Reenix: Implementing a Unix-Like Operating System in Rust

Alex Light (alexander_light@brown.edu)
Advisor: Tom Doeppner
Reader: Shriram Krishnamurthi
Brown University, Department of Computer Science

April 2015

Abstract

This paper describes the experience, problems and successes found in implementing a unix-like operating system kernel in rust. Using the basic design and much of the lowest-level support code from the Weenix operating system written for CS167/9 I was able to create a basic kernel supporting multiple kernel processes scheduled cooperatively, drivers for the basic devices and the beginnings of a virtual file system. I made note of where the rust programming language, and its safety and type systems, helped and hindered my work and made some, tentative, performance comparisons between the rust and C implementations of this kernel. I also include a short introduction to the rust programming language and the weenix project.

Contents

1 Introduction

Ever since it was first created in 1971 the UNIX operating system has been a fixture of software engineering. One of its largest contributions to the world of OS engineering, and software engineering in general, was the C programming language created to write it. In the 4 decades that have passed since being released, C has changed relatively little but the state-of-the-art in programming language design and checking, has advanced tremendously. Thanks to the success of unix almost all operating system kernels have been written largely in C or similar languages like C++. This means that these advances in language design have largely passed by one of the fields that could most benefit from the more expressive, more verifiable languages that have come after C.

The goal of this project is to try to create a unix-like operating system kernel using the rust programming language. As far as I know, this is a project that has never really been attempted seriously, nor had anyone made much progress on before now[1]. By doing so I will be able to explore the feasibility and convience of creating a kernel with a higher-level

[1] All other rust operating systems I was able to find were little more than toys capable of running basic rust code. Few had any notion of threads of execution and even fewwer had any form of process control beyond starting new threads. None had any form of drivers beyond painting the screen and maybe echoing key presses without processing them.

language such as rust, as well as find where the language could be improved to better suit this purpose. Furthermore, it will allow us to evaluate how well the basic design of unix holds up when examined through a language other than C. Finally, we will see how the much more sophisticated type and safety system of rust handle the complicated task of verifying a kernel.

In order to allow me to begin working on the more high-level parts of the kernel faster, I based my effort off of the weenix operating system that is implemented in CS169. This allowed me to not worry about implementing many of the lowest level pieces of the kernel, such as the memory allocators, which are not specific to operating system development.

1.1 The Weenix OS

The Weenix operating system is a small x86 based teaching OS created in 1998 for use with Brown's CS167 course on operating systems[12]. Today, students in the optional lab course CS169 attached to CS167 implement much of the higher level pieces of a unix-like OS with weenix. Students doing this project start out with the code necessary to get the OS booted and running C code, with memory-management, a debug-printing system, and a basic libc implementation. Using this as their base, CS169 students then proceed to implement a fairly complete unix OS. The project, and its support code, are written almost entirely in C, with some of the initial boot code being x86 assembly, and some python and shell scripts for running and testing the OS.

This project is split into multiple parts, commonly referred to as, PROCS, DRIVERS, VFS, S5FS, & VM. For PROCS, they implement a unix-style process model, with parent-child relationships among processes and a init process, as well as a simple scheduler and synchronization primitives. During DRIVERS, they implement large parts of a TTY driver, allowing user input and interaction, as well as a (very bare-bones) ATA driver allowing use of a hard-disk. For VFS, they implement a virtual file system type abstraction, using a provided ram-backed file system called RamFS for testing. In S5FS, a version of the sysv-fs file system, called the S5 file system, is implemented to allow real non-volatile storage. Finally for VM a virtual memory and user-space is implemented. There are also many provided user-space utilities that allow one to test the final OS.

1.2 The Rust language

The rust[2] programming language is a relatively new systems programming language being made by the Mozilla foundation. It is designed to be usable as a replacement for C in the low-level and embedded programming environments that are common for small and high-performance software. The Mozilla Foundation is currently using rust in a few official projects, including the rust compiler (rustc), and an experimental web-browser called Servo. It also plans to begin using rust code in its popular Firefox web-browser in the near future[4]. Rust is currently being developed and hosted on Github[3]. The project is very popular and open, with thousands of contributors, most of whom are not associated with Mozilla.

Rust itself is a procedural programming language with C-like syntax. It uses its very comprehensive type system, a data 'lifetime' system, and an extremely small runtime to ensure memory and thread safety during compile time. Specifically, rust uses its ownership and lifetime tracking system to ensure that data is not unexpectedly modified when it is still being used by another object. Furthermore, it uses the lifetime tracking system to ensure that there are no dangling pointers possible in the language. The runtime of rust is made up of several pieces, many of which are separable. Its only required (and most basic) function is to simply recover from out-of-memory or other fatal errors. In most cases, including most of reenix, it also provides an interface for allocation of heap memory. All other functions of the runtime are essentially just to provide a consistent interface to the underlying operating system it is running on, allowing disk-io, inter-process communications and the creation of threads, among other things.

1.2.1 Syntax & Semantics

The syntax of rust is similar to, but slightly different from most other C-like programming languages. Figure 1 contains a basic quicksort implementation in rust that I will use to illustrate some of the languages features. A full description of the rust syntax and semantics can be found online at doc.rust-lang.org[4].

The most notable difference is that rust has a somewhat different divide between expressions and statements. In rust an expression is any piece of code that yields a value. A statement, on the other

[2] http://www.rust-lang.org (April 2015)
[3] https://github.com/rust-lang/rust (April 2015)
[4] http://doc.rust-lang.org/reference.html (April 2015)

```rust
//! A basic quick-sort implementation

/// A type generic quick-sort. `T` is the type we are sorting, it must have a total ordering
/// (implement the `Ord` trait).  It takes a list by value and returns a sorted list containing the
/// same elements sorted. We say that this passed in list is mutable so we can modify it.
pub fn quicksort<T: Ord>(mut lst: Vec<T>) -> Vec<T> {
    // Get the first element as our pivot, Pop will return None (and go to the else branch) if this
    // list is empty, otherwise it will remove the first element from the list and return it.
    if let Some(pivot) = lst.pop() {
        // Split list around the pivot. We iterate through the list (into_iter function) and
        // partition it into two lists.  The partition function turns an iterator into a pair of
        // lists where the first is a list of all elements where the condition given is true and
        // the other is false.
        let (less, more): (Vec<_>, Vec<_>) = lst.into_iter().partition(|x| x < &pivot);
        // Recursively sort the half of the list less then the pivot. This will be the start of our
        // returned list.
        let mut res = quicksort(less);
        // Push the pivot element onto the end of the sorted lower half of the list. This appends
        // the pivot onto the `res` list.
        res.push(pivot);
        // Sort the larger half of the list and append it to the sorted lower half and pivot.
        // extend will append the entire given list onto the `res` list.
        res.extend(quicksort(more));
        // Return the now sorted list. Note the return statement is not required here. Simply
        // making this line 'res' (note the lack of a ';') would be equivalent since the function
        // will return the value of the last expression (this if-else) which takes the value of the
        // last expression in its branches (Vec<T>).
        return res;
    } else {
        // Since lst.pop() returned None the list passed in must be empty so we will return an
        // empty list here. Note that return is not needed because this is the last expression in a
        // block and this block is the last expression in the function. vec! is a standard macro
        // that creates a Vec<T>.
        vec![]
    }
}

fn main() {
    // Create a list to sort. vec! is a macro and will create a vec containing the elements listed.
    let lst = vec![3,1,5,9,2,8,4,2,0,3,12,4,9,0,11];
    println!("unsorted: {:?}", lst);
    // Call quicksort. This relinquishes ownership of lst.
    println!("sorted:   {:?}", quicksort(lst));
}
```

Figure 1: A rust quicksort

```rust
/// A trait. Structs and enums can implement this.
pub trait Id {
    /// A required function. All implementers must provide a definition for this function or else
    /// type-checking will fail. The 'static means the returned string must be statically
    /// allocated.
    fn username(&self) -> &'static str;
    /// A function with a default implementation. The returned string must be usable at least as
    /// long as the Id exists. The 'a means that the returned str must be usable at least as long
    /// as 'self' is. The type checker will ensure this is true.
    fn screenname<'a>(&'a self, _board: &str) -> &'a str { self.username() }
}

/// A structure. The derive provides default implementations for the given traits.  Only certain
/// traits may be implemented in this way.
#[derive(Debug, Eq, PartialEq)]
pub struct Account { name: &'static str, msgs: Vec<u64>, }

// Implementing the Id trait. Note we do not need to provide a 'screenname' implementation since
// there is a default version.
impl Id for Account {
    fn username(&self) -> &'static str { self.name }
}

// Functions associated with Account directly.
impl Account {
    pub fn get_messages(&self) -> &[u64] { &self.msgs[..] }
}

#[derive(Debug, Eq, PartialEq)]
pub enum Commenter {
    /// An enum variant with data
    User(Account),
    /// An enum variant without data
    Anon,
}

/// Implement the Id trait.
impl Id for Commenter {
    fn username(&self) -> &'static str {
        // We take different actions depending on the variant.
        match *self {
            Commenter::User(ref a) => a.username(),
            Commenter::Anon => "Anon",
        }
    }
}
```

Figure 2: Rust traits and types

hand does not create a value. Within functions everything is generally an expression except for (1) `let` variable bindings, such as on lines 14, 17, and 40 of Figure 1, (2) looping constructs, and (3) any expression or statement with a semicolon (`;`) placed after it. Note that blocks, delimited by curly-braces ({}) are also expressions, using the value of the last expression they contain as their value. In the same vein both `if-else` and `match` blocks are also expressions. In Figure 1 the `if-else` block beginning on line 9 is an expression of type `Vec<T>`, for example. `Rust` takes this idea of the final expression of a block being its value even further, placing an implicit `return` before the final top-level expression in a function (in Figure 1 this is the `if-else` starting on line 9); one may still use '`return <value>;`' to return earlier, however. This can be seen on lines 41-44 of Figure 2, where the result of the `match` is what is returned by the function. Furthermore, this means that we could change line 28 of Figure 1 to simply be '`res`' and the meaning of the program would remain the same.

Another notable difference from `C` is that `rust` is fully immutable by default. In order to use an object mutably one must declare it `mut`, as is done in line 17 of Figure 1. One must do this even for function arguments, which is why there is a `mut` before the argument `lst` on line 6 of Figure 1. This immutable default extends to pointers, which must be declared `&mut` to be used mutably.

`Rust` has a syntax for declaring structures and enumerations that is very similar to `C`. One of the main differences is that enumerations can have data associated with them. In Figure 2 on line 30 we see the definition of an enumeration where one of its variants (`User`) has data of the type `Account` associated with it. This data can be used in `match` expressions, such as on line 42 of Figure 2. In `rust` there are also `traits`, which are similar to `Java` interfaces and may have default function definitions associated with them. Using `traits` it is much easier to create generic functions than it is in `C`. For example, the quicksort implementation in Figure 1 only requires that the objects being sorted implement the `Ord` trait, meaning they have a total ordering. We can see the `Id` trait be defined on lines 2-11 of Figure 2, it is implemented for the `Account` type on line 20 and for the `Commenter` type on line 38. Both enumerations and structures can have methods implemented on them directly or through `traits`. The `Commenter` trait has a `get_messages` function implemented on it on line 26 of Figure 2. `Rust` will transparently redirect function calls through virtual method tables (vtables)[5] when appropriate to allow handling objects as `trait` pointers. This also makes it much easier to write generic code, as well as to hide implementation details in a much more straightforward way than is possible in `C`.

`Rust` also supports declaring anonymous functions. Anonymous functions are declared by having a list of arguments surrounded by pipes (`|`) followed by a single expression. Type annotations similar to those on normal functions are allowed, but optional. The return and argument types will be inferred if they are absent. An anonymous function is used on line 14 of Figure 1. On this line it is used to distinguish between items less then the pivot so that the `partition` function can split the items in the list into two lists.

Figure 1 also makes use of the `rust` macro system. In `rust` macros are pieces of code that transform the abstract syntax tree (AST)[6] at compile time, rather than just the raw text of the source code as `C` macros do. Macros may be implemented using a special macro Domain Specific Language (DSL)[7] or by writing a compiler-plugin for `rustc`[5]. Both systems allow the creation of hygienic macros, where there can be no name collision and the meaning is (mostly) independent of the context it is used in. The macro DSL does not allow any compile-time computation beyond pattern matching and has no explicit quasi-quote operator[8], however compiler plugins may do both these things. Macros are identified by the exclamation-mark (`!`) that ends their name. They may expand to be either statements or expressions and may (generally) be nested. In Figure 1 I make use of the `vec![...]` macro, which creates a `Vec<T>` filled with the arguments given to the macro, on lines 34 and 40.

`Rust` also has fairly robust support for pattern

[5] VTables are structures containing function-pointers used to allow types to specify different implementations of standard functions for their use. They are similar to interfaces in `Java`.

[6] An AST is a representation of a program as a tree, with the nodes and edges representing the syntactic elements of the language. The tree as a whole represents the parse of the program being examined. It is used as the representation of a program during compilation, optimization, and macro-expansion.

[7] A DSL is a programming language created for some specific purpose. It is usually quite well suited for use in this domain but is less powerful or more difficult to use than more general languages. Some commonly used DSLs are the regular-expression syntax used by perl, the Hyper-text Markup Language (HTML) commonly used for creating web-pages, and the typesetting language LaTeX.

[8] A quasi-quote is an operator that turns a given piece of text into an AST for the text. Using a related operation called "unquote" one is allowed to embed other ASTs as it does so.

matching. In 'let' statements one can 'de-structure' objects and tuples into their constituent parts, Figure 1 does this on lines 9 and 14. On line 14 we de-structure the two-tuple returned by partition into the two lists that make it up. On that line we also need to specify that we want Vec<_>s to tell the compiler which variant of partition to use. It is also possible to do this with enums, although there one must either use an if-let, as Figure 1 does on line 9, or use a match statement to cover all possible variants, as is done in the implementing of username in Figure 2 in lines 41-44.

1.2.2 Ownership

Another major feature of rust is its ownership system. In general, every object in rust has some, specific owner. The owner of an object is the one that is responsible for destroying the object when it goes out of scope. Ownership can be transfered by either passing the object 'by value' (without using a pointer) to another function or by returning an object from a function. When ownership is transfered in this way the object is said to be 'moved' to its new owner (although actual memory might or might not be moved). Once ownership has been transfered, the original owner of an object may no longer directly use the moved object, it must obtain a pointer from the new owner if it is to do anything. A transfer of ownership can be seen in Figure 1 on line 43 where ownership of the variable lst is passed into the quicksort function. If one attempted to make use of lst after this line the compiler would prevent it by saying that the lst variable has been moved out of scope. In the quicksort function itself the ownership of the variable res is transfered up the call stack by returning it on line 28. The fields of an object are said to be owned by the object that contains them. This forms a tree of ownership, the root of which is either in some stack frame of the current thread or in a statically allocated object.

There are, of course, some exceptions to this system, such as reference counted pointers, weak references[9] and mutexes. These types are all implemented using unsafe behavior, which allows one to ignore the rust type checking and safety system. This makes sharing data between threads difficult, as there is no obvious owner for shared data. Further, when sharing data like this, one needs to ensure that all references stay valid as long as they are in use. I discuss some of the ways I dealt with this problem in subsection 2.4.

[9] Reference counted pointers are special pointers that allow objects to have multiple, concurrent, owners. The object referenced will only be destroyed when all references to it are destroyed. Weak pointers are related to reference counted pointers. Weak pointers allow one to have a reference to an object contained in a reference counted pointer without contributing to the reference count of said object. The rust standard library implements these types as Rc<T> and Weak<T> respectively.

1.2.3 Type & Borrow checker

The rust type checker is a fairly standard statically-typed language checker with type-inference. One interesting feature of the rust type checker is that it does type inference in multiple directions. It will choose which variant of a function to use based on both the types of its arguments and the (declared) type of its return value. For example in Figure 1 I need to specify that less and more are both Vec or else the type checker would not be able to determine which version of partition to use. It is possible to use the underscore (_) to mark places where the type-inference system should supply the types, as is done in Figure 1 on line 14. This is done by default whenever one has a let statement where the type is not supplied, such as line 17 in Figure 1.

The other major feature of the rust checking system is that it takes into account the lifetimes of data. In rust whenever one creates an object the checking system automatically gives it a lifetime. An objects lifetime is from when it is created until when it is destroyed. The lifetime of an object can change when it is moved by value but is otherwise constant. Lifetimes may be given names like any other generic, or one may use the special 'static lifetime, as shown in Figure 2. The name of a lifetime is always marked with an unmatched single quote ('). The borrow checker of rust makes sure that the lifetimes of all objects in a rust program are consistent. It works by whenever a pointer is created to an object that pointer retains the same lifetime as the pointed to object. Rust will then ensure that no pointers outlive the lifetime of the object they point to and that the pointed to object cannot be moved (have its ownership transfered) as long as there is pointers of it still (potentially) alive. For example, in Figure 2 on line 10 we specify that the lifetime of the returned string is the same as the lifetime of the object whose screenname function is being called. The compiler will prevent one from using the string returned by this function anytime after the object that created it is destroyed. Lifetimes can also be incorporated as part of a type, allowing one to hold these pointers

inside of structs and enums. All of these checks are done purely at compile time, incurring no additional runtime overhead.

Rust does allow one to get around these checks if necessary. To do so one uses the unsafe environment. While in an unsafe environment one may do several things normally forbidden by the rust type and safety system. These include dereferencing raw memory and doing unchecked type casts.

2 Reenix

Reenix is the name of my project to reimplement as much of the weenix OS in rust as possible. I choose to split this work up in much the same way the original weenix project is split up. Of the five sections of weenix (see subsection 1.1), I was able to fully implement the first two, PROCS & DRIVERS. I was also able to finish a non-trivial part of the VFS project and made some of the auxiliary support code that is needed to complete S5FS & VM. In doing this I also had to convert and rewrite many large pieces of the weenix support code into rust. The results of my project, including all code and patches to the original weenix project, can be found on Github[10].

2.1 Organization

In building reenix I used the rust language's concept of crates. Rust crates are packages of related code that can be compiled into libraries, or (in some cases) binaries. They are generally identified by finding a folder with a 'lib.rs' file in it, which is the standard (though not obligatory) name for a crate-root. For reenix I ended up dividing the project into nine crates that are entirely new. There are also three compiler plugins that I used in this project, two of which I created. Finally, I make use of several standard library crates and even created a version of the standard library[11] that only uses crates available for reenix. Overall reenix makes use of 18 crates, of which 12 of them are entirely custom.

Many of the crates in reenix mirror the organization of the weenix project. The memory-management architecture is contained in the mm [12] crate, the processes related code is in the procs [13] crate, and so on. There are, however, several crates which have no real analog in weenix that are present in reenix.

The first and most basic of these is the base crate. This crate holds a large and rather diverse collection of types and traits. This is partly to make it possible for crates to know about some of the types of other crates further down the dependency tree. By declar-

[10] https://github.com/scialex/reenix (April 2015)

[11] The rust standard library is made up of a large number of separate crates which are all joined together in a single crate called std. This is commonly called the standard library facade by the rust community. The std crate is treated specially by the compiler which automatically includes some modules from it in every file, allowing one to use the standard rust types without explicitly importing them.

[12] See subsection 2.3

[13] See subsection 2.4

ing certain traits in this crate we can allow them to be used in all crates, even ones which are dependencies of where the trait is implemented. Another function of this crate is to hold the definitions for many of the basic data types used in reenix, such as errnos. I also placed the 'dbg!' macro[14] in this crate so it could be used everywhere without difficulty.

Another new crate is the (somewhat badly named) startup crate. This crate mostly contains stubs to C code that implements functions related to ACPI, PCI, and GDT[15], used mostly during boot, hence its name. These are all very tightly related to the actual hardware and managing them requires the ability to perform complicated bit-twidling and memory manipulation, something rust could be better at. It also includes an implementation of thread-specific data functions. These were mainly put here, and indeed created at all, due to the fact that implementing the rust stack overflow detection[16] for reenix threads made them fairly trivial.

The last totally new crate is the umem crate. This crate is not fully finished and currently holds some of the mechanisms needed to implement user-space virtual memory and page-frame caching. In weenix these are part of the mm hierarchy, however to do that with rust would require that we only have one crate and would make using the rust standard library much more difficult.

2.2 Booting

One of the first challenges I had while writing reenix was getting the system to boot at all. Weenix (at the time I started) made use of a custom 16-bit assembly code boot-loader. This boot-loader, unfortunately, did not support loading any kernel images larger than 4 megabytes. This turned into a problem very quickly as it turns out that rustc is far less adept than gcc at creating succinct output. In fact, I was hitting this problem so early I was barely able to make a "Hello World" before having to stop working on rust code.

Fixing this problem required rewriting most of the early boot code, all of which was x86 assembly. It also required rewriting parts of the build system to create boot disks that used GRUB[17], a common Linux boot-loader, and changing the boot sequence to support the multiboot specification[18]. This, in and of itself, was not terribly difficult, though the fact that this was absolutely critical to making any sort of attempt to do this project does show that some of the simple hacks that are possible with C cannot be done with rust. With C it is perfectly feasible to keep even a moderately complicated kernel like weenix's down under 4 megabytes, and in fact almost nobody has ever run into this limit during CS169's history. With rust, however, this limit was blown through almost immediately. While it is most likely that this is more to do with the rust compiler's optimizations (or lack thereof) than the language itself, the fact is that optimizations matter to any software developer. Rust's relative lack of them when compared to more established languages must be considered.

2.3 Memory & Initialization

Another early challenge I faced was getting memory allocation working. For many reasons, including feeling it was rather outside the scope of this project, I choose to make no attempt to implement a memory allocator in rust and instead use the existing weenix allocators. This led to a small problem since the weenix allocators are slab allocators, which allocate fixed-size data structures from contiguous slabs of memory. These types of allocators are actually extremely useful for kernel tasks and are used in many real-world kernels, such as FreeBSD[11] and Linux[3]. They are also commonly combined with object caching schemes[1, 6], although weenix does not use such a scheme. Since there are generally only a few structures with known sizes that need to be allocated at any one time this works quite well for most OSs.

The only problem is that rust was built with the idea that it will be used with a malloc style memory allocator. This kind of allocator is rather hard to implement using slab allocators since malloc must be able to allocate buffers of any size. One needs to create some system where the malloc function will find a suitable allocator from the many different sizes of slab allocators. There has been some debate about adding support for custom allocators to rust, which could allow one to use slab allocators easily, but this has been postponed until after rust 1.0 comes out at

[14] 'dbg!' is the debug printing macro. It prints out text to the standard output of the host computer, aiding debugging.

[15] ACPI is the Advanced Configuration and Power Interface, it controls basic power-management functions. PCI is the Peripheral Component Interface, it controls the use of peripherals such as video cards, keyboards and mice. GDT is the Global Descriptor Table, it affects how we address memory.

[16] See subsection 2.8

[17] https://www.gnu.org/software/grub/ (April 2015)

[18] https://www.gnu.org/software/grub/manual/multiboot/multiboot.html (April 2015)

least[7]. Further, the language and the compiler are also built with the idea that allocation is, in some sense, infallible. In an operating system kernel, this is an impossible standard to keep. This is, unfortunately a rather deep problem as well since the `rust` compiler has these assumptions built into it. I discuss this problem in more detail in subsection 3.3 bellow.

In order to support this I ended up needing to write a rather complicated shim around the `weenix` allocators so as to support the `rust` allocator model. I ended up making code that would search through all known allocators whenever memory was allocated and select the best one available to supply the memory. This though created problems as it would be useful to still have allocators perfectly sized for common types, to ensure good space utilization. To do this however we need to get a full list of all allocators that we will commonly be using. This required me to create a somewhat strange multi-stage initialization scheme for boot. I needed to have a first stage of initialization where allocation is not yet set up. During this phase each piece can request allocators be reserved for them, or do other startup related tasks that do not require allocation, including most initialization of the `C` parts of the project. This is added onto the other two phases of the initialization tapdance from `weenix`. Once this is done we do all the other initialization that is possible before we are running in a real thread context, then we do the final initialization once we have entered the idle-process.

2.4 Processes

In this section I will talk about how processes are created, controlled and stopped, covering the two most basic functions of a process system. Next, I will talk about how inter-thread synchronization is performed and maintained. Finally, I will examine the `reenix` scheduler and explain how it is made and behaves.

The first major part of `reenix` that is also implemented in the `CS169` `weenix` project is `PROCS`, encompassing process-control, scheduling and synchronization. I chose to follow the basic `weenix` design for my process structure. There is a separation between a process and a thread, where there is a hierarchy of processes each of which has threads of execution. Threads and processes are represented separately; each process has at least one thread. As in standard `unix`, we keep track of the parent-child relationships among processes and will move orphaned processes to the `init` processes. Processes hold information about the child processes and memory map;

```
pub struct KProc {
    /// our Process ID
    pid      : ProcId,
    /// Process name
    command  : String,
    /// Our threads
    threads  : HashMap<u64, Box<KThread>>,
    /// Our children
    children : HashMap<ProcId,
                       Rc<ProcRefCell<KProc>>>,
    /// Our exit status
    status   : ProcStatus,
    /// running/sleeping/etc.
    state    : ProcState,
    /// Our Parent
    parent   : Option<Weak<ProcRefCell<KProc>>>,
    /// Page directory
    pagedir  : PageDir,

    /// wait-queue for waitpid
    wait : WQueue,
}
```

Figure 3: The process structure in `reenix` is very similar to how it is in `weenix`, though it make use of `HashMaps` to store the children and threads instead of the interned lists of `weenix`

a slightly annotated definition of the process structure can be seen in Figure 3. If I had gotten farther on implementing it, processes would also hold information on open files and the current directory which is shared by all threads. Threads hold information about what the process is currently doing, are what the scheduler works with, may block and has a stack.

2.4.1 Control & Creation

`Reenix` has a very simple process control model. One may create a process at any time by calling the `KProc::new` function and giving it a function to be invoked by its first thread. This function returns the new process's unique id number, or a value identifying the error if something went wrong. Once this function has been called, the created process will continue running until either all threads finish or it explicitly stops itself. Currently `reenix` does not support multithreaded processes. A thread can therefore only be created by creating a process. This restriction is currently there for convenience. The design of processes and threads is made to allow one to switch to having multi-threaded processes rather easily. There is no facility analogous to `kill(2)` in `reenix`, one may `cancel` a thread or process, which might wake up a thread if it is sleeping and prevent it from sleeping

again, but there is no way to force a thread to exit, threads exit only under their own volition. Finally any process can wait for its children to exit, through the use of a `waitpid(2)` like function. Processes may wait either for any child process to exit, or for a specific one. Waiting for a process that has already exited to finish occurs without sleeping.

A major challenge in implementing all of this was simply the question of ownership of the process structures. The most obvious answer is that each process should own the process structures of all of its children. This mirrors the process tree we are creating with `waitpid` and is rather simple to implement. If we do this we need to deal with the fact that each process must keep track of its parent, to allow one to notify a parent sleeping in `waitpid`. Furthermore, it is important for usability that we not need to pass the current thread variable around all the time, therefore we need to have some way to turn an arbitrary process id into a process structure that we can cancel or query. In order to allow all of these uses we handle process structures mostly through reference counted `Rc<KProc>` pointers, with all non-owning references being `Weak<KProc>` weak references[19]. This lets us leave the owner of a process structure as its parent while still letting us access it safety since `rust` will not allow one to access a `Weak<T>` without checking that it is still present and getting a full reference to it.

A benefit of using `rust` here was that scope-based destructors allowed me to simplify the code somewhat. These destructors allowed me to define cleanup code that is run whenever an object goes out of scope, simplifying error cleanup. For example, I could generally just return an error code if creating a new process failed for some reason, knowing that all the temporary values, including the new process structure, would be destroyed. This meant I did not need to repeat cleanup actions in multiple places or use a '`goto error`' based cleanup system.

2.4.2 Synchronization

Since `reenix` is a strictly single-core operating system we can have a rather simple synchronization scheme. All synchronization in `reenix` is based around wait queues. A wait queue is a simple synchronization primitive similar to a condition variable. Anything may choose to `wait` on a wait queue and they will go to sleep until some other thread `signal`s the queue. These functions take care of masking off interrupts when they are being used, preventing interrupts from occurring while a thread is going to sleep or being woken up. Wait queues are implemented with a structure called a `KQueue`. In my implementation one may only wake-up the entire queue at once, furthermore one may optionally go into a sleep such that being canceled will cause the thread to wake up. Using this it is fairly straightforward to create all the other synchronization constructs one could desire, such as `Mutex`s, or condition variables. In order to make these structures more generic I also created a pair of `trait`s which encapsulated this behavior.

Like with process control above, implementing this form of synchronization led to some tricky questions of ownership. This is because, in the end, a wait queue is simply a list of threads that are currently paused. The wait queues clearly should not be the owners of the threads in any sense and should simply have temporary references to them. Unfortunately the `rust` lifetime system gets in our way since there is no clear lifetime we could give to the references that is consistent across all threads. This is because lifetimes in rust are always related to the call stack of the current thread of execution. `Rust` assumes that everything that any given thread can see is either (a) going to be around forever or (b) was created in some particular function call in the current call-stack and will be destroyed when we move through said frame. This makes working with references which, in some sense, live on entirely separate stacks very tricky. Since each thread has a totally separate lifetime from all of the others there is no way for `rust` to prove that the references are safe to use [20] and therefore it will not allow us to write the queue in this way. One possible solution would be to use weak references as we did with processes in the previous section, and, if I were to do this project again, I would likely do just that. Instead, I chose to exploit another one of the nice features of `rust` which is the ability to explicitly ignore some of `rust`'s safety checks. I therefore held the queued threads as simple pointers, casting them back to threads when they are removed. This solution is functionally equivalent to the naive method discussed above and is just as safe. It also conveniently avoids the extremely heavyweight nature of the weak-reference solution, by avoiding the need to keep around a reference counted pointer simply so we can get sleeping working.

[19] See subsubsection 1.2.2

[20] See subsection 3.2

2.4.3 Scheduling

The reenix OS employs a simple, First-in-First-out scheduler. It maintains a simple list of running threads. Each thread keeps track of its state in a special Context structure that holds the values of its instruction pointer and other registers, as well as the thread-local-storage data.

This naturally runs into many of the same issues that the wait queues do in terms of ownership and it solves them in the same way. One other, somewhat strange, issue I had was with the compiler optimizing away many of the checks in the loop the scheduler performs when there is no thread to run immediately. It would believe that since it had the only mutable reference to the list there was no way anything else could modify it. Unfortunately, this list is often modified by interrupt contexts and must be checked each time. While such an error could (and, with optimizations turned on, does) happen in C it is much more difficult to fix in rust. In C, one must simply mark the variable being used as volatile and the compiler will stop attempting to optimize its access. To do the same in rust, one must be sure that all reads of the variable are done through a special marker function to read the variable.

2.5 Drivers

The next major part of reenix is to add basic device drivers. I chose to follow the weenix project and have an (emulated) ATA hard disk driver and a basic TTY[21] driver stack. This allows one to have a (small) amount of interactivity with the OS, easing testing and analysis and also allow one to have persistent storage at some point in the future. Furthermore, both these drivers are rather simple, well understood, and well documented, easing the process of making them from scratch.

In order to make the use of drivers easier in other parts of the kernel I created a basic trait to encapsulate their basic functions. They can be seen in Figure 4. They nicely abstract the way one uses input/output devices inside the kernel. Furthermore, they illustrate a nice feature of rust in that one can reimplement the various Device traits multiple times with different type arguments. This is extremely useful with things like memory-devices such as /dev/zero, which one might want to either read whole blocks of memory from or read character by character.

This section will explore the implementation of each of the drivers I created for reenix. It will also examine any of the problems I faced while creating them.

2.5.1 TTY Stack

The TTY subsystem in reenix is split into five separate parts. At the lowest level is the keyboard driver, which allows one to receive input from the outside world, and a simple display driver, letting one print characters for the outside world. The display driver is a simple VGA mode 3 text video interface, avoiding the trouble of creating a real video driver that needs to draw everything to the screen. The keyboard driver is implemented as a standard PS/2 keyboard and supports all of the standard keys and some of the meta-keys. Both of these are implemented fully in rust but are rather simple ports of the C versions included with weenix.

Using the screen driver I created a simple virtual terminal driver. A virtual terminal is a common unix abstraction that represents a virtual screen with a keyboard. These virtual terminals implement scrolling and a cursor that keeps track of the next place to write to. This is another mostly straight port from C to rust.

Finally, above virtual terminal and keyboard driver I created the TTY driver and line-discipline. These are the actual drivers used in running a TTY. The TTY driver receives interrupts from the keyboard, passes them to the line discipline, which might process and record them, then echoes them out to the screen. When the tty is written to it uses the line discipline to translate the characters so they can be displayed and then passes them to the virtual terminal. When the tty is read from the next full line is returned, or the current thread goes to sleep until a full line is typed.

There were no major challenges that needed to be dealt with regarding the reenix design during this part. About the only somewhat difficult challenge was that I was unable to find any clean way to switch between different virtual terminals. I ended up having to just update a pointer so that the interrupt subsystem could know which TTY to send input to. Though this is also a problem in the C version, the fact that this was the best way to solve this prob-

[21] TTY (sometimes called tele-typewriters or text terminals) are a common interface for unix-like operating systems. They provide users with an area where they type commands and other data as text and the programs may write text in response.[6]

```rust
/// A device capable of reading in units of `T`.
pub trait RDevice<T> {
    /// Read buf.len() objects from the device starting at offset. Returns the number of objects
    /// read from the stream, or errno if it fails.
    fn read_from(&self, offset: usize, buf: &mut [T]) -> KResult<usize>;
}

/// A device capable of writing in units of `T`.
pub trait WDevice<T> {
    /// Write the buffer to the device, starting at the given offset from the start of the device.
    /// Returns the number of bytes written or errno if an error happens.
    fn write_to(&self, offset: usize, buf: &[T]) -> KResult<usize>;
}

/// A Device that can both read and write.
pub trait Device<T> : WDevice<T> + RDevice<T> + 'static {}

/// A device capable of reading and writing at byte granularity.
pub type ByteDevice = Device<u8>;

/// A device capable of reading and writing at block granularity.
pub trait BlockDevice : Device<[u8; page::SIZE]> + MMObj {}
```

Figure 4: The `Device` traits encapsulates a hardware device capable of reading and writing fixed size data

lem I could come up with is disappointing, since it is very much something that `rust` discourages. Actually switching the interrupt handler would be another possible solution but doing so seemed incredibly heavyweight, especially since the different functions would be almost identical. Furthermore, I am actually unsure what the implications of changing out interrupt handlers while handling an interrupt would be, and, as far as I am aware, no operating system changes out interrupt handlers as often as this would have to.

2.5.2 ATA Disk

The `ATA` disk driver in `reenix` is a fairly simple block-device driver. It is capable of reading and writing specified blocks to the disk, including performing multiple consecutive reads and writes at once. To do this the driver makes use of direct memory access (DMA). When using DMA the driver writes to specific memory locations which the memory bus uses to send instructions to the disk[6]. The disk then performs the specified operation, loads the result into a (user specified) memory location, and triggers an interrupt on the CPU.

As this whole process is fairly straightforward there are few differences between my `rust` version and the `C` version that one creates in `CS169`. The main differences being that `rust` allowed me to move away from somewhat arcane macros in controlling the DMA device.

Other than figuring out what the `C` code was doing in order to replicate it, the main difficulty I had was `rust`'s relative lack of alignment primitives. To use DMA one needs to deal with a data-structure called the Physical Region Descriptor Table (PRD). This table tells the PCI bus where the memory that the DMA is going to be using is. Because this table is so inextricably linked to the hardware it has very strict alignment requirements; it must be aligned on a 32 byte boundary. In `C` this is not a very large problem; one can simply statically allocate the table with an attribute about its alignment, or just dynamically allocate a buffer using an aligned allocator or just enough space to guarantee one can get the alignment. In `rust` this is a much more challenging problem for several reasons. First, there is no good way to force alignment of data in `rust`, the current recommendation is to put a zero-length array of a SIMD type to emulate it[13]. Second, the `reenix` allocators cannot guarantee alignment in any meaningful sense so it is a moot point since I could not get good alignment anyway. I could allocate these statically but I have been trying to avoid doing this as `rust` does not make doing so simple. Furthermore, since the CPU I am compiling for lacks support for SIMD types I am unsure if even the recommended approach would work. In the end I chose to over-allocate and manually align the data whenever I use the PRD table,

which is rather cumbersome, error-prone, and complicated.

2.6 KShell

Once I had the TTY driver finished one of the first things I worked on was a simple command shell, which I called the `KShell`. This allows one to interact with the operating system in a way that was previously impossible, dynamically selecting tasks using the keyboard. With the `KShell` it became so much easier to test and examine the operating system. It is difficult to overstate how useful this is for testing, suddenly it was possible to simply experiment with different sequences of commands without recompiling. Furthermore, without the `KShell` it would have been much more difficult to test the TTY driver. Finally, since I have not yet implemented any form of userspace, a kernel-space shell is the only possible way we could interactively control the system.

Like most command shells, the `KShell` is based on a simple read-evaluate-print loop. There are a number of commands that the `KShell` is able to use, including a `parallel` command to run other commands in their own threads. Each of the commands is written as a `rust` function which runs the command and returns a value indicating whether it succeeded.

One of the most interesting things about the `KShell` is how similar it is to a generic REPL shell written in `rust`. In many ways it is implemented in almost exactly the same way one would in normal `rust`; the types are the same, the loops are the same, etc. This is rather interesting since it shows just how much of the `rust` standard library can be used without modification in `reenix`. For most higher-level languages, the lack of a real runtime when running in kernel mode would severely constrain the sort of code one could write here.

Furthermore, while writing the `KShell` it struck me how much easier writing it in `rust` seemed to be than doing the same thing in `C`. By writing it in `rust` I was able to take advantage of features such as easy-to-use list and mapping types and being able to make use of `rust`'s high level string routines. These removed some of the more annoying aspects of creating a working command shell that would have been present in `C`.

2.7 Virtual File System

Once I had finished working on getting drivers working I next started to work on creating a virtual file system for `reenix`. A virtual file system (VFS) is a common abstraction in `unix`-like operating systems first created and popularized by Sun in the mid-1980s[8]. It is designed to smooth over the differences between various on-disk and network file systems. The VFS defines common interfaces for all the major file system manipulation and interaction routines, such as searching a directory, creating a file or directory, or opening, reading, and writing to files. It also commonly defines a data block cache to allow the caching of reads to underlying block devices by the file systems underneath it.[22] Both `weenix` and all major `unix`-like operating systems today make use of a VFS-like interface in order to ease the use and creation of file systems.

Unfortunately, due to time constraints, I was unable to get very far in implementing a VFS for `reenix`. In order to allow incremental testing of components I decided to follow the example of `weenix` and implement a rather simple, in memory, file system, called `RamFS`, to test some of the VFS specific features before moving on to implement the much more complicated `S5` file system. I was, however, able to implement parts of the block-device and caching layer that `S5FS` would have used to access the disk, and create a reasonably complete `RamFS`.

This section will examine each of the three parts of `VFS` that I was able to at least get a start on completing. We will first look at the design of the `VNode` trait and the decisions that went into it. Next, we shall look at the `RamFS` test file system, which I was able to mostly complete. Finally, we will examine the pageframe, memory object, and their attendant caching system which I was only able to get a small start on.

2.7.1 VNodes

The major data type in the virtual file system is the `VNode`. A `VNode` is the `VFS` representation of what a file or other file system object is and what it can do. It consists mainly of a large number of functions that will do the main file system operations on the `VNode`. Since it is mostly defined by the functions it implements, in `reenix` I made the `VNode` be a trait. `VNode`'s generally have multiple, concurrent, owners who all make use of the underlying file data. In `weenix` this is implemented by performing manual reference counting on the `VNode`s, which is one of the most difficult and error-prone parts of the assignment.

A part of the `reenix` `VNode` trait can be seen in Figure 5. One thing to note about this is that I make

[22] In `weenix` this cache is actually implemented by students during the `S5FS` project that follows `VFS`[12].

```rust
pub trait VNode : fmt::Debug {
    /// This is only here so that the type system works out. Needed b/c no HKT
    type Real: VNode;
    /// What type of vnode operations will create/get. This must be clone.
    /// We want to say it is borrow so that we can have this be a wrapper that deals with
    /// ref-counting.
    type Res: Borrow<Self::Real> + Clone;
    fn get_fs(&self) -> &FileSystem<Real=Self::Real, Node=Self::Res>;
    fn get_mode(&self) -> Mode;
    fn get_number(&self) -> InodeNum;
    fn stat(&self) -> KResult<Stat> { Err(self.get_mode().stat_err()) }
    fn len(&self) -> KResult<usize> { Err(self.get_mode().len_err()) }
    fn read(&self, _off: usize, _buf: &mut [u8]) -> KResult<usize> {
        Err(self.get_mode().read_err())
    }
    // ...
    fn create(&self, _name: &str) -> KResult<Self::Res> { Err(self.get_mode().create_err()) }
    fn lookup(&self, _name: &str) -> KResult<Self::Res> { Err(self.get_mode().lookup_err()) }
    fn link(&self, _from: &Self::Res, _to: &str) -> KResult<()> { Err(self.get_mode().link_err()) }
    // ...
}
```

Figure 5: The `VNode` trait

use of `rust`'s type system to represent the fact that we will often not be dealing with `VNode`s directly but instead with wrappers that handle reference counting for us. We say that the result type of functions like `create` must be usable as some type of `VNode`. This allows us to have each separate type of file system object[23] implement `VNode` and yet return a standard `enum` that might be any of these file system objects. By requiring that this result type is copyable (the `Clone` trait), we hide how the backing file data is shared between `VNode`s which reference the same object, allowing the reference counting to take place behind the scenes. We can simply return a type that is wrapped by some reference counter that makes sure we remove the `VNode` when it is no longer referenced. For example in `RamFS` the result type is `Rc<RVNode>`. This prevents a whole class of errors that is one of the trickiest parts of VFS in `weenix`, by removing the need for manual reference counting.

2.7.2 RamFS

`RamFS` is the name of a testing-focused, in memory, file system used to test ones VFS implementation in `weenix` and `reenix`. This "file system" implements all the calls needed to be used by VFS without actually being backed by a disk. In `weenix` this system is provided as support code to enable one to test VFS without having to be concurrently working on creating the `S5` file system.

I actually made several attempts to implement `RamFS` before coming up with its final design.

My initial design was to do a straight port from `C` to `rust`. I could not simply use a stubbed version that called directly into the already finished `C` code because it relied on a lot of the other `weenix` support code, such as the `VNode` representation. This turned out to be much more difficult than I had thought it would be.

The `C RamFS` is mostly implemented just as a normal file system is, but with a massively simplified system for keeping track of allocated file nodes and no system for allocating blocks of memory (all files and directories get exactly one block). This means that, for example, directories are implemented as an array of bytes which is cast to an array of directory entries. While this does mean that it is very similar to real file systems in this way, it is a problem since `rust` does not make working with raw, untyped, storage as simple as `C` does. In `C` one would simply cast the array and work with it as the array of (possibly uninitialized) directory entries that it is, doing so in `rust` is rather cumbersome. While this problem would need to be handled eventually with `S5FS` (probably through the use of a binary-serialization library) I decided that, in the interests of time, I would take a simpler route.

My second design for `RamFS` was to create as simple of a mock file system as I could. I abandoned

[23] For example, directory, file, device, etc.

any notion of creating something similar to a real file system in implementation. Each type of file system object is a different type, each directory is simply a map from file-names to nodes, reference counting is done through the standard-library `Rc<T>` type and so on. This did make the implementation far simpler, allowing me to quickly finish most of it in a short time. Unfortunately by the time I had gotten this done there was little time to work on the rest of VFS beyond what I had already done. It was during this that I made the final design decisions surrounding how `VNodes` work.

2.7.3 PFrames & MMObjs

When I began working on VFS I had initially not planned to create the `RamFS` file system and had therefore started by creating much of the support code that is needed by `S5FS`. This support code contains the structures necessary to implement the block-cache needed by most file-systems, which `RamFS` does use. These systems are often considered a part of the virtual file system subsystem by operating system kernel projects, although in `weenix` they are not implemented until the `S5FS` project. This system is made up of two major components, `MMObj`s, which are an abstraction of a data source that can provide (cacheable) pages of data, and `PFrame`s which are pages of data from an `MMObj` which can be cached and updated. A `PFrame` is conceptually owned by its source `MMObj` and is, in some sense, simply a current view into that `MMObj`. In fact though, `PFrame`s are cached separately from `MMObj`s, and the `MMObj` is mostly responsible for simply filling in the `PFrame` and writing the data back when the `PFrame` is destroyed. In `reenix` these pieces are not fully completed, although the interfaces they expose are fully written.

The basic `MMObj` trait can be found in Figure 6. One interesting thing to note about this design is that none of these functions is ever called directly by file system driver code other than the `PFrame` system. When a file system needs a page of memory from an `MMObj` the basic call structure would be to request a `PFrame` of the given page number on the `MMObj`. This will first search the global `PFrame` cache to see if it is already there, returning it if so. Otherwise it will create a new `PFrame` and fill it with data using the `MMObj`. The caller may then make use of the `PFrame` however it wants since they are reference-counted preventing them from being removed while in scope. When a `PFrame` goes out of scope it will check to see if it is reachable from anywhere, either as a reference or from its `MMObj`. If it is not (or if there is little memory left) it will check to see if it has been modified and if so have the `MMObj` write it back.

The most important challenge I faced while creating this was convincing myself that the call structure is actually what I want. For the first time many calls are moving through traits that hide implementation, making it difficult to determine what is going to be happening. Another challenge I have been having is that this system is very difficult to test in isolation.

2.8 Other Challenges

As I was working on `reenix` there were many challenges and decisions that I needed to make that were not part of any single piece of the project. These challenges included implementing `rust` stack-overflow detection, implementing thread-local-storage and simply getting the project building at all. The difficulties I had in doing these, and the need to do them at all, are mostly due to the differences between `C` and `rust`.

2.8.1 Stack Overflow Detection

One of the nice features that `rust` has over many other (commonly) compiled languages such as `C` is it supports built-in stack overflow detection. This means it is able to detect stack overflow even if there is no hardware memory protection available. `Rust` uses the `LLVM`[24] provided method of doing this by having all subroutine calls check a thread-local variable that holds the address of the end of the stack. On each subroutine call the function use this to check if there is enough space on the stack to hold all of its local variables. Specifically, it will check the value at offset `0x30` within the `%gs` segment[25] on `x86` systems.

Since this feature makes use of (implicit) thread-local variables, it requires specific runtime support and setup to work.

[24] `LLVM` is a compiler project and toolchain that provides a standard intermediate representation which may be targeted by other compilers[9]. This is then turned into binary code for several different machine architectures. It is used by the `rust` compiler to create the final binary code for a `rust` program.

[25] Memory segmentation is a method of addressing on `x86` and related processors. It was originally developed to allow 16 bit processors address up to 24 bits of memory. It works by having segments that start at a particular address and extend for a specified number of bytes. All memory accesses are performed relative some segment, either specified in the assembly code or a default one. The segment definitions are held in the global descriptor table.

```rust
pub trait MMObj : fmt::Debug {
    /// Return an MMObjId for this object.
    fn get_id(&self) -> MMObjId;

    /// Fill the given page frame with the data that should be in it.
    fn fill_page(&self, pf: &mut pframe::PFrame) -> KResult<()>;

    /// A hook; called when a request is made to dirty a non-dirty page.
    /// Perform any necessary actions that must take place in order for it
    /// to be possible to dirty (write to) the provided page. This may block.
    fn dirty_page(&self, pf: &pframe::PFrame) -> KResult<()>;

    /// Write the contents of the page frame starting at address
    /// pf.page to the page identified by pf.pagenum.
    /// This may block.
    fn clean_page(&self, pf: &pframe::PFrame) -> KResult<()>;
}
```

Figure 6: The `MMObj` trait

```
c0048f10 <kproc::KProc::waitpid>:
# Compare used stack with end of stack
c0048f10: lea    -0x10c(%esp),%ecx
c0048f17: cmp    %gs:0x30,%ecx
# Continue if we have enough stack space
c0048f1e: ja     c0048f30
                 <kproc::KProc::waitpid+0x20>
# Save meta-data about stack
c0048f20: push   $0xc
c0048f25: push   $0x10c
# __morestack will abort the current process.
# The name is a remnant from when rust supported
# segmented stacks.
c0048f2a: call   c000975c <__morestack>
c0048f2f: ret
# Standard x86 function prelude
c0048f30: push   %ebp
c0048f31: mov    %esp,%ebp
c0048f33: push   %ebx
c0048f34: push   %edi
c0048f35: push   %esi
c0048f36: sub    $0xfc,%esp
```

Figure 7: The disassembled function prelude from `KProc::waitpid`.

A copy of the standard `rust x86` function prelude can be seen in Figure 7. There we can see that the function will first calculate the farthest point on the stack it will use on line 3. Then it loads the end of the stack from the `%gs` segment and compares it to the required stack space on line 4. Finally, on line 6 it either jumps to the actual function or calls the `__morestack` function to abort the thread.

Unfortunately, `weenix` has no support for this type of stack checking so I needed to implement it myself if I wanted to use it. Initially, I had thought I could just disable this functionality, since `weenix` works perfectly fine without it. Unfortunately, however, at the time that I started this project there was no way to remove this stack checking code from functions globally. This meant that in order for my code to run, I had to support the stack checking method used by `LLVM`. Doing this was actually somewhat less difficult than I feared it might be. Using the Global Descriptor Table (`GDT`) manipulation routines provided by `weenix`, I was able to write a data-structure that would store the stack endpoint at the appropriate offset. This data structure can be seen in Figure 8.

From there I still had some small amount of work to do. I next needed to make sure that there was no call with stack-checking enabled until we had successfully set up the `%gs GDT` segment, which was actually somewhat tricky. Next I needed to ensure that during process switching we would correctly change the value of the `%gs` descriptor and, again, make no calls with stack-checking enabled until we had successfully switched processes. Once this was accomplished all that was left was to wire up the error-reporting functions, which was mostly made difficult by the relative lack of documentation. Ironically, less then a week after I had finally finished getting all of this stuff to work, the `rust` developers added a switch to the `rustc` compiler that let one disable the stack checking completely. Still, the experience of implementing this was very informative and helped me implement the thread-local storage described below. It also did legitimately help with doing the rest of the project as it helped me find several accidental infinite recursions

```
#[cfg(target_arch="x86")]
#[repr(C, packed)]
pub struct TSDInfo {
    vlow : [u8; 0x30],
    stack_high : u32, // At offset 0x30
    /// The other thread specific data.
    data : VecMap<Box<Any>>,
}
```

Figure 8: The TLS data structure

in my code.

2.8.2 Thread-Local Storage

Once I had finished getting the stack-overflow detection in `rust` working I realized that I had most of a thread-local storage system. Since there was so little of it left to do I decided to implement the rest of it. There is no comparable system in standard `weenix`. To do this I simply added a `VecMap` onto the end of the structure holding the stack data.

I chose to use this thread-local storage to hold some information that is held in static variables in `weenix`, specifically the current process and thread. By storing this data in this manner it also removes one of the obstacles to making `reenix` usable on multi-processor machines, although there are still a number of other issues with doing this. The thread-local data storage structure is shown in Figure 8. Note that it holds the thread-local data in a `VecMap<Box<Any>>`. The `Box<Any>` type is a special type that uses limited runtime type information to perform checked, dynamic casts. This lets me store arbitrary thread-local data in this map without having to know exactly what it is, while the users of said data may still check and make sure that the data they receive is the correct type. For example, the currently running process is stored using in this structure. This can occur despite the fact that the TLD structure has no notion of what a process structure is. When the current process is retrieved the user must manually check, using the `Any trait`, that the returned value is, in fact, a process.

2.8.3 Build System

One last unexpected challenge that I had while writing `reenix` was getting it to be built at all. The standard way to build `rust` code is to use its custom build tool, `cargo`. This tool is perfectly sufficient for most normal `rust` projects, it even includes fairly good support for linking to external libraries or invoking other build tools. Unfortunately, the tool is very difficult to use if one needs to perform more complicated linking or build operations. Therefore I had to figure out how to create a makefile that would work with `rust`.

This turned out to be somewhat difficult because I needed to make sure that `rustc` only uses the versions of the standard library and other libraries that I built. In addition the dependencies between crates are much more complicated than normal `C` dependencies, depending on the libraries created by other crates. Furthermore there are two slightly inconsistent naming conventions in use in `reenix`, the `weenix` naming conventions and the `rust` standard library naming conventions. All this ended up meaning that I had to create a rather complicated set of makefile macros that creates the rather large rules dynamically, using the `rustc` diagnostic functions to get the names correct.

2.9 Future Work

There is still much work to be done before `reenix` is at a point that could be called complete. The most obvious first step is to complete the remaining three parts of the `weenix` project; `VFS`, `S5FS` and `VM`. Doing this will require solving several design questions for `reenix`. Some major open questions are how `reenix` should keep track of open files, how we should interact with the serialized binary data-structures that make up the `S5` file system, and how we should keep track of memory mappings.

Furthermore, there are still large parts of the `weenix` support code that must be rewritten in `rust`. Major components that must be ported include the file system test suite, the actual `syscall` handler routines, the ELF file loader and a time slice system. Each of these are important, though ultimately rather simple, pieces of an operating system kernel and are very specific to the architecture of the kernel as a whole, preventing one from simply using the `weenix` versions of these components.

Finally, a more permanent solution should be found with respect to the problem of `rust` memory allocation failure. This problem is discussed in depth in subsection 3.3 below. Without this, getting the system to be truly stable and usable would be difficult at best and impossible at worst. Once these have been accomplished it should be possible to continue to experiment on `reenix` in even more interesting directions, such as possibly adding networking, SMP, or 64-bit support.

3 Rust Evaluation

As mentioned in subsection 1.2 rust is a fairly new, systems-oriented programming language with a focus on memory and type safety. Its quite comprehensive type and memory analysis make it seem like it would be a worthy replacement for C in a world which increasingly demands both speed and safety. Creating an operating system is something nobody has made any serious attempt to do with rust before, and through implementing reenix I gained a great appreciation for rust's strenghts and weaknesses in this arena. I was also able to see some places where rust could be changed to improve its usefulness in this field.

Here I will talk about some of the benefits of using rust that I found throughout the project. Next, I will examine some of the issues with using rust, then looking at the one truely major issue with the language as it stands today. Finally, I will end with a short evaluation of rust's performance in comparison to C.

3.1 Benefits of Rust

Over the course of this project I found that there were many benefits to using rust over C or C++ for systems programming. These range from simple matters of convenience and clarity to major improvements in usability and ones ability to create correct code.

3.1.1 High Level Language

One of the most obvious benefits of using rust over languages like C in operating systems development (and in general development as well) is the fact that it has many of the high-level language constructs programmers have come to expect from languages such as C++, Java, Python, and others. Rust gives one standard programming abstractions such as methods attached to objects, allowing a clearer association of data with its methods, (sane) operator overloading, and a module system to allow one to group related functionality together in a single name-space. All of these allow one to be much less ambiguous and verbose by creating a much stronger conceptual link between related pieces.

More important than simple syntactic sugars and conveniences, rust's more high-level nature also has some major effects on how we use the language. Through traits, rust gives one automatic use and creation of virtual method tables (vtables) making the use of interfaces that much easier. It will even perform the casts that are necessary in such functions in C to make the definition of vtable methods much less arcane.

```
1 typedef struct vnode_ops {
2     // ...
3     int (*stat)(struct vnode *vnode,
4                 struct stat *buf);
5     // ...
6 } vnode_ops_t;
```

Figure 9: The C stat function interface.

Even better, rust is smart enough to be able to compile code that omits the vtable lookup when it can prove that it knows the exact type being used beforehand. This allows one to easily make use of interfaces in many more places than before, creating a much more general system than is usually possible in C.

Rust has automatic scope-based object destruction semantics and support for custom destructors based on the type of the destroyed object. This allows one to make use of Resource Acquisition Is Initialization (RAII)[26] semantics. We can use this to ensure that we do not ever, for example, access synchronized data without locking its associated mutex, or fail to release the mutex when finished with the data it protects. Furthermore this can be (and is) used to create reference counting pointers that we are able to guarantee will always have accurate counts. This greatly simplifies the handling of failure in most functions, since when a function fails we can simply return an error and the rust compiler will ensure that the destructors of all initialized objects are properly run.

Finally, rust will automatically rewrite the function signatures slightly to avoid copying large amounts of data when possible. This means that rust will often rewrite the signatures of functions that return large objects to instead use out-pointers. By doing so it allows the caller to determine the placement of the returned object, preventing a memory copy to, for example, move the returned object onto the heap. The fact that rustc will perform this rewrite transparently greatly increases the readability of the rust source code and reduces ambiguity as to what a function actually does.

[26] RAII is the name of a set of programming semantics where by creating a value one automatically initialize it as well, and similarly deinitializing it destroys the object. This prevents the use of uninitialized data.

For example, compare the `stat` function seen in Figure 5 with the `C` version seen in Figure 9. Both functions semantically return a '`stat`' structure or an error. In `C` though we must change the signature to take in the '`stat`' structure as an out-pointer to avoid an expensive memory copy operation. Unfortunately, this obscures the meaning of the function to casual observation. The `rust` version of this function, however, contains the actual semantics in its type signature, returning a '`Stat`' or an error. The `rust` compiler, when compiling this function, will change its signature to match the `C` version, in order to avoid the same expensive copy operation. This means that the `rust` code will be of comparable speed but that its semantics are much clearer than `C` code which does the same thing.

3.1.2 Type, Lifetime and Borrow Checker

Another major benefit that using `rust` brings to operating systems developers are its very comprehensive type, lifetime, and borrow checking systems. Together these three tools work to help eliminate entire classes of errors from code. These systems allow one to express the meaning of code in a much richer way than is possible in `C`.

As mentioned in subsection 1.2, the `rust` language is type-safe and procedural. To assist with writing correct, type-safe code the standard library provides several wrapper types. Two of the most important of these wrapper types are the `Result<R,E>` type, which is either the successful return value of an operation (of type `R`) or a value indicating what when wrong (of type `E`), and the `Option<T>` type, which is either a value of type `T` or `None`, indicating that there is no value of that type. These are used to indicate that an operation might fail, or that a value might not be present. These types are used all the time in `rust` code, in fact one might notice that in the interfaces I have shown above (see Figure 4, Figure 5 and Figure 6) most functions return a `KResult` object, which is simply an alias for `Result<T, errno::Errno>`. Through the use of these types we can more concretely communicate exactly what the return values of functions mean, even without comments to explain them. Furthermore the type system will prevent one from making many of the mistakes that are so common in `C` when dealing with functions like this. In `rust` it is impossible to access the result of a function that returns a `Result` object without explicitly making sure one did not get an error, which must be handled somehow. This makes the semantics of, for example, the `stat` function in Figure 5 almost totally explicit and checked by the type system, while the `C` version in Figure 9, despite having the same semantics, is much less explicit and not checked at all.

In addition to the type checker, the borrow checker and lifetime system also help to prevent many common types of errors, as well as to make code somewhat clearer. The lifetime system requires that all pointer references one have are provably safe, and do not point to uninitialized, or possibly freed data. Sometimes obtaining this proof requires that one provide annotations which make explicit the relationships between different types and their lifetimes. Furthermore it also ensures that no pointers may be kept after the object they point to has been freed. Together they prevent almost all "use after free" bugs and "uninitialized data" bugs from occurring. Additionally, the lifetime system forces one to specify precisely in the type signature the dependencies between different types with respect to how long they are valid when holding pointers. Finally, the borrow checker prevents one from concurrently modifying data structures, using the lifetime system to verify the dependencies between them. This prevents bugs where one piece of code invalidates the pointers held by another piece of code.

3.1.3 Macros & Plugins

Another benefit of using `rust` over `C` is the fact that `rust` has a very comprehensive and powerful macro and plugin system. As mentioned in subsubsection 1.2.1, `rust` macros are hygienic transformations of the `rust` abstract syntax tree (AST). These are very different, and much more powerful, then the pure text transformation macros that are available through the `C` preprocessor.

With `C` macros one must always be aware that macros are expanded in the same context in which they are used. This means that in `C`, macros might overwrite or redefine variables if the author is not careful. In `rust`, macros will never overwrite any local variables that are not explicitly passed into the macro.

The standard way to define macros in `rust` is to use a pattern-matching Domain Specific Language (DSL). When using this DSL, a macro's arguments have basic type information. You can specify whether each argument is one of a small number of types, including expression, statement, type, identifier, and around six others. These mean that one does not gen-

erally need to place extra parentheses or other such delimiters to make sure that the macro parses correctly. Furthermore it means that one can have quite powerful pattern matching when expanding macros, creating different code depending on the exact format or number of the arguments, something nearly impossible with the `C` macro system.

In addition to the already powerful macro system, the `rust` compiler, `rustc` , also supports the creation of compiler plugins that change its behavior[5]. These plugins can be used to implement much more complicated macros that are impossible (or at least very difficult) to create using the standard macro DSL. For example, one can make use of quasi-quote operations and access to the file system only within compiler plugins. One can even use a compiler plugin to create an entire new DSL if one wishes (and, in fact, this is how the macro DSL is implemented). Furthermore these can be used to create tags similar to the `#[derive(...)]` seen in Figure 2, which modify the item they are in front of. Finally they can be used to create lints that can be used to check the AST for various kinds of errors and report them. All of these abilities are extremely useful to all developers, and thanks to the fact they are purely compile-time plugins they can all be used with kernel development.

3.1.4 Growing Ecosystem

One last benefit `rust` has over `C` is that `rust` has a fairly robust and modern standard library, most of which can be used with little effort in kernel development. The `rust` standard library has everything one would expect from any other high-level language. It has multiple types of containers, including lists, maps, and sets, it has reference counting smart-pointers, and a reasonably good string module. The `C` standard library, on the other hand, is mostly unusable in a kernel setting because it is largely an interface to kernel features, such as pipes, files, signal handling, and memory-mappings. While there are many parts of the `C` standard library that may be used in a kernel context, these parts are far more limited and difficult to use then their `rust` counterparts. Furthermore, there are many common data-types not included in a `C` standard library present in `rust`s, such as list types and maps.

Furthermore many community `rust` projects can be used with little to no modification since they mostly will use the standard library. This means it is fairly simple to include packages from a multitude of sources, which can be very useful when making something as complicated as an operating system. For example, for much of my time creating `reenix`, I made use of an external compiler plugin that would let me write tests to be run with a function. This plugin was very useful and helped me uncover multiple bugs related to how I was performing system initialization. Unfortunately, I was unable to make use of too many community projects in `reenix`. This is mostly due to the fast pace of changes `rust` was going through when I was creating `reenix`, which meant that projects would quickly stop working with the latest versions of `rust`. As `rust` moves past its first stable release (currently planned to occur on May 15, 2015) this will likely become less of an issue as developers can target a single, stable, version of the language.

3.2 Challenges of Rust

While creating `reenix` there were many challenges that I faced that were mainly caused by the `rust` compiler. These were mostly cases of the language not providing the tools I needed to solve the problem in the way I believed was best. While none of these problems prevent `rust` from being used as a language for operating systems development, and while I was able to work around them, I do feel that the language would be improved if they could be addressed.

3.2.1 Structure Inheritance

One challenge that I had with implementing `RamFS` was the lack of any form of inheritance among structures. In many cases there are multiple types which require access to different functions but hold much the same data. A perfect example of this is the `VNode` shown in Figure 5. You might notice that several of the methods are simply accessors, like `get_fs` and `get_mode` on lines 8 & 9. Both of these methods are there simply because these are fields that all `VNode`s, regardless of implementation, should have in order to work. Indeed, all of the implementors of `VNode` have a pair of fields that are simply the file system they come from and the type of node they are.

It would be much simpler if there was an abstract base class that had all the fields that will be common to `VNode`'s, and then subtypes could add to these fields and implement some of the class's methods. This idea is not new to the `rust` community and, indeed, more than half a dozen proposals to add this and similar forms of inheritance have been proposed, although all have been put on hold pending the release of `rust 1.0`.

3.2.2 Anonymous Enumerated Data Types

While implementing RamFS, another problem that I ran into was that I had to present a single type as being the RamFS VNode to the rest of the kernel. This is because I needed to make sure that all the VNode creation methods had the same type, or else rust would not have been able to type-check them. However, internally it made a lot more sense to have each different type of VNode be its own separate type, implementing the methods that specific type of VNode needs. Furthermore, doing it this way allowed me to have almost all of the VNode methods be optional and return the error expected if they were used on a VNode of the given type, as seen in Figure 5.

In order to meet this restriction, I created an enumeration manually that could hold all of the separate variants of RamFS VNodes. This enumeration then had all the VNode methods implemented for it by simply passing the arguments down to the actual VNodes it held. All the other types of VNodes then declared themselves as returning this enumeration in their create and lookup methods.

This worked but, given how many functions the VNode trait has, it was very time-consuming to code. One way to improve on the ergonomics of this would be to allow some sort of anonymous enumeration type in rust. If such a type were able to automatically implement and pass down, shared functions of its constituent types this sort of filler code could be avoided. Furthermore, this type would be guaranteed to have a known size at compile time, allowing it to be used without pointer indirection, as returning a trait would require. This would make presenting these collections of related data-structures as one type much easier.

3.2.3 Static Ownership

One last area where rust could be improved is in its handling of statically allocated data. In an operating system kernel there is a lot of data that is global and shared by all processes. This includes the scheduler's queue, the list of loaded drivers, the file systems, the cached pages of memory, and many other things besides. As this is all global data, and there is only a single copy in existence, it would make sense to have all these data-structures be held as statically allocated Option<T>[27] with the default set to None. This would clearly show in the types that the value being held is statically owned by the whole system and that it is initially not initialized, requiring setup before use.

Unfortunately, many of these, such as a Vec of active processes, or the collection of initialized device drivers, are data-structures that have special destructor code to make sure that its contents are properly de-initialized and its backing memory is destroyed.

Currently rust does not allow such structures to be allocated statically. Presumably this is because it does not have any way to ensure that the destructors are run when the program exits. While this does make sense it seems rather strange not to offer a way to tell the compiler to ignore this detail, since it would often be the case that we would not care if the destructors are not run on shutdown.

The workaround that I used in reenix was to dynamically allocate all of these data structures. I would then store them as a static pointer which was dereferenced whenever I wished to use these structures. While this worked, it had the unfortunate consequence that it took up heap space that could be better used for other things. Furthermore it is likely that the heap-allocated data was less efficiently packed then the static data would have been, which ate up even more memory. Finally, since dereferencing an arbitrary pointer is never a safe operation (as it might point to uninitialized, illegal, or otherwise reserved memory), the language is unnecessarily hampered in its ability to verify the kernel.

3.3 Critical Problem: Allocation

While many of the problems that I faced while using rust could be considered relatively minor there is one issue I had with the language that absolutely must be addressed before it can be used as a serious language for operating system development. The issue is with the semantics of heap-memory allocation in rust, more specifically what happens when allocation fails.

3.3.1 Allocation In Rust

In rust heap memory is represented by a construct called a box. An object that resides in heap allocated memory is called a boxed object. Boxes can be created by placing the box keyword[28] before the value to be stored on the heap, transferring ownership of the value to the newly created box. The box key-

[27] See subsubsection 1.2.2

[28] One may also use the Box::new function to create a heap allocated pointer, however this is inlined to create code identical to just using the box keyword.

```
1  #![feature(box_syntax)]
2
3  /// A buffer
4  struct Buf {
5      val: [u8;16],
6  }
7
8  impl Buf {
9      /// Create a buffer
10     #[inline(never)]
11     fn new() -> Buf {
12         Buf { val: [0;16], }
13     }
14 }
15
16 fn main() {
17     let x = box Buf::new();
18     // Do things with the buffer ...
19 }
```

Figure 10: A program using heap allocation.

```
1  main:
2  # Stack check function prelude removed.
3  # See Figure 7  for details.
4  # Reserve stack space for function.
5          subl    $56, %esp
6  # Pass the align argument to exchange_malloc
7          movl    $1, 4(%esp)
8  # Pass the size argument to exchange_malloc
9          movl    $16, (%esp)
10 # Call exchange_malloc.
11 # Note that no null check is performed on the
12 # returned pointer.
13         calll   heap::exchange_malloc
14 # Store returned pointer on the stack
15         movl    %eax, 36(%esp)
16 # Pass the returned pointer as first argument
17 # to Buf::new.
18         movl    %eax, (%esp)
19 # Call Buf::new. Note this returns nothing.
20 # The first argument is an out pointer.
21         calll   Buf::new
22 # 36(%esp) is now a pointer to an initialized
23 # Buf object.
```

Figure 12: The (cleaned up) assembly created when compiling Figure 10.

word handles both the allocation of heap space and the initialization of it for the programmer.

The standard recommendation in `rust` is to never write a function that directly returns a boxed object[5]. Instead, the function should return the object by value and the user should place it in a box using the `box` keyword. This is because (as mentioned in subsubsection 3.1.1) `rust` will automatically rewrite many functions returning objects to instead use out-pointers to avoid a copy.

In Figure 10 we see a small example program that makes use of heap-memory allocation. This program follows normal `rust` style recommendations and only defines functions which returns the buffer by value[29].

To make the buffer returned by `Buf::new()` heap allocated we use the `box` keyword on line 17. Semantically this line says that we should (1) create a `Buf` object using the `Buf::new` function, (2) allocate heap space to store the newly created `Buf` and (3) move the `Buf` into the heap allocated space. The compiler will make this operation more efficient by getting rid of the local copy and simply having `Buf::new` write directly into heap memory, keeping the same semantics with slightly different mechanics.

When compiling this code the `rust` compiler uses the `exchange_malloc` function, shown in Figure 11, to allocate the memory needed for the buffer. The `exchange_malloc` function is a special type of function called a `lang_item`. The `lang_item`s are functions or types that the `rust` compiler knows additional information and invariants about the function or type beyond what it normally would. In this case `rustc` knows that `exchange_malloc` can never return a null or invalid pointer for a non-zero length allocation[30]. This is guaranteed in the function itself by the check on line 14, calling `abort` if the allocation fails.

The compiler uses this information to optimize the compiled code it creates by not checking that the allocated pointer is null. It further uses the fact that the constructor is rewritten to use an out argument to allow it to make line 17 of Figure 10 run without a copy. These combine to cause the compiler to create code similar to Figure 12 from the code in Figure 10.

By aborting when unable to allocate memory and never returning `null`, `rust` entirely hides the fact that allocations may fail from the programmer. As far as the `rust` programmer (and compiler) are aware, allocation is infallible and perfect.

3.3.2 How Failure Works & Why it Matters

One might wonder why `rust` does not expose the fact that heap allocations may fail. There is no official reason present in the documentation for why this is kept

[29] The `#[inline(never)]` in the example simply prevents the `rust` compiler from inlining the call to the constructor, which would make the compiled code in Figure 12 more difficult to read.

[30] It will return the invalid `0x1` pointer for all zero length allocations.

```rust
// Copyright 2014 The Rust Project Developers.
// Licensed under the Apache License, Version 2.0
// Taken from liballoc/heap.rs

/// The allocator for unique pointers.
#[cfg(not(test))]
#[lang="exchange_malloc"]
#[inline]
unsafe fn exchange_malloc(size: usize, align: usize) -> *mut u8 {
    if size == 0 {
        EMPTY as *mut u8
    } else {
        let ptr = allocate(size, align);
        if ptr.is_null() { ::oom() }
        ptr
    }
}

// Taken from liballoc/lib.rs

/// Common out-of-memory routine
#[cold]
#[inline(never)]
pub fn oom() -> ! {
    // FIXME(#14674): This really needs to do something other than just abort
    //                here, but any printing done must be *guaranteed* to not
    //                allocate.
    unsafe { core::intrinsics::abort() }
}
```

Figure 11: `Rust`'s `liballoc` code defining allocation.

hidden but various `rust` developers have discussed it somewhat. The general idea is that having `box` expose the fact that allocation can fail is to "unwieldy" for how uncommon it is[10]. Furthermore when allocation does fail `rust` will use the call frame information (CFI) based exception-handling system created for `C++` to unwind the stack of the current task and ensure all destructors are run.

This is, in some ways, a quite understandable solution for most use cases. When writing user code this is generally not a problem as demand paging[31], swapping, and the large amount of physical memory in most systems mean that actual allocation failure is quite rare. In fact, a process is more likely to just be killed when there is very little memory than get a failed `malloc(3)`, `mmap(2)` or `brk(2)` call (at least on linux). Furthermore CFI exception handling has many good implementations for user-space thanks to the popularity of `C++`, meaning that in the (incredibly rare) case that allocation does fail the other threads in a `rust` process will still have a consistent state.

Unfortunately, while this is an acceptable strategy for user processes it is absolutely terrible for kernel code. Kernel code is generally unable to make use of demand paging. This is for several reasons. Firstly, unlike a user-space process almost all of the kernel's memory is actually being used, mostly to cache pages of data from disk and hold the actual memory from all of the system processes. This makes sense since any memory that is earmarked for the kernel but not being used is actually being wasted, unlike with user processes where such memory is most likely not even allocated thanks to demand paging. In fact, many kernels will deliberately fill any empty memory with data from the hard-disks if there is no other work going on. Secondly, large parts of the kernel are either accessed so often they cannot be paged out to secondary storage (for example, the syscall code) or are so critical they cannot be paged out without making it impossible to page them back in (for example, the

[31] Demand paging is the practice of saying that memory allocations have succeeded without actually having allocated or loaded the memory requested yet. When the memory is actually used by the process the kernel will allocate or load it. This is often combined with memory-overcommit, a practice where processes are allocated more memory then is actually available. These two techniques are some of the most important applications of virtual memory and are used extensively in the implementations of `mmap` and `fork`.

disk driver or scheduler).

The end result of this is that allocation failure is a reasonably common occurrence in operating system kernels. Even worse it is generally not even a big deal in kernel code. Generally when one has failed to allocate something, the solution will be to either return an `errno` indicating this failure (`ENOMEM`) or to simply tell the system to remove some of the cached, but unused, pages of memory and try again. Indeed one of the final, and hardest, tests a student's implementation of `weenix` must pass is being able to continue running even when a process is eating all available memory. When this test is being run it is common that almost all allocations in other parts of `weenix` will fail.

This is not even getting into the major practical problems with this error handling strategy in kernel development. The foremost among these is the fact that using CFI stack unwinding is much more difficult in a kernel context than it is in a user-land one. Many users of `C++` in a kernel context ban them altogether, such as Apple's I/O kit driver system[2]. Furthermore, even if CFI unwinding is implemented, there is often a lot more manipulation of system-wide data needed in making an operating system than there is in normal user-land processes, and creating destructors that will correctly undo these changes is difficult.

3.3.3 Solutions and Workaround

Thankfully, unlike many of the other problems with `rust` I have encountered, this one has a solution that is fairly simple in theory. All that would need to be done is to change it so that heap memory allocation might fail, and all heap memory must be explicitly checked before being used. In fact this could theoretically be done by patching the standard-library code (although one would not be able to use the `box` syntax). Unfortunately, there are some obstacles to doing this.

Firstly, as mentioned above, the allocation function in rust is a special `lang_item` function. This is why the `exchange_malloc` function is able to be called before the call to its contents constructor, as can be seen in Figure 12. Switching to using some other function would mean this optimization would not be performed without modifying the `rust` compiler. This would massively slow down `rust` code by forcing all heap allocated objects to be copied over from the stack. Depending on the size of the objects it could also easily cause stack overflows as the objects are temporarily stored on the stack. With the recent approval of RFC 809[32] this might not be the case in the near future, however as of April 2015 patching the `rust` compiler would be necessary.

Secondly, large parts of the `rust` standard library depend on the pseudo-infallible nature of allocation present at this time. At an absolute minimum every single one of the collections data structures and smart pointers will need to be rewritten to support a fallible allocation model, as well as all data-structures that directly depend on them. Furthermore many common interfaces will need to be changed since it can no longer be assumed that, for example, inserting a value into a list will succeed. In the end making this change would probably require rewriting almost the entire standard library, adding functions that expose the fallible nature of memory.

In the end, I had to create a workaround for this problem. This involved creating a simple allocator in `rust` that is given a small percentage of memory immediately upon system boot. When a function was unable to allocate memory through the normal methods it would make use of this backup allocator. Whenever the kernel did an operation that might allocate memory it was wrapped in a macro that would check to see if this backup allocator is in use. If it was, the macro would assume that the allocation had failed and would destroy the result of the operation, hopefully freeing the memory being used in the backup allocator. This is rather fragile for obvious reasons and cannot be expected to scale very well at all. Indeed, it is not even clear if this solution would have been workable for this project, since memory allocation failure would likely not become a real issue until well into the `VM` project, implementing virtual memory.

Unfortunately, this one issue largely prevents the use of `rust` in the creation of an operating system kernel. One must be able to safely and cleanly recover from memory allocation failure without aborting. Until it is possible to do this without sacrificing a huge amount of speed `rust` will likely not be a real possibility in operating system development.

3.4 Performance

One last comparison that must be made when evaluating a new language is its performance. Taking these types of measurements are rather difficult and are prone to random variance based on the host com-

[32] `https://github.com/rust-lang/rfcs/blob/master/text/0809-box-and-in-for-stdlib.md` (April 2015)

puter. Furthermore, it is difficult to come up with comparisons that are valid between the two implementations. Still I did set up identical test cases in both the current `reenix` and my implementation of `weenix` as it stood after finishing `DRIVERS`. Lacking any normal performance testing or call tracing tools, I restricted my comparison to the process control subsystem of `reenix` and only performed a few tests.

I ended up running two performance tests on the two operating systems. In the first I simulated a system where multiple threads are all accessing a contested resource, in this case a shared integer. All threads will co-operatively add together this integer, switching out occasionally. The code for this and the times are found in Figure 13. For this test the `rust` code took, on average, 3.040 times as long as the `C` code to run. This number was fairly consistent across all workloads, so it seems likely these costs scale with the amount of work that `rust` is doing.

I also tested the time it takes to perform a common system call, `waitpid`, on both systems. These times can be found in Figure 14. For this function we found that `rust` was, on average, 2.036 times slower than `C`.

This seems to indicate that the amount of slowdown in different functions will vary and is related to something other than the language's compiler. The most likely candidate for what causes this slowdown is probably allocation, since `rust` seems to do it much more often than the `C` code would, and in `rust` every allocation has a higher cost, since the right allocator must be found from a list. Other contributing factors could include the stack-checking code in `rust`, the greater amount of code to perform all the required error checks, or a larger amount of `vtable` lookups generated by the use of `rust traits`.

4 Conclusions

When I started this project I did so with the goal of creating a `unix`-like operating system on the level of `weenix` in `rust`. This goal proved to be much too ambitious given the amount of time I had to complete the project. I was able, however, to complete the first two parts of the `weenix` project, proving that a complete `reenix` is likely possible. Furthermore, I was able to draw some important conclusions about the use of `rust` in a kernel development context from this project. This is particularly important given that, as far as I know, nobody else has ever really attempted, or made as much progress on, an operating systems project of this scope in `rust` before now.

I was, I believe, able to conclusively show that the basic design of a `unix`-like operating system kernel is feasible to build using `rust`. Furthermore, the experience showed me that, although there are still some rough edges, the implementation work that would be done by students doing the `weenix` project would generally be simpler to accomplish in `rust` than in `C`. However, the implementation of the support code was sometimes more difficult in `rust` and resulted in code that was less clear. Still, overall I would say that `rust` would be an improvement.

Through doing this project I was able to get a good sense of some of the ways that `rust` could improve with regards to operating system kernel programming. Most of these desired improvements fix issues that are at the level of annoyances and are either easy to add to `rust`, being added to `rust` at some point in the future, or are simple enough to work-around. I only found one issue, memory allocation failure, during the project that is truely critical to address in order to make `rust` a serious choice in kernel programming. Unfortunately this issue is, at once, almost impossible to really work around, very complicated to fix, and not considered an important issue by the developers of `rust`.

The fact that `rust` ended up being substantially slower than `C` is rather discouraging. On the other hand, it is likely that much of this slowdown has to do with the overhead of allocating memory in `reenix`. To fully determine the scope of this issue one must do much more in depth research than I was able to do here. Furthermore, the `C` version can, and sometimes does, omit checks that the `rust` compiler will always include. Together all of these add up to the conclusion that, while worrying, the decrease in speed of `rust` is not something that is insurmountable. Still, this will need to be considered if one wants to con-

`Rust` version of the `Mutex` test function.

```rust
extern "C" fn time_mutex_thread(high: i32, mtx: *mut c_void) -> *mut c_void {
    let mtx : &Mutex<i32> = unsafe { transmute(mtx) };
    let mut breakit = false;
    loop {
        if breakit { kthread::kyield(); breakit = false; }
        let mut val = mtx.lock().unwrap();
        if *val == high { return 0 as *mut c_void; }
        *val += 1;
        if *val % 4 == 0 { kthread::kyield(); }
        else if *val % 3 == 0 { breakit = true; }
    }
}
```

`C` version of the `Mutex` test function.

```c
typedef struct { int cnt; kmutex_t mutex; } data_mutex;

void *time_mutex_thread(int high, void* dmtx) {
    data_mutex* d = (data_mutex*)dmtx;
    int breakit = 0;
    while (1) {
        if (breakit) { yield(); breakit = 0; }
        kmutex_lock(&d->mutex);
        if (d->cnt == high) { break; }
        d->cnt++;
        if (d->cnt % 4 == 0) { yield(); }
        else if (d->cnt % 3 == 0) { breakit = 1; }
        kmutex_unlock(&d->mutex);
    }
    kmutex_unlock(&d->mutex);
    return 0;
}
```

High	# of Threads	`Rust` Time (seconds)	`C` Time (seconds)	Slowdown
100	2	0.0238	0.0079	2.999
100	10	0.0851	0.0294	2.889
100	100	0.7865	0.2782	2.827
10000	2	1.9372	0.5920	3.272
10000	10	6.7841	2.1688	3.128
10000	100	61.611	19.697	3.128

Figure 13: `Rust` & `C` `Mutex` timing code and average of 10 runs

Child process State	`C` time	`Rust` time
Alive	0.51325 ms	1.1435 ms
Dead	0.37493 ms	0.6917 ms

Figure 14: Time in milliseconds to run `waitpid` function on average over 1000 runs.

sider using `rust` for an operating systems project.

Overall I would say that `reenix` was fairly successful. By getting as far as I did I was able to prove that the idea behind the project is fundamentally sound. I was also able to show that, overall, the `rust` language does help with creating this type of highly complicated system, and, with a small amount of modification, could effectively replace `C` in this field.

References

[1] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer 1994 Technical Conference*, jun 1994. https://www.usenix.org/legacy/publications/library/proceedings/bos94/full_papers/bonwick.a.

[2] Apple Computers. *IOKit Fundamentals*. https://developer.apple.com/library/mac/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/IOKitFundamentals.pdf.

[3] Jonathan Corbet. The slub allocator. https://lwn.net/Articles/229984/, apr 2007.

[4] Servo Developers. https://github.com/servo/servo/wiki/Roadmap, mar 2015.

[5] The Rust Developers. mar 2015. https://doc.rust-lang.org/book/.

[6] Thomas W. Doeppner. *Operating Systems In Depth: Design and Programming*. Wiley, 2010.

[7] Felix S Klock II. Allocator rfc, take ii. https://github.com/rust-lang/rfcs/pull/244, sep 2014.

[8] S.R. Kleiman. Vnodes: An architecture for multiple file system types in sun unix. 1986. http://www.solarisinternals.com/si/reading/vnode.pdf.

[9] Chris Lattner. *The Architecture of Open Source Applications*, volume 1, chapter 11: LLVM. lulu.com, 2012. http://www.aosabook.org/en/llvm.html.

[10] Chris Morgan. What's rust's mechanism for recovering from (say out-of-memory) error? http://www.reddit.com/r/rust/comments/279k7i/whats_rusts_mechanism_for_recovering_from_say/chyo6ug.

[11] Dag-Erling Smorgrav. feb 2014. https://www.freebsd.org/cgi/man.cgi?query=uma.

[12] Operating Systems Teaching Assistant Staff. *Weenix*. Brown University Department of Computer Science, mar 2015. http://cs.brown.edu/courses/cs167/docs/weenix.pdf.

[13] Nathan Zadoks. There is no syntax for struct alignment. https://github.com/rust-lang/rfcs/issues/325, sep 2014.