



# 本科生实验报告

实验课程 中山大学 2021 学年春季操作系统课程

实验名称 内核线程

专业名称 计算机科学与技术（超算）

学生姓名 黄玟瑜

学生学号 19335074

任课教师 陈鹏飞

实验地点

实验成绩

二〇二一年四月二十四日

# 目录

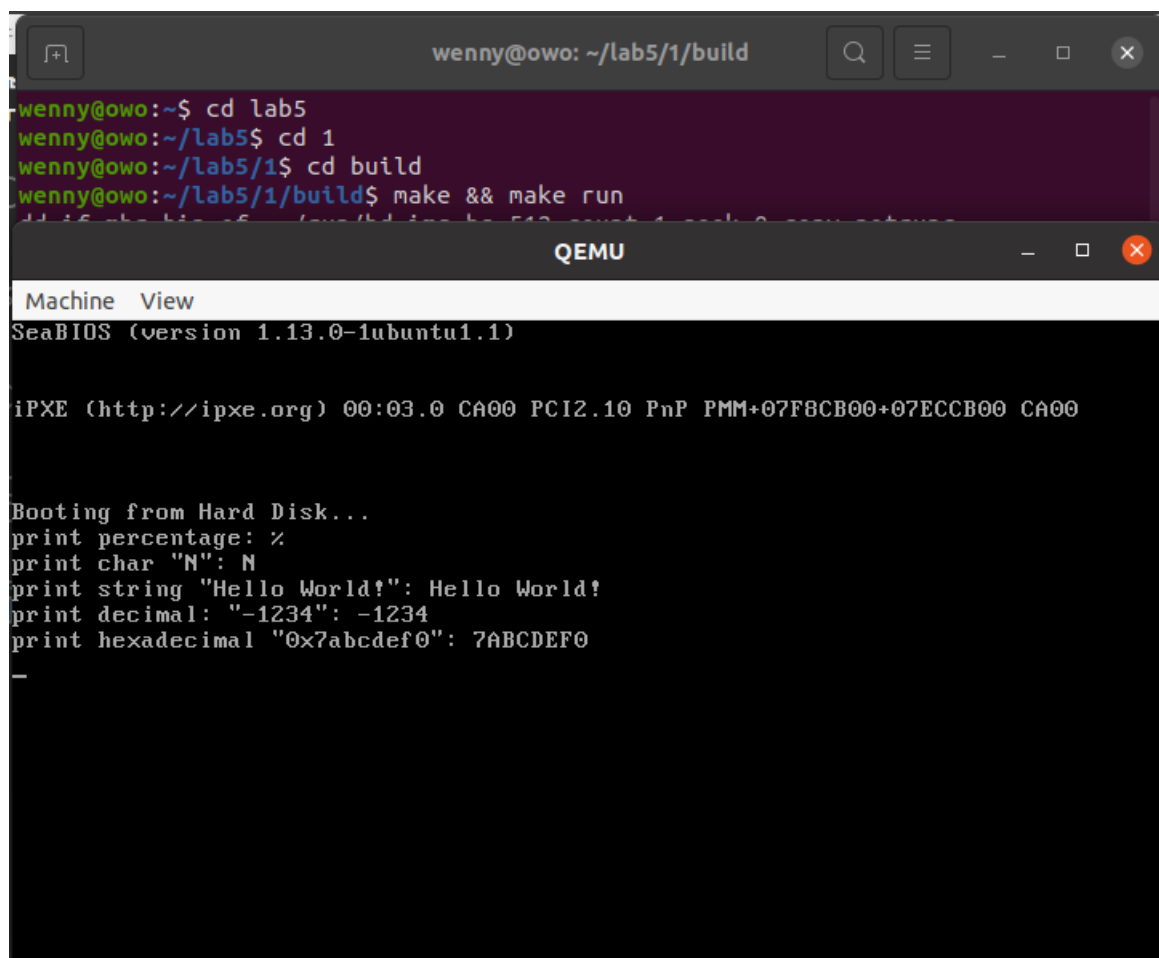
<b>1</b>	<b>Assignment 1: printf 的实现</b>	<b>1</b>
1.1	简单复现 . . . . .	1
1.2	printf 的改进 . . . . .	1
1.2.1	%p 的显示 . . . . .	2
1.2.2	%o 和%u 的显示 . . . . .	2
1.2.3	输出对齐 . . . . .	4
<b>2</b>	<b>Assignment 2: 线程的实现</b>	<b>9</b>
2.1	添加 PCB 的优先级属性 . . . . .	9
2.2	运行结果 . . . . .	10
<b>3</b>	<b>Assignment 3: 线程调度切换的秘密</b>	<b>14</b>
3.1	线程函数 . . . . .	14
3.2	第一个过程的说明 . . . . .	15
3.3	第二个过程的说明 . . . . .	19
<b>4</b>	<b>Assignment 4: 调度算法的实现</b>	<b>24</b>
4.1	最短作业优先 . . . . .	24
4.2	运行结果 . . . . .	26
<b>5</b>	<b>总结</b>	<b>30</b>

## 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。

### 简单复现

在本次实验的目录 lab5 下创建工作目录 1，按照实验指导所述方法，执行 make && make run，结果如下：



```
wenny@owo: ~/lab5/1/build
wenny@owo:~$ cd lab5
wenny@owo:~/lab5$ cd 1
wenny@owo:~/lab5/1$ cd build
wenny@owo:~/lab5/1/build$ make && make run
dd if=/dev/zero of=/dev/hda bs=512 count=1 seek=0 conv=notrunc
QEMU
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...
print percentage: %
print char "N": N
print string "Hello World!": Hello World!
print decimal: "-1234": -1234
print hexadecimal "0x7abcdef0": 7ABCDEF0
-
```

看到屏幕上成功打印了常规字符、实现了换行、% 的显示以及%c、%s、%d、%x 的输出。

### printf 的改进

考虑到本次实验实现换行、% 的显示以及%c、%s、%d、%x 的输出方法是较为高效的，且通过缓冲区避免了字符串超过函数调用占大小的问题，最重要的是本人水平有限，想不出明显优于上述方法的算法，因此在此处想到增加其他 printf 的功能。

下面将在教程的 printf 上增加以下功能。

- 对齐输出，包括左对齐、右对齐
- %p 输出地址符
- %o 无符号以八进制表示的整数
- %u 十进制无符号整数

## %p 的显示

在 32 位的操作系统中，地址的长度是 32 位的，一般情况下以十六进制显示，因此预期的结果应该是输出 “0x\*\*\*\*\*” (\* 为 0~F 的数字，最多不超过 8 个)

```

1  ...
2      case 'p':{
3          itos(number, va_arg(ap, uint32), 16);
4
5          counter += printf_add_to_buffer(buffer, '0', idx, BUF_LEN);
6          counter += printf_add_to_buffer(buffer, 'x', idx, BUF_LEN);
7
8          for (int j = 0; number[j]; ++j)
9          {
10             counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
11          }
12      }
13  ...
14      break;

```

使用 ap 和 va\_arg 来引用可变参数，将传入的地址的值转化为字符串并存储在字符串 numbers 中，先输出十六进制标识符 “0x”，再将十六进制的地址的值输出。

## %o 和 %u 的显示

在实现%d、%x 的显示之后发现%o 和%u 的显示并不难实现，最主要的进制的改变以及将参数视为有符号数还是无符号数的问题，%d 是将数值看作是有符号数，因此需要将小于 0 的数取反，并在输出之前加上 ‘-’，而%o、%u、%x 将数值看作无符号数来处理，只是输出时进制不同，分别为八进制、十进制和十六进制。

实现方法如下：

```

1  uint32 mod = 0;
2  ...
3      case 'o':
4          if(mod == 0) mod = 8;
5      case 'u':
6      case 'd':
7          if(mod == 0) mod = 10;
8      case 'x':
9          if(mod == 0) mod = 16;
10         int temp = va_arg(ap, int);
11
12         if (temp < 0 && fmt[i] == 'd')

```

```

13     {
14         counter += printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
15
16         temp = -temp;
17     }
18
19     itos(number, temp, mod);
20
21     for (int j = 0; number[j]; ++j)
22     {
23         counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
24     }
25     break;
26     ...

```

用 mod 来标识进制，先将变量 mod 初始化为 0，在各个 case 中判断 mod 是否为 0，若 mod 为 0 则将其根据进入该 case 的进制赋值，保证了最终 mod 的值是最先执行的 case 对应的进制，将传入的地址的值以 mod 所表示的进制转化为字符串并存储在字符串 numbers 中，再将字符串输出。

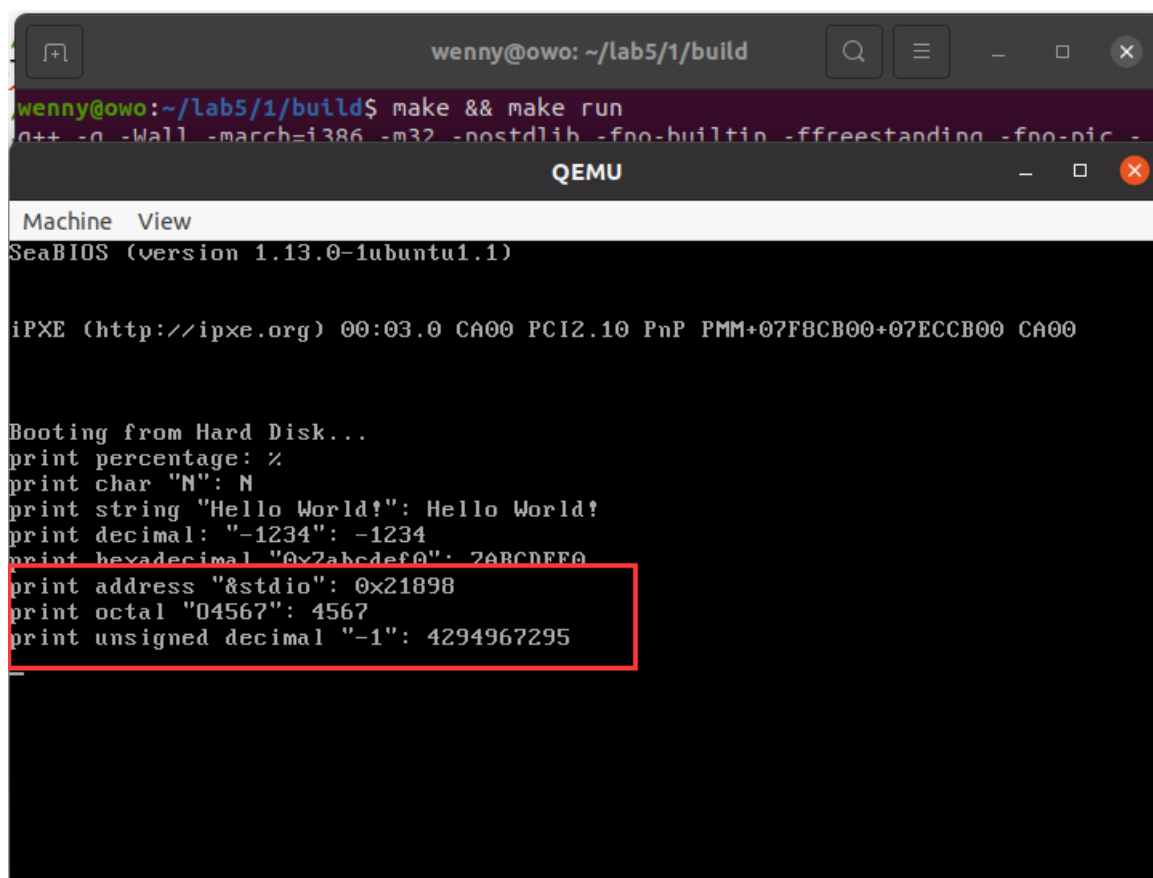
到此实现了%p、%o 和%u 的显示，下面进行验证，补充修改测试语句：

```

1  include "asm_utils.h"
2  #include "interrupt.h"
3  #include "stdio.h"
4
5  // 屏幕IO处理器
6  STDIO stdio;
7
8  // 中断管理器
9  InterruptManager interruptManager;
10
11 extern "C" void setup_kernel()
12 {
13     // 中断处理部件
14     interruptManager.initialize();
15
16     // 屏幕IO处理部件
17     stdio.initialize();
18
19     interruptManager.enableTimeInterrupt();
20     interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);
21     //asm_enable_interrupt();
22
23     printf("print percentage: %%\n"
24           "print char \"N\": %c\n"
25           "print string \"Hello World!\": %s\n"
26           "print decimal: \"-1234\": %d\n"
27           "print hexadecimal \"0x7abcdef0\": %x\n"
28           "print address \"%stdio\": %p\n"
29           "print octal \"04567\": %o\n"
30           "print unsigned decimal \"-1\": %u\n",
31           'N', "Hello World!", -1234, 0x7abcdef0, &stdio, 04567, -1);
32     //uint a = 1 / 0;
33     asm_halt();
34 }

```

输出如下：



```
wenny@owo: ~/lab5/1/build
wenny@owo:~/lab5/1/build$ make && make run
g++ -g -Wall -march=i386 -m32 -nostdlib -fno-builtin -ffreestanding -fno-pic -
QEMU
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...
print percentage: %
print char "N": N
print string "Hello World!": Hello World!
print decimal: "-1234": -1234
print hexadecimal "0x2abcdef0": 7aBCDEff
print address "&stdio": 0x21898
print octal "04567": 4567
print unsigned decimal "-1": 4294967295
```

printf 正确的输出了 32 位十六进制地址值、无符号八进制整数、无符号十进制整数。

## 输出对齐

输出对齐的预期结果是在“%”和字母之间插进数字表示最大场宽。

例如：%3d 表示输出 3 位整型数，不够 3 位右对齐。

可以控制输出左对齐或右对齐，即在“%”和字母之间加入一个“-”号即可说明输出为左对齐，否则为右对齐。

例如：% -7d 表示输出 7 位整数左对齐 %-10s 表示输出 10 个字符左对齐

由于储备知识有限，这里实现的输出对齐比较复杂，实现思想大致如下：先记录格式参数，如出现“-”说明需要左对齐，否则默认为右对齐，考虑使用 bool 类型的变量 right 来存储这个属性，若 right=true 说明需要右对齐，否则为左对齐，再依次读入数字字符，存储在字符串 align\_num 中，通过函数 stoi 将字符串转化为数值存储在 align 中，这样就得到了需要的场宽。

接下来根据字母获取参数，判断出所需要打印的字符的长度，若需要打印的字符长度小于场宽则进行相应的填充，否则按正常的方式输出。

记录格式参数过程如下：

```

1 char align_num[3];
2 ...
3     bool right = true;
4     uint32 length = 0, mod = 0, addr;
5     int align, align_length = 0;
6     int temp;
7
8     if(fmt[i]=='-'){
9         right = false;
10        i++;
11    }
12
13    //functions like strcpy
14    while(isdigit(fmt[i])){
15        align_num[align_length]=fmt[i];
16        align_length++;
17        i++;
18    }
19    align_num[align_length] = '\0';
20
21    stoi(align_num, align, 10);
22 ...

```

right 的值默认为 true，align 设置为 int 类型而不是 uint32，这点后面再做说明，若出现“-”说明需要左对齐，将 right 置为 false，同时读取下一个字符，接着继续读入数字存储在字符串 align\_num 中，isdigit 的实现十分简单，如下所示：

```

1 int isdigit(int c){
2
3     return (c >= '0' && c <= '9');
4
5 }

```

字符串转化为数字的函数 stoi 定义如下：

```

1
2 void stoi(char *numStr, int &num, uint32 mod){
3
4     if(mod != 10)
5         return;
6
7     num = 0;
8     uint32 length = 0;
9     uint32 radix = 1;
10
11    while(numStr[length]){
12        length++;
13    }
14
15    while(length){
16        num += radix * (numStr[--length] - '0');
17        radix *= 10;
18    }
19 }

```

该函数只能将字符串转化为十进制数（目前只需要转化十进制数，在以后的实验中

若需要会继续对该函数进行拓展)，将 num 初始化为 0，基数 radix 初始化为 1，首先获取字符串长度 length，位置 length-1 即字符串的最后一个字符的所在位置，也就是个位数字所对应的字符，从最低位开始向前遍历，每次循环基数增加到原来的十倍。在这里 numstr 最多只有 3 位数，因为屏幕长度只有 80，不需要也不允许过长的对齐。

接下来对字母 fmt[i] 通过 switch 语句进行两次条件判断，第一个 switch 根据 fmt[i] 和参数计算出所需要打印的字符长度 length，第二个 switch 输出所需要打印的字符。在第二个 switch 语句前后根据 align 和 length 打印空字符以进行适当的填充。

代码如下：

```
1      switch (fmt[i])
2      {
3          case '%':
4          case 'c':
5              length = 1;
6              break;
7
8          case 's':
9              s = va_arg(ap, const char *);
10             while(s[length])
11                 length++;
12             break;
13
14          case 'p':
15              addr = va_arg(ap, uint32);
16              number[0] = '0';
17              number[1] = 'x';
18              itos(number + 2, addr, 16);
19              for (int j = 0; number[j]; ++j){
20                  length++;
21              }
22              break;
23
24          case 'o':
25              if(mod == 0) mod = 8;
26          case 'u':
27          case 'd':
28              if(mod == 0) mod = 10;
29          case 'x':
30              if(mod == 0) mod = 16;
31
32              temp = va_arg(ap, int);
33              if (temp < 0 && fmt[i] == 'd')
34              {
35                  temp = -temp;
36                  number[0] = '-';
37                  itos(number + 1, temp, mod);
38              }
39              else
40                  itos(number, temp, mod);
41
42              for (int j = 0; number[j]; ++j){
43                  length++;
44              }
```



```

45     }
46     align -= length;
47
48     if(right)
49         while(align > 0){
50             counter += printf_add_to_buffer(buffer, ' ', idx, BUF_LEN);
51             align--;
52         }
53
54     switch (fmt[i])
55     {
56     case '%':
57         counter += printf_add_to_buffer(buffer, fmt[i], idx, BUF_LEN);
58         break;
59
60     case 'c':
61         counter += printf_add_to_buffer(buffer, va_arg(ap, char), idx, BUF_LEN);
62         break;
63
64     case 's':
65         buffer[idx] = '\0';
66         idx = 0;
67         counter += stdio.print(buffer);
68         counter += stdio.print(s);
69         break;
70
71     case 'p':
72     case 'd':
73     case 'o':
74     case 'u':
75     case 'x':
76
77         for (int j = 0; number[j]; ++j)
78         {
79             counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
80         }
81
82     }
83
84     while(align > 0){
85         counter += printf_add_to_buffer(buffer, ' ', idx, BUF_LEN);
86         align--;
87     }

```

该过程相当于将没有输出对齐的打印过程分为 2 步：统计长度、填充输出。

地址的输出前需打印 “0x”，因此将 number 数组的前两个字符赋值为 “0x”，再将十六进制的地址的字符形式存到 number 数组从第三个字符起的后续位置中。

负数的输出需先打印 “-”，因此将 number 数组的第一个字符赋值为 “-”，再将它的绝对值的字符形式存到 number 数组从第三个字符起的后续位置中。

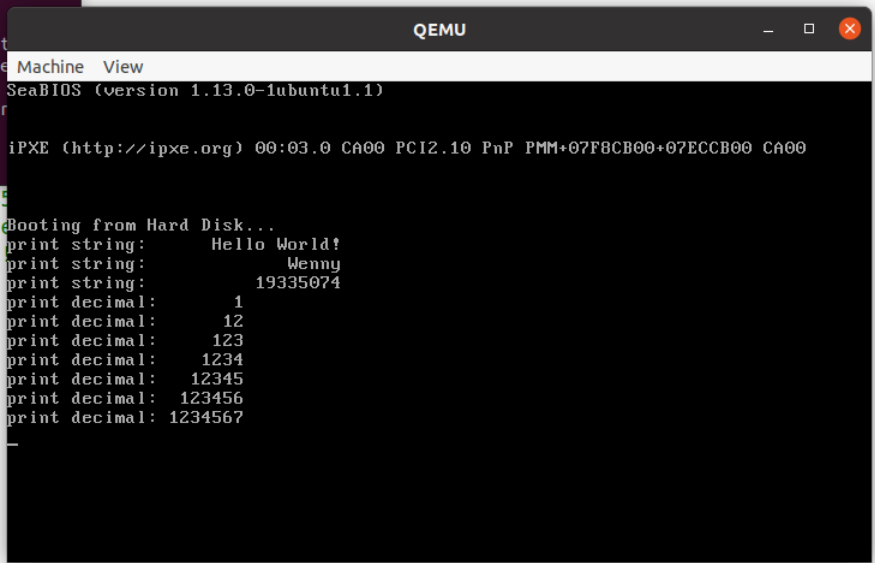
将上述涉及到的新定义的函数声明写道对应的头文件中。

运行结果如下：

字符串、负数的对齐：

```
042 s, 123 MB/s
l null -parallel stdio -no-reboot
r '../run/hd.img' and probing guest
t is dangerous for raw images, write
ly to remove the restrictions.

// "print octal \"042\"
// "print unsigned decimal: 123
// 'N', "Hello World"
printf(
    "print string: %17s\n"
    "print string: %17s\n"
    "print string: %17s\n"
    "print decimal: %7d\n"
    "print decimal: %7d\n"
    "print decimal: %7d\n"
    "print decimal: %7d\n"
    "print decimal: %7d\n"
    "print decimal: %7d\n"
    "print decimal: %7d\n",
    "Hello World!", "Wenny", "19335074", 1, 12, 123, 1234, 12345, 123456, 1234567);
```




QEMU Machine View  
SeaBIOS (version 1.13.0-1ubuntu1.1)  
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00  
Booting from Hard Disk...  
print string: Hello World!  
print string: Wenny  
print string: 19335074  
print decimal: 1  
print decimal: 12  
print decimal: 123  
print decimal: 1234  
print decimal: 12345  
print decimal: 123456  
print decimal: 1234567

字符按照对齐要求对齐输出。

字符串、正数的对齐：

```
interruptManager.setItimeInterrupt((void *)asm_time_interrupt_handler);
//asm_enable_interrupt();
// printf("print percentage
// "print char \"N\"
// "print string \"H
// "print decimal: \
// "print hexadecimal
// "print address \
// "print octal \"04
// "print unsigned d
// 'N', "Hello World"
printf(
    "print string: %17s\n"
    "print string: %17s\n"
    "print string: %17s\n"
    "print decimal: %7d\n"
    "print decimal: %7d\n"
    "print decimal: %7d\n"
    "print decimal: %7d\n"
    "print decimal: %7d\n"
    "print decimal: %7d\n"
    "print decimal: %7d\n",
    "Hello World!", "Wenny", "19335074", -1, -12, -123, -1234, -12345, -123456, -1234567);
```



QEMU Machine View  
SeaBIOS (version 1.13.0-1ubuntu1.1)  
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00  
Booting from Hard Disk...  
print string: Hello World!  
print string: Wenny  
print string: 19335074  
print decimal: -1  
print decimal: -12  
print decimal: -123  
print decimal: -1234  
print decimal: -12345  
print decimal: -123456  
print decimal: -1234567

可以看到未满足场宽的被填充到场宽长度，本身长度已经超过场宽的正常输出（最后一行输出字符数为 8，超过了 7 个字符，因此没有对齐），结果符合预期。

## 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。

### 添加 PCB 的优先级属性

示例中的 PCB 已包含优先级，用成员变量 `priority` 标识，但在线程调度函数 `ProgramManager::schedule` 中切换线程的方式只是根据就绪队列的顺序切换到下一个线程，而就绪队列中的线程顺序是线程创建的先后顺序，或者说，创建该线程的函数 `ProgramManager::executeThread` 的执行时间的先后，并未使用到线程的优先级属性。

下面将在此基础上设计调度算法，使线程的调度依据 PCB 的优先级，优先级高的线程先送到处理器上执行。

修改线程调度函数 `ProgramManager::schedule`，使就绪队列的出队顺序为优先级从高到低的顺序：

```
1 void ProgramManager::schedule()
2 {
3     bool status = interruptManager.getInterruptStatus();    // 获取中断管理器的当前状态
4     interruptManager.disableInterrupt();
5
6     if (readyPrograms.size() == 0)
7     {
8         interruptManager.setInterruptStatus(status);        // 恢复中断管理器的状态
9         return;
10    }
11
12    if (running->status == ProgramStatus::RUNNING)
13    {
14        running->status = ProgramStatus::READY;
15        running->ticks = running->priority * 10;
16        readyPrograms.push_back(&(running->tagInGenerallist));
17    }
18    else if (running->status == ProgramStatus::DEAD)
19    {
20        releasePCB(running);
21    }
22
23    ListItem *item = get_next_program();                    // 主要修改部分：获取下一个将要执行的线程
24    PCB *next = ListItem2PCB(item, tagInGenerallist);
25    PCB *cur = running;
26    next->status = ProgramStatus::RUNNING;
27    running = next;
28    readyPrograms.erase(item);                              // 出队
29
30    asm_switch_thread(cur, next);
31
32    interruptManager.setInterruptStatus(status);
33 }
```

其中函数 `ProgramManager::get_next_program` 从就绪队列 `readyPrograms` 中选择优先级最高的 PCB 并返回其对应的 `Listitem` 的地址，定义如下：

```
1 ListItem * ProgramManager::get_next_program(){
2     ListItem *item = readyPrograms.front();
3     if(readyPrograms.size() == 1)
4         return item;
5
6     PCB *ptr = ListItem2PCB(item, tagInGeneralList);
7     ListItem *target_item = item;
8     int max_priority = ptr->priority;
9
10    item = item->next;
11    while(item){
12        ptr = ListItem2PCB(item, tagInGeneralList);
13        if(ptr->priority > max_priority){
14            max_priority = ptr->priority;
15            target_item = item;
16        }
17        item = item->next;
18    }
19    return target_item;
20 }
```

先获取就绪队列的第一个线程的 `Listitem` 的地址，若就绪队列长度为 1，毫无疑问它就是就绪队列中优先级最高的，调用该函数前已确认就绪队列不为空，因此不必考虑 `readyPrograms.size() = 0` 的情况。

随后用变量 `max_priority` 暂存当前最高优先级，变量 `target_item` 暂存目标的地址值，从链表头开始遍历、比较、存储结果。

## 运行结果

下面在 `setup_kernel` 对上述实现进行测试，测试方法如下：

```
1 #include "asm_utils.h"
2 #include "interrupt.h"
3 #include "stdio.h"
4 #include "program.h"
5 #include "thread.h"
6 #include "stdlib.h"
7
8 // 屏幕IO处理器
9 STDIO stdio;
10 // 中断管理器
11 InterruptManager interruptManager;
12 // 程序管理器
13 ProgramManager programManager;
14 void fourth_thread(void *arg) {
15     printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid,
16           programManager.running->name);
17 }
18 void third_thread(void *arg) {
19     printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid,
20           programManager.running->name);
21 }
```

```

19 }
20 void second_thread(void *arg) {
21     printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid,
22           programManager.running->name);
23 }
24 void first_thread(void *arg)
25 {
26     // 第1个线程不可以返回
27     printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid,
28           programManager.running->name);
29     if (!programManager.running->pid)
30     {
31         // 创建线程分别执行second_thread、third_thread、fourth_thread
32         programManager.executeThread(second_thread, nullptr, "second thread", 2);
33
34         programManager.executeThread(third_thread, nullptr, "third thread", 3);
35
36         programManager.executeThread(fourth_thread, nullptr, "fourth thread", 4);
37     }
38     asm_halt();
39 }
40
41 extern "C" void setup_kernel()
42 {
43     // 中断管理器
44     interruptManager.initialize();
45     interruptManager.enableTimeInterrupt();
46     interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);
47
48     // 输出管理器
49     stdio.initialize();
50
51     // 进程/线程管理器
52     programManager.initialize();
53
54     // 创建第一个线程（并未执行）
55     int pid = programManager.executeThread(first_thread, nullptr, "first thread", 1);
56     if (pid == -1)
57     {
58         printf("can not execute thread\n");
59         asm_halt();
60     }
61
62     // 手动执行类似schedule
63     ListItem *item = programManager.readyPrograms.front();
64     PCB *firstThread = ListItem2PCB(item, tagInGeneralList); // 装载了函数first_thread的
65                                                                PCB
66     firstThread->status = RUNNING;
67     programManager.readyPrograms.pop_front();
68     programManager.running = firstThread; // 当前进程在执行
69     asm_switch_thread(0, firstThread); // 转到第一个线程
70
71     asm_halt();
72 }

```

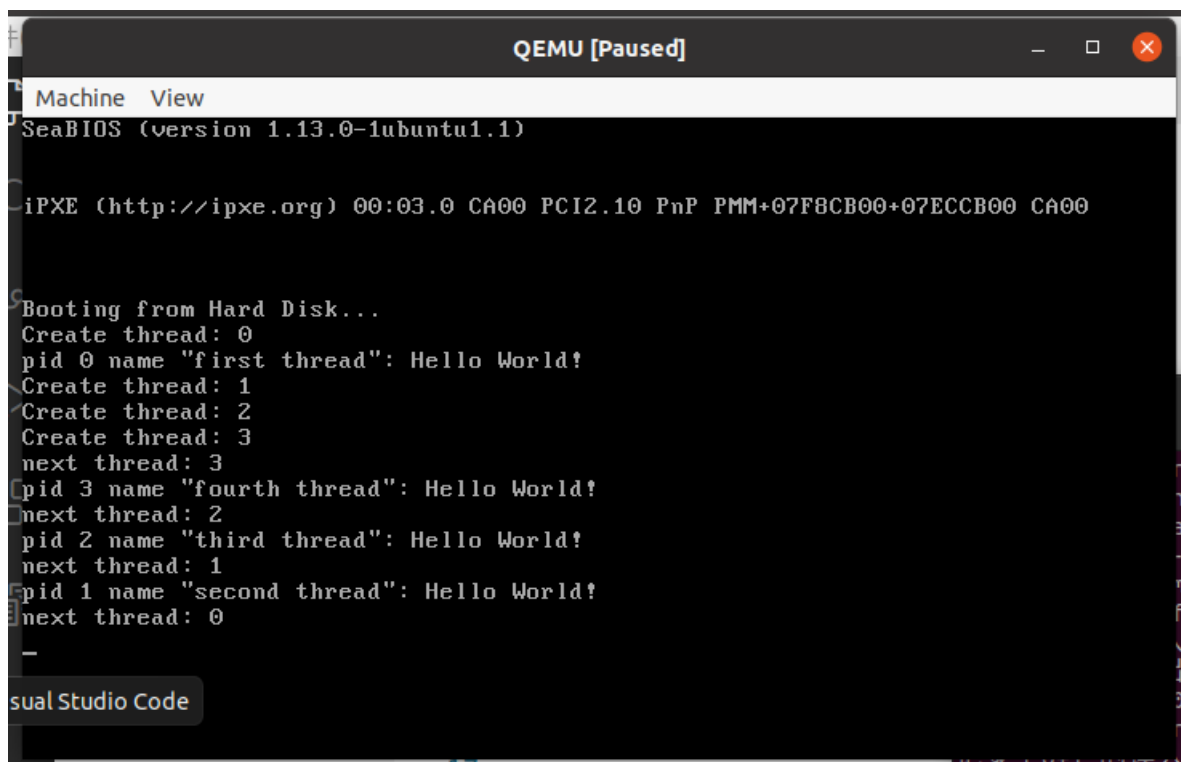
在 setup\_kernel 中，先创建第一个线程执行 first\_thread，在 first\_thread 再依次

创建线程执行 `second_thread`、`third_thread`、`fourth_thread`，线程的创建并不会立即执行，`ProgramManager::executeThread` 创建线程的过程只是将 PCB 的进行初始化再放入就绪队列中，需要通过 `ProgramManager::schedule` 调度执行，真正意义上的控制线程的执行函数为 `asm_switch_thread`。

在本程序中，第一个线程创建后先执行输出，随后连续创建第二、三、四个线程，它们的执行需要时间，且执行时间一定程度的大于线程入队所花费的时间，因此会出现第二、三、四个线程都在就绪队列中的时刻，这时候观察输出的顺序可知哪个线程先被调度执行了。（若线程的执行时间小于入队时间，说明线程几乎在创建时就已执行完毕，那么将无法观察调度方法所起的作用）。

以上程序第二、三、四个线程的优先级分别为 2、3、4，即第四个线程的优先级 > 第三个线程的优先级 > 第二个线程的优先级，输出结果如下：

（输出 “Create thread:” 的语句放置在线程创建函数 `ProgramManager::executeThread` 的返回语句之前，输出为创建的线程的 pid；输出 “next thread:” 的语句放置在调度函数 `ProgramManager::schedule` 中的 `asm_switch_thread` 前，输出为下一个线程 next 的 pid）



看到第 4 个线程（pid=3）先输出了 “Hello World! ”，其次是第 3 个线程（pid=2）第 2 个线程（pid=1），最后又回到第 1 个线程（pid=0），在 `halt` 处停止。

输出结果符合预期。

```
1 void first_thread(void *arg)
2 {
3     // 第1个线程不可以返回
```

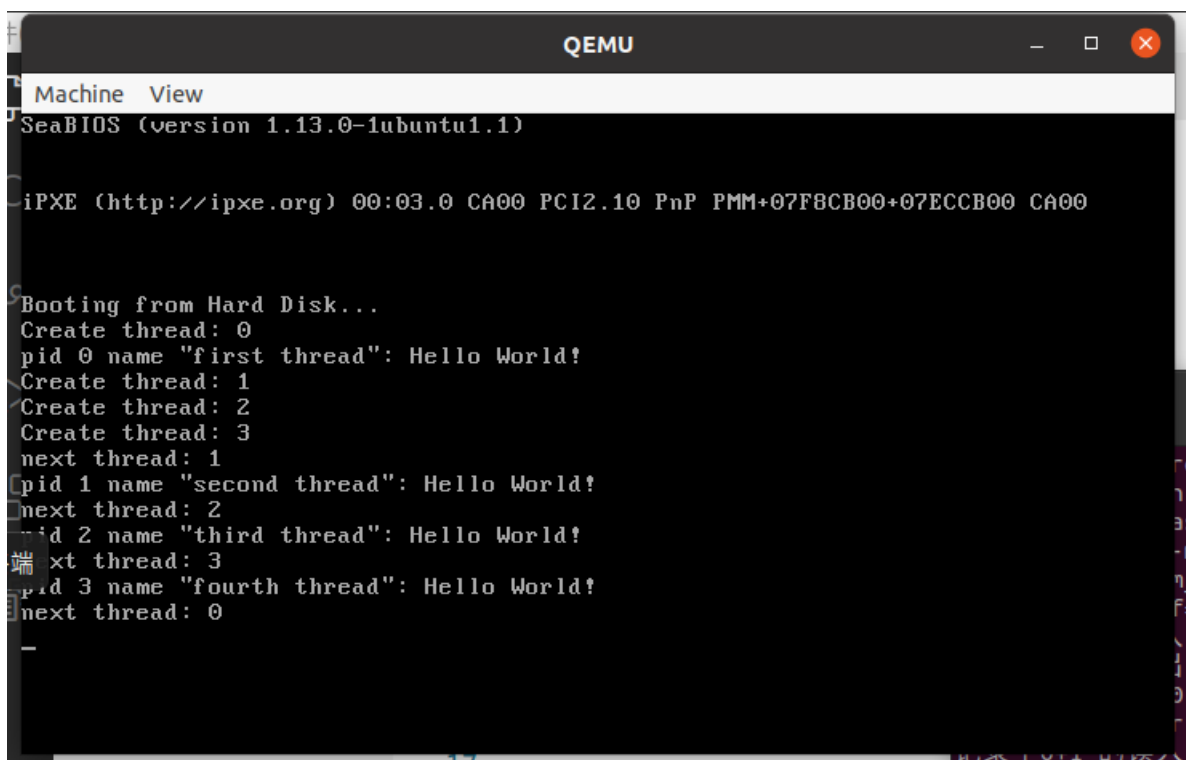
```

4     printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid,
           programManager.running->name);
5     if (!programManager.running->pid)
6     {
7         // 创建线程分别执行second_thread、third_thread、fourth_thread
8         programManager.executeThread(second_thread, nullptr, "second thread", 4);
9
10        programManager.executeThread(third_thread, nullptr, "third thread", 3);
11
12        programManager.executeThread(fourth_thread, nullptr, "fourth thread", 2);
13    }
14    asm_halt();
15 }

```

以上程序第二、三、四个线程的优先级分别为 4、3、2，即第二个线程的优先级 > 第三个线程的优先级 > 第四个线程的优先级，输出结果如下：

(输出 “Create thread:” 的语句放置在线程创建函数 ProgramManager::executeThread 的返回语句之前，输出为创建的线程的 pid；输出 “next thread:” 的语句放置在调度函数 ProgramManager::schedule 中的 asm\_switch\_thread 前，输出为下一个线程 next 的 pid)



```

QEMU
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...
Create thread: 0
pid 0 name "first thread": Hello World!
Create thread: 1
Create thread: 2
Create thread: 3
next thread: 1
pid 1 name "second thread": Hello World!
next thread: 2
pid 2 name "third thread": Hello World!
next thread: 3
pid 3 name "fourth thread": Hello World!
next thread: 0
asm_halt()

```

看到第 2 个线程 (pid=1) 先输出了 “Hello World! ”，其次是第 3 个线程 (pid=2) 第 4 个线程 (pid=3)，最后又回到第 1 个线程 (pid=0)，在 halt 处停止。输出结果符合预期。

## 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。

### 线程函数

涉及到 4 个线程函数,分别为 first\_thread、second\_thread、third\_thread、fourth\_thread，在 setup\_kernel 中，先创建第一个线程执行 first\_thread，在 first\_thread 再依次创建线程执行 second\_thread、third\_thread、fourth\_thread，创建好的线程被编入就绪队列中，等待执行。

```
1 void fourth_thread(void *arg) {
2     printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid,
3         programManager.running->name);
4 }
5 void third_thread(void *arg) {
6     printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid,
7         programManager.running->name);
8 }
9 void second_thread(void *arg) {
10    printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid,
11        programManager.running->name);
12 }
13 void first_thread(void *arg)
14 {
15     // 第1个线程不可以返回
16     printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid,
17         programManager.running->name);
18     if (!programManager.running->pid)
19     {
20         // 创建线程分别执行 second_thread、third_thread、fourth_thread
21         programManager.executeThread(second_thread, nullptr, "second thread", 1);
22
23         programManager.executeThread(third_thread, nullptr, "third thread", 1);
24     }
25 }
```



```

21     programManager.executeThread(fourth_thread, nullptr, "fourth thread", 1);
22 }
23 asm_halt();
24 }

```

实现过程如下：

创建线程1，转到 first\_thread

创建线程2、3、4 halt 产生时钟中断

10个时钟中断 schedule 转线程4

线程4执行完毕，program\_exit pval > 0 → schedule 转3

线程3执行完毕，program\_exit pval > 0 → schedule 转2

线程2执行完毕，program\_exit pval > 0 → schedule 转1

转至线程1 schedule 至线程4时的位置 (asm\_switch\_thread)

恢复中断，回到时钟中断处理函数 schedule 处，所回到 first\_thread

## 第一个过程的说明

每个线程都有一个时间片的数值，它代表了线程可以运行多长时间，直到时间片届满。这个数值不是一个时间的计时长度，而是一个整数，称之为“时间片单位 (quantum units)”，在本次实验中，一个时间片单位即一次 8259A 发出的时间中断的经过时间。

在本次实验中所使用的线程调度算法是最简单的时间片轮转算法。时间片轮转算法在每个时钟中断到来时将 PCB 的经过时间片加 1，剩余时间片减 1，当剩余时间片为 0 时说明该线程已经到达了它生命周期的尽头，将它从处理器上移出去，同时调度其他的线程进来，放到处理器上去执行。

在 c\_time\_interrupt\_handler 处设置断点，查看每次时钟中断发生后当前运行的线程的经过时间片和剩余时间片的变化：

```

Breakpoint 2, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:89
(gdb) p cur->pid
$6 = 0
(gdb) p cur->ticks
$7 = 8
(gdb) p cur->ticksPassedBy
$8 = 2
(gdb) c
Continuing.

Breakpoint 2, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:89
(gdb) p cur->ticks
$9 = 7
(gdb) p cur->ticksPaasedBy
There is no member named ticksPaasedBy.
(gdb) p cur->ticksPassedBy
$10 = 3
(gdb) █

```

可以看到，每次时钟中断发生后，当前运行的线程（first\_thread）经过时间片加 1，剩余时间片减 1。

```

Continuing.

Breakpoint 2, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:89
(gdb) c
Continuing.

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

Breakpoint 1, asm_switch_thread () at ../src/utils/asm_utils.asm:24
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) x/7aw 0x22f28
0x22f28 <PCB_SET+4040>: 0x0 0x0 0x0 0x22f64 <PCB_SET+4100>
0x22f38 <PCB_SET+4056>: 0x203a7 <ProgramManager::schedule()+329> 0x21fe0 <PCB_SET+128> 0x24fe0 <PCB_SET+12416>
(gdb) █

```

当剩余时间片为 0 时，分配给该线程的时间已耗尽，下面进行调度。本次实验中线程调度到下一个进程时，选择下一个线程的方式为先来先服务，即根据就绪队列的顺序切换到下一个线程，而就绪队列中的线程顺序是线程创建的先后顺序，或者说，创建该线程的函数 ProgramManager::executeThread 的执行时间的先后。

选择下一个线程，它的地址指针存储在 next 中，asm\_switch\_thread 实现了线程栈的切换。线程的所有信息都在线程栈中，只要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 asm\_switch\_thread 处设置断点，观察完成栈指针切换后各个寄存器和栈的变化：



```

2   thread->stack = (int *)((int)thread + PCB_SIZE); // 栈底
3   thread->stack -= 7;                               // 7个32位的位置
4   thread->stack[0] = 0;
5   thread->stack[1] = 0;
6   thread->stack[2] = 0;
7   thread->stack[3] = 0;
8   thread->stack[4] = (int)function;                 // 函数地址
9   thread->stack[5] = (int)program_exit;             // 出口函数
10  thread->stack[6] = (int)parameter;                // 最底部放参数的起始地址parameter: 指向函
      数的参数的指针

```

4 个 pop 后 esp 指向 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，且此时 esp 指向了返回地址 program\_exit。进入函数后，函数的栈顶是函数的返回地址 program\_exit，返回地址之后是指向函数的指针 parameter，符合函数的调用规则，从而线程函数得以执行。

下面是 ret 指令说明：

ret 是子程序的返回指令。

执行该指令时：esp 增加一个内存单元，栈顶数据出栈赋值给 eip 寄存器。

The screenshot shows a GDB terminal window with the following content:

**Register group: general**

eax	0x24fe0	151520
ecx	0x22ed2	143058
edx	0xf	15
ebx	0x0	0
esp	0x25fd8	0x25fd8 <PCB_SET+16504>
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0
eip	0x2053e	0x2053e <fourth_thread(void*)>
eflags	0x206	[ IOPL=0 IF PF ]
cs	0x20	32
ss	0x10	16
ds	0x8	8

**Source Code (./src/kernel/setup.cpp):**

```

9   STDIO stdio;
10  // ^$^ &^ '^ '^ %^
11  InterruptManager interruptManager;
12  // ^^ %^ '^ '^ %^
13  ProgramManager programManager;
14  void fourth_thread(void *arg) {
15      printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid, programManager.running->name);
16  }
17
18  void third_thread(void *arg) {
19      printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid, programManager.running->name);
20  }
21  void second_thread(void *arg) {

```

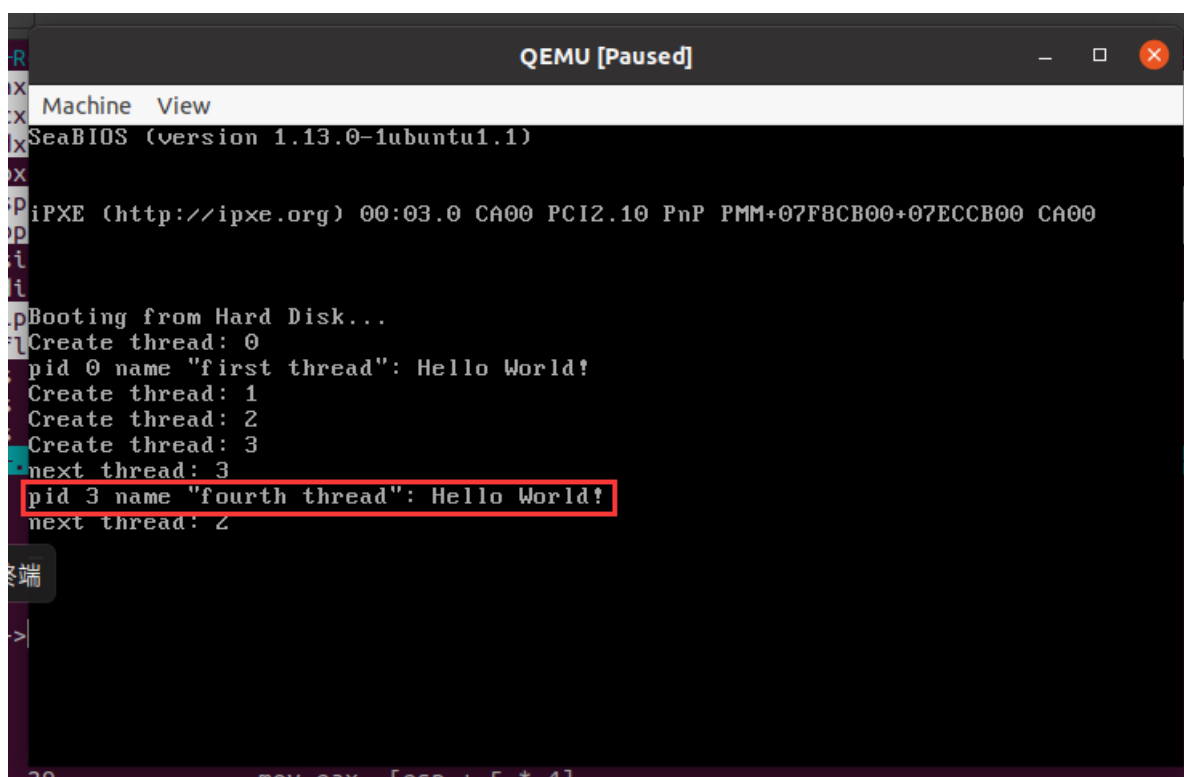
**GDB Session:**

```

remote Thread 1.1 In: fourth_thread
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
asm_switch_thread () at ./src/utls/asm_utils.asm:35
(gdb) s
asm_switch_thread () at ./src/utls/asm_utils.asm:36
(gdb) s
asm_switch_thread () at ./src/utls/asm_utils.asm:37
(gdb) s
asm_switch_thread () at ./src/utls/asm_utils.asm:38
(gdb) s
asm_switch_thread () at ./src/utls/asm_utils.asm:40
(gdb) x/3aw 0x25fd4
0x25fd4 <PCB_SET+16500>: 0x2053e <fourth_thread(void*)> 0x203c1 <program_exit(>) 0x0
(gdb) s
asm_switch_thread () at ./src/utls/asm_utils.asm:41
(gdb) s
fourth_thread (arg=0x0) at ./src/kernel/setup.cpp:14
(gdb)

```

执行 36 至 38 行后,esi,edi,ebx,ebp 均被 pop 为 0,栈顶指向线程函数 fourth\_thread 的地址 (即此时 esp 所保存的地址就是线程 next 的线程函数的地址),ret 后跳到函数 fourth\_thread 处执行。



```
QEMU [Paused]
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...
Create thread: 0
pid 0 name "first thread": Hello World!
Create thread: 1
Create thread: 2
Create thread: 3
next thread: 3
pid 3 name "fourth thread": Hello World!
next thread: 2
```

屏幕上的输出说明了函数 fourth\_thread 被成功执行,线程调度执行成功。

## 第二个过程的说明

正如前面所说,一个线程在它被分配得的时间片使用完毕后被换下处理器,时间片的更新和调用调度函数都是通过时钟中断处理函数来完成,时钟中断处理函数检测到当前运行的线程的时间片耗尽后调用线程调度函数 schedule, schedule 将保存当前线程的上下文,再换上另一个线程上去执行,对于被换上的线程,若它是新线程则从头开始执行 (线程函数的开头),若它是已被执行过的线程,则从它被中断换出的地方继续往下执行。执行完毕后的线程返回到 program\_exit,再调度另一个线程到处理器上执行 (若仅剩第一个线程则 halt)。

first\_thread 被换下的过程如前面所述,时钟中断处理函数检测到 first\_thread 的时间片耗尽后调用线程调度函数 schedule, schedule 保存 first\_thread 的栈指针、esi、edi、ebx、ebp 的值,通过 asm\_switch\_thread 实现线程的切换。

下面追踪 first\_thread 被换回的过程。

当第四、三、二个线程执行完毕后,将要从第二个线程换到第一个线程,在 asm\_switch\_thread 处设置断点:

```

Register group: general
eax      0xf      15
ecx      0x23f52   147282
edx      0xf      15
ebx      0x0       0
esp      0x23f6c   0x23f6c <PCB_SET+8204>
ebp      0x23fa8   0x23fa8 <PCB_SET+8264>
esi      0x0       0
edi      0x0       0
eip      0x21680   0x21680 <asm_switch_thread+4>
eflags   0x16     [ IOPL=0 AF PF ]
cs       0x20     32
ss       0x10     16
ds       0x8      8

../src/utls/asm_utils.asm
19      ASM_IDTR dw 0
20      dd 0
21
22      ; void asm_switch_thread(PCB *cur, PCB *next);
23      asm_switch_thread:
8+ 24      push ebp
25      push ebx
      push edi
      push esi
      push esi
28
>29      mov eax, [esp + 5 * 4]
30      mov [eax], esp ; ^$^ ^ % ^ % ^ & ^ & ^ ) ^ % ^ CB^$^ / ^ $^ $^ ^ ^&^ % ^ & ^ ^ % ^
31

remote Thread 1.1 In: asm_switch_thread
asm_interrupt_status () at ../src/utls/asm_utils.asm:47
(gdb) s
(gdb) s
InterruptManager::getInterruptStatus (this=0x31ff4) at ../src/kernel/interrupt.cpp:115
(gdb) s
(gdb) c
Continuing.

Breakpoint 2, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:89
(gdb) c
Continuing.

Breakpoint 1, asm_switch_thread () at ../src/utls/asm_utils.asm:24
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) x/7aw 0x23f6c
0x23f6c <PCB_SET+8204>: 0x0 0x0 0x0 0x23fa8 <PCB_SET+8264>
0x23f7c <PCB_SET+8220>: 0x203a7 <ProgramManager::schedule()+329>
(gdb)

第2个线程的地址 第1个线程的地址
0x22fe0 <PCB_SET+4224> 0x21fe0 <PCB_SET+128>

```

asm\_switch\_thread 的 4 个 pop 语句完成对 esi、edi、ebx、ebp 的恢复（结合本章“第一个过程的说明”，这 4 个寄存器的值和 first\_thread 被换下时这 4 个寄存器的值相同），并且此时栈顶为 first\_thread 被换下的地址（调用 asm\_switch\_thread 的函数，即 ProgramManager::schedule）：



终端

```

Register group: general
eax      0x21fe0      139232
ecx      0x23f52      147282
edx      0xf          15
ebx      0x0          0
esp      0x22f38      0x22f38 <PCB_SET+4056>
ebp      0x22f64      0x22f64 <PCB_SET+4100>
esi      0x0          0
edi      0x0          0
eip      0x21690      0x21690 <asm_switch_thread+20>
eflags   0x16        [ IOPL=0 AF PF ]
cs       0x20        32
ss       0x10        16
ds       0x8         8

```

esi、edi、ebx、ebp依次被恢复

```

../src/utls/asm_utils.asm
31
32     mov eax, [esp + 6 * 4]
33     mov esp, [eax]; ^&^ &^ &^ %^ ' ^ $^ ur^&^ %^ &^ %^ ext^&^
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

```

(gdb) s
(gdb) x/7aw 0x23f6c
0x23f6c <PCB_SET+8204>: 0x0 0x0 0x0 0x23fa8 <PCB_SET+8264>
0x23f7c <PCB_SET+8220>: 0x203a7 <ProgramManager::schedule()+329> 0x22fe0 <PCB_SET+4224> 0x21fe0 <PCB_SET+128>
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
asm_switch_thread () at ../src/utls/asm_utils.asm:35
(gdb) s
asm_switch_thread () at ../src/utls/asm_utils.asm:36
(gdb) s
asm_switch_thread () at ../src/utls/asm_utils.asm:37
(gdb) s
asm_switch_thread () at ../src/utls/asm_utils.asm:38
(gdb) s
asm_switch_thread () at ../src/utls/asm_utils.asm:40
(gdb) x/7aw 0x22f38
0x22f38 <PCB_SET+4056>: 0x203a7 <ProgramManager::schedule()+329> 0x21fe0 <PCB_SET+128> 0x24fe0 <PCB_SET+12416> 0x22f00 <PCB_SET+4000>
0x22f48 <PCB_SET+4072>: 0x20ac5 <InterruptManager::setInterruptStatus(bool)+33> 0x21fe0 <PCB_SET+128> 0x24fe0 <PCB_SET+12416>
(gdb)

```

栈顶是schedule中asm\_switch\_thread的位置

执行 ret 后，回到 ProgramManager::schedule，恢复中断状态：

```

Register group: general
eax      0x21fe0      139232
ecx      0x23f52      147282
edx      0xf          15
ebx      0x0          0
esp      0x22f4c      0x22f4c <PCB_SET+4076>
ebp      0x22f64      0x22f64 <PCB_SET+4100>
esi      0x0          0
edi      0x0          0
eip      0x203aa      0x203aa <ProgramManager::schedule()+332>
eflags   0x202        [ IOPL=0 IF ]
cs       0x20        32
ss       0x10        16
ds       0x8         8

```

```

../src/kernel/program.cpp
104     readyPrograms.erase(lten);
105
106     printf("next thread: %d\n", next->pid);
107     asm_switch_thread(cur, next);
108
>109     InterruptManager.setInterruptStatus(status);
110 }
111
112 void program_exit()
113 {
114     PCB *thread = programManager.running;
115     thread->status = ProgramStatus::DEAD;

```

remote Thread 1.1 In: ProgramManager::schedule

```

(gdb) s
(gdb) s
(gdb) s
(gdb) s
asm_switch_thread () at ../src/utls/asm_utils.asm:35
(gdb) s
asm_switch_thread () at ../src/utls/asm_utils.asm:36
(gdb) s
asm_switch_thread () at ../src/utls/asm_utils.asm:37
(gdb) s
asm_switch_thread () at ../src/utls/asm_utils.asm:38
(gdb) s
asm_switch_thread () at ../src/utls/asm_utils.asm:40
(gdb) x/7aw 0x22f38
0x22f38 <PCB_SET+4056>: 0x203a7 <ProgramManager::schedule()+329> 0x21fe0 <PCB_SET+128> 0x24fe0 <PCB_SET+12416> 0x22f00 <PCB_SET+4000>
0x22f48 <PCB_SET+4072>: 0x20ac5 <InterruptManager::setInterruptStatus(bool)+33> 0x21fe0 <PCB_SET+128> 0x24fe0 <PCB_SET+12416>
(gdb) s
asm_switch_thread () at ../src/utls/asm_utils.asm:41
(gdb) s
ProgramManager::schedule (this=0x32000) at ../src/kernel/program.cpp:109
(gdb)

```

从时钟中断中返回：

```
Register group: general
eax      0x0      0
ecx      0x23f52  147282
edx      0xf      15
ebx      0x0      0
esp      0x22f7c  0x22f7c <PCB_SET+4124>
ebp      0x22f94  0x22f94 <PCB_SET+4148>
esi      0x0      0
edi      0x0      0
eip      0x20a66  0x20a66 <c_time_interrupt_handler()+76>
eflags   0x2      [ IOPL=0 ]
cs       0x20     32
ss       0x10     16
ds       0x8      8

../src/kernel/interrupt.cpp
90      PCB *cur = programManager.running;
91
92      if (cur->ticks)
93      {
94          --cur->ticks;
95          ++cur->ticksPassedBy;
96      }
97      else
98      {
99          programManager.schedule();
100     }
101     }
102

终端

remote Thread 1.1 In: c_time_interrupt_handler
(gdb) s
ProgramManager::schedule (this=0x32000) at ../src/kernel/program.cpp:109
(gdb) s
InterruptManager::setInterruptStatus (this=0x31ff4, status=false) at ../src/kernel/interrupt.cpp:120
(gdb) s
(gdb) s
(gdb) s
InterruptManager::disableInterrupt (this=0x31ff4) at ../src/kernel/interrupt.cpp:109
(gdb) s
(gdb) s
asm_disable_interrupt () at ../src/utils/asm_utils.asm:52
(gdb) s
(gdb) s
InterruptManager::disableInterrupt (this=0x31ff4) at ../src/kernel/interrupt.cpp:111
(gdb) s
InterruptManager::setInterruptStatus (this=0x31ff4, status=false) at ../src/kernel/interrupt.cpp:129
(gdb) s
ProgramManager::schedule (this=0x32000) at ../src/kernel/program.cpp:110
(gdb) s
c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:101
(gdb) 
```

恢复到线程被中断的地方继续执行：



```
Register group: general
eax      0x3      3
ecx      0x22f12  143122
edx      0x11     17
ebx      0x0      0
esp      0x22fc8  0x22fc8 <PCB_SET+4200>
ebp      0x22fd4  0x22fd4 <PCB_SET+4212>
esi      0x0      0
edi      0x0      0
eip      0x2177b  0x2177b <asm_halt>
eflags   0x206    [ IOPL=0 IF PF ]
cs       0x20     32
ss       0x10     16
帮助     0x8       8

src/utils/asm_utils.asm
181
182     pop eax
183     ret
184
185     asm halt:
>186     jmp $

remote Thread 1.1 In: asm halt
(gdb) s
InterruptManager::disableInterrupt (this=0x31ff4) at ../src/kernel/interrupt.cpp:109
(gdb) s
(gdb) s
asm_disable_interrupt () at ../src/utils/asm_utils.asm:52
(gdb) s
(gdb) s
InterruptManager::disableInterrupt (this=0x31ff4) at ../src/kernel/interrupt.cpp:111
(gdb) s
InterruptManager::setInterruptStatus (this=0x31ff4, status=false) at ../src/kernel/interrupt.cpp:129
(gdb) s
ProgramManager::schedule (this=0x32000) at ../src/kernel/program.cpp:110
(gdb) s
c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:101
(gdb) s
asm_time_interrupt_handler () at ../src/utils/asm_utils.asm:69
(gdb) s
asm_time_interrupt_handler () at ../src/utils/asm_utils.asm:70
(gdb) s
asm_halt () at ../src/utils/asm_utils.asm:186
(gdb) |
```

first\_thread 被中断时运行到的位置为 halt，被换回时继续从 halt 执行。

## 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。

### 最短作业优先

本次实验的调度算法为先来先服务，根据就绪队列的顺序切换到下一个线程，而就绪队列中的线程顺序是线程创建的先后顺序。在 Assignment 2 中实现了优先级调度算法，即从就绪队列中选择优先级高的先执行。

下面将实现最短作业（进程）优先的调度算法。

先在 PCB 中添加运行周期属性 duration，如下所示：

```
1 struct PCB
2 {
3     int *stack;                // 栈指针，用于调度时保存 esp
4     char name[MAX_PROGRAM_NAME + 1]; // 线程名
5     enum ProgramStatus status;    // 线程的状态
6     int priority;                // 线程优先级
7     unsigned int duration;        // 线程请求的执行时间*
8     int pid;                    // 线程 pid
9     int ticks;                  // 线程时间片总时间
10    int ticksPassedBy;           // 线程已执行时间
11    ListItem tagInGeneralList;    // 线程队列标识
12    ListItem tagInAllList;        // 线程队列标识
13 };
```

接着在 ProgramManager::executeThread 的参数中添加线程运行周期，并对 PCB 的初始化部分进行修改：

```
1 int ProgramManager::executeThread(ThreadFunction function, void *parameter, const char *
    name, int priority, unsigned int duration);
```

```

1  thread->priority = priority;    //设置优先级
2  thread->duration = duration;    //设置运行周期
3  thread->ticks = duration;       //设置线程总时间片
4  thread->ticksPassedBy = 0;      //设置线程已执行时间

```

涉及获取运行周期最短的线程的算法 `get_next_program`:

```

1  ListItem * ProgramManager::get_next_program(){
2      ListItem *item = readyPrograms.front();
3      if(readyPrograms.size() == 1)
4          return item;
5
6      PCB *ptr = ListItem2PCB(item, tagInGeneralList);
7      ListItem *target_item = item;
8      unsigned int min_duration = ptr->duration;
9
10     item = item->next;
11     while(item){
12         ptr = ListItem2PCB(item, tagInGeneralList);
13         if(ptr->duration < min_duration){
14             min_duration = ptr->duration;
15             target_item = item;
16         }
17         item = item->next;
18     }
19
20     return target_item;
21 }

```

在线程调度函数 `ProgramManager::schedule` 中选取下一个线程的部分进行修改:

```

1  void ProgramManager::schedule()
2  {
3      bool status = interruptManager.getInterruptStatus();    //
4      interruptManager.disableInterrupt();
5
6
7      if (readyPrograms.size() == 0)
8      {
9          interruptManager.setInterruptStatus(status);        //恢复中断管理器的状态
10         return;
11     }
12
13     if (running->status == ProgramStatus::RUNNING)
14     {
15         running->status = ProgramStatus::READY;
16         running->ticks = running->priority * 10;
17         readyPrograms.push_back(&(running->tagInGeneralList));
18     }
19     else if (running->status == ProgramStatus::DEAD)
20     {
21         releasePCB(running);
22     }
23
24     ListItem *item = get_next_program();    // 获取就绪队列中运行时间最短的线程
25     PCB *next = ListItem2PCB(item, tagInGeneralList);
26     PCB *cur = running;
27     next->status = ProgramStatus::RUNNING;

```

```

28     running = next;
29     readyPrograms.erase(item);           //从就绪队列中删除该线程
30
31     printf("next thread: %d\n", next->pid);
32     asm_switch_thread(cur, next);
33
34     interruptManager.setInterruptStatus(status);
35
36 }

```

## 运行结果

在 setup\_kernel 中对函数进行测试, 修改 setup.cpp 如下:

```

1  #include "asm_utils.h"
2  #include "interrupt.h"
3  #include "stdio.h"
4  #include "program.h"
5  #include "thread.h"
6  #include "stdlib.h"
7
8  // 屏幕IO处理器
9  STDIO stdio;
10
11 // 中断管理器
12 InterruptManager interruptManager;
13
14 // 程序管理器
15 ProgramManager programManager;
16
17 void fourth_thread(void *arg) {
18     printf("pid %d name \"%s\" duration %d: Hello World!\n", programManager.running->pid,
19           programManager.running->name, programManager.running->duration);
20 }
21
22 void third_thread(void *arg) {
23     printf("pid %d name \"%s\" duration %d: Hello World!\n", programManager.running->pid,
24           programManager.running->name, programManager.running->duration);
25 }
26
27 void second_thread(void *arg) {
28     printf("pid %d name \"%s\" duration %d: Hello World!\n", programManager.running->pid,
29           programManager.running->name, programManager.running->duration);
30 }
31
32 void first_thread(void *arg)
33 {
34     // 第1个线程不可以返回
35     printf("pid %d name \"%s\" duration %d: Hello World!\n", programManager.running->pid,
36           programManager.running->name, programManager.running->duration);
37     if (!programManager.running->pid)
38     {
39     }
40     asm_halt();
41 }

```

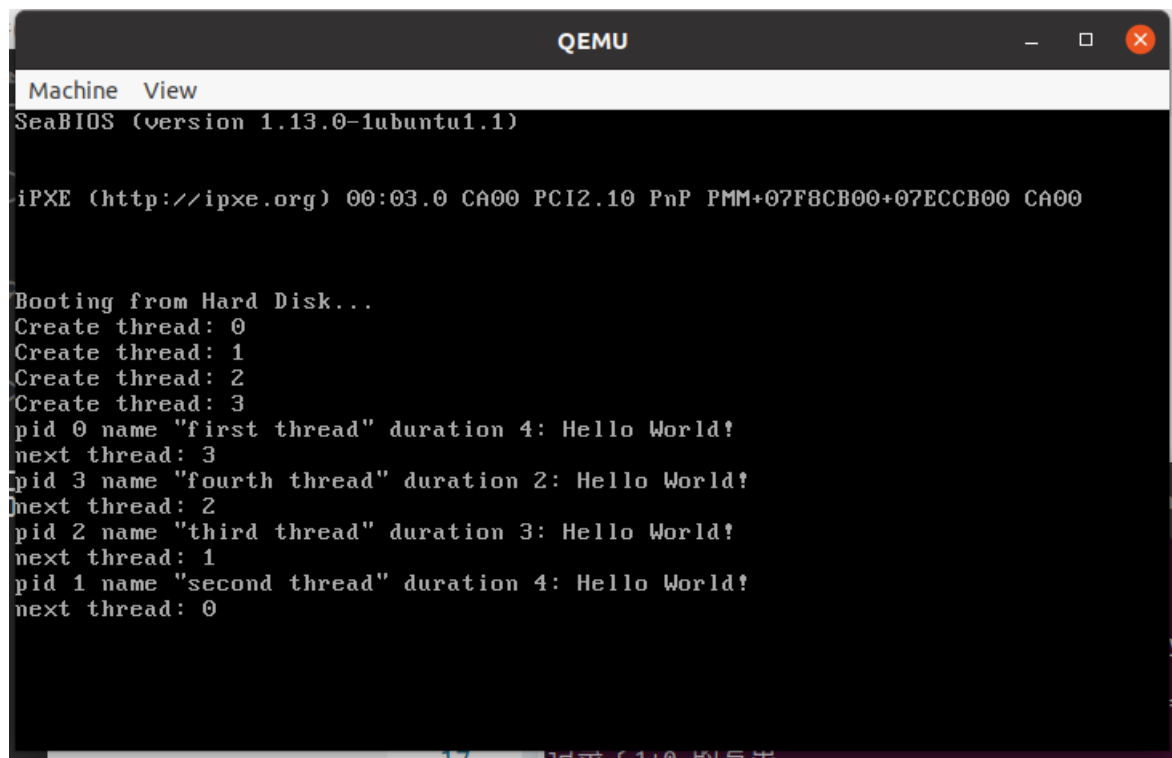
```

38 }
39
40
41 extern "C" void setup_kernel()
42 {
43     // 中断管理器
44     interruptManager.initialize();
45     interruptManager.enableTimeInterrupt();
46     interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);
47
48     // 输出管理器
49     stdio.initialize();
50
51     // 进程/线程管理器
52     programManager.initialize();
53
54     // 创建第一个线程
55     int pid = programManager.executeThread(first_thread, nullptr, "first thread", 1, 4);
56     if (pid == -1)
57     {
58         printf("can not execute thread\n");
59         asm_halt();
60     }
61     programManager.executeThread(second_thread, nullptr, "second thread", 1, 4);
62     programManager.executeThread(third_thread, nullptr, "third thread", 1, 3);
63     programManager.executeThread(fourth_thread, nullptr, "fourth thread", 1, 2);
64
65     ListItem *item = programManager.readyPrograms.front();
66     PCB *firstThread = ListItem2PCB(item, tagInGeneralList);
67     firstThread->status = RUNNING;
68     programManager.readyPrograms.pop_front();
69     programManager.running = firstThread;
70     asm_switch_thread(0, firstThread);
71
72     asm_halt();
73 }

```

一共创建了 4 个线程, 其中 first\_thread 是主线程, 主线程不可返回且最先执行, 4 个线程的优先级相等, 均为被置为 1, 但线程申请的执行时间为 fourth\_thread < third\_thread < second\_thread。

运行结果如下:



可以看到，几乎是同时创建且位于就绪队列的线程 `second_thread`、`third_thread`、`fourth_thread`，但运行周期最短的线程 `fourth_thread` 最先执行，其次是 `third_thread`，最后是 `second_thread`。

调整各个线程的运行周期如下：

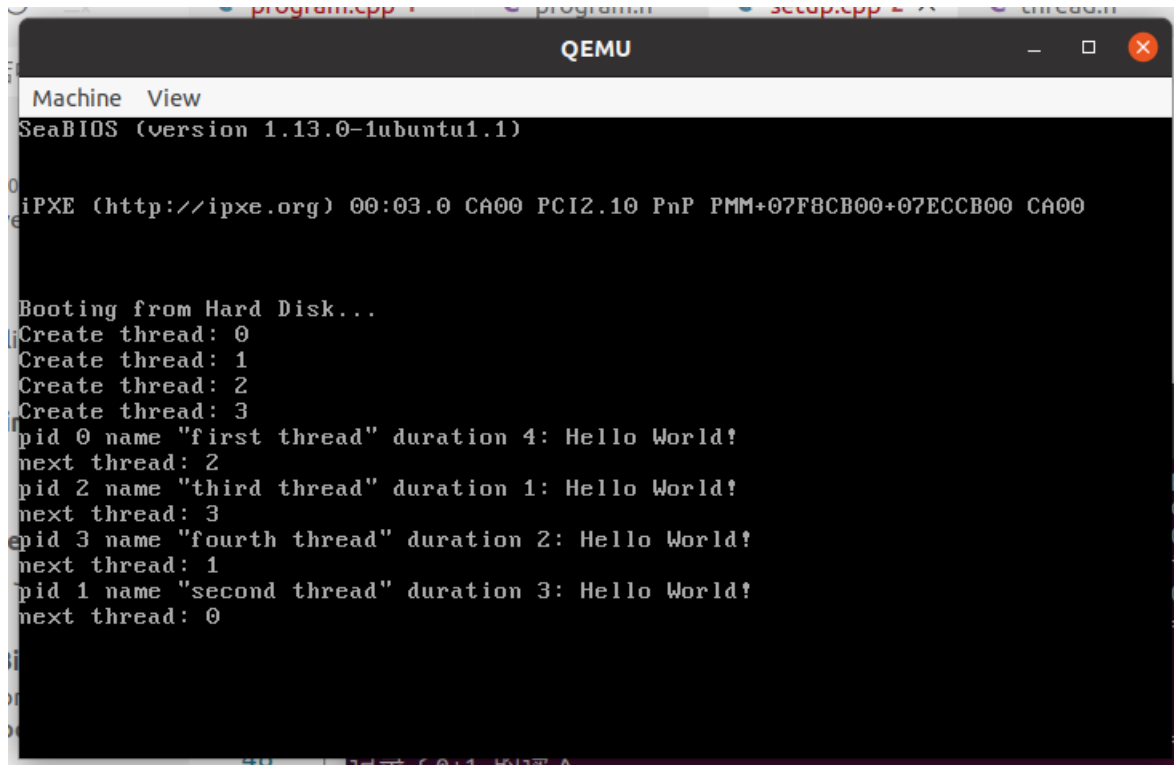
```

1 // 创建第一个线程
2 int pid = programManager.executeThread(first_thread, nullptr, "first thread", 1, 4);
3 if (pid == -1)
4 {
5     printf("can not execute thread\n");
6     asm_halt();
7 }
8 programManager.executeThread(second_thread, nullptr, "second thread", 1, 3);
9 programManager.executeThread(third_thread, nullptr, "third thread", 1, 1);
10 programManager.executeThread(fourth_thread, nullptr, "fourth thread", 1, 2);

```

线程申请的执行时间为  $\text{third\_thread} < \text{fourth\_thread} < \text{second\_thread}$ 。

运行结果如下：



```
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...
Create thread: 0
Create thread: 1
Create thread: 2
Create thread: 3
pid 0 name "first thread" duration 4: Hello World!
next thread: 2
pid 2 name "third thread" duration 1: Hello World!
next thread: 3
pid 3 name "fourth thread" duration 2: Hello World!
next thread: 1
pid 1 name "second thread" duration 3: Hello World!
next thread: 0
```

可以看到，运行周期最短的线程 `third_thread` 最先执行，其次是 `fourth_thread`，最后是 `second_thread`。

运行结果符合预期。

## 总结

通过此次实验，我对可变参数机制有了基本的认识，了解了它的原理，并掌握通过可变参数机制获取可变数量的参数的方法。

通过可变函数机制，我明白了 printf 的实现原理，并在此基础上对 printf 的功能进行拓展，从而使其能够打印更多类型的变量，使它的输出格式有更多的选择。

此外，我还基本了解了内核线程的概念、创建和调度方法。线程实际上是函数的载体，进程创建的所有线程共享进程所拥有的全部资源，进程和线程都通过 PCB 这个结构体来描述，在本次实验中我了解了简单的 PCB 结构，线程的创建实际上也就是向线程管理器申请 PCB 的内存空间并对 PCB 属性进行设置，并在此基础上学会了简单的线程调度方法，弄清楚了一个新创建的线程是如何被调度然后开始执行的以及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。在这个过程中实现了优先级调度算法、最短作业优先调度算法，但这两个方法只是在选择下一个换上处理器执行的线程时对就绪队列中的线程依其属性做一个选择，并没有实现抢占式的调度，希望能在以后的实验中有所涉及并实现。

总而言之，此次实验使我受益匪浅。