



# 本科生实验报告

实验课程：\_\_\_\_\_操作系统\_\_\_\_\_

实验名称：\_\_\_\_\_内核线程\_\_\_\_\_

专业名称：\_\_\_\_\_计算机科学与技术\_\_\_\_\_

学生姓名：\_\_\_\_\_凌国明\_\_\_\_\_

学生学号：\_\_\_\_\_21307077\_\_\_\_\_

实验地点：\_\_\_\_\_教室\_\_\_\_\_

实验成绩：\_\_\_\_\_

报告时间：\_\_\_\_\_2023. 05. 04\_\_\_\_\_

## 1. 实验要求

- 学习 C 语言的可变参数机制的实现方法，实现一个简单的 printf 函数
  - 学习线程控制块的数据结构——PCB，实现 RR 和其他的线程调度算法
- 1) 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。
  - 2) 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。
  - 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，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。
  - 4) 将线程调度算法修改为下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。
    - 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

## 2. 实验过程

### 1) 实现 printf

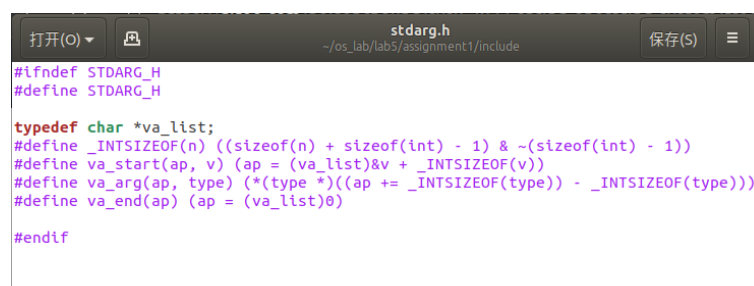
- 第一步，实现 print 一个字符的函数，在当前光标处打印一个字符，并将光标加一，注意光标在最后一个位置时要进行滚屏

```
void STDIO::print(uint8 c, uint8 color)
{
    uint cursor = getCursor();
    screen[2 * cursor] = c;
    screen[2 * cursor + 1] = color;
    cursor++;
    if (cursor == 25 * 80)
    {
        rollUp();
        cursor = 24 * 80;
    }
    moveCursor(cursor);
}
```

- 第二步，实现 print 一个字符串的函数，对字符串每个字符进行遍历，当遇到'\n'时换行，其他情况直接输出该字符

```
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;
}
```

- 第三步，在 include 中编写 stdarg 文件



```
打开(O) 保存(S) 菜单
stdarg.h
~/.os_lab/lab5/assignment1/include

#ifndef STDARG_H
#define STDARG_H

typedef char *va_list;
#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)

#endif
```

- 第四步，编写一个 `add_to_buffer` 函数，将要打印的字符放到缓冲区，等到缓冲区满或者遇到特殊条件时才输出

```
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;
}
```

- 第五步，实现 `printf`

```
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, char), idx, BUF_LEN);
                break;

            case 's':
                buffer[idx] = '\\0';
                idx = 0;
                counter += stdio.print(buffer);
                counter += stdio.print(va_arg(ap, const char *));
                break;
        }
    }
}
```

如果遇到普通字符，则直接输出；如果遇到%，则看%的下一个字符是什么。

如果 % 的下一个字符是 '\\0'，则退出；如果是 'c'，则以字符的格式返回一个可变参数列表中的参数，并将其加入缓冲区；如果是 's'，则输出并清空缓冲区，并以 char\* 的格式返回一个可变参数列表中的参数，并直接打印；如果是 'd'，则以 int 的格式返回一个可变参数列表中的参数，转换为字符串后加入缓冲区；如果是 'x'，则以 int 的格式返回一个可变参数列表中的参数，进行进制转换，然后转换为字符串，加入缓冲区；

- 最后, 在 setup\_kernel.cpp 中加入教程的测试语句, 并修改 makefile, make && make run 运行 qemu, 得到结果如下

```
linggm@linggm-virtual-machine:~/os_lab/lab5/assignment1/build$ make && make run
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP FMM+07F8DD0+07ECDD0 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
```

```
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10
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) 实现 PCB 和线程创建调度

- 第一步, 声明 PCB 结构体, 成员有栈指针, 线程名, 线程状态, 优先级, pid, 已执行时间等

```
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
};
```

- 第二步，声明 list 类，是带头结点的双向链表

```
class List
{
public:
    ListItem head;

public:
    // 初始化List
    List();
    // 显式初始化List
    void initialize();
}
```

- 第三步，编写“程序管理器”类，成员有所有线程的队列、就绪线程的队列、当前执行的线程，实现创建线程、分配 PCB、回收 PCB、进行线程调度的功能（“程序管理器”类的具体实现见“关键代码说明”部分）

```
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();
}
```

- 最后，加入教程的测试语句，make && make run 运行 qemu，得到结果如下

```
Booting from Hard Disk...
pid 0 name "first thread": Hello World!
 9  8  7  6  5  4  3  2  1  0

thread_switching
pid 1 name "second thread": Hello World!
 9  8  7  6  5  4  3  2  1  0

thread_switching
pid 2 name "third thread": Hello World!
 9  8  7  6  5  4  3  2  1  0

thread_switching
pid 3 name "fourth thread": Hello World!
 9  8  7  6  5  4  3  2  1  0
```

### 3) 对任务二的内容进行 debug，查看线程调度时的信息

- 第一步，在中断处理函数和调度函数入口设置断点

```
Breakpoint 1 at 0x2090c: file ../src/kernel/interrupt.cpp, line 89.
(gdb) b schedule
Breakpoint 2 at 0x2023c: file ../src/kernel/program.cpp, line 77.
```

- 时钟中断发生时，查看 ticks 的信息，发现每次时钟中断 ticks-=1，正确

```
$2 = 8
(gdb) c
Continuing.

Breakpoint 1, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:89
(gdb) p cur->ticks
$3 = 7
```

- ticks 为 0 时，发生调度，查看首次调度发生时，pid 为 0，ticks 为 0

```
76 void ProgramManager::schedule()
B+> 77 {
    78     bool status = interruptManager.getInterruptStatus();
    79     interruptManager.disableInterrupt();
    80
    81     if (readyPrograms.size() == 0)
    82     {
    83         interruptManager.setInterruptStatus(status);
    84         return;
    }

remote Thread 1 In: ProgramManager::schedule L77
    at ../src/kernel/program.cpp:77
(gdb) p running.pid
$1 = 0
(gdb) p running.ticks
$2 = 0
(gdb) p running.ticksPassedBy
$3 = 10
```

- 切换 running 的线程为下一个线程后，running 的 pid 变为 1，完成调度

```
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 printf("\nthread switching\n");
    105 asm switch_thread(cur, next);

remote Thread 1 In: ProgramManager::schedule L103
(gdb) ni
(gdb) p running->pid
$8 = 1
(gdb) p running->ticks
$9 = 10
(gdb) p running->ticksPassedBy
$10 = 0
```

- asm\_thread\_switch 前查看寄存器信息

```

../src/utils/asm_utils.asm
20     ASM_IDTR dw 0
21         dd 0
22
23     ; void asm_switch_thread(PCB *cur, PCB *next);
24     asm_switch_thread:
B+> 25         push ebp
26         push ebx
27         push edi
28         push esi
29
30         mov eax, [esp + 5 * 4]
31         mov [eax], esp ; 保存当前栈指针到PCB中，以便日后
32
remote Thread 1 In: asm_switch_thread                                L25    PC: 0x214fc
eax      0x21dc0  138688
ecx      0x21dc0  138688
edx      0x0      0
ebx      0x39000  233472
esp      0x7bd0   0x7bd0
ebp      0x7bfc   0x7bfc
esi      0x0      0
edi      0x0      0
eip      0x214fc  0x214fc <asm_switch_thread>
eflags   0x12     [ AF ]
cs       0x20     32

```

- asm\_thread\_switch 后查看寄存器信息

```

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

remote Thread 1 In: asm_switch_thread                                L42    PC: 0x21511
eax      0x21dc0  138688
ecx      0x21dc0  138688
edx      0x0      0
ebx      0x0      0
esp      0x22db4  0x22db4 <PCB_SET+4084>
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x21511  0x21511 <asm_switch_thread+21>
eflags   0x212    [ AF IF ]
cs       0x20     32

```



#### 4) 修改任务二中的代码，实现先来先服务的调度算法

- 第一步，修改 first\_thread, second\_thread 和 third\_thread，函数的最后不挂起，这样线程才能执行完毕，从而调度下一个线程


```
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);
        programManager.executeThread(fourth_thread, nullptr, "fourth thread", 1);
    }
}
```

- 第二步，修改中断处理函数，时钟中断发生时，不做任何事情
- 第三步，修改 program\_exit 函数，在某一个线程终止时，如果就绪队列非空（有线程处于 ready 状态），则进行一次调度；否则挂起

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

    if (!programManager.readyPrograms.empty())
    {
        programManager.schedule();
    }
    else
    {
        interruptManager.disableInterrupt();
        printf("halt\n");
        asm_halt();
    }
}
```

- 结果



```
Booting from Hard Disk...
pid 0 name "first thread": Hello World!

thread_switching
pid 1 name "second thread": Hello World!

thread_switching
pid 2 name "third thread": Hello World!

thread_switching
pid 3 name "fourth thread": Hello World!
```

### 3 关键代码

任务一的代码在“实验过程”中作了展示和说明，不赘述

任务四的代码与任务二的代码大同小异，不同点在“实验过程”中

下面主要介绍任务二的代码：

- 1) 程序管理类初始化，先初始化两个队列，然后将 running 置空指针，将 PCB 的状态置为“未分配”

```
void ProgramManager::initialize()
{
    allPrograms.initialize();
    readyPrograms.initialize();
    running = nullptr;

    for (int i = 0; i < MAX_PROGRAM_AMOUNT; ++i)
    {
        PCB_SET_STATUS[i] = false;
    }
}
```

- 2) 创建线程，先关中断，然后分配 PCB

```
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;
```

- 再初始化分配的页，全置 0，然后初始化名字、状态、优先级等

```
// 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;
```

- 然后对线程栈进行处理

```
// 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;
```

- 最后在两个队列中 push 新的线程，恢复中断，返回

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

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

return thread->pid;
```

- 3) **线程 exit 函数**：将当前 running 的线程的状态置 dead，如果不是主线程，则调度；如果是主线程，则 halt 挂起

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

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

- 4) **线程调度函数**：如果就绪队列里没有线程（没有就绪线程）则原封不动返回；如果有就绪的线程，而且当前线程处于 running 状态，则置当前线程为 ready，入就绪队列，将就绪队列头部的线程置 running 态；如果有就绪的线程，而且当前线程处于 dead 状态，则释放当前线程的 PCB，将就绪队列头部的线程置 running 态，运行。

```

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);
}

```

- 5) `interrupt_handler` 函数: 每次时钟中断, 将 `ticks--`, `ticks` 为 0 时调度, 这实现了 RR 调度算法

```

// 中断处理函数
extern "C" void c_time_interrupt_handler()
{
    PCB *cur = programManager.running;

    if (cur->ticks)
    {
        --cur->ticks;
        ++cur->ticksPassedBy;
    }
    else
    {
        programManager.schedule();
    }
}

```

## 4 实验结果

### 1) Assignment1

```
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) Assignment2

```
Booting from Hard Disk...
pid 0 name "first thread": Hello World!
 9 8 7 6 5 4 3 2 1 0

thread_switching
pid 1 name "second thread": Hello World!
 9 8 7 6 5 4 3 2 1 0

thread_switching
pid 2 name "third thread": Hello World!
 9 8 7 6 5 4 3 2 1 0

thread_switching
pid 3 name "fourth thread": Hello World!
 9 8 7 6 5 4 3 2 1 0
```

### 3) Assignment3 (debug 信息详见过程部分)

### 4) Assignment4

```
Booting from Hard Disk...
pid 0 name "first thread": Hello World!

thread_switching
pid 1 name "second thread": Hello World!

thread_switching
pid 2 name "third thread": Hello World!

thread_switching
pid 3 name "fourth thread": Hello World!
```