

操作系统原理实验报告

**实验名称:** 实验五 内核线程

**授课教师：** 张青

**学生姓名:** 曾慧蕾

**学生学号:** 21307358

1. **实验要求**

**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。

**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。

1. **实验过程**

**Assignment 1 Printf的实现：**

本次实验，我们主要学习了可变参数机制的实现。C语言首先提供我们定义可变参数列表需要的符号“…”，有了这个符号后，我们可以任意地改变函数的形式参数，这些函数的参数都按顺序会被放到栈上面。此时，<stdarg.h>提供了3个访问栈中的参数的宏。而这些可变参数的具体意义是什么，需要我们在使用这些宏的时候人为规定。这便是可变参数的本质。

在理解了可变参数的本质，并复现了教程中的printf函数效果后，我们来考虑如何进一步改进教程中的printf函数。

在教程中的printf，可以解析的参数如下所示：



在这些参数的基础上，考虑比较容易改进的一个地方便是：printf的格式化输出，在%后所读取的数字即为在输出字符前需要空出的位宽。如%15d则要在输出整数前先输出15个空格。

这个改进的实现比较容易，只需要在读取到%后连续读取其后的数字num，并在输出内容前先输出num个空格即可，算法改进代码如下所示：

stdio.cpp中：

//improve here

int printf(const char \*const fmt, ...)

{

    const int BUF\_LEN = 32;

    char buffer[BUF\_LEN + 1]; //last one put '\0'

    char number[33];

    int idx, counter;

    va\_list ap;

    va\_start(ap, fmt);

    idx = 0;

    counter = 0;

    for (int i = 0; fmt[i]; ++i)

    {

        int space=0;

        if (fmt[i] != '%')

        {

            counter += printf\_add\_to\_buffer(buffer, fmt[i], idx, BUF\_LEN);

        }

        else //检查%后面的参数

        {

            i++;

            while(fmt[i]<='9' && fmt[i]>='0'){

                space=space\*10+fmt[i]-'0';

                i++;

            }

            if (fmt[i] == '\0')

            {

                break;

            }

            if (fmt[i] !='%'){

                for(int k=0;k<space;k++){

                    counter+=printf\_add\_to\_buffer(buffer, ' ', idx, BUF\_LEN);

                }

            }

            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;

            case 'd':

                {int temp = va\_arg(ap,int);

                if(temp<0 && fmt[i]=='d'){

                  counter += printf\_add\_to\_buffer(buffer, '-', idx, BUF\_LEN);

                  temp = -temp;

                }

                itos(number,temp,10);

                for (int j = 0; number[j]; ++j)

                {

                    counter += printf\_add\_to\_buffer(buffer, number[j], idx, BUF\_LEN);

                }

                break;}

            case 'x':

                {int temp = va\_arg(ap, int);

                itos(number, temp,16);

                for (int j = 0; number[j]; ++j)

                {

                    counter += printf\_add\_to\_buffer(buffer, number[j], idx, BUF\_LEN);

                }

                break;}

//在未加大括号时的报错：crosses initialization of ‘int tem’int temp=va\_arg(ap,int);

//因为在C和C++中，一个变量的生命期在整个switch中都会有效，要使原先的temp到达其生命周期的话，则要打一个大括号来定义

            }

        }

    }

    buffer[idx] = '\0';

    counter += stdio.print(buffer);

    return counter;

}

setup.cpp中：

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 char \"N\": %3c\n"

           "print string \"I'm zenghlei\": %s\n"

           "print string \"I'm zenghlei\": %6s\n"

           "print decimal: \"-21307358\": %d\n"

           "print decimal: \"-21307358\": %9d\n"

           "print hexadecimal \"0x7abcdef0\": %x\n"

           "print hexadecimal \"0x7abcdef0\": %6x\n",

           'N', 'N',"I'm zenghlei","I'm zenghlei", -21307358,-21307358, 0x7abcdef0,0x7abcdef0);

    //uint a = 1 / 0;

    asm\_halt();

}

输出结果：



可见程序正确按照预期输出，改进部分结束。

**Assignment 2 线程的实现：**

本部分要求自行设计PCB，在原来的基础上可以添加更多的属性。

我们创建的线程的状态有5个，分别是创建态、运行态、就绪态、阻塞态和终止态。而线程的组成部分线程各自的栈，状态，优先级，运行时间，线程负责运行的函数，函数的参数等，这些组成部分被集中保存在一个结构中——PCB(Process Control Block)。其组成部分有：

* status是线程的状态，如运行态、阻塞态和就绪态等。
* name是线程的名称。
* priority是线程的优先级，线程的优先级决定了抢占式调度的过程和线程的执行时间。
* pid是线程的标识符，每一个线程的pid都是唯一的。
* ticks是线程剩余的执行次数。在时间片调度算法中，每发生中断一次记为一个tick，当ticks=0时，线程会被换下处理器，然后将其他线程换上处理器执行。
* ticksPassedBy是线程总共执行的tick的次数。
* tagInGeneralList和tagInAllList是线程在线程队列中的标识，用于在线程队列中找到线程的PCB。

在确定好PCB后，下一步来看线程的创建。在创建线程之前，我们需要向内存申请一个PCB。这一部分又被称为PCB的分配，其主要内容包括PCB的分配和释放。

对于线程的创建，主要步骤是：

1. 关中断
2. 向PCB\_SET申请一个线程的PCB，并进行初始化，向栈中存入需要的整数。
3. 等待时间中断，在时间中断到来后该线程即可被调度到处理器。
4. 恢复中断

本次实验的改进部分有PCB内容的添加，如：线程的优先级以及线程的大小。添加这两部分主要是为了后续调度算法的改进。具体代码如下所示：

thread.h：

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 priority\_now;                // 线程优先级(now)

    int size;                        // size of the thread

    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

};

program.cpp：

int ProgramManager::executeThread(ThreadFunction function, void \*parameter, const char \*name, int priority, int size)  // 创建线程

{

    // 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->priority\_now=priority;

    thread->ticks = priority \* 10;

    thread->ticksPassedBy = 0;

    thread->pid = ((int)thread - (int)PCB\_SET) / PCB\_SIZE;

    thread->size=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));

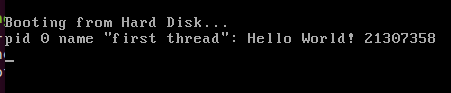
    // 恢复中断

    interruptManager.setInterruptStatus(status);

    return thread->pid;

}

在未改变setup.cpp时的实验结果：



在更改setup.cpp后（去掉第一个线程的注释）的实验结果：

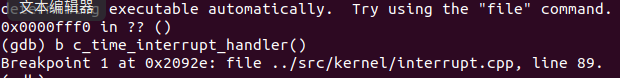


可见，无论是一个线程，还是三个线程，都能成功创建并执行。

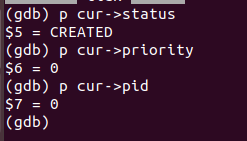
**Assignment 3 线程调度切换的秘密：**

本任务主要是在gdb的时候观察线程切换前后栈、寄存器、PC等变化并解释说明。

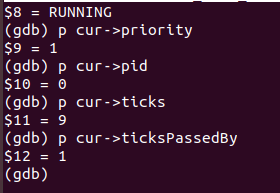
在gdb时，先给c\_time\_interrupt\_handler设置断点：



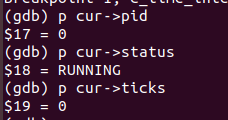
在第一次到达断点，线程处于创建态时，各参数为：



第二次到达断点，线程处于运行态时，各参数为：



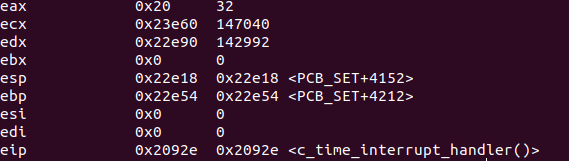
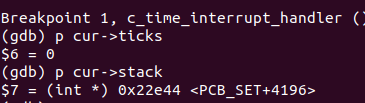
这样一直运行，每次运行，ticks就会减1，代表着剩余时间片的消耗。直到时间片耗尽前，线程都会一直以上面的方式运行。而到剩余时间片为0时：



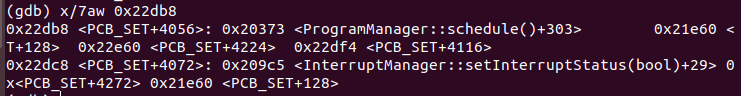
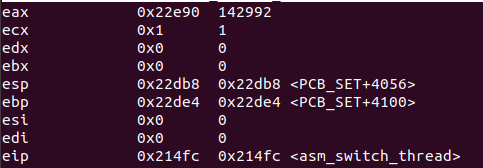
下面就要进行线程的调度。此时会执行asm\_switch\_thread。我们在这里设置断点：



第一次进行线程调度时，栈指针和寄存器情况为：



第二次进入到线程调度函数时的栈指针和寄存器情况：



可见eax保存了上一个线程的edx

asm调度函数的作用：实现线程栈的切换，保存下一个线程的栈指针。当下一次线程被换上时，asm\_switch\_thread会读取保存的内容（栈、寄存器、PC等），并通过这些内容恢复某特定线程，CPU的pc会跳转并继续执行。

**Assignment 4 调度算法的实现：**

在本任务的实现时，我选择的是最短作业（进程）优先。选用这一调度算法刚好可以利用任务二中增添的内容。具体做法为：

在创建线程时先声明好线程的大小，在进行线程调度时，优先调用最大的一个线程并执行。实验代码为：

Program.cpp：

//improve here

void ProgramManager::schedule()  //最短作业（进程）优先

{

    bool status = interruptManager.getInterruptStatus();

    interruptManager.disableInterrupt();

    if (readyPrograms.size() == 0)

    {

        interruptManager.setInterruptStatus(status);

        return;

    }

    if (running->status == ProgramStatus::RUNNING) //重新初始化其状态为就绪态

    {

        running->status = ProgramStatus::READY;

        running->ticks = running->priority \* 10;

        readyPrograms.push\_back(&(running->tagInGeneralList));

    }

    else if (running->status == ProgramStatus::DEAD) //回收线程的PCB

    {

        releasePCB(running);

    }

    int min=99,id=0;

    PCB \*next = nullptr;

    ListItem \*head = readyPrograms.front();

    int sum=readyPrograms.size();

    for(int i=0;i<sum;i++){

        PCB \*t = ListItem2PCB(head,tagInGeneralList);

        if(t->size<min){

            min=t->size;

            id=i;

            next=t;

        }

        head=head->next;

    }

    PCB \*cur = running;

    next->status = ProgramStatus::RUNNING;

    running = next;

    readyPrograms.erase(id);

    asm\_switch\_thread(cur, next);

    interruptManager.setInterruptStatus(status);

}

setup.cpp：

void first\_thread(void \*arg)

{

    // 第1个线程不可以返回

    printf("pid %d name \"%s\": Hello World! 21307358\n", programManager.running->pid, programManager.running->name);

    if (!programManager.running->pid)

    {

        programManager.executeThread(first\_thread, nullptr, "first thread", 1, 1);

        programManager.executeThread(third\_thread, nullptr, "second thread", 1, 5);

        programManager.executeThread(third\_thread, nullptr, "third thread", 1, 2);

    }

    asm\_halt();

}

Setup\_kernel:

    // 创建第一个线程

    int pid = programManager.executeThread(first\_thread, nullptr, "first thread", 1, 12);

    if (pid == -1)

    {

        printf("can not execute thread\n");

        asm\_halt();

    }

运行结果：



与预期相同，按线程大小1<3<2的顺序执行。

1. **总结**

通过本次实验我学习到了C/C++在Linux下的使用机制，了解并学习了线程在操作系统中的执行方式，以及PCB的作用，对线程调度算法也有了更深入的理解和思考，收获颇多。