Exploring a libc free Rust Standard Library

Nicolas Cai
ncai34@gatech.edu

Saiguatam Bonam
sbonam3@gatech.edu

Kinshuk Phalke
kphalke3@gatech.edu

1 Abstract

Our project aims to create a minimal replacement to the Rust Standard Library written for BuzzOS. This will be implemented completely free of libc, by using safe abstractions around system calls in the BuzzOS kernel, which can be then be layered with a minimal standard library implementation.

2 Motivation

- First step to getting userspace programs written in Rust in the BuzzOS kernel.
- Free of any C dependencies which should allow it to run with a single language and a single compiler.
- Would allow for cross compilation of any Rust Program made for Windows, MacOS, Linux on the BuzzOS kernel, if used in conjunction with a multi-arch linker (like lld).

3 Literature Review

1. https://github.com/japaric/steed
    - Steed is a Rust project that aimed to create a standard library that does not depend on C.
    - The goal was to use raw Linux system calls to implement the Rust standard library.
    - The project aimed to provide hassle-free cross compilation and better optimizations.
    - Steed implemented standard I/O operations, filesystem operations, collections, minimal support for threads and TLS, and UDP and TCP sockets.
    - However, Steed was missing a proper allocator, math stuff, unwinding, hostname lookup, and errno implementation.
2. https://github.com/rust-lang/rfcs/issues/2610
    - The proposal is for a standard library for Linux systems, free of C dependencies, as a successor to the inactive Steed project.
    - The proposed library would implement the Rust standard library using raw Linux system calls, and would be implemented for libc-less targets (e.g. x86_64-linux) as part of the rust-lang/rust project.
    - An RFC is suggested to evaluate the return on investment for a libc-less standard library and to determine if there are other ways to achieve the same results.
    - The status of Steed as of its last commit in 2017 is described, including what was working and what was missing, such as a proper allocator and support for unwinding and hostname lookup.
3. https://internals.rust-lang.org/t/refactoring-std-for-ultimate-portability/4301
    - This is a proposal to achieve various goals regarding the std library:

- Locate any platform-specific portions of std that are not in sys and relocate them there.
- Sort out std::sys's dependencies such that it only depends on the code in sys common and not on anything else in std.
- To prepare for switching to the platform-dependent pal common crate, disentangle dependencies in std::sys common so they do not depend on either std::sys or the rest of std.
- Determine the refactoring steps required to separate the different I/O modules in std so they may be extracted cleanly.

4. https://internals.rust-lang.org/t/libsystem-or-the-great-libstd-refactor/2765

    - Separate behavior that depends on the platform and make sure it only appears in libstd::sys. Address all the leaky abstractions that are now present to create a single location for the system implementation.
    - Increasing the binary target_family will enable libraries to recognize additional platforms.
    - In order for libsystem to be removed out of libstd::sys, remove the circular dependency between the two libraries.
    - Refrain from making irrational allocations and generally clean up the system code.
    - As much as possible, switch over to native Rust code from rust_builtin.

4 Work Plan

Our project will support the following userspace functions:

- I/O (e.g. Stdin, Stdout)
- Boxed
- Cell
- Clone
- Copy
- Panic
- Integer Types
- Option Monad
- String
- Collections (e.g Vec, VecDeque, HashMap, BTreeMap, HashSet, BTreeSet, BinaryHeap)

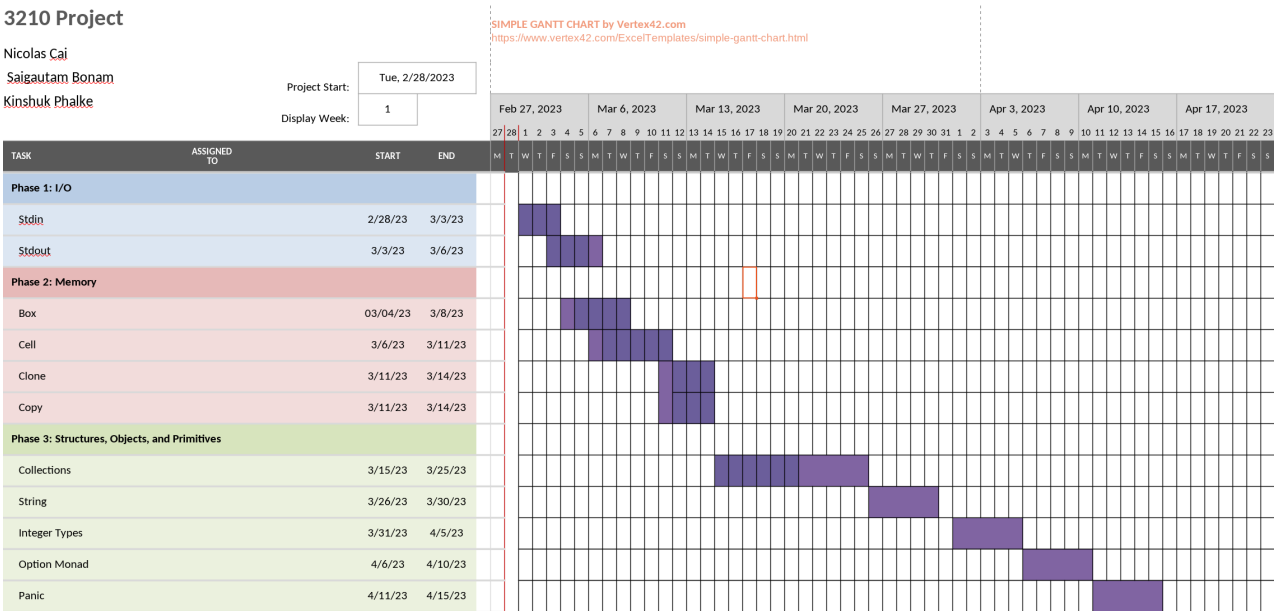

Figure 1. Our current Gantt chart plan for tackling each function.

5 Potential Blockers

1. We might face some problems matching the behaviour that the current rust standard library has.
2. Might face some problems having OS syscalls or having to match some unimplemented syscalls.
3. Might face some problems calling unsafe parts of the kernel code, could be overcome with a simple wrapper over the functional call.