Rust without libc

Paul Römer

What

- Rust with no `libc` → no `std`
- only x86_64 linux
- open ended deep dive into more complex parts if there is interest

Why

- please don’t do this in production
- learning about everything in between userspace Rust code and the linux kernel
- learning about low level Rust

How

Basic Hello World in Rust

```rust
fn main() {
    println!("Hello, World!");
}
```

```
> ldd target/debug/hello
    linux-vdso.so.1 (0x00007ffca67e3000)
    libgcc_s.so.1 => /usr/lib/libgcc_s.so.1 (0x00007ff2f8439000)
    libpthread.so.0 => /usr/lib/libpthread.so.0 (0x00007ff2f8418000)
    libdl.so.2 => /usr/lib/libdl.so.2 (0x00007ff2f8411000)
    libc.so.6 => /usr/lib/libc.so.6 (0x00007ff2f8244000)
    /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007ff2f84ba000)
```

Disabling std

Let’s tell the compiler we don’t want the standard library

```
#![no_std]
```

Now it doesn’t work anymore

```
error: cannot find macro `println` in this scope
 --> src/main.rs:5:5
  |
5 |     println!("Hello, world!");
  |     ^^^^^^^

error: language item required, but not found: `eh_personality`

error: `#[panic_handler]` function required, but not found

error: aborting due to 3 previous errors
```

panic_handler

```
error: `#[panic_handler]` function required, but not found
```

- We just removed Rust’s ability to write to `stderr`
- Rust doesn’t know what to do if we panic

panic_handler implementation

For now we also can't write to stderr, so if we panic, just abort the process:

```
#![feature(asm)]

/// Abort the process with "illegal hardware instruction"
fn abort() -> ! {
    unsafe { asm!("ud2", options(noreturn, nostack)) }
}

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

eh_personality

```
error: language item required, but not found: `eh_personality`
```

- Used for stack unwinding
- We never unwind the stack

Let's just abort if it is ever called

```
#![feature(lang_items)]

#[lang = "eh_personality"] extern fn rust_eh_personality() {
    abort();
}
```

we don’t know how to write to stdout

```
error: cannot find macro `println` in this scope
```

- `println!` is defined in std
- we still can’t actually write to stdout

For now let’s just remove it and see if our code compiles:

```
fn main() {}
```

It doesn’t

```
error: requires `start` lang_item
```

the `start` lang_item

```
error: requires `start` lang_item
```

The compiler needs to know where our program starts now that we don't have a `std` main anymore.

We can either use the (unstable) `#[start]` to specify the start function:

```
#[start] fn start(_argc: isize, _argv: *const *const u8) -> isize {
    42
}
```

Or we can just export a C `main` function and let the libc shim the compiler includes call it.

```
#[no_mangle] pub extern fn main(_argc: isize, _argv: *const *const u8) -> isize {
    42
}
```

Did I hear libc

Wait a minute…

Libc shim?

```
> man gcc | grep -A4 nostartfiles
- nostartfiles
    Do not use the standard system startup files when linking.
    The standard system libraries are used normally,
    unless -nostdlib, -nolibc, or -nodefaultlibs is used.
```

Let's remove that:

```
#![feature(link_args)]

#[link_args = "-nostartfiles"]
extern "C" {}
```

We're still missing an entrypoint

First let's tell `rustc` that we want to write our own `main`

```
#![no_main]
```

And now let's export our own `_start`

```
#[no_mangle] pub extern "C" fn _start() {}
```

So if it compiles it works, right?

```
> cargo r
[1]    346680 segmentation fault (core dumped)  cargo r
```

It turns out those `startfiles` actually did important things.

So what is happening?

- the linux kernel starts executing our program at `_start`
- our `_start` function does nothing and returns to the caller
- it does this by popping a value from the stack and jumping to it
- the value currently at the top of the stack is the number of arguments
- we jump to `0x1`

Let’s write a correct `_start`

Out start function needs to do a few things:

- not touch the (possibly unaligned) stack
- save the number of arguments and a pointer to the arguments
- align the stack
- call an _init function which must never return

preventing _start from touching the stack

Working on an unaligned stack

- normal Rust functions can and do arbitrarily touch the stack
- stack may not actually be aligned to acessing it would be UB
- our `_start` function thus needs to be pure assembly

Enter `#[naked]` functions

- experimental feature (`#![naked_functions]`)
- allow you to write all of the contents of the function
- need to contain a single `asm` block with the `noreturn` option

a working _start function

```
#![feature(naked_functions)]

unsafe extern "C" fn _init(_n_args: usize, _args_start: *const *const u8) -> ! {
    abort()
}

#[no_mangle] #[naked] unsafe extern "C" fn _start() {
    // C call: rdi, rsi, rdx, rcx, r8, r9
    asm!(
        "endbr64",

        // clear base pointer
        "xor rbp, rbp",

        // pop n_args into rdi (arg1)
        "pop rdi",

        // mov argument start pointer to start of args to rsi (arg2)
        "mov rsi, rsp",

        // align the stack pointer to multiples of 16
        "and rsp, 0xfffffffffffffff0",

        // call _init
        "call {}", sym _init,

        options(noreturn)
    );
}
```

It works!

```
> cargo r
[1]    356221 illegal hardware instruction (core dumped)  cargo r
```

No more dynamic dependencies

```
> ldd target/debug/hello
    statically linked
```

Questions

Hello World?

So how do we actually interact with the rest of the World?

Syscalls

- Syscalls are the main way for user programs to interface with the kernel
- more expensive than normal calls because they involve a context switch
- usually called through libc's `unistd.h`
- very machine dependent

How do we perform syscalls?

Syscalls are basically the same as normal calls with some exceptions

1. Syscall parameters are passed in `rdi`, `rsi`, `rdx`, `r10`, `r8` and `r9`.
2. Syscalls can't pass arguments through the stack
3. A syscall is done via the `syscall` instruction instead of `call`.
4. The syscall to execute is passed as a number in `rax` instead of as a pointer argument to `call`.
5. As with normal "C" functions, the result of the syscall is returned in `rax`.
6. Syscalls clobber `rcx` and `r11`.

For more details see Page 124 of Source 1

The exit syscall

```
> man 2 exit
SYNOPSIS
    noreturn void _exit(int status);

DESCRIPTION
    _exit()  terminates the calling process "immediately".
```

Implementing the exit syscall

```
pub fn sys_exit(status: i32) -> ! {
    const SYS_NO_EXIT: usize = 60;

    unsafe {
        asm!(
            "syscall",
            in("rdi") status,
            in("rax") SYS_NO_EXIT,
            options(nostack, noreturn)
        );
    }
}
```

```
hell::sys_exit:
    mov     eax, 60
    syscall
```

finally a proper `main` function

Now that we can actually exit the process, let's change _init:

```
unsafe extern "C" fn _init(n_args: usize, args_start: *const *const u8) -> ! {
    sys_exit(main(n_args, args_start) as i32)
}
```

and a proper `main` function

```
fn main(n_args: usize, args_start: *const *const u8) -> i8 {
    42
}
```

And...

```
> cargo r
> echo $?
42
```

success

The write syscall

```
> man 2 write
SYNOPSIS
    ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
    write()  writes up to `count` bytes from the buffer starting at `buf`
    to the file referred to by the file descriptor `fd`.
```

Implementing the write syscall

```
pub unsafe fn sys_write(fd: u32, buf: *const u8, count: usize) -> isize {
    const SYS_NO_WRITE: usize = 1;

    let ret: isize;
    asm!(
        "syscall",
        in("rdi") fd,
        in("rsi") buf,
        in("rdx") count,
        inout("rax") SYS_NO_WRITE => ret,
        out("rcx") _,
        out("r11") _,
        options(nostack)
    );
    ret
}
```

```
example::sys_write:
    mov     eax, 1
    syscall
    ret
```

Hello World

```
fn main() -> i8 {
    print_str("Hello, World!\n");
    42
}
fn print_str(s: &str) {
    unsafe {
        let res = sys_write(1, s.as_ptr(), s.len());

        if res < 0 {
            abort();
        }
    }
}
```

```
> cargo r
Hello, World!
```

Questions

Sources

1. System V Application Binary Interface - AMD64 Architecture Processor Supplement
2. Linux System Call Table for x86 64
3. My experiments with barebones x86 64
4. The Rustonomicon

Open-end deep dive

Things I have implemented that we could take a look at:

- allocator
- mimicking std I/O
- threading
- synchronisation (Mutex)
- stack overflow detection
- thread local storage

title: Rust without libc author: Paul Römer theme: Berlin colortheme: beaver pandoc-latex-fontsize:

- classes: [nasm] size: tiny
- classes: [rust] size: tiny
- classes: [bash] size: tiny

What

- Rust with no `libc` → no `std`
- only x86_64 linux
- open ended deep dive into more complex parts if there is interest

Why

- please don't do this in production
- learning about everything in between userspace Rust code and the linux kernel
- learning about low level Rust

How

Basic Hello World in Rust

```
fn main() {
    println!("Hello, World!");
}
```

```
> ldd target/debug/hello
    linux-vdso.so.1 (0x00007ffca67e3000)
    libgcc_s.so.1 => /usr/lib/libgcc_s.so.1 (0x00007ff2f8439000)
    libpthread.so.0 => /usr/lib/libpthread.so.0 (0x00007ff2f8418000)
    libdl.so.2 => /usr/lib/libdl.so.2 (0x00007ff2f8411000)
    libc.so.6 => /usr/lib/libc.so.6 (0x00007ff2f8244000)
    /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007ff2f84ba000)
```

Disabling std

Let's tell the compiler we don't want the standard library

```
#![no_std]
```

Now it doesn't work anymore

```
error: cannot find macro `println` in this scope
 --> src/main.rs:5:5
  |
5 |     println!("Hello, world!");
  |     ^^^^^^^

error: language item required, but not found: `eh_personality`

error: `#[panic_handler]` function required, but not found

error: aborting due to 3 previous errors
```

panic_handler

```
error: `#[panic_handler]` function required, but not found
```

- We just removed Rust's ability to write to `stderr`
- Rust doesn't know what to do if we panic

panic_handler implementation

For now we also can’t write to stderr, so if we panic, just abort the process:

```rust
#![feature(asm)]

/// Abort the process with "illegal hardware instruction"
fn abort() -> ! {
    unsafe { asm!("ud2", options(noreturn, nostack)) }
}

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

eh_personality

```
error: language item required, but not found: `eh_personality`
```

- Used for stack unwinding
- We never unwind the stack

Let’s just abort if it is ever called

```
#![feature(lang_items)]

#[lang = "eh_personality"] extern fn rust_eh_personality() {
    abort();
}
```

we don’t know how to write to stdout

```
error: cannot find macro `println` in this scope
```

- `println!` is defined in std
- we still can’t actually write to stdout

For now let’s just remove it and see if our code compiles:

```
fn main() {}
```

It doesn’t

```
error: requires `start` lang_item
```

the start lang_item

```
error: requires `start` lang_item
```

The compiler needs to know where our program starts now that we don't have a `std` main anymore.

We can either use the (unstable) `#[start]` to specify the start function:

```
#[start] fn start(_argc: isize, _argv: *const *const u8) -> isize {
    42
}
```

Or we can just export a C `main` function and let the libc shim the compiler includes call it.

```
#[no_mangle] pub extern fn main(_argc: isize, _argv: *const *const u8) -> isize {
    42
}
```

Did I hear libc

Wait a minute...

Libc shim?

```
> man gcc | grep -A4 nostartfiles
- nostartfiles
    Do not use the standard system startup files when linking.
    The standard system libraries are used normally,
    unless -nostdlib, -nolibc, or -nodefaultlibs is used.
```

Let’s remove that:

```
#![feature(link_args)]

#[link_args = "-nostartfiles"]
extern "C" {}
```

We’re still missing an entrypoint

First let’s tell `rustc` that we want to write our own `main`

```
#![no_main]
```

And now let’s export our own `_start`

```
#[no_mangle] pub extern "C" fn _start() {}
```

So if it compiles it works, right?

```
> cargo r
[1]    346680 segmentation fault (core dumped)  cargo r
```

It turns out those `startfiles` actually did important things.

So what is happening?

- the linux kernel starts executing our program at `_start`
- our `_start` function does nothing and returns to the caller
- it does this by popping a value from the stack and jumping to it
- the value currently at the top of the stack is the number of arguments
- we jump to `0x1`

Let's write a correct `_start`

Out start function needs to do a few things:

- not touch the (possibly unaligned) stack
- save the number of arguments and a pointer to the arguments
- align the stack
- call an _init function which must never return

preventing _start from touching the stack

Working on an unaligned stack

- normal Rust functions can and do arbitrarily touch the stack
- stack may not actually be aligned to acessing it would be UB
- our `_start` function thus needs to be pure assembly

Enter `#[naked]` functions

- experimental feature (`#![naked_functions]`)
- allow you to write all of the contents of the function
- need to contain a single `asm` block with the `noreturn` option

a working _start function

```
#![feature(naked_functions)]

unsafe extern "C" fn _init(_n_args: usize, _args_start: *const *const u8) -> ! {
    abort()
}

#[no_mangle] #[naked] unsafe extern "C" fn _start() {
    // C call: rdi, rsi, rdx, rcx, r8, r9
    asm!(
        "endbr64",

        // clear base pointer
        "xor rbp, rbp",

        // pop n_args into rdi (arg1)
        "pop rdi",

        // mov argument start pointer to start of args to rsi (arg2)
        "mov rsi, rsp",

        // align the stack pointer to multiples of 16
        "and rsp, 0xfffffffffffffff0",

        // call _init
        "call {}", sym _init,

        options(noreturn)
    );
}
```

It works!

```
> cargo r
[1]    356221 illegal hardware instruction (core dumped)  cargo r
```

No more dynamic dependencies

```
> ldd target/debug/hello
    statically linked
```

Questions

Hello World?

So how do we actually interact with the rest of the World?

Syscalls

- Syscalls are the main way for user programs to interface with the kernel
- more expensive than normal calls because they involve a context switch
- usually called through libc's `unistd.h`
- very machine dependent

How do we perform syscalls?

Syscalls are basically the same as normal calls with some exceptions

1. Syscall parameters are passed in `rdi`, `rsi`, `rdx`, `r10`, `r8` and `r9`.
2. Syscalls can't pass arguments through the stack
3. A syscall is done via the `syscall` instruction instead of `call`.
4. The syscall to execute is passed as a number in `rax` instead of as a pointer argument to `call`.
5. As with normal "C" functions, the result of the syscall is returned in `rax`.
6. Syscalls clobber `rcx` and `r11`.

For more details see Page 124 of Source 1

The exit syscall

```
> man 2 exit
SYNOPSIS
    noreturn void _exit(int status);

DESCRIPTION
    _exit()  terminates the calling process "immediately".
```

Implementing the exit syscall

```
pub fn sys_exit(status: i32) -> ! {
    const SYS_NO_EXIT: usize = 60;

    unsafe {
        asm!(
            "syscall",
            in("rdi") status,
            in("rax") SYS_NO_EXIT,
            options(nostack, noreturn)
        );
    }
}
```

```
hell::sys_exit:
    mov     eax, 60
    syscall
```

finally a proper `main` function

Now that we can actually exit the process, let's change _init:

```
unsafe extern "C" fn _init(n_args: usize, args_start: *const *const u8) -> ! {
    sys_exit(main(n_args, args_start) as i32)
}
```

and a proper `main` function

```
fn main(n_args: usize, args_start: *const *const u8) -> i8 {
    42
}
```

And...

```
> cargo r
> echo $?
42
```

success

The write syscall

```
> man 2 write
SYNOPSIS
    ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
    write()  writes up to `count` bytes from the buffer starting at `buf`
    to the file referred to by the file descriptor `fd`.
```

Implementing the write syscall

```rust
pub unsafe fn sys_write(fd: u32, buf: *const u8, count: usize) -> isize {
    const SYS_NO_WRITE: usize = 1;

    let ret: isize;
    asm!(
        "syscall",
        in("rdi") fd,
        in("rsi") buf,
        in("rdx") count,
        inout("rax") SYS_NO_WRITE => ret,
        out("rcx") _,
        out("r11") _,
        options(nostack)
    );
    ret
}
```

```asm
example::sys_write:
    mov     eax, 1
    syscall
    ret
```

Hello World

```
fn main() -> i8 {
    print_str("Hello, World!\n");
    42
}
fn print_str(s: &str) {
    unsafe {
        let res = sys_write(1, s.as_ptr(), s.len());

        if res < 0 {
            abort();
        }
    }
}
```

```
> cargo r
Hello, World!
```

Questions

Sources

1. System V Application Binary Interface - AMD64 Architecture Processor Supplement
2. Linux System Call Table for x86 64
3. My experiments with barebones x86 64
4. The Rustonomicon

Open-end deep dive

Things I have implemented that we could take a look at:

- allocator
- mimicking std I/O
- threading
- synchronisation (Mutex)
- stack overflow detection
- thread local storage