Using Assembly Language in Linux.

Intel和AT&T汇编语法差异：

1。前缀：

Intel汇编寄存器和立即数无需前缀。后者寄存器前缀为%，立即数前缀为$。

eg：

   Intex Syntax

   mov eax,1

   mov ebx,0ffh

   int 80h

   AT&T Syntax

   movl $1,%eax

   movl $0xff,%ebx

   int   $0x80

2。二者对操作数处理方向不同：

eg：

   Intel 汇编语法：

   instr dest,source

   mov eax,[ecx]

   AT&T 汇编语法：

   instr   source,dest

   movl (%ecx),%eax

3。内存变量语法差异：

Intel语法使用中括号[]，后者使用小括号()

eg：

   Intex Syntax

   mov eax,[ebx]

   mov eax,[ebx+3]

   AT&T Syntax

   movl (%ebx),%eax

   movl 3(%ebx),%eax

4。后缀：

AT&T汇编指令有后缀，以表明数据类型（8位、16位等）；Intel则根据寄存器自动识别。

eg：

   Intel Syntax

   mov al,bl

   mov ax,bx

   mov eax,ebx

   mov eax, dword ptr [ebx]

   AT&T Syntax

   movb %bl,%al

   movw %bx,%ax

   movl %ebx,%eax

   movl (%ebx),%eax

5。

Intel：segreg:[base+index\*scale+disp]

AT&T： %segreg:disp(base,index,scale).

例子：

       1 #include <stdio.h>

       2

       3 int main(void) {

       4      long eax=4;

       5      long ebx=2;

       6

       7      \_\_asm\_\_ \_\_volatile\_\_ ("addl %1, %0"

       8              : "=b"((long)ebx)

       9              : "a"((long)eax), "b"((long)ebx)

      10              : "1"

      11              );

      12

      13      printf("ebx=%d\n", ebx);

      14      return 0;

      15 }

输出结果：ebx=6

linux下c语言内嵌汇编格式：

\_\_asm\_\_("<asm routine>" : output : input : modify);

output, input, modify都是引号括起来的字符串，他们直接用冒号分隔。

每个都可以有多项，项之间用逗号分隔，最多总共不能超过10个。

output要用“=”开头

寄存器缩写约定

Abbrev Register

a   %eax/%ax/%al

b   %ebx/%bx/%bl

c   %ecx/%cx/%cl

d   %edx/%dx/%dl

S   %esi/%si

D   %edi/%di

m   memory

q 由编译器在a、b、c、d中任意选择

寄存器还可以使用数字形式的缩写，0～9 (%0-%9)，所以会有最多10个项的限制。

系统调用：

1。系统调用号存在eax中

2。系统调用参数要少于6个，分别存放在ebx, ecx, edx, esi, edi

3。返回值存在eax中

4。参数超过5个，用ebx指向存放参数的内存。如果压入堆栈，要注意参数顺序反过来。

汇编文件入口：

如果使用gcc编译，入口为main；如果用as和ld编译连接，入口使用\_start，可以用ld的-e选项指定程序入口：ld -e main -o hello hello.o。

汇编程序运行时出现segmentation fault：

程序运行完时，处理器试图去执行下一个内存单元的指令，内存中没有指令或者为非法指令，硬件或者os包含机制就会发现

这个异常而产生段包含错误。解决方法，加上退出程序的系统调用就可以了。

eg：

       1 #.include "defines.h"

       2 .data

       3 hello:

       4 .string "hello world\n"

       5

       6 .globl   main

       7 main:

       8 #write

       9 movl     $4,%eax

      10 #stdout

      11 movl     $1,%ebx

      12 #content

      13 movl     $hello,%ecx

      14 #length of string

      15 movl     $12,%edx

      16 #system call

      17 int      $0x80

      18

      19 #exit

      20 movl     $1,%eax

      21 int      $0x80

      22

      23 ret

//------------------------------------------------------------------------------------------------------------------------

//  详细说明：

//------------------------------------------------------------------------------------------------------------------------

1.内嵌汇编举例

在内嵌汇编中，可以将C语言表达式指定为汇编指令的操作数，而且不用去管如何将C语言表达式的值读入哪个寄存器，以及如何将计算结果写回C变量，你只要告诉程序中C语言表达式与汇编指令操作数之间的对应关系即可， GCC会自动插入代码完成必要的操作。使用内嵌汇编，要先编写汇编指令模板，然后将C语言表达式与指令的操作数相关联，并告诉GCC对这些操作有哪些限制条件。例如在下面的汇编语句：

\_\_asm\_\_ \_\_violate\_\_(

"movl %1,%0"

: "=r" (result)

: "m" (input)

);

其中“movl %1,%0”是指令模板；“%0”和“%1”代表指令的操作数，称为占位符，内嵌汇编靠它们将C语言表达式与指令操作数相对应。指令模板后面用小括号括起来的是C语言表达式也即我们通常所说的变量，本例中只有两个：“result”和“input”，他们按照出现的顺序分别与指令操作数“%0”，“%1，”对应；注意对应顺序：第一个C表达式对应“%0”；第二个表达式对应“%1”，依次类推，操作数至多有10个，分别用“%0”，“%1”….“%9，”表示。在每个操作数前面有一个用引号括起来的字符串，字符串的内容是对该操作数的限制或者说要求。“result”前面的限制字符串是“=r”，其中“=”表示“result”是输出操作数，“r”表示需要将“result”与某个通用寄存器相关联，先将操作数的值读入寄存器，然后在指令中使用相应寄存器，而不是“result”本身，当然指令执行完后需要将寄存器中的值存入变量“result”，从表面上看好像是指令直接对“result”进行操作，实际上GCC做了隐式处理，这样我们可以少写一些指令。“input”前面的“r”表示该表达式需要先放入某个寄存器，然后在指令中使用该寄存器参加运算。

我们将上面的内嵌代码放到一个C源文件中，然后使用gcc –c–S得到该C文件源代码相对应的汇编代码，然后查看一下汇编代码，看看GCC是如何处理的。

C源文件如下内容如下，注意该代码没有实际意义，仅仅作为例子。

extern int

input,result;

void test(void)

{

input= 1;

\_\_asm\_\_ \_\_volatile\_\_ ("movl %1,%0" :

"=r" (result) : "r" (input));

return;

}

对应的汇编代码如下;

行号          代码                 解释

1

7

8               movl $1, input     对应C语言语句input = 1;

9               movl input, %eax

10 #APP GCC插入的注释，表示内嵌汇编开始

11             movl %eax,%eax       我们的内嵌汇编语句

12 #NO\_APP GCC 插入的注释，表示内嵌汇编结束

13             movl %eax, result      将结果存入result变量

从汇编代码可以看出，第9行和第13行是GCC，自动增加的代码，GCC根据限定字符串决定如何处理C表达式，本例两个表达式都被指定为“r”型，所以先使用指令：

movl input, %eax

将input读入寄存器%eax；GCC，也指定一个寄存器与输出变量result相关，本例也是%eax，等得到操作结果后再使用指令：

movl %eax, result

将寄存器的值写回C变量result中。

从上面的汇编代码我们可以看出与result和input，相关连的寄存器都是%eax，GCC使用%eax，替换内嵌汇编指令模板中的%0，%1 。movl %eax,%eax显然这一句可以不要。但是没有优化，所以这一句没有被去掉。

由此可见，C表达式或者变量与寄存器的关系由GCC自动处理，我们只需使用限制字符串指导GCC如何处理即可。限制字符必须与指令对操作数的要求相匹配，否则产生的汇编代码将会有错，读者可以将上例中的两个“r”，都改为“m”(m，表示操作数放在内存，而不是寄存器中)，编译后得到的结果是：movl input, result  很明显这是一条非法指令，因此限制字符串必须与指令对操作数的要求匹配。例如指令movl允许寄存器到寄存器，立即数到寄存器等，但是不允许内存到内存的操作，因此两个操作数不能同时使用“m”作为限定字符。

2 语法

内嵌汇编语法如下：

\_\_asm\_\_(

汇编语句模板:

输出部分:

输入部分:

破坏描述部分)

共四个部分：汇编语句模板，输出部分，输入部分，破坏描述部分，各部分使用“:”格开，汇编语句模板必不可少，其他三部分可选，如果使用了后面的部分，而前面部分为空，也需要用“:”格开，相应部分内容为空。例如：

\_\_asm\_\_ \_\_volatile\_\_(

"CLI":

:"memory")

2.1 汇编语句模板

汇编语句模板由汇编语句序列组成，语句之间使用“;”、“\n”或“\n\t”分开。指令中的操作数可以使用占位符引用C语言变量，操作数占位符最多10个，名称如下：%0，%1…，%9。指令中使用占位符表示的操作数，总被视为long型（4，个字节），但对其施加的操作根据指令可以是字或者字节，当把操作数当作字或者字节使用时，默认为低字或者低字节。对字节操作可以显式的指明是低字节还是次字节。方法是在%和序号之间插入一个字母，“b”代表低字节，“h”代表高字节，例如：%h1。

2.2 输出部分

输出部分描述输出操作数，不同的操作数描述符之间用逗号格开，每个操作数描述符由限定字符串和C语言变量组成。每个输出操作数的限定字符串必须包含“=”表示他是一个输出操作数。

例：

\_\_asm\_\_ \_\_volatile\_\_(

"pushfl ;

 popl %0 ;

 cli"

:"=g" (x)

 )

描述符字符串表示对该变量的限制条件，这样GCC就可以根据这些条件决定如何分配寄存器，如何产生必要的代码处理指令操作数与C表达式或C变量之间的联系。

2.3 输入部分

输入部分描述输入操作数，不同的操作数描述符之间使用逗号格开，每个操作数描述符由限定字符串和C语言表达式或者C语言变量组成。

例1：

\_\_asm\_\_ \_\_volatile\_\_ (

"lidt %0"

 :

 : "m" (real\_mode\_idt)

);

例二（bitops.h）：

Static \_\_inline\_\_ void \_\_set\_bit(int nr, volatile void \* addr)

{

\_\_asm\_\_(

"btsl%1,%0" :

"=m"(ADDR) :

"Ir"(nr));

}

后例功能是将(\*addr)的第nr位设为1。第一个占位符%0与C语言变量ADDR对应，第二个占位符%1与C语言变量nr对应。因此上面的汇编语句代码与下面的伪代码等价：

btsl nr, ADDR，

该指令的两个操作数不能全是内存变量，因此将nr的限定字符串指定为“Ir”，将nr，与立即数或者寄存器相关联，这样两个操作数中只有ADDR为内存变量。

2.4 限制字符

2.4.1 限制字符列表

限制字符有很多种，有些是与特定体系结构相关，此处仅列出常用的限定字符和i386中可能用到的一些常用的限定符。它们的作用是指示编译器如何处理其后的C语言变量与指令操作数之间的关系，例如是将变量放在寄存器中还是放在内存中等，下表列出了常用的限定字母。

分类 限定符 描述 通用寄存器

“a”将输入变量放入eax

这里有一个问题：假设eax已经被使用，那怎么办？

其实很简单：因为GCC知道eax已经被使用，它在这段汇编代码的起始处插入一条语句pushl %eax，将eax内容保存到堆栈，然后在这段代码结束处再增加一条语句popl %eax，恢复eax的内容

“b”将输入变量放入ebx

“c”将输入变量放入ecx

“d”将输入变量放入edx

“s”将输入变量放入esi

“d”将输入变量放入edi

“q”将输入变量放入eax，ebx ,ecx ，edx中的一个

“r”将输入变量放入通用寄存器，也就是eax ，ebx，ecx,edx，esi，edi中的一个

“A”把eax和edx，合成一个64位的寄存器(uselong longs)

“m”内存变量

“o”操作数为内存变量，但是其寻址方式是偏移量类型，也即是基址寻址，或者是基址加变址寻址

“V”操作数为内存变量，但寻址方式不是偏移量类型

“,” 操作数为内存变量，但寻址方式为自动增量

“p”操作数是一个合法的内存地址（指针）

寄存器或内存

“g” 将输入变量放入eax，ebx，ecx ，edx中的一个或者作为内存变量

“X”操作数可以是任何类型

立即数

“I” 0-31 之间的立即数（用于32位移位指令）

“J” 0-63 之间的立即数（用于64 位移位指令）

“N” 0-255 ，之间的立即数（用于out 指令）

“i” 立即数

“n” 立即数，有些系统不支持除字以外的立即数，这些系统应该使用“n”而不是“i”

匹配

“0”，“1 ，”... “9 ”

表示用它限制的操作数与某个指定的操作数匹配，也即该操作数就是指定的那个操作数，例如用“0 ”去描述“％1”操作数，那么“%1”引用的其实就是“%0”操作数，注意作为限定符字母的0－9 ，与指令中的“％0”－“％9”的区别，前者描述操作数，后者代表操作数。

后面有详细描述 & 该输出操作数不能使用过和输入操作数相同的寄存器后面有详细描述

操作数类型

“=” 操作数在指令中是只写的（输出操作数）

“+” 操作数在指令中是读写类型的（输入输出操作数）

浮点数

“f”浮点寄存器“t”第一个浮点寄存器“u”第二个浮点寄存器“G”标准的80387浮点常数% 该操作数可以和下一个操作数交换位置例如addl的两个操作数可以交换顺序（当然两个操作数都不能是立即数）# 部分注释，从该字符到其后的逗号之间所有字母被忽略\* 表示如果选用寄存器，则其后的字母被忽略现在继续看上面的例子，"=m" (ADDR)表示ADDR为内存变量（“m”），而且是输出变量（“=”）；"Ir" (nr)表示nr，为0－31之间的立即数（“I”）或者一个寄存器操作数（“r”）。

2.4.2 匹配限制符

I386

指令集中许多指令的操作数是读写型的（读写型操作数指先读取原来的值然后参加运算，最后将结果写回操作数），例如addl %1,%0，它的作用是将操作数%0与操作数%1的和存入操作数%0，因此操作数%0是读写型操作数。老版本的GCC对这种类型操作数的支持不是很好，它将操作数严格分为输入和输出两种，分别放在输入部分和输出部分，而没有一个单独部分描述读写型操作数，因此在GCC中读写型的操作数需要在输入和输出部分分别描述，靠匹配限制符将两者关联到一起注意仅在输入和输出部分使用相同的C变量，但是不用匹配限制符，产生的代码很可能不对，后面会分析原因。匹配限制符是一位数字：“0”、“1”……“9，”，分别表示它限制的C表达式分别与占位符%0，%1，……%9对应的C变量匹配.

例如使用“0”作为%1，的限制字符，那么%0和％1表示同一个C，变量。看一下下面的代码就知道为什么要将读写型操作数，分别在输入和输出部分加以描述。该例功能是求input+result的和，然后存入result：

extern int input,result;

void test\_at\_t()

{

result= 0;

input = 1;

\_\_asm\_\_

\_\_volatile\_\_ ("addl %1,%0":"=r"(result): "r"(input));

}

对应的汇编代码为：

movl $0,\_result

movl $1,\_input

movl \_input,%edx /APP

addl %edx,%eax /NO\_APP

movl %eax,%edx

movl %edx,\_result

input 为输入型变量，而且需要放在寄存器中，GCC给它分配的寄存器是%edx，在执行addl之前%edx，的内容已经是input的值。可见对于使用“r”限制的输入型变量或者表达式，在使用之前GCC会插入必要的代码将他们的值读到寄存器；“m”型变量则不需要这一步。读入input后执行addl，显然%eax的值不对，需要先读入result的值才行。再往后看：movl %eax,%edx和movl %edx,\_result的作用是将结果存回result，分配给result的寄存器与分配给input的一样，都是%edx。

综上可以总结出如下几点：

1. 使用“r”限制的输入变量，GCC先分配一个寄存器，然后将值读入寄存器，最后用该寄存器替换占位符；

2. 使用“r”限制的输出变量，GCC会分配一个寄存器，然后用该寄存器替换占位符，但是在使用该寄存器之前并不将变量值先读入寄存器，GCC认为所有输出变量以前的值都没有用处，不读入寄存器（可能是因为AT&T汇编源于CISC架构处理器的汇编语言，在CISC处理器中大部分指令的输入输出明显分开，而不像RISC那样一个操作数既做输入又做输出，例如add r0,r1,r2，r0，和r1是输入，r2是输出，输入和输出分开，没有使用输入输出型操作数，这样我们就可以认为r2对应的操作数原来的值没有用处，也就没有必要先将操作数的值读入r2，因为这是浪费处理器的CPU周期），最后GCC插入代码，将寄存器的值写回变量；

3. 输入变量使用的寄存器在最后一处使用它的指令之后，就可以挪做其他用处，因为已经不再使用。例如上例中的%edx。在执行完addl之后就作为与result对应的寄存器。

因为第二条，上面的内嵌汇编指令不能奏效，因此需要在执行addl之前把result的值读入寄存器，也许再将result放入输入部分就可以了（因为第一条会保证将result先读入寄存器）。修改后的指令如下（为了更容易说明问题将input限制符由“r，”改为“m”）：

extern int input,result;

void test\_at\_t()

{

result = 0;

input = 1;

\_\_asm\_\_

\_\_volatile\_\_ ("addl %2,%0":"=r"(result):"r"(result),"m"(input));

}

看上去上面的代码可以正常工作，因为我们知道%0和%1都和result相关，应该使用同一个寄存器，但是GCC并不去判断%0和%1，是否和同一个C表达式或变量相关联（这样易于产生与内嵌汇编相应的汇编代码），因此%0和%1使用的寄存器可能不同。我们看一下汇编代码就知道了。

movl $0,\_result

movl $1,\_input

movl \_result,%edx /APP

addl \_input,%eax /NO\_APP

movl %eax,%edx

movl %edx,\_result

现在在执行addl之前将result的值被读入了寄存器%edx，但是addl指令的操作数%0却成了%eax，而不是%edx，与预料的不同，这是因为GCC给输出和输入部分的变量分配了不同的寄存器，GCC没有去判断两者是否都与result相关，后面会讲GCC如何翻译内嵌汇编，看完之后就不会惊奇啦。

使用匹配限制符后，GCC知道应将对应的操作数放在同一个位置（同一个寄存器或者同一个内存变量）。使用匹配限制字符的代码如下：

extern int input,result;

void test\_at\_t()

{

result = 0;

input = 1;

\_\_asm\_\_

\_\_volatile\_\_ ("addl %2,%0":"=r"(result):"0"(result),"m"(input));

}

输入部分中的result用匹配限制符“0”限制，表示%1与％0，代表同一个变量，输入部分说明该变量的输入功能，输出部分说明该变量的输出功能，两者结合表示result是读写型。因为%0和%1，表示同一个C变量，所以放在相同的位置，无论是寄存器还是内存。

相应的汇编代码为：

movl $0,\_result

movl $1,\_input

movl \_result,%edx

movl %edx,%eax /APP

addl \_input,%eax /NO\_APP

movl %eax,%edx

movl %edx,\_result

可以看到与result相关的寄存器是%edx，在执行指令addl之前先从%edx将result读入%eax，执行之后需要将结果从%eax读入%edx，最后存入result中。这里我们可以看出GCC处理内嵌汇编中输出操作数的一点点信息：addl并没有使用%edx，可见它不是简单的用result对应的寄存器%edx去替换%0，而是先分配一个寄存器，执行运算，最后才将运算结果存入对应的变量，因此GCC是先看该占位符对应的变量的限制符，发现是一个输出型寄存器变量，就为它分配一个寄存器，此时没有去管对应的C变量，最后GCC，知道还要将寄存器的值写回变量，与此同时，它发现该变量与%edx关联，因此先存入%edx，再存入变量。

至此读者应该明白了匹配限制符的意义和用法。在新版本的GCC中增加了一个限制字符“+”，它表示操作数是读写型的，GCC知道应将变量值先读入寄存器，然后计算，最后写回变量，而无需在输入部分再去描述该变量。

例;

extern int input,result;

void test\_at\_t()

{

result = 0;

input = 1;

\_\_asm\_\_

\_\_volatile\_\_ ("addl %1,%0":"+r"(result):"m"(input));

}

此处用“+”替换了“=”，而且去掉了输入部分关于result的描述，产生的汇编代码如下：

movl $0,\_result

movl $1,\_input

movl \_result,%eax /APP

addl \_input,%eax /NO\_APP

movl %eax,\_result

L2:

movl %ebp,%esp

处理的比使用匹配限制符的情况还要好，省去了好几条汇编代码。

2.4.3 “&”限制符

限制符“&”在内核中使用的比较多，它表示输入和输出操作数不能使用相同的寄存器，这样可以避免很多错误。

举一个例子，下面代码的作用是将函数foo的返回值存入变量ret中

\_\_asm\_\_ ( “call foo;movl %%edx,%1”, :”=a”(ret) : ”r”(bar) );

我们知道函数的int型返回值存放在%eax中，但是gcc编译的结果是输入和输出同时使用了寄存器%eax，如下：

movl bar, %eax

#APP

call foo

movl %ebx,%eax

#NO\_APP

movl %eax, ret

结果显然不对，原因是GCC并不知道%eax中的值是我们所要的。避免这种情况的方法是使用“&”限定符，这样bar就不会再使用%eax寄存器，因为已被ret指定使用。

\_asm\_\_ ( “call foo;movl %%edx,%1”,:”=&a”(ret) : ”r”(bar) );