

# 操作系统实验五报告

## 个人信息

- 姓名：李浩辉
- 学号：21307018

## 实验要求

- Assignment 1 学习可变参数，改进或者实现自己的printf函数
- Assignment 2 自行设计PCB，可以添加更多的属性，然后根据PCB来实现线程
- Assignment 3 编写若干个线程函数，使用gdb跟踪 `c_time_interrupt_handler`、`asm_switch_thread` 等函数
- Assignment 4 编写调度算法

## 实验过程

### Assignment 1

#### 实验要求

改进或者实现自己的printf函数

#### 实验原理

##### 可变参数函数

- 定义函数：  
需要在**最后**加 `...` 代表可变参数列表。  
并且前面**一定要有固定参数**

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

- 需要利用 `<stdarg.h>` 里面的函数定义，或者自己实现也可以。

```
// 一个没有特定类型的指针  
va_list
```

```
// 让ap指向第一个可变参数
```

```
// last_arg是最后一个固定参数
va_start(va_list ap, last_arg)

// 返回type (*ap), 并且让ap指向下一个可变参数
va_arg(va_list ap, type)

// ap=0 即清零
va_end(va_list ap)
```

这里要注意：我们实现的操作系统在保护模式下（32bit）所以需要4字节对齐。  
即

$$n' = \frac{n + 4 - 1}{4} \times 4$$

所以自己实现只需要几行宏定义

```
#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)
```

如果要自己实现，编译的时候要加上 -m32 参数，因为我们是在32位模式下的代码

## 实验过程

我们利用上次实验已经定义的STDIO类来减少操作，这时候要加上一个print函数这个函数作用为输出当前字符串（不带%d等参数）

```
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)
                {
                    rollUp();
                }
            }
        }
    }
```

```

        }
        else
        {
            ++row;
        }
        moveCursor(row * 80);
        break;
    default:
        print(str[i]);
        break;
    }
}
return i;
}

```

接下来我们的printf函数思路就是：

- 需要一个buffer来存储要输出的字符串（已经将参数转为字符）
- 从左往右阅读，碰到'%'进行特别分析，转换参数
- 加入buffer字符如果满了就先输出当前部分

**根据情况分类完成printf( ),提前设置需要的函数如itos( )等。**

```

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':
    ...

```

## 尝试

### 改进当前的printf ( )

### 增加浮点数输出 (.f)

## 在/src/kernel/stdio.cpp中printf () 函数下面新增以下内容

主要思路就是将double/float分为整数部分和小数部分分别记录。

注意点：

- float 类型放进可变参数里面会被提前处理成double，所以要按double类型来处理float。如果强行va\_arg(ap,float)只会截取double前4个字节，产生错误！

```
case 'f':{
    double temp=va_arg(ap,double);
    if (temp<0)
    {
        counter += printf_add_to_buffer(buffer,'-',idx,BUF_LEN);
        temp=-temp;
    }
    int temp2=temp;
    itos(number, temp2, 10);
    for (int j = 0; number[j]; ++j)
    {
        counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
    }

    temp-=temp2;
    counter += printf_add_to_buffer(buffer,'.',idx,BUF_LEN);
    ftos(number,temp);
    for (int j = 0; number[j]; ++j)
    {
        counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
    }
    break;
}
```

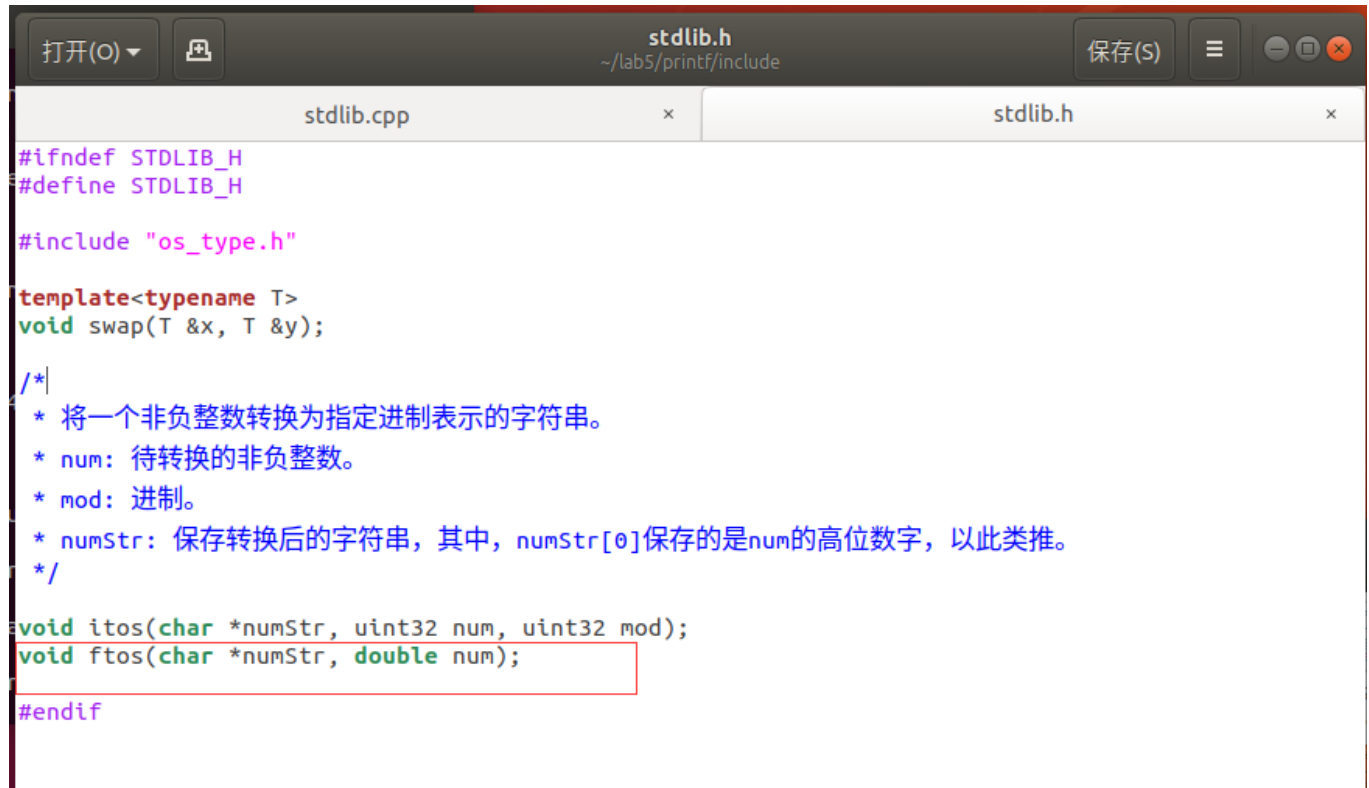
## 同时也要在/src/utils/新增ftos()函数

只是简单实现，所以并未能尽善尽美，无脑转为4位小数同时一些精度会有损失。

```
void ftos(char *numStr, double num){
    int temp;
    int len=0;
    double Num=num;
    for(len=0;len<4;len++){
        Num*=10;
        temp=Num;
```

```
        temp=temp%10;
        numStr[len]=temp+'0';
    }
    numStr[len]=0;
}
```

## 同时在/include/stdlib.h提前声明好



```
stdlib.h
~/lab5/printf/include

#ifndef STDLIB_H
#define STDLIB_H

#include "os_type.h"

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

/*
 * 将一个非负整数转换为指定进制表示的字符串。
 * num: 待转换的非负整数。
 * mod: 进制。
 * numStr: 保存转换后的字符串，其中，numStr[0]保存的是num的高位数字，以此类推。
 */

void itos(char *numStr, uint32 num, uint32 mod);
void ftos(char *numStr, double num);

#endif
```

## 效果展示

我们在src/kernel/setup.cpp增加检验输出

```
打开(O)  setup.cpp 保存(S)
~/lab5/printf/src/kernel

#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);
    double temp=2.33456;
    float temp2=5.1242;
    printf("%f\n%f\n",temp,temp2);
    //uint a = 1 / 0;
    asm_halt();
}
```

make & make run

虽然存在精度损失以及只能固定输出4位，但是也算简单实现了~

```
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
2.3345
5.1241
```

## Assignment 2

### 实验要求

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

## 实验原理

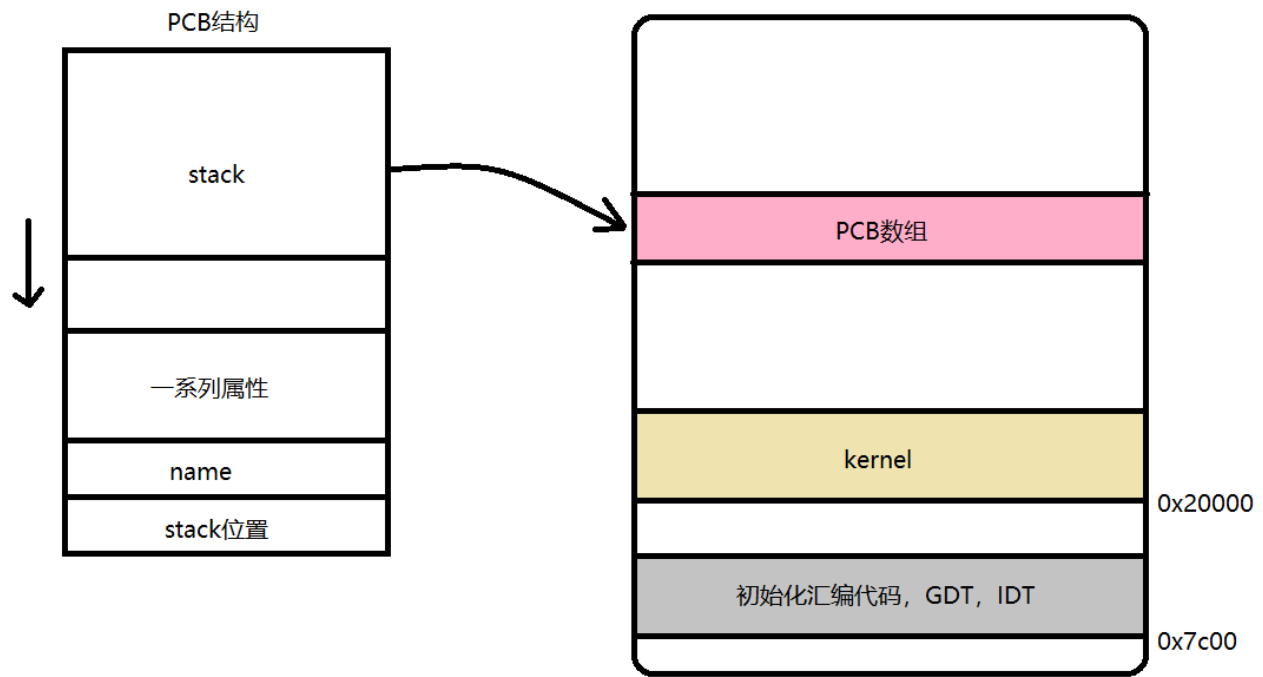
实现一个线程，需要一个数据结构PCB。

PCB应该有我们需要的属性，比如说线程栈、状态、优先级等。

```
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
};
```

现在我们在**内存中预留了一个PCB数组**用以存储所有PCB。

```
// 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];
```



将一切包装成一个统领全局的类。在编写内核的时候，我们就可以只依靠这个类的内容来完成线程的创建，释放等操作。

其中分配一个PCB，释放一个PCB不作赘述。

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

    // 创建一个线程并放入就绪队列
    // function: 线程执行的函数
    // parameter: 指向函数的参数的指针
    // name: 线程的名称
    // priority: 线程的优先级
    // 成功, 返回pid; 失败, 返回-1
    int executeThread(ThreadFunction function, void *parameter, const char *name,
int priority);

    // 分配一个PCB
```



```
PCB *allocatePCB();

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

};
```

分析一下executeThread( )

其代码主要分几部分：

- 分配一个PCB，这里用到allocatePCB( )。
- 设置好一些简单的属性，例如name,priority等
- 设置好stack内容
- 加入列表allPrograms以及readyPrograms。

**这里所有都需要有线程互斥，可以开关中断实现**

只分析其中一点

设置stack

这里stack先设置为线程PCB最后位置，然后在栈中加入一些东西。

```
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;
```

于是线程栈初始内容如下

parameter
program_exit
function
0
0
0
0

### 线程调度

我们用时钟中断完成RR算法

简单来说，就是让tick减到0就线程调度

```
extern "C" void c_time_interrupt_handler()
{
    PCB *cur = programManager.running;
    if (cur->ticks)
    {
        --cur->ticks;
        ++cur->ticksPassedBy;
    }
    else
    {
        programManager.schedule();
    }
}
```

线程调度代码如下

```

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);
}

```

这里需要提前定义一个宏

**这个宏作用就是根据listitem推出对应PCB的位置**

```

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

```

切换线程的汇编函数

```

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

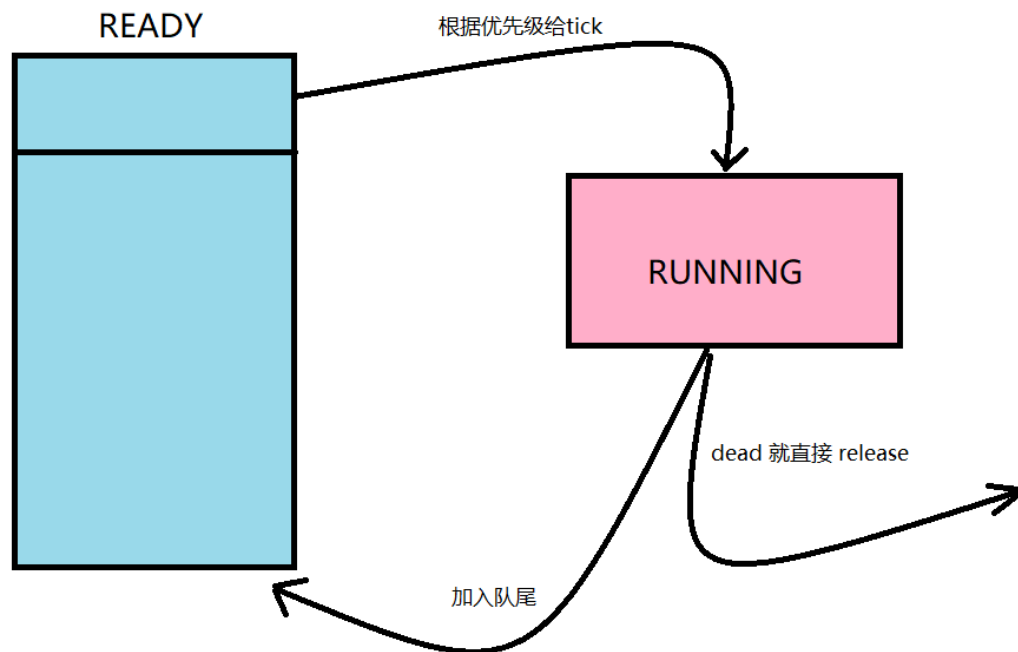
```

提前Push进来cur PCB 以及 next PCB地址, 然后调用该函数进行切换线程。

- 保存四个寄存器内容
- 保存当前栈指针到cur的stack中
- 切换线程栈
- 弹出四个寄存器内容, 这时候内容来自next栈

**为什么这样操作, 在Assignment3 中解答**

于是我们的调度就完成了, 效果如图



**实验结果**

编写一些函数放入线程，然后手动加入其中一个running

```
int pid = programManager.executeThread(first_thread, nullptr, "first thread", 1);
int pid1 = programManager.executeThread(second_thread, nullptr, "second thread",
1);
int pid2 = programManager.executeThread(third_thread, nullptr, "third thread", 1);

ListItem *item = programManager.readyPrograms.front();
PCB *firstThread = ListItem2PCB(item, tagInGeneralList);
firstThread->status = RUNNING;
programManager.readyPrograms.pop_front();
programManager.running = firstThread;
asm_switch_thread(0, firstThread);
```

make & make run



```
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": first Hello World!
pid 1 name "second thread": second Hello World!
pid 2 name "third thread": third Hello World!
```

## Assignment 3

### 实验要求

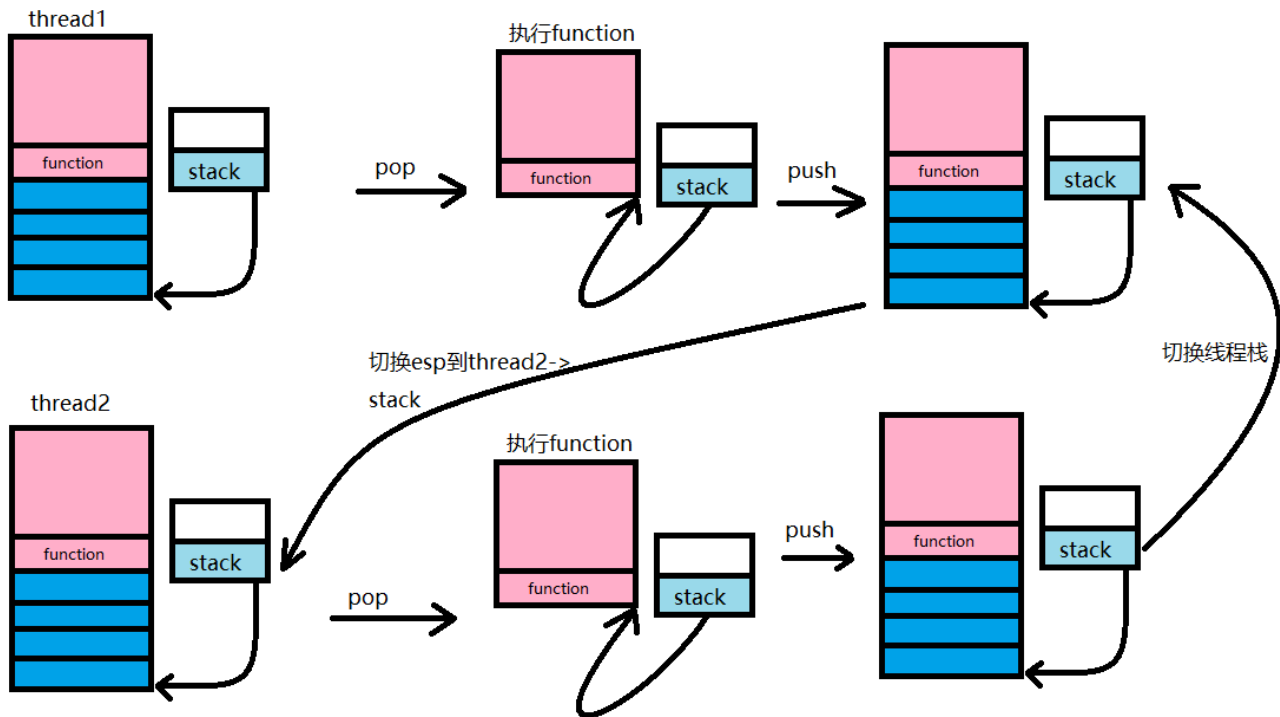
探究线程切换以及时钟中断完成RR调度的本质  
这里将可以解答我们的线程PCB stack 设计原理

### 实验过程

主要探究两点

- 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

给出一个线程切换大致流程图



我们直接用Assignment 2给出的模板以及调试工具gdb完成探究。

**准备好三个函数，三个函数均有死循环**

```
void third_thread(void *arg) {
    printf("pid %d name \"%s\": third Hello World!\n", programManager.running->pid,
programManager.running->name);
    while(1) {}
}
void second_thread(void *arg) {
    printf("pid %d name \"%s\": second Hello World!\n", programManager.running-
>pid, programManager.running->name);
    asm_halt();
}
void first_thread(void *arg)
{
    // 第1个线程不可以返回
    printf("pid %d name \"%s\": first Hello World!\n", programManager.running->pid,
programManager.running->name);
    asm_halt();
}
```

手动写入running 并进行一次switch\_thread即可开始运行

```

int pid0 = programManager.executeThread(first_thread, nullptr, "first thread",
1);
int pid1 = programManager.executeThread(second_thread, nullptr, "second
thread", 1);
int pid2 = programManager.executeThread(third_thread, nullptr, "third thread",
1);

ListItem *item = programManager.readyPrograms.front();
PCB *firstThread = ListItem2PCB(item, tagInGeneralList);
firstThread->status = RUNNING;
programManager.readyPrograms.pop_front();
programManager.running = firstThread;
asm_switch_thread(0, firstThread);

```

用gdb调试

第一个手动载入switch\_thread

我们在初始栈（起始位置是0x7c00）中装入了四个寄存器的内容

The screenshot shows a terminal window titled "终端" (Terminal). The top menu bar includes "文件(F)", "编辑(E)", "查看(V)", "搜索(S)", "终端(T)", and "帮助(H)". The main window displays assembly code from the file `../src/utils/asm_utils.asm`. The code includes comments and instructions for `asm_switch_thread`. A red box highlights the instructions `push ebp`, `push ebx`, `push edi`, and `push esi` on lines 24-27. Below the code, the GDB prompt `>` is followed by `29 mov eax, [esp + 5 * 4]` and `30 mov [eax], esp ; 保存当前栈指针到PCB中, 以便日后`. At the bottom, a GDB register dump for "remote Thread 1 In: asm\_switch\_thread" is shown, with the `esp` register highlighted by a red box. The register values are as follows:

Register	Value 1	Value 2
eax	0x21d20	138528
ecx	0x23d20	146720
edx	0x0	0
ebx	0x39000	233472
esp	0x7bb0	0x7bb0
ebp	0x7bfc	0x7bfc
esi	0x0	0

The terminal ends with the prompt `---Type <return> to continue, or q <return> to quit---`.

我们要切换的是PCB\_SET中第一个PCB板块中的第一个32bit对应的地址，即第一个线程对应的stack位置。

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

../src/utils/asm_utils.asm
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
31
32      mov eax, [esp + 6 * 4]
> 33      mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
34
35      pop esi
36      pop edi
37      pop ebx

remote Thread 1 In: asm_switch_thread L33 PC: 0x2147a
esi      0x0      0
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) p &PCB_SET
$1 = (char (*)[65536]) 0x21d20 <PCB_SET>
(gdb) info registers eax
eax      0x21d20  138528
(gdb)
```

接下来这里信息量很足，我们已经把esp切换到了0x22d04(这里就是thread1的stack)，然后我们即将弹出thread1的提前已经准备好的四个寄存器内容（初始化为0），以及我们要ret跳转到提前已经装载好的function (first\_thread())



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

../src/utils/asm_utils.asm
31
32     mov eax, [esp + 6 * 4]
33     mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
34
> 35     pop esi
36     pop edi
37     pop ebx
38     pop ebp
39
40     sti
41     ret
42     ; int asm_interrupt_status();
43     asm_interrupt_status:

remote Thread 1 In: asm_switch_thread L35 PC: 0x2147c
(gdb) info register esp
esp             0x22d04 0x22d04 <PCB_SET+4068>
(gdb) x/5w 0x22d04
0x22d04 <PCB_SET+4068>: 0      0      0      0
0x22d14 <PCB_SET+4084>: 132277
(gdb) info symbol 132277
first_thread(void*) in section .text
(gdb)
```

一路next我们就来到了预先设想的thread1  
这里就是一条输出语句和一个死循环

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

../src/kernel/setup.cpp
23     }
24     void first_thread(void *arg)
> 25     {
26         // 第1个线程不可以返回
27         printf("pid %d name \"%s\": first Hello World!\n", programManager->pid, programManager->name);
28         if (!programManager->running->pid)
29         {
30             //programManager->executeThread(second_thread, nullptr, "second");
31             //programManager->executeThread(third_thread, nullptr, "third");
32         }
33         asm_halt();
34     }
35
```

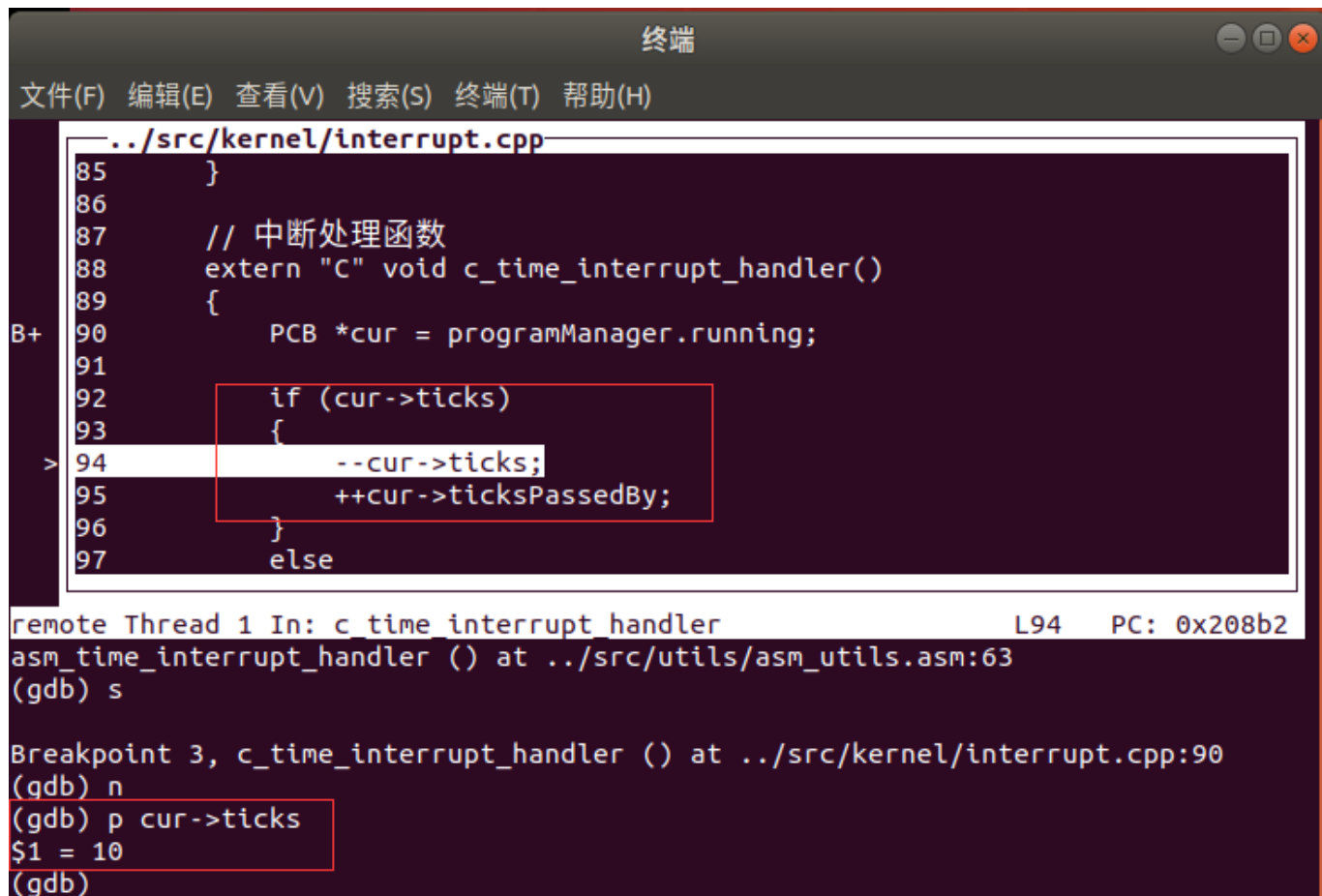
提前设置好时钟中断的断点，方便我们continue可以不会跑太远

```
(gdb) b 0x21490
Breakpoint 2 at 0x21490: file ../src/utils/asm_utils.asm, line 59.
(gdb) b ../src/kernel/interrupt.cpp:c_time_interrupt_handler()
Breakpoint 3 at 0x208a0: file ../src/kernel/interrupt.cpp, line 90.
```

就在我们准备printf的时候，我们突然跳转到了asm\_time\_interrupt\_handler。

**这是因为我们提前写好的时钟中断开始响应了**

我们的给thread1的tick一共有10次，这时候还没有跑完，所以这个中断只会给他tick减1。



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

./src/kernel/interrupt.cpp
85     }
86
87     // 中断处理函数
88     extern "C" void c_time_interrupt_handler()
89     {
90         PCB *cur = programManager.running;
91
92         if (cur->ticks)
93         {
94             --cur->ticks;
95             ++cur->ticksPassedBy;
96         }
97         else

remote Thread 1 In: c_time_interrupt_handler          L94   PC: 0x208b2
asm_time_interrupt_handler () at ./src/utils/asm_utils.asm:63
(gdb) s

Breakpoint 3, c_time_interrupt_handler () at ./src/kernel/interrupt.cpp:90
(gdb) n
(gdb) p cur->ticks
$1 = 10
(gdb)
```

中断完成，我们会继续thread1的内容，printf出第一条语句

同时我们到了死循环，期间我们会经过9次时钟中断



```
QEMU [Stopped]
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": first Hello World!
_
```

直到thread1的tick归零，我们将会来到schedule调度

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

../src/kernel/program.cpp
73     return thread->pid;
74 }
75
76 void ProgramManager::schedule()
77 {
B+> 78     bool status = interruptManager.getInterruptStatus();
79     interruptManager.disableInterrupt();
80
81     if (readyPrograms.size() == 0)
82     {
83         interruptManager.setInterruptStatus(status);
84         return;
85     }
```

因为我们thread1还在死循环中，所以永远都不会跑完。schedule就会把他tick按照优先级分配并且放到ready的队尾为了它下一次的调度上running。

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

../src/kernel/program.cpp
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         readyPrograms.push_back(&(running->tagInGeneralList));
92     }
93     else if (running->status == ProgramStatus::DEAD)
94     {
```

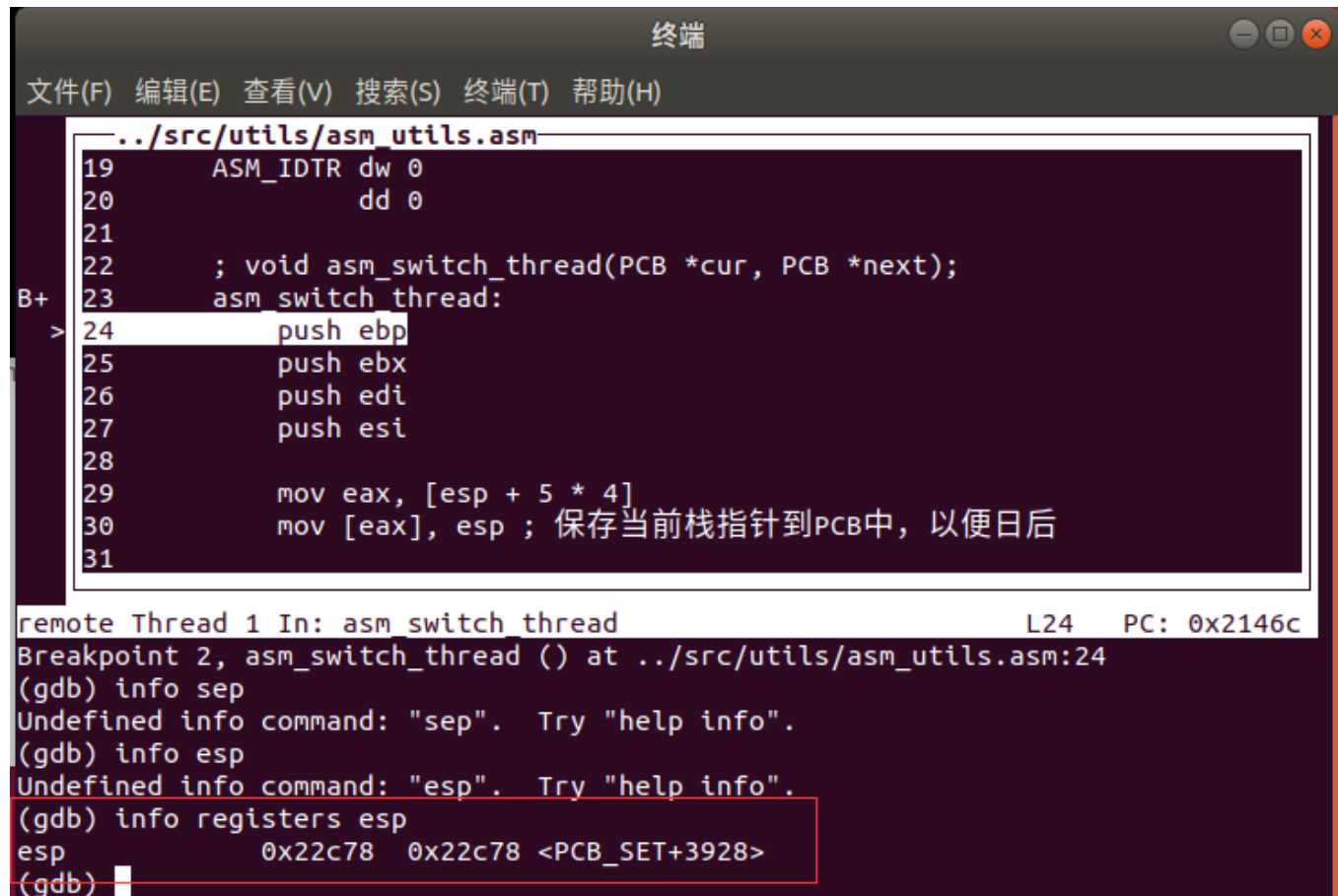
调整下各种参数，我们将会继续来到asm\_switch\_thread

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

../src/kernel/program.cpp
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
```

我们将会将目前的四个寄存器将会push进当前栈中（thread1的stack），同时改一下thread1->stack的指向改为当前esp的位置

实质上就是thread1的stack加入必要的寄存器，更改好stack指向



The screenshot shows a terminal window titled "终端" (Terminal) with a menu bar containing "文件(F)", "编辑(E)", "查看(V)", "搜索(S)", "终端(T)", and "帮助(H)". The main content is a GDB session. At the top, a window titled "../src/utils/asm\_utils.asm" displays assembly code from line 19 to 31. The code includes instructions for setting up a thread switch function, pushing registers (ebp, ebx, edi, esi) onto the stack, and saving the current stack pointer (esp) into a PCB structure. A comment in line 30 says "保存当前栈指针到PCB中，以便日后". Below the code window, the GDB console shows the execution of "asm\_switch\_thread" at address 0x2146c. It sets a breakpoint at line 24 of the same file. Subsequent GDB commands "info sep" and "info esp" result in "Undefined info command" errors. The command "info registers esp" is executed, and its output is highlighted with a red box: "esp 0x22c78 0x22c78 <PCB\_SET+3928>".

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

../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:
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
31

remote Thread 1 In: asm_switch_thread L24 PC: 0x2146c
Breakpoint 2, asm_switch_thread () at ../src/utils/asm_utils.asm:24
(gdb) info sep
Undefined info command: "sep". Try "help info".
(gdb) info esp
Undefined info command: "esp". Try "help info".
(gdb) info registers esp
esp      0x22c78 0x22c78 <PCB_SET+3928>
(gdb)
```

切换线程栈 (thread2的stack)

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

../src/utils/asm_utils.asm
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
31
32      mov eax, [esp + 6 * 4]
> 33      mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
34
35      pop esi
36      pop edi

remote Thread 1 In: asm_switch_thread L33 PC: 0x2147a
eip      0x21474 0x21474 <asm_switch_thread+8>
(gdb) n
(gdb) n
(gdb) info register eax
eax      0x22d20 142624
(gdb) info symbol 0x22d20
PCB_SET + 4096 in section .bss
(gdb)
```

同样的操作，将提前准备好的4个寄存器pop出来，以及ret跳转到对应的function

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

../src/utils/asm_utils.asm
31
32     mov eax, [esp + 6 * 4]
33     mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
34
> 35     pop esi
36     pop edi
37     pop ebx
38     pop ebp
39
40     sti
41     ret
42     ; int asm_interrupt_status();
43     asm_interrupt_status:

remote Thread 1 In: asm_switch_thread L35 PC: 0x2147c
(gdb) p/5w 0x23d04
Size letters are meaningless in "print" command.
(gdb) x/5w 0x23d04
0x23d04 <PCB_SET+8164>: 0      0      0      0
0x23d14 <PCB_SET+8180>: 132229
(gdb) info symbol 132229
second_thread(void*) in section .text
(gdb)
```

接下来，也是经过10个时钟中断切thread3

同理，thread3经过10个时钟中断切thread1，我们直接来看怎么切回thread1

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

这时候注意看，我们stack即将返回的函数地址已经是schedule里面的，为什么呢？

因为我们是**在schedule里面调用asm\_switch\_thread**，因此我们调用的时候就压入了返回地址

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

../src/utils/asm_utils.asm
31
32     mov eax, [esp + 6 * 4]
33     mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
34
> 35     pop esi
36     pop edi
37     pop ebx
38     pop ebp
39
40     sti
41     ret
42     ; int asm_interrupt_status();
43     asm_interrupt_status:

remote Thread 1 In: asm_switch_thread L35 PC: 0x2147c
(gdb) info registers esp
esp                0x22c68 0x22c68 <PCB_SET+3912>
(gdb) x/5w 0x22c68
0x22c68 <PCB_SET+3912>: 0      0      0      142500
0x22c78 <PCB_SET+3928>: 131947
(gdb) info symbol 131947
ProgramManager::schedule() + 303 in section .text
(gdb)
```

这时候schedule跑完就返回到调用者c\_time\_interrupt\_handler(), 然后同样返回到调用者asm\_time\_interrupt\_handler, 继续执行, 该次中断结束。

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

../src/utils/asm_utils.asm
59     asm_time_interrupt_handler:
60         pushad
61
62         ; 发送EOI消息, 否则下一次中断不发生
63         mov al, 0x20
64         out 0x20, al
65         out 0xa0, al
66
67         call c_time_interrupt_handler
68
69         popad
> 70     iret
71

remote Thread 1 In: asm time interrupt handler          L70   PC: 0x2149d
(gdb) info registers esp
esp             0x22cfc  0x22cfc  <PCB_SET+4060>
(gdb) x/5w 0x22cfc
0x22cfc <PCB_SET+4060>: 136555  32      518      132322
0x22d0c <PCB_SET+4076>: 0
(gdb) info symbol 136555
asm_halt in section .text
(gdb)
```

iret后就回来了thread1执行到的位置, 同时一切寄存器也恢复了

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

../src/utils/asm_utils.asm
180     mov [gs:2 * 10], ax
181
182     pop eax
183     ret
184
185     asm halt:
> 186     jmp $^?
187
188
189
190
191
192
```

## 实验总结

就实验样例给出的RR调度算法, 在PCB数据结构之上, 我们**通过时钟中断来计算时间**将ready队列和running的线程切换。

**时钟中断起到的作用就像是：**一个监工隔一段时间就来询问当前工人（当前线程）跑了多久，如



果到点了就让休息好的工人代替（ready队列中第一个）他来工作。当然，每次询问都要让工人停下手头工作（中断保护现场）。

## Assignment 4

### 实验要求

实现自己的调度算法

### 实验过程

#### FIFO调度算法

**算法描述：**这是一个最简单的实现，也是一个最贴近日常生活的做法。我们直接让先来的thread跑完，后面来的都排队

#### 实现

只需要更改下时钟中断，我们之前的时钟中断是一个计算频率次数来切换线程，这次我们时钟中断改为只要线程没跑完就一直让他跑。

**因为第一个线程我们设置为了不能跑完，所以加了一个或让他不能dead也可以切换**

```
// 中断处理函数
extern "C" void c_time_interrupt_handler()
{
    PCB *cur = programManager.running;
    if(cur->status == ProgramStatus::DEAD || cur->pid==0){
        programManager.schedule();
    }
}
```

同时我们的thread 除了thread1 可以有死循环，其他都不可以有死循环不然就一直卡在running出不来了。

效果如图



```
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": first Hello World!
pid 1 name "second thread": second Hello World!
pid 2 name "third thread": third Hello World!
_
```