



本科生实验报告

实验课程：____操作系统原理实验____

实验名称：____内核线程____

专业名称：____计算机科学与技术____

学生姓名：____熊彦钧____

学生学号：____23336266____

实验地点：____实验楼 B203____

实验成绩：____

报告时间：____2025 年 4 月 23 日____

Section 1 实验概述

- 实验任务 1: 学习可变参数机制, 然后实现 `printf`, 结果截图并说说是怎么做的。
- 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: 编写若干个线程函数, 使用 `gdb` 跟踪 `c_time_interrupt_handler`、`asm_switch_thread` 等函数, 观察线程切换前后栈、寄存器、PC 等变化, 结合 `gdb`、材料中“线程的调度”的内容来跟踪并说明下面两个过程: 一个新创建的线程是如何被调度然后开始执行的; 一个正在执行的线程是如何被中断然后被换下处理器的, 以及换上处理机后又是如何从被中断点开始执行的。
- 实验任务 4: 将线程调度算法修改为上面提到的算法或者是同学们自己设计的算法。然后, 同学们需要自行编写测试样例来呈现你的算法实现的正确性和基本逻辑。最后, 将结果截图并说说你是怎么做的。

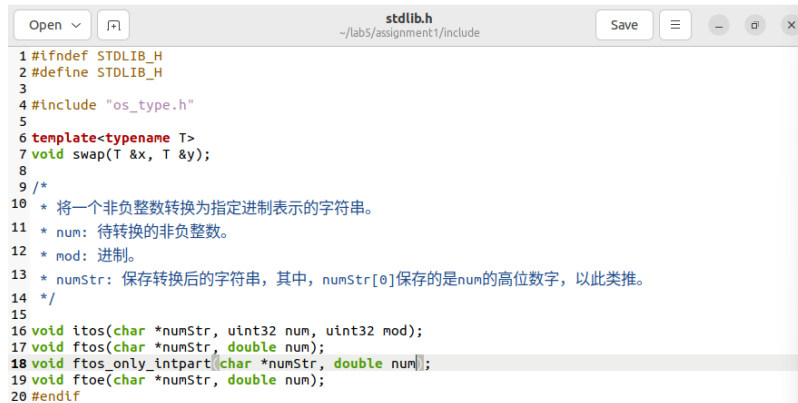
Section 2 实验步骤与实验结果

----- 实验任务 1 （对应 assignment1） -----

- 任务要求: 学习可变参数机制, 然后实现 `printf`, 你可以在材料中的 `printf` 上进行改进 (提示: 可以增加一些格式化输出类型, 比如 `%f`、`%.f`、`%e` 等), 或者从头开始实现自己的 `printf` 函数。
- 实验步骤: 首先我们把源代码复制到本地, 然后分析代码结构, 我们可以发现几个比较关键的文件: ①用于声明字符转换函数的 `stdlib.h`; ②用于实现打印函数的 `stdio.cpp`; ③用于实现字符转换函数的 `stdlib.cpp`; ④还有一个老生常谈的 `setup.cpp`。

随后, 针对要求, 我们对 `printf` 的功能进行扩展, 在我的程序中, 我主要扩展了 `%f`, `%.f`, `%e` 这三种格式化输出类型。

首先我们需要在 `stdlib.h` 声明三个类型分别需要用到的函数, 如下图所示:



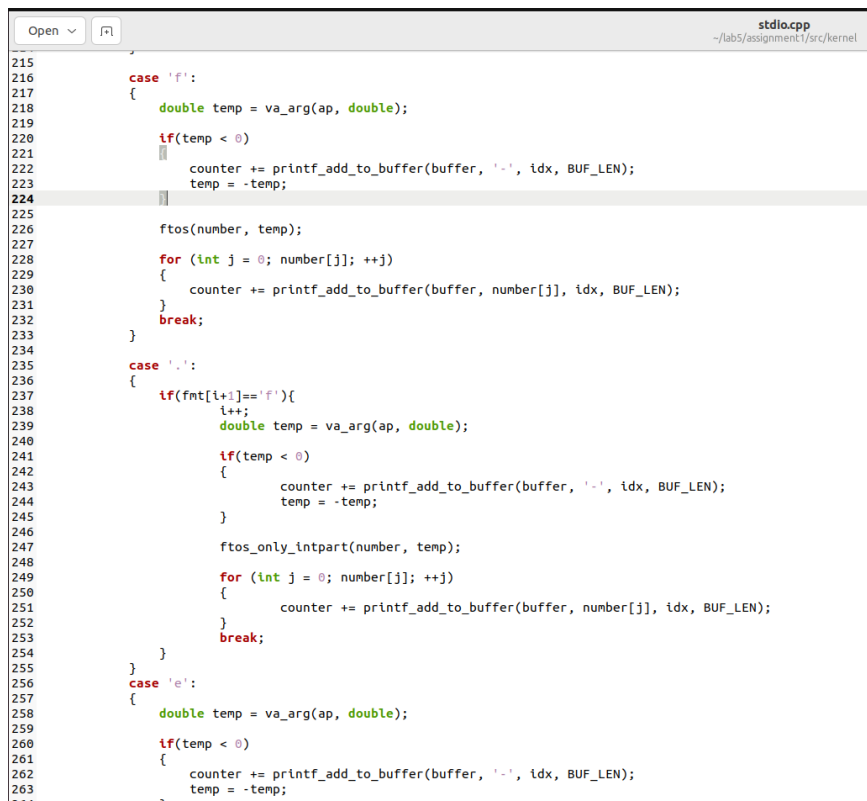
```

1 #ifndef STDLIB_H
2 #define STDLIB_H
3
4 #include "os_type.h"
5
6 template<typename T>
7 void swap(T &x, T &y);
8
9 /*
10 * 将一个非负整数转换为指定进制表示的字符串。
11 * num: 待转换的非负整数。
12 * mod: 进制。
13 * numStr: 保存转换后的字符串，其中，numStr[0]保存的是num的高位数字，以此类推。
14 */
15
16 void itos(char *numStr, uint32 num, uint32 mod);
17 void ftos(char *numStr, double num);
18 void ftos_only_intpart(char *numStr, double num);
19 void ftoe(char *numStr, double num);
20 #endif

```

声明的参数形式仿照了 `itos`，不过由于这三个参数格式类型都是以十进制来计算的，因此我们只需要传递两个参数就可以了。

随后，我们需要在 `stdio.cpp` 中加上这三个参数类型的判断和处理过程，思路也是仿照 `%d` 的处理方式，先声明一个 `double` 类型的 `temp` 函数，并赋值为宏定义函数 `va_arg(ap, double)`。随后判断 `temp` 函数的正负值，确保传入参数是正数。随后调用各自的转换函数，把浮点数转换为字符串形式。最后循环输出字符串中的每一个字符，这样我们就实现了参数类型的格式化输出。代码截图如下：



```

215
216     case 'f':
217     {
218         double temp = va_arg(ap, double);
219
220         if(temp < 0)
221         {
222             counter += printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
223             temp = -temp;
224         }
225
226         ftos(number, temp);
227
228         for (int j = 0; number[j]; ++j)
229         {
230             counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
231         }
232         break;
233     }
234
235     case 'e':
236     {
237         if(fmt[i+1]!='f'){
238             i++;
239             double temp = va_arg(ap, double);
240
241             if(temp < 0)
242             {
243                 counter += printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
244                 temp = -temp;
245             }
246
247             ftos_only_intpart(number, temp);
248
249             for (int j = 0; number[j]; ++j)
250             {
251                 counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
252             }
253             break;
254         }
255     }
256
257     case 'e':
258     {
259         double temp = va_arg(ap, double);
260
261         if(temp < 0)
262         {
263             counter += printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
264             temp = -temp;
265         }

```

由于 `case 'e'` 的代码格式和 `case 'f'` 的完全一样，这里就不截全了。

接着，就是我们这次编写代码的主要部分：实现将每个参数类型的浮点数转换为字符串。

(1) %f 将浮点数转换为字符串：代码如下：

```
void ftos(char *numStr, double num){
    int intPart = (int)num;
    char intStr[32];
    itos(intStr, intPart, 10);
    int i;
    for (i = 0; intStr[i]; i++) {
        numStr[i] = intStr[i];
    }
    numStr[i++] = '.';
    double fracPart = num - intPart;
    int precision = 6;
    for (int j = 0; j < precision; j++) {
        fracPart *= 10;
        int digit = (int)fracPart;
        numStr[i++] = digit + '0';
        fracPart -= digit;
    }
    numStr[i] = '\\0';
}
```

我们的思路是，先处理整数部分，再处理小数部分，于是我们先讲浮点数强制类型转换为整数，随后我们调用已有的函数 `itos`，将整数部分转换为字符串。随后对于小数部分，我们的思路和整数部分类似，不过，这次我们需要每次将 `fracPart` 乘 10，每次“提取”出一位小数。

还有一点需要注意的是起初我的条件判定是 `while(fracPart>0)`，结果发现出现了错误，思考后本人觉得是因为 `double` 的精度较高，因此导致输出了类似乱码的结果。因此，最后选择仅保留六位小数（这也是现在各个语言语法常见的处理）。

(2) %.f 将浮点数转换为四舍五入的整数：代码如下：

```
void ftos_only_intpart(char *numStr, double num){
    int intPart=(int)num;
    double fracPart = num - intPart;
    fracPart *= 10;
    int digit = (int)fracPart;
    if(digit>=5){
        intPart++;
    }
    char intStr[32];
    itos(intStr,intPart,10);
    int i;
    for (i = 0; intStr[i]; i++) {
        numStr[i] = intStr[i];
    }
    numStr[i] = '\\0';
}
```

我们的思路是，先提取出整数，然后判断小数点后的第一位数，如果比 4 大，就给整数+1，否则不变，最后再把整数转码。

(3) %e 将浮点数转换为科学记数法：代码如下：

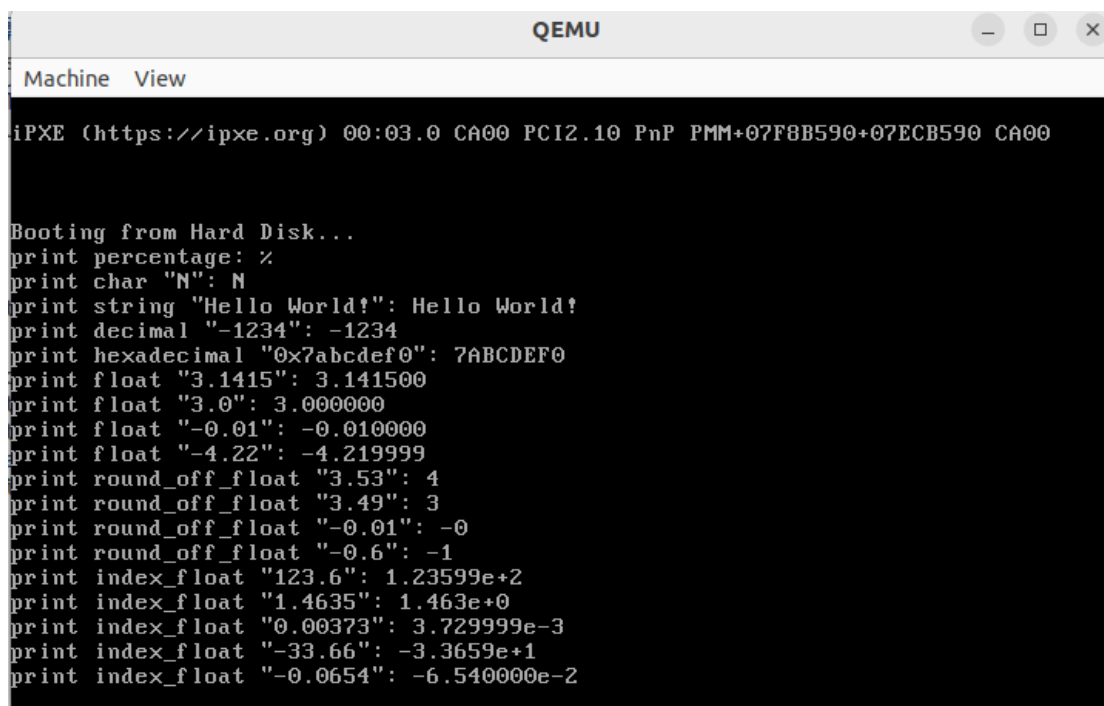
```
void ftoe(char *numStr, double num){
    if (num == 0) {
        numStr[0] = '0';
        numStr[1] = '\0';
        return;
    }
    if(num<1){
        int count=0;
        while(num<1){
            num*=10;
            count++;
        }
        int intPart=(int)num;
        int length=0;
        numStr[length]=intPart+'0';
        length++;
        numStr[length]='.';
        length++;
        double fracPart = num - intPart;
        int percision=6;
        while(percision>0){
            fracPart*=10;
            int digit=(int)fracPart;
            numStr[length]=digit+'0';
            length++;
            fracPart-=digit;
            percision--;
        }
        numStr[length]='e';
        length++;
        numStr[length]='-';
        length++;
        char countStr[32];
        itos(countStr,count,10);
        for(int j=0;countStr[j];j++){
            numStr[length]=countStr[j];
            length++;
        }
        numStr[length]='\0';
    }
    else{
        int intPart=(int)num;
        char intStr[32];
        itos(intStr,intPart,10);
        int i;
        int length=0;
        for(i=0;intStr[i];i++){
            numStr[length]=intStr[i];
            length++;
            if(i==0){
                numStr[length]='.';
                length++;
            }
        }
        double fracPart = num - intPart;
        int percision=3;
        while(percision>0){
            fracPart*=10;
            int digit=(int)fracPart;
            numStr[length]=digit+'0';
            length++;
            fracPart-=digit;
            percision--;
        }
        numStr[length]='e';
        length++;
        numStr[length]='+';
        length++;
        numStr[length]=i-1+'0';
        length++;
        numStr[length]='\0';
    }
}
```

我们的思路是，分别处理 num 小于 1 和大于 1 的情况，因为这两个情况他们的指数一个是正一个是负。其实转码的过程和浮点数比较类似，唯一的区别就是需要用个变量统计小数点移动的位数。和浮点数相同，我们也需要限定精度，否则也会输出“乱码”。

最后，我们需要修改 setup.cpp 中 printf 的内容，并进行测试：

```
19 //asm_enable_interrupt();
20 printf("print percentage: %%\n"
21 "print char \"N\": %c\n"
22 "print string \"Hello World!\": %s\n"
23 "print decimal \"-1234\": %d\n"
24 "print hexadecimal \"0x7abcdef0\": %x\n"
25 "print float \"3.1415\": %f\n"
26 "print float \"3.0\": %f\n"
27 "print float \"-0.01\": %f\n"
28 "print float \"-4.22\": %f\n"
29 "print round_off_float \"3.53\": %.f\n"
30 "print round_off_float \"3.49\": %.f\n"
31 "print round_off_float \"-0.01\": %.f\n"
32 "print round_off_float \"-0.6\": %.f\n"
33 "print index_float \"123.6\": %e\n"
34 "print index_float \"1.4635\": %e\n"
35 "print index_float \"0.00373\": %e\n"
36 "print index_float \"-33.66\": %e\n"
37 "print index_float \"-0.0654\": %e\n",
38 'N', "Hello World!", -1234, 0x7abcdef0, 3.1415, 3.0, -0.01, -4.22, 3.53, 3.49, -0.01, -0.6, 123.6, 1.4635, 0.00373, -33.66, -0.0654);
39 //uint a = 1 / 0;
40 asm_halt();
41 }
```

测试结果如下：



```
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 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
print float "3.1415": 3.141500
print float "3.0": 3.000000
print float "-0.01": -0.010000
print float "-4.22": -4.219999
print round_off_float "3.53": 4
print round_off_float "3.49": 3
print round_off_float "-0.01": -0
print round_off_float "-0.6": -1
print index_float "123.6": 1.23599e+2
print index_float "1.4635": 1.463e+0
print index_float "0.00373": 3.729999e-3
print index_float "-33.66": -3.3659e+1
print index_float "-0.0654": -6.540000e-2
```

我们可以发现，程序有点小瑕疵，比如：在浮点数和四舍五入数的输入时，我们发现输入一个整数（比如 3）会出错，而输入 3.0 则不会；此外，在科学记数法的输出，因为我们是通过设置 `precision` 强制截断，因此输出的数存在长度不一的情况，并且因为浮点数存储的问题，我们会发现：例如 33.66，在计算机内部存储为了 33.65999999……，所以我们的输出就变成了 3.3659e+1。

至此，这一任务也算是完成了吧.....

----- 实验任务 2 （对应 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。
- 实验步骤：这一部分的代码较多，多了很多关于线程的文件，比如声明 PCB 的 `thread.h`，实现线程创建、调度、退出等函数的 `program.h` 和 `program.cpp`，实现队列的 `list.h`，等等。经过考虑，发现在 PCB 中添加父子进程关系的维护（只维护父子关系，不处理内存共享和资源复制等操作）和进程创建时间的显示比较现实。

为了实现这一目的，首先我们需要在 `thread.h` 的 PCB 中添加相应的参数，截图如下：

```

struct PCB
{
    int *stack;                // 栈指针，用于调度时保存esp
    char name[MAX_PROGRAM_NAME + 1]; // 线程名
    enum ProgramStatus status;    // 线程的状态
    int priority;                // 线程优先级
    int pid;                    // 线程pid
    int 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
    int parent_pid;             // 父进程 pid
    List childrenList;          // 子进程链表（存储子进程的 PCB 指针）
    ListItem tagInChildrenList; // 子进程队列标识
    int createTime;            // 进程创建时间
};

#endif

```

如图，我们额外声明了四个变量，`parent_pid` 用于存储父进程 `pid`，`childrenList` 用于链式存储子进程，`tagInChildrenList` 用于标识子进程队列，而 `createTime` 用于存储进程的创建时间。

除此之外，我们还要在 `program.h` 中声明一个函数 `PCB *find_thread_ByPid(int pid);`，用于在进程退出时在父进程中找到自己以便于在父进程的子进程队列中删除自己。

```

class ProgramManager
{
public:
    List allPrograms; // 所有状态的线程/进程的队列
    List readyPrograms; // 处于ready(就绪态)的线程/进程的队列
    PCB *running;      // 当前执行的线程
public:
    ProgramManager();
    void initialize();

    // 创建一个线程并放入就绪队列

    // function: 线程执行的函数
    // parameter: 指向函数的参数的指针
    // name: 线程的名称
    // priority: 线程的优先级

    // 成功，返回pid；失败，返回-1
    int executeThread(ThreadFunction function, void *parameter, const char *name, int priority);

    // 分配一个PCB
    PCB *allocatePCB();
    // 归还一个PCB
    // program: 待释放的PCB
    void releasePCB(PCB *program);

    // 执行线程调度
    void schedule();

    PCB *find_thread_ByPid(int pid);
};

```

这一函数的定义则放在了 `program.cpp` 中，如下：我们直接在 `allPrograms` 队列中检索。

```

PCB *ProgramManager::findProgramByPid(int pid)
{
    ListItem *item = allPrograms.front();
    while (item != nullptr)
    {
        PCB *pcb = ListItem2PCB(item, tagInAllList);
        if (pcb->pid == pid)
        {
            return pcb;
        }
        item = item->next;
    }
    return nullptr;
}

```

此外，我们还需要一个前置准备，由于我们需要存储创建时间 `createTime`，必然需要赋值为时钟中断计数器。因此，我们需要在 `interrupt.cpp` 的中断处理函数中加入代码（`++times`），以更新全局时间计数器：

```

// 中断处理函数
extern "C" void c_time_interrupt_handler()
{
    PCB *cur = programManager.running;

    // 更新全局时间计数器
    ++times;

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

```

至此，我们的前置工作已经完成了，接下来，我们需要修改线程创建函数和线程退出函数：

对线程创建函数的修改如下：

```

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

// 设置父子进程关系
PCB *current = running;
thread->parent_pid = current ? current->pid : -1;
thread->childrenList.initialize();
thread->tagInChildrenList.next = nullptr;
thread->tagInChildrenList.previous = nullptr;

if (current) {
    current->childrenList.push_back(&(thread->tagInChildrenList));
}

// 设置创建时间
extern int times;
thread->createTime = times;

// 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;

```


首先我们用指针*current 获取当前运行中的进程（因为想要创建子进程，只能说运行中的进程主动申请系统调用），如果有运行中的进程，则说明创建的进程是一个子进程，需要为 parent_pid 赋值，否则赋值为-1，代表不是子进程。随后初始化新建进程的子进程队列。并判断是否需要加入父进程的子进程队列中。最后我们引入全局变量 times，并赋值给 createTime。这样，进程创建函数就修改好了。

接着是对进程退出函数的修改：

```
void program_exit()
{
    // 关中断
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    PCB *thread = programManager.running;
    thread->status = ProgramStatus::DEAD;

    // 处理父子进程关系
    if (thread->parent_pid != -1) {
        // 如果有父进程，从父进程的子进程列表中移除自己
        PCB *parent = programManager.find_thread_ByPid(thread->parent_pid);
        if (parent) {
            parent->childrenList.erase(&(thread->tagInChildrenList));
        }
    }

    // 处理子进程
    ListItem *item = thread->childrenList.front();
    while (item) {
        // 找到子进程的PCB
        PCB *child = ListItem2PCB(item, tagInChildrenList);
        // 将子进程的父进程设为-1（成为孤儿进程）
        child->parent_pid = -1;

        item = item->next;
    }

    if (thread->pid)
    {
        programManager.schedule();
    }
    else
    {
        interruptManager.disableInterrupt();
        printf("halt\n");
        asm_halt();
    }
}
```

首先，如果退出的进程有父进程，我们需要调用函数找父进程，并在父进程的子进程队列中移除自己。对于退出进程的子进程，我们需要遍历退出进程的子进程队列，并将子进程一一设置为孤儿进程。至此，进程退出的函数就修改好了。

最后，我们到了最后一步，在 setup.cpp 中编写线程函数，我们着重测试的方向如下：①能否（嵌套）创建子进程②能否在一个进程创建完子进程后，创建一个新的进程。依据这一目标，我们编写的线程函数如下：

```

void first_thread(void *arg)
{
    // 第1个线程不可以返回
    printf("pid %d name \"%s\": I am the first_thread! ... ",
           programManager.running->pid, programManager.running->name);
    printf("createtime: %d\n", programManager.running->createTime);

    programManager.executeThread(child_thread, nullptr, "first child", 1);
    programManager.executeThread(child_thread, nullptr, "second child", 1);
    programManager.executeThread(child_thread, nullptr, "third child", 1);

    printf("now show the childrenList of first_thread:");
    ListItem *item = programManager.running->childrenList.front();
    while (item) {
        PCB *child = ListItem2PCB(item, tagInChildrenList);
        printf("%d ", child->pid);
        item = item->next;
    }
    printf("\n");

    if (!programManager.running->pid)
    {
        programManager.executeThread(second_thread, nullptr, "second thread", 1);
        programManager.executeThread(third_thread, nullptr, "third thread", 1);
    }
    while(1){};
}

```

这是第一个线程，我们以这个线程为父进程，创造了三个子进程，并在程序中另外创造了两个进程 second thread 和 third thread。

```

void third_thread(void *arg) {
    printf("pid %d name \"%s\": I am the third thread! ... ",
           programManager.running->pid, programManager.running->name);
    printf("createtime: %d\n", programManager.running->createTime);
    while(1){};
}

void second_thread(void *arg) {
    printf("pid %d name \"%s\": I am the second thread! ... ",
           programManager.running->pid, programManager.running->name);
    printf("createtime: %d\n", programManager.running->createTime);

    programManager.executeThread(child_thread, nullptr, "first child", 1);
    programManager.executeThread(child_thread, nullptr, "second child", 1);

    printf("now show the childrenList of second_thread:");
    ListItem *item = programManager.running->childrenList.front();
    while (item) {
        PCB *child = ListItem2PCB(item, tagInChildrenList);
        printf("%d ", child->pid);
        item = item->next;
    }
    printf("\n");
    while(1){};
}

```

这是第二个线程和第三个线程，我们可以看到，我们又以第二个线程为父进程创造了两个子进程，而 third_thread 则没有新建进程。此时，我们还没有实现创建嵌套子进程的目的，因为 second thread 与 child 处于同等地位。

```

void grand_child_thread(void *arg){
    printf("pid %d name \"%s\": I am the grand child thread of %d! ... ",
        programManager.running->pid, programManager.running->name, programManager.running-
>parent_pid);
    printf("createtime: %d\n", programManager.running->createTime);
    while(1){};
}

void child_thread(void *arg){
    printf("pid %d name \"%s\": I am the child thread of %d! ... ",
        programManager.running->pid, programManager.running->name, programManager.running-
>parent_pid);
    printf("createtime: %d\n", programManager.running->createTime);

    programManager.executeThread(grand_child_thread, nullptr, "grand child", 1);

    printf("now show the childrenList of the child of thread %d:", programManager.running-
>pid);
    ListItem *item = programManager.running->childrenList.front();
    while (item) {
        PCB *child = ListItem2PCB(item, tagInChildrenList);
        printf("%d ", child->pid);
        item = item->next;
    }
    printf("\n");
    while(1){};
}

```

这是子进程和孙子进程的实现代码（孙子进程是子进程的子进程）。至此，我们就实现了创建嵌套子进程的目标，并且我们可以通过打印信息来观察每个进程的子进程队列。

接下来，我们运行程序观察输出结果：

```

Machine View
Booting from Hard Disk...
pid 0 name "first thread": I am the first_thread! ... createtime: 0
now show the childrenList of first_thread:1 2 3
pid 1 name "first child": I am the child thread of 0! ... createtime: 1
now show the childrenList of the child of thread 1:6
pid 2 name "second child": I am the child thread of 0! ... createtime: 1
now show the childrenList of the child of thread 2:7
pid 3 name "third child": I am the child thread of 0! ... createtime: 1
now show the childrenList of the child of thread 3:8
pid 4 name "second thread": I am the second thread! ... createtime: 1
now show the childrenList of second_thread:9 10
pid 5 name "third thread": I am the third thread! ... createtime: 1
pid 6 name "grand child": I am the grand child thread of 1! ... createtime: 11
pid 7 name "grand child": I am the grand child thread of 2! ... createtime: 22
pid 8 name "grand child": I am the grand child thread of 3! ... createtime: 33
pid 9 name "first child": I am the child thread of 4! ... createtime: 44
now show the childrenList of the child of thread 9:11
pid 10 name "second child": I am the child thread of 4! ... createtime: 44
now show the childrenList of the child of thread 10:12
pid 11 name "grand child": I am the grand child thread of 9! ... createtime: 143
pid 12 name "grand child": I am the grand child thread of 10! ... createtime: 154

```

我们可以看到，第一个进程的创建时间为 0，它有三个子进程：1 2 3；由于 first thread 的三个子进程和 second thread、third thread 都是在 first thread 中创建

的，所以他们的创建时间相等（都为 1），并且我们可以在他们的打印信息中看到他们的子进程队列。

接下来我们可以看到孙子进程的创建，以及他们的父进程是哪个。由于，孙子进程是三个子进程分别创建的，因此创建时间各不相同。

随后，我们可以看到 `second thread` 创建的子进程和孙子进程以及他们的创建时间。

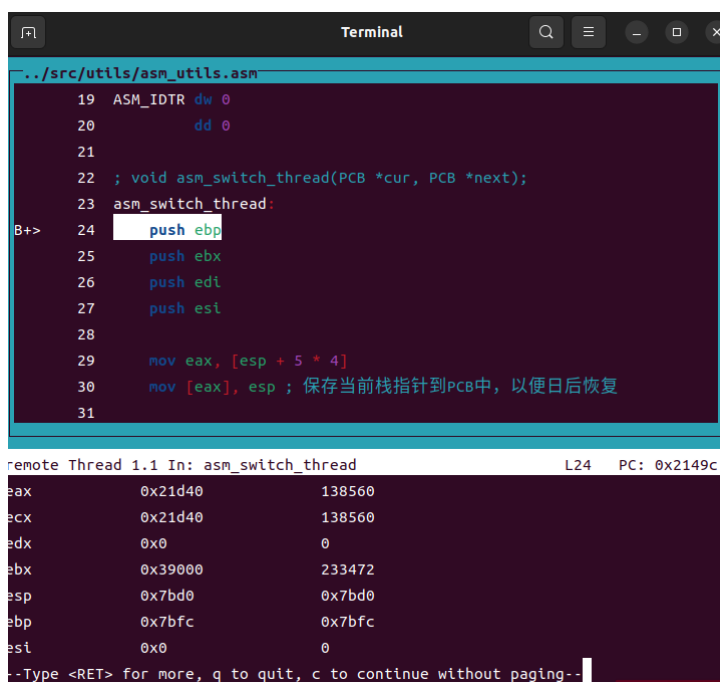
经过检验，父子进程的关系无误，实验任务 2 完成。

----- 实验任务 3（对应 assignment3） -----

● 任务要求：使用 `gdb` 跟踪 `c_time_interrupt_handler`、`asm_switch_thread` 等函数，观察线程切换前后栈、寄存器、PC 等变化，结合 `gdb`、材料中“线程的调度”的内容来跟踪并说明下面两个过程：一个新创建的线程是如何被调度然后开始执行的；一个正在执行的线程是如何被中断然后被换下处理器的，以及换上处理机后又是如何从被中断点开始执行的。

● 实验步骤：①启动 `gdb`：在这个实验任务重，为了方便，我没有沿用 `assignment2` 中修改后的代码，而是直接使用课程网站上的 `src` 中的代码。由于命令行操作已经整合到了 `makefile` 中，因此，我们只需要输入 `make debug`，就会自动远程连接。

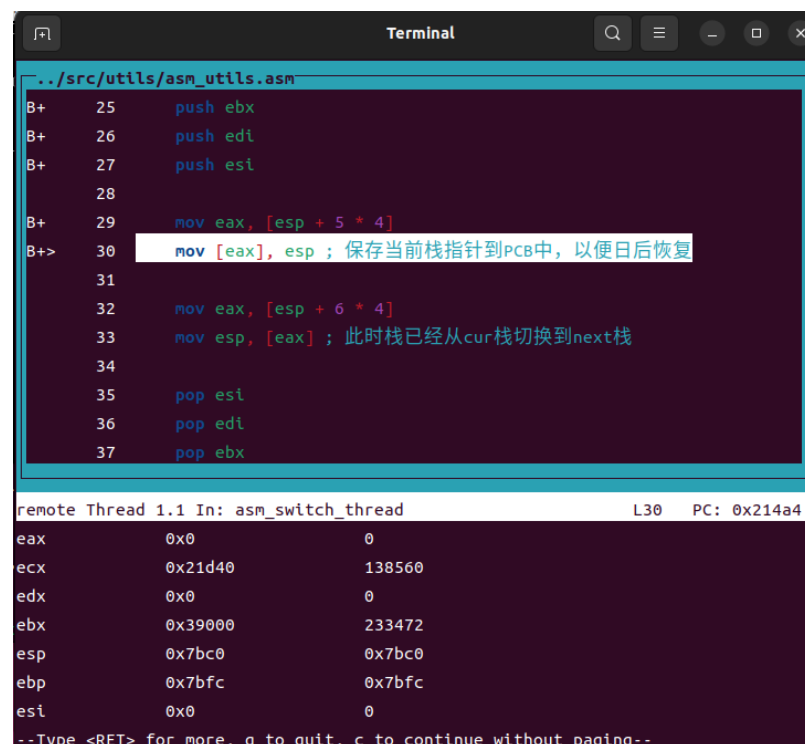
②观察函数 `asm_switch_thread`：首先我们查看进程转换前的寄存器值：



```
Terminal
../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:
B+> 24  push ebp
25  push ebx
26  push edi
27  push esi
28
29  mov eax, [esp + 5 * 4]
30  mov [eax], esp ; 保存当前栈指针到pcb中，以便日后恢复
31

remote Thread 1.1 In: asm_switch_thread L24 PC: 0x2149c
eax      0x21d40      138560
ecx      0x21d40      138560
edx      0x0          0
ebx      0x39000      233472
esp      0x7bd0       0x7bd0
ebp      0x7bfc       0x7bfc
esi      0x0          0
--Type <RET> for more, q to quit, c to continue without paging--
```

我们对 24-30 中的每一行设置断点，并观察寄存器值发现，在第三十行设置断点后，寄存器的值变为



The screenshot shows a debugger window with the following assembly code and register values:

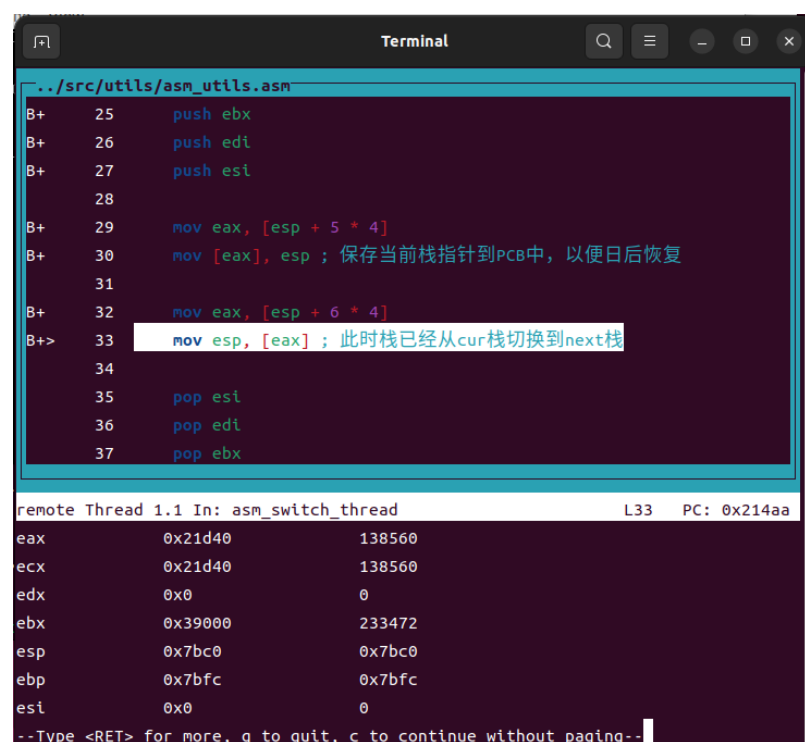
```
./src/utls/asm_utils.asm
B+ 25 push ebx
B+ 26 push edi
B+ 27 push esi
28
B+ 29 mov eax, [esp + 5 * 4]
B+> 30 mov [eax], esp ; 保存当前栈指针到PCB中，以便日后恢复
31
32 mov eax, [esp + 6 * 4]
33 mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
34
35 pop esi
36 pop edi
37 pop ebx
```

remote Thread 1.1 In: asm_switch_thread L30 PC: 0x214a4

Register	Value
eax	0x0
ecx	0x21d40
edx	0x0
ebx	0x39000
esp	0x7bc0
ebp	0x7bfc
esi	0x0

--Type <RET> for more, q to quit, c to continue without paging--

我们可以看到 eax 的值变为了 0。而在运行到第三十三行的时候，eax 又会重新变为 138560:



The screenshot shows a debugger window with the following assembly code and register values:

```
./src/utls/asm_utils.asm
B+ 25 push ebx
B+ 26 push edi
B+ 27 push esi
28
B+ 29 mov eax, [esp + 5 * 4]
B+ 30 mov [eax], esp ; 保存当前栈指针到PCB中，以便日后恢复
31
B+ 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
```

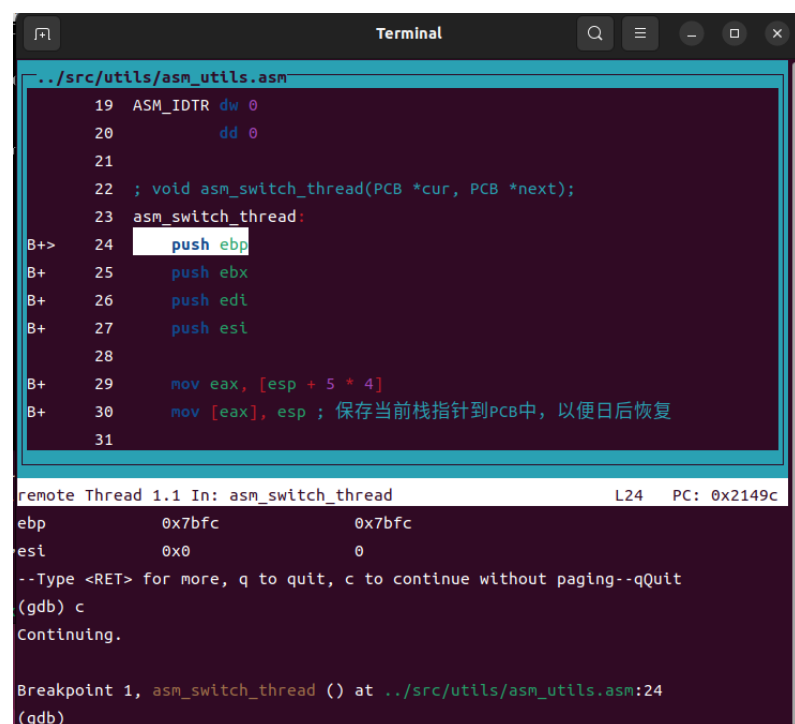
remote Thread 1.1 In: asm_switch_thread L33 PC: 0x214aa

Register	Value
eax	0x21d40
ecx	0x21d40
edx	0x0
ebx	0x39000
esp	0x7bc0
ebp	0x7bfc
esi	0x0

--Type <RET> for more, q to quit, c to continue without paging--

在此时，如果不设置断点，直接运行，我们会发现程序重新回到了

asm_switch_thread 的第一行:

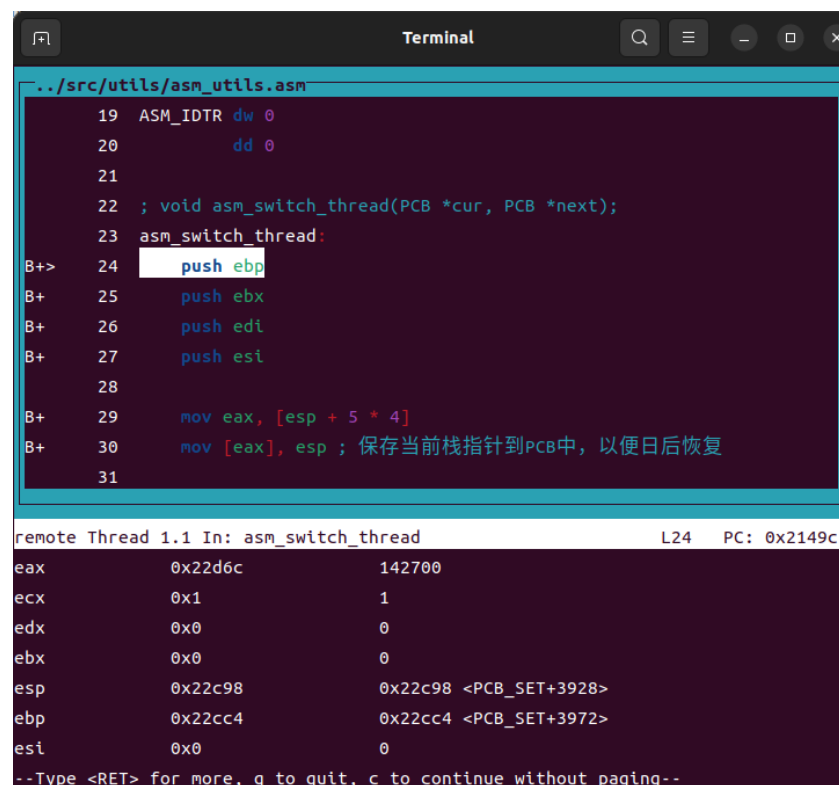


```
Terminal
../src/utils/asm_utils.asm
19 ASM_IDTR dw 0
20     dd 0
21
22 ; void asm_switch_thread(PCB *cur, PCB *next);
23 asm_switch_thread:
B+> 24     push ebp
B+ 25     push ebx
B+ 26     push edi
B+ 27     push esi
28
B+ 29     mov eax, [esp + 5 * 4]
B+ 30     mov [eax], esp ; 保存当前栈指针到PCB中，以便日后恢复
31

remote Thread 1.1 In: asm_switch_thread L24 PC: 0x2149c
ebp      0x7bfc      0x7bfc
esi      0x0        0
--Type <RET> for more, q to quit, c to continue without paging--qQuit
(gdb) c
Continuing.

Breakpoint 1, asm_switch_thread () at ../src/utils/asm_utils.asm:24
(gdb)
```

此时，寄存器的值发生了比较大的变化:



```
Terminal
../src/utils/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:
B+> 24     push ebp
B+ 25     push ebx
B+ 26     push edi
B+ 27     push esi
28
B+ 29     mov eax, [esp + 5 * 4]
B+ 30     mov [eax], esp ; 保存当前栈指针到PCB中，以便日后恢复
31

remote Thread 1.1 In: asm_switch_thread L24 PC: 0x2149c
eax      0x22d6c      142700
ecx      0x1          1
edx      0x0          0
ebx      0x0          0
esp      0x22c98      0x22c98 <PCB_SET+3928>
ebp      0x22cc4      0x22cc4 <PCB_SET+3972>
esi      0x0          0
--Type <RET> for more, q to quit, c to continue without paging--
```

反复循环运行后，我们发现：ecx 将保持 1 不变。而剩下的几个寄存器将会循环变化为几个数值。

③观察 c_time_interrupt_handler: 跳转后，我们先观察此时各个寄存器的初始

值:

```
./src/kernel/interrupt.cpp
85 }
86
87 // 中断处理函数
88 extern "C" void c_time_interrupt_handler()
89 {
90     PCB *cur = programManager.running;
91
92     // 更新全局时间计数器
93     ++times;
94
95     if (cur->ticks)
96     {
97         --cur->ticks;
98     }
99 }
100
remote Thread 1.1 In: c_time_interrupt_handler L90 PC: 0x2024a
eax      0x20      32
ecx      0x24cb2   150706
edx      0x10      16
ebx      0x0       0
esp      0x24ce0   0x24ce0 <PCB_SET+12192>
ebp      0x24cf8   0x24cf8 <PCB_SET+12216>
esi      0x0       0
--Type <RET> for more, q to quit, c to continue without paging--
```

通过观察我们发现，每次在执行 `cur->ticks` 前后，`edx` 寄存器的值都会减 1，说明时钟中断程序执行正确。

```
./src/kernel/interrupt.cpp
88 extern "C" void c_time_interrupt_handler()
89 {
90     PCB *cur = programManager.running;
91
92     // 更新全局时间计数器
93     ++times;
94
95     if (cur->ticks)
96     {
97         --cur->ticks;
98         ++cur->ticksPassedBy;
99     }
100     else
101     {
102         // ...
103     }
104 }
105
remote Thread 1.1 In: c_time_interrupt_handler L95 PC: 0x2025d
eax      0x39      57
ecx      0x24cb2   150706
edx      0x9       9
ebx      0x0       0
esp      0x24ce0   0x24ce0 <PCB_SET+12192>
ebp      0x24cf8   0x24cf8 <PCB_SET+12216>
esi      0x0       0
--Type <RET> for more, q to quit, c to continue without paging--

./src/kernel/interrupt.cpp
88 extern "C" void c_time_interrupt_handler()
89 {
90     PCB *cur = programManager.running;
91
92     // 更新全局时间计数器
93     ++times;
94
95     if (cur->ticks)
96     {
97         --cur->ticks;
98         ++cur->ticksPassedBy;
99     }
100     else
101     {
102         // ...
103     }
104 }
105
remote Thread 1.1 In: c_time_interrupt_handler L98 PC: 0x20276
eax      0x23d40   146752
ecx      0x24cb2   150706
edx      0x8       8
ebx      0x0       0
esp      0x24ce0   0x24ce0 <PCB_SET+12192>
ebp      0x24cf8   0x24cf8 <PCB_SET+12216>
esi      0x0       0
--Type <RET> for more, q to quit, c to continue without paging--
```

经过联系课内知识，加上上网搜索，我们可以回答上面提到的两个问题了：

(1) 一个新创建的线程是如何被调度然后开始执行的：当一个新线程被创建时，系统会为其分配 PCB 结构并初始化线程栈，栈中预先设置好线程入口函数地址和初始寄存器状态。该线程随后被加入到就绪队列中等待调度。当发生时钟中断或当前线程主动让出 CPU 时，调度器会从就绪队列中选择这个新线程，通过调用 `asm_switch_thread` 函数完成上下文切换。在这个汇编函数中，系统首先

保存当前线程的栈指针到其 PCB 中，然后从新线程的 PCB 加载其栈指针，并通过一系列 pop 指令恢复寄存器状态。最后，当 asm_switch_thread 执行 ret 指令时，CPU 会跳转到新线程的入口函数开始执行，新线程由此获得 CPU 控制权并开始运行。

(2) 正在执行的线程会在时钟中断发生时被强制中断，CPU 自动将当前指令指针 EIP 和标志寄存器 EFLAGS 压入线程栈，然后跳转到中断处理程序 asm_time_interrupt_handler。该处理程序首先用 pushad 保存所有通用寄存器，接着调用 c_time_interrupt_handler 进行时间片检查。若时间片耗尽，就会触发调度器进行线程切换。在 asm_switch_thread 函数中，系统保存当前线程的完整上下文到其 PCB，并加载被选中线程的上下文。当切换回先前被中断的线程时，系统通过 popad 恢复所有寄存器，最后通过 iret 指令使 CPU 从中断点继续执行，就像从未被中断过一样，从而实现了线程的无缝切换和精确恢复。

至此，实验任务 3 完成。

----- 实验任务 4 （对应 assignment4） -----

- 任务要求： 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。
- 实验步骤： ①前置准备：在修改调度算法和编写测试样例之前，我们需要做一些在我的调度算法里面会用到的准备：

由于源代码中每个线程 ticks 的初始值定义为 priority*10，这个定义只适用于基于优先级的 RR 算法。我想要自己手动设置每个进程的时间片，以更真实地模拟应用场景，因此，我在 program.h 和 program.cpp 中声明并定义了函数 setThreadTicks：


```

15 public:
16     ProgramManager();
17     void initialize();
18
19     // 创建一个线程并放入就绪队列
20
21     // function: 线程执行的函数
22     // parameter: 指向函数的参数的指针
23     // name: 线程的名称
24     // priority: 线程的优先级
25     // ticks: 线程的时间片总时间
26
27     // 成功, 返回pid; 失败, 返回-1
28     int executeThread(ThreadFunction function, void *parameter, const char *name, int
priority);
29
30     // 分配一个PCB
31     PCB *allocatePCB();
32     // 归还一个PCB
33     // program: 待释放的PCB
34     void releasePCB(PCB *program);
35
36     // 执行线程调度
37     void schedule();
38
39     // 设置线程的ticks值
40     void setThreadTicks(int pid, int ticks);
41 };

```

```

void ProgramManager::setThreadTicks(int pid, int ticks)
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    // 遍历所有程序查找指定pid
    ListItem *item = allPrograms.front();
    while (item)
    {
        PCB *program = ListItem2PCB(item, tagInAllList);
        if (program->pid == pid)
        {
            program->ticks = ticks;
            break;
        }
        item = item->next;
    }

    interruptManager.setInterruptStatus(status);
}

```

他的作用是利用 `executeThread` 函数返回的整数，通过遍历就绪队列找到对应的 PCB 并修改它的 `ticks` 值。这一函数通常跟在 `executeThread` 函数后面，作为新线程创建的一部分。

②修改内核启动程序：在 `assignment2` 中，我们的线程都编写在了 `setup.cpp` 中，这样写会导致几个父进程在同一时间被创建，无法很好地模拟实际工作情况中运行中途有新的进程创建这种情况，也无法很好地体现抢占式调度算法的先进之处。因此，在 `setup.cpp` 中，我们只创建了第一个进程（即永远不会被返回的 `pid0`），并给他的优先级进行了慎重的设置，确保它在整个运行周期的最后才被调度，以防止它长期占用 CPU 导致其他进程饥饿。代码如下：

```

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

extern "C" void setup_kernel()
{
    // 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(供FIFO、RR和SJF使用)
    //int pid = programManager.executeThread(initialise_thread, nullptr, "initialise thread", 8);

    // 创建第一个线程(供PSA使用)
    int pid = programManager.executeThread(initialise_thread, nullptr, "initialise thread", 1);

    if (pid == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }

    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);

    asm_halt();
}

```

可以看到，我们在 setup.cpp 中只创建了 initialise_thread，并且根据调度算法的不同，我们需要分别初始化它的优先级为最大/最小（SJF 算法中设置优先级为最大，则它的时间片 ticks 初始化为 priority*10，确保是所有进程中时间最长即可；而 PSA 算法中设置优先级为最低即可）。

③修改中断处理函数：由于在整个程序中，我们是通过中断处理函数的执行来模仿时间的流逝，即每中断一次就相当于 1 个时间单位。因此中断处理函数起到了我们大一程序设计中的 main 函数的作用，修改它是必然的。

前面提到，我们在 setup 中无法实现异步地创建线程，而在 interrupt.cpp 中，我们前面在 assignment2 中提到有一个全局变量 times，它会统计时钟中断的次数，我们可以用 times 作条件判断，从而进行异步创建线程。

首先，我们先在 interrupt 中声明线程：

```

10 extern void sixth_thread(void *arg) {
11     printf("pid %d name \"%s\": thread6 is created! \n", programManager.running->pid, programManager.running->name);
12     asm_halt();
13 }
14
15 extern void fifth_thread(void *arg) {
16     printf("pid %d name \"%s\": thread5 is created! \n", programManager.running->pid, programManager.running->name);
17     asm_halt();
18 }
19
20 extern void forth_thread(void *arg) {
21     printf("pid %d name \"%s\": thread4 is created! \n", programManager.running->pid, programManager.running->name);
22     asm_halt();
23 }
24
25
26 void third_thread(void *arg) {
27     printf("pid %d name \"%s\": thread3 is created! \n", programManager.running->pid, programManager.running->name);
28     asm_halt();
29 }
30
31 void second_thread(void *arg) {
32     printf("pid %d name \"%s\": thread2 is created! \n", programManager.running->pid, programManager.running->name);
33     asm_halt();
34 }
35
36 void first_thread(void *arg) {
37     printf("pid %d name \"%s\": thread1 is created! \n", programManager.running->pid, programManager.running->name);
38     asm_halt();
39 }

```

随后，我们修改中断处理函数 `c_time_interrupt_handler()`。这里有一点需要注意，抢占式调度算法的中断处理函数和非抢占式调度算法的中断处理函数是不一样的。非抢占式调度算法的中断处理函数只需要在正在运行的进程结束了再进行调度判断即可，而抢占式调度算法的中断处理函数则需要在每一次中断都判断是否需要调度。它们的代码如下：

```

// 中断处理函数(非抢占式)
// extern "C" void c_time_interrupt_handler()
// {
//     PCB *cur = programManager.running;
//
//     .....|
//
//     // 更新全局时间计数器
//     ++times;
//
//     if(cur->pid==0){
//         programManager.schedule();
//         return ;
//     }
//
//     if(cur->ticksPassedBy%10==0&&cur->ticks!=0){
//         printf("pid %d name \"%s\": is running, conducting ticks %d\\%d \n",
//             cur->pid, cur->name,cur->ticksPassedBy,cur->ticks+cur->ticksPassedBy);
//     }
//
//     if (cur->ticks)
//     {
//         --cur->ticks;
//         ++cur->ticksPassedBy;
//     }
//     else if(cur->ticks==0&&cur->ticksPassedBy!=0)
//     {
//         printf("pid %d name \"%s\": is dead, conducting ticks %d\\%d \n",
//             cur->pid, cur->name,cur->ticksPassedBy,cur->ticks+cur->ticksPassedBy);
//         cur->status = ProgramStatus::DEAD;
//         programManager.schedule();
//     }
// }

```

（注：截图中的.....是为了节省篇幅省略了进程创建的代码，在后面会说明并体现）。

这是非抢占式的中断处理函数，它有以下细节：(1)我们需要在比较靠前的地

方判断正在运行进程的 pid 是否为 0，为 0 的情况模拟了 CPU 空闲的状态（因为我们的程序假定了 pid0 不参与调度），根据理论知识，如果 CPU 空闲，我们就可以进行调度了；(2)我们在每个进程运行了 10 的倍数时间片的时候，就让它输出已经运行的时间，方便查看调度效果和 debug；(3)我们对程序结束的判断进行了修改：在 cur->ticks==0 的基础上加上了 &&cur->ticksPassedBy!=0，因为在一开始运行时发现进程创建的太多，运行到某个时间的时候，系统会统一判断一次进程，有还未执行的进程被当成了已经结束的进程，导致 PCB 被释放。

```
// 中断处理函数(抢占式)
extern "C" void c_time_interrupt_handler()
{
    PCB *cur = programManager.running;

    .....

    // 更新全局时间计数器
    ++times;

    if(cur->pid == 0){
        programManager.schedule();
        return;
    }

    if(cur->ticksPassedBy%10==0&&cur->ticks!=0){
        printf("pid %d name \"%s\": is running, remaining ticks %d/%d \n",
            cur->pid, cur->name, cur->ticksPassedBy, cur->ticks + cur->ticksPassedBy);
    }

    if (cur->ticks)
    {
        --cur->ticks;
        ++cur->ticksPassedBy;
        // 对于抢占式SJF, 我们需要在每次时间片结束时都检查是否有更短作业到达
        programManager.schedule();
    }
    else if (cur->ticks==0&&cur->ticksPassedBy!=0)
    {
        printf("pid %d name \"%s\": is dead, conducting ticks %d/%d \n",
            cur->pid, cur->name, cur->ticksPassedBy, cur->ticks+cur->ticksPassedBy);
        cur->status = ProgramStatus::DEAD;
        programManager.schedule();
    }
}
```

这是抢占式的中断处理函数，我们同样省略了创建线程的代码。与非抢占式的不同，抢占式的中断处理函数需要我们在每个时钟周期都判断是否需要调度，剩余的代码和非抢占式完全一样。

到这里，不同调度算法的通用修改地方已经讲完了，接下来我们将以调度算法为单位，接着展示代码修改以及运行结果。

(1) FIFO/FCFS 先到先服务调度算法

④调度算法修改：调度算法修改在 program.cpp 的 schedule()中：

```

// FIFO调度算法
// void ProgramManager::schedule()
// {
//     bool status = interruptManager.getInterruptStatus();
//     interruptManager.disableInterrupt();

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

//     if (running->status == ProgramStatus::RUNNING)
//     {
//         running->status = ProgramStatus::READY;
//         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();

//     asm_switch_thread(cur, next);

//     interruptManager.setInterruptStatus(status);
// }

```

FIFO 调度算法的实现比较简单，只需要在原理的算法上删掉对 ticks 的修改即可。算法的思路是每次进程运行结束后调用调度算法，调度算法直接去就绪队列中获取队头的进程，并进行更换即可。

⑤编写测试样例：这一部分就是前面提到省略号省略的地方。由于面对不同的调度算法，我们需要编写不同的测试样例才能最好地展现算法的正确性和特点，因此每个调度算法的测试样例不同是正常的。

FIFO 调度算法的测试样例如下：

```

if(times==1){
    int pid1=programManager.executeThread(first_thread, nullptr, "first thread", 3);
    programManager.setThreadTicks(pid1, 15);

    int pid2=programManager.executeThread(second_thread, nullptr, "second thread", 2);
    programManager.setThreadTicks(pid2, 22);

    int pid3=programManager.executeThread(third_thread, nullptr, "third thread", 4);
    programManager.setThreadTicks(pid3, 50);
}

if(times==20){
    int pid5=programManager.executeThread(fifth_thread, nullptr, "fifth thread", 6);
    programManager.setThreadTicks(pid5, 10);
}

if(times==63){
    int pid4=programManager.executeThread(forth_thread, nullptr, "forth thread", 3);
    programManager.setThreadTicks(pid4, 27);
}

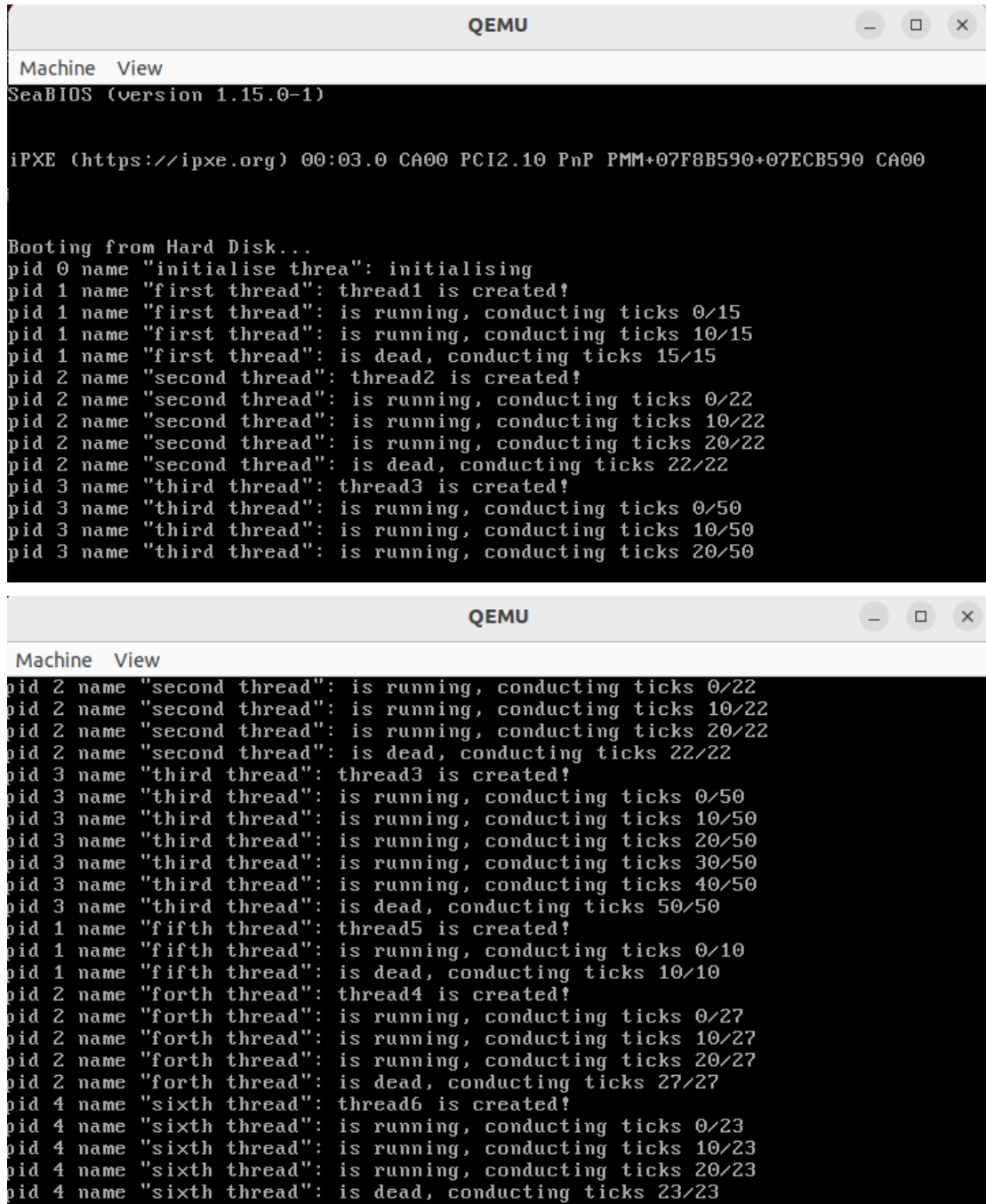
if(times==88){
    int pid6=programManager.executeThread(sixth_thread, nullptr, "sixth thread", 2);
    programManager.setThreadTicks(pid6, 23);
}

```

这里我们调换了 pid4 和 pid5 的创建顺序,他们都在自己被执行前创建完成,正好可以检验算法是否符合先进先出的原理。

理论上,线程运行的顺序为 1->2->3->5->4->6。

⑥运行程序:选择并替换好调度算法、初始化进程的优先级、测试样例后,程序运行结果如下:



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

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

Booting from Hard Disk...
pid 0 name "initialise threa": initialising
pid 1 name "first thread": thread1 is created!
pid 1 name "first thread": is running, conducting ticks 0/15
pid 1 name "first thread": is running, conducting ticks 10/15
pid 1 name "first thread": is dead, conducting ticks 15/15
pid 2 name "second thread": thread2 is created!
pid 2 name "second thread": is running, conducting ticks 0/22
pid 2 name "second thread": is running, conducting ticks 10/22
pid 2 name "second thread": is running, conducting ticks 20/22
pid 2 name "second thread": is dead, conducting ticks 22/22
pid 3 name "third thread": thread3 is created!
pid 3 name "third thread": is running, conducting ticks 0/50
pid 3 name "third thread": is running, conducting ticks 10/50
pid 3 name "third thread": is running, conducting ticks 20/50

pid 2 name "second thread": is running, conducting ticks 0/22
pid 2 name "second thread": is running, conducting ticks 10/22
pid 2 name "second thread": is running, conducting ticks 20/22
pid 2 name "second thread": is dead, conducting ticks 22/22
pid 3 name "third thread": thread3 is created!
pid 3 name "third thread": is running, conducting ticks 0/50
pid 3 name "third thread": is running, conducting ticks 10/50
pid 3 name "third thread": is running, conducting ticks 20/50
pid 3 name "third thread": is running, conducting ticks 30/50
pid 3 name "third thread": is running, conducting ticks 40/50
pid 3 name "third thread": is dead, conducting ticks 50/50
pid 1 name "fifth thread": thread5 is created!
pid 1 name "fifth thread": is running, conducting ticks 0/10
pid 1 name "fifth thread": is dead, conducting ticks 10/10
pid 2 name "forth thread": thread4 is created!
pid 2 name "forth thread": is running, conducting ticks 0/27
pid 2 name "forth thread": is running, conducting ticks 10/27
pid 2 name "forth thread": is running, conducting ticks 20/27
pid 2 name "forth thread": is dead, conducting ticks 27/27
pid 4 name "sixth thread": thread6 is created!
pid 4 name "sixth thread": is running, conducting ticks 0/23
pid 4 name "sixth thread": is running, conducting ticks 10/23
pid 4 name "sixth thread": is running, conducting ticks 20/23
pid 4 name "sixth thread": is dead, conducting ticks 23/23
```

我们可以看到,程序确实是按照顺序执行,且并没有发生抢占的情况。并且,thread5 和 thread4 的 pid 分别为 1 和 2,原因是他们两个在进程创建的时候,

thread1 和 thread2 已经执行完并回收了 PCB，而 thread6 的 pid 为 4 是因为它在被创建的时候实际上前三个进程还未被执行完。

为了展示区别，这里我们顺带附上 RR 调度算法的测试样例和运行结果：

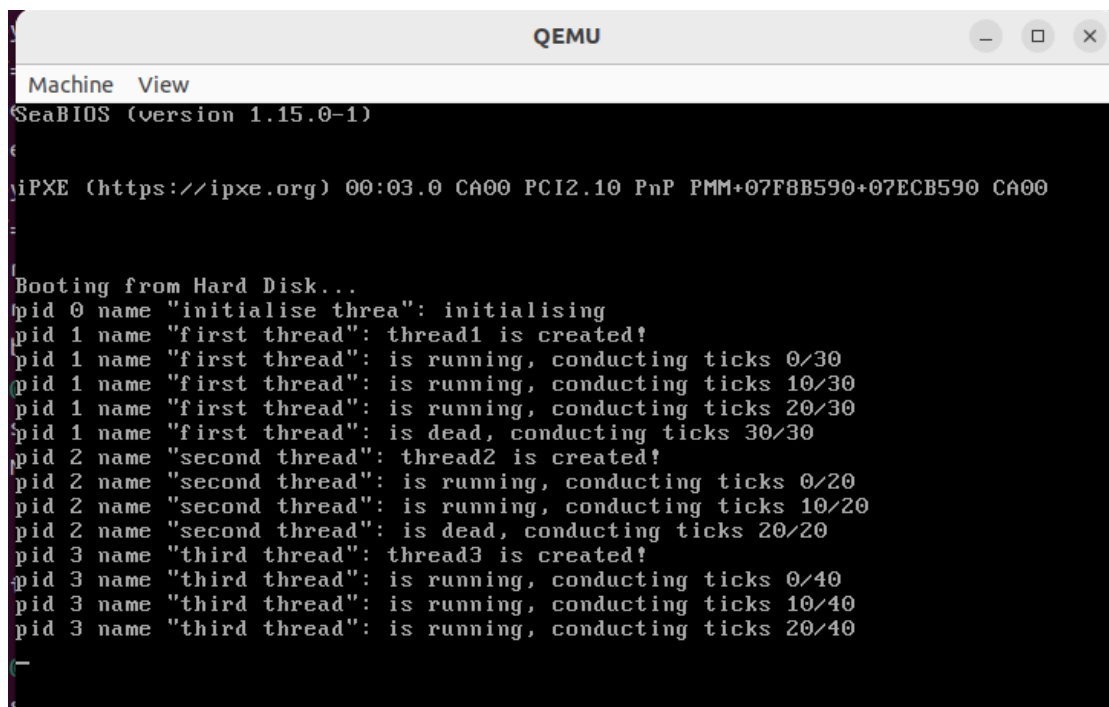
1、供RR调度算法使用

```
if(times==1){
    programManager.executeThread(first_thread, nullptr, "first thread", 3);
    programManager.executeThread(second_thread, nullptr, "second thread", 2);
    programManager.executeThread(third_thread, nullptr, "third thread", 4);
}

if(times==20){
    programManager.executeThread(forth_thread, nullptr, "forth thread", 3);
}

if(times==63){
    programManager.executeThread(fifth_thread, nullptr, "fifth thread", 6);
}

if(times==88){
    programManager.executeThread(sixth_thread, nullptr, "sixth thread", 2);
}
```



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

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

Booting from Hard Disk...
pid 0 name "initialise threa": initialising
pid 1 name "first thread": thread1 is created!
pid 1 name "first thread": is running, conducting ticks 0/30
pid 1 name "first thread": is running, conducting ticks 10/30
pid 1 name "first thread": is running, conducting ticks 20/30
pid 1 name "first thread": is dead, conducting ticks 30/30
pid 2 name "second thread": thread2 is created!
pid 2 name "second thread": is running, conducting ticks 0/20
pid 2 name "second thread": is running, conducting ticks 10/20
pid 2 name "second thread": is dead, conducting ticks 20/20
pid 3 name "third thread": thread3 is created!
pid 3 name "third thread": is running, conducting ticks 0/40
pid 3 name "third thread": is running, conducting ticks 10/40
pid 3 name "third thread": is running, conducting ticks 20/40
```



```
QEMU
Machine View
pid 2 name "second thread": is dead, conducting ticks 20/20
pid 3 name "third thread": thread3 is created!
pid 3 name "third thread": is running, conducting ticks 0/40
pid 3 name "third thread": is running, conducting ticks 10/40
pid 3 name "third thread": is running, conducting ticks 20/40
pid 3 name "third thread": is running, conducting ticks 30/40
pid 3 name "third thread": is dead, conducting ticks 40/40
pid 4 name "forth thread": thread4 is created!
pid 4 name "forth thread": is running, conducting ticks 0/30
pid 4 name "forth thread": is running, conducting ticks 10/30
pid 4 name "forth thread": is running, conducting ticks 20/30
pid 4 name "forth thread": is dead, conducting ticks 30/30
pid 1 name "fifth thread": thread5 is created!
pid 1 name "fifth thread": is running, conducting ticks 0/60
pid 1 name "fifth thread": is running, conducting ticks 10/60
pid 1 name "fifth thread": is running, conducting ticks 20/60
pid 1 name "fifth thread": is running, conducting ticks 30/60
pid 1 name "fifth thread": is running, conducting ticks 40/60
pid 1 name "fifth thread": is running, conducting ticks 50/60
pid 1 name "fifth thread": is dead, conducting ticks 60/60
pid 2 name "sixth thread": thread6 is created!
pid 2 name "sixth thread": is running, conducting ticks 0/20
pid 2 name "sixth thread": is running, conducting ticks 10/20
pid 2 name "sixth thread": is dead, conducting ticks 20/20
```

我们可以看到每个程序的时间片都是 $\text{priority} \times 10$ ，并且符合先进先出的原则。同时对比之下也能看出 FIFO 调度算法和它的区别。

综上，FIFO 调度算法的实现成功。

(2) SJF 非抢占式最短作业优先算法：

④ 调度算法修改：调度算法修改如下：

```
145 // 非抢占式最短作业优先调度算法
146 void ProgramManager::schedule()
147 {
148     bool status = interruptManager.getInterruptStatus();
149     interruptManager.disableInterrupt();
150
151     if (readyPrograms.size() == 0)
152     {
153         interruptManager.setInterruptStatus(status);
154         return;
155     }
156
157     if (running->status == ProgramStatus::RUNNING)
158     {
159         running->status = ProgramStatus::READY;
160         readyPrograms.push_back(&(running->tagInGeneralList));
161     }
162     else if (running->status == ProgramStatus::DEAD)
163     {
164         releasePCB(running);
165     }
166
167     // 寻找执行时间最短的进程
168     ListItem *item = readyPrograms.front();
169     int shortest_ticks = 0x7FFFFFFF;
170     ListItem *selectedItem = nullptr;
171     PCB *selectedPCB = nullptr;
172
173     // 遍历就绪队列，找到剩余时间最短的进程
174     while (item)
175     {
176         PCB *temp = ListItem2PCB(item, tagInGeneralList);
177         if (temp->ticks < shortest_ticks)
178         {
179             selectedItem = item;
180             selectedPCB = temp;
181             shortest_ticks = temp->ticks;
182         }
183         item = item->next;
184     }
185
186     // 从就绪队列中移除选中的进程
187     if (selectedItem)
188     {
189         readyPrograms.erase(selectedItem);
190     }
191
192     PCB *cur = running;
193     if (selectedPCB)
194     {
195         selectedPCB->status = ProgramStatus::RUNNING;
196         running = selectedPCB;
197     }
198
199     if (selectedPCB)
200     {
201         asm_switch_thread(cur, selectedPCB);
202     }
203
204     interruptManager.setInterruptStatus(status);
205 }
```


与前面两个算法的区别是，前面两个算法只需要把就绪队列队头的进程替换掉当前运行的进程即可，而在 SJF 算法中，我们需要遍历就绪队列，找到 ticks 最小的那个进程。

⑤编写测试样例：在 SJF 调度算法中，我们稍微修改了几个进程的时间片。测试样例如下：

```
if(times==1){
    int pid1=programManager.executeThread(first_thread, nullptr, "first thread", 3);
    programManager.setThreadTicks(pid1, 50);

    int pid2=programManager.executeThread(second_thread, nullptr, "second thread", 2);
    programManager.setThreadTicks(pid2, 22);

    int pid3=programManager.executeThread(third_thread, nullptr, "third thread", 4);
    programManager.setThreadTicks(pid3, 15);
}

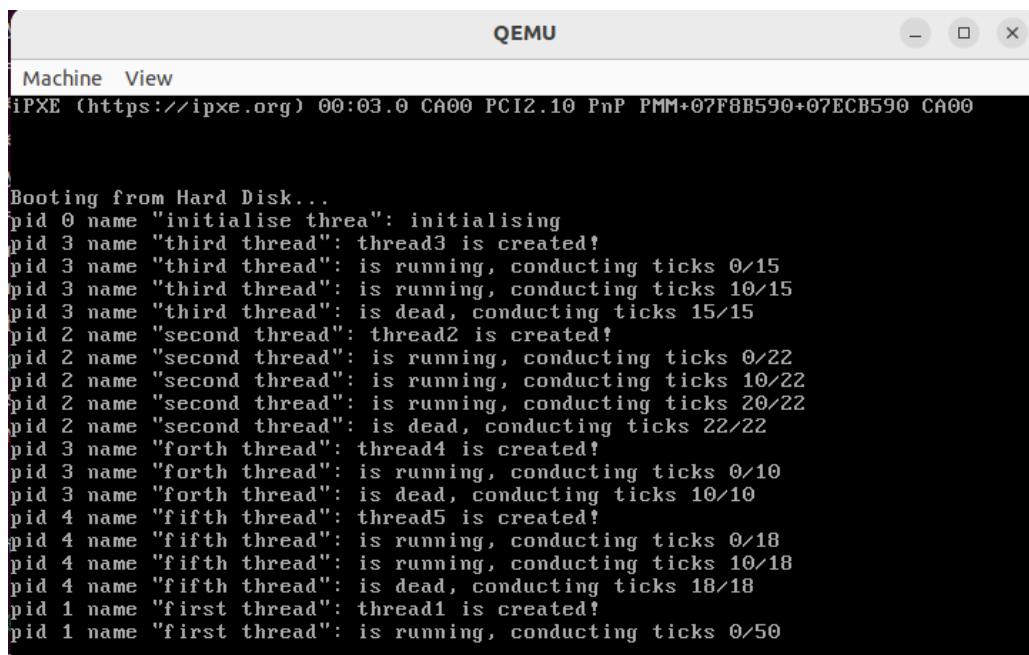
if(times==20){
    int pid4=programManager.executeThread(forth_thread, nullptr, "forth thread", 6);
    programManager.setThreadTicks(pid4, 10);
}

if(times==30){
    int pid5=programManager.executeThread(fifth_thread, nullptr, "fifth thread", 3);
    programManager.setThreadTicks(pid5, 18);
}

if(times==88){
    int pid6=programManager.executeThread(sixth_thread, nullptr, "sixth thread", 2);
    programManager.setThreadTicks(pid6, 23);
}
```

理论上的运行顺序是 3->2->4->5->1->6。

⑥运行程序：选择并替换好调度算法、初始化进程的优先级、测试样例后，程序运行结果如下：



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

Booting from Hard Disk...
pid 0 name "initialise threa": initialising
pid 3 name "third thread": thread3 is created!
pid 3 name "third thread": is running, conducting ticks 0/15
pid 3 name "third thread": is running, conducting ticks 10/15
pid 3 name "third thread": is dead, conducting ticks 15/15
pid 2 name "second thread": thread2 is created!
pid 2 name "second thread": is running, conducting ticks 0/22
pid 2 name "second thread": is running, conducting ticks 10/22
pid 2 name "second thread": is running, conducting ticks 20/22
pid 2 name "second thread": is dead, conducting ticks 22/22
pid 3 name "forth thread": thread4 is created!
pid 3 name "forth thread": is running, conducting ticks 0/10
pid 3 name "forth thread": is dead, conducting ticks 10/10
pid 4 name "fifth thread": thread5 is created!
pid 4 name "fifth thread": is running, conducting ticks 0/18
pid 4 name "fifth thread": is running, conducting ticks 10/18
pid 4 name "fifth thread": is dead, conducting ticks 18/18
pid 1 name "first thread": thread1 is created!
pid 1 name "first thread": is running, conducting ticks 0/50
```

```
Machine View
pid 2 name "second thread": thread2 is created!
pid 2 name "second thread": is running, conducting ticks 0/22
pid 2 name "second thread": is running, conducting ticks 10/22
pid 2 name "second thread": is running, conducting ticks 20/22
pid 2 name "second thread": is dead, conducting ticks 22/22
pid 3 name "forth thread": thread4 is created!
pid 3 name "forth thread": is running, conducting ticks 0/10
pid 3 name "forth thread": is dead, conducting ticks 10/10
pid 4 name "fifth thread": thread5 is created!
pid 4 name "fifth thread": is running, conducting ticks 0/18
pid 4 name "fifth thread": is running, conducting ticks 10/18
pid 4 name "fifth thread": is dead, conducting ticks 18/18
pid 1 name "first thread": thread1 is created!
pid 1 name "first thread": is running, conducting ticks 0/50
pid 1 name "first thread": is running, conducting ticks 10/50
pid 1 name "first thread": is running, conducting ticks 20/50
pid 1 name "first thread": is running, conducting ticks 30/50
pid 1 name "first thread": is running, conducting ticks 40/50
pid 1 name "first thread": is dead, conducting ticks 50/50
pid 2 name "sixth thread": thread6 is created!
pid 2 name "sixth thread": is running, conducting ticks 0/23
pid 2 name "sixth thread": is running, conducting ticks 10/23
pid 2 name "sixth thread": is running, conducting ticks 20/23
pid 2 name "sixth thread": is dead, conducting ticks 23/23
```

通过观察我们发现，线程执行顺序确实是 324516，调度算法实现成功。

(3) PSJF 抢占式最短作业优先算法：

④调度算法修改：调度算法修改如下：

```
// 抢占式最短作业优先调度算法（最短剩余时间优先）
void ProgramManager::schedule()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

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

    // 寻找剩余执行时间最短的进程
    ListItem *item = readyPrograms.front();
    PCB *shortestJob = nullptr;
    int shortestRemaining = 0x7FFFFFFF; // 初始化为最大整数值
    ListItem *selectedItem = nullptr;

    // 遍历就绪队列，找到剩余时间最短的进程
    while (item)
    {
        PCB *temp = ListItem2PCB(item, tagInGeneralList);
        if (temp->ticks < shortestRemaining)
        {
            shortestJob = temp;
            shortestRemaining = temp->ticks;
            selectedItem = item;
        }
        item = item->next;
    }

    // 获取当前运行进程的剩余时间（如果正在运行）
    int currentRemaining = 0x7FFFFFFF;
    if (running->status == ProgramStatus::RUNNING)
    {
        currentRemaining = running->ticks;
    }

    // 检查是否需要抢占当前进程
    bool needPreempt = (shortestRemaining < currentRemaining) || (running->status != ProgramStatus::RUNNING);

    if (needPreempt)
    {
        // 处理当前运行进程
        if (running->status == ProgramStatus::RUNNING)
        {
            running->status = ProgramStatus::READY;
            readyPrograms.push_back(&(running->tagInGeneralList));
        }
        else if (running->status == ProgramStatus::DEAD)
        {
            releasePCB(running);
        }

        // 从就绪队列中移除选中的进程
        if (selectedItem)
        {
            readyPrograms.erase(selectedItem);
        }

        // 切换到新进程
        PCB *cur = running;
        shortestJob->status = ProgramStatus::RUNNING;
        running = shortestJob;

        asm_switch_thread(cur, shortestJob);
    }

    interruptManager.setInterruptStatus(status);
}
```

与非抢占式不同的是，抢占式最短作业优先算法除了要找到就绪队列中 ticks 最少的作业，还需要判断当前运行的作业和找到的作业之间哪个剩余时间更短，

以决定是否需要抢占。

⑤编写测试样例：PSJF 算法的测试样例如下：

4、供抢占式 SJF 调度算法使用

```
if(times==1){
    int pid1=programManager.executeThread(first_thread, nullptr, "first thread", 3);
    programManager.setThreadTicks(pid1, 50);

    int pid2=programManager.executeThread(second_thread, nullptr, "second thread", 2);
    programManager.setThreadTicks(pid2, 60);

    int pid3=programManager.executeThread(third_thread, nullptr, "third thread", 4);
    programManager.setThreadTicks(pid3, 40);
}

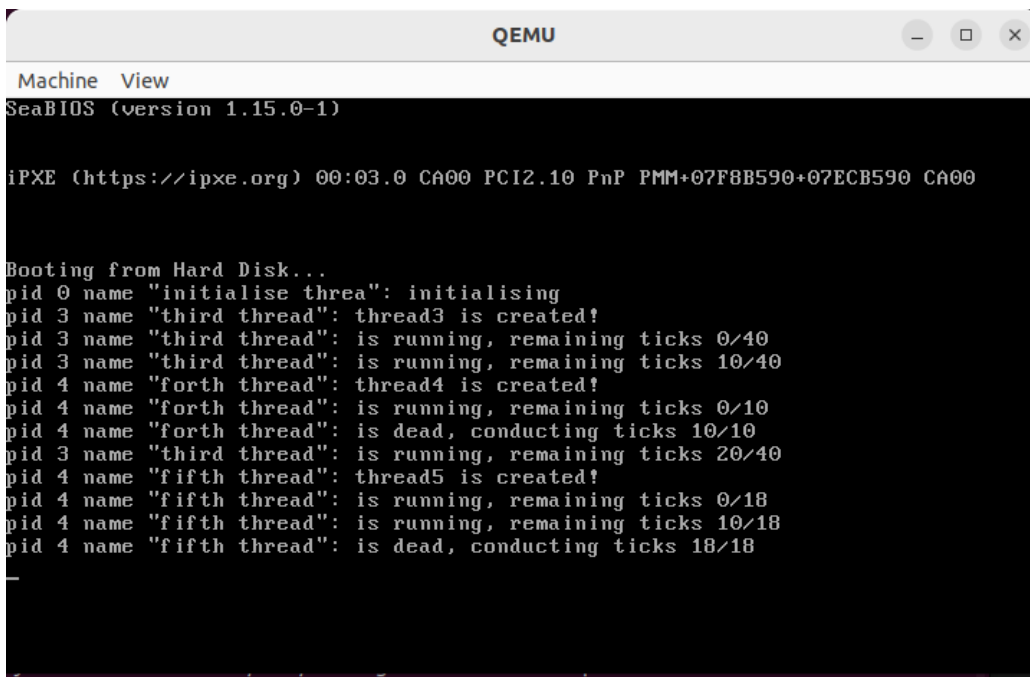
if(times==15){
    int pid4=programManager.executeThread(forth_thread, nullptr, "forth thread", 6);
    programManager.setThreadTicks(pid4, 10);
}

if(times==33){
    int pid5=programManager.executeThread(fifth_thread, nullptr, "fifth thread", 3);
    programManager.setThreadTicks(pid5, 18);
}

if(times==88){
    int pid6=programManager.executeThread(sixth_thread, nullptr, "sixth thread", 2);
    programManager.setThreadTicks(pid6, 23);
}
```

理论上的运行顺序为 3->4->3->5->3->1->6->1->2。

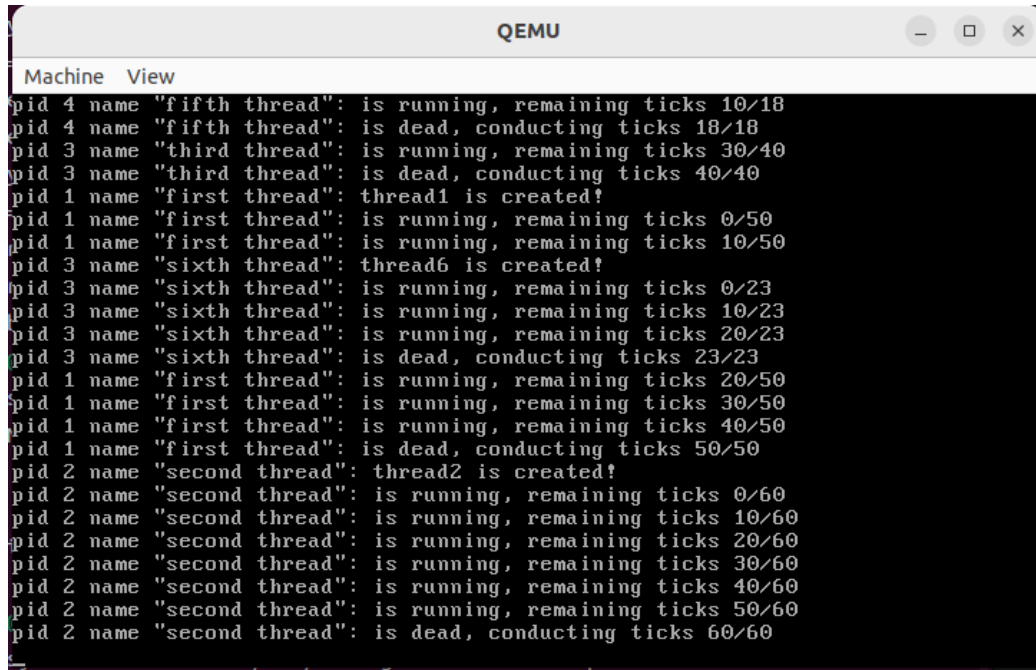
⑥运行程序：选择并替换好调度算法、中断处理函数、初始化进程的优先级、测试样例后，程序运行结果如下：



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

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

Booting from Hard Disk...
pid 0 name "initialise threa": initialising
pid 3 name "third thread": thread3 is created!
pid 3 name "third thread": is running, remaining ticks 0/40
pid 3 name "third thread": is running, remaining ticks 10/40
pid 4 name "forth thread": thread4 is created!
pid 4 name "forth thread": is running, remaining ticks 0/10
pid 4 name "forth thread": is dead, conducting ticks 10/10
pid 3 name "third thread": is running, remaining ticks 20/40
pid 4 name "fifth thread": thread5 is created!
pid 4 name "fifth thread": is running, remaining ticks 0/18
pid 4 name "fifth thread": is running, remaining ticks 10/18
pid 4 name "fifth thread": is dead, conducting ticks 18/18
```



```
Machine View
pid 4 name "fifth thread": is running, remaining ticks 10/18
pid 4 name "fifth thread": is dead, conducting ticks 18/18
pid 3 name "third thread": is running, remaining ticks 30/40
pid 3 name "third thread": is dead, conducting ticks 40/40
pid 1 name "first thread": thread1 is created!
pid 1 name "first thread": is running, remaining ticks 0/50
pid 1 name "first thread": is running, remaining ticks 10/50
pid 3 name "sixth thread": thread6 is created!
pid 3 name "sixth thread": is running, remaining ticks 0/23
pid 3 name "sixth thread": is running, remaining ticks 10/23
pid 3 name "sixth thread": is running, remaining ticks 20/23
pid 3 name "sixth thread": is dead, conducting ticks 23/23
pid 1 name "first thread": is running, remaining ticks 20/50
pid 1 name "first thread": is running, remaining ticks 30/50
pid 1 name "first thread": is running, remaining ticks 40/50
pid 1 name "first thread": is dead, conducting ticks 50/50
pid 2 name "second thread": thread2 is created!
pid 2 name "second thread": is running, remaining ticks 0/60
pid 2 name "second thread": is running, remaining ticks 10/60
pid 2 name "second thread": is running, remaining ticks 20/60
pid 2 name "second thread": is running, remaining ticks 30/60
pid 2 name "second thread": is running, remaining ticks 40/60
pid 2 name "second thread": is running, remaining ticks 50/60
pid 2 name "second thread": is dead, conducting ticks 60/60
```

通过观察我们发现，线程执行顺序确实是 343531612，调度算法实现成功。

(4) NP-Priority 优先级调度算法：

④ 调度算法修改：调度算法修改如下：

```
// 非抢占式优先级调度算法
void ProgramManager::schedule()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

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

    if (running->status == ProgramStatus::RUNNING)
    {
        running->status = ProgramStatus::READY;
        readyPrograms.push_back(&(running->tagInGeneralList));
    }
    else if (running->status == ProgramStatus::DEAD)
    {
        releasePCB(running);
    }

    // 寻找优先级最高的进程
    ListItem *item = readyPrograms.front();
    int highestPriority = 0x00000000;
    ListItem *selectedItem = nullptr;
    PCB *selectedPCB = nullptr;

    // 遍历就绪队列，找到优先级最高的进程
    while (item)
    {
        PCB *temp = ListItem2PCB(item, tagInGeneralList);
        if (temp->priority > highestPriority)
        {
            selectedItem = item;
            selectedPCB = temp;
            highestPriority = temp->priority;
        }
        item = item->next;
    }

    if (selectedItem)
    {
        readyPrograms.erase(selectedItem);

        PCB *cur = running;
        if (selectedPCB)
        {
            selectedPCB->status = ProgramStatus::RUNNING;
            running = selectedPCB;
        }

        if (selectedPCB)
        {
            asm_switch_thread(cur, selectedPCB);
        }

        interruptManager.setInterruptStatus(status);
    }
}
```

NP-P 算法的实现和 SJF 算法基本一样，唯一的区别就是把选择条件从 ticks 最小改成 Priority 最大，即选择优先级最高的进程进行替换。

⑤编写测试样例：NP-P 算法的测试样例如下：

5、供PSA调度算法使用

```
if(times==1){
    int pid1=programManager.executeThread(first_thread, nullptr, "first thread", 1);
    programManager.setThreadTicks(pid1, 50);

    int pid2=programManager.executeThread(second_thread, nullptr, "second thread", 2);
    programManager.setThreadTicks(pid2, 22);

    int pid3=programManager.executeThread(third_thread, nullptr, "third thread", 4);
    programManager.setThreadTicks(pid3, 15);
}

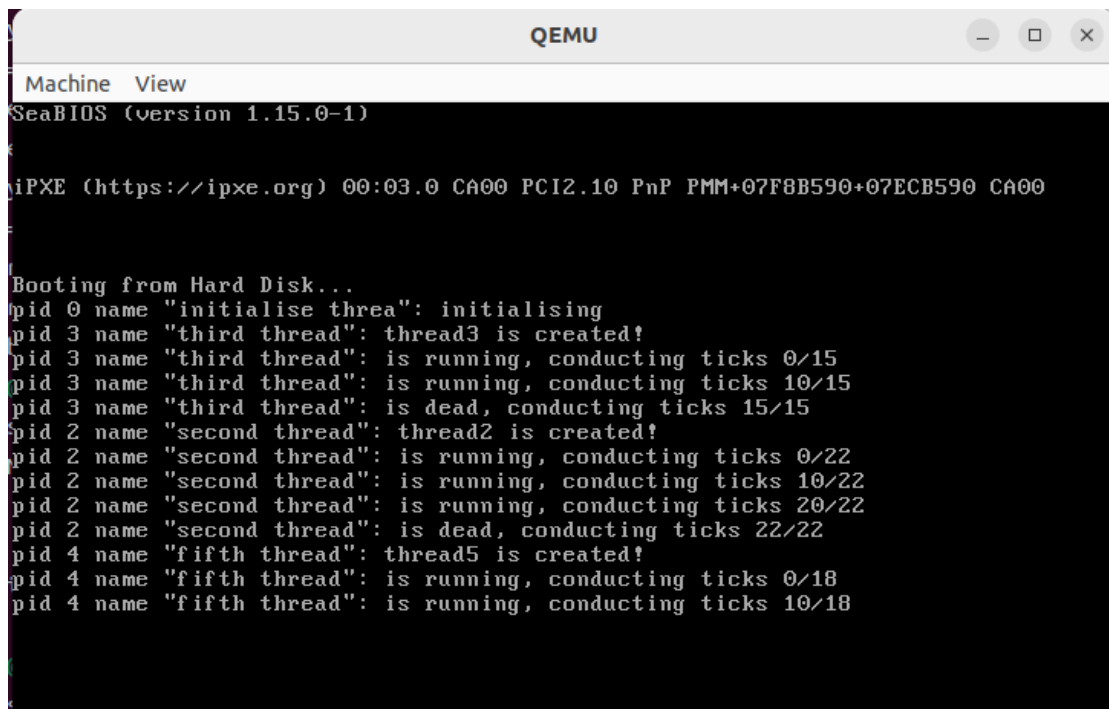
if(times==20){
    int pid4=programManager.executeThread(forth_thread, nullptr, "forth thread", 1);
    programManager.setThreadTicks(pid4, 10);
}

if(times==20){
    int pid5=programManager.executeThread(fifth_thread, nullptr, "fifth thread", 6);
    programManager.setThreadTicks(pid5, 18);
}

if(times==88){
    int pid6=programManager.executeThread(sixth_thread, nullptr, "sixth thread", 5);
    programManager.setThreadTicks(pid6, 23);
}
```

理论上的运行顺序为：3->2->5->1->6->4。

⑥运行程序：选择并替换好调度算法、初始化进程的优先级、测试样例后，程序运行结果如下：

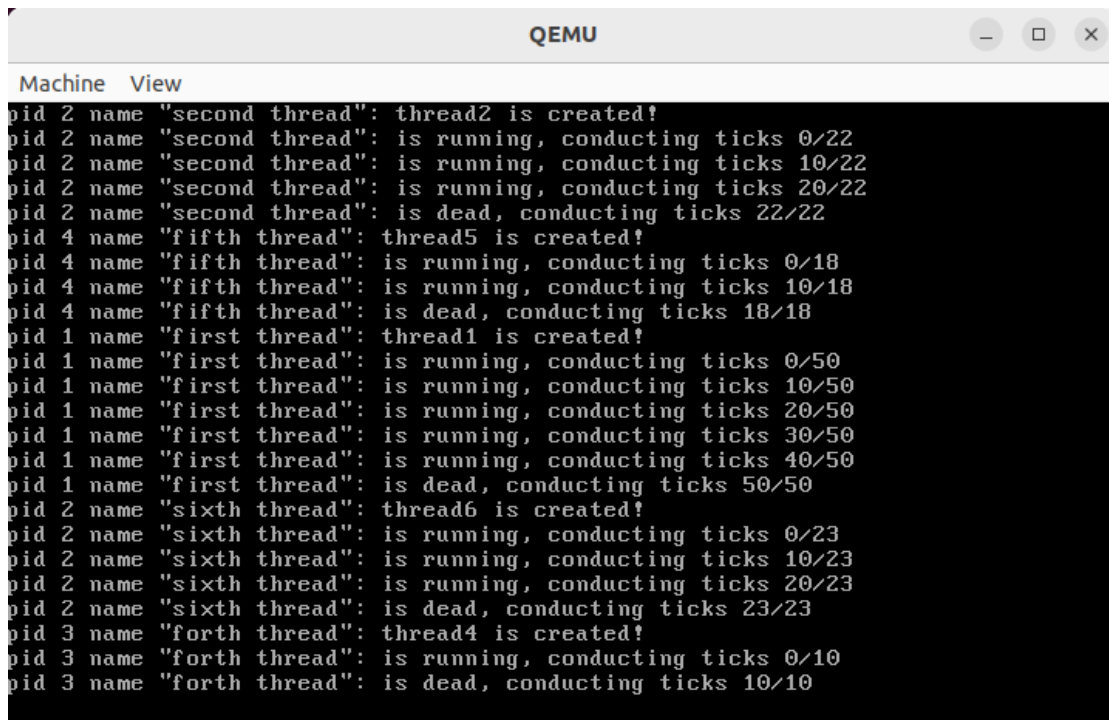


The screenshot shows a QEMU window titled "QEMU" with a "Machine View" tab. The output text is as follows:

```
SeaBIOS (version 1.15.0-1)

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

Booting from Hard Disk...
pid 0 name "initialise threa": initialising
pid 3 name "third thread": thread3 is created!
pid 3 name "third thread": is running, conducting ticks 0/15
pid 3 name "third thread": is running, conducting ticks 10/15
pid 3 name "third thread": is dead, conducting ticks 15/15
pid 2 name "second thread": thread2 is created!
pid 2 name "second thread": is running, conducting ticks 0/22
pid 2 name "second thread": is running, conducting ticks 10/22
pid 2 name "second thread": is running, conducting ticks 20/22
pid 2 name "second thread": is dead, conducting ticks 22/22
pid 4 name "fifth thread": thread5 is created!
pid 4 name "fifth thread": is running, conducting ticks 0/18
pid 4 name "fifth thread": is running, conducting ticks 10/18
```

A screenshot of a QEMU Machine View window. The window title is "QEMU". Below the title bar, the text "Machine View" is displayed. The main area shows a log of thread execution. The log entries are as follows:
pid 2 name "second thread": thread2 is created!
pid 2 name "second thread": is running, conducting ticks 0/22
pid 2 name "second thread": is running, conducting ticks 10/22
pid 2 name "second thread": is running, conducting ticks 20/22
pid 2 name "second thread": is dead, conducting ticks 22/22
pid 4 name "fifth thread": thread5 is created!
pid 4 name "fifth thread": is running, conducting ticks 0/18
pid 4 name "fifth thread": is running, conducting ticks 10/18
pid 4 name "fifth thread": is dead, conducting ticks 18/18
pid 1 name "first thread": thread1 is created!
pid 1 name "first thread": is running, conducting ticks 0/50
pid 1 name "first thread": is running, conducting ticks 10/50
pid 1 name "first thread": is running, conducting ticks 20/50
pid 1 name "first thread": is running, conducting ticks 30/50
pid 1 name "first thread": is running, conducting ticks 40/50
pid 1 name "first thread": is dead, conducting ticks 50/50
pid 2 name "sixth thread": thread6 is created!
pid 2 name "sixth thread": is running, conducting ticks 0/23
pid 2 name "sixth thread": is running, conducting ticks 10/23
pid 2 name "sixth thread": is running, conducting ticks 20/23
pid 2 name "sixth thread": is dead, conducting ticks 23/23
pid 3 name "forth thread": thread4 is created!
pid 3 name "forth thread": is running, conducting ticks 0/10
pid 3 name "forth thread": is dead, conducting ticks 10/10

通过观察我们发现，线程执行顺序确实是 325164，调度算法实现成功。

(5) P-Priority 抢占式优先级调度算法:

④调度算法修改：调度算法修改如下：

```
// 抢占式优先级调度算法
void ProgramManager::schedule()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

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

    // 寻找优先级最高的进程
    ListItem *item = readyPrograms.front();
    PCB *highestPriorityJob = nullptr;
    int highestPriority = 0x00000000;
    ListItem *selectedItem = nullptr;

    // 遍历就绪队列，找到优先级最高的进程
    while (item)
    {
        PCB *temp = ListItem2PCB(item, tagInGeneralList);
        if (temp->priority > highestPriority)
        {
            highestPriorityJob = temp;
            highestPriority = temp->priority;
            selectedItem = item;
        }
        item = item->next;
    }

    int currentPriority = 0x00000000;
    if (running->status == ProgramStatus::RUNNING){
        currentPriority = running->priority;
    }

    bool needPreempt = (highestPriority > currentPriority)
    || (running->status != ProgramStatus::RUNNING);

    if (needPreempt){
        //处理当前运行进程
        if (running->status == ProgramStatus::RUNNING){
            running->status = ProgramStatus::READY;
            readyPrograms.push_back(&(running->tagInGeneralList));
        }
        else if (running->status == ProgramStatus::DEAD){
            releasePCB(running);
        }
        // 从就绪队列中移除选中的进程
        if (selectedItem){
            readyPrograms.erase(selectedItem);
        }
        // 切换到新进程
        PCB *cur = running;
        highestPriorityJob->status = ProgramStatus::RUNNING;
        running = highestPriorityJob;

        asm_switch_thread(cur, highestPriorityJob);
    }
    interruptManager.setInterruptStatus(status);
}
```

P-P 算法的原理和 NP-P 一致，都是遍历就绪队列以查找优先级最大的线程，函数结构与 PSJF 一致，都需要再与当前运行的线程进行比较以确定是否需要进
行调度。

⑤编写测试样例：NP-P 算法的测试样例如下：


```

if(times==1){
    int pid1=programManager.executeThread(first_thread, nullptr, "first thread", 2);
    programManager.setThreadTicks(pid1, 50);

    int pid2=programManager.executeThread(second_thread, nullptr, "second thread", 3);
    programManager.setThreadTicks(pid2, 22);

    int pid3=programManager.executeThread(third_thread, nullptr, "third thread", 5);
    programManager.setThreadTicks(pid3, 15);
}

if(times==20){
    int pid4=programManager.executeThread(forth_thread, nullptr, "forth thread", 2);
    programManager.setThreadTicks(pid4, 10);
}

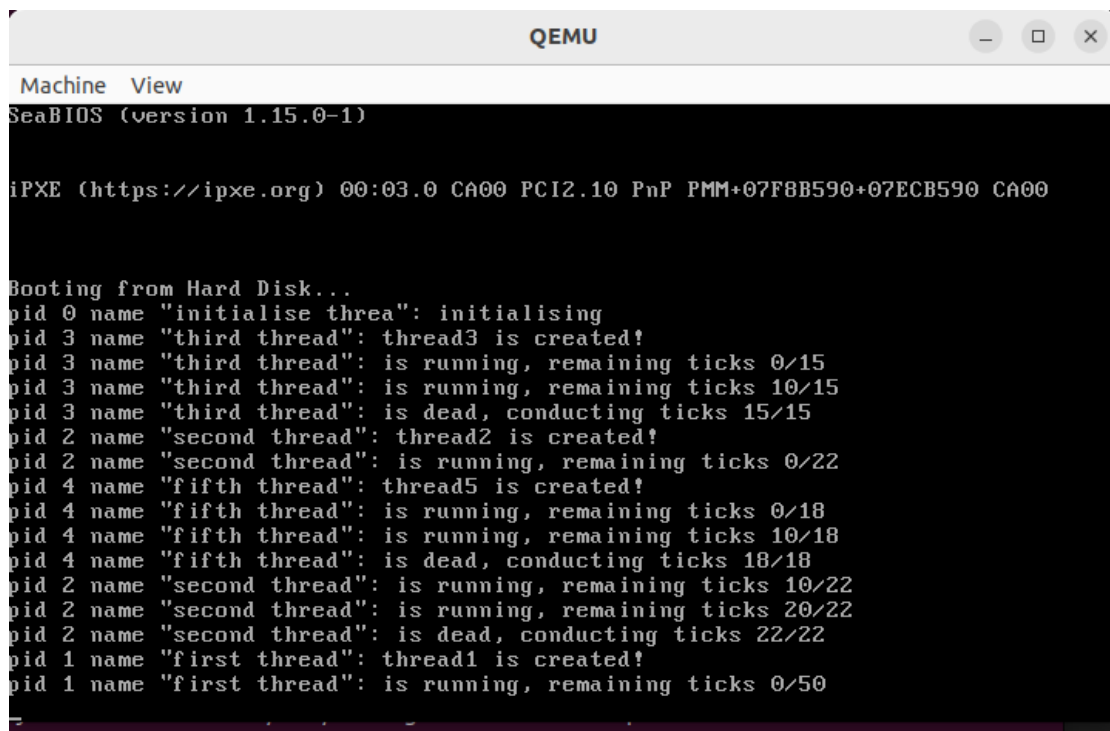
if(times==20){
    int pid5=programManager.executeThread(fifth_thread, nullptr, "fifth thread", 7);
    programManager.setThreadTicks(pid5, 18);
}

if(times==88){
    int pid6=programManager.executeThread(sixth_thread, nullptr, "sixth thread", 6);
    programManager.setThreadTicks(pid6, 23);
}

```

理论上的运行顺序为：3->2->5->2->1->6->4->1。

⑥运行程序：选择并替换好调度算法、时间中断函数、初始化进程的优先级、测试样例后，程序运行结果如下：



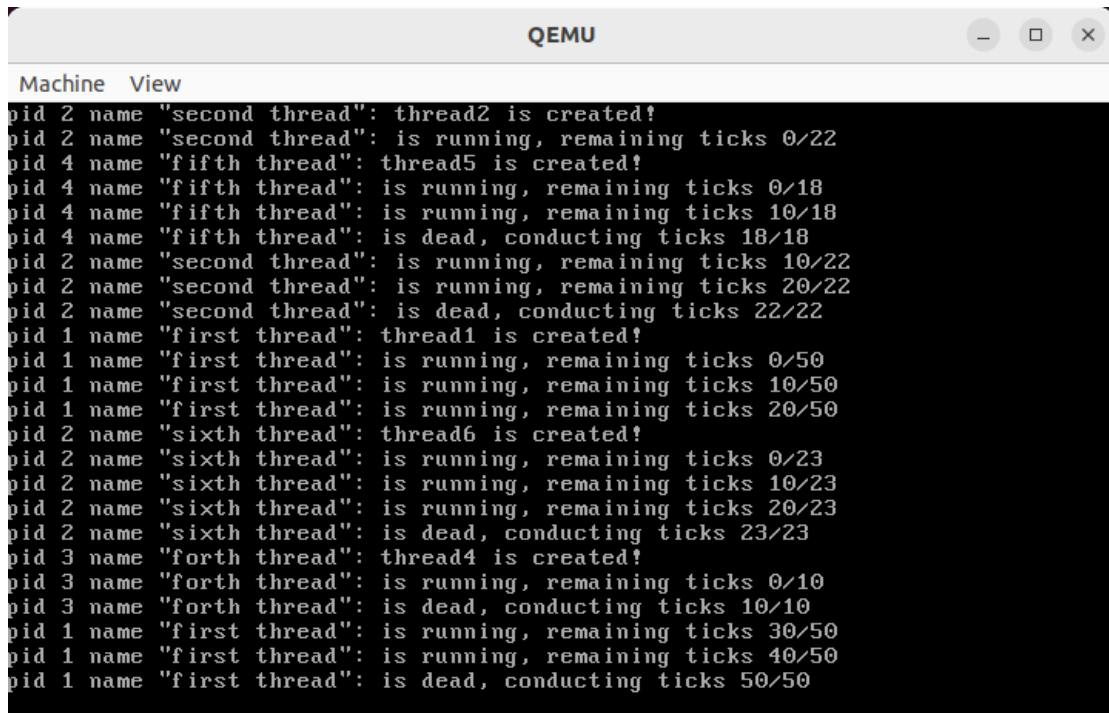
```

QEMU
Machine View
SeaBIOS (version 1.15.0-1)

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

Booting from Hard Disk...
pid 0 name "initialise threa": initialising
pid 3 name "third thread": thread3 is created!
pid 3 name "third thread": is running, remaining ticks 0/15
pid 3 name "third thread": is running, remaining ticks 10/15
pid 3 name "third thread": is dead, conducting ticks 15/15
pid 2 name "second thread": thread2 is created!
pid 2 name "second thread": is running, remaining ticks 0/22
pid 4 name "fifth thread": thread5 is created!
pid 4 name "fifth thread": is running, remaining ticks 0/18
pid 4 name "fifth thread": is running, remaining ticks 10/18
pid 4 name "fifth thread": is dead, conducting ticks 18/18
pid 2 name "second thread": is running, remaining ticks 10/22
pid 2 name "second thread": is running, remaining ticks 20/22
pid 2 name "second thread": is dead, conducting ticks 22/22
pid 1 name "first thread": thread1 is created!
pid 1 name "first thread": is running, remaining ticks 0/50

```



```
Machine View
pid 2 name "second thread": thread2 is created!
pid 2 name "second thread": is running, remaining ticks 0/22
pid 4 name "fifth thread": thread5 is created!
pid 4 name "fifth thread": is running, remaining ticks 0/18
pid 4 name "fifth thread": is running, remaining ticks 10/18
pid 4 name "fifth thread": is dead, conducting ticks 18/18
pid 2 name "second thread": is running, remaining ticks 10/22
pid 2 name "second thread": is running, remaining ticks 20/22
pid 2 name "second thread": is dead, conducting ticks 22/22
pid 1 name "first thread": thread1 is created!
pid 1 name "first thread": is running, remaining ticks 0/50
pid 1 name "first thread": is running, remaining ticks 10/50
pid 1 name "first thread": is running, remaining ticks 20/50
pid 2 name "sixth thread": thread6 is created!
pid 2 name "sixth thread": is running, remaining ticks 0/23
pid 2 name "sixth thread": is running, remaining ticks 10/23
pid 2 name "sixth thread": is running, remaining ticks 20/23
pid 2 name "sixth thread": is dead, conducting ticks 23/23
pid 3 name "forth thread": thread4 is created!
pid 3 name "forth thread": is running, remaining ticks 0/10
pid 3 name "forth thread": is dead, conducting ticks 10/10
pid 1 name "first thread": is running, remaining ticks 30/50
pid 1 name "first thread": is running, remaining ticks 40/50
pid 1 name "first thread": is dead, conducting ticks 50/50
```

通过观察我们发现，线程执行顺序确实是 32521641，调度算法实现成功。

至此，所有我实现了的调度算法均已调试完成并正确，实验任务 4 完成。

Section 5 实验总结与心得体会

这次实验是我觉得从 lab1 以来最有意思的实验，相比于前面实验中要么一头雾水地写汇编代码，要么是一些貌似和操作系统“没什么关系”的任务，这次实验切实通过自己的努力改变了操作系统的底层逻辑，并且得到了可观的效果。虽然 assignment3 的 debug 还是一头雾水.....

本次实验有以下几点需要注意：

(1)在 assignment2 中,各个子进程的结尾需要把 `asm_halt();`改为 `while(1){};`即让进程从停止运行改为空循环一直运行，否则到后面会因为前面的进程 PCB 被回收而导致后面进程的 pid 混乱，难以梳理。

(2)assignment2 中,我们只添加了父子进程关系的维护和创建时间的记录，或许是因为目前所学有限，似乎很难再添加一些别的常见的信息。

(3) assignment4 中,其实还有一些细节没有做好，比如说现实中第一个进程往往也是会参与调度的，但是因为实在是搞不定，而且本人想要创造一个为多个调度算法所通用的模板出来，因此只能采取“下位替代”：第一个进程不参与

调度。再之就是还有很多更复杂的调度算法没有尝试。

(4) assignment3 除了看到每次循环 `ticks+1` 以外，实在是不知道拿 `gdb` 能看些什么，希望以后助教和老师能在这个地方给多一些指导。

(5) assignment1 中，其实有一个用得很多的格式化参数：`%.nf`，就是保留多少位小数点浮点数，这个后续有时间的话可以挑战一下。

(6) 关于 assignment2 和 assignment4 中查看 `qemu` 输出，找了很久的资料也没找到怎么能够在输出被“滚走”之后回滚屏幕查看最开始的输出，最后只能采取最笨的方法：即抓紧时间截图，取两张截图的交集来看。

Section 6 附录：代码清单

代码请见附件。