用户程序

操作系统 lab5 实验报告

梁景铭　王一诺　强博
南开大学计算机学院

2025 年 12 月 15 日

摘要

本实验基于 ucore 操作系统，深入探究了**用户进程的创建与管理机制**。实验首先通过分析内核线程与用户进程的区别，实现了从**内核态到用户态的特权级切换**，并构建了首个用户进程。在此基础上，设计并实现了**系统调用框架**，完成了 `sys_fork`、`sys_exec`、`sys_wait` 和 `sys_exit` 等核心系统调用，实现了用户进程的**生命周期管理**。进一步，实验扩展实现了 **Copy-on-Write (COW) 机制**，通过共享物理页和缺页异常处理优化了内存使用，并在此基础上复现与修复了著名的 **Dirty COW 漏洞**。最后，利用 GDB 对 **QEMU 的页表查询过程**进行了追踪，深入理解了软硬件协同的地址翻译流程。

关键词：用户进程，系统调用，Copy-on-Write，Dirty COW，GDB 调试

目录

1 实验内容与目标

1.1 实验目的

- 了解第一个用户进程创建过程
- 了解系统调用框架的实现机制
- 了解 ucore 如何实现系统调用 sys_fork/sys_exec/sys_exit/sys_wait 来进行进程管理

1.2 实验内容

在实验 4 中，我们已经完成了内核线程的实现，但截止目前，系统所有的运行流均在**内核态**下执行。

本实验的核心目标是创建**用户进程**，使其在**用户态**下执行。当用户进程需要操作系统（ucore）支持时，必须通过**系统调用**机制陷入内核来请求服务。

具体实验任务如下：

1. **构造第一个用户进程**：建立用户环境，完成从内核态到用户态的切换。
2. **实现核心系统调用**：通过实现以下系统调用，支持不同应用程序的加载与运行，并完成对用户进程生命周期的基本管理：
   - `sys_fork`：用于创建子进程。
   - `sys_exec`：用于加载并执行新的程序。
   - `sys_exit`：用于进程退出。
   - `sys_wait`：用于等待子进程结束并回收资源。

2 ucore 实验文档学习

2.1 实验执行流程概述

2.1.1 RISC-V 特权级回顾

在 Lab1 中我们已经讲解过 RISC-V 的特权级。简要回顾如下：

- **M 态**：Machine mode，最高权限，运行固件 OpenSBI，负责早期引导并向 S 态提供服务接口。
- **S 态**：Supervisor mode，内核态，操作系统内核运行在此级别，负责管理内存、中断等关键资源。
- **U 态**：User mode，用户态，运行普通用户程序，权限受到严格限制。

2.1.2 用户程序的运行与系统调用

在之前的实验中，内存管理和内核进程建立均在内核态完成。本实验的目标是让编译好的用户程序在**用户态**运行。

用户程序是由编译器生成的、包含执行代码与内存分配信息的目标程序。操作系统负责为其分配内存、建立进程并通过调度使其执行。由于用户程序运行在受限环境，无法直接访问硬件，**系统调用**成为了连接用户态与内核态的必要桥梁。

例如，当 C 程序调用 `printf` 时，交互流程如下：

1. 标准库将其转换为 `write` 系统调用。
2. 执行 **ecall** 指令触发异常，CPU 权限从 **U 态**提升至 **S 态**。
3. 跳转至预设的中断处理程序 **trap** 进行层层转发与处理。
4. 处理完成后，通过 **sret** 指令返回用户态。

2.1.3 流程可视化

下图展示了用户进程的正常执行循环（右侧路径）以及系统启动时首个进程的特殊创建过程（左侧路径）。

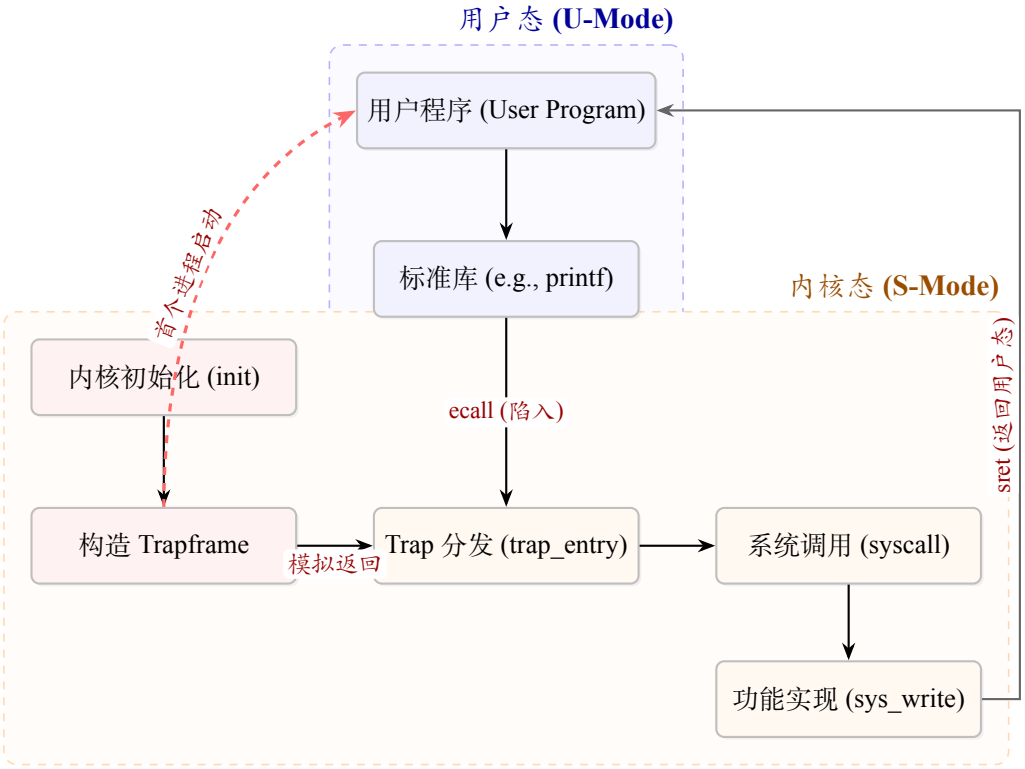

图 1: 用户进程执行循环与首次启动流程示意图

2.1.4 首个用户进程的启动

ucore 的初始化进程始终运行在内核态，无法像普通用户进程那样通过“从用户态陷入内核-再返回用户态”的完整闭环来切换特权级。

为了解决这一“鸡生蛋”问题，我们需要在内核态**构造一个 Trapframe**，模拟一次“从异常返回”的状态。通过这种方式，利用异常处理机制的返回流程（即执行 **sret**），我们得以第一次从 **S 态**切换进入 **U 态**，从而启动系统中的第一个用户进程。

2.2 用户进程的启动与管理

2.2.1 从内核态到用户态的入口：init_main 的演变

在之前的实验中，系统始终运行在**内核态**（S 态）。要实现 Lab5 的目标，即在用户态（U 态）运行程序并通过系统调用获取服务，首先需要解决“如何从内核态切换到用户态”的问题。

比较 Lab4 与 Lab5 的 `init_main` 函数，我们可以观察到核心逻辑的变化：

- **Lab4**：仅打印字符串，作为内核线程简单运行。
- **Lab5**：创建了一个新的内核线程 `user_main`，并使用 `do_wait` 等待其结束。这个 `user_main` 就是用户进程的“前身”。

Listing 1: Lab5 中的 init_main 函数核心逻辑

```
static int init_main(void *arg) {
    // 创建内核线程 user_main，它将负责加载并执行用户程序
    int pid = kernel_thread(user_main, NULL, 0);
    if (pid <= 0) {
        panic("create user_main failed.\n");
    }

    // 等待子进程（即 user_main 演变后的进程）退出
    while (do_wait(0, NULL) == 0) {
        schedule();
    }

    cprintf("all user-mode processes have quit.\n");
    return 0;
}
```

2.2.2 加载用户程序：user_main 与 execve

`user_main` 是一个内核线程，但它的任务是执行 `kernel_execve`，从而将自己“变身”为用户进程。

在 Lab5 的 Makefile 中，用户程序（如 `exit.c`）被编译并链接到了内核镜像中。编译器会自动生成 `_binary_..._start` 和 `_binary_..._size` 等符号，指向这些二进制代码在内存中的位置。

user_main 利用这些符号，通过宏 KERNEL_EXECVE 调用核心函数：

Listing 2: user_main 加载用户程序

```
// user_main 的实质操作
kern_execve("exit", _binary_obj___user_exit_out_start,
            _binary_obj___user_exit_out_size);
```

该函数会加载名为 exit 的用户程序二进制映像，覆盖当前进程的内存空间，并重新设置上下文。一旦执行成功，user_main 就从**内核进程**转变为**用户进程**，开始执行用户代码。

2.2.3 用户库与系统调用封装

用户程序（如 user/exit.c）运行在受限的 U 态，无法直接访问硬件或调用内核函数（如 sbi_console_putchar）。因此，必须通过**系统调用**来请求内核服务。

user/libs/ulib.c 和 stdio.c 提供了标准 C 库函数的封装，将普通函数调用转化为系统调用：

- **进程控制**：fork() 封装了 sys_fork()，exit() 封装了 sys_exit()。
- **输入输出**：用户态的 cprintf 不能直接调用 SBI 接口。它通过 sys_putc() 系统调用，请求内核代为打印字符。

Listing 3: 用户态输出函数的实现原理

```
// user/libs/stdio.c
static void cputch(int c, int *cnt) {
    // 关键点：用户态必须通过系统调用陷入内核来输出字符
    sys_putc(c);
    (*cnt) ++;
}
```

总结来说，用户程序的执行依赖于内核提供的加载机制（execve）以及标准库对系统调用接口的封装，从而实现了用户态代码与内核功能的交互。

2.3 系统调用的实现机制

系统调用是用户态程序获取内核态服务的唯一正规途径。在 ucore 中，其实现涉及用户态的触发与内核态的处理两个阶段。

2.3.1 核心系统调用概览

- sys_fork()：创建当前进程的副本（子进程）。
- sys_exec()：在当前进程内启动新程序，替换内存空间。
- sys_exit()：终止当前进程并释放资源。
- sys_wait()：挂起当前进程，等待子进程退出。

2.3.2 用户态：触发 Trap

在用户态，所有系统调用最终都会汇聚到 `syscall` 函数。该函数利用 RISC-V 的 `ecall` 指令触发一个同步异常（Trap），从而使 CPU 从 U 态陷入 S 态。

Listing 4: 用户态 syscall 函数（内联汇编）

```
// user/libs/syscall.c
static inline int syscall(int num, ...) {
    // ... 参数处理 ...
    asm volatile (
        "ld a0, %1\n"  // 系统调用号存入 a0
        "ld a1, %2\n"  // 参数存入 a1-a5
        // ...
        "ecall\n"      // 触发异常，陷入内核
        "sd a0, %0"    // 获取返回值
        : "=m" (ret)
        : "m"(num), "m"(a[0]), ...
        :"memory");
    return ret;
}
```

2.3.3 内核态：异常分发与处理

当 `ecall` 指令执行后，CPU 跳转到内核的 `trap` 入口。

1. **异常识别**：`trap.c` 中的 `exception_handler` 检查 `scause` 寄存器。若值为 `CAUSE_USER_ECALL`，则确认为系统调用。
2. **PC 调整**：为了避免死循环，处理程序必须将 `sepc`（异常返回地址）加 4，跳过触发异常的 `ecall` 指令。
3. **功能分发**：内核根据 `a0` 寄存器中的系统调用号，查找 `syscalls` 函数指针数组，调用对应的内核函数（如 `sys_fork` 最终调用 `do_fork`）。

Listing 5: 内核态系统调用分发

```
// kern/syscall/syscall.c
static int (*syscalls[])(uint64_t arg[]) = {
    [SYS_exit]    sys_exit,
    [SYS_fork]    sys_fork,
    // ... 其他系统调用映射
};

void syscall(void) {
    struct trapframe *tf = current->tf;
    int num = tf->gpr.a0; // 获取系统调用号
```

```
    // ...
    if (num >= 0 && num < NUM_SYSCALLS) {
        if (syscalls[num] != NULL) {
            // 转发给具体函数处理
            tf->gpr.a0 = syscalls[num](arg);
            return;
        }
    }
    // ...
}
```

2.4 第一次进入用户态的机制

我们需要通过 kernel_execve 来启动第一个用户进程，从而实现从内核态到用户态的“第一次跳跃”。

2.4.1 do_execve 的核心逻辑

do_execve 是加载新程序的后端实现。它负责为当前进程建立新的内存空间。其核心步骤包括：

1. **内存检查**：检查传入的程序名称等参数是否位于合法的内存区域。
2. **清理旧资源**：若当前进程已有内存空间（mm != NULL），则释放其页表和内存映射（exit_mmap, put_pgdir, mm_destroy）。
3. **加载新程序**：调用 load_icode 将二进制程序加载到内存，建立新的页表和 VMA。

2.4.2 内核态调用的困境与突破

理论上，我们可以让 user_main 直接调用 do_execve。但这行不通，因为 do_execve 仅负责构建用户程序的上下文（如页表、Trapframe），它本身是一个普通函数调用，无法完成 CPU 特权级的切换（从 S 态切换到 U 态）。

正常的特权级切换依赖于中断/异常的返回机制（sret）。在用户态，我们通过 ecall 触发异常来陷入内核；但在内核态（S 态），我们无法使用 ecall。

解决方案：利用 ebreak 模拟系统调用。我们采取了一种“取巧”的办法：在内核态执行 ebreak 触发断点异常。为了区分这是普通的断点还是特殊的系统调用模拟，我们将 a7 寄存器设为 10。

Listing 6: 利用 ebreak 模拟系统调用的 kernel_execve

```
// kern/process/proc.c
static int kernel_execve(const char *name, unsigned char *binary, size_t
    size) {
    int64_t ret=0, len = strlen(name);
    asm volatile(
        "li a0, %1\n"
```

```
        "li a7, 10\n" // 特殊标记，10 代表 SYS_exec
        "ebreak\n"    // 触发断点异常，陷入 trap handler
        "sw a0, %0\n"
        : "=m"(ret)
        : "i"(SYS_exec), ...
        : "memory");
    return ret;
}
```

在 trap.c 中，异常处理程序检测到 CAUSE_BREAKPOINT 且 a7 == 10 时，会将其转发给 syscall()，从而复用了系统调用的接口。

Listing 7: trap.c 中对断点异常的特殊处理

```
// kern/trap/trap.c
void exception_handler(struct trapframe *tf) {
    switch (tf->cause) {
        case CAUSE_BREAKPOINT:
            if(tf->gpr.a7 == 10){
                tf->epc += 4; // 跳过 ebreak 指令
                syscall();    // 执行系统调用
            }
            break;
        // ...
    }
}
```

2.4.3 特权级切换的关键：SSTATUS_SPP

当 sys_exec 执行完毕，从 Trap 返回时，CPU 会执行 sret 指令。

- sret 的行为：它会将特权级切换回 sstatus 寄存器中 SPP 位记录的状态。
- **问题**：由于我们是从内核态（S 态）通过 ebreak 进入的 Trap，硬件会自动将 SPP 设为 1 (S 态)。如果直接返回，CPU 将继续在 S 态运行用户程序，这不符合权限要求。
- **修正**：load_icode 函数在构造新进程的 Trapframe 时，会强制将 SSTATUS_SPP 位清零。这样，当 sret 执行时，CPU 就会依据被修改后的 SPP 位，正确地切换到 U 态，从而让首个用户进程在用户态开始执行。

2.5 中断处理与栈切换

由于用户进程拥有独立的“用户栈”,而内核处理中断时必须使用“内核栈”,因此 trapentry.S 中的汇编代码需要处理栈的切换。RISC-V 使用 sscratch 寄存器来协助这一过程。

2.5.1 双栈架构的挑战

- **用户栈**：用户进程运行时的栈。
- **内核栈**：内核处理中断和异常时的栈。
- **问题**：当从用户态陷入内核时，sp 指向用户栈。若直接使用 'sp' 压栈，会破坏用户栈。我们需要先将 sp 切换到内核栈。

2.5.2 SAVE_ALL 的栈切换逻辑

SAVE_ALL 宏利用 sscratch 寄存器来实现栈的交换。RISC-V 的约定是：

- 在 U 态运行时，sscratch 保存内核栈顶地址。
- 在 S 态运行时，sscratch 保存为 0。

Listing 8: trapentry.S 中的栈切换逻辑

```
    csrrw sp, sscratch, sp  # 交换 sp 和 sscratch

    # 此时:
    # 若 sp != 0, 说明之前在 U 态。交换后, sp 为内核栈, sscratch 为用户栈。
    # 若 sp == 0, 说明之前在 S 态。交换后, sp 为 0, sscratch 为内核栈。

    bnez sp, _save_context  # 若 sp 非 0, 跳过恢复步骤, 直接保存上下文

_restore_kernel_sp:
    csrr sp, sscratch       # 之前在 S 态, 需从 sscratch 拿回内核栈指针

_save_context:
    # ... 分配栈帧并保存寄存器 ...
```

2.5.3 RESTORE_ALL 的状态恢复

在中断返回前，RESTORE_ALL 需要根据即将返回的特权级（由 SSTATUS_SPP 决定）来正确设置 sscratch。

Listing 9: trapentry.S 中的状态恢复

```
    LOAD s1, 32*REGBYTES(sp) # 加载 sstatus
    andi s0, s1, SSTATUS_SPP # 检查 SPP 位
    bnez s0, _restore_context # 若返回 S 态, 无需处理 sscratch

_save_kernel_sp:
    # 若返回 U 态, 需将当前的内核栈指针保存到 sscratch
    # 以便下次从 U 态陷入时能找到内核栈
    addi s0, sp, 36 * REGBYTES
```

```
    csrw sscratch, s0

_restore_context:
    # ... 恢复寄存器 ...
```

2.6 进程退出机制

当进程完成任务后，必须执行退出操作以释放资源。ucore 采用“两步走”策略来完成资源回收：

1. **自身回收**：进程自行释放大部分资源（如用户虚拟内存空间、页表等）。
2. **父进程回收**：父进程负责释放子进程无法自我销毁的资源（如内核栈、进程控制块 `proc_struct`）。

这种设计的根本原因在于，进程在执行回收操作时仍然需要内核栈来执行指令，且进程控制块是其存在的唯一标识，因此无法由进程本身完全销毁。

2.6.1 do_exit 的核心流程

用户态的 `exit()` 函数通过系统调用 `sys_exit` 最终调用内核函数 `do_exit`。其主要步骤如下：

1. **合法性检查**：确保退出的不是 `idleproc` 或 `initproc`。
2. **释放内存资源**：如果存在内存管理结构 `mm`，且引用计数为 0，则切换回内核页表，并调用 `exit_mmap` 和 `put_pgdir` 释放用户虚拟内存和页表。
3. **更新状态**：将进程状态置为 `PROC_ZOMBIE`（僵尸态），并保存退出码。
4. **父子进程维护**：
   - 唤醒父进程：如果父进程处于等待子进程（`WT_CHILD`）状态，则唤醒它。
   - 托管子进程：将当前进程的所有子进程过继给 `initproc`。如果这些子进程已经是僵尸态，则通知 `initproc` 进行回收。
5. **调度切换**：执行 `schedule()`，主动让出 CPU，且不再返回。

Listing 10: do_exit 核心逻辑

```
int do_exit(int error_code) {
    // ... 合法性检查 ...

    if (mm != NULL) {
        lcr3(boot_cr3); // 切回内核页表
        // 释放用户内存资源
```

```
7           if (mm_count_dec(mm) == 0) {
8               exit_mmap(mm);
9               put_pgdir(mm);
10              mm_destroy(mm);
11          }
12          current->mm = NULL;
13      }
14
15      current->state = PROC_ZOMBIE; // 变为僵尸进程
16      current->exit_code = error_code;
17
18      // ... 唤醒父进程，将子进程过继给 initproc ...
19
20      schedule(); // 调度其他进程执行
21      panic("do_exit will not return!!");
22  }
```

3 实验练习解答

3.1 练习 1：加载应用程序并执行

3.1.1 问题重现

do_execve 函数调用 load_icode（位于 kern/process/proc.c 中）来加载并解析一个处于内存中的 ELF 执行文件格式的应用程序。你需要补充 load_icode 的第 6 步，建立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好 proc_struct 结构中的成员变量 trapframe 中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。需设置正确的 trapframe 内容。

请在实验报告中简要说明你的设计实现过程。

- 请简要描述这个用户态进程被 ucore 选择占用 CPU 执行（RUNNING 态）到具体执行应用程序第一条指令的整个经过。

3.1.2 实现

这一部分的目标是把内存中的 ELF 用户程序装载到当前进程的地址空间，并把内核态的执行流切换到用户态的入口。需要我们在 load_icode 中完成第 6 步的 trapframe 设置，同时梳理从 RUNNING 态到执行用户第一条指令的全链路。load_icode 先创建新的 mm 和页目录（mm_create、setup_pgdir），然后遍历 ELF 的每个 PT_LOAD 段，依据段标志拼出 VM_READ/WRITE/EXEC 和 RISC-V 的 PTE_R/W/X/U，用 mm_map 建立 VMA，再用 pgdir_alloc_page 把文件内容拷入物理页并补零 BSS 。用户栈被映射为 USTACKTOP-USTACKSIZE，一次性分配了 4 页用作栈顶防止越界。接着把新的 mm、pgdir 挂到进程并写入 satp，完成地址空间切换。

关键的第 6 步需要重置并构造 trapframe，确保 sret 能跳到用户态入口并使用用户栈：

Listing 11: 重置并构造 trapframe

```
    ...
    //(6) setup trapframe for user environment
    struct trapframe *tf = current->tf;
    // Keep sstatus
    uintptr_t sstatus = tf->status;
    memset(tf, 0, sizeof(struct trapframe));
    tf->gpr.sp = USTACKTOP;           // 用户栈顶
    tf->epc = elf->e_entry;           // ELF 入口
    tf->status = (sstatus & ~SSTATUS_SPP) | SSTATUS_SPIE; // 清 SPP=U, 开延迟中断
    tf->status &= ~SSTATUS_SIE;       // 返回时先关中断
    tf->gpr.a0 = 0;
    tf->gpr.a1 = 0                    // exec 成功返回 0
    ...
```

这样处理后，sret 依据 SSTATUS_SPP=0 自动切回 U 模式，epc 指向用户入口，sp 指向用户栈，保证用户程序能从预期的第一条指令开始运行。

3.1.3 思考题解答

在 ucore 中，一个用户态进程从被调度为 RUNNING 到执行用户程序第一条指令，实际经历了“选中 → 内核线程发起 exec→ 异常陷入 → 加载新映像 → 系统调用返回 →sret 切到 U 态”这一串动作：

- 调度与切换内核态上下文：调度器挑选可运行进程，调用 proc_run 把 current 设为目标进程，写 satp 切换到该进程页表，然后 switch_to 进行上下文切换，CPU 进入该进程的内核栈继续执行（通常是 user_main 内核线程）。
- 内核线程触发 exec：user_main 通过 kernel_execve 发起加载用户程序的请求。该函数用内联汇编设置 a7=10 后执行一次 ebreak，借断点异常复用 sys_exec 路径。
- 陷入异常，转为 syscall：断点异常到来，trap 入口 __alltraps 先把用户/内核寄存器和 sstatus/sepc 等保存到当前内核栈形成 trapframe。exception_handler 看到 a7==10，将 epc+=4 跳过 ebreak，再调用 syscall()，其中的 SYS_exec 最终执行 do_execve。
- 装载用户映像并构造新 trapframe：do_execve 释放旧用户态地址空间后调用 load_icode。load_icode 创建新的 mm/页表，遍历 ELF PT_LOAD 段映射代码/数据、补零 BSS，映射用户栈，然后把 current->mm/pgdir 和 satp 指向新页表。其中第 6 步清空并重建 trapframe：epc 设为 ELF 入口，sp 设为 USTACKTOP，status 清掉 SSTATUS_SPP 置位 SSTATUS_SPIE，让后续 sret 回到 U 模式并从入口开始执行。
- 返回路径与权限下降：load_icode 返回后，kernel_execve_ret 把内核栈上的 trapframe 调整到栈顶并跳到 __trapret。__trapret 用 trapframe 恢复寄存器和 sstatus/sepc，最后执行 sret。由

于 trapframe 中的 SSTATUS_SPP 已被清零，sret 会把 CPU 特权切到 U 态，PC 取 trapframe 的 epc（即用户程序入口），SP 取 trapframe 的 sp（用户栈顶）。

- 执行用户第一条指令：完成 sret 后，CPU 已在 U 模式、使用用户页表和用户栈，从 ELF 指定的入口地址取指，执行用户程序的第一条指令，进入正常的用户态运行。

3.2 练习 2：父进程复制自己的内存空间给子进程

3.2.1 问题重现

创建子进程的函数 do_fork 在执行中将拷贝当前进程（即父进程）的用户内存地址空间中的合法内容到新进程中（子进程），完成内存资源的复制。具体是通过 copy_range 函数（位于 kern/mm/pmm.c 中）实现的，请补充 copy_range 的实现，确保能够正确执行。

请在实验报告中简要说明你的设计实现过程。

- 如何设计实现 `Copy on Write` 机制？给出概要设计，鼓励给出详细设计。

3.2.2 实现

Listing 12: copy_range 函数实现

```
1  int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool
       share)
2  {
3      assert(start % PGSIZE == 0 && end % PGSIZE == 0);
4      assert(USER_ACCESS(start, end));
5
6      // copy content by page unit.
7      do {
8          // call get_pte to find process A's pte according to the addr start
9          pte_t *ptep = get_pte(from, start, 0);
10
11         if (ptep == NULL) {
12             start = ROUNDDOWN(start + PTSIZE, PTSIZE);
13             continue;
14         }
15
16         // 检查PTE是否有效
17         if (*ptep & PTE_V) {
18             // call get_pte to find process B's pte according to the addr
                   start
19             pte_t *nptep = get_pte(to, start, 1);  // 第二个参数为1表示如果
                   不存在就创建
20
21             if (nptep == NULL) {
22                 return -E_NO_MEM;
```

```
            }

            // 获取页面的权限位（保留用户权限位）
            uint32_t perm = (*ptep & PTE_USER);

            // 从源PTE获取物理页
            struct Page *page = pte2page(*ptep);

            // 为目标进程分配一个新的物理页
            struct Page *npage = alloc_page();

            if (npage == NULL) {
                return -E_NO_MEM;
            }

            // 获取内核虚拟地址以便进行内存拷贝
            void *src_kvaddr = page2kva(page);
            void *dst_kvaddr = page2kva(npage);

            // 将父进程页面的内容复制到子进程页面
            memcpy(dst_kvaddr, src_kvaddr, PGSIZE);

            // 将新页面插入到子进程的页表中
            int ret = page_insert(to, npage, start, perm);
            assert(ret == 0);

            // 增加新页面的引用计数
            page_ref_inc(npage);
        }

        start += PGSIZE;
    } while (start != 0 && start < end);

    return 0;
}
```

copy_range 函数在 do_fork 中被调用来实现进程内存空间的复制。该函数的核心逻辑是以页为单位，遍历指定的用户空间虚拟地址范围 [start, end)。对于每一个有效的虚拟地址，首先通过 get_pte 在父进程的页表中找到对应的页表项，确认其存在且有效（PTE_V 标志位被设置）。接着，为子进程分配一个新的物理页框，使用 memcpy 将父进程物理页中的全部内容（大小为 PGSIZE）完整地拷贝到这个新分配的页框中。然后，通过 page_insert 函数将子进程的虚拟地址映射到这个新分配的物理页框上，并继承父进程页面的用户访问权限。最后，更新新页面的引用计数，并循环处理下一页，直到所有地址都被复制。这个过程实现了物理内存的完全拷贝，确保父子进程在初始时刻拥有相同但完全独立的内存内容。

为了实现 Copy-on-Write 机制，需要进行系统性的修改。首先，需要扩展物理页的管理数据结构，例如在 struct Page 中增加 COW 引用计数字段 cow_refcount 和一个标志位 cow_shared，以跟踪有多少个进程共享该页以及其状态。其次，核心的修改点在于 copy_range 函数：当父进程的一个页面具有可写权限时，不再分配新页并拷贝内容，而是将该物理页的引用计数加一，然后让子进程的页表项直接指向这个相同的物理页。最关键的一步是，需要同时清除父进程和子进程对应页表项中的写权限位（PTE_W），并可以设置一个自定义的 COW 标志位（例如 PTE_C），从而将共享页面标记为只读。如果父进程的页面本来就是只读的，则可以直接共享而无需标记 COW。

当被标记为 COW 的页面发生写操作时，会触发页错误异常。这是 COW 机制的“写时复制”触发点。需要在页错误处理程序 pgfault_handler 中增加对 COW 情况的识别逻辑：检查出错的虚拟地址对应的页表项是否有效、是否缺少写权限但具有 COW 标志。如果是，则进入 COW 处理流程。处理时，首先检查该共享物理页的引用计数：如果当前只有这一个进程引用该页（即 cow_refcount 为 1），说明没有其他进程在共享，此时无需复制，只需要简单地恢复该页表项的写权限位，并清除 COW 标志即可。如果引用计数大于 1，说明仍有其他进程共享此页，此时才需要进行真正的复制：分配一个新的物理页，将原共享页的内容拷贝过来，然后将当前进程的页表项指向这个新页，并设置完整的可写权限。同时，需要将原共享物理页的引用计数减一。最后，更新相关页面的状态信息。

此外，还需要修改其他相关的内存管理函数以配合 COW 机制。例如，page_insert 在插入一个 COW 页面时需要特殊处理其权限和引用计数；page_remove_pte 在移除一个页表项时，如果对应的是 COW 页面，需要正确地递减引用计数而不是立即释放物理页，只有当最后一个引用被移除时才实际释放物理页框。通过这一系列的设计，COW 机制得以实现，它通过延迟物理页的拷贝操作到实际写入发生的那一刻，显著减少了 fork 系统调件的开销，提升了系统性能和内存利用效率。

3.3 练习 3：分析源代码理解实现

3.3.1 问题重现

练习内容：阅读分析源代码，理解进程执行 `fork/exec/wait/exit` 的实现，以及系统调用的实现（不需要编码）。

请在实验报告中简要说明你对 `fork/exec/wait/exit` 函数的分析。并回答如下问题：

- 请分析 `fork/exec/wait/exit` 的执行流程。重点关注哪些操作是在用户态完成，哪些是在内核态完成？内核态与用户态程序是如何交错执行的？内核态执行结果是如何返回给用户程序的？
- 请给出 ucore 中一个用户态进程的执行状态生命周期图（包含执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）。

执行：`make grade`。如果所显示的应用程序检测都输出 ok，则基本正确。（使用的是 qemu-4.1.1）

3.3.2 函数分析与执行流程

我们对核心的四个系统调用函数进行深入分析，重点阐述其在用户态与内核态的交互逻辑。

1. fork: 创建子进程 `fork` 用于创建一个与父进程完全相同的子进程，但在内核中拥有独立的资源。

- **用户态操作**：
  1. 调用标准库函数 `fork()`（位于 `user/libs/ulib.c`）。
  2. 该函数内部调用 `sys_fork()`，通过 `syscall(SYS_fork)` 触发系统调用。
  3. 执行 `ecall` 指令，CPU 陷入内核。
- **内核态操作**：
  1. **异常分发**：`trap.c` 捕获异常，转发给 `syscall`。
  2. **功能实现**：`syscall` 调用 `sys_fork`，最终执行 `do_fork`（位于 `kern/process/proc.c`）。
  3. **核心逻辑 (`do_fork`)**：
     - `alloc_proc`：为子进程分配 `proc_struct`。
     - `setup_kstack`：分配子进程的内核栈。
     - `copy_mm`：复制父进程的内存空间。如果是 COW（写时复制）机制，则仅复制页表并设置只读，提高效率。
     - `copy_thread`：复制父进程的 `trapframe`。**关键点**：此处将子进程 `tf->gpr.a0` 显式设置为 0，这是子进程 `fork` 返回值为 0 的根本原因。
     - `wakeup_proc`：将子进程状态设为 `PROC_RUNNABLE`，加入就绪队列。
  4. **返回值处理**：父进程的返回值（子进程 PID）由 `do_fork` 返回并存入父进程的 `tf->gpr.a0`。
- **返回用户态**：中断返回（`sret`）时，父子进程分别从各自的 `trapframe` 恢复上下文，父进程得到 PID，子进程得到 0。

2. exec: 加载新程序 `exec` 用于在当前进程的上下文中加载并运行一个新的程序，原有的代码和数据被替换。

- **用户态操作**：
  1. 准备参数，调用 `execve()`（库函数）。
  2. 通过 `syscall(SYS_exec, ...)` 触发 `ecall`。
- **内核态操作**：
  1. `syscall` 转发至 `sys_exec`，最终调用 `do_execve`。
  2. **核心逻辑 (`do_execve`)**：
     - `exit_mmap` & `put_pgdir`：释放当前进程旧的内存空间（页表、VMA）。

  - load_icode：解析新的 ELF 可执行文件，建立新的内存映射，分配用户栈。
  - **重置上下文**：关键在于 load_icode 会重置当前进程的 trapframe。其中 epc 被修改为新程序的入口地址（e_entry），sp 被修改为新的用户栈顶。

- **返回用户态**：当 trap 返回执行 sret 时，CPU 会跳转到 trapframe->epc 指向的地址。由于 epc 已被修改为新程序入口，因此 **exec 系统调用成功后不会返回到原调用点**，而是直接从新程序的第一条指令开始执行。

3. wait: 等待子进程 wait 允许父进程挂起，直到某个子进程退出，并获取其退出码。

- **用户态操作**：调用 wait() 或 waitpid()，触发 ecall。
- **内核态操作**：
  1. syscall 转发至 sys_wait，最终调用 do_wait。
  2. **核心逻辑 (do_wait)**：
     - 遍历子进程列表。
     - **情况 A：发现僵尸子进程**（状态为 PROC_ZOMBIE）：说明子进程已退出但资源未完全回收。父进程调用 put_proc 彻底释放子进程的内核栈和 PCB，并通过 copy_to_user 将子进程的 exit_code 写入用户提供的地址。
     - **情况 B：子进程还在运行**：父进程将自身状态设为 PROC_SLEEPING，标记等待原因为 WT_CHILD，然后调用 schedule() 主动让出 CPU。
- **交互与返回**：若父进程进入睡眠，它将一直阻塞。直到某个子进程调用 exit 时，会检测父进程状态并调用 wakeup_proc 唤醒父进程。父进程被唤醒后，在 do_wait 中循环继续，发现子进程已变僵尸，执行回收操作后返回用户态。

4. exit: 进程退出 exit 用于结束当前进程的运行。

- **用户态操作**：调用 exit(code)，触发 ecall。
- **内核态操作**：
  1. syscall 转发至 sys_exit，最终调用 do_exit。
  2. **核心逻辑 (do_exit)**：
     - **资源释放**：释放页表、虚拟内存空间（mm 结构）。注意，此时保留内核栈和 PCB。
     - **状态更新**：将进程状态设为 PROC_ZOMBIE，记录退出码。
     - **父子关系维护**：如果父进程正在等待（WT_CHILD），则唤醒父进程。同时，将当前进程的所有子进程过继给 initproc（1 号进程）领养。
     - **不可逆转的切换**：调用 schedule() 切换到其他进程。
- **返回**：do_exit **永远不会返回**。因为进程状态已变，且已让出 CPU，该进程的执行流在此终结，后续清理工作由父进程在 wait 中完成。

总结：用户态与内核态的交错与返回

- **交错执行**：程序正常在用户态执行指令。当需要内核服务（如创建进程、IO）时，通过 `ecall` 指令产生异常，硬件自动保存上下文并跳转到内核的 `trap` 入口，执行流进入内核态。内核处理完毕后，通过 `sret` 指令恢复上下文，执行流回到用户态。
- **结果返回**：内核函数的执行结果（如 PID 或错误码）并不直接“返回”给用户函数，而是通过修改当前进程的中断帧（`trapframe`）中的寄存器 `a0` 来实现。当 `sret` 恢复寄存器现场时，用户态程序就会在 `a0` 寄存器中看到返回值，从而模拟出函数调用的返回效果。

3.3.3 进程状态生命周期图

下图展示了 ucore 中用户进程从创建到销毁的完整状态流转。

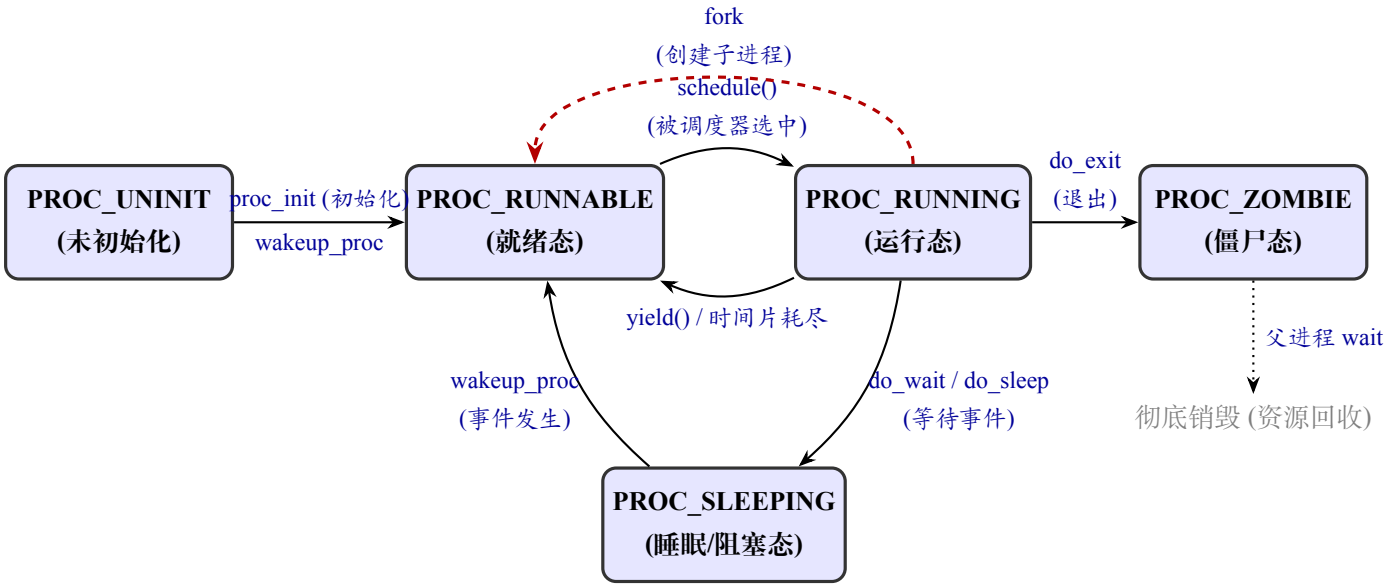

图 2: ucore 用户进程生命周期与状态转换图

3.4 扩展练习 Challenge1：实现 Copy on Write （COW）机制

3.4.1 问题重现

任务描述：在 ucore 中实现 Copy on Write (COW) 机制。
具体要求：

- 给出实现源码。
- 提供测试用例。
- 撰写设计报告（需包含 COW 机制下的各种状态转换说明，类似有限状态自动机）。

背景与原理：本扩展练习融合了进程管理与虚拟内存管理。在标准的 ucore 实现中，`fork` 操作会直接复制父进程的内存空间给子进程。为了提高效率，COW 机制要求：

1. **共享只读**：当父进程创建子进程时，不直接复制物理内存，而是将父进程的用户空间页表项设置为只读，子进程共享这些物理页面。

2. **按需复制**：当任一进程尝试修改这些只读页面时，CPU 触发 Page Fault 异常。ucore 捕获异常后，分配新的物理页，将原页面内容拷贝过去，并将页表项更新为可写，从而实现“写时复制”。

3. **隔离性**：确保一个进程的修改对另一个进程不可见，如同拥有独立的内存副本。

进阶挑战 (Big Challenge)：COW 的实现涉及复杂的并发与状态管理，极易引入竞态条件漏洞。请参考 https://dirtycow.ninja/ (Dirty COW 漏洞)，尝试分析能否在 ucore 的 COW 实现中模拟该类错误，并给出相应的解决方案与解释。

3.4.2 基础 COW 机制实现

为了实现 COW，我们需要修改内存管理的核心逻辑，涉及页表项定义、内存复制以及缺页异常处理。

1. 定义 PTE_COW 标志位 RISC-V 的页表项(PTE)保留了部分位供软件使用。我们在 `kern/mm/mmu.h` 中定义 `PTE_COW`，用于标记一个页面处于“写时复制”状态。

Listing 13: kern/mm/mmu.h 定义 PTE_COW

```
#define PTE_SOFT 0x300 // Reserved for Software
// 使用 RSW (Reserved for Software) 位中的一位作为 COW 标记
#define PTE_COW 0x100
```

2. 修改 fork 时的内存复制逻辑 在 `do_fork` → `copy_mm` → `dup_mmap` 的调用链中，最终会调用 `copy_range` 来复制页表。我们在 `kern/mm/pmm.c` 的 `copy_range` 函数中增加了 `share` 参数的处理逻辑：

- **共享物理页**：当 `share=1` 时，不分配新物理页。
- **权限降级**：将父子进程的 PTE 均设为**只读**（清除 `PTE_W`），并打上 `PTE_COW` 标记。
- **引用计数**：调用 `page_ref_inc` 增加物理页的引用计数，表示该页被多个进程共享。
- **刷新 TLB**：由于修改了页表权限，必须调用 `tlb_invalidate` 确保 CPU 看到最新的只读状态。

Listing 14: kern/mm/pmm.c 中的 copy_range 实现

```
int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool
    share) {
    // ... 遍历页表 ...
    if (*ptep & PTE_V) {
```

```
        if (share) {
            // COW: 去掉写权限(PTE_W), 设置软件 PTE_COW 位
            // 父子进程共享同一个物理页
            uint32_t perm = (*ptep & PTE_USER);
            perm = (perm | PTE_COW) & ~PTE_W;
            *ptep = (*ptep | PTE_COW) & ~PTE_W; // 修改父进程 PTE

            struct Page *page = pte2page(*ptep);
            // 映射到子进程, 引用计数 +1
            if (page_insert(to, page, start, perm) != 0) {
                return -E_NO_MEM;
            }
            tlb_invalidate(from, start); // 刷新父进程 TLB
        } else {
            // 非共享模式: 直接 alloc_page 并 memcpy (原逻辑)
        }
    }
    // ...
}
```

3. 实现写时复制异常处理 (do_pgfault) 当用户程序尝试写入被标记为 PTE_COW 的只读页面时，CPU 会触发 CAUSE_STORE_PAGE_FAULT。我们在 kern/mm/vmm.c 的 do_pgfault 中处理该逻辑：

- **判断触发条件**：PTE 有效 (PTE_V)、包含 PTE_COW 标记、且异常由写操作引起 (error_code & PTE_W)。
- **情况 A：引用计数 > 1 (共享中)**
  1. 分配一个新的物理页 (alloc_page)。
  2. 将原页面的内容完整拷贝到新页 (memcpy)。
  3. 更新 PTE 指向新页，设置为**可写** (PTE_W)，并清除 PTE_COW。
  4. 原物理页引用计数减 1。
- **情况 B：引用计数 == 1 (独占)**
  1. 说明其他共享进程已退出或已发生 COW，当前进程是该页的唯一拥有者。
  2. 直接修改当前 PTE，恢复 PTE_W，清除 PTE_COW，无需内存拷贝。

Listing 15: kern/mm/vmm.c 中的 do_pgfault 核心逻辑

```
int do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr) {
    // ... 获取 PTE ...
    if (*ptep & PTE_V) { // 页表项有效
        // 检查是否为写时复制触发的异常
```

```
5           if ((*ptep & PTE_COW) && (error_code & PTE_W)) {
6               struct Page *page = pte2page(*ptep);
7
8               // Case 1: 页面被多个进程共享
9               if (page_ref(page) > 1) {
10                  struct Page *npage = alloc_page(); // 分配新页
11                  memcpy(page2kva(npage), page2kva(page), PGSIZE); // 复制内
                        容
12
13                  // 重新映射: 新页拥有写权限, 且不再是 COW
14                  uint32_t perm = vma_perm(vma->vm_flags) | PTE_W;
15                  perm &= ~PTE_COW;
16                  return page_insert(mm->pgdir, npage, la, perm);
17              }
18              // Case 2: 页面只有一个引用, 直接恢复写权限
19              else {
20                  *ptep = (*ptep | PTE_W) & ~PTE_COW;
21                  tlb_invalidate(mm->pgdir, la);
22                  return 0;
23              }
24          }
25      }
26      // ... 处理普通缺页 ...
27  }
```

3.4.3 Dirty COW 漏洞复现与修复

为了深入理解 COW 的复杂性，我们在 ucore 中模拟了著名的 Dirty COW 漏洞场景。该漏洞的核心在于：**内核在处理某些写入操作时，若未正确处理 COW 状态，可能直接写入共享的物理页，导致只读数据被篡改**。

1. 用户态 PoC 我们编写了 `user/dirtycow.c`。该程序试图修改位于只读数据段（`.rodata`）的常量字符串 `victim`。

Listing 16: user/dirtycow.c 用户态攻击代码

```
1  // 位于只读段的受害者字符串
2  static const char victim[] = "DIRTYCOW DEMO: READ ONLY DATA";
3  static const char payload[] = "DIRTYCOW DEMO: OWNED BY USER!";
4
5  int main(void) {
6      // 1. 切换到修复模式, 预期写入失败
7      dirtycowctl(0);
8      try_patch("fix"); // 调用 mempoke 尝试写 victim
9
```

```
    // 2. 切换到漏洞模式，预期写入成功（篡改只读内存）
    dirtycowctl(1);
    try_patch("bug");

    return 0;
}
```

2. 内核态漏洞模拟 (mempoke) 我们增加了一个自定义系统调用 sys_mempoke，允许用户请求内核代写虚拟地址。其后端实现 dirtycow_mempoke 位于 kern/mm/cow.c。

这个函数根据 dirtycow_stats.emulate_bug 的状态决定行为，模拟了漏洞的存在与修复：

- **Bug 模式 (模拟漏洞)**：内核在写入用户页之前，**跳过**了 do_pgfault 的检查。这意味着，即使用户页是只读的共享页（PTE_W=0, PTE_COW=1），内核也利用自身的高权限直接写入了该物理页。**后果**：该物理页如果是共享的（例如 fork 出来的父子进程共享，或共享库代码段），所有映射该页的进程都会看到被篡改的数据，破坏了内存隔离。
- **Fix 模式 (修复方案)**：内核在写入前，强制调用 dirtycow_prepare_page → do_pgfault(mm, PTE_W, la)。**原理**：do_pgfault 会检测到该页是 COW 页，从而触发标准的 “分配新页-复制内容-断开共享” 流程。内核随后的写入操作只会发生在新的私有页上，原始的共享页不受影响。

Listing 17: kern/mm/cow.c 核心模拟逻辑

```
// 在写入用户内存前调用
static int dirtycow_prepare_page(struct mm_struct *mm, uintptr_t la) {
    pte_t *ptep = get_pte(mm->pgdir, la, 0);

    // Bug 模式：直接放行，模拟内核忽略了 COW 检查
    if (dirtycow_stats.emulate_bug) {
        dirtycow_stats.unsafe_writes++;
        return 0;
    }

    // Fix 模式：如果页面不可写，强制触发 Page Fault 处理流程
    // 这会调用 do_pgfault 进行 COW 拆分，确保写入的是私有页
    if ((*ptep & PTE_W) == 0) {
        int ret = do_pgfault(mm, PTE_W, la);
        if (ret != 0) return ret;
        dirtycow_stats.repaired_writes++;
    }
    return 0;
}
```

3. 系统调用支持 我们在 kern/syscall/syscall.c 中注册了两个新的系统调用：

- SYS_mempoke：调用 dirtycow_mempoke 执行写入。
- SYS_dirtycowctl：调用 dirtycow_set_mode 切换 Bug/Fix 模式。

同时在 kern/debug/kmonitor.c 中添加了 dirtycow 命令，用于在内核监视器中实时查看攻击统计数据（unsafe_writes vs repaired_writes）。

3.4.4 实验结果与验证

我们通过编写两个用户态程序来验证 COW 机制和 Dirty COW 修复的效果。

1. 验证基础 COW 机制 (user/cow.c) user/cow.c 演示了父子进程共享内存但写入隔离的特性。

1. 父进程初始化一个缓冲区，并在首尾写入'A' 和'X'。
2. 调用 fork()。此时父子共享物理页，权限为只读。
3. 子进程修改缓冲区，写入'B' 和'Y'。这会触发两次写时复制（Page Fault），内核为子进程分配新页。
4. 父进程检查自己的缓冲区，应当仍为'A' 和'X'，证明子进程的修改未影响父进程。

运行 make run-nox-cow，日志如下：

Listing 18: make run-nox-cow 运行日志

```
kernel_execve: pid = 2, name = "cow".
// ... 子进程触发写异常 ...
Store/AMO page fault: pid 3 epc=0x800572 badva=0x801000 in_kernel=0
Store/AMO page fault: pid 3 epc=0x80058a badva=0x802000 in_kernel=0
child wrote pages: B Y
cow pass.
all user-mode processes have quit.
```

日志中的 Store/AMO page fault 表明子进程在写入时确实触发了缺页异常，内核成功捕获并处理了 COW。最后输出 cow pass 证明父进程数据未被篡改。

2. 验证 Dirty COW 漏洞修复 (user/dirtycow.c) user/dirtycow.c 演示了如果内核不遵守 COW 规则会发生什么。

1. 程序定义了一个只读字符串常量 victim。
2. 使用 dirtycowctl(0) 切换到 **修复模式**。尝试使用 mempoke 系统调用覆写 victim。内核检测到写入只读页，强制触发 COW 分离。
3. 使用 dirtycowctl(1) 切换到 **漏洞模式**。尝试覆写 victim。内核直接写入物理地址，绕过 COW。

运行 `make run-nox-dirtycow`，日志如下：

Listing 19: make run-nox-dirtycow 运行日志

```
kernel_execve: pid = 2, name = "dirtycow".
[dirtycow] pid=2 victim@0x800a10 -> "DIRTYCOW DEMO: READ ONLY DATA"
[dirtycow] dirtycowctl(-1) queries mode, dirtycowctl(0/1) toggles fix/bug.
// Fix 模式：写入被阻拦或重定向到私有页，原只读数据未变
[dirtycow][fix] blocked (ret=-3), victim -> "DIRTYCOW DEMO: READ ONLY DATA"
// Bug 模式：写入成功，只读数据被篡改！
[dirtycow][bug] succeeded, victim -> "DIRTYCOW DEMO: OWNED BY USER!"
```

这直观地展示了 Dirty COW 漏洞的危害以及我们在 Fix 模式下修复的有效性。

3.4.5 总结：COW 机制与漏洞防御逻辑图

下图总结了 ucore 中 COW 的标准流程以及 Dirty COW 漏洞在逻辑上的差异点。

3.5 扩展练习 Challenge2：用户程序是何时被预先加载到内存中的

在 uCore 操作系统中，用户程序的加载是一个多阶段的过程。理解程序何时被加载到内存对于理解操作系统的进程管理机制至关重要。以下是用户程序加载的完整时间线和关键步骤。

3.5.1 1. 内核启动时的静态加载

在内核启动阶段，用户程序就已经被“预加载”到内存中，但此时它们还只是存储在磁盘镜像中的二进制文件。在 uCore 的实验环境中，这一过程发生在编译和制作磁盘镜像时：

- **编译阶段**: 用户程序（如 `sh`, `ls` 等）被编译为 ELF 格式的可执行文件
- **制作镜像**: 这些 ELF 文件被嵌入到磁盘镜像文件中（如 `ucore.img`）
- **启动加载**: QEMU 启动时，整个磁盘镜像被加载到模拟的物理内存中

此时的用户程序处于“静默”状态，尚未被操作系统识别为可执行实体。

3.5.2 2. 用户程序的激活加载时间点

真正的用户程序加载发生在以下几个关键时间点：

(1) 内核初始化完成时 (`kern_init`) 在系统初始化函数 `kern_init()` 的最后阶段，通过调用 `proc_init()` 和创建第一个用户进程时，会加载初始用户程序。

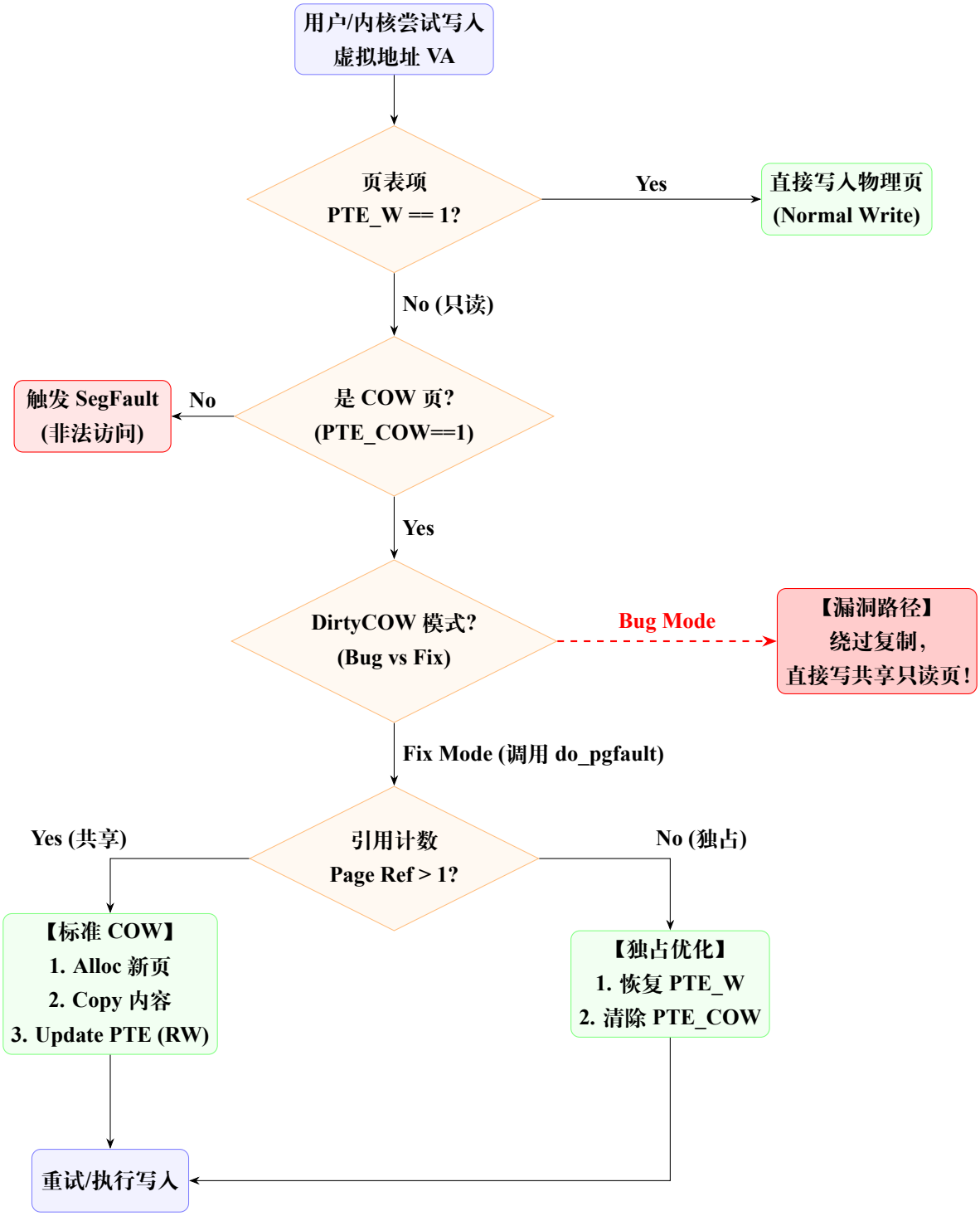

图 3: ucore COW 机制执行流程与 Dirty COW 漏洞逻辑对比图

(2) init 进程执行时 (init_main) 第一个用户进程 init 会执行以下操作：

```
// kern/process/proc.c
static int
init_main(void *arg)
{
    size_t nr_free_pages_store = nr_free_pages();
    size_t kernel_allocated_store = kallocated();

    int pid = kernel_thread(user_main, NULL, 0);
    if (pid <= 0)
    {
        panic("create␣user_main␣failed.\n");
    }

    while (do_wait(0, NULL) == 0)
    {
        schedule();
    }

    cprintf("all␣user-mode␣processes␣have␣quit.\n");
    assert(initproc->cptr == NULL && initproc->yptr == NULL && initproc->
       optr == NULL);
    assert(nr_process == 2);
    assert(list_next(&proc_list) == &(initproc->list_link));
    assert(list_prev(&proc_list) == &(initproc->list_link));

    cprintf("init␣check␣memory␣pass.\n");
    return 0;
}
```

(3) 实际程序加载 (do_execve) 真正的程序加载发生在 do_execve 系统调用中，该函数负责：

1. **参数验证**: 检查文件路径和参数的合法性
2. **文件查找**: 在文件系统中查找 ELF 可执行文件
3. **ELF 解析**: 解析 ELF 文件头，获取程序入口点、段信息等
4. **内存分配**: 为代码段、数据段、BSS 段等分配内存空间
5. **内容加载**: 将程序的各个段从磁盘读取到内存
6. **上下文设置**: 设置进程的执行上下文（EIP、ESP 等）

3.5.3 3. 加载过程的详细步骤

load_icode 函数是用户程序加载的核心，它执行以下操作：

```
// kern/process/proc.c
static int
load_icode(unsigned char *binary, size_t size)
{
    if (current->mm != NULL)
    {
        panic("load_icode: current->mm must be empty.\n");
    }

    int ret = -E_NO_MEM;
    struct mm_struct *mm;
    //(1) create a new mm for current process
    if ((mm = mm_create()) == NULL)
    {
        goto bad_mm;
    }
    //(2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
    if (setup_pgdir(mm) != 0)
    {
        goto bad_pgdir_cleanup_mm;
    }
    //(3) copy TEXT/DATA section, build BSS parts in binary to memory space
        of process
    struct Page *page;
    //(3.1) get the file header of the bianry program (ELF format)
    struct elfhdr *elf = (struct elfhdr *)binary;
    //(3.2) get the entry of the program section headers of the bianry
       program (ELF format)
    struct proghdr *ph = (struct proghdr *)(binary + elf->e_phoff);
    //(3.3) This program is valid?
    if (elf->e_magic != ELF_MAGIC)
    {
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
    }

    uint32_t vm_flags, perm;
    struct proghdr *ph_end = ph + elf->e_phnum;
    for (; ph < ph_end; ph++)
    {
        //(3.4) find every program section headers
        if (ph->p_type != ELF_PT_LOAD)
```

```
41          {
42              continue;
43          }
44          if (ph->p_filesz > ph->p_memsz)
45          {
46              ret = -E_INVAL_ELF;
47              goto bad_cleanup_mmap;
48          }
49          if (ph->p_filesz == 0)
50          {
51              // continue ;
52          }
53          //(3.5) call mm_map fun to setup the new vma ( ph->p_va, ph->
                p_memsz)
54          vm_flags = 0, perm = PTE_U | PTE_V;
55          if (ph->p_flags & ELF_PF_X)
56              vm_flags |= VM_EXEC;
57          if (ph->p_flags & ELF_PF_W)
58              vm_flags |= VM_WRITE;
59          if (ph->p_flags & ELF_PF_R)
60              vm_flags |= VM_READ;
61          // modify the perm bits here for RISC-V
62          if (vm_flags & VM_READ)
63              perm |= PTE_R;
64          if (vm_flags & VM_WRITE)
65              perm |= (PTE_W | PTE_R);
66          if (vm_flags & VM_EXEC)
67              perm |= PTE_X;
68          if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0)
69          {
70              goto bad_cleanup_mmap;
71          }
72          unsigned char *from = binary + ph->p_offset;
73          size_t off, size;
74          uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start, PGSIZE);
75
76          ret = -E_NO_MEM;
77
78          //(3.6) alloc memory, and  copy the contents of every program
                section (from, from+end) to process's memory (la, la+end)
79          end = ph->p_va + ph->p_filesz;
80          //(3.6.1) copy TEXT/DATA section of bianry program
81          while (start < end)
82          {
83              if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL)
```

```
84              {
85                  goto bad_cleanup_mmap;
86              }
87              off = start - la, size = PGSIZE - off, la += PGSIZE;
88              if (end < la)
89              {
90                  size -= la - end;
91              }
92              memcpy(page2kva(page) + off, from, size);
93              start += size, from += size;
94          }
95
96          //(3.6.2) build BSS section of binary program
97          end = ph->p_va + ph->p_memsz;
98          if (start < la)
99          {
100             /* ph->p_memsz == ph->p_filesz */
101             if (start == end)
102             {
103                 continue;
104             }
105             off = start + PGSIZE - la, size = PGSIZE - off;
106             if (end < la)
107             {
108                 size -= la - end;
109             }
110             memset(page2kva(page) + off, 0, size);
111             start += size;
112             assert((end < la && start == end) || (end >= la && start == la)
                    );
113         }
114         while (start < end)
115         {
116             if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL)
117             {
118                 goto bad_cleanup_mmap;
119             }
120             off = start - la, size = PGSIZE - off, la += PGSIZE;
121             if (end < la)
122             {
123                 size -= la - end;
124             }
125             memset(page2kva(page) + off, 0, size);
126             start += size;
127         }
```

```
128     }
129     //(4) build user stack memory
130     vm_flags = VM_READ | VM_WRITE | VM_STACK;
131     if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags,
            NULL)) != 0)
132     {
133         goto bad_cleanup_mmap;
134     }
135     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP - PGSIZE, PTE_USER) !=
            NULL);
136     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP - 2 * PGSIZE, PTE_USER) !=
             NULL);
137     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP - 3 * PGSIZE, PTE_USER) !=
             NULL);
138     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP - 4 * PGSIZE, PTE_USER) !=
             NULL);
139
140     //(5) set current process's mm, sr3, and set satp reg = physical addr
            of Page Directory
141     mm_count_inc(mm);
142     current->mm = mm;
143     current->pgdir = PADDR(mm->pgdir);
144     lsatp(PADDR(mm->pgdir));
145
146     //(6) setup trapframe for user environment
147     struct trapframe *tf = current->tf;
148     // Keep sstatus
149     uintptr_t sstatus = tf->status;
150     memset(tf, 0, sizeof(struct trapframe));
151     tf->gpr.sp = USTACKTOP;
152     tf->epc = elf->e_entry;
153     tf->status = (sstatus & ~SSTATUS_SPP) | SSTATUS_SPIE;
154     tf->status &= ~SSTATUS_SIE;
155     tf->gpr.a0 = 0;
156     tf->gpr.a1 = 0;
157
158     ret = 0;
159 out:
160     return ret;
161 bad_cleanup_mmap:
162     exit_mmap(mm);
163 bad_elf_cleanup_pgdir:
164     put_pgdir(mm);
165 bad_pgdir_cleanup_mm:
166     mm_destroy(mm);
```

```
167 bad_mm:
168     goto out;
169 }
```

3.5.4 4. 延迟加载与按需分页

在现代操作系统中，用户程序的加载往往采用延迟加载策略。uCore 也支持类似机制：

- **按需分页**: 程序段不会立即全部加载，而是在发生缺页异常时按需加载
- **VMA 机制**: 先建立虚拟内存区域映射，实际物理页分配延迟到访问时
- **写时复制**: 对于可写数据段，使用 COW 技术延迟实际的页面分配

这一机制的实现依赖于 do_pgfault 函数：

```
1  // kern/mm/vmm.c
2  int do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr)
3  {
4      uintptr_t la = ROUNDDOWN(addr, PGSIZE);
5      struct vma_struct *vma = find_vma(mm, la);
6      if (vma == NULL || vma->vm_start > la)
7      {
8          return -E_INVAL;
9      }
10
11     uint32_t perm = vma_perm(vma->vm_flags);
12     pte_t *ptep = get_pte(mm->pgdir, la, 1);
13     if (ptep == NULL)
14     {
15         return -E_NO_MEM;
16     }
17
18     bool need_write = (error_code & PTE_W) != 0;
19     if (*ptep & PTE_V)
20     {
21         // 已有映射，只可能是写时复制触发
22         if ((*ptep & PTE_COW) && need_write)
23         {
24             struct Page *page = pte2page(*ptep);
25             if (page == NULL)
26             {
27                 return -E_INVAL;
28             }
29             if (page_ref(page) > 1)
30             {
```

```
                struct Page *npage = alloc_page();
                if (npage == NULL)
                {
                    return -E_NO_MEM;
                }
                memcpy(page2kva(npage), page2kva(page), PGSIZE);
                perm &= ~PTE_COW;
                perm |= PTE_W;
                return page_insert(mm->pgdir, npage, la, perm);
            }
            // 只有一个引用，直接去掉 COW 标记即可
            *ptep = (*ptep | PTE_W) & ~PTE_COW;
            tlb_invalidate(mm->pgdir, la);
            return 0;
        }
        return -E_INVAL;
    }

    struct Page *npage = alloc_page();
    if (npage == NULL)
    {
        return -E_NO_MEM;
    }
    memset(page2kva(npage), 0, PGSIZE);
    return page_insert(mm->pgdir, npage, la, perm);
}
```

3.5.5 5. 实验环境中的特殊加载机制

在 uCore 实验环境中，用户程序的加载具有以下特点：

表 1: 用户程序加载的关键机制

机制	描述	相关函数
ELF 解析	解析可执行文件格式	`load_elf`
内存映射	建立虚拟地址到物理地址的映射	`mm_map`
页表设置	设置进程页表	`setup_pgdir`
上下文切换	设置执行上下文	`context_switch`
按需加载	延迟加载程序页面	`do_pgfault`

3.5.6 6. 总结

用户程序在 uCore 中的加载是一个分阶段的过程：

1. **编译时预置**: 用户程序被编译并嵌入到磁盘镜像中
2. **内核初始化**: 内核启动并初始化进程管理机制
3. **进程创建**: 通过 `do_execve` 创建用户进程
4. **程序加载**: 调用 `load_icode` 加载程序到内存
5. **按需加载**: 通过缺页异常机制实现延迟加载

3.6 lab2 分支任务：gdb 调试页表查询过程

3.6.1 任务简述

本次双重 GDB 调试的目标，是在 QEMU 内部观测一条 ucore 访存/取指指令的虚拟地址是如何被翻译成物理地址，并结合源码说明页表翻译的关键操作、TLB 查找逻辑，以及 QEMU 软件 TLB 与真实硬件 TLB 的差异。实验采用三终端：终端 1 启动调试版 QEMU，终端 2 用主机 gdb 附加 QEMU 进程、调试 QEMU 源码，终端 3 用 riscv64-unknown-elf-gdb 调试 ucore。

3.6.2 调试与观察

修改好 qemu 和 makefile 确保是使用调试版 qemu-system-riscv64 后，首先在终端 1 运行 make debug，QEMU 启动后暂停等待远程 GDB。

然后，在终端 2 用 sudo gdb 附加 QEMU 进程（利用 pgrep 找 PID），设置 handle SIGPIPE nostop noprint，避免信号打断。随后在 target/riscv/cpu_helper.c 的 riscv_cpu_tlb_fill 处下断点，并加条件 address==0xffffffffc02000d8, 也就是说正常情况下在取指 kern_init 虚拟地址时将会触发该断点，随后 continue 等待触发。

```
@:~/qemu-4.1.1$ pgrep -f qemu-system-riscv64
88097
@:~/qemu-4.1.1$ sudo gdb
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.
    html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
```

```
Type "apropos word" to search for commands related to "word".
warning: File "/home/nuo/qemu-4.1.1/.gdbinit" auto-loading has been
    declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-
    load".
To enable execution of this file add
        add-auto-load-safe-path /home/nuo/qemu-4.1.1/.gdbinit
line to your configuration file "/root/.config/gdb/gdbinit".
To completely disable this security protection add
--Type <RET> for more, q to quit, c to continue without paging--c
        set auto-load safe-path /
line to your configuration file "/root/.config/gdb/gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the
    shell:
        info "(gdb)Auto-loading safe path"
(gdb) attach 88097
Attaching to process 88097
[New LWP 88099]
[New LWP 88098]
warning: could not find '.gnu_debugaltlink' file for /lib/x86_64-linux-gnu/
    libglib-2.0.so.0
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
0x000077df7eb1ba30 in __GI_ppoll (fds=0x59068d126ad0,
    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      Print   Pass to program Description
SIGPIPE       No        No      Yes             Broken pipe
(gdb) b riscv_cpu_tlb_fill if address==0xffffffffc02000d8
Breakpoint 1 at 0x59066b19fa28: file /home/nuo/qemu-4.1.1/target/riscv/
    cpu_helper.c, line 438.
(gdb) c
Continuing.
```

接下来在终端 3 用 make gdb 连接 QEMU 的 GDB stub，并开始运行，并触发终端 2 设置的条件断点。

```
(gdb) c
Continuing.
[Switching to Thread 0x77df77fff6c0 (LWP 88099)]

Thread 2 "qemu-system-ris" hit Breakpoint 1, riscv_cpu_tlb_fill (cs=0
    x59068d0da650, address=18446744072637907160,
```

```
    size=0, access_type=MMU_INST_FETCH, mmu_idx=1,
    probe=false, retaddr=0)
    at /home/nuo/qemu-4.1.1/target/riscv/cpu_helper.c:438
438     {
```

执行 tb、info args 和 p/x address 观察此时的调用栈以及此处 riscv_cpu_tlb_fill 函数的参数：

```
(gdb) bt
#0  riscv_cpu_tlb_fill (cs=0x59068d0da650,
    address=18446744072637907160, size=0,
    access_type=MMU_INST_FETCH, mmu_idx=1, probe=false,
    retaddr=0)
    at /home/nuo/qemu-4.1.1/target/riscv/cpu_helper.c:438
#1  0x000059066b0e5d7f in tlb_fill (cpu=0x59068d0da650,
    addr=18446744072637907160, size=0,
    access_type=MMU_INST_FETCH, mmu_idx=1, retaddr=0)
    at /home/nuo/qemu-4.1.1/accel/tcg/cputlb.c:878
#2  0x000059066b0e6429 in get_page_addr_code (
    env=0x59068d0e3060, addr=18446744072637907160)
    at /home/nuo/qemu-4.1.1/accel/tcg/cputlb.c:1032
#3  0x000059066b10ae8f in tb_htable_lookup (
    cpu=0x59068d0da650, pc=18446744072637907160,
    cs_base=0, flags=24577, cf_mask=4278714368)
    at /home/nuo/qemu-4.1.1/accel/tcg/cpu-exec.c:339
#4  0x000059066b103e65 in tb_lookup__cpu_state (
    cpu=0x59068d0da650, pc=0x77df77ffe3e0,
    cs_base=0x77df77ffe3d8, flags=0x77df77ffe3d4,
    cf_mask=4278714368)
    at /home/nuo/qemu-4.1.1/include/exec/tb-lookup.h:43
#5  0x000059066b10421e in helper_lookup_tb_ptr (
    env=0x59068d0e3060)
    at /home/nuo/qemu-4.1.1/accel/tcg/tcg-runtime.c:154
#6  0x000077df7c134df7 in code_gen_buffer ()
#7  0x000059066b10a939 in cpu_tb_exec (cpu=0x59068d0da650,
    itb=0x77df7c134d00 <code_gen_buffer+330963>)
    at /home/nuo/qemu-4.1.1/accel/tcg/cpu-exec.c:173
#8  0x000059066b10b77f in cpu_loop_exec_tb (
    cpu=0x59068d0da650,
    tb=0x77df7c134d00 <code_gen_buffer+330963>,
    last_tb=0x77df77ffe988, tb_exit=0x77df77ffe980)
    at /home/nuo/qemu-4.1.1/accel/tcg/cpu-exec.c:621
#9  0x000059066b10baa5 in cpu_exec (cpu=0x59068d0da650)
    at /home/nuo/qemu-4.1.1/accel/tcg/cpu-exec.c:732
#10 0x000059066b0bdcdf in tcg_cpu_exec (cpu=0x59068d0da650)
    at /home/nuo/qemu-4.1.1/cpus.c:1435
#11 0x000059066b0be598 in qemu_tcg_cpu_thread_fn (
```

```
    arg=0x59068d0da650) at /home/nuo/qemu-4.1.1/cpus.c:1743
#12 0x000059066b541728 in qemu_thread_start (
    args=0x59068d0f0ce0) at util/qemu-thread-posix.c:502
#13 0x000077df7ea9caa4 in start_thread (
    arg=<optimized out>) at ./nptl/pthread_create.c:447
#14 0x000077df7eb29c6c in clone3 ()
    at ../sysdeps/unix/sysv/linux/x86_64/clone3.S:78

(gdb) info args
cs = 0x59068d0da650
address = 18446744072637907160
size = 0
access_type = MMU_INST_FETCH
mmu_idx = 1
probe = false
retaddr = 0

(gdb) p/x address
$1 = 0xffffffffc02000d8
```

实参显示 address=0xffffffffc02000d8，access_type=MMU_INST_FETCH （表明为指令取址），mmu_idx=1 （表明为 S 模式），说明这是内核首次取指 kern_init 虚拟地址时发生的 TLB 未命中，进入 QEMU 的 TLB 填充逻辑。回溯中能看到调用链：tlb_fill →get_page_addr_code → tb_htable_lookup → cpu_exec，对应硬件上 TLB miss 触发翻译的路径。

为了观察页表遍历，我又在 get_physical_address 和 tlb_set_page 下了断点。继续运行后命中 get_physical_address。

```
(gdb) b get_physical_address
Breakpoint 2 at 0x59066b19ee13: file /home/nuo/qemu-4.1.1/target/riscv/
    cpu_helper.c, line 158.
(gdb) b tlb_set_page
Breakpoint 3 at 0x59066b0e5c74: file /home/nuo/qemu-4.1.1/accel/tcg/cputlb.
    c, line 847.
(gdb) c
Continuing.

Thread 2 "qemu-system-ris" hit Breakpoint 2, get_physical_address (env=0
    x59068d0e3060, physical=0x77df77ffe1f0,
     prot=0x77df77ffe1e4, addr=18446744072637907160,
     access_type=2, mmu_idx=1)
     at /home/nuo/qemu-4.1.1/target/riscv/cpu_helper.c:158
158     {
(gdb) p/x addr
$2 = 0xffffffffc02000d8
```

接下来开始单步调试观察 get_physical_address 的执行过程，此处展示一些看到的关键信息：

```
...
(gdb) step
184             base = get_field(env->satp, SATP_PPN) << PGSHIFT;
```

这是模式与根基址解析，此处从 satp 取出 PPN 和模式，会得到 base = 0x80205000，vm = Sv39，levels=3，ptidxbits=9。这将确立三级页表，根页表物理基址在 0x80205000。

```
(gdb) step
get_physical_address (env=0x59068d0e3060,
    physical=0x77df77ffe1f0, prot=0x77df77ffe1e4,
    addr=18446744072637907160, access_type=2, mmu_idx=1)
    at /home/nuo/qemu-4.1.1/target/riscv/cpu_helper.c:224
224         int va_bits = PGSHIFT + levels * ptidxbits;
(gdb) step
225         target_ulong mask = (1L << (TARGET_LONG_BITS - (va_bits - 1)))
   - 1;
(gdb) step
226         target_ulong masked_msbs = (addr >> (va_bits - 1)) & mask;
(gdb) step
227         if (masked_msbs != 0 && masked_msbs != mask) {
(gdb) step
231         int ptshift = (levels - 1) * ptidxbits;
```

这是在进行虚拟地址合法性检查，可以看到计算了 va_bits、masked_msbs，确保 Sv39 要求的高位符号扩展正确，否则抛异常。这里地址符合规范，所以继续。

ptshift = (levels - 1) * ptidxbits ，表示从 VPN[2] 开始索引，这是做了初始层级偏移，接下来逐级执行：

```
(gdb) step
237         for (i = 0; i < levels; i++, ptshift -= ptidxbits) {
(gdb) step
238             target_ulong idx = (addr >> (PGSHIFT + ptshift)) &
(gdb) step
239                                ((1 << ptidxbits) - 1);
(gdb) step
238             target_ulong idx = (addr >> (PGSHIFT + ptshift)) &
(gdb) step
242             target_ulong pte_addr = base + idx * ptesize;
(gdb) step
244             if (riscv_feature(env, RISCV_FEATURE_PMP) &&
...
245                 !pmp_hart_has_privs(env, pte_addr, sizeof(target_ulong)
   ,
...
```

```
(gdb) step
252             target_ulong pte = ldq_phys(cs->as, pte_addr);
...
```

可以看到三次逐级循环中做了如下操作：

- 计算索引：idx = (addr » (PGSHIFT + ptshift)) & 0x1ff。对每一级从虚拟地址中取 9 位索引。
- 计算当前 PTE 物理地址：pte_addr = base + idx * 8。
- PMP 检查：调用 pmp_hart_has_privs(env, pte_addr, 8, PMP_READ, mode=1)，遍历 PMP 条目确保访问页表内存合法，模拟硬件对页表访问的权限限制。
- 读取 PTE：ldq_phys(cs->as, pte_addr)，经过 address_space_translate 等，定位到物理 RAM 区域并读取 8 字节的页表项。内部 address_space_translate 找到 RAM 段（offset 0x8000_0000，长度 128MB），确认是在物理内存中读页表。这一步是真正在物理内存中取出页表项的动作。
- 权限/有效性判断（后续在循环里完成）：检查 PTE 的 V/R/W/X/U 等位，决定是叶子还是继续下级；权限不符则抛 fault。

完成三级循环后：组合叶子 PTE 中的 PPN 和虚拟地址的页内偏移，得到物理地址。对于这次取指，结果页帧是 0x80200000。

继续运行，会触发 tlb_set_page 处的断点：

```
(gdb) c
Continuing.

Thread 2 "qemu-system-ris" hit Breakpoint 3, tlb_set_page (
    cpu=0x59068d0da650, vaddr=18446744072637906944,
    paddr=2149580800, prot=7, mmu_idx=1, size=4096)
    at /home/nuo/qemu-4.1.1/accel/tcg/cputlb.c:847
847         tlb_set_page_with_attrs(cpu, vaddr, paddr,
   MEMTXATTRS_UNSPECIFIED,
(gdb) p/x vaddr
$3 = 0xffffffffc0200000
(gdb) p/x paddr
$4 = 0x80200000
(gdb) bt
#0  tlb_set_page (cpu=0x59068d0da650,
    vaddr=18446744072637906944, paddr=2149580800, prot=7,
    mmu_idx=1, size=4096)
    at /home/nuo/qemu-4.1.1/accel/tcg/cputlb.c:847
#1  0x000059066b19fc18 in riscv_cpu_tlb_fill (
    cs=0x59068d0da650, address=18446744072637907160,
    size=0, access_type=MMU_INST_FETCH, mmu_idx=1,
    probe=false, retaddr=0)
```

```
    at /home/nuo/qemu-4.1.1/target/riscv/cpu_helper.c:472
#2  0x000059066b0e5d7f in tlb_fill (cpu=0x59068d0da650,
    addr=18446744072637907160, size=0,
    access_type=MMU_INST_FETCH, mmu_idx=1, retaddr=0)
    at /home/nuo/qemu-4.1.1/accel/tcg/cputlb.c:878
#3  0x000059066b0e6429 in get_page_addr_code (
    env=0x59068d0e3060, addr=18446744072637907160)
    at /home/nuo/qemu-4.1.1/accel/tcg/cputlb.c:1032
#4  0x000059066b10ae8f in tb_htable_lookup (
    cpu=0x59068d0da650, pc=18446744072637907160,
    cs_base=0, flags=24577, cf_mask=4278714368)
    at /home/nuo/qemu-4.1.1/accel/tcg/cpu-exec.c:339
#5  0x000059066b103e65 in tb_lookup__cpu_state (
    cpu=0x59068d0da650, pc=0x77df77ffe3e0,
    cs_base=0x77df77ffe3d8, flags=0x77df77ffe3d4,
    cf_mask=4278714368)
    at /home/nuo/qemu-4.1.1/include/exec/tb-lookup.h:43
#6  0x000059066b10421e in helper_lookup_tb_ptr (
--Type <RET> for more, q to quit, c to continue without paging--c
    env=0x59068d0e3060)
    at /home/nuo/qemu-4.1.1/accel/tcg/tcg-runtime.c:154
#7  0x000077df7c134df7 in code_gen_buffer ()
#8  0x000059066b10a939 in cpu_tb_exec (cpu=0x59068d0da650,
    itb=0x77df7c134d00 <code_gen_buffer+330963>)
    at /home/nuo/qemu-4.1.1/accel/tcg/cpu-exec.c:173
...
```

可以看到此时触发 tlb_set_page 将翻译结果填入 TLB，被填入的软件 TLB 条目：vaddr=0xffffffffc0200000，paddr=0x80200000，prot=7（RWX），size=4096。这等价于硬件 TLB 中插入一条“虚拟页号 → 物理页帧号”映射，覆盖包含 kern_init 的 4KiB 页。后续同页取指会直接命中，不再进入 riscv_cpu_tlb_fill 或页表遍历。

此外，此时的调用栈显示 tlb_fill/get_page_addr_code/ tb_htable_lookup 等函数负责在指令译码和执行前取得可执行页的物理地址或触发翻译，行为与硬件 MMU 流程一致。

综合来看，这些步骤每一步都对应硬件 MMU 的核心动作：satp 解析、地址合法性检查、按级索引页表、权限/PMP 检查、物理读取 PTE、生成物理地址、写 TLB，形成了完整的虚拟 → 物理翻译链条。

3.6.3 思考回答

1. 这一全过程回答了“某条访存/取指虚拟地址如何在 QEMU 模拟中被翻译成物理地址”：虚拟地址 0xffffffffc02000d8 触发 TLB miss → riscv_cpu_tlb_fill → get_physical_address 三级页表遍历（以 satp 根 0x80205000 逐级索引、PMP/权限检查、物理读 PTE）→ 生成物理地址并在 tlb_set_page 填入软件 TLB → 返回执行。

2. 页表翻译的关键操作流程，在 get_physical_address 中可以得出：读取 satp 确定模式和根基址；按 SV39 的 3 级、每级 9 位索引逐级取 PTE；PMP/权限校验；无效或权限不符则走异常分支；成功则组合物理页帧号与页内偏移返回。tlb_set_page 则将结果缓存，减少后续页表遍历开销。

3. 查找 TLB 的 C 代码集中在 target/riscv/cpu_helper.c 和 accel/tcg/cputlb.c。riscv_cpu_tlb_fill 是 TLB 未命中后的入口，get_physical_address 负责多级页表遍历，tlb_set_page 在 cputlb.c 中将翻译结果填入 TLB。调用栈显示 tlb_fill/get_page_addr_code/ tb_htable_lookup 等函数负责在指令译码和执行前取得可执行页的物理地址或触发翻译，行为与硬件 MMU 流程一致。

4. 关于 QEMU 软件 TLB 与真实硬件 TLB 的差异：逻辑层面相同——命中则绕开页表，未命中则翻译并填表，区分读/写/取指权限，并支持不同特权级别；实现层面不同——硬件 TLB 是不可见的专用缓存（常见 CAM/组相联），大小固定，失效由硬件指令处理；QEMU TLB 是软件数据结构（数组/哈希），容量可配置，填充和失效完全在 C 代码中实现并可调试；在未开启分页时，QEMU 可走简化路径直接用物理地址。软件 TLB 可被断点、条件断点观察内部状态，这也是此次调试得以直观看到虚拟页号、物理页帧号、权限和页尺寸的原因。

3.7 lab5 分支任务：gdb 调试系统调用以及返回

3.7.1 任务简述

本次分支任务的目标，是使用双重 GDB（内核 gdbstub + 附加 QEMU 进程的宿主 GDB），完整观测一次用户态系统调用触发、陷入内核、再通过 sret 返回用户态的流程。

在内核 GDB 侧：加载用户程序符号、在用户态 syscall 的 ecall 前断点，单步执行到陷入；在内核入口、__trapret/sret 处断点，验证 CSR 与返回地址的变化。

在 QEMU 宿主 GDB 侧：针对 ecall/sret 下断点（trans_ecall/ helper_raise_exception/riscv_cpu_do_interrupt、trans_sret/helper_sret），阅读并跟踪源码，理解 TCG 翻译和 helper 如何模拟特权切换与异常处理。

理解 TCG 翻译机制：关键特权指令被翻译成 helper 调用并退出 TB，确保异常/返回即时生效，和之前地址翻译调试的思路类同。

3.7.2 调试与观察

首先，终端 1 用 make debug 启动了带 gdbstub 的 QEMU，随后终端 2 启动 gdb 并确保 target remote localhost:1234 连上 QEMU 的 gdbstub，然后用 add-symbol-file obj/__user_exit.out 正确加载用户态符号, 目的是能在用户态源码打断点。

```
(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) set remotetimeout unlimited
```

然后在 syscall 内联汇编处打断点，continue 执行后会停在内联汇编起始，表明断点已覆盖用户态 syscall 包装函数。此时，查看 pc 附近的指令，可以看到 ecall 在其后不远处，用 si 单步执行到 ecall 前停下：

```
(gdb) break 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 (
(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
(gdb) si
0x00000000008000fa      19          asm volatile (
(gdb) si
0x00000000008000fc      19          asm volatile (
(gdb) si
0x00000000008000fe      19          asm volatile (
(gdb) si
0x0000000000800100      19          asm volatile (
(gdb) si
0x0000000000800102      19          asm volatile (
(gdb) si
0x0000000000800104      19          asm volatile (
(gdb) x/2i $pc
=> 0x800104 <syscall+44>:       ecall
   0x800108 <syscall+48>:       sd      a0,28(sp)
```

此时打开终端 3，按照和前面 lab2 分支任务一样的操作打开附加 gdb，在 ecall 相关路径上设置断点：trans_ecall、helper_raise_exception、riscv_cpu_do_interrupt，然后 continue 继续：

```
(gdb) attach 26499
Attaching to process 26499
[New LWP 26501]
[New LWP 26500]
warning: could not find '.gnu_debugaltlink' file for /lib/x86_64-linux-gnu/
```

```
   libglib-2.0.so.0
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
0x00007d42a7b1ba30 in __GI_ppoll (fds=0x62c577dd1ad0,
    nfds=6, 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      Print   Pass to program Description
SIGPIPE       No        No      Yes             Broken pipe
(gdb) break trans_ecall
Breakpoint 1 at 0x62c56582a87b: file /home/nuo/qemu-4.1.1/target/riscv/
   insn_trans/trans_privileged.inc.c, line 24.
(gdb) break riscv_cpu_do_interrupt
Breakpoint 2 at 0x62c56582ec87: file /home/nuo/qemu-4.1.1/target/riscv/
   cpu_helper.c, line 507.
(gdb) break helper_raise_exception
Breakpoint 3 at 0x62c56582d072: file /home/nuo/qemu-4.1.1/target/riscv/
   op_helper.c, line 39.
(gdb) c
Continuing.
```

回到终端 2 单步执行 ecall，可以看到立即跳到 __alltraps（trapentry.S），说明 QEMU 将 ecall 译为异常并切换到 S 态陷入。此时会触发终端 3 的断点：

```
(gdb) si
0xffffffffc0200f4c in __alltraps () at kern/trap/trapentry.S:123
```

此时终端 3 触发了断点 trans_ecall，我们可以输入 bt 看一下调用栈并用 layout src 看一下此处的详细 qemu 源码：

```
(gdb) c
Continuing.
[Switching to Thread 0x7d42a51fe6c0 (LWP 26501)]

Thread 2 "qemu-system-ris" hit Breakpoint 1, trans_ecall (
    ctx=0x7d42a51fd7d0, a=0x7d42a51fd6d0)
    at /home/nuo/qemu-4.1.1/target/riscv/insn_trans/trans_privileged.inc.c
       :24
24          generate_exception(ctx, RISCV_EXCP_U_ECALL);
(gdb) bt
#0  trans_ecall (ctx=0x7d42a51fd7d0, a=0x7d42a51fd6d0)
    at /home/nuo/qemu-4.1.1/target/riscv/insn_trans/trans_privileged.inc.c
       :24
#1  0x000062c56582443f in decode_insn32 (
```

```
    ctx=0x7d42a51fd7d0, insn=115)
    at target/riscv/decode_insn32.inc.c:1614
#2  0x000062c56582ca87 in decode_opc (ctx=0x7d42a51fd7d0)
    at /home/nuo/qemu-4.1.1/target/riscv/translate.c:746
#3  0x000062c56582cc96 in riscv_tr_translate_insn (
    dcbase=0x7d42a51fd7d0, cpu=0x62c577d85650)
    at /home/nuo/qemu-4.1.1/target/riscv/translate.c:800
#4  0x000062c56579f5ae in translator_loop (
    ops=0x62c566082440 <riscv_tr_ops>, db=0x7d42a51fd7d0,
    cpu=0x62c577d85650,
    tb=0x7d42a51ff040 <code_gen_buffer+19>, max_insns=1)
    at /home/nuo/qemu-4.1.1/accel/tcg/translator.c:95
#5  0x000062c56582ce11 in gen_intermediate_code (
    cs=0x62c577d85650,
    tb=0x7d42a51ff040 <code_gen_buffer+19>, max_insns=1)
    at /home/nuo/qemu-4.1.1/target/riscv/translate.c:848
#6  0x000062c56579d99a in tb_gen_code (cpu=0x62c577d85650,
    pc=8388868, cs_base=0, flags=24576, cflags=-16252928)
    at /home/nuo/qemu-4.1.1/accel/tcg/translate-all.c:1738
#7  0x000062c56579a18f in tb_find (cpu=0x62c577d85650,
    last_tb=0x0, tb_exit=0, cf_mask=524288)
    at /home/nuo/qemu-4.1.1/accel/tcg/cpu-exec.c:409
#8  0x000062c56579aa89 in cpu_exec (cpu=0x62c577d85650)
    at /home/nuo/qemu-4.1.1/accel/tcg/cpu-exec.c:731
#9  0x000062c56574ccdf in tcg_cpu_exec (cpu=0x62c577d85650)
    at /home/nuo/qemu-4.1.1/cpus.c:1435
#10 0x000062c56574d598 in qemu_tcg_cpu_thread_fn (
    arg=0x62c577d85650) at /home/nuo/qemu-4.1.1/cpus.c:1743
--Type <RET> for more, q to quit, c to continue without paging--q
Quit
(gdb) layout src
【此处会看到：
static bool trans_ecall(DisasContext *ctx, arg_ecall *a)
{
    /* always generates U-level ECALL, fixed in do_interrupt handler */
    generate_exception(ctx, RISCV_EXCP_U_ECALL);
    exit_tb(ctx); /* no chaining */
    ctx->base.is_jmp = DISAS_NORETURN;
    return true;
}
】
```

根据 bt 显示的回溯的调用栈来看，是由 translator_loop → decode_insn32 → trans_ecall 进入 ecall 的翻译；观察 trans_ecall 源代码，其参数 DisasContext *ctx 是当前 TB 的翻译上下文，arg_ecall *a 是解析出的指令参数（ecall 无操作数字段）；generate_exception(ctx, RISCV_EXCP_U_ECALL);

会生成触发 “U 态 ecall 异常” 的 TCG helper 调用，先按 U 态标记，最终由异常处理根据当前特权级映射；exit_tb(ctx); ctx->base.is_jmp = DISAS_NORETURN; 则会强制当前 TB 翻译到此为止，不再链到后续指令，保证异常立即生效。总结来看，ecall 不真正执行功能，硬件语义是 “触发同步异常”。QEMU 将其翻译成一个 helper，退出 TB，把控制权交给异常分发流程。

然后继续 continue，会触发 helper_raise_exception 断点：

```
(gdb) c
Continuing.

Thread 2 "qemu-system-ris" hit Breakpoint 3, helper_raise_exception (env=0
    x62c577d8e060, exception=8)
    at /home/nuo/qemu-4.1.1/target/riscv/op_helper.c:39
39          riscv_raise_exception(env, exception, 0);
```

helper_raise_exception 可以看作是 ecall 的执行阶段，它在 TCG 生成的异常 helper 中调用，用来把异常写入 QEMU 的 CPU 状态并中断当前 TB 执行。关键逻辑是调用 riscv_raise_exception(env, exception, 0); 设置 cs->exception_index 等内部字段，标记当前 CPU 需要进入异常处理。它不直接修改 PC/特权级，它只是把 “有异常” 这个事实通知给执行核心，随后 cpu_exec 会跳转到 riscv_cpu_do_interrupt 处理。

继续 continue，会触发 riscv_cpu_do_interrupt 断点：

```
(gdb) c
Continuing.

Thread 2 "qemu-system-ris" hit Breakpoint 2, riscv_cpu_do_interrupt (cs=0
    x62c577d85650)
    at /home/nuo/qemu-4.1.1/target/riscv/cpu_helper.c:507
507         RISCVCPU *cpu = RISCV_CPU(cs);
(gdb) p/x ((CPURISCVState*)cpu)->sepc
$1 = 0x8b4825ebf0458948
(gdb) p/x ((CPURISCVState*)cpu)->scause
$2 = 0xff80e8c78948f045
(gdb) layout src
【此处会看到：
void riscv_cpu_do_interrupt(CPUState *cs)
{
#if !defined(CONFIG_USER_ONLY)

    RISCVCPU *cpu = RISCV_CPU(cs);
    CPURISCVState *env = &cpu->env;

    /* cs->exception is 32-bits wide unlike mcause which is XLEN-bits wide
     * so we mask off the MSB and separate into trap type and cause.
     */
    bool async = !!(cs->exception_index & RISCV_EXCP_INT_FLAG);
```

```
24      target_ulong cause = cs->exception_index & RISCV_EXCP_INT_MASK;
25      target_ulong deleg = async ? env->mideleg : env->medeleg;
26      target_ulong tval = 0;
27
28      static const int ecall_cause_map[] = {
29          [PRV_U] = RISCV_EXCP_U_ECALL,
30          [PRV_S] = RISCV_EXCP_S_ECALL,
31          [PRV_H] = RISCV_EXCP_H_ECALL,
32          [PRV_M] = RISCV_EXCP_M_ECALL
33      };
34
35      if (!async) {
36          /* set tval to badaddr for traps with address information */
37          switch (cause) {
38          case RISCV_EXCP_INST_ADDR_MIS:
39          case RISCV_EXCP_INST_ACCESS_FAULT:
40          case RISCV_EXCP_LOAD_ADDR_MIS:
41          case RISCV_EXCP_STORE_AMO_ADDR_MIS:
42          case RISCV_EXCP_LOAD_ACCESS_FAULT:
43          case RISCV_EXCP_STORE_AMO_ACCESS_FAULT:
44          case RISCV_EXCP_INST_PAGE_FAULT:
45          case RISCV_EXCP_LOAD_PAGE_FAULT:
46          case RISCV_EXCP_STORE_PAGE_FAULT:
47              tval = env->badaddr;
48              break;
49          default:
50              break;
51          }
52          /* ecall is dispatched as one cause so translate based on mode */
53          if (cause == RISCV_EXCP_U_ECALL) {
54              assert(env->priv <= 3);
55              cause = ecall_cause_map[env->priv];
56          }
57      }
58
59      trace_riscv_trap(env->mhartid, async, cause, env->pc, tval, cause < 16
            ?
60          (async ? riscv_intr_names : riscv_excp_names)[cause] : "(unknown)")
                ;
61
62      if (env->priv <= PRV_S &&
63              cause < TARGET_LONG_BITS && ((deleg >> cause) & 1)) {
64          /* handle the trap in S-mode */
65          target_ulong s = env->mstatus;
66          s = set_field(s, MSTATUS_SPIE, env->priv_ver >= PRIV_VERSION_1_10_0
```

```
            ?
67              get_field(s, MSTATUS_SIE) : get_field(s, MSTATUS_UIE << env->
                    priv));
68          s = set_field(s, MSTATUS_SPP, env->priv);
69          s = set_field(s, MSTATUS_SIE, 0);
70          env->mstatus = s;
71          env->scause = cause | ((target_ulong)async << (TARGET_LONG_BITS -
                1));
72          env->sepc = env->pc;
73          env->sbadaddr = tval;
74          env->pc = (env->stvec >> 2 << 2) +
75              ((async && (env->stvec & 3) == 1) ? cause * 4 : 0);
76          riscv_cpu_set_mode(env, PRV_S);
77      } else {
78          /* handle the trap in M-mode */
79          target_ulong s = env->mstatus;
80          s = set_field(s, MSTATUS_MPIE, env->priv_ver >= PRIV_VERSION_1_10_0
                ?
81              get_field(s, MSTATUS_MIE) : get_field(s, MSTATUS_UIE << env->
                    priv));
82          s = set_field(s, MSTATUS_MPP, env->priv);
83          s = set_field(s, MSTATUS_MIE, 0);
84          env->mstatus = s;
85          env->mcause = cause | ~(((target_ulong)-1) >> async);
86          env->mepc = env->pc;
87          env->mbadaddr = tval;
88          env->pc = (env->mtvec >> 2 << 2) +
89              ((async && (env->mtvec & 3) == 1) ? cause * 4 : 0);
90          riscv_cpu_set_mode(env, PRV_M);
91      }
92
93      /* NOTE: it is not necessary to yield load reservations here. It is
            only
94       * necessary for an SC from "another hart" to cause a load reservation
95       * to be yielded. Refer to the memory consistency model section of the
96       * RISC-V ISA Specification.
97       */
98
99  #endif
100     cs->exception_index = EXCP_NONE; /* mark handled to qemu */
101 }
102 】
```

同样查看 riscv_cpu_do_interrupt 的源代码，其输入为 CPUState *cs，内部获取 RISCVCPU *cpu、env。异常信息来自 cs->exception_index（helper_raise_exception 写入）。关键逻辑为：

1. 判断 async（是否中断）。ecall 为同步，async=0。
2. 提取 cause = cs->exception_index & RISCV_EXCP_INT_MASK。若是 ecall（初始 U ecall），通过 ecall_cause_map[env->priv] 根据当前特权级映射到 U/S/M ecall。
3. 计算委派 deleg = async ? mideleg : medeleg，决定交给 S 还是 M 处理。Lab 中在 U 态且 medeleg/stedeleg 允许，走 S 态路径。
4. 生成 tval（地址类异常填 badaddr；ecall 不填）。
5. 更新 CSR 和模式。
6. 清 cs->exception_index = EXCP_NONE，表示异常已被处理。

总结来看，它模拟的就是硬件的异常入口：保存返回地址/原因、更新中断使能位、切换特权级、跳到陷阱入口，随后继续执行的就是内核的 __alltraps/trap。

随后继续 continue，ecall 执行完毕会回到终端 2，按照类似的方法在 sret 前打下断点然后执行到此处：

```
(gdb) break kern/trap/trapentry.S:132
Breakpoint 4 at 0xffffffffc020100e: file kern/trap/trapentry.S, line 133.
(gdb) c
Continuing.

Breakpoint 4, __trapret () at kern/trap/trapentry.S:133
133         sret
(gdb) x/4i $pc
=> 0xffffffffc020100e <__trapret+86>:   sret
   0xffffffffc0201012 <forkrets>:       mv      sp,a0
   0xffffffffc0201014 <forkrets+2>:     j       0xffffffffc0200fb8 <
      __trapret>
   0xffffffffc0201016 <kernel_execve_ret>:      addi    a1,a1,-288
(gdb) info registers sepc sstatus
sepc            0x800108 8388872
sstatus         0x8000000000046020       -9223372036854489056
```

可以看到 sret 即将执行。我们在此处执行 info registers sepc sstatus 检查 CSR，可以看到 sepc=0x800108 正好是用户态 ecall 后的下一条，sstatus 中 SPP=0，说明将返回 U 态且 SPIE 已设置。

然后我们先不执行，先在终端 3 中打下 sret 相关路径上的断点：

```
(gdb) c
Continuing.

^C
Thread 1 "qemu-system-ris" received signal SIGINT, Interrupt.
[Switching to Thread 0x7d42a7cbcf40 (LWP 26499)]
0x00007d42a7b1ba30 in __GI_ppoll (fds=0x62c577dd1ad0,
```

```
    nfds=6, 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) break trans_sret
Breakpoint 4 at 0x62c56582a918: file /home/nuo/qemu-4.1.1/target/riscv/
    insn_trans/trans_privileged.inc.c, line 46.
(gdb) break helper_sret
Breakpoint 5 at 0x62c56582d256: file /home/nuo/qemu-4.1.1/target/riscv/
    op_helper.c, line 76.
(gdb) c
Continuing.
```

然后再在终端 2 单步执行 sret，触发终端 3 设置的断点：

```
(gdb) si
0x0000000000800108 in syscall (num=0, num@entry=30) at user/libs/syscall.c
    :19
19          asm volatile (
```

这时候我们关注终端 3，可以看到触发了 trans_sret 断点，看一下此处的调用栈和该函数的源码：

```
(gdb) c
Continuing.
[Switching to Thread 0x7d42a51fe6c0 (LWP 26501)]

Thread 2 "qemu-system-ris" hit Breakpoint 4, trans_sret (
    ctx=0x7d42a51fd7d0, a=0x7d42a51fd6d0)
    at /home/nuo/qemu-4.1.1/target/riscv/insn_trans/trans_privileged.inc.c
        :46
46          tcg_gen_movi_tl(cpu_pc, ctx->base.pc_next);
(gdb) bt
#0  trans_sret (ctx=0x7d42a51fd7d0, a=0x7d42a51fd6d0)
    at /home/nuo/qemu-4.1.1/target/riscv/insn_trans/trans_privileged.inc.c
        :46
#1  0x000062c565824517 in decode_insn32 (
    ctx=0x7d42a51fd7d0, insn=270532723)
    at target/riscv/decode_insn32.inc.c:1638
#2  0x000062c56582ca87 in decode_opc (ctx=0x7d42a51fd7d0)
    at /home/nuo/qemu-4.1.1/target/riscv/translate.c:746
#3  0x000062c56582cc96 in riscv_tr_translate_insn (
    dcbase=0x7d42a51fd7d0, cpu=0x62c577d85650)
    at /home/nuo/qemu-4.1.1/target/riscv/translate.c:800
#4  0x000062c56579f5ae in translator_loop (
    ops=0x62c566082440 <riscv_tr_ops>, db=0x7d42a51fd7d0,
    cpu=0x62c577d85650,
    tb=0x7d42a51ff040 <code_gen_buffer+19>, max_insns=1)
```

```
    at /home/nuo/qemu-4.1.1/accel/tcg/translator.c:95
#5  0x000062c56582ce11 in gen_intermediate_code (
    cs=0x62c577d85650,
    tb=0x7d42a51ff040 <code_gen_buffer+19>, max_insns=1)
    at /home/nuo/qemu-4.1.1/target/riscv/translate.c:848
#6  0x000062c56579d99a in tb_gen_code (cpu=0x62c577d85650,
    pc=18446744072637911054, cs_base=0, flags=24577,
    cflags=-16252928)
    at /home/nuo/qemu-4.1.1/accel/tcg/translate-all.c:1738
#7  0x000062c56579a18f in tb_find (cpu=0x62c577d85650,
    last_tb=0x0, tb_exit=0, cf_mask=524288)
    at /home/nuo/qemu-4.1.1/accel/tcg/cpu-exec.c:409
#8  0x000062c56579aa89 in cpu_exec (cpu=0x62c577d85650)
    at /home/nuo/qemu-4.1.1/accel/tcg/cpu-exec.c:731
#9  0x000062c56574ccdf in tcg_cpu_exec (cpu=0x62c577d85650)
    at /home/nuo/qemu-4.1.1/cpus.c:1435
#10 0x000062c56574d598 in qemu_tcg_cpu_thread_fn (

(gdb) layout src
【此处会看到：
static bool trans_sret(DisasContext *ctx, arg_sret *a)
{
#ifndef CONFIG_USER_ONLY
    tcg_gen_movi_tl(cpu_pc, ctx->base.pc_next);

    if (has_ext(ctx, RVS)) {
        gen_helper_sret(cpu_pc, cpu_env, cpu_pc);
        exit_tb(ctx); /* no chaining */
        ctx->base.is_jmp = DISAS_NORETURN;
    } else {
        return false;
    }
    return true;
#else
    return false;
#endif
}
】
```

bt 显示翻译阶段生成 gen_helper_sret 并 exit_tb。观察源代码，trans_sret 在 TCG 翻译 sret 时生成调用 helper_sret 的代码，并强制结束当前 TB，不再链到后续指令。其关键操作是：先 tcg_gen_movi_tl(cpu_pc, ctx->base.pc_next); 把下一条 PC（用于调试/回填）写入 TCG 临时; 检查是否有 S 扩展，若有则 gen_helper_sret(cpu_pc, cpu_env, cpu_pc); exit_tb(ctx); ctx->base.is_jmp = DISAS_NORETURN; 则意味着本 TB 到此结束，控制流交给 helper_sret 处理特权切换和 PC 更新；若没

有 S 扩展直接返回 false，表示非法指令。

可以总结，sret 涉及特权级切换、CSR 更新，必须通过 helper 在执行阶段完成，并立即跳出 TB，确保状态变化立刻生效。

继续 continue，可以看到终端 3 又触发了 helper_sret 断点，同样用 layout src 看一下源代码：

```
(gdb) c
Continuing.

Thread 2 "qemu-system-ris" hit Breakpoint 5, helper_sret (
    env=0x62c577d8e060, cpu_pc_deb=18446744072637911054)
    at /home/nuo/qemu-4.1.1/target/riscv/op_helper.c:76
76          if (!(env->priv >= PRV_S)) {
(gdb) layout src
【此处会看到：
target_ulong helper_sret(CPURISCVState *env, target_ulong cpu_pc_deb)
{
    if (!(env->priv >= PRV_S)) {
        riscv_raise_exception(env, RISCV_EXCP_ILLEGAL_INST, GETPC());
    }

    target_ulong retpc = env->sepc;
    if (!riscv_has_ext(env, RVC) && (retpc & 0x3)) {
        riscv_raise_exception(env, RISCV_EXCP_INST_ADDR_MIS, GETPC());
    }

    if (env->priv_ver >= PRIV_VERSION_1_10_0 &&
        get_field(env->mstatus, MSTATUS_TSR)) {
        riscv_raise_exception(env, RISCV_EXCP_ILLEGAL_INST, GETPC());
    }

    target_ulong mstatus = env->mstatus;
    target_ulong prev_priv = get_field(mstatus, MSTATUS_SPP);
    mstatus = set_field(mstatus,
        env->priv_ver >= PRIV_VERSION_1_10_0 ?
        MSTATUS_SIE : MSTATUS_UIE << prev_priv,
        get_field(mstatus, MSTATUS_SPIE));
    mstatus = set_field(mstatus, MSTATUS_SPIE, 0);
    mstatus = set_field(mstatus, MSTATUS_SPP, PRV_U);
    riscv_cpu_set_mode(env, prev_priv);
    env->mstatus = mstatus;

    return retpc;
}
】
```

helper_sret 按 RISC-V 规范执行 sret：合法性检查、恢复 PC/特权级、中断状态。其关键步骤为：

1. 检查当前特权 env->priv >= PRV_S，否则抛非法指令异常。
2. retpc = env->sepc，若不支持压缩且 retpc 未对齐，则抛取址异常。
3. 若 mstatus.TSR 置位（禁止 sret），抛非法指令。
4. 读 mstatus，prev_priv = SPP；将 SIE 设为 SPIE，清 SPIE，将 SPP 置为 U；调用 riscv_cpu_set_mode(env,prev_priv) 切换特权级；写回更新后的 mstatus。
5. 返回 retpc，TCG 用这个返回值作为新的 PC。

可以总结，它模拟硬件 sret 的语义：用 sepc 作为返回地址，恢复中断使能位，清 SPP，切换回保存的特权级，实现从陷入返回用户态。

4 实验所遇问题

4.1 问题 1

问题描述：`make grade` 评分脚本全部输出 `no $qemu_out`，导致无法通过测试。

原因分析：评分脚本 `tools/grade.sh` 仍沿用老版本的 QEMU 启动方式，没有设置 `set architecture riscv:rv64`，导致 GDB 无法连接，QEMU 在 -S 状态下挂起，输出文件为空。

解决办法：修改 `tools/grade.sh`（以及 Makefile 保证 QEMU 路径正确），在生成 gdb 脚本时插入 `set architecture riscv:rv64`，并把 `qemuopts` 改成 `-machine virt -m 256M -bios default -kernel bin/ucore.img`。

4.2 问题 2

问题描述：`make run-nox-cow` 和 `make run-nox-dirtycow` 无法生成日志文件。

原因分析：Master 分支缺少 `user/cow.c` 等测试文件，导致链接时报符号未定义错误。

解决办法：从 lab5 分支将 `user/cow.c` 复制回 master 分支并纳入构建系统。

4.3 问题 3

问题描述：COW 改造后，forktest 测试出现 panic (Instruction page fault)。

原因分析：`trap.c` 在 `CAUSE_LOAD_PAGE_FAULT` 和 `CAUSE_STORE_PAGE_FAULT` 分支中不再调用 `do_pgfault()`，导致写保护页一直触发 fault 却无法处理。

解决办法：恢复 `do_pgfault` 的调用并正确处理返回值，使 COW 的写缺页能正常触发复制流程。

4.4 问题 4

问题描述：Dirty COW 演示模式在 Bug/Fix 切换时行为不符合预期。

原因分析：Bug 模式下需要绕过 COW 拆页才能复现漏洞，而 Fix 模式下必须强制触发 `do_pgfault`。

解决办法:在 kern/mm/cow.c 中实现 dirtycow_prepare_page(),根据全局标志 dirtycow_stats.emulate_bug 决定是否调用 do_pgfault()。