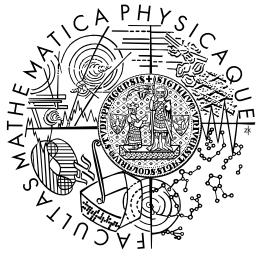

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Matěj Volf

Rust for HelenOS

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Vojtěch Horký, Ph.D.
Study programme: Computer Science

Prague 2025

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

To all those who patiently listened while I explained my frustration with an issue, only to hear me stop halfway through and say, “Oh never mind, I see the solution now!”—thank you. Your contributions to this project are invaluable.

Title: Rust for HelenOS

Author: Matěj Volf

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Vojtěch Horký, Ph.D., Department of Distributed and Dependable Systems

Abstract: Rust is a modern programming language that compiles to machine code and offers unique features such as data ownership and borrow checking. A comprehensive standard library and well integrated cross-compilation capabilities provide a significant advantage for developing applications that support many target platforms. The objective of this thesis is to introduce support for the HelenOS operating system into the Rust compiler and standard library. We discuss the steps of this process and examine the technical challenges that arise, including issues that need to be resolved in the underlying operating system implementation. A significant portion of the standard library is implemented, and its correctness is validated by running a custom test suite on all five HelenOS variants that we support. Practical applicability of this work is then demonstrated by running multiple real-world command-line and graphical applications on HelenOS.

Keywords: HelenOS, Rust, porting, compiler, runtime environment

Název práce: Rust v HelenOSu

Autor: Matěj Volf

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Vojtěch Horký, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Rust je moderní programovací jazyk, který se překládá do strojového kódu a nabízí unikátní vlastnosti, jako je vlastnictví dat a kontrola zápůjček. Komplexní standardní knihovna a dobře integrovaná schopnost křížového překladu poskytují významnou výhodu pro vývoj aplikací, které podporují mnoho cílových platforem. Cílem této práce je přidat podporu operačního systému HelenOS do překladače a standardní knihovny jazyka Rust. Pojednáváme o krocích tohoto procesu a zkoumáme nastalé technické výzvy, včetně problémů, které je třeba vyřešit v implementaci samotného operačního systému. Značná část standardní knihovny je implementována a její správnost je ověřena spuštěním naší vlastní sady testů na všech pěti podporovaných variantách HelenOSu. Praktická využitelnost této práce je následně předvedena spuštěním několika reálných aplikací pro příkazovou řádku i grafických aplikací na systému HelenOS.

Klíčová slova: HelenOS, Rust, portování, překladač, běhové prostředí

Contents

Introduction and motivation

An operating system is not too useful on its own, its utility heavily grows with the amount of programs that are able to run on it. HelenOS is a research operating system that can currently run almost exclusively[1] C programs that were specifically written for it. It also has support for C++ and Python, but with significant limitations, so these languages are not widely used for writing HelenOS programs.

Rust is a relatively new programming language with a rich standard library and modern tooling. It is considered to be the first language to bring the concepts of data ownership and borrow checking to the programming mainstream. Although Rust compiles directly to machine code and can achieve the same performance as C, it offers guarantees regarding memory safety and the absence of undefined behavior, unless the programmer uses blocks of `unsafe` code. There is also a large ecosystem of pure-Rust implementations of various libraries, and mature support for cross-compilation is available.

All these features make Rust a great candidate to port onto HelenOS, since support for the language itself could enable easy porting of many existing applications. Thus, the goal of this thesis is to add HelenOS support to the Rust compiler and standard library. It is not expected that we will implement support for every feature that Rust standard library offers, instead, our work should provide a solid baseline, and then focus on porting a set of applications that would provide interesting functionality for HelenOS.

Structure of this thesis

This thesis is divided into four main chapters. Chapter 1 provides necessary context and technical background about the systems we are working with. Chapter 2 analyzes the process necessary for achieving our set goal of porting Rust and discusses our selection of programs that we attempt to port. Chapter 3 describes the resulting implementation of the language support – this includes additions to the Rust compiler and standard library, as well as necessary changes in HelenOS. Finally, Chapter 4 documents the steps necessary to setup and use our toolchain, and showcases our success in running our selected programs on HelenOS.

[1] The HelenOS project includes a repository called `harbours`, located at `https://github.com/HelenOS/harbours`. This project contains a collection of ports of C and C++ programs to HelenOS. However, for each ported library or application, a configuration file must be created with the location of the source code, a script to build the program, and another script to install it in the correct location. This approach does not scale very well, and thus the amount of software ported to HelenOS is still very limited. This led to our interest in possible significant expansion of the software ecosystem of HelenOS by adding support for Rust.

1 Technical background

This thesis has required building up a deep understanding of two complex systems: the Rust compiler and standard library, and the HelenOS operating system. Our work lies at the border between these two systems, where we need to deal with tightly coupled and often times unobvious interactions between the involved technologies.

While we assume that the reader is knowledgeable about the general area of computer science and understands basic operating system terminology, we notice that our work involves internals two projects that may not be familiar to many. This chapter aims to provide sufficient explanations of all the relevant components involved in our work, as well as some terminology specific to HelenOS and Rust.

Reading this chapter in its entirety rightaway is not essential, we recommend the reader to use it as a reference that they can come back to when reading the main body of the thesis.

1.1 HelenOS

HelenOS is a multiserver microkernel operating system. This means that the system consists of a minimalistic kernel in combination with multiple servers running in userspace. These servers provide all system functionality which can run in non-privileged mode, i.e. services such as networking, filesystem, or device drivers. All components of the system communicate with each other through asynchronous message passing.

HelenOS draws significant inspiration from the Unix and POSIX standards, but does not accept these APIs as dogmatic. HelenOS avoids implementing legacy APIs which have modern alternatives,[1] and in some areas explores completely novel implementation approaches, uncommon on other platforms. For example, it avoids using network sockets and instead implements a custom API interface for reading and writing to TCP streams.[2]

Since we will be working on a compiler to native executable binaries, it should be mentioned that HelenOS executables are in the standard Executable and Linkable Format (ELF) format. Dynamic linking, position-independent code and thread-local storage are all supported. A loader server is responsible for setting up all of this before the startup of each process (called a *task* in HelenOS).

1.2 Rust compiler

Both the Rust compiler and standard library are highly modular. The Rust compiler, `rustc`, uses LLVM to generate assembly code, although alternative experimental code-generation backends (GCC and cranelift) have been added recently.

Rust allows installation of multiple *toolchains* on one system. A *toolchain* specifies the version of Rust to use and the target platform for which programs should be compiled.[1] Rust uses the informal convention of target triples in the form

[1] Please note that this term is a bit overloaded, and, contextually, we might use the name

`<architecture>-<vendor>-<operating system>`.[3] Each target triple is fully described by a target specification, which we will discuss in more detail shortly. The product of this thesis will be a set of new Rust targets and associated toolchains, one for each CPU architecture of HelenOS.

One version of `rustc`, the compiler itself, can be used for multiple toolchains, since configurations of all available targets are built into it. However, the rest of a toolchain must be downloaded separately for each platform – this most imporantly means a build of the standard library.

Build of any non-trivial Rust program is orchestrated by the Cargo package manager. Cargo selects the correct toolchain based on command-line flags (for example `cargo +nightly --target x86_64-unknown-helenos`) and uses `rustc` to compile each crate – a Rust compilation/linkage unit, i.e. either a library, or the main program with a `main` function – into an object file. Then it invokes a linker to produce the final binary. The linker may be of a completely different suite than the codegen backend, which is very important for us – HelenOS only provides a GCC-based compiler suite, and we will need to use the GCC linker configured for HelenOS to produce a HelenOS-compatible binary.

1.2.1 Rust target specifications

As is mentioned above, a single build of `rustc` can be used to compile programs for multiple platforms. This is thanks to a database of target specifications describing each target triple that Rust natively supports. These specifications are stored in the directory `compiler/rustc_target/src/spec/targets` in the Rust source tree, provided alongside this thesis as Attachment A.1. Each target definition describes the features of the CPU architecture as well as the operating system. This includes things like endianness, supported size of atomic operands, availability of dynamic linking, or the name of the linker to use.

The target specifications are stored in the compiler as Rust source files, to enable programmatic reuse of parts of configurations shared among the individual targets of each operating system. However, an equivalent JSON representation is specified, and Rust can export its specifications in this format, as well as load from JSON completely custom configurations and then use them to compile programs. An example of the complete configuration of the `i686-unknown-haiku` target is shown in Code fragment 1.1.

Loading of custom `target.json` specifications is essential for development for platforms not natively supported by Rust for various reasons. One such example is the toolchain for building Rust programs for the AVR family of microcontrollers (that are used – among other things – in Arduino boards).[4] It will be also helpful for our work, since it will allow us to delay the step of compiling `rustc` to test first basic Rust programs on HelenOS.

"toolchain" for the whole suite of Rust build tools, as well as for the HelenOS GCC-based cross-compilation *toolchain*.

Code fragment 1.1 Obtaining `target.json` for Haiku

```
$ rustc +nightly -Z unstable-options --print target-spec-json \
    --target i686-unknown-haiku
{
    "arch": "x86",
    "cpu": "pentium4",
    "crt-objects-fallback": "false",
    "data-layout": "e-m:e-p:32:32-p270:32:32-p271:32:32-p272:64:64-i128:
128-f64:32:64-f80:32-n8:16:32-S128",
    "dynamic-linking": true,
    "linker-flavor": "gnu-cc",
    "llvm-target": "i686-unknown-haiku",
    "max-atomic-width": 64,
    "metadata": {
        "description": "32-bit Haiku",
        "host_tools": true,
        "std": true,
        "tier": 3
    },
    "os": "haiku",
    "pre-link-args": {
    "gnu-cc": [
        "-m32"
    ],
    "gnu-lld-cc": [
        "-m32"
    ]
    },
    "relro-level": "full",
    "stack-probes": {
    "kind": "inline"
    },
    "target-family": [
    "unix"
    ],
    "target-pointer-width": "32"
}
```

1.3 Rust standard library

Rust has a quite extensive standard library. Apart from a large number of commonly used primitives, functions, and data structures, it also provides abstractions that wrap the platform-specific `libc` interfaces, allowing programs to use features such as filesystem or networking without having knowledge of the underlying platform implementation. This is a part of the reason why Rust programs can be highly portable, but it also means that there is a large API surface that will be necessary to implement for HelenOS.

The Rust standard library is divided into multiple crates with dependencies between them, just like standard Rust projects. The `core` crate contains the primitives that need to be available on any platform, including bare-metal environments without an operating system – this includes basic data types that do not require internal memory allocations, functions for pointer manipulation, fixed-size arrays, or the `Iterator` interface.

The `alloc` crate requires the program to specify a memory allocator, and provides common data structures that make use of it.

Finally, what is most commonly referred to as *Rust standard library* is the `std` crate. This package re-exports everything from `core` and `alloc`, and additionally provides any APIs that depend on operating system features. This will necessitate most of the work on supporting HelenOS.

There are further support crates for various Rust language features which will not be of much interest to this thesis, since they do not contain any OS-specific code. The only remaining part worth explaining is `panic_unwind`/`panic_abort`: these crates provide support for Rust's *panic* mechanism, akin to exceptions in C++. Rust panics also unwind the stack and run destructors on each stack frame, however, unlike C++ exceptions, they are not considered a standard control flow mechanism to be used in common situations. Instead, usage of the `Result` enum type is preferred for error handling.[5]

Stack unwinding requires underlying support from the operating system C libraries. For systems where this is not available, the `panic_abort` crate is used instead. With this crate, causing a panic will immediately print the description and location of the panic, and abort the program. For HelenOS, this is the behavior we will use, since stack unwinding is not yet implemented there.

1.4 CPU architectures

It is assumed that the reader is familiar with the concept of CPU architectures and instruction sets. However, we would like to note that there is an unexpected complexity therein.

Firstly, different systems might use different names for one platform. As an example, the *64-bit extension of the x86 instruction set architecture* is commonly referred to using at least three different names: `x86_64` (e.g. Rust), `amd64` (e.g. HelenOS) or `x64` (e.g. Windows). Same holds true for the 32-bit version, where `i386` and `i686` are often used interchangeably despite meaning slightly different revisions. The corresponding HelenOS toolchain is named `ia32`, which is a technically accurate label as well.

Secondly, almost every "architecture" is in fact a familiy of architectures with multiple versions, as features were being added. A good example of this is 32-bit ARM. On HelenOS, this target is called simply `arm32`, although there exists a large number of variations. Versions `armv6` to `armv8` are all currently used in multiple Rust targets. There is also the 16-bit *Thumb* instruction set designed for higher code density.[6] This instruction set is supported as a mode of operation on some ARM-architecture CPUs, and shares a lot of features with the standard ARM instruction set. And finally, hard- and soft-float ABI needs to be distinguished based on the presence of floating-point registers.[7]

All these features make the ARM architecture very versatile, enabling its usage for use-cases ranging from low-power embedded devices to personal computers and high-performance data centers. However, it simultaneously imposes additional complexity on compiler developers, requiring from them extra caution when determining the precise set of features to use for a given compilation target.

1.5 Related work

Rust has been ported to many platforms, including other research operating systems with size of user base similar to HelenOS. Yet, there is little academic work written about the process.

Authors of the Xous operating system have published a blog post about their experience porting a subset of Rust standard library to Xous, but the article focuses mostly on the practicalities of distributing the toolchain outside of the official Rust distribution.[8]

Another team of 10 authors describes porting Rust to Intel SGX, a hardware-supported trusted execution environment. This work is also very specific due to the strict security requirements, and therefore most of the paper is oriented at the methods of formal verification applied in the process.[9]

In conclusion, the author's main guiding resource for this work was solely official Rust documentation.[10] This guide sufficiently outlines the high-level steps, however, it assumes high familiarity with the internals of the Rust compiler. Eventually, the author has built a good enough understanding of the toolchain by reading pieces of the source code, as well as sometimes by trial and error. This thesis makes an attempt to slightly improve this gap and expose some of this siloed knowledge in a written form.

On the side of expanding capabilities of HelenOS, we can find multiple theses of Charles University students that have driven HelenOS forward in various areas. The aforementioned C++ runtime is a result of a Master thesis written by Jaroslav Jindrák.[11] However, the feature potentially most interesting to us, stack unwinding and exception handling, was determined to be too large of an undertaking and the implementation has not been done in this work. Therefore, we will not be able to use the unwinding behavior for Rust panics either.

2 Analysis

This chapter describes our analysis of the process of porting Rust to a new platform, the logical order of all the necessary steps, as well as our decisions regarding the applications we will be using to test our resulting compilation toolchain.

2.1 Process of enabling initial support

There is a large number of variables that affect whether an executable file will run on a given platform. Successfully running a program requires a correct description of the target platform in the compiler (see Chapter 1.2.1), finding the shared and static libraries to link, proper handling of ELF relocations by the operating system's dynamic linker, correct implementation of thread-local storage and many other things. We will come back in more detail to each of these areas in Chapter 3, which examines the specific issues that needed to be resolved.

With this amount of complexity in mind, our process should be focused on implementing and verifying the correctness of each of these features separately from the rest of the system. An approach that will suit our situation well and avoids exposing ourselves to all the potential issues at once, is the incremental build model. This software development method, as defined by Pressman,[12] describes starting with a minimal prototype, and then iteratively expanding its capabilities. Each step taken should be as small as possible, in order to make it easier to isolate the cause of any arising problems.

In our concrete case, the prototype comprises a minimal Rust program without any usage of the Rust standard library runtime, compiled using an external `target.json` specification on an unmodified version of `rustc`, and statically linked directly with HelenOS `libc.a`.

An example of a suitable program is shown in Code fragment 2.1. As this file opts out of using the `std` crate, any interaction with the outside world needs to be done by calling C functions and usage of raw pointers. An attempt at running this program will allow us to evaluate if support of Rust on HelenOS is at all feasible, and we will be able to use it as a starting point for further development.

Code fragment 2.1 A Minimal Rust program for HelenOS

```
#![feature(core_intrinsics)]
#![no_std]
#![no_main]
use core::{ffi, intrinsics::abort};

#[panic_handler]
fn panic(_info: &core::panic::PanicInfo) -> ! {
    abort()
}

#[link(name = "c")]
#[link(name = "startfiles")]
extern "C" {
    pub fn puts(c: *const ffi::c_char);
}

#[no_mangle]
fn main(
  argc: ffi::c_int,
  argv: *const *const ffi::c_char,
) -> ffi::c_int {
    unsafe {
        puts(b"Hello, world!\n\0".as_ptr() as *const ffi::c_char);
    }
    0
}
```

Once this program executes correctly, we can enable dynamic linking. Independently from that, we can also attempt to point the Rust memory allocator to `libc::malloc` and make use of dynamic structures from the `alloc` crate. When we determine all these things to work reliably, we shall start gradually integrating our target into the Rustc source tree, as native compiler targets are a prerequisite for building the `std` crate.

2.2 Linking with HelenOS libraries

As briefly noted in our introduction, by default, HelenOS only distributes C programs purpose-built for this OS. To build all these programs, it uses a build system based on Meson. This build system manages the include paths and linkage of correct libraries, but integrating external programs that have their own build systems, yet alone a build system as complex as Rust's bootstrap process, would not be practical.

As part of our implementation, we should attempt to enable installation of HelenOS libraries to a standard location where the linker can automatically find them, as is the case with libraries on Linux.

2.3 Rust standard library

Although Rust compiles to machine code and does not have a full runtime in the sense of Java, C#, or Go, there is still some support code that needs to run before the main function of a standard Rust program,[1] just like C has `__libc_main`. This runtime for Rust is located in `library/std/src/rt.rs`. Among other things, this code calls into `std::thread`, which makes use of thread-local storage (TLS) to store parts of state specific to each thread. TLS is also used in the `std::io::stdin` module. This means that we will need a very solid TLS implementation to run any Rust program with `std` at all.

After implementing the initial support as outlined above, our work can move on to supporting library features that interact with the operating system. To facilitate this, the standard library contains a *platform abstraction layer* for each supported operating system. In this directory are consolidated implementations of internal interfaces, used by the public high-level APIs of the standard library. Each module in `library/std/src/sys` contains a large block of conditional compilation directives in the following form:

```
cfg_if::cfg_if! {
    if #[cfg(target_family = "unix")] {
        mod unix;
        use unix as imp;
    } else if #[cfg(target_os = "windows")] {
        mod windows;
        use windows as imp;
    // [...]
    } else {
        mod unsupported;
        use unsupported as imp;
    }
}
```

For most modules, the fallback `unsupported` module is provided. This mock implementation causes the features to compile correctly, but fail at runtime when called. However, there are also a few modules that are required to be implemented for all target platforms and do not have any such fallback. These modules are:

- `sys/alloc` – memory allocation,
- `sys/pal/<os>/sync` – implementation of low-level locks and condition variables,
- `sys/random` – random number generation for the purpose of seeding `HashMap` for security,
- `sys/thread_local` – thread-local storage.

We will therefore need to implement all of these APIs, at minimum.

For memory allocations, HelenOS `libc` provides the standard `malloc` and `free` functions. Implementation of the Rust `Allocator` trait using these functions is

[1] Please do not confuse with the C `main` function. Rust standard library exposes a C `main`, performs initialization, and then calls the Rust `main`. The difference between these two symbols is that the Rust `main` name is mangled according to the Rust ABI. This is the reason why our code in Code fragment 2.1 needs to use the `#[no_mangle]` attribute.

already present in the Rust standard library, so it should be sufficient just activate this implementation for HelenOS.

The locking and condition variable interface fully matches the `fibril_mutex` and `fibril_condvar` APIs of HelenOS `libc`, so our implementation can simply defer to these C functions.[2] The `sys/random` module is also trivial, as it only needs to call the `rand` function.

The thread-local storage API in Rust is quite complex, as it needs to provide an abstraction that is possible to be implemented using diverse platform primitives. One of the existing implementations is based on POSIX `pthread_key_*` API, which is prepared in HelenOS, albeit not yet implemented. Nonetheless, we determine this to be the best method of enabling HelenOS support, as it offers a convenient reduction in complexity and avoid the need of studying the Rust abstraction, and we will only need to implement the standard POSIX API in HelenOS.

2.3.1 Selecting optional APIs

For the rest of the APIs in `std/sys`, we are free to choose which ones we want to implement, and we can base these choices on the applications we want to be able to run, as well as the maturity of the underlying HelenOS APIs.

Using admittedly a fairly unscientific approach, we decide to implement the `args`, `stdio`, `fs` and `thread` APIs. We hypothese that this set of APIs is commonly used in many CLI applications, so missing any of these APIs would make most applications unusable. At the same time, these APIs have solid underlying support in HelenOS and could be sufficient to support a large number of interesting CLI tools – such tools typically read the command line arguments, load the input data from disk, perform some computation and write the result back to disk.

2.3.2 Testing the standard library

Rust naturally has a large suite of tests for the standard library. However, these tests are integrated into the Rust build system, and there is no easy way of building the tests into a simple executable to then run on HelenOS. As an alternative solution, Rust provides a *remote test server* – this is a program that can be run on the target system that does not support running fully Rust tooling natively. The development machine with the Rust compiler then connects to this server and uses it to run the tests remotely.

We have looked into this option of running the tests, however, it would require solid implementation of the networking API. This we have opted not to do, since we expect it to involve non-trivial amount of extra work in the underlying HelenOS implementation. To the best of author's knowledge, no HelenOS application uses a single TCP stream from multiple fibrils, so it is unclear if the current implementation is thread-safe. That would be a necessity for the Rust standard library – the traits `Read` and `Write` are implemented for `&TcpStream`, i.e. reads and

[2]Although HelenOS supports threads as known from standard operating systems, usage of *fibrils* is preferred. A *fibril* is a lightweight thread that is implemented in HelenOS `libc` and is not visible to the kernel. A notable difference is that fibrils cannot be preempted to switch to another fibril of the same kernel thread, but for all practical purposes, the differences between fibrils and threads are irrelevant to our work.

writes can be made through a shared reference, which can be used from multiple threads simultaneously.

Instead of running the official test suite, we will therefore need to create a small custom test suite that will have the form of a standard Rust program with the `main` function. This test suite should cover the public APIs that use the internal platform-specific implementations that we will write for HelenOS.

2.4 Demo applications

To demonstrate the usefulness of our work, we want to choose a set of Rust programs that we will port to HelenOS. These applications should take advantage of the parts of the standard library that we will have implemented, and at the same time avoid having too many other platform-specific dependencies.

We have determined image processing utilities to be a good demonstration of the advantages of Rust that fits these constraints. An image processing library typically has a large number of dependencies that provide support for different image formats. For a traditional C program, each such library would require separate configuration with the C cross-compilation toolchain, making the porting process very tedious. However, in Rust, the build of all the crates should be fully automatic, and we expect the image codecs to be pure algorithms written in Rust, thus with no system-specific dependencies.

We have identified two command-line applications that fit this category: `resvg`,[3] a library for rendering vector files in the SVG format to PNG images, and `imagecli`,[4] a command-line tool for performing various transformations and format conversions on images, similarly to the famous[5] ImageMagick suite.

These applications will add the image processing capabilities to HelenOS. However, we also notice that there would be no way of displaying the resulting images. Although HelenOS has a native image viewer, it only supports the uncommon TGA format. Therefore, we shall also write a simple image viewer application that will take advantage of the available ecosystem of libraries to support more common image formats. This application will also demonstrate the capability of Rust to interface with native HelenOS libraries other than `libc`.

2.4.1 Choosing a GUI abstraction

Our image viewer application should aim for separation of the application logic from the underlying raw UI implementation of the platform. After a research of the common practices of writing GUIs in Rust, we have understood that the traditional way of programming graphical user interfaces (GUIs), focused on persistent widgets and interaction callbacks, is difficult to express in performant and safe Rust abstractions. Instead, a paradigm called *immediate mode* has gained significant popularity in the Rust ecosystem.[13][14]

Therefore, instead of trying to implement Rust abstractions on top of HelenOS native suite of GUI widgets, we will base our GUI framework on of the Iced library. As we will describe, this library should be the easiest to integrate into HelenOS,

[3] `https://github.com/linebender/resvg`

[4] `https://github.com/theotherphil/imagecli`

[5] See `https://xkcd.com/2347/`

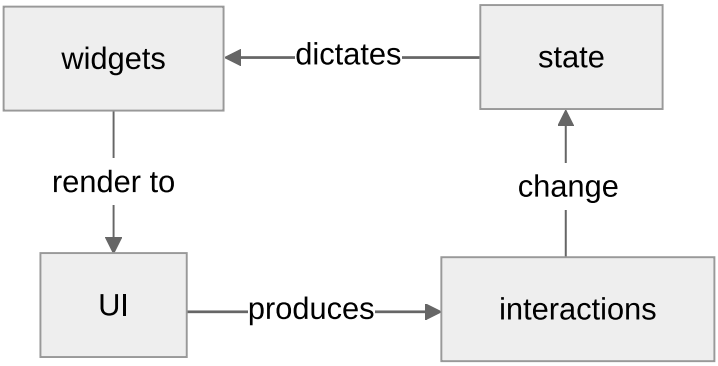

Figure 2.1 Operation of Iced applications

and will allow us to easily run the same application code on HelenOS as on other platforms.

This library uses an architectures inspired by Elm, descibed in the documentation as following:[15]

> Our quick dissection has successfully identified three foundational ideas in a user interface:
>
> - **Widgets** — the distinct visual elements of an interface.
> - **Interactions** — the actions that may be triggered by some widgets.
> - **State** — the underlying condition or information of an interface.
>
> These ideas are connected to each other, forming another feedback loop!
>
> Widgets produce interactions when a user interacts with them. These interactions then change the state of the interface. The changed state propagates and dictates the new widgets that must be displayed. These new widgets may then produce new interactions, which can change the state again… And so on.

The feedback loop described in the citation is implemented in Iced a fully platform-independent way. A component that is missing in this abstract description is the platform runtime that renders the widgets onto the screen and sends the detected interactions interaction events back to the framework. The full diagram of the Iced architecture is shown in Figure 2.1. The bottom left section of it is the part that we will need to implement specifically for HelenOS.

GUI libraries also typically use a GPU-based renderer using standardized APIs such as OpenGL to draw the widgets onto the screen. Integrating any such API into HelenOS would be enormously difficult, however, Iced luckily also provides a renderer based on `tiny-skia`, a pure-Rust 2D graphics library that runs without any GPU support.

All of these features should allow us to integrate Iced into HelenOS with relatively little effort, since we will only need to implement drawing of a bitmap into the application window, and translation of HelenOS events (mouse movement, key presses) into Iced events.

3 Implementation of required HelenOS and Rust changes

This chapter describes the notable parts of our implementation of the HelenOS-Rust toolchain, done as outlined in the previous chapter. The first part mainly describes the issues that needed to be resolved to enable correct execution of Rust programs on HelenOS, latter sections cover our code added to the standard library.

3.1 Thread-local storage handling

When a new task is started in HelenOS, the kernel creates the address space for the task, and the `loader` program is loaded into this address space. A connection to this loader instance is then passed to the caller, who needs to communicate with the loader to pass it the filesystem location of the ELF file to be executed. The task of the loader is then to populate the address spaces with the contents requested by the ELF, and finally jump to the entry point of the program.

Among the features that the ELF headers describe is the thread-local storage (TLS). This section describes the size and alignment of the thread-local storage, which has to be allocated for each thread before it starts executing. This means that there are multiple places that can call the TLS initialization code:

- The loader, to prepare data the main thread of the program being loaded.
- Thread initialization code in `libc`, to prepare data for the new thread. This also applies to HelenOS fibrils.
- The loader, to initialize the TLS for itself (!). That is because the loader is a program just like any other, it is only special in the sense that it gets started by the kernel in an empty address space. So the loader must be able to do some rudimentary initialization of this address space for itself. This is done in `__libc_main` by checking whether a program control block has been passed to this function (which the loader does when calling `__libc_main` of the actual program), and if not, calling `tcb_make_initial`.

3.1.1 Bug: reading the incorrect ELF file

The subtle bug we identified in the HelenOS implementation of the loader was that it read the wrong ELF file when initializing the TLS. The `_tcb_data_offset` function, which is responsible for reading the TLS information from the ELF file, used the global variable `__progsymbols.elfstart` to get the memory location of the ELF file. Notice that this is correct in latter two of the three cases listed above, but not in the first one. In that case, `__progsymbols` refers to symbols of the loader itself, but the caller needs to know the size of the TLS of the program being loaded.

Once uncovered, the fix is simple – a new parameter has to be added to `_tcb_data_offset`, and we update all call sites.[1] However, identification of this issue proved difficult, as it had only manifested as data corruption when a program allocates significantly larger amount of thread-local data than the `loader` executable.

3.1.2 Bug: still using TLS parameters of the loader

After the first issue had been resolved, we noticed that for statically linked programs, the thread-local storage was still being initialized with incorrect parameters. The cause of this issue lies between the loader and from `libc`, where various pieces of code need to determine whether a given program is dynamically linked or not. All of these functions appear to be implemented correctly and according to comments in the source code. But upon closer inspection, we find that the interaction between them again causes the loader to again use the parameters of its own ELF file for TLS allocation for statically linked programs.

The `ldr_load` function, seen in Code fragment 3.1, first calls to the `elf_load` function from `libc` to load data about the ELF file and link it with any dependencies. Notably, this function detects if a binary is statically linked, and if so, the field `env` is left empty. After ELF loading is done, the loader allocates the TLS area and stores it as the thread control block prepared for the main thread of the program. This is ensured by the `rtld` module for dynamically linked programs, and the presumably simpler `tls_make` implementation is called for statically linked binaries as well as when dynamic linking is completely disabled via a compilation flag.

Code fragment 3.1 `ldr_load` from HelenOS loader program

```
static elf_info_t prog_info;

static int ldr_load(ipc_call_t *req)
{
    errno_t rc = elf_load(program_fd, &prog_info);
    if (rc != EOK) return 1;

#ifdef CONFIG_RTLD
    if (prog_info.env) {
        pcb.tcb = rtld_tls_make(prog_info.env);
    } else {
        pcb.tcb = tls_make(prog_info.finfo.base);
    }
#else
    pcb.tcb = tls_make(prog_info.finfo.base);
#endif
}
```

When we carefully look at the `tls_make` function listed in Code fragment 3.2, we notice a problem very similar to the first issue – the function checks the value of a global variable, but this variable again describes the loader, not the linker.

[1] The patch has been merged into HelenOS in this pull request: `https://github.com/HelenOS/helenos/pull/240/files`

Code fragment 3.2 `tls_make` from HelenOS `libc`

```
tcb_t *tls_make(const void *elf)
{
#ifdef CONFIG_RTLD
    if (runtime_env != NULL)
        return rtld_tls_make(runtime_env);
#endif

    return tls_make_generic(elf, memalign);
}
```

The correct handling of this situation is not as clear as in the first issue. We have considered adding an `rtld_t* env` parameter to the `tls_make` function, but this parameter would be completely nonsensical if `CONFIG_RTLD` is disabled, and generally a dependency on the run-time dynamic linker does not make sense in TLS allocation API.

However, we notice that although the `rtld_*` functions seem to be avoided for statically linked programs, in `__libc_main`, a rudimentary runtime is anyways initialized for programs that do not have one. This leads us to a decision to fix this issue by changing `elf_load` to create the `rtld` environment for all programs, and then we will be able to use `rtld_tls_make` to allocate thread-local storage for all programs. In fact, this allows completely removing the `rtld_init_static` function, which duplicated a lot of code, and our patch in the end removes more code than it adds, which is always a good sign.[2]

3.2 The `pthread_key` API

As noted in Chapter 2.3, our work needs to complete the implementation of the `pthread_key` API in HelenOS. This task is relatively straightforward, as shown in Code fragment 3.3, we can make use of built-in C thread-local variables. We create such array that is used to store in each thread the values associated with the keys, and leverage an atomic counter to distribute unique indices into this array. Our implementation imposes an artificial limit on the number of keys that can be used in one program, but we determine this to be acceptable. It is highly uncommon for programs to use a large amount of thread-local variables, and future extension of our implementation to an unlimited version is possible by replacing the array with a dynamically allocated array, i.e. a pair of a pointer and size.

After a discussion with HelenOS developers, support for destructors is omitted in our implementation, as this feature has little use in practice, and would require a lot of work to implement correctly.[3] This causes all Rust programs to print a warning about using this unsupported feature upon first spawning of an additional thread, but we did not identify any effect on functionality or correctness.

[2]The patch has been merged into HelenOS in this pull request: `https://github.com/HelenOS/helenos/pull/242/files`

[3]The discussion can be found in the comments of the pull request where our changes have been eventually merged to HelenOS: `https://github.com/HelenOS/helenos/pull/245`

Code fragment 3.3 Implementation of the `pthread_key` API

```
#define PTHREAD_KEYS_MAX 100
static fibril_local void *key_data[PTHREAD_KEYS_MAX];

static atomic_ushort next_key = 1; // skip the key 'zero'

void *pthread_getspecific(pthread_key_t key)
{
  assert(key < PTHREAD_KEYS_MAX);
  assert(key < next_key);
  assert(key > 0);

  return key_data[key];
}
// ... pthread_setspecific likewise ...

int pthread_key_create(
    pthread_key_t *key, void (*destructor)(void *)
) {
  unsigned short k = atomic_fetch_add(&next_key, 1);
  if (k >= PTHREAD_KEYS_MAX) {
    atomic_store(&next_key, PTHREAD_KEYS_MAX + 1);
    return ELIMIT;
  }
  if (destructor != NULL) {
        fprintf_once(stderr,
            "pthread_key_create: destructors not supported\n");
  }

  *key = k;
  return EOK;
}
```

3.3 Changes in Rust

In the Rust compiler, the only changes needed were to add the HelenOS target specifications. Although this was not a trivial task, since it sometimes required trial and error as the author was not familiar with the precise effects of all the options available in the target specification, the resulting code is not particularly interesting.

The rest of this section therefore focuses on our changes in the Rust standard library.

3.3.1 Filesystem API

Most of the filesystem APIs simply call to the respective functions in HelenOS `libc`. Some portions of the API are left unsupported, since the functionality does not exist in HelenOS. This is the case namely for file permissions and access times.

Our testing suite has helped us to find a bug in our implementation of the `read_dir` function. Originally, our code only called the `libc::opendir` function, and we used `libc::readdir` to get the individual entries only when the caller requested them by iterating over the `ReadDir` struct. However, if the directory was modified during this iteration, this implementation would skip some entries. Therefore the logic had to be changed to load all the entries to a vector rightaway, and on iteration return results from this vector.

3.3.2 Thread API

The thread API in Rust is relatively minimalistic. It requires the platform to provide a method for starting threads, yielding, sleeping and joining a running thread. When creating a new thread, the closure passed to `pal::Thread::new` is of type `Box<dyn FnOnce()>`, i.e. an allocation of a trait object, which uses dynamic dispatch. This, however, means that the Box needs to hold two pointers – one to the data and one to the virtual function table. Thus the Box cannot be cast into a standard C pointer, and one more level of indirection is needed. This implementation is shown in Code fragment 3.4, and is re-used from similar implementation for UNIX platforms.

In HelenOS, there is one extra complication – there is no `fibril_join` API. The HelenOS USB drivers contain for this reason a wrapper called `joinable_fibril`, so we have decided to implement a similar wrapper in Rust. We use a pair of a mutex and a condition variable – the mutex stores information about whether the fibril has exited, and the condition variable is used to notify the parent thread waiting for the exit. Then we wrap the `fibril_main` function with the piece of code in Code fragment 3.5.

The rest of the API implementations are rather mechanical in nature, and can be inspected in Attachment A.1. As mentioned in the introduction of Chapter 1, the amount of code sufficient to support Rust programs is not substantial, and the above is sufficient to pass our test suite and run real-world applications. The majority of the value of our work lies in understanding the integration of these systems and resolution of all the issues that arised.

Code fragment 3.4 Implementation of the `Thread::new` function

```
pub unsafe fn new(
    stack: usize,
    thread_main: Box<dyn FnOnce()>
)-> io::Result<Thread> {
    let wrapper: Box<Box<dyn FnOnce()>> = Box::new(thread_main);
    let p = Box::into_raw(wrapper) as *mut c_void;
    let id = libc::fibril_create_generic(thread_start, p, stack);
    if id.is_null() {
        drop(Box::from_raw(p));
        return Err("fibril_create failed");
    }
    libc::fibril_start(id);
    return Ok(Thread { id });

    extern "C" fn thread_start(main: *mut c_void) -> errno_t {
        unsafe {
            Box::from_raw(main as *mut Box<dyn FnOnce()>)();
        }
        libc::EOK
    }
}
```

Code fragment 3.5 Wrapping the `fibril_main` function to enable joining threads

```
let done = Arc::new((Mutex::new(false), Condvar::new()));
let done_copy = Arc::clone(&done);
let fibril_main: Box<dyn FnOnce()> = Box::new(move || {
    original_fibril_main();
    *done_copy.0.lock().unwrap() = true;
    done_copy.1.notify_all();
});
```

4 Using the toolchain to build Rust applications

In the previous chapter, we have described the changes necessary to make the Rust toolchain for HelenOS work. This chapter documents the process of building and using this toolchain, and provides a demonstration of its functionality by compiling a suite of both CLI and GUI applications, as outlined in chapter 2.4.

4.1 Installing the toolchain

Since neither HelenOS provides ready-made builds of libraries and header files for cross-compilation, nor is our new Rust toolchain already available pre-packaged, each piece of the toolchain for compiling Rust programs for HelenOS needs to be built from source. As a consequence, the first setup of the development environment requires several steps. Description of this process is also provided as documentation for the HelenOS targets in the Rust compiler, in the file `src/doc/rustc/src/platform-support/helenos.md` in Attachment A.1. The following sections provide an abridged copy of this guide with the full set commands included, hopefully allowing the reader to reproduce our results. We assume a build for the x86_64 version of HelenOS, the process for other architectures only requires substitution of the architecture in relevant commands.

4.1.1 Building the HelenOS toolchain

The first requirement is building the HelenOS cross-compilation toolchain for C and using it to build HelenOS libraries and header files. This can be done by following the *Compiling HelenOS from source* guide.[1] Full listing of the required steps is provided in Code fragment 4.1.[2]

This builds everything that we need from HelenOS to build Rust programs, copies the shared and static libraries to a path where the linker automatically searches for it, and stores the location of the `include` directory in a variable for later use.

4.1.2 Building the Rust compiler

Rust's toolchain is well automated, so the necessary steps are relatively simple. After Rust is installed on the development machine, preferrably using the official

[1] `https://www.helenos.org/wiki/UsersGuide/CompilingFromSource`

[2] As of the date of submission of this thesis, not all of our pull requests to HelenOS have been merged. Therefore, the provided commands download our fork of HelenOS where the patches are applied. We expect this requirement to cease soon and the official HelenOS repository at `https://github.com/HelenOS/helenos.git` to be usable.

Also, we assume that the toolchain gets installed in the default location, which is `$HOME/.local/share/HelenOS/cross/`. The path is shown in the output of the `toolchain.sh` command, and can be modified by setting the `CROSS_PREFIX` environment variable. In that case, further usages of the relevant paths need to be updated.

Code fragment 4.1 Building the HelenOS toolchain

```
mkdir helenos helenos-build
git clone https://github.com/mvolfik/helenos.git --depth 1

# Build amd64-helenos-gcc
cd helenos
./tools/toolchain.sh amd64
PATH=$PATH:$HOME/.local/share/HelenOS/cross/bin

cd ../helenos-build

# Build the shared libraries
../helenos/configure.sh amd64
ninja export-dev

# Copy the libraries to where linker finds them automatically
cp -P export-dev/lib/* ~/.local/share/HelenOS/cross/amd64-helenos/lib/

# Save the location of header files for later use
HELENOS_INCLUDE_BASE=$PWD/export-dev/include
export HELENOS_INCLUDE_BASE
```

rustup installer,[3] the toolchain can be simply built by extracting our modified Rust source code from Attachment A.1 and running the following command.

```
./x build library --stage 1 \
  --target x86_64-unknown-linux-gnu,x86_64-unknown-helenos
```

This orchestrates the full boostrap process of downloading the previous compiler version, using it to build a new version of the compiler with the HelenOS target definitions, and then preparing a build of the standard library for the targets specified on the command line.

After we have the toolchain built, we can link it with Rustup, which will allow us to easily use this custom version of Rust to compile programs anywhere on our system. To link this toolchain with the name `local`, we run the following command in the Rust source directory.

```
rustup toolchain link local build/host/stage1
```

4.1.3 Docker-based automated toolchain build

To make the process of building the toolchain easier and reproducible, we have also made available an automated build system of the whole toolchain via a templated Dockerfile. It can be found in Attachment A.3. To use it, the Python script `gen.py` has to be invoked to generate the Dockerfile from a template. The

[3]Downloadable from `https://rustup.rs`

script allows selection of the CPU architecture-variant of HelenOS to build, and takes a list of URLs of Git repositories with Rust programs to build.

The Dockerfile then defines two sets of artifacts, implemented as two different build stages. The user can either ask to only build the applications, and then use them in their own version of HelenOS. Or the build can execute one more build stage, which copies these applications into a HelenOS build directory inside the container, and builds a bootable ISO image of the full system with the applications preinstalled.

4.2 Rust standard library tests and CLI utilities

As we explain in Chapter 2.3.2, running the Rust standard test suite on HelenOS is not easily possible. Instead, we have built a small custom test suite that verifies the functionality of the three main system APIs that we implemented – filesystem, multithreading and thread synchronization.[4] Source code of this test suite is attached as Attachment A.5, and can be built with the following command, assuming we have the toolchain linked under the name `local`.

```
cargo +local build --target x86_64-unknown-helenos --release
```

This will produce a binary in `target/x86_64-unknown-helenos/release/rtest`, that we can then run on HelenOS. The test suite is also preinstalled in the ISO image in Attachment A.2, where it can be simply run as the command `rtest`.

A passing result of the test suite on 64-bit ARM version of HelenOS is shown in Figure 4.1. We are showing the demo of this platform because, as we will explain later, this platform will not be able to run any of our GUI applications. This test suite has helped us to find some omissions in our filesystem support, and we have verified that it now passes on all HelenOS platforms we are able to run Rust programs.

4.3 Image processing tools

As we predicted, after we have written the compiler and standard library support, compiling Rust programs for HelenOS is very straightforward. The first of the selected applications, `resvg`, could be downloaded and compiled just like our test suite, without any modifications necessary.

Compiling `imagecli` did require some extra work, since it depends on crates `rand` and `atty`. These crates provide very simple wrappers around libc calls, so we need to add the necessary HelenOS bindings. This is trivial, as can be seen from the complete patch to `atty` listed in Code fragment 4.2. Similarly to our work on the standard library, we then just add links to our patched version to the `Cargo.toml` file of `imagecli` project, and then we are also able to build and run it on HelenOS.[5]

[4]Please note that parts of the test suite were generated using a large language model (specifically GPT 4o from OpenAI). The author has then only manually reviewed and slightly modified the tests. A disclaimer is also provided at the top of the relevant source files.

[5]A fork with this patch applied is available at `https://github.com/mvolfik/imagecli-rs`.

Figure 4.1 Passed test suite on 64-bit ARM HelenOS

Code fragment 4.2 HelenOS support in the `atty` crate

```
#[cfg(target_os = "helenos")]
pub fn is(stream: Stream) -> bool {
    extern crate libc;

    unsafe {
        let handle = match stream {
            Stream::Stdout => libc::stdout,
            Stream::Stderr => libc::stderr,
            Stream::Stdin => libc::stdin,
        };
        libc::isatty(libc::fileno(handle)) != 0
    }
}
```

The author assigns greater importance to demonstration that our work made it easy to download and build a non-trivial Rust application for HelenOS, than to the specifics of individual applications that we run as a demonstration. Consequentially, the mentioned patches are not attached to our thesis, as they contain relatively trivial modifications.

4.4 GUI applications

The implementation of a custom runtime of the Iced GUI library has been pretty straightforward and can be found in Attachment A.4. After implementing this runtime, we have created a simple image viewer as we planned in 2.4. In the examples section of the Iced library, we also found an implementation of Conway's Game of Life, which we have also successfully ported to HelenOS.

The application logic is not particularly interesting. However, the implementation of the Iced runtime contains multiple interesting design decisions, and we will dedicate the following section to describing our solution.

4.4.1 Wrapping the Iced GUI library

Our custom runtime is split into a shared part, and a platform-specific startup and event detection code. The shared part, implemented in `src/lib.rs`, manages the `tiny-skia` renderer and proxies calls between the running Iced application and the platform-specific code. The platform-specific code can queue any events and messages using the generic `Program` interface. Then, when a new frame should be drawn, the platform-specific code calls the `update` method from the shared part, which lets the application process all queued events, renders it to the graphical surface of the window, and then requests this to be presented onto the screen.

On the platform-specific side, we have first implemented a minimalistic custom mapping to the `winit` library, inspired by the existing `iced-winit` runtime. This has multiple advantages. Firstly, this work allowed us to familiarize ourselves with the Iced library and the required paradigm of event handling and rendering. Secondly, the option of easy compile-time swapping of the platform-specific runtime backend allowed us to later develop the image viewer in our local environment, allowing faster testing of our changes.

After gaining confidence that our winit wrapper properly processes input events and the shared part correctly calls the `tiny-skia` renderer, we have replicated the backend functionality using native calls to HelenOS libraries.

The HelenOS implementation required carefully crafted unsafe code to correctly provide an interface for the C callbacks. Definition of a callback is relatively simple, as it simply takes a pointer to a Mutex containing the application object, which it locks and calls the required method on it. An example of one such callback is shown in Code fragment 4.3.

However, to be able to soundly create and use such pointer, this mutex needs to be stored at a pinned memory location, i.e. it its adress must not ever change. To guarantee this, we use the `Pin` contract from the standard library. The startup code that handles this, included in Code fragment 4.4, is further complicated by the requirement of passing the `CreateSendMsg` closure to the application, which we explain in the following section.

Code fragment 4.3 Raw HelenOS event handler

```
type Arg<T> = Mutex<App<T>>;
impl<T> App<T> {
    const CALLBACKS: helenos_ui::ui_window_cb_t =
        helenos_ui::ui_window_cb_t {
            paint: Some(Self::paint_event),
            // ...
        };

    unsafe extern "C" fn paint_event(
        _window: *mut helenos_ui::ui_window_t,
        app: *mut ffi::c_void,
    ) -> i32 {
        let app = unsafe { &*(app as *const Arg<T>) };
        let app = &mut *app.lock().unwrap();
        app.paint();
        0 // EOK
    }

    fn paint(&mut self) {
        self.inner.update(self.cursor);
        unsafe { helenos_ui::gfx_update(
            helenos_ui::ui_window_get_gc(self.window.raw.as_ptr())
        ) };
    }
}
```

Code fragment 4.4 Initialization of application and callbacks[6]

```
fn run_app_in_window<App: ProgramExt>(
    window: Window,
    create_app: impl FnOnce(SendMsgFn<App>) -> App,
) {
    let mut app = std::pin::pin!(MaybeUninit::uninit());
    let app_ptr = app.as_ptr();
    let send_msg = Box::new(move |msg: App::Message| {
        // deref the pointer - the app must be initialized now
        let app = unsafe { &*(app_ptr.0) };
        app.lock().unwrap().inner.program.queue_message(msg);
    });
    app.set(MaybeUninit::new(Mutex::new(App {
        inner: AppInner::new(create_app(send_msg)),
        _pin: std::marker::PhantomPinned,
        // ...
    })));
    let app = unsafe { app.assume_init_ref() };

    let callbacks = std::pin::pin!(App::<T>::CALLBACKS);
    unsafe {
        helenos_ui::ui_window_set_cb(
            window.get_raw_ptr(),
            callbacks.as_mut_ptr(),
        );
    }

    // ... run the event loop

    // Stop the application, now we can safely destroy the
    // pinned values.
}
```

4.4.2 Background tasks

Our implementation of the image viewer initially did the image resizing in the main thread. However, this caused the application to freeze while the processing was running, and for Conway's Game of Life, implementing background tasks becomes a strict requirement, since the game needs to periodically send a `Tick` message to step the simulation without any user interaction.

To allow our Iced applications to run background tasks that interact with the state of the application, they need to be able to send messages to the main thread. Therefore, when creating the application object defined by the user of our Iced runtime, we need to pass it a closure that references the application object. This

[6]The actual implementation is more complicated, this simplified version serves to illustrate the design decisions described in the text.

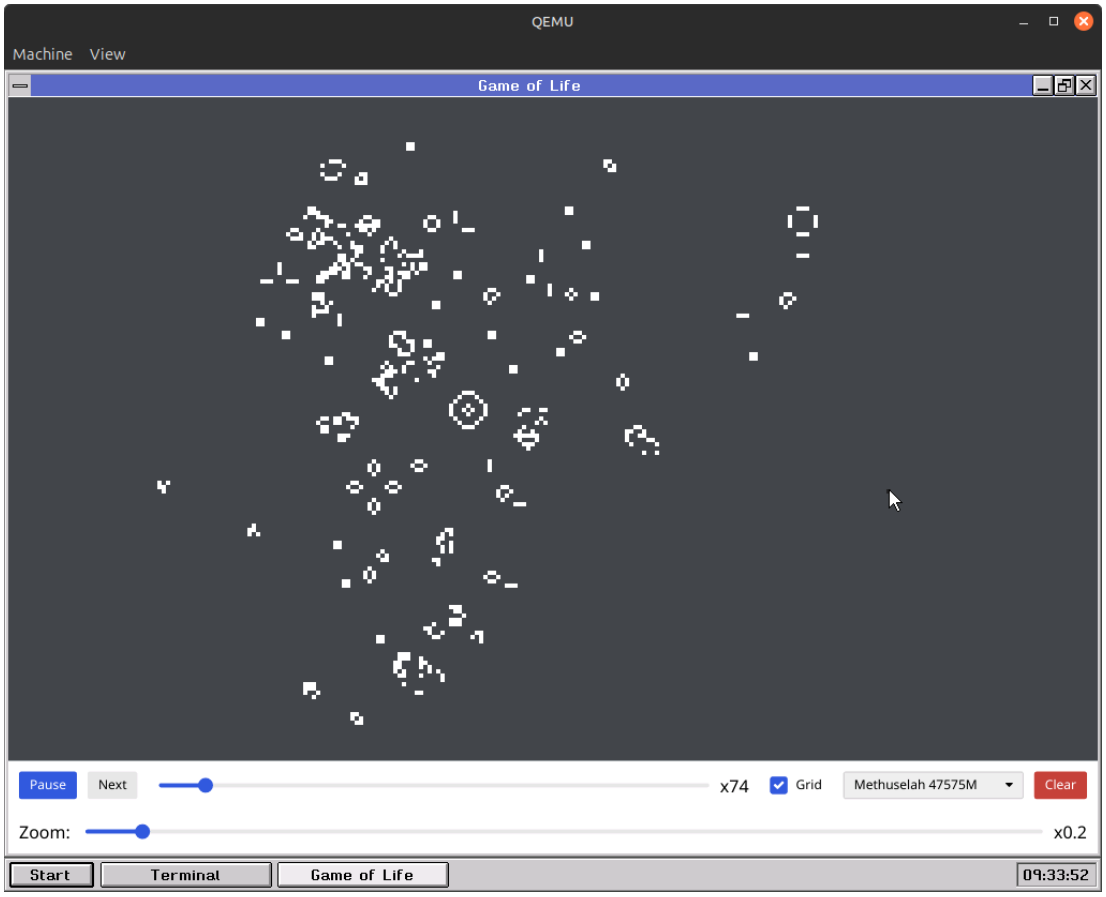

Figure 4.2 Game of Life running in HelenOS

creates the second complication for the code in Code fragment 4.4. We first need to create a pinned memory location that contains `MaybeUninit<Mutex<App>>`.[7] We use this memory location to create the `send_msg` closure, which we then use to truly initialize the application.

Implementation of a background task is then quite straightforward. The application uses either a channel, or a pair of mutex and condition variable, to request jobs from the worker, or to ask it to stop. The synchronization primitive selection depends on the nature of the jobs. For the worker in image viewer that handles resizing the viewed image, we use a mutex, since we are only interested in the latest zoom request, and any intermediate zoom levels, requested by the user while the worker was busy, can be ignored.

As an example of different requirements, Game of Life uses a channel to queue simulation steps to the worker. The worker then needs to maintain its internal queue, to detect the `Stop` request as soon as possible and ignore all other pending requests at that moment.

[7]Remember, safe Rust code guarantees no undefined behavior, i.e. among other things no access to uninitialized memory. Therefore, normal Rust types are required to always contain a valid value. The `MaybeUninit<T>` type serves as an "escape hatch" from this rule – it has the same memory layout as `T`, but is allowed to contain uninitialized memory. This allows taking a pointer to a value, using it to construct the value itself and later casting the type of the memory location to the fully initialized type with the unsafe function `assume_init`. This patterns aids in creating self-referential types and other advanced constructs common in C that cannot be efficiently expressed in safe Rust.

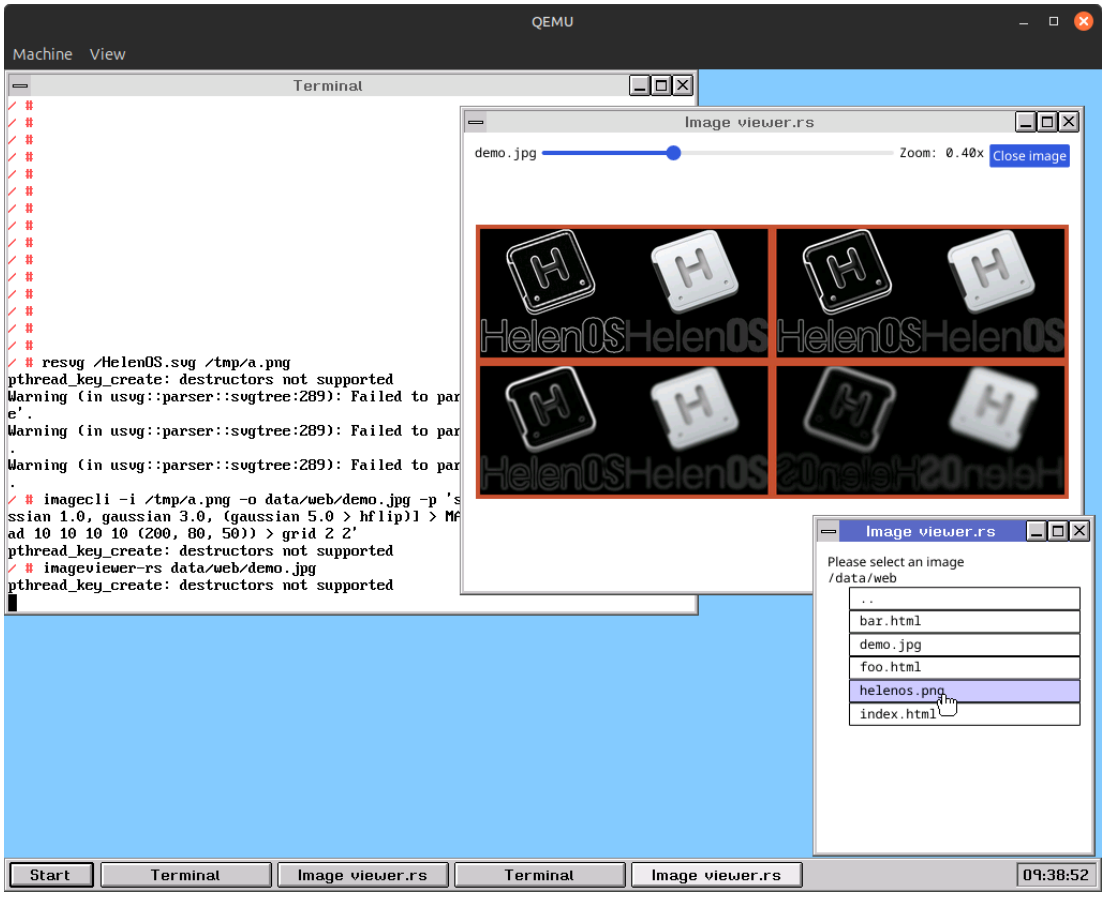

Figure 4.3 Image processing and image viewer in HelenOS

4.5 Summary of platform support

We have attempted to port and test all of the above Rust programs on as many architectures as possible. However, we encountered numerous issues of varying kind, specific to individual architectures. We have spent non-trivial amount of time trying to resolve each issue, but in many cases, even correctly identifying the cause of the problem proved challenging and provided no indication of the difficulty of the solution. We have therefore determined that fully resolving these issues would excessively inflate the scope of this thesis, and accepted that improving the platform support will remain a task for future work.

The full list of supported platforms is shown in Table 4.1, with footnotes shortly explaining the issues that prevented us from better support of each given platform. Since for some platforms, we were unable to run GUI applications and verify the results of image processing, we have also tested the `chksum`[8] CLI utility, which calculates checksums of files, and thus demonstrates some practical usability of Rust on the given platforms.

Architecture	Test suite and CLI utilities	Image processing	GUI apps
x86_64	✓	✓	✓
PowerPC	✓	✓	✓
IA-32[a]	✓	✓	✗
SPARC[b]	✓	✓*	✗
ARM 64-bit[c]	✓	✓*	✗
ARM 32-bit[d]	✗		
MIPS[e]	✗		
RISC-V[e]	✗		
IA-64[f]	✗		

Table 4.1 HelenOS platforms and their support for Rust programs

[8] `https://github.com/chksum-rs/cli`

*The image processing tools ran correctly, but we could neither display the resulting images on these systems, nor export them out from the system due to missing external disk support. Therefore we could not verify the correctness of the output.

[a] On the IA-32 architecture, some functions in native HelenOS libraries contain vector instructions, which trigger *General Protection Fault* due to accessing incorrectly aligned stack variables. We have identified this to be a fault on the Rust side, as it seemed to generate stack frames of size with insufficient alignment. Unfortunately, we were unable to find a way to fix this in the compiler. The data layout we specify for the platform correctly includes `S128`, i.e. 128-bit alignment of the stack, and we did not find out why this value is not respected.

[b] Due to SPARC bootloader limitations, executables of GUI applications could not be included directly in the system image, as it is too large and causes the full kernel image to exceed the maximum size of 8 MB. We have attempted optimizing the builds for code size, enabling LTO and stripping symbols from the binaries, but this was not sufficient. We have also considered the option of including the programs on a separate disk mounted into the system, but this functionality is missing or currently broken in the SPARC HelenOS port.

[c] The ARM64 port of HelenOS does not have window system support, so GUI applications cannot run there.

[d] On ARM32, we encountered issues with missing implementations of `__atomic_*` and `__sync_*` builtin functions when linking the Rust standard library. These functions are generated by compilers in places where atomic operations are used. On supported CPU architectures, these functions are implemented using the respective atomic instructions, but if an instruction is not available, a call to an external function is generated, in which the operating system can implement this using a global lock or other methods.[16] We have attempted to provide the missing implementations of these functions, but then we encountered further problems with handling of ELF relocations in the HelenOS dynamic loader. At that point, we determined that the effort required required to correctly support this platform would be excessive, and decided to leave it fully unsupported for now.

[e] These two variants of HelenOS we were unable to run at all, and therefore we did not attempt porting Rust to them.

[f] The Itanium architecture is unsupported by Rust/LLVM.

Conclusion

We have succeeded in enabling compilation of Rust programs for HelenOS. We have added five functioning HelenOS targets to the Rust compiler, and implemented a substantial part of the required support in the Rust standard library. We have verified that our implementation is correct by successfully running a test suite that covers these APIs on all five supported platforms.

We have demonstrated the practical usability of our subset of standard library support, since we were able to run multiple real-world Rust applications on HelenOS, albeit not on all platforms due to unresolved issues and limitations specific to individual HelenOS variants. Still, we have managed to run the full suite of our ported programs on two platforms, `x86_64` and 32-bit PowerPC, and a bootable image of `x86_64` HelenOS with our apps installed is provided in Attachment A.2. We also note that the remaining HelenOS platforms should still be able to benefit from our work when the outstanding issues are resolved.

The author also takes pride in integrating the Iced GUI library with the HelenOS window system, which allows running a completely new paradigm of graphical applications on HelenOS. Almost effortless porting of Conway's Game of Life, implemented in just around 1000 lines of Rust code, demonstrates the power of the library.

Overall, we have increased the number of applications available for HelenOS and provided the groundwork for porting further Rust programs. In many cases, this only requires adding HelenOS support in crates that integrate with C APIs of the operating system.

And finally, with no less importance, we have comprehensively described the process of porting Rust to a new platform, hopefully aiding future developers in their work on expanding Rust support to more operating systems.

Future work

This thesis has started in an unexplored area, with no prior efforts related to running Rust code on HelenOS. Given the vastness of the Rust ecosystem, there is a lot of work outstanding. Here is a list of a few areas of further work that the author considers interesting:

- **Expanding the standard library support and running the official test suite.** As explained in Sections 2.3.1 and 2.3.2, our work only implements the most important features of the standard library, and does not run the official test suite due to missing networking support. This task would require extra work in the underlying HelenOS implementation, but it would push our work to an important milestone, running the official Rust standard library test suite.

- **Improving the architecture support.** As we wrote in chapter 4.5, ports only to three[9] of the eight possible CPU architectures are fully functional.

[9]Here we dare to also count SPARC, although we did not run GUI applications on it. That is because the only restriction is the executable size limitation, which needs to be resolved on the side of HelenOS.

During our work, we have encountered multiple issues where it was not clear whether the problem was in HelenOS or in the output produced by Rust – this was the case with for example with thread-local storage access, as discussed in section 3.1. We suspect that the issues that prevented us from running Rust programs on some platforms are of similar nature, and thus expanding the architecture support would be a tedious undertaking. However, the author argues that it full platform support were achieved, Rust could become a first-class language for development of HelenOS userspace applications.

- **Supporting a selected async runtime.** We did not have the chance to test any async code on HelenOS, although it could open interesting possibilities for interacting with the asynchronous paradigm communication between HelenOS system services.
- **Porting to HelenOS the Rust compiler itself.** Although Rust by default uses LLVM for assembly generation, and porting whole LLVM to HelenOS would be an immensely difficult task (as it would likely require complete C++ support), the experimental Cranelift codegen backend is written purely in Rust code, and thus could be much easier to port. We did not explore what would be the other requirements for running `rustc` on HelenOS.

Bibliography

1. *HelenOS Wiki*. Design differences from UN*X and POSIX systems [online]. 2017. [visited on 2025-04-15]. Available from: `https://www.helenos.org/wiki/DiffFromUnix`.
2. *HelenOS Wiki*. HelenOS Network Transport API tutorial [online]. 2017. [visited on 2025-04-15]. Available from: `https://www.helenos.org/wiki/NetworkAPITutorial`.
3. Young de la Sota, Miguel. What the Hell Is a Target Triple? [online]. 2025 [visited on 2025-05-01]. Available from: `https://mcyoung.xyz/2025/04/14/target-triples/`.
4. *The AVR-Rust Guidebook*. Adding AVR support to a crate [online]. The AVR-Rust project, 2025 [visited on 2025-05-05]. Available from: `https://book.avr-rust.org/005-add-avr-support-to-crate.html`.
5. Klabnik, Steve; Nichols, Carol; Krycho, Chris; contributions from the Rust Community. *The Rust Programming Language*. To `panic!` or not to `panic!` [online]. 2024. [visited on 2025-04-15]. Available from: `https://doc.rust-lang.org/book/ch09-03-to-panic-or-not-to-panic.html`.
6. *ARM7TDMI Technical Reference Manual*. The Thumb instruction set [online]. ARM Limited, 2001. R4p1 [visited on 2025-04-15]. Available from: `https://developer.arm.com/documentation/ddi0210/c/CACBCAAE`.
7. *The rustc book*. `{arm,thumb}*-none-eabi(hf)?` targets [online]. [N.d.]. [visited on 2025-04-15]. Available from: `https://doc.rust-lang.org/nightly/rustc/platform-support/arm-none-eabi.html#instruction-sets`.
8. Adding Rust Stable libstd Support for Xous. *Xobs' Blog* [online]. 2021 [visited on 2025-04-15]. Available from: `https://xobs.io/porting-rust-standard-library-to-a-new-operating-system/`.
9. Wang, Huibo; Wang, Pei; Ding, Yu; Sun, Mingshen; Jing, Yiming; Duan, Ran; Li, Long; Zhang, Yulong; Wei, Tao; Lin, Zhiqiang. Towards Memory Safe Enclave Programming with Rust-SGX. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. London, United Kingdom: Association for Computing Machinery, 2019, pp. 2333–2350. CCS '19. isbn 9781450367479. Available from doi: `10.1145/3319535.3354241`.
10. *Rust Compiler Development Guide*. Adding a new target [online]. [N.d.]. [visited on 2025-04-15]. Available from: `https://rustc-dev-guide.rust-lang.org/building/new-target.html`.
11. Jindrák, Jaroslav. *C++ Runtime for HelenOS*. 2022. Mgr. thesis. Charles University, Faculty of Mathematics and Physics.
12. Pressman, Roger S. *Software engineering : practitioner´s approach*. 5th ed. New York: McGraw-Hill, 2001. McGraw-Hill series in computer science. isbn 0-07-365578-3.

13. *egui: an easy-to-use GUI in pure Rust*. Why immediate mode [online]. 2025. [visited on 2025-04-15]. Available from: https://github.com/emilk/egui/blob/d78fc39/README.md#why-immediate-mode.

14. HORN, Melody. A 2025 Survey of Rust GUI Libraries [online]. 2025 [visited on 2025-04-15]. Available from: https://www.boringcactus.com/2025/04/13/2025-survey-of-rust-gui-libraries.html.

15. JIMÉNEZ, Héctor Ramón. *iced – A Cross-Platform GUI Library for Rust*. Architecture [online]. 2024. [visited on 2025-05-01]. Available from: https://book.iced.rs/architecture.html.

16. *Using the GNU Compiler Collection (GCC)*. `__sync` builtins [online]. Free Software Foundation, Inc., 2025 [visited on 2025-04-29]. Available from: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fsync-Builtins.html.

A Attachments

A.1 `rust.tar.gz`: Rust compiler source code

This archive contains the full source code of the Rust compiler and standard library with HelenOS support included. Our work is in the following files and directories:

- `compiler/rustc_target/src/spec/base/helenos.rs`
- `compiler/rustc_target/src/spec/targets/*_unknown_helenos*.rs`
- `library/std/src/sys/pal/helenos/`
- `library/std/src/sys/fs/helenos.rs`
- `library/std/src/sys/random/helenos.rs`
- `library/std/src/sys/stdio/helenos.rs`
- `src/doc/rustc/src/platform-support/helenos.md`

The exact patch can be inspected at Github, where our work is also uploaded. See the `helenos` branch of `https://github.com/mvolfik/rust`, the patch can be seen at `https://github.com/rust-lang/rust/compare/master...eb465ad`.

Note that the attached archive purposely contains the `.git` directory. This is necessary because Rust bootstrap process looks at the Git history of the file `src/bootstrap/download-ci-llvm-stamp` to determine version of LLVM to download from Rust CI servers.

A.2 `helenos-rust-x86_64.iso`: Bootable image with Rust programs

This attachment is a bootable ISO image of the full HelenOS system with installation of the Rust programs that were mentioned in this thesis. It is easily runnable on any system with QEMU or VirtualBox. For QEMU, the command is:

```
qemu-system-x86_64 -device e1000,netdev=n1 -netdev \
  user,id=n1,hostfwd=tcp::8080-:8080,hostfwd=tcp::8081-:8081 \
  -usb -device nec-usb-xhci,id=xhci -device usb-tablet -device \
  intel-hda -device hda-duplex -serial stdio -boot d \
  -m 2G -enable-kvm -cdrom helenos-rust-x86_64.iso
```

The following Rust applications are installed:
`rtest`, `chksum`, `resvg`, `imagecli`, `imageviewer-rs`, `life`

A.3 `autobuild`: Automatic build system

A.4 `gui-apps`: Rust GUI applications for HelenOS

A.5 `rust-tests`: Simple test suite of Rust libstd