1. ***添加系统调用***

传递参数：在用户态将栈中的参数存入寄存器，ebx,ecx ,edi ，转入内核态时，将寄存器的值压入核心栈，便可将参数传给系统调用方法（堆栈的转换和现场的保护原代码已经写好，不需要再考虑，可直接使用堆栈，只要知道此时用的是哪个堆栈即可）

（参考：第六章，6.3特权级转换，用户栈，核心栈的转换，进程表，TSS

第七章 7.5 printf 传递参数的方式）

syscall.asm 添加系统调用的对外接口

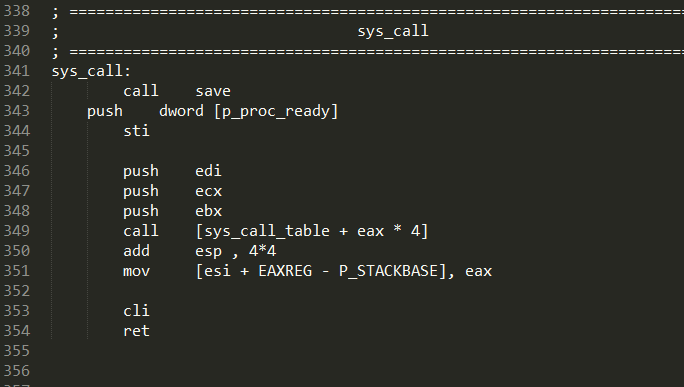
proc.c 添加系统调用的实际方法

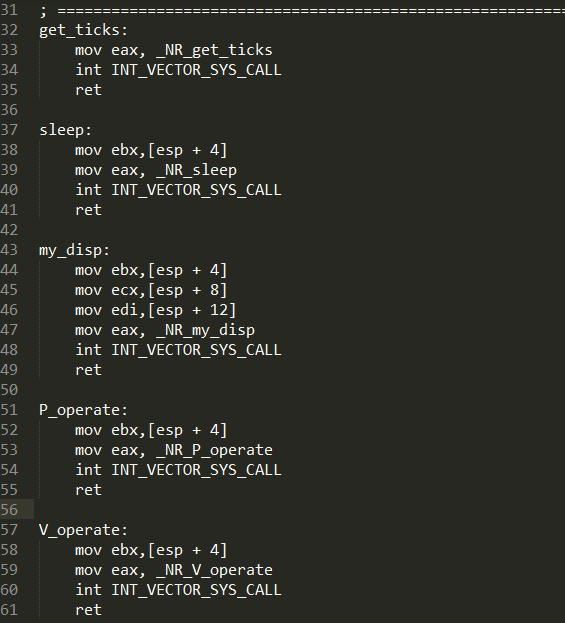
kernel.asm 修改了sys\_call方法，对寄存器中的参数进行压栈，默认传三个参数

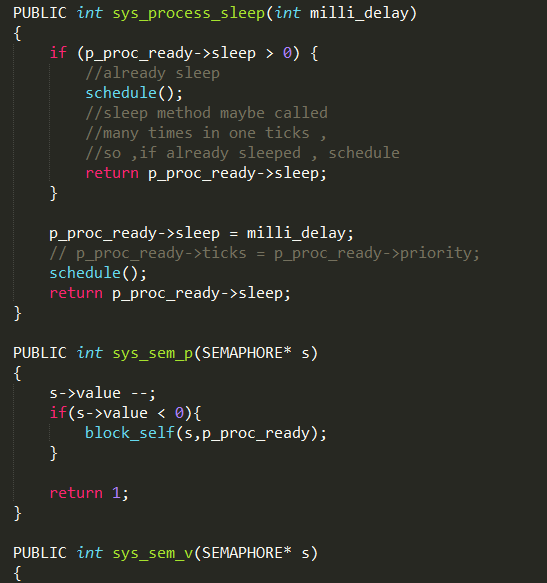
global.c : 系统调用表中添加具体系统调用方法

const.h: 修改系统调用数量

（参考，第六章 6.5 6.6）







1. ***增加进程***

main.c 添加进程体

proc.h 修改进程的个数，定义每个进程的用户栈大小

global.c 在进程表中添加对应的pcb

（参考6.4.6， 但要稍微做一点修改，因为第七章代码稍有不同）

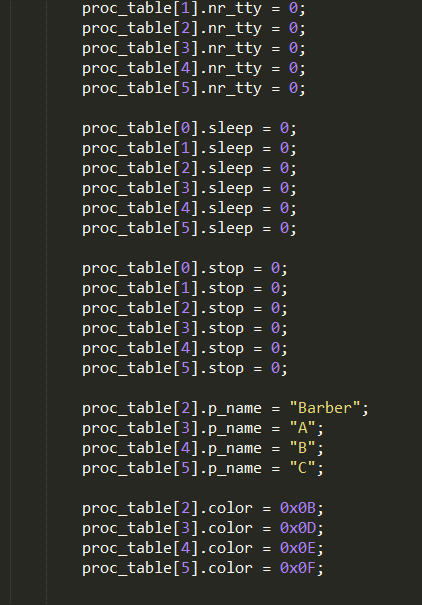
进程中添加属性：int sleep 计数剩余睡眠时间片数

int stop 标记进程是否阻塞

int color 进程输出的颜色

设置优先级：即规定允许连续运行的时间片数

以上属性均在main.c 中初始化



1. ***Sleep***

（参考 6.8 了解mills\_delay方法的实现）

在进程体中添加一个属性，保存sleep的时间

在main.c的kernel\_main方法中初始化为0

sys\_process\_sleep被调用时，将当前进程的sleep值设置为参数要求的时间（换算成时钟中断次数）

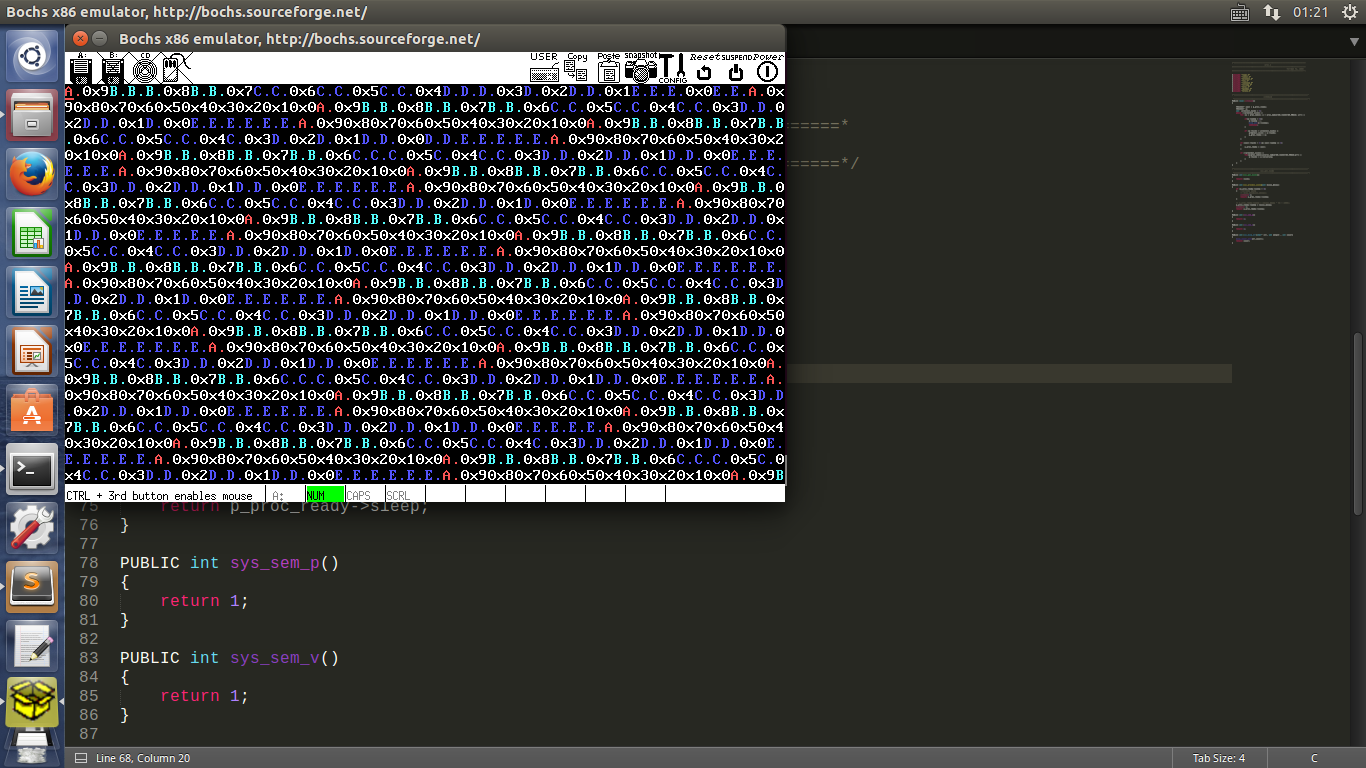
每次schedule时，检查每个进程sleep是否为0，不为0，则-1，跳过本次时间片分配

开始测试是否成功：我删掉了mills\_delay，并添加了自己的sleep

此时出现了一个问题，满屏幕只有第一个进程的输出

原因：每个进程都是死循环输出，修改前的代码在输出一次后会保持一个时钟周期死循环什么也不做，而现在在一个时钟周期内去掉了什么都不做的死循环而不停打印，且期间ticks不会减少，所以不会调度其他进程占用CPU

修改：保留mills\_delay(10)，并在sleep后强制进行一次schedule



但此处又出现一个问题：

原代码中，schedule方法并不是每个时钟周期都会调用，当前进程的ticks没有耗尽时，是不会进行调度的，所以没办法保证每个周期都让sleep的进程的sleep属性-1

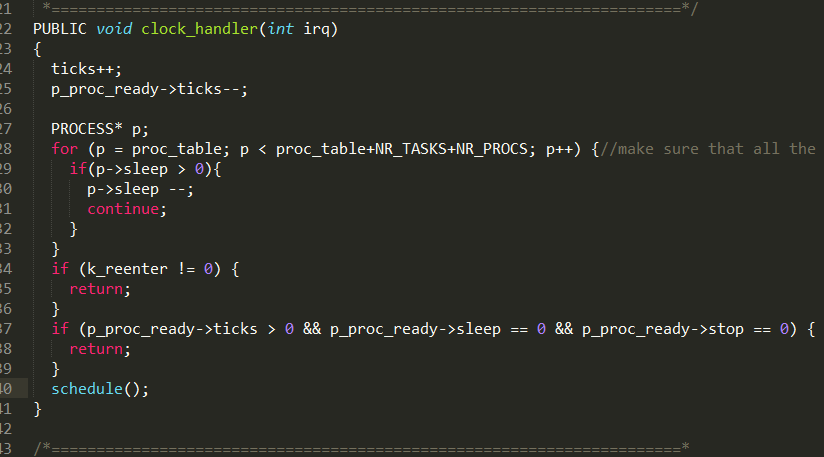
解决如下：

将clock\_handler方法修改，要求每个时钟周期都遍历一次进程表

将所有sleep的进程的sleep属性-1，并跳过该进程的调度

如果调度前的进程ticks没有耗尽，则归还控制权

修改在proc.c , clock.c



1. 添加信号量定义,椅子，等待人数 proc.h

信号量声明 global.c，global.h

信号量初始化 main.c

添加唤醒，阻塞方法 proc.c

为每个进程添加 stop 属性，标记是否阻塞

信号量：定义结构体semahore，包括：

int value , 计数资源数和等待进程数

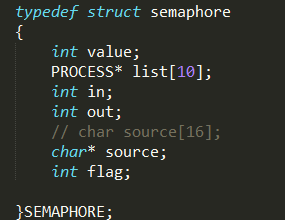
PROCESS×[] ， 等待进程队列（循环队列）

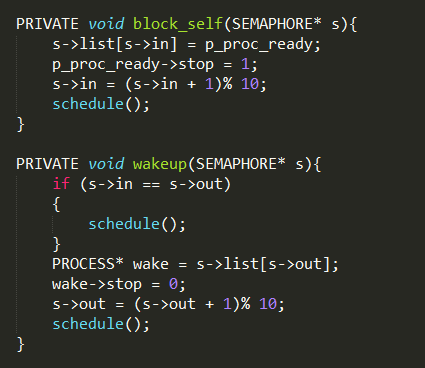
int in， int out ， 维护进程等待队列

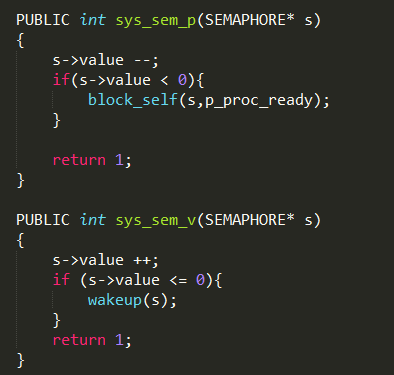
char\* source ， 所代表的资源名称

注意：

唤醒和阻塞方法的最后应强制进行一次schedule，否则还会出现之前sleep的问题



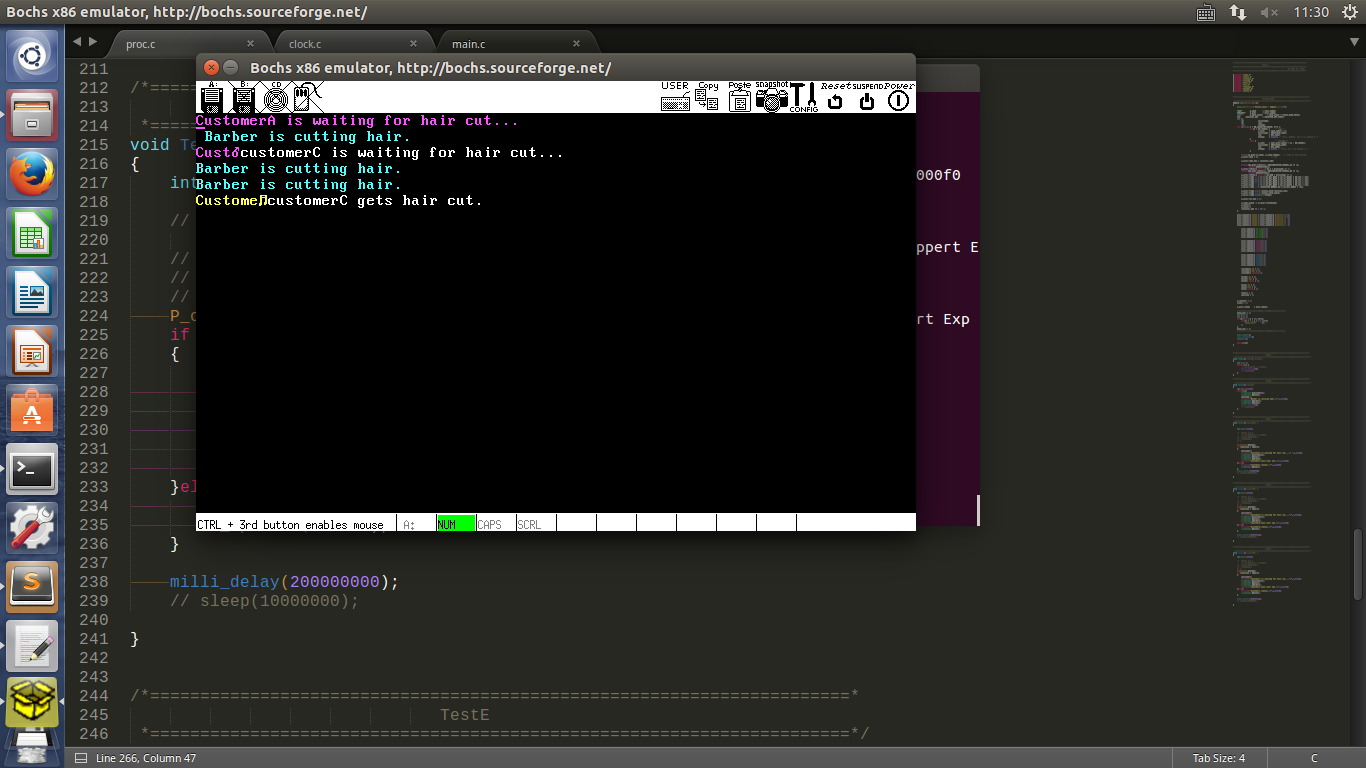




1. ***输出乱码问题***

利用disp\_color\_str实现的输出系统调用会出现乱码的偶然情况

解决：重新实现输出系统调用，改为用out\_char方法，依次打印字符



1. ***另附学习时的一些笔记***

笔记1：

准备工作：

ring0 运行进程调度

ring1 运行任务

一个数据结构记录各进程的状态

进程转换：

进程a运行

时钟中断发生，ring1 to ring0,中断处理程序启动

进程调度，选定进程b

进程b被恢复，ring0 to ring1

进程b运行

要点：

1.保存进程状态

保存所有寄存器内容，pushad(压入pcb) 写在时钟中断例程的最顶端（第一件事）

2.恢复b进程:pop 后 iretd即可

3.进程控制块pcb:保存现场,LDT也保存在PCB中

4.进程切换中esp指向不同位置：

a.进程栈，进程运行时自己的堆栈

b.进程表，pcb

c.内核栈，进程调度模块运行时的堆栈

（另：tss 任务状态段，通过它实现任务的挂起和恢复。tr寄存器指向当前任务的tss,进程切换时，将当前进程的寄存器值保存到该tss，然后将下一个任务的tss选择子加载进tr，取出当前tr所指向的tss中保存的寄存器信息http://www.cnblogs.com/guanlaiy/archive/2012/10/25/2738355.html）

5.ring1 to ring0

准备好tss

为每个进程准备Ldt（在pcb中）

6.ring0 to ring1

将下一进程的进程表地址加载给esp

根据进程表中的ldt描述符，加载Ldt

将进程表的栈顶存入tss的esp0中

出栈，恢复现场，返回

再次从ring1 to ring0 时，取出tss中的esp0，加载到esp，保存现场

（总结：调度下一进程之前，将下一进程的pcb地址加载进tss, 中断时，从tss中找到对应pcb地址，压栈保存现场）

problem: ring1 to ring0 时，esp何时指向pcb （保存当前进程现场）

answer: 中断发生时，ring1 to ring0 , esp的值变成tss中预设的ring0的esp值，也就指向了pcb

tss 和 pcb 什么关系？？为什么觉得是一样的

暂时认为：tss指向当前的pcb的地址，中断发生时，esp指向tss中指向的pcb的s\_stack，然后进行现场保存，进程调度时，将下一个进程的pcb 地址写入tss

笔记2：

ring0 to ring1

1. kernel.asm

(1)restart()

是操作系统启动第一个进程的入口

p\_proc\_ready 是一个指向进程表的指针，指向进程表

进程状态保存在 s\_proc中，即进程表项.

之后便是加载ldt.

esp + 4: 跳过retaddr 以便执行iretd ??

k\_reenter -- ??

2.时钟中断处理程序

kernel.asm

hwint 00

进程开始之前，初始化一系列寄存器，初始化局部描述符表ldt

在gdt中添加ldt的描述符以便于将选择子加载给ldtr

在gdt中添加tss的描述符以便于将选择子加载给tr

中断处理程序

1. 打开时钟中断init\_8295

2. 让时钟中断不停发生，发送eoi（中断处理结束信号）

3. 中断处理程序：

a.中断发生的开始，esp的值是刚刚从tss里面取到的进程表A的regs最高低至，然后压栈保存寄存器的值.

b.将esp指向内核栈，进行内核操作

c.将esp指回进程表

d.设置tss.esp0准备进程下次被调用

e.恢复寄存器的值

4. 中断嵌套 利用之前的全局变量k\_reenter（类似于一个信号量）来区别是否发生了中断嵌套

添加一个任务的步骤

1. 添加一个进程体

2. 在tasktable中加一条进程项 global.c

3. 为新的进程定义堆栈大小（宏）proc.h

4. 在proto.h中添加函数声明

笔记3：

系统调用

中断方式实现系统调用

问题：中断程序如何获取系统调用？如何获取参数？

前提：设置系统调用的中断号 int 0x90 (syscall.asm)

设置系统调用的中断门的处理方法sys\_call(protect.c)

系统调用的方法本身是通过一个函数指针数组储存的 global.c system\_call\_table[]

添加系统调用： 设置系统调用的具体方法的函数指针（global.c），以及各方法对应的indexs(syscall.asm)

添加一个对外的系统调用接口，用于传递具体系统调用方法的index（syscall.asm）,供用户进程调用

定义具体的系统调用函数（proc.c）

总的来说：系统调用相当于一个多对一对多的关系，多个外部接口，对一个中断门，对多个内核方法

系统调用过程：

进程调用系统调用的对外接口

接口将具体实现方法在system\_call\_table 中的index,作为参数，写入eax,调用系统调用的中断门sys\_call

sys\_call方法取出eax中的系统调用index，找到对应的函数指针，将返回值装在对应pcb中eax的位置，作为返回值

（详见代码 system.asm,kernel.asm,pro.c,global.c）

system.asm : 定义对外接口

proc.c: 定义具体调用方法

proto.h: 对外声明具体系统调用方法，和对外接口

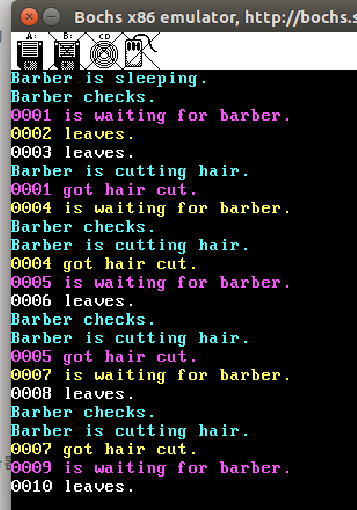
global.c : 系统调用表中添加具体系统调用方法

const.h: 修改系统调用数量

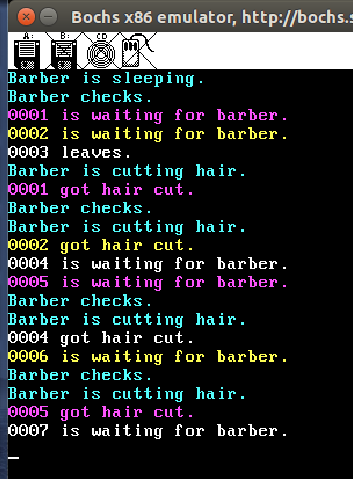
现在的mills\_delay 是分配时间片但是什么也不干

1. ***运行截图：***

Chair = 1



Chair = 2



Chair = 3

