操作系统课程设计实验报告

姓名：薛雨萌
学号：201605130109
指导老师：杨兴强

目录

实验简介

1. 阅读Linux 0.11内核源代码，找到自己感兴趣的部分
2. 提取相关部分数据
3. 对相关部分数据进行可视化

实验流程

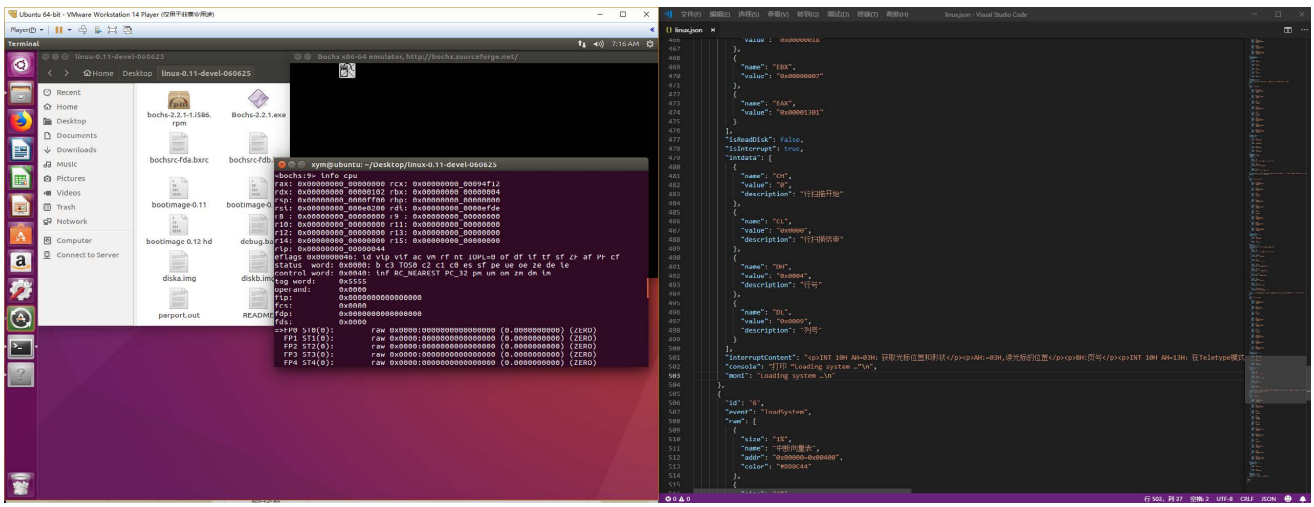
阅读代码

1. 阅读了重点部分的代码，如第三章、第四章、第五章，第九章和第十章等，确定了对开机启动部分更为感兴趣，决定对这部分进行可视化
2. 重点阅读bootsect.S, setup.S, head.S以及main.c，查阅资料了解BIOS如何进行引导
3. 写了针对三个汇编文件的分析笔记，见附录

提取数据

由于开机部分各种提取数据的方式都不可用，陈宇翔同学最初的版本也是不能够提取汇编部分数据的，因此直接在Bochs虚拟机里手动输入命令进行调试，过程十分复杂，差不多将整个运行过程自己一步步跟着走了一遍，不过收获也是巨大的，有了之前的分析笔记，实现起来就有条理多了。

在Bochs中直接调试也是有很多问题的，网上大多数Linux 0.11虚拟机配置文件基本都不能在比较新的Linux发行版下的Bochs中启动，我翻阅了Bochs的官方文档，找到了该如何配置虚拟机，一步一步解决了虚拟机配置的问题，直至能顺利启动。



与上一级的对比

对比去年杨浩然同学的数据，BIOS引导部分十分固定，差别不大，操作系统引导部分，主要是那三个汇编代码，有了非常大的改进，对比去年的数据，多了很多新的数据，详细丰富了很多，基本将整个开机阶段的数据完整提取了出来。并且数据真实，完全是手动调试保存下来的，不仅包括代码里写好的对寄存器操作的值，更是连中断返回的寄存器的值都包括，中断返回值的意义也有说明。这是去年所没有做到的。从数据量上就可以看出我的工作的改进程度。

我的JSON数据为1868行，全部为手工提取：

```
            ],
            "isCarry": false,
            "isChangeRegister": false,
            "isReadDisk": false,
            "isInterrupt": true,
            "interruptContent": "开启分页，设置了5个页表",
            "console": "开启分页，跳转到 main()",
            "moni": ""
        }
    ]
}
}
```


下面是去年杨浩然同学的数据量，总共296行：


```
}

{
    "module":"HEAD",
    "event": "setpage",
    "provider": "yhr",
    "time": 22,
    "data":{
        "description":"开启A20地址线"
    }
}
```


接通电源至运行bootsect.S之前的动作通过上网查询来了解，大致流程如下：

> 电源接通，主板供电，电源键被按下时会通过reset针脚给CPU发送复位信号，CS寄存器置为0xFFFF，IP寄存器置为0x0000。这样一来，CPU就会要求访问地址为0xFFFF0的这个地方，这个地址实际上不是内存的地址，它被 地址控制器（实际上是南桥北桥）映射到 BIOS ROM里，而这个地址的ROM中存放着一条跳转指令。
>
> 0xFFFF0中这条跳转指令，跳转至BIOS ROM的某个地址。从这开始是一段初始化程序，把这段程序复制到内存中执行。这段程序的作用是一方面初始化硬件（如内存等），另一方面 从硬盘加载引导程序（具体方法是从0柱0面0扇区开始寻找，如果扇区最后是”55 AA”，则说明找到该引导程序，否则继续寻找下一扇区，直到找到引导程序）。找到引导（boot）程序之后 复制到内存的 0x07C00——0x7DFF中，然后跳到该地址执行引导程序。

因此，我们可以知道操作系统引导之前的两大步骤，第一步是CPU Reset，访问0xFFFF0，第二步是BIOS引导。

CPU Reset

CPU Reset部分的流程都是硬件实现的，这部分数据无法提取，只能查阅资料大致了解到有寄存器的重置等操作，而BIOS引导就要复杂很多，有很多地方可以可视化。

BIOS引导

1. BIOS程序首先检查，计算机硬件能否满足运行的基本条件，这叫做"硬件自检"（Power-On Self-Test），缩写为POST。如果硬件出现问题，主板会发出不同含义的蜂鸣，启动中止。如果没有问题，屏幕就会显示出CPU、内存、硬盘等信息。
2. 针对动态内存(DRAM)，主板芯片组，显卡以及相关外围的寄存器做初始化设置，并检测是否能够工作。
3. 将系统设置值记录并存储在CMOS等存储区域内。
4. 将常驻程序库(Runtime Program)常驻于某一段内存内，内容大概有中断向量表和BIOS数据区
5. BIOS把控制权转交给下一阶段的启动程序。这时，BIOS需要知道，"下一阶段的启动程序"具体存放在哪一个设备。也就是说，BIOS需要有一个外部储存设备的排序，排在前面的设备就是优先转交控制权的设备。这种排序叫做"启动顺序"（Boot Sequence）。

主引导记录

然后就轮到主引导记录（MBR），BIOS按照"启动顺序"，把控制权转交给排在第一位的储存设备。

这时，计算机读取该设备的第一个扇区，也就是读取最前面的512个字节。如果这512个字节的最后两个字节是0x55和0xAA，表明这个设备可以用于启动；如果不是，表明设备不能用于启动，控制权于是被转交给"启动顺序"中的下一个设备。

这最前面的512个字节，就叫做"主引导记录"（Master boot record，缩写为MBR）。

主引导记录结构

主引导记录由三个部分组成：

位置	内容
第1-446字节	调用操作系统的机器码
第447-510字节	分区表（Partition table）
第511-512字节	主引导记录签名（0x55和0xAA）

分区表

硬盘分区有很多好处。考虑到每个区可以安装不同的操作系统，"主引导记录"因此必须知道将控制权转交给哪个区。

分区表的长度只有64个字节，里面又分成四项，每项16个字节。所以，一个硬盘最多只能分四个一级分区，又叫做"主分区"。

每个主分区的16个字节，由6个部分组成：

位置	内容
第1个字节	如果为0x80，就表示该主分区是激活分区，控制权要转交给这个分区。四个主分区里面只能有一个是激活的
第2-4个字节	主分区第一个扇区的物理位置（柱面、磁头、扇区号等等）
第5个字节	主分区类型
第6-8个字节	主分区最后一个扇区的物理位置
第9-12字节	该主分区第一个扇区的逻辑地址
第13-16字节	主分区的扇区总数

最后的四个字节（"主分区的扇区总数"），决定了这个主分区的长度。也就是说，一个主分区的扇区总数最多不超过2的32次方。

如果每个扇区为512个字节，就意味着单个分区最大不超过2TB。再考虑到扇区的逻辑地址也是32位，所以单个硬盘可利用的空间最大也不超过2TB。如果想使用更大的硬盘，只有2个方法：一是提高每个扇区的字节数，二是增加扇区总数。

硬盘启动

这时，计算机的控制权就要转交给硬盘的某个分区了，这里又分成三种情况。

情况A：卷引导记录

上一节提到，四个主分区里面，只有一个是激活的。计算机会读取激活分区的第一个扇区，叫做"卷引导记录"（Volume boot record，缩写为VBR）。

"卷引导记录"的主要作用是，告诉计算机，操作系统在这个分区里的位置。然后，计算机就会加载操作系统了。

情况B：扩展分区和逻辑分区

随着硬盘越来越大，四个主分区已经不够了，需要更多的分区。但是，分区表只有四项，因此规定有且仅有一个区可以被定义成"扩展分区"（Extended partition）。

所谓"扩展分区"，就是指这个区里面又分成多个区。这种分区里面的分区，就叫做"逻辑分区"（logical partition）。

计算机先读取扩展分区的第一个扇区，叫做"扩展引导记录"（Extended boot record，缩写为EBR）。它里面也包含一张64字节的分区表，但是最多只有两项（也就是两个逻辑分区）。

计算机接着读取第二个逻辑分区的第一个扇区，再从里面的分区表中找到第三个逻辑分区的位置，以此类推，直到某个逻辑分区的分区表只包含它自身为止（即只有一个分区项）。因此，扩展分区可以包含无数个逻辑分区。

但是，似乎很少通过这种方式启动操作系统。如果操作系统确实安装在扩展分区，一般采用下一种方式启动。

情况C：启动管理器

在这种情况下，计算机读取"主引导记录"前面446字节的机器码之后，不再把控制权转交给某一个分区，而是运行事先安装的"启动管理器"（boot loader），由用户选择启动哪一个操作系统。

Linux环境中，目前最流行的启动管理器是Grub。

操作系统

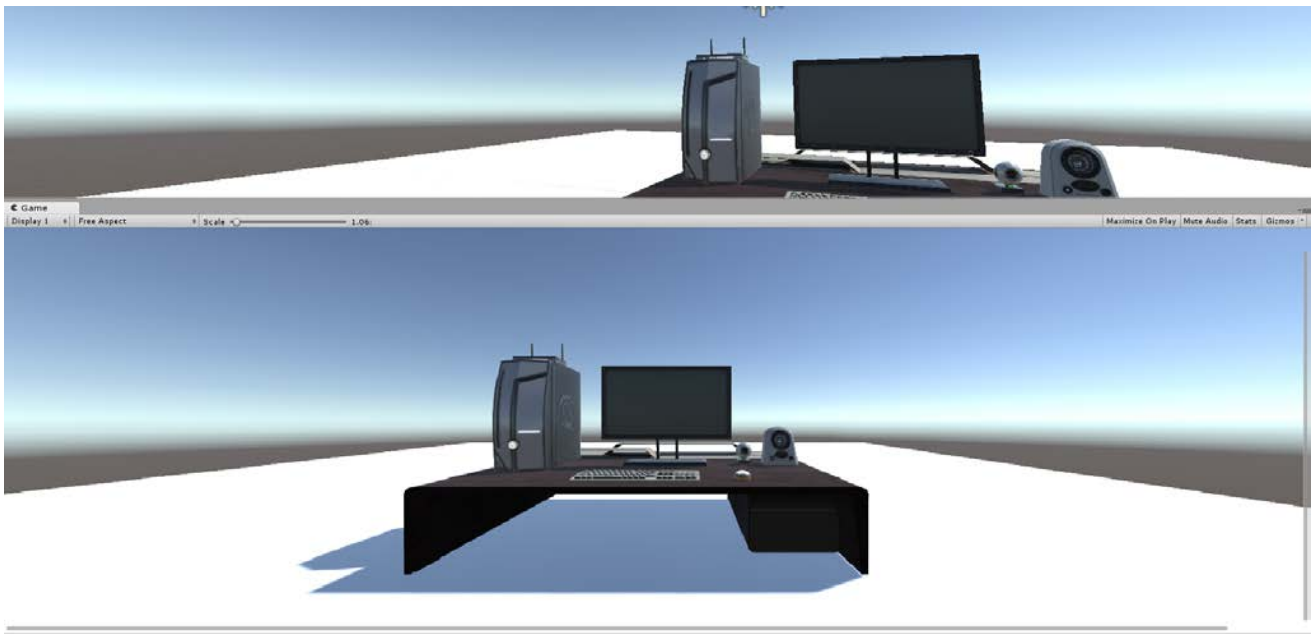
现在来到了操作系统部分，这也是可视化最重要的部分之一了，具体的引导开机过程详见附录的几篇笔记，这里不再赘述。

可视化

关于可视化

可视化部分我曾考虑过使用Processing等工具制作动画，这样效果也好，使用起来也方便，但是缺点在于重用性上面较差。做动画，虽然效果很好，但是不可复用，过程是死的，数据稍有变化就不能用了。因此封装可视化的组件，适应各种数据的变化才是可视化的目的。我的可视化页面的组件基本都可以复用，比如可视化内存布局的组件，按照我的数据格式写好，不同的数据都可以正常使用，可供下一级使用。

最初我用unity3D来进行可视化，效果挺好的，在进入Bootsect之前，开机自检部分表现的非常好，毕竟因为是3D引擎，但是之后的各种内存展现方面就很困难了，UGUI落后且编程效率低下，如果用这个来实现的话恐怕一整个学期也做不完可视化部分，因此最后使用vue前端实现可视化，做了个网页。下面是最初的unity可视化场景：



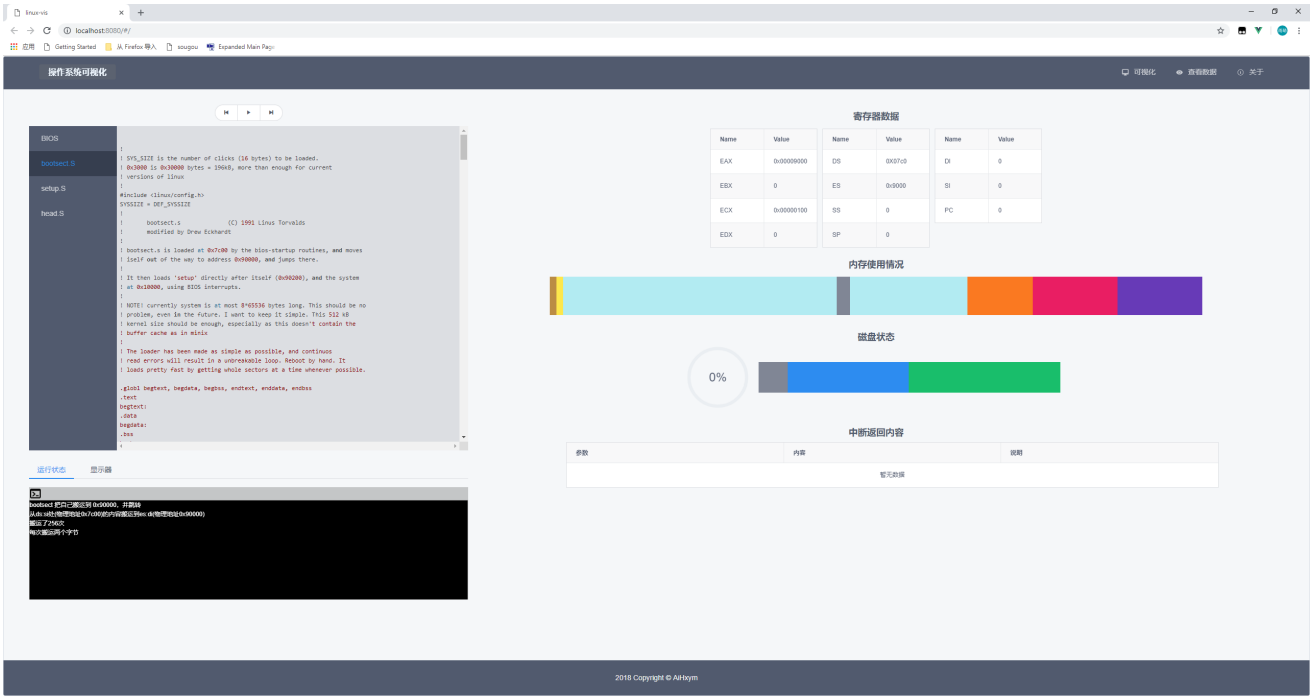
最终的可视化结果在下面的可视化介绍中会进行展示。

与上一级的对比

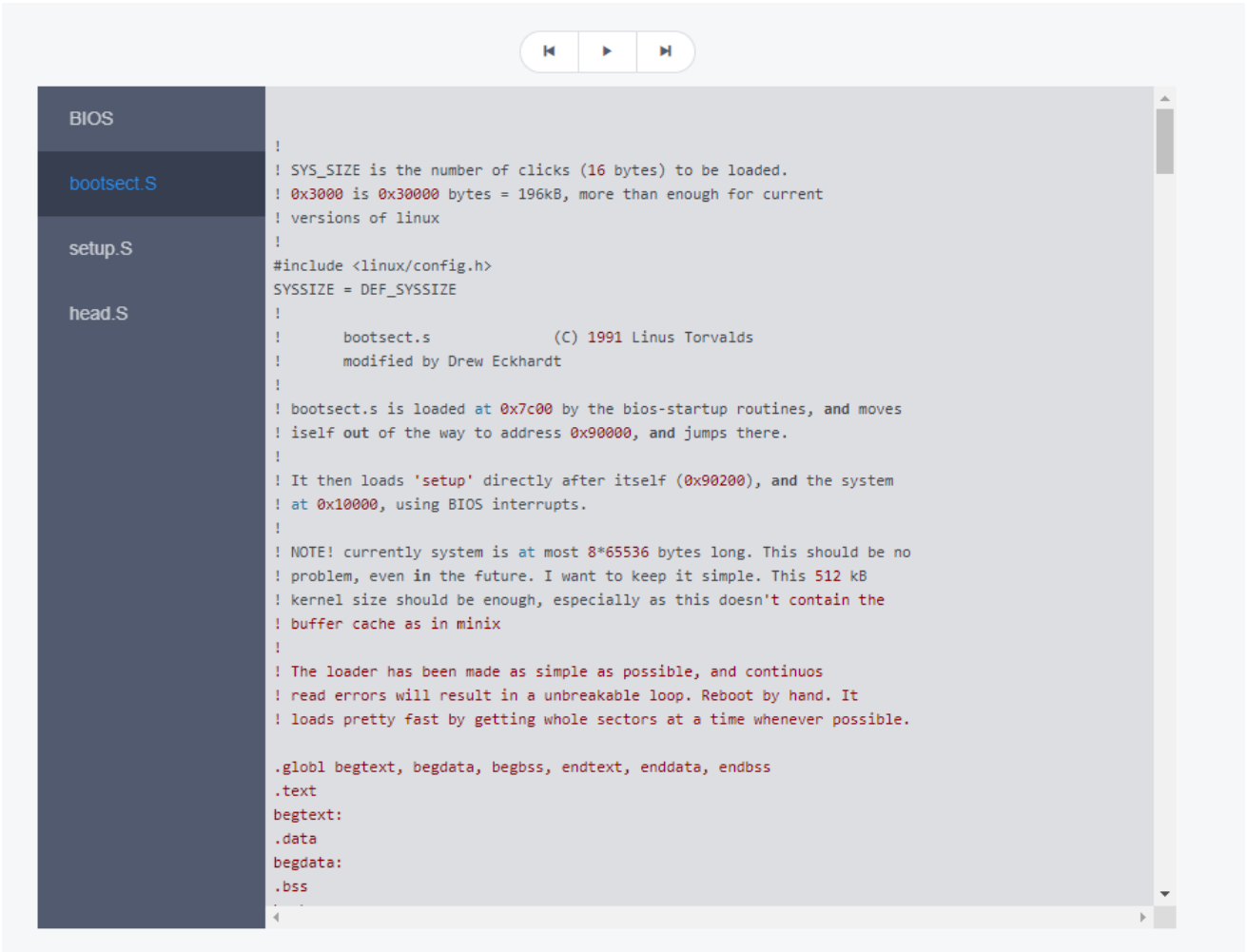
可视化部分由于我规范了数据，因此我需要做的就是实现标准的可视化组件，定义标准的接口，我的可视化系统只要是符合我数据规范的数据均能进行可视化展示，并且增加了寄存器内容可视化，以及十分费时费力但非常重要的中断返回内容可视化，并加入了返回内容说明，这是上一届同学没有做到的，界面更加美观，关键帧数目增加了很多。

可视化介绍

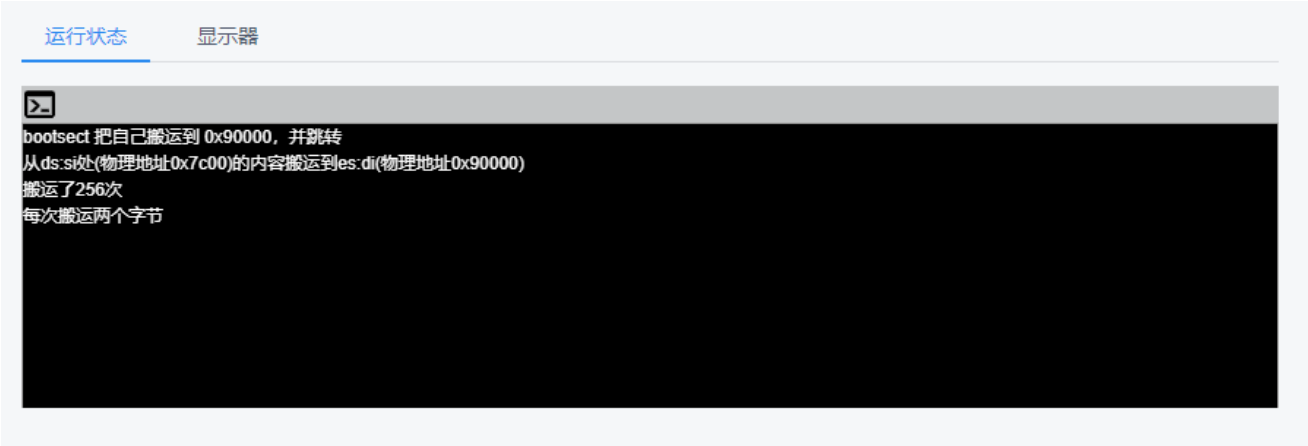
首先是整体布局，可以看到大致分为几个模块



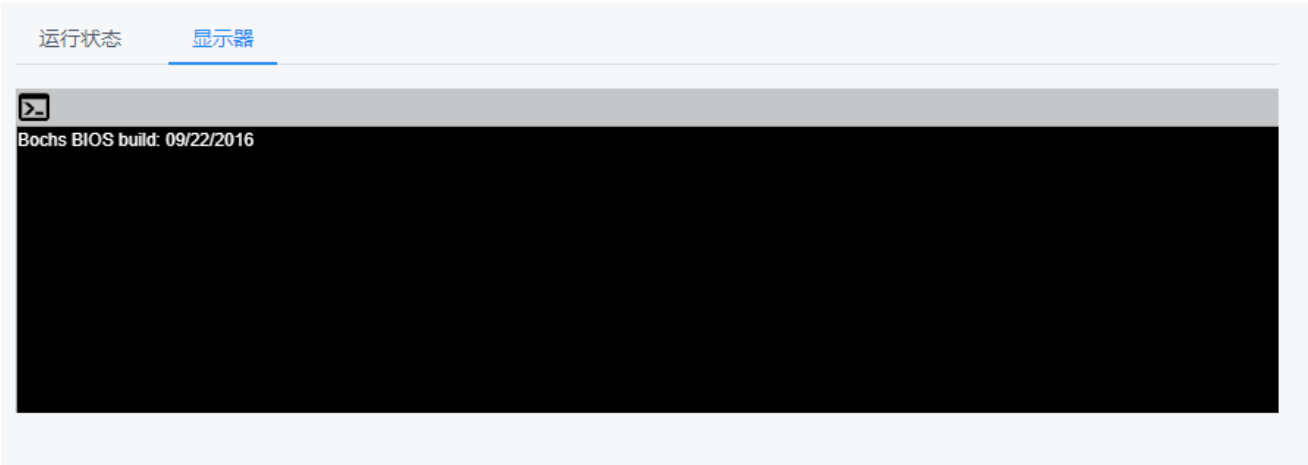
代码显示部分，有代码高亮，以及自动开始按钮，还有步展示按钮，点哪个代码部分就会跳到相应部分的可视化开始执行



运行状态是当前关键帧的具体动作的简介

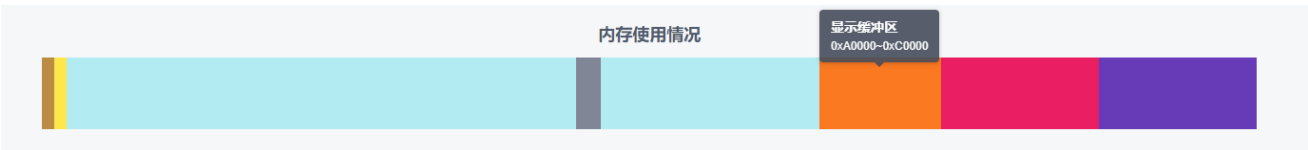


显示器则是虚拟机屏幕上的输出结果



寄存器数据是当前关键帧重要寄存器的状态

内存使用情况则是当前内存的布局，鼠标移动到某一块上方会显示当前这一块的相应信息，如下图，如果当前在进行搬运，还会看到相应动画，内存被一块一块搬运过去的动画



如下图，搬运动画某一帧截图

下面是磁盘状态，鼠标移动到某一块上会显示这块的信息，如果发生了磁盘读取操作，左侧进度环会显示读取进度，红框标注在读取哪一块

如果发生了中断，中断返回内容会更新，显示当前中断返回的内容，以及相关内容的说明

中断返回内容

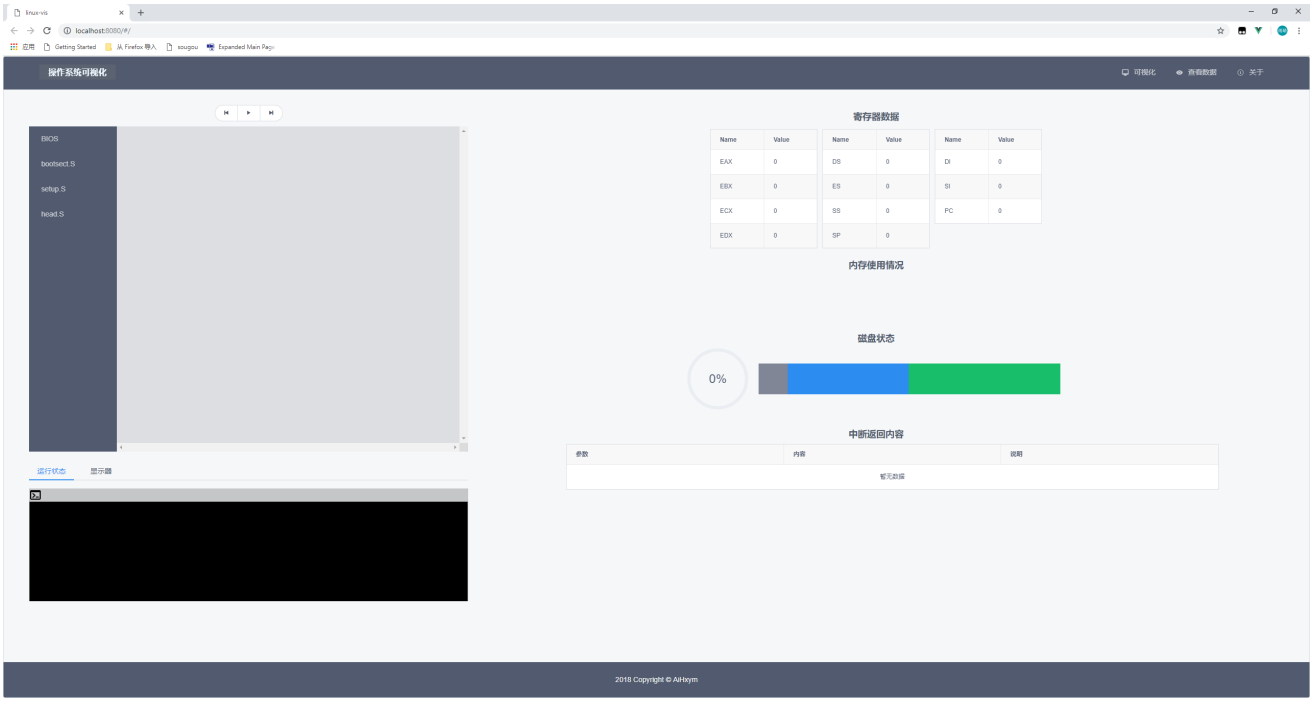
参数	内容	说明
CF	0x0000	=0，操作成功；=1，操作失败。
AH	0x0000	错误返回码
BL	0x0004	驱动器类型
CH	0x0009	最大磁道号的[7:0]
CL[7:6]	0x4	最大磁道号的[9:8]
CL[5:0]	0xf12	每磁道最大扇区数
DH	0x0000	最大磁头数
DL	0x0102	驱动器数量
ES:DI	0xf000:0xefde	指向软驱磁盘参数表

网页中直接查看数据

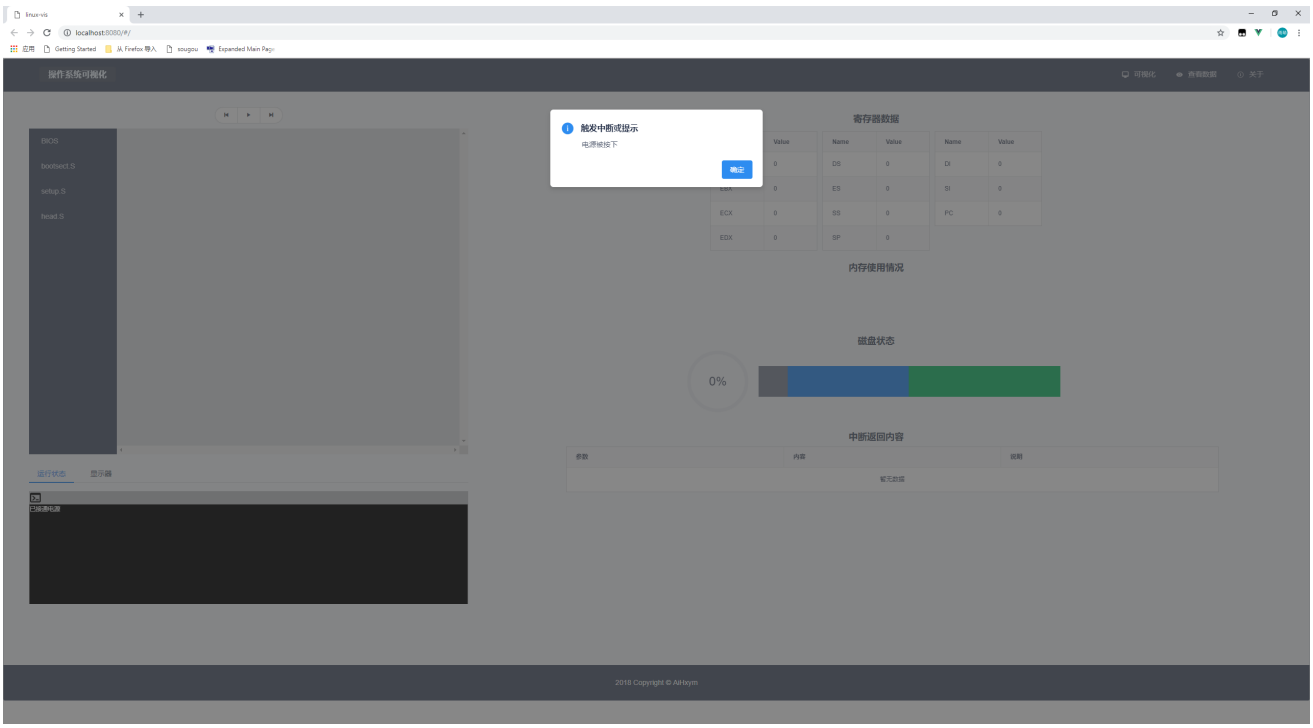

linux-vis
localhost:8080/#/datas
应用
Getting Started
从 Firefox 导入
sougou
Expanded Main Page
操作系统可视化
可视化
查看数据
关于
{
"BIOS": [...], // count: 9
"bootsect": [
{
"id" : "1",
"event" : "carryAndJump",
"ram": [
{
"size" : "1%",
"name" : "中断向量表",
"addr" : "0x00000~0x00400",
"color" : "#BB8C44"
},
{
"size" : "1%",
"name" : "BIOS数据区",
"addr" : "0x00400~0x00500",
"color" : "#FDE74C"
},
{
"size" : "5%",
"name" : "Unknown1",
"addr" : "",
"color" : "#B2EBF2"
},
{
"size" : "2%",
"name" : "bootsect",
"addr" : "0x07C00~0x07E00",
"color" : "#808695"
},
{
"size" : "35%",
"name" : "Unknown2",
"addr" : "",
"color" : "#B2EBF2"
},
{
"size" : "0%",
"name" : "bootsect(搬运后)",
"addr" : "0x90000~0x90200",
"color" : "#808695"
},
{

实验结果

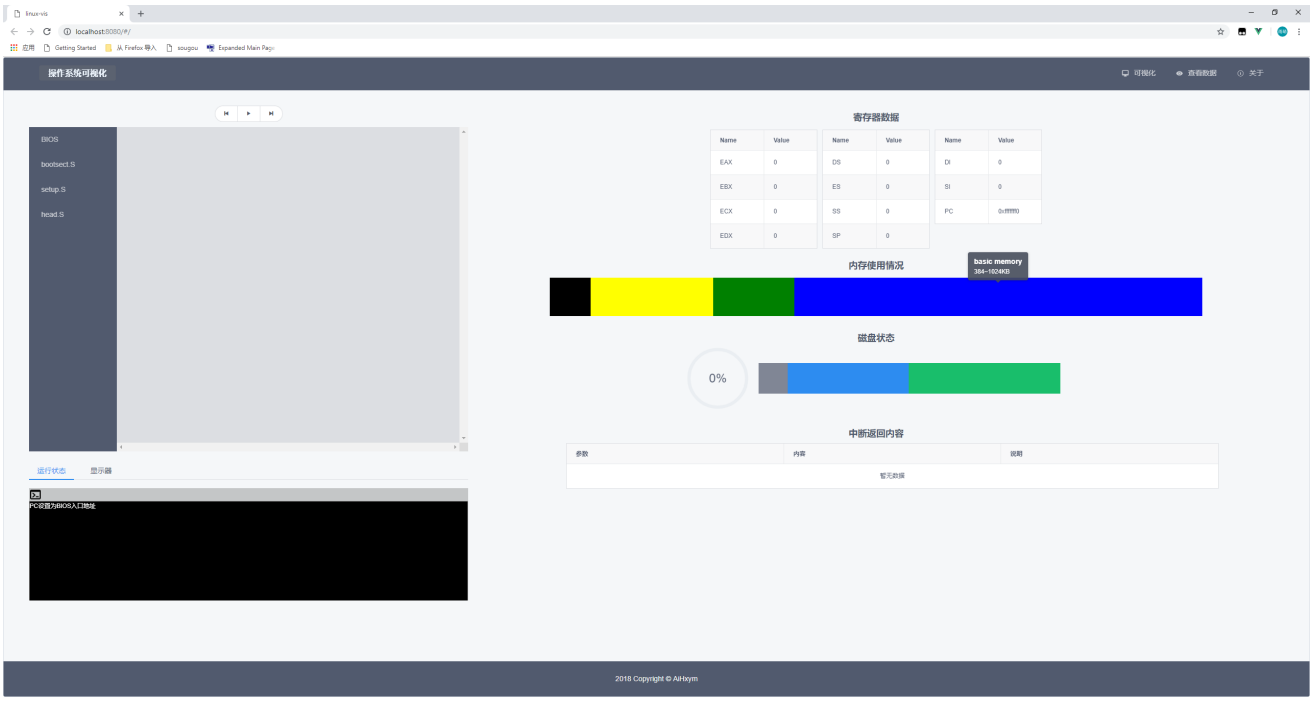
初始状态



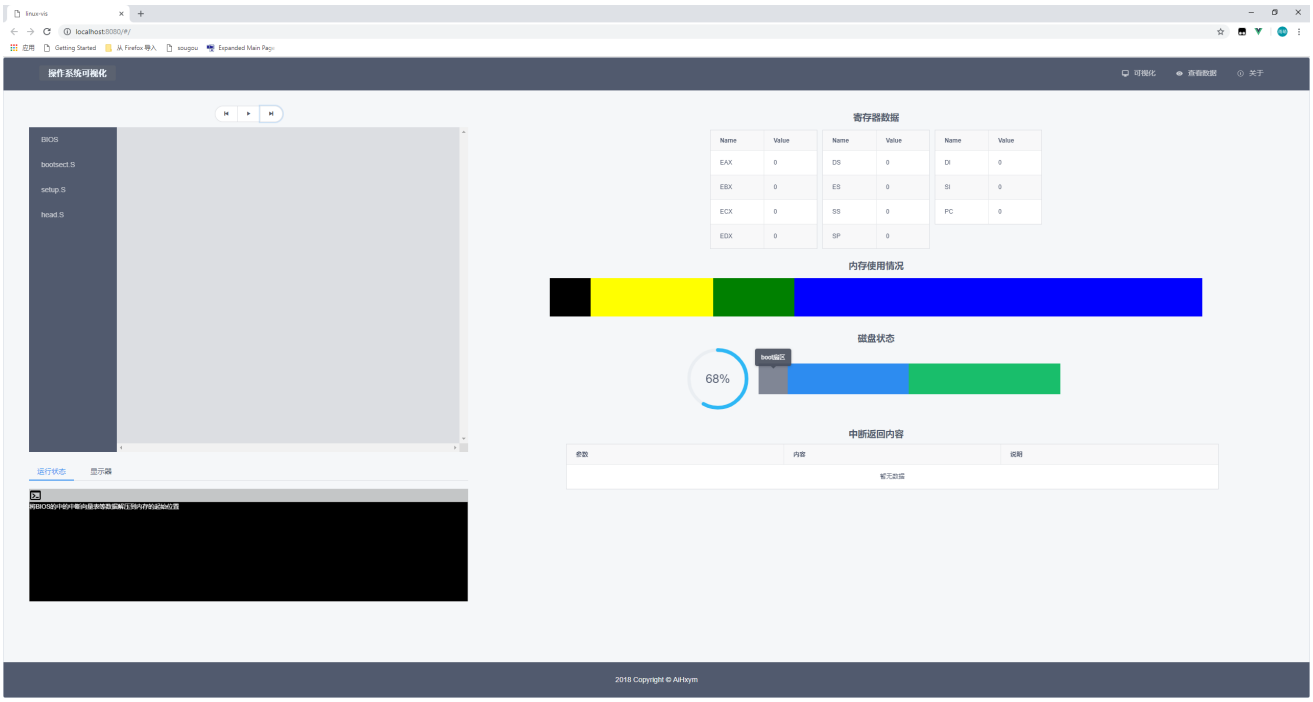
电源键被按下



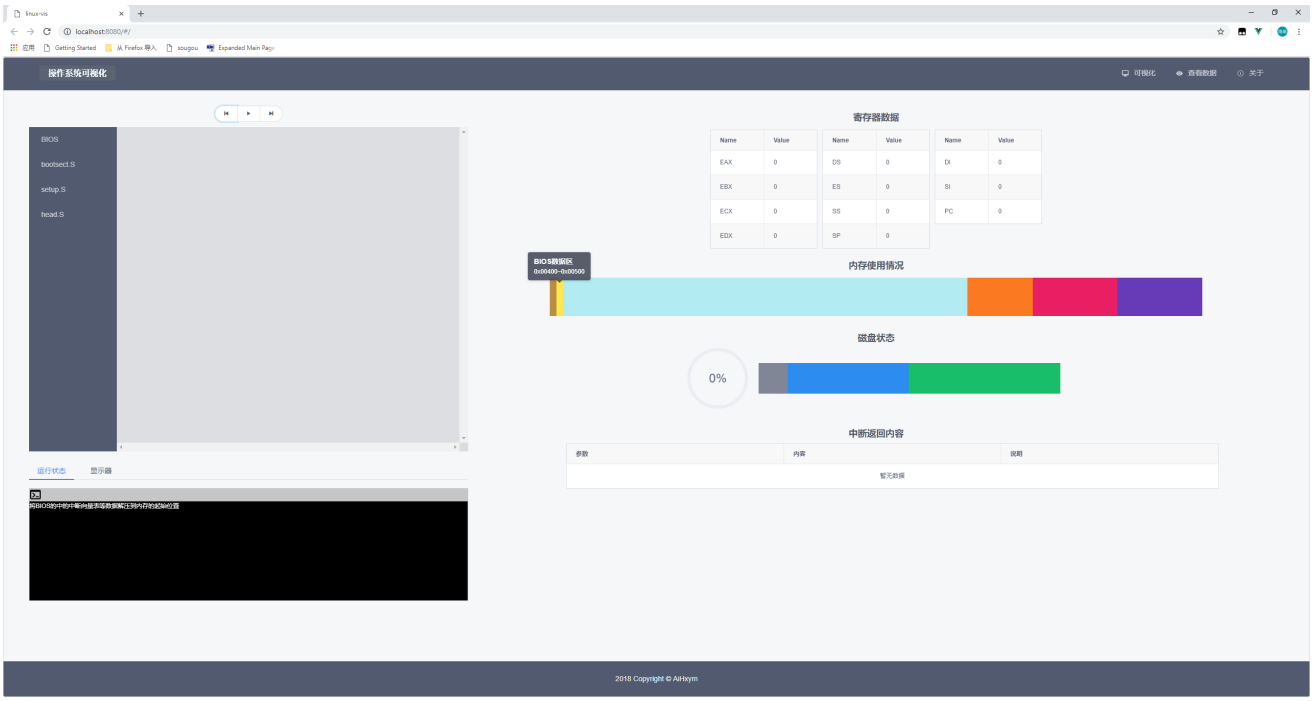
CPU Reset，PC设置为BIOS入口地址，内存布局变化



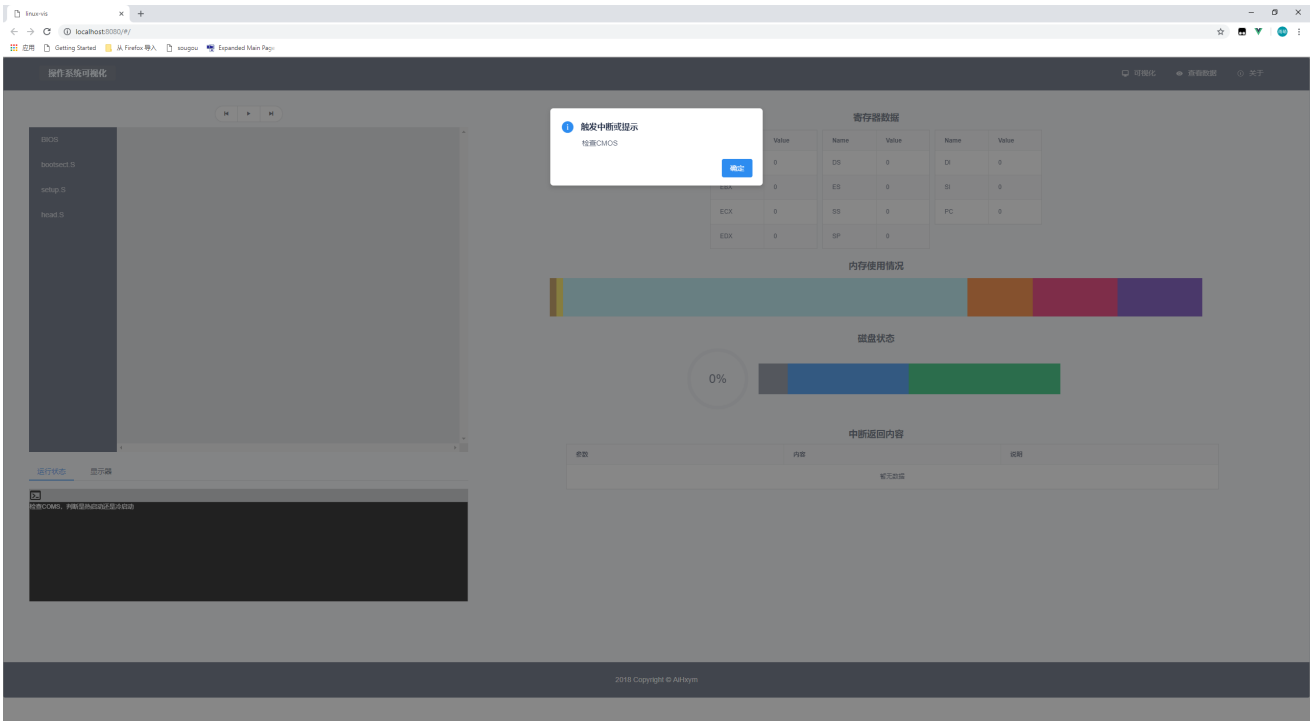
解压BIOS中的数据区和中断向量表等



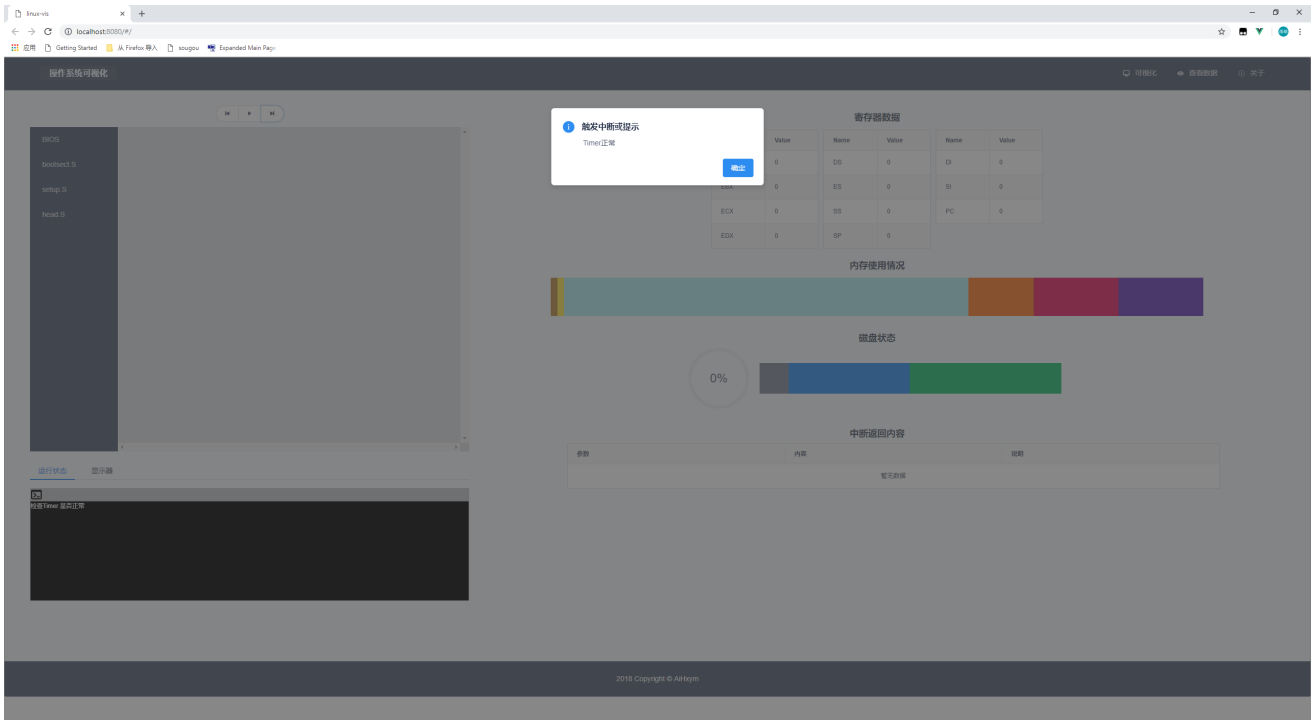
解压完毕后内存布局的变化



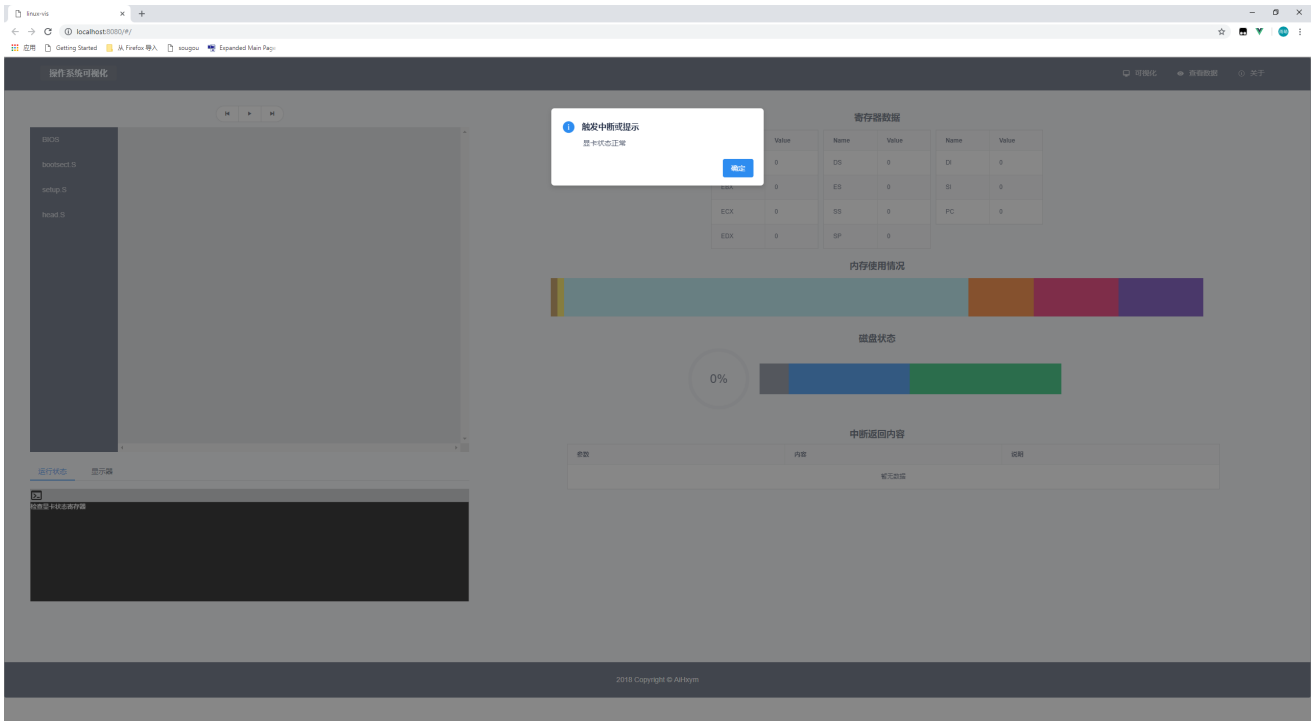
检查CMOS



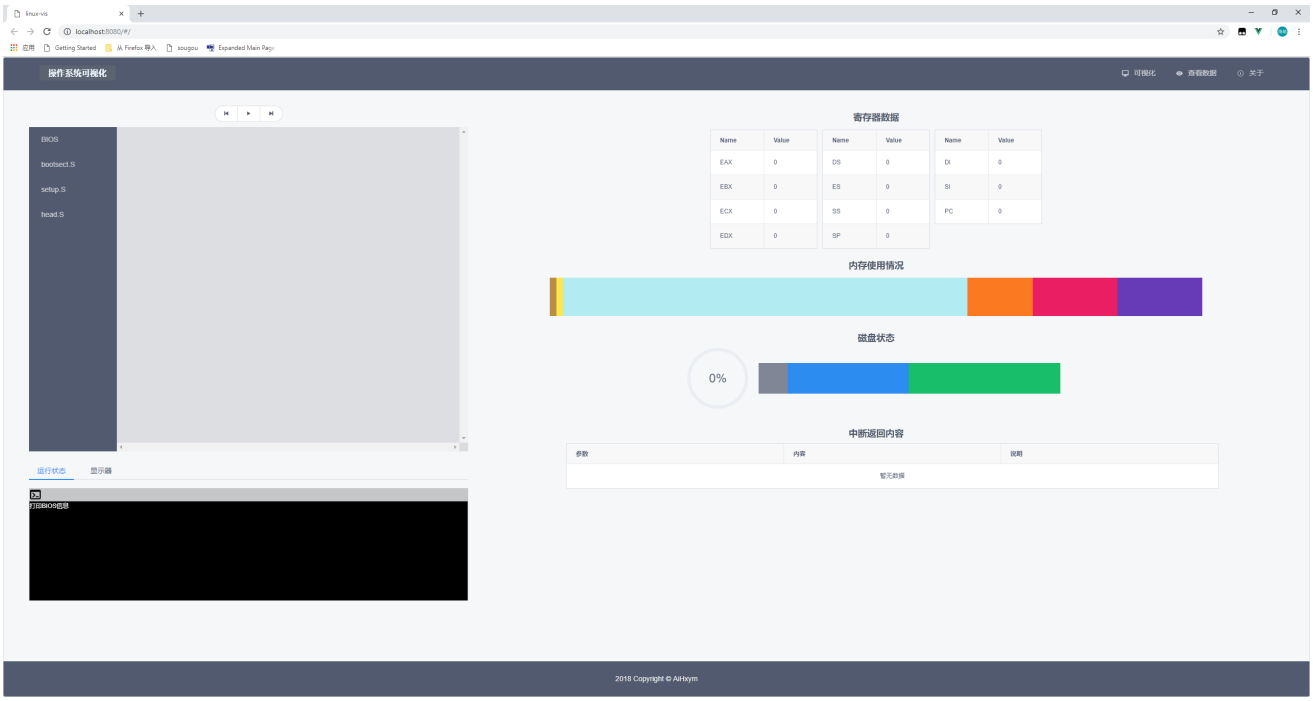
检查Timer



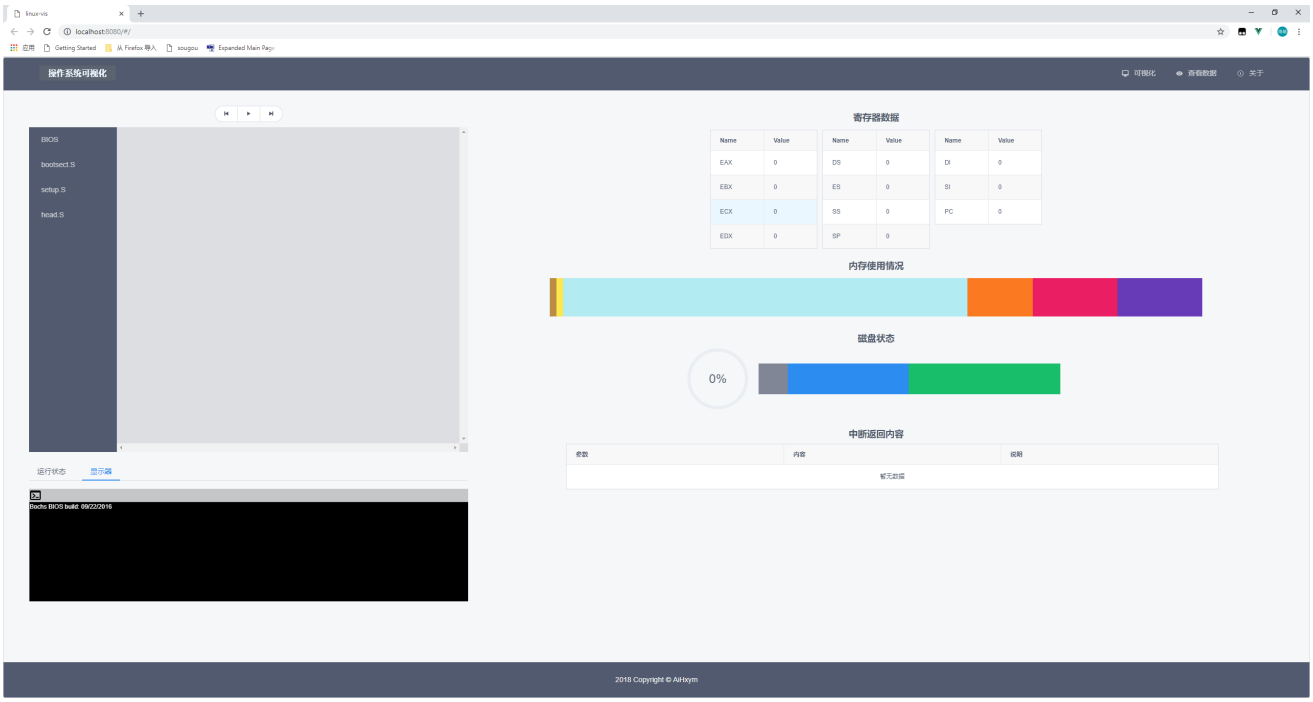
检查显卡状态寄存器



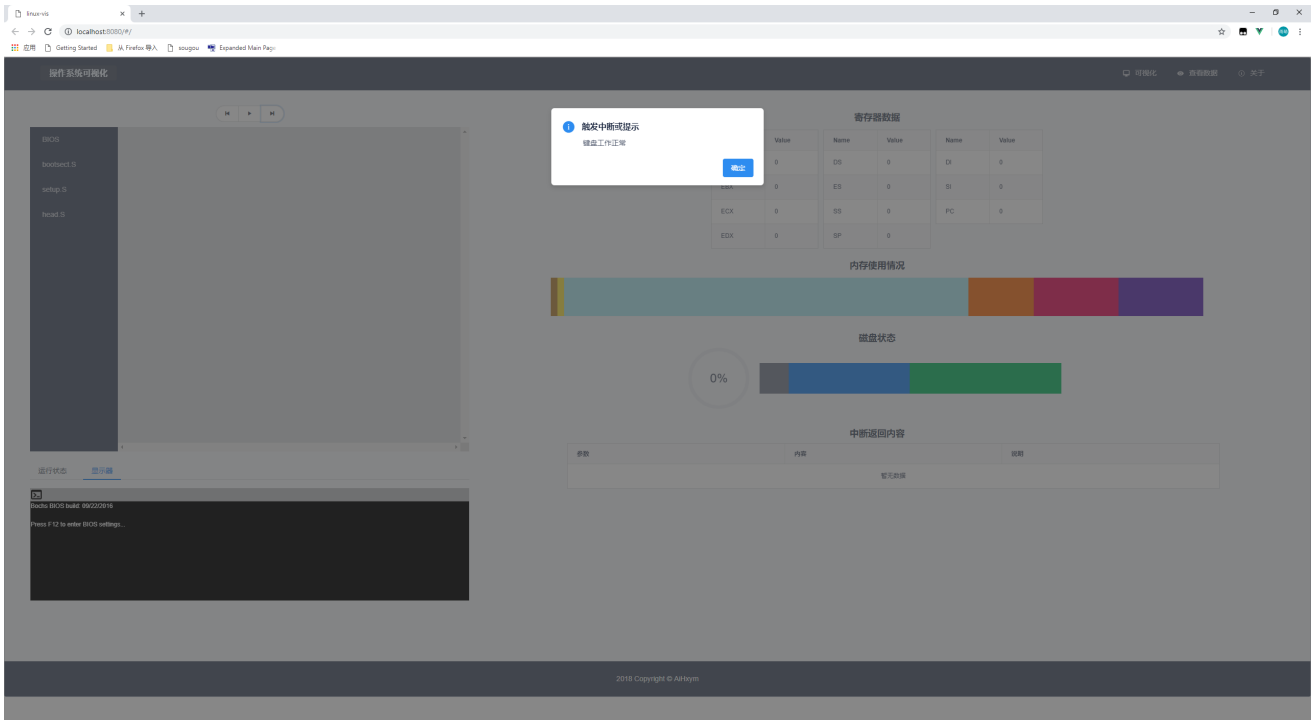
打印BIOS信息



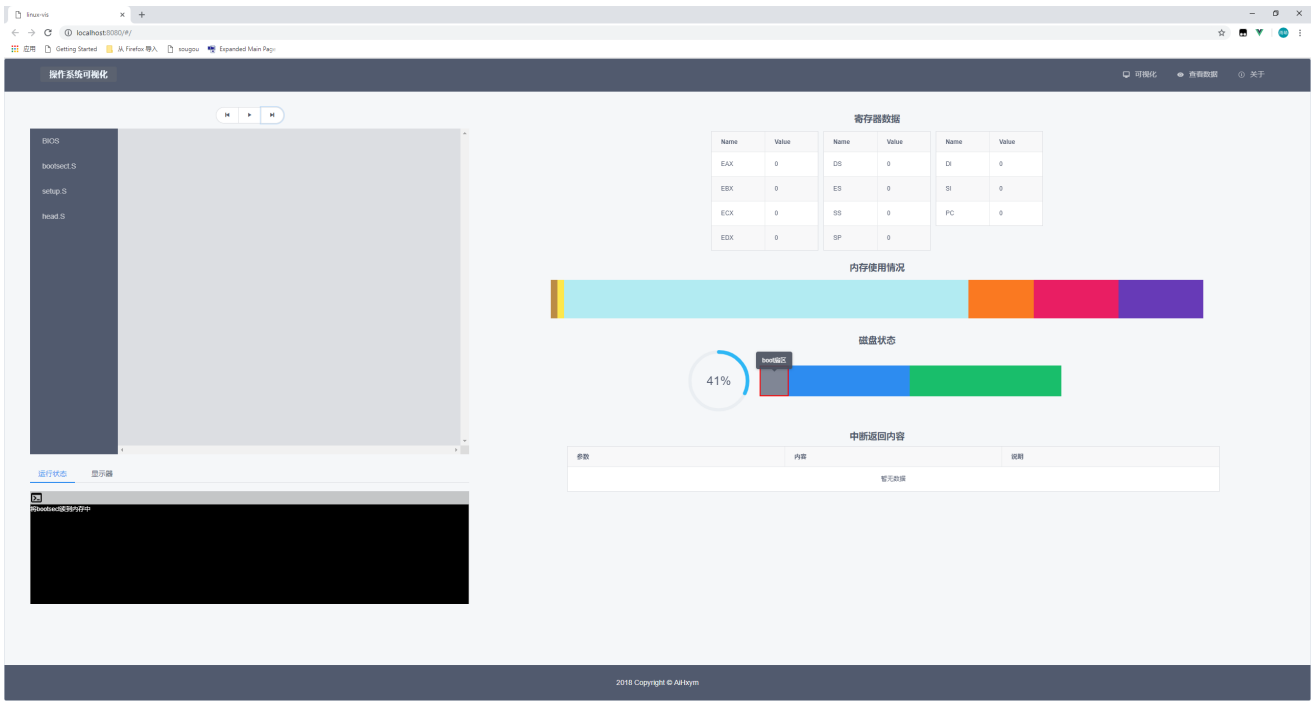
打印后的显示器



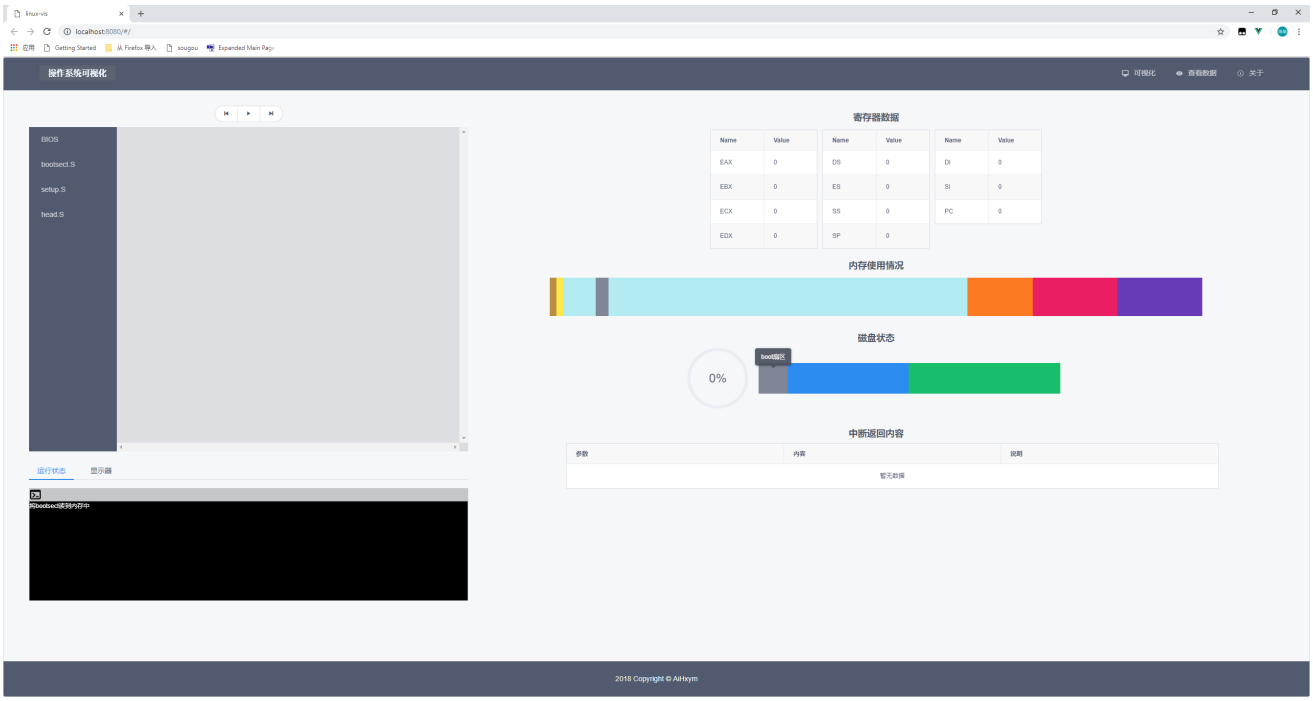
检查键盘



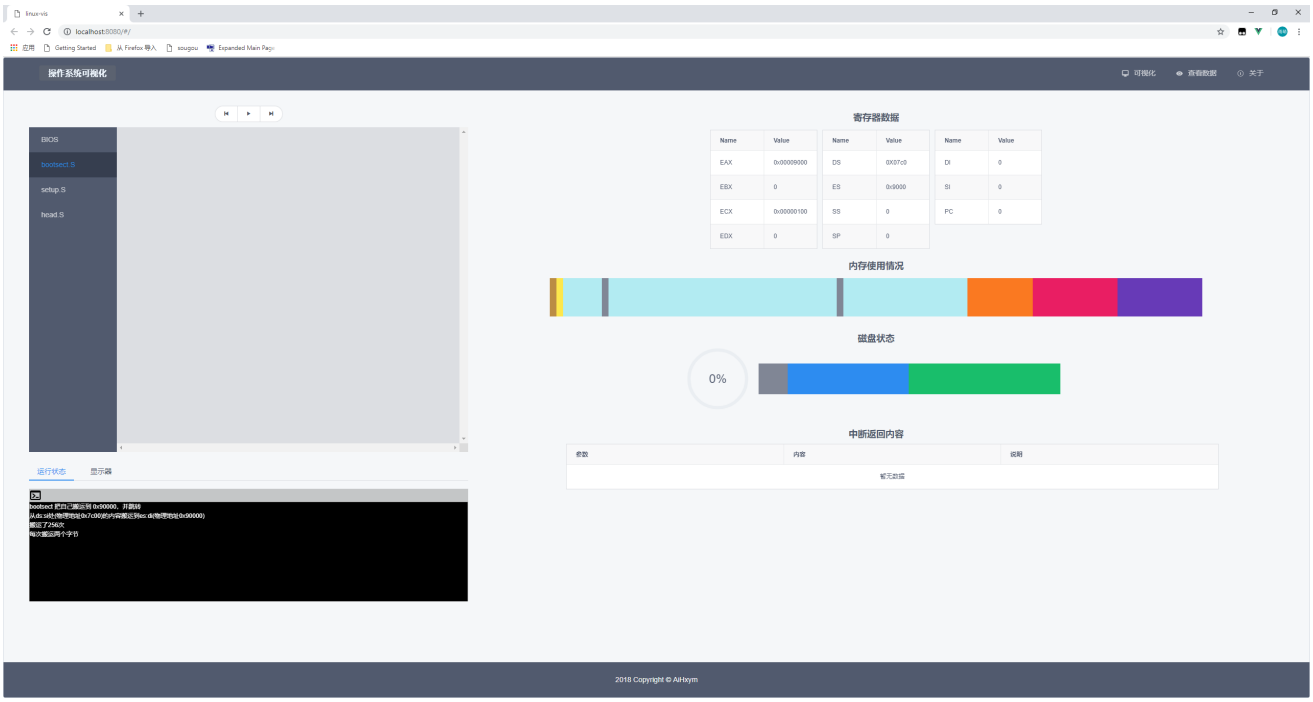
读取bootsect.S，注意进度环和红框区域



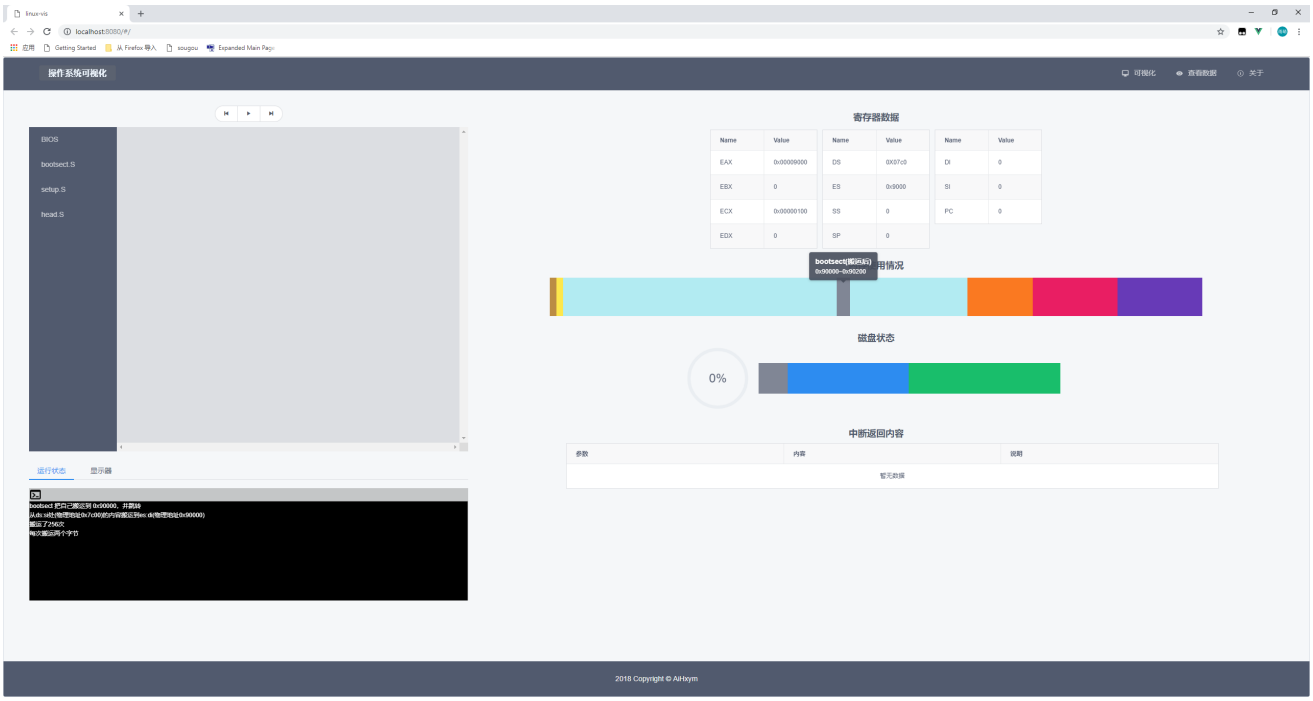
加载完毕bootsect.S后的内存布局



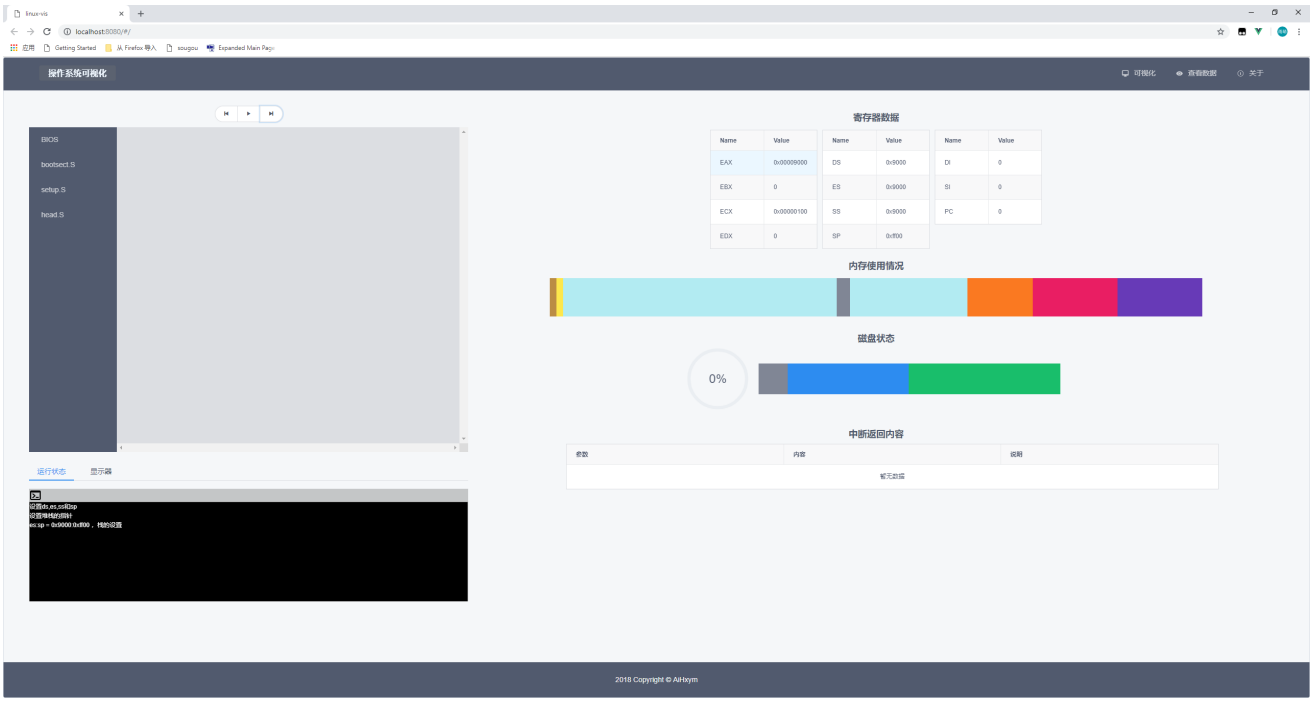
bootsect 把自己搬运到 0x90000，并跳转



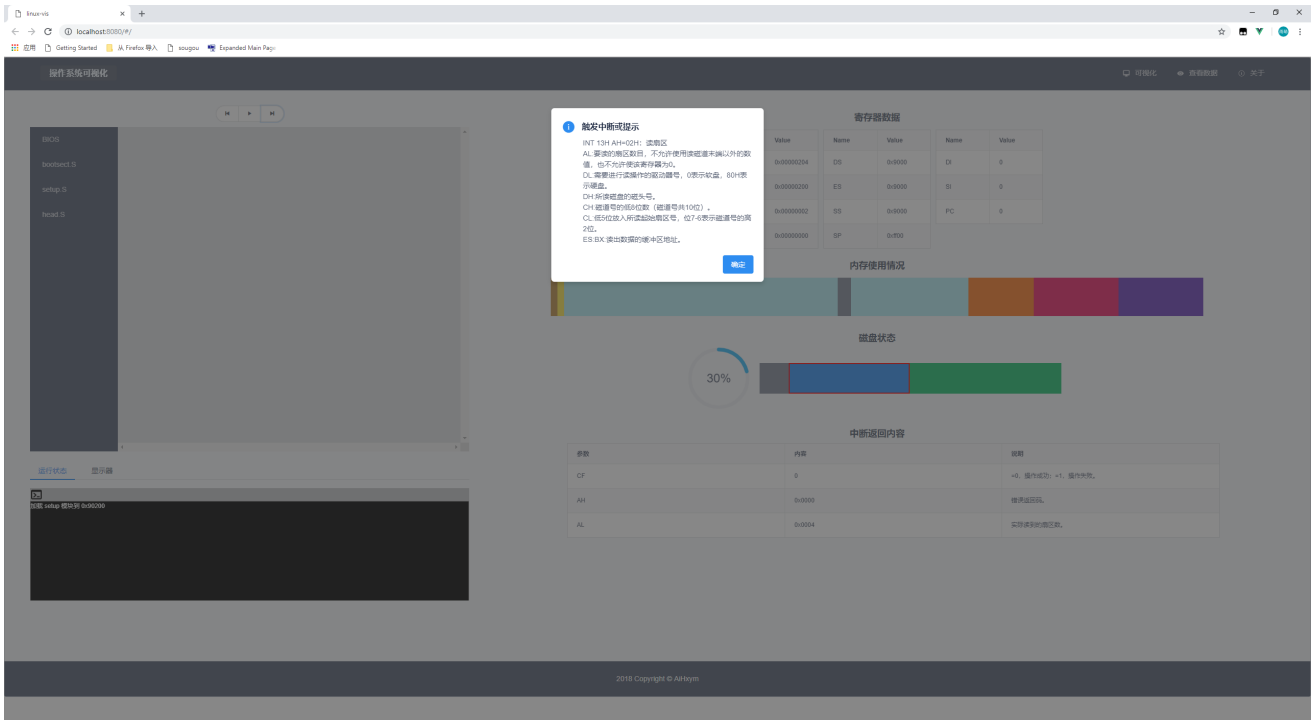
搬运后状态



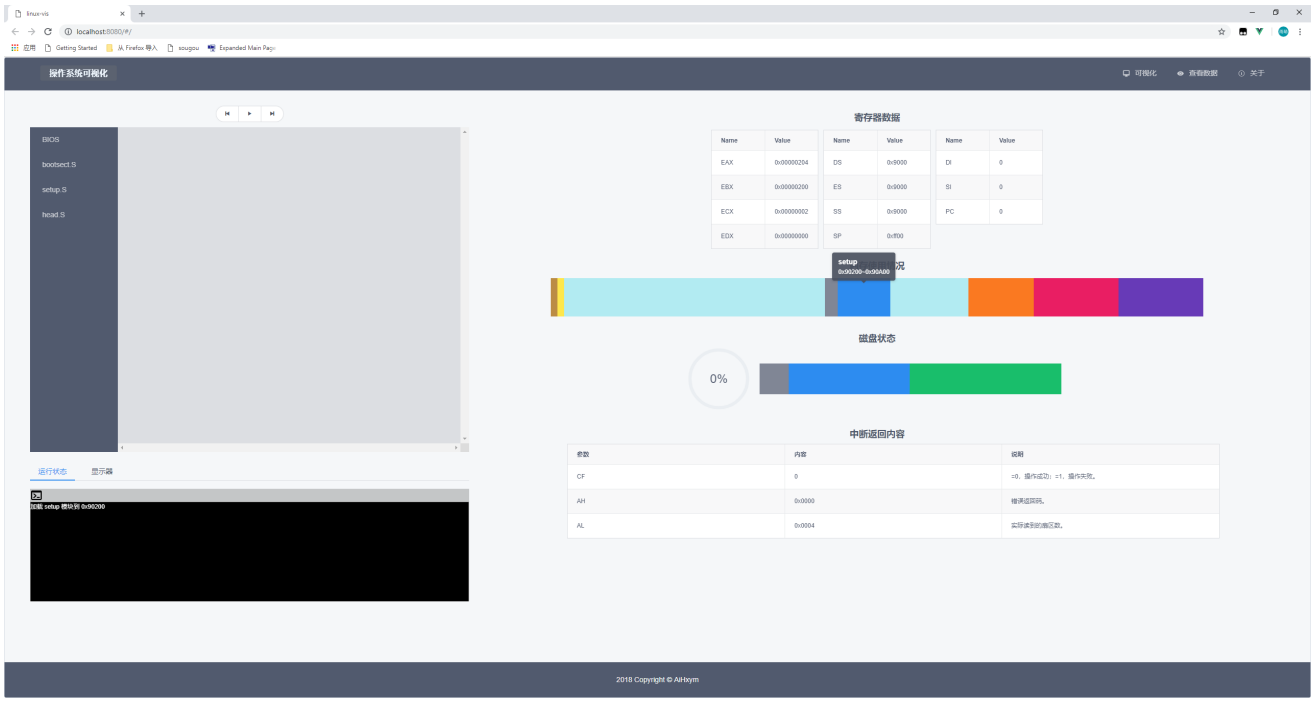
设置设置ds,es,ss和sp 设置堆栈的指针（注意寄存器变化）



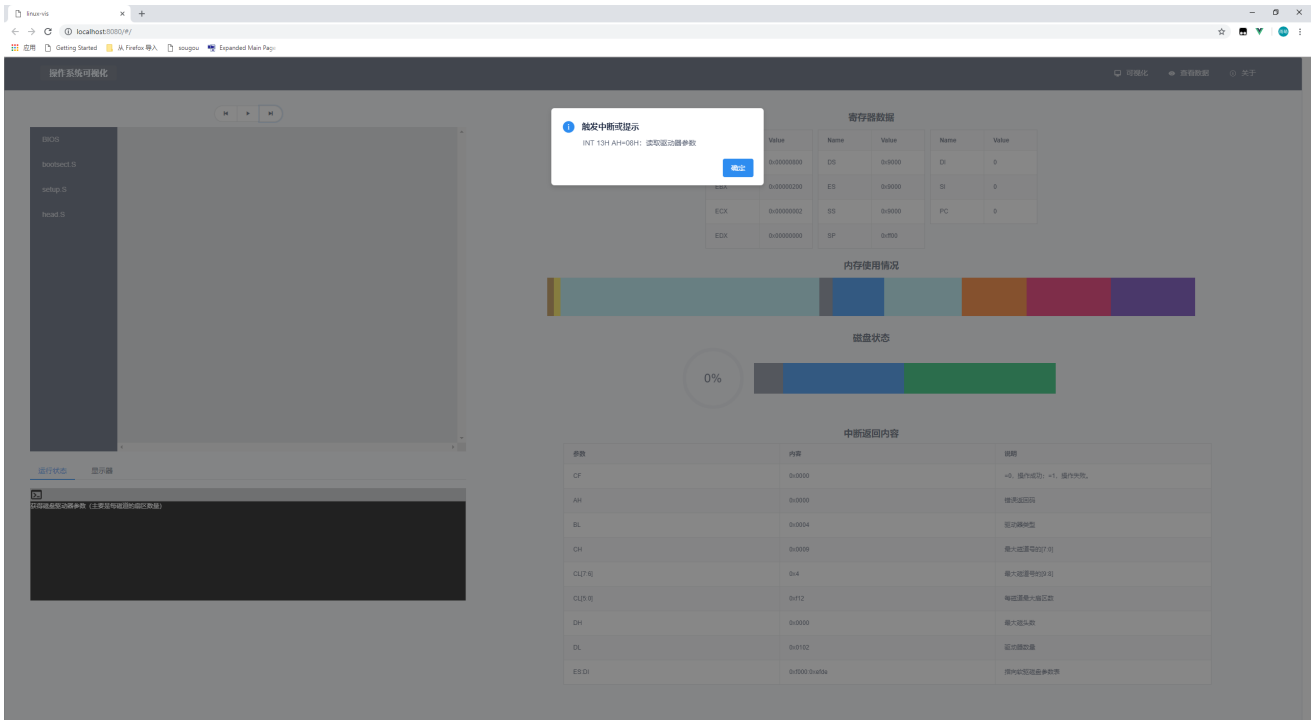
触发读扇区中断，读取setup.S



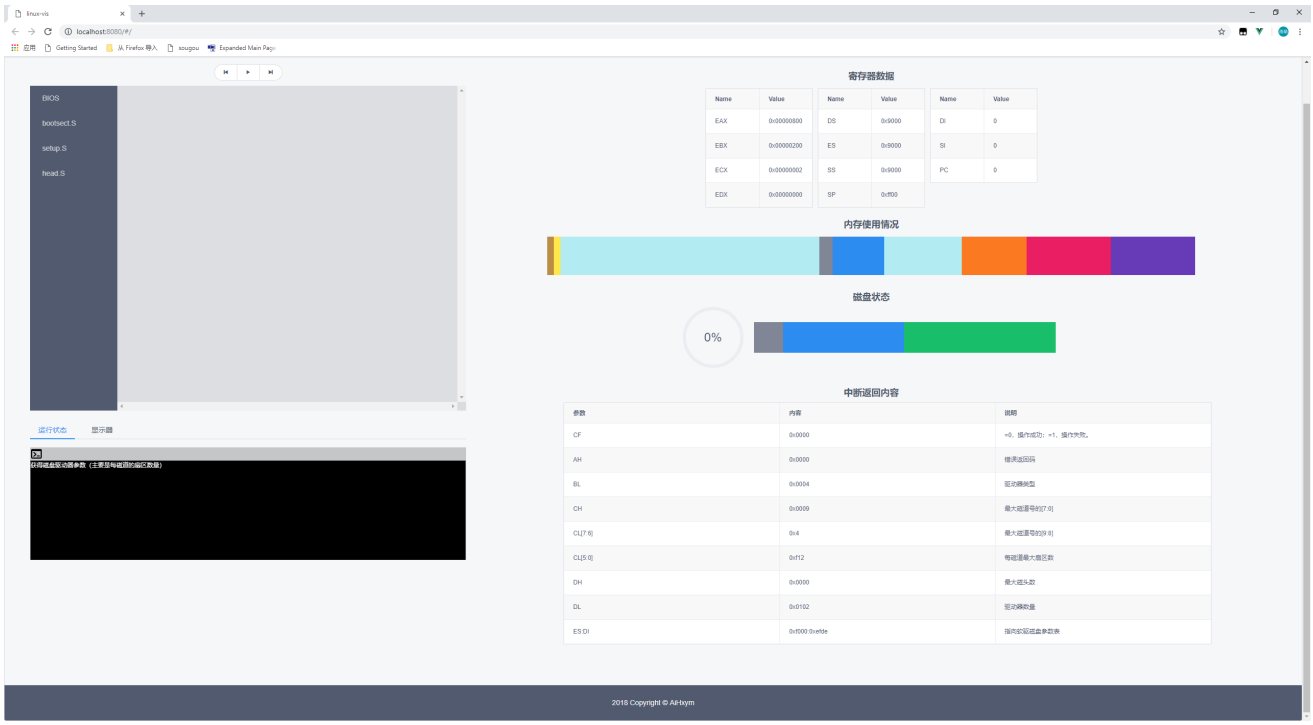
可以看到中断返回后的结果



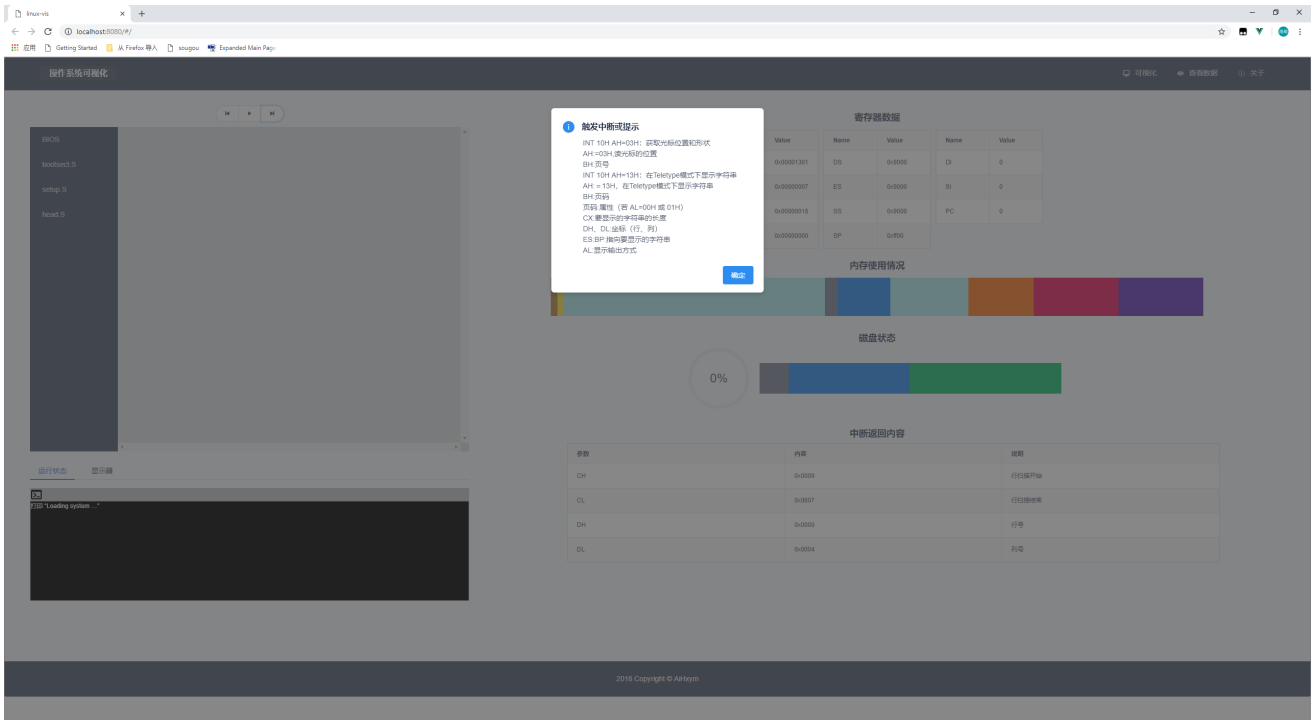
触发中断。读取驱动器参数



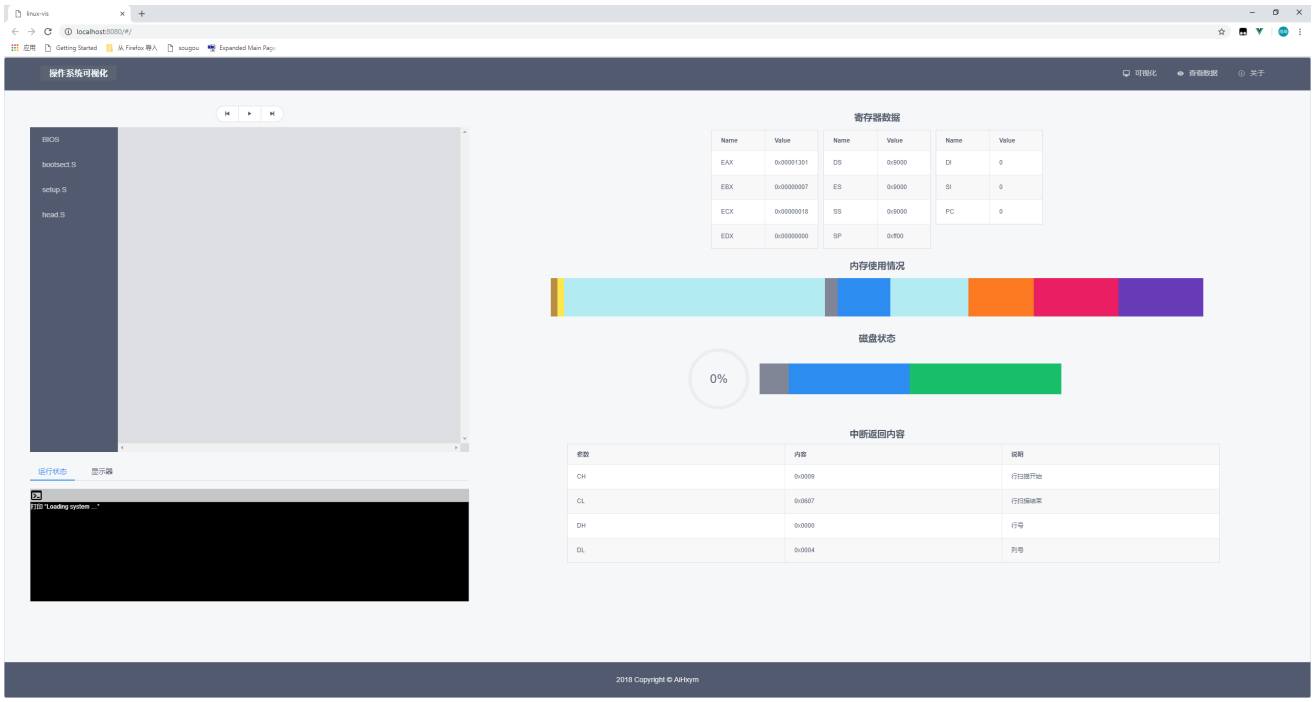
可以看到中断返回后的结果



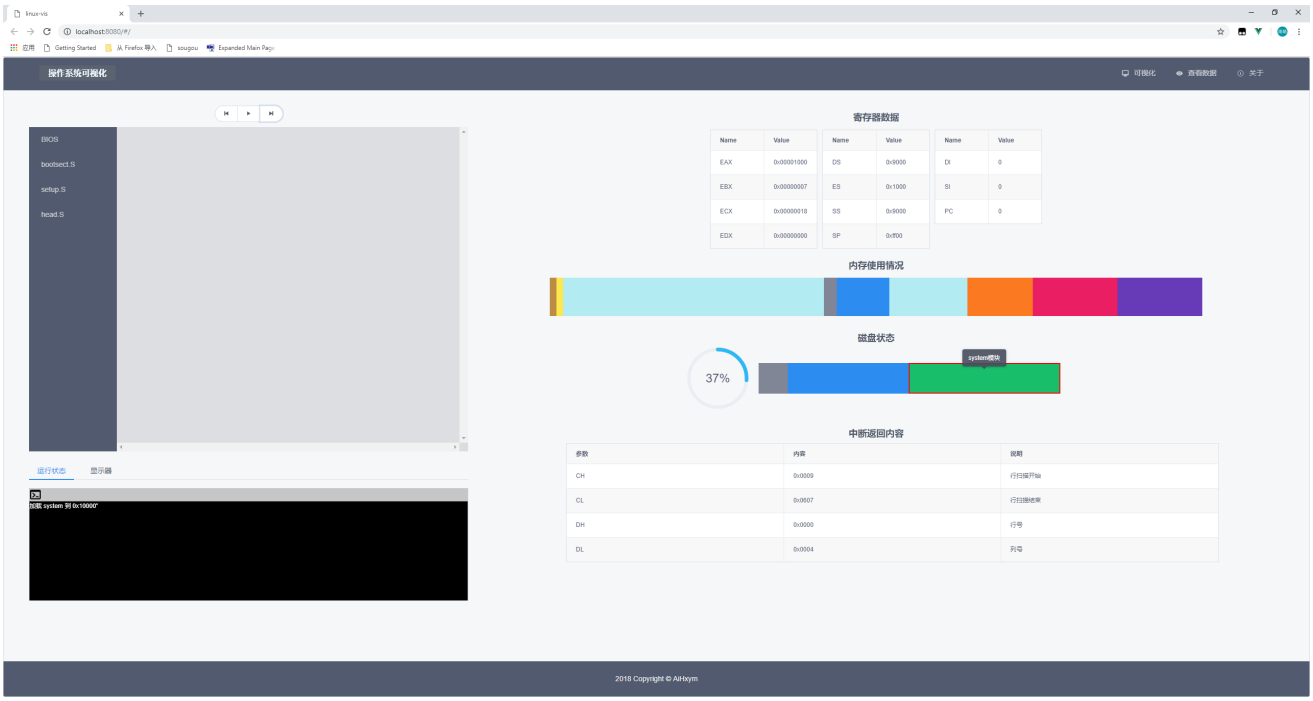
打印 “Loading system ...”，触发了中断



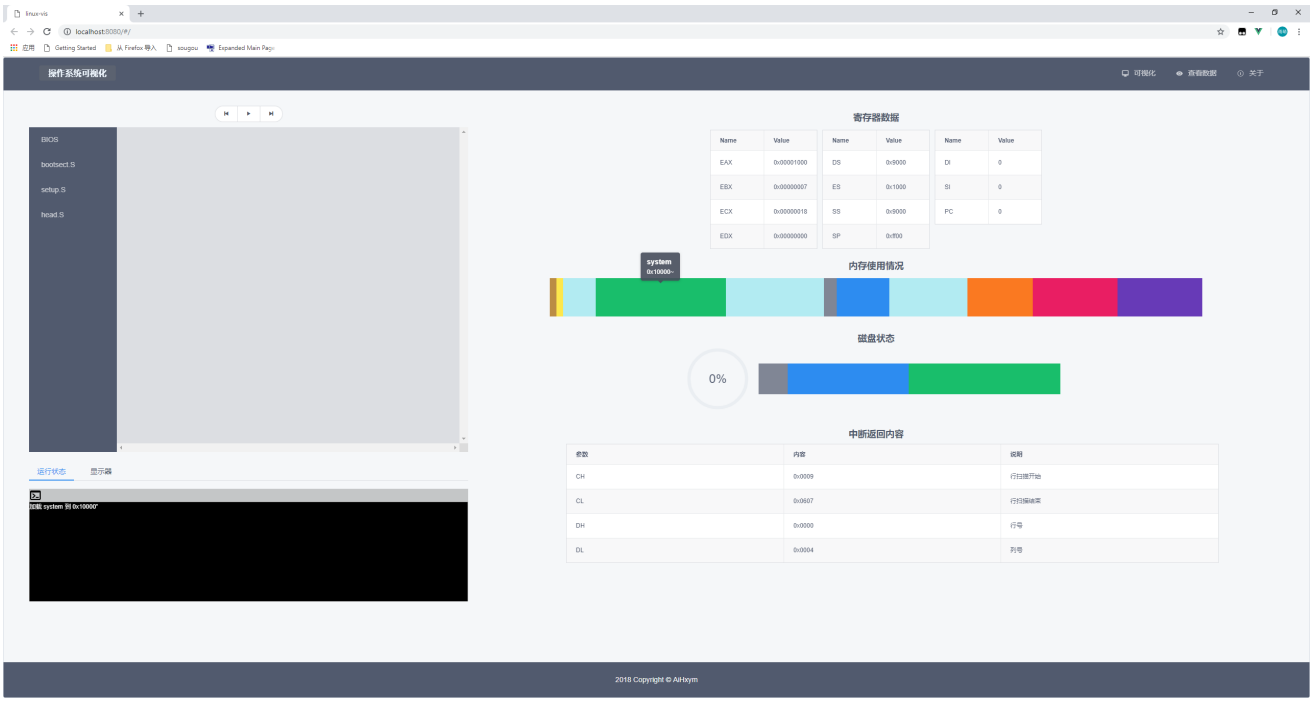
可以看到中断返回后的结果



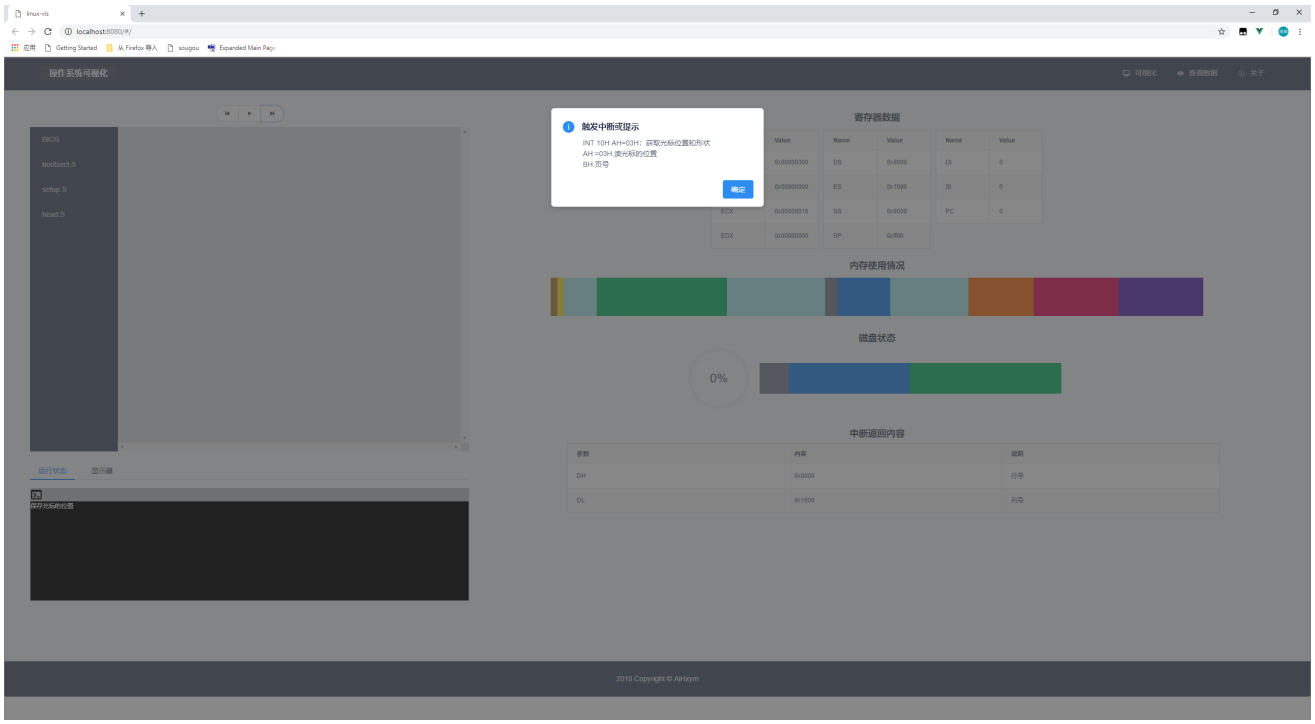
加载 system 到 0x10000



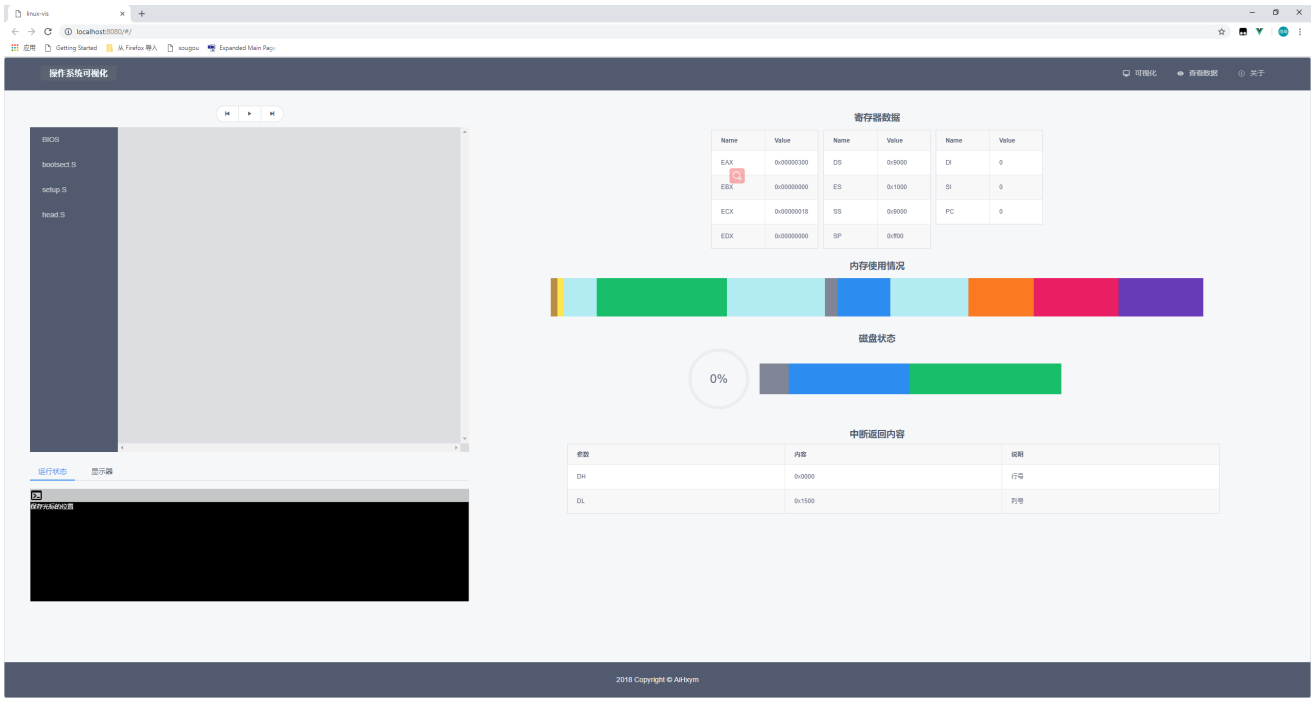
加载后的内存布局



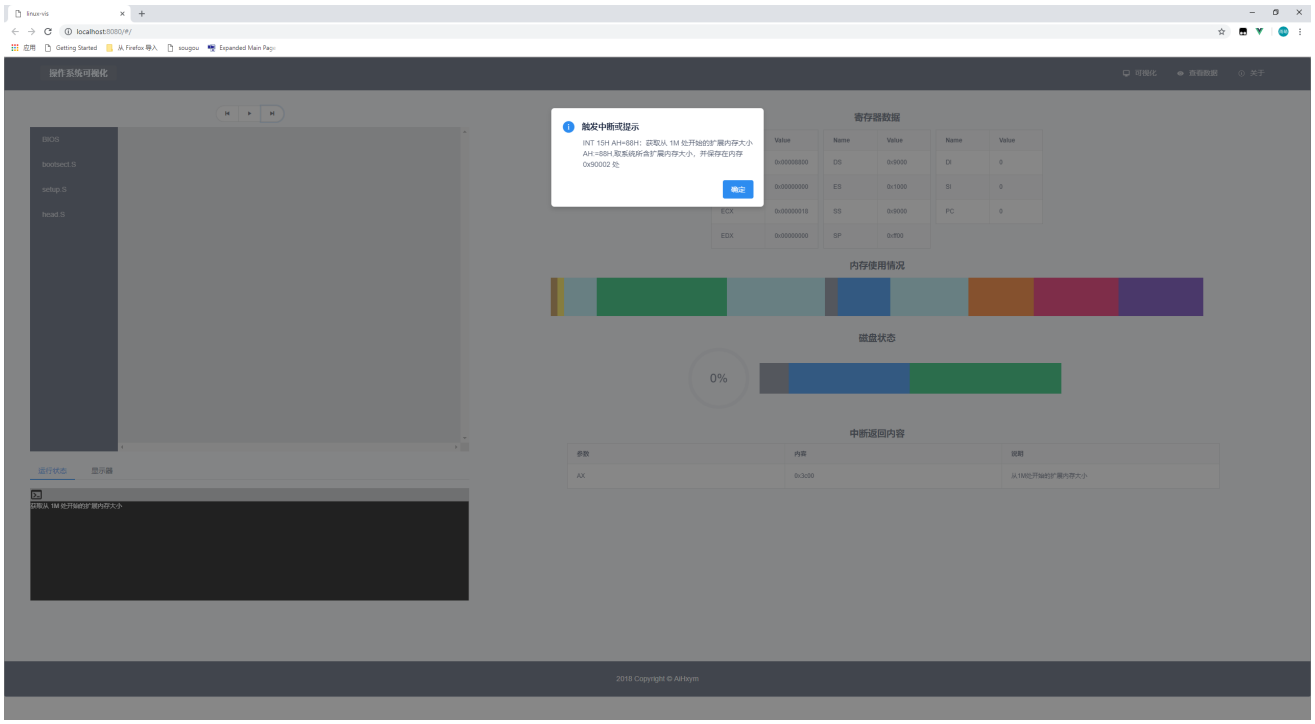
触发中断，保存光标位置



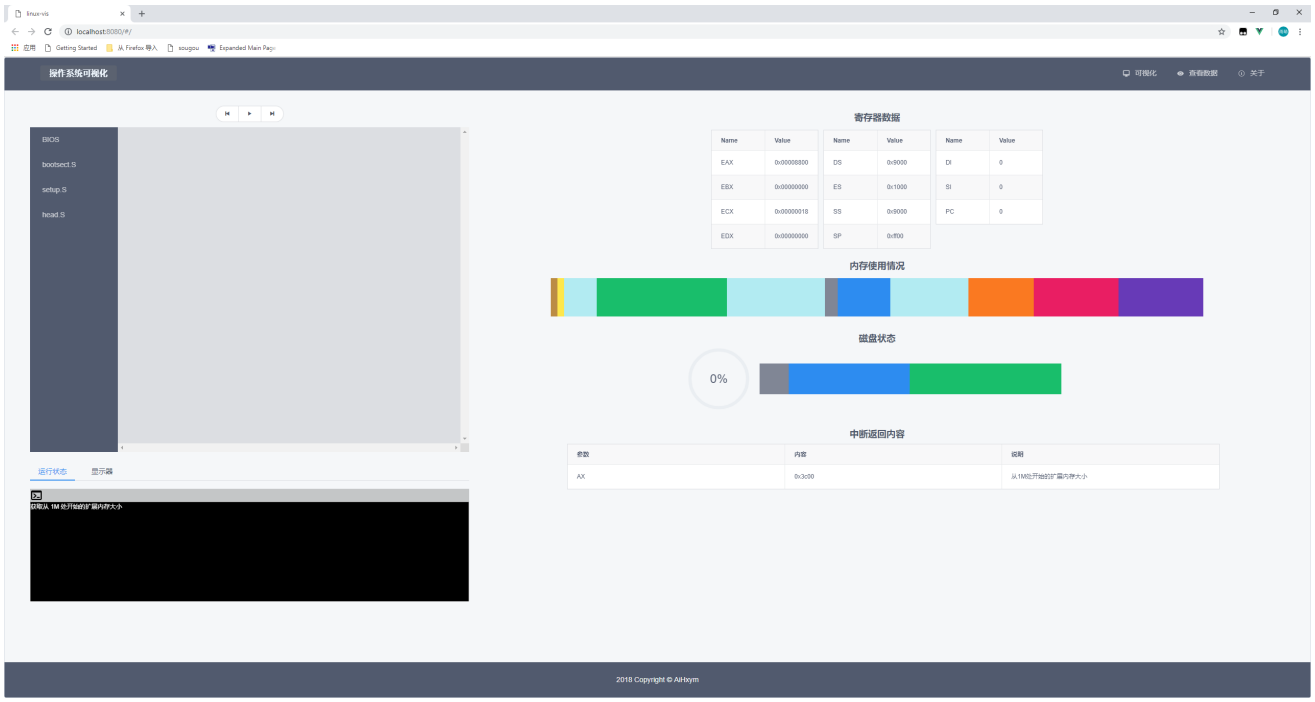
可以看到中断返回后的结果



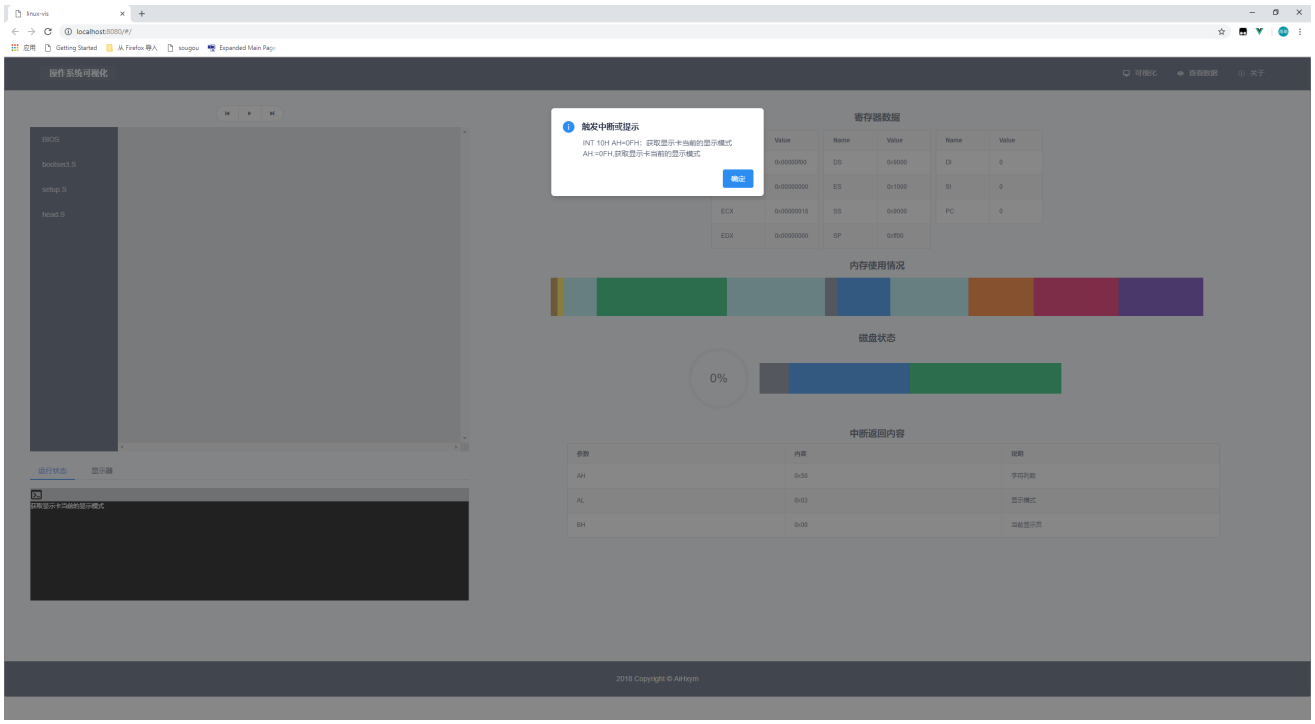
获取从 1M 处开始的扩展内存大小



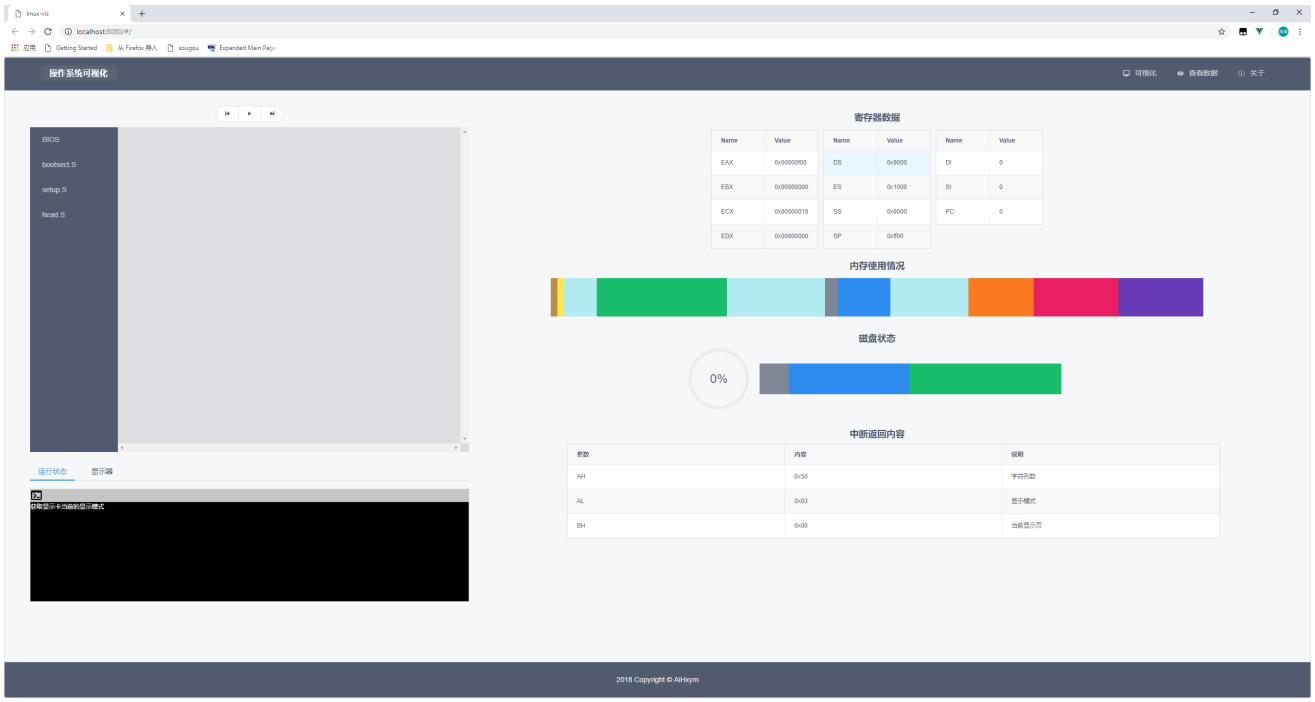
可以看到中断返回后的结果



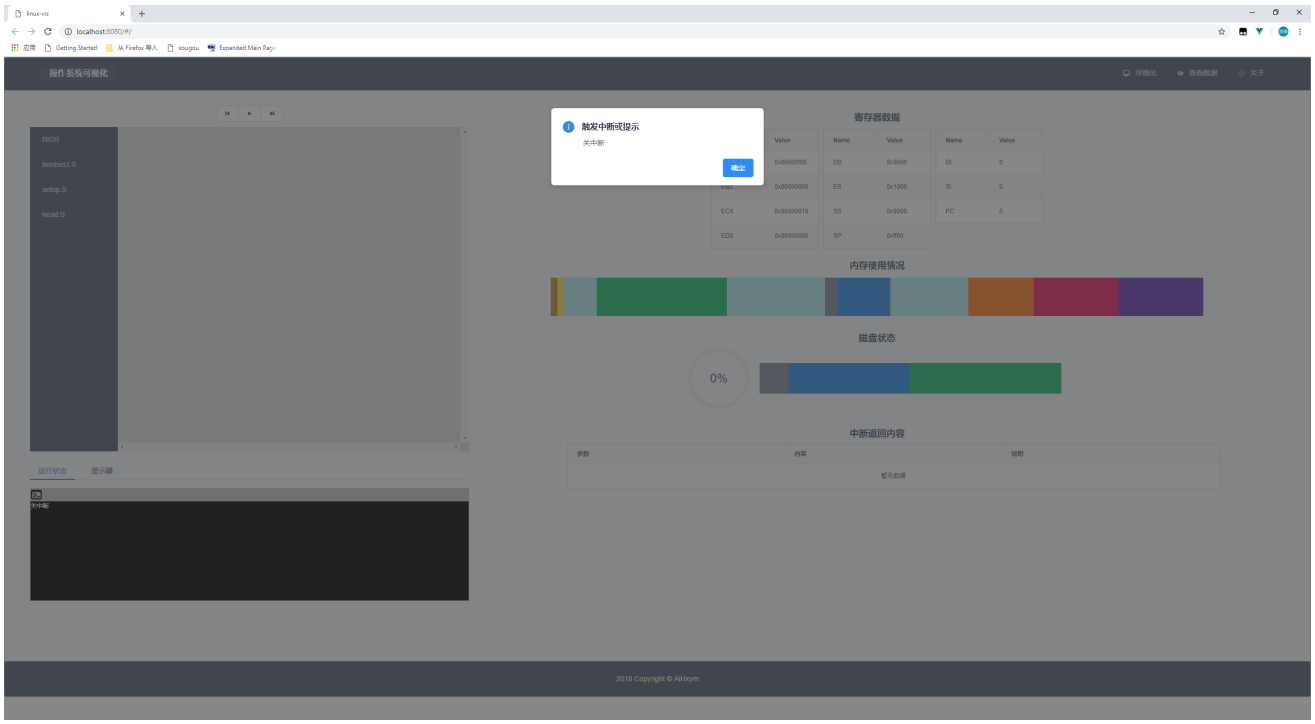
获取显示卡当前的显示模式



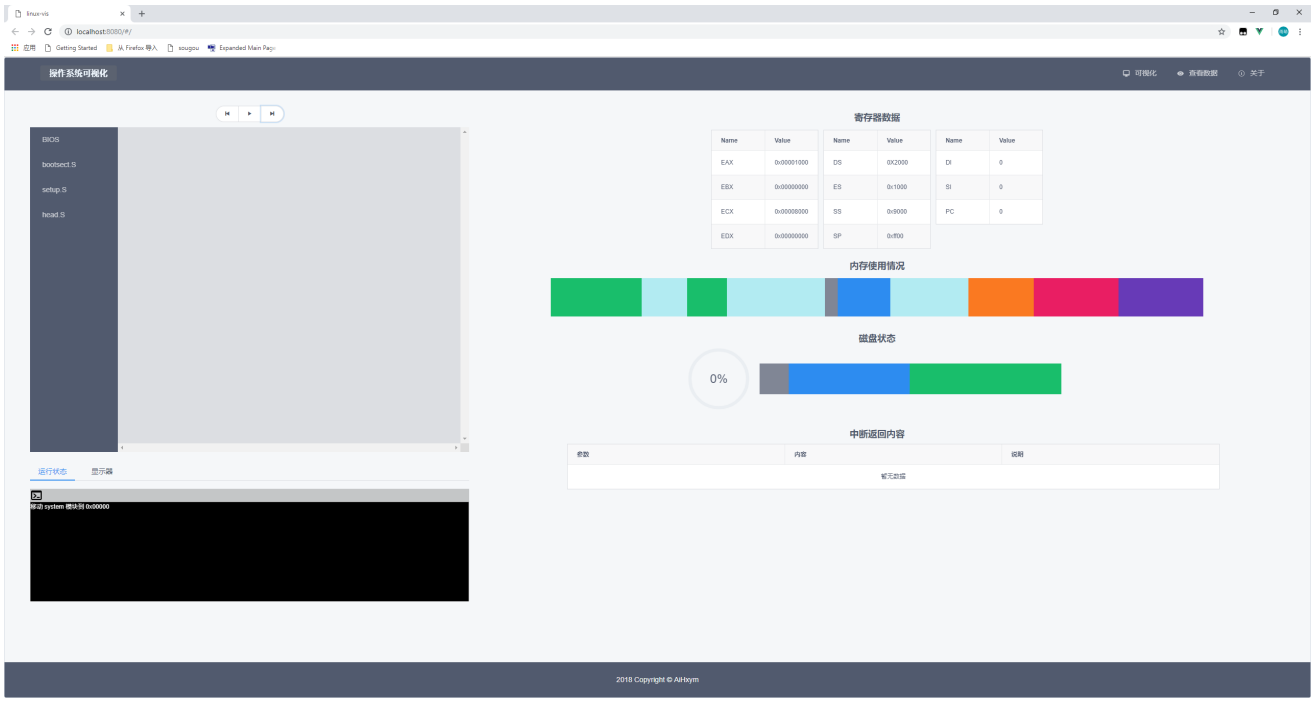
可以看到中断返回后的结果



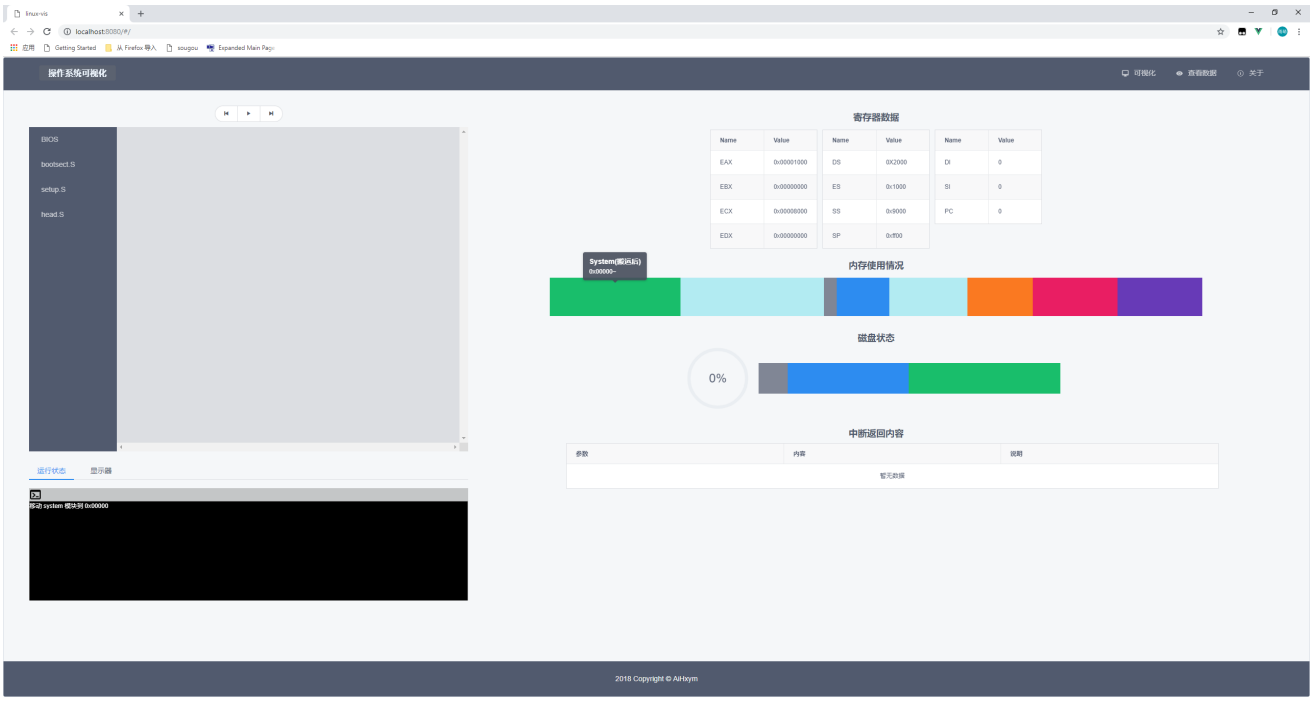
关中断



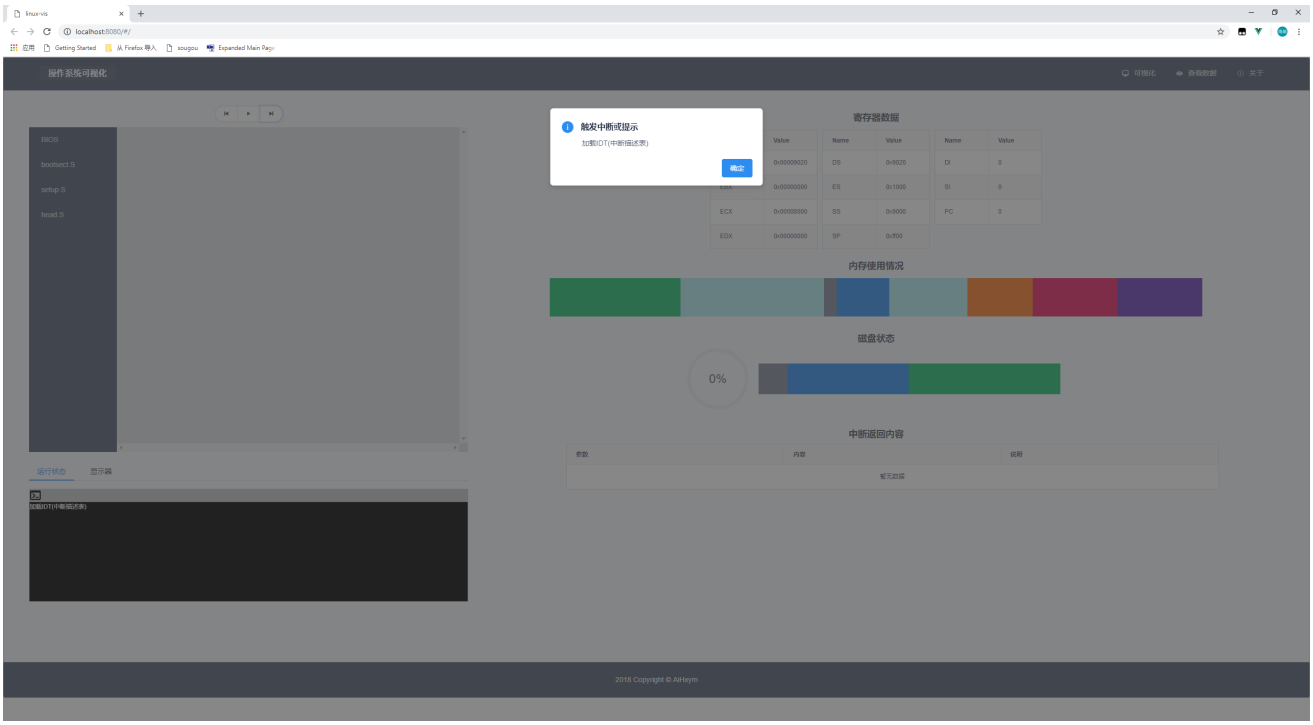
移动 system 模块到 0x00000



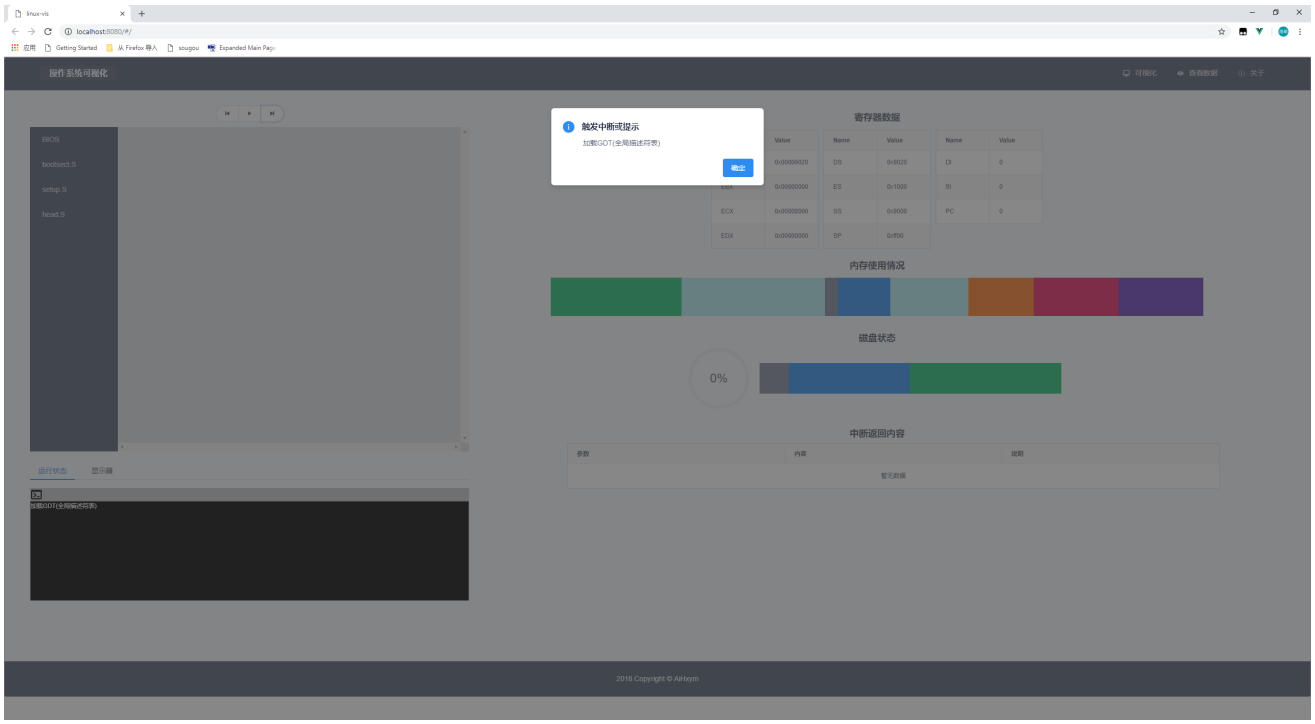
移动后的状态



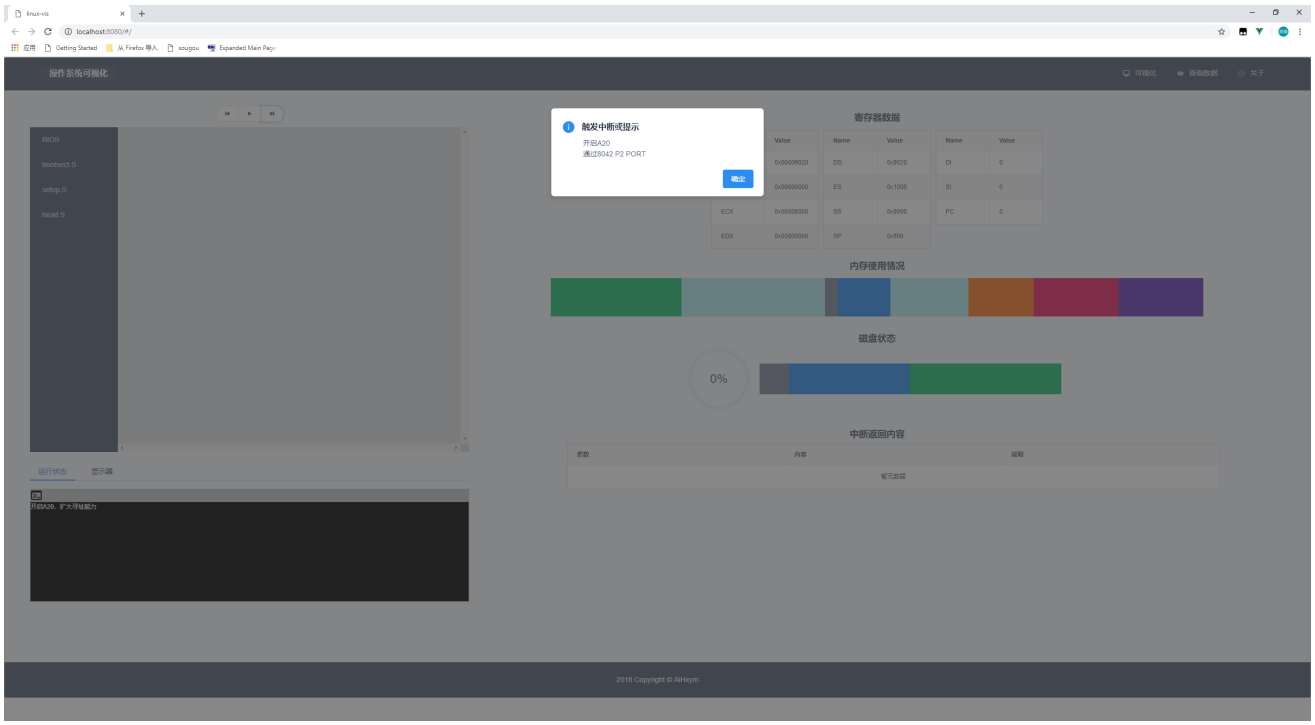
加载IDT(中断描述表)



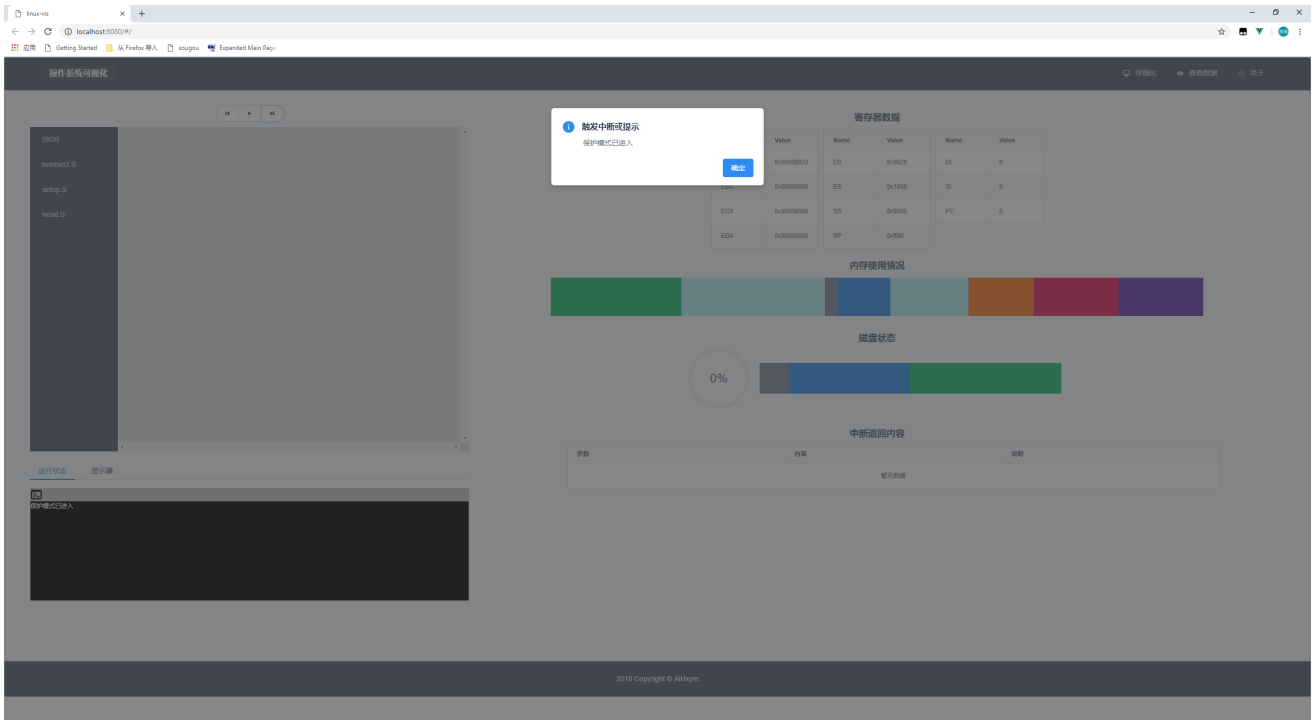
加载GDT(全局描述符表)



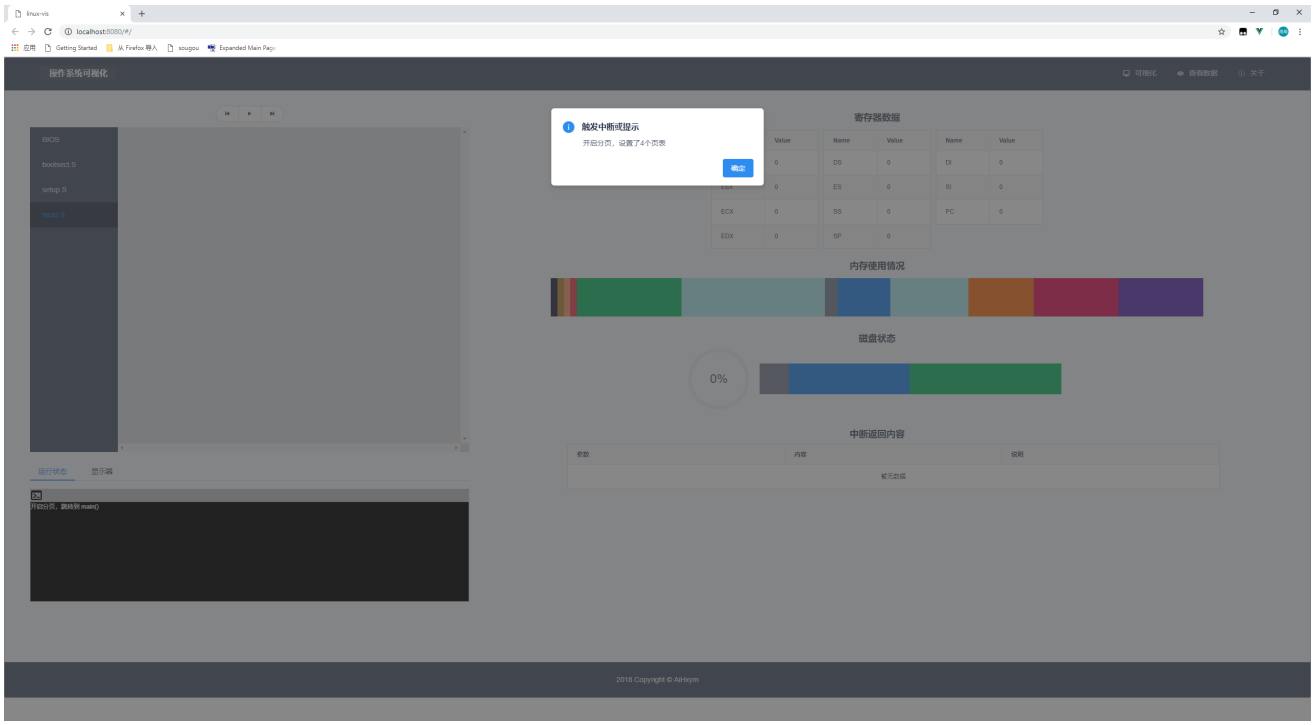
开启A20，扩大寻址能力



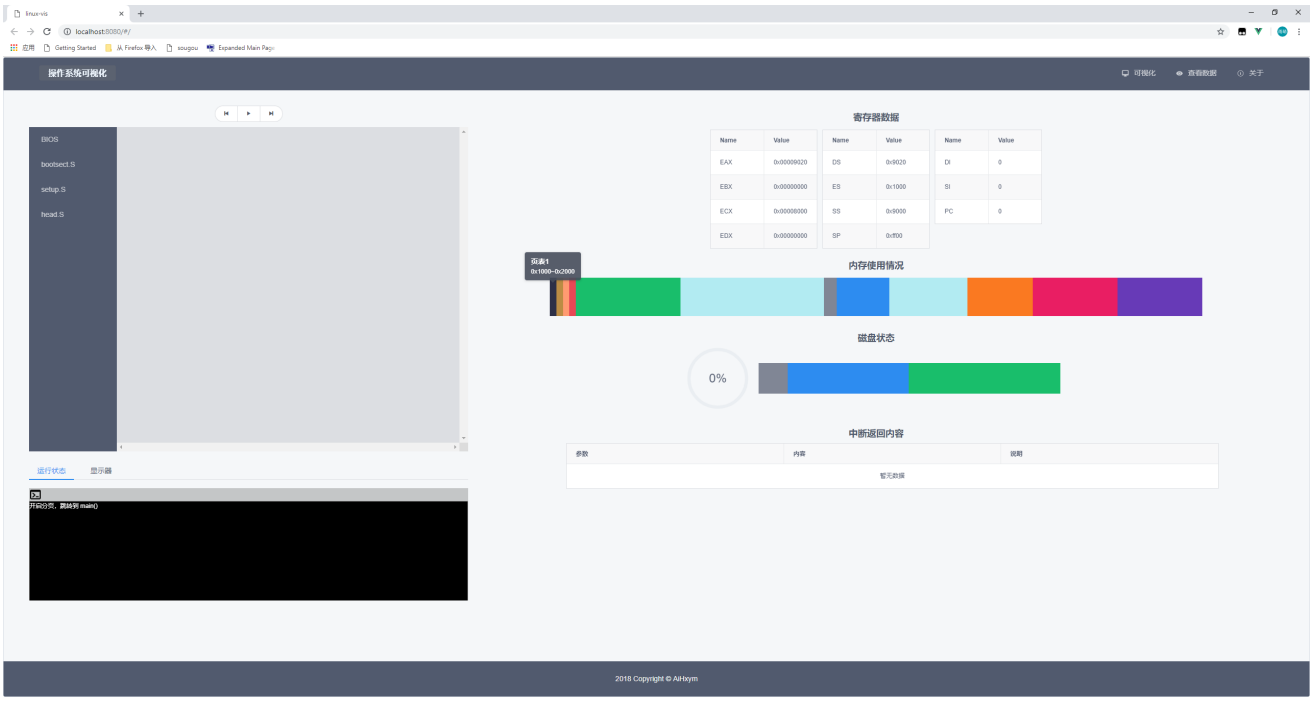
进入保护模式



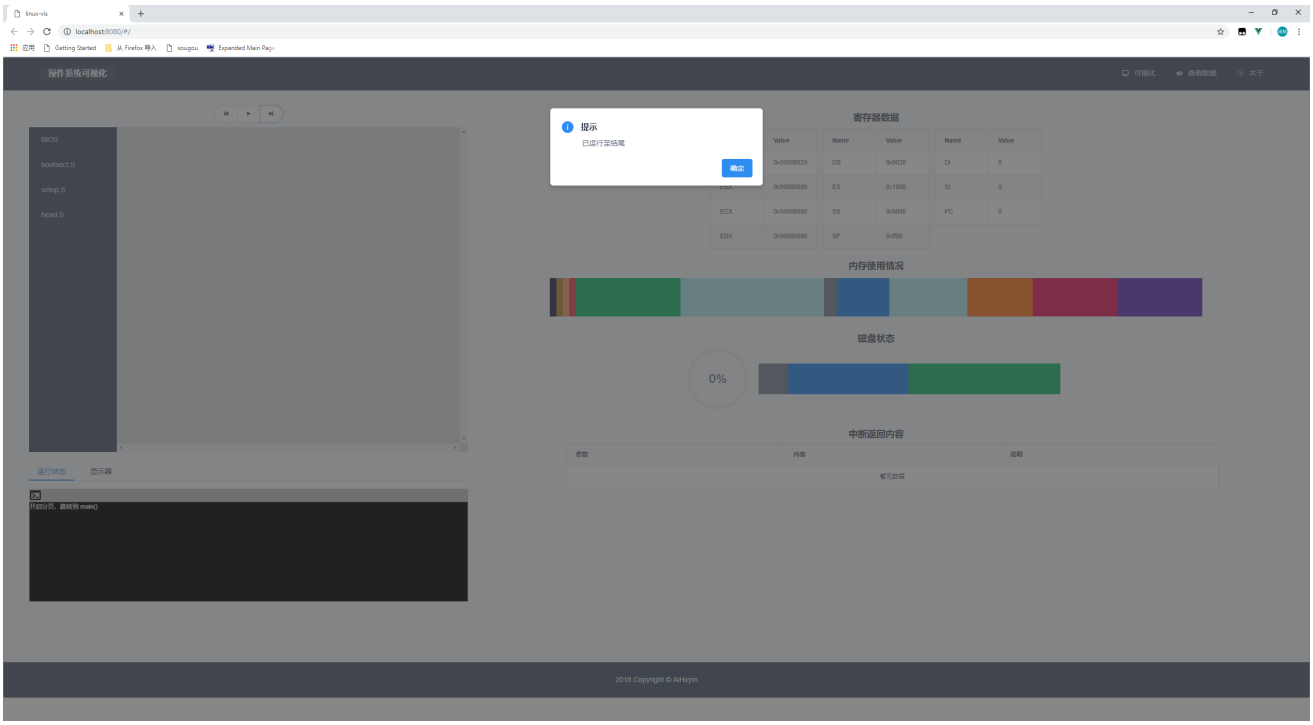
开启分页，跳转到 main()



加载的4个页表



可视化结束



实验总结

通过这次实验，我对整个Linux 0.11开机启动过程有了十分深入的理解，锻炼提高了自己的汇编语言能力和调试能力以及unity3D制作还有Web前端制作能力，每周的进度汇报也让我们有持续制作的动力，对比之前同学的样本，我认为我尽我所能将能够改进的部分都进行了改进，我觉得开机部分的可以可视化的内容差不多我都做到了。当然下一届同学如果有更好的想法的话欢迎找我来讨论，我很乐意提供帮助。

对下一届的建议与帮助

下一届同学可视化的时候不要拘泥于做动画与视频，努力尝试开发一些可视化工具，我这次实验封装了几个可以用于可视化的组件，也欢迎下一届同学使用，这样我们可以前人栽树后人乘凉，让可视化效果更好，更方便。数据提取我认为陈宇翔同学的最终版已经接近完美了，基本绝大部分的数据都可以提取，如果不是因为我提数据的时候他的最终版还没做出来的话我可能就没有必要一步步的进行手动调试了。

对老师的建议

建议老师将提取数据的时间延长一下，可视化部分减少一下，因为目前发现很多同学的问题在于数据没提出来或者没找到提取的方法就到了展示的日期了，这样很多人在数据提取部分的展示都有很多问题，最后有的同学数据没提取完就进行可视化，数据真实度也不太高，这样就有点违背实验本意了。

附录

bootsect.S

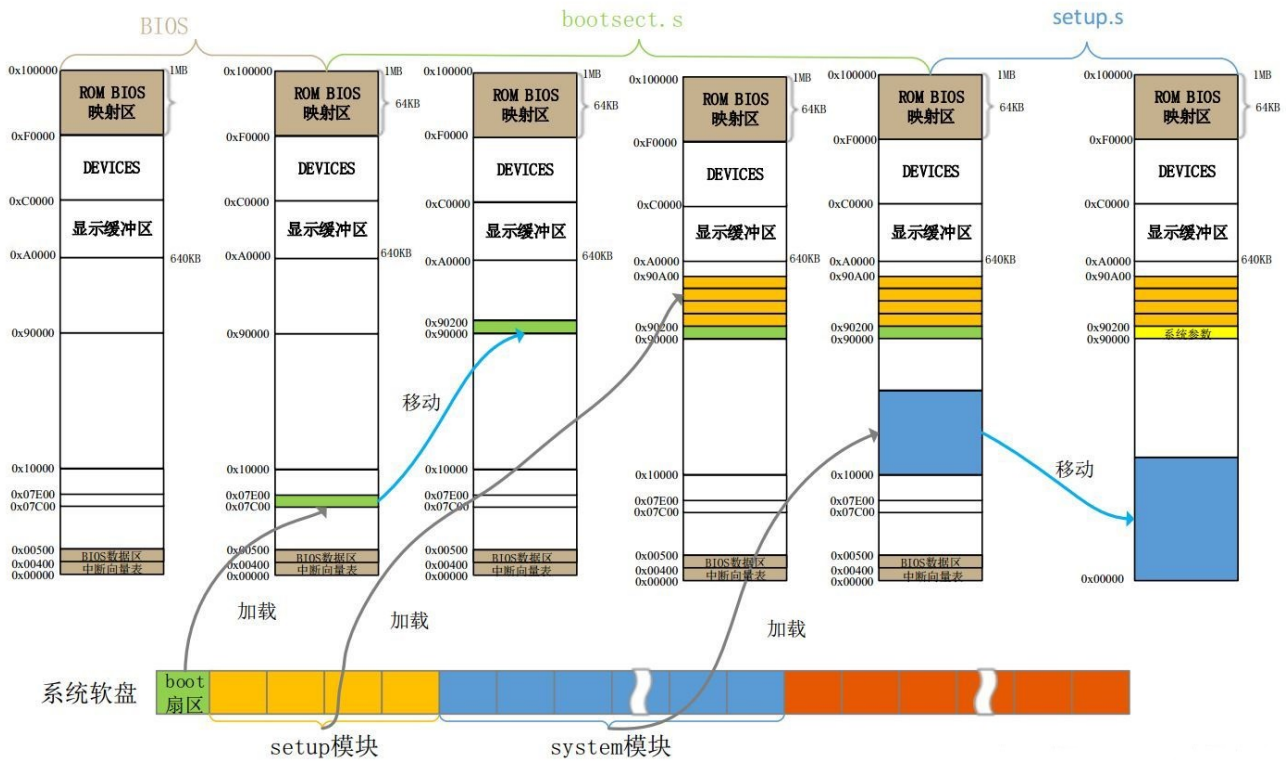
符号常量

```
SYSSIZE = 0x3000  ;system模块的长度


.globl begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text

SETUPLEN = 4                ! setup模块的长度，4个扇区
BOOTSEG  = 0x07c0           ! original address of boot-sector
INITSEG  = 0x9000           ! bootsect把自身搬运到0x90000
SETUPSEG = 0x9020           ! setup模块被加载到 0x90200
SYSSEG   = 0x1000           ! system模块被加载到0x10000
ENDSEG   = SYSSEG + SYSSIZE ! where to stop loading, 0x1000 + 0x3000 = 0x4000, 停止加载的
段地址

ROOT_DEV = 0x306            !第2个硬盘的第1个分区
```

bootsect 把自己搬运到 0x90000，并跳转

```
entry _start
_start:
    mov ax,#BOOTSEG
    mov ds,ax       !ds = 0x07c0
    mov ax,#INITSEG
    mov es,ax       !ex = 0x9000
    mov cx,#256     !搬运256次
    sub si,si       !si = 0
    sub di,di       !di = 0
                    !ds:si=0x07c0:0x0, es:di=0x9000:0x0
    rep
    movw            !每次搬运2个字节
    jmpi go,INITSEG   !跳转到 0x9000:go
```

以上代码表示把`ds:si`处(物理地址0x7c00)的内容搬运到`es:di`(物理地址0x90000),一共搬运512字节，即主引导扇区把自己移动到了0x90000处。



跳转后继续执行下面的指令，设置ds,es,ss和sp。

```
go:
    mov ax,cs
    mov ds,ax
    mov es,ax       !ds=es=cs=0x9000
    mov ss,ax
    mov sp,#0xFF00
                    !es:sp = 0x9000:0xff00 ，栈的设置
```

加载 setup 模块到 0x90200

```
load_setup:
    mov dx,#0x0000      ! 驱动器号(DL)0，磁头号(DH)0
    mov cx,#0x0002      ! 起始扇区号2, 磁道号0
    mov bx,#0x0200      ! 偏移地址0x200
    mov ax,#0x0200+SETUPLEN ! 功能号AH=0x02,AL=要读的扇区数目=SETUPLEN=4
    int 0x13            ! read it
    jnc ok_load_setup   ! ok - continue
    mov dx,#0x0000      !需要复位的驱动器号=DL=0
    mov ax,#0x0000      !功能号AH=0
    int 0x13            ! 复位磁盘
    j   load_setup
```

以上代码利用`INT 13H, AH=02H`把setup模块从磁盘（2~5扇区）加载到0x90200后面。 注意：柱面号和磁头号都从0开始，扇区号从1开始。

获得磁盘驱动器参数（主要是每磁道的扇区数量）

```
ok_load_setup:

! Get disk drive parameters, specifically nr of sectors/track

    mov dl,#0x00    !驱动器号为0，说明是软盘
    mov ax,#0x0800  ! AH=8 is get drive parameters
    int 0x13
    mov ch,#0x00    !这里用不上软盘的最大磁道号，可以使CH=0
    seg cs          !把段超越前缀设置为cs,只影响下一条语句
    mov sectors,cx
    !保存每磁道最大扇区数。对于软盘，最大磁道号不会超过256，所以CH足以表示，CL[7:6]为0
    !以上两句可以写为  mov cs:[sectors], cx
    mov ax,#INITSEG
    mov es,ax       !因为上面ES的值被修改，所以令ES=0x9000
```

打印 “Loading system ...”

```
    mov ah,#0x03    !读光标的位置
    xor bh,bh       !bh=页号
    int 0x10
    mov cx,#24          ! 24个字符
    mov bx,#0x0007      ! page 0, attribute 7 (normal)
    mov bp,#msg1
    mov ax,#0x1301      ! write string, move cursor
    int 0x10
msg1:
    .byte 13,10
    .ascii "Loading system ..."
    .byte 13,10,13,10
```

加载 system 到 0x10000

```
! we want to load the system (at 0x10000)

    mov ax,#SYSSEG  ! SYSSEG=0x1000
    mov es,ax       ! segment of 0x010000
    call    read_it
    call    kill_motor
```

过程read_it

这个过程的功能是把还未读取的扇区加载到`es:0x0000`处。注意：es必须是0x1000的整数倍，否则会陷入死循环。每读64KB，都会使es的值增加0x1000，当es=0x4000的时候，停止读取。

```
sread:  .word 1+SETUPLEN !当前磁道已经读取的扇区数, 前面的1表示引导扇区bootsect.s
head:   .word 0          ! current head，当前磁头号
track:  .word 0          ! current track，当前磁道号

read_it:
    mov ax,es
    test ax,#0x0fff     !使ax与0xfff按位与，测试es是否为0x1000的整数倍
die:    jne die         !结果不为0（说明es不是0x1000的整数倍）则陷入死循环
    xor bx,bx           ! bx（作为段内偏移地址）清零
rp_read:
    mov ax,es
    cmp ax,#ENDSEG      ! 实际上求(ax-ENDSEG)
    jb ok1_read         ! 当CF=1（ax<ENDSEG，有借位）时跳转到ok1_read
    ret                 ! 当ax>=ENDSEG时返回（我认为不会出现大于的情况）
ok1_read:
    seg cs
    mov ax,sectors      ! 这两句相当于 mov ax, cs:[sectors]; 获得每磁道扇区数
    sub ax,sread        ! ax = ax - sread, 得出本磁道未读扇区数
    mov cx,ax
    shl cx,#9           ! cx乘以512，求出字节数
    add cx,bx           ! 以上3行相当于 cx = ax * 512 + bx
                        ! 假设再读ax个扇区，cx就是段内共读入的字节数
    jnc ok2_read        ! 若cx < 0x10000（CF=0,没有进位）则跳转到ok2_read
    je ok2_read         ! 若cx = 0（ZF=1），说明刚好读入64KB，则跳转到ok2_read
    xor ax,ax           ! ax = 0x0000
    sub ax,bx           ! 求bx对0x10000的补数，结果在ax中
    shr ax,#9           ! 除以512,得到扇区数，AL作为参数，传给read_track
ok2_read:
    call read_track  ! 调用read_track过程，用AL传参,读取AL个扇区到ES:BX
    mov cx,ax        ! cx是该次操作已经读取的扇区数
    add ax,sread     ! ax是当前磁道已经读取的扇区数
    seg cs
    cmp ax,sectors
    jne ok3_read     ! 如果当前磁道还有扇区未读，跳转到ok3_read
    mov ax,#1        ! 说明当前磁道的扇区都已读完
    sub ax,head      ! ax = 1 - 磁头号
    jne ok4_read     ! 不为0则跳转到 ok4_read，说明磁头号为0
    inc track        ! 说明磁头号为1，磁道号增加1
ok4_read:
    mov head,ax  !更新磁头号（如果是37行跳转过来，则 head=1；否则 head=0）
```

```
    xor ax,ax     !ax=0, 因为更换了磁道，所以当前磁道已读扇区数置0
ok3_read:
    mov sread,ax        ! 更新当前磁道已经读取的扇区数
    shl cx,#9
    add bx,cx           ! 更新偏移地址
    jnc rp_read         ! 没有进位，则跳转到rp_read
    mov ax,es           ! 有进位，说明BX达到了64KB边界
    add ax,#0x1000
    mov es,ax           ! es增加0x1000
    xor bx,bx           ! bx = 0
    jmp rp_read         ! 继续读取
```

过程read_track

读取`AL`个扇区到`ES:BX`。此过程的入口参数是：

AL-要读的扇区数目

ES:BX-缓冲区地址

```
read_track:
    push ax
    push bx
    push cx
    push dx
    mov dx,track  ! 当前磁道号
    mov cx,sread  ! 已经读取的扇区数
    inc cx        ! CL是起始扇区号
    mov ch,dl     ! CH是磁道号----
    mov dx,head   ! 当前磁头号
    mov dh,dl     ! DH是磁头号
    mov dl,#0      ! DL是驱动器号，0表示软盘
    and dx,#0x0100 ! DH是磁头号，不是0就是1
    mov ah,#2      ! 功能号2，读扇区
    int 0x13
    jc bad_rt       ! CF=1，表示出错，复位磁盘
    pop dx
    pop cx
    pop bx
    pop ax
    ret
bad_rt: mov ax,#0   ! AH=0，磁盘复位功能
    mov dx,#0       ! DL=0，驱动器号
    int 0x13
    pop dx
    pop cx
    pop bx
    pop ax
    jmp read_track  ! 重新读取
```

过程kill_motor

```
kill_motor:
    push dx
    mov dx,#0x3f2 !软盘控制器的端口-数字输出寄存器端口，只写
    mov al,#0     !驱动器A，关闭FDC，禁止DMA和中断请求，关闭马达
    outb          !将al的值写入端口dx
    pop dx
    ret
```

确认根文件系统设备号

```
    seg cs
    mov ax,root_dev     !ax = ROOT_DEV
    cmp ax,#0
    jne root_defined    !如果 ROOT_DEV 不等于0则跳转到 root_defined
    seg cs
    mov bx,sectors       ! 取每磁道扇区数
    mov ax,#0x0208       ! /dev/ps0 - 1.2Mb
    cmp bx,#15           ! 判断每磁道扇区数是否等于15
    je  root_defined     ! 说明是1.2MB的软盘
    mov ax,#0x021c       ! /dev/PS0 - 1.44Mb
    cmp bx,#18           ! 判断每磁道扇区数是否等于18
    je  root_defined     ! 说明是1.44MB的软盘
undef_root:
    jmp undef_root       ! 死循环
root_defined:
    seg cs
    mov root_dev,ax      ! 将检查过的设备号保存到 root_dev 中
```

在Linux中软驱的主设备号是2，次设备号 = type * 4 + nr。

其中，nr等于0~3时分别对应软驱A、B、C、D；type是软驱的类型，比如2表示1.2MB，7表示1.44MB等。

因为是可引导的驱动器，所以肯定是A驱。对于1.2MB，设备号 = 2 << 8 + 2 * 4 + 0 = 0x208；对于1.44MB，设备号 = 2 << 8 + 7 * 4 + 0 = 0x21C。

跳转到 **setup** 去执行

```
jmpi    0,SETUPSEG   !到此本程序就结束了。
```

setup.S

符号常量

```
INITSEG  = 0x9000   ! bootsect.s 的段地址
SYSSEG   = 0x1000   ! system loaded at 0x10000
SETUPSEG = 0x9020   ! 本程序的段地址
```

获取一些参数保存在 **0x90000** 处

保存光标的位置

```
    mov ax,#INITSEG  !INITSEG = 0x9000
    mov ds,ax        ! ds = 0x9000
    mov ah,#0x03     ! 功能号=3，获取光标的位置
    xor bh,bh        ! bh = 页号 = 0（输入）
    int 0x10         ! 输出：  DH=行号，DL=列号
    mov [0],dx       ! 保存光标的行号和列号到 0x90000，共占2字节
```

获取从 **1M** 处开始的扩展内存大小

```
    ! 利用 BIOS 中断 0x15 功能号 ah = 0x88 取系统所含扩展内存大小，并保存在内存 0x90002 处
    ! 返回：ax=从0x100000(1M)处开始的扩展内存大小(KB).若出错则CF置位,ax=出错码
    mov ah,#0x88
    int 0x15
    mov [2],ax ! ax = 从1M处开始的扩展内存大小
```

获取显示模式

```
! 获取显示卡当前的显示模式
! 调用 BIOS 中断 0x10，功能号 ah = 0x0f
! 返回：  ah=字符列数；  al=显示模式；bh=当前显示页。
! 0x90004(1个字)存放当前页；0x90006(1字节)存放显示模式；0x90007(1字节)存放字符列数。
mov ah,#0x0f
int 0x10
mov [4],bx      ! bh = 当前显示页
mov [6],ax      ! al = 显示模式，ah = 字符列数（窗口宽度）
```

检查显示方式**(EGA/VGA)**并获取参数

```
! 检查显示方式(EGA/VGA)并获取参数。
! 调用 BIOS 中断 0x10，功能号：  ah = 0x12，子功能号：  bl = 0x10
! 返回：bh=显示状态。  0x00-彩色模式，I/O 端口=0x3dX
!                     0x01-单色模式，I/O 端口=0x3bX
! bl = 安装的显示内存。0x00 - 64k
!                       0x01 - 128k
!                       0x02 - 192k
!                       0x03 - 256k
! cx = 显示卡特性参数。
!
mov ah,#0x12 ! 功能号
mov bl,#0x10 ! 子功能号
int 0x10
mov [8],ax      ! 我也不知道这个是什么(╯﹏╰)
mov [10],bx     ! bh=显示状态(单色模式/彩色模式),bl=已安装的显存大小
mov [12],cx     ! ch=特性连接器比特位信息,cl=视频开关设置信息
```

复制硬盘参数表

复制 **HD0** 的硬盘参数表

```

! 复制 hd0 的硬盘参数表，参数表地址是中断向量0x41的值，表长度16B
```

```
! 中断向量在中断向量表中的位置 = 中断类型号N × 4
! (N*4)的字单元存放偏移地址;
! (N*4+2)的字单元存放段基址。

    mov ax,#0x0000
    mov ds,ax       ! ds=0
! 将内存[4*0x41]处的低2字节(偏移地址)传给si,高2字节(段地址)传给ds
    lds si,[4*0x41]
    mov ax,#INITSEG
    mov es,ax           !es = 0x9000
    mov di,#0x0080
    mov cx,#0x10        !重复16次
! ds:si --> es:di(0x9000:0x0080),共传送16B
    rep
    movsb
```

复制 **HD1** 的硬盘参数表

```
! 复制 hd1 的硬盘参数表，参数表地址是中断向量0x46的值，表长度16B
! 道理同上一小节,此处不赘述
    mov ax,#0x0000
    mov ds,ax
    lds si,[4*0x46]
    mov ax,#INITSEG ! INITSEG = 0x9000
    mov es,ax
    mov di,#0x0090
    mov cx,#0x10
! ds:si --> es:di(0x9000:0x0090),共传送16B
    rep
    movsb
```

检查系统是否有第**2**个硬盘

```
! 检查系统是否有第2个硬盘，如果没有就把第2个参数表清零
! 利用 BIOS 中断调用 0x13 的取盘类型功能，功能号 ah = 0x15;
! 输入： dl=驱动器号（0x8X 是硬盘：0x80 指第 1 个硬盘，0x81 第 2 个硬盘）
! 输出： ah=类型码；00-没有这个盘，CF 置位；
!                 01-是软驱，没有 change-line 支持；
!                 02 -是软驱(或其他可移动设备），有 change-line 支持；
!                 03 -是硬盘。
!
    mov ax,#0x01500 ! 功能号 ah=0x15，读取盘类型
    mov dl,#0x81    ! dl=驱动器号，0x81代表第2个硬盘
    int 0x13
    jc  no_disk1    ! CF置位,表示没有这个盘
    cmp ah,#3
    je  is_disk1    ! ah=3表示存在第2个硬盘,跳转到is_disk1
no_disk1:
! 清空第2个表
    mov ax,#INITSEG
    mov es,ax
    mov di,#0x0090  ! es:di = 0x9000:0x0090
```

```
    mov cx,#0x10
    mov ax,#0x00    ! AL=0
    rep
    stosb           ! Store AL at address es:di
is_disk1:
```

关中断

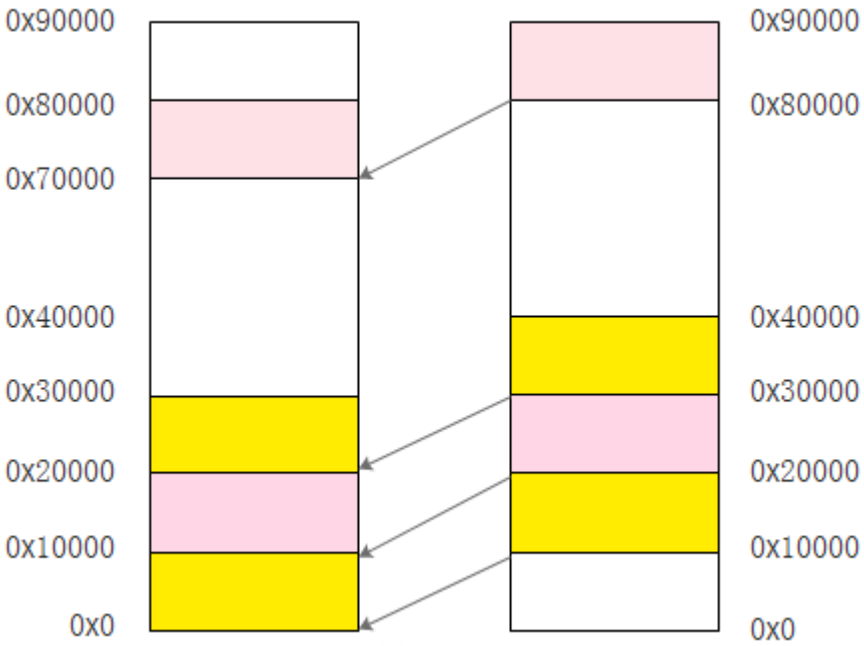
```
! 为进入保护模式做准备

    cli         ! no interrupts allowed !
```

移动 system 模块到 0x00000

bootsect.S引导程序将 system 模块读入到 0xl0000 开始的位置。由于当时假设 system 模块最大长度不会超过 0x80000 (512KB)，即其末端不会超过内存地址 0x90000，所以bootsect.S会把自己移动到0x90000 开始的地方，并把 setup 加载到它的后面。下面这段程序的用途是再把整个 system 模块移动到 0x00000 位置，即把从 0x10000 到 0x8ffff 的内存数据块(共512KB)整块地向内存低端移动了0x10000(64KB)。

```
! 从代码实现来看，是一小块(0x10000B=64KB)一小块移动的，共移动8小块。
    mov ax,#0x0000
    cld             ! 'direction'=0, movs moves forward
do_move:
    mov es,ax       ! es是目的段地址
    add ax,#0x1000
    cmp ax,#0x9000   ! 当 ax==0x9000 时结束移动
    jz  end_move
    mov ds,ax       ! ds是源段地址，ds比es大0x1000
    sub di,di        ! di = 0
    sub si,si       !  si = 0
    mov cx,#0x8000  ! 重复 0x8000次
    rep             ! ds:si --> es:di
    movsw           ! 每次移动2B.
    jmp do_move    !  本轮一共移动 0x8000*2B = 0x10000B=64KB. 准备下一轮移动

end_move:
```

示意图：



加载IDT

```
end_move:
    mov ax,#SETUPSEG
    mov ds,ax             !ds = 0x9020,指向本程序段,setup.s 被加载到 0x90200
    !idt_48 标号处的内容如下
    !idt_48:
    !        .word   0           ! idt 界限值=0
    !       .word   0,0          ! idt 基地址=0L

    lidt    idt_48      ! load idt with 0,0
```

加载GDT

```
    !gdt_48 标号处的内容如下
    !gdt_48:
    !.word  0x800         ! 0x800 = 2048, 2048/8=256,可容纳256个描述符， 其实0x7ff即可
    !.word  512+gdt,0x9   ! setup.s被加载到0x90200, gdt base = 0x90200+gdt =
0x90000+512+gdt
    lgdt    gdt_48
```

开启A20

```
    call empty_8042  ! 等待输入缓冲器为空
    mov al,#0xD1
    out #0x64,al
    call    empty_8042   ! 等待输入缓冲器为空，即命令被接受
    mov al,#0xDF         ! A20 on
    out #0x60,al
    call    empty_8042   ! 等待输入缓冲器为空，即参数被接受
```

进入保护模式

首先加载机器状态字（Load Machine Status Word），也称控制寄存器 CR0，其比特位 0 置 1 将使 CPU 切换到保护模式，并且运行在特权级0，即当前特权级 CPL = 0。此时各个段寄存器仍然指向与实地址模式中相同的线性地址处（在实地址模式下线性地址与物理地址相同）。在设置该比特位后，随后一条指令必须是一条段间跳转指令，用于刷新CPU当前指令队列。因为 CPU 是在执行一条指令之前就已从内存读取该指令并对其进行译码。然而在进入保护模式以后那些属于实模式的预先取得的指令信息就变得不再有效。而一条段间跳转指令就会刷新 CPU 的当前指令队列，即丢弃这些无效信息。另外，Intel手册上建议 80386 或以上 CPU 应该使用指令 mov cr0, ax 切换到保护模式。lmsw 指令仅用于兼容以前的 286 CPU。

```
mov ax,#0x0001  ! Protection Enable (bit 0 of CR0).
lmsw ax         ! 实际上lmsw指令仅仅加载CR0的低4位，由低到高分别是PE，MP，EM，TS
jmpi 0,8        ! jmp offset 0 of segment 8 (cs)
```

实际上`lmsw`指令仅仅加载CR0的低4位，由低到高分别是PE，MP，EM，TS. 这里我们仅关注 PE，其他的都设为0.

`jmpi 0,8` 段间跳转指令。执行后，CS=8，IP=0.

下面就进入system模块了，head.S就是system模块的头部。

head.S

从这里开始，内核完全是在保护模式下运行了。head.s 汇编程序与前面的语法格式不同，它采用的是`AT&T`汇编语言格式，并且需要使用 GNU 的 as 和 ld 进行编译和连接。代码中赋值的方向是从左到右。

程序实际上处于内存地址0处。

加载段寄存器

```
.text
.globl idt,gdt,pg_dir,tmp_floppy_area
pg_dir:         # 页目录将会存放在这里，把这里的代码覆盖掉
.globl startup_32
startup_32:
    movl $0x10,%eax
    mov %ax,%ds
    mov %ax,%es
    mov %ax,%fs
    mov %ax,%gs
    lss stack_start,%esp
```

设置中断描述符表（IDT）

```
call setup_idt
```

```
setup_idt:
    lea ignore_int,%edx
    movl $0x00080000,%eax
```

```
    movw %dx,%ax              /* selector = 0x0008 = cs */
    movw $0x8E00,%dx /* interrupt gate: dpl=0, present */

    lea idt,%edi      # 取idt的偏移给edi
    mov $256,%ecx     # 循环256次
rp_sidt:
    movl %eax,(%edi)     # eax -> [edi]
    movl %edx,4(%edi)    # edx -> [edi+4]
    addl $8,%edi         # edi + 8 -> edi
    dec %ecx
    jne rp_sidt
    lidt idt_descr       # 加载IDTR
    ret

...



idt_descr:
    .word 256*8-1       # idt contains 256 entries
    .long idt           # IDT 的线性基地址

...

idt:
    .fill 256,8,0       # idt is uninitialized
```

设置全局描述符表（**GDT**），加载 **GDTR**

```
call setup_gdt
```

```
setup_gdt:
    lgdt gdt_descr # 加载GDTR
    ret
```

重新加载段寄存器

```
call setup_idt
call setup_gdt
movl $0x10,%eax         # reload all the segment registers
mov %ax,%ds          # after changing gdt. CS was already
mov %ax,%es          # reloaded in 'setup_gdt'
mov %ax,%fs
mov %ax,%gs
lss stack_start,%esp
```

检测**A20**是否开启

```
xorl %eax,%eax
1:  incl %eax           # check that A20 really is enabled
    movl %eax,0x000000  # loop forever if it isn't
    cmpl %eax,0x100000
    je 1b
```

用于测试 A20 地址线是否已开启。采用的方法是向内存地址 0x0_0000 处写入任意一个数值，然后看内存地址 Ox10_0000(1M)处是否也是这个数值。如果一直相同的话，就一直比较下去。死机表示 A20 线没有选通。

开启分页，跳转到 main()

将内核的页表直接放在页目录之后，使用了4个页表来寻址16MB的物理内存。如果有多于16MB的内存，就需要在这里进行扩充修改。

Linus 在物理地址0x0处开始存放1页页目录和4页页表。页目录是系统所有进程公用的，而其后的4页页表则属于内核专用，它们把线性地址 0x000000~0xFFFFFF 一一映射到物理地址 0x000000~0xFFFFFF。

```
.org 0x1000    #从偏移 0x1000 处开始放第1个页表（偏移0开始处将存放页目录）
pg0:

.org 0x2000    #从偏移 0x2000 处开始放第2个页表
pg1:

.org 0x3000    #从偏移 0x3000 处开始放第3个页表
pg2:

.org 0x4000    #从偏移 0x4000 处开始放第4个页表
pg3:

.org 0x5000    #定义下面的内存数据块从偏移 0x5000 处开始
```

`.ORG` 伪指令用来表示起始的偏移地址，紧接着ORG的数值就是偏移地址的起始值。ORG伪操作常用来指定数据的存储地址，有时也用来指定代码段的起始地址。

A20

A20是什么

1981年8月，IBM 公司最初推出的个人计算机所使用的 CPU 是 Intel 8088。在该微机中地址线只有 20 根（A0~A19）。当时，计算机的 RAM 只有几百 KB 或不到 1MB 时，20 根地址线已足够用来寻址。其所能寻址的最高地址是0xFFFF:0xFFFF，即0x10FFEF。对于超出 0x100000(1MB)的寻址地址， 将默认回卷到0x0FFEF。

IBM 公司于 1985 年引入 AT 机时，使用的是 Intel 80286 CPU，具有 24 根地址线，最高可寻址 16MB，并且有一个与 8088 完全兼容的实模式运行方式。

然而，在寻址值超过 1MB 时它却不像 8088 那样实现地址的回卷。但是当时已经有一些程序是利用这种地址环绕机制进行工作的。为了实现完全的兼容性，IBM 公司发明了一种方法——使用一个开关来开启或禁止地址线的比特位20到23。

由于当时的 8042 键盘控制器上恰好有空闲的端口引脚（输出端口P2，引脚名是 P21 ），于是便使用了该引脚来作为与门控制这个地址比特位。该信号被称为 A20。如果它为零，则比特 20 及以上地址都被清除。

在机器启动时，默认条件下，A20 地址线是禁止的，所以操作系统必须使用适当的方法来开启它。但是由于各种兼容机所使用的芯片集不同，要做到这一点是非常麻烦的，通常要在几种控制方案中选择。

方案一

对 A20 信号线进行控制的常用方法是通过设置键盘控制器的端口值。 比如以下代码可以开启A20：

```
    call empty_8042
    mov al,#0xD1        ! command write
    out #0x64,al
    call empty_8042
    mov al,#0xDF        ! A20 on
    out #0x60,al
    call empty_8042

empty_8042:
    .word   0x00eb,0x00eb !机器码，跳转到下一句，为了延时
    in  al,#0x64          ! 8042 status port
    test al,#2            ! is input buffer full?
    jnz empty_8042        ! yes - loop
    ret
```

方案二

有些操作系统将 A20 的开启和禁止作为实模式与保护模式之间进行转换的标准过程的一部分。由于键盘控制器的速度很慢，因此就不能使用键盘控制器对 A20 线来进行操作。为此引进了一个 A20 快速门选项（Fast Gate A20），它使用 I/O 端口 0x92 来处理 A20 信号线。

```
    in al,0x92                          ;南桥芯片内的端口
    or al,0000_0010B
    out 0x92,al                         ;打开A20
```

方案三

通过INT 0x15中断

```
/*
 * 打开A20
 * 返回结果：
 *    成功：CF=0,  AH=0
 *    失败：CF=1, AH=0x01    键盘控制器处于secur模式
 *                AH=0x86    功能不支持
 */
    mov ax, 0x2401
    int 0x15
```

方案四

还有一种方式是通过读 0xee 端口来开启 A20 信号线，写该端口则会禁止 A20 信号线。

```
in al,0xee    ;开启A20
```