LAB5 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: 内核线程

专业名称: 计算机科学与技术

学生姓名: 马福泉

学生学号: 23336179

实验地点: 实验中心 B202

实验时间:2025.4.9

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

- 预备知识：汇编语言基础，计算机体系结构，操作系统基础，调试工具的使用（如 gdb）等。
- 实验环境：
  - 虚拟机版本/处理器型号：VMware-Ubuntu22.04.5/i386（32 位）
  - 代码编辑环境： 编辑器：VSCode
    插件：C/C++插件，汇编插件
  - 代码编译工具： 编译器：gcc
    工具链：binutils、make、qemu、nasm 等
    调试工具：gdb
  - 重要三方库信息：GNU binutils：工具集（如 as、ld）
    gdb：调试工具
    QEMU、Bochs：模拟器用于测试 等

Section 3 实验任务

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，例如先来先服务，最短作业（进程）优先，响应比最高优先算法，优先级调度算法，多级反馈队列调度算法

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。

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

---------- **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：

  1. 复现一个可变参数的例子
  2. 复现可变参数机制的实现（main2.cpp）
  3. 改进 printf 函数

- 实验步骤：

  1. 复现一个可变参数的例子（main.cpp）

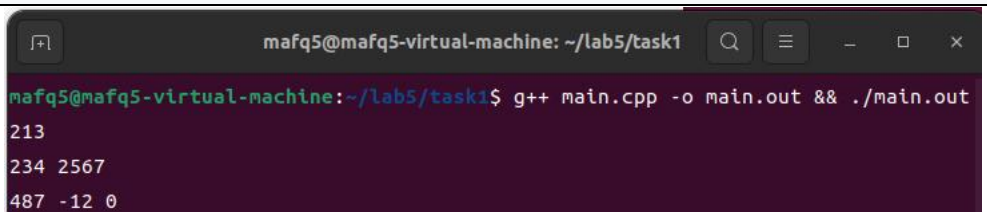
  复制代码到本地

编译运行。

```
g++ main.cpp -o main.out && ./main.out
```

输出如下结果。

```
213
234 2567
487 -12 0
```



2. 复现可变参数机制的实现（main2.cpp）

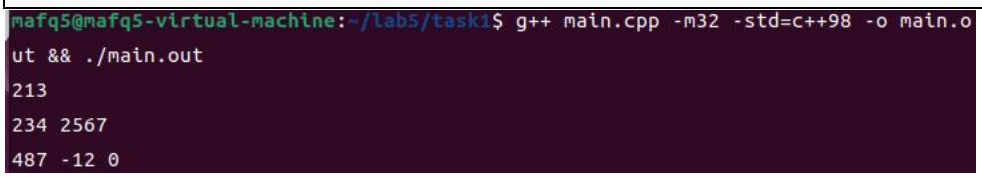
复制代码到本地

编译运行。

```
g++ main2.cpp -m32 -std=c++98 -o main2.out && ./main2.out
```

输出如下结果。

```
213
234 2567
487 -12 0
```



3. 在材料中（src/3）的 printf 上进行改进 printf 函数。添加`%f,%e` 这两种格式化输出类型。

(1) 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：用于声明字符转换函数的 `stdlib.h`；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`;还有 `setup.cpp`。

(2) 修改 stdlib.h 如下：

我们需要在 `stdlib.h` 声明分别需要用到的函数
由于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。

```h
#ifndef STDLIB_H
#define STDLIB_H

#include "os_type.h"

template<typename T>
void swap(T &x, T &y);

/*
 * 将一个非负整数转换为指定进制表示的字符串。
 * num: 待转换的非负整数。
 * mod: 进制。
 * 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。
 */

void itos(char *numStr, uint32 num, uint32 mod);
void ftos(char *numStr, double num);
void ftoe(char *numStr, double num);
#endif
```

(3) 修改 stdio.cpp ，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。
f 和 e 实则一样，如下：

```h
case 'f':
{
    double temp = va_arg(ap, double);

    if(temp < 0)
    {
        counter += printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
        temp = -temp;
    }

    ftos(number, temp);

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

case 'e':
{
    double temp = va_arg(ap, double);

    if(temp < 0)
    {
        counter += printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
        temp = -temp;
    }

    ftoe(number, temp);

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

(4) %f 将浮点数转换为字符串：
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。 结果保留 6 位小数，代码如下：

```
void ftos(char *numStr, double num){
    if (num == 0.0) {
        numStr[0] = '0';
        numStr[1] = '.';
        numStr[2] = '0';
        numStr[3] = '\0';
        return;
    }
    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';
}
```

(5) %e 将浮点数转换为科学记数法：

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一个变量统计小数点移动的位数。代码如下：

```c
void ftoe(char *numStr, double num){
    if (num == 0.0) {
        numStr[0] = '0';
        numStr[1] = '.';
        numStr[2] = '0';
        numStr[3] = 'e';
        numStr[4] = '+';
        numStr[5] = '0';
        numStr[6] = '\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';
    }
```

```c
    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++]='.';
            }
        }
        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';
    }
}
```

(6) 测试

```c
printf("print percentage: %%\n"
       "print char \"N\": %c\n"
       "print string \"Hello World!\": %s\n"
       "print decimal \"-1234\": %d\n"
       "print hexadecimal \"0x7abcdef0\": %x\n"
       "print float \"3.1415\": %f\n"
       "print float \"3.0\": %f\n"
       "print float \"-0.01\": %f\n"
       "print float \"-4.22\": %f\n"
       "print index_float \"123.6\": %e\n"
       "print index_float \"1.4635\": %e\n"
       "print index_float \"0.00373\": %e\n"
       "print index_float \"-33.66\": %e\n"
       "print index_float \"-0.0654\": %e\n",
       '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);
```

```
make build
make run
```

- 实验结果展示：

```
mafq5@mafq5-virtual-machine:~/lab5/task1/build$ make build
g++ -g -Wall -march=i386 -m32 -nostdlib -fno-builtin -ffreestanding -fno-pic -I.
ld -o kernel.o -melf_i386 -N entry.obj interrupt.o setup.o stdio.o stdlib.o asm_
objcopy -O binary kernel.o kernel.bin
dd if=mbr.bin of=../run/hd.img bs=512 count=1 seek=0 conv=notrunc
记录了1+0 的读入
记录了1+0 的写出
512字节已复制，0.000426066 s，1.2 MB/s
dd if=bootloader.bin of=../run/hd.img bs=512 count=5 seek=1 conv=notrunc
记录了0+1 的读入
记录了0+1 的写出
281字节已复制，0.00122827 s，229 kB/s
dd if=kernel.bin of=../run/hd.img bs=512 count=145 seek=6 conv=notrunc
记录了12+1 的读入
记录了12+1 的写出
6320字节（6.3 kB，6.2 KiB）已复制，0.000218022 s，29.0 MB/s
mafq5@mafq5-virtual-machine:~/lab5/task1/build$ make run
qemu-system-i386 -hda ../run/hd.img -serial null -parallel stdio -no-reboot
WARNING: Image format was not specified for '../run/hd.img' and probing guessed
         Automatically detecting the format is dangerous for raw images, write o
         Specify the 'raw' format explicitly to remove the restrictions.
```

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...
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 index_float "123.6": 3.529e+0
print index_float "1.4635": 3.490e+0
print index_float "0.00373": -1.000000e-2
print index_float "-33.66": -6.000000e-1
print index_float "-0.0654": 1.23599e+2
```

---------- 实验任务 2 ----------

- 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。

- 思路分析：

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。我们在 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。

- 实验步骤：

（1） 首先我们需要在 thread.h 的 PCB 中添加相应的参数

```
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
};
```

（2） 要在 program.h 中声明一个函数 PCB* find_thread_ByPid(int pid)，并在 program.cpp 中实现，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

    // 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);
};

void program_exit();
```

```
PCB *ProgramManager::allocatePCB()
{
    for (int i = 0; i < MAX_PROGRAM_AMOUNT; ++i)
    {
        if (!PCB_SET_STATUS[i])
        {
            PCB_SET_STATUS[i] = true;
            return (PCB *)((int)PCB_SET + PCB_SIZE * i);
        }
    }

    return nullptr;
}
```

(3) 我们需要存储创建时间 createTime，，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();
    }
}
```

(4) 对线程创建函数的修改：

用指针*current 获取当前运行中的进程，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。

对进程退出函数的修改：

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，并将子进程设置为单独进程。

```cpp
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;

    // 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;
    thread->stack[4] = (int)function;
    thread->stack[5] = (int)program_exit;
    thread->stack[6] = (int)parameter;

    allPrograms.push_back(&(thread->tagInAllList));
    readyPrograms.push_back(&(thread->tagInGeneralList));

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

    return thread->pid;
}
```

```
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();
    }
}
```

（5） 编写测试任务

① 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。

② 又以第二个线程为父进程创造了两个子进程

③ 第三个线程创建嵌套子进程的目的，因为 second thread 与 child 处于同等地位。

④ 实现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。

```cpp
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){};
}

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
    int pid = programManager.executeThread(first_thread, nullptr, "first 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();
}
```

```
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){};
}

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){};
}
```

```
make build
make run
```

- 实验结果展示：

```
mafq5@mafq5-virtual-machine:~/lab5/task2/build$ make run
```

QEMU

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: 0
now show the childrenList of the child of thread 1:6
pid 2 name "second child": I am the child thread of 0! ... createtime: 0
now show the childrenList of the child of thread 2:7
pid 3 name "third child": I am the child thread of 0! ... createtime: 0
now show the childrenList of the child of thread 3:8
pid 4 name "second thread": I am the second thread! ... createtime: 0
now show the childrenList of second_thread:9 10
pid 5 name "third thread": I am the third thread! ... createtime: 0
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: 15
4
```

---------- 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 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：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。

- 实验步骤：

(1) 启动 gdb

```
make debug
```

(2) 设置断点

```
break c_time_interrupt_handler
break asm_switch_thread
```

(3) 单步执行

```
stepi  # 单步执行一条指令
nexti  # 单步执行一条指令，但不会进入函数内部
```

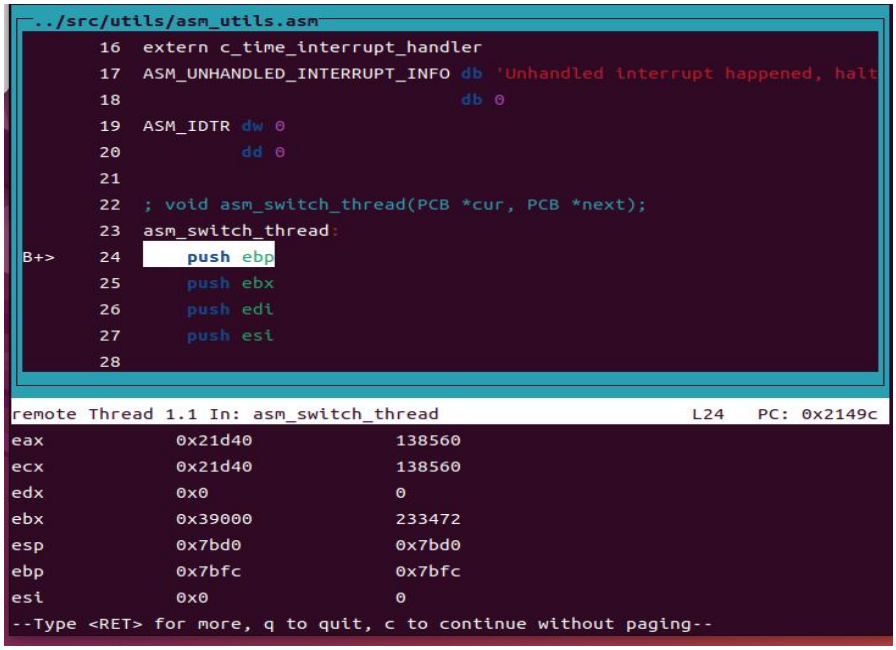
(4) 运行观察

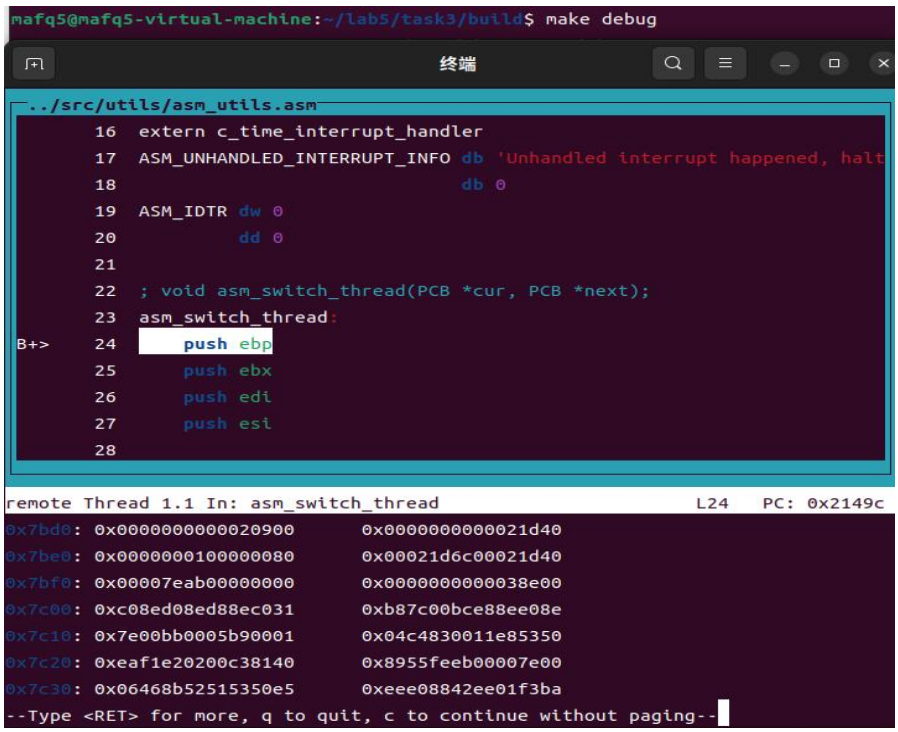
```
Continue
info registers
x/16gx $esp
```

- 实验结果展示：

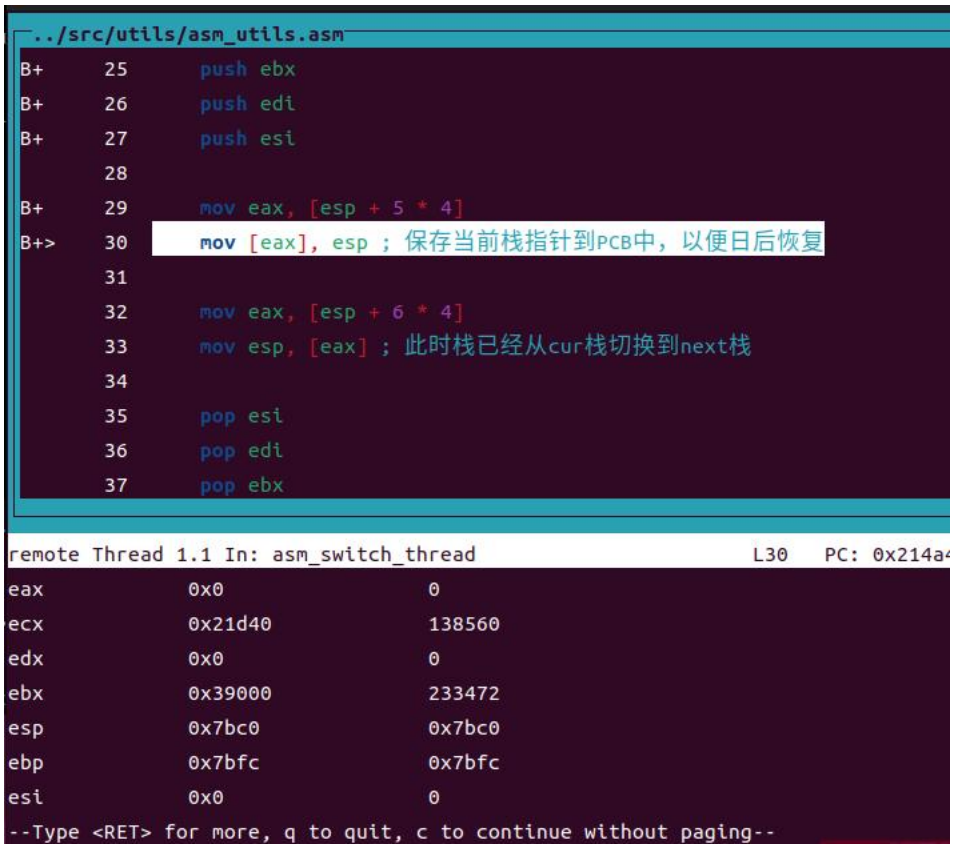
（1）asm_switch_thread

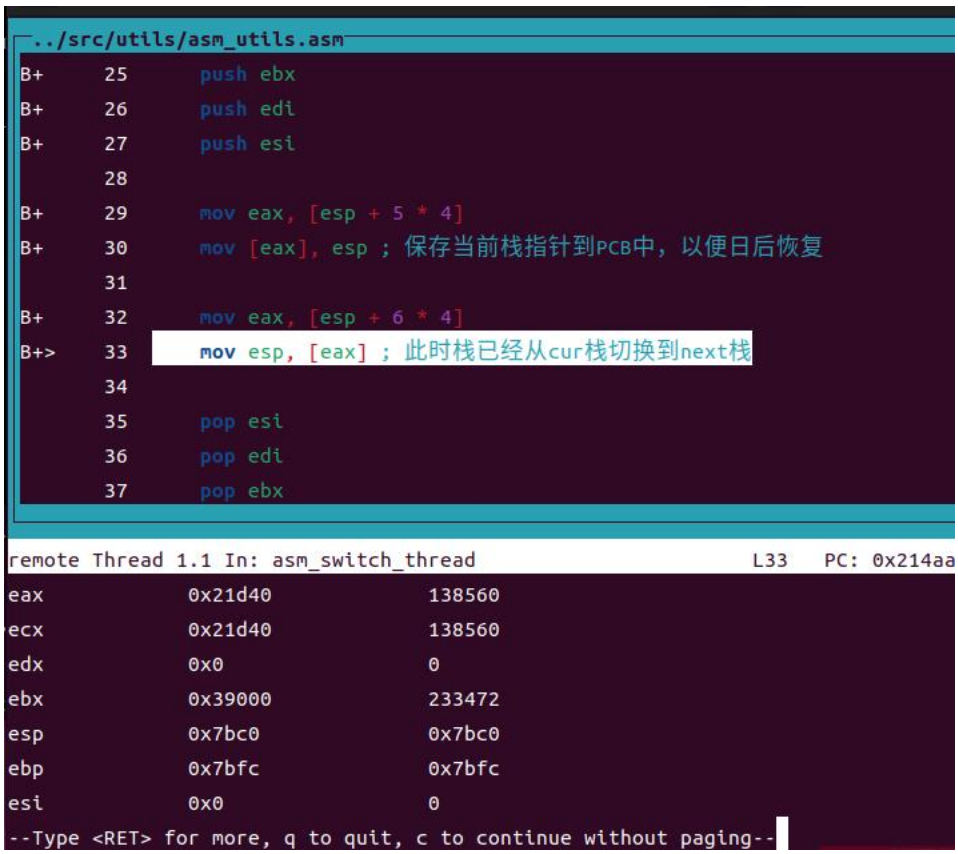
进程切换前寄存器和当前栈指针（$esp）开始的 16 个 8 字节数据显示如下：





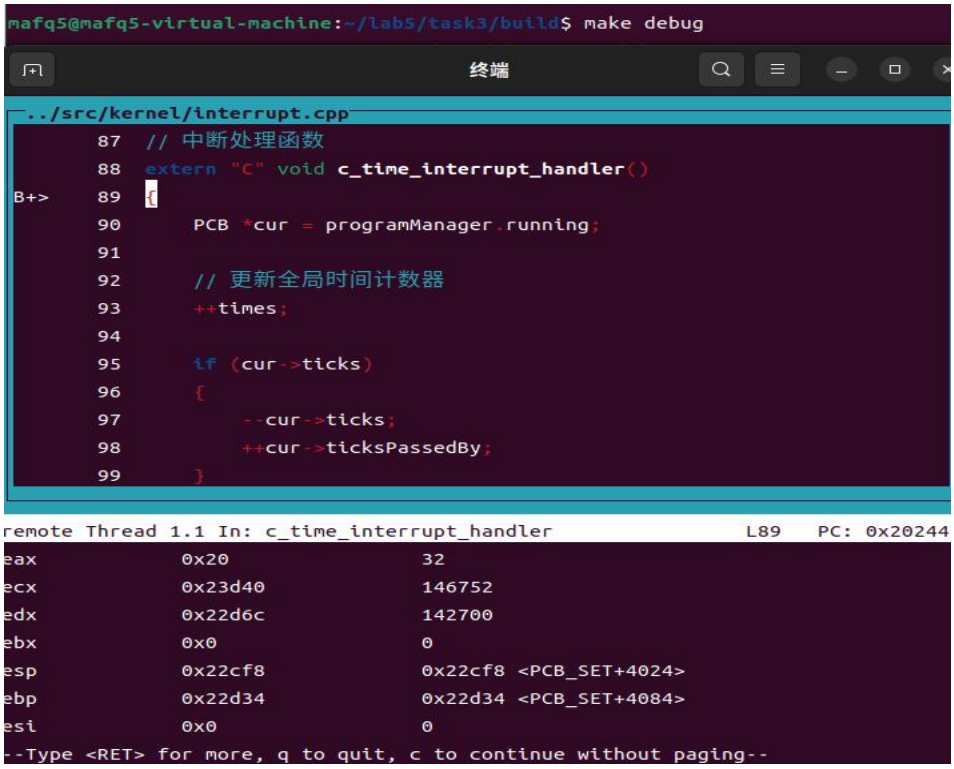
24-30 中的每一行设置断点，并观察寄存器值发现，在第三十行设置断点后，eax 的值变为了 0。而在运行到第三十三行的时候，eax 又会复原：

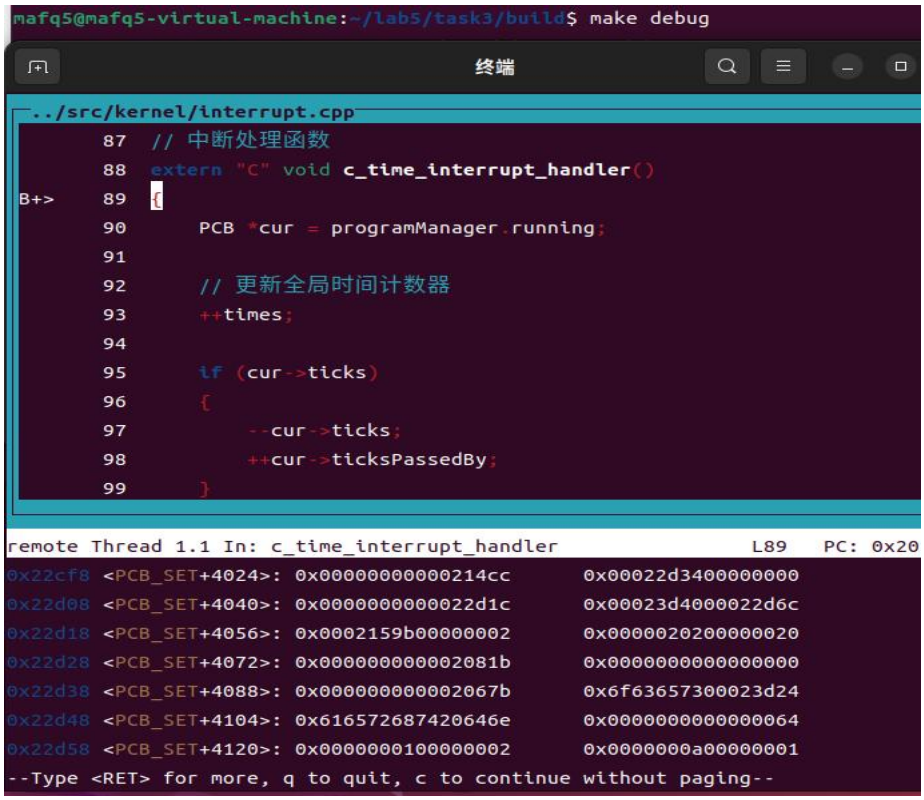

../src/utils/asm_utils.asm
B+ 25 push ebx
B+ 26 push edi
B+ 27 push esi
28
B+ 29 mov eax, [esp + 5 * 4]
B+> 30 mov [eax], esp ; 保存当前栈指针到PCB中，以便日后恢复
31
32 mov eax, [esp + 6 * 4]
33 mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
34
35 pop esi
36 pop edi
37 pop ebx
remote Thread 1.1 In: asm_switch_thread L30 PC: 0x214a
eax 0x0 0
ecx 0x21d40 138560
edx 0x0 0
ebx 0x39000 233472
esp 0x7bc0 0x7bc0
ebp 0x7bfc 0x7bfc
esi 0x0 0
--Type <RET> for more, q to quit, c to continue without paging--


../src/utils/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
eax 0x21d40 138560
ecx 0x21d40 138560
edx 0x0 0
ebx 0x39000 233472
esp 0x7bc0 0x7bc0
ebp 0x7bfc 0x7bfc
esi 0x0 0
--Type <RET> for more, q to quit, c to continue without paging--

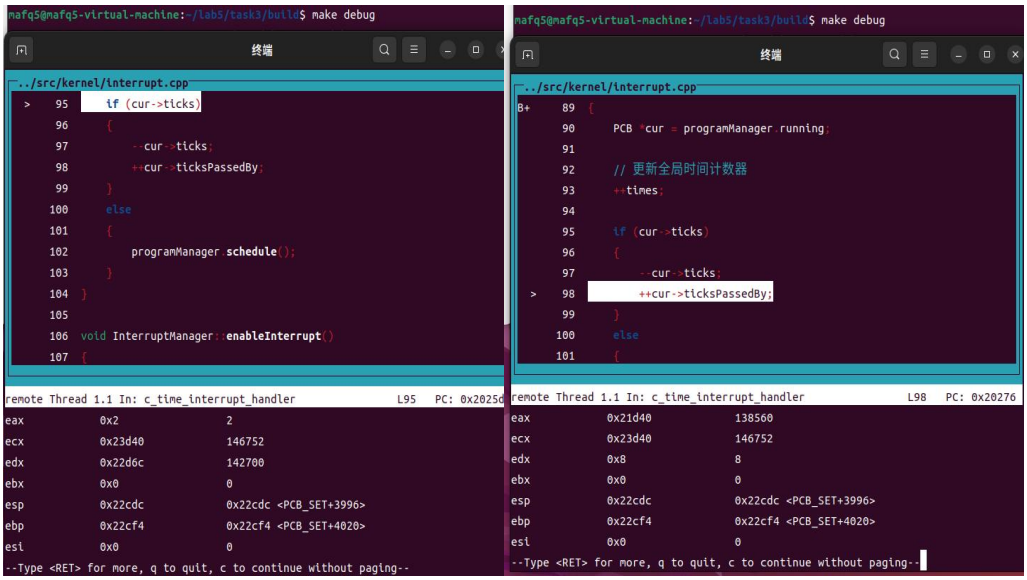
（2） c_time_interrupt_handler

函数执行前寄存器和当前栈指针（$esp）开始的 16 个 8 字节数据显示如下：





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。



总结：通过 GDB 的调试功能，详细观察线程切换的全过程，包括寄存器的保存和恢复、栈指针的切换以及程序计数器的变化。

以下是两个过程的总结：

新线程被调度并开始执行

1. 触发时钟中断，进入 c_time_interrupt_handler 。
2. 调用 schedule ，从就绪队列中取出新线程。
3. 调用 asm_switch_thread ，保存当前线程的上下文，切换栈指针到新线程的栈。
4. 恢复新线程的寄存器值，跳转到新线程的函数入口。

正在执行的线程被中断并换下处理器

1. 触发时钟中断，进入 c_time_interrupt_handler 。
2. 调用 schedule ，将当前线程的状态设置为就绪态，并重新加入就绪队列。
3. 调用 asm_switch_thread ，保存当前线程的上下文，切换栈指针到下一个线程的栈。
4. 恢复下一个线程的寄存器值，从被中断点继续执行。通过这些步骤，你可以清晰地理解线程调度和切换的机制。

---------- 实验任务 4 ----------

- 任务要求：

在材料中，我们已经学习了如何使用时间片轮转算法来实现线程调度。但线程调度算法不止一种，例如先来先服务，最短作业（进程）优先，响应比最高优先算法，优先级调度算法，多级反馈队列调度算法

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。

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。

- 思路分析：
- 实验步骤：

（1） 在 program.h 和 program.cpp 中声明并定义了函数 setThreadTicks：

```
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);
}

void ProgramManager::releasePCB(PCB *program)
{
    int index = ((int)program - (int)PCB_SET) / PCB_SIZE;
    PCB_SET_STATUS[index] = false;
}
```

（2） 修改内核启动程序：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。在 setup.cpp 中只创建 initialise_thread，并且根据调度算法 的不同，我们需要分初始化它的优先级为最大/最小。

```
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();
}
```

（3） 修改中断处理函数：通过中断处理函数的执行模拟时间，即每中断一次就相当于 1 个时间单位。在 interrupt.cpp 中，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。

在 interrupt 中声明线程：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。但是我们现在只需要实现 fifo, 所以本身就是采用的非抢占式。

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

    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);
    }


    // 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();
    }
}
```

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

```cpp
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);
}
```

（5） 编写测试样例：

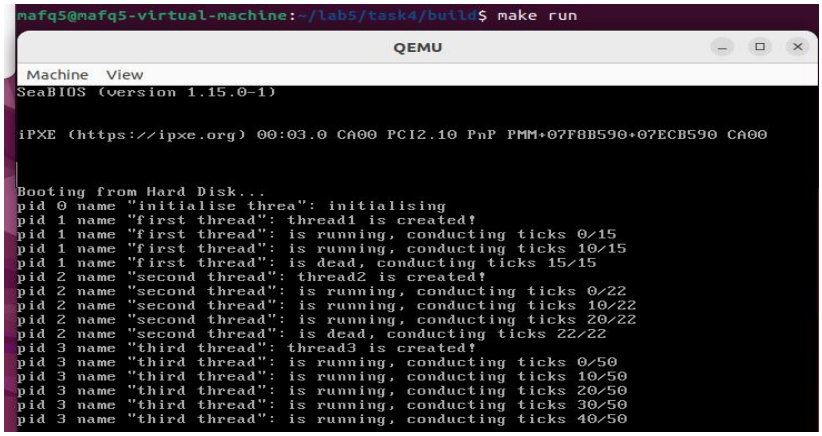
见第三步编写中断函数代码，已经呈现

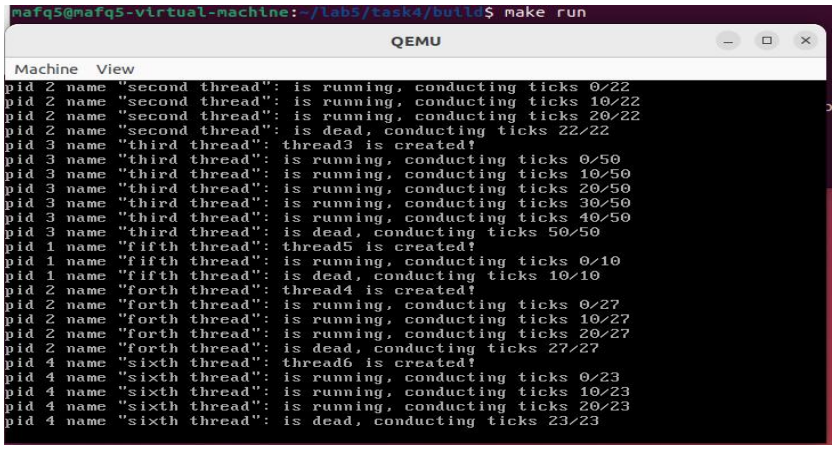
```
make build
make run
```

- 实验结果展示：

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。

程序运行结果如下：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





Section 5 实验总结与心得体会

（1）注意

① 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。

② 后续实现其他调度算法有一点需要注意，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。但是我们现在只需要实现 fifo，所以影响不大。

（2）调度算法的选择应基于应用场景的需求• 非抢占式 FIFO：适用于任务运行时间较短、对实时性要求不高的场景。例如，在简单的批处理系统中，任务可以按顺序依次完成，无需频繁中断，这种调度方式简单高效，减少了上下文切换的开销。• 抢占式 FIFO：适用于对实时性要求较高的场景，如嵌入式系统或需要快速响应的多任务环境。通过引入时间片轮转或优先级抢占机制，可以确保高优先级任务或实时任务能够及时获得处理器资源，从而提高系统的响应性和效率。

（3）抢占式调度虽然高效但也增加了复杂性• 抢占式调度虽然能够更好地满足实时性需求，但其实现比非抢占式调度复杂得多。它需要处理任务的中断和恢复逻辑，频繁的上下文切换可能会增加系统开销。因此，在选择调度算法时，需要权衡实时性和系统复杂度之间的关系，根据实际需求做出合理的选择。

Section 6 附录：参考资料清单

1. 部分代码 debug 与指令除了参考实验指导书，还参考了大语言模型

2. 参考文章，博客部分如下：

```
[https://blog.csdn.net/brainkick/article/details/7583727]
https://zhuanlan.zhihu.com/p/97071815
```