本实验在 orange’s 的第八章的代码基础上修改，增加了不占用时间片的 sys_process_sleep, 系统调用，还有支持 PV 原语操作的 sys_sem_p 和 sys_tem_v 系统调用,模拟了睡眠理发师的问题。代码中的打印系统调用已经存在为 sys_printx，而且再次基础上封装好了 printf 与 C 库中的 prinf(orange’s7.5.2)相媲美，省去了不少工作。

7.5.2 printf()的实现

函数printf()对于我们来说肯定是非常熟悉，从学习HelloWorld的时候就开始用它了。但它的实现却并不简单，首先是它的参数个数和类型都可变，而且其表示格式的参数（比如“%d”、“%x”等）形式多样，在printf()中都要加以识别。

不过，按照我们一贯的风格，开始时只实现一个简单点的。下面的printf只支持“%x”一种格式（代码7.55）。

代码 7.55 printf（chapter7/o/kernel/printf.c）

```
52 int printf(const char *fmt, ...)
53 {
54         int i;
55         char buf[256];
56
57         va_list arg = (va_list)((char*)(&fmt) + 4); /*4是参数fmt所占堆栈中的大小*/
58         i = vsprintf(buf, fmt, arg);
59         write(buf, i);
60
61         return i;
62 }
```

其中，vsprintf()的实现方法如代码7.56所示。

代码 7.56 vsprintf（chapter7/o/kernel/vsprintf.c）

```
19 int vsprintf(char *buf, const char *fmt, va_list args)
20 {
21         char*   p;
22         char    tmp[256];
23         va_list p_next_arg = args;
24
25         for (p=buf;*fmt;fmt++) {
26                 if (*fmt != '%') {
27                         *p++ = *fmt;
28                         continue;
29                 }
30
31                 fmt++;
32
33                 switch (*fmt) {
34                 case 'x':
35                         itoa(tmp, *((int*)p_next_arg));
36                         strcpy(p, tmp);
37                         p_next_arg += 4;
38                         p += strlen(tmp);
39                         break;
40                 case 's':
41                         break;
42                 default:
43                         break;
44                 }
45         }
46
47         return (p - buf);
48 }
```

新增的内容及解释：

1. 增加 D(customer2).E(customer3)用户进程（原来已有 A(任务)B(barber)C（customer1）用

户进程），参考 ORANGE’S 第 6 章 6.4.6（207 页）

2.

果然，简单几步就成功地添加了一个任务，我们把添加任务的步骤总结一下。增加一个任务需要的步骤：

1. 在task_table中增加一项（global.c）。
2. 让NR_TASKS加1（proc.h）。
3. 定义任务堆栈（proc.h）。
4. 修改STACK_SIZE_TOTAL（proc.h）。
5. 添加新任务执行体的函数声明（proto.h）。

除了任务本身的代码和一些宏定义之外，原来的代码几乎不需要做任何改变，看来我们的代码的自动化程度还是不错的，这真让人高兴！

7.4 区分任务和用户进程

现在，我们有了4个进程，分别是TTY、A、B、C。其中A、B和C是可有可无的，它其实不是操作系统的一部分，而更像用户在执行的程序。而TTY则不同，它肩负着重大的职责，没有它我们连键盘都无法使用。

所以，我们有必要把它们区分开来，分为两类。我们称TTY为“任务”，而称A、B、C为“用户进程”。

在具体的实现上，也来做一些相应的改变，让用户进程运行在ring3，任务继续留在ring1。这样就形成了图7.24所示的情形。

然后在所有用到NR_TASKS的地方都要做相应修改，首先是proc_table和task_table（代码7.49）。

代码7.49 区分task和proc（chapter7/n/kernel/global.c）

```
20 PUBLIC PROCESS proc_table[NR_TASKS + NR_PROCS];
21
22 PUBLIC TASK task_table[NR_TASKS] = {
23 {task_tty, STACK_SIZE_TTY, "tty"}};
24
25 PUBLIC TASK user_proc_table[NR_PROCS] = {
26 {TestA, STACK_SIZE_TESTA, "TestA"},
27 {TestB, STACK_SIZE_TESTB, "TestB"},
28 {TestC, STACK_SIZE_TESTC, "TestC"}};
```

我们新声明了一个数组user_proc_table[ ]，实际上这是权宜之计，因为完善的操作系统应该有专门的方法来新建一个用户进程，不过目前使用与任务相同的方法来做无疑是简单的。

添加步骤：a.因为新增为用户进程在 user_proc_table 中增加一项（global.c）.

```
22 PUBLIC   struct task     task_table[NR_TASKS] = {
23          {task_tty, STACK_SIZE_TTY, "TTY"},
24          {task_sys, STACK_SIZE_SYS, "SYS"},
25          {TestA, STACK_SIZE_TESTA, "PROCA"}};
26
27 PUBLIC   struct task     user_proc_table[NR_PROCS] = {
28          {TestB, STACK_SIZE_TESTB, "Barber"},
29          {TestC, STACK_SIZE_TESTC, "Customer1"},
30          {TestD,STACK_SIZE_TESTD,"Customer2"},
31          {TestE,STACK_SIZE_TESTE,"Customer3"}
32          };
```

b.让 NR_TASK 加 1（proc.h）.

```
83
84 /* Number of tasks & procs */
85 #define NR_TASKS        3
86 #define NR_PROCS        4
87 #define FIRST_PROC      proc_table[0]
88 #define LAST_PROC       proc_table[NR_TASKS + NR_PROCS - 1]
89
```

c.定义任务堆栈（proc.h）

d.修改 STACK_SIZE_TOTAL(proc.h).

```
 89
 90 /* stacks of tasks */
 91 #define STACK_SIZE_TTY          0x8000
 92 #define STACK_SIZE_SYS          0x8000
 93 #define STACK_SIZE_TESTA        0x8000
 94 #define STACK_SIZE_TESTB        0x8000
 95 #define STACK_SIZE_TESTC        0x8000
 96 #define STACK_SIZE_TESTD        0x8000
 97 #define STACK_SIZE_TESTE        0x8000
 98
 99 #define STACK_SIZE_TOTAL        (STACK_SIZE_TTY + \
100                                 STACK_SIZE_SYS + \
101                                 STACK_SIZE_TESTA + \
102                                 STACK_SIZE_TESTB + \
103                                 STACK_SIZE_TESTC+STACK_SIZE_TESTD
    +STACK_SIZE_TESTE\
104                                     )
```

e.添加新任务执行体的函数声明（proto.h）

```
37 PUBLIC void TestA();                 //normal proc
38 PUBLIC void TestB();                 //barber proc
39 PUBLIC void TestC();                 //customers1 proc
40 PUBLIC void TestD();                 //customers2 proc
41 PUBLIC void TestE();                 //customer33 proc
```

D.E 进程体位于 main.c

```
341 void TestD()
342 {
343     int i  = 1;
344    while (1)
345    {
346         /* assert(0); */
347     p(&mutex);
348    printf("customer  %d come!\n",++customerNum);
349    i = customerNum;
350    if (waiting < CHAIR)
351    {
352     printf("customer  %d wait!\n",i);
353     waiting++;
354     v(&customers);
355     v(&mutex);
356     p(&barbers);
357     printf("customer  %d get service!\n",i);
358     }
359    else
360    {
361     v(&mutex);
362    }
363    printf("customer  %d leaves\n",i);
364    ++i;
365   }
366 }
367 void TestE()
368 {
369     int i  = 1;
370    while (1)
371    {
372         /* assert(0); */
373     p(&mutex);
374    printf("customer  %d come!\n",++customerNum);
375    i = customerNum;
376    if (waiting < CHAIR)
377    {
378     printf("customer  %d wait!\n",i);
379     waiting++;
380     v(&customers);
```

```
381     v(&mutex);
382     p(&barbers);
383     printf("customer  %d get service!\n",i);
384     }
385    else
386    {
387     v(&mutex);
388    }
389    printf("customer  %d leaves\n",i);
390    ++i;
391   }
392 }
```

2.增加系统调用,Orange’s

8.3 节中提到 sys.printx 的系统调用,于是模仿该系统调用的实现增加 sys_process_sleep 、sys_tem_p 和 sys_tem_v 系统调用.

步骤如下：

1. 添加函数声明,用户级系统调用（sleep,p,v）文件 proto.h

```
104 /* 系统调用 - 用户级 */
105 PUBLIC  int     sendrec(int function, int src_dest, MESSAGE* p_msg);
106 PUBLIC  int     printx(char* str);
107 PUBLIC  int     sleep(int milli_sec);
108 PUBLIC  int     p(struct semaphore * s);
109 PUBLIC  int     v(struct semaphore * s );
```

2. 用汇编代码实现 1.中添加的用户级系统调用,并设为 global，模仿着 printx 的_NR_printx 常量的使用，添加_NR_sleep,_NR_P,_NR_V 常量，分别为 2,3,4(syscall.asm)，添加函数体（syscall.asm）

```
10 INT_VECTOR_SYS_CALL equ 0x90
11 _NR_printx          equ 0
12 _NR_sendrec         equ 1
13 _NR_sleep           equ 2
14 _NR_P               equ 3
15 _NR_V               equ 4
```

```
45 sleep:
46         mov     eax,_NR_sleep
47         mov     edx,[esp + 4]
48         int     INT_VECTOR_SYS_CALL
49         ret
50
51 p:      mov      eax,_NR_P
52         mov      edx,[esp+4]
53         int      INT_VECTOR_SYS_CALL
54         ret
55
56 v:      mov      eax,_NR_V
57         mov      edx,[esp+4]
58         int     INT_VECTOR_SYS_CALL
59         ret
```

3.global.c 的 sys_call_table 的调用列表中添加新添加的系统调用（sys_process_sleep,sys_tem_p,sys_tem_v）并给 NR_SYS_CALL（const.h）设置成 5

```
42 PUBLIC  system_call     sys_call_table[NR_SYS_CALL] = {sys_printx,
43
   sys_sendrec,sys_process_sleep,sys_tem_p,sys_tem_v};
```

```
126 /* system call */
127 #define NR_SYS_CALL     5
```

3. 查看 sys_call_table 的汇编实现（kernel.asm）发现 eax 代表 sys_call_table 的下标，根据此下标调用 sys_call_table 中的函数而 call 上面的 4 个 push 操作分别为函数中的四个形参，最后一个形参为当前进程的的指针（即指向一个 proc 函数体的指针）

```
350 sys_call:
351         call    save
352
353         sti
354         push    esi
355
356         push    dword [p_proc_ready]
357         push    edx
358         push    ecx
359         push    ebx
360         call    [sys_call_table + eax * 4]
361         add     esp, 4 * 4
362
363         pop     esi
364         mov     [esi + EAXREG - P_STACKBASE], eax
365         cli
366
367         ret
```

由于当前运行的进程就是通过设置p_proc_ready来恢复执行的，所以当进程切换到未发生之前，p_proc_ready的值就是指向当前进程的指针。把它压栈就将当前进程，即write( )的调用者指针传递给了sys_write( )。

6.模仿 sys_printx 函数的声明，分别对 sys_process_sleep、sys_tem_p、sys_tem_v 进行声明。根据实参的个数将最左边的两个形参为 unused 未使用,函数声明位于
proto.h,

```
 98 PUBLIC void sys_process_sleep(int unused1,int unused2,int
    milli_sec,struct proc * p);
 99 PUBLIC void sys_tem_p(int unused1,int unused2,struct semaphore *
    s,struct proc * p);
100 PUBLIC void sys_tem_v(int unused1,int unused2,struct semaphore *
    s,struct proc * p);
```

其中 semaphore 为信号量的数据结构 proc.h，为实现方便将课本中的链表实现的队列改成了用数组实现的队列

```
75 struct semaphore
76 {
77   int value;     //信号量的值
78   int len;
79   struct proc * list[10]; //等待队列
80 };
```

7.步骤 6 中的三个函数体的不同实现：

sys_process_sleep(proc.c):

```
40 PUBLIC void sys_process_sleep(int unused1,int unused2,int
   milli_sec,struct proc * p)
41 {
42   int tick_time =  milli_sec * HZ /1000;
43   setSleep_ticks(tick_time,p);
44 }
```

基本思想：用户调用 sleep 函数后传入毫秒数，在 sys_process_sleep 将该毫秒转化为相应的 ticks 数（即时钟中断次数），在 proc 结构体中已经添加 sleep_ticks 变量，设置该变量的数值，并在时钟中断促发调用的进程调度函数 schedule 中规定 sleep_ticks 不为 0 的进程不被分配时间片，且每次进程调度，该 sleep_ticks 都会减去 1 直到为 0.最后返回到 sleep 函数，返回到 goToSleep 函数，将剩下的时间片用循环耗尽。（currentSleep 在下一次时钟中断会被设为 0）

主要代码：main.c

sys_tem_p 与 sys_tem_v 主要算法和思想参照课本中 3.3 信号量和 PV 操作（课本第 136 页至第 137 页）

```
38 PUBLIC void sys_tem_p(int unused1,int unused2,struct semaphore *
   s,struct proc * p)
39 {
40   s->value--; //信号量减1
41   int i  =0;
42 /*若信号量值小于0，执行P操作的进程调用sleep1(s)阻塞自己，被
43 移入等待队列，转向进程调度程序*/
44  if (s->value<0)
45   sleep1(s);
46 }
```

```
49 PUBLIC void sys_tem_v(int unused1,int unused2,struct semaphore *
   s,struct proc * p)
50 {
51  s->value++;//信号量值加1
52 /*若信号量值小于等于0，则调用wakeup1(s)从信号量队列中释放一个等待信号量s的进程并
   转换成就绪态，进程则继续执行*/
53   if (s->value<=0)
54     wakeup1(s);
55 }
```

由于中断处理过程中默认关中断，因此以上 PV 操作均为原子操作

睡眠理发师模拟：

B 进程为理发师，CDE 为顾客，理发师处于循环体中持续提供服务，顾客只来一次

设置全局变量：global.c

```
44 PUBLIC int waiting = 0; //等候理发的顾客人数
45 PUBLIC int CHAIR = 3;//为顾客准备的椅子数 可在检查时设定
46 PUBLIC struct semaphore customers,barbers,mutex;
```

在 main.c 的 kernnel_main 中为初始化

```
137         //初始化
138          mutex.value = 1;
139          waiting = 0;
140          CHAIR = 3;
```

testB: 理发师

```
299 void TestB()
300 {
301   printf("\n");
302   while (1)
303 {
304     p(&customers);//判断是否有顾客，若无顾客，理发师睡眠
305     p(&mutex);//若有顾客，进入临界区
306     waiting--;//等候顾客数减1
307     v(&barbers);//理发师准备为顾客理发
308     v(&mutex);//退出临界区
309     printf("barber cutting!\n");//理发师开始理发（消耗2个时间片 ticks）
    10ms一次
310 //时钟中断
311     goToSleep(20);
312 }
313 }
```

顾客进程：以 TestC 为例：

全局变量记录顾客号码

```
18 int customerNum = 0;
```

customer1:

```
318 void TestC()
319 {
320    int i  = 1;
321    while (1)
322    {
323         /* assert(0); */
324
325     p(&mutex);//进入临界区
326   printf("customer  %d come!\n",++customerNum);
327 //顾客号码
328    i = customerNum;
329  //判断是否有空椅子
330    if (waiting < CHAIR)
331    {
332     printf("customer  %d wait!\n",i);
333     waiting++;//等候顾客数加1
334     v(&customers); //唤醒理发师
335     v(&mutex);//退出临界区
336     p(&barbers);//理发师忙，顾客坐着等待
337     printf("customer  %d get service!\n",i);//否则顾客可以理发
338     }
```

```
339    else
340    {
341     v(&mutex);//人满了，顾客离开
342    }
343    printf("customer  %d leaves\n",i);
344    ++i;
345   }
346
347 }
```

不同进程变色：

sys_printx 会调用 out_char 输出

在 console.c 中添加

```
 93 PUBLIC void out_char(CONSOLE* p_con, char ch)
 94 {
 95         char ch_color = DEFAULT_CHAR_COLOR;
 96         switch(p_proc_ready->pid)
 97         {
 98         case 3:
 99            ch_color = 0x0C;
100            break;
101         case 4:
102           ch_color = 0X0A;
103           break;
104         case 5:
105            ch_color = 0X03;
106            break;
107         case 6:
108            ch_color = 0X06;
109            break;
110          }
```

```
            *p_vmem++ = ch;
            *p_vmem++ = ch_color;
```

Proc.h

```
31 struct proc {
32         struct stackframe regs;    /* process registers saved in
   stack frame */
33
34         u16 ldt_sel;               /* gdt selector giving ldt base
   and limit */
35         struct descriptor ldts[LDT_SIZE]; /* local descs for code
   and data */
36
37         int ticks;                 /* remained ticks */
38         int priority;
39
40         u32 pid;                   /* process id passed in from MM *
```

Orange’s 7.5.3

由于当前运行的进程就是通过设置p_proc_ready来恢复执行的，所以当进程切换到未发生之前，p_proc_ready的值就是指向当前进程的指针。把它压栈就将当前进程，即write( )的调用者指针传递给了sys_write( )。

遇到的问题：

此为 TTY 代码基础上改的，理发师问题的输出在 2 号窗口上（main.c 第 125 行至第 129 行），在 tty.c 第 42 行设置 select_console(1)默认显示第二个窗口

为演示方便。为显示的 console 分配了一屏的显存空间，到底部时进行清屏，会有刷屏的效果。

Screenshot from 2016-06-12 02:40:42