gdb 调试页表查询过程

实验流程及解读

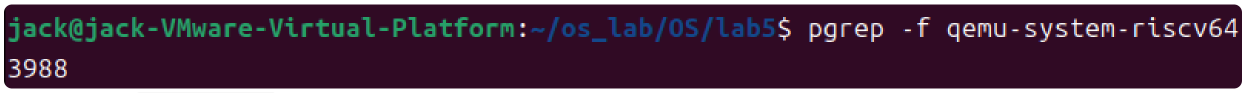
1. 启动调试环境

- **终端1**：在项目目录下启动第一个终端，并输入以下命令以开始调试模式：

  ```
  make debug
  ```

  这条命令会编译uCore并启动QEMU进入调试模式，等待GDB连接。
- **终端2**：在同一目录下打开第二个终端，首先找到正在运行的QEMU进程ID（PID）：

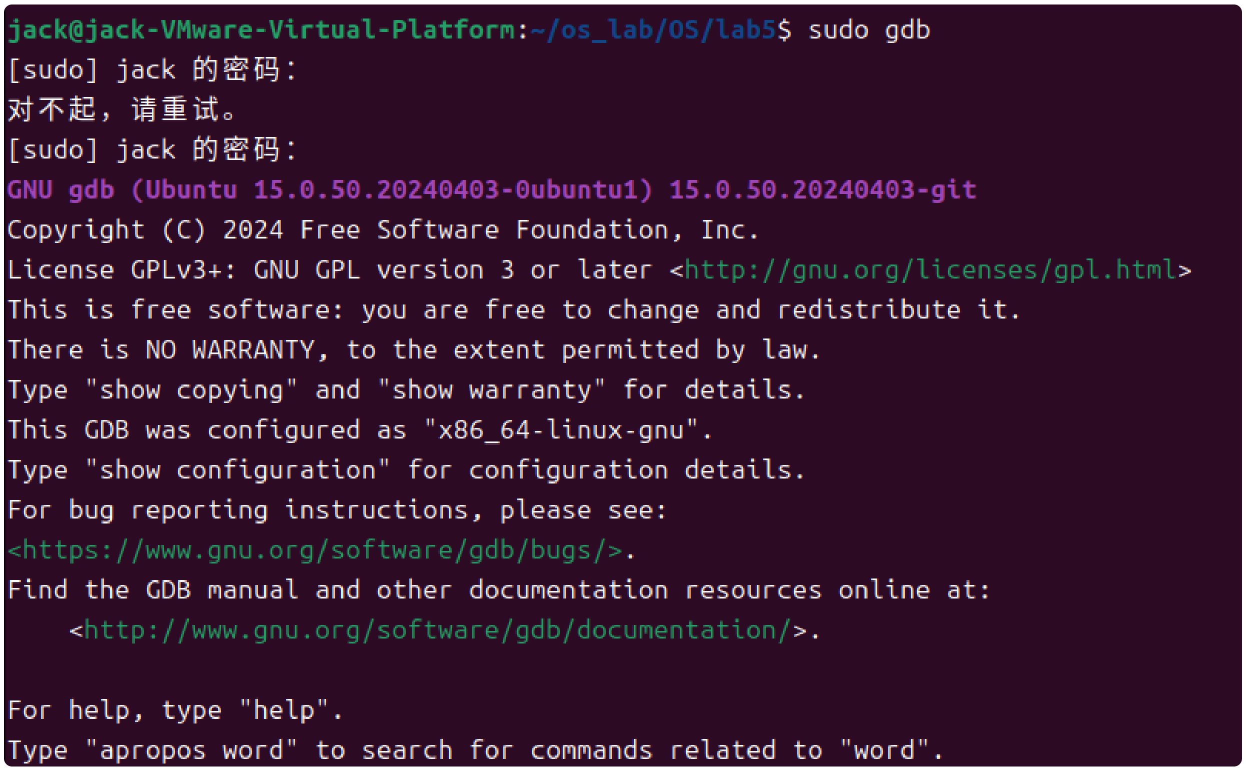
  ```
  pgrep -f qemu-system-riscv64
  ```

  ```
  jack@jack-VMware-Virtual-Platform:~/os_lab/OS/lab5$ pgrep -f qemu-system-riscv64
  3988
  ```

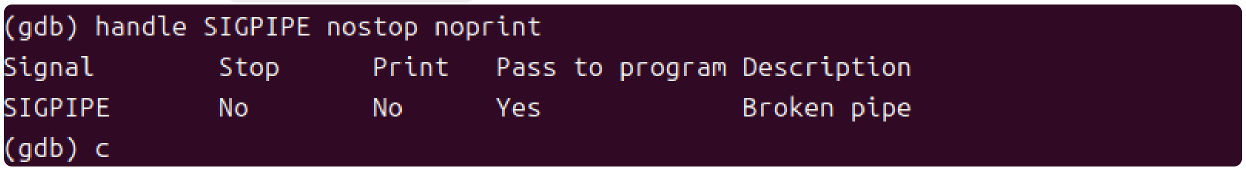
  接着使用`sudo gdb`启动GDB，并附加到该QEMU进程：

  ```
  (gdb) attach <刚才查到的PID>
  (gdb) handle SIGPIPE nostop noprint
  ```



```
jack@jack-VMware-Virtual-Platform:~/os_lab/OS/lab5$ sudo gdb
[sudo] jack 的密码：
对不起，请重试。
[sudo] jack 的密码：
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".
```

  - `attach <PID>`：将GDB附加到指定的QEMU进程上。
  - `handle SIGPIPE nostop noprint`：配置GDB忽略SIGPIPE信号，避免不必要的中断。
- **继续执行**：输入`(gdb) continue`让程序继续运行，直到遇到设定的断点。

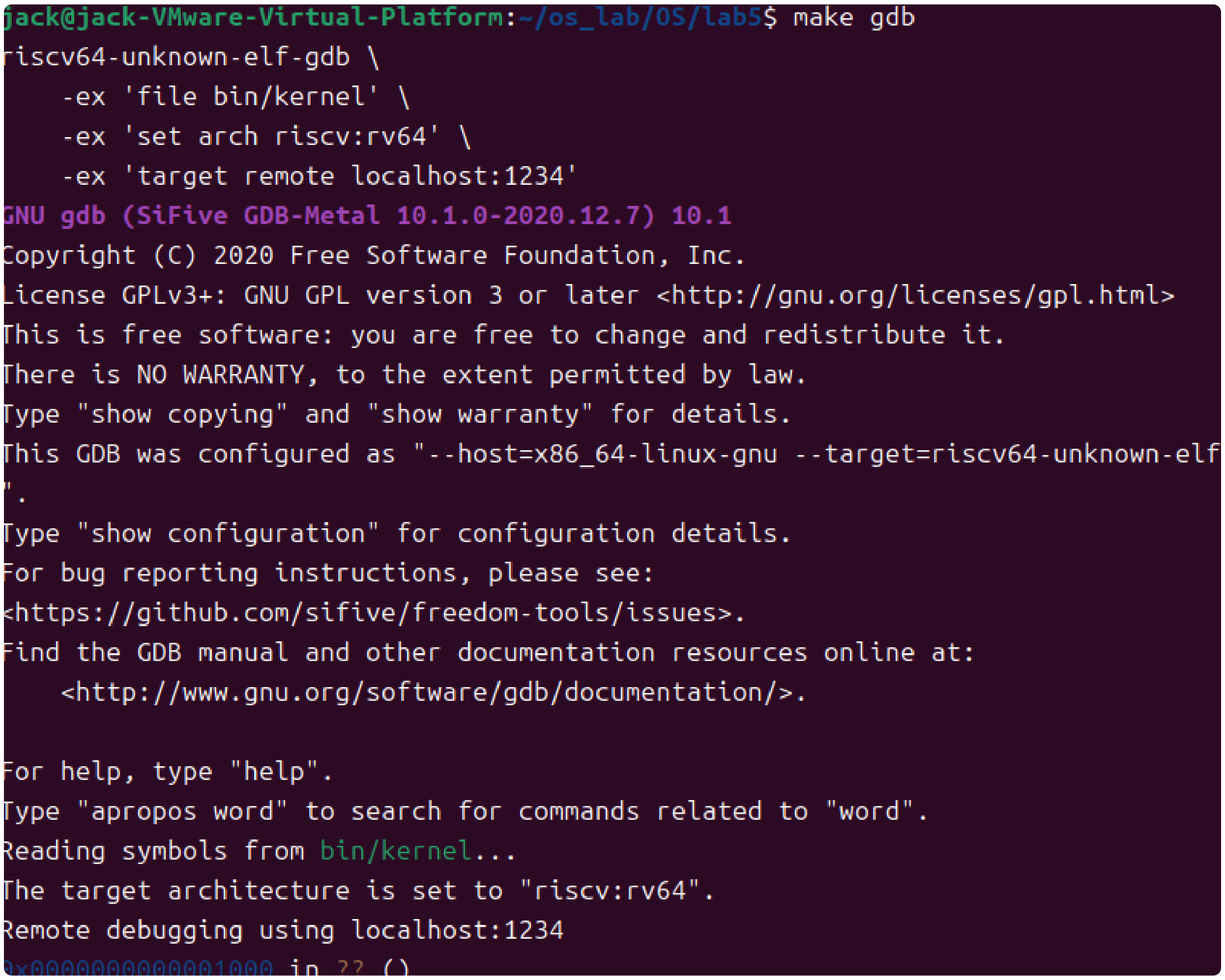


```
(gdb) handle SIGPIPE nostop noprint
Signal        Stop      Print   Pass to program Description
SIGPIPE       No        No      Yes             Broken pipe
(gdb) c
```

1. **设置初始断点并启动uCore调试**

- 在**终端3**中，同样在项目目录下启动第三个终端，并输入以下命令来启动针对uCore代码的调试会话：

```
make gdb
```

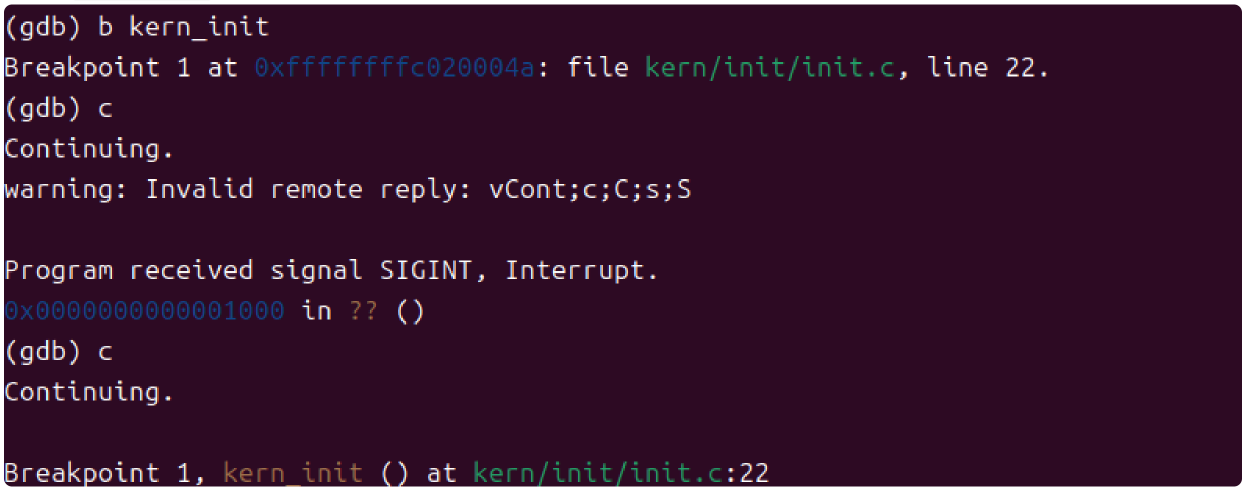


```
jack@jack-VMware-Virtual-Platform:~/os_lab/OS/lab5$ make gdb
riscv64-unknown-elf-gdb \
    -ex 'file bin/kernel' \
    -ex 'set arch riscv:rv64' \
    -ex 'target remote localhost:1234'
GNU gdb (SiFive GDB-Metal 10.1.0-2020.12.7) 10.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: 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 "--host=x86_64-linux-gnu --target=riscv64-unknown-elf
".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://github.com/sifive/freedom-tools/issues>.
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".
Reading symbols from bin/kernel...
The target architecture is set to "riscv:rv64".
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
```

- 设置一个断点于内核初始化函数处：

```
(gdb) break kern_init
```

使用 `continue` 命令执行到此断点。

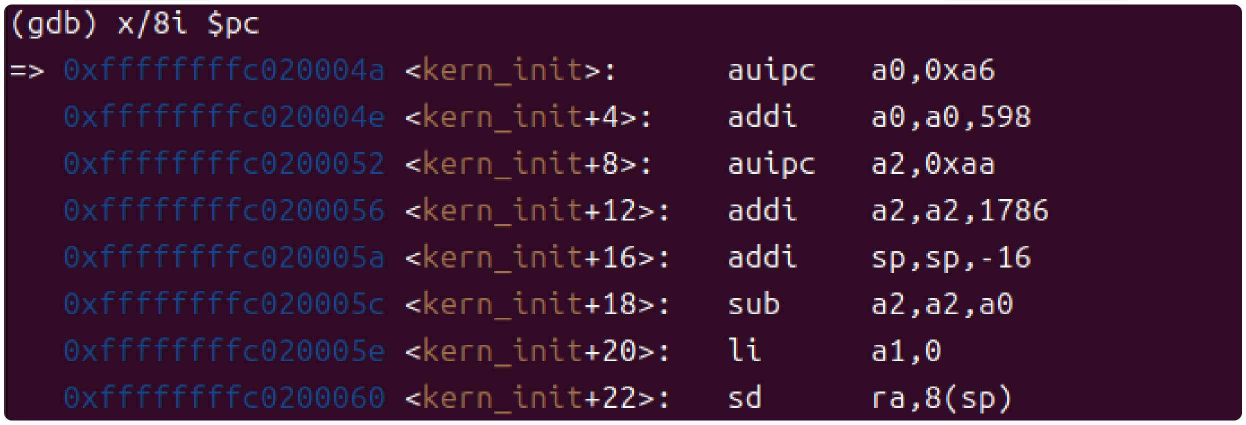


```
(gdb) b kern_init
Breakpoint 1 at 0xffffffffc020004a: file kern/init/init.c, line 22.
(gdb) c
Continuing.
warning: Invalid remote reply: vCont;c;C;s;S

Program received signal SIGINT, Interrupt.
0x0000000000001000 in ?? ()
(gdb) c
Continuing.

Breakpoint 1, kern_init () at kern/init/init.c:22
```

  - `break kern_init`：设置断点在内核初始化函数 `kern_init` 处。
  - `continue`：继续执行程序直到遇到断点。

2. 分析即将执行的指令

- 输入`x/bi $pc`查看接下来将要执行的8条指令，发现目标访存指令`sd ra,8(sp)`。

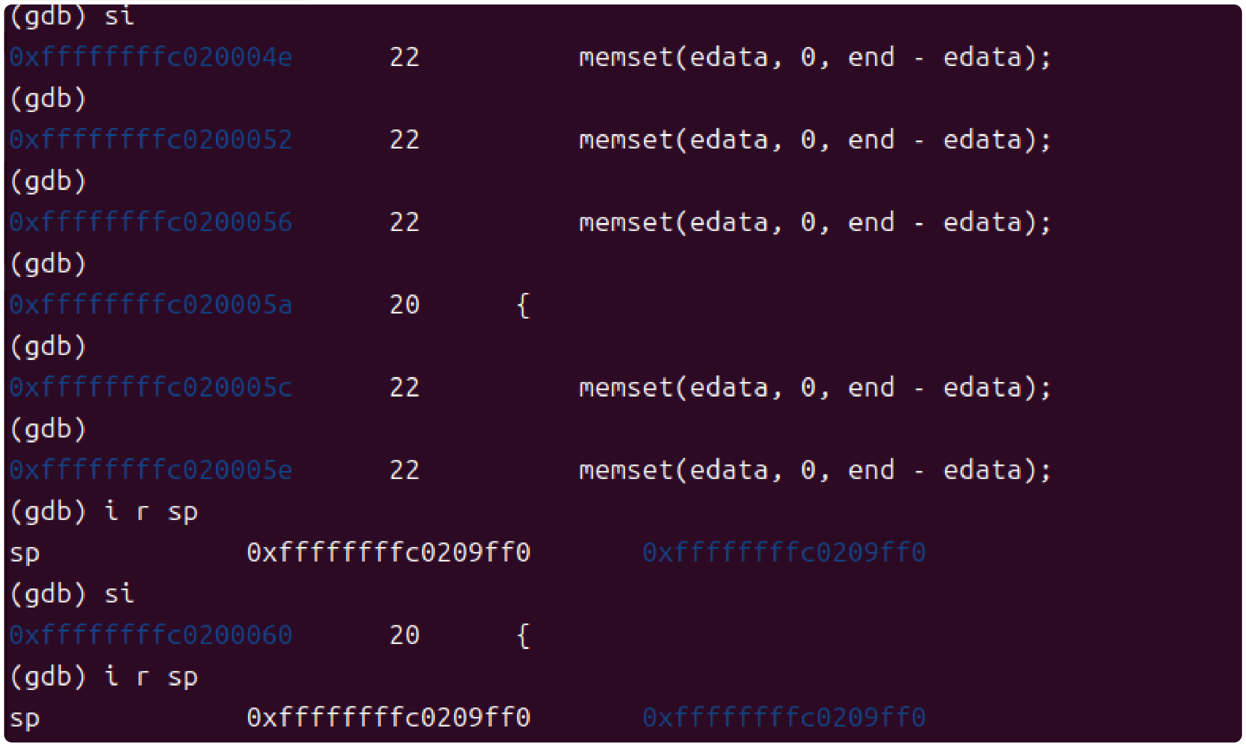


```
(gdb) x/8i $pc
=> 0xffffffffc020004a <kern_init>:      auipc   a0,0xa6
   0xffffffffc020004e <kern_init+4>:    addi    a0,a0,598
   0xffffffffc0200052 <kern_init+8>:    auipc   a2,0xaa
   0xffffffffc0200056 <kern_init+12>:   addi    a2,a2,1786
   0xffffffffc020005a <kern_init+16>:   addi    sp,sp,-16
   0xffffffffc020005c <kern_init+18>:   sub     a2,a2,a0
   0xffffffffc020005e <kern_init+20>:   li      a1,0
   0xffffffffc0200060 <kern_init+22>:   sd      ra,8(sp)
```

  - `x/bi $pc`：显示当前PC指针位置之后的8条汇编指令。

3. 获取栈指针值

- 使用`info registers sp`获取当前栈指针的值，并计算出实际的访存地址为`0xffffffffc0209ff8`。



```
(gdb) si
0xffffffffc020004e      22          memset(edata, 0, end - edata);
(gdb)
0xffffffffc0200052      22          memset(edata, 0, end - edata);
(gdb)
0xffffffffc0200056      22          memset(edata, 0, end - edata);
(gdb)
0xffffffffc020005a      20      {
(gdb)
0xffffffffc020005c      22          memset(edata, 0, end - edata);
(gdb)
0xffffffffc020005e      22          memset(edata, 0, end - edata);
(gdb) i r sp
sp             0xffffffffc0209ff0       0xffffffffc0209ff0
(gdb) si
0xffffffffc0200060      20      {
(gdb) i r sp
sp             0xffffffffc0209ff0       0xffffffffc0209ff0
```

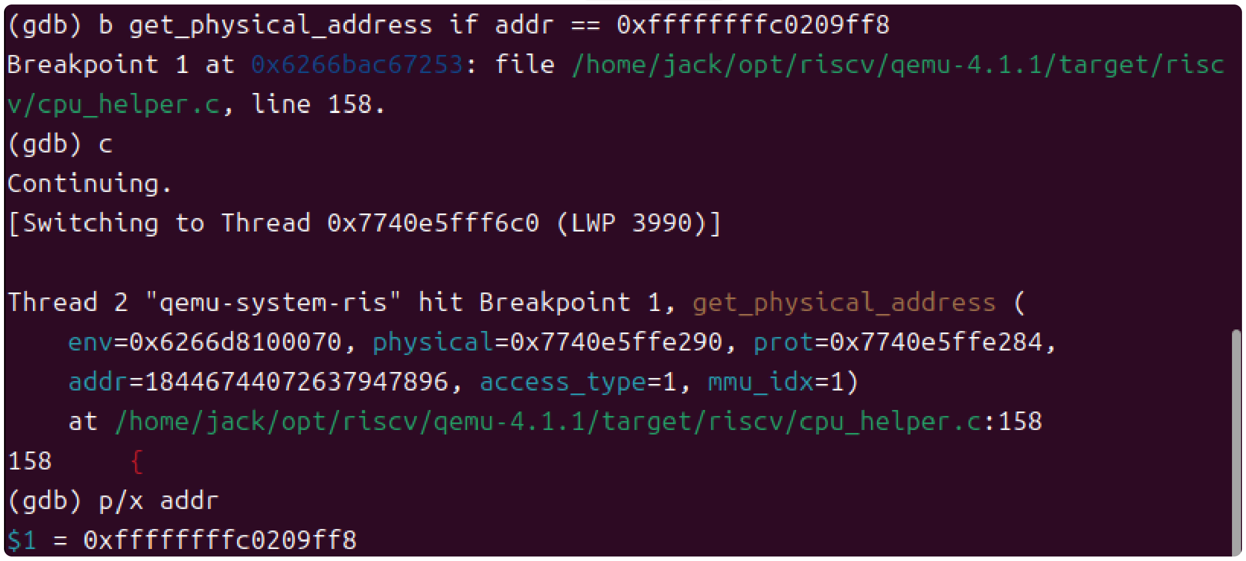
  - `info registers sp`：查询栈指针寄存器`sp`的值。

4. 设置条件断点并验证

- 回到**终端2**中的GDB，设置条件断点：

```
(gdb) break get_physical_address if addr == 0xffffffffc0209ff8
```

然后继续执行程序，直到触发断点。使用`p/x addr`验证是否为预期的地址。



```
(gdb) b get_physical_address if addr == 0xffffffffc0209ff8
Breakpoint 1 at 0x6266bac67253: file /home/jack/opt/riscv/qemu-4.1.1/target/risc
v/cpu_helper.c, line 158.
(gdb) c
Continuing.
[Switching to Thread 0x7740e5fff6c0 (LWP 3990)]

Thread 2 "qemu-system-ris" hit Breakpoint 1, get_physical_address (
    env=0x6266d8100070, physical=0x7740e5ffe290, prot=0x7740e5ffe284,
    addr=18446744072637947896, access_type=1, mmu_idx=1)
    at /home/jack/opt/riscv/qemu-4.1.1/target/riscv/cpu_helper.c:158
158     {
(gdb) p/x addr
$1 = 0xffffffffc0209ff8
```

- `break get_physical_address if addr == 0xffffffffc0209ff8`：当`addr`等于指定值时，在`get_physical_address`函数处设置断点。
- `p/x addr`：打印变量`addr`的十六进制值。

5. 堆栈回溯与源码分析

- 使用 `backtrace` 命令显示调用栈，了解函数调用链路。

```
(gdb) bt
#0  get_physical_address (env=0x6266d8100070, physical=0x7740e5ffe290,
    prot=0x7740e5ffe284, addr=18446744072637947896, access_type=1, mmu_idx=1)
    at /home/jack/opt/riscv/qemu-4.1.1/target/riscv/cpu_helper.c:158
#1  0x00006266bac67f2f in riscv_cpu_tlb_fill (cs=0x6266d80f7660,
    address=18446744072637947896, size=8, access_type=MMU_DATA_STORE,
    mmu_idx=1, probe=false, retaddr=131120620372293)
    at /home/jack/opt/riscv/qemu-4.1.1/target/riscv/cpu_helper.c:451
#2  0x00006266babae1bf in tlb_fill (cpu=0x6266d80f7660,
    addr=18446744072637947896, size=8, access_type=MMU_DATA_STORE, mmu_idx=1,
    retaddr=131120620372293)
    at /home/jack/opt/riscv/qemu-4.1.1/accel/tcg/cputlb.c:878
#3  0x00006266babb4698 in store_helper (big_endian=false, size=8,
    retaddr=131120620372293, oi=49, val=2147486210, addr=18446744072637947896,
    env=0x6266d8100070)
    at /home/jack/opt/riscv/qemu-4.1.1/accel/tcg/cputlb.c:1522
#4  helper_le_stq_mmu (env=0x6266d8100070, addr=18446744072637947896,
    val=2147486210, oi=49, retaddr=131120620372293)
    at /home/jack/opt/riscv/qemu-4.1.1/accel/tcg/cputlb.c:1672
#5  0x00007740e6000145 in code_gen_buffer ()
#6  0x00006266babd2d79 in cpu_tb_exec (cpu=0x6266d80f7660,
    itb=0x7740e6000040 <code_gen_buffer+19>)
    at /home/jack/opt/riscv/qemu-4.1.1/accel/tcg/cpu-exec.c:173
#7  0x00006266babd3bbf in cpu_loop_exec_tb (cpu=0x6266d80f7660,
--Type <RET> for more, q to quit, c to continue without paging--
    tb=0x7740e6000040 <code_gen_buffer+19>, last_tb=0x7740e5ffe988,
    tb_exit=0x7740e5ffe980)
    at /home/jack/opt/riscv/qemu-4.1.1/accel/tcg/cpu-exec.c:621
#8  0x00006266babd3ee5 in cpu_exec (cpu=0x6266d80f7660)
    at /home/jack/opt/riscv/qemu-4.1.1/accel/tcg/cpu-exec.c:732
#9  0x00006266bab8611f in tcg_cpu_exec (cpu=0x6266d80f7660)
    at /home/jack/opt/riscv/qemu-4.1.1/cpus.c:1435
#10 0x00006266bab869d8 in qemu_tcg_cpu_thread_fn (arg=0x6266d80f7660)
    at /home/jack/opt/riscv/qemu-4.1.1/cpus.c:1743
#11 0x00006266bb00becf in qemu_thread_start (args=0x6266d810dd50)
    at util/qemu-thread-posix.c:502
#12 0x00007740e8a9caa4 in start_thread (arg=<optimized out>)
    at ./nptl/pthread_create.c:447
#13 0x00007740e8b29c6c in clone3 ()
    at ../sysdeps/unix/sysv/linux/x86_64/clone3.S:78
```

  - `backtrace`：显示当前堆栈跟踪信息，帮助理解函数调用关系。
- 最后，利用 `list <函数名>` 或 `list <函数名>,+200` 命令查看特定函数的具体实现细节，深入理解虚拟地址到物理地址的转换过程。
  - `list <函数名>`：列出指定函数的源代码。

- `list <函数名>,+200`：如果需要查看更多的源代码行，可以使用此命令增加显示的行数。

6. 单步调试

在终端2中：
使用如下指令来查看下一条运行的指令并运行

```
(gdb)x/1i $pc #可以多看几条更方便理解上下文
(gdb)si   #执行这条指令
```

下面是部分过程（不详细列出）

代码分析

store helper

```
static inline void __attribute__((always_inline))

store_helper(CPUArchState *env, target_ulong addr, uint64_t val,

             TCGMemOpIdx oi, uintptr_t retaddr, size_t size, bool big_endian)

{

    uintptr_t mmu_idx = get_mmuidx(oi);

    uintptr_t index = tlb_index(env, mmu_idx, addr);

    CPUTLBEntry *entry = tlb_entry(env, mmu_idx, addr);

    target_ulong tlb_addr = tlb_addr_write(entry);

    const size_t tlb_off = offsetof(CPUTLBEntry, addr_write);

    unsigned a_bits = get_alignment_bits(get_memop(oi));

    void *haddr;
```

```c
    /* Handle CPU specific unaligned behaviour */
    if (addr & ((1 << a_bits) - 1)) {
        cpu_unaligned_access(env_cpu(env), addr, MMU_DATA_STORE,
                             mmu_idx, retaddr);
    }

    /* If the TLB entry is for a different page, reload and try again.  */
    if (!tlb_hit(tlb_addr, addr)) {
        if (!victim_tlb_hit(env, mmu_idx, index, tlb_off,
            addr & TARGET_PAGE_MASK)) {
            tlb_fill(env_cpu(env), addr, size, MMU_DATA_STORE,
                     mmu_idx, retaddr);
            index = tlb_index(env, mmu_idx, addr);
            entry = tlb_entry(env, mmu_idx, addr);
        }
        tlb_addr = tlb_addr_write(entry) & ~TLB_INVALID_MASK;
    }


    /* Handle an IO access.  */
    if (unlikely(tlb_addr & ~TARGET_PAGE_MASK)) {
        if ((addr & (size - 1)) != 0) {
            goto do_unaligned_access;
        }
        if (tlb_addr & TLB_RECHECK) {
            /*
             * This is a TLB_RECHECK access, where the MMU protection
             * covers a smaller range than a target page, and we must
             * repeat the MMU check here. This tlb_fill() call might
             * longjump out if this access should cause a guest exception.
             */
            tlb_fill(env_cpu(env), addr, size, MMU_DATA_STORE,
                     mmu_idx, retaddr);
            index = tlb_index(env, mmu_idx, addr);
            entry = tlb_entry(env, mmu_idx, addr);

            tlb_addr = tlb_addr_write(entry);
            tlb_addr &= ~TLB_RECHECK;
            if (!(tlb_addr & ~TARGET_PAGE_MASK)) {
                /* RAM access */
                goto do_aligned_access;
            }
        }

        io_writex(env, &env_tlb(env)->d[mmu_idx].iotlb[index], mmu_idx,
                  handle_bswap(val, size, big_endian),
                  addr, retaddr, size);
```

```
    return;

}


/* Handle slow unaligned access (it spans two pages or IO).  */

if (size > 1

    && unlikely((addr & ~TARGET_PAGE_MASK) + size - 1

                 >= TARGET_PAGE_SIZE)) {

    int i;

    uintptr_t index2;

    CPUTLBEntry *entry2;

    target_ulong page2, tlb_addr2;

do_unaligned_access:

    /*

     * Ensure the second page is in the TLB.  Note that the first page

     * is already guaranteed to be filled, and that the second page

     * cannot evict the first.

     */

    page2 = (addr + size) & TARGET_PAGE_MASK;

    index2 = tlb_index(env, mmu_idx, page2);

    entry2 = tlb_entry(env, mmu_idx, page2);

    tlb_addr2 = tlb_addr_write(entry2);

    if (!tlb_hit_page(tlb_addr2, page2)

        && !victim_tlb_hit(env, mmu_idx, index2, tlb_off,
                           page2 & TARGET_PAGE_MASK)) {
        tlb_fill(env_cpu(env), page2, size, MMU_DATA_STORE,
                 mmu_idx, retaddr);
    }

    /*
     * XXX: not efficient, but simple.
     * This loop must go in the forward direction to avoid issues
     * with self-modifying code in Windows 64-bit.
     */
    for (i = 0; i < size; ++i) {
        uint8_t val8;
        if (big_endian) {
            /* Big-endian extract.  */
            val8 = val >> (((size - 1) * 8) - (i * 8));
        } else {
            /* Little-endian extract.  */
            val8 = val >> (i * 8);
        }
        helper_ret_stb_mmu(env, addr + i, val8, oi, retaddr);
    }
    return;
}
```

```
do_aligned_access:
   haddr = (void *)((uintptr_t)addr + entry->addend);
   switch (size) {
   case 1:
       stb_p(haddr, val);
       break;
   case 2:
       if (big_endian) {
           stw_be_p(haddr, val);
       } else {
           stw_le_p(haddr, val);
       }
       break;
   case 4:
       if (big_endian) {
           stl_be_p(haddr, val);
       } else {
           stl_le_p(haddr, val);
       }
       break;
   case 8:
       if (big_endian) {
            stq_be_p(haddr, val);

        } else {

            stq_le_p(haddr, val);

        }

        break;

    default:

        g_assert_not_reached();

        break;

    }

}
```

`store_helper` 是由 **TCG（Tiny Code Generator）** 生成的代码所调用的关键函数，用于处理客户机（Guest）的内存写操作。其主要职责包括：

1. **TLB 查找与填充**

- 首先尝试从当前 TLB **（Translation Lookaside Buffer）** 中查找虚拟地址对应的物理页。
- 若 **TLB 未命中**（`!tlb_hit`），则调用 `tlb_fill()` 触发完整的地址翻译流程（最终会调用架构相关的函数如 `get_physical_address()`）。

2. **对齐检查**

- 检查内存访问是否满足对齐要求。
- 若不满足，则调用 `cpu_unaligned_access()`，可能抛出异常或模拟非对齐访问行为。

3. **I/O 访问处理**

- 如果 TLB 条目标记为 **I/O 区域**（即 `tlb_addr & ~TARGET_PAGE_MASK != 0`），则调用 `io_writex()` 执行设备寄存器写入。

4. **跨页未对齐访问处理**

- 若一次写操作跨越两个页面（例如在页边界写入一个 4 字节值），则 **逐字节处理**，确保每个字节都能正确触发 TLB 和 MMU 检查。

5. **普通 RAM 写入**

- 对于 **对齐的、单页内的 RAM 写入**：
  - 通过 `entry->addend` 将客户机虚拟地址转换为宿主机指针 `haddr`。
  - 使用 `st*_p` 系列函数（如 `stl_p`, `stw_p` 等）直接写入宿主机内存。

`tlb_fill`

```
/*
 * This is not a probe, so only valid return is success; failure
 * should result in exception + longjmp to the cpu loop.
 */
static void tlb_fill(CPUState *cpu, target_ulong addr, int size,
                     MMUAccessType access_type, int mmu_idx, uintptr_t retadd
{
    CPUClass *cc = CPU_GET_CLASS(cpu);
    bool ok;

    /*
     * This is not a probe, so only valid return is success; failure
     * should result in exception + longjmp to the cpu loop.
     */
    ok = cc->tlb_fill(cpu, addr, size, access_type, mmu_idx, false, retaddr);
    assert(ok);
}
```

`tlb_fill` 是 QEMU TCG（Tiny Code Generator）层中 **处理 TLB（Translation Lookaside Buffer）缺失** 的通用函数。它的作用是在当前 TLB 中找不到对应虚拟地址的映射时，触发一次完整的 **地址翻译流程**，并将结果填入 TLB。

关键点如下：

1. **调用架构特定的 `tlb_fill` 实现**
   - 通过 `CPU_GET_CLASS(cpu)->tlb_fill` 调用具体架构（如 RISC-V、x86、ARM 等）注册的 `tlb_fill` 方法。

- 例如，在 RISC-V 中，这会最终调用 `riscv_cpu_tlb_fill()`，而该函数内部会调用 `get_physical_address()` 进行页表遍历和虚拟地址到物理地址的转换。

2. **非探测性访问（non-probe）**
   - 注释明确指出：这不是一次“探测”(probe)，因此 **必须成功**。
   - 如果地址翻译失败（如缺页、权限错误），目标架构的 `tlb_fill` 实现 **不会返回 `false`，而是直接抛出异常并 `longjmp` 回 CPU 主循环**（例如触发 Page Fault）。
3. **参数含义：**
   - `addr`：要访问的虚拟地址。
   - `size`：访问的数据大小（字节）。
   - `access_type`：访问类型（`MMU_DATA_LOAD`、`MMU_DATA_STORE`、`MMU_INST_FETCH`）。
   - `mmu_idx`：当前的 MMU 上下文索引（如用户态/内核态、特权级等）。
   - `retaddr`：TCG 生成代码中的返回地址，用于异常时跳转回正确位置。
4. **断言 `assert(ok)`**
   - 因为失败会触发异常并 `longjmp`，正常返回时 `ok` 必为 `true`，否则说明逻辑错误。

`riscv_cpu_tlb_fill`

```
bool riscv_cpu_tlb_fill(CPUState *cs, vaddr address, int size,
                        MMUAccessType access_type, int mmu_idx,
                        bool probe, uintptr_t retaddr)
{
#ifndef CONFIG_USER_ONLY
    RISCVCPU *cpu = RISCV_CPU(cs);
    CPURISCVState *env = &cpu->env;
    hwaddr pa = 0;
    int prot;
    bool pmp_violation = false;
    int ret = TRANSLATE_FAIL;
    int mode = mmu_idx;

    qemu_log_mask(CPU_LOG_MMU, "%s ad %" VADDR_PRIx " rw %d mmu_idx %d\n",
                  __func__, address, access_type, mmu_idx);

    ret = get_physical_address(env, &pa, &prot, address, access_type, mmu_idx

    if (mode == PRV_M && access_type != MMU_INST_FETCH) {
        if (get_field(env->mstatus, MSTATUS_MPRV)) {
            mode = get_field(env->mstatus, MSTATUS_MPP);
        }
    }

    qemu_log_mask(CPU_LOG_MMU,
                  "%s address=%" VADDR_PRIx " ret %d physical " TARGET_FMT_pl
                  " prot %d\n", __func__, address, ret, pa, prot);

    if (riscv_feature(env, RISCV_FEATURE_PMP) &&
        (ret == TRANSLATE_SUCCESS) &&
        !pmp_hart_has_privs(env, pa, size, 1 << access_type, mode)) {
        ret = TRANSLATE_PMP_FAIL;
    }
    if (ret == TRANSLATE_PMP_FAIL) {
        pmp_violation = true;
    }
    if (ret == TRANSLATE_SUCCESS) {
        tlb_set_page(cs, address & TARGET_PAGE_MASK, pa & TARGET_PAGE_MASK,
                     prot, mmu_idx, TARGET_PAGE_SIZE);
        return true;
    } else if (probe) {
        return false;
    } else {
        raise_mmu_exception(env, address, access_type, pmp_violation);
        riscv_raise_exception(env, cs->exception_index, retaddr);
    }
#else
    switch (access_type) {
    case MMU_INST_FETCH:
        cs->exception_index = RISCV_EXCP_INST_PAGE_FAULT;
        break;
    case MMU_DATA_LOAD:
        cs->exception_index = RISCV_EXCP_LOAD_PAGE_FAULT;
        break;
    case MMU_DATA_STORE:
        cs->exception_index = RISCV_EXCP_STORE_PAGE_FAULT;
        break;
    }
    cpu_loop_exit_restore(cs, retaddr);
#endif
}
```

`riscv_cpu_tlb_fill` 是 **RISC-V 架构在 QEMU 中处理 TLB 缺失（TLB miss）的核心函数**，由通用 TCG 层的 `tlb_fill()` 调用（通过 `CPUClass::tlb_fill` 虚函数指针）。

它的主要功能包括：

1. **调用 `get_physical_address()` 进行地址翻译**

- 将 guest 虚拟地址 `address` 翻译为物理地址 `pa`，并获取访问权限 `prot`（如可读、可写、可执行）。
- 这是 **真正的页表遍历逻辑所在**，模拟 SATP 寄存器控制的多级页表机制。

2. **处理 MPRV（Modify PRiVilege）特殊情况**
- 在 Machine 模式（`PRV_M`）下，如果 `MSTATUS.MPRV=1`，则数据访问的权限检查应使用 `MSTATUS.MPP` 指定的特权级（通常是 S 或 U 模式），而非当前的 M 模式。
- 这是为了支持操作系统在 M 模式下代表用户程序访问内存时的权限检查。

3. **PMP（Physical Memory Protection）检查**
- 如果启用了 PMP 特性，则即使地址翻译成功，还需调用 `pmp_hart_has_privs()` 检查该物理地址是否允许当前访问。
- 若 PMP 拒绝访问，则标记为 `TRANSLATE_PMP_FAIL`，后续会触发异常。

4. **填充 TLB 或抛出异常**
- **成功时**：调用 `tlb_set_page()` 将虚拟页 → 物理页的映射填入 TLB，供后续快速访问。
- **失败时**：
    - 如果是 **探测模式（`probe == true`）**，仅返回 `false`，不触发异常（用于预检查）。
    - 否则，调用 `raise_mmu_exception()` 设置异常类型（如 Page Fault），并通过 `riscv_raise_exception()` 跳转回 CPU 主循环，模拟硬件异常行为。

5. **用户模式（CONFIG_USER_ONLY）简化处理**
- 在用户态模拟（如 Linux 用户程序仿真）中，没有完整的 MMU，任何 TLB 缺失都直接视为 Page Fault 并退出。

关键点总结

- **核心翻译函数**：`get_physical_address()` 是 VA → PA 的真正实现。
- **异常路径**：地址翻译失败或 PMP 拒绝 → 触发 RISC-V 异常（如 Store/Load Page Fault）。
- **TLB 填充**：成功翻译后通过 `tlb_set_page()` 建立映射，加速后续访存。
- **MPRV 与 PMP**：体现了 RISC-V 特有的权限模型，QEMU 必须精确模拟。

这个函数是你调试 **RISC-V 虚拟内存行为**（如缺页、权限错误、PMP 限制）时最重要的入口之一。

`get_physical_address`

```
/*
 * Translate a virtual address to a physical address.
 */
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) {
        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:
```

```c
#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 updat
                 *   and we return TRANSLATE_FAIL.
                 * - if the PTE changed by the time we went to update it, the
                 *   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;
}
```

重要函数功能总结： `get_physical_address`

`get_physical_address` 是 RISC-V 在 QEMU 中实现虚拟地址到物理地址翻译的核心函数。它模拟了 RISC-V 的多级页表机制（如 SV39、SV48），并严格遵循 RISC-V 特权架构规范。

主要功能流程：

1. **特权模式处理（MPRV）**
   如果当前是 Machine 模式（`PRV_M`）且启用了 `MSTATUS.MPRV`，则数据访问使用

`MSTATUS.MPP` 指定的特权级进行权限检查（常用于操作系统内核代表用户程序访问内存）。
2. **直通模式（Bare / Machine Mode）**
   若处于 Machine 模式或 MMU 被禁用（`MBARE`），虚拟地址直接等于物理地址，全权限返回。
3. **确定页表格式**
   根据 `satp.MODE`（v1.10+）或 `mstatus.VM`（旧版）判断使用哪种分页模式（SV32/SV39/SV48 等），设置：
   - `levels`：页表级数
   - `ptidxbits`：每级索引位数
   - `ptesize`：PTE 大小（4 字节或 8 字节）
4. **虚拟地址合法性检查**
   检查高位是否符合“符号扩展”规则（canonical address），否则返回失败。
5. **页表遍历（Page Table Walk）**
   从 `satp.PPN` 指向的根页表开始，逐级解析 PTE。每一级都检查：
   - PTE 是否有效（`PTE_V`）
   - 是否为中间节点（无 `R/W/X`）→ 继续下一级
   - 是否为非法组合（如仅 `PTE_W`）
   - 权限是否匹配当前模式（U/S/M）和 `SUM`/`MXR` 控制位
   - PPN 是否对齐
6. **PMP 检查**
   在读取每一级 PTE 前，检查该 PTE 所在物理地址是否允许 Supervisor 模式读取（防止 PMP 阻止页表访问）。
7. **访问/脏位（A/D Bits）更新**
   - 若需要设置 `PTE_A`（Accessed）或 `PTE_D`（Dirty），则原子地更新 PTE（使用 `atomic_cmpxchg`）。
   - 若更新失败（PTE 被并发修改），则重新遍历（`goto restart`）。
   - 若 PTE 位于 ROM 或 IO 区域（不可写），则拒绝访问。
8. **生成物理地址与权限**
   - 对于大页（superpage），将 VPN 的低位拼接到 PPN 上，形成完整物理地址。
   - 根据 PTE 标志和 MXR 设置 `*prot`（`PAGE_READ`/`WRITE`/`EXEC`），供 TLB 使用。

返回值含义：

- `TRANSLATE_SUCCESS`：成功，`*physical` 和 `*prot` 已设置。
- `TRANSLATE_FAIL`：页表项无效、权限不足、地址非法等。

- `TRANSLATE_PMP_FAIL`：PMP 拒绝访问页表或目标页。