Exploring a libc free Rust Standard Library

Nicolas Cai
ncai34@gatech.edu

Saigautam 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.

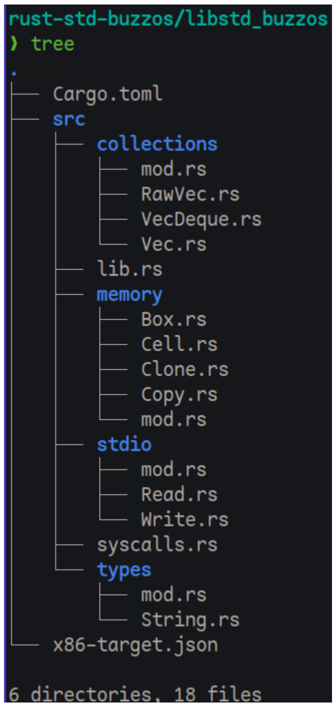
4. https://internals.rust-lang.org/t/libsystem-or-the-great-libstd-refactor/2765
   - Separate behavior that depends on the platform and make sure it only appears in libstd::sys. Address all the leaky abstractions that are now present to create a single location for the system implementation.
   - Increasing 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 Overview of Completed Work

As a high-level overview (specific details are given in the next section on Design and Implementation, we completed the implementation of several things related to the standard library:

- System call handlers
  - Sbrk
  - I/O (stdout), implemented a write syscall that writes to console as well as a temporary read syscall that relies on file descriptors which was not in the scope of our project.
- Creation of a standard library and making system calls to the kernel
- Standard library classes and wrappers:
  - Box
  - Cell
  - Clone
  - Copy
  - String
  - Collections (e.g Vec, VecDeque)
- Integration with userspace that Joao pushed to BuzzOS this semester. That is, we can now run a program using our library in BuzzOS userspace!

5 Design and Implementation



```
rust-std-buzzos/libstd_buzzos
❯ tree
.
├── Cargo.toml
├── src
│   ├── collections
│   │   ├── mod.rs
│   │   ├── RawVec.rs
│   │   ├── VecDeque.rs
│   │   └── Vec.rs
│   ├── lib.rs
│   ├── memory
│   │   ├── Box.rs
│   │   ├── Cell.rs
│   │   ├── Clone.rs
│   │   ├── Copy.rs
│   │   └── mod.rs
│   ├── stdio
│   │   ├── mod.rs
│   │   ├── Read.rs
│   │   └── Write.rs
│   ├── syscalls.rs
│   └── types
│       ├── mod.rs
│       └── String.rs
└── x86-target.json

6 directories, 18 files
```

Figure 1: Project layout for our standard library implementation.

We began our project by adding support for syscalls inside BuzzOS. We made a fork of BuzzOS on an external github repository located here. We first created syscall handlers located in `kernel/src/interrupt_handlers.rs` initially trying to use traps (namely the `x86-interrupt` type. However, we realized that BuzzOS did not have a working trap handler, so using interrupts would result in Page Faults that would cause kernel panic.

Instead, we created our own `TrapHandler` wrapper without using `x86-interrupts` and modified the Interrupt Descriptor Table (IDT) to include 3 syscalls: `sbrk, read, write`. Later, Joao pushed an update to BuzzOS that changed the layout of our syscalls, as we had initially made three separate addresses in the IDT for each syscall. Additionally, his changes added BuzzOS trap handling, so we merged our syscall handlers to utilize the more proper trap handlers.

We'll now discuss how we implemented the `sbrk` syscall handler. Here is the handler shown in code:

```rust
pub fn sbrk() {
    println!("[KERNEL] SBRK called");
    let trapframe = unsafe { *SCHEDULER.lock().get_trapframe().unwrap().clone() };
    let mut res: usize = 0;
    let mut req_size: usize = trapframe.ecx;
    let mut addr: *mut u8;

    let layout: Layout;
    match Layout::from_size_align(req_size, 4) {
        Ok(x) => layout = x,
        Err(y) => {
            println!("[KERNEL] Layout Error: {}", y);
            unsafe {
                asm!("mov eax, 0");
            };
            return;
        }
    };

    unsafe {
        addr = HEAP_ALLOCATOR.alloc(layout);
    };

    if addr.is_null() {
        println!("[KERNEL] SBRK got no free memory\n");
        unsafe {
            asm!("mov eax, 0");
        };
    }
    println!(
        "[KERNEL] SBRK got {:#?} bytes starting at {:#x?}",
        req_size, addr
    );
    unsafe {
        asm!(
            "mov eax, {}",
            in(reg) addr,
        );
    };
}
```

`sbrk` takes in one input, which is the required size. Initially, before Joao pushed updates, we were utilizing Rust inline assembly to store arguments in the general purpose registers (from the standard library), and reading these values in the kernel. With Joao's trap handling and process scheduler updates, we switched arguments to use the trapframe object.

For the `sbrk` functionality, we utilized the existing `Layout` align function in the kernel with a packing size of 4 bytes. We then utilize the global `HEAP_ALLOCATOR` to find a block of memory with that layout, and return a pointer to the location in memory. In order for the endpoint in the standard library to receive the address, we store the pointer inside the register `eax` using inline assembly.

We similarly defined a system call for write (which just writes to console), and read. Since our project did not involve file descriptors, the read syscall does not have any functionality in it except for passing in an integer for file descriptor and the string buffer.

Now, the second hardest part of this project (the first being developing the sbrk handler to work correctly with inline assmebly) was developing the standard library to interface with the syscall handlers we created in the kernel. Here is an example of a syscall from our std library with just one argument:

```
#[inline]
pub unsafe fn syscall1(n: Sysno, arg1: usize) -> usize {
    let mut ret: usize = 0;

    asm!("int 64",
        inlateout("eax") n as usize => ret,
        in("ecx") arg1,
        options(nostack, preserves_flags));
    ret
}
```

We made an enum called Sysno which has values for sbrk, read, and write. With Joao's changes, are syscalls are at address 0x64 and the specific syscall is specified using the eax register. We store the argument in ecx (for additional arguments, edx and edi are used). Previously, we had implemented it such that each syscall was a different address, so we added the syscall number to the address 0x20 in the IDT. We would use this function syscall1 to call sbrk, as sbrk takes in only one parameter: the required block size. We defined functions syscall2 and syscall3 similarly.

The next section of this project was developing some libraries to use in the stdlib. We implemented a variety of memory operations and data structures. We'll overview our implementation of the Vector data structure, specifically the portion utilizing the sbrk system call below:

```
 1      pub fn grow(&mut self, is_init: bool) {
 2          let new_cap = if self.cap == 0 {
 3              1
 4          } else if is_init {
 5              self.cap
 6          } else {
 7              2 * self.cap
 8          };
 9
10          let new_layout = Layout::array::<T>(new_cap).unwrap();
11          assert!(
12              new_layout.size() <= isize::MAX as usize,
13              "Allocation too large"
14          );
15
16          let new_ptr: *mut u8 = if self.cap == 0 {
17              unsafe { syscall1(Sysno::Sbrk, new_cap) as *mut u8 }
18          } else {
19              unsafe { syscall1(Sysno::Sbrk, self.cap) as *mut u8 }
20          };
21
22          self.ptr = match NonNull::new(new_ptr as *mut T) {
23              Some(p) => p,
24              None => NonNull::new(null_mut()).unwrap(),
25          };
26
27          self.cap = new_cap;
28      }
```

This is the grow function of the Vec object (specifically an object called RawVec that we use as a wrapper for Vec). We have standard vector functionality such as doubling the capacity when the vector is full. The sbrk syscall occurs in lines 17-19. We pass in the new vector capacity into sbrk to get a new block of memory for the vector. We implemented additional data structures such as VecDeque, as well as memory wrappers such as Box and Clone.

Finally, we worked on creating a program to utilize our standard library and run it in BuzzOS. Before Joao created the userspace, we had developed a set of test cases and ran them in the kernel itself. However, this was definitely not ideal, since the goal is to run user programs in BuzzOS, and if we run a program in the kernel and it page faults, then the kernel will panic. While we were able to ensure that syscalls worked this way, we implemented a way to get a program running with our library in the *userspace*.

Appending a userspace to the end of an OS image is a bit of a tricky process. It can be done, however, and is often done when there is no existing file system in place. The process involves taking the userspace code, which is most likely in the form of an ELF binary, and appending it to the end of the OS image. This can be done through a variety of methods, such as manually compiling the userspace code and linking it to the kernel image, or using a script to perform the task automatically. Currently, we have that the userspace binary gets appended to the BuzzOS image and is ready to be read by the kernel and interpret the ELF.

6 Lessons Learned

- We learned how syscalls are implemented and handled within the kernel since our project is heavily dependent on them. Specifically, we learned about how trap handling worked in x86 and how to incorporate those syscalls in our library functions in order to provide the user the ability to use such functions just as they would with Rust's standard library.
- Our original scope for the project was only to implement a standard library in BuzzOS, but the lack of a file system and user space upon starting development proved to be a lot more challenging than expected. Each step of the process to support our library functions would typically rely on these systems to be in place in order to properly define syscalls, use them correctly in the context of a standard library, and execute functions in a user space program. Having to work around these limitations was challenging but also enlightening for us to discover different design choices within the kernel to implement the same features with a file system and userspace.

7 Reflections

While developing the project, we discovered several things:

- Rust had many surprising features we weren't aware of beforehand, some beneficial but most tedious to work with. The safety it provides during compilation is helpful for determining areas of concern in our code that could be unsafe and result in segfaults, page faults, etc. However, it's strict type checking, unfamiliar syntax, and finicky build process for our custom standard library made it difficult for us to properly compile, test, and push our changes.
- Our syscall implementations worked well in spite of the frequent changes made to the repository. The main functionality of our handlers ported was easily ported over to the new functions created to handle traps and, by extension, syscalls. Having an existing framework for memory management made it significantly more convenient to write functions that relied on it and test them.
- Since it took us quite some time to fully support our initial library functions including the additional functionality that Joao contributed, we weren't able to complete all the functions we initially planned to support. This was likely due to unrealistic goals we had set prior to starting the project without a realistic idea of what our timeline would look like.

8 Challenges

Our team faced several roadblocks during the project's development cycle, namely syscall support, unsafe kernel interaction, and updating functionality:

- We initially had no access to any syscalls crucial for defining our standard library's modules, and this proved difficult at first to work around. Using the approach listed in our Design and Implementation section, we managed to create our own wrapper for executing syscall handlers for `sbrk, read, and write` so that we could support memory allocation and console read/write.
- We also had issues with calling unsafe parts of the kernel code, especially when implementing our `sbrk` handler to write our `Vec` and `Box` functions. To account for this unsafe and potentially unknown behavior, we added error handlers that printed warnings to the console.
- Rebasing our work with Joao's updated functionality proved to be a consistent challenge since new changes were being made on a consistent basis towards the latter half of our project timeline, with

the main, notable features being the addition of a user space, scheduler, and trap handling. We had to raise several pull requests to fix these merge conflicts and incorporate these new features with our existing functionality to make our user space program run properly.

9 Future Work

Since our project was a proof-of-concept, there are several things we can expand on and add to the existing functionality:

- Match the full behavior of the current rust standard library with our custom one.
- Implement additional syscalls to handle this complete behavior (e.g. mmap, open, etc.).
- Incorporate a file system into BuzzOS and refactor the existing I/O syscalls around this.
- Add additional library packages that weren't included in time (e.g. Option Monad, BTreeSet, etc.).