RISC-V 移植任务

任务说明

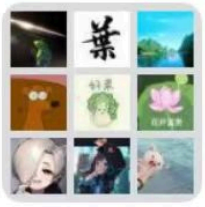
> 本文档未经迭代，如存在模糊、歧义、错误之处，欢迎向课程组提出反馈。有关 RISC-V 架构的细节，请以官方文档为准：RISC-V 指令集规范。

该任务需要同学实现支持 RISC-V 指令集(32位/64位均可)的 MOS，并通过 RISC-V 的 QEMU 仿真运行。任务总共分为 **内核启动**，**内存管理**，**例外处理**，**进程与调度**，**用户库** 五个部分，在每个部分中除去给定的目标外，不会对其他实现细节做要求。在完成每个部分后，将提供一些**不计入总分的拓展任务（可选）**，最终实现的内核需要支持原 MOS 中 SHELL 的功能。

任务要求

- 完整可运行的内核代码。
- 内核文档(简要地介绍说明每部分的所实际实现的内容即可)。
- 通过挑战性任务截止日期后的答辩和检查。
- 查重。

交流群



群聊：OS 2024 移植交流组队群

该二维码7天内(4月24日前)有效，重新进入将更新

RISC-V介绍

背景介绍

> RISC-V 是一种开放标准的精简指令集架构。其具有高度模块化的特征，可以通过不同指令集拓展，来支持不同复杂度的应用。

RISC-V 指令集的处理器，除了必须实现的基本整数指令集（`RV32I` 或 `RV64I` 等）外，还提供了可选指令集，下面是一些重要的RISC-V扩展指令集：

- **I（Integer）**：基本整数指令集，为所有实现提供了基础。
- **M（Multiply/Divide）**：提供整数乘法和除法指令。
- **A（Atomic）**：提供原子操作指令，支持多核和并行计算环境中的同步。
- **F（Floating-Point）**：提供单精度浮点数运算指令。
- **D（Double-precision Floating-Point）**：提供双精度浮点数运算指令。
- **C（Compressed）**：提供压缩指令，以减少程序大小和提高执行效率。这对于嵌入式系统尤其重要。
- **Q（Quad-precision Floating-Point）**：提供四倍精度浮点数运算指令，主要用于高精度科学计算。
- **L (Decimal Floating-Point)**: 提供十进制浮点运算指令，主要用于金融和商业应用。
- **B（Bit Manipulation）**：提供位操作指令，以支持更高效的位级操作。
- **V（Vector）**：提供向量计算指令，支持高效的并行数据处理，适用于科学计算、机器学习等领域。
- **P（Packed-SIMD）**：提供SIMD（单指令多数据）操作指令，用于处理多个数据的操作，增强了处理图形、声音处理、科学计算等应用的能力。

RISC-V的这种模块化扩展设计允许芯片设计者根据具体的应用需求灵活选择需要的指令集组合，既可以优化性能，又能控制成本。随着RISC-V社区的发展，还可能出现更多的扩展指令集，进一步扩大其应用范围和性能优势。

指令集

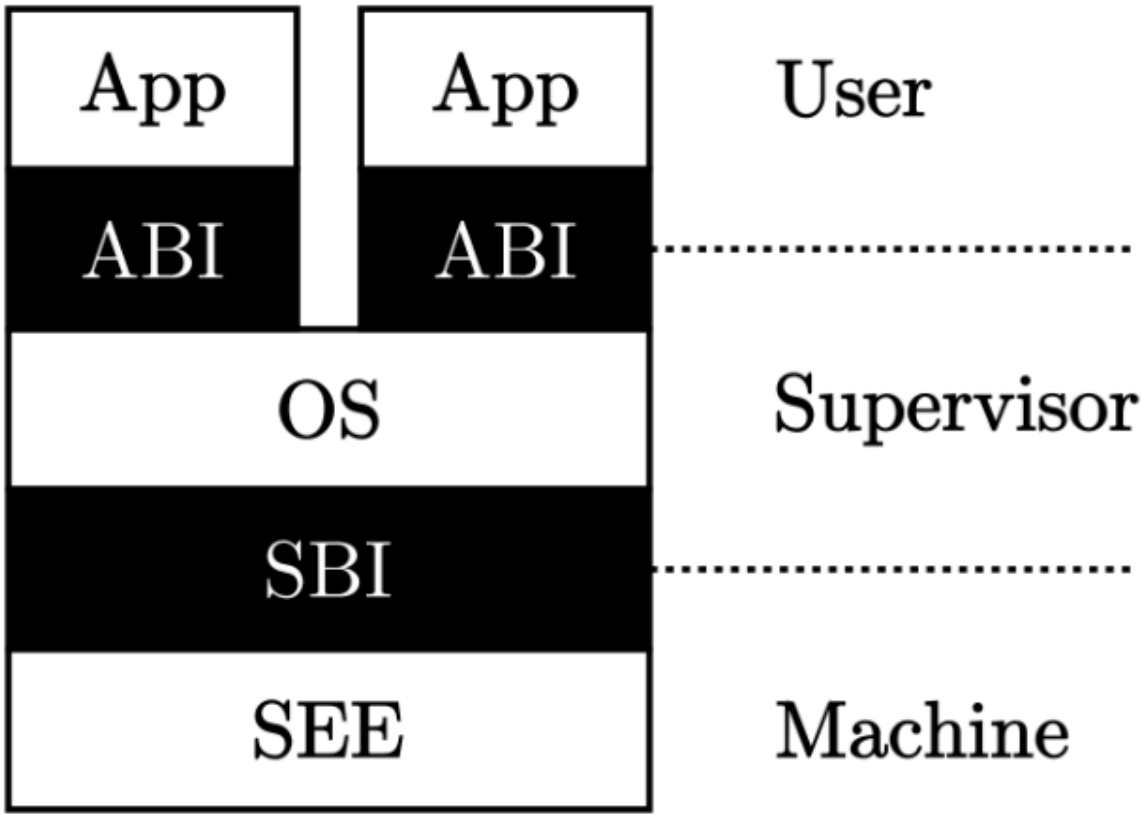
RISC-V 指令集与 MIPS 指令集同属精简指令集，有一定相似性，具体内容可以查阅 RISC-V指令集手册 学习。

特权架构

RISC-V的特权架构定义了实现RISC-V指令集的处理器在执行特权操作时的行为。这些特权操作包括访问控制、设备管理、以及中断和异常处理等。RISC-V的特权架构被设计为灵活和可扩展的，以支持各种系统级的应用，从简单的嵌入式系统到复杂的操作系统。特权架构主要分为三个级别：

1. **机器模式（Machine Mode, M-Mode）**： 这是最高的权限级别，拥有对所有硬件资源的完全访问权限。M-Mode通常用于引导过程和操作系统的最底层，比如中断处理和硬件异常。在M-Mode中，可以配置物理内存保护（PMP），管理中断和异常，以及设置其他特权模式的运行环境。
2. **监督者模式（Supervisor Mode, S-Mode）**： S-Mode用于操作系统的核心部分，比如内核。它允许操作系统管理用户程序和系统资源，但是相对于M-Mode，它的权限有所限制。S-Mode提供了虚拟内存支持、中断管理以及对某些硬件设备的限制访问能力。
3. **用户模式（User Mode, U-Mode）**： 这是最低的权限级别，通常用于运行应用程序。用户程序在U-Mode中运行，受到操作系统的约束和保护，无法直接访问硬件资源。

除了这些基本的模式，RISC-V特权架构还引入了可选的**虚拟化模式（Hypervisor Mode, H-Mode）**，用于支持虚拟机和更复杂的系统级虚拟化功能。（注：下文将M-mode称为M态，将S-Mode称为S态，将U-Mode成为U态）



RISC-V的特权架构设计考虑到了灵活性和模块化，允许基于不同的应用需求选择和配置不同的特权级别。这种设计支持了广泛的应用场景，从高效率的嵌入式设备到需要复杂操作系统支持的高性能计算系统。在本项任务中，OpenSBI承担了近乎M态所有的工作，你只需要负责S态和U态的实现。

状态寄存器

与MIPS中的协处理器寄存器类似，RISC-V也提供了用于控制系统的控制寄存器。控制寄存器是特权级别指令集的一部分，用于管理和控制处理器的各种硬件功能（每个特权级别都有**属于自己级别**的控制寄存器，越级访问会触发异常）。这些控制寄存器允许操作系统和其他特权软件执行如配置中断处理、设置虚拟内存映射、监视和控制硬件执行状态等任务。下面是一些RISC-V特权架构S态中重要的控制寄存器：

1. **sstatus（Supervisor Status Register）**：这是S态一个核心的控制寄存器，包含了全局中断使能位和其他用于控制机器模式执行环境的状态位。
2. **stvec（Supervisor Trap-Vector Base-Address Register）**：存储S态中断和异常的入口地址，指向处理中断时应该执行的代码。
3. **sepc（Supervisor Exception Program Counter）**：在发生异常时，保存触发异常指令的地址，以便异常处理完成后能够返回到正确的位置继续执行。
4. **sie 和 sip（Supervisor Interrupt-Enable and Interrupt-Pending Registers）**：这两个寄存器分别用于S态控制中断的使能和检查哪些中断是挂起状态，用于中断管理。
5. **satp（Supervisor Address Translation and Protection）**：在支持S-Mode的系统中，这个寄存器用于控制虚拟内存系统，包括页表的基地址和当前使用的页表模式。

除了上述寄存器，RISC-V还定义了其他多种控制寄存器，用于不同的监控、管理和优化任务。这些寄存器的具体集合和功能可能根据具体的RISC-V实现和所支持的特权级别不同。通过这些寄存器，操作系统能够实现对硬件的精细控制，优化性能，保证系统安全性和稳定性。需要使用`csrw`,`csrr`,`csrrw`等特权指令来访问这些控制寄存器。

RISC-V文档

关于RISC-V的内容细节这里不再过多赘述，请查阅下列文档。

- RISC-V指令集 （汇编指令说明与使用）
- RISC-V调用规范 (函数调用规范说明)
- RISC-V非特权级别文档
- RISC-V特权级别文档

内核启动

环境配置

笔者推荐Windows用户在WSL上进行开发，其他用户也请选择合适的方式。

下面是一些你可能会用到的工具：

- `build-essentials`
- `riscv64-unknown-elf-` 交叉编译工具链
- `gdb-multiarch`
- `qemu-system-riscv32` 或 `qemu-system-riscv64`
- `OpenSBI` 引导

对于交叉编译工具链和GDB，推荐通过apt等包管理器进行安装;对于QEMU，则推荐通过源码编译安装QEMU，很多源上的QEMU版本过老，并且缺乏相应的库文件；对于OpenSBI，其实较新版本的QEMU默认内置了OpenSBI，如果有修改OpenSBI代码的需求，可以自行编译OpenSBI。

如果 MacOS 用户有自行构建交叉编译器的需求，可以在 riscv-gnu-toolchain 仓库获取源代码并编译安装。

MacOS 用户在编译 GDB 时可能存在依赖缺失的问题，可尝试替换 LLDB ，具体命令可参考 GDB 与 LLDB 的命令对照表。

内核启动

在完成环境的搭建后，我们正式开始内核开发。你需要：

- **创建内核项目**

你可以参考原MOS文件夹结构，并通过Makefile管理代码。下面是你可能会在Makefile中用到的函数：

```
# pattern表示要查找的模式，replacement表示替换为的文本，text表示被搜索和替换的文本列表。该函数会在text中查找符合pattern的部分，并将其替换为replacement指定的文本。
$(patsubst <pattern>,<replacement>,<text>)
# wildcard 用于匹配模式下的所有文件
$(wildcard <pattern>)
# filename 用于去除所有传入参数的后缀名，返回结果列表
$(filename <list>)
# dir 获取传入参数的所有路径，去除完整文件名，返回结果列表
$(dir <list>)
# notdir 用于去除所有传入参数的路径，只保留完整文件名，返回结果列表
```

在Makefile中你可能还需要一些Bash脚本的知识，这里不再过多赘述。

- **编写内核链接脚本**

链接脚本的详细内容可以参考LD文档，这里给出简单的示例和介绍:

```
# 指定架构为riscv
OUTPUT_ARCH(riscv)
# 指定入口函数为_start
# 定义变量
ENTRY(_start)
BASE_ADDR = 0x80200000
KERNEL_END_ADDR = 0x80400000;
# 定义段
SECTIONS {
    # 将当前地址设为BASE_ADDR
    . = BASE_ADDR;
    # 将大括号内段放入text段
    .text : {
        *(.text.boot)
        *(.text)
        # PROVIDE表示如果未定义etext，则设置etext为当前地址
        PROVIDE(etext = .);
    }
    .data : {
        *(.data .data.* .sdata .sdata.*)
        PROVIDE(edata = .);
    }

    .bss : {
        *(.bss .bss.* .sbss .sbss.*)
        PROVIDE(ebss = .);
    }
    . = KERNEL_END_ADDR;
}
```

在链接脚本中定义的变量可在代码中使用，需要注意如果你自定义了段，连接脚本会默认该段为PROGBITS，这会使得最后链接生成的二进制文件会完整地包含该段的大小，你可以使用`NOLOAD`修饰该段，但是这样会使得该段不会被初始化，所以需要在后续代码中显式初始化。

- **编写入口函数**

在内核链接脚本中，我们可以指定程序的入口函数（通过`ENTRY(<function_name>)`），只需要编写相应的汇编文件即可，在汇编文件中，由于OpenSBI已经为你初始化了BSS段和重要的CSR寄存器，所以你只需要,**为内核设置栈指针**，最后跳转到内核的主函数中。这里同样给出示例代码：

```
.section .text.boot ,"ax",%progbits
; "ax"表示段的属性(a代表可分配，x表示可执行，@progbits表示该段包含程序代码或数据)
.globl _start
_start:
  la  sp,KENREL_END_ADDR
  j  rv_main
```

这里需要注意OpenSBI会传递若干参数给内核，包括：**当前的hartid**(CPU核心ID)，**设备树地址。**

字符输出

这里你只需要在S态利用SBI实现字符的输出即可。具体内容可以参考OpenSBI Calls。之后可以参考原MOS实现`vfmtprint`和`printk`函数，(如果你喜欢的话还可以搞一个LOGO)。

在完成上述内容后，你还需要使用工具链编译链接内核，具体使用的编译参数可以参考RV常见gcc编译选项 (选择rv64的同学请特别注意abi的选择)。在内核编译成功后你可使用QEMU来运行内核:

```
# 64位
qemu-system-riscv64 -kernel <kernel_file> -M virt -m 64 -nographic
# 32位
qemu-system-riscv32 -kernel <kernel_file> -M virt -m 64 -nographic
## -M 指定机器种类，-m表示内存（默认MB单位） -nographic 表示不使用图形界面 -kernel指定内核文件
## -bios 可以指定你自编译的OpenSBI。
```

可选项1-设备树解析

> **注**： 如果未选择该可选项也可以选择硬编码硬件信息的方式。
>
> **设备树**是一种数据结构，用于描述硬件设备的属性和配置以及其之间的关系。它通常以树结构组织，每个节点代表系统中的一个硬件组件，包括CPU,内存,外设等。设备树可由文本形式的**设备树源文件(.dts)**文件描述，该文件可通过编译器转化为**设备树二进制文件(.dtb)**,该文件在系统启动时被内核读取(OpenSBI在启动时通过`a1`寄存器传递了设备树地址)。
>
> **平板设备树(FDT)**是设备树的一种表现形式，它将设备树结构扁平成一个二进制文件，便于读取和解析。

你可以实现对FDT的简易解析以获得内存的分布，外设的地址等内容用于设备的注册和使用(对于不同设备的使用和注册，不妨创建设备和相应的操作抽象，根据解析出的节点，对相应的设备设置相应的操作结构体即可；对于多个设备，则可以采用树进行管理)。设备树的具体内容可以参考：

- 设备树文档1
- 设备树文档2
- FDT文档1
- FDT文档2

可选项2-mem函数优化

你可以优化MOS中`memcpy`与`memset`等函数，优化方法包括但不限于：

- 向量指令(V扩展)
- 数据对齐
- 位操作优化(B扩展)
- Packed SIMD (P扩展)

内存管理

> 你需要将内存分页，开启MMU并且使用页表管理内存。

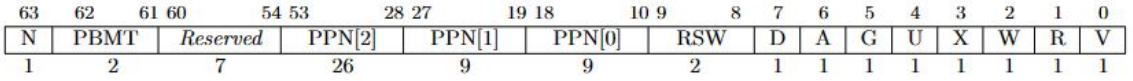
物理内存管理

你需要将内存分页并且实现一种物理内存分配方式。MOS使用链表对页面进行管理并且每次只能分配物理地址连续的一页内存，你可以尝试实现支持同时分配多片物理地址连续的分配方式(事实上这对之后的页表自映射与文件系统驱动的实现**可能都是必要**的)。

虚拟内存管理

RISC-V同样使用多级页表管理虚拟内存，并且对于32/64位有着多种不同的地址映射方案： 在RV32中你可以使用`Sv32`模式,而在RV64中你可以使用`Sv39`,`Sv48`,`Sv57`,`Sv64`模式，这些模式的区别只在于是否使用了更多级的页表来支持更大的内存空间，关于这些模式的具体内容可以参考RISC-V特权级文档第4章的3,4,5节，这里不再对方案细节进行赘述。

RISC-V的页表项类似于(下图为Sv39,对于其他方案权限位也是相同的，权限位的具体内容请同样参考RV特权文档的4.4.1处描述)：



> - 需要注意RISC-V的非叶页表项中如果存在非`PTE_V`权限位，则会触发巨页(megapage)机制。这里以`Sv39`为例(你仍然可以在特权文档中找到不同方案对于巨页的处理)：
>
> > 任何级别的页表项都可以是一个叶页表项，Sv39支持4KB 页，2MB 巨页和1GB 吉页。这些页都必须满足**物理地址和虚拟地址边界对齐其大小**的要求，否则会产生页错误异常。
> >
> > 事实上巨页对于减小页表体积，加速地址翻译是比较有效的。
>
> - 页表权限位中`PTE_D`表示地址代表的内存是否被写入过，而`PTE_A`表示地址是否被访问过。`RWX`分别表示可读，可写，可执行。`PTE_U`表明为是否能在U态被访问 ,需要注意,`D`,`A`和`U`**不能在非叶页表项中出现**，否则会产生叶错误异常。`PTE_G`表示在TLB匹配中不使用ASID,`PTE_V`表示页表项有效。

RISC-V同样可以使用MMU: 在开启MMU前，RISC-V的S态与U态采用地址的直接映射;在开启MMU后，RISC-V的S态与U态所有地址都完全依赖于页表。

RISC-V和MIPS不同的是其使用了硬件来管理地址翻译和TLB处理：

- MMU使用`satp`控制寄存器来设置当前所使用页表和地址映射方案，虚拟地址向物理地址的翻译在设置了satp后由硬件自动进行。

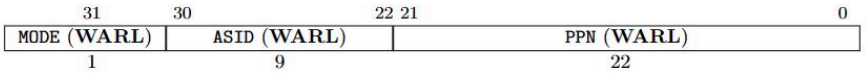
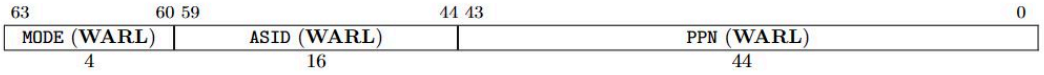


Figure 4.14: Supervisor address translation and protection register satp when SXLEN=32.

Storing a PPN in satp*, rather than a physical address, supports a physical address space larger than 4 GiB for RV32.*

The satp.*PPN field might not be capable of holding all physical page numbers. Some platform standards might place constraints on the values* satp.*PPN may assume, e.g., by requiring that all physical page numbers corresponding to main memory be representable.*



- - `ASID`:用于标识不同地址空间，同时加速TLB翻译。
- `PPN`:页表所在的物理地址高位。
- `MODE`:所使用的地址映射方案，在RV32/RV64中取值有所不同：

  img
- 你可以使用 `sfence.vma` 实现对 TLB 的刷新(可针对某一地址,某一ASID单独刷新)，而 TLB 的重填由硬件自动完成。

你需要实现页表功能，完成对页表的初始化，将内核按照所在的地址属性映射入页表，并且设置`stap`的PTE位和MODE位，从而开启MMU实现虚拟内存管理。

可选项1-移动内核至高地址

低地址空间在操作系统中是十分宝贵的.将内核空间放在高地址空间，不仅可以增强系统的安全性，还可以使得用户空间连续，便于用户程序的地址空间管理。

一种可行的方法是将内核空间直接映射的同时，将其映射到目的高地址空间，然后跳转到目标地址中的某一函数，该函数必须清除源内核地址的映射，并且必须修改返回地址(内核空间一般在最高若干位设置为1，你可以使用位操作设置ra寄存器)。

在移动至高地址后，再使用GDB调试会出现无法访问源代码的问题，你需要修改GDB所加载符号表的基地址到你所迁移到的高地址。

可选项2- 简易kmalloc

> kmalloc是Linux内存管理中的一种内存分配机制，用于分配物理地址连续，虚拟地址连续的小块内存，被用来处理特定要求包括缓冲区的动态分配，PCB和其他资源的创建等。在MOS中所有的资源描述符如Page，Env等通过SLAB分配提前分配在了BSS段中，这其实并不安全。通过将kmalloc所管理的内存映射到某一高地址，可以很好地将内核管理资源和内核所隔离。

如上所述，你可以选取一种内存管理技术(如伙伴系统)，划分出某一段连续物理内存并将其映射到某一地址区域，用于内核特殊要求。

可选项3- 支持巨页

> 巨页内容请参考 本节对页表的描述。

你可以在你实现的页表功能中增加对巨页的支持，并用巨页来映射内核等大内存占用从而提高地址转换效率。你可能需要同时实现支持同等巨页大小物理内存的分配功能。

例外处理

异常

> 注: RISC-V所有的异常和中断其实都默认由M态处理，但是通过控制寄存器`medeleg`和`mideleg`可以将异常和中断委托S态处理。（OpenSBI自动设置了这两个寄存器）

和MIPS类似，RISC-V在S态同样存在着类似功能的状态寄存器`sstatus`：

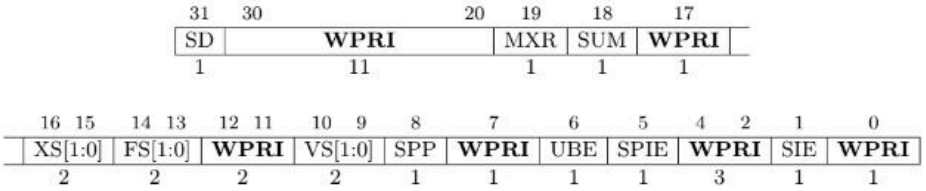


Figure 4.1: Supervisor-mode status register (**sstatus**) when SXLEN=32.

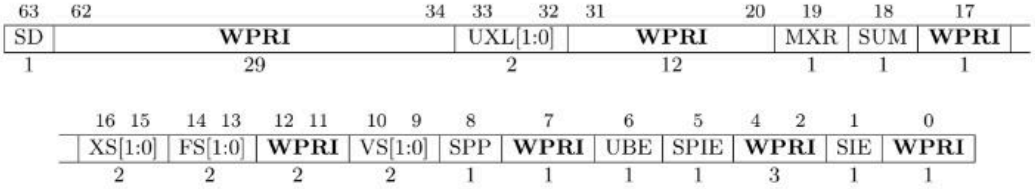


Figure 4.2: Supervisor-mode status register (**sstatus**) when SXLEN=64.

其中有一些比较重要的特征位：

- `SIE`:全局中断使能位。
- `SPIE`:表示进入异常前是否开启了中断。

当异常发生时，机器会将SIE赋与SPIE,而将SIE设为0来达到屏蔽中断的目的。当从异常返回(执行`sret`指令)时，就会复原SIE的值，并将SPIE设置为1。

RISC-V的S态异常入口由控制寄存器`stvec`决定，并且具有直接入口和向量入口两种模式：

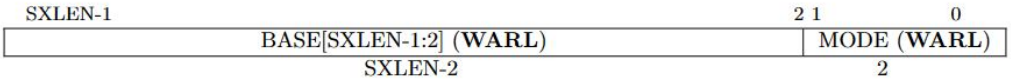


Figure 4.3: Supervisor trap vector base address register (**stvec**).

- `BASE`:异常入口的高位地址，四字节对齐。
- `MODE`:异常入口模式。

Value	Name	Description
0	Direct	All exceptions set pc to BASE.
1	Vectored	Asynchronous interrupts set pc to BASE+4×cause.
≥2	—	*Reserved*

下面是RISC-V的异常表：

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

你可根据需要决定是否实现向量异常处理入口，其他具体内容可以参考RISC-V特权级文档 4.1.2节。

中断

在RISC-V中，主要有控制寄存器`sip`和`sie`控制和响应中断。只有当sie和sip同时设置相应的位，并且控制寄存器`sstatus`中开启了全局中断使能，该中断才能够被响应并处理。在中断响应后，PC就会跳转至stvec寄存器中的和Mode所设置相对应的异常处理入口。

其中，`sip`负责标识当前待处理中断的信息,`SEIP`表示是否存在外部中断，`STIP`表示是否存在时钟中断，`SSIP`表示是否存在软件中断。 SIE中存在和这些状态位相对应的权限位`SSIE`,`STIE`,`SSIE`，分别表示对应位中断的使能。

你可以使用控制寄存器`mtime`和`mtimecmp`实现，`mtime`寄存器会以固定频率递增，并且会在发生溢出时回绕，当其大于`mtimecmp`时，就会由`CLINT`产生时钟中断，在S态你可以通过**SBI调用**来设置时钟中断。

系统调用

在 RISC-V 中，用户态的系统调用对应 8 号例外，你可以在用户态通过 `ecall` 指令陷入内核，并在内核态根据传入的参数选择将要执行的系统调用。

可选项1- 利用Sstc实现S态时钟中断

> RISC-V拥有许多已经成为标准，或者正在成为标准的扩展，`Sstc`扩展增加了S态stimecmp控制寄存器和VS态vstimecmp控制器。利用该拓展，我们可以在S态直接通过寄存器设置时钟中断，不需要通过ecall前往M态处理，这大大节省了中断开销。

之前用于时钟中断的`mtimecmp`控制寄存器只能在M态访问，如果我们所使用的机型支持RISC-V的`Sstc`扩展（该扩展已被批准但还未集成到规范中，QEMU中只有**virt**,**spike**机型支持该拓展，其他机型QEMU并不完善，你可以在OpenSBI的输出或是设备树文件对CPU的描述中看到是否支持Sstc），我们就可以使用该拓展提供的`stimecmp`和`time`控制寄存器来设置时钟中断。

可选项2-中断向量化

在RISC-V中`stvec`寄存器`Mode`位可以设置异常处理的模式，当`Mode`为0时所有的异常都会直接跳转到`stvec`寄存器所设置的异常入口，而到`Mode`域设置为1时，则开启向量化模式：所以的异常会进入异常入口基地址，而中断则会根据中断号直接跳转到`base + 4 * i`（i为中断号）的地址。你可以利用这点来加速中断处理。

> GNU也为中断函数提供了额外的优化，`__attribute__((interrupt("<supervisor>|<machine>|<user>")))`可以用来修饰对应特权的中断函数，编译器会自己生成上下文的保存。

进程与调度

> 目标： 实现对ELF文件的解析，加载；实现进程的创建，运行，调度，销毁。

进程加载

ELF相关信息请参考ELF文档，通过解析ELF文件的程序头表，并将所有`PT_LOAD`类型的段加载到内存中（创建页表，分配物理页，并复制内容），完成进程加载。你可能需要为用户程序编写相应的链接脚本，确保每个段的页对齐以便能够很方便地为相应的段设置正确的权限。

调度

在加载完进程后，可以恢复进程上下文，并切换到进程所使用的页表来实现进程的运行。MOS采用时间片轮转调度算法实现进程调度，设置单一调度队列。每个进程分配一定数量的时间片，每次运行消耗一个时间片。本实验对调度算法没有具体要求，鼓励尝试其他调度算法。

进程切换时需保存上下文，保存的位置取决于你的设计。

可选项1-共享内存池

MOS的进程间共享内存通信基于单页，限于双方进程。考虑实现一个简易的共享内存池，以实现多进程间的不定页数通信（你需要实现简单的锁机制，原子操作请参考RISC-V的A扩展）。

可选项2-多级反馈队列调度

> 不同于完全公平的时间片轮转算法，**多级队列**算法将一个就绪队列拆分为多个，并且将不同类型的进程分配在不同的就绪队列，这些队列可以采用不同的调度算法。**多级反馈队列**为每个队列赋予不同的优先级，优先级越高，时间片越少，一旦用完则会降低优先级到下一级队列，这既能使高优先级进程得到响应又能让短进程迅速完成，被公认为一种较好的调度算法。

你可以尝试实现多级调度队列算法来进行进程调度，关于该算法的内容可以参考教材和多级反馈队列。

可选项3-多核调度

> **SMP**中文名为对称多处理，其中若干个相同的CPU核心共享同一主存，并且具有完全相同的访问权限。SMP有时也被称为UMA(统一内存访问)，和它相对应的是NUMA(非统一内存访问，在该架构中，处理器被分组到不同的节点，并且都有自己的本地内存)。

你需要使用SMP实现能够利用多核运行调度进程,这里以64位为例。

```
qemu-system-riscv32 -smp <core_num> ...
# -smp 指定cpu核数
```

SBI

这里以较新的OpenSBI为例，OpenSBI为CPU核心设计了一套状态机，OpenSBI会使用`lottery`算法随机选取一个核心作为启动核通过`init_coldboot`函数进入内核，其他核心则是通过`init_warmboot`在M态OpenSBI内部初始化后将自身状态修改为`STOPPED`。当你在主核心中初始化资源完成后，就可以通过OpenSBI使用SBI调用来启动从核（你可以通过SBI调用可以操作**所有核心的状态**），进入内核。当核心使用了`wfi`指令陷入低功耗状态时，你可以通过核间中断（软件中断）来唤醒他们。

这里的处理方法很多，你可以在启动时使用`gp`寄存器存入hartid，并且根据是否是启动核做出不同的操作，最后所有核心完全初始化后开始调度。

锁

你可能需要使用RISC-V中`A`拓展的一些原子指令如`awoswap`等用于实现`lock`，并且由于RISC-V的内存模型的是弱内存模型而非Sequence模型，所以你需要注意解锁时的内存屏障，在RISC-V中使用`fence`指令可以做到这一点。

调度

你可以实现全局调度，使用一个队列然后使用每个核心运行其中的进程，或者是实现分区调度，让每个核心拥有自己的队列。

用户库

fork与IPC

> 目标: 实现支持用户态COW机制的fork，实现原MOS中的IPC通信。

你可以通过 RISC-V 中保留的特权位实现 MOS 中的 `PTE_COW` 位与 `PTE_LIBRARY` 位。特别地，MOS 在 RISC-V 架构上运行时不要求实现异常重入，这是因为硬件可以保证原子地执行 TLB 重填。

同样的，你也需要实现原MOS中的IPC通信机制。

文件系统

> 目标: 实现FS(并不强制实现在U态)。

QEMU 中的 Virt 并没有IDE总线所以不能使用原有的IDE硬件协议,我们需要使用`virtio`以实现文件系统，并实现相应的驱动,下面是`virtio`以及`virtio MMIO`以及virtio传输数据用的`virtqueue`介绍:

> Virtio 是一种虚拟化 I/O 设备的标准，旨在提供一个通用的、高效的接口，让客户机操作系统可以与虚拟化环境中的各种设备进行交互。
>
> - Virtio 定义了一系列设备规范，包括网络设备、块设备、控制台等。
> - 客户机操作系统需要实现相应的 Virtio 驱动程序，以识别和控制 Virtio 设备。
> - Virtio 使用基于队列的机制进行数据传输，减少了上下文切换和数据拷贝的开销。
> - 通过使用 Virtio，可以获得接近原生硬件的 I/O 性能。

> Virtio MMIO 是 Virtio 设备规范的一种实现方式，它使用内存映射 I/O（MMIO）来实现 Virtio 设备与客户机操作系统之间的通信。
>
> - 在 QEMU 中，Virtio MMIO 设备被模拟为一个 PCI 设备。
> - 客户机操作系统通过读写设备的 MMIO 寄存器来配置设备、发送命令和传输数据。
> - 每个 Virtio MMIO 设备使用一段连续的内存区域，其中包含了控制寄存器、状态寄存器以及用于数据传输的缓冲区。
> - 相比于传统的 PCI 设备，Virtio MMIO 设备更易于在各种体系结构上实现，特别适用于嵌入式系统和非 x86 架构。

> Virtqueue用于virtio中设备与宿主机的数据传输，由**描述符表**,**可用环**,**已用环**三部分组成。在初始化后，驱动需要将描述符放入可用环中，并对其进行数据处理，完成后通知设备，最后设备通过中断或者特定寄存器的方式通知驱动完成了操作，结果保存在已用环中。

你可以通过在启动 QEMU 时采用 `-M virt,dumpdtb=target/virt.dtb` 参数导出 Virt 的 DTB 文件。然后，你可以通过 `dtc -I dtb -O dts target/virt.dtb target/virt.dts` 命令将 DTB 文件导出我们可读的 DTS 文件。通过观察 DTS 文件，可以发现 Virt 支持 8 个 virtio 设备，你可以通过其挂载虚拟磁盘，类似于：

```
// EXAMPLE: virtio_block device taking 512 bytes at 0x1e000, interrupt 42.
virtio_block@1e000 {
    compatible = "virtio,mmio";
    reg = <0x1e000 0x200>;
    interrupts = <42>;
}
```

为此，你需要将 QEMU 的启动命令修改为(这里以32位为例)：

```
qemu-system-riscv32 $(QEMU_FLAGS) -drive file=<img_file>,if=none,format=raw,id=hd
-device virtio-blk-device,drive=hd
# -drive 创建驱动，指定文件系统file。
# -device创建设备，指定为 virtio-blk-deivce。drive指定所配套的驱动。
```

> 注: 如果你不想实现**Legacy**版本的驱动，你需要添加额外的QEMU参数：`-global virtio-mmio.force-legacy=false`。

上面的参数会在Virt的8个virtio mmio设备中选取一个挂载(通常是0x1000_8000或者是0x1000_0000,如果你实现了设备树解析，你可以尝试创建相应的Virtio设备，并探测其挂载在哪个点)。并且0x1000_0000这些地址所指的是物理地址，你需要将其映射到某一虚拟地址，在映射完成后，你可以访问该外设的所有控制寄存器，你需要完成设备的初始化，并实现扇区的读写。具体的Virtio内容请参考：Virtio文档(ps:初始化请注意**legacy**与**非legacy**的区别)，初始化与数据的读写请参考该文档的**3.1.1节**以及**5.2.5节**，这里不再过多赘述。

在完成virtio驱动后，你便能够读写扇区，你还需要实现一个文件系统，本次任务不会对文件系统的类型和实现方式做出要求，你可以实现在S态或者是U态，可以实现原MOS的文件系统也可以实现其他文件类型的系统。

Shell

你需要实现MOS中的SHELL，并且提供原MOS中相应的指令。

可选项-1 优化FS

> 原MOS中的文件缓冲被设置在0X1000_0000至0X5000_0000,至多支持1GB大小的文件系统，并且缓冲块与文件块是1-1映射，你可以为FS实现动态的文件缓冲以提高FS的灵活性，并且尽可能地提升FS性能，包括但不限于HASH缓冲目录项来避免逐级目录查询，在Page中增加对所缓冲的文件的偏移和目录项指针等，
>
> 同样的，你也可以尝试改进原MOS的文件系统，包括但不限于增加文件最大大小，单独设置Inode和目录结构体缓冲以提高访问效率。或者你也可以选择和LAB5挑战性任务的同学进行沟通，尝试使用他们实现的文件系统。