lab5

1、实验要求

Assignment 1 printf的实现

学习可变参数机制，然后实现printf，你可以在材料中的printf上进行改进，或者从头开始实现自己的printf函数。结果截图并说说你是怎么做的。

Assignment 2 线程的实现

自行设计PCB，可以添加更多的属性，如优先级等，然后根据你的PCB来实现线程，演示执行结果。

Assignment 3 线程调度切换

操作系统的线程能够并发执行的秘密在于我们需要中断线程的执行，保存当前线程的状态，然后调度下一个线程上处理机，最后使被调度上处理机的线程从之前被中断点处恢复执行。现在，同学们可以亲手揭开这个秘密。

编写若干个线程函数，使用gdb跟踪 `c_time_interrupt_handler` 、 `asm_switch_thread` 等函数，观察线程切换前后栈、寄存器、PC等变化，结合gdb、材料中“线程的调度”的内容来跟踪并说明下面两个过程。

- 一个新创建的线程是如何被调度然后开始执行的。
- 一个正在执行的线程是如何被中断然后被换下处理器的，以及换上处理机后又是如何从被中断点开始执行的。

通过上面这个练习，同学们应该能够进一步理解操作系统是如何实现线程的并发执行的。

Assignment 4 调度算法的实现

在材料中，我们已经学习了如何使用时间片轮转算法来实现线程调度。但线程调度算法不止一种，例如

- 先来先服务。
- 最短作业（进程）优先。
- 响应比最高者优先算法。
- 优先级调度算法。
- 多级反馈队列调度算法。

此外，我们的调度算法还可以是抢占式的。

现在，需要将线程调度算法修改为上面提到的算法或者是自己设计的算法。然后，自行编写测试样例来呈现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。

2、实验过程

Assignment 1 printf的实现

可变参数的机制

printf的函数参数并不是固定的，这类函数称为可变参数函数。但在平时的编程当中，我们都需要在函数头清晰地指出函数所需要的参数。

C语言允许我们定义可变参数函数，此时函数的参数列表分为两部分，前一部分是固定下来的参数，如 `int` ， `char *` ， `double` 等用数据类型写出来的参数，称为“固定参数”；后一部分是“可变参数”，其使用 `...` 来表示。

例如，声明一个 `printf` 函数

```
int printf(const char* const _Format, ...);
//参数分为两部分，字符串指针(需要格式化输出的字符串)+可变参数。
```

可变参数实现需要遵循相关规则

1. 对于可变参数函数，参数列表中至少有一个固定参数。
2. 可变参数列表必须放在形参列表最后，也就是说， `...` 必须放在函数的参数列表的最后，并且最多只有一个 `...` 。

在函数内部引用可变参数列表中的参数

为了引用可变参数列表中的参数，需要用到 `<stdarg.h>` 头文件定义的一个变量类型 `va_list` 和三个宏 `va_start`， `va_arg`， `va_end`，这三个宏用于获取可变参数列表中的参数。

用法如下：

```
// 定义一个指向可变参数列表的指针。
va_list

// 初始化可变参数列表指针 ap，使其指向可变参数列表的起始位置，
// 即函数的固定参数列表的最后一个参数 last_arg 的后面第一个参数。
va_start(va_list ap, last_arg)

// 以类型 type 返回可变参数，并使 ap 指向下一个参数。
va_arg(va_list ap, type)

// 清零 ap。
va_end(va_list ap)
```

- 可变参数必须从头到尾逐个访问。如果在访问了几个可变参数之后想半途中止，这是可以的。但是，如果想一开始就访问参数列表中间的参数，那是不行的(可以把想访问的中间参数之前的参数依次读取，但是不使用)。
- 宏是无法直接判断实际存在参数的数量。
- 宏无法判断每个参数的类型，所以在使用 `va_arg` 的时候一定要指定正确的类型。
- 如果在 `va_arg` 中指定了错误的类型，那么将会影响到后面的参数的读取。
- 第一个参数也未必要是可变参数个数，例如 `printf` 的第一个参数就是字符串指针。

教程样例：实现 `print_any_numbersof_integers`，用来输出若干个整数的函数

```
void print_any_number_of_integers(int n, ...);
//n是可变参数的数量，...是可变参数，表示若干个待输出的整数
```

具体实现：

```
void print_any_number_of_integers(int n, ...)
{
    // 定义一个指向可变参数的指针parameter
    va_list parameter;
    // 使用固定参数列表的最后一个参数来初始化parameter
    // parameter指向可变参数列表的第一个参数
    va_start(parameter, n);
    for ( int i = 0; i < n; ++i ) {
        // 引用parameter指向的int参数，并使parameter指向下一个参数
        std::cout << va_arg(parameter, int) << " ";
    }
    // 清零parameter
    va_end(parameter);
    std::cout << std::endl;
}
```

解释说明：

- 定义一个指向可变参数列表的指针 `parameter`，来帮助我们引用可变参数列表的参数。
- 用 `va_start` 来初始化 `parameter`，使其指向可变参数列表的第一个参数。C/C++函数调用规则——在函数调用前，函数的参数会被从右到左依次入栈。
- 使用 `va_arg` 来指定参数的类型后才能引用函数的可变参数。只有到了函数的实现这一步，函数才会知道可变参数放置的是什么内容。
- 栈的增长方式是从高地址向低地址增长的，因此函数的参数从左到右，地址依次增大。
- 固定参数列表的最后一个参数的作用是表明可变参数列表的起始地址，计算方式如下(`va_start` 的实现)
-

$$\text{可变参数列表的起始地址} = \text{固定参数列表的最后一个参数的地址} + \text{这个参数的大小} \tag{1}$$

- 使用 `parameter` 和 `va_arg` 来引用可变参数。可变参数的函数并不知道每一个可变参数的类型和具体含义，它只是在调用前把这些参数放到了栈上。我们人为地在 `<stdarg.h>` 中定义了一些访问栈地址的宏，因此可以指定这些参数的具体类型和使用这些宏来取出参数，这就是访问可变参数的实现思想。
- 从本质上来说， `parameter` 就是指向函数调用栈的一个指针，类似 `esp` 、 `ebp` ， `va_arg` 按照指定的类型来返回 `parameter` 指向的内容。
- 在 `va_arg` 返回后， `parameter` 会指向下一个参数，无需我们手动调整。
- 访问完可变参数后，我们需要使用 `va_end` 对 `parameter` 进行清零，防止后面再使用 `va_arg` 和 `parameter` 来引用栈中的内容，从而导致错误。

测试：

```
int main()
{
    print_any_number_of_integers(1, 213);
    print_any_number_of_integers(2, 234, 2567);
    print_any_number_of_integers(3, 487, -12, 0);
}
```

结果：

```
dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab5/1$ ./main.out
213
234 2567
487 -12 0
```

可变参数机制的实现

三个宏的实现

`va_list` 是指向可变参数列表的指针，是字节类型的指针，而 `char` 类型就是1个字节

```
typedef char * va_list;
```

`va_start` 用于初始化一个指向可变参数列表起始地址的指针 `ap` ，需要用到固定参数列表的最后一个变量 `v`

```
#define _INTSIZEOF(n) ( (sizeof(n)+sizeof(int)-1) & ~(sizeof(int)-1) )
//_INTSIZEOF(n)返回的是n的大小进行4字节对齐的结果
#define va_start(ap,v) ( ap = (va_list)&v + _INTSIZEOF(v) )
```

`va_arg` 的作用是返回 `ap` 指向的， `type` 类型的变量，并同时使 `ap` 指向下一个参数

```
#define va_arg(ap, type) ( *(type *)((ap += _INTSIZEOF(type)) - _INTSIZEOF(type)))
```

`va_end` 的作用是将 `ap` 清零

```
#define va_end(ap) ( ap = (va_list)0 )
```

结合上述样例代码，使用自定义实现的宏来引用可变参数

```
#include <iostream>
typedef char *va_list;
#define _INTSIZEOF(n) ((sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1))
#define va_start(ap, v) (ap = (va_list)&v + _INTSIZEOF(v))
#define va_arg(ap, type) (*(type *)((ap += _INTSIZEOF(type)) - _INTSIZEOF(type)))
#define va_end(ap) (ap = (va_list)0)

void print_any_number_of_integers(int n, ...);

int main()
{
```

```
    print_any_number_of_integers(1, 213);
    print_any_number_of_integers(2, 234, 2567);
    print_any_number_of_integers(3, 487, -12, 0);
}

void print_any_number_of_integers(int n, ...)
{
    // 定义一个指向可变参数的指针parameter
    va_list parameter;
    // 使用固定参数列表的最后一个参数来初始化parameter
    // parameter指向可变参数列表的第一个参数
    va_start(parameter, n);

    for (int i = 0; i < n; ++i)
    {
        // 引用parameter指向的int参数，并使parameter指向下一个参数
        std::cout << va_arg(parameter, int) << " ";
    }
    // 清零parameter
    va_end(parameter);
    std::cout << std::endl;
}
```

编译运行

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

在arm架构下的linux运行以上命令会报错

```
dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab5/2$ g++ main.cpp -m32 -std=c++
98 -o main.out && ./main.out
g++: error: unrecognized command-line option ‘-m32’
```

编写 `Makefile` 文件，使用 `cpio` 打包 `initramfs`， `qemu` 启动内核，并加载 `initramfs`，即可看到 `main` 的输出

```
TARTGET = main
build:
  @i686-linux-gnu-g++ -m32 -g -static -c main.cpp
  @i686-linux-gnu-g++ -m32 -g -static -o $(TARTGET) main.o
  @echo main | cpio -o --format=newc > hwinitramfs
run:
  @qemu-system-i386 -kernel ~/lab1/linux-5.10.172/arch/x86/boot/bzImage -initrd hwinitramfs -append
"console=ttyS0 rdinit=main" -nographic
clean:
  @rm -rf *.o
```

```
dengxy@dengxy-Parallels-ARM-Virtual-Machine: ~/lab5/2
[    1.568592] Freeing unused kernel image (initmem) memory: 680K
[    1.572412] Write protecting kernel text and read-only data: 15604k
[    1.572758] Run main as init process
213
234 2567
487 -12 0
```

格式化输出实现printf

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

在格式化输出字符串中，会包含 `%c,%d,%x,%s` 等来实现格式化输出，对应的参数在可变参数中可以找到。

fmt的解析: printf首先找到fmt中的形如 `%c,%d,%x,%s` 对应的参数，然后用这些参数具体的值来替换 `%c,%d,%x,%s` 等，得到一个新的格式化输出字符串，printf将这个新的格式化输出字符串即可。但是这个字符串可能会超过函数调用栈的大小。因此，我们需要定义一个缓冲区，然后对fmt进行逐字符地解析，将结果逐字符的放到缓冲区中。放入一个字符后，我们会检查缓冲区，如果缓冲区已满，则将其输出，然后清空缓冲区，否则不做处理。

处理字符串中的 `\n` 换行字符：滚屏操作

```
int STDIO::print(const char *const str);
```

`printf` 的实现：

定义一个大小为 `BUF_LEN` 的缓冲区 `buffer` ， `buffer` 多出来的1个字符是用来放置 `\0` 的。由于我们后面会将一个整数转化为字符串表示， `number` 使用来存放转换后的数字字符串。由于保护模式是运行在32位环境下的，最大的数字字符串也不会超过32位，因此number分配33个字节也就足够了。

```
    const int BUF_LEN = 32;
    char buffer[BUF_LEN + 1];
    char number[33];
```

`printf_add_to_buffer` 的实现：

对 `fmt` 进行逐字符解析，对于每一个字符 `fmt[i]` ，如果 `fmt[i]` 不是 `%` ，则说明是普通字符，直接放到缓冲区即可。将 `fmt[i]` 放到缓冲区后可能会使缓冲区变满，此时如果缓冲区满，则将缓冲区输出并清空。

```
int printf_add_to_buffer(char *buffer, char c, int &idx, const int BUF_LEN)
{
    int counter = 0;
    buffer[idx] = c;
    ++idx;
    if (idx == BUF_LEN)
    {
        buffer[idx] = '\0';
        counter = stdio.print(buffer);
        idx = 0;
    }
    return counter;
}
```

如果 `fmt[i]` 是 `%` ，则说明这可能是一个格式化输出的参数，需要检查 `%` 后面的参数

- `%%` ：输出一个 `%` 。
- `%c` ：输出 `ap` 指向的字符。
- `%s` ：输出 `ap` 指向的字符串的地址对应的字符串。
- `%d` ：输出 `ap` 指向的数字对应的十进制表示。
- `%x` ：输出 `ap` 指向的数字对应的16进制表示。
- 其他：不做处理。

特别地，对于 `%d` 和 `%x` ，将数字转换为对应的字符串，实现 `itos` 函数

`itos` 转换的是非负整数，对于 `%d` 的情况，如果我们输出的整数是负数，那么就要使用 `itos` 转换其相反数，在输出数字字符串前输出一个负号

```
/*
 * 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);
```

`printf` 完整代码如下：

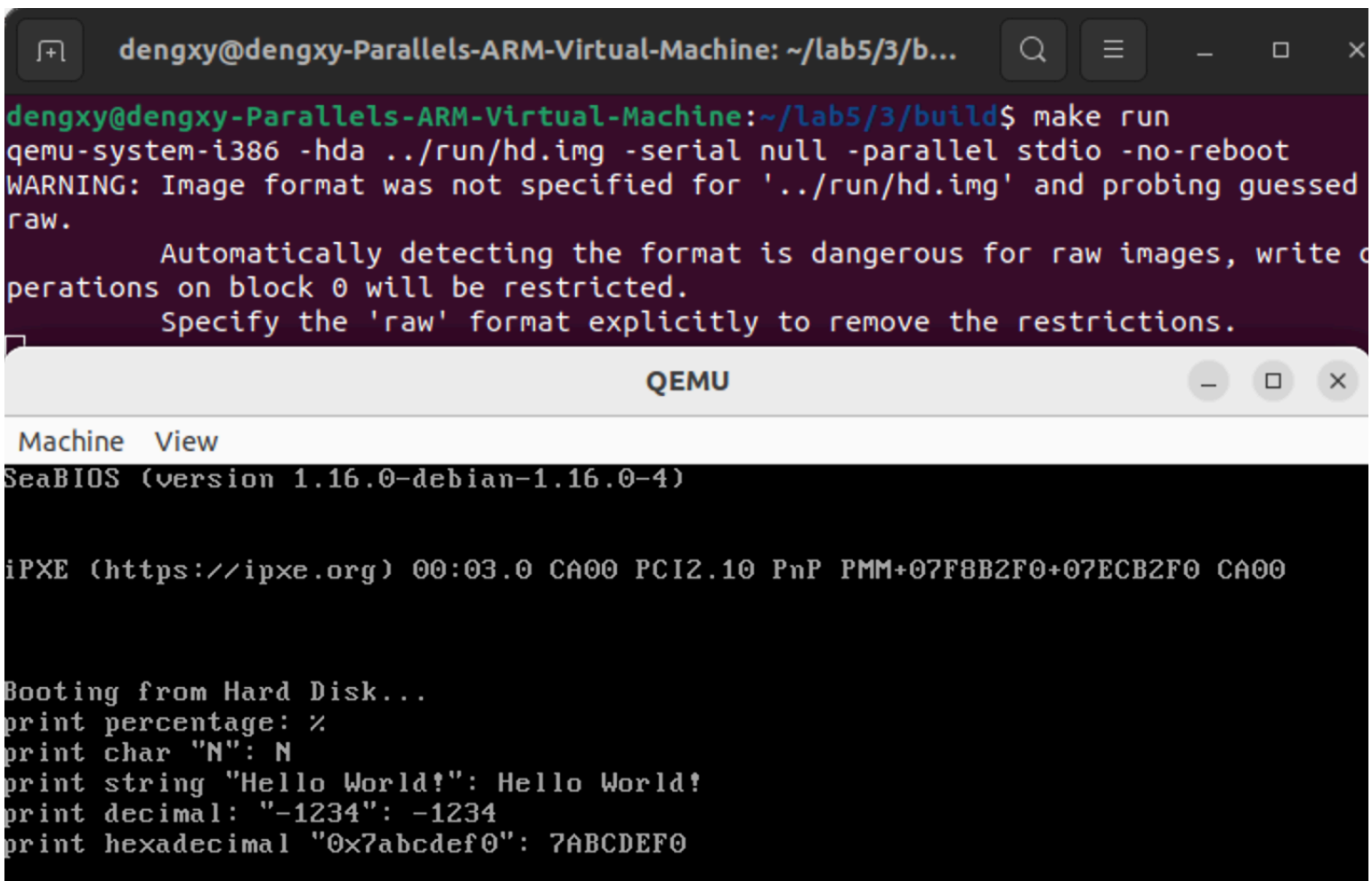
```
int printf(const char *const fmt, ...)
{
```

```
    const int BUF_LEN = 32;
    char buffer[BUF_LEN + 1];
    char number[33];

    int idx, counter;
    va_list ap;

    va_start(ap, fmt);
    idx = 0;
    counter = 0;

    for (int i = 0; fmt[i]; ++i)
    {
        if (fmt[i] != '%')
        {
            counter += printf_add_to_buffer(buffer, fmt[i], idx, BUF_LEN);
        }
        else
        {
            i++;
            if (fmt[i] == '\0')
            {
                break;
            }
            switch (fmt[i])
            {
            case '%':
                counter += printf_add_to_buffer(buffer, fmt[i], idx, BUF_LEN);
                break;

            case 'c':
                counter += printf_add_to_buffer(buffer, va_arg(ap, char), idx, BUF_LEN);
                break;

            case 's':
                buffer[idx] = '\0';
                idx = 0;
                counter += stdio.print(buffer);
                counter += stdio.print(va_arg(ap, const char *));
                break;

            case 'd':
            case 'x':
                int temp = va_arg(ap, int);

                if (temp < 0 && fmt[i] == 'd')
                {
                    counter += printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
                    temp = -temp;
                }
                itos(number, temp, (fmt[i] == 'd' ? 10 : 16));
                for (int j = 0; number[j]; ++j)
                {
                    counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
                }
                break;
            }
        }
    }
    buffer[idx] = '\0';
    counter += stdio.print(buffer);
    return counter;
}
```

测试 `printf`：在 `setup.cpp` 中修改 `setup_kernel` 函数

```
#include "asm_utils.h"
#include "interrupt.h"
#include "stdio.h"

// 屏幕IO处理器
STDIO stdio;
// 中断管理器
InterruptManager interruptManager;
extern "C" void setup_kernel()
{
    // 中断处理部件
    interruptManager.initialize();
    // 屏幕IO处理部件
    stdio.initialize();
    interruptManager.enableTimeInterrupt();
    interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);
    //asm_enable_interrupt();
    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",
           'N', "Hello World!", -1234, 0x7abcdef0);
    //uint a = 1 / 0;
    asm_halt();
}
```

编译运行



自定义实现 `printf` 部分详见实验结果和关键代码的Assignment 1部分

Assignment 2 线程的实现

线程的介绍

线程的状态有5个，分别是创建态、运行态、就绪态、阻塞态和终止态。

用 `ProgramStatus` 来描述线程的5个状态

```
enum ProgramStatus
{
    CREATED,//创建
    RUNNING,//运行
    READY,//就绪
    BLOCKED,//阻塞
    DEAD//终止
};
```

线程的组成部分，包括线程栈、状态、优先级、运行时间、线程负责运行的函数和函数的参数等，集中保存在PCB(Process Control Block)

```
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
};
```

- `stack`：在线程被换下处理器时保存esp的内容，当线程被换上处理器后，用`stack`去替换esp的内容，从而实现恢复线程运行的效果
- `status`：线程的状态
- `name`：线程的名称
- `priority`：线程的优先级，决定了抢占式调度的过程和线程的执行时间
- `pid`：线程的标识符，每个线程的pid是唯一的
- `ticks`：线程剩余的执行次数。在时间片调度算法中，每发生中断一次记为一个`tick`，当ticks=0时，线程会被换下处理器，并将其他线程换上处理器执行。
- `ticksPassedBy`：线程总共执行的`tick`的次数
- `tagInGeneralList`和`tagInAllList`：线程在线程队列中的标识，用于在线程队列中找到线程的PCB
- 声明一个程序管理类`ProgramManager`用于线程和进程的创建和管理

PCB的分配

将一个PCB的大小设置为4096个字节，即一个页的大小。

```
// PCB的大小，4KB。
const int PCB_SIZE = 4096;
// 存放PCB的数组，预留了MAX_PROGRAM_AMOUNT个PCB的大小空间。
char PCB_SET[PCB_SIZE * MAX_PROGRAM_AMOUNT];
// PCB的分配状态，true表示已经分配，false表示未分配。
bool PCB_SET_STATUS[MAX_PROGRAM_AMOUNT];
```

在`ProgramManager`中声明两个管理PCB所在的内存空间函数。

```
// 分配一个PCB
PCB *allocatePCB();
// 归还一个PCB
void releasePCB(PCB *program);
```

PCB分配`allocatePCB`的实现：

```
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;
}
```

检查 `PCB_SET` 中每一个PCB的状态，如果找到一个未被分配的PCB，则返回这个PCB的起始地址

> `PCB_SET` 中的PCB是连续存放的，对于第i个PCB， `PCB_SET` 的首地址加上i×PCBSIZE就是第i个PCB的起始地址。

PCB的状态保存在 `PCB_SET_STATUS` 中，并且 `PCB_SET_STATUS` 的每一项会在 `ProgramManager` 总被初始化为 `false` ，表示所有的PCB都未被分配。被分配的PCB用 `true` 来标识。

如果 `PCB_SET_STATUS` 的所有元素都是 `true` ，表示所有的PCB都已经被分配，此时应该返回 `nullptr` ，表示PCB分配失败。

PCB的释放 `releasePCB` 的实现：

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

接受一个PCB指针 `program` ，然后计算出 `program` 指向的PCB在 `PCB_SET` 中的位置，然后将 `PCB_SET_STATUS` 中的对应位置设置 `false`

线程的创建

线程实际上执行的是某一个函数的代码，但并不是所有的函数都可以放入到线程中执行的。

规定线程只能执行返回值为 `void` ，参数为 `void *` 的函数。其中， `void *` 指向了函数的参数。将这个函数定义为 `ThreadFunction` 。

```
typedef void(*ThreadFunction)(void *);
```

并在 `ProgramManager` 中声明一个用于创建线程的函数 `executeThread`

```
// 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);
```

`executeThread` 的实现：

在时钟中断发生时来进行线程调度的，关中断后，时钟中断无法被响应，线程就无法被调度，直到再次开中断。只要线程无法被调度，那么线程的工作也就无法被其他线程打断，因此实现了线程互斥。

```
    // 关中断，防止创建线程的过程被打断
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();
```

相应地，在函数返回前，回复中断

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

关中断后，向 `PCB_SET` 申请一个线程的PCB

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

后面使用 `memeset` 将PCB清0

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

设置PCB的成员 `name` 、 `status` 、 `priority` 、 `ticks` 、 `ticksPassedBy` 和 `pid` 。线程初始的 `ticks` 设置为 `10` 倍的 `priority` ， `pid` 使用PCB在 `PCB_SET` 的位置来代替。

```
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;
```

初始化线程的栈：初始地址是PCB的起始地址加上 `PCB_SIZE` 。

```
thread->stack = (int *)((int)thread + PCB_SIZE);
```

在栈中放入7个整数值：

- 4个为0的值是放到ebp，ebx，edi，esi中。
- `thread->stack[4]` ：线程执行的函数的起始地址。
- `thread->stack[5]` ：线程的返回地址，所有的线程执行完毕后都会返回到这个地址。
- `thread->stack[6]` ：线程的参数的地址。

```
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;
```

创建完线程的PCB后，将其放入到 `allPrograms` 和 `readyPrograms` 中，等待时钟中断来的时候，这个新创建的线程就被调度上处理器。

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

恢复中断后，返回线程的pid

```
return thread->pid;
```

`make && make run` 完成第一个线程的实现

```
SeaBIOS (version 1.16.0-debian-1.16.0-4)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00



Booting from Hard Disk...
pid 0 name "first thread": assignment 2
```

自行设计PCB实现线程

PCB的基本结构，其中包括了优先级的属性

```
typedef struct
{
    int id;
    int priority;//优先级越高，数字越小
    State state;
} PCB;
```

Thread的数据结构设计， `Run` 为实际调用的函数

```
typedef struct
{
    void (*Run)(void*);
    PCB pcb;
} Thread;
```

`thread_create` 线程创建函数的实现

```
void thread_create(Thread* t, void (*Run)(void*), int id, int priority)
{
    t->Run = Run;
    t->pcb.id = id;
    t->pcb.priority = priority;
    t->pcb.state = CREATED;//初始状态为CREATED
    threads[num_threads] = t;//将线程存入线程数组
    num_threads++;//总线程数量+1
}
```

`thread_start` 和 `thread_start_execute` 函数的实现

```
void thread_start(Thread* t)
{
    t->pcb.state = RUNNING;//修改状态为RUNNING
    t->Run(t);
    t->pcb.state = DEAD;//执行完毕，修改状态为DEAD
}
void thread_start_execute(Thread* t)
{
    t->pcb.state = READY;//修改状态为READY后开始执行
    thread_start(t);
}
```

`thread_dead` 线程终止函数的实现：

```
void thread_dead(Thread* t)
{
    t->pcb.state = DEAD;//修改状态为DEAD，终止
}
```

`sort_threads` 根据每个线程的 `priority` 对线程进行排序， `priority` 越小的优先级越高

```
void sort_threads()
{
    int i, j;
    Thread* temp;
    for (i = 0; i < num_threads - 1; i++)
    {
        for (j = i + 1; j < num_threads; j++)
        {
            if (threads[i]->pcb.priority > threads[j]->pcb.priority)
            {
                temp = threads[i];
                threads[i] = threads[j];
```

```
                threads[j] = temp;
            }
        }
    }
}
```

`get_next_thread` 获取下一个要执行的线程

```
Thread* get_next_thread()
{
    int i;
    Thread* next = NULL;
    for (i = 0; i < num_threads; i++)
    {
        if (threads[i]->pcb.state == CREATED)
        {
            next = threads[i];
            break;
        }
    }
    return next;
}
```

`my_thread_run` 函数的实现

```
void my_thread_run(void* arg)
{
    Thread* t = (Thread*) arg;
    //打印信息
    printf("Thread %d is running\n", t->pcb.id);
    printf("Thread %d is finished\n", t->pcb.id);
    thread_dead(t);//终止线程
}
```

编写 `main` 函数来测试

```
int main()
{
    Thread t1, t2, t3;
    //创建三个进程，分别赋予不同的优先级
    thread_create(&t1, my_thread_run, 1, 3);
    thread_create(&t2, my_thread_run, 2, 2);
    thread_create(&t3, my_thread_run, 3, 1);
    while (1)
    {
        sort_threads();
        Thread* next = get_next_thread();
        if (next)
        {
            thread_start_execute(next);//执行每个线程
        }
        else
        {
            break;
        }
    }
    return 0;
}
```

输出结果：

```
Thread 3 is running
Thread 3 is finished
Thread 2 is running
Thread 2 is finished
Thread 1 is running
Thread 1 is finished
```

符合预期

Assignment 3 线程调度切换

线程的调度

在`ProgramManager`中放入成员`running`，表示当前在处理机上执行的线程的PCB。

```
PCB *running;       // 当前执行的线程
```

样例实现了时间片轮转算法（Round Robin, RR）

线程调度函数`ProgramManager::schedule`的实现：当时钟中断到来时，对当前线程的`ticks`减1，直到`ticks`等于0，然后执行线程调度

实现线程互斥，在进程线程调度前，关中断，退出时恢复中断。

```
interruptManager.disableInterrupt();
```

判断当前可调度的线程数量，如果`readyProgram`为空，说明当前系统中只有一个线程，无需进行调度，直接返回即可。

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

判断当前线程的状态，如果是运行态(RUNNING)，则重新初始化其状态为就绪态(READY)和`ticks`，并放入就绪队列

```
    if (running->status == ProgramStatus::RUNNING)
    {
        running->status = ProgramStatus::READY;
        running->ticks = running->priority * 10;
        readyPrograms.push_back(&(running->tagInGeneralList));
    }
```

如果是终止态(DEAD)，则回收线程的PCB

```
    else if (running->status == ProgramStatus::DEAD)
    {
        releasePCB(running);
    }
```

就绪队列的第一个线程作为下一个执行的线程。

```
ListItem *item = readyPrograms.front();
```

就绪队列的第一个元素是`ListItem *`类型的，需要将其转换为`PCB`。

放入就绪队列`readyPrograms`的是每一个PCB的`&tagInGeneralList`，而`tagInGeneralList`在PCB中的偏移地址是固定的。

```
PCB *next = ListItem2PCB(item, tagInGeneralList);
```

将`item`的值减去`tagInGeneralList`在PCB中的偏移地址就得到PCB的起始地址，该操作使用宏定义，其中`(int)&((PCB *)0)->LIST_ITEM`是`LIST_ITEM`在PCB中的偏移地址

```
#define ListItem2PCB(ADDRESS, LIST_ITEM) ((PCB *)((int)(ADDRESS) - (int)&((PCB *)0)->LIST_ITEM))
```

从就绪队列中删去第一个线程，设置其状态为运行态和当前正在执行的线程。

```
    running = next;
    readyPrograms.pop_front();
```

将线程从 `cur` 切换到 `next`

```
asm_switch_thread(cur, next);
```

`asm_switch_thread` 的实现：

保存寄存器 `ebp` 、 `ebx` 、 `edi` 、 `esi`

```
    push ebp
    push ebx
    push edi
    push esi
```

保存esp的值到线程的 `PCB::stack` 中，用做下次恢复。

`PCB::stack` 在 `PCB` 的偏移地址是0。因此，首先将 `cur->stack` 的地址放到 `eax` 中，再向 `[eax]` 中写入 `esp` 的值，也就是向 `cur->stack` 中写入esp

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

将 `next->stack` 的值写入到esp中，从而完成线程栈的切换。

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

`pop` 语句将4个0值放到 `esi` ， `edi` ， `ebx` ， `ebp` 中。

```
    pop esi
    pop edi
    pop ebx
    pop ebp

    sti
    ret
```

完成线程栈的切换后， `next` 指向的线程有两种状态

1、刚创建还未调度运行

此时栈顶的数据是线程需要执行的函数的地址 `function` 。执行ret返回后， `function` 会被加载进eip，从而使得CPU跳转到这个函数中执行。此时进入函数后，函数的栈顶是函数的返回地址，返回地址之上是函数的参数，符合函数的调用规则。而函数执行完成时，其执行ret指令后会跳转到返回地址 `program_exit` 。

`program_exit` 的实现：

将返回的线程的状态置为DEAD，然后调度下一个可执行的线程上处理器。在这里规定第一个线程是不可以返回的，这个线程的pid为0。

```
void program_exit()
{
    PCB *thread = programManager.running;
    //将返回的线程的状态置为DEAD
    thread->status = ThreadStatus::DEAD;
    //调度下一个可执行的线程上处理器
    if (thread->pid)
    {
        programManager.schedule();
    }
    else
    {
        interruptManager.disableInterrupt();
        printf("halt\n");
```

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

2、之前被换下处理器现在又被调度

执行4个 `pop` 后，之前保存在线程栈中的内容会被恢复到这4个寄存器中，然后执行ret后会返回调用 `asm_switch_thread` 的函数，也就是 `ProgramManager::schedule` ，然后在 `ProgramManager::schedule` 中恢复中断状态，返回到时钟中断处理函数，最后从时钟中断中返回，恢复到线程被中断的地方继续执行。

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。

在 `setup.cpp` 中编写相关的线程函数

```
void third_thread(void *arg) {
    printf("pid %d name \"%s\": assignment3.3\n", programManager.running->pid, programManager.running->name);
    while(1) {
    }
}
```

```
void second_thread(void *arg) {
    printf("pid %d name \"%s\": assignment3.2\n", programManager.running->pid, programManager.running->name);
}
```

```
void first_thread(void *arg)
{
    // 第1个线程不可以返回
    printf("pid %d name \"%s\": assignment3.1\n", programManager.running->pid, programManager.running->name);
    asm_halt();
}
```

创建三个线程

```
    int pid1 = programManager.executeThread(first_thread, nullptr, "first thread", 1);
    if (pid1 == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }
    int pid2 = programManager.executeThread(second_thread, nullptr, "second thread", 2);
    if (pid2 == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }
    int pid3 = programManager.executeThread(third_thread, nullptr, "third thread", 3);
    if (pid3 == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }
```

用gdb查看寄存器的值

在 `c_time_interrupt_handler` 设置断点

```
../src/kernel/interrupt.cpp
        84      setInterruptDescriptor(IRQ0_8259A_MASTER, (uint32)handler, 0);
        85  }
        86
        87  // 中断处理函数
        88  extern "C" void c_time_interrupt_handler()
B+>     89  {
        90      PCB *cur = programManager.running;
        91
        92      if (cur->ticks)
        93      {
        94          --cur->ticks;
        95          ++cur->ticksPassedBy;
remote Thread 1.1 In: c_time_interrupt_handler                L89   PC: 0x20244
eax            0x20                32
ecx            0x23da0             146848
edx            0x11                17
ebx            0x0                 0
esp            0x22d58             0x22d58 <PCB_SET+4024>
ebp            0x22d94             0x22d94 <PCB_SET+4084>
esi            0x0                 0
```

寄存器 `esp` 的值为PCB_SET+4024

在 `thread_Execute` 中有

```
    thread->stack = (int *)((int)thread + PCB_SIZE);
    thread->stack -= 7;
```

`PCB_SET` 中的PCB是连续存放的，对于第i个PCB， `PCB_SET` 的首地址加上i×PCB_SIZE就是第i个PCB的起始地址。

在 `asm_switch_thread` 设置断点

```
../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
        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: 0x214bc
eax            0x21da0             138656
ecx            0x23da0             146848
edx            0x0                 0
ebx            0x39000             233472
esp            0x7bc0              0x7bc0
ebp            0x7bfc              0x7bfc
esi            0x0                 0
```

`esp` =0x7bc0

`ebp` =0x7bfc

```
(gdb) x /20x 0x7bb0
0x7bb0: 0x000208bf      0x00031dc0      0x00021dcc      0x00007bfc
0x7bc0: 0x0002092b      0x00000000      0x00021da0      0x00000000
0x7bd0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7be0: 0x00021da0      0x00021dcc      0x00000002      0x00000001
0x7bf0: 0x00000000      0x00007eab      0x00038e00      0x00000000
```

在0x7bcc的地址中，有 `ebp` =0x7bfc

continue

```
../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
        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: 0x214bc
eax            0x22dcc             142796
ecx            0x1                 1
edx            0x0                 0
ebx            0x0                 0
esp            0x22cf8             0x22cf8 <PCB_SET+3928>
ebp            0x22d24             0x22d24 <PCB_SET+3972>
esi            0x0                 0
```

`esp` =0x22cf8

`ebp` =0x22d24

```
(gdb) x /20x 0x22ce8
0x22ce8 <PCB_SET+3912>: 0x000202a9      0x00000002      0x00022dcc      0x00022d24
0x22cf8 <PCB_SET+3928>: 0x00020659      0x00021da0      0x00022da0      0x00022d0c
0x22d08 <PCB_SET+3944>: 0x00000020      0x00021da0      0x00022da0      0x00022dcc
0x22d18 <PCB_SET+3960>: 0x00000000      0x00000000      0x00000000      0x00022d54
0x22d28 <PCB_SET+3976>: 0x00020289      0x00031dc0      0x22000000      0x7361203a
```

在PCB_SET+3924的地址中，有 `ebp` =0x22d24

continue

```
../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
        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
        32      mov eax, [esp + 6 * 4]
```

```
remote Thread 1.1 In: asm_switch_thread                    L24   PC: 0x214bc
eax            0x23dcc             146892
ecx            0x23d12             146706
edx            0x0                 0
ebx            0x0                 0
esp            0x23d3c             0x23d3c <PCB_SET+8092>
ebp            0x23d68             0x23d68 <PCB_SET+8136>
esi            0x0                 0
```

`esp` =0x23d3c

`ebp` =0x23d68

```
(gdb) x /20x 0x23d2c
0x23d2c <PCB_SET+8076>: 0x000202a9      0x00000002      0x00023dcc      0x00023d68
0x23d3c <PCB_SET+8092>: 0x00020659      0x00022da0      0x00023da0      0x00000000
0x23d4c <PCB_SET+8108>: 0x00000000      0x00022da0      0x00023da0      0x00023dcc
0x23d5c <PCB_SET+8124>: 0x01000001      0x00000020      0x00023d88      0x00023d98
0x23d6c <PCB_SET+8140>: 0x000206a2      0x00031dc0      0x00023d94      0x0002079b
```

在PCB_SET+8088的地址中，有 `ebp` =0x23d68

continue

```
../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
        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
        32      mov eax, [esp + 6 * 4]
```

```
remote Thread 1.1 In: asm_switch_thread                    L24   PC: 0x214bc
eax            0x21dcc             138700
ecx            0x3                 3
edx            0x0                 0
ebx            0x0                 0
esp            0x24cfc             0x24cfc <PCB_SET+12124>
ebp            0x24d28             0x24d28 <PCB_SET+12168>
esi            0x0                 0
edi            0x0                 0
```

`esp` =0x24cfc

`ebp` =0x24d28

```
(gdb) x /20x 0x24cec
0x24cec <PCB_SET+12108>:        0x000202a9      0x00000001      0x00021dcc      0x00024d28
0x24cfc <PCB_SET+12124>:        0x00020659      0x00023da0      0x00021da0      0x00000020
0x24d0c <PCB_SET+12140>:        0x00000011      0x00023da0      0x00021da0      0x00021dcc
0x24d1c <PCB_SET+12156>:        0x00000000      0x00000000      0x00000000      0x00024d58
0x24d2c <PCB_SET+12172>:        0x00020289      0x00031dc0      0x7361203a      0x6e676973
```

在PCB_SET+12120的地址中，有 `ebp` =0x24d28

Assignment 4

实现了先来先服务FCFS算法

step：

1、将线程调度算法修改为先来先服务FCFS算法

FCFS（First-Come, First-Served）算法，即先来先服务算法。在FCFS算法中，按照线程进入就绪队列的顺序进行调度，即先执行先进入就绪队列的线程，直到该线程执行完毕或者阻塞，才继续执行下一个作线程。

优点：实现简单、适用于短作业或进程、不会出现某些线程不能够执行的情况

缺点：因为没有考虑线程的执行时间而导致平均等待时间长

在 `program.cpp` 中，修改 `schedule` 函数为FCFS算法

```
void ProgramManager::schedule()
{
    bool status = interruptManager.getInterruptStatus();
    //在进程线程调度前，关中断
    interruptManager.disableInterrupt();
    //判断当前可调度的线程数量，如果readyProgram为空，说明当前系统中只有一个线程，无需进行调度，直接返回即可。
    if (readyPrograms.size() == 0)
    {
        interruptManager.setInterruptStatus(status);
        return;
    }
    // FCFS算法不需要进行RUNNING到READY的处理
    // if (running->status == ProgramStatus::RUNNING)
    // {
    //     running->status = ProgramStatus::READY;
    //     running->ticks = running->priority * 10;
    //     readyPrograms.push_back(&(running->tagInGeneralList));
    // }
    //当前线程是终止态(DEAD)，回收线程的PCB
    if (running->status == ProgramStatus::DEAD)
    {
        releasePCB(running);
    }
    //就绪队列的第一个线程作为下一个执行的线程
    ListItem *item = readyPrograms.front();
    PCB *next = ListItem2PCB(item, tagInGeneralList);
    PCB *cur = running;
    //从就绪队列中删去第一个线程，设置其状态为运行态和当前正在执行的线程
    next->status = ProgramStatus::RUNNING;
    running = next;
    readyPrograms.pop_front();
    //将线程从cur切换到next
    asm_switch_thread(cur, next);
    interruptManager.setInterruptStatus(status);
}
```

2、编写代码，测试算法的正确性

按照以下顺序创建四个线程：first_thread、forth_thread、third_thread、second_thread

分别赋予四个线程不同的优先级，分别为1,10,5,3

```
    int pid1 = programManager.executeThread(first_thread, nullptr, "first thread", 1);
    if (pid1 == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }
    int pid4 = programManager.executeThread(forth_thread, nullptr, "forth thread", 10);
    if (pid4 == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }
    int pid3 = programManager.executeThread(third_thread, nullptr, "third thread", 5);
    if (pid3 == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }

    int pid2 = programManager.executeThread(second_thread, nullptr, "second thread", 3);
    if (pid2 == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }
```

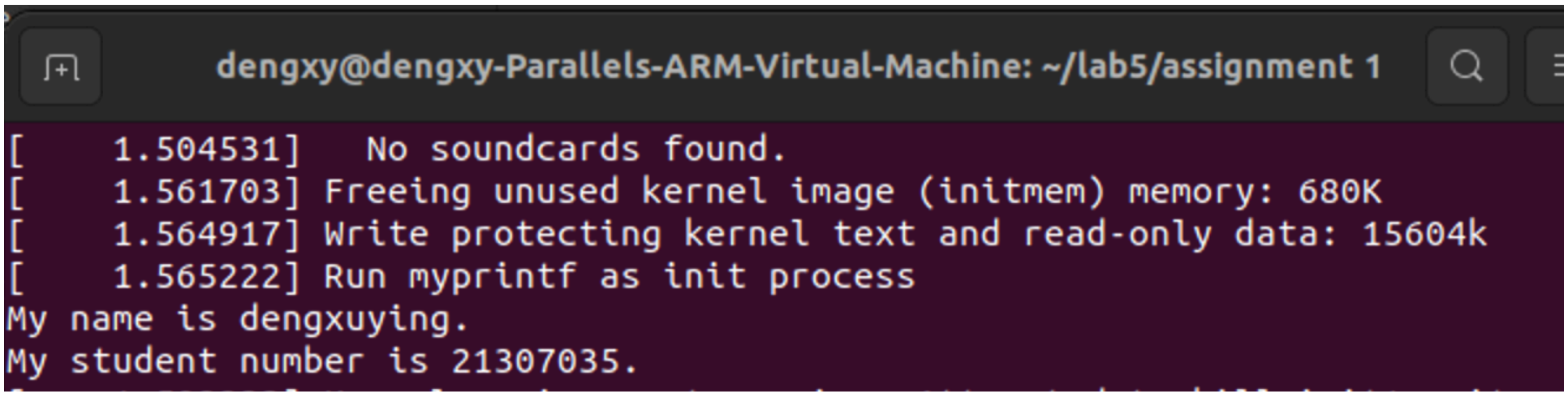
3、运行结果(运行截图详见：实验结果→Assignment 4)

```
pid 0 name "first thread": assignment4.1
pid 1 name "forth thread": assignment4.4
pid 2 name "third thread": assignment4.3
pid 3 name "second thread": assignment4.2
```

结果符合预期

3、实验结果

Assignment 1



Assignment 2

运行示例代码，完成线程的实现

```
SeaBIOS (version 1.16.0-debian-1.16.0-4)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00



Booting from Hard Disk...
pid 0 name "first thread": assignment 2
```

自定义实现PCB，完成线程的实现

```
dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab5/assignment 2$ ./pcb.out
Thread 3 is running
Thread 3 is finished
Thread 2 is running
Thread 2 is finished
Thread 1 is running
Thread 1 is finished
```

Assignment 3

```
SeaBIOS (version 1.16.0-debian-1.16.0-4)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00



Booting from Hard Disk...
pid 0 name "first thread": assignment3.1
```

```
SeaBIOS (version 1.16.0-debian-1.16.0-4)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00



Booting from Hard Disk...
pid 0 name "first thread": assignment3.1
pid 1 name "second thread": assignment3.2
```

```
SeaBIOS (version 1.16.0-debian-1.16.0-4)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00



Booting from Hard Disk...
pid 0 name "first thread": assignment3.1
pid 1 name "second thread": assignment3.2
pid 2 name "third thread": assignment3.3
```

Assignment 4

```
SeaBIOS (version 1.16.0-debian-1.16.0-4)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00



Booting from Hard Disk...
pid 0 name "first thread": assignment4.1
pid 1 name "forth thread": assignment4.4
pid 2 name "third thread": assignment4.3
pid 3 name "second thread": assignment4.2
```

4、关键代码

Assignment 1

```cpp
#include <iostream>
typedef char *va_list;
#define _INTSIZEOF(n) ((sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1))
#define va_start(ap, v) (ap = (va_list)&v + _INTSIZEOF(v))
#define va_arg(ap, type) (*(type *)((ap += _INTSIZEOF(type)) - _INTSIZEOF(type)))
#define va_end(ap) (ap = (va_list)0)

void myprintf(const char* format, ...)
{
    va_list args;
    va_start(args, format);
    while (*format != '\0')
    {
        if (*format == '%')
        {
            ++format;
            switch (*format)
            {
                case 'd':
                {
                    int value = va_arg(args, int);
                    std::cout<<value;
                    break;
                }
                case 's':
                {
                    char* value = va_arg(args, char*);
                    std::cout<<value;
                    break;
                }
                default:
                {
                    std::cout<<"Error"<<std::endl;
                    break;
                }
            }
        }
        else
        {
            std::putchar(*format);
        }
        ++format;
    }
    va_end(args);
}
```

```c
int main()
{
    myprintf("My name is %s.\n", "dengxuying");
    myprintf("My student number is %d.\n", 21307035);
    return 0;
}
```

定义了一个函数 `myprintf`，通过使用可变参数列表实现了类似于C语言中的 `printf` 函数的功能。

- `myprintf` 函数的第一个参数是格式化字符串，后面的参数是根据格式化字符串中的占位符来传递的。
- 定义了一个 `va_list` 类型的变量 `args`，然后使用 `va_start` 来初始化 `args`，使其指向可变参数列表中的第一个参数。
- 在循环中，通过检查格式化字符串中的每个字符，来判断是否需要进行格式化输出。如果当前字符是 `%`，则表示需要进行格式化输出，此时会继续读取下一个字符以确定输出类型，然后使用 `va_arg` 来获取可变参数列表中的下一个参数，并将其输出到标准输出流中。如果当前字符不是 `%`，则直接输出该字符。
- 在 `main` 函数中，通过调用 `myprintf` 函数来输出字符串和数字。

Assignment 2

代码解析详见：实验过程→Assignment 2 线程的实现→自行设计PCB实现线程

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

typedef enum
{
    CREATED,
    RUNNING,
    READY,
    BLOCKED,
    DEAD,
} State;

typedef struct
{
    int id;
    int priority;//优先级越高，数字越小
    State state;
} PCB;

typedef struct
{
    void (*Run)(void*);
    PCB pcb;
} Thread;

Thread* threads[3];
int num_threads = 0;

void thread_start(Thread* t)
{
    t->pcb.state = RUNNING;
    t->Run(t);
    t->pcb.state = DEAD;
}

void thread_create(Thread* t, void (*Run)(void*), int id, int priority)
{
    t->Run = Run;
    t->pcb.id = id;
    t->pcb.priority = priority;
    t->pcb.state = CREATED;
```

```
    threads[num_threads] = t;
    num_threads++;
}

void thread_start_execute(Thread* t)
{
    t->pcb.state = READY;
    thread_start(t);
}

void thread_dead(Thread* t)
{
    t->pcb.state = DEAD;
}

void my_thread_run(void* arg)
{
    Thread* t = (Thread*) arg;
    printf("Thread %d is running\n", t->pcb.id);
    printf("Thread %d is finished\n", t->pcb.id);
    thread_dead(t);
}

void sort_threads()
{
    int i, j;
    Thread* temp;
    for (i = 0; i < num_threads - 1; i++)
    {
        for (j = i + 1; j < num_threads; j++)
        {
            if (threads[i]->pcb.priority > threads[j]->pcb.priority)
            {
                temp = threads[i];
                threads[i] = threads[j];
                threads[j] = temp;
            }
        }
    }
}

Thread* get_next_thread()
{
    int i;
    Thread* next = NULL;
    for (i = 0; i < num_threads; i++)
    {
        if (threads[i]->pcb.state == CREATED)
        {
            next = threads[i];
            break;
        }
    }
    return next;
}

int main()
{
    Thread t1, t2, t3;
    thread_create(&t1, my_thread_run, 1, 3);
    thread_create(&t2, my_thread_run, 2, 2);
    thread_create(&t3, my_thread_run, 3, 1);
    while (1)
    {
        sort_threads();
```

```
        Thread* next = get_next_thread();
        if (next)
        {
            thread_start_execute(next);
        }
        else
        {
            break;
        }
    }
    return 0;
}
```

Assignment 4

```
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;
    //     running->ticks = running->priority * 10;
    //     readyPrograms.push_back(&(running->tagInGeneralList));
    // }
    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、总结

- 学习了C语言的可变参数机制的实现方法，能够实现一个简单的printf函数
- 学习了内核线程的实现，理解了 `asm_switch_thread` 实现线程的切换的原理
- 了解了不同的线程调度算法及其背后的逻辑