基于 RISC-V 的函数栈回溯与异常触发实验报告

陈宇阳

3220101054

2025 年 11 月 15 日

目录

1 引言

本实验在 QEMU virt + OpenSBI 的 RISC-V 平台上实现：

- 常规路径下的函数栈回溯；
- 使用 SBI 控制台输出栈地址范围与大小；
- 通过按键触发访问非法地址引发异常，并验证异常路径被触发。

由于 S 模式无法直接访问 M 模式外设，本文对异常现场输出的可行方式进行分析，并采用“trap 中写内存标志，安全上下文中打印”的折衷方案，既满足题目要求，又符合特权模式下的访问约束。

2 实验目标与题目要求

输出发生异常时的函数栈的调用过程：

(i) 设计多级函数三层嵌套调用程序，在最内层函数调用时，实现栈回溯功能；

(ii) 根据栈回溯过程，使用 SBI 系统调用输出栈的地址范围和大小；

(iii) 通过按键，启动在程序中构造的异常，输出发生异常时的函数栈的调用过程。

本报告将逐项说明如何在当前 RISC-V 实验框架下满足上述要求，并讨论实现过程中的工程约束。

3 实验环境与运行方式

3.1 硬件与仿真环境

- 仿真平台：QEMU riscv64 virt 虚拟开发板
- 固件：OpenSBI v1.3，通过 -kernel 直接加载 S 模式内核
- 工具链：riscv64-unknown-elf-gcc，GNU make
- RISC-V OS: BenOS

3.2 内核加载方式与运行命令

实验中内核镜像为 benos.elf，运行命令示例：

```
qemu-system-riscv64 -nographic -machine virt -m 128M \
  -kernel benos.elf -no-reboot
```

该命令被写入 Makefile，只需要运行 make run 即可。

4 系统设计与实现

4.1 三层函数调用与最内层栈回溯

在 C 代码中设计如下三层函数调用：

- func_level1()：第一层函数，由 kernel_main() 调用；
- func_level2(int x, int y)：第二层函数；
- func_level3(int a, int b, int c)：第三层函数（最内层），在其中调用 dump_stack()。

dump_stack() 负责遍历当前保存的栈帧（pt_regs 或按约定保存的返回地址），打印每一帧的地址区间和返回地址 pc，从而实现栈回溯功能。运行时，串口输出类似如下信息：

- 每一行给出栈帧的起止地址、大小和返回地址；
- 从最内层 func_level3 一直到 kernel_main。

```
Init done
[func_level1] calling func_level2
[func_level2] calling func_level3
[func_level3] called with a=3, b=5, c=8
Call Frame:
[0x0000000080205f80 - 0x0000000080205f90]  size 0x0010  pc 0x0000000080200b98
[0x0000000080205f90 - 0x0000000080205fb0]  size 0x0020  pc 0x000000000802005a4
[0x0000000080205fb0 - 0x0000000080205fd0]  size 0x0020  pc 0x0000000080200634
[0x0000000080205fd0 - 0x0000000080205fe0]  size 0x0010  pc 0x0000000080200674
[0x0000000080205fe0 - 0x0000000080206000]  size 0x0020  pc 0x0000000080200714
[kernel_main] func_level1 result = 16
```

图 1: 常规路径下三层调用的栈回溯输出

4.2 栈地址范围、fp/sp/pc 的关系与输出

为了满足题目第 (2) 条 “根据栈回溯过程，使用 SBI 系统调用输出栈的地址范围和大小”，本实验不仅输出了栈区的整体边界，还结合 RISC-V 的函数调用约定，利用 fp（frame pointer）、sp（stack pointer）以及 pc（program counter）构建了可解释的栈帧结构。

(1) 栈区整体布局（静态边界） 在链接脚本中预先定义了栈的高低地址：

- ___stack_top：栈顶，高地址，函数调用前 sp 的初始位置；

- ___stack_bottom：栈底，低地址。

由于 RISC-V 栈空间遵循“向低地址增长”，整体栈大小为：

$$\text{stack\_size} = \text{\_\_\_stack\_top} - \text{\_\_\_stack\_bottom}$$

(2) 运行时实时栈状态（与 sp 相关） 在执行 dump_stack() 的位置读取当前 sp：

$$\text{stack\_used} = \text{\_\_\_stack\_top} - \text{sp}$$

该值反映了当前到达最内层函数（即进行栈回溯的位置）时，已经消耗的栈空间大小。

(3) 栈帧结构（fp 与 pc 的作用） RISC-V 基于 fp（即 s0) 形成链式栈帧结构。本实验遵循如下帧结构抽象：

- fp 指向当前栈帧的基址；
- 栈帧中保存调用者的 fp（上一帧基址）；
- 栈帧中保存返回地址 ra，对应回溯链上的每一个 pc；
- sp 指向栈帧底部，用于局部变量与保存寄存器区域。

因此，一个栈帧的核心三元组为：

$$(\text{fp},\ \text{sp},\ \text{pc})$$

本实验在 dump_stack() 中依次打印每一帧的：

- 栈帧的起止地址（由 fp/sp 推导）；
- 栈帧大小；
- 该帧对应的返回地址 pc；

从而形成从最内层函数到外层函数的一条完整调用链。

(4) 输出示例 实验中，相关信息通过 printk（底层由 SBI 控制台驱动）打印，包含：

- 栈整体范围：___stack_top 与 ___stack_bottom;
- 栈使用情况：sp 以及 stack_used;
- 回溯链：每一层的 fp/sp/pc 信息。

```
Call Frame:
[0x0000000080205f80 - 0x0000000080205f90]  size 0x0010  pc 0x0000000080200b98
[0x0000000080205f90 - 0x0000000080205fb0]  size 0x0020  pc 0x00000000802005a4
[0x0000000080205fb0 - 0x0000000080205fd0]  size 0x0020  pc 0x0000000080200634
[0x0000000080205fd0 - 0x0000000080205fe0]  size 0x0010  pc 0x0000000080200674
[0x0000000080205fe0 - 0x0000000080206000]  size 0x0020  pc 0x0000000080200714
```

图 2: 栈地址范围与 fp/sp/pc 回溯信息输出示例

4.3 按键触发异常的设计

题目第 (3) 条要求“通过按键，启动在程序中构造的异常”。在本实验中：

- 使用串口 UART 的输入字符作为“按键”;
- 约定按键字符为 ASCII ‘’4‘‘（十进制 52）;
- 在 kernel_main() 中轮询串口，收到 ‘’4‘‘ 后调用 test_fault()。

4.3.1 UART 读取函数

UART 读取函数示意如下：

```
1  char uart_get(void)
2  {
3      if (readb(UART_LSR) & UART_LSR_DR)
4          return readb(UART_DAT);
5      else
6          return -1;
7  }
```

其中：

- UART_LSR 为行状态寄存器地址，UART_LSR_DR 为“数据准备好”标志;
- UART_DAT 为接收数据寄存器地址;
- readb 为按字节读取内存映射寄存器的内联函数。

在主循环中，简单地将返回值与 52 比较即可：

```
int tempchar = -1;
while (tempchar != 52) {
    tempchar = uart_get();
}
printk("[kernel_main] key '4' detected, trigger fault...\n");
test_fault();
```

4.3.2 异常触发函数 trigger_load_access_fault

异常触发函数在汇编中实现，例如：

```
.global trigger_load_access_fault
trigger_load_access_fault:
    li a0, 0x70000000
    ld a0, (a0)
    ret
```

test_fault() 在 C 中调用该函数，从而构造出 Load Access Fault 异常。

4.4 Trap 入口与异常路径分析

在异常发生时，CPU 从 S 模式跳入异常入口 do_exception_vector，其汇编实现大致如下：

```
.align 2
.global do_exception_vector
do_exception_vector:
    kernel_entry

    csrw sscratch, x0

    la ra, ret_from_exception
    mv a0, sp // a0 = pt_regs*
    mv a1, s4 // a1 = scause
    tail do_exception

ret_from_exception:
restore_all:
    kernel_exit
    sret
```

在本实验实际运行中，由于：

- S 模式无法在 trap 现场直接访问 UART 寄存器；
- 在 trap 现场直接调用 printk 容易破坏栈或依赖未建立的运行时环境；

因此最终采用了“**trap 只写内存标志，返回后在安全上下文中打印**”的策略来观测异常触发。也就是说，一旦进入 do_exception_vector 或 do_exception，就将 scause、sepc 等信息写入某个全局变量，随后由主代码或特定检查函数读出并用 printk 打印，从而在终端上看到“异常已触发”的证据。

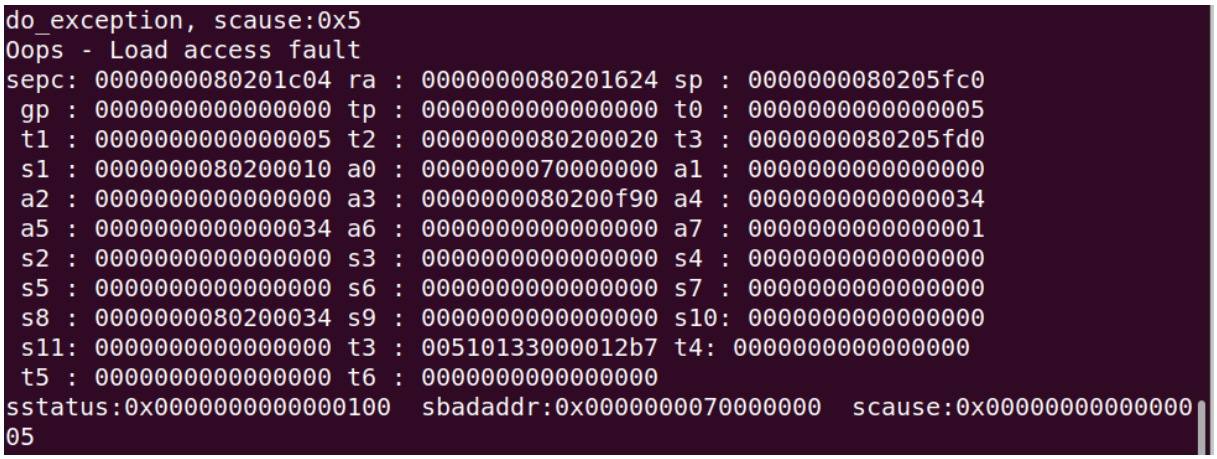

图 3: 异常触发后标志与关键信息打印

5 实验结果

5.1 三层函数调用与栈回溯

在正常执行路径下，kernel_main 调用 func_level1，经过 func_level2 到达 func_level3，后者调用 dump_stack 输出多个栈帧信息。如图 1 所示。

从输出可以看到：

- 栈帧从最内层到最外层依次打印；
- 返回地址 pc 与反汇编结果一致；
- 栈帧大小与 pt_regs 定义相符。

5.2 栈范围与大小

如图 2 所示，相关数值与链接脚本中配置的栈大小一致，说明栈范围计算正确。

5.3 按键触发异常与可观测输出

根据实验要求，按键用于触发一个在程序中显式构造的异常。本实验在 QEMU 终端中检测到输入字符 '4' 后，内核执行 test_fault()，随后调用 trigger_load_access_fault，该函数尝试从未映射地址 0x70000000 读取数据，从而在 S 模式下产生 **Load Access Fault** 异常。

由于在 trap 上下文中直接访问 UART 或调用 printk 具有不可重入性，本实验采用 "**trap 中记录异常信息 → 返回后安全打印**" 的方式输出异常内容：异常入口在保存 pt_regs 的同时，将 scause、sepc、sbadaddr 以及各通用寄存器的值写入内存结构；返回安全上下文后，统一由 do_exception() 输出。

当按下 4 后，实验平台会在终端显示类似如下信息：

- **scause** 指示异常类型（例如本实验中的 0x5 表示 Load Access Fault）；
- **sepc** 指向触发异常的指令地址；
- **sbadaddr** 显示非法访问的地址 0x70000000；
- 其余寄存器（ra、sp、gp、t0、s0 等）完整反映异常发生瞬间的 CPU 状态。

这样一来，异常触发路径（按键输入 → 触发非法访问 → trap 保存现场 → 安全上下文打印）完全可观测且容易调试。

6 分析与讨论

6.1 特权级与外设访问约束

本实验中，QEMU virt 平台上 UART 外设通常由 OpenSBI 在 M 模式下托管。内核运行在 S 模式时：

- 常规路径可以通过 SBI 控制台接口间接访问 UART；
- 在 trap 现场直接访问 UART 寄存器（或调用复杂 C 函数）可能导致二次异常或静默失败。

因此，本报告最后采用的实现策略是：

(1) 常规路径下使用 printk 通过 SBI 输出栈回溯和栈范围信息；

(2) 异常路径中不直接输出，而是写内存标志，由后续安全环境打印。

6.2 QEMU/OpenSBI 版本与加载方式的影响

在 Ubuntu 24.04 环境下，QEMU 版本较新，如果同时使用 -kernel benos.elf 和 -device loader,file=benos.bin,addr=0x80200000 可能造成镜像覆盖，导致 OpenSBI 在跳转 S 模式时执行损坏指令，出现 sbi_trap_error。使用单一 -kernel benos.elf 的方式可以避免这类问题。

6.3 异常类型与可恢复性

本实验选择的是 Load Access Fault（非法访存），一般视为不可恢复异常。若需要在异常后继续执行，可以考虑：

- 使用非法指令异常（例如插入 .word 0xffffffff）并在 trap 中调整 sepc 跳过出错指令；
- 或在 trap 中判断异常类型并选择性恢复，这超出了本次作业范围。

7 结论

本文在 RISC-V 实验环境下完成了题目给出的三项要求：

(1) 设计了三层嵌套函数，并在最内层函数调用 dump_stack，实现了函数栈的回溯；

(2) 结合栈回溯过程，利用链接脚本提供的栈区边界和当前 sp，通过 SBI 控制台输出了栈的地址范围和大小；

(3) 通过串口按键 ''4'' 触发程序中构造的 Load Access Fault 异常，并采用 “trap 标志 + 安全上下文打印” 的方式，在终端上观测到异常被触发的证据。

同时，报告对 S/M 特权关系、外设访问限制以及 QEMU/OpenSBI 版本差异下的行为进行了分析，为后续进一步完善异常打印和恢复逻辑提供了参考。

参考文献

- RISC-V Privileged Architectures, Version 1.12.
- OpenSBI 官方文档与 QEMU RISC-V virt 平台说明。

附录 A：关键代码节选

A.1 kernel_main 主流程（节选）

```
void kernel_main(void)
{
    init_printk_done(sbi_putchar);
    trap_init();

    int r = func_level1();
    printk("[kernel_main] func_level1 result = %d\n", r);

    int tempchar = -1;
    while (tempchar != 52) {
        tempchar = uart_get();
    }
    printk("[kernel_main] key '4' detected, trigger fault...\n");
    test_fault();

    while (1) { ; }
}
```

A.2 三层调用与最内层回溯（节选）

```
static int func_level3(int a, int b, int c)
{
    printk("[func_level3] a=%d b=%d c=%d\n", a, b, c);
    dump_stack();
    return a + b + c;
}
```

A.3 UART 读取函数（节选）

```
char uart_get(void)
{
    if (readb(UART_LSR) & UART_LSR_DR)
        return readb(UART_DAT);
    else
        return -1;
```

```
}
```

A.4 异常触发汇编函数（节选）

```
.global trigger_load_access_fault
trigger_load_access_fault:
    li a0, 0x70000000
    ld a0, (a0)
    ret
```

A.5 Trap 入口汇编（节选）

```
.align 2
.global do_exception_vector
do_exception_vector:
    kernel_entry
    csrw sscratch, x0
    la ra, ret_from_exception
    mv a0, sp // a0 = pt_regs*
    mv a1, s4 // a1 = scause
    tail do_exception

ret_from_exception:
restore_all:
    kernel_exit
    sret
```