实现特权级的切换

本节导读

由于有特权级机制的存在，应用程序在用户态特权级运行时，是无法直接通过函数调用访问处于内核态特权级的批处理操作系统内核中的函数的。所以会通过某种机制进行特权级之间的切换，使得用户态应用程序可以得到内核态操作系统函数的服务。本节将讲解在RISC-V 64处理器提供的U/S特权级下，批处理操作系统和应用程序如何相互配合，完成特权级切换的。

RISC-V特权级切换

特权级切换的起因

我们知道，批处理操作系统被设计为运行在 S 模式，这是由RustSBI提供的 SEE(Supervisor Execution Environment) 所保证的；而应用程序被设计为运行在 U 模式，这个则是由我们的批处理操作系统提供的 AEE(Application Execution Environment) 所保证的。批处理操作系统为了建立好应用程序的执行环境，需要在执行应用程序之前进行一些初始化工作，并监控应用程序的执行，具体体现在：

- 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）。

这些处理都涉及到特权级切换，因此都需要硬件和操作系统协同提供的特权级切换机制。

特权级切换相关的控制状态寄存器

当从一般意义上讨论 RISC-V 架构的 Trap 机制时，通常需要注意两点：

- 在触发 Trap 之前 CPU 运行在哪个特权级；
- 以及 CPU 需要切换到哪个特权级来处理该 Trap 并在处理完成之后返回原特权级。

但本章中我们仅考虑当 CPU 在 U 特权级运行用户程序的时候触发 Trap，并切换到 S 特权级的批处理操作系统的对应服务代码来进行处理。

在 RISC-V 架构中，关于 Trap 有一条重要的规则：在 Trap 前的特权级不会高于Trap后的特权级。因此如果触发 Trap 之后切换到 S 特权级（下称 Trap 到 S），说明 Trap 发生之前 CPU 只能运行在 S/U 特权级。但无论如何，只要是 Trap 到 S 特权级，操作系统就会使用 S 特权级中与 Trap 相关的 **控制状态寄存器** (CSR, Control and Status Register) 来辅助 Trap 处理。我们在编写运行在 S 特权级的批处理操作系统中的 Trap 处理相关代码的时候，就需要使用如下所示的S模式的CSR寄存器。

进入 S 特权级 Trap 的相关 CSR

CSR 名	该 CSR 与 Trap 相关的功能
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

CSR 名	该 CSR 与 Trap 相关的功能
stvec	控制 Trap 处理代码的入口地址

注解

S模式下最重要的 sstatus 寄存器

注意 `sstatus` 是 S 特权级最重要的 CSR，可以从很多方面控制 S 特权级的CPU行为和执行状态。

特权级切换

大多数的 Trap (陷入) 发生的场景都是在执行某条指令（如 `ecall` ）之后，CPU 发现触发了一个 Trap并需要进行特殊处理，并涉及到 执行环境切换 。具体而言，用户态执行环境中的应用程序通过 `ecall` 指令 向内核态执行环境在的操作系统请求某项服务功能，那么处理器和操作系统会完成到内核态执行环境的切换，并在操作系统完成服务后，再次切换到用户态执行环境，然后应用程序会紧接着``ecall`` 指令的后一条指令位置处继续执行，参考 图示 。

应用程序被切换回来之后需要从暂停的位置恢复并继续执行，这需要在切换前后维持应用程序的上下文保持不变。应用程序的上下文可以分为通用寄存器和栈两部分。 由于每个 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，这将应用程序的上下文。因此，就和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 处理过程的最开始， 即修改这些寄存器之前，我们需要在某个地方（就是某内存块或内核的栈）保存这些寄存器并在后续恢复这些寄存器。

除了通用寄存器之外还有一些可能在处理 Trap 过程中会被修改的 CSR，比如 CPU 所在的特权级。我们要保证它们的变化在我们的预期之内，比如对于特权级而言应该是 Trap 之前在 U 特权级，处理 Trap 的时候在 S 特权级，返回之后又需要回到 U 特权级。而对于栈问题则相对简单，只要两个执行流用来记录执行历史的栈所对应的内存区域不相交，就不会产生令我们 头痛的覆盖问题，也就无需进行保存/恢复。

执行流切换的相关机制一部分由硬件帮我们完成，另一部分则需要由操作系统来实现。

特权级切换的硬件控制机制

当 CPU 执行完一条指令并准备从用户特权级 Trap 到 S 特权级的时候，硬件会自动帮我们做这些事情：

- `sstatus` 的 `SPP` 字段会被修改为 CPU 当前的特权级（U/S）。
- `sepc` 会被修改为 Trap 回来之后默认会执行的下一条指令的地址。当 Trap 是一个异常的时候，它实际会被修改成 Trap 之前执行的最后一条 指令的地址。
- `scause/stval` 分别会被修改成这次 Trap 的原因以及相关的附加信息。
- 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 ，然后开始向下执行。

注解

stvec 相关细节

在 RV64 中， `stvec` 是一个 64 位的 CSR，在中断使能的情况下，保存了中断处理的入口地址。它有两个字段：

- MODE 位于 [1:0]，长度为 2 bits；
- BASE 位于 [63:2]，长度为 62 bits。

当 MODE 字段为 0 的时候， `stvec` 被设置为 Direct 模式，此时进入 S 模式的 Trap 无论原因如何，处理 Trap 的入口地址都是 `BASE<<2` ， CPU 会跳转到这个地方进行异常处理。本书中我们只会将 `stvec` 设置为 Direct 模式。而 `stvec` 还可以被设置为 Vectored 模式， 有兴趣的读者可以自行参考 RISC-V 指令集特权级规范。

而当 CPU 完成 Trap 处理准备返回的时候，需要通过一条 S 特权级的特权指令 `sret` 来完成，这一条指令具体完成以下功能：

- CPU 会将当前的特权级按照 `sstatus` 的 `SPP` 字段设置为 U 或者 S；
- CPU 会跳转到 `sepc` 寄存器指向的那条指令，然后开始向下执行。

从上面可以看出硬件主要负责特权级切换、跳转到异常处理入口地址（要在使能异常/中断前设置好）以及在 CSR 中保存一些只有硬件才方便探测到的硬件内的 Trap 相关信息。这基本上都是硬件不得不完成的事情，剩下的工作都交给软件，让软件能有更大的灵活性。

用户栈与内核栈

在 Trap 触发的一瞬间， CPU 就会切换到 S 特权级并跳转到 `stvec` 所指示的位置。但是在正式进入 S 特权级的 Trap 处理之前，上面 提到过我们必须保存原执行流的寄存器状态，这一般通过栈来完成。但我们需要用专门为操作系统准备的内核栈，而不是应用程序运行时用到的用户栈。

使用两个不同的栈是为了安全性：如果两个执行流使用同一个栈，在返回之后应用程序就有能力看到 Trap 执行流的 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加入一段汇编代码中，实现从用户栈切换到内核栈， 并在内核栈上保存应用程序执行流的寄存器状态。

我们声明两个类型 `KernelStack` 和 `UserStack` 分别表示用户栈和内核栈，它们都只是字节数组的简单包装：

```rust
// os/src/batch.rs

const USER_STACK_SIZE: usize = 4096 * 2;
const KERNEL_STACK_SIZE: usize = 4096 * 2;

#[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] };
```

常数 `USER_STACK_SIZE` 和 `KERNEL_STACK_SIZE` 指出内核栈和用户栈的大小分别为 $8\mathrm{KiB}$ 。两个类型是以全局变量 的形式实例化在批处理操作系统的 `.bss` 段中的。

我们为两个类型实现了 `get_sp` 方法来获取栈顶地址。由于在 RISC-V 中栈是向下增长的，我们只需返回包裹的数组的终止地址，以用户栈 类型 `UserStack` 为例：

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

于是换栈是非常简单的，只需将 `sp` 寄存器的值修改为 `get_sp` 的返回值即可。

接下来是Trap上下文（即数据结构 `TrapContext` ），类似前面提到的函数调用上下文，即在 Trap 发生时需要保存的物理资源内容，并将其一起放在一个名为 `TrapContext` 的类型中，定义如下：

```rust
// os/src/trap/context.rs

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

可以看到里面包含所有的通用寄存器 `x0~x31` ，还有 `sstatus` 和 `sepc` 。那么为什么需要保存它们呢?

- 对于通用寄存器而言，两条执行流运行在不同的特权级，所属的软件也可能由不同的编程语言编写，虽然在 Trap 控制流中只是会执行 Trap 处理 相关的代码，但依然可能直接或间接调用很多模块，因此很难甚至不可能找出哪些寄存器无需保存。既然如此我们就只能全部保存了。但这里也有一些例外，如 `x0` 被硬编码为 0 ，它自然不会有变化；还有 `tp(x4)` 除非我们手动出于一些特殊用途使用它，否则一般也不会被用到。它们无需保存， 但我们仍然在 `TrapContext` 中为它们预留空间，主要是为了后续的实现方便。
- 对于 CSR 而言，我们知道进入 Trap 的时候，硬件会立即覆盖掉 `scause/stval/sstatus/sepc` 的全部或是其中一部分。 `scause/stval` 的情况是：它总是在 Trap 处理的第一时间就被使用或者是在其他地方保存下来了，因此它没有被修改并造成不良影响的风险。 而对于 `sstatus/sepc` 而言，它们会在 Trap 处理的全程有意义（在 Trap 执行流最后 `sret` 的时候还用到了它们），而且确实会出现 Trap 嵌套的情况使得它们的值被覆盖掉。所以我们需要将它们也一起保存下来，并在 `sret` 之前恢复原样。

Trap 管理

特权级切换的核心是对Trap的管理。这主要涉及到如下一下内容：

- 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。
- 操作系统根据与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。

接下来我们具体介绍上述内容。

Trap 上下文的保存与恢复

首先是具体实现 Trap 上下文保存和恢复的汇编代码。

在批处理操作系统初始化的时候，我们需要修改 `stvec` 寄存器来指向正确的 Trap 处理入口点。

```rust
// os/src/trap/mod.rs

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

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

这里我们引入了一个外部符号 `__alltraps` ，并将 `stvec` 设置为 Direct 模式指向它的地址。我们在 `os/src/trap/trap.S` 中实现 Trap 上下文保存/恢复的汇编代码，分别用外部符号 `__alltraps` 和 `__restore` 标记，并将这段汇编代码中插入进来。

Trap 处理的总体流程如下：首先通过 `__alltraps` 将 Trap 上下文保存在内核栈上，然后跳转到使用 Rust 编写的 `trap_handler` 函数 完成 Trap 分发及处理。当 `trap_handler` 返回之后，使用 `__restore` 从保存在内核栈上的 Trap 上下文恢复寄存器。最后通过一条 `sret` 指令回到应用程序执行。

首先是保存 Trap 上下文的 `__alltraps` 的实现：

```
1  # os/src/trap/trap.S
2
3  .macro SAVE_GP n
4      sd x\n, \n*8(sp)
5  .endm
6
7  .align 2
8  __alltraps:
9      csrrw sp, sscratch, sp
10     # now sp->kernel stack, sscratch->user stack
11     # allocate a TrapContext on kernel stack
12     addi sp, sp, -34*8
13     # save general-purpose registers
14     sd x1, 1*8(sp)
15     # skip sp(x2), we will save it later
16     sd x3, 3*8(sp)
17     # skip tp(x4), application does not use it
18     # save x5~x31
19     .set n, 5
20     .rept 27
21         SAVE_GP %n
22         .set n, n+1
23     .endr
24     # we can use t0/t1/t2 freely, because they were saved on kernel stack
25     csrr t0, sstatus
26     csrr t1, sepc
27     sd t0, 32*8(sp)
28     sd t1, 33*8(sp)
29     # read user stack from sscratch and save it on the kernel stack
30     csrr t2, sscratch
31     sd t2, 2*8(sp)
32     # set input argument of trap_handler(cx: &mut TrapContext)
33     mv a0, sp
34     call trap_handler
```

- 第 7 行我们使用 `.align` 将 `__alltraps` 的地址 4 字节对齐，这是 RISC-V 特权级规范的要求；
- 第 8 行的 `csrrw` 原型是 csrrw rd, csr, rs 可以将 CSR 当前的值读到通用寄存器 rd 中，然后将 通用寄存器 rs 的值写入该 CSR 。因此这里起到的是交换 sscratch 和 sp 的效果。在这一行之前 sp 指向用户栈， sscratch 指向内核栈（原因稍后说明），现在 sp 指向内核栈， sscratch 指向用户栈。
- 第 12 行，我们准备在内核栈上保存 Trap 上下文，于是预先分配 34×8 字节的栈帧，这里改动的是 sp ，说明确实是在内核栈上。
- 第 13~24 行，保存 Trap 上下文的通用寄存器 x0~x31，跳过 x0 和 tp(x4)，原因之前已经说明。我们在这里也不保存 sp(x2)，因为我们要基于 它来找到每个寄存器应该被保存到的正确的位置。实际上，在栈帧分配之后，我们可用于保存 Trap 上下文的地址区间为 $[\text{sp}, \text{sp} + 8 \times 34)$ ，

  按照 `TrapContext` 结构体的内存布局，它从低地址到高地址分别按顺序放置 x0~x31，最后是 sstatus 和 sepc 。因此通用寄存器 xn 应该被保存在地址区间 $[\text{sp} + 8n, \text{sp} + 8(n+1))$ 。 在这里我们正是这样基于 sp 来保存这些通用寄存器的。

  为了简化代码，x5~x31 这 27 个通用寄存器我们通过类似循环的 `.rept` 每次使用 `SAVE_GP` 宏来保存，其实质是相同的。注意我们需要在 `Trap.S` 开头加上 `.altmacro` 才能正常使用 `.rept` 命令。

- 第 25~28 行，我们将 CSR sstatus 和 sepc 的值分别读到寄存器 t0 和 t1 中然后保存到内核栈对应的位置上。指令 csrr rd, csr 的功能就是将 CSR 的值读到寄存器 rd 中。这里我们不用担心 t0 和 t1 被覆盖， 因为它们刚刚已经被保存了。
- 第 30~31 行专门处理 sp 的问题。首先将 sscratch 的值读到寄存器 t2 并保存到内核栈上，注意它里面是进入 Trap 之前的 sp 的值，指向 用户栈。而现在的 sp 则指向内核栈。
- 第 33 行令 $\text{a}_0 \leftarrow \text{sp}$，让寄存器 a0 指向内核栈的栈指针也就是我们刚刚保存的 Trap 上下文的地址， 这是由于我们接下来要调用 `trap_handler` 进行 Trap 处理，它的第一个参数 `cx` 由调用规范要从 a0 中获取。而 Trap 处理函数 `trap_handler` 需要 Trap 上下文的原因在于：它需要知道其中某些寄存器的值，比如在系统调用的时候应用程序传过来的 syscall ID 和 对应参数。我们不能直接使用这些寄存器现在的值，因为它们可能已经被修改了，因此要去内核栈

上找已经被保存下来的值。

注解

CSR 相关原子指令

RISC-V 中读写 CSR 的指令通常都能只需一条指令就能完成多项功能。这样的指令被称为 **原子指令** (Atomic Instruction)。这里 的原子的含义是“不可分割的最小个体”，也就是说指令的多项功能要么都不完成，要么全部完成，而不会处于某种中间状态。

当 `trap_handler` 返回之后会从调用 `trap_handler` 的下一条指令开始执行，也就是从栈上的 Trap 上下文恢复的 `__restore` ：

```
1  .macro LOAD_GP n
2      ld x\n, \n*8(sp)
3  .endm
4
5  __restore:
6      # case1: start running app by __restore
7      # case2: back to U after handling trap
8      mv sp, a0
9      # now sp->kernel stack(after allocated), sscratch->user stack
10     # restore sstatus/sepc
11     ld t0, 32*8(sp)
12     ld t1, 33*8(sp)
13     ld t2, 2*8(sp)
14     csrw sstatus, t0
15     csrw sepc, t1
16     csrw sscratch, t2
17     # restore general-purpuse registers except sp/tp
18     ld x1, 1*8(sp)
19     ld x3, 3*8(sp)
20     .set n, 5
21     .rept 27
22         LOAD_GP %n
23         .set n, n+1
24     .endr
25     # release TrapContext on kernel stack
26     addi sp, sp, 34*8
27     # now sp->kernel stack, sscratch->user stack
28     csrrw sp, sscratch, sp
29     sret
```

- 第 8 行比较奇怪我们暂且不管，假设它从未发生，那么 sp 仍然指向内核栈的栈顶。
- 第 11~24 行负责从内核栈顶的 Trap 上下文恢复通用寄存器和 CSR 。注意我们要先恢复 CSR 再恢复通用寄存器，这样我们使用的三个临时寄存器 才能被正确恢复。
- 在第 26 行之前，sp 指向保存了 Trap 上下文之后的内核栈栈顶， sscratch 指向用户栈栈顶。我们在第 26 行在内核栈上回收 Trap 上下文所 占用的内存，回归进入 Trap 之前的内核栈栈顶。第 27 行，再次交换 sscratch 和 sp，现在 sp 重新指向用户栈栈顶，sscratch 也依然保存 进入 Trap 之前的状态并指向内核栈栈顶。
- 在应用程序执行流状态被还原之后，第 28 行我们使用 `sret` 指令回到 U 特权级继续运行应用程序执行流。

注解

sscratch CSR 的用途

在特权级切换的时候，我们需要将 Trap 上下文保存在内核栈上，因此需要一个寄存器暂存内核栈地址，并以它作为基地址来依次保存 Trap 上下文 的内容。但是所有的通用寄存器都不能够用来暂存，因为它们都需要被保存，如果覆盖掉它们会影响应用执行流的执行。

事实上我们缺少了一个重要的中转寄存器，而 `sscratch` CSR 正是为此而生。从上面的汇编代码中可以看出，在保存 Trap 上下文的时候，它 起到了两个作用：首先是保存了内核栈的地址，其次它作为一个中转站让 sp 目前指向的用户栈的地址可以暂时保存下来。于是，我们仅需一条 `csrrw` 指令就完成了从用户栈到内核栈的切换，这是一种极其精巧的实现。

Trap 分发与处理

Trap 在使用 Rust 实现的 `trap_handler` 函数中完成分发和处理：

```rust
// os/src/trap/mod.rs

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

- 第 4 行声明返回值为 `&mut TrapContext` 并在第 25 行实际将传入的 `cx` 原样返回，因此在 `__restore` 的时候 a0 在调用 `trap_handler` 前后并没有发生变化，仍然指向分配 Trap 上下文之后的内核栈栈顶，和此时 sp 的值相同，我们 $\mathrm{sp} \leftarrow \mathrm{a}_0$ 并不会有问题；
- 第 7 行根据 scause 寄存器所保存的 Trap 的原因进行分发处理。这里我们无需手动操作这些 CSR ，而是使用 Rust 的 riscv 库来更加方便的 做这些事情。要引入 riscv 库，我们需要：

  ```
  # os/Cargo.toml

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

- 第 8~11 行，发现 Trap 的原因是来自 U 特权级的 Environment Call，也就是系统调用。这里我们首先修改保存在内核栈上的 Trap 上下文里面 sepc，让其增加 4。这是因为我们知道这是一个由 `ecall` 指令触发的系统调用，在进入 Trap 的时候，硬件会将 sepc 设置为这条 `ecall` 指令所在的地址（因为它是进入 Trap 之前最后一条执行的指令）。而在 Trap 返回之后，我们希望应用程序执行流从 `ecall` 的下一条指令 开始执行。因此我们只需修改 Trap 上下文里面的 sepc，让它增加 `ecall` 指令的码长，也即 4 字节。这样在 `__restore` 的时候 sepc 在恢复之后就会指向 `ecall` 的下一条指令，并在 `sret` 之后从那里开始执行。这属于我们之前提到过的——用户程序能够预知到的执行流 状态所发生的变化。

  用来保存系统调用返回值的 a0 寄存器也会同样发生变化。我们从 Trap 上下文取出作为 syscall ID 的 a7 和系统调用的三个参数 a0~a2 传给 `syscall` 函数并获取返回值。 `syscall` 函数是在 `syscall` 子模块中实现的。

- 第 12~20 行，分别处理应用程序出现访存错误和非法指令错误的情形。此时需要打印错误信息并调用 `run_next_app` 直接切换并运行下一个 应用程序。
- 第 21 行开始，当遇到目前还不支持的 Trap 类型的时候，我们的批处理操作系统整个 panic 报错退出。

对于系统调用而言， `syscall` 函数并不会实际处理系统调用而只是会根据 syscall ID 分发到具体的处理函数：

```
// os/src/syscall/mod.rs

pub fn syscall(syscall_id: usize, args: [usize; 3]) -> isize {
    match syscall_id {
        SYSCALL_WRITE => sys_write(args[0], args[1] as *const u8, args[2]),
        SYSCALL_EXIT => sys_exit(args[0] as i32),
        _ => panic!("Unsupported syscall_id: {}", syscall_id),
    }
}
```

这里我们会将传进来的参数 `args` 转化成能够被具体的系统调用处理函数接受的类型。它们的实现都非常简单：

```
// os/src/syscall/fs.rs

const FD_STDOUT: usize = 1;

pub fn sys_write(fd: usize, buf: *const u8, len: usize) -> isize {
    match fd {
        FD_STDOUT => {
            let slice = unsafe { core::slice::from_raw_parts(buf, len) };
            let str = core::str::from_utf8(slice).unwrap();
            print!("{}", str);
            len as isize
        },
        _ => {
            panic!("Unsupported fd in sys_write!");
        }
    }
}

// os/src/syscall/process.rs

pub fn sys_exit(xstate: i32) -> ! {
    println!("[kernel] Application exited with code {}", xstate);
    run_next_app()
}
```

- `sys_write` 我们将传入的位于应用程序内的缓冲区的开始地址和长度转化为一个字符串 `&str` ，然后使用批处理操作系统已经实现的 `print!` 宏打印出来。注意这里我们并没有检查传入参数的安全性，即使会在出错严重的时候 panic，还是会存在安全隐患。这里我们出于实现方便暂且不做修补。
- `sys_exit` 打印退出的应用程序的返回值并同样调用 `run_next_app` 切换到下一个应用程序。

执行应用程序

当批处理操作系统初始化完成，或者是某个应用程序运行结束或出错的时候，我们要调用 `run_next_app` 函数切换到下一个应用程序。此时 CPU 运行在 S 特权级，而它希望能够切换到 U 特权级。在 RISC-V 架构中，唯一一种能够使得 CPU 特权级下降的方法就是通过 Trap 返回系列指令，比如 `sret` 。事实上，在运行应用程序之前要完成如下这些工作：

- 跳转到应用程序入口点 `0x80400000` 。
- 将使用的栈切换到用户栈。
- 在 `__alltraps` 时我们要求 `sscratch` 指向内核栈，这个也需要在此时完成。
- 从 S 特权级切换到 U 特权级。

它们可以通过复用 `__restore` 的代码更容易的实现。我们只需要在内核栈上压入一个相应构造的 Trap 上下文，再通过 `__restore` ，就能 让这些寄存器到达我们希望的状态。

```rust
// os/src/trap/context.rs

impl TrapContext {
    pub fn set_sp(&mut self, sp: usize) { self.x[2] = sp; }
    pub fn app_init_context(entry: usize, sp: usize) -> Self {
        let mut sstatus = sstatus::read();
        sstatus.set_spp(SPP::User);
        let mut cx = Self {
            x: [0; 32],
            sstatus,
            sepc: entry,
        };
        cx.set_sp(sp);
        cx
    }
}
```

为 `TrapContext` 实现 `app_init_context` 方法，修改其中的 sepc 寄存器为应用程序入口点 `entry`， sp 寄存器为我们设定的一个栈指针，并将 sstatus 寄存器的 `SPP` 字段设置为 User 。

在 `run_next_app` 函数中我们能够看到：

```rust
// os/src/batch.rs

pub fn run_next_app() -> ! {
    let current_app = APP_MANAGER.inner.borrow().get_current_app();
    unsafe {
        APP_MANAGER.inner.borrow().load_app(current_app);
    }
    APP_MANAGER.inner.borrow_mut().move_to_next_app();
    extern "C" { fn __restore(cx_addr: usize); }
    unsafe {
        __restore(KERNEL_STACK.push_context(
            TrapContext::app_init_context(APP_BASE_ADDRESS, USER_STACK.get_sp())
        ) as *const _ as usize);
    }
    panic!("Unreachable in batch::run_current_app!");
}
```

在高亮行所做的事情是在内核栈上压入一个 Trap 上下文，其 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 上下文之后的栈顶，它会被作为 `__restore` 的参数（回看 __restore 代码，这时我们可以理解为何 `__restore` 的开头会做 $\mathrm{sp} \leftarrow \mathrm{a}_0$ ）使得在 `__restore` 中 sp 仍然可以指向内核栈的栈顶。这之后，就和一次普通的 `__restore` 一样了。

注解

有兴趣的读者可以思考： sscratch 是何时被设置为内核栈顶的?