



中山大學
SUN YAT-SEN UNIVERSITY

操作系统实验 5

内核线程

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：钟旺烜

学生学号：23336342

实验地点：实验楼 B203

实验成绩：

报告时间：2025 年 5 月 3 日

Section 1 实验概述

●实验任务 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（提示：可以增加一些格式化输出类型，比如 `%f`、`%.f`、`%e` 等），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。

●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: 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。

●实验任务 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。

Section 2 实验步骤与实验结果

----- 实验任务 1 -----

●任务要求:

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（提示：可以增加一些格式化输出类型，比如%f、%f、%e等），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。

●思路分析:

我们在实验材料中所已经实现的 `printf` 函数上进行改进，增加了四个格式化输出类型：`%f`（浮点数输出），`%.f`（可设置小数位精度的浮点数输出），`%e`（科学记数法输出）和`%.e`（可设置小数位精度的科学记数法输出）。四个格式化输出类型的实现思路来源于已经实现的 `printf` 函数中，首先是判断输入格式的部分，其次是实现格式化输出的部分。前者可以通过 `switch` 函数判断字符串中`%`符号后的类型即可。后者我们分别通过构建 `ftos` 和 `ftoes` 两个函数来实现对相应数据类型转化为字符串以便显示屏输出。

●实验步骤:

首先我们在实验资料已经实现的解析`%`参数的代码块中添加识别`%f`、`%e`、`%.f`和`%.e`参数的代码，新添加的参数的 `fmt` 的解析过程的代码如下：

```
        case 'f':
        case 'e':
        {
            float tempf = (float)va_arg(ap, double); //使用 va_arg 来找到可变参数列表中的 double 值
            if(tempf < 0) //如果输入为负数，则输出负号
            {
                counter += printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
                tempf = -tempf;
            }
            if(fmt[i] == 'f') //如果是参数%.f，调用 ftos 函数；反之，调用 ftoes 函数
                ftos(number, tempf);
            else
                ftoes(number, tempf);

            for (int j = 0; number[j]; ++j)
            {
                counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
            }
            break;
        }
        case '.': //处理%.f 和%.e 参数
        {
            i++;
```

```

        char precision_char[33]; // 储存输入小数位精确度的字符数组
        uint32 precision = 0;
        int j = 0;
        while('0' <= fmt[i] && fmt[i] <= '9') // 读取小数点后设置的精确度数字
        {
            precision_char[j++] = fmt[i];
            i++;
        }
        precision_char[j] = '\0';
        stoi(precision_char, precision); // 将精确度字符串转化为整型数据类型 int

        switch(fmt[i])
        {
            case 'f':
            case 'e':
            {
                float tempf = (float)va_arg(ap, double);

                if(tempf < 0)
                {
                    counter += printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
                    tempf = -tempf;
                }
                if(fmt[i] == 'f')
                    ftos(number, tempf, precision); // 将上面得到的精确度传递给 ftos 和 ftoes
                Else 函数
                    ftoes(number, tempf, precision);

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

```

我们可以看到在%f和%.f中都是用到了ftos函数，而在%e和%.e中也使用到了ftoes函数。其实%f和%.f参数在实现的功能上只有显示小数位精确度的不同，我们默认%f参数显示的浮点数小数位精确度为6位，%.f可以实现的小数位精确度为1-9位。因此我们设计的ftos函数有三个参数，输出的字符串头指针，输入的浮点数和小数位精确度。在%f的fmt解析过程中，我们不向ftos函数中传入小数位精确度的值，则ftos函数默认的小数位精确度的值为6。而在%.f的fmt解析过程中，我们通过读取%.后输入的具体保留小数位的值来设置ftos函数的小数位精确度，进行相应的操作。参数%e和%.e使用的ftoes函数的处理方式与参数%f和%.f类似。

下面是ftos函数的具体实现代码：

```

void ftos(char *numStr, float32 num,uint32 precision = 6)
{
    uint32 length,digits,tempi,temp;
    float32 tempf;

    length = 0;
    digits = 0;
    tempf = num;

    while(tempf - (int)tempf != 0)//用 digits 来储存浮点数小数点后的位数
    {
        digits += 1;
        tempf = tempf - (int)tempf;
        tempf *= 10;
    }

    tempf = num;

    for(uint32 i = 0;i < precision;i++)//将浮点数转化为整数来进行数字到字符串的转化
    {
        tempf *= 10;
    }
    if(digits > precision)//如果浮点数小数点后的位数大于精准度，那么要进行舍去操作，舍去操作的
    {
        规则设置为大于 5 时向前一位进一。
        if((tempf - (int)tempf) * 10 > 5)
        {
            tempf += 1;
        }
    }

    tempi = tempf;

    while(tempi) {
        temp = tempi % 10;
        tempi /= 10;
        numStr[length] = temp + '0';
        ++length;
        if(length == precision)//当字符串长度来到设置小数位精确度时，向字符串中添加小数点‘.’
        {
            numStr[length] = '.';
            ++length;
        }
    }
    if((int)num == 0)//如果输入的浮点数小于 1，那么要在小数点前添加一个 0
        numStr[length++] = '0';
    //将得到的字符串倒转
    for(uint32 i = 0, j = length - 1; i < j; ++i, --j) {
        swap(numStr[i], numStr[j]);
    }
    numStr[length] = '\\0';
}

```

类似的，下面是 `ftoes` 函数的实现代码：

```
void ftoes(char *numStr, float32 num, uint32 precision = 6)
{
    uint32 length,digits,tempi,temp;
    float32 tempf;
    int index = 0;
    length = 0;
    digits = 0;

    if((int)num == 0)//将浮点数转化为科学技术法的数字部分的表现形式，并计算相乘的 10 的次数
    {
        while((int)num == 0)//计算小于 1 的浮点数
        {
            num *= 10;
            index--;
        }
    }
    else
    {
        while((int)num / 10 != 0 && num != 10)//计算大于 10 的浮点数
        {
            num /= 10;
            index++;
        }
    }
    tempf = num;                //将转化为科学技术法形式的数字部分进行转化为字符串的操作，操作
    while(num - (int)num != 0)   流程同 ftos 函数中一致
    {
        digits += 1;
        num = num - (int)num;
        num *= 10;
    }

    for(uint32 i = 0;i < precision;i++)
    {
        tempf *= 10;
    }
    if(digits > precision)
    {
        if((tempf - (int)tempf) * 10 > 5)
        {
            tempf += 1;
        }
    }

    tempi = tempf;

    while(tempi) {
        temp = tempi % 10;
```

```

        tempi /= 10;
        numStr[length] = temp + '0';
        ++length;
        if(length == precision)
        {
            numStr[length] = '.';
            ++length;
        }
    }
    for(uint32 i = 0, j = length - 1; i < j; ++i, --j) {
        swap(numStr[i], numStr[j]);
    }

    numStr[length++] = 'e';//添加科学记数法标记符
    if(index < 0)
    {
        numStr[length++] = '-';
        index = -index;
    }
    char indexnum[33];
    itos(indexnum,index,10); 将10的次数从整数形式转化为字符串
    for (int j = 0; indexnum[j]; ++j)
    {
        numStr[length++] = indexnum[j];
    }

    numStr[length] = '\\0';
}

```

注意到我们在新添加的 `fmt` 的解析 `%e` 和 `%f` 中还加入了一个 `stoi` 函数,其作用是将读取到的精确度从字符串形式转化为整型数据类型,以便传递给 `ftos` 或 `ftoes` 函数。

`stoi` 函数实现代码如下:

```

void stoi(char *numStr, uint32 &num)
{
    for(uint32 i = 0;numStr[i]; ++i) {
        num *= 10;
        num += numStr[i] - '0';
    }
}

```

以上新添加和实现的三个函数都保存在 `stdlib.cpp` 中,我们在 `stdlib.h` 中添加相关的函数定义:

```

void ftos(char *numStr, float32 num,uint32 precision = 6);
void ftoes(char *numStr, float32 num, uint32 precision = 6);
void stoi(char *numStr, uint32 &num);

```

以上，我们完成了在实验资料提供的 `printf` 函数基础上添加四个格式化输出类型：`%f`（浮点数输出），`%.f`（可设置小数位精度的浮点数输出），`%e`（科学记数法输出）和`%.e`（可设置小数位精度的科学记数法输出）的过程。接下来我们在 `setup_kernal` 函数中书写相应的测试案例来验证我们实现的四个新的 `printf` 函数格式化输出类型的正确性：

在 `setup_kernal` 函数中添加的测试案例如下：

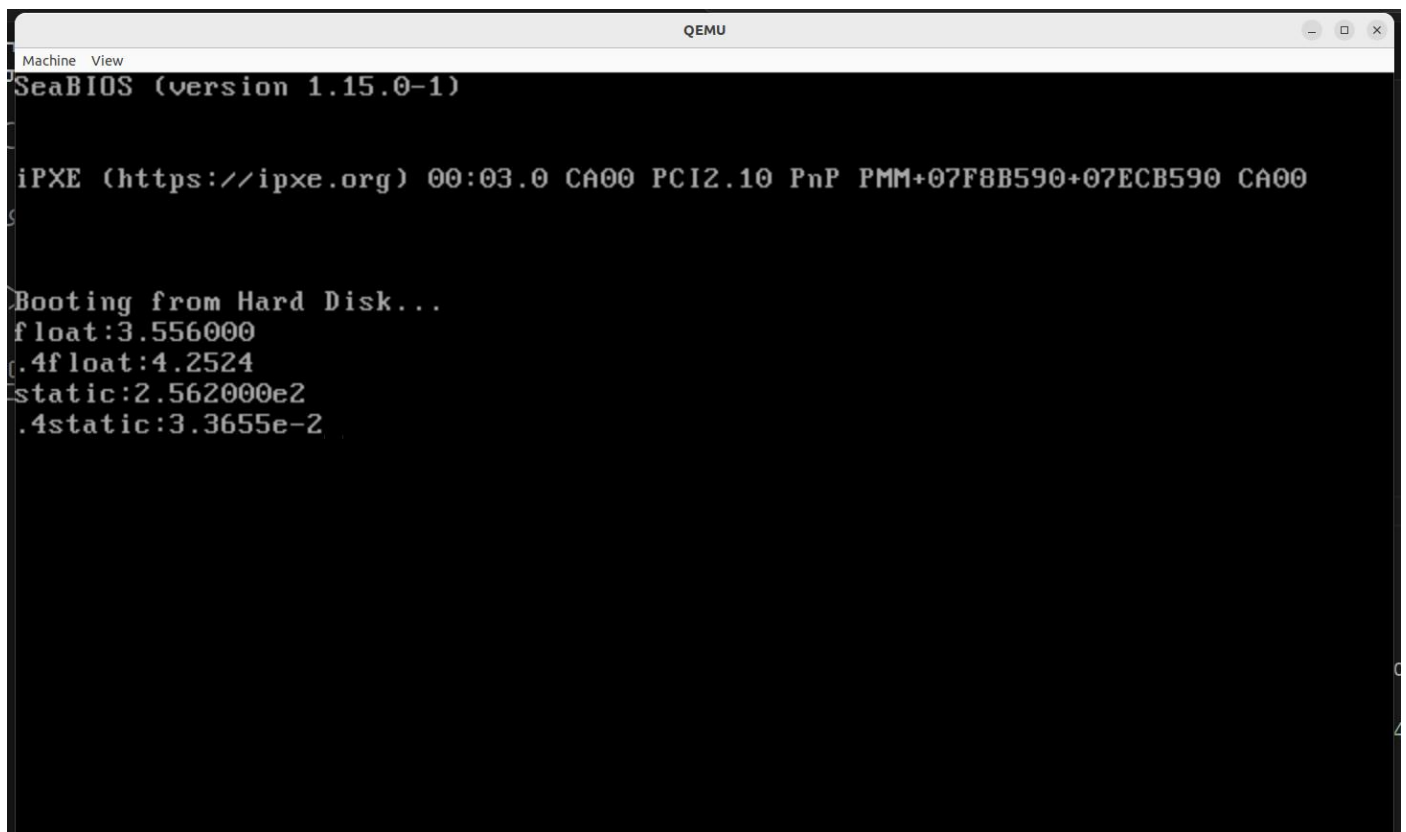
```
printf("float:%f\n.4float:%.4f\nstatic:%e\n.4static:%.4e",3.556,4.25243,256.2,0.0336548;
```

我们在屏幕中从上到下分别打印 3.556 的标准浮点数形式，4.25243 保留 4 位小数的形式，256.3 的标准科学技术法的形式，还有 0.0336548 的保留四位小数的科学记数法形式。

在 terminal 中输入相应 `make` 指令（`Makefile` 文件已经书写）即可启动 `qemu` 并观察到相应结果

●实验结果展示：通过执行前述代码，可得下图结果。

上面的测试案例在 `qemu` 显示屏上输出的结果如下图所示



```
Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00

Booting from Hard Disk...
float:3.556000
.4float:4.2524
static:2.562000e2
.4static:3.3655e-2
```


实验任务 2

●任务要求:

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。

●思路分析:

根据实验材料已经给出的设计的 PCB, 在其基础上进行适当修改, 通过实验资料中已经实现的创建线程的函数 `executeThread` 来实现线程并演示执行结果。

●实验步骤:

设计的 PCB 代码保存在 `thread.h` 中, 代码如下:

```
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 runningticks;     // 线程已执行时间
    ListItem tagInGeneralList; // 线程队列标识
    ListItem tagInAllList;   // 线程队列标识
};
```

接下来我们在 `setup_kernel` 函数中运用已经实现的创建线程的函数 `executeThread` 来创建一个线程:
第一个线程的代码如下所示:

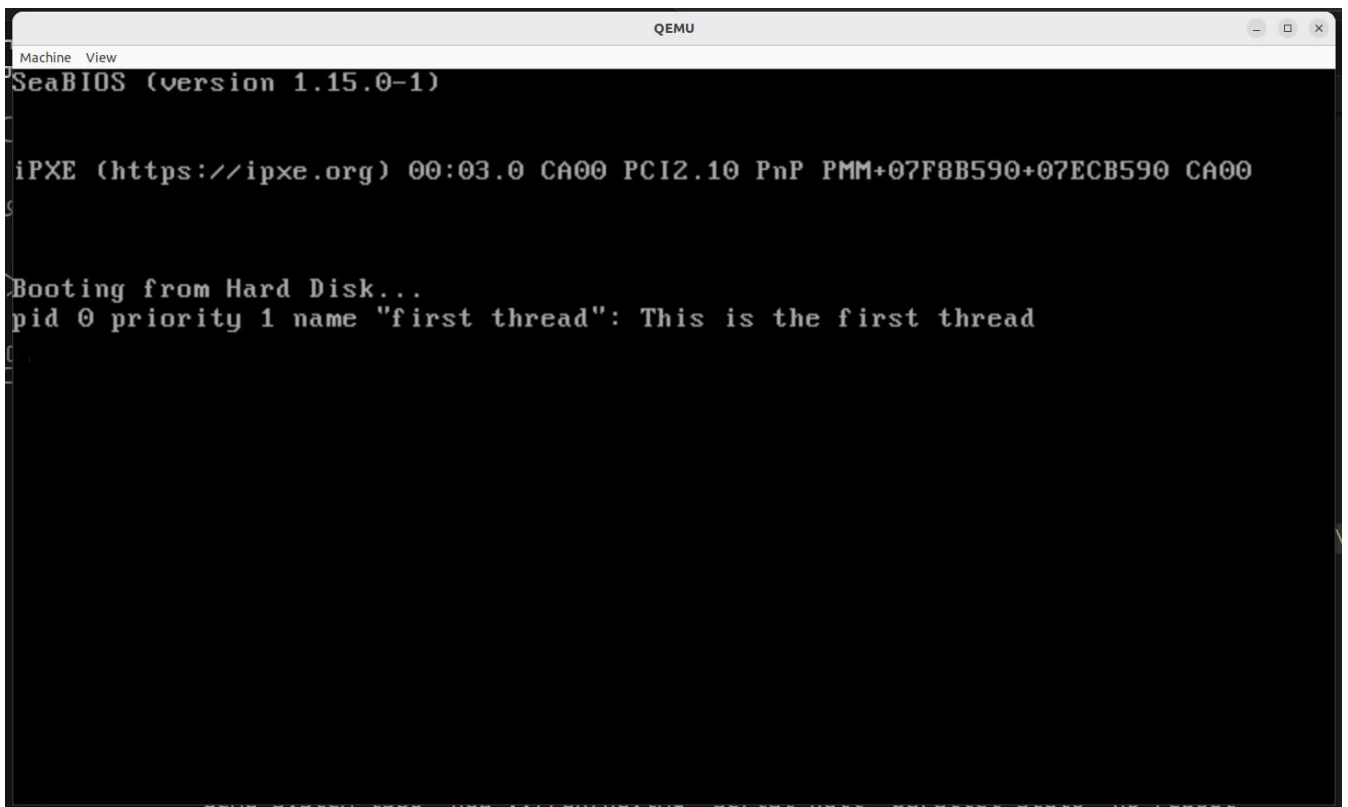
```
void first_thread(void *arg)
{
    printf("pid %d priority %d name \"%s\": This is the first thread\n",
        programManager.running->pid, programManager.running->priority, programManager.running->name);

    asm_halt();
}
```

在 `setup_kernel` 函数中创建并启动第一个线程, 可以在 `qemu` 显示屏上得到相应的输出。

●实验结果展示：通过执行前述代码，可得下图结果。

第一个线程的输出结果如下图所示：



```
Machine  View
QEMU

SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00

Booting from Hard Disk...
pid 0 priority 1 name "first thread": This is the first thread
```

实验任务 3

●任务要求:

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。

●思路分析:

我们通过编写 4 个线程函数，使用实验材料中所给出的时间片轮转算法（Round Robin, RR）进行线程的运行和调度。通过使用 gdb 跟踪跟线程调度和切换相关紧密的几个函数，来依次说明题目中所要求说明的两个过程。

对于第一个过程：一个新创建的线程是如何被调度然后开始执行的。可以观察由线程 1 第一次因时间片用尽切换到线程 2 的过程中，在 `schedule` 函数中是如何进行对应的操作并且将线程 2 调度执行的。

对于第二个过程：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。我们可以观察线程 1 切换到线程 2 的过程，再观察当线程 1 被再次调度回来的过程，来看看线程 1 是如何被中断然后被换下处理器以及换上处理器后如何从被中断点开始执行的。

使用 gdb 调试的过程图片以及相应的结果将在实验步骤中展开

●实验步骤:

首先我们编写 4 个线程函数，注意我们规定第一个线程不能返回。为了观察线程函数被换下处理器后重新调度回来的状态，我们编写的线程函数都将不进行返回操作，即将一直在处理器中不停的调度运行。

编写的线性函数代码保存在在 `setup.cpp` 文件中，内容如下：

```

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

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

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

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);
    if (!programManager.running->pid)
    {
        programManager.executeThread(second_thread, nullptr, "second thread",5);
        programManager.executeThread(third_thread, nullptr, "third thread", 3);
        programManager.executeThread(forth_thread, nullptr, "forth thread",10);
    }
    asm_halt();
}

```

我们所构建的 4 个线程函数均为打印相应线程中属性内容的字符串，注意到 2-4 线程是通过执行 1 线程后添加到线程列表中的。接下来我们通过这些线程函数来使用 gdb 调试观察题目所要求解释说明的两个过程：

首先我们知道，一个线程在被通过 ProgramManager 类中的 executeThread 函数创建后，其对应的线程队列标识便会放入到 allPrograms 和 readyPrograms 中，等待时钟中断来的时候，这个新创建的线程就可以被调度上处理器。

在线程 1 因为时间片耗尽而调用 schedule 函数进行线程调用时，如果线程 1 正处于运行态，那么其便会从运行态设置为就绪态，并行放到线程就绪队列的尾部，随后，线程调度器将读取线程就绪队列头部的线程队列标识，在我们调试的案例中也就是线程 2，然后将此线程标识符从线程就绪队列中弹出。通过线程 2 标识符使用 ListItem2PCB 宏定义读取线程 2 本体，将线程 2 设置为运行态，并通过 asm_switch_thread 函数来实现线程的转化，使处理器从线程 1 上转为执行线程 2。

```
../src/kernel/program.cpp
96     }
97
98     ListItem *item = readyPrograms.front();
99     PCB *next = ListItem2PCB(item, tagInGeneralList);
100    PCB *cur = running;
101    next->status = ProgramStatus::RUNNING;
102    running = next;
103    readyPrograms.pop_front();
104
B+> 105    asm_switch_thread(cur, next);
106
107    interruptManager.setInterruptStatus(status);
108 }
```

线程的所有信息都在线程栈中，只要我们切换线程栈就能够实现线程的切换，线程栈的切换实际上就是将线程的栈指针放到 esp 中。

在跳转进入 asm_switch_thread 函数中后，我们观察切换线程前后寄存器、栈以及 PC 的变化，来观察线程二的栈指针是否放到了 esp 中：

```
../src/utils/asm_utils.asm
22 ; void asm_switch_thread(PCB *cur, PCB *next);
23 asm_switch_thread:
> 24    push ebp
25    push ebx
26    push edi
27    push esi
28
29    mov eax, [esp + 5 * 4]
30    mov [eax], esp ; 保存当前栈指针到PCB中，以便日后恢复
31
32    mov eax, [esp + 6 * 4]
33    mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
34

remote Thread 1.1 In: asm_switch_thread          L24    PC: 0x214ec
eax          0x22e0c          142860
ecx          0x1              1
edx          0x0              0
ebx          0x0              0
esp          0x22d38          0x22d38 <PCB_SET+3928>
ebp          0x22d64          0x22d64 <PCB_SET+3972>
esi          0x0              0
--Type <RET> for more, q to quit, c to continue without paging--
```

可以看到，在进行切换线程操作前，esp 指针指向的地址为 0x22d38<PCB_SET + 3928>

```
../src/utils/asm_utils.asm
30     mov [eax], esp ; 保存当前栈指针到PCB中，以便日后恢复
31
32     mov eax, [esp + 6 * 4]
33     mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
34
> 35     pop esi
36     pop edi
37     pop ebx
38     pop ebp
39
40     sti
41     ret
42 ; int asm_interrupt_status();

remote Thread 1.1 In: asm_switch_thread L35 PC: 0x214fc
eax      0x22de0      142816
ecx      0x1          1
edx      0x0          0
ebx      0x0          0
esp      0x23dc4      0x23dc4 <PCB_SET+8164>
ebp      0x22d64      0x22d64 <PCB_SET+3972>
esi      0x0          0
--Type <RET> for more, q to quit, c to continue without paging--
```

由图可知，我们从原先栈指针中读取 asm_switch_thread 函数接收到的 cur 栈指针 stack 地址参数，储存在 eax 中，再将 eax 的值赋给 esp 指针，可以看到 esp 指针成功的切换到了 cur 栈指针的地址处，esp 指针指向的地址为 0x23dc4<PCB_SET + 8164>

接下来的 pop 语句会将 4 个 0 值放到 esi, edi, ebx, ebp 中。此时，栈顶的数据是线程需要执行的函数的地址 function。执行 ret 返回后，function 会被加载进 eip，从而使得 CPU 跳转到这个函数中执行。此时，进入函数后，函数的栈顶是函数的返回地址，返回地址之上是函数的参数，符合函数的调用规则。

```
34
35     pop esi
36     pop edi
37     pop ebx
38     pop ebp
39
> 40     sti
41     ret
42 ; int asm_interrupt_status();
43 asm_interrupt_status:

remote Thread 1.1 In: asm_switch_thread L40 PC: 0x21500
0x23dd4 <PCB_SET+8180>: 0x000207a9      0x00020671      0x00000000      0x00024dc4
0x23de4 <PCB_SET+8196>: 0x72696874      0x68742064      0x64616572      0x00000000
0x23df4 <PCB_SET+8212>: 0x00000000      0x00000002      0x00000003      0x00000002
0x23e04 <PCB_SET+8228>: 0x0000001e      0x00000000      0x00031e08      0x00024e0c
0x23e14 <PCB_SET+8244>: 0x00022e14      0x00024e14      0x00000000      0x00000000
0x23e24 <PCB_SET+8260>: 0x00000000      0x00000000      0x00000000      0x00000000
0x23e34 <PCB_SET+8276>: 0x00000000      0x00000000      0x00000000      0x00000000
--Type <RET> for more, q to quit, c to continue without paging--
```


我们下面给出我们编写的几个线程函数的地址：

```
remote Thread 1.1 In: setup_kernel L42 PC: 0x20870
Symbol "first_thread(void*)" is a function at address 0x207da.
(gdb) info address second_thread
Symbol "second_thread(void*)" is a function at address 0x207a9.
(gdb) info address third_thread
Symbol "third_thread(void*)" is a function at address 0x20778.
(gdb) info address forth_thread
Symbol "forth_thread(void*)" is a function at address 0x20747.
(gdb) |
```

可以看到，上面栈顶的数据是线程需要执行的线程 2 函数的地址 0x207a9。

下面我们来看线程 1 是如何被中断换下又是如何从被中断点重新开始执行的。

根据时间片轮转算法（Round Robin, RR）的原理，每个线程都有一个运行时间片，而在 `c_time_interrupt_handler` 函数中，其操作过程代码如下所示：

```
extern "C" void c_time_interrupt_handler()
{
    PCB *cur = programManager.running;

    if (cur->ticks)
    {
        --cur->ticks;
        ++cur->ticksPassedBy;
    }
    else
    {
        programManager.schedule();
    }
}
```

可以看到，每一次时间中断后，`c_time_interrupt_handler` 函数都会将当前运行线程的 `cur->ticks`（即当前程序的时间片数）减一，当线程的时间片数量减为 0 时，就会调用线程调度函数，从而是当前运行线程从处理器上中断并放入就绪线程队列的尾部。

我们从上面线程 1 切换到线程 2 的过程中可以看到，我们已经提前将中断时线程 1 的 `esp` 指针指向的地址，即 `0x22d28<PCB_SET + 3912>` 储存在了线程一的 `stack` 指针所指向的地址处，那么在切换回线程 1 时，我们重新读取线程 1 的 `stack` 指针指向的地址，将其重新赋给 `esp` 指针，这首 `esp` 指针便成功的跳转回了线程 1 的 `stack` 指针地址：

```
../src/utils/asm_utils.asm
25     push ebx
26     push edi
27     push esi
28
29     mov eax, [esp + 5 * 4]
30     mov [eax], esp ; 保存当前栈指针到PCB中，以便日后恢复
31
32     mov eax, [esp + 6 * 4]
33     mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
34
> 35     pop esi
36     pop edi
37     pop ebx

remote Thread 1.1 In: asm_switch_thread L35 PC: 0x214fc
eax      0x21de0      138720
ecx      0x0          0
edx      0x0          0
ebx      0x0          0
esp      0x22d28      0x22d28 <PCB_SET+3912>
ebp      0x25da8      0x25da8 <PCB_SET+16328>
esi      0x0          0
--Type <RET> for more, q to quit, c to continue without paging--
```

执行 4 个 pop 后，之前保存在线程栈中的内容会被恢复到这 4 个寄存器中，然后执行 ret 后会返回调用 asm_switch_thread 的函数，也就是 ProgramManager::schedule，然后在 ProgramManager::schedule 中恢复中断状态，返回到时钟中断处理函数，最后从时钟中断中返回，恢复到线程被中断的地方继续执行。

```
../src/utils/asm_utils.asm
31
32     mov eax, [esp + 6 * 4]
33     mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
34
35     pop esi
36     pop edi
37     pop ebx
38     pop ebp
39
> 40     sti
41     ret
42 ; int asm_interrupt_status();
43 asm_interrupt_status:

remote Thread 1.1 In: asm_switch_thread L40 PC: 0x21500
0x22d38 <PCB_SET+3928>: 0x00020657      0x00021de0      0x00022de0      0x00022d74
0x22d48 <PCB_SET+3944>: 0x000202db      0x00021de0      0x00022de0      0x00022e0c
0x22d58 <PCB_SET+3960>: 0x00022d74      0x000210ca      0x00031e08      0x00022d94
0x22d68 <PCB_SET+3976>: 0x00020289      0x00031e00      0x00023e0c      0x00022da4
0x22d78 <PCB_SET+3992>: 0x0002051c      0x00031df4      0x00000001      0x00001000
0x22d88 <PCB_SET+4008>: 0x00021de0      0x00000000      0x00024de0      0x00022dd4
0x22d98 <PCB_SET+4024>: 0x0002151c      0x00000000      0x00000000      0x00022dd4
--Type <RET> for more, q to quit, c to continue without paging--
```


实验任务 4

●任务要求:

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。

●思路分析:

这里选择实现非抢占式的优先级调度算法，此算法的调度规则是，在每次进行线程调度时，都将就绪队列中处于就绪状态的线程中优先级最大（这里线程的 **Priority** 值越大，优先级越高）的线程调入处理器中运行。同时，为了防止线程一直运行而不进行调度导致堵塞，在 `c_time_interrupt_handler` 函数中设置了计算线程运行总时间的功能，每触发一次 `c_time_interrupt_handler` 函数，当前运行线程的总运行时间便会加 1，如果当前线程运行总时间超过了设置的最大值，就会立即调用线程调用函数，将此运行超时的线程从处理器上换下，将运行时间清零，放入就绪队列，再运行其他的线程。

●实验步骤:

首先对线程管理类中的 `schedule` 函数进行修改，代码保存在 `program.cpp` 中。修改后的代码如下：

```
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->runningticks = 0; // 将线程运行时间归零，重新加入就绪队列中
        readyPrograms.push_back(&(running->tagInGenerallist));
    }
    else if (running->status == ProgramStatus::DEAD)
    {
        releasePCB(running);
    }
}
```

```

int max_priority = 0;//最高优先级
ListItem *item = readyPrograms.front();
ListItem *run = item;
PCB *next = ListItem2PCB(item, tagInGeneralList);
while(item)//查看线程就绪链表中的最高优先级线程
{
    PCB *temp = ListItem2PCB(item, tagInGeneralList);
    if(temp->priority > max_priority && temp->status == ProgramStatus::READY)
    {
        max_priority = temp->priority;
        run = item;//将找到的最高优先级线程用临时索引取出
        next = ListItem2PCB(item, tagInGeneralList);
    }
    item = item->next;
}

PCB *cur = running;
next->status = ProgramStatus::RUNNING;
running = next;
readyPrograms.erase(run);

asm_switch_thread(cur, next);

interruptManager.setInterruptStatus(status);
}

```

随后我们修改 `c_time_interrupt_handler` 函数,来实现实验思路中所描述的超时中断机制:

```

extern "C" void c_time_interrupt_handler()
{
    PCB *cur = programManager.running;

    if (cur->runningticks < 100)
    {
        ++cur->runningticks;
    }
    else
    {
        programManager.schedule();
    }
}

```

接下来,我们编写几个线程函数来测试我们的调度算法。

首先创建一个不返回的线程 1 函数,因为此时就绪队列中只有线程 1,因此先调度线程 1 函数。又因为线程 1 没有进行返回操作,因此其会触发超时机制,从而从处理器上调下运行就绪队列的其他进程。

在线程 1 函数中,同时创建优先级不同的 3 个线程函数,具体的优先级和线程函数的代码保存在 `setup.cpp` 文件中,代码如下所示:

```

void forth_thread(void *arg) {
    printf("pid  %d  priority  %d  name  \"%s\":  This  is  the  forth  thread\n",
programManager.running->pid, programManager.running->priority, programManager.running->name);
    return;
}

void third_thread(void *arg) {
    printf("pid  %d  priority  %d  name  \"%s\":  This  is  the  third  thread\n",
programManager.running->pid, programManager.running->priority, programManager.running->name);
    return;
}

void second_thread(void *arg) {
    printf("pid  %d  priority  %d  name  \"%s\":  This  is  the  second  thread\n",
programManager.running->pid, programManager.running->priority, programManager.running->name);
    return;
}

void first_thread(void *arg)
{
    // 第 1 个线程不可以返回
    printf("pid  %d  name  \"%s\":  This  is  first  thread!\n",  programManager.running->pid,
programManager.running->name);
    if (!programManager.running->pid)
    {
        programManager.executeThread(second_thread, nullptr, "second thread",5);
        programManager.executeThread(third_thread, nullptr, "third thread", 3);
        programManager.executeThread(forth_thread, nullptr, "forth thread",10);
    }
    asm_halt();
}

```

可以从上面代码中看到，我们在线程函数 1 中创建的 3 个线程函数的优先级分别是：

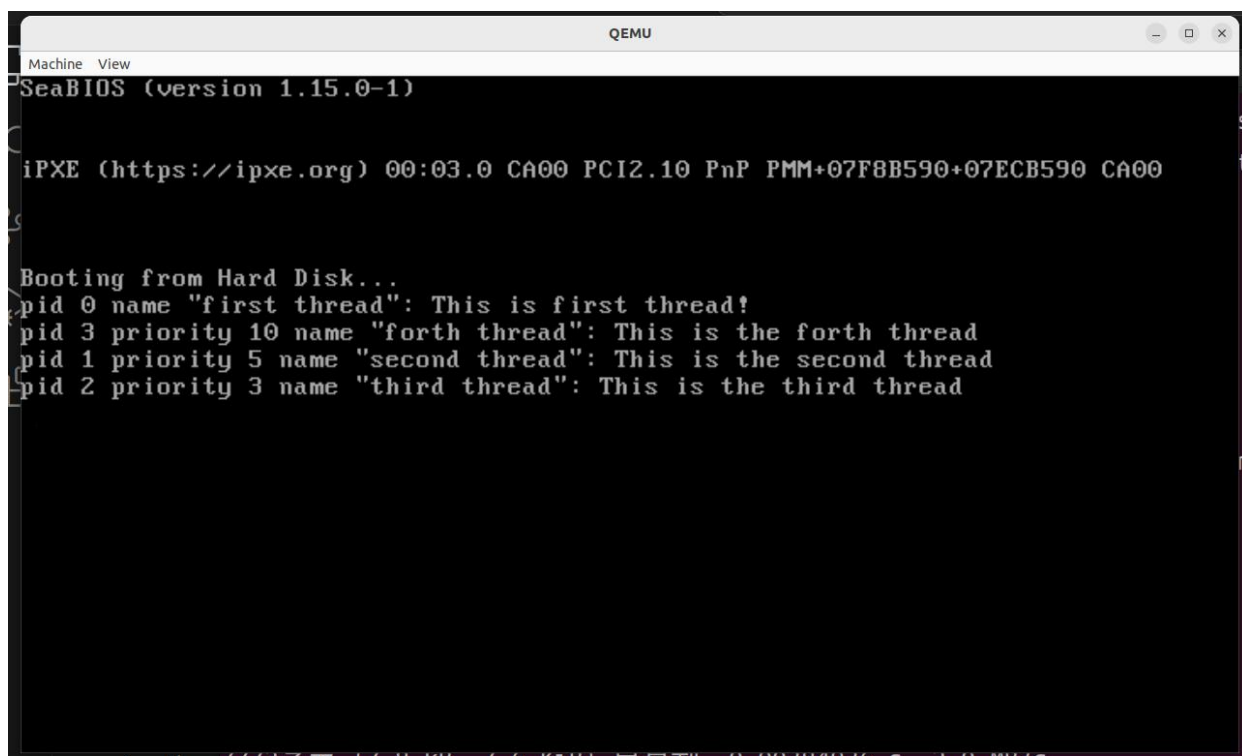
Second thread: priority: 5 ; Third thread: priority: 3 ; Forth thread: priority: 10

因此在线程 1 后执行调度算法时，先执行线程 4，在执行线程 2，最后执行线程 3。

（考虑到线程 1 没有设置返回操作，为了防止优先调度算法一直调度线程 1，设置线程 1 的优先级为最低的 1）

●实验结果展示：通过执行前述代码，可得下图结果。

执行上面设计好的线程函数后，线程调度算法的执行结果如下：



```
Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00

Booting from Hard Disk...
pid 0 name "first thread": This is first thread!
pid 3 priority 10 name "forth thread": This is the forth thread
pid 1 priority 5 name "second thread": This is the second thread
pid 2 priority 3 name "third thread": This is the third thread
```

Section 3 实验总结与心得体会

1、通过对 `printf` 函数的构建，对于如何使用可变参数列表，如何使用宏定义来寻找对应参数地址以及抽象包装寻址操作，以及如何实现 `printf` 函数的各种参数功能有了深刻理解并进行了实际的实验操作，实现的 `printf` 函数对以后的调试操作有了更加方便的工具。

2、对于如何设计 PCB，如何构建线程，以及线程的基本运行逻辑有了清晰的认识。做到了能够自行设计相应的 PCB 并进行对应的赋值操作，并且知道了如何创建新的线程，又是如何退出线程。

3、学习了时间片轮转线程调度算法的基本逻辑以及实现设计代码，对应如何将线程进行中断调下处理器后切换到其他线程的操作函数有了清晰的认识，对于如何将换下的线程重新调回处理器并恢复运行状态有了清晰的认识。

4、学习了其他线程调度算法的操作过程以及实现关键操作，并且具体实现了优先级调度算法，在调度算法的实现运行方面有了更深的理解。

5、编写优先级调度算法时，发现如果线程函数没有返回操作，那么其将一直占用处理器而不进行调度。因此利用设计的 PCB 中的运行时间这一参数以及时间中断处理函数，设计了超时中断处理过程，使处理器能够从超时的线程中跳出，处理其他线程。

Section 4 对实验的改进建议和意见

在处理器如何进行线程切换的说明上，可以进行更加详细的说明。

在设计线程调度的时间片轮转算法中，对于为什么第一个线程不进行返回操作进行解释说明。

Section 5 附录：参考资料清单

[lab5 · apshuang/SYSU-2025-Spring-Operating-System - 码云 - 开源中国](#)

<https://zhuanlan.zhihu.com/p/97071815>