

Linux 0.11 在ubuntu-11和gcc-4.6.1下编译调试至正常运行的过程 详解

分类： LINUX 2012-04-12 20:40:30

李显杰

引言

1、 工作背景

研究linux-0.11已有月余，代码看了好几遍，虽然能看懂，却总感觉如隔靴搔痒一般。这当然是正常的，毕竟不是自己写的东西，毕竟这是个操作系统。但若能亲自修改并调试一番，对其的理解就能更加深刻了。然而在linux-0.11的编译要在装有gcc1.4的系统里进行，那个系统太老了，修改起来也不方便，又没法安装源码阅读器...那能不能在ubuntu下用高版本的gcc编译并运行呢？

2、 工作意义

简单讲下本文所描述工作以及撰写本文的意义吧。

Linux-0.11在高版本gcc-4.6.11下编译调试并能正常运行可以使学习linux-0.11更加方便、容易，也为进一步研究linux2.6奠定基础。更加重要的是，本项工作的过程其实就是一个深入学习的过程，对了解新旧版本gcc之间的差异、linux内核运行机制、linux调试方法有很大帮助。另外，对于linux-0.11在高版本gcc下的编译工作，网上已有不少先驱者，但大都没能让编译过后的os正常运行起来，偶有高人完成，也未能提供足够的调试材料，这让不少初学者对此失去了兴趣。现在有兴趣于此的大哥mm们High起来吧！

当然这项工作在很多人眼里并没多重要，他们认为它一没有科研创新价值，而没有商业盈利价值，还耗费不少时间，他们有更重要的事情去做。这个浮躁的社会里，人们做什么事情总要讲究足够的理由。对此我只能说，你们是没错的。

然我道中人终究还有不少，撰写本文也是希望跟我相同想法的人能从中获得他们需要的帮助，并借此与他们互相勉励一下。有句老话怎么说来着？重要的不是目的地，而是沿途的风景。

当然若有高人前辈路过，愿意批评指正的，望不吝赐教，我将不甚感激涕零。

3、 工作目的：

本文工作的目的在ubuntu-11中使用gcc-4.6.1成功编译linux-0.11源码，并经过调试使之能正常运行。由于时间关系，并未解决系统中所有存在的bug，十分遗憾，若有大哥mm们愿继续完成，我将非常高兴。虽如此，拥有了在这个过程中学到的这些东西，我的工作已有足够的意义。本文的特点是图文并茂，对错误原因解释详细，关键地方有对linux一些机制的阐述，尽量减少读者的理解障碍。

代码包来源：赵炯博士网站<http://oldlinux.org/Linux.old>中下载bochs安装文件、linux-0.11.tar.Z源码包、linux-0.11-devel-040923.zip源码包（该源码在gcc1.4下编译通过，作为对照使用）。

平台搭建：我使用的环境是VirtualBox中安装ubuntu-11，与win7共享工作文件夹（其中存有linux-0.11源码），win7中使用editplus和source insight进行源码的编辑和阅读。

一、编译过程

对linux-0.11在高版本gcc下的编译过程，网上也有一些散碎材料，赵炯博士的论坛里和《linux内核完全剖析》中也有一部分介绍。本节依据先后顺序将编译过程中每个错误的解决方法依次完整地

罗列出来。

1、除错工具：

(1) sourceInsight，源码阅读器肯定需要了，否则岂不是找崩溃吗？

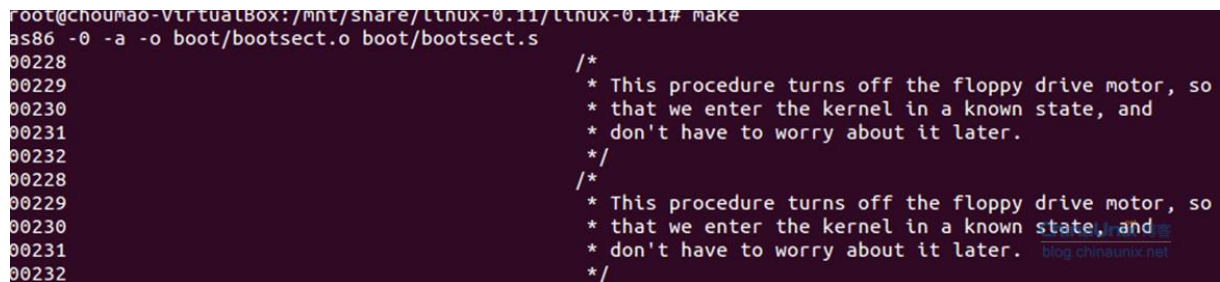
(2) internet，编译过程网上有不少人完成了，赵炯博士也完成了，对于一些问题可以查到相应资料。

(3) editplus，这是个编辑器了，我用之多年，比较喜欢，其特点就是小但功能全。

2、问题及解决方法

问题1：

第一次make会出现如图1.1所示的问题。



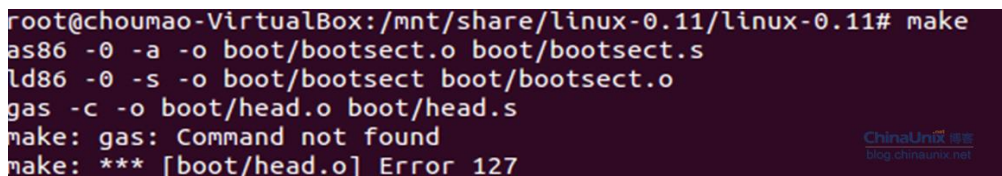
```
root@choumao-VirtualBox:/mnt/share/linux-0.11/linux-0.11# make
as86 -0 -a -o boot/bootsect.o boot/bootsect.s
00228      /*
00229      * This procedure turns off the floppy drive motor, so
00230      * that we enter the kernel in a known state, and
00231      * don't have to worry about it later.
00232      */
00228      /*
00229      * This procedure turns off the floppy drive motor, so
00230      * that we enter the kernel in a known state, and
00231      * don't have to worry about it later.
00232      */
```

图1.1

这是注释方式问题，as86不支持/**/, 改为每行前面用!注释。

问题2：

再次make后出现如图1.2所示的错误。



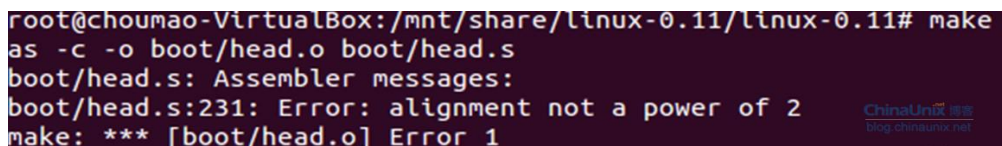
```
root@choumao-VirtualBox:/mnt/share/linux-0.11/linux-0.11# make
as86 -0 -a -o boot/bootsect.o boot/bootsect.s
ld86 -0 -s -o boot/bootsect boot/bootsect.o
gas -c -o boot/head.o boot/head.s
make: gas: Command not found
make: *** [boot/head.o] Error 127
```

图1.2

修改所有makefile文件的编译器修改一下，将gas,gld改成as和ld，所有AS的选项参数去掉-c（as不用这个选项），gcc的选项参数去掉-fcombine-regs -mstring-insns（这两个参数现在的gcc不支持了）。

问题3：

接着又出现如图1.3所示的编译错误。



```
root@choumao-VirtualBox:/mnt/share/linux-0.11/linux-0.11# make
as -c -o boot/head.o boot/head.s
boot/head.s: Assembler messages:
boot/head.s:231: Error: alignment not a power of 2
make: *** [boot/head.o] Error 1
```

图1.3

这是新旧版本gcc对齐的表达方式不同所造成的，.align n n改为 2ⁿ便可。

问题4：

接着又出现如图1.4所示的编译错误。

```

root@choumao-VirtualBox: /mnt/share/linux-0.11/linux-0.11# make
as -c -o boot/head.o boot/head.s
gcc -Wall -O -fstrength-reduce -fomit-frame-pointer \
-nostdinc -Iinclude -c -o init/main.o init/main.c
In file included from init/main.c:8:0:
include/unistd.h:207:1: warning: function return types not compatible due to 'volatile' [enabled by default]
include/unistd.h:207:1: warning: function return types not compatible due to 'volatile' [enabled by default]
include/unistd.h:208:1: warning: function return types not compatible due to 'volatile' [enabled by default]
include/unistd.h:208:1: warning: function return types not compatible due to 'volatile' [enabled by default]
init/main.c:23:15: error: static declaration of 'fork' follows non-static declaration
init/main.c:24:15: error: static declaration of 'pause' follows non-static declaration
include/unistd.h:224:5: note: previous declaration of 'pause' was here
init/main.c:26:15: error: static declaration of 'sync' follows non-static declaration
include/unistd.h:235:5: note: previous declaration of 'sync' was here
init/main.c:104:6: warning: return type of 'main' is not 'int' [-Wmain]
init/main.c: In function 'init':
init/main.c:182:9: warning: function with qualified void return type called [enabled by default]
init/main.c:184:8: warning: function with qualified void return type called [enabled by default]
init/main.c:200:9: warning: function with qualified void return type called [enabled by default]
init/main.c:208:7: warning: function with qualified void return type called [enabled by default]
make: *** [init/main.o] Error 1

```

图1.4

上面的错误只有static declaration of 'pause' follows non-static declaration这类是必须修改的，查看main.c和unistd.h发现产生错误的原因：

Main.c中有如图1.5所示的语句：

```

24 static inline _syscall0(int, fork)
25 static inline _syscall0(int, pause)
26 static inline _syscall1(int, setup, void * BIOS)
27 static inline _syscall0(int, sync)

```

图1.5

而unistd.h文件中有图1.6所示的声明语句：

```

int fork(void);
int pause(void);
int sync(void);

```

图1.6

Main.c头部包含了unistd.h，那么关于fork、pause、sync系统调用的声明就重复了，不知道为什么旧版本的gcc可以编译通过，需要详查，在main.c中增加这些函数的内联声明的原因是linux-0.11的进程0和1共用一个栈，进程0在创建了进程1后只有等待pause的功能，为了避免进程1栈中数据被破坏，进程0不可使用栈，那么就不可使用call来调用fork和pause库函数，而应该用内联的方法，防止参数传递将栈搞乱。

修改这个问题的方法可以利用条件宏的预处理作用来进行。在main.c开始包含#define _IN_MAIN，而unistd.h中的声明做如图1.7所示修改即可。

```

210 #ifndef _IN_MAIN
211 int fork(void);
212 int pause(void);
213 int sync(void);
214 #endif

```

图1.7

问题5：

Make后又发现如图1.8所示的错误。

```

fork.c: In function 'copy_mem':
fork.c:54:2: error: can't find a register in class 'DREG' while reloading 'asm'
fork.c:55:2: error: can't find a register in class 'DREG' while reloading 'asm'
fork.c:46:18: error: 'asm' operand has impossible constraints
fork.c:47:18: error: 'asm' operand has impossible constraints
fork.c:54:2: error: 'asm' operand has impossible constraints
fork.c:55:2: error: 'asm' operand has impossible constraints
make[1]: *** [fork.o] Error 1
make[1]: Leaving directory `/mnt/share/linux-0.11/linux-0.11/kernel'

```


图1.8

这是编译器对嵌入式汇编已用过的寄存器声明无效的报错信息。查看源文件发现问题出现在 `_set_base` 宏这里。

在 `include/linux/sched.h` 中，有如下语句：

```
188 #define _set_base(addr,base) \
189 __asm__ ("movw %%dx,%0\n\t" \
190         "rorl $16,%%edx\n\t" \
191         "movb %%dl,%1\n\t" \
192         "movb %%dh,%2" \
193         :: "m" (*((addr)+2)), \
194         "m" (*((addr)+4)), \
195         "m" (*((addr)+7)), \
196         "d" (base) \
197         : "dx") //无效声明
```

去掉197行中的 `:"dx"` 这项。也即197行变成了：

```
197 )
```

然后去掉所有汇编中的类似的地方：`String.h`, `memory.c`, `buffer.c`, `super.c`（原因是 `Bitmap.c`）。

这是由于 `as` 不断改进，目前自动化程度越来越高，不用人工指定一个变量需要的寄存器了，所以代码中的 `__asm__("ax")` 需要全部去掉。如 `bitmap.c` 和 `namei.c`。

问题6：

Make后又发现如图1.9所示的错误。

```
../include/asm/segment.h: Assembler messages:
../include/asm/segment.h:27: Error: bad register name '%bpl'
make[1]: *** [file_dev.o] Error 1
make[1]: Leaving directory `/mnt/share/linux-0.11/linux-0.11/fs'
make: *** [fs/fs.o] Error 2
```

图1.9

查看 `segment.h` 有如图1.10所示代码。

```
25 extern inline void put_fs_byte(char val, char *addr),
26 {
27     __asm__ ("movb %0,%%fs:%1"::"r" (val), "m" (*addr));
28 }
```

图1.10

将 `r` 改为 `q`，该问题就解决了。`r` 表示使用任意动态分配的寄存器，`q` 表示使用动态分配字节可寻址寄存器（`eax`, `ebx`, `ecx` 或 `edx`）。

问题7：

Make后又发现如图1.11所示的错误。

```
exec.c: In function 'copy_strings':
exec.c:139:44: error: lvalue required as left operand of assignment
exec.c: In function 'do_execve':
```

图1.11

这是个左值问题，查看 `exec.c` 源码如图1.12所示：

```

00138:         if (!(pag = (char *) page[p/PAGE_SIZE])) 00
00139:         ! (pag = (char *) page[p/PAGE_SIZE]) =|
00140:         (unsigned long *) get_free_page())
00141:         return 0;

```

图1.12

这个写法确实有些别扭，使用get_free_page()申请新页面，页面指针pag指向新页面，申请不到空闲页面则返回0。如图1.13修改便可。

```

j136:         if (from_kmem--<0)
j137:             set_fs(old_fs);
j138:         if (!(pag = (char *) page[p/PAGE_SIZE]))
j139:         {
j140:             page[p/PAGE_SIZE] = (unsigned long *) get_free_page();
j141:             pag = (char *) page[p/PAGE_SIZE];
j142:             if(!pag)
j143:             {return 0;}
j144:         }
j145:

```

图1.13

类似问题在malloc.c中也有。

问题8:

Make后又发现如图1.14所示的错误。

```

In file included from floppy.c:42:0:
blk.h:87:6: error: #elif with no expression
make[1]: *** [floppy.o] Error 1

```

图1.14

查看代码blk.h发现是#elif木有判断的逻辑表达式，改为#elif 1便可。

问题9:

Make后又发现如图1.15所示的错误。

```

../include/string.h: In function 'strtok':
../include/string.h:280:1: error: can't find a register in class 'SIREG' whi
le reloading 'asm'
../include/string.h:280:1: error: 'asm' operand has impossible constraints
make[1]: *** [string.o] Error 1
make[1]: Leaving directory `/mnt/share/linux-0.11/linux-0.11/lib'
make: *** [lib/lib.a] Error 2

```

图1.15

这个问题不懂怎么改，按照之前类似问题将第三个冒号后面的寄存器删掉也不行，**求指点**。这里我果断把所有汇编注释掉，因为strtok这个函数在内核中没有被调用。

至此，编译问题全部解决。

二、链接过程

编译通过了，但请别高兴太早，因为链接过程中遇到的问题一点也不少。

1、除错工具:

- (1) sourceInsight，源码阅读器肯定需要了，否则岂不是找崩溃吗？
- (2) internet，链接过程网上有不少人完成了，赵炯博士也完成了，对于一些问题可以查到相应资料。
- (3) editplus，这是个编辑器了，我用之多年，比较喜欢，其特点就是小但功能全。

2、问题及解决方法

问题1:

链接错误中，程序入口问题首当其冲。错误信息如图2.1所示。

```
ld -s -x -M boot/head.o init/main.o \
    kernel/kernel.o mm/mm.o fs/fs.o \
    kernel/blk_drv/blk_drv.a kernel/chr_drv/chr_drv.a \
    kernel/math/math.a \
    lib/lib.a \
    -o tools/system > System.map
ld: warning: cannot find entry symbol _start; defaulting to 00000000080480a
```

图2.1

这个链接问题的原因就是链接器在将一堆目标文件组合到一起的时候，不知道那个应该放在开头，也就是生成的可执行文件的第一条指令应该是哪个目标文件的第一条。根据内核知识可知，内核最先运行的是head程序，所以打开head.s在.text段中增加.globl start_32。然后在ld的参数选项中增加-m elf_i386 -Ttext 0 -e startup_32。

问题2:

之后又出现如图2.2所示的问题。

```
boot/head.o: In function `_pg_dir':
(.text+0x10): undefined reference to `_stack_start'
boot/head.o: In function `_pg_dir':
(.text+0x2e): undefined reference to `_stack_start'
boot/head.o: In function `after_page_tables':
(.text+0x540c): undefined reference to `_main'
boot/head.o: In function `ignore int':
```

图2.2

这个链接问题是由于找不到变量所致，原因是早期编译linux-0.11的汇编器需要在c变量之前增加下划线来识别，而目前的汇编器已足够强大，不用下划线了，所以要全部去掉，这是个枯燥的工程啊。

问题3:

然后又出现如图2.3所示的问题。

```
serial.c:(.text+0x60): undefined reference to `rs1_interrupt'
serial.c:(.text+0x77): undefined reference to `rs2_interrupt'
```

图2.3

这种问题是由于没有这两个串口中断处理程序，可以定义一个空程序，或者直接去掉初始化内容

问题4:

之后又出现如图2.4所示的问题。

```
root@choumao-VirtualBox:/mnt/share/linux-0.11/linux-0.11/kernel/chr_drv# m
e
gcc -E -nostdinc -I../include -traditional keyboard.S -o keyboard.s
cc1: fatal error: opening output file keyboard.s: Operation not permitted
compilation terminated.
make: *** [keyboard.s] Error 1
```

图2.4

这个问题比较有意思，对keyboard.S做一些处理后生成keyboard.s，但是gcc-4.6.1不区分文件名大小写，故而源文件和目标文件成了同一个。解决方法很简单，将源文件keyboard.S改名为keyboard1.S，同时修改其makefile便可。

问题5:

接着又出现如图2.5所示的问题。


```
kernel/kernel.o: In function `number':
vsprintf.c:(.text+0x173d): undefined reference to `__stack_chk_fail'
fs/fs.o: In function `do_execve':
(.text+0x3241): undefined reference to `__stack_chk_fail'
kernel/chr_drv/chr_drv.a(tty_ioctl.o): In function `tty_ioctl':
tty_ioctl.c:(.text+0x32e): undefined reference to `__stack_chk_fail'
make: *** [tools/system] Error 1
```

图2.5

在所有错误涉及到的目录下（这里是kernel/, fs/, kernel/chr_drv/）的Makefile里找到CFLAGS然后添加-fno-stack-protector标志！其实这是传给GCC的一个编译选项。-fno-stack-protector参数用来disable Stack-smashing protection,高版本gcc默认用-fstack-protector参数进行编译。

问题6:

之后又出现如图2.6所示的问题。

```
gcc -Wall -O -fstrength-reduce -fomit-frame-pointer -fno-stack-protector \
-o tools/build tools/build.c
tools/build.c: In function `main':
tools/build.c:72:4: warning: implicit declaration of function `MAJOR' [-Wimplicit-function-declaration]
tools/build.c:73:4: warning: implicit declaration of function `MINOR' [-Wimplicit-function-declaration]
/tmp/cchHa5oB.o: In function `main':
build.c:(.text+0xd9): undefined reference to `MAJOR'
build.c:(.text+0xf1): undefined reference to `MINOR'
collect2: ld returned 1 exit status
make: *** [tools/build] Error 1
```

图2.6

这个问题简单，就是build.c程序找不到MAJOR和MINOR的实现体，那么就打开build.c将这两个宏的实现加入便可，如图2.7所示。

```
00019: /*
00020:  * Changes by tytso to allow root device specification
00021:  */
00022:
00023: #include <stdio.h> /* fprintf */
00024: #include <string.h>
00025: #include <stdlib.h> /* contains exit */
00026: #include <sys/types.h> /* unistd.h needs this */
00027: #include <sys/stat.h>
00028: #include <linux/fs.h>
00029: #include <unistd.h> /* contains read/write */
00030: #include <fcntl.h>
00031:
00032: #define MINIX_HEADER 32
00033: #define GCC_HEADER 1024
00034:
00035: #define SYS_SIZE 0x2000
00036:
00037: #define DEFAULT_MAJOR_ROOT 3
00038: #define DEFAULT_MINOR_ROOT 6
00039: #define MAJOR(a) (((unsigned)(a))>>8)
00040: #define MINOR(a) ((a)&0xff)
00041: /* MAJOR and MINOR are used in setup, don't change unless you also change
00042:  * bootsect etc */
00043: #define SETUP_SECTS 4
00044:
```

图2.7

至此，make已经可以通过了。

三、调试过程

编译链接成功后，运行如图3.2所示。大多数linux-0.11的玩家都停到这里了，但是咱不是得做小部分人嘛，那继续吧。

1、调试方法与工具：

（1）插桩（printf、while(1)()等手段）。注意 Printk在内核态使用，比如中断处理函数，系统调用处理函数等，而printf在用户态使用，比如init进程体里面。

（2）bochs单步（s、n）、断点（b、blist、del、c）、内存检测（x、watch）等调试手段。

(3) make时自动生成的可执行文件链接图system.map，如图3.1所示，LDFLAGS中的-M选项实现此功能。这个文件在调试内核时相当重要，它展示了每个全局变量、全局可见的函数的地址，该文件配合bochs的断点和内存查看、监控命令，几乎无所不能。

(4) objdump生成的反汇编文件（如图3.1红线标注部分所示修改makefile，然后make DISASM便可生成该文件）调试时可以用来确认汇编码所属的函数，也可用来阅读汇编码。

```
AS      =as
LD      =ld
DISASM  =objdump
LDFLAGS =-s -x -M -m elf_i386 -Ttext 0 -e startup_32
#LDFLAGS      =-g -M -m elf_i386
CC      =gcc $(RAMDISK)
#CFLAGS =-Wall -g -fstrength-reduce -fomit-frame-pointer -fno-stack-protector \
CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fno-stack-protector \
#-fcombine-regs -mstring-insns
DISASMFLAGS= -D
CPP      =cpp -nostdinc -Iinclude
```

图3.1

- (5) 赵炯博士oldlinux网站上下载的linux-0.11-devel-040923包，可获得用gcc1.4编译后可正常运行linux-0.11。调试时我们用gcc-4.6.1编译的内核运行情况和正确的对比，有确认错误的功能。
- (6) hexdump -C rootimage-0.11 |less命令可用来查看文件系统数据，并确认其正确性，调试过程涉及到execve系统调用后，每次运行系统都需重新更换文件系统软盘镜像，因为上次运行可能修改了其中的数据。

2、问题及解决方法

问题1:

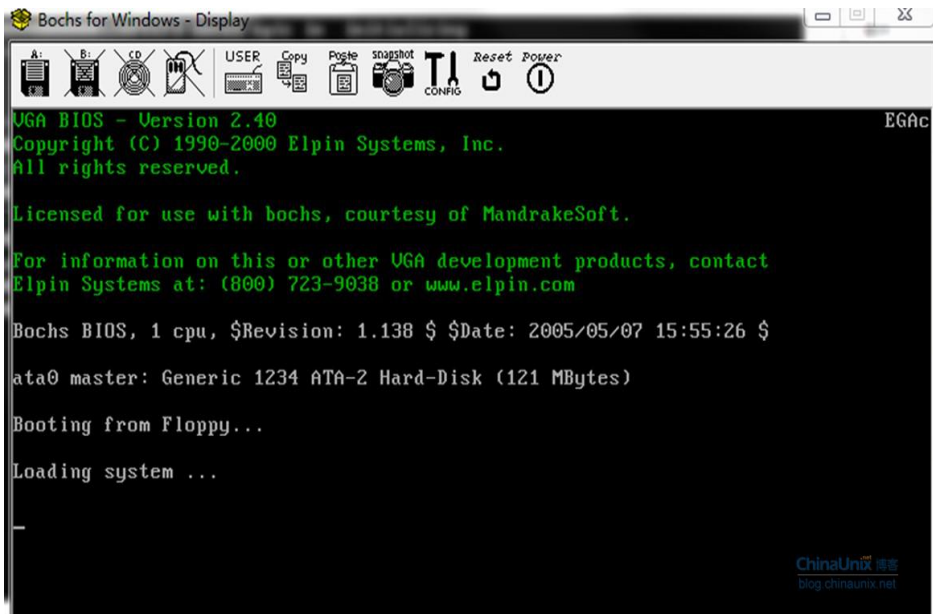


图3.2

首先需要确定的是图3.2所示的运行到底停在了那里。内核刚开始正式运行时进入了main函数，在这里进入第0个进程后，创建出第一个进程运行init函数，如图3.3所示。插桩输出是最普通也常常是最有效的调试方法。在init函数中使用printf发现并没有输出，说明系统没有进入第一个进程，那么fork系统调用（功能：创建新进程）就有问题。使用bochs调试功能查看main函数汇编源码，发现调用fork时使用了call指令而非应有的内联。


```

137     sti();
138     move_to_user_mode();
139     if (!fork()) {          /* we count on this going ok */
140         init();
141     }

```

ChinaUnix 博客
blog.chinaunix.net

图3.3

linux-0.11的进程0和1共用一个栈，进程0在创建了进程1后只有等待pause的功能，为了避免进程1栈中数据被破坏，进程0不可使用栈，那么就不可使用call来调用fork和pause库函数，而应该用内联的方法，防止参数传递将栈搞乱。为了实现这一点，在main.c开始为几个系统调用出了内联声明，如图3.4所示，static inline表示仅仅在该文件中这些系统调用时内联的。

```

static inline _syscall0(int, fork)
static inline _syscall0(int, pause)
static inline _syscall1(int, setup, void *, BIOS)
static inline _syscall0(int, sync)

```

ChinaUnix 博客
blog.chinaunix.net

图3.4

然而通过调试可知，该内联功能并未实现（具体原因不详，求高人指点），以至于进程1栈混乱。但仍可以做如图3.5所示修改使之运行通过，其实就是直接将这两个库函数的实现拷入main.c中。

```

146     move_to_user_mode();
147
148     long __res, __rs;
149     __asm__ volatile ("int $0x80" \
150     : "=a" (__res) \
151     : "0" (__NR_fork));
152     //if (__res >= 0)
153     //    return (int) __res;
154
155     if (__res == 0) {          /* we count on this going ok */
156         init();
157     }
158
159     /*
160     * NOTE!! For any other task 'pause()' would mean we have to get a
161     * signal to awaken, but task0 is the sole exception (see 'schedule()')
162     * as task 0 gets activated at every idle moment (when no other tasks
163     * can run). For task0 'pause()' just means we go check if some other
164     * task can run, and if not we return here.
165     */
166     for(;;)
167     {
168         __asm__ volatile ("int $0x80" \
169         : "=a" (__rs) \
170         : "0" (__NR_pause));
171         //pause();
172     }

```

ChinaUnix 博客
blog.chinaunix.net

图3.5

问题2:

经过上述修改后，编译运行发现系统运行状况依然如故。天呐，求你动一动吧...

说明fork系统调用还有其他问题。Bochs在fork软中断的处理函数copy_process中设断点，但不执行并使用x /nwx addr命令验证每一步的操作是否成功。发现*p=*current并没有将进程0的进程描述体赋值给进程p地址指向的内存，而是赋给了以p地址为末尾、以sizeof(task_struct)为大小的一段内存。那么问题明显了，这句c语句一定是用了rep指令，df位为向下拷贝（之前某个地方使用了std指令）。而新编译器编译c代码时，会认为已经cld过了，因此没加修改df位的指令。所以在所有c语言汇编后使用了rep指令拷贝数据的地方（使用objdump生成的反汇编文件中，搜索rep），都加上cld内嵌汇

编。其实只有copy_process一个，其他用到rep的地方本来就是汇编，拷贝前都有cld或者std。做如图3.6所示修改后，编译运行如图3.7所示。

```
00081:      task[nr] = p;
00082:      asm volatile ("cld");
00083:      *p = *current; /* NOTE! this doesn't copy the sup
00084:      p->state = TASK_UNINTERRUPTIBLE;
00085:      p->pid = last_pid;
00086:      p->father = current->pid;
00087:      p->counter = p->priority;
00088:      p->signal = 0;
00089:      p->alarm = 0;
00090:      p->leader = 0; /* process leadership doesn't
```

图3.6

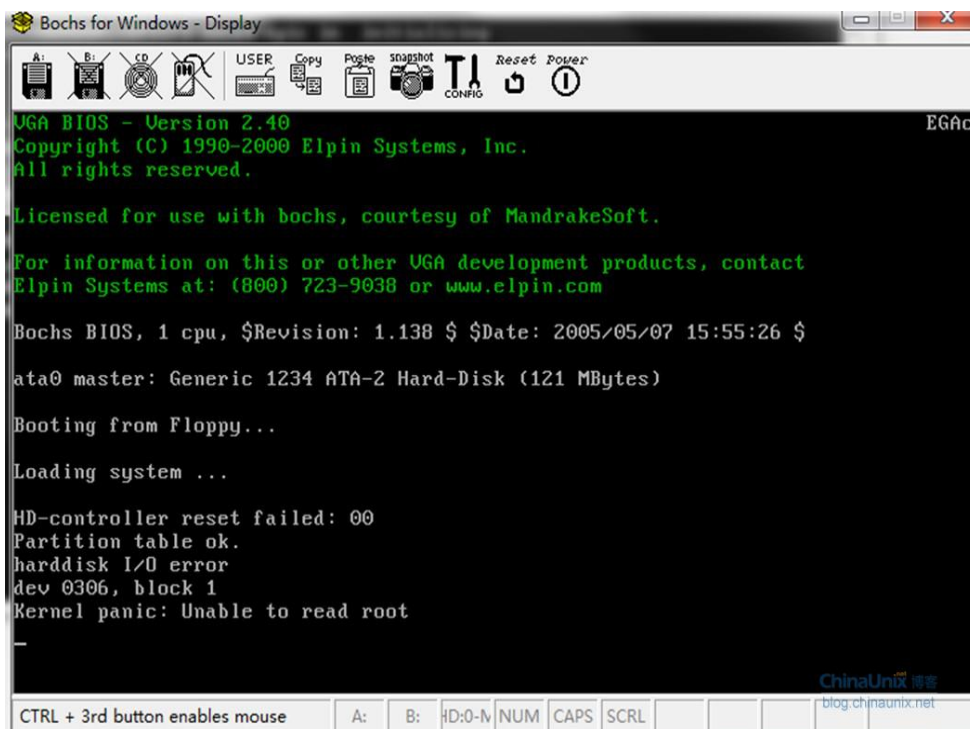


图3.7

问题3

这个问题提示信息比较直接一些，就是文件系统无法读取。那肯定是软盘或者硬盘根设备没有配置好了。为了简单，本文且使用软盘装着文件系统。需要配置以下几个地方。

(1) bochsrc-hd.bxrc中配置，使用软盘驱动器b来装载文件系统盘。

floppya: 1_44="Image", status=inserted #内核映像

floppyb: 1_44=rootimage-0.11, status=inserted #软盘镜像及其文件系统，该软盘从赵炯博士的oldlinux网站上下载的linux-0.11-devel-040923压缩包中可以找到，当然也可以自己制作。

(2) 修改Image的生成工具----build。在tool目录下打开build.c，做如图3.8所示修改，将设备号设置为021d。

```

00058: int main(int argc, char ** argv)
00059: {
00060:     int i, c, id;
00061:     char buf[1024];
00062:     char major_root, minor_root;
00063:     struct stat sb;
00064:
00065:     if ((argc != 4) && (argc != 5))
00066:         usage();
00067:     if (argc == 5) {
00068:         /*if (strcmp(argv[4], "FLOPPY")) { //相同返回0, 否则-1
00069:             if (stat(argv[4], &sb)) {
00070:                 perror(argv[4]);
00071:                 die("Couldn't stat root device.");
00072:             }
00073:             major_root = MAJOR(sb.st_rdev);
00074:             minor_root = MINOR(sb.st_rdev);
00075:         } else*/ {
00076:             //major_root = 0;
00077:             //minor_root = 0;
00078:             major_root = 0x02;
00079:             minor_root = 0x1d;
00080:         }
00081:     } else {
00082:         major_root = DEFAULT_MAJOR_ROOT;
00083:         minor_root = DEFAULT_MINOR_ROOT;
00084:         //major_root = 3;
00085:         //minor_root = 1;
00086:     }

```

ChinaUnix 博客
blog.chinaunix.net

图3.8

(3) 在引导程序bootsect.S中修改ROOT_DEV常量的值为根设备号，如图3.9所示。

```

00040:
00041: ! ROOT_DEV: 0x000 - same type of floppy as boot.
00042: !          0x301 - first partition on first drive etc
00043: ROOT_DEV = 0x021d

```

图3.9

内核中的ROOT_DEV不用修改或者复制，因为在系统初始化时（main函数开始），使用ORIG_ROOT_DEV初始化了该变量，而ORIG_ROOT_DEV指向bootsect中INITSEG（初始化段）中的内存单元，在引导程序中已被赋值成021d。

完成上述配置后，从linux-0.11-devel-040923中拷贝rootimage-0.11软盘文件系统到本编译的根目录下，make、运行，结果如图3.10所示。

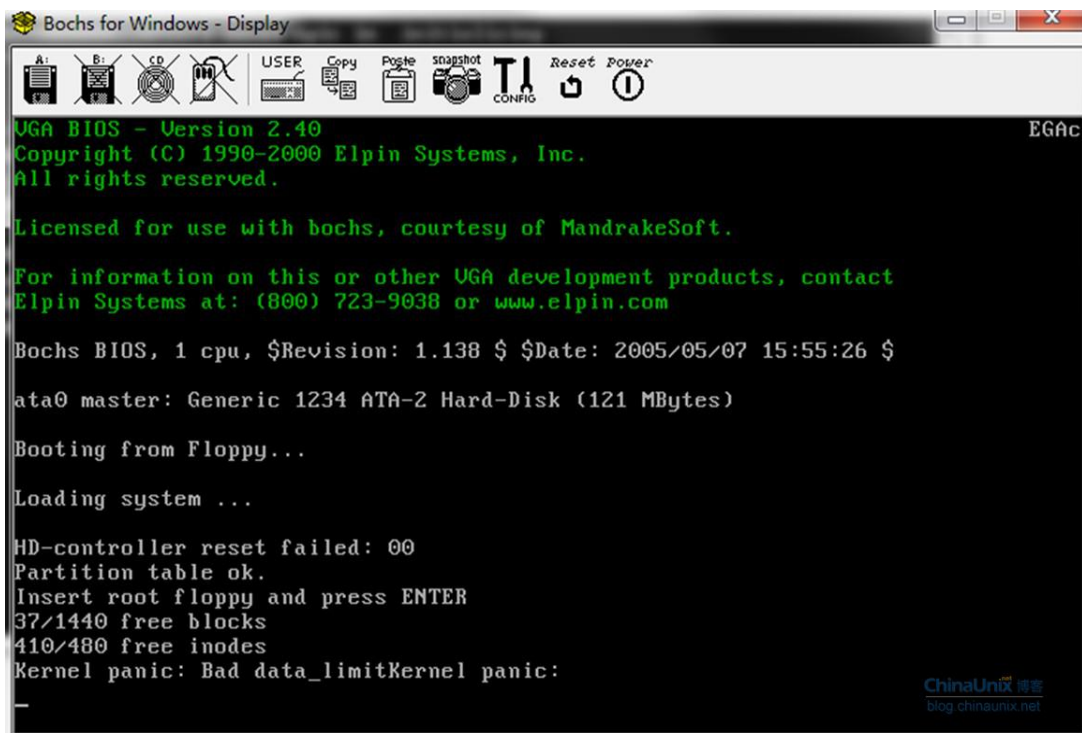


图3.10

问题4:

这个问题也比较清晰，在source insight中搜索Bad data_limit的引用便可以查到是在copy_mem中的出错信息，可见fork调用仍然创建进程失败。copy_mem()在fork.c中由copy_process()调用。如图3.11所示，copy_mem首先用get_base函数取进程0的代码段和数据段基址，检查是否相等（早期的linux二者是相等的），然后就出错了。

```
00039: int copy_mem(int nr, struct task_struct * p)
00040: {
00041:     unsigned long old_data_base, new_data_base, data_limit;
00042:     unsigned long old_code_base, new_code_base, code_limit;
00043:
00044:     code_limit = get_limit(0x0f);
00045:     data_limit = get_limit(0x17);
00046:     [old_code_base = get_base(current->ldt[1]);
00047:     old_data_base = get_base(current->ldt[2]);
00048:     if (old_data_base != old_code_base)
00049:         panic("We don't support separate I&D");
00050:     if (data_limit < code_limit)
00051:         panic("Bad data_limit");
00052:     new_data_base = new_code_base = nr * 0x4000000;
00053:     p->start_code = new_code_base;
00054:     //printfk("p' addr: %x, ldt: %x\n", p, &(p->ldt[1]));
00055:     set_base(p->ldt[1], new_code_base);
00056:     set_base(p->ldt[2], new_data_base);
00057:     if (copy_page_tables(old_data_base, new_data_base, data_limit)) {
00058:         free_page_tables(new_data_base, data_limit);
00059:         return -ENOMEM;
00060:     }
00061:     return 0;
00062: } ? end copy_mem ?
00063:
```

图3.11

Bochs调试跟踪并查看copy_mem函数汇编如图3.12所示。每个红色括号包括一个get_base内联函数的汇编码。显而易见，两个get_base调用的汇编语句数量居然不同，而且后者直接使用了前者生成的edx数据，对于两个独立的函数调用这当然是不允许的。后一个get_base必须重新给edx赋值。

```

00007bcd: <          >: inc edi                ; 47
00007bce: <          >: mov ecx, dword ptr ds:0x17140 ; 8b0d40710100
00007bd4: <          >: mov dh, byte ptr ds:[ecx+0x2df1] ; 8ab1df020000

00007bda: <          >: mov dl, byte ptr ds:[ecx+0x2dc1] ; 8a91dc020000

00007be0: <          >: shl edx, 0x10                ; c1e210
00007be3: <          >: mov dx, word ptr ds:[ecx+0x2da1] ; 668b91da0200
00          <
00007bea: <          >: mov eax, edx                ; 89d0
00007bec: <          >: mov dh, byte ptr ds:[ecx+0x2e71] ; 8ab1e7020000

00007bf2: <          >: mov dl, byte ptr ds:[ecx+0x2e41] ; 8a91e4020000

00007bf8: <          >: shl edx, 0x10                ; c1e210
00007bf9: <          >: mov dx, word ptr ds:[ecx+0x2e21] ; 668b91e20200
00          <
00007c02: <          >: mov ebp, edx                ; 89d5

```

图3.12

如图3.13所示，增加下划线标注的语句，每次调用先保存和恢复edx的值。

```

00218: #define _get_base(addr) ({
00219: unsigned long __base; \
00220: __asm__ ("pushl %%edx\n\t" \
00221:         "movb %3, %%dh\n\t" \
00222:         "movb %2, %%dl\n\t" \
00223:         "shll $16, %%edx\n\t" \
00224:         "movw %1, %%dx\n\t" \
00225:         "pop %%edx" \
00226:         : "=d" (__base) \
00227:         : "m" (*(addr)+2), \
00228:         "m" (*(addr)+4), \
00229:         "m" (*(addr)+7)) \
00230: __base; })

```

图3.13

问题5

十分遗憾，这个问题的运行图没留下来，因为那几天太忙了，总想赶快调完，结果欲速却不达。好了，进行上述修改后，init进程可以运行起来了，但却显示打不开文件，查看init代码可知，现在无法打开/dev/tty0。到底open系统调用哪里出现了问题呢？先看看open系统调用的函数调用关系：

sys_open->open_namei->dir_namei->get_dir。使用bochs调试功能一步一步进行跟踪，发现get_dir中使用get_fs_byte无法获取路径字符串的值。

get_fs_byte的作用是使用fs中的值作为数据段选择子来访问内存地址一个字节数据。进程1的局部变量“/dev/tty0”的首地址addr一定保存在该进程的栈里，若在进程1中访问该变量应该是这种形式：ds:addr（ds就是数据段的意思，根据ds可得到数据段选择子，从而得到数据段基地址，加上偏移地址addr便是线性地址，linux-0.11内存完全分页，故而已0为基地址的数据段其局部变量的线性地址=物理地址）。在进程内部ds、fs的值都是该进程数据段的首地址，而若在内核态访问该进程的局部变量就不应该使用ds（包含内核态数据段选择子）了，而应该用：fs: addr（fs指向的数据段的基地址+addr就是线性地址），因为linux-0.11为了能在内核态访问用户态的数据，由用户态在进入内核态时没有修改fs的值，即fs仍然包含用户态某一进程的数据段选择子。

因此要特别注意不能在get_dir中直接使用printk输出pathname来验证函数传参的正确性，因为printk访问pathname肯定是以ds作为数据段的，而pathname指向的变量在fs（用户态）数据段中。而且

这里还有一个特别迷惑性的状况：使用printf输出"/dev/tty0"字符串是成功的，让人误以为这里没问题，其实大错特错，因为这只是一种特殊的情况。由于编译器在编译时，把所有它能看见的代码放到一起，并以0为基地址为每个函数局部变量分配地址addr。编译器不知道init函数是一个数据段基地址非0的进程体，因此不论在内核态还是用户态，以0为数据段基地址，addr为偏移地址肯定能访问得到想要的局部变量。因此这里get_dir函数虽然在内核态，还是可以通过ds:addr访问得到pathname的值。而进程1的数据段基地址是0x4000000（每个进程64M虚拟空间），在fork创建的时候共享了父进程的页表，故而0x4000000+addr同样可以访问该字符串的值，而且这种方式才是正确的方式，get_fs_byte使用的就是这种方式。总之，这里无论是用0+addr（ds:addr）还是0x4000000+addr（fs:addr）都应该可以访问到该字符串。如果进程的代码不是编译器可见的，比如execeve之后从文件系统加载的shell可执行文件，那么它的局部变量就不能从以0为基地址的数据段访问（编译器未为其局部变量分配空间），只能用该进程数据段的基地址加偏移来访问了，这时，内核态就无法使用printf输出pathname了。这一点曾经差点纠结死我。

还要注意，在linux-0.11中倘若进程的数据段以0为基地址，那么其局部变量的线性地址==物理地址，因为linux分页管理是从0开始完全分页的（学习内存管理时，理解这一点很重要）。

根据get_dir中使用get_fs_byte无法获取路径字符串的值这一现象，我用x命令查看了fs选择子所指向的数据段描述符，从中提取的基地址却不是0x4000000，那么这就是原因所在了。一定是fork创建进程1时，在copy_mem()中没能给新进程设置正确的数据段基地址，查看copy_mem()汇编码如图3.14所示，红色括号标注的是该函数连续两次调用的set_base函数，为数据段和代码段设置基地址。很明显先后两次调用的函数体汇编码数量却不一样，而且第二次调用直接使用了第一次使用的edx数值，跟问题4一样，这是不允许的。第二次调用之前一定要恢复edx第一次调用修改之前的数值。

```

00007a1b: <                                >: mov edx, esi                ; 89f2
00007a1d: <                                >: mov word ptr ds:[ebx+0x2da], dx ; 668993da0200
00                                >
00007a24: <                                >: ror edx, 0x10                ; c1ca10
00007a27: <                                >: mov byte ptr ds:[ebx+0x2dc], dl ; 8893dc020000
00                                >
00007a2d: <                                >: mov byte ptr ds:[ebx+0x2df], dh ; 88b3df020000
00                                >
00007a33: <                                >: mov word ptr ds:[ebx+0x2e1], dx ; 668993e20200
00                                >
00007a3a: <                                >: ror edx, 0x10                ; c1ca10
00007a3d: <                                >: mov byte ptr ds:[ebx+0x2e4], dl ; 8893e4020000
00                                >
00007a43: <                                >: mov byte ptr ds:[ebx+0x2e7], dh ; 88b3e7020000

```

图3.14

因此增加如图3.15所标注的指令。edx之前循环右移16位，函数功能完成后，再循环右移16位，恢复原值。

```

188 #define _set_base(addr,base) \
189 _asm_ ("movw %%dx,%0\n\t" \
190 "rorl $16,%%edx\n\t" \
191 "movb %%dl,%1\n\t" \
192 "movb %%dh,%2\n\t" \
193 "rorl $16,%%edx" \
194 :: "m" (*(addr)+2), \
195 "m" (*(addr)+4), \
196 "m" (*(addr)+7), \
197 "d" (base) \
198 : )

```

图3.15

编译，运行后，情况如图3.16所示，新问题出现了。

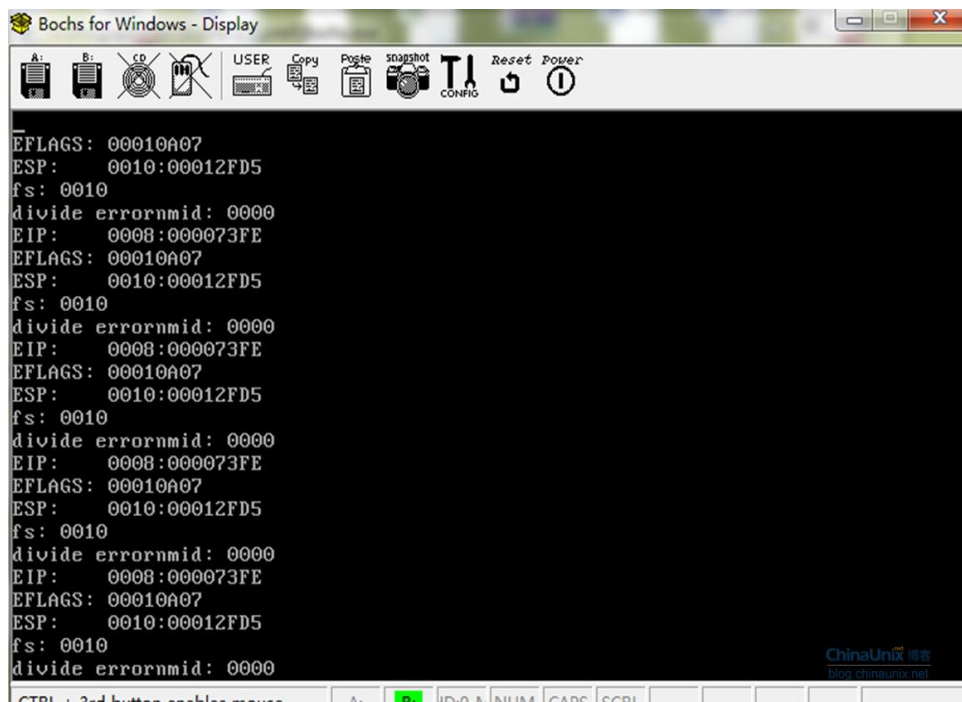


图3.16

问题6:

这个问题提示信息很不明确，调试也相对麻烦一些。

根据单步、断点等一顿乱调，最后确定出错的位置是do_execve()中两个free_page_table调用出现了问题，源码如图3.17所示。（请容我稍微complaint一下：看看do_execve那近两百行的c代码，想想它可能编译出的汇编码数量，你也许可以image一下我定位这个错误费了多大周章，因为这种调试最悲催的一点就是即使用插桩得到了大致位置，也得从函数头单步走过去详细查看，除非你愿意花大量时间将汇编和c代码一一对应起来...）。

当我们单步运行第一个free_page_table时，发现仍旧是get_base()函数出了问题。

```
00340:         current->reuse_val_base = 0;
00341:         free_page_tables(get_base(current->ldt[1]), get_limit(0x0f));
00342:         free_page_tables(get_base(current->ldt[2]), get_limit(0x17));
00343:         if (last_task_used_math == current)
```

图3.17

分析eflags寄存器的值，其溢出位为1，即发生了溢出错误。（注意：linux-0.11被新编译器编译后，错误提示并不正常，提示是divide异常，但eflags确实溢出错误。这些错误我们姑且放任，先让其能正常运行起来）。

经过单步调试后发现，还是在上述free_page_tables调用的get_base内部发生了溢出。查看溢出时的处理器信息如图3.18所示，edx左移16位时，eflags的值由0x207变成0xa07，edx溢出了，说明左移时edx高16位不是0。那么在get_base函数开始处增加一句and \$0x0,%%edx，以防止高位溢出便可。

```

text at 00000000:
(0) [0x0000c8c0] 0008:0000c8c0 (unk. ctxt): mov dx, word ptr ds:[edx+0x2da] ; 668b92da020000
(bochs:10) dump_cpu
eax:0x000a0000, ebx:0x00000014, ecx:0x000003d5, edx:0x08000000
ebp:0x00ffbf2c, esp:0x00ffba9c, esi:0x00041000, edi:0x0001c748
eip:0x0000c8c0, eflags:0x00000a07, inhibit_mask:0
cs:s=0x0008, dl=0x0000ffff, dh=0x00c09a00, valid=1
ss:s=0x0010, dl=0x0000ffff, dh=0x00c09300, valid=7
ds:s=0x0010, dl=0x0000ffff, dh=0x00c09300, valid=7
es:s=0x0010, dl=0x0000ffff, dh=0x00c09300, valid=7
fs:s=0x0017, dl=0x0000009f, dh=0x08c0f300, valid=1
gs:s=0x0017, dl=0x0000009f, dh=0x08c0f300, valid=1
ldtr:s=0x0048, dl=0xb2d00068, dh=0x000082ff, valid=1
tr:s=0x0040, dl=0xb2e80068, dh=0x000089ff, valid=1
gdt:base=0x00005cb8, limit=0x7ff
ldtr:base=0x000054b8, limit=0x7ff
tr0:0x00000000, dr1:0x00000000, dr2:0x00000000
tr3:0x00000000, dr6:0xfffff0ff0, dr7:0x00000400
cr0:0x8000001b, cr1:0x00000000, cr2:0x0801aa40
cr3:0x00000000, cr4:0x00000000
none
(bochs:11)

```

图3.18

编译运行后发现此处仍有错误，再次查看get_base()汇编码如图3.19所示，特别注意标注红色下划线的语句，这些语句拷贝的源地址和目的地址使用了相同的寄存器edx，以edx为基地址取值赋给dh，这时候edx的值一定改变了；接下来又用edx为基地址取值赋给dl时，取到的值肯定不是应该的那个。晕，编译器脑残了吗？

与图3.12对比，发现新编译器对这种老代码的编译并不稳定，之前编译的目的寄存器是ecx，现在却是edx，这中间我做什么了吗？**好吧，求指点。**

```

0000c8c4: <
>: mov dh, byte ptr ds:[edx+0x2df] ; 8ab2df020000
0000c8ca: <
>: mov dl, byte ptr ds:[edx+0x2dc] ; 8a92dc020000
0000c8d0: <
>: shl edx, 0x10 ; c1e210
0000c8d3: <
>: mov dx, word ptr ds:[edx+0x2da] ; 668b92da020000
0000c8da: <
>: mov dword ptr ss:[esp+0x4], eax ; 89442404
0000c8de: <
>: mov dword ptr ss:[esp], edx ; 89142404
0000c8e1: <
>: call .+0x9182 ; e89cc8ffff

```

图3.19

将get_base代码由图3.13修改为如图3.20所示，更换目的寄存器为ecx便可区分了。

```

00218: #define _get_base(addr) ({
00219: unsigned long __base; \
00220: __asm__ ("pushl %%ecx\n\t" \
00221: "andl $0x0, %%ecx\n\t" \
00222: "movb %3, %%ch\n\t" \
00223: "movb %2, %%cl\n\t" \
00224: "shll $16, %%ecx\n\t" \
00225: "movw %1, %%cx\n\t" \
00226: "pop %%ecx" \
00227: : "=c" (__base) \
00228: : "m" (*((addr)+2)), \
00229: "m" (*((addr)+4)), \
00230: "m" (*((addr)+7)); \
00231: __base; })
00232:

```

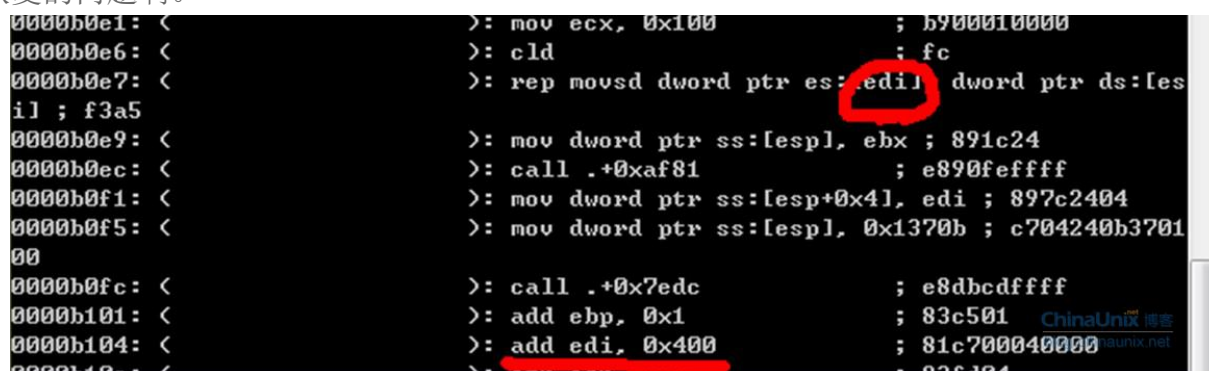
图3.20

编译运行后，execve系统调用终于通过了。

问题7:

虽然execve成功了，但是加载的shell程序却无法正常运行。具体的错误界面没能留下，十分后悔当初记录不仔细。但shell程序没能正常运行是明确的。这个问题就更加难调试了，因为在执行了exeve之后，首先要进行需求加载，发生79(大约是这个数字)次do_no_page缺页中断从文件系统中加载shell文件，那错误就可能是中断处理时发生的，也可能是shell运行过程中发生的，而shell又是这个系统里面最大的程序，从中寻找一个错误何其困难。

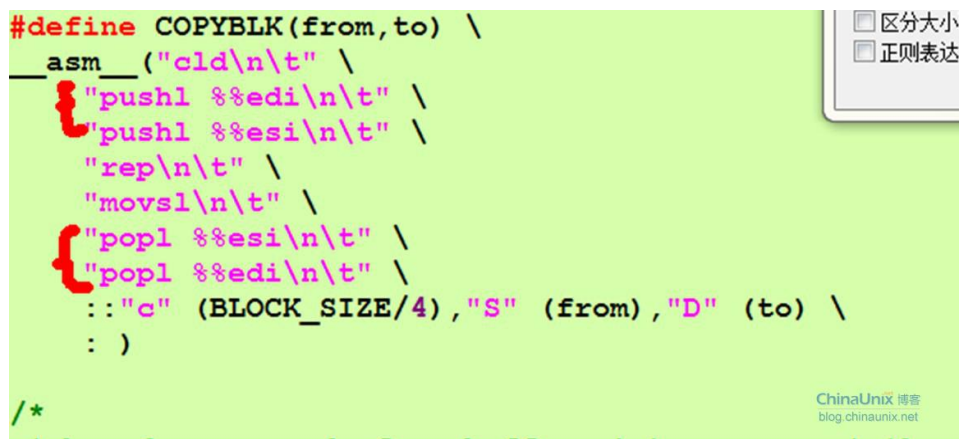
然而幸运的是，这个错误被我偶然间找到了。我在过滤缺页中断处理函数时，每次逻辑块拷贝（共四次）都查看了拷贝结果，发现shell的第一块数据被复制到申请的空闲页的0-0x3ff处，第二个块却被拷贝到0x800-0xbff处，依次类推，隔1024B复制一块数据，这样导致内存中的可执行文件是不连续的。查看COPYBLK函数，其汇编码如图3.21所示。思考一下可知，第一次调用COPYBLK()拷贝数据时，目的地址寄存器edi最后增加了0x400B，这个修改影响了下一次块拷贝。可见还是寄存器保存和恢复的问题啊。



```
0000b0e1: <      >: mov ecx, 0x100          ; b900010000
0000b0e6: <      >: cld                      ; fc
0000b0e7: <      >: rep movsd dword ptr es:edi, dword ptr ds:[esi] ; f3a5
0000b0e9: <      >: mov dword ptr ss:[esp], ebx ; 891c24
0000b0ec: <      >: call .+0xaf81             ; e890feffff
0000b0f1: <      >: mov dword ptr ss:[esp+0x4], edi ; 897c2404
0000b0f5: <      >: mov dword ptr ss:[esp], 0x1370b ; c704240b3701
0000b0fc: <      >: call .+0x7edc             ; e8dbcdffff
0000b101: <      >: add ebp, 0x1              ; 83c501
0000b104: <      >: add edi, 0x400B           ; 81c70004000B
0000b107: <      >: jmp .+0x7edc              ; e8dbcdffff
```

图3.21

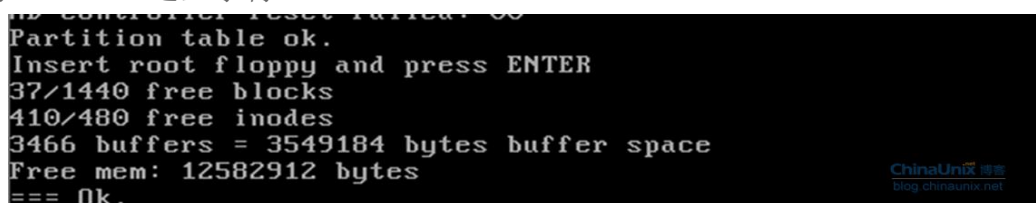
增加红色括号标注的语句后COPYBLK代码如图3.22所示。



```
#define COPYBLK(from,to) \
asm_ ("cld\n\t" \
{"pushl %%edi\n\t" \
"pushl %%esi\n\t" \
"rep\n\t" \
"movsl\n\t" \
{"popl %%esi\n\t" \
"popl %%edi\n\t" \
::"c" (BLOCK_SIZE/4), "S" (from), "D" (to) \
: )
/*
```

图3.22

从这个问题的调试过程也说明问题越是复杂时候，细心越是重要。编译运行后结果如图3.23所示，打印出了“==Ok.”这些字符。



```
Partition table ok.
Insert root floppy and press ENTER
37/1440 free blocks
410/480 free inodes
3466 buffers = 3549184 bytes buffer space
Free mem: 12582912 bytes
== Ok.
```

图3.23

阅读init函数源码可知，该段功能是用shell执行/etc/rc脚本程序，打开使用hexdump |C rootimage-0.11|less命令打开rc文件，这里需要好好研究下minix文件系统（可参考赵博士的linux内核完全剖析）才能用找到具体的文件。如图3.24所示。可见运行结果正确。

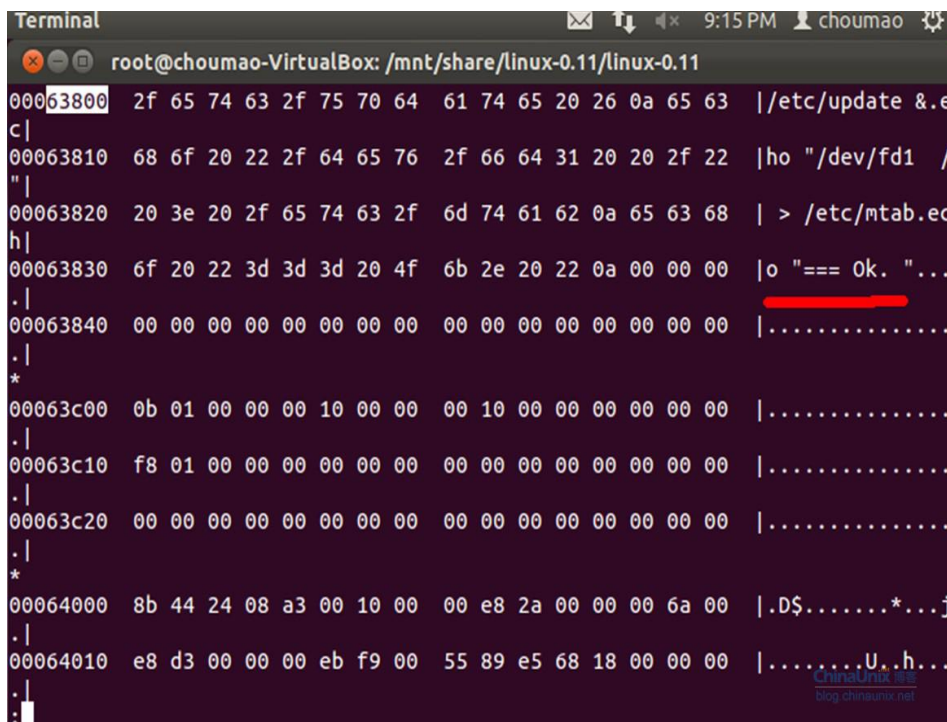


图3.24

其实在这个问题解决后，如果我当时足够冷静，没有其他事情烦扰，应该会首先完成一件事情再继续下去。这件事情就是：为所有汇编写的函数增加寄存器的保存和恢复操作。然而，那几天我太忙了，忙的忘记了思考，于是发生了问题8这个惨剧，是的，对我来说真是个惨剧。

问题8:

上述美丽的输出结果下面却是如图3.25的错误提示信息。革命尚未成功，同志仍需努力啊。错误性质倒是比较明确，就是上述sh程序运行结束后，系统又创建了一个shell进程，用于和用户进行交互，这个进程无法打开某个文件，原因是进程结构体中的pwd元素指向的当前目录inode找不到了，还是有点诡异的。这个shell结束后，再创建下一个，如此循环往复。这个过程就比较熟悉了，就是平时使用linux时的tty里面的命令行嘛。因此凭我超常的男人直觉认为，这是最后一个问题了。

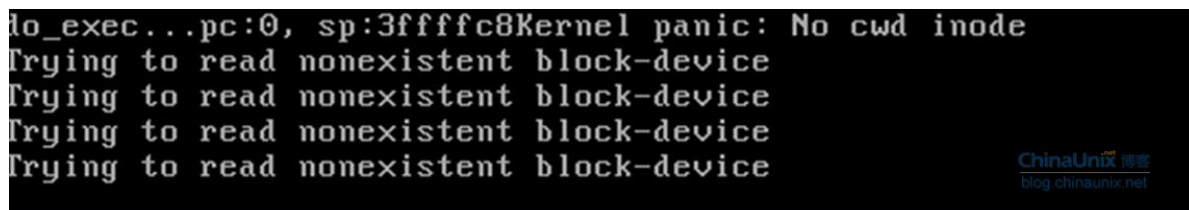


图3.25

然而黎明前的黑暗真黑啊，这个问题也是最难调试的问题，原因是出现错误的程序不在内核源码中，而是从文件系统加载的sh可执行文件，如果分析bash源码可以参考

http://www.linjian.cn/files/articles/bash_study/bash_linjian.html，但是比较费时耗力。因为sh可执行文件是不会出错的，错误肯定发生在被高级编译器编译后的内核程序中，所以可以先偷个懒。经过插桩调试，发现出错的是main.c中init函数第二个execve系统调用，如图3.26所示。该系统调用加载/bin/sh文件并运行它。使用souce insight的查找引用功能，寻找“No cwd inode”的引用，发现该语句出现

在Namei.c的get_dir()函数中，而get_dir()却是个静态函数，在system.map中不可见，还会被优化编译到sys_mkdir()、sys_open()、Namei()等多函数中。这下彻底迷失了。

```
while (1) {
    if ((pid=fork())<0) {
        printf("Fork failed in init\r\n");
        continue;
    }
    if (!pid) {
        close(0);close(1);close(2);
        setsid();
        (void) open("/dev/tty0",O_RDWR,0);
        (void) dup(0);
        (void) dup(0);
        _exit(execve("/bin/sh", argv, envp));
    }
    while (1)
        if (pid == wait(&i))
            break;
    printf("\n\rchild %d died with code %04x\r\n", pid, i);
    sync();
}
```

图3.26

只好一步一步来了。在系统运行进入第三个do_execve函数后，在Namei()处设置断点，到出错会经过两三个Namei()，跟踪最后一个，单步执行发现get_dir()如图3.27所示的判断没通过，当前进程root和pwd指向的i节点的引用数变为0了。这就相当诡异了，到底是调用什么函数更改了当前进程的root目录或pwd目录的i节点（此刻二者指向同一个inode）呢？

```
if (!current->root || !current->root->i_count)
    panic("No root inode");
if (!current->pwd || !current->pwd->i_count)
    panic("No cwd inode");
```

图3.27

为了跟踪出错的位置，必须找到修改根目录inode（通过插桩输出得知，其内存地址为0x1c550，其i_count地址为0x1c580）的函数，然而通过bochs断点运行发现，sh程序在运行时，发生了79次缺页异常，即触发了79次需求加载功能（为了加快创建进程的速度，改善用户体验，linux不是一下子把可执行文件从文件系统中拷贝到内存中，而是拷贝一页就运行该页的指令，运行完后触发缺页异常，进行下一页的拷贝，这样用户就不会有等待的感觉），通过反复测试得知，在第20页0x1c580中的值由8改为1和第21页由1改为0，从而直接导致Namei无法找到根目录。在第20页，使用bochs的watch write 0x1c580命令，发现系统在执行到0xa85e和0xa964之间的一条指令时修改了该值，根据system.map文件可知该区域是get_empty_inode()函数。而21页修改该值的是iput()，该函数释放一个inode会将其引用数减1，功能上没有问题。

好了，经过九牛二虎之力把问题找到了，get_empty_inode()就是罪魁祸首。阅读其源码并单步跟踪其执行过程，该函数遍历inode_table（首地址为1c4e0），寻找一个空闲的inode，然后将其memset 0。如图3.28所示，发现一个运行bug，红色圆圈处原来的语句是break，找到空闲并且没上锁的inode后，本应该跳出while循环，而源码仅仅跳出for循环，之后还得等待解锁，显然是不合理的，应该改为goto语句跳出来。

```

struct m_inode * get_empty_inode(void)
{
    struct m_inode * inode;
    static struct m_inode * last_inode = inode_table;
    int i;

    do {
        inode = NULL;
        for (i = NR_INODE; i ; i--) {
            if (++last_inode >= inode_table + NR_INODE)
                last_inode = inode_table;
            if (!last_inode->i_count) {
                inode = last_inode;
                if (!inode->i_dirt && !inode->i_lock)
                {
                    goto HERE;
                }
            }
        }
        if (!inode) {
            for (i=0 ; i<NR_INODE ; i++)
                printk("04x: %6d\t", inode_table[i].i_dev,
                    inode_table[i].i_num);
            panic("No free inodes in mem");
        }
        wait_on_inode(inode);
        while (inode->i_dirt) {
            write_inode(inode);
            wait_on_inode(inode);
        }
    } ? end do ? while (inode->i_count);

HERE:
    memset(inode, 0, sizeof(*inode));
    inode->i_count = 1;
    return inode;
} ? end get_empty_inode ?

```

ChinaUnix 博客
blog.chinaunix.net

图3.28

此外，跟踪memset汇编代码如图3.29所示，发现将1c518指向的inode清零后，edi变为1c550，这时memset函数已经结束了，接着get_empty_inode()就利用这个edi为1c518处的空闲inode引用置1，却错误地将1c550出的inode引用置1了，这样正好修改了进程根目录的引用数。

```

0000a8e1: <          >: mov ecx, 0x38          ; b938000000
0000a8e6: <          >: mov eax, 0x0             ; b800000000
0000a8eb: <          >: cld                     ; fc
0000a8ec: <          >: rep stosh byte ptr es:[edi], al ; cf3aa/nix 博客
0000a8ee: <          >: mov word ptr ds:[edi+0x30], 0x1 ; 66c747300100

```

图3.29

修改过程如图3.30所示，其实就是用栈保存和恢复edi寄存器而已。

```

extern inline void * memset(void * s, char c, int count)
{
    __asm__ ("pushl %%edi\n\t"
            "pushl %%eax\n\t"
            "pushl %%ecx\n\t"
            "cld\n\t"
            "rep\n\t"
            "stosh\n\t"
            "popl %%ecx\n\t"
            "popl %%eax\n\t"
            "popl %%edi\n\t"
            :: "a" (c), "D" (s), "c" (count)
            :);
    return s;
}

```

ChinaUnix 博客
blog.chinaunix.net

图3.30

建议将所有的汇编函数都加上push和pop语句保存和恢复函数中使用的寄存器（比如get_base函数），不然很可能以后还会出现其他错误。我猜想有两个可能的原因：早期gcc与现在gcc的不同，要么就是makefile中去掉的-fcombine-regs编译选项导致没有保存寄存器吧？[这里求高人指点。](#)

如图3.31所示，终于可以跑通了，庆贺一下.....

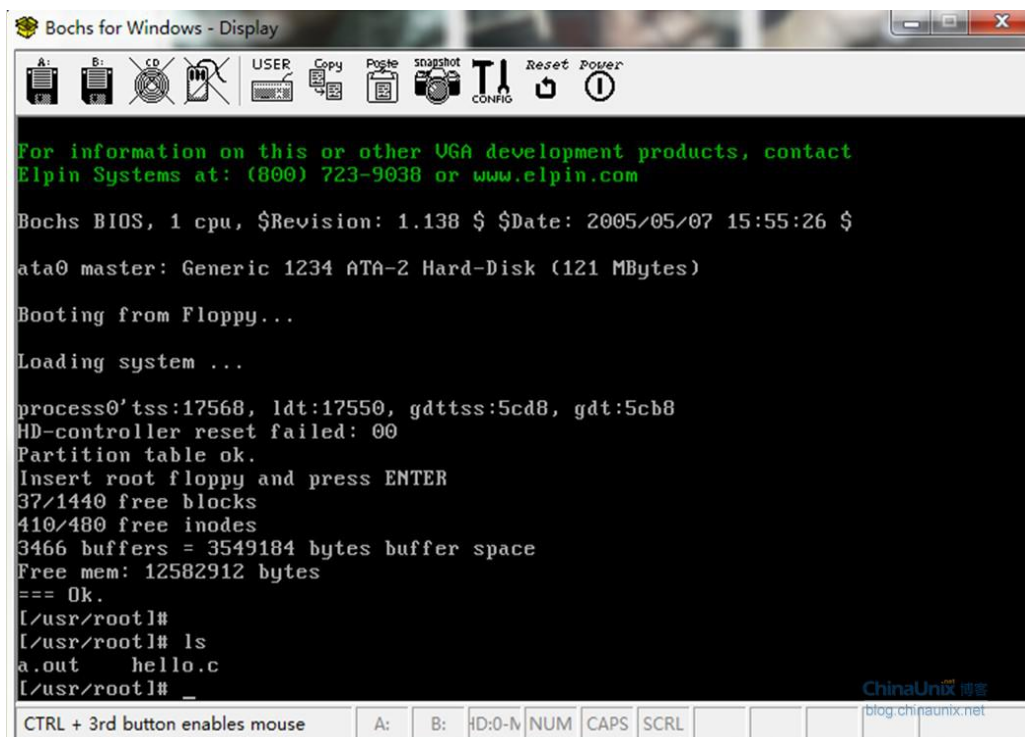


图3.31

四、总结及展望

花了近一个月的时间，终于把上述工作完成，并记录了下来。这项工作完成的过程中耳边一直萦绕着老师要论文的催促声，算是不太容易吧。其实总结一下，错误都出现在c和汇编合作的地方，多关注这些地方一定能省好多时间。本文开头已经说过，这个linux-0.11并未完全除尽其BUG，但是用上述方法一直做下去，一定能完成的。不过，我得先去干点别的事情了^_^。