

## lab4 记录

5-29 阅读 Orange's Chapter6

一小時后卒。

5-30 阅读 Orange's Chapter6

## 6.3 节

运行代码 b，效果如图 1 所示。对比 Orange's 中的运行结果，发现 `^` 普遍要少，思考一下，应该是时钟频率较高，所以中断执行时间较短，因此打印出的 `^` 也就少一些了。具体参见后文。

[illegible]

图 1 时钟中断模块

运行代码 e，结果如图 2 所示。可以发现每次打印的 `#` 次数并不相同，而且 A、B 进程也不是严格意义上的交替执行。

Bochs x86-64 emulator, http://bochs.sourceforge.net/

USER Copy Paste Snapshot T Reset suspend/Power

Booting ....., Ready.

Loading ....., Ready.

RAM size: 01FF0000h

-----"cstart" begins-----

-----"cstart" finished-----

-----"kernel main" begins-----

#A0x0. #B0x1000. #A0x1. #B0x1001. ##B0x1002. #A0x2. ##A0x3. #B0x1003. #A0x4. #B0x1004. ##B0x1005. #A0x5. ##A0x6. #B0x1006. ##B0x1007. #A0x7. #B0x1008. #A0x8. ##A0x9. #B0x1009. ##B0x100A. #A0A. ##A0xB. #B0x100B. #A0xC. #B0x100C. ##B0x100D. #A0xD. ##A0xE. #B0x100E. ##B0x100F. #A0xF. #B0x1010. #A0x10. ##A0x11. #B0x1011. ##B0x1012. #A0x12. #B0x1013. #A0x13. ##A0x14. #B0x1014. ##B0x1015. #A0x15. #A0x16. #B0x1016. #A0x17. #B0x1017. ##B0x1018. #A0x18. ##A0x19. #B0x1019. ##B0x101A. #A0x1A. #B0x101B. #A0x1B. ##A0x1C. #B0x101C. ##B0x101D. #A0x1D. ##A0x1E. #B0x101E. #A0x1F. #B0x101F. ##B0x1020. #A0x20. ##A0x21. #B0x1021. #A0x22. ##B0x1023. #A0x24. #B0x1024. ##B0x1025. #A0x26. #B0x1026. ##B0x1027. #A0x28. #B0x1028. ##B0x1029. #A0x2A. #B0x102A. ##B0x102B. #A0x2C. #B0x102C. ##B0x102D. #A0x2E. #B0x102E. ##B0x102F. #A0x30. #B0x1030. ##B0x1031. #A0x32. #B0x1032. ##B0x1033. #A0x34. #B0x1034. ##B0x1035. #A0x36. #B0x1036. ##B0x1037. #A0x38. #B0x1038. ##B0x1039. #A0x3A. #B0x103A. ##B0x103B. #A0x3C. #B0x103C. ##B0x103D. #A0x3E. #B0x103E. ##B0x103F. #A0x40. #B0x1040. ##B0x1041. #A0x42. #B0x1042. ##B0x1043. #A0x44. #B0x1044. ##B0x1045. #A0x46. #B0x1046. ##B0x1047. #A0x48. #B0x1048. ##B0x1049. #A0x4A. #B0x104A. ##B0x104B. #A0x4C. #B0x104C. ##B0x104D. #A0x4E. #B0x104E. ##B0x104F. #A0x50. #B0x1050. ##B0x1051. #A0x52. #B0x1052. ##B0x1053. #A0x54. #B0x1054. ##B0x1055. #A0x56. #B0x1056. ##B0x1057. #A0x58. #B0x1058. ##B0x1059. #A0x5A. #B0x105A. ##B0x105B. #A0x5C. #B0x105C. ##B0x105D. #A0x5E. #B0x105E. ##B0x105F. #A0x60. #B0x1060. ##B0x1061. #A0x62. #B0x1062. ##B0x1063. #A0x64. #B0x1064. ##B0x1065. #A0x66. #B0x1066. ##B0x1067. #A0x68. #B0x1068. ##B0x1069. #A0x6A. #B0x106A. ##B0x106B. #A0x6C. #B0x106C. ##B0x106D. #A0x6E. #B0x106E. ##B0x106F. #A0x70. #B0x1070. ##B0x1071. #A0x72. #B0x1072. ##B0x1073. #A0x74. #B0x1074. ##B0x1075. #A0x76. #B0x1076. ##B0x1077. #A0x78. #B0x1078. ##B0x1079. #A0x7A. #B0x107A. ##B0x107B. #A0x7C. #B0x107C. ##B0x107D. #A0x7E. #B0x107E. ##B0x107F. #A0x80. #B0x1080. ##B0x1081. #A0x82. #B0x1082. ##B0x1083. #A0x84. #B0x1084. ##B0x1085. #A0x86. #B0x1086. ##B0x1087. #A0x88. #B0x1088. ##B0x1089. #A0x8A. #B0x108A. ##B0x108B. #A0x8C. #B0x108C. ##B0x108D. #A0x8E. #B0x108E. ##B0x108F. #A0x90. #B0x1090. ##B0x1091. #A0x92. #B0x1092. ##B0x1093. #A0x94. #B0x1094. ##B0x1095. #A0x96. #B0x1096. ##B0x1097. #A0x98. #B0x1098. ##B0x1099. #A0x9A. #B0x109A. ##B0x109B. #A0x9C. #B0x109C. ##B0x109D. #A0x9E. #B0x109E. ##B0x109F. #A0xA0. #B0x10A0. ##B0x10A1. #A0xA2. #B0x10A2. ##B0x10A3. #A0xA4. #B0x10A4. ##B0x10A5. #A0xA6. #B0x10A6. ##B0x10A7. #A0xA8. #B0x10A8. ##B0x10A9. #A0xAA. #B0x10AA. ##B0x10AB. #A0xAC. #B0x10AC. ##B0x10AD. #A0xAE. #B0x10AE. ##B0x10AF. #A0xB0. #B0x10B0. ##B0x10B1. #A0xB2. #B0x10B2. ##B0x10B3. #A0xB4. #B0x10B4. ##B0x10B5. #A0xB6. #B0x10B6. ##B0x10B7. #A0xB8. #B0x10B8. ##B0x10B9. #A0xBA. #B0x10BA. ##B0x10BB. #A0xBC. #B0x10BC. ##B0x10BD. #A0xBE. #B0x10BE. ##B0x10BF. #A0xC0. #B0x10C0. ##B0x10C1. #A0xC2. #B0x10C2. ##B0x10C3. #A0xC4. #B0x10C4. ##B0x10C5. #A0xC6. #B0x10C6. ##B0x10C7. #A0xC8. #B0x10C8. ##B0x10C9. #A0xCA. #B0x10CA. ##B0x10CB. #A0xCC. #B0x10CC. ##B0x10CD. #A0xCE. #B0x10CE. ##B0x10CF. #A0xD0. #B0x10D0. ##B0x10D1. #A0xD2. #B0x10D2. ##B0x10D3. #A0xD4. #B0x10D4. ##B0x10D5. #A0xD6. #B0x10D6. ##B0x10D7. #A0xD8. #B0x10D8. ##B0x10D9. #A0xDA. #B0x10DA. ##B0x10DB. #A0xDC. #B0x10DC. ##B0x10DD. #A0xDE. #B0x10DE. ##B0x10DF. #A0xE0. #B0x10E0. ##B0x10E1. #A0xE2. #B0x10E2. ##B0x10E3. #A0xE4. #B0x10E4. ##B0x10E5. #A0xE6. #B0x10E6. ##B0x10E7. #A0xE8. #B0x10E8. ##B0x10E9. #A0xEA. #B0x10EA. ##B0x10EB. #A0xEC. #B0x10EC. ##B0x10ED. #A0xEE. #B0x10EE. ##B0x10EF. #A0xF0. #B0x10F0. ##B0x10F1. #A0xF2. #B0x10F2. ##B0x10F3. #A0xF4. #B0x10F4. ##B0x10F5. #A0xF6. #B0x10F6. ##B0x10F7. #A0xF8. #B0x10F8. ##B0x10F9. #A0xFA. #B0x10FA. ##B0x10FB. #A0xFC. #B0x10FC. ##B0x10FD. #A0xFE. #B0x10FE. ##B0x10FF. #A0x00. #B0x1000. ##B0x1001. #A0x02. #B0x1002. ##B0x1003. #A0x04. #B0x1004. ##B0x1005. #A0x06. #B0x1006. ##B0x1007. #A0x08. #B0x1008. ##B0x1009. #A0x0A. #B0x100A. ##B0x100B. #A0x0C. #B0x100C. ##B0x100D. #A0x0E. #B0x100E. ##B0x100F. #A0x10. #B0x1010. ##B0x1011. #A0x12. #B0x1012. ##B0x1013. #A0x14. #B0x1014. ##B0x1015. #A0x16. #B0x1016. ##B0x1017. #A0x18. #B0x1018. ##B0x1019. #A0x1A. #B0x101A. ##B0x101B. #A0x1C. #B0x101C. ##B0x101D. #A0x1E. #B0x101E. ##B0x101F. #A0x20. #B0x1020. ##B0x1021. #A0x22. #B0x1022. ##B0x1023. #A0x24. #B0x1024. ##B0x1025. #A0x26. #B0x1026. ##B0x1027. #A0x28. #B0x1028. ##B0x1029. #A0x2A. #B0x102A. ##B0x102B. #A0x2C. #B0

图 2 实现多进程

## 6.5 节

Orange's 中 Intel8253 的输入频率为 1.2MHz，查看了下自己电脑的 CPU 频率，2.00GHz。差了 3 个数量级，显然 8253 的输入频率并不是 CPU 频率。

用时四小时，总算是艰难地看完了一遍。

### 5-31 添加进程

参考 6.4 节 多进程，在 Chapter6 代码 r 的基础上，添加 TestD、TestE 两个进程，并没有打印出 D 和 E。参考 6.6 节 系统调度，为进程 D 和 E 设置优先级，并将所有进程的优先级均设为为 3，运行结果如图 3 所示。



图 3 添加进程 D、E

### 5-31 添加系统调用 sys\_disp\_str

写 lab2 时，要不断地翻前翻后找不同的章节，于是顺手给全书做了份目录。现在到时派上了用场。一眼看到 7.5 节 printf，粗略看了下，发现表 7.6 增加一个系统调用的过程比 6.5 节要详细不少，鉴于 sys\_disp\_str 比其他系统调用要简单一些，先用它练练手。

因为 Orange's 中已经有 disp\_str 函数了，换个名字，叫 disp\_str\_1。disp\_str\_1 的函数体参照代码 7.58，由于只需要传递一个参数，所以去掉对 ecx 的赋值。sys\_disp\_str 需要一个参数，而已有的 sys\_get\_ticks 不需要，所以参考代码 7.60，修改 sys\_call()。这里只需要一个参数，所以无需将 ecx 压栈，相应的，esp 只需回退 8 个字节。

```

; =====
;                                     sys_call
; =====
sys_call:
    call    save
    push    dword [p_proc_ready]
    sti

    push    ebx
    call    [sys_call_table + eax * 4]
    add     esp, 4 * 2

    mov     [esi + EAXREG - P_STACKBASE], eax
    cli
    ret
```

添加 sys\_disp\_str 的函数体如下：

```

/*=====
sys_disp_str
=====*/
PUBLIC void sys_disp_str(char* str)
{
    disp_str(str);
}

```

把 main.c 中调用 disp\_str 的地方全部换成 disp\_str\_1，运行一下。效果如图 4 所示，输出和图 3 近似相同。至此，第一个系统调用已经完成。感觉不错，明天继续努力。

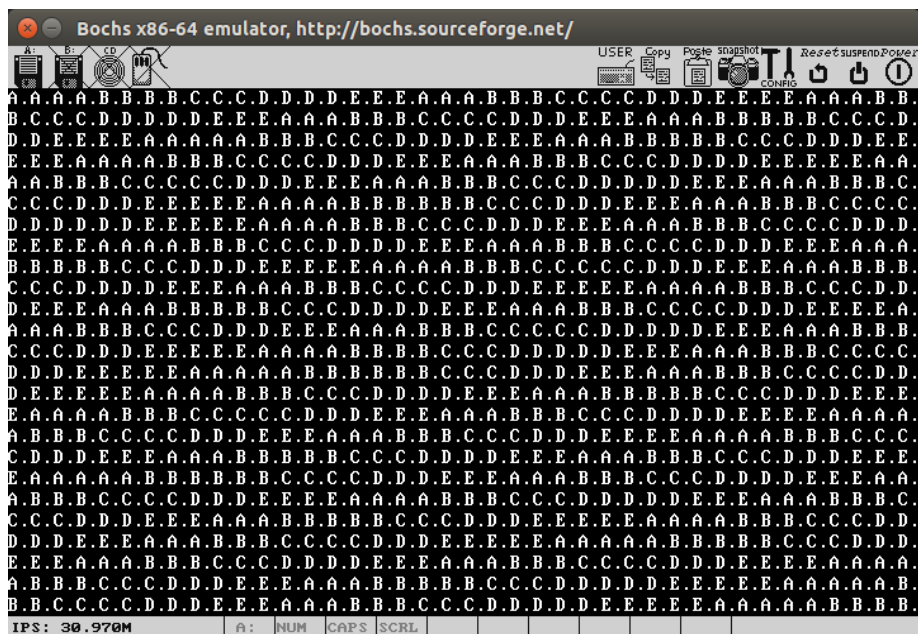


图 4 添加系统调用 sys\_disp\_str

## 6-2

作业要求使用不同颜色打印不同进程，所以再添加一个系统调用 sys\_disp\_color\_str，这个调用需要用到两个参数，所以又把 sys\_call 改了回来。。。

依葫芦画瓢，写完 sys\_disp\_color\_str，把打印函数换为 disp\_color\_str\_1，运行发现可以打印输出但颜色没有任何变化。检查了下代码，发现 disp\_color\_str\_1 中忘了给 ecx 赋值。加入赋值语句之后再运行，结果如图 5 所示。

接着添加 sys\_process\_sleep，在进程表中添加 int 变量 sleep，记录进程需要睡眠的时间。添加函数 milli\_delay\_1，以间接调用 sys\_process\_sleep，并在其中设置其 sleep 时间，函数体如下：

```

/*=====
milli_delay_1
=====*/
PUBLIC void milli_delay_1(int milli_sec)
{
    process_sleep(milli_sec);
    while (p_proc_ready->sleep) {}
}

/*=====
sys_process_sleep
=====*/
PUBLIC void sys_process_sleep(int mill_seconds)
{
    p_proc_ready->sleep = mill_seconds * HZ / 1000;
}

```

为了便于观察，在 TestA 中调用 milli\_delay\_1，其余均调用 milli\_delay，时间均为 1000。同时将所有进程优先级均设为 1。现在运行可以发现，A 打印一次后，就不再打印。证明我们已经可以做到让进程 ‘sleep’了，接下来唤醒进程。在 clock\_handler 中添加：

```
for (int i = 0; i < NR_TASKS; i++) {
    if (proc_table[i].sleep != 0) {
        proc_table[i].sleep--;
    }
}
```

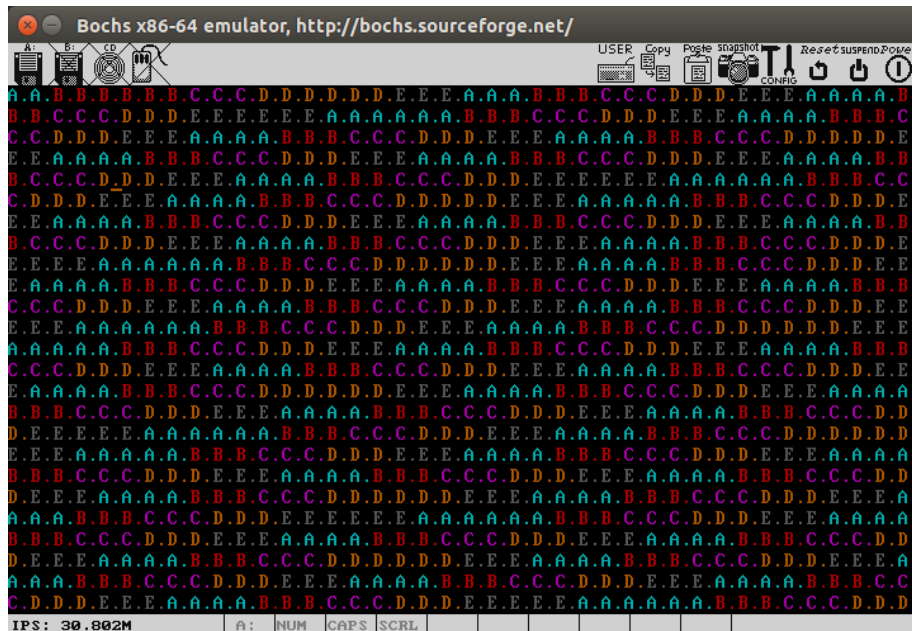


图 5 添加系统调用 sys\_disp\_color\_str

现在运行，结果与图 5 基本相同，但是这个操作其实还是为进程 A 分配时间片的，接下来取消分配。在函数 schedule 中，添加判断条件，当目标进程不处于睡眠状态时，切换进程。

```
if (p->ticks > greatest_ticks && !p->sleep) {
    greatest_ticks = p->ticks;
    p_proc_ready = p;
}
```

接下来验证一下以上操作确实可以做到不分配时间片。在 milli\_delay\_1 方法中添加一条打印语句，为了防止打印内容过多而撑爆显存，打印之前稍作延迟，代码如下：

```
/*=====
                                milli_delay_1
=====*/
PUBLIC void milli_delay_1(int milli_sec)
{
    process_sleep(milli_sec);
    while (p_proc_ready->sleep) {
        milli_delay(10);
        disp_str("1");
    }
}
```

图 6 和图 7 分别对应分配时间片和不分配时间片两种情况，可以很明显地看到两种情况下的差异，证明不分配时间片的操作是成功的。至此，系统调用 sys\_process\_sleep 成功完成。

其实添加 sleep 变量后，优先级完全可以去除了，调度时无需判断优先级大小，只需轮询所有进程就可以了。但现在功能已经可以实现，也就懒得改了。



图 6 分配时间片



图 7 不分配时间片

## 6-4 添加 PV 操作

先在 proc.h 中声明一个结构体表征信号量，具体参考教材 3.3.2 节。list 为进程数组，记录等待队列，这里暂时设长度为 10；head 和 tail 分别用来记录队列首部和尾部。

```
typedef struct semaphore {
    int value;
    PROCESS* list[10];
    int head, tail;
} SEMAPHORE;
```

接着在 proto.h 中声明 sys\_sem\_p、sys\_sem\_v、sem\_p 和 sem\_v，传入参数均为一个信号量，编译，果断报错，提示找不到信号量的定义。将函数的声明挪到 proto.h 中，问题解决。这里本来想写一个 sem.c 文件来存放关于信号量的操作，修改了下 Makefile，一堆报错，还是写在 proc.c 中了，虽然结构不够紧凑，但好在不会报错。

```
PUBLIC void sys_sem_p(SEMAPHORE*);
PUBLIC void sys_sem_v(SEMAPHORE*);
PUBLIC void sem_p(SEMAPHORE*);
PUBLIC void sem_v(SEMAPHORE*);
```

一开始我使用的是引用传递，略微感受了下，在 kernel\_main() 中初始化信号量 customers.value=0，调用方法 sem\_p(customers)，在 sys\_sem\_p 中简单打印 customers.value，结果不是 0，而是一个很怪异的数。分析一下可以发现，若使用引用传递，则在使用系统调用时，需要将整个 customers 压栈，而实际操作中却按照一个参数的情形，简单地进行一次压栈，导致了错误的产生。简便起见，将参数类型改为信号量指针，这样只需要进行一次压栈就可以解决问题了。再次打印一下 customers.value，这次显示正常。

P 操作和 V 操作，参考教材中的代码。这里使用循环数组配合属性 head 和 tail 来实现进程队列。

```
/*=====
                        sys_sem_p
=====*/
PUBLIC void sys_sem_p(SEMAPHORE* s)
{
    s->value--;
    if (s->value < 0) {
        s->list[s->head] = p_proc_ready;
        milli_delay_1(1000);
        if (s->head != s->tail) {
            s->head = (s->head + 1) % 10;
        }
    }
}
```

```

/*=====
                                sys_sem_v
=====*/
PUBLIC void sys_sem_v(SEMAPHORE* s)
{
    s->value++;
    if (s->value <= 0) {
        s->list[s->tail]->sleep = 0;
        s->tail = (s->tail + 1) % 10;
    }
}

```

写到这里，本来想验证以下PV操作的正确性，但奈何没有想出靠谱的测试用例，只能作罢。这里应该显示出程序员天生的乐观精神，相信自己的代码是对的。

## 6-5 尝试睡眠解决理发师问题

完全参照教材上的代码写了一边，但是只有第一遍输出勉强能看，以后的输出就完全乱套了。仔细检查发现P操作写错了，在P操作中的sleep()应该是永久睡眠，也就是如果不被其它进程唤醒的话，就会一直睡下去。发现问题，修改P操作，设置进程睡眠时间为-10，即将进程的sleep设为-1。

```

/*=====
                                sys_sem_p
=====*/
PUBLIC void sys_sem_p(SEMAPHORE* s)
{
    s->value--;
    if (s->value < 0) {
        s->list[s->head] = p_proc_ready;
        milli_delay_1(-10);
        if (s->head != s->tail) {
            s->head = (s->head + 1) % 10;
        }
    }
}

```

相应的，在每次时钟中断发生时，只将sleep大于0的进程减少睡眠时间就可以了。

```

if (proc_table[i].sleep > 0) {
    proc_table[i].sleep--;
}

```

但是，出现很严重的bug，已经好久没见过的Exception又重新出现了，找了一下午的bug，一点进展也没有，暂时放弃。

## 6-7 重新来过

先前没有验证PV操作的正确性，有可能是导致bug的主要原因。仔细阅读教材3.3节后，验证PV操作的正确性。为了降低复杂度，将进程数设为3，A进程为普通进程，B进程为理发师进程，C进程为顾客进程。

```

/*=====
                                TestB (理发师进程)
=====*/
void TestB()
{
    while(1){
        disp_str("1");
        sem_p(&customers);           /*判断是否有顾客，若无顾客，理发师睡眠*/
        disp_str("2");
        // sem_p(&mutex);             /*若有顾客，进入临界区*/
        // waiting--;                /*等待顾客数减1*/
        // sem_v(&mutex);             /*退出临界区*/
        // sem_v(&barbers);           /*理发师准备为顾客理发*/
    }
}

```

```

/*=====
                                TestC (顾客进程)
=====*/
void TestC()
{
    while(1) {
        disp_color_str_1("C.", 0x05);
        milli_delay_1(1000);
    }
}

```

现在正确结果应该是只打印“1”、不打印“2”，然后持续打印“C”，然而实际运行结果却是只能打印一次“1”，然后就没反应了。从P操作开始，一步一步跟踪打印发现，问题出现在了中断重入这里。

```

if (k_reenter != 0) {
    return;
}

```

翻了翻 Orange's 6.4.7 和 6.4.8 节，发现似乎没有什么可以改动的地方 ☹️。

P 操作的作用是使进程睡眠，直到被其它进程唤醒，而在睡眠过程中，本次中断处理过程是没有结束的，直到被唤醒才算结束处理过程。因此以后发生的时钟中断均被视为中断重入，也就无法被调度程序。简单起见，直接注释掉以上中断重入的代码，再次运行，结果如图 8 所示，和预期效果一样。

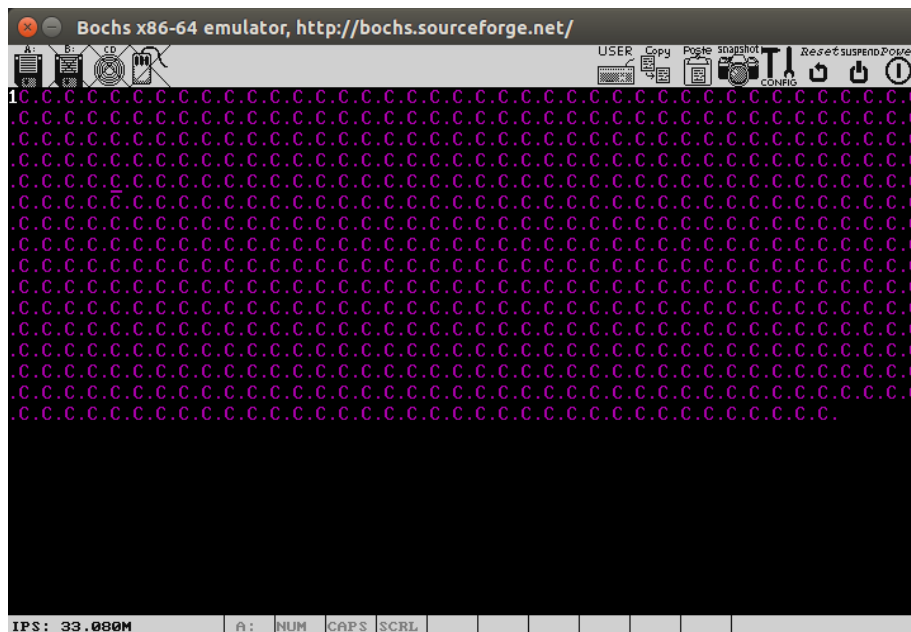


图 8 测试 P 操作

查看 V 操作时发现一个明显的 bug，源代码中 V 操作调用 wakeup 方法，其中调用了新添加的系统调用 process\_wakeup，这里错误地模仿了 sys\_process\_sleep。当前运行进程的 sleep 必然为 0，而要唤醒的进程却没有指定。

```

/*=====
                                sys_process_wakeup
=====*/
PUBLIC void sys_process_wakeup()
{
    p_proc_ready->sleep = 0;
}

```

为了标识目标进程，可以采用两种方式。可以传递进程指针，也可以传递进程的标识---进程数组中的下标。二者在参数传递时消耗的堆栈是相同的，但信号量中持有的变量是进程指针，所以这里采用传递进程指针的方案。



为上述系统调用添加一个参数 PROCESS\*，记录要唤醒的目标进程。这里存在一个潜在的问题，在唤醒进程时，是否要切换进程。感觉切换和不切换都有一定的道理，纠结了一下，还是选择了不切换进程。一来在唤醒进程的系统调用中切换进程，显得有点越俎代庖；二来在系统调用中切换进程不晓得会不会出现什么传奇的 bug。这里因为用到了在 proc.h 中定义的 PROCESS，所以将 sys\_process\_wakeup 和 process\_wakeup 的声明放在了 proc.h 中。

```
/*=====*\n\n                        sys_process_wakeup\n\n*=====*/\nPUBLIC void sys_process_wakeup(PROCESS* p)\n{\n    p->sleep = 0;\n}
```

修改后再次运行，结果如图 9 所示，可以看到当只有 B、C 两个进程唱二人转时，效果还是可以的，没有什么错误。

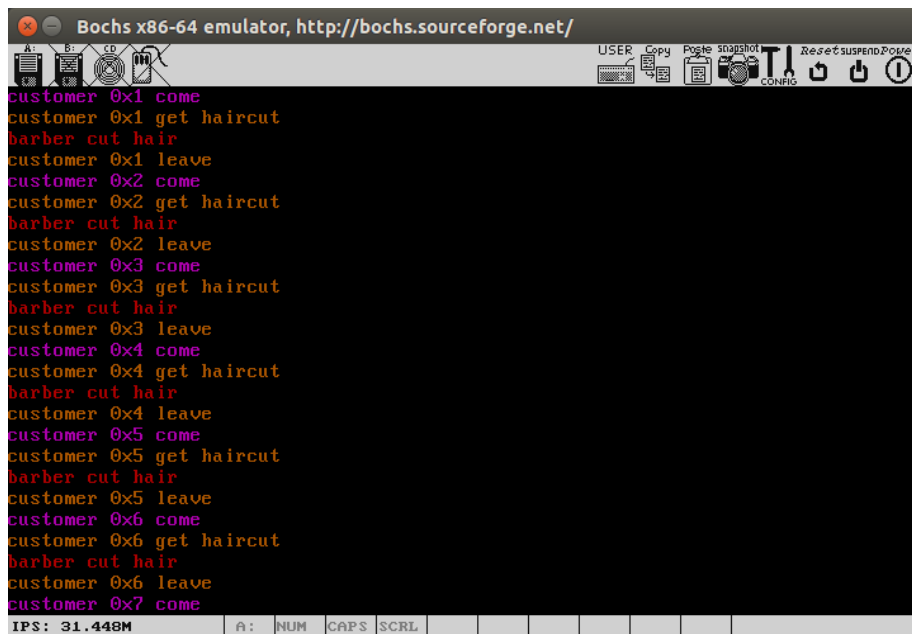


图 9 B、C 进程二人转

接下来，加入进程 D、E 看看效果如何。无法打印 ‘customer get haircut’，即顾客无法得到服务。为了降低复杂度，去掉进程 E，出现类似 bug。问题根源在于理发师睡眠后，无法被唤醒。再次认真阅读教材。

无意间发现了一个以前写的小 bug，教材中 P 操作调用 sleep(s.list)，会转向进程调度程序，而我实现时，让执行 P 操作的进程进入死循环，直到下一次时钟中断产生。将 milli\_delay\_1 方法修改如下：

```
/*=====*\n\n                        milli_delay_1\n\n*=====*/\nPUBLIC void milli_delay_1(int milli_sec)\n{\n    process_sleep(milli_sec);\n    // while (p_proc_ready->sleep) {}\n    schedule();\n}
```



## 6-10 找 bug

bug 依然没有搞定，每次打印到第四个顾客时就卡住不动了，专业点讲就是发生了死锁。很自然地想起有可能是椅子数量的问题。接下来把椅子数量分别改为 1 和 5，然而没有任何变化。看来应该是将进程放进等待队列然后取出的操作出了问题。虽然非常好奇为什么偏偏停在了第四个，但依然没有任何进展。经历了近一小时的疯狂打印后，终于找到了问题所在。

在 P、V 操作中，移动 tail 和 head 指针时需要这样，这也合理，因为进程队列的长度为 10，当然我一直是这样认为的，然而在定义结构体 SEMAPHORE 时，进程队列的长度居然是 3！！

```
s->head = (s->head + 1) % 10;  
s->tail = (s->tail + 1) % 10;
```

这样为什么每次打印都会停在第四个也就不足为奇了，经过这次教训，赶紧乖乖地把进程队列的长度表示成宏定义，避免出错。

现在来运行，虽然顺序有点乱，但好在是可以正常跑了，结果如图 10 所示。

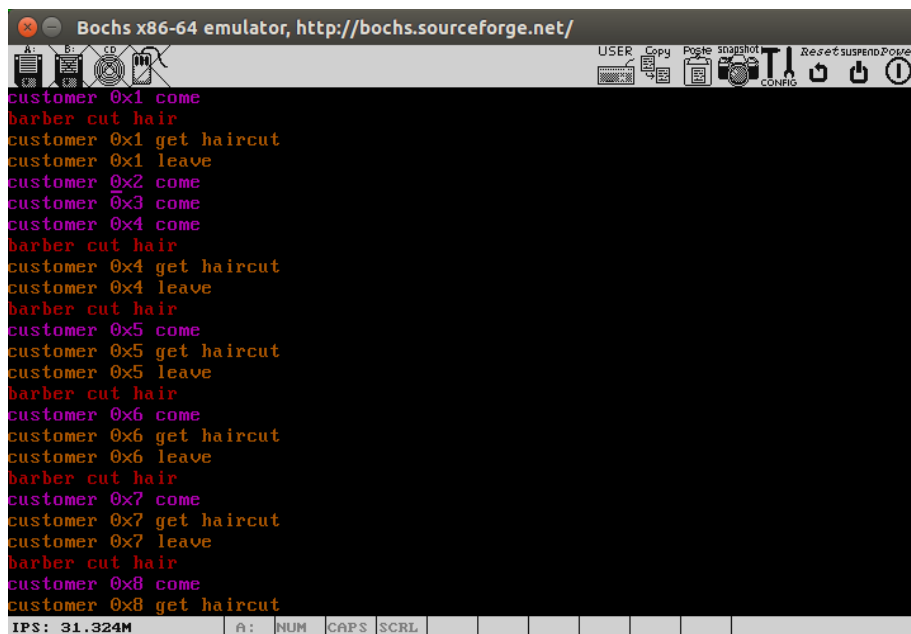


图 10 顺序有点乱的理发师

## 6-11 调理

现在程序运行顺序有点混乱，是同步出了问题。再次参考教材，发现所有步骤都是和教材一样的。想来应该不是进程逻辑错误。仔细分析图 10，发现一个问题，理发师理发只是打印 barber cut hair，却不知道究竟是给哪个顾客理发，所以在理发操作中加入顾客编号。打印后发现，第二次理发其实是 2 号顾客理发，而顾客进程中的编号却已经超前了 3 个，也就是椅子的数量。改变椅子数量，证明猜想正确。

这也容易理解，程序一直按照 B、C、D、E 的顺序执行调度。理发师好不容易结束一次理发，而顾客却已经来了好几个，顾客编号也增加了很多。为此，在每个顾客进程中添加一个变量 temp，用来记录顾客的编号，而全局变量 number 则用于记录所有顾客的数量。

接下来整理一下代码，把打印操作的代码全部写成函数，一共四个操作，come、getHaircut、leave 和 full，分别对应于顾客到来、顾客得到服务、顾客离开、人满了顾客离开。

```
void come(int number)  
{  
    disp_color_int(number, 0x05);  
    disp_color_str_1(" come and wait\n", 0x05);  
}
```

```

    milli_delay(1000);
}

void getHaircut(int number)
{
    disp_color_int(number, 0x06);
    disp_color_str_1(" get haircut\n", 0x06);
    milli_delay(2000);
}

void leave(int number)
{
    disp_color_int(number, 0x06);
    disp_color_str_1(" leave\n", 0x06);
    milli_delay(1000);
}

void full()
{
    milli_delay(1000);
    disp_color_str_1("full, leave\n", 0x09);
}

```

把顾客进程写成单独的函数 customer，在每个进程中调用即可。

```

void customer()
{
    int temp;
    while(1) {
        sem_p(&mutex);          /*进入临界区*/
        if (waiting < CHAIRS) {  /*判断是否有空椅子*/
            waiting++;          /*等待顾客加1*/
            number++;           /*顾客编号加1*/
            temp = number;
            come(temp);
            sem_v(&customers);   /*唤醒理发师*/
            sem_v(&mutex);       /*退出临界区*/
            sem_p(&barbers);     /*理发师忙，顾客坐着等待*/
            sem_p(&mutex);
            getHaircut(temp);
            leave(temp);
            sem_v(&mutex);
        } else {
            sem_v(&mutex);       /*人满了，顾客离开*/
            full();
        }
    }
}

```

至此，代码全部结束，在结尾贴出运行结果图。

```

Bochs x86-64 emulator, http://bochs.sourceforge.net/
0x1 come and wait
0x2 come and wait
0x3 come and wait
cut hair
0x1 get haircut
0x1 leave
cut hair
0x4 come and wait
0x2 get haircut
0x2 leave
cut hair
0x5 come and wait
0x3 get haircut
0x3 leave
cut hair
0x6 come and wait
0x4 get haircut
0x4 leave
cut hair
0x7 come and wait
0x5 get haircut
0x5 leave
cut hair
0x8 come and wait
0x6 get haircut
IPS: 31.508M
A: NUM CAPS SCRL

```