Linux 中 x86 的内联汇编

将各个部分组合起来

Bharata B. Rao
2001 年 3 月 01 日发布

如果您是 Linux 内核的开发人员，您会发现自己经常要对与体系结构高度相关的功能进行编码或优化代码路径。您很可能是通过将汇编语言指令插入到 C 语句的中间（又称为内联汇编的一种方法）来执行这些任务的。让我们看一下 Linux 中内联汇编的特定用法。（我们将讨论限制在 IA32 汇编。）

GNU 汇编程序简述

让我们首先看一下 Linux 中使用的基本汇编程序语法。GCC（用于 Linux 的 GNU C 编译器）使用 AT&T 汇编语法。下面列出了这种语法的一些基本规则。（该列表肯定不完整；只包括了与内联汇编相关的那些规则。）

寄存器命名
寄存器名称有 % 前缀。即，如果必须使用 eax，它应该用作 %eax。

源操作数和目的操作数的顺序
在所有指令中，先是源操作数，然后才是目的操作数。这与将源操作数放在目的操作数之后的 Intel 语法不同。

```
1 mov %eax, %ebx, transfers the contents of eax to ebx
```

操作数大小
根据操作数是字节 (byte)、字 (word) 还是长型 (long)，指令的后缀可以是 b、w 或 l。这并不是强制性的；GCC 会尝试通过读取操作数来提供相应的后缀。但手工指定后缀可以改善代码的可读性，并可以消除编译器猜测不正确的可能性。

```
1 movb %al, %bl -- Byte move
2     movw %ax, %bx -- Word move
3     movl %eax, %ebx -- Longword move
```

立即操作数
通过使用 $ 指定直接操作数。

```
movl $0xffff, %eax -- will move the value of 0xffff
```

间接内存引用

任何对内存的间接引用都是通过使用 ( ) 来完成的。

```
movb (%esi), %al -- will transfer the byte in the me
 pointed by esi into al
register
```

内联汇编

GCC 为内联汇编提供特殊结构，它具有以下格式：

GCG 的 "asm" 结构

```
    asm ( assembler template

: output operands               (optional)

: input operands                (optional)

: list of clobbered registers
    (optional)

);
```

本例中，汇编程序模板由汇编指令组成。输入操作数是充当指令输入操作数使用的 C 表达式。输出操作数是将对其执行汇编指令输出的 C 表达式。

内联汇编的重要性体现在它能够灵活操作，而且可以使其输出通过 C 变量显示出来。因为它具有这种能力，所以 "asm" 可以用作汇编指令和包含它的 C 程序之间的接口。

一个非常基本但很重要的区别在于 *简单内联汇编*只包括指令，而*扩展内联汇编*包括操作数。要说明这一点，考虑以下示例：

内联汇编的基本要素

```
{
    int a=10, b;
```

```
    asm ("movl %1, %%eax;

movl %%eax, %0;"
        :"=r"(b)  /* output */
        :"r"(a)       /* input */
        :"%eax"); /* clobbered register */
}
```

在上例中，我们使用汇编指令使 "b" 的值等于 "a"。请注意以下几点：

"b" 是输出操作数，由 %0 引用，"a" 是输入操作数，由 %1 引用。

"r" 是操作数的约束，它指定将变量 "a" 和 "b" 存储在寄存器中。请注意，输出操作数约束应该带有一个约束修饰符 "="，指定它是输出操作数。

要在 "asm" 内使用寄存器 %eax，%eax 的前面应该再加一个 %，换句话说就是 %%eax，因为 "asm" 使用 %0、%1 等来标识变量。任何带有一个 % 的数都看作是输入 / 输出操作数，而不认为是寄存器。

第三个冒号后的修饰寄存器 %eax 告诉将在 "asm" 中修改 GCC %eax 的值，这样 GCC 就不使用该寄存器存储任何其它的值。

`movl %1, %%eax` 将 "a" 的值移到 %eax 中，`movl %%eax, %0` 将 %eax 的内容移到 "b" 中。

因为 "b" 被指定成输出操作数，因此当 "asm" 的执行完成后，它将反映出更新的值。换句话说，对 "asm" 内 "b" 所做的更改将在 "asm" 外反映出来。

现在让我们更详细的了解每一项的含义。

汇编程序模板

汇编程序模板是一组插入到 C 程序中的汇编指令（可以是单个指令，也可以是一组指令）。每条指令都应该由双引号括起，或者整组指令应该由双引号括起。每条指令还应该用一个定界符结尾。有效的定界符为新行 (\n) 和分号 (;)。 '\n' 后可以跟一个 tab(\t) 作为格式化符号，增加 GCC 在汇编文件中生成的指令的可读性。 指令通过数 %0、%1 等来引用 C 表达式（指定为操作数）。

如果希望确保编译器不会在 "asm" 内部优化指令，可以在 "asm" 后使用关键字 "volatile"。如果程序必须与 ANSI C 兼容，则应该使用 __asm__ 和 __volatile__，而不是 asm 和 volatile。

操作数

C 表达式用作 "asm" 内的汇编指令操作数。在汇编指令通过对 C 程序的 C 表达式进行操作来执行有意义的作业的情况下，操作数是内联汇编的主要特性。

每个操作数都由操作数约束字符串指定，后面跟用括弧括起的 C 表达式，例如：`"constraint" (C expression)`。操作数约束的主要功能是确定操作数的寻址方式。

可以在输入和输出部分中同时使用多个操作数。每个操作数由逗号分隔开。

在汇编程序模板内部，操作数由数字引用。如果总共有 *n* 个操作数（包括输入和输出），那么第一个输出操作数的编号为 0，逐项递增，最后那个输入操作数的编号为 *n* -1。总操作数的数目限制在 10，如果机器描述中任何指令模式中的最大操作数数目大于 10，则使用后者作为限制。

修饰寄存器列表

如果 "asm" 中的指令指的是硬件寄存器，可以告诉 GCC 我们将自己使用和修改它们。这样，GCC 就不会假设它装入到这些寄存器中的值是有效值。通常不需要将输入和输出寄存器列为 clobbered，因为 GCC 知道 "asm" 使用它们（因为它们被明确指定为约束）。不过，如果指令使用任何其它的寄存器，无论是明确的还是隐含的（寄存器不在输入约束列表中出现，也不在输出约束列表中出现），寄存器都必须被指定为修饰列表。修饰寄存器列在第三个冒号之后，其名称被指定为字符串。

至于关键字，如果指令以某些不可预知且不明确的方式修改了内存，则可能将 "memory" 关键字添加到修饰寄存器列表中。这样就告诉 GCC 不要在不同指令之间将内存值高速缓存在寄存器中。

操作数约束

前面提到过，"asm" 中的每个操作数都应该由操作数约束字符串描述，后面跟用括弧括起的 C 表达式。操作数约束主要是确定指令中操作数的寻址方式。约束也可以指定：

- 是否允许操作数位于寄存器中，以及它可以包括在哪些种类的寄存器中
- 操作数是否可以是内存引用，以及在这种情况下使用哪些种类的地址
- 操作数是否可以是立即数

约束还要求两个操作数匹配。

常用约束

在可用的操作数约束中，只有一小部分是常用的；下面列出了这些约束以及简要描述。有关操作数约束的完整列表，请参考 GCC 和 GAS 手册。

寄存器操作数约束 (r)
使用这种约束指定操作数时，它们存储在通用寄存器中。请看下例：

```
asm ("movl %%cr3, %0\n" :"=r"(cr3val));
```

这里，变量 cr3val 保存在寄存器中，%cr3 的值复制到寄存器上，cr3val 的值从该寄存器更新到内存中。指定 "r" 约束时，GCC 可以将变量 cr3val 保存在任何可用的 GPR 中。要指定寄存器，必须通过使用特定的寄存器约束直接指定寄存器名。

```
a   %eax
b   %ebx
c   %ecx
d   %edx
S   %esi
D   %edi
```

内存操作数约束 (m)
当操作数位于内存中时，任何对它们执行的操作都将在内存位置中直接发生，这与寄存器约束正好相反，后者先将值存储在要修改的寄存器中，然后将它写回内存位置中。但寄存器约束通常只在对于指令来说它们是绝对必需的，或者它们可以大大提高进程速度时使用。当需要在 "asm" 内部更新 C 变量，而您又确实不希望使用寄存器来保存其值时，使用内存约束最为有效。例如，idtr 的值存储在内存位置 loc 中：

```
("sidt %0\n" : :"m"(loc));
```

匹配（数字）约束
在某些情况下，一个变量既要充当输入操作数，也要充当输出操作数。可以通过使用匹配约束在 "asm" 中指定这种情况。

```
asm ("incl %0" :"=a"(var):"0"(var));
```

在匹配约束的示例中，寄存器 %eax 既用作输入变量，也用作输出变量。将 var 输入读取到 %eax，增加后将更新的 %eax 再次存储在 var 中。这里的 "0" 指定第 0 个输出变量相同的约

束。即，它指定 var 的输出实例只应该存储在 %eax 中。该约束可以用于以下情况：

输入从变量中读取，或者变量被修改后，修改写回到同一变量中

不需要将输入操作数和输出操作数的实例分开

使用匹配约束最重要的意义在于它们可以导致有效地使用可用寄存器。

一般内联汇编用法示例

以下示例通过各种不同的操作数约束说明了用法。有如此多的约束以至于无法将它们一一列出，这里只列出了最经常使用的那些约束类型。

"asm" 和寄存器约束 "r" 让我们先看一下使用寄存器约束 r 的 "asm"。我们的示例显示了 GCC 如何分配寄存器，以及它如何更新输出变量的值。

```
int main(void)
{
    int x = 10, y;

    asm ("movl %1, %%eax;

 "movl %%eax, %0;"
        :"=r"(y)  /* y is output operand */
        :"r"(x)       /* x is input operand */
        :"%eax"); /* %eax is clobbered register */
}
```

在该例中，x 的值复制为 "asm" 中的 y。x 和 y 都通过存储在寄存器中传递给 "asm"。为该例生成的汇编代码如下：

```
main:
pushl %ebp
movl %esp,%ebp
subl $8,%esp
movl $10,-4(%ebp)
movl -4(%ebp),%edx  /* x=10 is stored in %edx */
#APP    /* asm starts here */
movl %edx, %eax     /* x is moved to %eax */
movl %eax, %edx     /* y is allocated in edx and up
#NO_APP /* asm ends here */
movl %edx,-8(%ebp)  /* value of y in stack is updat

 the value in %edx */
```

当使用 "r" 约束时，GCC 在这里可以自由分配任何寄存器。在我们的示例中，它选择 %edx 来存储 x。在读取了 %edx 中 x 的值后，它为 y 也分配了相同的寄存器。

因为 y 是在输出操作数部分中指定的，所以 %edx 中更新的值存储在 -8(%ebp)，堆栈上 y 的位置中。如果 y 是在输入部分中指定的，那么即使它在 y 的临时寄存器存储值 (%edx) 中被更新，堆栈上 y 的值也不会更新。

因为 %eax 是在修饰列表中指定的，GCC 不在任何其它地方使用它来存储数据。

输入 x 和输出 y 都分配在同一个 %edx 寄存器中，假设输入在输出产生之前被消耗。请注意，如果您有许多指令，就不是这种情况了。要确保输入和输出分配到不同的寄存器中，可以指定 & 约束修饰符。下面是添加了约束修饰符的示例。

```
int main(void)
{
    int x = 10, y;

    asm ("movl %1, %%eax;

 "movl %%eax, %0;"
        :"=&r"(y) /* y is output operand, note the

 & constraint modifier. */
        :"r"(x)       /* x is input operand */
        :"%eax"); /* %eax is clobbered register */
}
```

以下是为该示例生成的汇编代码，从中可以明显地看出 x 和 y 存储在 "asm" 中不同的寄存器中。

```
main:
pushl %ebp
movl %esp,%ebp
subl $8,%esp
movl $10,-4(%ebp)
movl -4(%ebp),%ecx  /* x, the input is in %ecx */
#APP
    movl %ecx, %eax
    movl %eax, %edx     /* y, the output is in %edx
#NO_APP
movl %edx,-8(%ebp)
```

特定寄存器约束的使用

现在让我们看一下如何将个别寄存器作为操作数的约束指定。在下面的示例中，cpuid 指令采用 %eax 寄存器中的输入，然后在四个寄存器中给出输出：%eax、%ebx、%ecx、%edx。对 cpuid 的输入（变量 "op"）传递到 "asm" 的 eax 寄存器中，因为 cpuid 希望它这样做。在输出中使用 a、b、c 和 d 约束，分别收集四个寄存器中的值。

```
asm ("cpuid"
: "=a" (_eax),
"=b" (_ebx),
"=c" (_ecx),
"=d" (_edx)
: "a" (op));
```

在下面可以看到为它生成的汇编代码（假设 _eax、_ebx 等... 变量都存储在堆栈上）：

```
movl -20(%ebp),%eax /* store 'op' in %eax -- input *
#APP
cpuid
#NO_APP
movl %eax,-4(%ebp)  /* store %eax in _eax -- output
movl %ebx,-8(%ebp)  /* store other registers in
movl %ecx,-12(%ebp)
 respective output variables */
movl %edx,-16(%ebp)
```

strcpy 函数可以通过以下方式使用 "S" 和 "D" 约束来实现：

```
asm ("cld\n

rep\n

movsb"

: /* no input */

:"S"(src), "D"(dst), "c"(count));
```

通过使用 "S" 约束将源指针 src 放入 %esi 中，使用 "D" 约束将目的指针 dst 放入 %edi 中。因为 rep 前缀需要 count 值，所以将它放入 %ecx 中。

在下面可以看到另一个约束，它使用两个寄存器 %eax 和 %edx 将两个 32 位的值合并在一起，然后生成一个64 位的值：

```
#define rdtscll(val) \
 __asm__ __volatile__ ("rdtsc" : "=A" (val))
The generated assembly looks like this (if val has
#APP
rdtsc
#NO_APP
movl %eax,-8(%ebp)  /* As a result of A constraint
movl %edx,-4(%ebp)
 %eax and %edx serve as outputs */
Note here that the values in %edx:%eax serve as 64
```

使用匹配约束

在下面将看到系统调用的代码，它有四个参数：

```
#define _syscall4(type,name,type1,arg1,type2,arg2,t
type name (type1 arg1, type2 arg2, type3 arg3, type
{ \
long __res; \
__asm__ volatile ("int $0x80" \
: "=a" (__res) \
: "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(
"d" ((long)(arg3)),"S" ((long)(arg4))); \
__syscall_return(type,__res); \
}
```

在上例中，通过使用 b、c、d 和 S 约束将系统调用的四个自变量放入 %ebx、%ecx、%edx 和 %esi 中。请注意，在输出中使用了 "=a" 约束，这样，位于 %eax 中的系统调用的返回值就被放入变量 __res 中。通过将匹配约束 "0" 用作输入部分中第一个操作数约束，syscall 号 __NR_##name 被放入 %eax 中，并用作对系统调用的输入。这样，这里的 %eax 既可以用作输入寄存器，又可以用作输出寄存器。没有其它寄存器用于这个目的。另请注意，输入（syscall 号）在产生输出（syscall 的返回值）之前被消耗（使用）。

内存操作数约束的使用

请考虑下面的原子递减操作：

```
__asm__ __volatile__(
"lock; decl %0"
:"=m" (counter)
:"m" (counter));
```

为它生成的汇编类似于：

```
#APP
    lock
    decl -24(%ebp) /* counter is modified on its mem
#NO_APP.
```

您可能考虑在这里为 counter 使用寄存器约束。如果这样做，counter 的值必须先复制到寄存器，递减，然后对其内存更新。但这样您会无法理解锁定和原子性的全部意图，这些明确显示了使用内存约束的必要性。

使用修饰寄存器

请考虑内存拷贝的基本实现。

```
    asm ("movl $count, %%ecx;

up: lodsl;

stosl;

loop up;"
        :           /* no output */
        :"S"(src), "D"(dst) /* input */
        :"%ecx", "%eax" );  /* clobbered list */
```

当 lodsl 修改 %eax 时，lodsl 和 stosl 指令隐含地使用它。%ecx 寄存器明确装入 count。但 GCC 在我们通知它以前是不知道这些的，我们是通过将 %eax 和 %ecx 包括在修饰寄存器集中来通知 GCC 的。在完成这一步之前，GCC 假设 %eax 和 %ecx 是自由的，它可能决定将它们用作存储其它的数据。请注意，%esi 和 %edi 由 "asm" 使用，它们不在修饰列表中。这是因为已经声明 "asm" 将在输入操作数列表中使用它们。这里最低限度是，如果在 "asm" 内部使用寄存器（无论是明确还是隐含地），既不出现在输入操作数列表中，也不出现在输出操作数列表中，必须将它列为修饰寄存器。

结束语

总的来说，内联汇编非常巨大，它提供的许多特性我们甚至在这里根本没有涉及到。但如果掌握了本文描述的基本材料，您应该可以开始对自己的内联汇编进行编码了。