Linux 运行前探秘之一
——内核编译及引导

徐 炜

摘 要：分析了 Linux 内核编译生成、引导的过程以及关键的源代码。

关键词：内核编译连接；引导；bootsect setup

1 内核编译流程

（1）编译内核时，在 linux/目录下执行 make bzImage，整个流程如下：

由于 linux/Makefile 中没有目标 bzImage，但有“include arch/$（ARCH）/Makefile”一句把 arch/i386/Makefile 包含进来了，因此 make bzImage 创建的是 linux/arch/i386/Makefile 中的 bzImage：

```
MAKEBOOT = $(MAKE) –C arch/$(ARCH)/boot
bzImage: vmlinux
	@$(MAKEBOOT) bzImage
```

它依赖于 vmlinux，因此首先创建 linux/Makefile 中的 vmlinux：

```
vmlinux: $(CONFIGURATION) init/main.o init/version.o linuxsubdirs
$(LD) $(LINKFLAGS) $(HEAD) init/main.o init/version.o \
		––start–group \
		$(CORE_FILES) \
		$(DRIVERS) \
		$(NETWORKS) \
		$(LIBS) \
		––end–group \
		–o vmlinux
	$(NM) vmlinux | grep –v '\(compiled\)\|\(\.o$$\)\|\( [aUw]\)\|\(\.\.ng$$\)\|\(LASH[RL]DI\)' | sort > System.map
```

其中：

```
LDFLAGS=–e stext
/* 以 linux/arch/i386/vmlinux.lds 作为连接器的脚本 */
LINKFLAGS =–T $ (TOPDIR)/arch/i386/vmlinux.lds $ (LDFLAGS)
CORE_FILES	=kernel/kernel.o mm/mm.o fs/fs.o ipc/ipc.o
NETWORKS	=net/network.o
DRIVERS		=drivers/block/block.o \
		drivers/char/char.o \
		drivers/misc/misc.o \
		drivers/net/net.o \
		drivers/media/media.o
LIBS		=$(TOPDIR)/lib/lib.a
```

以上生成了从 0xC0000000 + 0x100000 开始编址的 ELF 可执行文件 vmlinux。将其剥离 ELF 文件头后，就是真正的系统内核，引导解压合并结束后，内核会位于内存物理地址 0x100000 开始处。

（2）创建完 linux/vmlinux，回到 linux/arch/i386/Makefile 中，语句 @$（MAKEBOOT） bzImage 生成 bzImage，其中 MAKEBOOT = $（MAKE） –C arch/$（ARCH）/boot，就是说要创建 linux/arch/i386/boot/Makefile 中的 bzImage 目标，linux/arch/i386/boot/Makefile 中的语句如下：

```
bzImage: $ (CONFIGURE) bbootsect bsetup compressed/bvmlinux tools/build
	$ (OBJCOPY) compressed/bvmlinux compressed/bvmlinux.out
	tools/build –b bbootsect bsetup compressed/bvmlinux.out $(ROOT_DEV) > bzImage
```

于是转到 linux/arch/i386/boot 目录，分别生成 bbootsect，bsetup，compressed/bvmlinux，tools/build。然后将 compressed/bvmlinux 去掉 ELF 文件头后得到二进制文件 bvmlinux.out，然后通过 tools/build 将 bbootsect bsetup compressed/bvmlinux.out 合成 bzImage，即为可引导的内核。

（3）关键是 compressed/bvmlinux 的生成。linux/arch/i386/boot/Makefile 中的相关语句如下：

```
compressed/bvmlinux: $(TOPDIR)/vmlinux
	@$(MAKE) –C compressed bvmlinux
```

就是通过创建 linux/arch/i386/boot/compressed/Makefile 中的 bvmlinux 目标来生成 compressed/bvmlinux。分析 linux/arch/i386/boot/compressed/Makefile 中的语句可以得到以下步骤：

1）通过以下步骤生成压缩内核 piggy.o：

```
$(OBJCOPY) $(SYSTEM) $$tmppiggy /* 将 linux\vmlinux 的 ELF 文件头剥离，存为 linux/arch/i386/boot/compressed/$$tmppiggy，这就是真正的系统核心 */
gzip –f –9 < $$tmppiggy > $$tmppiggy.gz;  /* 用 gzip 压缩 $$tmppiggy 得到 $$tmppiggy.gz，这就是压缩的内核部分 */
/* 生成连接脚本 $$tmppiggy.lnk，将整个压缩内核作为数据段部分，input_data 指向起始处，input_len 指向压缩内核的长度，这两个符号都被 misc.s 中的解压函数用到了 */
echo "SECTIONS { .data : { input_len = .; LONG(input_data_end – input_data) input_data = .; * (.data) input_data_end
```

```
= .; }}" > $$tmppiggy.lnk;
$ (LD) –r –o piggy.o –b binary $$tmppiggy.gz –b elf32–i386 –T $$tmppiggy.lnk; /* 用连接器由压缩内核得到 ELF 目标文件 piggy.o */
```

2) 编译生成 head.o 和 misc.o：

```
head.o: head.S
	$(CC) $(AFLAGS) –traditional –c head.S
misc.o: misc.c
	$(CC) $(CFLAGS) –c misc.c
```

3) 将 head.o,misc.o,piggy.o 连接成 ELF 文件 bvmlinux，注意是也从 0x100000 开始编址的，因为引导时会被 setup.s 运送到内存 1M 开始处运行：

```
bvmlinux: piggy.o $(OBJECTS)
 $(LD) $(BZLINKFLAGS) –o bvmlinux $(OBJECTS) piggy.o
```

这是一个带 elf 文件头的自解压内核。其中：

```
ZLDFLAGS = –e startup_32	/*entry point: startup_32*/
BZIMAGE_OFFSET = 0x100000  /* 连接后从 0x100000 开始编址 */
BZLINKFLAGS = –Ttext $(BZIMAGE_OFFSET) $(ZLDFLAGS)
```

综上所述，bvmlinux 是从 0x100000，vmlinux 是从 3G+0x100000 开始编址的，因为 arch/i386/boot/compressed/head.o 和 arch/i386/kernel/head.o 都曾位于内存 0x100000，只不过前者只是在解压内核阶段，而后者自系统真正运行后就一直位于 0x100000。

（4）简要的说，生成 bzImage 的大致过程是：

1) 在 linux 目录下编译连接生成 linux\vmlinux。

2) 进入 linux/arch/i386/boot/compressed 目录,把 linux/vmlinux 去掉 ELF 头后存为$$tmppiggy，这就是真正的内核，常驻于内存 0x100000 开始处的系统核心。

3) 用 gzip 压缩 compressed/$$tmppiggy 得到 compressed/$$tmppiggy.gz，这就是压缩的内核部分。

4) 用连接器把 compressed/$$tmppiggy.gz 生成只含数据段的目标文件 compressed/piggy.o，这个数据段就是压缩的内核$$tmppiggy.gz。

5) 编译得到 compressed/head.o 和 compressed/misc.o。

6) 将 compressed/head.o,compressed/misc.o,compressed/piggy.o 连接成 ELF 文件 compressed/bvmlinux。

7) bvmlinux 剥离 ELF 文件头得到 compressed/bvmlinux.out。

8) 进入 linux/arch/i386/boot 目录，分别生成 bbootsect、bsetup，tools/build。

9) 用 tools/build 把 bbootsect,bsetup,compressed/bvmlinux.out 合为可引导的内核 bzImage （放在 linux/arch/i386/boot 中）。

用到了以下 Makefile 文件：

```
linux\Makefile –>linux\arch\i386\M akefile –>linux\arch\i386\bo
ot\Makefile–>linux/arch/i386/boot/compressed/Makefile
```

2 系统引导

2.1 流程描述

（1）bios 读入引导盘上的 bootsect 到 0x07c0:0x0000 处，然后 bootsect 把自己移到 0x9000:0x0000。

（2）bootsect 从磁盘读入 setup 的 4 个扇区到内存 0x9020:0000。

（3）bootsect 接着开始读入自解压内核 (压缩的)。如果是 _BIG_KERNEL_ 编译的内核 (bzImage)，读入到内存 0x100000 处 (使用 int 15h 的 87h 号功能，否则 (zImage) 内核被读入到 0x10000 (SYSSEG) 处。

（4）运行 0x9020:0000 处的 setup，获取相关系统参数，进行一定的初始化工作，设置 idt 和 gdt，开启保护模式，此时系统使用一个数据段和一个代码段，基址都是 0。如果自解压内核是 _BIG_KERNEL_ 的，从 0x10000 移到 0x1000 处，否则不移动;然后根据编译参数 (_BIG_KERNEL_) 跳转到 0x1000 或 0x100000 运行 boot/compressed/head 部分。

（5）compressed/head 检查 A20 地址开启后，设置堆栈，清空 bss 段，调用 compressed/misc.o 中的 decompress_kernel () 函数，对数据段 input_data 至 input_data+input_len 之间的压缩内核 (即上述的 compressed/piggy.o) 进行解压，解压后的真正的内核 vmlinux 仍然位于物理内存 0x100000 (虚拟内存 3G+0x100000) 处。

最后跳转到 0x100000 处执行，也就是 arch/i386/kernel/head.S 的代码。

注：zImage 或 bzImage 由 16 位引导代码和 32 位内核自解压映象两个部分组成。对于 zImage，内核自解压映象被加载到物理地址 0x1000，内核被解压到 1MB 的部位；对于 bzImage, 内核自解压映象被加载到 1M 开始的地方， 内核被解压为两个片段, 一个位于物理地址 0x2000–0x90000，另一个起始于内核自解压映象之后 0x3000 字节，并且离 0x100000 不小于低端片段最大长度的区域。解压完成后，这两个片段被合并到内存 0x100000 处。

2.2 关键代码分析

（1）引导扇区代码 linux/arch/i386/boot/bootsect.S：

1) 电脑启动时，bios 从磁盘读入引导扇区到 0x07c0:0000 处执行，也就是 bootsect.S 的代码。首先，它会将自身移动到 0x9000:0000 处执行。

```
SETUPSECS	= 4	/* default nr of setup–sectors */
BOOTSEG= 0x07C0	/* original address of boot–sector */
INITSEG = DEF_INITSEG /* we move boot here – out of the way */ 0x9000:0x0000
SETUPSEG = DEF_SETUPSEG /* setup starts here */ 0x9020:0x0000
SYSSEG = DEF_SY SSEG/* system loaded at 0x10000 (65536) */
```

```
SYSSIZE = DEF_SYSSIZE
/* 把启动扇区代码（即 bootsect）从 0x7c0:0000 移到 0x9000:
0000,然后用长转指令转移过去继续执行。*/
movw    $BOOTSEG, %ax
    movw    %ax, %ds        /* ds=0x07C0 */
    movw    $INITSEG, %ax
    movw    %ax, %es        /* es=0x9000 */
    movw    $256, %cx  /* 移动 256 个双字节=512Bytes */
    subw    %si, %si        /* si=0 */
    subw    %di, %di        /* di=0 */
    cld
    rep
    movsw    /*0x07C0:0000 处的 512 字节移动到 0x9000:
0000*/
    ljmp  $INITSEG, $go     /* 转到0x9000:go 处运行 */
```

2) 引导代码从磁盘载入引导扇区后的 4 个扇区的 setup.S 的目标代码：

```
load_setup:
    xorb  %ah, %ah              # ah=0,
    xorb  %dl, %dl     # 调用 0x13 中断 reset 软驱控制器
    int    $0x13
    xorw %dx, %dx           # 驱动器号 0, 磁头号 0
    movb    $0x02, %cl          # 第 2 扇区, 0 磁道
    movw    $0x0200, %bx  # 读入到 es:bx (0x9000:0200)
    movb    $0x02, %ah          # ah=0x02,读扇区
    movb    setup_sects, %al    # al=读入扇区数(set-
up_sects 是 4)
    int    $0x13                # 将软盘从第二扇区开始的 4
# 个扇区(setup 代码)读入内存(0x9000:0x0200)处。
    jnc    ok_load_setup        # 成功则转到 ok_load_setup
# 继续执行
# 否则打印错误码并循环重试
    pushw    %ax
   call    print_nl
    movw    %sp, %bp
    call   print_hex
    popw    %ax
    jmp   load_setup   /* 不成功则转到 load_setup 处循环重
试 */
```

3) 读入压缩内核（自解压的）：

```
# 以下程序(read_it)把核心装载到内存 0x10000 处,并保证
# 64KB 对齐。尽可能快的读入内核,只要可能一次就读入整个
# 磁道
sread:    .word 0                   # sectors read of cur-
rent track  ;sread 中存放当前磁道已读扇区数
head:     .word 0                   # current head   0 磁头
track:    .word 0                   # current track  0 磁道
# 读循环 rp_read 中的参数 ax=读扇区数,bx=段内偏移,head=
# 磁头号,track=磁道号,sread=当前磁道已读扇区数
read_it:    # 读内核程序的起点
 movb    setup_sects, %al   # setup_sects=SETUPSECS=4
    incb  %al                 # 加上 1 个 bootsect 扇区
    movb    %al, sread          # 当前已读入 5 个扇区,
# 把这个值存入 sread
    movw    %es, %ax
    testw    $0x0fff, %ax    # es 的值必须是 64KB 对齐的
die:  jne   die     # 否则在此处死循环
    xorw %bx, %bx   # 内核读到 es:bx,即 0x1000:0x0000
# 开始处
rp_read:    # 读循环的起点,每轮循环要么读完一个磁道,要么
# 读满一个 64KB
#ifdef __BIG_KERNEL__    # 如果是编译成 bzImage,则 lcall
# 0x220,即调用 setup 中的子程序 bootsect_helper(setup 中偏
# 移 0x20 处的指针指向的程序),把 0x10000 处的 64KB 内核移
# 扩展内存 0x100000 开始处
    bootsect_kludge = 0x220         # 0x200  (size of boot-
sector) + 0x20 (offset
    lcall  bootsect_kludge# 返回后 ax=(已转移的内核字节数/
# 16)。每次调用 setup 中的 bootsect_helper 把读入 0x10000
# 的 64K 内核移到扩展内存(0x100000 开始处),bootsect_hel
# per 中会检查,不满 64K 是不会转移的,直至满 64K 为止。
#else          # 否则为 zImage,读入内存 0x10000 处
 movw%es, %ax    #ax=es-SYSSEG=读入内核的字节数/16
    subw    $SYSSEG, %ax
#endif
    cmpw    syssize, %ax          # ax 中是已读入的字节
# 数,检查内核是否已全部读入内存,是则返回,否则继续
  jbeok1_read   # 内核还没全部读入,转到 ok1_read 继续读
    ret
ok1_read:
    movw    sectors, % ax          # sectors 中存放着
probe_loop 时检测出的每道扇区数
    subw    sread, %ax            # 每道扇区数-已读入扇
# 区数(bootsect+setup=1+4=5),即 0 磁道还需读入的扇区数
    movw    %ax, %cx              #
    shlw $9, %cx      # cx=0 磁道还需读入的字节数
    addw    %bx, %cx
    jnc   ok2_read              # 不超过 64KB 边界(16 位加法
# 不溢出),转向 ok2_read
    je    ok2_read              # 正好 64KB 也转向 ok2_read
    xorw %ax, %ax               #ax 清 0
    subw    %bx, %ax              #ax 中存放着段内起始
# 地址 bx 的补码,其实就是为了 64KB 对齐此次能够读入的字
# 节数(因为 ax+bx=0x0000,是 64KB 边界对齐的)
    shrw $9, %ax            # 字节数右移 9 位,得到此次可以
# 读入的扇区数,放在 ax 中
ok2_read:
    call   read_track           # 读磁道。参数:ax=读扇区数,
# bx=段内偏移,head=磁头号,track=磁道号,sread=当前磁道已
# 读扇区数
    movw    %ax, %cx            #cx=ax=读扇区数
    addw    sread, %ax          #ax=sread+ax,即当前磁
# 道已读入扇区数
    cmpw    sectors, %ax          # 比较每磁道扇区数
    jne   ok3_read    # 若当前磁道还没读完,转到 ok3_read
      # 否则准备读下一磁道,ax、sread 清 0
    movw    $1, %ax             # ax=1
```

```
    subw    head, %ax    # 1-head,当前是 0 磁头的话,则
# ax=1,表示下一次读 1 磁头;当前已是 1 磁头,则磁道号增 1
    jne   ok4_read          # 每个磁道有 0 磁头和 1 磁头
# 两面,ax-head=0,说明某磁道已读完,此次该读下一磁道
    incw track              # 调整 track(磁道号增 1)
ok4_read:
    movw    %ax, head   # 调整 head(新的磁头号存入
# head)
    xorw %ax, %ax           #ax=0
ok3_read:
    movw    %ax, sread          # 调整 sread 值(sread 中
# 存放当前磁道已读扇区数。如果是从 ok4_read 下来的,则磁
# 道号刚增 1,sread=ax=0;否则是从 ok2_read 下来的,是新的
# 已读扇区数(sread=ax+sread))
    shlw $9, %cx            # 刚读字节数
    addw    %cx, %bx    # 调整段内偏移(范围(0,64K))
    jnc   rp_read           # 若偏移<64KB,则 es 不变,直接
# 转到 rp_read 进行下一轮读;否则(即已读完一个 64KB 了)要
# 调整 es(es=es+0x1000)和 bx(bx=0)
    movw    %es, %ax            #ax=es
    addb $0x10, %ah         #ax=ax+0x1000
    movw    %ax, %es            #es=ax,整个过程相当于
es=es+0x1000
    xorw %bx, %bx           #bx 清 0,所以 es:bx 表示的
# 内存地址增加了 0x10000(即 64KB)
    jmp  rp_read            # 返回 rp_read 进行下一轮读
# 以下读磁道。参数:ax=读扇区数,bx=段内偏移,head=磁头号,
# track=磁道号,sread=当前磁道已读扇区数
read_track:
    pusha
    pusha
    movw    $0xe2e, %ax   # loading... message 2e = .
    movw    $7, %bx
    int   $0x10
    popa
    movw    track, %dx          #dl=磁道号
    movw    sread, %cx
    incw %cx   #cl=读起始扇区号(当前磁道已读扇区数+1)
    movb    %dl, %ch            #ch=磁道号
    movw    head, %dx
    movb    %dl, %dh            #dh=磁头号
    andw    $0x0100, %dx        #dl=0(读软驱)
    movb    $2, %ah     #ah=2,表示读磁盘;al=读扇区数
    pushw   %dx                 # save for error dump
    pushw   %cx
    pushw   %bx
    pushw   %ax
    int   $0x13             # 调用 0x13 号功能读入
    jc    bad_rt            //不成功转到 bat_rt
    addw    $8, %sp         //以下为堆栈恢复,函数返回
    popa
    ret
```

整个读入过程的核心是 rp_read 主循环，下面总结一下其流程：

1）rp_read 是循环的起点，每轮循环要么读完一个磁道，要么读满一个 64KB，如果定义了—BIG_KERNEL—(生成 bzImage 的情况)，则把刚读入的 64KB 移到内存 0x100000 以上。

2）rp_read：检查内核读完否，是则返回 (ret) ,否则转到 ok1_read

3）ok1_read：计算当前磁道还需读入的扇区、字节数，检查(此字节数+bx 段内偏移) 是否超出 64KB 边界，若不超出或正好满 64KB，则转向 ok2_read;否则调整需读字节数，使 (字节数+bx) =64KB，将字节数折合到扇区数 (左移 9 位)，执行 ok2_read。

4）ok2_read：调用 read_track 读磁道 (之前已准备好的参数有 ax=读扇区数，bx=段内偏移，head=磁头号，track=磁道号，sread=当前磁道已读扇区数)。检查当前磁道读完否：若已读完，调整 track，调整 head (在 ok4_read 处)，转到 ok3_read;否则直接转到 ok3_read。

5）ok3_read：调整 sread，调整段内偏移 bx。检查段内偏移是否达到 64KB，是则调整 es=es+0x1000，bx=0，然后回到 rp_read 进行下一轮循环；否则直接转到 rp_read，无需调整 es。

(2) setup.S 中供 bootsect.S 调用的子程序 bootsect_helper 分析：

```
bootsect_helper:
    cmpw    $0, %cs:bootsect_es         # cs:boot
# sect_es 在编译时被静态的设置为 0。因此 bootsect 第一次调
# 用这个子程序时(当时还未开始读内核),不会转到 boot
# sect_second。以后调用时,该处的值已是 0x1000,所以会直
# 接转到 bootsect_second。
    jnz   bootsect_second           # cs:bootsect_es<>0
# 时直接跳到 bootsect_second 处执行
    movb    $0x20, %cs:type_of_loader
    movw    %es, %ax
    shrw $4, %ax                # ax=es=0x1000,ah=
# 0x10,因此 shrw ax 后,ah=1
    movb    %ah, %cs:bootsect_src_base+2  # 将 cs:
# bootsect_src_base 处的值设置为 0x10000,这个值将保持不变
    movw    %es, %ax      #ax=es=0x1000=SYSSEG
    movw    %ax, %cs:bootsect_es        # cs:boot
# sect_es=es=0x1000,这个值也将保持不变
    subw    $SYSSEG, %ax            # ax=0，可看作
# 是(已转移的内核总字节数/16)
    lret    # 返回。第一次被调用时只做以上的这些工作,并
# 不做内核转移。以后被调用时就不做这些工作了,而是直接转
# 到 bootsect_second
bootsect_second:  # 从这开始是主体代码。当读到 0x10000
# 处的内核达到 64K 的话，就调用 0x15 号功能将 0x10000 处
# 的 64K 移到扩展内存(0x100000 处)
    pushw   %cx
    pushw   %si
    pushw   %bx
    testw   %bx, %bx            # 64KB full? 当读入而
# 尚未转移的内核达到 64K 时才实施转移(bx=0 时);否则不转
```

```
# 移而跳到 bootsect_ex 处,那里把 ax 赋值为已转移到 0x100000
# 的内核字节数后就返回(当>=syssize 时,返回 bootsect 中后
# 还会继续返回;否则返回后会继续从磁盘读,直至读满 64K 为止)
    jne   bootsect_ex
    movw    $0x8000, %cx            # full 64KB, INT15
# moves words 满 64K 时, 即调用 int 0x15 进行转移;cx 为转
# 移的双字节数,即转移 0x8000*2=0x10000 字节
    pushw   %cs
    popw    %es
    movw    $bootsect_gdt, %si
    movw    $0x8700, %ax                # 调用 int 0x15
# 的 0x87 号功能实行转移。ah=0x87,es:si 指向 gdt 表(这里是
# bootsect_gdt),cx=待转移的双字节数
    int   $0x15  # 这个 gdt 中包含目标地址和源地址的描述符
    jc    bootsect_panic      # this, if INT15 fails 。 转移
# 失败则跳到 bootsect_panic 执行
    movw    %cs:bootsect_es, %es        # we reset
# %es to always point 每次转移 64K 结束后,es 始终恢复为
# 0x1000。这就使每次转移 64K 后,返回 bootsect 后还从磁盘
# 读入 64K 到 0x10000
    incb  %cs:bootsect_dst_base+2     # to 0x10000   目
# 标基地址增加 0x10000
bootsect_ex:
/* 这里说明一下。movb   % cs:bootsect_dst_base +2,  % ah
以后,ah 的初始值为 0x10;
shlb $4,%ah 后是 0x100,那么 ax 初始值是 0x0000(最高位"溢
出"了)
以后目标地址每次增加 0x10000,ah 每次会增加 1，经过左移后,
ax 每次增加 0x1000,正好是(已转移到扩展内存的内核字节数/16)
*/
    movb    %cs:bootsect_dst_base+2, %ah   # 进行一
# 些运算,保证 ax 的值=(已转移到扩展内存的内核字节数/16)
    shlb  $4, %ah   # we now have the number of
            # moved frames in %ax
    xorb  %al, %al
    popw    %bx
    popw    %si
    popw    %cx
    lret                            # 返回到 bootsect 中
```

在启动阶段，CPU 运行于实地址模式下，只能借助（16 位段寄存器:16 位逻辑地址）的模式访问不高于 1MB 的内存，int 0x15 的 0x87 号功能实际是借助了保护模式的访存能力把内核数据移动到扩展内存中的，因此在调用前预先设置了 GDT 中的两个段描述符，一个段描述符是指向源地址 0x10000，这是不变的；另一个目标段描述符的基址 bootsect_dst_base 指向目标地址 0x100000，每次移完 0x10000 字节后，目标段址也后移 0x10000，两个段长度都被预设为 0x10000（每次移动 64KB 的内存块）。当读内核循环结束后，bzImage 的自解压内核映像就位于内存 0x100000 处。

（3）linux/arch/i386/boot/compressed/head.S 代码分析

setup.S 的目标代码运行结束前，跳转到 0x1000（或 0x100000）处的内核自解压映像处执行那里的 linux/arch/i386/boot/compressed/head.S 的目标代码。

```
/* 小内核zImage 的话,现在是在 0x1000 运行;否则(bzImage)
是在 0x100000 运行,运行至此,已经在保护模式下了,但是没
有开启分页 */
startup_32:
    cld
    cli
    movl $ (__KERNEL_DS),%eax     # 数据段选择子 _KER-
NEL_DS,基址=0
    movl %eax,%ds               /* 以下设置 ds、es、fs、gs 为
基址 0 的核心数据段 */
    movl %eax,%es
    movl %eax,%fs
    movl %eax,%gs
    lss SYMBOL_NAME(stack_start),%esp     # 堆栈设置
    xorl %eax,%eax
1:  incl %eax       # check that A20 really IS enabled
/* 检查 A20 地址线是否开启 */
    movl %eax,0x000000      # loop forever if it isn't
    cmpl %eax,0x100000
    je 1b               /* 循环检测 A20 地址线 */
/*
 * Initialize eflags.  Some BIOS's leave bits like NT set.  This
would
 * confuse the debugger if this code is traced.
 * XXX – best to initialize before switching to protected
mode.
 */
    pushl $0      #EFLAGS 寄存器清 0
    popfl
/* BSS 段清 0  */
    xorl %eax,%eax
    movl $ SYMBOL_NAME(_edata),%edi
    movl $ SYMBOL_NAME(_end),%ecx
    subl %edi,%ecx
    cld
    rep
    stosb
/* 调用 misc.c 中的 decompressed_kernel()进行内核解压,返回
该函数 high_loaded 标志,如果返回非零值表示解压的内核代码分
两块存放的,转去执行合并代码。最终,解压后的内核代码会位于
内存 0x100000 处,最后一跳是执行 0x100000 开始的内核代码。
 */
    subl $16,%esp # place for structure on the stack /* 堆
栈中留出 16 字节,作为 decompressed_kernel()的第一个参数
(指向 16 字节的结构 moveparams)*/
    movl %esp,%eax
    pushl %esi      # real mode pointer as second arg
/* 第二个参数是 seup.S 传下来的 esi 值,即 0x90000*/
    pushl %eax      # address of structure as first arg /* 第
一个参数是 moveparams 结构指针，指向上边留出的 16 字节
*/
```

```
        call SYMBOL_NAME(decompress_kernel)          /* 调用decompressed_kernel()解压数据段中的内核代码(input_data~input_data_end 处)*/
        orl  %eax,%eax
        jnz  3f
        popl %esi # discard address
        popl %esi # real mode pointer
        xorl %ebx,%ebx
        ljmp $(__KERNEL_CS), $0x100000
/* 对于大内核 bzImage,会跳转到这里运行,把 low_buffer 和 high_buffer 处的解压代码合并到 0x100000。大内核在 decompressed_kernel()完成后,解压的代码分成两部分存放,分别在 low_buffer 和 high_buffer 处 */
/* 大内核的 compressed/head.S 本身是在 0x100000 运行,合并中这里的代码会被覆盖,因此先把合并功能代码 (从 move_routine_start 到 move_routine_end) 移至 0x1000 处,然后转到 0x1000 处运行 */
3:
        movl $move_routine_start,%esi
        movl $0x1000,%edi
        movl $move_routine_end,%ecx
        subl %esi,%ecx
        addl $3,%ecx
        shrl $2,%ecx
        cld
        rep
        movsl
        popl %esi # discard the address
        popl %ebx       # real mode pointer
        popl %esi          /*low_buffer 的起始地址 */
        popl %ecx          /*low_buffer 的大小(字节数)*/
        popl %edx          /*high_buffer 的起始地址 */
        popl %eax                          /*high_buffer 的大小 */
        movl $0x100000,%edi                /* edi=0x100000,合并后的内核地址 */
        cli             # make sure we don't get interrupted
        ljmp $ (__KERNEL_CS), $0x1000 # and jump to the move routine
/*
 * Routine (template) for moving the decompressed kernel in place,
 * if we were high loaded. This _must_ PIC-code !
 */
/* 以下是合并功能代码:把 low_buffer_start 开始的 lcount 字节和 high_buffer_start 开始的 hcount 字节合并到 0x100000,这就是完整的解压后的内核代码小内核不会运行到这里:因为从磁盘读入到 0x10000 (后转移到 0x1000) 后就直接解压到 0x100000 运行了大内核从磁盘读入转移到 0x100000 处后,是解压成两部分的,因此这里要再合并到 0x100000。注意此时,%esi\%ecx\%edx\%eax 分别存放有 low_buffer_start\lcount\high_buffer_start\hcount */
move_routine_start:
        movl  % ecx,% ebp       /* 先把 low_buffer_start 处的 lcount 字节移到 0x100000,这是第一部分 */
        shrl $2,%ecx
        rep
        movsl
        movl %ebp,%ecx
        andl $3,%ecx
        rep
        movsb
        movl  % edx,% esi       /* 再把第二部分——high_buffer_start 处的 hcount 字节移到第一部分解压代码后,这样就合并了 */
        movl %eax,%ecx     # NOTE: rep movsb won't move if %ecx == 0
        addl $3,%ecx
        shrl $2,%ecx
        rep
        movsl
        movl %ebx,%esi     # Restore setup pointer
        xorl %ebx,%ebx
        ljmp $ (__KERNEL_CS), $0x100000  /* 最后跳转到 0x100000 开始真正内核的运行,即 boot/head.S 开始的代码 */
move_routine_end:
```

参考文献

[1] Linux 内核 2.4.0 源代码,从 www.kernel.org 下载.

[2] Linux Kernel 2.4 Internals.

(收稿日期:2011-04-11)

(上接第 10 页)

容器 vector。以上程序片段中随着搜索进程的进行,搜索到的文件每增加一个,如果需要再放入元素则 vector 会自动扩大自己的容量。push_back 是 vector 容器的一个类属成员函数,用来在容器尾端插入一个元素。这种用法比用 new、delete 操作符简洁,也减少了出错的概率。

4 结语

通过对 C++标准模板库 (STL) 的使用,有效解决了拆分字符串存储计算机 IP 地址和搜索目录内所有文件路径列表中需要动态添加数组元素的问题。vector 可以动态增长,可以容易地增加新元素。与 MFC 中的 CArray、CObArray 等方法相比,STL 解决动态数组问题很有效,使用了 vector 容器,可以事先不用定义数组大小,可以动态添加数组元素,参考文中的 vector<CString>,也可以对 vector<HWND>,vector<HTREEITEM>,vector<PointF>等举一反三,灵活运用。随着应用的深入,STL 还提供许多典型的算法,如排序等,STL 使用起来更简洁、方便。

(收稿日期:2011-03-23)