本科生实验报告

实验课程：操作系统实验

实验名称：lab5 内核线程

专业名称：计算机科学与技术

学生姓名：杜翊菲

学生学号：23336062

实验地点：实验大楼 B201

实验成绩：

报告时间：2025 年 4 月 22 日

Lab5 内核线程

一、实验要求

1. 学习 C 语言的可变参数机制的实现方法，并了解可变参数背后的原理，进而实现可变参数机制；实现可变参数机制后，自主实现一个较为简单的 printf 函数，并学习同时使用 printf 和 gdb 来帮助 debug。

2. 本次实验另外一个重点是内核线程的实现。首先要学会定义线程控制块的数据结构——PCB，然后学会创建 PCB，在 PCB 中放入线程执行所需的参数，最后学会实现基于时钟中断的时间片轮转(RR)调度算法。这一部分，需重点理解 asm_switch_thread 是如何实现线程切换的，体会操作系统实现并发执行的原理。

二、实验过程（每个部分包括实验操作、关键代码和结果截图）

（一）Assignment 1：printf 的实现

1. 复现 Example 1

学会了可变参数后，printf的实现便不再困难。在实现printf前，我们首先要明白printf的作用。printf的作用是格式化输出，并返回输出的字符个数，其定义如下。

```
int printf(const char *const fmt, ...);
```

在格式化输出字符串中，会包含 `%c,%d,%x,%s` 等来实现格式化输出，对应的参数在可变参数中可以找到。明白了printf的作用，printf的实现便迎刃而解，实现思路如下。

printf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。

在实现printf前，我们需要一个能够输出字符串的函数，这个函数能够正确处理字符串中的 `\n` 换行字符。这里，有同学会产生疑问，`\n` 不是直接输出就可以了吗？其实 `\n` 的换行效果是我们人为规定的，换行的实现需要我们把光标放到下一行的起始位置，如果光标超过了屏幕的表示范围，则需要滚屏。因此，我们实现一个能够输出字符串的函数 `STDIO::print`，声明和实现分别放在 `include/stdio.h` 和 `src/kernel/stdio.cpp` 中，如下所示。

```
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;
}
```

> 在程序设计中，命名是一件令人苦恼的事情，而借助于C++的函数重载，我们可以将许多功能类似的函数用统一的名字来表示。

我们实现的printf比较简单，只能解析如下参数。

符号	含义
%d	按十进制整数输出
%c	输出一个字符
%s	输出一个字符串
%x	按16进制输出

按照前面描述的过程，printf的实现如下。

```
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;
}
```

首先我们定义一个大小为`BUF_LEN`的缓冲区`buffer`，`buffer`多出来的1个字符是用来放置`\0`的。由于我们后面会将一个整数转化为字符串表示，`number`使用来存放转换后的数字字符串。由于保护模式是运行在32位环境下的，最大的数字字符串也不会超过32位，因此number分配33个字节也就足够了。

接着我们开始对`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，我们不妨上述过程写成一个函数来实现，如下所示。

```
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;
}
```

如果`fmt[i]`是`%`，则说明这可能是一个格式化输出的参数。因此我们检查`%`后面的参数，分为如下情况分别处理。

- `%%`。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。
- 其他。不做任何处理。

对于`%d`和`%x`，我们需要将数字转换为对应的字符串。一个数字向任意进制表示的字符串的转换函数如下所示，声明放置在`include/stdlib.h`中，实现放置在`src/utils/stdlib.cpp`中。

```
/*
 * 将一个非负整数转换为指定进制表示的字符串。
 * num: 待转换的非负整数。
 * mod: 进制。
 * numStr: 保存转换后的字符串，其中，numStr[0]保存的是num的高位数字，以此类推。
 */

void itos(char *numStr, uint32 num, uint32 mod);
```

```
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';
}
```

其中，`swap` 函数也是声明在 `include/stdlib.h`，实现在 `src/utils/stdlib.cpp` 中。

```
template<typename T>
void swap(T &x, T &y);
```

```
template<typename T>
void swap(T &x, T &y) {
    T z = x;
    x = y;
    y = z;
}
```

上述函数比较简单，我们不再赘述。

由于 `itos` 转换的是非负整数，对于 `%d` 的情况，如果我们输出的整数是负数，那么就要使用 `itos` 转换其相反数，在输出数字字符串前输出一个负号。

最后，当我们逐字符解析完 `fmt` 后， `buffer` 中可能还会有未输出的字符，我们要将缓冲区的字符全部输出，返回输出的总字符 `counter` 。

接下来我们测试这个函数，我们在 `setup_kernel` 中加入对应的测试语句。

```
#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();
}
```

然后修改makefile。

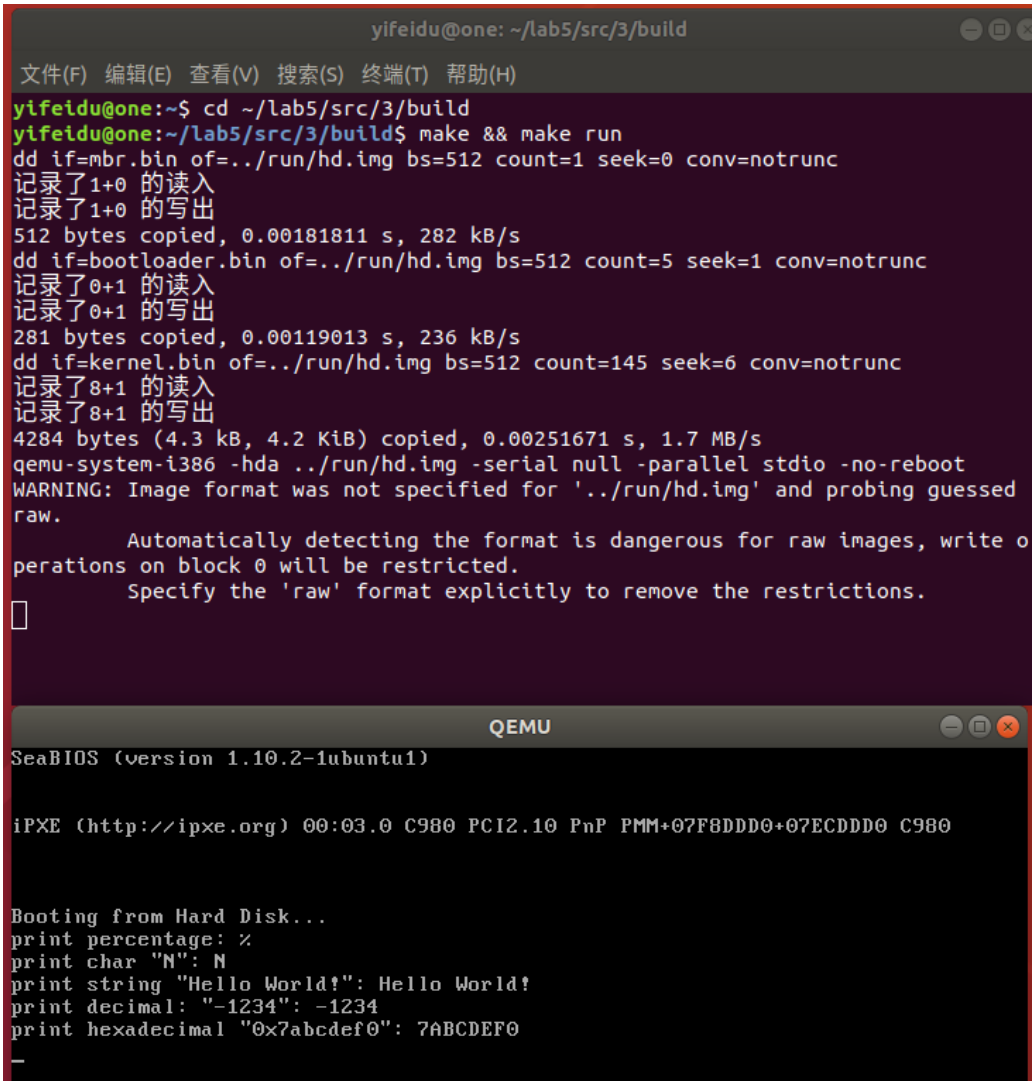
```
...

RUNDIR = ../run
BUILDDIR = build
INCLUDE_PATH = ../include


KERNEL_SOURCE = $(wildcard $(SRCDIR)/kernel/*.cpp)
CXX_SOURCE += $(KERNEL_SOURCE)
CXX_OBJ += $(KERNEL_SOURCE:$(SRCDIR)/kernel/%.cpp=%.o)
```

```
UTILS_SOURCE = $(wildcard $(SRCDIR)/utils/*.cpp)
CXX_SOURCE += $(UTILS_SOURCE)
CXX_OBJ += $(UTILS_SOURCE:$(SRCDIR)/utils/%.cpp=%.o)

ASM_SOURCE += $(wildcard $(SRCDIR)/utils/*.asm)
ASM_OBJ += $(ASM_SOURCE:$(SRCDIR)/utils/%.asm=%.o)

OBJ += $(CXX_OBJ)
OBJ += $(ASM_OBJ)

...
```

编译运行，输出如下结果。

```
make && make run
```

要求是根据上述教程复现 Example 1，由于关键代码在教程中已经给出，所以过程比较简单，只需要正确输入命令即可得到预计结果。实验截图如下所示：



2. 代码解读

（1） 可变参数机制的实现

```
#ifndef STDARG_H
#define STDARG_H

typedef char *va_list;
#define _INTSIZEOF(n) ((sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1))
#define va_start(ap, v) (ap = (va_list)&v + _INTSIZEOF(v))
#define va_arg(ap, type) (*(type *)((ap += _INTSIZEOF(type)) - _INTSIZEOF(type)))
#define va_end(ap) (ap = (va_list)0)

#endif
```

① **数据类型定义**：va_list 是字符指针，指向可变参数（字节级指针），作用是按字节访问栈中的参数。

② **字节对齐宏_INTSIZEOF**：通过+sizeof(int)-1 向上取整，再通过按位与~(sizeof(int)-1)确保结果为 4 的倍数。作用是确保参数在栈中按 4 字节对齐（32 位系统）。

③ **va_start 宏**：利用固定参数 v 的地址，加上其对齐后的大小，得到可变参数的起始地址。作用是初始化可变参数指针 ap，使其指向最后一个固定参数 v 的下一个参数位置。

④ **va_arg 宏**：先将 ap 增加 type 对齐后的大小（跳过当前参数），再回退对齐大小，获取当前参数的地址，强制转换为 type 类型后取值。作用是按指定类型 type 获取当前参数，并将指针 ap 指向下一个参数。

⑤ **va_end 宏**：作用是清空指针 ap，避免后续非法访问栈内存。

（2） printf 函数的实现

① **核心逻辑**：输入格式字符串 fmt 和可变参数列表，输出格式化后的字符串到屏幕，返回输出字符总数。

② **缓冲区管理**:暂存待输出字符，满缓冲区时（BUF_LEN 字符）触发输出，避免频繁屏幕操作。

```
int printf(const char *const fmt, ...)
{
    const int BUF_LEN = 32;

    char buffer[BUF_LEN + 1];
    char number[33];
```

③ **格式解析与处理**：遍历格式字符串 fmt，区分普通字符和格式说明符。

普通字符直接写入缓冲区：

```
    if (fmt[i] != '%')
    {
        counter += printf_add_to_buffer(buffer, fmt[i], idx, BUF_LEN);
    }
```

格式说明符用于处理%c、%s、%d、%x 等情况。

%c：获取字符参数并写入缓冲区；**%s**：直接输出字符串参数；

%d %x：将整数转换为字符串后写入缓冲区；**%%**：输出单个%。

```
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, char), 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;
    }

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

    for (int j = 0; number[j]; ++j)
    {
        counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
    }
    break;
}
```

④ **屏幕输出函数**：处理\n 换行，确保光标移动到下一行行首，超出屏幕范围时滚屏。

```
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;
}
```

（3） 整数到字符串的转换函数 itos

```
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';
}
```

3. 输入待转换的非负整数 num，进制 mod（10 或 16），目标字符串 numStr。步骤为：①逐位取余，转换为字符（数字或大写字母）；②处理 num=0 的特殊情况；③逆序字符数组，确保高位在前

（二）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。

1. 复现 Example 2

我们创建第一个线程，并输出“Hello World”，pid和线程的name。注意，第一个线程不可以返回。代码在 `src/kernel/setup.cpp` 中，如下所示。

```
#include "asm_utils.h"
#include "interrupt.h"
#include "stdio.h"
#include "program.h"
#include "thread.h"

// 屏幕IO处理器
STDIO stdio;
```

```
// 中断管理器
InterruptManager interruptManager;
// 程序管理器
ProgramManager programManager;

void first_thread(void *arg)
{
    // 第1个线程不可以返回
    printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid, programManager.running->name);
    asm_halt();
}

extern "C" void setup_kernel()
{

    // 中断管理器
    interruptManager.initialize();
    interruptManager.enableTimeInterrupt();
    interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);

    // 输出管理器
    stdio.initialize();

    // 进程/线程管理器
    programManager.initialize();

    // 创建第一个线程
    int pid = programManager.executeThread(first_thread, nullptr, "first thread", 1);
    if (pid == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }
```

```
    ListItem *item = programManager.readyPrograms.front();
    PCB *firstThread = ListItem2PCB(item, tagInGeneralList);
    firstThread->status = RUNNING;
    programManager.readyPrograms.pop_front();
    programManager.running = firstThread;
    asm_switch_thread(0, firstThread);

    asm_halt();
}
```

第12行，我们定义全局变量`ProgramManager`。

第36-48行，我们创建第一个线程。由于当前系统中没有线程，因此我们无法通过在时钟中断调度的方式将第一个线程换上处理器执行。因此我们的做法是找出第一个线程的PCB，然后手动执行类似`schedule`的过程，最后执行的`asm_switch_thread`会强制将第一个线程换上处理器执行。

我们编译运行，

```
make && make run
```

该部分的主要目标是在操作系统实验环境中创建并启动第一个内核线程，由于系统初始时没有正在运行的线程，无法依赖常规的时钟中断调度机制将新线程投入运行，因此采用手动调度的方式来达成目的。其具体实现步骤为：初始化相关管理器（中断管理器、输出管理器、进程/线程管理器）、创建第一个线程、手

动调度第一个线程（获取并修改线程状态、调整线程队列和当前运行线程、强制线程切换）。

根据上述教程成功复现 Example 2，实验结果如下所示：

2. 自行设计 PCB 实现线程

在 example 2 的基础上修改了 setup.cpp，主要改动涉及线程函数的定义和线程的创建、调度过程，目的是创建多个线程并手动调度第一个线程执行，从而多个线程被创建且第一个线程率先运行并输出信息。

有改动的部分如下所示：

```
void fourth_thread(void *arg) {
    printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid, programManager.running->name);
    asm_halt();
}

void third_thread(void *arg) {
    printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid, programManager.running->name);
    asm_halt();
}

void second_thread(void *arg) {
    printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid, programManager.running->name);
    asm_halt();
}

void first_thread(void *arg)
{               // 输出信息，表明线程已经开始运行
    printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid, programManager.running->name);
    asm_halt();
}
```

```
    int pid0 = programManager.executeThread(first_thread, nullptr, "first thread", 1);
    int pid1 = programManager.executeThread(second_thread, nullptr, "second thread", 1);
    int pid2 = programManager.executeThread(third_thread, nullptr, "third thread", 1);
    int pid3 = programManager.executeThread(fourth_thread, nullptr, "fourth thread", 1);
    //手动切换到第一个线程
    ListItem *item = programManager.readyPrograms.front();
    PCB *firstThread = ListItem2PCB(item, tagInGeneralList);
    firstThread->status = RUNNING;
    programManager.readyPrograms.pop_front();
    programManager.running = firstThread;
    asm_switch_thread(0, firstThread); //从内核态切换到用户态
```

相比原代码，修改后的代码创建了四个线程，增加了系统中的线程数量，丰富了多线程环境，输出信息增多。多个线程的创建使得线程调度变得更为复杂。原代码只需处理单个线程的调度问题，修改后需要考虑多个线程的调度顺序、时间片分配等问题 ，为后续实现更复杂的调度算法提供了场景。

（三）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。

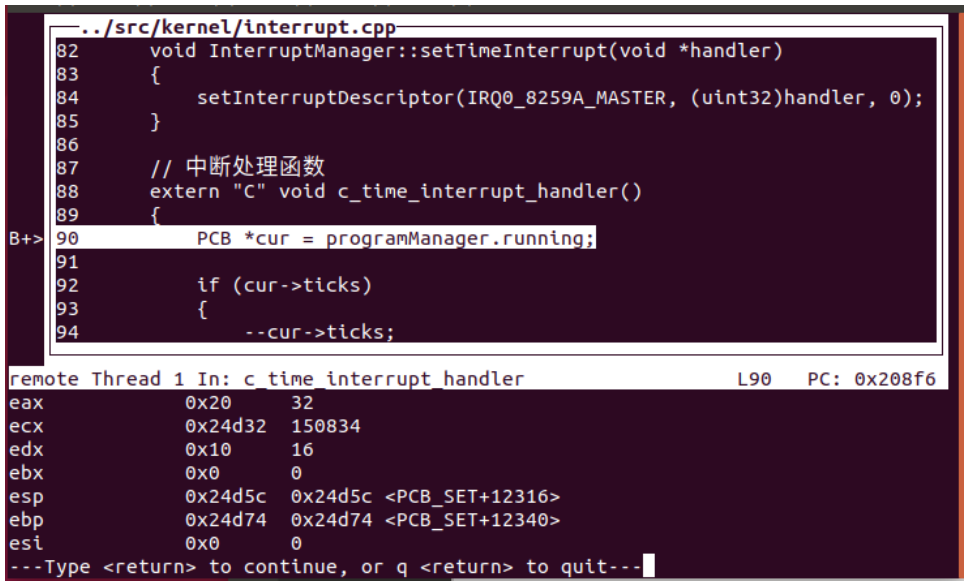
- 一个新创建的线程是如何被调度然后开始执行的。
- 一个正在执行的线程是如何被中断然后被换下处理器的，以及换上处理机后又是如何从被中断点开始执行的。

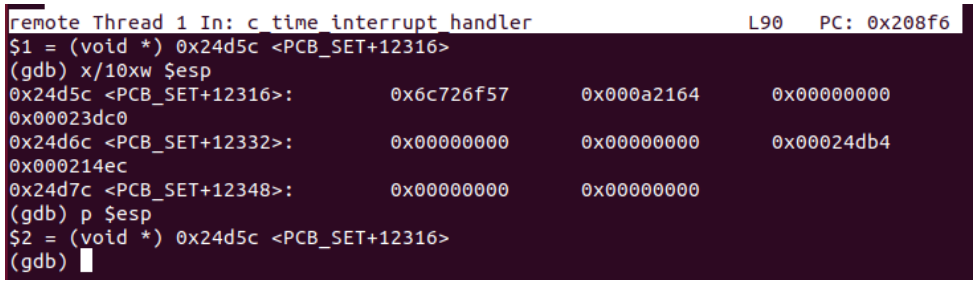
1. 对 assignment 2 的内容 debug

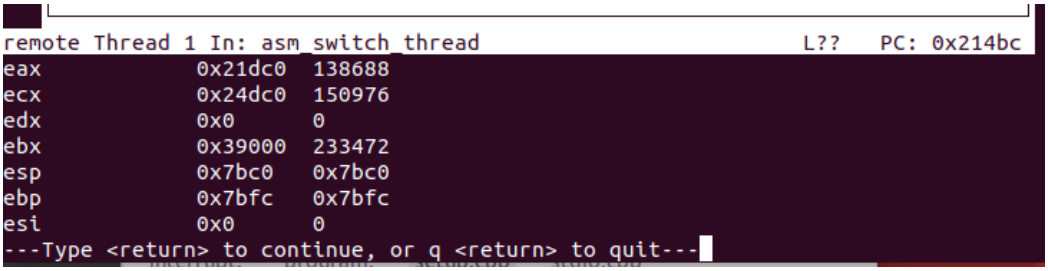
多个线程函数在任务二中已完成编写，只需对其 debug 即可。

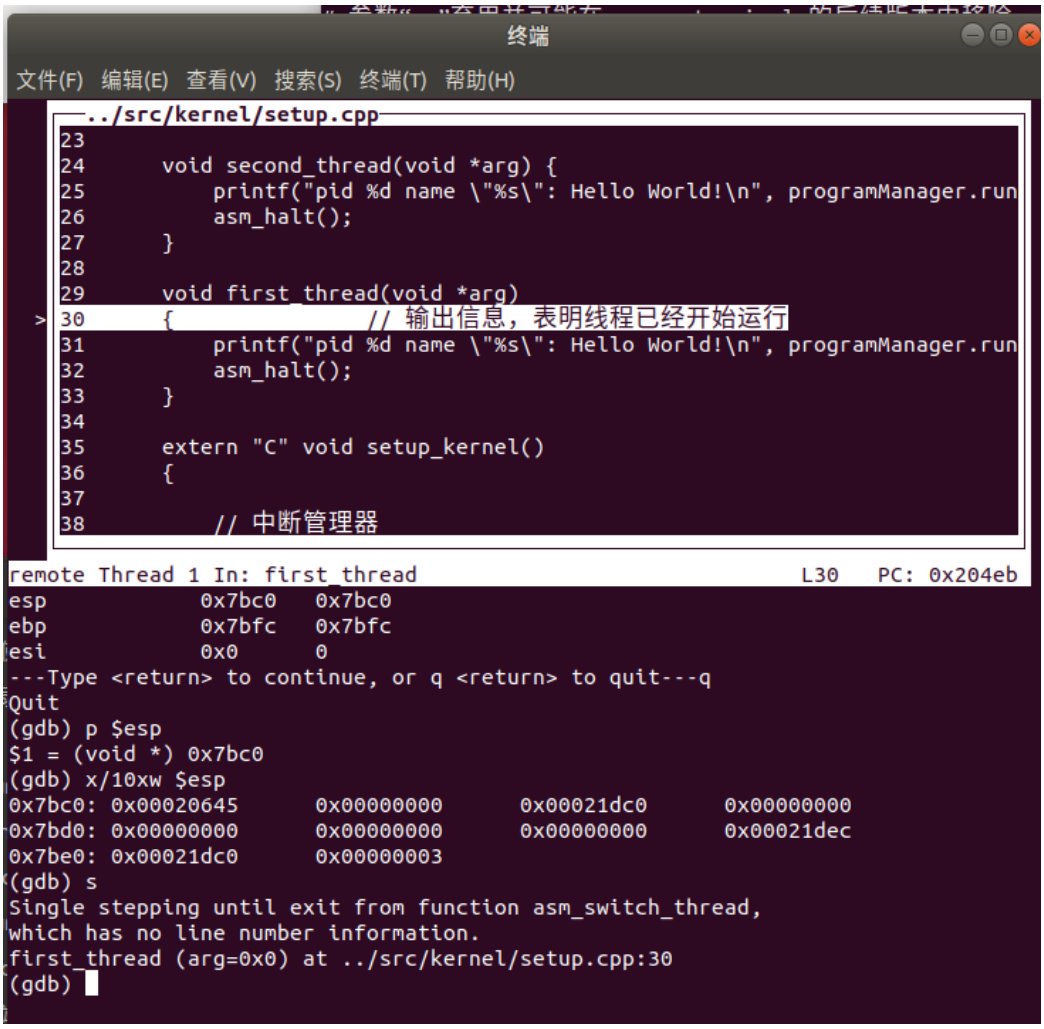
使用 gdb 调试主要用于观察线程切换前后栈、寄存器、PC（程序计数器，在 x86 架构中一般用 EIP 表示）等的变化，以理解线程并发执行的机制。调试的主要步骤为：设置断点、运行程序、观察寄存器和栈状态、继续执行、分析线程调度和切换过程、结束调试。

调试过程截图如下：









2. 结论

（1） 新创建线程的调度执行过程

① **线程创建**：通过 programManager.executeThread 函数创建新线程，该函数会为线程分配 PCB（进程控制块），初始化线程的相关信息，如线程函数指针、参数、优先级等，并将线程状态设置为 READY，加入到 programManager 的就绪队列 readyPrograms 中。

② **调度选择**：在 setup_kernel 函数中，手动将第一个线程从就绪队列中取出，将其状态设置为 RUNNING，并赋值给 programManager.running，然后通过 asm_switch_thread 函数进行

线程切换，让第一个线程开始执行。对于其他新创建的线程，它们会在就绪队列中等待被调度。当时钟中断发生时，在 c_time_interrupt_handler 函数中会调用 schedule 函数进行线程调度，schedule 函数会从就绪队列中选择一个合适的线程（这里简单按照优先级选择，优先级相同则按顺序选择），将其状态设置为 RUNNING，并更新 programManager.running 指针。

③ **线程执行**：asm_switch_thread 函数会进行上下文切换，将当前线程的寄存器值等上下文信息保存到其 PCB 中，然后恢复要执行线程的上下文信息，包括设置栈指针、程序计数器等，从而让新线程从其线程函数开始执行，在这个例子中就是执行 first_thread、second_thread 等函数中的代码，输出信息并执行 asm_halt 暂停。

（2） 线程中断与恢复执行

① **中断发生**：当时钟中断发生时，硬件会触发中断，跳转到中断处理程序。在 setup_kernel 函数中，通过 interruptManager.setTimeInterrupt 设置了时钟中断处理函数为 asm_time_interrupt_handler，该函数会保存当前线程的上下文，包括寄存器值等，将其保存到当前 PCB 中。然后调用 c_time_interrupt_handler 函数，在这个函数中会进行线程调度相关的操作，如更新时间片、检查是否需要进行线程切换等。如果当前线程的时间片用完或者有更高优先级的线程就绪，就会调用 schedule 函数选择一个新的线程来执行。

② **线程换下处理器**：schedule 函数会将当前线程的状态设置为 READY 或其他相应状态（如 BLOCKED 等，这里简单处理为 READY），并将其放回就绪队列。然后选择一个新的线程，将其状态设置为 RUNNING，并通过 asm_switch_thread 函数进行上下文切换，将新线程的上下文信息从其 PCB 中恢复到寄存器等，从而实现将当前线程换下处理器，让新线程开始执行。

③ **线程恢复执行**：当一个被中断的线程再次被调度到处理器上执行时，asm_switch_thread 函数会从该线程的 PCB 中恢复其之前保存的上下文信息，包括寄存器值、栈指针等。这样，线程就可以从上次被中断的地方继续执行，因为程序计数器（PC）的值也被恢复了，它会指向下一条要执行的指令，从而实现了从被中断点开始执行。

（四）Assignment 4：调度算法的实现

任务要求修改 assignment 2 中的代码，实现先来先服务算法。先来先服务调度算法的核心思想是按照线程创建的先后顺序依次执行线程，即先创建的线程

先获得处理器资源并执行。

修改后的关键代码如下：

```
// 先来先服务调度函数
void fcfs_schedule() {
    if (!programManager.readyPrograms.empty()) {
        ListItem *item = programManager.readyPrograms.front();
        PCB *nextThread = ListItem2PCB(item, tagInGeneralList);
        nextThread->status = RUNNING;
        programManager.readyPrograms.pop_front();
        programManager.running = nextThread;
        asm_switch_thread(0, nextThread);
    }
}
```

实现先来先服务算法的过程主要为：

①首先检查了 programManager.readyPrograms 就绪队列是否为空。如果不为空，说明有线程在等待调度执行；

②再从就绪队列的头部取出第一个线程节点 item，并通过 ListItem2PCB 函数将其转换为 PCB（进程控制块）指针 nextThread；

③将选中的线程状态设置为 RUNNING，表示该线程即将开始执行，然后从就绪队列中移除该线程节点，因为它即将进入运行状态；

④再将 programManager.running 指针指向选中的线程，标记其为当前正在运行的线程；

⑤调用 asm_switch_thread(0, nextThread) 函数进行线程上下文的切换，将处理器的控制权交给选中的线程，使其开始执行。

运行结果如图：

三、实验总结

本次实验主要涉及到以下知识点：

（1） 线程函数的编写

每个线程函数都有特定的功能，主要是打印当前线程的相关信息，并通过调用 programManager.exit() 来处理线程的退出操作，包括资源

释放和状态更新等。

（2） 进程/线程管理

进程和线程的创建、就绪队列的管理、线程状态的维护以及线程的调度等操作，是整个多线程程序的核心管理模块。