专题1-Linux 汇编语言开发指南

一、简介

作为最基本的编程语言之一，汇编语言虽然应用的范围不算很广，但重要性却勿庸置疑，因为它能够完成许多其它语言所无法完成的功能。就拿 Linux 内核来讲，虽然绝大部分代码是用 C 语言编写的，但仍然不可避免地在某些关键地方使用了汇编代码，其中主要是在 Linux 的启动部分。由于这部分代码与硬件的关系非常密切，即使是 C 语言也会有些力不从心，而汇编语言则能够很好扬长避短，最大限度地发挥硬件的性能。

大多数情况下 Linux 程序员不需要使用汇编语言，因为即便是硬件驱动这样的底层程序在 Linux 操作系统中也可以用完全用 C 语言来实现，再加上 GCC 这一优秀的编译器目前已经能够对最终生成的代码进行很好的优化，的确有足够的理由让我们可以暂时将汇编语言抛在一边了。但实现情况是 Linux 程序员有时还是需要使用汇编，或者不得不使用汇编，理由很简单：精简、高效和 libc 无关性。假设要移植 Linux 到某一特定的嵌入式硬件环境下，首先必然面临如何减少系统大小、提高执行效率等问题，此时或许只有汇编语言能帮上忙了。

汇编语言直接同计算机的底层软件甚至硬件进行交互，它具有如下一些优点：

- 能够直接访问与硬件相关的存储器或 I/O 端口；
- 能够不受编译器的限制，对生成的二进制代码进行完全的控制；
- 能够对关键代码进行更准确的控制，避免因线程共同访问或者硬件设备共享引起的死锁；
- 能够根据特定的应用对代码做最佳的优化，提高运行速度；
- 能够最大限度地发挥硬件的功能。

同时还应该认识到，汇编语言是一种层次非常低的语言，它仅仅高于直接手工编写二进制的机器指令码，因此不可避免地存在一些缺点：

- 编写的代码非常难懂，不好维护；
- 很容易产生 bug，难于调试；
- 只能针对特定的体系结构和处理器进行优化；
- 开发效率很低，时间长且单调。

Linux 下用汇编语言编写的代码具有两种不同的形式。第一种是完全的汇编代码，指的是整个程序全部用汇编语言编写。尽管是完全的汇编代码，Linux 平台下的汇编工具也吸收了 C 语言的长处，使得程序员可以使用 #include、#ifdef 等预处理指令，并能够通过宏定义来简化代码。第二种是内嵌的汇编代码，指的是可以嵌入到C语言程序中的汇编代码片段。虽然 ANSI 的 C 语言标准中没有关于内嵌汇编代码的相应规定，但各种实际使用的 C 编译器都做了这方面的扩充，这其中当然就包括 Linux 平台下的 GCC。

二、Linux 汇编语法格式

绝大多数 Linux 程序员以前只接触过DOS/Windows 下的汇编语言，这些汇编代码都是 Intel 风格的。但在 Unix 和 Linux 系统中，更多采用的还是 AT&T 格式，两者在语法格式上有着很大的不同：

1. 在 AT&T 汇编格式中，寄存器名要加上 '%' 作为前缀；而在 Intel 汇编格式中，寄存器名不需要加前缀。例如：

AT&T 格式	Intel 格式
pushl %eax	push eax

2. 在 AT&T 汇编格式中，用 '$' 前缀表示一个立即操作数；而在 Intel 汇编格式中，立即数的表示不用带任何前缀。例如：

AT&T 格式	Intel 格式
pushl $1	push 1

3. AT&T 和 Intel 格式中的源操作数和目标操作数的位置正好相反。在 Intel 汇编格式中，目标操作数在源操作数的左边；而在 AT&T 汇编格式中，目标操作数在源操作数的右边。例如：

AT&T 格式	Intel 格式
addl $1, %eax	add eax, 1

4. 在 AT&T 汇编格式中，操作数的字长由操作符的最后一个字母决定，后缀'b'、'w'、'l'分别表示操作数为字节（byte，8 比特）、字（word，16 比特）和长字（long，32比特）；而在 Intel 汇编格式中，操作数的字长是用 "byte ptr" 和 "word ptr" 等前缀来表示的。例如：

AT&T 格式	Intel 格式
movb val, %al	mov al, byte ptr val

5. 在 AT&T 汇编格式中，绝对转移和调用指令（jump/call）的操作数前要加上'*'作为前缀，而在 Intel 格式中则不需要。

6. 远程转移指令和远程子调用指令的操作码，在 AT&T 汇编格式中为 "ljump" 和 "lcall"，而在 Intel 汇编格式中则为 "jmp far" 和 "call far"，即：

AT&T 格式	Intel 格式
ljump $section, $offset	jmp far section:offset
lcall $section, $offset	call far section:offset

与之相应的远程返回指令则为：

AT&T 格式	Intel 格式
lret $stack_adjust	ret far stack_adjust

7. 在 AT&T 汇编格式中，内存操作数的寻址方式是

```
section:disp(base, index, scale)
```

而在 Intel 汇编格式中，内存操作数的寻址方式为：

```
section:[base + index*scale + disp]
```

由于 Linux 工作在保护模式下，用的是 32 位线性地址，所以在计算地址时不用考虑段基址和偏移量，而是采用如下的地址计算方法：

```
disp + base + index * scale
```

下面是一些内存操作数的例子：

AT&T 格式	Intel 格式
movl -4(%ebp), %eax	mov eax, [ebp - 4]
movl array(, %eax, 4), %eax	mov eax, [eax*4 + array]
movw array(%ebx, %eax, 4), %cx	mov cx, [ebx + 4*eax + array]
movb $4, %fs:(%eax)	mov fs:eax, 4

三、Hello World!

真不知道打破这个传统会带来什么样的后果，但既然所有程序设计语言的第一个例子都是在屏幕上打印一个字符串 "Hello World!"，那我们也以这种方式来开始介绍 Linux 下的汇编语言程序设计。

在 Linux 操作系统中，你有很多办法可以实现在屏幕上显示一个字符串，但最简洁的方式是使用 Linux 内核提供的系统调用。使用这种方法最大的好处是可以直接和操作系统的内核进行通讯，不需要链接诸如 libc 这样的函数库，也不需要使用 ELF 解释器，因而代码尺寸小且执行速度快。

Linux 是一个运行在保护模式下的 32 位操作系统，采用 flat memory 模式，目前最常用到的是 ELF 格式的二进制代码。一个 ELF 格式的可执行程序通常划分为如下几个部分：.text、.data 和 .bss，其中 .text 是只读的代码区，.data 是可读可写的数据区，而 .bss 则是可读可写且没有初始化的数据区。代码区和数据区在 ELF 中统称为 section，根据实际需要你可以使用其它标准的 section，也可以添加自定义 section，但一个 ELF 可执行程序至少应该有一个 .text 部分。 下面给出我们的第一个汇编程序，用的是 AT&T 汇编语言格式：

例1. AT&T 格式

```
#hello.s
.data                   # 数据段声明
        msg : .string "Hello, world!\\n" # 要输出的字符串
        len = . - msg                   # 字串长度
.text                   # 代码段声明
.global _start          # 指定入口函数

_start:                 # 在屏幕上显示一个字符串
        movl $len, %edx  # 参数三：字符串长度
        movl $msg, %ecx  # 参数二：要显示的字符串
        movl $1, %ebx    # 参数一：文件描述符(stdout)
        movl $4, %eax    # 系统调用号(sys_write)
        int  $0x80       # 调用内核功能

                         # 退出程序
        movl $0,%ebx     # 参数一：退出代码
        movl $1,%eax     # 系统调用号(sys_exit)
        int  $0x80       # 调用内核功能
```

初次接触到 AT&T 格式的汇编代码时，很多程序员都认为太晦涩难懂了，没有关系，在 Linux 平台上你同样可以使用 Intel 格式来编写汇编程序：

例2. Intel 格式

```
; hello.asm
section .data           ; 数据段声明
        msg db "Hello, world!", 0xA     ; 要输出的字符串
        len equ $ - msg                 ; 字串长度
section .text           ; 代码段声明
global _start           ; 指定入口函数
_start:                 ; 在屏幕上显示一个字符串
        mov edx, len    ; 参数三：字符串长度
        mov ecx, msg    ; 参数二：要显示的字符串
        mov ebx, 1      ; 参数一：文件描述符(stdout)
        mov eax, 4      ; 系统调用号(sys_write)
        int 0x80        ; 调用内核功能
                        ; 退出程序
        mov ebx, 0      ; 参数一：退出代码
        mov eax, 1      ; 系统调用号(sys_exit)
        int 0x80        ; 调用内核功能
```

上面两个汇编程序采用的语法虽然完全不同，但功能却都是调用 Linux 内核提供的 sys_write 来显示一个字符串，然后再调用 sys_exit 退出程序。在 Linux 内核源文件 include/asm-i386/unistd.h 中，可以找到所有系统调用的定义。

四、Linux 汇编工具

Linux 平台下的汇编工具虽然种类很多，但同 DOS/Windows 一样，最基本的仍然是汇编器、连接器和调试器。

1.汇编器

汇编器（assembler）的作用是将用汇编语言编写的源程序转换成二进制形式的目标代码。Linux 平台的标准汇编器是 GAS，它是 GCC 所依赖的后台汇编工具，通常包含在 binutils 软件包中。GAS 使用标准的 AT&T 汇编语法，可以用来汇编用 AT&T 格式编写的程序：

```
[xiaowp@gary code]$ as -o hello.o hello.s
```

Linux 平台上另一个经常用到的汇编器是 NASM，它提供了很好的宏指令功能，并能够支持相当多的目标代码格式，包括 bin、a.out、coff、elf、rdf 等。NASM 采用的是人工编写的语法分析器，因而执行速度要比 GAS 快很多，更重要的是它使用的是 Intel 汇编语法，可以用来编译用 Intel 语法格式编写的汇编程序：

```
[xiaowp@gary code]$ nasm -f elf hello.asm
```

2.链接器

由汇编器产生的目标代码是不能直接在计算机上运行的，它必须经过链接器的处理才能生成可执行代码。链接器通常用来将多个目标代码连接成一个可执行代码，这样可以先将整个程序分成几个模块来单独开发，然后才将它们组合(链接)成一个应用程序。 Linux 使用 ld 作为标准的链接程序，它同样也包含在 binutils 软件包中。汇编程序在成功通过 GAS 或 NASM 的编译并生成目标代码后，就可以使用 ld 将其链接成可执行程序了：

```
[xiaowp@gary code]$ ld -s -o hello hello.o
```

3.调试器

有人说程序不是编出来而是调出来的，足见调试在软件开发中的重要作用，在用汇编语言编写程序时尤其如此。Linux 下调试汇编代码既可以用 GDB、DDD 这类通用的调试器，也可以使用专门用来调试汇编代码的 ALD(Assembly Language Debugger)。

从调试的角度来看，使用 GAS 的好处是可以在生成的目标代码中包含符号表(symbol table)，这样就可以使用 GDB 和 DDD 来进行源码级的调试了。要在生成的可执行程序中包含符号表，可以采用下面的方式进行编译和链接：

```
[xiaowp@gary code]$ as --gstabs -o hello.o hello.s
[xiaowp@gary code]$ ld -o hello hello.o
```

执行 as 命令时带上参数 --gstabs 可以告诉汇编器在生成的目标代码中加上符号表，同时需要注意的是，在用 ld 命令进行链接时不要加上 -s 参数，否则目标代码中的符号表在链接时将被删去。

在 GDB 和 DDD 中调试汇编代码和调试 C 语言代码是一样的，你可以通过设置断点来中断程序的运行，查看变量和寄存器的当前值，并可以对代码进行单步跟踪。图1 是在 DDD 中调试汇编代码时的情景：

点击查看大图

图1 用 DDD 中调试汇编程序

汇编程序员通常面对的都是一些比较苛刻的软硬件环境，短小精悍的ALD可能更能符合实际的需要，因此下面主要介绍一下如何用ALD来调试汇编程序。首先在命令行方式下执行ald命令来启动调试器，该命令的参数是将要被调试的可执行程序：

```
[xiaowp@gary doc]$ ald hello
Assembly Language Debugger 0.1.3
Copyright (C) 2000-2002 Patrick Alken
hello: ELF Intel 80386 (32 bit), LSB, Executable, Version 1 (current)
Loading debugging symbols...(15 symbols loaded)
ald>
```

当 ALD 的提示符出现之后，用 disassemble 命令对代码段进行反汇编：

```
ald> disassemble -s .text
Disassembling section .text (0x08048074 - 0x08048096)
08048074  BA0F000000                mov edx, 0xf
08048079  B998900408                mov ecx, 0x8049098
0804807E  BB01000000                mov ebx, 0x1
08048083  B804000000                mov eax, 0x4
08048088  CD80                      int 0x80
0804808A  BB00000000                mov ebx, 0x0
0804808F  B801000000                mov eax, 0x1
08048094  CD80                      int 0x80
```

上述输出信息的第一列是指令对应的地址码，利用它可以设置在程序执行时的断点：

```
ald> break 0x08048088
Breakpoint 1 set for 0x08048088
```

断点设置好后，使用 run 命令开始执行程序。ALD 在遇到断点时将自动暂停程序的运行，同时会显示所有寄存器的当前值：

```
ald> run
Starting program: hello
Breakpoint 1 encountered at 0x08048088
eax = 0x00000004 ebx = 0x00000001 ecx = 0x08049098 edx = 0x0000000F
esp = 0xBFFFF6C0 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds  = 0x0000002B es  = 0x0000002B fs  = 0x00000000 gs  = 0x00000000
ss  = 0x0000002B cs  = 0x00000023 eip = 0x08048088 eflags = 0x00000246
Flags: PF ZF IF
08048088  CD80                      int 0x80
```

如果需要对汇编代码进行单步调试，可以使用 next 命令：

```
ald> next
Hello, world!
eax = 0x0000000F ebx = 0x00000000 ecx = 0x08049098 edx = 0x0000000F
esp = 0xBFFFF6C0 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds  = 0x0000002B es  = 0x0000002B fs  = 0x00000000 gs  = 0x00000000
ss  = 0x0000002B cs  = 0x00000023 eip = 0x0804808F eflags = 0x00000346
Flags: PF ZF TF IF
0804808F  B801000000                mov eax, 0x1
```

若想获得 ALD 支持的所有调试命令的详细列表，可以使用 help 命令：

```
ald> help
Commands may be abbreviated.
If a blank command is entered, the last command is repeated.
Type `help <command>' for more specific information on <command>.
General commands
attach          clear           continue        detach          disassemble
enter           examine         file            help            load
next            quit            register        run             set
step            unload          window          write
Breakpoint related commands
break           delete          disable         enable          ignore
lbreak          tbreak
```

五、系统调用

即便是最简单的汇编程序，也难免要用到诸如输入、输出以及退出等操作，而要进行这些操作则需要调用操作系统所提供的服务，也就是系统调用。除非你的程序只完成加减乘除等数学运算，否则将很难避免使用系统调用，事实上除了系统调用不同之外，各种操作系统的汇编编程往往都是很类似的。

在 Linux 平台下有两种方式来使用系统调用：利用封装后的 C 库（libc）或者通过汇编直接调用。其中通过汇编语言来直接调用系统调用，是最高效地使用 Linux 内核服务的方法，因为最终生成的程序不需要与任何库进行链接，而是直接和内核通信。

和 DOS 一样，Linux 下的系统调用也是通过中断（int 0x80）来实现的。在执行 int 80 指令时，寄存器 eax 中存放的是系统调用的功能号，而传给系统调用的参数则必须按顺序放到寄存器 ebx，ecx，edx，esi，edi 中，当系统调用完成之后，返回值可以在寄存器 eax 中获得。

所有的系统调用功能号都可以在文件 /usr/include/bits/syscall.h 中找到，为了便于使用，它们是用 SYS_<name> 这样的宏来定义的，如 SYS_write、SYS_exit 等。例如，经常用到的 write 函数是如下定义的：

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

该函数的功能最终是通过 SYS_write 这一系统调用来实现的。根据上面的约定，参数 fb、buf 和 count 分别存在寄存器 ebx、ecx 和 edx 中，而系统调用号 SYS_write 则放在寄存器 eax 中，当 int 0x80 指令执行完毕后，返回值可以从寄存器 eax 中获得。

或许你已经发现，在进行系统调用时至多只有 5 个寄存器能够用来保存参数，难道所有系统调用的参数个数都不超过 5 吗？当然不是，例如 mmap 函数就有 6 个参数，这些参数最后都需要传递给系统调用 SYS_mmap：

```
void  *  mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
```

当一个系统调用所需的参数个数大于 5 时，执行int 0x80 指令时仍需将系统调用功能号保存在寄存器 eax 中，所不同的只是全部参数应该依次放在一块连续的内存区域里，同时在寄存器 ebx 中保存指向该内存区域的指针。系统调用完成之后，返回值仍将保存在寄存器 eax 中。

由于只是需要一块连续的内存区域来保存系统调用的参数，因此完全可以像普通的函数调用一样使用栈(stack)来传递系统调用所需的参数。但要注意一点，Linux 采用的是 C 语言的调用模式，这就意味着所有参数必须以相反的顺序进栈，即最后一个参数先入栈，而第一个参数则最后入栈。如果采用栈来传递系统调用所需的参数，在执行int 0x80 指令时还应该将栈指针的当前值复制到寄存器 ebx中。

六、命令行参数

在 Linux 操作系统中，当一个可执行程序通过命令行启动时，其所需的参数将被保存到栈中：首先是 argc，然后是指向各个命令行参数的指针数组 argv，最后是指向环境变量的指针数据 envp。在编写汇编语言程序时，很多时候需要对这些参数进行处理，下面的代码示范了如何在汇编代码中进行命令行参数的处理：

例3. 处理命令行参数

```
# args.s
.text
.globl _start

_start:
        popl    %ecx        # argc
vnext:
        popl    %ecx        # argv
        test    %ecx, %ecx      # 空指针表明结束
        jz  exit
        movl    %ecx, %ebx
        xorl    %edx, %edx
strlen:
        movb    (%ebx), %al
        inc %edx
        inc %ebx
        test    %al, %al
        jnz strlen
        movb    $10, -1(%ebx)
        movl    $4, %eax        # 系统调用号(sys_write)
        movl    $1, %ebx        # 文件描述符(stdout)
        int $0x80
        jmp vnext
exit:
        movl    $1,%eax         # 系统调用号(sys_exit)
        xorl    %ebx, %ebx      # 退出代码
        int     $0x80

        ret
```

七、GCC 内联汇编

用汇编编写的程序虽然运行速度快，但开发速度非常慢，效率也很低。如果只是想对关键代码段进行优化，或许更好的办法是将汇编指令嵌入到 C 语言程序中，从而充分利用高级语言和汇编语言各自的特点。但一般来讲，在 C 代码中嵌入汇编语句要比"纯粹"的汇编语言代码复杂得多，因为需要解决如何分配寄存器，以及如何与C代码中的变量相结合等问题。

GCC 提供了很好的内联汇编支持，最基本的格式是：

```
__asm__("asm statements");
```

例如：

```
__asm__("nop");
```

如果需要同时执行多条汇编语句，则应该用"\\n\\t"将各个语句分隔开，例如：

```
__asm__( "pushl %%eax \\n\\t"
         "movl $0, %%eax \\n\\t"
         "popl %eax");
```

通常嵌入到 C 代码中的汇编语句很难做到与其它部分没有任何关系，因此更多时候需要用到完整的内联汇编格式：

```
__asm__("asm statements" : outputs : inputs : registers-modified);
```

插入到 C 代码中的汇编语句是以":"分隔的四个部分，其中第一部分就是汇编代码本身，通常称为指令部，其格式和在汇编语言中使用的格式基本相同。指令部分是必须的，而其它部分则可以根据实际情况而省略。

在将汇编语句嵌入到C代码中时，操作数如何与C代码中的变量相结合是个很大的问题。GCC采用如下方法来解决这个问题：程序员提供具体的指令，而对寄存器的使用则只需给出"样板"和约束条件就可以了，具体如何将寄存器与变量结合起来完全由GCC和GAS来负责。

在GCC内联汇编语句的指令部中，加上前缀'%'的数字(如%0，%1)表示的就是需要使用寄存器的"样板"操作数。指令部中使用了几个样板操作数，就表明有几个变量需要与寄存器相结合，这样GCC和GAS在编译和汇编时会根据后面给定的约束条件进行恰当的处理。由于样板操作数也使用'%'作为前缀，因此在涉及到具体的寄存器时，寄存器名前面应该加上两个'%'，以免产生混淆。

紧跟在指令部后面的是输出部，是规定输出变量如何与样板操作数进行结合的条件，每个条件称为一个"约束"，必要时可以包含多个约束，相互之间用逗号分隔开就可以了。每个输出约束都以'='号开始，然后紧跟一个对操作数类型进行说明的字后，最后是如何与变量相结合的约束。凡是与输出部中说明的操作数相结合的寄存器或操作数本身，在执行完嵌入的汇编代码后均不保留执行之前的内容，这是GCC在调度寄存器时所使用的依据。

输出部后面是输入部，输入约束的格式和输出约束相似，但不带'='号。如果一个输入约束要求使用寄存器，则GCC在预处理时就会为之分配一个寄存器，并插入必要的指令将操作数装入该寄存器。与输入部中说明的操作数结合的寄存器或操作数本身，在执行完嵌入的汇编代码后也不保留执行之前的内容。

有时在进行某些操作时，除了要用到进行数据输入和输出的寄存器外，还要使用多个寄存器来保存中间计算结果，这样就难免会破坏原有寄存器的内容。在GCC内联汇编格式中的最后一个部分中，可以对将产生副作用的寄存器进行说明，以便GCC能够采用相应的措施。

下面是一个内联汇编的简单例子：

例4.内联汇编

```
/* inline.c */
int main()
{
    int a = 10, b = 0;
    __asm__ __volatile__("movl %1, %%eax;\\n\\r"
                         "movl %%eax, %0;"
                         :"=r"(b)       /* 输出 */
                         :"r"(a)        /* 输入 */
                         :"%eax");      /* 不受影响的寄存器 */

    printf("Result: %d, %d\\n", a, b);
}
```

上面的程序完成将变量a的值赋予变量b，有几点需要说明：

- 变量b是输出操作数，通过%0来引用，而变量a是输入操作数，通过%1来引用。
- 输入操作数和输出操作数都使用r进行约束，表示将变量a和变量b存储在寄存器中。输入约束和输出约束的不同点在于输出约束多一个约束修饰符'='。
- 在内联汇编语句中使用寄存器eax时，寄存器名前应该加两个'%'，即%%eax。内联汇编中使用%0、%1等来标识变量，任何只带一个'%'的标识符都看成是操作数，而不是寄存器。
- 内联汇编语句的最后一个部分告诉GCC它将改变寄存器eax中的值，GCC在处理时不应使用该寄存器来存储任何其它的值。
- 由于变量b被指定成输出操作数，当内联汇编语句执行完毕后，它所保存的值将被更新。

在内联汇编中用到的操作数从输出部的第一个约束开始编号，序号从0开始，每个约束记数一次，指令部要引用这些操作数时，只需在序号前加上'%'作为前缀就可以了。需要注意的是，内联汇编语句的指令部在引用一个操作数时总是将其作为32位的长字使用，但实际情况可能需要的是字或字节，因此应该在约束中指明正确的限定符：

限定符	意义
"m"、"v"、"o"	内存单元
"r"	任何寄存器
"q"	寄存器eax、ebx、ecx、edx之一
"i"、"h"	直接操作数
"E"和"F"	浮点数
"g"	任意
"a"、"b"、"c"、"d"	分别表示寄存器eax、ebx、ecx和edx
"S"和"D"	寄存器esi、edi
"I"	常数（0至31）

八、小结

Linux操作系统是用C语言编写的，汇编只在必要的时候才被人们想到，但它却是减少代码尺寸和优化代码性能的一种非常重要的手段，特别是在与硬件直接交互的时候，汇编可以说是最佳的选择。Linux提供了非常优秀的工具来支持汇编程序的开发，使用GCC的内联汇编能够充分地发挥C语言和汇编语言各自的优点。