Doc No.: DR-N927-LX.11

Pages: 28

Creator: bob

Create time:

读书笔记

ACTEL onetouch Copyright © 2015

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License,

Version 1.3 or any later version published by the Free Software Foundation;

|  |  |  |  |
| --- | --- | --- | --- |
| Change List |  |  |  |
| **Version** | **Date** | **Author** | **Comment** |
|  |  |  |  |
| 0.0.1 | 2015-09-27 | bob | Create this document. |

# Contents

[Linux内核0.11完全注释 1](#_Toc432234444)

[Contents 3](#_Toc432234445)

[Linux 0.11实验环境搭建 4](#_Toc432234446)

[软件资源 4](#_Toc432234447)

[运行Linux0.11 5](#_Toc432234448)

[制作硬盘镜像文件 5](#_Toc432234449)

[创建MINIX文件系统 6](#_Toc432234450)

[在hdc.img上建立根文件系统 10](#_Toc432234451)

[运行linux0.11 OS 10](#_Toc432234452)

[建立文件系统 11](#_Toc432234453)

[运行硬盘镜像上的Linux0.11 12](#_Toc432234454)

[编译环境 14](#_Toc432234455)

[文件传输 14](#_Toc432234456)

[Linux0.11编译环境 17](#_Toc432234457)

[编译hello world 18](#_Toc432234458)

[编译运行mtools-2.0.5 19](#_Toc432234459)

[Linux0.11源码编辑环境 22](#_Toc432234460)

[Source Insight管理源码 22](#_Toc432234461)

[修改源码文件 24](#_Toc432234462)

[编译运行Linux0.11 25](#_Toc432234463)

[拷贝源码 25](#_Toc432234464)

[编译源码 25](#_Toc432234465)

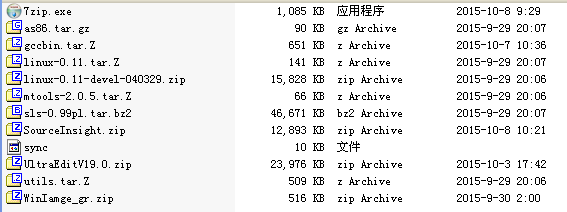
[安装运行 26](#_Toc432234466)

[appendix a. Ref Links 28](#_Toc432234467)

Linux 0.11实验环境搭建

软件资源

|  |  |
| --- | --- |
| 7-zip | 打包/解压软件 |
| as86.tar.gz | as86和ld86 |
| gccbin.tar.Z | gas，gld，gar等编译工具 |
| linux-0.11.tar.Z | Linux0.11源码程序 |
| linux-0.11-devel-040329.zip | Linux0.11资源包 |
| mtools-2.0.5.tar.Z | Linux0.11上操作dos文件系统文件 |
| sls-1.0.rar | 可运行的linux系统 |
| SourceInsight.zip | 源码查看工具 |
| sync | Linux0.11上的可执行程序，编译Linux0.11时会用到 |
| UltraEditV19.0.zip | 文本、二进制文件编辑器 |
| utils.tar.Z | Linux0.11上的Make程序 |
| WinIamge\_gr.zip | 软盘镜像文件编辑器 |

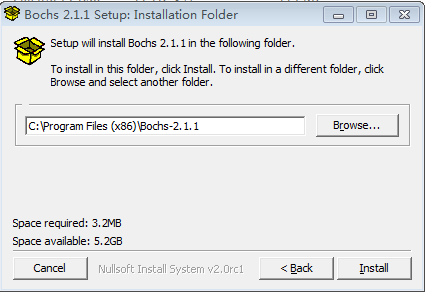


资源下载：http://pan.baidu.com/s/1dDhSEKL

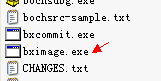
## 运行Linux0.11

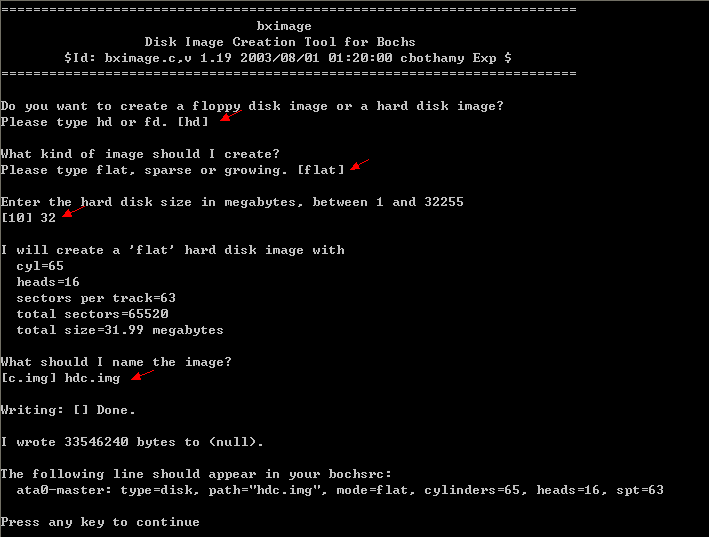
制作硬盘镜像文件

解压linux-0.11-devel-040329.zip文件，安装Bochs-2.1.1.exe



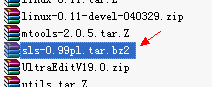
Bochs系统带有一个Image生成工具“Disk Image Creation Tool”（bximage.exe）。创建一个32M的硬盘镜像文件。

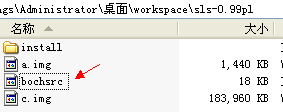




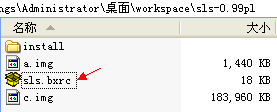
创建MINIX文件系统

解压文件sls-0.99pl.tar.bz2文件



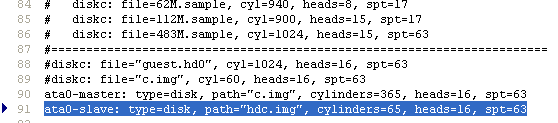


将文件bochsrc改名为sls.bxrc

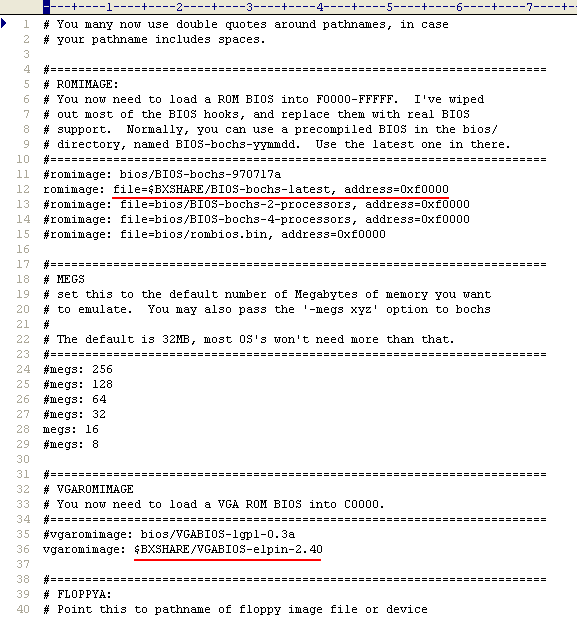


编辑sls.bxrc文件ata0-master一行下加入如下内容

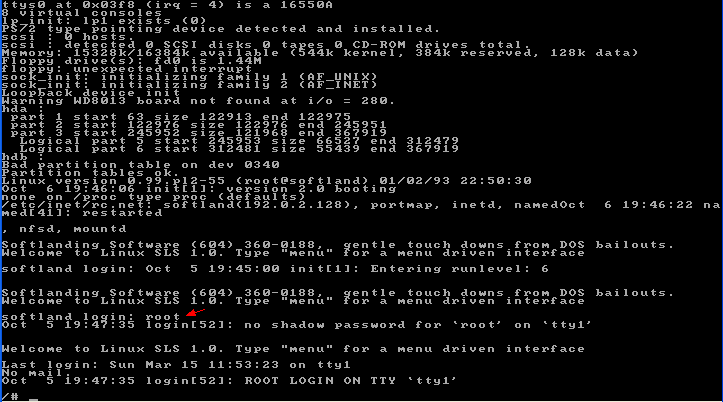
|  |
| --- |
| ata0-slave: type=disk, path="hdc.img", cylinders=65, heads=16, spt=63 |



修改romimage文件和vgaromimage文件路径



双击sls.bxrc的图标，运行SLS Linux模拟系统。在出现login提示符时键入’root’并按回车键。



利用 fdisk 命令在 hdc.img 文件中建立 1 个分区

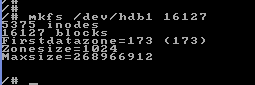


fdisk对应的help描述如下：



请记住该分区中数据块数大小（这里是16127），在创建文件系统时会使用到这个值。当分区建立好后，按照通常的做法需要重新启动一次系统，以让SLS Linux系统内核能正确识别这个新加的分区。

再次进入SLS Linux模拟系统后，我们使用mkfs命令在刚建立的第1个分区上创建MINIX文件系统。命令与信息如下所示。这里创建了具有16127个数据块的分区（一个数据块为1KB字节）。



至此，我们完成了在hdc.img 文件的第1 个分区中创建文件系统的工作。

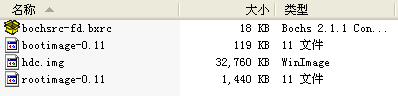
在hdc.img上建立根文件系统

运行linux0.11 OS

从” linux-0.11-devel-040329.zip”文件中解压如下文件：

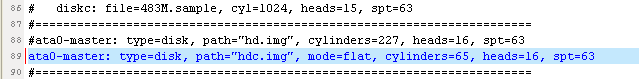


将创建的Minix文件系统硬盘镜像文件（hdc.img）拷贝到相同目录下，将bootimage-0.11-fd改名为bootimage-0.11。所以文件如下图所示：



在bochsrc-fd.bxrc文件中添加如下内容

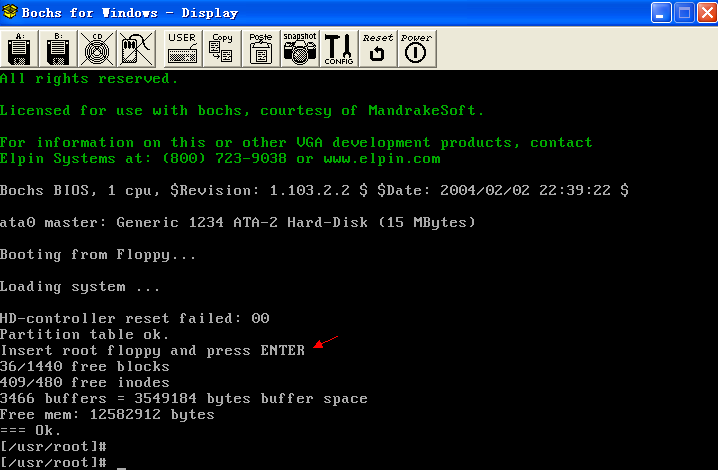
|  |
| --- |
| ata0-master: type=disk, path="hdc.img", mode=flat, cylinders=65, heads=16, spt=63 |



修改bochsrc-fd.bxrc文件中如下内容（可选）：

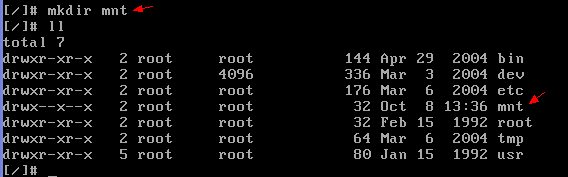
|  |
| --- |
| romimage: file="C:\Program Files\Bochs-2.1.1\BIOS-bochs-latest", address=0xf0000  vgaromimage: "C:\Program Files\Bochs-2.1.1\VGABIOS-elpin-2.40" |

双击bochsrc-fd.bxrc文件运行虚拟机：

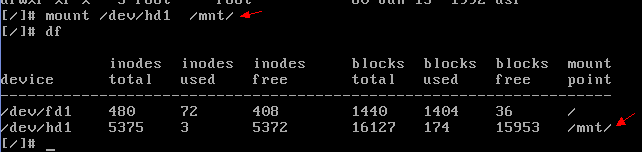


建立文件系统

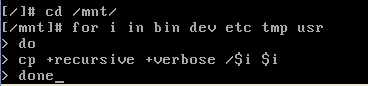
创建挂载目录

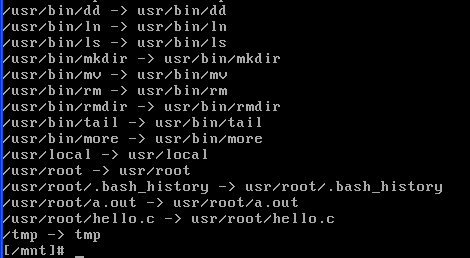


挂载硬盘镜像

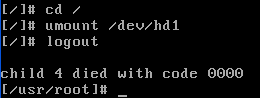


复制文件系统



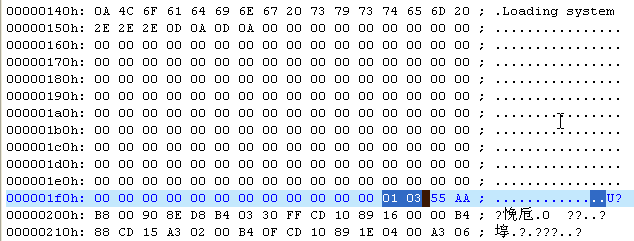


卸载硬盘后退出

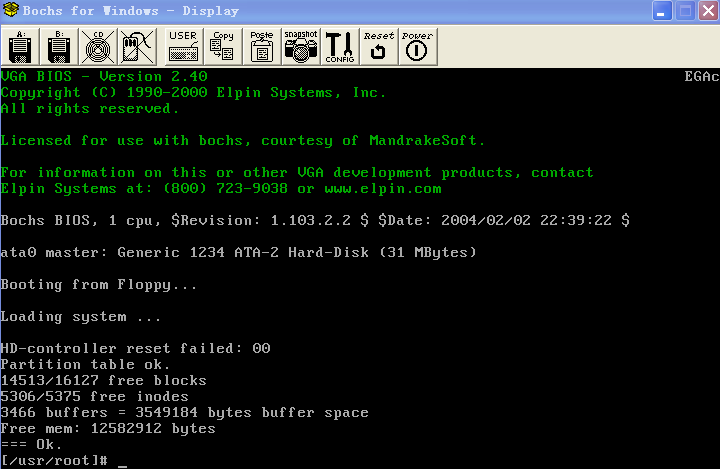


运行硬盘镜像上的Linux0.11

用UltraEdit编辑bootimage-0.11二进制文件。修改第509、510 字节（原值应该是00、00）为01、03，表示根文件系统设备在硬盘Image的第1个分区上。然后存盘退出。如果把文件系统安装在了别的分区上，那么需要修改前1个字节以对应到你的分区上。



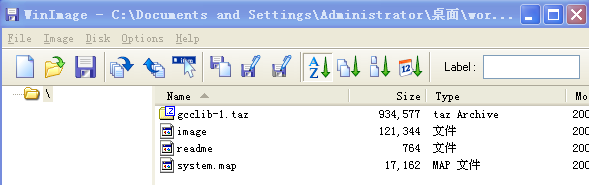
双击bochsrc-fd.bxrc运行虚拟机



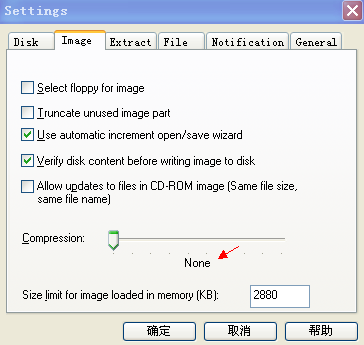
编译环境

文件传输

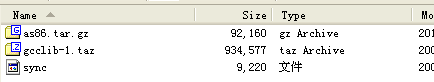
从linux-0.11-devel-040329.zip中解压出diskb.img文件，用WinIamge打开该软盘镜像文件：



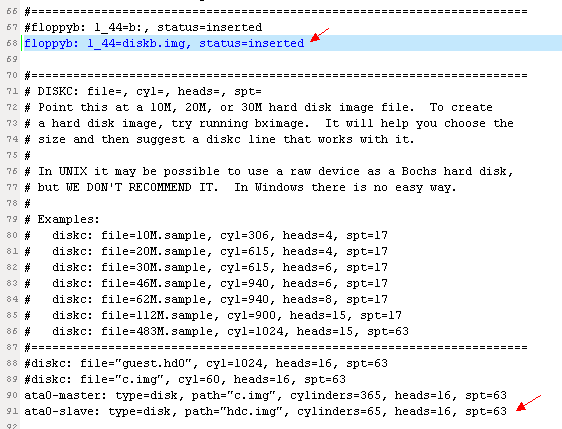
选择菜单Options->Settings，将Image选项卡中的Compression设置成None，如下所示：



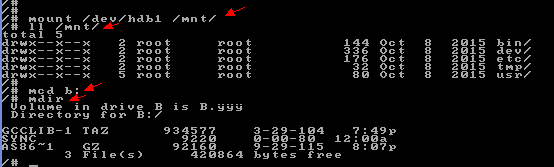
删除diskb.img中image、readme和system.map文件，将as86.tar.gz和sync文件导入该镜像，保存后退出。



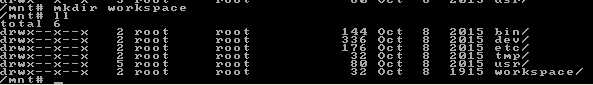
将hdc.img和diskb.img拷贝到sls linux系统目录下，修改sls.bxrc配置文件，修改内容如下：



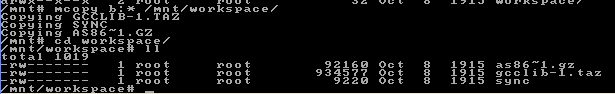
运行sls linux系统，用root登录系统。进入系统后，将hdc.img硬盘镜像文件mount到/mnt目录下。通过mcd命令进入b:目录。如下：



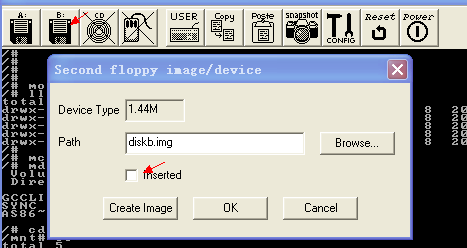
进入/mnt目录，创建workspace目录：



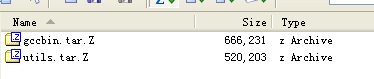
用mcopy命令将软盘镜像中的文件拷贝到workspace：



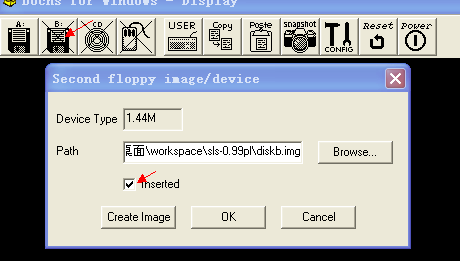
弹出软盘以使WinImage可以修改软盘镜像中的文件：



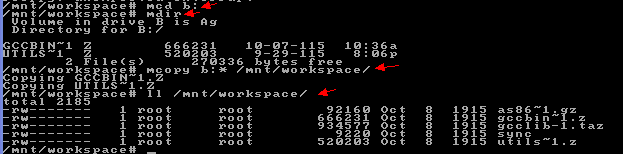
使用WinImage放入如下文件后，保存退出。



重新插入修改后的diskb.img文件：

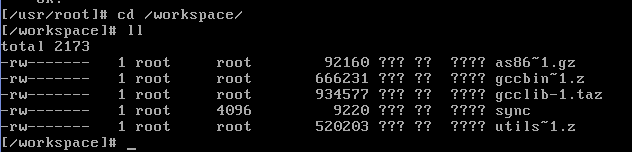


将diskb.img上的文件拷贝到/mnt/workspace/目录下



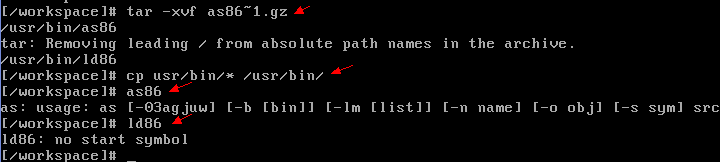
Linux0.11编译环境

将slslinux目录下的hdc.img文件写回Linux0.11目录，运行Linux0.11系统，进入workspace目录：



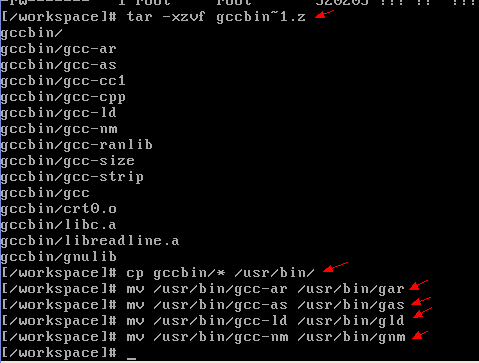
**as86~1.gz**

解压后将文件拷贝到/usr/bin目录下：



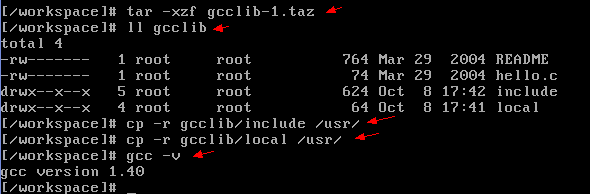
**gccbin~1.z**

解压文件后拷贝到/usr/bin目录，为部分文件改名：



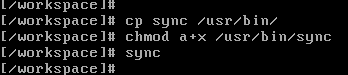
**gcclib-1.taz**

解压后做如下操作：



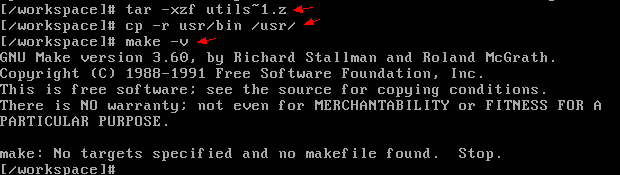
**sync**

将sync文件拷贝到/usr/bin文件夹下，添加可执行权限。



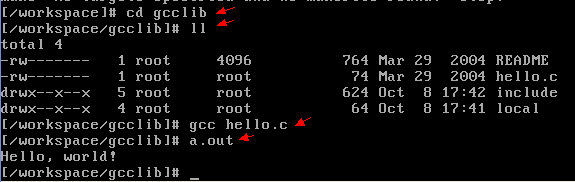
**utils~1.z**

解压文件后做如下操作：



编译hello world

Gcclib文件中提供了hello.c示例程序，编译后运行结果如下：



编译运行mtools-2.0.5

mtools用于读写 DOS 软盘，mtools-2.0.5.tar.Z是源码文件，需要在Linux0.11上编译后运行。

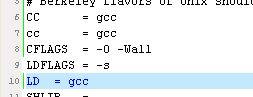
解压mtools-2.0.5.tar.Z文件后，需要修改Makefile，device.c和mread.c这三个文件。

**Makefile**

将 Makefile 6-10 行改为：

|  |
| --- |
| CC = gcc  cc = gcc  CFLAGS = -O -Wall  LDFLAGS = -s  LD = gcc |

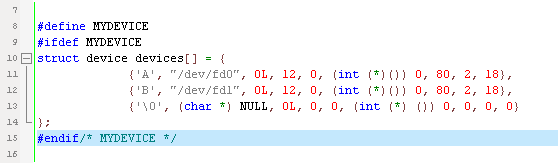
去掉原来的“LINT=lint”这一行。



**device.c**

在文件头下面添加

|  |
| --- |
| #define MYDEVICE  #ifdef MYDEVICE  struct device devices[] = {  {'A', "/dev/fd0", 0L, 12, 0, (int (\*)()) 0, 80, 2, 18},  {'B', "/dev/fd1", 0L, 12, 0, (int (\*)()) 0, 80, 2, 18},  {'\0', (char \*) NULL, 0L, 0, 0, (int (\*) ()) 0, 0, 0, 0}  };  #endif/\* MYDEVICE \*/ |



上面这些是为了 mtools 命令添加的设备，其实就是两个软盘。

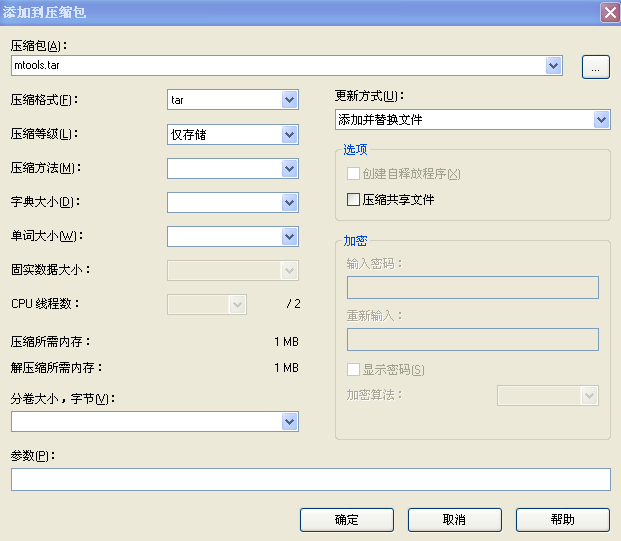
**mread.c**

在第 19 行也就是”#else /\* BSD \*/“这一行下面添加

|  |
| --- |
| #include <utime.h> |

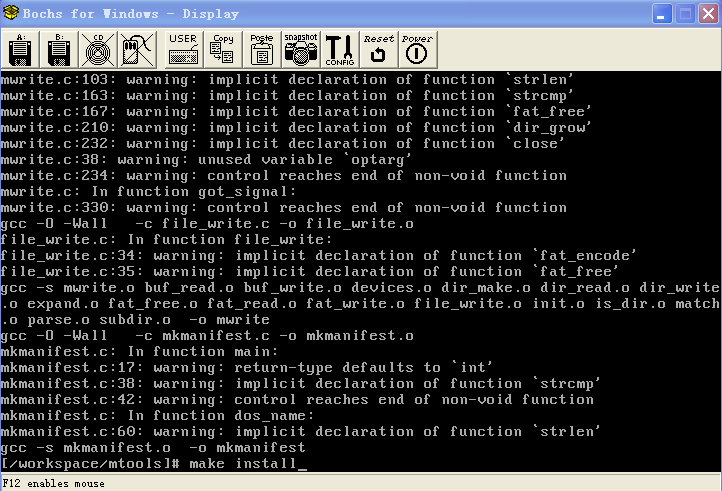


修改完毕后，用7zip程序将mtools文件夹打包成tar文件。

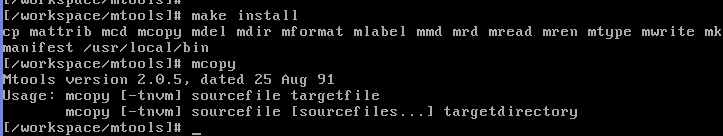


通过slslinux将mtools.tar文件放入hdc.img镜像文件。在Linux0.11中解压mtools.tar文件，进入mtools文件夹后运行make进行编译。





编译完成后运行make install安装mtools

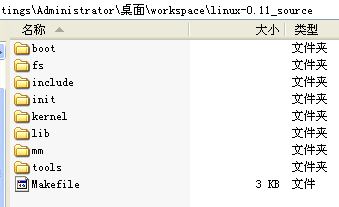


至此，在Linux0.11上可以操作dos文件系统了。

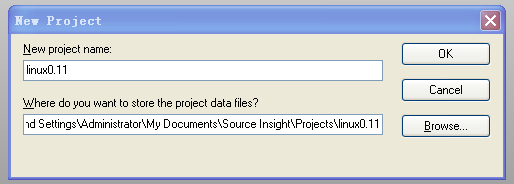
Linux0.11源码编辑环境

Source Insight管理源码

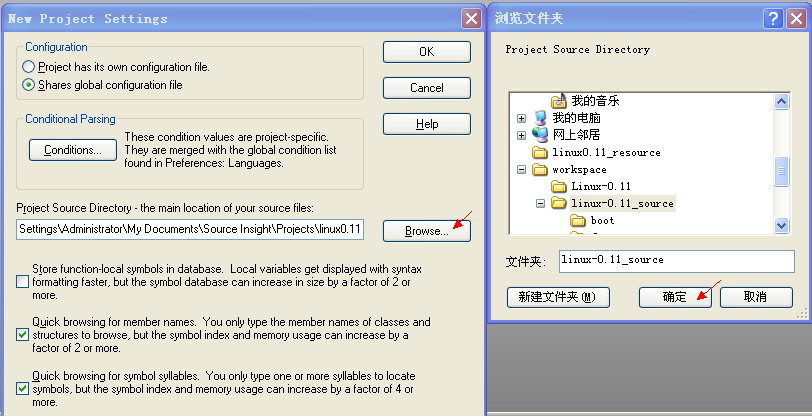
解压linux-0.11.tar.Z，得到源码文件：



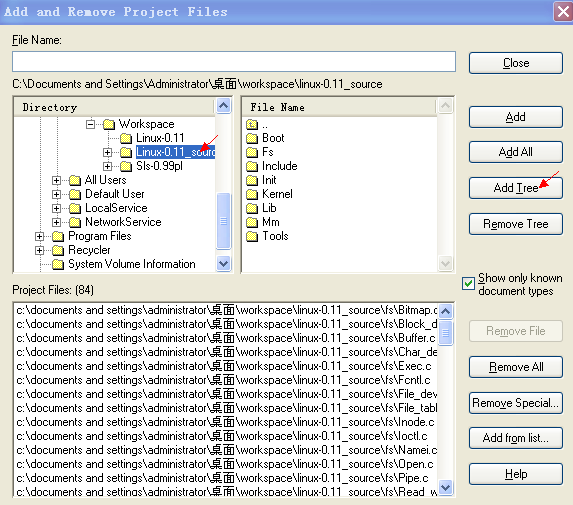
打开Source Insight，新建工程：



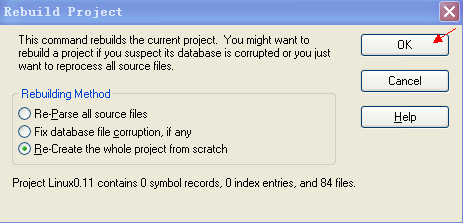
将工程源码目录定位到刚解压出来的位置



添加文件到工程



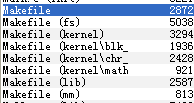
重建文件index





修改源码文件

在所有的Makefile中的CFLAGS项下去除-mstring-insns，重新打包源码文件。



编译运行Linux0.11

拷贝源码

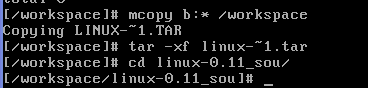
用winimage将linux0.11.tar文件导入到diskb.img文件中。



修改bochsrc-fd.bxrc文件，将软驱B修改为diskb.img文件。



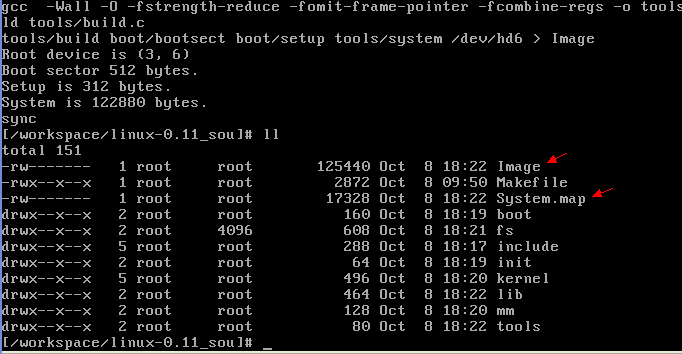
启动虚拟机，将源码tar文件拷贝到Linux0.11 /workspace目录下：



编译源码

进入源码运行make命令：

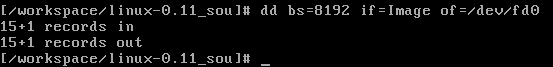


编译完成后生成如下文件：

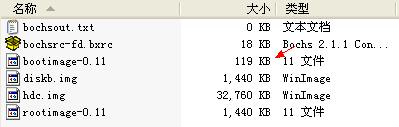
### 安装运行

通过如下命令，将boot image写入启动盘：

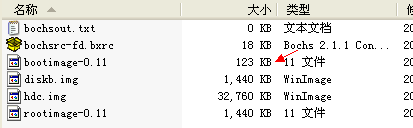
|  |
| --- |
| dd bs=8192 if=Image of=/dev/fd0 |



写入前：



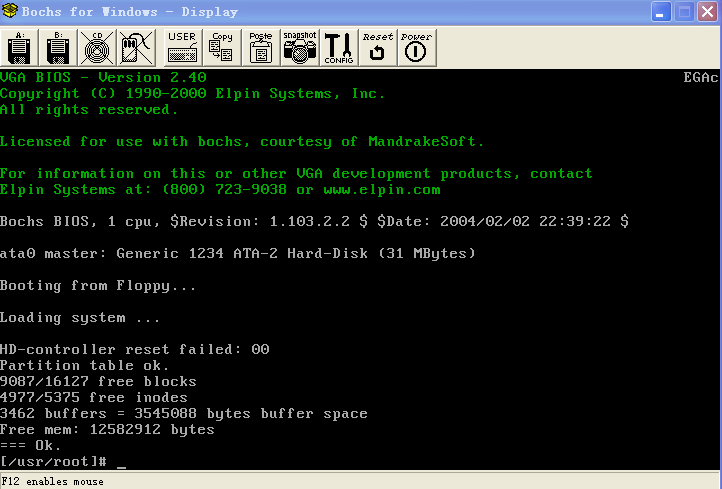
写入后：



关闭虚拟机，修改bootimage-0.11中如下字节，使之从硬盘启动：



双击bochsrc-fd.bxrc配置文件在虚拟机上运行编译好的linux0.11系统。



# 知识预备

## X86硬件

### 解析SS、SP、BP寄存器

SS:存放栈的段地址；

SP:堆栈寄存器SP(stack pointer)存放栈的偏移地址;

BP: 基数指针寄存器BP(base pointer)是一个寄存器，它的用途有点特殊，是和堆栈指针SP联合使用的，作为SP校准使用的，只有在寻找堆栈里的数据和使用个别的寻址方式时候才能用到

比如说，堆栈中压入了很多数据或者地址，你肯定想通过SP来访问这些数据或者地址，但SP是要指向栈顶的，是不能随便乱改的，这时候你就需要使用BP，把SP的值传递给BP，通过BP来寻找堆栈里数据或者地址．一般除了保存数据外,可以作为指针寄存器用于存储器寻址,此时它默认搭配的段寄存器是SS-堆栈段寄存器.BP是16位的,再扩充16位就是EBP,用于32位编程环境的.一般高级语言的参数传递等等,转换为汇编后经常由BP/EBP来负责寻址\处理.

知道ss，sp求物理地址的公式是：物理地址=ss\* 16+ sp，或者物理地址=ss\* 10H+ sp 也就是对于2进制来说，ss里的数要加上4个0 再加sp得到物理地址。比如ss 存放 1234H sp 存放 2000H ，那么物理地址就是 1234H\*10H+2000H=14340H。如果ss是2000H sp是 1234H，那么物理地址就是 21234H

SP,BP一般与段寄存器SS 联用，以确定堆栈寄存器中某一单元的地址，SP用以指示栈顶的偏移地址，而BP可作为堆栈区中的一个基地址，用以确定在堆栈中的操作数地址。

(下面这个像Win32汇编中的)

bp为基址寄存器，一般在函数中用来保存进入函数时的sp的栈顶基址

每次子函数调用时，系统在开始时都会保存这个两个指针并在函数结束时恢复sp和bp的值。像下面这样：

在函数进入时：

|  |
| --- |
|  |
| push bp // 保存bp指针  mov bp,sp // 将sp指针传给bp，此时bp指向sp的基地址。  // 这个时候，如果该函数有参数，则[bp + 2\*4]则是该子函数的第一个参数，[bp+3\*4]则是该子函数的 第二个参数，以此类推，有多少个参数则[bp+(n-1)\*4]。 |

函数结束时：

|  |
| --- |
|  |
| mov sp,bp // 将原sp指针传回给sp  pop bp // 恢复原bp的值。  ret // 退出子函数 |

下面是按调用约定\_\_stdcall 调用函数test(int p1,int p2)的汇编代码

|  |
| --- |
| http://my.oschina.net/orion/blog/15879 |
| ;假设执行函数前堆栈指针ESP为NN  push p2 ;参数2入栈, ESP -= 4h , ESP = NN - 4h  push p1 ;参数1入栈, ESP -= 4h , ESP = NN - 8h  call test ;压入返回地址 ESP -= 4h, ESP = NN - 0Ch (注意CALL指令会把返回地址压入堆栈)  ;//进入函数内  {  push ebp ;保护先前EBP指针， EBP入栈， ESP-=4h, ESP = NN - 10h  mov ebp, esp ;设置EBP指针指向栈顶 NN-10h  mov eax, dword ptr [ebp+0ch] ;ebp+0ch为NN-4h,即参数2的位置 这里可以看到了BP的作用了  mov ebx, dword ptr [ebp+08h] ;ebp+08h为NN-8h,即参数1的位置 这里可以看到了BP的作用了  ub esp, 8 ;局部变量所占空间ESP-=8, ESP = NN-18h (栈底的地址大)  ;这里就是为局部变量申请空间.  ...  add esp, 8 ;释放局部变量, ESP+=8, ESP = NN-10h (假设在上面的指令中EBP没变的话, 直接MOV ESP, EBP即可达到堆栈平衡,事实上也经常这么用)  pop ebp ;出栈,恢复EBP, ESP+=4, ESP = NN-0Ch  ret 8 ;ret返回,弹出返回地址,ESP+=4, ESP=NN-08h,后面加操作数8为平衡堆栈,ESP+=8,ESP=NN, 恢复进入函数前的堆栈为什么是8? 因为Test子函数有两个参数, 8就是对应了两个参数入栈时SP减少了8  } |

原来ESP就是一直指向栈顶的指针,而EBP只是存取某时刻的栈顶指针,以方便对栈的操作,如获取函数参数、局部变量等。

### 中断描述符表（Interrupt Descriptor Table，IDT）

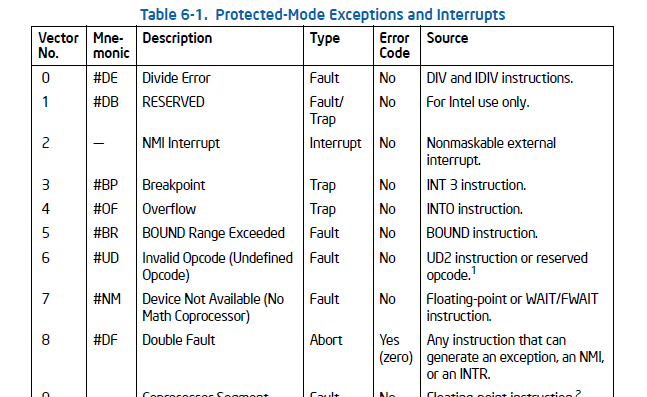
中断描述符表（Interrupt Descriptor Table，IDT）将每个异常或中断向量分别与它们的处理过程联系起来。与GDT和LDT表类似，IDT也是由8字节长描述符组成的一个数组。

#### 一、有关IDT的基本知识

1、中断是一种机制，用来处理硬件需要向CPU输入信息的情况。比如鼠标，键盘等。

2、中断和异常的产生是随机的，在CPU正常运行过程中随时可能产生。CPU的中断处理机制

3、中断可以由硬件产生（称为外部中断），也可以由软件产生（称为内部中断），在程序中写入int n指令可以产生n号中断和异常（n从0-ffh）。



4、同时CPU的中断异常机制还是重要特性的支持原理，比如程序调试，程序运行过程中的异常处理（如零除数异常、内存分页错误等）。

5、早期的操作系统甚至是通过中断来进行内核调用的。int指令是一种c从ring3 进入ring 0的方法。比如windows在xp版本之前使用的int 2e。在x86 CPU提供了sysenter指令后，这种方式才被放弃。

6、每一种中断对应一个中断号。CPU执行中断指令时，会去IDT表中查找对应的中断服务程序（interrupt service routine ，ISR）。ISR（为了表述方便后面用ISR n表示n号中断的处理程序），x86CPU最大可以支持256种中断。

7、中断是CPU的机制，不管运行的是什么操作系统，只有是运行于x86架构，IDT结构式必然存在的。IDT表中的ISRs应该有操作系统提供。

8、异常分为错误（Faults）、陷阱（Traps）和终止（Aborts）三种，其区别是“错误”允许产生错误的程序继续允许，“陷阱”也可以，但是“错误”产生于指令执行后，而“陷阱”需要在产生异常的指令执行后，因此从“错误”中返回时继续执行产生错误的指令，而从“陷阱返”回时应当从产生异常的指令后开始执行。终止产生时，处理程序不能得到精确的异常产生的代码位置，程序不能继续进行，硬件错误会产生“终止”。 例如page faults就是一个faults，mov [eax], ecx，产生了一个分页错误，表面[eax]内存是无效，需要先进行分页处理从新映射物理内存然后才能继续进行mov操作；而断点是一个Trap。

9、Intel指定或保留了前32个中断号的作用，操作系统可以指定其余的中断号的作用。

10、中断的过程中可以产生新的中断。中断时有优先级的，高优先级的中断可以“中断”低优先级的中断。有的ISR不能被中断，可以使用STI (set interrupt-enable flag) and CLI(clear interrupt-enable flag)设置IF标志来启动和关闭中断。

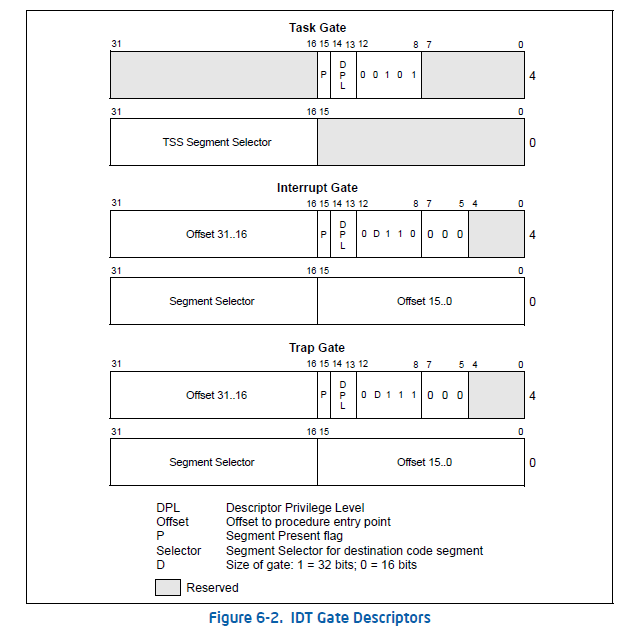
（更详细的内容可以参考[Intel® 64 and IA-32 Architectures Software Developer’s Manual](http://www.intel.com/products/processor/manuals/)）

#### 二、IDT表的结构

中断处理过程是有CPU直接调用的，CPU有专门的寄存器IDTR来保存IDT在内存中的位置，本文写为IDTR.base。IDTR有48为，前32为是IDT在内存中的位置（线性地址），后16为是IDT的大小，本文写为IDT.limit。程序可以使用LIDT和SIDT指令来读写IDTR。

IDT是一个最大为256项的表，每个表项为8字节。称为中断门。CPU通过IDT.base+n\*8来寻找门。

根据中断号对应的异常类型不同（Faults/Traps/Aborts）8个字节的意义也不同。



如上图所示IDT门中的最后两个byte是ISR在内存中的高16位，最开始前两个字节是ISR在内存中地址的低16位。

三、使用WinDbg观察、调试IDT

#### 使用Vmware配置好内核调试环境。

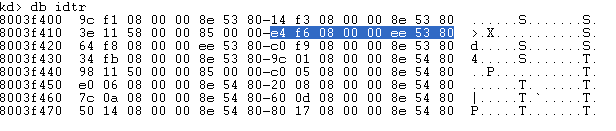
WinDbg有支持查看IDT的扩展命令!idt

!idt –a 命令可以看到所有中断处理函数的地址。  
r idtr 参看idtr寄存器中保存的idt表地址。  
idt每个表项有8bytes，两个dword大。

（注意.reload /i 加载符号文件）

|  |
| --- |
|  |
| kd> !idt -a  Dumping IDT:  00: 8053f19c nt!KiTrap00  01: 8053f314 nt!KiTrap01  02: Task Selector = 0x0058  03: 8053f6e4 nt!KiTrap03  04: 8053f864 nt!KiTrap04  05: 8053f9c0 nt!KiTrap05  06: 8053fb34 nt!KiTrap06  07: 8054019c nt!KiTrap07  08: Task Selector = 0x0050  09: 805405c0 nt!KiTrap09  0a: 805406e0 nt!KiTrap0A  0b: 80540820 nt!KiTrap0B  0c: 80540a7c nt!KiTrap0C  0d: 80540d60 nt!KiTrap0D  0e: 80541450 nt!KiTrap0E  … …  kd> r idtr  idtr=8003f400  kd> db idtr  8003f400 9c f1 08 00 00 8e 53 80-14 f3 08 00 00 8e 53 80 ......S.......S.  8003f410 3e 11 58 00 00 85 00 00-e4 f6 08 00 00 ee 53 80 >.X...........S.  8003f420 64 f8 08 00 00 ee 53 80-c0 f9 08 00 00 8e 53 80 d.....S.......S.  8003f430 34 fb 08 00 00 8e 53 80-9c 01 08 00 00 8e 54 80 4.....S.......T.  8003f440 98 11 50 00 00 85 00 00-c0 05 08 00 00 8e 54 80 ..P...........T.  8003f450 e0 06 08 00 00 8e 54 80-20 08 08 00 00 8e 54 80 ......T. .....T.  8003f460 7c 0a 08 00 00 8e 54 80-60 0d 08 00 00 8e 54 80 |.....T.`.....T.  8003f470 50 14 08 00 00 8e 54 80-80 17 08 00 00 8e 54 80 P.....T.......T. |

可以看到，这个时候IDT的基地址是在803f400处。  
注意观察其中3号中断门 e4 f6 08 00 00 ee 53 80 ISR 3   
int 3是trap门,按照之前说的ISR地址构造方法，得到8053f6e4的中断服务程序地址。



好，我们观察了IDT表（知道了IDT表的结构，自己写一个!idt WinDbg扩展也是很简单的事情），但是，使用windbg调试IDT表式不可能的。WinDbg的工作就要依赖于debuggee系统的中断向量，如果在ISR 3里写个int 3，那么BSOD是绝对的。Windbg和Debuggee系统是通过com口连接的，当Debuggee系统中产生了int3中断，系统首先进入到ISR 3中。在ISR 3中判断存在内核调试器，然后通过com口和WinDbg通行。这个处理过程本身就依赖于Debuggee系统的ISR。如果在ISR 3中又存在一个int 3那么就无限递归了。

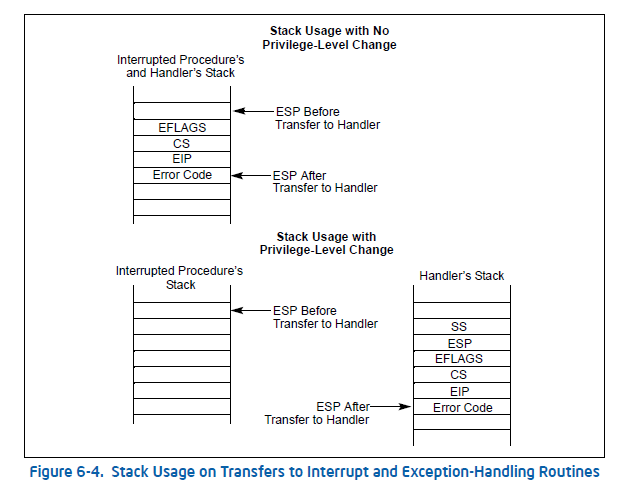
### 中断处理过程，使用bochs调试IDT中的中断服务程序

#### 一、中断处理的过程

根据Intel 64 and IA-32 Architectures Software Developer’s Manual 的介绍，在中断或异常产生是，CPU会将当前执行的指令（或下一条指令）在内存中的地址，也就是EIP的值，放入栈中，同时还会放入CS段寄存器和eflags标志寄存器的值等。

根据当前的优先级不同（ring0或ring3，也就是执行与用户态还是内核态）会有较大的不同。如果异常或中断发生时，系统正执行在内核态，那么CPU不会切换栈，直接将EFLAGS、CS、EIP和Error Code压入栈中，如果正执行在非内核态，那么先切换到内核，切换到内核栈，然后依次压入SS、ESP（用户态线程的）、EFLAGS、CS、EIP和Error Code。（上述过程CPU自行完成，操作系统只需要把ISR放到正确的位置等待调用。）然后通过IDT找到对应的ISR 开始执行。

ISR完成处理后通过IRET（IRETD）指令返回到被中断的程序中继续执行即可。

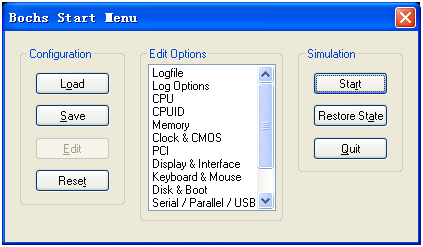


#### 二、安装配置Bochs和XP Guest系统

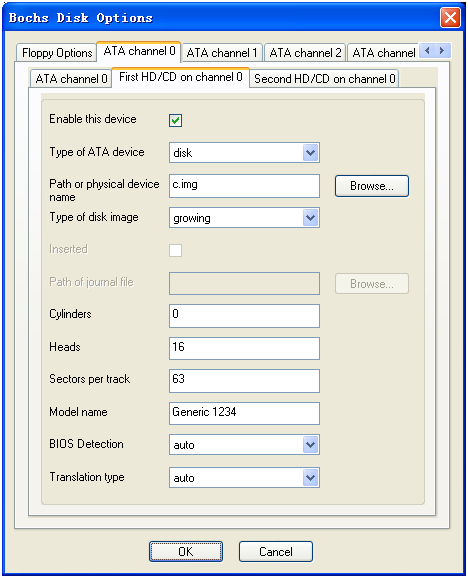
在上一篇文章中，说了为什么不能使用WinDbg调试ISR。

好在我们还有bochs这个带调试功能的软件虚拟机。bochs的调试功能比起WinDbg这个专业调试器来，功能弱了十万八千里。但是，由于bochs是一个软件实现的CPU（对Guest系统来说相当于硬件），比WinDbg要低一个层次，在一些特殊的情况下bochs就能出奇制胜了。bochs是CPU，是一切的主宰，不需要你guest系统给我提供任何支持。

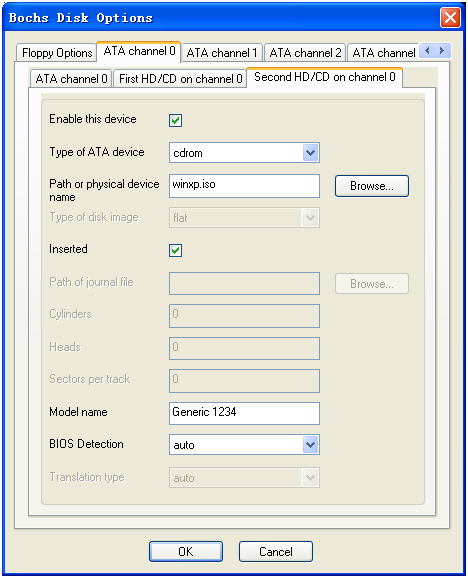
安装好bochs以后，可以在bochs中安装windows xp。bochs运行所依赖的一切硬件的配置都是依赖于bxrc配置文件。把安装目录下的bochsrc.bxrc复制一份，双击打开。在下图的对话框里可以编辑配置文件。



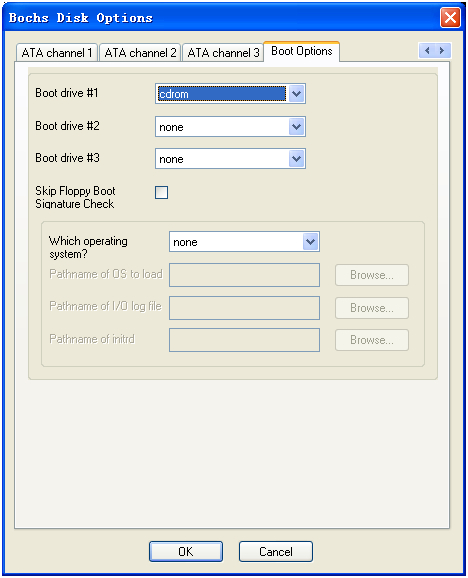
如果安装xp需要将CPU的主频调高一些（我用的50000000）memory调得合适一些，比如512Mb。在这之前还需要使用安装目录下的bximage.exe创建一个growing的硬盘文件，比如c.img。growing的方式可以避免以后guest系统硬盘不够用。



然后配置“Disk & Boot”选项，挂载上c.img

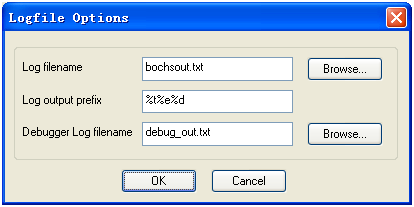


然后配置光盘。需要一个windows xp安装光盘的iso文件，也可以映射到物理光驱。



然后配置有cdrom启动。

调试过程最好log下来，便于后面分析。在“logfile”选项中配置调试的日志文件。



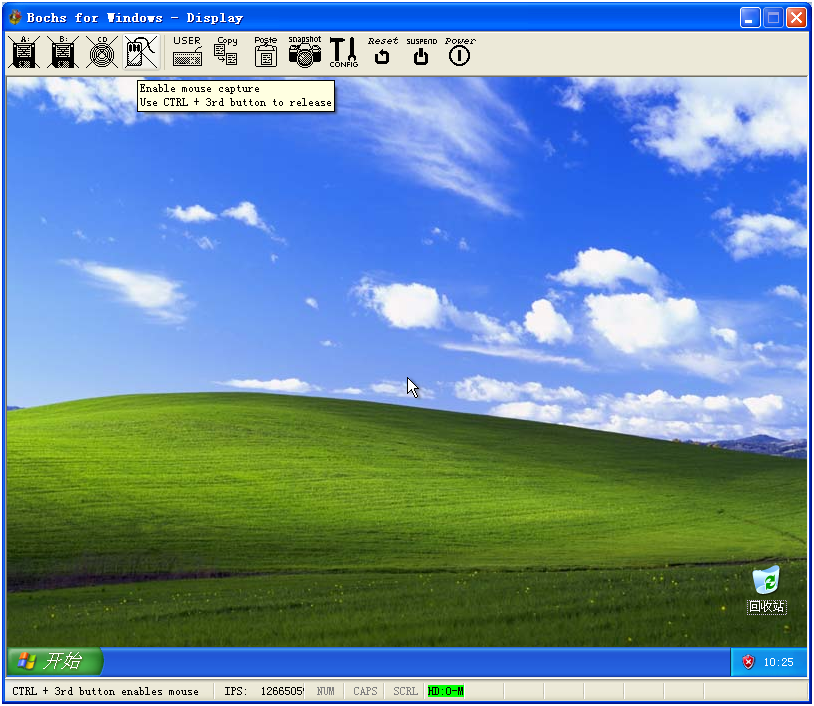
然后save，start就可以开始运行bochs了。

安装过程需要比较漫长的时间，毕竟软件CPU的性能和物理CPU不在一个数量级上。打个球，洗个澡，再吃个饭的时间应该差不多了。  
安装好以后注意备份一下，在bochs里安装xp可是一个宏大的系统工程了。然后再把boot选项设置为从disk启动。

安装好以后，就可以使用bochs调试器bochsdbg.exe加载配置文件运行了。在“Bochs Start Menu”里面load然后start。

bochsdbg会停在f000:fff0处（这个时候CPU还没有切换到保护模式）是bios的起始处。  
运行命令c（continue）可以继续系统运行。不久windows系统启动。在console里Ctrl-C可以中断运行输入各种调试命令。

操作虚拟机的鼠标需要点击按钮，释放时Ctrl+鼠标第三键（滚轮）



#### 三、使用bochs调试观察中断过程

先介绍几个调试指令（不是全部，只是在本文中用的的，其他的可以参考help或Bochs的文档）

r 查看通用寄存器

sreg 参看段寄存器（idtr被归到这里了）

creg 参看系统寄存器（cr0等）

x 参看内存（线性地址）

xp 参看内存（物理地址）

pb 通过物理地址下执行断点

lb 通过线性地址下执行断点

setpmem 修改物理内存（Bochs只支持通过物理地址修改内存，不过可以通过info tab参看整个分页表，手工转换一下，而且单步跟踪时Bochs也会同时打印指令对应的物理地址和线性地址）

help 帮助

info idt 参看idt信息

Crtl-C中断下XP系统的执行，就可以在Console中输入调试命令。

##### 1、静态参看IDT信息

<bochs:5> sreg  
es:0x0023, dh=0x00cff300, dl=0x0000ffff, valid=7  
 Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed  
cs:0x0008, dh=0x00cf9b00, dl=0x0000ffff, valid=1  
 Code segment, base=0x00000000, limit=0xffffffff, Execute/Read, Accessed, 32-bit  
ss:0x0010, dh=0x00cf9300, dl=0x0000ffff, valid=7  
 Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed  
ds:0x0023, dh=0x00cff300, dl=0x0000ffff, valid=7  
 Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed  
fs:0x0030, dh=0xffc093df, dl=0xf0000001, valid=7  
 Data segment, base=0xffdff000, limit=0x00001fff, Read/Write, Accessed  
gs:0x0000, dh=0x00001000, dl=0x00000000, valid=0  
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=0  
tr:0x0028, dh=0x80008b04, dl=0x200020ab, valid=1  
gdtr:base=0x000000008003f000, limit=0x3ff  
idtr:base=0x000000008003f400, limit=0x7ff

<bochs:6> x /100w 0x8003f400  
[bochs]:  
0x000000008003f400 <bogus+ 0>: 0x0008f19c 0x80538e00 0x0008f314 0x80538e00  
0x000000008003f410 <bogus+ 16>: 0x0058113e 0x00008500 0x0008f6e4 0x8053ee00  
0x000000008003f420 <bogus+ 32>: 0x0008f864 0x8053ee00 0x0008f9c0 0x80538e00  
0x000000008003f430 <bogus+ 48>: 0x0008fb34 0x80538e00 0x0008019c 0x80548e00  
0x000000008003f440 <bogus+ 64>: 0x00501198 0x00008500 0x000805c0 0x80548e00  
0x000000008003f450 <bogus+ 80>: 0x000806e0 0x80548e00 0x00080820 0x80548e00  
0x000000008003f460 <bogus+ 96>: 0x00080a7c 0x80548e00 0x00080d60 0x80548e00  
0x000000008003f470 <bogus+ 112>: 0x00081450 0x80548e00 0x00081780 0x80548e00  
0x000000008003f480 <bogus+ 128>: 0x000818a0 0x80548e00 0x000819d8 0x80548e00  
0x000000008003f490 <bogus+ 144>: 0x00a01780 0x80548500 0x00081b40 0x80548e00  
0x000000008003f4a0 <bogus+ 160>: 0x00081780 0x80548e00 0x00081780 0x80548e00

... ...

<bochs:3> info idt  
Interrupt Descriptor Table (base=0x000000008003f400, limit=2047):  
IDT[0x00]=32-Bit Interrupt Gate target=0x0008:0x8053f19c, DPL=0  
IDT[0x01]=32-Bit Interrupt Gate target=0x0008:0x8053f314, DPL=0  
IDT[0x02]=Task Gate target=0x0058:0x0000113e, DPL=0  
IDT[0x03]=32-Bit Interrupt Gate target=0x0008:0x8053f6e4, DPL=3  
IDT[0x04]=32-Bit Interrupt Gate target=0x0008:0x8053f864, DPL=3  
IDT[0x05]=32-Bit Interrupt Gate target=0x0008:0x8053f9c0, DPL=0  
IDT[0x06]=32-Bit Interrupt Gate target=0x0008:0x8053fb34, DPL=0

... ...

<bochs:4> info idt 3  
Interrupt Descriptor Table (base=0x000000008003f400, limit=2047):  
IDT[0x03]=32-Bit Interrupt Gate target=0x0008:0x8053f6e4, DPL=3

<bochs:8> u 0x8053f6e4 0x8053f6ff  
8053f6e4: ( ): push 0x00000000 ; 6a00  
8053f6e6: ( ): mov word ptr ss:[esp+2], 0x0000 ; 66c74424020000  
8053f6ed: ( ): push ebp ; 55  
8053f6ee: ( ): push ebx ; 53  
8053f6ef: ( ): push esi ; 56  
8053f6f0: ( ): push edi ; 57  
8053f6f1: ( ): push fs ; 0fa0  
8053f6f3: ( ): mov ebx, 0x00000030 ; bb30000000  
8053f6f8: ( ): mov fs, bx ; 668ee3  
8053f6fb: ( ): mov ebx, dword ptr fs:0x0 ; 648b1d00000000

##### 2、ring0 内核态的INT 3中断

进入Console后单步一次，看一下目前运行在什么地方。

<bochs:3> s  
Next at t=5931168643  
(0) [0x04f0833f] 0008:00000000bf80533f (unk. ctxt): pop edi ; 5f

线性地址bf80533f大于80000000应是在内核。

反汇编看一下指令

<bochs:5> u 0xbf80533f 0xbf805350  
bf80533f: ( ): pop edi ; 5f  
bf805340: ( ): pop esi ; 5e  
bf805341: ( ): leave ; c9  
bf805342: ( ): ret 0x0004 ; c20400  
bf805345: ( ): nop ; 90  
... ...

看一下bf805342对应物理内存中的数据。(s指令时知道 (0) [0x04f0833f] 0008:00000000bf80533f 的内存映射关系）。

<bochs:8> xp /20b 0x04f08342  
[bochs]:  
0x0000000004f08342 <bogus+ 0>: 0xc2 0x04 0x00 0x90 0x90 0x90 0x90 0x90  
0x0000000004f0834a <bogus+ 8>: 0x8b 0xff 0x55 0x8b 0xec 0x83 0xec 0x10  
0x0000000004f08352 <bogus+ 16>: 0x56 0x8b 0xf1 0x8b

将 bf805342: ret 0x0004 修改为int 3

<bochs:9> setpmem 0x04f08342 1 0xcc

看一下修改的结果

<bochs:11> u 0xbf80533f 0xbf805350  
bf80533f: ( ): pop edi ; 5f  
bf805340: ( ): pop esi ; 5e  
bf805341: ( ): leave ; c9  
bf805342: ( ): int3 ; cc  
bf805343: ( ): add al, 0x00 ; 0400  
... ...

中断之前的栈  
<bochs:15> x /100w rsp  
[bochs]:  
0x00000000f78807b8 <bogus+ 0>: 0xf7880808 0xe13b1008 0x00000000 0x00000000  
0x00000000f78807c8 <bogus+ 16>: 0x00000001 0x00000001 0xf7880818 0xbf808ae6  
0x00000000f78807d8 <bogus+ 32>: 0xf78807f8 0xe13b1008 0x81f187d8 0x00000001  
0x00000000f78807e8 <bogus+ 48>: 0x00000000 0x00000000 0x00000001 0x00000001  
0x00000000f78807f8 <bogus+ 64>: 0x00000000 0x00000000 0x00000001 0x00000001  
0x00000000f7880808 <bogus+ 80>: 0x820c92a8 0xe13b1008 0x00000001 0xe183c918  
0x00000000f7880818 <bogus+ 96>: 0xf788084c 0xbf80d31c 0x81f187d8 0xe13b1008  
0x00000000f7880828 <bogus+ 112>: 0x8052890c 0x820c92a8 0xe13b1008 0x00000001  
0x00000000f7880838 <bogus+ 128>: 0xf7880894 0xf7880864 0xe18c5008 0x00000000

... ...  
单步到执行到int 3

<bochs:20> s  
Next at t=5931168646  
(0) [0x04f08342] 0008:00000000bf805342 (unk. ctxt): int3 ; cc

int 3 之前寄存器和栈的情况。

<bochs:21> r  
rax: 0x00000000:f78807c0 rcx: 0x00000000:00000001  
rdx: 0x00000000:00000001 rbx: 0x00000000:e18c5008  
rsp: 0x00000000:f78807d4 rbp: 0x00000000:f7880818  
rsi: 0x00000000:e13b1008 rdi: 0x00000000:f7880808  
r8 : 0x00000000:00000000 r9 : 0x00000000:00000000  
r10: 0x00000000:00000000 r11: 0x00000000:00000000  
r12: 0x00000000:00000000 r13: 0x00000000:00000000  
r14: 0x00000000:00000000 r15: 0x00000000:00000000  
rip: 0x00000000:bf805342  
eflags 0x00000286: id vip vif ac vm rf nt IOPL=0 of df IF tf SF zf af PF cf  
<bochs:22> x /100w rsp  
[bochs]:  
0x00000000f78807d4 <bogus+ 0>: 0xbf808ae6 0xf78807f8 0xe13b1008 0x81f187d8  
0x00000000f78807e4 <bogus+ 16>: 0x00000001 0x00000000 0x00000000 0x00000001  
0x00000000f78807f4 <bogus+ 32>: 0x00000001 0x00000000 0x00000000 0x00000001  
0x00000000f7880804 <bogus+ 48>: 0x00000001 0x820c92a8 0xe13b1008 0x00000001  
0x00000000f7880814 <bogus+ 64>: 0xe183c918 0xf788084c 0xbf80d31c 0x81f187d8  
0x00000000f7880824 <bogus+ 80>: 0xe13b1008 0x8052890c 0x820c92a8 0xe13b1008  
0x00000000f7880834 <bogus+ 96>: 0x00000001 0xf7880894 0xf7880864 0xe18c5008  
0x00000000f7880844 <bogus+ 112>: 0x00000000 0x00000000 0xf788086c 0xbf81f825  
0x00000000f7880854 <bogus+ 128>: 0xe13b1008 0x00000001 0x00000000 0x00000000  
0x00000000f7880864 <bogus+ 144>: 0x0006f2e4 0xbf80cf90 0xf7880888 0xbf80d003

执行int 3，程序跳转到了 ISR 3 8053f6e4  
<bochs:23> s  
Next at t=5931168647  
(0) [0x0053f6e4] 0008:000000008053f6e4 (unk. ctxt): push 0x00000000 ; 6a00

寄存器和栈变化了：

<bochs:24> r  
rax: 0x00000000:f78807c0 rcx: 0x00000000:00000001  
rdx: 0x00000000:00000001 rbx: 0x00000000:e18c5008  
rsp: 0x00000000:f78807c8 rbp: 0x00000000:f7880818  
rsi: 0x00000000:e13b1008 rdi: 0x00000000:f7880808  
r8 : 0x00000000:00000000 r9 : 0x00000000:00000000  
r10: 0x00000000:00000000 r11: 0x00000000:00000000  
r12: 0x00000000:00000000 r13: 0x00000000:00000000  
r14: 0x00000000:00000000 r15: 0x00000000:00000000  
rip: 0x00000000:8053f6e4  
eflags 0x00000086: id vip vif ac vm rf nt IOPL=0 of df if tf SF zf af PF cf  
<bochs:25> x /100w rsp  
[bochs]:  
0x00000000f78807c8 <bogus+ 0>: 0xbf805343 0x00000008 0x00000286 0xbf808ae6  
0x00000000f78807d8 <bogus+ 16>: 0xf78807f8 0xe13b1008 0x81f187d8 0x00000001  
0x00000000f78807e8 <bogus+ 32>: 0x00000000 0x00000000 0x00000001 0x00000001  
0x00000000f78807f8 <bogus+ 48>: 0x00000000 0x00000000 0x00000001 0x00000001  
0x00000000f7880808 <bogus+ 64>: 0x820c92a8 0xe13b1008 0x00000001 0xe183c918  
0x00000000f7880818 <bogus+ 80>: 0xf788084c 0xbf80d31c 0x81f187d8 0xe13b1008  
... ...

CPU 在执行int 3时入栈的内容为 0xbf805343 0x00000008 0x00000286 共三个dword 其中 0xbf805343是产生异常的指令的下一条指令（3号中断是Trap类型），0x00000008是CS寄存器的值，0x00000286是eflags寄存器的值。  
由于int 3指令也是在ring0执行的，在这个异常过程中没有优先级的切换，所以没有切换栈，入栈的内容只有三个。

继续执行可以调试观察ISR过程

<bochs:26> trace on  
Tracing enabled for CPU0  
<bochs:27> s 100  
(0).[5931168647] [0x0053f6e4] 0008:000000008053f6e4 (unk. ctxt): push 0x00000000 ; 6a00  
(0).[5931168648] [0x0053f6e6] 0008:000000008053f6e6 (unk. ctxt): mov word ptr ss:[esp+2], 0x0000 ; 66c74424020000  
(0).[5931168649] [0x0053f6ed] 0008:000000008053f6ed (unk. ctxt): push ebp ; 55  
(0).[5931168650] [0x0053f6ee] 0008:000000008053f6ee (unk. ctxt): push ebx ; 53  
(0).[5931168651] [0x0053f6ef] 0008:000000008053f6ef (unk. ctxt): push esi ; 56  
(0).[5931168652] [0x0053f6f0] 0008:000000008053f6f0 (unk. ctxt): push edi ; 57  
(0).[5931168653] [0x0053f6f1] 0008:000000008053f6f1 (unk. ctxt): push fs ; 0fa0  
(0).[5931168654] [0x0053f6f3] 0008:000000008053f6f3 (unk. ctxt): mov ebx, 0x00000030 ; bb30000000  
(0).[5931168655] [0x0053f6f8] 0008:000000008053f6f8 (unk. ctxt): mov fs, bx ; 668ee3  
(0).[5931168656] [0x0053f6fb] 0008:000000008053f6fb (unk. ctxt): mov ebx, dword ptr fs:0x0 ; 648b1d00000000  
(0).[5931168657] [0x0053f702] 0008:000000008053f702 (unk. ctxt): push ebx ; 53  
(0).[5931168658] [0x0053f703] 0008:000000008053f703 (unk. ctxt): sub esp, 0x00000004 ; 83ec04  
... ...

略。

##### 3、ring3 用户态下的INT 3中断。

下面我们可以在ring 3 状态下产生中断，看看优先级变化时的异常处理。

重启bochs，启动XP Ctrl-C尝试停在ring 3（如果不是Ring 3 多尝试几次，CPU还是有很大比例的时间运行在ring 3态的）  
如果暂停时eip线性地址小于0x80000000就是在ring 3态下

在ring 3态下的程序重复上述过程，观察int 3前后的变化。

<bochs:9> r  
rax: 0x00000000:1e7a18d1 rcx: 0x00000000:4d532745  
rdx: 0x00000000:13f91976 rbx: 0x00000000:25dc3f3b  
rsp: 0x00000000:0063efc8 rbp: 0x00000000:ffffffe0  
rsi: 0x00000000:0009f7e0 rdi: 0x00000000:0009f870  
r8 : 0x00000000:00000000 r9 : 0x00000000:00000000  
r10: 0x00000000:00000000 r11: 0x00000000:00000000  
r12: 0x00000000:00000000 r13: 0x00000000:00000000  
r14: 0x00000000:00000000 r15: 0x00000000:00000000  
rip: 0x00000000:68021d3b  
eflags 0x00000202: id vip vif ac vm rf nt IOPL=0 of df IF tf sf zf af pf cf

<bochs:10> x /100w rsp  
[bochs]:  
0x000000000063efc8 <bogus+ 0>: 0x0063efd8 0x00000020 0x0009f7e0 0x0009f7f0  
0x000000000063efd8 <bogus+ 16>: 0x0063f004 0x6802145f 0x0009f7f0 0x25dc3f3b  
0x000000000063efe8 <bogus+ 32>: 0x0009f760 0x00000020 0x0000003b 0x0009f7e0  
0x000000000063eff8 <bogus+ 48>: 0x000a5ef8 0x0009f760 0x0009f6e0 0x0063f05c  
0x000000000063f008 <bogus+ 64>: 0x6802192d 0x0009f660 0x0009f7e0 0x000a5ef8  
0x000000000063f018 <bogus+ 80>: 0x000a4f04 0x000a500c 0x00000021 0x00000020  
0x000000000063f028 <bogus+ 96>: 0x0009f660 0x0009f6e0 0x0009f760 0x0009f7e0  
0x000000000063f038 <bogus+ 112>: 0x00000010 0x00000006 0x00000000 0x000001c5  
0x000000000063f048 <bogus+ 128>: 0x000001c5 0x00000020 0x000a5ff8 0x00000080  
... ...

<bochs:12> sreg  
es:0x0023, dh=0x00cff300, dl=0x0000ffff, valid=1  
 Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed  
cs:0x001b, dh=0x00cffb00, dl=0x0000ffff, valid=1  
 Code segment, base=0x00000000, limit=0xffffffff, Execute/Read, Accessed, 32-bit  
ss:0x0023, dh=0x00cff300, dl=0x0000ffff, valid=7  
 Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed  
ds:0x0023, dh=0x00cff300, dl=0x0000ffff, valid=7  
 Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed  
fs:0x003b, dh=0x7f40f3fd, dl=0xe0000fff, valid=1  
 Data segment, base=0x7ffde000, limit=0x00000fff, Read/Write, Accessed  
gs:0x0000, dh=0x00001000, dl=0x00000000, valid=0  
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=0  
tr:0x0028, dh=0x80008b04, dl=0x200020ab, valid=1  
gdtr:base=0x000000008003f000, limit=0x3ff  
idtr:base=0x000000008003f400, limit=0x7ff

<bochs:26> s  
Next at t=2257012356  
(0) [0x05488d4a] 001b:0000000068021d4a (unk. ctxt): int3 ; cc

<bochs:27> s  
Next at t=2257012357  
(0) [0x0053f6e4] 0008:000000008053f6e4 (unk. ctxt): push 0x00000000 ; 6a00

<bochs:28> r  
rax: 0x00000000:3edb3b9d rcx: 0x00000000:13f91977  
rdx: 0x00000000:13f91977 rbx: 0x00000000:25dc3f3b  
rsp: 0x00000000:f8149dcc rbp: 0x00000000:ffffffe0  
rsi: 0x00000000:0009f7e0 rdi: 0x00000000:0009f870  
r8 : 0x00000000:00000000 r9 : 0x00000000:00000000  
r10: 0x00000000:00000000 r11: 0x00000000:00000000  
r12: 0x00000000:00000000 r13: 0x00000000:00000000  
r14: 0x00000000:00000000 r15: 0x00000000:00000000  
rip: 0x00000000:8053f6e4  
eflags 0x00000006: id vip vif ac vm rf nt IOPL=0 of df if tf sf zf af PF cf

<bochs:30> x /100w rsp  
[bochs]:  
0x00000000f8149dcc <bogus+ 0>: 0x68021d4b 0x0000001b 0x00000206 0x0063efc8  
0x00000000f8149ddc <bogus+ 16>: 0x00000023 0x00000000 0x00000000 0x00000000  
0x00000000f8149dec <bogus+ 32>: 0x00000000 0x0000027f 0x00000000 0x00000000  
0x00000000f8149dfc <bogus+ 48>: 0x00000000 0x00000000 0x00000000 0x00001f80  
0x00000000f8149e0c <bogus+ 64>: 0x0000ffff 0x00000000 0x00000000 0x00000000  
0x00000000f8149e1c <bogus+ 80>: 0x00000000 0x00000000 0x8a885d04 0x00000048  
0x00000000f8149e2c <bogus+ 96>: 0x00000000 0x77db612a 0x00000002 0x00006148  
0x00000000f8149e3c <bogus+ 112>: 0x00000000 0x00000000 0x00000000 0x00006134  
... ...

在int 3前后的寄存器和栈都存在变化，首先不是同一栈，int 3之前是用户栈，int 3之后，切换到了系统栈。esp的值由0063efc8变为了f8149dcc，由于CONTEXT的切换，寄存器的值都发生了变化。  
在int 3的内核栈中 0x68021d4b 0x0000001b 0x00000206 0x0063efc8 0x00000023 五个值是新压入的，分别是：  
（1）0x68021d4b 产生异常的EIP的后一条指令  
（2）0x0000001b CS  
（3）0x00000206 EFLAGS  
（4）0x0063efc8 ESP  
（5）0x00000023 SS  
与文档中描述相同。

##### 4、TF标志和中断处理

调试器的单步跟踪（比如WinDbg的t命令）一般是通过CPU的TF标志实现的。如果TF标志被置位，那么在每执行一条指令后即产生一个int 1中断。

<bochs:3> r  
rax: 0x00000000:00a18f6a rcx: 0x00000000:ffdffc70  
rdx: 0x00000000:00000000 rbx: 0x00000000:ffdffc70  
rsp: 0x00000000:8054ac34 rbp: 0x00000000:8054ac50  
rsi: 0x00000000:ffdffc50 rdi: 0x00000000:821e7b68  
r8 : 0x00000000:00000000 r9 : 0x00000000:00000000  
r10: 0x00000000:00000000 r11: 0x00000000:00000000  
r12: 0x00000000:00000000 r13: 0x00000000:00000000  
r14: 0x00000000:00000000 r15: 0x00000000:00000000  
rip: 0x00000000:f871d162  
eflags 0x00000246: id vip vif ac vm rf nt IOPL=0 of df IF tf sf ZF af PF cf

eflags的第八位是TF标志位，也就是将eflag and 0x100即可将TF置位

<bochs:38> setpmem 0x006d1d35 4 0x00034668  
<bochs:39> setpmem 0x006d1d39 2 0x6600  
<bochs:40> setpmem 0x006d1d3b 1 0x9d  
<bochs:41> u rip rip+20  
806d1d35: ( ): push 0x00000346 ; 6846030000  
806d1d3a: ( ): popf ; 669d  
806d1d3c: ( ): mov ebp, esp ; 8bec  
806d1d3e: ( ): mov dword ptr ss:[esp+68], eax ; 89442444  
806d1d42: ( ): mov dword ptr ss:[esp+64], ecx ; 894c2440  
806d1d46: ( ): mov dword ptr ss:[esp+60], edx ; 8954243c

由于bochsdbg不支持设置eflag基础器的值，只能设置通用寄存器的值，这就通过修改代码将eflags置位。

单步运行：

<bochs:43> s  
Next at t=36941382598  
(0) [0x006d1d3a] 0008:00000000806d1d3a (unk. ctxt): popf ; 669d  
<bochs:44> s  
Next at t=36941382599  
(0) [0x006d1d3c] 0008:00000000806d1d3c (unk. ctxt): mov ebp, esp ; 8bec  
<bochs:46> s  
Next at t=36941382600  
(0) [0x006d1d3e] 0008:00000000806d1d3e (unk. ctxt): mov dword ptr ss:[esp+68], eax ; 89442444

TF置位是成功了的:  
<bochs:47> r  
rax: 0x00000000:00a18f6a rcx: 0x00000000:ffdffc70  
rdx: 0x00000000:00000000 rbx: 0x00000000:ffdffc70  
rsp: 0x00000000:8054ac22 rbp: 0x00000000:8054ac22  
rsi: 0x00000000:ffdffc50 rdi: 0x00000000:821e7b68  
r8 : 0x00000000:00000000 r9 : 0x00000000:00000000  
r10: 0x00000000:00000000 r11: 0x00000000:00000000  
r12: 0x00000000:00000000 r13: 0x00000000:00000000  
r14: 0x00000000:00000000 r15: 0x00000000:00000000  
rip: 0x00000000:806d1d3e  
eflags 0x00000346: id vip vif ac vm rf nt IOPL=0 of df IF TF sf ZF af PF cf  
<bochs:48> s  
Next at t=36941382601  
(0) [0x0053f314] 0008:000000008053f314 (unk. ctxt): push 0x00000000 ; 6a00

已经进入ISR 1  
（这里不知为是么置位了TF后单步了两条指令才跳入ISR 1，是bochs的实现原理问题？存疑在此。）

<bochs:49> info idt  
Interrupt Descriptor Table (base=0x000000008003f400, limit=2047):  
IDT[0x00]=32-Bit Interrupt Gate target=0x0008:0x8053f19c, DPL=0  
IDT[0x01]=32-Bit Interrupt Gate target=0x0008:0x8053f314, DPL=0  
... ...

### DEBUG.EXE各命令详细说明

#### 1.打开Windows命令窗口

在Windows 95/98的环境中，打开命令窗口的步骤为：点击“开始”→“运行”，输入“command”命令； 在WindowsXP及WIN7的环境中，打开命令窗口的步骤为：点击“开始”→“运行”，输入“cmd”命令；

#### 2.启动DEBUG

在命令窗口中启动DEBUG，启动命令一般为：DEBUG [文件名] [参数表]。其中：文件名指定被调试的文件，其包括名和后缀，参数表是被调试文件运行时所需要的参数。被调试的文件可以是系统中的任何文件，但通常它们的后缀为.EXE或.COM。 当DEBUG启动成功后，将显示连接符“-”，这时，可输入各种DEBUG命令。DEBUG中所有命令及其含义如[DEBUG各命令功能说明](http://www.shelwee.com/html/archives/65556.html)表所示。 关于使用命令的几点说明：在提示符“-”下才能输入命令，在按“回车”键后，该命令才开始执行命令是单个字母，命令和参数的大小写可混合输入可用F1、F2、F3、Ins、Del、左移键、右移键等编辑键来编辑本行命令当命令出现语法错误时，将在出错位置显示“^ Error”可用Ctrl+C或Ctrl+Break来终止当前命令的执行，还可用Ctrl+S或Ctrl+Num Lock来暂停屏幕显示(当连续不断地显示信息时)  
以下通过实现十九个示例来熟悉DEBUG的命令集和基本的汇编指令。

#### R命令的使用

R命令作用：观看和修改寄存器的值。

在提示符“-”下输入以下命令：R。DEBUG将会显示出当前所有寄存器和标志位的状态。

接下来再输入命令RCX。在提示符“：”后输入100。该命令的作用是将寄存器CX的值设置为100（注意：DEBUG使用的是十六进制，这里的100相当于十进制的256。）

最后再执行R命令，观看修改后的寄存器值。



#### H命令的使用

H命令作用：计算两个十六进制数的和与差。

在提示符“–”下输入以下命令：H 10 1。观看命令执行结果。

http://shelwee.qiniudn.com/wp-content/uploads/2011/04/debug-h.png

运行结果的前一个数是计算出来的和，后一个数是计算出来的差。计算结果均用十六进制形式表示。

#### D命令的使用

D命令作用：显示内存区域的内容。

在提示符“–”下连续执行命令R、D、D。观看命令执行结果。

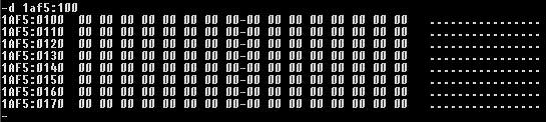


前面已经介绍过了，命令R的作用是显示当前寄存器的值。而命令D的作用是显示内存区域的内容，最左边是内存的起始地址，中间以十六进制的形式显示内存值，最右边是以ASCII码的形式显示内存值。每行最多显示16个字节的内容。

命令D可以带参数也可省略参数。设DEBUG启动时DS的值为X，当省略参数时，命令D显示内容以X：100为起始，每次显示128个字节的内容。以后再执行不带参数的命令D时，DEBUG将按上次的位置接着显示下去。

带参数时DEBUG能够显示指定地址范围的内容。带参数的方式有三种：

方式一：d [起始位置]。DEBUG从起始位置开始显示128个字节的内容。在提示符“-”下执行命令D 1AF5:100。观看命令执行结果。



方式二：d [起始位置] [结束位置]。DEBUG从起始位置开始一直显示到结束位置。在提示符“-”下执行命令D DS:100 1FF。观看命令执行结果。

http://shelwee.qiniudn.com/wp-content/uploads/2011/04/debug-d3.png

方式三：d [起始位置] [L长度]，长度以L参数为标识。DEBUG从起始位置开始显示指定长度的内容。在提示符“-”下执行命令D DS:100 L10。观看命令执行结果。

http://shelwee.qiniudn.com/wp-content/uploads/2011/04/debug-d4.png

#### E命令的使用

E命令作用：改变内存单位的内容。

E命令的使用方式为：E [起始位置]。

在提示符“-”下输入以下命令：E 1AF5:100。

http://shelwee.qiniudn.com/wp-content/uploads/2011/04/debug-e.png

DEBUG首先显示[1AF5:0000]的内容00.，这时可以修改该字节的值。如果还要修改后续的内容，可以按空格键继续。当要跳过某个字节时，可以按连续的两个空格跳到后一个字节去。

#### F命令的使用

F命令作用：使用指定的值填充指定内存区域中的地址。

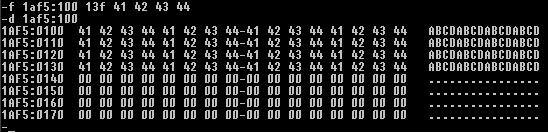
F命令的使用方式为：F [范围] [填充列表]。

在提示符“-”下输入以下命令：F 1AF5:100 L20 1 2 3 4 5。执行命令D 1AF5:100观看命令执行结果。

http://shelwee.qiniudn.com/wp-content/uploads/2011/04/debug-f1.png

说明：该命令是用字节序列01、02、03、04、05轮流填充从1AF5:100开始长度为20H的内存区域。

在提示符“-”下输入以下命令：F 1AF5:100 13F 41 42 43 44。



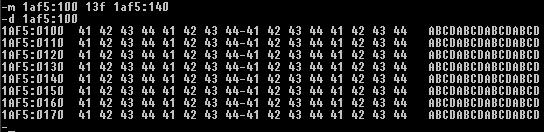
说明：该命令是用字节序列41、42、43、44轮流填充从1AF5:100开始一直到1AF5:13F的内存区域。

#### M命令的使用

M命令作用：将指定内存区域的数据复制到指定的地址去。

M命令的使用方式为：M [范围] [指定地址]。

在提示符“-”下输入以下命令：M 1AF5:100 13F 1AF5:140。执行命令D 1AF5:100观看命令执行结果。



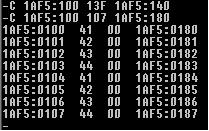
#### C命令的使用

C命令作用：将两块内存的内容进行比较。

C命令的使用方式为：C [范围] [指定地址]，意思就是将指定范围的内存区域与从指定地址开始的相同长度的内存区域逐个字节进行比较，列出不同的内容。

在提示符“-”下输入以下命令：C 1AF5:100 13F 1AF5:140。由于两块内容完全相同，所以命令执行后没有任何显示。

在提示符“-”下输入以下命令：C 1AF5:100 107 1AF5:180，比较的区域长度为8个字节。命令执行后列出比较结果不同的各个字节。



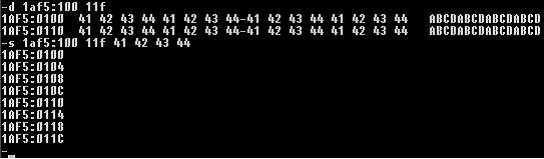
#### S命令的使用

S命令作用：在指定的内存区域中搜索指定的串。

S命令的使用方式为：S [范围] [指定串]。

在提示符“-”下输入以下命令：D 1AF5:100 11F。显示该区域的内存值。

在提示符“-”下输入以下命令：S 1AF5:100 11F 41 42 43 44。搜索该区域是否存在字节串41 42 43 44，并将搜索结果一一列出。



从执行结果可以看出，总共搜索到八处。

#### A命令的使用

A命令作用：输入汇编指令。

以下的程序要在屏幕上显示“ABCD”四个字符。

首先用E命令将“ABCD$”四个字符预先放在内存CS:200处，然后执行A100命令输入汇编程序代码：

MOV AX,CS

MOV DS,AX

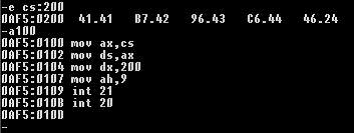
MOV DX,200

MOV AH,9

INT 21

INT 20

（说明：前两行汇编指令用于将段寄存器CS的值赋给段寄存器DS。第三到第五行汇编代码的作用是显示以“$”为结尾的字符串。最后一行用于结束程序。

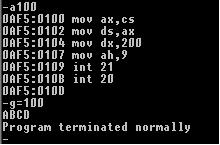


#### G命令的使用

G命令作用：执行汇编指令。

G命令的使用方法是：G [=起始地址] [断点地址]，意思是从起始地址开始执行到断点地址。如果不设置断点，则程序一直运行到中止指令才停止。

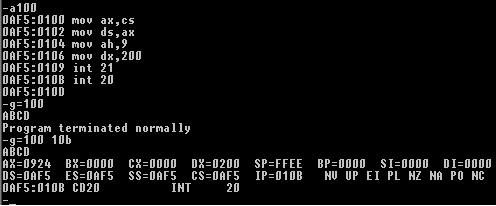
在设置完示例九的的内存数据并且输入完示例九的程序后运行这些汇编代码。在DEBUG中执行命令G=100，观看运行结果。



汇编程序运行后在屏幕上显示出“ABCD”四个字符。

接下来在DEBUG中执行G=100 10B，意思是从地址CS：100开始，一直运行到CS：10B停止。观看运行结果。

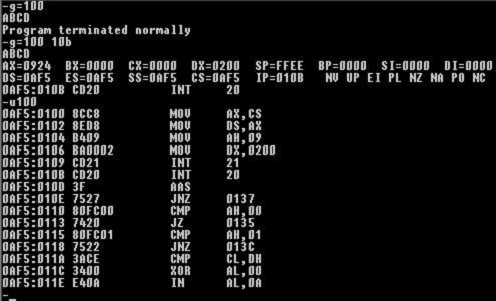
命令执行后，不但显示出字符串“ABCD”，而且列出当前寄存器和标志位的值。



#### U命令的使用

U命令作用：对机器代码反汇编显示。

U命令的使用方法是：U [范围]。如果范围参数只输入了起始地址，则只对20H个字节的机器代码反汇编。执行命令U100，观看反汇编结果。



执行命令U100 10B，观看反汇编结果。该命令的作用是对从100到10B的机器代码进行反汇编。



#### N命令的使用

N命令作用：设置文件名，为将刚才编写的汇编程序存盘做准备。

以下的DEBUG命令序列作用将刚才的汇编程序存为磁盘的COM可执行程序。

D200 20F

U100 10C

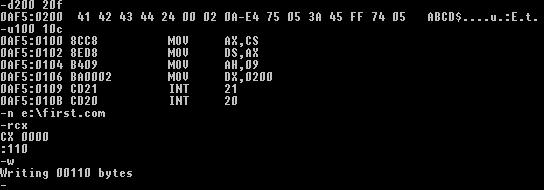
N E:\FIRST.COM

RCX

:110

W

第一和第二条命令的作用是检查一下刚才编写的汇编指令。第三条命令的作用是设置存盘文件名为E:\FIRST.COM，第四条命令的作用是设置存盘文件大小为110H个字节。最后一条命令是将文件存盘。



文件存盘后执行E:\FIRST.COM，观看存盘的可执行文件的运行效果。

http://shelwee.qiniudn.com/wp-content/uploads/2011/04/debug-n2.png

#### W命令的使用

W命令作用：将文件或者特定扇区写入磁盘。

在示例“N命令的使用”中已经实验了如何使用W命令将文件存盘。

在没有很好地掌握汇编语言和磁盘文件系统前，暂时不要使用W命令写磁盘扇区，否则很容易损坏磁盘文件，甚至破坏整个磁盘的文件系统。

L命令的使用

L命令作用：从磁盘中将文件或扇区内容读入内存。

将文件调入内存必须先用DEBUG的N命令设定文件名。以下例子是将E:\FIRST.COM读入内容。

N FIRST.COM

L

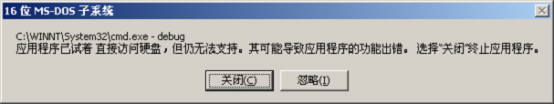
观看调入程序的汇编代码可以使用DEBUG的U命令，用U100观看调入的COM文件。



读取磁盘扇区的方式是：L [内存地址] [磁盘驱动器号] [起始扇区] [扇区数]。“内存地址”指定要在其中加载文件或扇区内容的内存位置，如果不指定“内存地址”的话，DEBUG将使用CS寄存器中的当前地址。“磁盘驱动器号”指定包含读取指定扇区的磁盘的驱动器，该值是数值型：0=A，1=B，2=C等。“起始扇区”指定要加载其内容的第一个扇区的十六进制数。“扇区数”指定要加载其内容的连续扇区的十六进制数。

只有要加载特定扇区的内容而不是加载文件时，才能使用[磁盘驱动器号] [起始扇区] [扇区数]参数。

例如：要将C盘第一扇区读取到内存DS:300的位置，相应的DEBUG命令为L DS:300 2 1 1。但是由于Windows操作系统对文件系统的保护，这条命令可能会被操作系统禁止运行。

[](http://shelwee.qiniudn.com/wp-content/uploads/2011/04/debug-l1.png)

#### T命令的使用

T命令作用：执行汇编程序，单步跟踪。

T命令的使用方式是T [=地址] [指令数]。如果忽略“地址”的话，T命令从CS:IP处开始运行。“指令数”是要单步执行的指令的数量。

以下示例对E:\FIRST.COM进行单步跟踪。

N E:\FIRST.COM

L

U100 10B

R

T=100

T

[](http://shelwee.qiniudn.com/wp-content/uploads/2011/04/debug-t.png)

第一、二条命令是装入文件，第三条命令是列出程序反汇编代码，第四条命令是显示当前寄存器值，第五条命令是从CS:100处开始单步跟踪，第六条命令是继续跟踪后续的指令。

P命令的使用

P命令作用：执行汇编程序，单步跟踪。与T命令不同的是：P命令不会跟踪进入子程序或软中断。

P命令的使用方式与T命令的使用方式完全相同。

[](http://shelwee.qiniudn.com/wp-content/uploads/2011/04/debug-p.png)

#### I命令的使用

I命令作用：从计算机输入端口读取数据并显示。

I命令的用法是I [端口地址]。例如从3F8号端口读取数据并显示的命令为：I 3F8。这里不对该命令做解释。

O命令的使用

O命令作用：向计算机输出端口送出数据。

O命令的用法是O [端口地址] [字节值]。例如向278号端口发出数据20H的命令为：I 278 20。这里不对该命令做解释。

Q命令的使用

Q命令的作用是退出DEBUG，回到DOS状态。

## 汇编语言

### F&Q

**已知(SP)=2000H，指令PUSH AX 执行后（AX)=?**

执行PUSH指令时，先执行SP-2，然后把AX赋给SP，AX内容不变SP=2000H-2,2000H是十六进制，借位时要借16，所以-2=1FFEH,即SP=1FFEH

# Linux 0.11

### fork()详解

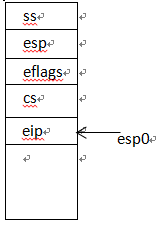
由于创建新进程时是完全通过复制父进程的代码段和数据段的方式来完成的。因此在创建新进程时，为了确保子进程的用户态堆栈没有父进程的多余信息，要求父进程在创建子进程时不要调用函数。因此fork()不能以函数形式被调用。V0.11中通过定义内嵌宏代码，调用linux的系统调用中断0x80首先fork()调用。

宏定义如下：

|  |
| --- |
|  |
| #define \_syscall0(type,name) \  type name(void) \  { \  long \_\_res; \  \_\_asm\_\_ volatile ( "int $0x80" \ // 调用系统中断0x80。  :"=a"  (\_\_res) \ // 返回值??eax(\_\_res)。  :""  (\_\_NR\_  ##name)); \ // 输入为系统中断调用号\_\_NR\_name。  if (\_\_res >= 0) \ // 如果返回值>=0，则直接返回该值。  return (type) \_\_res; errno = -\_\_res; \ // 否则置出错号，并返回-1。  return -1;  }    static inline \_syscall0(int, fork) |

当调用fork创建子进程时上面的内联宏被调用，系统的中断调用号为\_NR\_fork被存入eax中，同时调用系统调用中断0x80，将用户态的ss, esp, eflags, cs, eip压栈

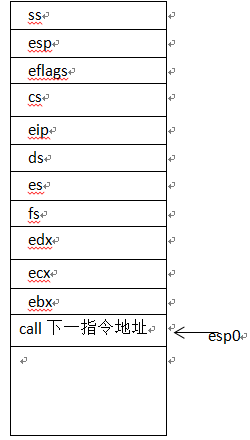
进程堆栈转入内核栈:

[](http://photo.blog.sina.com.cn/showpic.html#blogid=7d55e87a01016b8m&url=http://s15.sinaimg.cn/orignal/7d55e87axcdd4756738de)

0x80调用system\_call:

|  |
| --- |
|  |
| \_system\_call:  cmpl $nr\_system\_calls-1,eax #调用号若超出范围就在eax中置-1并退出  ja bad\_sys\_call  push ds 保存原寄存器的值  push es  push fs  pushl edx #若有参数，则edx,ecx,ebx中装有函数的参数  pushl ecx  pushl ebx  movl $0x10,edx  mov dx,ds 将ds,es指向内核数据段（全局描述附表GDT中数据段描述符）  mov dx,es  movl $0x17,edx 将fs指向局部数据段描述符（局部描述表LDT中数据段描述符）  mov dx,fs  call \_sys\_call\_table(,eax,4)调用系统调用处理函数数组中的第eax个函数  pushl eax |

调用call \_sys\_call\_table(\_NR\_fork,4)内核栈状态：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=7d55e87a01016b8m&url=http://s9.sinaimg.cn/orignal/7d55e87axcdd471264418)

\_sys\_call\_table[eax\*4]指向函数\_sys\_fork

|  |
| --- |
|  |
| \_sys\_fork:  call \_find\_empty\_process #获得新进程的进程号pid=last\_pid，返回进程在进程数组task[]中的任务号nr存入eax，nr最大为63，因为linux 0.11最多同时支持64个进程同时运行，  testl eax,eax  js 1f  push gs  pushl esi  pushl edi  pushl ebp  pushl eax  call \_copy\_process  addl $20,esp  ret |

相信此时调用\_copy\_process时内核堆栈的使用情况你应该了解了，这里我就不在画出来了。

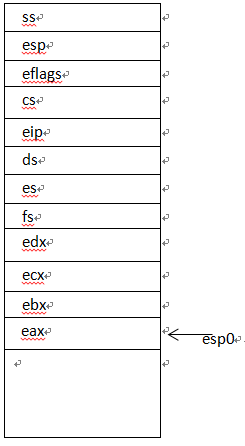
\_copy\_process实现功能

1. 在内存中为新进程申请一页内存作为新进程的内核堆栈，并将新进程的任务数据结构存入栈底部。
2. 对从父进程复制来的任务数据结构做一定的修改。
3. 在线性地址空间中设置新进程的基地址和限长，并将父进程的数据段和代码段（其实是同一段）复制到子进程的数据段和代码段。
4. 如果父进程打开某文件，则将次文件的引用次数加一， 将父进程的pwd,root, executable引用次数加一，因为子进程也引用了这些i节点。
5. 在全局段描述符表GDT中加入子进程的TSS，LDT段描述符，返回子进程的pid，回到\_sys\_fork

addl $20,esp 将\_copy\_process压栈的所有内容全部退栈。此时eax中存的是子进程的pid

从\_sys\_fork返回到\_system\_call，将子进程号压栈

此时的内核堆栈：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=7d55e87a01016b8m&url=http://s13.sinaimg.cn/orignal/7d55e87axcdd46ba354bc)

这里我们不在考虑进程处于非就绪态和时间片用完的情况。

将内核栈中内容依次pop出栈存到相应寄存器中，再调用iret使得堆栈指针重新指到进程用户态堆栈，回到进程用户态继续执行程序，子进程会在进程调度函数schedule()执行时被调用

1. Ref Links

http://www.oldlinux.org/Linux.old/kernel/0.1x/