

第四次实验说明文档

141210026 宋奎熹

*注：代码均运行在 Ubuntu 16.04 32位系统上

本次实验是在 Orange's 第六章 r 文件夹中代码的基础上修改的。

主要修改之处如下：

include/const.h

为解决理发师问题，增加全局变量 CHAIRS 指示可供顾客等待的椅子数，QUEUE_LENGTH 指示信号量结构体中可等待的最大进程数量。

```
69  /*为顾客准备的椅子数*/  
70  /**  
71   *   Modified Here  
72   */  
73  #define CHAIRS          1  
74  #define QUEUE_LENGTH    10  
75
```

include/global.h

理发师问题中的变量，有 waiting：等候理发的顾客人数，number：当前顾客编号，和顾客、理发师、互斥三个信号量。同时为了事先按行清屏，设置记录当前输入到第几行的值 currentLineNum，范围是 0 ~ 25。

```
18  EXTERN  int      waiting;      //等候理发的顾客人数  
19  EXTERN  int      number;      //顾客编号  
20  EXTERN  SEMAPHORE customers;  //顾客信号量  
21  EXTERN  SEMAPHORE barbers;    //理发师信号量  
22  EXTERN  SEMAPHORE mutex;      //互斥信号量  
23  
24  EXTERN  int      currentLineNum; //当前输入到第几行
```

include/proc.h

为实现 P、V 操作，在进程结构体中增加 block 变量，指示当前进程是否被阻塞（1：阻塞，0：未阻塞）。同时新增信号量结构体，head、tail 指示当前信号量进程队列的首尾 index。

```

30 typedef struct s_proc {
31     STACK_FRAME regs;           /* process registers saved in stack frame */
32
33     u16 ldt_sel;                 /* gdt selector giving ldt base and limit */
34     DESCRIPTOR ldts[LDT_SIZE]; /* local descriptors for code and data */
35
36     int ticks;                  /* remained ticks */
37     int priority;               /* 优先级 */
38     int block;                  /* 是否被阻挡 */
39
40     u32 pid;                    /* process id passed in from MM */
41     char p_name[16];            /* name of the process */
42 }PROCESS;
43
44 typedef struct semaphore {
45     int value;
46     PROCESS* list[QUEUE_LENGTH];
47     int head;
48     int tail;
49 }SEMAPHORE;

```

include/proto.h

增加了系统调用和 main.c 的 kernel_main() 函数中的符号。

```

42 /* 以下是系统调用相关 */
43 /* proc.c */
44 PUBLIC int sys_get_ticks();           /* sys_call */
45 PUBLIC void sys_disp_str(char *);
46 PUBLIC void sys_disp_color_str(char *, int);
47 PUBLIC void sys_process_sleep(int);
48
49 /* syscall.asm */
50 PUBLIC void sys_call();               /* int_handler */
51 PUBLIC int get_ticks();
52 PUBLIC void disp_str_1(char *);
53 PUBLIC void disp_color_str_1(char *, int);
54 PUBLIC void process_sleep(int);
55
56
57 /* main.c */
58 void TestA();
59 void TestB();
60 void TestC();
61 void TestD();
62 void TestE();
63

```

kernel/clock.c

删去了原有的 p_proc_ready->ticks -= 1，将这个操作挪到 schedule() 函数中。

```
22 PUBLIC void clock_handler(int irq){
23     /**
24      *   Modified Here
25      */
26
27     //总 ticks + 1
28     ticks++;
29     //避免中断重入（楼下的函数会导致）
30     if (k_reenter != 0) {
31         return;
32     }
33     //在一个进程的 ticks 没变成 0 之前，其他进程没有机会执行
34     if (p_proc_ready->ticks > 0) {
35         return;
36     }
37     //调度进程
38     schedule();
39 }
```

kernel/global.c

在任务数组中添加新的 CDE 任务；在系统调用表中添加新的系统调用。

```
29 PUBLIC TASK    task_table[NR_TASKS] = {
30     {TestA, STACK_SIZE_TESTA, "TestA"},
31     {TestB, STACK_SIZE_TESTB, "TestB"},
32     {TestC, STACK_SIZE_TESTC, "TestC"},
33     {TestD, STACK_SIZE_TESTD, "TestD"},
34     {TestE, STACK_SIZE_TESTE, "TestE"}
35 };
36
37 PUBLIC irq_handler    irq_table[NR_IRQ];    //中断数组
38
39 PUBLIC system_call    sys_call_table[NR_SYS_CALL] = {
40     sys_get_ticks,
41     sys_disp_str,
42     sys_disp_color_str,
43     sys_process_sleep,
44     sys_sem_p,
45     sys_sem_v,
46     sys_process_wakeup
47     };
```

kernel/kernel.asm

考虑到会有 `disp_color_str(char* str, int color);` 的方法需要传入两个参数，所以将此处系统调用统一变为 `push` 两个参数到寄存器中。同时也要 `push` 进程的当前进程，以进行系统调用。

```
341 sys_call:
342     call    save
343     push    dword    [p_proc_ready]
344     sti
345
346     push    ecx
347     push    ebx
348     call    [sys_call_table + eax * 4] ;e
349     add     esp, 4 * 3
350
351     mov     [esi + EAXREG - P_STACKBASE], eax
352     cli
353     ret
```

kernel/syscall.asm

添加系统调用的序号、符号和调用。

```
10  _NR_get_ticks           equ 0 ;
11  _NR_disp_str_1          equ 1 ;
12  _NR_disp_color_str_1    equ 2 ;
13  _NR_process_sleep       equ 3 ;
14  _NR_sem_p               equ 4 ;
15  _NR_sem_v               equ 5 ;
16  _NR_process_wakeup      equ 6 ;
17
18  INT_VECTOR_SYS_CALL     equ 0x90
19
20  ; 导入全局变量
21  extern disp_pos
22
23  ; 导出符号
24  global get_ticks
25  global disp_str_1
26  global disp_color_str_1
27  global process_sleep
28  global sem_p
29  global sem_v
30  global process_wakeup
```

```

43 ; =====
44 ;                                     disp_str_1
45 ; =====
46 disp_str_1:
47     mov     eax, _NR_disp_str_1
48     mov     ebx, [esp + 4]
49     int     INT_VECTOR_SYS_CALL
50     ret
51
52 ; =====
53 ;                                     disp_color_str_1
54 ; =====
55 disp_color_str_1:
56     mov     eax, _NR_disp_color_str_1
57     mov     ebx, [esp + 4] ;argument1
58     mov     ecx, [esp + 8] ;argument2
59     int     INT_VECTOR_SYS_CALL
60     ret
61
62 ; =====
63 ;                                     process_sleep
64 ; =====
65 process_sleep:
66     mov     eax, _NR_process_sleep
67     mov     ebx, [esp + 4]
68     int     INT_VECTOR_SYS_CALL
69     ret
70
71 ; =====
72 ;                                     process_wakeup
73 ; =====
74 process_wakeup:
75     mov     eax, _NR_process_wakeup
76     mov     ebx, [esp + 4]
77     int     INT_VECTOR_SYS_CALL
78     ret
79
80 ; =====
81 ;                                     sem_p
82 ; =====
83 sem_p:
84     mov     eax, _NR_sem_p
85     mov     ebx, [esp + 4]
86     int     INT_VECTOR_SYS_CALL
87     ret
88
89 ; =====
90 ;                                     sem_v
91 ; =====
92 sem_v:
93     mov     eax, _NR_sem_v
94     mov     ebx, [esp + 4]
95     int     INT_VECTOR_SYS_CALL
96     ret

```

kernel/proc.c

首先修改 `schedule()`，让他在调度下一个进程时首先检查是否该进程被 `block`。同时给当前睡眠中的进程的 `ticks - 1`，以便后来可以被唤醒。

```
23 PUBLIC void schedule(){
24     PROCESS* p;
25     //调度进程
26     for (p = proc_table; p < proc_table + NR_TASKS; p++) {
27         if (p->ticks > 0) {
28             //给每个正在睡眠的进程的 ticks - 1
29             p->ticks--;
30         }
31     }
32
33     //寻找下一个进程
34     while(1){
35         //遍历任务表
36         if(++p_proc_ready >= proc_table + NR_TASKS){
37             p_proc_ready = proc_table;
38         }
39         //当前进程没被 block, 且可以运行
40         if(p_proc_ready->ticks <= 0 && p_proc_ready->block != 1){
41             break;
42         }
43     }
44 }
45
46 PUBLIC void sys_disp_str(char* str){
47     disp_str(str);
48 }
49
50 /*=====
51                                sys_disp_color_str
52     *=====*/
53 PUBLIC void sys_disp_color_str(char* str, int color){
54     disp_color_str(str, color);
55     //判断是否满屏要清屏
56     int i = 0;
57     while(1){
58         if(str[i] == '\0'){
59             break;
60         }
61         if(str[i] == '\n'){
62             currentLineNum++;
63         }
64         i++;
65     }
66     if(currentLineNum == 25){
67         clearScreen();
68     }
69 }
70
71 ..
```

上一页的后半部分和这一页是增加的系统调用（PV 操作、沉睡/唤醒进程、打印彩色字符串）的 c 实现，均由 syscall.asm 以中断方式调用。

```
85 PUBLIC void sys_process_sleep(int mill_seconds){
86     //进程沉睡, 设置 ticks 数
87     p_proc_ready->ticks = mill_seconds / 1000 * HZ;
88     //沉睡不分配时间片, 调度下一个进程
89     schedule();
90 }
91
92 /*=====
93 sys_process_wakeup
94 *=====*/
95 PUBLIC void sys_process_wakeup(PROCESS* p) {
96     p->block = 0;
97     p_proc_ready = p;
98 }
99
100 /*=====
101 sys_sem_p
102 *=====*/
103 PUBLIC void sys_sem_p(SEMAPHORE* s){
104     s->value--; //信号量值 - 1
105     if (s->value < 0) { //结果小于 0, 执行 P 操作的进程被阻塞
106         s->list[s->head] = p_proc_ready; //排入 list 队列中
107         //阻止当前进程, 调度下一个进程
108         p_proc_ready->block = 1;
109         schedule();
110         s->head = (s->head + 1) % QUEUE_LENGTH;
111     }
112     //否则继续执行
113 }
114
115 /*=====
116 sys_sem_v
117 *=====*/
118 PUBLIC void sys_sem_v(SEMAPHORE* s){
119     s->value++; //信号量值 + 1
120     if (s->value <= 0) { //结果不大于 0, 执行 V 操作的进程从 list 队列中释放一个进程
121         PROCESS* p = s->list[s->tail];
122         wakeup(p); //并将其转换为就绪态
123         s->tail = (s->tail + 1) % QUEUE_LENGTH;
124     }
125 }
126
127 /*=====
128 wakeup
129 *=====*/
130 PUBLIC void wakeup(PROCESS* p){ //唤醒一个进程
131     process_wakeup(p);
132 }
133
```


kernel/main.c

首先在 kernel_main() 中增加对理发师问题的全局变量的初始化，并定义一个清屏操作。

```
85     waiting      = 0;
86     number       = 0;
87
88     customers.value = 0;
89     customers.head  = 0;
90     customers.tail  = 0;
91
92     barbers.value  = 0;
93     barbers.head   = 0;
94     barbers.tail   = 0;
95
96     mutex.value    = 1;
97     mutex.head     = 0;
98     mutex.tail     = 0;
107 void clearScreen(){
108     int i = 0;
109     disp_pos = 0;
110     for(i = 0; i < 80 * 25; i++){
111         disp_str(" \0");
112     }
113     disp_pos = 0;
114     currentLineNum = 0;
115 }
```

其次增加/修改 TestA、B、C、D、E，其中 A 为一般进程，B 为理发师进程，C、D、E 为顾客进程。

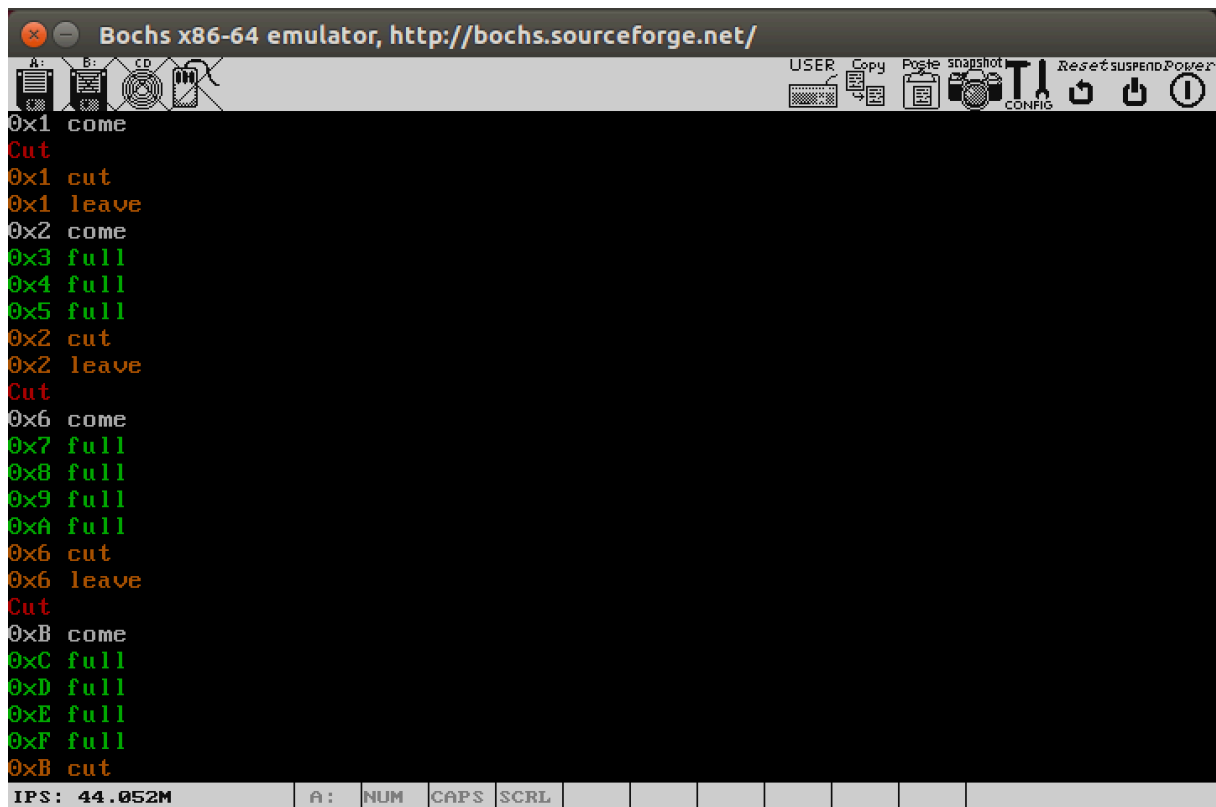
```
163 void TestA(){
164     while (1) {
165     }
166 }
167
168
169 void TestB(){
170     while(1){
171         sem_p(&customers);           //判断是否有顾客，若无顾客，理发师睡眠
172         sem_p(&mutex);               //若有顾客，进入临界区
173         waiting--;                  //等待顾客数减1
174         sem_v(&barbers);             //理发师准备为顾客理发
175         //理发师正在理发（非临界区）
176         sem_v(&mutex);               //退出临界区
177         disp_color_str_1("Cut\n\0", 0x04);
178         process_sleep(2000);
179     }
180 }
181
182 void TestC(){
183     customer();
184 }
185
186 void TestD(){
187     customer();
188 }
189
190 void TestE(){
191     customer();
192 }
```


还有添加一些打印字符串和顾客进程的实现，均参考了教材第三章代码。

```
121 void come(int customer){
122     disp_color_int(customer, 0x07);
123     disp_color_str_1(" come\n\0", 0x07);
124 }
125
126 void haircut(int customer){
127     disp_color_int(customer, 0x06);
128     disp_color_str_1(" cut\n\0", 0x06);
129 }
130
131 void leave(int customer){
132     disp_color_int(customer, 0x06);
133     disp_color_str_1(" leave\n\0", 0x06);
134 }
135
136 void full(int customer){
137     disp_color_int(customer, 0x02);
138     disp_color_str_1(" full\n\0", 0x02);
139 }
140
141 void customer(){
142     int temp;
143     while(1) {
144         sem_p(&mutex);                //进入临界区
145         number++;                      //顾客编号加1
146         temp = number;                //暂时记录现在的顾客是几号
147         if (waiting < CHAIRS) {       //判断是否有空椅子
148             waiting++;                //等待顾客加1
149             come(temp);               //打印来了哪个顾客
150             sem_v(&customers);         //唤醒理发师
151             sem_v(&mutex);             //退出临界区
152             sem_p(&barbers);           //理发师忙，顾客坐着等待
153             haircut(temp);             //给顾客剪头发
154             leave(temp);              //顾客离开
155         } else {
156             sem_v(&mutex);             //退出临界区
157             full(temp);               //人满了，顾客离开
158         }
159         process_sleep(1000);
160     }
161 }
```

至此全部已修改完毕。更改椅子数后运行的结果如下（第一页）：

CHAIR = 1

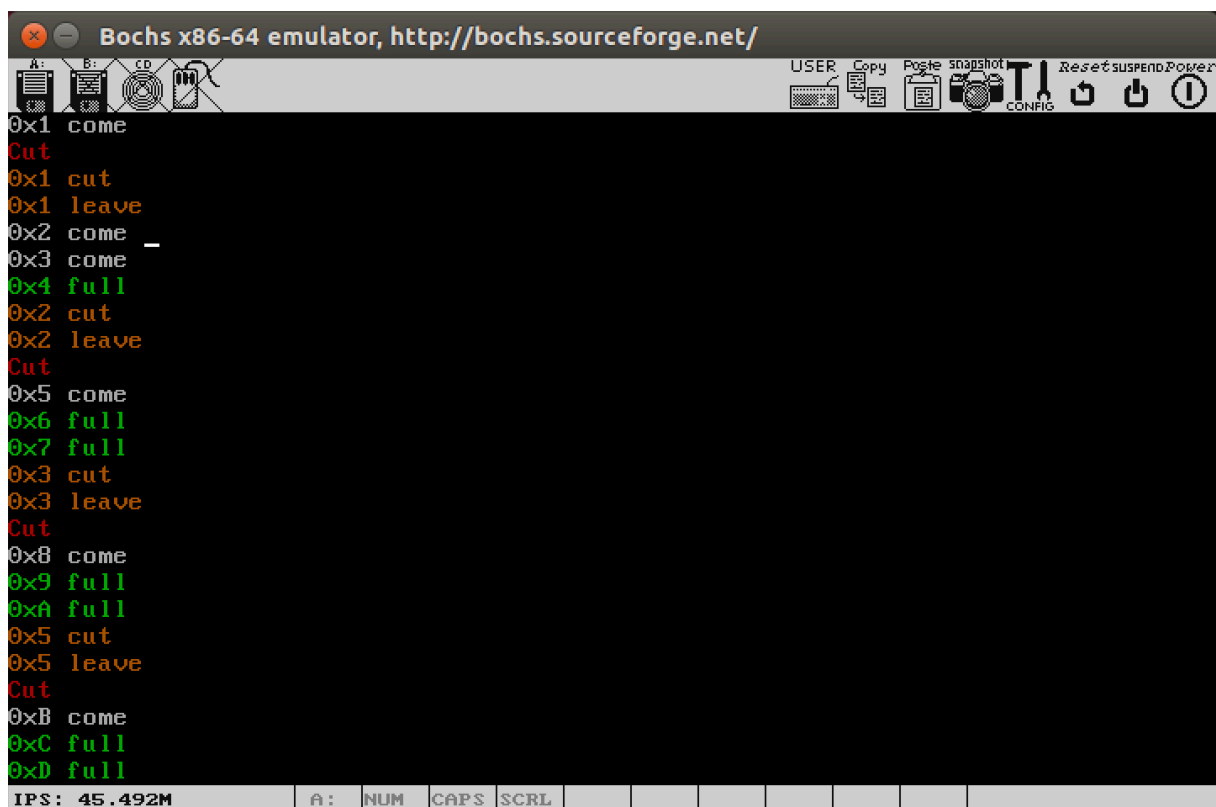


The screenshot shows the Bochs x86-64 emulator window. The title bar reads "Bochs x86-64 emulator, http://bochs.sourceforge.net/". The window contains a list of assembly instructions with their addresses and comments. The instructions are as follows:

```
0x1 come
Cut
0x1 cut
0x1 leave
0x2 come
0x3 full
0x4 full
0x5 full
0x2 cut
0x2 leave
Cut
0x6 come
0x7 full
0x8 full
0x9 full
0xA full
0x6 cut
0x6 leave
Cut
0xB come
0xC full
0xD full
0xE full
0xF full
0xB cut
```

The status bar at the bottom shows "IPS: 44.052M" and a row of buttons: A, NUM, CAPS, SCRL, and several empty slots.

CHAIR = 2



The screenshot shows the Bochs x86-64 emulator window. The title bar reads "Bochs x86-64 emulator, http://bochs.sourceforge.net/". The window contains a list of assembly instructions with their addresses and comments. The instructions are as follows:

```
0x1 come
Cut
0x1 cut
0x1 leave
0x2 come
0x3 come
0x4 full
0x2 cut
0x2 leave
Cut
0x5 come
0x6 full
0x7 full
0x3 cut
0x3 leave
Cut
0x8 come
0x9 full
0xA full
0x5 cut
0x5 leave
Cut
0xB come
0xC full
0xD full
```

The status bar at the bottom shows "IPS: 45.492M" and a row of buttons: A, NUM, CAPS, SCRL, and several empty slots.

CHAIR = 3

The screenshot shows a Bochs x86-64 emulator window. The title bar reads "Bochs x86-64 emulator, http://bochs.sourceforge.net/". The interface includes a toolbar with icons for USER, Copy, Paste, snapshot, CONFIG, Reset, suspend, and Power. The main display area shows a list of instructions and their execution status:

Instruction	Status
0x1 come	
Cut	
0x1 cut	
0x1 leave	
0x2 come	
0x3 full	
0x4 full	
0x5 full	
0x2 cut	
0x2 leave	
Cut	
0x6 come	
0x7 full	
0x8 full	
0x9 full	
0xA full	
0x6 cut	
0x6 leave	
Cut	
0xB come	
0xC full	
0xD full	
0xE full	
0xF full	
0xB cut	

At the bottom, a status bar shows "IPS: 44.052M" and a table of registers: A, NUM, CAPS, SCRL, and several empty slots.