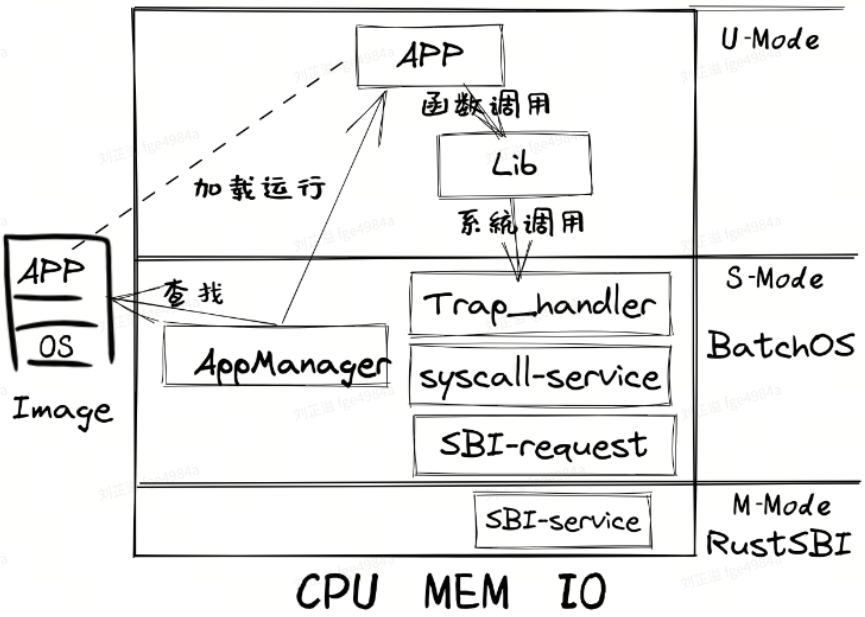
BatchOS

BatchOS



- 实现批处理程序功能的OS
- APP与OS隔离
- 自动加载并运行多个程序

特权级机制

- `ecall` 具有用户态到内核态的执行环境切换能力的函数调用指令；
- `sret` ： 具有内核态到用户态的执行环境切换能力的函数返回指令。
- 首先，操作系统需要提供相应的功能代码，能在执行 `sret` 前准备和恢复用户态执行应用程序的上下文。其次，在应用程序调用 `ecall` 指令后，能够检查应用程序的系统调用参数，确保参数不会破坏操作系统。

RISC-V异常

Interrupt	Exception Code	Description
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	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store/AMO page fault

RISC-V S模式特权指令

- sert：从S模式返回U模式
- wfi：处理器在空闲时进入低功耗状态等待终端
- sfence.vma：刷新TLB缓存
- 访问S模式CSR指令：改变系统状态

控制状态寄存器

- sstatus：SPP字段给出Trap发生前CPU的特权级
- sepc：记录异常发生前执行的最后一条指令的地址

- scause：描述Trap的原因
- stval：给出Trap的附加信息
- stvec：控制Trap处理代码的入口地址

硬件切换的硬件控制机制

ecall

- `sstatus` 中的 `SPP` 字段切换到CPU当前特权级
- `sepc` 修改为Trap处理完成后默认执行的下一条指令的地址
- `scause\stval` 修改成Trap原因和Trap额外信息
- CPU跳转到 `stvec` 设置的Trap处理入口函数，设置特权级为S

sret

- CPU按照 `sstatus` 设置特权级
- CPU跳转到 `sepc` 指向的地址，然后继续执行

用户库

- 使用 Rust 的宏将其函数符号 `main` 标志为弱链接。这样在最后链接的时候，虽然在 `lib.rs` 和 `bin` 目录下的某个应用程序都有 `main` 符号，但由于 `lib.rs` 中的 `main` 符号是弱链接，链接器会使用 `bin` 目录下的应用主逻辑作为 `main`
- 这里我们主要是进行某种程度上的保护，如果在 `bin` 目录下找不到任何 `main` ，那么编译也能够通过，但会在运行时报错。

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

- `#![feature(linkage)]` 支持链接操作

系统调用

- `&[u8]` 切片类型来描述缓冲区，这是一个 **胖指针** (Fat Pointer)，里面既包含缓冲区的起始地址，还 包含缓冲区的长度。

```
// user/src/syscall.rs
use core::arch::asm;
```

```rust
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
        );
    }
    ret
}
const SYSCALL_WRITE: usize = 64;
const SYSCALL_EXIT: usize = 93;

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

pub fn sys_exit(xstate: i32) -> isize {
    syscall(SYSCALL_EXIT, [xstate as usize, 0, 0])
}
```

加载不同的APP

AppManager

- 在 `RefCell` 的基础上再封装一个 `UPSafeCell` ，它名字的含义是：允许我们在 *单核* 上安全使用可变全局变量。
- 当我们要访问数据时（无论读还是写），需要首先调用 `exclusive_access` 获得数据的可变借用标记，通过它可以完成数据的读写，在操作完成之后我们需要销毁这个标记，此后才能开始对该数据的下一次访问

```rust
// os/src/sync/up.rs

pub struct UPSafeCell<T> {
    /// inner data
    inner: RefCell<T>,
}

// unsafe向编译器保证只在单核上进行操作
unsafe impl<T> Sync for UPSafeCell<T> {}

```

```rust
impl<T> UPSafeCell<T> {
    /// User is responsible to guarantee that inner struct is only used in
    /// uniprocessor.
    pub unsafe fn new(value: T) -> Self {
        Self { inner: RefCell::new(value) }
    }
    /// Panic if the data has been borrowed.
    pub fn exclusive_access(&self) -> RefMut<'_, T> {
        self.inner.borrow_mut()
    }
}
```

New

- 使用 `core::slice::from_raw_parts` 将指针解释为 `&[usize]` 切片；
- 使用 `copy_from_slice` 将切片上的元素复制到 `app_start` 上

```rust
static ref APP_MANAGER: UPSafeCell<AppManager> = unsafe {
    UPSafeCell::new({
        extern "C" {
            fn _num_app();
        }
        let num_app_ptr = _num_app as usize as *const usize;
        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] =
            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,
        }
    })
};
```

load_app

- CPU用存在指令缓存，使用load_app加载新的程序，需要让OS知道取指内存的变化
- OS 在这里必须使用取指屏障指令 `fence.i` ，它的功能是保证 **在它之后的取指过程必须能够看到在它之前的所有对于取指内存区域的修改**

```rust
unsafe fn load_app(&self, app_id: usize) {
```

```
    if app_id >= self.num_app {
        println!("All applications completed!");
        shutdown(false);
    }
    println!("[kernel] Loading app_{}", app_id);
    // clear app area
    core::slice::from_raw_parts_mut(APP_BASE_ADDRESS as *mut u8,
APP_SIZE_LIMIT).fill(0);
    // 将指针看作切片，使用拷贝实现App切换
    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],
    );
    let app_dst = core::slice::from_raw_parts_mut(APP_BASE_ADDRESS as *mut
u8, app_src.len());
    app_dst.copy_from_slice(app_src);
    // Memory fence about fetching the instruction memory
    // It is guaranteed that a subsequent instruction fetch must
    // observes all previous writes to the instruction memory.
    // Therefore, fence.i must be executed after we have loaded
    // the code of the next app into the instruction memory.
    // See also: riscv non-priv spec chapter 3, 'Zifencei' extension.
    asm!("fence.i");
}
```

run_next_app

- 先将一个上下文压入内核栈中；在 `__restore` 中更新 `sscrath` 指针指向内核栈栈顶
- 如果发生系统调用，从 `__alltraps` 开始执行

```
/// run next app
pub fn run_next_app() -> ! {
    let mut app_manager = APP_MANAGER.exclusive_access();
    let current_app = app_manager.get_current_app();
    unsafe {
        app_manager.load_app(current_app);
    }
    app_manager.move_to_next_app();
    drop(app_manager);
    // before this we have to drop local variables related to resources
manually
    // and release the resources
    extern "C" {
        fn __restore(cx_addr: usize);
    }
```

```
15      unsafe {
16          __restore(KERNEL_STACK.push_context(TrapContext::app_init_context(
17              APP_BASE_ADDRESS,
18              USER_STACK.get_sp(),
19          )) as *const _ as usize);
20      }
21      panic!("Unreachable in batch::run_current_app!");
22  }
```

Trap管理

TrapContext

- Trap 发生时需要保存的物理资源内容，包括32个通用寄存器、sstatus以及sepc
- 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 控制流最后 `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。

```
1   // os/src/trap/context.rs
2
3   #[repr(C)]
4   pub struct TrapContext {
5       pub x: [usize; 32],
6       pub sstatus: Sstatus,
7       pub sepc: usize,
8   }
```

TrapContext的保存与恢复

- 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。

```
1   // os/src/trap/mod.rs
2
3   global_asm!(include_str!("trap.S"));
```

```rust

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

__alltraps

- `sscratch` 在 `__restore` 中设置为内核栈栈顶， `run_next_app` 从 `__restore` 先执行

```
# os/src/trap/trap.S

.macro SAVE_GP n
    sd x\n, \n*8(sp)
.endm

.align 2
__alltraps:
    csrrw sp, sscratch, sp # exchange sp and sscratch(point to kernel stack)
    # now sp->kernel stack, sscratch->user stack
    # allocate a TrapContext on kernel stack
    addi sp, sp, -34*8
    # save general-purpose registers
    sd x1, 1*8(sp)
    # skip sp(x2), we will save it later
    sd x3, 3*8(sp)
    # skip tp(x4), application does not use it
    # save x5~x31
    .set n, 5
    .rept 27
        SAVE_GP %n
        .set n, n+1
    .endr
    # we can use t0/t1/t2 freely, because they were saved on kernel stack
    csrr t0, sstatus
    csrr t1, sepc
    sd t0, 32*8(sp)
    sd t1, 33*8(sp)
    # read user stack from sscratch and save it on the kernel stack
    csrr t2, sscratch
    sd t2, 2*8(sp)
    # set input argument of trap_handler(cx: &mut TrapContext)
    mv a0, sp # a0 point to trap context and as arguement of trap_handler
    call trap_handler
```

__restore

- 先恢复CSR寄存器再恢复通用寄存器
- `csrrw sp, sscratch, sp` 此时sscratch设置为内核栈栈顶(**由应用程序在执行前压入内核栈**)

```
# os/src/trap/trap.S

.macro LOAD_GP n
    ld x\n, \n*8(sp)
.endm

__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 # 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。
    sret
```

trap_handler

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

```

```rust
#[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
}
```

Practice

扩展内核，能够统计多个应用的执行过程中系统调用编号和访问此系统调用的次数

- 增加一个SyscallNum的结构体记录发生系统调用的次数

```rust
// batch.rs
/**
 * syscall num
 */
pub struct SyscallNum {
    num: [usize; SYSCALL_NUM],
}

impl SyscallNum {
    /**
```

```
11          * get syscall num
12          */
13         pub fn get_syscall_num(&self, syscall_id: usize) -> usize {
14             self.num[syscall_id]
15         }
16         /**
17          * inc syscall num
18          */
19         pub fn inc_syscall_num(&mut self, syscall_id: usize) {
20             self.num[syscall_id] += 1;
21         }
22     }
23
24     lazy_static!{
25         // ch2 add begin
26         /**
27          * syscall use syscall num
28          */
29         pub static ref NUM: UPSafeCell<SyscallNum> = unsafe {
30             UPSafeCell::new({
31                 SyscallNum {
32                     num: [0; SYSCALL_NUM],
33                 }
34             })
35         };
36     }
37
38     // run_next_app
39     if current_app == APP_MANAGER.exclusive_access().num_app - 1 || current_app
       == 0 {
40             println!(
41                 "sys_write num: {}",
42                 NUM.exclusive_access().get_syscall_num(0)
43             );
44             println!(
45                 "sys_exit num: {}",
46                 NUM.exclusive_access().get_syscall_num(1)
47             );
48         }
```

```
1   // syscall/mod.rs
2   use crate::batch::NUM;
3   /// handle syscall exception with `syscall_id` and other arguments
4   pub fn syscall(syscall_id: usize, args: [usize; 3]) -> isize {
5       match syscall_id {
```

```
        SYSCALL_WRITE => {
            let ret = sys_write(args[0], args[1] as *const u8, args[2]);
            NUM.exclusive_access().inc_syscall_num(0);
            ret
        }
        SYSCALL_EXIT => {
            NUM.exclusive_access().inc_syscall_num(1);
            sys_exit(args[0] as i32)
        }
        _ => panic!("Unsupported syscall_id: {}", syscall_id),
    }
```

扩展内核，能够统计每个应用执行后的完成时间

- 在 `AppManager` 中增加应用执行时长字段；在 `run_next_app` 中获取系统时钟，记录应用时间

```
// batch.rs
/**
 * AppManager
 */
struct AppManager {
    num_app: usize,
    current_app: usize,
    app_start: [usize; MAX_APP_NUM + 1],
    // ch2 add begin
    app_runtime: [u64; MAX_APP_NUM],
    // ch2 add end
}

// run_next_app
// ch2 add begin

let time: u64;
unsafe {
    asm!("rdtime {0}", out(reg) time);
}
if current_app > 0 {
    let runtime = time -
APP_MANAGER.exclusive_access().app_runtime[current_app - 1];
    APP_MANAGER
        .exclusive_access()
        .set_app_runtime(current_app - 1, runtime);
    println!(
        "[kernel] app_{} runs {} cycles",
        current_app - 1,
```

```
        APP_MANAGER
            .exclusive_access()
            .get_app_runtime(current_app - 1)
    );
}
```

sys_write 仅能输出位于程序本身内存空间内的数据，否则报错

- 构建ReliableAddr结构体记录应用程序起始和终止地址

```
// batch.rs
/**
 * Reliable address of program
 */
pub struct ReliableAddr {
    start: usize,
    end: usize,
}

impl ReliableAddr {
    /**
     * get reliable start addr
     */
    pub fn get_reliable_start(&self) -> usize {
        self.start
    }
    /**
     * get reliable end addr
     */
    pub fn get_reliable_end(&self) -> usize {
        self.end
    }
}

lazy_static!{
    /**
     * reliable addr init
     */
    pub static ref RE_ADDR:UPSafeCell<ReliableAddr> = unsafe {
        UPSafeCell::new({
            ReliableAddr {
                start: APP_BASE_ADDRESS,
                end: APP_BASE_ADDRESS + APP_SIZE_LIMIT,
            }
        })
```

```
    };
}

// run_next_app
// 注意数据需要使用clone()
let mut reliable_addr = RE_ADDR.exclusive_access();
reliable_addr.start = app_manager.app_start[current_app].clone();
reliable_addr.end = app_manager.app_start[current_app + 1].clone();
drop(reliable_addr);
```

```
// syscall/mod.rs
use crate::batch::RE_ADDR;

/// handle syscall exception with `syscall_id` and other arguments
pub fn syscall(syscall_id: usize, args: [usize; 3]) -> isize {
    match syscall_id {
        SYSCALL_WRITE => {
            let start_addr = RE_ADDR.exclusive_access().get_reliable_start();
            let end_addr = RE_ADDR.exclusive_access().get_reliable_end();
            if args[1] < start_addr {
                return -1;
            }
            if args[1] >= end_addr {
                return -1;
            }
            if args[1] + args[2] >= end_addr {
                return -1;
            }
            let ret = sys_write(args[0], args[1] as *const u8, args[2]);
            NUM.exclusive_access().inc_syscall_num(0);
            ret
        }
        SYSCALL_EXIT => {
            NUM.exclusive_access().inc_syscall_num(1);
            sys_exit(args[0] as i32)
        }
        _ => panic!("Unsupported syscall_id: {}", syscall_id),
    }
}
```