

# 操作系统课程实验报告

## rcore 第二章

陈林峰 1120192739

2021 年 11 月 26 日

### 1 实验环境

1. OS: Ubuntu 20.04 focal(on the Windows Subsystem for Linux)
2. Kernel: x86\_64 Linux 5.10.16.3-microsoft-standard-WSL2
3. Rust: rustc 1.58.0-nightly (936f2600b 2021-11-22)

### 2 实验目的

完成能够进行批处理的系统，既可以动态将应用程序加载到内存的指定位置，当执行完当前程序后会自动加载下一个应用程序继续执行。完成特权级的转换，将应用程序的执行环境和内核代码区分开来，尽量减少应用程序对内核代码的影响。

### 3 实验步骤

1. 编写多个应用程序，增加系统调用，修改应用程序的链接脚本调整应用程序的内存布局。
2. 通过将应用程序从 ELF 执行格式转换为 Binary 格式，并使用一个汇编文件生成其辅助信息，用于内核找到其位置然后拷贝到指定位置。
3. 实现应用程序的 Trap 上下文保存，使得应用程序可以通过系统调用进入内核态，执行完相应功能后回到用户态。
4. 编写应用管理器，管理应用程序的执行和资源分配，包括用户栈和内核栈。

### 4 实验内容

#### 4.1 为什么需要特权级机制

如果将应用程序与操作系统内核代码放在一起执行，那么带来的后果就是应用程序可以任意的访问本该只有内核才能访问的内容，而且应用程序还可以篡改系统的各种功能，这就会导致内核的不安全性。而如果将应用程序与内核在不同特权级上执行，就可以避免应用程序使用内核才可以使用的指令，应用程序必须通过特权级转换，才能使用内核为其提供的功能。

## 4.2 RISC-V 特权级架构

risc-v 提供了 4 中特权级，但实验中只涉及到 S 和 U 特权级，M 特权级已经由 RustSBI 实现。应用程序运行于 U 特权级，内核运行于 S 特权级，因此当用户程序执行时产生一些错误或者向系统提出请求时就需要暂停应用程序的执行，转到 S 特权级的去执行相应的代码，这是一种不同于传统的函数调用的异常控制流，因为传统的调用不涉及到特权级的切换，而异常控制流需要做更多工作。在 risc-v 体系下有不同的异常产生原因，包括缺页错误，非法访问，系统调用，断点等。这一节的实验主要关注系统调用 ecall，这是应用程序可以向内核请求资源和功能的关键。下图是一个应用程序通过 ecall 进入内核的示意图。

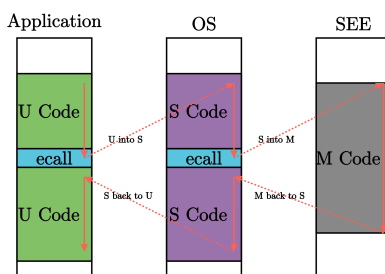


图 1: ecall 系统调用

### 4.2.1 用户态应用程序

这里编写了 3 个应用程序，第一个在屏幕上打印字符串并尝试调用一个 s 特权级的指令。第二个尝试访问一个非法的物理地址，第三个不断进行计算并打印结果。为了实现打印功能，需要向内核发出打印字符的系统调用，为了程序能正常退出，需要向内核发出退出的系统调用。按照前面内核代码的设计，我们也需要告诉编译器禁止使用标准库，并重新定义程序入口，还需要指定编译程序时的链接脚本，这里需要将起始地址硬编码到 0x80400000，这是我们加载应用程序的地址。下面的代码是用户程序的系统调用。

```
const SYSCALL_EXIT:usize = 93;
const SYSCALL_WRITE:usize = 64;

/// 功能：将内存中缓冲区中的数据写入文件。
/// 参数：'fd' 表示待写入文件的文件描述符；
///       'buf' 表示内存中缓冲区的起始地址；
///       'len' 表示内存中缓冲区的长度。
/// 返回值：返回成功写入的长度。
/// syscall ID: 64

pub fn sys_write(fd:usize,buffer:&[u8])->isize{
    syscall(SYSCALL_WRITE,[fd,buffer.as_ptr() as usize,buffer.len()])
}

/// 功能：退出应用程序并将返回值告知批处理系统。
/// 参数：'xstate' 表示应用程序的返回值。
/// 返回值：该系统调用不应该返回。
/// syscall ID: 93
pub fn sys_exit(state:i32) ->isize{
```

```

        syscall(SYSCALL_EXIT,[state as usize,0,0])//执行退出
    }

fn syscall(id: usize, args: [usize; 3]) -> isize {
    let mut ret: isize;
    unsafe {
        asm!("ecall",
            inlateout("x10") args[0] => ret,
            in("x11") args[1],
            in("x12") args[2],
            in("x17") id,
            options(nostack)
        )
    }
    ret
}

```

---

#### 4.2.2 批处理操作系统设计

##### 4.2.3 应用程序管理

1. 将应用程序的二进制镜像文件作为内核的数据段链接到内核中，内核需要知道应用程序的数量以及各个应用程序的位置。对于应用程序的二进制文件信息，我们使用一个 build.rs 脚本生成，其会读取已经处理过的用户程序的二进制数据，并将其嵌入到汇编代码中，而汇编代码也会被嵌入进内核代码中。产生的汇编代码如下：

```

.align 3
.section .data
.global _num_app
_num_app:
.quad 3
.quad app_0_start #quad是一个8字节的伪指令
.quad app_1_start #这里标识了有3个应用程序，并依次放置了3个应用程序二进制数据的起始位置
.quad app_2_start
.quad app_2_end

.section .data
.global app_0_start
.global app_0_end
app_0_start:##第一个应用程序的二进制数据
.incbin "../user/target/riscv64gc-unknown-none-elf/release/00hello.bin"
app_0_end:
.....

```

---

2. 管理各个应用程序的进入与离开，我们使用一个 AppManager 结构体管理，并对其进行全局初始化。

```

struct AppManager {

```

```

    inner: RefCell<AppManagerInner>,
}
struct AppManagerInner {
    num_app: usize, //app数量
    current_app: usize, //当前的app
    app_start: [usize; MAX_APP_NUM + 1], //app的起始地址
}

lazy_static! {
    static ref APP_MANAGER: AppManager = AppManager{
        inner: RefCell::new({
            extern "C" { fn _num_app(); }
            //取出app所在位置的地址
            //link_apps按8字节对其
            let num_app_ptr = _num_app as usize as *const usize; //取地址
            let num_app = unsafe{ num_app_ptr.read_volatile() }; //读内容 =3
            let mut app_start: [usize; MAX_APP_NUM+1] = [0; MAX_APP_NUM+1];
            let app_start_raw: &[usize] = unsafe{
                //形成一个指针切片, 存放的三个应用的起始地址和最后一个应用的开始地址
                from_raw_parts(num_app_ptr.add(1), num_app+1)
            };
            app_start[..=num_app].copy_from_slice(app_start_raw); //复制地址
            AppManagerInner{
                num_app,
                current_app: 0,
                app_start,
            } //初始化
        }),
    };
}

```

---

3. 初始化完管理器, 就可以加载应用程序, 这里需要注意的是 fence.i 指令, 其负责清除 cpu 中的指令缓存, 由于我们切换了应用程序, 其各个地址上的指令已经不是原来的了, 如果不清除缓存里面的数据, 那么在下次访问某个地址上的指令时就会发生执行上一个应用程序指令的情况。

---

```

unsafe fn load_app(&self, app_id: usize) {
    if app_id >= self.num_app {
        panic!("All application completed!"); //保证不会加载不存在的应用程序
    }
    println!("[kernel] Loading app-{}", app_id);
    //重要 clear i-cache
    asm!(
        "fence.i",
        options(nostack)
    );
    //清除应用程序段
    (APP_BASE_ADDRESS..APP_BASE_ADDRESS + APP_SIZE_LIMIT).for_each(|addr|
        (addr as *mut u8).write_volatile(0); //取地址并写入0, 以字节写入
    );
}

```

```

});
let app_src = core::slice::from_raw_parts(
    self.app_start[app_id] as *const u8, //起始地址
    self.app_start[app_id + 1] - self.app_start[app_id], //长度, 以字节记
); //获取应用二进制数据
let app_dst = core::slice::from_raw_parts_mut(APP_BASE_ADDRESS as *mut u8,
    app_src.len()); //将要写入应用二进制数据的位置
app_dst.copy_from_slice(app_src); //写入数据
}

```

---

### 4.3 risc-v 特权级切换的时机

用户程序运行在 U 模式下，操作系统运行在 S 模式下，在执行运用程序的时候，我们需要为应用程序做一些初始化工作。

- 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，即将应用程序放到合适的位置，并将 pc 指针设置到此处运行
- 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，当完成系统提供的服务后需要回到应用程序继续执行，此时就需要维持应用程序的上下文保持不变
- 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）

应用程序的上下文包括通用寄存器和栈两个主要部分。由于 CPU 在不同特权级下共享一套通用寄存器，所以在运行操作系统的 Trap 处理过程中，操作系统也会用到这些寄存器，这会改变应用程序的上下文。因此，与函数调用需要保存函数调用上下文活动记录一样，在执行操作系统的 Trap 处理过程（会修改通用寄存器）之前，我们需要在某个地方（某内存块或内核的栈）保存这些寄存器并在 Trap 处理结束后恢复这些寄存器。

特权级切换过程一部分硬件会自行完成改变，剩余的需要操作系统进行处理。

### 4.4 RISC-V 中断相关寄存器

通用寄存器: x0 x31,x10 x17 => a0 a7, x1=>ra ,x2=>sp 栈顶寄存器.

在触发中断进入 S 态进行处理时，硬件会将下面的寄存器进行设置:

- 1.sepc(exception program counter)，它会记录触发中断的那条指令的地址
- 2.scause，它会记录中断发生的原因，还会记录该中断是不是一个外部中断
- 3.stval，它会记录一些中断处理所需要的辅助信息，比如取指、访存、缺页异常，它会把发生问题的目标地址记录下来，这样我们在中断处理程序中就知道处理目标了。

4.stve,设置如何寻找 S 态中断处理程序的起始地址,保存了中断向量表基址 BASE,同时还有模式 MODE。当 MODE=0，设置为 Direct 模式时，无论中断因何发生我们都直接跳转到基址 BASE。当 MODE=1 时，设置为 Vectored 模式时，遇到中断我们会进行另一个转换然后跳转。我们只需将各中断处理程序放在正确的位置，并设置好 stvec，遇到中断的时候硬件根据中断原因就会自动跳转到对应的中断处理程序了

5.sstatus, S 态控制状态寄存器。保存全局中断使能标志，以及许多其他的状态。可设置此寄存器来中断使能与否。sstatus 的 ssp 字段会被修改为 cpu 当前的特权级

## 4.5 Trap 上下文保存恢复

在中断/异常/系统调用发生的时候，cpu 就会 trap 切换到 S 态并跳转到 stvec 指向的位置进行相关的处理。而在进入 S 态之前，操作系统需要完成对上面所提到的应用程序上下文进行保存，而这些信息需要保存在操作系统的内核栈中。

使用中断栈帧来保存 trap 上下文

---

```
#[repr("C")]
pub struct TrapFrame{
    pub reg:[usize;32],//32个通用寄存器
    pub sstatus:Sstatus,
    pub sepc:usize,
}
```

---

特权级切换的核心是对 Trap 的管理

- 应用程序通过 ecall 进入到内核状态时，操作系统保存被打断的应用程序的 Trap 上下文
- 操作系统根据 Trap 相关的 CSR 寄存器内容，完成系统调用服务的分发与处理
- 操作系统完成系统调用服务后，需要恢复被打断的应用程序的 Trap 上下文，并通 sret 让应用程序继续执行

上下文保存恢复的汇编代码

---

```
#[repr("C")]
_alltraps:
    csrrw sp, sscratch, sp #交换sp和sscratch, 此时sp是内核栈顶地址, sscratch是用户栈顶地址
    addi sp, sp, -34*8 #开辟一部分空间保存寄存器
    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) #保存sstatus的值
    sd t1, 33*8(sp) #保存sepc的值

    csrr t2, sscratch
    sd t2, 2*8(sp)
    #让寄存器a0保存sp的值, 即栈顶指针
    mv a0, sp
    #让a0保存这个上下文的起始地址, 在trap_handle中需要基于此进行trap的分发
    call trap_handler

_restore:
    # 把sp的地址重新设置为栈顶地址
```

```

mv sp, a0
ld t0, 32*8(sp)
ld t1, 33*8(sp)
ld t2, 2*8(sp)
csrw sstatus, t0
csrw sepc, t1
csrw sscratch, t2
ld x1, 1*8(sp)
ld x3, 3*8(sp)
.set n, 5
.rept 27
    LOAD_GP %n
    .set n, n+1
.endr #恢复所有寄存器的值
addi sp, sp, 34*8 #释放栈空间
csrrw sp, sscratch, sp #再次交换两个寄存器, 此时sp指向了用户栈
sret #返回用户态

```

---

## 4.6 应用程序执行的完整过程

第一步, 设置好中断/异常/系统调用进入 trap 时进行相应处理的入口。

```

trap::init();//trap初始化, 设置stvec的入口地址
+++++
pub fn init(){
    unsafe {
        extern "C"{
            fn _alltraps();
        }
        stvec::write(_alltraps as
            usize, stvec::TrapMode::Direct);//设置了steval的值为_allstrap进行上下文保存
    }
    println!("++++ setup trap! +++");
}

```

---

第二步, 初始化应用管理器, 获取应用的相关信息, 即开始位置, 结束位置。

第三步, 准备进入用户态开始运行程序。在 `run_next_app()` 中, 我们按顺序加载应用到相应的位置, 然后复用 `_restore` 函数, 我们将 `sepc` 的寄存器设置为程序的入口地址, 将特权级模式设置为用户模式, 在内核栈执行完 `push_context 0` 操作后, 此时 `sp` 寄存器是用户栈地址, 而 `pc` 指针由于执行了 `sret` 指令已经变成了 `sepc` 寄存器的内容即用户程序的入口, 因此此时就会转到用户程序执行。

```

batch::run_next_app();
+++++
pub fn run_next_app() ->!{
    let current_app = APP_MANAGER.inner.borrow().current_app;
    unsafe {
        APP_MANAGER.inner.borrow().load_app(current_app);//加载application到0x80400000位置开始运行
    }
}

```

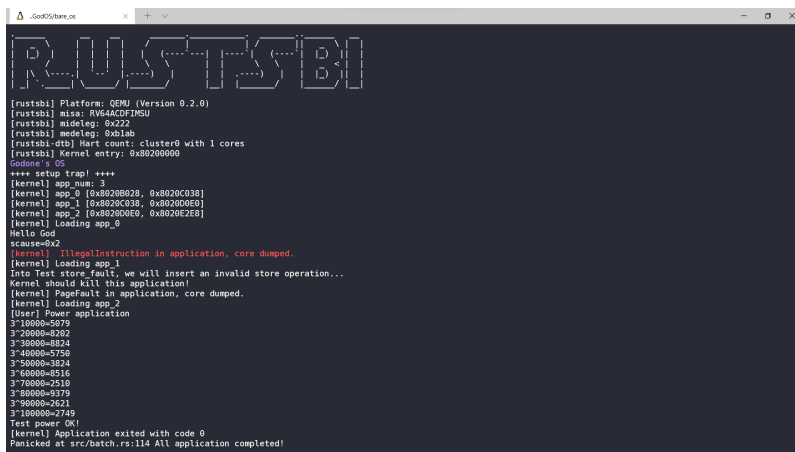
```

//设置下一个应用
APP_MANAGER.inner.borrow_mut().move_to_next_app();
extern "C"{
    fn _restore(cx_addr:usize); //定义外部接口，来自trap.asm用于恢复上下文
}
// 复用_restore函数
// 在内核栈上压入一个Trap上下文
// sepc 是应用程序入口地址 0x80400000，
// 其 sp 寄存器指向用户栈，其sstatus 的 SPP 字段被设置为 User。
// push_context 的返回值是内核栈压入 Trap 上下文之后的内核栈顶，
// 它会被作为 __restore 的参数
unsafe {
    // println!("[kernel] Begin run application!");
    _restore(KERNEL_STACK.push_context(
        TrapFrame::app_into_context(
            APP_BASE_ADDRESS,
            USER_STACK.get_sp()))as * const _ as usize
    );
    //此时sp指向的是用户栈地址，sscratch指向的是内核栈地址
}
panic!("The end of application");
}

```

在应用程序发起系统调用或触发一些中断时，就会进入 trap 处理入口，pub fn trap\_handler(tf:&mut TrapFrame)->&mut TrapFrame 函数会工具中断原因进行分发处理，并在必要时结束某些应用运行下一个应用。

## 4.7 运行结果



```

RUSTSB1
[rustsb1] Platform: QEMU (Version 0.2.0)
[rustsb1] misa: RV64ACDFIMSU
[rustsb1] mdeleg: 0x222
[rustsb1] mdeleg: 0x51ab
[rustsb1-dtb] Hart count: cluster0 with 1 cores
[rustsb1] Kernel entry: 0x80200000
Godone's OS
**** setup trap! ****
[kernel] app num: 3
[kernel] app_0 [0x80200020, 0x8020C030]
[kernel] app_1 [0x8020C030, 0x8020D060]
[kernel] app_2 [0x8020D060, 0x8020E2E0]
[kernel] Loading app_0
Hello God
scause=0x2
[kernel] Illegal instruction in application, core dumped.
[kernel] Loading app_1
Into test store fault, we will insert an invalid store operation...
Kernel should kill this application!
[kernel] PageFault in application, core dumped.
[kernel] Loading app_2
[user] Power application
3*10000-5679
3*20000-4202
3*30000-8924
3*40000-5750
3*50000-3824
3*60000-8516
3*70000-2510
3*80000-8379
3*90000-2621
3*100000-2749
Test power OK!
[kernel] Application exited with code 0
Panic! at src/batch.rs:114 All application completed!

```

图 2: 运行结果