中南大學
CENTRAL SOUTH UNIVERSITY

操作系统原理
实验报告

学生姓名	林谦
学　　号	8208200137
专业班级	计科 2005
指导教师	李玺
学　　院	计算机学院
完成时间	2022/6/18

实验内容与目的

批处理系统仅仅能够依次的处理使用者载入的应用程序，没有任何运行时交互，与复杂作业调度的能力。尽管如此，这也是一个初具雏形的操作系统，具备了使用者应用程序的概念，也代表我们要面对**实现特权级切换的挑战**，让内核代码与应用程序在不同特权级下运行，保护硬件与内核的安全。

在第二章要实现的批处理系统中，rCore的应用程序是“静态绑定”，“动态加载”的。静态绑定是指，透过汇编，所有的应用程序的二进制代码会被放入最终的操作系统二进制文件里（实际上是位于内核二进制文件的数据段）。动态加载则是指，在运行过程中，操作系统是透过在内存寻址，将应用程序的二进制内容加载到一个指定的地址并执行。

因此，在实现的过程中，我们首先要先写出可以在rCore上运行的应用程序，将其打包成为二进制文件，并透过构建脚本与汇编将其绑到最终的二进制内核执行文件中。

应用程序

实现

在根目录下透过cargo新建一个项目user，其结构安排如下。

```
user
├── Cargo.lock
├── Cargo.toml
├── Makefile
└── src
    ├── bin
    │   ├── 00hello_world.rs
    │   ├── 01store_fault.rs
    │   ├── 02power.rs
    │   ├── 03priv_inst.rs
    │   └── 04priv_csr.rs
    ├── console.rs
    ├── lang_items.rs
    ├── lib.rs
    ├── linker.ld
    └── syscall.rs
```

Makefile

user中的makefile最重要的功能是，透过cargo build指令将./bin底下的每个rust文件编译为elf，再透过rust-objcopy以及 --strip-all 选项将一些不必要的调试内容去除。

syscall.rs

syscall.rs容许应用程序透过ecall调用S模式下才得以执行的指令。由于rCore的批处理系统应用程序目前只需要很少的功能，syscall.rs底下只对两个系统调用做了封装。 一个是sys_write，将指定的buffer地址与长度，将buffer内容打印到fd参数指定的输出处。 另一个是sys_exit，传入exit_code并退出系统。 至于 ecall 具体做了什么事情，留到后续实现特权级切换的地方做解释。

lang_items.rs

和在OS中一樣，rust要求我們對panic handler進行實現。在实现后，透过调用panic!宏，便会运行由```

```
#[panic_handler]
```

标签所标识的函数,对错误进行处理。

console.rs

在这个文件中，主要对syscall.rs中提供的sys_write进一步进行rust风格的封装，以 print! / println! 宏的形式提供给应用程序调用。

lib.rs

lib.rs是rust约定的包程序入口点。但实际上user包是不会直接被运行的。而是会编译成二进制文件最终被链接进内核。所以lib.rs内实际上提供的是

```
#[no_mangle]
#[link_section = ".text.entry"]
pub extern "C" fn _start() -> ! {
    clear_bss();
    exit(main());
    panic!("unreachable after sys_exit!");
}
```

作为应用程序被调用时的入口点。并且在main函数中透过标签将其设置为弱链接。

```
#[linkage = "weak"]
#[no_mangle]
fn main() -> i32 {
    panic!("Cannot find main!");
}
```

如此一来，当应用程序执行时，便会优先将非弱链接的main（也就是接下来应用程序实现的main)当作执行入口。 此外，在_start中还调用了 clear_bss， 这个函数和之前os实现的同名函数是一样的。

bin/*.rs

在rust对project结构的约定中，若一个project需要分别编译成多个二进制文件，则需要在project根目录下的bin文件夹内分别实现。 以 00hello_world.rs 为例，其实现如下

```
#![no_std]

#![no_main]



#[macro_use]

extern crate user_lib;



#[no_mangle]

fn main() -> i32 {

    println!("Hello, world!");

    0

}
```

其透过 user_lib 标识符引入以lib.rs为入口的整个user包。标识符则是在 ./Cargo.toml 里面定义的

```
[package]
name = "user_lib"
```

如此一来，每个bin底下实现的应用程序就可以调用user包所提供的所有功能，并且被分别编译成各个二进制文件。

link.ld

在v3教程中，对这个文件给出的解释如下

在 user/.cargo/config 中，我们和第一章一样设置链接时使用链接脚本 user/src/linker.ld 。在其中我们做的重要的事情是：

1. 将程序的起始物理地址调整为 0x80400000 ， 三个应用程序都会被加载到这个物理地址上运行；
2. 将 _start 所在的 .text.entry 放在整个程序的开头，也就是说批处理系统只要在加载之后跳转到 0x80400000 就已经进入了 用户库的入口点，并会在初始化之后跳转到应用程序主逻辑；
3. 提供了最终生成可执行文件的 .bss 段的起始和终止地址，方便 clear_bss 函数使用。

这边要注意的一件事情便是，`0x80400000` 并不是应用程序存放的地址。所有应用程序的二进制数据会先被放在内核的数据段中，在要运行时才被复制到 `0x80400000`运行。而设置这个地址后，我们在内核依然要编程手动实现前面所说的这个动作，在链接脚本中的设置只是为了确保应用程序中的绝对地址能够指向正确的代码和数据,请参考

到这一步时，每个二进制文件便已经是可以被执行的应用程序了。我们可以透过

```
qemu-riscv64 <filename>
```

指令来执行。

执行应用程序

涉及到的文件(以 * 标记)

```
./os

├── Cargo.lock
├── Cargo.toml *
├── Makefile *
├── build.rs *
└── src
    ├── batch.rs *
    ├── console.rs
    ├── entry.asm
    ├── lang_items.rs
    ├── link_app.S *
    ├── linker-k210.ld
    ├── linker-qemu.ld
    ├── main.rs
    ├── sbi.rs
```

```
    ├── sync
    │   ├── mod.rs
    │   └── up.rs *
    ├── syscall
    │   ├── fs.rs
    │   ├── mod.rs
    │   └── process.rs
    └── trap
        ├── context.rs
        ├── mod.rs
        └── trap.S
```

Makefile

在rCore仓库中，os内的Makefile看上去很复杂，实际上是因为这个Makefile处理了编译到qemu和k210的两种不同编译流程，以及在过程中对依赖问题做了解决。实际上 `make run`也不过是执行了构建project,去除调试信息，将内核加载入qemu并从指定地址开始运行等过程。

build.rs & link_app.S

在应用程序实现了之后，我们便可以开始修改内核以运行应用程序。rCore（邓式鱼）透过cargo的build script （build.rs）实现了根据 ../user/bin 底下的文件数量，于运行建构指令时生成所需的 link_app.s 汇编代码的功能。以下是 link_app.s 的代表性的一段

```
    .align 3
    .section .data
    .global _num_app
_num_app:
    .quad 5
    .quad app_0_start
    .quad app_1_start
    .quad app_2_start
    .quad app_3_start
    .quad app_4_start
    .quad app_4_end

    .section .data
    .global app_0_start
    .global app_0_end
app_0_start:
    .incbin "../user/target/riscv64gc-unknown-none-elf/release/00hello_world.bin"
app_0_end:
```

（.align的作用是啥？删掉之后似乎没有影响） 在_num_app标识符下，我们可以看到其记录了应用程序的数目，接着是每个应用程序的二进制内容的位置。 透过这些内容，我们便可以在内核代码中实现动态加载应用程序的功能。这段代码会在 os/main.rs 下引入

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

接着，我们需要实现一下所需的工具。

up.rs

UPSafeCell

“邓式鱼”批处理系统需要一个可以于内核全局存取的 **静态可变实例** 来维护目前运行的应用程序状态，对大部分语言来说，这不是什么麻烦的问题，但在拥有**所有权模型**的rust底下是一个难题，当然，这是为了内存安全所付出的代价。 粗略的来说，rust的所有权模型限制了我们在安全模式下简单声明一个静态可变实例，我们需要透过 RefCell 结构体对这些不安全行为做封装,将借用检查放到运行时而非编译时进行。对于RefCell提供的功能，可以用《 Rust 程序设计语言》的内容来解释

参考 不安全的rust refCell与内部可变性模式

> 当创建不可变和可变引用时，我们分别使用 `&` 和 `&mut` 语法。对于 `RefCell<T>` 来说，则是 `borrow` 和 `borrow_mut` 方法，这属于 `RefCell<T>` 安全 API 的一部分。`borrow` 方法返回 `Ref<T>` 类型的智能指针，`borrow_mut` 方法返回 `RefMut<T>` 类型的智能指针。这两个类型都实现了 `Deref`，所以可以当作常规引用对待。
>
> `RefCell<T>` 记录当前有多少个活动的 `Ref<T>` 和 `RefMut<T>` 智能指针。每次调用 `borrow`，`RefCell<T>` 将活动的不可变借用计数加一。当 `Ref<T>` 值离开作用域时，不可变借用计数减一。就像编译时借用规则一样，`RefCell<T>` 在任何时候只允许有多个不可变借用或一个可变借用。

而“邓式鱼”批处理系统是单核的，在任意时刻，只有唯一一处会需要读写系统的应用程序状态，因此我们可以进一步做封装，确保其在单核上运行的安全性。

```
use core::cell::{RefCell, RefMut};
pub struct UPSafeCell<T> {
    inner: RefCell<T>,
}
unsafe impl<T> Sync for UPSafeCell<T> {}
impl<T> UPSafeCell<T> {
    pub unsafe fn new(value: T) -> Self {
```

```
            Self {
                inner: RefCell::new(value),
            }
        }
        pub fn exclusive_access(&self) -> RefMut<'_, T> {
            self.inner.borrow_mut()
        }
    }
```

这边在 rCore-tutorial-Book-v3中，有那么一段内容

> 首先 new 被声明为一个 unsafe 函数，是因为我们希望使用者在创建一个 UPSafeCell 的时候保证在访问 UPSafeCell 内包裹的数据的时候始终不违背上述模式：即访问之前调用 exclusive_access ，访问之后销毁借用标记再进行下一次访问。这只能依靠使用者自己来保证，但我们提供了一个保底措施：当使用者违背了上述模式，比如访问之后忘记销毁就开启下一次访问时，程序会 panic 并退出。

或是是我理解有误，但就算不将new函数声明为unsafe，仅仅是exclusive_access的实现就确保了若引用未销毁就开启下一次访问，便会导致panic的功能（因为对refCell来说，只能同时允许一个borrow_mut引用，而我们只允许了外部使用borrow_mut来访问实例，所以我在rCore教程有留言，而维护者给出了解释。

至此，我们便可以在安全模式下，透过UPSafeCell实现一个单核安全的静态全局可变实例，我们可以将其标记未Sync,避开rust编译器的检查。

batch.rs

接着，便可以写一个AppManager结构体来描述我们的应用程序状态实例，并将批处理系统所需的一些方法封装

```
const MAX_APP_NUM: usize = 16;
const APP_BASE_ADDRESS: usize = 0x80400000;
const APP_SIZE_LIMIT: usize = 0x20000;
//...
struct AppManager {
        //应用程序总数
    num_app: usize,
    //当前应用程序于app_start数组的下标索引
    current_app: usize,
    //所有应用程序代码的地址
    app_start: [usize; MAX_APP_NUM + 1],
}

impl AppManager {
    pub fn print_app_info(&self) {
        println!("[kernel] num_app = {}", self.num_app);
```

```
        for i in 0..self.num_app {
            println!(
                "[kernel] app_{} [{:#x}, {:#x})",
                i,
                self.app_start[i],
                self.app_start[i + 1]
            );
        }
    }

    unsafe fn load_app(&self, app_id: usize) {
        if app_id >= self.num_app {
            panic!("All applications completed!");
        }
        println!("[kernel] Loading app_{}", app_id);
        // clear icache
        asm!("fence.i");
        // clear app area
        core::slice::from_raw_parts_mut(APP_BASE_ADDRESS as *mut u8,
APP_SIZE_LIMIT).fill(0);
        // 由于我们没有保存每个应用程序的长度，所以只能通过下一个地址 - 当前地址取得长度，
        //并传入from_raw_parts中
        let app_src = core::slice::from_raw_parts(
            self.app_start[app_id] as *const u8,
            self.app_start[app_id + 1] - self.app_start[app_id],
        );
        //此时，app_src便是我们要载入的应用程序的地址的slice引用，以u8为一个单位。
        //所以app_src.len()便是应用程序长度了。
        //现在以APP_BASE_ADDRESS为起点，创建一个长度为app_src.len()的slice。
        let app_dst = core::slice::from_raw_parts_mut(APP_BASE_ADDRESS as *mut u8,
app_src.len());
        //将应用程序内容从原本的位置（内核数据段 ）复制到 （应用程序运行段）
        app_dst.copy_from_slice(app_src);
    }

    pub fn get_current_app(&self) -> usize {
        self.current_app
    }

    pub fn move_to_next_app(&mut self) {
        self.current_app += 1;
    }
}
```

对 load_app我整理出了一些问题与答案:

1. 为什么对指针的转换是 u8呢？u8代表的是8bit无符号整型，而8bit正好构成一个字节（byte)。现代计算机的字长一般都是8bit的整数倍，所以按照8bit读取内存可以兼容大部分计算机的主存储器字长。
2. 在 load_app中，有些语句让我有点费解。其中一个便是 asm!(fence.i)。这在教学中给出了详细的解释。

> 注意第 7 行我们插入了一条奇怪的汇编指令 `fence.i` ，它是用来清理 i-cache 的。我们知道缓存是存储层级结构中提高访存速度的很重要一环。而 CPU 对物理内存所做的缓存又分成 **数据缓存** (d-cache) 和 **指令缓存** (i-cache) 两部分，分别在 CPU 访存和取指的时候使用。在取指的时候，对于一个指令地址， CPU 会先去 i-cache 里面看一下它是否在某个已缓存的缓存行内，如果在的话它就会直接从高速缓存中拿到指令而不是通过总线访问内存。通常情况下， CPU 会认为程序的代码段不会发生变化，因此 i-cache 是一种只读缓存。但在这里，OS将修改会被 CPU 取指的内存区域，这会使得 i-cache 中含有与内存中不一致的内容。因此OS在这里必须使用 `fence.i` 指令手动清空 i-cache ，让里面所有的内容全部失效，才能够保证CPU访问内存数据和代码的正确性。

其中 `OS将修改会被 CPU 取指的内存区域` 指的便是 `app_dst.copy_from_slice(app_src);` 这个语句所作的事情。其从指定的app_src地址将应用程序的二进制代码复制到了app_dst地址为起点的空间，修改了实际上 `APP_BASE_ADDRESS`地址所指向的内存空间的内容。

3. 虽然语义十分的明确。但我依然好奇rust提供的 `from_raw_parts`做了哪些工作。这个地方我参考了rust官方文档，以及rust的 `as`是怎么在地址和指针中做转换的，我参考了这里。

最后，透过lazy_static宏将其实例化为全局引用APP_MANAGER。

```
use lazy_static::*;
//...
const MAX_APP_NUM: usize = 16;
//...
lazy_static! {
    static ref APP_MANAGER: UPSafeCell<AppManager> = unsafe {
        UPSafeCell::new({
            extern "C" {
            //在link_app.s中给出的标识符，其是一个地址，指向保存应用程序数量的内存
                fn _num_app();
            }
            //将_num_app的值转为裸指针
            let num_app_ptr = _num_app as usize as *const usize;
            //从裸指针指向的地址读取应用程序总数的值
            let num_app = num_app_ptr.read_volatile();
            let mut app_start: [usize; MAX_APP_NUM + 1] = [0; MAX_APP_NUM + 1];
            let app_start_raw: &[usize] =
            //还记得在link_app.的_num_app标识符其后，先是应用程序数量的值，再来就是第0个应用程序的起始地址
                core::slice::from_raw_parts(num_app_ptr.add(1), num_app + 1);
            app_start[..=num_app].copy_from_slice(app_start_raw);
            AppManager {
                num_app,
                current_app: 0,
                app_start,
            }
```

```
        })
    };
}
```

写到这边的时候，我对ref关键字有点疑惑，实际上这和使用&创建的引用是等价的，可以参考此处。

同时，这里使用了lazy_static! 宏包。rCore教程中对此做了解释

> lazy_static! 宏提供了全局变量的运行时初始化功能。一般情况下，全局变量必须在编译期设置一个初始值，但是有些全局变量依赖于运行期间才能得到的数据作为初始值。这导致这些全局变量需要在运行时发生变化，即需要重新设置初始值之后才能使用。如果我们手动实现的话有诸多不便之处，比如需要把这种全局变量声明为 static mut 并衍生出很多 unsafe 代码 。这种情况下我们可以使用 lazy_static! 宏来帮助我们解决这个问题。这里我们借助 lazy_static! 声明了一个 AppManager 结构的名为 APP_MANAGER 的全局实例，且只有在它第一次被使用到的时候，才会进行实际的初始化工作。

这也代表着我们必须在os底下的Cargo.toml引入这个依赖

```
[dependencies]
lazy_static = { version = "1.4.0", features = ["spin_no_std"] }
```

read_volatile则是rust对裸指针提供的方法，允许对易失性内存指针的读取。 至此，批处理操作系统已经可以载入我们写好的应用程序了，接着需要实现的是特权级机制。

实现特权级的切换

涉及到的文件(以 * 标记)

```
./os

├── Cargo.lock
├── Cargo.toml
├── Makefile
├── build.rs
└── src
    ├── batch.rs *
    ├── console.rs
    ├── entry.asm
    ├── lang_items.rs
```

```
├── link_app.S
├── linker-k210.ld
├── linker-qemu.ld
├── main.rs
├── sbi.rs
├── sync
│   ├── mod.rs
│   └── up.rs
├── syscall
│   ├── fs.rs
│   ├── mod.rs
│   └── process.rs
└── trap
    ├── context.rs *
    ├── mod.rs *
    └── trap.S *
```

接着，需要为执行应用程序实现特权级切换的功能。在此之前，我内心便一直有个问题： 内核是运作在什么特权级下的？教程内做了解答

> 我们知道，批处理操作系统被设计为运行在内核态特权级（RISC-V 的 S 模式），这是作为 SEE（Supervisor Execution Environment）的 RustSBI 所保证的。

而应用程序需要透过特权级切换做到的事情如下：

> - 当启动应用程序的时候，需要初始化应用程序的用户态上下文，并能切换到用户态执行应用程序；
> - 当应用程序发起系统调用（即发出 Trap）之后，需要到批处理操作系统中进行处理；
> - 当应用程序执行出错的时候，需要到批处理操作系统中杀死该应用并加载运行下一个应用；
> - 当应用程序执行结束的时候，需要到批处理操作系统中加载运行下一个应用（实际上也是通过系统调用 `sys_exit` 来实现的）。

那么，我们特权级切换的流程又是怎么样的呢？粗略来说的步骤如下：

1. 内核首先在 `stvec` 设置 trap 处理代码的入口地址
2. 应用程序透过 `ecall`指令向内核发起系统调用。此时

> 1. `sstatus` 的 `SPP` 字段会被修改为 CPU 当前的特权级（U/S）。 2. `sepc` 会被修改为 Trap 处理完成后默认会执行的下一条指令的地址。 3. `scause/stval` 分别会被修改成这次 Trap 的原因以及相关的附加信息。 4. CPU 会跳转到 `stvec` 所设置的 Trap 处理入口地址，并将当前特权级设置为 S ，然后从Trap 处理入口地址处开始执行。

3. 由内核保存应用程序所需的上下文

4. 透过 `ecall`的附加信息，处理需要在内核态执行的作业
5. 内核恢复应用程序上下文（主要是通用寄存器和栈，还有一些有影响的 csr ），设置 `sstatus`字段为返回的特权级
6. 内核调用 `sret`，使cpu跳转回sepc指向的指令

备注

1. **trap:由系统调用产生的中断**
2. **csr: 控制状态寄存器(CSR, Control and Status Register)**
3. 和trap 有关的 csr ： | csr | 功能 | | :------ | :---------- | | sstatus | `SPP` 等字段给出 Trap 发生之前 CPU 处在哪个特权级（S/U）等信息 | | sepc | 当 Trap 是一个异常的时候，记录 Trap 发生之前执行的最后一条指令的地址 | | scause | 描述 Trap 的原因 | | stval | 给出 Trap 附加信息 | | stvec | 控制 Trap 处理代码的入口地址 |

那么，为了要能够保存和复原应用程序信息，我们将其保存在内核栈中，并另外提供应用程序所使用的用户栈。

用户栈与内核栈

用户栈和内核栈的区分主要是为了安全性，在教程中有指明

> 使用两个不同的栈主要是为了安全性：如果两个控制流（即应用程序的控制流和内核的控制流）使用同一个栈，在返回之后应用程序就能读到 Trap 控制流的历史信息，比如内核一些函数的地址，这样会带来安全隐患。于是，我们要做的是，在批处理操作系统中添加一段汇编代码，实现从用户栈切换到内核栈，并在内核栈上保存应用程序控制流的寄存器状态。

对于两个结构体，都实现了 `get_sp`方法来取得栈顶地址。当需要换栈的时候，将 `sp`寄存器（栈寄存器）的内容修改为 `get_sp`的返回值即可。

```
// batch.rs

#[repr(align(4096))]
struct KernelStack {
    data: [u8; KERNEL_STACK_SIZE],
}

#[repr(align(4096))]
struct UserStack {
    data: [u8; USER_STACK_SIZE],
}
```

```
static KERNEL_STACK: KernelStack = KernelStack {
    data: [0; KERNEL_STACK_SIZE],
};

static USER_STACK: UserStack = UserStack {
    data: [0; USER_STACK_SIZE],
};

impl KernelStack {
    fn get_sp(&self) -> usize {
        self.data.as_ptr() as usize + KERNEL_STACK_SIZE
    }
}

impl UserStack {
    fn get_sp(&self) -> usize {
        self.data.as_ptr() as usize + USER_STACK_SIZE
    }
}
```

Trap 管理

Trap 上下文

trap 上下文和函数调用上下文类似，其用来保存trap切换时需要保存的物理资源内容。

```
// trap/context.rs

#[repr(C)]
pub struct TrapContext {
    pub x: [usize; 32],
    pub sstatus: Sstatus,
    pub sepc: usize,
}
```

其中 `repr`主要是控制 `TrapContext`的对齐，但是我还不清楚这个对齐的必要性是什么（删除了一般不出问题）。 具体而言，`TrapContext` 保存了 `x0~x31`等通用寄存器在 `x` 数组，以及 `sstatus`和 `sepc`两个 `csr`。通用寄存器的保存自然是为了控制流的上下文，不过保存了所有通用寄存器的主要原因是，执行Trap处理相关代码是依然会直接或间接调用很多模块，很难确定哪些通用寄存器会被修改，不如全部保存来的方便。 而 `sstatus`和 `spec`之所以要保存，则是因为其和其他的 `csr`不同，在Trap控制流的过程全程都有意义，而且 `sret`还将依赖 `spec`保存的最后执行位置返回。

Trap 上下文的保存与恢复

在实现Trap分发与处理函数，（定为 trap_handler)之前，我们必须先透过汇编实现保存Trap上下文的功能，代码如下

```
.macro SAVE_GP n
    sd x\n, \n*8(sp)
.endm
.macro LOAD_GP n
    ld x\n, \n*8(sp)
.endm
    .section .text
    .globl __alltraps
    .globl __restore
    .align 2
__alltraps:
    csrrw sp, sscratch, sp
    addi sp, sp, -34*8
    sd x1, 1*8(sp)
    sd x3, 3*8(sp)
    .set n, 5
    .rept 27
        SAVE_GP %n
        .set n, n+1
    .endr
    csrr t0, sstatus
    csrr t1, sepc
    sd t0, 32*8(sp)
    sd t1, 33*8(sp)
    csrr t2, sscratch
    sd t2, 2*8(sp)
    mv a0, sp
    call trap_handler
```

接着在 trap/mod.rs内透过外部符号引入 __alltraps过程，并透过设置 stvec来将其指向 __alltraps作为 trap入口点。

```
use core::arch::global_asm;
use riscv::register::{
    mtvec::TrapMode,
    scause::{self, Exception, Trap},
    stval, stvec,
};

global_asm!(include_str!("trap.S"));

pub fn init() {
    extern "C" {
        fn __alltraps();
    }
    unsafe {
        stvec::write(__alltraps as usize, TrapMode::Direct);
    }
}
```

接着，我们需要实现 trap_handler返回后，所需要调用的回复上下文的过程。

```
__restore:
    # case1: start running app by __restore
    # case2: back to U after handling trap
    mv sp, a0
    # now sp->kernel stack(after allocated), sscratch->user stack
    # restore sstatus/sepc
    ld t0, 32*8(sp)
    ld t1, 33*8(sp)
    ld t2, 2*8(sp)
    csrw sstatus, t0
    csrw sepc, t1
    csrw sscratch, t2
    # restore general-purpuse registers except sp/tp
    ld x1, 1*8(sp)
    ld x3, 3*8(sp)
    .set n, 5
    .rept 27
        LOAD_GP %n
        .set n, n+1
    .endr
    # release TrapContext on kernel stack
    addi sp, sp, 34*8
    # now sp->kernel stack, sscratch->user stack
    csrrw sp, sscratch, sp
    sret
```

Trap分发与处理

接着，我们便可以回到 rust来实现Trap的分发与处理功能。

```
use riscv::register::{
    mtvec::TrapMode,
    scause::{self, Exception, Trap},
    stval, stvec,
};
//...
#[no_mangle]
pub fn trap_handler(cx: &mut TrapContext) -> &mut TrapContext {
    let scause = scause::read();
    let stval = stval::read();
    match scause.cause() {
        Trap::Exception(Exception::UserEnvCall) => {
            cx.sepc += 4;
            cx.x[10] = syscall(cx.x[17], [cx.x[10], cx.x[11], cx.x[12]]) as usize;
        }
        Trap::Exception(Exception::StoreFault) |
Trap::Exception(Exception::StorePageFault) => {
            println!("[kernel] PageFault in application, kernel killed it.");
```

```
                run_next_app();
            }
            Trap::Exception(Exception::IllegalInstruction) => {
                println!("[kernel] IllegalInstruction in application, kernel killed
it.");
                run_next_app();
            }
            _ => {
                panic!(
                    "Unsupported trap {:?}, stval = {:#x}!",
                    scause.cause(),
                    stval
                );
            }
        }
        cx
    }
```

为了能够在汇编中透过 `call`调用，我们同样要使用 `no_mangle`宏避免其对函数名进行混肴。这段代码依赖了 `rust`的 `riscv`库对 `scause`寄存器所保存的Trap原因进行分发处理，因此需要在 `os/Cargo.toml`内进行修改来引入依赖。

```
[dependencies]
riscv = { git = "https://github.com/rcore-os/riscv", features = ["inline-asm"] }
```

分发过程则涉及到 `rust`的 `match`控制流运算符，可以参考这里。其中前两个case语义都是比较明确的，而 `_`则和 `C`中 `switch`的 `default`类似，当没有匹配case的时候便会跳转到该分支。 至此，关于 Trap 处理的部分便大致完成。