P2-part3-4 的总结

思路总结

在 debug 的过程中，发现开启时钟中断之后，大概能够运行 30 秒左右出现报错。

报错的种类大致集中在三个地方。

1. interrupt_helper,

epc = 0 报错，显示返回地址是 interrupt_helper 最终结束的 nop指令

2. handle_syscall

epc = 0 报错，显示返回地址是 handle_syscall 最终结束的 nop 指令

3. printk 调度的 strlen

错误出现在 lbu a5, 0(a5)

发现错误的分支，大致都是走了不期望进入的另一种分支。

因此，debug 的方法，直接打印 interrupt_helper 的 5 个参数，

观察执行前和返回后，参数值的变化情况。

经过打印发现，在某次执行后，arg0, arg1, arg2, arg3, arg4, 也就是 a0-a5 寄存器中，出现了除系统调用号，(long)buff 等参数外，

出现了类似 SCAUSE, SCAUSE + 1(时钟中断时的 scause = 0x8000000005)

非常明显的传递参数有误。

参数传递有误

参数传递有误的检查分为两种：

1. 检查对应寄存器的值是否正确
2. 打印对应 sp，以及 sp 附近的值，观察是否正确。

逻辑梳理

第一步，对于中断处理程序而言，进入后，需要将 sp 改为内核栈对应的指针。

第二步，对于中断处理程序而言，退出前，恢复现场前，需要将sp的值保存进内核栈

SAVE_CONTEXT 部分的修改

save_context 叫做保存现场，就是把当前的 32 个寄存器的值，写入到 pcb 对应的用户结构体里。

之前的写法使用了 t0 寄存器作为 pcb 对应用户结构体地址的索引，

但实际上，如果需要使用 t0 的值，但 t0 的值已经被修改了。

就丢失了原始的 t0 的值，保存现场最多只保存 31 个有效的寄存器，引起传参错误。

解决方案: tp 寄存器不会被用户程序操作，而且只在 switch_to 的时候进行赋值，

统一使用 tp 寄存器作为索引，tp 寄存器在保存和恢复现场时不去修改。

```
.macro SAVE_CONTEXT
    addi tp, tp, PCB_USER_REGS_CONTEXT_OFFSET

    sd zero, OFFSET_REG_ZERO(tp)
    sd ra,   OFFSET_REG_RA(tp)
    sd sp,   OFFSET_REG_SP(tp)
    sd gp,   OFFSET_REG_GP(tp)
    sd t0,   OFFSET_REG_T0(tp)
    sd t1,   OFFSET_REG_T1(tp)
    sd t2,   OFFSET_REG_T2(tp)
    sd s0,   OFFSET_REG_S0(tp)
    sd s1,   OFFSET_REG_S1(tp)
    sd a0,   OFFSET_REG_A0(tp)

    sd a1,   OFFSET_REG_A1(tp)
    sd a2,   OFFSET_REG_A2(tp)
    sd a3,   OFFSET_REG_A3(tp)
    sd a4,   OFFSET_REG_A4(tp)
    sd a5,   OFFSET_REG_A5(tp)
    sd a6,   OFFSET_REG_A6(tp)
    sd a7,   OFFSET_REG_A7(tp)
    sd s2,   OFFSET_REG_S2(tp)
    sd s3,   OFFSET_REG_S3(tp)
    sd s4,   OFFSET_REG_S4(tp)

    sd s5,   OFFSET_REG_S5(tp)
    sd s6,   OFFSET_REG_S6(tp)
    sd s7,   OFFSET_REG_S7(tp)
    sd s8,   OFFSET_REG_S8(tp)
    sd s9,   OFFSET_REG_S9(tp)
    sd s10,  OFFSET_REG_S10(tp)
    sd s11,  OFFSET_REG_S11(tp)
    sd t3,   OFFSET_REG_T3(tp)
    sd t4,   OFFSET_REG_T4(tp)
    sd t5,   OFFSET_REG_T5(tp)

    sd t6,   OFFSET_REG_T6(tp)

```

```
    csrr s0,  CSR_SSTATUS
    sd s0,    OFFSET_REG_SSTATUS(tp)

    csrr s0,   CSR_SEPC
    sd s0,    OFFSET_REG_SEPC(tp)

    csrr s0,    CSR_SCAUSE
    sd s0,    OFFSET_REG_SCAUSE(tp)
    addi tp, tp, -(PCB_USER_REGS_CONTEXT_OFFSET)
.endm
```

保存现场时，顺便保存 sstatus, sepc, scause 寄存器。

这里对当前进程的 CSR 寄存器的处理。

1. 在进入异常时，保存现场，存储当前进程对应的 CSR 寄存器
2. 在离开异常后，恢复现场，读取当前进程对应的 CSR 寄存器。

RESTORE_CONTEXT 部分的修改

这里是把内存的值读取到寄存器，因此即使在之前修改了 t0 寄存器，只要最后恢复了 t0 寄存器的值，就可以使用。

```
.macro RESTORE_CONTEXT
    addi t0, tp, PCB_USER_REGS_CONTEXT_OFFSET

    ld zero,    OFFSET_REG_ZERO(t0)
    ld ra,      OFFSET_REG_RA(t0)
    ld sp,      OFFSET_REG_SP(t0)
    ld gp,      OFFSET_REG_GP(t0)
    ld t1,      OFFSET_REG_T1(t0)
    ld t2,      OFFSET_REG_T2(t0)
    ld s1,      OFFSET_REG_S1(t0)
    ld a0,      OFFSET_REG_A0(t0)
    ld a1,      OFFSET_REG_A1(t0)
    ld a2,      OFFSET_REG_A2(t0)

    ld a3,      OFFSET_REG_A3(t0)
    ld a4,      OFFSET_REG_A4(t0)
    ld a5,      OFFSET_REG_A5(t0)
    ld a6,      OFFSET_REG_A6(t0)
    ld a7,      OFFSET_REG_A7(t0)
    ld s2,      OFFSET_REG_S2(t0)
    ld s3,      OFFSET_REG_S3(t0)
    ld s4,      OFFSET_REG_S4(t0)
    ld s5,      OFFSET_REG_S5(t0)
```

```
    ld s6,      OFFSET_REG_S6(t0)

    ld s7,      OFFSET_REG_S7(t0)
    ld s8,      OFFSET_REG_S8(t0)
    ld s9,      OFFSET_REG_S9(t0)
    ld s10,     OFFSET_REG_S10(t0)
    ld s11,     OFFSET_REG_S11(t0)
    ld t3,      OFFSET_REG_T3(t0)
    ld t4,      OFFSET_REG_T4(t0)
    ld t5,      OFFSET_REG_T5(t0)
    ld t6,      OFFSET_REG_T6(t0)

    ld s0,    OFFSET_REG_SSTATUS(t0)
    csrw CSR_SSTATUS, s0

    ld s0,    OFFSET_REG_SCAUSE(t0)
    csrw CSR_SCAUSE, s0

    ld s0,    OFFSET_REG_S0(t0)
    ld t0,    OFFSET_REG_T0(t0)
.endm
```

恢复现场中，暂时没有涉及写 sepc 寄存器的处理逻辑。

因为在 ret_from_exception 中，需要根据条件判断是否直接 sepc + 4,

还是 sepc 不变。这里需要写 sepc, 不如先把逻辑都设置好。

因为一旦恢复现场，这些寄存器的值都不能被再次修改了。

如果前面写了 sepc = sepc + 4, 但是写 sepc 时，又重新写成了 sepc = sepc,

相当于前面的修改被覆盖，没有起到作用。

ret_from_exception 部分

```
ENTRY(ret_from_exception)
    mv t0, tp                                   // 离开异常处理，需要将 sp 指针存
    addi t0, t0, PCB_SWICH_TO_CONTEXT_OFFSET    // 还没有恢复现场，t0随便用
    sd sp, SWITCH_TO_SP(t0)                     // 如果恢复现场后再使用 t0, t0 的现

    addi t0, tp, PCB_USER_REGS_CONTEXT_OFFSET   // scause 和 sepc 需要从当前进程
    ld a0, OFFSET_REG_SCAUSE(t0)
    ld s0, OFFSET_REG_SEPC(t0)
    csrw CSR_SEPC, s0

    blt a0, zero, interrupt_case
```

```
    li a1, 0x8
    bne a0, a1, interrupt_case

    addi s0, s0, 4
    csrw CSR_SEPC, s0                          // 如果需要修改，直接写 sepc

interrupt_case:
    RESOTRE_CONTEXT                           // 恢复现场之前， sepc 已经被恢复了
    sret                                      // sp 也设置成了用户栈的  sp
ENDPROC(ret_from_exception)
```

1. 退出中断处理程序，把内核栈的 sp 存储起来。
2. 因为还需要使用一些寄存器作为位置索引，所以把恢复现场放在最后。
3. 恢复现场放在最后，导致真正的 scause, sepc 寄存器的值，还没有被恢复，需要先读出来。
4. 不在恢复现场里写 sepc 寄存器，提前设置写的逻辑，将sepc = sepc + 4 和 sepc 分情况写入，防止恢复现场时覆盖了 sepc += 4 的情况。

Exception_handler_entry 部分

```
ENTRY(exception_handler_entry)
    SAVE_CONTEXT
    mv t1, tp
    addi t1, t1, PCB_SWITCH_TO_CONTEXT_OFFSET
    ld sp, SWITCH_TO_SP(t1)

    mv t0, tp
    addi t0, t0, PCB_USER_REGS_CONTEXT_OFFSET

    sd a0, OFFSET_REG_A0(t0)
    sd a1, OFFSET_REG_A1(t0)
    sd a2, OFFSET_REG_A2(t0)
    sd a3, OFFSET_REG_A3(t0)
    sd a4, OFFSET_REG_A4(t0)
    sd a5, OFFSET_REG_A5(t0)

    ld a0, OFFSET_REG_REGS_POINTER(t0)
    csrr a1, CSR_STVAL
    csrr a2, CSR_SCAUSE
    call interrupt_helper
    j ret_from_exception
ENDPROC(exception_handler_entry)
```

1. 进入中断处理程序，一上来先保存现场，保存之后可以随意使用寄存器

与之对应的，恢复现场之前可以随意使用寄存器，一旦恢复现象，这些寄存器的值就不能再变了。

2. 将 sp 写成内核栈的 sp，表示这里由内核态进行处理
3. 在 call interrupt_helper 之前，即使写了 ra, 也会在执行 interrupt_helper 的过程中被修改，所以 jr ra 没有起作用。

涉及带返回值的 用户态程序，例如 sys_mutex_init

```
1 static long invoke_syscall(long sysno, long arg0, long arg1, long arg2, long arg
2 long arg4)
3 {
4     asm volatile("nop");
5     asm volatile("ecall");
6     register uintptr_t a0 asm("a0");
7     return a0;
8 }
```

关于 a0 返回值的一个 bug

调度顺序

sys_mutex_init ->

invoke_syscall ->

[exception_handler_entry] ->

interrupt_helper ->

handle_syscall ->

handle_syscall 给出返回值 a0,

ret_from_exception

```
1 int sys_mutex_init(){
2     long invoke_syscall(){
3             void exception_handler_entry(){
4 
5                 void interrupt_helper(){
6                         void handle_syscall(){
7                             syscall[syscall_num](arg0, arg1, arg3,
8                             arg3, arg4);
9                             register uintptr_t a0 asm("a0");
```

```
10                              regs -> regs[10] = a0;
11                          }
12                  }
13                  void ret_from_exception(){
14                      RESTORE_CONTEXT;
15                  }
16              }
17          return a0;
18      }
19      return invoke_syscall();
20  }
```

因此在 handle_syscall 需要通过 a0 寄存器返回，

但是最终返回之前，要经过 RESTORE_CONTEXT 的恢复现场，a0 的值被修改为 OFFSET_REG_A0(t0) 中的值，

最稳妥的写法是，直接在 handle_syscall() 执行结束后，将 a0 的值写入到寄存器 OFFSET_REG_A0(t0) 中。