分支任务：gdb 调试系统调用以及返回

ecall调试过程

调试环境搭建

我们使用三个终端并行进行调试：

终端 1 (QEMU运行端):

- 执行 `make debug` 命令启动QEMU并挂起系统

终端 2 (硬件调试端):

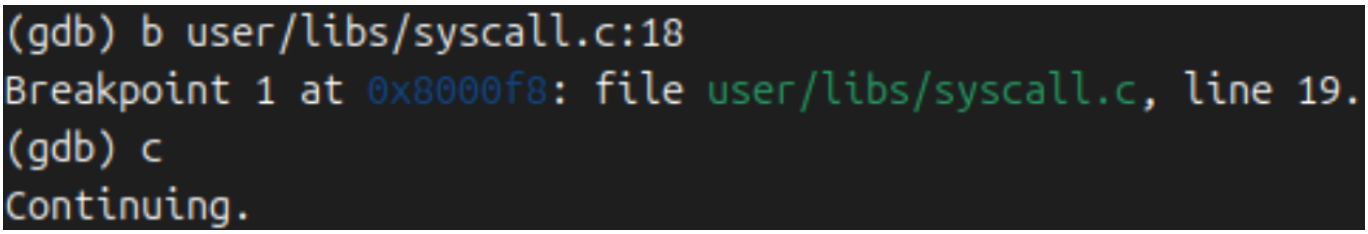
```
pgrep -a qemu        # 查找QEMU进程PID
sudo gdb             # 启动GDB调试器
attach <PID>         # 附加到QEMU进程
handle SIGPIPE nostop noprint  # 忽略SIGPIPE信号干扰
continue             # 继续执行等待断点
```

终端 3 (内核调试端):

```
make gdb                      # 启动内核调试会话
set remotetimeout unlimited   # 设置无限远程超时
add-symbol-file obj/__user_exit.out  # 加载用户态程序符号表
```

用户态系统调用跟踪

由于系统调用代码属于用户态程序，默认的内核符号表不包含这些信息。手动加载用户符号表后，GDB成功识别了`user/libs/syscall.c`文件，允许我们在该文件设置断点：



- 程序在`syscall`函数入口处断点停止
- 使用`x/8i $pc`查看汇编指令流，发现程序正通过`ld`指令将参数加载到`a0`-`a5`寄存器
- 通过6次连续的`si`指令单步执行，精准控制CPU执行流

```
Breakpoint 1, syscall (num=num@entry=30) at user/libs/syscall.c:19
19          asm volatile (
(gdb) x/10i $pc
=> 0x8000f8 <syscall+32>:       ld      a0,8(sp)
   0x8000fa <syscall+34>:       ld      a1,40(sp)
   0x8000fc <syscall+36>:       ld      a2,48(sp)
   0x8000fe <syscall+38>:       ld      a3,56(sp)
   0x800100 <syscall+40>:       ld      a4,64(sp)
   0x800102 <syscall+42>:       ld      a5,72(sp)
   0x800104 <syscall+44>:       ecall
   0x800108 <syscall+48>:       sd      a0,28(sp)
   0x80010c <syscall+52>:       lw      a0,28(sp)
   0x80010e <syscall+54>:       addi    sp,sp,144
```

最终停在地址`0x800104`处，此时下一条待执行指令正是`ecall`，用户态参数已全部就绪，等待系统调用跳转。

捕获ecall异常

在终端2中：

1. 首先按Ctrl+C中断执行
2. 设置断点1：`break riscv_raise_exception`
3. 设置断点2：`break riscv_cpu_do_interrupt`
4. 输入`c`继续执行

```
(gdb) ^C
Thread 1 "qemu-system-ris" received signal SIGINT, Interrupt.
0x000073d0de91ba30 in __GI_ppoll (fds=0x63fec7382ab0, nfds=6, timeout=<optimized out>, sigmask=0x0) a
42      in ../sysdeps/unix/sysv/linux/ppoll.c
(gdb) break riscv_raise_exception
Breakpoint 1 at 0x63fea687dffa: file /home/yang/riscv/qemu-4.1.1/target/riscv/op_helper.c, line 31.
(gdb) break riscv_cpu_do_interrupt
Breakpoint 2 at 0x63fea687fc87: file /home/yang/riscv/qemu-4.1.1/target/riscv/cpu_helper.c, line 507.
(gdb) c
Continuing.
```

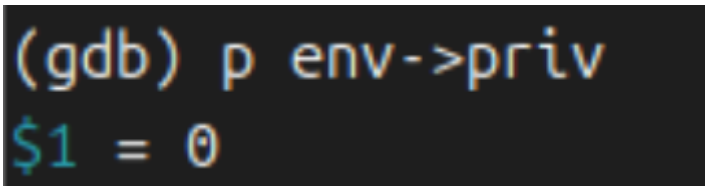
在终端3按下`si`执行`ecall`指令后，硬件端断点立即触发：

```
[Switching to Thread 0x73d0ddee36c0 (LWP 2699)]

Thread 2 "qemu-system-ris" hit Breakpoint 1, riscv_raise_exception (env=0x63fec733f040, exception=8, pc=0)
    at /home/yang/riscv/qemu-4.1.1/target/riscv/op_helper.c:31
31          CPUState *cs = env_cpu(env);
```

验证异常号`exception=8`确认捕获成功： 异常号8对应`RISCV_EXCP_U_ECALL`，代表成功捕获了用户态程序向内核发起的系统调用请求。

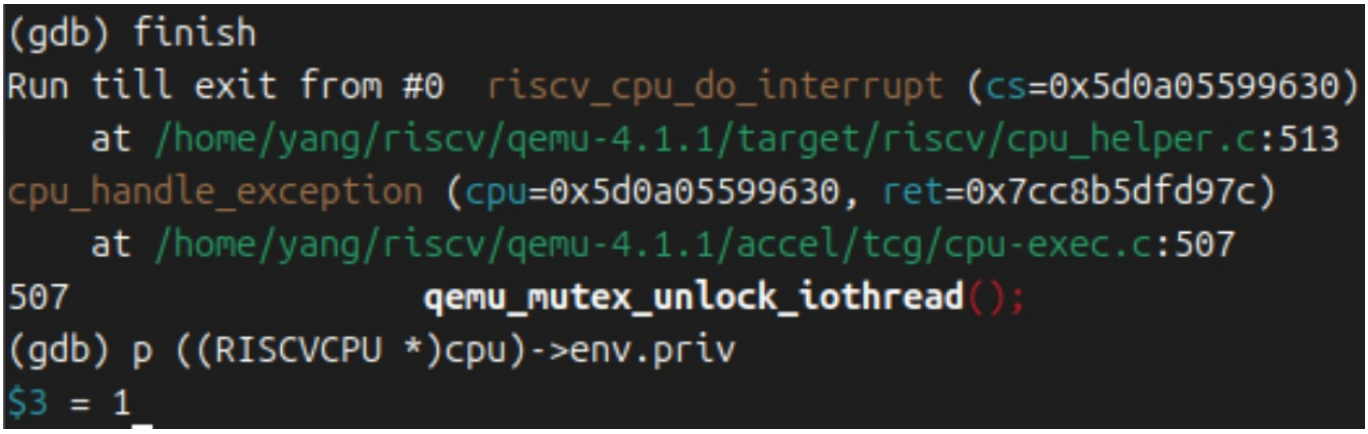
特权级切换验证

继续执行到env被赋值后，检查当前特权级：



此处env代表QEMU中虚拟CPU的状态结构体指针，priv成员表示当前特权级（0=用户态，1=监督态，2=保留，3=机器态）。priv=0证实CPU当前确实处于用户态(U-Mode)。

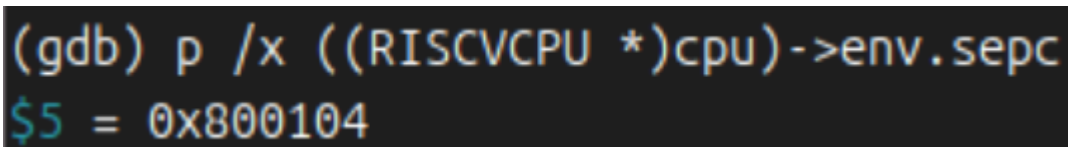
执行finish指令让QEMU完成riscv_cpu_do_interrupt函数的执行：



函数返回后，我们通过强制类型转换验证CPU上下文

```
(gdb) p ((RISCVCPU *)cpu)->env.priv
$3 = 1
```

priv从0变为1，证明CPU已成功从用户态切换至内核态(S-Mode)。



sepc寄存器精准记录了触发异常的ecall指令地址(0x800104)，确保未来sret能正确返回。

QEMU源码分析

核心函数一：riscv_raise_exception (抛出异常)

- 文件位置：target/riscv/op_helper.c
- 触发时机：ecall指令执行时被调用
- 功能：接收翻译阶段传来的异常号（本实验中为RISCV_EXCP_U_ECALL = 8），并将异常状态标记在CPU的环境结构体(env)中，随后通知CPU主循环处理中断

核心函数二：riscv_cpu_do_interrupt (处理中断)

- 文件位置：target/riscv/cpu_helper.c
- 触发时机：CPU主循环检测到有异常挂起时调用
- 功能：执行RISC-V硬件手册规定的所有"Trap发生时硬件自动完成的动作"

关键代码逻辑分析：

```
void riscv_cpu_do_interrupt(CPUState *cs)
{
...
    /*
     *  判断陷阱应该在哪个特权级处理
     * 条件：当前特权级 ≤ S模式 且 陷阱被委托到S模式
     */
    if (env->priv <= PRV_S &&
        cause < TARGET_LONG_BITS && ((deleg >> cause) & 1)) {

        /* ---------- 在S模式处理陷阱 ---------- */

        target_ulong s = env->mstatus;  // 获取当前mstatus值

        /* 保存当前中断使能状态到SPIE */
        s = set_field(s, MSTATUS_SPIE, env->priv_ver >= PRIV_VERSION_1_10_0 ?
            get_field(s, MSTATUS_SIE) : get_field(s, MSTATUS_UIE << env->priv));

        s = set_field(s, MSTATUS_SPP, env->priv);  // 保存当前特权级到SPP
        s = set_field(s, MSTATUS_SIE, 0);           // 禁用S模式中断

        env->mstatus = s;  // 更新mstatus

        /* 设置S模式陷阱相关CSR寄存器 */
        env->scause = cause | ((target_ulong)async << (TARGET_LONG_BITS - 1));  //
原因+异步标志
        env->sepc = env->pc;          // 保存返回地址
        env->sbadaddr = tval;         // 保存额外信息

        /*
         * 跳转到S模式陷阱处理程序：
         * - 直接模式(stvec[1:0]=00): 跳转到stvec[XLEN-1:2] << 2
         * - 向量模式(stvec[1:0]=01): 跳转到stvec_base + cause*4
         */
        env->pc = (env->stvec >> 2 << 2) +
            ((async && (env->stvec & 3) == 1) ? cause * 4 : 0);

        riscv_cpu_set_mode(env, PRV_S);  // 切换到S模式

    } else {
        /* ---------- 在M模式处理陷阱 ---------- */

        target_ulong s = env->mstatus;  // 获取当前mstatus值

        /* 保存当前中断使能状态到MPIE */
        s = set_field(s, MSTATUS_MPIE, env->priv_ver >= PRIV_VERSION_1_10_0 ?
            get_field(s, MSTATUS_MIE) : get_field(s, MSTATUS_UIE << env->priv));

        s = set_field(s, MSTATUS_MPP, env->priv);  // 保存当前特权级到MPP
        s = set_field(s, MSTATUS_MIE, 0);           // 禁用M模式中断
```

```
        env->mstatus = s;  // 更新mstatus

        /* 设置M模式陷阱相关CSR寄存器 */
        env->mcause = cause | ~(((target_ulong)-1) >> async);  // 原因+异步标志（高
位）
        env->mepc = env->pc;          // 保存返回地址
        env->mbadaddr = tval;         // 保存额外信息

        /*
         * 跳转到M模式陷阱处理程序:
         * - 直接模式(mtvec[1:0]=00): 跳转到mtvec[XLEN-1:2] << 2
         * - 向量模式(mtvec[1:0]=01): 跳转到mtvec_base + cause*4
         */
        env->pc = (env->mtvec >> 2 << 2) +
            ((async && (env->mtvec & 3) == 1) ? cause * 4 : 0);

        riscv_cpu_set_mode(env, PRV_M);  // 切换到M模式
    }

#endif  /* !CONFIG_USER_ONLY */

    cs->exception_index = EXCP_NONE;  // 标记异常已处理，QEMU可以继续执行
}
```

在`riscv_cpu_do_interrupt`函数中，QEMU首先检查`medeleg`(委托寄存器)。由于uCore运行在S模式，且`ecall`异常被委托给了S模式，代码进入了第一个`if`分支，执行了与GDB调试完全一致的行为。

sret调试过程

我们继续对sret进行调试

尝试在`__trapret`符号处设置断点时，我们遇到了符号查找问题：

```
(gdb) info address __trapret
Symbol "__trapret" is at 0xffffffffc0200eb4 in a file compiled without debugging
```

问题分析： GDB提示`__trapret`符号位于一个没有调试信息的文件中。这表明当前加载的内核镜像可能不包含完整的调试符号，或者调试信息不完整。这通常是由于：

1. 内核编译时未包含完整的调试信息（-g选项）
2. 当前加载的符号表文件不正确
3. 内核镜像文件与符号表不匹配

解决方案： 我们可以通过以下步骤确认符号地址并设置断点：

1. **确认符号地址：**

```
(gdb) info address __trapret
Symbol "__trapret" is at 0xffffffffc0200eb4 in a file compiled without
```

```
debugging
```

2. **使用获取的地址直接设置硬件断点：**

```
(gdb) b *0xffffffffc0200eb4
Breakpoint 2 at 0xffffffffc0200eb4
(gdb) c
```

3. **查看汇编代码确认是否为sret指令：**

```
(gdb) x/10i 0xffffffffc0200eb4
0xffffffffc0200eb4 <__trapret>:    addi    sp,sp,-272
0xffffffffc0200eb8 <__trapret+4>:  sd      ra,264(sp)
0xffffffffc0200ebc <__trapret+8>:  sd      sp,256(sp)
...
0xffffffffc0200f0a <__trapret+86>: sret
```

通过反汇编可以看到，`__trapret`函数的开头地址是`0xffffffffc0200eb4`，而`sret`指令位于`0xffffffffc0200f0a`。

4. **直接在sret指令地址设置断点：**

```
(gdb) b *0xffffffffc0200f0a
Breakpoint 3 at 0xffffffffc0200f0a: file kern/trap/trapentry.S, line 133.
(gdb) c
```

最后我们查找源代码，直接在sret指令地址设置断点

```
(gdb) break kern/trap/trapentry.S:133
Breakpoint 2 at 0xffffffffc0200f0a: file kern/trap/trapentry.S, line 133.
```

continue继续执行，此时CPU停在执行`sret`指令的前一刻：

```
(gdb) c
Continuing.

Breakpoint 2, __trapret () at kern/trap/trapentry.S:133
133         sret
(gdb) x/8i $pc
=> 0xffffffffc0200f0a <__trapret+86>:   sret
   0xffffffffc0200f0e <forkrets>:       mv      sp,a0
   0xffffffffc0200f10 <forkrets+2>:     j       0xffffffffc0200eb4 <__trapret>
```

拦截sret执行过程

在终端2中拦截QEMU模拟sret指令的辅助函数helper_sret：

```
(gdb) break helper_sret
Breakpoint 1 at 0x62fb2a1f3256: file /home/yang/
_helper.c, line 76.
(gdb) c
Continuing.
```

在终端3进行一次单步执行，触发终端2的断点。

观测特权级切换

在终端2中观测CPU特权级结构体env->priv的变化：

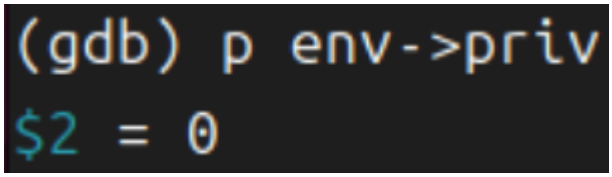
```
[Switching to Thread 0x7909d89ff6c0 (LWP 3093)]

Thread 2 "qemu-system-ris" hit Breakpoint 1, helper_sret (env=0x62fb3b7ad040,
    cpu_pc_deb=18446744072637910794)
    at /home/yang/riscv/qemu-4.1.1/target/riscv/op_helper.c:76
76          if (!(env->priv >= PRV_S)) {
(gdb) p env->priv
$1 = 1
```

执行前，priv=1CPU处于Supervisor Mode：

输入n单步执行C代码，直到执行完riscv_cpu_set_mode(env, prev_priv);：

```
(gdb) n
80          target_ulong retpc = env->sepc;
(gdb)
81          if (!riscv_has_ext(env, RVC) && (retpc & 0x3)) {
(gdb)
85          if (env->priv_ver >= PRIV_VERSION_1_10_0 &&
(gdb)
86              get_field(env->mstatus, MSTATUS_TSR)) {
(gdb)
85          if (env->priv_ver >= PRIV_VERSION_1_10_0 &&
(gdb)
90          target_ulong mstatus = env->mstatus;
(gdb)
91          target_ulong prev_priv = get_field(mstatus, MSTATUS_SPP);
(gdb)
92          mstatus = set_field(mstatus,
(gdb)
96          mstatus = set_field(mstatus, MSTATUS_SPIE, 0);
(gdb)
97          mstatus = set_field(mstatus, MSTATUS_SPP, PRV_U);
(gdb)
98          riscv_cpu_set_mode(env, prev_priv);
```

证实sret指令正确读取了mstatus中的SPP位（此时为User），并将CPU当前特权级从1降为0，完成了从内核到用户的切换。



QEMU源码分析

helper_sret函数完整模拟了RISC-V硬件执行sret指令的逻辑：

```
target_ulong helper_sret(CPURISCVState *env, target_ulong cpu_pc_deb) {
    // 1. 权限与合法性检查
    // sret指令只能在Supervisor(内核态)或更高权限下执行
    if (!(env->priv >= PRV_S)) {
        riscv_raise_exception(env, RISCV_EXCP_ILLEGAL_INST, GETPC());
    }

    // 2. 获取返回地址
    target_ulong retpc = env->sepc;  // 通常存放ecall的下一条指令地址
```

```
    // 3. 准备恢复上下文
    target_ulong mstatus = env->mstatus;
    target_ulong prev_priv = get_field(mstatus, MSTATUS_SPP); // 读取进入内核前的特
权级

    // 4. 恢复中断使能状态
    mstatus = set_field(mstatus,
        env->priv_ver >= PRIV_VERSION_1_10_0 ?
        MSTATUS_SIE : MSTATUS_UIE << prev_priv,
        get_field(mstatus, MSTATUS_SPIE));

    // 5. 重置mstatus状态位
    mstatus = set_field(mstatus, MSTATUS_SPIE, 0);
    mstatus = set_field(mstatus, MSTATUS_SPP, PRV_U);

    // 6. 执行特权级切换(关键)
    riscv_cpu_set_mode(env, prev_priv);  // 切换特权级

    // 7. 写回更新后的mstatus
    env->mstatus = mstatus;

    // 8. 跳转返回
    return retpc;
}
```

这段代码实现了sret指令的辅助函数，用于从内核态异常或中断处理中返回用户态。代码检查当前特权级是否允许执行该指令（需在S特权级及以上）；从sepc寄存器获取返回地址；通过操作mstatus寄存器恢复之前的特权级和中断使能状态（包括恢复SPIE字段、清除SPP字段等）；最后执行特权级切换并返回目标地址，从而完成从内核态到用户态或之前特权级的上下文恢复与切换流程。

TCG说明及与双重gdb实验的关联

TCG机制简介

TCG（Tiny Code Generator）是QEMU的动态二进制翻译子系统。其工作原理如下：

1. 将客户机（guest）指令块转换成一组中间操作（TCG ops）
2. 由后端生成宿主（host）机器码并缓存成翻译块（translation block，TB）
3. 执行时若遇到已生成的TB则直接执行宿主机器码，否则触发翻译流程生成并缓存新的TB

TCG的源码主要分布在：

- `accel/tcg`目录：核心翻译引擎
- `target/<arch>`目录：目标架构的指令解码与翻译入口
- `accel/tcg/*`文件：执行循环和TB管理实现

与双重gdb调试的关联

在我们的实验中，TCG机制与双重gdb调试方法密切相关：

1. **特权指令翻译**：`ecall`和`sret`这类特权指令在TCG翻译时，通常被生成为对运行时代码（helpers）的调用，或直接生成等价的宿主代码。

2. **调试策略有效性**：由于helper函数（如`riscv_raise_exception`、`riscv_cpu_do_interrupt`、`helper_sret`）会被明确调用，因此在QEMU进程中为这些helper函数设置断点是可靠的调试策略。

3. **TB缓存的影响**：TCG可能将多条客户机指令合并为一个宿主TB。如果只在宿主进程层设断点而不考虑helper，单步客户机指令时可能观察不到期望的边界。幸运的是，处理特权转换和异常的逻辑一般由helper函数实现，因此在QEMU上拦截这些helper可以稳定地捕获特权切换事件，不受TB缓存的影响。

4. **调试局限性**：TCG有时会内联某些简单操作以减少调用开销，这会让某些细粒度事件难以直接通过宿主函数边界观察到。如需完全跟踪每一条客户机指令的逐条执行，可选择：

    - 禁用TCG（使用解释器模式）
    - 在QEMU源码中查找并在生成代码前的翻译入口处插入断点或日志

实验总结

本次实验通过双重GDB调试方法，深入分析了RISC-V架构下系统调用的完整流程：

1. **ecall过程**：用户态程序通过`ecall`指令触发异常，硬件自动完成现场保存、原因记录、PC跳转和特权级提升，将控制权移交内核。

2. **sret过程**：内核通过`sret`指令恢复先前特权级，安全返回用户态，同时恢复用户程序的中断响应能力。

3. **调试方法**：通过用户态、内核态和QEMU硬件模拟三层调试，验证了特权级切换的完整过程，加深了对操作系统内核与硬件交互机制的理解。

4. **TCG关联**：理解了QEMU的动态二进制翻译机制如何影响调试过程，并掌握了在TCG环境下进行有效调试的策略。

关键调试技巧：

- 使用`info address`命令获取符号地址，即使没有完整调试信息
- 直接使用地址设置硬件断点`b *<地址>`
- 在QEMU helper函数中设置断点捕获特权指令执行
- 使用`si`指令单步跟踪关键执行路径

调试中遇到的问题及解决方案：

1. **符号表问题**：使用`info address`命令获取地址，然后直接设置硬件断点
2. **内存访问错误**：确保在正确的时间点设置断点（如内核已加载后）
3. **调试信息缺失**：通过反汇编验证指令位置，使用绝对地址调试

本实验不仅验证了理论知识的正确性，还提供了在复杂虚拟化环境下进行系统级调试的实践经验，为后续操作系统内核开发与调试工作奠定了坚实基础。通过解决调试过程中的实际问题，我们加深了对GDB调试工具、QEMU虚拟化机制以及RISC-V特权架构的理解。