CPU异常

x86体系下的异常

常见的CPU异常类型

- 缺页错误（Page Fault）：缺页错误发生在非法的内存访问操作中。例如：当前指令试图访问没有映射的内存页或试图写入只读的内存页。
- 非法操作码（Invalid Opcode）：非法操作码发生在当前指令不正确的情况下。例如：试图在不支持 SSE 指令集 的老旧CPU上使用该指令集。
- 通用保护错误（General Protection Fault）：这是一个触发原因相对宽泛的异常。试图在用户态程序中执行特权指令或试图写入配置寄存器的保留位等非法访问操作均会触发该异常。
- 双重异常（Double Fault）：异常发生后，CPU会调用对应的异常处理函数。在调用过程中如果发生另一个异常，CPU会触发双重异常。双重异常也会在找不到对应的异常处理函数的情况下发生。
- 三重异常（Triple Fault）：如果异常发生在CPU调用双重异常处理函数的过程中，这会导致严重的三重异常。我们不能捕获或者处理三重异常。大多数处理器会选择复位并重启操作系统。

中断描述符表（interrupt descriptor table，IDT)

在IDT中，我们可以为每种异常指定一个处理函数

Type	Name	Description
u16	函数指针 [0:15]	处理函数（handler function)指针的低16位
u16	GDT 选择子	global descriptor table 代码段的选择子
u16	选项参数	参见下文
u16	函数指针 [16:31]	处理函数（handler function)指针的中间16位
u32	函数指针 [32:63]	处理函数（handler function)指针剩下的32位
u32	保留位	

Bits	Name	Description
0-2	中断栈表索引	0: 不切换栈, 1-7:当处理函数被调用时，切换到中断栈表（Interrupt Stack Table）的第n个栈
3-7	保留位	
8	0: 中断门, 1: 陷阱门	如果这个bit被设置为0，处理函数被调用的时候，中断会被禁用。
9-11	必须为1	
12	必须为0	
13-14	特权等级描述符 (DPL)	允许调用该处理函数的最小特权等级。
15	Present	

当异常发生时，CPU大致遵循下面的流程：

1. 将一些寄存器的内容压入栈中，包括当前指令的指针和 RFLAGS 寄存器的内容（我们会在文章的后续部分用到这些值）。
2. 读取中断描述符表（IDT）中对应的条目。例如：缺页错误发生时，CPU会读取IDT的第十四个条目。
3. 检查这个条目是否存在，如果没有则升级为双重错误（double fault)。
4. 如果条目是一个中断门（第40个bit没有被设置为1），则禁用硬件中断。
5. 装载指定的GDT 选择子到CS段。

6. 跳转到指定的处理函数。

risc-v体系下的异常

用户态应用程序设计

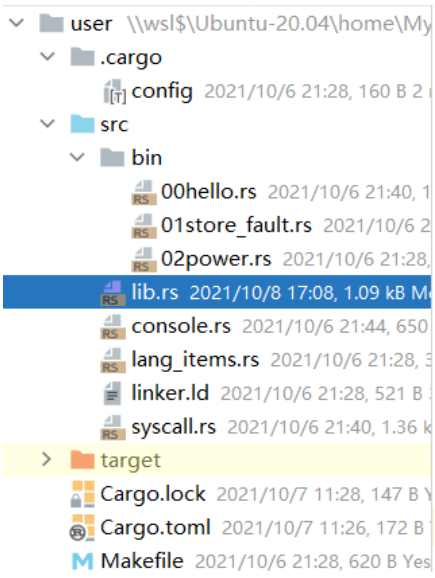
实现的要点：

- 应用程序的内存布局
- 应用程序使用系统调用

如下是文件的目录结构：

bin文件下是我们的应用程序

lib.rs文件中是我们导出的系统调用接口，供应用程序使用



在链接脚本中，我们需要将应用程序搭搭起始物理地址调整为 0x80400000 。

- *.text* 段，即代码段，存放汇编代码；
- *.rodata* 段，即只读数据段，顾名思义里面存放只读数据，通常是程序中的常量；
- *.data* 段，存放被初始化的可读写数据，通常保存程序中的全局变量；
- *.bss* 段，存放被初始化为 00 的可读写数据，与 *.data* 段的不同之处在于我们知道它要被初始化为 00 ，因此在可执行文件中只需记录这个段的大小以及所在位置即可，而不用记录里面的数据。

risc-v特权级切换的原因

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。

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

- sepc (exception program counter)，它会记录触发中断的那条指令的地址；
- 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；

- 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。
- stvec ，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 $\mathrm{MODE}=0$ ，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 $\mathrm{pc} \leftarrow \mathrm{BASE}$ 当 $\mathrm{MODE}=1$ 时，设置为 Vectored 模式时，遇到中断我们会进行跳转如下：
  $\mathrm{pc} \leftarrow \mathrm{BASE}+4 \times \mathrm{cause}$ 。而这样，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；
- 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

risc-v中断特权指令

`ecall`在s态执行这条指令时，会触发 ecall-from-s-mode-exception，从而进入 M 模式中的中断处理流程（如设置定时器等）在 U 态执行这条指令时，会触发一个 ecall-from-u-mode-exception，从而进入 S 模式中的中断处理流程（常用来进行系统调用）。

`sret`,从s态返回u态，即$PC \leftarrow sepc$

ebreak(environment break)，执行这条指令会触发一个断点中断从而进入中断处理流程

mret，用于 M 态中断返回到 S 态或 U 态，实际作用为pc ← mepc

risc-v函数调用

指令	指令功能
jal rd, imm[20:1]	rd←pc+4pc←pc+imm
jalr rd, (imm[11:0])rs	rd←pc+4

在设置PC寄存器完成跳转功能以前，会将当前跳转指令的下一条指令到村在rd寄存器中，在 RISC-V 架构中， 通常使用 `ra` 寄存器（即 `x1` 寄存器）作为其中的 `rd` 对应的具体寄存器，因此在函数返回的时候，只需跳转回 `ra` 所保存的地址即可。事实上在函数返回的时候我们常常使用一条 伪指令 (Pseudo Instruction) 跳转回调用之前的位置： `ret`

批处理操作系统设计

H5 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。
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

```
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
}
```

应用的的初始化关键代码

```
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
    }),
};
}
```

加载应用程序

```
unsafe fn load_app(&self, app_id: usize) {
    if app_id >= self.num_app {
        panic!("All application completed!");
    }
    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
    );
    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); //写入数据
}
```

H5 用户栈与内核栈

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

```
.altmacro
.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
    # 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)
    #让寄存器a0保存sp的值，即栈顶指针
    mv a0, sp
    call trap_handler
```

```
_restore:
    # case1: start running app by __restore
    # case2: back to U after handling trap
    # 把sp的地址重新设置为栈顶地址
    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
```

H5 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);
        // sstatus::set_sie();//s态全局使能位
    }
    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。

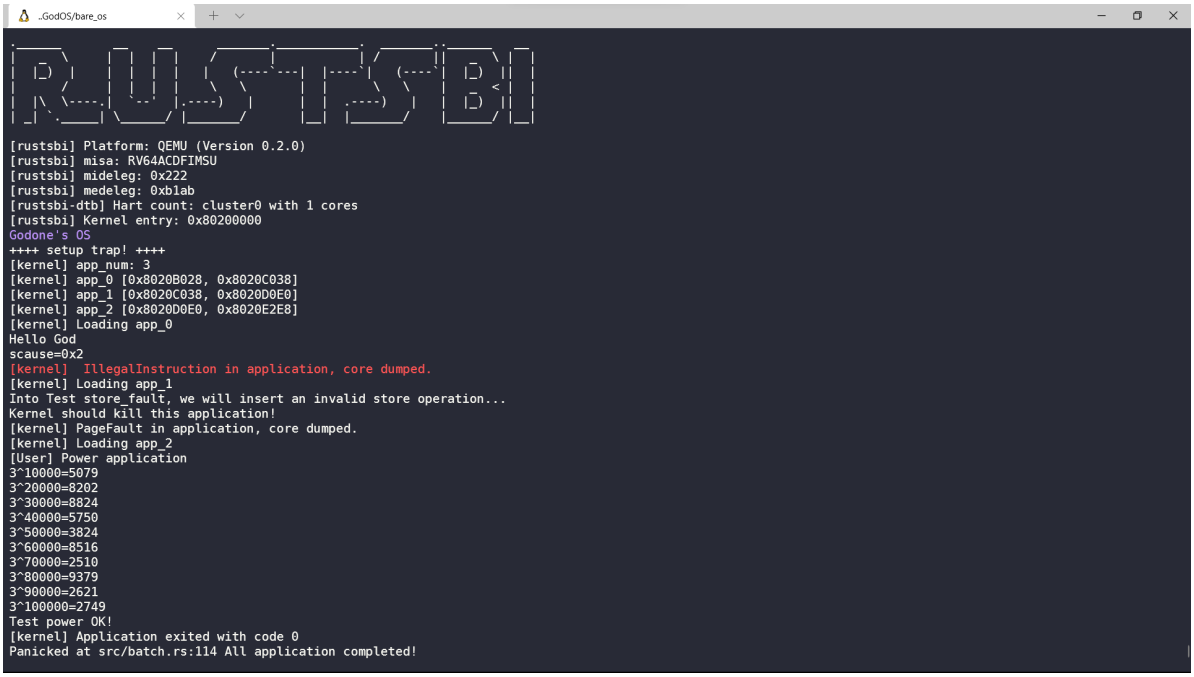
```
//初始应用管理器，答应应用地址
 batch::init();
```

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（）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。

最后整个操作系统的运行过程如下所示


../GodOS/bare_os
[rustsbi] Platform: QEMU (Version 0.2.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000
Godone's OS
++++ setup trap! ++++
[kernel] app_num: 3
[kernel] app_0 [0x8020B028, 0x8020C038]
[kernel] app_1 [0x8020C038, 0x8020D0E0]
[kernel] app_2 [0x8020D0E0, 0x8020E2E8]
[kernel] Loading app_0
Hello God
scause=0x2
[kernel] IllegalInstruction 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=5079
3^20000=8202
3^30000=8824
3^40000=5750
3^50000=3824
3^60000=8516
3^70000=2510
3^80000=9379
3^90000=2621
3^100000=2749
Test power OK!
[kernel] Application exited with code 0
Panicked at src/batch.rs:114 All application completed!