第一个实验

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

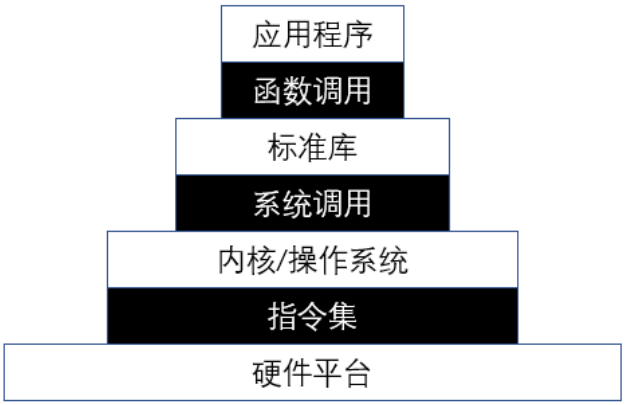
```
riscv32gc-unknown-linux-gnu
riscv32gc-unknown-linux-musl
riscv32i-unknown-none-elf
riscv32imac-unknown-none-elf
riscv32imc-esp-espidf
riscv32imc-unknown-none-elf
riscv64gc-unknown-linux-gnu
riscv64gc-unknown-linux-musl
riscv64gc-unknown-none-elf
riscv64imac-unknown-none-elf
```

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。

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。



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，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
```

语言标准库和三方库作为应用程序的执行环境，需要负责在执行应用程序之前进行一些初始化工作，然后才跳转到应用程序的入口点（也就是跳转到我们编写的`main`函数）开始执行。事实上`start`语义项代表了标准库 std 在执行应用程序之前需要进行的一些初始化工作。由于我们禁用了标准库，编译器也就找不到这项功能的实现了。

最粗暴的方式我们可以直接删除`main`函数并告诉编译器没有main函数即可。

注意

```
#注意，在不同目标平台上，cargo build产生的错误可能不一致
#在x86_64-unknown-linux-gnu平台上，会报下面的错误
error: language item required, but not found: `eh_personality`
```

`eh_personality`标记一个函数用于实现`stack unwinding`。默认情况下，当出现`panic`时，Rust使用unwinding调用所有stack上活动变量的destructors，以完成内容的释放，确保父线程catch panic异常并继续执行。Unwinding是个复杂的操作，并且依赖一些OS库支持，因为我们正在编写OS，因此这里不能使用Unwinding

关闭Unwinding的简单方法即在`cargo.toml`文件设置

```
[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"
```

上述设计起到的作用即时在panic时直接退出。

接下来，我们需要重新定义程序的入口，main函数并非rust的入口，大多数的编程语言都有一个运行时系统，其用来初始化一些设置，或者时垃圾回收(Java), 这个运行时系统在main函数之前运行。在rust中，二进制程序运行前会先调用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运行时会触发Rust的运行时入口，这被标记为 `start` lang_item, 当rust运行时完成任务后会调用main函数进入主程序。

我们不能直接去实现`start`，因为它需要C语言的运行时库，所以我们需要重新定义C运行时库`crt0`

```
#[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_std]
#![no_main]
mod lang_items;

#[no_mangle]
extern "C" fn _start(){
    // loop{};
}
```

来自Blog_OS教程的知识点

注意

```
目标平台的不同会导致一些其它的问题，因为我们已经在config文件设置了相关的目标平台，因此此处没有报错。
```

如果我们切换另一个目标平台，此时就会报下列的错误：

```
error: linking with `cc` failed: exit status: 1
```

```
链接器（linker）是一个程序，它将生成的目标文件组合为一个可执行文件。不同的操作系统如 Windows、macOS、Linux，规定了不同的可执行文件格式，因此也各有自己的链接器，抛出不同的错误；但这些错误的根本原因还是相同的：链接器的默认配置假定程序依赖于C语言的运行时环境，但我们的程序并不依赖于它。
```

解决此错误的方法就是像上述的操作添加config文件。或者在编译时使用参数`cargo build --target *`，需要指定一个没有底层操作系统的目标平台。

除了我们实验使用的`riscv64gc-unknown-none-elf`目标平台外，我们可以自定义自己的三元组。目标配置清单（target specification）

一个基于`x86_64-unknown-linux-gnu`目标系统的配置清单大概长这样：

```
{
    "llvm-target": "x86_64-unknown-linux-gnu",
    "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",
    "arch": "x86_64",
    "target-endian": "little",
    "target-pointer-width": "64",
    "target-c-int-width": "32",
    "os": "linux",
    "executables": true,
    "linker-flavor": "gcc",
    "pre-link-args": ["-m64"],//配置项指定了应该向链接器（linker）传入的参数。
    "morestack": false
}
```

一个配置清单中包含多个配置项（field）。大多数的配置项都是 LLVM 需求的，它们将配置为特定平台生成的代码

我们自定义的目标配置清单如下：

```
{
    "llvm-target": "x86_64-unknown-linux-gnu",
    "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",//定义了不同的整数、浮点数、指针类型的长度
    "arch": "x86_64",
    "target-endian": "little",
    "target-pointer-width": "64",//rust用作条件编译时的选项
    "target-c-int-width": "32",
    "os": "none",
    "executables": true,
    "linker-flavor": "ld.lld",
    "linker": "rust-lld",
    "panic-strategy": "abort",
    "disable-redzone": true,
    "features": "-mmx,-sse,+soft-float"
}
```

继续r-Core

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系统调用转换为本机处理器（如x86-64）上的Linux系统调用，这样就可以让本机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。

实现有显示支持的用户态执行环境

```
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()])
}

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

pub fn print(args:fmt::Arguments){
    Stdout.write_fmt(args).unwrap();
}
```

完成上述代码编写后，就可以根据编写的函数实现标准输出宏 print!，println!

```
#[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)+)?));
    }
}
```

在执行编译运行后，可以在屏幕上得到下面的输出

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

理解：qemu-riscv64 程序模拟不同处理器的用户态指令的执行，当执行我们程序中相关的系统调用时，它会把不同处理器（如RISC-V）的Linux系统调用转换为本机处理器（如x86-64）上的Linux系统调用，这样就可以让本机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。因此我们上述的代码并不能在裸机上运行，而是建立在存在linux操作系统之上，我们所写的的代码中调用的指令也是本机上的linux系统调用。

实验硬件的组成

- 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

QEMU模拟的物理内存空间

```
static const struct MemmapEntry {
    hwaddr base;
    hwaddr size;
} virt_memmap[] = {
    [VIRT_DEBUG] =        {        0x0,         0x100 },
    [VIRT_MROM] =         {     0x1000,        0xf000 },
    [VIRT_TEST] =         {   0x100000,        0x1000 },
    [VIRT_RTC] =          {   0x101000,        0x1000 },
    [VIRT_CLINT] =        {  0x2000000,       0x10000 },
    [VIRT_PCIE_PIO] =     {  0x3000000,       0x10000 },
    [VIRT_PLIC] =         {  0xc000000, VIRT_PLIC_SIZE(VIRT_CPUS_MAX * 2) },
    [VIRT_UART0] =        { 0x10000000,         0x100 },
    [VIRT_VIRTIO] =       { 0x10001000,        0x1000 },
    [VIRT_FLASH] =        { 0x20000000,     0x4000000 },
    [VIRT_PCIE_ECAM] =    { 0x30000000,    0x10000000 },
    [VIRT_PCIE_MMIO] =    { 0x40000000,    0x40000000 },
    [VIRT_DRAM] =         { 0x80000000,           0x0 },
};
```

重要的两个物理内存空间

- VIRT_DRAM：这是计算机的物理内存，DRAM的内存起始地址是 `0x80000000` ，缺省大小为128MB。在本书中一般限制为8MB。
- VIRT_UART0：这是串口的控制寄存器区域，串口相关的寄存器起始地址是 `0x10000000` ，范围是 `0x100` ，我们通过访问这段特殊的区域来实现字符输入输出的管理与控制。

一些常用指令

```
rust-readobj -h 此命令可以查看文件信息，可以看到程序的入口信息
rust-objdump -S 反汇编

cargo build --release   编译生成ELF格式的执行文件

rust-objcopy --binary-architecture=riscv64 target/riscv64gc-unknown-none-elf/release/bare_os --strip-all -O binary target/riscv64gc-unknown-none-elf/release/bare_os.bin      把ELF执行文件转成bianary文件

 qemu-system-riscv64 -machine virt -nographic -bios ../bootloader/rustsbi-qemu.bin -device loader,file=target/riscv64gc-unknown-none-elf/release/bare_os.bin,addr=0x80200000  加载运行
```

裸机的启动过程

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。

因此我们需要做的任务就是 设置正确的程序内存布局

- 改写链接脚本调整链接器的行为
- 配置栈空间的布局

链接脚本

```
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字，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)
    }
}
```

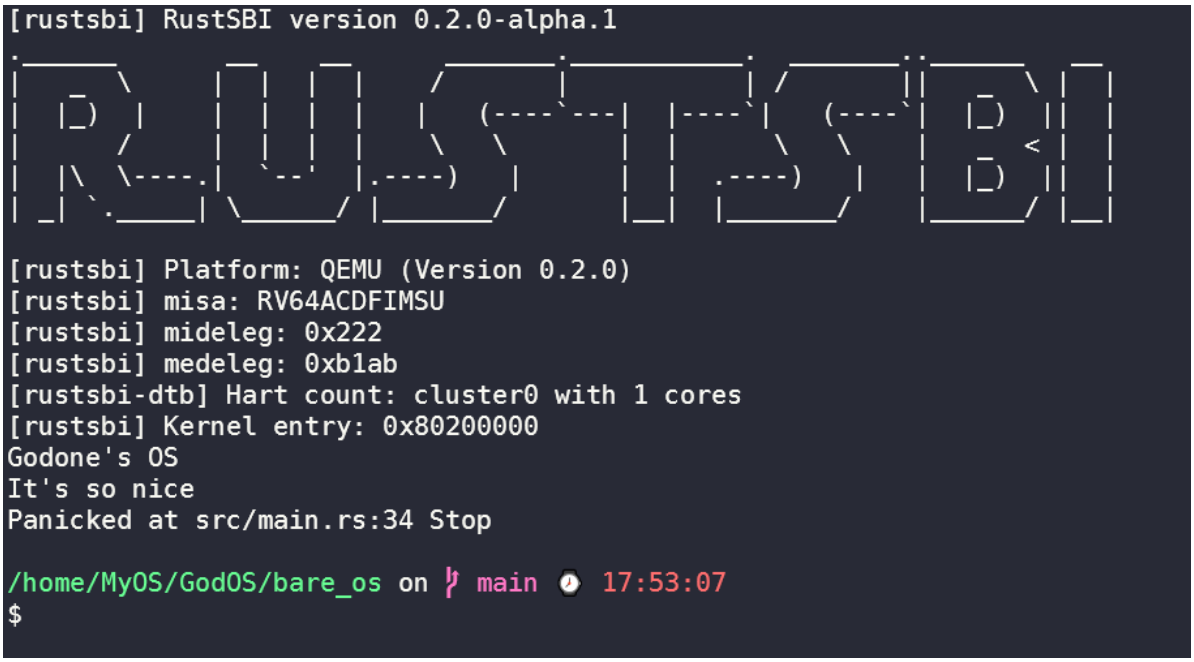
entry.asm汇编代码

```
    .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
```

在配置完这些链接脚本和进行初始化的汇编代码后，我们利用RustSBI使用riscv提供的二进制接口，实现答应字符的功能以此来检验我们的工作是否完成。

最终实现的效果如下



到此，我们已经可以完成操作系统的启动以及一部分的显示功能。