

# 本科生实验报告

实验课程: 操作系统原理实验

实验名称: 内核线程

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4月19日

## 实验要求

### Assignment 1 printf的实现

学习可变参数机制，然后实现printf，你可以在材料中的printf上进行改进，或者从头开始实现自己的printf函数。结果截图并说说你是怎么做的。

### Assignment 2 线程的实现

自行设计PCB，可以添加更多的属性，如优先级等，然后根据你的PCB来实现线程，演示执行结果。

### Assignment 3 线程调度切换的秘密

操作系统的线程能够并发执行的秘密在于我们需要中断线程的执行，保存当前线程的状态，然后调度下一个线程上处理机，最后使被调度上处理机的线程从之前被中断点处恢复执行。现在，同学们可以亲手揭开这个秘密。

编写若干个线程函数，使用gdb跟踪c\_time\_interrupt\_handler、asm\_switch\_thread等函数，观察线程切换前后栈、寄存器、PC等变化，结合gdb、材料中“线程的调度”的内容来跟踪并说明下面两个过程。

* 一个新创建的线程是如何被调度然后开始执行的。
* 一个正在执行的线程是如何被中断然后被换下处理器的，以及换上处理机后又是如何从被中断点开始执行的。

通过上面这个练习，同学们应该能够进一步理解操作系统是如何实现线程的并发执行的。

### Assignment 4 调度算法的实现

在材料中，我们已经学习了如何使用时间片轮转算法来实现线程调度。但线程调度算法不止一种，例如

* 先来先服务。
* 最短作业（进程）优先。
* 响应比最高者优先算法。
* 优先级调度算法。
* 多级反馈队列调度算法。

此外，我们的调度算法还可以是抢占式的。

现在，同学们需要将线程调度算法修改为上面提到的算法或者是同学们自己设计的算法。然后，同学们需要自行编写测试样例来呈现你的算法实现的正确性和基本逻辑。最后，将结果截图并说说你是怎么做的。

## 实验过程

### Assignment 1 printf的实现

printf 的基本框架流程为：

1. 函数接收一个格式化字符串 format 和一个可变参数列表。使用 va\_list 类型来处理可变参数，并通过 va\_start 宏初始化参数列表。
2. 遍历格式化字符串 format。对于每个字符，判断其是否为格式控制符 %
3. 格式控制符处理
   1. 对于整数输出 (%d 和 %x)，分别调用 write\_int 和 write\_hex 函数，将整数转换为字符串并输出。
   2. 对于字符串输出 (%s)，直接获取字符串指针，然后逐字符调用 putchar 函数输出。
   3. 对于字符输出（%c）和输出百分号本身（%%），均直接调用 putchar 输出。
4. 转义字符处理：特别处理了换行符\n和制表符\t。通过调用getCursor获取当前光标位置，并根据换行和制表符的规则移动光标位置，并通过 moveCursor移动光标。同时，如果光标超出屏幕范围，则调用rollUp实现向上滚动屏幕的功能。

write\_int 函数采用递归而非字符缓冲区的方式处理多位数输出，write\_hex 函数则支持十六进制输出。

### Assignment 2 线程的实现

通过ProgramManager类管理线程，负责线程的创建、调度和销毁，主要由以下几部分组成：

1. 其内部成员有：
   1. allPrograms: 存储所有状态的线程/进程
   2. readyPrograms: 存储就绪状态的线程
   3. running: 指向当前执行的线程
2. 创建线程方法，流程为：
   1. 分配空闲PCB
   2. 初始化线程相关信息（名称、优先级等）
   3. 设置线程执行函数和参数
   4. 设置线程栈空间
   5. 将线程加入就绪队列
   6. 返回线程ID (pid)
3. 内部的释放线程方法，流程为：
   1. 通过线程指针反推线程在线程池中的下标位置
   2. 将对应位置的标志位置为空闲
4. 线程调度方法，使用时间片流转，每次调度时如果当前线程还在运行，则将其放回队列末尾，重新从队列头取出一个新的线程执行。

使用8259A时钟中断触发线程调度，达到线程切换的效果。

测试运行：创建两个线程first\_thread和second\_thread

### Assignment 3 线程调度切换的秘密

对Assignment 2的代码进行修改，将两个线程改为了易于追踪中断前位置的代码。

原理上，函数调用之前会将下一条指令的地址压入栈中，这样函数返回后就能找到调用前的地址。通过交换两个线程的栈，就可以在asm\_switch\_thread结束后来到另一个线程的地址空间。

对asm\_switch\_thread、schedule\_time\_interrupt\_handler、programManager.schedule分别下断，观察其执行前后的行为。

### Assignment 4 调度算法的实现

基于RR实现了四个调度算法：FCFS、SJF、PR、MLFQ，实现均为非抢占式。

FCFS只需要将调度函数的逻辑改为只有线程结束才切换任务；SJF和PR的实现几乎完全一致，因为没有任务量估算算法的情况下，执行时间和优先级在非抢占式调度下没什么区别。MLFQ实现了5层队列，为每层队列赋了不同的时间片权值。

## 关键代码

### Assignment 1 printf的实现

两个输出整数的函数：

void STDIO::write\_int(int x) {

    if (x < 0) {

        putchar('-');

        x = -x;

    }

    if (x >= 10) write\_int(x / 10), x %= 10;

    putchar(char(x + '0'));

}

void STDIO::write\_hex(int x) {

    if (x < 0) {

        putchar('-');

        x = -x;

    }

    if (x >= 16) write\_hex(x / 16), x %= 16;

    if (x < 10) putchar(char(x + '0'));

    else putchar(char(x - 10 + 'A'));

}

printf实现：

void printf(const char\* format, ...) {

    va\_list args;

    va\_start(args, format);

    for (const char\* p = format; \*p != '\0'; ++p) {

        if (\*p == '%') {

            ++p;

            if (\*p == 'd') {

                int num = va\_arg(args, int);

                stdio.write\_int(num);

            }

            else if (\*p == 'x') {

                int num = va\_arg(args, int);

                stdio.write\_hex(num);

            }

            else if (\*p == 's') {

                const char\* str = va\_arg(args, const char\*);

                while (\*str != '\0') {

                    stdio.putchar(\*str++);

                }

            }

            else if (\*p == 'c') {

                char ch = (char)va\_arg(args, int);

                stdio.putchar(ch);

            }

            else if (\*p == '%') {

                stdio.putchar('%');

            }

        }

        else if (\*p == '\n') { // 换行

            uint cursor = stdio.getCursor();

            cursor += 80 - (cursor % 80);

            if (cursor >= 25 \* 80) {

                stdio.rollUp();

                cursor = 24 \* 80;

            }

            stdio.moveCursor(cursor);

        }

        else if (\*p == '\t') { // 制表符

            uint cursor = stdio.getCursor();

            cursor += 4 - (cursor % 4);

            if (cursor >= 25 \* 80) {

                stdio.rollUp();

                cursor = 24 \* 80;

            }

            stdio.moveCursor(cursor);

        }

        else {

            stdio.putchar(\*p);

        }

    }

}

### Assignment 2 线程的实现

PCB定义：

struct PCB {

    int\* stack;                      // 栈指针，用于调度时保存esp

    char name[MAX\_PROGRAM\_NAME + 1]; // 线程名

    ProgramStatus status;            // 线程的状态

    int priority;                    // 线程优先级

    int pid;                         // 线程pid

    int ticks;                       // 线程时间片总时间

    int ticksPassedBy;               // 线程已执行时间

};

基础设施，用于PCB管理：

const int PCB\_SIZE = 4096;                      // PCB的大小，4KB。

char PCB\_buffer[PCB\_SIZE \* MAX\_PROGRAM\_AMOUNT]; // 存放PCB的数组，预留了MAX\_PROGRAM\_AMOUNT个PCB的大小空间。

bool PCB\_used[MAX\_PROGRAM\_AMOUNT];              // PCB的分配状态，true表示已经分配，false表示未分配。

int PCB\_free\_list[MAX\_PROGRAM\_AMOUNT];          // PCB的空闲链表，存放未分配的PCB的索引。

int PCB\_free\_count;

int getPCBIndex(PCB\* program) {

    return ((int)program - (int)PCB\_buffer) / PCB\_SIZE;

}

PCB\* getPCB(int pid) {

    PCB\* program = (PCB\*)(PCB\_buffer + pid \* PCB\_SIZE);

    return program;

}

PCB\* ProgramManager::allocatePCB() {

    if (PCB\_free\_count == 0) {

        return nullptr;

    }

    int index = PCB\_free\_list[--PCB\_free\_count];

    PCB\_used[index] = true;

    return (PCB\*)(PCB\_buffer + index \* PCB\_SIZE);

}

void ProgramManager::releasePCB(PCB\* program) {

    int index = getPCBIndex(program);

    PCB\_used[index] = false;

    PCB\_free\_list[PCB\_free\_count++] = index;

}

创建线程：

int ProgramManager::executeThread

(ThreadFunction function, void\* parameter, const char\* name, int priority) {

    // 关中断，防止创建线程的过程被打断

    bool status = interruptManager.getInterruptStatus();

    interruptManager.disableInterrupt();

    // 分配一页作为PCB

    PCB\* thread = allocatePCB();

    if (!thread)

        return -1;

    // 初始化分配的页

    memset(thread, 0, PCB\_SIZE);

    for (int i = 0; i < MAX\_PROGRAM\_NAME && name[i]; ++i) {

        thread->name[i] = name[i];

    }

    thread->status = ProgramStatus::READY;

    thread->priority = priority;

    thread->ticks = priority;

    thread->ticksPassedBy = 0;

    thread->pid = getPCBIndex(thread); // 线程的pid为其在PCB\_buffer中的索引

    // 线程栈

    thread->stack = (int\*)((int)thread + PCB\_SIZE);

    thread->stack -= 7;                   // 栈顶从高地址向低地址移动7x4字节，存放

    thread->stack[0] = 0;                 // 保存esi

    thread->stack[1] = 0;                 // 保存edi

    thread->stack[2] = 0;                 // 保存ebx

    thread->stack[3] = 0;                 // 保存ebp

    thread->stack[4] = (int)function;     // 保存线程函数地址

    thread->stack[5] = (int)program\_exit; // 保存线程退出函数地址

    thread->stack[6] = (int)parameter;    // 保存线程函数参数

    allPrograms.push(thread->pid);   // 将线程加入所有线程列表

    readyPrograms.push(thread->pid); // 将线程加入就绪线程列表

    // 恢复中断

    interruptManager.setInterruptStatus(status);

    return thread->pid;

}

线程调度：

void ProgramManager::schedule() {

    bool status = interruptManager.getInterruptStatus();

    interruptManager.disableInterrupt();

    if (readyPrograms.size() == 0) {

        interruptManager.setInterruptStatus(status);

        return;

    }

    // printf("schedule\n");

    // printf("pid %d name \"%s\"\n", running->pid, running->name);

    if (running->status == ProgramStatus::RUNNING) {

        running->status = ProgramStatus::READY;

        running->ticks = running->priority;

        readyPrograms.push(running->pid);

    }

    else if (running->status == ProgramStatus::DEAD) {

        releasePCB(running);

    }

    int item = readyPrograms.front();

    PCB\* next = getPCB(item);

    PCB\* cur = running;

    next->status = ProgramStatus::RUNNING;

    running = next;

    readyPrograms.pop();

    asm\_switch\_thread(cur, next);

    interruptManager.setInterruptStatus(status);

}

两个测试线程工作函数，作用是不断输出同一个字符串：

void first\_thread(void\* arg) {

    char\* msg = (char\*)arg;

    while (true) {

        printf("from %s: %s\n", programManager.running->name, msg);

    }

}

void second\_thread(void\* arg) {

    char\* msg = (char\*)arg;

    while (true) {

        printf("from %s: %s\n", programManager.running->name, msg);

    }

}

创建线程：

char first\_thread\_para[] = "Hello World 1!";

    char second\_thread\_para[] = "Hello World 2!";

    int pid1 = programManager.executeThread(first\_thread, first\_thread\_para, "first thread", 1);

    int pid2 = programManager.executeThread(second\_thread, second\_thread\_para, "second thread", 1);

    if (pid1 == -1 || pid2 == -1) {

        printf("can not execute thread\n");

        asm\_halt();

    }

    int item = programManager.readyPrograms.front();

    PCB\* firstThread = getPCB(item);

    firstThread->status = ProgramStatus::RUNNING;

    programManager.readyPrograms.pop();

    programManager.running = firstThread;

    asm\_switch\_thread(nullptr, firstThread);

### Assignment 3 线程调度切换的秘密

两个调度函数：

void first\_thread(void\*) {

    for (int i = 0;; i++) {

        printf("from %s: %d, %d \n", programManager.running->name, i, 1);

        for (int j = 0; j < 100000000; j++) {}

        printf("from %s: %d, %d \n", programManager.running->name, i, 2);

        for (int j = 0; j < 100000000; j++) {}

        printf("from %s: %d, %d \n", programManager.running->name, i, 3);

        for (int j = 0; j < 100000000; j++) {}

    }

}

void second\_thread(void\*) {

    asm\_halt();

}

### Assignment 4 调度算法的实现

为了尽可能做到代码复用，将ProgramManager部分方法移到了全局。

使用宏指向真正的ProgramManager：

#include "rr.hpp"

#include "fcfs.hpp"

#include "sjf.hpp"

#include "ps.hpp"

#include "mlfq.hpp"

#define ProgramManager MLFQProgramManager

先来先服务调度，只在线程结束才切换：

void FCFSProgramManager::schedule() {

    bool status = interruptManager.getInterruptStatus();

    interruptManager.disableInterrupt();

    if (readyPrograms.size() == 0) {

        interruptManager.setInterruptStatus(status);

        return;

    }

    if (running->status == ProgramStatus::DEAD) {

        releasePCB(running);

        int item = readyPrograms.front();

        readyPrograms.pop();

        PCB\* next = getPCB(item);

        PCB\* cur = running;

        next->status = ProgramStatus::RUNNING;

        running = next;

        asm\_switch\_thread(cur, next);

    }

    interruptManager.setInterruptStatus(status);

}

非抢占式最短作业优先调度，为了方便直接用priority作为时长：

void SJFProgramManager::schedule() {

    bool status = interruptManager.getInterruptStatus();

    interruptManager.disableInterrupt();

    if (readyCount == 0) {

        interruptManager.setInterruptStatus(status);

        return;

    }

    if (!running || running->status == ProgramStatus::DEAD) {

        if (running) releasePCB(running);

        int best = -1;

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

            PCB\* program = getPCB(readyPrograms[i]);

            if (best == -1 || program->priority < getPCB(best)->priority) {

                best = readyPrograms[i];

            }

        }

        if (best != -1) {

            readyPrograms[best] = readyPrograms[--readyCount];

            PCB\* next = getPCB(best);

            PCB\* cur = running;

            next->status = ProgramStatus::RUNNING;

            running = next;

            asm\_switch\_thread(cur, next);

        }

    }

    interruptManager.setInterruptStatus(status);

}

优先级调度，和SJF几乎一致：

void PSProgramManager::schedule() {

    bool status = interruptManager.getInterruptStatus();

    interruptManager.disableInterrupt();

    if (readyCount == 0) {

        interruptManager.setInterruptStatus(status);

        return;

    }

    if (!running || running->status == ProgramStatus::DEAD) {

        if (running) releasePCB(running);

        int best = -1;

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

            PCB\* program = getPCB(readyPrograms[i]);

            if (best == -1 || program->priority > getPCB(best)->priority) {

                best = readyPrograms[i];

            }

        }

        if (best != -1) {

            readyPrograms[best] = readyPrograms[--readyCount];

            PCB\* next = getPCB(best);

            PCB\* cur = running;

            next->status = ProgramStatus::RUNNING;

            running = next;

            asm\_switch\_thread(cur, next);

        }

    }

    interruptManager.setInterruptStatus(status);

}

非抢占式多级反馈队列调度算法：

定义五层队列及其时间片权重：

class MLFQProgramManager {

public:

    FlatQueue<int, MAX\_PROGRAM\_AMOUNT> readyPrograms[5]; // 就绪队列

    static constexpr int weight[5] = { 1, 2, 4, 8, 16 }; // 各个优先级的权重

    PCB\* running;                                        // 当前执行的线程

    int running\_layer;                                   // 当前执行的线程所在的层级

调度算法：

void MLFQProgramManager::schedule() {

    bool status = interruptManager.getInterruptStatus();

    interruptManager.disableInterrupt();

    if (running && running->status == ProgramStatus::RUNNING) {

        int next\_layer = std::min(4, running\_layer + 1);

        running->status = ProgramStatus::READY;

        running->ticks = running->priority \* weight[next\_layer];

        readyPrograms[next\_layer].push(running->pid);

    }

    else if (running && running->status == ProgramStatus::DEAD) {

        releasePCB(running);

    }

    for (int i = 0; i < 5; i++) {

        if (readyPrograms[i].empty()) continue;

        int item = readyPrograms[i].front();

        PCB\* next = getPCB(item);

        PCB\* cur = running;

        next->status = ProgramStatus::RUNNING;

        running = next;

        printf("switch from %s(layer %d) to %s(layer %d)\n", cur->name, running\_layer, next->name, i);

        running\_layer = i;

        readyPrograms[i].pop();

        asm\_switch\_thread(cur, next);

        break;

    }

    interruptManager.setInterruptStatus(status);

}

测试线程，一个在有限时间内完成，一个无限循环：

void first\_thread(void\*) {

    for (int i = 0; i < 1000000000; i++) {}

}

void second\_thread(void\*) {

    asm\_halt();

}

修改 schedule\_time\_interrupt\_handler以记录线程切换：

void schedule\_time\_interrupt\_handler() {

    PCB\* cur = programManager.running;

    if (!cur) return;

    ++cur->ticksPassedBy;

    if (--cur->ticks <= 0) {

        printf("before schedule: %s\n", programManager.running->name);

        programManager.schedule();

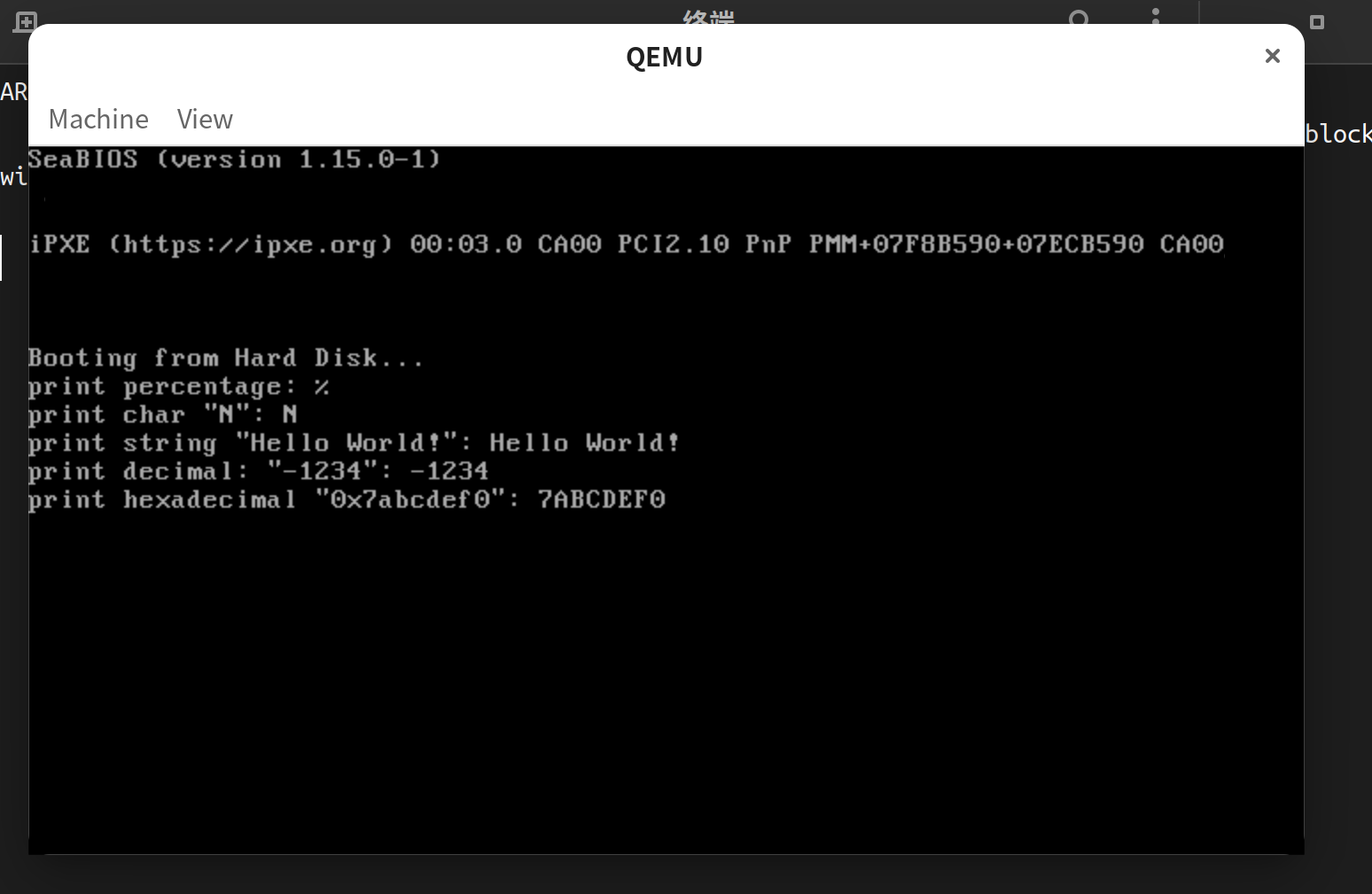
        printf("after schedule: %s\n", programManager.running->name);

    }

}

## 实验结果

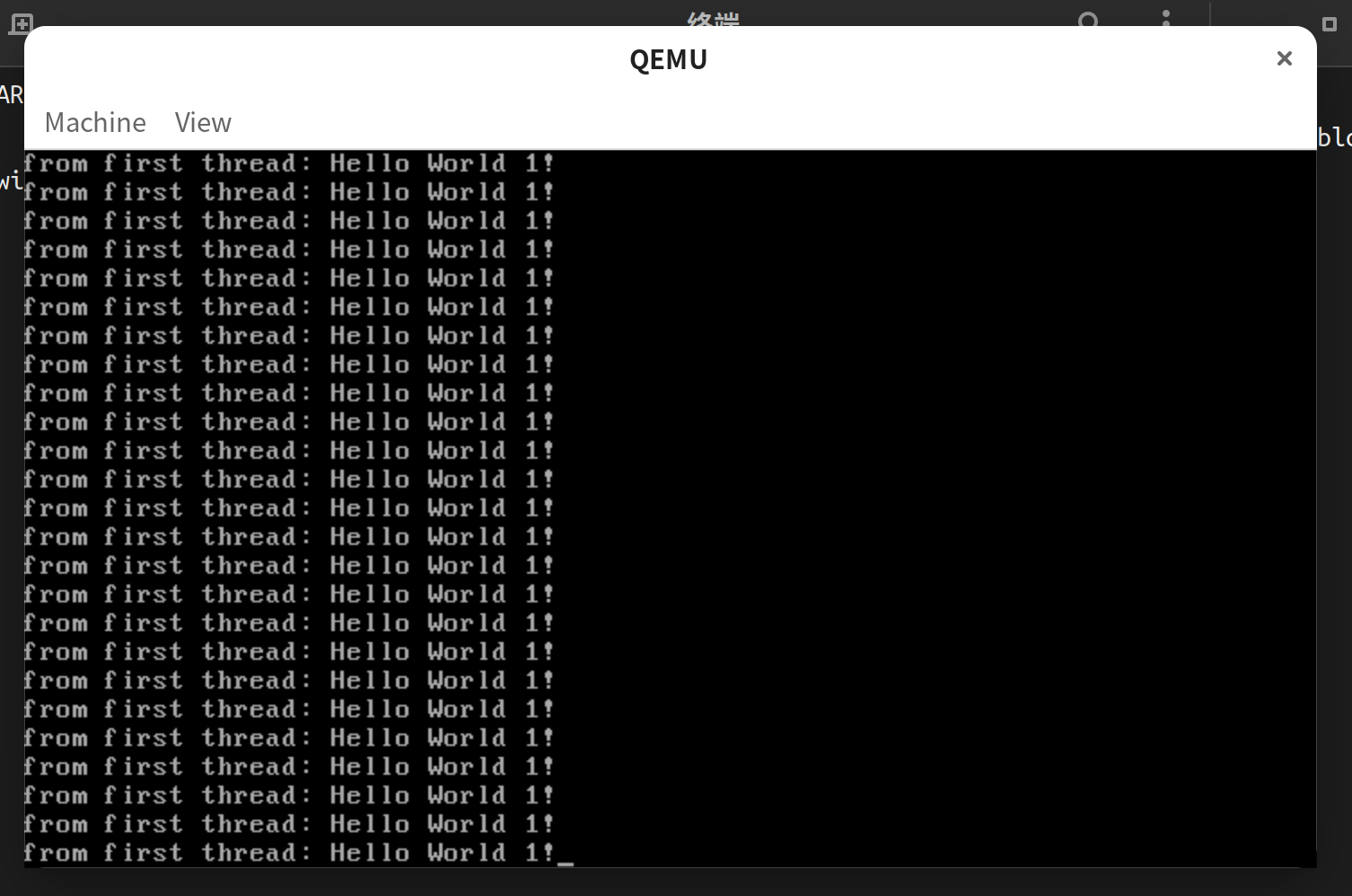
### Assignment 1 printf的实现

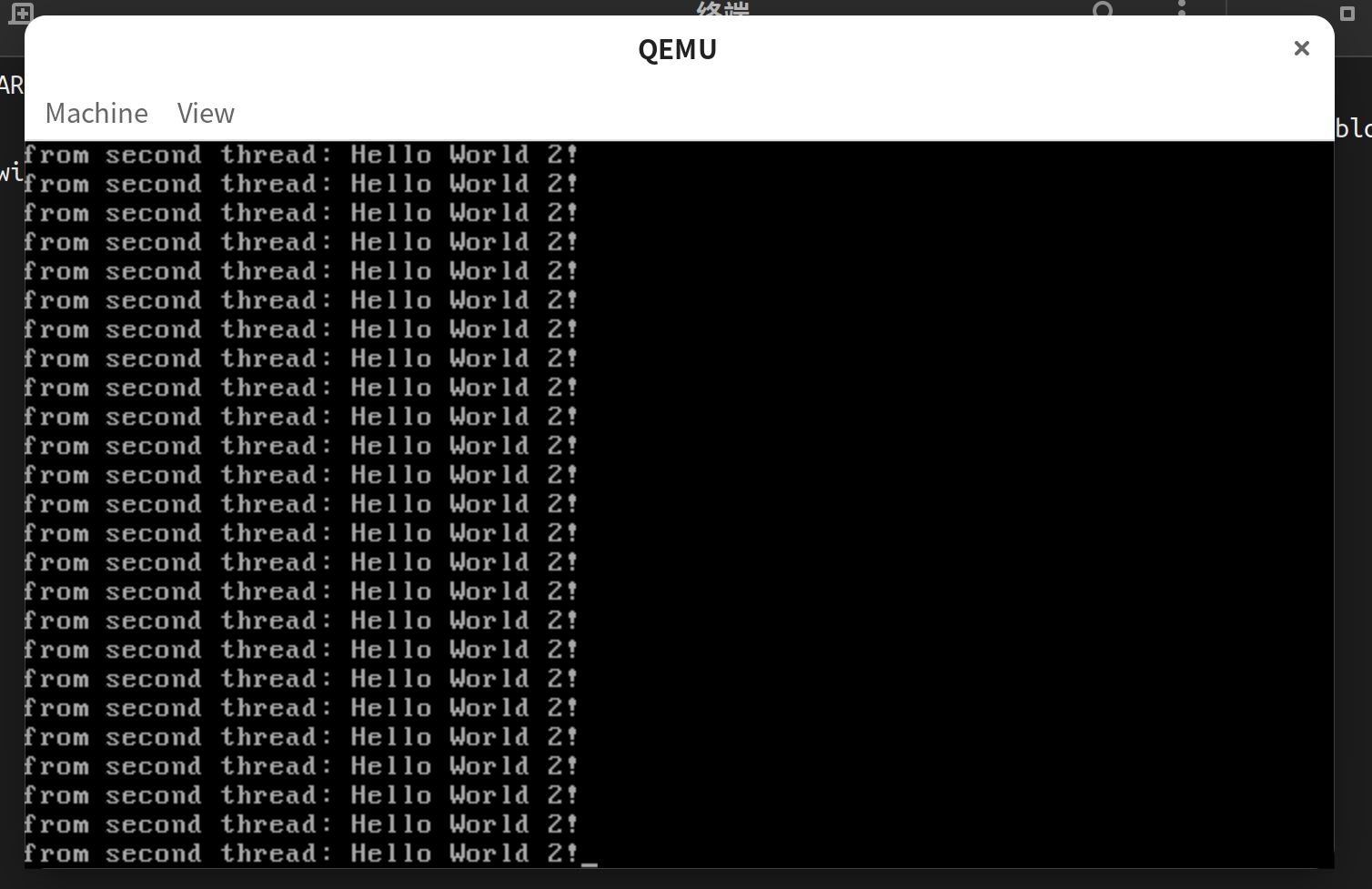


### Assignment 2 线程的实现

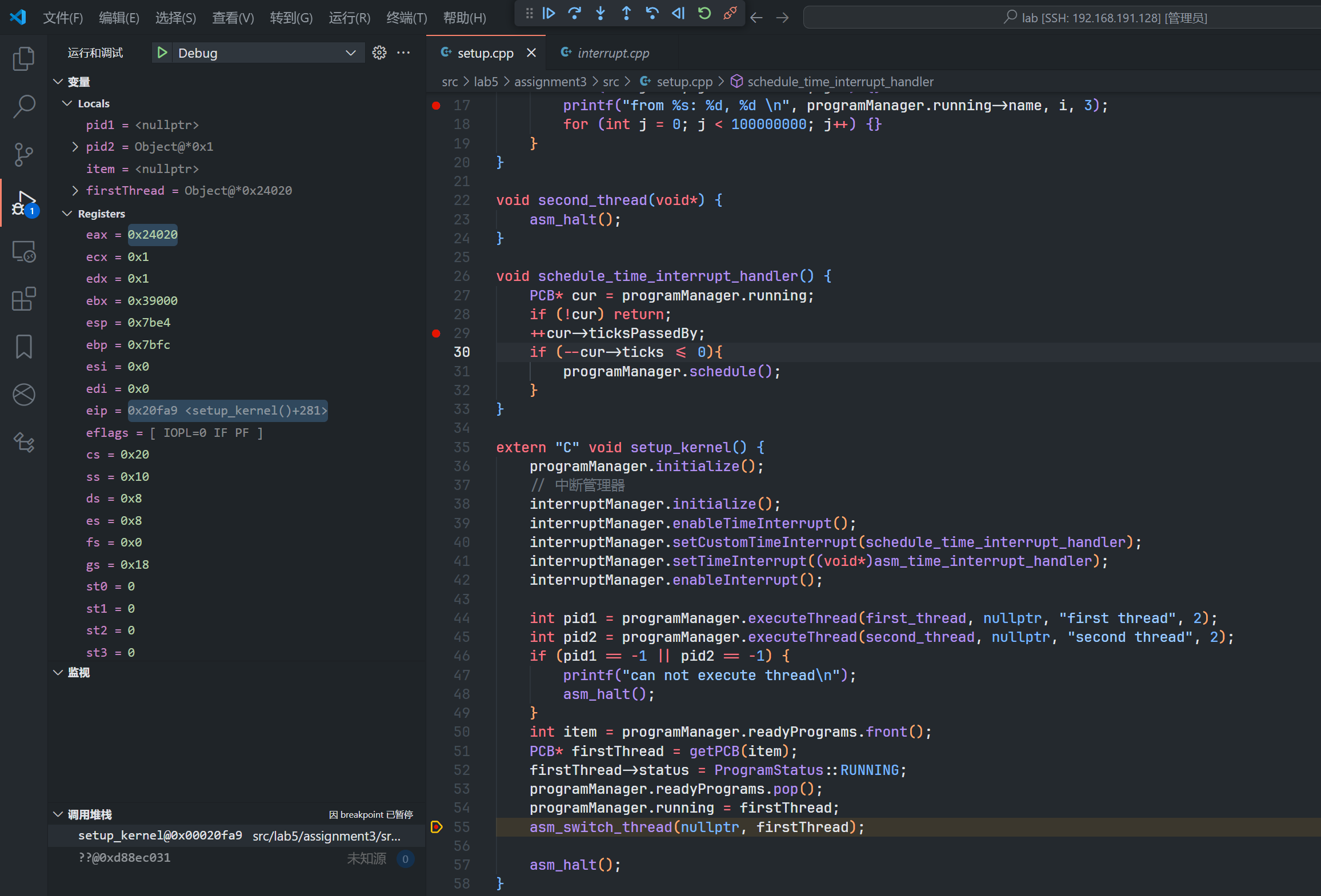
假设能看见动图 。

效果是每个线程分别轮流占据一定长度的时间片，在时间片内输出自己的字符串。

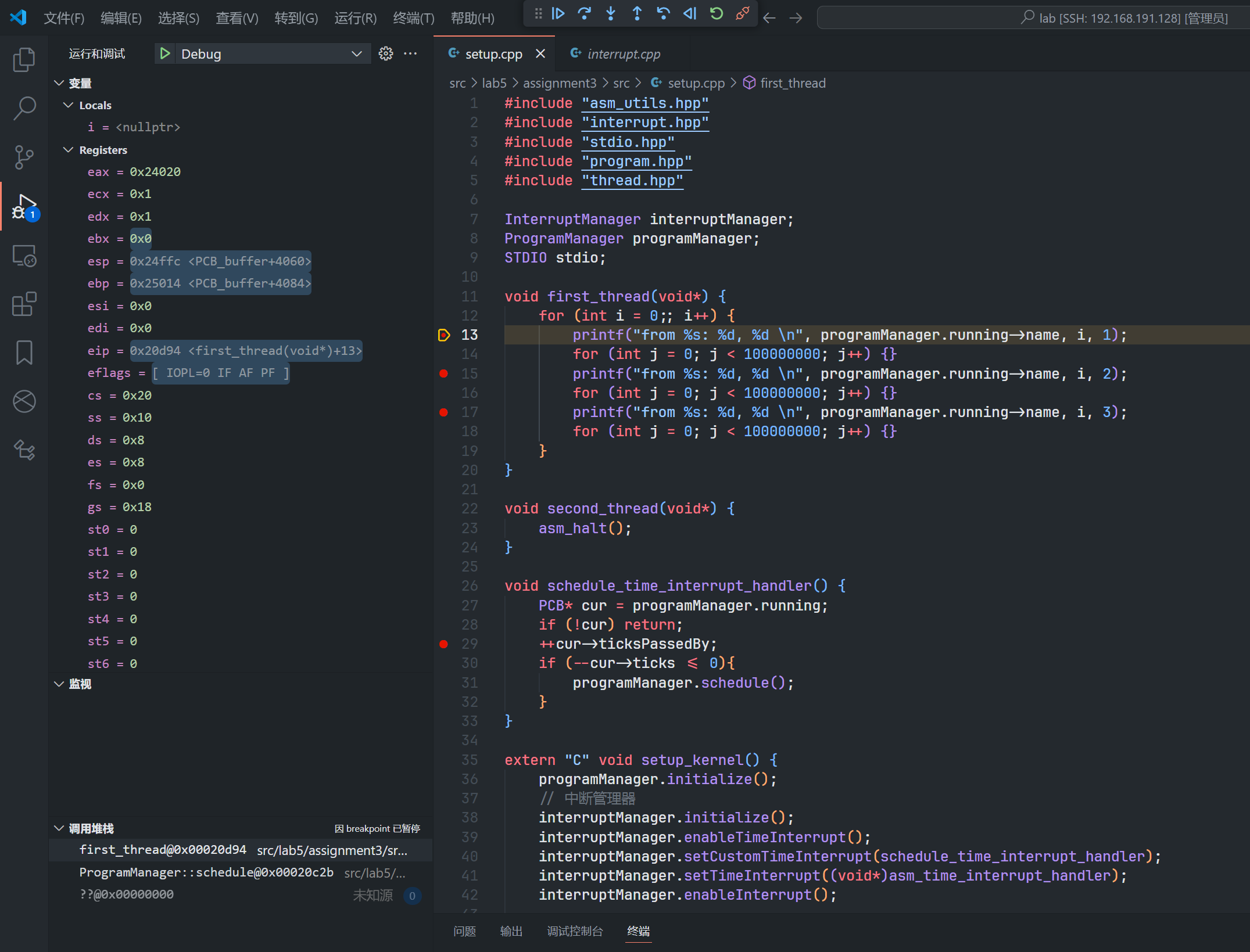




### Assignment 3 线程调度切换的秘密

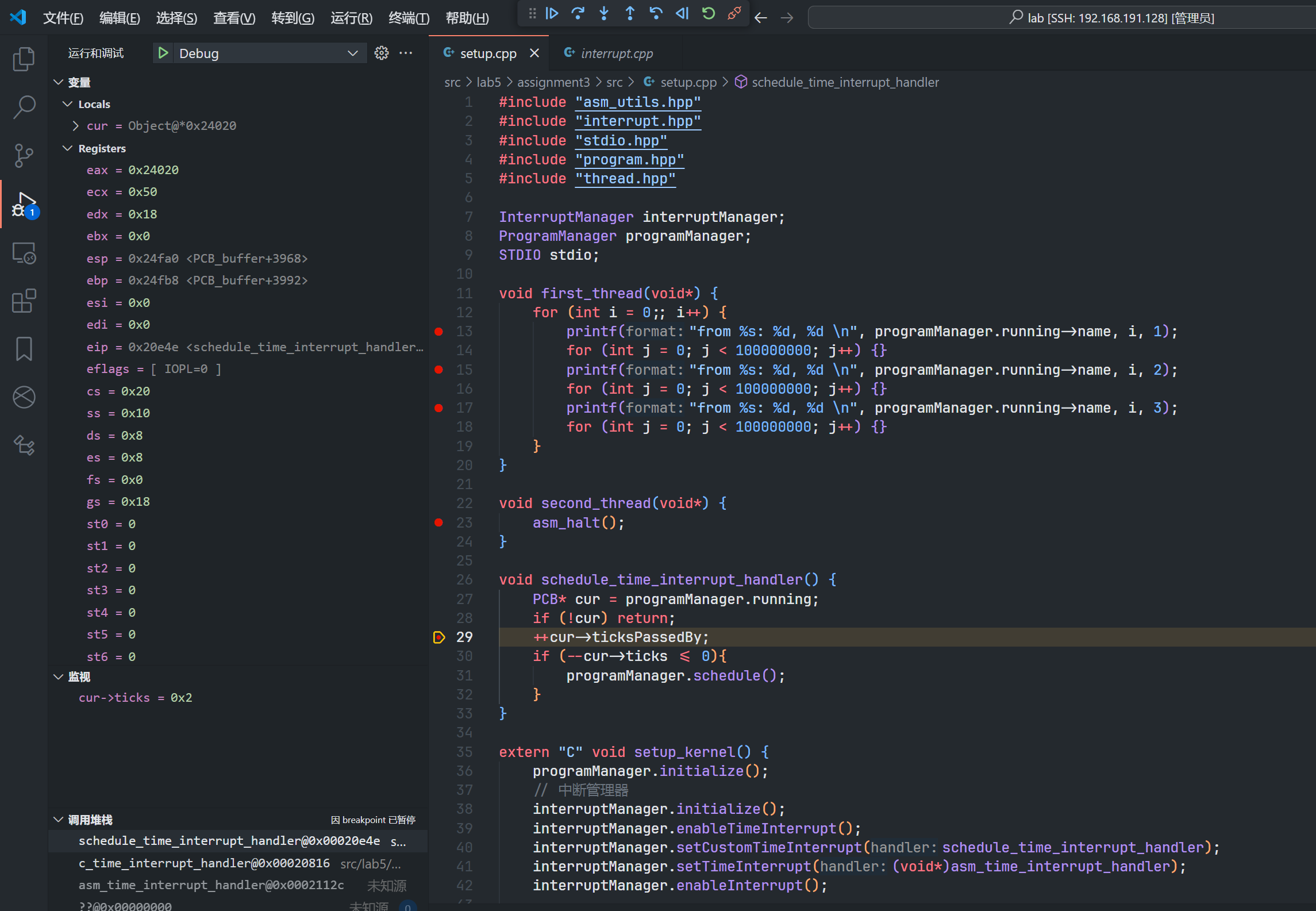


在setup\_kernel中的asm\_switch\_thread下断，可见从队列取出的线程pid（item）为0，即pid1的值。



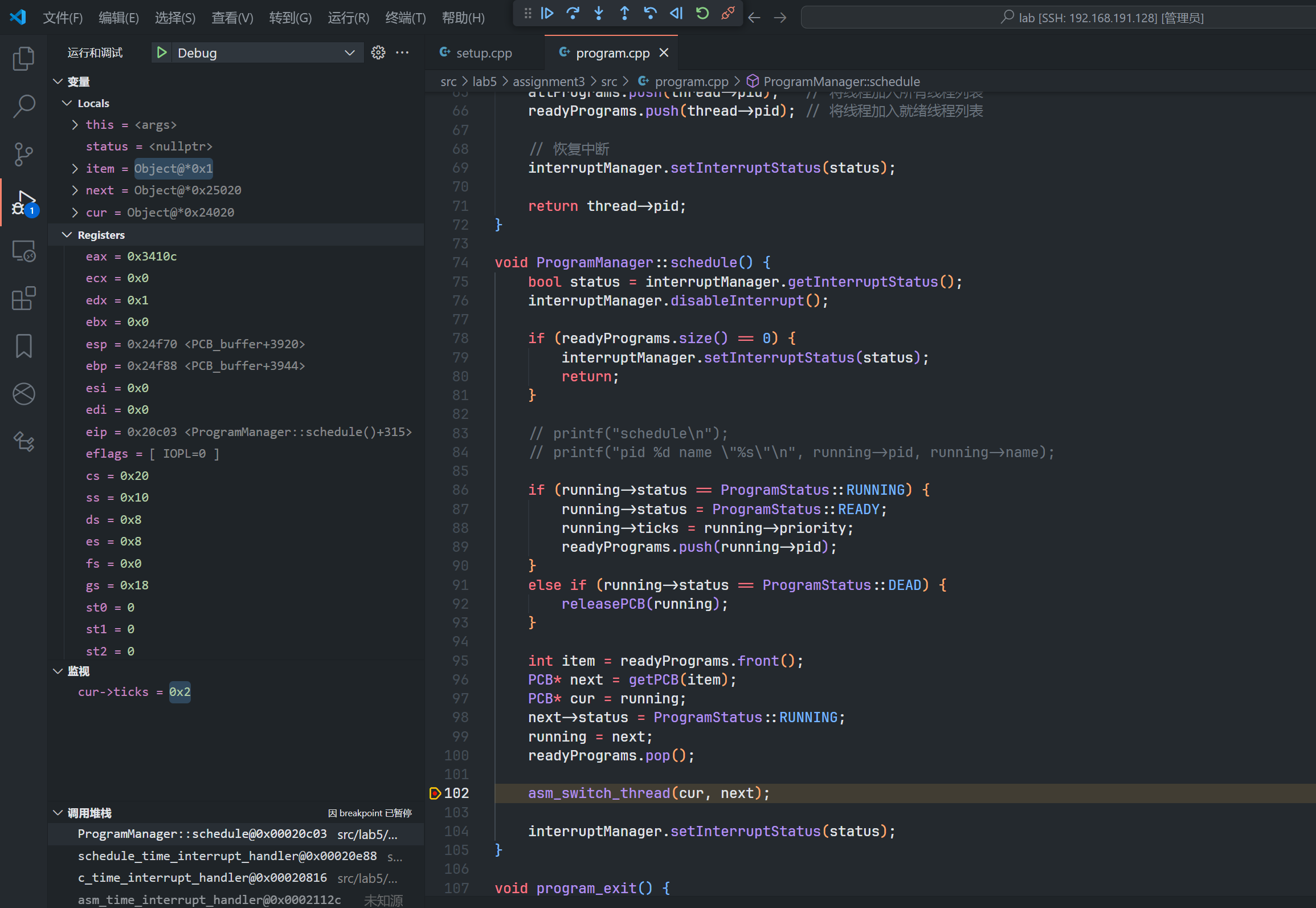
继续执行，程序如预期在first\_thread断下。

若干次在first\_thread断下后，时钟中断触发：



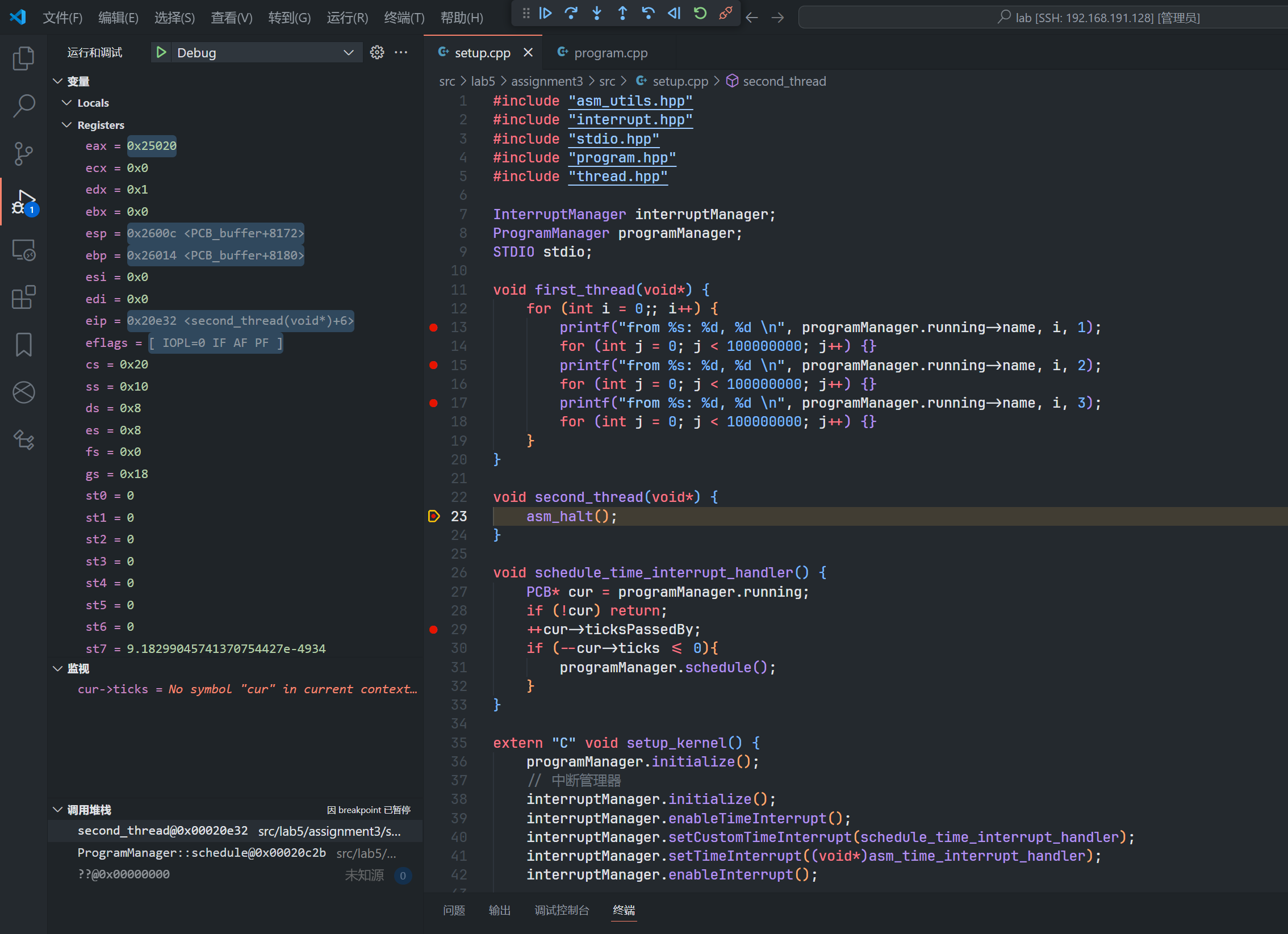
此时cur->ticks等于2，因此不会触发线程调度，返回到原来的位置继续执行。

第二次进入schedule\_time\_interrupt\_handler后，调用program.schedule()：



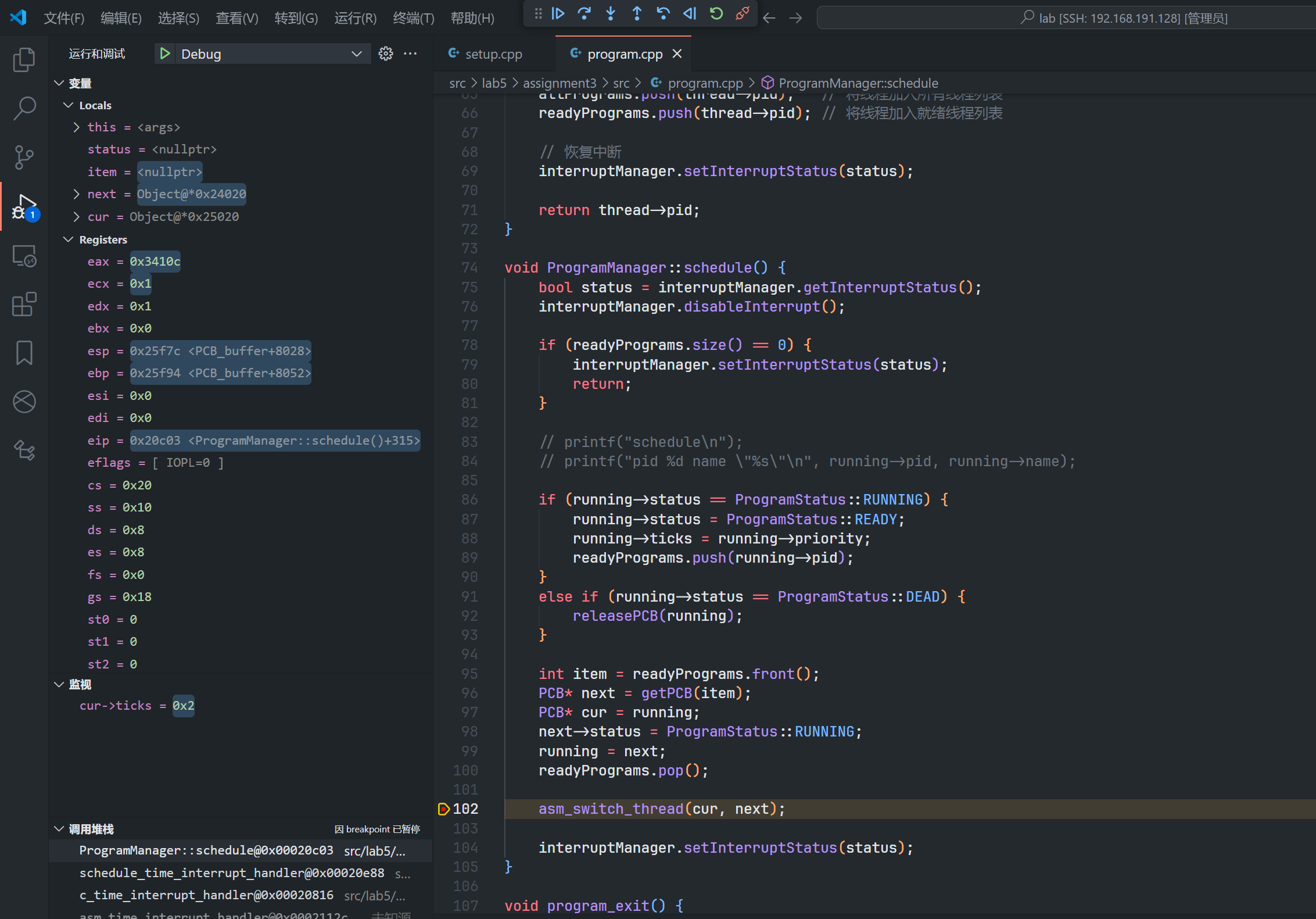
由变量窗口可知此时从队列里取出的线程pid为1，即second\_thread。

继续执行，线程断在second\_thread的asm\_halt上：

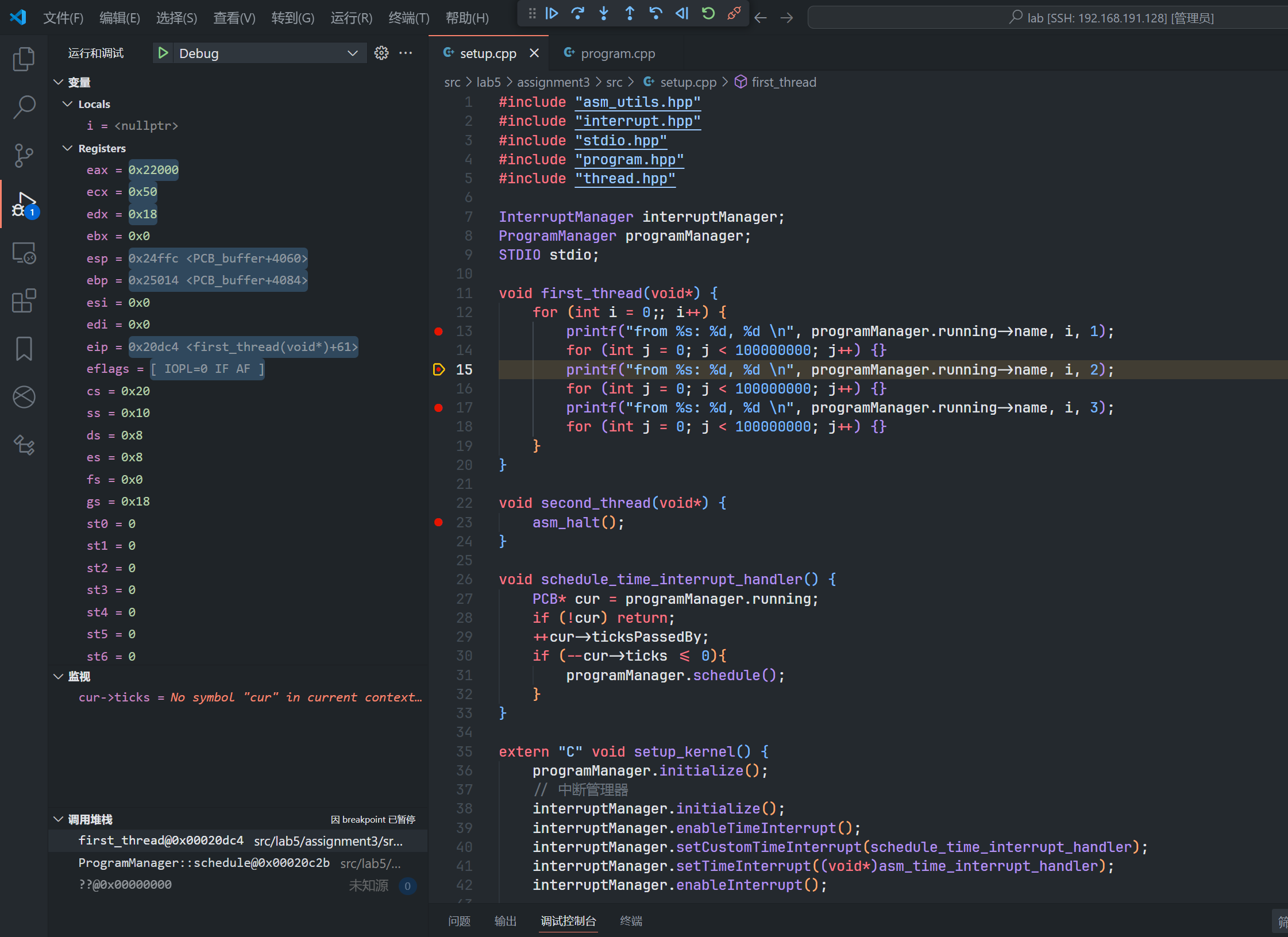


对比两个函数的ebp，恰好相差0x1000=4096，即一个PCB的大小。

继续执行，在若干次断点后在此调用program.schedule()：



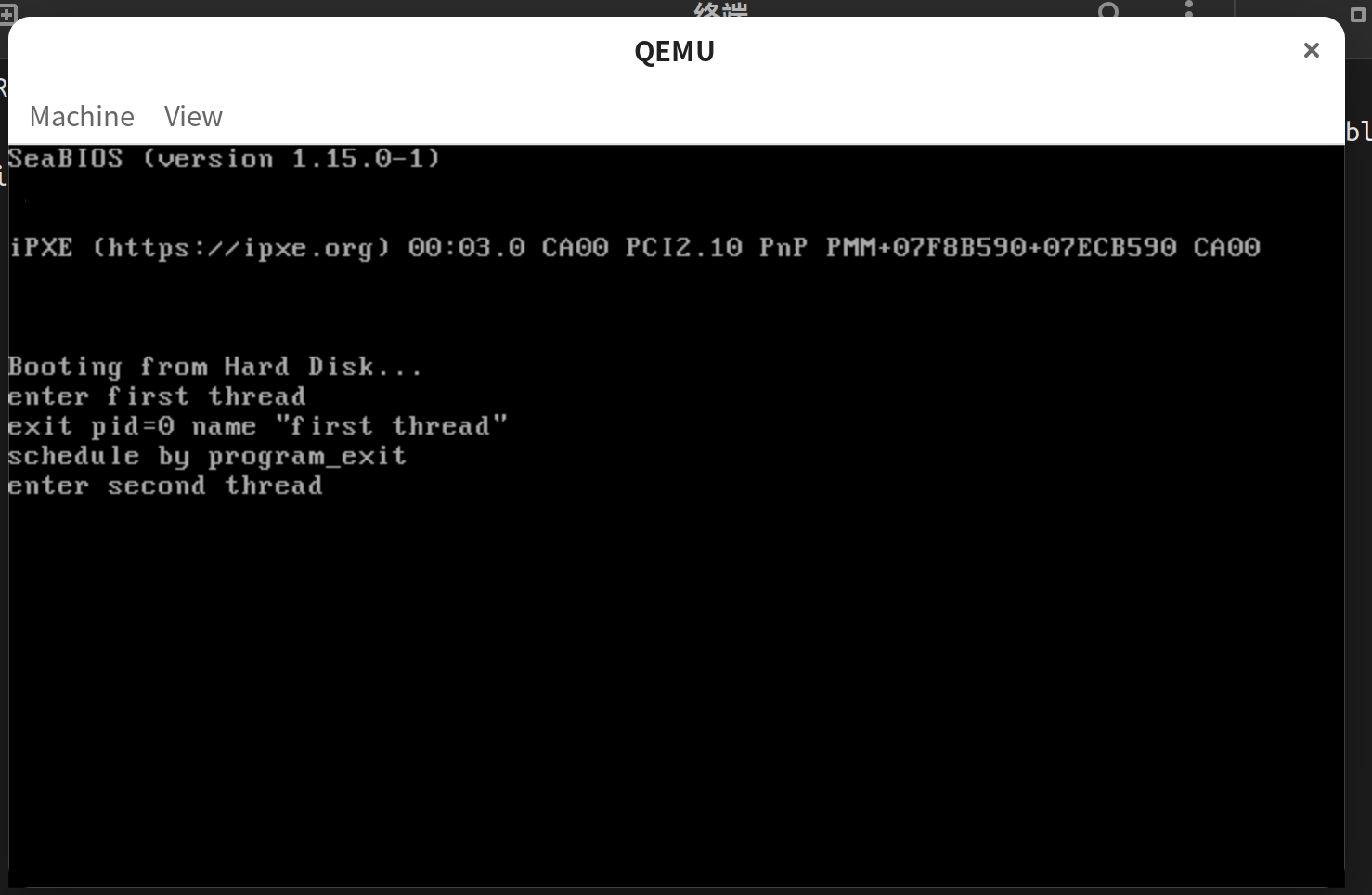
此时取出的item为0，即pid1的值。



成功回到first\_thread的下一条指令。

### Assignment 4 调度算法的实现

FCFS：



线程执行的过程没有被打断，直到线程结束在program\_exit被释放并调度下一个线程。

SJF：将thread2放到上面，并将pid1优先级（实际上是时间）改为10：

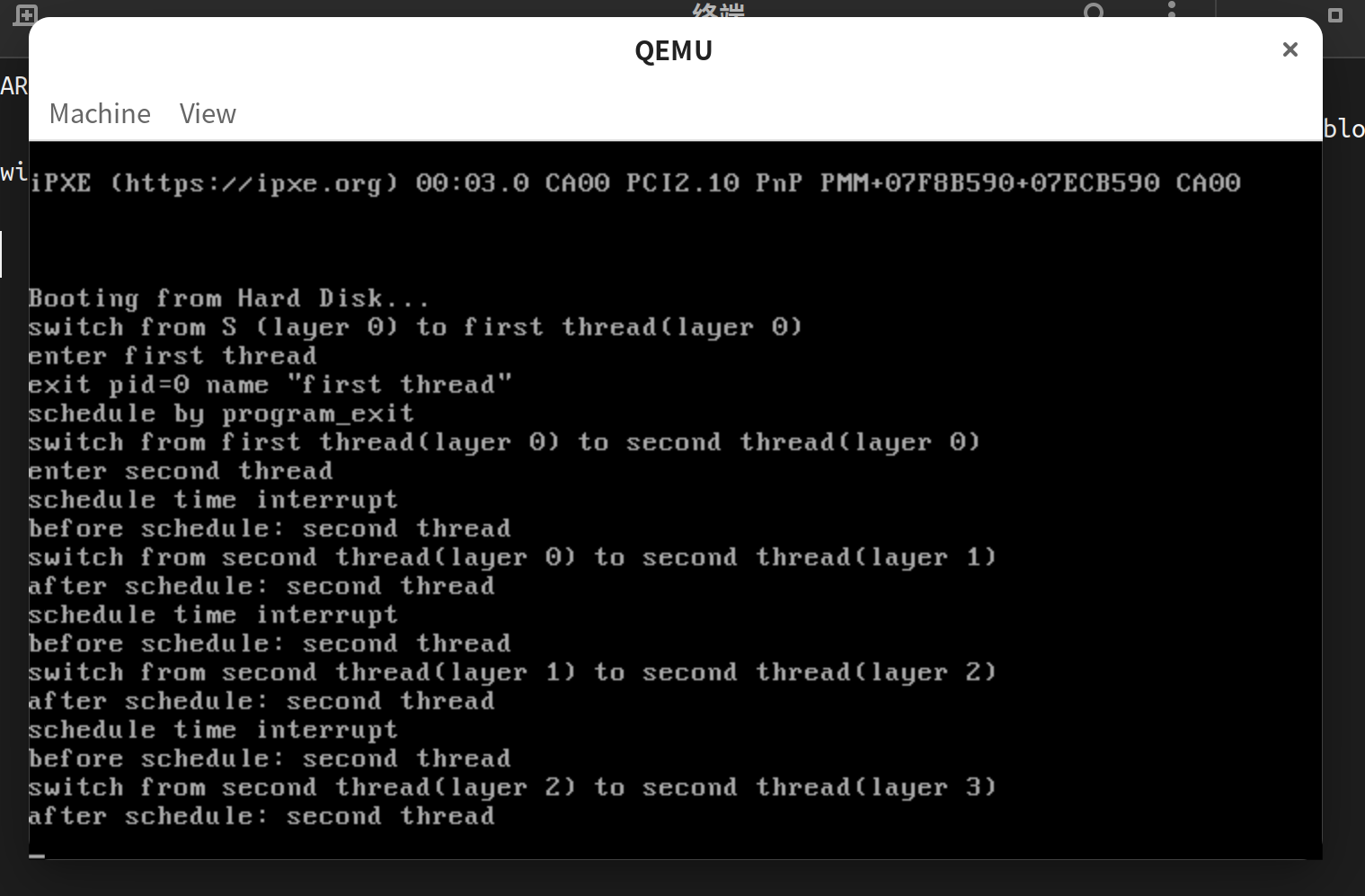
int pid2 = programManager.executeThread(second\_thread, nullptr, "second thread", 1);

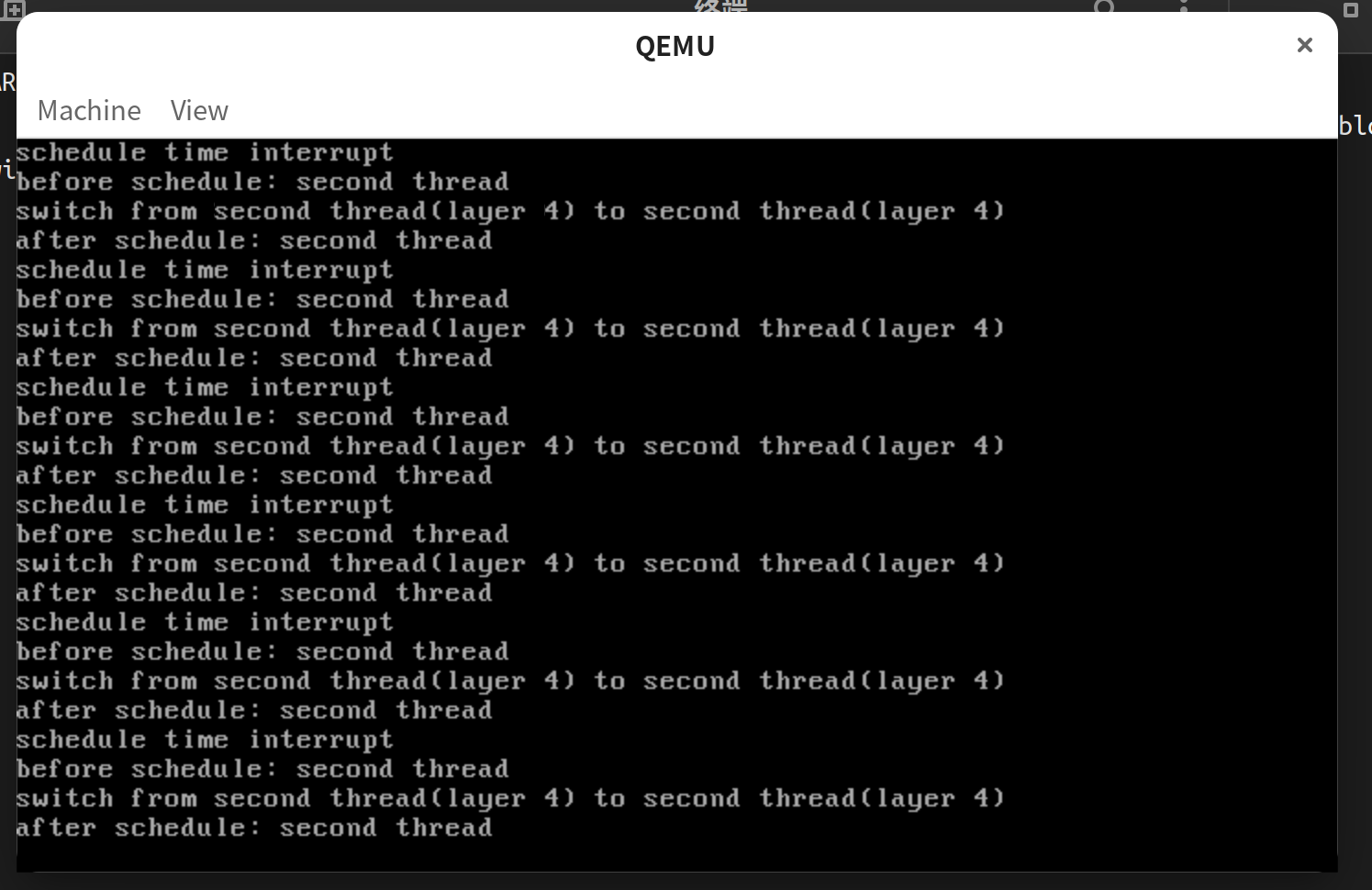
int pid1 = programManager.executeThread(first\_thread, nullptr, "first thread", 10);

结果和FCFS一样就不放了，测试结果表明SJF能每次选取时间最短的任务调度。

PR和上面一样的也不放了。

MLFQ：





每次从尽可能从高优先级队列选取任务并调度，如果一次调度时间片结束还没结束，就将其放到低优先级的队列。

## 总结

被迫写C style代码。没有C++标准库的情况下连虚函数和RTTI都没法用，不知道是不是ld的参数没有配置好，不是很想折腾。

Assignment 3中asm\_switch\_thread的汇编层面上的调试还是太难了，即使知道栈会怎么变化也没什么好的办法能漂亮的展示出栈的切换前后函数返回地址的变化，索性放弃。