操作系统lab5实验报告

小组成员：

陈秋彤（2311815）徐盈蕊（2311344）杨欣瑞（2312246）

练习1: 加载应用程序并执行（需要编码）

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

请在实验报告中简要说明你的设计实现过程。

load_icode第6步的设计实现过程

在 `load_icode` 函数的第6步中，需要设置用户态进程的 `trapframe`，以便进程能够从内核态正确返回到用户态并开始执行应用程序的第一条指令。具体实现如下：

```
    //(6) setup trapframe for user environment
    struct trapframe *tf = current->tf;
    // Keep sstatus
    uintptr_t sstatus = tf->status;
    memset(tf, 0, sizeof(struct trapframe));
    /* LAB5:EXERCISE1 YOUR CODE
     * should set tf->gpr.sp, tf->epc, tf->status
     * NOTICE: If we set trapframe correctly, then the user level process can
return to USER MODE from kernel. So
     *          tf->gpr.sp should be user stack top (the value of sp)
     *          tf->epc should be entry point of user program (the value of sepc)
     *          tf->status should be appropriate for user program (the value of
sstatus)
     *          hint: check meaning of SPP, SPIE in SSTATUS, use them by
SSTATUS_SPP, SSTATUS_SPIE(defined in risv.h)
     */
    // 设置用户栈指针为用户栈顶
    tf->gpr.sp = USTACKTOP;
    // 设置程序入口点为 ELF 文件头中的入口地址
    tf->epc = elf->e_entry;
    // 设置 sstatus：从 CSR 读取当前 sstatus，清除 SPP 位（表示之前是用户态），设置 SPIE
位（保存中断使能状态），清除 SIE 位（当前禁用中断）
    uintptr_t current_sstatus = read_csr(sstatus);
    tf->status = ((current_sstatus & ~SSTATUS_SPP) | SSTATUS_SPIE) &
~SSTATUS_SIE;
```

设计要点：

1. **设置用户栈指针**（`tf->gpr.sp = USTACKTOP`）：
    - 将用户栈指针设置为用户栈的顶部地址，确保进程在用户态有正确的栈空间。
2. **设置程序入口点**（`tf->epc = elf->e_entry`）：

- 将异常程序计数器（EPC）设置为ELF文件头中的入口地址，这是应用程序的第一条指令地址。

3. **设置sstatus寄存器状态**：
   - 清除`SSTATUS_SPP`位：表示之前处于用户态（SPP=0表示用户态，SPP=1表示内核态）
   - 设置`SSTATUS_SPIE`位：保存中断使能状态，使得从用户态返回后能够恢复中断使能
   - 清除`SSTATUS_SIE`位：当前在返回用户态前禁用中断，确保状态切换的原子性

当进程从内核态返回到用户态时，CPU会从`tf->epc`指向的地址开始执行，即应用程序的入口点。

Lab4代码完善：proc_alloc函数

首先，我们需要对`Lab4`的部分代码进行完善，第一个需要完善的是`alloc_proc`函数（注意函数名是`alloc_proc`而不是`proc_alloc`），这个函数用于分配新进程的初始化，在`Lab5`中，我们的`proc_struct`结构体新增了几个成员：

```
struct proc_struct
{
    enum proc_state state;                  // Process state
    int pid;                                // Process ID
    int runs;                               // the running times of Proces
    uintptr_t kstack;                       // Process kernel stack
    volatile bool need_resched;             // bool value: need to be rescheduled
to release CPU?
    struct proc_struct *parent;             // the parent process
    struct mm_struct *mm;                   // Process's memory management field
    struct context context;                 // Switch here to run process
    struct trapframe *tf;                   // Trap frame for current interrupt
    uintptr_t pgdir;                        // the base addr of Page Directroy
Table(PDT)
    uint32_t flags;                         // Process flag
    char name[PROC_NAME_LEN + 1];           // Process name
    list_entry_t list_link;                 // Process link list
    list_entry_t hash_link;                 // Process hash list
    int exit_code;                          // exit code (be sent to parent proc)
    uint32_t wait_state;                    // waiting state
    struct proc_struct *cptr, *yptr, *optr; // relations between processes
};
```

在`alloc_proc`函数中，新增的成员变量初始化如下：

```
        // LAB5 YOUR CODE : (update LAB4 steps)
        /*
         * below fields(add in LAB5) in proc_struct need to be initialized
         *       uint32_t wait_state;                        // waiting state
         *       struct proc_struct *cptr, *yptr, *optr;     // relations between
processes
         */
        memset(proc, 0, sizeof(struct proc_struct));
        proc->state = PROC_UNINIT;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = 0;
        proc->need_resched = 0;
```

```
        proc->parent = NULL;
        proc->mm = NULL;
        proc->tf = NULL;
        proc->pgdir = boot_pgdir_pa;
        proc->flags = 0;
        proc->wait_state = 0;
        proc->cptr = NULL;
        proc->yptr = NULL;
        proc->optr = NULL;
```

Lab4代码完善：do_fork函数

在Lab5中，我们还需要对`do_fork`函数进行完善，主要是两个方面：

更新步骤1：设置父子关系和等待状态

在Lab4中，我们在步骤1中只是简单地设置了子进程的父进程指针。在Lab5中，我们还需要确保当前进程（父进程）的`wait_state`为0：

```
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }

    proc->parent = current;
    if (current->wait_state != 0) {
        current->wait_state = 0;
    }
```

这样做的原因是：如果父进程之前在等待子进程（wait_state != 0），而在fork过程中创建了新的子进程，此时应该清除父进程的等待状态。

更新步骤5：使用set_links设置进程关系链接

在Lab4中，我们直接将进程添加到`proc_list`中。在Lab5中，由于引入了进程间的兄弟关系（通过`cptr`、`yptr`、`optr`指针管理），我们需要使用`set_links`函数来正确设置这些关系：

```
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);
        set_links(proc);
    }
    local_intr_restore(intr_flag);
```

`set_links`函数的功能如下（位于`kern/process/proc.c`的153-165行）：

```
// set_links - set the relation links of process
static void
set_links(struct proc_struct *proc)
{
    list_add(&proc_list, &(proc->list_link));
    proc->yptr = NULL;
    if ((proc->optr = proc->parent->cptr) != NULL)
    {
        proc->optr->yptr = proc;
    }
    proc->parent->cptr = proc;
    nr_process++;
}
```

`set_links`函数完成以下工作：

1. **将进程添加到进程列表**：`list_add(&proc_list, &(proc->list_link))`
2. **设置进程的年轻兄弟指针**：`proc->yptr = NULL`（新创建的子进程是最年轻的）
3. **建立与年长兄弟的关系**：
   - 将父进程的第一个子进程（如果存在）设为新进程的年长兄弟：`proc->optr = proc->parent->cptr`
   - 如果年长兄弟存在，更新其年轻兄弟指针指向新进程：`proc->optr->yptr = proc`
4. **更新父进程的子进程指针**：`proc->parent->cptr = proc`（新进程成为父进程的第一个子进程）
5. **增加进程计数**：`nr_process++`

这样，进程之间的父子关系和兄弟关系就被正确地建立起来了，这对于后续的进程管理（如`do_wait`、`do_exit`等）非常重要。

完整的do_fork实现

完整的`do_fork`函数核心实现如下（位于`kern/process/proc.c`的471-511行）：

```
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }

    proc->parent = current;
    if (current->wait_state != 0) {
        current->wait_state = 0;
    }

    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }

    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }

    copy_thread(proc, stack, tf);

    bool intr_flag;
```

```
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);
        set_links(proc);
    }
    local_intr_restore(intr_flag);

    wakeup_proc(proc);

    ret = proc->pid;

fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
```

这里的`wait_state`表示**进程的等待状态**，用于标识进程当前在等待什么事件。在ucore中，`wait_state`有以下用途：

1. `WT_CHILD`：表示进程正在等待子进程退出。当父进程调用`do_wait`等待子进程时，如果子进程尚未处于`PROC_ZOMBIE`状态，父进程会将`wait_state`设置为`WT_CHILD`并进入睡眠状态：

```
    if (haskid)
    {
        current->state = PROC_SLEEPING;
        current->wait_state = WT_CHILD;
        schedule();
```

2. `WT_INTERRUPTED`：表示等待状态可以被中断。当进程被kill时，如果进程处于可中断的等待状态，会唤醒该进程。
3. **初始化为0**：新创建的进程`wait_state`初始化为0，表示没有在等待任何事件。

用户态进程执行流程：从RUNNING态到执行第一条指令

当用户态进程被ucore调度器选择并占用CPU执行（RUNNING态）后，到具体执行应用程序第一条指令的整个过程如下：

1. 进程调度选择

调度器（如`schedule()`函数）从可运行进程队列中选择一个进程，调用`proc_run(proc)`切换到该进程。

2. 进程上下文切换（`proc_run`）

```
void proc_run(struct proc_struct *proc)
{
    if (proc != current)
    {
```

```
        // LAB4:EXERCISE3 YOUR CODE
        /*
         * Some Useful MACROs, Functions and DEFINEs, you can use them in below
implementation.
         * MACROs or Functions:
         *   local_intr_save():        Disable interrupts
         *   local_intr_restore():     Enable Interrupts
         *   lsatp():                   Modify the value of satp register
         *   switch_to():              Context switching between two processes
         */
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);
        {
            current = proc;
            lsatp(next->pgdir);
            switch_to(&prev->context,&next->context);
        }
        local_intr_restore(intr_flag);
    }
}
```

- 关闭中断，保护上下文切换过程
- 更新当前进程指针`current = proc`
- 切换页目录：`lsatp(next->pgdir)`，加载新进程的页表基址
- 调用`switch_to`进行上下文切换，保存旧进程的寄存器上下文，恢复新进程的寄存器上下文

3. 执行forkret入口

`switch_to`恢复新进程的`context`后，由于`context.ra`在`copy_thread`中被设置为`forkret`的地址：

```
    proc->context.ra = (uintptr_t)forkret;
    proc->context.sp = (uintptr_t)(proc->tf);
```

CPU会跳转到`forkret`函数执行：

```
static void
forkret(void)
{
    forkrets(current->tf);
}
```

4. 执行forkrets恢复trapframe

`forkret`调用`forkrets(current->tf)`，这是一个汇编函数：

```
    .globl forkrets
forkrets:
    # set stack to this new process's trapframe
    move sp, a0
    j __trapret
```

`forkrets`将栈指针设置为trapframe的地址（通过参数a0传入），然后跳转到`__trapret`。

5. 执行__trapret恢复所有寄存器

`__trapret`执行`RESTORE_ALL`宏，恢复所有保存的寄存器：

```
    .globl __trapret
__trapret:
    RESTORE_ALL
    # return from supervisor call
    sret
```

`RESTORE_ALL`宏会：

- 恢复所有通用寄存器（x1-x31）
- 恢复`sepc`寄存器（从`tf->epc`）
- 恢复`sstatus`寄存器（从`tf->status`）
- 恢复栈指针`sp`（从`tf->gpr.sp`，即用户栈顶`USTACKTOP`）

6. 执行sret返回用户态

执行`sret`指令后：

- CPU从`sepc`寄存器读取地址（即`elf->e_entry`，应用程序的入口点）
- 根据`sstatus`中的`SPP`位，CPU切换到用户态
- 根据`sstatus`中的`SPIE`位，恢复中断使能状态
- CPU开始从用户态入口地址执行第一条指令

总结

整个流程可以概括为：
调度器选择进程 → proc_run切换进程 → switch_to切换上下文 → forkret → forkrets → __trapret恢复寄存器 → sret返回用户态 → 执行应用程序第一条指令（elf->e_entry）

这个过程确保了进程能够从内核态安全地返回到用户态，并且从正确的入口地址开始执行用户程序。

练习2: 父进程复制自己的内存空间给子进程（需要编码）

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

请在实验报告中简要说明你的设计实现过程。

copy_range函数的设计实现过程

`copy_range`函数的作用是将一个进程（进程A，通常是父进程）的用户内存地址空间中的内容复制到另一个进程（进程B，通常是子进程）的内存空间中。这个函数在`do_fork`调用链中被使用：`copy_mm` → `dup_mmap` → `copy_range`。

函数参数和整体流程

函数签名如下：

```
int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool
share)
```

- `to`：目标进程（子进程）的页目录表地址
- `from`：源进程（父进程）的页目录表地址
- `start`：要复制的内存范围的起始地址
- `end`：要复制的内存范围的结束地址
- `share`：是否共享内存（在Lab5中不使用，始终使用复制方式）

函数按页（PGSIZE）为单位逐页复制内存内容：

```
do
{
    // 获取父进程的页表项
    pte_t *ptep = get_pte(from, start, 0), *nptep;
    if (ptep == NULL)
    {
        start = ROUNDDOWN(start + PTSIZE, PTSIZE);
        continue;
    }
    // 如果页表项有效，进行复制
    if (*ptep & PTE_V)
    {
        // ... 复制逻辑
    }
    start += PGSIZE;
} while (start != 0 && start < end);
```

核心实现步骤

对于每个有效的页，需要完成以下四个步骤：

步骤1：获取父进程物理页的内核虚拟地址

```
// (1) 获取父进程物理页的内核虚拟地址
void *src_kvaddr = page2kva(page);
```

通过`page2kva`宏将父进程的物理页`Page`结构转换为内核虚拟地址，这样我们就可以在内核态访问这个页的内容。

步骤2：获取子进程新物理页的内核虚拟地址

```
// (2) 获取子进程新物理页的内核虚拟地址
void *dst_kvaddr = page2kva(npage);
```

同样地，为子进程新分配的物理页获取内核虚拟地址，作为内存复制的目标地址。

步骤3：复制内存内容

```
// (3) 复制内存内容：从父进程的物理页复制到子进程的新物理页
memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
```

使用`memcpy`函数将一个页（PGSIZE大小，通常是4KB）的内容从父进程的物理页复制到子进程的新物理页。这样，子进程就有了与父进程相同的页面内容。

步骤4：建立页表映射

```
// (4) 建立页表映射：将子进程的虚拟地址 start 映射到新物理页 npage
ret = page_insert(to, npage, start, perm);
assert(ret == 0);
```

使用`page_insert`函数在子进程的页目录表中建立虚拟地址`start`到新物理页`npage`的映射，权限从父进程的页表项中获取（`perm = (*ptep & PTE_USER)`）。

完整的实现代码

完整的实现代码如下（位于`kern/mm/pmm.c`的425-433行）：

```
            // (1) 获取父进程物理页的内核虚拟地址
            void *src_kvaddr = page2kva(page);
            // (2) 获取子进程新物理页的内核虚拟地址
            void *dst_kvaddr = page2kva(npage);
            // (3) 复制内存内容：从父进程的物理页复制到子进程的新物理页
            memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
            // (4) 建立页表映射：将子进程的虚拟地址 start 映射到新物理页 npage
            ret = page_insert(to, npage, start, perm);
            assert(ret == 0);
```

设计要点

1. **逐页复制**：内存复制以页为单位进行，这样可以简化页表管理，每页都是一个独立的映射单元。
2. **独立的物理页**：子进程获得的是独立的物理页，而不是与父进程共享同一物理页。这样，子进程和父进程的内存修改不会相互影响（写时复制机制需要额外的支持）。
3. **权限继承**：子进程继承父进程页面的权限位（通过`perm = (*ptep & PTE_USER)`获取）。
4. **页表同步**：通过`page_insert`建立映射后，页表会自动更新，TLB也会被刷新，确保映射生效。

trap.c中时钟中断处理的改进

在测试过程中，发现在时钟中断处理函数`interrupt_handler`的`IRQ_S_TIMER`分支中，需要添加以下代码来支持用户态进程的时间片调度。这段代码应该放在设置下次时钟中断和更新计数器之后：

```
case IRQ_S_TIMER:
    // 设置下次时钟中断
    clock_set_next_event();

    // 更新计数器
    ticks++;
    if (ticks % TICK_NUM == 0) {
```

```
            print_ticks();
            print_times++;
        }

        // 添加的代码：在用户态下标记进程需要调度
        if (current != NULL && (tf->status & SSTATUS_SPP) == 0) {
            current->need_resched = 1;
        }

        // ... 其他代码
        break;
```

关键代码片段位于`kern/trap/trap.c`的142-144行：

```
if (current != NULL && (tf->status & SSTATUS_SPP) == 0) {
    current->need_resched = 1;
}
```

为什么需要这个检查？

1. 时间片轮转调度的需要

在支持用户态进程后，系统需要实现时间片轮转调度机制。当时钟中断发生时，如果当前有进程在用户态运行，应该标记该进程需要重新调度，以便在适当的时机切换到其他进程，实现CPU时间的公平分配。

2. 用户态和内核态的区别

关键点在于需要检查`(tf->status & SSTATUS_SPP) == 0`，这个条件判断中断是否发生在用户态：

- `SSTATUS_SPP`**位为0**：表示中断前CPU处于用户态（User Mode）
- `SSTATUS_SPP`**位为1**：表示中断前CPU处于内核态（Supervisor Mode）

3. 只在用户态触发调度的原因

只在用户态下设置`need_resched`的原因：

1. **内核代码的原子性**：内核代码可能正在执行关键操作（如修改共享数据结构、进行进程切换等），如果在内核态主动触发调度，可能导致：
    - 数据不一致
    - 死锁
    - 不可预测的行为
2. **调度的安全时机**：在`trap`函数的实现中（`kern/trap/trap.c`的280-289行），可以看到只有在非内核态（`!in_kernel`）时才会检查`need_resched`并调用`schedule()`：

```
if (!in_kernel)
{
    if (current->flags & PF_EXITING)
    {
        do_exit(-E_KILLED);
    }
    if (current->need_resched)
    {
        schedule();
    }
}
```

这确保了调度只在从内核返回到用户态之前进行，这是一个安全的时机点。

4. 完整的工作流程

1. **用户态进程运行**：用户态进程正在执行
2. **时钟中断发生**：时钟中断触发，CPU进入内核态处理中断
3. **检查中断发生的位置**：`(tf->status & SSTATUS_SPP) == 0` 判断中断来自用户态
4. **设置调度标志**：`current->need_resched = 1`
5. **中断处理返回**：中断处理完成后，`trap` 函数检查 `need_resched`
6. **调用调度器**：由于不是在内核态，调用 `schedule()` 切换到其他进程

与Lab3的区别

在Lab3中，还没有用户态进程的概念，因此不需要区分用户态和内核态。但在Lab5中，由于引入了用户态进程，必须在中断处理中判断中断发生的模式，只有在用户态才允许调度，这是实现多进程时间片调度的关键机制。

这个改进确保了系统能够正确地实现用户态进程的时间片轮转调度，同时保护了内核代码的执行不受干扰。

练习3: 阅读分析源代码，理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的实现（不需要编码）

简要说明对 fork/exec/wait/exit函数的分析和执行流程

fork (创建进程)

- **用户态: 程序调用 `fork()`，最终执行 `sys_fork`，通过 `ecall` 指令触发系统调用中断，陷入内核。**
- **内核态**
    1. 中断处理程序分发到 `syscall` -> `sys_fork` -> `do_fork`。
    2. `do_fork` 分配新的 `proc_struct` 和内核栈。
    3. `copy_mm`: 复制父进程的内存空间（或建立 COW 共享）。

4. `copy_thread`：复制父进程的 `trapframe`（寄存器状态）到子进程，但将子进程的返回值寄存器 `a0` 设置为 0。
5. 将子进程加入进程链表和调度队列（`wakeup_proc`）。
6. `do_fork` 返回子进程的 PID 给父进程。

返回：父进程从系统调用返回，`a0` 为子进程 PID；子进程被调度执行时，从 `forkret` 开始，最终返回用户态，`a0` 为 0。

exec (执行新程序)

- **用户态: 程序调用 `exec()` (对应 `sys_exec`)，传入可执行文件名等参数，触发系统调用。**
- **内核态**

1. `sys_exec` -> `do_execve`。
2. 检查并回收当前进程的内存空间（`exit_mmap`，`put_pgdir`）。
3. `load_icode`：读取 ELF 格式的二进制文件，建立新的内存映射（代码段、数据段、BSS、用户栈）。
4. **关键**：重置当前进程的 `trapframe`。将 `epc` 设置为新程序的入口地址，`sp` 设置为新的用户栈顶。

返回：系统调用"返回"时，实际上是跳转到了新程序的入口点开始执行，原程序的代码和数据已被替换。

wait (等待子进程)

- **用户态: 父进程调用 `wait()`，触发系统调用。**
- **内核态**

1. `sys_wait` -> `do_wait`。
2. 查找是否有状态为 `PROC_ZOMBIE` (已退出) 的子进程。
3. **如果找到**：回收该子进程剩余的内核资源（如内核栈、`proc_struct`），将子进程的退出码保存在参数中，返回 0。
4. **如果没找到但有运行中的子进程**：将当前进程状态设为 `PROC_SLEEPING`，标记等待原因为 `WT_CHILD`，调用 `schedule()` 让出 CPU。

交互：当子进程调用 `exit` 时会唤醒父进程，父进程从 `schedule()` 返回继续执行回收操作。

exit (进程退出)

- **用户态: 进程执行完毕或出错，调用 `exit()`。**
- **内核态**

1. `sys_exit` -> `do_exit`。
2. 回收大部分内存资源（`mm_destroy`）。
3. 将状态设为 `PROC_ZOMBIE`，设置退出码。
4. 如果有父进程在等待（`WT_CHILD`），唤醒父进程。

5. 将所有子进程过继给 `initproc`。
6. 调用 `schedule()` 切换到其他进程，当前进程不再执行。

哪些操作是在用户态完成，哪些是在内核态完成?

- **用户态完成:**
  - 调用库函数（如 `fork()`，`printf()` 等）进行参数准备。
  - 执行 `ecall` 指令前的参数传递（将系统调用号和参数放入寄存器）。
  - 系统调用返回后的逻辑处理（如判断 `fork` 返回值决定是父进程还是子进程）。
  - `exec` 加载的新程序代码的执行。
- **内核态完成:**
  - **资源管理**: 进程控制块 (`proc_struct`) 的分配与回收、内存页表的建立与销毁 (`do_fork`，`do_exit`，`do_execve`)。
  - **上下文切换**: 修改 `trapframe`，保存/恢复寄存器，切换内核栈。
  - **调度**: `schedule()` 选择下一个运行的进程，`do_wait` 中的睡眠等待。
  - **文件加载**: `load_icode` 解析 ELF 文件并拷贝到内存。

内核态与用户态程序是如何交错执行的?

交错执行主要通过 中断（Trap） 和 调度（Scheduling） 机制实现：

1. **系统调用 (User -> Kernel): 用户程序执行 `ecall`，CPU 模式从 User 切换到 Supervisor，跳转到内核的 `trap` 处理入口。**
2. **调度 (Kernel 内部)**
   - 在 `do_wait` 中，父进程如果需要等待，会主动调用 `schedule()` 放弃 CPU，内核保存当前上下文，切换到另一个进程（如子进程）的上下文。
   - 在 `do_exit` 中，进程执行完清理工作后，调用 `schedule()` 永久放弃 CPU。
   - 时钟中断也可能强制触发 `schedule()`。
3. **中断返回 (Kernel -> User): 调度器选择一个进程后，恢复其内核栈上下文，最后执行 `sret` 指令。CPU 模式切回 User，程序计数器 (PC) 跳转到 `trapframe->epc` 指向的地址（原程序断点或新程序入口）。**

比如: 父进程 `fork` 后继续执行 -> 调用 `wait` 陷入内核 -> 发现子进程未结束 -> 父进程睡眠 (`schedule`) -> 切换到子进程执行 -> 子进程 `exit` 陷入内核 -> 唤醒父进程 -> 子进程僵死 (`schedule`) -> 切换回父进程 -> 父进程从 `wait` 返回用户态。

内核态执行结果是如何返回给用户程序的?

执行结果通过 **修改中断帧 (`trapframe`) 中的寄存器** 返回：

1. 在内核处理完系统调用后（如 `syscall.c` 中的 `syscall` 函数），会将函数的返回值（通常是 `int` 类型）写入当前进程 `trapframe` 的 `a0` 寄存器(`tf->gpr.a0`)

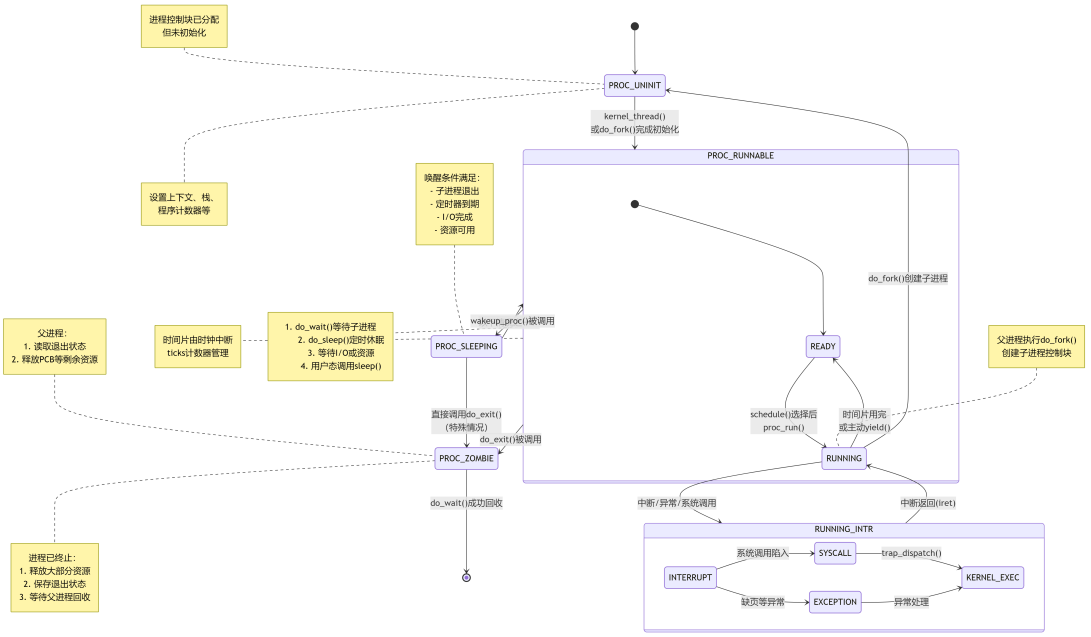
   例如：`tf->gpr.a0 = syscalls[num](arg);`

2. 当内核执行 `trap_return`（或类似的返回汇编代码）时，会将 `trapframe` 中的值恢复到物理寄存器中。
3. 执行 `sret` 返回用户态后，用户程序读取 `a0` **寄存器**，即获取了系统调用的返回值（如 `fork` 返回的 PID，或 `read` 返回的字节数）。

- **特殊情况**: 对于 `fork` 的子进程，内核在 `copy_thread` 中显式将子进程 `trapframe` 的 `a0` 设为 0，因此子进程醒

来时看到的返回值是 0。

给出ucore中一个用户态进程的执行状态生命周期图（包执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）



关键状态说明：

1. PROC_UNINIT（未初始化）

- 刚刚通过 `alloc_proc()` 分配了进程控制块
- 进程结构体中的字段还未正确设置
- 典型事件：进程创建的开始阶段

2. PROC_RUNNABLE（可运行）

- **READY**：进程就绪，等待调度器选择
- **RUNNING**：进程正在CPU上执行
- **转换机制**：
  - `schedule()`：调度器选择下一个进程
  - `proc_run()`：实际切换上下文执行
  - 时钟中断：强制 `RUNNING→READY`

3. PROC_SLEEPING（睡眠等待）

- 进程主动放弃CPU，等待特定事件
- **唤醒机制**：
  - `wakeup_proc()`：当等待条件满足时被调用
  - 通过等待队列管理

4. PROC_ZOMBIE（僵尸状态）

- 进程已终止，但父进程尚未回收
- 保留进程控制块和退出状态码
- 避免"孤儿进程"问题

5. RUNNING_INTR（中断上下文）

- 用户态进程执行被中断/异常/系统调用打断
- 进入内核态执行
- **关键函数**：
  - `trap()`：中断/异常入口
  - `trap_dispatch()`：分发处理
  - `syscall()`：系统调用处理

Challenge1：Copy-on-Write (COW) 机制实现

在Lab5的基础实现中，`copy_range`函数在`do_fork`时会立即复制父进程的所有可写页面到子进程，这会导致大量的内存复制操作，即使子进程可能永远不会修改这些页面。Copy-on-Write (COW) 机制通过延迟复制策略，只在真正需要时才复制页面，从而显著提高fork操作的效率并节省内存。

COW机制的核心思想

COW机制的基本思想是：

1. **fork时共享**：父进程和子进程在fork时共享相同的物理页面，而不是立即复制
2. **标记为只读**：将共享的页面标记为只读，并添加COW标志
3. **写时复制**：当任一进程尝试写入COW页面时，触发页错误，此时才真正复制页面

实现细节

1. PTE_COW标志位的定义

在`kern/mm/mmu.h`中定义了COW标志位：

```
#define PTE_COW  0x200 // Copy-On-Write flag (bit 9)
```

`PTE_COW`使用页表项中保留给软件使用的位（bit 9），与硬件定义的标志位（如PTE_V、PTE_R、PTE_W等）不冲突。这个标志位用于标识一个页面是COW页面，需要特殊处理。

2. copy_range函数的COW实现

在`kern/mm/pmm.c`中，`copy_range`函数被修改为支持COW机制：

```
int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end,
               bool share)
{
    assert(start % PGSIZE == 0 && end % PGSIZE == 0);
    assert(USER_ACCESS(start, end));
    // copy content by page unit.
    do
    {
        // call get_pte to find process A's pte according to the addr start
        pte_t *ptep = get_pte(from, start, 0), *nptep;
        if (ptep == NULL)
        {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue;
        }
        // call get_pte to find process B's pte according to the addr start. If
        // pte is NULL, just alloc a PT
        if (*ptep & PTE_V)
        {
            if ((nptep = get_pte(to, start, 1)) == NULL)
            {
                return -E_NO_MEM;
            }
            uint32_t perm = (*ptep & PTE_USER);
            // get page from ptep
            struct Page *page = pte2page(*ptep);
            assert(page != NULL);
            int ret = 0;
            /* COW Implementation:
             * Instead of copying the page immediately, we share it between
             * parent and child processes. For writable pages, we mark them
             * as read-only with COW flag. When either process tries to write,
             * a page fault will occur and the page will be copied at that time.
             */
            // Check if the page is writable (has PTE_W flag)
            if ((*ptep & PTE_W) && !(*ptep & PTE_COW))
            {
                // This is a writable page that hasn't been marked as COW yet
                // (1) Increment page reference count (child process will also
reference it)
                // Note: We increment before marking as COW because we're about
to share it
                page_ref_inc(page);

                // (2) Mark parent's page as COW and read-only
                *ptep = (*ptep & ~PTE_W) | PTE_COW;
                tlb_invalidate(from, start);

                // (3) Map parent's page to child process, also as read-only with
COW flag
```

```
                // page_insert will increment ref count again (for child's
mapping)
                uint32_t cow_perm = (perm & ~PTE_W) | PTE_COW;
                ret = page_insert(to, page, start, cow_perm);
                assert(ret == 0);
            }
            else
            {
                // Page is already COW or read-only, just share it
                // page_insert will handle the reference count
                ret = page_insert(to, page, start, perm);
                assert(ret == 0);
            }
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

关键实现步骤：

1. **判断页面类型**：
   - 检查页面是否可写（`PTE_W`）且尚未标记为COW（`!PTE_COW`）
   - 只有可写页面才需要COW处理，只读页面（如代码段）可以直接共享
2. **增加引用计数**：

   ```
   page_ref_inc(page);
   ```

   - 在标记COW之前增加引用计数，因为子进程即将共享这个页面
   - 注意：`page_insert`函数内部也会增加引用计数，所以最终引用计数会正确反映共享该页面的进程数
3. **标记父进程页面为COW**：

   ```
   *ptep = (*ptep & ~PTE_W) | PTE_COW;
   tlb_invalidate(from, start);
   ```

   - 清除写权限位（`~PTE_W`），添加COW标志（`PTE_COW`）
   - 使TLB失效，确保新的页表项生效
4. **映射子进程页面**：

   ```
   uint32_t cow_perm = (perm & ~PTE_W) | PTE_COW;
   ret = page_insert(to, page, start, cow_perm);
   ```

   - 子进程的页表项也设置为只读并带有COW标志
   - 父子进程共享同一个物理页面

3. do_pgfault函数的COW页错误处理

在`kern/mm/vmm.c`中，`do_pgfault`函数负责处理COW页错误：

```
    // Check if this is a COW page fault
    if (*ptep & PTE_V && (*ptep & PTE_COW))
    {
        // This is a COW page fault - we need to copy the page
        struct Page *page = pte2page(*ptep);
        if (page == NULL)
        {
            cprintf("pte2page in do_pgfault failed\n");
            goto failed;
        }

        // This is a COW page fault - we need to copy the page if it's a write
fault
        // For read/execute faults on COW pages, the page is already mapped and
readable
        // so we can just return success (the page fault was likely due to other
reasons)
        if (!is_write)
        {
            // Read/execute fault on COW page - page is already mapped, just
return success
            ret = 0;
            goto out;
        }

        // Write fault on COW page - need to copy if multiple references exist
        // If page reference count > 1, we need to copy the page
        if (page_ref(page) > 1)
        {
            // Allocate a new page
            struct Page *npage = alloc_page();
            if (npage == NULL)
            {
                cprintf("alloc_page in do_pgfault failed\n");
                goto failed;
            }

            // Copy the content from old page to new page
            void *src_kvaddr = page2kva(page);
            void *dst_kvaddr = page2kva(npage);
            memcpy(dst_kvaddr, src_kvaddr, PGSIZE);

            // Decrement reference count of old page
            page_ref_dec(page);

            // Map the new page with write permission (remove COW flag)
            uint32_t new_perm = (perm | PTE_R | PTE_W) & ~PTE_COW;
            if (page_insert(mm->pgdir, npage, addr, new_perm) != 0)
            {
                free_page(npage);
                cprintf("page_insert in do_pgfault failed\n");
```

```
                goto failed;
            }
        }
        else
        {
            // Only one reference, we can just remove COW flag and add write
permission
            uint32_t new_perm = (*ptep & PTE_USER) | PTE_R | PTE_W;
            new_perm &= ~PTE_COW;
            *ptep = pte_create(page2ppn(page), new_perm);
            tlb_invalidate(mm->pgdir, addr);
        }
        ret = 0;
    }
```

COW页错误处理的关键逻辑：

1. **检测COW页错误**：
   - 检查页表项是否有效（`PTE_V`）且带有COW标志（`PTE_COW`）
2. **处理读/执行错误**：
   - 如果是读或执行错误（`!is_write`），COW页面已经映射且可读，直接返回成功
   - 这种情况可能发生在TLB失效或其他原因导致的页错误
3. **处理写错误 - 多引用情况**：
   - 当`page_ref(page) > 1`时，说明还有其他进程共享该页面
   - 需要分配新页面并复制内容：

     ```
     struct Page *npage = alloc_page();
     memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
     ```

   - 减少原页面的引用计数：`page_ref_dec(page)`
   - 将新页面映射到当前进程，权限为可读写，移除COW标志
4. **处理写错误 - 单引用情况**：
   - 当`page_ref(page) == 1`时，说明只有当前进程使用该页面
   - 不需要复制，只需修改页表项：
     - 移除COW标志（`~PTE_COW`）
     - 添加写权限（`PTE_R | PTE_W`）
   - 使TLB失效，确保新权限生效

特别注意的细节

1. 引用计数的管理

引用计数是COW机制正确性的关键。需要注意以下几点：

- **copy_range中的引用计数**：
  - 在标记COW之前调用`page_ref_inc(page)`，因为子进程即将共享该页面
  - `page_insert`函数内部也会调用`page_ref_inc`，所以最终引用计数 = 共享该页面的进程数

- **do_pgfault中的引用计数**：
  - 复制页面后，需要调用`page_ref_dec(page)`减少原页面的引用计数
  - 新页面的引用计数由`page_insert`设置为1
- **引用计数为1时的优化**：
  - 当引用计数为1时，不需要复制页面，只需修改页表项
  - 这避免了不必要的内存复制，提高了性能

2. TLB失效的处理

在修改页表项后，必须使TLB失效，确保CPU使用新的页表项：

```
tlb_invalidate(from, start);  // 在copy_range中
tlb_invalidate(mm->pgdir, addr);  // 在do_pgfault中
```

如果不使TLB失效，CPU可能继续使用缓存的旧页表项，导致权限检查错误。

3. 只读页面的处理

只读页面（如代码段）不需要COW处理，可以直接共享：

```
else
{
    // Page is already COW or read-only, just share it
    ret = page_insert(to, page, start, perm);
}
```

这些页面永远不会被写入，因此不需要COW机制。

4. 页错误类型的区分

在`do_pgfault`中需要区分不同类型的页错误：

- **读/执行错误**：COW页面已经映射且可读，直接返回成功
- **写错误**：需要根据引用计数决定是复制还是直接修改权限

这种区分避免了不必要的页面复制。

5. 错误处理

COW实现中需要处理各种错误情况：

- 分配新页面失败：需要释放已分配的资源并返回错误
- 页表项获取失败：需要正确处理并继续处理下一页
- 页面插入失败：需要释放新分配的页面

6. 与原有代码的兼容性

COW实现需要与原有的内存管理代码兼容：

- `page_insert`函数会自动处理引用计数
- `page_remove_pte`函数会自动减少引用计数并在引用计数为0时释放页面

- 这些机制确保了COW页面在不再被引用时能够正确释放

COW机制的工作流程

Fork时的流程

1. 父进程调用 `do_fork`
2. `copy_mm` → `dup_mmap` → `copy_range`
3. 对于每个可写页面：
    - 增加引用计数
    - 将父进程页表项标记为只读+COW
    - 将子进程页表项映射到同一物理页，标记为只读+COW
4. 父子进程共享物理页面，但页表项都标记为只读

写操作时的流程

1. 进程尝试写入COW页面
2. CPU检测到写权限违规，触发页错误
3. 进入 `exception_handler` → `do_pgfault`
4. `do_pgfault` 检测到COW标志：
    - 如果是读错误：直接返回成功
    - 如果是写错误：
        - 引用计数 > 1：分配新页面，复制内容，映射新页面，减少原页面引用计数
        - 引用计数 = 1：直接修改页表项，移除COW标志，添加写权限
5. 返回用户态，重新执行写操作，此时页面已可写

Challenge2：说明该用户程序是何时被预先加载到内存中的？与我们常用操作系统的加载有何区别，原因是什么?

一、lab5 中用户程序的加载时机

ucore 作为教学操作系统，Lab5 中用户程序的“预先加载”并非运行时动态加载，而是编译期嵌入内核 + 内核启动后映射到进程地址空间，全程无磁盘 IO 参与，具体分三个阶段：

1. 编译阶段：用户程序嵌入内核镜像

- 用户程序先被编译为独立 ELF 文件，通过 `user.ld` 链接脚本静态链接到**固定用户态虚拟地址**，且所有用户程序的虚拟地址空间互不重叠；
- 内核编译时，通过链接脚本将用户程序的 ELF 二进制数据直接嵌入内核镜像的只读数据段，成为内核的一部分。

此时用户程序并未真正“加载到内存”，但已固化到内核镜像中，是“预先加载”的核心体现。

2. 内核启动阶段：内核镜像加载到物理内存

- QEMU 启动时，将内核镜像（包含嵌入的用户程序）加载到物理内存的高地址，内核完成自身初始化（页表、进程管理、中断等）后，用户程序的二进制数据已存在于物理内存中（只是属于内核地址空间）。

3. 进程创建阶段：映射到用户地址空间

- 内核创建用户进程（如通过 `do_execve` / `fork`）时，无需从磁盘读取用户程序，只需：
  1. 为进程创建用户态页表（`mm_struct`）；
  2. 将内核中嵌入的用户程序二进制数据，从内核物理地址映射到进程的用户虚拟地址；
  3. 设置页表权限（用户态可访问、可执行），完成用户程序的“最终加载”。

二、与常用操作系统（如 Linux）的核心区别

对比维度	ucore（Lab5）	常用 OS（Linux/Windows）
加载时机	编译期嵌入内核，进程创建时仅做地址映射	运行时（`execve` 系统调用）动态加载
数据来源	内核镜像（物理内存）	磁盘文件系统（ext4/NTFS 等）
加载触发者	内核初始化进程时主动映射	用户进程调用 `execve` 触发内核加载
地址特性	静态链接到固定虚拟地址，无重定位	动态链接（可执行文件+共享库），运行时重定位
依赖组件	无需文件系统、块设备驱动	依赖文件系统、磁盘驱动、动态链接器（ld.so）
加载粒度	整个程序一次性嵌入内核，无按需加载	支持按需加载（缺页触发）、共享库复用

三、区别产生的核心原因

ucore 的设计目标是聚焦操作系统核心机制教学，而非模拟真实 OS 的全功能，因此通过“简化加载流程”降低复杂度，具体原因如下：

1. 规避复杂组件的实现成本

真实 OS 的程序加载依赖：

- 文件系统（解析 ELF/PE 文件、管理文件元数据）；
- 块设备驱动（磁盘 IO、缓存管理）；
- 动态链接器（解析共享库依赖、重定位地址）。

ucore 作为教学 OS，若实现这些组件，会偏离“进程管理、内存管理、系统调用”的核心实验目标，因此选择“嵌入内核”的方式跳过这些复杂环节。

2. 聚焦核心机制（如 COW、进程切换）

Lab5 的核心是“用户进程创建、系统调用、COW 机制”，简化程序加载后：

- 无需处理磁盘 IO 延迟、文件权限等无关问题；
- 可直接通过内存映射完成用户程序加载，让实验者聚焦“页表操作、特权级切换、写时拷贝”等核心逻辑。

3. 教学场景的需求

教学实验中，用户程序的功能是“验证内核机制”（如 COW 测试、系统调用测试），而非模拟真实应用的动态性：

- 固定虚拟地址避免了动态重定位的复杂度，便于 GDB 调试（可直接通过地址断点跟踪）；
- 一次性嵌入内核让实验者无需关注“文件加载失败、路径错误”等非核心问题，降低实验门槛。

4. 性能并非教学重点

真实 OS 采用动态加载的核心原因是：

- 节省物理内存（仅加载当前运行的程序）；
- 支持程序动态更新（无需重新编译内核）。

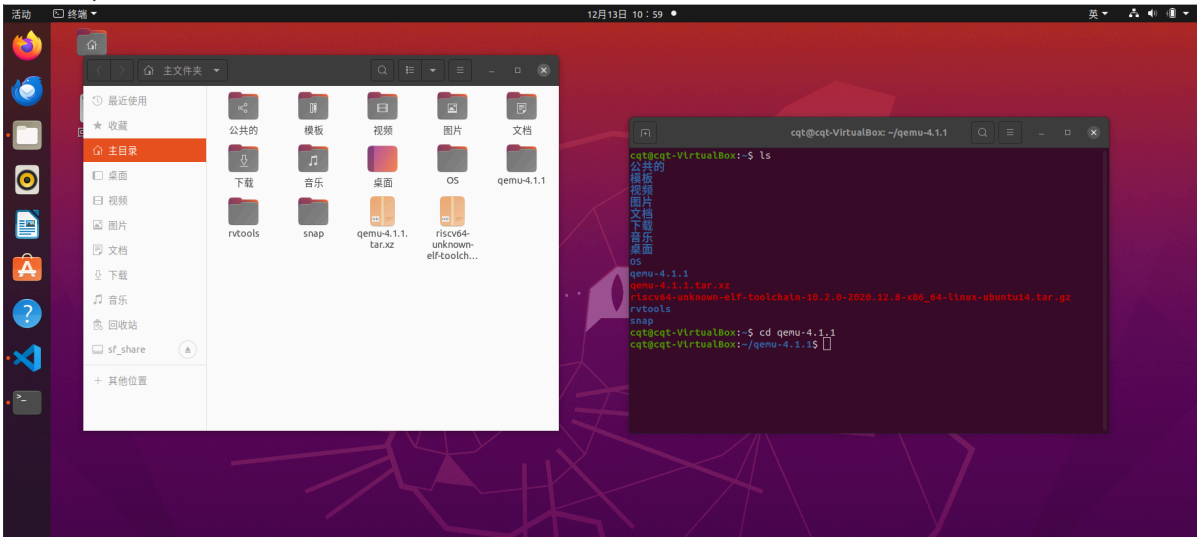
但 ucore 运行在 QEMU 模拟器中，物理内存资源充足，且教学场景无需考虑程序动态更新，因此“预先嵌入”是更优的简化选择。

分支任务：

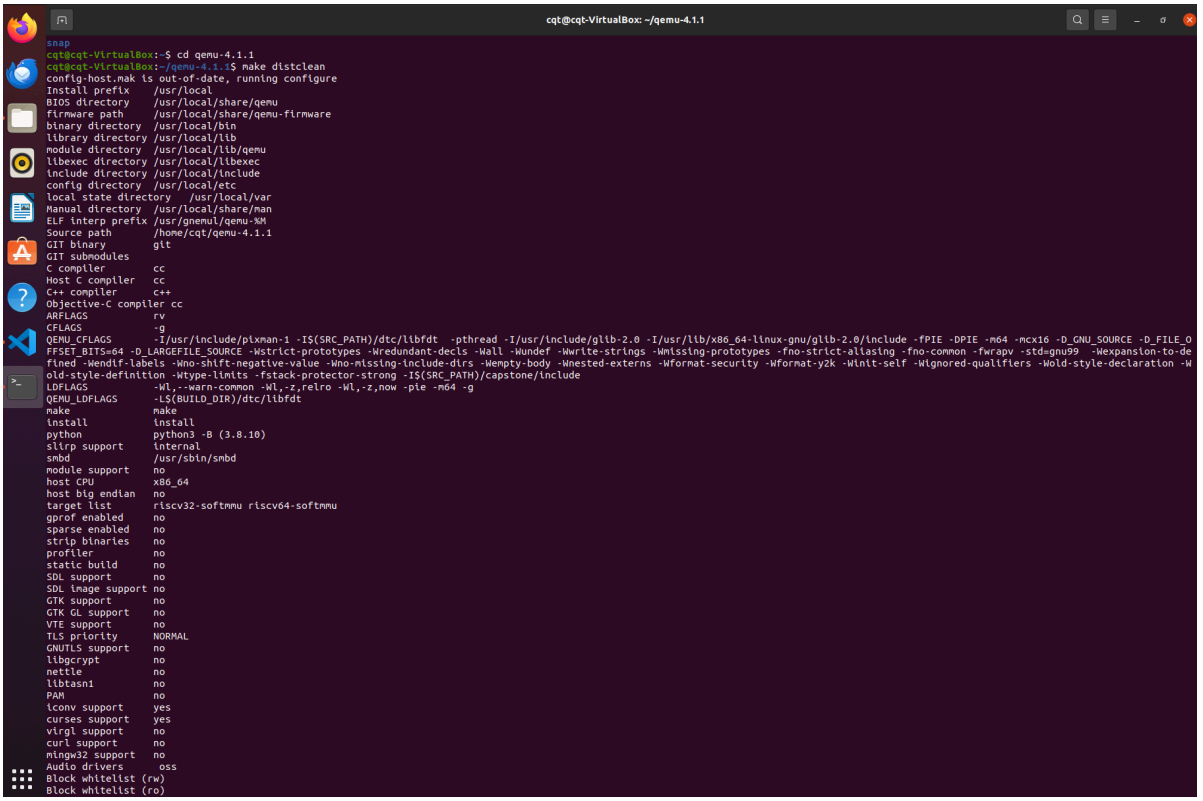
lab2调试：

调试过程

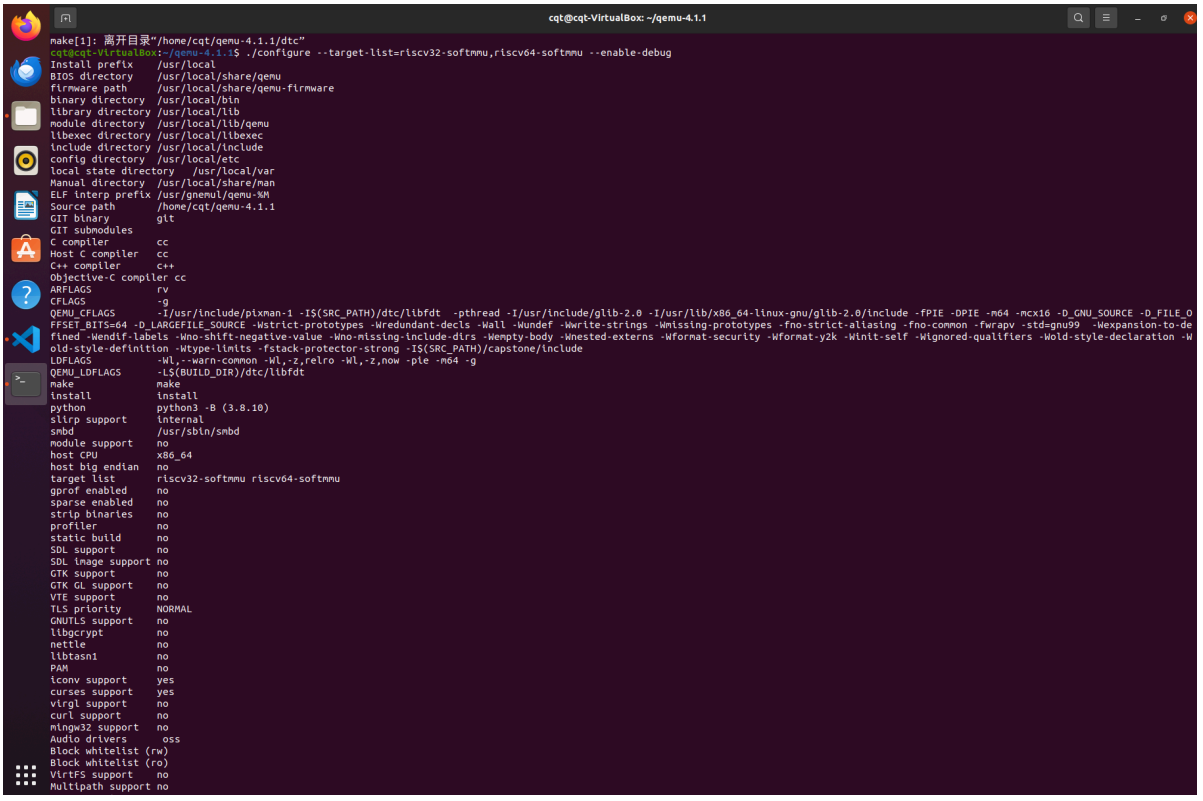
先进入qemu的目录



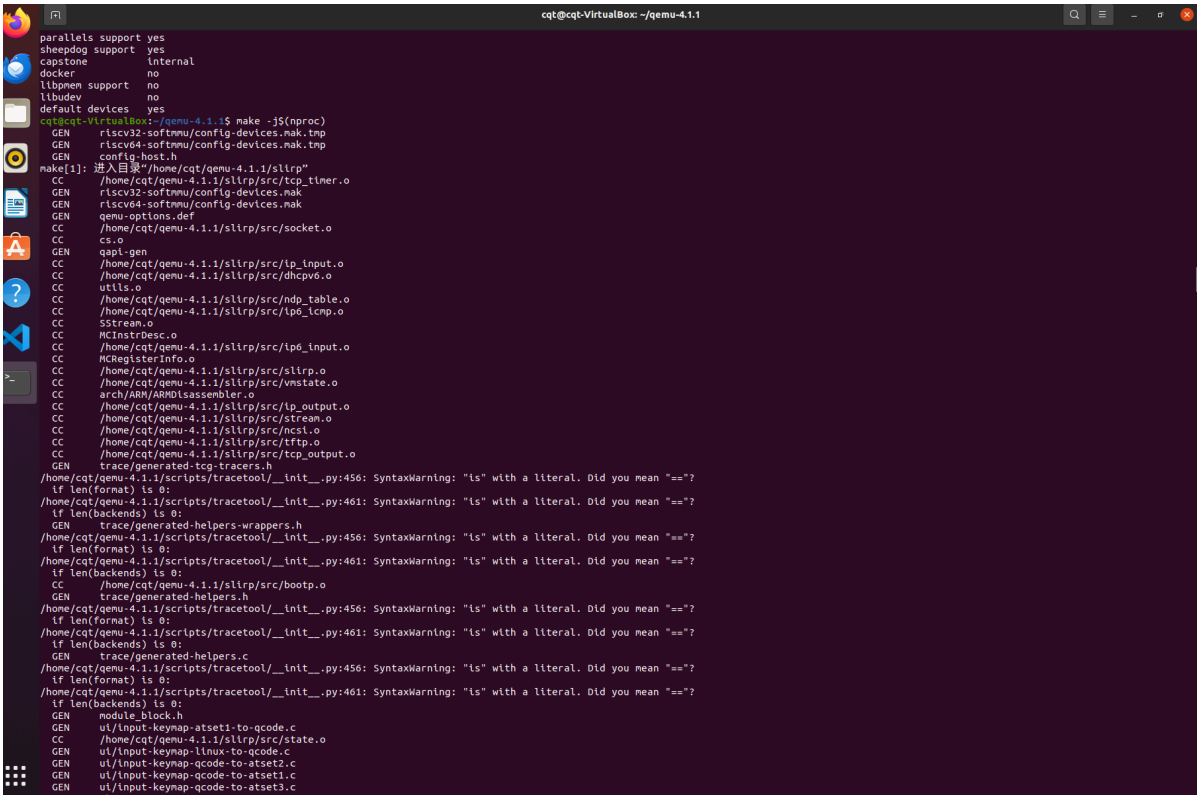
接下来按照实验指导书的指导，重新配置并编译qemu

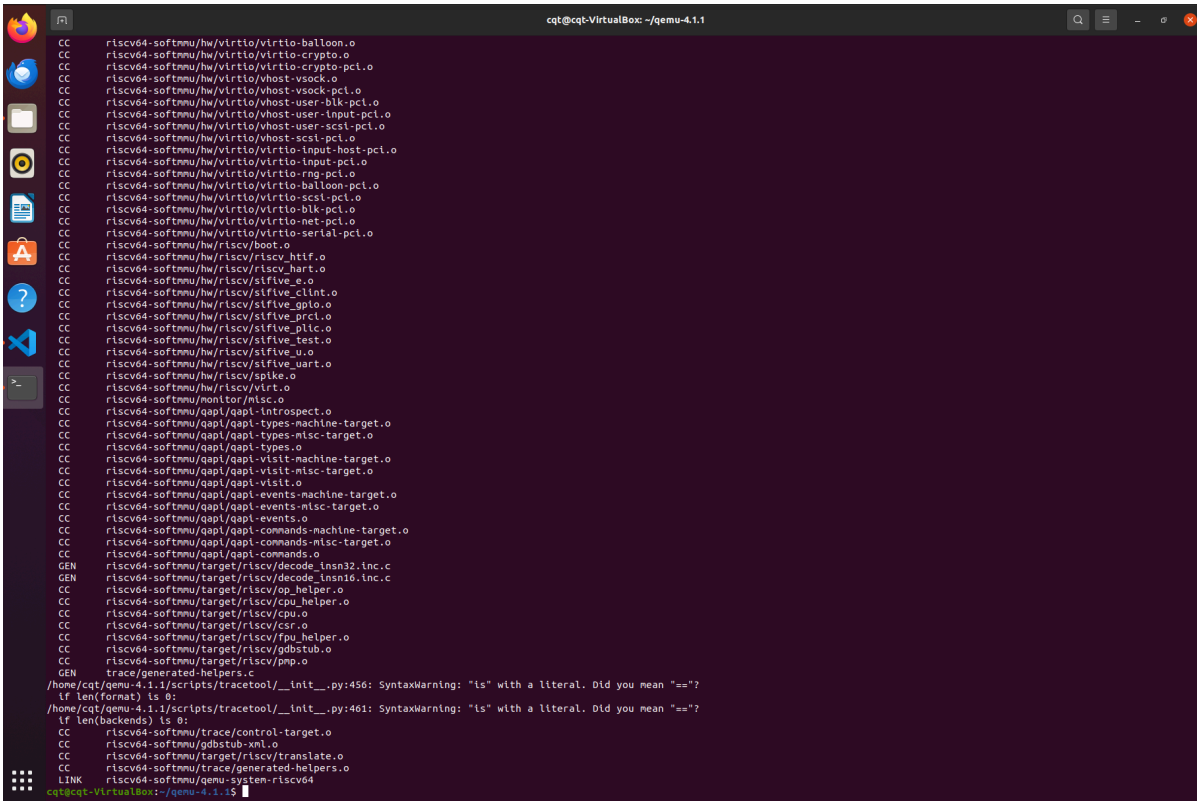


然后重新配置，关键是打开`--enable-debug`：

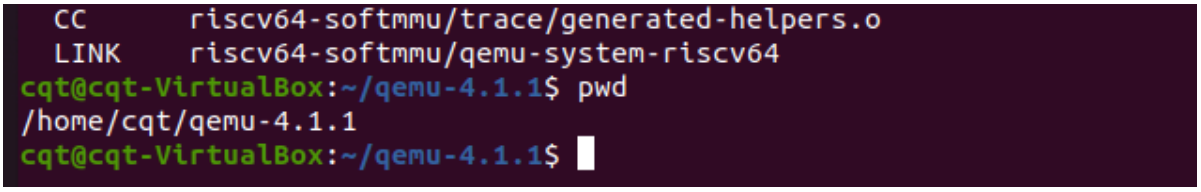


最后编译：





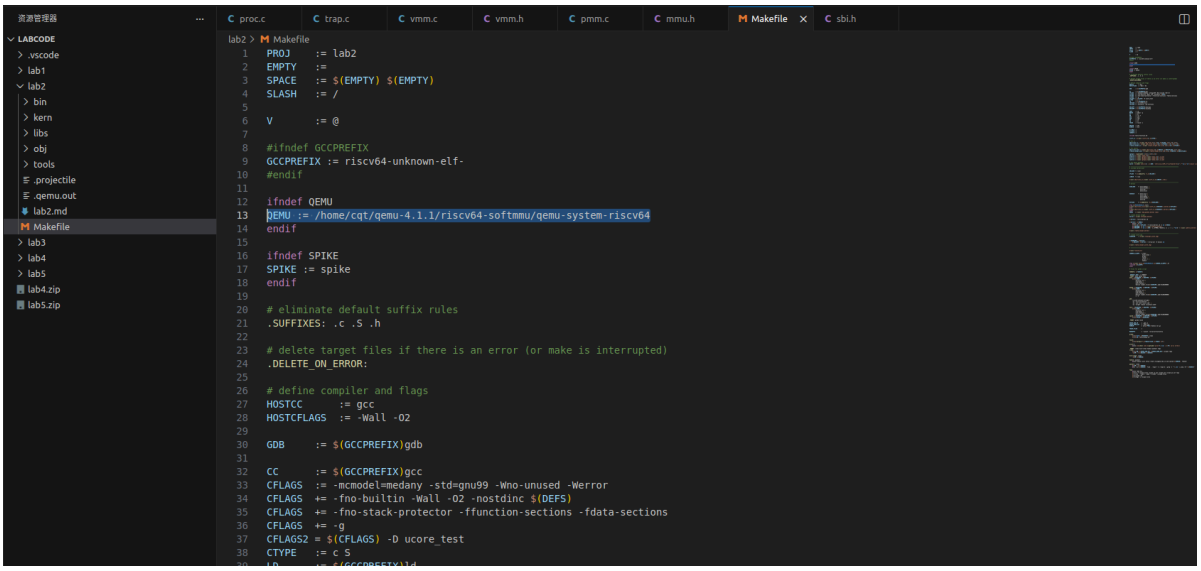
确认一下我们`qemu`的所在位置：



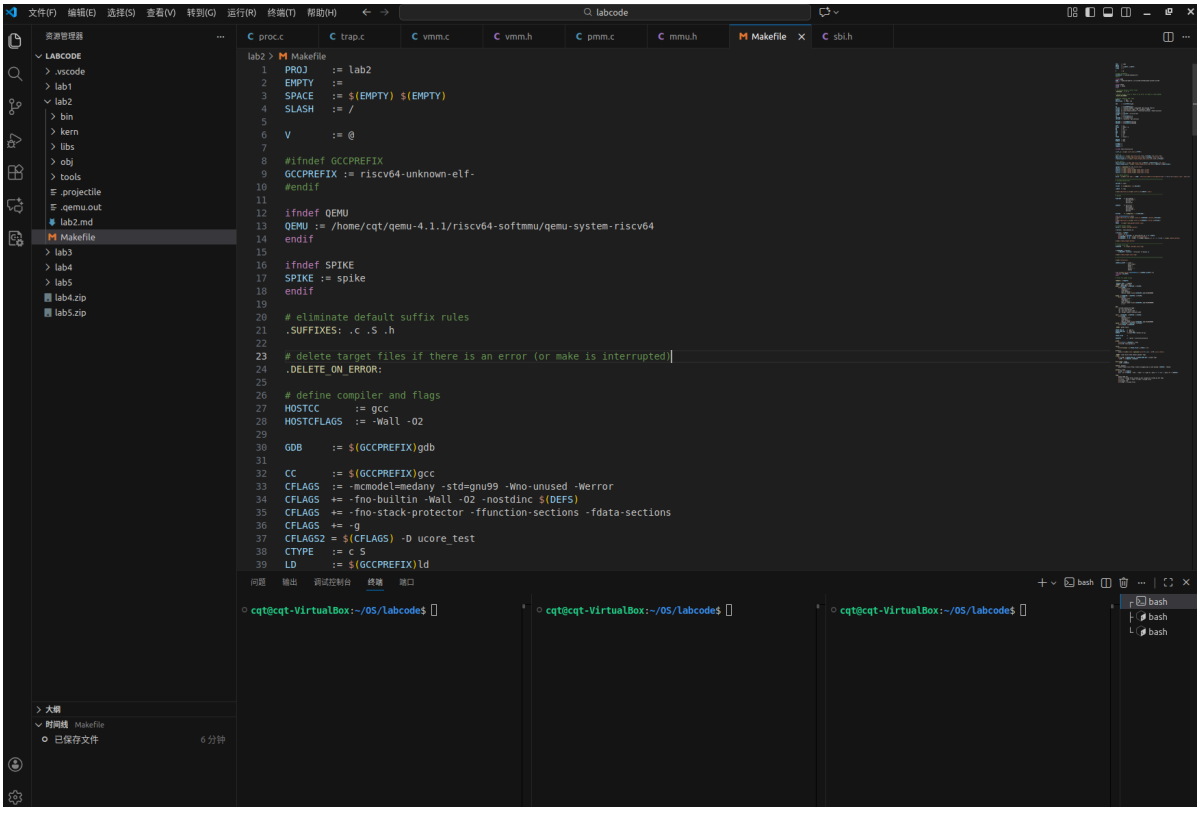
接下来进入`lab2`的目录，开始调试，我们首先需要修改lab2的Makefile，将`qemu`的路径改为刚刚编译出来的文件的真实路径，具体是修改这一段：

```
ifndef QEMU
QEMU := qemu-system-riscv64
endif
```

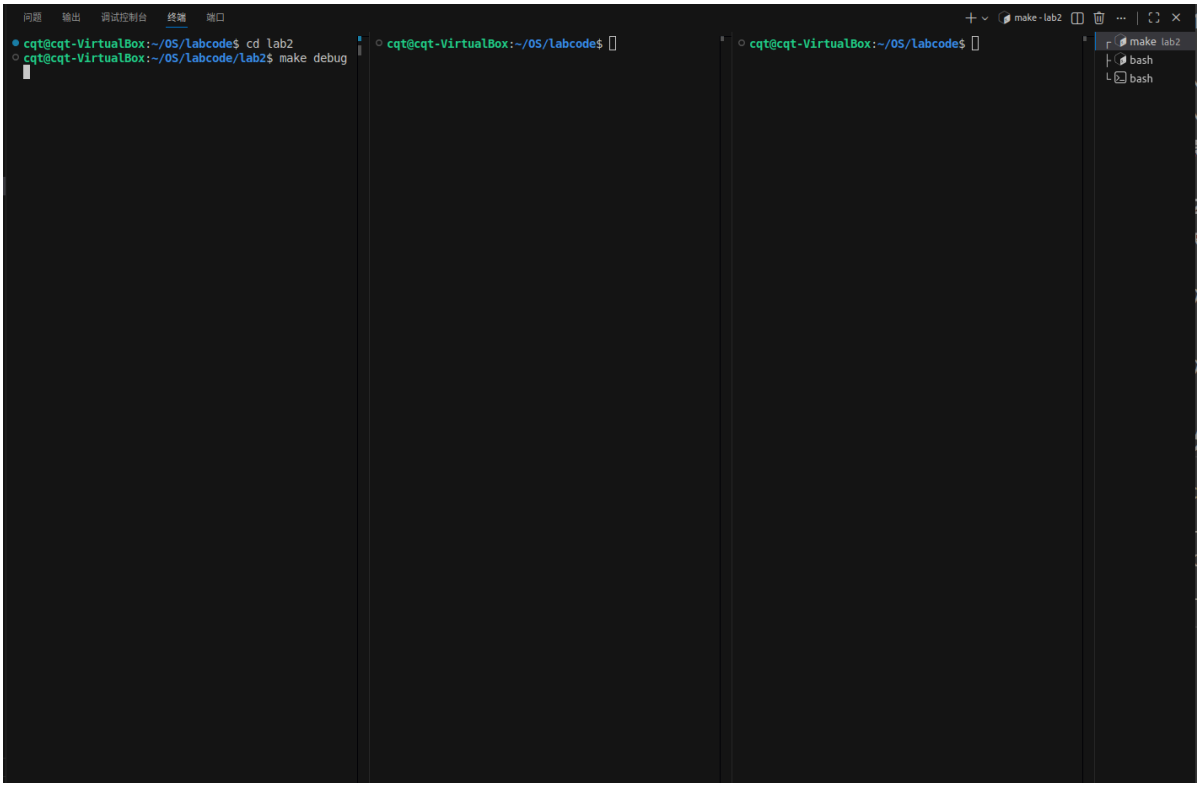
改为图中的样子：



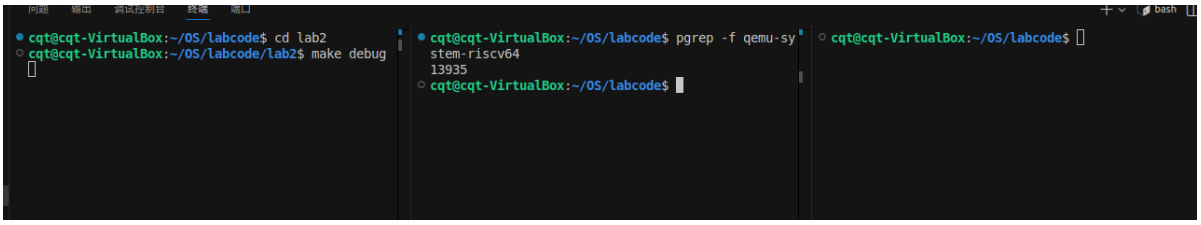
调试需要3个终端窗口：



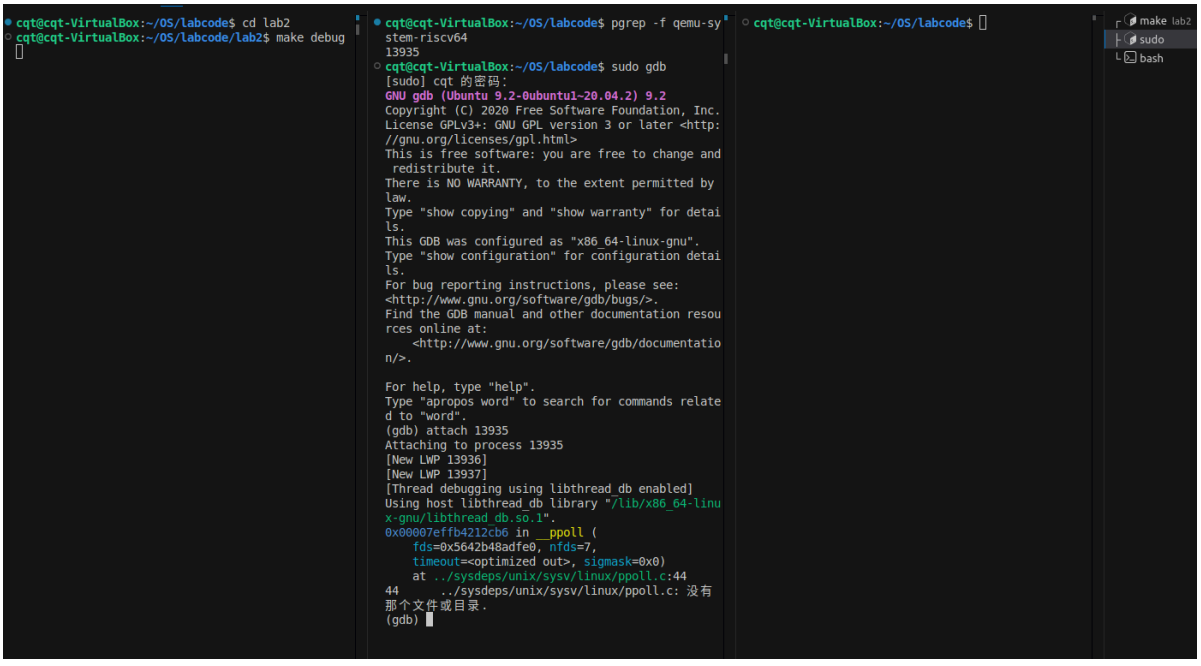
第一个窗口用于启动电源，进入lab2目录，输入`make debug`，屏幕会停住，等调试器连接：



在第二个窗口中，我们输入`pgrep -f qemu-system-riscv64`，找到qemu的进程号：

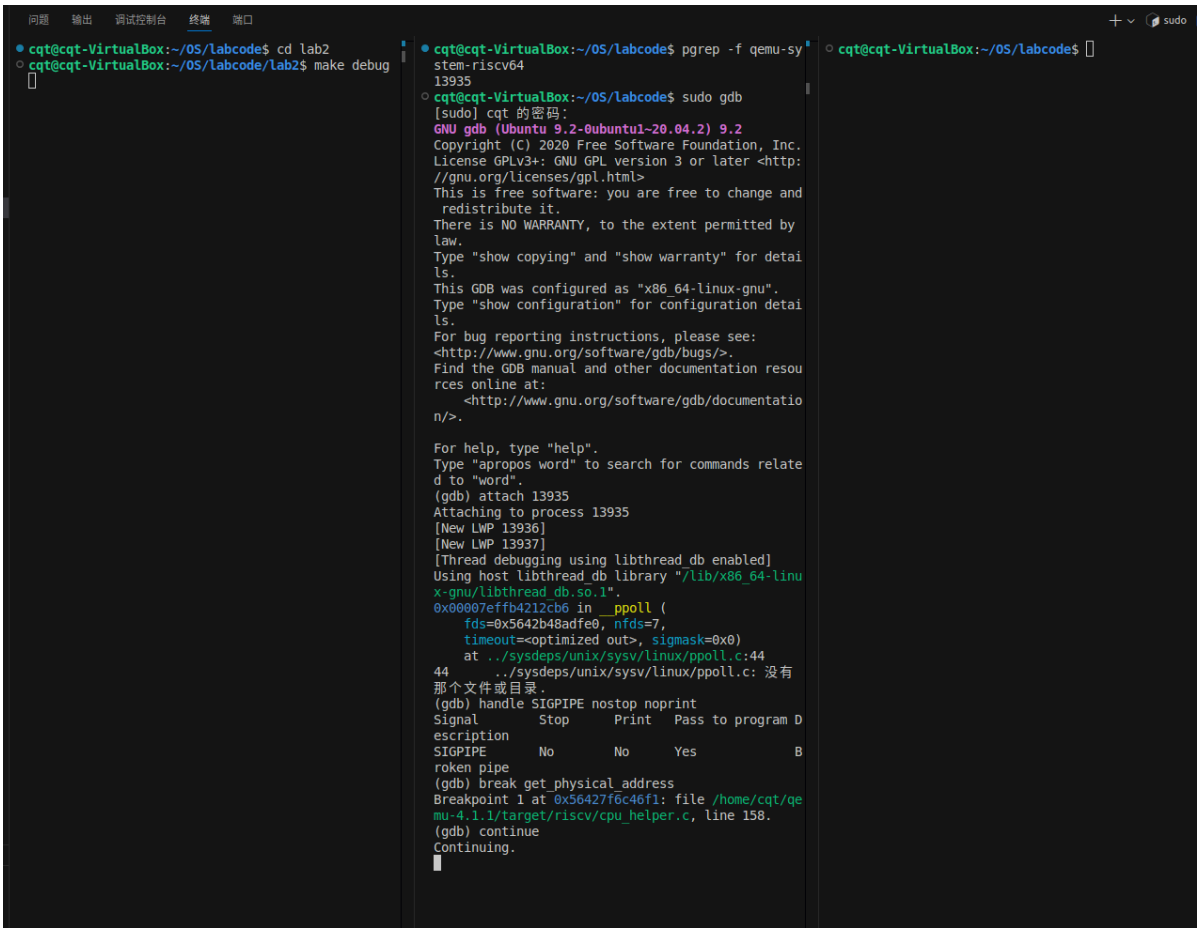


可以看到是`13935`，接下来我们输入`sudo gdb`，进入gdb，然后输入`attach 13935`，将gdb连接到qemu：



接下来输入以下命令，设置断点：

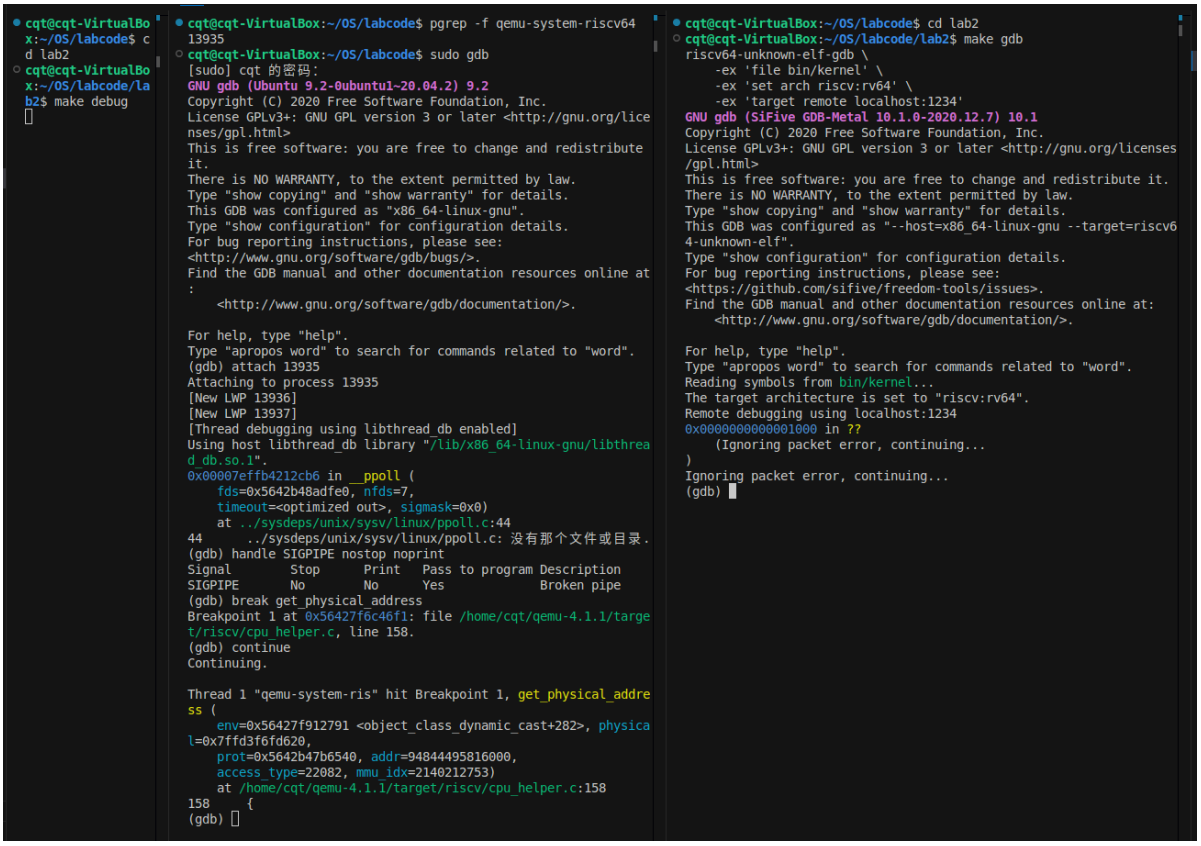
```
handle SIGPIPE nostop noprint
break get_physical_address
continue
```



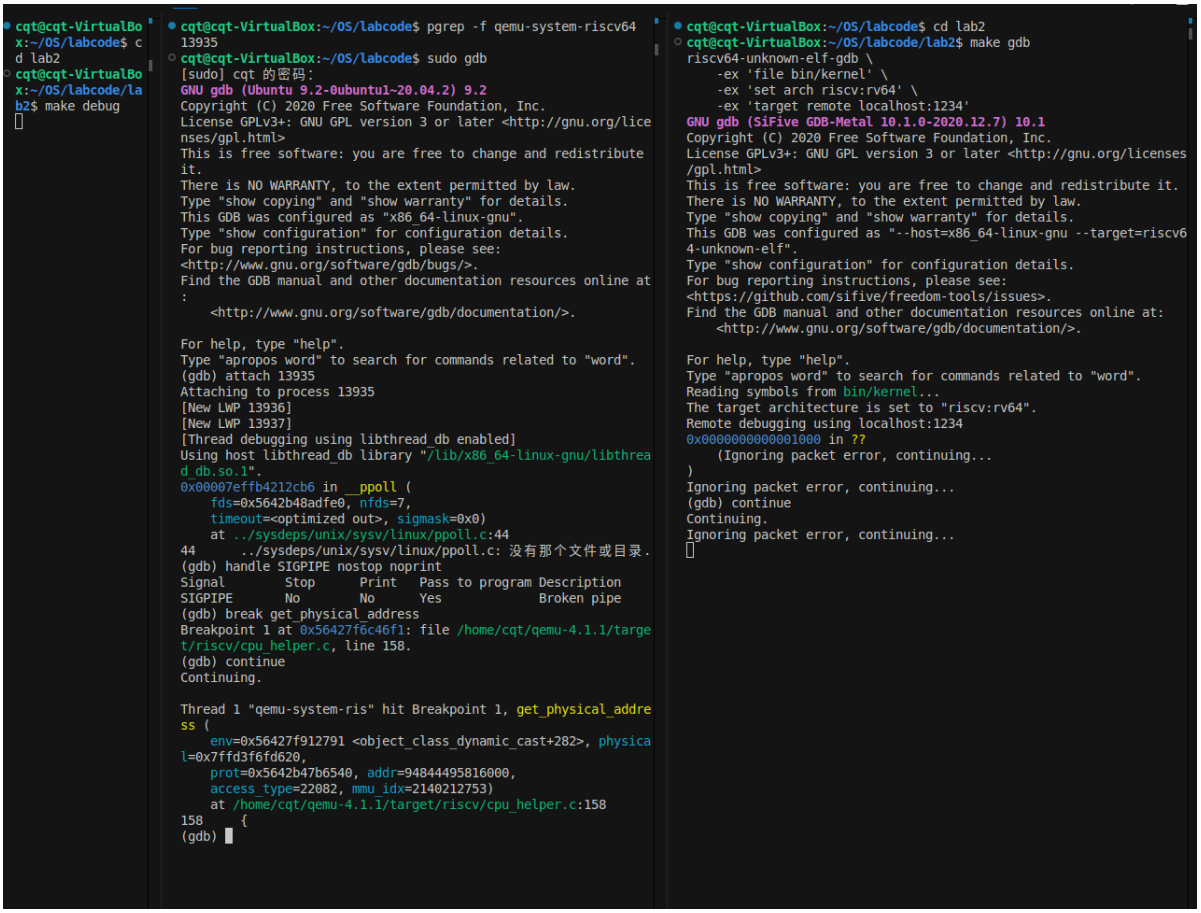
`break get_physical_address` 的意思是：当电脑试图把“虚拟地址”翻译成“物理地址”时，立刻暂停

输入 `continue` 后，它会显示 `Continuing.`，然后不动了。这是正常的，它在埋伏

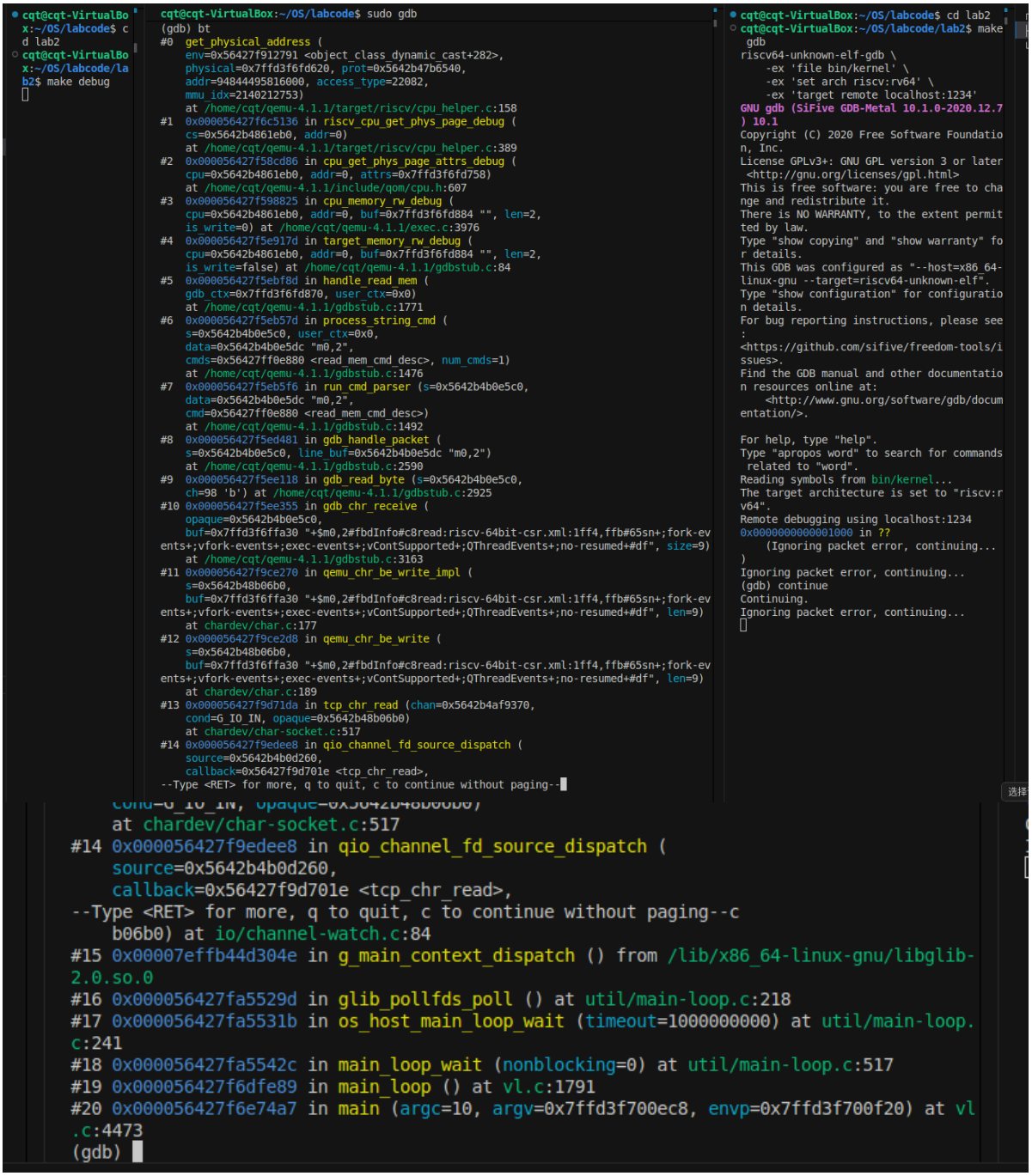
在第三个窗口，我们进入`lab2`的目录，输入`make gdb`，会进入另一个gdb：



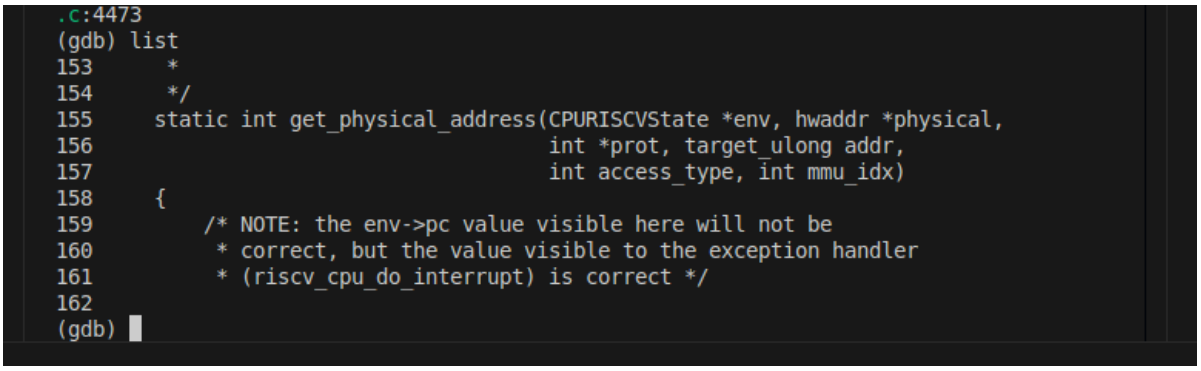
我们输入`continue`，操作系统开始运行，它立刻就会尝试访问内存：



在窗口 2 的 (gdb) 里输入 `bt`，可以看到调用栈：



看看当前代码长啥样：



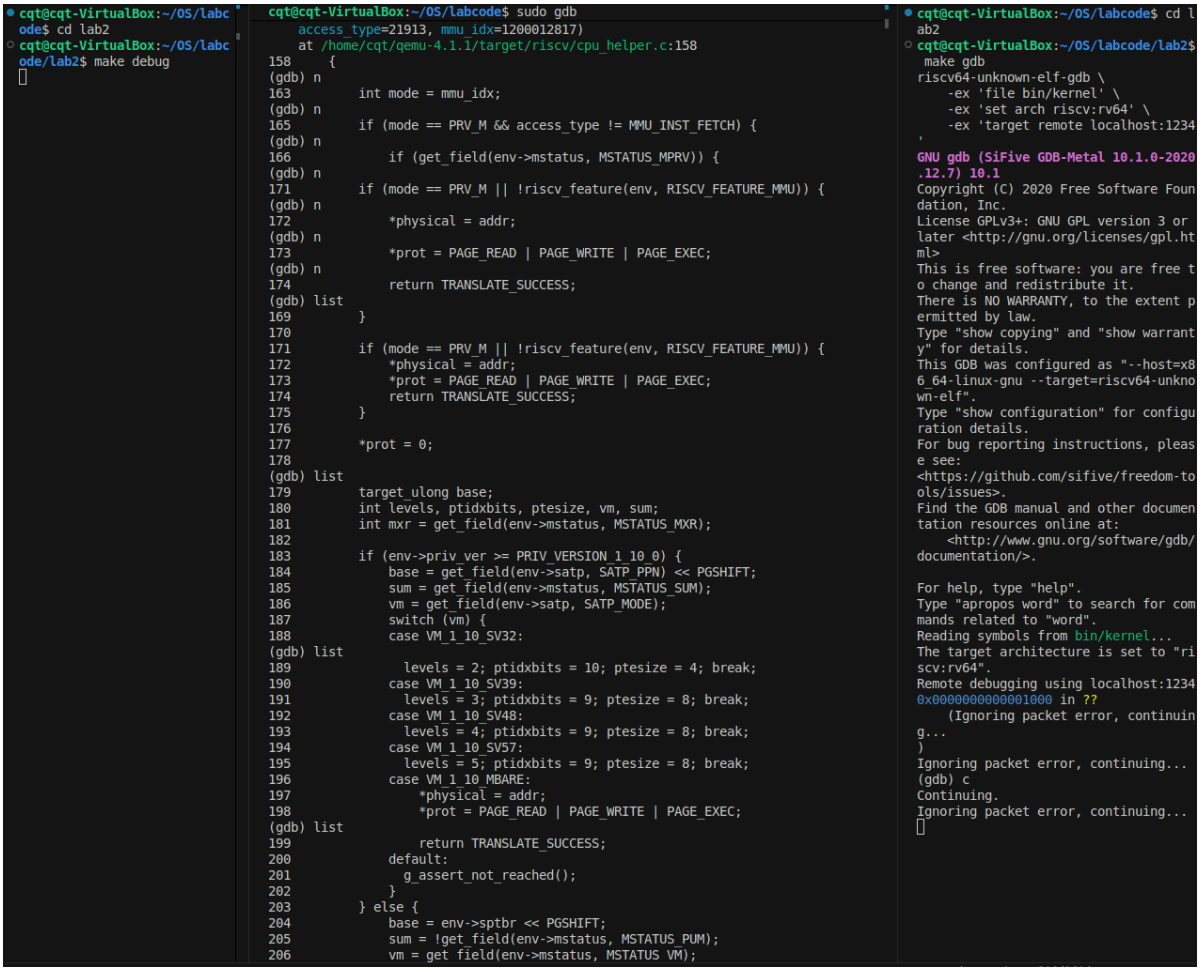
这就是 QEMU 模拟硬件 MMU 的代码。

看看当前正在翻译哪个地址：

```
161           * (riscv_cpu_do_interrupt) is correct */
162
(gdb) print addr
$1 = 94844495816000
(gdb)
```

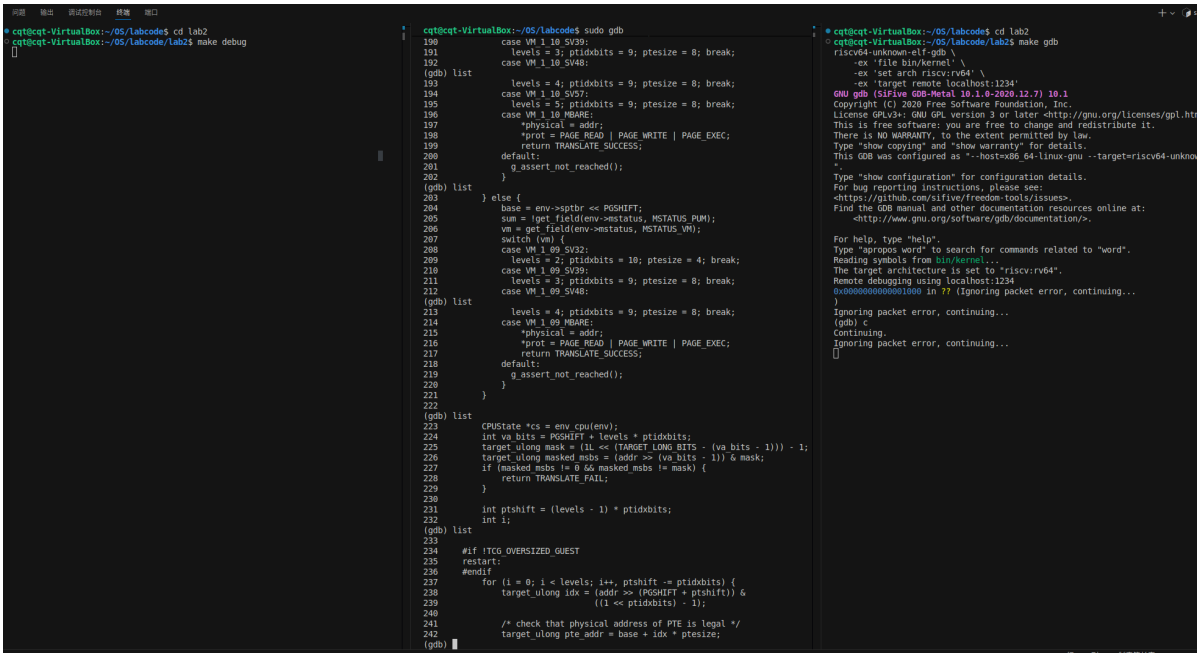
CPU 现在正在访问这个虚拟地址，正准备把它转换成物理地址。

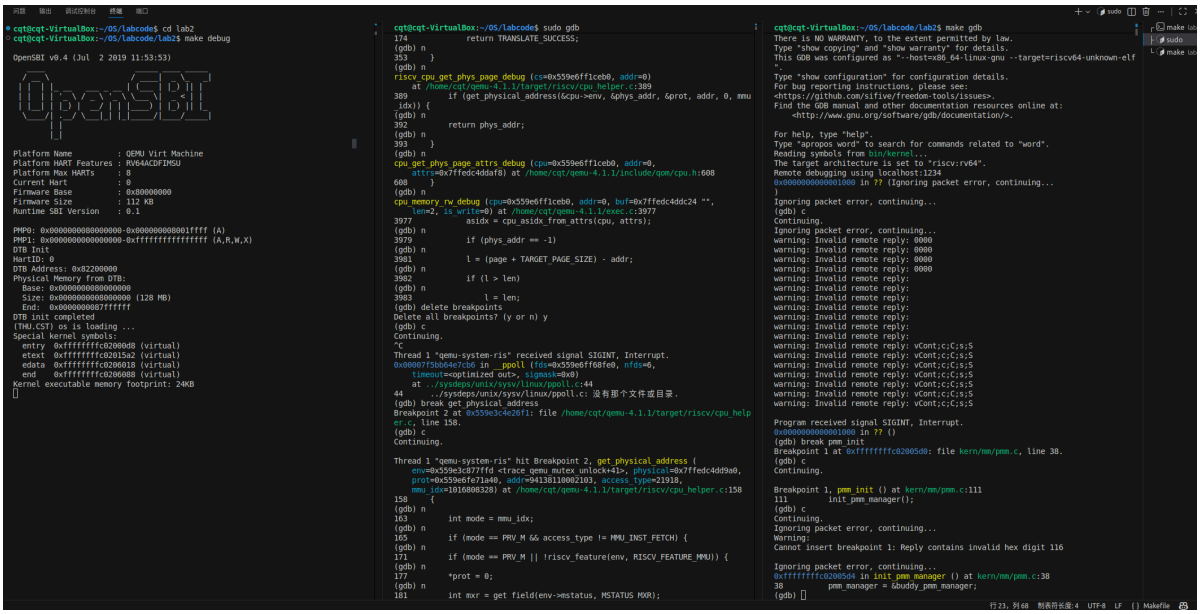
按n/list单步调试，可以看到：



```
200             default:
201               g_assert_not_reached();
202             }
203         } else {
204             base = env->sptbr << PGSHIFT;
205             sum = !get_field(env->mstatus, MSTATUS_PUM);
206             vm = get_field(env->mstatus, MSTATUS_VM);
207             switch (vm) {
208             case VM_1_09_SV32:
(gdb) list
209               levels = 2; ptidxbits = 10; ptesize = 4; break;
210             case VM_1_09_SV39:
211               levels = 3; ptidxbits = 9; ptesize = 8; break;
212             case VM_1_09_SV48:
213               levels = 4; ptidxbits = 9; ptesize = 8; break;
214             case VM_1_09_MBARE:
215                 *physical = addr;
216                 *prot = PAGE_READ | PAGE_WRITE | PAGE_EXEC;
217                 return TRANSLATE_SUCCESS;
218             default:
(gdb) list
219               g_assert_not_reached();
220             }
221         }
222
223         CPUState *cs = env_cpu(env);
224         int va_bits = PGSHIFT + levels * ptidxbits;
225         target_ulong mask = (1L << (TARGET_LONG_BITS - (va_bits - 1))) - 1;
226         target_ulong masked_msbs = (addr >> (va_bits - 1)) & mask;
227         if (masked_msbs != 0 && masked_msbs != mask) {
228             return TRANSLATE_FAIL;
(gdb) list
229         }
230
231         int ptshift = (levels - 1) * ptidxbits;
232         int i;
233
234     #if !TCG_OVERSIZED_GUEST
235     restart:
236     #endif
237         for (i = 0; i < levels; i++, ptshift -= ptidxbits) {
238             target_ulong idx = (addr >> (PGSHIFT + ptshift)) &
(gdb)
```

这里出现了一个for 循环，这个循环就是模拟 SV39 的多级页表查找：





```
cqt@cqt-VirtualBox:~/OS/labcode$ cd lab2
cqt@cqt-VirtualBox:~/OS/labcode/lab2$ make debug

OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x0000000080001fff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000008000000 (128 MB)
  End:  0x0000000087ffffff
DTB init completed
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc02000d8 (virtual)
  etext  0xffffffffc02015a2 (virtual)
  edata  0xffffffffc0206018 (virtual)
  end    0xffffffffc0206088 (virtual)
Kernel executable memory footprint: 24KB
```

```
cqt@cqt-VirtualBox:~/OS/labcode$ sudo gdb
174             return TRANSLATE_SUCCESS;
(gdb) n
353         }
(gdb) n
riscv_cpu_get_phys_page_debug (cs=0x559e6ff1ceb0, addr=0)
    at /home/cqt/qemu-4.1.1/target/riscv/cpu_helper.c:389
389         if (get_physical_address(&cpu->env, &phys_addr, &prot, addr, 0, mmu
 idx)) {
(gdb) n
392             return phys_addr;
(gdb) n
393         }
(gdb) n
cpu_get_phys_page_attrs_debug (cpu=0x559e6ff1ceb0, addr=0,
    attrs=0x7ffedc4ddaf8) at /home/cqt/qemu-4.1.1/include/qom/cpu.h:608
608     }
(gdb) n
cpu_memory_rw_debug (cpu=0x559e6ff1ceb0, addr=0, buf=0x7ffedc4ddc24 "",
    len=2, is_write=0) at /home/cqt/qemu-4.1.1/exec.c:3977
3977            asidx = cpu_asidx_from_attrs(cpu, attrs);
(gdb) n
3979            if (phys_addr == -1)
(gdb) n
3981            l = (page + TARGET_PAGE_SIZE) - addr;
(gdb) n
3982            if (l > len)
(gdb) n
3983                l = len;
(gdb) delete breakpoints
Delete all breakpoints? (y or n) y
(gdb) c
Continuing.
^C
Thread 1 "qemu-system-ris" received signal SIGINT, Interrupt.
0x00007f5bb6e7cb6 in __ppoll (fds=0x559e6ff68fe0, nfds=6,
    timeout=<optimized out>, sigmask=0x0)
    at ../sysdeps/unix/sysv/linux/ppoll.c:44
44      ../sysdeps/unix/sysv/linux/ppoll.c: 没有那个文件或目录.
(gdb) break get_physical_address
Breakpoint 2 at 0x559e3c4e26f1: file /home/cqt/qemu-4.1.1/target/riscv/cpu_help
er.c, line 158.
(gdb) c
Continuing.

Thread 1 "qemu-system-ris" hit Breakpoint 2, get_physical_address (
    env=0x559e3c877ffd <trace_qemu_mutex_unlock+41>, physical=0x7ffedc4dd9a0,
    prot=0x559e6fe71a40, addr=94138110002103, access_type=21918,
    mmu_idx=1016808328) at /home/cqt/qemu-4.1.1/target/riscv/cpu_helper.c:158
158     {
(gdb) n
163         int mode = mmu_idx;
(gdb) n
165         if (mode == PRV_M && access_type != MMU_INST_FETCH) {
(gdb) n
171         if (mode == PRV_M || !riscv_feature(env, RISCV_FEATURE_MMU)) {
(gdb) n
177         *prot = 0;
(gdb) n
181         int mxr = get_field(env->mstatus, MSTATUS_MXR);
```

```
cqt@cqt-VirtualBox:~/OS/labcode/lab2$ make gdb
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 ?? (Ignoring packet error, continuing...
)
Ignoring packet error, continuing...
(gdb) c
Continuing.
Ignoring packet error, continuing...
warning: Invalid remote reply: 0000
warning: Invalid remote reply: 0000
warning: Invalid remote reply: 0000
warning: Invalid remote reply: 0000
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S

Program received signal SIGINT, Interrupt.
0x0000000000001000 in ?? ()
(gdb) break pmm_init
Breakpoint 1 at 0xffffffffc02005d0: file kern/mm/pmm.c, line 38.
(gdb) c
Continuing.

Breakpoint 1, pmm_init () at kern/mm/pmm.c:111
111         init_pmm_manager();
(gdb) c
Continuing.
Ignoring packet error, continuing...
Warning:
Cannot insert breakpoint 1: Reply contains invalid hex digit 116

Ignoring packet error, continuing...
0xffffffffc02005d4 in init_pmm_manager () at kern/mm/pmm.c:38
38          pmm_manager = &buddy_pmm_manager;
(gdb)
```


```
cqt@cqt-VirtualBox:~/OS/labcode$ cd lab2
cqt@cqt-VirtualBox:~/OS/labcode/lab2$ make debug

OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x0000000080001fff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000008000000 (128 MB)
  End:  0x0000000087ffffff
DTB init completed
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc02000d8 (virtual)
  etext  0xffffffffc02015a2 (virtual)
  edata  0xffffffffc0206018 (virtual)
  end    0xffffffffc0206088 (virtual)
Kernel executable memory footprint: 24KB
```

```
cqt@cqt-VirtualBox:~/OS/labcode$ sudo gdb
(gdb) c
Continuing.

Thread 1 "qemu-system-ris" hit Breakpoint 2, get_physical_address (
    env=0x559e3c877ffd <trace_qemu_mutex_unlock+41>, physical=0x7ffedc4dd9a0,
    prot=0x559e6fe71a40, addr=94138110002103, access_type=21918,
    mmu_idx=1016808328) at /home/cqt/qemu-4.1.1/target/riscv/cpu_helper.c:158
158     {
(gdb) n
163         int mode = mmu_idx;
(gdb) n
165         if (mode == PRV_M && access_type != MMU_INST_FETCH) {
(gdb) n
171         if (mode == PRV_M || !riscv_feature(env, RISCV_FEATURE_MMU)) {
(gdb) n
177         *prot = 0;
(gdb) n
181         int mxr = get_field(env->mstatus, MSTATUS_MXR);
(gdb) n
183         if (env->priv_ver >= PRIV_VERSION_1_10_0) {
(gdb) n
184             base = get_field(env->satp, SATP_PPN) << PGSHIFT;
(gdb) n
185             sum = get_field(env->mstatus, MSTATUS_SUM);
(gdb) n
186             vm = get_field(env->satp, SATP_MODE);
(gdb) n
191               levels = 3; ptidxbits = 9; ptesize = 8; break;
(gdb) n
223         CPUState *cs = env_cpu(env);
(gdb) n
224         int va_bits = PGSHIFT + levels * ptidxbits;
(gdb) n
225         target_ulong mask = (1L << (TARGET_LONG_BITS - (va_bits - 1))) - 1;
(gdb) n
226         target_ulong masked_msbs = (addr >> (va_bits - 1)) & mask;
(gdb) n
227         if (masked_msbs != 0 && masked_msbs != mask) {
(gdb) n
231         int ptshift = (levels - 1) * ptidxbits;
(gdb) n
237         for (i = 0; i < levels; i++, ptshift -= ptidxbits) {
(gdb) n
238             target_ulong idx = (addr >> (PGSHIFT + ptshift)) &
(gdb) n
239                                ((1 << ptidxbits) - 1);
(gdb) n
238             target_ulong idx = (addr >> (PGSHIFT + ptshift)) &
(gdb) n
242             target_ulong pte_addr = base + idx * ptesize;
(gdb) n
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)
```

```
cqt@cqt-VirtualBox:~/OS/labcode/lab2$ make gdb
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 ?? (Ignoring packet error, continuing...
)
Ignoring packet error, continuing...
(gdb) c
Continuing.
Ignoring packet error, continuing...
warning: Invalid remote reply: 0000
warning: Invalid remote reply: 0000
warning: Invalid remote reply: 0000
warning: Invalid remote reply: 0000
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S

Program received signal SIGINT, Interrupt.
0x0000000000001000 in ?? ()
(gdb) break pmm_init
Breakpoint 1 at 0xffffffffc02005d0: file kern/mm/pmm.c, line 38.
(gdb) c
Continuing.

Breakpoint 1, pmm_init () at kern/mm/pmm.c:111
111         init_pmm_manager();
(gdb) c
Continuing.
Ignoring packet error, continuing...
Warning:
Cannot insert breakpoint 1: Reply contains invalid hex digit 116

Ignoring packet error, continuing...
0xffffffffc02005d4 in init_pmm_manager () at kern/mm/pmm.c:38
38          pmm_manager = &buddy_pmm_manager;
(gdb)
```



```
cqt@cqt-VirtualBox:~/OS/labcode$ cd lab2
cqt@cqt-VirtualBox:~/OS/labcode/lab2$ make debug

OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x0000000080001fff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000008000000 (128 MB)
  End:  0x0000000087ffffff
DTB init completed
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc02000d8 (virtual)
  etext  0xffffffffc02015a2 (virtual)
  edata  0xffffffffc0206018 (virtual)
  end    0xffffffffc0206088 (virtual)
Kernel executable memory footprint: 24KB
```

```
cqt@cqt-VirtualBox:~/OS/labcode$ sudo gdb

Thread 1 "qemu-system-ris" hit Breakpoint 2, get_physical_address (
    env=0x559e3c877ffd <trace_qemu_mutex_unlock+41>, physical=0x7ffedc4dd9a0,
    prot=0x559e6fe71a40, addr=94138110002103, access_type=21918,
    mmu_idx=1016808328) at /home/cqt/qemu-4.1.1/target/riscv/cpu_helper.c:158
158     {
(gdb) n
163         int mode = mmu_idx;
(gdb) n
165         if (mode == PRV_M && access_type != MMU_INST_FETCH) {
(gdb) n
171         if (mode == PRV_M || !riscv_feature(env, RISCV_FEATURE_MMU)) {
(gdb) n
177         *prot = 0;
(gdb) n
181         int mxr = get_field(env->mstatus, MSTATUS_MXR);
(gdb) n
183         if (env->priv_ver >= PRIV_VERSION_1_10_0) {
(gdb) n
184             base = get_field(env->satp, SATP_PPN) << PGSHIFT;
(gdb) n
185             sum = get_field(env->mstatus, MSTATUS_SUM);
(gdb) n
186             vm = get_field(env->satp, SATP_MODE);
(gdb) n
191               levels = 3; ptidxbits = 9; ptesize = 8; break;
(gdb) n
223         CPUState *cs = env_cpu(env);
(gdb) n
224         int va_bits = PGSHIFT + levels * ptidxbits;
(gdb) n
225         target_ulong mask = (1L << (TARGET_LONG_BITS - (va_bits - 1))) - 1;
(gdb) n
226         target_ulong masked_msbs = (addr >> (va_bits - 1)) & mask;
(gdb) n
227         if (masked_msbs != 0 && masked_msbs != mask) {
(gdb) n
231         int ptshift = (levels - 1) * ptidxbits;
(gdb) n
237         for (i = 0; i < levels; i++, ptshift -= ptidxbits) {
(gdb) n
238             target_ulong idx = (addr >> (PGSHIFT + ptshift)) &
(gdb) n
239                                ((1 << ptidxbits) - 1);
(gdb) n
238             target_ulong idx = (addr >> (PGSHIFT + ptshift)) &
(gdb) n
242             target_ulong pte_addr = base + idx * ptesize;
(gdb) n
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) print /x pte
$1 = 0x559e3c9b4388
(gdb)
```

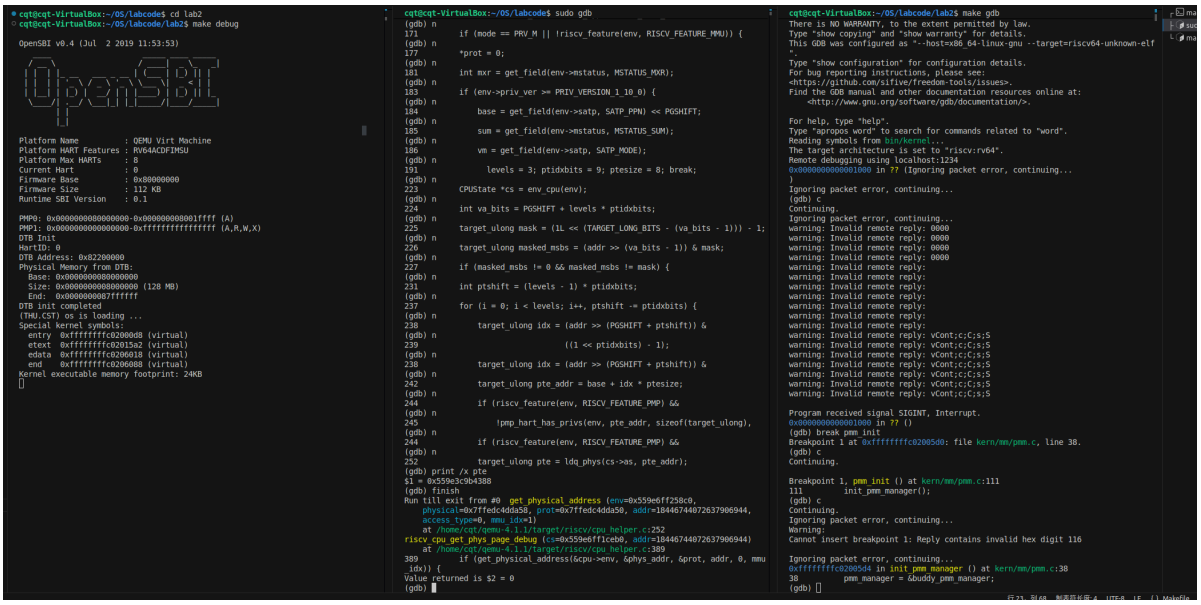
```
cqt@cqt-VirtualBox:~/OS/labcode/lab2$ make gdb
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-linux-gnu --target=riscv64-unknown-elf
".
Type "show configuration" for configuration details.
For bug 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 ?? (Ignoring packet error, continuing...
)
Ignoring packet error, continuing...
(gdb) c
Continuing.
Ignoring packet error, continuing...
warning: Invalid remote reply: 0000
warning: Invalid remote reply: 0000
warning: Invalid remote reply: 0000
warning: Invalid remote reply: 0000
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply:
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S
warning: Invalid remote reply: vCont;c;C;s;S

Program received signal SIGINT, Interrupt.
0x0000000000001000 in ?? ()
(gdb) break pmm_init
Breakpoint 1 at 0xffffffffc02005d0: file kern/mm/pmm.c, line 38.
(gdb) c
Continuing.

Breakpoint 1, pmm_init () at kern/mm/pmm.c:111
111         init_pmm_manager();
(gdb) c
Continuing.
Ignoring packet error, continuing...
Warning:
Cannot insert breakpoint 1: Reply contains invalid hex digit 116

Ignoring packet error, continuing...
0xffffffffc02005d4 in init_pmm_manager () at kern/mm/pmm.c:38
38          pmm_manager = &buddy_pmm_manager;
(gdb)
```



深入分析：页表翻译与 TLB 填充

通过单步调试（图 11-17），我成功观察到了 QEMU 模拟 RISC-V SV39 页表查找的完整过程：

1. **进入页表查找循环**（图 13）：
   代码进入了 `for (i = 0; i < levels; ...)` 循环。这里的 `levels` 为 3，对应 SV39 架构的三级页表机制。这个循环模拟了硬件逐级查询页表的过程。
2. **模拟硬件读取页表项 (PTE)**（图 14-16）：
   在循环内部，QEMU 使用 `ldq_phys` 函数（Load Quadword Physical）从计算出的物理地址 `pte_addr` 中读取 64 位的页表项 `pte`。这行代码精确模拟了硬件 MMU 访问物理内存的行为。
3. **计算下一级地址**（图 17）：
   读取到 PTE 后，代码检查其有效位（V位）。如果有效且不是叶子节点，代码通过 `base = ppn << PGSHIFT` 提取出物理页号（PPN），作为下一级页表的基地址，准备进行下一次循环查找。
4. **TLB 填充逻辑**（补充说明）：
   当 `get_physical_address` 函数执行完毕并返回物理地址后，控制权会回到 `riscv_cpu_tlb_fill` 函数。该函数随后会调用 `tlb_set_page`，将刚刚查找到的【虚拟页号 -> 物理页帧号】映射关系填入 QEMU 的软件 TLB 中。这样，后续对同一页面的访问将直接命中 TLB，无需再次进行耗时的页表遍历。

实验结论

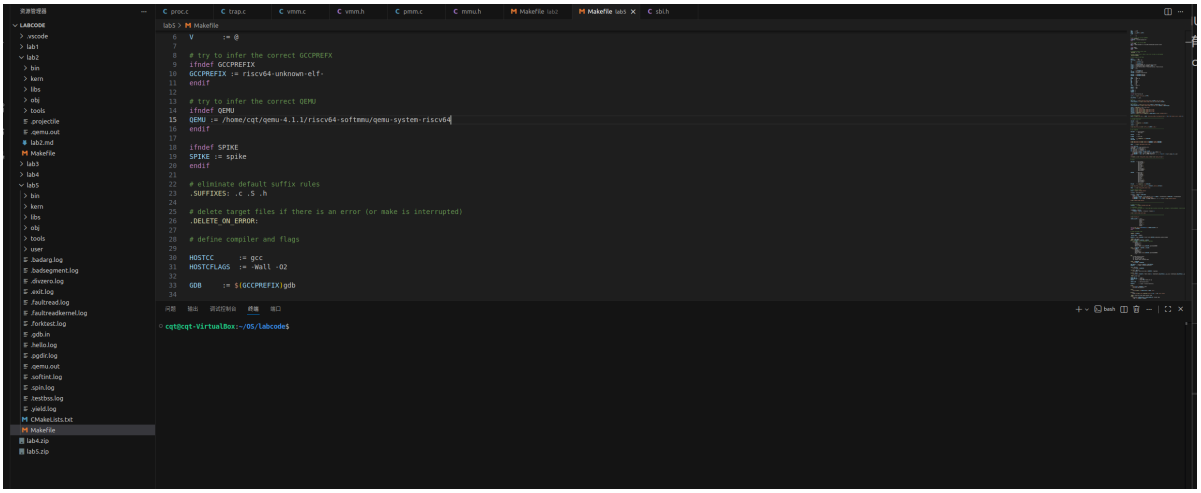
通过本次“双重 GDB”调试实验，我深入理解了软硬件协同工作的原理：

- **ucore 层面**：操作系统负责维护页表的内容（建立映射）。
- **QEMU 层面**：模拟器通过 C 代码（`get_physical_address`）模拟了硬件 MMU 的行为，包括 TLB 查找失败后的硬件页表遍历（Page Table Walk）。
- **差异点**：真实硬件的 TLB 是高速电路，而 QEMU 的 TLB 是为了加速模拟而设计的软件缓存（映射到宿主机虚拟地址），两者在实现机制上不同，但达到的加速目的是一致的。

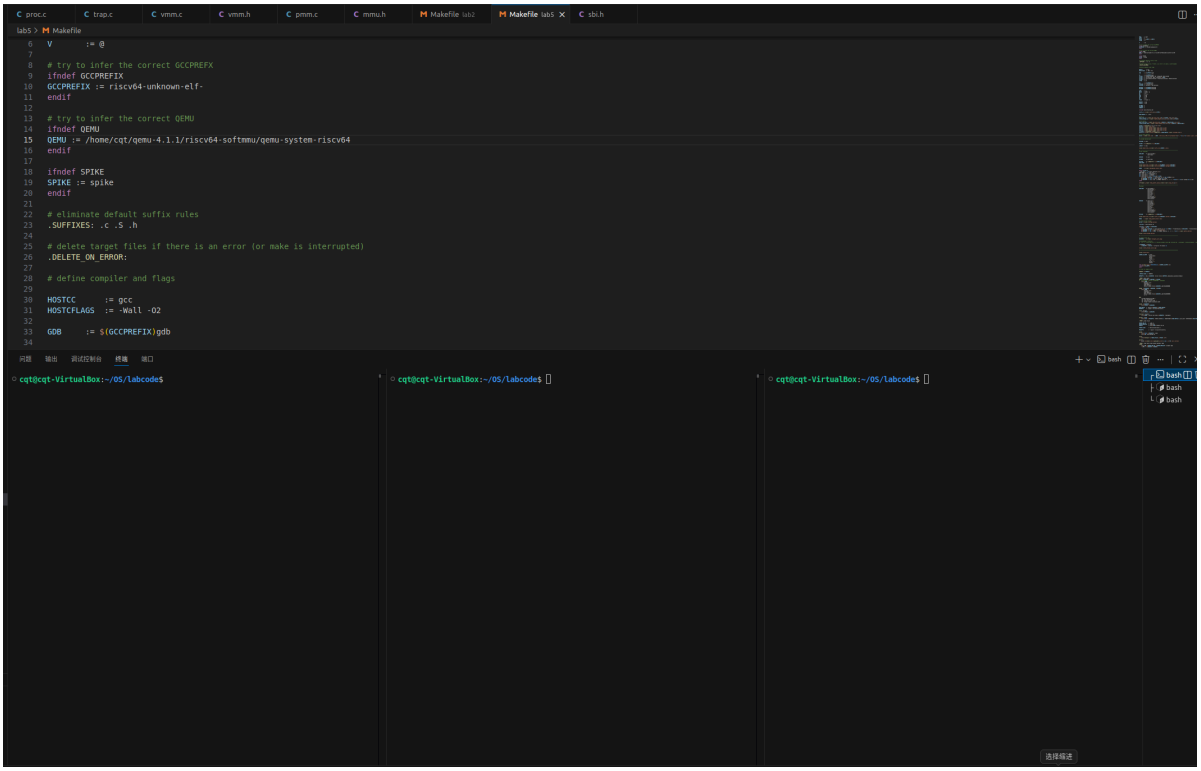
lab5调试：

调试过程

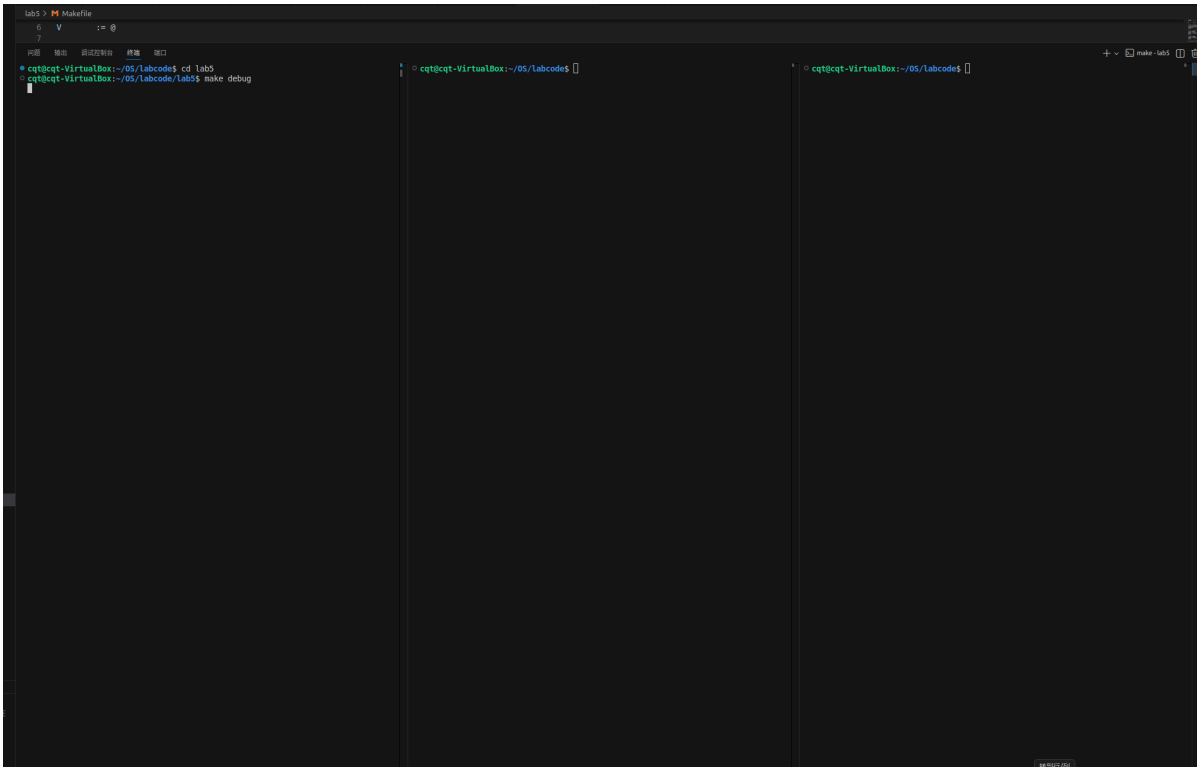
一样的，先修改makefile里qemu文件的路径：



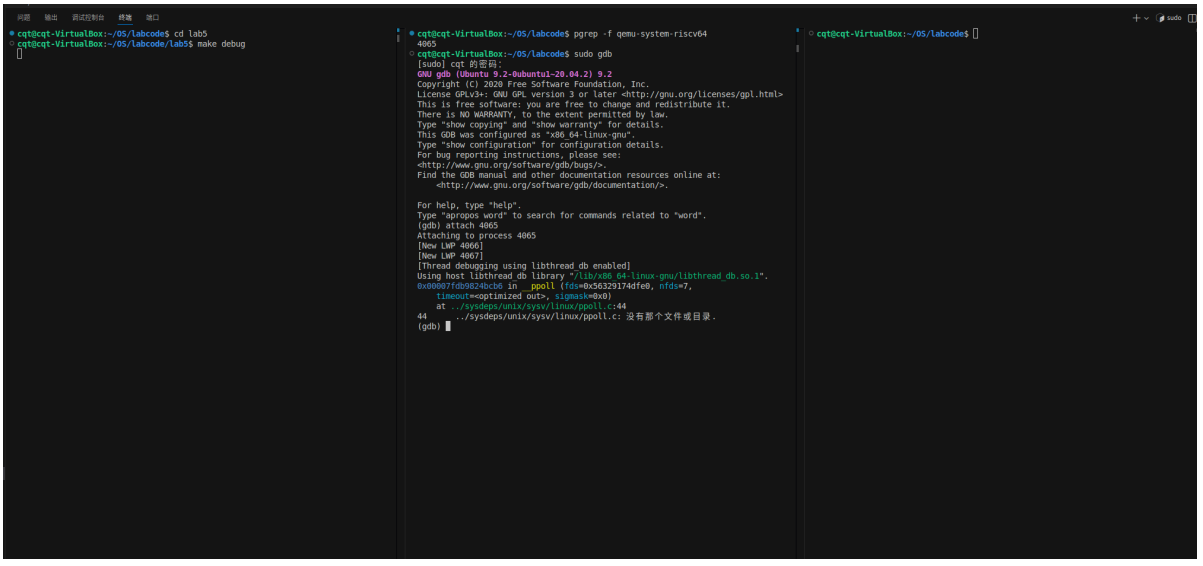
再次打开三个窗口：



窗口一依然是进入lab5，然后运行make debug：

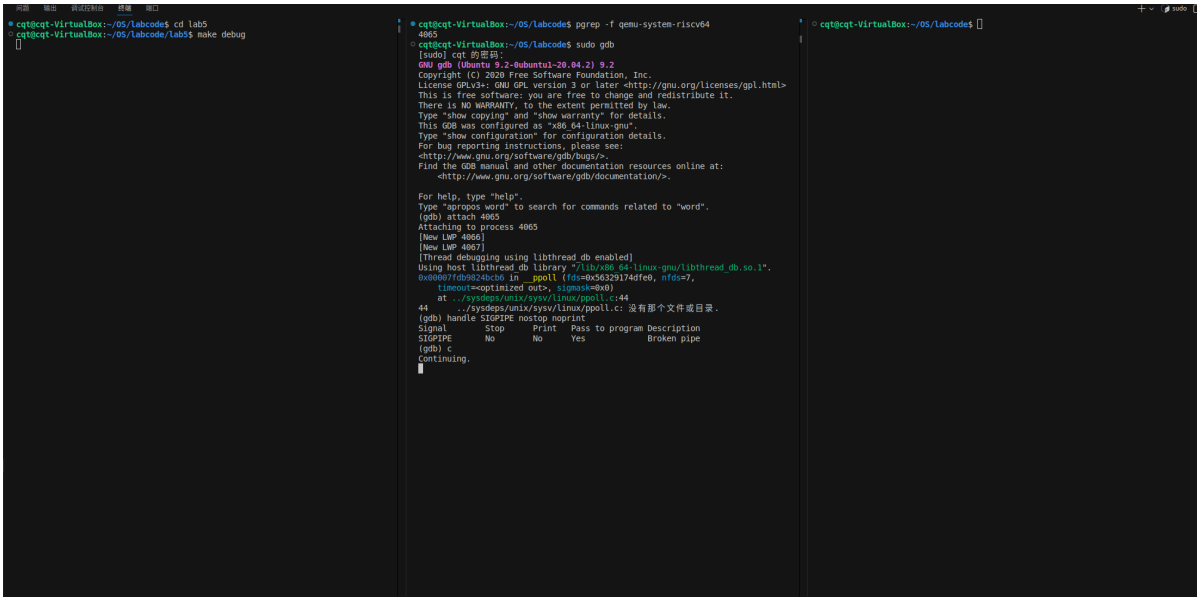


窗口二准备用来调试qemu，和之前的方法一样：

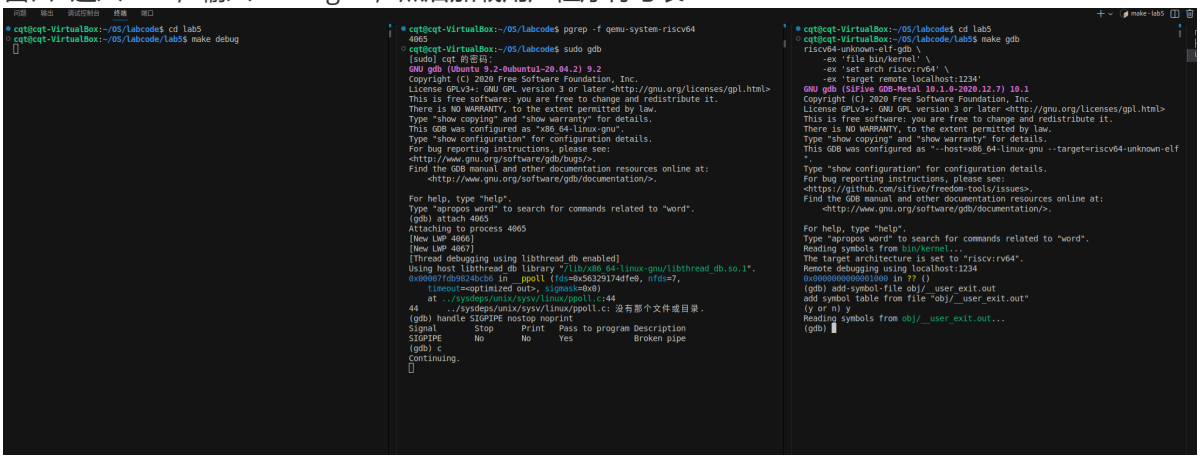


输入handle SIGPIPE nostop noprint。这次我们要抓的是 ecall 指令。在 QEMU 源码里，处理 ecall 的函数通常叫 helper_raise_exception 或者跟 CSR 处理有关。

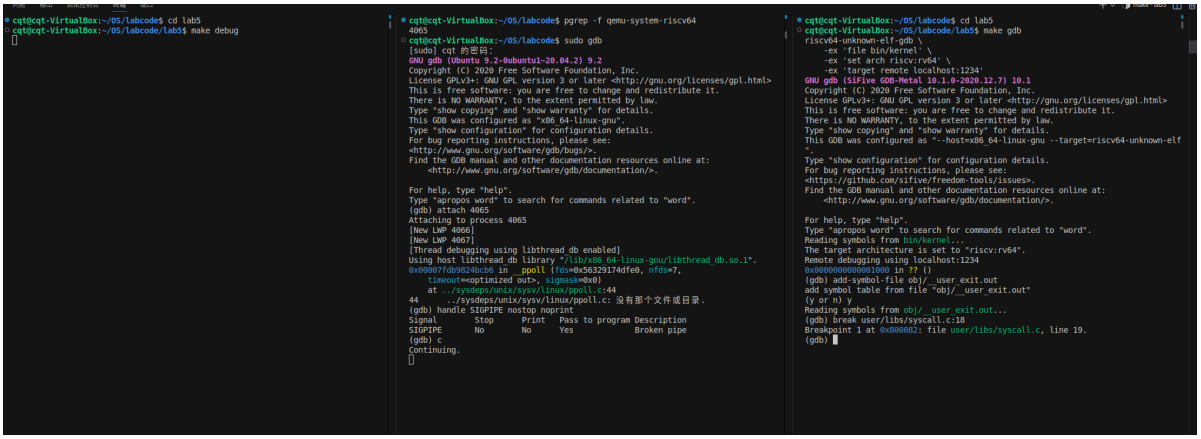
我们先不打断点，先让它跑起来，等 ucore 停在 ecall 前面了，我们再回来打断点。输入 c (continue)：

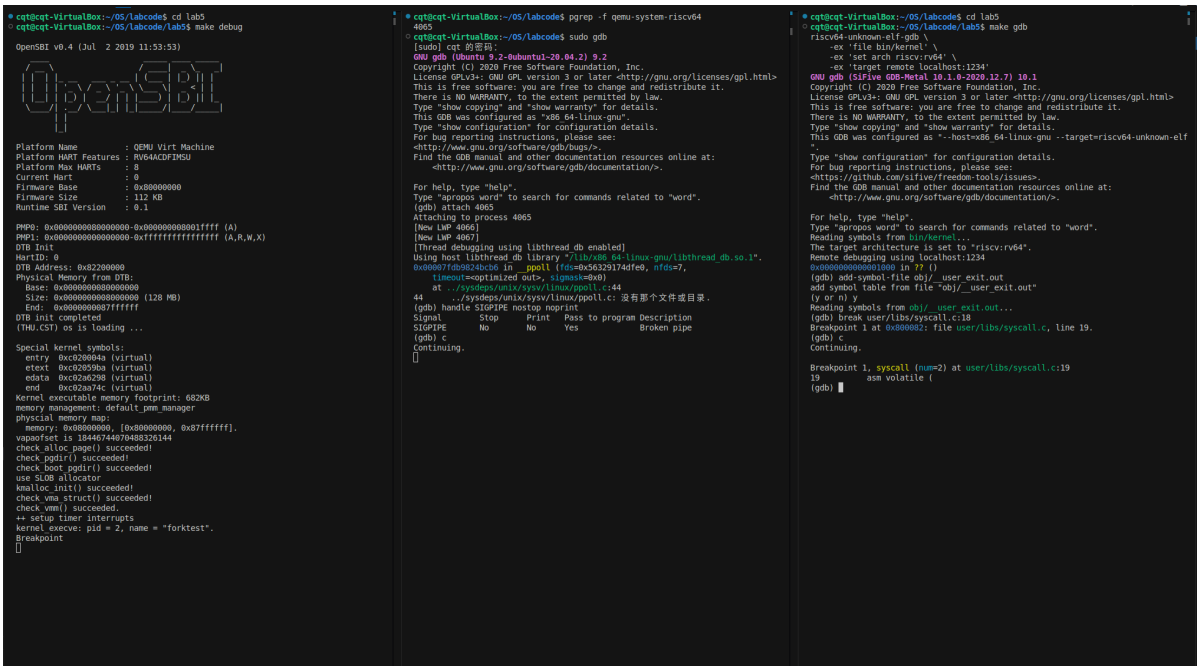


窗口3进入lab5，输入make gdb，然后加载用户程序符号表：

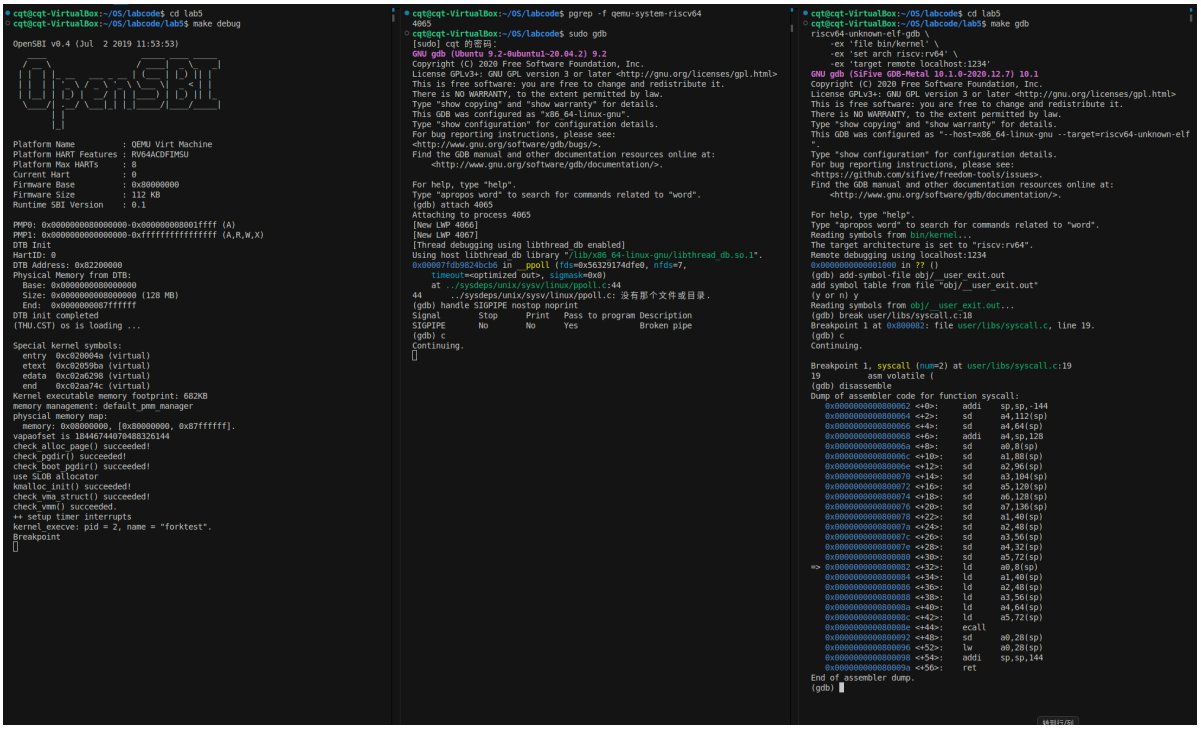


我们需要在用户态打断点，然后输入c：



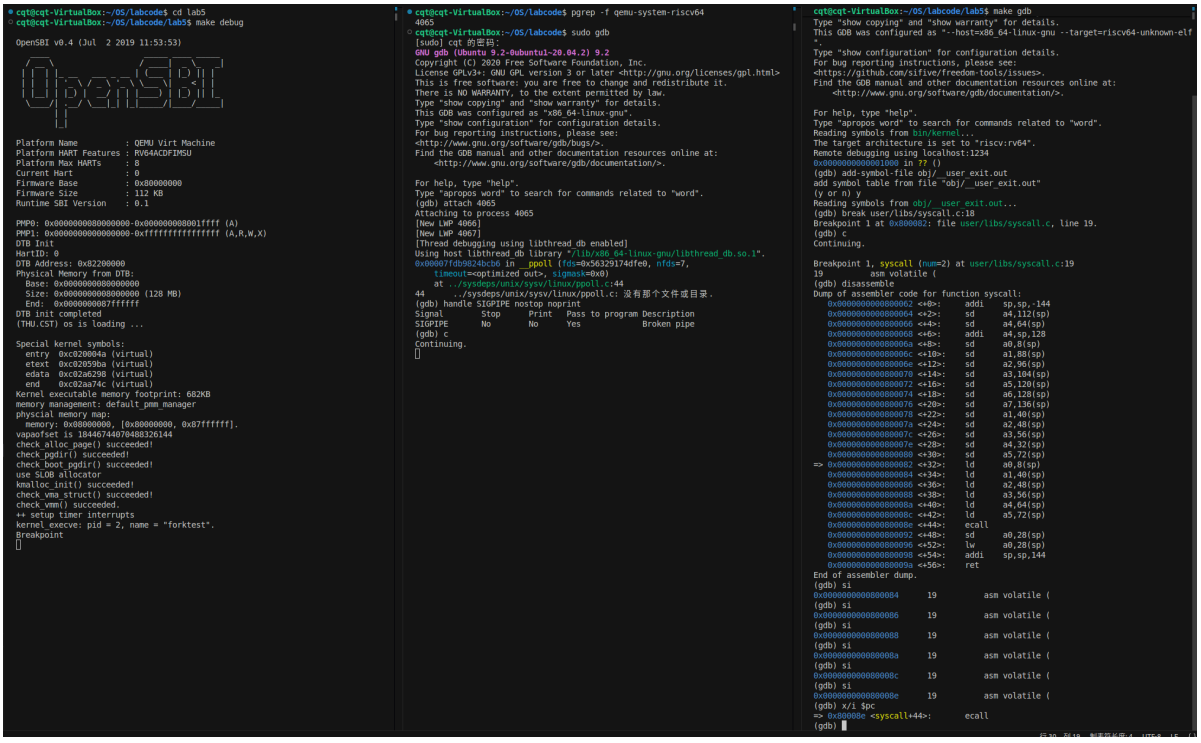


终端 3 (ucore) 会停在 syscall 函数里。输入 disassemble查看汇编代码：

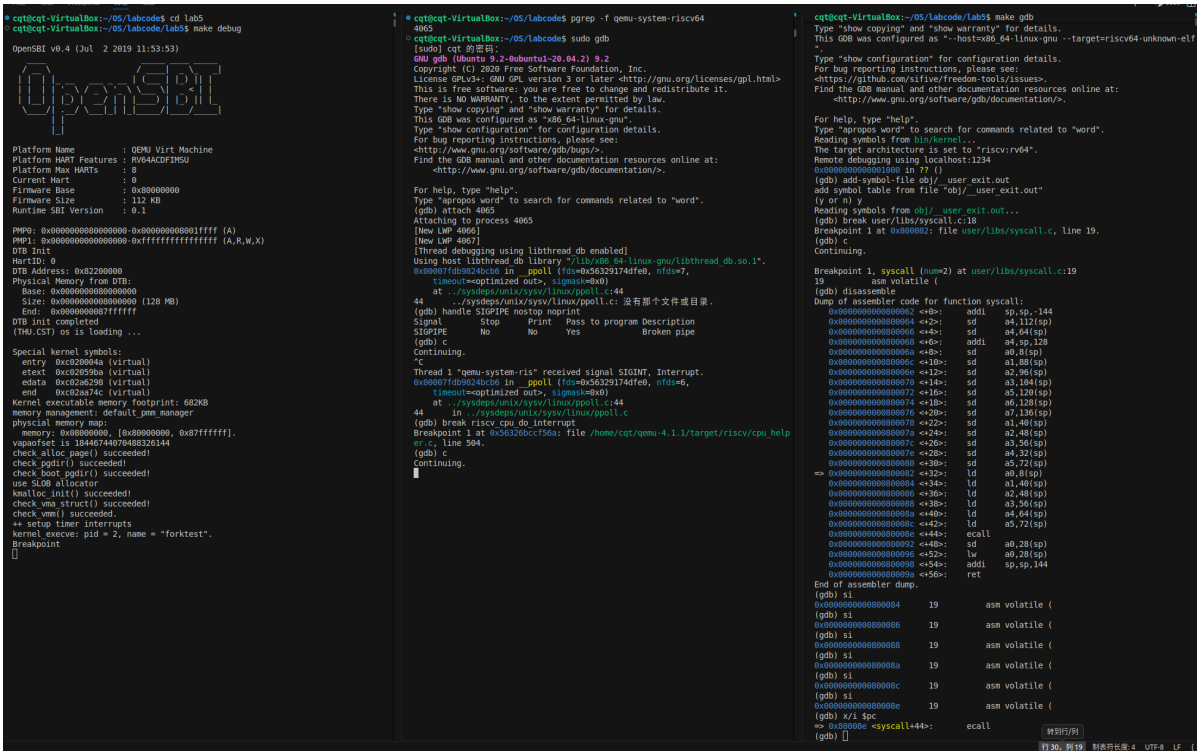


可以找到 ecall 指令的地址，在`0x000000000080008e`

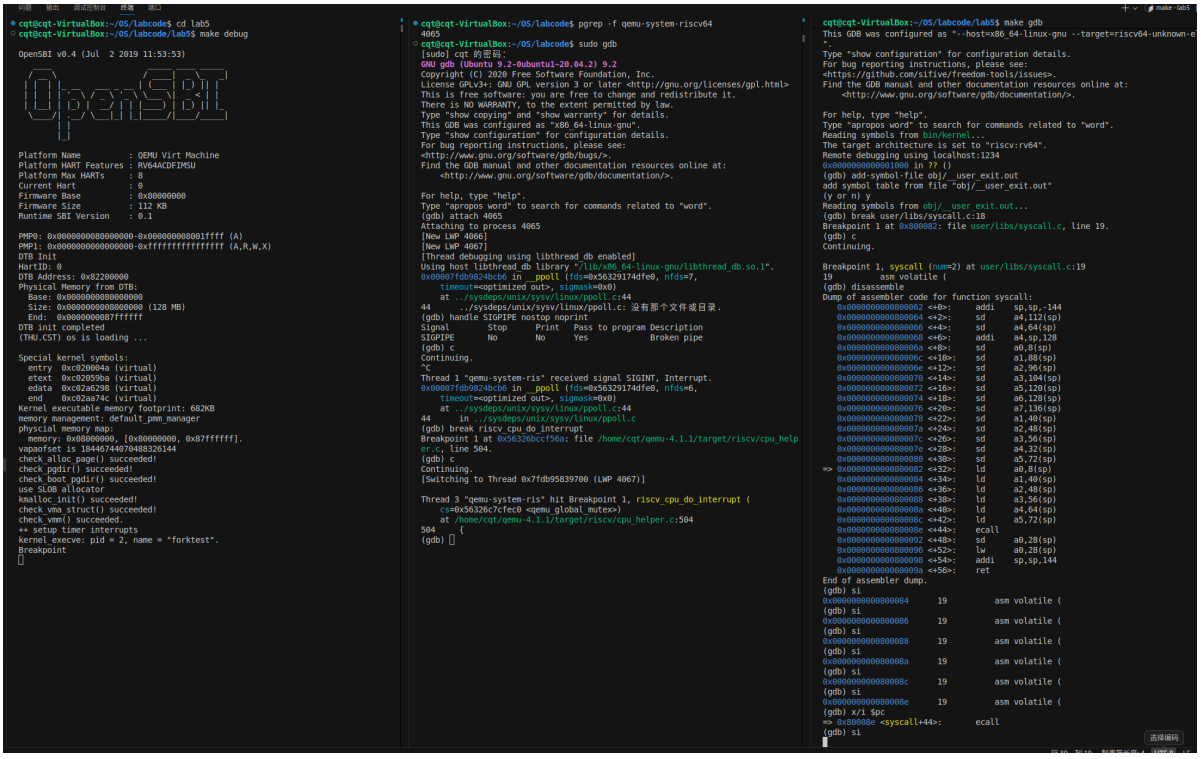
输入 si (step instruction) 单步执行汇编，直到 $pc 指向 ecall 指令的那一行，然后停在这里，现在 CPU 正准备执行 ecall：



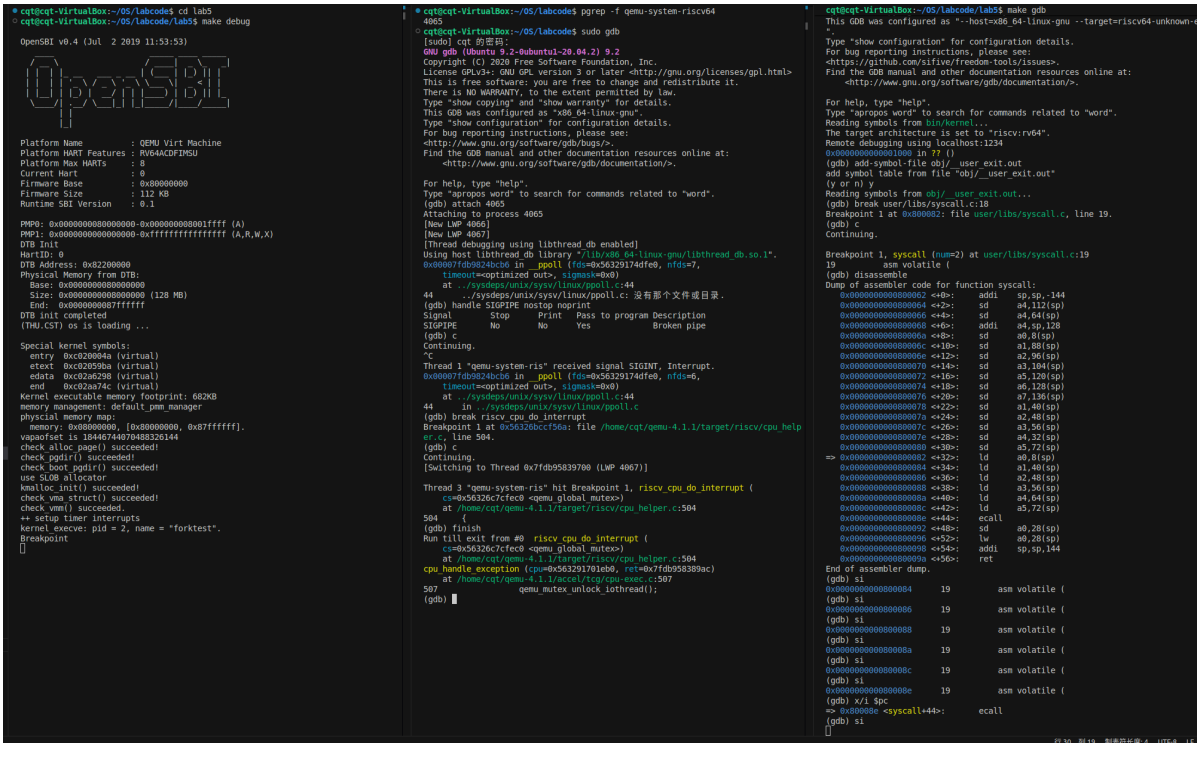
然后去窗口2，按Ctrl+C 暂停，输入 break riscv_cpu_do_interrupt，再输入c：



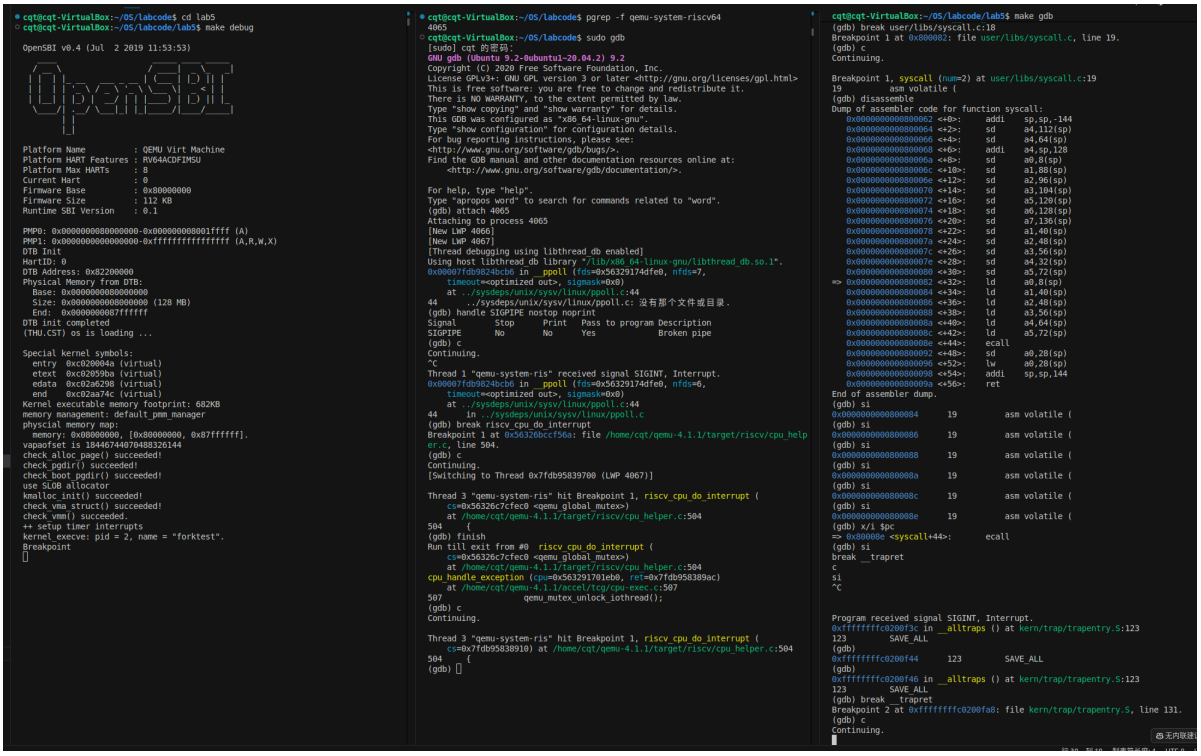
然后回到窗口3，输入si，执行那条 ecall，终端 2 会瞬间跳出 Breakpoint 1, riscv_cpu_do_interrupt …，这就是系统调用的现场！



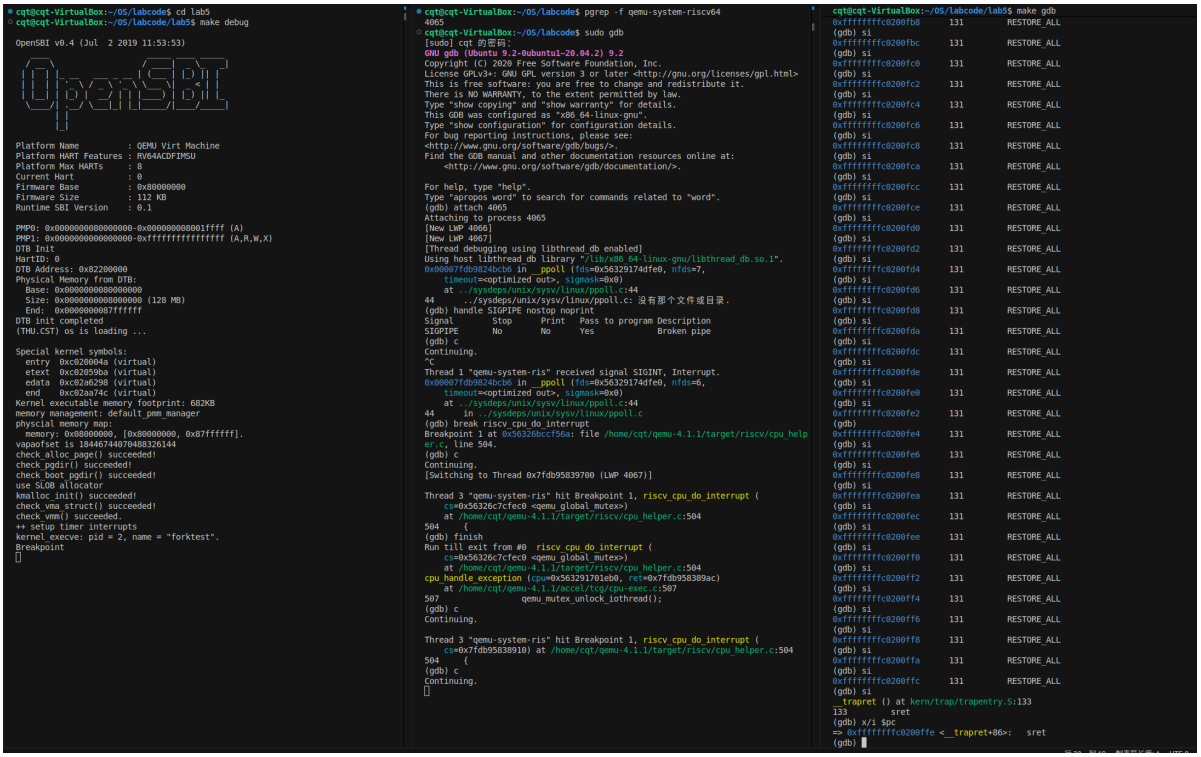
现在我们回到窗口2，输入finish，让它跑完中断处理



在窗口3中，现在我们进入了内核态，需要找到返回用户态的地方。输入 break __trapret (这是 ucore 从中断返回的汇编代码标签)，然后输入 c：

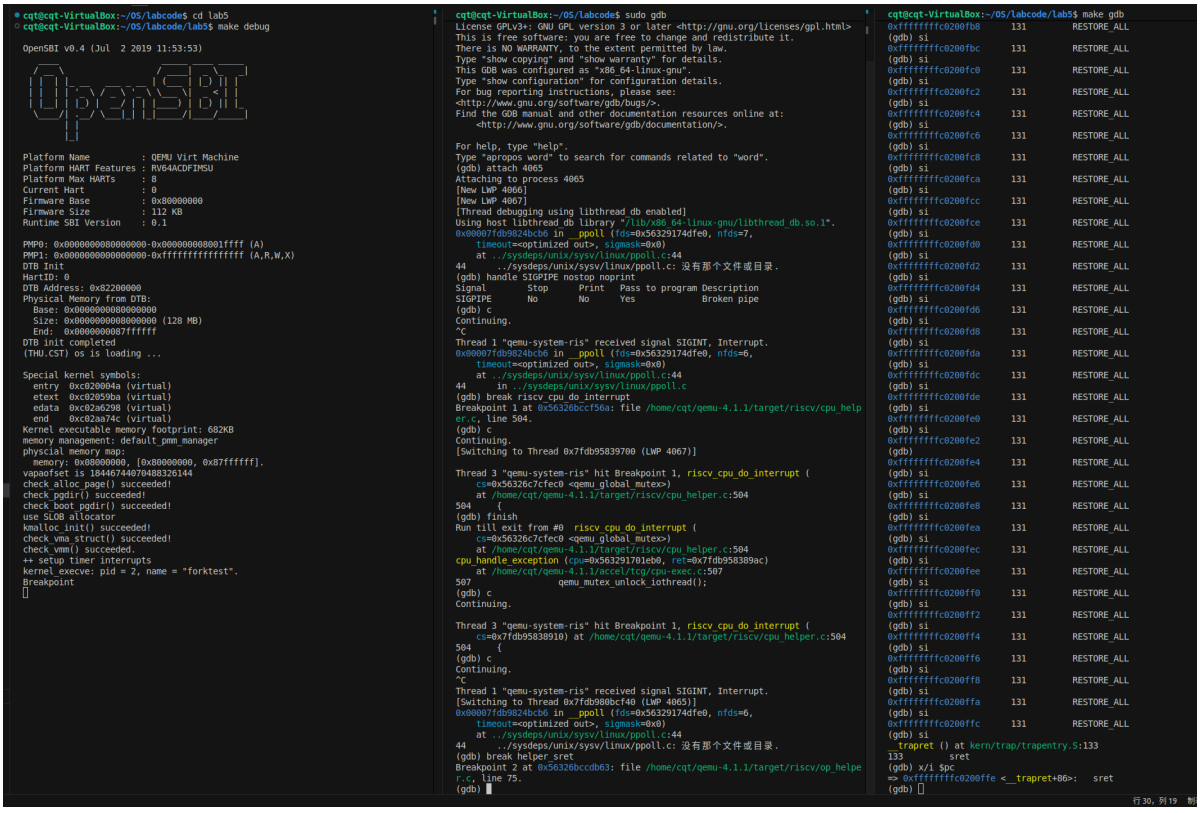


停下来后，单步 si 直到看到 sret 指令：



现在 ucore 已经站在了返回用户态的边上（sret 指令前）。我们要去 QEMU 里看看它是怎么模拟这个“跳跃”动作的。

在窗口2打断点，这次我们要抓的是 sret 的处理函数。在 QEMU RISC-V 源码里，这个函数通常叫 helper_sret：



然后continue，再去窗口3单步执行sret，捕获到系统调用的返回现场：