昵称：taek
园龄：3年8个月
粉丝：6
关注：2
+加关注

< 2012年2月 >

日	一	二	三	四	五	六
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	1	2	3
4	5	6	7	8	9	10

搜索

找找看
谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

随笔分类

apache(1)
ASM(11)
ecshop(1)
free-software(1)
html(1)
JS(3)
PHP(17)
ubuntu(1)

随笔档案

汇编语言---GCC内联汇编

GCC支持在C/C++代码中嵌入汇编代码,这些代码被称作是"GCC Inline ASM"(GCC内联汇编);

一、基本内联汇编

GCC中基本的内联汇编非常易懂,格式如下:

```
__asm__ [__volatile__] ("instruction list");
```

其中,

1.__asm__:

它是GCC定义的关键字asm的宏定义(#define __asm__ asm),它用来声明一个内联汇编表达式,所以,任何一个内联汇编表达式都以它开头,它是必不可少的;如果要编写符合ANSI C标准的代码(即:与ANSI C兼容),那就要使用__asm__;

2.__volatile__:

它是GCC关键字volatile的宏定义;这个选项是可选的;它向GCC声明"不要动我所写的instruction list,我需要原封不动地保留每一条指令";如果不使用__volatile__,则当你使用了优化选项-O进行优化编译时,GCC将会根据自己的判断来决定是否将这个内联汇编表达式中的指令优化掉;如果要编写符合ANSI C标准的代码(即:与ANSI C兼容),那就要使用__volatile__;

3.instruction list:

它是汇编指令列表;它可以是空列表,比如:__asm__ __volatile__("");或__asm__("");都是合法的内联汇编表达式,只不过这两条语句什么都不做,没有什么意义;但并非所有"instruction list"为空的内联汇编表达式都是没意义的,比如:__asm__("":::"memory");就是非常有意义的,它向GCC声明:"我对内存做了改动",这样,GCC在编译的时候,就会将此因素考虑进去;

例如:

```
__asm__("movl %esp,%eax");
```

或者是

```
__asm__("movl $1,%eax
         xor %ebx,%ebx
         int $0x80");
```

或者是

```
__asm__("movl $1,%eax\n\t"\
        "xor %ebx,%ebx\n\t"\
        "int $0x80");
```

instruction list的编写规则:当指令列表里面有多条指令时,可以在一对双引号中全部写出,也可将一条或多条指令放在一对双引号中,所有指令放在多对双引号中;如果是将所有指令写在一对双引号中,那么,相邻俩条指令之间必须用分号";"或换行符(\n)隔开,如果使用换行符(\n),通常\n后面还要跟一个\t;或者是相邻两条指令分别单独写在两行中;

规则1:任意两条指令之间要么被分号(;)或换行符(\n)或(\n\t)分隔开,要么单独放在两行;

规则2:单独放在两行的方法既可以通过\n或\n\t的方法来实现,也可以真正地放在两行;
规则3:可以使用1对或多对双引号,每1对双引号里面可以放1条或多条指令,所有的指令都要放在双引号中;
例如,下面的内联汇编语句都是合法的:

```
__asm__("movl %eax,%ebx
        sti
        popl %edi
        subl %ecx,%ebx");
__asm__("movl %eax,%ebx; sti
         popl %edi; subl %ecx,%ebx");
__asm__("movl %eax,%ebx; sti\n\t popl %edi
         subl %ecx,%ebx");
```

如果将指令放在多对双引号中,则,除了最后一对双引号之外,前面的所有双引号里的最后一条指令后面都要有一个分号(;)或(\n)或(\n\t);比如,下面的内联汇编语句都是合法的:

```
__asm__("movl %eax,%ebx
         sti\n"
         "popl %edi;"
         "subl %ecx,%bx");
__asm__("movl %eax,%ebx; sti\n\t"
        "popl %edi; subl %ecx,%ebx");
__asm__("movl %eax,%ebx; sti\n\t popl %edi\n"
        "subl %ecx,%ebx");
```

二、带有C/C++表达式的内联汇编

GCC允许你通过C/C++表达式指定内联汇编中"instruction list"中的指令的输入和输出,你甚至可以不关心到底使用哪些寄存器,完全依靠GCC来安排和指定;这一点可以让程序员免去考虑有限的寄存器的使用,也可以提高目标代码的效率;

1.带有C/C++表达式的内联汇编语句的格式:

```
__asm__ [__volatile__]("instruction list":Output:Input:Clobber/Modify);
```

圆括号中的内容被冒号":"分为四个部分:
A.如果第四部分的"Clobber/Modify"可以为空;如果"Clobber/Modify"为空,则其前面的冒号(:)必须省略;比如:语句__asm__("movl %%eax,%%ebx":"=b"(foo):"a"(inp):);是非法的,而语句__asm__("movl %%eax,%%ebx":"=b"(foo):"a"(inp));则是合法的;
B.如果第一部分的"instruction list"为空,则input、output、Clobber/Modify可以为空,也可以不为空;比如,语句__asm__("":::"memory");和语句__asm__("":);都是合法的写法;
C.如果Output、Input和Clobber/Modify都为空,那么,Output、Input之前的冒号(:)可以省略,也可以不省略;如果都省略,则此汇编就退化为一个基本汇编,否则,仍然是一个带有C/C++表达式的内联汇编,此时"instruction list"中的寄存器的写法要遵循相关规定,比如:寄存器名称前面必须使用两个百分号(%%);基本内联汇编中的寄存器名称前面只有一个百分号(%);比如,语句__asm__("movl %%eax,%%ebx"::);__asm__("movl %%eax,%%ebx":);和语句__asm__("movl %%eax,%%ebx");都是正确的写法,而语句__asm__("movl %eax,%ebx"::);__asm__("movl %eax,%ebx":);和语句__asm__("movl %%eax,%%ebx");都是错误的写法;
D.如果Input、Clobber/Modify为空,但Output不为空,则,Input前面的冒号(:)可以省略,也可以不省略;比如,语句__asm__("movl %%eax,%%ebx":"=b"(foo):);和语句__asm__("movl %%eax,%%ebx":"=b"(foo));都是正确的;
E.如果后面的部分不为空,而前面的部分为空,则,前面的冒号(:)都必须保留,否则无法说明不为空的部分究竟是第几部分;比如,Clobber/Modify、Output为空,而Input不为空,则Clobber/Modify前面的冒号必须省略,而Output前面的冒号必须保留;如果Clobber/Modify不为空,而Input和Output都为空,则Input和Output前面的冒号都必须保留;比如,语句__asm__("movl %%eax,%%ebx"::"a"(foo));和__asm__("movl %%eax,%%ebx":::"ebx");
注意:基本内联汇编中的寄存器名称前面只能有一个百分号(%),而带有C/C++表达式的内联汇编中的寄存器名臣前面必须有两个百分号(%%);

2.Output:

Output部分用来指定当前内联汇编语句的输出,称为输出表达式;

aa

http://code.google.com/p/stblog/downloads/detail?name=stblog-0.1.2.zip

最新评论

1. Re:c malloc分配内存
@ohmygirl 你说的对，当时是我写错了，你看下这篇文章，说的挺详细的...
--taek
2. Re:c malloc分配内存
2）数据段：定义的全局变量和静态变量 3）BSS段：未定义的全局变量和静态变量----------似乎有点错误。数据段和BSS段不都是存储定义的全局变量和静态变量.....
--ohmygirl
3. Re:以div代替frameset，用css实现仿框架布局
彼此彼此，前端很复杂的特效写不了了
--声音的旋律
4. Re:以div代替frameset，用css实现仿框架布局
@声音的旋律这个我不会呀，这个帖子是好几年前的了，现在我不怎么写前端页面了...
--taek
5. Re:以div代替frameset，用css实现仿框架布局
怎么实现左侧链接在main右页面打开啊亲
--声音的旋律

阅读排行榜

格式为: "操作约束"(输出表达式)
例如:
__asm__("movl %%rc0,%1":"=a"(cr0));
这个语句中的Output部分就是("=a"(cr0)),它是一个操作表达式,指定了一个内联汇编语句的输出部分;
Output部分由两个部分组成:由双引号括起来的部分和由圆括号括起来的部分,这两个部分是一个Output部分所不可缺少的部分;
用双引号括起来的部分就是C/C++表达式,它用于保存当前内联汇编语句的一个输出值,其操作就是C/C++赋值语句"="的左值部分,因此,圆括号中指定的表达式只能是C/C++中赋值语句的左值表达式,即:放在等号=左边的表达式;也就是说,Output部分只能作为C/C++赋值操作左边的表达式使用;
用双引号括起来的部分就指定了C/C++中赋值表达式的右值来源;这个部分被称作是"操作约束"(Operation Constraint),也可以称为"输出约束";在这个例子中的操作约束是"=a",这个操作约束包含两个组成部分:等号(=)和字母a,其中,等号(=)说明圆括号中的表达式cr0是一个只写的表达式,只能被用作当前内联汇编语句的输出,而不能作为输入;字母a是寄存器EAX/AX/AL的缩写,说明cr0的值要从寄存器EAX中获取,也就是说cr0=eax,最终这一点被转化成汇编指令就是:movl %eax,address_of_cr0;
注意:很多文档中都声明,所有输出操作的的操作约束都必须包含一个等号(=),但是GCC的文档中却明确地声明,并非如此;因为等号(=)约束说明当前的表达式是一个只写的,但是还有另外一个符号:加号(+),也可以用来说明当前表达式是可读可写的;如果一个操作约束中没有给出这两个符号中的任何一个,则说明当前表达式是只读的;因此,对于输出操作来说,肯定必须是可写的,而等号(=)和加号(+)都可表示可写,只不过加号(+)同时也可以表示可读;所以,对于一个输出操作来说,其操作约束中只要包含等号(=)或加号(+)中的任意一个就可以了;
等号(=)与加号(+)的区别:等号(=)表示当前表达式是一个纯粹的输出操作,而加号(+)则表示当前表达式不仅仅是一个输出操作,还是一个输入操作;但无论是等号(=)还是加号(+),所表示的都是可写,只能用于输出,只能出现在Output部分,而不能出现在Input部分;
在Output部分可以出现多个输出操作表达式,多个输出操作表达式之间必须用逗号(,)隔开;

3、Input:

Input部分用来指定当前内联汇编语句的输入;称为输入表达式;
格式为: "操作约束"(输入表达式)
例如:
__asm__("movl %0,%%db7"::"a"(cpu->db7));
其中,表达式"a"(cpu->db7)就称为输入表达式,用于表示一个对当前内联汇编的输入;
Input同样也由两部分组成:由双引号括起来的部分和由圆括号括起来的部分;这两个部分对于当前内联汇编语句的输入来说也是必不可少的;
在这个例子中,由双引号括起来的部分是"a",用圆括号括起来的部分是(cpu->db7);
用双引号括起来的部分就是C/C++表达式,它为当前内联汇编语句提供一个输入值;在这里,圆括号中的表达式cpu->db7是一个C/C++语言的表达式,它不必是左值表达式,也就是说,它不仅可以是放在C/C++赋值操作左边的表达式,还可以是放在C/C++赋值操作右边的表达式;所以,Input可以是一个变量、一个数字,还可以是一个复杂的表达式(如:a+b/c*d);
比如,上例还可以这样写:
__asm__("movl %0,%%db7"::"a"(foo));__asm__("movl %0,%%db7"::"a"(0x12345));__asm__("movl %0,%%db7"::"a"(va:vb/vc));
用双引号括起来的部分就是C/C++中赋值表达式的右值表达式,用于约束当前内联汇编语句中的当前输入;这个部分也成为"操作约束",也可以成为是"输入约束";
与输出表达式中的操作约束不同的是,输入表达式中的操作约束不允许指定等号(=)约束或加号(+)约束,也就是说,它只能是只读的;约束中必须指定一个寄存器约束;例子中的字母a表示当前输入变量cpu->db7要通过寄存器EAX输入到当前内联汇编语句中;

三、操作约束:Operation Constraint

操作约束只会出现在带有C/C++表达式的内联汇编语句中;
每一个Input和Output表达式都必须指定自己的操作约束Operation Constraint;约束的类型有:寄存器约束、内存约束、立即数约束、通用约束;
操作表达式的格式:
"约束"(C/C++表达式)
即:"Constraint"(C/C++ expression)

1.寄存器约束:

当你的输入或输出需要借助于一个寄存器时,你需要为其指定一个寄存器约束;
可以直接指定一个寄存器名字;比如:

__asm__ __volatile__("movl %0,%%cr0"::"eax"(cr0));
也可以指定寄存器的缩写名称;比如:
__asm__ __volatile__("movl %0,%%cr0"::"a"(cr0));
如果指定的是寄存器的缩写名称,比如:字母a;那么,GCC将会根据当前操作表达式中C/C++表达式的宽度来决定使用%eax、%ax还是%al;比如:
unsigned short __shrt;
__asm__ __volatile__("movl %0,%%bx"::"a"(__shrt));
由于变量__shrt是16位无符号类型m大小是两个字节,所以,编译器编译出来的汇编代码中,则会让此变量使用寄存器%ax;
无论是Input还是Output操作约束,都可以使用寄存器约束;
常用的寄存器约束的缩写:
r:I/O,表示使用一个通用寄存器,由GCC在%eax/%ax/%al、%ebx/%bx/%bl、%ecx/%cx/%cl、%edx/%dx/%dl中选取一个GCC认为是合适的;
q:I/O,表示使用一个通用寄存器,与r的意义相同;
g:I/O,表示使用寄存器或内存地址;
m:I/O,表示使用内存地址;
a:I/O,表示使用%eax/%ax/%al;
b:I/O,表示使用%ebx/%bx/%bl;
c:I/O,表示使用%ecx/%cx/%cl;
d:I/O,表示使用%edx/%dx/%dl;
D:I/O,表示使用%edi/%di;
S:I/O,表示使用%esi/%si;
f:I/O,表示使用浮点寄存器;
t:I/O,表示使用第一个浮点寄存器;
u:I/O,表示使用第二个浮点寄存器;
A:I/O,表示把%eax与%edx组合成一个64位的整数值;
o:I/O,表示使用一个内存位置的偏移量;
V:I/O,表示仅仅使用一个直接内存位置;
i:I/O,表示使用一个整数类型的立即数;
n:I/O,表示使用一个带有已知整数值的立即数;
F:I/O,表示使用一个浮点类型的立即数;
2.内存约束:
如果一个Input/Output操作表达式的C/C++表达式表现为一个内存地址(指针变量),不想借助于任何寄存器,则可以使用内存约束;比如:
__asm__("lidt %0":"=m"(__idt_addr));或__asm__("lidt %0"::"m"(__idt_addr));
内存约束使用约束名"m",表示的是使用系统支持的任何一种内存方式,不需要借助于寄存器;
使用内存约束方式进行输入输出时,由于不借助于寄存器,所以,GCC不会按照你的声明对其做任何的输入输出处理;GCC只会直接拿来使用,对这个C/C++表达式而言,究竟是输入还是输出,完全依赖于你写在"instruction list"中的指令对其操作的方式;所以,不管你把操作约束和操作表达式放在Input部分还是放在Output部分,GCC编译生成的汇编代码都是一样的,程序的执行结果也都是正确的;本来我们将一个操作表达式放在Input或Output部分是希望GCC能为我们自动通过寄存器将表达式的值输入或输出;既然对于内存约束类型的操作表达式来说,GCC不会为它做任何事情,那么放在哪里就无所谓了;但是从程序员的角度来看,为了增强代码的可读性,最好能够把它放在符合实际情况的地方;
3.立即数约束:
如果一个Input/Output操作表达式的C/C++表达式是一个数字常数,不想借助于任何寄存器或内存,则可以使用立即数约束;
由于立即数在C/C++表达式中只能作为右值使用,所以,对于使用立即数约束的表达式而言,只能放在Input部分;比如:
__asm__ __volatile__("movl %0,%%eax"::"i"(100));
立即数约束使用约束名"i"表示输入表达式是一个整数类型的立即数,不需要借助于任何寄存器,只能用于Input部分;使用约束名"F"表示输入表达式是一个浮点数类型的立即数,不需要借助于任何寄存器,只能用于Input部分;

4.通用约束:

约束名"g"可以用于输入和输出,表示可以使用通用寄存器、内存、立即数等任何一种处理方式;

约束名"0,1,2,3,4,5,6,7,8,9"只能用于输入,表示与第n个操作表达式使用相同的寄存器/内存;

通用约束"g"是一个非常灵活的约束,当程序员认为一个C/C++表达式在实际操作中,无论使用寄存器方式、内存方式还是立即数方式都无所谓时,或者程序员想实现一个灵活的模板,以让GCC可以根据不同的C/C++表达式生成不同的访问方式时,就可以使用通用约束g;

例如:

```
#define JUST_MOV(foo) __asm__("movl %0,%%eax"::"g"(foo))
```

则,JUST_MOV(100)和JUST_MOV(var)就会让编译器产生不同的汇编代码;

对于JUST_MOV(100)的汇编代码为:

```
#APP
 movl $100,%eax       #立即数方式;
#NO_APP
```

对于JUST_MOV(var)的汇编代码为:

```
#APP
 movl 8(%ebp),%eax    #内存方式;
#NO_APP
```

像这样的效果,就是通用约束g的作用;

5.修饰符:

等号(=)和加号(+)作为修饰符,只能用于Output部分;等号(=)表示当前输出表达式的属性为只写,加号(+)表示当前输出表达式的属性为可读可写;这两个修饰符用于约束对输出表达式的操作,它们俩被写在输出表达式的约束部分中,并且只能写在第一个字符的位置;

符号&也写在输出表达式的约束部分,用于约束寄存器的分配,但是只能写在约束部分的第二个字符的位置上;

用符号&进行修饰时,等于向GCC声明:"GCC不得为任何Input操作表达式分配与此Output操作表达式相同的寄存器";

其原因是修饰符&意味着被其修饰的Output操作表达式要在所有的Input操作表达式被输入之前输出;

即:GCC会先使用输出值对被修饰符&修饰的Output操作表达式进行填充,然后,才对Input操作表达式进行输入;

这样的话,如果不使用修饰符&对Output操作表达式进行修饰,一旦后面的Input操作表达式使用了与Output操作表达式相同的寄存器,就会产生输入输出数据混乱的情况;相反,如果没有用修饰符&修饰输出操作表达式,那么,就意味着GCC会先把Input操作表达式的值输入到选定的寄存器中,然后经过处理,最后才用输出值填充对应的Output操作表达式;

所以,修饰符&的作用就是要求GCC编译器为所有的Input操作表达式分配别的寄存器,而不会分配与被修饰符&修饰的Output操作表达式相同的寄存器;修饰符&也写在操作约束中,即:&约束;由于GCC已经规定加号(+)或等号(=)占据约束的第一个字符,那么&约束只能占用第二个字符;

例如:

```
int __out, __in1, __in2;
__asm__("popl %0\n\t"
        "movl %1,%%esi\n\t"
        "movl %2,%%edi\n\t"
        :"=&a"(__out)
        :"r"(__in1),"r"(__in2));
```

注意:如果一个Output操作表达式的寄存器约束被指定为某个寄存器,只有当至少存在一个Input操作表达式的寄存器约束为可选约束(意思是GCC可以从多个寄存器中选取一个,或使用非寄存器方式)时,比如"r"或"g"时,此Output操作表达式使用符号&修饰才有意义;如果你为所有的Input操作表达式指定了固定的寄存器,或使用内存/立即数约束时,则此Output操作表达式使用符号&修饰没有任何意义;

比如:

```
__asm__("popl %0\n\t"
        "movl %1,%esi\n\t"
        "movl %2,%edi\n\t"
        :"=&a"(__out)
        :"m"(__in1),"c"(__in2));
```

此例中的Output操作表达式完全没有必要使用符号&来修饰,因为__in1和__in2都已经被指定了固定的寄存器,或使用了内存方式,GCC无从选择;

如果你已经为某个Output操作表达式指定了修饰符&,并指定了固定的寄存器,那么,就不能再为任何Input操作表达式指定这个寄存器了,否则会出现编译报错;

比如:

```
__asm__("popl %0; movl %1,%%esi; movl %2,%%edi;":"=&a"(__out):"a"(__in1),"c"(__in2));
```

对这条语句的编译就会报错;

相反,你也可以为Output指定可选约束,比如"r"或"g"等,让GCC为此Output操作表达式选择合适的寄存器,或使用内存方式,GCC在选择的时候,会排除掉已经被Input操作表达式所使用过的所有寄存器,然后在剩下的寄存器中选择,或者干脆使用内存方式;

比如:

```
__asm__("popl %0; movl %1,%%esi; movl %2,%%edi;":"=&r"(__out):"a"(__in1),"c"(__in2));
```

这三个修饰符只能用在Output操作表达式中,而修饰符%则恰恰相反,它只能用在Input操作表达式中;

修饰符%用于向GCC声明:"当前Input操作表达式中的C/C++表达式可以与下一个Input操作表达式中的C/C++表达式互换";这个修饰符一般用于符合交换律运算的地方;比如:加、乘、按位与&、按位或|等等;

例如:

```
__asm__("addl %1,%0\n\t":"=r"(__out):"%r"(__in1),"0"(__in2));
```

其中,"0"(__in2)表示使用与第一个Input操作表达式("r"(__in1))相同的寄存器或内存;

由于使用符号%修饰__in1的寄存器方式r,那么就表示,__in1与__in2可以互换位置;加法的两个操作数交换位置之后,和不变;

修饰符	I/O	意义
=	O	表示此Output操作表达式是只写的
+	O	表示此Output操作表达式是可读可写的
&	O	表示此Output操作表达式独占为其指定的寄存器
%	I	表示此Input操作表达式中的C/C++表达式可以与下一个Input操作表达式中的C/C++表达式互换

四、占位符

每一个占位符对应一个Input/Output操作表达式;

带C/C++表达式的内联汇编中有两种占位符:序号占位符和名称占位符;

1.序号占位符:

GCC规定:一个内联汇编语句中最多只能有10个Input/Output操作表达式,这些操作表达式按照他们被列出来的顺序依次赋予编号0到9;对于占位符中的数字而言,与这些编号是对应的;比如:占位符%0对应编号为0的操作表达式,占位符%1对应编号为1的操作表达式,依次类推;

由于占位符前面要有一个百分号%,为了去边占位符与寄存器,GCC规定:在带有C/C++表达式的内联汇编语句的指令列表里列出的寄存器名称前面必须使用两个百分号(%%),一区别于占位符语法;

GCC对占位符进行编译的时候,会将每一个占位符替换为对应的Input/Output操作表达式所指定的寄存器/内存/立即数;

例如:

```
__asm__("addl %1,%0\n\t":"=a"(__out):"m"(__in1),"a"(__in2));
```

这个语句中,%0对应Output操作表达式"=a"(__out),而"=a"(__out)指定的寄存器是%eax,所以,占位符%0被替换为%eax;占位符%1对应Input操作表达式"m"(__in1),而"m"(__in1)被指定为内存,所以,占位符%1被替换位__in1的内存地址;

用一句话描述:序号占位符就是前面描述的%0、%1、%2、%3、%4、%5、%6、%7、%8、%9;其中,每一个占位符对应一个Input/Output的C/C++表达式;

2.名称占位符:

由于GCC中限制这种占位符的个数最多只能由这10个,这也就限制了Input/Output操作表达式中C/C++表达式的数量做多只能有10个;如果需要的C/C++表达式的数量超过10个,那么,这些需要占位符就不够用了;

GCC内联汇编提供了名称占位符来解决这个问题;即:使用一个名字字符串与一个C/C++表达式对应;这个名字字符串就称为名称占位符;而这个名字通常使用与C/C++表达式中的变量完全相同的名字;

使用名字占位符时,内联汇编的Input/Output操作表达式中的C/C++表达式的格式如下:

[name] "constraint"(变量)

此时,指令列表中的占位符的书写格式如下:

%[name]

这个格式等价于序号占位符中的%0,%1,$2等等;

使用名称占位符时,一个name对应一个变量;

例如:

```
__asm__("imull %[value1],%[value2]"
        :[value2] "=r"(data2)
        :[value1] "r"(data1),"0"(data2));
```

此例中,名称占位符value1就对应变量data1,名称占位符value2对应变量data2;GCC编译的时候,同样会把这两个占位符分别替换成对应的变量所使用的寄存器/内存地址/立即数;而且也增强了代码的可读性;

这个例子,使用序号占位符的写法如下:

```
__asm__("imull %1,%0"
        :"=r"(data2)
        :"r"(data1),"0"(data2));
```

五、寄存器/内存修改标示(Clobber/Modify)

有时候,当你想通知GCC当前内联汇编语句可能会对某些寄存器或内存进行修改,希望GCC在编译时能够将这一点考虑进去;那么你就可以在Clobber/Modify部分声明这些寄存器或内存;

1.寄存器修改通知:

这种情况一般发生在一个寄存器出现在指令列表中,但又不是Input/Output操作表达式所指定的,也不是在一些Input/Output操作表达式中使用"r"或"g"约束时由GCC选择的,同时,此寄存器被指令列表中的指令所修改,而这个寄存器只供当前内联汇编语句使用的情况;比如:

```
__asm__("movl %0,%%ebx"::"a"(__foo):"bx");
```

//这个内联汇编语句中,%ebx出现在指令列表中,并且被指令修改了,但是却未被任何Input/Output操作表达式是所指定,所以,你需要在Clobber/Modify部分指定"bx",以让GCC知道这一点;

因为你在Input/Output操作表达式中指定的寄存器,或当你为一些Input/Output操作表达式使用"r"/"g"约束,让GCC为你选择一个寄存器时,GCC对这些寄存器的状态是非常清楚的,它知道这些寄存器是被修改的,你根本不需要在Clobber/Modify部分声明它们;但除此之外,GCC对剩下的寄存器中哪些会被当前内联汇编语句所修改则一无所知;所以,如果你真的在当前内联汇编指令中修改了它们,那么就最好在Clobber/Modify部分声明它们,让GCC针对这些寄存器做相应的处理;否则,有可能会造成寄存器不一致,从而造成程序执行错误;

在Clobber/Modify部分声明这些寄存器的方法很简单,只需要将寄存器的名字用双引号括起来就可以;如果要声明多个寄存器,则相邻两个寄存器名字之间用逗号隔开;

例如:

```
__asm__("movl %0,%%ebx; popl %%ecx"::"a"(__foo):"bx","cx");
```

这个语句中,声明了bx和cx,告诉GCC:寄存器%ebx和%ecx可能会被修改,要求GCC考虑这个因素;

寄存器名称串:

"al"/"ax"/"eax":代表寄存器%eax
"bl"/"bx"/"ebx":代表寄存器%ebx
"cl"/"cx"/"ecx":代表寄存器%ecx
"dl"/"dx"/"edx":代表寄存器%edx
"si"/"esi":代表寄存器%esi
"di"/"edi":代表寄存器%edi
所以,只需要使用"ax","bx","cx","dx","si","di"就可以了,因为他们都代表对应的寄存器;
如果你在一个内敛汇编语句的Clobber/Modify部分向GCC声明了某个寄存器内存发生了改变,GCC在编译时,如果发现这个被声明的寄存器的内容在此内联汇编之后还要继续使用,那么,GCC会首先将此寄存器的内容保存起来,然后在此内联汇编语句的相关代码生成之后,再将其内容回复;
另外需要注意的是,如果你在Clobber/Modify部分声明了一个寄存器,那么这个寄存器将不能再被用作当前内敛汇编语句的Input/Output操作表达式的寄存器约束,如果Input/Output操作表达式的寄存器约束被指定为"r"/"g",GCC也不会选择已经被声明在Clobber/Modify部分中的寄存器;
例如:

```
__asm__("movl %0,%%ebx"::"a"(__foo):"ax","bx");
```

这条语句中的Input操作表达式"a"(__foo)中已经指定了寄存器%eax,那么在Clobber/Modify部分中个列出的"ax"就是非法的;编译时,GCC会报错;

2.内存修改通知:

除了寄存器的内容会被修改之外,内存的内容也会被修改;如果一个内联汇编语句的指令列表中的指令对内存进行了修改,或者在此内联汇编出现的地方,内存内容可能发生改变,而被改变的内存地址你没有在其Output操作表达式中使用"m"约束,这种情况下,你需要使用在Clobber/Modify部分使用字符串"memory"向GCC声明:"在这里,内存发生了,或可能发生了改变";
例如:

```
void* memset(void* s, char c, size_t count)
{
  __asm__("cld\n\d"
          "rep\n\t"
          "stosb"
          :/*no output*/
          :"a"(c),"D"(s),"c"(count)
          :"cx","di","memory");
  return s;
}
```

如果一个内联汇编语句的Clobber/Modify部分存在"memory",那么GCC会保证在此内联汇编之前,如果某个内存的内容被装入了寄存器,那么,在这个内联汇编之后,如果需要使用这个内存处的内容,就会直接到这个内存处重新读取,而不是使用被存放在寄存器中的拷贝;因为这个时候寄存器中的拷贝很可能已经和内存处的内容不一致了;

3.标志寄存器修改通知:

当一个内联汇编中包含影响标志寄存器eflags的条件,那么也需要在Clobber/Modify部分中使用"cc"来向GCC声明这一点;

分类: ASM

taek
关注 - 2
粉丝 - 6
+加关注

« 上一篇: Linux 汇编语言（GNU GAS汇编）开发指南

» 下一篇：汇编语言---函数调用栈

posted @ 2012-02-05 11:36 taek 阅读(1839) 评论(0) 编辑 收藏

刷新评论 刷新页面 返回顶部