



中山大學
SUN YAT-SEN UNIVERSITY

中大校名校徽-Green

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: 可变参数机制和内核线程

专业名称: 信息与计算科学

学生姓名: 罗弘杰

学生学号: 22336173

实验地点: 实验中心D503

实验时间: 2024/3/15

Section 1 实验概述

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。

Section 2 预备知识与实验环境

略

Section 3 实验任务

该节描述需要完成的几个实验任务，即重述实验题目的总要求，建议使用项目编号分点阐述。详细要求可在下一节【实验步骤与实验结果】中列出。

实验任务1:

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。

实验任务2:

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:

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（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。

实验任务4:

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

1. 先来先服务
2. 最短作业（进程）优先
3. 响应比最高优先算法
4. 优先级调度算法
5. 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。参考资料：<https://zhuanlan.zhihu.com/p/97071815>
Tips：先来先服务最简单。有些调度算法的实现可能需要用到中断。

Section 4 实验步骤与实验结果

该节描述每个实验任务的具体的完成过程，包括思路分析、代码实现与执行、结果展示三个部分，实验任务之间的划分应当清晰明了，实验思路分析做到有逻辑、有条理。

----- 实验任务1-----

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。

思路分析:

添加输出自己的学号ID的方法：使用%v来输出这个学号ID

```
case 'v': //输出作者水印 “22336173--LHJ”
    buffer[idx] = '\0'; //最后一个字符是\0
```

```

idx = 0;
counter += stdio.print(buffer); // 将缓冲区中的内容输出
counter += printf_add_to_buffer(buffer, '2', idx, BUF_LEN);
counter += printf_add_to_buffer(buffer, '2', idx, BUF_LEN);
counter += printf_add_to_buffer(buffer, '3', idx, BUF_LEN);
counter += printf_add_to_buffer(buffer, '3', idx, BUF_LEN);
counter += printf_add_to_buffer(buffer, '6', idx, BUF_LEN);
counter += printf_add_to_buffer(buffer, '1', idx, BUF_LEN);
counter += printf_add_to_buffer(buffer, '7', idx, BUF_LEN);
counter += printf_add_to_buffer(buffer, '3', idx, BUF_LEN);
counter += printf_add_to_buffer(buffer, '--', idx, BUF_LEN);
counter += printf_add_to_buffer(buffer, 'L', idx, BUF_LEN);
counter += printf_add_to_buffer(buffer, 'H', idx, BUF_LEN);
counter += printf_add_to_buffer(buffer, 'J', idx, BUF_LEN);
break;

```

实验结果：

```

b
d SeaBIOS (version 1.10.2-1ubuntu1)
+
+
1: iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD0+07ECDDDD0 C980
d
+
+
+
8: Booting from Hard Disk...
d print percentage: %
d print char "N": N
+ print string "Hello World!": Hello World!
+ print decimal: "-1234": -1234
8: print hexadecimal "0x7abcdef0": 7ABCDEF0
o print MY_ID: 223361736LHJ
er-
IAF
w.
er-
og
er
IAF
w.
Automatically detecting the format is dangerous for raw images, write on

```

----- 实验任务2 -----

该任务需要更新自己的PCB,更改线程调度方式,我在实验资料基础上,实现了优先级优先调度方法,在任务4中会一并说明

实验任务3

任务要求:

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 (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。

思路分析:

设计四个线程的程序，每一个线程对应一个函数，函数中先输出该线程的信息，然后进入死循环；、

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

asm_halt()
```

```
//四个函数对应四个线程
void fourth_thread(void *arg) {
    printf("pid %d name \"%s\": Hello world!\n", programManager.running->pid,
programManager.running->name);
    asm_halt();
}

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

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

```

}

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

```

调试命令：

```

b setup_kernel
b asm_switch_thread; 线程调度函数需要调用这个函数来切换栈指针
b c_time_interrupt_handler ; 这是中断处理函数，每一次中断都会使得时间片减1，若为0就进入线程调度函数
b schedule; 这是线程调度函数

```

调试与结果展示：

首先介绍第一个线程手动被调入处理器的过程。

如下图，进入asm_switch_thread函数，先保存原来线程的栈指针，可以看到原来栈指针的起始位置是0x7bc0,由于栈是从高位到低位的，在压入4个32位寄存器以后变为0x7bb0(减去4个4字节)

```

QEMU [Stopped]
SeaBIOS (version 1.10.2-1ubuntu1)
iPXE File Edit View Search Terminal Help
Boot
B+
B+>
remote Thread 1 In: asm_switch_thread L29 PC: 0x214e0
eax 0x21d80 138624
ecx 0x24d80 150912
edx 0x0 0
ebx 0x39000 233472
esp 0x7bb0 0x7bb0
ebp 0x7bfc 0x7bfc
esi 0x0 0
#--Type <return> to continue, or q <return> to quit--
t.
rogers2@ubuntu:~/os/lab5/task2/build$

```

下一步，转化esp到下一个线程的栈指针，可以看到PCB_SET起始也就是线程1和eax是一致的

```
Terminal
File Edit View Search Terminal Help
../src/utils/asm_utils.asm
28
B+ 29      mov eax, [esp + 5 * 4]
30      mov [eax], esp ; 保存当前栈指针到PCB中，以便日后
31
32      mov eax, [esp + 6 * 4]
B+> 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

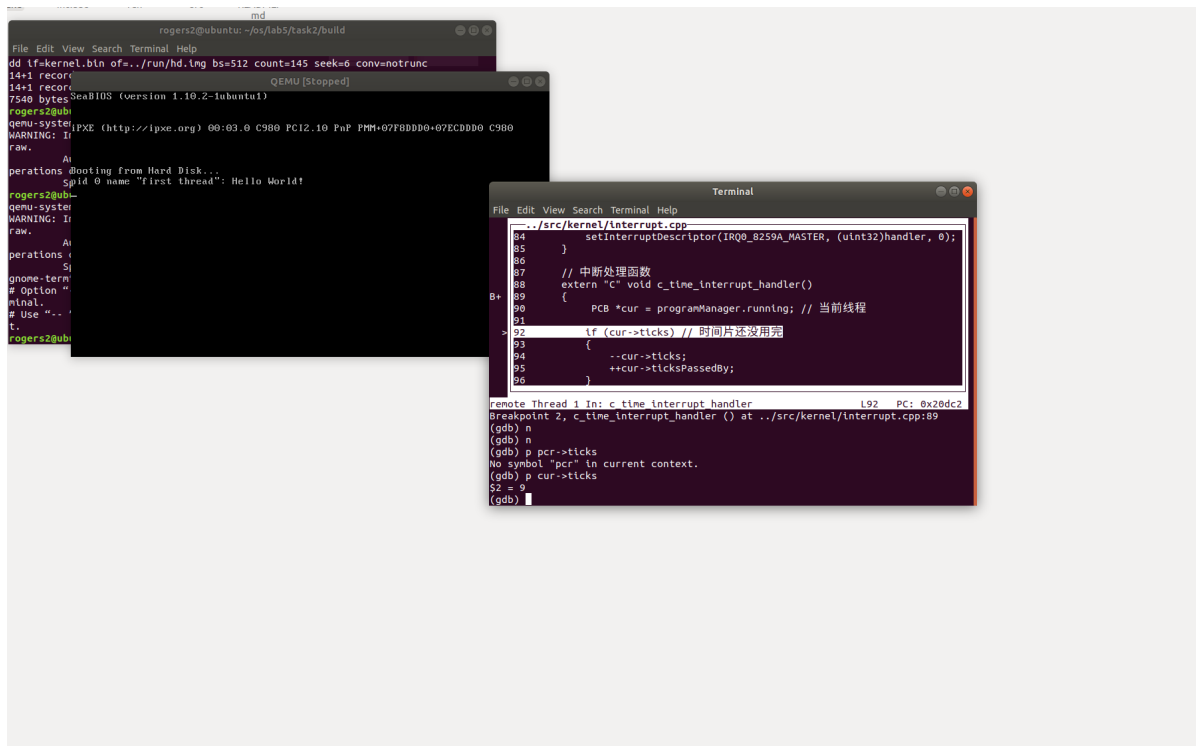
remote Thread 1 In: asm_switch_thread L33 PC: 0x214ea
esi 0x0 0
---Type <return> to continue, or q <return> to quit---
Quit
(gdb) p &PCB_SET
$1 = (char (*)[65536]) 0x21d80 <PCB_SET>
(gdb) info registers eax
eax 0x21d80 138624
(gdb)
```

在完成线程切换以后，查看

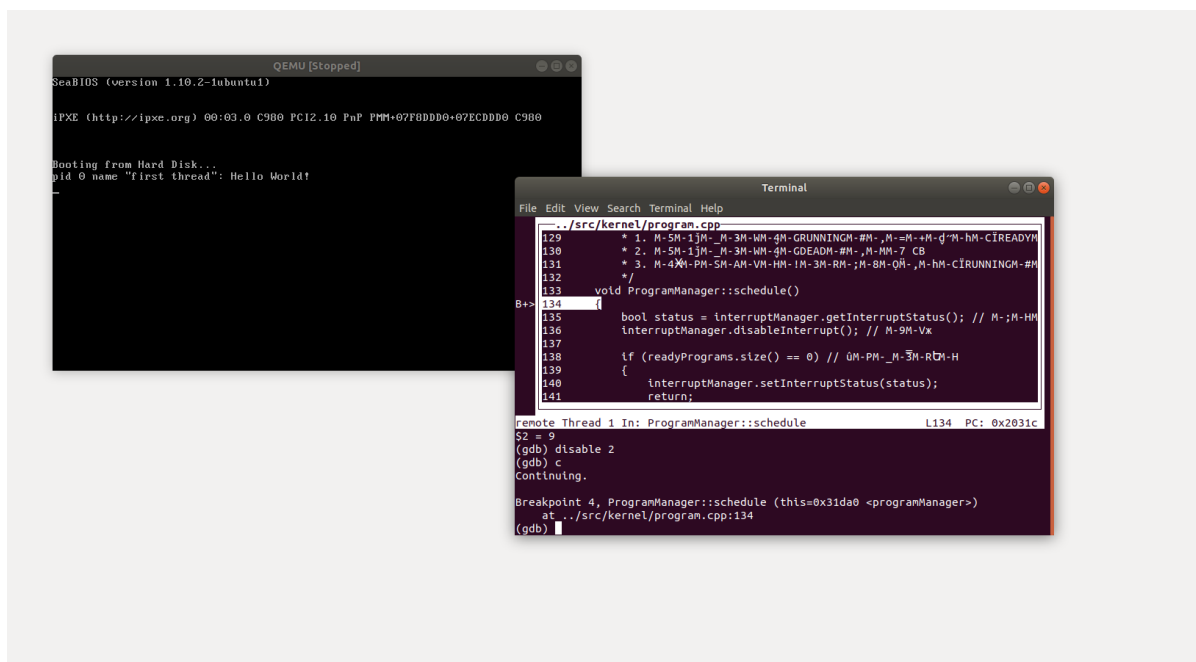
```
Terminal
File Edit View Search Terminal Help
../src/utils/asm_utils.asm
28
B+ 29      mov eax, [esp + 5 * 4]
30      mov [eax], esp ; 保存当前栈指针到PCB中，以便日后
31
32      mov eax, [esp + 6 * 4]
B+ 33      mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
34
> 35      pop esi
36      pop edi
37      pop ebx
38      pop ebp
39
40      sti

remote Thread 1 In: asm_switch_thread L35 PC: 0x214ec
(gdb) info registers esp
esp 0x22d64 0x22d64 <PCB_SET+4068>
(gdb) x/5w 0x22d64
0x22d64 <PCB_SET+4068>: 0 0 0 0
0x22d74 <PCB_SET+4084>: 132341
(gdb) info symbol 132341
first_thread(void*) in section .text
(gdb)
```

通过栈指针的切换，成功跳转到线程1指向的函数1：



如图，进入schedule函数



线程1还在执行态，将其改为就绪态，加入就绪队列，恢复时间片，根据RR调度方式选择线程2


```
Terminal
File Edit View Search Terminal Help
../src/utils/asm_utils.asm
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
31
32     mov eax, [esp + 6 * 4]
B+> 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

remote Thread 1 In: asm_switch_thread L33 PC: 0x214ea
Continuing.

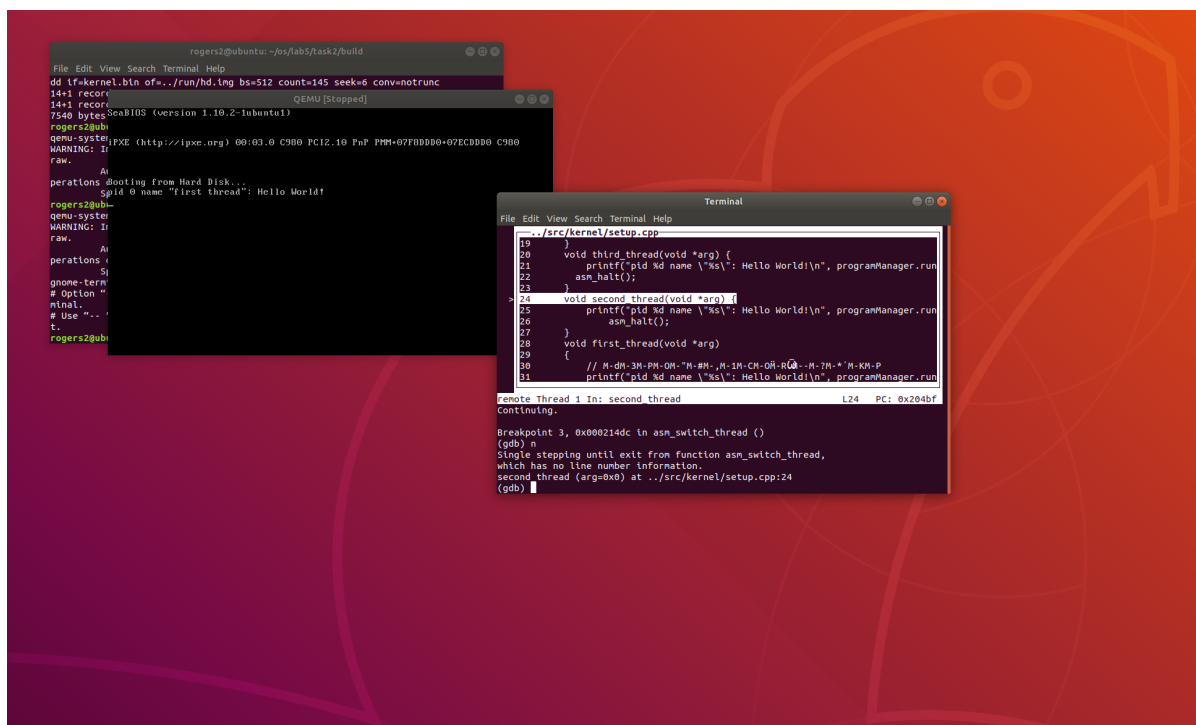
Breakpoint 4, asm_switch_thread () at ../src/utils/asm_utils.asm:33
(gdb) info registers eax
eax                0x22d80  142720
(gdb) info symbol 0x22d80
PCB_SET + 4096 in section .bss
(gdb)
```

查看切换后的栈指针, 发现其指向线程2, 继续跳转, 发现果然进入到第二个函数。

```
rogers2@ubuntu: ~/os/lab5/task2/build
QEMU [Stopped]
SeaBIOS (version 1.10.2-1ubuntu1)
ipXE File Edit View Search Terminal Help
ntu Software
gnom
# OpBoot
mina pid
# Us-
t.
roge
qemu
WARN
raw.
pera
gnom
# Op
mina
# Us
t.
rogers2@

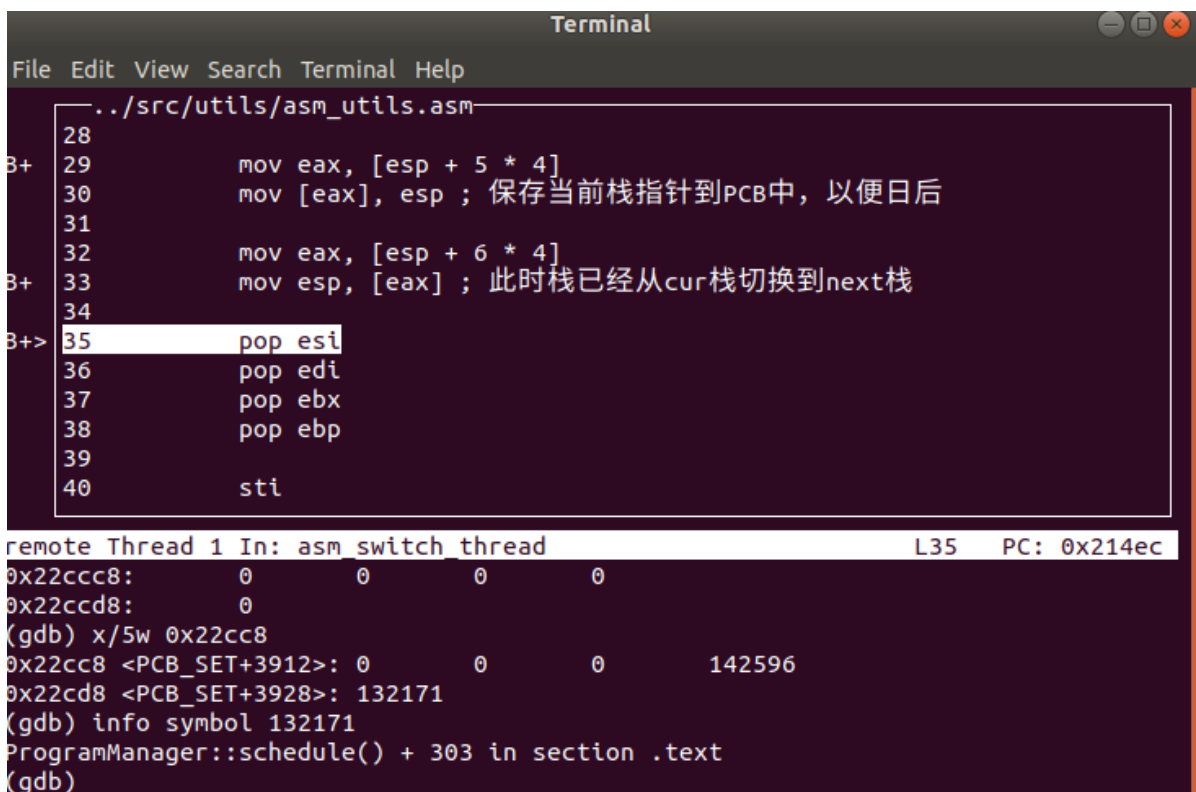
../src/utils/asm_utils.asm
28
29     mov eax, [esp + 5 * 4]
30     mov [eax], esp ; 保存当前栈指针到PCB中, 以便日后
31
32     mov eax, [esp + 6 * 4]
B+> 33     mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
34
35     pop esi
36     pop edi
37     pop ebx
38     pop ebp
39
40     sti

remote Thread 1 In: asm_switch_thread L35 PC: 0x214ec
(gdb) info registers esp
esp                0x23d64  0x23d64 <PCB_SET+8164>
(gdb) x/5w 0x23d64
0x23d64 <PCB_SET+8164>: 0      0      0      0
0x23d74 <PCB_SET+8180>: 132293
(gdb) info symbol 132293
second_thread(void*) in section .text
(gdb)
```



最后阐释，在所有线程流转过以后，查看线程循环的过程（怎么切回到线程1）：

在线程1切换的时候，时钟中断处理函数调用`schedule`，`schedule`调用`asm_switch_thread`，因此在`asm_switch_thread`中我们有返回地址，查看这个返回地址，发现是`schedule`



同理，在时钟中断处理函数中，我们会返回到线程1，如图：

由于所有线程都是先输出字符串然后陷入死循环，所以在后续的线程切换中，我们实际上是在死循环中切换

```
Terminal
File Edit View Search Terminal Help

../src/utils/asm_utils.asm
64         out 0x20, al
65         out 0xa0, al
66
67         call c_time_interrupt_handler
68
69         popad
> 70        iret
71
72         ; void asm_in_port(uint16 port, uint8 *value)
73         asm_in_port:
74             push ebp
75             mov ebp, esp
76
remote Thread 1 In: asm time interrupt handler      L70    PC: 0x2150d
(gdb) info registers esp
esp                0x22d5c  0x22d5c  <PCB_SET+4060>
(gdb) x/5w 0x22d5c
0x22d5c <PCB_SET+4060>: 136667  32      518      132386
0x22d6c <PCB_SET+4076>: 0
(gdb) info symbol 136667
asm_halt in section .text
(gdb)
```

总结

在线程切换时，需要注意以下几个关键点：

1. **保存和恢复上下文**：在线程切换时，需要保存当前线程的执行状态（包括寄存器的值、程序计数器、堆栈指针等）到线程的控制块（Thread Control Block, TCB）中，以便之后能够正确地恢复执行状态。同时，需要从下一个要执行的线程的TCB中恢复其执行状态。**在操作系统中，保护进程的关键机制确实是通过进程控制块（PCB）和栈指针（Stack Pointer）来实现的**
2. **同步共享资源**：在线程切换时，需要确保共享资源的一致性。如果有多个线程共享同一份数据或资源，那么在线程切换时需要采取适当的同步措施，以避免出现竞态条件或数据不一致的情况。
3. **调度策略**：线程切换的时机和顺序由调度器负责管理。因此，在线程切换时，需要考虑当前的调度策略，以确保按照优先级、时间片轮转等规则选择下一个要执行的线程。
4. **性能开销**：线程切换是有成本的，包括保存和恢复上下文的开销、同步共享资源的开销等。因此，在设计和实现线程切换时，需要考虑性能开销，并尽量减少不必要的线程切换。
5. **死锁和饥饿**：在线程切换时，需要避免死锁和饥饿等并发编程中常见的问题。确保线程切换不会导致线程之间的相互等待，以及某些线程长时间无法获得执行的情况。

----- 实验任务2&4 -----

任务要求：

实现线程的优先级调度方法

在准备队列中线程会按照优先级大小排列，弹出的是优先级最高的线程进入执行队列。

代码分析：

setup.cpp中更改线程函数为不进入死循环，取消时钟中断函数

```
extern "C" void setup_kernel()
{
```

```

// 初始化类
interruptManager.initialize();
interruptManager.disableTimeInterrupt(); // 开启时钟中断
interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler); // 设置
时钟中断处理函数
    stdio.initialize();
    programManager.initialize();

// 线程初始化
    int pid0 = programManager.executeThread(first_thread, nullptr, "first
thread", 1); //优先级为1
    int pid1 = programManager.executeThread(second_thread, nullptr, "second
thread", 3); //优先级为3
    int pid2 = programManager.executeThread(third_thread, nullptr, "third
thread", 2); //优先级为2
    int pid3 = programManager.executeThread(fourth_thread, nullptr, "fourth
thread", 4); //优先级为4
    if (pid0 == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }
//手动切换到第一个线程
    ListItem *item = programManager.readyPrograms.front();
    PCB *firstThread = ListItem2PCB(item, tagInGeneralList);
    firstThread->status = RUNNING;
    programManager.readyPrograms.pop_front();
    programManager.running = firstThread;
    asm_switch_thread(0, firstThread); //从内核态切换到用户态

    asm_halt(); //
}

```

在list类中添加一个按照优先级插入元素的函数，以后准备队列会调用这个函数来添加线程

```

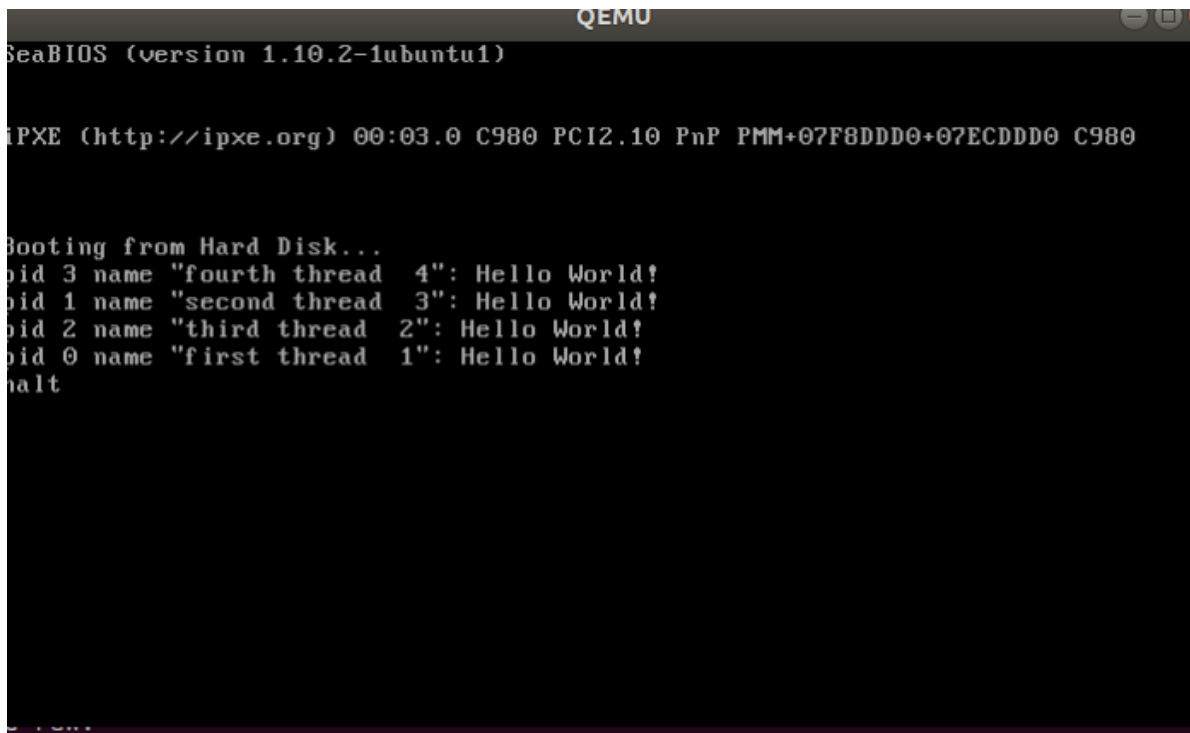
void List::insert_priority(ListItem *itemPtr, int priority) //按照优先级插入
{
    ListItem *temp = head.next;
    while (temp && ListItem2PCB(temp, tagInGeneralList)->priority > priority) //
找到第一个优先级比它小的
    {
        temp = temp->next;
    }
    if (temp) //找到了
    {
        itemPtr->next = temp;
        itemPtr->previous = temp->previous;
        temp->previous->next = itemPtr;
        temp->previous = itemPtr;
    }
    else //没找到
    {
        push_back(itemPtr);
    }
}

```

在每一个线程结束后，会进入返回函数`program_exit()`，这个函数会判断线程为死亡态，然后调用`schedule()`函数，将准备队列的队首也就是优先级最高的线程调入处理器。

成果展示：

可以看到优先级为4的线程4先执行，然后是优先级为3的线程2，然后是优先级为2的线程2，最后是优先级为1的线程1。由此实现了按照优先级调度的调度算法。



```
SeaBIOS (version 1.10.2-1ubuntu1)

IPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD0+07ECDDDD0 C980

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

Section 5 实验总结与心得体会

printf的实现

在C语言中，`printf` 函数是一个可变参数函数，即它可以接受不定数量的参数。它的原型在 `stdio.h` 头文件中定义如下：

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

其中，`...` 表示可变参数列表，可以接受任意数量的参数。`printf` 函数通过 `format` 参数来识别格式字符串，并根据格式字符串中的格式说明符来确定需要提取的参数数量和类型。

当 `printf` 函数在运行时遇到格式字符串中的格式说明符时，它会从参数列表中按顺序取出相应的参数，根据格式说明符的要求对参数进行格式化，并输出到标准输出流中。

例如，对于下面的 `printf` 调用：

```
printf("Hello, %s! You are %d years old.\n", "Alice", 25);
```

`printf` 会首先识别 `%s` 格式说明符，然后从参数列表中取出下一个参数 `"Alice"`，将其作为字符串输出；然后识别 `%d` 格式说明符，再从参数列表中取出下一个参数 `25`，将其作为整数输出。

识别参数

printf 函数通常使用了宏来实现对不定参数的处理。在标准库的实现中，常见的方式是使用 stdarg.h 头文件中提供的宏来处理可变参数列表。这些宏包括了 va_list、va_start、va_arg 和 va_end。

va_list: 这是一个类型，用于声明一个指向参数列表的对象。

va_start: 这个宏用于初始化 va_list 对象，以便开始访问参数列表。

va_arg: 这个宏用于访问 va_list 对象中的下一个参数，并返回其值。

va_end: 这个宏用于结束对参数列表的访问。

线程的实现，调度

实现

实现线程时，PCB（进程控制块）是一个重要的数据结构，用于描述和管理线程的状态和相关信息。在设计和实现 PCB 时，需要考虑以下几个方面：

1. **线程状态管理**：PCB 应该包含线程的状态信息，如就绪、运行、阻塞等状态，以及相应的状态转换操作。
2. **线程标识**：每个线程都应该有一个唯一的标识符，PCB 中应包含线程的标识信息，以便于系统对线程进行唯一标识和管理。
3. **线程上下文**：PCB 中应该保存线程的上下文信息，包括程序计数器（PC）、堆栈指针（SP）、寄存器状态等，以便于线程的上下文切换和恢复。
4. **线程优先级**：如果系统支持线程优先级调度，PCB 应该包含线程的优先级信息，以便于调度器根据优先级来进行调度决策。
5. **线程资源管理**：PCB 应该包含线程所拥有的资源信息，如打开的文件、分配的内存等，以便于资源的管理和释放。
6. **同步与通信**：如果线程需要进行同步和通信操作，PCB 应该包含相应的同步和通信机制，如互斥锁、条件变量等。
7. **调度信息**：PCB 中可以包含与调度相关的信息，如线程的调度状态、调度优先级、等待时间等，以便于调度器对线程进行调度和管理。
8. **扩展信息**：根据需要，PCB 还可以包含其他扩展信息，如线程的创建时间、运行时间、等待时间等，以便于系统对线程的统计和监控。

综上所述，PCB 在实现线程时起着关键作用，它是线程状态和相关信息的载体，对于线程的管理和调度具有重要意义。因此，在设计和实现 PCB 时，需要充分考虑线程的各种需求和特性，确保 PCB 能够满足线程管理的各种要求。

调度

调度算法是操作系统中的关键组成部分，负责决定哪些进程或线程应该在 CPU 上运行以及运行多长时间。以下是在设计调度算法时需要考虑的一些重要因素：

1. **公平性**：确保所有进程或线程都有公平的机会获得 CPU 时间，避免某些进程或线程长时间占用 CPU 而导致其他进程或线程无法执行的情况。
2. **优先级**：考虑进程或线程的优先级，确保优先级高的进程或线程能够优先被调度执行。不同的调度算法可能采用不同的优先级策略，如静态优先级、动态优先级等。
3. **响应时间**：考虑进程或线程的响应时间，确保关键任务能够在规定的时间内得到及时执行。
4. **吞吐量**：考虑系统的吞吐量，即单位时间内完成的进程或线程数量，以提高系统的整体性能。
5. **CPU利用率**：优化调度算法，以提高 CPU 的利用率，尽可能地保持 CPU 处于忙碌状态。

6. **上下文切换开销**：尽量减少调度算法导致的上下文切换开销，以提高系统的运行效率和性能。
7. **适应性**：考虑系统的负载情况和运行环境变化，选择合适的调度算法以适应不同的工作负载和环境需求。
8. **实时性**：对于实时系统，确保调度算法能够满足实时性的要求，保证关键任务能够在规定的时间内得到执行。

与时钟中断函数的配合：

时钟中断函数是调度算法的重要组成部分，它周期性地被调用，用于触发调度器对进程或线程进行调度。在设计时钟中断函数时需要考虑以下几点：

1. **触发频率**：时钟中断函数的触发频率应该合适，既要保证系统的实时性，又要尽量减少中断开销。
2. **调度决策**：在时钟中断函数中，调度器根据当前的调度策略和进程或线程的状态进行调度决策，并决定下一个要执行的进程或线程。
3. **上下文切换**：如果发生了进程或线程的切换，时钟中断函数负责保存当前进程或线程的上下文，并加载下一个进程或线程的上下文，以实现上下文切换。
4. **调度算法优化**：可以在时钟中断函数中进行调度算法的优化，如动态调整优先级、调整时间片大小等，以适应不同的工作负载和系统需求。

综上所述，调度算法和时钟中断函数之间密切配合，共同确保系统的稳定性、实时性和性能。调度算法决定了进程或线程的执行顺序和时长，而时钟中断函数则负责周期性地触发调度器进行调度，以实现进程或线程的合理调度和管理。

有些调度算法需要时钟中断的配合比如时间片轮转调度；有些则不需要，比如优先级调度，先到先服务等。