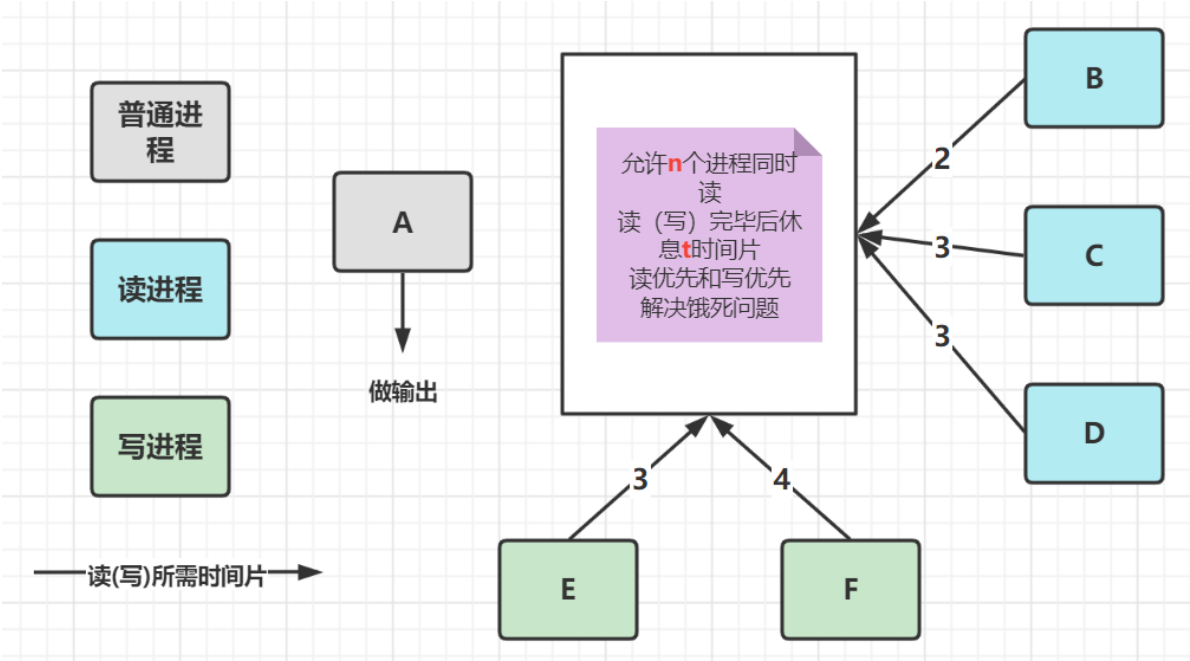
Lab4

1 实验要求



2 源码理解

2.1 运行源码

`make run` 之后报错

```
gcc -I include/ -c -fno-builtin -o kernel/proc.o kernel/proc.c
nasm -I include/ -f elf -o lib/kliba.o lib/kliba.asm
gcc -I include/ -c -fno-builtin -o lib/klib.o lib/klib.c
nasm -I include/ -f elf -o lib/string.o lib/string.asm
ld -s -Ttext 0x30400 -o kernel.bin kernel/kernel.o kernel/syscall.o kernel/start
.o kernel/main.o kernel/clock.o kernel/i8259.o kernel/global.o kernel/protect.o
kernel/proc.o lib/kliba.o lib/klib.o lib/string.o
kernel/protect.o: In function `exception_handler':
protect.c:(.text+0x64f): undefined reference to `__stack_chk_fail'
lib/klib.o: In function `disp_int':
klib.c:(.text+0xe8): undefined reference to `__stack_chk_fail'
Makefile:72: recipe for target 'kernel.bin' failed
make: *** [kernel.bin] Error 1
zhhc@zhhc-VirtualBox:~/OS/lab4/r$
```

和Lab3一样

zhhc@zhhc-VirtualBox: ~/OS/lab4/r

```
 1 ##########################
 2 # Makefile for Orange'S #
 3 ##########################
 4
 5 # Entry point of Orange'S
 6 # It must have the same value with 'KernelEntryPointPhyAddr' in load.inc!
 7 ENTRYPOINT  = 0x30400
 8
 9 # Offset of entry point in kernel file
10 # It depends on ENTRYPOINT
11 ENTRYOFFSET =   0x400
12
13 # Programs, flags, etc.
14 ASM      = nasm
15 DASM         = ndisasm
16 CC       = gcc
17 LD       = ld
18 ASMBFLAGS    = -I boot/include/
19 ASMKFLAGS    = -I include/ -f elf
20 CFLAGS       = -I include/ -c -fno-builtin -fno-stack-protector
21 LDFLAGS      = -s -Ttext $(ENTRYPOINT)
22 DASMFLAGS    = -u -o $(ENTRYPOINT) -e $(ENTRYOFFSET)
23
"Makefile" [dos] 113L, 3402C
```

2.2 进程调度

在 `kernel_main` 函数中，为三个任务分配时间片和优先级如下

```
proc_table[0].ticks = proc_table[0].priority = 15;
proc_table[1].ticks = proc_table[1].priority =  5;
proc_table[2].ticks = proc_table[2].priority =  3;
```

书中说是时钟中断每隔10ms发生一次，时钟中断处理程序如下

```
PUBLIC void clock_handler(int irq)
{
    ticks++; // 时钟中断次数+1
    p_proc_ready->ticks--; // 该进程剩余时间片-1

    if (k_reenter != 0) { // 上一个中断未处理完成之前又发生了一次中断
        return;
    }

    if (p_proc_ready->ticks > 0) { // 如果当前进程还有剩余时间片，则继续执行
        return;
    }

    schedule(); // 调度进程

}
```

进程调度函数 `schedule`

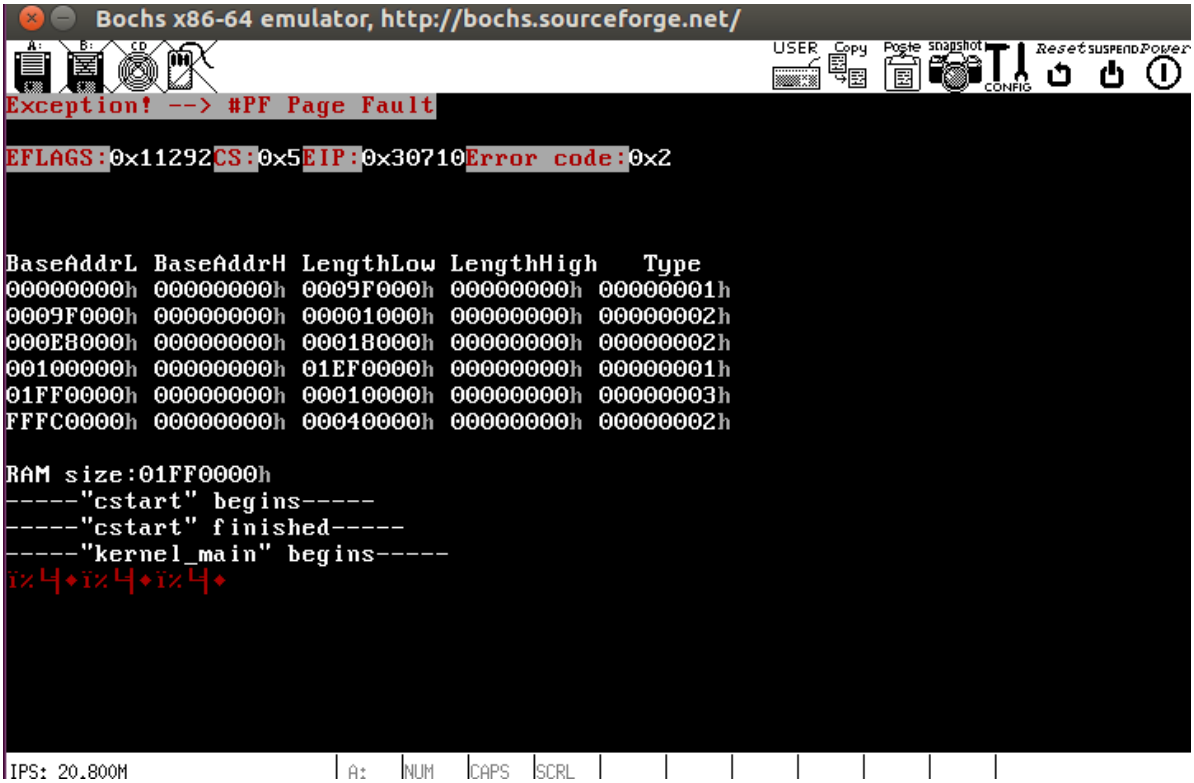
调度的逻辑就是：**找到剩余时间片最多（也就是优先级最高）的那个进程**

```
PUBLIC void schedule()
{
    PROCESS* p;
    int  greatest_ticks = 0;

    while (!greatest_ticks) {
        // 找到ticks最大的作为下一个运行进程
        for (p = proc_table; p < proc_table+NR_TASKS; p++) {
            if (p->ticks > greatest_ticks) {
                greatest_ticks = p->ticks;
                p_proc_ready = p;
            }
        }

        if (!greatest_ticks) { // 如果p->ticks全部都是0, 就将优先级priority作为
        ticks
            for (p = proc_table; p < proc_table+NR_TASKS; p++) {
                p->ticks = p->priority;
            }
        }
    }
}
```

3 实现过程

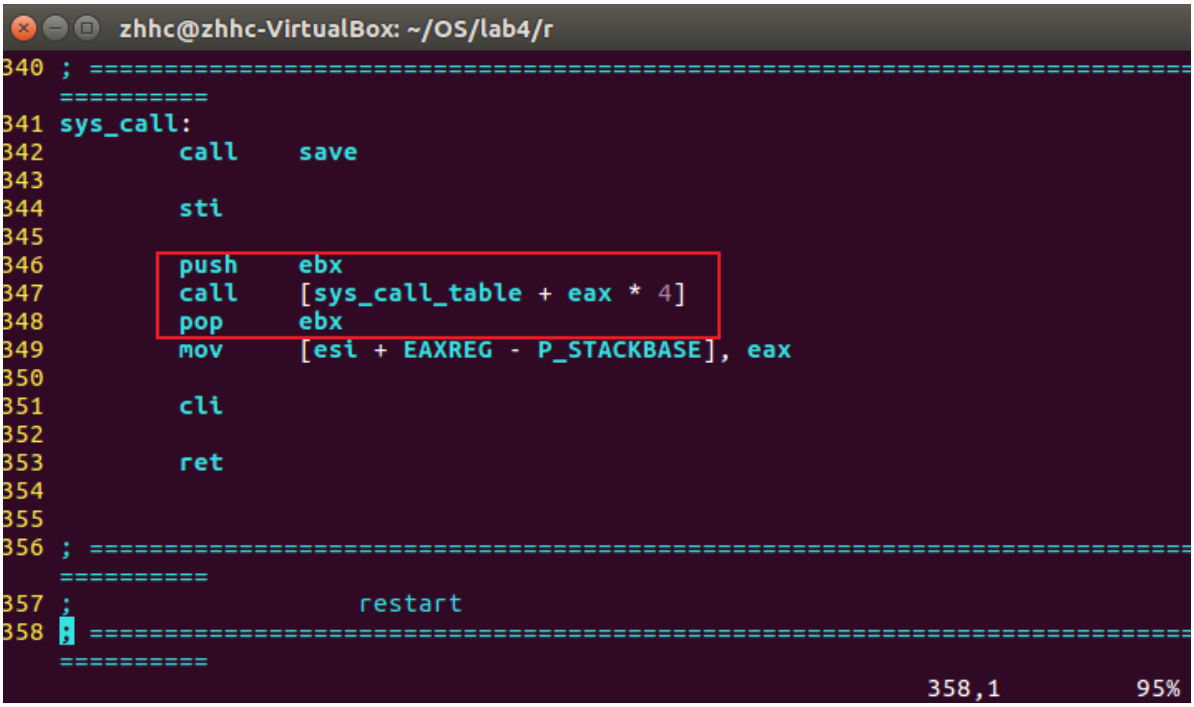
添加一个系统调用：print_str(char* s)

实现完毕后出现bug：



原因是：该系统调用需要传递参数，但是由于源代码中的`sys_call`不支持传递参数，所以报错

解决方法：修改`sys_call`的汇编代码，将`ebx`中内容压入栈中，然后再进行系统调用，这样只要将参数存放在`ebx`中即可。（后续如果需要更多参数，就对应压入更多的寄存器中的值）



添加一个系统调用：sleep(int milli_seconds)

该系统调用实现的是在指定的`milli_seconds`中不被分配时间片。

实现方式，手册中给了提示

1、修改`PROCESS`结构体，增加一个`wake_tick`字段，指示该进程醒来的时间片。

2、修改`schedule`函数，在进程调度时需要增加对`wake_tick`的判断，可以参与进程调度的应该是`wake_tick<=current_tick`的进程。

模拟读者写者

> **注意点**
>
> 一个读进程被选中开始读一个时间片之后，另一个读进程被调入开始读的这一个时间中，前一个读进程应该仍然保持读状态，直至读结束。

> 需要实现三种策略：读者优先、写者优先、读写公平（防止饿死）
>
> 允许同时读的进程数需要能够改变（`n=1,2,3`）
>
> 每个进程读写结束之后休息的时间片可以随意修改($t \geq 0$)
>
> （具体的PV操作以及三种策略对应的读写函数见代码）

1、**三种策略**

使用表驱动的实现方式，建立函数数组，依据不同的策略去调用数组中对应的读写函数即可。

```
1  // type.h
2  typedef void (* reader_func)(); // 读者函数
3  typedef void (* writer_func)(); // 写者函数
```

```
4
5  // const.h
6  #define STRATEGY    3
7  // 读写策略
8  #define READER_FIRST 0
9  #define WRITER_FIRST 1
10 #define RW_EQUALITY  2
11
12 // global.c
13 PUBLIC reader_func readers[STRATEGY] = {
14     reader_first_r,
15     writer_first_r,
16     rw_equality_r
17 };
18
19 PUBLIC writer_func writers[STRATEGY] = {
20     reader_first_w,
21     writer_first_w,
22     rw_equality_w
23 };
24
25 // main.c
26 strategy = READER_FIRST;
27
28 void ReadB(){
29     readers[strategy]();
30 }
31
32 void WriteE(){
33     writers[strategy]();
34 }
```

2、同时读的进程数

使用一个信号量控制读者进程数量`r_mutex`，其`value`初值为`READER_MAX`

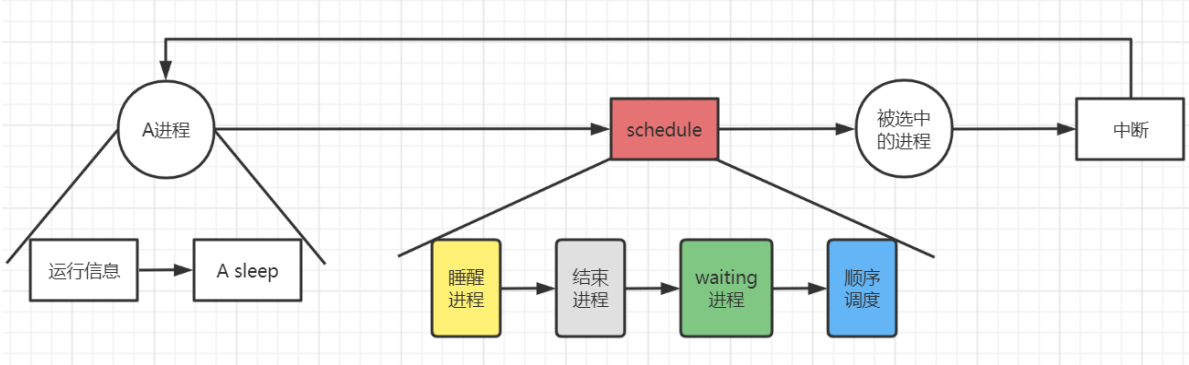
3、读写之后的休息时间

在`PROCESS`结构体中增加一个`sleep_time`字段，在完成读写后调用`sleep`函数即可

4、`PROCESS`结构体

```c
typedef struct s_proc {
    ...

    int wake_tick;  // 进程醒来的时间片
    int state;  // 进程的状态
    int type; // 进程的类型
    int run_after_sleep; // 判断进程此时是否是醒来立刻运行的
    int sleep_time; // 进程休息的时间片数量
}PROCESS;
```

5、`schedule`函数



> 注意：要调度**所有的**睡醒的进程、调度**所有的**结束进程释放资源、调度**所有的**waiting进程。因为逻辑上他们就是要同步进行的。

```c
PUBLIC void schedule()
{
    PROCESS* p = proc_table;

    if(isRunnable(p)){ // 优先选择A
        p_proc_ready = p;
    } else {
        // 先去找刚睡醒的进程，尝试调度他们
        for (PROCESS* i = proc_table + 1; i < proc_table + NR_TASKS; i++){
            if (i->state == SLEEPING && i->wake_tick <= get_ticks()){
                i->state = RUNNING;
                i->run_after_sleep = 1;
                p_proc_ready = i;
                return;
            }
        }
        // 再找到tick=0的进程 去释放资源
        for (PROCESS* i = proc_table + 1; i < proc_table + NR_TASKS; i++){
            if (i->ticks == 0){
                p_proc_ready = i;
                return;
            }
```

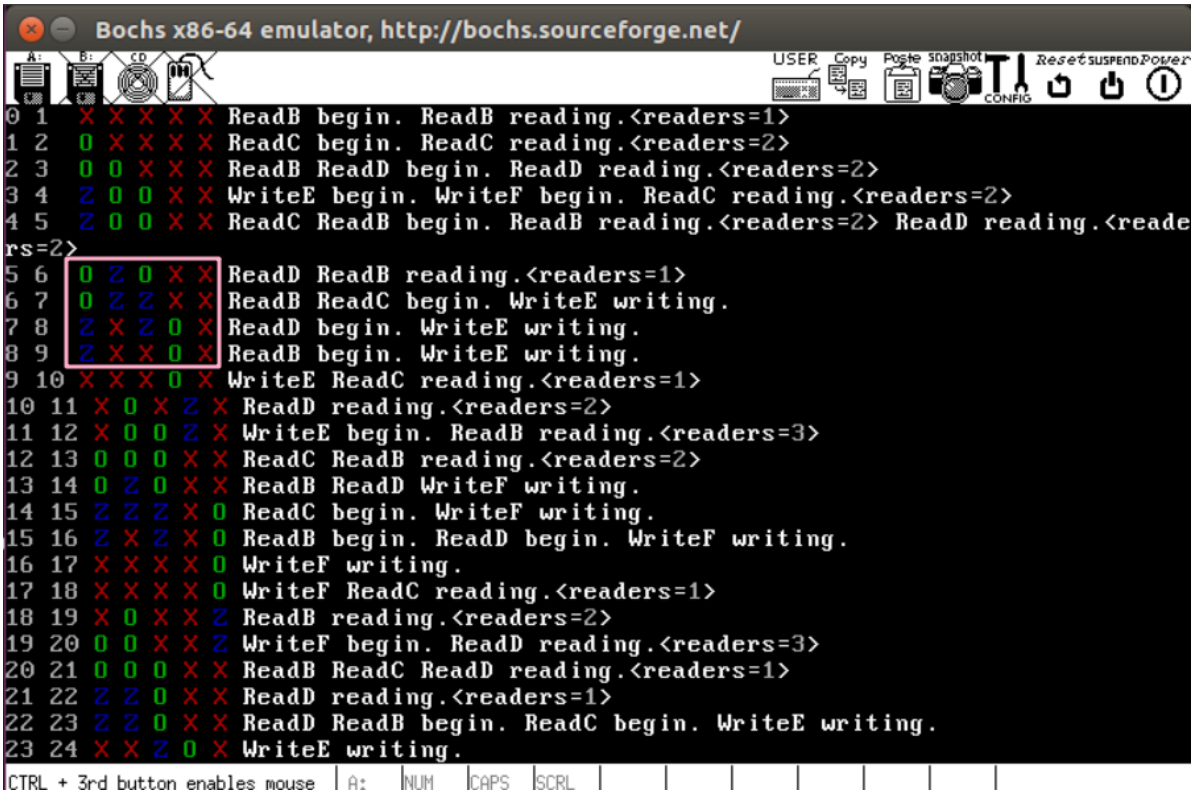
```
        }
        // 调度waiting的进程 不用等信号量的进程
        for (PROCESS* i = proc_table + 1; i < proc_table + NR_TASKS; i++){
            if(i->state == WAITING){
                i->state = RUNNING;
                p_proc_ready = i;
                return;
            }
        }
        // 都不存在按之前的顺序继续调度
        p = prev_proc + 1;
        while(!isRunnable(p)){
            p++;
            if(p >= proc_table + NR_TASKS){
                p = proc_table;
            }
        }
        p_proc_ready = p;
        prev_proc = p;
    }
    p_proc_ready->state = RUNNING;
}
```

> 注意：调度了刚睡醒的进程后，如果其可以成功执行，要在开始执行后停止再次执行调度函数，因为需要将结束的进程的资源返回，同时尝试开启其他进程。（`run_after_sleep`字段就是为了执行这个功能）

6、一个问题

`reader_max=2 && sleep_time=2`会有一个问题：B进程结束开始睡觉，C进程睡醒，D进程在睡觉，此时`rw_mutex`被写者进程抢走。

解决：先调度刚睡醒的进程，然后调度要结束的进程，保证刚睡醒的进程可以和其他进程一样在结束的进程返回资源之后一起竞争。

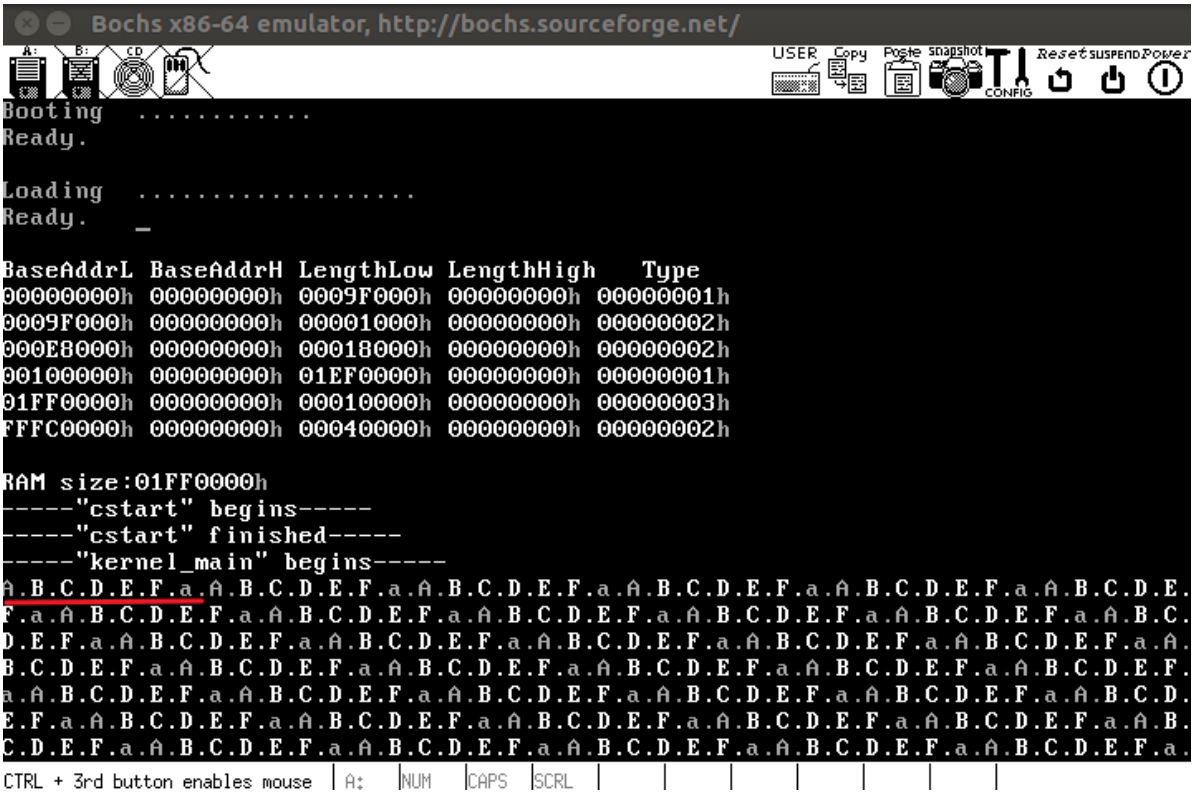


4 实验

4.1 中断返回

```
void CommonA()
{
    int i = 0;
    while (1) {
        print_str("A.");
        milli_delay(10);
        print_str("a.");
    }
}
```

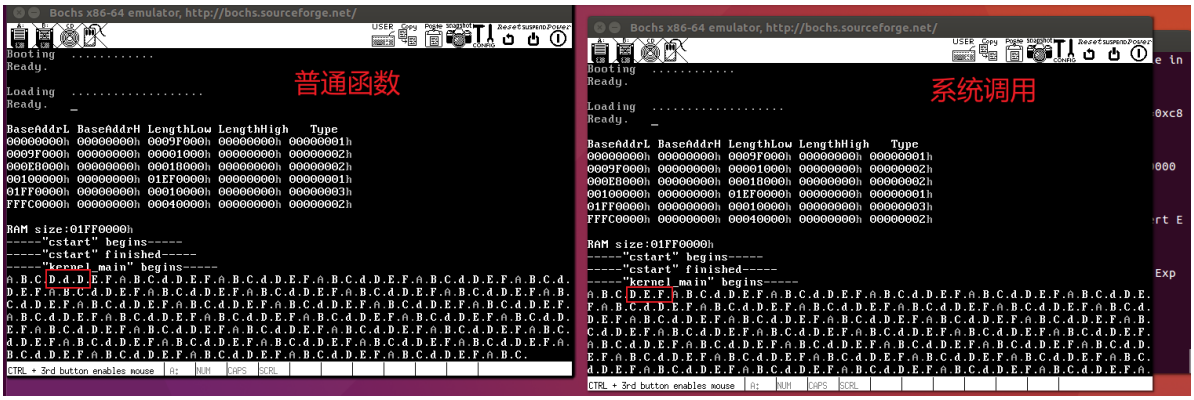
schedule调度策略采用顺序调度，如下结果可以发现，当再次调度到A的时候是先打印的是`a.`，可见再次调度之后确实是从中断处继续执行的。



4.2 进程调度

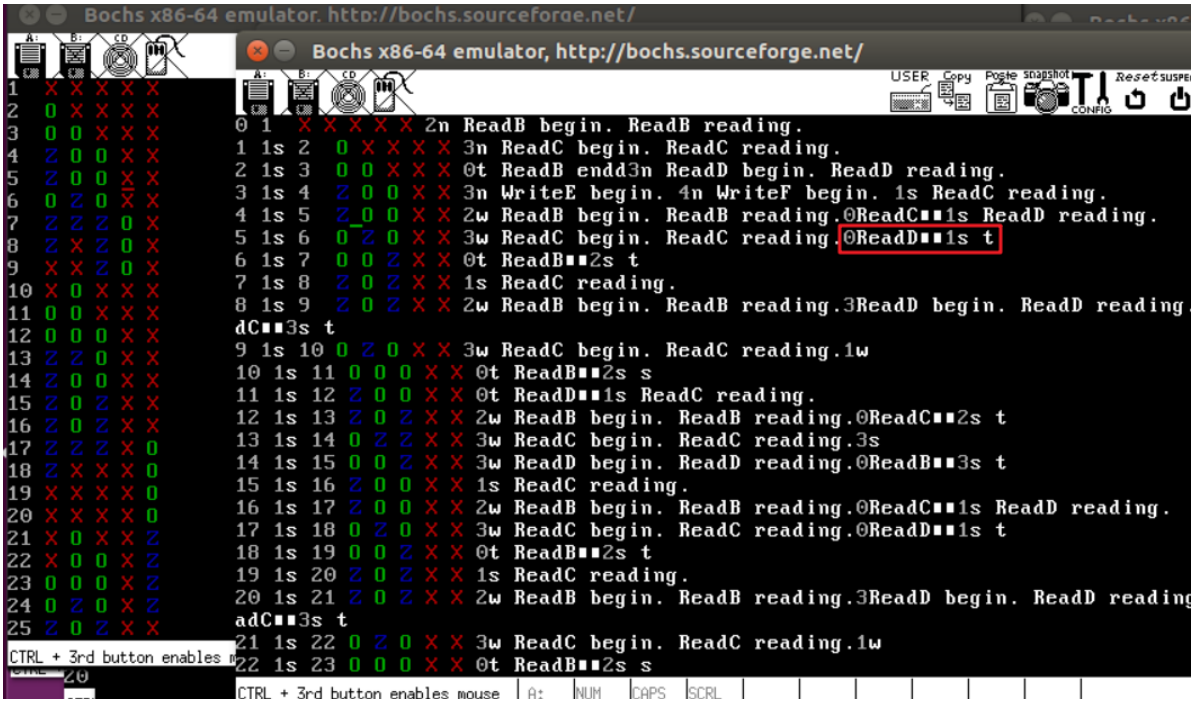
同样的sleep函数，一个是普通函数实现，一个是系统调用实现，结果是只有系统调用才能正确实现进程调度。

因为系统调用返回后回到`p_proc_ready`所指的进程中执行，而普通函数返回后仍然在原来函数中执行，即使`p_proc_ready`已经改变。



奇怪的问题：

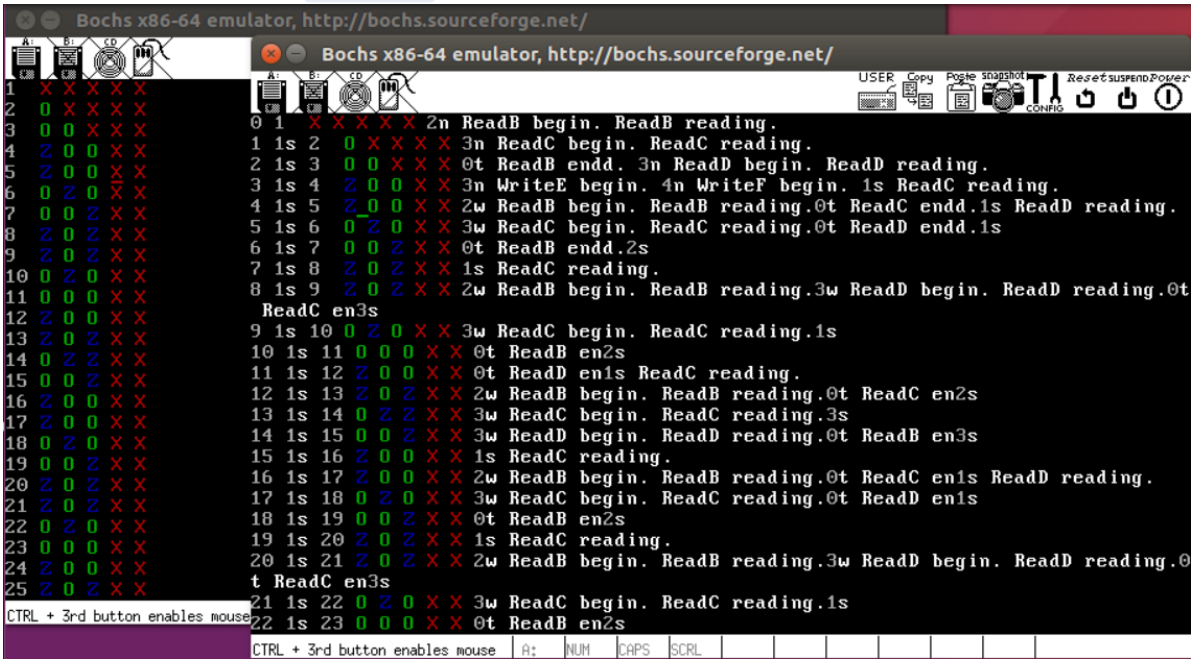
右侧只是做了一些输出，结果就和左边不一样！



原因：在进程中直接调用了 `schedule` 函数改变了 `p_proc_ready`，但是由于 `schedule` 函数在用户态执行完毕后返回原进程，就没有起到预想中的调度的作用。但是增加了一些输出就做到了，是因为输出方法是系统调用，会进入内核态，然后从内核态返回之后，就进入了 `p_proc_ready` 所指定的进程中执行，起到了调度的作用！

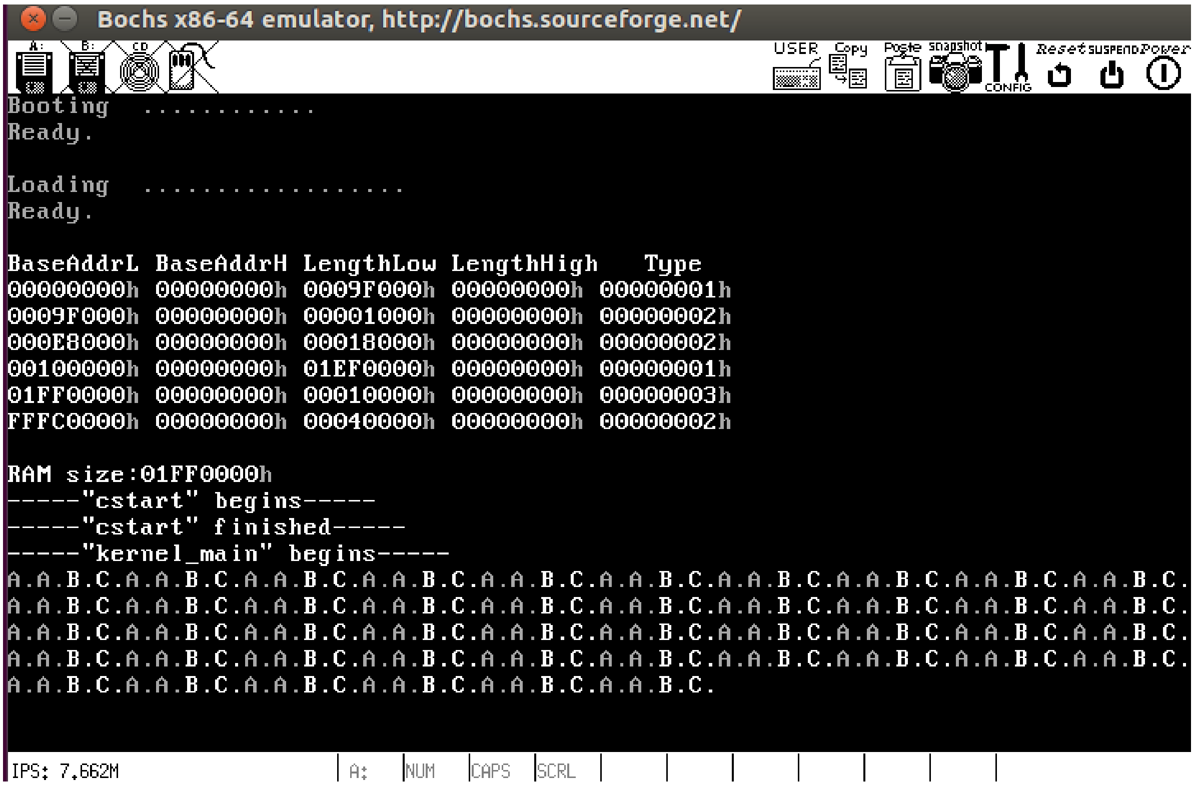
在图中红框部分，`0` 和 `t` 本应该是 `schedule` 函数中相邻的两次输出，但是中间却夹着一部分字符，这一部分字符就是从第一个打印系统调用中返回后进入新的 `p_proc_ready` 指定的进程中执行的输出，出现乱码也应该是因为之前不正当使用 `schedule` 函数引起的。

解决：用一个系统调用封装 `schedule` 函数供进程调用，修改后发现程序行为正常且一致。

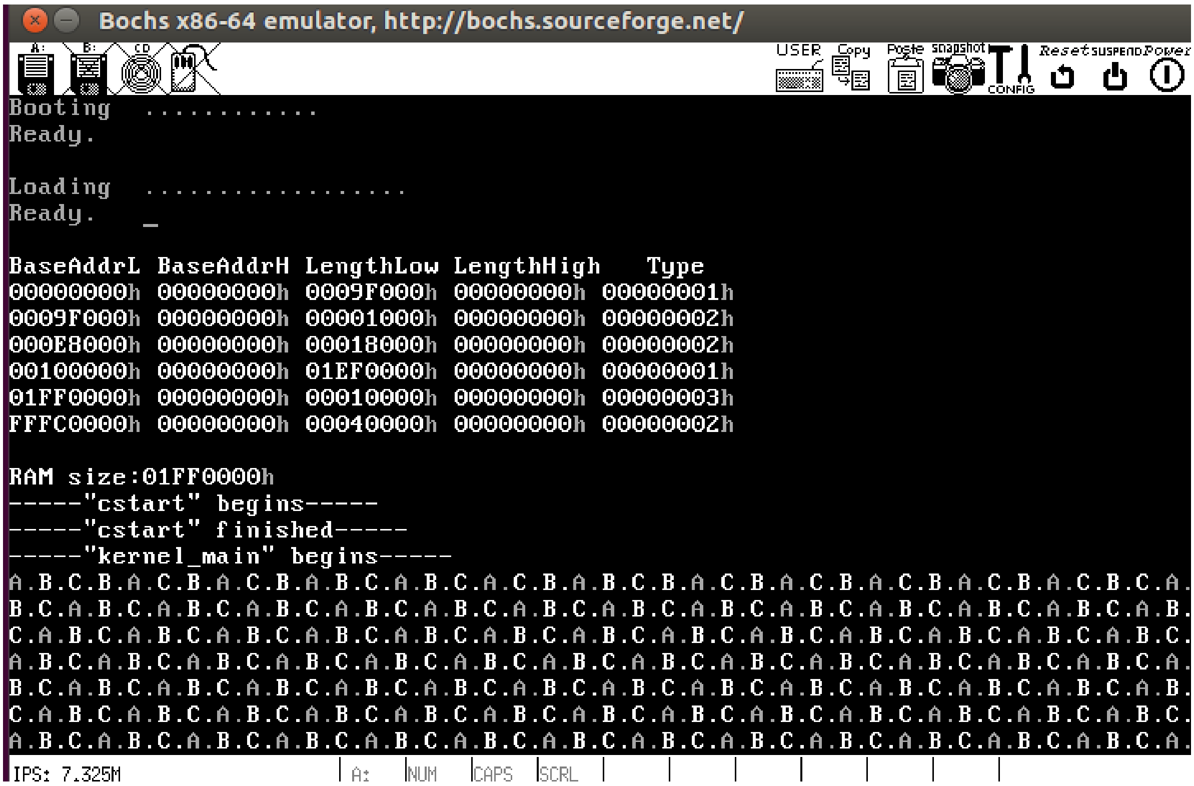


4.3 sleep和milli_delay的对比

对A任务进程执行 `milli_delay(100)`



对A任务进程执行`sleep(100)`



两者的区别就在于：对A执行`sleep(100)`后就不参与进程调度了，所以只有B和C进程参与调度；但是对A执行`miili_delay(100)`，A只是在这`100ms`中不执行任何操作而已，仍然参与进程的调度。