

# 本科生实验报告

实验课程: 操作系统原理实验

实验名称: 从内核态到用户态

专业名称: 计算机科学与技术

学生姓名: 数据删除

学生学号: 数据删除

实验成绩:

报告时间: 2025年5月29日

## 实验要求

### Assignment 1 系统调用

编写一个系统调用，然后在进程中调用之，根据结果回答以下问题。

* 展现系统调用执行结果的正确性，结果截图并并说说你的实现思路。
* 请根据gdb来分析执行系统调用后的栈的变化情况。
* 请根据gdb来说明TSS在系统调用执行过程中的作用。

### Assignment 2 Fork的奥秘

实现fork函数，并回答以下问题。

* 请根据代码逻辑和执行结果来分析fork实现的基本思路。
* 从子进程第一次被调度执行时开始，逐步跟踪子进程的执行流程一直到子进程从fork返回，根据gdb来分析子进程的跳转地址、数据寄存器和段寄存器的变化。同时，比较上述过程和父进程执行完ProgramManager::fork后的返回过程的异同。
* 请根据代码逻辑和gdb来解释fork是如何保证子进程的fork返回值是0，而父进程的fork返回值是子进程的pid。

### Assignment 3 哼哈二将 wait & exit

实现wait函数和exit函数，并回答以下问题。

* 请结合代码逻辑和具体的实例来分析exit的执行过程。
* 请分析进程退出后能够隐式地调用exit和此时的exit返回值是0的原因。
* 请结合代码逻辑和具体的实例来分析wait的执行过程。
* 如果一个父进程先于子进程退出，那么子进程在退出之前会被称为孤儿进程。子进程在退出后，从状态被标记为DEAD开始到被回收，子进程会被称为僵尸进程。请对代码做出修改，实现回收僵尸进程的有效方法。

## 实验过程

### Assignment 1 系统调用

系统调用的具体流程为：

1. 初始化系统调用服务，设置中断描述符表中0x80号中断为系统调用入口
2. 将系统调用处理函数注册到系统调用表中
3. 在用户程序中使用asm\_system\_call调用系统调用
4. 系统调用参数通过寄存器传递（eax, ebx, ecx, edx, esi）
5. 触发0x80中断，CPU从用户态切换到内核态
6. 执行相应的系统调用处理函数
7. 从内核态返回用户态，继续执行用户程序

系统调用从用户态切换到内核态时，会发生特权级切换，CPU自动从TSS中加载内核栈指针（esp0），并在内核栈中保存用户态的上下文（ss, esp, eflags, cs, eip）。CPU送入esp的值是在将当前进程调度上的CPU执行时，通过ProgramManager::activateProgramPage放入了TSS的。

因此测试系统调用的基本流程为：

1. 创建一个用户进程，观察esp是否从内核空间转移到用户空间，同时观察CPU是否已经准备好esp0以供切换。
2. 编写简单的系统调用并从用户进程调用之。
3. 在int 0x80前后下断，观察esp的变化情况。
4. 进入系统调用后，检查用户态上下文是否正确保存于系统栈。
5. 检查系统调用参数是否正确传递。
6. 检查系统调用后是否正确回到用户空间。

### Assignment 2 Fork的奥秘

Fork函数的主要功能是创建当前进程的一个副本，即子进程。子进程与父进程几乎完全相同，但有以下区别：

1. 它们有不同的进程ID（PID）
2. 子进程的父进程ID是父进程的PID
3. 子进程获得父进程资源的副本，而不是共享这些资源
4. 在父进程中，fork返回子进程的PID；在子进程中，fork返回0
5. 实现fork的关键是完整复制父进程的资源，包括：
   1. 0特权级栈（内核栈）
   2. PCB（进程控制块）
   3. 页目录表和页表
   4. 物理页内容
   5. 虚拟地址池
6. fork函数在子进程中返回0，在父进程中返回子进程的PID。这是通过在复制进程0级栈时设置返回值寄存器eax来实现的。

fork的具体步骤为：

1. 创建一个新的进程
2. 复制父进程的PCB信息到子进程
3. 复制父进程的用户虚拟地址池到子进程
4. 复制父进程的页目录表、页表和物理页到子进程
5. 设置子进程fork的返回值为0
6. 设置父进程的返回值为子进程的PID

父进程在fork的返回值是子进程pid，这是显然的，但要让子进程的pid返回值是0就需要费一番功夫。具体来说，需要改动子进程0级栈的eax值为0，然后调用asm\_load\_process装载pss，让子进程在相同上下文但不同的栈和物理页上恢复运行。

### Assignment 3 哼哈二将 wait & exit

exit函数的实现步骤为：

1. 标记当前进程的PCB状态为DEAD
2. 保存退出码到PCB的retValue字段
3. 释放进程占用的资源（物理页、页表、页目录表、虚拟地址池bitmap等）
4. 调用调度器进行进程切换
5. 要在进程正常结束时自动调用exit(0)，需要在创建进程时，在用户栈中设置返回地址为exit，参数为0。

wait函数的实现步骤为：

1. 遍历所有进程，查找当前进程的子进程
2. 如果找到一个DEAD状态的子进程，获取其返回值，释放其PCB，并返回其PID
3. 如果没有子进程，则返回-1
4. 如果存在子进程但都未终止，则调用ProgramManager::schedule进行调度。
5. 子进程的回收由父进程处理。

如果一个父进程先于子进程退出，那么子进程在退出之前会被称为孤儿进程。如果放置不管，子进程退出后会成为僵尸进程。因为子进程回收由父进程处理，父进程退出后必须指定另外一个进程接管子进程。因此仿照linux的实现创建一个init进程，该进程不会退出且pid固定为1，当父进程退出时，在exit函数里找到其所有子进程，把这些子进程的父进程pid全部改成1，就可以做到接管的效果。init进程不断进行wait操作，把处于DEAD状态的子进程及时释放。

在内核线程里创建的进程，其父进程pid均为0，退出后同样无法回收，同理在exit时进行检查，如果即将退出的进程没有父进程，也将其挂到init用以销毁。

## 关键代码

### Assignment 1 系统调用

测试代码：

int syscall\_0(int first, int second, int third, int forth, int fifth) {

    printf("system call 0: %d, %d, %d, %d, %d\n",

           first, second, third, forth, fifth);

    return first + second + third + forth + fifth;

}

void first\_process() {

    asm\_system\_call(0, 132, 324, 12, 124);

    asm\_halt();

}

void first\_thread(void\*) {

    printf("start process\n");

    programManager.executeProcess((char\*)first\_process, 1);

    programManager.executeProcess((char\*)first\_process, 1);

    programManager.executeProcess((char\*)first\_process, 1);

    asm\_halt();

}

### Assignment 2 Fork的奥秘

测试代码：

void first\_process() {

    int pid = fork();

    if (pid == -1) {

        printf("can not fork\n");

    }

    else {

        if (pid) {

            printf("I am father, fork return: %d\n", pid);

        }

        else {

            printf("I am child, fork return: %d, my pid: %d\n", pid, programManager.running->pid);

        }

    }

    asm\_halt();

}

void first\_thread(void\*) {

    printf("start process\n");

    programManager.executeProcess((const char\*)first\_process, 1);

    printf("first thread finished\n");

    asm\_halt();

}

### Assignment 3 哼哈二将 wait & exit

修改ProgramManager::exit，使其能够处理孤儿进程和僵尸进程：

void ProgramManager::exit(int ret) {

    // 关中断

    interruptManager.disableInterrupt();

    // 第一步，标记PCB状态为`DEAD`并放入返回值。

    PCB\* program = this->running;

    program->retValue = ret;

    program->status = ProgramStatus::DEAD;

    printf("exit program pid=%d name \"%s\" ret=%d\n", program->pid, program->name, ret);

    uint \*pageDir, \*page;

    uint paddr;

    // 第二步，如果PCB标识的是进程，则释放进程所占用的物理页、页表、页目录表和虚拟地址池bitmap的空间。

    if (program->pageDirectoryAddress) {

        pageDir = (uint\*)program->pageDirectoryAddress;

        for (int i = 0; i < 768; ++i) {

            if (!(pageDir[i] & 0x1)) {

                continue;

            }

            page = (uint\*)(0xffc00000 + (i << 12));

            for (int j = 0; j < 1024; ++j) {

                if (!(page[j] & 0x1)) {

                    continue;

                }

                paddr = memoryManager.vaddr2paddr((i << 22) + (j << 12));

                memoryManager.releasePhysicalPages(AddressPoolType::USER, paddr, 1);

            }

            paddr = memoryManager.vaddr2paddr((int)page);

            memoryManager.releasePhysicalPages(AddressPoolType::USER, paddr, 1);

        }

        memoryManager.releasePages(AddressPoolType::KERNEL, (int)pageDir, 1);

        int bitmapBytes = ceil(program->userVirtual.resources.length, 8);

        int bitmapPages = ceil(bitmapBytes, PAGE\_SIZE);

        memoryManager.releasePages(AddressPoolType::KERNEL,

                                   (uint)program->userVirtual.resources.bitmap,

                                   bitmapPages);

    }

    // 第三步，将未完成的子进程挂到 init 进程中

    for (auto pid : allPrograms) {

        PCB\* child = getPCB(pid);

        if (child->parentPid == program->pid) {

            child->parentPid = 1; // 将子进程的父进程设置为 init 进程

        }

    }

    // 第四步，如果当前进程没有父进程，则将自己也挂到 init 进程中。

    if (program->parentPid == 0) {

        program->parentPid = 1; // 将当前进程的父进程设置为 init 进程

    }

    // 第五步，立即执行线程/进程调度。

    schedule();

}

测试代码：

void first\_process() {

    int pid = fork();

    int retval;

    if (pid) {

        pid = fork();

        if (pid) {

            // while ((pid = wait(&retval)) != -1) {

            //  printf("wait for a child process, pid: %d, return value: %d\n",

            //         pid, retval);

            // }

            // printf("all child process exit, programs: %d\n",

            //     programManager.allPrograms.size());

            printf("father process exit, pid: %d\n", programManager.running->pid);

            // 直接退出父进程，不使用exit

        }

        else {

            uint32 tmp = 0xffffff;

            while (tmp)

                --tmp;

            printf("exit, pid: %d\n", programManager.running->pid);

            exit(123934);

        }

    }

    else {

        uint32 tmp = 0xffffff;

        while (tmp)

            --tmp;

        printf("exit, pid: %d\n", programManager.running->pid);

        exit(-123);

    }

}

void second\_thread(void\*) {

    printf("thread exit\n");

    // exit(0);

}

void init() {

    while (int pid = wait(nullptr)) {

        if (pid != -1) {

            printf("process %d exit. current programs count = %d\n", pid, programManager.allPrograms.size());

            for (int pid : programManager.allPrograms) {

                PCB\* program = programManager.getPCB(pid);

                printf("pid: %d, name: %s, status: %d\n", pid, program->name, (int)program->status);

            }

        }

    }

}

void first\_thread(void\*) {

    printf("start process\n");

    programManager.executeProcess((const char\*)init, 1);

    programManager.executeProcess((const char\*)first\_process, 1);

    programManager.executeThread(second\_thread, nullptr, "second", 1);

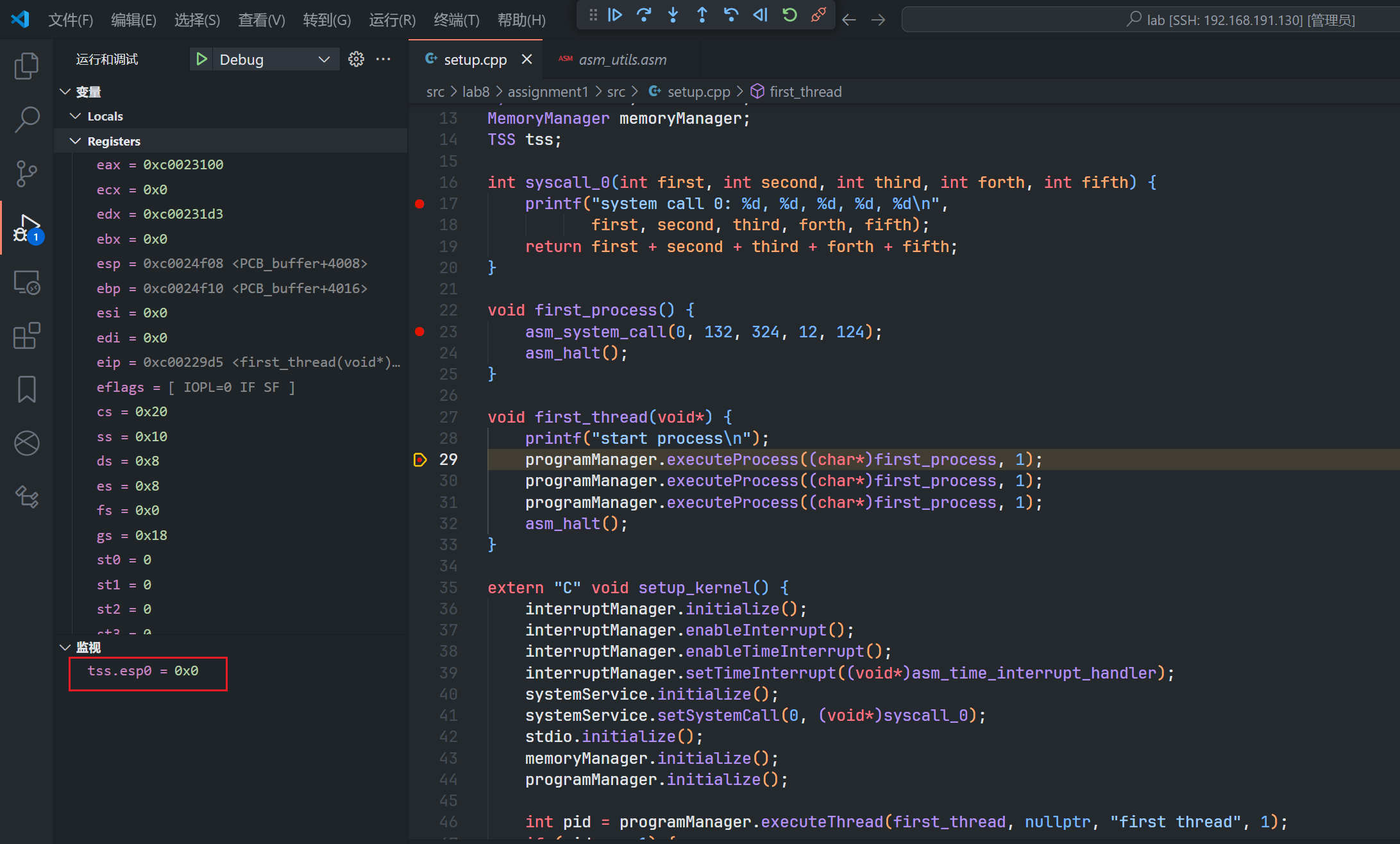
    asm\_halt();

}

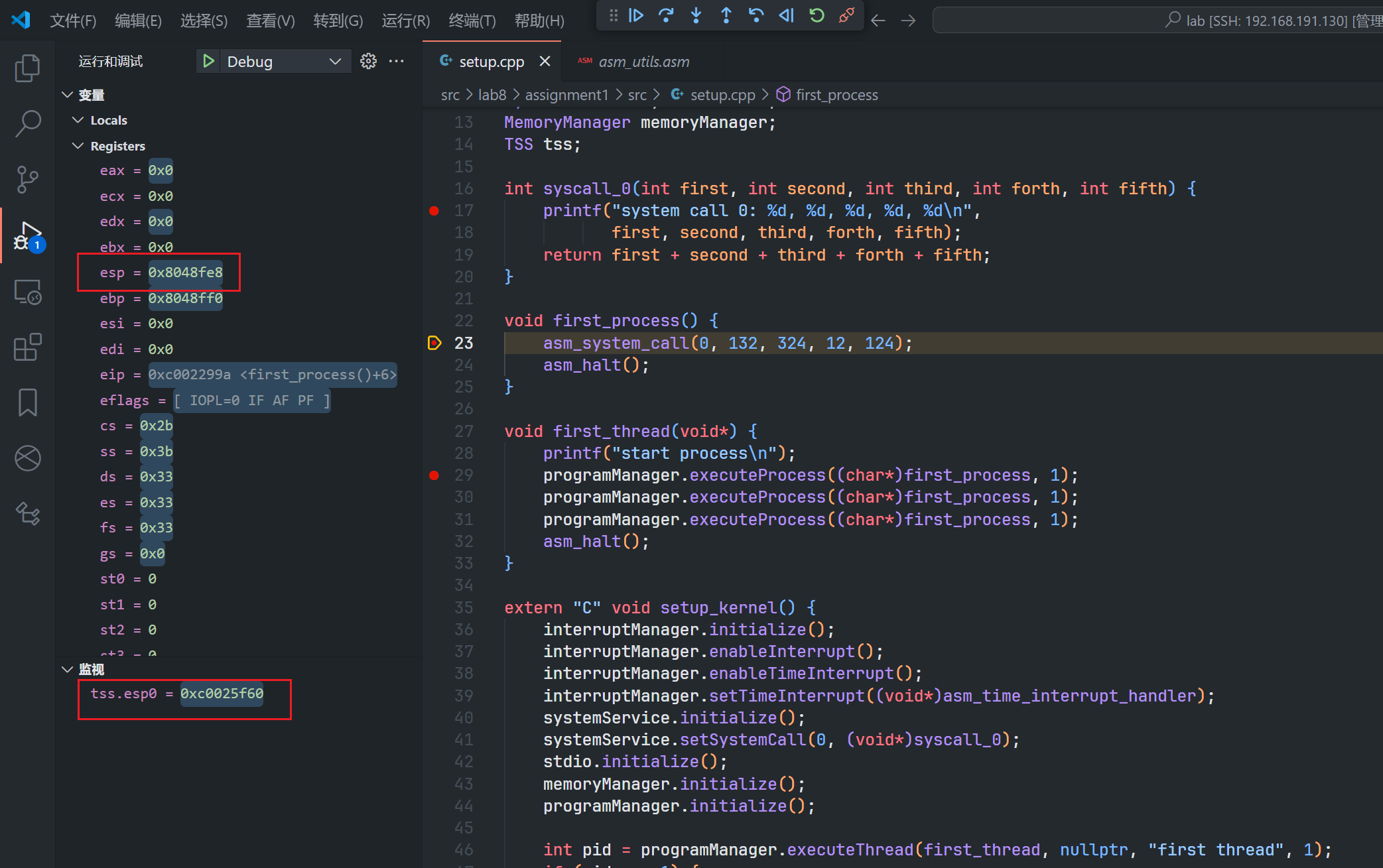
## 实验结果

### Assignment 1 系统调用

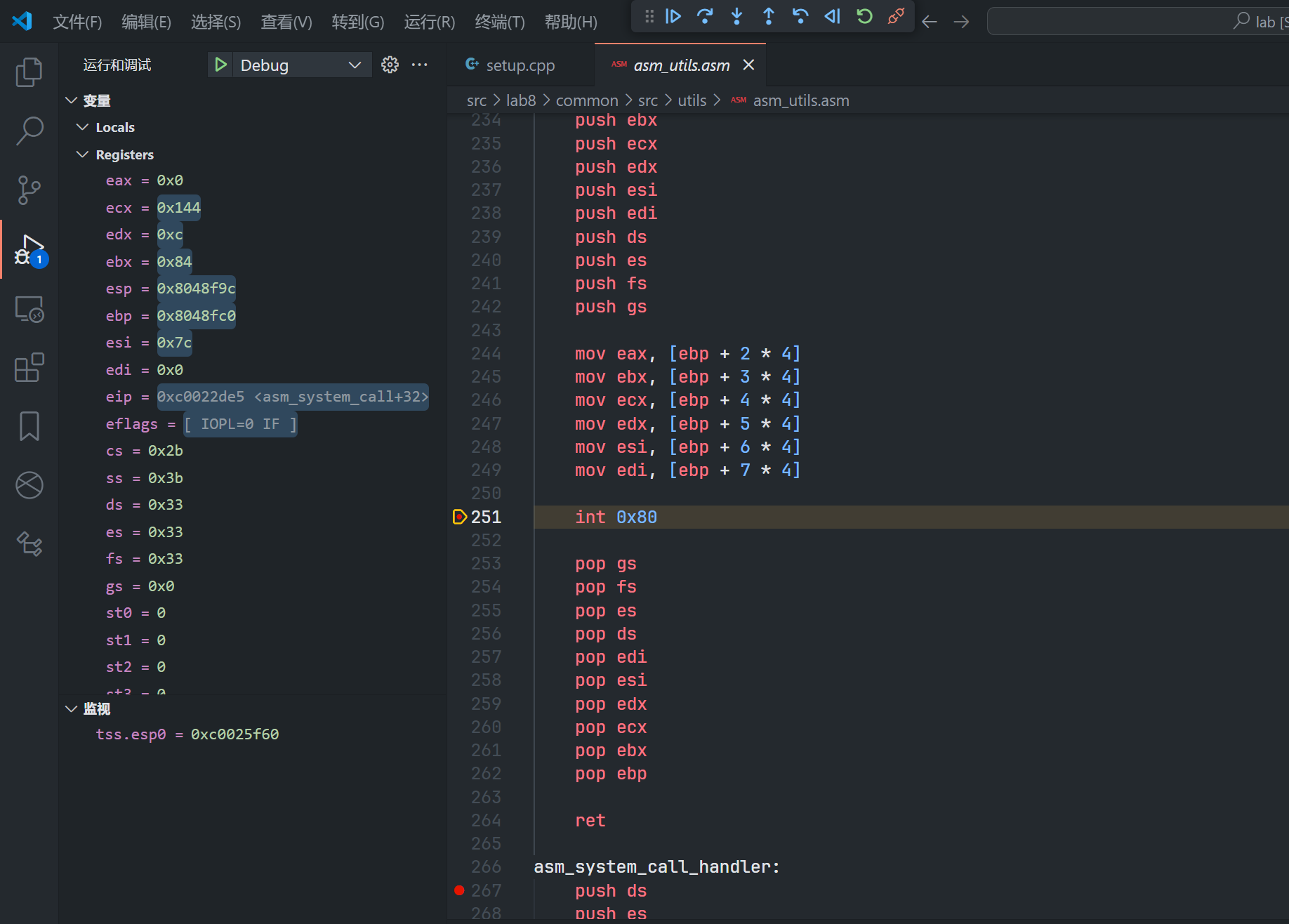
首先在内核态下断，观察tss.esp0，目前仍然是0：



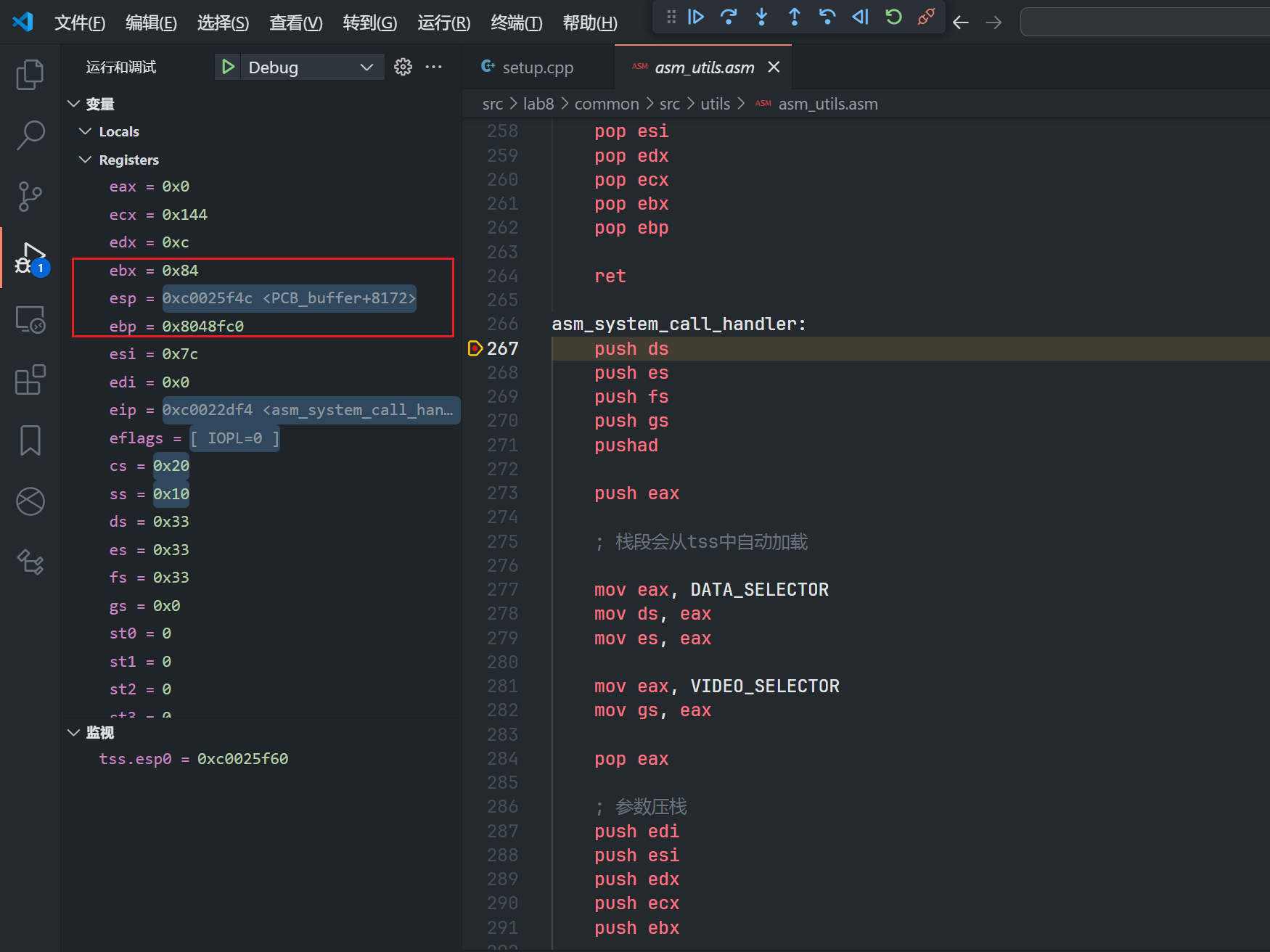
创建进程进入用户态，经过调度和activateProgramPage后，esp0和esp均变化，esp0现在变为内核栈地址，而esp所在的地址区域变为用户内存：



单步进入来到中断前一刻：

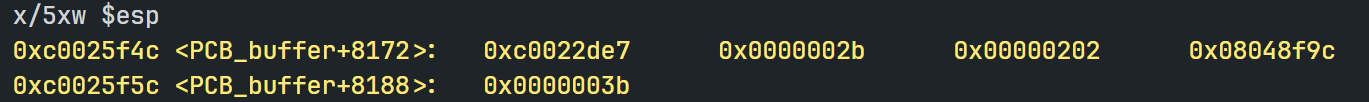


进入中断，esp变化到内核区：

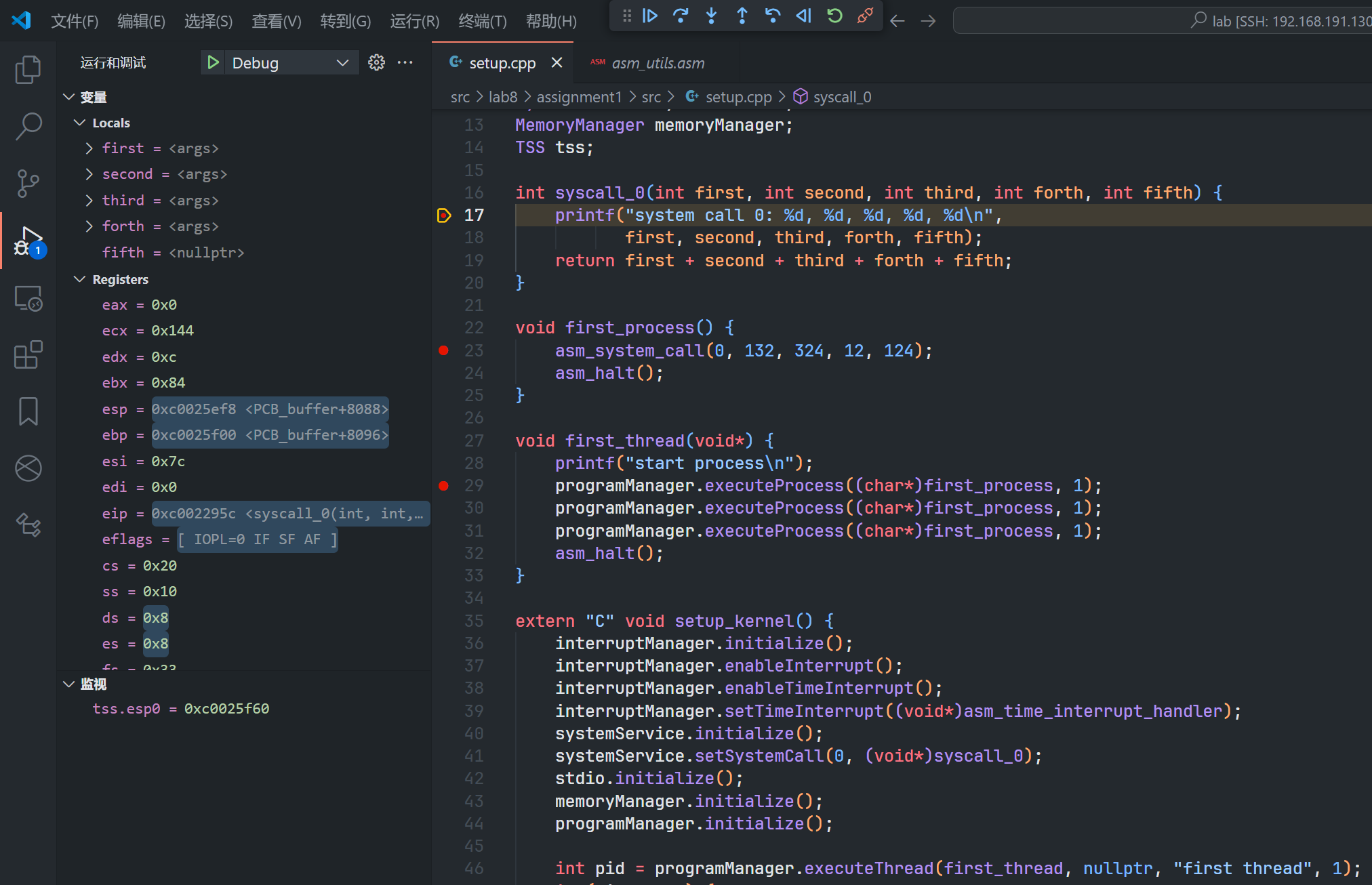


发现此时esp为0xc0025f4c，比esp0少了20，说明CPU自动将一些数据写入了内核栈，用于中断返回。

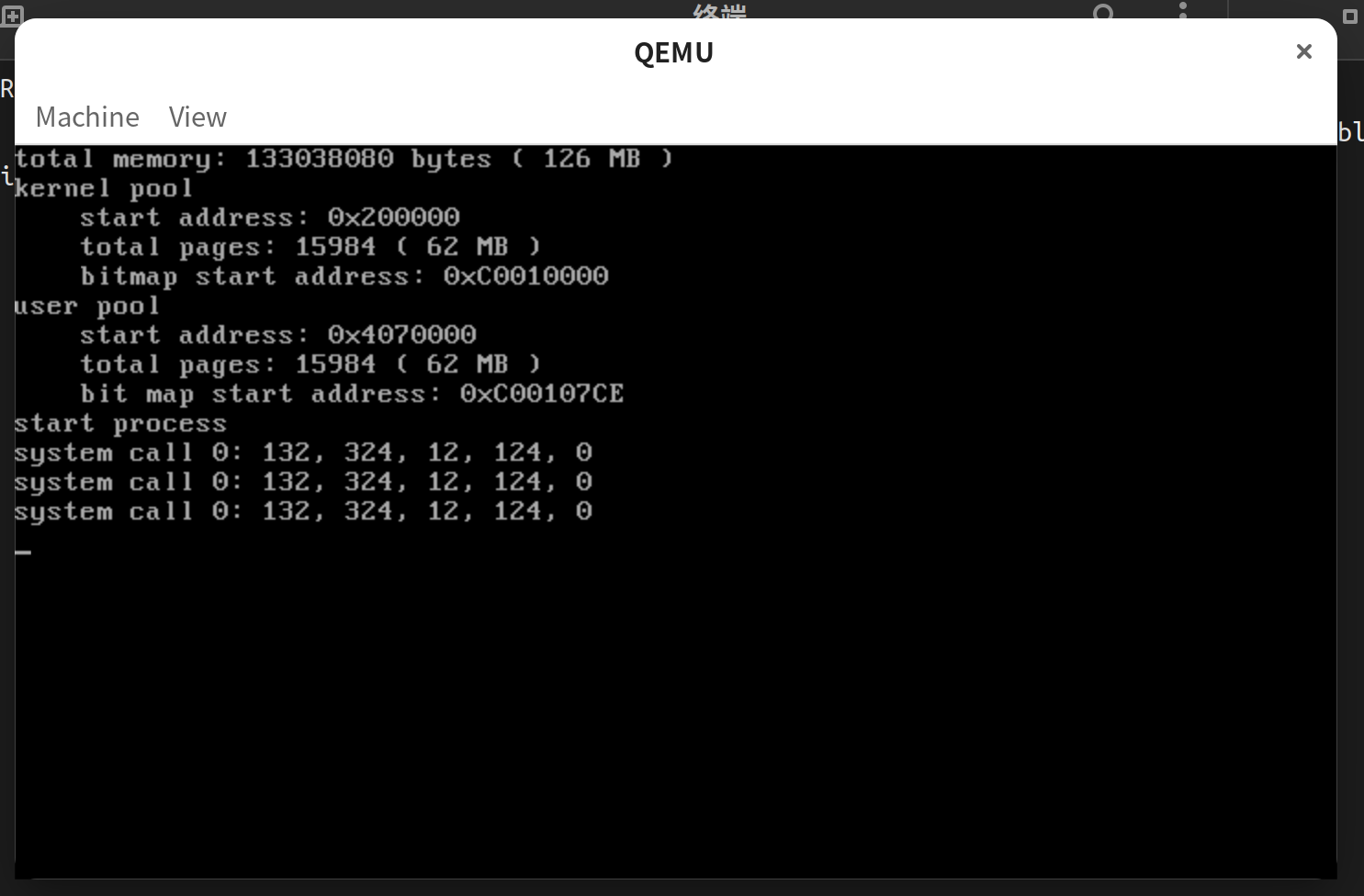
打印栈顶的5个4字节整数：



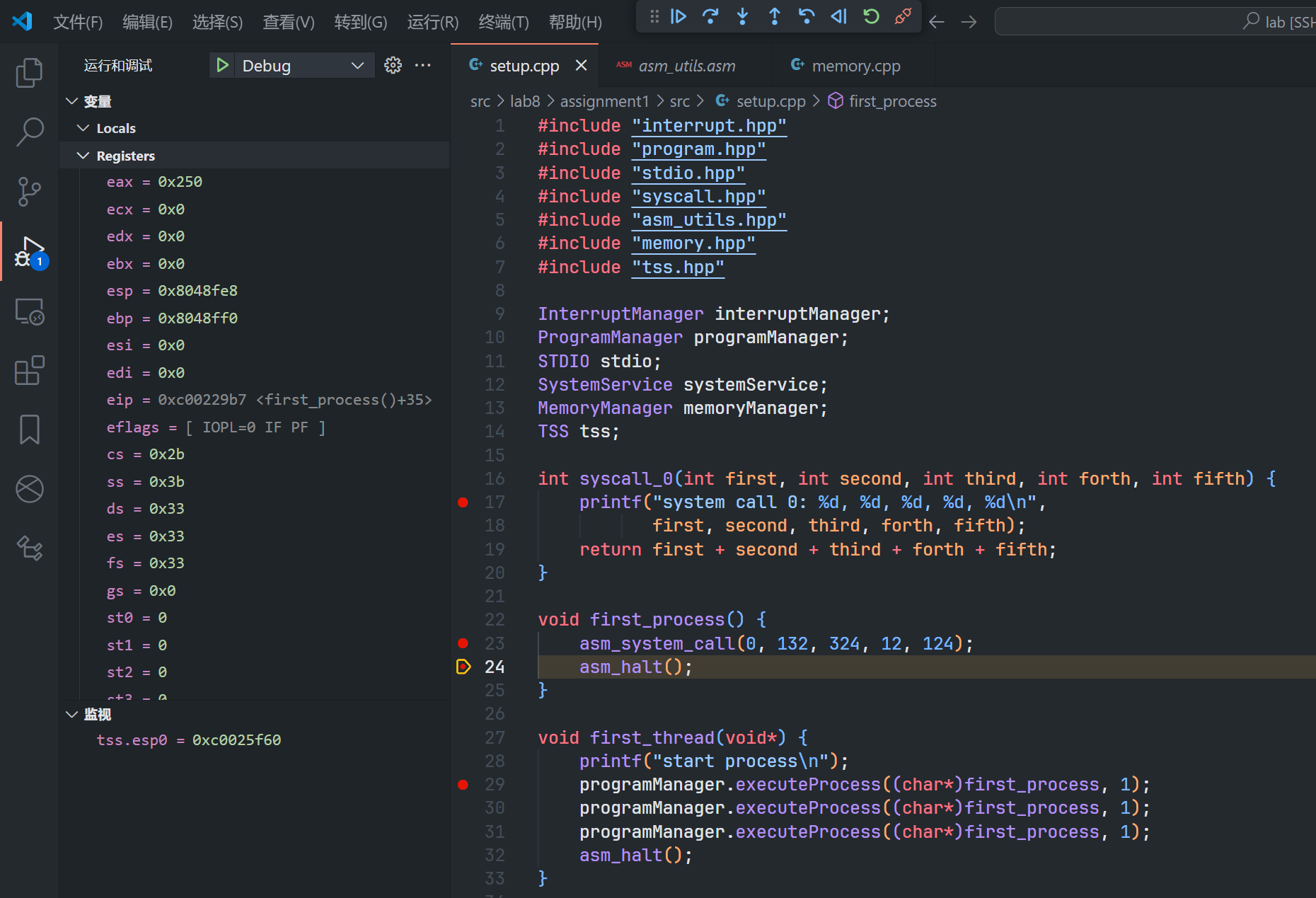
发现这5个数（从先到后）分别是原用户栈的ss, esp, eflags, cs, next eip，可以对比前两张图得到该结论。



继续执行，成功跳转到syscall\_0并输出：



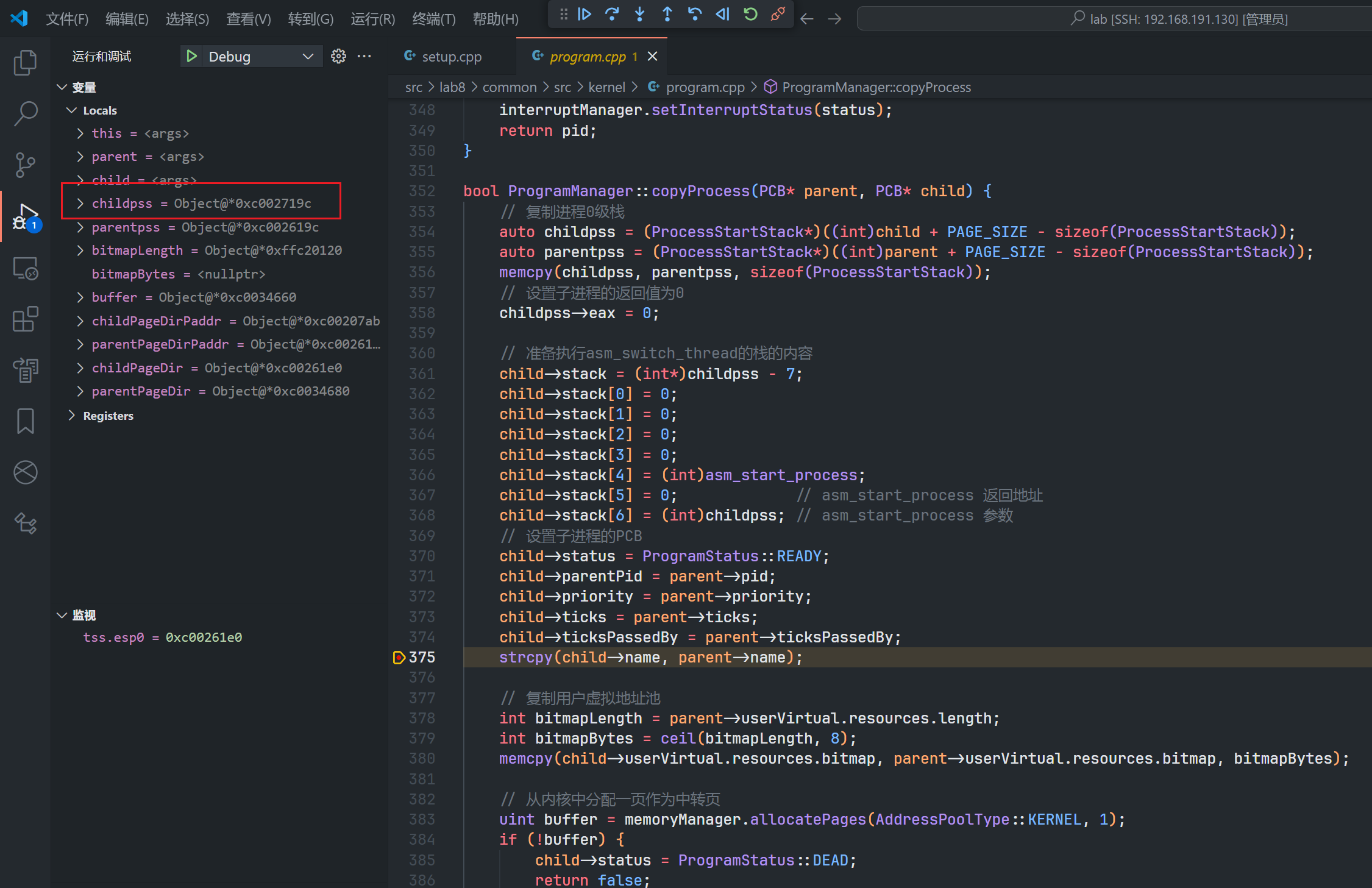
输出了正确的数字，说明参数成功从用户内存转移到内核内存。



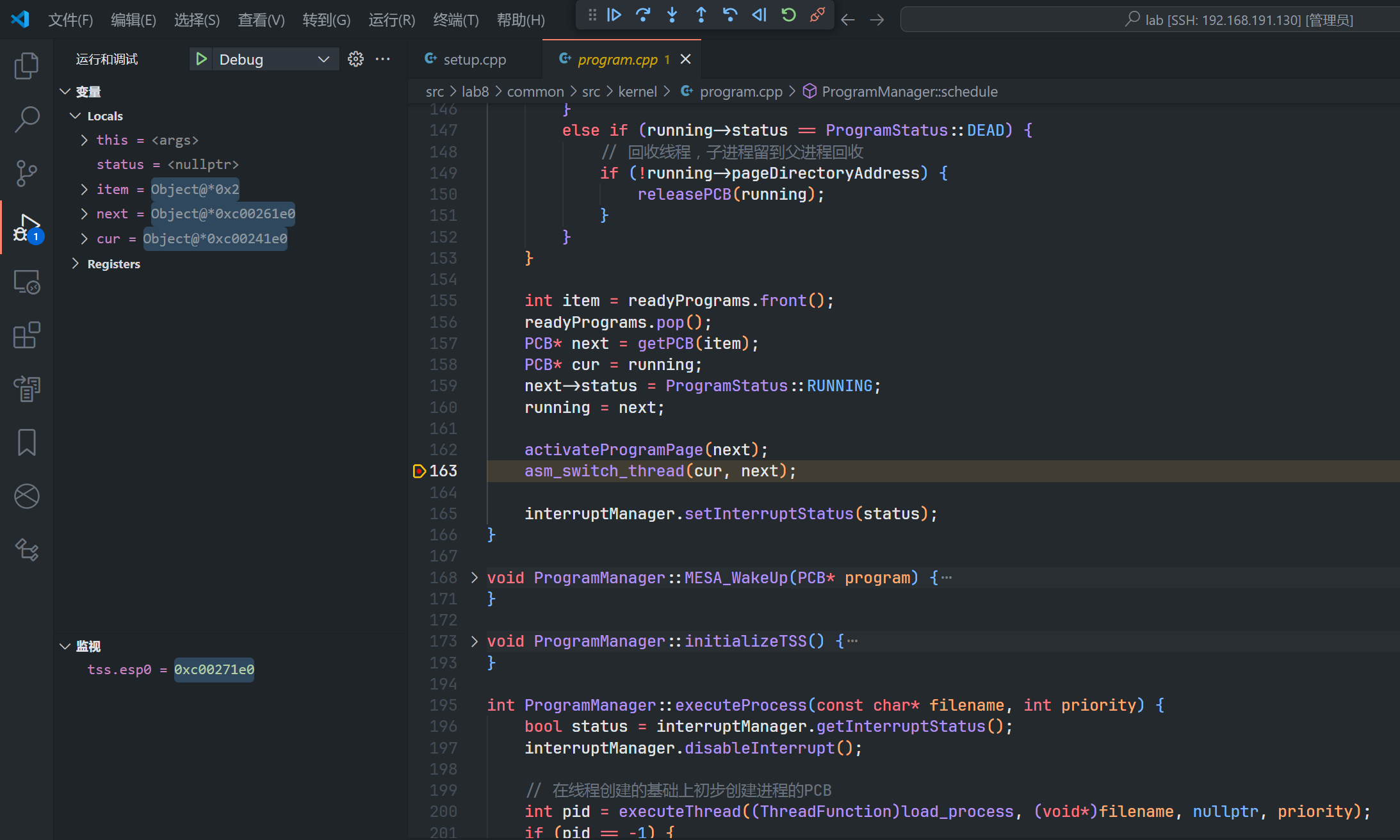
调用结束，成功回到用户态的下一条语句。

### Assignment 2 Fork的奥秘

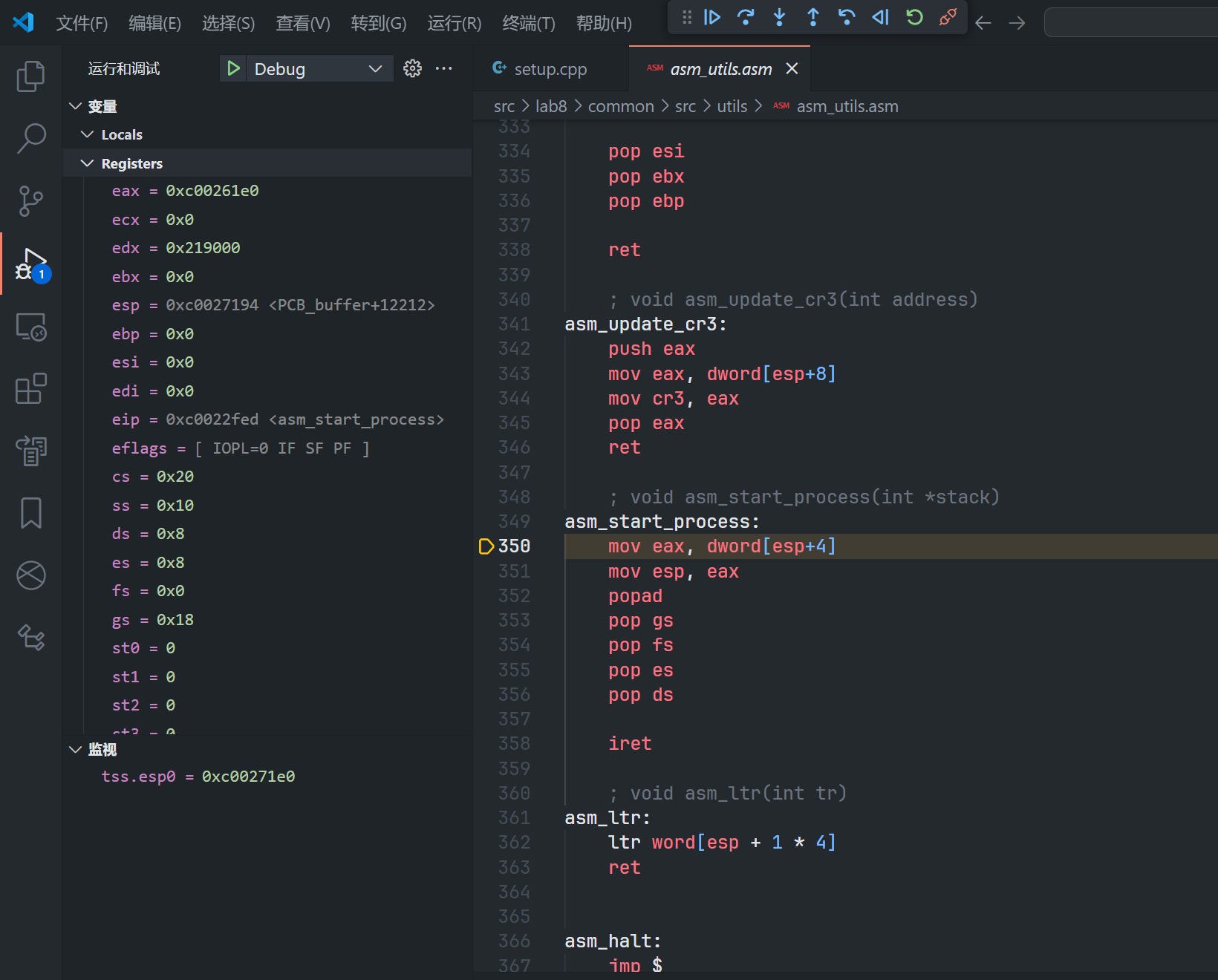
追踪数据复制的过程，copyProcess首先把0级栈复制，然后将childpss->eax设为0，子进程的上下文就和父进程一致，除了返回值，因此子进程在进行asm\_start\_process之后就能直接从fork返回，该过程全程对用户透明。



已知父进程的pid是1，因此子进程pid应当为2。在asm\_switch\_thread下断，找到pid为2的调度：

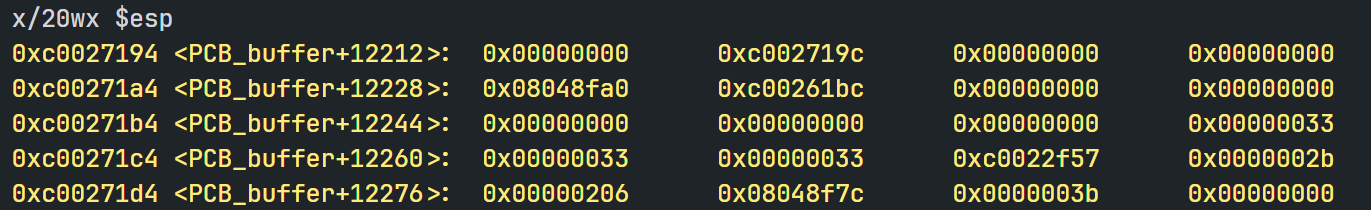


单步进入，在asm\_switch\_thread结束后预期跳转到asm\_start\_process：

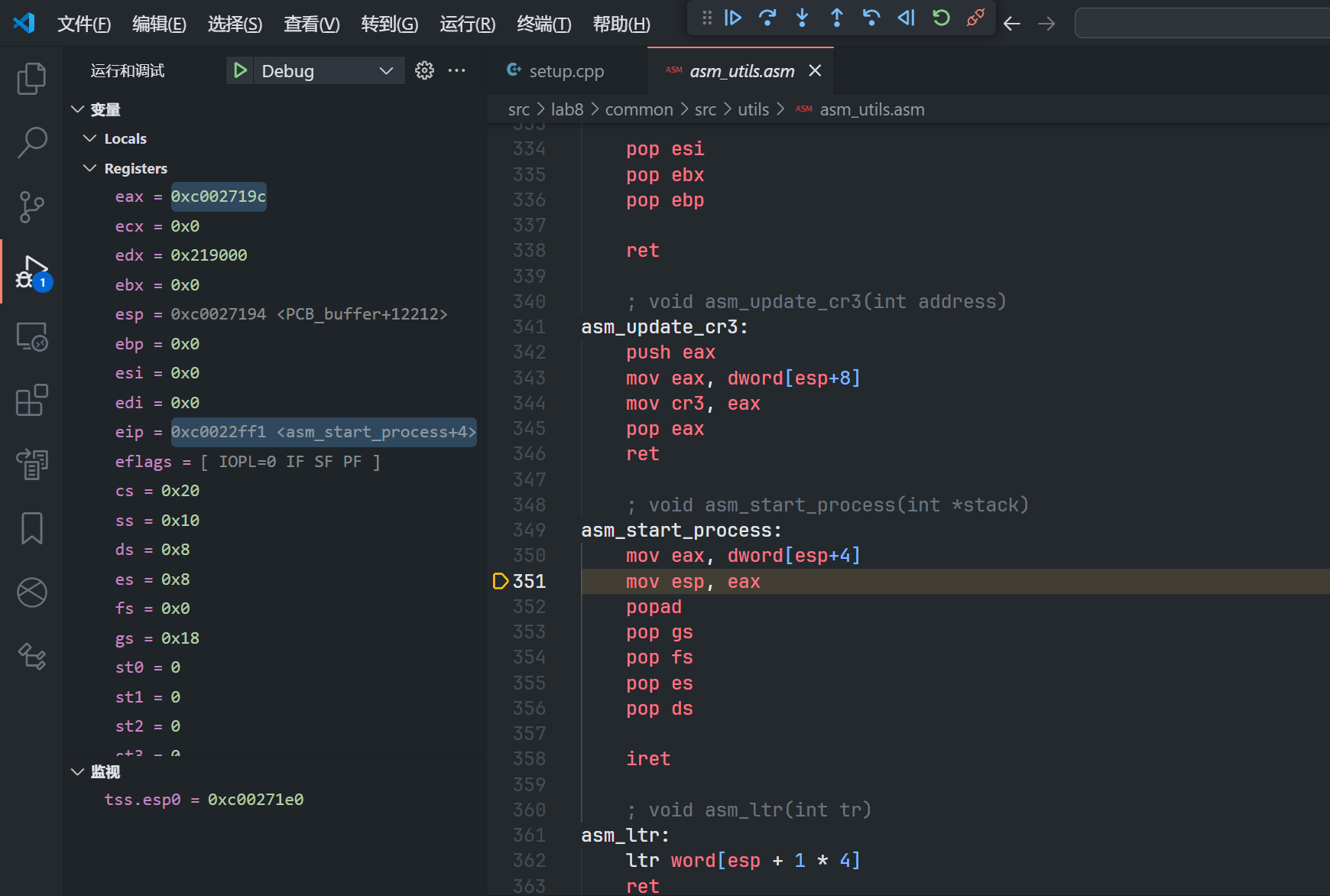


这是因为在ProgramManager::copyProcess中，子进程栈的返回地址被设为了asm\_start\_process,参数则指向了复制了的0级栈。

打印栈顶数据：

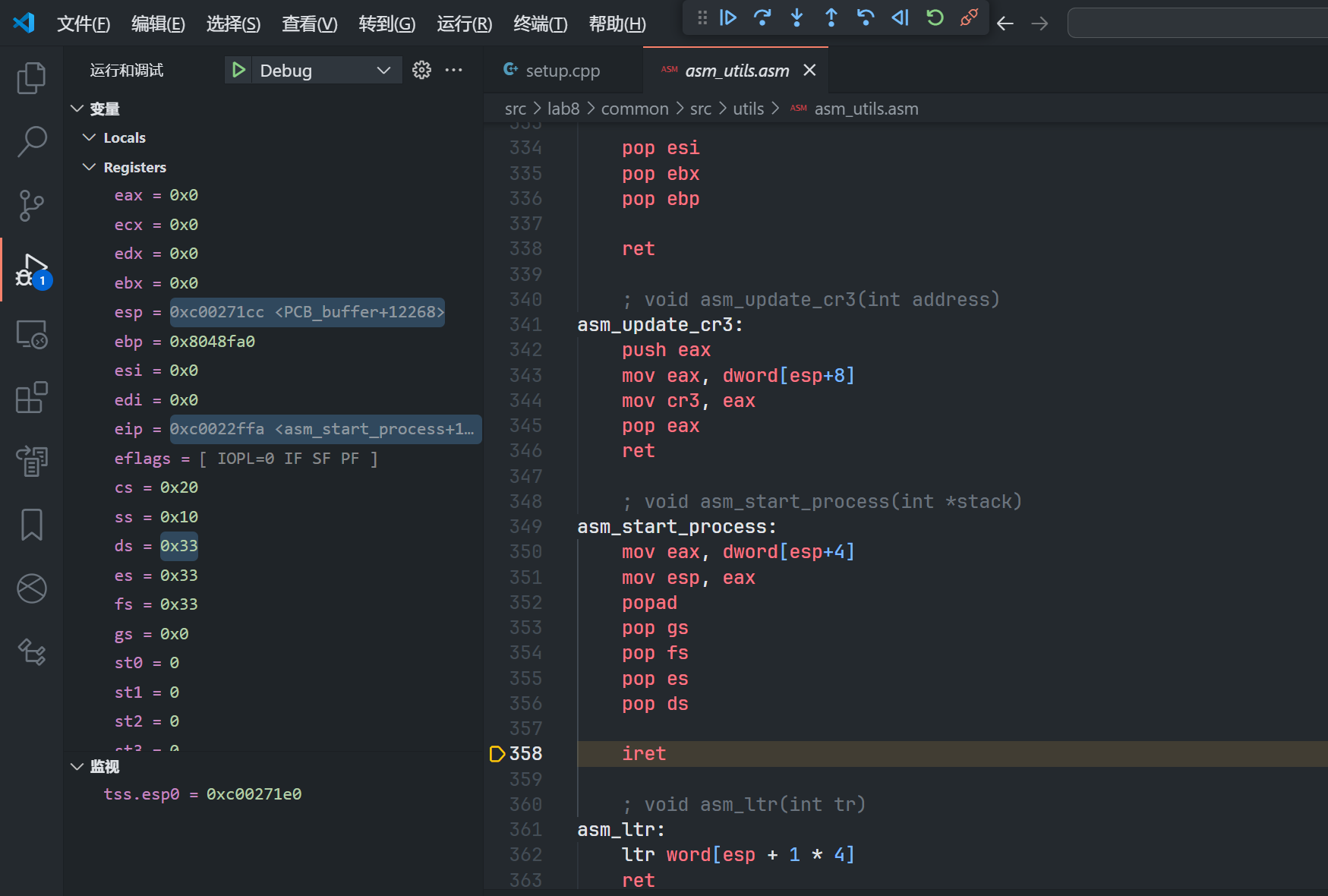


可以看到esp+4为0xc002719c，即childpss的地址。该地址即将被赋至esp，这样栈空间就转移到了子进程的栈空间：

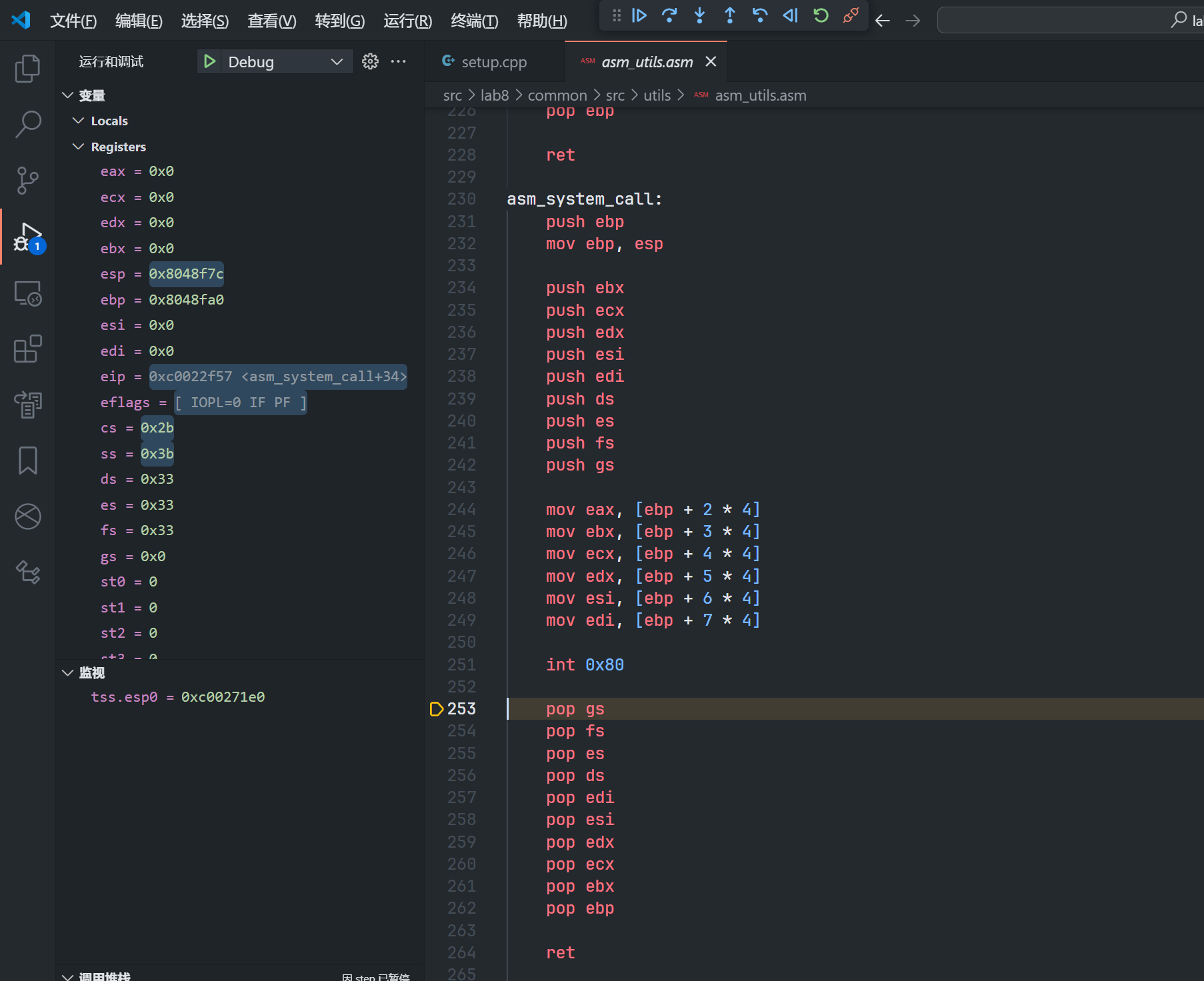


同时esp+10为0x8048fa0，这个是进程调用int 0x80后一条指令的地址，通过它来返回用户空间。

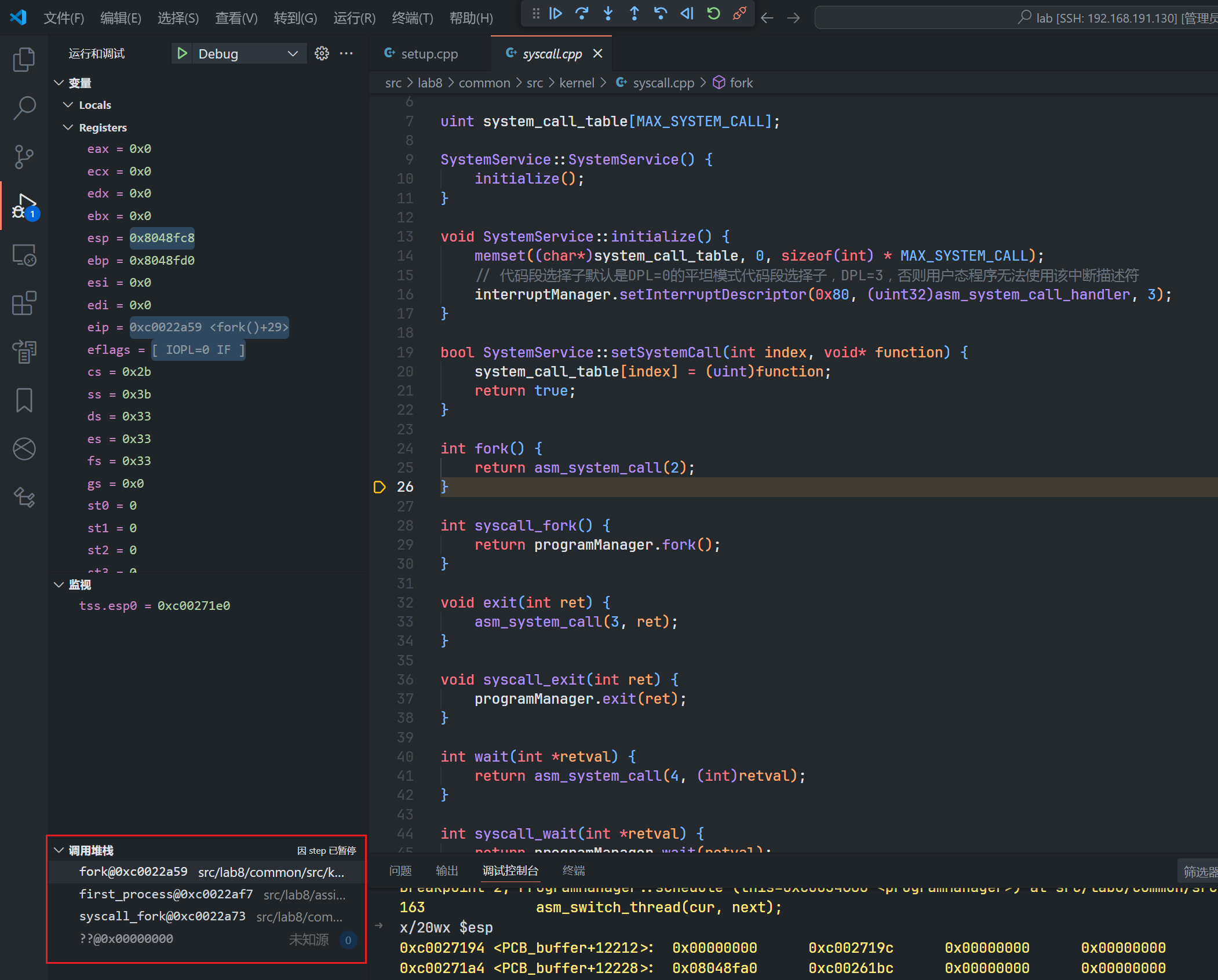
popad取出通用寄存器的值，此时eax变为0，即fork返回值在子进程的结果，同时段寄存器也变为用户段选择子，ebp变为0x8048fa0，完全恢复为调用系统调用前的状态：



iret后成功返回asm\_system\_call在int 0x80后一条语句的位置：



成功返回，eax=0，调用栈和父进程一致：



可以认为返回过程和父进程几乎一致，但从上下文的角度也可以说子进程是在copyProcess(parent, child)之后就返回，没有进行接下来的interruptManager.setInterruptStatus(status);和return pid;操作，因此不会返回自己的pid。

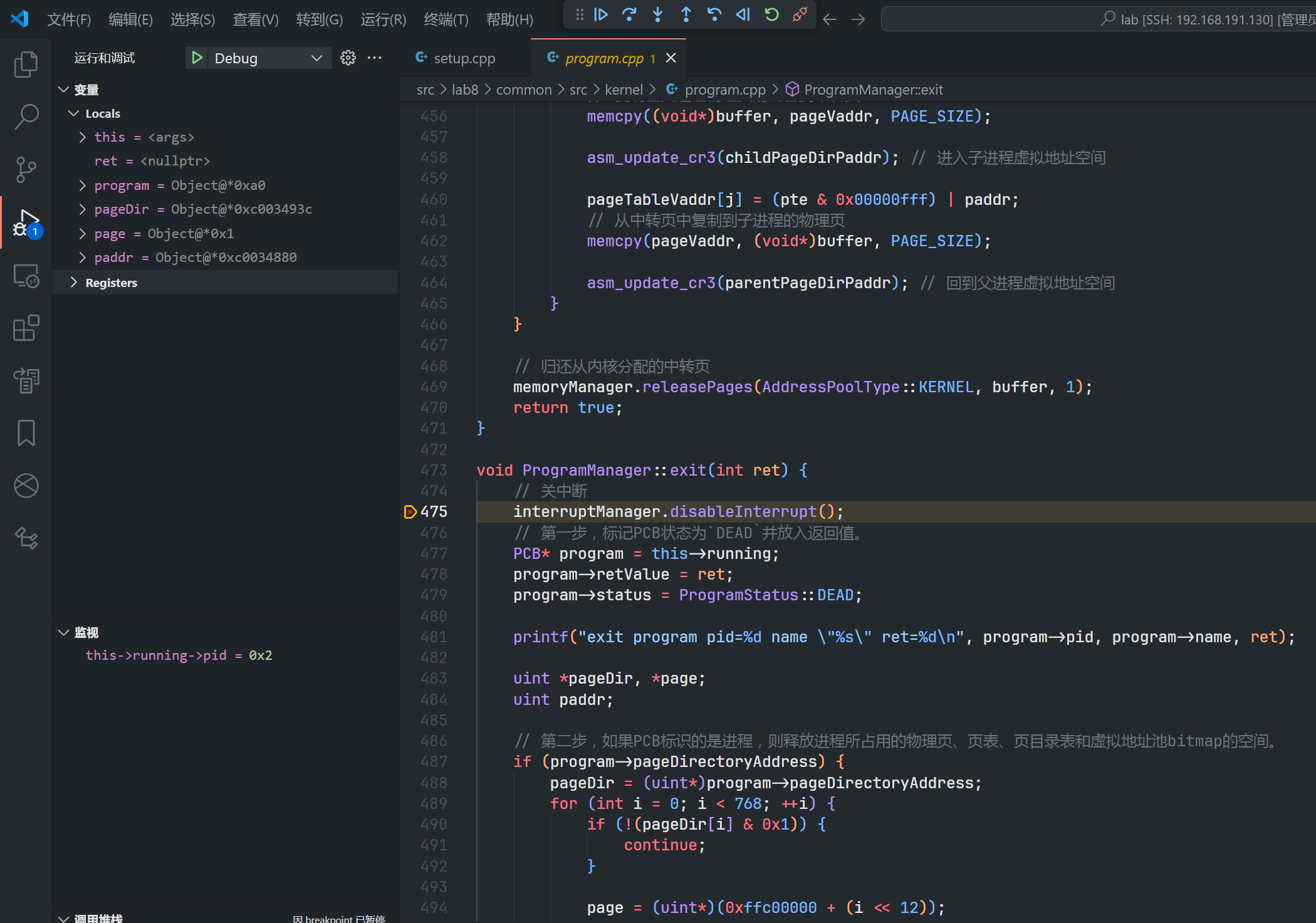
运行结果：



### Assignment 3 哼哈二将 wait & exit

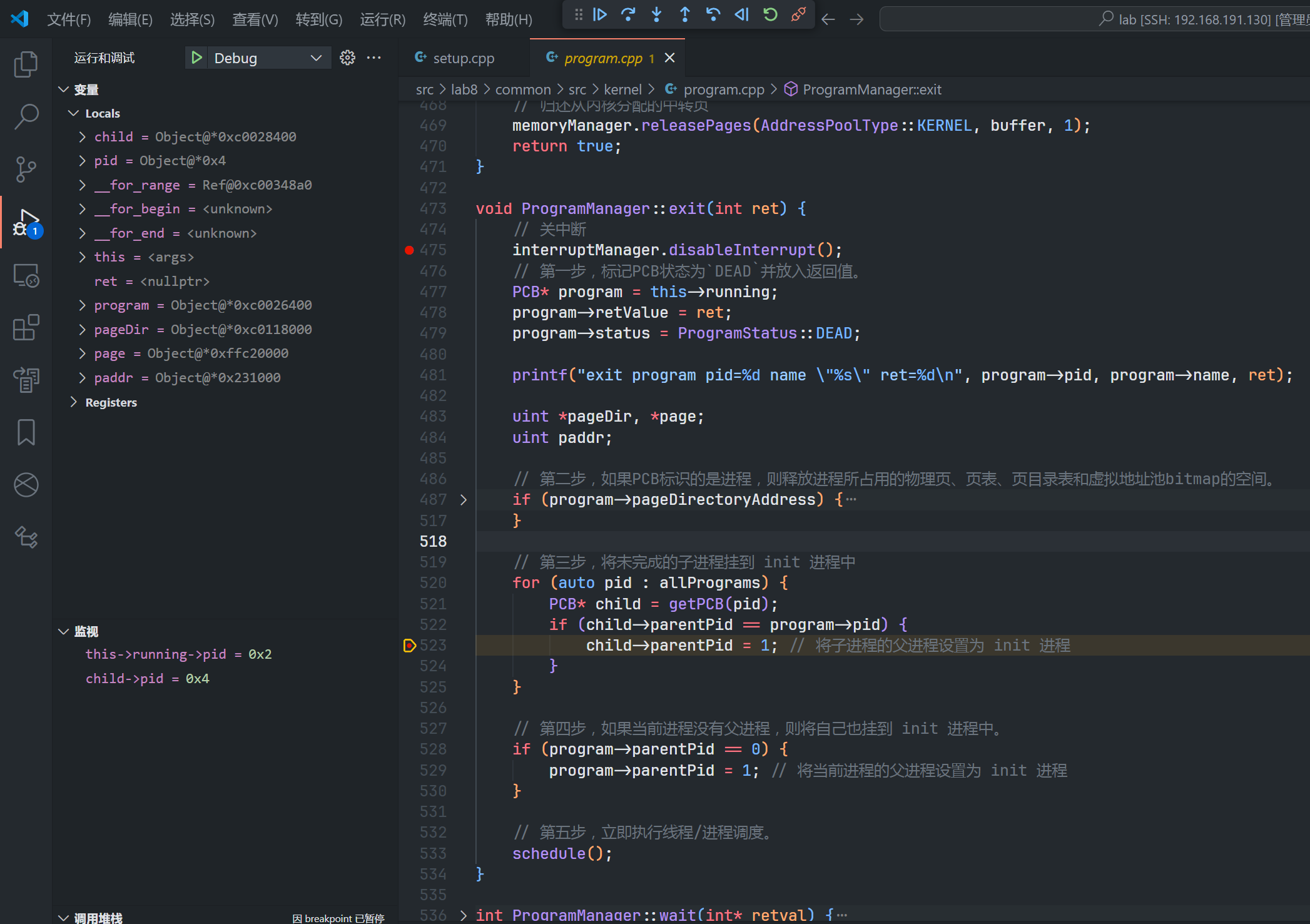
exit和wait的验证比较简单，只需要在对应函数下断然后对关键过程进行分析即可。

首先断在入口点，通过监视可知此时将要exit的是pid=2的进程，分析可知这是first\_process的主进程，它在进行两次fork之后直接退出，留下两个孤儿进程：

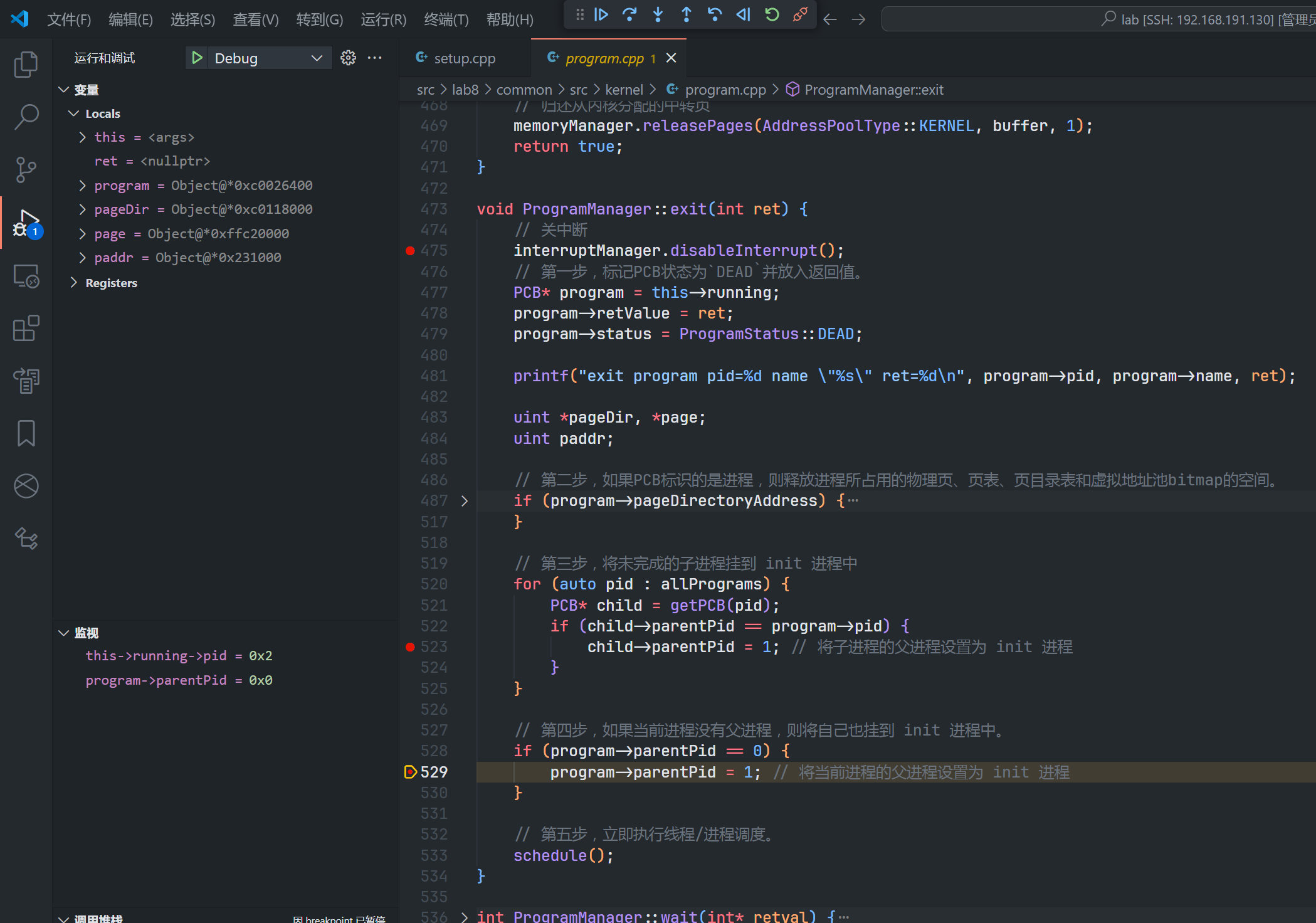


从变量窗口读出ret=0，是正常退出时的默认参数。

第一第二步忽略，来到第三步，现在需要将两个孤儿进程交给init进程管理：

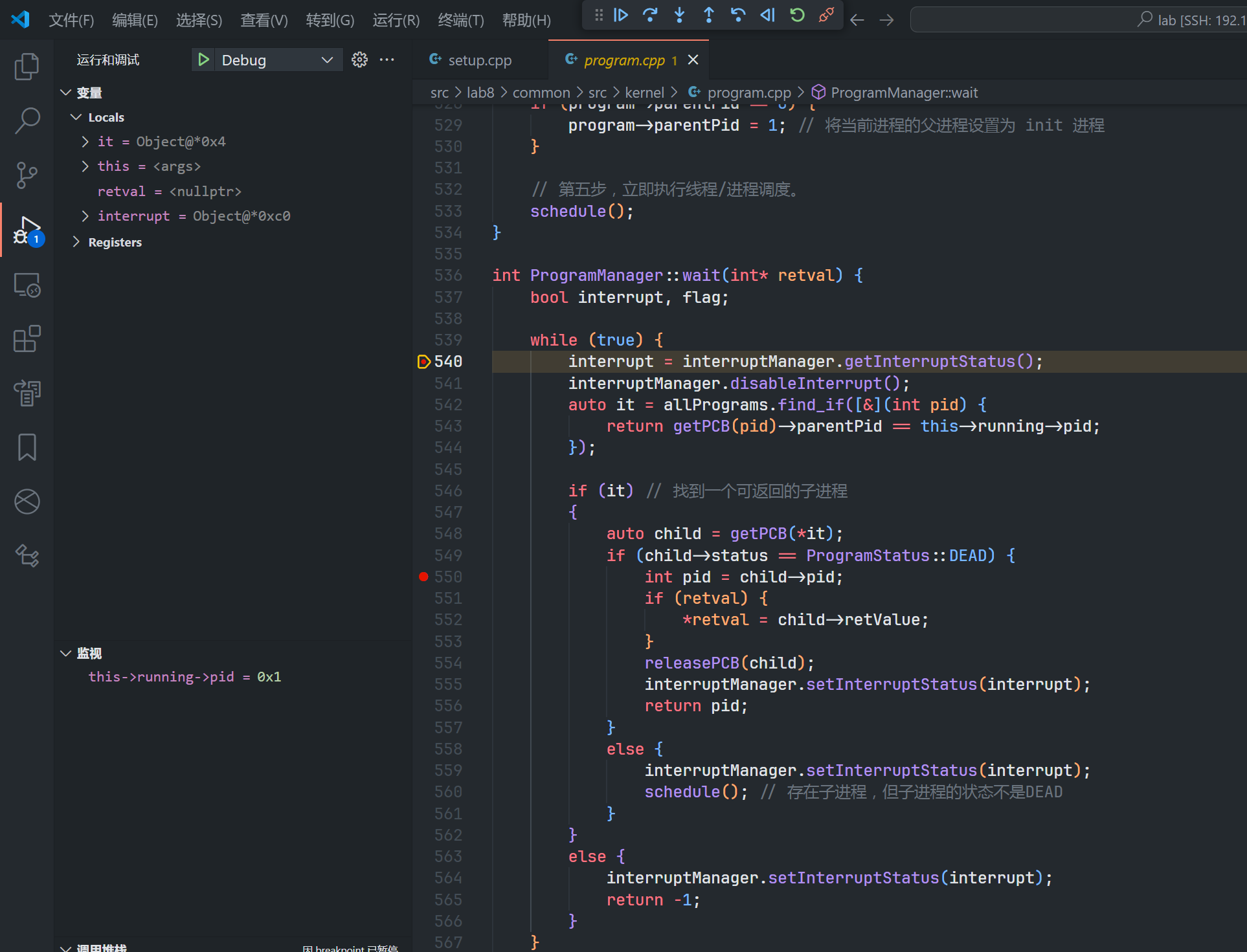


然后发现父进程的父进程为pid=0，也将他挂到init下，方便销毁：



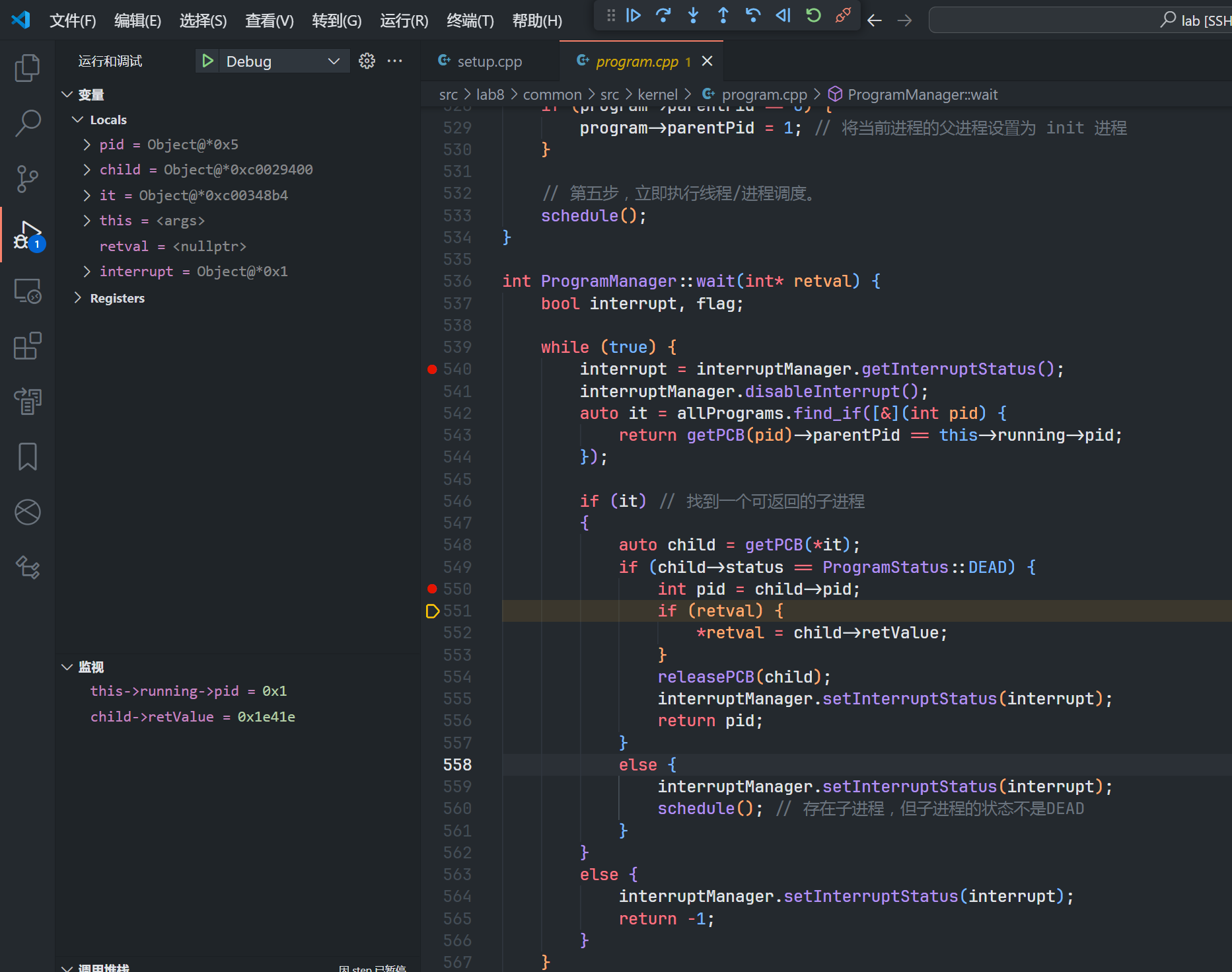
之后调用schedule，exit流程结束。

wait函数的流程也类似：



从running->pid读出当前正在wait的进程pid为1，即init进程。

在找到可返回子进程位置下断，以pid=5的子进程的销毁过程为例：



该子进程的返回值在child->retValue中，读出0x1e41e，即123934，符合预期，该值将被赋给retval。然后调用releasePCB释放PCB，并返回子进程pid。

运行结果：



最终只剩下内核线程和init进程在执行，符合预期。

## 总结

下发的参考代码里memcpy的参数src和dst顺序是反过来的，因为这个调assignment 2调了一晚。