Rust 移植挑战性任务

任务说明

该任务分为**内核启动**、**内存管理**、**异常处理**以及**进程与调度**四大核心部分。对于每一部分，除了具体明确的要求之外，不会对其他实现细节提出额外要求。完成每个部分后，将提供一些**不计入总分的可选任务**供选择。此任务的最终目标是实现基于MIPS32R2的内核，能够**兼容并加载执行原MOS用户程序**（利用原MOS的C程序，涵盖文件系统和SHELL等内容）。该任务可由**最多两人合作完成**。

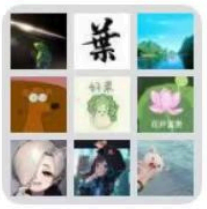
任务要求

- 在提交的代码中必须包含完整可导出的 rust doc 注释。
- 内核能够加载并调度运行原MOS中的C程序。
- 如果是组队完成请同时上传代码和组队信息，说明学号和姓名。
- 通过挑战性任务截止日期后的答辩和检查。

参考资料

- 《使用Rust编写操作系统》
- Rust 基础入门
- Rust 教程
- The Rust Programming Language - The Rust Programming Language

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

工具链

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。

下面是你可能会用到的工具：

- `mipsel-linux-gnu-` 交叉编译工具链。(你可以通过包管理器安装，或者源码编译安装)
- `gdb-multiarch` （安装方式同上）

- `qemu-system-mipsel`（推荐编译安装）

在RUST中你可能需要安装：(你可能需要切换到`nightly`，因为`rust`尚且不支持`mipsel-unknown-none`中`stable`版本的内嵌汇编)：

- `llvm-tools`
- `rust-src`
- `rust-docs`
- `clippy`

兼容原MOS

为了兼容原MOS，你可能不能修改用户态中直接访问的结构体，并且你可能需要使用`extern "C"`来修饰需要和用户态直接接触的函数或是变量来避免rust对其进行命名修饰。

内核启动

> **目标**:使用`qemu-system-mipsel`启动内核，并且实现内核态字符输入与输出。

由于`mipsel-unknown-none`在rust中属于`tier 3`级别，rust并不会提供预编译好的核心库，这里有几种方式处理：

- 使用`-Zbuild-std=`参数自编译核心库。

如`cargo build -Zbuild-std=core,alloc`，该参数需要nightly环境，`-Zbuild-std`会指定需要编译的库。（该方法会使得每次编译都需要重新编译）

- 使用`xbuild`工具。

使用该工具会自动进行核心库的编译，需要通过`cargo install cargo-xbuild`指令进行安装。（该方法只会在clean后的第一次编译时编译核心库）

- 本地指定核心库的位置。

在项目配置文件中指定相应核心库的本地位置，你可能需要修改这其中一些核心库的编译选项。（同xbuild)

在解决了核心库的问题后，我们正式进行mos内核开发，考虑如下入口示例文件`main.rs`：

```
/// main.rs
#![cfg_attr(target_arch = "mips", feature(asm_experimental_arch))]
#![no_std]
#![no_main]
use core::{arch::global_asm,include_str};
extern crate mos_lib;

#[no_mangle]
fn rust_main() {
    // ...
}

global_asm!(include_str!("start.S"));
```

其中有几个重要的属性：

- `#![cfg_attr(condtion,feature)]`：条件编译，如果满足condition，则添加后面的feature。
- `#![no_std]`：程序不使用std库。
- `#![no_main]`:不使用标准的main入口。
- `#[no_mangle]`:不对函数名进行修改（rust默认会对函数名进行修饰）。

rust通过`global_asm!`以及`asm!`宏引入汇编，而`include_str!`宏则引入文件中的字符内容。

在创建完项目后，你需要：

- 编写链接脚本。
- 实现`_start`函数和入口函数。
- 实现`panic`函数以及`print!`和`println!`宏。

关于`print!`以及`println!`宏的实现，你可以查阅`core::fmt`库以及rust中宏定义的相关资料，下面是rust宏定义的简单介绍：

> 在 Rust 中，宏是通过`macro_rules!`宏来定义的，它们允许你写出可匹配不同模式的代码，并在译时展开。这些宏是一种元编程手段，可以用来减少重复代码、实现变参函数。使用`#[macro_export]`可以将宏导出，这样在库中就不需要使用`use`导出再使用。

一个宏定义就像：

```
#[macro_export]
macro_rules! DEFINE_ELF_BYTES {
    ($var_name : ident,$path : literal) => {
        static $var_name: &[u8] = include_bytes!($path);
    };
}
```

宏定义中的`($var_name : ident, $path : literal)`表达了需要匹配的模式，其中一个宏可以尝试匹配多个模式，以此来实现变参功能。在此示例中，匹配的参数为`$var_name`（代表一个变量名）和`$path`（代表一个字面量），这两个参数以逗号`,`分隔。若成功匹配到该模式，则会在宏被调用的位置生成大括号内的代码。

完成上述操作后，你可以尝试使用QEMU启动内核并测试相关功能。

注意事项

- Rust 在编译时会进行多种优化，这可能导致出现预料之外的行为。
- Rust 编译过程中会对变量和函数名进行修饰，因此在 Rust 中通常使用的是`.text.*`、`.data.*`等子段命名，这一点在编写链接脚本时需特别注意。
- 尽量避免使用`asm`形式的内嵌汇编，并且Cargo不会跟踪`global_asm!`引入的汇编文件及其引用的C头文件的变化。
- 本次任务并未指定CPU型号，而是指定了指令集为`MIPS32R2`,可使用24Kc或其他支持MIPS32R2的CPU。
- 所有任务内容由rust实现。

内存管理

> 你需要实现页式内存管理，并实现页表的相关功能。

在MIPS架构中，KUSEG区域的地址需通过TLB转换，TLB由软件管理，通过访问页表来实现重填，这便是页表工作的机制。你需要将MOS使用的内存分页，并利用两级页表来映射32位的4GB空间，同时需设置页表项的权限位，可以参照原始MOS的设置。

可选项1 - GlobalAlloc

> 你需要基于伙伴系统，实现一个内核态的全局堆分配器。

关于伙伴系统的信息，可以参考伙伴系统概述。有关Rust中全局堆的资料，可参考GlobalAlloc。你需实现`alloc`库中的`GlobalAlloc` trait，并使用`#[global_allocator]`属性来标记该变量。

完成此可选项后，你便可以在内核中使用`alloc`库提供的动态类型。

异常处理

> 目标： 实现异常处理,系统调用和时钟中断。

寄存器编号	名称	描述
9	Count	递增的一个计数器——大概每个两个时钟周期自增的计数器位，计数速度由硬件决定。CPU芯片决定计数速度的一个
11	Compare	当count寄存器大于等于该寄存器时就会触发时钟中断
8	BadVAddr	寄存器记录的地址相关异常的地址。(如果和地址相关的话)
4	Context	所有这些都是内存管理/硬件/软件接口（TLB）相关的寄存器
EntryHi		
EntryLo0-1		
Index		
PageMask		
Random		
Wired		

上面是MIPS中CP0协处理器中常用寄存器。

一个异常处理的流程如下：

> - 引导：在异常处理入口，被中断地程序只有非常少的状态被存储起来，所以，第一件事情就是安魂被足够的空间来存储寄存器地状态。通常使用k0和k1寄存器来索引一块内存区域。
> - 构建异常处理环境： 你需要提供一块内存区域用作栈并不能被其他程序占用，处理例程可能用高级编程语言实现。
> - 处理异常：做你想要做的任何事情。
> - 准备返回：高级函数通常作为子例程调用，所以，最终要返回到底层的异常处理程序。在此，恢复存储的寄存器，CPU通过修改SR,设置成安全的模式（内核态，禁止异常），也就是异常发生后的模式。
> - 从异常返回：异常返回的处理，因CPU不同而各异。

下面是MIPS32R2中常用的异常处理号，更多内容可以查阅文档。

ExcCode	助记符	描述
0	Int	中断
1	Mod	写TLB表项中标记为只读的页
2	TLBL	参考指导书中TLB一节
3	TLBS	参考指导书中TLB一节
4	AdEL	地址错误（加载，指令读取或存储操作引起）
5	AdEs	地址错误（加载，指令读取或存储操作引起）
8	Syscall	由syscall执行引起
9	Bp	由break指令执行，被调试器所使用

中断处理

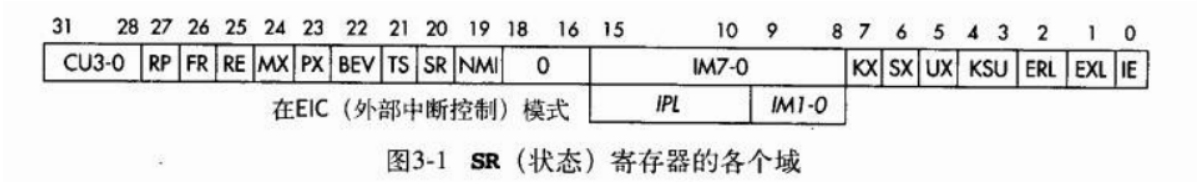


图3-1 **SR**（状态）寄存器的各个域

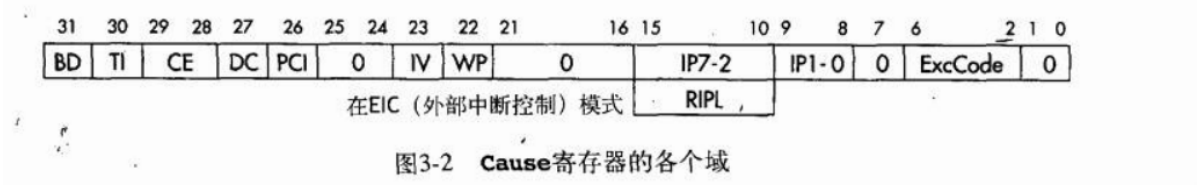


图3-2 **Cause**寄存器的各个域

`sr`寄存器中的`IM`7-0位表示8种中断的使能位，而`IE`位是全局中断使能位。当`cause`寄存器的`IP`对应位被设为1（表示相应的中断发生），并且`sr`寄存器中的`IE`位也被设为1（全局中断使能），同时sr寄存器中对应的`IM`位也为1，系统则会响应并处理该中断。

对于`count/compare`寄存器产生的时钟中断，其中断使能位是`IM7`。在MIPS32R2中，`count`寄存器以固定的频率增加，而`compare`寄存器用于设置下一个时钟中断发生的`count`值。当`count`寄存器的值大于等于`compare`寄存器的值时，会触发一个时钟中断。

系统调用

在用户模式下运行的程序可以通过执行`syscall`指令来进行系统调用，从而实现用户态与内核态之间的交互。系统调用通过传递系统调用号和参数，执行相应的内核操作，并将结果返回给用户态程序。为了与原有的操作系统（MOS）保持兼容，实现系统调用时需要使用原有的系统调用号。

TLB处理

TLB是一种用于加速虚拟地址到物理地址转换的缓存。与TLB相关的控制寄存器主要有：

名称	CP0寄存器号	描述
EntryHi	10	这些寄存器共同存储一个TLB表项所需的所有信息，所有对TLB的读写操作都必须通过它们。EntryHi存储VPN和ASID,在MIPS32/64 CPU中，每个表项映射两个连续的虚拟页到不同的物理页，这两个独立页的PFN和权限标志由EntryLo0和EntryLo1指定。EntryHi(ASID)有双重职责，还负责记住当前活跃的ASID位。 PageMask用来创建大小超过4KB的页表项。
EntryLo0-1	2-3	
PageMask	5	
Index	0	使用适当的指令时决定读或写哪个TLB表项
Random	1	这个伪随机值用于让tlbwr指令写入tlb表项到随机的位置
Context	4	这是很方便的寄存器，用来加速重填TLB,高位可读写，低位则来自不被翻译的VPN地址，如果内存翻译记录 的内存副本使用了合适的安排方式，那么context这样的布置就会指向触发异常地址的页表记录。。

- `entry_hi` 结构如下：

`VPN2` 表示虚拟地址的高位部分(忽略0~13 位，对应着2个lo寄存器)，如果发生重填异常，该寄存器会被自动置位。

`ASID` 负责标识不同地址空间。

- `entry_lo` 结构如下：

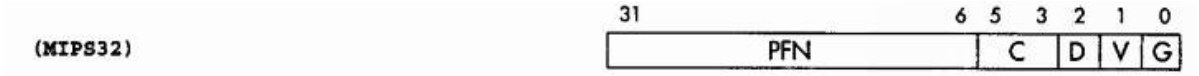


图6-3 EntryLo0-1寄存器域

`PFN` 代表对应物理地址的高位部分。`C` 代表该页可缓存，`D` 代表该页可写，`V` 代表该页有效，而 `G` 代表忽略 `ASID` 域。

以下是几种用于管理TLB的指令：

- `tlbwi`：将 `entryhi` 和两个 `lo` 寄存器的内容写入由 `Index` 寄存器指定的TLB表项中。
- `tlbwr`：使用 `random` 寄存器随机选择一个TLB表项，并将 `entryhi` 和两个 `lo` 寄存器的内容写入其中。
- `tlbp`：搜索TLB以找到与 `entryhi` 寄存器内容匹配的表项，并将相应信息填充到 `Index` 寄存器中。

TLB异常分为三类：

- `TLB MOD`：当尝试对一个需要通过TLB翻译的地址进行写操作时，如果相应的TLB表项中 `PTE_D` 未设置，则会触发此异常。

- `TLBS`：当尝试对一个需要通过TLB翻译的地址进行写操作时，若该地址在TLB中不存在，则触发此异常。
- `TLBL`：当尝试对一个需要通过TLB翻译的地址进行读操作时，若该地址在TLB中不存在，则触发此异常。

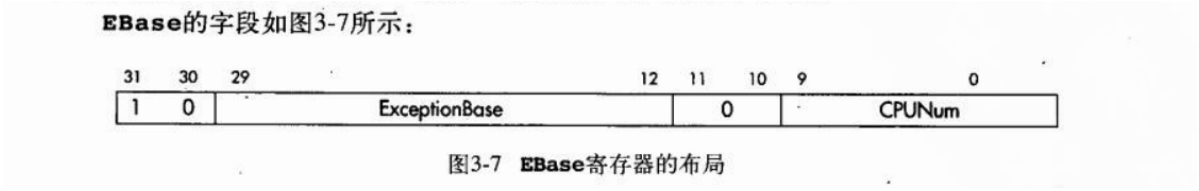
在MIPS32R2架构中，TLB的管理是通过软件实现的。当上述异常之一被触发时，需要通过查看`hi`寄存器中的值来在页表中找到相应的表项，并将其信息填充到两个`lo`寄存器中，最后通过`tlbwr`指令写入TLB。

可选项1-动态异常入口

MIPS32R2的异常处理入口如下：

内存区域	入口地址	异常处理
片上调试	0xFF20_0200	EJTAG调试，映射到探测内存区域时。
0XBFC0_0480	EJTAG调试，用普通ROM内存映射时	
重启(只对ROM)	0XBFC0_0000	重启和NMI入口点
ROM中断入口点(当SR(BEV)为1时)	0XBFC0_0400	中断专用--只用于Cause(IV)设置时，并不是所有CPU都支持
0XBFC0_0380	所有其他异常	
0xBFC_0300	缓存错误	
0XBFC0_0200	简单的TLB重填(SR(EXL)为0时)	
RAM中的入口(当SR(BEV)为0时)	BASE+ 0X200 + ...	多中断入口点，IV模式为7个以上，EIC模式为62个
BASE + 0X2000	特殊的中断(Cause(IV)为1)	
BASE + 0X180	所有其他异常	
BASE + 0X100	缓存错误	
BASE + 0X000	简单的TLB重填(SR(EXL)为0时	

在mips32r2中，默认的异常处理入口（BASE地址）被设置为`0x80000000`。为了动态设置异常处理入口，需要使用协处理器CP0中的`EBase`寄存器。通过修改`EBase`寄存器的值，可以改变异常处理的入口地址，实现对异常处理程序位置的自定义。



- `ExceptionBase`：异常处理入口的基地址，其最高两位为10，表明其必须位于KSEG0内。

- `CPUNum`：此特征位在多核环境中用于标识当前核心。你还可以利用它为不同核心设置各自的异常处理入口。

你可以尝试显式地创建一个异常处理段，并在链接脚本中获取其地址信息。

可选项2-TLB快重填

MIPS架构中，TLB重填是一种频繁触发的异常。为了加速TLB重填，MIPS32R2设置了`context`寄存器。此寄存器需保存当前页表的基地址（必须位于KSEG1内）。每次TLB重填异常触发时，该寄存器会像`hi`寄存器一样设置触发异常的虚拟页号。因此，可以利用此寄存器直接访问页表中的相应页表项。

进程与进程调度

进程加载

相关信息参照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。

运行与调度

加载进程后，通过加载进程上下文并使用`eret`指令，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，MOS用户态产生的异常处理上下文保存在`KSTACKTOP`之下。

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（实现简单的锁机制，借助Rust核心库中的`atomic`等功能可轻松实现）。