

武汉大学国家网络安全学院 教学实验报告



课程名称	操作系统设计与实践	实验日期	2024年11月4日
实验名称	多进程与进程调度	实验周次	第9周
姓名	学号	专业	班级
黄东威	2022302181148	信息安全	5班
王浚杰	2022302181143	网络空间安全	5班
程序	2022302181131	信息安全	4班
周业营	2022302181145	信息安全	5班

Contents

1 实验目的及实验内容	3
1.1 实验目的	3
1.2 实验内容	3
1.3 本次实验要解决的问题	3
2 实验环境	3
3 实验步骤	4
3.1 多进程问题，如何扩展单进程到多进程？	4
3.1.1 添加一个进程体	4
3.1.2 修改宏定义	4
3.1.3 进程表初始化代码扩充	5
3.1.4 LDT 初始化代码	6
3.2 如何扩展中断支持多进程？	7
3.3 如何实现系统调用	8
3.3.1 实现过程	8
3.3.2 改进和应用	11
3.4 进程调度问题，弄清楚实现调度的基本思路	12
4 实验过程分析	17
4.1 设计多级反馈队列时出现缺页错误异常	17
5 实验结果总结	17
5.1 在单进程的基础上扩展实现多进程要考虑哪些问题？	17
5.2 画出以下关键技术的流程图：初始化多进程控制块的过程、扩展初始化 LDT 和 TSS	19
5.3 如何修改时钟中断来支持多进程管理，画出新的流程图	20
5.4 系统调用的基本框架是如何的，应该包含哪些基本功能，画出流程图	20
5.5 如何操控可编程计数器？	20
5.5.1 原理	20
5.5.2 相关代码	21
5.6 进程调度的框架是怎样的？优先级调度如何实现	22
5.7 动手做：修改例子程序的调度算法，模拟实现一个多级反馈队列调度算法，并用其尝试调度多个任务。注意，抢占问题，注意时间片问题。鼓励使用其他更复杂的调度算法，如 CFS 等	22
5.7.1 作用	22
5.7.2 基本概念	22
5.7.3 代码实现	23
5.8 思考题：从用户态进程读和写内核段的数据，看能否成功	29
5.9 实验改进意见	29
6 各人实验贡献与体会（每人各自撰写）	29
7 教师评语	31

1 实验目的及实验内容

(本次实验所涉及并要求掌握的知识；实验内容；必要的原理分析)

1.1 实验目的

1. 多进程的实现机理与进程调度
2. 对应章节：第六章 6.4、6.5、6.6

1.2 实验内容

1. 多进程问题，如何扩展单进程到多进程，如何扩展中断支持多进程？
2. 如何实现系统调用
3. 进程调度问题，弄清楚实现调度的基本思路

1.3 本次实验要解决的问题

1. 在单进程的基础上扩展实现多进程要考虑哪些问题？
2. 画出以下关键技术的流程图：初始化多进程控制块的过程、扩展初始化 LDT 和 TSS
3. 如何修改时钟中断来支持多进程管理，画出新的流程图
4. 系统调用的基本框架是如何的，应该包含哪些基本功能，画出流程图。
5. 如何操控可编程计数器？
6. 进程调度的框架是怎样的？优先级调度如何实现？
7. 动手做：修改例子程序的调度算法，模拟实现一个多级反馈队列调度算法，并用其尝试调度多个任务。注意，抢占问题，注意时间片问题。鼓励使用其他更复杂的调度算法，如 CFS 等。
8. 思考题：从用户态进程读和写内核段的数据，看能否成功。

2 实验环境

- Windows Subsystem for Linux 2 (WSL 2)
- Ubuntu 20.04
- NASM 2.14.02
- Bochs 2.7
- Visual Studio Code (VSCode)

3 实验步骤

3.1 多进程问题，如何扩展单进程到多进程？

在上一次创建和运行一个进程的过程中，我们已经学习过了需要定义和初始化四个部分的内容：添加和实现进程体（进程要实现的功能）、定义进程表（PCB）的内容、GDT（至少新增一个描述符指向新进程的 LDT）、TSS（帮助实现在不同特权级之间的切换）。现在我们来逐一进行。

3.1.1 添加一个进程体

扩展单进程为多进程，首先肯定要新加一个进程的内容，也就是新增一个进程体。

```
1  /*=====
2   *          TestB
3   *=====
4
5 void TestB()
6 {
7     int i = 0x1000;
8     while(1){
9         disp_str("B");
10        disp_int(i++);
11        disp_str(".");
12        delay(1);
13    }
14 }
```

3.1.2 修改宏定义

这部分内容是一个承上启下的内容，既包含对上一步中《添加一个进程体》的完善和补充，也包含在做下一个步骤《进程表初始化代码填充》之前需要下修改进程表的定义。在修改进程表的定义，因为在/kernel/main.c 中添加完一个进程体的内容之后，要想真正在我们的内核中调用这个进程，肯定还少不了对这个进程的声明和初始化定义，这部分内容就要对新增的进程进行声明和初始化定义。

第一步是要给出进程体地址。在上一次实验中我们对进程体的操作是在 kernel/main.c 中对 p_proc->regs.eip 进行赋值。

```
1  p_proc->regs.eip      = (u32)TestA;
```

但是如果我们每加一个进程就要重新（定制化）为每一个进程定制一份初始化定义显得太单纯，而且代码结构会显得太冗余了，所以需要通过结构化的代码实现自动化的处理。例如在多进程中，我们对进程体的定义是通过以下步骤实现的：

先针对进程级的任务定义出一个 s_task 结构体（在/include/proc.h 中），结构体中定义出进程的进程体地址（initial_eip），其中的 task_f 定义于/include/type.h 中。

```
1  typedef void      (*task_f)      ();
2  ...
3  typedef struct s_task {
4      task_f    initial_eip;
5      int       stacksize;
6      char     name[32];
```

```
7 } TASK;
```

这时候定义出来的还只是类型，创建变量是在 global.c 文件中定义的，同时在 global.h 中进行了声明。

```
1 PUBLIC TASK task_table[NR_TASKS] = {{TestA, STACK_SIZE_TESTA, "TestA"}  
2 ,{TestB, STACK_SIZE_TESTB, "TestB"},{TestC, STACK_SIZE_TESTC, "TestC"}  
3 };  
4 ...  
5 extern TASK task_table[];
```

在这样定义出进程体之后，我们就可以在 main.c 中这样给出进程体地址。

```
1 p_proc->regs.eip = (u32)p_task->initial_eip;
```

第二步定义和实现好堆栈大小我们在单进程中是在 /include/proc.h 中通过宏定义出来的，然后在 main.c 中进行初始化赋值：

```
1 /* Number of tasks */  
2 #define NR_TASKS 1  
3 /* stacks of tasks */  
4 #define STACK_SIZE_TESTA 0x8000  
5 #define STACK_SIZE_TOTAL STACK_SIZE_TESTA  
6 ...  
7 p_proc->regs.esp = (u32) task_stack + STACK_SIZE_TOTAL;
```

所以在多进程中我们也先从宏定义的层面入手修改，然后在 main.c 中循环赋值：

```
1 /* Number of tasks */  
2 #define NR_TASKS 3  
3 /* stacks of tasks */  
4 #define STACK_SIZE_TESTA 0x8000  
5 #define STACK_SIZE_TESTB 0x8000  
6 #define STACK_SIZE_TESTC 0x8000  
7 #define STACK_SIZE_TOTAL (STACK_SIZE_TESTA + \  
8     STACK_SIZE_TESTB + \  
9     STACK_SIZE_TESTC)  
10 ...  
11 p_task_stack -= p_task->stacksize;
```

第三步就是在头文件 /inlcude/proto.h 中将新添加的函数声明一下。

```
1 /* main.c */  
2 void TestA();  
3 void TestB();  
4 void TestC();
```

3.1.3 进程表初始化代码扩充

进程表的初始化在原来单进程的代码中，是针对 Test A 单独进行的，但在我们拓展为多进程之后为了避免代码的冗余，也需要将其变得自动化一些，那么就用一个 for 循环来帮我们干这件事吧。

```
1 // 初始化任务和进程的指针，以及任务栈指针和LDT选择子  
2 TASK* p_task = task_table;
```

```

3      PROCESS*      p_proc      = proc_table;
4      char*        p_task_stack    = task_stack + STACK_SIZE_TOTAL;
5      u16         selector_ldt   = SELECTOR_LDT_FIRST;
6      // 对每一个任务，设置其对应进程的各个属性，并更新任务栈和进程、任务指
7      // 针
8      int i;
9      for (i = 0; i < NR_TASKS; i++) {
10         strcpy(p_proc->p_name, p_task->name); // 设置进程的名字
11         p_proc->pid = i; // 设置进程的PID
12
13         p_proc->ldt_sel = selector_ldt; // 设置进程的LDT选择子
14
15         // 设置进程的LDT中的两个描述符，一个为代码段，一个为数据段和栈段
16         memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS >> 3],
17                sizeof(DESCRIPTOR));
18         p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5;
19         memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS >> 3],
20                sizeof(DESCRIPTOR));
21         p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5;
22         // 设置进程的段寄存器
23         p_proc->regs.cs = ((8 * 0) & SA_RPL_MASK & SA_TI_MASK)
24             | SA_TIL | RPL_TASK;
25         p_proc->regs.ds = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
26             | SA_TIL | RPL_TASK;
27         p_proc->regs.es = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
28             | SA_TIL | RPL_TASK;
29         p_proc->regs.fs = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
30             | SA_TIL | RPL_TASK;
31         p_proc->regs.ss = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
32             | SA_TIL | RPL_TASK;
33         p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK)
34             | RPL_TASK;
35         // 设置进程的EIP、ESP和EFLAGS寄存器
36         p_proc->regs.eip = (u32)p_task->initial_eip;
37         p_proc->regs.esp = (u32)p_task_stack;
38         p_proc->regs.eflags = 0x1202; /* IF=1, IOPL=1 */
39         // 更新任务栈指针，进程指针，任务指针和LDT选择子
40         p_task_stack -= p_task->stacksize;
41         p_proc++;
42         p_task++;
43         selector_ldt += 1 << 3;
}

```

3.1.4 LDT 初始化代码

每一个进程都需要在 GDT 中有自己的 LDT。在上诉进程表初始化的过程中，对每一个进程 LDT 选择子的位置进行了定义。

```
p_proc->ldt_sel = selector_ldt; // 设置进程的LDT选择子
```

然后对每一个 LDT 的代码段和数据段进行了定义。

```

1 // 设置进程的LDT中的两个描述符，一个为代码段，一个为数据段和栈段
2 memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS >> 3], sizeof(
3     DESCRIPTOR));
4
4 p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5; memcpy(&p_proc->
5     ldts[1], &gdt[SELECTOR_KERNEL_DS >> 3], sizeof(DESCRIPTOR));
6
6 p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5;

```

3.2 如何扩展中断支持多进程？

对于如何扩展中断支持多进程这个问题的理解应该是如何通过修改一个中断使得这个中断可以实现多进程之间的切换。切换进程的过程是

1. 将 esp 指向将要切换到的进程的 PCB
2. 执行 lldt 将 ldt 切换到将要运行的进程的 ldt 处
3. 使用一系列 pop 恢复各个寄存器的值

重新回到代码中，发现原来在时钟中断中做的唤醒一个进程的代码做了类似的事情。

```

1 mov esp, [p_proc_ready] ; 离开内核栈
2
3 lea eax, [esp + P_STACKTOP]
4 mov dword [tss + TSS3_S_SP0], eax
5
6 .re_enter: ; 如果(k_reenter != 0)，会跳转到这里
7     dec dword [k_reenter]
8     pop gs ; .
9     pop fs ; /
10    pop es ; / 恢复原寄存器值
11    pop ds ; /
12    popad ; /
13    add esp, 4
14
15    iretd

```

那么现在只需要在这之前调用一个函数来将即将跳转到的进程放到 p_proc_ready 指向之处就可以了，在新建的 clock.c 文件中实现这个函数，并在时钟中断处调用这个函数。

```

1 /*=====
2          clock_handler
3 =====*/
4
4 PUBLIC void clock_handler(int irq)
5 {
6     disp_str("#");
7     p_proc_ready++;
8     if (p_proc_ready >= proc_table + NR_TASKS)
9         p_proc_ready = proc_table;
10 }

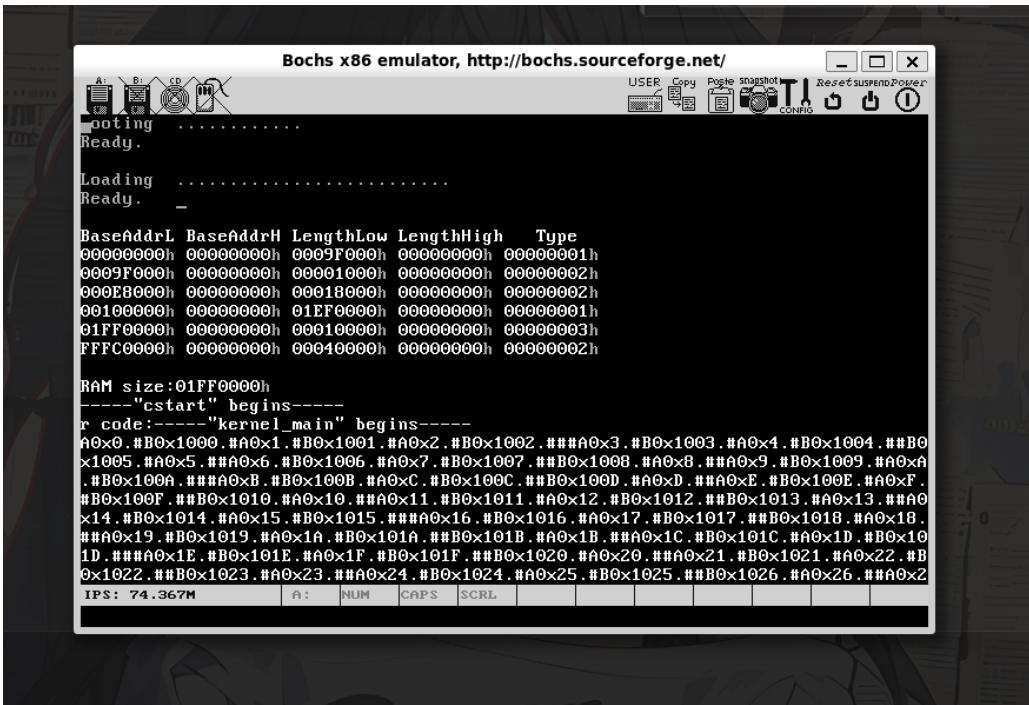
```

```

11
12 ...
13
14 /*=====
15          kernel.asm 中 hwint00 处
16 =====*/
17
18     sti
19
20     push    0
21     call    clock_handler
22     add esp, 4
23
24     cli

```

实现的效果如下：



可以看到在切换的过程中输出了‘#’并且实现了从 Test A 输出 A 到 Test B 输出 B 的转换。

3.3 如何实现系统调用

系统调用是应用程序和操作系统之间的桥梁。一件事情可能是应用程序做一部分，操作系统做另外一部分，因此，完成一个系统调用涉及到特权级的变换。

下面以实现一个获取时钟 tick 的系统调用为例，讲述如何实现系统调用。

3.3.1 实现过程

实现一个叫做 int get_ticks() 的函数，用于得到当前发生了多少次时钟中断

- get_ticks() 代码如下：

```

1 %include "sconst.inc"
2
3 _NR_get_ticks      equ 0 ; 要跟 global.c 中 sys_call_table 的定义相
4   对应!
5 INT_VECTOR_SYS_CALL equ 0x90
6
7 global get_ticks ; 导出符号
8
9 bits 32
10 [section .text]
11
12 get_ticks:
13   mov eax, _NR_get_ticks
14   int INT_VECTOR_SYS_CALL
15   ret

```

我们将 _NR_get_ticks 的值赋给了 eax，同时将该系统调用的中断号设置为 0x90。

- 完成中断门的初始化，将下面代码添加到 protect.c 的 init_prot 函数中

```

1     init_idt_desc(INT_VECTOR_SYS_CALL, DA_386IGate,
2                     sys_call,             PRIVILEGE_USER);

```

- 修改 save 函数，将 eax 寄存器改成 esi 寄存器，因为 eax 中存储的是系统调用的功能号，如果继续使用 eax 则会覆盖这个系统调用中断的功能号，所以将 eax 替换为一个在上下文中都未使用过的 esi 寄存器。

```

1 save:
2   pushad           ; `.
3   push  ds          ; /
4   push  es          ; / 保存原寄存器值
5   push  fs          ; /
6   push  gs          ; /
7   mov   dx, ss
8   mov   ds, dx
9   mov   es, dx
10
11  mov   esi, esp           ; esi = 进程表起始地址
12
13  inc   dword [k_reenter]    ; k_reenter++;
14  cmp   dword [k_reenter], 0  ; if(k_reenter == 0)
15  jne   .1                  ; {
16  mov   esp, StackTop        ;   mov esp, StackTop <-
17  push  restart              ;   push restart
18  jmp   [esi + RETADR - P_STACKBASE]; return;
19 .1:                         ; } else { 已经在内核栈,
20  不需要再切换
21  push  restart_reenter     ;   push restart_reenter
22  jmp   [esi + RETADR - P_STACKBASE]; return;

```

- 函数 sys_call 基本上是 hwint_master 的简化，对相应处理程序的调用也是类似的，在 hwint_master 中是 call [irq_table+4%1] (即调用了 irq_table[*1])，这里变成 call [sys_call_table+eax*4] (调用的是 sys_call_table [eax])。代码如下：

```

1 sys_call:
2     call    save
3
4     sti
5
6     call    [sys_call_table + eax * 4]
7     mov     [esi + EAXREG - P_STACKBASE], eax
8
9     cli
10
11    ret

```

- 将 sys_call_table 在 global.c 中定义，system_call 在 type.h 中定义

```

1 PUBLIC system_call sys_call_table[NR_SYS_CALL] = {
2     sys_get_ticks};
3
4
5     typedef void* system_call;

```

- 实现 sys_get_ticks 函数功能（目前为打印字符“+”）

```

1 PUBLIC int sys_get_ticks()
2 {
3     disp_str("++");
4     return 0;
5 }

```

- 在 proto.h 中添加函数声明

```

1 PUBLIC int sys_get_ticks();           /* sys_call */
2
3 /* syscall.asm */
4 PUBLIC void sys_call();              /* int_handler */
5 PUBLIC int get_ticks();

```

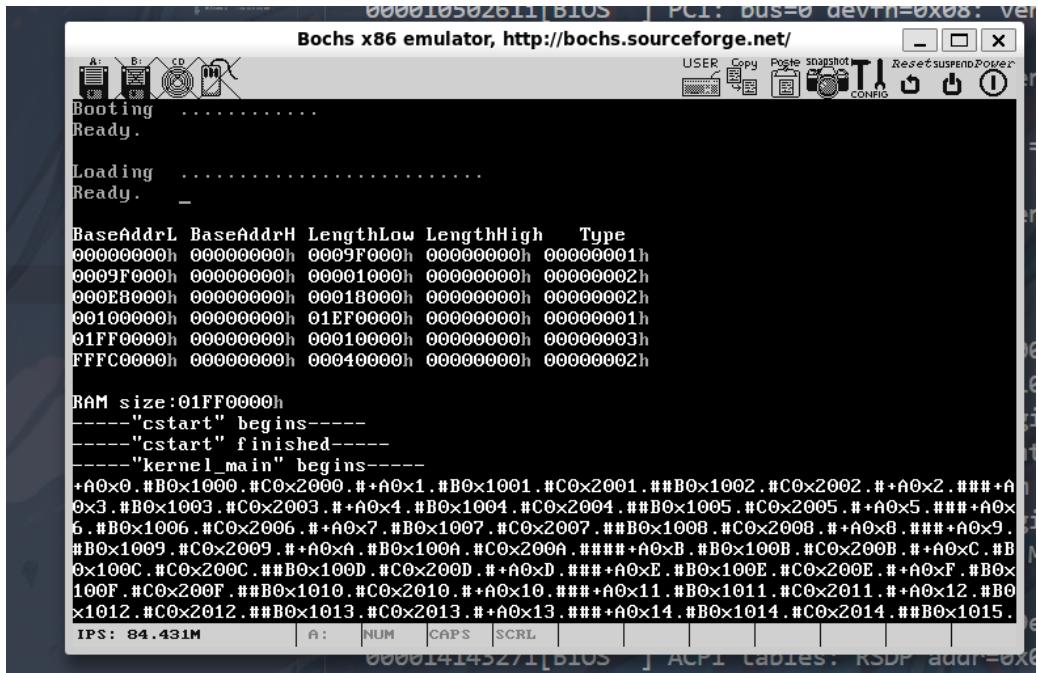
- 在进程中添加调用 get_ticks 的代码

```

1 void TestA()
2 {
3     int i = 0;
4     while (1) {
5         get_ticks();
6         disp_str("A");
7         disp_int(i++);
8         disp_str(".");
9         delay(1);
10    }
11 }

```

- 在 kernel.asm 和 syscall.asm 中添加导入导出相应符号后修改 Makefile 运行



可以看到，每次打印”A”字符之前都会打印一个”+”字符，证明系统调用成功添加

3.3.2 改进和应用

- 改进函数 sys_get_ticks，它返回当前的 ticks，声明一个全局变量 ticks，在 clock_handler 中让其递增。

```

1 PUBLIC void clock_handler(int irq)
2 {
3     disp_str("#");
4     ticks++;
5
6     if (k_reenter != 0) {
7         disp_str("!!");
8         return;
9     }
10
11    p_proc_ready++;
12
13    if (p_proc_ready >= proc_table + NR_TASKS) {
14        p_proc_ready = proc_table;
15    }
16}

```

```

1 PUBLIC int sys_get_ticks()
2 {
3     return ticks;
4 }

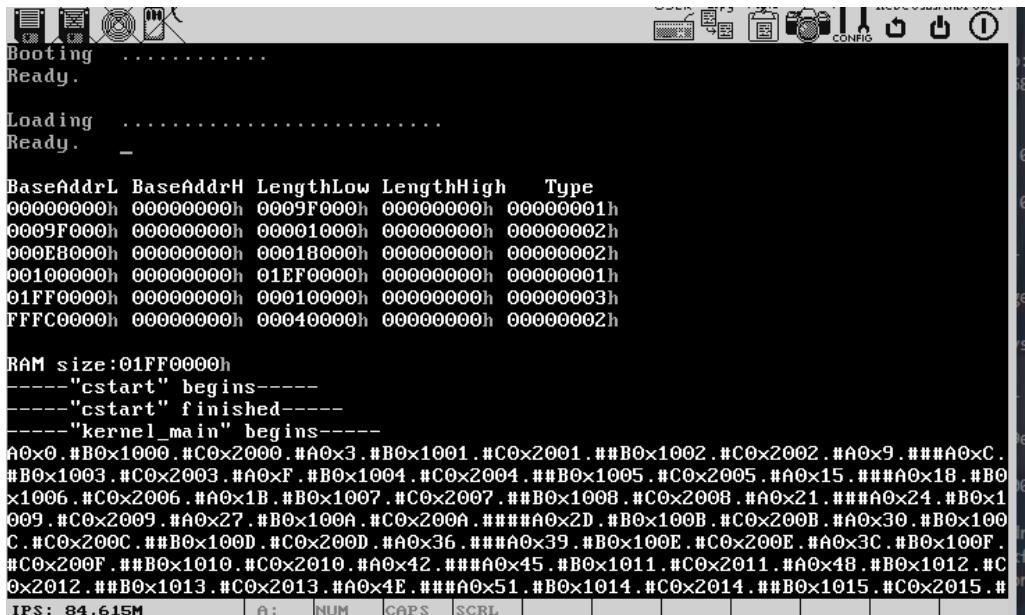
```

- 修改 TestA()，让其不打印递增的 i，转而打印当前的 ticks

```

1 void TestA()
2 {
3     int i = 0;
4     while (1) {
5         disp_str("A");
6         disp_int(get_ticks());
7         disp_str(".");
8         delay(1);
9     }
10 }
```

效果如下：



可以看到两个 A 进程后紧跟的数字，和两个 A 之间打印的“#”数目对得上。

3.4 进程调度问题，弄清楚实现调度的基本思路

orange ‘s 一书实现进程调度的方法是通过将不同的进程延迟不同的时间，这样各自运行的时间不同而实现调度，以此为例，我们探究如何实现调度。首先，利用 mili_delays 函数，让三个进程延迟不同的时间（A、B、C 分别延迟 300、900、1500ms）

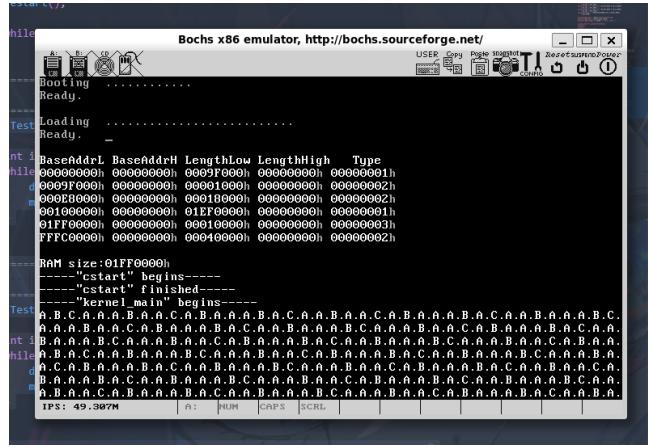
```

1 void TestA()
2 {
3     int i = 0;
4     while (1) {
5         disp_str("A.");
6         milli_delay(300);
7     }
8 }
9
10 /*=====
11          TestB
```

```

12  ****
13 void TestB()
14 {
15     int i = 0x1000;
16     while(1){
17         disp_str("B.");
18         milli_delay(900);
19     }
20 }
21 /**
22          TestB
23 */
24 ****
25 void TestC()
26 {
27     int i = 0x2000;
28     while(1){
29         disp_str("C.");
30         milli_delay(1500);
31     }
32 }

```



可以统计得到打印的 ABC 字符数量比值约为 5: 3: 1 和延迟时间有反比关系我们在此基础上实现进程的优先级。给每一个进程添加一个变量，用这个变量当作进程的“生命周期”，进程每获得一个运行周期，该变量就减一，当减到 0 时，此进程不再获得执行的机会，直到所有进程的变量都减到 0 为止。给进程结构体添加 priority 和 ticks 变量，赋值如下：

```

1 proc_table[0].ticks = proc_table[0].priority = 150;
2 proc_table[1].ticks = proc_table[1].priority = 50;
3 proc_table[2].ticks = proc_table[2].priority = 30;

```

编写 schedule 函数，该函数是实现根据三个队列中的进程情况，以及每个进程在列表之中的剩余时间，来实现确定 p_proc_ready 的指向，以及进程在三个队列之间的调度。

```

1 PUBLIC void schedule()
2 {

```

```

3     PROCESS* p;
4     int greatest_ticks = 0;
5
6     while (!greatest_ticks) {
7         for (p = proc_table; p < proc_table+NR_TASKS; p++) {
8             if (p->ticks > greatest_ticks) {
9                 greatest_ticks = p->ticks;
10                p_proc_ready = p;
11            }
12        }
13
14        if (!greatest_ticks) {
15            for (p = proc_table; p < proc_table+NR_TASKS; p++) {
16                p->ticks = p->priority;
17            }
18        }
19    }
20}

```

同时对时钟中断处理函数修改一下，这样在一个进程没有运行结束 (ticks>0) 之前就不会发生切换进程的 schedule 操作。

```

1 PUBLIC void clock_handler(int irq)
2 {
3     ticks++;
4     p_proc_ready->ticks--;
5
6     if (k_reenter != 0) {
7         return;
8     }
9
10    schedule();
11}

```

将 ABC 进程的延迟改成相同的

```

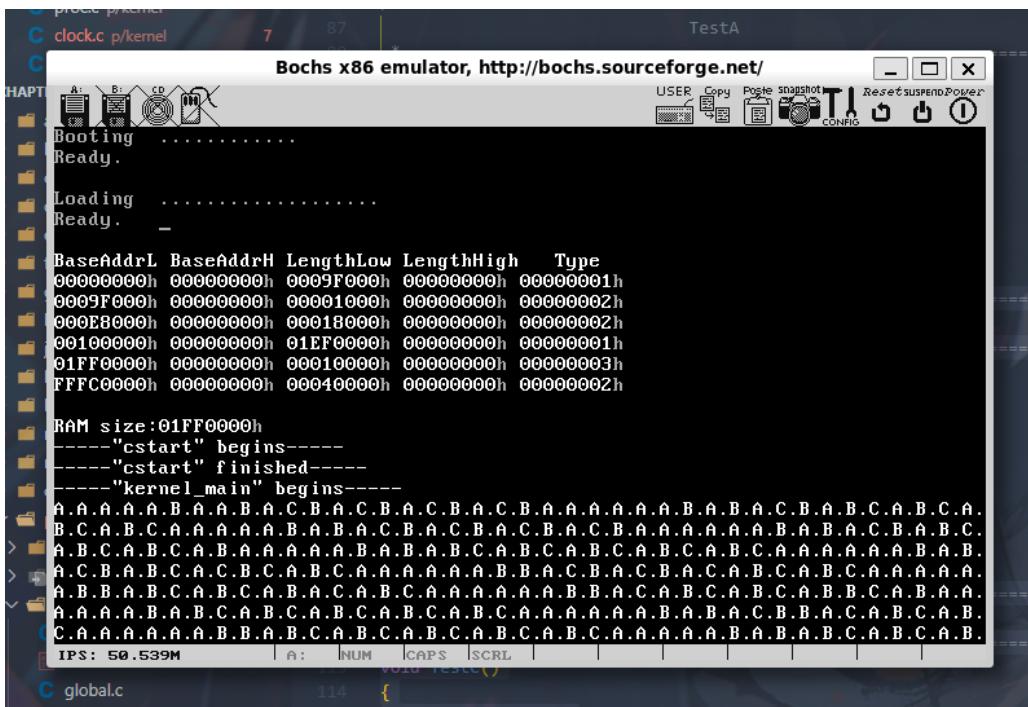
1 void TestA()
2 {
3     int i = 0;
4     while (1) {
5         disp_str("A.");
6         milli_delay(200);
7     }
8
9
10 /*=====
11          TestB
12 =====*/
13
14 void TestB()
15 {
16     int i = 0x1000;
17     while(1){

```

```

17     disp_str("B.");
18     milli_delay(200);
19 }
20
21 /*=====
22          TestB
23 =====*/
24
25 void TestC()
26 {
27     int i = 0x2000;
28     while(1){
29         disp_str("C.");
30         milli_delay(200);
31     }
32 }
```

效果如下：



可以看到，虽然修改了延迟时间，但因为优先级的不同，ABC 字母的打印数量是不一致的。为了更好观察效果，将 ABC 打印时带上颜色，修改 schedule 同时添加清屏函数

```

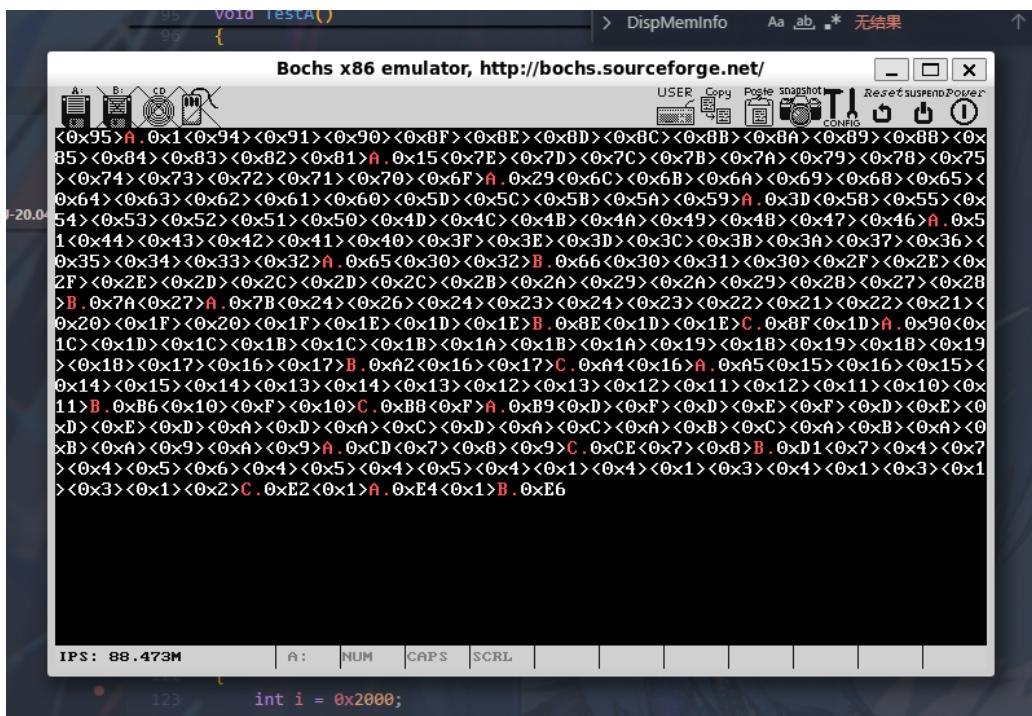
1 PUBLIC void schedule()
2 {
3     PROCESS* p;
4     int greatest_ticks = 0;
5
6     while (!greatest_ticks) {
7         for (p = proc_table; p < proc_table+NR_TASKS; p++) {
8             if (p->ticks > greatest_ticks) {
9                 disp_str("<");
```

```

10     disp_int(p->ticks);
11     disp_str(">");
12     greatest_ticks = p->ticks;
13     p_proc_ready = p;
14 }
15 }
16
17 /* if (!greatest_ticks) { */
18 /*   for (p = proc_table; p < proc_table+NR_TASKS; p++) { */
19 /*     p->ticks = p->priority; */
20 /*   } */
21 /* } */
22 }
23 }
```

```

1 disp_pos = 0;
2 for (i = 0; i < 80*25; i++) {
3   disp_str(" ");
4 }
5 disp_pos = 0;
```



在 3 个阶段中，最初阶段的时间跨度为 100ticks，之后由于 A 的 ticks 值小于 50，已经与进程 B 的 ticks 值相当，所以之后会同时受到 A 和 B 的调度。在最后一个阶段，就变成 A、B、C 三个进程同时受到调度。现在每个进程运行的时间比较长，如果修改运行时间，能够更好的观察到 ABC 字符打印数量比接近 15: 5: 5。

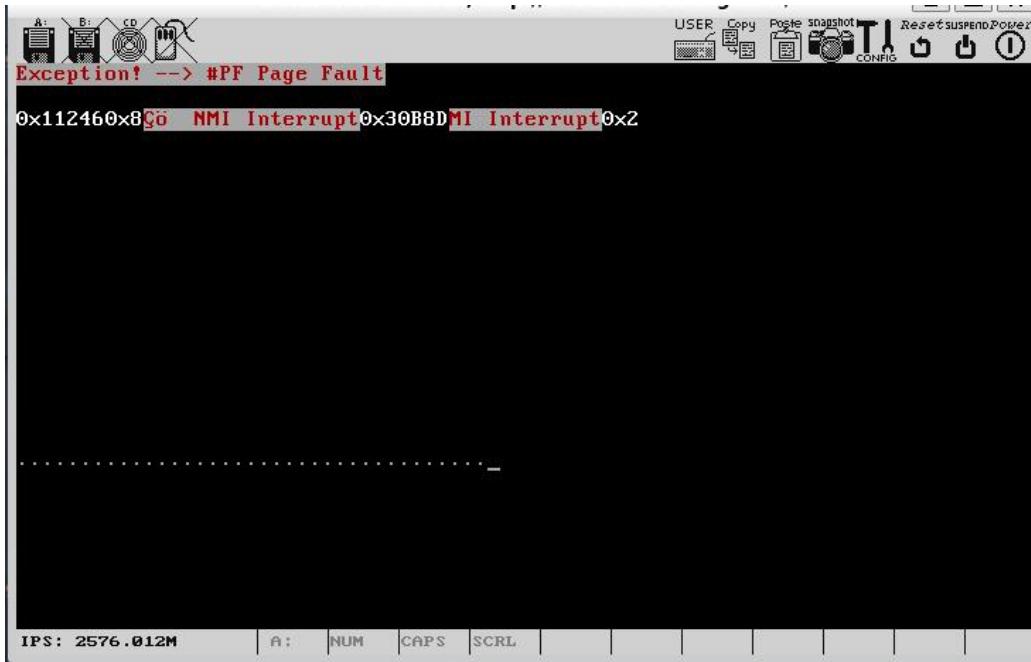
4 实验过程分析

(详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格等)

4.1 设计多级反馈队列时出现缺页错误异常

问题描述：在设计多级反馈队列时，为了获取每级队列的长度，定义了 Q1_len、Q2_len、Q3_len 三个变量分别存储三级队列的长度，但在输出队列长度时出现了缺页错误。

```
1 =====global.h=====
2 EXTERN u32 Q1_len, Q2_len, Q3_len;
3
4 =====proc.c=====
5 dis_int(Q1_len);
```



原因分析：在 C 语言中，EXTERN u32 Q1_len, Q2_len, Q3_len; 的定义仅仅声明了这三个变量的类型和作用范围，而并没有分配实际的内存空间。因此，编译器认为这些变量在其他地方已经被定义并初始化过了。如果没有这样的定义和初始化，程序运行时访问这些变量会导致未定义的行为，例如缺页错误（page fault），因为变量没有实际的存储空间。

解决方案：在定义 Q1_len、Q2_len、Q3_len 时赋初值为 0，分配实际的内存空间。

```
1 =====global.h=====
2 EXTERN u32 Q1_len = 0, Q2_len = 0, Q3_len = 0;
```

5 实验结果总结

5.1 在单进程的基础上扩展实现多进程要考虑哪些问题？

在单进程的基础上扩展实现多进程需要注意以下几个问题：

1. 定义出进程体 (main.c)。

```
1      /*=====
2          TestC
3      =====*/
4 void TestC()
5 {
6     int i = 0x2000;
7     while(1){
8         disp_str("C");
9         disp_int(i++);
10        disp_str(".");
11        delay(1);
12    }
13 }
```

2. 修改原有变量和宏定义。 (global.c、 proc.h)

先定义和实现一个任务表，存储不同进程的地址、栈段大小和名字。

```
1 PUBLIC TASK task_table[NR_TASKS] = {{TestA,
2     STACK_SIZE_TESTA, "TestA"}, {TestB, STACK_SIZE_TESTB, "
3     TestB"}, {TestC, STACK_SIZE_TESTC, "TestC"}};
```

然后修改进程数的宏定义和栈段大小的定义。

```
1 /* Number of tasks */
2 #define NR_TASKS 3
3 /* stacks of tasks */
4 #define STACK_SIZE_TESTA 0x8000
5 #define STACK_SIZE_TESTB 0x8000
6 #define STACK_SIZE_TESTC 0x8000
7 #define STACK_SIZE_TOTAL (STACK_SIZE_TESTA + STACK_SIZE_TESTB +
8     STACK_SIZE_TESTC)
```

3. 进行函数声明。 (proto.h)

```
1 /* main.c */
2 void TestA();
3 void TestB();
4 void TestC();
```

4. 初始化进程表和 LDT。 (main.c)

```
1     for (i = 0; i < NR_TASKS; i++) {
2         strcpy(p_proc->p_name, p_task->name); // name of the
3             process
4         p_proc->pid = i; // pid
5
6         p_proc->ldt_sel = selector_ldt;
```

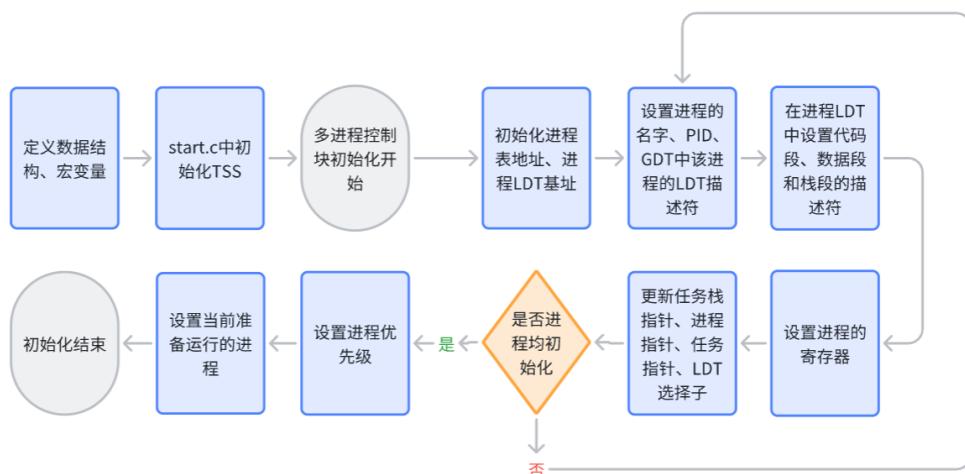
```

7     memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS >> 3],
8             sizeof(DESCRIPTOR));
9     p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5;
10    memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS >> 3],
11            sizeof(DESCRIPTOR));
12    p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5;
13    p_proc->regs.cs = ((8 * 0) & SA_RPL_MASK & SA_TI_MASK)
14        | SA_TIL | RPL_TASK;
15    p_proc->regs.ds = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
16        | SA_TIL | RPL_TASK;
17    p_proc->regs.es = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
18        | SA_TIL | RPL_TASK;
19    p_proc->regs.fs = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
20        | SA_TIL | RPL_TASK;
21    p_proc->regs.ss = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
22        | SA_TIL | RPL_TASK;
23    p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK)
24        | RPL_TASK;

25
26    p_proc->regs.eip = (u32)p_task->initial_eip;
27    p_proc->regs.esp = (u32)p_task_stack;
28    p_proc->regs.eflags = 0x1202; /* IF=1, IOPL=1 */
29
30    p_task_stack -= p_task->stacksize;
31    p_proc++;
32    p_task++;
33    selector_ldt += 1 << 3;
34}

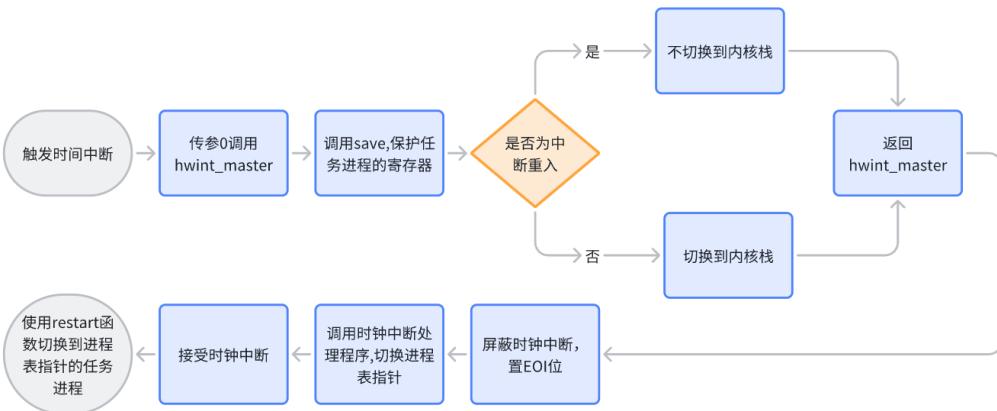
```

5.2 画出以下关键技术的流程图：初始化多进程控制块的过程、扩展初始化 LDT 和 TSS



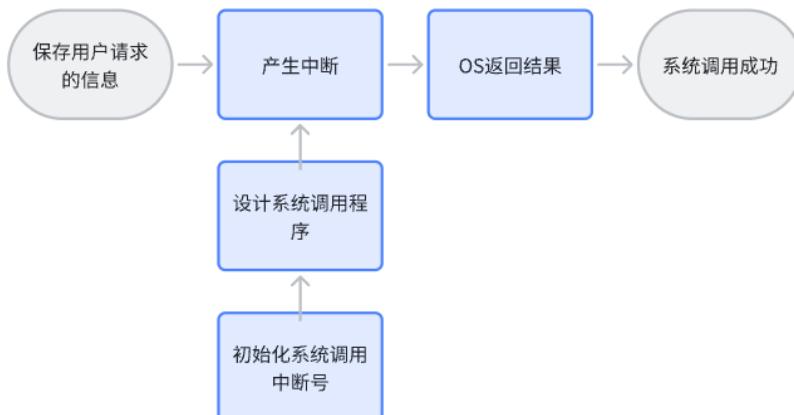
5.3 如何修改时钟中断来支持多进程管理，画出新的流程图

修改时钟中断来支持多进程管理的部分请见3.2部分。



5.4 系统调用的基本框架是如何的，应该包含哪些基本功能，画出流程图

系统调用的过程包括保存用户请求的信息、触发中断、操作系统处理请求并返回结果，最终完成系统调用。具体来说，系统调用首先会初始化中断号并设计对应的中断处理程序，将中断号与处理程序在中断向量表中关联，完成中断初始化。当用户程序发起系统调用时，通过触发中断切换到操作系统内核，内核根据中断号进入相应的处理程序执行请求并返回结果，从而实现系统调用的完整流程。



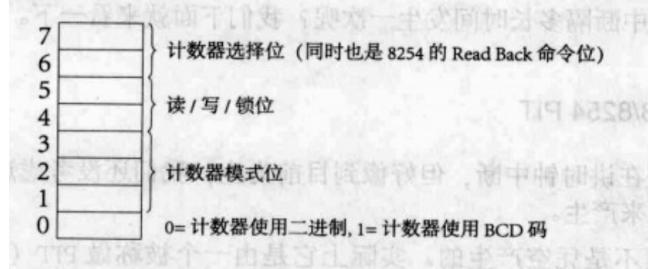
5.5 如何操控可编程计数器？

5.5.1 原理

8253 有 3 个计数器，它们都是 16 位的，各有不同的作用，如下表所示：

计数器	作用
Counter0	输出到 IRQ0，以便每隔一段时间让系统产生一次时钟中断
Counter1	通常被设为 18，以便大约每 15 μs 做一次 RAM 刷新
Counter2	连接 PC 喇叭

时钟中断实际上是由 8253 的 Counter0 产生的，计数器的工作原理是这样的：PC 上计数器的输入频率是 1193180Hz，每一个时钟周期，值会减 1，减到 0 时，就会触发一个输出。默认的频率是 18.2Hz。可以通过 8253 的端口改变计数器的计数值，其中改变 Counter0 需要操作端口 40h，需要先通过 43h 写 8253 模式控制寄存器。8253 控制寄存器模式如下：



而其中计数器模式位如下：

模式位值	模式	名称
0 0 0	模式 0	interrupt on terminal count
0 0 1	模式 1	programmable one-shot
0 1 0	模式 2	rate generator ← 我们的时钟中断采用此模式
0 1 1	模式 3	square wave rate generator
1 0 0	模式 4	software triggered strobe
1 0 1	模式 5	hardware triggered strobe

如果要操作 Counter0，第 7、6 位应该是 00，低字节和高字节都要写入，因此第 5、4 位是 11，使用模式 2，因此第 3、2、1 应该是 010。第 0 位是 0

5.5.2 相关代码

设置计数值

```

1  /* 初始化 8253 PIT */
2  out_byte(TIMER_MODE, RATE_GENERATOR);
3  out_byte(TIMER0, (u8) (TIMER_FREQ/HZ));
4  out_byte(TIMER0, (u8) ((TIMER_FREQ/HZ) >> 8));

```

有关 8253 的宏定义

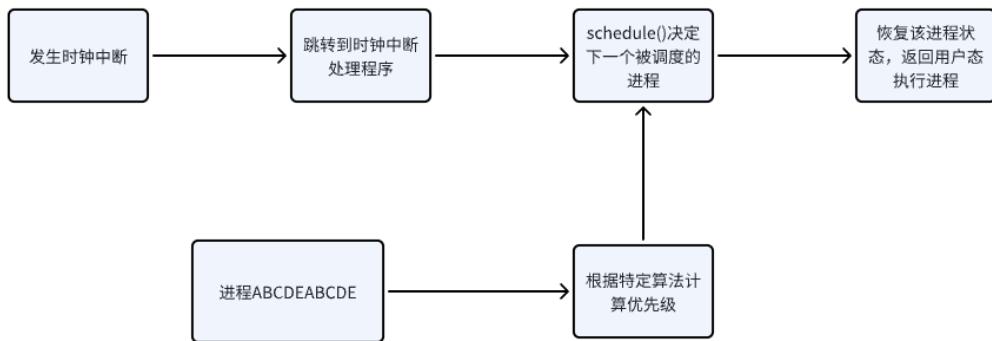
```

1  /* 8253/8254 PIT (Programmable Interval Timer) */
2  #define TIMER0          0x40 /* I/O port for timer channel 0 */
3  #define TIMER_MODE       0x43 /* I/O port for timer mode control */
4  #define RATE_GENERATOR  0x34 /* 00-11-010-0 :
5           * Counter0 - LSB then MSB - rate generator - binary
6           */
7  #define TIMER_FREQ      1193182L/* clock frequency for timer in PC and AT
8           */
#define HZ              100 /* clock freq (software settable on IBM-PC)

```

5.6 进程调度的框架是怎样的？优先级调度如何实现

进程调度的框架如图所示：



优先级调度算法有很多类型，示例代码中是在初始化时划定进程的优先级，并且按照下图的逻辑进行调度

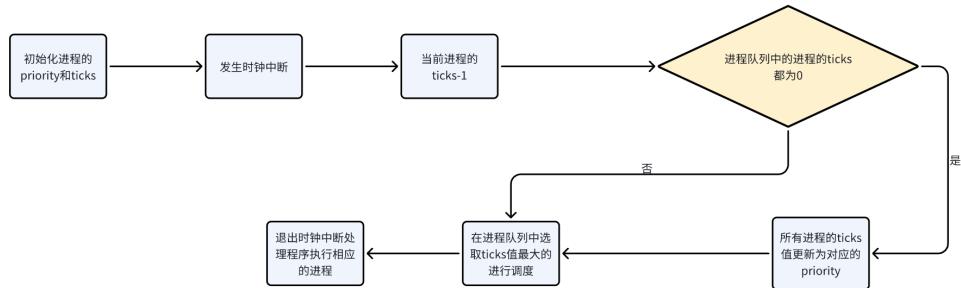


Figure 1: Enter Caption

在代码实现时，将优先级有关的信息以属性的方式加在进程表结构体中，在每次进行调度时，根据属性值计算出优先级，然后在必要的时候对属性值更新。

5.7 动手做：修改例子程序的调度算法，模拟实现一个多级反馈队列调度算法，并用其尝试调度多个任务。注意，抢占问题，注意时间片问题。鼓励使用其他更复杂的调度算法，如 CFS 等

5.7.1 作用

如果有很多任务排队等着被处理，哪个任务先被处理，哪个任务后处理，这个需要由操作系统决定，这就是调度。多级反馈队列调度算法是目前操作系统调度算法中被公认的一种较好的调度算法。它可以满足各种类型进程的需要，既能使高优先级的作业得到响应又能使短作业（进程）迅速完成。

5.7.2 基本概念

多级反馈队列调度算法是一种根据先来先服务原则给就绪队列排序，为就绪队列赋予不同的优先级数，不同的时间片，按照优先级抢占 CPU 的调度算法。算法的实施过程如下：

- 按照先来先服务原则排序，设置 N 个就绪队列为 Q_1, Q_2, \dots, Q_N ，每个队列中都可以放很多作业。

- 为这 N 个就绪队列赋予不同的优先级，第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。
- 设置每个就绪队列的时间片，优先权越高，算法赋予队列的时间片越小。时间片大小的设定按照实际作业（进程）的需要调整。
- 进程在进入待调度的队列等待时，首先进入优先级最高的 Q_1 等待。
- 首先调度优先级高的队列中的进程。若高优先级中队列中已没有调度的进程，则调度次优先级队列中的进程。例如： Q_1, Q_2, Q_3 三个队列，只有在 Q_1 中没有进程等待时才去调度 Q_2 ，同理，只有 Q_1, Q_2 都为空时才会去调度 Q_3 。
- 对于同一个队列中的各个进程，按照时间片轮转法调度。比如 Q_1 队列的时间片为 N ，那么 Q_1 中的作业在经历了时间片为 N 的时间后，若还没有完成，则进入 Q_2 队列等待，若 Q_2 的时间片用完后作业还不能完成，则进入下一级队列，直至完成。
- 在低优先级的队列中的进程在运行时，又有新到达的作业，那么在运行完这个时间片后，CPU 马上分配给新到达的作业，即抢占式调度 CPU。

5.7.3 代码实现

我们先定义五个进程，这样有足够的进程用于体现多级调度算法

```

1 PUBLIC  PROCESS          proc_table[NR_TASKS];
2
3 PUBLIC  char             task_stack[STACK_SIZE_TOTAL];
4
5 PUBLIC  TASK   task_table[NR_TASKS] = {{TestA, STACK_SIZE_TESTA, "TestA"
6   },
7   {TestB, STACK_SIZE_TESTB, "TestB"
8   },
9   {TestC, STACK_SIZE_TESTC, "TestC"
10  },
11  {TestD, STACK_SIZE_TESTD, "TestD"
12  },
13  {TestE, STACK_SIZE_TESTE, "TestE"
14  }
15  ;
16
17 PUBLIC  irq_handler      irq_table[NR_IRQ];
18
19 PUBLIC  system_call      sys_call_table[NR_SYS_CALL] = {sys_get_ticks};

```

在头文件中也加上关于 ABCDE 进程的定义

```

1 PUBLIC void delay(int time);
2
3 /* kernel.asm */
4 void restart();
5
6 /* main.c */
7 void TestA();
8 void TestB();

```

```

9 void TestC();
10 void TestD();
11 void TestE();
12 /* i8259.c */
13 PUBLIC void put_irq_handler(int irq, irq_handler handler);
14 PUBLIC void spurious_irq(int irq);

15
16
17 EXTERN u32 Q1[NR_TASKS];
18 EXTERN u32 Q2[NR_TASKS];
19 EXTERN u32 Q3[NR_TASKS];
20 #define Q1_TICK 1
21 #define Q2_TICK 3
22 #define Q3_TICK 5
23 EXTERN u32 Q1_len;
24 EXTERN u32 Q2_len;
25 EXTERN u32 Q3_len;

```

接下来修改 PROCESS 结构体 NR_TASKS 需要修改为 5 其中 queue_ticks 是每个队列所分配的时间片

```

1 typedef struct s_proc {
2     STACK_FRAME regs;           /* process registers saved in stack frame
3     */
4
5     u16 ldt_sel;               /* gdt selector giving ldt base and limit
6     */
7     DESCRIPTOR ldts[LDT_SIZE]; /* local descriptors for code and data */
8
9     int ticks;                 /* remained ticks */
10    int priority;              /*各级队列中进程被分配的时间片数量 */
11    int queueticks;            /* process id passed in from MM */
12    u32 pid;                  /* name of the process */
13 }PROCESS;

```

在 main.c 中实现进程运行的时间

```

1 PUBLIC int kernel_main()
2 {
3     //disp_str("-----\"kernel_main\" begins-----\n");
4
5     TASK* p_task = task_table;
6     PROCESS* p_proc = proc_table;
7     char* p_task_stack = task_stack + STACK_SIZE_TOTAL;
8     u16 selector_ldt = SELECTOR_LDT_FIRST;
9     int i;
10    for (i = 0; i < NR_TASKS; i++) {
11        strcpy(p_proc->p_name, p_task->name); // name of the process
12        p_proc->pid = i;                      // pid
13        p_proc->ldt_sel = selector_ldt;
14
15        memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS >> 3],

```

```

16         sizeof(DESCRIPTOR));
17     p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5;
18     memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS >> 3],
19            sizeof(DESCRIPTOR));
20     p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5;
21     p_proc->regs.cs = ((8 * 0) & SA_RPL_MASK & SA_TI_MASK)
22         | SA_TIL | RPL_TASK;
23     p_proc->regs.ds = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
24         | SA_TIL | RPL_TASK;
25     p_proc->regs.es = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
26         | SA_TIL | RPL_TASK;
27     p_proc->regs.fs = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
28         | SA_TIL | RPL_TASK;
29     p_proc->regs.ss = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
30         | SA_TIL | RPL_TASK;
31     p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK)
32         | RPL_TASK;
33
34     p_proc->regs.eip = (u32)p_task->initial_eip;
35     p_proc->regs.esp = (u32)p_task_stack;
36     p_proc->regs.eflags = 0x1202; /* IF=1, IOPL=1 */
37
38     p_task_stack -= p_task->stacksize;
39     p_proc++;
40     p_task++;
41     selector_ldt += 1 << 3;
42 }
43
44 proc_table[0].ticks = proc_table[0].priority = 5;
45 proc_table[1].ticks = proc_table[1].priority = 4;
46 proc_table[2].ticks = proc_table[2].priority = 3;
47 proc_table[3].ticks = proc_table[3].priority = 4;
48 proc_table[4].ticks = proc_table[4].priority = 5;
49 k_reenter = 0;
50 ticks = 0;
51
52 p_proc_ready = proc_table;
53
54 /* 初始化 8253 PIT */
55 out_byte(TIMER_MODE, RATE_GENERATOR);
56 out_byte(TIMERO, (u8) (TIMER_FREQ/HZ));
57 out_byte(TIMERO, (u8) ((TIMER_FREQ/HZ) >> 8));
58
59 put_irq_handler(CLOCK_IRQ, clock_handler); /* 设定时钟中断处理程
60 序 */
61 enable_irq(CLOCK_IRQ); /* 让 8259A 可以接收时钟
62 中断 */
63 Q1_len = 0;
64 Q2_len = 0;
65 Q3_len = 0;
66 restart();

```

```

65     while(1){}
66 }
67 }
```

将时钟中断函数修改，用于控制在不同的时间，将 ABCDE 进程读入

```

1 PUBLIC void clock_handler(int irq) {
2     // 根据时间点将进程 A-E 加入 Q1 队列
3     if (ticks == 0) {
4         Q1[Q1_len++] = 0; // A
5         proc_table[Q1[Q1_len - 1]].queueticks = Q1_TICK;
6     }
7     if (ticks == 3) {
8         Q1[Q1_len++] = 1; // B
9         proc_table[Q1[Q1_len - 1]].queueticks = Q1_TICK;
10    }
11    if (ticks == 6) {
12        Q1[Q1_len++] = 2; // C
13        proc_table[Q1[Q1_len - 1]].queueticks = Q1_TICK;
14    }
15    if (ticks == 8) {
16        Q1[Q1_len++] = 3; // D
17        proc_table[Q1[Q1_len - 1]].queueticks = Q1_TICK;
18    }
19    if (ticks == 11) {
20        Q1[Q1_len++] = 4; // E
21        proc_table[Q1[Q1_len - 1]].queueticks = Q1_TICK;
22    }
23
24    // 增加系统时钟计数
25    ticks++;
26    if (ticks > 1) {
27        p_proc_ready->ticks--; // 减少当前进程的剩余时间
28    }
29
30    // 调用调度函数进行进程调度
31    schedule();
32 }
```

对 schedule 调度算法进行实现

```

1 PUBLIC void schedule() {
2     if (Q1_len != 0) { // Q1队列非空
3         int proc_id = Q1[0];
4         PROCESS* p = &proc_table[proc_id]; // 取出队首元素
5         p_proc_ready = p;
6
7         // 打印有关内容
8         disp_str("\n");
9         disp_str(p_proc_ready->p_name);
10        disp_str("[proc-ticks=");
11        disp_int(p->ticks);
12        disp_str(" & Q1-ticks=");
```

```

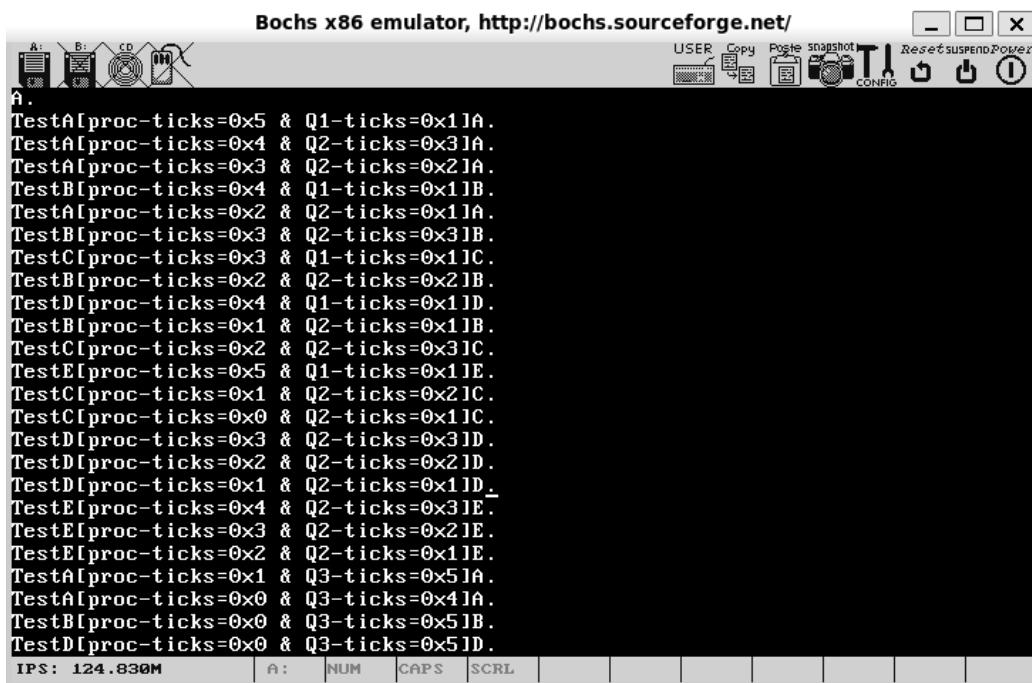
13     disp_int(p->queueticks);
14     disp_str("]");
15
16     p->queueticks--;
17
18     if (p->ticks <= 0) { // 该进程结束
19         Q1_len--;
20         for (int i = 0; i < Q1_len; i++)
21             Q1[i] = Q1[i + 1]; // 队首出队
22     } else if (p->queueticks <= 0) { // 时间片用完
23         Q1_len--;
24         for (int i = 0; i < Q1_len; i++)
25             Q1[i] = Q1[i + 1]; // 队首出队
26         p->queueticks = Q2_TICK;
27         Q2[Q2_len] = proc_id;
28         Q2_len++;
29     }
30 } else if (Q2_len != 0) { // Q2队列非空
31     int proc_id = Q2[0];
32     PROCESS* p = &proc_table[proc_id]; // 取出队首元素
33     p_proc_ready = p;
34
35     // 打印有关内容
36     disp_str("\n");
37     disp_str(p_proc_ready->p_name);
38     disp_str("[proc-ticks=");
39     disp_int(p->ticks);
40     disp_str(" & Q2-ticks=");
41     disp_int(p->queueticks);
42     disp_str("]");
43
44     p->queueticks--;
45
46     if (p->ticks <= 0) { // 该进程结束
47         Q2_len--;
48         for (int i = 0; i < Q2_len; i++)
49             Q2[i] = Q2[i + 1]; // 队首出队
50     } else if (p->queueticks <= 0) { // 时间片用完
51         Q2_len--;
52         for (int i = 0; i < Q2_len; i++)
53             Q2[i] = Q2[i + 1]; // 队首出队
54         p->queueticks = Q3_TICK;
55         Q3[Q3_len] = proc_id;
56         Q3_len++;
57     }
58 } else if (Q3_len != 0) { // Q3队列非空
59     int proc_id = Q3[0];
60     PROCESS* p = &proc_table[proc_id]; // 取出队首元素
61     p_proc_ready = p;
62
63     // 打印有关内容

```

```

64     disp_str("\n");
65     disp_str(p_proc_ready->p_name);
66     disp_str("[proc-ticks=");
67     disp_int(p->ticks);
68     disp_str(" & Q3-ticks=");
69     disp_int(p->queueticks);
70     disp_str("] ");
71
72     p->queueticks--;
73
74     if (p->ticks <= 0) { // 该进程结束
75         Q3_len--;
76         for (int i = 0; i < Q3_len; i++)
77             Q3[i] = Q3[i + 1]; // 队首出队
78     } else if (p->queueticks <= 0) { // 时间片用完
79         Q3_len--;
80         for (int i = 0; i < Q3_len; i++)
81             Q3[i] = Q3[i + 1]; // 队首出队
82         p->queueticks = Q3_TICK;
83         Q3[Q3_len] = proc_id; // 放回Q3队列队尾
84         Q3_len++;
85     }
86 } else { // 所有队列都为空，实验结束
87     while (1) {} // 死循环
88 }
89 }

```



可以看到打印的结果，我们用表格和甘特图观察一下结果：多级调度过程如下：

作业	到达时间	持续时间	Q1 时间片 (最高优先级)	Q2 时间片	Q3 时间片 (最低优先级)
A	0	5	1	3	5
B	3	4	1	3	5
C	6	3	1	3	5
D	8	4	1	3	5
E	11	3	1	3	5

时间	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
作业	A	A	A	B	B	A	A	C	C	A	D	D	E	E	A	A	B	B	D	D	E	E

可以看到，我们的实验结果和预期结果是一致的

5.8 思考题：从用户态进程读和写内核段的数据，看能否成功

- 如果使用地址读写内核数据段，可以成功
- 如果使用选择子，因为进程处于 ring1，要访问的描述符处于 ring0，会触发保护若想让进程就算直接使用地址也无法访问内核段的数据，应该合理填写进程的页表，让其线性地址无法获得内核段的物理地址。

5.9 实验改进意见

1. 希望能够提供更详细的实验指导：在每个实验步骤中，提供更详细的指导和说明，比如预期结果等，这样能帮助我们更好地完成实验。
2. 希望能够提供常见错误类型及纠错指南，帮助同学们在遇到问题的时候能够自己解决问题。
3. 希望老师能讲解汇编代码的一些关键部分，同学们自己阅读汇编码容易忽略一些代码细节问题。

6 各人实验贡献与体会（每人各自撰写）

1. 黄东威同学：回答了关于如何操控可编程计数器、如何实现系统调用、从用户态进程读取内核态数据的相关问题，编写了多级反馈队列调度算法等问题
个人体会：在本次实验中，我考虑到了 oranges 实验书上给的进程调度算法的不足之处，因此设计实现了全新的多级反馈队列调度算法用于更合理的调度进程。同时反复调试程序，修改代码框架，使得代码展示结果良好又防止出现进程饥饿等情况。我通过亲身实验，理解了上学期的 OS 课程所学习的调度算法是如何在代码层面实现的，熟悉了进程表、进程体等结构。实验过程中我多次遇到“PF”报错，即缺页错误，经过不断的打印字符查看结果，最终定位到了错误源头，增强了我对于该项目代码是如何运行的理解。这次实验很好的将 OS 的理论课和实验课结合起来，实现和创新的过程让我收获颇丰。

2. 周业营同学：回答了拓展中断支持多进程的步骤，在深度理解代码逻辑的基础上绘制了单进程拓展多进程、初始化多进程控制块、拓展初始化 LDT、修改时钟中断实支持多进程的流程图，帮助队友解决实验中遇到的问题，共同讨论多级反馈队列调度算法的实现。

个人体会：这次实验要做的是将单进程拓展为多进程，并且通过设计和实现系统调用的方式实现不同进程运行时间的差异化和进程间的调度。

在做实验的过程中对我影响最深的是在不断拓展功能实现的“屎山”代码的基础上将各个功能部分的代码隔离开来实现模块化。这是一个让整体代码逻辑变得更加清晰，并且能够实现各个模块代码复用的过程，这种做法是我之前一直想学习但一直很难有所进步的地方。在教材的带领下逐步实现这个代码瘦身的过程十分美妙，让我进一步加深了对代码模块化的理解。

在这次做 OS 实验的过程中，融合了微机原理中学的 8253 可编程计数器、OS 理论课的进程调度算法、保护模式下的中断处理和系统调用，正是在这种“大杂烩”的背景下才更锻炼我对不同知识的整合能力，也在实践中加深了我对各个部分的理解。

在实现多级反馈队列的过程中，一开始我和队友讨论的是通过设置 ticks 的下阈值来实现进程在不同队列上的调度，但在讨论的过程中发现这样做虽然节省了 PCB 部分的一点内存空间和简化了算法逻辑，但是无法让各个进程公平地分配同样数量的时间片。正是在这种实践的过程中不断精进我对进程调度算法的理解，在实践中考虑和实现调度算法的细节才是真正学习和设计出一个操作系统内核的精髓所在。

3. 王浚杰同学：独立完成了实验，并撰写了与多进程扩展和进程调度相关的思考题解答。此外，对多级反馈队列调度算法进行了优化和改进，实现了更高效的任务管理。

个人体会：本实验是在之前单进程基础上实现多进程调度。实验中，需要创建任务表 Tasktab，将所有进程的入口地址、堆栈大小和名称存储其中，并通过循环初始化每个进程控制块 proc_table。设置了进程表指针 p_proc_ready 以便在调度算法中管理多进程，形成了多进程管理的基本框架。

在多进程调度中，使用 ticks 和 priority 两个字段分别管理时间片和优先级，实现了对进程的合理调度。在调度算法上，成功实现了多级反馈队列调度，能够根据任务的优先级和时间片动态调整进程队列，确保高优先级的任务优先处理。

在系统调用方面，通过定义接口函数触发中断、设置中断门与处理函数、并使用函数指针数组管理系统调用，逐步掌握了系统调用的基础框架，巩固了我对中断机制的理解。

实验的完成让我认识到多进程操作系统中调度算法的重要性和实现的细腻之处。掌握了多级反馈队列调度后，我也对 CFS 等更高级调度算法有了兴趣，深感多进程调度是现代操作系统性能优化的关键。此次实验的实际编程操作和调试，为后续深入学习打下了坚实的基础。

4. 程序同学：独立完成实验，完成系统调用和进程调度部分的实验步骤书写，补充部分实验改进建议。

个人体会：本次实验是对操作系统理论课的补充，在理论课上，我们学习了时间片轮转、FCFS、优先级、多级反馈队列等多种进程调度算法，而本次实验中我们以一个优先级调度算法为案例，分析设计了多级反馈队列调度算法的实现，加深了对各种进程调度算法的理解。此外，此次实验是多进程的调度，涉及从单进程到多进程的拓展，进一步巩固了上一次实验的内容，并将创建一个进程改进为了一个相对自动化的过程。进程可以刻画系统的动态性，提高资源的利用率，对于程序的并发执行有着关键的作用，对于后续操作系统的进一步开发有着重要的意义。

7 教师评语

(实验报告的考评：依据实验内容完整度、实验步骤清晰度、实验结果与分析正确性、实验心得与思考的全面性、实验报告文档的规范性等五个维度综合考评)

分数	评语
85-100	<ul style="list-style-type: none">• 实验内容完整或者有超出课程实验大纲的内容；• 实验步骤详尽，能够体现完整的实验过程；• 实验结果正确且实验数据分析得当；• 实验心得与思考全面并且有自己的独立思考；• 实验报告文档规范、排版整齐。
75-84	<ul style="list-style-type: none">• 实验内容较为完整；• 实验步骤较为详尽，能够体现实验过程；• 实验结果正确且实验数据分析较为得当；• 实验心得与思考全面；• 实验报告文档规范、排版较为整齐。
60-74	<ul style="list-style-type: none">• 实验内容有缺失；• 实验步骤不够详尽，不能够体现完整的实验过程；• 实验结果部分正确；• 实验心得与思考无或者不够深入；• 实验报告文档规范性有待增强。
60 以下	<ul style="list-style-type: none">• 实验内容严重缺失、实验态度不够端正；• 实验步骤不够详尽，不能够体现完整的实验过程；• 实验结果部分正确；• 实验心得与思考无或者不够深入；• 实验报告文档规范性有待增强。

教师评分 (请填写好姓名、学号)

姓名	学号	分数
黄东威	2022302181148	
王浚杰	2022302181143	
程序	2022302181131	
周业营	2022302181145	

教师签名：

年 月 日