

操作系统课程设计项目文档

1352959 冯馨荷

1352993 黄航

一、项目概述

1.1 项目名称：MyOS 操作系统

1.2 开发环境：Windows 操作系统

VMware 虚拟机软件

Ubuntu Linux 系统

Bochs 虚拟机

Gcc 编译器

Nasm 编译器

1.3 使用说明:

1.3.1 打开 VMware 虚拟机，启动 ubuntu 系统。

1.3.2 打开命令行，执行项目的 makefile 文件。

1.3.3 编译成功后，启动 bochs 虚拟机运行目标操作系统。

1.3.4 在 bochs 环境里与目标操作系统进行交互。

1.4 参考书籍：《Orange' S 一个操作系统的实现 》

二、系统模块

- BootLoader：作为系统的 bootloader，用于加载系统的其它模块进入内存指定位置，并将系统控制转移至后续模块。
- SysBoot：初始化系统 GDT、GDT Selector、8259A 可编程中断控制器，为系统进入保护模式做准备工作，并进入保护模式。
- System：真正常驻内存的系统内核。内核首先初始化并启动 IDT 和内核分页机制，设置系统调用及中断处理程序。最后，系统加载任务并执行。

三、功能设计

3.1 进程调度

3.1.1 分析

在原书中,进程调度采用时间片加优先级算法。我们将该算法修改为非抢占式多级反馈队列调度算法。

我们定义三个队列,分别赋予不同的时间片长度,依次递减。原则是当一个新进程进入内存后,将它放入第一队列(即优先级最高的队列)的末尾,按照 FCFS 原则排队等待调度。因为在同一队列中的进程是采用时间片轮转法,则不存在抢占剥夺式的情况。当轮到该进程执行时,若它在所在队列的时间片内执行完,则撤离系统,若没有执行完,则将它调入低一级队列的末尾,等待调度,同样是 FCFS 原则。

当高级队列空闲时,低级队列才会开始调度。

3.1.2 算法

添加 5 个测试进程 Test1(),Test2(),Test3(),Test4(),Test5(),为每个进程赋值一个运行时间,每调用一次总时间减一。

```
void test1(int time)
{
    while(time)
    {
        time--;printf("a");
        milli_delay(10);}
}
```

定义三个队列，用数组表示，因为在定义结构体时，无法在结构体里定义结构体，所以弃用。

```
int queue1[5];
```

```
int queue2[25];
```

```
int queue3[50];
```

开始调用，如果在该队列未执行完毕，则放入低一级队列。

```
if(t1pro>t1)
```

```
{
```

```
    test1(t1);
```

```
    t1pro=t1pro-4;
```

```
    queue1[0]=1;
```

```
    j=j+1;
```

```
    printf(t1);
```

```
}
```

```
else
```

```
{
```

```
    test1(t1pro);t1pro=0;
```

```
}
```

```
if(t1pro>t2&&queue1[i]==1)
```

```
{
```

```
    test1(t2);
```

```
    t1pro=t1pro-t2;
```

```

        queue2[k]=1;

        k=k+1;

    }

    else if(t1pro<=t2&&queue1[i]==1)

    {

        test1(t1pro);t1pro=0;

    }

    if(t2pro>t3&&queue2[i]==2)

    {

        test2(t3);

        t2pro=t2pro-t3;

        queue2[sum+i]=2;  //放入队尾等待调度

    }

    else if(t2pro<=t3&&queue2[i]==2)

    {

        test2(t2pro);t2pro=0;

    }

```

3.1.3 运行效果

输入 process 命令

```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
=====
=                               =
=           MyOS                =
=      Kernel on Orange's       =
=           Main                 =
=Designed by FengXinHe & HuangHang=
=====

[myos@option/] help
=====
Command List      :
1. process        : Show the Multilevel queue feedback scheduling algorithm
2. caculator      : Make a simple caculate
3. file           : Enter the file management system
4. clock          : Show the system time
5. clear          : Clear the screen
6. help           : Show this help message
=====

[myos@option/] process
```

展示多级反馈队列调度算法

```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
=====
=                               =
=           MyOS                =
=      Kernel on Orange's       =
=           Main                 =
=Designed by FengXinHe & HuangHang=
=====

[myos@option/] process
=====
test process      |      time
test1()           |      15
test2()           |      40
test3()           |       5
test4()           |      30
test5()           |       1
=====
que2323ue 1 running
aaaa$ bbbccccddde
queue 1 over
queue2 running
aaaaaaaaabbbbbbbbcdddddd
queue 2 over
queue 3 running
aaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
queue 3 over
[myos@option/]
```

3.2 文件管理系统

3.21 分析

文件管理系统是我们比较熟悉的，在操作系统的课程中就已经实现过，现在所写的文件管理系统是基于自己的操作系统之上的，所有许多的库函数都是不可以调用的，都是从底层实现做起，在这个操作系统之下的文件管理系统的实现是平面的，即没有子目录，所有的文件保存于根目录下，由于汇编方面的问题，在关闭这个操作系统之后文件无法实现保存，而在保持运行操作系统的时候，文件能够实现保存和支持其他编辑、删除操作。

3.22 实现

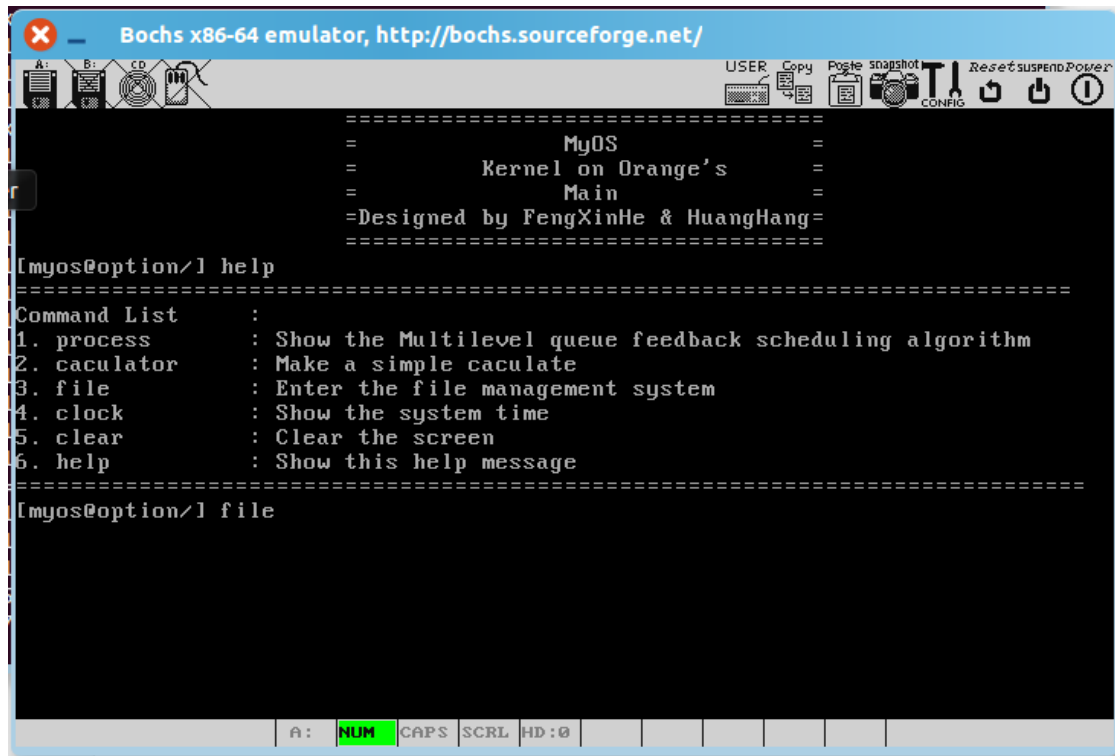
操作系统本身就是一个生命周期十分长的进程，从启动操作系统到关闭操作系统。这里我们用 struct 来保存文件的名称、内容。

```
struct file
{
    char* name;    //文件名
    char* content; //文件内容
}
```

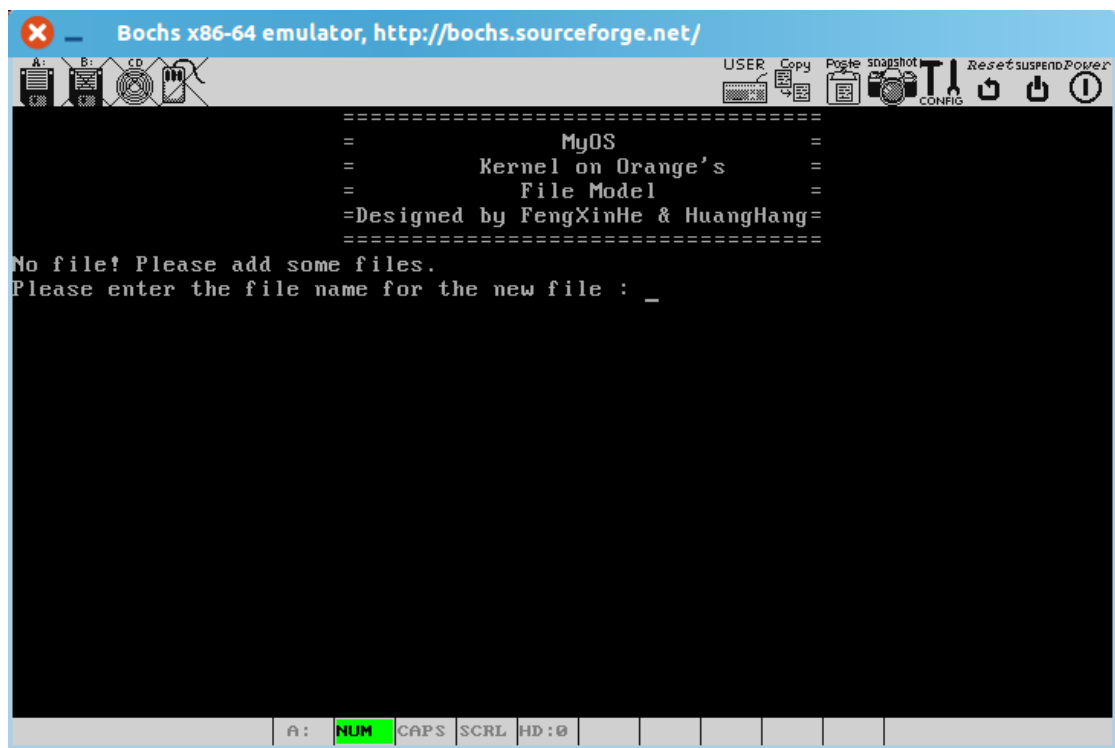
对文件的操作有添加内容、展示文件信息、删除文件内容和关闭文件。这些操作在一个循环中实现，当用户键入 quit 即关闭文件的操作时跳出循环来实现关闭文件。

3.23 运行效果

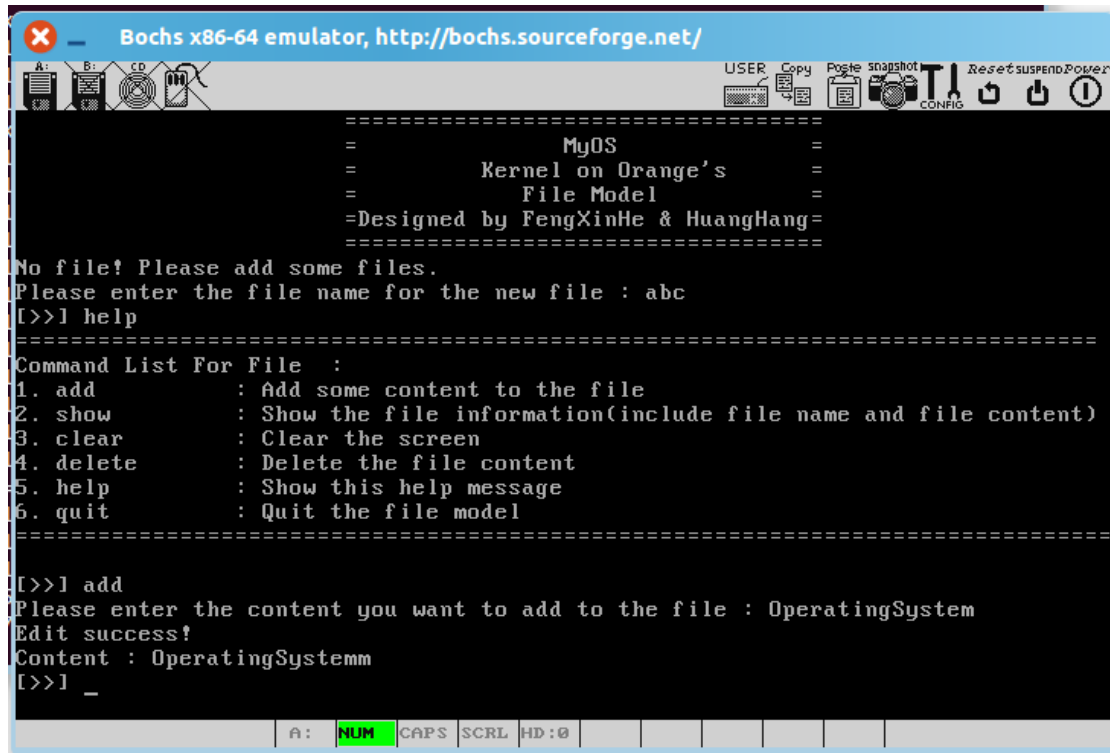
输入 file 命令



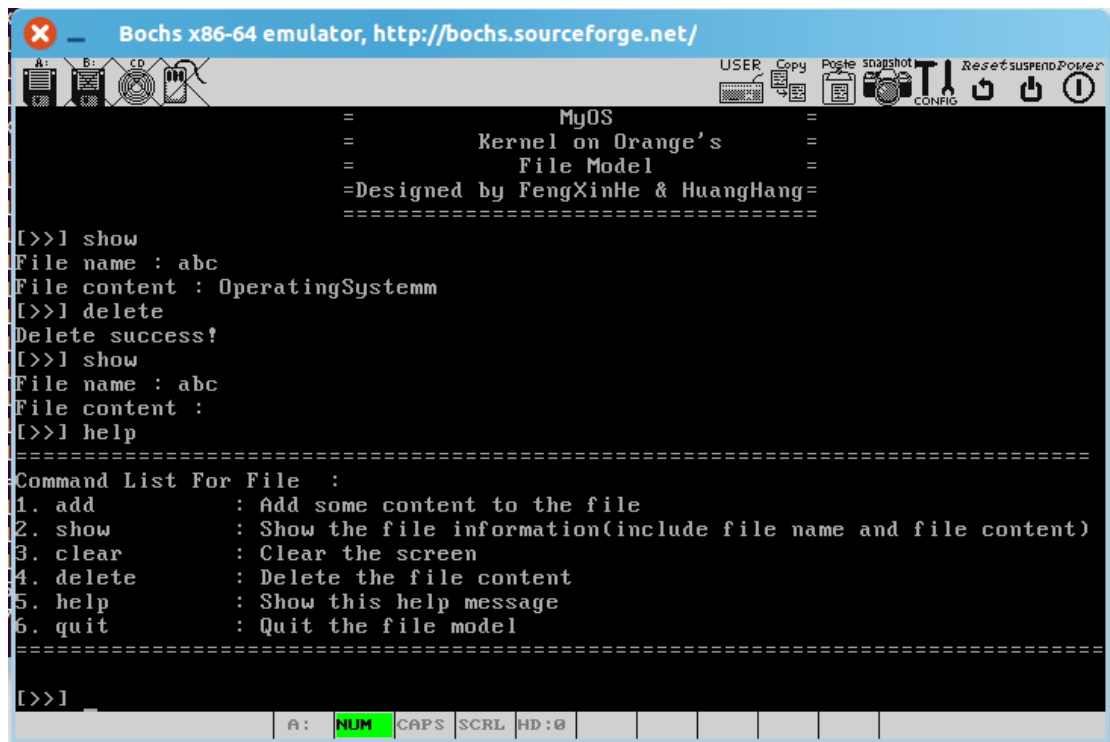
没有文件，创建一个新文件



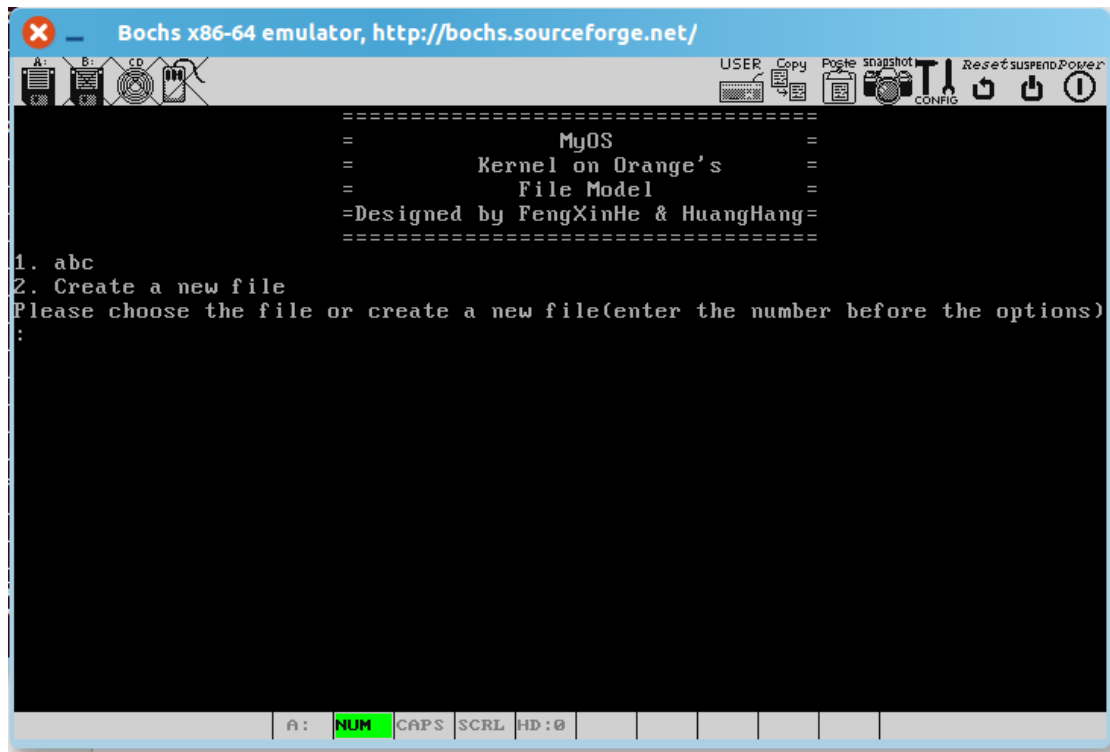
命名文件名字，添加文件内容



基本功能展示



再次进入文件系统



3.3 计算器

3.31 分析

计算器属于用户级应用，当用户进入计算器时候，系统会请求用户输入两个数字和对这两个数字进行的操作的运算符，输入完毕之后系统返回运算结果，支持两个整数间的运算，支持加减乘除四种基本运算。用户做完一次计算之后可以选择继续，也可以选择跳出计算器，回到主界面输入其他功能的指令来进行其他功能的操作。

3.32 实现

计算器属于用户调用的一个函数，主要由 `pinrf()` 函数、`read()` 函数、`atio()` 函数和系统本身的普通运算来实现，`read()` 函数需要给出三个变量，读取完毕后返回读取字符的数量并将 `*buf` 指向用户输入信息保存的地址，由于保存的内容是字符串，调用系统所包含的 `atio()` 函数，将字符串转化为数字进行保存，`atio(source, destination)` 函数中需要两个变量，将 `source` 变两个的内容赋值给 `destination`，将数字保存于 `destination` 之中，之后使用系统

的四则基本运算对两个 destination 中的值进行运算，返回得出的值。

```
PUBLIC int read(int fd, void *buf, int count)

{

    MESSAGE msg;

    msg.type = READ;

    msg.FD    = fd;

    msg.BUF   = buf;

    msg.CNT   = count;

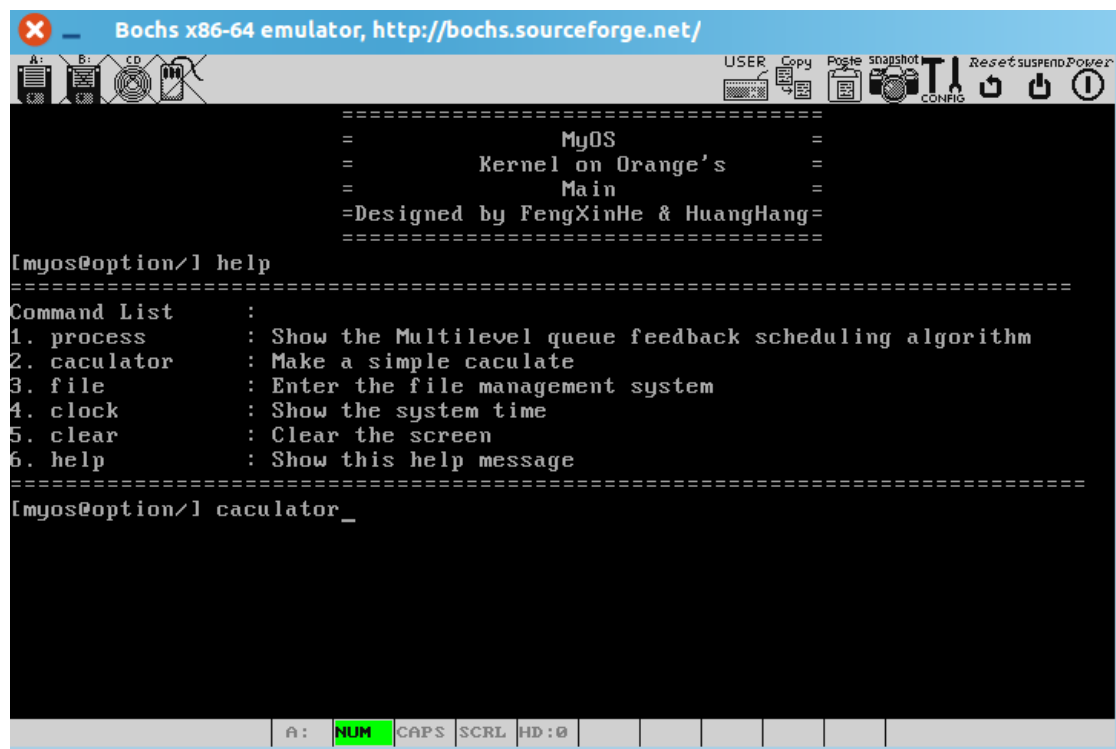
    send_recv(BOTH, TASK_FS, &msg);

    return msg.CNT;

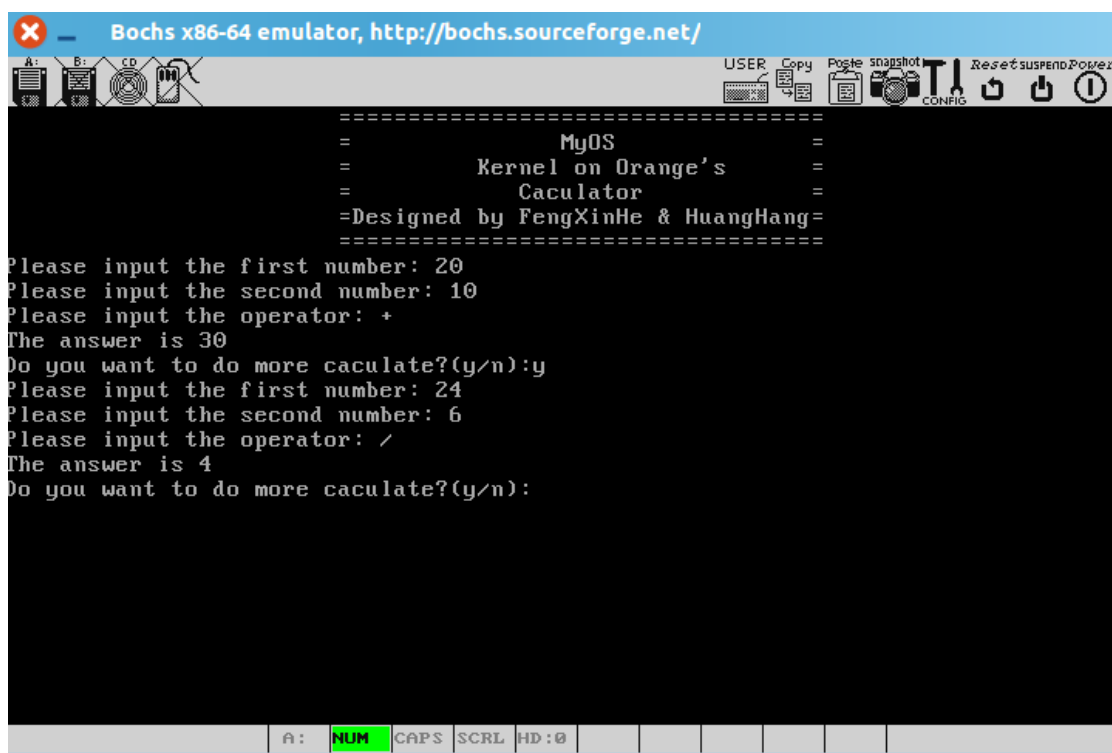
}
```

3.33 运行效果

输入 caculator 命令



进行简单计算



```
=====
=                               =
=           MyOS                =
=       Kernel on Orange's      =
=           Caculator           =
=Designed by FengXinHe & HuangHang=
=====

Please input the first number: 20
Please input the second number: 10
Please input the operator: +
The answer is 30
Do you want to do more caculate?(y/n):y
Please input the first number: 24
Please input the second number: 6
Please input the operator: /
The answer is 4
Do you want to do more caculate?(y/n):
```

3.4 时钟

3.4.1 分析

本身系统没有 C 语言中所带有的 time.h 库函数，在没有 api 支持的情况下，我们实现的时钟需要用户先初始化时间，设定好时间的初值，当设定完成之后就不需要重复设置，用户可以调用 clock 指令来对时间进行查看，10 秒之后，会从时钟界面跳回到主界面，如果想要继续查看，可以继续在主界面中输入 clock 来查看时间。

3.4.2 实现

我们使用 struct 对时间进行封装，一共有 6 个变量：年、月、日、小时、分钟和秒钟。同时我们设定全局变量 initial 来判断是否已经初始化，initial = 0 时表示未初始化，initial = 1 时表示已经进行过初始化可以直接查看时间。初始化的过程中，为了时间的正确表示，对于时间初始化的内容也是严格限制的，首先会判断用户输入的是否为数字，如果不是数字

那么就会返回错误信息，让用户重新输入，然后会判断用户输入的数字是否符合常理，比如月份要保持在 1~12 之间，日根据月份进行限制，小时保持在 0~23 之间,分钟和秒钟都保持在 0~59 之间，如果存在有有一项的输入有误则都会返回错误信息，让用户重新输入。当完成时间的初始化之后，则会跳到显示时间的界面，10 秒之后自动返回主界面，再次进入不需要初始化操作。

```
//clock

struct time
{
    int year;

    int month;

    int day;

    int hour;

    int minute;

    int second;

};

int initial = 0; //0 表示未初始化，1 表示已初始化
```

其中对于时间进位的控制，我们保存在 update()函数中

```
void update(){

    t.second++;

    if(t.second == 60)

    {
```

```
t.second = 0;

t.minute++;

}

if(t.minute == 60)

{

    t.minute = 0;

    t.hour++;

}

if(t.hour == 24)

{

    t.hour = 0;

    t.day++;

}

if(t.day == 32)

{

    t.day = 1;

    t.month++;

}

if(t.month == 13)

{

    t.month = 1;

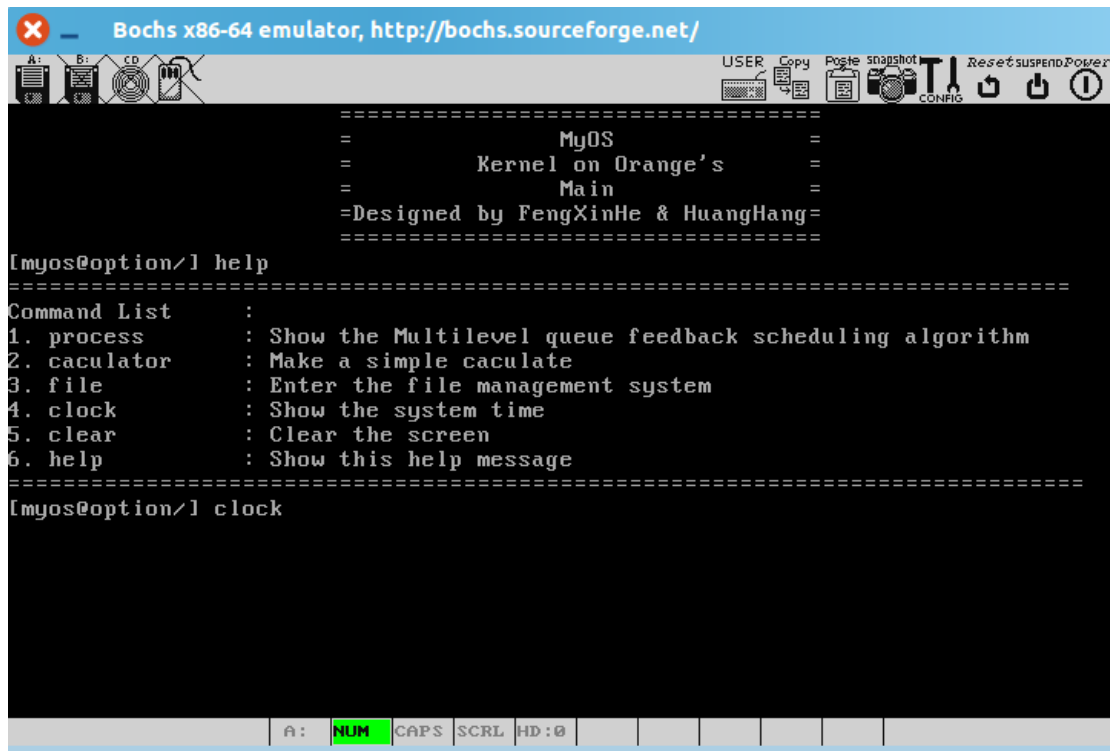
    t.year++;
```

}

}

3.4.3 运行效果

输入 clock 命令



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

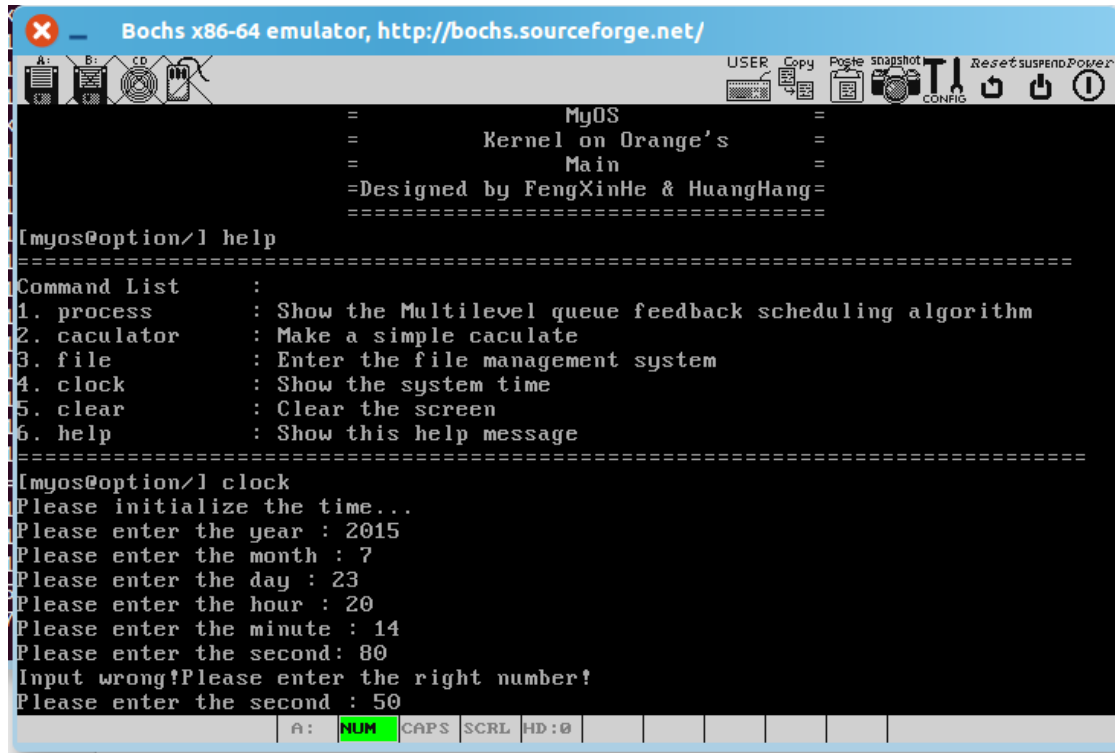
=====
=                               =
=           MyOS                =
=   Kernel on Orange's         =
=           Main                =
=Designed by FengXinHe & HuangHang=
=====

[myos@option/] help
=====
Command List      :
1. process        : Show the Multilevel queue feedback scheduling algorithm
2. caculator      : Make a simple caculate
3. file           : Enter the file management system
4. clock          : Show the system time
5. clear          : Clear the screen
6. help           : Show this help message
=====

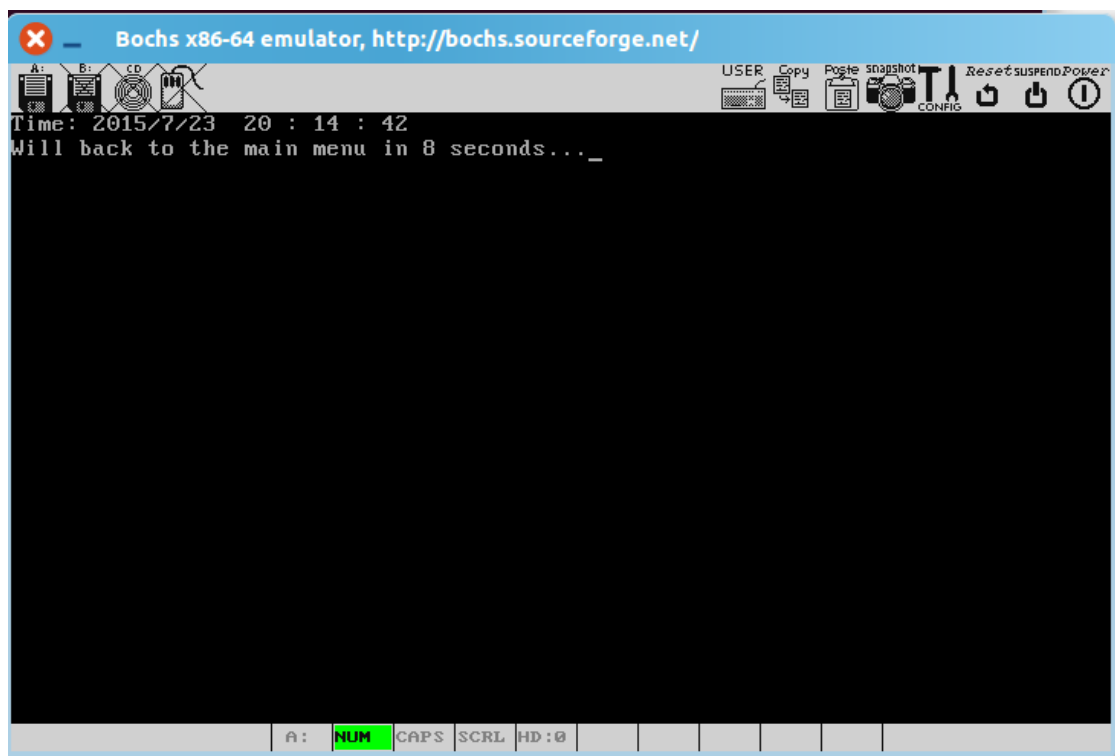
[myos@option/] clock

A: NUM CAPS SCRL HD:0
```

初始化时钟



时钟界面



四、操作流程

- 4.1 使用 Bochs 模拟器进行开机
- 4.2 等待开机完成，当开机完成后跳入主界面进行操作
- 4.3 输入 help 命令查看该系统下所有的操作指令
- 4.4 选择不同的操作指令跳转到不同的模块或者功能。

五、遇到的问题与解决方法

5.1 编译为启动文件

```
nasm boot.asm -o boot.bin
```

5.2 将引导扇区写进软盘

```
dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
```

5.3 将 loader.bin 复制到虚拟软盘 a.img 上

```
sudo mount -o loop a.img /mnt/floppy/
```

```
sudo cp loader.bin /mnt/floppy/ -v
```

```
sudo umount /mnt/floppy
```

5.4 关于 floppy 无法挂载问题

```
##在根目录的 mnt 下通过终端 ( sudo mkdir floppy ) 创建 floppy 文件夹
```

```
##若 floppy 文件已存在，则先删除该文件 ( 管理员身份执行 rm -rf floppy )
```

5.5 启动模拟操作系统

`bochs -f bochsrc` 可以简化为 `bochs` 系统会自动搜索该目录下的 `bochsrc` 文件

5.6 创建软盘映像

命令行下执行 `bximage`

5.7 bochsrc 文件 添加 lib

由于我们安装 `bochs` 模拟器的方法和书中的略有不同，所有安装的 `bochs` 中缺少配置文件和一些库文件 我们在终端下使用 `sudo apt-get install bochs-sdl` 来下载 `sdl` 的库，然后在配置文件中使如下命令来进行库文件的调用。

同时将 `keyboard_mapping` 使用“`#`”注释掉

`display_library: sdl`

5.8 编译链接

`nasm -f elf hello.asm -o hello.o` `##-f elf` 输出 format 为 elf 格式

`ld -s hello.o -o hello` `##-s` 为 `stripi` 去掉符号表内容 减少字符串的冗余

`./hello`

5.9 编译汇编&&C

`nasm -f elf foo.asm -o foo.o`

`gcc -c bar.c -o bar.o`

`ld -s foo.o bar.o -o foobar`

5.10 ELF 格式

executable and linkable format

5.11 启动与内核连接的地址问题

使用 loader.bin 连接内核，对内核的入口地址进行修改,改为 0x30400

```
nasm -f elf kernel.asm -o kernel.o
```

```
ld -s -Ttext 0x30400 kernel.o -o kernel.bin
```

5.12 内核进行输出

```
nasm -f elf kernel.asm -o kernel.o
```

```
nasm -f elf string.asm -o string.o
```

```
nasm -f elf kliba.asm -o kliba.o
```

```
gcc -c -fno-builtin start.c -o start.o
```

```
ld -s -Ttext 0x30400 -o kernel.bin kernel.o string.o start.o kliba.o
```

5.13 gcc 编译错误问题

Makefile 因 gcc 版本问题编译错误时，在 CFLAGS 的定义里添加

```
-fno-stack-protector
```

```
CFLAGS = -I include/ -c -fno-builtin -fno-stack-protector
```

因为考虑到虚拟机的时间，我们使用 32 位的系统来减少对电脑的负荷，所以没有

遇到 x86 64 位下无法完成编译的问题，这也算是意外的收获，避免了不少麻烦。