



本科生实验报告

实验课程：_____操作系统原理实验_____

实验名称：_____实验 5 内核线程_____

专业名称：_____计算机科学与技术(超级计算方向) _____

学生姓名：_____林天皓_____

学生学号：_____18324034_____

实验地点：_____

实验成绩：_____

报告时间：_____2021. 4. 30_____

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。

1. 可变参数函数的实现

传统函数的实现都是固定参数的类型和数量的，为了实现可变参数的函数，我们需要了解 x86 函数调用参数的压栈方法，x86 内存中的对齐等规则。

函数调用压栈规则：从右向左参数依次压栈，栈的增长方向为向低地址方向，因此函数调用过程中参数从左到右依次对应了栈中低地址到高地址。

内存对齐：在 32 位 x86 系统中，对齐方式与其数据类型的大小基本相同。编译器在其自然长度边界上对齐变量。

Data Type	Alignment (bytes)
char	1
short	2
int	4
float	4

不过对于函数调用栈来说，无论是 `char`、`short` 还是 `int`，这些变量在栈中都是以 4 字节对齐的。在我们实现的 `printf` 中，为了正确读取参数需要进行对齐计算。计算对齐的宏如下

```
#define _INTSIZEOF(n) ((sizeof(n)+sizeof(int)- 1) & ~(sizeof(int) - 1))
```

然而我们不会遇到超过 4 个字节的参数，所以直接

```
#define _INTSIZEOF(n) 4
```

也可以获得正确的结果。

然后通过 `va_start(ap, v)` 将 `ap` 指向第一个参数

```
#define va_start(ap, v) (ap = (va_list)&v + _INTSIZEOF(v))
```

在对参数进行处理后，通过将 `ap` 指向下一个参数的同时返回当前 `ap` 对应的参数值。

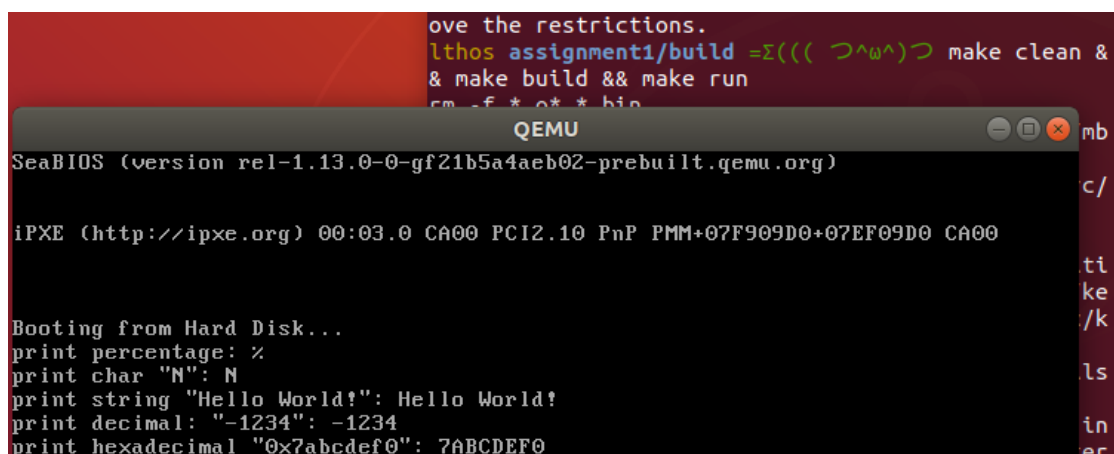
```
#define va_arg(ap, type) (*(type *)((ap += _INTSIZEOF(type)) - _INTSIZEOF(type)))
```

使用完成之后将 `ap=0`，防止因为 bug 读取到意外的参数。

变长参数的 `int printf(const char *const fmt, ...)` 函数较长，此处不在列出，大概过程为每读取到格式化字符，则读取一个参数，对参数做出对应处理，加入到输出的 buffer 中，然后通过 `stdio.print(buffer)` 输出 buffer 中的值。

调用方式同 c 语言中打 printf

下面运行测试



```
SeaBIOS (version rel-1.13.0-0-gf21b5a4aeb02-prebuilt.qemu.org)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F909D0+07EF09D0 CA00

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

能够输出十进制，十六进制，字符，字符串等参数。

2. 给 printf 加入 log 级别和颜色属性。

从日志中弄清楚并非易事。日志管理解决方案从多个来源收集并接受数据。这些源可以具有不同的日志事件结构，我们可以通过设置日志级别在阅读日志并试图赋予它们含义，通过不同的颜色和状态展示要输出的日志。

那么我们定义一个 `log.h`

```
#ifndef LOG_H
#define LOG_H

#define DEBUG 5
#define INFO 10
#define ERROR 15
```

```
int printf_debug(const char *const fmt, ...);
int printf_error(const char *const fmt, ...);
int printf_info(const char *const fmt, ...);

#endif
```

定义三种 log 级别，分别为 debug，error，info。

在 log.cpp 中实现函数：例如对于 printf_debug，在除了和普通的 printf 一样的代码之外，需要加入判断当前 LOG_LEVEL 的语句，函数判断部分如下：

```
int printf_debug(const char *const fmt, ...)
{
    if(LOG_LEVEL>DEBUG)return 0;
```

通过比较宏定义中的值，确定是否执行 print 语句。并且在对应的输出部分加上颜色。其他函数结构类似，不再赘述。

使用说明：通过编辑 Makefile 中的 CXX_FLAGS 设定 LOG_LEVEL,例如添加-D LOG_LEVEL=0 即可输出所有的 printf，添加-D LOG_LEVEL=15 仅仅输出 error 级别的语句。

运行测试：setup_kernel 函数如下

```
extern "C" void setup_kernel()
{
    interruptManager.initialize();
    stdio.initialize();
    interruptManager.enableTimeInterrupt();
    interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);
    printf_error("error message :%d %x\n", -1234, 0x7abcdef0);
    printf_debug("debug message :%d %x\n", -1234, 0x7abcdef0);
    printf_info ("info message :%d %x\n", -1234, 0x7abcdef0);
    asm_halt();
}
```

测试 1: LOG_LEVEL=0

```
QEMU
SeaBIOS (version rel-1.13.0-0-gf21b5a4aeb02-prebuilt.gel
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F
Booting from Hard Disk...
error message :-1234 7ABCDEF0
debug message :-1234 7ABCDEF0
info message :-1234 7ABCDEF0
```

三条 printf 语句全部输出了。

测试 2: LOG_LEVEL=15

```
QEMU
SeaBIOS (version rel-1.13.0-0-gf21b5a4aeb02-prebuilt.gel
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+0
Booting from Hard Disk...
error message :-1234 7ABCDEF0
```

只输出 printf_error 的语句。

综上所述，该部分完成了对可变参数 printf 的编写，并且在此基础上添加了 printf_info, printf_debug, printf_error 三种 log 级别，可通过编译选项调节开关。

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。

该部分通过一个进程管理器进行进程的调度和切换，

执行进程管理器的流程如下：

第一步：初始化进程管理器

```
programManager.initialize();
```

对应代码为

```
void ProgramManager::initialize()
{
    allPrograms.initialize();//初始化 all 链表
    readyPrograms.initialize();//初始化 ready 链表
    running = nullptr;//没有 running
    for (int i = 0; i < MAX_PROGRAM_AMOUNT; ++i)
    {
        PCB_SET_STATUS[i] = false;//标记没有进程占用 pcb
    }
}
```

这部分将 ready 和 all 链表清空，指定不存在正在运行的程序，清空 pcb 占用。

第二步：创建第一条进程

```
int pid = programManager.executeThread(first_thread, nullptr, "first thread", 1);
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();//申请 PCB 空间
    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];//设定进程名
    }
    thread->status = ProgramStatus::READY;//设置进程状态为 READY
    thread->priority = priority;//设置进程优先级
    thread->ticks = priority * 10;//设定优先级
    thread->ticksPassedBy = 0;//初始化经过的时间
    thread->pid = ((int)thread - (int)PCB_SET) / PCB_SIZE;
    // 线程栈
    thread->stack = (int *)((int)thread + PCB_SIZE);//重置程序栈为最大值
    thread->stack -= 7;//申请 7 个栈空间
    thread->stack[0] = 0;
    thread->stack[1] = 0;
```

```

thread->stack[2] = 0;
thread->stack[3] = 0;
thread->stack[4] = (int)function;//进程要执行的函数
thread->stack[5] = (int)program_exit;//函数退出后执行
thread->stack[6] = (int)parameter;//函数参数

allPrograms.push_back(&(thread->tagInAllList));//加入 all 链表
readyPrograms.push_back(&(thread->tagInGeneralList));//加入 ready 链表
// 恢复中断
interruptManager.setInterruptStatus(status);

return thread->pid;
}

```

具体流程：关中断，申请 PCB 空间，设置 PCB 状态，将进程加入 all 和 target 链表中。

这时候仅仅创建了线程，但是没有经过 schedule 并不会被调用，而且第一个执行的进程需要我们手动使用 asm_switch_thread 调用。

```

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

```

对于其他的线程切换，是通过时钟中断调用 `c_time_interrupt_handler` 切换的。

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

    if (cur->ticks)
    {
        --cur->ticks; //消耗时间片
        ++cur->ticksPassedBy;
    }
    else //时间片耗尽则执行切换程序
    {
        programManager.schedule();
    }
}
```

每一次时钟中断都会减少当前正在运行的线程的时间片，当时间片用尽后，则执行 `schedule` 切换进程。

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

    if (readyPrograms.size() == 0) //只有一个运行进程，不切换
    {
        interruptManager.setInterruptStatus(status);
        return;
    }

    if (running->status == ProgramStatus::RUNNING) //否则切换当前运行的进程
    {
        running->status = ProgramStatus::READY; //状态设置为 ready 状态
        running->ticks = running->priority * 10; //重新设置时间片
        readyPrograms.push_back(&(running->tagInGeneralList)); //加入 readylist 中
    }
    else if (running->status == ProgramStatus::DEAD)
    {
        releasePCB(running); //进程已经结束，释放 PCB 空间
    }

    ListItem *item = readyPrograms.front(); //选取最前的 ready 程序
    PCB *next = ListItem2PCB(item, tagInGeneralList); //获取 PCB 地址
```



```

PCB *cur = running;//获取当前地址
next->status = ProgramStatus::RUNNING;//设置为 running 状态
running = next;//设置为 running 状态
readyPrograms.pop_front();//从 ready 中删除

asm_switch_thread(cur, next);

interruptManager.setInterruptStatus(status);
}

```

Schedule 部分为找到将要切换的下一个进程，并且维护 PCB 状态，例如重新设置时间片，设置运行状态，然后通过 `asm_switch_thread` 切换运行栈。

下面运行测试，使用的进程代码如下

```

void third_thread(void *arg) {
    printf("pid %d name \"%s\": Hello third World!\n", programManager.running->pid, programManager.running->name);
    while(1) {
        printf("pid %d name \"%s\": Hello third World!\n", programManager.running->pid, programManager.running->name);
        for(int i=0;i<100000000;i++);
    }
}

void second_thread(void *arg) {
    printf("pid %d name \"%s\": Hello second World!\n", programManager.running->pid, programManager.running->name);
    while(1) {
        printf("pid %d name \"%s\": Hello second World!\n", programManager.running->pid, programManager.running->name);
        for(int i=0;i<100000000;i++);
    }
}

void first_thread(void *arg)
{
    // 第 1 个线程不可以返回
    printf("pid %d name \"%s\": Hello first 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();  
}
```

运行测试：



```
QEMU  
SeaBIOS (version rel-1.13.0-0-gf21b5a4aeb02-prebuilt.qemu.org)  
  
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F909D0+07EF09D0 CA00  
  
Booting from Hard Disk...  
pid 0 name "first thread": Hello first World!  
pid 1 name "second thread": Hello second World!  
pid 1 name "second thread": Hello second World!  
pid 2 name "third thread": Hello third World!  
pid 2 name "third thread": Hello third World!  
pid 1 name "second thread": Hello second World!  
pid 2 name "third thread": Hello third World!  
pid 1 name "second thread": Hello second World!  
pid 2 name "third thread": Hello third World!
```

当三个线程轮流切换时候，会出现线程 2 和线程 3 轮流输出语句的现象。

综上这部分完成了对 PCB 的定义，以及通过进程管理器的多线程的创建和切换执行。

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。

在 assignment2 中我们通过说明代码指定了程序运行的流程，下面通过实际的 debug 观察线程的执行。

1.firstThread 线程如何创建的过程

```
59     asm_halt();
60 }
61
62 ListItem *item = programManager.readyPrograms.front();
63 PCB *firstThread = ListItem2PCB(item, tagInGeneralList);
> 64 firstThread->status = RUNNING;
65 programManager.readyPrograms.pop_front();
66 programManager.running = firstThread;
67 asm_switch_thread(0, firstThread);
68
69     asm_halt();
70 }
71
72
73
74
75
```

```
remote Thread 1.1 In: setup_kernel
0x23704 <PCB_SET+4196>: 0x0 0x0 0x0 0x0
0x23714 <PCB_SET+4212>: 0x2052f <first_thread(void*)> 0x20385 <program_exit(> 0x0 0x0
0x23724 <PCB_SET+4228>: 0x0 0x0
(gdb) print item
$4 = (ListItem *) 0x2274c <PCB_SET+172>
(gdb) print firstThread
$5 = (PCB *) 0x22720 <PCB_SET+128>
(gdb) print firstThread->stack
$6 = (int *) 0x23704 <PCB_SET+4196>
(gdb) x/8a firstThread->stack
0x23704 <PCB_SET+4196>: 0x0 0x0 0x0 0x0
0x23714 <PCB_SET+4212>: 0x2052f <first_thread(void*)> 0x20385 <program_exit(> 0x0 0x0
(gdb) |
```

运行中，可见 firstThread 位于 PCB_SET+128，firstThread->stack 位于 PCB_SET+4196，其中已经写入了函数地址。

2. firstThread 线程开始运行的过程

由于 firstThread 是我们通过手动切换线程执行的，与通过定时器 schedule 切换的线程有所不同。

```
asm_switch_thread(0, firstThread);
```

```
Register group: general
eax      0x0      0
ecx      0x22720  141038
edx      0x0      0
ebx      0x39000  233472
esp      0x7bc0   0x7bc0
ebp      0x7bfc   0x7bfc
esi      0x0      0
edi      0x0      0
eip      0x21d24  0x21d24 <asm_switch_thread+8>
eflags   0x12     [ IOPL=0 AF ]
cs       0x20     32
ss       0x10     16
ds       0x8      8
es       0x8      8
fs       0x0      0
gs       0x18     24
fs_base  0x0      0

../src/utils/asm_utils.asm
17  ASM_UNHANDLED_INTERRUPT_INFO db 'Unhandled interrupt happened, halt ... '
18                                  db 0
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, 以便日后恢复
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

remote Thread 1.1 In: asm_switch_thread
(gdb) ni
(gdb) si
asm_switch_thread () at ../src/utils/asm_utils.asm:24
(gdb) layout regs
(gdb) si
(gdb) print esp+5*4
No symbol "esp" in current context.
(gdb) print $esp+5*4
$7 = (void *) 0x7bd4
(gdb) print [$esp+5*4]
A syntax error in expression, near `[$esp+5*4]'.
(gdb) si
(gdb)
```

我们将 firstThread 和 0 切换, 因此将 esp 保存在了 0 中, 不过之后我们不再回到 setup_kernel 函数中, 因此这里没有影响。

```
Register group: general
eax      0x22720  141088
ecx      0x22720  141088
edx      0x0      0
ebx      0x0      0
esp      0x23714  0x23714 <PCB_SET+4212>
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x21d31  0x21d31 <asm_switch_thread+21>
eflags   0x212    [ IOPL=0 IF AF ]
cs       0x20     32
ss       0x10     16
ds       0x8      8
es       0x8      8
fs       0x0      0

../src/utls/asm_utils.asm
33      mov esp, [eax] ; 此时栈已经是从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:
44      xor eax, eax
45      pushfd
46      pop eax
47      and eax, 0x200
48      ret

remote Thread 1.1 In: asm_switch_thread
(gdb) print firstThread->stack
$1 = (int *) 0x23704 <PCB_SET+4196>
(gdb) x/20a firstThread->stack
0x23704 <PCB_SET+4196>: 0x0 0x0 0x0 0x0
0x23714 <PCB_SET+4212>: 0x2052f <first_thread(void*)> 0x20385 <program_exit()>
0x23724 <PCB_SET+4228>: 0x0 0x0 0x0 0x0
0x23734 <PCB_SET+4244>: 0x0 0x0 0x0 0x0
0x23744 <PCB_SET+4260>: 0x0 0x0 0x0 0x0
(gdb) si
asm_switch_thread () at ../src/utls/asm_utils.asm:24
asm_switch_thread () at ../src/utls/asm_utils.asm:35
asm_switch_thread () at ../src/utls/asm_utils.asm:36
asm_switch_thread () at ../src/utls/asm_utils.asm:37
asm_switch_thread () at ../src/utls/asm_utils.asm:38
asm_switch_thread () at ../src/utls/asm_utils.asm:40
asm_switch_thread () at ../src/utls/asm_utils.asm:41
(gdb) |
```

当 `asm_switch_thread` 执行到 `ret` 语句时，可见栈已经为 `0x23714`，`ret` 之后正好可以切换到 `first_thread` 函数开始执行，因此，第一个线程的创建与运行成功结束了。

4. 进程如何被中断然后被换下处理器的？

例如第二和第三个线程的开始运行，是第一个线程被时钟中断所触发的 `schedule` 函数切换下来。

线程的创建过程与第一个线程相同，这里我们关注的是时钟中断引发的 schedule 进行的进程与进程之间的切换。通过设置 break schedule 断点进入进程切换函数，

```
76
77 void ProgramManager::schedule()
78 {
B+ 79     bool status = interruptManager.getInterruptStatus();
80     interruptManager.disableInterrupt();
81
82     if (readyPrograms.size() == 0)
83     {
84         interruptManager.setInterruptStatus(status);
85         return;
86     }
87
88     if (running->status == ProgramStatus::RUNNING)
89     {
90         running->status = ProgramStatus::READY;
91         running->ticks = running->priority * 10;
92         readyPrograms.push_back(&(running->tagInGeneralList));
93     }
94     else if (running->status == ProgramStatus::DEAD)
95     {
96         releasePCB(running);
97     }
98
99     ListItem *item = readyPrograms.front();
100    PCB *next = ListItem2PCB(item, tagInGeneralList);
101    PCB *cur = running;
> 102    next->status = ProgramStatus::RUNNING;
103    running = next;
104    readyPrograms.pop_front();
105
106    asm_switch_thread(cur, next);
107
```

```
remote Thread 1.1 In: ProgramManager::schedule
$7 = 0x2374c <PCB_SET+4268> "H'\003"
(gdb) print (ListItem *) (PCB_SET+4268)
$8 = (ListItem *) 0x2374c <PCB_SET+4268>
(gdb) print (ListItem) (PCB_SET+4268)
Invalid cast.
(gdb) print readyPrograms->next->next
There is no member or method named next.
(gdb) print readyPrograms->next
There is no member or method named next.
(gdb) print readyPrograms->head->next
$9 = (ListItem *) 0x2374c <PCB_SET+4268>
(gdb) n1
(gdb) print next
$10 = (PCB *) 0x23720 <PCB_SET+4224>
(gdb) print cur
$11 = (PCB *) 0x22720 <PCB_SET+128>
(gdb)
```

现在将会进行切换的进程的 PCB 地址分别为 PCB_SET+128 和 PCB_SET+4224，对应第一个线程和第二个线程。下面进入汇编切换线程查看过程。

```
Register group: general
eax      0x23720  145184
ecx      0x1      1
edx      0x0      0
ebx      0x0      0
esp      0x24704  0x24704 <PCB_SET+8292>
ebp      0x236a4  0x236a4 <PCB_SET+4100>
esi      0x0      0
edi      0x0      0
eip      0x21d2c  0x21d2c <asm_switch_thread+16>
eflags   0x6      [ IOPL=0 PF ]
cs       0x20     32
ss       0x10     16
ds       0x8      8
es       0x8      8
fs       0x0      0

-- ./src/utils/asm_utils.asm
26      push edi
27      push esi
28
29      mov eax, [esp + 5 * 4]
30      mov [eax], esp ; 保存当前栈指针到PCB中, 以便日后恢复
31
32      mov eax, [esp + 6 * 4]
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

remote Thread 1.1 In: asm_switch_thread
(gdb) si
(gdb) x/1i 0x22720
0x22720 <PCB_SET+128>:      add    al,0x37
(gdb) x/10a $esp
0x23668 <PCB_SET+4040>: 0x0      0x0      0x0      0x236a4 <PCB_SET+4100>
0x23678 <PCB_SET+4056>: 0x2036b <ProgramManager::schedule()+303>      0x22720 <
+4116>
0x23688 <PCB_SET+4072>: 0x209cb <InterruptManager::setInterruptStatus(bool)+29> 0
(gdb) print second_thread
$13 = {void (void *)} 0x204d0 <second_thread(void*)>
(gdb) si
asm_switch_thread () at ../src/utils/asm_utils.asm:35
(gdb) x/10a $esp
0x24704 <PCB_SET+8292>: 0x0      0x0      0x0      0x0
0x24714 <PCB_SET+8308>: 0x204d0 <second_thread(void*)> 0x20385 <program_exit()>
0x24724 <PCB_SET+8324>: 0x72696874      0x68742064
(gdb) |
```

根据调用规则将会在 ret 之后进入 second_thread 执行第二个线程。

```

eax      0x24720  149280
ecx      0x1      1
edx      0x0      0
ebx      0x0      0
esp      0x25714  0x25714 <PCB_SET+12404>
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x21d30  0x21d30 <asm_switch_thread+20>
eflags   0x16     [ IOPL=0 AF PF ]
cs       0x20     32
ss       0x10     16
ds       0x8      8
es       0x8      8
fs       0x0      0
--./src/utils/asm_utils.asm--
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:
44      xor eax, eax
45      pushfd
46      pop eax
47      and eax, 0x200
48      ret

remote Thread 1.1 In: asm_switch_thread
0x2467d <PCB_SET+8157>:      add     BYTE PTR [eax],al
0x2467f <PCB_SET+8159>:      add     BYTE PTR [edi+eax*2+0x2],cl
(gdb) x/10a $esp
0x2466c <PCB_SET+8140>: 0x2036b <ProgramManager::schedule()+303>      0x23720 <P
0x2467c <PCB_SET+8156>: 0x17      0x2474c <PCB_SET+8364>      0x23720 <PCB_SET+4224> 0x
0x2468c <PCB_SET+8172>: 0x0      0x0
(gdb) si
asm_switch_thread () at ../src/utils/asm_utils.asm:35
asm_switch_thread () at ../src/utils/asm_utils.asm:36
asm_switch_thread () at ../src/utils/asm_utils.asm:37
asm_switch_thread () at ../src/utils/asm_utils.asm:38
asm_switch_thread () at ../src/utils/asm_utils.asm:40
(gdb) x/10a $esp
0x25714 <PCB_SET+12404>:      0x20471 <third_thread(void*)>      0x20385 <program_e
0x25724 <PCB_SET+12420>:      0x0      0x0      0x0      0x0
0x25734 <PCB_SET+12436>:      0x0      0x0
(gdb) |

```



```
esi      0x0      0
edi      0x0      0
eip      0x21d31  0x21d31 <asm_switch_thread+21>
eflags   0x216    [ IOPL=0 IF AF PF ]
cs       0x20     32
ss       0x10     16
ds       0x8      8
es       0x8      8
fs       0x0      0

../src/utls/asm_utils.asm
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:
44      xor eax, eax
45      pushfd
46      pop eax
47      and eax, 0x200
48      ret

remote Thread 1.1 In: asm_switch_thread L41 PC: 0x21d31
0x2566c <PCB_SET+12236>: 0x2036b <ProgramManager::schedule()+303> 0x24720 <PCB_SET+8320> 0x22720 <PCB_SET+128> 0x20
0x2567c <PCB_SET+12252>: 0x16 0x2274c <PCB_SET+172> 0x24720 <PCB_SET+8320> 0x22720 <PCB_SET+128>
0x2568c <PCB_SET+12268>: 0x0 0x0
(gdb) si
asm_switch_thread () at ../src/utls/asm_utils.asm:35
asm_switch_thread () at ../src/utls/asm_utils.asm:36
asm_switch_thread () at ../src/utls/asm_utils.asm:37
asm_switch_thread () at ../src/utls/asm_utils.asm:38
asm_switch_thread () at ../src/utls/asm_utils.asm:40
asm_switch_thread () at ../src/utls/asm_utils.asm:41
(gdb) x/10a $esp
0x23678 <PCB_SET+4056>: 0x2036b <ProgramManager::schedule()+303> 0x22720 <PCB_SET+128> 0x23720 <PCB_SET+4224> 0x236b4 <PCB_SET+4116>
0x23688 <PCB_SET+4072>: 0x209cb <InterruptManager::setInterruptStatus(bool)+29> 0x2374c <PCB_SET+4268> 0x22720 <PCB_SET+128> 0x23720 <PCB_SET+4224>
0x23698 <PCB_SET+4088>: 0x236b4 <PCB_SET+4116> 0x21a20 <List::push_back(ListItem*)+14>
(gdb) |
```

第二次切换回进程 1 的时候有些许不一样，这里没有返回 firstThread 而是返回了 schedule 函数和第一个线程第一次切换回第二个线程的栈状态完全一样。

5. 进程被以及换上处理机后又是如何从被中断点开始执行的？

进程在 asm_switch_thread 中被切换走，因此，函数切换回来的过程会回到这一句的结束，即执行下一句 interruptManager.setInterruptStatus(status); 然后 ret 回到 c_time_interrupt_handler 中，然后 ret 回到 asm_time_interrupt_handler 中，最后 ret 之后才回到了原来 thread 中被切换的位置。

```
> 18     for(int i=0;i<100000000;i++);
19     }
20     }
21     void second_thread(void *arg) {
22         printf("pid %d name \"%s\": Hello second World!\n", programManager.running->pid, programManager.running->name);
23         while(1) {
24             printf("pid %d name \"%s\": Hello second World!\n", programManager.running->pid, programManager.running->name);
25             for(int i=0;i<100000000;i++);
26         }
27     }
28     void first_thread(void *arg)
29     {
30         // 第1个线程不可以返回
31         printf("pid %d name \"%s\": Hello first World!\n", programManager.running->pid, programManager.running->name);
32         if (!programManager.running->pid)
33         {
34             programManager.executeThread(second_thread, nullptr, "second thread", 1);
35         }
36     }
37 }
38
remote Thread 1.1 In: third_thread L18
asm_switch_thread () at ../src/utils/asm_utils.asm:38
asm_switch_thread () at ../src/utils/asm_utils.asm:40
asm_switch_thread () at ../src/utils/asm_utils.asm:41
0x0002036b in ProgramManager::schedule (this=0x32740) at ../src/kernel/program.cpp:106
(gdb) x/10a $esp
0x25670 <PCB_SET+12240>: 0x24720 <PCB_SET+8320> 0x22720 <PCB_SET+128> 0x20 0x16
0x25680 <PCB_SET+12256>: 0x2274c <PCB_SET+172> 0x24720 <PCB_SET+8320> 0x22720 <PCB_SET+128> 0x0
0x25690 <PCB_SET+12272>: 0x0 0x0
(gdb) si
InterruptManager::setInterruptStatus (this=0x32734, status=false) at ../src/kernel/interrupt.cpp:120
(gdb) ni
0x00020380 in ProgramManager::schedule (this=0x32740) at ../src/kernel/program.cpp:108
0x00020979 in c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:99
asm_time_interrupt_handler () at ../src/utils/asm_utils.asm:69
asm_time_interrupt_handler () at ../src/utils/asm_utils.asm:70
0x000204cb in third_thread (arg=0x0) at ../src/kernel/setup.cpp:18
(gdb)
```

因此执行一次线程切换需要数百条指令的执行，对于 CPU 时间的消耗是较大的。

因此需要尽可能少的进行线程的切换。

这部分中完成了对一号线程的创建与执行，以及线程之间的切换过程，线程如何被切换回来继续运行的过程。

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。

我们可以改变调度算法，来完成不同操作系统中应对不同情况的任务。例如抢占式调度会使得先运行优先级高的任务，再运行优先级低的任务。

我们改写 setup_kernel 函数，分别设置三个进程的优先级为 1，3，20

```
void third_thread(void *arg) {
    printf("pid %d name \"%s\": Hello third World!\n", programManager.running->pid, programManager.running->name);
    int k=5;
```

```

        while(k--) {
            printf("pid %d name \"%s\": Hello third World!\n", programManager.running->pid, programManager.running->name);
            for(int i=0;i<100000000;i++);
        }
    }
void second_thread(void *arg) {
    printf("pid %d name \"%s\": Hello second World!\n", programManager.running->pid, programManager.running->name);
    int k=5;
    while(k--) {
        printf("pid %d name \"%s\": Hello second World!\n", programManager.running->pid, programManager.running->name);
        for(int i=0;i<100000000;i++);
    }
}
void first_thread(void *arg)
{
    // 第1个线程不可以返回
    printf("pid %d name \"%s\": Hello first World!\n", programManager.running->pid, programManager.running->name);
    if (!programManager.running->pid)
    {
        programManager.executeThread(second_thread, nullptr, "second thread", 3);
        programManager.executeThread(third_thread, nullptr, "third thread", 20);
    }
    asm_halt();
}

```

然后改写进程调度器，使得优先级更高的程序能一直执行。

```

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

    if (readyPrograms.size() == 0) // 只有一个运行进程，不切换
    {
        interruptManager.setInterruptStatus(status);
        return;
    }
    if (running->status == ProgramStatus::DEAD)
    {

```

```

        releasePCB(running);//进程已经结束，释放 PCB 空间
        running->priority=-1;
    }
    int pro_size=readyPrograms.size();
    int max_priority=running->priority;
    PCB *max_priority_pcb=nullptr;
    ListItem *cur_item=readyPrograms.front();
    ListItem *max_item=nullptr;
    for(int i=0;i<pro_size;i++){
        PCB *cur_pcb = ListItem2PCB(cur_item, tagInGenerallist);//获取 PCB 地址
        if(cur_pcb->priority>max_priority){//寻找最高优先级的进程
            max_priority=cur_pcb->priority;
            max_priority_pcb=cur_pcb;
            max_item=cur_item;
        }
        cur_item=cur_item->next;
    }
    if(max_priority_pcb !=nullptr){
        printf_debug("switch to higher priority %d pid:%d\n",max_priority,max_
priority_pcb->pid);
        running->status = ProgramStatus::READY;//状态设置为 ready 状态
        running->ticks = running->priority * 10;//重新设置时间片
        readyPrograms.push_back(&(running->tagInGenerallist));//加入 readylist 中
        PCB *cur = running;//获取当前地址
        max_priority_pcb->status = ProgramStatus::RUNNING;//设置为 running 状态
        running = max_priority_pcb;//设置为 running 状态
        readyPrograms.erase(max_item);//从 ready 中删除
        asm_switch_thread(cur, max_priority_pcb);
    }

    interruptManager.setInterruptStatus(status);
}

```

通过找 ready 进程中有没有比当前进程更高的进程来切换，如果没有则不执行切

换，同时结束后需要将它的优先级设为-1，防止其他任务不能调度。

下面进行运行测试：

```
QEMU
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F909D0+07EF09D0 CA00

Booting from Hard Disk...
pid 0 name "first thread": Hello first World!
switch to higher priority 20 pid:2
pid 2 name "third thread": Hello third World!
pid 2 name "third thread": Hello third World!
pid 2 name "third thread": Hello third World!
pid 2 name "third thread": Hello third World!
pid 2 name "third thread": Hello third World!
exit thread 2
switch to higher priority 3 pid:1
pid 1 name "second thread": Hello second World!
pid 1 name "second thread": Hello second World!
pid 1 name "second thread": Hello second World!
pid 1 name "second thread": Hello second World!
pid 1 name "second thread": Hello second World!
exit thread 1
switch to higher priority 1 pid:0
```

看到切换到更加高优先级任务的过程，只有当该优先级的过程结束之后才能运行低优先级的任务。

下面我们改变优先级，再测试一次

```
QEMU
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F909D0+07EF09D0 CA00

Booting from Hard Disk...
pid 0 name "first thread": Hello first World!
switch to higher priority 20 pid:1
pid 1 name "second thread": Hello second World!
pid 1 name "second thread": Hello second World!
pid 1 name "second thread": Hello second World!
pid 1 name "second thread": Hello second World!
pid 1 name "second thread": Hello second World!
pid 1 name "second thread": Hello second World!
exit thread 1
switch to higher priority 3 pid:2
pid 2 name "third thread": Hello third World!
pid 2 name "third thread": Hello third World!
pid 2 name "third thread": Hello third World!
pid 2 name "third thread": Hello third World!
pid 2 name "third thread": Hello third World!
exit thread 2
switch to higher priority 1 pid:0
```

再测将线程 2 的优先级改为 20 之后，则线程 2 必须最先执行完毕。

综上，该部分完成了基于优先级的线程调度方法。

实验感想

本次实验是对线程的一次实验，在这次实验中，我们首先学习了 C 语言中变长参数函数的使用，并且加入 log 级别方便调试代码，在 assignment2-4 中中学会调度多个线程的创建，运行，与切换的原理与实现，同时在时间片轮转调度的基础上，改写实现了优先级优先的调度方法。对于操作系统中线程的了解更加深入了。

在本次实验中，主要已研究示例代码为主，在对代码的调试过程中。操作系统的运行也与其他程序设计有很大的区别，例如有进程 0 这样不会返回，而是一直在运行的函数。所谓进程切换就是在当前的运行栈切换到了另一个函数运行栈。

在即将到来的五一节假期中，会尝试将已有的 c++ 代码移植到 rust 上，为进一步的学习打下基础。

参考资料

<http://www.songho.ca/misc/alignment/dataalign.html>