

实验题目：

内核线程

实验要求：

- DDL：2021.4.30 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。

实验内容：

Assignment 1 printf的实现

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。

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。

实验步骤：

Assignment1printf的实现：

1、实现可变参数机制

printf函数是可变参数函数，因此实现之前我们需要先实现可变参数机制。

参数的传入顺序是从右至左入栈的，因此printf传入的第一个参数位于栈顶。因此我们可以通过第一个参数进行地址偏移得到第二个参数、第三个参数等。主要通过一个类型（va_list）和三个宏（va_start、va_arg、va_end）来实现。将定义放在stdarg.h中。

①其中va_list用来储存参数的类型信息。

```
typedef char * va_list;
```

②void va_start(va_list ap,paramN) 用来初始化可变参数列表。ap是可变参数列表地址，paramN是参数个数。

```
#define _INTSIZEOF(n) ( (sizeof(n)+sizeof(int)-1) & ~(sizeof(int)-1) )  
#define va_start(ap,v) ( ap = (va_list)&v + _INTSIZEOF(v) )
```

因为栈是用4个字节保存变量的，所以每次移动ap，都要4个字节对齐。_INTSIZEOF(n)返回的是n的大小进行4字节对齐的结果。是用于移动ap时对齐的。

③type va_arg(va_list ap, type);返回当前ap所指的type类型的数据，并将ap指向下一个参数。

void va_end(va_list ap);将ap清零。完成清理工作。

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

2、实现printf函数

printf函数的声明int printf(const char *const fmt, ...);在stdio.h中，实现在stdio.cpp中。

上个实验我们已经解决了简单字符的输出和屏幕输出的换行，换页等问题。这次实现printf将用到上次实验的函数。

①实现一个字符串的输出，包括换行功能。实现声明在stdio.h中，实现在stdio.cpp。

```
int STDIO::print(const char *const str)  
{  
    int i = 0;  
    for (i = 0; str[i]; ++i)  
    {  
        switch (str[i])  
        {  
            case '\n': //换行符的实现  
                uint row;  
                row = getCursor() / 80;  
                if (row == 24) //满页的情况，往上滑动  
                {  
                    rollup();  
                }  
            }  
        }  
    }
```

```

        }
        else
        {
            ++row; //否则，直接换行即可
        }
        moveCursor(row * 80);
        break;
    default:
        print(str[i]); //其他字符正常输出
        break;
    }
}
return i;
}

```

实现了字符串的输出之后，我们就可以通过一个函数把 `printf()` 中要输出的内容保存在一个缓冲区 `buffer` 中，该 `buffer` 是一个字符串，然后我们只需要调用 `print(const char *const str)` 函数就可以把 `buffer` 输出。

②实现函数 `int printf(const char *const fmt, ...)`，将要打印的内容转成字符串存在 `buffer` 中。函数在 `stdio.h` 中，实现在 `stdio.cpp` 中。这里只实现了 `%d`, `%c`, `%s`, `%x` 四个转义符。

```

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, int), 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;
        }

        temp = itos(number, temp, (fmt[i] == 'd' ? 10 : 16));

        for (int j = temp - 1; j >= 0; --j)
        {
            counter += printf_add_to_buffer(buffer, number[j], idx,
BUF_LEN);
        }
        break;

    }

}

buffer[idx] = '\0';
counter += stdio.print(buffer);

return counter;
}

```

对 `fmt` 进行逐字符解析。遇到转义字符则进行相应的处理，其他字符正常存入 `buffer`。

`number` 用来存放转换后的数字字符串。

`int printf_add_to_buffer(char *buffer, char c, int &idx, const int BUF_LEN)` 函数监控缓冲区是否已经满了，如果满了就先打印字符串，再清空缓冲区。实现如下：

```

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;
}

```

③实现数字向任意进制字符串的转换

为了实现 %d 和 %x, 需要将数字转换成对应的进制的字符串。对于负数的情况, 需要将负数转化成正数, 并在字符串前加个 '-' 号, 然后进行相同的处理。将函数声明在 `stdlib.h` 中, 实现在 `stdlib.cpp` 中。

```

void itos(char *numStr, uint32 num, uint32 mod) {
    // 只能转换2~26进制的整数
    if (mod < 2 || mod > 26 || num < 0) {
        return;
    }

    uint32 length, temp;

    // 进制转换
    length = 0;
    while(num) {
        temp = num % mod;
        num /= mod;
        numStr[length] = temp > 9 ? temp - 10 + 'A' : temp + '0';
        ++length;
    }

    // 特别处理num=0的情况
    if(!length) {
        numStr[0] = '0';
        ++length;
    }

    // 将字符串倒转, 使得numStr[0]保存的是num的高位数字
    for(int i = 0, j = length - 1; i < j; ++i, --j) {
        swap(numStr[i], numStr[j]);
    }

    numStr[length] = '\\0';
}

```

由于将数字转化成字符串存起来是从数字的地位开始存的 (因为使用的是 mod 运算存储), 如果直接输出字符串, 我们得到的数字的高位和地位是反过来的。因此在最后我们需要对数字字符串进行换位操作。在 `stdlib.h` 中声明一个简单的交换字符函数, 在 `stdlib.cpp` 中实现。如下:

```
template<typename T>
void swap(T &x, T &y) {
    T z = x;
    x = y;
    y = z;
}
```

至此，`printf` 函数已经实现，现在来验证一下它的正确性。

④验证函数的正确性。

在 `setup.cpp` 中添加语句

```
printf("print percentage: %%\n");
printf("print char \"N\": %c\n",'N');
printf("print string \"Hello world!\": %s\n","Hello world!");
printf("print decimal: \"-1234\": %d\n",-1234);
printf("print hexadecimal \"0x7abcdef0\": %x\n",0x7abcdef0);
```

修改 `makefile`

```
RUNDIR = ../run
BUILDDIR = build
INCLUDE_PATH = ../include

KERNEL_SOURCE = $(wildcard $(SRCDIR)/kernel/*.cpp)
CXX_SOURCE += $(KERNEL_SOURCE)
CXX_OBJ += $(KERNEL_SOURCE:$(SRCDIR)/kernel/%.cpp=%o)

UTILS_SOURCE = $(wildcard $(SRCDIR)/utils/*.cpp)
CXX_SOURCE += $(UTILS_SOURCE)
CXX_OBJ += $(UTILS_SOURCE:$(SRCDIR)/utils/%.cpp=%o)

ASM_SOURCE += $(wildcard $(SRCDIR)/utils/*.asm)
ASM_OBJ += $(ASM_SOURCE:$(SRCDIR)/utils/%.asm=%o)

OBJ += $(CXX_OBJ)
OBJ += $(ASM_OBJ)
```

然后在 `build` 目录下打开终端，执行命令 `make && make run`。查看运行结果。

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

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

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

可以看到，`printf` 输出符合预期。

Assignment2 线程的实现

这里实现的是内核线程，内核线程只在内核虚拟地址空间范围内活动。内核线程对内核是可见的，因此进程和内核线程都可以通过特殊结构PCB来描述。

每个PCB包含的内容：

- (1) 线程标识，如线程名和线程id，线程时间片总时间，线程的优先级；
- (2) 处理线程状态的信息，如线程的状态，线程已执行时间，线程队列标识，栈指针；

PCB的组织方式：

用链表组织，便于插入和删除。

同一个状态的线程PCB在同一个链表里，多个状态对应多个不同的链表。`list`声明在 `include/list.h` 中，实现在 `list.cpp` 中，`list`是个双向链表。

线程的实现：

线程的实现包括对PCB的申请和程序的运行以及内核对线程的调度。这里在 `include/ProgramManager.h` 中声明一个 `ProgramManager` 类帮我们实现。

该类中有处于就绪态的链表和一个记录所有（还没死）线程的总链表以及申请PCB和释放PCB的函数还有线程的调度函数。

步骤1：

在 `include/thread.h` 中声明一个PCB类。

```
//定义ThreadFunction函数指针类型；
//它指向返回类型为void，传入参数为void*型的函数。
typedef void(*ThreadFunction)(void*);
```

```

//线程的状态
enum ProgramStatus
{
    CREATED,
    RUNNING,
    READY,
    BLOCKED,
    DEAD
};

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
};

```

步骤2:

在 `include/program.h` 中声明 `ProgramManager` 类，并在 `kernel/program.cpp` 中实现具体函数。

```

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; //当前正在执行的线程
public:
    //初始化
    ProgramManager();
    void initialize();

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

    //创建线程的函数
    // function: 线程执行的函数
    // parameter: 指向函数的参数的指针，这里的参数类型一定要是void
    // 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);

    //线程的调度
    void schedule();
};

```


`initial()` 函数初始化ProgramManager中的两个list以及让标记每个PCB状态分配信息的PCB_SET_STATUS为false。

`PCB *allocatePCB()` 在PCB_SET 中寻找PCB_SET_STATUS 为false的PCB，然后将其标记为ture, 分配给新的线程。`void releasePCB(PCB* program)` 释放已执行完的线程的PCB。

创建线程用函数 `int executeThread(ThreadFunction function, void *parameter, const char *name, int priority)` 实现。

①首先，创建线程我们需要进行互斥处理。这里实现互斥用的是时钟中断，而没有用信号量。原因是这里在实现线程的调度时，用的是时钟中断。因此利用时钟中断实现互斥也是可以的。有关时钟中断的相关函数在 `include/interrupt.h` 中定义。

②关闭中断后，申请资源PCB，如果申请不到，则返回-1；否则分配到的PCB进行清零操作。

③初始化该线程的信息。包括线程名称，优先级，状态，线程id，线程总时间片和已执行时间。

④初始化线程栈。线程栈的初始地址是PCB的起始地址加上PCB_SIZE。线程的栈是从PCB的顶部开始向下增长的。在栈中放入7个整数值。

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

⑤创建完线程后，我们将该线程放入 `allPrograms` 和 `readyPrograms` 中链表中，等待时钟中断来的时候，这个新创建的线程就可以被调度上处理器。

⑥恢复原来的中断状态。

如下图：

```
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;
```

```

// 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;
}

```

Assignment3 线程调度切换的秘密

这里实现的是时间片轮转算法 (Round Robin, RR)，即当时钟中断到来时，对当前线程的 `ticks` 减1，直到 `ticks` 等于0，然后执行线程调度。线程的调度在函数 `schedule` 中实现。`schedule` 主要完成以下事情：

```

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));
    }
    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);
}

```

①在调度之前要关中断，以实现互斥；

②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；如果是终止态(DEAD)，则回收线程的PCB。

③找就绪队列的第一个线程作为下一个执行的线程。就绪队列的第一个元素是 `Listitem *` 类型的，需要将其转换为 `PCB`。注意到放入就绪队列 `readyPrograms` 的是每一个PCB的 `&tagInGeneralList`，而 `tagInGeneralList` 在PCB中的偏移地址是固定的。即将 `item` 的值减去 `tagInGeneralList` 在PCB中的偏移地址就能够得到PCB的起始地址。我们将上述过程写成一个宏。于是定义一个宏 `#define ListItem2PCB(ADDRESS, LIST_ITEM) ((PCB *)((int)(ADDRESS) - (int)&((PCB *)0)->LIST_ITEM))` 实现。其中，`(int)&((PCB *)0)->LIST_ITEM` 求出的是 `LIST_ITEM` 这个属性在PCB中的偏移地址。

④删除就绪队列的第一个线程，并将其状态设置为运行态和当前正在执行的线程。

⑤最后，将线程从 `cur` 切换到 `next`。线程的所有信息都在线程栈中，所以只要切换线程栈就能够实现线程的切换，线程栈的切换实际上就是将线程的栈指针放到 `esp` 中。栈的切换在 `asm_switch_thread` 函数中完成。

```

asm_switch_thread:
    push ebp
    push ebx
    push edi
    push esi

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

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

    pop esi
    pop edi
    pop ebx
    pop ebp

    sti
    ret

```

①先保存寄存器 `ebp`，`ebx`，`edi`，`esi`，因为C语言要求被调函数主动为主调函数保存这4个寄存器的值。如果不遵循这个规则，后面线程切换到C语言编写的代码时就会出错。

②保存 `esp` 的值到线程的 `PCB::stack` 中，用做下次恢复。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`。

③将 `next->stack` 的值写入到 `esp` 中，从而完成线程栈的切换。首先将 `next->stack` 的地址放到 `eax` 中，再向 `esp` 中写入 `[eax]`。

④对于被换下处理器的线程现在又被调度，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。

对于刚创建还未调度运行的线程，`pop` 语句会将4个0值放到 `esi`，`edi`，`ebx`，`ebp` 中。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`。

⑤ `program_exit` 如下：

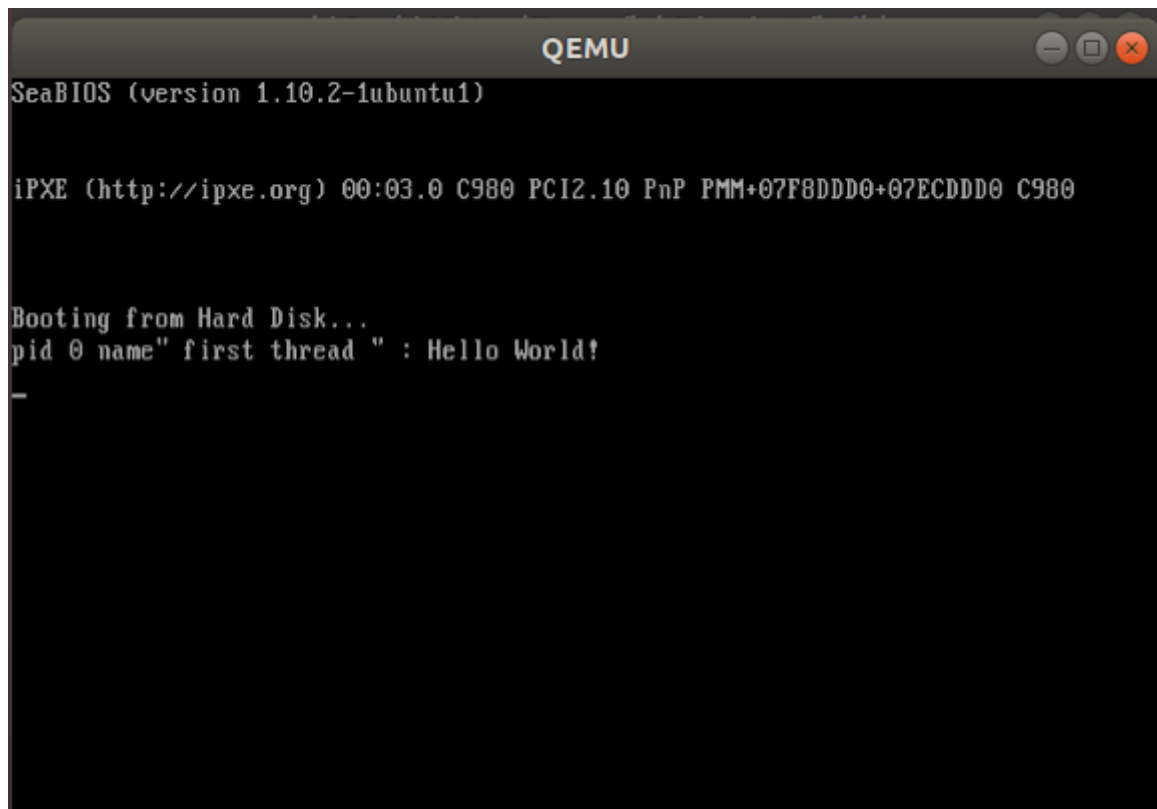
```
void program_exit()
{
    PCB *thread = programManager.running;
    thread->status = ThreadStatus::DEAD;

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

`program_exit` 会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。第一个线程是不可以返回的，这个线程的 `pid` 为0。

⑤这样，通过 `asm_switch_thread` 中的 `ret` 指令和 `esp` 的变化，我们便实现了线程的调度。

第一个线程，执行结果：



用gdb调试，查看

1、一个新创建的线程是如何被调度然后开始执行的。

(1) 先只创建一个线程，查看该线程是如何被调度并开始执行的。

在 `executeThread` 函数中我们创建了一个线程，为线程申请了一个PCB，为线程分配了一个唯一的id，设置该线程的状态为 `ready`，然后指定了线程要执行的函数，此时线程等待被调度。调度由 `asm_switch_thread` 函数完成。跟踪该函数运行时，记录寄存器的变化情况：

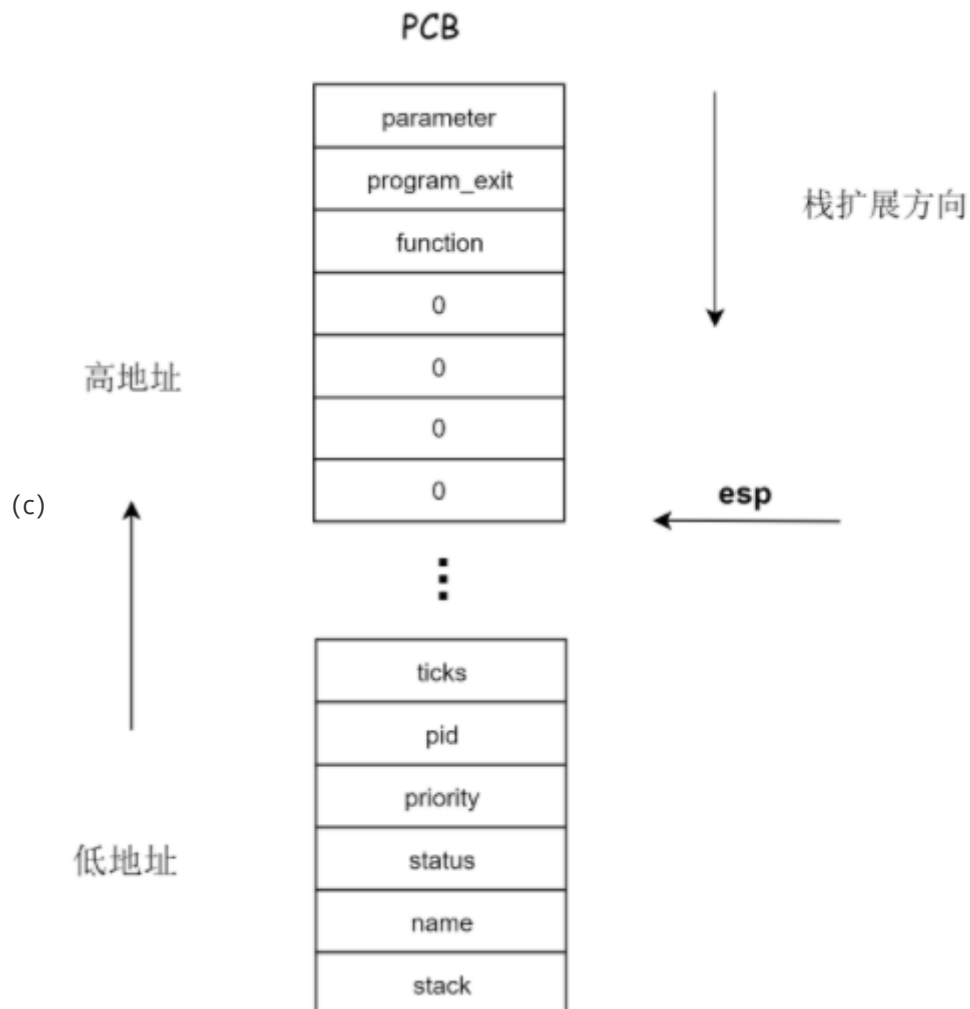
时间	esp	eip	ebp	eax	注释
开始	0x7bd0	0x21470	0x7bfc	0x21da0	
push ebp 后	0x7bcc	0x21471	0x7bfc	0x21da0	保存寄存器ebp
push ebx 后	0x7bc8	0x21472	0x7bfc	0x21da0	保存寄存器ebx
push edi 后	0x7bc4	0x21473	0x7bfc	0x21da0	保存寄存器edi
push esi 后	0x7bc0	0x21474	0x7bfc	0x21da0	保存寄存器esi
mov eax, [esp+5*4] 后	0x7bc0	0x21478	0x7bfc	0	将cur->stack的地址放到eax 中, PCB::stack 在 PCB 的偏 移地址是0, 所以此时eax的值 为0.
mov [eax], esp 后	0x7bc0	0x2147a	0x7bfc	0	向[eax]中写入esp的值, 下图 (a), 显示写入esp后, [eax]所存的值。到此, 已经将 esp的值存到了PC::stack中。
mov eax, [esp+6*4]	0x7bc0	0x2147e	0x7bfc	0x21da0	将 next->stack 的地址存入 eax
mov esp, [eax]	0x22d84	0x21480	0x7bfc	0x21da0	然后再将[eax]赋给esp, 这样 就将 next->stack 的值写入到 esp中。下图 (b) 可以看到 0x21da0的值为0x22d84
pop esi	0x22d88	0x21481	0x7bfc	0x21da0	esi变为0
pop edi	0x22d8c	0x21482	0x7bfc	0x21da0	edi变为0
pop ebx	0x22d90	0x21483	0x7bfc	0x21da0	ebx变为0
pop ebp	0x22d94	0x21484	0x0	0x21da0	ebp变为0, 四个pop完之后, 栈顶的数据是线程需要执行的 函数的地址 function。如图 (c)
sti	0x22d94	0x21485	0x0	0x21da0	允许中断
ret	0x22d98	0x2050c	0x0	0x21da0	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。图 (d) 查看栈顶数据。

```

remote Thread 1 In: first thread
(a) (gdb) x/1xw 0x0
0x0: 0x00007bc0
(gdb)

(gdb) x/1xw 0x21da0
(b) 0x21da0 <PCB_SET+96>: 0x00022d84
(gdb)

```



```

(gdb) x/1xw 0x22d94
(d) 0x22d94 <PCB_SET+4180>: 0x0002050c
(gdb)

```

这样，第一个新创建的线程就被调度执行了。（图片说明结合表格注释）

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。

创建4个进程。第一个线程在 `void setup_kernel()` 中创建，剩下的线程在第一个线程的执行函数中创建，即剩下的线程由第一个线程创建。如下图：

创建第一个线程并执行：

```

75     int pid = programManager.executeThread( first_thread, nullptr, "first thread", 1);
76
77
78     //fail
79     if( pid == -1 )
80     {
81         printf( "can not execute thread\n" );
82         asm_halt();
83     }
84
85     ListItem *item = programManager.readyPrograms.front();
86     PCB *firstThread = ListItem2PCB(item,tagInGeneralList);
87     firstThread->status = RUNNING;
88     programManager.readyPrograms.pop_front();
89     programManager.running = firstThread;
90     asm_switch_thread(0,firstThread);
91     asm_enable_interrupt();

```

在第一个线程中创建其他线程：

```

25 void first_thread( void *arg )
26 {
27     printf( "pid %d name\ " %s\ " : Hello World! \n", programManager.running->pid,programManager
28     //create the other threads in the first thread!!!
29     if( !programManager.running->pid )
30     {
31         int pid2 = programManager.executeThread( second_thread, nullptr, "second thread", 2);
32
33         int pid3 = programManager.executeThread( third_thread, nullptr, "third thread", 3);
34         int pid4 = programManager.executeThread( third_thread, nullptr, "4th thread", 4);
35         //fail
36         if( pid2 == -1 )
37         {
38             printf( "can not execute thread\n" );
39             asm_halt();
40         }
41
42         if( pid3 == -1 )
43         {
44             printf( "can not execute thread\n" );
45             asm_halt();
46         }
47     }

```

其余线程函数分别为：

```

13 void third_thread( void *arg ) {
14     printf( "pid %d name\ " %s\ " : Hello World! \n", programManager .running->pid, programManage
15     while(1){
16         printf( "%d\n", programManager .running->pid);
17     }
18 }
19
20 void second_thread( void *arg ) {
21     printf( "pid %d name\ " %s\ " : Hello World! \n", programManager.running->pid,programManager
22     //return;
23 }
24

```

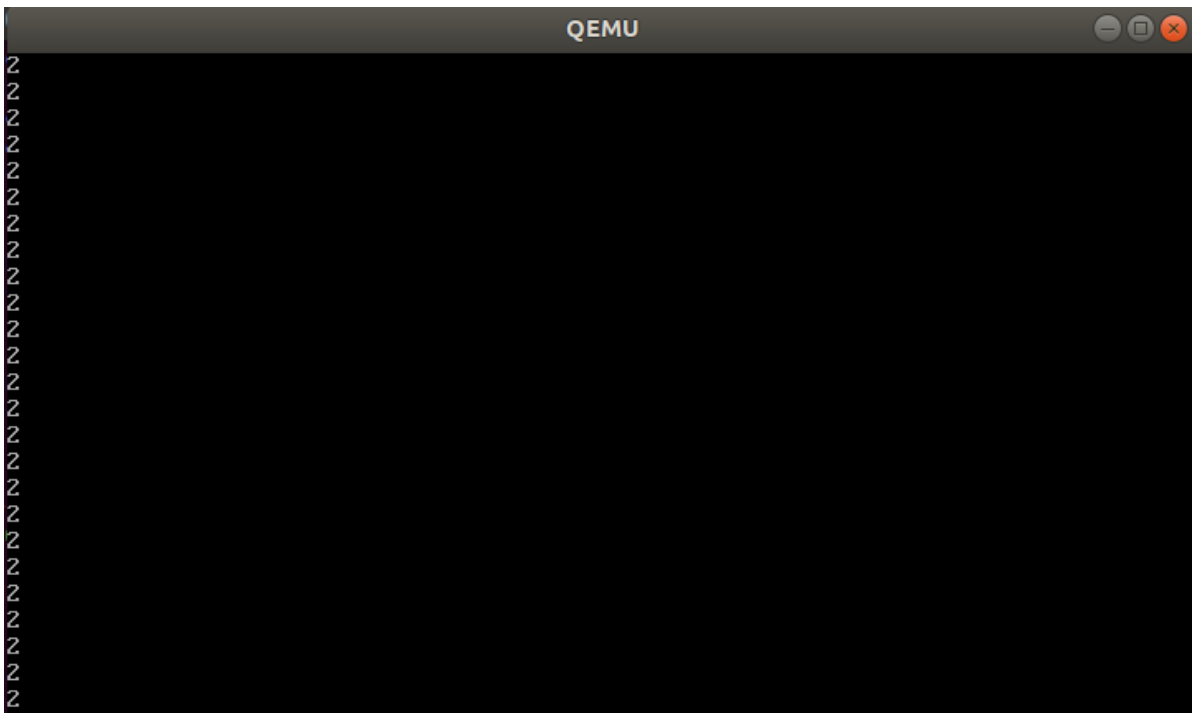
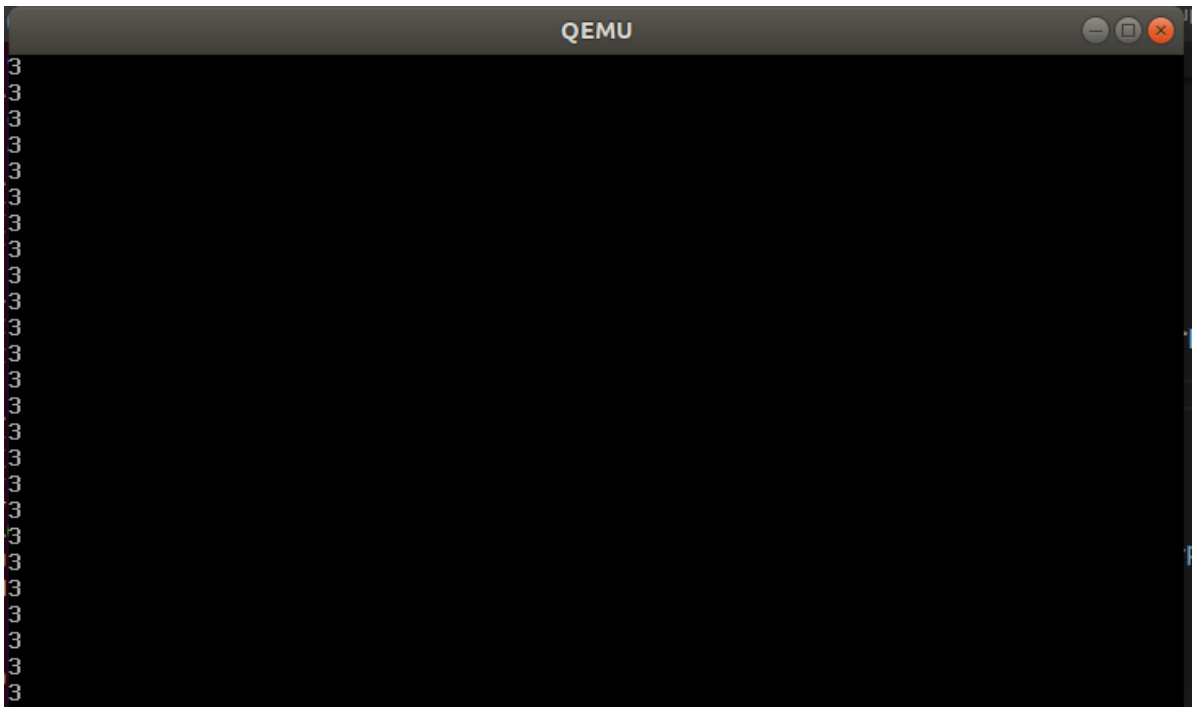
其中注意到 `third_thread` 中有个while循环。而第三个和第四个线程用的都是 `third_thread`。

运行结果：

刚开始的时候会输出各个线程的id和线程名称，到了第三个线程（线程id为2）和第四个线程（线程id为3）会开始只输出大量的线程id。

[illegible]

```
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2  
2pid 3 name " 4th thread": Hello World!  
3_
```



设置好断点：

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

[ No Source Available ]

> 0xffff0 add BYTE PTR [eax],al
0xffff2 add BYTE PTR [eax],al
0xffff4 add BYTE PTR [eax],al
0xffff6 add BYTE PTR [eax],al
0xffff8 add BYTE PTR [eax],al
0xffffa add BYTE PTR [eax],al

remote Thread 1 In: L?? PC: 0xffff0
0x0000ffff0 in ?? ()
The target architecture is assumed to be i386
(gdb) b program.cpp:118
Breakpoint 1 at 0x203b9: file ../src/kernel/program.cpp, line 118.
(gdb) b interrupt.cpp:92
Breakpoint 2 at 0x2096e: file ../src/kernel/interrupt.cpp, line 92.
(gdb) layout split
(gdb)
```

开始执行:

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

../src/kernel/interrupt.cpp
90 // 中断处理函数
91 void c_time_interrupt_handler()
B+> 92 {
93     PCB *cur = programManager.running;
94
95     if (cur->ticks)

B+> 0x2096e <c_time_interrupt_handler()> push ebp
0x2096f <c_time_interrupt_handler()+1> mov ebp,esp
0x20971 <c_time_interrupt_handler()+3> sub esp,0x18
0x20974 <c_time_interrupt_handler()+6> mov eax,ds:0x31e90
0x20979 <c_time_interrupt_handler()+11> mov DWORD PTR [ebp-0xc],eax
0x2097c <c_time_interrupt_handler()+14> mov eax,DWORD PTR [ebp-0xc]

remote Thread 1 In: c_time_interrupt_handler L92 PC: 0x2096e
Continuing.

Breakpoint 1, ProgramManager::schedule (this=0x31e80)
at ../src/kernel/program.cpp:118
Continuing.

Breakpoint 2, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:92
(gdb)
```

可以看到程序执行过程中在这两个函数来回切换，函数 `c_time_interrupt_handler` 对线程时间片进行计数，若超时，则返回 `schedule` 函数进行调度。`schedule` 若发现当前线程的执行时间超时，则重新更新该线程的已执行时间为0，并暂时将该线程由 `running` 状态变为 `ready` 状态，让其他已经在 `ready` 队列中的线程先执行，这样一个正在执行的线程就被中断然后被换下处理器。

若线程执行完没有超时，则 `asm_switch_thread` 执行完线程的函数后，跳转到 `program_exit()` 函数，将该线程状态设置为 `DEAD`，然后再返回函数 `schedule`，对状态为 `DEAD` 的函数进行“清理”（释放掉该线程申请的PCB，出队等）。

函数 `asm_switch_thread` 之所以能够跳转到 `program_exit()`，是因为线程PCB的栈中栈顶是 `function`，紧接着 `function` 的是 `program_exit` 的地址，因此执行完 `function` 后，寄存器 `eip` 取的是 `program_exit` 地址的值。

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

../src/kernel/setup.cpp
18     }
19
20     void second_thread( void *arg ) {
> 21         printf( "pid %d name\" %s \" : Hello World! \n", programManager
22             //return;
23     }

0x204e1 <second_thread(void*)>      push    ebp
0x204e2 <second_thread(void*)+1>    mov     ebp,esp
0x204e4 <second_thread(void*)+3>    sub     esp,0x8
> 0x204e7 <second_thread(void*)+6>    mov     eax,ds:0x31e90
0x204ec <second_thread(void*)+11>   lea     edx,[eax+0x4]
0x204ef <second_thread(void*)+14>   mov     eax,ds:0x31e90

remote Thread 1 In: second_thread      L21  PC: 0x204e7
0x00021511 in asm_switch_thread ()
0x00021512 in asm_switch_thread ()
0x00021513 in asm_switch_thread ()
0x00021514 in asm_switch_thread ()
0x00021515 in asm_switch_thread ()
second_thread (arg=0x0) at ../src/kernel/setup.cpp:20
(gdb) s
(gdb)
```

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

../src/kernel/program.cpp
122
123     void program_exit()
> 124     {
125         PCB *thread = programManager.running;
126         thread->status = ProgramStatus::DEAD;
127
> 0x203e1 <program_exit()>      push    ebp
0x203e2 <program_exit() +1>      mov     ebp,esp
0x203e4 <program_exit() +3>      sub     esp,0x18
0x203e7 <program_exit() +6>      mov     eax,ds:0x31e90
0x203ec <program_exit() +11>     mov     DWORD PTR [ebp-0xc],eax
0x203ef <program_exit() +14>     mov     eax,DWORD PTR [ebp-0xc]

remote Thread 1 In: program_exit      L124  PC: 0x203e1
(gdb) c
Continuing.

Breakpoint 2, second_thread (arg=0x0) at ../src/kernel/setup.cpp:23
(gdb) si
program_exit () at ../src/kernel/program.cpp:124
(gdb)
```

```

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

../src/kernel/program.cpp
106         else if (running->status == ProgramStatus::DEAD)
107         {
> 108             releasePCB(running);
109         }
110
111         ListItem *item = readyPrograms.front();

0x20350 <ProgramManager::schedule()+184>    mov     eax,DWORD PTR [eax+0
0x20353 <ProgramManager::schedule()+187>    cmp     eax,0x4
0x20356 <ProgramManager::schedule()+190>    jne     0x2036d <ProgramMana
> 0x20358 <ProgramManager::schedule()+192>    mov     eax,DWORD PTR [ebp+0
0x2035b <ProgramManager::schedule()+195>    mov     eax,DWORD PTR [eax+0
0x2035e <ProgramManager::schedule()+198>    sub     esp,0x8

remote Thread 1 In: ProgramManager::schedule          L108  PC: 0x20358
    at ../src/kernel/program.cpp:92
List::size (this=0x31e88) at ../src/utils/list.cpp:14
0x000202d0 in ProgramManager::schedule (this=0x31e80)
    at ../src/kernel/program.cpp:94
(gdb) si
(gdb) si
(gdb)

```

然后，将 ready 队列中的第一个线程设置为下一个running，然后将该线程pop出 ready 队列，然后跳转到 asm_switch_thread 执行当前running线程的函数。同时 c_time_interrupt_handler 会一直保持计数。

对于被换下处理器的线程现在又被调度，其过程是在 schedule 的 asm_switch_thread 中执行完四个 pop之后，之前保存在线程栈中的内容会被恢复到这4个寄存器中，然后又回到函数 schedule，然后在 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。

```

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

Register group: general
eax      0xb8020  753696          ecx      0xb89cb  756171
edx      0xb892b  756011          ebx      0x0      0
esp      0x24d64  0x24d64 <PCB_SET+12132>    ebp      0x24da4  0x24da4 <PCB_SET+12196>
esi      0x0      0              edi      0x0      0
ebp      0x20986  0x20986 <c_time_interrupt_handler()>  eflags    0x2      [ ]
cs       0x20     32              ss       0x10     16
ds       0x8      8              es       0x8      8
fs       0x0      0              gs       0x18     24

B> 0x20986 <c_time_interrupt_handler()> push    ebp
0x20987 <c_time_interrupt_handler()+1> mov     ebp,esp
0x20989 <c_time_interrupt_handler()+3> sub     esp,0x18
0x2098c <c_time_interrupt_handler()+6> mov     eax,ds:0x31eb0
0x20991 <c_time_interrupt_handler()+11> mov     DWORD PTR [ebp-0xc],eax
0x20994 <c_time_interrupt_handler()+14> mov     eax,DWORD PTR [ebp-0xc]
0x20997 <c_time_interrupt_handler()+17> mov     eax,DWORD PTR [eax+0x24]
0x2099a <c_time_interrupt_handler()+20> test    eax,eax
0x2099c <c_time_interrupt_handler()+22> je      0x209be <c_time_interrupt_handler()+56>

remote Thread 1 In: c_time_interrupt_handler          L92  PC: 0x20986
Breakpoint 1, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:92
(gdb) s
0x0002162e in asm_time_interrupt_handler ()
Single stepping until exit from function asm_time_interrupt_handler,
which has no line number information.
STDIO:rollUp (this=0x31e90) at ../src/kernel/stdio.cpp:106
(gdb) c
Continuing.

Breakpoint 1, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:92
(gdb) layout regs
(gdb)

```

```
终端
星期二 21:54
zh
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

Register group: general
eax 0x23e80 147072 ecx 0xb89cb 756171
edx 0x8 8 ebx 0x0 0
esp 0x24d68 0x24d68 <PCB_SET+12136> ebp 0x24da4 0x24da4 <PCB_SET+12196>
esi 0x0 0 edi 0x0 0
eip 0x2162e 0x2162e <asm_time_interrupt_handler+12> eflags 0x6 [ PF ]
cs 0x20 32 ss 0x10 16
ds 0x8 8 es 0x8 8
fs 0x0 0 gs 0x18 24

0x21627 <asm_time_interrupt_handler+5> out 0xa0,al
0x21629 <asm_time_interrupt_handler+7> call 0x20986 <c_time_interrupt_handler()>
> 0x2162e <asm_time_interrupt_handler+12> popa
0x2162f <asm_time_interrupt_handler+13> iret
0x21630 <asm_halt> jmp 0x21630 <asm_halt>
0x21632 add BYTE PTR [eax],al
0x21634 <_ZL8PCB_SIZE> add BYTE PTR [eax],dl
0x21636 <_ZL8PCB_SIZE+2> add BYTE PTR [eax],al
0x21638 push 0xa746c61

remote Thread 1 In: asm_time_interrupt_handler L?? PC: 0x2162e
which has no line number information.
STDIO::rollUp (this=0x31e90) at ../src/kernel/stdio.cpp:106
(gdb) c
Continuing.

Breakpoint 1, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:92
(gdb) layout regs
(gdb) si
(gdb) s
(gdb) s
0x0002162e in asm_time_interrupt_handler ()
(gdb)
```

时钟中断先保存 `schedule` 中的状态信息，然后回到之前被中断的地方开始执行。执行完后又回到 `schedule`。如此反复，直到该线程时间片用完，又切换到下一个线程。

Assignment4 调度算法的实现

1、先到先服务调度——FCFS(FIFO)算法

实现：

```
77 void ProgramManager::FIFO()
78 {
79     bool status = interruptManager.getInterruptStatus();
80     interruptManager.disableInterrupt();
81     if (readyPrograms.size() == 0)
82     {
83         interruptManager.setInterruptStatus(status);
84         return;
85     }
86
87     releasePCB(running);
88
89     ListItem *item = readyPrograms.front();
90     PCB *next = ListItem2PCB(item, tagInGeneralList);
91     PCB *cur = running;
92     next->status = ProgramStatus::RUNNING;
93     running = next;
94     readyPrograms.pop_front();
95
96     asm_switch_thread(cur, next);
97
98     interruptManager.setInterruptStatus(status);
99
100 }
```

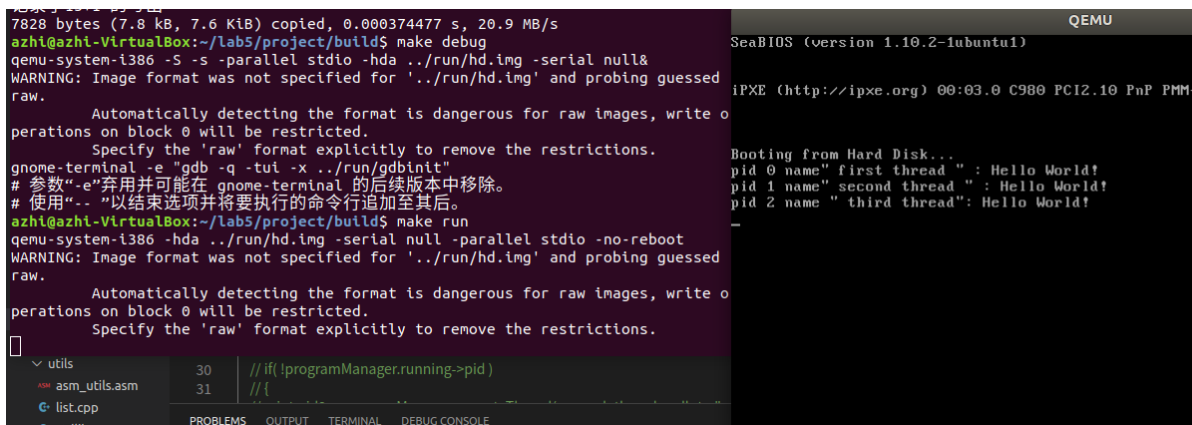
这里只有在每个程序执行完功能函数后，执行 `program.exti` 后状态变为 `DEAD` 才会调用 `FIFO` 函数进行调度。因此使用 `FIFO` 调度时，不能在一个线程中创建另一个线程，因为在父线程执行完之前，子线程是不能执行的，而子线程没有执行，父线程就不会结束，此时就会进入死锁状态。因此这里的线程都在 `setup.cpp` 中创建。如下：

```
77     int pid = programManager.executeThread( first_thread, nullptr, "first thread", 1);
78
79
80     //fail
81     if( pid == -1 )
82     {
83         printf( "can not execute thread\n" );
84         asm_halt();
85     }
86
87
88     int pid2= programManager.executeThread( second_thread, nullptr, "second thread", 2);
89
90     int pid3= programManager.executeThread( third_thread, nullptr, "third thread", 3);
91     int pid4= programManager.executeThread( third_thread, nullptr, "4th thread", 4);
92     //fail
93     if( pid2 == -1 )
94     {
95         printf( "can not execute thread\n" );
96         asm_halt();
97     }
```

值得注意的是 `third_thread` 中有一个 `while(1){}` 循环，即线程 `third thread` 和 `4th thread` 只要执行了就不会退出。因此，按照 `FIFO` 的思想，这里的 `4th thread` 永远不会被执行，因为 `third thread` 永远不会执行完。

```
13 void third_thread( void *arg ){
14     printf( "pid %d name \" %s\": Hello World! \n", programManager .running->pid, programManager.running->name);
15     while(1){
16         //printf( "%d\n", programManager .running->pid);
17     }
18 }
```

查看运行结果：



```
7828 bytes (7.8 kB, 7.6 KiB) copied, 0.000374477 s, 20.9 MB/s
azhi@azhi-VirtualBox:~/lab5/project/build$ make debug
qemu-system-i386 -S -s -parallel stdio -hda ../run/hd.img -serial null&
WARNING: Image format was not specified for '../run/hd.img' and probing guessed
raw.
Automatically detecting the format is dangerous for raw images, write operations
on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
gnome-terminal -e "gdb -q -tui -x ../run/gdbinit"
# 参数"-e"弃用并可能在 gnome-terminal 的后续版本中移除。
# 使用"--"以结束选项并将要执行的命令追加至其后。
azhi@azhi-VirtualBox:~/lab5/project/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
raw.
Automatically detecting the format is dangerous for raw images, write operations
on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM
Booting from Hard Disk...
pid 0 name " first thread " : Hello World!
pid 1 name " second thread " : Hello World!
pid 2 name " third thread": Hello World!
-
30 // if( !programManager.running->pid)
31 // {
```

由输出信息可以看到线程的执行顺序符合 `FIFO` 思想，`4th thread` 没有执行，也说明 `FIFO` 调度实现成功。

2、优先级调度

算法思想：

根据线程优先级选择优先级高的先执行。这里结合了时间片轮转算法的实现。若一个线程正在执行，即使现在来了一个比该线程优先级更高的线程，也必须等该线程执行完或者时间片用完才能执行次优先级更高的线程。

实现方法：

主要在 `readyPrograms` 队列中对新进来的线程进行一个有序的入队操作，优先级高的排在队列前面，优先级低的排在较后。通过一个 `int findPos(PCB* my)` 函数找到线程 `my` 在 `readyPrograms` 队列中的位置，然后将线程插入该位置。若优先级相等的，就插在后面，即服从先到先服务原则。

```
146 int ProgramManager::findPos( PCB* my )
147 {
148     int pri = my->priority;
149     int pos = 0;
150     for( pos = 0; pos < readyPrograms.size(); pos++ )
151     {
152         ListItem *item = readyPrograms.at( pos );
153         PCB *cur = ListItem2PCB(item, tagInGeneralList);
154         if( pri > cur->priority )
155             return pos;
156     }
157     return pos;
158 }
```

测试：

这里创建了四个线程。其中在第一个线程里创建了三个线程。如下：

```
83
84 int pid = programManager.executeThread( first_thread, nullptr, "first thread", 1 );
85
86
87 //fail
88 if( pid == -1 )
89 {
90     printf( "can not execute thread\n" );
91     asm_halt();
92 }
93
```

```
29
30 void first_thread( void *arg )
31 {
32     printf( "pid %d name\ " %s \ " : Hello World! \n", programManager.running->pid, programManager.running->name );
33     printf( "my priority is %d \n", programManager.running->priority );
34     //printf( "my ticks is %d \n", programManager.running->ticks );
35     //create the other threads in the first thread!!!
36     if( !programManager.running->pid )
37     {
38         int pid2 = programManager.executeThread( fun, nullptr, "second thread", 2 );
39
40         int pid3 = programManager.executeThread( fun, nullptr, "third thread", 3 );
41         int pid4 = programManager.executeThread( fun, nullptr, "4th thread", 4 );
42         //fail
43         if( pid2 == -1 )
44         {
45             printf( "can not execute thread\n" );
46             asm_halt();
47         }
48
49         if( pid3 == -1 )
50         {
51             printf( "can not execute thread\n" );
52         }
53     }
54 }
```

线程优先级从高到低排列。线程功能函数实现如下：


```

13 void fun( void *arg ){
14     printf( "pid %d name \" %s\": Hello World! \n", programManager .running->pid, programManager.running->name);
15     printf( "my priority is %d \n", programManager.running->priority);
16 }

```

运行结果：

```

SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDD0+07ECDDD0 C980

Booting from Hard Disk...
pid 0 name " first thread " : Hello World!
my priority is 1
pid 3 name " 4th thread": Hello World!
my priority is 4
pid 2 name " third thread": Hello World!
my priority is 3
pid 1 name " second thread": Hello World!
my priority is 2

```

可以看到输出结果中，除了第一个线程，其他线程按优先级从高到低排列。这是符合该调度算法的思想的。

实验感想：

此次实验难度较大。特别是在assignment3时，我遇到了很大的困难。困难之一是用makefile+gdb调试我不是很熟练，摸索了好久才知道大概该怎么操作；困难之二是线程的调度过程本身比较复杂，要弄清楚整个来龙去脉实属不易，特别是涉及到的文件和函数比较多，要弄清楚原理就必须熟悉各个函数的功能和函数间的关系要；困难之三是对汇编语言的函数调用过程不熟悉。

经过不懈努力，我一步一步地对线程的调度的过程有所了解，在此过程中，我不仅重新温习了一下上学期学过的与汇编相关的知识，也对时钟中断有更深入的了解，同时，我也掌握了makefile+gdb下调试的技能，也学会了如何通过查看寄存器的值的变化来观察函数的调用。最重要的是，我对基本的线程的调度算法有了比较牢固的记忆和理解。

遇到困难毫无头绪时是痛苦的，但是一步一步找到头绪，到最后能够凭自己的思路写简单的线程调度算法的过程是有成就感的。