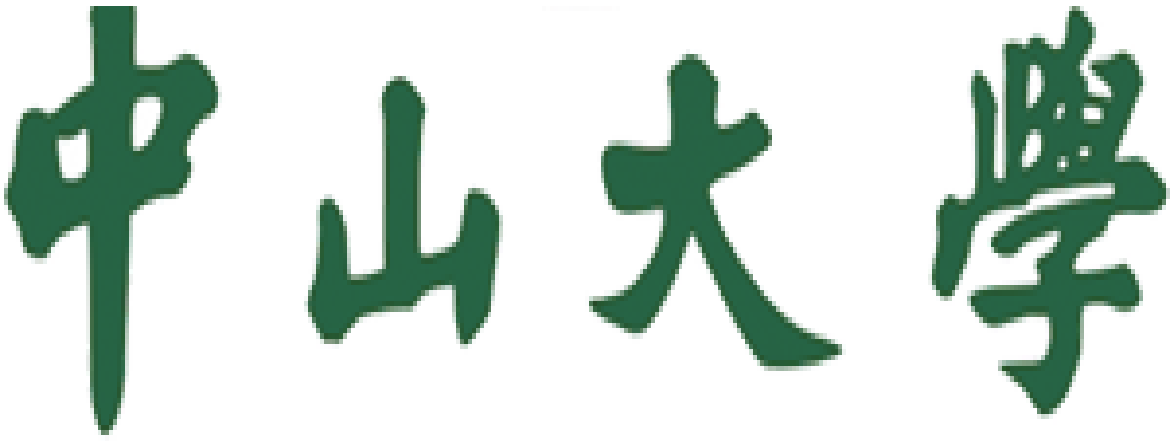


实验课程：操作系统

实验名称：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：吴臻

学生学号：21307371

实验地点：实验中心大楼D栋501

实验成绩：

报告时间：2023/4/22

1. 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。

2. 预备知识与实验环境

3. 实验任务

1. printf的实现
2. 线程的实现
3. 时钟中断的处理
4. 调度算法的实现

4. 实验步骤/关键代码/实验结果

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

实验步骤：

1. 增加二进制和八进制的输出

2. 增加浮点数的输出（默认保留后六位）

   > 先用int类型转换得到浮点数的整数部分，然后用整数的处理方式输出，对于小数部分，将小数乘十（将小数点右移一位）得到其整数并输出，不断重复

> 上述操作即可

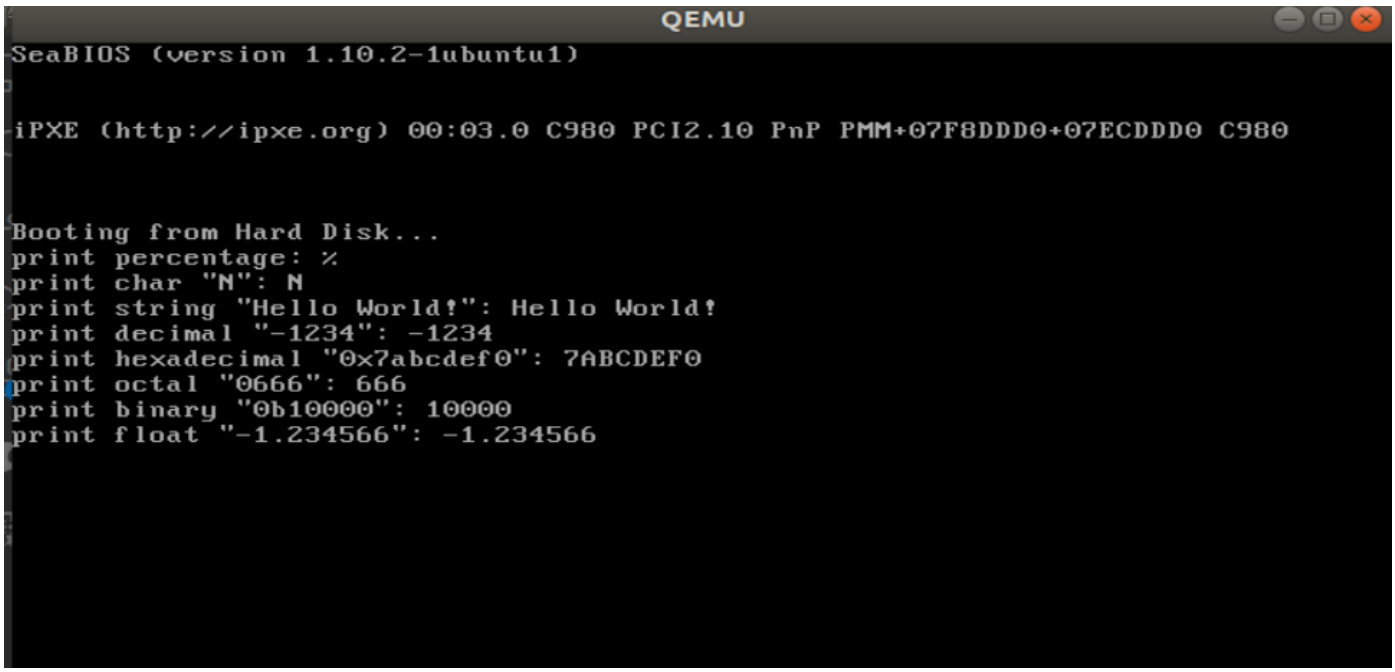
关键代码：

```
//十进制
case 'd':
//十六进制
case 'x':
//八进制
case 'o':
//二进制
case 'b':
    int system = 10;
    int temp = va_arg(ap, int);

    if (temp < 0 && fmt[i] == 'd')
    {
        counter += printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
        temp = -temp;
    }
    //system为转换进制数
    switch(fmt[i]){
        //十进制
        case 'd':
            system = 10;
            break;
        //十六进制
        case 'x':
            system = 16;
            break;
        //八进制
        case 'o':
            system = 8;
            break;
        //二进制
        case 'b':
            system = 2;
            break;
        default:
            system = 10;
            break;
    }
    itos(number, temp, system);

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

```
case 'f':
{
```

```
        double f = va_arg(ap,double);
        if(f < 0){
            counter += printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
            f = -f;
        }
        //整数部分
        int integer = static_cast<int> (f);
        //小数部分
        double dec = f - static_cast<double>(integer);
        //处理整数
        itos(number,integer,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);
        //处理小数(六位)
        for(int pos = 0; pos < 6; pos++){
            int num = int(dec * 10 );
            char c = num + '0';
            counter += printf_add_to_buffer(buffer, c, idx, BUF_LEN);
            dec = (dec*10) - num;
        }
        break;
    }
```

结果展示：



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。

实验步骤

1. 在PCB结构中加入fpid变量来记录父进程的编号（第一个进程默认fpid=-1）
2. 在调用该进程的函数时，输出父进程的编号

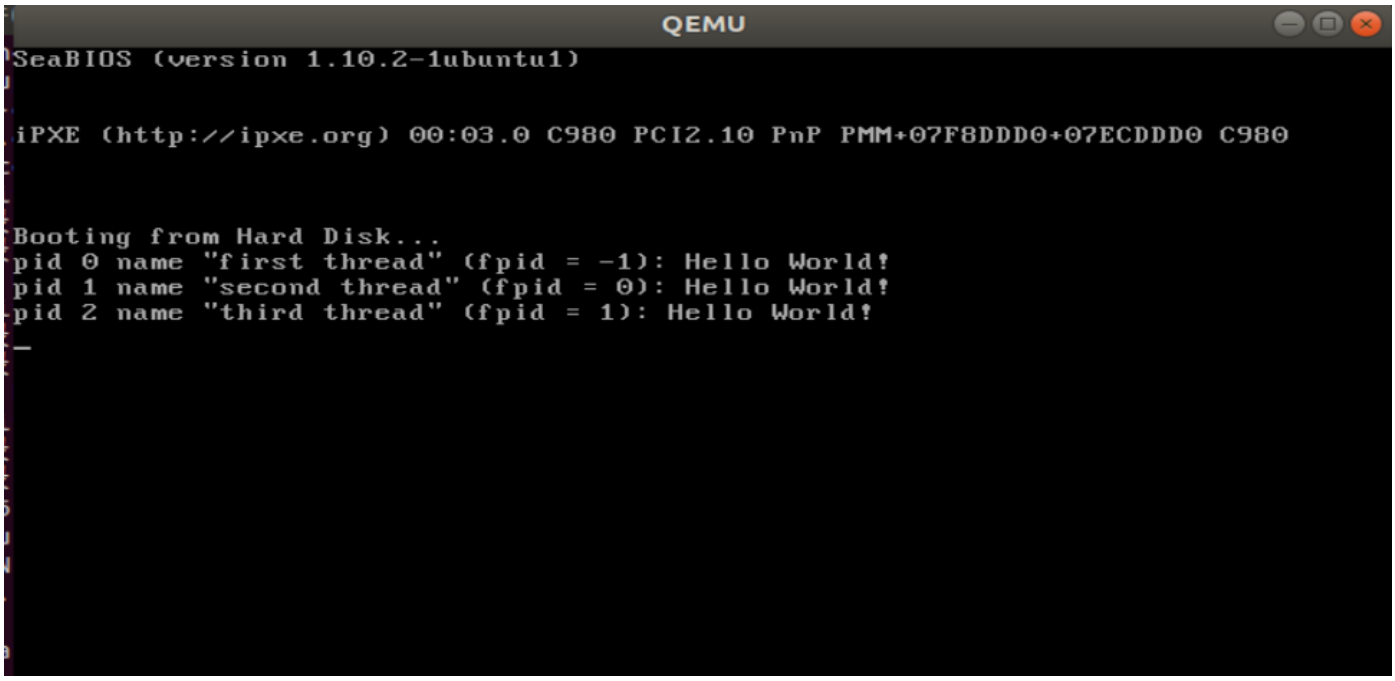
关键代码

```
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 fpid;                       //父进程
    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
};
```

```
void third_thread(void *arg) {
    printf("pid %d name \"%s\" (fpid = %d): Hello World!\n",
programManager.running->pid, programManager.running->name,programManager.running-
>fpid);
    while(1) {
            // for (int i = 0; i < 100000000; i++)
            //     if(i % 2500000 == 0)
            //         printf("Thread2 time: %d\n",programManager.running -
>ticksPassedBy);
    }
}
void second_thread(void *arg) {
    printf("pid %d name \"%s\" (fpid = %d): Hello World!\n",
programManager.running->pid, programManager.running->name,programManager.running-
>fpid);
    programManager.executeThread(third_thread, nullptr, "third thread", 1);
    // for (int i = 0; i < 100000000; i++)
    //     if(i % 2500000 == 0)
    //         printf("Thread1 time: %d\n",programManager.running ->ticksPassedBy);
}
void first_thread(void *arg)
{
    // 第1个线程不可以返回
    printf("pid %d name \"%s\" (fpid = %d): Hello World!\n",
programManager.running->pid, programManager.running->name,programManager.running-
>fpid);
    if (!programManager.running->pid)
    {
        programManager.executeThread(second_thread, nullptr, "second thread", 1);
        //programManager.executeThread(third_thread, nullptr, "third thread", 1);
```

```
    }
    asm_halt();
}
```

实验结果



Assignment 3 时钟中断的处理

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，以及换上处理器后又是如何从被中断点开始执行的

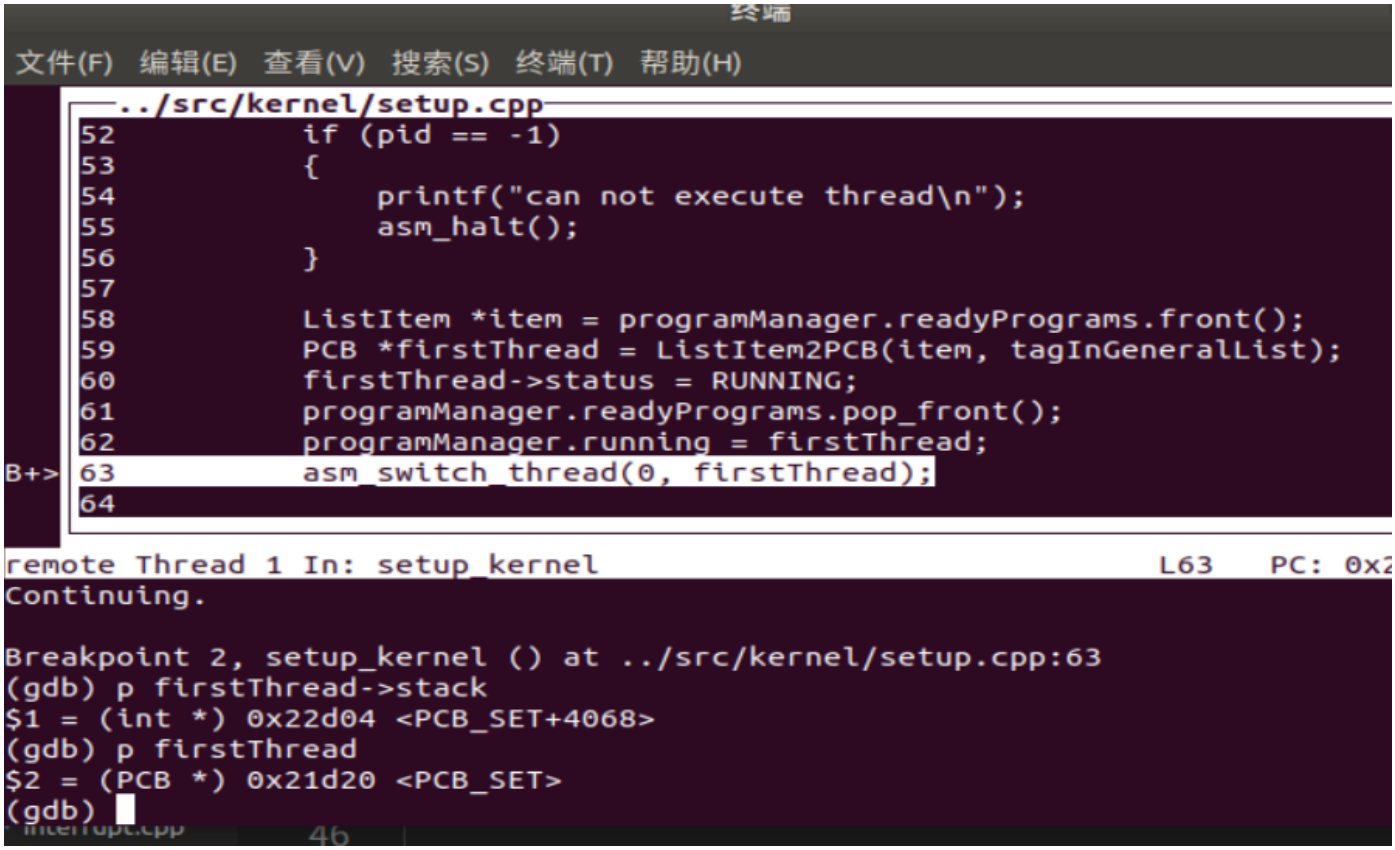
实验步骤（附代码和结果展示）

1. 初始化，并创建第一个进程

- 关中断；
- 申请对应线程空间；
- 初始化线程栈空间；
- 将 PCB 加入全进程队列和就绪队列；
- 开中断。

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

// 输出管理器
stdio.initialize();

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

// 创建第一个线程
int pid = programManager.executeThread(first_thread, nullptr, "first thread", 1,
-1);
```

第一个PCB地址



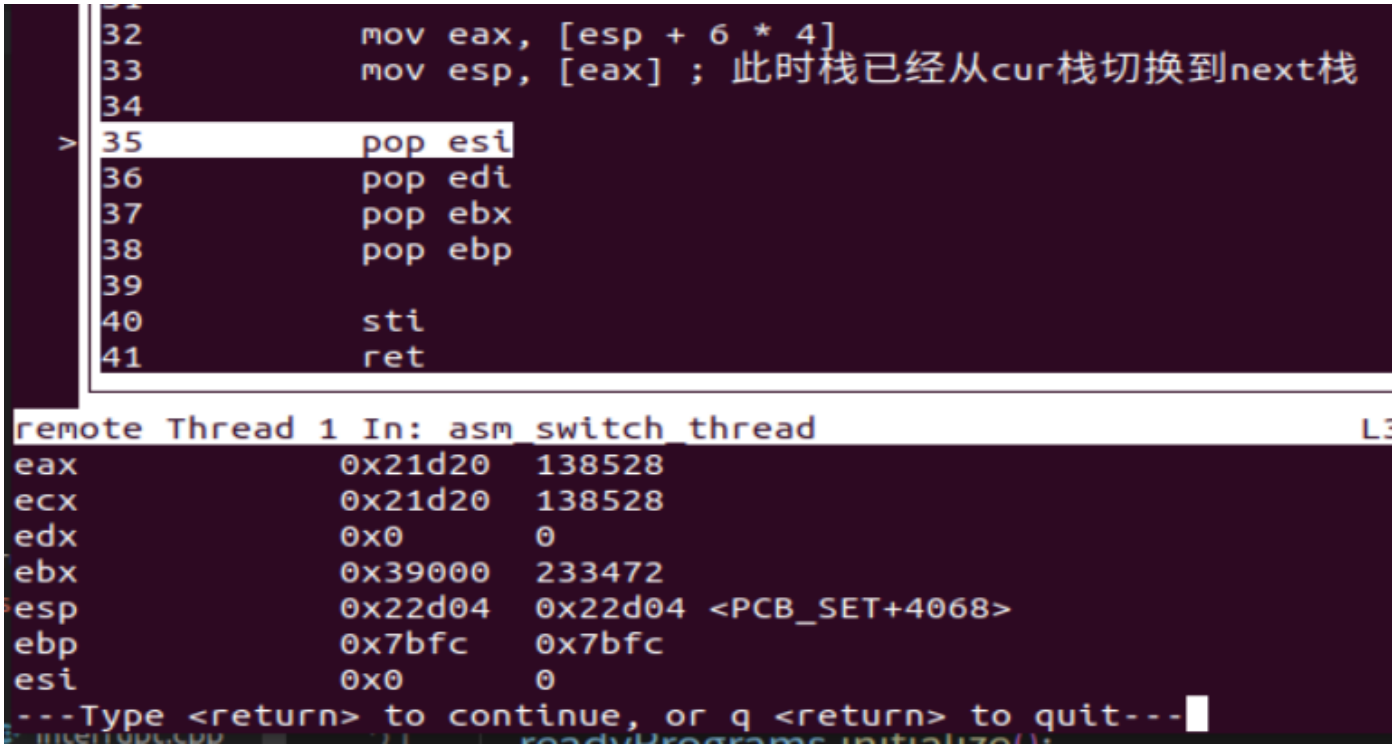
2. 创建完第一个进程并将其放入ready队列，因为没有其他进程所以需要手动进行上下文切换

> 解释以下汇编代码

```
push ebp
push ebx
push edi
push esi
mov eax, [esp + 5 * 4]
mov [eax], esp ; 保存当前栈指针到PCB中，以便日后恢复
```

```
mov eax, [esp + 6 * 4]
mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
```

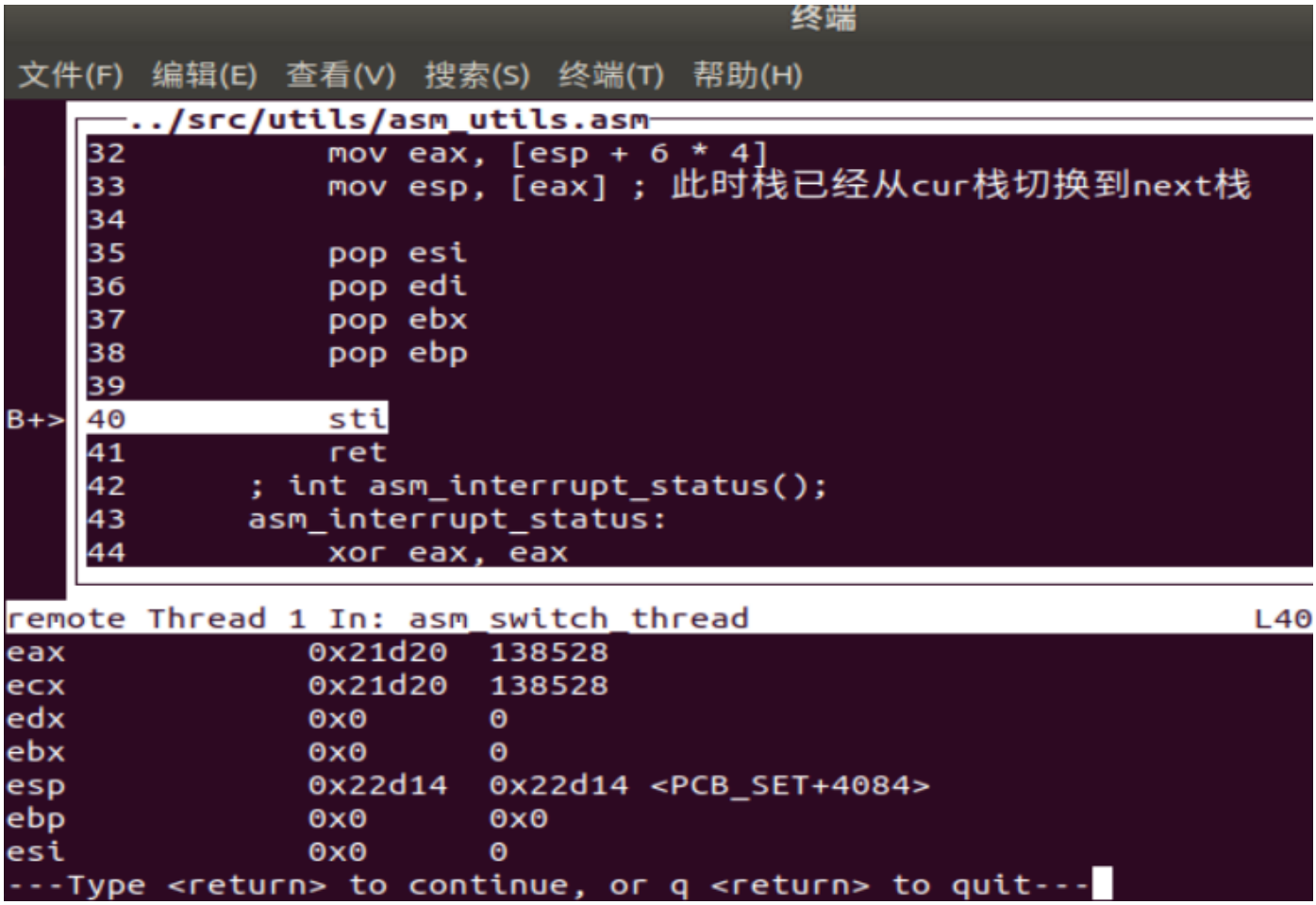
> 调用asm_switch_thread(0, firstThread)时，将firstThread和0依次压入栈中，然后进入该函数，执行以上代码，函数一开始先压入四个寄存器，所以 `[esp + 5 * 4]`为0，将此时的栈指针放入0地址，`[esp + 6 * 4]`为firstThread（PCB*指针），指针（无偏移）一开始指向PCB首地址（即存放stack指针的地址），取其内容（得到stack）并赋值给esp就完成了栈的切换（stack初始化时预留了七个位置，所以指向PCB_SET+4068）



由下图可以看到，PCB_SET+4084存着第一个线程需要执行的函数

---

asm_switch_thread函数执行到ret时，esp指向PCB_SET+4084, 调用ret即可跳转到第一个线程需要执行的函数



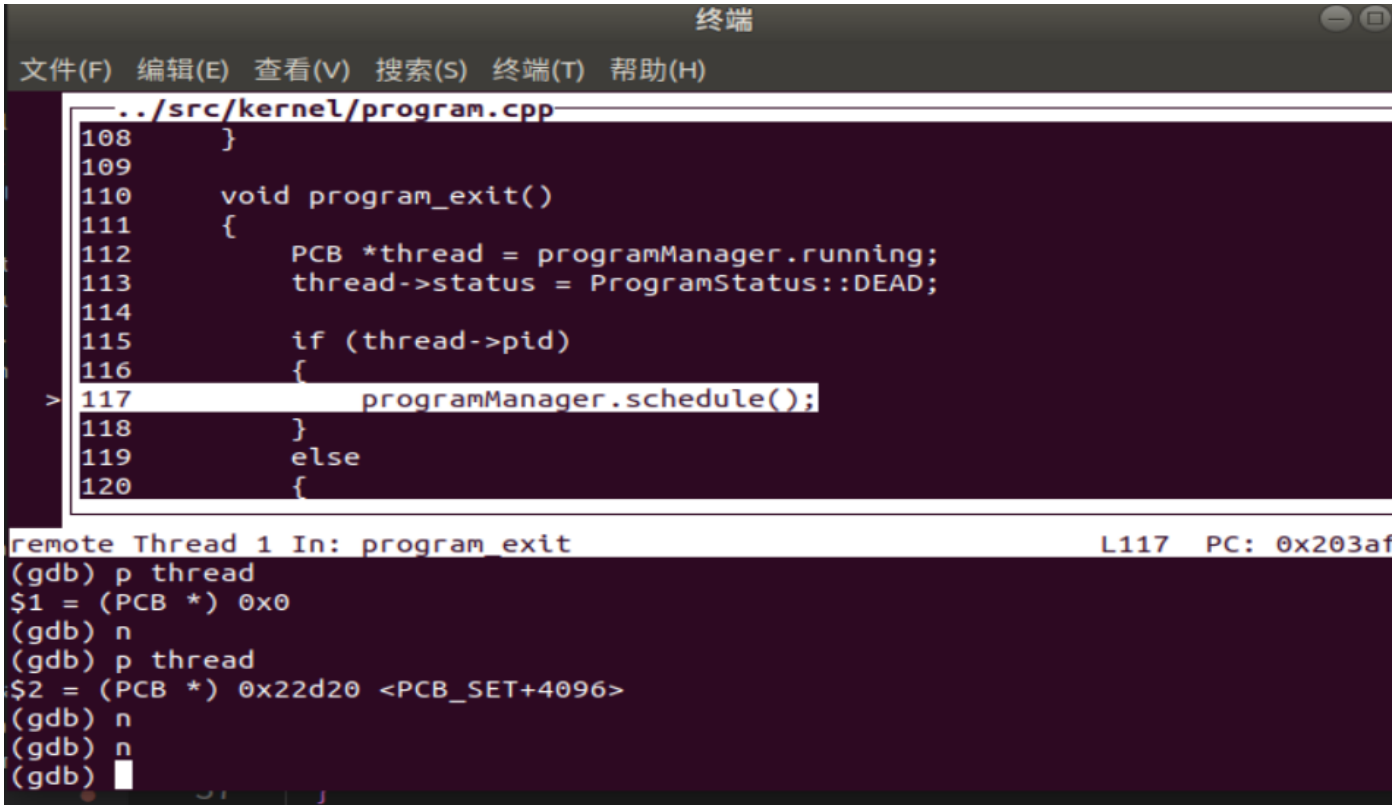

```
void first_thread(void *arg)
{
    // 第1个线程不可以返回
```

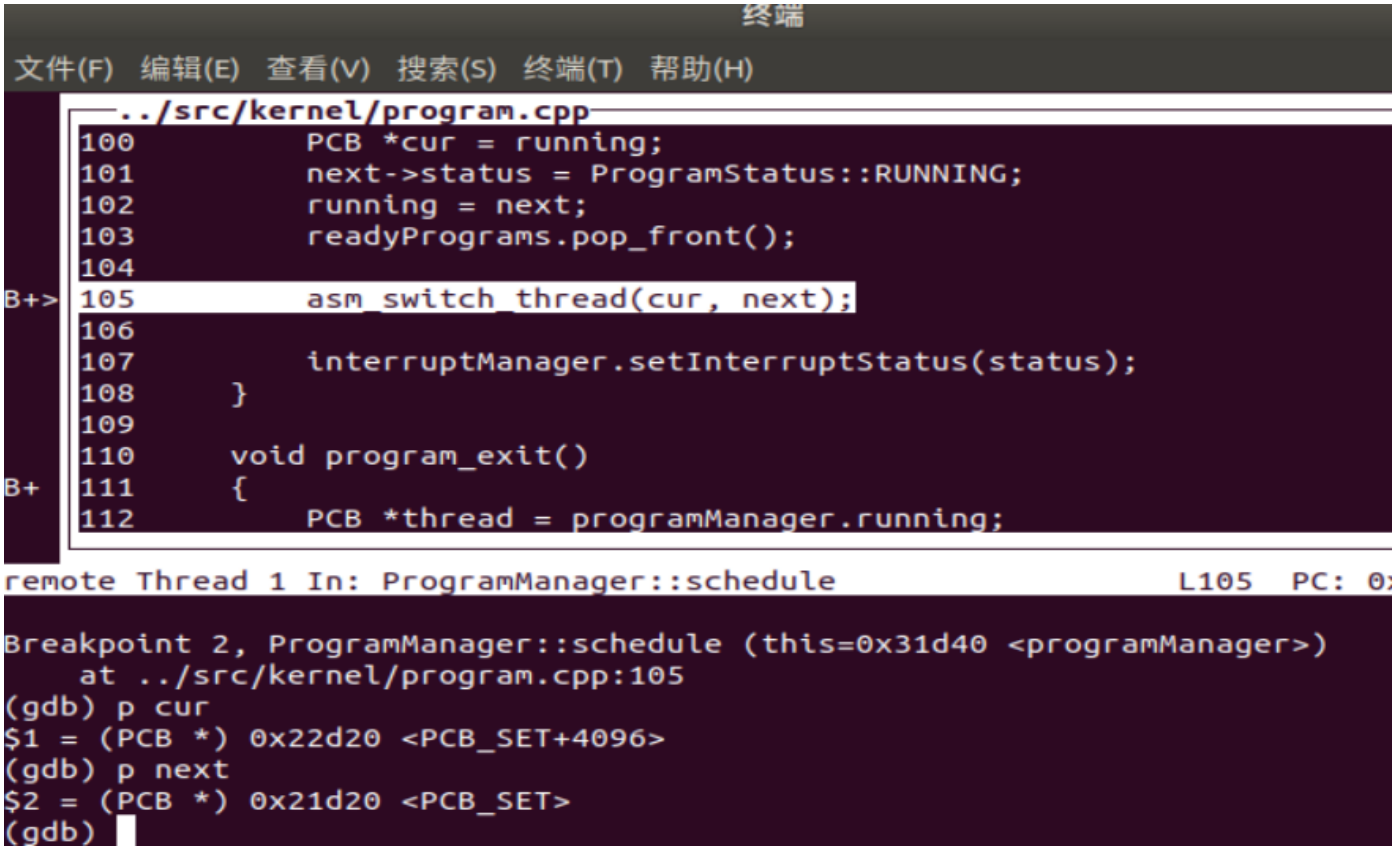
```
    printf("pid %d name \"%s\" (fpid = %d): Hello World!\n",
programManager.running->pid, programManager.running->name,programManager.running-
>fpid);
    if (!programManager.running->pid)
    {
        programManager.executeThread(second_thread, nullptr, "second thread",
1,programManager.running->pid);
        //programManager.executeThread(third_thread, nullptr, "third thread", 1);
    }
    asm_halt();
}
```

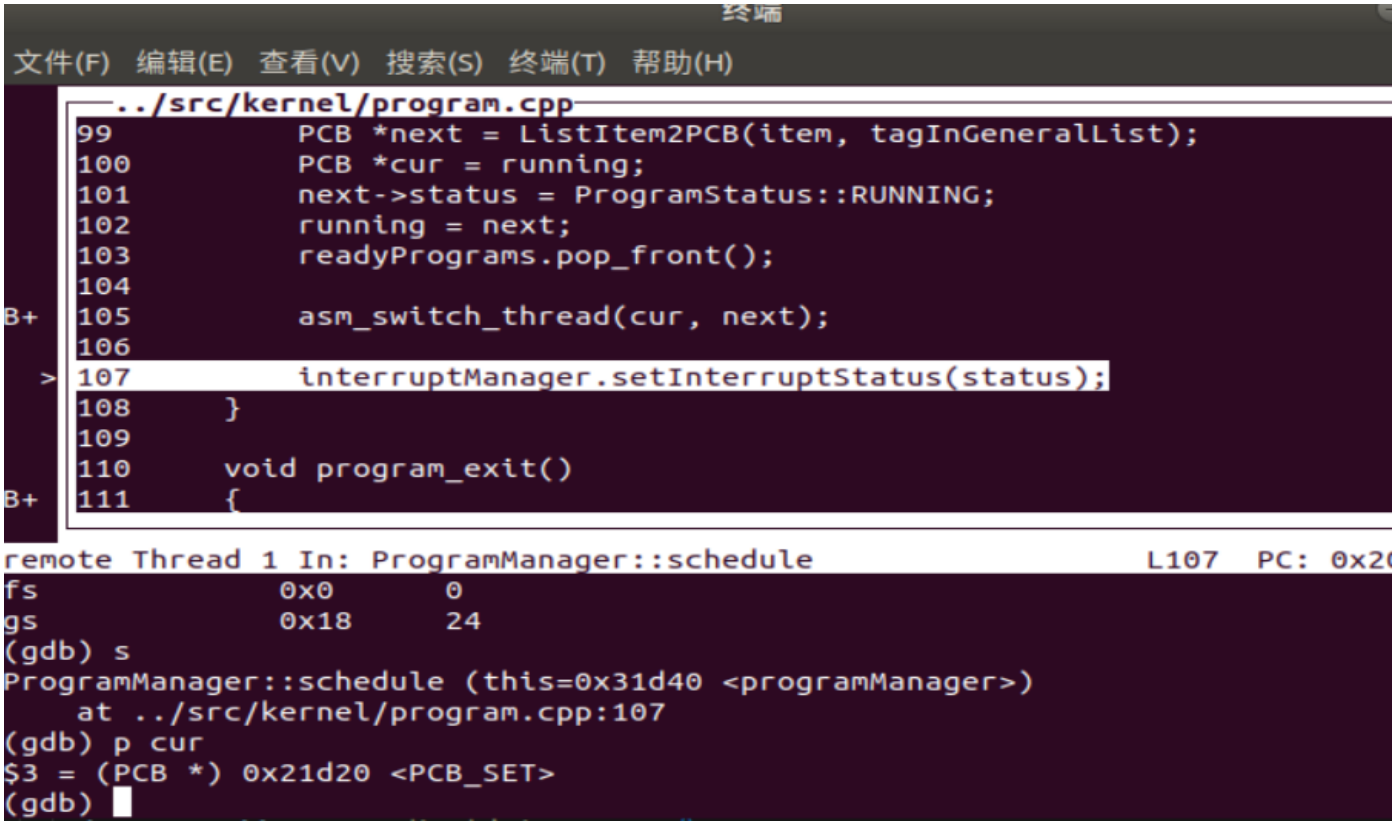
第一个线程函数很快便执行到asm_halt()，然后陷入死循环，当时间片用完之后，便会使用schedule（）函数调度到其他线程（即pid1）

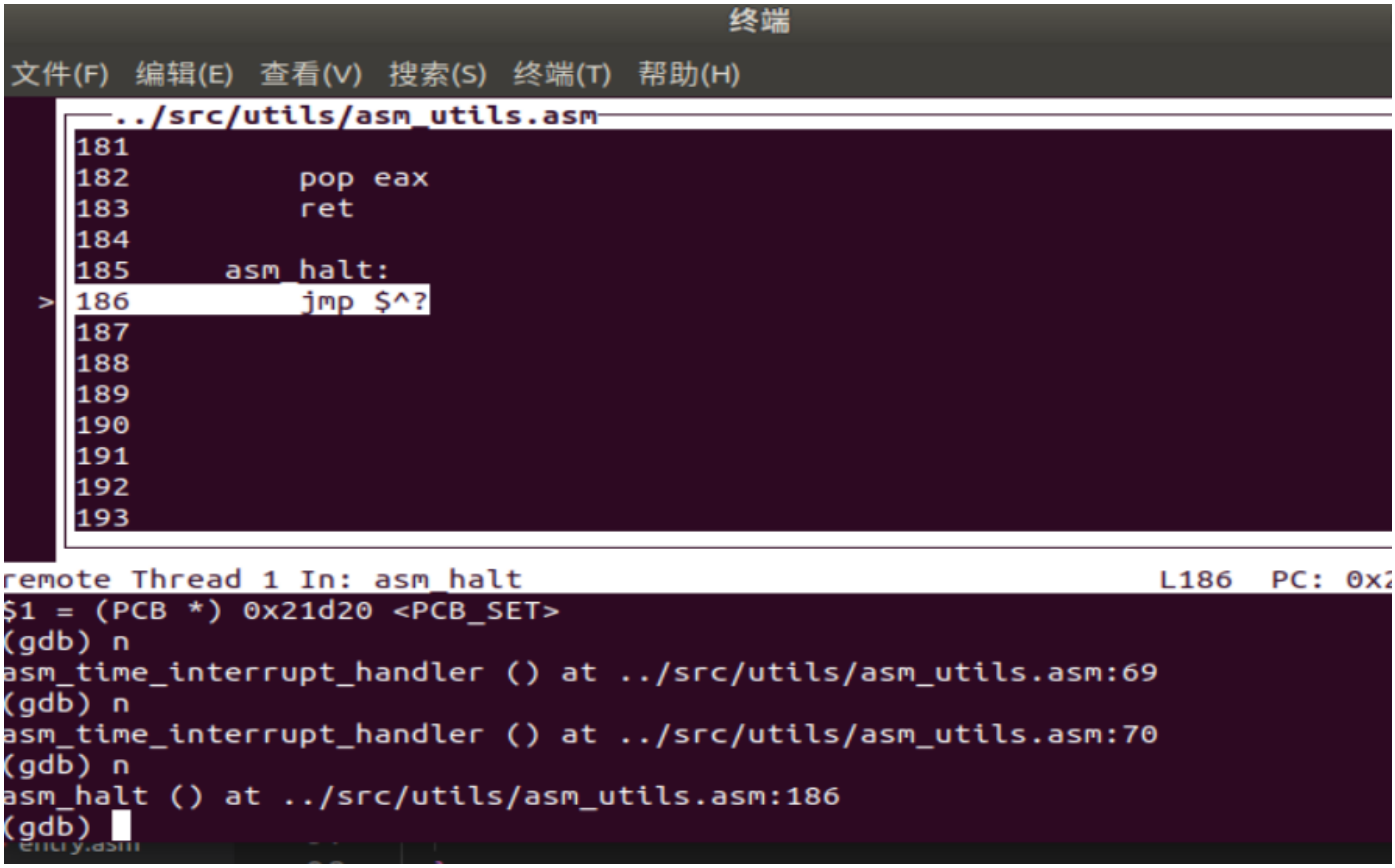
```
void second_thread(void *arg) {
    printf("pid %d name \"%s\" (fpid = %d): Hello World!\n",
programManager.running->pid, programManager.running->name,programManager.running-
>fpid);
    programManager.executeThread(third_thread, nullptr, "third thread",
1,programManager.running->pid);
}
```

第二个线程很快便执行结束（没有死循环），然后跳到program_exit（）函数，并调用schedule（）函数，因为之前调用第二个线程时将第一个线程加入就绪队列，所以会重新回到第一个线程的asm_halt()（执行ret后会返回调用asm_switch_thread的函数，也就是 ProgramManager::schedule，然后在ProgramManager::schedule中恢复中断状态，返回到时钟中断处理函数，最后从时钟中断中返回，恢复到线程被中断的地方继续执行。）


终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
../src/kernel/program.cpp
108 }
109
110 void program_exit()
111 {
112 PCB *thread = programManager.running;
113 thread->status = ProgramStatus::DEAD;
114
115 if (thread->pid)
116 {
> 117 programManager.schedule();
118 }
119 else
120 {
remote Thread 1 In: program_exit L117 PC: 0x203af
(gdb) p thread
$1 = (PCB *) 0x0
(gdb) n
(gdb) p thread
$2 = (PCB *) 0x22d20 <PCB_SET+4096>
(gdb) n
(gdb) n
(gdb)


终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
../src/kernel/program.cpp
100 PCB *cur = running;
101 next->status = ProgramStatus::RUNNING;
102 running = next;
103 readyPrograms.pop_front();
104
B+> 105 asm_switch_thread(cur, next);
106
107 interruptManager.setInterruptStatus(status);
108 }
109
110 void program_exit()
B+ 111 {
112 PCB *thread = programManager.running;
remote Thread 1 In: ProgramManager::schedule L105 PC: 0
Breakpoint 2, ProgramManager::schedule (this=0x31d40 <programManager>)
at ../src/kernel/program.cpp:105
(gdb) p cur
$1 = (PCB *) 0x22d20 <PCB_SET+4096>
(gdb) p next
$2 = (PCB *) 0x21d20 <PCB_SET>
(gdb)





Assignment 4 调度算法的实现

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。

实验目标：实现多级反馈队列调度算法（MFQ）

> MFQ 有以下特点： ①MFQ 维护不同优先级的队列，②对于不同的优先级的任务，高优先级的任务比低优先级的任务更先执行；③相同优先级的任务间采用 Round Robin 算法；⑤当一个任务时间片用完后该任务将会重新进入等待队列中，其优先级会降低，下一次执行的时间片会增加。

实验步骤（附代码）

1. 修改 program 类声明，将其中 ready 队列改为多队列：

```
public:
    List allPrograms;   // 所有状态的线程/进程的队列
    List readyPrograms[6]; // 处于ready(就绪态)的线程/进程的队列
    int readyPrograms_num;//当前ready队列的线程数量
    PCB *running;       // 当前执行的线程
```

2. 修改线程生成函数，将新生成的线程优先级缺省的声明为 1 并加入对 应优先级的队列

```
int ProgramManager::executeThread(ThreadFunction function, void *parameter, const
char *name, int priority)
{
    // 关中断，防止创建线程的过程被打断
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    // 分配一页作为PCB
    PCB *thread = allocatePCB();

    if (!thread)
        return -1;

    // 初始化分配的页
    memset(thread, 0, PCB_SIZE);

    for (int i = 0; i < MAX_PROGRAM_NAME && name[i]; ++i)
    {
        thread->name[i] = name[i];
    }

    thread->status = ProgramStatus::READY;
    thread->priority = priority;
    thread->ticks = priority * 10;
    thread->ticksPassedBy = 0;
    thread->pid = ((int)thread - (int)PCB_SET) / PCB_SIZE;

    thread->fpid = running ? running->pid: -1;
    // 线程栈
```

```
    thread->stack = (int *)((int)thread + PCB_SIZE);
    thread->stack -= 7;
    thread->stack[0] = 0;
    thread->stack[1] = 0;
    thread->stack[2] = 0;
    thread->stack[3] = 0;
    thread->stack[4] = (int)function;
    thread->stack[5] = (int)program_exit;
    thread->stack[6] = (int)parameter;

    allPrograms.push_back(&(thread->tagInAllList));
    readyPrograms[priority].push_back(&(thread->tagInGeneralList));
    readyPrograms_num++;

    // 恢复中断
    interruptManager.setInterruptStatus(status);

    return thread->pid;
}
```

3. 修改调度函数 schedule()

```
void ProgramManager::schedule()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

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

    if (running->status == ProgramStatus::RUNNING)
    {
        running->status = ProgramStatus::READY;
        //优先级降低（++）
        running->priority = running->priority >= 5 ? 5: running->priority + 1;
        running->ticks = running->priority * 10;
        readyPrograms[running->priority].push_back(&(running->tagInGeneralList));
    }
    else if (running->status == ProgramStatus::DEAD)
    {
        readyPrograms_num--;
        releasePCB(running);
    }

    //从优先级最高的的队列开始找到第一个ready进程
    ListItem *item = nullptr;
    for(int i = 1; i < 6; i++){
        if(readyPrograms[i].size() != 0){
            item = readyPrograms[i].front();
            readyPrograms[i].pop_front();
            break;
```

```
        }
    }

    PCB *next = ListItem2PCB(item, tagInGeneralList);
    PCB *cur = running;
    next->status = ProgramStatus::RUNNING;
    running = next;
    asm_switch_thread(cur, next);

    interruptManager.setInterruptStatus(status);
}
```

4. 设置线程运行函数

第一个线程执行的函数

```
void multithread(void *arg){
    //一开始先创建多个（10个）优先级不同的线程
    for(int i = 1; i < 6 ; i++){
        programManager.executeThread(second_thread, nullptr, "second thread", i%6);
        programManager.executeThread(third_thread, nullptr, "third thread", i%6);
    }
    while(1) {
            for (int i = 0; i < 100000000000; i++){
                if(i % 100000000 == 0){
                    printf("pid %d name \"%s\": Hello World!,fpid is :%d,priority
is %d,time:%d\n", programManager.running->pid, programManager.running-
>name,programManager.running->fpid,programManager.running-
>priority,programManager.running->ticksPassedBy);
                }
                //创建一个优先级高的线程
                if(i  == 200000000){
                    programManager.executeThread(high_thread, nullptr, "high
thread", 1);
                }
            }
    }
    asm_halt();
}
```

高优先级线程执行的函数

```
void high_thread(void *arg) {
    while(1) {
            for (int i = 0; i < 100000000000; i++)
                if(i % 100000000 == 0)
                    printf("pid %d name \"%s\": Hello World!,fpid is :%d,priority
is %d,time:%d\n", programManager.running->pid, programManager.running-
>name,programManager.running->fpid,programManager.running-
>priority,programManager.running->ticksPassedBy);
```

```
    }
}
```

```
void second_thread(void *arg) {
    while(1) {
            for (int i = 0; i < 100000000000; i++)
                if(i % 100000000 == 0)
                    printf("pid %d name \"%s\": Hello World!,fpid is :%d,priority
is %d,time:%d\n", programManager.running->pid, programManager.running-
>name,programManager.running->fpid,programManager.running-
>priority,programManager.running->ticksPassedBy);
    }
}
```

实验结果

> 上图展示的是同优先级的进程按照RR算法交替进行，执行后优先级降低，执行时间增加（time是当前进程的执行总时间）

> 上图展示的是当高优先级的进程进入时，会优先执行（pid11便是高优先级进程）

5. 总结(对实验过程中遇到的问题进行总结，可以提出对实验设置的改进意见)

此次实验的内容与理论课上的内容较为接近，大致内容理解起来比较快，但是在进程切换的汇编代码上思考了比较久，因为这涉及到esp栈指针在地址上的移动，这需要对每个地址存储的内容有明确的了解

6. 参考资料清单