

操作系统课程实验报告

rcore 第一章

陈林峰 1120192739

2021 年 11 月 25 日

1 实验环境

1. OS: Ubuntu 20.04 focal(on the Windows Subsystem for Linux)
2. Kernel: x86_64 Linux 5.10.16.3-microsoft-standard-WSL2
3. Rust: rustc 1.58.0-nightly (936f2600b 2021-11-22)

2 实验目的

将应用程序与计算机硬件隔离，移除标准库依赖，实现在屏幕上打印字符。此时的操作系统作用类似于一个函数库，为应用程序提供最简单的服务。

3 实验步骤

1. 将应用程序依赖的标准库去掉，编译出不需要标准库的应用程序。
2. 实现打印功能，可以运行在 qemu 模拟的 risc-v 虚拟系统上面
3. 调整内核代码布局，设置相应的栈，实现在裸机上打印字符

4 实验内容

4.1 目标平台和三元组

对于程序源代码而言，编译器在将其通过编译、链接得到可执行文件的时候需要知道程序要在哪个平台 (Platform) 上运行。这里平台主要是指 CPU 类型、操作系统类型和标准运行时库的组合。

目标三元组 (Target Triplet) 用来描述一个目标平台。

host: x86_64-unknown-linux-gnu

cpu 架构: x86-64,cpu 厂商: unknow, 操作系统: linux, 运行时库: gnu libc

Rust 编译器支持下面的基于 RISC-V 的平台

riscv32gc-unknown-linux-gnu

riscv32gc-unknown-linux-musl

riscv32i-unknown-none-elf

```
riscv32imac-unknown-none-elf
riscv32imc-esp-espidf
riscv32imc-unknown-none-elf
riscv64gc-unknown-linux-gnu
riscv64gc-unknown-linux-musl
riscv64gc-unknown-none-elf
riscv64imac-unknown-none-elf
```

选择 `riscv64gc-unknown-none-elf`: 没有操作系统支持, `elf` 表示没有标准的运行时库 (表明没有任何系统调用的封装支持), 可以生成 `elf` 格式的文件。

4.2 移除标准库支持

现在操作系统上的应用程序运行需要多个层次的执行环境的支持。

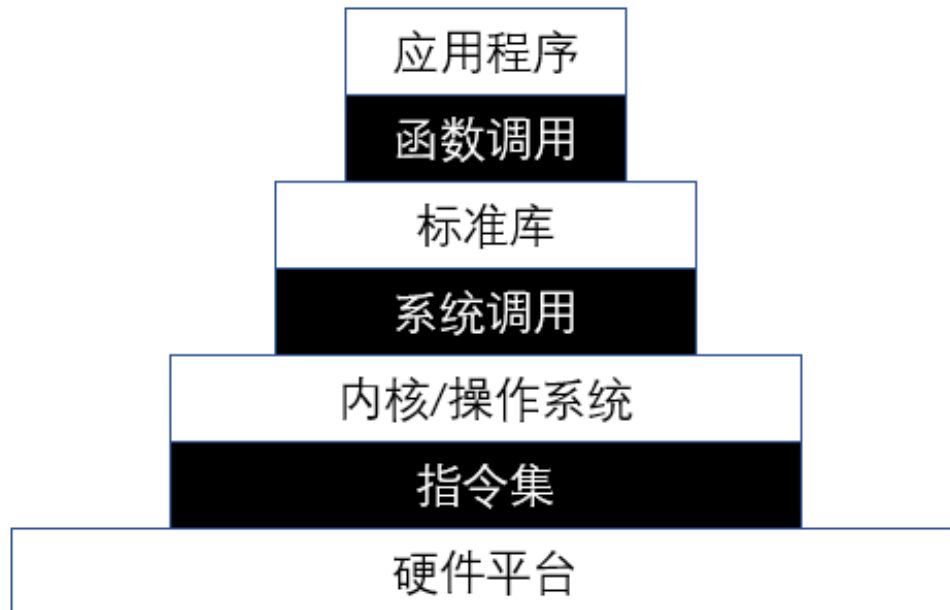


图 1: 计算机系统各层执行环境

当我们在写出下述代码时:

```
fn main(){
    println!("hello world");
}
```

`println!` 宏所在的 Rust 标准库 `std` 需要通过系统调用获得操作系统的服务。但此时我们没有操作系统没有系统调用, 因此需要去掉这个宏。

```
#![no_std]
fn main(){}
```

Rust 的标准库 `std`, 为绝大多数的 Rust 应用程序开发提供基础支持、跨硬件和操作系统平台支持, 是应用范围最广、地位最重要的库, 但需要有底层操作系统的支持。

Rust 的核心库 core，可以理解为是经过大幅精简的标准库，它被应用在标准库不能覆盖到的某些特定领域，如裸机 (bare metal) 环境下，用于操作系统和嵌入式系统的开发，它不需要底层操作系统的支持

我们不能使用 Rust 的标准库，但可以使用核心库，核心库不依赖于操作系统移除掉标准库后，通过在配置文件指定目标平台为上述我们选择的目标平台后，此时再使用 cargo run 运行会产生下列错误

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

核心库中不存在这个语义项的实现，因此我们需要实现它

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

加入此语义项再运行 cargo run 会产生下列错误

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

这里表示我们的程序缺少了 start 语义项，这个语义项定义了一个程序的入口。对于大多数语言来说，其第一个执行的函数并不是 main，而是其它的一些进行初始化的函数，对于一个链接了标准库的 rust 程序来说，这个程序首先会跳转到 C 语言的 crt0 运行时库，其用来设置允许 C 应用的环境，包括栈的分配和参数在寄存器的设置，最后 C 运行时触发 Rust 的运行时入口，这被标记为 start lang_item，当 rust 运行时完成任务后才会调用 main 函数进入主程序。

我们的独立式可执行程序并不能访问 Rust 运行时或 crt0 库，因为此时标准库已经被关闭，所以我们需要定义自己的入口点。实现一个 start 语言项并不能解决问题，因为这之后程序依然要求 crt0 库。所以，我们要做的是，直接重写整个 crt0 库和它定义的入口点。

```
#![no_std]
#![no_main]
mod lang_items;
//#[no_mangle]使用这个属性后，编译器就会将这个函数的名称编码为_start，否则就会编译成其他名字
#[no_mangle]
extern "C" fn _start(){
    loop{};
}
```

我们这里定义了一个 _start 函数，因为 _start 是 C 运行时的入口点，因此运行这个程序时会跳转到这个入口点开始执行，而我们标记了 no_main 则表明此时没有 main 函数了，因此 _start 也不会再跳转到 main 函数了。

QEMU 有两种运行模式：User mode 模式，即用户态模拟，如 qemu-riscv64 程序，能够模拟不同处理器的用户态指令的执行，并可以直接解析 ELF 可执行文件，加载运行那些为不同处理器编译的用户级 Linux 应用程序（ELF 可执行文件）；在翻译并执行不同应用程序中的不同处理器的指令时，如果碰到是系统调用相关的汇编指令，它会把不同处理器（如 RISC-V）的 Linux 系统调用转换为本地处理器（如 x86-64）上的 Linux 系统调用，这样就可以让本地 Linux 完成系统调用，并返回结果（再转换成 RISC-V 能识别的数据）给这些应用。System mode 模式，即系统态模式，如 qemu-system-riscv64 程序，能够模拟一个完整的基于不同 CPU

的硬件系统，包括处理器、内存及其他外部设备，支持运行完整的操作系统。

此时我们的程序虽然可以在 Qemu 的 user mode 模式下运行，但其却不能正常的退出，因为我们没有为其实现相应的退出机制，简单来说，就是还没有为其实现一个执行退出的系统调用。

4.3 系统调用

```
use core::fmt;
use core::fmt::Write;
//linux系统下的系统调用
const SYSCALL_EXIT:usize = 93;
const SYSCALL_WRITE:usize = 64;

#[warn(deprecated)]
fn syscall(id:usize,args:[usize;3])->isize{
    let mut ret: isize = 0;
    unsafe{//汇编指令
        llvm_asm!(
            "ecall"//调用中断指令
            :="{x10}"(ret) //输出操作数，只写操作数，位于x10寄存器
            :"{x10}"(args[0]),"{x11}"(args[1]),"{x12}"(args[2]),"{x17}"(id)
            //输入操作数
            : "memory" //代码将会改变内存的内容
            : "volatile" //禁止编译器对汇编程序进行优化
        );
        ret
    }
}

pub fn sys_exit(state:i32) ->isize{
    syscall(SYSCALL_EXIT,[state as usize,0,0])//执行退出
}

pub fn sys_write(fd:usize,buffer:&[u8])->isize{
    syscall(SYSCALL_WRITE,[fd,buffer.as_ptr() as usize,buffer.len()])//打印字符串
}

struct Stdout;
//为Stdout实现trait write_str
impl Write for Stdout{
    fn write_str(&mut self,s:&str)->fmt::Result{
        sys_write(1,s.as_bytes());
        Ok(())
    }
}

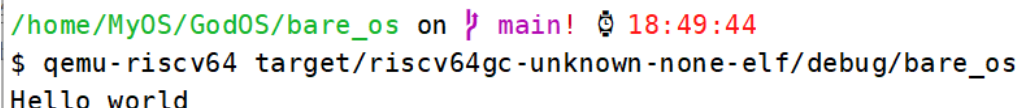
//实现了write_str的Stdout可以进一步实现格式化输出，这里使用的是rust提供的函数
pub fn print(args:fmt::Arguments){
    Stdout.write_fmt(args).unwrap();
}

//实现打印字符的宏
```

```
#[macro_export]
macro_rules! print {
    ($fmt:literal $(,$(arg:tt)+)?) => {
        $crate::print::print(format_args!($fmt $(,$(args)+)?));
    }
}

#[macro_export]
macro_rules! println {
    ($fmt:literal $(,$(arg:tt)+)?) => {
        $crate::print::print(format_args!(concat!($fmt, "\n") $(,$(arg)+)?));
    }
}
```

完成上述代码的编写，就可以在应用程序中使用打印字符的功能和正常退出，如下图所示



```
/home/MyOS/GodOS/bare_os on 🍯 main! 🕒 18:49:44
$ qemu-riscv64 target/riscv64gc-unknown-none-elf/debug/bare_os
Hello world
```

图 2: 打印字符

这里需要注意的是，这个程序是在 qemu 的 user mode 模式下运行，上面已经解释到这个模式会将代码的系统调用代码转为向本机进行系统调用，我们使用的 linux 操作系统符合上述的系统调用，因此会产生正常的打印和退出。

4.4 裸机打印

一个常用操作系统，其启动过程包括硬件初始化 (bootloader)，在硬件初始化后进入操作系统，操作系统开始接管整个机器的资源分配和向应用程序提供各种系统功能。

4.4.1 Qemu 硬件组成

外设：16550A UART, virtio-net/block/console/gpu 等和设备树

硬件特权级：priv v1.10, user v2.2

中断控制器：可参数化的 CLINT (核心本地中断器)、可参数化的 PLIC (平台级中断控制器)

可参数化的 RAM 内存

可配置的多核 RV64GC M/S/U mode CPU

在 QEMU 模拟的硬件中，物理内存和外设都是通过对内存读写的方式来进行访问。计算机的物理内存，DRAM 的内存起始地址是 0x80000000，缺省大小为 128MB。实验中限制为 8MB。

对于 QEMU 模拟的 risc-v64 计算机来说，当运行这台计算机时，其 CPU 的通用寄存器执行清零操作，此时 PC 寄存器指向的是 0x1000 的位置，这段指令是固化的，理论上来说这段指令会很快将 PC 指针指向 0x80000000 处，这里是 BootLoader 程序所在的位置，完成性格的硬件初始化后，PC 指针就会跳转到 0x80200000 处，这里是操作系统所在的位置，然后开始执行操作系统的内容。

4.4.2 内核的系统调用

在前面实现的系统调用中，都是操作系统为应用程序提供的服务，而操作系统又需要向实际硬件发出系统调用，真在落实到硬件上，因此这里需要完成的就是操作系统是如何为应用程序提供系统调用的。而为了不直接与硬件打交道，我们使用了 rustSBI 给我们提供的功能，这个 SBI 相当于给操作系统提供了操作硬件的功能，从而将操作系统与硬件隔离开。

```
///使用RustSBI接口进行相关操作

const SBI_CONSOLE_PUTCHAR: usize = 1;
const SBI_SHUTDOWN: usize = 8;

fn sbi_call(function: usize, arg0: usize, arg1: usize, arg2: usize) -> usize {
    let mut ret;
    unsafe {
        //汇编指令
        asm!(
            "ecall",//调用中断指令
            inlateout("x10") arg0 => ret, //输出操作数，只写操作数，位于x10寄存器
            in("x11") arg1,
            in("x12") arg2,
            in("x17") function,
            //输入操作数
            options(nostack)//代码将会改变内存的内容
            // : "volatile" //禁止编译器对汇编程序进行优化
        );
        ret
    }
}
```

这里可以看到与之前为应用程序提供的系统调用非常相似，但其向寄存器传入的参数是不一样的。而且两个代码执行的特权级也不一样。

4.4.3 内存布局

由于在裸机上执行时需要遵守 rustSBI 的规定，其在检查完机器后就会跳转到 0x80200000 地址处，这里应该放置操作系统的入口。因此需要将内核代码的布局进行调整。

```
OUTPUT_ARCH(riscv)/*指定目标平台*/
ENTRY(_start) /*程序入口*/
BASE_ADDRESS = 0X80200000; /* 基本变量，基准地址*/
/*描述输出文件的内存布局*/
SECTIONS
{
    . = BASE_ADDRESS;
    /*其中 . 表示当前地址，也就是链接器会从它指向的位置开始往下放置从输入的目标文件
    中收集来的段*/
    skernel = .;
```

```

stext = .; /*.text段的开始地址
/*
    冒号前面表示最终生成的可执行文件的一个段的名称，花括号内按照放置顺序
    描述将所有输入目标文件的哪些段放在这个段中
    每一行格式为 <ObjectFile>(SectionName)，表示目标文件 ObjectFile 的名为 SectionName
    的段需要被放进去。
    我们也可以使用通配符来书写 <ObjectFile> 和 <SectionName> 分别表示可能的输入目标文件和段名
*/

/*
    .text是所有的代码段
*/
.text : {
    *(.text.entry) /* 第一个是来自entry.asm的.text.entry
    *(.text .text.*)
}

. = ALIGN(4K);
etext = .; /*.text段的结束地址

/* 只读数据段，通常保存程序里面的常量
srodata = .;
.rodata : {
    *(.rodata .rodata.*)
}

. = ALIGN(4K);
erodata = .;

/*存放被初始化了的数据
sdata = .;
.data : {
    *(.data .data.*)
}

. = ALIGN(4K);
edata = .;

/*存放被初始化为 0 的可读写数据
.bss : {
    *(.bss.stack)
    sbss = .;
    *(.bss .bss.*)
}

. = ALIGN(4K);
ebss = .;
ekernel = .;

```

```

/ DISCARD : {
    *(.eh_frame)
}
}

.section .text.entry
.globl _start
_start:
    la sp, boot_stack_top //设置栈顶指针
    call rust_main

.section .bss.stack //预留一个64kb大小的空间作为运行程序的栈空间
.globl boot_stack
boot_stack://栈底地址
.space 4096 * 16
.globl boot_stack_top
boot_stack_top: //栈顶地址

```

上述第一段代码指定了内核代码起始的位置为 0x80200000, 其入口位置为 `_start`, 在 `.text` 代码段中, 又指定了最开始的代码为第二段代码定义的 `.text.entry`, 第二段代码定义了栈指针的位置, 为内核代码申请了栈空间, 这就相当于之前所说的 C 运行时库所作的初始化, 完成初始化后, 就会跳转到 `rust_main` 函数, 即现在 `rust_main` 函数代替了原来的 `main` 函数。因此可以从代码中删除之前定义的 `_start` 函数, 并重新定义内核代码入口 `rust_main`

```

global_asm!(include_str!("entry.asm"));

#[no_mangle]
pub fn rust_main() -> ! {
    println!("Godone's OS");
    println!("It's so nice");
    panic!("Stop");
}

```

此时的内核已经可以在裸机上执行。

5 其它功能

利用字符转义实现彩色化输出, 实现五个等级的输出。这里列出了 `ERROR`, `WARN` 两个等级的实现, 其它与之类似, 主要使用的是 `rust` 的 `cfg` 属性, 这个属性可以在编译时传入, 并影响不同代码的执行。

```

///彩色输出, 用于不同信息之间的分隔
///表示发生严重错误, 很可能或者已经导致程序崩溃
#[macro_export]
macro_rules! ERROR {
    () => ($crate::print!("\n"));
    ($($arg:tt)*) => {

```



```

        ($crate::print!("\x1b[31m{}\x1b[0m\n", format_args!($($arg)*)));
    }
}

// 表示发生不常见情况，但是并不一定导致系统错误
#[macro_export]
macro_rules! WARN {
    () => (
        #[cfg(any(feature = "WARN", feature = "INFO", feature = "DEBUG", feature = "TRACE"))]
        $crate::print!("\n");
        $crate::print!("");
    );
    ($($arg:tt)*) => {
        #[cfg(any(feature = "WARN", feature = "INFO", feature = "DEBUG", feature = "TRACE"))]
        ($crate::print!("\x1b[93m{}\x1b[0m\n", format_args!($($arg)*)));
        $crate::print!("");
    }
}

```

6 实验结果

```

[rustsbi] RustSBI version 0.2.0-alpha.1

RUSTSBI

[rustsbi] Platform: QEMU (Version 0.2.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000
Godone's OS
It's so nice
Panic'd at src/main.rs:34 Stop

/home/MyOS/GodOS/bare_os on main 17:53:07
$

```

图 3: 裸机打印字符并关机

```

[rustsbi] Implementation: RustSBI-QEMU Version 0.0.2
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: ssoft, stimer, sext (0x222)
[rustsbi] medeleg: lma, ia, bkpt, lb, sa, uecall, ipage, lpage, spage (0xb1ab)
[rustsbi] pmp0: 0x10000000 ..= 0x10001fff (rwx)
[rustsbi] pmp1: 0x80000000 ..= 0x8fffffff (rwx)
[rustsbi] pmp2: 0x0 ..= 0xffffffffffff (-..)
qemu-system-riscv64: clint: invalid write: 00000004
[rustsbi] enter supervisor 0x80200000
[kernel] Godone OS
Hello, world!
.text [0x80200000, 0x80203000)
.rodata [0x80203000, 0x80204000)
.data [0x80204000, 0x80204000)
.boot_stack [0x80204000, 0x80214000)
.bss [0x80214000, 0x80215000)
Panic'd at src/main.rs:65 The main_end!

/home/MyOS/GodOS/bare_os on Task! 18:47:42
$

```

图 4: 彩色打印