本次实验使用 Orange‘s 第 8 章源码，在此基础上修改完成，并参考了往年学长学姐的代码。

1、添加系统调用

（1）在 proto.h 里添加函数声明，这里是用户调用的函数声明

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

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

将 NR_SYS_CALL 设置为 5

（2）在 syscall.asm 里设置调用号和调用的 nasm 代码

```
INT_VECTOR_SYS_CALL equ 0x90
_NR_printx      equ 0
_NR_sendrec     equ 1
_NR_sleep       equ 2
_NR_P           equ 3
_NR_V           equ 4
```

nasm 代码模仿书上的格式：

```
    ret
sleep:
        mov     eax,_NR_sleep
        mov edx,[esp + 4]
        int INT_VECTOR_SYS_CALL
        ret
```

（3）在 global.c 系统调用表中加入新的系统调用

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

（4）在 proto.h 里添加系统调用的函数申明，在 kernel.asm 里 syscall 传入了四个参数，这里不需要全部用到，最后一个参数是指向当前进程的指针。

```
/* 系统调用 - 系统级 */
/* proc.c */
PUBLIC  int sys_sendrec(int function, int src_dest, MESSAGE* m, struct proc* p);
PUBLIC  int sys_printx(int _unused1, int _unused2, char* s, struct proc * p_proc);
PUBLIC void sys_process_sleep(int unused1,int unused2,int milli_sec,struct proc * p);
PUBLIC void sys_tem_p(int unused1,int unused2,struct semaphore * s,struct proc * p);
PUBLIC void sys_tem_v(int unused1,int unused2,struct semaphore * s,struct proc * p);
```

其中 semaphore 是信号量的数据结构，定义在 proc.h 文件里,这里使用数组代表等待队列:

```
struct semaphore
{
  int value;
  int len;
  struct proc * list[10];
};
```

（5）系统调用函数的实现

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

sleep：p->sleep_ticks = ticks;设置进程的 sleep_ticks，在进程调度时若 sleep 不为 0 则不参与进程调度，但会将 sleep_ticks 减 1，直到为 0 时重新参与进程调度。

p、v 操作实现在 main.c 里，因为中断的时候这里不允许再中断，所以 pv 操作就变成了原子操作

p 操作：信号量减 1，若小于 0 则阻塞自己移入等待队列，开始下一次进程调度

```
s->value--;
if (s->value<0) {
    s->list[s->len++] = p_proc_ready;
    p_proc_ready->p_flags = 1;
    schedule();
}
```

v 操作：信号量加 1，若小于等于 0 则从等待队列释放一个进程

```
s->value++;
if (s->value<=0) {
    s->list[0]->p_flags = 0;
    int i = 1;
    for (i=1;i<=s->len;++i) {
        s->list[i-1] = s->list[i];
    }
    s->len--;
}
```

print 系统调用的代码原来就有，在 tty.c 里：

```
/*==========================================================================
                                    sys_printx
*==========================================================================
PUBLIC int sys_printx(int _unused1, int _unused2, char* s, struct p
{
    const char * p;
```

封装好的 printf 在 printf.c 里：

```
int printf(const char *fmt, ...) {
    int i;
    char buf[256];

    va_list arg = (va_list)((char*)(&fmt) + 4); /*4是参数fmt所占堆栈中的大
    i = vsprintf(buf, fmt, arg);
    buf[i] = 0;
    printx(buf);

    return i;
}
```

2、睡眠理发师问题

（1）添加顾客进程：

在 global.c 的用户进程添加 D、E

```
PUBLIC  struct task user_proc_table[NR_PROCS] = {
    {TestB, STACK_SIZE_TESTB, "Barber"},
    {TestC, STACK_SIZE_TESTC, "Customer1"},
    {TestD,STACK_SIZE_TESTD,"Customer2"},
    {TestE,STACK_SIZE_TESTE,"Customer3"}
};
```

proc.h 里的修改，修改任务数量

```
#define NR_TASKS     3
#define NR_PROCS     4
```

定义任务堆栈和堆栈大小

```
#define STACK_SIZE_TTY      0x8000
#define STACK_SIZE_SYS      0x8000
#define STACK_SIZE_TESTA    0x8000
#define STACK_SIZE_TESTB    0x8000
#define STACK_SIZE_TESTC    0x8000
#define STACK_SIZE_TESTD    0x8000
#define STACK_SIZE_TESTE    0x8000

#define STACK_SIZE_TOTAL    (STACK_SIZE_TTY + \
                STACK_SIZE_SYS + \
                STACK_SIZE_TESTA + \
                STACK_SIZE_TESTB + \
                STACK_SIZE_TESTC+STACK_SIZE_TESTD+STACK_SIZE_TESTE\
                                )
```

添加函数申明（proto.h）：

```
PUBLIC void TestC();                //customers1 proc
PUBLIC void TestD();                //customers2 proc
PUBLIC void TestE();                //customer33 proc
```

（2）具体实现理发师问题

global.c 里定义全局变量：等待人数，椅子数和信号量

```
PUBLIC int waiting = 0;
PUBLIC int CHAIR = 3;
PUBLIC struct semaphore customers,barbers,mutex;
```

main.c 的 kernal_main（）里初始化变量

```
    mutex.value = 1;
    waiting = 0;
    CHAIR = 2;
```

理发师进程：

```
void TestB() {
  printf("\n");
  while (1) {
    p(&customers);
    p(&mutex);
    waiting--;
    printf("barber cutting!\n");
    v(&mutex);
    v(&barbers);
    goToSleep(60);
  }
}
```

解释：首先查看是否有顾客，没有则阻塞自己，有则进入临界区，将等待人数减 1，退出临界区，开始理发

顾客进程：

```c
void TestD() {
  int i  = 1;
  while (1) {
  /* assert(0); */
    p(&mutex);
    printf("customer  %d come!\n",++customerNum);
    i = customerNum;
    if (waiting < CHAIR) {
      printf("customer  %d wait!\n",i,waiting);
      waiting++;
      v(&customers);
      v(&mutex);
      p(&barbers);
      printf("customer  %d get service!\n",i);
    } else {
      v(&mutex);
      printf("not enough chairs!-------------------\n");
    }
    printf("customer  %d leaves\n",i);
    ++i;
    goToSleep(5);
  }
}
```

解释：进入临界区，如果有椅子则等待，增加顾客数，申请理发师，否则直接离开。

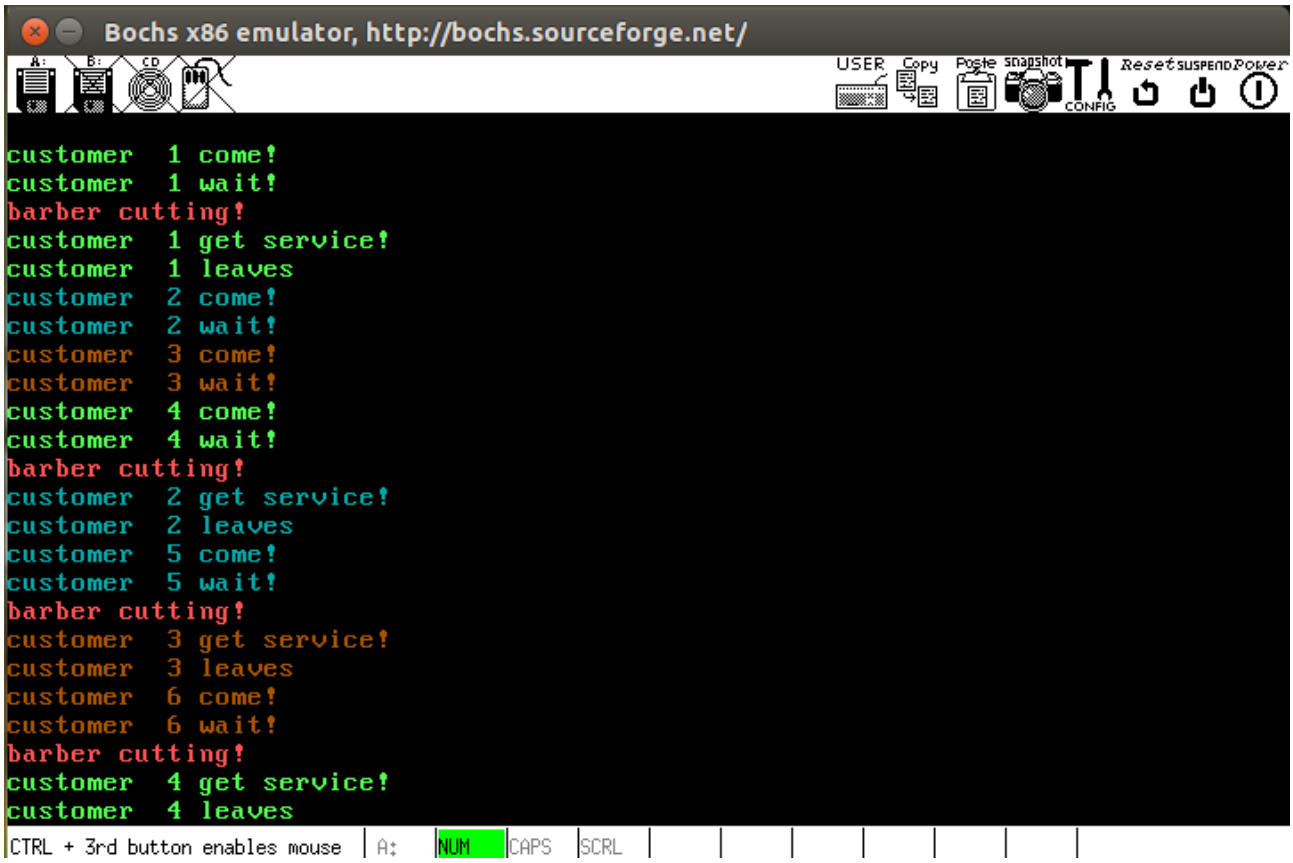
3、输出颜色改变

在 console.c 的 outchar 函数里判断进程号

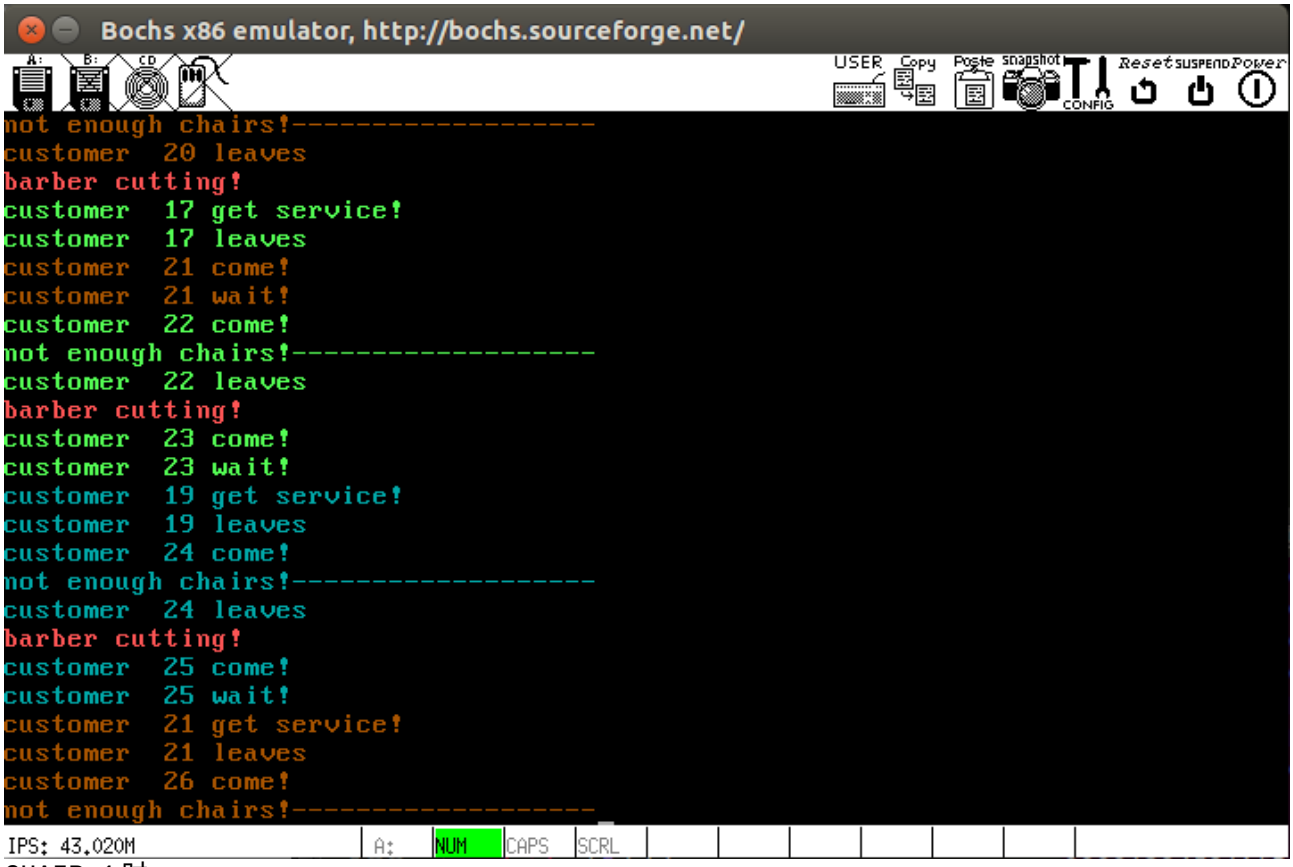
```c
PUBLIC void out_char(CONSOLE* p_con, char ch)
{
        char ch_color = DEFAULT_CHAR_COLOR;
        switch(p_proc_ready->pid) {
        case 3:
            ch_color = 0x0C;
            break;
        case 4:
            ch_color = 0X0A;
            break;
        case 5:
            ch_color = 0X03;
            break;
        case 6:
            ch_color = 0X06;
            break;
        }
```

输出结果：

CHAIR=3 时：



CHAIR=2 时：



CHAIR=1 时：

Bochs x86 emulator, http://bochs.sourceforge.net/

```
customer  1 come!
customer  1 wait!
barber cutting!
customer  1 get service!
customer  1 leaves
customer  2 come!
customer  2 wait!
customer  3 come!
not enough chairs!-------------------
customer  3 leaves
customer  4 come!
not enough chairs!-------------------
customer  4 leaves
barber cutting!
customer  2 get service!
customer  2 leaves
customer  5 come!
customer  5 wait!
customer  6 come!
not enough chairs!-------------------
customer  6 leaves
customer  7 come!
not enough chairs!-------------------
customer  7 leaves
```

CTRL + 3rd button enables mouse | A: | NUM | CAPS | SCRL