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。

新增的内容及解释：

1.增加 D.E 用户进程（原来已有 ABC 用户进程），参考 ORANGE’S 第 6 章 6.4.6（207 页）

添加步骤：

(1).在 task_table 中增加一项（global.c）.

(2).让 NR_TASK 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（proc.h）

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），proto.h 第 104 行至 109 行

(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 第 13 行至第 15 行)，添加函数体 （syscall.asm 第 45 行至第 59 行）

(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 设置成 5（const.h 第 127 行）

(4).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。

更具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 第 98 行至第 100 行,其中 semaphore 为信号量的数据结构

(5).步骤(4)中的三个函数体的不同实现：

sys_process_sleep(proc.c 第 40 行):

基本思想：用户调用 sleep 函数后传入个毫秒数，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：mainc.c

int tick_time = milli_sec * HZ /1000;

setSleep_ticks(tick_time,p);

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），其中 semaphore 结构为

struct semaphore{

int value; //信号量的值

int len;

struct proc * list[10]; //等待队列

};位于 proc.h 第 75 行至第 79 行。不同的是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

PUBLIC void sys_tem_p(int unused1,int unused2,struct semaphore * s,struct proc * p){

s->value--; //信号量减 1

int i =0;

/*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

移入等待队列，转向进程调度程序*/

if (s->value<0)

sleep1(s);

//sys_printx(0,0,"sdfsdf",p_proc_ready);

}

PUBLIC void sys_tem_v(int unused1,int unused2,struct semaphore * s,struct proc * p){

s->value++; //信号量值加 1

/*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*/

if (s->value<=0)

wakeup1(s);

}

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 第 46、47 行

PUBLIC int waiting = 0; //等候理发的顾客人数

PUBLIC int CHAIR = 3; //为顾客准备的椅子数 可在检查时设定

PUBLIC struct semaphore customers,barbers,mutex;

在 main.c 的 kernnel_main 中为

mutex.value = 1;

waiting = 0;

CHAIR = 2;

初始化

testB: 理发师

```
void TestB(){
printf("\n");
while (1){
p(&customers); //判断是否有顾客，若无顾客，理发师睡眠
p(&mutex); //若有顾客，进入临界区
waiting--; //等候顾客数减 1
v(&barbers); //理发师准备为顾客理发
```

v(&mutex); //退出临界区

printf("barber cutting!\n"); //理发师开始理发（消耗 2 个时间片 ticks）10ms 一次

//时钟中断

goToSleep(20);

}

}

顾客进程：以 TestC 为例：

customer1:

void TestC(){

printf("customer1 come!\n");

p(&mutex); //进入临界区

if (waiting < CHAIR) { //判断是否有空椅子

printf("customer1 wait!\n");

waiting++; //等候顾客数加 1

v(&customers); //唤醒理发师

v(&mutex); //退出临界区

p(&barbers); //理发师忙，顾客坐着等待

printf("customer1 get service!\n"); //否则顾客可以理发

}else{

v(&mutex);

}

printf("customer1 leaves\n");

while (1){};

}

不同进程变色：

console.c 第 73 行

sys_printx 会调用 out_char 输出在 console.c 中添加

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;

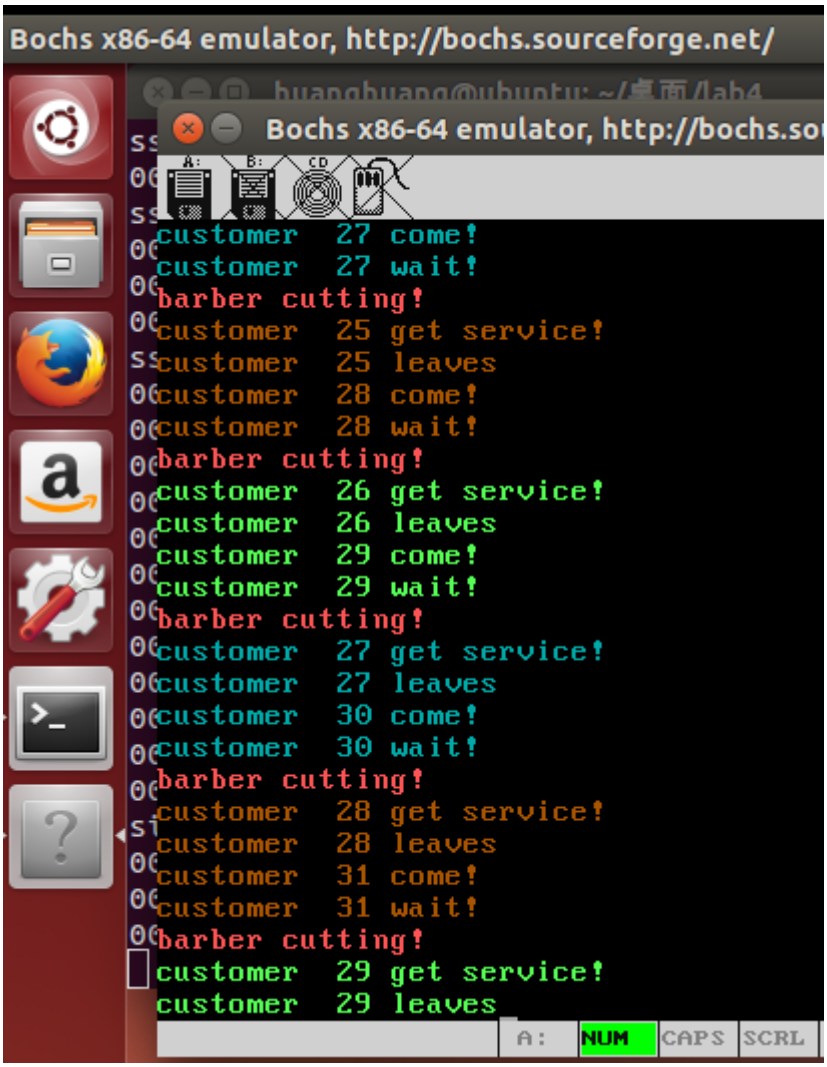
break;

case 6:

ch_color = 0X06;

break;

}


Bochs x86-64 emulator, http://bochs.sourceforge.net/
Bochs x86-64 emulator, http://bochs.so
customer 27 come!
customer 27 wait!
barber cutting!
customer 25 get service!
customer 25 leaves
customer 28 come!
customer 28 wait!
barber cutting!
customer 26 get service!
customer 26 leaves
customer 29 come!
customer 29 wait!
barber cutting!
customer 27 get service!
customer 27 leaves
customer 30 come!
customer 30 wait!
barber cutting!
customer 28 get service!
customer 28 leaves
customer 31 come!
customer 31 wait!
barber cutting!
customer 29 get service!
customer 29 leaves
A:
NUM
CAPS
SCRL