

**本科生实验报告**

实验课程：操作系统原理实验

任课教师：刘宁

实验题目：内核线程

专业名称：计算机科学与技术

学生姓名：

学生学号：

实验地点：实验中心B202

实验时间：2025.5.6

**Section 1 实验概述**

在本次实验中，将先学习C语言的可变参数机制。在此基础上，进一步揭开可变参数背后的原理，进而实现C语言的可变参数机制。在实现了可变参数机制后，将实现一个较为简单的printf函数。此后，变多了一种调试方式，可以使用printf的方式结合gdb来帮助我们debug。

然后，需要实现内核线程，首先会定义线程控制块的数据结构——PCB。然后，创建PCB，在PCB中放入线程执行所需的参数。最后，在实现基于时钟中断的时间片轮转(RR)调度算法的基础上实现另外一种调度算法，比如先来先服务调度算法。

在本次实验，我们可以学习到asm\_switch\_thread是如何实现线程切换的，以及体会操作系统实现并发执行的原理。

**Section 2 预备知识与实验环境**

* 预备知识：x86汇编语言、IA-32处理器体系结构、C语言的可变参数函数、线程的切换、操作系统并发执行的原理
* 实验环境：
  + 虚拟机版本/处理器型号：Virtualbox7.0.6、Ubuntu18.04
  + 代码编辑环境：gedit文本编辑器、gdb调试器、qemu
  + 代码编译工具：nasm

**Section 3 实验任务**

* 实验任务1：**printf的实现**，学习可变参数机制，然后实现printf函数
* 实验任务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
* 实验任务3：**时钟中断的实现**，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
* 实验任务4：**调度算法的实现**，自己设计的算法。然后，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

**Section 4 实验步骤与实验结果**

------------------------- **实验任务1** -------------------------

* 任务要求：学习可变参数机制，然后实现printf函数，结果截图保存并说说你是怎么做的。
* 思路分析：

printf函数首先找到fmt中的形如%c,%d,%x,%s对应的参数，然后用这些参数具体的值来替换%c,%d,%x,%s等，得到一个新的格式化输出字符串，这个过程称为fmt的解析。最后，printf将这个新的格式化输出字符串即可。然而，这个字符串可能非常大，会超过函数调用栈的大小。实际上，我们会定义一个缓冲区，然后对fmt进行逐字符地解析，将结果逐字符的放到缓冲区中。放入一个字符后，我们会检查缓冲区，如果缓冲区已满，则将其输出，然后清空缓冲区，否则不做处理。

* 实验步骤：

Step1:编写字符输出函数，这个函数能够正确处理字符串中的\n换行字符

int STDIO::print(const char \*const str)

{

int i = 0;

for (i = 0; str[i]; ++i)

{

switch (str[i])

{

case '\n':

uint row;

row = getCursor() / 80;

if (row == 24)

{

rollUp();

}

else

{

++row;

}

moveCursor(row \* 80);

break;

default:

print(str[i]);

break;

}

}

return i;

}

Step2：实现printf函数，

int printf(const char \*const fmt, ...)

{

const int BUF\_LEN = 32;

char buffer[BUF\_LEN + 1];

char number[33];

int idx, counter;

va\_list ap;

va\_start(ap, fmt);

idx = 0;

counter = 0;

for (int i = 0; fmt[i]; ++i)

{

if (fmt[i] != '%')

{

counter += printf\_add\_to\_buffer(buffer, fmt[i], idx, BUF\_LEN);

}

else

{

i++;

if (fmt[i] == '\0')

{

break;

}

switch (fmt[i])

{

case '%':

counter += printf\_add\_to\_buffer(buffer, fmt[i], idx, BUF\_LEN);

break;

case 'c':

counter += printf\_add\_to\_buffer(buffer, va\_arg(ap, int), idx, BUF\_LEN);

break;

case 's':

buffer[idx] = '\0';

idx = 0;

counter += stdio.print(buffer);

counter += stdio.print(va\_arg(ap, const char \*));

break;

case 'd':

case 'x':

int temp = va\_arg(ap, int);

if (temp < 0 && fmt[i] == 'd')

{

counter += printf\_add\_to\_buffer(buffer, '-', idx, BUF\_LEN);

temp = -temp;

}

temp = itos(number, temp, (fmt[i] == 'd' ? 10 : 16));

for (int j = temp - 1; j >= 0; --j)

{

counter += printf\_add\_to\_buffer(buffer, number[j], idx, BUF\_LEN);

}

break;

}

}

}

buffer[idx] = '\0';

counter += stdio.print(buffer);

return counter;

}

Step3：实现printf\_add\_to\_buffer函数，对fmt进行逐字符解析，对于每一个字符fmt[i]，如果fmt[i]不是%，则说明是普通字符，直接放到缓冲区即可。注意，将fmt[i]放到缓冲区后可能会使缓冲区变满，此时如果缓冲区满，则将缓冲区输出并清空，如果是%那么说明是格式控制符，

格式控制如下

|  |  |
| --- | --- |
| %% | 输出一个% |
| %c | 输出ap指向的字符 |
| %s | 输出ap指向的字符串的地址对应的字符串 |
| %d | 输出ap指向的数字对应的十进制表示 |
| %x | 输出ap指向的数字对应的16进制表示 |

int printf\_add\_to\_buffer(char \*buffer, char c, int &idx, const int BUF\_LEN)

{

int counter = 0;

buffer[idx] = c;

++idx;

if (idx == BUF\_LEN)

{

buffer[idx] = '\0';

counter = stdio.print(buffer);

idx = 0;

}

return counter;

}

Step4：实现对应数字转换成对应进制的字符串函数

void itos(char \*numStr, uint32 num, uint32 mod) {

// 只能转换2~26进制的整数

if (mod < 2 || mod > 26 || num < 0) {

return;

}

uint32 length, temp;

// 进制转换

length = 0;

while(num) {

temp = num % mod;

num /= mod;

numStr[length] = temp > 9 ? temp - 10 + 'A' : temp + '0';

++length;

}

// 特别处理num=0的情况

if(!length) {

numStr[0] = '0';

++length;

}

// 将字符串倒转，使得numStr[0]保存的是num的高位数字

for(int i = 0, j = length - 1; i < j; ++i, --j) {

swap(numStr[i], numStr[j]);

}

numStr[length] = '\0';

}

Step5：最后在setup\_kernel中测试我们所完成的printf函数

#include "asm\_utils.h"

#include "interrupt.h"

#include "stdio.h"

// 屏幕IO处理器

STDIO stdio;

// 中断管理器

InterruptManager interruptManager;

extern "C" void setup\_kernel()

{

// 中断处理部件

interruptManager.initialize();

// 屏幕IO处理部件

stdio.initialize();

interruptManager.enableTimeInterrupt();

interruptManager.setTimeInterrupt((void \*)asm\_time\_interrupt\_handler);

//asm\_enable\_interrupt();

printf("print percentage: %%\n"

"print char \"N\": %c\n"

"print string \"Hello World!\": %s\n"

"print decimal: \"-1234\": %d\n"

"print hexadecimal \"0x7abcdef0\": %x\n",

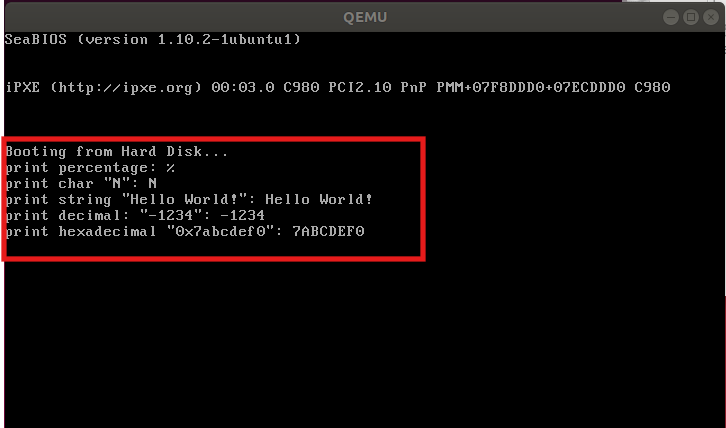
'N', "Hello World!", -1234, 0x7abcdef0);

//uint a = 1 / 0;

asm\_halt();

}

* 实验结果展示：通过执行前述代码，可得下图结果。



------------------------- **实验任务2** -------------------------

* 任务要求：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
* 思路分析：实现线程的，我们首先要设置PCB来存放和描述我们的线程，然后编写函数来为我们的线程分配和回收PCB资源，然后编写创建线程函数，同时创建完之后编写线程退出函数，在实现线程的创建之后，编写线程调度函数来调度我们的线程，然后编写线程切换函数来切换我们的线程
* 实验步骤：

Step1：描述线程的状态以及编写PCB数据结构表示线程

线程的状态有5个，分别是创建态、运行态、就绪态、阻塞态和终止态

enum ProgramStatus

{

CREATED,

RUNNING,

READY,

BLOCKED,

DEAD

};

线程有很多的信息，我们创建一个PCB的数据结构集中保存

他们的含义分别如下表所示：

|  |  |
| --- | --- |
| stack | 每一个线程都有自己的栈 |
| status | 线程的状态，如运行态、阻塞态和就绪态等。 |
| name | 线程的名称 |
| priority | 线程的优先级，线程的优先级决定了抢占式调度 |
| pid | 线程的标识符，线程的pid唯一 |
| ticks | 线程剩余的执行次数 |
| ticksPassedBy | 线程总共执行的tick的次数 |
| tagInGeneralList | 线程在线程队列中的标识 |
| tagInAllList | 线程在线程队列中的标识 |

struct PCB

{

int \*stack; // 栈指针，用于调度时保存esp

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

enum 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

ListItem tagInGeneralList; // 线程队列标识

ListItem tagInAllList; // 线程队列标识

};

Step2：PCB的分配

存放所有PCB的数据结构

// PCB的大小，4KB。

const int PCB\_SIZE = 4096;

// 存放PCB的数组，预留了MAX\_PROGRAM\_AMOUNT个PCB的大小空间。

char PCB\_SET[PCB\_SIZE \* MAX\_PROGRAM\_AMOUNT];

// PCB的分配状态，true表示已经分配，false表示未分配。

bool PCB\_SET\_STATUS[MAX\_PROGRAM\_AMOUNT];

**allocatePCB分配PCB**：allocatePCB会去检查PCB\_SET中每一个PCB的状态，如果找到一个未被分配的PCB，则返回这个PCB的起始地址。

PCB \*ProgramManager::allocatePCB(){

for (int i = 0; i < MAX\_PROGRAM\_AMOUNT; ++i)

{

if (!PCB\_SET\_STATUS[i])

{

PCB\_SET\_STATUS[i] = true;

return (PCB \*)((int)PCB\_SET + PCB\_SIZE \* i);

}

}

return nullptr;

}

**releasePCB释放PCB空间**：releasePCB接受一个PCB指针program，然后计算出program指向的PCB在PCB\_SET中的位置，然后将PCB\_SET\_STATUS中的对应位置设置false即可

void ProgramManager::releasePCB(PCB \*program)

{

int index = ((int)program - (int)PCB\_SET) / PCB\_SIZE;

PCB\_SET\_STATUS[index] = false;

}

Step3：线程的创建

**ProgramManager的初始化：**

ProgramManager::ProgramManager()

{

initialize();

}

void ProgramManager::initialize(){

allPrograms.initialize();

readyPrograms.initialize();

running = nullptr;

for (int i = 0; i < MAX\_PROGRAM\_AMOUNT; ++i)

{

PCB\_SET\_STATUS[i] = false;

}

}

**executeThread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 \* 10;

thread->ticksPassedBy = 0;

thread->pid = ((int)thread - (int)PCB\_SET) / PCB\_SIZE;

// 线程栈

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

thread->stack -= 7;

thread->stack[0] = 0;

thread->stack[1] = 0;

thread->stack[2] = 0;

thread->stack[3] = 0;

thread->stack[4] = (int)function;

thread->stack[5] = (int)program\_exit;

thread->stack[6] = (int)parameter;

allPrograms.push\_back(&(thread->tagInAllList));

readyPrograms.push\_back(&(thread->tagInGeneralList));

// 恢复中断

interruptManager.setInterruptStatus(status);

return thread->pid;

}

Step4：线程调度

**schedule实现线程调度：**

首先，为了实现线程互斥，在进程线程调度前，我们需要关中断，退出时再恢复中断。我们判断当前可调度的线程数量，如果readyProgram为空，那么说明当前系统中只有一个线程，因此无需进行调度，直接返回即可。我们判断当前线程的状态，如果是RUNNING，则重新初始化其状态为READY和ticks，并放入就绪队列；如果是DEAD，则回收线程的PCB。然后去就绪队列的第一个线程作为下一个执行的线程。从就绪队列中删去第一个线程，设置其状态为运行态和当前正在执行的线程。

void ProgramManager::schedule(){

bool status = interruptManager.getInterruptStatus();

interruptManager.disableInterrupt();

if (readyPrograms.size() == 0)

{

interruptManager.setInterruptStatus(status);

return;

}

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

{

running->status = ProgramStatus::READY;

running->ticks = running->priority \* 10;

readyPrograms.push\_back(&(running->tagInGeneralList));

}

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

{

releasePCB(running);

}

ListItem \*item = readyPrograms.front();

PCB \*next = ListItem2PCB(item, tagInGeneralList);

PCB \*cur = running;

next->status = ProgramStatus::RUNNING;

running = next;

readyPrograms.pop\_front();

asm\_switch\_thread(cur, next);

interruptManager.setInterruptStatus(status);

}

**asm\_switch\_thread实现线程切换：**

首先保存寄存器ebp，ebx，edi，esi，然后保存esp的值到线程的PCB::statck中，用做下次恢复。然后将cur->stack的地址放到eax中，第8行向[eax]中写入esp的值，也就是向cur->stack中写入esp，将next->stack的值写入到esp中，从而完成线程栈的切换。

asm\_switch\_thread:

push ebp

push ebx

push edi

push esi

mov eax, [esp + 5 \* 4]

mov [eax], esp ; 保存当前栈指针到PCB中，以便日后恢复

mov eax, [esp + 6 \* 4]

mov esp, [eax] ; 此时栈已经从cur栈切换到next栈

pop esi

pop edi

pop ebx

pop ebp

sti

ret

program\_exit()实现线程的结束后的操作：program\_exit会将返回的线程的状态置为DEAD，然后调度下一个可执行的线程上处理器

void program\_exit(){

PCB \*thread = programManager.running;

thread->status = ThreadStatus::DEAD;

if (thread->pid)

{

programManager.schedule();

}

else

{

interruptManager.disableInterrupt();

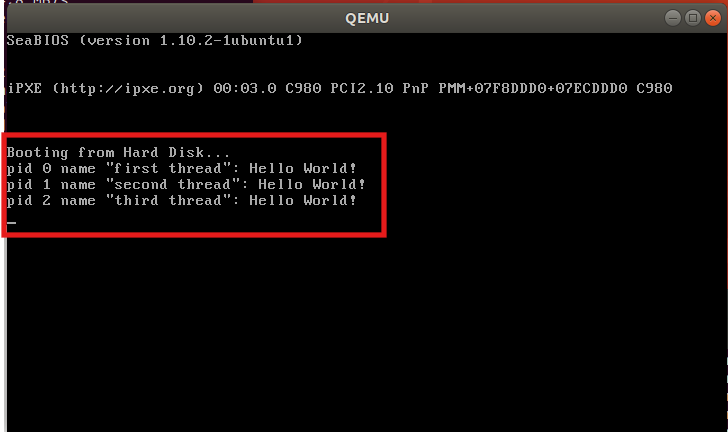
printf("halt\n");

asm\_halt();

}

}

* 实验结果展示：通过执行前述代码，可得下图结果。



------------------------- **实验任务3** -------------------------

* 任务要求：编写若干个线程函数，使用gdb跟踪c\_time\_interrupt\_handler、 asm\_switch\_thread（eg: bc\_time\_interrupt\_handler）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。

1. 一个新创建的线程是如何被调度然后开始执行的。

2. 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。

* 思路分析：

首先在c\_time\_interrupt\_handler和asm\_switch\_thread设置断点两个函数设置断点，在每次中断的时候查看当前进程运行进程的pid、ticks、stack，观察中断是怎么驱动线程运行的

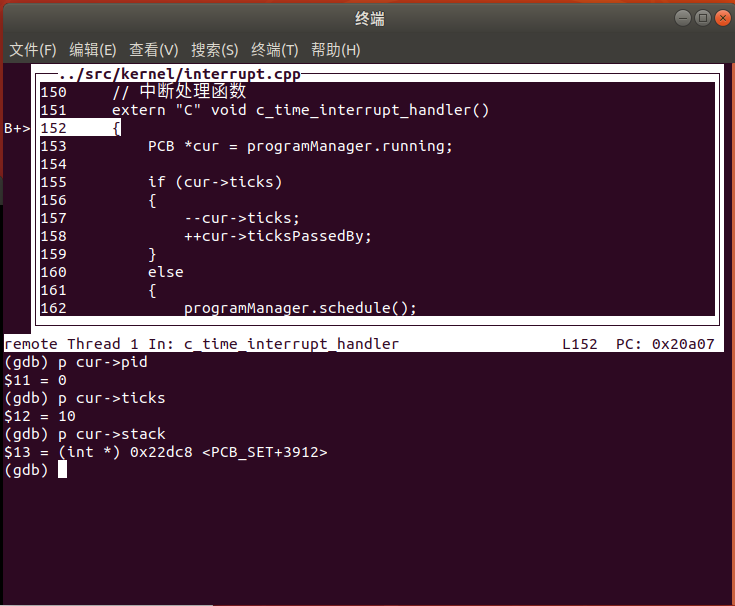
然后在进程切换函数asm\_switch\_thread，观察前后寄存器以及栈的值，观察当前线程如何被换下的以及下一个进程如何从中断点开始执行的

* 实验步骤：

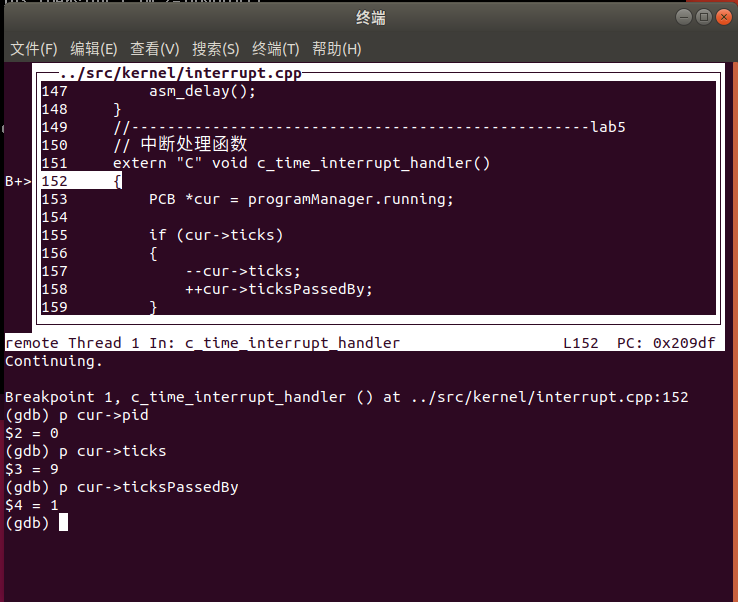
Step1：首先为c\_time\_interrupt\_handler和asm\_switch\_thread设置断点

Step2：观察一个新创建的线程是如何被调度然后开始执行的

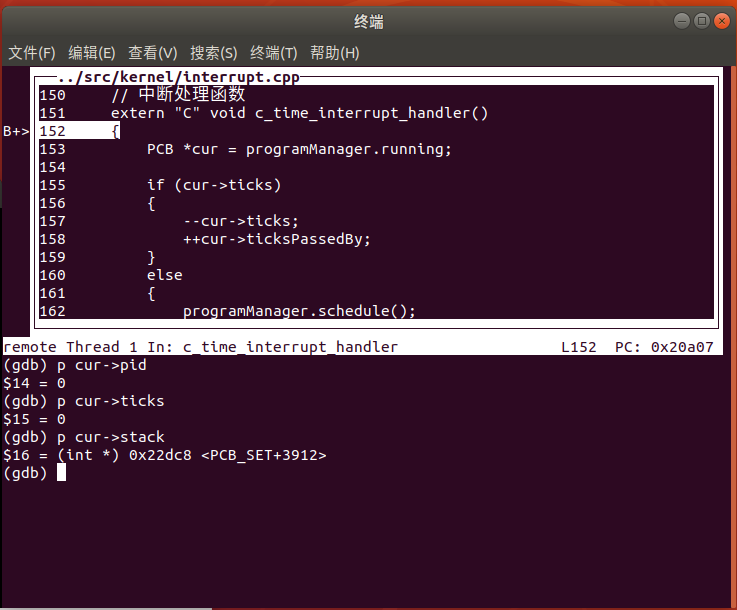
c运行可以看到，当前0号进程正在运行，ticks为10，



当到下一次时钟中断的时候，我们可以发现还是进程0在运行，但是ticks减少了1

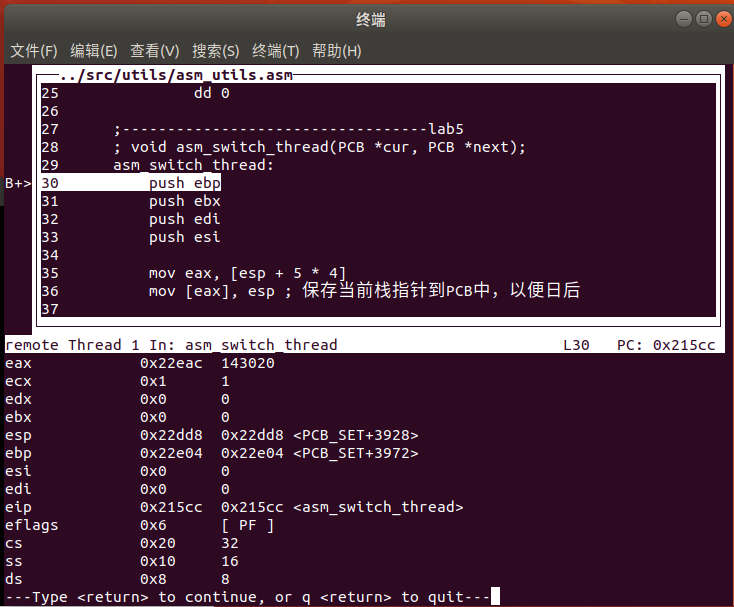


当进程0的时间片用完时，ticks为0，这个时候要进行进程的切换

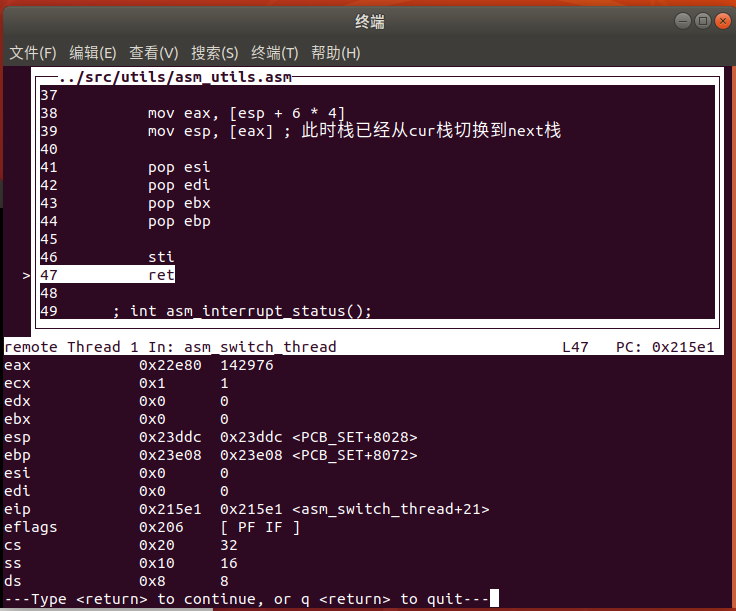


这时就会运行调度算法寻找下一个就绪进程，然后asm\_switch\_thread函数就会实现进程的切换，

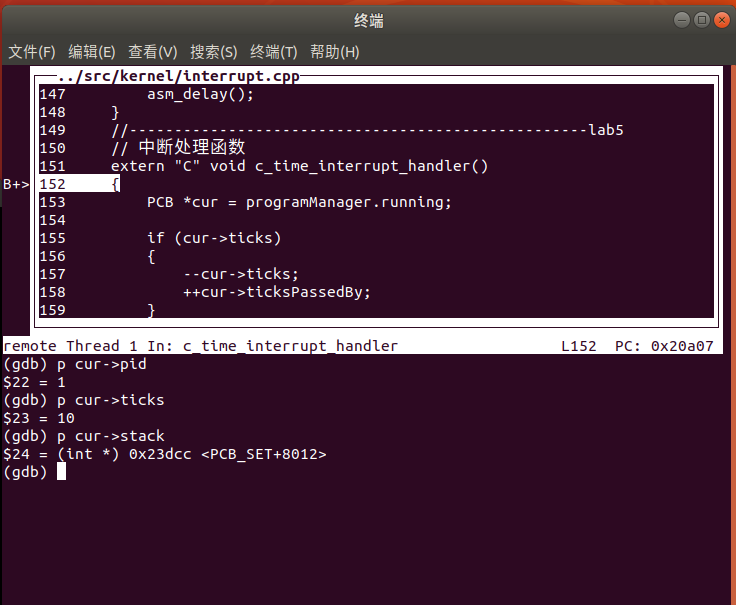
切换前各个寄存器以及栈指针的值，在切换之前，进程先将自己的栈指针保存下来，等待后续的恢复



调度完成之后，我们可以看到栈指针切换成了下一个进程的栈指针，这样我们下一个进程就顺利的运行了起来

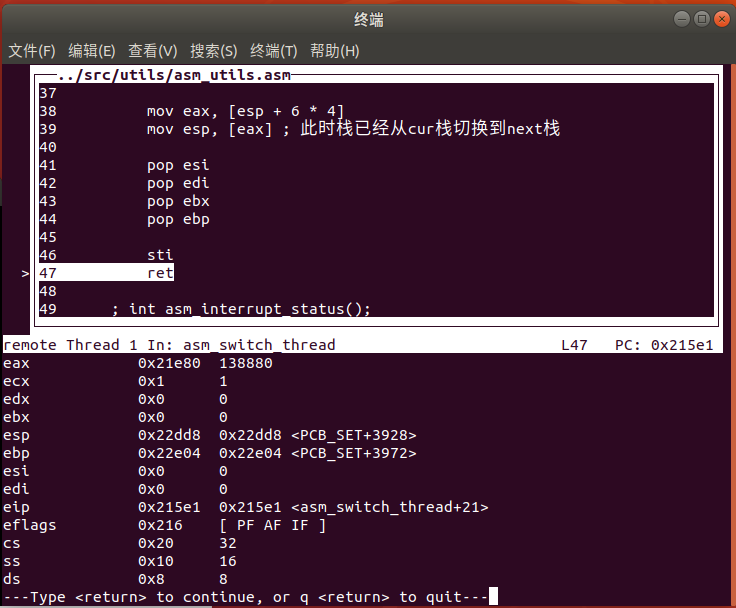


切换之后，当前运行的进程为1，ticks为10

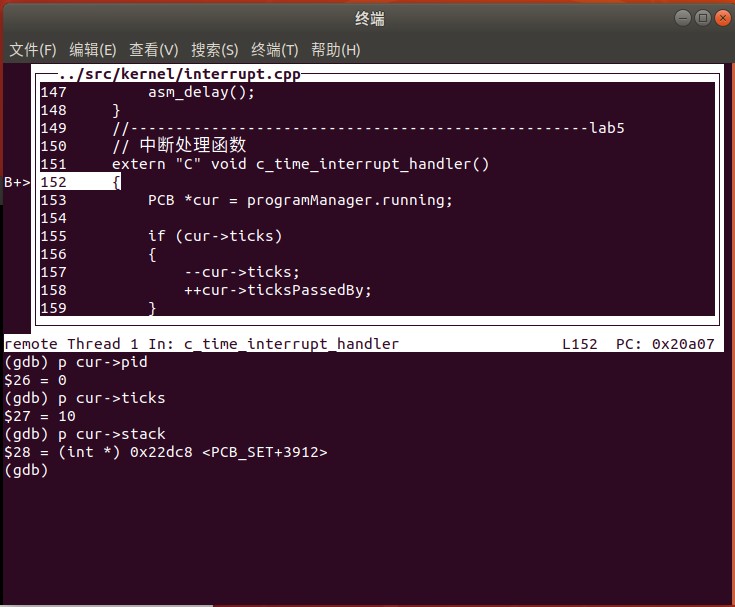


Step3：接下来我们看看再次切换为进程0，进程0是怎么从上一次的中断点恢复的

可以看到当再次切换到进程0时，之前保存在进程0的PCB中的栈指针进行恢复，可以看到当前的栈指针就是进程0的退出之前的栈指针



再次获得CPU资源后，进程0继续之前的程序继续运行了



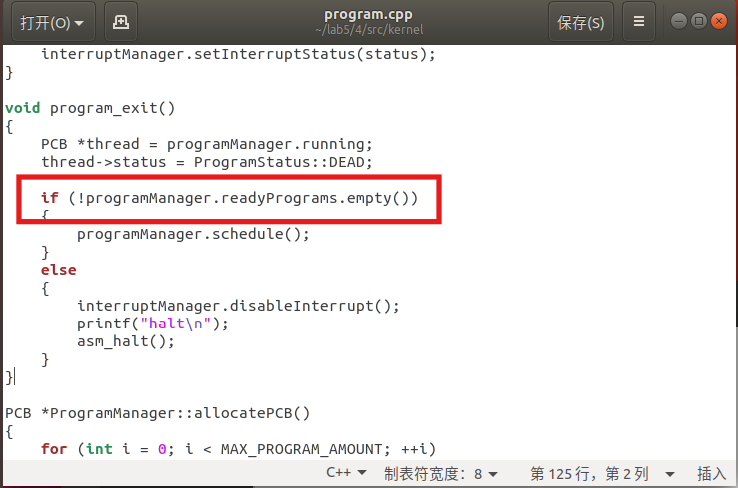
------------------------- **实验任务4** -------------------------

* 任务要求：现在，将线程调度算法修改为上面提到的算法或者是自己设计的算法。然后，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
* 思路分析：调度算法有很多种，本次实验任务选择完成先来先服务调度策略，首先我们将中断处理函数的处理都进行注释，也就是说发生时钟中断时不进行任何操作，然后修改program\_exit，当我们的就绪队列不为空时，我们进行调度。这样我们的先来先服务调度算法就修改好了
* 实验步骤：

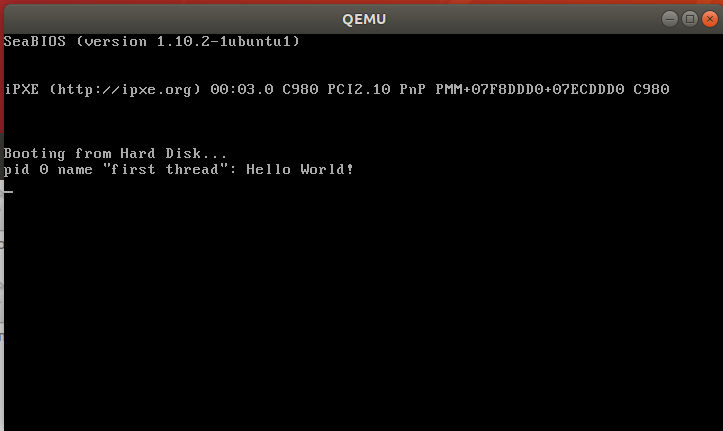
Step1：修改中断处理函数



Step2：线程退出函数，也就是当当前线程退出的时候，如果就绪队列不为空，那么就进行调度



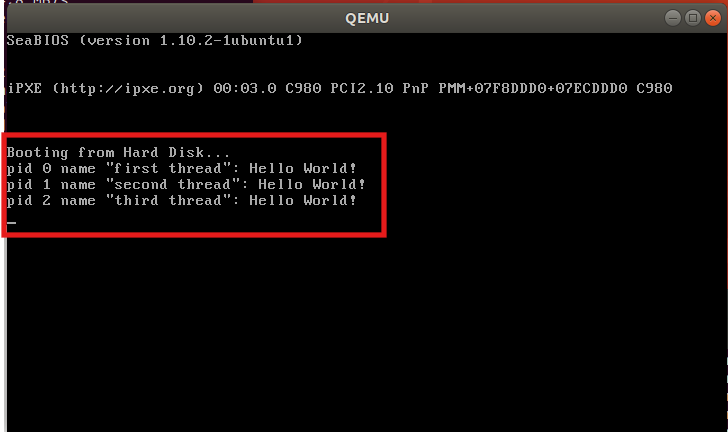
运行结果如下，可以看到由于我们线程0在最后是死循环，导致我们的进程0不会退出，而先来先服务的调度是一定要等待进程退出时才能调度的



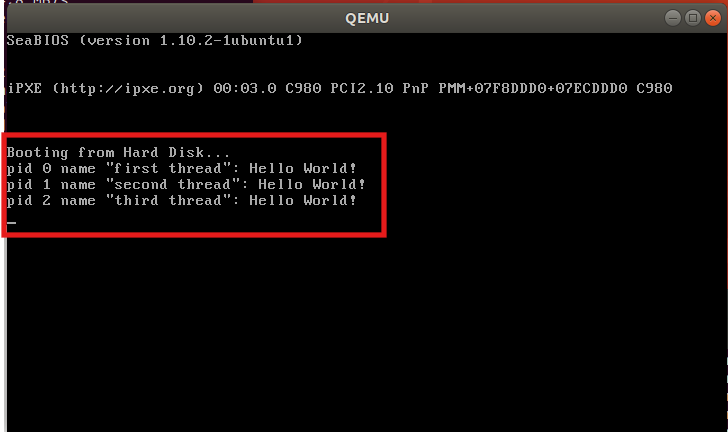
Step3：修改线程函数，只让最后一个线程死循环



结果如下，可以看到我们的三个线程就顺利的执行了



* 实验结果展示：通过执行前述代码，可得下图结果，



**Section 5 实验总结与心得体会**

本次实验，首先学习到了C语言中的可变参数机制的原理，进而实现了可变参数的机制，通过这个机制我们编写了printf函数来帮助我对我们的程序进行调试，同时我们可以结合printf和gdb来进行调试。

同时本次实验也进一步复习了程序、进程、线程之间的关系以及他们的之间区别，程序是指静态的、存储在文件系统上的、尚未运行的指令代码，进程是指正在运行的程序，即进行中的程序，线程实际上是函数的载体，属于创建它的进程。

在了解了这些区别之后由于进程要使用到二级分页机制，所以本次实验我们先实现操作系统的线程，然后我们从线程的描述开始，首先为每一个线程设置一个PCB的数据结构，用于存放线程的所有信息，然后一步步完成线程的创建、线程的RR调度算法、线程的切换。这样实现了我们操作系统线程功能。

然后我们通过gdb调试跟踪中断处理函数和切换线城时的寄存器值，进一步体会到了操作系统通过中断来驱动我们的程序，以及操作系统如何实现线程的并发。

最后本次实验还实现了另一种先来先服务的调度算法，进一步体会到了不同调度算法的优劣之处，在实际应用中应该根据实际情况来选择我们的调度算法。

**Section 6 附录：参考资料清单**

[1][详解操作系统内核对线程的调度算法 - 知乎](https://zhuanlan.zhihu.com/p/97071815)

**Section 7 附录：代码清单**

由于代码文件较多， 代码量较大， 为了篇幅的精简， 具体代码以附件的形式给出