RKOS: UNIKERNEL DESIGN FOR SAFETY AND PERFORMANCE

by

Peter H. Marheine

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in
Computing

School of Computing

The University of Utah

June 2016

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of Peter H. Marheine
has been approved by the following supervisory committee members:

THIS PAGE IS A PLACE HOLDER ONLY

Please use the updated form on the Thesis Office website

John Regehr , Chair enter date

Date Approved

Eric Eide , Member

Date Approved

Kobus Van der Merwe , Member

Date Approved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Peter H. Marheine in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

John Regehr
Chair, Supervisory Committee

Approved for the Major Department

Ross T. Whitaker
Chair/Dean

Approved for the Graduate Council

David B. Kieda
Dean of The Graduate School

ABSTRACT

Software appliances are an increasingly common method to deploy services or groups of services for public consumption, and unikernels have appeared as an attractive way to build software appliances into self-contained packages from tightly-integrated collections of source code, avoiding undesirable overhead in memory footprint and performance. This work presents RKOS, a unikernel implemented in the Rust programming language which offers safety guarantees comparable to implementations which depend on complex runtime libraries while being capable of providing predictable application performance demanded by real-time applications in a relatively simple implementation. The design of RKOS and parts of its implementation include aspects not previously considered in the literature, and have inspired several extensions to the Rust language and its current implementation.

CONTENTS

GLOSSARY

ELF The executable and linkable format, a file format commonly used for programs and associated libraries in Unix systems.

Floating point numeric representations are commonly used in computer systems to represent non-integral and very large values at the cost of some precision. Typical range of a 32-bit floating point value is approximately $2^{\pm 128}$ with 24 bits of precision.

Heap memory refers to parts of main memory used by a program to contain storage for data allocated dynamically, usually for items which either consume a large amount of space or have size which cannot be known statically. Contrast with stack memory.

Interrupt An event in which a CPU suspends current execution and switches to a different context for later return to normal execution. Usually triggered by events external to the CPU.

IRQ Interrupt request; the signal provided to a CPU causing it to trigger an interrupt.

ISR An interrupt service routine; the part of a program which is executed in response to a CPU interrupt.

TLB Translation lookaside buffer, a cache of virtual-to-physical address mappings used in CPUs to improve performance of virtual memory systems.

Virtual memory (VM) systems split main memory into independent virtual and physical address spaces, where programs work exclusively within the virtual address space. Virtual addresses can be mapped to arbitrary physical addresses within minimum granularity of the virtual memory page size, or left unmapped to cause a fault if access to the virtual address is attempted.

CHAPTER 1

INTRODUCTION

With the recent growth of cloud computing, software appliances have become a common method to deploy self-contained services as virtual appliances on shared infrastructure, with isolation provided through virtualization. Unikernels in particular have been positioned as a way to improve the efficiency of services deployed as virtual appliances, avoiding the overhead of a conventional operating system in applications which do not require the full capabilities of a typical operating system.

1.1 Unikernels

Unikernels are a relatively new concept, but derive from earlier work on "library operating systems", with the earliest of the newer wave of such systems being MirageOS [1]. A unikernel (and library operating system) places isolation boundaries at the lowest possible level of the target hardware, effectively discarding traditional levels of isolation like the split between user- and kernel-space code (where only the latter is permitted full access to system hardware) or processes (which share no visible resources). An application gets compiled with the library portions of a unikernel (or library OS) into a single-purpose image, which runs on the target system. Figure 1.1 illustrates the construction of a unikernel system compared to a conventional operating system stack. For PC-based applications, the weakness of this approach is the requirement that the library code include drivers for the target hardware. For embedded applications running monolithic firmware[1] this is relatively simple to ensure because the target hardware is if not homogenous at least well-understood and unchanging for a given configuration, but on PC-class hardware like that used to run most Internet services there is wide variety in hardware and often the hardware configuration changes without a user changing the software running on it.

[1] Compare monolithic firmware to a firmware based on a general-purpose operating system like Android.

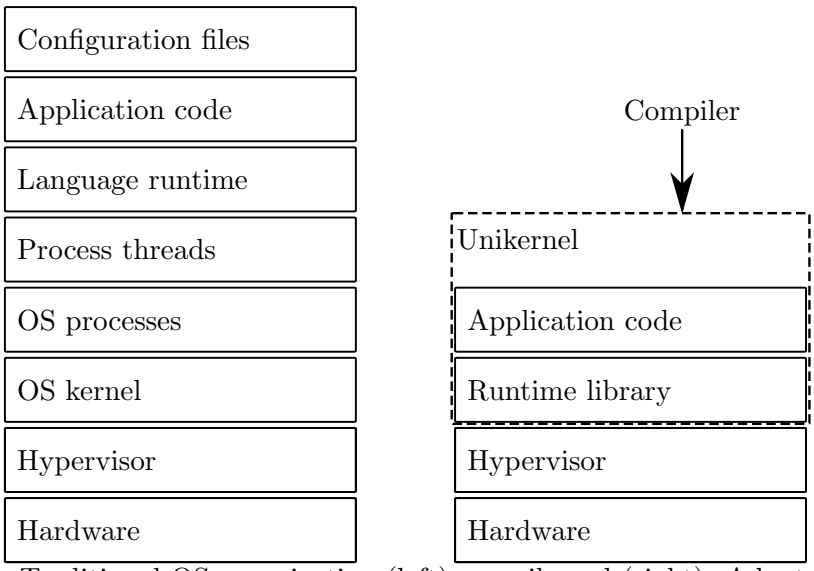


Figure 1.1: Traditional OS organization (left) vs unikernel (right). Adapted from [1]).

The motivating observation for the new wave of unikernels is that in virtualized environments the visible hardware (especially in paravirtualized environments) can be made to appear homogenous even if the underlying hardware has great variance, so the unikernel must support only a small set of hardware devices. The cost of virtualization compared to running on bare metal without a hypervisor is non-zero, but the isolation granularity between guests of a single hypervisor is lower than that between processes under a single operating system [1] (so can be assumed to be higher-performing), and the cost of using a hypervisor is effectively zero if a given deployment would be virtualizing applications in any case (such as in the environments provided by typical cloud service providers).

1.2 Existing Implementations

The implementations of these unikernel systems have taken two main approaches to date: providing a minimal runtime library to host software implemented in a managed-memory language, or adapting existing code from general-purpose operating systems. Systems taking the former approach include MirageOS [2] (OCaml), HaLVM [3] (Haskell) and Erlang on Xen [4] (Erlang), while those taking the latter approach include Rumprun [5] (derived from BSD source components) and OSv [6] (BSD components with custom C++ and supporting Linux binaries).

1.2.1 Safe Language Unikernels

Those systems depending on language runtime support for enforcing program safety tend to depend on garbage collection, which typically has a negative effect on performance, especially in predictability. Earlier work attempting to characterize the performance of unikernels under realistic workloads saw generally good performance from Mirage for example, but latency had generally higher variance and slower worst-case performance [7]. Other measurements of GC pauses with modern runtime environments show comparable pause durations of around 1 millisecond, with worst cases as high as 50 milliseconds [8].

While there exists previous work on implementation of garbage collection algorithms with predictable characteristics suitable for real-time applications [9, 10], these still sacrifice performance in exchange for freeing application code from the need to manage memory. Bacon et al.'s Metronome garbage collector [11, 12] exhibits predictable pause times of as little as one millisecond, but with actual memory use more than doubling the size of the application's working set in some benchmarks and minimum application CPU utilization of about 45%. This implies an application may require overprovisioning in both CPU time and memory capacity by approximately a factor of two in order to be able to meet real-

time guarantees using a garbage collector. For systems requiring sub-millisecond response times, not even Metronome is a reliable choice. Later work has demonstrated garbage collection implementations achieving 100% application CPU utilization in some applications through implementation of garbage collection in specialized hardware, but still sacrificing some absolute throughput in exchange for a higher level of abstraction [13].

The performance characteristics of unikernel garbage collectors have not been measured in the literature, but profiling of Mirage OS applications appears to show performance similar to that of the mainline OCaml garbage collector measured by Scherer [8], with typical GC pauses in the 1 to 10 millisecond range [14].

1.2.2 C-based Unikernels

The ability of a unikernel to correctly and securely perform its designed tasks is strongly dependent on the implementation being free of bugs, because it has by design discarded common levels of protection, especially regarding memory safety; any part of the system can write anywhere in memory, easily corrupting state in case of an error. The runtime-managed category of systems depend on their runtime implementations for correctness, which is difficult to characterize. In general these implementations are as immature as the unikernel concept itself, but they may be based on high-quality mature runtimes with small modification.

Among those implementations based on existing code bases capable of running existing programs, the code tends to be very mature, and as such is assumed to be largely bug-free. However, given that these are typically implemented in C, the very nature of the implementations suggests a high probability that potentially high-impact bugs continue to exist undiscovered. Tools like Csmith [15] and KLEE [16] illustrate the prevalence of bugs even in well-tested C and C++ programs (C compilers in the former case, usually implemented in C or C++ themselves), and the ever-present danger of undefined behavior in C programs may even be heightened within a unikernel where parts of system code which are not usually concurrently visible to the compiler become visible, potentially allowing the compiler to exploit new and exciting undefined behaviors which are not observed in the domains in which the application and system code are well-tested.

1.3 A Rust Unikernel

Rust is a new systems programming language designed to fit the same niches currently occupied by C and C++, but with improved compile-time safety guarantees [17]. The safe subset of Rust statically prevents most classes of memory errors and many concurrency-

related errors, while the 'unsafe' subset of the language enables the types of low-level operations which are necessary in systems code like creating and using arbitrary pointers to memory.

There are a number of other Rust-based operating systems projects extant, but no notable ones targeting the same use cases as RKOS. rustboot [18] and intermezzOS [19] are small OS kernels targeting simple experimentation with the language or teaching. Redox [20] is the most useful, designed as a conventional microkernel with the usual split between user- and kernel-space.

The use of Rust permits unique approaches to some implementation primitives with improved compile-time checking and encourages the use of safe code in applications without sacrificing performance. The language's expressiveness (especially when compared with C) also allows simpler code, better suited for experimentation and pluggable implementations. This work has also inspired several extensions to the Rust language and its current compiler which are useful to other projects with similar operating-system-like goals.

This work presents RKOS, a unikernel system written in Rust, intended to be suitable for real-time applications where unpredictable latency due to runtime overhead is unacceptable without sacrificing performance and providing stronger safety guarantees than those systems with similar performance properties but implemented in less strict languages (namely, C). The APIs available to a RKOS program should be familiar to programmers with some experience programming Rust in conventional contexts. Development of RKOS also spurred development of additional language and compiler features which may simplify systems programming tasks and permit more efficient code.

Chapter 2 describes in some detail the Rust language and how it enforces safety, and Chapter 3 describes the language and compiler changes implemented. Chapter 4 describes the design of RKOS, and Chapter 5 the current implementation of the system. Chapter 6 summarizes the work and discusses possible avenues for future work.

CHAPTER 2

THE RUST LANGUAGE

Rust is a relatively new programming language targeting systems tasks, especially those where C or C++ might otherwise be used. Its design focuses on safety, while imposing performance costs on programs only for those features that a program uses [21]. Of particular note, the "safe" subset of Rust statically prevents many common errors including use-after-free, buffer overflow and data races, all of which is done without an underlying runtime environment providing memory safety via garbage collection or reference counting. This is often characterized as "free of segfaults," which is not strictly accurate but characterizes the language's unique features well. "Unsafe" Rust code is permitted to perform arbitrary operations which are not provably safe, allowing implementation of items which provide safety through enforcing their own invariants which the compiler is not capable of tracking. There are no limits on where unsafe code may be used or guarantees regarding its safety however, so programmers writing unsafe code must be careful.

Influences on Rust's design notably include the ML family (OCaml and SML), C++ and Haskell [22], with the approach to memory safety coming from research-oriented languages, particularly Cyclone [23].

This chapter discusses some of the aspects of Rust's design that make it attractive for implementation of a unikernel system like RKOS and how it enforces safety in low-level code while minimizing the runtime costs of its features.

2.1 Traits

Generic programming in Rust is supported by traits, which are analogous to interfaces in languages like Java and C#, or typeclasses in Haskell. The trait definition defines a set of function or method signatures and associated types, with optional default definitions for the defined items. This approach can be unfamiliar to programmers steeped in a tradition of object inheritance, but offers similar expressiveness while offering some unique capabilities.

A sample trait definition and implementation is given in Listing 2.1, approximately matching the language definition of addition. The `Add` trait marks types for which addition with addends of types `Self` (the type for which the trait is implemented) and `RHS` yield a result of type `Output`.

```
trait Add<RHS> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}

impl Add<i32> for i32 {
    type Output = i32;
    fn add(self, rhs: i32) -> i32 {
        self + rhs
    }
}
```

Listing 2.1: A trait for types supporting addition

`Add` is a language-level trait which, among some others, is understood specially by the compiler rather than being defined entirely in libraries. These allow language operators to be used with custom types in addition to those provided by the compiler. The implementation of `Add` for `i32` in Listing 2.1 is one such implementation that is inherent in the language, written explicitly here only as an example: addition between signed 32-bit integers is defined at the language-level, and in fact the `self + rhs` expression would cause unbounded recursion with such an implementation.

2.1.1 Marker Traits

Traits may be defined without any members, useful for collecting multiple traits into a single supertrait or to indicate that a type has properties which cannot be expressed in terms of associated types or functions. A library might require that types used for a particular optimized implementation have specific memory layout, so may define a trait to be implemented by types with the required layout.

A trait can be marked `unsafe`, for which implementations must also be marked `unsafe`. This ability is of particular value to definitions of marker traits, because they are generally dependent on properties of the implementing types which are not enforceable exclusively through the language's type system.

The language defines two traits important to the correctness of concurrent programs:

- `Send` marks types which are safe to move between threads.

- `Sync` applies to types which can be safely shared between threads (implying `&T` is `Send` for any `T` that satisfies `Sync`).

These traits are implemented automatically by the compiler according to type-based heuristics. In general, all types implement both of these bounds except raw pointers and the `UnsafeCell` primitive which permits mutation through immutable references and thus can cause data races if shared between threads. In cases where a type is automatically assumed to be non-`Sync` (or `Send`) such as because it contains raw pointers, it can be opted in to (asserting that the type is safe but not possible for the compiler to prove so automatically) with `unsafe impl`. Similarly, a negative trait bound of `!Sync` or `!Send` can be used on a type which is automatically determined to be thread-safe but is in fact not.

2.2 Generic Programming

Traits provide Rust's basis for generic programming, in which both types and traits may be generic over arbitrary types. The definitions in Listing 2.2 illustrate a struct containing values of two arbitrary types `T` and `U`, and a trait generic over an arbitrary type `T`.

```
struct Pair<T, U>(T, U);

trait Combine<T> {
    fn combine(&self, t: &T) -> Self;
}
```

Listing 2.2: A simple generic struct and trait definition

Without the ability to constrain the properties of type parameters, it is difficult or impossible to provide more useful functionality. For instance, if the `Combine` trait were to be implemented for the `Pair` struct such that invoking the `combine` function performs pairwise addition, constraints like those in Listing 2.3 could be specified.

```
impl<T, U, V, W> Combine<Pair<V, W>> for Pair<T, U>
        where T: Add<V, Output=T> + Copy,
              U: Add<W, Output=U> + Copy,
              V: Copy, W: Copy {
    fn combine(&self, x: &Pair<V, W>) -> Pair<T, U> {
        Pair(self.0 + x.0, self.1 + x.1)
    }
}
```

Listing 2.3: A complex generic trait implementation

In effect, this trait implementation defines the operation of `Combine` between any two `Pair`s, provided addition is defined between the elements of each `Pair`. The `where` clauses mark the traits which each named type must satisfy, in this case that addition between

values of types `T` and `V` and `U` and `W` must be defined to yield values of types `T` and `U`, respectively. The types contained in each pair must also be copyable, because the inputs to `combine` are immutable references so the contained values cannot be moved or otherwise modified.

Generic items are monomorphized at compile-time, like templates in C++. This ensures that there is no runtime cost in checking of trait bounds because the types are statically known and thus the appropriate code can be used directly, but this approach can increase the size of compiled code if many different types are used with the same generic item. Recursive generics are also forbidden, because in general it is not possible to determine whether a chain of recursive generic invocations (adding layers of type parameters with each invocation) will terminate. A similar pattern in a language without strict monomorphization could be allowed, because types (and implementing code for those types) may be constructed at runtime.

2.2.1 Dynamic Dispatch

It is possible to perform trait-based dynamic dispatch, but doing so requires explicit action on the programmer's part. When a trait name is used in a position where a type is expected, this is interpreted as a *trait object*, encapsulating a vtable for the trait (set of function pointers for the trait implementation) and actual instance of the implementing type. The size in memory of a trait object depends on the size of the implementing type, so trait objects must usually be placed behind pointers, like `&Trait`, a reference to a value with known type elsewhere, or `Box<Trait>`, an owned trait object stored on the heap where it need not have a statically-known size.

2.3 Enumerations

Rust's enumerated types are an important building block for many library primitives, and can be used either as tagged unions or simple names for constants. One of the most common examples is `Option`, which has two variants; `Some` and `None`:

```
enum Option<T> {
    None,
    Some(T)
}
```

The `Some` variant embeds an instance of an arbitrary type, while `None` does not. Logically, these can represent the presence or absence of a value which is of particular utility when optionally using a value by reference. A reference must refer to a valid value so the

C-like solution of using `NULL` as a special value is not legal, but `None` is explicit about the absence of a value. Additionally, the compiler applies an optimization to non-nullable types (references included) such that `None` in such a type is represented as a null value so additional storage need not be allocated to discriminate between `Option` variants.

Enumerated types are often destructured through pattern matching to branch according to the current variant. It is forbidden to use a contained value without first ensuring the correct variant is in use, a major pillar of memory safety.

2.3.1 Error Handling

The `Result` enumeration, shown in Listing 2.4, is another commonly-used type, in this case for indicating the presence or absence of an error, allowing the consumer of a `Result` value to get the result of an operation on success or the reason for failure on error. In order to get the result on success, this type enforces that a user at least acknowledge the possibility of error, like the example in Listing 2.5.

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

Listing 2.4: The `Result` type

```
let file_handle = match File::open("data.bin") {
    Ok(f) => f,
    Err(ref e) => {
        match e.kind() {
            ErrorKind::NotFound => {
                /* File does not exist */
            },
            _ => panic!("Unexpected error: {}", e)
        }
    }
}
```

Listing 2.5: Representative consumption of a `Result` when performing I/O.

In this example the program takes some unspecified action if the file to open is not found, and carries on normally if the operation is successful. In case of other errors, the program panics: the current thread terminates after reporting the given error message, and unless specifically handled will terminate the entire program. The `panic` macro is used for handling of unexpected errors from which the program cannot recover (including errors like out-of-bounds array accesses), while `Result` requires handling for errors which may occur during normal operation (but a program may choose to panic in response).

2.4 Ownership Semantics

The heart of the Rust compiler's ability to prevent many types of memory unsafety is in the language's ownership semantics. In general, a given value has exactly one owner at any given time, and ownership changes through assignment. Ownership moves by default, in which a value assigned from one variable binding to another becomes usable only with the last name. Use of any binding with values moved out is an error. In Listing 2.6, a vector value is bound to the name `v`, then to `v2`. Following the second assignment, use of `v` is an error.

```
let v = vec![1, 2, 3];
let v2 = v;
```

Listing 2.6: Moving a value between bindings

The `Copy` trait marks types which can safely be copied with no additional logic, often referred to as "plain old data" (POD). Most primitive types are copyable, including integers and pointers, but user-defined types are not `Copy` unless specifically defined to be. `Vec` as used in Listing 2.6 is not a copyable type, but if a different (copyable) type were used, like `i32`, `v` and —v2— would be usable concurrently because the initial value would be copied rather than moved.

Mutability is a property of bindings rather than values, and bindings are immutable by default. An immutable binding may not be assigned to, while a mutable one may; mutation of values regardless of their depth may only happen through a mutable binding, so a struct field for example may only be modified if a binding to the struct as a whole is mutable. This "immutability by default" approach can help prevent some bugs, but is not critical in enforcing safety.

2.4.1 References

Rust references act like C++ references: they are guaranteed to refer to a valid value and are typically represented at the machine level as a simple pointer. Taking a reference to a value is often referred to as "borrowing" the value, in that ownership does not move but access is restricted as long as the reference is live. Similar to variable bindings, references are immutable by default but mutable references may only be taken from mutable bindings. The example in Listing 2.7 creates a mutable reference to an integer and increments it via the reference.

```
let mut x = 0i32;
let p = &mut x;
*p += 1;
```

Listing 2.7: Modifying a value through a mutable reference

Taking a mutable reference to a value precludes any further use of that value as long as the reference is live, including creating other references; mutable references never alias. Immutable references are permitted to alias, because it is guaranteed that during the lifetime of any given immutable reference there can be no mutation of the referent.

As a practical example of how these rules prevent unsafety, we consider the problem of iterator invalidation in C++ where an iterator over a value may be invalidated by mutation of that same value. Listing 2.8 illustrates this pattern in equivalent Rust code.

```
let mut v: Vec<i32> = vec![1, 2, 3];
for i in v.iter() {
    if *i < 6 {
        v.push(*i * 2);
    }
}
```

Listing 2.8: Iterator invalidation in Rust

Here a list of integers is iterated over, binding references (`&i32`) to `i`. When the value from the iterator is less than six, it is doubled and appended to the list. However, the Rust compiler rejects this code because appending to the list may invalidate the iterator:

```
error: cannot borrow ‘v‘ as mutable because it is also borrowed
       as immutable [E0502]
    v.push(*i * 2);
    ^
note: previous borrow of ‘v‘ occurs here; the immutable borrow
      prevents subsequent moves or mutable borrows of ‘v‘ until
      the borrow ends
    for i in v.iter() {
    ^
```

Calling the `iter` method of `v` creates an immutable reference to the list, and using the `push` method would require taking a mutable reference to the same value. Because these references may not be live at the same time, this code is rejected by the compiler, preventing memory-unsafety in the case where appending to the list requires allocating memory and invalidating references contained by the iterator.

Some programs require a way to express shared ownership of a value, which remains possible in Rust but not in purely safe code. The standard library type `Rc<T>` is one such implementation, providing reference counting for the contained value. This is often com-

bined with `Cell<T>`, which permits mutation of the contained value through an immutable reference, maintaining language mutability requirements through runtime checking.

2.4.2 Lifetime Annotation

The compiler does not have perfect visibility into implementing code; the language depends on lifetime annotations for references passing through API boundaries to determine what operations are safe rather than having oracular checking abilities, with design largely derived from earlier work on region annotations in Cyclone [23].

An example signature of a binary search function in C is given in Listing 2.9. It receives a pointer to a contiguous block of memory, a target element and comparator function pointer, returning a pointer of unspecified type. With only this type signature, it is impossible to know which of the input parameters the output is in reference to.

```
void *bsearch(
    const void *key, const void *base,
    size_t nel, size_t width,
    int(*compar)(const void *, const void *)
);
```

Listing 2.9: A binary search function in C

The function signature in Listing 2.10 is an idiomatic Rust interpretation of the same function. This version also receives a pointer to a contiguous block of memory, implicitly including the number of elements in the referenced region. This function is generic over all types which have a defined total ordering, indicated by the `Ord` trait, and returns a reference to a value of type `T` or `None`.

```
fn bsearch<'a, T: Ord>(
    key: &T, items: &'a [T]
) -> Option<&'a T>;
```

Listing 2.10: A binary search function in Rust

In addition to the type parameter `T`, this function also receives a lifetime parameter `'a` which is applied to the reference to `items` and the output reference, indicating the output reference refers to a member of the input array, and not the input key.

Revisiting the example of Listing 2.8, the signature of `iter` is `fn iter<'a>(&'a self)-> Iter<'a>`, where `'a` is the name of the `Vec`'s lifetime which the returned iterator must not outlive. This signature in effect includes information to the compiler that the iterator includes a reference to the list itself without explicitly stating the internal structure.

2.4.3 Static Lifetimes

A common error committed by beginning C programmers is returning a pointer to a value with automatic storage duration, illustrated in Listing 2.11. Modern C compilers are capable of emitting warnings in response to such an incorrect construct, but warnings are often ignored by the programmer, especially so when the programmer is inexperienced. Any use of the resulting pointer is undefined behavior [24] so users might expect an error from the compiler, but merely creating such an invalid pointer is not in itself an error. Even when built with a compiler which is incapable of leveraging the undefined behavior in this function, the referent can trivially be modified erroneously by changes to the stack (the typical location for values with automatic storage duration) like calling another function.

```
int *returnPointer() {
    int x = 0;
    return &x;
}
```

Listing 2.11: A C function allowing for undefined behavior with an invalid pointer

This function is an error in Rust unless the output reference is given a valid lifetime. The function in Listing 2.12 is rejected by the Rust compiler because there is no inferable lifetime for the output reference which makes it valid. The compiler's messages (below) indicate that the programmer must specify a named lifetime for the reference because there is no lifetime to implicitly assign to it.

```
fn returnPointer() -> &i32 {
    let x: i32 = 0;
    &x
}
```

Listing 2.12: An illegal Rust function with invalid reference lifetime

```
error: missing lifetime specifier [E0106]
fn returnPointer() -> &i32 {
                      ^~~~

help: run `rustc --explain E0106` to see a detailed explanation
help: this function's return type contains a borrowed value, but
      there is no value for it to be borrowed from
help: consider giving it a 'static lifetime
```

Naively taking the compiler's advice to annotate the function in Listing 2.12 with a `'static` lifetime (Listing 2.13) results in a different error:

```
fn returnPointer() -> &'static i32 {
    let x: i32 = 0;
    &x
}
```

Listing 2.13: The function from Listing 2.12 with incompatible lifetimes

```
error: ‘x‘ does not live long enough
    &x
     ^
note: reference must be valid for the static lifetime...
note: ...but borrowed value is only valid for the block suffix
     following statement 0 at 2:19
    let x: i32 = 0;
                 ^
```

Here the declaration of the function is legal because the returned reference is assigned a valid lifetime, but the implementation does not match the lifetime specified by the function signature. The compiler automatically determines the lifetime of values in the program and in this case the anonymous lifetime assigned to the value `x` corresponding to the time during which the function is executing is shorter than the entire time during which the program is executing.

This illustrates that lifetimes are descriptive rather than prescriptive. The compiler will never automatically satisfy lifetime requirements: the programmer must design code such that the requirements are satisfied, leaving the programmer in full control over any necessary tradeoffs. A legal implementation of the function (but with limited utility, expanded upon in Section 2.5) is shown in Listing 2.14.

```
fn returnPointer() -> &'static i32 {
    static X: i32 = 0;
    &X
}
```

Listing 2.14: A legal implementation of the signature from Listing 2.13

The declaration of `X` as `static` (allocated in the program's data section rather than on the stack) makes annotating the reference to it with the static lifetime legal, and implies that all calls to `returnPointer` will receive a pointer to the same memory.

2.4.4 Exclusivity of Mutable Borrows

As previously discussed, any access to a value when there exists a mutable reference to it is invalid. What may not be evident however is why doing so may be unsafe.

```
enum T {
    A(bool),
    B(i32)
}

let mut value: T = T::A(true);
if let mut T::A(ref internal) = value {
    value = T::B(-1);
}
```

Listing 2.15: Creating a reference into an enum-type value

In Listing 2.15, potentially unsafe behavior is prevented by the compiler:

```
error: cannot assign to ‘value‘ because it is borrowed [E0506]
    value = T::B(-1);
    ^~~~~~~~~~~~~~~~

note: borrow of ‘value‘ occurs here
    if let T::A(ref internal) = value {
                ^~~~~~~~~~~~
```

The assignment in this example would invalidate the reference `internal` because while the memory it refers to remains valid, its interpretation has changed; it contains an integer rather than `bool`. Attempted use of the same memory after its meaning has changed would be an error, especially so in that `bool` permits only 0 or 1 as values while an integer in the same location may contain any bit pattern.

The rationale here can be interpreted as an anonymous lifetime being assigned to the reference, referring to the lifetime of the borrowed value. Mutating `value` ends that lifetime In this case mutating `value` does not require taking a mutable reference to it, but writing to it ends the lifetime associated with `internal` which is an error because a reference must not outlive its referent.

2.5 Shared Mutability

The function in Listing 2.14 is legal, but of limited utility because a plain reference does not permit mutation of its referent and thus a user of the returned reference cannot perform any useful operations other than reading the constant value of its referent. Additionally,

mutable values with static storage duration are always unsafe to access and may not be used within safe code because they are invariably accessible to all threads within the program by dint of having a fixed, known location in memory. Thus, modifying the function to return a mutable reference would be incorrect.

Safe use of a value with static storage duration requires use of a type that satisfies the `Sync` marker trait. For the function in question, the value might be used as a shared counter, for which the primitive type `AtomicIsize` would be an appropriate choice. Types like the atomic primitives and mutexes allow mutation through immutable references because they synchronize access, preventing data races and include any necessary memory barriers to ensure ordering constraints are followed.

The function in Listing 2.16 applies this to the function from Listing 2.14, making it a useful piece of global state that would be usable as an event counter, for example.

```
fn getGlobalCounter() -> &'static AtomicIsize {
    static X: AtomicIsize = AtomicIsize::new(0);
    &X
}
```

Listing 2.16: A useful version of the function in Listing 2.14

2.6 Undefined Behavior

In the interest of portability and efficiency, many systems languages leave certain operations undefined in their formal specifications. In some cases the exact result of an operation is left to the language implementation, allowing the implementer to choose a behavior most sensible for the target platform. For example, the result of applying the `>>` to a signed type is specified by the C99 specification to be implementation defined [24]. In practice, compilers usually implement either a logical or arithmetic bit shift but are according to the letter of the specification allowed to yield any value from such an operation.

While implementation-defined behavior can harm a program's portability, it is rare enough that it rarely causes problems for programmers. By comparison, *undefined* behavior is much more dangerous, in that a compiler is permitted to make arbitrary assumptions about code which performs undefined behavior. Use of undefined behavior can result in surprising bugs, often when an optimizing compiler removes code which inadvertently performs undefined operations [25]. While tools like Clang's UBSan [26] allow programmers to instrument their applications and detect undefined behavior at runtime, these require awareness of the problem in order for the programmer to begin to use the tool, and can only detect errors which actually happen rather than those with mere potential to occur.

Consequently, effective use of such tools requires either good test cases or willingness to accept a (usually small) loss in performance for improved hardening [27].

Rust attempts to eliminate many classes of undefined behavior, but not all. Safe code can be assumed to be free of undefined behavior given any unsafe code it relies on is free of bugs. This correctness is enforced in most cases within the language's type system, but in some cases such errors are checked for at runtime. When writing unsafe code, the Rust programmer must still be aware of what operations are illegal so they can be avoided, but this is a somewhat easier task than doing so in C.

2.6.1 Integer Overflow

In C, the effect of integer overflow on reaching the limit of a type's representable range depends on its signedness. Unsigned overflow is specified to wrap (so `UINT_MAX` + 1 becomes 0), but overflow of signed values is undefined [24]. Both types of overflow can introduce surprising bugs in programs, and overflow has been noted as a common source of vulnerabilities, especially in operating systems code [28, 29].

Rust treats all implicit integer overflows as errors, but not as undefined behavior [22]. The compiler currently assigns wrapping semantics to all operations for "release" builds, but causes a `panic` on overflow in "debug" mode.[1] For those cases where overflow is intended, wrapping primitives are provided for which overflow and underflow are not errors.

2.6.2 Out-of-Bounds Access

Indexing operations on arrays and array-like types are checked in Rust, preventing errors due to out-of-bounds memory access. Whereas C arrays are usually referred to as bare pointers with separate (sometimes implicit) information regarding the actual size, Rust arrays always include size information for runtime checking. Both languages specify that referring to out-of-bounds memory is undefined behavior.

Sized arrays carry length information in their type, like `[u8; 8]` indicating an array of eight unsigned 8-bit integers. Unsized arrays (`[T]` for a type `T`) are represented internally as a tuple of *pointer-to-storage* and *number-of-elements*. Library types like `Vec<T>` include bounds checks in their implementations of the `Index` family of operators.

Runtime checking of indexes has a performance cost, but failure to perform correct bounds checking is a common source of bugs and especially security problems due to buffer overflows [29]. Rust attempts to mitigate the performance concerns associated with bounds

[1] There is a performance cost to checking for overflow, so release builds targeting maximum performance do not currently do it.

checking by encouraging the use of iterators (implemented with unsafe code) which bypass the usual indexing (and thus bounds checks) but still offer a memory-safe API.

2.6.3 Strict Aliasing

Low-level software like operating systems often performs type-punning on data in memory, reinterpreting data behind a pointer as different types according to the code's needs. This often runs afoul of C's strict aliasing rules, which state that the behavior of dereferencing a pointer which aliases (points to the same address as) another pointer of an incompatible type is undefined. Compilers taking advantage of this undefined behavior has been an issue for OS programmers for more than a decade [30].

Rust's pointer aliasing rules are weaker than those in C, and are inherited from those applied by the compiler's LLVM backend [31]. In general, a raw pointer must be used only to access memory associated with the value from which it is derived. Pointers derived from other pointers (such as by casting a `i32*` to `i8*`) have the same basis as the original pointer: that is, the compiler treats the pointers as aliasable and use of the latter is not undefined behavior.

Rust references are defined to never alias mutable and immutable bindings, and this is enforced by borrow checking in safe code (see Section 2.4). Unsafe code must ensure that references generated from raw pointers do not break the borrow-checking rules.

2.6.4 Data Races

Rust and C use the same semantics for data races, in which threads fail to correctly synchronize accesses to shared state: a data race is always undefined behavior. This is a relatively rare concern even among C programmers, because a race condition is usually nondeterministic and potentially buggy even before a compiler might take advantage of a data race for optimization purposes.

Rust prevents race conditions in safe code by requiring the `Sync` marker trait in locations where state-sharing is made possible, and these requirements are often also relevant to unsafe code.

2.6.5 Additional Dangers

The goal of a unikernel system is to provide as much of the system as is possible at compile-time, leveraging trust in the code to eliminate runtime protections and gain efficiency. No matter the implementing language, this requires trust that the implementation is free of bugs. In a C implementation, undefined behavior might lurk unknown, until

ever-more-aggressive optimizers become capable of finding performance gains in previously unexploited parts of a system.

In directly exposing functionality to a compiler that is disjoint in a traditional operating system (userspace APIs and kernel implementations), an unusually high amount of logic becomes exposed to the optimizer. Consequently, the chance of new exploitation of undefined behavior is increased. This is a particular danger to unikernel systems built on existing (mature) operating systems code, since the trusted code is being used well outside the domain in which it has been widely tested.

CHAPTER 3

COMPILER INTERRUPT SUPPORT

While useful for writing operating systems code, the Rust language is yet relatively immature and also welcoming of extensions to improve the language. While developing RKOS, I determined the available language tools for managing system state especially around interrupts were lacking and proposed and implemented solutions to simplify Rust code which must handle interrupts. While not strictly necessary from a functional perspective, these features allow simplification of code and offer some efficiency improvements in compiled code.

3.1 Motivation

A responsibility unique to operating systems programming that does not appear in other (higher-level) software is in servicing hardware interrupts. While it is possible to write operating system software without handling interrupts, doing so limits its ability to handle unexpected errors and respond to external events in a timely fashion. As a practical example, Table 3.1 lists some of the architecturally-defined interrupts in x86.

Of those listed, the majority should not occur in a bug-free program, but some are used to support features of the operating system. The page fault vector is very commonly used, for example, to implement demand paging. Without the hardware support embodied in that interrupt, it is impossible for the software to implement demand paging in an efficient fashion.[1]

External interrupts are usually configurable through a programmable interrupt controller, with connectivity to most major system peripherals like network interfaces and disk controllers. These can be polled with relative ease, but high-speed peripherals can trigger events with very high frequency such that timely processing of external events requires very

[1] The alternative to using hardware support to implement demand paging is for the operating system to emulate all memory accesses performed by code which may cause a benign page fault, at significant performance cost.

Vector	Name	Cause
0	Divide by zero	Integer division by zero
1	Debug	Hardware debug event triggered
2	Non-maskable interrupt	External signal
3	Breakpoint	Executed `INT3` instruction
4	Overflow	Executed `INTO` instruction while overflow flag was set
5	Bound-range	`BOUND` instruction check failed (not in 64-bit code)
...	...	...
13	General protection	Attempted invalid memory operation in protected region
14	Page fault	No virtual memory mapping available for memory access
...	...	...
17	Alignment check	Misaligned memory access
18	Machine check	Hardware-detected error
19	SIMD floating point	SSE floating-point exception
...	...	...
255	External interrupt	Peripheral hardware interrupts

Table 3.1: Partial list of x86 interrupt vectors

frequent polls, in turn demanding an application that is aware of the response needs or very frequent preemption in order to perform polling. Though the complexity of writing efficient interrupt handlers is high, servicing peripheral interrupts is the best way to ensure good performance with minimal complexity.

3.2 Traditional Implementation

The typical approach to handling interrupts uses handwritten assembly language to implement the minimal required functionality as specified by the architecture, using that as a bridge into compiled code.

For example, Linux on x86 implements interrupt context switches in assembly which is linked against the C components of the kernel [32, `arch/x86/entry/entry_64.S`, line 423]. Stubs are generated for individual interrupt vectors which save the IRQ number, then jump to a common function which collects machine state in a known format, then calls the desired C function according to IRQ number. The flow of logic for this implementation is illustrated in Figure 3.1.

The `pt_regs` structure captures register contents on interrupt entry which can be (and are) used or modified by logic further down the call stack, under `do_IRQ`. Passed by reference, the pointer to this structure is constructed such that it includes the machine state saved on the stack by hardware on interrupt entry, which contains meaningful information like the address at which the interrupt triggered.

Similar logic for interrupt dispatching is possible to implement in Rust, in largely the same way. Linkage between assembly and Rust merely requires that the desired name be provided with public linkage and a known calling convention. For example, the Rust entry point in RKOS in called from an assembly stub which performs low-level system initialization, and is declared as shown in Listing 3.1. The `no_mangle` attribute prevents the compiler from applying its usual name mangling to the function's name so the name is predictable at link-time, and marking it as `extern "C"` causes the generated code to be compatible with the target platform's C calling convention.

```
#[no_mangle]
pub extern "C" fn kmain() {
    /* ... */
}
```

Listing 3.1: A Rust function with external linkage

It is thus possible to do the same kind of double dispatch performed by Linux in response to an IRQ in Rust, and this is in fact what some other experimental Rust

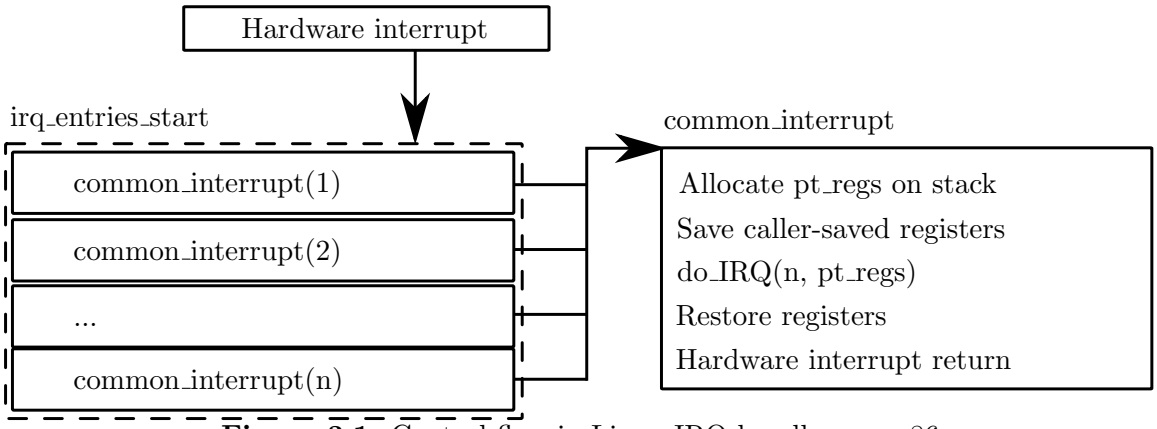


Figure 3.1: Control flow in Linux IRQ handlers on x86

operating system projects do [19, `src/src/arch/x86_64/interrupt_handlers.asm`] [18, `arch/x86/cpu/interrupt.rs`, line 89]. In systems which are designed to permit runtime configuration this approach is logical, in that only a small amount of code need be aware of the system-specific details of interrupt handling, abstracting away those details and dispatching to other components like device drivers.

In RKOS however, the goal of static configuration at the cost of support for varying hardware configurations in a single binary permits some improvements.

3.2.1 Performance Costs

In the dynamically-dispatched system, control flow will typically go through a trampoline before reaching a dispatch function which in turn calls the desired function. The trampolines and dispatcher are logically separate, especially if implemented in assembly and a compiled language respectively. Consequently, the structure of this system can be difficult for humans to follow. Further complicating the code of such a system, the requirement that individual vectors have largely similar but slightly different code implies a good implementation will add additional levels of indirection in source code to avoid excessive repetition.

The code to perform dispatch through those two levels incurs its own cost at runtime, as well as in logical code complexity. The cost of memory to hold the code is small but nonzero[2] and thus incurs a cost in cache consumption, but assuming large caches and relatively frequent interrupts makes memory consumption a minor concern. More relevant to real-world performance on modern CPUs is branching, particularly in making branches predictable [33].

The branch predictors in modern processors tend to be very good, but are still limited by the amount of resources allocated to branch prediction. With multiple trampolines feeding a single dispatch function, the trampolines themselves consume entries in the CPU's branch target buffer, and the target of the jump out of the dispatch function (to the final target) is an indirect jump, which may not even be predicted by the CPU. Most recent Intel CPUs (as a common example) have branch target buffers with on the order of 2^{10} entries, though this number varies widely between microarchitectures [34]. The comparison of buffer size to the number of interrupt vectors typically in use is favorable, with the majority of interrupts serviced by a given CPU coming from a small number of vectors. The number of interrupts for each vector on two representative Linux systems are given in Table 3.2 as

[2] The IRQ trampolines in x86_64 Linux are eight bytes each, for instance.

representative examples.[3] Despite the expected low ratio of unique vectors used in normal operation to BTB entries, the CPU may still choose to evict BTB entries corresponding to IRQ trampolines due to capacity-related pressure depending on branch frequency in non-interrupt code and frequency of IRQs.

By eliminating dynamic dispatch through interrupts, it is possible to avoid unnecessary pressure on the processor's branch prediction machinery, even eliminating branches other than those which are inherently part of the relevant interrupt service routine. Doing so imposes an important limitation on implementation: because ideally there are zero jumps the compiler must be capable of building the entire code path through the interrupt without external linkage (which would require at least a jump from human-written code to compiler-generated code). If the compiler is capable of generating the entire interrupt code path, it also becomes easier to avoid generating code for unused interrupt vectors. A similar (desirable) abstraction over machine-level details to the one that exists in a dynamically-dispatched system continues to exist in this scheme but with the complexity moved into the compiler rather than in a (potentially difficult to reuse) single-use implementation.

3.3 Interrupt Calling Conventions

In order to expose maximal interrupt code to the compiler to enable the efficiency improvements discussed above, a logical approach is to add functionality to the compiler such that it is capable of managing the entire interrupt lifecycle, from hardware context switch through returning to mainline execution. In support of that goal for the implementation of RKOS, I proposed and implemented support for assembly-free interrupt handlers in LLVM, and proposed use of such functionality in the Rust compiler (which uses LLVM as a backend for code generation).

The layout of the stack on entry to an interrupt on x86_64 is illustrated in Figure 3.2. An error code is provided with some interrupt vectors, and the flags register is automatically saved. The saved stack and code segment registers (CS and SS) combine with the instruction and stack pointers to indicate exactly the state of the machine when the interrupt was serviced. General-purpose registers are notable in their absence from the interrupt stack frame, so the ISR must save all registers which it uses. By inspecting or modifying the interrupt stack frame's contents and doing the same to the contents of the general purpose

[3] Note that no global statistics are tracked for common exceptions like page faults or division by zero. Page faults are expected to be common during normal operation, but most other exceptions usually cause termination of the offending process.

Vector	Machine A CPU 0	Machine B CPU 0	1	2	3	Use
0	40	18	0	0	0	timer
1	9	0	0	0	0	i8042
6	3	0	0	0	0	floppy
8	0	0	0	1	0	RTC
9	0	3	0	0	0	ACPI
10	362110	0	0	0	0	USB, VirtIO
12	134	3	0	0	1	i8042
23	0	128	34	14	6	USB
25	10507232	0	0	0	0	VirtIO request
27	26553978	18697815	4982250	3513106	2218695	VirtIO input, USB
28	488	116979589	44907620	27830861	16111854	VirtIO output, Disk
29	0	0	15	0	0	Management coprocessor
30	0	190847	32942	27592	23414	Sound
31	0	294227	94710	78589	64704	Graphics
32	0	171726536	19419748	17190010	13361988	Network
NMI	0	2315	1396	1384	1374	Non-maskable
LOC	580251090	53234543	48506830	48831507	48923785	Timer preemption
PMI	0	2315	1396	1384	1374	Performance monitor
IWI	47744	1	0	0	2	IRQ work
RES	0	15945685	15509188	13043577	12752507	Rescheduling
CAL	0	15805	14504	16008	12931	Function call
TLB	0	214956	193343	227811	202658	TLB shootdown
MCP	12070	1497	1497	1497	1477	Machine check poll

Table 3.2: Linux interrupt counts, by vector, for two representative systems. Vectors with zero recorded interrupts on either system are omitted. Machine A is a virtualized server, and machine B is a workstation. Where a vector is used by both, the use per-machine is separated by a comma.

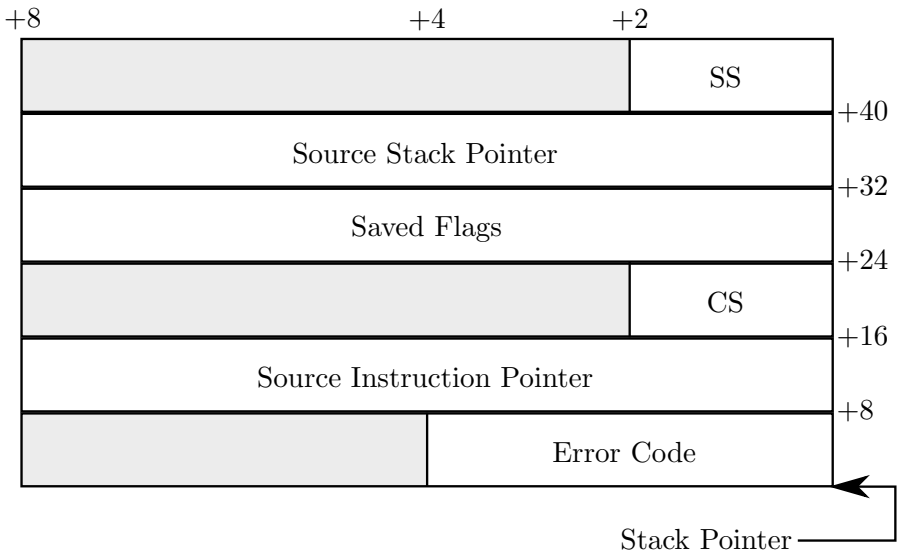


Figure 3.2: Stack layout on x86_64 interrupt entry

registers, an interrupt handler can both determine what the interrupted code was doing and arbitrarily redirect execution (usually first saving the state elsewhere for later resumption).[4]

The initial proposal for x86 interrupt support in LLVM allowed constructs like those in Figure 3.3 [35]. A function declared with the `x86_intrcc` calling convention may receive zero or one parameters, corresponding to whether the intended interrupt vector provides an error code. The error code is in the stack slot which would be occupied by the return address in a normal call, and all modified registers must be saved on use and restored before returning. The return from interrupt uses the `iret` instruction rather than `ret`.

This implementation inspired a proposal for support of a similar feature in Clang, LLVM's C frontend [36]. In this proposal, functions declared with a particular attribute gain access to an intrinsic function for access to the machine state stored on the stack, including the error code if applicable. Some individuals expressed skepticism regarding the performance benefits of the feature against the status quo combined with the additional compiler complexity to support it, and these concerns are also valid with Rust. However, some aspects of Rust (discussed further in Section 3.4.2) invalidate assumptions that are correct when programming in C and make a similar feature more desirable than it might be for a C compiler.

I proposed support for this compiler-supported form of interrupt handling in Rust as a user-visible calling convention, and generalized to a family of "interrupt" calling conventions varying across target architectures and interrupt types (for those architectures with multiple interrupt types), such that an x86 interrupt handler might be written as illustrated in Listing 3.2 [37].

```
extern "x86_64_interrupt" fn my_isr(ecode: u32) {
    /* ... */
}
```

Listing 3.2: Possible use of an x86 interrupt calling convention in Rust code

This proposal was met with a favorable response from interested Rust users and developers, but was ultimately not accepted for inclusion in the language in favor of support for naked functions (Section 3.4), a feature with greater precedent in other compilers and relevance to more use cases.[5]

[4] Legacy x86 provides hardware-assisted task switching in 32-bit code, but this support is not available for 64-bit code which must instead take advantage of interrupt state-saving.

[5] Though not accepted, the good reception for language-level interrupt calling conventions leaves the door open to later revisitation.

```
define x86_intrcc void
@test_isr_noparams() {
  ret void
}

define x86_intrcc void
@test_isr_oneparam(i32 %ecode) {
  call void asm sideeffect
    "movl $0, %eax",
      "m,~{eax}"(i32 %ecode)
  ret void
}
```

(a) LLVM IR

```
test_isr_noparams:
  iretq

test_isr_oneparam:
  pushq %rax
  movl 8(%rsp), %eax
  popq %rax
  addq $8, %rsp
  iretq
```

(b) Generated assembly

Figure 3.3: Sample LLVM use of the proposed x86 interrupt calling convention

Despite deferral of the proposal for language support in Rust, there was sufficient interest among LLVM developers to include a modified version of the original proposal for an x86 interrupt calling convention in LLVM itself. The feature which was eventually added adds a pointer parameter to interrupt handler functions which points to the stack region allocated by hardware on interrupt entry, roughly equivalent to part of the `pt_regs` structure in Figure 3.1, providing direct access to the interrupt stack frame which was lacking in the original proposal [38].

3.4 Naked Functions

Many C compilers, including MSVC, GCC and Clang support so-called "naked" functions [39, 40, 31], which do not have compiler-generated prolog or epilog code. Naked functions are most useful when the program must utilize a statically known calling convention which is not supported by the compiler. In other cases, the programmer may wish to implement a function depending on system-level details which are not exposed by the compiler, such as getting the address of a function's caller by directly reading from the stack.

In each of these cases the compiler's own prolog and epilog must be suppressed. In the former case register allocation may differ between the compiler's assumed calling convention and the actual one in use, causing the compiler to corrupt machine state, for example by using a register used for parameter passing as a temporary and modifying its value or inadvertently corrupting the caller's stack frame. In the latter example, the programmer may have no guarantees regarding the compiler's stack frame layout and so must construct their own to ensure the machine state is predictable.

In general, naked functions are not meant to contain regular statements in the compiled language but instead only inline assembly statements. While none of the aforementioned compilers enforce such a limitation, writing C statements in naked functions with GCC is specifically called out as an unsupported use and the consequences of doing anything with a naked function in LLVM are described as being "very system-specific." [31] MSVC does support mixed C and assembly in naked functions, made safe by inclusion of a `__LOCAL_SIZE` macro which provides the size of the containing function's stack frame. Use of that macro allows the programmer to write a custom function prolog and epilog which satisfy the needs of the function's body.

3.4.1 Naked functions in Rust

Inclusion of support for naked functions in Rust is a logical choice, given the prevalence of similar features in major C compilers. The Rust approach to code safety calls for a more precise design than the more cavalier approach taken by GCC and Clang, however.

For a Rust naked function, it should be impossible to generate incorrect code outside of a `unsafe` block. Consequently, the design which I proposed and was later accepted on an experimental basis stipulates that a naked function must contain no non-`unsafe` statements. Some individuals expressed a desire to be able to write Rust code in addition to inline assembly in naked functions despite the utter lack of safety guarantees regarding stack layout, so it was required that the entire body of such a function be marked unsafe rather than forbidding non-assembly code outright. Listing 3.3 illustrates a valid naked function under my design as it is currently implemented in the language [41].

```
#[naked]
unsafe extern "C" fn naked() {
    /* ... */
}
```

Listing 3.3: A naked function in Rust

3.4.2 Performance of Naked Functions

While not capable of exploiting the same optimizations that can be made when using a compiler-controlled interrupt calling convention, a Rust naked function for interrupt handling still permits compile-time code generation via macros so a single jump from a naked landing pad to a given IRQ's handler is possible to write without additional tooling, offering a slight improvement in code complexity (and thus expected performance) over Section 3.2.1's double dispatch.

An interrupt calling convention as discussed in Section 3.3 offers unique advantages when placed in the context of Rust compared to that of C. In particular, the Rust calling convention is deliberately unspecified (allowing for future changes without concern for backwards compatibility [42]) so it is impossible for a programmer to conform to that ABI in the general case, which has two important implications:

1. It is unknown which registers are callee-saved, so the programmer must pessimistically treat all registers as caller-saved.

2. Storage for parameter passing is unknown, so passing values to or receiving values from a Rust-ABI function is impossible (a Rust function declared with the C calling convention must be used as a bridge).

When compared to the C calling convention, the cost of the first point is particularly salient. In the common x86_64 C calling convention, there are six callee-saved registers of fifteen total general purpose registers, with up to six of the others being used to pass parameters [43]. In the case of interfacing from an ISR trampoline to a Rust-ABI function containing actual ISR logic, the cost of six additional stack pushes compared to a C implementation of similar logic may have a meaningful impact on performance.

CHAPTER 4

RKOS DESIGN

The overall design goal of RKOS is to expose large amounts of code to the optimizer at compile-time, taking advantage of the properties of a unikernel-type system and typical deployment patterns to simplify the implementing code and offer efficiency improvements over other software. Two particular observations guide the design:

- Mutual trust between components allows a shared, uniform address space.
- Virtualized runtime environments have uniform hardware configuration.

This chapter describes the particulars of my design, discussing the rationale for the embodied design decisions to build a system suitable for high-performance programs with relatively low complexity.

4.1 System Configuration

Configuration of the system is entirely performed at compile-time, within reasonable limits. Static configuration is desirable in that it can expose larger amounts of code to the optimizer where it might otherwise require dynamic dispatch according to what actual resources are available to meet system requirements. Additionally, avoiding the need for hardware-probing code (instead depending on having a configuration corresponding to that specified at compile-time) can simplify implementation.

Some aspects of a running system should be detected at runtime, either due to difficulty in manually specifying the requisite details or hardware features which can reasonably vary. Of particular note, memory capacity of the running system should be detected rather than specified in configuration, both because the physical memory map of a system can be onerously complex to specify manually and it may be useful to dynamically reprovision memory for a system without recompiling. In this instance, the cost of detecting memory at runtime is outweighed by the benefits.

Notable among aspects of the system which are configured dynamically in other systems but statically here is IRQ assignments: code generation may instead assign IRQs for particular peripherals to unique interrupt vectors. Combined with a compiler-mediated interrupt calling convention (Section 3.3), exposing the entire ISR code path to optimization may yield significant benefits.

The current implementation of configuration is minimal (described in Section 5.2), but a complete system for configuration would read a textual configuration file at compile-time, applying additional logic where appropriate (such as for assigning interrupt vectors to components servicing IRQs), generating the source code for a module definition linking abstract interfaces to the implementation for a given configuration. The configuration format should allow specification of hardware and system resources used by the hardware, for example the I/O ports allocated to a UART or the PCI address of an external controller or the drivers being used for implementation of a higher-level driver. From the top level, the system configuration file should represent a tree with hardware resources at the leaves, each owned by a hardware driver providing abstraction of the device's function.

External devices like network interfaces or consoles conform to a defined interface implemented via traits, and the application code uses well-known module paths to retrieve an implementation of a given feature matching the current configuration. In cases where multiple items provide a particular interface, either a single one may be specified as the default with application code specifically using a non-default at times, or a meta-driver may be used to implement dispatch to an appropriate lower-level implementation. The choice of a default or meta-driver should be specified by configuration.

Application configuration should also be specified in the compiled configuration, but can easily be modeled with a tree of names, each corresponding to a configuration value. These should be accessible through a configuration API, where lookup of a field's name (as a string) yields a value of a primitive type (such as string or integer). This approach also permits a filesystem-like interface for applications, but with the difference that the file contents would be part of the system binary image rather than stored elsewhere.

4.2 Memory Management

The overall memory map of the system is diagrammed in Figure 4.1. The symbolic names for region borders have statically-known values. `KIMAGE_BASE` and `KIMAGE_TOP` delimit the compiled system binary, where the image is loaded into memory for execution. The heap grows upward from `HEAP_BASE`, a fixed address into an unused region, and the stack mapping

for the executing thread grows downward from `THREAD_STACK_TOP` into the same unused region. The per-CPU interrupt stack grows downward from `KIMAGE_BASE`.

4.2.1 Virtual Memory

Changes to virtual memory mappings during normal operation are minimal, but virtual memory is important to allow a simple view of memory and mapping of stacks. On x86 machines especially, physical memory is usually highly fragmented like the representative memory map shown in Figure 4.2. The usable memory regions are highly fragmented, with regions reserved by firmware dispersed throughout the physical address space.

Virtual memory is used to remap the potentially-fragmented "available" memory regions into two contiguous blocks to back the system binary image at `KIMAGE_BASE` and heap at `HEAP_BASE`.

Stacks are allocated from the heap with appropriate alignment to virtual memory pages so they may be mapped to the stack regions of virtual memory. Interrupt stacks are fixed-size and allocated per-CPU, so once mapped during system startup the interrupt stack mapping is not modified. Individual thread stacks are mapped into the thread stack region from their true location on the heap for each context switch. Because any given thread can by definition execute on only one CPU at a time, the differences between the view of memory for any two CPUs are minimized and permit useful gains in performance.

The other cost of virtual memory is in maintaining page tables, which must be stored in memory. The typical minimum granularity of page allocation is four kilobytes with each page requiring four or eight bytes in a page table, implying memory overhead for page tables of between approximately 0.02 and 0.01 percent. Though relatively small, this cost is still as much as 8 megabytes per gigabyte of virtual address space, typically allocated on a per-CPU and per-process basis.

In RKOS, the majority of the system's memory map is filled by global mappings for the system binary image and heap, which do not change once configured. With appropriate choice of constant addresses, the system's mappings can be configured such that page tables are shared between CPUs excepting the minimal amount for per-CPU stacks.

Some systems also permit multiple page sizes (4 KB, 2 MB and 1 GB for x86_64, for instance), which can be used to further reduce page table overhead by mapping virtual memory regions with the largest possible granularity. Doing so is subject to alignment restrictions in both virtual and physical address spaces so tends to be difficult to do in the general case, but one-time mapping setup in RKOS for the largest portions of memory makes large-page allocation relatively easy to ensure with a greedy algorithm, provided

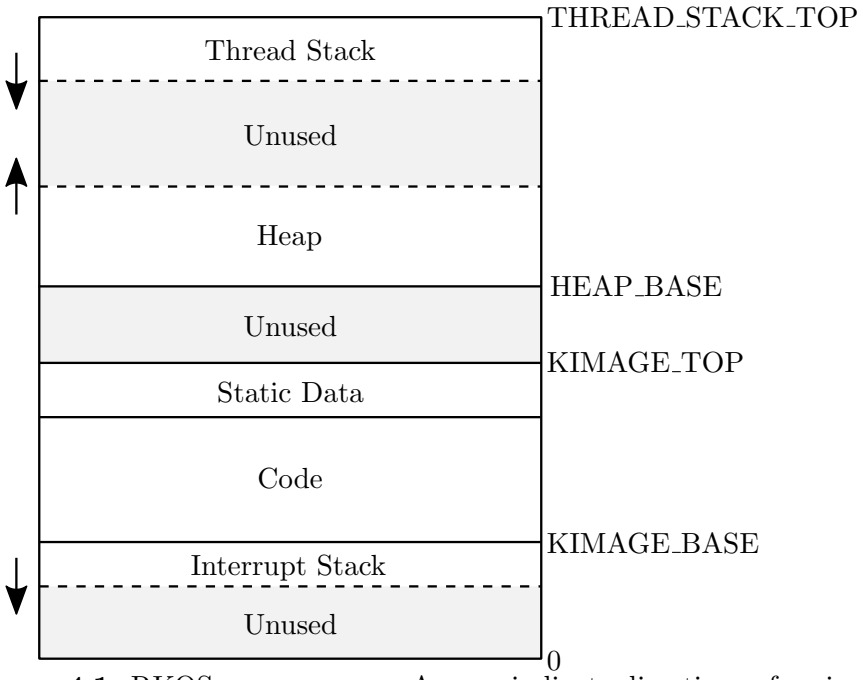


Figure 4.1: RKOS memory map. Arrows indicate directions of region growth.

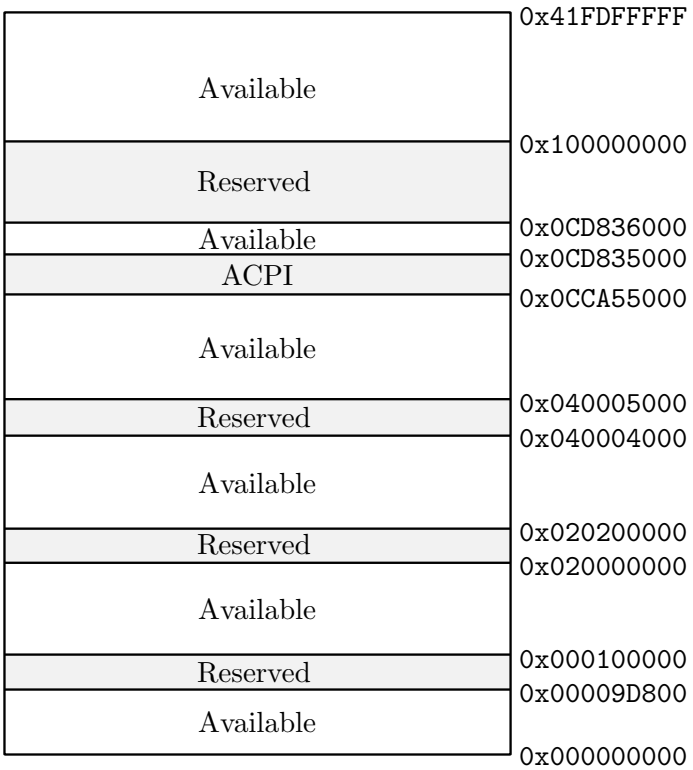


Figure 4.2: Representative x86 physical memory map

appropriate choices for base addresses. By contrast, opportunistic large-page allocation in typical operating systems can be difficult due to memory fragmentation.

The RKOS design also offers benefits in TLB efficiency: a CPU's TLB caches virtual memory mappings to avoid the cost of additional memory accesses (reading page tables) when computing addresses, but these caches must be invalidated when mappings change. TLB entries correspond to page table entries, so page table entries with larger granularity can cover larger address ranges while consuming fewer TLB entries. Between using large virtual memory pages where possible and making relatively few changes to mappings, RKOS attempts to maximize TLB hit rates by minimizing evictions, both explicit invalidations due to changed mappings and implicit due to the TLB having finite capacity. Existing work quantifying the costs of TLB misses shows performance improvements of between approximately two and fifty percent when switching from 4KB to 1GB virtual memory pages under workloads with large working sets [44], so significant improvements may also be expected from RKOS' goal of maximizing page sizes through early allocation.

4.2.2 TLB Shootdowns

In multiprocessor systems supporting shared memory, it is necessary to perform TLB shootdown operations when virtual memory mappings shared by multiple CPUs are updated to ensure the view of memory remains consistent. For instance, if CPU 'A' were to remap a region of memory in the page tables (stored in shared memory) for which CPU 'B' has the mapping cached in its TLB, a memory access to that region by CPU B will refer to the wrong portion of physical memory. At best this situation causes loss of written data or read of unrelated memory, and at worst could corrupt memory being used for some other purpose.

Performing a TLB shootdown involves sending an interrupt to any CPU which may have a copy of the mapping in question and waiting for acknowledgment of completion from every interrupted CPU. The minimum time cost of this operation on a modern CPU is around 500 nanoseconds in a two-processor system, with approximately linear scaling in servicing time against the number of CPUs. Under certain workloads the overhead of performing TLB shootdowns in a 128-CPU system approaches 50 percent, though more typical applications and hardware configurations see less drastic but still significant overhead [45]. While there exists work in the literature attempting to reduce the costs of TLB coherence, the focus largely tends toward modifying hardware architecture to provide partially- or fully-automatic interprocessor TLB coherence [46, 45, 47, 48] and is thus not relevant to a software-only design.

By dint of having a uniform global view of memory, RKOS is entirely freed of the need for TLB shootdowns, both simplifying the management of virtual memory and eliminating the performance costs of performing shootdowns.

4.2.3 Dynamic Allocation

The heap memory region is managed exclusively by a global allocator, the implementation of which is constrained only by the need to be thread-safe. The system does not necessarily have a page allocator like those contained in most operating system kernels, but such an abstraction might be useful to certain memory allocator implementations.

4.3 Threading Model

Threads are preemptible, scheduled non-cooperatively. Periodic timer interrupts preempt the running thread on each CPU and may cause the processor to switch context to a different thread, suspending the current one. Preemptive multitasking was chosen for the implementation largely due to precedent within existing systems, especially those supported by the mainline Rust standard library.

Cooperative multitasking could be a reasonable choice for a unikernel system because mutual trust between components obviates the concern of a single task consuming more than its fair share of resources, and can simplify concurrent programs because operations can be assumed to be atomic with respect to other threads unless they specifically yield the CPU. In order to work properly on a shared-memory multiprocessor however, such coarse-grained parallelism may not be used because other CPUs may modify memory even when a thread does not yield to other threads. Given the increasing prevalence of multiprocessor systems at all levels of complexity, the complexity of preemptive multitasking becomes a necessary aspect of utilizing hardware effectively.

4.3.1 Stack Checking

Stack checking is of high importance to Rust applications in general, because the application does not necessarily have direct knowledge of its stack usage and availability. In case of a stack overflow, the system must prevent memory corruption, either by writes to the stack overwriting some other part of memory or writes to invalid stack memory addresses having no effect and thus reading back meaningless values.

Historically, Rust used split stacks for this application, in which all function which allocate stack memory would perform a call to the `__morestack` runtime support function, which would check how much stack memory is free. If there is insufficient stack space to

contain the function's stack frame, it could either terminate the thread or allocate more memory. This approach is advantageous compared to traditional stack management in multithreaded applications, where stacks must be preallocated when created and cannot be resized.

With split stacks, it becomes easier to support a very large number of concurrent threads without concern for address space exhaustion, which is a particular concern on 32-bit platforms. For instance, the default thread size used on 32-bit x86 Linux is 2 MB, for an absolute maximum of only 2000 threads $\left(\frac{2^{32}}{2\times 2^{20}}\right)$ in the 32-bit address space [49]. For some highly-concurrent applications, this is not enough. On a platform with a larger address space however (such as modern 64-bit CPUs), the cost of allocating virtual memory with lazy commit is near-zero with address space capacity as a minimal concern.

Split stacks do however impose a cost in performance: code size increases by a few percentage points, and potentially large performance losses of nearly 20% were observed in Rust code when compiled with split stacks rather than conventional contiguous stacks [50]. In addition to the fixed costs, when a program's stack repeatedly crosses a size limit (such as repeated calling of a leaf function), the cost of opportunistically allocating and then freeing stack memory can reduce performance by more than a factor of four [51]. Motivated by the costs of split stacks, the Rust developers abandoned them as a way to perform stack checking [52].

4.3.1.1 Stack Probes

The intended replacement for split stacks in Rust is stack probes, in which functions merely attempt to access the stack memory required for the stack frame with maximum granularity of one virtual memory page. By probing every page the function's stack frame might touch, it is ensured that if the stack frame overflows the thread's stack it will attempt to access a guard page placed beyond the stack allocation and trigger a page fault which can be handled as desired.

The cost of this stack probe operation is very small, only one memory access per page filled by a function's stack frame. Further, because the probe only goes as far as the function's own stack frame there will in general not be any excessive cost of the access to memory itself: even if not already in cache, the probe will bring the memory in question into cache where it can be assumed to be needed by the function later in its execution.

4.3.2 Stack Management

In RKOS, the combination of stack probes and use of virtual memory in mapping stacks allows for management of stacks in a fashion invisible to the application. By leaving at least one virtual memory page unmapped between the top of the heap and bottom of a thread's stack, exhaustion of a thread's stack will trigger a page fault. To this point, such operation is the same as under typical operating systems in which a thread's stack must have its size specified at time of creation (though the system is free to lazily commit memory to the stack).

Because RKOS maps all thread stacks in a shared address space to a single location with contents depending on which thread is active on a given CPU, it is possible to transparently increase the size of all stacks no matter the initial reservation. The implementation is free to choose either a split or contiguous stack allocation according to its needs, balancing the costs of large allocations with those of managing a larger number of small allocations.

A limitation of this approach is that it becomes more difficult to pass pointers into a given thread's stack, because naively taking the address of an item in an executing thread's stack will result in a dangling pointer when context switches to any other thread. It remains possible to tie the lifetime of such a pointer to a sort of critical section preventing stack allocations (which might move the referent) and make the pointer refer to the stack as allocated in the heap memory region rather than as mapped in the stack region, but the implementation of such a feature is rather more complex than the same logic in a system without this kind of stack remapping.

4.4 Interrupts

In RKOS, only two interrupt vectors are used in a minimal configuration: the page fault handler and one external interrupt tied to an interval timer. RKOS does not implement demand paging of general memory, but the page fault handler is used to detect stack overflows as described in Section 4.3.2.

Servicing an interrupt requires switching stacks in the general case, usually with functionality provided by hardware. At the least, changing stack mappings requires either precise control over register allocation by the compiler to ensure that no stack memory is used by the code making the switch, or an independent stack from the one being remapped. Given that any practical implementation of the former would require implementation entirely with handwritten assembly, having independent stacks is a logical choice. An independent stack for all interrupts also helps avoid excess growth of thread stacks, which might otherwise be

expanded beyond the limits of that needed by a running thread due to interrupt demand for stack memory.

As discussed in previous sections, the interrupt stack is allocated on a per-CPU basis and mapped at a low address. It may be possible to implement growth of interrupt stacks in a fashion similar to that of thread stacks, but the per-CPU interrupt stack is specified as non-growable. This decision reduces complexity and the spectre of interrupt stack overflow might help encourage implementers of interrupt service routines to keep the contained logic simple and perform as much interrupt-related processing outside of interrupt contexts as possible, but it does run the risk of stack overflow during normal operation. The risk is low however, especially if the interrupt stack size can be tuned on a per-configuration basis.

CHAPTER 5

RKOS IMPLEMENTATION

This chapter discusses the implementation of the design outlined in Chapter 4 at a high level, and details when those details provide insights into the considerations involved in creating the implementation.

5.1 Source Organization

The source code of RKOS is split into a number of compilation units ("crates" in Rust parlance), with the overall structure depicted in Figure 5.1. Some of the crates compiled into a final binary are not RKOS-specific, and these are indicated with dashed outlines.

Within elements shown in Figure 5.1, dependencies and build order are managed by Cargo, the Rust ecosystem's package management tool [53]. Compiling all of these crates together emits a static library archive which is then linked with any necessary platform-specific startup code to make an executable system binary.

5.1.1 External Modules

Among the external modules (those that are not RKOS-specific), the majority of the code comes from the Rust standard library. `core` is the portion of the standard library which makes no assumptions about the execution environment, containing commonly-used types like `Option<T>` and traits like `Iterator`. While it is possible to write a Rust program without these constructs, doing so would be needlessly restrictive.

`collections` provides a variety of common data structures, including `Vec<T>` and several different mapping types. These depend on having a usable memory allocator (so cannot be part of `core`), provided by `alloc`. The actual allocator used by `alloc` is implemented in `rkos_alloc_imp`, with linkage through known symbol names.

`rlibc` provides simple implementations of intrinsic functions which are assumed to exist by the compiler backend. Specifically, these functions are `memcpy`, `memmove`, `memset` and

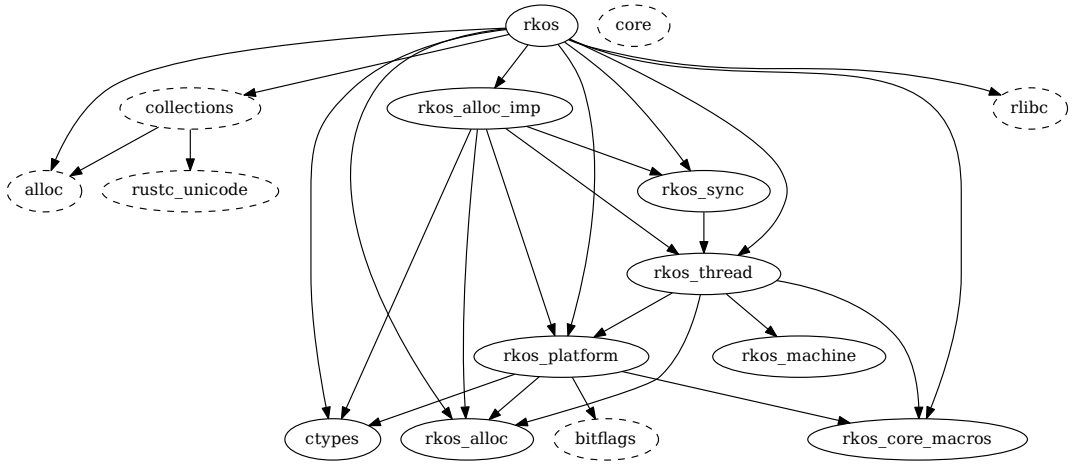


Figure 5.1: Dependency graph of RKOS compilation units. External packages are indicated by dashed outlines.

`memcmp`. The compiler may emit calls to these functions in some situations, so they must be provided even if not explicitly used.

5.1.2 C Dependencies

There are several external modules not depicted in Figure 5.1 because they are implemented in C and are thus not Cargo-built crates. In each of these cases it would be possible to implement the same functionality in Rust, but the nature of the functions they perform makes it such that the benefits of doing so would be minor.

Memory allocators are compiled into the `rkos_alloc_imp` crate, but with the majority of the implementation in mature C allocators instead. The current implementation supports TLSF [54] and dlmalloc [55], both with no source-level changes from their distribution versions. Because the allocators work almost exclusively with raw pointers and are quite mature, the benefits of reimplementing them in Rust would be minimal.

Runtime support libraries in use are compiler-rt [56] and openlibm [57]. Functions provided by compiler-rt may be used implicitly by the compiler to implement operations without simple translations to the target machine such as floating-point type conversions. Openlibm is used in support of certain more complex mathematical operations which may be useful, such as computation of floating-point square roots.

5.2 Configuration

Current support for configuration is minimal, because work to date has had no need for a system with the complexity outlined in Section 4.1. Compilation with Cargo supports "build scripts", Rust programs which are invoked before compiling a crate and can be used for code generation like that called for by the design for the RKOS configuration system.

The existing configuration capabilities are entirely through modification of constants in select modules. For example, the timer interrupt interval is currently specified by a constant in the `platform` crate. With implementation of the full configuration system, these constants would be moved to generated code but be used in the same way. Selection of implementations for external interfaces (currently the debug and regular consoles) is currently done by re-exporting the implementing type of each interface with a known name, again modified in the source code as desired but easily movable to generated code.

5.3 Targets

At the time of this writing, RKOS has only one fully-functional target but has been designed with the ability to support multiple targets. Portions of the necessary function-

ality for running on bare-metal x86_64 PC-like systems are implemented, but the working platform is a hosted x86_64 environment under Linux.

The Linux target builds a static ELF binary, in effect using the underlying OS kernel as a hardware abstraction layer. Running the system on top of a full-featured operating system in this fashion permits simpler testing and debugging, but at some performance cost.

UNIX signals are used to implement interrupts in the Linux target, which map accurately to the semantics of hardware interrupts: delivery of a signal preempts a running thread, and can be configured to switch to a different stack. Interrupt vectors may correspond to individual signals, though the existing meanings for most signals suggest particular interrupts and peripheral interrupts should go through a single signal with another field indicating which peripheral.

Timer interrupts are implemented with `SIGALRM`, triggered at a programmed interval by a kernel timer configured with the `setitimer` syscall. Page fault handling is done by handling `SIGSEGV`. Within signal handlers, the state of the system prior to preemption is provided as a parameter, in addition to details on the cause of the signal (such as fault address in case of page fault).

Virtual memory is implemented with the `mmap` syscall, with the equivalent of physical memory provided by an anonymous memory-mapped file. Regions of that file can then be mapped to multiple addresses, allowing dynamic stack mappings. Initialization of RKOS involves creating the heap mapping, allocating an interrupt stack, then switching to the interrupt stack and unmapping everything which was provided at process launch, including the original stack. At that point, RKOS has complete control over address space layout and may safely create its own threads. Multi-CPU systems can be implemented where each CPU is a process on the host operating system, with a shared file backing the "physical memory." Each process maintains its own virtual address space (analogous to the x86 CR3 register, specifying the page table root) but with shared physical memory.

5.4 Memory Management

The design of RKOS places the heap memory mapping at a fixed address, with size determined at runtime. All dynamic allocations, including both interrupt and thread stacks, come out of the pool embodied by that mapping, managed by a global allocator. The choice of allocator is part of the system's configuration, allowing an application to select an allocator appropriate to its needs.

5.4.1 Physical Memory

In order to adjust virtual memory mappings (such as creating a mapping to make a thread's heap-allocated stack appear in the stack region), the system must work with pointers representing regions of physical memory. In order to ensure a pointer to physical memory is not accidentally misinterpreted, RKOS defines a wrapper type, `PhysPtr<T>`.

A `PhysPtr` encapselates any necessary information to identify a location in physical memory, which is necessary in order to create virtual memory mappings by making the page table entry corresponding to a particular address refer to a given physical address. On plain hardware this is simply an address, but the Linux target extends this to include the file descriptor for the operating system file backing that memory because creating multiple virtual addresses referring to the same memory with `mmap` requires that they refer to the same backing file.

A physical pointer can be created from virtual by walking the current page tables, but modification of virtual memory mappings is more often concerned with mapping regions of memory which may not necessarily be contiguous in physical memory. The `BackingPARegions` iterator yields `PhysPtr`s for each contiguous block of physical memory backing a region of virtual memory, and each physical pointer can then be used to create a new mapping. There is not a safe way to convert a physical pointer to virtual address (explicit or otherwise), preventing accidental misuse.

5.4.2 Allocators

Memory allocators must be able to operate with nothing more than a pool of available memory (the heap memory region). The dlmalloc and TLSF allocator implementations both provide simple APIs following this requirement. The allocator must also be thread-safe, which in these two implementations is done with a simple global lock on the allocator. Because allocators are usable by only one thread at a time, ISRs must be able to run without allocating any memory because the preempted thread might be holding the allocator lock.

More sophisticated allocators could be used, but would require modification. ptmalloc [58], implemented on top of dlmalloc and used by default in the GNU C library, allocates multiple arenas for concurrent use, but has higher complexity as a result. jemalloc [59] is the preferred allocator for Rust applications on officially-supported platforms and exhibits very good performance, but is significantly more complex yet and would likely require the equivalent of a page allocator within the RKOS memory pool.

5.5 Threading

Threads are managed with their thread control block (TCB) as a handle. The TCB contains flags regarding the thread's current state, handles to other threads for when placed in a queue when waiting on external operations or ready to run but not scheduled to a CPU, and space for saved CPU state when not running.

Saved CPU state includes all user-accessible registers, excepting the FS and GS segment registers. While running, GS is loaded with a pointer to the thread's TCB (allowing code, especially interrupt handlers, to determine which thread is currently running) and FS is currently unused. CPU floating-point state is always saved, partially to reduce complexity and partially because FPU usage is expected to be relatively common in optimized code.

It is possible to conditionally enable the FPU on x86 in response to the exception raised by attempting to performing floating-point operations, thus avoiding the overhead in memory and time of saving or restoring the large (512 bytes or more) FPU state to or from memory. x86_64 unconditionally provides vector extensions to x86 through SSE2, which count as floating-point operations but are frequently used by optimized code to perform larger memory accesses or vectorize simple operations. Linux recently began performing "eager" FPU context switching motivated by these points [60], so doing the same in RKOS appears to be an appropriate choice.

A thread's TCB is placed inside its stack, reducing the overhead of tracking individual threads compared to traditional operating systems; allowing a thread access to its own state is not a security risk in a unikernel like RKOS because any executing code already has full access to system resources. By embedding the TCB inside the stack, the minimum demand for memory on a per-thread basis is reduced from two allocations (the TCB and stack) to only one, with minimum size dictated by the requirement that it be possible to perform virtual memory mappings for the thread.

The current scheduler for threads is a round-robin algorithm in which threads to run are taken from the head of the system ready queue and those that become ready are placed at the tail of the queue.

5.6 Sample Application

A sample application which computes an approximation of pi via Monte-Carlo methods demonstrates the basic system functionality of thread management with idiomatic Rust code. The source code to this application is provided in Appendix A, but this section highlights notable portions.

The demonstration is implemented as a single function, receiving a number of threads to run the simulation on and the number of points to test on each thread, returning a double-precision floating point value approximating pi.

```
fn monte_carlo_pi(npts: usize, nthreads: usize) -> f64
```

A vector (exactly the same data structure available no a typical rust application) is allocated to hold handles to threads which will be spawned to perform actual computation.

```
let mut handles = Vec::with_capacity(nthreads);

for _ in 0..nthreads {
    handles.push(spawn(move || {
        /* Generate and test points */
    }
}
```

In each thread, a random number generator based on the processor's time stamp counter (TSC) register is prepared. The random number generator implementation is a modified version of the Rust standard library's `librand`, but modified to remove OS-specific code which does not compile under RKOS for operations like reading random data from `/dev/random`.

```
use rand::distributions::{Range, IndependentSample};
use rand::{IsaacRng, SeedableRng};

let tsc = platform::get_timestamp();
let words: [u32; 2] = [(tsc >> 32) as u32, tsc as u32];
let mut rng = IsaacRng::from_seed(&words);
```

The random number generator is used to create random points in quadrant I of the Cartesian plane. If the magnitude of the resulting vector is less than 1, the point is noted as being inside a unit circle.

```
let range = Range::new(0f64, 1f64);
let mut hits: usize = 0;
for _ in 0..npts {
    let x = range.ind_sample(&mut rng);
    let y = range.ind_sample(&mut rng);

    if x*x + y*y <= 1f64 {
        hits += 1;
    }
}
```

Each thread's worker function returns the number of points it generated which were inside the unit circle. These are collected by the original thread, which waits for each of

the worker threads to complete, then computes the ratio of points inside the circle to total points generated which is directly proportional to the value of pi, assuming a truly random set of generated points and arbitrarily large number of samples.

```
let nhits: usize = handles.into_iter().map(|h| {
    h.join()
}).fold(0usize, |a, x| a + x);
4f64 * (nhits as f64) / (nthreads * npts) as f64
```

The `join` method of a thread's handle waits for it to complete and returns the value returned by the thread's worker function, with functionality similar to that of the Rust standard library. This is however a reimplementation rather than a port because a true port would require extensive modification to `libstd` for little gain.

5.7 Performance Evaluation

To characterize RKOS, the sample application described in Section 5.6 was used, and compared against a port of the same application compiled with the normal Rust Linux tooling. Both binaries were built with full optimization (including link-time optimization) and had debug symbols stripped. Sizes of the resulting binaries are compared in Figure 5.2.

The normal Linux application links against several dynamic libraries which are also shown. The RKOS binary image is larger than the Linux application by around a factor of four, but when shared libraries are included the Linux application's footprint is slightly larger.

Runtime memory footprint of each binary was measured by polling `/proc/pid/status` automatically at regular short intervals while the program was running, and the committed memory (`VmRSS`) and total virtual memory allocation (`VMSize`) were measured at each point in time. Additionally, the running program was constrained to run on only a single physical CPU with the `sched_setaffinity` syscall. This constraint was added to help ensure predictable performance, free of interference from any other programs running on the system and to maintain comparable timing because the RKOS Linux target currently lacks the ability to spawn multiple processes for multi-CPU operation. The memory use over time for each binary is plotted in Figure 5.3. In both versions of the program, four threads each testing 10^8 points are spawned.

Both versions of the program show quick startup, with the normal Linux application's virtual memory allocation reaching its final size in just over 20 milliseconds and with committed memory reaching steady-state even faster than that. The steps in the Linux application's virtual memory size appear to correspond to creation of new threads, and if

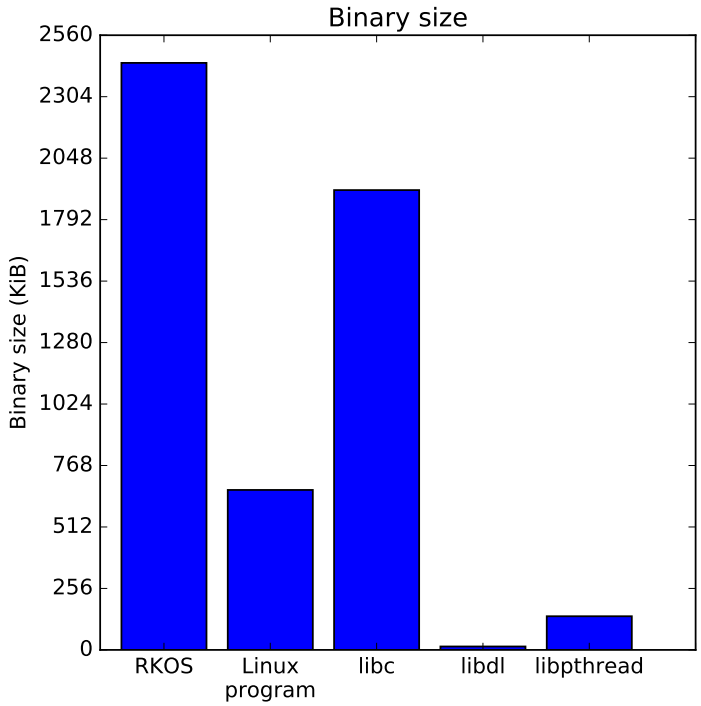
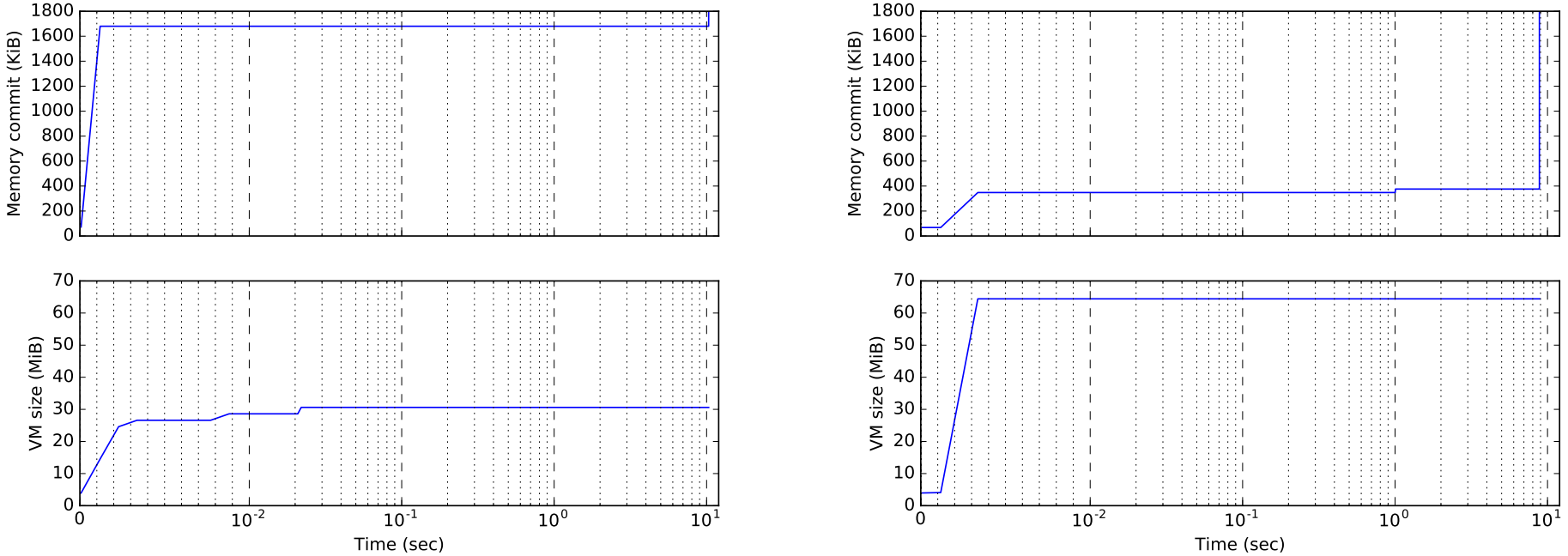


Figure 5.2: Binary size of the Monte Carlo sample application compared between the Linux application and Linux-hosted RKOS image with dynamic library dependencies.



(a) Linux application

(b) RKOS image

Figure 5.3: Memory usage of the Monte Carlo sample application over time for the normal Linux application and Linux-hosted RKOS image.

so should correspond with increases in committed memory as well. The actual amount of stack memory used by each thread however is very small compared to the entire working set, so this is not visible on the graph of committed memory simply because its magnitude is much smaller than the overall commit.

The RKOS application sees slightly faster startup to steady-state, but with a significant jump in commit at about one second after start. This appears to be the end of system startup, where the application code immediately takes over and allocates stacks for four threads and begins processing. The one second delay is due to an inefficiency of the current implementation where the initial switch to an interrupt stack occurs only after an initial timeout on the interval timer for preemption expires- the timeout is currently one second. The relatively large virtual memory allocation is accounted for by the configuration of a 64MB heap for the RKOS image, most of which goes unused.

The increase in committed memory for RKOS in the first four milliseconds of execution appears to correspond with the allocation of memory for the heap region, suggesting that much of its measured memory use is in kernel-managed page tables. A 64 MB memory region like the one allocated for the heap should encompass 16k 4KB VM pages, and a page table entry for each of those is eight bytes, requiring 128KB just for last-level page tables. Some additional space is required for higher page table levels as well, given the system's page table structure is a four-level 512-ary tree. Given complete control over its page tables (particularly for allocating large pages wherever possible) as intended for bare-metal targets, the committed memory could be significantly reduced from these measurements.

Both versions of the program show a large increase in committed memory at the end of execution, the cause of which is unknown. It would seem to be related to process shutdown as performed by the operating system, because the RKOS version is known not to perform any memory allocations at the end of its execution which would require that additional memory be committed.

In terms of absolute performance, the RKOS version of this program completes (exits) about one second faster than the regular Linux application. The RKOS program is configured with a very low preemption rate, so much of this difference appears to be the result of relatively frequent context switches,[1] while the RKOS application's threads are opaque to the operating system scheduler and are not switched with the same frequency. Some of the performance difference may also be accounted for by better optimization, but the fact that

[1]The underlying Linux kernel for this test was configured with a 300 Hz timer interrupt rate, which is expected to context-switch at the same rate.

the majority of this application's processing occurs in a tight loop which should be fully visible to the optimizer suggests that this should make little or no difference.

The observation of significantly improved performance with the RKOS program suggests the Linux target could be useful for real applications, despite the target's original goal of simplifying testing of platform-agnostic code in RKOS. The configurability of timer preemption permits non-interactive applications like this one to reduce overhead with a reduced timer frequency, while those demanding lower response times might increase timer frequency and attempt to strike a balance between overhead and responsiveness, even when running on systems where the user does not have kernel-level control over the operating system configuration. Provided a reasonably efficient implementation of platform support for running on bare metal (or virtualized systems), comparable or better performance could be expected in configurations deployed without an underlying operating system.

CHAPTER 6

CONCLUSIONS

Though the current implementation of this system is very limited in its functionality, it is possible to profile its performance to demonstrate that even if not maximally performant, the Linux-hosted RKOS system is at least competitive with a normal Linux-based Rust program.

6.1 Further Work

Though the current implementation of RKOS demonstrates functionality, parts of the implementation are insufficient for serious use. Each of these is relatively self-contained so these improvements could be made in an incremental basis and there are not any known design barriers to doing these things, but time constraints have prevented their implementation.

Though it is by design possible to expand thread stacks on demand, it is not possible in the current implementation. Expanding a stack requires allocation of memory, which as discussed in Section 5.4.2 demands an allocator capable of concurrent use by multiple threads. Because none of the implemented allocators currently support concurrent use of any kind, stack growth is not yet possible.

Rust on officially-supported platforms supports stack unwinding in case of panic in addition to simply terminating the program. The ability to recover from unexpected failures is an important feature for a unikernel which should not simply terminate, so implementation of stack unwinding to recover from panics at known points in the application should be an important consideration.

Thread scheduling is currently very limited with its purely round-robin scheme. In order to be useful for time-critical use cases (where other systems with different approaches of memory safety guarantees may not be appropriate) a more intelligent scheduler will likely be needed, in addition to careful characterization and optimization throughout RKOS.

The inability of the Linux target to support multiple CPUs precludes true concurrent applications. Similarly, support for bare-metal or virtualized targets must be completed. Earlier versions of the system were run on bare metal, but those parts of the code base must be reconditioned to account for later changes generalizing the system out to multiple targets and have some functionality added.

APPENDIX

MONTE-CARLO SAMPLE APPLICATION

This appendix contains the relevant source code to the Monte-Carlo pi computation described in Section 5.6 in its entirety, and representative output from running the program.

First, the computation function itself, which spawns threads which generate random points and collects their results. Each thread prints its own statistics to the debug console before returning.

```
fn monte_carlo_pi(npts: usize, nthreads: usize) -> f64 {
    use collections::Vec;
    let mut handles = Vec::with_capacity(nthreads);

    for _ in 0..nthreads {
        handles.push(spawn(move || {
            use rand::distributions::{
                Range, IndependentSample
            };
            use rand::{IsaacRng, SeedableRng};

            let tsc = platform::get_timestamp();
            let words: [u32; 2] = [(tsc >> 32) as u32,
                                   tsc as u32];
            let mut rng = IsaacRng::from_seed(&words);

            // Test npts points
            let range = Range::new(0f64, 1f64);
            let mut hits: usize = 0;
            for _ in 0..npts {
                let x = range.ind_sample(&mut rng);
                let y = range.ind_sample(&mut rng);

                if x*x + y*y <= 1f64 {
                    hits += 1;
                }
            }

            debug!("Computation thread finished with {}/{}
```

```
                    hits", hits, npts);
            hits
        }));
    }

    let nhits: usize = handles.into_iter().map(|h|
       h.join()).fold(0usize, |a, x| a + x);
    4f64 * (nhits as f64) / (nthreads * npts) as f64
}
```

This function is called from within `kmain` as follows, passing constants in and printing the results to the main console:

```
const npts: usize = 100000000;
const nthreads: usize = 4;
let pi = monte_carlo_pi(npts, nthreads);
println!("Pi is approximately {}", pi);
println!("(Computed with {} points over {} threads)",
         npts * nthreads, nthreads);
```

The run of such a binary resembles the following, including a large number of debug messages from various subsystems.

```
rkos_platform::imp: Mapped 0x4000000 bytes from memfd FileHandle(3) for heap at 0x40000000
rkos_thread: RSP at 0x7ffffffed28, offset 8 from ToS
rkos_platform::imp::threading: Allocated signal stack (0x4000 bytes) at 0x40006000
rkos_platform::imp::threading: mapping 0x4000 bytes from 0x40006000 at 0x1fc000 for interrupt stack
rkos_platform::imp: map_at_va(PhysPtr { fileno: FileHandle(3), offset: 24576, _mark: PhantomData })
                    to 0x1fc000 len 0x4000
rkos_platform::imp::threading: Mapped signal stack at 0x1fc000
rkos_platform::imp: map_at_va(PhysPtr { fileno: FileHandle(3), offset: 4096, _mark: PhantomData })
                    to 0x7fffffffb000 len 0x4000
rkos_thread: Mapped thread stack of 16384 bytes for thread Id(2)
rkos_thread: Boxed closure for spawn at 0x40000420
rkos_thread: RSP at 0x7ffffffed28, offset 8 from ToS
rkos_thread: Boxed closure for spawn at 0x40000470
rkos_thread: RSP at 0x7ffffffed28, offset 8 from ToS
rkos_thread: Boxed closure for spawn at 0x400004c0
rkos_thread: RSP at 0x7ffffffed28, offset 8 from ToS
rkos_thread: Boxed closure for spawn at 0x40000510
rkos_thread: RSP at 0x7ffffffed28, offset 8 from ToS
rkos_platform::imp::threading: PREEMPT with GS (assumed TCB) = 0x40004d30
rkos_platform::imp: map_at_va(PhysPtr { fileno: FileHandle(3), offset: 45056, _mark: PhantomData })
                    to 0x7fffffffa000 len 0x5000
rkos_thread: Mapped thread stack of 20480 bytes for thread Id(3)
rkos_thread: PREEMPT schedule(2) -> 3
rkos_thread: In ABI bridge, closure reading from 0x40000420
rkos: Computation thread finished with 78543190/100000000 hits
rkos_platform::imp::threading: PREEMPT with GS (assumed TCB) = 0x4000fd30
rkos_platform::imp::threading: PREEMPT with GS (assumed TCB) = 0x4000fd30
rkos_platform::imp: map_at_va(PhysPtr { fileno: FileHandle(3), offset: 69632, _mark: PhantomData })
                    to 0x7fffffffa000 len 0x5000
rkos_thread: Mapped thread stack of 20480 bytes for thread Id(4)
rkos_thread: PREEMPT schedule(3) -> 4
rkos_thread: In ABI bridge, closure reading from 0x40000470
rkos: Computation thread finished with 78540233/100000000 hits
rkos_platform::imp::threading: PREEMPT with GS (assumed TCB) = 0x40015d30
rkos_platform::imp: map_at_va(PhysPtr { fileno: FileHandle(3), offset: 94208, _mark: PhantomData })
                    to 0x7fffffffa000 len 0x5000
```

```
rkos_thread: Mapped thread stack of 20480 bytes for thread Id(5)
rkos_thread: PREEMPT schedule(4) -> 5
rkos_thread: In ABI bridge, closure reading from 0x400004c0
rkos: Computation thread finished with 78542870/100000000 hits
rkos_platform::imp::threading: PREEMPT with GS (assumed TCB) = 0x4001bd30
rkos_platform::imp: map_at_va(PhysPtr { fileno: FileHandle(3), offset: 118784, _mark: PhantomData })
                    to 0x7fffffffa000 len 0x5000
rkos_thread: Mapped thread stack of 20480 bytes for thread Id(6)
rkos_thread: PREEMPT schedule(5) -> 6
rkos_thread: In ABI bridge, closure reading from 0x40000510
rkos: Computation thread finished with 78542650/100000000 hits
rkos_platform::imp::threading: PREEMPT with GS (assumed TCB) = 0x40021d30
rkos_platform::imp: map_at_va(PhysPtr { fileno: FileHandle(3), offset: 4096, _mark: PhantomData }) to
                    0x7fffffffb000 len 0x4000
rkos_thread: Mapped thread stack of 16384 bytes for thread Id(2)
rkos_thread: PREEMPT schedule(6) -> 2
Pi is approximately 3.14162466
(Computed with 400000000 points over 4 threads)
```

REFERENCES

[1] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the virtual library operating system," *Queue*, vol. 11, no. 11, p. 30, 2013.

[2] MirageOS developers, "Mirage OS: A programming framework for building type-safe, modular systems." https://mirage.io/. Accessed May 30, 2016.

[3] Galios, Inc, "The Haskell Lightweight Virtual Machine (HaLVM): GHC running on Xen." https://github.com/GaloisInc/HaLVM. Accessed May 30, 2016.

[4] Erlang on Xen team, "Erlang on Xen - at the heart of super-elastic clouds." http://erlangonxen.org/, 2012. Accessed May 30, 2016.

[5] Rumprun authors, "Rump kernels." http://rumpkernel.org/. Accessed May 30, 2016.

[6] A. Kivity, D. Laor, G. Costa, P. Enberg, N. HarEl, D. Marti, and V. Zolotarov, "OSv: optimizing the operating system for virtual machines," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pp. 61–72, 2014.

[7] I. Briggs, M. Day, Y. Guo, P. Marheine, and E. Eide, "A performance evaluation of unikernels," 2015.

[8] G. Scherer, "Measuring GC latencies in Haskell, OCaml, Racket." http://prl.ccs.neu.edu/blog/2016/05/24/measuring-gc-latencies-in-haskell-ocaml-racket/. Accessed May 30, 2016.

[9] H. G. Baker, "The treadmill: real-time garbage collection without motion sickness," *ACM Sigplan Notices*, vol. 27, no. 3, pp. 66–70, 1992.

[10] P. Cheng and G. E. Blelloch, *A parallel, real-time garbage collector*, vol. 36. ACM, 2001.

[11] D. F. Bacon, P. Cheng, and V. Rajan, "A real-time garbage collector with low overhead and consistent utilization," *ACM SIGPLAN Notices*, vol. 38, no. 1, pp. 285–298, 2003.

[12] E. Altman, J. S. Auerbach, P. S. Cheng, and D. P. Grove, "Metronome." http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=175. Accessed June 6, 2016.

[13] D. F. Bacon, P. Cheng, and S. Shukla, "And then there were none: a stall-free real-time garbage collector for reconfigurable hardware," in *ACM SIGPLAN Notices*, vol. 47, pp. 23–34, ACM, 2012.

[14] T. Leonard, "Optimizing the unikernel." http://roscidus.com/blog/blog/2014/08/15/optimising-the-unikernel/, 2014. Accessed June 6, 2016.

[15] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *ACM SIGPLAN Notices*, vol. 46, pp. 283–294, ACM, 2011.

[16] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.," in *OSDI*, vol. 8, pp. 209–224, 2008.

[17] "The Rust Programming Language." https://www.rust-lang.org/. Retrieved May 26, 2016.

[18] rustboot contributors. https://github.com/pczarn/rustboot/. Revision 6e502fac402571811b8482f4460e850d58df5f5c.

[19] intermezzOS contributors. https://github.com/intermezzOS/kernel/. Revision 8d1511fee676f79f6a03b73e761f556fc6d81ded.

[20] Redox Developers, "Redox - Your Next(Gen) OS." http://www.redox-os.org/. Accessed May 26, 2016.

[21] A. Turon, "Abstraction without overhead: traits in Rust." http://blog.rust-lang.org/2015/05/11/traits.html, 2015.

[22] The Rust Project Developers, "The Rust reference." https://doc.rust-lang.org/reference.html. Version 1.8.0.

[23] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, "Region-based memory management in Cyclone," in *ACM Sigplan Notices*, vol. 37, pp. 282–293, ACM, 2002.

[24] ISO, "Programming languages – C," ISO/IEC 9899:1999, Internationanl Organization for Standardization, Geneva, Switzerland, 1999.

[25] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: Analyzing the impact of undefined behavior," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 260–275, ACM, 2013.

[26] LLVM Project, "UndefinedBehaviorSanitizer." http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. For Clang version 3.9.

[27] D. Austin and J. V. Stoep, "Hardening the media stack." http://android-developers.blogspot.com/2016/05/hardening-media-stack.html, 2016.

[28] S. Christey and R. A. Martin, "Vulnerability type distributions in CVE," *MITRE report, May*, 2007.

[29] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey, "2011 CWE/SANS top 25 most dangerous software errors," *Common Weakness Enumeration*, vol. 7515, 2011.

[30] L. Torvalds, "Re: Invalid compilation without -fno-strict-aliasing." https://lkml.org/lkml/2003/2/26/158, 2003. Linux Kernel mailing list archives.

[31] LLVM Project, *LLVM Language Reference Manual*, LLVM 3.9 ed. http://llvm.org/docs/LangRef.html.

[32] L. Torvalds, "Linux 4.6." https://kernel.org/.

[33] S.-T. Pan, K. So, and J. T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," in *ACM Sigplan Notices*, vol. 27, pp. 76–84, ACM, 1992.

[34] A. Fog, *The microarchitecture of Intel, AMD and VIA CPUs*. http://www.agner.org/optimize/microarchitecture.pdf, 2016-01-16 ed.

[35] P. Marheine, "D12498 X86: add an interrupt calling convention." http://reviews.llvm.org/D12498, 2015. LLVM patch for review.

[36] H. Lu, "RFC: Support x86 interrupt and exception handlers." http://lists.llvm.org/pipermail/cfe-dev/2015-September/044924.html, 2015. LLVM C frontend mailing list archive.

[37] P. Marheine, "Support interrupt calling conventions." https://github.com/rust-lang/rfcs/pull/1275, 2015. Rust language RFC.

[38] A. Aboud, "D15567 support of x86 interrupt and exception handlers in LLVM." http://reviews.llvm.org/D15567, 2015. LLVM patch for review.

[39] Microsoft, "Naked function calls." https://msdn.microsoft.com/en-us/library/5ekezyy2.aspx.

[40] Free Software Foundation, "Arm function attributes." https://gcc.gnu.org/onlinedocs/gcc-6.1.0/gcc/ARM-Function-Attributes.html. Part of the GNU Compiler Collection manual for version 6.1.

[41] P. Marheine, "Naked functions." https://github.com/rust-lang/rfcs/blob/master/text/1201-naked-fns.md. Rust RFC 1201.

[42] C. Gaebel, "Tail call compatibility." https://mail.mozilla.org/pipermail/rust-dev/2014-December/011471.html. Rust development mailing list archive.

[43] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, "System V Application Binary Interface," 2005.

[44] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 237–248, ACM, 2013.

[45] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 340–349, IEEE, 2011.

[46] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 62–63, IEEE, 2011.

[47] S. Srikantaiah and M. Kandemir, "Synergistic TLBs for high performance address translation in chip multiprocessors," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 313–324, IEEE Computer Society, 2010.

[48] P. J. Teller, "Translation-lookaside buffer consistency," *Computer*, vol. 23, no. 6, pp. 26–36, 1990.

[49] The Linux man-pages Project, "pthread_create(3)," 2015. Version 4.06.

[50] Zoxc, "Replace stack overflow checking with stack probes." https://github.com/rust-lang/rust/issues/16012. Rust issue #16012.

[51] The Go Programming Language developers, "Contiguous stacks." https://docs.google.com/document/d/1wAaf1rYoM4S4gtnPh0zOlGzWtrZFQ5suE8qr2sD8uWQ/pub. Accessed May 26, 2016.

[52] B. Anderson, "Abandoning segmented stacks in rust." https://mail.mozilla.org/pipermail/rust-dev/2013-November/006314.html, 2013. Rust development mailing list archive.

[53] "Cargo, Rust's package manager." http://doc.crates.io/. Accessed May 26, 2016.

[54] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: A new dynamic memory allocator for real-time systems," in *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pp. 79–88, IEEE, 2004.

[55] D. Lea and W. Gloger, "A memory allocator." http://g.oswego.edu/dl/html/malloc.html, 1996. Accessed May 28, 2016.

[56] H. Hinnant and compiler-rt authors, "compiler-rt runtime libraries." http://compiler-rt.llvm.org/.

[57] The Julia Project, "openlibm." https://github.com/JuliaLang/openlibm.

[58] W. Gloger, "ptmalloc3." http://www.malloc.de/en/. Accessed May 28, 2016.

[59] J. Evans, "A scalable concurrent malloc (3) implementation for FreeBSD," in *Proc. of the BSDCan conference, Ottawa, Canada*, 2006.

[60] A. Lutomirski, "[PATCH v2 5/5] x86/fpu: Default eagerfpu=on on all CPUs." http://lkml.kernel.org/r/ac290de61bf08d9cfc2664a4f5080257ffc1075a.1453675014.git.luto@kernel.org, 2016. Linux kernel patch, accessed May 30, 2016.