



gdb 调试系统调用以及返回

实验概述

本实验通过双重GDB调试方案，观察uCore操作系统中系统调用的完整流程：从用户态通过 `ecall` 指令进入内核态，经过系统调用处理，再通过 `sret` 指令返回用户态。实验的核心目标是理解特权级切换的硬件机制，以及QEMU如何通过软件模拟这一过程。

实验环境

- 操作系统：Ubuntu 24.04
- QEMU版本：4.1.1（带调试信息重新编译）
- GDB版本：
 - x86_64 GDB：15.0.50（调试QEMU）
 - riscv64-unknown-elf-gdb：10.1（调试uCore）

调试架构

```
终端1: make debug          → 启动QEMU等待连接
终端2: sudo gdb + attach   → 调试QEMU源码（观察硬件模拟）
终端3: make gdb            → 调试uCore内核（观察软件行为）
```

实验流程及解读

第一阶段：启动调试环境

1. 启动QEMU调试模式

在**终端1**中执行：

```
make debug
```

2. 附加到QEMU进程

在终端2中：

```
# 获取QEMU进程号  
pgrep -f qemu-system-riscv64  
# 输出: 33957  
  
sudo gdb
```

在GDB中执行：

```
(gdb) attach 33957  
(gdb) handle SIGPIPE nostop noprint  
(gdb) continue
```

```
(gdb) attach 33957
Attaching to process 33957
[New LWP 33959]
[New LWP 33958]
warning: could not find '.gnu_debugaltlink' file for /lib/x86_64-linux-gnu/libpulse.so.0
warning: could not find '.gnu_debugaltlink' file for /lib/x86_64-linux-gnu/libglib-2.0.so.0
warning: could not find '.gnu_debugaltlink' file for /usr/lib/x86_64-linux-gnu/pulseaudio/libpulsecommon-16.1.so
warning: could not find '.gnu_debugaltlink' file for /lib/x86_64-linux-gnu/libgbm.so.1
warning: could not find '.gnu_debugaltlink' file for /lib/x86_64-linux-gnu/libcap.so.2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
0x0000766aca31ba30 in __GI_ppoll (fds=0x5bc7949e9c30,
    nfds=7,
    timeout=<optimized out>, sigmask=0x0)
    at ../sysdeps/unix/sysv/linux/ppoll.c:42
warning: 42      ../sysdeps/unix/sysv/linux/ppoll.c: No such
file or directory
(gdb) handle SIGPIPE nostop noprint
Signal      Stop      PrintPass to program Description
SIGPIPE      No       No   Yes           Broken pipe
(gdb) c
Continuing.
```

- `attach 33957` : 将GDB附加到正在运行的QEMU进程
- `handle SIGPIPE nostop noprint` : 忽略SIGPIPE信号，避免不必要的中断
- `continue` : 让QEMU继续运行，这一步**至关重要**，否则终端3无法调试uCore

3. 连接uCore内核调试

在**终端3**中执行：

```
make gdb
```

在GDB中：

```
# 加载用户程序符号表
(gdb) add-symbol-file obj/__user_exit.out
Reading symbols from obj/__user_exit.out...
(y or n) y

# 在用户态syscall函数打断点
(gdb) b user/libs/syscall.c:19
Breakpoint 1 at 0x8000f8: file user/libs/syscall.c, line 19.

# 继续执行到断点
(gdb) c
```

llzf@LZF:~/OS/lab5\$ make debug

OpenSBI v0.4 (Jul 2 2019 11:53:53)



Platform Name : QEMU Virt Machine

Platform HART Features : RV64ACDFIMSU

Platform Max HARTs : 8

Current Hart : 0

Firmware Base : 0x80000000

Firmware Size : 112 KB

Runtime SBI Version : 0.1

PMP0: 0x000000080000000-0x00000008001ffff (A)

PMP1: 0x000000000000000-0xffffffffffff (A,R,W,X)

DTB Init

HartID: 0

DTB Address: 0x82200000

Physical Memory from DTB:

Base: 0x000000080000000

Size: 0x000000008000000 (128 MB)

End: 0x000000087fffff

DTB init completed

(THU.CST) os is loading ...

Special kernel symbols:

entry 0xc020004a (virtual)

etext 0xc020583c (virtual)

edata 0xc02a6240 (virtual)

end 0xc02aa6ec (virtual)

Kernel executable memory footprint: 682KB

```
(gdb) add-symbol-file obj/__user_exit.out
add symbol table from file "obj/__user_exit.out"
(y or n) y
Reading symbols from obj/__user_exit.out...
(gdb) b user/libs/syscall.c:19
Breakpoint 1 at 0x8000f8: file user/libs/syscall.c, line 19.
(gdb) c
Continuing.

Breakpoint 1, syscall (num=num@entry=30)
  at user/libs/syscall.c:19
19          asm volatile (
```

程序停在 `syscall` 函数处：

```
Breakpoint 1, syscall (num=num@entry=30)
  at user/libs/syscall.c:19
19          asm volatile (
```

第二阶段：调试ecall指令（用户态→内核态）

1. 定位ecall指令

在终端3中：

```
# 查看后续指令
(gdb) x/8i $pc
=> 0x8000f8 <syscall+32>:  ld      a0,8(sp)
    0x8000fa <syscall+34>:  ld      a1,40(sp)
    ...
    0x800104 <syscall+44>:  ecall           ← 目标指令
    0x800108 <syscall+48>:  sd      a0,28(sp)

# 持续显示当前指令
(gdb) display/i $pc

# 单步执行到ecall
(gdb) si
(gdb) si
... (重复直到)
=> 0x800104 <syscall+44>:  ecall

# 确认当前PC
(gdb) info registers pc
pc 0x800104
```

```
(gdb) x/8i $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)

(gdb) display/i $pc
1: x/i $pc
=> 0x8000f8 <syscall+32>:    ld      a0,8(sp)
(gdb) si
0x00000000008000fa      19      asm volatile (
1: x/i $pc
=> 0x8000fa <syscall+34>:    ld      a1,40(sp)
(gdb)
0x00000000008000fc      19      asm volatile (
1: x/i $pc
=> 0x8000fc <syscall+36>:    ld      a2,48(sp)
(gdb)
0x00000000008000fe      19      asm volatile (
1: x/i $pc
=> 0x8000fe <syscall+38>:    ld      a3,56(sp)
(gdb)
0x0000000000800100      19      asm volatile (
1: x/i $pc
=> 0x800100 <syscall+40>:    ld      a4,64(sp)
(gdb)
0x0000000000800102      19      asm volatile (
1: x/i $pc
=> 0x800102 <syscall+42>:    ld      a5,72(sp)
(gdb)
0x0000000000800104      19      asm volatile (
1: x/i $pc
=> 0x800104 <syscall+44>:    ecall
(gdb) i r pc
pc          0x800104 0x800104 <syscall+44>
```

此时，CPU即将执行 `ecall` 指令，这将触发一个异常，从用户态陷入内核态。

2. 在QEMU中设置断点

与大模型交互记录：

提问："QEMU源码中处理RISC-V `ecall`指令的关键函数是什么？"

大模型回答："`ecall`指令会触发异常，QEMU通过 `riscv_cpu_do_interrupt` 函数处理所有异常和中断。该函数位于 `target/riscv/cpu_helper.c`。"

切换到**终端2**（应显示 `Continuing`）：

```
# 按 Ctrl+C 中断QEMU
^C
Thread 1 "qemu-system-ris" received signal SIGINT, Interrupt.

# 设置断点
(gdb) break riscv_cpu_do_interrupt
Breakpoint 1 at 0x5bc78c1bc857: file /home/lizf/qemu-4.1.1/target/riscv/cpu_helper.c, line 507.

# 继续执行
(gdb) continue
```

```
(gdb) break riscv_cpu_do_interrupt
Breakpoint 1 at 0x5bc78c1bc857: file /home/lizf/qemu-4.1.1/t
arget/riscv/cpu_helper.c, line 507.
(gdb) c
Continuing.
```

3. 执行`ecall`并观察异常处理

在**终端3**执行：

```
(gdb) si
```

此时**终端2**会自动停在断点处：

```
Thread 2 "qemu-system-ris" hit Breakpoint 1, riscv_cpu_do_interrupt (
  cs=0x5bc79499d790)
  at /home/l1zf/qemu-4.1.1/target/riscv/cpu_helper.c:507
507      RISCVCPU *cpu = RISCV_CPU(cs);
```

```
[Switching to Thread 0x766ac8e606c0 (LWP 33959)]
```

```
Thread 2 "qemu-system-ris" hit Breakpoint 1, riscv_cpu_do_interrupt (
    cs=0x5bc79499d790)
  at /home/llzf/qemu-4.1.1/target/riscv/cpu_helper.c:507
507      RISCVCPU *cpu = RISCV_CPU(cs);
(gdb) list
502      */
503      void riscv_cpu_do_interrupt(CPUState *cs)
504      {
505 #if !defined(CONFIG_USER_ONLY)
506
507      RISCVCPU *cpu = RISCV_CPU(cs);
508      CPURISCVState *env = &cpu->env;
509
510      /* cs->exception is 32-bits wide unlike mcause which is XLEN-bits wide
511      * so we mask off the MSB and separate into trap type and cause.
(gdb) backtrace
#0  riscv_cpu_do_interrupt (
    cs=0x5bc79499d790)
  at /home/llzf/qemu-4.1.1/target/riscv/cpu_helper.c:507
#1  0x00005bc78c12802d in cpu_handle_exception (
    cpu=0x5bc79499d790,
    ret=0x766ac8e5f8bc)
  at /home/llzf/qemu-4.1.1/accel/tcg/cpu-exec.c:506
#2  0x00005bc78c1286b9 in cpu_exec (cpu=0x5bc79499d790)
  at /home/llzf/qemu-4.1.1/accel/tcg/cpu-exec.c:712
#3  0x00005bc78c0da8af in tcg_cpu_exec (
    cpu=0x5bc79499d790)
  at /home/llzf/qemu-4.1.1/cpus.c:1435
#4  0x00005bc78c0db168 in qemu_tcg_cpu_thread_fn (
    arg=0x5bc79499d790)
  at /home/llzf/qemu-4.1.1/cpus.c:1743
#5  0x00005bc78c56075e in qemu_thread_start (
    args=0x5bc7949b3e20)
  at util/qemu-thread-posix.c:502
#6  0x00005bc78c56075e in qemu_thread_start (
    args=0x5bc7949b3e20)
  at util/qemu-thread-posix.c:502
```

查看调用栈：

```
(gdb) backtrace
#0  riscv_cpu_do_interrupt (cs=0x5bc79499d790)
#1  cpu_handle_exception (cpu=0x5bc79499d790, ret=0x766ac8e5f8bc)
#2  cpu_exec (cpu=0x5bc79499d790)
#3  tcg_cpu_exec (cpu=0x5bc79499d790)
#4  qemu_tcg_cpu_thread_fn (arg=0x5bc79499d790)
...
...
```

调用栈显示：ecall → TCG异常捕获 → CPU异常处理器 → 中断处理函数。

4. 观察关键寄存器的设置

与大模型交互记录：

提问："如何观察ecall处理过程中CSR寄存器的变化？"

大模型回答："使用GDB的 watch 功能可以监控变量变化。当 env->sepc 被写入时，watch会自动中断程序，显示旧值和新值。"

查看异常处理的关键信息：

```
(gdb) p/x env->scause
$4 = 0x8          ← 异常原因: User ECALL

(gdb) p/x env->sepc
$5 = 0x800104    ← 保存的返回地址

(gdb) p/x env->pc
$6 = 0x800104    ← 当前PC (即将跳转)

(gdb) p env->priv
$7 = 0           ← 当前特权级: User mode
```

监控PC跳转到内核：

```
(gdb) watch env->pc
Hardware watchpoint 3: env->pc

(gdb) c
Continuing.

Thread 2 hit Hardware watchpoint 3: env->pc
Old value = 8388868          (0x800104)
New value = 18446744072637910736 (0xfffffffffc0200ed0 - __alltraps!)
```

PC从用户态地址 0x800104 跳转到内核虚拟地址 0xfffffffffc0200ed0 (__alltraps 入口) 。

验证特权级提升：

```
(gdb) watch env->pc
Hardware watchpoint 2: env->pc
(gdb) c
Continuing.

Thread 2 "qemu-system-ris" hit Hardware watchpoint 2: env->pc

Old value = <unreadable>
New value = 8388868
riscv_cpu_do_interrupt (cs=0x5e23397a2790)
    at /home/llzf/qemu-4.1.1/target/riscv/cpu_helper.c:513
513          bool async = !(cs->exception_index & RISCV_EXCP_
INT_FLAG);
(gdb) c
Continuing.

Thread 2 "qemu-system-ris" hit Hardware watchpoint 2: env->pc

Old value = 8388868
New value = 18446744072637910736
riscv_cpu_do_interrupt (cs=0x5e23397a2790)
    at /home/llzf/qemu-4.1.1/target/riscv/cpu_helper.c:566
566          riscv_cpu_set_mode(env, PRV_S);
(gdb) p env->priv
$1 = 0
(gdb) n
553          cause < TARGET_LONG_BITS && ((deleg >> ca
use) & 1)) {
(gdb) p env->priv
$2 = 1
(gdb) p/x env->scause
$3 = 0x8
(gdb) p/x env->sepc
$4 = 0x800104
```

```
(gdb) n
(gdb) p env->priv
$9 = 1           ← 已提升到 Supervisor mode
```

第三阶段：调试sret指令（内核态→用户态）

1. 定位sret指令

在**终端2**清理断点并继续：

```
(gdb) delete breakpoints
(gdb) continue
```

在**终端3**中：

```
# 在sret所在行打断点
(gdb) b kern/trap/trapentry.S:133
Breakpoint 3 at 0xfffffffffc0200f96: file kern/trap/trapentry.S, line 133.

(gdb) c
Continuing.

Breakpoint 3, __trapret () at kern/trap/trapentry.S:133
133      sret
1: x/i $pc
=> 0xfffffffffc0200f96 <__trapret+86>: sret

# 查看返回地址
(gdb) info registers sepc
sepc    0x800108    8388872
```

```
(gdb) b kern/trap/trap<rn/trap/trapentry .S:133
Breakpoint 3 at 0xfffffffffc0200f96: file kern/trap/trapentry
S, line 133.
(gdb) c
Continuing.

Breakpoint 3, __trapret ()
  at kern/trap/trapentry.S:133
133          sret
1: x/i $pc
=> 0xfffffffffc0200f96 <__trapret+86>: sret
(gdb) i r sepc
sepc          0x800108          8388872
```

sepc = 0x800108 = 0x800104 + 4 , 正好是ecall的下一条指令。

2. 在QEMU中设置sret断点

与大模型交互记录：

提问："QEMU源码中如何处理sret指令？"

大模型回答："sret指令由 helper_sret 函数处理，位于 target/riscv/op_helper.c 。该函数负责从 sepc 恢复PC，从 sstatus.SPP 恢复特权级。"

切换到终端2：

```
# Ctrl+C 中断
^C

(gdb) break helper_sret
Breakpoint 1 at 0x5bc78c1bae26: file /home/lrzf/qemu-4.1.1/target/riscv/op_helper.c, line 76.

(gdb) continue
```

3. 执行sret并观察返回流程

在终端3执行：

```
(gdb) si
```

```
(gdb) si
0x00000000000800108 in syscall (num=0, num@entry=30)
  at user/libs/syscall.c:19
19          asm volatile (
1: x/i $pc
=> 0x800108 <syscall+48>:      sd      a0,28(sp)
(gdb) i r pc
pc          0x800108 0x800108 <syscall+48>
```

终端2停在断点：

```
Thread 2 hit Breakpoint 1, helper_sret (
  env=0x5bc7949a61a0,
  cpu_pc_deb=18446744072637910934)
  at /home/l1zf/qemu-4.1.1/target/riscv/op_helper.c:76
76      if (!(env->priv >= PRV_S)) {
```

查看源码关键部分：

```
(gdb) list 80,110
80  target_ulong retpc = env->sepc;           // ① 读取返回地址
...
90  target_ulong mstatus = env->mstatus;
91  target_ulong prev_priv = get_field(mstatus, MSTATUS_SPP); // ② 获取之前特权级
92-95 // 恢复中断使能 (SPIE → SIE)
96  mstatus = set_field(mstatus, MSTATUS_SPIE, 0);
97  mstatus = set_field(mstatus, MSTATUS_SPP, PRV_U);
98  riscv_cpu_set_mode(env, prev_priv);        // ③ 设置特权级
99  env->mstatus = mstatus;
101 return retpc;                         // ④ 返回PC值
```

查看关键寄存器：

```
(gdb) p env->priv
$1 = 1
(gdb) p/x env->pc
$2 = 0xfffffffffc0200f96
(gdb) p/x env->sepc
$3 = 0x800108
(gdb) n
81          if (!riscv_has_ex-
t(env, RVC) && (retpc & 0x3))
{
(gdb) p/x retpc
$4 = 0x800108
(gdb) p/x env->mstatus
$5 = 0x80000000000046020
(gdb) p (env->mstatus >> 8) &
1
$6 = 0
(gdb) p (env->mstatus >> 5) &
1
```

```
(gdb) p/x env->sepc
$3 = 0x800108      ← 返回地址

(gdb) p env->priv
$1 = 1      ← 当前S-mode

(gdb) p/x env->pc
$2 = 0xfffffffffc0200f96 ← 当前在内核

# 查看mstatus的SPP和SPIE位
(gdb) p (env->mstatus >> 8) & 1
$6 = 0      ← SPP=0 (之前是U-mode)

(gdb) p (env->mstatus >> 5) & 1
$7 = 1      ← SPIE=1 (之前中断使能)
```

监控特权级降低：

```
(gdb) watch env->priv
Hardware watchpoint 3: env->priv

(gdb) c
Continuing.

Thread 2 hit Hardware watchpoint 3: env->priv
Old value = 1          ← S-mode
New value = 0          ← U-mode (成功降级! )
```

```
(gdb) p env->priv
$8 = 0
(gdb) p/x env->pc
$9 = 0xfffffffffc02000f96
```

完成返回：

```
(gdb) delete breakpoints
(gdb) continue
```

在终端3验证：

```
(gdb) i r pc
pc    0x800108          ← 已返回用户态
```

代码分析

ecall 异常处理：riscv_cpu_do_interrupt

位置： qemu-4.1.1/target/riscv/cpu_helper.c

```

void riscv_cpu_do_interrupt(CPUState *cs)
{
    RISCVCPU *cpu = RISCV_CPU(cs);
    CPURISCVState *env = &cpu->env;

    bool async = !(cs->exception_index & RISCV_EXCP_INT_FLAG);
    target_ulong cause = cs->exception_index & RISCV_EXCP_INT_MASK;
    target_ulong deleg = async ? env->mideleg : env->medeleg;

    // ecall根据特权级映射异常码
    if (cause == RISCV_EXCP_U_ECALL) {
        assert(env->priv <= 3);
        cause = ecall_cause_map[env->priv]; // U-mode ecall → 8
    }

    // 判断是否委托给S-mode处理
    if (env->priv <= PRV_S &&
        cause < TARGET_LONG_BITS && ((deleg >> cause) & 1)) {
        // 在S-mode处理异常
        target_ulong s = env->mstatus;

        // 保存中断使能状态到SPIE
        s = set_field(s, MSTATUS_SPIE,
                      get_field(s, MSTATUS_SIE));

        // 保存之前的特权级到SPP
        s = set_field(s, MSTATUS_SPP, env->priv);

        // 关闭S-mode中断
        s = set_field(s, MSTATUS_SIE, 0);

        env->mstatus = s; // 设置异常原因
        env->scause = cause; // 保存返回地址
        env->sepc = env->pc; // 保存错误地址
        env->sbadaddr = tval;

        // 跳转到中断向量
        env->pc = (env->stvec >> 2 << 2) +
            ((async && (env->stvec & 3) == 1) ? cause * 4 : 0);
    }
}

```

```

    // 提升到S-mode
    riscv_cpu_set_mode(env, PRV_S);
}
}

```

关键操作对照表：

操作	代码位置	寄存器变化
保存返回地址	562行	sepc \leftarrow pc (0x800104)
设置异常原因	561行	scause \leftarrow 8 (User ECALL)
保存旧特权级	558行	sstatus.SPP \leftarrow 0 (U-mode)
关闭中断	559行	sstatus.SIE \leftarrow 0
跳转内核	564-565行	pc \leftarrow stvec (0xfffffffffc0200ed0)
提升特权级	566行	priv \leftarrow 1 (S-mode)

sret 返回处理：helper_sret

位置： qemu-4.1.1/target/riscv/op_helper.c

```
target_ulong helper_sret(CPURISCVState *env, target_ulong cpu_pc_deb)
{
    // 检查权限（必须在S-mode或M-mode）
    if (!(env->priv >= PRV_S)) {
        riscv_raise_exception(env, RISCV_EXCP_ILLEGAL_INST, GETPC());
    }

    // 读取返回地址
    target_ulong retpc = env->sepc;

    // 检查返回地址对齐
    if (!riscv_has_ext(env, RVC) && (retpc & 0x3)) {
        riscv_raise_exception(env, RISCV_EXCP_INST_ADDR_MIS, GETPC());
    }

    target_ulong mstatus = env->mstatus;

    // 从SPP恢复之前的特权级
    target_ulong prev_priv = get_field(mstatus, MSTATUS_SPP);

    // 恢复中断使能: SPIE → SIE
    mstatus = set_field(mstatus,
                        env->priv_ver >= PRIV_VERSION_1_10_0 ?
                        MSTATUS_SIE : MSTATUS_UIE << prev_priv,
                        get_field(mstatus, MSTATUS_SPIE));

    // 清除SPIE
    mstatus = set_field(mstatus, MSTATUS_SPIE, 0);

    // 清除SPP（设为U-mode）
    mstatus = set_field(mstatus, MSTATUS_SPP, PRV_U);

    // 设置特权级
    riscv_cpu_set_mode(env, prev_priv);

    // 更新mstatus
    env->mstatus = mstatus;

    // 返回PC值（TCG会将其设置给env->pc）
}
```

```
    return retpc;  
}
```

关键操作对照表：

操作	代码位置	寄存器变化
读取返回地址	80行	retpc \leftarrow sepc (0x800108)
获取旧特权级	91行	prev_priv \leftarrow SPP (0)
恢复中断	92-95行	SIE \leftarrow SPIE (1)
清除SPIE	96行	SPIE \leftarrow 1
清除SPP	97行	SPP \leftarrow 0 (U-mode)
降低特权级	98行	priv \leftarrow 0 (U-mode)
返回PC	101行	pc \leftarrow retpc (0x800108)

实验总结

完整的系统调用流程

用户态程序 (U-mode, PC=0x800104)

↓ 执行 ecall 指令

```
| QEMU硬件模拟 (riscv_cpu_do_interrupt)
| 1. scause ← 8 (User ECALL)
| 2. sepc ← 0x800104 (保存返回地址)
| 3. sstatus.SPP ← 0 (保存U-mode)
| 4. sstatus.SIE ← 0 (关闭中断)
| 5. pc ← 0xfffffffffc0200ed0 (__alltraps)
| 6. priv ← 1 (提升到S-mode)
```

↓

内核态处理 (S-mode, __alltraps → trap → syscall)

↓ 系统调用处理完成

↓ 准备返回 (__trapret)

```
| QEMU硬件模拟 (helper_sret)
| 1. retpc ← sepc (0x800108)
| 2. prev_priv ← sstatus.SPP (0)
| 3. SIE ← SPIE (恢复中断)
| 4. priv ← 0 (降低到U-mode)
| 5. pc ← retpc (0x800108)
```

↓

用户态程序继续执行 (U-mode, PC=0x800108)

关键发现

1. 特权级切换机制：

- ecall: User mode (0) → Supervisor mode (1)
- sret: Supervisor mode (1) → User mode (0)
- 通过 sstatus.SPP 保存和恢复特权级

2. PC跳转机制：

- ecall: 用户地址 → 内核地址 (stvec指向 __alltraps)
- sret: 内核地址 → 用户地址 (sepc指向ecall下一条指令)

3. 中断状态管理:

- 进入内核时关闭中断 (SIE ← 0)
- 返回用户态时恢复中断 (SIE ← SPIE)

4. QEMU模拟精度:

- QEMU精确模拟了RISC-V特权架构规范
- 每个CSR寄存器的每一位都严格按照规范设置
- TCG动态翻译确保了执行的正确性

与大模型交互总结

本次实验充分利用了大模型辅助学习，主要交互包括：

1. 定位关键函数:

- 提问： "QEMU源码中处理ecall的函数是什么？"
- 回答： `riscv_cpu_do_interrupt`

2. 理解调试工具:

- 提问： "如何观察CSR寄存器的变化？"
- 回答： 使用 `watch` 命令监控变量

3. 源码理解:

- 提问： "helper_sret的返回值如何设置给PC？"
- 回答： TCG生成的代码会将返回值写入 `env->pc`

4. 问题排查:

- 问题： "为什么PC从内核地址变成了用户地址？"
- 分析： 这正是sret的核心功能——恢复用户态PC

通过大模型的辅助，显著提高了调试效率，加深了对系统调用机制的理解。

附录：调试技巧总结

双重GDB协调原则

场景	终端2状态	终端3操作
正常调试uCore	Continuing	自由使用si/n/c
观察QEMU内部	Ctrl+c 中断	暂停操作
设置QEMU断点	设置后立即 continue	继续调试

常用GDB命令

```
# 查看调用栈  
backtrace (bt)  
  
# 查看源码  
list [行号]  
list [函数名]  
  
# 监控变量变化  
watch [变量]  
  
# 查看寄存器  
info registers [寄存器名]  
p/x [变量]  
  
# 反汇编  
disassemble [函数]  
x/Ni $pc # 查看N条指令
```

注意事项

1. **终端2必须continue**: 设置断点后必须立即continue, 否则终端3无法操作
2. **符号表加载**: 必须加载 obj/__user_exit.out 才能调试用户程序
3. **watch作用域**: 离开函数后watch会自动删除

4. **特权级含义**: 0=U-mode, 1=S-mode, 3=M-mode