with. However, if one desired a dynamic array (`Vec`) of elements that implement the `Bar` trait, using `Vec<T>` would only allow elements of the same type. If a collection of heterogeneous types or, specifically, dynamic dispatch is required, then the `dyn Trait` key word can be used. However, the type `dyn Bar` is "unsized", the size cannot be determined at compile time and therefore it would be impossible to know how much stack space to allocate. Instead it must be heap allocated,

which can be achieved by using the `Box` struct (the simplest form of heap allocation in Rust). `Box<dyn Bar>` does have a size, equal to the size of a pointer on the target platform, and so can be passed as a parameter or stored on the stack.

Rust has language-level support for writing asynchronous software. This is exposed with the `async` and `await` keywords and standard library support in the form of the `Future` trait. The `async` keyword can be placed before a function or block to make it asynchronous and allow the use of the `await` syntax. It will return the `Future` type which when itself awaited returns the output value within the future. An executor is used to poll a collection of `Future`s. When synchronous functions block the thread is also blocked, however when a (correctly implemented) asynchronous function blocks it yields control allowing the executor to run other asynchronous functions. The Rust compiler transforms `async` blocks or functions into state machines which implement the `Future` trait where `await`s are transformed into yield points.

The API for mutexes in Rust used by both the standard library[29] and 3rd-party libraries also prevents misuse to a greater degree than in C or C++[35]. Mutexes in the latter languages must be added as fields to the structure being guarded and make it the users responsibility to acquire a lock before mutating any other fields. A user will not be prevented from accessing the shared resources without locking. However, `std::lock_guard` in C++ does provide RAII-style release-on-drop semantics. In Rust however, a mutex takes ownership of the data it is guarding and acquiring a lock returns a handle to a mutable reference to the inner data. This makes it impossible (in safe Rust) to access or mutate the guarded data without first acquiring a lock. When the handle is reaches the end of its scope it is dropped and the mutex unlocked. Additionally, Rust's error handling also represents an improvement here; in C++ `std::mutex::try_lock` returns a boolean that the user must check but failing to acquire the lock does not prevent later misuse. In Rust the return type of `std::sync::Mutex::try_lock` is `Result<Guard, TryLockError<Guard>>`. If acquiring the lock fails the user receives an error structure (`TryLockError`) and cannot access the guarded data.

The `Drop`[11] trait is how custom destructor functionality is implemented in Rust. Destructors for fields are called recursively and handled by the language so manually implementing `Drop` is not usually required. However when managing more complex resources (such as a network connection) being able to run a routine to cleanly destroy the resource is useful.

7 Software Engineering Process

Test driven development[1] is usually well suited when porting one project to another language or system; if the goal is to produce software with the same behaviours and functionality, and success is determined by how closely it matches those behaviours and functionality, then creating a large collection of tests up front is reasonable. The development process can then be a cycle of picking a failing test, writing software until it passes, then repeating.

However that model was not suitable for this project. Most of the implementation work was for Xen-specific interfaces. There is generally not much behaviour to be observed and measured other than it simply working or not. While there are some features that can be tested (such as the XenStore where a meaningful test of writing a key-value pair then reading back was possible), in general validating behaviour through unit tests was not possible as the behaviour was not measurable from inside the OS (such as printing text to the console). There is a high degree of stability in the Xen API and so once a feature has been manually shown to work, writing a regression test would offer little value in ensuring the correctness going forward. Neither of the two largest bugs encountered during development would have been caught with unit tests, nor would unit tests have sped up development.

Instead a waterfall approach was taken; the goal was running a multitasking TCP server as the application, this has a significant tree of dependencies on features, and so the development process was to implement each feature working upwards until Stardust Oxide could support the desired application. Running the TCP server example uses every aspect of the implementation; during development manually running the features required for the TCP server as they were being implemented was more than sufficient.

8 Design

8.1 Anti-goals

Rust has both declarative and procedural macros. Declarative macros defined using `macro_rules!` allow for general metaprogramming with substitution based on matched patterns. Procedural macros more powerful, they are programs which take Rust source as input and produce Rust source as output. While both are very useful tools, due to how heavy use of macros in the MiniOS implementation negatively affected readability, it was a primary goal in the Stardust Oxide to minimise their use.

Conditional compilation also negatively impacts readability; Gazzillo and Wei describe it as "a serious impediment to the quality of C code"[13]. Having n options results in n^2 configurations where each configuration must be tested unless compilation options are written carefully to not interfere. For this reason this project will rely on dead code elimination to remove unused features rather than requiring users to manually specify which options they require.

8.2 Project Structure

The project was split into three crates, `xen-sys`, `xen`, and `stardust`, each dependent on the last and exposing a higher level interface. The module structures can be seen in Figures 4 and 5. Crates are shown in blue, modules which are public are green, modules which are public only within the crate are yellow and private modules are red. As `xen-sys` is generated automatically from C headers and C does not have namespaces or any hierarchy, the module is internally flat with every item appearing in the crate root.

8.3 Continuous Integration

GitHub Actions[14] was chosen as the continuous integration (CI) provider for this project due to the convenience of using the same provider as for hosting the project repository.

It is not possible to run the Xen hypervisor within a GitHub Actions runner so while Stardust Oxide cannot be run in CI, there are other useful tests that can be performed. The first

Figure 4: Module structure of `xen` crate.

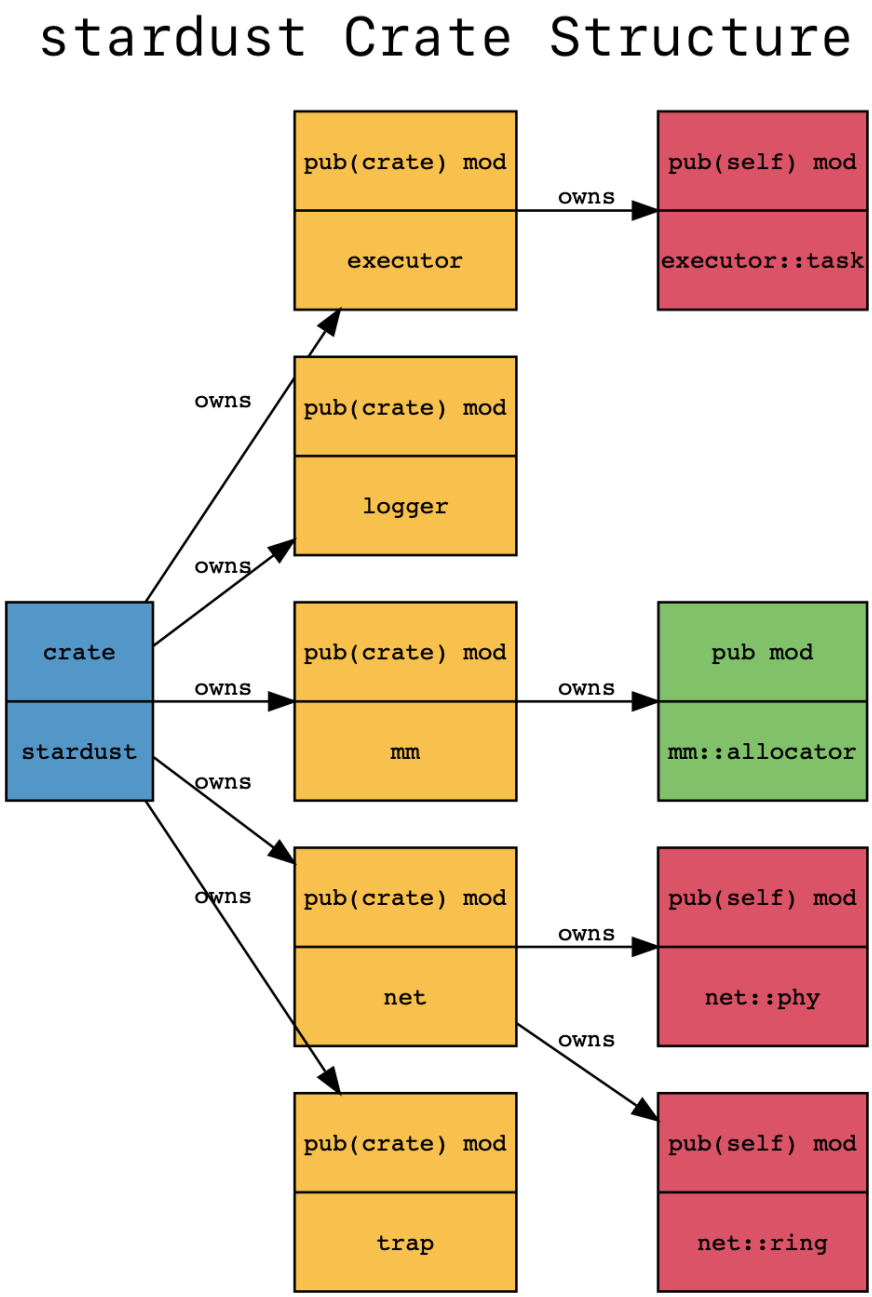


Figure 5: Module structure of `stardust` crate.

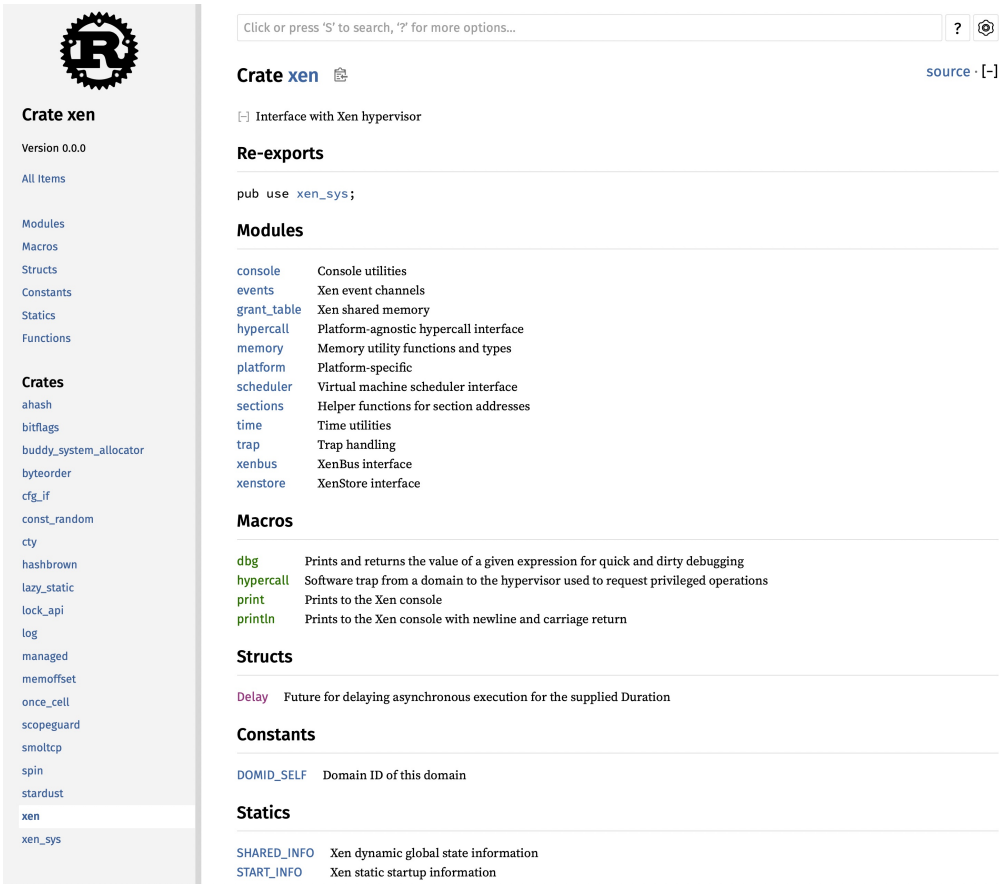


Figure 6: Screenshot of generated documentation for `xen` crate.

is a check that the codebase is formatted using `cargo fmt`. This is important to maintain consistent styling, especially as the number of contributors to a project grows. The second is to build the project, failing if there are any warnings. Compiler warnings are acceptable during local development but it was important that all branches at all times contain valid, working software that compiles without issue (although active branches may have incomplete or non-functional features).

8.4 Documentation

`rustdoc`[38] is the documentation tool included in the Rust toolchain which runs on a Rust crate and produces a static website containing documentation for that crate and all its dependencies. The generated documentation for the `xen` crate is shown in Figure 6, and is deployed to GitHub Pages automatically as part of the continuous integration pipeline and can be found at `https://stardustos.github.io/stardust-oxide/xen/index.html`.

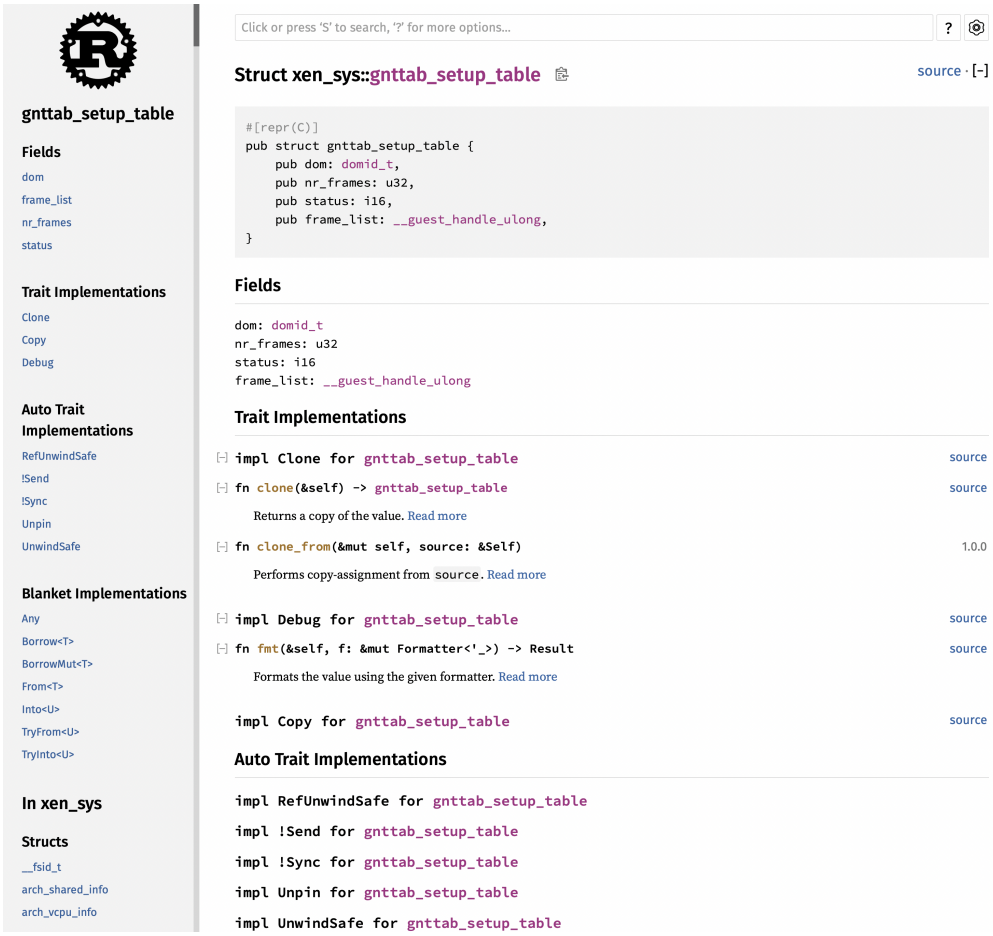


Figure 7: Screenshot of generated documentation for the `xen-sys::gnttab_setup_table` struct.

As an example, Figure 7 shows the documentation for the `gnttab_setup_table` struct in `xen-sys`. All the public fields are shown with their type; non-primitive types have hyperlinks to their documentation pages. There is a list of all traits implemented by the struct including the methods for each. "Auto Trait Implementations" are traits automatically implemented by the compiler. In this example the field `frame_list` of type `__guest_handle_ulong` is a raw mutable pointer; the compiler has determined that it is not safe to send to another thread (`!Send`), nor safe to share between threads (`!Sync`), which means neither is the `gnttab_setup_table` struct.

8.5 Single Initialisation of Shared Resource

Interfaces to several Xen systems can be neatly expressed in an object-oriented fashion. For example the XenBus (discussed in section 9.9) was implemented as a structure containing a pointer to the ring buffer that messages are placed onto, an event channel port number, and a B-Tree map of message IDs and their responses. Users of the operating system need to be able to write software that can interact with XenBus safely in a variety of contexts. One initial solution might be to return the structure from an initialisation function and make it the user's responsibility to use it safely. However this does not prevent the user from creating multiple instances of the structure and since they are managing the same underlying resources this would not be thread safe.

A better alternative might be to expose many public functions which enforce mutual exclusion internally before accessing the underlying resources stored in mutable static variables. This is reasonable in C, but in Rust accessing or modifying mutable static variables is unsafe. Additionally, the initialisation of these variables could result in an incorrect state where some are valid and some are invalid, and could introduce subtle issues due to the use of uninitialised memory.

The solution arrived at solves all the above issues. A structure is defined containing all the resources required. Initialisation and all required functionality are implemented as methods on this structure. This results in tidy encapsulation and type of the `self` parameter indicates clearly which methods mutate the underlying resources. The `lazy_static`[22] crate enables runtime initialization of static variables, otherwise not possible in safe Rust. A mutable static variable containing a mutex containing this structure is created with the body of the `lazy_static` macro calling the initialisation method of the structure. Finally functions are exposed for each method; inside each a lock on the mutex is acquired and then the associated method called. Since the structure is private, the only way for users to interact with the resources are through the public functions, each of which ensure mutual exclusion and therefore thread safety.

8.6 Platform Support

Only supporting x86_64 was in scope for this project; while ARM64 would be important to showcase the portability of Rust software, obtaining the necessary hardware would be

infeasible. [1]

Despite conditional compilation being a design anti-goal it must be used in platform-specific software; use of inline assembly for one architecture will not compile when targeting another so dead-code elimination is insufficient. To keep the remainder of the codebase portable the "platform" module keeps all the platform-specific implementations private then re-exports whichever platform has been enabled. This way other modules can use `platform::time::get_system_time` or `platform::consts::PAGE_SIZE`, which internally will be platform-specific.

However this implementation only requires that all platform trees export the same module names, and the consistency between types and functions (and their existence) are only tested at the location if and when they are used. For example if a future ARM64 tree implemented a `get_system_time` function that returned a pair of `u32` values for seconds and nanoseconds since the UNIX epoch instead of a single `u64` in nanoseconds, the compiler error would occur if the return value was being assigned to a `u64`, and not in the ARM64 tree where the error truly lies.

An improved approach to enforcing consistency might be to define a `Platform` trait with a long list of associated constants and methods (such as `PAGE_SIZE` and `get_system_time`). This then clearly documents what all platforms must support and would result in an error at the semantically-correct location if not met. However, this would break the module structure; a flat list of methods and constants is not as tidy as a nested structure. Additionally the concrete platform type would have to be passed around or kept in a static, and most importantly, would still necessitate the use of conditional compilation.

[1] The author does have a Raspberry Pi 3 and while it exceeds the minimum system requirements of Stardust Oxide, the development experience would be severely impacted by high compilation time if developing locally. Cross-compilation would be a solution, although it might complicate the execution process requiring to first copy the image then run.

9 Implementation

9.1 Bootstrapping

By default, starting and running a new Rust project (using the `cargo` package manager's `cargo new` and `cargo run` commands) will create an empty crate with a single `main.rs` file containing an entrypoint function `main` printing "Hello, world!". `cargo` will automatically build a binary for the host architecture, linked against the Rust standard library, and execute it. This is not a suitable environment for building an operating system.

One primary issue is that the operating system image is running standalone on emulated x86_64 hardware not as a program running within an operating system; this means it cannot use the standard library and the executable must meet the specification set by Xen. Rust shares the build target naming scheme with LLVM[21] so the target can be set to `x86_64-unknown-none` to enable cross-compililation. Many targets have pre-build standard library artefacts distributed with the toolchain but newer and less popular targets do not. This means the `core` and `alloc` libraries must be built manually. This used to require external tools and a more complex build process however `cargo` now has an (unstable) feature[37] to build the standard libraries as part of regular compilation invoked with `cargo build` or `cargo run`.

The second developer experience improvement is to enable the use of `cargo run` in the development process. The default behaviour is to build the binary and attempt to execute it, which does not apply for this project. Instead the `runner` configuration key can be set to the `run.sh` script which creates a Xen guest configuration file (containing the correct path to the emitted binary) and then starts a new Xen virtual machine from that configuration.

By default the Rust compiler will emit position-independent code. This is useful for shared libraries, for example, placed in different locations in each address space of the program they are being used by. However, this causes a linker error when the inline assembly for hypercalls attempt to use the address of an external symbol, and so the Rust compiler flag for a static relocation model is set. A custom linker script (`link.x`) is also required to define the memory layout required by Xen. `xen/link.x` defines the ordering, location and size of the various sections and also defines the stack region after `.bss`. `bootstrap.S` contains the x86_64 assembly required by Xen to boot the operating system. The text section contains Xen specific definitions such as the guest OS name and Xen interface version. The `_start`

label (called by Xen on boot) calls `start_kernel` which is defined in `stardust/main.rs`. Cargo accepts a file named "`build.rs`" as a pre-build script. This script uses the `cc`[10] crate, a library wrapping the local C or C++ compiler, to compile the assembly file.

9.2 Xen Bindings

To interact with Xen requires the use of structs, unions, and constants defined in the Xen header files. Rust has strong support for integrating with existing C and C++ software, even so far as having first class support for setting the layout of a structure to be identical to C with the `repr(C)` attribute. However manually writing Rust structures that match the C library being interfaced with is time consuming and error prone; the Bindgen[4] crate solves this problem by automatically generating FFI bindings from C headers. Since these are a direct mapping the generated software is not guaranteed to be safe or likely to be particularly easy to use. A frequent pattern in the Rust ecosystem is that there is the need to use a C or C++ library from Rust and so Bindgen is used to generate a crate containing the FFI software (typically given a `-sys` suffix). A second Rust crate, depending on the former, then creates an improved (both in terms of safety and usability) interface to the C or C++ library. In this project the FFI crate is `xen-sys` and the second crate containing the improved interface is `xen`.

The `xen-sys` crate contains `wrapper.h` which includes all required Xen headers. A `build.rs` script is used to run Bindgen on the `wrapper.h` and emit a Rust source file. Then in the crate's `lib.rs` the `include!` macro parses the emitted file as an expression, thereby including all the generated software in the final crate when compiled.

9.3 Hypercall

Implementation found in `xen::hypercall` and `xen::platform::x86_64::hypercall`.

To perform privileged operations (such as an MMU update, setting a timer, or binding an event to a virtual IRQ) from a guest Xen uses "hypercalls". These fulfill a similar function to system calls in most operating system, but instead of placing arguments in registers and issuing an interrupt to jump to the privileged interrupt handler to be processed, hypercalls are issued by calling an address at some offset into the "hypercall page". Hypercalls can have from 0 to 5 arguments and therefore need 6 corresponding functions.

In Rust these functions were implemented using inline assembly in the x86_64 platform tree, then exposed as a single `hypercall!` macro which takes a variable number of arguments and expands to the corresponding platform hypercall function. While it was a design goal to avoid macros, it was justified here due to the simplicity of the macro and ergonomic improvement. However it does have the consequence that if an argument of the wrong type is supplied the error span is in the macro definition rather than the site of the invocation of the macro, which can be confusing.

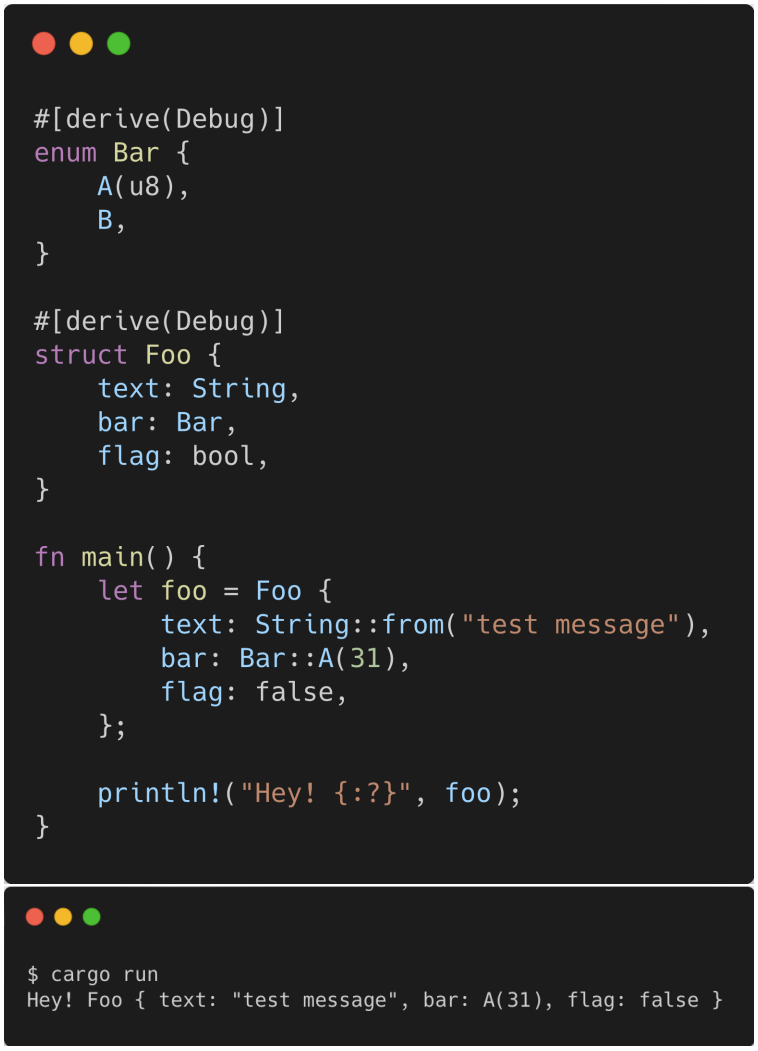
9.4 Console

Implementation found in `xen::console`.

In the startup information provided by Xen there is a pointer called `xencons_interface` which points to the console ring buffer structure containing the data array and producer/consumer index pairs for both console input and output. Writing text was implemented by waiting for the buffer to be flushed in a while loop until the producer and consumer indices were equal, then writing each byte (*not* character as Rust characters are UTF-8 codepoints) to the array at the output producer index. After a memory fence the producer index is incremented and a notification sent on the console event channel to alert Xen that there is new data to be read.

The `println!` macro is a convenient way to print text to the screen in Rust programs. In hosted applications this would be `STDOUT`. Supplying only a string literal (text within a pair of double quotation marks) prints the string literal. However by using the Rust formatting syntax it is possible to supply variables and have those printed. The two traits are `Display` and `Debug`, which are intended to format the implementing types for user facing and debugging output respectively. Using the `#[derive(Debug)]` attribute an implementation of the Debug trait will be created automatically. This is demonstrated in Figure 8, using the {`:?`} syntax in the `println!` macro. Figure 9 shows a program using the "pretty-print" syntax {`:#?`}, which inserts newlines and tab characters to better format structures. The full list of formatting options can be found in the std::fmt documentation.

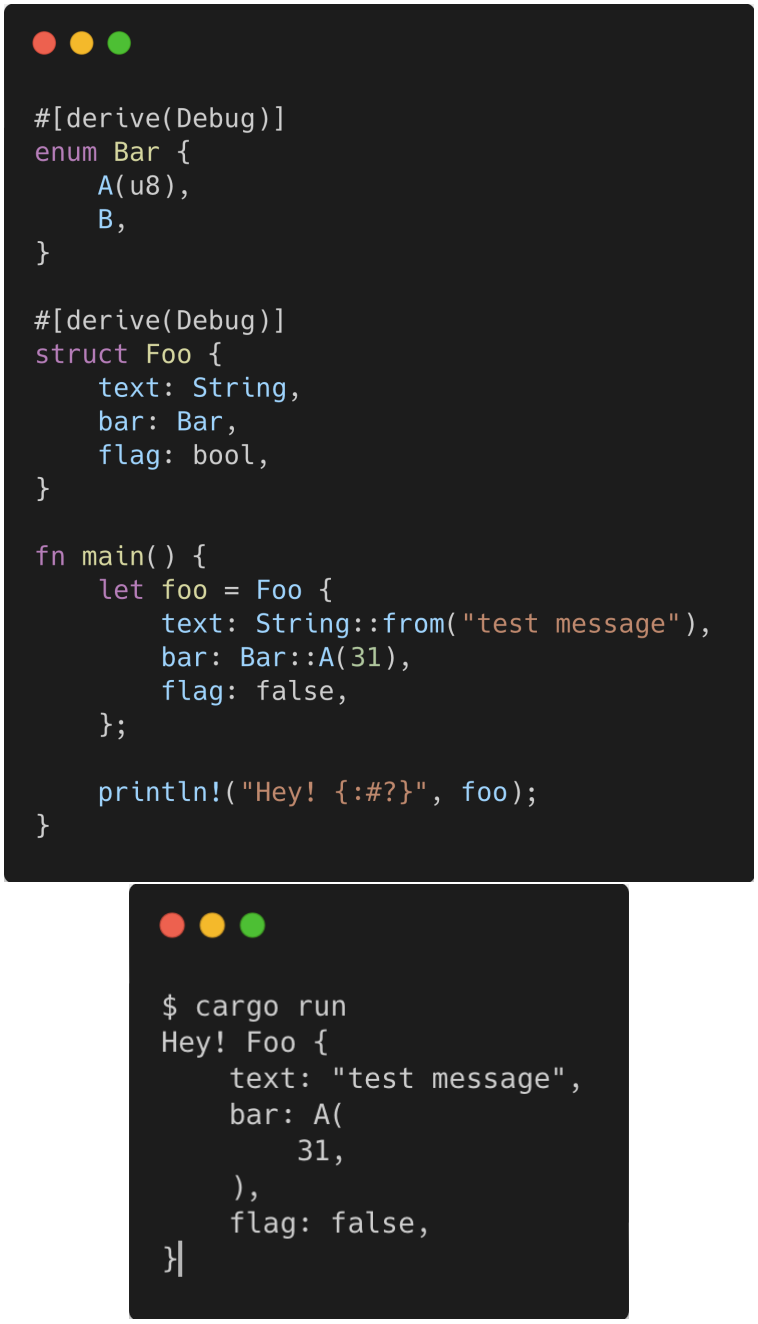
As the `println!` (and associated `print!`) macro is from the Rust standard library and not included in `core` or `alloc` it was re-implemented in this project, writing the formatted output to the console.



```rust
#[derive(Debug)]
enum Bar {
    A(u8),
    B,
}

#[derive(Debug)]
struct Foo {
    text: String,
    bar: Bar,
    flag: bool,
}

fn main() {
    let foo = Foo {
        text: String::from("test message"),
        bar: Bar::A(31),
        flag: false,
    };

    println!("Hey! {:?}", foo);
}
```

```
$ cargo run
Hey! Foo { text: "test message", bar: A(31), flag: false }
```

Figure 8: Rust program demonstrating debug printing and the console output when executed.



```rust
#[derive(Debug)]
enum Bar {
    A(u8),
    B,
}

#[derive(Debug)]
struct Foo {
    text: String,
    bar: Bar,
    flag: bool,
}

fn main() {
    let foo = Foo {
        text: String::from("test message"),
        bar: Bar::A(31),
        flag: false,
    };

    println!("Hey! {:#?}", foo);
}
```

```
$ cargo run
Hey! Foo {
    text: "test message",
    bar: A(
        31,
    ),
    flag: false,
}
```

Figure 9: Rust program demonstrating debug printing and the console output when executed.

Note that the Xen console does not perform carriage returns automatically; a newline character only moves the cursor down vertically. The `println!` macro terminates lines with `\n \r` however internal Rust formatting infrastructure expects newlines to also perform a carriage return. `Debug` printing a large structure with the "pretty-print" feature enabled is hard to read due to lines being cut off.

The `log`[24] crate is the standard logging library maintained by the Rust language team. The `Log` trait is implemented by the console driver and so can be used to output log messages from this project and all its dependencies. The logging verbosity determined by selecting the minimal log level (among `error`, `warn`, `info`, `debug`, and `trace`) at compile time. Compiler optimisations then remove all unused messages to massively reduce the `.text` section size of the emitted image.

9.4.1 Console Performance Improvement

9.5 Memory Management

Implementation found in `stardust::mm` and `xen::memory`.

The virtual addresses in the x86_64 architecture are 48-bits consisting of 4 9-bit indices into the 4 page tree levels, followed by a 12-bit offset into each 4KiB page. Xen needs to present each guest virtual machine with it's own address space and so requires the pseudo-physical layer as shown in Figure 10. In a paravirtualised guest the translation is handled by Xen instead of an emulated memory management unit. This requires issuing `__HYPERVISOR_mmu_update` hypercalls containing an array of machine addresses for page table entires and corresponding new contents of page table entries. The page table is constructed by walking the available memory range and for each page:

1. Look up level 3 entry in level 4 page table, mapping the L3 page frame using an MMU update if it does not exist.
2. Look up level 2 entry in level 3 page table, mapping the L2 page frame using an MMU update if it does not exist.
3. Look up page in level 2 page table, if it does not have the "page present" flag, insert a new `mmu_update` entry (containing the offset into the page table and machine frame number) in the `mmu_updates` array.

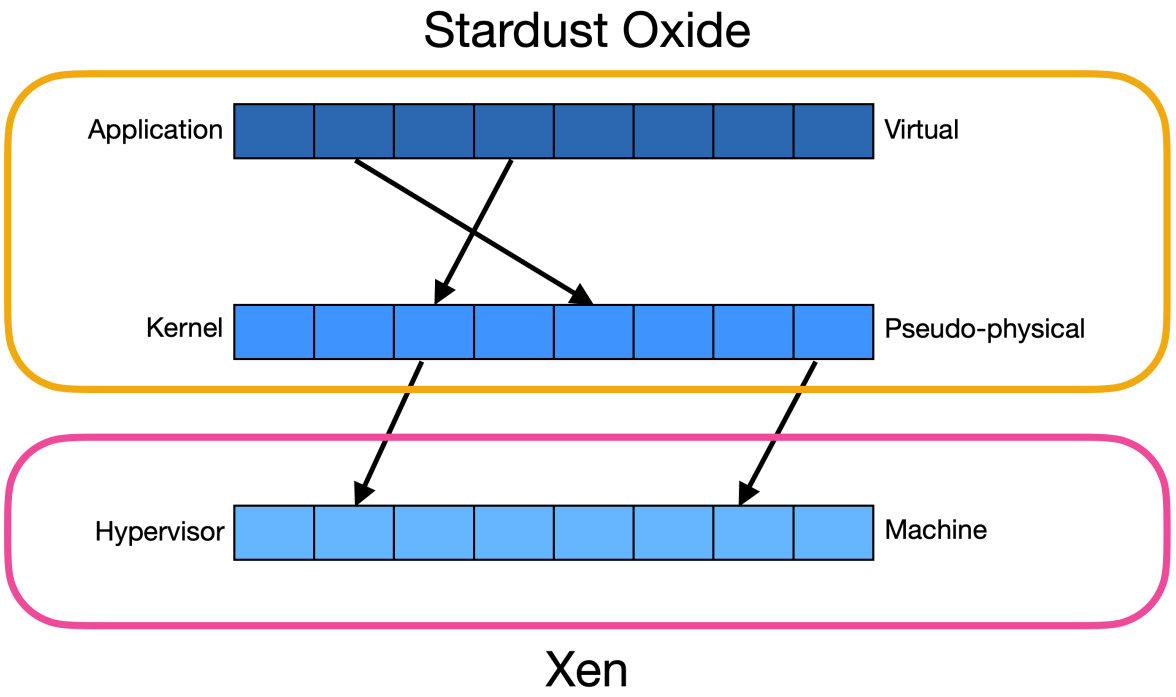


Figure 10: Diagram of memory address spaces in Stardust Oxide and Xen.

If the array is full or the end of the available memory range has been reached, an `__HYPERVISOR_mmu_update` hypercalls is issued. If finished, the start address and length of the mapped memory region are then passed to whichever allocator is being used. The `#[global_allocator]` attribute is used to route all allocation requests to the supplied allocator.

This implementation made use of Rust's ability to assign the value of a block, allowing for the limited scope of temporary variables. This was used for the three separate table lookups making bugs related to the reuse of old data less likely. This is a key distinction from the C implementation which defines all variables at the beginning of the function and has no clear separation between the four steps involved; each lookup follows into the next without comments and using the same variable names for different tables and frame numbers.

9.5.1 Typed Memory Addresses

For primitive numeric types, Rust uses `i` for signed integers, `u` for unsigned, and `f` for floating point, followed by the size in bits. For example, `f64` is a 64-bit floating point number and `i16` is a signed 16-bit integer. `isize` and `usize` are signed and unsigned pointer-sized integers

respectively. This represents a major readability and portability improvement over C's `char`, `short`, `int` and `long` types, which can vary in size between compilers and platforms.

In MiniOS and Stardust the `unsigned long` type is used to store all kinds of memory addresses, from machine frame numbers to physical addresses. There are a series of nested conversion functions, of the form `A_to_B` and some of which use macros internally, to convert between kinds of memory. However this offers no type safety; passing in a machine frame number where a virtual address was expected would result in a runtime error and no warning of the issue at compile time.

Rust structs containing a single field have the same size as the size of the field; there is no overhead. This means adding type safety to memory handling in Stardust Oxide has major safety benefits with no performance cost. Each memory address is given a type defined as a tuple struct around a `usize`, such as `PhysicalAddress(usize);`. The `From` trait (and its reciprocal `Into`) is used for conversions in Rust. These were implemented for all valid conversions between memory types instead of unassociated conversion functions. This makes it impossible to incorrectly convert one type to another, represents another improvement in safety, readability, and developer experience over the C implementation. While `VirtualAddress::from(phys_addr)` is perhaps only slightly more readable than `phys_to_virt(phys_addr)`, it allows functions to be written with generic parameters such as `fn foo<T: Into<VirtualAddress>>(bar: T)`, in which any type which can be converted to a virtual address may be passed to `foo`. This makes functions more composable, reusable, and adaptable as no changes would be required if some other code change resulted in a different type being passed in to `foo`.

Figure 11 shows the embedded ASCII diagram in the documentation showing what types may be converted.

9.5.2 Memory Corruption Bug

Commits preceding 41cbd65 were found to have a bug whereby page table initialisation will fail if more than roughly 100MB of memory is assigned to the virtual machine. The error is shown in Figure 12.

The following steps were undertaken to debug the issue:

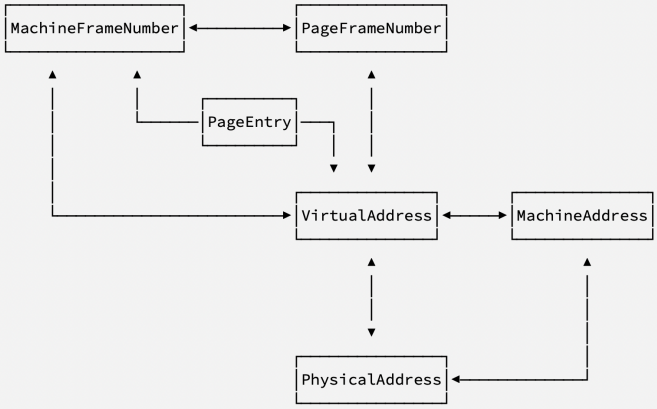
1. The `-16` shown corresponds to the `EBUSY` variant of the `ERRNO` type.

Module xen::memory

source · [-]

[−] Memory utility functions and types

Consists of wrapper types representing different kinds of memory locations. The following diagram describes the conversions between them:



Modules

page_table	Xen Paravirtualized Page Table Interface

Structs

MachineAddress	Machine address
MachineFrameNumber	Number of a page in the machine's address space
PageEntry	Page Entry
PageFrameNumber	Number for page frame
PhysicalAddress	Pseudo-Physical address
TLBFlushFlags	Flags for `update_va_mapping`
VirtualAddress	Virtual address

Figure 11: `rustdoc` page for `xen::memory` module.

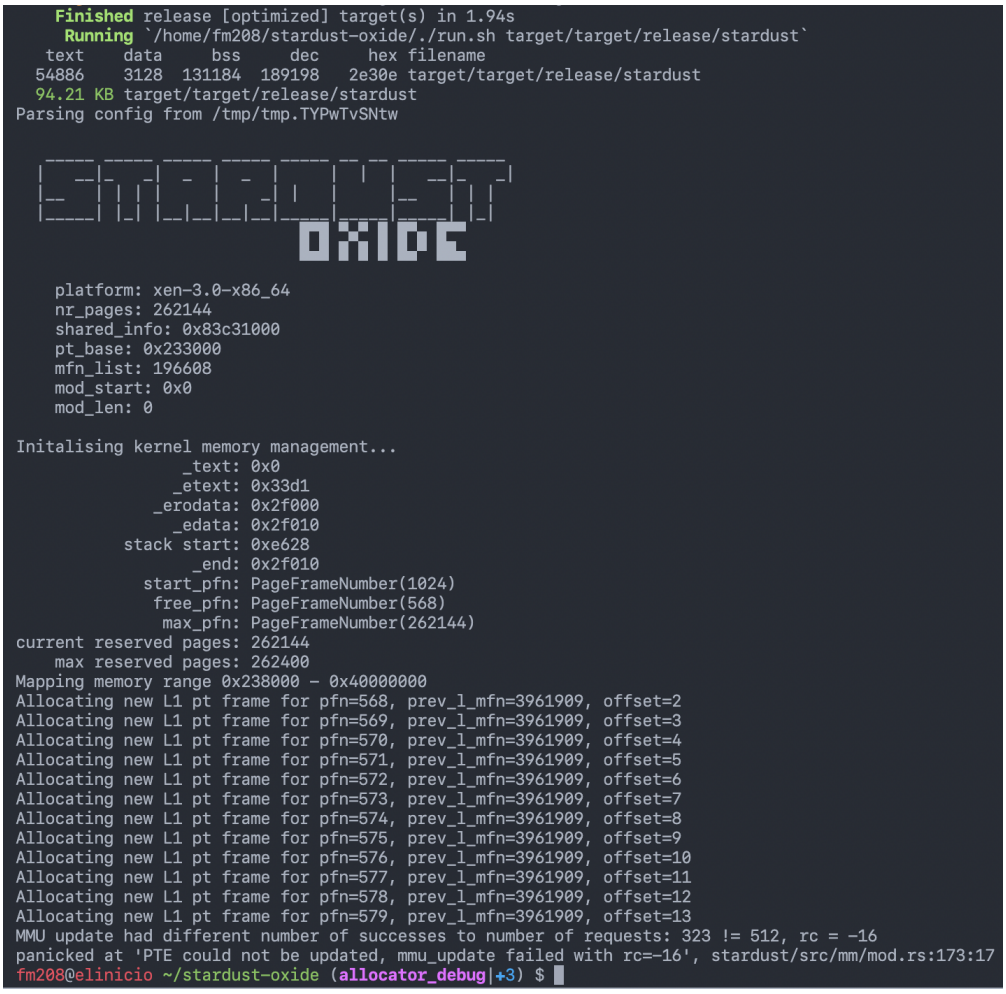


Figure 12: Allocation failure when assigned 128MB of memory.

2. Printing the `mmu_updates` array shows that the first 323 elements appear normally, the `ptr` field increases by 8 bytes each time (as expected for 64-bit pointers) and the `val` field increases by 4096 each time (as expected for 4KiB pages). However the 324th and later elements have "normal" `ptr` fields but the `val` field contains seemingly random[2] values.

3. The value of the `val` field is the value of this expression: `MachineFrameNumber::from(pfn_to_map).0 << PAGE_SHIFT | L1_PROT`. The left shifting and "or"ing with `L1_PROT` explain the high frequency of the value of `val` ending in decimal 27. `pfn_to_map` increments once per loop and so the issue must be in the conversion from `PageFrameNumber` to `MachineFrameNumber`

4. The `impl From<PageFrameNumber> for MachineFrameNumber` body calculates the offset into the `MFN_LIST` array using the supplied page frame number. `MFN_LIST` is an array created by Xen containing mappings where indexing by a page frame number contains the corresponding machine frame number.

5. Dumping `MFN_LIST` reveals it contains normal machine frame numbers up until the 7000 element at which point the contents appears random, this then explains the erroneous values in the `mmu_updates` array.

6. It was discovered that if the stack size is changed, so does the point in the page table creation at which the failure occurs.

7. It was assumed this was evidence of a stack-related issue but had no clear direction in which to continue other than reading the Xen source to determine how `MFN_LIST` could be corrupted or incorrectly generated.

Dr. Spink assisted in debugging and suggest the following changes:

- Do not define the stack as a `u8` array in `xen/src/lib.rs`; this does not guarantee any alignment
- Instead define the stack in the linker script by creating labels at the end of `.bss`
- Alter the bootstrap assembly to place the value of the `__STACK_END` label in `%rsp` using `movq` instead of calculating it using the stack start label and size.

[2]Exact distribution was not analysed

This fixed the error and page table initialisation ran successfully. The cause of the `MFN_LIST` corruption was determined to be due to Xen placing `MFN_LIST` at an address within the `.bss` section. Changing how the stack was defined in the linker script appears to have allowed Xen to correctly detect the end of the program data and place `MFN_LIST` beyond it.

On reflection while this was a challenging issue to solve and took a significant amount of time only to arrive at a very simple solution, it does appear to be an isolated, "one-off" issue. The linker script and bootstrap assembly only need to be written once per project so while this forms a useful learning experience, there is little to apply to the remainder of the software development experience. However, since debugging required to view the same structures in C and Rust to determine at what point does the data differ, it offered a direct comparison between the two languages. Debugging was found to be significantly easier and faster in Rust, mainly due to being able to quickly and easily print large structures using the `Debug` trait. Printing the same in C required manually writing functions to print each field of a structure or element of an array. Additionally, Rust's error types were significantly easier to manage and read than returning a signed integer status as is common in C.

9.6 Time

Implementation found in `xen::platform::time::get_system_time`.

The `get_system_time() -> u64` function retrieves the number of nanoseconds between 1970-01-01 00:00:00 UTC and the current time. This requires reading the time since boot, the processor cycle counter (and values for the shift and multiplication required to convert cycles into nanoseconds), and wall-clock start value seconds and nanoseconds from the `shared_info` struct. However, this must be done in a loop testing that the data is not being updated and the version counter was not modified before and after reading from the struct.

9.7 Scheduling

Implementation found in `stardust::executor`.

Tasks in this implementation are simply `async` functions which return a future containing the unit type (`()`, equivalent to `void` in C). The `Executor` polls tasks in a round-robin schedule until they are completed, although most tasks would be designed to run in a loop

indefinitely. A `core::task::Waker` is a handle for notifying executors when their tasks are ready to be run and should be polled again. The waker supplied to all tasks in this implementation performs a "no-op" and the tasks are polled in a loop; replacing this with an implementation that yields the operating system until an event arrives to wake a task would be more efficient.

9.8 XenStore

Implementation found in `xen::xenstore`.

XenStore is a filesystem-like interface for sharing information between Xen guests. Internally it uses a ring buffer on a shared page and an event channel, both provided in the Xen startup information. The `lazy_static` initialisation pattern is used for the `XenStore` struct. There are private methods for reading and writing byte arrays to the ring buffer, which are used to implement the higher level message transmitting and receiving, which in turn are used for the five public functions:

- `pub fn init()`: initialises the XenStore
- `pub fn write<K: AsRef<str>, V: AsRef<str>>(key: K, value: V)`: writes a key-value pair to the XenStore
- `pub fn read<K: AsRef<str>>(key: K) -> String`: reads a key's value from the XenStore
- `pub fn ls<K: AsRef<str>>(key: K) -> Vec<String>`: lists contents of a directory
- `pub fn domain_id() -> u32`: reads the current domain's ID

Using `AsRef<str>` allows any type that may be taken as reference as a `str` may be supplied: string literals, `str` by value or by reference, and heap allocated `String` by value or by reference.

9.9 XenBus

Implementation found in `xen::xenbus`.

The XenBus a protocol built upon the XenStore mainly used for negotiating connections between the two halves of device drivers.

The `lazy_static` initialisation pattern is used for the `XenBus` struct. It contains a pointer to the `xenstore_domain_interface` type. Responses are held in a `BTreeMap` of `u32` request IDs to a tuple containing the response message header and body. Writing data involves serialising the `MessageHeader` structure which defines the message kind (found in the `MessageKind` enum) and the request and transaction IDs, followed by the byte arrays forming the message body.

The implementation uses an `async` function which is designed to run as a task to asynchronously process message responses, placing them in the B-Tree map. Currently the public function `pub async fn request(kind: MessageKind, data: &[&[u8]], tx_id: u32) -> (MessageHeader, String)` writes the request then blocks in a loop until a response is received. Instead it should register to be awoken when the background task receives its corresponding response. However since the XenBus is only used once during network driver initialisation this work was not prioritised.

9.10 Events

Implementation found in `xen::events`.

When an event arrives, Xen calls the `hypervisor_callback` routine in `bootstrap.S`. The assembly routine after guarding against re-entrant events, calls the `do_hypervisor_callback` function defined in `stardust::trap`. This function resets the event pending flag in the vCPU information structure, calculates which port the event occurred on then calls the event handler. The event handler uses the port to index into the array of event handlers to execute the correct closure.

9.11 Shared Memory

Implementation found in `xen::grant_table`.

Initialising the `GrantTable` struct (again using the same initialisation pattern) calls the platform grant table initialisation function. This builds a page table heirarchy in the grant

table address range which begins after the maximum physical memory address. The page table builder is similar to the main memory one, except instead of mapping machine frame numbers, frames are obtained from the global allocator. `GrantTable` contains a free list of grant reference numbers initialised so that all are available. Granting access and transferring a frame are implemented by writing an entry into the grant table containing the frame, domain, and flags to manage permissions. The four public functions are

- `pub fn init()`: initialises grant table

  `pub fn grant_access(domain: domid_t, frame: MachineFrameNumber, readonly: bool) -> grant_ref_t`: grants the supplied domain access to the supplied frame

  `pub fn grant_transfer(domain: domid_t, frame: MachineFrameNumber) -> grant_ref_t`: transfers the supplied frame to the supplied domain

  `pub fn grant_end(reference: grant_ref_t)`: ends access to the supplied grant reference

Grant mapping is the process for using a frame to which access has been granted. This is implemented using the `GrantHandle` structure. `GrantHandle::new(address: *const u8, reference: u32, domain: domid_t, readonly: bool) -> Result<Self, Error>` maps the frame with the supplied reference at the supplied pointer. The return type is a Result, indicating the function is fallible, but if it succeeds the caller will receive an instance of `GrantHandle`. To unmap a frame the `unmap` method can be called. Crucially, since the `GrantHandle` structure contains private fields, it cannot be instantiated by any users of the `xen` crate: the only way to obtain a handle is by successfully mapping a frame. This means that unmap will never fail as it cannot be called on an invalid reference. This is an example of the ability to create misuse-resistant APIs in Rust; the equivalent would be impossible in C due to the lack of privacy on struct fields.

9.12 Ring Buffers

Implementation found in `stardust::net::ring`.

There are two shared ring buffer types in the network interface for Xen, `netif_tx_sring` and `netif_rx_sring`. These types are generated from the `DEFINE_RING_TYPES` macro which is being used as a form of C preprocessor-based monomorphisation. In Rust this can be

replaced with a more robust implementation using traits and compiler monomorphisation. However, there is the caveat that the driver backend must be givena access to the page the shared ring resides in, meaning the raw pointer must be kept even though the ring being pointed to may be one of several types and whose elements may also be distinct types. For this reason a ring wrapper structure cannot simply have a generic field to store the underlying shared ring.

Instead the solution implemented uses a common `Ring` struct, containing a mutable reference with a static lifetime to a generic `S: RawRing`. `RawRing` is a trait implemented by both `netif_tx_sring` and `netif_rx_sring`. The trait uses an associated type called `Element`, and a pointer to an `Element` is the return type of the `get` method (`fn get(&mut self, index: usize) -> *mut Self::Element;`). This allows the transmit and receive rings to contain different element types.

The allocation for the page occurs in the `RawRing::new` method, which returns a mutable pointer to `Self`. The `Drop` trait is implemented for `Ring` so that if the `Ring` ever leaves scope the memory is not leaked.

9.13 Networking

Implementation found in `stardust::net::phy`.

The Xen virtual network interface driver uses two ring buffers of packets, one for incoming and one for outgoing. Each packet is placed in a shared page and each ring buffer contains request or response structs for each packet being sent or received, containing the page number, offset, and length. The configured event channel is used by each half of the driver to indicate to the other when there is processing to be done.

The driver is initialised by Stardust Oxide with the following steps:

1. Setup free list of transmission packet buffers
2. Setup arrays of page number-grant reference pairs.
3. Allocate and enable event channel.
4. Allocate both ring buffers.

5. Share ring buffer pages and obtain grant reference for each.
6. Fill receive ring buffer with shared, allocated pages, storing the grant references in the arrays setup in step 1.
7. Initiate XenBus transaction: set transmit and receive ring grant references, event channel, and switch from `Connecting` to `Connected` state.

There exists Xen bug in outbound packet checksum offloading, affecting both MiniOS and Stardust Oxide, and is fixed by running `ethtool -K vifXXX.0 tx off` where `XXX` is the virtual interface number.

9.13.1 Grant Table Initialisation Bug

After implemented network device initialisation there were no network events corresponding to interface initialisation, despite the MiniOS receiving events after startup. The following steps were then undertaken to debug this issue.

1. Testing whether events are registered using `schedule(Block)`, which yields the domain until an event is ready to be processed. This caused an immediate termination of the guest. The reason being that if a guest has no registered event handlers then instead of yielding indefinitely Xen terminates instead.
2. After using the MiniOS assembly routines for event handling the event handler function was then logging a very large number of events. This was due to not applying a mask on event channels and these were console events. Since each was causing data to be printed to the console this was causing a feedback loop. Correctly masking the events showed that network events were still not arriving.
3. Network packet receiving was implemented in the hope that asynchronous processing with events could be implemented later, however the producer index of the ring buffer was not being incremented like in MiniOS (even when events were disabled in MiniOS).
4. Network packet transmission was then implemented but the consumer index not being updated by Xen, and `ifconfig` was showing no traffic on the interface.
5. The `xenstore-ls` tool shows no difference between MiniOS and Stardust Oxide virtual network interface configuration, however it does reveal several hundred Xen errors that

had previously gone unnoticed. The error messages were `1 mapping shared-frames 2047/2046 port tx 4 rx 4`. This message corresponds to Stardust Oxide logs indicating the mapped shared frame numbers of 2047 for the transmission ring buffer and 2046 for the reception ring buffer, on event channel number 4. The `1` at the beginning of the message was believed to be a Xen `errnoval` corresponding to `EPERM`, operation not permitted.

6. These messages were also present in `/var/log/kern.log` and by comparing timestamps it was determined the error was occuring after negotiating the device status over XenBus, not while updating the grant table.

7. Grant tables were re-implemented almost entirely, switching from the on-demand page method used in MiniOS to the page table pointing to allocated pages method used in Stardust, but the issue persisted.

8. If invalid grant reference numbers are supplied the error number changes from 1 to 3, which corresponds to "No such process". However if the read only flag is set or invalid domain or frame numbers are supplied, the error stays as 1, suggesting that any of the three values being set in the grant table could be the issue.

9. Reading through Linux XenBus drivers the line that is emitting the error in the kernel log can be seen at `xenbus_client.c`, line 524.

10. Error was previously misidentified as `errnoval` but is instead `GNTST_`, a grant table status type, where -3 corresponds to "bad grant reference" which is expected when an invalid reference is passed, but the -1 observed is a "general error".

11. There were several locations in the Xen source for handling grant table hypercalls where general errors are returned. After recompiling Xen with debug logs enabled to determine which is the true source of the error, during a debugging session Dr Spink noticed a misuse of pointers in the section of code which writes entries to the grant table. The dereference was not occurring on the left-side of the expression so instead of modifying the table entry, a local variable was being modified then discarded.

10 Evaluation and Critical Appraisal

10.1 Compatibility

The Definitive Guide To The Xen Hypervisor[7] claimed that paravirtualisation was the most performant of all the virtualisation modes offered by Xen. While this was correct at the time of publishing, since 2008 and the introduction of hardware extensions to accelerate virtualisation have meant that emulating a complete hardware interface is more performant that a paravirtualised interface[32].

Not only would have writing a "bare-metal" x86_64 operating system been faster, but there are far more learning resources available. An overarching difficulty throughout this project was the lack of documentation and few resources for understanding how certain features should be implemented. Furthermore, a "bare-metal" operating system would be compatible with many hypervisors, not just Xen.

10.2 Stardust Comparison

Stardust Oxide has almost reached feature parity with Stardust but also has several features not present in Stardust. Stardust has block device and file system support, as well as support for multicore processing and preemptive multitasking. However Stardust Oxide supports networking and cooperative multitasking.

10.3 Community Feedback

Figure 13 shows a selection of comments on a post in a Rust forum about the project. Feedback was generally positive with no major criticisms made, only questions about how the project was undertaken. The third comment in the figure was particularly rewarding as encouraging more research and work in the field of operating system development was a key goal.

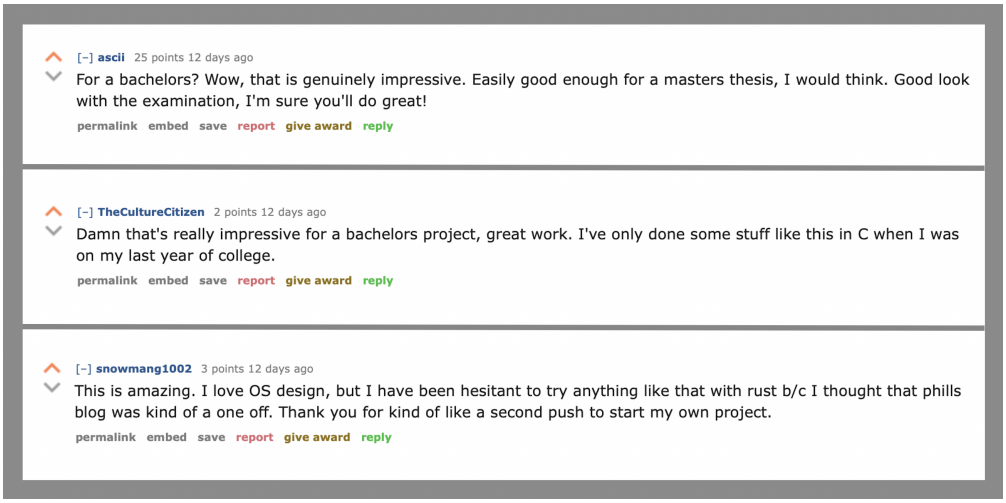


Figure 13: Selected comments on post about Stardust Oxide.

11 Conclusion

This project consisted of the implementation of a unikernel operating system in Rust. The benefits and feasibility of writing systems software in Rust were demonstrated and specific software patterns that improve upon standard practice in C or C++ operating system development were highlighted.

The primary achievement is implementing an operating system for which useful programs that do interesting things can be written. In its current state it serves as a useful reference for anyone writing their own Xen operating system, adding support for Xen to an existing operating system, or other operating system research. Another achievement is the breadth of features implemented: events, shared memory (both initiator and target "sides"), multitasking, networking, XenBus, XenStore, and time management.

The main drawback is that it does not support running in a "bare-metal" x86_64 environment, only Xen paravirtualisation, which prevents compatibility with other hypervisors and running directly on an x86_64 host. It is also difficult for users to write efficient event based software; there would need to be improvements to the the `async` executor, with a system for calling `Waker`s from event handlers automatically when registered by a future.

There are many future directions to take this work. These include adding the previously discussed support for non-paravirtualised x86_64 (and therefore the Xen PVHVM mode), supporting multiple cores for parallel (and not merely concurrent) task execution, ARM64

platform support, and implementing interfaces for the VirtIO collection of standardized virtual device drivers. This project was written with the priority placed on reaching the desired scope; there are certainly readability, performance, and efficiency improvements to the codebase that could be achieved with a refactor.

12 Appendix

12.1 Ethics Self Assessment Form

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.
Tick one box

- ☐ **Staff Project**
- ☐ **Postgraduate Project**
- ☒ **Undergraduate Project**

Title of project

Stardust Oxide: Unikernel single-address space OS

Name of researcher(s)

Ferdia McKeogh

Name of supervisor (for student research)

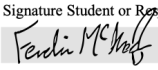
Alan Dearle

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted **YES** ☒ **NO** ☐

There are no ethical issues raised by this project

Signature Student or Researcher



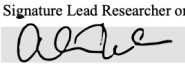
Print Name

Ferdia McKeogh

Date

2021-09-16

Signature Lead Researcher or Supervisor



Print Name

Alan Dearle

Date

16/9/21

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms. The School Ethics Committee will be responsible for monitoring assessments.

Computer Science Preliminary Ethics Self-Assessment Form

Research with human subjects

Does your research involve human subjects or have potential adverse consequences for human welfare and wellbeing?

YES ☐ **NO** ☒

If YES, full ethics review required
For example:
Will you be surveying, observing or interviewing human subjects?
Will you be analysing secondary data that could significantly affect human subjects?
Does your research have the potential to have a significant negative effect on people in the study area?

Potential physical or psychological harm, discomfort or stress

Are there any foreseeable risks to the researcher, or to any participants in this research?

YES ☐ **NO** ☒

If YES, full ethics review required
For example:
Is there any potential that there could be physical harm for anyone involved in the research?
Is there any potential for psychological harm, discomfort or stress for anyone involved in the research?

Conflicts of interest

Do any conflicts of interest arise?

YES ☐ **NO** ☒

If YES, full ethics review required
For example:
Might research objectivity be compromised by sponsorship?
Might any issues of intellectual property or roles in research be raised?

Funding

Is your research funded externally?

YES ☐ **NO** ☒

If YES, does the funder appear on the 'currently automatically approved' list on the UTREC website?

YES ☐ **NO** ☐

If NO, you will need to submit a Funding Approval Application as per instructions on the UTREC website.

Research with animals

Does your research involve the use of living animals?

YES ☐ **NO** ☒

If YES, your proposal must be referred to the University's Animal Welfare and Ethics Committee (AWEC)

University Teaching and Research Ethics Committee (UTREC) pages
http://www.st-andrews.ac.uk/utrec/

References

[1] Dave Astels. *Test Driven Development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003. ISBN: 0131016490.

[2] *Async-await on stable Rust! — Rust Blog*. en. URL: https://blog.rust-lang.org/2019/11/07/Async-await-stable.html (visited on 03/22/2022).

[3] Paul Barham et al. "Xen and the Art of Virtualization". In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 164–177. ISSN: 0163-5980. DOI: 10.1145/1165389.945462. URL: https://doi.org/10.1145/1165389.945462.

[4] *bindgen*. original-date: 2016-06-22T15:05:51Z. Mar. 2022. URL: https://github.com/rust-lang/rust-bindgen (visited on 03/26/2022).

[5] Jenny Blessing, Michael A. Specter, and Daniel J. Weitzner. *You Really Shouldn't Roll Your Own Crypto: An Empirical Study of Vulnerabilities in Cryptographic Libraries*. 2021. arXiv: 2107.04940 [cs.CR].

[6] *buddy_system_allocator*. original-date: 2019-03-17T17:54:28Z. Mar. 2022. URL: https://github.com/rcore-os/buddy_system_allocator (visited on 03/22/2022).

[7] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. First. USA: Prentice Hall Press, 2007. ISBN: 9780132349710.

[8] IncludeOS Contributors. *IncludeOS*. URL: https://github.com/includeos/IncludeOS.

[9] Xen Contributors. *Mini-OS*. URL: http://xenbits.xen.org/gitweb/?p=mini-os.git;a=summary.

[10] Alex Crichton. *cc-rs*. original-date: 2014-11-01T02:21:22Z. Mar. 2022. URL: https://github.com/alexcrichton/cc-rs (visited on 03/26/2022).

[11] *Drop in std::ops - Rust*. URL: https://doc.rust-lang.org/std/ops/trait.Drop.html (visited on 03/27/2022).

[12] Jason Evans. "A Scalable Concurrent malloc(3) Implementation for FreeBSD". en. In: (), p. 14.

[13] Paul Gazzillo and Shiyi Wei. "Conditional Compilation is Dead, Long Live Conditional Compilation!" en. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. Montreal, QC, Canada: IEEE, May 2019, pp. 105–108. ISBN: 978-1-72811-758-4. DOI: 10.1109/ICSE-NIER.2019.00035. URL: https://ieeexplore.ieee.org/document/8805666/ (visited on 03/27/2022).

[14] *GitHub Actions Documentation*. en. URL: https://docs.github.com/en/actions (visited on 03/28/2022).

[15] W Jaradat. "Towards Unikernel Support for Distributed Microservices". In: *Adobe Tech Summit* (2019).

[16] W Jaradat, A Dearle, and J Lewis. "Unikernel support for the deployment of light-weight, self-contained, and latency avoiding services". In: *Third Annual UK System Research Challenges Workshop* (2018).

[17] W. Jaradat, A. Dearle, and J. Lewis. "Unikernel Support for Lambda Functions". In: *Fifth Annual UK System Research Challenges Workshop* (2020).

[18] Kenneth C. Knowlton. "A Fast Storage Allocator". In: *Commun. ACM* 8.10 (1965), pp. 623–624. ISSN: 0001-0782. DOI: 10.1145/365628.365655. URL: https://doi.org/10.1145/365628.365655.

[19] Markus Kohlhase. *Rust web framework comparison*. original-date: 2015-08-16T11:42:44Z. Mar. 2022. URL: https://github.com/flosse/rust-web-framework-comparison (visited on 03/22/2022).

[20] Simon Kuenzer et al. "Unikraft". In: *Proceedings of the Sixteenth European Conference on Computer Systems* (2021). DOI: 10.1145/3447786.3456248. URL: http://dx.doi.org/10.1145/3447786.3456248.

[21] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: San Jose, CA, USA, 2004, pp. 75–88.

[22] *lazy_static - Rust*. URL: https://docs.rs/lazy_static/latest/lazy_static/ (visited on 03/26/2022).

[23] Yi Lin et al. "Rust as a Language for High Performance GC Implementation". In: *SIGPLAN Not.* 51.11 (2016), pp. 89–98. ISSN: 0362-1340. DOI: 10.1145/3241624.2926707. URL: https://doi.org/10.1145/3241624.2926707.

[24] *log*. original-date: 2014-12-13T21:45:04Z. Mar. 2022. URL: https://github.com/rust-lang/log (visited on 03/28/2022).

[25] Anil Madhavapeddy and David J. Scott. "Unikernels: Rise of the Virtual Library Operating System: What If All the Software Layers in a Virtual Appliance Were Compiled within the Same Safe, High-Level Language Framework?" In: *Queue* 11.11 (2013), pp. 30–44. ISSN: 1542-7730. DOI: 10.1145/2557963.2566628. URL: https://doi.org/10.1145/2557963.2566628.

[26] Anil Madhavapeddy et al. “Unikernels: Library Operating Systems for the Cloud”. In: *SIGPLAN Not.* 48.4 (Mar. 2013), pp. 461–472. ISSN: 0362-1340. DOI: 10.1145/2499368.2451167. URL: https://doi.org/10.1145/2499368.2451167.

[27] Merriam-Webster. *OPERATING SYSTEM*. URL: https://www.merriam-webster.com/dictionary/operating+system.

[28] *musl libc*. URL: https://musl.libc.org/ (visited on 04/01/2022).

[29] *Mutex in std::sync - Rust*. URL: https://doc.rust-lang.org/std/sync/struct.Mutex.html (visited on 03/26/2022).

[30] Philipp Oppermann. *linked-list-allocator*. original-date: 2016-01-19T17:46:59Z. Mar. 2022. URL: https://github.com/phil-opp/linked-list-allocator (visited on 03/22/2022).

[31] *Rust Programming Language*. en-US. URL: https://www.rust-lang.org/ (visited on 03/21/2022).

[32] Adam Schwalm. “Benchmarking Xen Virtualisation”. In: (). URL: https://www.starlab.io/blog/benchmarking-xen-virtualization.

[33] *smoltcp*. original-date: 2016-12-02T15:21:02Z. Mar. 2022. URL: https://github.com/smoltcp-rs/smoltcp (visited on 03/22/2022).

[34] *Stack Overflow Developer Survey 2021*. en. URL: https://insights.stackoverflow.com/survey/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2021 (visited on 03/21/2022).

[35] *std::mutex - cppreference.com*. URL: https://en.cppreference.com/w/cpp/thread/mutex (visited on 03/26/2022).

[36] Chromium Team. *Chromium Security: Memory Safety*. URL: https://www.chromium.org/Home/chromium-security/memory-safety/.

[37] *Unstable Features - The Cargo Book*. URL: https://doc.rust-lang.org/cargo/reference/unstable.html#build-std (visited on 03/25/2022).

[38] *What is rustdoc? - The rustdoc book*. URL: https://doc.rust-lang.org/rustdoc/what-is-rustdoc.html (visited on 03/28/2022).

[39] *Xen Project Software Overview - Xen*. URL: https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview (visited on 03/22/2022).