从一个Bare Metal APP开始

- 本节课的内容
  - Rust APP -> Rust Bare-metal APP
  - 通过批处理系统引入**特权级**的概念
  - 批处理->多道程序，引入任务调度与时钟中断的概念
  - 👉 完成 ch3 实验

引言

> 目标是以简洁的 RISC-V 基本架构为底层硬件基础，根据上层应用从小到大的需求，按 OS 发展的历史脉络，逐步讲解如何设计实现能满足“从简单到复杂”应用需求的多个“小”操作系统。

预备知识

程序设计语言（汇编、C 和 Rust）

- 😖 不是开发应用程序
- 😄 而是开发系统程序

数据结构

- 😄 理解基本数据结构即可

Why Rust

目前常见的操作系统内核都是基于 C 语言的，为何要推荐 Rust 语言？

- 事实上， C 语言就是为写 UNIX 而诞生的。Dennis Ritchie 和 KenThompson 没有期望设计一种新语言能帮助高效地开发复杂与并发的操作系统逻辑(面向未来)，而是希望用一种简洁的方式来代替难以使用的汇编语言抽象出计算机的行为，便于编写控制计算机硬件的操作系统（符合当时实际情况）。
- C 语言的指针既是天使又是魔鬼。它灵活且易于使用，但语言本身几乎不保证安全性，且缺少有效的并发支持。这导致内存和并发漏洞成为当前基于 C 语言的主流操作系统的噩梦。
- Rust 语言具有与 C 一样的硬件控制能力，且大大强化了安全编程和抽象编程能力。从某种角度上看，新出现的 Rust 语言的核心目标是解决 C 的短板，取代 C 。所以用 Rust 写 OS 具有很好的开发和运行体验。

- 用 Rust 写 OS 的代价仅仅是学会用 Rust 编程。

Why RISC-V ?

- 目前为止最常见的指令集架构是 x86 和 ARM ，它们已广泛应用在服务器、台式机、移动终端和很多嵌入式系统中。由于它们的通用性和向后兼容性需求，需要支持非常多（包括几十年前实现）的软件系统和应用需求，导致这些指令集架构越来越复杂。
- x86 后向兼容的策略确保了它在桌面和服务器领域的江湖地位，但导致其丢不掉很多已经比较过时的硬件设计，让操作系统通过冗余的代码来适配各种新老硬件特征。
- x86 和 ARM 在商业上都很成功，其广泛使用使得其 CPU 硬件逻辑越来越复杂，且不够开放，不能改变，不是开源的，难以让感兴趣探索硬件的学生了解硬件细节，在某种程度上让CPU成为了一个黑盒子，并使得操作系统与硬件的交互变得不那么透明，增加了学习操作系统的负担。
- 从某种角度上看，新出现的 RISC-V 的核心目标是灵活适应未来的 AIoT （人工智能物联网, AI + IoT）场景，保证基本功能，提供可配置的扩展功能。其开源特征使得学生都可以深入CPU的运行细节，甚至可以方便地设计一个 RISC-V CPU。从而可帮助学生深入了解操作系统与硬件的协同执行过程。
- 编写面向 RISC-V 的 OS 的硬件学习代价仅仅是你了解 RISC-V 的 Supervisor 特权模式，知道 OS 在 Supervisor 特权模式下的控制能力。

RISC-V

RISC-V Specifications

chapter1_riscv.md · 华中科技大学操作系统团队/pke-doc - Gitee.com

- RISC
  - Reduced instruction set computer
- 寄存器

表1.1 RV64G的32个通用寄存器（寄存器的宽度都是64位）

寄存器	编程接口名称 （ABI）	描述	使用
x0	zero	Hard-wired zero	硬件零
x1	ra	Return address	常用于保存（函数的）返回地址
x2	sp	Stack pointer	栈顶指针
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-7	t0-2	Temporary	临时寄存器
x8	s0/fp	Saved Register/ Frame pointer	（函数调用时）保存的寄存器和栈顶指针
x9	s1	Saved register	（函数调用时）保存的寄存器
x10-11	a0-1	Function argument/ return value	（函数调用时）的参数/函数的返回值
x12-17	a2-7	Function argument	（函数调用时）的参数
x18-27	s2-11	Saved register	（函数调用时）保存的寄存器
x28-31	t3-6	Temporary	临时寄存器

- 特权等级

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

Table 1.1: RISC-V privilege levels.

  - U态：用户程序
  - S态：操作系统内核
  - M态：the only mandatory privilege level for a RISC-V hardware platform
    - 内核之下，内核之前
    - RustSBI：附录 C:深入机器模式:RustSBI
    - OpenSBI

git 基本操作

```
git add ${file_name}
git add .
git commit -m"feat: hello"
git push
git push origin ${branch_name}
git push origin master
git pull
git pull origin ${branch_name}
git remote -v
git remote add origin https://path/to/your/git/repo
git cherry-pick ${commit_hash}
git diff
```

章节导引

实验环境配置

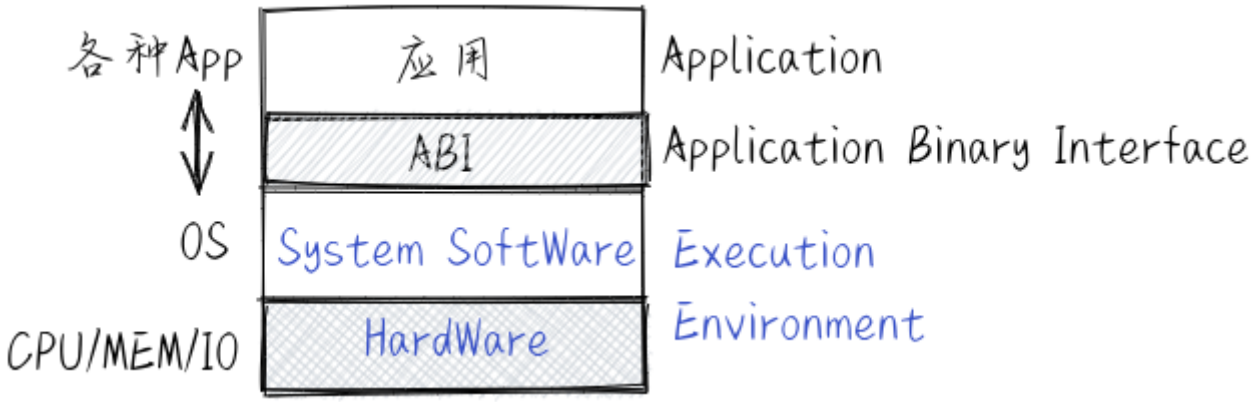
How to build a Rust APP?

- `cargo new helloworld`
- `cargo build [--release]`
- `exec`
- You get your own "hello world"!!!
  - inside Linux ❌
- Let's get it from bare-metal

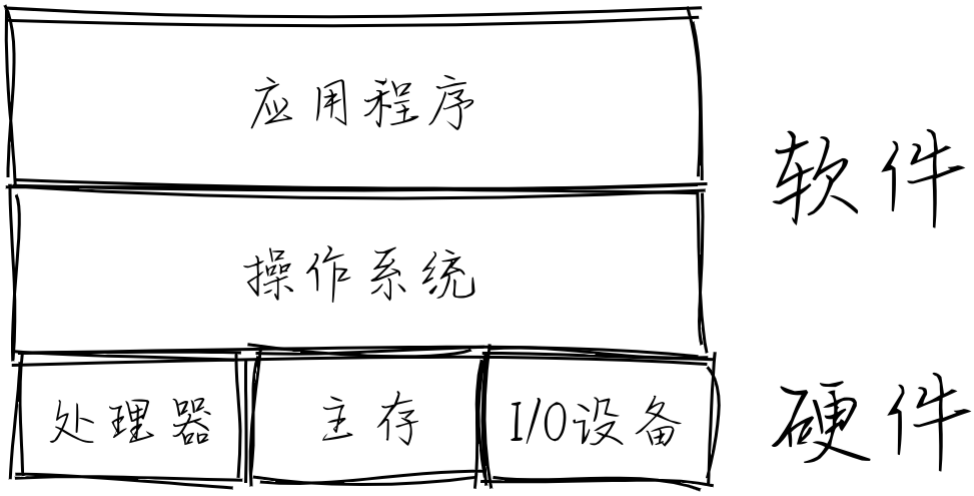
操作系统概述

什么是操作系统

站在我们自己的hello world应用的角度来看，我们可以发现常见的应用程序其实是运行在由硬件、操作系统内核、运行时库、图形界面支持库等所包起来的一个 执行环境 (Execution Environment) 中。执行环境提供了运行应用软件所需的运行时服务，包括内存管理、文件系统访问、网络连接等，这些服务大部分是由操作系统来提供的



- 操作系统
  - 向下管理并控制计算机硬件和各种外设
  - 向上管理应用软件并提供各种服务



操作系统是一种系统软件，主要功能是向下管理CPU、内存和各种外设等硬件资源，并形成软件执行环境来向上管理和服务应用软件。

在一般情况下，操作系统的主要组成包括：

1. 操作系统内核：操作系统的核心部分，负责控制计算机的硬件资源并为用户和应用程序提供服务。
2. 系统工具和软件库：为操作系统提供基本功能的软件，包括工具软件和系统软件库等。
3. 用户接口：是操作系统的外壳，是用户与操作系统交互的方式。用户接口包括图形用户界面（GUI）和命令行界面（CLI）等。

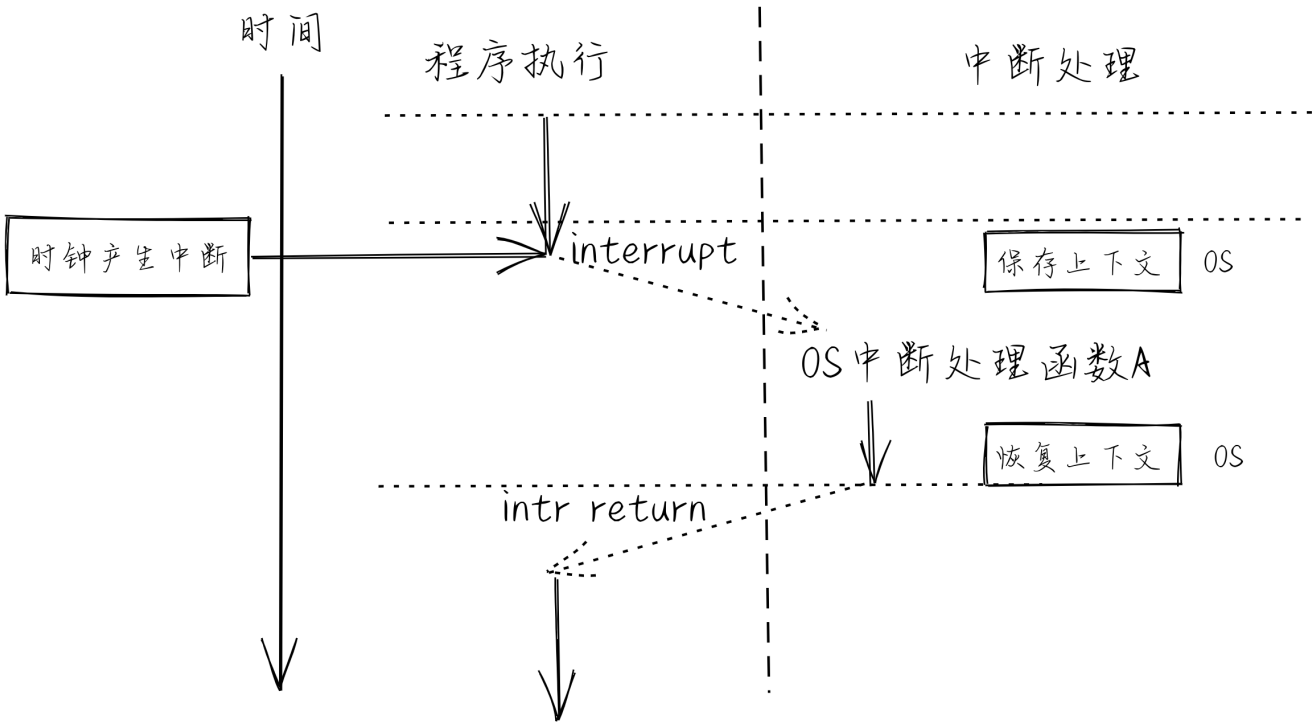
而咱们重点讲述的对象是操作系统内核，它的主要组成部分包括：

1. 进程/线程管理：内核负责管理系统中的进程或线程，创建、销毁、调度和切换进程或线程。
2. 内存管理：内核负责管理系统的内存，分配和回收内存空间，并保证进程之间的内存隔离。
3. 文件系统：内核提供文件系统接口，负责管理存储设备上的文件和目录，并允许应用访问文件系统。
4. 网络通信：内核提供网络通信接口，负责管理网络连接并允许应用进行网络通信。

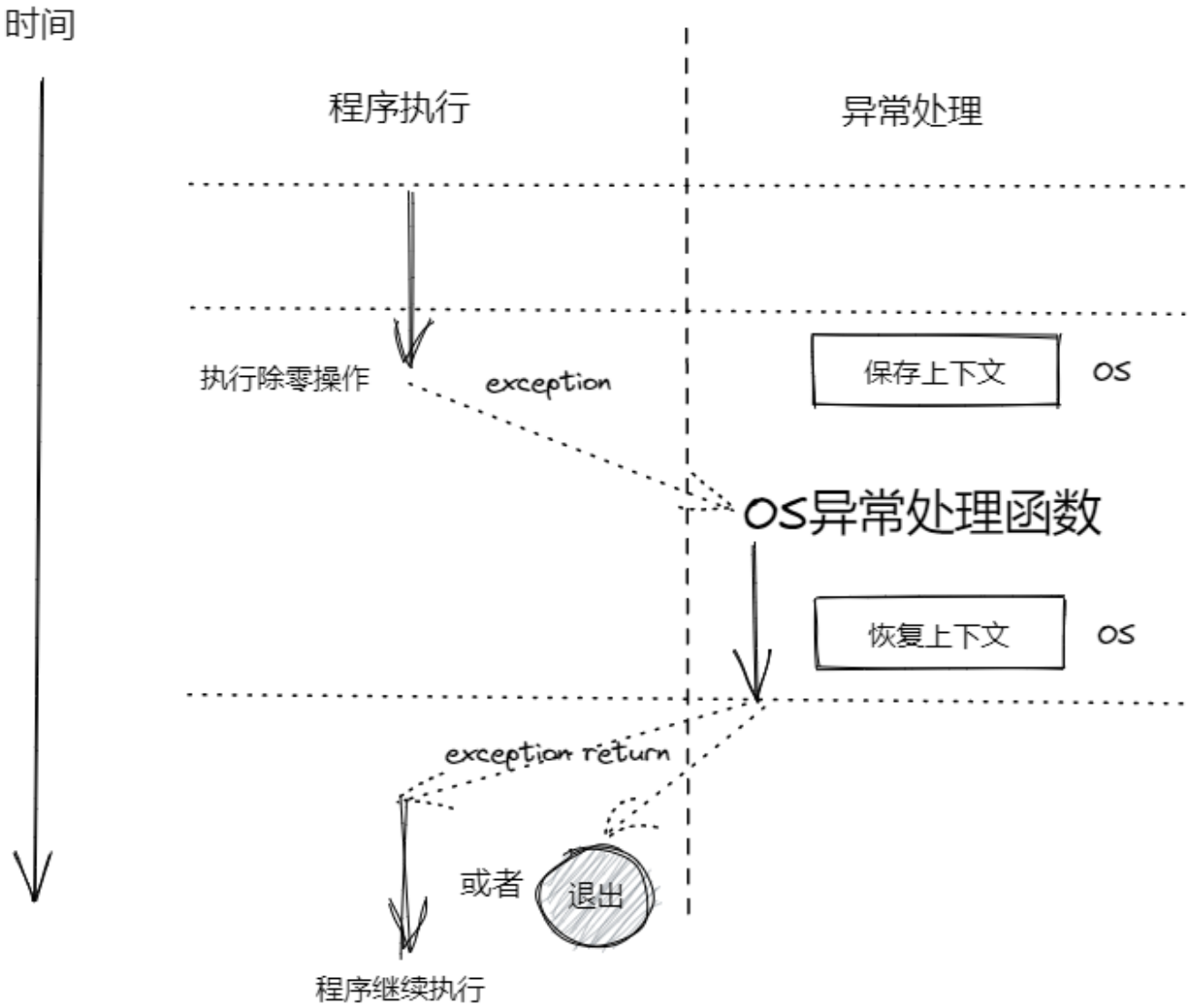
5. 设备驱动：内核提供设备驱动接口，负责管理硬件设备并允许应用和内核其他部分访问设备。
6. 同步互斥：内核负责协调多个进程或线程之间对共享资源的访问。同步功能主要用于解决进程或线程之间的协作问题，互斥功能主要用于解决进程或线程之间的竞争问题。
7. 系统调用接口：内核提供给应用程序访问系统服务的入口，应用程序通过系统调用接口调用操作系统提供的服务，如文件系统、网络通信、进程管理等。

几个基本概念

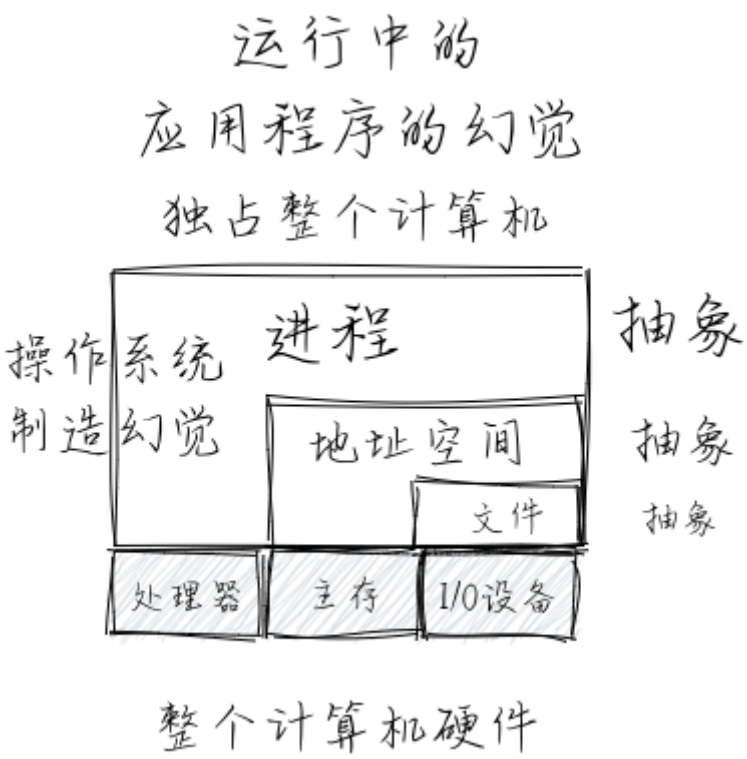
- 用户态（U态）与内核态（S态）的区分
- 用户态 普通控制流
  - 正常函数调用之类的，编译器负责函数调用上下文的管理
- 内核态 异常控制流
  - 中断
    - 操作系统与外设的沟通接口



  - 异常&陷入
    - 用户态与内核态的沟通接口



    - 除0异常
    - 缺页异常
    - syscall 操作系统的系统调用接口 - rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档
- 进程
    - 一个正在运行的程序实例



- 地址空间
  - 虚拟地址空间与物理地址空间的区分
  - 虚拟地址空间：用户代码看到的内存
  - 物理地址空间：真实的物理内存
  - 页式地址空间管理->缺页错误
- 文件
  - fd
  - **文件** (File) 主要用于对持久存储的抽象，并进一步扩展到为外设的抽象

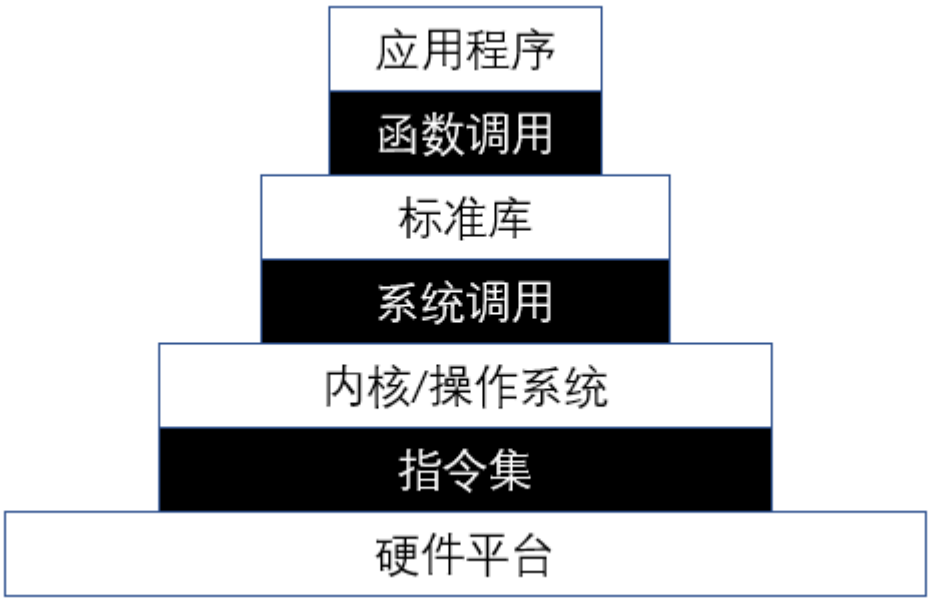
第一章：应用程序与基本执行环境

Back to our Rust APP：

展开看看这玩意怎么输出hello world的

`strace target/release/os`

应用程序执行环境



How to build a bare metal APP by Rust?

- What is bare-metal
  - 裸金属
  - 也就是应用直接在硬件上跑，没有操作系统支持
- ppt Compiler与OS
- How to build
  - 修改编译目标
  - 剥掉标准库
  - 剥掉内核/操作系统
  - 应用程序直接操作硬件平台

1. change target

目标平台与目标三元组

> 现代编译器工具集（以C或Rust编译器为例）的主要工作流程如下：
>
> 1. 源代码（source code） –> 预处理器（preprocessor） –> 宏展开的源代码
> 2. 宏展开的源代码 –> 编译器（compiler） –> 汇编程序
> 3. 汇编程序 –> 汇编器（assembler）–> 目标代码（object code）
> 4. 目标代码 –> 链接器（linker） –> 可执行文件（executables）

对于一份用某种编程语言实现的应用程序源代码而言，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。

从上面给出的 应用程序执行环境栈 可以看出：

- 如果用户态基于的内核不同，会导致系统调用接口不同或者语义不一致；
- 如果底层硬件不同，对于硬件资源的访问方式会有差异
  - 特别是如果 ISA 不同，则向软件提供的指令集和寄存器都不同。

它们都会导致最终生成的可执行文件有很大不同。需要指出的是，某些编译器支持同一份源代码无需修改就可编译到多个不同的目标平台并在上面运行。这种情况下，源代码是 **跨平台** 的。而另一些编译器则已经预设好了一个固定的目标平台。

Rust编译器通过 **目标三元组** (Target Triplet) 来描述一个软件运行的目标平台。它一般包括 CPU、操作系统和运行时库等信息，从而控制Rust编译器可执行代码生成。

- cargo run或者直接执行 -> target 为你当前的os
- 当前的编译目标
  - `rustc --version --verbose`
  - host: x86_64-unknown-linux-gnu
    - 体系结构：x86_64
    - CPU厂商：unknown
    - 运行的系统：Linux
    - 运行时库：gcc
- bare-metal target 为你的目标硬件平台
- 让Rust compiler知道你的目标硬件平台长什么样->编译出目标体系结构的机器码

修改目标平台

我们的主线任务：希望能够在另一个硬件平台上运行 `Hello, world!`，而与之前的默认平台不同的地方在于，我们将 CPU 架构从 x86_64 换成 RISC-V。

- `rustc --print target-list | grep riscv` 瞅瞅Rust编译器支持哪些RISC-V的目标平台
  - 这里我们选择 `riscv64gc-unknown-none-elf` 目标平台
  - CPU 架构是 riscv64gc
  - CPU厂商是 unknown
  - 操作系统是 none
  - 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），但可以生成 ELF 格式的执行程序。
- 修改目标平台为 `riscv64gc-unknown-none-elf`
  - 体系结构：riscv64gc
    - riscv64 64位的RISC-V
    - g拓展：基本整数指令集和四个标准扩展指令集（即“IMAFD”）的总称

        - I：整数拓展
        - M：乘除拓展
        - A：标准原子拓展
        - F/D：单/双精度浮点数
      - c拓展：压缩扩展，用于指令压缩，32bit->16bit
    - CPU厂商：unknown
    - 运行的系统：none
    - 运行时库：elf，表示没有标准的运行时库，没有任何系统调用的封装支持，生成 ELF 格式的执行程序

> 这里我们之所以不选择有 linux-gnu 系统调用支持的目标平台 `riscv64gc-unknown-linux-gnu`，是因为我们只是想跑一个在裸机环境上运行的 `Hello, world!` 应用程序，没有必要使用Linux操作系统提供的那么高级的抽象和多余的操作系统服务。而且我们很清楚后续我们要开发的是一个操作系统内核，它必须直面底层物理硬件（bare-metal）来提供精简的操作系统服务功能，通用操作系统（如 Linux）提供的很多系统调用服务对这个内核而言是多余的。

- `cargo run --target riscv64gc-unknown-none-elf`
  - `error[E0463]: can't find crate for std`
  - 在none-elf平台上，没有操作系统为Rust标准库提供支持
  - **裸机平台** (bare-metal)

> 幸运的是，除了 std 之外，Rust 还有一个不需要任何操作系统支持的核心库 core， 它包含了 Rust 语言相当一部分核心机制，可以满足本门课程的需求。 有很多第三方库也不依赖标准库 std，而仅仅依赖核心库 core。

这也是Rust可以用于系统开发的关键！！！

为了以裸机平台为目标编译程序，我们要将对标准库 std 的引用换成核心库 core。

2. 移除Rust标准库支持

> 在之前的开发环境配置中，我们已经在 rustup 工具链中安装了这个目标平台支持，因此并不是该目标平台未安装的问题。这个问题只是单纯的表示在这个目标平台上找不到 Rust 标准库 std。我们之前曾经提到过，编程语言的标准库或三方库的某些功能会直接或间接的用到操作系统提供的系统调用。但目前我们所选的目标平台不存在任何操作系统支持，于是 Rust 并没有为这个目标平台支持完整的标准库 std。类似这样的平台通常被我们称为 **裸机平台** (bare-metal)。这意味着在裸机平台上的软件没有传统操作系统支持。

为了不要每次都指定target

- 添加.cargo配置

```
```

```
[build]
target = "riscv64gc-unknown-none-elf"
```

- 三个错误
  - error[E0463]: can't find crate for `std
  - error: cannot find macro `println` in this scope
  - error: `#[panic_handler]` function required, but not found
- 添加 `#![no_std]` 注解，告诉 Rust 编译器不使用 Rust 标准库 std 转而使用核心库 core
- 提供 panic_handller

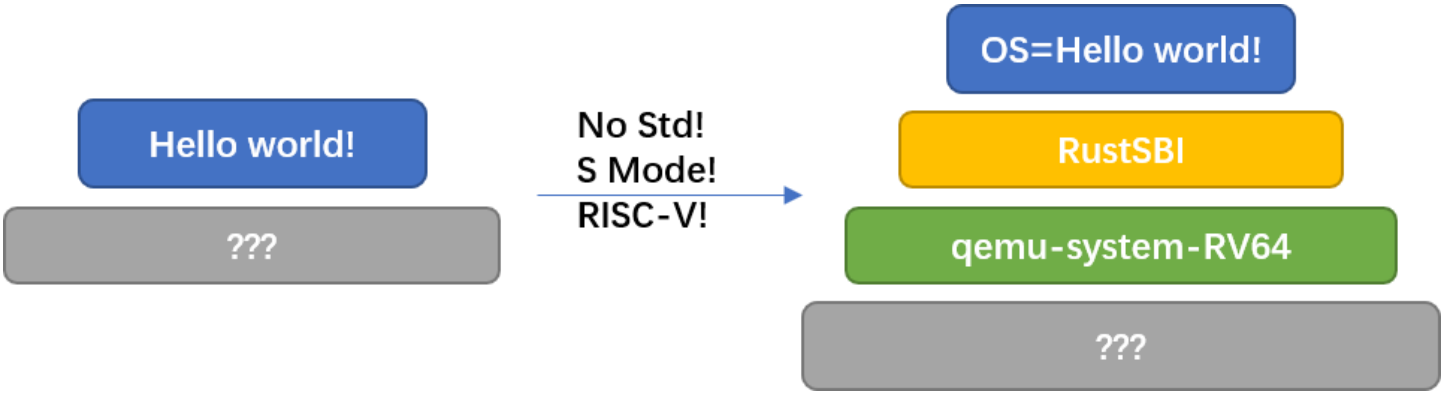
```
// os/src/lang_items.rs
use core::panic::PanicInfo;

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

- 添加 `#![no_main]` 注解，告诉编译器我们没有一般意义上的 `main` 函数
- 注释掉 main 函数
- 成功编译！
- 这是一个合法的RISC-V 64执行程序，但是里面啥都没有
  - 分析被移除标准库的程序
- 我们把所有依赖都剥掉了，现在我们得一层层给他加回来
- 在下面几节，我们将建立有支持显示字符串的最小执行环境。

3. 先写一个用户态程序试试？

- 构建用户态执行环境
  - 实现一堆syscall
  - 封装println宏
- 这毕竟是一个用户态程序，依赖syscall帮我们实现退出&输出
- syscall->qemu-riscv64提供的riscv系统内核模拟
- 我们得想办法整一个裸机环境，实现我们自己的内核态
- 让我们的os跑在一个裸机环境上，在内核态下运行



- QEMU为我们提供了一个模拟的硬件环境 qemu-system-riscv64
- 内核第一条指令(基础篇)

```
qemu-system-riscv64 \
            -machine virt \
            -nographic \
            -bios $(BOOTLOADER) \
            -device loader,file=$(KERNEL_BIN),addr=$(KERNEL_ENTRY_PA)
```

QEMU引导与启动

- 上电
- QEMU CPU 0x1000
- 跳转到 0x80000000（QEMU固定）
- RustSBI初始化
- 跳转到 0x80200000（SBI固定）
- 在反汇编里面找找我们的 _start
- 我们得想办法让我们的内核的入口(_start)刚好在0x80200000这个地址

4. 把 syscall 换成 sbi_call

- 使用RustSBI提供的服务
  - sbi.rs

```
use core::arch::asm;
#[inline(always)]
fn sbi_call(which: usize, arg0: usize, arg1: usize, arg2: usize) -> usize
    let mut ret;
    unsafe {
```

```
        asm!(
            "ecall",
            inlateout("x10") arg0 => ret,
            in("x11") arg1,
            in("x12") arg2,
            in("x17") which,
        );
    }
    ret
}
```

  - sbi_call 的函数签名
    - which 表示请求 RustSBI 的服务的类型（RustSBI 可以提供多种不同类型的服务）
    - arg0 ~ arg2 表示传递给 RustSBI 的 3 个参数
    - RustSBI 在将请求处理完毕后，会给内核一个返回值，这个返回值也会被 sbi_call 函数返回

- syscall_exit -> shutdown

```
pub fn shutdown() -> ! {
    sbi_call(SBI_SHUTDOWN, 0, 0, 0);
    panic!("It should shutdown!");
}
```

- syscall_write -> console_putchar

```
pub fn console_putchar(c: usize) {
    sbi_call(SBI_CONSOLE_PUTCHAR, c, 0, 0);
}
```

- 修改一些对应的实现
  - `Write Trait`

```
impl Write for Stdout {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        // sys_write(1, s.as_bytes());
        for c in s.chars(){
            console_putchar(c as usize);
        }
```

```
    7
    8         Ok(())
    9     }
   10 }
```

5. 设置正确的程序内存布局

- 前置知识
  - 内核第一条指令(基础篇)
  - 程序编译与链接
  - 函数调用与栈
- 增加链接脚本
- 增加内核第一条指令（为内核函数调用准备一个栈空间）

```
    1     .section .text.entry
    2     .globl _start
    3 _start:
    4     la sp, boot_stack_top
    5     call rust_main
    6
    7     .section .bss.stack
    8     .globl boot_stack_lower_bound
    9 boot_stack_lower_bound:
   10     .space 4096 * 16
   11     .globl boot_stack_top
   12 boot_stack_top:
```

然后跳转到main

Finally: 看到 hello world

第一章总结

- 从一个 Rust App 开始，我们一步步去掉依赖，再一步步把依赖加回来
  - 修改编译目标
  - 移除标准库支持
  - 实现sbi_call与硬件交互
  - 准备基本的内核环境并跳转到内核入口
  - 这个 hello world 属于 App 直接操作硬件

- 还有一些要做的。。。
  - clear BSS
  - 使用log库
- 从一个 Bare Metal App 开始，我们又要一步步往上加东西
- 让我们的bare metal App 变成能够支持其他应用运行的简易os

第二章 批处理系统

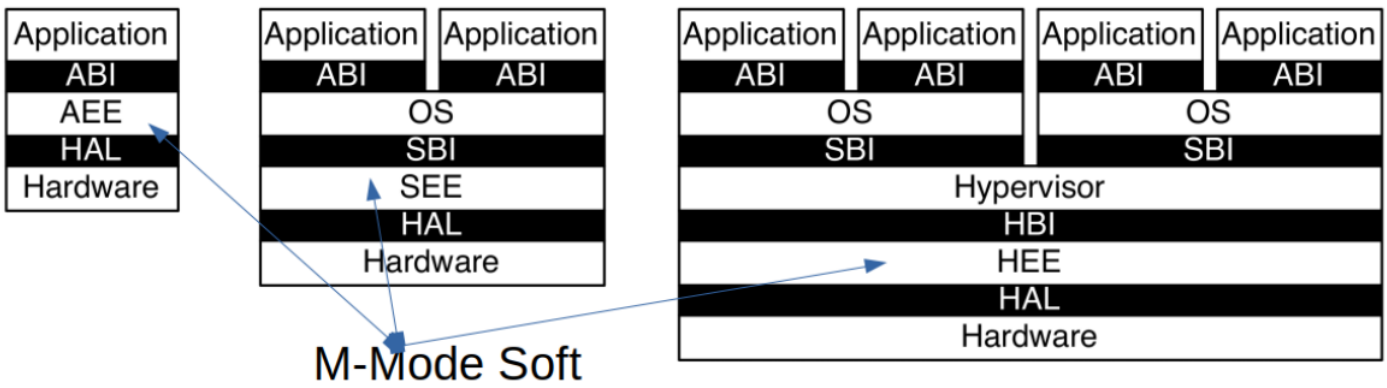
- 建议自行阅读这两个课件
  - 第三讲 基于特权级的隔离与批处理
    - 第一节 从OS角度看计算机系统
    - 第二节 从OS角度看RISC-V
    - 第三节（我们实验中的ch2分支） 实践：批处理操作系统

将多个程序打包一起输入计算机并一个一个执行

> 应用程序难免会出错，如果一个程序的错误导致整个操作系统都无法运行，那就太糟糕了。 *保护* 操作系统不受出错程序破坏的机制被称为 **特权级** (Privilege) 机制， 它实现了用户态和内核态的隔离。

重点：特权级机制

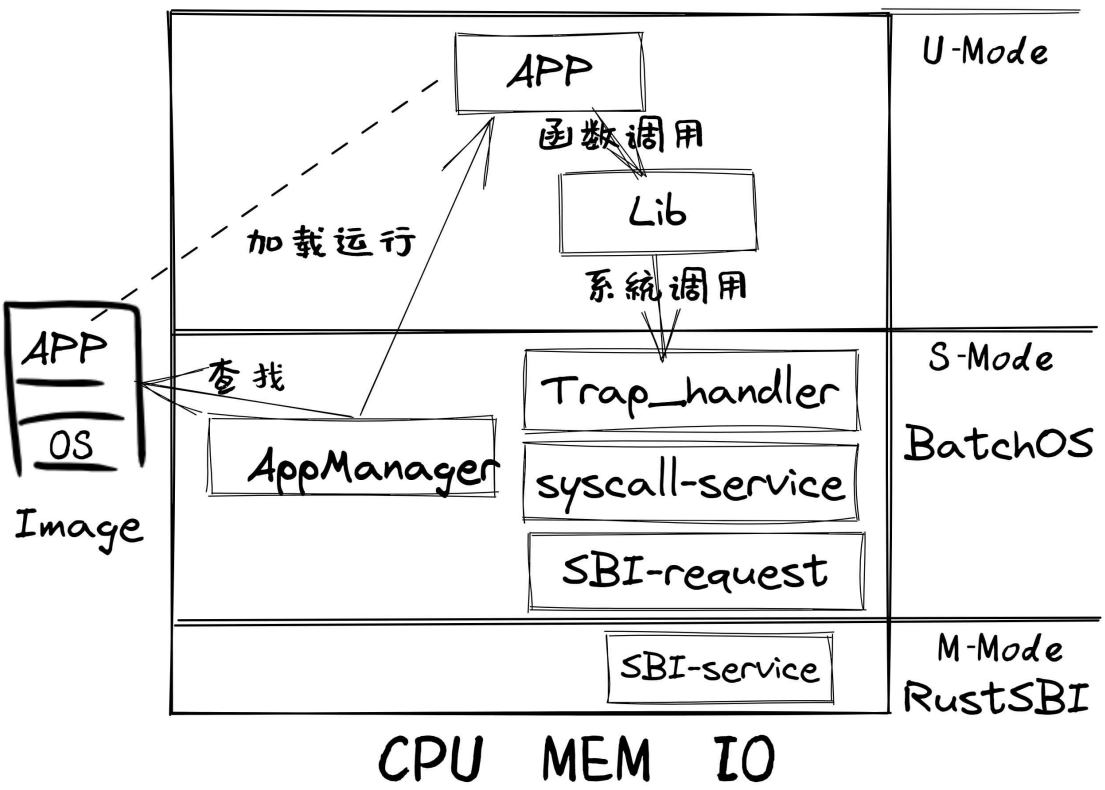
（`qemu-riscv` 在之前为我们模拟的用户态，我们得在 `qemu-system-riscv` 上帮别人模拟）



RISC-V 系统模式：执行环境

执行环境	编码	含义	跨越特权级
APP	00	User/Application	`ecall`
OS	01	Supervisor	`ecall` `sret`
VMM	10	Hypervisor	---
BIOS	11	Machine	`ecall` `mret`

- M, S, U 组合在一起的硬件系统适合运行类似UNIX的操作系统



需要注意的：

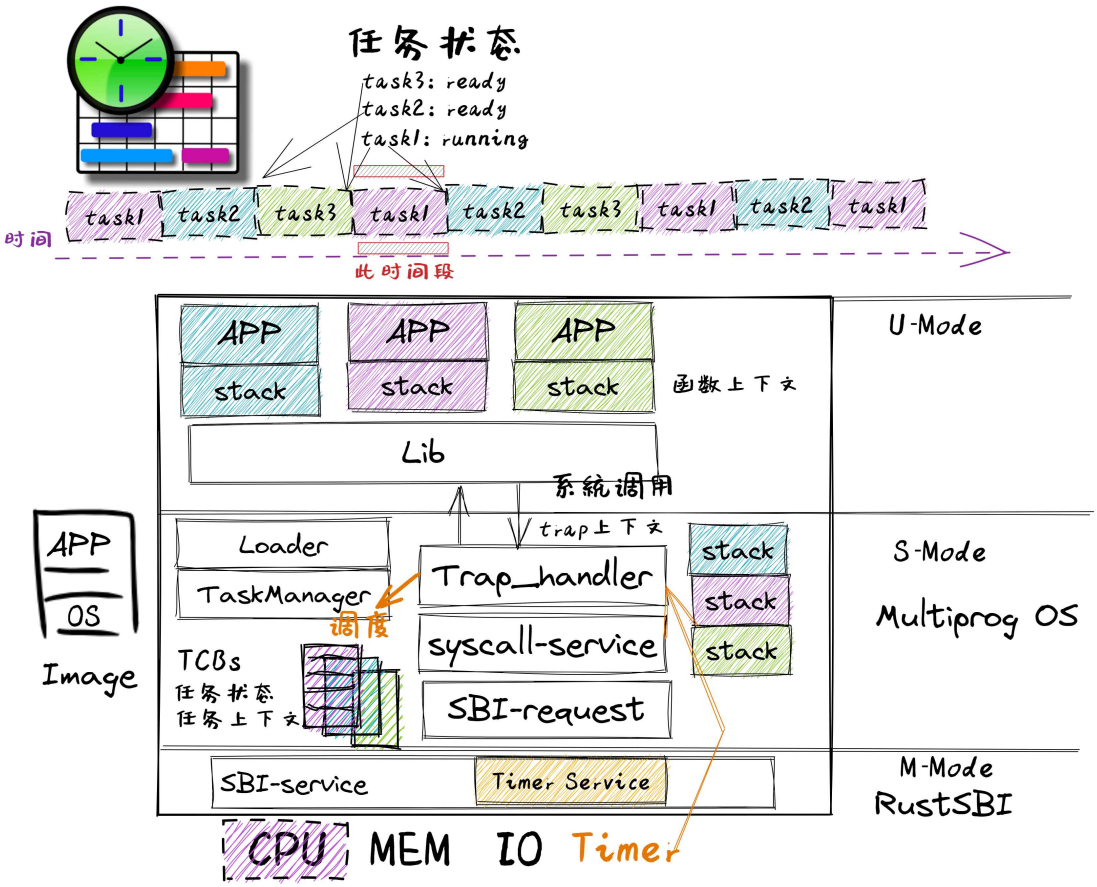
- run_next_app 中 需要手动 drop app_manager
  - 后面的unsafe代码逃脱了Rust语言的生命周期机制
  - 需要手动释放这个对象
- __restore
  - 从内存中 pop 出下一个 task 的通用寄存器、状态寄存器
  - sret
- 这是一个跳出函数调用规范的过程-> 特权等级切换

第三章 多道程序与分时多任务

- 建议自行阅读这两个课件
  - 第四讲 多道程序与分时多任务
    - 第一节 进程和进程模型
    - 第二节 实践：多道程序与分时多任务操作系统

> **批处理与多道程序的区别是什么?**
>
> 对于批处理系统而言，它在一段时间内可以处理一批程序，但内存中只放一个程序，处理器一次只能运行一个程序，只有在一个程序运行完毕后再把另外一个程序调入内存，并执行。即批处理系统不能交错执行多个程序。

> 对于支持多道程序的系统而言，它在一段时间内也可以处理一批程序，但内存中可以放多个程序，一个程序在执行过程中，可以主动（协作式）或被动（抢占式）地放弃自己的执行，让另外一个程序执行。即支持多道程序的系统可以交错地执行多个程序，这样系统的利用率会更高。



- 不同类型的上下文切换
- yield主动让出CPU时间片
- 时钟中断导致的调度

本文在线链接（可以直接评论orz）