

# 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	2023.11.27
实验名称	进程（二）：多进程与进程调度	实验周次	第八周
姓名	学号	专业	班级
程子洋	2021301051114	网络空间安全	9
王卓	2021302191791	网络空间安全	9
聂森	2021302191536	网络空间安全	9
刘琥	2021302121234	网络空间安全	9

## 1 实验目的及实验内容

### 1.1 实验目的

1. 掌握多进程的实现机制
2. 学会 minix 的中断重入处理过程
3. 掌握系统调用、带优先级的进程调度实现方法

### 1.2 实验内容

1. 扩展单进程到多进程，扩展中断处理程序支持进程之间的切换
2. 优化中断重入的处理方法，响应并处理其他中断
3. 实现一个获取时钟 tick 的系统调用
4. 实现一个带优先级的进程调度
5. 在单进程的基础上如果高效扩展实现多进程？
6. 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
7. 如何修改时钟中断来支持多进程调度，画出新的流程图。
8. 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。
9. 在获取时钟 tick 时，如何操控 8253 可编程计数器？

10. 进程调度的框架是怎样的？优先级调度如何实现？
11. 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。
12. 思考题：从用户态进程读和写内核段的数据，看能否成功，是否会触发保护，并解释原因

## 2 实验环境及实验步骤

### 2.1 实验环境

- Ubuntu 16.04.1;
- VMWare Workstation 16 player;
- bochs 2.7。

### 2.2 实验步骤

1. 扩展单进程到多进程，扩展中断处理程序支持进程之间的切换
2. 优化中断重入的处理方法，响应并处理其他中断
3. 实现一个获取时钟 tick 的系统调用
4. 实现一个带优先级的进程调度

## 3 实验过程分析

### 3.1 扩展单进程到多进程，扩展中断处理程序支持进程之间的切换

#### 3.1.1 进程体

扩展单进程到多进程需要不止一个进程体，所以先实现两个不同的进程体。在 main.c 中进程 A 的代码的下面添加进程 B。

```
1  /*-----*
2                                     TestA
3  /*-----*/
4  void TestA()
5  {
6      int i = 0;
7      while(1){
8          disp_str("A");
9          disp_int(i++);
10         disp_str(".");
```

```

11     delay(1);
12 }
13 }
14
15 /*=====
16                                TestB
17 =====*/
18 void TestB()
19 {
20     int i = 0x1000;
21     while(1){
22         disp_str("B");
23         disp_int(i++);
24         disp_str(".");
25         delay(1);
26     }
27 }

```

在这里，除了打印的字母换成了 B 之外，i 的初始值被设成了 0x1000，为的就是在将来程序运行时能清晰地分辨两个进程。进程体其余的地方跟进程 A 完全一样。

### 3.1.2 相关的变量和宏

需要一个进程的运行需要 4 个变量宏：进程表，进程体，GDT，TSS。

与之前单进程的进程初始化步骤完全相同，但我们为了方便，构造一个循环来进行初始化步骤，使得以后每多一个进程只需要添加一次循环即可。

Minix 中定义了一个数组叫做 tasktab。这个数组的每一项定义好一个进程的开始地址、堆栈等，在初始化的时候，只要用一个 for 循环依次读取每一项，然后填充到相应的进程表项中就可以了。

在 proc.h 中声明一个数组类型，一个进程只要有一个进程体和堆栈就可以运行了，所以，这个数组只要有前两个成员其实就已经够了。这里，我们还定义了 name，以便给每个进程起一个名字。

```

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

```

下面为进程表宏定义：

```

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                                             {TestB, STACK_SIZE_TESTB, "TestB"}};

```

### 3.1.3 进程表与初始化

初始化进程的每一次循环的不同在于，从 TASK 结构中读取不同的任务入口地址、堆栈栈顶和进程名，然后赋给相应的进程表项。需要注意的地方有两点。

- 由于堆栈是从高地址往低地址生长的，所以在给每一个进程分配堆栈空间的时候也是从高地址往低地址进行。
- 在这里，我们为每一个进程都在 GDT 中分配一个描述符用来对应进程的 LDT。在 protect.h 中可以看到，SELECTOR\_LDT\_FIRST 是 GDT 中被定义的最后一个描述符，但是正如它的名字所表示的，它仅仅是“第一个”和“唯一一个”被明白指出来的而已。实际上，我们在 task\_table 中定义了几个任务，通过上文的 for 循环中的代码，GDT 中就会有几个描述符被初始化，它们列在 SELECTOR\_LDT\_FIRST 之后。

```

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
14        p_proc->ldt_sel = selector_ldt;
15
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        p_proc->regs.cs = ((8 * 0) & SA_RPL_MASK & SA_TI_MASK)
23            | SA_TIL | RPL_TASK;
24        p_proc->regs.ds = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
25            | SA_TIL | RPL_TASK;
26        p_proc->regs.es = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
27            | SA_TIL | RPL_TASK;
28        p_proc->regs.fs = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
29            | SA_TIL | RPL_TASK;
30        p_proc->regs.ss = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
31            | SA_TIL | RPL_TASK;
32        p_proc->regs.gs = (SELECTOR_KERNEL_CS & SA_RPL_MASK)
33            | RPL_TASK;
34
35        p_proc->regs.eip = (u32)p_task->initial_eip;
36        p_proc->regs.esp = (u32)p_task_stack;

```

```

37     p_proc->regs.eflags = 0x1202; /* IF=1, IOPL=1 */
38
39     p_task_stack -= p_task->stacksize;
40     p_proc++;
41     p_task++;
42     selector_ldt += 1 << 3;
43 }
44
45 k_reenter = -1;
46
47 p_proc_ready = proc_table;
48 restart();
49
50 while(1){
51 }

```

### 3.1.4 LDT 与初始化

每一个进程都会在 GDT 中对应一个 LDT 描述符。于是在 for 循环中，我们将每个进程表项中的成员 `p_proc->ldt_sel` 赋值。可是，选择子仅仅是解决了 where 问题，通过它，我们能在 GDT 中找到相应的描述符，但描述符的具体内容是什么，即 what 问题还没有解决。

当初只有一个进程时，我们是在 `init_prot()` 这个函数中通过一个语句解决了 what 的问题。现在，我们同样需要把它变成一个循环。

```

1  int i;
2  PROCESS* p_proc = proc_table;
3  u16 selector_ldt = INDEX_LDT_FIRST << 3;
4  for(i=0; i<NR_TASKS; i++){
5      init_descriptor(&gdt[selector_ldt >> 3],
6                     vir2phys(seg2phys(SELECTOR_KERNEL_DS),
7                               proc_table[i].ldts),
8                     LDT_SIZE * sizeof(DESCRIPTOR) - 1,
9                     DA_LDT);
10     p_proc++;
11     selector_ldt += 1 << 3;
12 }

```

另外，每个进程都有自己的 LDT，所以当进程切换时需要重新加载 `ldtr`。

### 3.1.5 中断处理程序

一个进程如何由“睡眠”状态变成“运行”状态？无非是将 `esp` 指向进程表项的开始处，然后在执行 `lldt` 之后经历一系列 `pop` 指令恢复各个寄存器的值。一切信息都包含在进程表中，所以，要想恢复不同的进程，只需要将 `esp` 指向不同的进程表就可以了。

但进程切换是一个稍稍有点复杂的过程，因为涉及进程调度等内容。

我们在时钟中断例程中时钟中断处理程序，完成字符的打印与进程切换。

```

1  hwint00:          ; Interrupt routine for irq 0 (the clock).
2      sub esp, 4
3      pushad        ; .
4      push    ds    ; |
5      push    es    ; | 保存原寄存器值
6      push    fs    ; |
7      push    gs    ; /
8      mov dx, ss
9      mov ds, dx
10     mov es, dx
11
12     inc byte [gs:0]    ; 改变屏幕第 0 行, 第 0 列的字符
13
14     mov al, EOI        ; . reenable
15     out INT_M_CTL, al    ; / master 8259
16
17     inc dword [k_reenter]
18     cmp dword [k_reenter], 0
19     jne .re_enter
20
21     mov esp, StackTop    ; 切到内核栈
22
23     sti
24     push    0
25     call    clock_handler
26     add esp, 4
27     cli
28
29     mov esp, [p_proc_ready] ; 离开内核栈
30     lldt    [esp + P_LDT_SEL]
31     lea eax, [esp + P_STACKTOP]
32     mov dword [tss + TSS3_S_SP0], eax
33
34 .re_enter:    ; 如果(k_reenter != 0), 会跳转到这里
35     dec dword [k_reenter]
36     pop gs    ; .
37     pop fs    ; |
38     pop es    ; | 恢复原寄存器值
39     pop ds    ; |
40     popad     ; /
41     add esp, 4
42     iretd

```

每一次我们让 p\_proc\_ready 指向进程表中的下一个表项, 如果切换前已经到达进程表结尾则回到第一个表项。

```

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

```

下面是多进程切换的结果展示：

```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
Booting .....
Ready.
Loading .....
Ready.

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0000F000h 00000000h 00000001h
0000F000h 00000000h 00001000h 00000000h 00000002h
0000E000h 00000000h 00001000h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size:01FF0000h
-----"cstart" begins-----
-----"cstart" finished-----
-----"kernel_main" begins-----
A0x0. #B0x1000. #A0x1. #B0x1001. ##B0x1002. #A0x2. ##A0x3. #B0x1003. #A0x4. #B0x1004. ##B0
x1005. #A0x5. ##A0x6. #B0x1006. #A0x7. #B0x1007. ##B0x1008. #A0x8. ##A0x9. #B0x1009. #A0xA
. #B0x100A. ##A0xB. #B0x100B. #A0xC. #B0x100C. ##B0x100D. #A0xD. ##A0xE. #B0x100E. #A0xF.
#B0x100F. ##B0x1010. #A0x10. ##A0x11. #B0x1011. #A0x12. #B0x1012. ##B0x1013. #A0x13. ##A0
x14. #B0x1014. ##B0x1015. #A0x15. ##A0x16. #B0x1016. #A0x17. #B0x1017. ##B0x1018. #A0x18.
##A0x19. #B0x1019. #A0x1A. #B0x101A. ##B0x101B. #A0x1B. ##A0x1C. #B0x101C. #A0x1D. #B0x10
1D. ##A0x1E. #B0x101E. #A0x1F. #B0x101F. ##B0x1020. #A0x20. ##A0x21. #B0x1021. #A0x22. #B
CTRL + 3rd button enables mouse  A2 NUM CAPS SCRL
```

图 1: 多进程切换结果

## 3.2 优化中断重入的处理方法，响应并处理其他中断

### 3.2.1 Minix 的中断处理

相同于时钟中断程序，其他的中断程序也会在开始处保存当前进程的信息，在结束处恢复一个进程，中间也会遇到中断重入、内核栈的问题。首先看看 Minix 操作系统是怎么处理中断的。我们后续来模仿它的中断处理方式。

```
#define hwint_master(irq) \
call    save                /* save interrupted process state */;\
inb     INT_CTLMASK         ;\
orb     al, [1<<irq]        ;\
outb    INT_CTLMASK         /* disable the irq */;\
movb    al, ENABLE          ;\
outb    INT_CTL             /* reenale master 8259 */;\
sti     /* enable interrupts */;\
push    irq                 /* irq */;\
call    (_irq_table + 4*irq) /* eax = (*irq_table[irq])(irq) */;\
pop     ecx                 ;\
cli     /* disable interrupts */;\
test    eax, eax            /* need to reenale irq? */;\
jz      of                  ;\
inb     INT_CTLMASK         ;\
andb    al, ~[1<<irq]       ;\
outb    INT_CTLMASK         /* enable the irq */;\
0:      ret                 /* restart (another) process */;\

! Each of these entry points is an expansion of the hwint_master macro
.align 16
_hwint00:      ! Interrupt routine for irq 0 (the clock).
               hwint_master(0)
```

图 2: Minix 的中断处理

代码后面的 `_hwint00` 等就是中断程序的入口。`hwint_master` 首先调用一个函数 `save`，将寄存器的值保存起来，然后操纵 8259A 避免在处理当前中断的同时发生同样类型的中断。紧接着，给 8259A 的中断命令寄存器发出中断结束命令（EOI）。然后用 `sti`



指令打开中断，调用函数 (\*irq\_table[irq]) (irq)，这是与当前中断相关的一个例程。再用 cli 关中断、test 指令判断函数 (\*irq\_table[irq]) (irq) 的返回值。如果非零的话就重新打开当前发生的中断；如果是零的话就直接 ret。

对于 save 函数，具体代码如下：

```

save:
    cld                                ! set direction flag to a known value
    pushad                             ! save "general" registers
    o16 push    ds                      ! save ds
    o16 push    es                      ! save es
    o16 push    fs                      ! save fs
    o16 push    gs                      ! save gs
    mov     dx, ss                      ! ss is kernel data segment
    mov     ds, dx                      ! load rest of kernel segments
    mov     es, dx                      ! kernel does not use fs, gs
    mov     eax, esp                    ! prepare to return
    incb    (_k_reenter)                ! from -1 if not reentering
    jnz     set_restart1                ! stack is already kernel stack
    mov     esp, k_stktop
    push    _restart                    ! build return address for int handler
    xor     ebp, ebp                    ! for stacktrace
    jmp     RETADR-P_STACKBASE(eax)

    .align 4
set_restart1:
    push    restart1
    jmp     RETADR-P_STACKBASE(eax)

```

图 3: Minix 的中断处理

可以看到，当发生中断重入，就跳过切换内核栈的代码（因为已经在内核栈了），并且把不同的地址压栈。此外，中断重入与否除了切换内核栈以外，可以发现：push 语句也不相同。假设非中断重入，将会执行 push \_restart 一句；假设发生中断重入，就会执行 push restart1 一句，不再进行进程的切换。

### 3.2.2 代码修改

- 从中断例程中分离出 restart

这里，我们删除了 hwint00 中的最后一段代码，并修改了涉及删除代码中标号的两句指令，同时修改用到标号.restart\_v2 和.restart\_reenter\_v2 的地方。

```

1  ;=====
2  ;                                     restart
3  ;=====
4  restart:
5      mov esp, [p_proc_ready]
6      lldt    [esp + P_LDT_SEL]
7      lea eax, [esp + P_STACKTOP]
8      mov dword [tss + TSS3_S_SP0], eax
9  restart_reenter:
10     dec dword [k_reenter]
11     pop gs
12     pop fs

```



```

13     pop es
14     pop ds
15     popad
16     add esp, 4
17     iretd

```

- 从中断例程中分离 save

这里将 Minix 中有关的 save 函数的代码挪进我们的 save 函数中

```

1  ;=====
2  ;                                     save
3  ;=====
4  save:
5      pushad                ; .
6      push    ds            ; |
7      push    es            ; | 保存原寄存器值
8      push    fs            ; |
9      push    gs            ; /
10     mov     dx, ss
11     mov     ds, dx
12     mov     es, dx
13
14     mov     eax, esp        ; eax = 进程表起始地址
15
16     inc     dword [k_reenter] ; k_reenter++;
17     cmp     dword [k_reenter], 0 ; if(k_reenter == 0)
18     jne     .1              ; {
19     mov     esp, StackTop    ; mov esp, StackTop <—切换到内核栈
20     push    restart          ; push restart
21     jmp     [eax + RETADR - P_STACKBASE] ; return;
22 .1:                          ; } else { 已经在内核栈, 不需要再切换
23     push    restart_reenter ; push restart_reenter
24     jmp     [eax + RETADR - P_STACKBASE] ; return;
25                          ; }

```

save 函数准备好之后, 我们修改中断例程:

```

1  hwint00:                ; Interrupt routine for irq 0 (the clock).
2      call     save
3
4      in  al, INT_M_CTLMASK ; .
5      or  al, 1             ; | 不允许再发生时钟中断
6      out INT_M_CTLMASK, al ; /
7
8      mov al, EOI           ; . reenale
9      out INT_M_CTL, al     ; / master 8259
10
11     sti
12     push    0
13     call    clock_handler
14     add esp, 4
15     cli
16
17     in  al, INT_M_CTLMASK ; .

```

```

18     and al, 0xFE             ; | 又允许时钟中断发生
19     out INT_M_CTLMASK, al    ; /
20
21     ret

```

在这里添加了两端代码，在调用 `clock_handler` 之前屏蔽掉时钟中断，在调用之后重新打开。这样，在只打开时钟中断的时候不再会发生中断重入的情况。

至此，我们的时钟中断处理程序与 Minix 的 `hwint_master` 基本一致，我们将其修改为一个类似的宏，用来替换原有的宏。

```

1  extern  irq_table
2  ...
3          ; 中断和异常 —— 硬件中断
4  ; -----
5  %macro  hwint_master      1
6      call    save
7      in  al, INT_M_CTLMASK    ; .
8      or  al, (1 << %1)       ; | 屏蔽当前中断
9      out INT_M_CTLMASK, al    ; /
10     mov al, EOI              ; . 置EOI位
11     out INT_M_CTL, al        ; /
12     sti ; CPU在响应中断的过程中会自动关中断，这句之后就允许响应新的中断
13     push    %1              ; .
14     call    [irq_table + 4 * %1] ; | 中断处理程序
15     pop ecx                  ; /
16     cli
17     in  al, INT_M_CTLMASK    ; .
18     and al, ~(1 << %1)      ; | 恢复接受当前中断
19     out INT_M_CTLMASK, al    ; /
20     ret
21 %endmacro
22
23 ALIGN    16
24 hwint00:    ; Interrupt routine for irq 0 (the clock).
25     hwint_master    0

```

到此为止，代码的修改就结束了。编译并运行，结果如23所示。

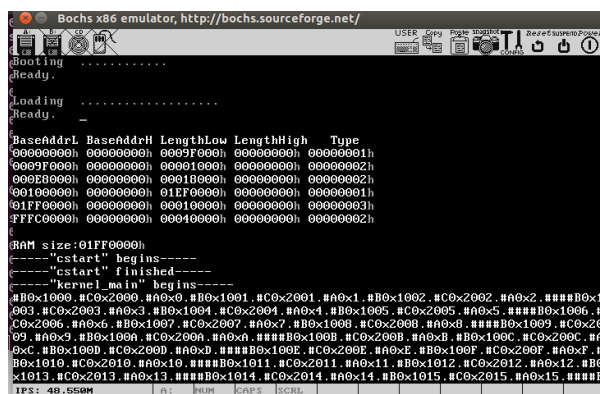


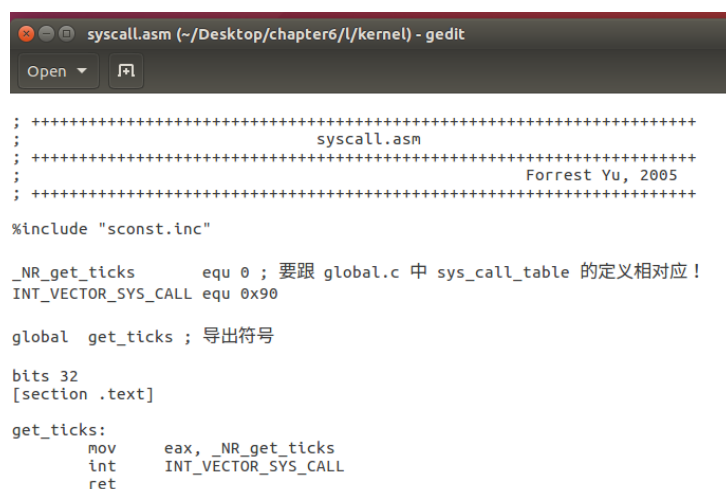
图 4: 编译运行结果

结果和原先大致相同，但现在的代码更有条理，而且更容易扩展。实际上，这里对于原始代码的修改不仅仅是一个时钟中断处理程序，同时也是一套方便扩展的中断处理的接口。现在，如果想添加某个中断处理模块。只需要将中断处理的函数入口地址赋给 `irq_table` 中的相应元素即可。

### 3.3 实现一个获取时钟 tick 的系统调用

#### 3.3.1 实现基本过程

`get_ticks()` 的代码如下，这里将 `eax` 的值赋为 `_NR_get_ticks`，并且将系统调用的对应的终端号设置为 `0x90`：



```
; ++++++ syscall.asm ++++++
;                                     syscall.asm
;                                     Forrest Yu, 2005
; ++++++
%include "sconst.inc"

_NR_get_ticks      equ 0 ; 要跟 global.c 中 sys_call_table 的定义相对应！
INT_VECTOR_SYS_CALL equ 0x90

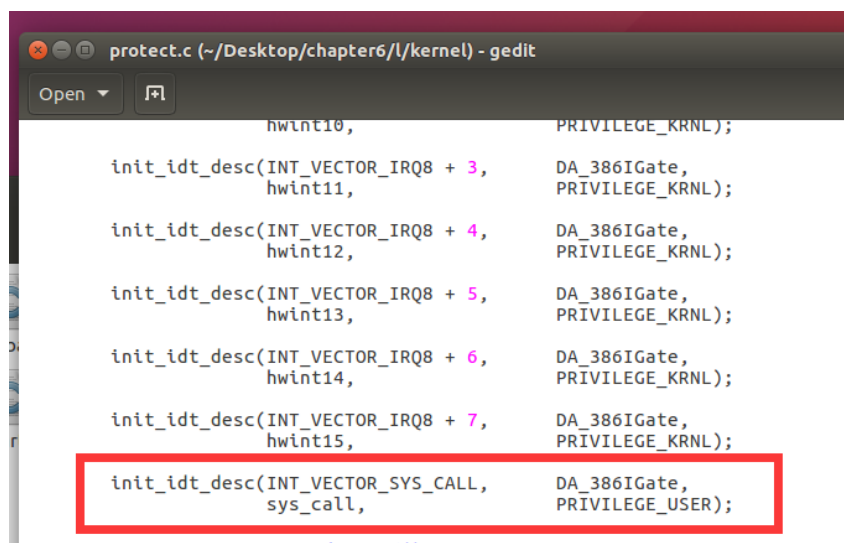
global get_ticks ; 导出符号

bits 32
[section .text]

get_ticks:
    mov     eax, _NR_get_ticks
    int     INT_VECTOR_SYS_CALL
    ret
```

图 5: `get_ticks()` 代码

接下来初始化系统调用的中断门：



```
hwint10, PRIVILEGE_KRNL);

init_idt_desc(INT_VECTOR_IRQ8 + 3, DA_386IGate,
hwint11, PRIVILEGE_KRNL);

init_idt_desc(INT_VECTOR_IRQ8 + 4, DA_386IGate,
hwint12, PRIVILEGE_KRNL);

init_idt_desc(INT_VECTOR_IRQ8 + 5, DA_386IGate,
hwint13, PRIVILEGE_KRNL);

init_idt_desc(INT_VECTOR_IRQ8 + 6, DA_386IGate,
hwint14, PRIVILEGE_KRNL);

init_idt_desc(INT_VECTOR_IRQ8 + 7, DA_386IGate,
hwint15, PRIVILEGE_KRNL);

init_idt_desc(INT_VECTOR_SYS_CALL, DA_386IGate,
sys_call, PRIVILEGE_USER);
```

图 6: 初始化系统调用的中断门

由于中使用了 `eax` 寄存器, 因此需要修改 `save` 的代码, 将其中的 `eax` 全部改成 `esi`:

```

; =====
;                                     save
; =====
save:
    pushad                ; \
    push    ds            ; |
    push    es            ; | 保存原寄存器值
    push    fs            ; |
    push    gs            ; /
    mov     dx, ss
    mov     ds, dx
    mov     es, dx

    mov     esi, esp      ; esi = 进程表起始地址

    inc     dword [k_reenter] ; k_reenter++;
    cmp     dword [k_reenter], 0 ; if(k_reenter == 0)
    jne     .1            ; {
    mov     esp, StackTop ; mov esp, StackTop <-- 切换到内核栈
    push    restart       ; push restart
    jmp     [esi + RETADR - P_STACKBASE]; return;

.1:
    push    restart_reenter ; push restart_reenter
    jmp     [esi + RETADR - P_STACKBASE]; return;
; }

```

图 7: 修改 `save`

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 `nwint_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:

```

; =====
;                                     sys_call
; =====
sys_call:
    call    save

    sti

    call    [sys_call_table + eax * 4]
    mov     [esi + EAXREG - P_STACKBASE], eax

    cli

    ret

```

图 8: 函数 `sys_call`

为了简单起见, 这里打印一个字符 “+” 后就返回, 而不做其他任何操作, 代码如下:

```

proc.c (~/Desktop/chapter6/kernel) - gedit

/*=====
proc.c
=====
Forrest Yu, 2005
=====*/

#include "type.h"
#include "const.h"
#include "protect.h"
#include "proto.h"
#include "string.h"
#include "proc.h"
#include "global.h"

/*=====
sys_get_ticks
=====*/
PUBLIC int sys_get_ticks()
{
    disp_str("+");
    return 0;
}

```

图 9: 函数 `sys_call`

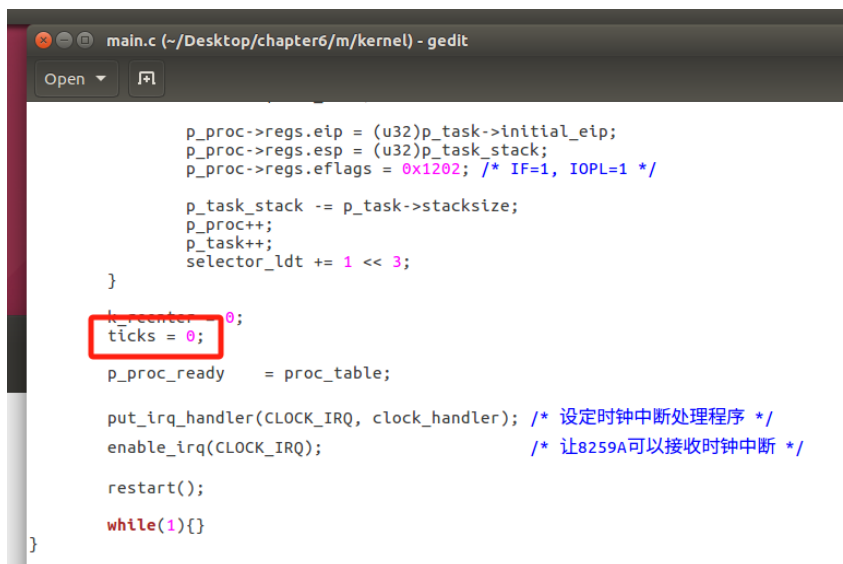
之后在 proto.h 中添加函数声明，最后在 TestA 中测试系统调用，代码如下：

```
/*=====
TestA
=====*/
void TestA()
{
    int i = 0;
    while (1) {
        get_ticks();
        disp_str("A");
        disp_int(i++);
        disp_str(".");
        delay(1);
    }
}
```

图 10: TestA 代码

### 3.3.2 改进

在 main.c 中初始化 ticks:



```
main.c (~/Desktop/chapter6/m/kernel) - gedit
Open

p_proc->regs.eip = (u32)p_task->initial_eip;
p_proc->regs.esp = (u32)p_task_stack;
p_proc->regs.eflags = 0x1202; /* IF=1, IOPL=1 */

p_task_stack -= p_task->stacksize;
p_proc++;
p_task++;
selector_ldt += 1 << 3;
}
k_reenter = 0;
ticks = 0;
p_proc_ready = proc_table;

put_irq_handler(CLOCK_IRQ, clock_handler); /* 设定时钟中断处理程序 */
enable_irq(CLOCK_IRQ);                    /* 让8259A可以接收时钟中断 */

restart();
while(1){}
```

图 11: 初始化 ticks

在 clock\_handler(int irq) 中添加一句话:

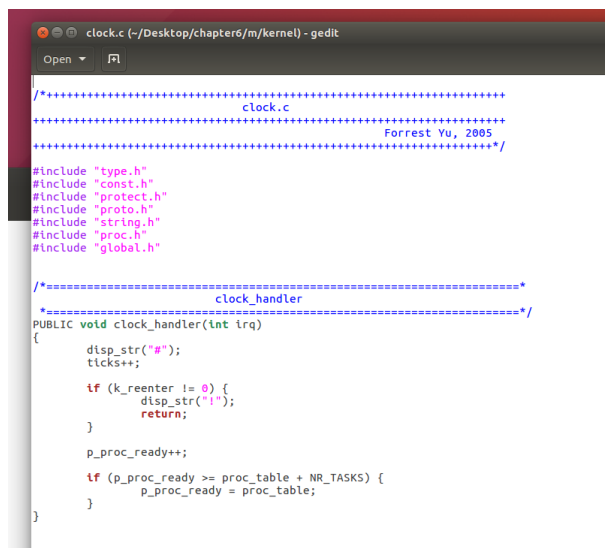


图 12: 时钟中断处理程序

然后修改 sys\_get\_ticks (), 最后修改 TestA, 不再递增的打印 i, 而是改成打印当前的 ticks, 代码如下所示:

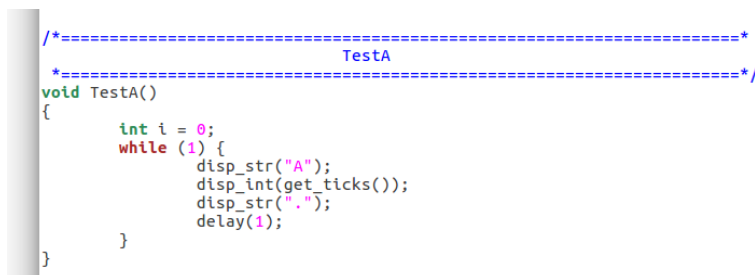


图 13: 修改 TestA

验证系统调用:

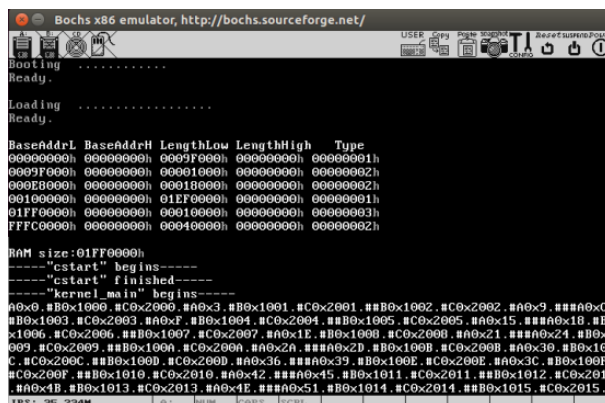


图 14: 系统调用的运行结果

第一次打印出的是 A0x0, 第二次打印出的是 A0x9, 且发现两次打印之间的“#”恰好是 9 个, 说明 get ticks 一切正常!

### 3.4 实现一个带优先级的进程调度

#### 1. 修改 clock\_handler

```

/*=====
                                clock_handler
=====*/
PUBLIC void clock_handler(int irq)
{
    ticks++;
    p_proc_ready->ticks--;

    if (k_reenter != 0) {
        return;
    }

    if (p_proc_ready->ticks > 0) {
        return;
    }

    schedule();
}

```

图 15: clock-handler

#### 2. 修改优先级:

```

proc_table[0].ticks = proc_table[0].priority = 15;
proc_table[1].ticks = proc_table[1].priority = 5;
proc_table[2].ticks = proc_table[2].priority = 3;

```

图 16: 优先级

#### 3. 把各个进程的延迟时间改为 10ms:

```

/*=====
                                TestA
=====*/
void TestA()
{
    int i = 0;
    while (1) {
        disp_str("A.");
        mlll_delay(10);
    }
}

/*=====
                                TestB
=====*/
void TestB()
{
    int i = 0x1000;
    while(1){
        disp_str("B.");
        mlll_delay(10);
    }
}

/*=====
                                TestB
=====*/
void TestC()
{
    int i = 0x2000;
    while(1){
        disp_str("C.");
        mlll_delay(10);
    }
}

```

图 17: 延迟时间



## 4. 运行结果：

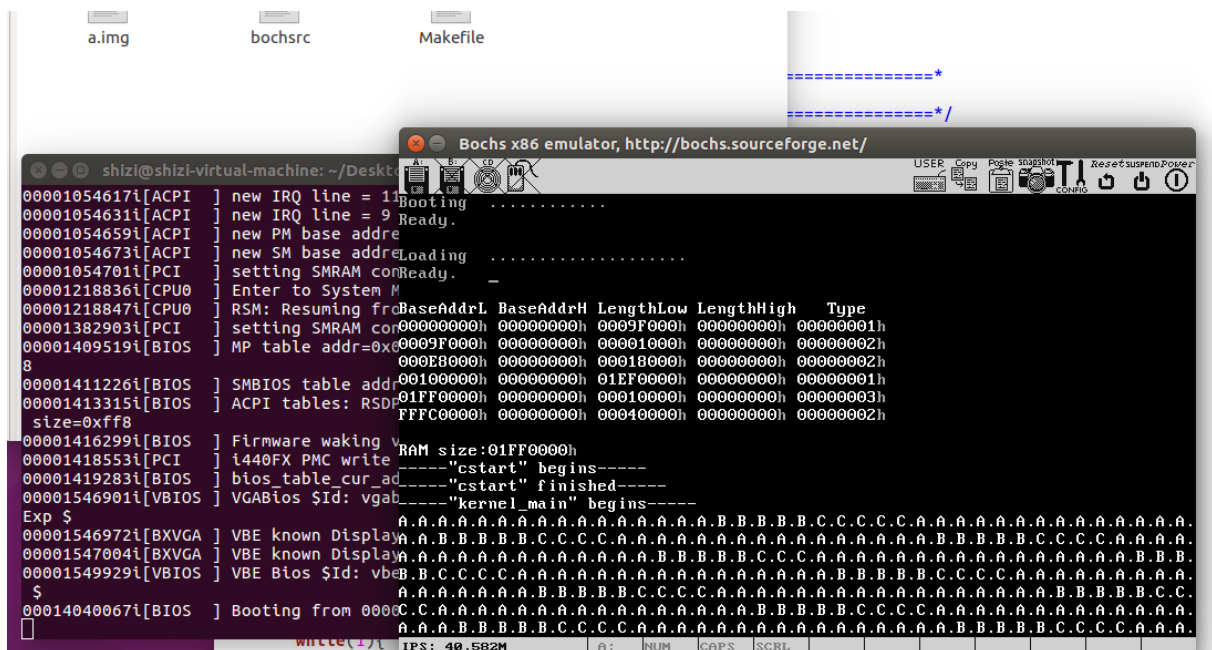


图 18: 结果

可以看出打印出的结果个数非常接近 15: 5: 3

## 4 实验结果总结

### 4.1 思考题汇总

#### 4.1.1 在单进程的基础上如果高效扩展实现多进程？

详见 3.1 内容。简单来说，我们只需要让其中一个进程处在运行态，其余进程处在睡眠态就可以了。进程不外乎 4 个要素：进程表、进程体、GDT、TSS。Minix 中定义了一个数组叫做 tasktab。这个数组的每一项定义好一个进程的开始地址、堆栈等，在初始化的时候，只要用一个 for 循环依次读取每一项，然后填充到相应的进程表项中就可以了。让增加一个进程变得简单而迅速。

#### 4.1.2 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。

如下图所示：

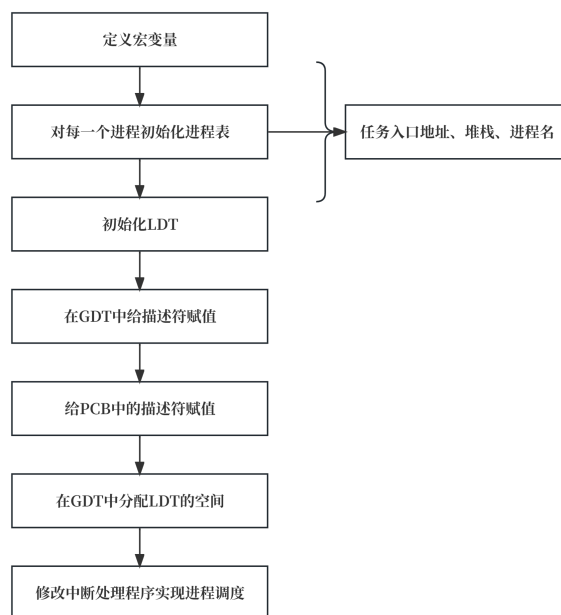


图 19: 思考题 2 流程图

#### 4.1.3 如何修改时钟中断来支持多进程调度，画出新的流程图。

流程图如下图22所示：

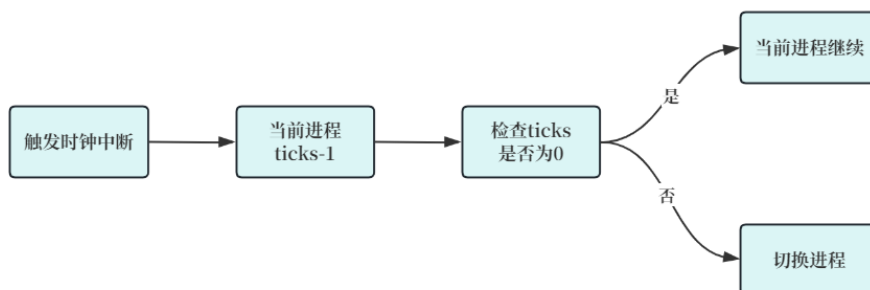


图 20: 修改时钟中断支持多进程调度流程图

## 4.1.4 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。

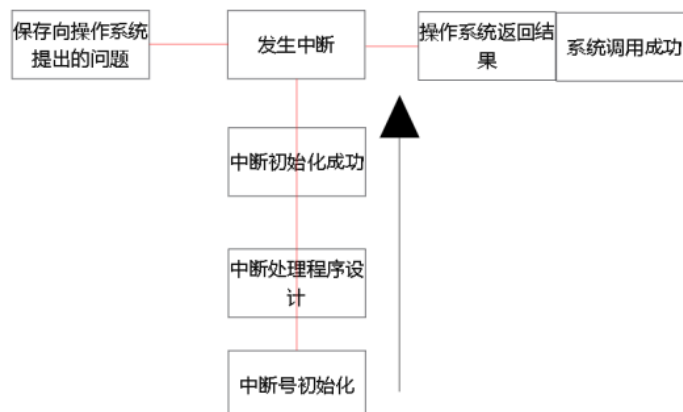


图 21: 系统调用流程图

## 4.1.5 在获取时钟 tick 时，如何操控 8253 可编程计数器？

根据不同硬件决定的两次时钟中断的间隔，来决定经过多少次时钟中断后，计数器计数。

```

PUBLIC void milli_delay(int milli_sec)
{
    int t = get_ticks();
    while(((get_ticks() - t) * 1000 / HZ) < milli_sec) {}
}
  
```

图 22: 代码说明

## 4.1.6 进程调度的框架是怎样的？优先级调度如何实现？

进程调度的框架如下图23所示：

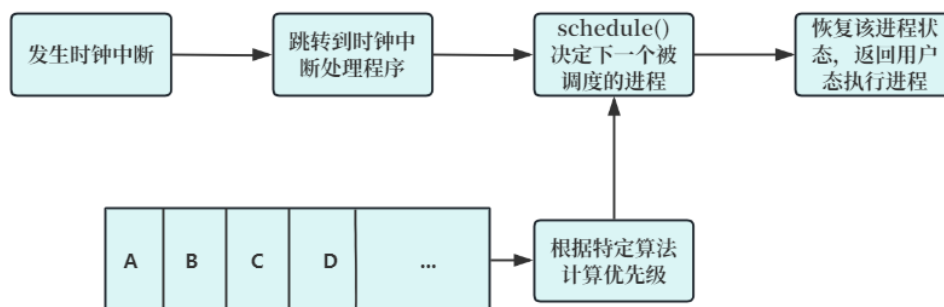


图 23: 进程调度框架

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。

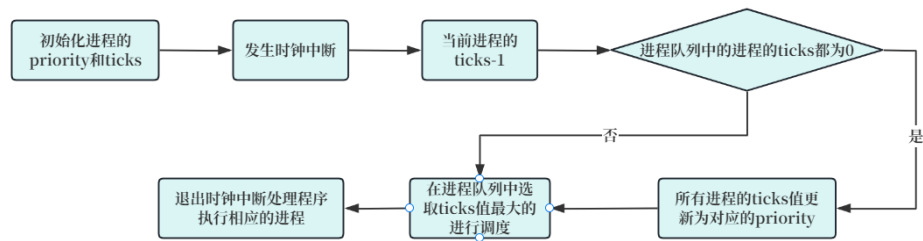


图 24: 优先级调度

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当中，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。

#### 4.1.7 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。

1. 头文件中添加相应数据结构，主要有三个队列和对应的时间片，队列用数组来实现，存储 u32 类型的 pid，优先级越高的队列对应时间片越少。

```

#define Q1_TICK 1
#define Q2_TICK 3
#define Q3_TICK 5
#define NR_PROC 32
EXTERN u32 Q1[NR_PROC];
EXTERN u32 Q1_len;
EXTERN u32 Q2[NR_PROC];
EXTERN u32 Q2_len;
EXTERN u32 Q3[NR_PROC];
EXTERN u32 Q3_len;

```

图 25: 修改 global.h

2. 修改 PROCESS 结构体：

```

typedef struct s_proc {
    STACK_FRAME regs; /* process registers saved in stack frame */

    u16 ldt_sel; /* gdt selector giving ldt base and limit */
    DESCRIPTOR lds[LDT_SIZE]; /* local descriptors for code and data */

    int ticks; /* remained ticks */
    int priority;
    int queueticks;

    u32 pid; /* process id passed in from MM */
    char p_name[16]; /* name of the process */
}PROCESS;

```

图 26: 修改 global.h

3. 编写 schedule 函数，该函数是实现根据当前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。

```

PUBLIC void schedule()
{
    if(Q1_len != 0){ // Q1队列非空
        int proc_id = Q1[0];
        PROCESS* p = &proc_table[proc_id]; // 取出队首元素
        p_proc_ready = p;

        disp_str("\n");
        disp_str(p_proc_ready->p_name);
        disp_str("[proc-ticks=");
        disp_int(p->ticks);
        disp_str(" & Q1-ticks=");
        disp_int(p->queueticks);
        disp_str("]");

        p->queueticks--;

        if(p->ticks<=0){ //该进程结束了
            Q1_len--;
            for(int i=0;i<Q1_len;i++)
                Q1[i] = Q1[i+1]; // 此处实现的是将队首元素出队列
        }else if(p->queueticks<=0){ // 该进程在Q1队列之中的时间已经用完
            Q1_len--;
            for(int i=0;i<Q1_len;i++)
                Q1[i] = Q1[i+1]; // 此处实现的是将队首元素出队列
            p->queueticks=Q2_TICK;
            Q2[Q2_len]=proc_id;
            Q2_len++;
        }else if(Q2_len != 0){
            int proc_id = Q2[0];
            PROCESS* p = &proc_table[proc_id];
            p_proc_ready = p;

            disp_str("\n");
            disp_str(p_proc_ready->p_name);
            disp_str("[p-ticks=");
            disp_int(p->ticks);
            disp_str(" & Q2-ticks=");
            disp_int(p->queueticks);
            disp_str("]");

            p->queueticks--;

            if(p->ticks<=0){
                Q2_len--;
                for(int i=0;i<Q2_len;i++)
                    Q2[i] = Q2[i+1];
            }else if(p->queueticks<=0){
                Q2_len--;
                for(int i=0;i<Q2_len;i++)
                    Q2[i] = Q2[i+1];
                p->queueticks=Q3_TICK;
                Q3[Q3_len]=proc_id;
                Q3_len++;
            }else if(Q3_len != 0){
                int proc_id = Q3[0];
                PROCESS* p = &proc_table[proc_id];
                p_proc_ready = p;

                disp_str("\n");
                disp_str(p_proc_ready->p_name);
                disp_str("[p-ticks=");
                disp_int(p->ticks);
                disp_str(" & Q3-ticks=");
                disp_int(p->queueticks);
                disp_str("]");

                p->queueticks--;
                if(p->ticks<=0){
                    Q3_len--;

```

图 27: 修改 proc.h

4. 在 clock.c 之中实现，即分别在 t=0,5,6,8,11 时分别来 A E 五个进程，其所需要执行时间分别为 15,1,2,3,3

```

/*=====
                                clock_handler
=====*/
PUBLIC void clock_handler(int irq)
{
    if (ticks == 0) {
        Q1[Q1_len++] = 0; //A
        proc_table[Q1[Q1_len - 1]].queueticks = Q1_TICK;
    }
    if (ticks == 5) {
        Q1[Q1_len++] = 1; //B
        proc_table[Q1[Q1_len - 1]].queueticks = Q1_TICK;
    }
    if (ticks == 6) {
        Q1[Q1_len++] = 2; //C
        proc_table[Q1[Q1_len - 1]].queueticks = Q1_TICK;
    }
    if (ticks == 8) {
        Q1[Q1_len++] = 3; //D
        proc_table[Q1[Q1_len - 1]].queueticks = Q1_TICK;
    }
    if (ticks == 11) {
        Q1[Q1_len++] = 4; //E
        proc_table[Q1[Q1_len - 1]].queueticks = Q1_TICK;
    }

    ticks++;
    if (ticks > 1) {
        p_proc_ready->ticks--;
    }
    schedule();
}

```

图 28: 修改 clock.c

进程所需的运行时间的定义在 main.c 之中实现：

```

proc_table[0].ticks = proc_table[0].priority = 15;
proc_table[1].ticks = proc_table[1].priority = 1;
proc_table[2].ticks = proc_table[2].priority = 2;
proc_table[3].ticks = proc_table[3].priority = 3;
proc_table[4].ticks = proc_table[4].priority = 3;

```

图 29: 修改 main.c

## 5. 结果展示：

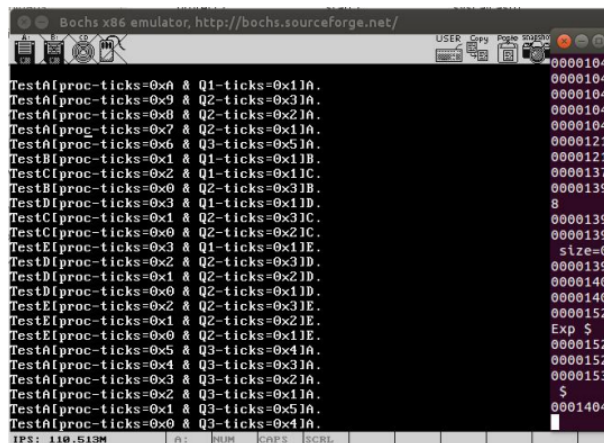
The image shows a screenshot of a Bochs x86 emulator window. The title bar reads "Bochs x86 emulator, http://bochs.sourceforge.net/". The window is divided into several panes. The main pane on the left displays assembly code with comments, such as "TestAProc-ticks=0xA", "Q1-ticks=0x11A", "TestBProc-ticks=0x1", "Q1-ticks=0x11B", etc. The right pane shows memory addresses and their corresponding values in hexadecimal, such as "00001040", "00001041", "00001042", etc. The bottom status bar shows "IPS: 110.513M" and other emulator statistics.

图 30: 运行结果

#### 4.1.8 思考题：从用户态进程读和写内核段的数据，看能否成功，是否会触发保护，并解释原因

会触发保护，

原因：操作系统将内核空间 and 用户空间分开来保护系统的安全性和稳定性。用户态进程只能访问属于用户空间的地址空间，而内核态进程才能访问属于内核空间的地址空间。内核段是属于内核空间的一部分，包含操作系统的代码、数据结构以及关键资源。

当用户态进程试图访问内核段时，通常会触发异常或错误，这是由硬件和操作系统的保护机制实施的。这些保护机制可以检测到非法的内存访问，例如访问未授权的内核段，从而防止用户态进程对内核造成潜在的损害或安全漏洞。

因此，为了确保系统的安全性和稳定性，用户态进程应该通过提供的系统调用接口来与内核进行通信，而不是直接读取和写入内核段的数据。系统调用提供了一种受控的接口，允许用户态进程向内核发起请求，并由内核代表用户完成对内核数据的访问和操作。这样的设计可以确保内核的完整性和安全性。

## 4.2 实验问题总结

## 4.3 实验改进意见

王卓：

1. 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. 介绍实验目的和背景：实验开始之前提供实验的目的和背景，解释一下为什么需要进行该实验以及其与操作系统的关系，这能帮助我们更好地理解实验的意义和重要性。



程子洋：

1. 错误排除指南：提供学生在遇到常见问题时进行自我排除的指南。这可以包括常见错误消息的解释以及如何解决这些问题的步骤。

聂森：

1. 可以先让同学阅读教材资料后再介绍实验内容和实验相关的知识，可以帮助同学更快更好的了解实验的目标和实验的原理及任务。
2. 给出具体的实验指导或给出更加丰富的实验参考资料，可以帮助同学更快上手实验并在有问题时找到解决方式。

刘琥：

1. 如果能有详细的操作教学可以帮助学生更快地上手，并且希望老师能讲解汇编代码的一些关键部分，学生自己阅读汇编代码容易忽略一些问题。

## 5 各人实验贡献与体会

王卓：

此次实验为本人独立完成全部实验内容，并主要负责实验内容的第二题、思考题的第三题和第六题的实验报告的撰写。

这次实验我有很大的收货，总的来说，本次的实验比较复杂，很许多的重难点和需要理解的新知识。在本章需要学习的知识中，也有许多值得慢慢分析和理解的地方，首先有很多新的概念要去熟悉；同时，阅读汇编代码、熟悉语句和段落的功能也是比较困难的任务。

通过本次实验，我深入分析、理解并且掌握了以下内容：单进程到多进程的扩展方法、系统调用的实现、进程调度的基本实现……对其中各种相关知识都有了一定程度的理解和自我的掌握；同时，对汇编代码的阅读和分析过程，也对我自己汇编的语法和代码编写的知识和能力带来了极大的锻炼和提升。

本次实验中最大的困难点在于动手改部分：基于示例代码，模拟实现一个多级反馈队列调度算法。虽然需要实现的功能是基于现有代码进行相关改动，可是我对汇编不是十分熟练、对各种繁杂的代码难以快速掌握和吸收、对本章知识不是完全理解等原因，导致编写和调试还是花了很长时间，不过最终还是得到了正确的结果，感觉自己收获颇丰，极大加深了我对内核和汇编编写的理解！

程子洋：

此次实验为本人独立完成全部实验内容，并主要负责实验内容的第一题、思考题的第一题实验报告的撰写。

实验让我深入理解了操作系统内核的关键结构，如进程控制块、GDT/LDT、TSS 等，以及它们之间的关系。这种理解不仅提升了我对操作系统内部运作的认识，也为理解多任务、内存管理等方面的概念奠定了基础。

实验中涉及到的关键技术，如初始化进程控制块、GDT 和 TSS，实现进程的启动、现场保护与恢复等，使我对操作系统内核设计的关键技术有了更加深入的了解。这些技术对于实现一个稳定、高效的操作系统至关重要。从 Ring0 切换到 Ring1 的实现，让我明白了保持操作系统内核的安全性的必要性。通过合理切换特权级，可以限制用户态程序的权限，提高系统的安全性。

对其各种相关知识都有了一定程度的理解和自我的掌握；同时，对汇编代码的阅读和分析过程，也对我自己汇编的语法和代码编写的知识和能力带来的极大的锻炼和提升。

**聂森：**

此次实验为本人独立完成全部实验内容，并主要负责实验内容的第三题、思考题的第四、五题的实验报告撰写。

本次实验使我深刻认识到系统调用是操作系统设计中不可或缺的组成部分。通过实现获取时钟 tick 的系统调用，我不仅加深了对系统调用基本框架的理解，也学会了如何与硬件组件交互，特别是控制 8253 可编程计数器的技巧。

实验中，我首先掌握了系统调用的流程和结构，包括用户空间向内核空间的转换，系统调用接口的设计，以及如何通过服务例程来处理具体的系统调用请求。在操作 8253 计数器时，我学习了如何设置计数器以产生周期性的时钟中断，这对于理解操作系统中的时间管理和进程调度机制非常有帮助。通过直接操控硬件，我更加深入地理解了内核如何管理和控制计算机的物理资源。同时，这也要求我具备对硬件操作的细致理解，以及对汇编语言的熟练运用。实际编写控制 8253 计数器的代码，让我对汇编语言有了更深刻的认识，也锻炼了我的编程技能。

这次实验不仅让我获得了关于系统调用和硬件控制的实际经验，也促进了我对操作系统底层工作原理的理解。通过亲手实现系统调用，我感受到了从理论到实践的转变，并且在遇到困难和解决问题的过程中，我的问题解决能力和调试编程能力都得到了提高。

**刘琥：**

此次实验为本人独立完成全部实验内容，并主要负责实验内容的第四题，动手做和思考题实验报告撰写。

通过对实现多级反馈队列调度算法的实现，我对进程的有了更进一步的理解，对系统调用的实现、进程调度等方面的知识有了更好地掌握，理解了上学期的 OS 课程中所学习的进程调度是如何在代码层面具体实现的，熟悉了进程表、进程体等结构，然后学会修改进程表中 PROCESS 结构体的内容，以及在 schedule 函数之中实现进程调度算法。这次实验有大量的示例代码需要理解，在初接触时确实造成了不小的麻烦，但是通过静下心来细读，能够对本章知识极为迅速地掌握，也让之前学的知识也不再只是停留

在纸面上，有感受到通过真正的编程和实践把知识融会贯通。

教师评语		
姓名	学号	分数
程子洋	2021301051114	
聂森	2021302191536	
王卓	2021302191791	
刘琥	2021302121234	
教师签名： <div style="text-align: right; margin-top: 20px;">                         年 月 日                     </div>		