Binary-Compatibility with Linux Application for a Unikernel Written in Rust

Christopher Densham

Supervisor: Dr. Pierre Olivier

April 2021

Abstract

Unikernels offer an alternative to traditional virtual machines and containers as a lightweight virtualization mechanism for use in cloud computing. However, barriers to their widespread adoption include the difficulty in porting existing applications, and their lack of safety and security. Unikernels are typically categorized as either written in memory unsafe languages with easy application porting or written in a memory safe language with a significant cost to port applications.

This project explores the possibility of having a unikernel written in the memory safe language Rust which can execute precompiled Linux binary applications. This would provide a unikernel which is both memory-safe and yet requires minimal to no effort to port existing Linux applications. Thus, eliminating two of the barriers to the adoption of unikernels in the cloud computing space.

The project attempts to replicate some of the work in the existing binary compatible HermiTux unikernel written in C, applying this work to the RustyHermit unikernel which is written in Rust.

Acknowledgements

I would like to thank my project supervisor Dr. Pierre Olivier for his help and guidance throughout the project. My project partner Laurent Pool for his contribution and assistance during the project. My parents for their belief and support in enabling me to re-enter education. Finally, I would like to thank my friends and family, in particular Hayley, for their perseverance and support over the years and particularly over this last difficult year.

Impact of Covid-19

The lockdown due to the pandemic has had a minimal impact on the planning and execution of the project in a practical sense. Remote access was provided to a machine capable of the virtualization required for development and testing. Meetings have taken place using video conferencing instead of face to face.

However, it has limited the ability to engage in collaborative practices such as pair programming with my project partner or have ad hoc discussions and meetings. The lack of contact with other students and academics outside of the project also limits the opportunity to explain and discuss your work with those who do not already know about it. This is often a useful way of both gauging your own knowledge and gaining an outside perspective.

Table of Contents

1 Introduction 6
1.1 Motivation 6
1.2 Aims 7
1.3 Summary 7
2 Background & Related Works 7
2.1 Background 7
2.2 Related Works 9
3 Design 10
4 Implementation 12
4.1 RustyHermit 13
4.2 First-stage Loader 13
4.2.1 Relocate the Kernel 13
4.2.2 Load the Application 15
4.2.3 Pass Application Metadata to the Kernel 15
4.3 Second-stage Loader 15
4.4 Challenges 21
4.4.1 Remote Access 21
4.4.2 Learning a New Language 22
4.4.3 Dependencies 22
4.4.4 System Bugs 22
5 Functional Evaluation 23
6 Comparison to Similar Work 23
7 Conclusion & Future Works 24
7.1 Future Works 24
7.2 Conclusion 25

Appendix A hello_world_asm.s 27

References 28

Table of Figures

Figure 1: A VM / hypervisor stack showing a traditional VM alongside a traditional unikernel and a binary-compatible unikernel. 8

Figure 2: Graphical overview of the system 10

Figure 3: Guest address space. Showing separation of application and kernel. 14

Figure 4: The initialised stack 16

1 Introduction

1.1 Motivation

The explosion of cloud computing usage has resulted in an expansion in multi-tenancy servers from large cloud service providers such as Amazon, Google, and Microsoft [11]. Multi-tenancy allows many applications, such as Infrastructure as a Service (IaaS), Function as a Service (FaaS), Software as a Service (SaaS) products and microservices, from multiple users to run on a single physical server [3]. Container technology is an increasingly common choice for running applications due to their low start-up overheads and minimal memory footprint when compared to virtual machines (VMs) [3, 11].

Despite the advantages that containers provide they have less isolation than VMs and are therefore inherently less secure. This constitutes a risk of information leakage [3, 11, 15]. Unikernels are a minimal operating system (OS) suitable for cloud applications and could provide both the performance and disk size advantages of a container and the security and safety of a VM. Unikernels can provide a minimal, lightweight kernel with only the system calls required to run the application and which terminates when the application finishes execution [11, 15]. They are short-lived so inherently more secure, due to the reduced attack surface, and more efficient than a long-lived VM, yet still have isolation for safety and security.

There are two distinct categories of unikernel. Firstly, there are unikernels which support legacy applications and are language agnostic. They are written in a memory unsafe language such as C. Secondly are the unikernels written in a memory safe language providing greater security and safety but which only support applications written in a high-level language [19].

Unikernels are designed to run only a single application, this is usually compiled as part of the unikernel code [11]. This can limit the uptake of unikernels for running existing applications due to the effort required to create them. The desired application must be rewritten for the unikernel. This causes a significant amount of work for the programmer to port existing applications. It may even be impossible for applications where the source code is not available [15].

A unikernel which can run existing applications without any porting effort could help with the uptake of unikernels as an alternative to containers. Such research projects exist in legacy languages such as C [15]. However, memory safety in C is difficult to enforce and prone to errors from the programmer [8]. It is also a significant source of vulnerabilities [8]. Although unikernels written in a memory safe language exist they still require the effort of porting the application to the unikernel [8]. A binary compatible unikernel written in a memory safe language would remove the burden on the programmer to port applications whilst also providing a safe and secure runtime environment.

1.2 Aims

The primary aim of the project is to modify a unikernel written in the memory safe language Rust so that it can execute a single Linux binary application passed to it at run-time. The ultimate target is the ability to run simple pre-compiled static C programs using this Rust unikernel which would allow the execution of NPB benchmarks and therefore measure the performance of the unikernel.

1.3 Summary

The project took the RustyHermit unikernel project and adapted it to run Linux binary applications using the work done with HermiTux on the HermitCore project as a reference. The exploratory nature of the project meant there were many unknowns about what was ultimately possible, and the potential challenges involved. Although we were not able to reach the goal of providing full binary compatibility with C programs, the ability to run compiled Linux applications written in assembly was demonstrated. The project was able to provide a proof of concept that it is possible to write a unikernel in Rust that can run statically compiled Linux binary applications and has laid much of the groundwork for compatibility with static C applications.

2 Background & Related Works

2.1 Background

VMs running standard operating systems are very useful. They allow users to run unmodified application code as though directly on a physical machine [10]. The guest VM itself is an application running on a host, providing transparent access to system

calls through a hypervisor layer [6]. This is particularly useful in cloud computing where compute time is leased to a user and therefore a physical server can have many users running many different VMs. VMs provide a means of separating the running systems securely and safely [10].

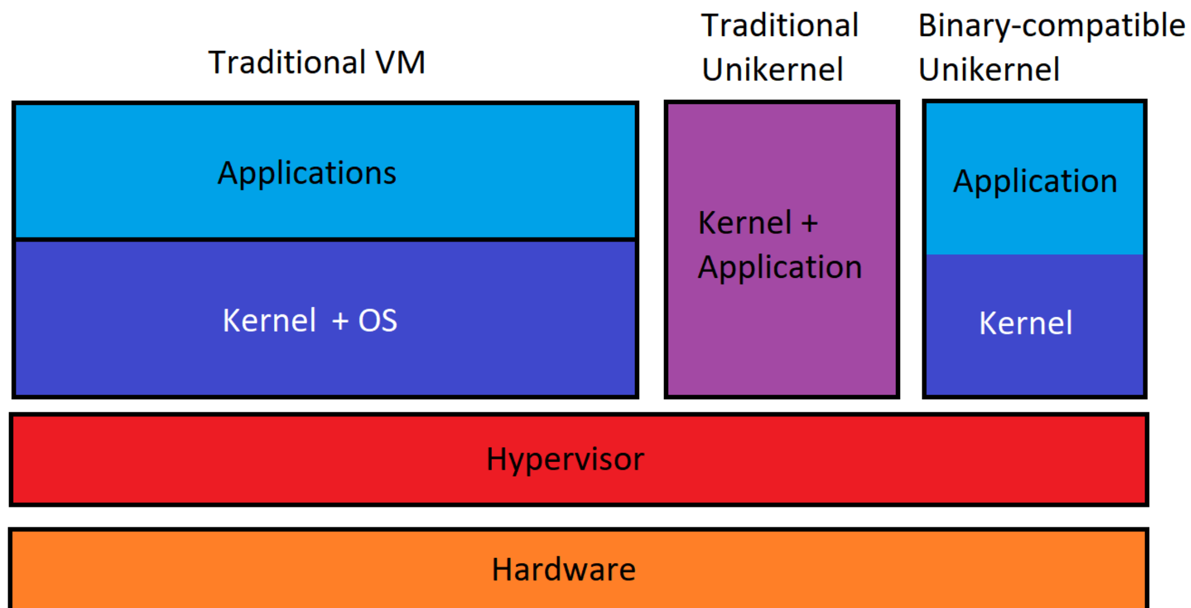

Figure 1: A VM / hypervisor stack showing a traditional VM alongside a traditional unikernel and a binary-compatible unikernel.

VMs are typically large, running full OSs which contain legacy code for supporting many years of software and hardware as well as adding an additional layer to the existing software layers [10]. The increasing need for an efficient way of running many applications on multi-tenancy servers has led to a widespread adoption of containers [3]. Like containers, unikernels are a form of lightweight virtualization but unlike containers they are still VMs so can benefit from isolation and therefore the security that comes with that [10].

Even though unikernels may solve some of the issues related to containers they are still immature, mostly existing as research projects, and their security has not been proven [13]. Whilst they do provide better isolation than containers, the application and kernel code run together in the same address space in privileged mode. Security mechanisms to protect against exploits have yet to be reliably developed [13, 16].

In addition to the security problems yet to be solved for unikernels another barrier to adoption is the requirement for applications to be rewritten and compiled as part of the unikernel. This can range from being reasonably straight-forward, for instance Lankes

et al. claim that porting a Rust application for RustyHermit is “almost trivial” [8], to being impossible in the case of proprietary software where the source code is unavailable [15]. Increasing compatibility of applications with unikernels is therefore an important area of research. If a unikernel can run existing binary applications, then one of the important hurdles to unikernels becoming production ready is overcome.

Unikernels can be written in either legacy languages such as C or in a memory safe language such as Rust. Unikernels run on the hypervisor layer and need to replicate the interface between the OS and the application. Legacy languages such as C are commonly used for system-level development as it provides unchecked access to memory and allows for greater performance for certain tasks [8]. Avoiding errors when writing C can be difficult even for experienced programmers [8] and memory-related bugs contribute to a significant number of vulnerabilities [4, 14]. Writing a unikernel in a memory safe language such as Rust would help to reduce bugs and therefore vulnerabilities resulting from incorrect use of memory.

Rust is a memory safe language; it uses the concept of ownership to manage memory instead of garbage collection or manually allocating and freeing memory as in other languages. This allows Rust to be safe as well as high performance [7]. These properties of Rust make it a good choice for developing system software such as a unikernel.

On the one hand there are unikernels written in legacy languages which are unsafe but provide easy porting of applications. Whilst on the other hand there are unikernels written in memory safe languages, but which require costly application porting. The objectives of the project are to explore the possibility of creating a unikernel which can port applications with minimal effort in addition to providing memory safety. If this can be shown to be possible then we will be able to demonstrate the best features of both legacy and memory safe unikernels.

2.2 Related Works

Research projects exist which use unikernels to run a Linux binary application for instance HermiTux[1] [15]. HermiTux is based on HermitCore[2], a unikernel operating system written in C and therefore suffers from the security issues inherent in memory

[1] https://ssrg-vt.github.io/hermitux/
[2] https://github.com/hermitcore/libhermit

unsafe languages, such as memory leaks and vulnerabilities [15]. It is adapted to run native Linux binary applications written in several languages such as C, C++, Fortran, and Python [15].

RustyHermit[3] is a project that has rewritten HermitCore in Rust for memory safety [8]. Rust provides a memory safety guarantee which it implements using the concept of ownership, the rules for which are checked at compile time [7], thus ensuring that memory leaks do not occur. RustyHermit is a lightweight kernel which runs on top of a hypervisor layer. In this case a KVM-based hypervisor called Uhyve. Adapting RustyHermit to load and run binary applications will provide both safety and compatibility.

The Linux Application Binary Interface (ABI) determines the interface between two binary applications, for example between a library and a user program. It consists of a load-time element and a runtime element [15]. The load-time ABI determines what binary formats are supported, how an application is loaded, and how the stack and the registers should be initialised when the application is started. The runtime ABI defines how to make system calls; the instructions and registers to use [12]. The ABI conventions will need to be applied in the project to load and run the binary application.

3 Design

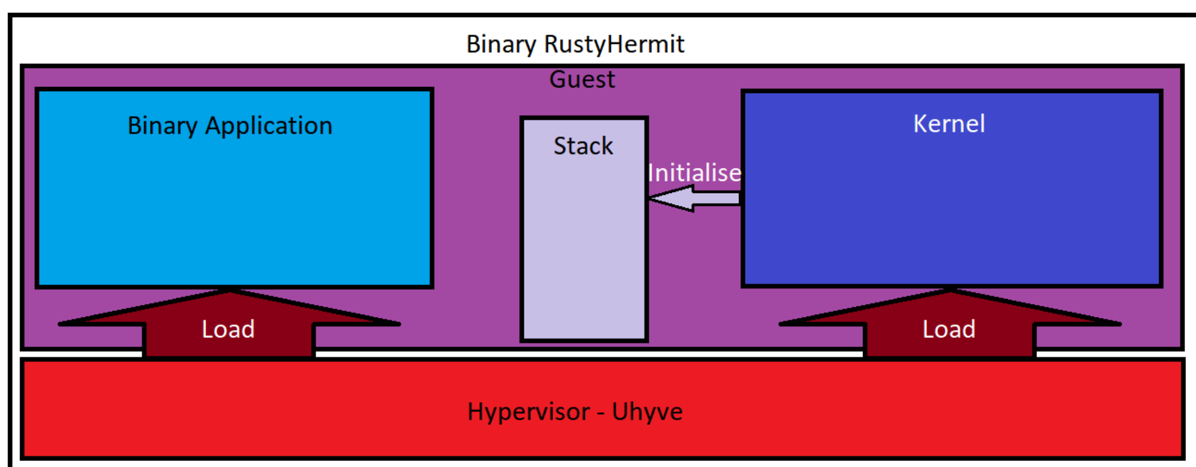

Figure 2: Graphical overview of the system

The binary compatible RustyHermit project consists of a hypervisor layer and a guest layer. Within the guest layer the kernel is loaded in a separate area of memory from

[3] https://github.com/hermitcore/rusty-hermit

the binary application code and data. The loading of both the application and the kernel is performed by the hypervisor. The hypervisor then passes control to the kernel. The kernel initialises the stack and registers ready for the application to execute and finally hands over control to the binary application. These stages constitute the load-time ABI. The development work to achieve this was completed by Christopher Densham.

When the application is running it requires the use of system calls. Unikernels always run in supervisor mode, therefore in the unikernel the system calls are simply function calls. For binary compatibility, the system calls need to be implemented as functions within the kernel. The runtime ABI determines how these system calls are implemented. Laurent Pool developed the system calls for the project.

As RustyHermit is a rewrite of HermitCore written in Rust, and HermiTux is a binary compatible version of HermitCore it makes sense to study both the HermiTux and the RustyHermit projects. HermiTux provides the functionality and RustyHermit the base from which we can adapt.

The RustyHermit binaries run on top of a hypervisor layer. There are two supported hypervisors for use with RustyHermit: Qemu and Uhyve. For this project Uhyve was selected as the hypervisor as the load process is much simpler and it would need to be amended. In addition, this is the same hypervisor used by HermiTux which was used as the example project. HermiTux uses a version of Uhyve written in C, whilst this project uses a version written in Rust, in keeping with the rest of the project. Uhyve was developed by the authors of HermitCore as a minimal hypervisor for running the unikernel on a Linux system using KVM [9, 15].

The hypervisor needs to initialise, allocate memory, and load the guest kernel into memory. In a traditional unikernel the kernel and application are compiled into a single binary. However, for binary compatibility, the application code and data are loaded separately from the kernel, although the kernel and the application will run in the same address space. For the project, the hypervisor will need to be adapted to load the application into a separate area of the address space, ensuring that the two binaries' areas do not overlap.

Once the hypervisor has initialised the system it can pass control to the kernel. The kernel then initialises, enabling system call support by setting a control register. The kernel can then set up a heap and a stack in memory and execute the application

code. For a binary compatible unikernel additional steps are required. The kernel needs to initialise the stack with the information expected by libraries to execute the application code. This can include command line arguments, environment variables, and the OS auxiliary variables that are expected by the early initialisation code that runs first in most modern application's binaries. Once this has been initialised the kernel can pass control to the application to execute its code.

The work to create a binary compatible unikernel in Rust was divided into two sections. The work to implement the runtime part of the ABI, and therefore the required system calls written in Rust was completed by Laurent Pool. A subset of all available system calls was chosen to be implemented. The choice was based on the most common system calls used in the kernel. Writing to stdout, file operations such as opening, closing and writing to files, and memory allocation. Implementing these common and basic system calls enables a wide range of binary applications to run. One of the focuses was on implementing the system calls required to run NPB benchmarks so that we could compare the performance of the binary compatible Rust unikernel with the performance of a standard Linux kernel.

The application loader, implementing the load-time ABI, was completed by Christopher Densham. This required first ensuring that the kernel and application were loaded into separate areas of the same address space. It is important to ensure that the kernel and application are loaded far enough apart that they do not overlap. As the project focused on static binaries the application and kernel locations were hardcoded. Once the application and the kernel are loaded into memory then the stack must be initialised with the required variables expected by libc. Once these are loaded then the kernel can start execution of the application.

4 Implementation

This section will discuss the details of the implementation of the loader in the unikernel and the reasons for the decisions which were made. It will also discuss some of the challenges faced when implementing this. For details on the implementation of the system calls please refer to the project report by Laurent Pool.

4.1 RustyHermit

There are two stages to initialising and loading a binary application in the unikernel, comprising of four main tasks.

1. First stage
    a. Relocate the kernel from the original location in RustyHermit.
    b. Load the application alongside the kernel code in the same address space, ensuring that there is no overlap within the memory.
    c. Pass the application point and other metadata to the kernel.
2. Second stage
    a. Craft the stack required for the application to execute and hand control to the application.

In between the two stages the kernel initialises, writing to a control register to enable system call support.

4.2 First-stage Loader

The first stage requires three main tasks to be completed. First the start point of the kernel needs to be relocated. Secondly, the binary application needs to be loaded into memory. Finally, metadata about the application needs to be passed to the kernel, most crucially the application entry point.

4.2.1 Relocate the Kernel

In the original version of RustyHermit, the kernel, which also contains the application code, is loaded by Uhyve at the address 0x400000. This is at the start of the traditional user space for standard Linux static binaries. For simplicity, the project focused on static binaries, so we wanted to load the application at this address. The kernel space would usually occupy the second half of the virtual address space which is usually 48 bits in modern x86-64 CPUs. In HermiTux it was noted that it was possible to fit the entire kernel into the address space below 0x400000 starting at 0x200000 [15]. This allowed the application code to be loaded at 0x400000 without any conflicts and gave the maximum amount of space for the application to use. As the project was studying HermiTux it was attempted to replicate this approach and load the kernel at the same location. The RustyHermit kernel already supports relocation so the option to move the kernel was explored. Unfortunately, the size of the kernel was 2.4 MB and therefore

larger than the available 2 MB of space, so it would not fit in the same location as in HermiTux. It is unclear if the kernel code can be refactored to fit in the 2 MB of available space. This is a potential area for future exploration.

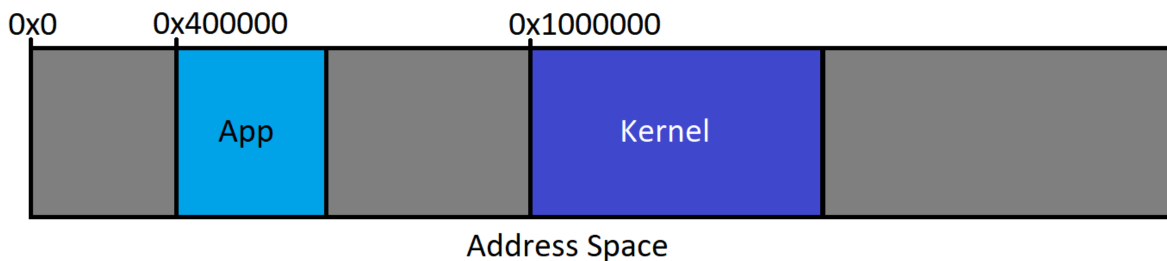

Figure 3: Guest address space. Showing separation of application and kernel.

As it was still desired to load the application at 0x400000, and it was not possible to locate the kernel below this the decision was made to load the kernel higher up in the address space. The kernel would not have a separate address space to the application code so it was important that there would be no interference. The upper limit to the address space is dependent on the size of the guest memory. This can be declared at load time by setting the `HEMIT_MEM` environment variable before invoking the hypervisor, if this is not set it defaults to 64 MB. If it were attempted to load the kernel too high in the address space Uhyve would crash with an error that the size of the guest memory was not large enough. The final decision was to locate the kernel start at 0x1000000 leaving enough room for small applications to be tested and to also fit into the default memory. The option to load the application first and then locate the kernel based on the size of the application was explored. However, structures containing boot info are created at kernel load time and these need to be updated with information about the application when it is loaded so it was not possible to load the application first and then load the kernel.

The hypervisor also allocates the remainder of the memory space above the kernel for the heap. It was felt that it was not necessary for the remainder of the memory to be allocated for the heap. Therefore, the heap space was initially reduced during the project. A subsequent change in the original RustyHermit project refactored the way heap allocation was performed. This change was incorporated into the project code base late in the project. As the reduction in the heap size was not essential for functionality the upstream changes were not rewritten. Future research could look at reducing the heap size.

4.2.2 Load the Application

The hypervisor must load the application into memory. To begin, the hypervisor needs to know where to find the application code. The way this is provided is to pass the path of the application binary to Uhyve as a command line argument when it is invoked. When the hypervisor creates a new VM it needs to do several things. It first checks the relevant ELF headers to make sure it is a compatible executable for the specified architecture and that it does not require any additional libraries. Uhyve identifies the application's loadable ELF segments and loads them into memory at the desired location. The application was loaded at 0x400000 for this project, leaving enough space to the start of the kernel. For dynamic position independent executables (PIE) the application code can be located anywhere in the guest memory. A future extension could be to place the application code at a random address. This would help with security through address space layout randomization (ASLR). This random layout would have to be coordinated with the kernel location to ensure that one does not corrupt the memory of the other.

4.2.3 Pass Application Metadata to the Kernel

Uhyve also reads values from the ELF metadata relating to the application and sets these values in the boot info structure in the kernel. The boot info structure is created when the kernel is loaded and is a struct containing variables shared between the host and the guest which are required by the kernel. The additional values required for the application are used to initialise the stack and jump to the application entry point. The structure was amended to accommodate the required values from the application. This includes the application start point, its size in memory, the entry point i.e. the first instruction in the application, as well as information about the program header tables.

4.3 Second-stage Loader

The second stage of the binary loader is run in the kernel itself. Once the hypervisor has finished initialising and loading the kernel and the application it hands control over to the kernel. The guest kernel first initialises and then it can run the second stage loader. The kernel needs to initialise the stack with certain values which include ELF auxiliary vectors, environment variables, and the command line arguments vectors and count `argv` and `argc`. ELF auxiliary vectors are a mechanism to transfer kernel information to the user application. The values need to be pushed to the stack in the

reverse order that they are to be read during execution. The ELF auxiliary vectors are pushed first, followed by the pointers to the environment variables, then the command line argument vectors `argv`, and finally the command line argument count `argc`.

The initial approach was to create the second stage loader as a Rust application which would be compiled and bundled as part of the unikernel, in the same way that user code would be compiled with the unikernel in a traditional RustyHermit unikernel. This Rust application would define the constant variables to be pushed to the stack, define the value of `argc`, and create vectors of pointers to the environment variables and argument vectors. Rust provides access to the environment variables and the command line arguments through the `std::env` module. It would then use inline assembly code to push the values to the stack and jump to the entry point of the binary application.

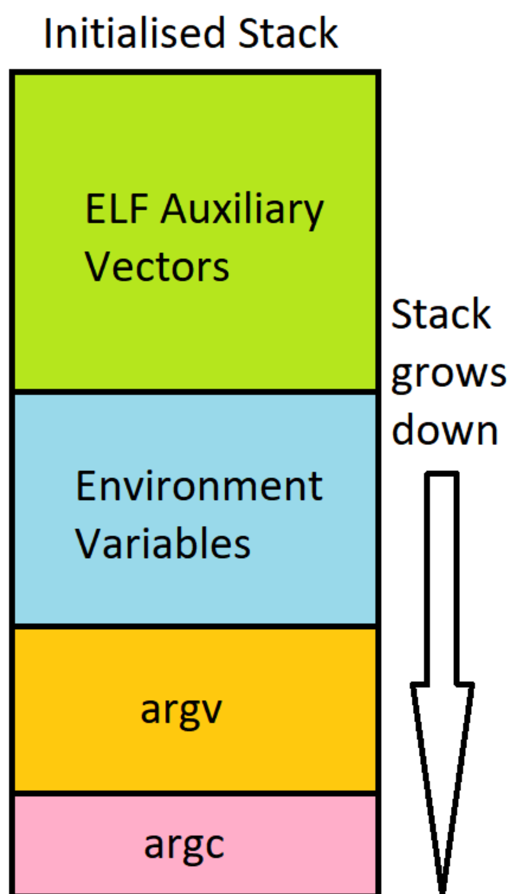

Figure 4: The initialised stack

This approach had to be changed though as some of the system calls that were to be implemented required building with a feature called Newlib. Newlib is a C library

intended for embedded systems [20]. Building with Newlib required support for networking via lwIP. After consulting with one of the key developers of the RustyHermit project they advised that we could build and run in an environment called Hermit Playground[4]. This environment provided support for lwIP and was already configured to build with Newlib. lwIP is a small, lightweight implementation of the TCP/IP stack [1]. For further details on the reasons for building with Newlib and wishing to support lwIP please refer to the report by Laurent Pool.

Unfortunately, Hermit Playground was not configured to run Rust applications with the unikernel, it is configured to run C, C++, Go and Fortran applications that can be compiled against Rust code in a traditional RustyHermit unikernel. Instead of creating a Rust application to load the binary application it was decided to move the initialisation of the stack to the kernel boot process. This provided the advantage that it was easier to access the application values in the boot info using helper functions created in the kernel during the development of the first stage. These helper functions were not accessible outside the kernel so were not available using the traditional Rust unikernel application method.

The disadvantage of this was that the kernel uses a `no_std` Rust environment. This is a way of ensuring that the Rust library used is platform agnostic and is used for kernels and similar bootstrapping code [2]. This meant that there was no access to the `std::env` module which provides iterators over the environment variables and the command line arguments. The iterators were required to create the vectors of values which are subsequently pushed to the stack. No access to `std` also meant that it was not possible to use certain non-primitive types like `CStrings` which allows a representation of strings in a C-like format. These are required as they provide the expected format for the environment variables, command line arguments and auxv on the stack. The `environ`, `argc` and `argv` variables are available in the kernel through functions. These values can then be passed to the application-loading function. Due to the inability to use `CStrings` vectors had to be created manually with C-like strings by adding the characters to the vector as bytes and terminating with a null value. A pointer to the vector containing the string can then be created and passed as required.

[4] https://github.com/hermitcore/hermit-playground

To push the values to the stack in the required order, a vector containing the pointers to environment variables and the command line arguments was created in the order that they would eventually be read. The ELF auxiliary vectors need to be pushed to the stack as a tuple containing a key (or type) and a value. The type is represented by a numerical value which is stored as a constant in the kernel code. A lot of the ELF auxiliary vector values are hardcoded as they are specific to the Linux system or can otherwise be stubbed. Other values are dynamic, for instance the ELF program header table values and the application entry point. The ELF auxiliary vectors are read last so they must be pushed to the stack first in reverse order. The environment variable pointers and command line argument pointers are pushed next, again in reverse order. The final value to be pushed is the command line argument count, `argc`.

A separate function was created to push the ELF auxiliary vectors to the stack, as this required pushing the type-value tuple in reverse order. However, even though this was marked as inline assembly the local variables were becoming corrupted once the stack push began. It was attempted to write the assembly instructions directly into the loader code, but this did not resolve the problem either. The problem was found to be that the stack pointer was obviously being changed as we pushed values to the stack. This meant that the references to the local variables were now pointing to the incorrect location as they were addressed with an offset relative to the stack pointer. The solution was to force frame pointers by setting the environment variable `RUSTFLAGS="-C force-frame-pointers=yes"`. Frame pointers use a separate register to store the beginning of the stack frame for each function. The location of local variables and arguments are addressed relative to the frame pointer and are no longer dependent on the stack pointer. The frame pointer is only updated on a function call or return and therefore is not affected by the stack crafting process. This problem was therefore not related to the use of the separate function so this could be used.

Unfortunately, forcing frame pointers causes the Rust build process to recompile all its dependencies every time it is rebuilt. This became quite a bottleneck when developing. Every change, even if it was small, required several minutes to rebuild and run to test it. This is exacerbated when working with kernel code as even changing a print statement, which is often required for debugging due to the inability to run higher level debugging software, can cause a bug to appear or disappear. Further to this all

development and testing was done on a nested VM, which is slower than a traditional VM, which are themselves slower than native systems.

Once the ELF auxiliary vectors are pushed to the stack successfully the environment variables and command line arguments can be pushed. These are stored as pointers in a single vector. As vectors can return an iterator it is possible iterate over the vector in reverse order and push the pointers to the stack. After this the only remaining value to push to the stack is `argc`. The stack is now initialised, and we can hand execution over to the application. This is simply a case of using an assembly instruction to jump to the application entry point.

The program that was being used to test the success of the binary loading was a Hello World application, written in C and compiled using GCC with `glibc`. The program simply prints “Hello World!” to stdout and execution ends. As was mentioned earlier even seemingly unrelated changes could lead to undefined behaviour. Sometimes some parts of the code would work, and others fail, while a print statement placed after a failing block for debugging could cause it to work, but the previously working block would now fail. This made it very difficult to locate the causes of bugs or find successful solutions. Successful execution of a compiled binary version of Hello World written in assembly was achieved. When loading and executing the `glibc` version of the binary application the unikernel was crashing with the following error:

```
thread '<unnamed>' panicked at 'called `Option::unwrap()` on a
`None` value', /root/.rustup/toolchains/nightly-x86_64-
unknown-linux-
gnu/lib/rustlib/src/rust/library/std/src/sys/hermit/os.rs:143:
28
stack backtrace:
thread panicked while panicking. aborting.
```

The error references code in the Rust std library which is not being called by the function. The stack was printed out for debugging and was determined to be initialised correctly. The error appears to happen after the final jump instruction to the application code. The output after an error provides limited details about why the application crashed but it does provide the exception type and the instruction pointer at the time of the crash.

Using `objdump` it was possible to disassemble the code and examine the instruction which caused the error. In this case the exception was an invalid opcode exception,

and the instruction was a ud2 assembly instruction. The ud2 instruction simply generates an invalid opcode exception and is provided solely to supply this exception for software testing where an invalid opcode is needed [5]. It was not possible to determine why this instruction was being executed as the backtrace was unable to display the call stack. The std library referenced in the error message points to a function to get environment variables but was not part of the code that is called. The environment variables are acquired successfully and passed to the loading function before the jump to the application code.

To eliminate any possible corruption from amending the stack a version of Hello World was tested which was compiled using MuslC LibC instead of GLibC. The advantage of this is that it is simpler and requires fewer elements of the stack to be set up to run. It was tested using a very basic stack initialisation using HermiTux and was shown to work. The same stack initialisation was tried in the project code. Only four of the ELF auxiliary vectors needed to be pushed to the stack and the values for command line arguments and environment variables were stubbed as though there were none. This meant simply pushing null to the stack twice to simulate two empty vectors, and then pushing zero for `argc` to simulate no command line arguments. This would provide the simplest possible stack alteration and avoid using functions to iterate over the vectors in reverse order.

The MuslC-compiled version of Hello World also crashed, with a general-protection exception. Again, `objdump` was used to examine the instruction which had caused the exception and it was found to be a `movaps` instruction using the stack pointer as its source operand. The `movaps` instruction is used to move aligned floating-point numbers. It causes a general-protection exception if the memory operand is not aligned on a boundary [5]. It was still not providing a call stack to trace back the source of the error. The code in the `objdump` referenced functions from a module in the std Rust library related to backtraces `std::backtrace`. To try to prevent code related to this library from running as per the documentation backtracing was turned off by setting the environment variable `RUST_LIB_BACKTRACE` to zero. This did not solve the problem; the same instruction was causing the exception. In the end it was not possible to determine the cause of the problem with the C versions of the binaries before the project deadline. The reasons for the exceptions would be a good exploration for a future work.

Although it was not possible to get a version of RustyHermit working with C binaries it was possible to demonstrate the successful loading and execution of compiled assembly versions of programs. Testing with a Hello World program and a program which manipulates files were both successful. As a proof-of-concept the project was successful in demonstrating that it is indeed possible to use a Rust unikernel to load and execute a binary application passed to it at load time without the need to compile it as part of the unikernel.

Whilst not being able to run C code means that it is not possible to run the NPB benchmark tests and evaluate the performance of the unikernel compared to a standard Linux system, we have created a starting point for future research.

4.4 Challenges

There were many challenges faced on this project. Not least the impact of Covid-19 which has denied access to university equipment and resources and prevented face-to-face contact with my supervisor and co-worker.

4.4.1 Remote Access

The first challenge that we faced was having an accessible machine with KVM enabled to run the hypervisor. The CS Linux appliance VM provided by the university does not allow nested virtualization so it was not possible to run the unikernel on that. As we had no access to university resources, we had to look at alternative approaches. One approach was to use cloud computing resources such as Amazon AWS EC2 instances. After looking into this option, it became clear that the only cloud compute instances which supported KVM were the metal instances. This involves renting the entire physical server and was prohibitively expensive and inefficient in resource use.

An ad hoc approach which was initially used to get started was to provide remote access to a VM on the project supervisor’s computer. This VM supported nested virtualization meaning that it was possible to create another VM inside of it. However, this comes at a performance cost and can cause problems when there is a lot of compilation to do. All development and testing had to be carried out over SSH on the remote machine. This solution worked and was used for the rest of the project. It did mean that we had no access to an IDE but as we were working on systems software this was less of a hindrance than it might have been for a higher-level software engineering project.

4.4.2 Learning a New Language

Most of the code for the project was written in Rust with a small amount in x86 assembly. Neither of which I had any experience in before. Rust uses a different paradigm to other programming languages I had written in before. It uses the concept of ownership to manage memory instead of garbage collection like in other memory safe languages. It also uses a different syntax to other languages I am familiar with which are generally all very similar to C. There was very little time to learn the fundamentals of Rust from scratch, so this had to be learned as I went along.

4.4.3 Dependencies

Another challenge that we faced was the dependencies on other repositories. At the start of the project RustyHermit was broken due to a change in Rust libstd. The code changes had been made in the RustyHermit repository but were awaiting a code review and acceptance into the main Rust repository. This meant we were not able to start experimenting with RustyHermit for a couple of weeks at the beginning of the project.

In February 2021, a change to a dependency of Uhyve prevented it from installing. A significant amount of time was spent trying to locate the source of the error. It was not possible to simply lock the dependency to a working version in Uhyve due to a further dependency chain. An issue was raised by us on the Uhyve GitHub repository whose contributors then in turn raised an issue with the maintainers of the dependent repository for a change to lock the upstream dependency. A related issue occurred shortly after with another dependency but as the fix to lock the version was previously implemented it just required a code change at our end.

This provided an opportunity to learn how to raise issues with an external project and communicate technically with repository maintainers.

4.4.4 System Bugs

Working with system software can produce unintended effects which are hard to determine why they happen. Changes to seemingly unrelated code can cause parts of the program to start or stop working unexpectedly. In part this seems to be related to the fact that some of the implementation of Rust code is not well defined in the way it needed to be used in the project. The Rust documentation is very good and makes

it clear when this may be the case. It is also partly related to performing changes to memory directly using assembly code.

5 Functional Evaluation

As explained in the previous chapter it was not possible to run NPB benchmarks against the unikernel to compare its performance to a standard Linux kernel or to other unikernels. This is due to the inability to support C code. In terms of functionality the project was successful in its aim of running pre-compiled binary applications written in assembly code.

The implementation of system calls, the loading of the application into memory and the ability to transfer control from the hypervisor to the kernel and finally to the application were all demonstrated to work. Given further time and resources it should be possible to find the cause of the bug which prevents C code from executing properly. Much of the groundwork has already been laid to make this work.

The project can successfully demonstrate the running of a Hello World application (see Appendix A). Printing a pre-programmed message to stdout. Operations on files has also been demonstrated using a simple application which can read the contents of a text file and print it to stdout whilst changing the last word in the file. Many more system calls have been implemented which binary applications can use.

6 Comparison to Similar Work

A similar project which looked at binary compatibility in a unikernel, albeit in a legacy language, is HermiTux. Although the HermiTux project was more sophisticated, using binary rewriting techniques and analysis of dynamic code it was often used as a comparison and a source of study for the project. In fact, the course of this project started off by studying both the HermiTux and RustyHermit projects. An understanding of how HermiTux loads the application and sets up the stack was crucial to understanding the steps required to run a binary application in a unikernel. Several unfamiliar, low level concepts of system software were illustrated through HermiTux, which were subsequently applied in the project. These include the ELF format, position independent code, the initialisation of kernels and applications in memory, the difference between kernel and user space, how the stack is arranged for execution and the workings of hypervisors.

The other project which required study was RustyHermit as this was the basis of the project. Being unfamiliar with Rust code, its package manager Crate and its build environments presented a steep learning curve. Using RustyHermit as a starting point, running the provided examples then extending and modifying them to run our own programs before learning the code base and making modifications was immensely useful. A lot of the development in Rust was quite advanced and this made it difficult for someone new to the language. The fundamentals of the language had to be learned on the fly and it often required using the language in esoteric and uncommon ways. Having existing code which could be studied and used as a guide was useful in a lot of situations, but a large amount of creativity and further research was still required.

Although the project would have been easier in C, which is not only more familiar but provides much more unchecked access to low level functions, the advantages of writing in a memory safe language are clear. For an inexperienced developer writing system software in C, especially when handling memory, there are numerous pitfalls. Many of which would not be caught at compile-time, some of which may only occur occasionally at run-time and others which may never affect the functionality but could leave the software vulnerable to exploits and data leaks. Rust prevents many of these issues by checking and enforcing rules at compile-time. Given the same functionality as HermiTux the additional challenge of writing in Rust would be worth the extra effort.

7 Conclusion & Future Works

7.1 Future Works

There are numerous opportunities to expand this project and take this research further. The first would be to investigate in more detail the bug which prevented the running of C code. If this could be identified, then the ability to run C code would not only easily expand the type of applications that could be run but also allow the running of NPB benchmarks and allow the assessment of the unikernel performance.

Certain system calls require a feature called Newlib as well as lwIP. Providing support for lwIP would enable networking which would be a desirable feature. Support for lwIP and Newlib is available in Hermit Playground. Currently Hermit Playground does not support Rust applications and requires the C version of Uhyve, not the Rust version

which we amended. It should be possible to change Hermit Playground to support these which would give access to more system calls and networking.

Another area for future research would be the ability to support even more applications. Develop more system calls, provide support for dynamic applications instead of just static binaries. It would also be possible to support more languages including interpreted languages such as Python.

Security of unikernels is important if they are to be used in production. There are several different areas that could be investigated in future work. Some of these problems will be more challenging than others. A simple improvement would be implementing Address Space Layout Randomization (ASLR). Both the kernel and application can be easily relocated in the available address space. Care would have to be taken to keep them well separated as there is no separate kernel and user space.

A more challenging security improvement related to the above would be preventing an adversary running code which could access protected memory space. All code in the unikernel runs in privileged mode, a mechanism is required to prevent exploitation of this fact. Recent attempts at intra-unikernel protection technologies include Intel Memory Protection Keys [16, 18].

7.2 Conclusion

At the start of the project, I knew very little about many of the concepts we would be working with. I knew hardly anything about unikernels, I knew only the basics about hypervisors and VMs, I had no experience with Rust and had no knowledge of ELF. I had also not done any significant development work using an existing open code base. There was a lot to learn in a limited amount of time and it was at times challenging.

The nature of the project was highly exploratory and was successful in achieving a proof of concept demonstrating the ability to build a binary compatible unikernel written in Rust. Even though the project may not have achieved all its aims of being able to run a binary application written in C and hence acquire benchmarks against the unikernel we have provided a starting point for further research and completed much of the groundwork towards this goal.

The applications of unikernels are still relatively immature and this is an active area of research as an alternative to containers in the rapidly expanding cloud computing

sector. I am thankful to have been able to contribute to this research and will monitor the progress of unikernels and, in particular, those based on HermitCore.

Appendix A hello_world_asm.s

```
        .global _start

        .text
_start:
        # write(1, message, 14)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout
        mov     $message, %rsi          # address of string to output
        mov     $14, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write

        # exit(0)
        mov     $60, %rax               # system call 60 is exit
        xor     %rdi, %rdi              # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii  "Hello, World!\n"
```

References

[1] Adam Dunkels and Leon Woestenberg. 2018. *lwIP: Overview*. Retrieved April 22, 2021 from https://www.nongnu.org/lwip/2_1_x/index.html

[2] Embedded devices Working Group. 2019. *The embedded Rust book*. Retrieved April 19, 2021 from https://docs.rust-embedded.org/book/intro/no-std.html

[3] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis and Haining Wang. 2017. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *Proceedings of 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Denver, CO, USA. 237-248, DOI: https://doi.org/10.1109/DSN.2017.49

[4] Diane Hosfelt. 2019. *Implications of Rewriting a Browser Component in Rust*. Retrieved April 18, 2021 from https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/

[5] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z.* (September 2016). Retrieved April, 22 2021 from https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

[6] Shashank M. Jain. 2020. Virtualization Basics. In *Linux Containers and Virtualization*. Apress, Berkeley, CA. DOI: https://doi.org/10.1007/978-1-4842-6283-2_1

[7] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press, USA. Retrieved April 18, 2021 from https://doc.rust-lang.org/book/

[8] Stefan Lankes, Jens Breitbart, and Simon Pickartz. 2019. Exploring Rust for Unikernel Development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems* (*PLOS'19*). Association for Computing Machinery, New York, NY, USA, 8–15. DOI: https://doi.org/10.1145/3365137.3365395

[9] Stefan Lankes, Simon Pickartz, and Jens Breibart. 2019. HermitCore. In *Gerofi B., Ishikawa Y., Riesen R., Wisniewski R.W. (eds) Operating Systems for Supercomputers and High Performance Computing*. High-Performance Computing

Series, vol 1. Springer, Singapore. DOI: https://doi.org/10.1007/978-981-13-6624-6_20

[10] Anil Madhavapeddy and David J. Scott. 2014. Unikernels: the rise of the virtual library operating system. *Commun. ACM* 57, 1 (January 2014), 61–69. DOI: https://doi.org/10.1145/2541883.2541895

[11] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of SOSP '17: ACM SIGOPS 26th Symposium on Operating Systems Principles, Shanghai, China, October 28, 2017 (SOSP '17)*, 16 pages. DOI: https://doi.org/10.1145/3132747.3132763

[12] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2013. System V Application Binary Interface. AMD64 Architecture Processor Supplement, Draft v0 99 (2013).

[13] Spencer Michaels, and Jeff Dileo. 2019. *Assessing unikernel security.* Technical report. NCC group.

[14] Microsoft Security Response Center. 2019. *Why Rust for safe systems programming.* Retrieved April 18, 2021 from https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming/

[15] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19), April 13–14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. DOI: https://doi.org/10.1145/3313808.3313817

[16] Pierre Olivier, Antonio Barbalace, and Binoy Ravindran. 2020. The Case for Intra-Unikernel Isolation. In *Proceedings of the 10th Workshop on Systems for Post-Moore Architectures (SPMA).*

[17] James E. Smith and Ravi Nair. 2005. The architecture of virtual machines. In *Computer* 38, 5 (May 2005), 32-38. DOI: 10.1109/MC.2005.173.

[18] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE*

'20), March 17, 2020, Lausanne, Switzerland. ACM, New York, NY, USA, 14 pages. DOI: https://doi.org/10.1145/3381052.3381326

[19] Unikernel. 2018. *Projects*. Retrieved April 22, 2021 from http://unikernel.org/projects/

[20] Corinna Vinschen and Jeff Johnston. 2020. *The Newlib Homepage*. Retrieved April 22, 2021 from https://sourceware.org/newlib/