lab2、lab5分支任务实验报告

lab2

准备阶段：

终端1：

```
tmux
make debug
```

终端2：

```
# 查看哪个进程在监听1234端口（GDB调试端口）
sudo lsof -i :1234 -P -n

# 启动GDB
sudo gdb

(gdb) attach <刚才查到的PID>

# 这个命令非常重要！让我解释一下：
# handle SIGPIPE nostop noprint
# SIGPIPE: 管道破裂信号（当向已关闭的管道写数据时产生）
# nostop: 当信号发生时不要停止（继续运行）
# noprint: 不要打印相关信息
# 如果不加这个，调试过程中可能会因为管道信号频繁中断
(gdb) handle SIGPIPE nostop noprint
(gdb) set remotetimeout unlimited

(gdb) continue # 启动执行，之后需要再设置断点
```

终端3：

```
make gdb
```

1.尝试理解我们调试流程中涉及到的qemu的源码，给出关键的调用路径，以及路径上一些关键的分支语句（不是让你只看分支语句），并通过调试演示某个访存指令访问的虚拟地址是如何在qemu的模拟中被翻译成一个物理地址的。

关键调用路径：

1. **CPU 执行访存指令 → TCG helper**

- 文件/函数：*cputlb.c* 中的各类 helper（例如 `helper_le_ldq_mmu` / `helper_le_ldul_mmu` / `helper_ret_ldub_mmu` / `helper_be_ldq_mmu` / 对应的 store helper 等；以及 `load_helper` / `store_helper`）。
- 说明：TCG 生成代码会调用这些 helper 来做访问前的 TLB 检查与最终内存读写。

2. **TLB 查找（快速路径）** → `tlb_hit()` **检查**

- 文件/函数：*cputlb.c*：`tlb_hit` 宏/函数、`tlb_index()`、`tlb_entry()`、`tlb_read_ofs()`、`tlb_hit_page()`、`tlb_hit_page_anyprot()`。
- 说明：helper 内部先取 `entry = tlb_entry(env, mmu_idx, addr)`，然后用 `tlb_hit` / `tlb_hit_page` 判断命中。

3. **如果 TLB 命中 → 直接使用 TLB 中的物理地址（host 地址）继续访问**

- 文件/函数：*cputlb.c* 中 `load_helper` / `store_helper`（命中时使用 `entry->addend` / `entry->addr_read|addr_write|addr_code` 得到 host 地址并直接访问主机内存或 IO）。
- 说明：命中路径会把虚拟地址与 `entry->addend` 组合得到 host 指针（或识别为 MMIO）。

4. **如果 TLB 未命中** → `tlb_fill()` → **最终会调用 RISC-V 的填充实现** `riscv_cpu_tlb_fill()` → `get_physical_address()`

- 文件/函数链：`accel/tcg/cputlb.c::tlb_fill()`（调用 `cc->tlb_fill`） → cpu.c 在初始化时把 `cc->tlb_fill = riscv_cpu_tlb_fill`（见 *cpu.c*） → `target/riscv/cpu_helper.c::riscv_cpu_tlb_fill()` → 内部调用 `get_physical_address()`（同文件 `get_physical_address`，实现 SV32/SV39/... 多级页表遍历与 PTE 检查）。
- 说明：`tlb_fill()` 是 cputlb 的入口（assert 成功），实际行为由 CPU 目标实现负责（RISC-V 的实现位于 `riscv_cpu_tlb_fill`）。

5. **页表遍历后 → 填充 TLB**

- 实际调用：`riscv_cpu_tlb_fill()` 在成功时调用 `tlb_set_page(...)`；`tlb_set_page` 在 cputlb.c 内部会调用 `tlb_set_page_with_attrs(...)` 来真正填充条目并处理 RAM vs MMIO vs ROM、iotlb 映射 等。
- 文件/函数：`target/riscv/cpu_helper.c::riscv_cpu_tlb_fill()` → `accel/tcg/cputlb.c::tlb_set_page()` → `accel/tcg/cputlb.c::tlb_set_page_with_attrs()`。

6. **填充后 → 继续访存操作（重试或直接返回 host 地址）**

- 文件/函数：回到 `accel/tcg/cputlb.c::load_helper` / `store_helper` 等，重新读取 `entry` 并完成访问；若需要 IO 则走 `io_readx` / `io_writex`（同文件）或 address_space 映射函数（exec.c 中的 `address_space_translate*`）用于把物理页映射到 `MemoryRegion` /RAM 指针。

关键分支语句：

- 要观察「TLB 命中/未命中」：在 cputlb.c 的 `tlb_hit(...)` 检查点和 `victim_tlb_hit(...)` 内下断点。
- 要观察「触发页表 walk / 页表判断与 A/D 更新与 CAS 重试」：在 `target/riscv/cpu_helper.c::get_physical_address` 中下断点，重点看 PTE 读取处（`ldl_phys` / `ldq_phys`）和随后的一系列 `if` 分支（上面列出那些）。
- 要观察「TLB 填充与 MemoryRegion 判定」：在 `accel/tcg/cputlb.c::tlb_set_page_with_attrs` 中下断点，检查 `memory_region_is_ram`、`address |= TLB_MMIO` 等分支，以及 `prot` 决定 `addr_read/addr_write/addr_code` 的分支。
- 要观察「SATP / VM 模式 / TL B flush 影响」：查看 csr.c 的 `write_satp`（SATP 改变时有分支触发 tlb flush 逻辑）。

找访存指令

```
(gdb) b kern_init
Breakpoint 1 at 0xffffffffc02000d8: file kern/init/init.c, line 30.
(gdb) c
Continuing.

Breakpoint 1, kern_init () at kern/init/init.c:30
30          memset(edata, 0, end - edata);
(gdb) x/10i $pc
=> 0xffffffffc02000d8 <kern_init>:      auipc   a0,0x5
   0xffffffffc02000dc <kern_init+4>:    addi    a0,a0,-192
   0xffffffffc02000e0 <kern_init+8>:    auipc   a2,0x5
   0xffffffffc02000e4 <kern_init+12>:   addi    a2,a2,-104
   0xffffffffc02000e8 <kern_init+16>:   addi    sp,sp,-16
   0xffffffffc02000ea <kern_init+18>:   sub     a2,a2,a0
   0xffffffffc02000ec <kern_init+20>:   li      a1,0
   0xffffffffc02000ee <kern_init+22>:   sd      ra,8(sp)
   0xffffffffc02000f0 <kern_init+24>:   jal     ra,0xffffffffc0201650 <memset>
   0xffffffffc02000f4 <kern_init+28>:   jal     ra,0xffffffffc0200220 <dtb_init>
```

选择 `0xffffffffc02000ee <kern_init+22>:    sd      ra,8(sp)` 进行观察。

```
(gdb) x/i $pc
=> 0xffffffffc02000ee <kern_init+22>:   sd      ra,8(sp)
(gdb) p/x $sp
$1 = 0xffffffffc0203ff0
(gdb) p/x $sp + 8
$2 = 0xffffffffc0203ff8
```

`0xffffffffc0203ff8` 是我们要使用的地址。

翻译过程调试

在终端3设置断点：

```
(gdb) b *0xffffffffc02000ee
Breakpoint 2 at 0xffffffffc02000ee: file kern/init/init.c, line 28.
(gdb) c
Continuing.
```

在终端2设置断点并调试：

```
set pagination off

(gdb) break get_physical_address if addr==0xffffffffc0203ff8

Breakpoint 1 at 0x5b0f9b224804: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/cpu_helper.c, line 158.

(gdb) break riscv_cpu_tlb_fill if address==0xffffffffc0203ff8

Breakpoint 2 at 0x5b0f9b225419: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/cpu_helper.c, line 438.

(gdb) break tlb_set_page

Breakpoint 3 at 0x5b0f9b16b704: file /home/foxcier/riscv/qemu-4.1.1/accel/tcg/cputlb.c, line
847.
(gdb) break raise_mmu_exception
Breakpoint 4 at 0x5b0f9b225090: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/cpu_helper.c, line 358.

(gdb) c
Continuing.
[Switching to Thread 0x73adede54640 (LWP 2215)]

Thread 3 "qemu-system-ris" hit Breakpoint 2, riscv_cpu_tlb_fill (cs=0x5b0fb47b5870,
address=18446744072637923320, size=8, access_type=MMU_DATA_STORE, mmu_idx=1, probe=false,
retaddr=127190725296453) at /home/foxcier/riscv/qemu-4.1.1/target/riscv/cpu_helper.c:438
438     {
(gdb) p/x addr
No symbol "addr" in current context.
(gdb) p/x address
$1 = 0xffffffffc0203ff8
(gdb) p/x pa
$2 = 0x36700000001
(gdb) p/x prot
$3 = 0x0
(gdb) c
Continuing.
```

```
Thread 3 "qemu-system-ris" hit Breakpoint 1, get_physical_address (env=0x5b0fb47be280,
physical=0x73adede53060, prot=0x73adede53054, addr=18446744072637923320, access_type=1,
mmu_idx=1) at /home/foxcier/riscv/qemu-4.1.1/target/riscv/cpu_helper.c:158
158     {
(gdb) p/x addr
$4 = 0xfffffffffc0203ff8
(gdb) p/x *physical
$5 = 0x0
(gdb) p/x *prot
$6 = 0x0
(gdb) p/x access_type
$7 = 0x1
(gdb) c
Continuing.

Thread 3 "qemu-system-ris" hit Breakpoint 3, tlb_set_page (cpu=0x5b0fb47b5870,
vaddr=18446744072637919232, paddr=2149593088, prot=7, mmu_idx=1, size=4096) at
/home/foxcier/ris
cv/qemu-4.1.1/accel/tcg/cputlb.c:847
847         tlb_set_page_with_attrs(cpu, vaddr, paddr, MEMTXATTRS_UNSPECIFIED,

(gdb) p/x vaddr
$8 = 0xfffffffffc0203000
(gdb) p/x paddr
$9 = 0x80203000
(gdb) p/x prot
$10 = 0x7
```

观察到的完整路径：

1. TLB 未命中 → 调用 `riscv_cpu_tlb_fill`。
2. `riscv_cpu_tlb_fill` 调用 `get_physical_address` 执行页表走查（page-walk）。
3. 页表走查成功后返回物理页与权限，`riscv_cpu_tlb_fill` 随即调用 `tlb_set_page` 填充软件 TLB。
4. `tlb_set_page` / `tlb_set_page_with_attrs` 会把物理页映射到 host 的 MemoryRegion（RAM vs MMIO）并建立最终宿主访问地址。

```
(gdb) bt
#0  get_physical_address
    (env=0x612490437280, physical=0x7edfdfffe060, prot=0x7edfdfffe054,
addr=18446744072637923320, access_type=1, mmu_idx=1) at /home/foxcier/riscv/qemu-
4.1.1/target/riscv/cpu_helper.c:158
#1  0x000061248488e4e0 in riscv_cpu_tlb_fill
    (cs=0x61249042e870, address=18446744072637923320, size=8, access_type=MMU_DATA_STORE,
mmu_idx=1, probe=false, retaddr=139500076400965) at /home/foxcier/riscv/qemu-
4.1.1/target/riscv/cpu_helper.c:451
#2  0x00006124847d480f in tlb_fill
    (cpu=0x61249042e870, addr=18446744072637923320, size=8, access_type=MMU_DATA_STORE,
mmu_idx=1, retaddr=139500076400965) at /home/foxcier/riscv/qemu-4.1.1/accel/tcg/cputlb.c:878
#3  0x00006124847dad1e in store_helper
```

```
    (big_endian=false, size=8, retaddr=139500076400965, oi=49, val=2147486210,
addr=18446744072637923320, env=0x612490437280) at /home/foxcier/riscv/qemu-
4.1.1/accel/tcg/cputlb.c:1522
#4  helper_le_stq_mmu
    (env=0x612490437280, addr=18446744072637923320, val=2147486210, oi=49,
retaddr=139500076400965)
    at /home/foxcier/riscv/qemu-4.1.1/accel/tcg/cputlb.c:1672
#5  0x00007edfe4800145 in code_gen_buffer ()
#6  0x00006124847f943b in cpu_tb_exec (cpu=0x61249042e870, itb=0x7edfe4800040
<code_gen_buffer+19>)
    at /home/foxcier/riscv/qemu-4.1.1/accel/tcg/cpu-exec.c:173
#7  0x00006124847fa281 in cpu_loop_exec_tb
    (cpu=0x61249042e870, tb=0x7edfe4800040 <code_gen_buffer+19>, last_tb=0x7edfdfffe758,
tb_exit=0x7edfdfffe750)
    at /home/foxcier/riscv/qemu-4.1.1/accel/tcg/cpu-exec.c:621
#8  0x00006124847fa5b6 in cpu_exec (cpu=0x61249042e870) at /home/foxcier/riscv/qemu-
4.1.1/accel/tcg/cpu-exec.c:732
#9  0x00006124847acc16 in tcg_cpu_exec (cpu=0x61249042e870) at /home/foxcier/riscv/qemu-
4.1.1/cpus.c:1435
```

2.单步调试页表翻译的部分，解释一下关键的操作流程。（这段是地址翻译的流程吗，我还是没有理解，给我解释的详细一点 / 这三个循环是在做什么，这两行的操作是从当前页表取出页表项吗，我还是没有理解）

设置断点：

```
set pagination off
set $TARGET = 0xffffffffc0203ff8

# 1) 当到达页表项地址计算处停下，打印 pte_addr
break target/riscv/cpu_helper.c:242
commands
  silent
  printf ">>> hit: PTE addr compute\n"
  p/x pte_addr
  printf "--> now use 'step' to execute the PTE read (ldl_phys/ldq_phys), then 'p/x pte' and
'p/x ppn'\n"
end

# 2) 在试图更新 A/D 位并写回 PTE 前停下（观察 updated_pte 和 address_space_translate 返回）
break target/riscv/cpu_helper.c:317
commands
  silent
  printf ">>> hit: about to update PTE (A/D)\n"
  p/x pte
  p/x updated_pte
  printf "--> step to execute address_space_translate/atomic_cmpxchg; then check if CAS
caused a restart\n"
end
```

```
# 3) 在 get_physical_address 入口处（仅当目标 addr 匹配）停下并打印上下文
break get_physical_address if addr==$TARGET
commands
  silent
  printf ">>> hit: get_physical_address (entry)\n"
  p/x addr
  p/x access_type
  p/x mmu_idx
  p/x env->mstatus
  p/x env->satp
  printf "--> step/next through the function, or use 'finish' to return and inspect caller's
pa/prot.\n"
end

# 4) caller: riscv_cpu_tlb_fill（在 translate 返回处查看 pa/prot）
break riscv_cpu_tlb_fill if address==$TARGET
commands
  silent
  printf ">>> hit: riscv_cpu_tlb_fill\n"
  p/x address
  p/x pa
  p/x prot
  printf "--> these are the PA/PROT used to fill the software TLB\n"
end

# 5) 最终：软件 TLB 写入点，确认 vaddr->paddr 映射
break tlb_set_page
commands
  silent
  printf ">>> hit: tlb_set_page\n"
  p/x vaddr
  p/x paddr
  p/x prot
  printf "--> TLB entry created; use 'continue' to resume emulation.\n"
end
```

```
(gdb) c
                Continuing.
[Switching to Thread 0x74f95d153640 (LWP 2294)]
>>> hit: riscv_cpu_tlb_fill
$1 = 0xffffffffc0203ff8
$2 = 0x36700000001
$3 = 0x0
--> these are the PA/PROT used to fill the software TLB
(gdb) c
Continuing.
>>> hit: get_physical_address (entry)
$4 = 0xffffffffc0203ff8
$5 = 0x1
$6 = 0x1
```

```
$7 = 0x8000000000006000
$8 = 0x8000000000080204
--> step/next through the function, or use 'finish' to return and inspect caller's pa/prot.

(gdb) tbreak target/riscv/cpu_helper.c:242
Note: breakpoint 1 also set at pc 0x5b3aa88f7c90.
Temporary breakpoint 6 at 0x5b3aa88f7c90: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/cpu_helper.c, line 242.
(gdb) tbreak target/riscv/cpu_helper.c:317
Note: breakpoint 2 also set at pc 0x5b3aa88f7f08.
Temporary breakpoint 7 at 0x5b3aa88f7f08: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/cpu_helper.c, line 317.
(gdb) tbreak riscv_cpu_tlb_fill
Note: breakpoint 4 also set at pc 0x5b3aa88f8419.
Temporary breakpoint 8 at 0x5b3aa88f8419: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/cpu_helper.c, line 438.

(gdb) c
Continuing.

Thread 3 "qemu-system-ris" hit Temporary breakpoint 6, get_physical_address
(env=0x5b3ad1dc4280, physical=0x74f95d152060, prot=0x74f95d152054,
addr=18446744072637923320, acces
s_type=1, mmu_idx=1) at /home/foxcier/riscv/qemu-4.1.1/target/riscv/cpu_helper.c:242
242             target_ulong pte_addr = base + idx * ptesize;
>>> hit: PTE addr compute
$9 = 0x8aa40ca5a883c3b0
--> now use 'step' to execute the PTE read (ldl_phys/ldq_phys), then 'p/x pte' and 'p/x ppn'
(gdb) p/x pte_addr
$10 = 0x8aa40ca5a883c3b0
(gdb) step
244             if (riscv_feature(env, RISCV_FEATURE_PMP) &&
(gdb) n
245                 !pmp_hart_has_privs(env, pte_addr, sizeof(target_ulong),
(gdb) n
244             if (riscv_feature(env, RISCV_FEATURE_PMP) &&
(gdb) n
252             target_ulong pte = ldq_phys(cs->as, pte_addr);
(gdb) p/x pte
$11 = 0x5b3aa8de2478
(gdb) p/x ppn
$12 = 0x5b3aa8de2630
(gdb) n
254             target_ulong ppn = pte >> PTE_PPN_SHIFT;
(gdb) n
256             if (!(pte & PTE_V)) {
(gdb) n
259             } else if (!(pte & (PTE_R | PTE_W | PTE_X))) {
(gdb) n
262             } else if ((pte & (PTE_R | PTE_W | PTE_X)) == PTE_W) {
(gdb) n
265             } else if ((pte & (PTE_R | PTE_W | PTE_X)) == (PTE_W | PTE_X)) {
(gdb) n
```

```
268             } else if ((pte & PTE_U) && ((mode != PRV_U) &&
(gdb) n
273             } else if (!(pte & PTE_U) && (mode != PRV_S)) {
(gdb) n
276             } else if (ppn & ((1ULL << ptshift) - 1)) {
(gdb) n
279             } else if (access_type == MMU_DATA_LOAD && !((pte & PTE_R) ||
(gdb) n
283             } else if (access_type == MMU_DATA_STORE && !(pte & PTE_W)) {
(gdb) n
286             } else if (access_type == MMU_INST_FETCH && !(pte & PTE_X)) {
(gdb) n
292                     (access_type == MMU_DATA_STORE ? PTE_D : 0);
(gdb) n
291                 target_ulong updated_pte = pte | PTE_A |
(gdb) n
295                 if (updated_pte != pte) {
(gdb) p/x pte
$13 = 0x200000cf
(gdb) p/x ppn
$14 = 0x80000
(gdb) c
Continuing.
>>> hit: tlb_set_page
$15 = 0xffffffffc0203000
$16 = 0x80203000
$17 = 0x7
--> TLB entry created; use 'continue' to resume emulation.
(gdb) n
849     }
(gdb) n
riscv_cpu_tlb_fill (cs=0x5b3ad1dbb870, address=18446744072637923320, size=8,
access_type=MMU_DATA_STORE, mmu_idx=1, probe=false, retaddr=128614238519621) at
/home/foxcier/risc
v/qemu-4.1.1/target/riscv/cpu_helper.c:474
474             return true;
(gdb) c
Continuing.
[Switching to Thread 0x74f95d9d6cc0 (LWP 2292)]
>>> hit: PTE addr compute
$18 = 0x5b3aa8ed4d91
--> now use 'step' to execute the PTE read (ldl_phys/ldq_phys), then 'p/x pte' and 'p/x ppn'
```

说明：

- `ldq_phys` / `ldl_phys` 成功读取了 PTE（成功打印了 `pte` 与 `ppn`）。
- 条件链（VALID、R/W/X、U/S、对齐和权限检查等）都通过，因此走到 leaf-PTE 分支并计算出 `*physical`，函数返回 `TRANSLATE_SUCCESS`。
- 因为没有进入需要写回/更新 PTE 的分支（或 CAS 也成功/未触发重试），函数很快返回；随后 `continue`，下一个断点 `tlb_set_page` 被命中，显示 `vaddr=0xffffffffc0203000` → `paddr=0x80203000`，`prot=0x7`。这就是完整的 VA→PA→TLB 填充链路。

`get_physical_address` 的详细逐步说明

- 判断当前 MMU 模式与特权（`mmu_idx` / `mstatus` / `satp`），若处于裸模式（bare）或 M 模式且不需转换则直接把 VA 当作 PA 返回。
- 从 `satp`（或旧版的 `sptbr`）得到页表基地址 `base`，根据 `satp` 中的 `MODE` 决定页表的层数（SV32→2 层，SV39→3 层，SV48→4 层）以及每层索引宽度和每个 PTE 的字节数。
- 做 VA 合法性检查（高位符号扩展 / 超出 VA 位宽则翻译失败）。
- 进入页表 walk 的循环（按级数从顶层到叶子）：
  - 计算当前层的索引 `idx = (addr >> (PGSHIFT + ptshift)) & ((1<<ptidxbits)-1)`。
  - 计算 PTE 的物理地址 `pte_addr = base + idx * ptesize`（这里 `base` 初始是页表根，遇 inner-PTE 会更新为下一级页表的物理基地址）。
  - （可选）做 PMP 检查，确保可以读取该 PTE。
  - 从物理内存读取 PTE（`ldl_phys` / `ldq_phys`），得到 `pte`。
  - 根据 PTE 的位判断：
    - 若 `PTE_V==0` → invalid → 翻译失败。
    - 若没有 R/W/X 位（即为 inner PTE） → 这是指向下一层页表，更新 `base = ppn << PGSHIFT` 并继续下一层。
    - 若是叶子 PTE（含 R/W/X）→ 检查保留组合、不合法的标志（PTE_W 单独或 PTE_W|PTE_X），以及 U/S 权限、ppn 对齐等。
    - 根据访问类型（读/写/指令取指）做权限检查（R/W/X 或 MXR）。
  - 若需要设置 Accessed/Dirty（A/D）位且 PTE 在 RAM 中，QEMU 会计算 `updated_pte = pte | PTE_A | (PTE_D if store)`，并尝试原子写回（`atomic_cmpxchg`）；若 CAS 失败则重试（goto restart），否则把 PTE 更新后继续。
  - 对 superpage（大页）做特殊处理：合成物理帧号并计算最终 `*physical`。
  - 最终设置 `prot`（PAGE_READ/WRITE/EXEC）并返回 TRANSLATE_SUCCESS。

“这三个循环”通常指：

1. TLB 查找循环（先在软件/硬件 TLB 查找）；
2. 页表 walk 的循环（`for (i = 0; i < levels; i++, ptshift -= ptidxbits)`：对每一层页表都做一次索引/读取/判定）；
3. （在写回 AD 时可能触发的）CAS-重试循环（如果 atomic_cmpxchg 失败会 goto restart 重走 page-walk）。
   你可以向老师解释：主循环是页表层级循环（SV39 就是三层），每层都计算索引、读 PTE、判断是“内节点”还是“叶子”。

代码中那两行：

- `target_ulong pte = ldq_phys(cs->as, pte_addr);`

  这行从物理地址 `pte_addr` 读取出页表项（PTE），这是“从当前页表取出页表项”的操作（用的是 QEMU 的物理读 helper）。

- `target_ulong ppn = pte >> PTE_PPN_SHIFT;`

这行把 PTE 中的物理页号（PPN）抽取出来，后续用 `ppn << PGSHIFT` 作为下一级页表的 `base`（如果这是 inner PTE）或作为最终物理页基址的一部分（如果是叶子/大页）。

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，通过调试说明其中的细节。（按照riscv的流程，是不是应该先查tlb，tlbmiss之后才从页表中查找，给我找一下查找tlb的代码）

TLB 查找逻辑主要在 cputlb.c（TLB 命中/未命中处理、load/store helper），未命中时调用目标 CPU 的 `tlb_fill` 回调（riscv 为 `riscv_cpu_tlb_fill`），该回调在目标代码中做 page-walk（`get_physical_address`）并最终调用 `tlb_set_page` 填表。

关键文件与函数：

- cputlb.c — 软件 TLB / helpers
  - TLB 填充入口（调用目标回调）：cputlb.c (`tlb_fill`)
  - 创建/写入 TLB 条目：cputlb.c (`tlb_set_page_with_attrs`) 和 cputlb.c (`tlb_set_page`)
  - TLB 命中/查找和回退逻辑（victim TLB）：cputlb.c (`victim_tlb_hit`)、以及页面检查宏 `tlb_hit` / `tlb_hit_page_anyprot`（附近实现）
  - TLB 查找点（TCG/softmmu 访问时调用）：
    - 通用 load helper（先查 TLB，再在未命中时调用 `tlb_fill`）：cputlb.c (`load_helper`)
    - atomic/atomic_mmu_lookup / probe helpers 也包含相同查找逻辑：见同一文件（约 1000–1300 行范围）
- 目标（RISC-V）页表回调
  - cpu_helper.c — 页表 walk 与转换
    - 页表走查实现：cpu_helper.c (`get_physical_address`)
    - RISC-V 的 tlb_fill 回调（由 CPU class 指向）：cpu_helper.c (`riscv_cpu_tlb_fill`)

流程：

1. 程序访问内存时，生成的代码/helper 调用 `load_helper` / `atomic_mmu_lookup` / `tlb_vaddr_to_host` 等。
2. 这些 helper 先在 cputlb.c 的软件 TLB 中检查（`tlb_hit`）；若命中则使用 entry 中的 host addend 直接获得 host 地址。
3. 若未命中，尝试从 victim-TLB 恢复（`victim_tlb_hit`）；若仍未命中，调用 `tlb_fill`。
4. `tlb_fill` 调用 CPU 的 `cc->tlb_fill` 回调（RISC-V 为 `riscv_cpu_tlb_fill`），该回调调用 `get_physical_address` 做页表 walk 并返回物理页与权限。
5. `riscv_cpu_tlb_fill` 在成功后调用 `tlb_set_page` / `tlb_set_page_with_attrs` 将映射加入软件 TLB。随后内存访问重试并命中 TLB。

终端2：

```
# qemu-tlb-trace.gdb
set pagination off
```

```
# Print helper for env and cpu
define print_env
  printf "env->pc=%#lx env->priv=%d\n", env->pc, env->priv
end

# 1) Observe TCG attempting a translation / page lookup (vaddr->host addr path)
# Probe functions in accel/tcg/cputlb.c
break accel/tcg/cputlb.c:tlb_vaddr_to_host
commands
  silent
  printf "== hit tlb_vaddr_to_host (vaddr lookup)\n"
  # arg: env, addr, access_type, mmu_idx
  # try to print addr and access_type (may be optimized inlined)
  print/x addr
  print access_type
  print mmu_idx
  continue
end

break accel/tcg/cputlb.c:get_page_addr_code
commands
  silent
  printf "== hit get_page_addr_code (inst fetch host addr)\n"
  print/x addr
  continue
end

break accel/tcg/cputlb.c:atomic_mmu_lookup
commands
  silent
  printf "== hit atomic_mmu_lookup (RMW path)\n"
  print/x addr
  print/x retaddr
  continue
end

# 2) TLB fill / set page
break accel/tcg/cputlb.c:tlb_fill
commands
  silent
  printf "== hit tlb_fill (soft TLB miss -> fill)\n"
  print/x addr
  print size
  print access_type
  print mmu_idx
  # retaddr is return address into generated code
  print/x retaddr
  continue
end

break accel/tcg/cputlb.c:tlb_set_page_with_attrs
commands
```

```
  silent
  printf "== hit tlb_set_page_with_attrs (inserting TLB entry)\n"
  print/x vaddr
  print/x paddr
  print prot
  print mmu_idx
  print size
  continue
end

break accel/tcg/cputlb.c:tlb_set_page
commands
  silent
  printf "== hit tlb_set_page (wrapper)\n"
  print/x vaddr
  print/x paddr
  print prot
  print mmu_idx
  continue
end

# 3) Observe runtime call into target riscv tlb_fill / page-walk
break target/riscv/cpu_helper.c:riscv_cpu_tlb_fill
commands
  silent
  printf "== hit riscv_cpu_tlb_fill (target TLB fill)\n"
  print/x address
  print size
  print access_type
  print mmu_idx
  # print result of get_physical_address after it runs
  continue
end

break target/riscv/cpu_helper.c:get_physical_address
commands
  silent
  printf "== hit get_physical_address (page table walk)\n"
  # print addr being walked
  print/x addr
  printf "(after walk, physical/ prot printed by caller)\n"
  continue
end

# 4) If exception raised during translation, observe path
break target/riscv/op_helper.c:28
commands
  silent
  printf "== hit riscv_raise_exception (exception raised)\n"
  print/x env->pc
  print env->priv
  print ((CPUState *)env_cpu(env))->exception_index
  continue
```

```
end
```

```
(gdb) source qemu-tlb-trace.gdb

Breakpoint 1 at 0x631546496089: file /home/foxcier/riscv/qemu-4.1.1/accel/tcg/cputlb.c, line
1079.
Breakpoint 2 at 0x631546495df4: file /home/foxcier/riscv/qemu-4.1.1/accel/tcg/cputlb.c, line
1025.
Breakpoint 3 at 0x63154649629e: file /home/foxcier/riscv/qemu-4.1.1/accel/tcg/cputlb.c, line
1132.
Breakpoint 4 at 0x6315464957a5: file /home/foxcier/riscv/qemu-4.1.1/accel/tcg/cputlb.c, line
871.
Breakpoint 5 at 0x6315464951b3: file /home/foxcier/riscv/qemu-4.1.1/accel/tcg/cputlb.c, line
700.
Breakpoint 6 at 0x6315464957d4: file /home/foxcier/riscv/qemu-4.1.1/accel/tcg/cputlb.c, line
847.
Breakpoint 7 at 0x63154654f419: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/cpu_helper.c, line 438.

Breakpoint 8 at 0x63154654e8d4: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/cpu_helper.c, line 158.

Breakpoint 9 at 0x63154654d9e8: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/op_helper.c, line 31.

(gdb) c

Continuing.
== hit get_physical_address (page table walk)

$1 = 0xffffffffc02000ee

(after walk, physical/ prot printed by caller)

[Switching to Thread 0x7534abfff640 (LWP 2093)]

== hit get_page_addr_code (inst fetch host addr)

$2 = 0x1000

== hit tlb_fill (soft TLB miss -> fill)

$3 = 0x1000

$4 = 0

$5 = MMU_INST_FETCH

$6 = 3
```

```
$7 = 0x0

== hit riscv_cpu_tlb_fill (target TLB fill)

$8 = 0x1000

$9 = 0

$10 = MMU_INST_FETCH

$11 = 3

== hit get_physical_address (page table walk)

$12 = 0x1000

(after walk, physical/ prot printed by caller)

== hit tlb_set_page (wrapper)

$13 = 0x1000

$14 = 0x1000

$15 = 7

$16 = 3

== hit tlb_set_page_with_attrs (inserting TLB entry)

$17 = 0x1000

$18 = 0x1000

$19 = 7

$20 = 3

$21 = 4096

== hit get_page_addr_code (inst fetch host addr)
$22 = 0x1000
== hit riscv_raise_exception (exception raised)
$23 = 0x1004
$24 = 3
$25 = -1
[Switching to Thread 0x7534b314acc0 (LWP 2091)]
== hit get_physical_address (page table walk)
$26 = 0x0
(after walk, physical/ prot printed by caller)
== hit get_physical_address (page table walk)
$27 = 0x0
(after walk, physical/ prot printed by caller)
```

```
== hit get_physical_address (page table walk)
$28 = 0x0
(after walk, physical/ prot printed by caller)
== hit get_physical_address (page table walk)
$29 = 0x0
(after walk, physical/ prot printed by caller)

...
== hit get_physical_address (page table walk)
$75 = 0x0
(after walk, physical/ prot printed by caller)
== hit get_physical_address (page table walk)
$76 = 0xffffffffc02000ee
(after walk, physical/ prot printed by caller)
== hit get_physical_address (page table walk)
$77 = 0x0
(after walk, physical/ prot printed by caller)
...
== hit get_physical_address (page table walk)
$124 = 0x0
(after walk, physical/ prot printed by caller)
== hit get_physical_address (page table walk)
$125 = 0x0
(after walk, physical/ prot printed by caller)
== hit get_physical_address (page table walk)
$126 = 0x0
(after walk, physical/ prot printed by caller)
```

顺序：

1. TCG 发起地址解析：`get_page_addr_code` / `tlb_vaddr_to_host`（在 cputlb.c）——这发现 TLB 未命中后触发
2. 软 TLB miss → `tlb_fill`（`accel/tcg/cputlb.c:tlb_fill`）
3. 进入 target 的填充回调：`riscv_cpu_tlb_fill`（cpu_helper.c）
4. 在 `riscv_cpu_tlb_fill` 内调用 `get_physical_address` 做页表 walk（cpu_helper.c）
5. 若成功，调用 `tlb_set_page[_with_attrs]` 插入 TLB（cputlb.c）
6. 之后再次请求页代码地址（`get_page_addr_code`）就能拿到 host 地址；若失败则会走异常路径（你看到的 `riscv_raise_exception`）

输出中各字段：

- `$3/$8/$13/... = 0x1000`：被传入的虚拟地址（`addr` / `address` / `vaddr`）。
- `$4 = 0` / `$21 = 4096`：`size` 参数（0 表示 default / page-size 情况），或者 `tlb_set_page_with_attrs` 的 `size`（4096）。
- `$5 = MMU_INST_FETCH`：访问类型（说明是指令抓取）。
- `$15/$19 = 7`（`prot`）：权限位（通常是 PAGE_READ|PAGE_WRITE|PAGE_EXEC 的组合）。
- `tlb_set_page` / `tlb_set_page_with_attrs` 出现且 `paddr` 与 `vaddr` 被打印，说明成功得到了物理页并把映射插入软件 TLB。

- `== hit riscv_raise_exception` 出现表示某处页表检查或 PMP/对齐等失败，走了异常路径（你看到的 `env->pc` / `exception_index` 可用于判断是 page-fault 还是非法指令等）。

大量 `get_physical_address` 输出是你在断点处打印的 `addr` 参数（不是函数返回值），因此不同线程 / 不同调用上下文会传入不同地址；某些调用用 `addr == 0` 是正常的（或表示 probe/non-faulting路径），也可能是来自其他 hart/线程。

、

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（提示：可以尝试找一条未开启虚拟地址空间的访存语句进行调试，看看调用路径，和开启虚拟地址空间之后的访存语句对比）

QEMU 的 TLB 是软件数据结构（含 host-specific 字段如 `addend`、`TLB_MMIO`、`TLB_RECHECK` 等），与硬件 TLB 在实现细节、原子性、替换策略和可观测性上有显著区别。QEMU 实现遵循"先查 TLB；TLB miss -> 调用 `tlb_fill()` -> 由目标架构的 riscv_cpu_tlb_fill() 做页表遍历 -> 成功后由 `tlb_set_page_with_attrs()` 填充 TLB"的流程。

QEMU 的"软件 TLB"与真实 CPU 的 TLB 在逻辑上的主要差别：

- 实体性质
  - 硬件 TLB：由处理器内建、与 MMU 硬件紧密耦合、并行且低延迟（用在瞬时物理地址翻译）。
  - QEMU 软件 TLB（见 cputlb.c）：纯软件数据结构和查找/填充逻辑，运行在主机线程上，翻译由 C 代码实现并且比硬件慢很多。
- 命中语义与存储内容
  - 硬件：保存真实 PPN、ASID、权限位（在硬件内判断并强制执行），可能支持多级 associativity、快表并行查找。
  - QEMU：保存翻译后的"host 地址 addend"（host 偏移）加上标志（`TLB_MMIO`、`TLB_RECHECK`、`TLB_NOTDIRTY` 等）与 `prot`，并把虚拟页映射到 host memory region 信息，用于帮助快速访问主机内存而不是直接保存 guest PPN。
- 权限与异常处理位置
  - 硬件：权限检查在 MMU/TLB 硬件层面完成，非法访问直接产生异常。
  - QEMU：helpers（`load_helper`/`store_helper` 等）负责先查软件 TLB；未命中时走 `riscv_cpu_tlb_fill` 做页表 walk 并在软件中做权限检查，再由 `tlb_set_page` 填入条目；如果检查失败，QEMU 在软件中触发异常（`raise_mmu_exception`）。
- 页面表 A/D（Accessed/Dirty）位更新
  - 硬件：通常由 MMU 自动在硬件层更新 A/D；原子性由硬件保证。
  - QEMU：页面走查时通过软件判断并尝试用 `atomic_cmpxchg` 在 RAM 上原子更新 PTE；若 PTE 在 IO/ROM 或 CAS 失败会返回失败或重试（`goto restart`）。因此更新是显式且受软件/host 内存映射限制的。
- MMIO / 非 RAM 页处理
  - 硬件：MMIO 通常映射为特权不可缓存区域，硬件直接处理。
  - QEMU：会在 TLB 条目上设置 `TLB_MMIO` / `TLB_RECHECK`，helpers 在命中时检测并走 IO helper（`io_readx`/`io_writex`）或重新检查权限/边界。
- 同步、射频（shootdown）与并发

    - 硬件：在多核上有硬件 TLB shootdown / ASID 支持，硬件保证并发语义。
    - QEMU：需要软件同步（`tlb_flush`、spinlock、async_run_on_cpu 等）；不同 vCPU 间同步、TLB 大小调整、victim-TLB 管理等由 cputlb.c 控制，开销更大且实现策略可变。
- 可重入 / 重试行为
    - 硬件：页表更新/竞争由 ISA/硬件保证的一致性语义处理。
    - QEMU：如果 `atomic_cmpxchg` 失败，软件会 `goto restart` 重走 page-walk；这会在调试时出现可见的重试路径（你能在源码中看到）。
- 可见性与可调试性
    - 硬件：TLB 状态对软件不可直接读取（通常受限），调试需要内核/硬件支持。
    - QEMU：TLB 是普通 C 数据结构，任意断点/打印都可观察，且含有 host 映射信息（利于理解 guest→host 映射）。
- 语义差异总结（影响调试/性能/正确性）
    - QEMU 的 TLB 更像「缓存的翻译后主机访问信息 + 一些标志」，而不是对 guest 硬件 TLB 的 1:1 模拟。
    - 因为软件实现，某些 race / timing /架构特性（如硬件 prefetch、并行查找延迟、TLB shootdown 时序）可能与真实硬件差异较大。
    - QEMU 为可移植性和正确性在软件层做了更多检查（PMP、A/D 写回的原子 CAS、MMIO 标记、TLB_RECHECK），这些在硬件上通常透明或以不同机制实现。
- 调试建议（基于上面差异）
    - 若要验证 guest 行为偏差，检查是否因为 `TLB_RECHECK` / `TLB_MMIO` 标志或 A/D CAS 导致不同路径。
    - 在 `tlb_set_page_with_attrs`（cputlb.c）处观察 `MemoryRegion` 与 `addend`，确认 host 映射正确。
    - 在 `get_physical_address`（cpu_helper.c）观察 PTE、`updated_pte` 与 CAS 重试以确认软件更新语义。

lab5

准备工作：

```
(gdb) add-symbol-file obj/__user_exit.out
add symbol table from file "obj/__user_exit.out"
(y or n) y
Reading symbols from obj/__user_exit.out...
(gdb) break user/libs/syscall.c:18
Breakpoint 1 at 0x8000f8: file user/libs/syscall.c, line 19.
```

1.在大模型的帮助下，完成整个调试的流程，观察一下ecall指令和sret指令是如何被qemu处理的，并简单阅读一下调试中涉及到的qemu源码，解释其中的关键流程。

1.1 ecall指令

流程：译码(ecall) → 翻译器 emit exception helper → 运行时 helper 设置 `cs->exception_index` 并退出 TB → 主循环调用 riscv_cpu_do_interrupt 完成 trap 分派并跳到 trap 向量。

关键代码位置（按顺序）

- 翻译阶段（把 ecall 变成对 raise-helper 的调用）：trans_privileged.inc.c:21 (函数 `trans_ecall`)。
- 生成 helper（把异常号传给 helper）：translate.c:87 (函数 `generate_exception`)。
- helper / 运行时抛出异常：op_helper.c:36(`helper_raise_exception`) → op_helper.c:28 (riscv_raise_exception)，该函数设置 `cs->exception_index` 并调用 `cpu_loop_exit_restore`。
- trap 分派与寄存器设置（把 U-level ecall 根据当前特权转换为相应的 cause，设置 sepc/mepc/... 并切换模式）：cpu_helper.c:503 (riscv_cpu_do_interrupt)，ecall 映射表在 cpu_helper.c:518 (`ecall_cause_map`)。

终端3：

```
0x0000000000001000 in ?? ()
(gdb) add-symbol-file obj/__user_exit.out
add symbol table from file "obj/__user_exit.out"
(y or n) y
Reading symbols from obj/__user_exit.out...
(gdb) break user/libs/syscall.c:18
Breakpoint 1 at 0x8000f8: file user/libs/syscall.c, line 19.
(gdb) b *0x800104
Breakpoint 2 at 0x800104: file user/libs/syscall.c, line 19.
(gdb) c
Continuing.

Breakpoint 1, syscall (num=2) at user/libs/syscall.c:19
19          asm volatile (
(gdb) x/10i $pc
=> 0x8000f8 <syscall+32>:       ld      a0,8(sp)
   0x8000fa <syscall+34>:       ld      a1,40(sp)
   0x8000fc <syscall+36>:       ld      a2,48(sp)
   0x8000fe <syscall+38>:       ld      a3,56(sp)
   0x800100 <syscall+40>:       ld      a4,64(sp)
   0x800102 <syscall+42>:       ld      a5,72(sp)
   0x800104 <syscall+44>:       ecall
   0x800108 <syscall+48>:       sd      a0,28(sp)
   0x80010c <syscall+52>:       lw      a0,28(sp)
   0x80010e <syscall+54>:       addi    sp,sp,144
(gdb) c
Continuing.

Breakpoint 2, 0x0000000000800104 in syscall (num=2) at user/libs/syscall.c:19
```

```
19          asm volatile (
(gdb) si
```

终端2:

```
# source qemu-ecall-breaks.gdb
set pagination off

# 1) ecall translation (trans_ecall)
break target/riscv/insn_trans/trans_privileged.inc.c:21
commands
  silent
  printf "== hit trans_ecall (ecall translated)\n"
  bt 5
  continue
end

# 2) generate_exception (translator emits helper)
break target/riscv/translate.c:87
commands
  silent
  printf "== hit generate_exception (translator emitted raise helper)\n"
  continue
end

# 3) runtime: riscv_raise_exception (sets exception_index and exits TB)
# riscv_raise_exception is at line ~28 in op_helper.c
break target/riscv/op_helper.c:28
commands
  silent
  printf "== hit riscv_raise_exception\n"
  # Try printing useful state (may be optimized out depending on build)
  printf "env->pc=%#lx priv=%d\n", env->pc, env->priv
  # CPUState pointer and exception_index
  print ((CPUState *)env_cpu(env))->exception_index
  continue
end

# 4) trap dispatch: riscv_cpu_do_interrupt
break target/riscv/cpu_helper.c:503
commands
  silent
  printf "== hit riscv_cpu_do_interrupt (trap dispatch)\n"
  printf "env->pc=%#lx env->priv=%d\n", env->pc, env->priv
  # print cause and relevant CSRs if available
  print env->mcause
  print env->mepc
  continue
end

# 5) inspect ecall mapping point (ecall_cause_map)
break target/riscv/cpu_helper.c:518
```

```
commands
  silent
  printf "== hit ecall mapping location\n"
  continue
end
```

```
(gdb) source qemu-ecall-breaks.gdb
Breakpoint 1 at 0x641924032269: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/insn_trans/trans_privileged.inc.c, line 24.
Breakpoint 2 at 0x641924027c40: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/translate.c, line 89.
Breakpoint 3 at 0x6419240349e8: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/op_helper.c, line 31.
Breakpoint 4 at 0x641924036678: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/cpu_helper.c, line 507.
Breakpoint 5 at 0x6419240366ff: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/cpu_helper.c, line 525.
(gdb) continue
Continuing.
[Switching to Thread 0x7ba91ca80640 (LWP 2281)]
== hit trans_ecall (ecall translated)
#0  trans_ecall (ctx=0x7ba91ca7f5a0, a=0x7ba91ca7f4a0) at /home/foxcier/riscv/qemu-
4.1.1/target/riscv/insn_trans/trans_privileged.inc.c:24
#1  0x0000641924202be2d in decode_insn32 (ctx=0x7ba91ca7f5a0, insn=115) at
target/riscv/decode_insn32.inc.c:1614
#2  0x0000641924034475 in decode_opc (ctx=0x7ba91ca7f5a0) at /home/foxcier/riscv/qemu-
4.1.1/target/riscv/translate.c:746
#3  0x0000641924034684 in riscv_tr_translate_insn (dcbase=0x7ba91ca7f5a0,
cpu=0x64193f7b6870) at /home/foxcier/riscv/qemu-4.1.1/target/riscv/translate.c:800
#4  0x0000641923fa70b9 in translator_loop (ops=0x641924897680 <riscv_tr_ops>,
db=0x7ba91ca7f5a0, cpu=0x64193f7b6870, tb=0x7ba916000040 <code_gen_buffer+19>, max_insns=1)
at /home/foxcier/riscv/qemu-4.1.1/accel/tcg/translator.c:95
== hit generate_exception (translator emitted raise helper)
== hit riscv_raise_exception
env->pc=0x800104 priv=0
$1 = -1
== hit riscv_cpu_do_interrupt (trap dispatch)
env->pc=Cannot access memory at address 0x1f924500d71
```

解释：

- 命中 `trans_ecall`：译码/翻译阶段识别到 `ecall`，翻译器把它生成为“抛出异常”的代码（即 emit exception helper）。
- 命中 `generate_exception`：翻译器真实 emit 出对 raise-helper 的调用（在 TB 内生成 helper 调用）。
- 命中`riscv_raise_exception`：运行时 helper 被调用；`env->pc=0x800104` 是发生 ecall 的来宾 PC，`priv=0` 表示 U（user）模式。该函数会设置 `cs->exception_index` 并调用退出/恢复入口，让主循环处理异常。
- 命中`riscv_cpu_do_interrupt`：QEMU 主循环进入 trap 分派路径，按照特权/委托规则设置 sepc/mepc、

mcause/scause、badaddr 并跳到对应的 trap 向量。

1.2 sret指令

关键代码与流程

- `sret` 在译码/翻译层：见 trans_privileged.inc.c:43-51（函数 `trans_sret`）。译器生成对 helper 的调用：`gen_helper_sret(cpu_pc, cpu_env, cpu_pc)` 并随后 `exit_tb()`。
- helper 的接口（生成器声明）：见helper.h:70-76（`DEF_HELPER_2(sret, tl, env, tl)`），表明 `sret` helper 返回一个 `tl`（target_long）。
- 运行时实现：见 op_helper.c:74-96（`helper_sret`）——要点：
  - 检查 `env->priv >= PRV_S`（若不满足，抛非法指令异常）。
  - 读取 `retpc = env->sepc`，检查对齐（RVC 与否）。
  - 检查和拒绝在某些 `mstatus` 条件下（如 TSR）。
  - 更新 `mstatus` 字段（把 `SPIE`/`SPP`/`mstatus` 调整为返回前的状态）、调用 `riscv_cpu_set_mode(env, prev_priv)` 切换到先前权限，然后把 `env->mstatus` 写回。
  - 返回 `retpc`（TCG 将使用返回值作为下一个 PC，并因 `exit_tb()` 退出 TB，由主循环继续执行所返回的 PC）。

一句话总结：译码阶段把 `sret` 翻译为对 `gen_helper_sret` 的调用；运行时 `helper_sret` 验证特权/对齐、更新 `mstatus` 并切换特权级，返回 `sepc` 作为返回地址；翻译器通过 `exit_tb()` 或接收到 helper 返回值来使控制流跳转到该返回地址。

终端3:

```
b kern/trap/trapentry.S:133
```

终端2:

```
# qemu-sret-breaks.gdb
set pagination off

# 1) sret translation (trans_sret)
break target/riscv/insn_trans/trans_privileged.inc.c:43
commands
  silent
  printf "== hit trans_sret (sret translated)\n"
  bt 5
  continue
end

# 2) point where helper is emitted (gen_helper_sret)
break target/riscv/insn_trans/trans_privileged.inc.c:46
commands
  silent
  printf "== translator emitted gen_helper_sret\n"
```

```
  continue
end

# 3) runtime helper: helper_sret
break target/riscv/op_helper.c:74
commands
  silent
  printf "== hit helper_sret\n"
  # Print return target and status
  print/x env->sepc
  print/x env->mstatus
  print env->priv
  # Show relevant mstatus fields if available
  continue
end

# 4) (optional) trap/dispatch entry to observe mode switch if it occurs
break target/riscv/cpu_helper.c:503
commands
  silent
  printf "== hit riscv_cpu_do_interrupt (trap dispatch)\n"
  print cs->exception_index
  print/x env->sepc
  print/x env->mepc
  print/x env->mcause
  print env->priv
  continue
end
```

```
(gdb) source qemu-sret-breaks.gdb
Breakpoint 1 at 0x57cf68c46306: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/insn_trans/trans_privileged.inc.c, line 46.
Breakpoint 2 at 0x57cf68c46306: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/insn_trans/trans_privileged.inc.c, line 46.
Breakpoint 3 at 0x57cf68c48c44: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/op_helper.c, line 76.
Breakpoint 4 at 0x57cf68c4a678: file /home/foxcier/riscv/qemu-
4.1.1/target/riscv/cpu_helper.c, line 507.
(gdb) c
Continuing.
[Switching to Thread 0x7466ae04a640 (LWP 2060)]
== hit trans_sret (sret translated)
#0  trans_sret (ctx=0x7466ae0495a0, a=0x7466ae0494a0) at /home/foxcier/riscv/qemu-
4.1.1/target/riscv/insn_trans/trans_privileged.inc.c:46
#1  0x000057cf68c3ff05 in decode_insn32 (ctx=0x7466ae0495a0, insn=270532723) at
target/riscv/decode_insn32.inc.c:1638
#2  0x000057cf68c48475 in decode_opc (ctx=0x7466ae0495a0) at /home/foxcier/riscv/qemu-
4.1.1/target/riscv/translate.c:746
#3  0x000057cf68c48684 in riscv_tr_translate_insn (dcbase=0x7466ae0495a0,
cpu=0x57cf96812870) at /home/foxcier/riscv/qemu-4.1.1/target/riscv/translate.c:800
```

```
#4  0x000057cf68bbb0b9 in translator_loop (ops=0x57cf694ab680 <riscv_tr_ops>,
db=0x7466ae0495a0, cpu=0x57cf96812870, tb=0x7466a6000040 <code_gen_buffer+19>, max_insns=1)
at /home/foxcier/riscv/qemu-4.1.1/accel/tcg/translator.c:95
== hit helper_sret
$1 = 0xffffffffc0203de6
$2 = 0x8000000000046120
$3 = 1
```

流程:

- 命中 `trans_sret`：译码/翻译把 `sret` 识别并生成 helper 调用（TCG 内生成 `gen_helper_sret`），随后 `exit_tb()`。
- 命中 `helper_sret`：运行时进入 `helper_sret`（在 C 实现里执行权限/对齐检查、更新 `mstatus`、切换特权并返回 `sepc`）。TCG 会将 helper 的返回值当作 next-PC 并退出 TB，从而完成返回。

打印的值含义（对应脚本中顺序）：

- `$1 = 0xffffffffc0203de6` —— 这是 `env->sepc`（helper 读取并返回的 `sepc`）。它是 CPU 要返回到的 guest PC（64 位表示）。通常这是 helper 想要跳回的地址。
- `$2 = 0x8000000000046120` —— 这是 `env->mstatus` 的原始位域值（十六进制）。要解码关键位，可参考 cpu_bits.h（例如 `MSTATUS_SPIE = 0x20`, `MSTATUS_SPP = 0x100`, `MSTATUS_MPP = 0x1800` 等）。
- `$3 = 1` —— 这是 `env->priv`，值 `1` 表示当前在 Supervisor 模式（`PRV_S`），说明执行 `sret` 时处于 S 模式（符合 `sret` 的语义）。

`helper_sret` 的主要动作（参考 op_helper.c）：

- 检查 `env->priv >= PRV_S`，否则抛出非法指令。
- 读取 `retpc = env->sepc` 并对齐检查（RVC 情况除外）。
- 在某些 `mstatus` 标志（如 `TSR`）被置位时抛出异常。
- 计算 `prev_priv = get_field(mstatus, MSTATUS_SPP)`（即返回前的权限），把 `SPIE` 设置为先前的 `SIE`，清 `SPIE`，把 `SPP` 设为 `PRV_U`，调用 `riscv_cpu_set_mode(env, prev_priv)` 切换到先前权限，然后写回 `env->mstatus`。
- 返回 `retpc`；译器/TCG 用返回值作为下一条要执行的 PC，并从 TB 退出。

2.在执行ecall和sret这类汇编指令的时候，qemu进行了很关键的一步——指令翻译（TCG Translation），了解一下这个功能，思考一下另一个双重gdb调试的实验是否也涉及到了一些相关的内容。

TCG translation：把一段 guest 指令序列（TB）编译成等效的宿主机器代码（TCG 中间表示再转为 host 指令），并在需要时在 TB 中插入对"helper"的调用（helper 是运行时的 C 函数），或生成对 QEMU 内存访问 helpers 的 tcg_gen_qemu_ld/st。翻译阶段发生在 [target/riscv/translate.c] 的译码/生成流程（例如 `trans_ecall`/`trans_sret`），并用 `exit_tb()` 或 `tcg_gen_exit_tb()` 控制 TB 边界与返回。

翻译只是生成/缓存代码；实际的地址翻译（VA→PA）、页表遍历、PTE 更新等是在运行时由生成的 helper 或 TCG 的内存访问路径触发的（不是静态翻译时完成）。

与之前"虚拟地址→物理地址"双重 GDB 实验的关系:

- 共同点：翻译阶段决定了"在哪里/如何"进行内存访问（是内联的 tcg qemu_ld/st，还是调用一个 helper）。因此在翻译时可以看到"将要做内存访问"的代码被生成（例如 translator emit 出 `tcg_gen_qemu_ld_tl` 或 `gen_helper_*`），这部分在 host-GDB 会命中 translate.c、`trans_*`、`generate_exception()` 等位置。
- 不同点：实际的 VA→PA 查找/页表 walk 是运行时行为，发生在：
  - TCG 的内存访问路径（宿主代码执行时）会调用 TCG 的 load/store helpers（例如 `load_helper` / `store_helper`），
  - soft-TLB 层（cputlb.c）在 miss 时调用 `tlb_fill` → 最终调用 target 的 `tlb_fill` 回调（RISC-V 的 `riscv_cpu_tlb_fill`），
  - `riscv_cpu_tlb_fill` 调用 `get_physical_address`（在 cpu_helper.c）做页表遍历、PTE 检查与 A/D 更新。
    因此 VA→PA 实验主要命中的是运行时路径（`cputlb.c`, cpu_helper.c），而不是翻译器本身；但翻译器决定了是否会触发这些 runtime helpers。