操作系统实验五

小组成员：何叶2313487 乔煜2312221 朱子昂2311283

练习0：填写已有实验

本实验依赖实验2/3/4。请把你做的实验2/3/4的代码填入本实验中代码中有“LAB2”/“LAB3”/“LAB4”的注释相应部分。注意：为了能够正确执行 `lab5` 的测试应用程序，可能需对已完成的实验2/3/4的代码进行进一步改进。

练习1: 加载应用程序并执行（需要编码）

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。

回答：

一、设计过程

1.设置用户栈顶 `tf->gpr.sp = USTACKTOP`

用户程序依赖栈进行函数调用和局部变量存储；

USTACKTOP 是预定义的高地址（如 0x7FC00000），作为用户栈的起始栈顶；

确保进程一进入用户态就有合法、可用的栈空间。

2.设置程序入口 `tf->epc = elf->e_entry`

e_entry 是 ELF 文件头中指定的程序入口地址（如 _start）；

sret 指令会从 sepc（即 tf->epc）开始取指执行；

实现“跳转到用户程序第一条指令”的控制权转移。

3.配置特权级与中断状态 `tf->status = ...`

清除 SPP 位（~SSTATUS_SPP）：指示 sret 后切换到 用户态（U-mode）；

设置 SPIE = 1：使 sret 后 开启中断（SIE=1），保证系统可响应时钟、I/O 等事件；

保留其他状态位（如 SIE 原值被覆盖，但通过 SPIE 控制恢复逻辑）。

二、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

首先是调度器选中下一个进程指定为这个应用程序，然后调用proc_run将current进程指向新的进程，并把页表切换到当前页表

并用switch_to切换旧的到新的上下文，此时cpu就运行在了新的进程的内核栈上了，但是仍然处于之前老进程遗留的supervisor模式，在执行

```
__trapret();            // 进入 kern/trap/trapentry.S 中的通用返回用户态代码
```

__trapret() 汇编序列

在这段内联汇编中先从新的trapframe中回复32个通用寄存器，然后将内核 的栈指针指向用户分配的栈顶，csrw sstatus, t0 → 恢复 sstatus（SPP=0, SPIE=1, FS=initial/clean），这时将epc指向新用户进程的 elf程序程序中的entry第一条指令开始执行，特权级：Supervisor → User。此时把SPIE的值赋回给PIE，恢复中断，地址 = elf->e_entry → 这就是用户程序的真正第一条指令（通常是程序入口的 _start 或编译器生成的 CRT 代码）。

流程：

1.创建用户进程：

init_main() 中通过 kernel_thread(user_main, ...) 创建一个内核线程，执行 user_main()。

2.触发 exec 系统调用：

user_main() 调用 kernel_execve("exit", ...)，通过内联汇编执行 ebreak 指令，主动陷入内核（产生断点异常）。

3.进入 trap 处理流程：

CPU 跳转到 __alltraps → trap() → trap_dispatch()，识别为 CAUSE_BREAKPOINT，进而调用 syscall()。

4.分发系统调用：

syscall() 根据系统调用号（SYS_exec）调用 sys_exec()，后者调用 do_execve()。

5.加载用户程序：

do_execve() 释放原内存空间（如有），调用 load_icode() 将 exit 程序的二进制代码加载到用户地址空间，并设置好用户栈和入口点。

6.返回用户态执行：

系统调用逐层返回至 __trapret，执行 RESTORE_ALL 恢复寄存器（包括将 PC 指向用户程序入口），再通过 sret 从内核态切换回用户态，开始执行应用程序的第一条指令。

练习2: 父进程复制自己的内存空间给子进程（需要编码）

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。

Copy-on-write（简称COW）的基本概念是指如果有多个使用者对一个资源A（比如内存块）进行读操作，则每个使用者只需获得一个指向同一个资源A的指针，就可以该资源了。若某使用者需要对这个资源A进行写操作，系统会对该资源进行拷贝操作，从而使得该“写操作”使用者获得一个该资源A的“私有”拷贝—资源B，可对资源B进行写操作。该“写操作”使用者对资源B的改变对于其他的使用者而言是不可见的，因为其他使用者看到的还是资源A。

回答：

在实现COW机制时，关键是复制内存页表的时机，在dofork阶段不再直接复制页表，而是增加一个指针指向这个物理地址空间，在真正遇到了需要写的时候再去复制页，在这之前**父子页表都清写权限（只读）并增加页框引用计数。**具体设计如下：

1. copy_range()（或 fork 时页表复制逻辑）：建立共享映射，清写权限，增加引用计数（不 alloc 不 memcpy）。
2. 页框管理（struct Page）支持引用计数：page_ref_inc(page), page_ref_dec(page)，并在计数到 0 时释放物理页。
3. 页表操作（page_insert/page_remove）正确处理引用计数与 PTE 权限变更。

4. 页错误处理（page fault handler）：检测写缺页且为 COW 页，执行按需复制或直接恢复写权限（当引用计数 == 1）。
5. TLB flush（tlb_invalidate）与同步。
6. （可选）支持 MAP_SHARED 的不同语义：shared mapping 不走 COW（或按策略）。
7. 代码实现示例：

```
int copy_range_cow(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end,
bool share)
{
    assert(start % PGSIZE == 0 && end % PGSIZE == 0);
    assert(USER_ACCESS(start, end));
    do {
        pte_t *ptep = get_pte(from, start, 0);
        pte_t *nptep;
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue;
        }
        if (*ptep & PTE_V) {
            if ((nptep = get_pte(to, start, 1)) == NULL)
                return -E_NO_MEM;
            uint32_t perm = (*ptep & PTE_USER);
            struct Page *page = pte2page(*ptep);
            assert(page != NULL);

            // 1) make both parent and child read-only (clear write bit)
            uint32_t new_perm = perm & ~PTE_W;

            // 2) map child to the same physical page (do NOT alloc or
memcpy)
            int ret = page_insert(to, page, start, new_perm);
            if (ret < 0) return ret;

            // 3) update parent's pte to be read-only if it was writable
            if ((*ptep & PTE_W) != 0) {
                *ptep = (*ptep & ~PTE_W); // 保留 PTE_V, U 等位
                // 如果需要维护硬件页表同步，可能调用 mmu更新函数
            }

            // 4) increase refcount
            page_ref_inc(page);
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

8. 在写时遇到页错误的处理，就复制页再执行写：

```
int handle_cow_fault(pde_t *pgdir, uintptr_t addr, int fault_write)
{
    pte_t *ptep = get_pte(pgdir, addr, 0);
    if (ptep == NULL || !(*ptep & PTE_V)) return -E_INVAL;

    // 不是写导致的缺页，或页当前就可写，绕过
    if (!fault_write) return -E_INVAL;
```

```
        // 如果该页有写权限，那可能不是 COW（直接错误或其它）
        if ((*ptep & PTE_W) != 0) return -E_INVAL;

        struct Page *old = pte2page(*ptep);
        if (old == NULL) return -E_INVAL;

        // 原子读取引用计数（注意并发）
        if (page_ref_count(old) > 1) {
            // 多个引用：实际复制
            struct Page *new = alloc_page();
            if (!new) return -E_NO_MEM;
            memcpy(page2kva(new), page2kva(old), PGSIZE);

            // 更新页表：把当前页表项替换为 new（带写权限）
            int ret = page_insert(pgdir, new, addr, ( (*ptep & PTE_USER) |
PTE_W ));
            if (ret < 0) {
                free_page(new); // 或 dec ref 并释放
                return ret;
            }
            page_ref_dec(old); // old 引用减少
        } else {
            // 只有 1 个引用（即当前进程独占），可以直接把 PTE 标成可写
            *ptep |= PTE_W;
            // 如果有需要，刷新 mmu 或 tlb
        }
        tlb_invalidate(addr);
        return 0;
}
```

练习3: 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？

回答：

一、函数分析

1.do_fork

中的return syscall(SYS_fork);开始进入内核态，然后分配一个新的进程控制块，并按照传统复制，将父进程的内核栈和上下文的trapframe复制给子进程，让子进程从同一位置和状态恢复执行，处理内存也是按照copyrange全复制，然后把父子进程写进不同的trap frame，父进程的fork返回子进程的pid，然后返回用户态，内核将要运行哪一个进程交给调度器恢复 trapframe → 回到用户态。

2.do_exec

用户程序执行：

```
execve("bin/sh", argv, envp);
```

然后陷入内核态，清空旧地址空间（摧毁旧的代码/数据/栈）从 ELF 加载程序，加载 .text、.data、堆栈，建立新的 VMA（虚拟内存区域）构造新的用户栈（含参数与环境变量），重置 trapframe 的 PC = 程序入口点。恢复到用户态内核执行 `return-from-trap`，跳到新程序的入口，用户进程开始运行完全新的一套指令。

3.do_wait

用户态

```
pid = wait(&status);
```

陷入内核态，检查是否有已退出的子进程

如果没有：**挂起**当前进程（sleep/wait_queue）

如果有：

回收子进程资源（free address space、内核栈）

将退出码写入父进程用户态地址空间中的 `status`

返回用户态

把子进程的 pid 作为返回值写入 trapframe

用户态 wait() 返回

4.do_exit

用户态 exit(code)

陷入内核态

内核做：关闭所有文件、回收内存空间

1. 保存退出码到 PCB
2. 将进程标记为 ZOMBIE，等待父进程 wait 回收
3. 唤醒父进程（如果父在 wait)

但是exit不会返回用户态，直接进入调度器切换到其他进程。

二、执行流程

用户态通过 `ecall`/`ebreak` 发起系统调用，陷入内核。

内核在 `trap_dispatch` 中识别系统调用号，调用 `syscall()` 分发至对应处理函数（如 `sys_fork`、`sys_exec` 等）。

`fork`：复制进程结构、内核栈、页表（深拷贝或 COW）、寄存器上下文（子进程返回 0，父进程返回子 PID）。

`exec`：释放旧地址空间，加载新程序（ELF），建立新页表和用户栈，设置 trapframe 的 PC（入口点）和 SP（栈顶），为返回用户态做准备。

`wait/exit`：`exit` 将进程置为僵尸态并唤醒父进程；`wait` 若无退出子进程则阻塞父进程，待子退出后回收其资源。

三、用户态和内核态分工

1. **用户态**：准备参数、触发陷入、接收返回值。
2. **内核态**：完成所有资源管理（内存、进程结构、页表）、调度、同步及返回值设置。

四、内核与用户态切换机制

1. 陷入时由 `SAVE_ALL` 保存用户上下文到 `trapframe`；
2. 内核处理完后修改 `trapframe->gpr.a0` 作为返回值；
3. 通过 `RESTORE_ALL` + `sret` 恢复寄存器并切回用户态，用户程序从系统调用下一条指令继续执行。

五、用户态进程的执行状态生命周期图

扩展练习 Challenge

一、实现 Copy on Write （COW）机制

给出实现源码,测试用例和设计报告（包括在cow情况下的各种状态转换（类似有限状态自动机）的说明）。

这个扩展练习涉及到本实验和上一个实验“虚拟内存管理”。在ucore操作系统中，当一个用户父进程创建自己的子进程时，父进程会把其申请的用户空间设置为只读，子进程可共享父进程占用的用户内存空间中的页面（这就是一个共享的资源）。当其中任何一个进程修改此用户内存空间中的某页面时，ucore会通过page fault异常获知该操作，并完成拷贝内存页面，使得两个进程都有各自的内存页面。这样一个进程所做的修改不会被另外一个进程可见了。请在ucore中实现这样的COW机制。

由于COW实现比较复杂，容易引入bug，请参考 https://dirtycow.ninja/ 看看能否在ucore的COW实现中模拟这个错误和解决方案。需要有解释。

这是一个big challenge.

1.COW 机制设计概述

a.核心思想

当父进程调用 fork() 创建子进程时，不立即复制用户地址空间的物理页，而是让父子进程共享相同的物理页，并将这些页在各自的页表中设为 只读（Read-Only），同时标记为“COW 页”。

读操作：正常执行，无异常。

写操作：触发 Page Fault（缺页异常），内核捕获后：

分配新物理页；

复制原页内容；

更新当前进程的页表项指向新页，并设为可写；

恢复执行，写操作成功。

这样，只有真正发生写入的页面才会被复制，显著节省内存和启动时间。

b.关键数据结构支持

物理页引用计数：每个物理页维护一个引用计数，记录有多少进程共享该页。

COW 标记：在页表项（PTE）中利用保留位标记该页是否为 COW 页，用于区分普通只读页与 COW 页。

c.状态转换（有限状态机）

每个用户物理页在其生命周期中经历以下状态：

当前状态	触发事件	动作	新状态
Shared (RO, COW)	进程读访问	允许访问	保持 Shared
Shared (RO, COW)	进程写访问	触发 Page Fault → 执行 COW 拷贝	Private (RW)
Private (RW)	任意访问	正常读写	保持 Private
Shared (RO, COW)	进程退出	引用计数减 1；若为 0 则释放物理页	—

2.Dirty COW 漏洞的模拟与分析

a.在 ucore 中的模拟场景

假设 ucore 的 COW 实现未对 Page Fault 处理加锁，则可构造如下攻击：

父进程映射一个只读文件（或初始化数据段）到用户空间；

调用 fork()，子进程与父进程共享该页（COW 状态）；

子进程中启动两个线程：

Thread A：

不断尝试写该页（触发 Page Fault，进入 COW 处理流程）；

Thread B：

在 Thread A 执行 COW 拷贝的中间阶段（已判断需拷贝但尚未更新页表），通过某种方式（如直接内存写，或利用调试接口）尝试写同一地址。

由于缺乏同步，内核可能：

- 错误地认为该页已变为可写；
- 或重复处理导致页表项被错误设置为可写；
- 最终使 Thread B 直接写入原始共享页，破坏父进程或其他子进程的数据。

c.根本原因

- Page Fault 处理不是原子操作；
- COW 状态检查与页表更新之间存在竞态窗口；
- 缺乏对“同一虚拟地址并发访问”的互斥保护。

3.解决方案

a.加锁保护关键路径

在处理 Page Fault 时，对涉及的虚拟内存区域（VMA）或整个地址空间（mm_struct）加锁，确保：

同一时间只有一个线程能处理该地址的缺页；

COW 的“检查 → 分配 → 复制 → 更新页表”全过程原子执行。

b.二次验证（Double-Check）

在更新页表前，再次检查：

该页是否仍为 COW 页；

引用计数是否仍大于 1； 防止在等待分配内存期间状态已被其他线程改变。

c.设计原则

最小临界区：锁粒度尽可能细（如 per-VMA 锁），避免性能瓶颈；

死锁预防：锁顺序固定，避免嵌套死锁；

失败回滚：若分配失败，恢复原状态，不留下中间状态。

二、说明该用户程序是何时被预先加载到内存中的？与我们常用操作系统的加载有何区别，原因是什么?

1.ucore 采用一次性加载方式：

在 do_execve 中通过 load_icode 解析 ELF 文件，为所有 PT_LOAD 段（代码、数据、BSS）立即分配物理页、建立页表映射，并将内容从内核中的二进制镜像复制到用户空间，同时设置用户栈和 trapframe。程序执行前，全部代码和数据已驻留内存。该方法实现简单，无需处理缺页异常，适合教学和资源受限环境，但内存开销大，可能加载未使用代码。

2.现代操作系统（如 Linux）采用懒加载（Lazy Loading）：

execve 仅建立虚拟内存布局，不加载实际内容；当访问未加载页面时触发缺页异常，内核再按需从磁盘读入对应页。这种方式节省内存、加快启动速度、提高系统整体效率，适用于通用场景。

分支实验

lab5分支：gdb 调试系统调用以及返回

一、U态触发系统调用到S态处理流程简述

1.U态发起系统调用：用户程序通过 ecall 指令触发陷入（trap），进入内核（S态）。

2.陷入处理：硬件自动切换到 S 态，跳转到 trap 入口（如 __alltraps），保存上下文。

3.trap 处理：trap handler 检查 trap 原因（如 scause），如果是系统调用，则分发到对应的 syscall 处理函数。

4.系统调用处理：内核完成系统调用逻辑，准备返回值。

5.trap 返回：恢复上下文，执行 sret，返回 U 态，用户程序继续执行。

二、实验过程

1.启动qemu和gdb调试。

```
heye@DESKTOP-63A0U9G:~/qemu-4.1.1$
 sudo gdb
[sudo] password for heye:
GNU gdb (Ubuntu 15.0.50.20240403-0
ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software F
oundation, Inc.
```

2.设置gdb的远程超时时间为无限大，避免因为qemu运行时间过长而导致gdb超时。

```
0x0000000000001000 in ?? ()
(gdb) set remotetimeout unlimited
```

3.附着进程13848

```
heye@DESKTOP-63A0U9G:~/qemu-4.1.1$ pgrep -f qemu-system-riscv64
13848
```

```
(gdb) attach 13848
Attaching to process 13848
```

4.加载额外的符号信息文件

```
(gdb) set remotetimeout unlimited
(gdb) add-symbol-file obj/__usr_exit.out
add symbol table from file "obj/__usr_exit.out"
(y or n) y
obj/__usr_exit.out: No such file or directory.
(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...
```

add-symbol-file obj/__usr_exit.out 是 GDB（GNU 调试器）中的一个命令，用于在调试时加载额外的符号信息文件。其作用如下：

当你的程序（如操作系统或内核）动态加载了某个模块、用户程序或二进制文件（如 __usr_exit.out），而这些文件的符号（函数名、变量名等）没有包含在主调试信息中时，可以用 add-symbol-file 命令手动加载它们的符号表。

这样，GDB 就能识别和显示该文件中的函数、变量等符号，便于断点设置、单步调试和变量查看。

5.syscall.c打断点

```
(gdb) break user/libs/syscall.c:18
Breakpoint 1 at 0x8000f0: file user/libs/syscall
.c, line 19.
(gdb) c
Continuing.
```

```
    asm volatile (
        "ld a0, %1\n"
        "ld a1, %2\n"
        "ld a2, %3\n"
        "ld a3, %4\n"
        "ld a4, %5\n"
        "ld a5, %6\n"
        "ecall\n"
        "sd a0, %0"
        : "=m" (ret)
        : "m"(num), "m"(a[0]), "m"(a[1]), "m"(a[2]),
        :"memory");
    return ret;
}
```

汇编指令	作用
"ld a0, %1\n"	将参数 num（即系统调用号）加载到寄存器 a0
"ld a1, %2\n"	将 a[0] 加载到 a1
"ld a2, %3\n"	将 a[1] 加载到 a2
"ld a3, %4\n"	将 a[2] 加载到 a3

汇编指令	作用
"ld a4, %5\n"	将 a[3] 加载到 a4
"ld a5, %6\n"	将 a[4] 加载到 a5
"ecall\n"	触发 系统调用异常，CPU 从 U-mode 切换到 S-mode，跳转到内核的 trap 处理函数（如 trap_handler）
"sd a0, %0\n"	将内核返回值（在 a0 中）保存回变量 ret

它把系统调用号和最多5个参数分别装入 RISC-V 的 a0 到 a5 寄存器，然后执行 ecall 指令陷入内核；内核处理完后，将返回值通过 a0 传回，并存入变量 ret。

让用户程序安全地请求内核服务（如输出、退出等）。

6.观察ecall的位置

```
Breakpoint 1, syscall (num=2)
    at user/libs/syscall.c:19
19          asm volatile (
(gdb) x/8i $pc
=> 0x8000f0 <syscall+28>:
    sd  a6,128(sp)
   0x8000f2 <syscall+30>:
    sd  a7,136(sp)
   0x8000f4 <syscall+32>:       sd      t1,32(sp)
   0x8000f6 <syscall+34>:       ld      a0,8(sp)
   0x8000f8 <syscall+36>:       ld      a1,40(sp)
   0x8000fa <syscall+38>:       ld      a2,48(sp)
   0x8000fc <syscall+40>:       ld      a3,56(sp)
   0x8000fe <syscall+42>:       ld      a4,64(sp)
(gdb) si
0x00000000008000f2      19          asm volatile (
(gdb)
```

7.si单步执行到ecall

```
   0x80010e <syscall+58>:       ret
   0x800110 <sys_exit>: mv      a1,a0
(gdb) si
0x0000000000800100      19          asm volatile (
(gdb) x/8i $pc
=> 0x800100 <syscall+44>:       ld      a5,72(sp)
   0x800102 <syscall+46>:       ecall
   0x800106 <syscall+50>:       sd      a0,28(sp)
   0x80010a <syscall+54>:       lw      a0,28(sp)
   0x80010c <syscall+56>:       addi    sp,sp,144
   0x80010e <syscall+58>:       ret
   0x800110 <sys_exit>: mv      a1,a0
   0x800112 <sys_exit+2>:       li      a0,1
(gdb) si
0x0000000000800102      19          asm volatile (
```

CPU 仍处于 用户态，把系统调用号和最多5个参数分别装入 RISC-V 的 a0 到 a5 寄存器，然后执行 ecall 指令陷入内核；内核处理完后，将返回值通过 a0 传回，并存入变量 ret。

```
(gdb) i r pc
pc             0x800102 0x800102 <syscall+46>
```

8.执行单步，进入内核态

0xffffffffc0200e48 in __alltraps () at kern/trap/trapentry.S:123

9.在trapentry.S处的133行指令处打上断点

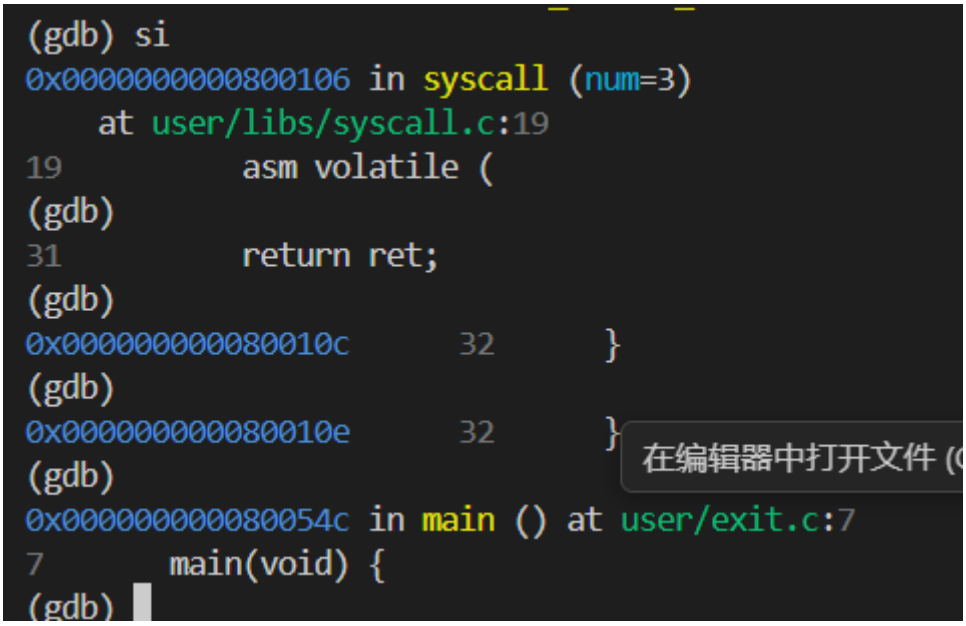
```
(gdb) b kern/trap/trapentry.S:133
Breakpoint 2 at 0xffffffffc0200fb2: file kern/trap/trapent
ry.S, line 133.
(gdb) c
Continuing.

Breakpoint 2, __trapret ()
    at kern/trap/trapentry.S:133
133         sret
```

```
127     # sp should be the same as before "jal trap"
128
129     .globl __trapret
130 __trapret:
131     RESTORE_ALL
132     # return from supervisor call
133     sret
134
135     .globl forkrets
136 forkrets:
137     # set stack to this new process's trapframe
138     move sp, a0
139     j __trapret
140
141     .global kernel_execve_ret
142 kernel_execve_ret:
143     // adjust sp to beneath kstacktop of current process
144     addi a1, a1, -36*REGBYTES
```

10.内核已完成系统调用处理，用户态寄存器即将被恢复，执行 sret 将切换回用户态并恢复 PC。

```
(gdb) x/7i $pc
=> 0xffffffffc0200fb2 <__trapret+86>:   sret
   0xffffffffc0200fb6 <forkrets>:       mv      sp,a0
   0xffffffffc0200fb8 <forkrets+2>:
    j   0xffffffffc0200f5c <__trapret>
   0xffffffffc0200fba <kernel_execve_ret>:      addi    a1,a1,-288
   0xffffffffc0200fbe <kernel_execve_ret+4>:    ld      s1,280(a0)
   0xffffffffc0200fc2 <kernel_execve_ret+8>:    sd      s1,280(a1)
   0xffffffffc0200fc6 <kernel_execve_ret+12>:   ld      s1,272(a0)
(gdb)
```

11. si单步调试返回到用户程序中：



三、ecall 和 sret 指令在 QEMU 中的处理流程

1.ecall 指令的处理

ecall 是用户/内核发起系统调用的指令，会触发陷入（trap），QEMU 需要模拟这种异常。

在 QEMU 的 riscv 架构实现中，ecall 相关的处理主要在 target/riscv/insn_trans/trans_rvv.c、target/riscv/op_helper.c 和 target/riscv/cpu_helper.c 等文件。

关键流程如下：

1. QEMU 译码到 ecall 指令时，会调用 helper 函数（如 helper_ecall）。
2. helper_ecall 会调用 raise_exception 函数，向 QEMU 的 CPU 结构体注入一个 ECALL 异常（EXCP_ENVCALL）。
3. QEMU 的 CPU 执行主循环（cpu_exec）检测到异常后，会跳转到异常处理流程，模拟硬件 trap 过程，保存上下文，切换到 trap handler。
4. QEMU 会设置 sepc、scause、stval 等寄存器，并跳转到 stvec 指定的异常入口（如 trapentry.S 里的 __alltraps）。

2.sret 指令的处理

sret 是从 supervisor trap 返回的指令，恢复 trap 前的上下文。

QEMU 译码到 sret 时，会调用 helper_sret（target/riscv/op_helper.c）。

关键流程如下：

1.helper_sret 检查当前特权级和 sstatus 寄存器，恢复 sepc 到 PC，恢复 sstatus。

2.QEMU 会模拟硬件行为，把 PC 设置为 sepc，sstatus 的 SPP 位恢复到 SIE 等。

3.继续执行 trap 前的指令流。

四、指令翻译（TCG Translation）

TCG（Tiny Code Generator）是 QEMU 的动态二进制翻译引擎。QEMU 运行时会将目标架构（如 RISC-V）的指令（包括 ecall、sret 等）翻译为宿主机可执行的中间代码（TCG IR），再进一步生成本地机器码执行。

1.TCG Translation 关键点

QEMU 不是直接解释执行每条指令，而是先将目标指令块（如 ecall、sret）翻译为 TCG IR，再编译成本地代码，提高执行效率。

对于特殊指令（如 ecall、sret），QEMU 会在 TCG translation 阶段插入调用 helper 函数（如 helper_ecall、helper_sret），这些 helper 负责模拟异常、陷入、返回等硬件行为。

这样，QEMU 能高效且准确地模拟目标 CPU 的行为，包括异常处理、上下文切换等。

2.与双重 GDB 调试的关系

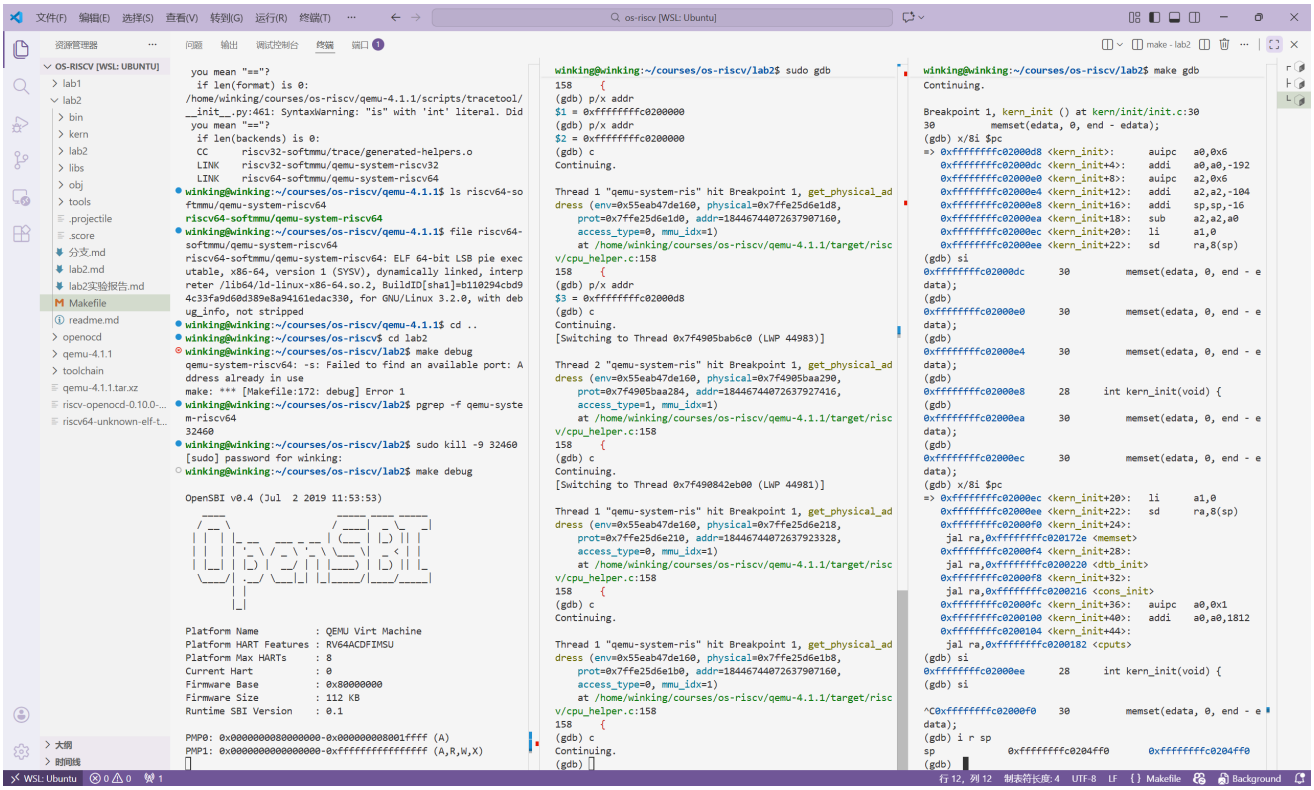
在双 GDB 调试实验（如同时调试 QEMU 和被模拟的内核/用户程序）中，QEMU 作为被调试对象，其 TCG translation 过程和 helper 函数的调用都可以被 GDB 跟踪。

你可以在 QEMU 的源码（如 helper_ecall）设置断点，观察 QEMU 如何响应 ecall/sret，并进一步跟踪 guest 代码的 trap handler。

这种调试方式让你既能看到 guest 代码的执行，也能深入 QEMU 的实现细节，理解指令翻译和异常分发的全过程。

TCG translation 是 QEMU 高效模拟的核心，所有指令（包括 ecall/sret）都要经过这一翻译流程。双重 GDB 调试实验正是利用了 QEMU 的这一机制，既能调试 guest 代码，也能调试 QEMU 的指令翻译和异常处理逻辑，两者密切相关。

lab2分支:基于 QEMU-4.1.1 的 RISC-V 虚拟地址翻译过程调试与分析实验报告



一、实验背景与实验目标

在 uCore 实验中，我们已经完成了物理内存管理以及基于 RISC-V Sv39 的虚拟内存机制实现，并成功让内核运行在开启分页机制的虚拟地址空间中。从操作系统视角来看，当 CPU 执行一条访存指令（load / store）时，虚拟地址到物理地址的转换是由硬件 MMU 自动完成的，操作系统只负责构建页表。

在本实验环境中，uCore 并非运行在真实的 RISC-V 硬件上，而是运行在 QEMU 模拟器所模拟的 CPU 上。原本由硬件完成的地址翻译流程，在 QEMU 中是由软件精确模拟的。

本实验的目标正是借助 GDB 对 QEMU 源码进行调试，直接观察一次访存指令从“虚拟地址”到“物理地址”的完整翻译过程，具体包括：

- 理解 QEMU 中一次 RISC-V 访存指令的整体执行路径；
- 观察 QEMU 如何模拟 TLB 查找以及 TLB miss 处理；
- 单步分析 QEMU 中 Sv39 多级页表遍历的具体实现；
- 对比开启与关闭虚拟地址空间时，访存路径在 QEMU 中的差异；
- 理解 QEMU 软件 TLB 与真实 CPU 硬件 TLB 在逻辑上的异同。

二、实验环境与调试方式说明

实验采用 **QEMU-4.1.1 源码版本**，并通过 `--enable-debug` 选项重新编译生成带有完整调试信息的调试版 QEMU。该版本不执行 `make install`，而是作为独立的可执行文件使用，从而避免覆盖系统中原有的 QEMU。

整个实验采用 **双 GDB + 三终端** 的调试架构：

- **终端 1**：启动调试版 QEMU，加载 uCore 内核，并通过 `-S -gdb tcp::1234` 方式暂停在初始状态；
- **终端 2**：使用系统自带的 gdb，直接 attach 到 QEMU 进程，用于调试 QEMU 源码；

- **终端 3**：使用 `riscv64-unknown-elf-gdb`，通过 QEMU 提供的 GDB stub 连接，调试 uCore 内核代码。

这种调试方式的核心优势在于：**可以在 uCore 停在某一条访存指令的同时，切换到 QEMU 侧观察这条指令是如何被“硬件”处理的**，从而在同一时间尺度上观察软件与模拟硬件的交互。

三、QEMU 中一次访存指令的完整执行路径

在 QEMU 的 TCG（Tiny Code Generator）模式下，RISC-V 指令并不是逐条解释执行，而是被翻译为中间表示并生成宿主机代码执行。然而，从逻辑上看，一条访存指令仍然遵循“地址翻译 → 访问内存”的顺序。

通过对 QEMU 源码的跟踪，可以总结出一次访存指令在 **开启虚拟地址空间** 时的大致执行路径如下：

1. uCore 执行一条 RISC-V 的 load 或 store 指令；
2. 该指令在 QEMU 中被 TCG 翻译为对应的访存 helper 或内联访存逻辑；
3. 在真正访问宿主机内存之前，QEMU 会首先查询其内部的软件 TLB；
4. 如果 TLB 命中，QEMU 直接得到对应的物理地址，完成访存；
5. 如果 TLB 未命中，QEMU 进入架构相关的 TLB miss 处理流程；
6. 对于 RISC-V，该流程由 `riscv_cpu_tlb_fill()` 负责；
7. 在 TLB fill 过程中，QEMU 调用 `get_physical_address()`，执行完整的 Sv39 页表遍历；
8. 成功得到物理地址后，QEMU 将该映射写入软件 TLB；
9. 返回并重新执行原始访存操作。

这一流程与真实 RISC-V 硬件的行为在**逻辑顺序上是高度一致的**，只是硬件的并行电路被软件代码所取代。

四、TLB 查找与 TLB miss 处理机制分析

观察发现，**TLB 查找本身并不位于 RISC-V 架构相关目录中。**
这是因为 QEMU 采用了分层设计：

TLB 的数据结构与查找逻辑位于通用的 TCG 层，主要实现于 `accel/tcg/cputlb.c`；

不同架构只需要在 TLB miss 时提供地址翻译逻辑。

TLB 查找通常以内联代码或宏的形式完成，而当查找失败时，才会显式调用架构相关的函数。对于 RISC-V 来说，TLB miss 后进入的关键函数为：

```
bool riscv_cpu_tlb_fill(CPUState *cs, vaddr address,
                        int size, MMUAccessType access_type,
                        int mmu_idx, bool probe,
                        uintptr_t retaddr)
```

五、Sv39 页表遍历的单步调试与原理解释

在 `riscv_cpu_tlb_fill()` 内部，QEMU 会进一步调用 `get_physical_address()`，该函数完整实现了 RISC-V Sv39 的多级页表遍历逻辑。

Sv39 模式下，一个 39 位虚拟地址被划分为：VPN[2]、VPN[1]、VPN[0]：三层页表索引，page offset：页内偏移。

在调试中可以观察到，`get_physical_address()` 中存在一个从高层到低层的循环结构。该循环并非“普通算法”，而是对硬件 MMU 行为的逐行展开：

- 第一轮循环使用 VPN[2]，从 SATP 指定的根页表中查找一级页表项；
- 如果页表项有效且不是叶子项，则取出其中的 PPN，作为下一层页表的物理地址；
- 第二轮循环使用 VPN[1]，查找二级页表；
- 第三轮循环使用 VPN[0]，查找三级页表；
- 当遇到设置了 R/W/X 位的页表项时，判定为叶子页表项，结束遍历。

在每一层中，QEMU 都会通过物理地址读取页表项。这一点与真实硬件一致：**页表访问始终发生在物理地址空间中，而不是递归地使用虚拟地址。**

当遍历结束后，QEMU 将页表项中的物理页号与原虚拟地址的页内偏移组合，最终生成对应的物理地址。

六、开启与关闭虚拟地址空间时的访存差异

为了进一步理解 QEMU 的模拟逻辑，实验中还对比了虚拟地址空间关闭（SATP = 0）时的访存行为。

通过调试可以发现：

- 在 SATP 为 0 的情况下，MMU 被视为关闭；
- QEMU 不会进入 `riscv_cpu_tlb_fill()`；
- 也不会调用 `get_physical_address()`；
- 虚拟地址被直接当作物理地址使用。

严格按照 RISC-V 架构规范，在软件层面条件性地启用 MMU 逻辑

七、QEMU 软件 TLB 与真实硬件 TLB 的对比

通过本次实验，可以对 QEMU 中的软件 TLB 与真实 CPU 中的硬件 TLB 进行如下对比：

- 真实 CPU 的 TLB 通常使用并行 CAM 结构，查找在一个时钟周期内完成；
- QEMU 的 TLB 使用普通的数据结构（数组、哈希等）在软件中实现；
- 在逻辑上，两者都遵循“先查 TLB，miss 后页表遍历”的流程；
- 区别仅在于实现方式和性能，而非语义。

这也说明：**QEMU 的目标并非模拟微架构，而是精确模拟架构层面的可见行为。**

八、调试过程中遇到的问题与经验总结

在实际调试过程中，遇到了一些具有代表性的问题：

- 使用 `step` 单步执行时，GDB 经常进入 `pthread` 或 glibc 相关代码，这是由于 QEMU 是多线程程序，且 `step` 会进入库函数；
- 在错误的栈帧中尝试访问 `env` 等变量，会出现“符号不存在”的情况，其本质并非调试信息缺失，而是当前上下文已经不在 RISC-V CPU 模拟代码中。

这些问题的解决过程本身，也加深了对 QEMU 运行机制的理解。

九、实验总结

通过本次实验，可以得出以下结论：

1. QEMU 通过软件方式完整模拟了 RISC-V 虚拟地址翻译机制；
2. TLB 查找位于通用 TCG 层，页表遍历由架构相关代码实现；
3. Sv39 页表遍历代码本质上是硬件 MMU 行为的逐步展开；
4. 开启或关闭虚拟地址空间，会显著改变访存的执行路径；
5. 通过调试模拟器源码，可以将抽象的体系结构概念转化为具体可观察的执行过程。

本实验不仅加深了对虚拟内存机制的理解，也展示了借助大型模型辅助学习复杂系统源码的一种有效方法。