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:　　**Lab5 可变参数机制与内核线程**

专业名称:　　计算机科学与技术

学生姓名:　　孙凯

学生学号:　　23336212

实验地点:　　实验中心B202

实验时间:　　2025.4.28

一、实验要求:

在本次实验中,我们将会学习到C语言的可变参数机制的实现方法。在此基础上,我们会揭开可变参数背后的原理,进而实现可变参数机制。实现了可变参数机制后,我们将实现一个较为简单的printf函数。此后,我们可以同时使用printf和gdb来帮助我们debug。

本次实验另外一个重点是内核线程的实现,我们首先会定义线程控制块的数据结构——PCB。然后,我们会创建PCB,在PCB中放入线程执行所需的参数。最后,我们会实现基于时钟中断的时间片轮转(RR)调度算法。在这一部分中,我们需要重点理解 `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。

具体内容如下:

1. 可变参数机制的实现
2. 实现printf函数
3. 内核线程的创建和调度

二、预备知识和实验环境:

预备知识:qemu+gdb调试方法,线程调度算法,C++语法

实验环境:

```
1. 虚拟机版本/处理器型号: Ubuntu 20.04 LTS
2. 代码编辑环境: Vscode+nasm+C/C++插件+qemu仿真平台
3. 代码编译工具: gcc/g++ (64位)
4.  重要三方库信息: 无
5. 代码程序调试工具: gdb
```

三、实验任务:

- Assignment 1 printf的实现

- Assignment 2 线程的实现
- Assignment 3 时钟中断的实现
- Assignment 4 调度算法的实现

四、实验步骤和实验结果：

Assignment 1： printf函数的实现

代码在 `lab5/assignment1`

- 任务要求：

学习可变参数机制，然后实现printf函数，你可以在材料中 （src/3）的printf上进行改进，或者从头开始实现自己的printf 函数。结果截图保存并说说你是怎么做的。

- 实验思路：仿照实验操作仓库的步骤，搭建 `printf` 函数
  - 在原基础的 `printf` 函数上，添加八进制输出格式 `%o`，无符号整数输出格式 `%u` 和浮点数输出格式 `%f` 三种功能。

- 实验步骤：

1. 修改 `stdio.cpp` 文件，添加 `%o` `%u` `%f` 三种输出格式。

`%o` 的实现思路和前面的输出格式基本相同，只需要修改一下 `itos` 函数的格式参数8；

`%u` 的实现思路和 `%d` 相似，只需要删除一下负号的输出逻辑；

`%f` 的实现思路为将浮点数分解为整数部分和小数部分，然后分别处理，这里保留了6位小数的精度。

```
case 'u':
    {
        unsigned int temp = va_arg(ap, unsigned int);
        itos(number, temp, 10);
        for (int j = 0; number[j]; ++j)
        {
            counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
        }
        break;
    }
case 'o':
    {
        unsigned int temp = va_arg(ap, unsigned int);
        itos(number, temp, 8);
        for (int j = 0; number[j]; ++j)
        {
            counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
        }
        break;
    }
case 'f':
    {
        double temp = va_arg(ap, double);
        int int_part = (int)temp;
```

```
                double frac_part = temp - int_part;
                if (temp < 0)
                {
                    counter += printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
                    int_part = -int_part;
                    frac_part = -frac_part;
                }
                itos(number, int_part, 10);
                for (int j = 0; number[j]; ++j)
                {
                    counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
                }
                counter += printf_add_to_buffer(buffer, '.', idx, BUF_LEN);
                // 保留6位小数
                for (int k = 0; k < 6; ++k)
                {
                    frac_part *= 10;
                    int digit = (int)frac_part;
                    counter += printf_add_to_buffer(buffer, '0' + digit, idx, BUF_LEN);
                    frac_part -= digit;
                }
                break;
            }
```

2. 在 `setup.cpp` 文件中添加测例

```
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"
        "print unsigned: %u\n"
        "print octal: %o\n"
        "print floating point: %f\n",
        'N', "Hello World!", -1234, 0x7abcdef0,4294967295U, 123, 3.141592);
```

3. `make && make run` 构建运行

实验结果如下:

可以看到输出最大无符号数 `4294967295`，十进制数 `123` 输出八进制格式为 `173`，输出浮点数 `3.141592`

Assignment 2 线程的实现

代码在 `lab5/assignment2`

- 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。

- 实验思路：在 `PCB` 结构体中加入相应的属性，然后在 `ProgramManager` 添加相应属性的处理。

- 实验步骤：

1. 修改 `thread.h` 文件，添加创建时间 `createTime` ，等待时间 `waitTime` 和调度策略 `schedulingPolicy`

这里创建了一个枚举结构，用于说明对应的线程调度策略。

```
enum SchedulingPolicy
{
    SCHEDULED_RR,
    SCHEDULED_SJF,
    SCHEDULED_PRIORITY,
    SCHEDULED_FCFS
};
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
    unsigned long createTime; // 线程创建时间
    unsigned long waitTime;   // 线程等待时间
    enum SchedulingPolicy schedulingPolicy; // 线程调度策略

};
```

2. 修改 `program.cpp` 文件，给 `ProgramManager` 类中添加 `schedulingPolicy` 的成员，用于对应线程调度策略。

```
    thread->createTime = time_counter++;
    thread->waitTime = 0;
    // 设置线程的调度策略
    // 0:RR, 1:SJF, 2:PRIORITY, 3:FCFS
    if(schedulingPolicy == 0)
        thread->schedulingPolicy = SchedulingPolicy::SCHEDULED_RR;
    else if(schedulingPolicy == 1)
        thread->schedulingPolicy = SchedulingPolicy::SCHEDULED_SJF;
    else if(schedulingPolicy == 2)
        thread->schedulingPolicy = SchedulingPolicy::SCHEDULED_PRIORITY;
    else if(schedulingPolicy == 3)
        thread->schedulingPolicy = SchedulingPolicy::SCHEDULED_FCFS;
    else
        thread->schedulingPolicy = SchedulingPolicy::SCHEDULED_RR;
```

3. 修改 `setup.cpp` 文件，添加两个进程，然后在打印函数中添加 `scheduling policy` 和 `creatTime` 两个输出内容，用于查看对应 `PCB` 的属性。

添加一个辅助函数 `get_scheduling_policy_name`，将调度策略枚举转换为字符串。

```
printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :Hello World!\n",
          programManager.running->pid,
          programManager.running->name,
          get_scheduling_policy_name(programManager.running->schedulingPolicy),
          programManager.running->createTime);


// 添加一个辅助函数，将调度策略枚举转换为字符串
const char* get_scheduling_policy_name(enum SchedulingPolicy policy) {
    switch (policy) {
        case SCHEDULED_RR:
            return "Round Robin";
        case SCHEDULED_SJF:
            return "Shortest Job First";
        case SCHEDULED_PRIORITY:
            return "Priority";
        case SCHEDULED_FCFS:
            return "First Come First Served";
        default:
            return "Unknown";
    }
}
```

4. `make && make run` 构建运行

实验结果如下：

可以看到对应的输出。

Assignment 3 时钟中断的实现

代码在 `lab5/assignment3`

- 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: b c_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。
     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。

- 实验思路：通过 `gdb` 调试查看 `first thread` 的创建调度和 `third thread` 如何中断之后换上 `first thread` 。

- 实验步骤：

1. 打断点到 `first thread` 创建语句，然后单步执行，进入 `executeThread` 函数，然后再进入 `allocatePCB` 函数，查看当前分配的 `PCB` 地址。

```
 64                     }
 65
 66                     // ^%^  &^  %^  '^  '^  %^  /^  (^   ^ %^  )^  (^
 67                     if (isNameSame) {
 68                         return -1;
 69                     }
 70                 }
 71
 72                 // ^&^    ^&^  '^   ^'^  &^  )^  &^  %^  (^  (^   ^  )^  %^
 73                 if (allPrograms.size() >= MAX_PROGRAM_AMOUNT)
 74                 {
 75                     return -1;
 76                 }
 77
 78
 79
 80                 // ^%^  $^  &^  /^  )^  &^  %^  %^  '^   ^'^  '^  (^   ^ '^  (^  &^  &^
 81                 bool status = interruptManager.getInterruptStatus();
 82                 interruptManager.disableInterrupt();
 83
 84                 // ^%^   )^  $^    ^)^  $^  $^  CB
B+>85                 PCB *thread = allocatePCB();
 86
 87                 if (!thread)
 88                     return -1;
 89
```

remote Thread 1.1 In: ProgramManager::executeThread   L85   PC: 0x20187
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000fff0 in ?? ()
(gdb) b setup.cpp:80
Breakpoint 1 at 0x207a4: file ../src/kernel/setup.cpp, line 80.
(gdb) c
Continuing.

Breakpoint 1, setup_kernel () at ../src/kernel/setup.cpp:80
(gdb) s
ProgramManager::executeThread (this=0x320c4 <programManager>, function=0x206b4 <first_thread(void*)>,
    parameter=0x0, name=0x21941 "first thread", priority=1, schedulingPolicy=0)
    at ../src/kernel/program.cpp:34
(gdb) b program.cpp:85
Breakpoint 2 at 0x20187: file ../src/kernel/program.cpp, line 85.
(gdb) c
Continuing.

Breakpoint 2, ProgramManager::executeThread (this=0x320c4 <programManager>,
    function=0x206b4 <first_thread(void*)>, parameter=0x0, name=0x21941 "first thread", priority=1,
    schedulingPolicy=0) at ../src/kernel/program.cpp:85
(gdb)

(gdb) s
ProgramManager::executeThread (this=0x320c4 <programManager>, function=0x206b4 <first_thread(void*)>,
    parameter=0x0, name=0x21941 "first thread", priority=1, schedulingPolicy=0)
    at ../src/kernel/program.cpp:87
(gdb) p thread
$1 = (PCB *) 0x220a0 <PCB_SET>
```

2. 打断点进入 `asm_switch_thread` 中，查看 `first thread` 线程调度前后寄存器的值。

由于 `first thread` 是手动创建并调度的，所以它之前的线程并不存在，对应 `esp和ebp` 寄存器的值是默认的。

此时 `esp和ebp` 保存上一个线程的状态。

```
┌─ ../src/utils/asm_utils.asm ──────────────────────────────────────────────
│     19                              db 0
│     20       ASM_IDTR dw 0
│     21              dd 0
│     22
│     23
│     24       ; void asm_switch_thread(PCB *cur, PCB *next);
│     25       asm_switch_thread:
│B+  26          push ebp
│     27          push ebx
│     28          push edi
│     29          push esi
│     30
│     31          mov eax, [esp + 5 * 4]
│     32          mov [eax], esp ; ^$^ ^ %^  %^  %^  &^  &^  )^  %^  CB^$^  /^  $^  $^  ^ ^&^  %^  &^  ^ %^
│     33
│>   34          mov eax, [esp + 6 * 4]
│     35          mov esp, [eax] ; ^&^  &^  &^  %^  '^  $^  ur^&^  %^  &^  %^  ext^&^
│     36
│     37          pop esi
│     38          pop edi
└───────────────────────────────────────────────────────────────────────────
remote Thread 1.1 In: asm_switch_thread
eax            0x0                   0
ecx            0x220a0               139424
edx            0x0                   0
ebx            0x39000               233472
esp            0x7bc0                0x7bc0
ebp            0x7bfc                0x7bfc
esi            0x0                   0
edi            0x0                   0
eip            0x21766               0x21766 <asm_switch_thread+10>
eflags         0x12                  [ IOPL=0 AF ]
```

调度之后，线程1的栈地址 `esp` 为 `0x23084` 。


```
┌─ ../src/utils/asm_utils.asm ──────────────────────────────────────────────
│     28          push edi
│     29          push esi
│     30
│     31          mov eax, [esp + 5 * 4]
│     32          mov [eax], esp ; ^$^ ^ %^  %^  %^  &^  &^  )^  %^  CB^$^  /^  $^  $^  ^ ^&^  %^  &^  ^ %^
│     33
│     34          mov eax, [esp + 6 * 4]
│     35          mov esp, [eax] ; ^&^  &^  &^  %^  '^  $^  ur^&^  %^  &^  %^  ext^&^
│     36
│>   37          pop esi
│     38          pop edi
│     39          pop ebx
│     40          pop ebp
│     41
│     42          sti
│     43          ret
│     44       ; int asm_interrupt_status();
│     45       asm_interrupt_status:
│     46          xor eax, eax
│     47          pushfd
└───────────────────────────────────────────────────────────────────────────
remote Thread 1.1 In: asm_switch_thread
 {0x0, 0x0}, uint128 = 0x0}
xmm5          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 ti
mes>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 =
 {0x0, 0x0}, uint128 = 0x0}
xmm6          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 ti
mes>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 =
 {0x0, 0x0}, uint128 = 0x0}
xmm7          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 12 ti
mes>, 0x80, 0x1f, 0x0, 0x0}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x1f80, 0x0}, v4_int32 = {0x0, 0
--Type <RET> for more, q to quit, c to continue without paging--
x0, 0x0, 0x1f80}, v2_int64 = {0x0, 0x1f8000000000}, uint128 = 0x1f800000000000000000000000000000}
mxcsr         0x1f80                [ IM DM ZM OM UM PM ]
(gdb) n
(gdb) n
asm_switch_thread () at ../src/utils/asm_utils.asm:37
(gdb) info registers
eax            0x220a0               139424
ecx            0x220a0               139424
edx            0x0                   0
ebx            0x39000               233472
esp            0x23084               0x23084 <PCB_SET+4068>
ebp            0x7bfc                0x7bfc
esi            0x0                   0
edi            0x0                   0
eip            0x2176c               0x2176c <asm_switch_thread+16>
eflags         0x12                  [ IOPL=0 AF ]
```

3. 打断点到 `c_time_interrupt_handler`，然后查看时间片和时间片剩余，观察线程如何运行。

```
(gdb) b c_time_interrupt_handler
Breakpoint 2 at 0x20afc: file ../src/kernel/interrupt.cpp, line 89.
(gdb) c
Continuing.

Breakpoint 2, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:89
(gdb) n
(gdb) n
(gdb) p cur->ticks
$1 = 10
(gdb) p cur->ticksPassedBy
$2 = 0
(gdb)
```

```
(gdb) b interrupt.cpp:99
Breakpoint 4 at 0x20b38: file ../src/kernel/interrupt.cpp, line 99.
(gdb) c
Continuing.

Breakpoint 4, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:99
(gdb) p cur->ticks
$3 = 0
(gdb) p cur->ticksPassedBy
$4 = 10
(gdb)
```

当时间片为0时，Assignment 4 调度算法的实现打断点到 `asm_switch_thread` 查看当前线程寄存器的值和线程调度之后的线程寄存器的值。

可以看调度之前先保存线程 `first thread` 当前的状态，`esp` 和 `ebp` 为 `0x22fe8` 和 `0x23024`

```
 ┌─../src/utils/asm_utils.asm────────────────────────────────────┐
      17            extern c_time_interrupt_handler
      18            ASM_UNHANDLED_INTERRUPT_INFO db 'Unhandled interrupt happened, halt...'
      19                                db 0
      20            ASM_IDTR dw 0
      21                    dd 0
      22
      23
      24            ; void asm_switch_thread(PCB *cur, PCB *next);
      25            asm_switch_thread:
 B+ 26                push ebp
      27                push ebx
      28                push edi
      29                push esi
      30
      31                mov eax, [esp + 5 * 4]
      32                mov [eax], esp ; ^$^   ^ %^  %^  %^  &^  &^  )^  %^  CB^$^  /^  $^  $^   ^ ^&^  %^  &^   ^ %^
      33
     >34                mov eax, [esp + 6 * 4]
      35                mov esp, [eax] ; ^&^  &^  &^  %^  '^  $^  ur^&^  %^  &^  %^  ext^&^
      36
remote Thread 1.1 In: asm_switch_thread
$4 = 10
(gdb) s
ProgramManager::schedule (this=0x320c4 <programManager>) at ../src/kernel/program.cpp:140
(gdb) b asm_switch_thread
Breakpoint 5 at 0x2175c: file ../src/utils/asm_utils.asm, line 26.
(gdb) c
Continuing.

Breakpoint 5, asm_switch_thread () at ../src/utils/asm_utils.asm:26
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) info registers
eax            0x220a0            139424
ecx            0x22f92            143250
edx            0x8                8
ebx            0x0                0
esp            0x22fe8            0x22fe8 <PCB_SET+3912>
ebp            0x23024            0x23024 <PCB_SET+3972>
esi            0x0                0
edi            0x0                0
eip            0x21766            0x21766 <asm_switch_thread+10>
eflags         0x2                [ IOPL=0 ]
```

调度之后，线程 `second thread` 的栈地址为 `0x24084`

```
  ../src/utils/asm_utils.asm
B+ 26              push ebp
   27              push ebx
   28              push edi
   29              push esi
   30
   31              mov eax, [esp + 5 * 4]
   32              mov [eax], esp ; ^$^  ^ %^  %^  %^  &^  &^  )^  %^  CB^$^  /^  $^  $^    ^   ^&^  %^  &^
   33
   34              mov eax, [esp + 6 * 4]
   35              mov esp, [eax] ; ^&^  &^  &^  %^  '^  $^  ur^&^  %^  &^  %^  ext^&^
   36
  >37              pop esi
   38              pop edi
   39              pop ebx
   40              pop ebp
   41
   42              sti
   43              ret
   44          ; int asm_interrupt_status();
   45          asm_interrupt_status:

remote Thread 1.1 In: asm_switch_thread
cr8             0x0                 0
efer            0x0                 [ ]
xmm0            {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>
xmm1            {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>
xmm2            {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>
xmm3            {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>
xmm4            {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>
xmm5            {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>
xmm6            {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>
xmm7            {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 12 times>,
00000000000000000000000000}
mxcsr           0x1f80              [ IM DM ZM OM UM PM ]
(gdb) n
(gdb) n
asm_switch_thread () at ../src/utils/asm_utils.asm:37
(gdb) info registers
eax             0x230a0             143520
ecx             0x22f92             143250
edx             0x8                 8
ebx             0x0                 0
esp             0x24084             0x24084 <PCB_SET+8164>
ebp             0x23024             0x23024 <PCB_SET+3972>
esi             0x0                 0
edi             0x0                 0
eip             0x2176c             0x2176c <asm_switch_thread+16>
eflags          0x2                 [ IOPL=0 ]
```

中间second thread这里先略过，因为我们要查看线程如何从中断恢复运行，所以我直接调试 `third thread`

4. 这里忽略 `second thread` 和 `third thread` 之间的调度，因为和前面的调度基本一样，主要是查看一下线程如何从中断中恢复。

   打断点到 `asm_switch_thread`，然后查看 `third thread` 线程寄存器值，`esp` 和 `ebp` 为 `0x24fec` 和 `0x25028`

```
       ../src/utils/asm_utils.asm
        17          extern c_time_interrupt_handler
        18          ASM_UNHANDLED_INTERRUPT_INFO db 'Unhandled interrupt happened, halt...'
        19                   db 0
        20          ASM_IDTR dw 0
        21               dd 0
        22
        23
        24          ; void asm_switch_thread(PCB *cur, PCB *next);
        25          asm_switch_thread:
    B+ 26              push ebp
        27              push ebx
        28              push edi
        29              push esi
        30
        31              mov eax, [esp + 5 * 4]
        32              mov [eax], esp ; ^$^  ^ %^  %^  %^  &^  &^  )^  %^  CB^$^  /^  $^  $^  ^  ^&^  %^  &^  ^ %^
        33
       >34              mov eax, [esp + 6 * 4]
        35              mov esp, [eax] ; ^&^  &^  &^  %^  '^  $^  ur^&^  %^  &^  %^  ext^&^
        36
   remote Thread 1.1 In: asm_switch_thread
   Continuing.

   Breakpoint 11, ProgramManager::schedule (this=0x320c4 <programManager>) at ../src/kernel/program.cpp:140
   (gdb) b asm_switch_thread
   Breakpoint 12 at 0x2175c: file ../src/utils/asm_utils.asm, line 26.
   (gdb) c
   Continuing.

   Breakpoint 12, asm_switch_thread () at ../src/utils/asm_utils.asm:26
   (gdb) n
   (gdb) n
   (gdb) n
   (gdb) n
   (gdb) n
   (gdb) n
   (gdb) info registers
   eax            0x240a0              147616
   ecx            0x24f96              151446
   edx            0x8                  8
   ebx            0x2                  2
   esp            0x24fec              0x24fec <PCB_SET+12108>
   ebp            0x25028              0x25028 <PCB_SET+12168>
   esi            0x0                  0
   edi            0x0                  0
   eip            0x21766              0x21766 <asm_switch_thread+10>
   eflags         0x12                 [ IOPL=0 AF ]
```

调度之后，可以观察到 `esp` 为 `0x22fe8`，为 `first thread` 中断时保存的地址，说明中断线程从中断点恢复执行的调度成功。

```
../src/utils/asm_utils.asm
B+ 26              push ebp
   27              push ebx
   28              push edi
   29              push esi
   30
   31              mov eax, [esp + 5 * 4]
   32              mov [eax], esp ; ^$^ ^ %^ %^ %^ &^ &^ )^ %^ CB^$^ /^ $^ $^ ^ ^&^ %^ &^ ^ %^
   33
   34              mov eax, [esp + 6 * 4]
   35              mov esp, [eax] ; ^&^ &^ &^ %^ '^ $^ ur^&^ %^ &^ %^ ext^&^
   36
>B 37             pop esi
   38              pop edi
   39              pop ebx
   40              pop ebp
   41
   42              sti
   43              ret
   44           ; int asm_interrupt_status();
   45            asm_interrupt_status:
remote Thread 1.1 In: asm_switch_thread
cr8           0x0                0
efer          0x0                [ ]
xmm0          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int1
xmm1          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int1
xmm2          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int1
xmm3          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int1
xmm4          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int1
xmm5          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int1
xmm6          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int1
xmm7          {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 12 times>, 0x80, 0}
00000000000000000000000000}
mxcsr         0x1f80             [ IM DM ZM OM UM PM ]
(gdb) n
(gdb) n
asm_switch_thread () at ../src/utils/asm_utils.asm:37
(gdb) info registers
eax           0x220a0            139424
ecx           0x24f96            151446
edx           0x8                8
ebx           0x2                2
esp           0x22fe8            0x22fe8 <PCB_SET+3912>
ebp           0x25028            0x25028 <PCB_SET+12168>
esi           0x0                0
edi           0x0                0
eip           0x2176c            0x2176c <asm_switch_thread+16>
eflags        0x12               [ IOPL=0 AF ]
```

Assignment 4 调度算法的实现

代码在 `lab5/assignment4`

- 任务要求:

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
- 响应比最高优先算法
- 优先级调度算法
- 多级反馈队列调度算法

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。

- 任务思路: 在 `program.cpp` 中添加最短作业优先 `SJF`，先来先服务 `FCFS` 和优先级调度 `PRI` 三种调度算法；然后在 `setup.cpp` 中添加多个线程，用于测试算法思路。

- 任务步骤:

注意: `ProgramManager::executeThread` 函数中调度策略 `schedulingPolicy` 对应的调度算法为 `0:RR, 1:SJF, 2:PRIORITY, 3:FCFS`

1. 为了实现多种调度算法的选择，我这里修改了 `program.cpp` 中 `ProgramManager::schedule()` 函数，进行调度算法的选择。

```cpp
void ProgramManager::schedule(){
    if(running){
        switch (running->schedulingPolicy)
        {
            case SchedulingPolicy::SCHEDULED_RR:
                schedule_RR();
                break;
            case SchedulingPolicy::SCHEDULED_SJF:
                schedule_SJF();
                break;
            case SchedulingPolicy::SCHEDULED_PRIORITY:
                schedule_PRI();
                break;
            case SchedulingPolicy::SCHEDULED_FCFS:
                schedule_FCFS();
                break;
            default:
                break;
        }
    }
}
```

2. 添加先来先服务调度算法 `FCFS`

```cpp
//先进先服务调度
void ProgramManager::schedule_FCFS()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    if (readyPrograms.size() == 0)
    {
        interruptManager.setInterruptStatus(status);
        return;
    }

    if (running->status == ProgramStatus::RUNNING)
    {
        running->status = ProgramStatus::READY;
        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();
    // 显示进程切换信息
    //printf("pid %d is stop, and pid %d is continue...\n", cur->pid, next->pid);
    asm_switch_thread(cur, next);

    interruptManager.setInterruptStatus(status);
}
```

在 `setup.cpp` 中添加线程测例:

```
oid fifth_thread(void*arg){
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
            programManager.running->name,
            get_scheduling_policy_name(programManager.running->schedulingPolicy),
            programManager.running->createTime);
}

void forth_thread(void*arg){
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
            programManager.running->name,
            get_scheduling_policy_name(programManager.running->schedulingPolicy),
            programManager.running->createTime);
            asm_halt();


}

void third_thread(void *arg) {
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
            programManager.running->name,
            get_scheduling_policy_name(programManager.running->schedulingPolicy),
            programManager.running->createTime);
            programManager.executeThread(forth_thread, nullptr, "forth thread", 1, 3);
            //asm_halt();


}

void second_thread(void *arg) {
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
            programManager.running->name,
```

```
            get_scheduling_policy_name(programManager.running->schedulingPolicy),
            programManager.running->createTime);
            programManager.executeThread(fifth_thread, nullptr, "fifth thread", 1, 3);
    //asm_halt();
}


void first_thread(void *arg)
{
    // 第1个线程不可以返回
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
            programManager.running->name,
            get_scheduling_policy_name(programManager.running->schedulingPolicy));
    if (programManager.running->pid==0)
    {
        programManager.executeThread(second_thread, nullptr, "second thread", 1, 3);
        programManager.executeThread(third_thread, nullptr, "third thread", 1, 3);

    }
    asm_halt();
}
```

根据代码，可以得知五个线程调度的顺序为 `first thread -> second thread -> third thread -> fifth` `thread -> forth thread`，运行得到线程顺序符合调度算法。

3. 添加短作业优先调度算法 `SJF`：

这里设置使用ticksPassedBy来预测未来执行时间（已执行时间越短，预期剩余时间可能越长），从而实现短作业的判断。

```cpp
//短作业优先调度
void ProgramManager::schedule_SJF()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    if (readyPrograms.size() == 0)
    {
        interruptManager.setInterruptStatus(status);
        return;
    }


    // 如果当前线程仍在运行，且不是因为时间片耗尽被调度的
    // SJF是非抢占式的，应该让当前线程继续执行，除非它主动放弃CPU或结束
    if (running->status == ProgramStatus::RUNNING && running->ticks > 0)
    {
        interruptManager.setInterruptStatus(status);
        return;
```

```cpp
    }

    // 处理当前线程状态
    if (running->status == ProgramStatus::RUNNING)
    {
        running->status = ProgramStatus::READY;
        readyPrograms.push_back(&(running->tagInGeneralList));
    }
    else if (running->status == ProgramStatus::DEAD)
    {
        releasePCB(running);
    }

    // 找到估计运行时间最短的线程
    ListItem *item = readyPrograms.front();
    PCB *next = ListItem2PCB(item, tagInGeneralList);
    PCB *cur = running;
    // 使用合适的指标来评估作业长度
    // 这里使用ticksPassedBy来预测未来执行时间（已执行时间越短，预期剩余时间可能越长）
    for (ListItem *i = readyPrograms.front(); i; i = i->next)
    {
        PCB *program = ListItem2PCB(i, tagInGeneralList);

        // 主要排序条件：预期总执行时间（这里用priority*10-ticksPassedBy估计）
        int current_estimate = next->priority * 10 - next->ticksPassedBy;
        int program_estimate = program->priority * 10 - program->ticksPassedBy;

        if (program_estimate < current_estimate ||
            (program_estimate == current_estimate && program->createTime < next-
>createTime))
        {
            // 如果预期执行时间相同，选择先创建的
            next = program;
        }
    }

    next->status = ProgramStatus::RUNNING;
    running = next;

    // 显示进程切换信息
    /*printf("pid %d pid %d (lass time: %d)\n",
            cur->pid, next->pid,
            next->priority * 10 - next->ticksPassedBy);*/

    readyPrograms.erase(&(next->tagInGeneralList));

    asm_switch_thread(cur, next);

    // 恢复中断状态
    interruptManager.setInterruptStatus(status);
}
```

在 `setup.cpp` 中添加线程测例:

由于这里设置的时间片与优先级有关，所以通过设置不同的优先级来设置不同的测例的运行时间。

短进程排序为：`first thread -> second thread -> fifth thread -> third thread -> forth thread`

```c
void fifth_thread(void*arg){
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
            programManager.running->name,
            get_scheduling_policy_name(programManager.running->schedulingPolicy),
            programManager.running->createTime);
}

void forth_thread(void*arg){
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
            programManager.running->name,
            get_scheduling_policy_name(programManager.running->schedulingPolicy),
            programManager.running->createTime);
            asm_halt();


}

void third_thread(void *arg) {
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
            programManager.running->name,
            get_scheduling_policy_name(programManager.running->schedulingPolicy),
            programManager.running->createTime);
            //asm_halt();

}

void second_thread(void *arg) {
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
            programManager.running->name,
            get_scheduling_policy_name(programManager.running->schedulingPolicy),
            programManager.running->createTime);
    //asm_halt();
}

void first_thread(void *arg)
{
    // 第1个线程不可以返回
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
            programManager.running->name,
            get_scheduling_policy_name(programManager.running->schedulingPolicy));
    if (programManager.running->pid==0)
    {
        programManager.executeThread(second_thread, nullptr, "second thread", 2, 1);
        programManager.executeThread(third_thread, nullptr, "third thread", 4, 1);
```

```cpp
        programManager.executeThread(fifth_thread, nullptr, "fifth thread", 3, 1);
        programManager.executeThread(forth_thread, nullptr, "forth thread", 5, 1);
        // 主动让出CPU
        programManager.running->status = READY;
        programManager.schedule();
    }
    asm_halt();
}
```

根据代码设置的线程顺序应该为 `first thread -> second thread -> fifth thread -> third thread -> forth thread`，运行得到线程顺序符合调度算法。

4. 添加优先级调度算法 `PRI`：

利用前面在 `PCB` 中优先级属性 `priorty` 来进行优先级判断，从而实现优先级调度。

```cpp
//优先级调度
void ProgramManager::schedule_PRI()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    if (readyPrograms.size() == 0)
    {
        interruptManager.setInterruptStatus(status);
        return;
    }

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

```
    {
        running->status = ProgramStatus::READY;
        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;

    // 找到优先级最高的进程
    for (ListItem *i = readyPrograms.front(); i; i = i->next)
    {
        PCB *program = ListItem2PCB(i, tagInGeneralList);
        if (program->priority > next->priority||(program->priority==next->priority&&program-
>createTime<next->createTime))
        {
            next = program;
        }
    }

    next->status = ProgramStatus::RUNNING;
    running = next;

    // 显示进程切换信息

    readyPrograms.erase(&(next->tagInGeneralList));

    asm_switch_thread(cur, next);

    interruptManager.setInterruptStatus(status);
}
```

在 `setup.cpp` 中添加测例:

这里设置优先级顺序为 `first thread -> forth thread -> second thread -> third thread -> fifth thread`

```
void fifth_thread(void*arg){
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
            programManager.running->name,
            get_scheduling_policy_name(programManager.running->schedulingPolicy),
            programManager.running->createTime);
            asm_halt();
}

void forth_thread(void*arg){
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
```

```
            programManager.running->name,
            get_scheduling_policy_name(programManager.running->schedulingPolicy),
            programManager.running->createTime);
        //asm_halt();



}

void third_thread(void *arg) {
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
            programManager.running->name,
            get_scheduling_policy_name(programManager.running->schedulingPolicy),
            programManager.running->createTime);
            programManager.executeThread(forth_thread, nullptr, "forth thread", 4, 2);
        //asm_halt();



}

void second_thread(void *arg) {
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
            programManager.running->name,
            get_scheduling_policy_name(programManager.running->schedulingPolicy),
            programManager.running->createTime);

    //asm_halt();
}

void first_thread(void *arg)
{
    // 第1个线程不可以返回
    printf("pid %d name \"%s\", scheduling policy: %s ,creatTime: %d :thread has done!\n",
            programManager.running->pid,
            programManager.running->name,
            get_scheduling_policy_name(programManager.running->schedulingPolicy));
    if (programManager.running->pid==0)
    {
        programManager.executeThread(second_thread, nullptr, "second thread", 3, 2);
        programManager.executeThread(third_thread, nullptr, "third thread", 2, 2);
        programManager.executeThread(fifth_thread, nullptr, "fifth thread", 1, 2);
        programManager.executeThread(forth_thread, nullptr, "forth thread", 4, 2);

        programManager.running->status = READY;
        programManager.schedule();
    }
    asm_halt();
}
```

根据实验结果，符合预期效果。
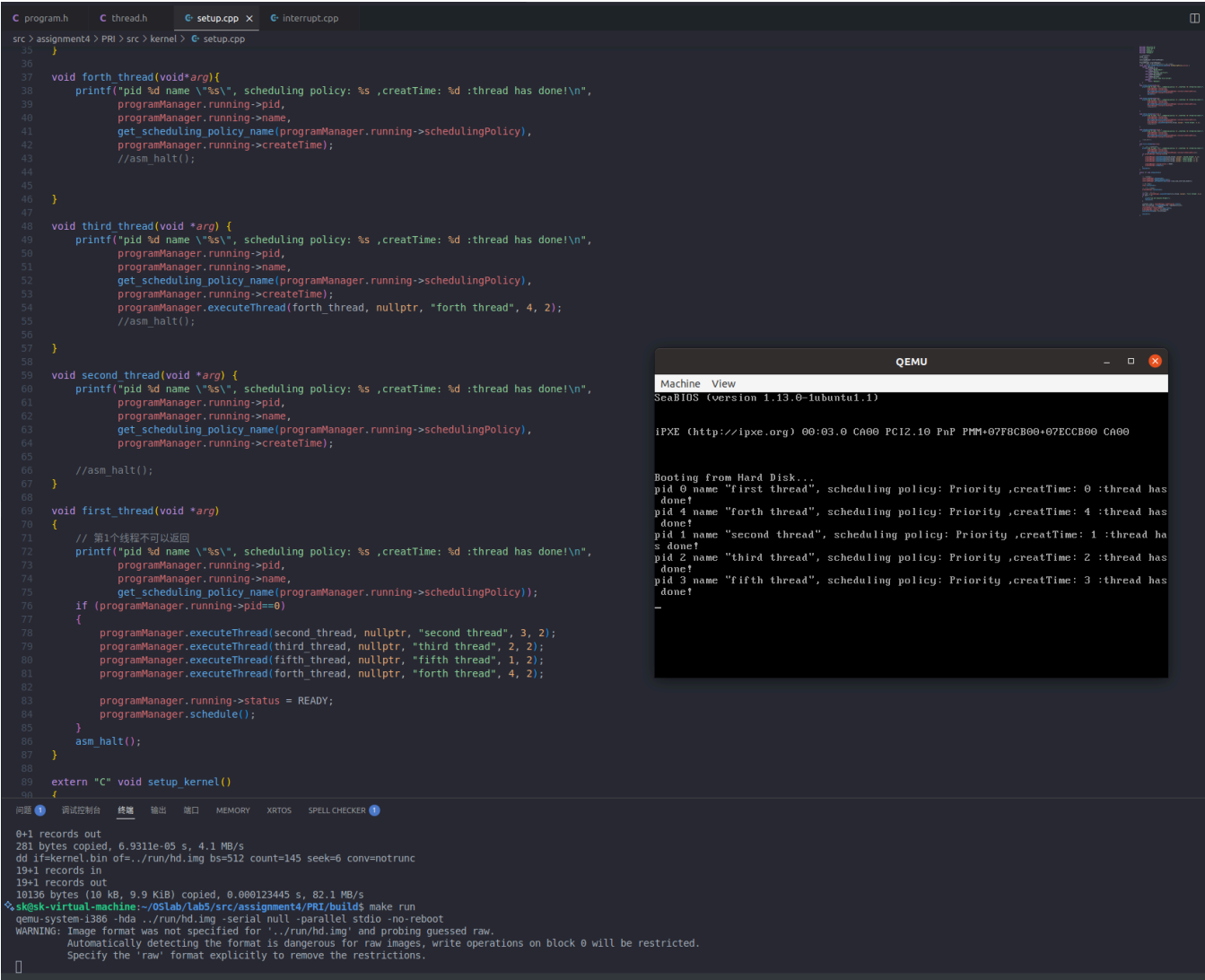
五、实验总结和心得体会

1. 通过这次实验，我对于线程调度的过程有了更加深入的了解，初步实现了四种调度算法。

2. 通过这次实验，我已经能熟练使用 gdb 进行调试，并且通过调试，我对于整个线程的创建，调度，中断恢复的过程更加熟悉。

3. 在这次实验中，我本来想通过时间中断去实现 短作业优先调度算法，但是在实现的过程中发现，当前中断体系中没有实现时间中断的处理，所以无法实现时间中断，本来想自己实现的，但是能力有限，总是出现bug，所以暂时放弃，以后有时间的话再实现。

六、参考资料

1. https://gitee.com/sysu2024-osta/sysu-2024-spring-operating-system/tree/main/lab5

2. https://blog.csdn.net/DreamIPossible/article/details/108277473