Lab2_GDB调试页表查询过程

基础知识：在虚拟内存访问模式下，CPU产生的虚拟地址由MMU负责转换为物理地址。MMU首先查询TLB，若命中则直接获得物理页帧号并组合成物理地址完成访问；若未命中，则根据SATP寄存器基址逐级查询页表，在获得物理页帧号后完成访问，同时将该映射存入TLB以加速后续转换。这些功能都是由**硬件**完成的。

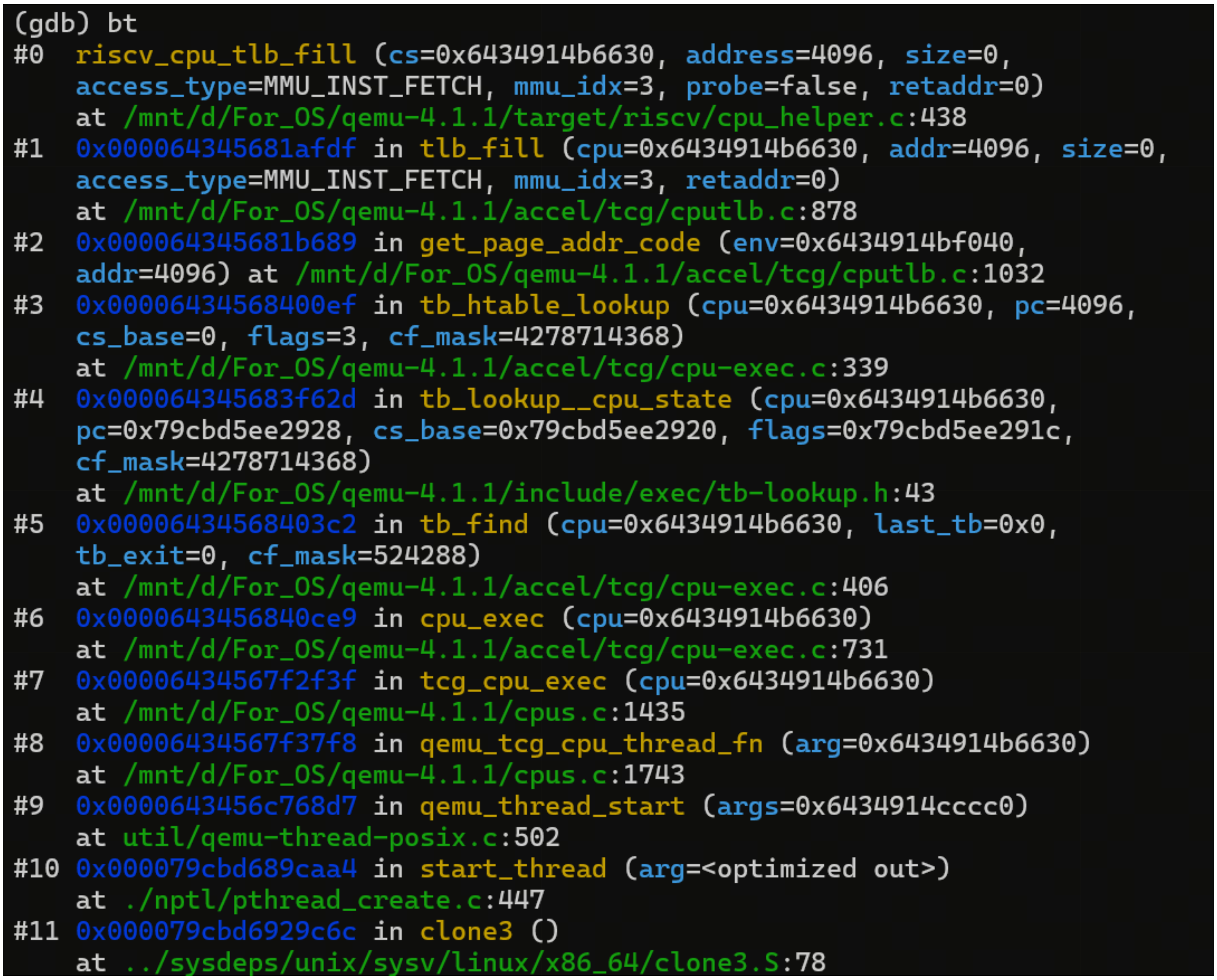
调试目的：尝试观察qemu在接收到一个访存指令时是如何一步步操作的，深入对比模拟器的实现，充分理解整个地址翻译的过程。

步骤：

终端1：输入make debug，然后没反应，这是正常的

终端2：输入pgrep -f qemu-system-riscv64查看PID，然后sudo gdb，然后attach [刚才的PID]，然后handle SIGPIPE nostop noprint，然后设置断点break riscv_cpu_tlb_fill，然后c

终端3：make gdb，然后c，此时终端2停在了断点处，然后输入 `bt` 查看栈：

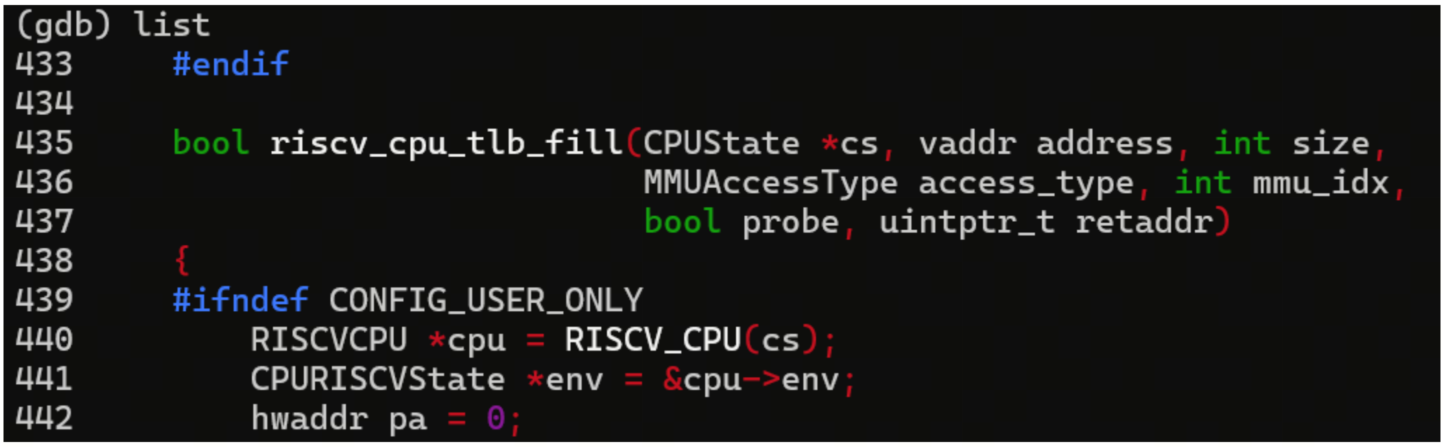


```
(gdb) bt
#0  riscv_cpu_tlb_fill (cs=0x6434914b6630, address=4096, size=0,
    access_type=MMU_INST_FETCH, mmu_idx=3, probe=false, retaddr=0)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:438
#1  0x00006434568laf df in tlb_fill (cpu=0x6434914b6630, addr=4096, size=0,
    access_type=MMU_INST_FETCH, mmu_idx=3, retaddr=0)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cputlb.c:878
#2  0x00006434568lb689 in get_page_addr_code (env=0x6434914bf040,
    addr=4096) at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cputlb.c:1032
#3  0x00006434568400ef in tb_htable_lookup (cpu=0x6434914b6630, pc=4096,
    cs_base=0, flags=3, cf_mask=4278714368)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cpu-exec.c:339
#4  0x000064345683f62d in tb_lookup__cpu_state (cpu=0x6434914b6630,
    pc=0x79cbd5ee2928, cs_base=0x79cbd5ee2920, flags=0x79cbd5ee291c,
    cf_mask=4278714368)
    at /mnt/d/For_OS/qemu-4.1.1/include/exec/tb-lookup.h:43
#5  0x00006434568403c2 in tb_find (cpu=0x6434914b6630, last_tb=0x0,
    tb_exit=0, cf_mask=524288)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cpu-exec.c:406
#6  0x000064345684 0ce9 in cpu_exec (cpu=0x6434914b6630)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cpu-exec.c:731
#7  0x00006434567f2f3f in tcg_cpu_exec (cpu=0x6434914b6630)
    at /mnt/d/For_OS/qemu-4.1.1/cpus.c:1435
#8  0x00006434567f37f8 in qemu_tcg_cpu_thread_fn (arg=0x6434914b6630)
    at /mnt/d/For_OS/qemu-4.1.1/cpus.c:1743
#9  0x000064345 6c768d7 in qemu_thread_start (args=0x6434914cccc0)
    at util/qemu-thread-posix.c:502
#10 0x000079cbd689caa4 in start_thread (arg=<optimized out>)
    at ./nptl/pthread_create.c:447
#11 0x000079cbd6929c6c in clone3 ()
    at ../sysdeps/unix/sysv/linux/x86_64/clone3.S:78
```

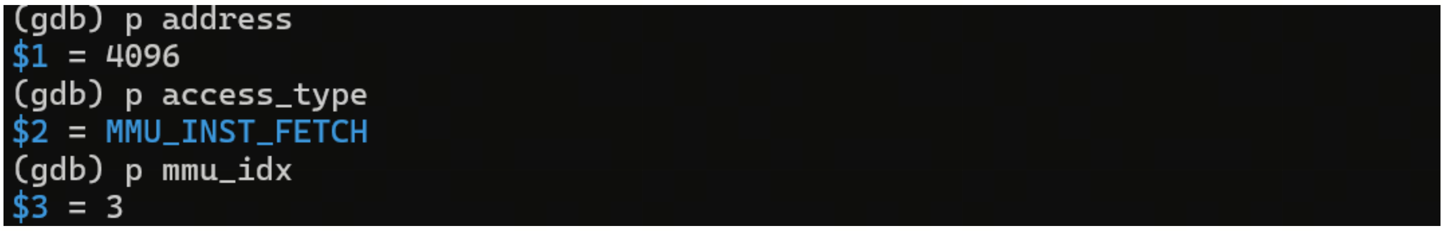
上述调用栈展示了QEMU模拟RISC-V CPU时**TLB未命中**的处理流程：当CPU执行到reset vector地址 `0x1000` （4096）进行指令取指时，TLB中没有该地址的映射，于是触发TLB重填。调用从 `cpu_exec` （CPU主执行循环）开始，通过 `tb_find` 查找翻译块，发现需要翻译 `0x1000` 地址的代码。 `tb_htable_lookup` 调用 `get_page_addr_code` 获取物理地址，但发现页表未缓存，于是调用 `tlb_fill` 进行TLB填充。最终进入 `riscv_cpu_tlb_fill` 函数，该函数会调用 `get_physical_address` 进行实际的页表遍历，将虚拟地址 `0x1000` 转换为物理地址，并填充到TLB中供后续使用。

整个过程体现了**指令执行 → TLB未命中 → 页表遍历 → TLB填充**的完整MMU工作流程。

然后输入 `(gdb) list` ，我们可以看到 `riscv_cpu_tlb_fill()` 的开头部分：

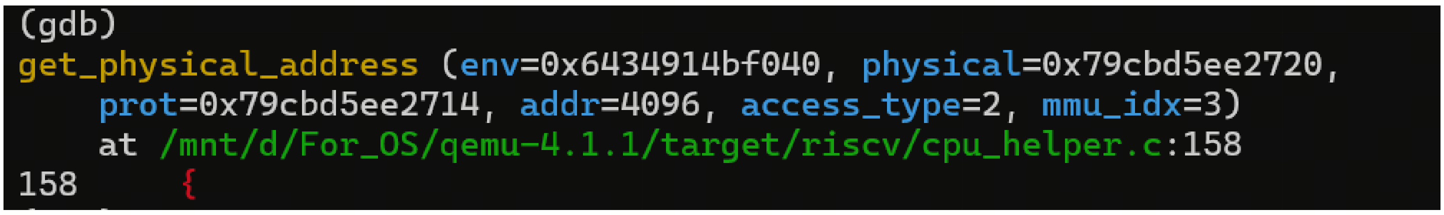

```
(gdb) list
433     #endif
434
435     bool riscv_cpu_tlb_fill(CPUState *cs, vaddr address, int size,
436                             MMUAccessType access_type, int mmu_idx,
437                             bool probe, uintptr_t retaddr)
438     {
439     #ifndef CONFIG_USER_ONLY
440         RISCVCPU *cpu = RISCV_CPU(cs);
441         CPURISCVState *env = &cpu->env;
442         hwaddr pa = 0;
```

然后输入 `(gdb) print address` 查看访存地址，输入 `(gdb) print access_type` 查看访存类型，输入 `(gdb) print mmu_idx` 查看MMU模式：


```
(gdb) p address
$1 = 4096
(gdb) p access_type
$2 = MMU_INST_FETCH
(gdb) p mmu_idx
$3 = 3
```

可以看到，访存地址为 `$1 = 4096`，也就是 `0x1000`。访问类型为 `MMU_INST_FETCH`，说明是执行指令时触发翻译，这是正常的。MMU模式为3，表示当前为M模式。

接下来在终端2输入多个 `(gdb) step`，直至到达函数 `get_physical_address()`：


```
(gdb)
get_physical_address (env=0x6434914bf040, physical=0x79cbd5ee2720,
    prot=0x79cbd5ee2714, addr=4096, access_type=2, mmu_idx=3)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:158
158     {
```

我们去对应的路径 `/mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c` 中看看这个函数的qemu源码：

代码块

```
static int get_physical_address(CPURISCVState *env, hwaddr *physical,
                                int *prot, target_ulong addr,
                                int access_type, int mmu_idx)
{
    /* NOTE: the env->pc value visible here will not be
     * correct, but the value visible to the exception handler
     * (riscv_cpu_do_interrupt) is correct */

    int mode = mmu_idx;

    if (mode == PRV_M && access_type != MMU_INST_FETCH) {
```

```c
        if (get_field(env->mstatus, MSTATUS_MPRV)) {
            mode = get_field(env->mstatus, MSTATUS_MPP);
        }
    }

    if (mode == PRV_M || !riscv_feature(env, RISCV_FEATURE_MMU)) {
        *physical = addr;
        *prot = PAGE_READ | PAGE_WRITE | PAGE_EXEC;
        return TRANSLATE_SUCCESS;
    }

    *prot = 0;

    target_ulong base;
    int levels, ptidxbits, ptesize, vm, sum;
    int mxr = get_field(env->mstatus, MSTATUS_MXR);

    if (env->priv_ver >= PRIV_VERSION_1_10_0) {
        base = get_field(env->satp, SATP_PPN) << PGSHIFT;
        sum = get_field(env->mstatus, MSTATUS_SUM);
        vm = get_field(env->satp, SATP_MODE);
        switch (vm) {
        case VM_1_10_SV32:
          levels = 2; ptidxbits = 10; ptesize = 4; break;
        case VM_1_10_SV39:
          levels = 3; ptidxbits = 9; ptesize = 8; break;
        case VM_1_10_SV48:
          levels = 4; ptidxbits = 9; ptesize = 8; break;
        case VM_1_10_SV57:
          levels = 5; ptidxbits = 9; ptesize = 8; break;
        case VM_1_10_MBARE:
            *physical = addr;
            *prot = PAGE_READ | PAGE_WRITE | PAGE_EXEC;
            return TRANSLATE_SUCCESS;
        default:
          g_assert_not_reached();
        }
    } else {
        base = env->sptbr << PGSHIFT;
        sum = !get_field(env->mstatus, MSTATUS_PUM);
        vm = get_field(env->mstatus, MSTATUS_VM);
        switch (vm) {
        case VM_1_09_SV32:
          levels = 2; ptidxbits = 10; ptesize = 4; break;
        case VM_1_09_SV39:
          levels = 3; ptidxbits = 9; ptesize = 8; break;
        case VM_1_09_SV48:
          levels = 4; ptidxbits = 9; ptesize = 8; break;
        case VM_1_09_MBARE:
            *physical = addr;
            *prot = PAGE_READ | PAGE_WRITE | PAGE_EXEC;
            return TRANSLATE_SUCCESS;
        default:
          g_assert_not_reached();
        }
    }

    CPUState *cs = env_cpu(env);
    int va_bits = PGSHIFT + levels * ptidxbits;
    target_ulong mask = (1L << (TARGET_LONG_BITS - (va_bits - 1))) - 1;
    target_ulong masked_msbs = (addr >> (va_bits - 1)) & mask;
    if (masked_msbs != 0 && masked_msbs != mask) {
        return TRANSLATE_FAIL;
    }

    int ptshift = (levels - 1) * ptidxbits;
    int i;

#if !TCG_OVERSIZED_GUEST
restart:
#endif
    for (i = 0; i < levels; i++, ptshift -= ptidxbits) {
        target_ulong idx = (addr >> (PGSHIFT + ptshift)) &
                           ((1 << ptidxbits) - 1);

        /* check that physical address of PTE is legal */
        target_ulong pte_addr = base + idx * ptesize;

        if (riscv_feature(env, RISCV_FEATURE_PMP) &&
            !pmp_hart_has_privs(env, pte_addr, sizeof(target_ulong),
            1 << MMU_DATA_LOAD, PRV_S)) {
            return TRANSLATE_PMP_FAIL;
        }
#if defined(TARGET_RISCV32)
        target_ulong pte = ldl_phys(cs->as, pte_addr);
#elif defined(TARGET_RISCV64)
        target_ulong pte = ldq_phys(cs->as, pte_addr);
#endif
        target_ulong ppn = pte >> PTE_PPN_SHIFT;

        if (!(pte & PTE_V)) {
            /* Invalid PTE */
            return TRANSLATE_FAIL;
        } else if (!(pte & (PTE_R | PTE_W | PTE_X))) {
            /* Inner PTE, continue walking */
            base = ppn << PGSHIFT;
        } else if ((pte & (PTE_R | PTE_W | PTE_X)) == PTE_W) {
            /* Reserved leaf PTE flags: PTE_W */
            return TRANSLATE_FAIL;
        } else if ((pte & (PTE_R | PTE_W | PTE_X)) == (PTE_W | PTE_X)) {
            /* Reserved leaf PTE flags: PTE_W + PTE_X */
            return TRANSLATE_FAIL;
        } else if ((pte & PTE_U) && ((mode != PRV_U) &&
                   (!sum || access_type == MMU_INST_FETCH))) {
            /* User PTE flags when not U mode and mstatus.SUM is not set,
               or the access type is an instruction fetch */
            return TRANSLATE_FAIL;
        } else if (!(pte & PTE_U) && (mode != PRV_S)) {
            /* Supervisor PTE flags when not S mode */
            return TRANSLATE_FAIL;
        } else if (ppn & ((1ULL << ptshift) - 1)) {
            /* Misaligned PPN */
            return TRANSLATE_FAIL;
        } else if (access_type == MMU_DATA_LOAD && !((pte & PTE_R) ||
                   ((pte & PTE_X) && mxr))) {
            /* Read access check failed */
            return TRANSLATE_FAIL;
        } else if (access_type == MMU_DATA_STORE && !(pte & PTE_W)) {
            /* Write access check failed */
            return TRANSLATE_FAIL;
        } else if (access_type == MMU_INST_FETCH && !(pte & PTE_X)) {
            /* Fetch access check failed */
            return TRANSLATE_FAIL;
        } else {
            /* if necessary, set accessed and dirty bits. */
            target_ulong updated_pte = pte | PTE_A |
                (access_type == MMU_DATA_STORE ? PTE_D : 0);

            /* Page table updates need to be atomic with MTTCG enabled */
            if (updated_pte != pte) {
                /*
                 * - if accessed or dirty bits need updating, and the PTE is
                 *   in RAM, then we do so atomically with a compare and swap.
                 * - if the PTE is in IO space or ROM, then it can't be updated
                 *   and we return TRANSLATE_FAIL.
                 * - if the PTE changed by the time we went to update it, then
                 *   it is no longer valid and we must re-walk the page table.
                 */
                MemoryRegion *mr;
                hwaddr l = sizeof(target_ulong), addr1;
                mr = address_space_translate(cs->as, pte_addr,
                    &addr1, &l, false, MEMTXATTRS_UNSPECIFIED);
                if (memory_region_is_ram(mr)) {
                    target_ulong *pte_pa =
                        qemu_map_ram_ptr(mr->ram_block, addr1);
#if TCG_OVERSIZED_GUEST
                    /* MTTCG is not enabled on oversized TCG guests so
                     * page table updates do not need to be atomic */
                    *pte_pa = pte = updated_pte;
#else
                    target_ulong old_pte =
                        atomic_cmpxchg(pte_pa, pte, updated_pte);
                    if (old_pte != pte) {
                        goto restart;
                    } else {
                        pte = updated_pte;
                    }
#endif
                } else {
                    /* misconfigured PTE in ROM (AD bits are not preset) or
                     * PTE is in IO space and can't be updated atomically */
                    return TRANSLATE_FAIL;
                }
            }

            /* for superpage mappings, make a fake leaf PTE for the TLB's
               benefit. */
            target_ulong vpn = addr >> PGSHIFT;
            *physical = (ppn | (vpn & ((1L << ptshift) - 1))) << PGSHIFT;

            /* set permissions on the TLB entry */
            if ((pte & PTE_R) || ((pte & PTE_X) && mxr)) {
                *prot |= PAGE_READ;
            }
            if ((pte & PTE_X)) {
                *prot |= PAGE_EXEC;
            }
            /* add write permission on stores or if the page is already dirty,
               so that we TLB miss on later writes to update the dirty bit */
            if ((pte & PTE_W) &&
                    (access_type == MMU_DATA_STORE || (pte & PTE_D))) {
                *prot |= PAGE_WRITE;
            }
            return TRANSLATE_SUCCESS;
        }
    }
    return TRANSLATE_FAIL;
}
```

这是QEMU中RISC-V架构的页表遍历核心函数，负责将虚拟地址转换为物理地址，是MMU（内存管理单元）的核心实现。

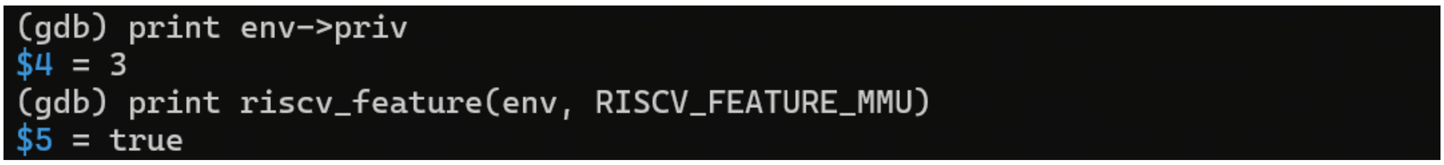
函数参数：

- `env`：CPU状态寄存器环境
- `physical`：输出参数，存放转换后的物理地址
- `prot`：输出参数，存放页面访问权限
- `addr`：输入的虚拟地址
- `access_type`：访问类型（取指/读/写）
- `mmu_idx`：MMU索引，对应CPU特权级

执行流程：

1. 模式检查与快速路径：M模式或无MMU则直接映射
2. 页表配置解析：读取sapt寄存器
3. 地址有效性检查：检查虚拟地址的高位是否符号扩展正确
4. 多级页表遍历（核心）：SV39三级遍历循环 for循环
5. PTE权限检查：
   - `PTE_V`：有效位（必须为1）
   - `PTE_R/W/X`：读/写/执行权限
   - `PTE_U`：用户页标志
   - `PTE_A/D`：访问/脏位
6. 物理地址生成：将PTE中的PPN与虚拟地址的低位组合
7. 权限设置：根据PTE权限设置`prot`参数
   - `PAGE_READ`：可读
   - `PAGE_WRITE`：可写
   - `PAGE_EXEC`：可执行

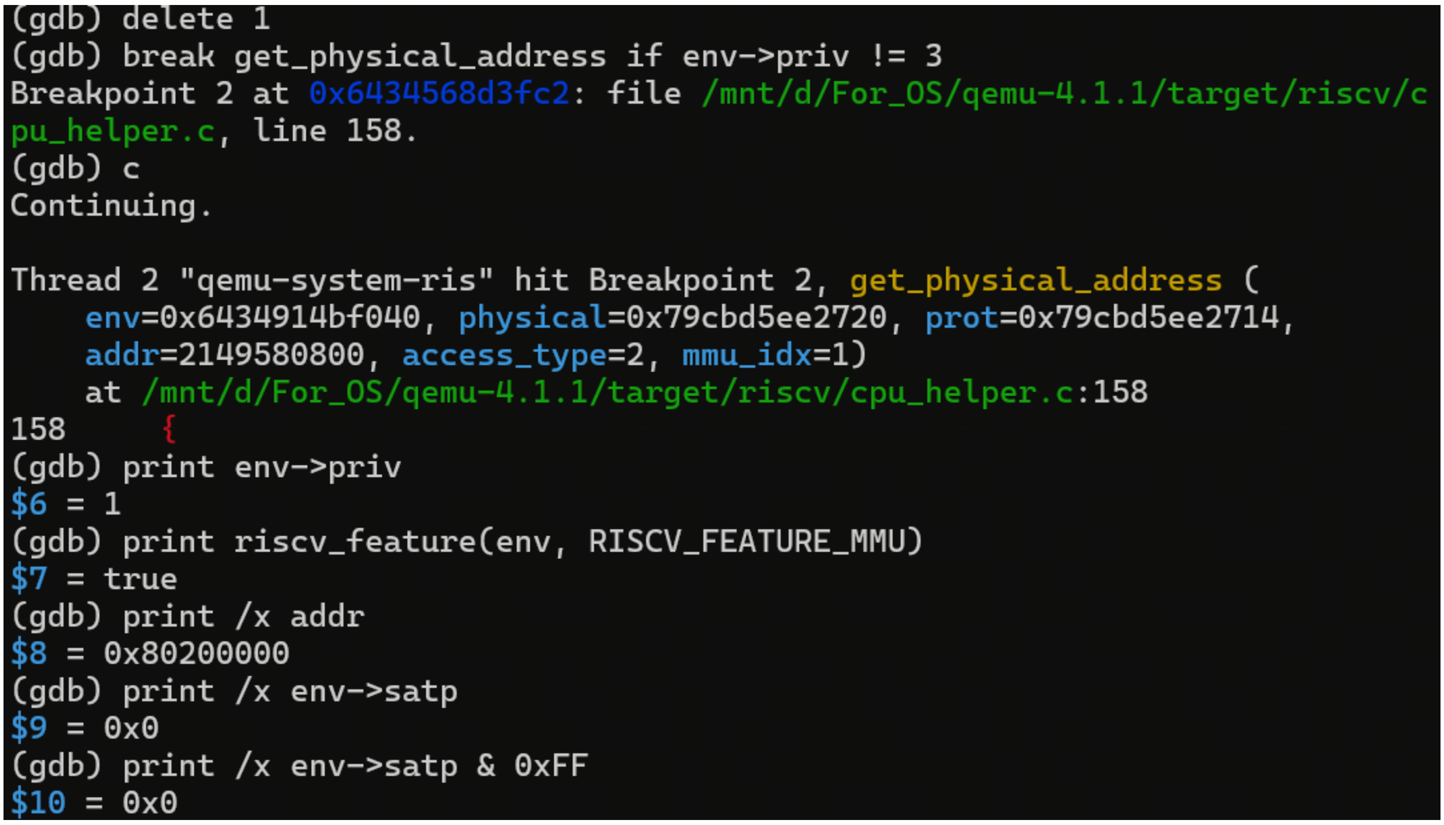
然后我们尝试遍历三级页表：


```
(gdb) print env->priv
$4 = 3
(gdb) print riscv_feature(env, RISCV_FEATURE_MMU)
$5 = true
```

代码块

```
(gdb) print env->priv
$4 = 3 #仍然是M模式，
(gdb) print riscv_feature(env, RISCV_FEATURE_MMU)
$5 = true #虽然是true，但因为是M模式，所以不会遍历页表
```

所以要等待S或U模式的访问，我们设置条件断点：



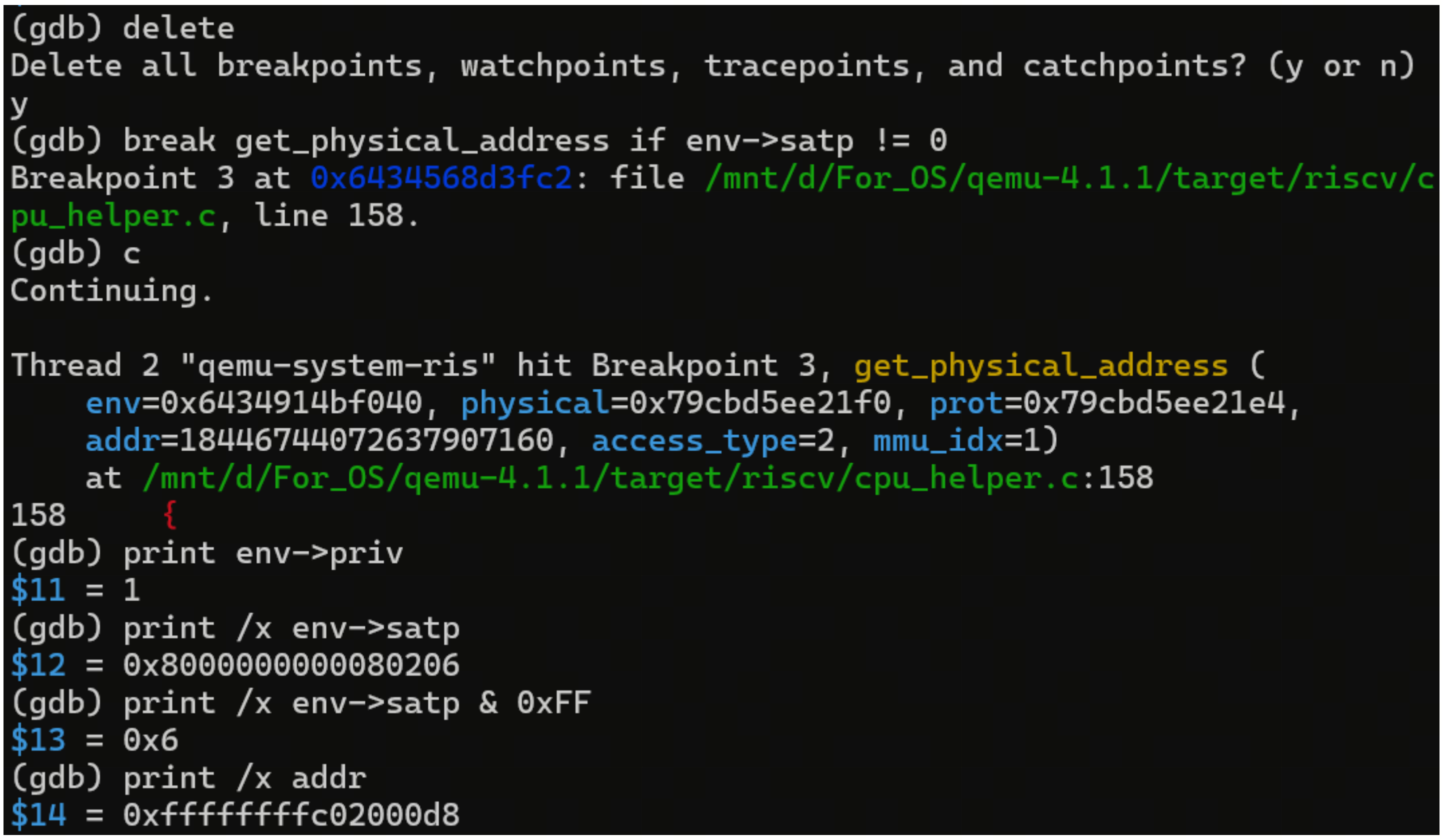
```
(gdb) delete 1
(gdb) break get_physical_address if env->priv != 3
Breakpoint 2 at 0x6434568d3fc2: file /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c, line 158.
(gdb) c
Continuing.

Thread 2 "qemu-system-ris" hit Breakpoint 2, get_physical_address (
    env=0x6434914bf040, physical=0x79cbd5ee2720, prot=0x79cbd5ee2714,
    addr=2149580800, access_type=2, mmu_idx=1)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:158
158       {
(gdb) print env->priv
$6 = 1
(gdb) print riscv_feature(env, RISCV_FEATURE_MMU)
$7 = true
(gdb) print /x addr
$8 = 0x80200000
(gdb) print /x env->satp
$9 = 0x0
(gdb) print /x env->satp & 0xFF
$10 = 0x0
```

代码块

```
(gdb) delete 1 #删除当前断点
(gdb) break get_physical_address if env->priv != 3 #设置条件断点：只在非M模式时触发
Breakpoint 2 at 0x6434568d3fc2: file /mnt/d/For_OS/qemu-
4.1.1/target/riscv/cpu_helper.c, line 158.
(gdb) c #继续执行
Continuing.

Thread 2 "qemu-system-ris" hit Breakpoint 2, get_physical_address (
    env=0x6434914bf040, physical=0x79cbd5ee2720, prot=0x79cbd5ee2714,
    addr=2149580800, access_type=2, mmu_idx=1)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:158
158       {
(gdb) print env->priv
$6 = 1 #现在切换到S模式了！
(gdb) print riscv_feature(env, RISCV_FEATURE_MMU)
$7 = true

(gdb) print /x addr
```

```
$8 = 0x80200000
(gdb) print /x env->satp
$9 = 0x0 # satp = 0, 表示分页还没有启用
(gdb) print /x env->satp & 0xFF
$10 = 0x0
```

我们继续设置断点：



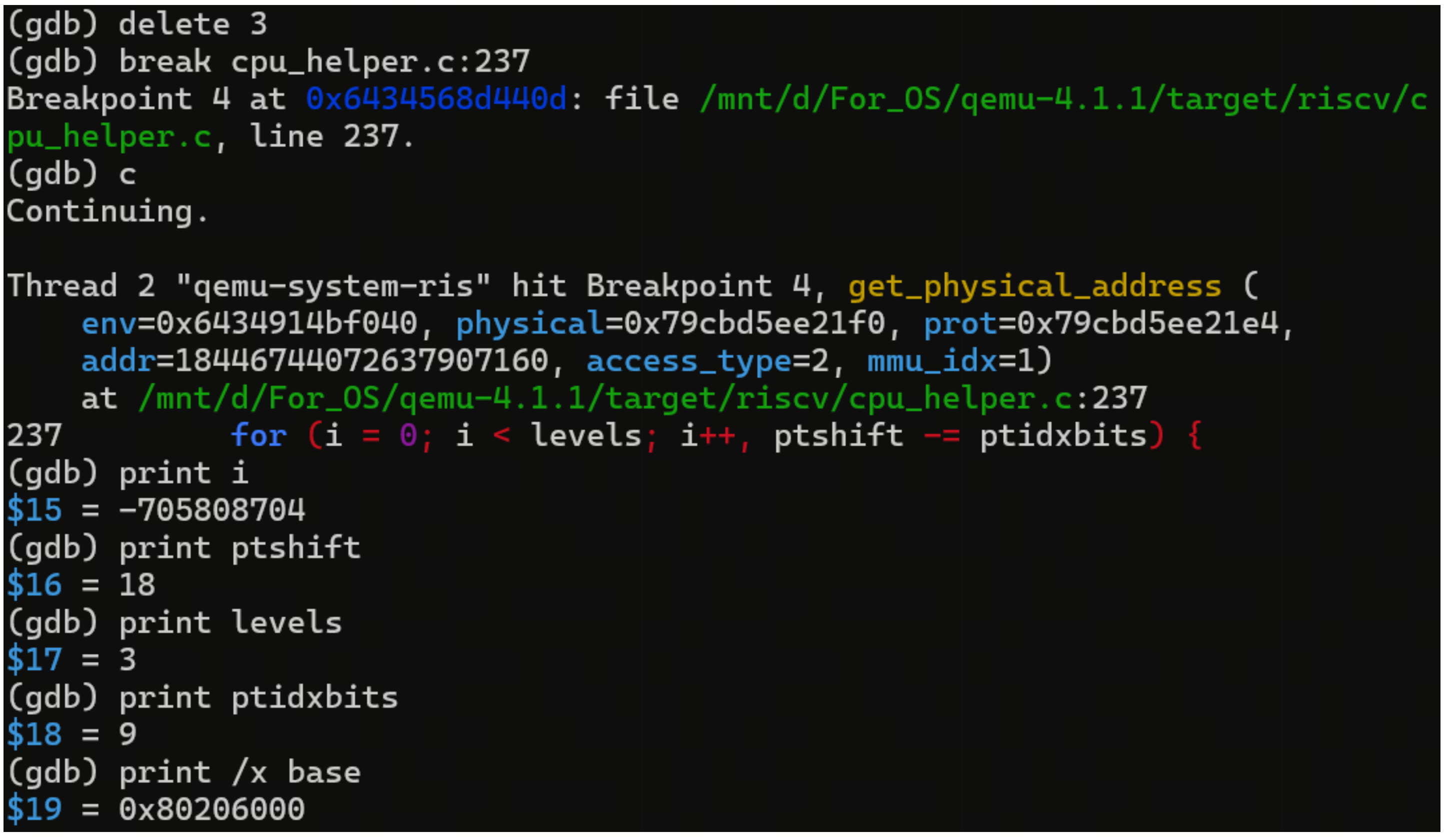
```
(gdb) delete
Delete all breakpoints, watchpoints, tracepoints, and catchpoints? (y or n)
y
(gdb) break get_physical_address if env->satp != 0
Breakpoint 3 at 0x6434568d3fc2: file /mnt/d/For_OS/qemu-4.1.1/target/riscv/c
pu_helper.c, line 158.
(gdb) c
Continuing.

Thread 2 "qemu-system-ris" hit Breakpoint 3, get_physical_address (
    env=0x6434914bf040, physical=0x79cbd5ee21f0, prot=0x79cbd5ee21e4,
    addr=18446744072637907160, access_type=2, mmu_idx=1)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:158
158     {
(gdb) print env->priv
$11 = 1
(gdb) print /x env->satp
$12 = 0x8000000000080206
(gdb) print /x env->satp & 0xFF
$13 = 0x6
(gdb) print /x addr
$14 = 0xffffffffc02000d8
```

代码块

```
(gdb) delete
Delete all breakpoints, watchpoints, tracepoints, and catchpoints? (y or n) y
(gdb) break get_physical_address if env->satp != 0 #继续设置条件断点
Breakpoint 3 at 0x6434568d3fc2: file /mnt/d/For_OS/qemu-
4.1.1/target/riscv/cpu_helper.c, line 158.
(gdb) c
Continuing.

Thread 2 "qemu-system-ris" hit Breakpoint 3, get_physical_address (
    env=0x6434914bf040, physical=0x79cbd5ee21f0, prot=0x79cbd5ee21e4,
    addr=18446744072637907160, access_type=2, mmu_idx=1)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:158
158     {
(gdb) print env->priv
$11 = 1 # S模式✅
(gdb) print /x env->satp
$12 = 0x8000000000080206 # satp 不为0，表示分页已启用✅
```

```
(gdb) print /x env->satp & 0xFF
$13 = 0x6
(gdb) print /x addr
$14 = 0xffffffffc02000d8 # 高地址，是用户空间地址✅
```

现在条件都满足了，可以开始调试了，输入下述命令：



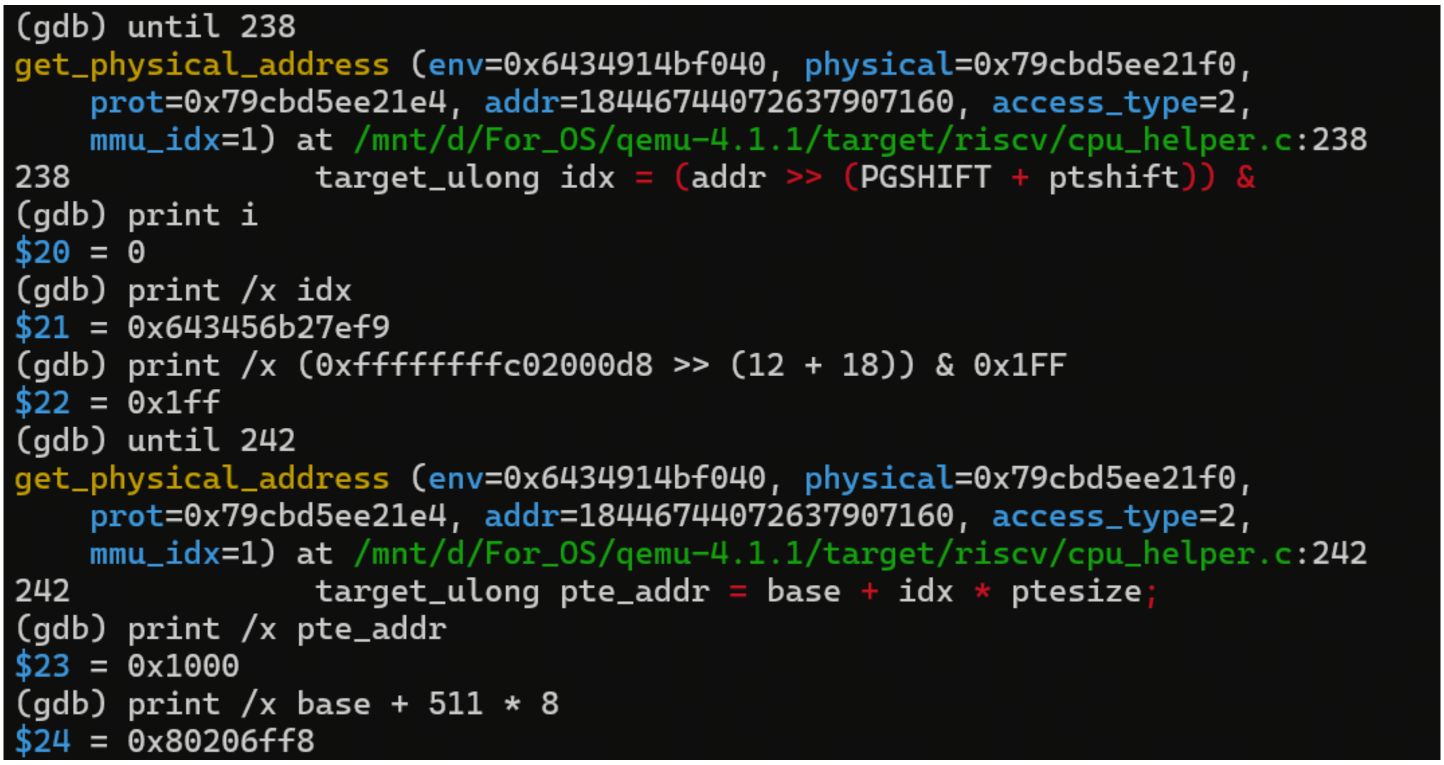
```
(gdb) delete 3
(gdb) break cpu_helper.c:237
Breakpoint 4 at 0x6434568d440d: file /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c, line 237.
(gdb) c
Continuing.

Thread 2 "qemu-system-ris" hit Breakpoint 4, get_physical_address (
    env=0x6434914bf040, physical=0x79cbd5ee21f0, prot=0x79cbd5ee21e4,
    addr=18446744072637907160, access_type=2, mmu_idx=1)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:237
237         for (i = 0; i < levels; i++, ptshift -= ptidxbits) {
(gdb) print i
$15 = -705808704
(gdb) print ptshift
$16 = 18
(gdb) print levels
$17 = 3
(gdb) print ptidxbits
$18 = 9
(gdb) print /x base
$19 = 0x80206000
```

代码块

```
(gdb) delete 3 #删除原有断点
(gdb) break cpu_helper.c:237 #设置新断点：get_physical_address()的for循环处
Breakpoint 4 at 0x6434568d440d: file /mnt/d/For_OS/qemu-
4.1.1/target/riscv/cpu_helper.c, line 237.
(gdb) c
Continuing.

Thread 2 "qemu-system-ris" hit Breakpoint 4, get_physical_address (
    env=0x6434914bf040, physical=0x79cbd5ee21f0, prot=0x79cbd5ee21e4,
    addr=18446744072637907160, access_type=2, mmu_idx=1)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:237
237         for (i = 0; i < levels; i++, ptshift -= ptidxbits) {
#查看循环参数：
(gdb) print i
$15 = -705808704 # 奇奇怪怪：此时循环还没开始，i还未被赋初值，也合理
(gdb) print ptshift
$16 = 18 # levels=3，(3-1)*9=18，表示第一级偏移，正确！
(gdb) print levels
```

```
$17 = 3  # levels=3，确实是SV39三级页表，正确!
(gdb) print ptidxbits
$18 = 9  # 每级9位索引，正确!
(gdb) print /x base
$19 = 0x80206000 # 页表基地址
```

继续调试：



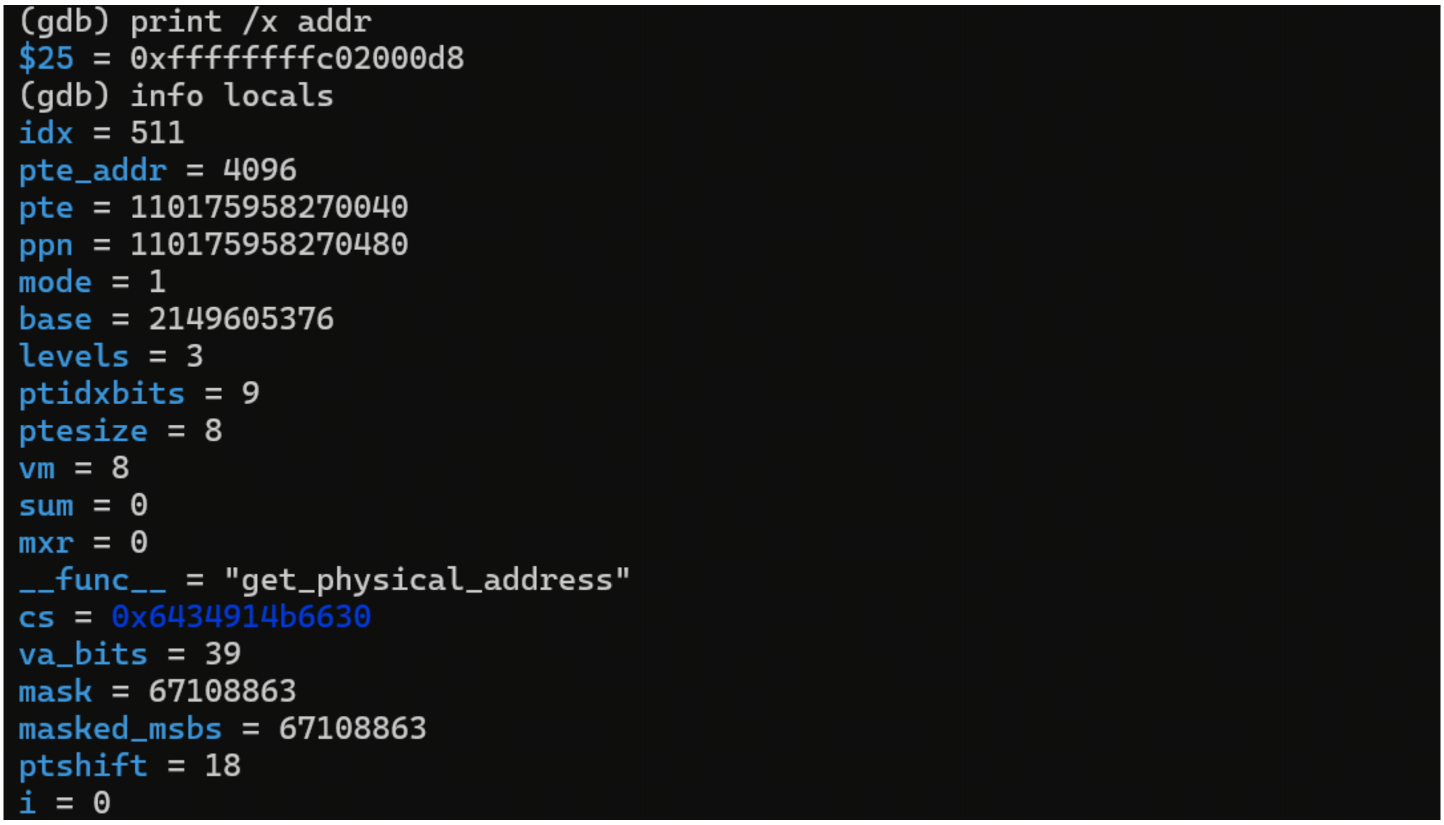
```
(gdb) until 238
get_physical_address (env=0x6434914bf040, physical=0x79cbd5ee21f0,
    prot=0x79cbd5ee21e4, addr=18446744072637907160, access_type=2,
    mmu_idx=1) at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:238
238             target_ulong idx = (addr >> (PGSHIFT + ptshift)) &
(gdb) print i
$20 = 0
(gdb) print /x idx
$21 = 0x643456b27ef9
(gdb) print /x (0xffffffffc02000d8 >> (12 + 18)) & 0x1FF
$22 = 0x1ff
(gdb) until 242
get_physical_address (env=0x6434914bf040, physical=0x79cbd5ee21f0,
    prot=0x79cbd5ee21e4, addr=18446744072637907160, access_type=2,
    mmu_idx=1) at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:242
242             target_ulong pte_addr = base + idx * ptesize;
(gdb) print /x pte_addr
$23 = 0x1000
(gdb) print /x base + 511 * 8
$24 = 0x80206ff8
```

代码块

```
(gdb) until 238 #继续执行到计算 idx 的地方
get_physical_address (env=0x6434914bf040, physical=0x79cbd5ee21f0,
    prot=0x79cbd5ee21e4, addr=18446744072637907160, access_type=2,
    mmu_idx=1) at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:238
238             target_ulong idx = (addr >> (PGSHIFT + ptshift)) &
(gdb) print i
$20 = 0 #这回 i = 0，正确!
(gdb) print /x idx
$21 = 0x643456b27ef9
(gdb) print /x (0xffffffffc02000d8 >> (12 + 18)) & 0x1FF # 验证计算
$22 = 0x1ff
(gdb) until 242 #继续到计算 PTE 地址
get_physical_address (env=0x6434914bf040, physical=0x79cbd5ee21f0,
    prot=0x79cbd5ee21e4, addr=18446744072637907160, access_type=2,
    mmu_idx=1) at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:242
242             target_ulong pte_addr = base + idx * ptesize;
(gdb) print /x pte_addr
$23 = 0x1000
(gdb) print /x base + 511 * 8 #计算期望的 PTE 地址
```

```
$24 = 0x80206ff8
```

我们再确认一下当前的状态：



```
(gdb) print /x addr
$25 = 0xffffffffc02000d8
(gdb) info locals
idx = 511
pte_addr = 4096
pte = 110175958270040
ppn = 110175958270480
mode = 1
base = 2149605376
levels = 3
ptidxbits = 9
ptesize = 8
vm = 8
sum = 0
mxr = 0
__func__ = "get_physical_address"
cs = 0x6434914b6630
va_bits = 39
mask = 67108863
masked_msbs = 67108863
ptshift = 18
i = 0
```

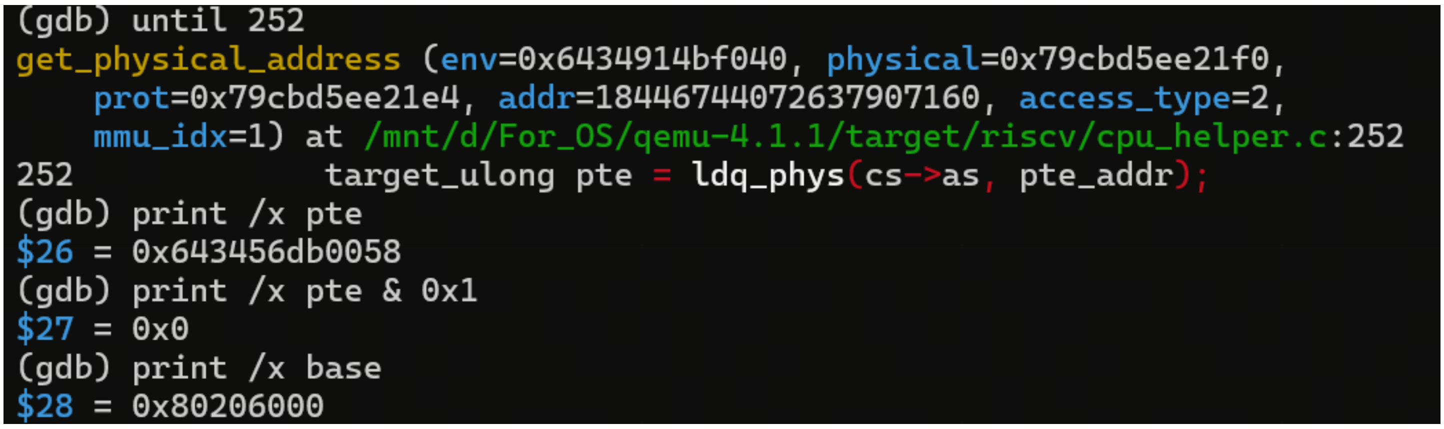
代码块

```
(gdb) print /x addr
$25 = 0xffffffffc02000d8 #正确! ✅
(gdb) info locals
idx = 511 #VPN[2] = 511, 正确! ✅
pte_addr = 4096 #应该是base + idx * ptesize = 0x80206FF8, 有问题
pte = 110175958270040
ppn = 110175958270480
mode = 1
base = 2149605376
levels = 3
ptidxbits = 9
ptesize = 8
vm = 8
sum = 0
mxr = 0
__func__ = "get_physical_address"
cs = 0x6434914b6630
va_bits = 39
mask = 67108863
masked_msbs = 67108863
ptshift = 18
```

```
i = 0
```

现在 `pte_addr` 有问题，这说明GDB显示的 `pte_addr` 不是当前值，可能是变量还未被赋值，或GDB显示的是旧值，或实际内存地址不同。

我们继续努力：

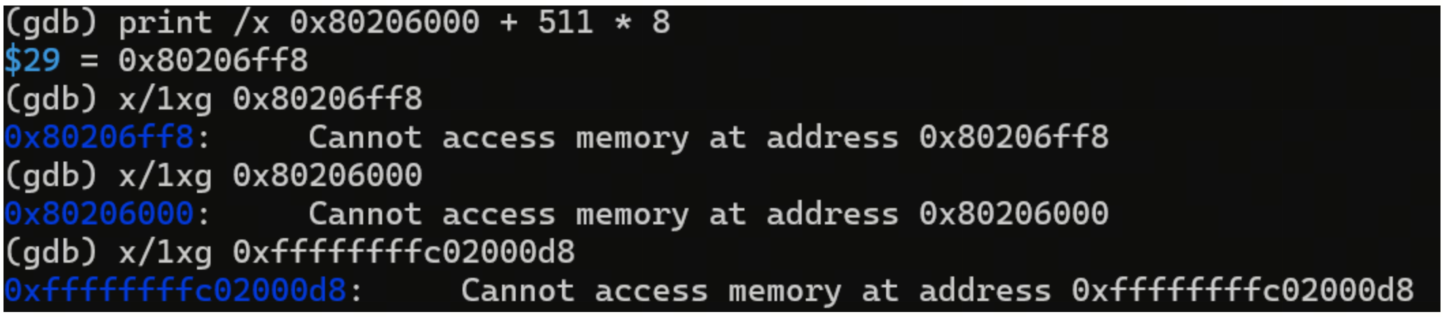


```
(gdb) until 252
get_physical_address (env=0x6434914bf040, physical=0x79cbd5ee21f0,
    prot=0x79cbd5ee21e4, addr=18446744072637907160, access_type=2,
    mmu_idx=1) at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:252
252             target_ulong pte = ldq_phys(cs->as, pte_addr);
(gdb) print /x pte
$26 = 0x643456db0058
(gdb) print /x pte & 0x1
$27 = 0x0
(gdb) print /x base
$28 = 0x80206000
```

代码块

```
(gdb) until 252 #继续到读取PTE后
get_physical_address (env=0x6434914bf040, physical=0x79cbd5ee21f0,
    prot=0x79cbd5ee21e4, addr=18446744072637907160, access_type=2,
    mmu_idx=1) at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:252
252             target_ulong pte = ldq_phys(cs->as, pte_addr);
(gdb) print /x pte #查看PTE内容
$26 = 0x643456db0058
(gdb) print /x pte & 0x1 #检查是否有效
$27 = 0x0
(gdb) print /x base #如果是中间PTE，查看base是否更新
$28 = 0x80206000
```

我们发现了关键问题：PTE无效……当前pte的值为0x643456db0058，V位为0，表示无效。R|W|X位：从值看可能都不为0，但因为V=0，所以无效。这就意味着：页表遍历失败，会触发页面错误，OS需要建立这个地址的页表映射。虽然PTE无效，但我们可以验证地址：



```
(gdb) print /x 0x80206000 + 511 * 8
$29 = 0x80206ff8
(gdb) x/1xg 0x80206ff8
0x80206ff8:     Cannot access memory at address 0x80206ff8
(gdb) x/1xg 0x80206000
0x80206000:     Cannot access memory at address 0x80206000
(gdb) x/1xg 0xffffffffc02000d8
0xffffffffc02000d8:     Cannot access memory at address 0xffffffffc02000d8
```

代码块

```
(gdb) print /x 0x80206000 + 511 * 8 #查看PTE地址是否正确
```

```
$29 = 0x80206ff8
(gdb) x/1xg 0x80206ff8 #查看内存是否可访问
0x80206ff8:     Cannot access memory at address 0x80206ff8
```

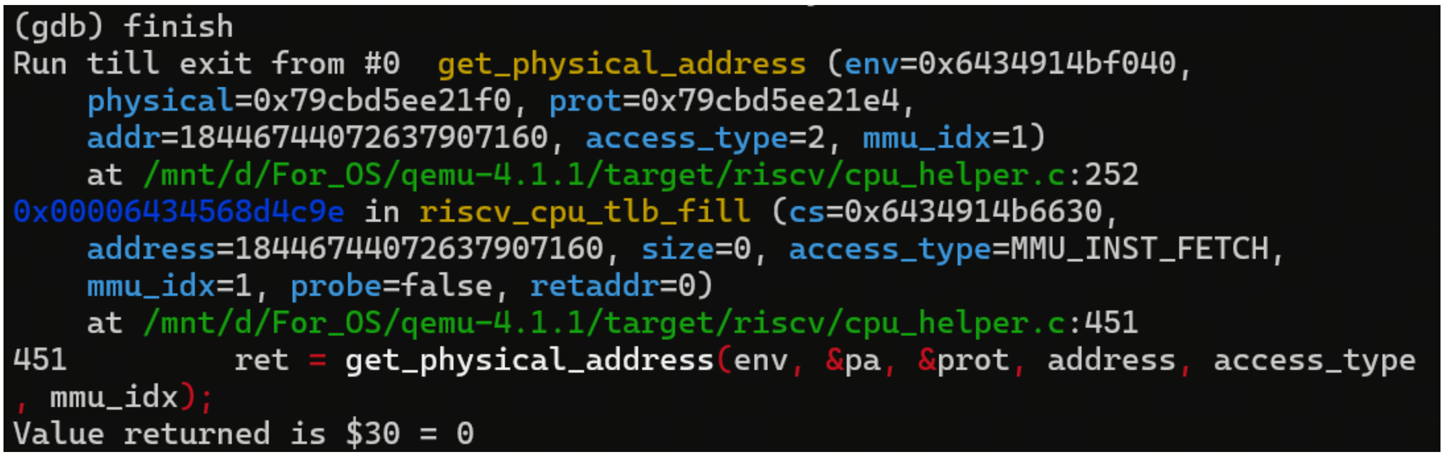
我们又发现了关键问题：内存访问失败。这说明物理地址 `0x80206ff8` 不存在或不可访问，QEMU无法访问guest的物理内存，可能页表所在的物理内存区域未映射。我又试了其他几个之前出现过的地址，都是无法访问。

以下是大模型为我分析的原因：

1. 早期启动阶段：页表物理内存还没被正确映射到QEMU的地址空间
2. PMP限制：物理内存保护阻止了访问
3. QEMU内存管理：某些物理内存区域对host不可直接访问

到这里本来都想放弃了，但我注意到之前大模型给出的一句话”OS需要建立这个地址的页表映射“，那我们就让它去建立这个映射，然后记录调试过程就可以了呀！

向大模型求助后，它让我用finish命令从当前get_physical_address函数中跳出来：


```
(gdb) finish
Run till exit from #0  get_physical_address (env=0x6434914bf040,
    physical=0x79cbd5ee21f0, prot=0x79cbd5ee21e4,
    addr=18446744072637907160, access_type=2, mmu_idx=1)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:252
0x00006434568d4c9e in riscv_cpu_tlb_fill (cs=0x6434914b6630,
    address=18446744072637907160, size=0, access_type=MMU_INST_FETCH,
    mmu_idx=1, probe=false, retaddr=0)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:451
451         ret = get_physical_address(env, &pa, &prot, address, access_type
, mmu_idx);
Value returned is $30 = 0
```

代码块

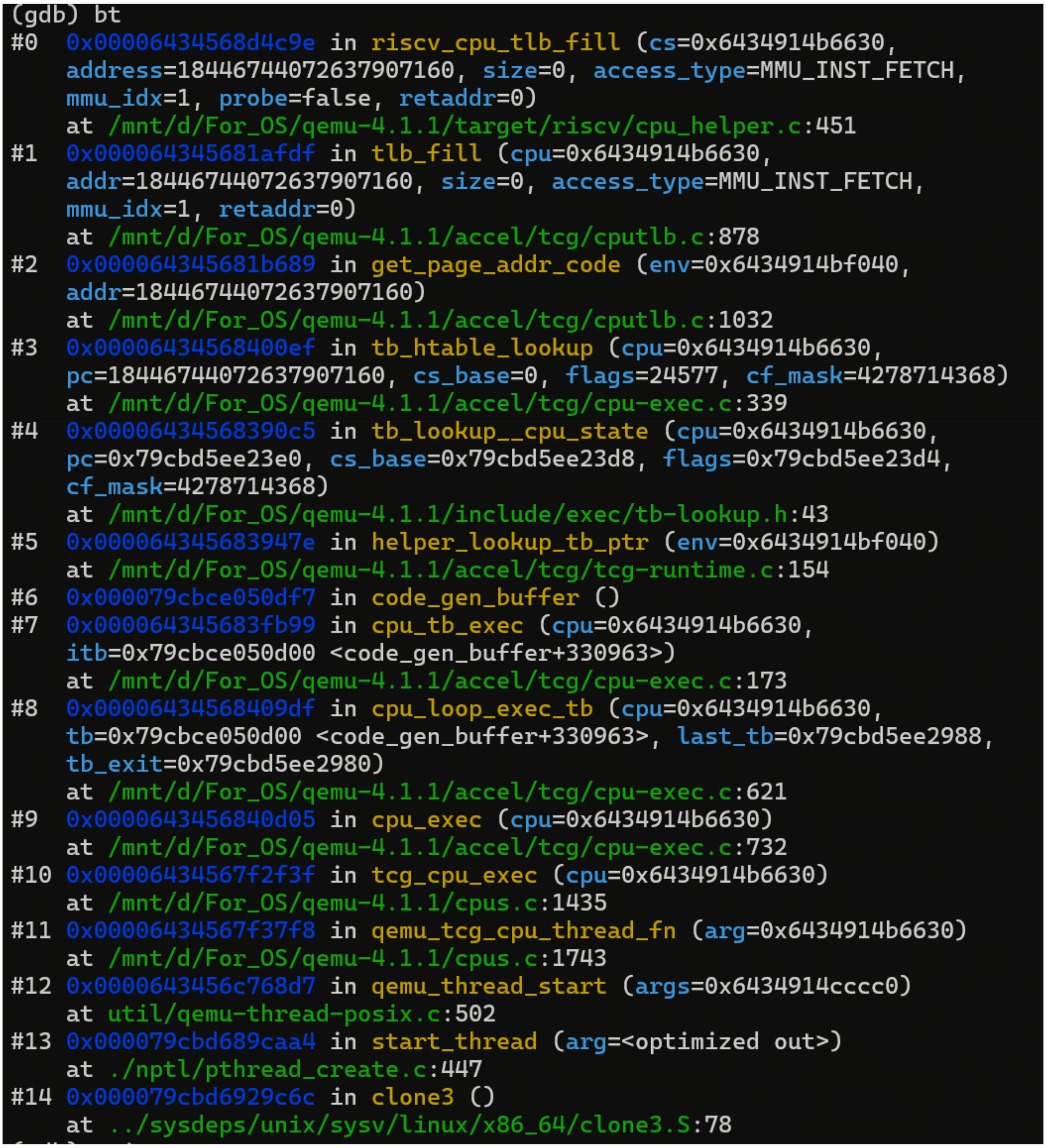
```
(gdb) finish
Run till exit from #0  get_physical_address (env=0x6434914bf040,
    physical=0x79cbd5ee21f0, prot=0x79cbd5ee21e4,
    addr=18446744072637907160, access_type=2, mmu_idx=1)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:252
0x00006434568d4c9e in riscv_cpu_tlb_fill (cs=0x6434914b6630,
    address=18446744072637907160, size=0, access_type=MMU_INST_FETCH,
    mmu_idx=1, probe=false, retaddr=0)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:451
451         ret = get_physical_address(env, &pa, &prot, address, access_type,
mmu_idx);
Value returned is $30 = 0 #返回结果为0，表示还是成功了
```

通过执行 `finish` 命令，我们从正在单步调试的 `get_physical_address` 函数中返回到其调用者 `riscv_cpu_tlb_fill`。输出显示：

1. `get_physical_address` 函数返回值为 `0` （`$30 = 0`）
2. 这个返回值对应源码中的 `TRANSLATE_SUCCESS`，表示页表遍历成功
3. 返回到 `riscv_cpu_tlb_fill` 函数的第451行，这是调用 `get_physical_address` 之后的位置
4. 传入的参数包括：虚拟地址 `18446744072637907160` （即 `0xffffffffc02000d8`）、访问类型 `MMU_INST_FETCH` （值为2，表示指令获取）、MMU索引 `1` （S模式）
5. 返回值 `0` 将被存储在局部变量 `ret` 中，后续代码会根据这个值决定是建立TLB映射还是触发异常

这表明尽管之前观察到的页表项（PTE）的V位为0，但在实际的页表遍历过程中，该地址的翻译最终成功完成。可能是由于多级页表遍历找到了有效的最终页表项，或者在并发环境下页表被其他CPU核心更新。

然后我输入bt命令，查看当前的调用栈：



```
(gdb) bt
#0  0x00006434568d4c9e in riscv_cpu_tlb_fill (cs=0x6434914b6630,
    address=18446744072637907160, size=0, access_type=MMU_INST_FETCH,
    mmu_idx=1, probe=false, retaddr=0)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:451
#1  0x000064345681afdf in tlb_fill (cpu=0x6434914b6630,
    addr=18446744072637907160, size=0, access_type=MMU_INST_FETCH,
    mmu_idx=1, retaddr=0)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cputlb.c:878
#2  0x000064345681b689 in get_page_addr_code (env=0x6434914bf040,
    addr=18446744072637907160)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cputlb.c:1032
#3  0x00006434568400ef in tb_htable_lookup (cpu=0x6434914b6630,
    pc=18446744072637907160, cs_base=0, flags=24577, cf_mask=4278714368)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cpu-exec.c:339
#4  0x00006434568390c5 in tb_lookup__cpu_state (cpu=0x6434914b6630,
    pc=0x79cbd5ee23e0, cs_base=0x79cbd5ee23d8, flags=0x79cbd5ee23d4,
    cf_mask=4278714368)
    at /mnt/d/For_OS/qemu-4.1.1/include/exec/tb-lookup.h:43
#5  0x000064345683947e in helper_lookup_tb_ptr (env=0x6434914bf040)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/tcg-runtime.c:154
#6  0x000079cbce050df7 in code_gen_buffer ()
#7  0x0000643456883fb99 in cpu_tb_exec (cpu=0x6434914b6630,
    itb=0x79cbce050d00 <code_gen_buffer+330963>)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cpu-exec.c:173
#8  0x00006434568409df in cpu_loop_exec_tb (cpu=0x6434914b6630,
    tb=0x79cbce050d00 <code_gen_buffer+330963>, last_tb=0x79cbd5ee2988,
    tb_exit=0x79cbd5ee2980)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cpu-exec.c:621
#9  0x0000643456840d05 in cpu_exec (cpu=0x6434914b6630)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cpu-exec.c:732
#10 0x00006434567f2f3f in tcg_cpu_exec (cpu=0x6434914b6630)
    at /mnt/d/For_OS/qemu-4.1.1/cpus.c:1435
#11 0x00006434567f37f8 in qemu_tcg_cpu_thread_fn (arg=0x6434914b6630)
    at /mnt/d/For_OS/qemu-4.1.1/cpus.c:1743
#12 0x0000643456c768d7 in qemu_thread_start (args=0x6434914cccc0)
    at util/qemu-thread-posix.c:502
#13 0x000079cbd689caa4 in start_thread (arg=<optimized out>)
    at ./nptl/pthread_create.c:447
#14 0x000079cbd6929c6c in clone3 ()
    at ../sysdeps/unix/sysv/linux/x86_64/clone3.S:78
```

代码块

```
(gdb) bt
#0  0x00006434568d4c9e in riscv_cpu_tlb_fill (cs=0x6434914b6630,
    address=18446744072637907160, size=0, access_type=MMU_INST_FETCH,
    mmu_idx=1, probe=false, retaddr=0)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:451
#1  0x000064345681afdf in tlb_fill (cpu=0x6434914b6630,
    addr=18446744072637907160, size=0, access_type=MMU_INST_FETCH,
    mmu_idx=1, retaddr=0)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cputlb.c:878
```

```
#2  0x000064345681b689 in get_page_addr_code (env=0x6434914bf040,
    addr=18446744072637907160)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cputlb.c:1032
#3  0x00006434568400ef in tb_htable_lookup (cpu=0x6434914b6630,
    pc=18446744072637907160, cs_base=0, flags=24577, cf_mask=4278714368)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cpu-exec.c:339
#4  0x00006434568390c5 in tb_lookup__cpu_state (cpu=0x6434914b6630,
    pc=0x79cbd5ee23e0, cs_base=0x79cbd5ee23d8, flags=0x79cbd5ee23d4,
    cf_mask=4278714368)
    at /mnt/d/For_OS/qemu-4.1.1/include/exec/tb-lookup.h:43
#5  0x000064345683947e in helper_lookup_tb_ptr (env=0x6434914bf040)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/tcg-runtime.c:154
#6  0x000079cbce050df7 in code_gen_buffer ()
#7  0x000064345683fb99 in cpu_tb_exec (cpu=0x6434914b6630,
    itb=0x79cbce050d00 <code_gen_buffer+330963>)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cpu-exec.c:173
#8  0x00006434568409df in cpu_loop_exec_tb (cpu=0x6434914b6630,
    tb=0x79cbce050d00 <code_gen_buffer+330963>, last_tb=0x79cbd5ee2988,
    tb_exit=0x79cbd5ee2980)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cpu-exec.c:621
#9  0x0000643456840d05 in cpu_exec (cpu=0x6434914b6630)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cpu-exec.c:732
#10 0x00006434567f2f3f in tcg_cpu_exec (cpu=0x6434914b6630)
    at /mnt/d/For_OS/qemu-4.1.1/cpus.c:1435
#11 0x00006434567f37f8 in qemu_tcg_cpu_thread_fn (arg=0x6434914b6630)
    at /mnt/d/For_OS/qemu-4.1.1/cpus.c:1743
#12 0x0000643456c768d7 in qemu_thread_start (args=0x6434914cccc0)
    at util/qemu-thread-posix.c:502
#13 0x000079cbd689caa4 in start_thread (arg=<optimized out>)
    at ./nptl/pthread_create.c:447
#14 0x000079cbd6929c6c in clone3 ()
    at ../sysdeps/unix/sysv/linux/x86_64/clone3.S:78
```

对于上述调用栈：

底层系统调用层（#13-#14）：

- QEMU用户态线程的创建和调度（`start_thread`，`clone3`）

QEMU线程调度层（#10-#12）：

- QEMU的TCG（Tiny Code Generator）CPU执行循环
- `qemu_tcg_cpu_thread_fn`：QEMU的TCG CPU线程主函数
- `tcg_cpu_exec`：TCG模式的CPU执行入口

CPU执行核心层（#7-#9）：

- `cpu_exec`：CPU主执行循环

- `cpu_loop_exec_tb`：执行翻译块（Translation Block）
- `cpu_tb_exec`：执行单个翻译块

代码生成与查找层（#3-#6）：

- `helper_lookup_tb_ptr`：帮助函数查找翻译块指针
- `tb_lookup__cpu_state`：根据CPU状态查找翻译块
- `tb_htable_lookup`：在翻译块哈希表中查找
- `code_gen_buffer`：QEMU动态生成的机器代码区域

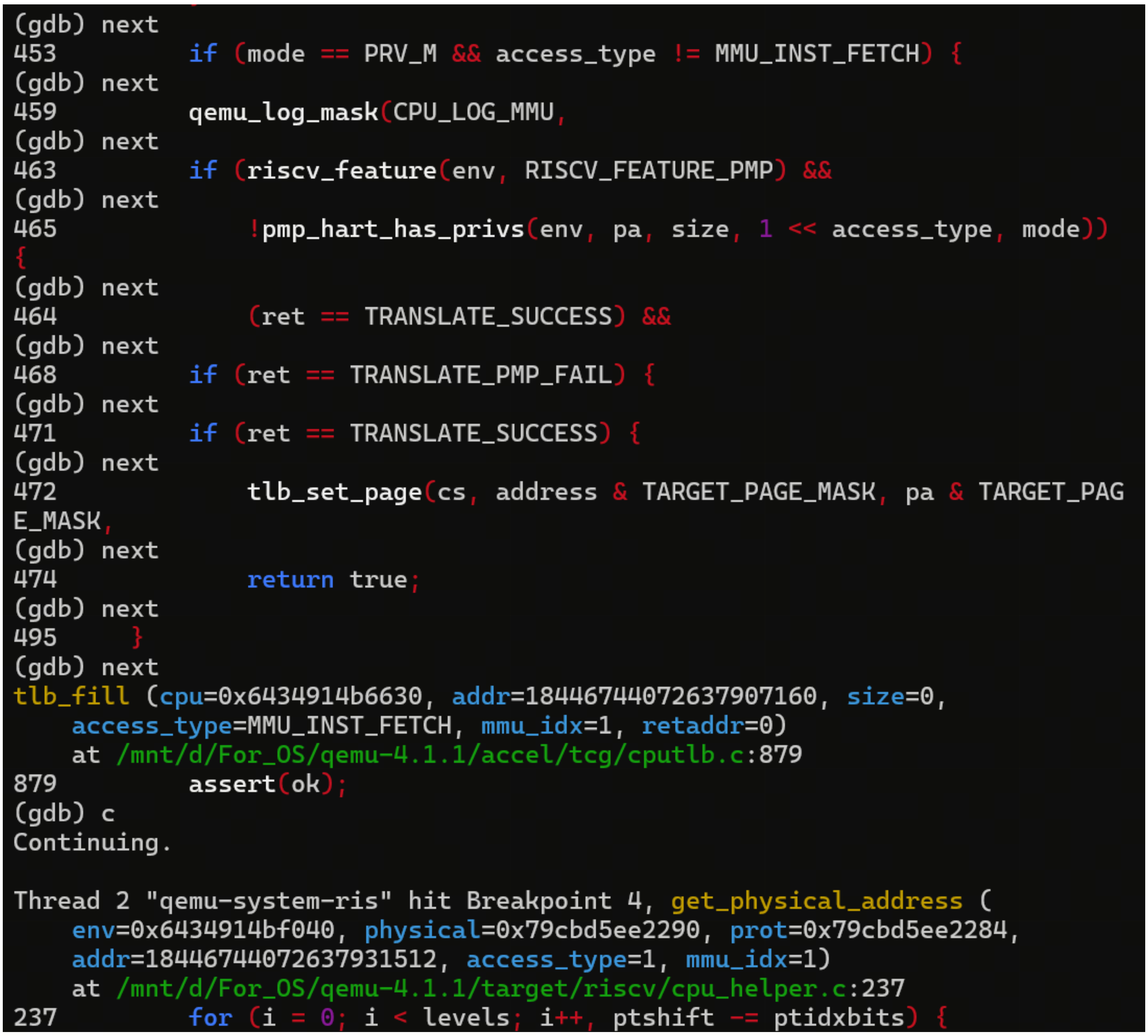
内存管理/TLB层（#0-#2）：（核心）

- `get_page_addr_code`：获取代码页的物理地址
- `tlb_fill`：TLB填充函数，处理TLB缺失
- `riscv_cpu_tlb_fill`：RISC-V架构特定的TLB填充实现（当前执行位置）

上述信息说明：

1. 当前正执行在 `riscv_cpu_tlb_fill` 中（栈帧#0），这是处理虚拟地址翻译的入口
2. 触发原因是获取代码页地址（`get_page_addr_code`），说明是指令获取导致的页表访问
3. 虚拟地址为 `0xffffffffc02000d8`，位于内核空间
4. 调用链展示了QEMU从**执行用户代码→翻译块管理→内存访问→页表遍历**的完整路径
5. 这表明操作系统内核代码执行时遇到了**页表缺失**，需要建立映射

接下来我输入多个next（next会执行当前代码，遇到函数时会执行整个函数，并在函数返回后暂停）：



```
(gdb) next
453         if (mode == PRV_M && access_type != MMU_INST_FETCH) {
(gdb) next
459         qemu_log_mask(CPU_LOG_MMU,
(gdb) next
463         if (riscv_feature(env, RISCV_FEATURE_PMP) &&
(gdb) next
465             !pmp_hart_has_privs(env, pa, size, 1 << access_type, mode))
{
(gdb) next
464             (ret == TRANSLATE_SUCCESS) &&
(gdb) next
468         if (ret == TRANSLATE_PMP_FAIL) {
(gdb) next
471         if (ret == TRANSLATE_SUCCESS) {
(gdb) next
472             tlb_set_page(cs, address & TARGET_PAGE_MASK, pa & TARGET_PAG
E_MASK,
(gdb) next
474             return true;
(gdb) next
495     }
(gdb) next
tlb_fill (cpu=0x6434914b6630, addr=18446744072637907160, size=0,
    access_type=MMU_INST_FETCH, mmu_idx=1, retaddr=0)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cputlb.c:879
879         assert(ok);
(gdb) c
Continuing.

Thread 2 "qemu-system-ris" hit Breakpoint 4, get_physical_address (
    env=0x6434914bf040, physical=0x79cbd5ee2290, prot=0x79cbd5ee2284,
    addr=18446744072637931512, access_type=1, mmu_idx=1)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:237
237         for (i = 0; i < levels; i++, ptshift -= ptidxbits) {
```

代码块

```
(gdb) next
453         if (mode == PRV_M && access_type != MMU_INST_FETCH) {
(gdb) next
459         qemu_log_mask(CPU_LOG_MMU,
(gdb) next
463         if (riscv_feature(env, RISCV_FEATURE_PMP) &&
(gdb) next
465             !pmp_hart_has_privs(env, pa, size, 1 << access_type, mode)) {
(gdb) next
464             (ret == TRANSLATE_SUCCESS) &&
(gdb) next
468         if (ret == TRANSLATE_PMP_FAIL) {
(gdb) next
471         if (ret == TRANSLATE_SUCCESS) {
(gdb) next
```

```
472             tlb_set_page(cs, address & TARGET_PAGE_MASK, pa &
TARGET_PAGE_MASK,
(gdb) next
474             return true;
(gdb) next
495     }
(gdb) next
tlb_fill (cpu=0x6434914b6630, addr=18446744072637907160, size=0,
    access_type=MMU_INST_FETCH, mmu_idx=1, retaddr=0)
    at /mnt/d/For_OS/qemu-4.1.1/accel/tcg/cputlb.c:879
879         assert(ok); #TLB填充成功
(gdb) c
Continuing.

Thread 2 "qemu-system-ris" hit Breakpoint 4, get_physical_address (
    env=0x6434914bf040, physical=0x79cbd5ee2290, prot=0x79cbd5ee2284,
    addr=18446744072637931512, access_type=1, mmu_idx=1)
    at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:237
237         for (i = 0; i < levels; i++, ptshift -= ptidxbits) {
```

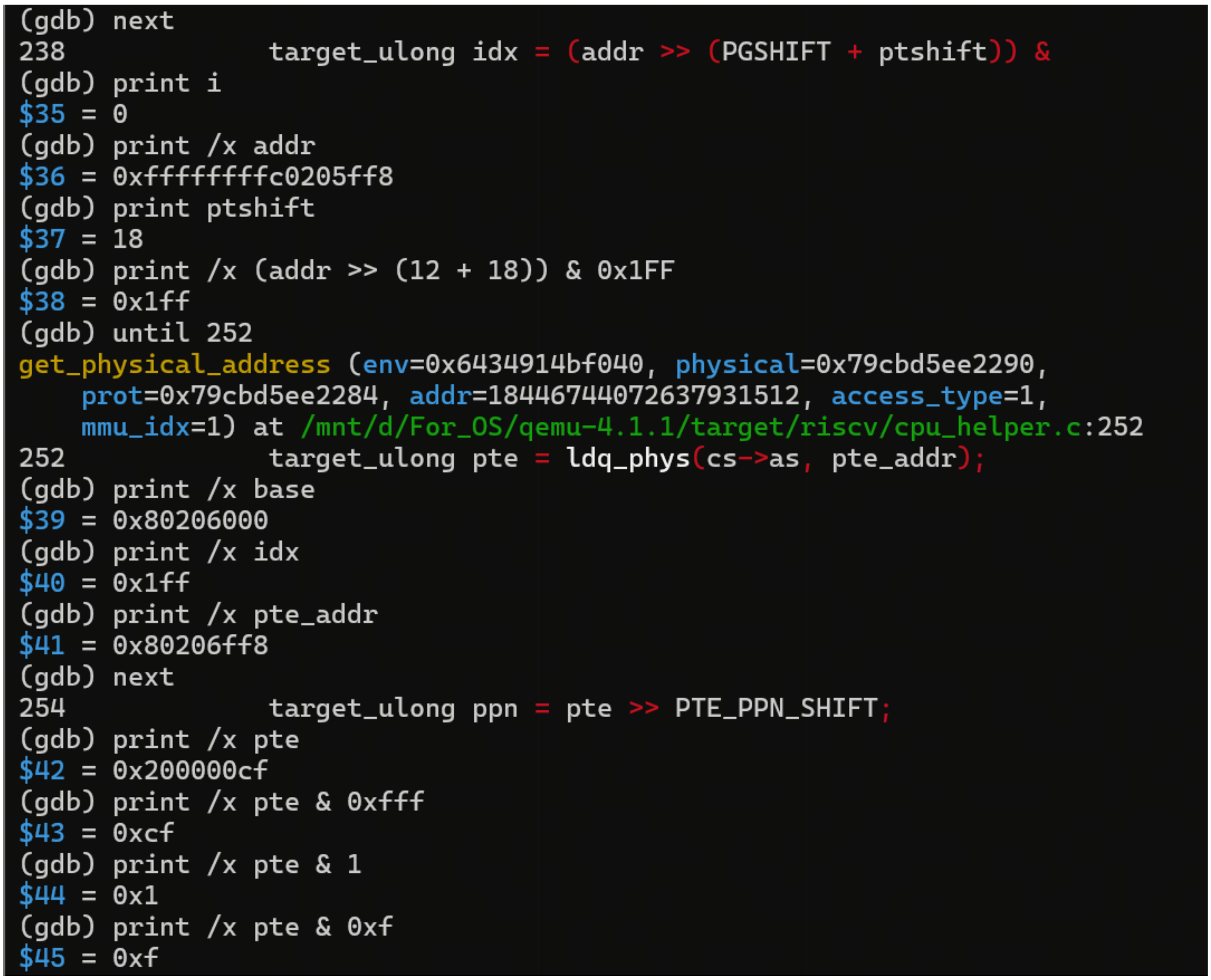
发现一直 `next` 下去，最终到了 `tlb_fill` 函数。这是因为 `riscv_cpu_tlb_fill` 返回后，控制流返回到它的调用者 `tlb_fill` 处。调用关系是：

代码块

```
tlb_fill()  # 通用TLB填充函数
  ↓
riscv_cpu_tlb_fill()  # RISC-V架构特定的实现
  ↓ (返回true)
tlb_fill()  # 继续执行879行
```

其中879行的 `assert(ok);` 表示TLB填充成功。接下来我们输入c继续执行，发现很快就遇到了新的内存访问（虚拟地址 `0xffffffffc02000f8`），触发了新的页表遍历，命中断点4，再次进入了 `get_physical_address`。

然后我们像之前一样继续去调试：



```
(gdb) next
238             target_ulong idx = (addr >> (PGSHIFT + ptshift)) &
(gdb) print i
$35 = 0
(gdb) print /x addr
$36 = 0xffffffffc0205ff8
(gdb) print ptshift
$37 = 18
(gdb) print /x (addr >> (12 + 18)) & 0x1FF
$38 = 0x1ff
(gdb) until 252
get_physical_address (env=0x6434914bf040, physical=0x79cbd5ee2290,
    prot=0x79cbd5ee2284, addr=18446744072637931512, access_type=1,
    mmu_idx=1) at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:252
252             target_ulong pte = ldq_phys(cs->as, pte_addr);
(gdb) print /x base
$39 = 0x80206000
(gdb) print /x idx
$40 = 0x1ff
(gdb) print /x pte_addr
$41 = 0x80206ff8
(gdb) next
254             target_ulong ppn = pte >> PTE_PPN_SHIFT;
(gdb) print /x pte
$42 = 0x200000cf
(gdb) print /x pte & 0xfff
$43 = 0xcf
(gdb) print /x pte & 1
$44 = 0x1
(gdb) print /x pte & 0xf
$45 = 0xf
```

代码块

```
(gdb) next #计算页表索引 idx: idx = (addr >> (12 + 18)) & 0x1FF
238             target_ulong idx = (addr >> (PGSHIFT + ptshift)) &
(gdb) print i #验证参数
$35 = 0       #i = 0, 表示在第0级页表
(gdb) print /x addr
$36 = 0xffffffffc0205ff8   #虚拟地址
(gdb) print ptshift
$37 = 18   #当前偏移量
(gdb) print /x (addr >> (12 + 18)) & 0x1FF
$38 = 0x1ff #计算得到索引511
(gdb) until 252 #计算PTE地址
get_physical_address (env=0x6434914bf040, physical=0x79cbd5ee2290,
    prot=0x79cbd5ee2284, addr=18446744072637931512, access_type=1,
    mmu_idx=1) at /mnt/d/For_OS/qemu-4.1.1/target/riscv/cpu_helper.c:252
252             target_ulong pte = ldq_phys(cs->as, pte_addr);
(gdb) print /x base
$39 = 0x80206000      #页表基址
(gdb) print /x idx
```

```
$40 = 0x1ff           #页表索引
(gdb) print /x pte_addr
$41 = 0x80206ff8      #PTE物理地址
# 计算: 0x80206000 + 511 * 8 = 0x80206000 + 0xFF8 = 0x80206ff8
(gdb) next
254             target_ulong ppn = pte >> PTE_PPN_SHIFT;
(gdb) print /x pte
$42 = 0x200000cf   #页表项值
(gdb) print /x pte & 0xfff
$43 = 0xcf         #低12位标志位
(gdb) print /x pte & 1
$44 = 0x1          #V位=1, 有效!
(gdb) print /x pte & 0xf
$45 = 0xf          #低4位: V=1, R=0, W=1, X=1
```

上述的步骤：计算页表索引→验证参数→计算PTE地址→读取PTE并分析。

PTE标志位详细分析：

```
代码块

0xcf = 1100 1111₂
位分解:
- 位0 (0x01): V = 1     有效
- 位1 (0x02): R = 0     不可读
- 位2 (0x04): W = 1     可写
- 位3 (0x08): X = 1     可执行
- 位4 (0x10): U = 0     超级用户
- 位5 (0x20): G = 0     非全局
- 位6 (0x40): A = 1     已访问
- 位7 (0x80): D = 1     已修改
```

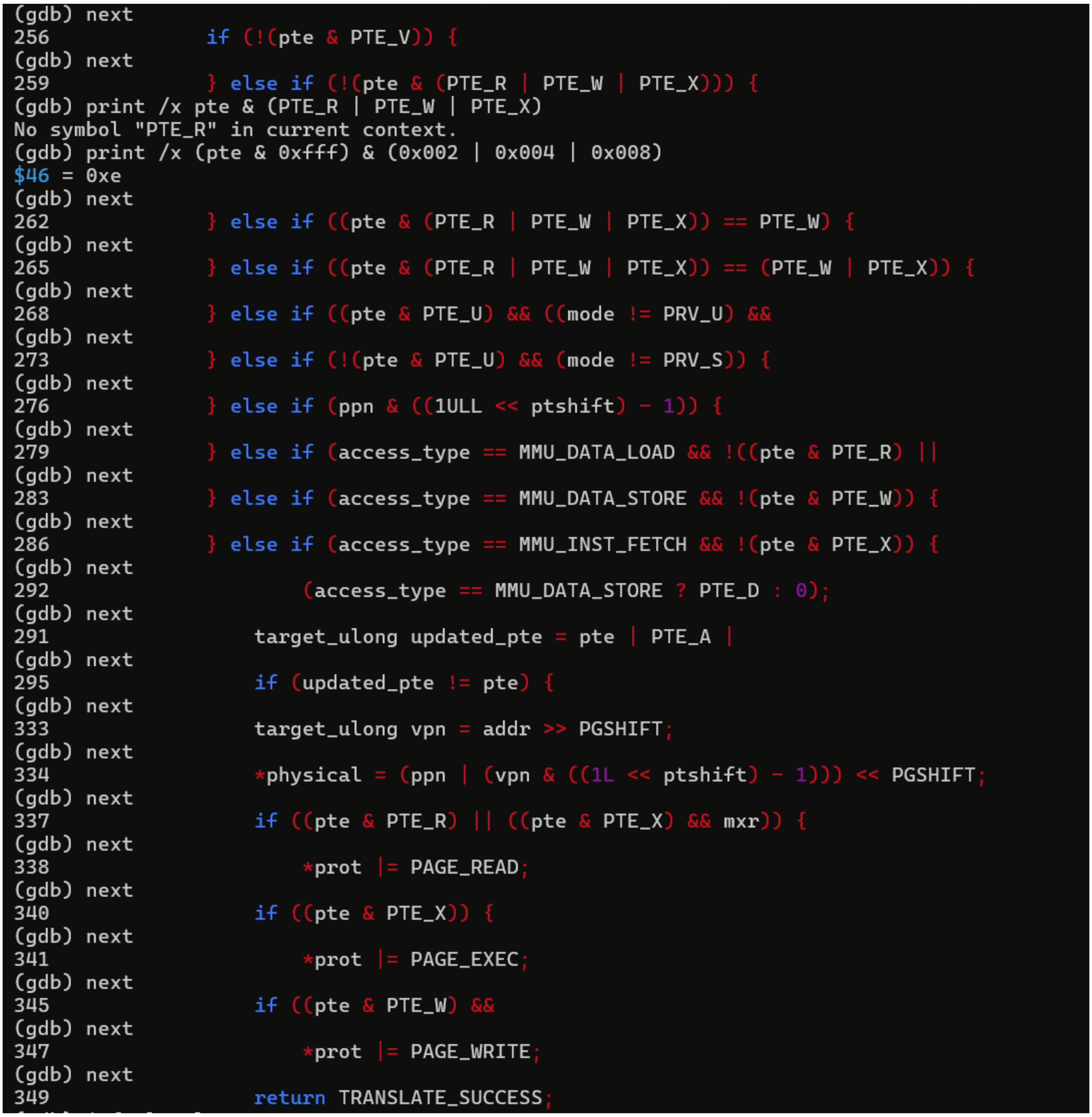
为什么之前PTE的V位是0，现在是1了：

```
代码块

时间点1: 访问 0xffffffffc02000d8 (指令获取)
  → PTE.V = 0 (无效)
  → 触发页错误异常 (Instruction Page Fault)

时间点2: 访问 0xffffffffc0205ff8 (数据加载)
  → PTE.V = 1 (有效)
  → 页表遍历成功
```

大模型说这个正常OS的行为：

1. **按需分配**：页面在第一次访问时才分配
2. **惰性加载**：页表项初始化为无效，触发异常时才建立映射
3. **写时复制**：某些页面初始为只读，写时触发异常再设为可写
4. **内存共享**：通过精心控制页表权限实现

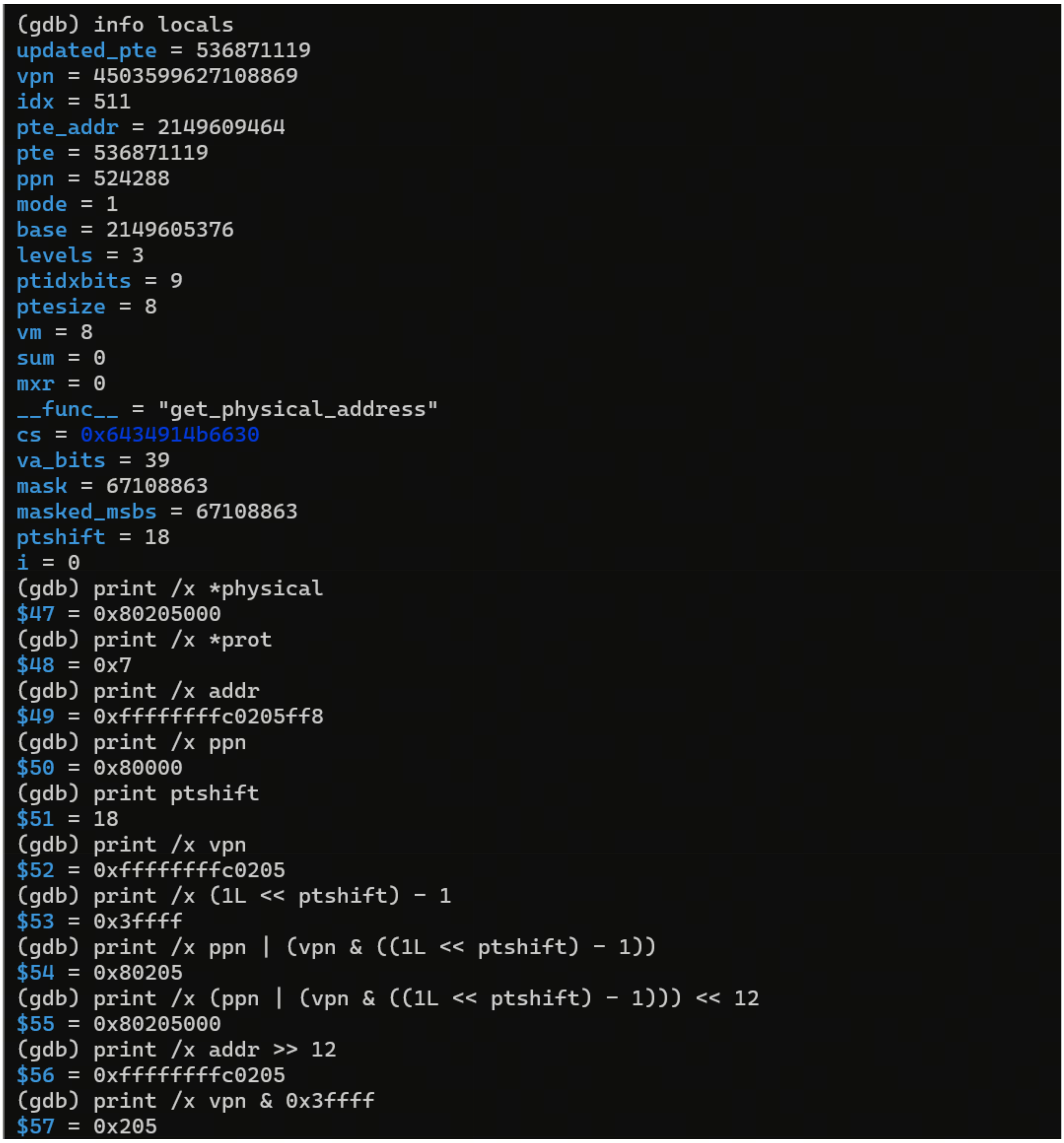
然后我又执行了很多步的next：



```
(gdb) next
256             if (!(pte & PTE_V)) {
(gdb) next
259             } else if (!(pte & (PTE_R | PTE_W | PTE_X))) {
(gdb) print /x pte & (PTE_R | PTE_W | PTE_X)
No symbol "PTE_R" in current context.
(gdb) print /x (pte & 0xfff) & (0x002 | 0x004 | 0x008)
$46 = 0xe
(gdb) next
262             } else if ((pte & (PTE_R | PTE_W | PTE_X)) == PTE_W) {
(gdb) next
265             } else if ((pte & (PTE_R | PTE_W | PTE_X)) == (PTE_W | PTE_X)) {
(gdb) next
268             } else if ((pte & PTE_U) && ((mode != PRV_U) &&
(gdb) next
273             } else if (!(pte & PTE_U) && (mode != PRV_S)) {
(gdb) next
276             } else if (ppn & ((1ULL << ptshift) - 1)) {
(gdb) next
279             } else if (access_type == MMU_DATA_LOAD && !((pte & PTE_R) ||
(gdb) next
283             } else if (access_type == MMU_DATA_STORE && !(pte & PTE_W)) {
(gdb) next
286             } else if (access_type == MMU_INST_FETCH && !(pte & PTE_X)) {
(gdb) next
292                     (access_type == MMU_DATA_STORE ? PTE_D : 0);
(gdb) next
291                 target_ulong updated_pte = pte | PTE_A |
(gdb) next
295                 if (updated_pte != pte) {
(gdb) next
333                 target_ulong vpn = addr >> PGSHIFT;
(gdb) next
334                 *physical = (ppn | (vpn & ((1L << ptshift) - 1))) << PGSHIFT;
(gdb) next
337                 if ((pte & PTE_R) || ((pte & PTE_X) && mxr)) {
(gdb) next
338                     *prot |= PAGE_READ;
(gdb) next
340                 if ((pte & PTE_X)) {
(gdb) next
341                     *prot |= PAGE_EXEC;
(gdb) next
345                 if ((pte & PTE_W) &&
(gdb) next
347                     *prot |= PAGE_WRITE;
(gdb) next
349                 return TRANSLATE_SUCCESS;
```

可以看到，这次程序跳过了所有失败分支，执行了成功路径，最终返回的是 `return TRANSLATE_SUCCESS;` 说明这回的虚拟地址到物理地址的翻译成功了。

我们看一下现在的参数值：



```
(gdb) info locals
updated_pte = 536871119
vpn = 4503599627108869
idx = 511
pte_addr = 2149609464
pte = 536871119
ppn = 524288
mode = 1
base = 2149605376
levels = 3
ptidxbits = 9
ptesize = 8
vm = 8
sum = 0
mxr = 0
__func__ = "get_physical_address"
cs = 0x6434914b6630
va_bits = 39
mask = 67108863
masked_msbs = 67108863
ptshift = 18
i = 0
(gdb) print /x *physical
$47 = 0x80205000
(gdb) print /x *prot
$48 = 0x7
(gdb) print /x addr
$49 = 0xffffffffc0205ff8
(gdb) print /x ppn
$50 = 0x80000
(gdb) print ptshift
$51 = 18
(gdb) print /x vpn
$52 = 0xfffffffffc0205
(gdb) print /x (1L << ptshift) - 1
$53 = 0x3ffff
(gdb) print /x ppn | (vpn & ((1L << ptshift) - 1))
$54 = 0x80205
(gdb) print /x (ppn | (vpn & ((1L << ptshift) - 1))) << 12
$55 = 0x80205000
(gdb) print /x addr >> 12
$56 = 0xfffffffffc0205
(gdb) print /x vpn & 0x3ffff
$57 = 0x205
```

代码块

```
(gdb) info locals
updated_pte = 536871119
vpn = 4503599627108869
idx = 511
pte_addr = 2149609464
pte = 536871119
ppn = 524288
mode = 1
base = 2149605376
levels = 3
```

```
ptidxbits = 9
ptesize = 8
vm = 8
sum = 0
mxr = 0
__func__ = "get_physical_address"
cs = 0x6434914b6630
va_bits = 39
mask = 67108863
masked_msbs = 67108863
ptshift = 18
i = 0
(gdb) print /x *physical
$47 = 0x80205000
(gdb) print /x *prot
$48 = 0x7
(gdb) print /x addr
$49 = 0xffffffffc0205ff8
(gdb) print /x ppn
$50 = 0x80000
(gdb) print ptshift
$51 = 18
(gdb) print /x vpn
$52 = 0xffffffffc0205
(gdb) print /x (1L << ptshift) - 1
$53 = 0x3ffff
(gdb) print /x ppn | (vpn & ((1L << ptshift) - 1))
$54 = 0x80205
(gdb) print /x (ppn | (vpn & ((1L << ptshift) - 1))) << 12
$55 = 0x80205000
(gdb) print /x addr >> 12
$56 = 0xffffffffc0205
(gdb) print /x vpn & 0x3ffff
$57 = 0x205
```

核心参数验证：

参数	值（十六进制）	解释	正确性
虚拟地址	0xffffffffc0205ff8	要翻译的地址	✅
PTE值	0x200000cf (536871119)	页表项	✅
PPN	0x80000 (524288)	物理页号	✅ (从PTE提取)
物理地址	0x80205000	计算结果	✅
权限	0x7	读+写+执行	✅

代码块

```
PTE = 0x200000cf (PTE是页表项)
PPN = (pte >> 10) & 0x3fffffffff = 0x80000  # ✅ (PPN是物理页号)
VPN偏移 = (addr >> 12) & 0x3ffff = 0x205 (多级页表遍历过程中，当前级别页表索引未覆盖的那部分虚拟地址位)
物理页号 = PPN | VPN偏移 = 0x80000 | 0x205 = 0x80205
物理地址 = 物理页号 << 12 = 0x80205000  # ✅ 与输出一致!
```

以上我们就成功跟踪了本次虚拟地址到物理地址的转换！

我们总结一下关键调试步骤：

1. 触发了页表访问：虚拟地址 `0xffffffffc0205ff8` 的访问
2. 进入页表遍历：`get_physical_address` 函数
3. 读取页表项：从物理地址 `0x80206ff8` 读取 PTE `0x200000cf`
4. 分析PTE标志：
   - V=1 (有效)
   - R=0, W=1, X=1 (可写、可执行但不可读？)
   - U=0 (超级用户)
   - A=1, D=1 (已访问、已修改)
5. 通过权限检查：尽管看起来R=0，但可能由于 `mxr` 或其他机制通过
6. 计算物理地址：
   - VPN = `0xffffffffc0205`
   - PPN = `0x80000`

   - 物理地址 = `0x80205000`
7. 建立TLB映射：权限为 `0x7`（读+写+执行）

更重要的是，发现了一个关键信息：这是一个1GB大页映射！

- 在第一级页表（i=0, ptshift=18）就找到了叶子PTE
- 虚拟地址范围：`0xffffffffc0000000` 到 `0xffffffffffffffff` → 物理地址范围：`0x80000000` 到 `0xbfffffff`
- 这是操作系统内核空间的典型映射方式

调试成果：

✅ 理解了RISC-V三级页表遍历过程

✅ 观察了PTE标志位的检查和解析

✅ 验证了虚拟地址到物理地址的计算公式

✅ 发现了操作系统使用1GB大页优化内核映射

✅ 跟踪了TLB缓存的建立过程

指导书问题

3. 是否能够在qemu-4.1.1的源码中找到模拟CPU查找TLB的C代码？

在 `qemu-4.1.1/accel/tcg/cputlb.c` 中

✅ 主要 TLB 查找相关函数

1. `tlb_set_page_with_attrs()`
   - 功能：向 TLB 中添加一个新的条目（虚拟地址 → 物理地址映射）。
   - 在 TCG 生成代码时调用，用于填充 TLB。
   - 包含地址映射、权限检查、缓存设置等逻辑。
2. `tlb_fill()`
   - 功能：当 TLB 未命中时，触发异常处理并填充 TLB。
   - 调用 CPU 特定的 `tlb_fill` 方法（例如 `cpu_arm_tlb_fill`）。
3. `get_page_addr_code()`
   - 功能：获取代码页的物理地址，用于指令翻译。
   - 如果 TLB 命中，返回物理地址；否则触发 TLB 填充。
4. `probe_write()`

   - 功能：检查写访问权限，如果未命中则触发 TLB 填充。
5. `tlb_vaddr_to_host()`
   - 功能：将虚拟地址转换为宿主（host）地址指针。
   - 如果 TLB 命中且是 RAM 访问，返回 host 指针；否则返回 `NULL`。
6. `atomic_mmu_lookup()`
   - 功能：原子操作（RMW）的 TLB 查找，支持对齐检查和权限验证。
7. `load_helper()` / `store_helper()`
   - 功能：通用的加载和存储辅助函数，封装了 TLB 查找、对齐检查、IO 访问、未对齐访问等逻辑。

✅ TLB 数据结构

在文件中涉及以下关键数据结构：

- `CPUTLBEntry`：表示一个 TLB 条目，包含地址、权限等信息。
- `CPUTLBDesc`：描述 TLB 的状态（如大小、使用条目数等）。
- `CPUIOTLBEntry`：IO 地址映射条目。

---

4. 仍然是TLB，qemu中模拟出来的TLB和我们真实cpu中的TLB有什么逻辑上的区别?

对比维度	QEMU 模拟 TLB	真实 CPU TLB	核心差异说明
根本性质	软件实现的缓存机制	硬件实现的缓存单元	QEMU TLB是软件算法，真实TLB是物理电路
设计目的	加速虚拟机的地址翻译	加速真实程序的地址翻译	QEMU是为仿真加速，CPU是为应用加速
翻译层级	两级翻译： Guest虚拟地址 → Guest物理地址 → Host虚拟地址	一级翻译： CPU虚拟地址 → 物理地址	QEMU需要多一层Guest到Host的映射
物理位置	在QEMU进程的虚拟地址空间中	在CPU芯片内部（专用SRAM）	一个在内存，一个在芯片
访问速度	纳秒级（内存访问）	皮秒级（片上缓存访问）	硬件TLB快2-3个数量级
大小管理	动态可调整（tlb_mmu_resize_locked）	固定大小（硬件设计时确定）	QEMU可根据负载调整，硬件TLB固定
失效机制	软件显式调用tlb_flush*()函数	硬件自动管理+软件指令（如INVLPG）	QEMU完全软件控制，真实CPU软硬结合
命中处理	返回Host内存指针，继续软件执行	直接提供物理地址给ALU	QEMU仍需软件处理，CPU硬件直接使用
未命中处理	调用tlb_fill()触发软件异常处理	触发Page Walk硬件单元遍历页表	QEMU走异常处理流程，CPU走硬件流程
替换策略	软件实现（可自定义算法）	硬件固定（通常伪LRU或随机）	QEMU灵活可调，CPU策略固化
与MMU关系	模拟Guest的MMU行为	是CPU MMU的一部分	QEMU TLB模拟MMU功能，真实TLB是MMU组件
异常触发	可能导致QEMU内部的异常处理	可能导致CPU异常（如Page Fault）	QEMU异常在软件层，CPU异常在硬件层