

Linux 汇编语言使用向导

Derick Swanepoel (derick@maple.up.ac.za)

李 峰 (lifeng@telebyte.com.cn)

2007年5月21日

目 录

1 介 绍	3
2 为什么写这个手册	4
3 Netwide汇编器(NASM)	4
3.1 关于汇编器的说明	4
3.2 我从哪找到NASM?	5
4 对Linux下的汇编的介绍	5
4.1 DOS和Linux下汇编的主要不同	5
4.2 汇编程序的各部分	6
4.2.1 .data部分	6
4.2.2 .bss部分	6
4.2.3 .text部分	7
4.3 Linux系统调用	7
4.4 读Manpages页	8
4.5 在Linux下的“Hello World!”汇编程序	9
4.6 编译和链接	9
5 更多高深的概念	10
5.1 命令行参数和堆栈	10
5.2 “Procedures”和Jumping	11
5.3 处理任何情况的程序代码	13
6 结 论	13

版权说明

这篇文章是译者(李峰)翻译自互联网上的,关于 Linux 下的汇编程序的说明文档。这篇文章的中文版权归本文的翻译者所有。读者出于学习目的,可自由地从译者的个人网站(<http://lifeng.telebyte.com.cn>)下载本文 PDF 格式的文档。如用户文章中引用本文档内容或部分文字,请指明本文出处。

对于本文中的任何错误,欢迎读者指正,译者将在后继的版本中更正相应的错误。对于本文的错误更正,请发送邮件到 lifeng@telebyte.com.cn
邮件主题:“汇编文档错误”。

英文版出处为以下网址:

http://members.save-net.com/jko@save-net.com/asm/h_code_tut.htm

鸣 谢

在这里这要感谢原文作者对于 Linux 下汇编编程的精采论述。同时感谢 GUN 软件 Emacs 作者的杰出贡献。没有这个功能强大的编辑器,我不可能完成这对我来说“过于复杂”的“L^AT_EX2e”的文档的编写。在这里也感谢杰出的排版软件 L^AT_EX2e 的作者们与 T_EX 作者的杰出贡献。

第一节

介 绍

这是一篇介绍 Linux 下汇编语言的文章。这里有两个不同的“版本”来适用于不同的读者:

详细向导: 这个版本解释每一步的细节信息。这里假设读者至少有一些 DOS 方面的汇编知识,并且在你的计算机上装有 Linux 操作系统(尽管你可能不常使用它)。因为还不是所有的人都知道如何使用 Linux 操作系统,这里有一个关于如何使用基本的命令,如终端和象 DOS 命令行命令的说明。

快速说明: 如果你非常急并且只想看 Linux 汇编程序,并且你知道如何使用终端,基本上,这里只是指出 Linux 和 DOS 汇编语言的不同,以免使你感到迷惑。

我们在这里使用的汇编器是NASM(Netwide Assembler)。在这里的许多材料得自NASM的 man 手册和NASM文档。—如需更多信息，请查看引用页。

第二节

为什么写这个手册

写这篇文章的主要目的是使汇编程序更容易编程，在Linux上更容易实践，而不仅在DOS上来实现汇编编程。另外，这里也将教你一些关于Linux的使用知识(除非你已经在家里使用它了)。

汇编程序看起来是非常神秘(用简单的方式写程序语句),尤其是目前使用超级优化的编译器和可视化开发工具可完成一切的编程现实情况下，汇编更显得神秘。然而，用汇编编程有一个好处是你更好地了解你的操作系统内核和处理器的的工作状况，并且汇编语言是学习操作系统内核和处理器工作的最好方法。有时，在C/C++程序中使用汇编代码是非常有效的编程手段。并且如果你的程序真的有“运行速度”的需求，你可以通过汇编语言来优化编译器产生的代码(当然你需要有更出色的才华来写出比今天的编译器更好的代码。)

因此这里的目的是“授之以渔”，而不是“授之以鱼”。在这里我们将代码放在Linux环境下。但不是Linux环境下组装汇编DOS下的汇编环境—DOS汇编，DOS仿真器，和DOS文本编辑器。当然，这全是为实现汇编的默认想法。这里写这篇文章的主要原因是没有许多如同DOS下详细的技术资料，来介绍关于Linux下如何使用汇编语言编写代码。好了，现在我们在这里将教你基本的有关Linux汇编的知识。

第三节

Netwide汇编器(NASM)

3.1 关于汇编器的说明

Linux操作系统中几乎总是默认安装的汇编器是as和as86，并且很可能还有gas，然而，我们将使用NASM，Netwide汇编器。它使用Intel语法结构，就如同DOS下的汇编器TASM,MASM等一样，并且语法结构也非常相似。(无用的信息：as和gas使用的是AT&T语法,这和Intel的语法有些不同—如：所有的寄存器地址前必须有一个%，并且源操作数在目的操作数前出现。相关的信息请查阅AT&T语法的使用手册。)

NASM 非常“酷”是因为它可移植性(有 Linux, Unix 和 DOS 版本),它是自由软件(可免费得到)和具有很多功能的、强有力的汇编工具,请相信我在这里发表的意见。

3.2 我从哪找到 NASM ?

当你安装 Linux 时选择“开发工具”,并选择安装 NASM。在大多数的 Linux 发行版本中都有 NASM 这个安装包,所以,你不需要下载它。如果想查看是不是你已经安装了它,仅需在 Linux 下使用命令“where is NASM”。这里是如何操作的步骤:

- 打开一个终端。(如需得到基本的 Linux 终端使用技巧,请访问终端是你的朋友——如何使用它¹)
- 输入 `whereis nasm` 并点击 ENTER。

如果你看到有 `nasm` 的一行如: `nasm: /usr/bin/nasm` 那代表你系统已安装了。如是 `nasm:` 这代表你需要安装 NASM。这里有一些相关的介绍²,说明了如何安装 NASM (或其它细节)与 Linux 使用的一些相关资料。

如果你想得到最新版本的 NASM,访问站点 www.cryogen.com/Nasm,或用 FTP 工具软件从我们的 Linux 镜像服务器:
<ftp.kernel.za.org/pub/software/devel/nasm/binaries> 上下载。

第四节

对 Linux 下的汇编的介绍

4.1 DOS 和 Linux 下汇编的主要不同

- 对于 DOS 下的汇编,大多数情况下是得到 DOS 的服务请求中断 `int 21h`, 和 BIOS 服务中断如 `int 10h` 和 `int 16h`。在 Linux 中,所有这些功能都是由操作系统内核来处理。所有的结果都是由“内核系统调用”来实现的,并且是你通过向操作系统内核发出中断请求 `int 80h` 来完成。Linux 相对于 DOS 操作系来说的一个优点是所使用的系统调用要少,但是他需要你有更多的实践(你不能荒废如函数等一些基本编程概念)。Linux 系统调用建立文件,处理过程和完成其它工作。

¹http://members.save-net.com/jko@save-net.com/asm/h_code_tut.htm/#terminal

²http://members.save-net.com/jko@save-net.com/asm/h_code_tut.htm#installingnasm

- Linux是一个真正的32-bit保护模式的操作系统，所以这允许我们来做一些真正的、现代的32-bit汇编。这些32-bit的代码运行于平面³形式内存模型中，这意味着你根本不用担心内存分段的问题。这使编程变的更加容易，因为你根本不需要用分段和更改分段寄存器的操作，并且每个内存地址都是32-bit长，并且只保留内存地址的偏移地址部分。(如果这里的介绍使你感到困惑，没有关系，你仅仅需要知道这使用编程更简单，并且有更好的功能就可以了) :-)
- 在32-bit汇编环境下,你使用扩展的32-bit寄存器EAX,EBX,ECX等等来替换正常的16-bit的寄存器AX,BX,CX等等。
- DOS系统已经死亡了。它是16-bit，并且已荒废了的操作系统。只有疯狂的黑客和钟情于386s的用户还使用这个系统和上面的汇编工具。Linux汇编有实际的实践应用(部分的操作系统代码是用汇编写的，硬件驱动通常是用汇编代码来编写)。

4.2 汇编程序的各部分

一个汇编程序可以分成三部分：

4.2.1 .data 部分

这部分代码是用来“声明初始化数据”，换种说法是定义已有内容的“变量”。然而这里定义的代码在程序运行时是不能改变的，所以它们不是真正的变量。.data部分被用来设置文件名和缓存尺寸等，并且你可以使用EQU语句来定义常量。在这部分，你可以使用DB,DW,DD,DQ和DT指示符。如：

```
section .data
    message db 'Hello world!' ;声明信息为字符串'Hello world!'  
    msglength: equ 12 ;声明字符串长度  
    buffersize: dw 1024 ;声明缓存尺寸的内容为1024
```

4.2.2 .bss 部分

这部分代码用来声明变量。你可以用RESB,RESW,RESD,RESQ和REST指令来在内存中为变量保留相应的内存空间。如下面的代码：

```
section .bss
    filename: resb 255 ;保留255字节  
    number: resb 1 ;保留1字节
```

³flat mode

```
bignum: resw 1 ;保留1个字 (1个字=2个字节)
realarray: resq 10 ; 保留有10个元素的数组
```

4.2.3 .text 部分

在这部分代码是我们实际编写的汇编程序代码。 .text 部分必需以声明全局 `global _start`，它仅仅告诉内核程序从哪里开始。(这如同在 C 或 Java 语言中的 `main` 函数,仅作为一个程序开始的点。)例如:

```
section .text
    global _start

_start:
    pop    ebx ;这里是程序实际的开始部分
    ...
    ...
    ...
```

如同你看到的，大多数代码还是非常类似于 DOS 下的汇编语法。下面我们将更多的关注于系统调用细节并开始编写你第一个 Linux 下的汇编程序！

4.3 Linux 系统调用

Linux 的系统调用严格来说使用和 DOS 同样的方法来完成系统调用:

1. 你将系统调用号放入 EAX 中 (我们这里用的 32-bit 的寄存器，记住)
2. 你将系统调用的参数放入寄存器 EBX,ECX,等等
3. 你调用相关的中断(对于 DOS 是 21h; Linux 是 80h)
4. 处理结果通常返回到寄存器 EAX

这里有六个寄存器来存储系统调用时用到的参数。第一个参数放在 EBX 中，第二个参数放在 ECX 中，之后的参数放在 EDX,ESI,EDI,和最后一个寄存器 EBP 中，如果这里有太多的系统调用参数，如多于六个系统调用参数，EBX 必需存放一组系统调用参数所存放的内存地址(一个指向参数存放内存位置的指针).但不用为此担心，因为通常不大可能用多于六个参数的系统调用 —— 所有的系统调用都按这种方式来设计，希望这里不将你搞糊涂。

下面的一些示例代码将帮助你理解上面的概念:

```
mov    eax,1    ; exit 的系统调用号
mov    ebx,0    ; 有一个exit的返回码0
int    80h      ; 80h中断, 是提醒内核并说, “嘿, 运算这个”
```

但是你如何找到系统调用代号, 以及它们执行的操作, 和它们所用的参数? 首先, 所有的系统调用都在文件 `/usr/include/asm/unistd.h`⁴ 中列出, 系统调用名和代号都放在这个文件中(这个代号必须在调用系统中断 80h 前放入 EAX 中)。然而为了方便使用, 你可以查找Linux系统调用表⁵, 这个表中还有一些有用的信息(如,它用什么参数)。查看一下系统调用表——这里有如 `sys_write(4)`, `sys_nice(34)` 和 `sys_exit(1)`。要指出它们执行了什么动作, 你可以查看 Linux manual pages (通常被称作 “manpages”)。这将是下面的章节中谈论的内容。

4.4 读 Manpages 页

首先, 打开一个终端(或用 CTRL+ALT+F1 切换到六个终端中的一个——如想返回图形界面按 CTRL+ALT+F7)。现在我们想知道“write”系统调用的实现。输入 `man 2 write` 并按 ENTER。这将打开手册页中关于“write”的第 2 部分的内容。

在 NAME 章节, 介绍了系统调用名和如何使用这个系统调用——象下面这样:

```
write - write to a file descriptor
```

这是一个系统调用, 用来写到一个文件, 但你也可以用它来打印一些东西到屏幕上。“为什么黑客能?” 当你问这个问题时, 你要想到, 在 Linux 环境下任何事物都被看做是一个文件。如屏幕, 鼠标、打印机, 等等, 这些是特殊的文件, 被称作“设备文件”, 但你可以象对文本文件一样来对它们进行读、写。这实际上是为了清楚了解过程, 因为读/写文件是在编程中最简单的工作, 所以为什么不将所有的事情都按最简单的方式来处理呢——这里我跑题了。

下面, 在 SYNOPSIS 章节你将看到相当难于理解的一行:

```
ssize_t write(int fd, const void *buf, size_t count);
```

好的, 如果你懂 C 它将不再看起来难于理解, 因为这里仅是一个 C 语言中的系统调用定义。如同你看到的, 它有三个参数: 文件说明符, 之后是一个缓存, 最后是多少字节将被写入, 总之, 缓存区的长度应该更长些。(DESCRIPTION 章节告诉我们需要什么参数)。文件说明符(fd)是一个整数, 缓存(buf)是一个指向内存位置的指针(这是 C 语言中 * 代表的意义)。所以它也是一个整数, 并且写 size_t 个相应类型的数, 这个参数也是

⁴这个文件因不同的处理器, 而放置的位置也不同, 通常这是指的内核源代码树中的 `include/i386/asm/unistd.h` 这个文件

⁵原文中的这个链接已失效

一个整数。这明确地说明我们放到三个寄存器 EBX,ECX 和 EDX 中的参数，都是 32-bit 的整形数。最终，write 系统调用返回一个值到 EAX：实际写的位数。这可以用来验证是否程序执行正确。

现在，我们可以写出我们第一个 Linux 的汇编程序！

4.5 在 Linux 下的“Hello World!”汇编程序

当然，最合适的方法是打印“Hello world!”字符到屏幕上，我们将字符写到特殊的文件说明符“STDOUT”(标准输出)，这个说明符的代码为1。下面列出相应的汇编代码：

```
section .data
hello:      db 'Hello world!',10      ; 'Hello world!'加上行填充符
helloLen:   equ $-hello              ; 'Hello world!'串的长度
                                                ; (这将很快在后面解释)

section .text
global _start

_start:
mov eax,4          ; write系统调用(sys_write)
mov ebx,1          ; 文件说明符 1 是标准输出
mov ecx,hello      ; 将hello的偏移地址放入ecx寄存器中
mov edx,helloLen   ; helloLen是一个常量，所以我们不需要
                    ; mov edx,[helloLen] 来得到其实际值
int 80h           ; 调用内核

mov eax,1          ; exit (sys_exit)的系统调用
mov ebx,0          ; 以返回码0来退出 (无错误)
int 80h
```

拷贝这些程序代码到一个文本编辑器中 (根据自己的喜好，我选择的编辑器是 vi 或 SciTE)，并且将这些代码保存在你的家目录下的 (/home/你的用户名) hello.asm 。

4.6 编译和链接

1. 如果你没有打开一个终端或控制台，打开一个。
2. 确保你在与你所保存的 hello.asm 文件目录相同。

3. 汇编程序，输入 `nasm -f elf hello.asm`
如果有错误，NASM将告诉你哪行的代码有问题。
4. 现在输入 `ld -s -o hello hello.o` 这将链接目标文件到一个可执行文件。
5. 运行你的程序 `./hello` (为了在当前目录中运行你的程序/脚本，你需要在要运行的文件名前输入 `./`，除非当前目录在你的环境变量的搜索路径中。)

现在，你可以看到屏幕输出 `Hello world!`。祝贺你！你已经写了第一个能在 Linux 下运行的汇编程序！

第五节

更多高深的概念

在我们继续之前，你可能想知道在前面的 Hello World 程序中的(代码的第三行) `equ $-hello` 是什么意思。也许你会记得，当你用 `equ` 来声明一个变量(替换 `db`，如示例)，你实际上声明了一个常量。声明字符串的长度为一个常量是非常明智的，因为这确保它不可以被改变。但是 `$-hello` 为什么能返回“Hello world!”的长度呢？当 NASM 看到 `'$'` 这个符号，NASM 将这个符号替换成在汇编代码中本行开始处的内存位置(这也可以理解为上行结束的位置)。所以从 `'$'` 处减去变量的内存位置将使我们得到从 `'$'` 到变量开始处的字节数。如果我们想声明一个变量保存有我们所声明长度的内容，如 `hello: db 'Hello world!',10`，那么我们仅需在下面一行输入 `helloLen: equ $-hello` 来得到 `hello` 在内存中所占的字节数，这里是 13 (linefeed 字符也被计算在内)。如果你理解不了，没有关系，你仅需记住这是一个巧妙的和最容易的声明串长度的方法。

如果你不只是简单的被替换，我将鼓励你查看一下 NASM 的文档来得到更多的信息。如何使用其它巧妙的方法我在这里就不介绍了。

5.1 命令行参数和堆栈

从 DOS 环境下得到命令的参数不是令人愉快的经历，因为用 PSP 工作必需考虑使段尽量简单，这是非常令人痛苦的工作。在 Linux 环境下，这十分简单：在程序开始时，所有的参数在堆栈中都可得到，所以如想得到他们，仅需要使用 `pop` 指令从堆栈中得到它们。

这里举一个例子，运行一个被称作 `program` 的程序并给它三个参数：

```
./program foo bar 42
```

这个栈看起来如下面的样子：

4	参数的个数(argc),包括程序名
program	程序名称(argv[0])
foo	第一个参数(argv[1])
bar	第二个参数(argv[2])
42	第三个参数(argv[3])(注意:这里是字符串“42”不是数字42)

现在让我们写一个命名为 program 的，有三个参数的程序：

```
section .text
global _start

_start:
pop eax ; 得到参数的数目
pop ebx ; 得到程序名
pop ebx ; 得到第一个实际参数('foo')
pop ecx ; "bar"
pop edx ; "42"

mov eax,1
mov ebx,0
int 80h ; 退出
```

在所有的出栈操作完成后,EAX寄存器中存放的内容是参数的数目,EBX指向存放“foo”的内存的位置,ECX指向存放“bar”内存的位置且EDX指向存放“42”的内存的位置。这比DOS来说,明显的简单。它让我们仅用5行代码就得到参数,并且可以得知它们是多少,然而如在DOS环境下,需要14行代码仅能得到一个参数!注意第三个出栈操作重写了第三个出栈操作中的值(这里是程序名)。除非你有更充分的理由,通常你可以同我们现在一样在程序中抛弃程序名这个参数。

5.2 “Procedures” 和 Jumping

注意： NASM没有象我们在TASM中所定义的procedure语句。这是因为在汇编语言中过程不是真实存在的：所有的语句都是一个标签。所

以如果你想写一个“过程”在NASM中，你不用 `proc` 和 `endp`，而只是用一个在“procedure”(过程)代码前加一个标签来替换(例如，`fileWrite:`)。如果你想这么做，你可以在这段代码之前和之后放一些注释语句来说明这个过程。这样可以使这段代码看起来更象过程。这里有一组在Linux和DOS下的对比代码段：

Linux	DOS
<pre>;proc fileWrite-write a string to a file fileWrite: mov eax,4 ;write system call mov ebx,[filedesc] ;File descriptor mov ecx,stuffToWrite mov edx,[stuffLen] int 80h ret ;endp fileWrite</pre>	<pre>proc fileWrite mov ah,40h ; write DOS service mov bx,[filehandle] ;File handle mov cl,[stuffLen] mov dx,offset stuffToWrite int 21h ret endp fileWrite</pre>

注意:我假设你熟悉标签和跳转指令，如 `JMP`, `JE` 或 `JGE`, 现在你已经看到了“procedures”是如何用标签指令来实现的，这是你需要记住的非常重要的一点：如果你想从过程中返回(使用 `RET` 指令), **不是用跳转到相应位置!** 你千万不要用跳转语句，因为在Linux环境下这将引起段失败错误(这可能没有关系—因为你所有的程序是在终端中运行的)，但是在DOS中这样做将会使你面对很多的麻烦。你需要记住的规则是：

你可以跳转到标签，但你必需 `call` 一个 `procedure`。

如果调用一个过程，当然使用 `CALL` 指令。这在你想写“if-then-else”这样的语句时会带来些麻烦。如果你已有一个这样的情况需要编程，如“如果这种情况发生，调用过程1，否则调用过程2”这里只有一件事需要做：如同袋鼠在意大利面条间跳转一样编写代码。让我们看一下示例。首先，这里有些正常的代码，如：

```
if(Ax=='w'){
    writeFile();
}else{
    doSomethingElse();
}
```

下面是你如何利用汇编来实现上面的功能：

```
cmp  Ax, 'w'    ;AX寄存器中有'w'吗?
jne  skipWrite ;如果没有跳转
```

```
                ;通过跳到其它标签，
                ;并在这里完成其它的一些任务
call writeFile ; ...否则调用 writeFile 过程
jmp outOfThisMess ;...在这里跳出

skipWrite:
    call doSomethingElse
outOfThisMess:
...           ;这里是其它的程序代码
...
```

注意，这里情况适用于任何环境下的汇编，不仅仅是在 Linux 和 DOS 下的 NASM 适用。

5.3 处理任何情况的程序代码

现在我们可以最终看一段程序，它在远程调用时很有用，这段代码几乎有我们提到的所有的注意点。在快速说明版本中，其中已经包含了一段 Linux 和 DOS 版本的练习 3 程序(将“Hello world!”写入命令行参数的文件中)。并且来比较 Linux 下的代码比 DOS 下的代码简单多少。

第六节

结 论

好了，上面就是这个向导的内容。我希望这对你写 Linux 下的汇编程序代码有所帮助。如果你有任何问题和建议，请通过电子邮件和我联系，我的邮件地址是: derick@maple.up.ac.za。这是我写的第一版手册而我不是汇编程序的黑客，所以欢迎提出宝贵意见。

祝你好运并且编程愉快！