



中山大學
SUN YAT-SEN UNIVERSITY

本科生实验报告

实验课程: 操作系统原理实验

任课教师: 刘宁

实验题目: 第四章 内核线程

专业名称: 信息与计算科学

学生姓名: 郑鸿鑫

学生学号: 22336313

实验地点: 实验中心 D503

实验时间: 2024/4/17

Section 1 实验概述

在本次实验中，我们将会学习到 C 语言的可变参数机制的实现方法。在此基础上，我们会揭开可变参数背后的原理，进而实现可变参数机制。实现了可变参数机制后，我们将实现一个较为简单的 `printf` 函数。此后，我们可以同时使用 `printf` 和 `gdb` 来帮助我们 debug。

本次实验另外一个重点是内核线程的实现，我们首先会定义线程控制块的数据结构——PCB。然后，我们会创建 PCB，在 PCB 中放入线程执行所需的参数。最后，我们会实现基于时钟中断的时间片轮转 (RR) 调度算法。在这一部分中，我们需要重点理解 `asm_switch_thread` 是如何实现线程切换的，体会操作系统实现并发执行的原理。

Section 2 预备知识与实验环境

- 预备知识：x86 汇编语言程序设计、IA-32 处理器体系结构，LBA 方式读写硬盘和 CHS 方式读写硬盘的相关知识。
- 实验环境：
 - 虚拟机版本/处理器型号：
11th Gen Intel® Core™ i5-11320H @ 3.20GHz × 2
 - 代码编辑环境：VS Code
 - 代码编译工具：g++
 - 重要三方库信息：Linux 内核版本号：linux-5.10.210
Ubuntu 版本号：Ubuntu 18.04.6LTS，Busybox 版本号：
Busybox_1_33_0

Section 3 实验任务

- 实验任务 1：printf 的实现
- 实验任务 2：线程的实现

- 实验任务 3: 时钟中断的实现
- 实验任务 4: 调度算法的实现

Section 4 实验步骤与实验结果

----- 实验任务 1 -----

- 任务要求: 学习可变参数机制, 在材料中的(src/3)的 printf 函数上进行改进。
- 思路分析: 主要学习指导书中的“printf 的实现”这一节, 修改代码以改进 printf 函数
- 实验步骤:
 1. 学习“一个可变参数的例子”, 运行 src/1 下的代码并理解定义可变参数的规则。
 2. 学习“可变参数的机制实现”, 运行 src/2 下的代码并理解可变参数的具体意义和本质。
 3. 学习“实现 printf”这一节, 将 src/3 下的代码进行修改以改进 printf 函数, 在原本代码基础上添加 %b 和 %f 这两种格式化输出。
 4. 在 stdio.cpp 文件中修改 printf 函数如下:

```
int printf(const char *const fmt, ...)
{
    const int BUF_LEN = 32;
    char buffer[BUF_LEN + 1];
    char number[33];
    int idx, counter;
    va_list ap;
    va_start(ap, fmt);
    idx = 0;
    counter = 0;
    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;
            }
            double temp1;
            int int_part;
            double fractional_part;
            int temp;
            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':
    case 'b':
    case 'x':
        temp = va_arg(ap, int);
        if (temp < 0 && fmt[i] == 'd')
        {
            counter += printf_add_to_buffer(buffer, '-', idx, BUF_LEN);
            temp = -temp;
        }
        if (fmt[i] == 'd')    itos(number, temp, 10);
        else if (fmt[i] == 'b')    itos(number, temp, 2);
        else if (fmt[i] == 'x')    itos(number, temp, 16);
        for (int j = 0; number[j]; ++j)
        {
            counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
        }
        break;
    case 'f':
        temp1 = va_arg(ap, double);
        int_part = (int)temp1; // Extract the integer part of the floating-point
number
        fractional_part = temp1 - int_part; // Extract the fractional part

        // Print the integer part
        itos(number, int_part, 10);
        for (int j = 0; number[j]; ++j) {
            counter += printf_add_to_buffer(buffer, number[j], idx, BUF_LEN);
        }

        // Print the decimal point
        counter += printf_add_to_buffer(buffer, '.', idx, BUF_LEN);

        // Print the fractional part (up to 6 decimal places)
        for (int i = 0; i < 6; i++) {
            fractional_part *= 10;
            int digit = (int)fractional_part;
            counter += printf_add_to_buffer(buffer, digit + '0', idx, BUF_LEN);
            fractional_part -= digit;
        }
        break;
    }
}

buffer[idx] = '\0';
counter += stdio.print(buffer);

return counter;
}

```

其核心在于修改了 switch 语句中的 case: 首先是在 %d 和 %x 的基础上增加了 %b, 并且增加对应的判断条件和相应的进制转化操作。

然后是增加 case %f: 先将整数部分加入缓冲区, 然后加入一个小数点“.”, 用于输出浮点数, 再将小数部分的前 6 位放入缓冲区。

5. 再对 setup.cpp 中的 setup_kernel() 函数进行修改, 代码如下所示:

```

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"
    "print float \"1.23456: %f\\n"
    "print binary\" 0b101010 \": %b\\n",
    'N', "Hello World!", -1234, 0x7abcdef0, 1.23456, 0xb101010);
//uint a = 1 / 0;
asm_halt();
}

```

主要的改动是在于调用 `printf` 函数进行测试时增加了两个测试用例，分别用于测试对于浮点数和二进制的格式化输出。

6. 进入 `build` 文件夹，输入 `make && make run` 进行测试运行得到结果。

● 实验结果展示：

编译运行后结果如下所示：

```

kernel > setup.cpp > setup_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"
        "print float \"1.23456: %f\\n"
        "print binary\" 0b101010 \": %b\\n",
        'N', "Hello World!", -1234, 0x7abcdef0, 1.23456, 0xb101010);
}

```

```

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
tdio.cpp ../src/utills/stdlib.cpp
nasm -o asm_utils.o -f elf32 ../src/utills/asm_utils.asm
ld -o kernel.o -melf_i386 -N entry.obj setup.o interrupt.o stdio.o stdlib.o asm_utils.o -e enter_kernel -Ttext 0x00020000
objcopy -O binary kernel.o kernel.bin
dd if=kernel.bin of=../run/hd.img bs=512 count=1 seek=0 conv=notrunc
记录了1+0 的写入
512 bytes copied, 0.00015181 s, 3.4 MB/s
dd if=bootloader.bin of=../run/hd.img bs=512 count=5 seek=1 conv=notrunc
记录了0+1 的写入
记录了0+1 的写入
281 bytes copied, 0.00014201 s, 2.0 MB/s
dd if=kernel.bin of=../run/hd.img bs=512 count=145 seek=6 conv=notrunc
记录了9+1 的写入
记录了9+1 的写入
4740 bytes (4.7 kB, 4.6 KiB) copied, 0.000132697 s, 35.7 MB/s
qemu-system-i386 -hda ../run/hd.img -serial null -parallel stdio -no-reboot
WARNING: Image format was not specified for '../run/hd.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.

```

```

ScalID5 (version 1.10.2-1ubuntu1)
QEMU
iPXE (http://ipxe.org) 00:03:0 C980 PC12.10 PaP PPM-07F8BBD0-07ECDD0 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
print float "1.23456": 1.234560
print binary "0b101010 ": 101010

```

----- 实验任务 2 -----

- 任务要求：自行设计 PCB，添加更多的属性如优先级等，然后根据 PCB 来实现进程，演示执行结果。
- 思路分析：主要学习指导书中“PCB 的分配”“线程的创建”“线程的调度”等，设计 PCB，为其添加优先级属性。
- 实验步骤：

1. 完善 PCB 结构体的属性如下：

```

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
};

```

2. 编写 setup.cpp 的代码如下所示，以演示优先级属性：

```

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

void second_thread(void *arg) {
    printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid,
programManager.running->name);
    printf("Second thread's priority: %d\n", programManager.running->priority);
}

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);
    printf("First thread's priority: %d\n", programManager.running->priority);
    if (!programManager.running->pid)
    {
        programManager.executeThread(second_thread, nullptr, "second thread", 2);
        programManager.executeThread(third_thread, nullptr, "third thread", 3);
    }
    asm_halt();
}

extern "C" void setup_kernel()
{
    // 中断管理器
    interruptManager.initialize();
    interruptManager.enableTimeInterrupt();
    interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);

    // 输出管理器
    stdio.initialize();

    // 进程/线程管理器
    programManager.initialize();

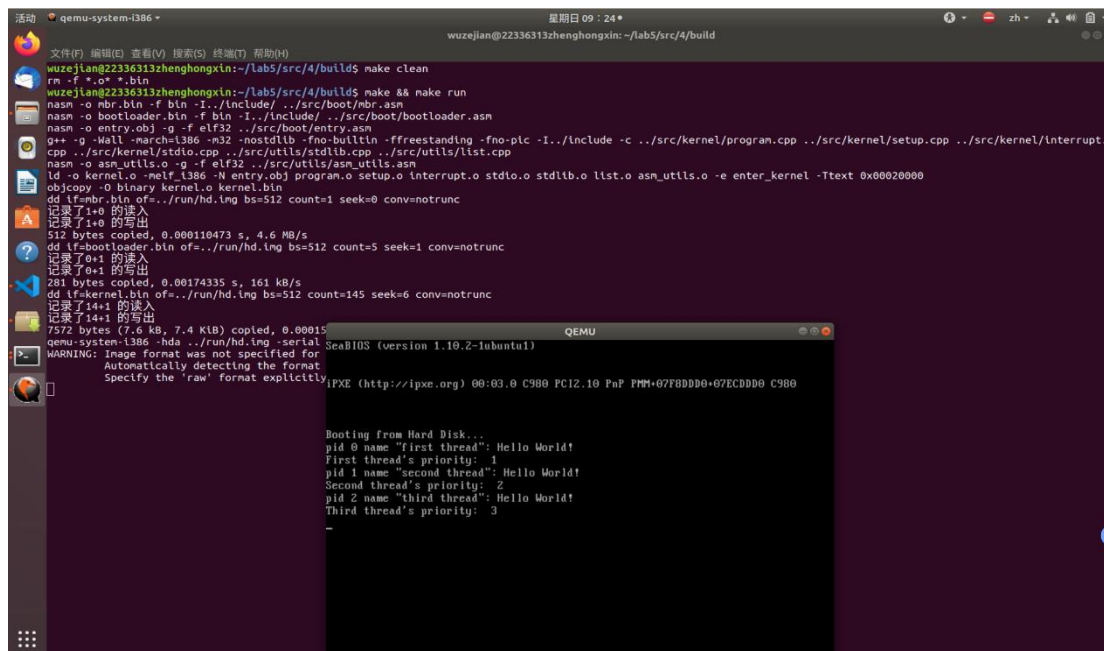
    // 创建第一个线程
    int pid = programManager.executeThread(first_thread, nullptr, "first thread", 1);
    if (pid == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }

    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);

    asm_halt();
}

```

- 实验结果展示：通过执行前述代码，可得下图结果：

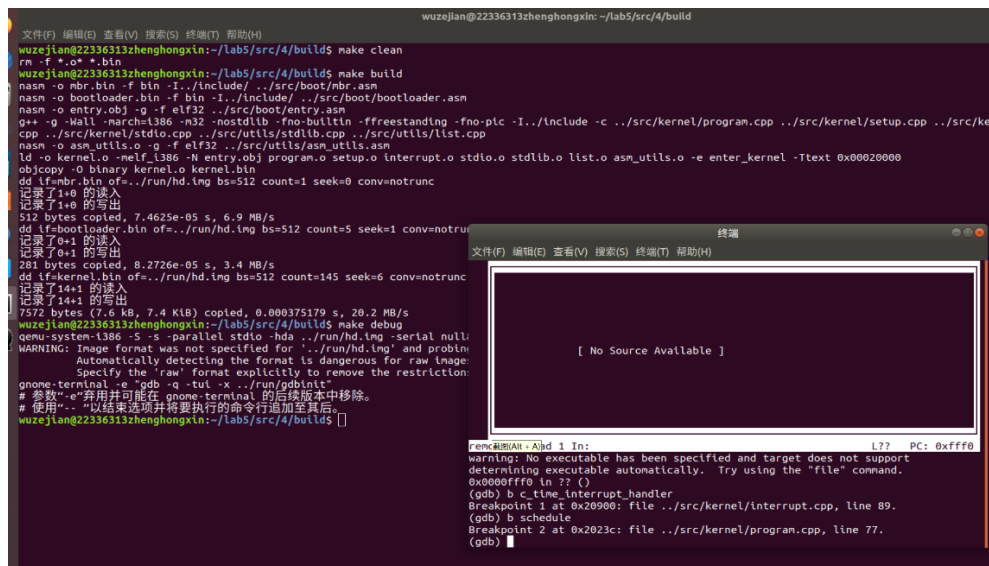


实验任务 3

- 任务要求：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，材料中的“线程的调度”的内容来跟踪并说明下面两个过程。
 1. 一个新创建的进程是如何被调度然后开始执行的。
 2. 一个正在执行的线程是如何被中断然后被换下处理器的，以及换上处理器后又是如何从被中断点开始执行的。
- 思路分析：学习指导书中“线程的调度”的内容，结合 gdb 调试来跟踪线程切换前后 PCB 的变化。
- 实验步骤：
 1. 配合 makefile 文件使用 `make debug` 命令启动 gdb。
 2. 使用 `b` 命令在两个函数的起始设置断点。
 3. 运行并每次时钟中断发生时查看 `ticks` 的信息。
 4. 等到 `ticks` 为 0 后会发生调度，使用 `p running.*` 命令来查看首次调度发生后的 `pid` 和 `ticks`。
 5. 继续运行并查看当前的 `pid`。
 6. 在进入 `asm_thread_switch` 代码段之前设置断点，查看寄存器信息，在函数 `ret` 后再次查看寄存器信息。

- 实验结果展示:

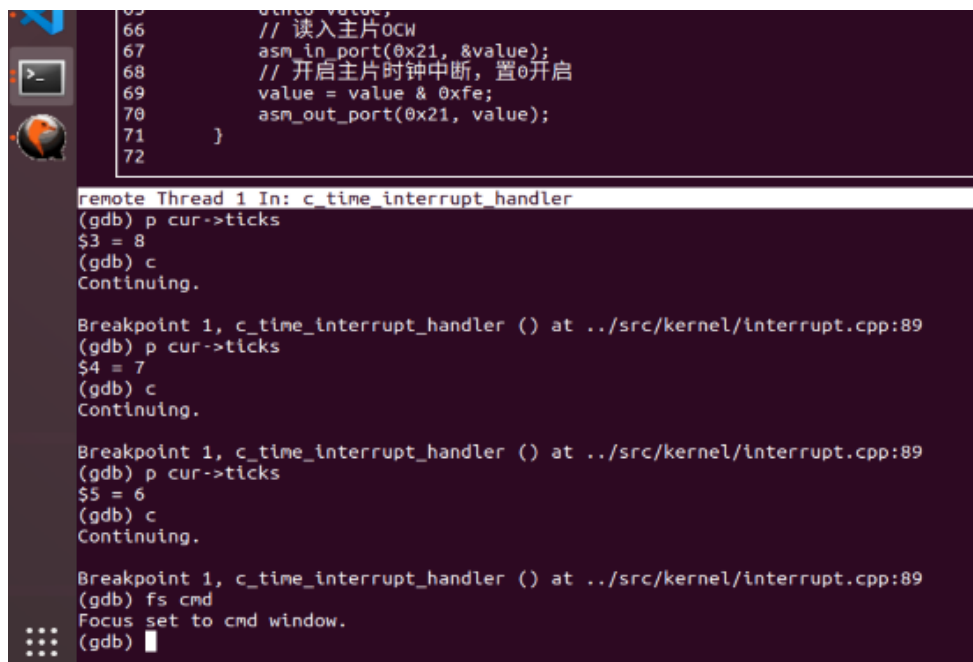
1. 启动 gdb 并设置断点:



```
wuzejian@22336313zhenghongxin: ~/lab5/src/4/build$ make clean
rm -f *.o *.bin
wuzejian@22336313zhenghongxin:~/lab5/src/4/build$ make build
nasm -o nabr.bin -f bin -I. ./include/ ../src/boot/nabr.asm
nasm -o bootloader.bin -f bin -I. ./include/ ../src/boot/bootloader.asm
nasm -o entry.obj -g -f elf32 ../src/boot/entry.asm
g++ -g -Wall -march=i386 -m32 -nostdlib -fno-builtin -ffreestanding -fno-pic -I. ./include -c ../src/kernel/program.cpp ../src/kernel/setup.cpp ../src/kernel/stdlib.o ../src/kernel/stdio.o ../src/utills/stdlib.o ../src/utills/list.o
nasm -o asm_utils.o -g -f elf32 ../src/utills/asm_utils.asm
ld -o kernel.o -melf_i386 -N entry.obj program.o setup.o interrupt.o stdio.o stdlib.o list.o asm_utils.o -e enter_kernel -Ttext 0x00020000
objcopy -O binary kernel.o kernel.bin
dd if=kernel.bin of=../run/hd.img bs=512 count=1 seek=0 conv=notrunc
记录了1+0 的写入
512 bytes copied, 7.4625e-05 s, 6.9 MB/s
dd if=bootloader.bin of=../run/hd.img bs=512 count=5 seek=1 conv=notrunc
记录了0+1 的写入
281 bytes copied, 8.2726e-05 s, 3.4 MB/s
dd if=kernel.o of=../run/hd.img bs=512 count=145 seek=6 conv=notrunc
记录了14+1 的写入
7572 bytes (7.6 kB, 7.4 KiB) copied, 0.000375179 s, 20.2 MB/s
wuzejian@22336313zhenghongxin:~/lab5/src/4/build$ make debug
qemu-system-i386 -S -s -parallel stdio -hda ../run/hd.img -serial null
WARNING: Image format was not specified for \'.run/hd.img\' and problem:
Automatically detecting the format is dangerous for raw image:
Specify the \'raw\' format explicitly to remove the restriction.
gnome-terminal -e "gdb -q -tui -x ./run/gdbinit"
# 参数"-e"弃用并可能在 gnome-terminal 的后续版本中移除。
# 使用"--" 以结束选项并将要执行的命令追加至其后。
wuzejian@22336313zhenghongxin:~/lab5/src/4/build$
```

2. 查看 ticks 的信息:

可以看到每次 ticks 都在递减, 符合预期。



```
66 // 读入主片OCW
67 asm_in_port(0x21, &value);
68 // 开启主片时钟中断, 置0开启
69 value = value & 0xfe;
70 asm_out_port(0x21, value);
71 }
72

remote Thread 1 In: c_time_interrupt_handler
(gdb) p cur->ticks
$3 = 8
(gdb) c
Continuing.

Breakpoint 1, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:89
(gdb) p cur->ticks
$4 = 7
(gdb) c
Continuing.

Breakpoint 1, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:89
(gdb) p cur->ticks
$5 = 6
(gdb) c
Continuing.

Breakpoint 1, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:89
(gdb) fs cmd
Focus set to cmd window.
(gdb)
```

3. ticks 为 0 后继续运行, 可以看到发生的调度, 跳转到了 schedule 函数的断点:


```

(gdb) p cur->ticks
$8 = 2
(gdb) c
Continuing.

Breakpoint 1, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:89
(gdb) p cur->ticks
$9 = 1
(gdb) c
Continuing.

Breakpoint 1, c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:89
(gdb) p cur->ticks
$10 = 0
(gdb) p running.pid
No symbol "running" in current context.
(gdb) c
Continuing.

Breakpoint 2, ProgramManager::schedule (this=0x31dc0 <programManager>) at ../src/kernel/program.cpp:77

```

4. 使用 `p running.*` 命令来查看值, 可以看到 `pid` 为 0, `ticks` 为 0, `ticksPasserBy` 为 10:

```

Continuing.

Breakpoint 2, ProgramManager::schedule (this=0x31dc0 <programManager>) at ../src/kernel/program.cpp:77
(gdb) p running.pid
$11 = 0
(gdb) p running.ticks
$12 = 0
(gdb) p running.ticksPassedBy
$13 = 10

```

5. 查看进程调度后的 `pid`, 可以看到为 1, 说明调度完成:

```

$13 = 10
(gdb) c
Continuing.

Breakpoint 2, ProgramManager::schedule (this=0x31dc0 <programManager>) at ../src/kernel/program.cpp:77
(gdb) p pod
No symbol "pod" in current context.
(gdb) p pid
No symbol "pid" in current context.
(gdb) p running.pid
$14 = 1

```

6. 进入 `asm_thread_switch` 代码段之前:

```

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
../src/utils/asm_utils.asm
16 extern c_time_interrupt_handler
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
35     pop esi
36     pop edi
37     pop ebx
38     pop ebp
39
40     stl

remote Thread 1 In: asm_switch_thread
Continuing.

Breakpoint 1, asm_switch_thread () at ../src/utils/asm_utils.asm:24
(gdb) info register
eax             0x21da0 138656
ecx             0x21da0 138656
edx             0x0     0
ebx             0x39000 233472
esp             0x7bd0  0x7bd0
ebp             0x7bfc  0x7bfc
esi             0x0     0
edi             0x0     0
eip             0x214cc 0x214cc <asm_switch_thread>
eflags          0x12    [ AF ]
cs              0x20    32
ss              0x10    16
ds              0x8     8
es              0x8     8
fs              0x0     0
gs              0x18    24

```

7. 在 ret 返回以后:

```
42 ; int asm_interrupt_status();
43 asm_interrupt_status:
> 44 xor eax, eax
45 pushfd
46 pop eax
47 and eax, 0x200
48 ret
49
50 ; void asm_disable_interrupt();
51 asm_disable_interrupt:
52 cli
53 ret
54 ; void asm_init_page_reg(int *directory);
55
56 asm_enable_interrupt:
57 sti

remote Thread 1 In: asm_interrupt_status
Continuing.

Breakpoint 2, asm_interrupt_status () at ../src/utils/asm_utils.asm:44
(gdb) info register
eax                0x0          0
ecx                0x22d12     142610
edx                0x1c         28
ebx                0x0          0
esp                0x22d28     0x22d28 <PCB_SET+3976>
ebp                0x22d34     0x22d34 <PCB_SET+3988>
esi                0x0          0
edi                0x0          0
eip                0x214e2     0x214e2 <asm_interrupt_status>
eflags             0x212       [ AF IF ]
cs                 0x20        32
ss                 0x10        16
ds                 0x8         8
es                 0x8         8
fs                 0x0         0
gs                 0x18        24
```

至此回答实验要求中提到的两个问题:

1. 答:

在操作系统中,新创建的进程(或线程)被调度并开始执行的过程通常涉及以下几个步骤:

- 创建线程: 首先,操作系统会创建一个新的线程,并为其分配一个线程控制块(PCB)。
- 初始化线程属性: 操作系统会初始化线程的属性,包括线程的状态、优先级、PID 等。
- 线程调度: 线程创建后,操作系统的调度器会根据调度策略决定何时运行该线程。调度策略可能包括时间片轮转、优先级调度等。
- 就绪队列: 新线程通常被放入就绪队列,等待被调度器选中并赋予 CPU 时间片。

上下文切换: 当调度器选择该线程运行时,会进行上下文切换,保存当前线程的状态,并加载新线程的上下文(寄存器、栈指针等)。

通过 GDB 观察到的现象包括:

ticks 递减至 0,表示当前线程的时间片即将耗尽。

调度器被触发，操作系统调用 `schedule` 函数来选择新的线程运行。

使用 `p running.*` 命令查看当前运行线程的 PCB 信息，可以看到 `pid` 为 0（为第一个进程），`ticks` 为 0（时间片耗尽），`ticksPassedBy` 为 10（表示线程已经运行的时间片数）。

2. 答：

线程被中断并换下处理器的过程通常涉及以下步骤：

- 时间片耗尽：当一个线程的时间片用完时，操作系统会通过时钟中断来中断该线程的执行。
- 保存状态：操作系统会保存被中断线程的状态，包括寄存器、程序计数器（PC）等。
- 上下文切换：操作系统会将控制权转给调度器，调度器选择另一个线程运行，并进行上下文切换，加载新线程的状态。
- 恢复执行：当被中断的线程再次被调度器选中时，操作系统会恢复之前保存的状态，线程从中断点继续执行。

通过 GDB 观察到的现象包括：

在 `asm_thread_switch` 代码段之前设置断点，查看寄存器信息，这些信息表示当前线程的状态。

在 `ret` 返回后再次查看寄存器信息，这些信息表示新线程的状态，此时可以看到栈指针、程序计数器等的变化。

这个过程涉及到操作系统的线程管理和上下文切换机制，是操作系统内核的重要组成部分

----- 实验任务 4 -----

- 任务要求：在材料中，我们已经学习了如何使用时间片轮转算法（RR）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
 - 最短作业优先
 - 响应比最高优先算法
 - 优先级调度算法
 - 多级队列反馈调度算法

此外，调度算法可以是抢占式的。

现在同学们要将上述线程调度算法修改为上面提到的算法，自行编写测试样例来测试算法的正确性。

- 思路分析：学习指导书中“线程的调度”这一节后，实现先到先服务线程调度算法。
- 实验步骤：

1. 首先将每次调用线程后的挂起语句注释，代码如下：

```
void third_thread(void *arg) {
    printf("pid %d name \"%s\": Hello World!\n", programManager.running->pid,
programManager.running->name);
    // asm_halt();
}
void second_thread(void *arg) {
    printf("pid %d name \"%s\": 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\": 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();
}
```

2. 修改 program.exit()函数，将其按照先来先服务的逻辑修改，主要改动：将原来的 RR 算法的注释掉，更改为当有进程终止且就绪队列非空时，则进行一次调度，否则挂起，详细见下代码：

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

3. 修改中断处理函数，在时钟中断发生时，即一个时间片可能用完时，RR 算

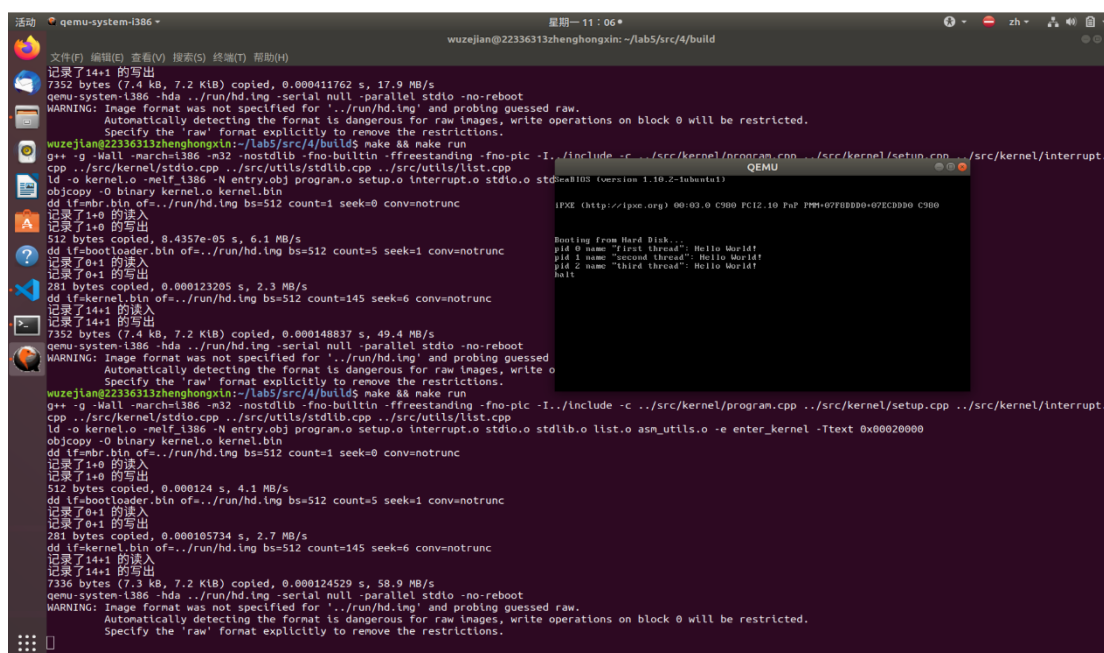
法可能需要调度，但是对于 FCFS 算法不需要，所以直接将原来的代码注释掉，即时钟中断发生时，不做任何操作：

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

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

4. 在 build 文件夹下测试代码，首先使用 make clean 命令将原来的文件清除，然后通过 make && make run 命令编译运行，查看测试结果。

- 实验结果展示：通过执行前述代码，可得下图结果：



Section 5 实验总结与心得体会

通过完成这个实验，本人获得了以下几个方面的总结与体会：

- ◆ 深入理解线程调度：实验加深了本人对操作系统线程调度机制的理解，包括线程状态的转换、时间片的概念以及调度算法的工作原理。
- ◆ 掌握 GDB 调试技巧：通过使用 GDB 来跟踪线程的执行和上下文切换，本人学会了如何设置断点、单步执行、查看和修改寄存器内容以及程序计数器（PC）。

- ◆ 直观观察上下文切换：实验让本人亲眼见证了上下文切换的过程，包括栈的变化、寄存器的保存和恢复，以及程序如何在中断后从相同的状态恢复执行。
- ◆ 理论与实践相结合：通过编写线程函数并观察它们的执行，本人能够将理论知识应用到实践中，加深了对操作系统工作原理的认识。

Section 6 附录：参考资料清单

- 指导书网站：[SYSU-2023-Spring-Operating-System: 中山大学 2023 学年春季操作系统课程 - Gitee.com](#)
- [详解操作系统内核对线程的调度算法 - 知乎 \(zhihu.com\)](#)
- [c/c++中 printf 函数的底层实现_printf 底层实现-CSDN 博客](#)