AT&T汇编

AT&T汇编语法

Ted posted @ 2008年9月01日 19:15 in asm with tags ASM , 1929 阅读

在阅读linux内核源代码的时候，必须先掌握汇编，大家都知道，内核代码用的编译器是gcc，而gcc采用的是AT&T的汇编格式，与MS的intel有些区别。

一　AT&T的基本语法

语法上主要有以下几个不同.

★ 寄存器命名原则

AT&T: %eax Intel: eax

★ 源/目的操作数顺序

AT&T: movl %eax,%ebx Intel: mov ebx,eax

★ 常数/立即数的格式

AT&T: movl $_value,%ebx Intel: mov eax,_value

把_value的地址放入eax寄存器

AT&T: movl $0xd00d,%ebx Intel: mov ebx,0xd00d

★ 操作数长度标识

AT&T: movw %ax,%bx Intel: mov bx,ax

★寻址方式

AT&T: immed32(basepointer,indexpointer,indexscale)

Intel: [basepointer + indexpointer*indexscale + imm32)

Linux工作于保护模式下，用的是 3 2 位线性地址，所以在计算地址时

不用考虑segment:offset的问题. 上式中的地址应为:

imm32 + basepointer + indexpointer*indexscale

下面是一些例子:

★直接寻址

AT&T: _booga ; _booga是一个全局的C变量

注意加上$是表示地址引用，不加是表示值引用.

注：对于局部变量，可以通过堆栈指针引用.

Intel: [_booga]

★寄存器间接寻址

AT&T: (%eax)

Intel: [eax]

★变址寻址

AT&T: _variable(%eax)

Intel: [eax + _variable]

AT&T: _array(,%eax,4)

Intel: [eax*4 + _array]

AT&T: _array(%ebx,%eax,8)

Intel: [ebx + eax*8 + _array]

二　基本的行内汇编

　　基本的行内汇编很简单，一般是按照下面的格式

asm("statements");

例如：asm("nop"); asm("cli");

asm　和　__asm__是完全一样的.

如果有多行汇编，则每一行都要加上　"\n\t"

例如：

asm("pushl %eax\n\t"

"movl $0,%eax\n\t"

"popl %eax");

实际上gcc在处理汇编时，是要把asm(...)的内容"打印"到汇编

文件中，所以格式控制字符是必要的.

再例如：

asm("movl %eax,%ebx");

asm("xorl %ebx,%edx");

asm("movl $0,_booga);

在上面的例子中，由于我们在行内汇编中改变了edx和ebx的值，但是

由于gcc的特殊的处理方法，即先形成汇编文件，再交给GAS去汇编，

所以GAS并不知道我们已经改变了edx和ebx的值，如果程序的上下文

需要edx或ebx作暂存，这样就会引起严重的后果. 对于变量_booga也

存在一样的问题. 为了解决这个问题，就要用到扩展的行内汇编语法.

三　扩展的行内汇编

　　扩展的行内汇编类似于Watcom.

基本的格式是：

asm ("statements" : output_regs : input_regs : clobbered_regs);

clobbered_regs指的是被改变的寄存器.

下面是一个例子(为方便起见，我使用全局变量）：

int count=1;

int value=1;

```
int buf[10];
void main()
{
asm(
"cld \n\t"
"rep \n\t"
"stosl"
:
:   "c" (count), "a" (value) , "D" (buf[0])
:   "%ecx","%edi" );
}
```

得到的主要汇编代码为：

```
movl count,%ecx
movl value,%eax
movl buf,%edi
#APP
cld
rep
stosl
#NO_APP
```

cld,rep,stos就不用多解释了.

这几条语句的功能是向buf中写上count个value值.

冒号后的语句指明输入，输出和被改变的寄存器.

通过冒号以后的语句，编译器就知道你的指令需要和改变哪些寄存器，

从而可以优化寄存器的分配.

其中符号"c"(count)指示要把count的值放入ecx寄存器

类似的还有：

a eax

b ebx

c ecx

d edx

S esi

D edi

I 常数值，(0 - 31)

q,r 动态分配的寄存器

g eax,ebx,ecx,edx或内存变量

A 把eax和edx合成一个64位的寄存器(use long longs)

我们也可以让gcc自己选择合适的寄存器.

如下面的例子：

```
asm("leal (%1,%1,4),%0"
:   "=r" (x)
:   "0" (x) );
```

这段代码实现5*x的快速乘法.

得到的主要汇编代码为：

```
movl x,%eax
#APP
leal (%eax,%eax,4),%eax
#NO_APP
movl %eax,x
```

几点说明：

1.使用q指示编译器从eax,ebx,ecx,edx分配寄存器.

使用r指示编译器从eax,ebx,ecx,edx,esi,edi分配寄存器.

2.我们不必把编译器分配的寄存器放入改变的寄存器列表，因为寄存器

已经记住了它们.

3."="是标示输出寄存器，必须这样用.

4.数字%n的用法：

数字表示的寄存器是按照出现和从左到右的顺序映射到用"r"或"q"请求

的寄存器. 如果我们要重用"r"或"q"请求的寄存器的话，就可以使用它们.

5.如果强制使用固定的寄存器的话，如不用%1,而用ebx,则

```
asm("leal (%%ebx,%%ebx,4),%0"
:   "=r" (x)
:   "0" (x) );
```

注意要使用两个%,因为一个%的语法已经被%n用掉了.

下面可以来解释letter 4854-4855的问题：

1、变量加下划线和双下划线有什么特殊含义吗？

加下划线是指全局变量，但我的gcc中加不加都无所谓.

2、以上定义用如下调用时展开会是什么意思？

```
#define _syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
```

```
long __res; \
/* __res应该是一个全局变量  */
__asm__ volatile ("int $0x80" \
/* volatile 的意思是不允许优化，使编译器严格按照你的汇编代码汇编*/
:   "=a" (__res) \
/* 产生代码  movl %eax, __res */
:   "0" (__NR_##name),"b" ((long)(arg1))); \
/* 如果我没记错的话，这里##指的是两次宏展开.
    即用实际的系统调用名字代替"name",然后再把__NR_...展开.
    接着把展开的常数放入eax，把arg1放入ebx */
if (__res >= 0) \
return (type) __res; \
errno = -__res; \
return -1; \
}
/////////////////////////////////////////////////////////////////////
```

四.AT&T汇编与Intel汇编的比较

Intel和AT&T语法的区别
Intel和AT&T汇编语言的语法表面上各不相同,这将导致刚刚学会INTEL汇编的人第一次见到AT&T汇编时会感到困惑,或者反之。因此让我们从基础的东西开始。

前缀
在Intel汇编中没有寄存器前缀或者立即数前缀。而在AT&T汇编中寄存器有一个"%"前缀，立即数有一个"$"前缀。Intel语句中十六进制和二进制数据分别带有"h"和"b"后缀，并且如果十六进制数字的第一位是字母的话，那么数值的前面要加一个"0"前缀。
例如，
Intex Syntax
```
mov    eax,1
mov    ebx,0ffh
int    80h
```

AT&T Syntax
```
movl    $1,%eax
movl    $0xff,%ebx
int     $0x80
```
就像你看到的，AT&T非常难懂。[base+index*scale+disp] 看起来比disp(base,index,scale)更好理解。

操作数的用法
intel语句中操作数的用法和AT&T中的用法相反。在Intel语句中，第一个操作数表示目的，第二个操作数表示源。然而在AT&T语句中第一个操作数表示源而第二个操作数表示目的。在这种情形下AT&T语法的好处是显而易见的。我们从左向右读，也从左向右写，这样比较自然。
例如，
Intex Syntax
```
instr    dest,source
```

```
mov    eax,[ecx]
```

AT&T Syntax

```
instr     source,dest
movl    (%ecx),%eax
```

存储器操作数

如同上面所看到的，存储器操作数的用法也不相同。在Intel语句中基址寄存器用“[”和“]”括起来而在AT&T语句中是用“（”和“）”括起来的。

例如，

Intex Syntax

```
mov    eax,[ebx]
mov    eax,[ebx+3]
```

AT&T Syntax

```
movl    (%ebx),%eax
movl    3(%ebx),%eax
```

AT&T语法中用来处理复杂的操作的指令的形式和Intel语法中的形式比较起来要难懂得多。在Intel语句中这样的形式是segreg:[base+index*scale+disp]。在AT&T语句中这样的形式是%segreg:disp(base,index,scale)。

Index/scale/disp/segreg 都是可选并且可以去掉的。Scale在本身没有说明而index已指定的情况下缺省值为1。segreg的确定依赖于指令本身以及程序运行在实模式还是pmode。在实模式下它依赖于指令本身而pmode模式下它是不需要的。在AT&T语句中用作scale/disp的立即数不要加“$”前缀。

例如

Intel Syntax

```
instr     foo,segreg:[base+index*scale+disp]
mov    eax,[ebx+20h]
add    eax,[ebx+ecx*2h]
lea    eax,[ebx+ecx]
sub    eax,[ebx+ecx*4h-20h]
```

AT&T Syntax

```
instr    %segreg:disp(base,index,scale),foo
movl    0x20(%ebx),%eax
addl    (%ebx,%ecx,0x2),%eax
leal    (%ebx,%ecx),%eax
subl    -0x20(%ebx,%ecx,0x4),%eax
```

后缀

就像你已经注意到的，AT&T语法中有一个后缀，它的意义是表示操作数的大小。“l”代表long，“w”代表word，“b”代表byte。Intel语法中在处理存储器操作数时也有类似的表示，如byte ptr, word ptr, dword ptr。"dword" 显然对应于“long”。这有点类似于C语言中定义的类型，但是既然使用的寄存器的大小对应着假定的数据类型，这样就显得不必要了。

例子:

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
```

注意：从此开始所有的例子都使用AT&T语法
系统调用
本节将介绍linux中汇编语言系统调用的用法。系统调用包括位于/usr/man/man2的手册里第二部分所有的函数。这些函数也在/usr/include/sys/syscall.h中列出来了。一个重要的关于这些函数的列表是在http://www.linuxassembly.org/syscall.html里。这些函数通过linux中断服务：int $0x80来被执行
小于六个参数的系统调用
对于所有的系统调用，系统调用号在%eax中。对于小于六个参数的系统调用，参数依次存放在%ebx,%ecx,%edx,%esi,%edi中，系统调用的返回值保存在%eax中。
系统调用号可以在/usr/include/sys/syscall.h中找到。宏被定义成SYS_的形式，
如SYS_exit, SYS_close等。
例子：（hello world 程序）
参照write（2）的帮助手册，写操作被声明为ssize_t write(int fd, const void *buf, size_t count);
这样，fd应存放在%ebx中，buf放在 %ecx, count 放在 %edx ， SYS_write 放在 %eax中，紧跟着是int $0x80语句来执行系统调用。系统调用的返回值保存在%eax中。

```
$ cat write.s
.include "defines.h"
.data
hello:
    .string "hello world\n"

.globl    main
main:
    movl    $SYS_write,%eax
    movl    $STDOUT,%ebx
    movl    $hello,%ecx
    movl    $12,%edx
    int    $0x80

    ret
$
```

少于5个参数的系统调用的处理也是这样的。只是没有用到的寄存器保持不变罢了。象open或者fcntl这样带有一个可选的额外参数的系统调用也就知道怎么用了。
大于5个参数的系统调用
参数个数大于五个的系统调用仍然把系统调用号保存在%eax中，但是参数存放在内存中，并且指向第一个参数的指针保存在%ebx中。
如果你使用栈，参数必须被逆序压进栈里，即按最后一个参数到第一个参数的顺序。然后将栈的指针拷贝到%ebx中。或者将参数拷贝到一块分配的内存区域，然后把第一个参数的地址保存在%ebx中。
例子：（使用mmap作为系统调用的例子）。在C中使用mmap（ ）：

```
#i nclude
#i nclude
#i nclude
#i nclude
#i nclude

#define STDOUT    1

void main(void) {
    char file[]="mmap.s";
    char *mappedptr;
    int fd,filelen;

    fd=fopen(file, O_RDONLY);
    filelen=lseek(fd,0,SEEK_END);
    mappedptr=mmap(NULL,filelen,PROT_READ,MAP_SHARED,fd,0);
```

```
    write(STDOUT, mappedptr, filelen);
    munmap(mappedptr, filelen);
    close(fd);
}
```

mmap ( ) 参数在内存中的排列：

```
%esp    %esp+4    %esp+8    %esp+12    %esp+16    %esp+20
00000000    filelen    00000001    00000001    fd    00000000
```

等价的汇编程序：

```
$ cat mmap.s
.include "defines.h"

.data
file:
    .string "mmap.s"
fd:
    .long     0
filelen:
    .long     0
mappedptr:
    .long     0

.globl main
main:
    push    %ebp
    movl    %esp,%ebp
    subl    $24,%esp

//    open($file, $O_RDONLY);

    movl    $fd,%ebx    // save fd
    movl    %eax,(%ebx)

//    lseek($fd,0,$SEEK_END);

    movl    $filelen,%ebx    // save file length
    movl    %eax,(%ebx)

    xorl    %edx,%edx

//    mmap(NULL,$filelen,PROT_READ,MAP_SHARED,$fd,0);
    movl    %edx,(%esp)
    movl    %eax,4(%esp)    // file length still in %eax
    movl    $PROT_READ,8(%esp)
    movl    $MAP_SHARED,12(%esp)
    movl    $fd,%ebx    // load file descriptor
    movl    (%ebx),%eax
    movl    %eax,16(%esp)
    movl    %edx,20(%esp)
    movl    $SYS_mmap,%eax
    movl    %esp,%ebx
    int    $0x80

    movl    $mappedptr,%ebx    // save ptr
    movl    %eax,(%ebx)


//     write($stdout, $mappedptr, $filelen);
```

```
//    munmap($mappedptr, $filelen);
//    close($fd);

    movl    %ebp,%esp
    popl    %ebp

    ret
$
```

注意：上面所列出的源代码和本文结束部分的例子的源代码不同。上面列出的代码中没有说明其它的系统调用，因为这不是本节的重点，上面列出的源代码仅仅打开mmap.s文件，而例子的源代码要读命令行的参数。这个mmap的例子还用到lseek来获取文件大小。

Socket系统调用

Socket系统调用使用唯一的系统调用号：SYS_socketcall，它保存在%eax中。Socket函数是通过位于/usr/include/linux/net.h的一个子函数号来确定的，并且它们被保存在%ebx中。指向系统调用参数的一个指针存放在%ecx中。Socket系统调用也是通过int $0x80来执行的。

```
$ cat socket.s
.include "defines.h"

.globl    _start
_start:
    pushl    %ebp
    movl    %esp,%ebp
    sub    $12,%esp

//    socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
    movl    $AF_INET,(%esp)
    movl    $SOCK_STREAM,4(%esp)
    movl    $IPPROTO_TCP,8(%esp)

    movl    $SYS_socketcall,%eax
    movl    $SYS_socketcall_socket,%ebx
    movl    %esp,%ecx
    int    $0x80

    movl     $SYS_exit,%eax
    xorl     %ebx,%ebx
    int     $0x80

    movl    %ebp,%esp
    popl    %ebp
    ret
$
```

命令行参数

在linux中执行的时候命令行参数是放在栈上的。先是argc，跟着是一个由指向命令行中各字符串的指针组成的数组（**argv）并以空指针结束。接下来是一个由指向环境变量的指针组成的数组（**envp）。这些东西在asm中都可以很容易的获得，并且在例子代码（args.s)中有示范。

GCC内联汇编

本节中GCC内联汇编仅涉及x86的应用程序。操作数约束会和其它处理器上的有所不同。关于这部分的说明放在本文的最后。

gcc中基本的内联汇编非常易懂，如

```
__asm__("movl    %esp,%eax");    // look familiar ?
```

或者是

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

如果指定了用作asm的输入、输出数据并指出哪一个寄存器会被修改，会使程序的执行效率提高。
input/output/modify都不是必需的。格式如下：
__asm__("" : output : input : modify);
output和input中必须包含一个操作数约束字符串，并紧跟一个用圆括号括起来的C语言表达式。
输出操作数约束的前面必须有一个"="，表示这是一个输出。可能会有多个输出，多个输入和
多个修改过的寄存器。每个"入口"应该用"，"分隔开，并且入口的总数不多有10个。
操作数约束字符串可以是包含整个寄存器的名称也可以是简写。
Abbrev Table

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

例如:

```
    __asm__("test    %%eax,%%eax", : /* no output */ : "a"(foo));
```

或者是

```
    __asm__("test    %%eax,%%eax", : /* no output */ : "eax"(foo));
```

你可以在__asm__后使用关键字__volatile__："你可以利用在__asm__后使用关键字__volatile__的
方法防止一条'asm'指令被删除、移动或者被重新组合。"（出自gcc的info文件中"Assembler
Instructions with C Expression Operands" 部分)

```
$ cat inline1.c
#i nclude

int main(void) {
    int foo=10,bar=15;

    __asm__ __volatile__ ("addl     %%ebxx,%%eax"
        : "=eax"(foo)         // ouput
        : "eax"(foo), "ebx"(bar)// input
        : "eax"             // modify
    );
    printf("foo+bar=%d\n", foo);
    return 0;
}
$
```

你可能已经注意到现在寄存器使用"%%"前缀而不是"%"。这在使用output/input/modify域时是必要的，
这是因为此时基于其它域的寄存器的别名的使用。我马上来讨论这个问题。
你可以很简单的指定"a"而不是写"eax"或者强制使用一个特殊寄存器如"eax"、"ax"、"al"，
这同样适用于其它一般用途的寄存器（在Abbrev表中列出的）。当你在当前的代码中使用特殊的寄存器
时这好像毫无用处，因此gcc提供了寄存器别名。最多有10个别名（%0—%9），这也是为什么只允许10个
输入/输出的原因。

```
$ cat inline2.c
int main(void) {
   long eax;
   short bx;
   char cl;

   __asm__("nop;nop;nop"); // to separate inline asm from the rest of
            // the code
   __volatile__ __asm__("
      test    %0,%0
      test    %1,%1
      test    %2,%2"
      : /* no outputs */
      : "a"((long)eax), "b"((short)bx), "c"((char)cl)
   );
   __asm__("nop;nop;nop");
   return 0;
}
$ gcc -o inline2 inline2.c
$ gdb ./inline2
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnulibc1"...
(no debugging symbols found)...
(gdb) disassemble main
Dump of assembler code for function main:
... start: inline asm ...
0x8048427 : nop
0x8048428 : nop
0x8048429 : nop
0x804842a : mov 0xfffffffc(%ebp),%eax
0x804842d : mov 0xfffffffa(%ebp),%bx
0x8048431 : mov 0xfffffff9(%ebp),%cl
0x8048434 : test %eax,%eax
0x8048436 : test %bx,%bx
0x8048439 : test %cl,%cl
0x804843b : nop
0x804843c : nop
0x804843d : nop
... end: inline asm ...
End of assembler dump.
$
```

就像你看到的，由内联汇编生成的代码将变量的值放入它们在input域中指定的寄存器中，然后继续执行当前的代码。编译器自动根据变量的大小来侦测操作数的大小，这样相应的寄存器就被别名%0, %1 和 %2代替了（当使用寄存器别名时在存储器里指定操作数的大小回导致编译时发生错误）在操作数约束里也可以使用别名。这不允许你在输入/输出域中指定多于10个的入口。我能想到的这样做的唯一用法是在你指定操作数约束为“q”以便让编译器在a，b，c，d寄存器之间进行选择的时候。当这个寄存器被修改时，我们不会知道选中了那个寄存器，因而不能在modify域中指定它。

这种情况下你只需指定""。
例子：
$ cat inline3.c
#i nclude

```
int main(void) {
    long eax=1,ebx=2;

    __asm__ __volatile__ ("add %0,%2"
        : "=b"((long)ebx)
        : "a"((long)eax), "q"(ebx)
        : "2"
    );
    printf("ebx=%x\n", ebx);
    return 0;
}
```