Enhancing Rust Support in the Nautilus Aerokernel

Matthew Sinclair
matthewsinclair2024@u.northwestern.edu
Northwestern University
Evanston, Illinois, USA

Conor Kotwasinski
conorkotwasinski2024@u.northwestern.edu
Northwestern University
Evanston, Illinois, USA

Ian Armstrong
ianarmstrong2022@u.northwestern.edu
Northwestern University
Evanston, Illinois, USA

Charles Zhou
charleszhou2023@u.northwestern.edu
Northwestern University
Evanston, Illinois, USA

ABSTRACT

Expanding upon previous efforts, we present a framework for writing components of the Nautilus kernel in the Rust programming language. We provide wrappers around core parts of Nautilus, like the character device and GPU device subsystems, allowing Rust drivers to integrate with the existing C software stack. We bring first-class concurrency support to Rust modules in Nautilus, providing a registration API for interrupt handlers, an asynchronous executor for cooperative multitasking, and an idiomatic interface to native Nautilus threads. Our API leverages Rust's strong type system and borrow checker to prevent many common but subtle sources of undefined behavior, including use-after-free bugs and data races. We showcase our framework with highly concurrent shell programs and with drivers for two devices–a parallel port and a Virtio GPU. We demonstrate the effectiveness of the latter driver by porting the original DOOM engine to Nautilus and running DOOM in the shell.

1 INTRODUCTION

Low-level software development has long been plagued with a class of bugs deriving from undefined behavior, including buffer overflows, null-pointer dereferences, use-after-frees, and data races. These bugs are apparently the price that must be paid for performance or low-level control of programs. Certain settings, including embedded and kernel development, may entirely preclude the possibility of working in a safer, garbage-collected programming language. While languages like C++, through RAII and smart pointers and an urge to adhere to core guidelines, have made improvements in this area, memory safety and concurrency bugs have nevertheless remained a major issue. These bugs can be particularly severe in kernels—the most popular still largely written in C—where undefined behavior may introduce serious security vulnerabilities. In fact, Microsoft estimates that 70 percent of their vulnerabilities from 2006 to 2018 were caused by memory safety issues [10]. These vulnerabilities have led to a growing interest in Rust.

Rust is a modern systems programming language, created at Mozilla in 2006 [1], and has seen a surge in popularity in recent years due to its emphasis on performance and low-level control while maintaining strong safety guarantees. Rust's distinctive model of ownership and lifetimes rule out many major sources of undefined behavior at compile time, and the memory- and thread-safety guarantees the compiler provides have caught the attention of kernel developers. Rust became the first language offered as an alternative to C in the Linux kernel in late 2022, when support for basic components was merged [9], and Rust code appeared in the Windows kernel in early 2023 [16]. Even the National Security Agency expressed support for Rust in an advisory issued in 2022, leaving C and C++ out of the list of advised languages [2].

The benefits, then, of bringing Rust support to Nautilus—a research kernel developed by Northwestern University and the Illinois Institute of Technology [7]—are clear. A framework for Rust components in the kernel would allow new drivers to be written with more assurance that memory safety and concurrency invariants are being upheld, with less time spent debugging subtle errors due to undefined behavior, and with access to the features of a more modern programming language.

2 PREVIOUS WORK

Previous groups have worked on Rust in the Nautilus kernel. In 2019, a proof-of-concept example showed that Rust functions could be called from C in a simple shell command. Rust was not as mature in 2019 as it is today, especially in environments without the standard library, but the basic possibility of using Rust in Nautilus had been showcased. [8]

A separate project by Qingwei Lan, Hanming Wang, and Michael Polinski from Northwestern University expanded Rust's capabilities within the Nautilus kernel by implementing a parallel port driver. This served as a demonstration of Rust's features for kernel programming, including the writing of kernel modules like device drivers. This group also properly integrated Rust code into Nautilus' build system without the need for linking hacks. We build directly off their work. [12]

3 THE RUST PROGRAMMING LANGUAGE

The Rust Programming Language is a systems programming language with an emphasis on safety and performance. Noted for its reliability and efficiency, Rust boasts a rich type system and ownership model to maintain memory safety, a robust compiler, and a powerful dependency management system. A brief discussion of some key features follows [14].

3.0.1 Ownership. Rust’s unique ownership model, unlike Java’s automatic garbage collection or C’s explicit memory allocation, manages memory by assigning ownership of a value to a variable, which is responsible for that value’s memory. The compiler enforces two rules: a value has only one owner at a time; the value is de-allocated or “dropped" when the owning variable goes out of scope. Ownership transfers occur during an assignment or when passing function parameters. Rather than performing costly data copies, Rust simply transfers ownership from one variable to another, rendering the previous owner invalid, thereby preventing double freeing, a potential source of memory corruption and security vulnerabilities. Shallow copying is preferred, moving a value’s pointer and metadata from the stack. While Rust does not automatically deep copy heap data, it accommodates this through the `clone` method. Simple scalar primitives, or values that implement the `Copy` trait, are an exception. As these entities are of fixed size and stored on the stack, there is no difference between a deep or shallow copy, nor any requirement to invalidate the previous owner.

3.0.2 Referencing. In Rust, references are indicated with &. “Borrowing" a value in this way allows variables to refer to data without taking ownership, mitigating ownership transfer and scoping drops.

```rust
fn main() {
    let mut s = String::from("hello");
    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

In this example, we create a mutable reference to a String, `&s`, and append “world" via a function accepting such a reference. The Rust compiler allows only one mutable reference per scope, and guarantees that data will not be dropped prematurely. Therefore, unlike common sources of errors in C, Rust prevents null references or dangling pointers by nature.

3.0.3 Unsafe Rust. Despite compile-time memory safety guarantees, Rust permits C-like manipulation of raw pointers using the `unsafe` keyword. Normally, Rust’s compiler enforces ownership and borrowing rules, ensuring references always point to valid memory, cannot be null, and are automatically cleaned. But in an `unsafe` block, raw pointers may be directly dereferenced, bypassing these safety measures.

```rust
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1, r2 : {}, {}", *r1, *r2);
}
```

Raw pointers, are legal in safe code (as in `r1` and `r2`), but dereferencing them is not. Note that we created two references, one mutable and one immutable, to the same value, introducing the risk of data races.

Moreover, the `unsafe` keyword is particularly useful when interacting with code written in another language. Rust’s Foreign Function Interface (FFI) provides such a mechanism using the `extern` keyword, followed by the desired application binary interface (ABI) definition.

```rust
extern "C" {
 fn abs(input: i32) -> i32;
}

fn main() {
 unsafe {
  println!("abs of -3 according to C: {}",
            abs(-3));
 }
}
```

In this example, we declare and call the `abs` function from the C standard library in Rust. Declarations within the `extern` block are always unsafe, and their use again requires the keyword. The `"C"` signature indicates to the compiler which ABI to use and, in this case, follows the C programming language’s ABI. Note that the `extern` keyword can be used in the opposite direction, meaning Rust functions can be called from C.

3.0.4 Error Handling. Rust’s strong typing system ensures errors must be handled appropriately, avoiding the concept of exceptions in other languages.

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

The `Result` enum is generic over types `T` and `E`, where the former represents the type within `Ok(T)` on success, and the latter represents the type within `Err(E)` on failure. Associated helper functions like `unwrap` and `expect` provide shortcuts and customization that simplify error handling. Furthermore, error propagation is supported via the ? operator. When called on a `Result` type, this operator will unwrap the inner type, and return `E` to the calling code immediately. As a result, the ? operator makes error propagation trivial, and even allows for conversions between the error types.

3.0.5 Send and Sync. Rust's core library houses most of its concurrency features. Two relevant concepts, Send and Sync, provide options to handle concurrency. `Send` is a marker trait signifying that a type's value can transfer ownership between threads. Most Rust types, except raw pointers, implement `Send`, and types composed of other `Send` types are automatically `Send`.

```
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Rc::new(Mutex::new(0));
    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            // some work here
        });
    }
}
```

In the above example, we attempt to move a cloned reference counter Rc<T> between threads. Because Rc<T> is not Send, the code will not compile. Prohibiting ownership transfers of the cloned reference prevents two threads from updating the count simultaneously.

`Sync` is another maker trait signifying a type may be referenced from multiple threads. Specifically, if an immutable reference to a given type is Send, then that type is also Sync. As before, primitives are Sync, and types composed of Sync types are automatically Sync. See Section 5 for more.

4 RUST FRAMEWORK

4.1 Project Structure

We separate wrappers around core parts of the Nautilus kernel from the drivers and modules which use them. These wrappers can be found in the `src/kernel` folder under `src/rust`. Modules for shell commands and device drivers live under `src/rust/src` in sibling folders to `kernel`. This separation is essential—wrapping Nautilus' C subsystems introduces opportunities for bugs due to translation. By keeping the core wrappers minimal, documented, and in one place, we reduce the potential bug surface due to interaction with C.

4.2 C Interoperability

Rust programs can call C functions through FFI. As we are not rewriting Nautilus in Rust, but rather extending it to support Rust, we must have a way for Rust modules to call into C subsystems. FFI signatures can be generated automatically with `bindgen` or by hand. Wrapping these bindings into idiomatic Rust APIs is the purpose of the previously mentioned `kernel` library.

4.2.1 bindgen. `bindgen` is a tool that automatically generates Rust FFI bindings to C libraries. We use `bindgen` to generate the appropriate Rust function signatures from C headers for core parts of Nautilus. The bindings can be used within `unsafe` code to call functions like `kmem_malloc`, `nk_thread_create`, `register_irq_handler`, etc. without implementing them from scratch. [3]

4.2.2 Manual bindings. Nautilus' C code also makes heavy use of procedural macros and static inline functions, which do not live behind any symbol after compilation. Without a symbol to link to, direct FFI bindings for these macros and functions is impossible. The solution is to manually wrap them with C glue code—effectively creating a function with the sole purpose of calling the macro or inlined function. This C wrapper function can then be called from Rust.

4.3 Hooking into KConfig

Supporting conditional compilation based on KConfig options is necessary for Rust modules to be first-class citizens of the Nautilus code base. Unfortunately, stable Rust has no direct analogue to an `#ifdef` in the C preprocessor language. To get this functionality, we use the feature `cfg_accessible` from RFC 2523, which is yet to be stabilized [4]. This feature allows us to compile code depending on KConfig parameters.

4.4 Building the project

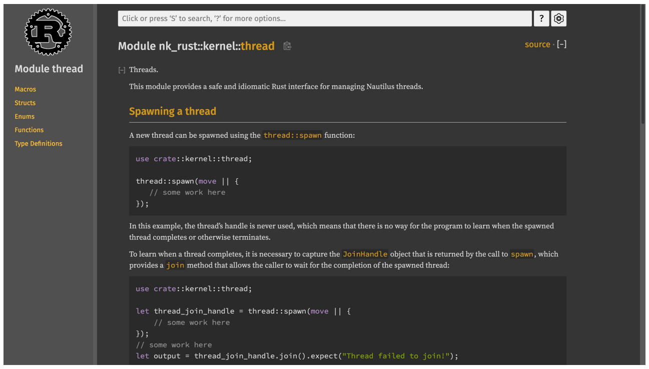
Rust modules are managed by `Cargo`, Rust's build system and package manager. Thanks to last year's effort, building with `Cargo` has already been integrated into Nautilus' `Makefiles`. However, a recent change to Rust's compiler built-ins led to multiple definitions of the `fmod` symbol when linking with C. We are forced to strip the symbol from the Rust object file to fix this. See `src/rust/fmod_hack.sh` for more details.

With this fixed, building Rust modules is as easy as enabling Rust support in `menuconfig` and building the kernel as usual.

4.5 Documentation

To assist future works, we document our code thoroughly. Documentation can be built with `rustdoc` and opened in a web browser. Simply run the following command in the shell while in the `src/rust` directory:

```
$ cargo doc --document-private-items --open
```



Module nk_rust::kernel::thread

Threads.

This module provides a safe and idiomatic Rust interface for managing Nautilus threads.

Spawning a thread

A new thread can be spawned using the `thread::spawn` function:

```
use crate::kernel::thread;

thread::spawn(move || {
    // some work here
});
```

In this example, the thread's handle is never used, which means that there is no way for the program to learn when the spawned thread completes or otherwise terminates.

To learn when a thread completes, it is necessary to capture the `JoinHandle` object that is returned by the call to `spawn`, which provides a `join` method that allows the caller to wait for the completion of the spawned thread:

```
use crate::kernel::thread;

let thread_join_handle = thread::spawn(move || {
    // some work here
});
// some work here
let output = thread_join_handle.join().expect("Thread failed to join!");
```

Figure 1: Documentation for Rust APIs

5 CONCURRENCY SUPPORT

Concurrency, tough as it can often be to get right, is an unavoidable reality of complex programs. This is no less true in the context of kernel development, where interrupts may make even single-threaded code concurrent. Nautilus has support for interrupt handling and preemptive multitasking with threads, providing synchronization primitives for shared state and message passing. We provide idiomatic Rust interfaces to Nautilus' interrupt request subsystem and to native Nautilus threads. Furthermore, we introduce an asynchronous executor into Nautilus, allowing for cooperative multitasking.

5.1 Interrupts

Interrupt requests (IRQs) are signals raised externally by the chipset, telling the CPU to stop its current task and react to the state of some device [18]. A keyboard, for example, may send an IRQ to notify the CPU when a key has been pressed. Because the CPU suspends normal tasks to handle interrupts, IRQs are a source of concurrency in addition to being a central part of communication with many devices.

In Nautilus, `irq.h` provides the C interface for IRQ handling. IRQ handlers may be registered with `register_irq_handler`, and IRQs can be unmasked (listened to) with `nk_unmask_irq` and masked (ignored) with `nk_mask_irq`. Following the design of Rust for Linux [5], we use RAII to wrap these functions, binding the management of IRQ registration and masking to the lifetime of an object.

The `irq::Handler` trait and the `irq::Registration` struct (defined in `irq.rs`) are the foundation of IRQ handling in our Rust framework. `Handler` is defined as follows:

```
pub trait Handler {
  type State: Send + Sync;
  fn handle_irq(data: &Self::State) -> Result;
}
```

To register an interrupt handler, one must implement `irq::Handler` for the appropriate type. For example:

```
struct Foo(i32);

impl irq::Handler for Foo {
  type State = Spinlock<Foo>;

  fn handle_irq(foo: &Self::State) -> Result {
   debug!("foo = {}", foo.lock().0);
   Ok(())
  }
}
```

This `impl` does not actually register the interrupt handler. After (and only after—the code will not compile if the `impl` is forgotten) implementing the `Handler` trait, `irq::Registration::try_new` can be called for the corresponding type with the desired interrupt vector, in order to do the handler registration.

This simple example illustrates multiple key ways our interface differs from the C API.

(1) The call to `apic_do_eoi`, which alerts the APIC that we are done handling the interrupt, is implicitly inserted after the handler returns. A handler cannot block future interrupts by forgetting to alert the APIC that it has finished.
(2) The calls that manage the life cycle of an interrupt handler are bound to an instance of `irq::Registration`. When the registration is created, the IRQ handler is registered and the associated interrupt is unmasked. When the registration is dropped, the interrupt is masked.
(3) The handler function knows the type of its state: it is not a `void*`. Thanks to generics, the handler's signature is more concrete, and bugs due to miscasts in the function body are avoided.
(4) Data races on state shared with an interrupted handler are eliminated at compile time. The `Send+Sync` bound on `State` in `Handler` requires that a concurrency-safe type be used for shared state. If, for example, we forgot

to guard the counter with an appropriate type like a Spinlock the compiler would catch the error.[1]

```
error[E0277]: `RefCell<Foo>` cannot be
shared between threads safely
  --> src/foo/mod.rs:5:18
   |
61 |     type State = RefCell<Foo>;
   |                  ^^^^^^^^^^^^
   | `RefCell<Foo>` cannot be shared
   | between threads safely
   |
  = help: the trait `Sync` is not
    implemented for `RefCell<Foo>`
note: required by a bound in
`Handler::State`
  --> src/kernel/irq.rs:99:24
   |
99 |     type State: Send + Sync;
   |                        ^^^^
   | required by this bound in
   | `Handler::State`
```

Points 2 through 4 illustrate some general principles we follow throughout our framework. In the APIs exposed to driver developers, we use generics over void pointers, bind resource life cycles to object lifetimes with RAII, and enforce thread-safe types where applicable.

5.2 Native Nautilus Threads

Threads are the basic units of preemptive multitasking, and concurrent programming almost always involves threading. Nautilus supports threads natively, and its C interface provides a suite of functions for threading, defined in thread.h. Given the near-ubiquity of threading in many contexts, we provide idiomatic Rust wrappers for working with threads. Our API is implemented in thread.rs and is similar in design to the Rust standard library.

5.2.1 The threading API. The simplest way to run threads from Rust is with thread::spawn. The only argument to this function is the function to run in the new thread. Unlike C, Rust supports first-class functions. The thread function, therefore, may be a simple function pointer (as in C) or an anonymous closure, capable of capturing its environment (like the lambdas of many other languages).

Using thread::spawn is simple:

```
use crate::kernel::thread;

let handle = thread::spawn(|| {
  "it works!"
});
let output = handle.join().unwrap();
assert_eq!(output, "it works!");
```

This (contrived) example creates and runs a single thread that immediately returns "it works!". The parent uses the JoinHandle returned by spawn to block until the child thread finishes, via the associated function JoinHandle::join. Finally, the child's output is verified.

This simple example highlights key parts of our threading API.

- No unsafe code is required to use our threading interface, even though C code is calling Rust functions.
- Thread handles are managed using RAII. A JoinHandle is an owned permission to join on a thread, and the output of that thread can only be used when it has been joined on, at which point the JoinHandle is consumed. Bugs caused by accessing the output of an unfinished thread or trying to join on a thread twice are made inexpressible. And detaching a thread is as simple as dropping the JoinHandle without calling join.

The thread module also provides a Builder type, which can be used to configure various aspects of a new thread, such as its name, stack size, bound CPU, and whether it should inherit the virtual console of its parent. Because Rust does not support default function arguments, we use the Builder pattern. A Builder can be used to configure a thread-to-be, and the Builder::spawn method can be used to actually launch the new thread:

```
use crate::kernel::thread;

let handle = thread::Builder::new()
    .name("my_thread")
    .stack_size(thread::StackSize::_4KB)
    .bound_cpu(0)
    .inherit_vc()
    .spawn(move || {
        // some work here
    })
    .unwrap();
```

Working with a shared state is as easy as capturing the environment with a closure:

```
let x = Arc::new(Spinlock::new(0_usize));

for _ in 0..10 {
```

[1] The inappropriate type substituted for Spinlock here is RefCell. This is essentially the non-thread-safe version of a mutex, and the reason why such a thing would ever be necessary (interior mutability) is a quirk of Rust's borrow checking.

```
    let my_x = Arc::clone(&x);
    handles.push(thread::spawn(move || {
        for _ in 0..10000 {
            *my_x.lock() += 1;
        }
    }));
}
```

As with IRQs, spawn ensures that the captured environment is safe to send between threads; it's a compile-time error if it is not.

5.2.2 Thread demo. To show the usage of our threading API, we have created a threading demo shell command. Run rust_thread in the Nautilus shell to see the API in action.

5.2.3 Threading internals. Creating an idiomatic Rust API for threading was one of the more challenging tasks of the project. The interactions with C that support the abstractions above are fairly involved, requiring Rust callbacks to run from C and dealing with undocumented parts of Nautilus' C interface. Without going into tedious detail, this presents a chance to peek behind the curtain and investigate just one part of the many wrappers which form our framework.

Calling Rust functions from C is nontrivial. Rust functions do not, by default, conform to the C ABI (in fact, the Rust calling convention is intentionally unspecified). And while regular functions can be made to use the C calling convention, there is no way to coerce anonymous closures to the C ABI. Moreover, Nautilus' C interface expects the thread callback to accept raw pointers as its arguments—not at all idiomatic in Rust. The solution, inspired by a Rust ORM's interaction with SQLite's C library [6], is to define *one*[2] callback function that conforms to the C ABI and expected signature, which is generic over the 'actual' Rust callback called within. The compiler's monomorphization effectively builds each thread callback, as necessary, from the single generic function, and the 'actual' Rust callback is passed through the input pointer in nk_thread_create.

Enabling thread output is also subtly difficult. Nautilus' C API seemingly provides a way to get output from finished threads (via the output pointer given to nk_thread_create), but the behavior of this output in our testing was bizarre, and no C code using thread output to reference was found. As a result, our library actually places the output at the end of the memory behind the input pointer.

What about the actual thread input—do we place this behind the input pointer too? As it turns out—yes, though indirectly. Like the Rust standard library, we require thread functions to take no arguments. Input passed to threads must therefore be captured with a closure rather than passed as arguments, and this closure, as previously mentioned, is stored behind the input pointer.

5.3 Cooperative Multitasking

The Nautilus Kernel supports only preemptive multitasking, making the addition of cooperative multitasking a valuable feature.

5.3.1 The Async/Await Framework. The Future trait in Rust represents a value that may or may not currently exist, but hopefully will at some point in the future, and consists of a response type, and a poll function to check the value's status - either Ready or Pending. This lightweight abstraction eschews the need for thread creation and allocation, avoiding the costs of context switching. In addition, they leverage Rust's borrowing and ownership system, effectively mitigating the risk of data races common when using locks. Futures are well-suited to asynchronous operations, especially I/O, as they are non-blocking during execution, thereby enhancing responsiveness. Additional features, such as combinators, chaining, and composition, further enrich its capabilities.

The framework relies heavily on executors and wakers, key components that orchestrate asynchronous execution. When async code is compiled in Rust, it can be thought of as a state machine where each await call on a future symbolizes a state, and every poll call signifies a possible state transition. The compiler generates code that permits the future to pause execution, surrender control, and resume at a later time when the operation concludes.

An Executor serves as a principal orchestrator, responsible for spawning, scheduling, and executing tasks. It avoids the inefficiency of continuous polling by utilizing the Waker API to track futures' statuses. Every future is launched by the Executor as an independent task, inclusive of the Waker, facilitating task switching from a single centralized object. Consequently, executors efficiently handle thousands of tasks without requiring a corresponding number of threads. This scalability offers considerable benefits for operating systems.

The Waker plays a crucial role in notifying the executor upon a future's completion. It allows asynchronous operations to run concurrently, with the executor keeping the CPU active by scheduling tasks based on waker feedback [11].

5.3.2 Async/Await in Nautilus. In our framework, the foundation of the cooperative multitasking model is embodied by the Task module. It is specifically tailored to operate in environments lacking a standard Rust runtime, such as no_std environments or bare metal systems. The module provides an Executor struct that manages and executes tasks, and a

[2] Actually, there are two thread callback functions to support the optional virtual console inheritance behavior of new threads. But the main point still holds.

utility function `yield_now` for voluntarily yielding execution from the current task.

```rust
let mut executor = Executor::new();

let task1 = Task::new(async {
    // Some work here...
    utils::yield_now().await;
    // The task will resume here
    // More work...
});

let task2 = Task::new(async {
    // Task2 can run while task1 is yielded
});

executor.spawn(task1);
executor.spawn(task2);
executor.run(true);
```

In this example, `task1` does some work, then yields execution using `yield_now`. The executor then has the opportunity to run other tasks (like `task2` in this case) before `task1` resumes. `yield_now` is a cooperative mechanism: a task must choose to call it to yield execution. Tasks that do not call `yield_now` will not yield execution to other tasks and can monopolize the executor if they run for a long time without completing.

Despite its effectiveness, the task execution system is quite basic and may not be suitable for all use cases. It does not support task prioritization, preemption, or many other features found in more sophisticated task scheduling systems. Its simplistic implementation likewise precludes techniques which take advantage of multiple CPUs, like work stealing. Therefore, further work is needed to extend the capabilities of this system to meet diverse scheduling requirements.

This cooperative multitasking feature serves as a valuable addition to Nautilus. By integrating this feature, tasks can voluntarily yield time to the CPU, thereby enabling efficient management of computing resources. This is particularly useful in environments with scarce computational resources, such as in embedded systems. Furthermore, this feature also contributes to the goal of integrating the Rust language into Nautilus, thereby leveraging Rust's safety features to ensure the reliable execution of kernel tasks.

6 DEVICE DRIVERS

We implement a Rust-based Parport device driver, utilizing the preexisting Character Devices (`chardev`) abstraction in the Nautilus kernel, to provide a simple I/O interface for byte-sized read and write operations.

We also implement a Rust-based Virtio GPU driver, capitalizing on the GPU device abstraction present in Nautilus, for creation and management of para-virtualized devices within virtual machines.

6.1 Parallel Port

6.1.1 The parallel port as a character device. The `Parport`, also recognized as the `printerport`, serves as the simplest I/O device implemented, echoing the first parport, `LPT1`, from the original IBM PC. This device transforms a byte input into an 8-bit output via 8 unique wires within the connector. By strobing a distinct wire from high to low, the device becomes aware of a byte change, thus, effectively 'clocking' the data. The driver's interaction with the device is maintained through three single-byte registers: `control`, `status`, and `data`.

The `chardev.rs` Rust module facilitates an interface for the construction of a character device for Nautilus, predominantly through its integration with a parallel port driver. By initiating an `InternalRegistration<T>` structure, it forms a secure repository for pointers tied to the device and related data, along with a device name. The `PhantomData` function within this structure is employed to signal the ownership of type `T`. For device registration, a `try_new` method is implemented, while the `Drop` trait is integrated for secure deregistration and data management when the device goes out of scope. In order to symbolize device status and the outcome of read and write operations, two enums, `Status` and `RwResult<T>`, are established.

The `CharDev` trait, which serves as the primary interface for a character device, incorporates methods for status checks, read and write operations, and fetching device features. The `Registration` struct, brings supplementary functions into play, including registering a character device, signaling idle threads, fetching the device name, and a `Drop` trait for freeing memory. To lay the groundwork for interaction between Nautilus and the Rust-defined character device, it constructs the required C interfaces, via extern `"C"` functions like `status`, `read`, `write`, and `get_characteristics`.

6.1.2 The Rust parallel port driver. The Rust implementation for the parallel port driver allows us to interact with a parallel port, enabling data read and write operations for a connected device.

This code establishes the primary structures and methods that enable the operation of the parallel port. At the heart of this driver is the `Parport` struct, which symbolizes the parallel port. This structure houses the details required to operate the port, such as the port's character device registration, IRQ registration, I/O port, and status.

Two structures, `StatReg` and `CtrlReg`, are implemented using bitfields to map the port's control and status registers, allowing for the individual bits within a byte to be manipulated easily.

```
bitfield! {
    pub struct StatReg(u8);
    reserved, _: 1, 0;
    irq, _: 2;
    err, _: 3;
    sel, _: 4;
    pout, _: 5;
    ack, _: 6;
    busy, set_busy: 7;
}
// ...
```

The status of the port is represented by the `ParportStatus` enumeration.

Interrupts are handled via the `irq::Handler` trait for the `Parport` structure. The interrupt handler is designed to update the port's status to `Ready` once an interrupt is received, indicating the completion of a read/write operation.

The driver also includes implementations for reading and writing to the parallel port, taking care to ensure that no operation is attempted while the device is `Busy`.

For use of this driver, a command-line interface is included which provides shell commands “parport up" and “parport down" for manual control of the parallel port.

```
register_shell_command!(
    "parport",
    "parport up | parport down",
    |command| {
        // ...
    }
);
```

These commands can be particularly useful for troubleshooting and testing the driver.

All of these components come together to form a complete, Rust-based driver for interacting with a parallel port. This driver demonstrates the feasibility and effectiveness of implementing kernel-level drivers in Rust and serves as an excellent example of the advantages that Rust brings to systems programming, including memory safety, thread safety, and expressive, high-level abstractions.

6.2 GPU Devices

A GPU device is a specialized processor responsible for rendering and accelerating graphics-related tasks. For instance, when playing a video game, the GPU is tasked with rendering graphics to the screen for the user's visual experience. To facilitate this, an operating system needs to communicate efficiently with the GPU, a task handled by a device driver. The GPU device driver acts as a translator between the system's high-level instructions and the machine code understood by the device.

In Nautilus, `gpudev.h` provides the C interface for GPU devices via the `nk_gpu_dev_int` struct, an abstraction which currently supports only 2D accelerated graphics cards; available video modes are limited to Text and Graphics only. GPU devices may be be registered with `nk_gpu_dev_register`, and unregistered with `nk_gpu_dev_unregister`. The interface declares various functions to initialize and de-initialize the device, interact with the device, and defines the abstractions for scanouts, pixels, etc. With a similar approach used to implement IRQs and the character devices, we use RAII to wrap this interface, bind the registration, and allow Rust's lifetime system to handle de-allocation.

Registering a GPU device is achieved through the RAII compliant `Registration` impl. We use `bindgen` to enable FFI calls into the C function `nk_gpu_dev_register`. Deregistration is achieved similarly, calling into the `nk_gpu_dev_unregister` function when the registration is dropped.

The `GpuDev` trait (defined in `gpudev.rs`) defines the idiomatic Rust interface for a GPU device.

```
pub trait GpuDev {
    type State: Send + Sync;
    /// various interactive functions
};
```

Each of the interactive functions to command the device was implemented in Rust using external calls to C. `Bindgen` was used extensively to match the struct unions and alignments needed by C to function. Because the C interface has certain design choices stylistically opposed to Rust, many `unsafe` calls for direct pointer manipulation were needed. Documentation for each is included.

To create, register, and employ a specific GPU device, the device driver must implement this trait. Graphics in Nautilus can be generated by the VGA or by GPU devices, the latter of which we implement with a Virtio GPU driver in Rust.

6.3 Virtio GPU

6.3.1 Virtio Model in Nautilus. In the Nautilus kernel, the GPU is simultaneously a GPU device, a Virtio device, and a PCI device with MSI-X interrupts. Virtio GPU devices offer their registers for read and write operations via Memory-Mapped I/O, with the PCI abstraction providing atomic load and store functions for thread safety in these operations. We implement the Virtio GPU device driver in Rust.

The Virtio driver writes into a frame buffer, an array of pixel objects stored in memory. After rendering pixels and preparing them for display, the frame buffer is flushed to a scanout (the abstraction of the monitor), and is thus visible to

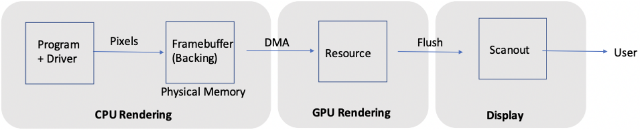


Figure 2: Conceptual model of a GPU device.

the user. Pixels in the buffer are copied to the resource by the GPU device via Direct Memory Access (DMA). A virtqueue abstraction is used to issue requests and responses, again via DMA, from the driver. In the Nautilus Kernel, all Virtio devices utilize the same virtqueue framework, however, their use depends on the specific Virtio device – this includes the number of virtqueues and the request/response API, among other variations. The GPU device utilizes two virtqueues: one to configure and trigger the rendering pipeline, and another to control a mouse cursor overlay to the scanout. The cursor implementation is omitted.

Using the GPU device requires the allocation of request and response objects which may or may not be fragmented over non-contiguous memory chunks. The implementation assembles these request and response fragments into a linked list - a descriptor chain - where each node describes the fragment’s relevant information including the starting address, its length, a flag to indicate it is writable, and a pointer to the next descriptor in the list. The descriptor chain is a scatter/gather list. The virtqueue data structure is shared with the hardware.

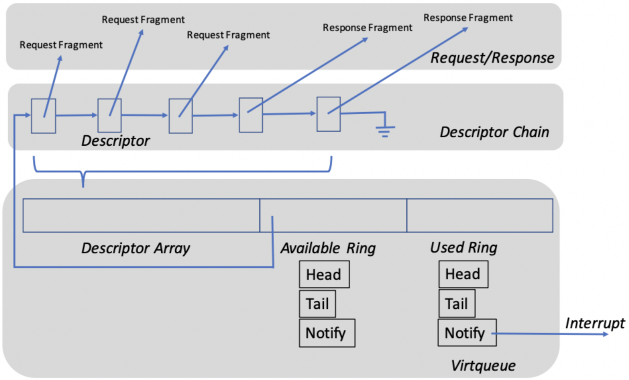


Figure 3: The relationship between a request, a response, and a virtqueue.

Reads and writes occur directly, while hardware reads and writes occur via DMA. The virtqueue is comprised of the descriptor chain, and two ring buffers, *available* and *used.* These are producer/consumer rings where the driver is the producer and the device is the consumer. To command the device, a pointer to the index of the first descriptor in the chain is pushed into the *available* ring, and the notification register is written to signal the device of the change in the virtqueue. The device reads the descriptor pointer from the *available* ring via DMA. It then reads the request fragments, again using DMA, by walking the chain. A response is written to the response fragments, and a pointer to the head of the descriptor chain is written to the *used* ring, all with DMA, before the device raises an interrupt to signal the driver of the change in the virtqueue’s state [15].

6.3.2 Implementing the driver. Implementing the Virtio GPU device drier in Rust relied heavily on the use of `bindgen`, as we did not have time to make a full Rust interface in the `kernel` folder. The C driver (defined in `virtio_gpu.c`) lays out the ways to manipulate feature bits on the PCI device, provides the abstractions and data types defined by virtio, defines core functions of the driver, and implements the command functions defined in the device interface.

Thus, we implement the trait requirements defined in the `GpuDev` trait discussed previously, and write the command functions in the Rust language. A `VirtioGpu` object is defined in `virtio_gpu/mod.rs`, and mirrors the C definition lacking the cursor related data.

```
struct VirtioGpu {
 gpu_dev: Option<gpudev::Registration<Self>>,
 pci_dev: *mut bindings::virtio_pci_dev,
 have_disp_info: bool,
 disp_info_resp: RespDisplayInfo,
 cur_mode: CurModeType,
 frame_buffer: Option<Box<[Pixel]>>,
 frame_box: Rect,
 clipping_box: Rect,
 text_snapshot: [u16; 80 * 25],
}

impl gpudev::GpuDev for VirtioGpu {
    type State = Spinlock<VirtioGpu>;
    // ...
}
```

With the existing C implementation, `bindgen` was used for many of the type definitions and function calls to avoid redundancy (`RespDisplayInfo`, `Pixel`, and `Rect` above are examples). Structs used in device transactions required the use of the `#[repr(C)]` tag to match the exact memory layout expected by the Virtio GPU device, as the layout of Rust structs is otherwise unspecified.

Having created a Virtio device driver in Rust, we test it with Nautilus’s `gputest` shell command. The test is rudimentary, filling the screen, drawing boxes in clipping regions, drawing lines, and moving boxes.

7 NAUTILUS RUNS DOOM

To further test the Rust Virtio GPU driver, we sought out something more intensive and less contrived than the existing shell command. Seeing an opportunity to add something fun to Nautilus, we decided to port the original DOOM engine. We base our implementation off a libc-independent source port of DOOM built for this purpose [13]. Implementing many of the functions required by this abstraction layer was simple enough, like, for example, using Nautilus' `malloc` for `doom_malloc` and `nk_vc_printf` for `doom_print`. Other requirements were harder to satisfy.

7.0.1 *Getting the current time.*

DOOM needs a way to get the current time in order to drive the game loop forward at the right speed. This was easy to implement using `clock_gettime` from `libccompat.h`, but required enabling the HPET device driver.

7.0.2 *Reading the game files.*

DOOM uses file operations to read from the game files. Nautilus provides the necessary functions in `fs.h`, but the file system drivers proved buggy. The Fat32 driver read the files incorrectly (correct enough to launch the game, but with serious bugs), while the Ext2 driver was unable to read the files at all. We thank Nick Wanninger for sharing his implementation of a functional LittleFS driver.

7.0.3 *Getting keyboard input.*

Nautilus' virtual console abstraction makes it easy to get keyboard input, but the typical way of using one (with a `COOKED` virtual console) does not seem to allow for distinguishing between press and release. We used a `RAW_NOQUEUE` virtual console, parsing the scancodes ourselves, to get around this.

With these implementations done, it was simple enough to use our Rust Virtio GPU driver's bitmap drawing functionality to play DOOM, with tolerable input lag and frame rate, from the Nautilus shell.

To try it out, first download the shareware version of the DOOM game files (e.g. from [17]), or use your own. Enable HPET support, LittleFS, the core Virtio PCI driver, the Virtio block driver, and the Virtio GPU driver (select either the C driver or the Rust one). Finally, enable the DOOM engine in the 'Build' submenu.

After the first build with these options, a folder in the working directory called `root` will be made, to serve as the root of the LittleFS file system given to Nautilus. Place the game file (e.g. `doom1.wad` for the shareware version) in `./root/home/`. Simply run `doom` in the shell and wait for the game to boot. The player can move with `W`, `A`, `S`, `D` and turn with `H` and `L`. Use `E` to interact and `SPACE` to shoot.

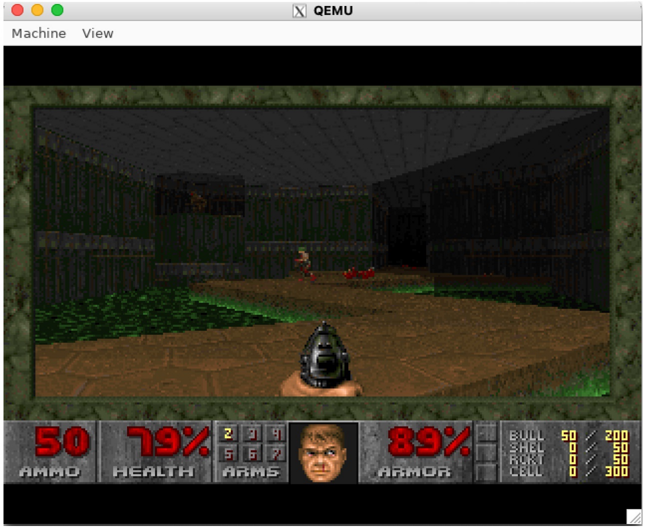


Figure 4: Playing DOOM in Nautilus.

8 FUTURE WORK

Future groups working on Rust in Nautilus may consider addressing the following items.

- We have developed strong concurrency support for Rust modules, but have only used them in simple shell commands. Find a better use case for our threading API or asynchronous executor.
- Nautilus provides functions for thread-local storage. Try wrapping them in the threading API in a way consistent with the design of the Rust standard library.
- Ideally, the Virtio GPU driver would not need to use C bindings directly. Additionally, it is not written in very idiomatic Rust and uses `unsafe` code more than necessary. Try tidying up this driver.
- We had hoped to integrate sound with our DOOM port but ran out of time. See if you can use one of the sound card drivers to play sound effects and MIDI audio in DOOM.
- Develop new drivers in Rust! The core goal of this project was establishing a framework for Rust modules, and, as a result, the drivers we wrote to showcase it are relatively simple and already have C counterparts. Take advantage of our framework to write Rust drivers for devices Nautilus currently does not support.

REFERENCES

[1] Avram Abel. 2012. Interview on Rust, a Systems Programming Language Developed by Mozilla. https://www.infoq.com/news/2012/08/Interview-Rust/

[2] National Security Agency. 2022. Software Memory Safety. https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-

software-memory-safety-issues/
[3] Bindgen Documentation. [n. d.]. The bindgen User Guide. https://rust-lang.github.io/rust-bindgen/
[4] Mazdak Farrokhzad. 2019. Tracking issue for RFC 2523. https://github.com/rust-lang/rust/issues/64797
[5] Rust for Linux. [n. d.]. The kernel crate. https://rust-for-linux.github.io/docs/kernel/
[6] Sage Griffin. 2019. Neat Rust Tricks: Passing Rust Closures to C. http://blog.sagetheprogrammer.com/neat-rust-tricks-passing-rust-closures-to-c
[7] Kyle Hale. 2018. Nautilus AeroKernel. http://cs.iit.edu/~khale/nautilus/
[8] Peter Dinda John Albers. 2019. Rust in Nautilus.
[9] Michael Larabel. 2022. The Initial Rust Infrastructure Has Been Merged Into Linux 6.1. https://www.phoronix.com/news/Rust-Is-Merged-Linux-6.1
[10] Microsoft©. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf
[11] Philipp Oppermann. 2020. Writing an OS in Rust. https://os.phil-opp.com/async-await/#async-await-in-rust
[12] Michael Polinski Qinqwei Lan, Hanming Wang. 2022. Porting Rust to Nautilus.
[13] David St-Louis. 2022. Pure DOOM. https://github.com/Daivuk/PureDOOM
[14] Carol Nichols Steve Klabnik. 2023. The Rust Programming Language. https://doc.rust-lang.org/book/
[15] Northwestern University CS 343: Operating Systems. 2022. Driver Lab: Interfacing with Hardware and the Rest of the Kernel.
[16] Paul Thurrott. 2023. First Rust Code Shows Up in the Windows 11 Kernel. https://www.thurrott.com/windows/windows-11/282995/first-rust-code-shows-up-in-the-windows-11-kernel
[17] Doom Wiki. [n. d.]. DOOM1.WAD. https://doomwiki.org/wiki/DOOM1.WAD
[18] OSDev Wiki. [n. d.]. Interrupts. https://wiki.osdev.org/Interrupts