软硬件协同的用户态中断扩展

操作系统专题训练 技术报告

尤予阳　贺鲲鹏

清华大学

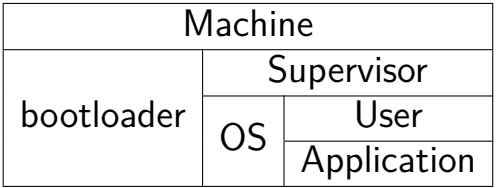
2021 年 9 月 27 日

背景

- 用户程序想知道外界发生了什么，或者通过主动轮询，或者通过内核通知
- 前者消耗 CPU 资源，后者延迟较大
- 能否由某种硬件机制执行通知过程？
- ——用户态中断

RISC-V 特权级架构

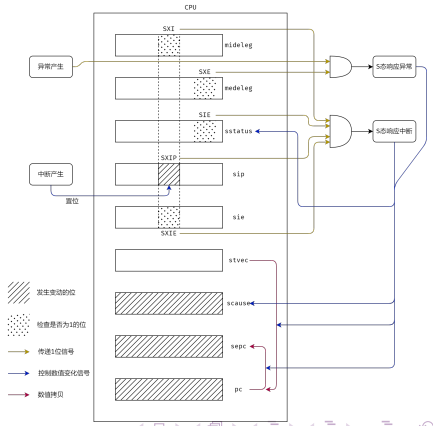
- MHSU 四层结构
  - H 暂未完全实现与应用
- 特权级不是必需的
- 通过中断和异常，陷入高特权级



特权级数	特权级	系统
1	M	简单嵌入式系统
2	M U	安全嵌入式系统
3	M S U	类 Unix 操作系统

RISC-V 中断规范

- 控制寄存器
  - xstatus
  - xie xip
  - xideleg xedeleg
  - xtvec xepc xcause xtval xscratch
- 特权指令
  - MRET SRET URET WFI ⋯

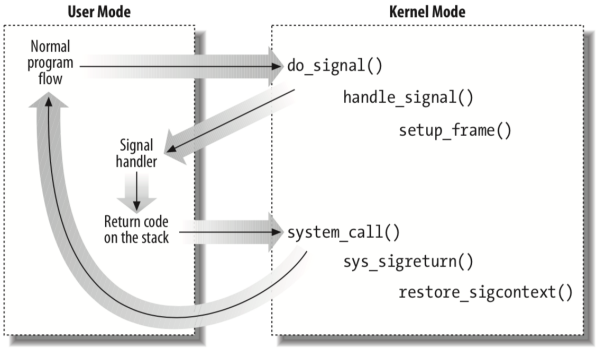


RISC-V N 扩展

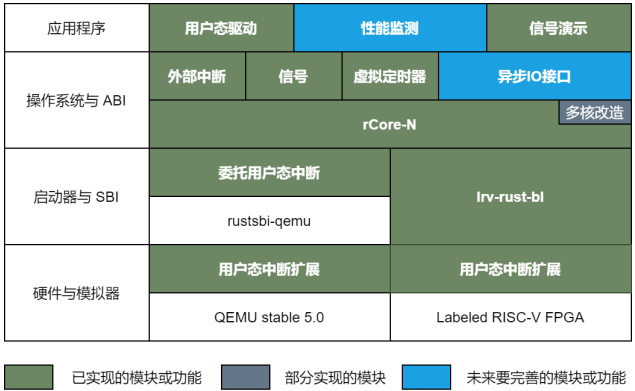
- 未完成的用户态中断扩展草案
- 设计初衷主要是为嵌入式系统提供安全性扩展
  - 可信代码运行在 M 态，不可信代码运行在 U 态
  - 允许不可信代码处理中断
- 在最新的 RISC-V 指令集手册 1.12-draft 中被移除
  - 部分原因是认为设计目标可以通过 M+bare S 来实现
  - 还有部分原因是没有人再推动这个扩展草案的完善和实现

Linux 信号

- 一种 IPC 机制
- 传递很短的一段信息
- 从内核态返回用户态时处理
- 内核软件模拟的中断机制
  - 注意与 RISC-V 的“软件中断”（Software Intterupt）区分



系统框架


应用程序
用户态驱动
性能监测
信号演示
操作系统与 ABI
外部中断
信号
虚拟定时器
异步IO接口
多核改造
rCore-N
启动器与 SBI
委托用户态中断
rustsbi-qemu
lrv-rust-bl
硬件与模拟器
用户态中断扩展
用户态中断扩展
QEMU stable 5.0
Labeled RISC-V FPGA
已实现的模块或功能
部分实现的模块
未来要完善的模块或功能

新增寄存器

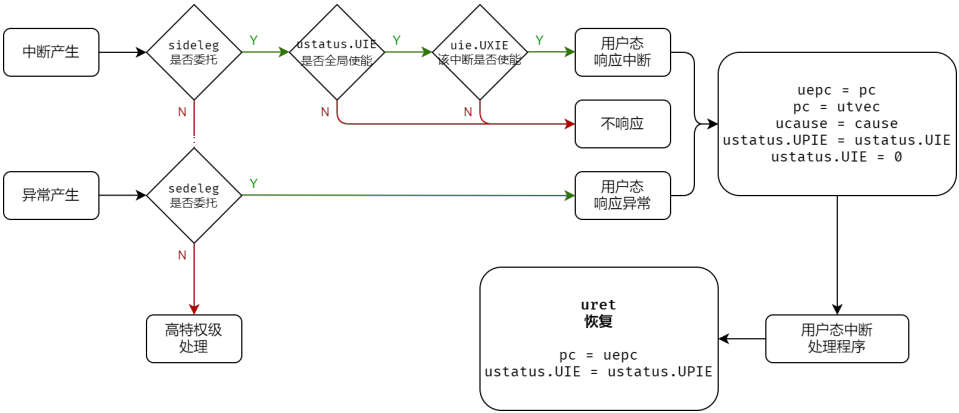
- 状态：ustatus
- 中断：uip, uie
- 陷入信息：uepc, ucause, utval
- 处理函数入口地址：utvec
- 委托：sedeleg, sideleg
- uscratch

新增指令

URET

- pc = uepc
- ustatus.UIE = ustatus.UPIE
- ustatus.UPIE = 1

用户态陷入的硬件处理流程



QEMU 和 FPGA 中的实现

- QEMU 基于 5.0 稳定版，FPGA 基于标签化 RISC-V 项目，核心为 Rocket Core
- 添加相关 CSR（ustatus, uie, uip, sideleg, sedeleg, ...）
- 添加处理逻辑（使能，屏蔽，跳转，委托等）
- 添加 URET 指令
- 添加 PLIC 上下文

QEMU 的开发环境

- Ubuntu 20.04，安装 QEMU 官方推荐的依赖
- 或者使用已经安装依赖的 docker 镜像
  duskmoon/dev-env:ubuntu-20.04-qemu-dep
- 修改后的 QEMU 5
  duskmoon314/qemu:riscv-N-stable-5.0

QEMU 5 缺什么功能？

- QEMU 5 支持 RISC-V v1.10 的中断规范
  - N 扩展（用户态中断）基本为草案，未实现
- 缺失部分
  - 用户态中断相关 CSR（存储结构和读写函数）
  - 用户态中断触发硬件逻辑
  - URET 指令
  - PLIC 触发 UEI 硬件逻辑和配置项

用户态 CSRs

- ustatus: 镜像 mstatus
- 使用 ustatus_mask 控制读写位

```
ustatus_mask = USTATUS_UIE | USTATUS_UPIE;

*val = env->mstatus & ustatus_mask;
newval = (env->mstatus & ~ustatus_mask)
       | (val & ustatus_mask);
```

用户态 CSRs

- sideleg sedeleg
- 只允许委托 M 委托给 S 的部分

```
env->sideleg = val & env->mideleg;

env->sedeleg = val & env->medeleg;
```

用户态 CSRs

- uip uie: 镜像 mip mie
- 使用 uip_writable_mask 和 sideleg 控制读写位

```
uip_writable_mask = MIP_USIP |
                    MIP_UEIP | MIP_UTIP;

env->mie & env->sideleg;
```

用户态中断触发逻辑

1. mip & mie != 0 => 存在待处理的中断/异常
2. 检查是否被委托，以及各特权级中断使能

```
irqs = (pending & ~env->mideleg & -mie) |
       (pending & env->mideleg &
        ~env->sideleg & -sie) |
       (pending & env->sideleg & -uie);
```

用户态中断触发逻辑

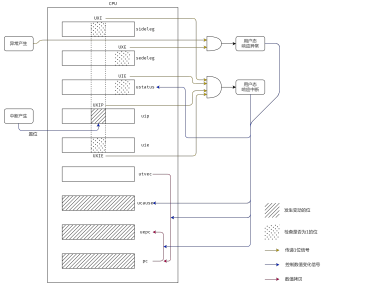
3. 检查是否符合用户态中断触发条件

```
if (riscv_has_ext(env, RVN) &&
    env->priv == PRV_U &&
    cause < TARGET_LONG_BITS &&
    ((sdeleg >> cause) & 1)) {
    // execute interrupt
```

用户态中断触发逻辑

4. “硬件”进行状态保存

```
s = env->mstatus;
s = set_field(s, MSTATUS_UPIE,
    get_field(s, MSTATUS_UIE));
s = set_field(s, MSTATUS_UIE, 0);
env->ucause = cause;
env->uepc = env->pc;
env->utval = tval;
env->pc = (env->utvec >> 2 << 2) +
    ((async &&
    (env->utvec & 3) == 1)
    ? cause * 4 : 0);
```



指令 URET

- 恢复硬件保存的状态

```
mstatus = set_field(mstatus, USTATUS_UIE,
    get_field(mstatus, USTATUS_UPIE));
mstatus = set_field(mstatus, USTATUS_UPIE, 1);

return env->uepc;
```

FPGA

需要实现的内容与 QEMU 中类似

```
549     read_mapping += CSRs.mideleg -> reg_mideleg
550     read_mapping += CSRs.medeleg -> reg_medeleg
551+
552+    val read_uie = reg_mie & reg_sideleg
553+    val read_uip = read_mip & reg_sideleg
554+    val read_ustatus = Wire(init = 0.U.asTypeOf(new
       MStatus))
555+
556+    read_ustatus.upie := io.status.upie
557+    read_ustatus.uie := io.status.uie
558+
559+    read_mapping += CSRs.ustatus ->  read_ustatus.asUInt
       (  xLen-1,0
560+    read_mapping += CSRs.uip -> read_uip.asUInt
561+    read_mapping += CSRs.uie -> read_uie.asUInt
562+    read_mapping += CSRs.uscratch -> reg_uscratch
563+    read_mapping += CSRs.ucause -> reg_ucause
564+    read_mapping += CSRs.utval -> reg_utval.sextTo xLen
565+    read_mapping += CSRs.uepc -> readEPC reg_uepc .
       sextTo xLen
566+    read_mapping += CSRs.utvec -> read_utvec
567+    read_mapping += CSRs.sideleg -> reg_sideleg
568+    read_mapping += CSRs.sedeleg -> reg_sedeleg
569   }
```

图 1: 读取 CSR

```
405   val d_interrupts = debug_int_assert << CSR.
      debugIntCause
406   val m_interrupts = Mux(reg_mstatus.prv <= PRV.S ||
      reg_mstatus.mie, ~(~pending_interrupts | reg_mideleg)
      , UInt(0))
407+  val s_interrupts = Mux(reg_mstatus.prv < PRV.S ||
    +  (reg_mstatus.prv === PRV.S && reg_mstatus.sie),
    +  pending_interrupts & reg_mideleg & ~reg_sideleg, UInt
    +  (0))
408+  val u_interrupts = Mux( reg_mstatus.prv === PRV.U &&
    +  reg_mstatus.uie , pending_interrupts & reg_sideleg,
    +  UInt(0))
409+  val (anyInterrupt, whichInterrupt) = chooseInterrupt
    +  (Seq(u_interrupts, s_interrupts, m_interrupts,
    +  d_interrupts))
410   val interruptMSB = BigInt(1) << (xLen-1)
411   val interruptCause = UInt(interruptMSB) +
      whichInterrupt
```

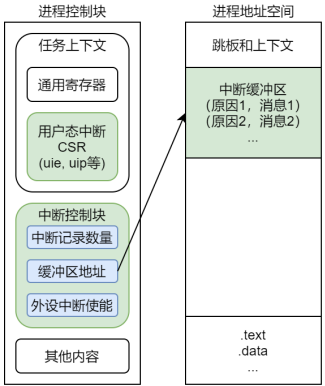
图 2: 中断委托

```
781   when (insn_ret) {
782+     when (Bool(usingVM) && !io.rw.addr(9) && !io.rw.addr
    +     (8)) {
783+       reg_mstatus.uie := reg_mstatus.upie
784+       reg_mstatus.upie := true
785+       new_prv := PRV.U
786+       io.evec := readEPC(reg_uepc)
787+     }.elsewhen (Bool(usingVM) && !io.rw.addr(9)) {
788       reg_mstatus.sie := reg_mstatus.spie
789       reg_mstatus.spie := true
790       reg_mstatus.spp := PRV.U
791       new_prv := reg_mstatus.spp
792       io.evec := readEPC(reg_sepc)
793     }.elsewhen (Bool(usingDebug) && io.rw.addr(10)) {
794       new_prv := reg_dcsr.prv
795       reg_debug := false
796       io.evec := readEPC(reg_dpc)
797     }.otherwise {
798       reg_mstatus.mie := reg_mstatus.mpie
799       reg_mstatus.mpie := true
800       reg_mstatus.mpp := legalizePrivilege(PRV.U)
801       new_prv := reg_mstatus.mpp
802       io.evec := readEPC(reg_mepc)
803     }
804   }
```

图 3: URET

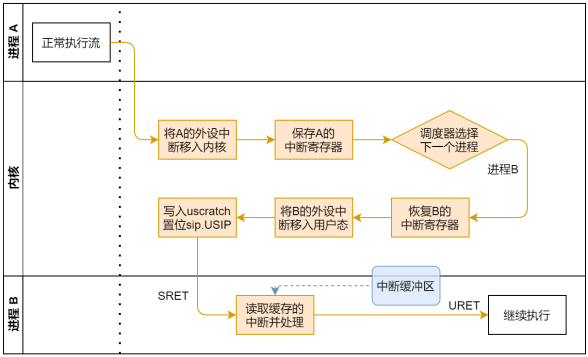
用户态中断上下文

- 记录每个进程的用户态中断 CSR、中断缓冲区和中断记录数目
- 中断缓冲区为一个内存页
- 一条中断记录包括原因和附加消息
  - 时钟中断和外部中断原因分别为 4 和 8，与 xcause 寄存器编码保持一致
  - 外部中断附加消息为中断外设号
  - 信号的中断原因为源进程 PID « 4



进程切换

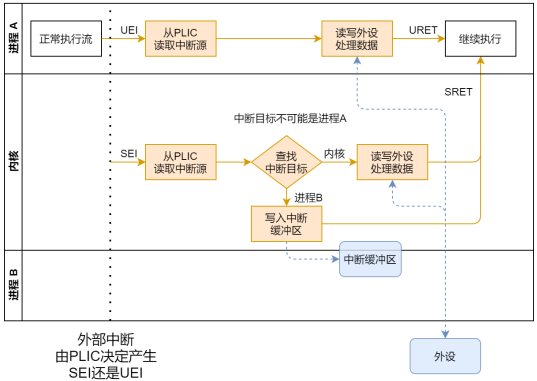
- 进程切换时，保存当前进程的中断 CSR，恢复下一进程的中断 CSR，以及外设中断使能配置
- 从内核返回用户态时，将缓存的中断数量写入 uscratch 寄存器
- URET 返回正常执行流，无需系统调用



进入调度
抢占或主动让出

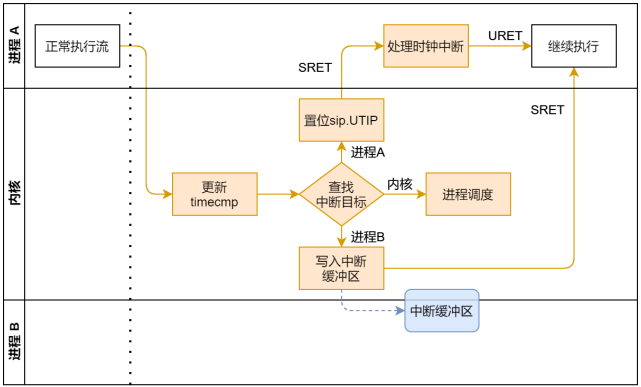
外部中断

- 内核记录每个外设对应的进程编号
- 如果外设对应的驱动进程正在 CPU 上运行，PLIC 直接产生 UEI，无需经过内核
- 否则产生 SEI，由内核转发



时钟中断

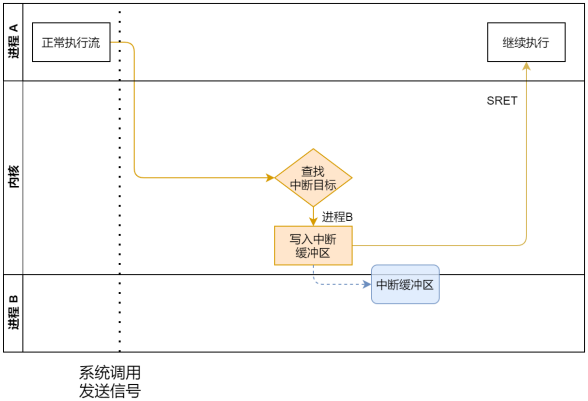
- 内核根据定时器到期时间维护一个有序列表
- 产生中断时传递给到期时间最早的申请者



时钟中断

信号发送

- 多核情况下，若目标进程正在另一核上运行，可直接发送跨核中断，无需等待下一次调度



新增系统调用

- init_user_trap()
  - 分配缓冲区页，初始化进程中断信息结构体
- send_msg(pid, msg)
  - 向 pid 对应进程发送一条消息 msg，相当于 signal()
- set_timer(time_us)
  - 设置一个 time_us 后到期的定时器，相当于 alarm()
- claim_ext_int(device_id)
  - 注册成为 device_id 对应外设的用户态驱动
  - 将 PLIC 对应的 claim/complete 寄存器地址区间和外设的地址区间映射到用户进程

用户程序库

- 提供跳板代码和缺省的中断处理函数
- 前者作为中断入口地址被写入 utvec
- 后者被标记为弱链接，用户进程可以定义同名的函数，这样跳板代码会跳转到用户的处理函数
- 用户进程也可以修改 utvec，使用自定义跳板

用户程序库

```
__alltraps_u:
    # csrw uscratch, sp
    addi sp, sp, -35*8; # sp = sp + -35*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, ustatus
    csrr t1, uepc
    csrr t2, utvec
    sd t0, 32*8(sp)
    sd t1, 33*8(sp)
    sd t2, 34*8(sp)
    csrr t3, uscratch
    sd t3, 2*8(sp)
    mv  a0, sp # a0 = sp
    call user_trap_handler
```

图 4: 跳板代码

```
#[linkage = "weak"]
#[no_mangle]
pub fn user_trap_handler(cx: &mut UserTrapContext) -> &mut UserTrapContext {
    let ucause: Ucause = ucause::read();
    let utval: usize = utval::read();
    match ucause.cause() {
        ucause::Trap::Interrupt(ucause::Interrupt::UserSoft) => {
            println!("[user mode trap] user soft");
            unsafe {
                uip::clear_usoft();
            }
        }
        _ => {
            println!(
                "Unsupported trap {:?}, utval = {:#x}, uepc = {:#x}!",
                ucause.cause(),
                utval,
                uepc::read()
            );
        }
    }
    cx
}
```

图 5: 缺省的中断处理函数

演示程序

- 用户态的串口驱动
  - 串口接收到输入时产生中断
  - 驱动将用户输入回显到串口
  - 打印自己收到的信号，并在收到 15 时退出
- 信号和定时器演示
  - 设置 10 个间隔 1 秒的定时器
  - 在时钟中断处理函数中，向用户态串口驱动发送一条消息
  - 在最后一次定时器生效时发送 15 (SIGTERM)

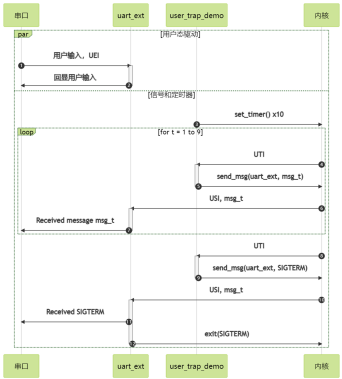


图 6: 演示程序执行流

演示程序


[DEBUG] : restore user trap
[user trap demo] user soft interrupt, num: 1
[user trap demo] cause: 4, message 209462206
[user trap demo] sending msg: deadbeef06
[DEBUG] : [push trap record] pid: 2, cause: 32, message: 956397711110
[DEBUG] : restore user trap
[uart ext] user soft interrupt
[uart ext] user external interrupt, irq: 9
[uart ext] user external interrupt, irq: 9
[uart ext] user external interrupt, irq: 9
[uart ext] user external interrupt, irq: 9
[uart ext] user external interrupt, irq: 9
[uart ext] user external interrupt, irq: 9
[uart ext] user external interrupt, irq: 9
[uart ext] user external interrupt, irq: 9
[uart ext] user external interrupt, irq: 9
[uart ext] user external interrupt, irq: 9
[user trap demo] user timer interrupt at 17757 ms
[user trap demo] sending msg: deadbeef07
[DEBUG] : [push trap record] pid: 2, cause: 32, message: 956397711111
[DEBUG] : restore user trap
[uart ext] user soft interrupt
[uart ext] user external interrupt, irq: 9
[DEBUG] : [push trap record] pid: 1, cause: 4, message: 234464554
[DEBUG] : restore user trap
[user trap demo] user soft interrupt, num: 1
[user trap demo] cause: 4, message 234464554
[user trap demo] sending msg: deadbeef08
[DEBUG] : [push trap record] pid: 2, cause: 32, message: 956397711112
[DEBUG] : restore user trap
[uart ext] user soft interrupt
[user trap demo] user timer interrupt at 19757 ms
[user trap demo] sending msg: deadbeef09
[DEBUG] : [push trap record] pid: 2, cause: 32, message: 956397711113
[DEBUG] : restore user trap
[uart ext] user soft interrupt
[DEBUG] : [push trap record] pid: 1, cause: 4, message: 259468042
[DEBUG] : restore user trap
[user trap demo] user soft interrupt, num: 1
[user trap demo] cause: 4, message 259468042
[user trap demo] sending SIGTERM
[DEBUG] : [push trap record] pid: 2, cause: 32, message: 15
[DEBUG] : pid: 1 exited with code 0
[DEBUG] : restore user trap
[uart ext] user soft interrupt
[DEBUG] : pid: 2 exited with code 15
Shell: Process 1 exited with code 0
>>
串口收到的内容及回显
串口经PLIC直接产生的用户态外部中断
接收到的内核直接写uip.UTIP注入的时钟中断
并发送相应的IPC信息
内核通过中断硬冲队列注入的时钟中断
cause 4 对应用户态时钟中断的编号
内核注入信号，及两个进程的退出返回值
Hello from UART1!
[uart ext] Received message 0xdeadbeef01 from pid 2
[uart ext] Received message 0xdeadbeef02 from pid 2
[uart ext] Received message 0xdeadbeef03 from pid 2
[uart ext] Received message 0xdeadbeef04 from pid 2
[uart ext] Received message 0xdeadbeef05 from pid 2
[uart ext] Received message 0xdeadbeef06 from pid 2
hello world
[uart ext] Received message 0xdeadbeef07 from pid 2
hello world
[uart ext] Received message 0xdeadbeef08 from pid 2
[uart ext] Received message 0xdeadbeef09 from pid 2
[uart ext] Received SIGTERM, exiting...
接收到的的IPC消息和信号
[uart ext] 为用户态串口驱动进程，接收串口产生的外部中断，将用户输入回显到串口
[user trap demo] 进程注册了10个间隔1秒的定时器，到时产生用户态时钟中断，在中断处理函数中向 [uart ext] 发送IPC
[DEBUG] 为内核调试输出

后续工作

- 性能测试，针对基于用户态中断实现的驱动、IPC 等
- 更多硬件支持，进一步分担内核在管理中断时的负担
- 适配 Linux 内核
- 与硬件虚拟化扩展的联系
- 基于用户态中断的异步内核

项目信息

- repo：https://github.com/Gallium70/rv-n-ext-impl/
- 文档：https://gallium70.github.io/rv-n-ext-impl/

Thanks!