



本科生实验报告

实验课程: 操作系统

实验名称: 内核线程

专业名称: 信息与计算科学

学生姓名: 张文沁

学生学号: 20337268

实验地点:

实验成绩:

报告时间: 2022.4.23

代码全部贴在了文档中

1. *实验要求*

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。

实验要求

- DDL: 2021.4.29 23:59
 - 提交的内容: 将4个assignment的代码和实验报告放到压缩包中, 命名为“lab5-姓名-学号”, 并交到课程网站上[<http://course.dds-sysu.tech/course/3/homework>]
 - 材料的Example的代码放置在 `src` 目录下。
1. 实验不限语言, C/C++/Rust都可以。
 2. 实验不限平台, Windows、Linux和MacOS等都可以。
 3. 实验不限CPU, ARM/Intel/Risc-V都可以。

2. *实验过程*

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。

```
print("%a %d %c", a, b, c);
```

对于固定参数列表的函数, 每个参数的名称、类型都是直接可见的, 他们的地址也都是可以直接得到的, 比如: 通过函数原型声明了解到a是int类型的; b是double类型的; c是char*类型的, 地址也可以通过&a直接得到。

但是对于变长参数的函数, 我们就没有这么顺利了。还好, 按照C标准的说明, 支持变长参数的函数在原型声明中, 必须有至少一个最左固定参数(这一点与传统C有区别, 传统C允许不带任何固定参数的纯变长参数函数), 这样我们可以得到其中固定参数的地址, 但是依然无法从声明中得到其他变长参数的地址, 比如:

```
void func(const char * fmt, ... ) {  
    ...  
}
```

这里我们只能得到fmt这固定参数的地址, 仅从函数原型我们是无法确定"..."中有几个参数、参数都是什么类型的, 自然也就无法确定其位置了。我们可以通过如下程序分析得到输出函数的传参方式, 显而易见, 是栈操作。

```
#include<iostream>  
using namespace std;  
void Address(int a, double b, char *c) {  
    printf("a = 0x%p\n", &a);  
    printf("b = 0x%p\n", &b);  
    printf("c = 0x%p\n", &c);  
}  
int main() {  
    Address(1, 2.3, "hello world");  
    return 0;  
}
```

```
address.cpp > main()
1  #include<iostream>
2  using namespace std;
3  void Address(int a, double b, char *c) {
4      printf("a = 0x%p\n", &a);
5      printf("b = 0x%p\n", &b);
6      printf("c = 0x%p\n", &c);
7  }
8  int main() {
9      Address(1, 2.3, "hello world");
10     return 0;
11 }
```

输出 调试控制台 问题 1 终端

Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。

安装最新的 PowerShell, 了解新功能和改进! <https://aka.ms/PSWindows>

PS F:\Codefile of visual studio code> & 'c:\Users\张文沁\.vscode\extensions\ms-vscode.cpptools-1.9.8-win32-x64\debugAdapters\bin\WindowsDebugLaunc
her.exe' '--stdin=Microsoft-MIEngine-In-ntluepr4.3p3' '--stdout=Microsoft-MIEngine-Out-quu3rkt4.2or' '--stderr=Microsoft-MIEngine-Error-gqlq15dp.u2
5' '--pid=Microsoft-MIEngine-Pid-f2z4wrswhfj' '--dbgExe=F:\gcc\mingw64\bin\gdb.exe' '--interpreter=mi'
a = 0x000000000062fe00
b = 0x000000000062fe08
c = 0x000000000062fe10
PS F:\Codefile of visual studio code>

所以可以推导出变长参数传递和固定参数的传参过程是一样的, 简单来讲都是栈操作, 而栈这个东西对我们是开放的。这样一来, 一旦我们知道某函数帧的栈上的一个固定参数的位置, 我们完全有可能推导出其他变长参数的位置。

通过这个思路, 可以通过可变参数模板实现可变参数输出函数:

```
#include <iostream>

void FormatPrint()
{
    std::cout << std::endl;
}

template <class T, class ...Args>
void FormatPrint(T first, Args... args)
{
    std::cout << "[" << first << "];
    FormatPrint(args...);
}

int main(void)
{
    FormatPrint(1, 2, 3, 4);
    FormatPrint("hello", 1, "world", 2, 3, 'A');
    return 0;
}
```

结果如下:

```
anoi.cpp > main(void)
1  #include <iostream>
2
3  void FormatPrint()
4  {
5      std::cout << std::endl;
6  }
7  template <class T, class ...Args>
8  void FormatPrint(T first, Args... args)
9  {
10     std::cout << "[" << first << " ";
11     FormatPrint(args...);
12 }
13
14 int main(void)
15 {
16     FormatPrint(1, 2, 3, 4);
17     FormatPrint("hello", 1, "world", 2, 3, 'A');
18     return 0;
19 }
20
```

输出 调试控制台 问题 终端

Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。

安装最新的 PowerShell，了解新功能和改进！<https://aka.ms/PSWindows>

PS F:\Codefile of visual studio code> & 'c:\Users\张文沁\.vscode\extensions\ms-vscode.cpptools-1.9.8-win32-x64\debugAdapters\bin\WindowsDebugLaunc
her.exe' '--stdin=Microsoft-MIEngine-In-ah0nzmtm.xow' '--stdout=Microsoft-MIEngine-Out-tif5jyhm.45d' '--stderr=Microsoft-MIEngine-Error-yeqzur3.ts
i' '--pid=Microsoft-MIEngine-Pid-dmyoljfd.2p4' '--dbgExe=F:\gcc\mingw64\bin\gdb.exe' '--interpreter=mi'
[1][2][3][4]
[hello][1][world][2][3][A]
PS F:\Codefile of visual studio code>

或者有以下有第一个参数的边长输出：

```
void var_args_func(const char * fmt, ... ) {
    char *ap;

    ap = ((char*)&fmt) + sizeof(fmt);
    printf("%d\n", *(int*)ap);

    ap = ap + sizeof(int);
    printf("%d\n", *(int*)ap);

    ap = ap + sizeof(int);
    printf("%s\n", *((char**)ap));
}

int main(){
    var_args_func("%d %d %s\n", 4, 5, "hello world");
}
```

至此，我们实现了变长参数在C++下的输出。有了第二个例子的输出方式，我们只需得到fmt中的%后面的字符便可以确定要输出的参数类型，下面进行printf的实现：

首先为了实现进行参数 `fmt` 的解析，定义下面的函数作为缓冲区，如果 `fmt[i]` 不是%，则是普通字符，放入缓存区。

```
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;
}

```

定义适用qemu的I/O函数，用于输出到屏幕、光标位置选取、换行、滚屏等操作。

```

STDIO::STDIO()
{
    initialize();
}

void STDIO::initialize()
{
    screen = (uint8 *)0xb8000; //初始化
}

void STDIO::print(uint x, uint y, uint8 c, uint8 color)//基本输出，给定位置
{
    if (x >= 25 || y >= 80) // 超出屏幕
    {
        return;
    }

    uint pos = x * 80 + y;//计算输出的位置
    screen[2 * pos] = c;
    screen[2 * pos + 1] = color;
}

void STDIO::print(uint8 c, uint8 color)//无初始位置的输出
{
    uint cursor = getCursor();
    screen[2 * cursor] = c;
    screen[2 * cursor + 1] = color;
    cursor++;
    if (cursor == 25 * 80) //本页满，则滚动到下一行
    {
        rollUp();
        cursor = 24 * 80;
    }
    moveCursor(cursor);//屏幕的滚动实际上是指针的位置的变化
}

void STDIO::print(uint8 c)
{
    print(c, 0x07);
}

void STDIO::moveCursor(uint position)// 移动指针
{
    if (position >= 80 * 25)

```

```

{
    return;
}

uint8 temp;

// 处理高8位
temp = (position >> 8) & 0xff;
asm_out_port(0x3d4, 0x0e);
asm_out_port(0x3d5, temp);

// 处理低8位
temp = position & 0xff;
asm_out_port(0x3d4, 0x0f);
asm_out_port(0x3d5, temp);
}

uint STDIO::getCursor() //得到指针位置
{
    uint pos;
    uint8 temp;

    pos = 0;
    temp = 0;
    // 处理高8位
    asm_out_port(0x3d4, 0x0e);
    asm_in_port(0x3d5, &temp);
    pos = ((uint)temp) << 8;

    // 处理低8位
    asm_out_port(0x3d4, 0x0f);
    asm_in_port(0x3d5, &temp);
    pos = pos | ((uint)temp);

    return pos;
}

void STDIO::moveCursor(uint x, uint y) //给定x,y的移动
{
    if (x >= 25 || y >= 80)
    {
        return;
    }

    moveCursor(x * 80 + y);
}

void STDIO::rollUp() //滚屏
{
    uint length;
    length = 25 * 80;
    for (uint i = 80; i < length; ++i)
    {
        screen[2 * (i - 80)] = screen[2 * i];
        screen[2 * (i - 80) + 1] = screen[2 * i + 1];
    }

    for (uint i = 24 * 80; i < length; ++i)

```

```

    {
        screen[2 * i] = ' ';
        screen[2 * i + 1] = 0x07;
    }
}

int STDIO::print(const char *const str)
{
    int i = 0;
    for (i = 0; str[i]; ++i)
    {
        switch (str[i])
        {
            case '\n':
                uint row;
                row = getCursor() / 80;
                if (row == 24) //row==25执行rollUp()似乎也没有什么影响
                {
                    rollUp();
                }
                else
                {
                    ++row;
                }
                moveCursor(row * 80);
                break;

            default:
                print(str[i]);
                break;
        }
    }

    return i;
}

```

printf的实现如下:

```

int printf(const char *const fmt, ...)
{
    const int BUF_LEN = 32;
    char buffer[BUF_LEN + 1]; //定义缓冲区
    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])//根据%后字符的不同进行相应的输出
        {
            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': //本处需要进行进制转换
                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;
}

```

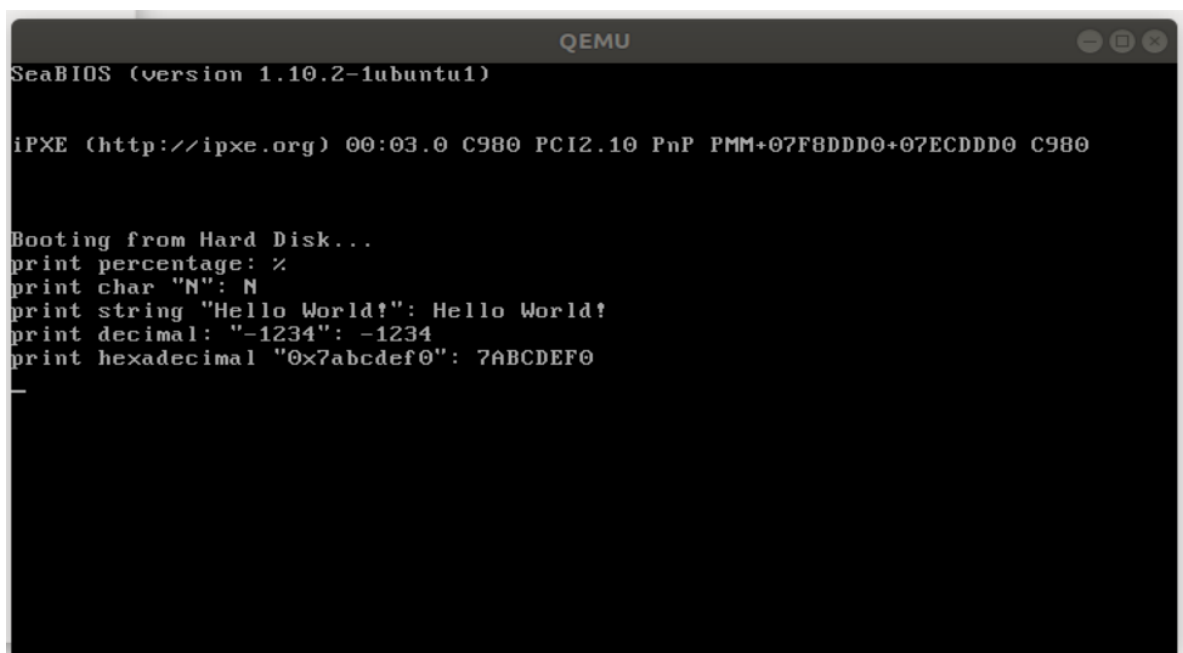

测试函数:

```
#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();
}
```

最终结果如下:



```
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980

Booting from Hard Disk...
print percentage: %
print char "N": N
print string "Hello World!": Hello World!
print decimal: "-1234": -1234
print hexadecimal "0x7abcdef0": 7ABCDEF0
_
```

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。

设计线程，线程最重要的部分也是线程切换的基本单位,PCB，相当于一个结构体，则第一步设计PCB：

```
struct PCB
{
    int *stack;                // 栈指针，用于调度（挂起和就绪）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
};
```

在设计完PCB内容之后，内部还需要特别设置的是状态模型，此处使用五状态模型：

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

ListItem是线程的队列标识符，如下定义，双向list。分为两个队列，所有进程的队列和处于就绪态的队列

```
struct ListItem
{
    ListItem *previous;
    ListItem *next;
};
```

至此，线程的定义结束，下面实现线程的相关函数：

首先为实现线程创建一个类框架：

```
#ifndef PROGRAM_H
#define PROGRAM_H

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

    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
// 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
void releasePCB(PCB *program);
};
#endif

```

PCB创建

在线程创建的时候, 需要申请一个PCB, 一般大小为页大小 (4k), PCB基本参数如下定义:

```

// 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];

```

分配和回收PCB:

```

PCB *ProgramManager::allocatePCB()
{
    for (int i = 0; i < MAX_PROGRAM_AMOUNT; ++i)
    {
        if (!PCB_SET_STATUS[i])
        {
            PCB_SET_STATUS[i] = true;
            //返回当前程序的PCB
            return (PCB *)((int)PCB_SET + PCB_SIZE * i);
        }
    }

    return nullptr;
}

void ProgramManager::releasePCB(PCB *program)
{
    int index = ((int)program - (int)PCB_SET) / PCB_SIZE;
    //直接将状态设置为未分配即可
    PCB_SET_STATUS[index] = false;
}

```

创建进程并放入就绪队列:

```

int ProgramManager::executeThread(ThreadFunction function, void *parameter,
const char *name, int priority)
{
    // 关中断, 防止创建线程的过程被打断

```

```

bool status = interruptManager.getInterruptStatus();
interruptManager.disableInterrupt();

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

if (!thread)
    return -1;

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

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

//对线程进行初始化
thread->status = ProgramStatus::READY;
thread->priority = priority;
thread->ticks = priority * 10;
thread->ticksPassedBy = 0;
thread->pid = ((int)thread - (int)PCB_SET) / PCB_SIZE;

// 线程栈
thread->stack = (int *)((int)thread + PCB_SIZE);
thread->stack -= 7;
thread->stack[0] = 0; //ebp
thread->stack[1] = 0; //ebx
thread->stack[2] = 0; //edi
thread->stack[3] = 0; //esi
thread->stack[4] = (int)function; //线程函数的起始地址
thread->stack[5] = (int)program_exit; //线程的返回地址
thread->stack[6] = (int)parameter; //线程的参数地址

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

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

return thread->pid;
}

```

至此，线程的创建完成，下面要实现线程的调度：

```

extern "C" void c_time_interrupt_handler()
{
    PCB *cur = programManager.running;

    if (cur->ticks)
    {
        --cur->ticks; //如果轮到该线程执行，则需要将当前线程的总时间减一
        ++cur->ticksPassedBy;
    }
    else
    {
        programManager.schedule();
    }
}

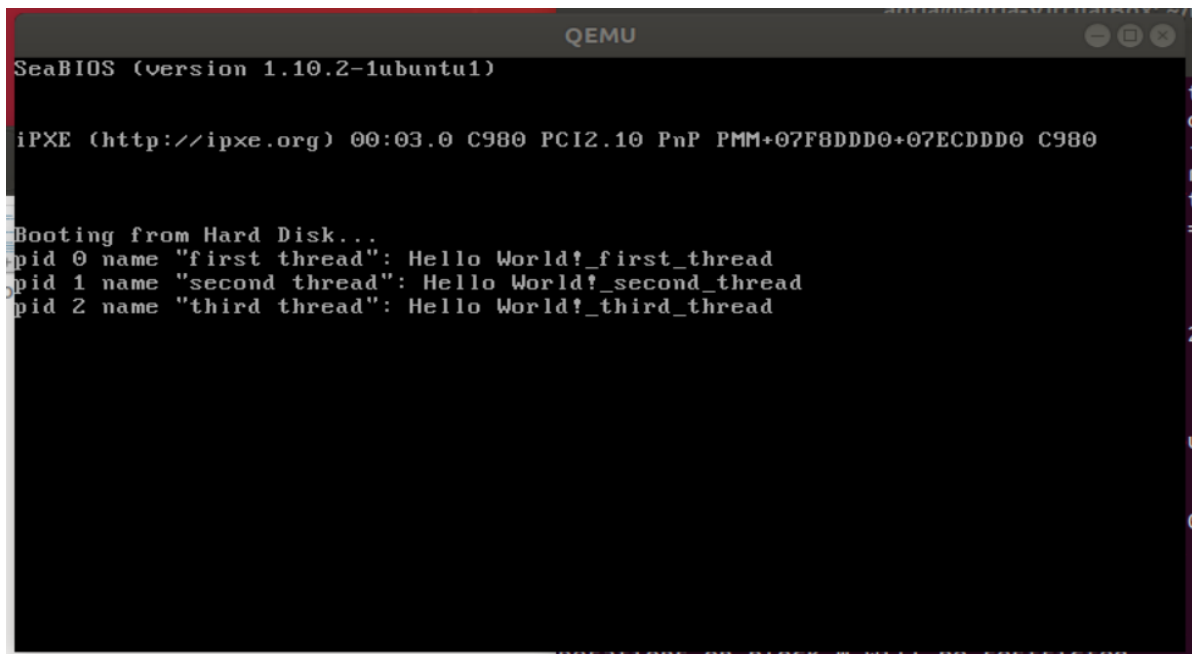
```

```
}  
}
```

线程调度函数:

```
void ProgramManager::schedule()  
{  
    //开始关中断  
    bool status = interruptManager.getInterruptStatus();  
    interruptManager.disableInterrupt();  
    //如果就绪态的队列为空,则无需再调度,直接返回  
    if (readyPrograms.size() == 0)  
    {  
        interruptManager.setInterruptStatus(status);  
        return;  
    }  
    //如果是运行态,则变为就绪态,时间片重新分配  
    if (running->status == ProgramStatus::RUNNING)  
    {  
        running->status = ProgramStatus::READY;  
        running->ticks = running->priority * 10;  
        readyPrograms.push_back(&(running->tagInGeneralList));  
    }  
    //如果是结束,则释放该程序的PCB即可  
    else if (running->status == ProgramStatus::DEAD)  
    {  
        releasePCB(running);  
    }  
    //准备将下一个就绪线程放入运行  
    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);  
    //执行完之后开中断  
    interruptManager.setInterruptStatus(status);  
}
```

结果如下:



```
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980

Booting from Hard Disk...
pid 0 name "first thread": Hello World!_first_thread
pid 1 name "second thread": Hello World!_second_thread
pid 2 name "third thread": Hello World!_third_thread
```

Assignment 3 线程调度切换的秘密

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。

调度并执行

1. 线程 `firstThread` 通过 `executeThread` 创建后进入就绪队列, `firstThread` 为第一个线程, 所以为了使它运行起来, 我们手动设置它的状态为 `RUNNING`, 移出就绪队列, 并调用函数 `asm_switch_thread`。使得 `firstThread` 成功运行 (该过程类似线程调度函数 `schedule`)。

首先获取`first_thread`的相关信息:

通过

`p/x firstThread`

`p/x &firstThread`

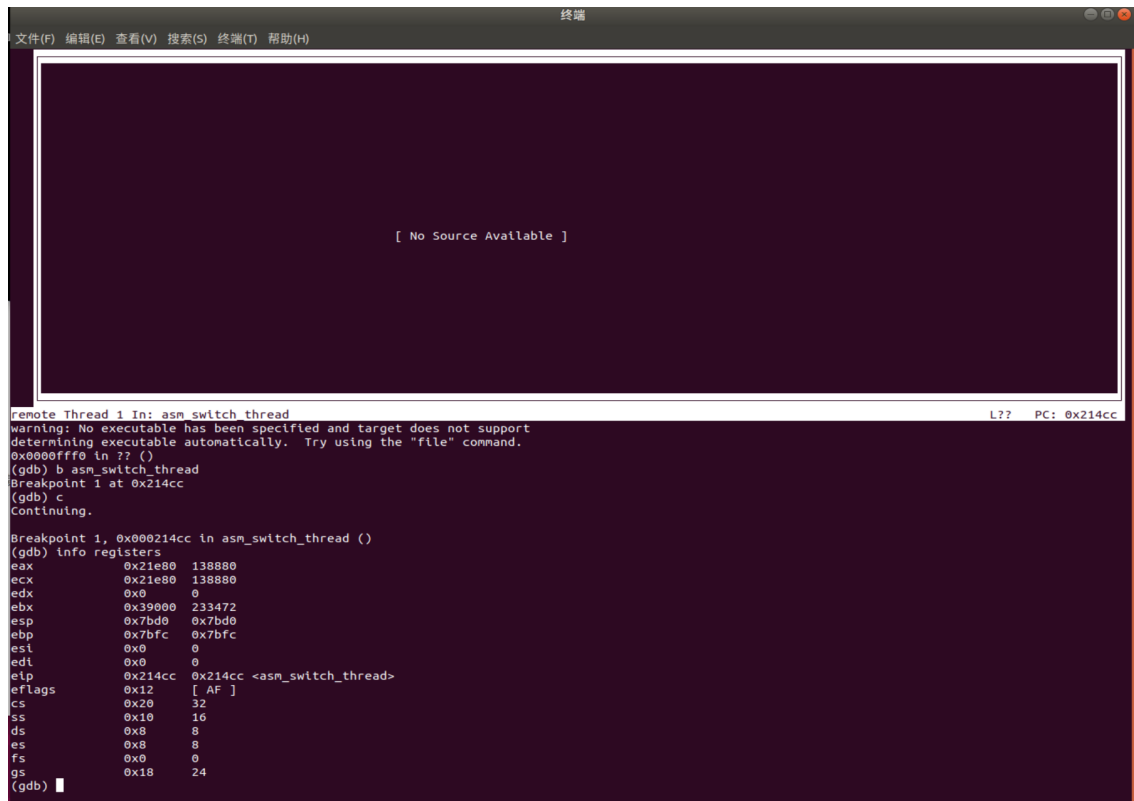
`p/x firstThread->stack`

`p/x &firstThread->stack`

可以获得`firstThread`的内容和地址

如下图所示:

2. 下面进入asm_switch_thread函数，进入函数之前寄存器的数值如下



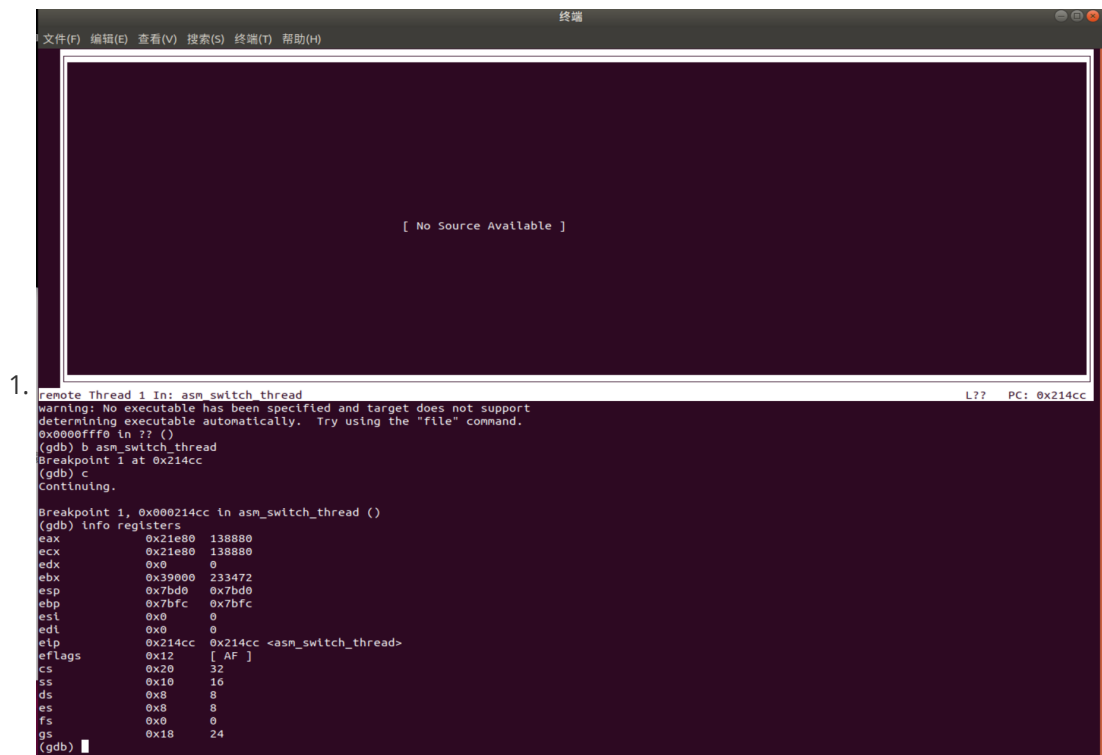
```
remote Thread 1 In: asm_switch_thread
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000ffff in ?? ()
(gdb) b asm_switch_thread
Breakpoint 1 at 0x214cc
(gdb) c
Continuing.

Breakpoint 1, 0x000214cc in asm_switch_thread ()
(gdb) info registers
eax      0x21e80  138880
ecx      0x21e80  138880
edx      0x0      0
ebx      0x39000  233472
esp      0x7bd0   0x7bd0
ebp      0x7bfc   0x7bfc
esi      0x0      0
edi      0x0      0
eip      0x214cc  0x214cc <asm_switch_thread>
eflags   0x12     [ AF ]
cs       0x20     32
ss       0x10     16
ds       0x0      0
es       0x8      8
fs       0x0      0
gs       0x18     24
(gdb)
```

3. 第一步是保存ebp ebx edi esi ,为了匹配C语言的特性，否则切换之后会报错，本步对应四个push语句

```
; void asm_switch_thread(PCB *cur, PCB *next);
asm_switch_thread:
    push ebp
    push ebx
    push edi
    push esi
```

四步对应寄存器如下：



```
remote Thread 1 In: asm_switch_thread
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000ffff in ?? ()
(gdb) b asm_switch_thread
Breakpoint 1 at 0x214cc
(gdb) c
Continuing.

Breakpoint 1, 0x000214cc in asm_switch_thread ()
(gdb) info registers
eax      0x21e80  138880
ecx      0x21e80  138880
edx      0x0      0
ebx      0x39000  233472
esp      0x7bd0   0x7bd0
ebp      0x7bfc   0x7bfc
esi      0x0      0
edi      0x0      0
eip      0x214cc  0x214cc <asm_switch_thread>
eflags   0x12     [ AF ]
cs       0x20     32
ss       0x10     16
ds       0x0      0
es       0x8      8
fs       0x0      0
gs       0x18     24
(gdb)
```



2. remote Thread 1 In: asm_switch_thread L?? PC: 0x214cd
determining executable automatically. Try using the "file" command.
0x0000fff0 in ?? ()
(gdb) b asm_switch_thread
Breakpoint 1 at 0x214cc
(gdb) c
continuing.
Breakpoint 1, 0x000214cc in asm_switch_thread ()
(gdb) si
0x000214cd in asm_switch_thread ()
(gdb) info registers
eax 0x21e80 138880
ecx 0x21e80 138880
edx 0x0 0
ebx 0x39000 233472
esp 0x7bcc 0x7bcc
ebp 0x7bfc 0x7bfc
esi 0x0 0
edi 0x0 0
eip 0x214cd 0x214cd <asm_switch_thread+1>
eflags 0x12 [AF]
cs 0x20 32
ss 0x10 16
ds 0x8 8
es 0x8 8
fs 0x0 0
gs 0x18 24
(gdb) █

(gdb) info registers
eax 0x21e80 138880
ecx 0x21e80 138880
edx 0x0 0
ebx 0x39000 233472
esp 0x7bc8 0x7bc8
ebp 0x7bfc 0x7bfc
esi 0x0 0
edi 0x0 0
3. eip 0x214ce 0x214ce <asm_switch_thread+2>
eflags 0x12 [AF]
cs 0x20 32
ss 0x10 16
ds 0x8 8
es 0x8 8
fs 0x0 0
gs 0x18 24
(gdb)


```

0x000214cf in asm_switch_thread ()
(gdb) info registers
eax             0x21e80    138880
ecx             0x21e80    138880
edx             0x0        0
ebx             0x39000    233472
esp             0x7bc4     0x7bc4
ebp             0x7bfc     0x7bfc
esi             0x0        0
edi             0x0        0
eip             0x214cf     0x214cf <asm_switch_thread+3>
eflags          0x12      [ AF ]
cs              0x20      32
ss              0x10      16
ds              0x8       8
es              0x8       8
fs              0x0       0
gs              0x18      24
(gdb) █

```

4. 下面保存 esp 的值，用做下次恢复使用。先将 cur -> stack 的地址放到 eax 中，第8行向[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

```

对应寄存器如下：保存esp

```

(gdb) sc
0x000214da in asm_switch_thread ()
(gdb) info registers
eax             0x21e80    138880
ecx             0x21e80    138880
edx             0x0        0
ebx             0x39000    233472
esp             0x7bc0     0x7bc0
ebp             0x7bfc     0x7bfc
esi             0x0        0
edi             0x0        0
eip             0x214da     0x214da <asm_switch_thread+14>
eflags          0x12      [ AF ]
cs              0x20      32
ss              0x10      16
ds              0x8       8
es              0x8       8
fs              0x0       0
gs              0x18      24
(gdb)

```

5. 将 next -> stack 的值写入到 esp 中，从而完成线程栈的切换。

由前面知道 firstThread -> stack 的值为0x22e64，地址为0x22000。首先将 firstThread -> stack 的地址放入 eax，然后在把 [eax] 的值赋给 esp（即从 firstThread -> stack 的地址中取出 firstThread -> stack 的值0x22e64保存在 esp 中），这样便完成了线程栈的切换

```

(gdb) si
0x000214dc in asm_switch_thread ()
(gdb) info registers
eax             0x21e80    138880
ecx             0x21e80    138880
edx             0x0        0
ebx             0x39000    233472
esp             0x22e64    0x22e64 <PCB_SET+4196>
ebp             0x7bfc     0x7bfc
esi             0x0        0
edi             0x0        0
eip             0x214dc    0x214dc <asm_switch_thread+16>
eflags          0x12      [ AF ]
cs              0x20      32
ss              0x10      16
ds              0x8        8
es              0x8        8
fs              0x0        0
gs              0x18      24
(gdb)

```

6. 恢复数据:

```

    pop esi
    pop edi
    pop ebx
    pop ebp

```

```

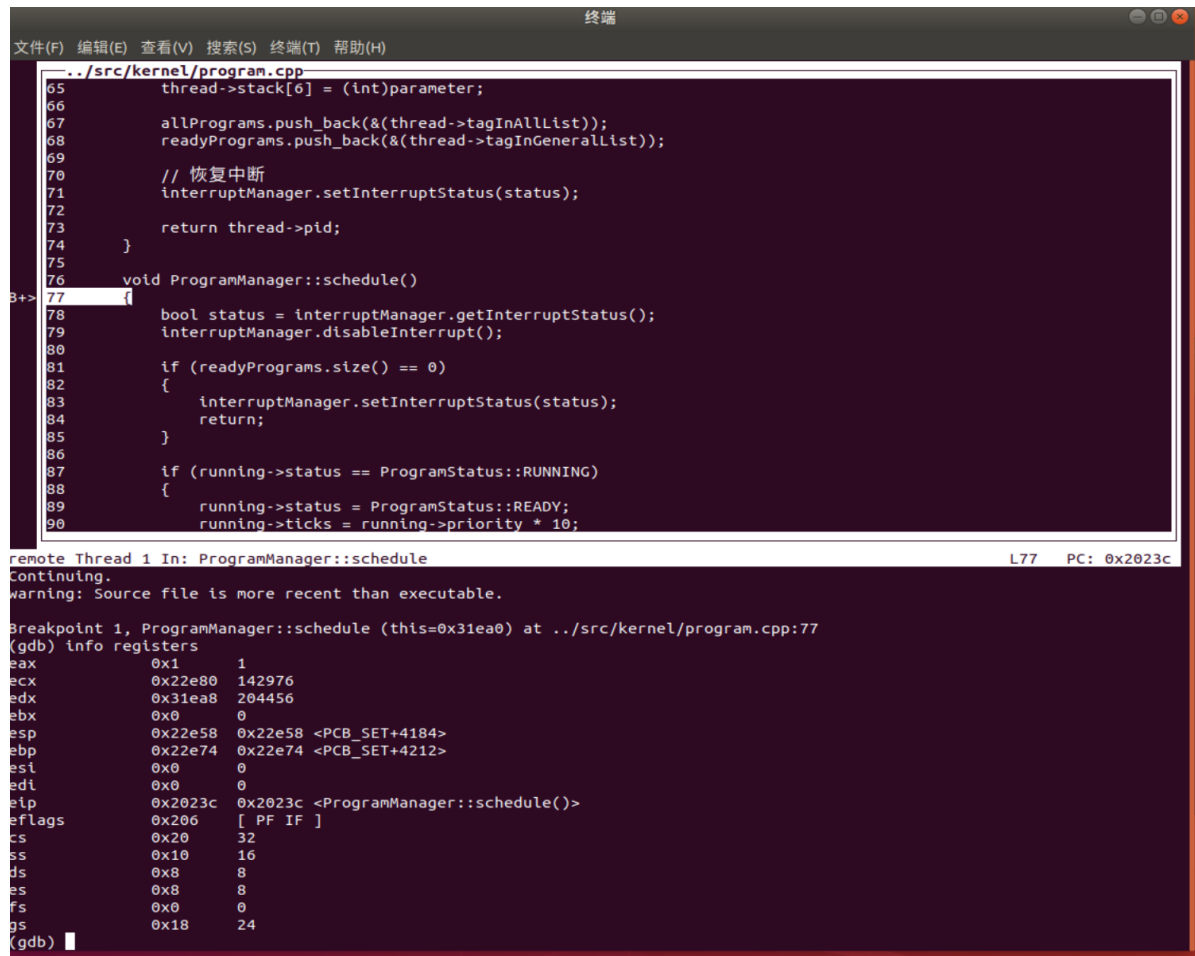
gs              0x18      24
(gdb) si
0x000214e0 in asm_switch_thread ()
(gdb) info registers
eax             0x21e80    138880
ecx             0x21e80    138880
edx             0x0        0
ebx             0x0        0
esp             0x22e74    0x22e74 <PCB_SET+4212>
ebp             0x0        0x0
esi             0x0        0
edi             0x0        0
eip             0x214e0    0x214e0 <asm_switch_thread+20>
eflags          0x12      [ AF ]
cs              0x20      32
ss              0x10      16
ds              0x8        8
es              0x8        8
fs              0x0        0
gs              0x18      24
(gdb)

```

线程的切换:

线程的切换依赖于PCB的切换和寄存器的恢复

1. 创建断点查看在firstThread执行完之后的寄存器：



```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/kernel/program.cpp
65     thread->stack[0] = (int)parameter;
66
67     allPrograms.push_back(&(thread->tagInAllList));
68     readyPrograms.push_back(&(thread->tagInGeneralList));
69
70     // 恢复中断
71     interruptManager.setInterruptStatus(status);
72
73     return thread->pid;
74 }
75
76 void ProgramManager::schedule()
77 {
78     bool status = interruptManager.getInterruptStatus();
79     interruptManager.disableInterrupt();
80
81     if (readyPrograms.size() == 0)
82     {
83         interruptManager.setInterruptStatus(status);
84         return;
85     }
86
87     if (running->status == ProgramStatus::RUNNING)
88     {
89         running->status = ProgramStatus::READY;
90         running->ticks = running->priority * 10;
91     }
92 }

remote Thread 1 In: ProgramManager::schedule L77 PC: 0x2023c
Continuing.
warning: Source file is more recent than executable.

Breakpoint 1, ProgramManager::schedule (this=0x31ea0) at ../src/kernel/program.cpp:77
(gdb) info registers
eax             0x1          1
ecx             0x22e80     142976
edx             0x31ea8     204456
ebx             0x0          0
esp             0x22e58     0x22e58 <PCB_SET+4184>
ebp             0x22e74     0x22e74 <PCB_SET+4212>
esi             0x0          0
edi             0x0          0
eip             0x2023c     0x2023c <ProgramManager::schedule()>
eflags          0x206       [ PF IF ]
cs              0x20        32
ss              0x10        16
ds              0x8         8
es              0x8         8
fs              0x0         0
gs              0x18        24
(gdb)
```

2. 继续往下执行，会进入运行态变就绪态时间片重新分配的函数：其中寄存器如下



```
remote Thread 1 In: List::push_back L48 PC: 0x211c0
ss              0x10        16
ds              0x8         8
es              0x8         8
fs              0x0         0
gs              0x18        24
eax             0x21eac     138924
ecx             0x1          1
edx             0x31ea8     204456
ebx             0x0          0
esp             0x22e14     0x22e14 <PCB_SET+4116>
ebp             0x22e24     0x22e24 <PCB_SET+4132>
esi             0x0          0
edi             0x0          0
eip             0x211c0     0x211c0 <List::push_back(ListItem*)+6>
eflags          0x6         [ PF ]
cs              0x20        32
ss              0x10        16
ds              0x8         8
es              0x8         8
fs              0x0         0
gs              0x18        24
(gdb)
```

3. 在加载下一个进程的时候：可以查看next的内容和地址，如下p/x next

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/kernel/program.cpp
87     if (running->status == ProgramStatus::RUNNING)
88     {
89         running->status = ProgramStatus::READY;
90         running->ticks = running->priority * 10;
91         readyPrograms.push_back(&(running->tagInGeneralList));
92     }
93     else if (running->status == ProgramStatus::DEAD)
94     {
95         releasePCB(running);
96     }
97
98     ListItem *item = readyPrograms.front();
99     PCB *next = ListItem2PCB(item, tagInGeneralList);
100    > PCB *cur = running;
101    next->status = ProgramStatus::RUNNING;
102    running = next;
103    readyPrograms.pop_front();
104
105    asm_switch_thread(cur, next);
106
107    interruptManager.setInterruptStatus(status);
108 }
109
110 void program_exit()
111 {
112     PCB *thread = programManager.running;
```

```
remote Thread 1 In: ProgramManager::schedule L100 PC: 0x2032f
ebx      0x0      0
esp      0x22e3c  0x22e3c <PCB_SET+4156>
ebp      0x22e54  0x22e54 <PCB_SET+4180>
esi      0x0      0
edi      0x0      0
eip      0x2032f  0x2032f <ProgramManager::schedule()+243>
eflags   0x2      [ ]
cs       0x20     32
ss       0x10     16
ds       0x8      8
es       0x8      8
fs       0x0      0
gs       0x18     24
(gdb) p/x secondThread
No symbol "secondThread" in current context.
(gdb) p/x &nexxt
No symbol "nexxt" in current context.
(gdb) p/x &next
$1 = 0x22e44
(gdb) p/x next
$2 = 0x22e80
(gdb) █
```

4. 继续向下运行，可以看到程序在 (running = next) 已经将下一个线程的内容写入eax:

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/kernel/program.cpp
87     if (running->status == ProgramStatus::RUNNING)
88     {
89         running->status = ProgramStatus::READY;
90         running->ticks = running->priority * 10;
91         readyPrograms.push_back(&(running->tagInGeneralList));
92     }
93     else if (running->status == ProgramStatus::DEAD)
94     {
95         releasePCB(running);
96     }
97
98     ListItem *item = readyPrograms.front();
99     PCB *next = ListItem2PCB(item, tagInGeneralList);
100    PCB *cur = running;
101    > next->status = ProgramStatus::RUNNING;
102    running = next;
103    readyPrograms.pop_front();
104
105    asm_switch_thread(cur, next);
106
107    interruptManager.setInterruptStatus(status);
108 }
109
110 void program_exit()
111 {
112     PCB *thread = programManager.running;
```

```
remote Thread 1 In: ProgramManager::schedule L101 PC: 0x20338
(gdb) info registers
eax      0x21e80  138880
ecx      0x1      1
edx      0x22eac  143020
ebx      0x0      0
esp      0x22e3c  0x22e3c <PCB_SET+4156>
ebp      0x22e54  0x22e54 <PCB_SET+4180>
esi      0x0      0
edi      0x0      0
eip      0x20338  0x20338 <ProgramManager::schedule()+252>
eflags   0x2      [ ]
cs       0x20     32
ss       0x10     16
ds       0x8      8
es       0x8      8
fs       0x0      0
gs       0x18     24
(gdb) p/x next
$3 = 0x22e80
(gdb) p/x &next
$4 = 0x22e44
(gdb) █
```

5. 之后会跳转到asm_switch_thread函数，和（调度并执行）的执行过程一致

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，同学们需要将线程调度算法修改为上面提到的算法或者是同学们自己设计的算法。然后，同学们需要自行编写测试样例来呈现你的算法实现的正确性和基本逻辑。最后，将结果截图并说说你是怎么做的。

参考资料：<https://zhuanlan.zhihu.com/p/97071815>

Tips:

- 先来先服务最简单。
- 有些调度算法的实现可能需要用到中断。

FIFO算法，和时间片轮转算法区别在于时间片的分配与否，在FIFO中只需要判断当前线程状态进行响应，然后将就绪队列中第一个线程加入运行：

```
void ProgramManager::schedule()
{
    //开始关中断
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();
    //如果就绪态的队列为空，则无需再调度，直接返回
    if (readyPrograms.size() == 0)
    {
        interruptManager.setInterruptStatus(status);
        return;
    }
    //如果是运行态，则变为就绪态
    if (running->status == ProgramStatus::RUNNING)
    {
        running->status = ProgramStatus::READY;
        readyPrograms.push_back(&(running->tagInGeneralList));
    }
    //如果是结束，则释放该程序的PCB即可
    else if (running->status == ProgramStatus::DEAD)
    {
        releasePCB(running);
    }
    //准备将下一个就绪线程放入运行
    //取得第一个就绪线程
    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);
//执行完之后开中断
interruptManager.setInterruptStatus(status);
}

```

结果如下图所示，可以正常运行：

