

实验1 - 独立的可执行程序

21301021 肖斌

所用设备及系统: Macbook Pro M2 Max, MacOS Ventura 13.5.2

GitHub 仓库: <https://github.com/AzurIce/OperatingSystem-2023>

一、实验步骤

1. 创建 Rust 项目

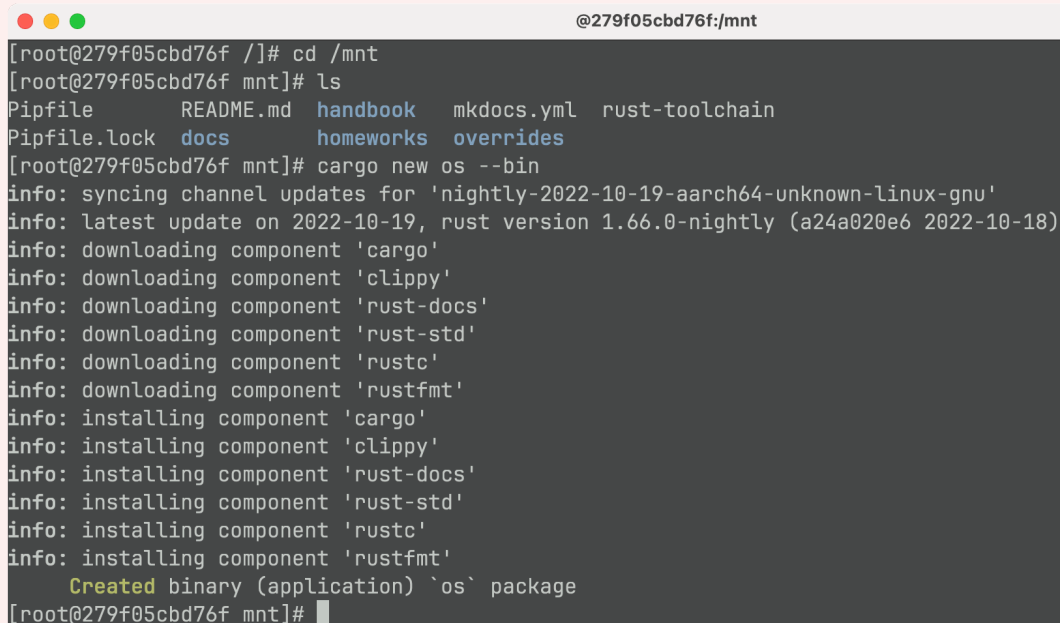
本实验的主要目的是构建一个独立的不依赖于 Rust 标准库的可执行程序。

首先进入到项目目录, 然后启动包含上一节配置好的环境的容器:

```
1 | docker run mystifying_kowalevski
2 | docker attach mystifying_kowalevski
```

进入 `/mnt` 目录, 并创建 Rust 项目:

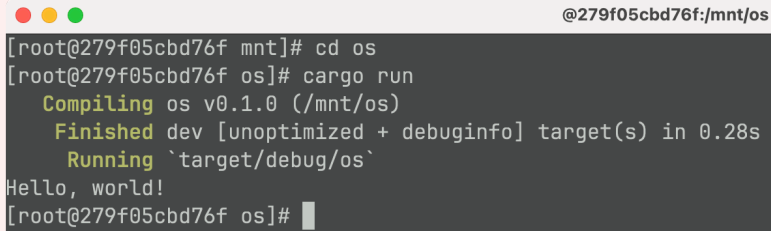
```
1 | cd /mnt
2 | cargo new os --bin
```



```
@279f05cbd76f:/mnt
[root@279f05cbd76f /]# cd /mnt
[root@279f05cbd76f mnt]# ls
Pipfile      README.md    handbook    mkdocs.yml  rust-toolchain
Pipfile.lock docs         homeworks   overrides
[root@279f05cbd76f mnt]# cargo new os --bin
info: syncing channel updates for 'nightly-2022-10-19-aarch64-unknown-linux-gnu'
info: latest update on 2022-10-19, rust version 1.66.0-nightly (a24a020e6 2022-10-18)
info: downloading component 'cargo'
info: downloading component 'clippy'
info: downloading component 'rust-docs'
info: downloading component 'rust-std'
info: downloading component 'rustc'
info: downloading component 'rustfmt'
info: installing component 'cargo'
info: installing component 'clippy'
info: installing component 'rust-docs'
info: installing component 'rust-std'
info: installing component 'rustc'
info: installing component 'rustfmt'
Created binary (application) `os` package
[root@279f05cbd76f mnt]#
```

运行查看结果:

```
1 | cd os
2 | cargo run
```



```
@279f05cbd76f:/mnt/os
[root@279f05cbd76f mnt]# cd os
[root@279f05cbd76f os]# cargo run
   Compiling os v0.1.0 (/mnt/os)
   Finished dev [unoptimized + debuginfo] target(s) in 0.28s
   Running `target/debug/os`
Hello, world!
[root@279f05cbd76f os]#
```

2. 移除标准库依赖

首先, 修改 target 为 riscv64, 在 `os/.cargo/` 目录下创建 `config` 文件, 并添加如下内容:

```
1 | # os/.cargo/config
2 | [build]
3 | target = "riscv64gc-unknown-none-elf"
```

然后修改 `main.rs`, 在开头加入如下内容, 并删除 `main` 函数:

```
1 | #![no_std]
2 | #![no_main]
```

同时, 因为标准库中提供了 `panic` 的处理函数 `#[panic_handler]` 所以我们还需要实现 `panic handler`, 添加如下内容:

```
1 | use core::panic::PanicInfo;
2 |
3 | #[panic_handler]
4 | fn panic(_info: &PanicInfo) → ! {
5 |     loop {}
6 | }
```

现在如果直接使用 `cargo build` 可能会出现编译错误:

```

@279f05cbd76f:/mnt/os
[root@279f05cbd76f os]# vim
[root@279f05cbd76f os]# cargo build
    Compiling os v0.1.0 (/mnt/os)
error[E0463]: can't find crate for `core`
|
= note: the `riscv64gc-unknown-none-elf` target may not be installed
= help: consider downloading the target with `rustup target add riscv64gc-unknown-none-elf`
= help: consider building the standard library from source with `cargo build -Zbuild-std`

error[E0463]: can't find crate for `compiler_builtins`

error[E0463]: can't find crate for `core`
--> src/main.rs:4:5
|
4 | use core::panic::PanicInfo;
|     ^^^^^ can't find crate
|
= note: the `riscv64gc-unknown-none-elf` target may not be installed
= help: consider downloading the target with `rustup target add riscv64gc-unknown-none-elf`
= help: consider building the standard library from source with `cargo build -Zbuild-std`

error: requires `sized` lang_item

For more information about this error, try `rustc --explain E0463`.
error: could not compile `os` due to 4 previous errors
[root@279f05cbd76f os]#

```

需要执行如下命令添加相关软件包:

```

1 | rustup target add riscv64gc-unknown-none-elf
2 | cargo install cargo-binutils
3 | rustup component add llvm-tools-preview
4 | rustup component add rust-src

```

再进行构建:

```

[root@279f05cbd76f os]# cargo build
    Compiling os v0.1.0 (/mnt/os)
    Finished dev [unoptimized + debuginfo] target(s) in 0.06s

```

然后可以对独立的可执行程序进行分析:

```

1 | file target/riscv64gc-unknown-none-elf/debug/os
2 | rust-readobj -h target/riscv64gc-unknown-none-elf/debug/os
3 | rust-objdump -S target/riscv64gc-unknown-none-elf/debug/os

```

```
@279f05cbd76f:/mnt/os
[root@279f05cbd76f os]# file target/riscv64gc-unknown-none-elf/debug/os
target/riscv64gc-unknown-none-elf/debug/os: ELF 64-bit LSB executable, UCB RISC-V, RVC, double-float ABI,
version 1 (SYSV), statically linked, with debug_info, not stripped
[root@279f05cbd76f os]# rust-readobj -h target/riscv64gc-unknown-none-elf/debug/os

File: target/riscv64gc-unknown-none-elf/debug/os
Format: elf64-littleriscv
Arch: riscv64
AddressSize: 64bit
LoadName: <Not found>
ElfHeader {
  Ident {
    Magic: (7F 45 4C 46)
    Class: 64-bit (0x2)
    DataEncoding: LittleEndian (0x1)
    FileVersion: 1
    OS/ABI: SystemV (0x0)
    ABIVersion: 0
    Unused: (00 00 00 00 00 00 00)
  }
  Type: Executable (0x2)
  Machine: EM_RISCV (0xF3)
  Version: 1
  Entry: 0x0
  ProgramHeaderOffset: 0x40
  SectionHeaderOffset: 0x1AF0
  Flags [ (0x5)
    EF_RISCV_FLOAT_ABI_DOUBLE (0x4)
    EF_RISCV_RVC (0x1)
  ]
  HeaderSize: 64
  ProgramHeaderEntrySize: 56
  ProgramHeaderCount: 3
  SectionHeaderEntrySize: 64
  SectionHeaderCount: 14
  StringTableSectionIndex: 12
}
[root@279f05cbd76f os]# rust-objdump -S target/riscv64gc-unknown-none-elf/debug/os

target/riscv64gc-unknown-none-elf/debug/os:      file format elf64-littleriscv
[root@279f05cbd76f os]# █
```

分析可以发现编译生成的二进制程序是一个空程序，这是因为编译器找不到入口函数，所以没有生成后续的代码。

3. 用户态可执行的环境

1> 实现入口函数

首先增加入口函数：

```
1 | #[no_mangle]
2 | extern "C" fn _start() {
3 |     loop{};
4 | }
```

Rust 编译器要找的入口函数为 `_start()`。

然后重新编译。

通过如下命令可以执行编译生成的程序：

```
1 | qemu-riscv64 target/riscv64gc-unknown-none-elf/debug/os
```

可以发现似乎是在执行一个死循环，即程序无输出，也不结束：

```
[root@279f05cbd76f os]# cargo build
    Compiling os v0.1.0 (/mnt/os)
    Finished dev [unoptimized + debuginfo] target(s) in 0.06s
[root@279f05cbd76f os]# qemu-riscv64 target/riscv64gc-unknown-none-elf/debug/os
^C
```

2> 实现退出机制

添加如下代码：

```
1 | #![no_std]
2 | #![no_main]
3 |
4 | use core::panic::PanicInfo;
5 | +use core::arch::asm;
6 | +
7 | +const SYSCALL_EXIT: usize = 93;
8 | +
9 | +fn syscall(id: usize, args: [usize; 3]) → isize {
10 | +    let mut ret: isize;
11 | +    unsafe {
12 | +        asm!("ecall",
13 | +            in("x10") args[0],
14 | +            in("x11") args[1],
15 | +            in("x12") args[2],
16 | +            in("x17") id,
17 | +            lateout("x10") ret
18 | +        );
19 | +    }
20 | +    ret
```

```

21 +}
22 +
23 +pub fn sys_exit(xstate: i32) → isize {
24 +     syscall(SYSCALL_EXIT, [xstate as usize, 0, 0])
25 +}
26
27 #[panic_handler]
28 fn panic(_info: &PanicInfo) → ! {
29     loop {}
30 }
31
32 #[no_mangle]
33 extern "C" fn _start() {
34 -     loop{};
35 +     sys_exit(9);
36 }

```

再编译运行，发现程序可以直接正常退出：

```

[root@279f05cbd76f os]# cargo build
    Compiling os v0.1.0 (/mnt/os)
    Finished dev [unoptimized + debuginfo] target(s) in 0.06s
[root@279f05cbd76f os]# qemu-riscv64 target/riscv64gc-unknown-none-elf/debug/os
[root@279f05cbd76f os]#

```

3> 实现输出支持

首先封装一下对 Linux 操作系统内核提供的系统调用 `SYSCALL_WRITE`：

```

1  const SYSCALL_WRITE: usize = 64;
2
3  pub fn sys_write(fd: usize, buffer: &[u8]) → isize {
4      syscall(SYSCALL_WRITE, [fd, buffer.as_ptr() as usize, buffer.len()])
5  }

```

然后声明一个 `Stdout` 结构体并为其实现 `Write Trait`：

```

1  struct Stdout;
2
3  impl Write for Stdout {
4      fn write_str(&mut self, s: &str) → fmt::Result {
5          sys_write(1, s.as_bytes());
6          Ok(())
7      }
8  }
9
10 pub fn print(args: fmt::Arguments) {
11     Stdout.write_fmt(args).unwrap();
12 }

```

然后给予 `print` 函数，实现本存在于 Rust 语言标准库中的的输出宏 `print!` 和 `println!`：

```
1 use core::fmt::{self, Write};
2
3 #[macro_export]
4 macro_rules! print {
5     ($fmt: literal $(, $($arg: tt)+)?) => {
6         $crate::console::print(format_args!($fmt $(, $($arg)+)??));
7     }
8 }
9
10 #[macro_export]
11 macro_rules! println {
12     ($fmt: literal $(, $($arg: tt)+)?) => {
13         print(format_args!(concat!($fmt, "\n") $(, $($arg)+)??));
14     }
15 }
```

然后我们可以在入口函数 `_start` 中使用我们实现的 `println!` 宏打印 `Hello, world!`。

完整 diff 如下：

```
1 -- a/os/src/main.rs
2 +++ b/os/src/main.rs
3 @@ -4,6 +4,46 @@
4     use core::panic::PanicInfo;
5     use core::arch::asm;
6
7 +// Wrap of SYSCALL_WRITE
8 +const SYSCALL_WRITE: usize = 64;
9 +
10 +pub fn sys_write(fd: usize, buffer: &[u8]) -> isize {
11 +    syscall(SYSCALL_WRITE, [fd, buffer.as_ptr() as usize, buffer.len()])
12 +}
13 +
14 +// Implement Write trait for Stdout
15 +struct Stdout;
16 +
17 +impl Write for Stdout {
18 +    fn write_str(&mut self, s: &str) -> fmt::Result {
19 +        sys_write(1, s.as_bytes());
20 +        Ok(())
21 +    }
22 +}
23 +
24 +pub fn print(args: fmt::Arguments) {
25 +    Stdout.write_fmt(args).unwrap();
26 +}
27 +
28 +// Implement print macro
```

```

29 +use core::fmt::{self, Write};
30 +
31 +#[macro_export]
32 +macro_rules! print {
33 +    ($fmt: literal $(, $($arg: tt)+)?) => {
34 +        $crate::console::print(format_args!($fmt $(, $($arg)+)?));
35 +    }
36 +}
37 +
38 +#[macro_export]
39 +macro_rules! println {
40 +    ($fmt: literal $(, $($arg: tt)+)?) => {
41 +        print(format_args!(concat!($fmt, "\n") $(, $($arg)+)?));
42 +    }
43 +}
44 +
45 +
46 +// Wrap of SYSCALL_EXIT
47 +const SYSCALL_EXIT: usize = 93;
48 +
49 +fn syscall(id: usize, args: [usize; 3]) -> isize {
50 @@ -31,5 +71,6 @@ fn panic(_info: &PanicInfo) -> ! {
51
52 +#[no_mangle]
53 +extern "C" fn _start() {
54 +    println!("Hello, world!");
55 +    sys_exit(9);
56 +}

```

编译运行：

```

[root@279f05cbd76f os]# cargo build
    Compiling os v0.1.0 (/mnt/os)
    Finished dev [unoptimized + debuginfo] target(s) in 0.07s
[root@279f05cbd76f os]# qemu-riscv64 target/riscv64gc-unknown-none-elf/debug/os
Hello, world!

```

二、思考问题

1. 为什么称最后实现的程序为独立的可执行程序，它和标准的程序有什么区别？

标准的程序中包含了标准库中实现的一系列内容，可以理解为最终的可执行文件中还包含标准库中的代码。

之所以称最后实现的程序为 *独立的可执行程序*，就是因为它并不依赖于标准库，而是完全通过操作系统内核提供的系统调用来实现一系列功能。

比如 *退出程序* 和 *输出宏*。

`print!` 和 `println!` 在标准库中是有实现的，其实在标准库中的底层部分也是通过 *系统调用* 来实现的，只不过可能有针对各种不同平台的详细代码，比如在 Linux 下通过 Linux 的系统调用实现，在 Windows 下通过 Windows 的 API 实现，等等。

2. 实现和编译独立可执行程序的目的是什么？

其实这一次实验相当于从底层，从最基本的积木「系统调用」来与系统交互，搭出了「标准库」的冰山一角。后面的实验可能会要我们手动去实现一个操作系统，而编程语言的标准库中是不包含对我们自己实现的操作系统的具体实现的，所以我们不可能依赖标准库来写我们的代码，只能同样用最基本的积木来与我们自己的操作系统进行交互。

三、Git 提交截图

Commits

main

Commits on Oct 20, 2023

experiment2 by 21301021 Xiao Bin Azurice committed 42 minutes ago ✓	22ac30a	<>
exp2: implemented printing Azurice committed 43 minutes ago	3f2c9b7	<>
exp2: implemented exit mechanism Azurice committed 1 hour ago	fa8e404	<>
exp2: implemented entrance function Azurice committed 1 hour ago	8affc09	<>
exp2: remove dep on std Azurice committed 1 hour ago	7cf6dc5	<>
exp2: Init project Azurice committed 1 hour ago	833e80e	<>
Update main.yml Azurice committed 1 hour ago ✓	27e19fc	<>
added experiment handbook Azurice committed 1 hour ago	9e6564c	<>
added .DS_Store to ignore file Azurice committed 1 hour ago	a5f9322	<>

四、其他说明

做这次实验之余简单看了一下整体的实验内容与脉络，感觉内容很有意思。

我认为我比较幸运，因为我个人在平时是比较喜欢折腾各种技术之类的小东西的，所以对 Linux、Rust、Git、vim 等一系列东西都有一定的理解，也因此能够从实验中获得更多的收获。

最初一次接触 Linux 可能是小学四五年级的时候奔着「国产操作系统」摸索着给自己的笔记本装了个 KyLin，不过当时什么也不知道，简单吧完了两下便换回了 Linux。后面从打 OI 使用 NOI-Linux、自己使用 Ubuntu 来熟悉，到后面自己开始使用、折腾 Linux 系统 (Manjaro Linux → Arch Linux → NixOS)，因此能对 Linux 及相关的工具有一定的理解。

我自己也很乐于探索新兴的编程语言，同时在自己的小项目里做各种尝试，于是从最初因打 OI 了解的 C/C++ 到 Python、Java、前端的那套东西和各种框架、Golang、Rust，也因此对 Rust 有一定的理解。

但是有很多人并不像我一样幸运：

舍友因为将 `.cargo` 目录创建为 `cargo`，而导致 cargo 读取不到正确的 `config` 文件，进而无法编译。

我说：「你 cat 一下 `.cargo` 里的 `config` 我看看」

舍友：「啊？」「cat？」

我说：「cat, 空格」

舍友：艰难地用着 vim

我：「你可以用 VSCode 在本机编辑，要运行编译哪些命令的时候再到 docker 里运行」

舍友：「哦哦哦哦哦哦哦哦哦哦！！！！」

舍友：「我发现我都做完了，但是我发现 GardenerOS 里啥都没有」

我：「...」

那么我想，「标准库是什么」、「extern "C" 是啥」、「Trait 是什么」、「asm!在做什么」、「宏是什么」等等这些问题，或许更不可能让大家理解。对于很多人来说，机械性地复制-粘贴手册中的指令，对着各种小问题焦头烂额，“碰运气”式地得到了和手册中描述相同的结果便已经皆大欢喜。

但是，似乎也没有什么更好的解决办法，很多人接触相关内容的时间本就很短。这个领域的信息量实在是十分庞大，短时间内将它们全部接受、吸收近乎不可能。

突然有些感悟，简单写了下（）