

1 实验概述

- 学习可变参数机制, 然后实现 printf, 并在 example 的基础上扩展 %f, %.2f, %e 的功能。
- 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。
- 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。
- 编写一种调度算法 (我使用抢占式最短剩余时间 SRTN), 并编写测试样例呈现算法的正确性

2 实验过程

2.1 实验任务一

本实验中, 我扩展实现了 printf 函数的 %f, %.2f, %e 功能, 支持输出浮点数, 格式化浮点数, 以及科学计数法输出。主要在 `stdio.cpp` 和 `stdlib.cpp` 中进行了扩展。

2.1.1 实验步骤

Printf 函数基于 c 语言的可变参数的函数机制, 通过 `va_list`, `va_start`, `va_arg`, `va_end` 等宏来获取可变参数。`va_list` 是指向可变参数列表的指针, `va_start` 初始化指针, `va_arg` 获取可变参数, `va_end` 结束可变参数的获取。

而为了把内容输出到 qemu 显示屏, 需要先把内容存到 buffer 缓冲区中, 接着使用 Lab4 中封装好的 `stdio.h` 中的各种函数, 配合 printf 中传入的参数进行输出。

由于 `stdio.print` 只接受传入字符串, 因此在设计 printf 函数时的各种格式化输出都需要进行特殊处理, 转化为字符串, 因此在处理整数, 保留特定小数点的小数以及科学计数法时, 要写专门的函数进行处理。

%f 和 %.3f 的差异主要在于精度 (自定义和默认为 6), 因此可以通过 c++ 的 switch 的机制统一进行处理:

```
case '.':
case 'f': {
    int precision=0;
    if (fmt[i]=='.') {
        i++;
        for (precision=0;fmt[i]>='0'&&fmt[i]<='9';i++)
        {
            precision=precision*10+fmt[i]-'0';
        }
    } else precision=6;
    double temp = va_arg(ap, double);
    if (temp<0) { //处理负号的情况
        counter+= printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
        temp=-temp;
    }
    ftos(number,temp,precision);
    for (int j=0;number[j];++j)
    {
        counter+=printf_add_to_buffer(buffer,number[j],idx,BUF_LEN);
    }
    break;
}
```

而为了在 qemu 显示屏输出, 我们需要先把浮点数转化成对应精度的字符串, 此函数为 ftos, 在 stdlib.cpp 中实现, 代码逻辑在注释中给出:

```
void ftos(char *numStr, double num, int precision)
{
    int i=0;
    int intpart=(int)num; //整数部分
    double frac=num-intpart; //小数部分

    char intBuf[33];
    int int_idx=0;
    if (intpart==0) {
        intBuf[int_idx++]='0';
    } else {
        while(intpart>0) {
            intBuf[int_idx++]='0'+(intpart%10);
            intpart/=10;
        }
    }
    for (int j=int_idx-1;j>=0;j--)
        numStr[i++]=intBuf[j]; //输出整数部分
    if (precision>0) {
        numStr[i++]='.';
        int x=1;
        for (int j=0;j<precision;j++) {
            x*=10;
            frac=frac*x+0.5; //四舍五入
            int fracint=(int)frac; //转化为整数
            int fracidx=0;
            char fracBuf[33];
            for (int j=0;j<precision;j++) { //整数转化为字符串
                fracBuf[precision-1-j]='0'+(fracint%10);
                fracint/=10;
            }
            for (int j=0;j<precision;j++)
            {
                numStr[i++]=fracBuf[j]; //输出小数部分
            }
        }
        numStr[i]='\0';
    }
}
```

接着是%e 的实现, 默认为 6 位有效数字的科学计数法:

```
case 'e':{
    int precision=6;
    double temp = va_arg(ap, double);
    if (temp<0) {
        counter+=printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
        temp=-temp;
    }
    double_to_e(number, temp, precision);
    for (int j=0;number[j];++j)
    {
        counter+=printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
    }
    break;
}
```

同样, 需要在 stdlib.cpp 中实现 double_to_e 函数, 实现将任意浮点数转化为默认保留六位有效数字的科学计数法:

```
void double_to_e(char *str, double num, int precision) {
    int i = 0;
    if (num < 0.0) {
        str[i++] = '-';
    }
```

```

    num = -num;
}
if (num == 0.0) {
    str[i++] = '0';
    str[i++] = '.';
    for (int j = 0; j < precision; j++) {
        str[i++] = '0';
    }
    str[i++] = 'e';
    str[i++] = '+';
    str[i++] = '0';
    str[i++] = '0';
    str[i] = '\0';
    return;
}
//归一化,统计指数
int exp = 0;
while (num >= 10.0) { num /= 10.0; exp++; }
while (num < 1.0) { num *= 10.0; exp--; }

//四舍五入
int mult = 1;
for (int j = 0; j < precision; j++) {
    mult *= 10;
}
int total = (int)(num * mult + 0.5);

// 5) 处理四舍五入导致的进位溢出
if (total >= mult * 10) {
    total /= 10;
    exp++;
}

//拆整数位和小数位
int intPart = total / mult; // 1~9
int fracPart = total % mult; // 0 ... mult-1

//输出小数点前的部分
str[i++] = '0' + intPart;
str[i++] = '.';

//输出小数部分
{
    char buf[10];
    for (int j = precision - 1; j >= 0; j--) {
        buf[j] = '0' + (fracPart % 10);
        fracPart /= 10;
    }
    for (int j = 0; j < precision; j++) {
        str[i++] = buf[j];
    }
}

//输出指数标志和符号
str[i++] = 'e';
if (exp >= 0) {
    str[i++] = '+';
} else {
    str[i++] = '-';
    exp = -exp;
}

//输出指数,至少两位(不足补零),可支持更多位
{

```

```

char buf[10];
int idx = 0;
do {
    buf[idx++] = '0' + (exp % 10);
    exp /= 10;
} while (exp > 0);
// 至少两位
while (idx < 2) {
    buf[idx++] = '0';
}
// 反向写入
for (int j = idx - 1; j >= 0; j--) {
    str[i++] = buf[j];
}
}
str[i] = '\0';
}
}

```

2.1.2 实验结果展示

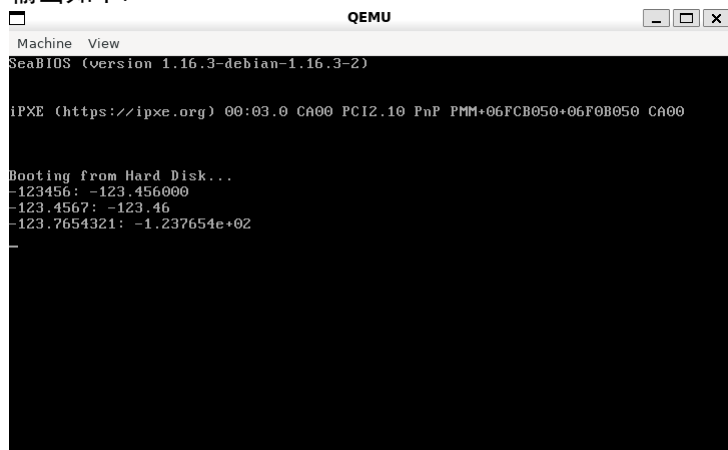
在 setup_kernel 中, 输入以下语句进行测试:

```

printf("-123456: %f\n", -123.456);
printf("-123.4567: %.2f\n", -123.4567);
printf("-123.7654321: %e\n", -123.7654321);

```

输出如下:



如图, 成功实现了%f,%.2f,%e 的格式化输出, 有效数字, 四舍五入等均处理正确。

2.2 实验任务二

对于自行设计的 PCB, 在原有 example 的基础上, 我增加了父子线程的机制, 并实现了父子线程之间关系的管理。

2.2.1 实验步骤

首先定义 PCB, 如果它是一个父线程, 那么它会含有一个包含它所有子线程的链表; 如果它是一个子线程, 那么它会被挂到某个父线程的子线程链表下, 又会存储它的父线程的指针。

```

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
    ListItem tagInChildrenList;  // 子线程队列标识
    PCB *parent;                 // 父线程
    List children;               // 子线程列表
};

```

在创建线程时, 传入的参数必须包含它的父线程的 PCB:

```

int ProgramManager::executeThread(ThreadFunction function, void *parameter, const char *name, int priority, PCB *
parent);

```

接着, 在 executeThread 函数中, 需要初始化子线程列表, 且将新线程的 PCB 的 parent 指针指向传入的父线程的 PCB, 并将其挂到父线程的子线程链表中。

```

thread->children.initialize(); // 初始化子线程列表
if (parent) {
    parent->children.push_back(&(thread->tagInChildrenList));
}

```

在 setup.cpp 中, 先实现一个可以遍历当前线程的所有子线程的函数, 在此函数中尤其需要判断它的子线程是否已经死亡, 如果死亡则不输出。

```

void print_children_pid(PCB *parent) {
    printf("%d 's children pid list:\n", parent->pid);
    ListItem *current = parent->children.head.next;
    while (current){
        PCB *child = ListItem2PCB(current, tagInChildrenList);
        if (child->status != DEAD) {
            printf("%d\n", child->pid);
        }
        current = current->next;
    }
}

```

接着我创建了四个线程, 各线程的逻辑如下:

- 线程 1: Init 线程,pid=0, 不会结束
- 线程 2: 线程 1 的子线程,pid=1, 执行完 printf 语句就结束
- 线程 3: 线程 1 的子线程,pid=2, 在此线程执行过程中创建子线程 4, 此线程不会结束
- 线程 4: 线程 3 的子线程,pid=1(因为线程 2 结束后 pid1 空闲, 分配给此线程), 此线程不会结束

```

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

void four_thread(void *arg) {
    printf("pid %d name \"%s\": The fourth thread\n", programManager.running->pid, programManager.running->name);
    print_children_pid(Init_PCB);
    asm_halt();
}

void third_thread(void *arg) {
    printf("pid %d name \"%s\": The third thread\n", programManager.running->pid, programManager.running->name);
}

```

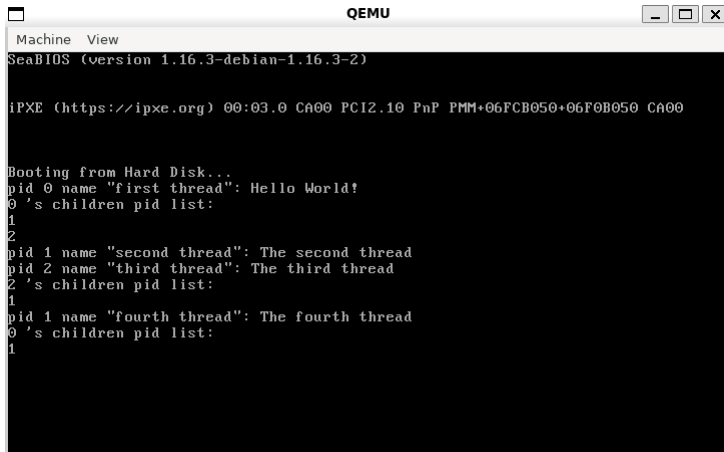
```

    programManager.executeThread(four_thread, nullptr, "fourth thread", 1, programManager.running);
    print_children_pid(programManager.running);
    asm_halt();
}
void second_thread(void *arg) {
    printf("pid %d name \"%s\": The second thread\n", programManager.running->pid, programManager.running->name);
    //asm_halt();
}

```

2.2.2 实验结果展示

在 setup_kernel 中, 创建线程 1。上述程序会分别在开始阶段和结束阶段查看线程 1 的子线程列表 (两者的区别是, 最后阶段线程 2 已经结束, 因此不会出现在子线程列表中), 以及查看线程 3 的子线程列表, 结果如图所示:



```

Machine View
SeaBIOS (version 1.16.3-debian-1.16.3-2)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+06FCB050+06F0B050 CA00

Booting from Hard Disk...
pid 0 name "first thread": Hello World!
0's children pid list:
1
2
pid 1 name "second thread": The second thread
pid 2 name "third thread": The third thread
2's children pid list:
1
pid 1 name "fourth thread": The fourth thread
0's children pid list:
1

```

如图, 各个阶段的各线程的子线程列表均符合预期。

2.3 实验任务三

运用 gdb 进行 debug, 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。

2.3.1 实验步骤

本实验中线程的设定如下:

```

void third_thread(void *arg) {
    printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid, programManager.running->name);
    while(1) {

    }
}
void second_thread(void *arg) {
    printf("pid %d name \"%s\": Hello World!\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\": Hello World!\n", programManager.running->pid, programManager.running->name);
    if (!programManager.running->pid)
    {
        programManager.executeThread(second_thread, nullptr, "second thread", 1);
        programManager.executeThread(third_thread, nullptr, "third thread", 1);
    }
    asm_halt();
}

```

重点关注由时间片轮转进行的线程调度, 因此线程的设置较为简单。

2.3.2 调试过程

首先创建线程 1:

```
Breakpoint 5, ProgramManager:
(gdb) p thread->pid
$1 = 0
```

接着是第一次调度: 由”空线程”调度 init 线程 (线程 1) 运行

```
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; 0x22d004 <PCB_SET+4068>
31
```

如图显示了栈指针的切换过程, 线程 1 的栈指针指向 0x21d20

```
(gdb) info register eax
eax             0x00000000
(gdb) info register esp
esp             0x77bc0000
(gdb) s
(gdb) s
(gdb) info register eax
eax             0x21d20000
(gdb) s
asm_switch_thread () at ../src/utils/asm_utils.asm:35
(gdb) info register esp
esp             0x22d00400 <PCB_SET+4068>
```

接着在线程 1 的执行过程中, 创建了线程 2:

```
Breakpoint 5, ProgramManager:
(gdb) p thread->pid
$2 = 1
(gdb)
```

在线程 1 的执行过程中, 创建了线程 3

```
Breakpoint 5, ProgramManager:
(gdb) p thread->pid
$3 = 2
(gdb)
```

线程 1 的执行过程中, 发生了时钟中断, 查询可知线程 1 的时间片初始为 10, 也就是说 10 次时钟中断后会发生线程调度

```
(gdb) p cur->ticks
$1 = 10
(gdb)
```

第二次调度: 时间片轮转调度, 发生调度时, 进入了 schedule() 函数, 在此 debug 进行调度的两个线程

```
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 }
```

cur->id=0,next->id=1, 调度: 线程 1→线程 2

```
(gdb) p cur->pid
$3 = 0
(gdb) p next->pid
$4 = 1
```

线程 1→线程 2 的调度过程: 栈指针切换, 先将线程 1 的栈指针入栈保存断点, 接着将 esp 赋值为线程 2 的栈指针。如图, 线程 1 的栈指针 0x21d20 入栈, 与先前一致。

```

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 ; 4F 58 5F 52 68
                       0D 58 53 4D 98
31

```

```

(gdb) info register eax
eax                0x21d20                138528

```

进入线程 2 后, 查看线程 2 的 pc, 已经发生变化

```

Breakpoint 6, second_thread (arg=0x0) at ../src/kernel/setup.cpp:21
(gdb) i r eip
eip                0x2077c                0x2077c <second_thread(void*)+6>

```

第三次调度: 线程 2 执行完毕, 线程 3 开始执行

```

Breakpoint 5, ProgramManager::schedule (
(gdb) p cur->pid
$5 = 1
(gdb) p next->pid
$6 = 2
(gdb)

```

查询线程 3 中的 pc, 发生了变化

```

Breakpoint 7, third_thread (arg=0x0) at ../src/kernel/setup.cpp:15
(gdb) i r eip
eip                0x20751                0x20751 <third_thread(void*)+6>
(gdb)

```

第四次调度: 线程 3 执行结束, 调度回线程 1(init 线程, 不会结束)

```

Breakpoint 5, ProgramManager::schedule (t
(gdb) p cur->pid
$7 = 2
(gdb) p next->pid
$8 = 0

```

2.3.3 一个新创建的线程是如何被调度然后开始执行的

新创建的线程先初始化线程栈, 然后将线程的 tagInGeneralList 加入 readyPrograms 就绪队列, tagInAllList 加入 all-Programs 全队列, 等待调度。

对于第一个创建的线程, 无法通过时钟中断触发调度, 需要手动调度: 调用 asm_switch_thread(0, firstThread), 将 init 线程切换到 firstThread 线程。

2.3.4 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

线程调度的关键机制: asm_switch_thread 函数, 它进行了两个线程之间的栈切换:

保存断点: 将旧线程的寄存器内容, 栈指针压栈存储。

恢复现场: 进行栈切换, 将 esp 指针切换到新线程的断点的位置。并 pop 出此线程的寄存器, 恢复现场。

2.4 实验任务四

本实验要求选择一种调度算法并实现, 我选择实现基于抢占式的 SRTN 调度算法, 即最短剩余时间优先。

2.4.1 实验步骤

首先在线程调度器中, 增加定义线程最短剩余时间以及其对应的线程, 并实现 update_shortest_remain_time() 函数, 更新最短剩余时间的线程。

```

class ProgramManager
{
public:
    List allPrograms; // 所有状态的线程/线程的队列
    List readyPrograms; // 处于 ready(就绪态)的线程/线程的队列

```



```

PCB *running;          // 当前执行的线程
int shortest_remain_time;
PCB *shortest_program;

public:
    ProgramManager();
    void initialize();
    void update_shortest_remain_time();

    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);
    void schedule();
};

```

update_shortest_remain_time() 函数的实现: 不需要对链表进行排序, 只需要从就绪队列中一个个遍历, 取出剩余时间最短者即可。

```

void ProgramManager::update_shortest_remain_time()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    shortest_remain_time = 0x7fffffff; // 初始化为最大整数
    shortest_program = nullptr;

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

    // 遍历就绪队列
    ListItem *item = readyPrograms.front(); // 使用 front() 获取第一个节点
    while (item) { // 通过 next 指针遍历直到 nullptr
        PCB *program = ListItem2PCB(item, tagInGeneralList);
        if (program->ticks < shortest_remain_time) {
            shortest_remain_time = program->ticks;
            shortest_program = program;
        }
        item = item->next;
    }

    interruptManager.setInterruptStatus(status);
}

```

抢占的实现在调度函数 schedule() 中, 调用 schedule() 时先检查是否需要抢占, 若需要抢占则立即进行调度。同时 schedule() 还对线程剩余时间进行检查, 若时间已用完, 则结束线程释放 PCB。

```

void ProgramManager::schedule()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

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

    // 更新最短剩余时间
    update_shortest_remain_time();
    if (running->status == ProgramStatus::RUNNING)

```

```

{
    // 检查是否需要抢占
    if (shortest_program && running->ticks > shortest_remain_time) {
        // 当前运行线程的剩余时间比就绪队列中最短的还要长,需要抢占
        running->status = ProgramStatus::READY;
        readyPrograms.push_back(&(running->tagInGeneralList));
    }
    else if (running->ticksPassedBy >= running->priority) {
        // 线程已经完成了预定的执行时间
        running->status = ProgramStatus::DEAD;
        printf("system tick:%d dead: %d\n", system_ticks, running->pid);
        releasePCB(running);
    }
    else {
        // 不需要抢占且未完成执行,继续运行当前线程
        interruptManager.setInterruptStatus(status);
        return;
    }
}
else if (running->status == ProgramStatus::DEAD)
{
    releasePCB(running);
}

// 选择最短剩余时间的线程
ListItem *item;
if (shortest_program) {
    item = &(shortest_program->tagInGeneralList);
    readyPrograms.erase(item);
    readyPrograms.push_front(item);
}

item = readyPrograms.front();
PCB *next = ListItem2PCB(item, tagInGeneralList);
PCB *cur = running;
printf("System_ticks:%d change: %d to %d\n", system_ticks, cur->pid, next->pid);
next->status = ProgramStatus::RUNNING;
running = next;
readyPrograms.pop_front();

asm_switch_thread(cur, next);

interruptManager.setInterruptStatus(status);
}

```

一次时钟中断为线程的一个执行时间单位, 每次时钟中断时, 先对线程时间片减 1, 接着更新最短剩余时间信息并判断是否需要抢占并进行调度 (通过 `shedule` 函数完成), 以下为时钟中断函数的处理

```

if (cur->ticks) {
    --cur->ticks;
    printf("pid:%d curticks: %d\n", cur->pid, cur->ticks);
    ++cur->ticksPassedBy;
    programManager.schedule();
}

```

2.4.2 实验结果展示

以下为我编写的测试样例, 其中以时钟中断为一个时间单位:

- 线程 1: 到达时间为 0, 执行时间为 8
- 线程 2: 到达时间为 3, 执行时间为 4
- 线程 3: 到达时间为 4, 执行时间为 2

线程 2、3 的创建在时钟中断处理函数中实现:

```
extern "C" void c_time_interrupt_handler()
{
    // 增加系统时钟计数
    system_ticks++;
    //printf("-----system_ticks: %d-----\n", system_ticks);
    // 在特定时间创建新线程
    if (system_ticks == 3 ) {
        printf("System_ticks: %d [Time interrupt] Create second thread\n", system_ticks);
        programManager.executeThread(second_thread, nullptr, "second thread", 4);
    }
    else if (system_ticks == 4 ) {
        printf("System_ticks: %d [Time interrupt] Create third thread\n", system_ticks);
        programManager.executeThread(third_thread, nullptr, "third thread", 2);
    }
    PCB *cur = programManager.running;
    if (cur->ticks) {
        --cur->ticks;
        printf("pid:%d curticks: %d\n", cur->pid, cur->ticks);
        ++cur->ticksPassedBy;
        programManager.schedule();
    }
}
```

线程 1、2、3 的实现如下:

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

    }
}

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

void first_thread(void *arg)
{
    printf("Processing pid %d name \"%s\": Thread 1\n", programManager.running->pid, programManager.running->name);
    asm_halt();
}
```

并在线程创建, 线程调度, 线程结束, 以及每个时间单位运行的线程的剩余时间片, 通过 printf 展示结果, 结果如下:

```
Machine View
-----system_ticks: 1-----
pid:0 curticks: 7
-----system_ticks: 2-----
pid:0 curticks: 6
-----system_ticks: 3-----
System_ticks: 3 [Time interrupt] Create second thread
pid:0 curticks: 5
System_ticks:3 change: 0 to 1
Processing pid 1 name "second thread": Thread 2
-----system_ticks: 4-----
System_ticks: 4 [Time interrupt] Create third thread
pid:1 curticks: 3
System_ticks:4 change: 1 to 2
Processing pid 2 name "third thread": Thread 3
-----system_ticks: 5-----
pid:2 curticks: 1
-----system_ticks: 6-----
pid:2 curticks: 0
system tick:6 dead: 2
System_ticks:6 change: 2 to 1
-----system_ticks: 7-----
pid:1 curticks: 2
-----system_ticks: 8-----
pid:1 curticks: 1
-----system_ticks: 9-----
pid:1 curticks: 0
system tick:9 dead: 1
System_ticks:9 change: 1 to 0
-----system_ticks: 10-----
pid:0 curticks: 4
-----system_ticks: 11-----
pid:0 curticks: 3
-----system_ticks: 12-----
pid:0 curticks: 2
-----system_ticks: 13-----
pid:0 curticks: 1
-----system_ticks: 14-----
pid:0 curticks: 0
```

如图可得, 按照 SRTN 算法, 线程的执行顺序为:

- 时间 0-3: 线程 1 执行
- 时间 3: 创建线程 2, 线程 2 剩余时间为 4, 抢占线程 1
- 时间 3-4: 线程 2 执行
- 时间 4: 创建线程 3, 线程 3 剩余时间为 2, 抢占线程 2
- 时间 4-6: 线程 3 执行
- 时间 6: 线程 3 执行完毕, 调度线程 2 执行
- 时间 6-9: 线程 2 执行
- 时间 9: 线程 2 执行完毕, 调度线程 1 执行
- 时间 9-14: 线程 1 执行

实验结果与预期结果一致。

3 实验总结与心得体会

- 擅于利用 printf 输出进行 debug: 在实现 SRTN 算法时, 通过 printf 输出线程的剩余时间片, 可以直观地看到线程的执行顺序, 以及抢占是否发生, 帮助检验代码逻辑是否正确。
- 在实现子线程时, 需要注意子线程是否已经结束, 尽量避免”僵尸线程”和”孤儿线程”的出现。
- 在实现 printf 函数中的%f、%d 时, 需要考虑多种情况, 如负数, 舍入溢出等情况, 需要利用多个样例进行全方面的测试。