Making plain binary files using a C compiler (i386+)

作者：Cornelis Frank April 10, 2000
翻译：coly li

我写这篇文章是因为在 internet 上关于这个主题的信息很少，而我的 EduOS 项目又需要它。对于由此文中信息所引申、引起的意外或不利之处，作者均不负有任何责任。如果因为我的糟糕英语导致你的机器故障，那是你的问题，而不是我的。

1，你需要什么工具

- 一个 i386 或者更高 x86CPU 配置的 PC
- 一个 Linux 发行版本，Redhat 或者 Slackware 就不错。
- 一个 GNU GCC 编译器。一般 linux 发行版本都会自带，可以通过如下命令检查你的 GCC 的版本和类型：

  ```
  gcc --version
  ```

  这将会产生如下的输出：

  ```
  2.7.2.3
  ```

  可能你的输出和上面的不一样，但是这不碍事。
- Linux 下的 binutils 工具包。
- NASM 0.97 或者更高的版本汇编器。Netwide Assembler，即 NASM，是一个 80x86 的模块化的可移植的汇编器。他支持一系列的对象文件格式，包括 linux 的”a.out”，和 ELF，NetBSD/FreeBSD，COFF，Microsoft 16 位 OBJ 和 Win32。他也可以输出无格式二进制文件。他的语法设计的非常简单并且易于理解，同 intel 的非常相似但是没有那么复杂。它支持 Pentium，P6 和 MMX 操作数，并且支持宏。

  如果你的机器上没有 NASM，可以从下述网站下载：

  http://sunsite.unc.edu/pub/Linux/devel/lang/assemblers/
- 一个象 emacs 或者 pico 那样的文本编辑器。

1.1 安装 Netwide 汇编器（NASM）

假定当前目录下存有 nasm-0.97 的压缩包，输入：

```
gunzip nasm-0.97.tar.gz
tar -vxf nasm-0.97.tar
```

这将建立一个名为 nasm-0.97 的目录，进入这个目录，下一步我们将输入如下命令来编译这个汇编器：

```
./configure
make
```

这将建立可执行文件 nasm 和 ndisasm。你可以将这些文件复制到你的/usr/bin 目录下，这样更方便调用。现在你可以从你的计算机上删除目录 nasm-0.97 了。我一般会在 RedHat linux 5.1 或者 Slackware 3.1 下编译，这样不会引起太多麻烦。

2，使用 C 语言生成第一个二进制文件

使用你的文本编辑器建立一个名为 test.c 的文件，输入如下内容：

```
int main(){
}
```

输入如下命令进行编译：

```
gcc -c test.c
ld -o test -Ttext 0x0 -e main test.o
objcopy -R .note -R .comment -S -O binary test test.bin
```

这将建立我们的名为 test.bin 的二进制文件。我们可以使用 ndisasm 查看这个文件，需输入如下命令：

```
ndisasm -b 32 test.bin
```

这将生成如下输出：

```
00000000    55          push ebp
00000001    89E5        mov ebp,esp
00000003    C9          leave
00000004    C3          ret
```

我们看到 3 列。第一列是指令的内存地址，第二列是指令的字节代码，最后一行是对应的汇编指令本身。

2.1 解剖 test.bin

我们刚才看到的代码紧紧建立了一个函数的基本框架。保存寄存器 ebp 为将来在处理函数参数时使用。如你所发现的，代码是 32 位的，这是因为 GNU GCC 只能生成 32 位的代码。因此，如果你要运行这个代码，你首先需要搭建一个 32 位的运行环境，如 linux。另外运行这个代码之前你还需进入保护模式。

你还可以直接使用 ld 创建二进制文件，可以按如下方法编译 test.c 文件：

```
gcc -c test.c
ld test.o -o test.bin -Ttext 0x0 -e main -oformat binary
```

这将建立同上面看到的一模一样的二进制代码。

3，使用局部变量编程

Next we will take a look on how GCC handles the reservation of a local variable. Here fore we

下面我们看看 GCC 是如何为局部变量预留空间的。这里我们将建立一个包含如下内容的 test.c 文件：

```
int main () {
    int i;          /* declaration of an int */
    i = 0x12345678; /* hexadecimal */
}
```

用如下命令编译这个文件：

```
gcc -c test.c
ld -o test -Ttext 0x0 -e main test.o
objcopy -R .note -R .comment -S -O binary test test.bin
```

当我们编译完后就获得了如下内容的二进制文件：

```
00000000    55              push ebp
00000001    89E5            mov ebp,esp
00000003    83EC04          sub esp,byte +0x4
00000006    C745FC78563412  mov dword [ebp-0x4],0x12345678
0000000D    C9              leave
0000000E    C3              ret
```

3.1 解剖 test.bin

现在得到的二进制文件中的头两个和末尾两个指令，同先前例子中的完全相同。在这两部分之间加入了 2 个新的指令。第一个是将 esp 减少 4，这是 GCC 为一个 int 类型预留空间的方法，因为在堆栈中 int 类型占了 4 个字节。下一个指令告诉了我们对于 ebp 寄存器的使用。这个寄存器在函数中是保持不变的，被用来查找堆栈中的局部变量。这些局部变量被保存在称之为 local stack frame 的地方。在这里的上下文中 ebp 寄存器被称为 frame 指针（frame pionter）。接下来的指令将堆栈顶部的整型变量设置为数值0x12345678。注意到处理器是采用反序来存储变量的，所以我们在对应的第二列看到的是78563412。这种现象被成为“倒序存储”（参见 Intel Architecture Software Developer’s Manual, Volume 1: Basic Architecture, 1.4.1. Bit and Byte Order）。注意你也可以按照前面提到的方法直接建立二进制文件，如下：

```
gcc -c test.c
ld -o test.bin -Ttext 0x0 -e main -oformat binary test.o
```

这将产生同样的二进制代码。

3.2 直接赋值

现在我们将原来语句中的：

```
int i;
i = 0x12345678;
```

改为：

```
int i = 0x12345678;
```

这时，我们得到的是完全一样的二进制文件。但是当我们将 I 作为全局变量来使用的时候，这就不一样了。这个可一定要注意。

4，使用全局变量编程

下面我们将看看 GCC 如何处理全局变量的，我们这一次使用如下的测试 c 代码内容，文件名还是 test.c：

```
int i; /* declaration of global variable */
int main () {
    i = 0x12345678;
}
```

使用如下命令编译：

```
gcc -c test.c
```

```
ld -o test -Ttext 0x0 -e main test.o
objcopy -R .note -R .comment -S -O binary test test.bin
```

这将产生如下二进制代码：

```
00000000    55                  push ebp
00000001    89E5                mov ebp,esp
00000003    C705101000007856    mov dword [0x1010],0x12345678
            -3412
0000000D    C9                  leave
0000000E    C3                  ret
```

4.1 解剖 test.bin

这段代码中部的指令将会将我们设置的数值赋值给内存中某个位置，在我们这个例子中这个地址是 0x1010。这是因为缺省情况下连接器 ld 的给数据段以页为单位分配地址。我们可以使用参数-N 将连接器 ld 的该功能 disable。这样我们就得到了如下代码：

```
00000000    55                  push ebp
00000001    89E5                mov ebp,esp
00000003    C705100000007856    mov dword [0x10],0x12345678
            -3412
0000000D    C9                  leave
0000000E    C3                  ret
```

正如我们现在所见，数据被紧跟着保存在了代码的后面。我们也可以自己指定数据段的位置，使用如下命令编译：

```
gcc -c test.c
ld -o test -Ttext 0x0 -Tdata 0x1234 -e main -N test.o
objcopy -R .note -R .comment -S -O binary test test.bin
```

这样我们将得到如下的二进制文件：

```
00000000    55                  push ebp
00000001    89E5                mov ebp,esp
00000003    C705341200007856    mov dword [0x1234],0x12345678
            -3412
0000000D    C9                  leave
0000000E    C3                  ret
```

这时，全局变量就被保存到我们指定的地址 0x1234 中了。因此，如果我们在运行 ld 时使用-Tdata，我们就可以自己定义数据段的位置，否则，数据段会被保存在紧跟着代码段的位置上。这就是为什么我们称 int I 为一个全局变量（就是说全局变量是定义在数据段中的，而局部变量是定义在堆栈中的）。我们同样可以直接使用带有-oformat 参数的 ld 直接生成二进制文件。

4.2 直接赋值

通过我们的试验可以看出直接对全局变量赋值可以有两种方法来处理：作为普通的全局变量来处理，或者在二进制文件中作为数据直接在代码后面保存。当变量被赋予 const 属性时，ld 就将全局变量作为数据处理了。

看如下代码：

```
const int c = 0x12345678;
int main () {
}
```

使用如下命令编译：

```
gcc -c test.c
ld -o test.bin -Ttext 0x0 -e main -N -oformat binary test.o
```

我们将得到如下的二进制文件：

```
00000000    55      push ebp
00000001    89E5    mov ebp,esp
00000003    C9      leave
00000004    C3      ret
00000005    0000    add [eax],al
00000007    007856  add [eax+0x56],bh
0000000A    3412    xor al,0x12
```

我们可以看到在我们的二进制文件的末尾，有一些多出来的字节，这就是分配了 4 个字节的保存设为 const 属性的全局变量的只读数据段。

4.2.1 objdump 的使用方法

使用 objdump，我们可以获得更多的信息：

```
objdump  -disassemble-all test.o
```

这将给出如下转储信息（注意这里就是 AT&T 的代码风格了，本文有一部分很混乱，按照 AT&T 的顺序、Intel 的指令符号来表示，这里一定要看清楚了！）：

```
test.o:     file format elf32-i386


Disassembly of section .text:


00000000 <main>:
    0:      55      pushl %ebp
    1:      89 e5   movl %esp,%ebp
    3:      c9      leave
    4:      c3      ret
Disassembly of section .data:
```

```
Disassembly of section .rodata:

00000000 <c>:
    0:      78 56       js 58 <main+0x58>
    2:      34 12       xorb $0x12,%al
```

我们可以清楚的看到只读数据段保存着我们的全局“常值变量”cont int c。再看看下面这段程序：

```
int i = 0x12345678;
const int c = 0x12346578;
int main () {
}
```

当我们对这个程序编译完之后，生成它的 objdump 文件如下：

```
test.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
    0:      55      pushl %ebp
    1:      89 e5   movl %esp,%ebp
    3:      c9      leave
    4:      c3      ret
Disassembly of section .data:

00000000 <i>:
    0:      78 56   js 58 <main+0x58>
    2:      34 12   xorb $0x12,%al
Disassembly of section .rodata:

00000000 <c>:
    0:      78 56   js 58 <main+0x58>
    2:      34 12   xorb $0x12,%al
```

（注意 js 58 和 xorb $0x12, %al 都是根据 78 56 和 34 12 反编译过来的指令，实际上我们这里只是只读的数据，所以这些指令没有任何意义。）

我们可以看到 int i 在数据段中，而 int c 在只读数据段中。所以当 ld 不得不使用全局常量时，他将自动的将全部变量保存在数据段（应该是只读数据段吧？）。

5，指针

现在我们看看 GCC 如何使用指针来处理变量，这里我们使用如下的程序代码：

```
int main () {
    int i;
    int *p;             /* a pointer to an integer */
    p = &i;             /* let pointer p points to integer i */
    *p = 0x12345678;    /* makes i = 0x12345678 */
}
```

上述程序将会生成如下的二进制代码和汇编指令：

```
00000000        55              push ebp
00000001        89E5            mov ebp,esp
00000003        83EC08          sub esp,byte +0x8
00000006        8D55FC          lea edx,[ebp-0x4]
00000009        8955F8          mov [ebp-0x8],edx
0000000C        8B45F8          mov eax,[ebp-0x8]
0000000F        C70078563412    mov dword [eax],0x12345678
00000015        C9              leave
00000016        C3              ret
```

（Intel 风格代码，小心了，不是 AT&T 的）

5.1 解剖 test.bin

同样，头两个和最后两个指令还是一样的，下面我们遇到这个指令：

```
sub esp,byte +0x8
```

这个指令将会在堆栈中为局部变量预留 8 个字节。看起来似乎是使用 4 个字节来保存一个指针变量。这时堆栈会如图 1 所示结构。

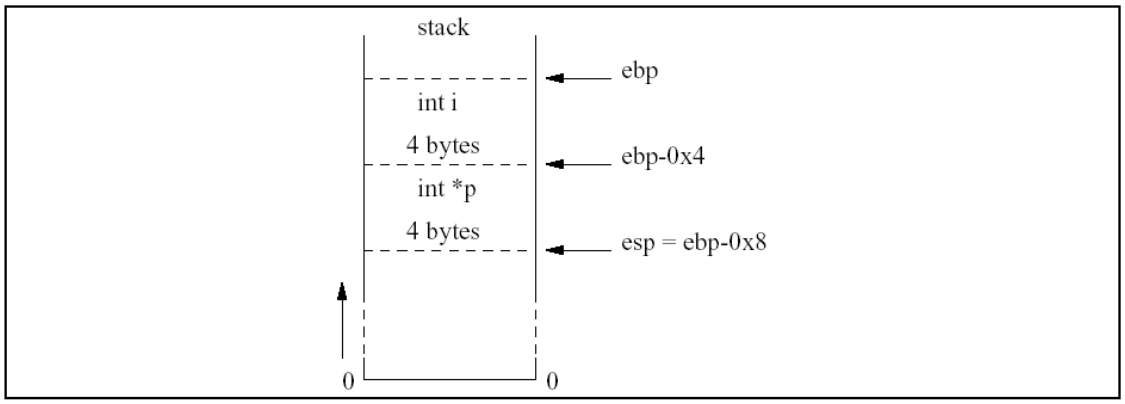

Figure 1: The stack

图 1　堆栈

如你所见，lea 指令将加载 int i 的有效地址，接下来这个数值被保存到了 int p。在这之后，*p 被用作一个指向 int i 的指针，向指针所指向的内容赋值 0x12345678。

6，调用函数

现在我们来看看 GCC 是如何处理函数调用的，使用如下测试代码：

```
void f (); /* function prototype */

int main () {
    f (); /* function call */
}

void f () { /* function definition */
}
```

这将生成如下的二进制代码（Intel 风格）：

```
00000000    55          push ebp
00000001    89E5        mov ebp,esp
00000003    E804000000  call 0xc
00000008    C9          leave
00000009    C3          ret
0000000A    89F6        mov esi,esi
0000000C    55          push ebp
0000000D    89E5        mov ebp,esp
0000000F    C9          leave
00000010    C3          ret
```

6.1 解剖 test.bin

在 main 函数中我们可以清晰的看到调用了位于地址 0xC 的空函数 f。这个空函数 f 具有同 main 函数相同的基本框架。这也就说明在入口函数和别的函数之间，并没有结构性的差异。当你使用带参数-M>mem.txt 的 ld 进行链接时，你将得到一个文本文件，这里面记录了关于每个东西都是如何连接和存储到内存中去的详细而有用的文档。在文件 mem.txt 中，你将会看到如下两行：

```
Address of section .text set to 0x0
Address of section .data set to 0x1234
```

这意味着二进制代码从地址 0x 开始，并且保存全局变量的数据段从地址 0x1234 开始。你还可以看到类似如下的信息：

```
.text          0x00000000      0x11
*(.text)
.text          0x00000000      0x11 test.o
               0x0000000c              f
               0x00000000              main
```

第一列是段的名称，在这个例子中，它是一个.text 段。第二列是以数字形式表示的段。第三列是段的长度；最后一列是一些额外的信息，例如函数名字和使用的 obj 文件。我们现在可以清晰的看到函数 f 从偏移量地址 0xC 的位置开始，主函数作为了这个二进制文件的入口点。这里长度是 0x11 也是正确的，因为最后一个指令(ret)在地址 0x10，占用了 1 个字节。

6.2 objdump 的使用

objdump 工具可以用来显示 object 文件的信息。这些信息对于验证 object 文件的内部结构是非常有用的。可以使用如下命令使用 objdump：

```
objdump --disassemble-all test.o
```

屏幕上将获得如下信息：

```
test.o:         file format elf32-i386

Disassembly of section .text:

00000000 <main>:
    0:      55                  pushl %ebp
    1:      89 e5               movl %esp,%ebp
    3:      e8 04 00 00 00      call c <f>
    8:      c9                  leave
    9:      c3                  ret
```

```
    a:      89 f6                movl %esi,%esi

0000000c <f>:
    c:      55                   pushl %ebp
    d:      89 e5                movl %esp,%ebp
    f:      c9                   leave
    10:     c3                   ret
Disassembly of section .data:
```

当你想要学习 GCC 生成的二进制代码时，这些信息仍然是非常有用的。注意他们不是按照 Intel 的语法风格在显示这些指令。他们使用类似 pushl 和 movl 的指令。指令尾部的 l 指明这个指令是在操作一个 32 位的操作数。另一个于 Intel 风格不同的是参数的语法顺序，这里的操作数顺序同 Intel 风格中正好是相反的。下面的例子（将数据从 EBX 移动到 EAX）指明了这里同 Intel 风格的 2 个主要区别：

```
MOV EAX,EBX ;       Intel 风格
movl %ebx,%eax ;    GNU 风格
```

对于 Intel 风格来说，目标操作数在前，源操作数在后。

7，返回码

你可能注意到我总是使用 int main（）作为我的函数的定义，但是我从来没有返回一个 int 型数值。现在我们试试。

```
int main () {
    return 0x12345678;
}
```

这个程序生成下列二进制代码：

```
00000000    55          push ebp
00000001    89E5        mov ebp,esp
00000003    B878563412  mov eax,0x12345678
00000008    EB02        jmp short 0xc
0000000A    89F6        mov esi,esi
0000000C    C9          leave
0000000D    C3          ret
```

7.1 解剖 test.bin

正如你所见返回值是通过寄存器 eax 返回的。因为 eax 并不是一个需要我们必须明确的写入返

回值的寄存器，因此我们在返回的时候也可以什么都不写。这样还有一个好处，就是返回码保存在寄存器里，我们同样也没有必要明确的去读取这个返回码。当我们调用 ANSI C 的 printf 函数向屏幕上输出一些信息的时候，就总是这么做的。我们总是如下使用这个函数：

```
printf (...);
```

虽然 printf 函数精确的返回了一个 int 型数值给调用者。当然如果返回类型大于 4 个字节时编译器不会使用这个方法，下面我们就会讨论当返回值大于 4 个字节时的情况。

7.2 返回数据结构

考虑如下代码：

```
typedef struct {
    int a,b,c,d;
    int i [10];
} MyDef;

MyDef MyFunc (); /* function prototype */

int main () { /* entry point */
    MyDef d;
    d = MyFunc ();
}

MyDef MyFunc () { /* a local function */
    MyDef d;
    return d;
}
```

可以得到如下的二进制代码：

```
00000000    55          push ebp
00000001    89E5        mov ebp,esp
00000003    83EC38      sub esp,byte +0x38
00000006    8D45C8      lea eax,[ebp-0x38]
00000009    50          push eax
0000000A    E805000000  call 0x14
0000000F    83C404      add esp,byte +0x4
00000012    C9          leave
00000013    C3          ret
00000014    55          push ebp
00000015    89E5        mov ebp,esp
```

```
00000017    83EC38      sub esp,byte +0x38
0000001A    57          push edi
0000001B    56          push esi
0000001C    8B4508      mov eax,[ebp+0x8]
0000001F    89C7        mov edi,eax
00000021    8D75C8      lea esi,[ebp-0x38]
00000024    FC          cld
00000025    B90E000000  mov ecx,0xe
0000002A    F3A5        rep movsd
0000002C    EB02        jmp short 0x30
0000002E    89F6        mov esi,esi
00000030    89C0        mov eax,eax
00000032    8D65C0      lea esp,[ebp-0x40]
00000035    5E          pop esi
00000036    5F          pop edi
00000037    C9          leave
00000038    C3          ret
```

解剖 test.bin

在 main 函数的地址 0x3 处，我们看到编译器在堆栈中预留了 38 个字节。这正是 MyDef 结构的大小。从地址 0x6 到 0x9，我们看到了返回大于 4 个字节信息的解决方法——由于 Mydef 大于 4 个字节，编译器将指向 d 的指针传递给了位于 0x14 的函数 MyFunc。这个函数然后可以使用这个指针去填充数据。请注意 MyFunc 在它的 C 语言函数声明中并没有声明任何的参数，而有一个参数被传递到了函数 MyFunc。为了填充这个数据结构，MyFunc 使用了一个 32bit 的移动指令：

```
0000002A F3A5 rep movsd
```

7.3 返回数据结构 II

当然我们可以问我们自己：如果我们不想存储返回的数据结构，那么哪一个指针将会给 MyFunc？看看下面的程序：

```
typedef struct {
    int a,b,c,d;
    int i [10];
} MyDef;

MyDef MyFunc (); /* function prototype */

int main () { /* entry point */
    MyFunc ();
```

```
}

MyDef MyFunc () { /* a local function */
    MyDef d;
    return d;
}
```

这个程序将得到如下的二进制代码：

```
00000000    55              push ebp
00000001    89E5            mov ebp,esp
00000003    83EC38          sub esp,byte +0x38
00000006    8D45C8          lea eax,[ebp-0x38]
00000009    50              push eax
0000000A    E805000000      call 0x14
0000000F    83C404          add esp,byte +0x4
00000012    C9              leave
00000013    C3              ret
00000014    55              push ebp
00000015    89E5            mov ebp,esp
00000017    83EC38          sub esp,byte +0x38
0000001A    57              push edi
0000001B    56              push esi
0000001C    8B4508          mov eax,[ebp+0x8]
0000001F    89C7            mov edi,eax
00000021    8D75C8          lea esi,[ebp-0x38]
00000024    FC              cld
00000025    B90E000000      mov ecx,0xe
0000002A    F3A5            rep movsd
0000002C    EB02            jmp short 0x30
0000002E    89F6            mov esi,esi
00000030    89C0            mov eax,eax
00000032    8D65C0          lea esp,[ebp-0x40]
00000035    5E              pop esi
00000036    5F              pop edi
00000037    C9              leave
00000038    C3              ret
```

解剖

这个代码告诉我们——虽然在位于 0x0 的入口函数 main 中没有任何的局部变量——函数还是为一个变量预留了 38 个字节的空间。然后一个指向这个数据结构的指针被传递到了位于 0x14 的函数 MyFunc，就像上面的例子一样。同时要注意到 MyFunc 函数内部没有改变。

8，传递函数指针

在这部分，我们将看看函数指针是如何被传递给函数的，让我们看看如下例程：

```
char res; /* global variable */
char f (char a, char b); /* function prototype */

int main () { /* entry point */
    res = f (0x12, 0x23); /* function call */
}

char f (char a, char b) { /* function definition */
    return a + b; /* return code */
}
```

上面的例程将生成如下的二进制代码：

```
00000000        55              push ebp
00000001        89E5            mov ebp,esp
00000003        6A23            push byte +0x23
00000005        6A12            push byte +0x12
00000007        E810000000      call 0x1c
0000000C        83C408          add esp,byte +0x8
0000000F        88C0            mov al,al
00000011        880534120000    mov [0x1234],al
00000017        C9              leave
00000018        C3              ret
00000019        8D7600          lea esi,[esi+0x0]
0000001C        55              push ebp
0000001D        89E5            mov ebp,esp
0000001F        83EC04          sub esp,byte +0x4
00000022        53              push ebx
00000023        8B5508          mov edx,[ebp+0x8]
00000026        8B4D0C          mov ecx,[ebp+0xc]
00000029        8855FF          mov [ebp-0x1],dl
0000002C        884DFE          mov [ebp-0x2],cl
0000002F        8A45FF          mov al,[ebp-0x1]
00000032        0245FE          add al,[ebp-0x2]
00000035        0FBED8          movsx ebx,al
00000038        89D8            mov eax,ebx
```

```
0000003A        EB00            jmp short 0x3c
0000003C        8B5DF8          mov ebx,[ebp-0x8]
0000003F        C9              leave
00000040        C3              ret
```

8.1 C调用惯例

我们注意到的第一件事情是参数按照相反的顺序压入了堆栈。这是C的调用惯例。在32位的C程序中遵循这个调用惯例。在下面的描述中，调用者指调用一个函数的函数，被调用者指被一个函数调用的函数。

- 调用者将函数的参数压入栈中，一个接着一个，按照逆序（右边的在左边，所以传递给函数的第一个参数最后一个压栈）。
- 调用者执行一个near CALL将控制权传递给被调用者。
- 被调用者获得控制权后，一般首先会（虽然在那些不需要访问传递给他们的参数的函数中这并不是必须的）保存在EBP中保存ESP的值，这样就可以使用EBP做为基指针去查找保存在堆栈中的传递给他自己的参数了。但是，调用者也要适当的做一部分工作。所以一部分的调用惯例说明EBP在任何的C函数中一定要保存下来。因此，作为被调用者，如果它准备将EBP作为一个frame pointer（这是什么东东？）设置的话，就必须首先将寄存器中当前数值压栈。
- 被调用者然后可以通过参考EBP访问传递给它的参数。[EBP]保存的双字保存了EBP压入栈中时的EBP数值；接下来的位于[EBP+4]的双字，保存了返回地址，当CALL调用后立刻就被压入栈中的。接下来，就是传递给函数的参数了，从[EBP+8]的地址开始。传递给函数的参数中最左边参数，由于它是最后一个压入栈中，所以可以通过EBP的这个偏移量访问到它。然后别的参数紧紧的逆着存储在他的前方，只是偏移量一个比一个大（使用[EBP+offset]访问）。因此，在一个例如printf这样有一大堆参数的的函数中，按照逆序将参数压栈意味着函数知道从哪里去发现它的第一个参数，这个参数表明了剩余参数的个数和类型。
- 被调用者可能也希望进一步的减少ESP，这样就可以在堆栈中为局部变量分配空间，这些相对于EBP来说，都是负的偏移量。
- 对于被调用者，如果他想将一个数值返回给调用者，可以根据数值长度不同将这个数值保存在AL，AX或者EAX中。浮点数的返回结果一般都保存在STO中。
- 一旦被调用者处理完毕之后，如果它分配了局部的堆栈空间，那它使用EBP将ESP恢复为调用前的原值（即EBP中的值）。然后将EBP之前的原数据推出堆栈，恢复到EBP中，再通过RET指令返回（等效于RETN）。
- 当调用者从被调用者处重新获得控制权后，函数的参数还仍然在堆栈中，所以一般会将ESP加上一个常量来将这些数值从堆栈中删除（用来替换一系列缓慢的POP指令）。因此，如果一个函数由于原型不匹配而通过错误个数的参数被调用了，堆栈仍然可以返回正确的

状态，因为调用者知道它压入栈中多少个参数，然后删除同样多个。

8.2 解剖

所以当 2 个字节被压入栈中以后，有一个对位于 0x1C 处的函数 f 的调用。这个函数首先为局部的用处减少 esp 4 个字节。然后函数函数将他的参数进行了本地赋值。之后 a＋b 被计算出来，并放在 eax 中被返回。

9，32 位栈对齐

请注意——即使当两个参数以字节为单位被压栈——函数从堆栈中将他们读出的时候，仍然是以双字（dword）为单位的！它看起来好像是处理器按照 32 位的模式按双字（dword）方式压栈的。这是因为栈被按照 32 位对齐的。了解这一点非常重要，尤其是当你不得不使用汇编程序编写遵循 C 调用惯例的 32 位函数时。

10，其它声明

当然我们还可以看看 GCC 是如何处理循环的，包括 for 循环，while 循环，if-else 循环和 case 结构等等。但是这并不影响你自己实现他们。当然，不过你不想使用汇编来实现他们的时候，也不必为此烦恼。

11，基本数据类型之间的转换

这部分我们将仔细研究一下 C 编译器对基本数据类型的转换。这些基本数据类型包括：

- singed char 和 unsigned char（1byte）
- singed short 和 singed char（2bytes）
- signed int 和 unsigned int（4 bytes）

首先我们看看计算机是如何处理有符号的数据类型的。

11.1 Two’s complement

11.1 2进制补码

在Intel的IA-32架构中使用2的补码的方式来表示有符号的整数。对于一个非负的整数n，他的2的补码形式就是这个数的2进制表示形式构成的位串。如果我们把位串的每一位都取反，然后将结果再加1，就得到-n的补码表示形式。对于在内存中使用2进制补码作为二进制表示形式的机器，我们成为二进制补码机器(two’s complement machines)。注意在二进制补码表示中0和-0都将使用同样的位串来表示。例如：

$$
\begin{aligned}
(\ 0)_{10} &= (00000000)_2 \\
(-0)_{10} &= (00000000)_2+1 \\
&= (11111111)_2+1 \\
&= (00000000)_2 \\
&= (\ 0)_{10}
\end{aligned}
$$

$(\ 0)_{10}$即以十进制表示的0。注意，对于负数的最高为被置为1。当然你不需要对某个特定的数自己来做负值转换。在IA-32架构中有一个特别的指令叫做NEG。表1展示了使用2进制补码表示的字符。使用二进制补码符号的优点是你可以象处理正数一样处理负数（减去一个整数可以使用加上一个负数的补码完成）。

	Range						
unsigned	128	...	255	0	1	...	127
signed	-128	...	-1	0	1	...	127

Table 1: The two’s complement of a char

11.2 赋值

下面我们将看看C语言中的赋值和生成的相应汇编代码，使用过的C程序显示在上面。

```
main () {
    unsigned int i = 251;
}
```

当我们编译后生成如下纯二进制文件：

```
00000000    55              push ebp
00000001    89E5            mov ebp,esp
00000003    83EC04          sub esp,byte +0x4
00000006    C745FCFB000000  mov dword [ebp-0x4],0xfb
0000000D    C9              leave
0000000E    C3              ret
```

当我们把用过的赋值语句换为：

```
unsigned int i = -5;
```

我们则得到在地址 0x6 的如下指令：

```
00000006 C745FCFBFFFFFF     mov dword [ebp-0x4],0xfffffffb
```

现在让我们看看这个有符号的寄存器。语句：

```
int i = 251;
```

的结果如下：

```
00000006 C745FCFB000000     mov dword [ebp-0x4],0xfb
```

对于使用负值的语句：

```
int i = -5;
```

生成的代码如下：

```
00000006 C745FCFBFFFFFF mov dword [ebp-0x4],0xfffffffb
```

看起来好像是有符号和无符号的数据都是按照同样的方法对待的。

11.3 signed char 到 signed int 的转换

这里我们研究一下下面的这个小程序：

```
main () {
    char c = -5;
    int i;
    i = c;
}
```

然后我们得到如下的二进制代码：

```
00000000    55          push ebp
00000001    89E5        mov ebp,esp
00000003    83EC08      sub esp,byte +0x8
00000006    C645FFFB    mov byte [ebp-0x1],0xfb
0000000A    0FBE45FF    movsx eax,byte [ebp-0x1]
0000000E    8945F8      mov [ebp-0x8],eax
00000011    C9          leave
00000012    C3          ret
```

解剖：

首先我们看到在地址 0x3 处，为存储局部变量 c 和 I 在栈上预留了 8 个字节。编译器使用 8 个字节使得他可以对齐整数 I。下面我们看到位于[ebp-0x1]的字符 c 被填充为 0xfb（代表-5，因为 0xfb=251，251-256=-5）。注意到编译器使用[ebp-0x1]替代[ebp-0x4]。这是因为这是使用 little endian 表示，所以低字节在前。下一个指令 movsx 进行了对于 signed char 到 signed int 的实际转换工作。MOVSX 指令以保留符号的方式将源操作数扩展为目标操作数的长度，然后将结果复制到目标操作数。最后一个指令（leave 之前）将保存在 eax 中的有符号整数的数值保

存到 int i 中。

11.4 signed int 到 signed char 的转换

我们看看同上一节相反的转换情景。代码如下：

```
main () {
    char c;
    int i = -5;
    c = i;
}
```

注意到语句 c=i 只有在 i 的数值在-128 到 127 之间的时候才有意义，因为 c 的范围定义在有符号 char 的范围内（否则就会发生溢出之类的错误）。

编译后得到如下二进制代码：

```
00000000    55              push ebp
00000001    89E5            mov ebp,esp
00000003    83EC08          sub esp,byte +0x8
00000006    C745F8FBFFFFFF  mov dword [ebp-0x8],0xfffffffb
0000000D    8A45F8          mov al,[ebp-0x8]
00000010    8845FF          mov [ebp-0x1],al
00000013    C9              leave
00000014    C3              ret
```

解剖：

0xfffffffb 就是-5。当我们只看低字节的 0xfb 时，如果把它赋给一个字节，我们仍然可以得到-5。因此将一个 signed int 转换成 signed char，我们只需要使用一个简单的 mov 指令即可。

11.5 unsigned char 到 unsigned int 的转换

看看下面这个 C 程序：

```
main () {
    unsigned char c = 5;
    unsigned int i;
    i = c;
}
```

我们将得到如下的二进制代码：

```
00000000    55          push ebp
00000001    89E5        mov ebp,esp
00000003    83EC08      sub esp,byte +0x8
```

```
00000006    C645FF05    mov byte [ebp-0x1],0x5
0000000A    0FB645FF    movzx eax,byte [ebp-0x1]
0000000E    8945F8      mov [ebp-0x8],eax
00000011    C9          leave
00000012    C3          ret
```

解剖：

我们的到了同 signed char 转换为 signed int 几乎一样的二进制代码，除了位于 0xA 处的指令。这里我们发现了指令 movzx。MOVZX 指令将他的源操作数 0 扩展为他的目标操作数的长度（即不保留最高位的符号属性），然后将结果复制到目标操作数中。

11.6 unsigned int 到 unsigned char 的转换

我满看看这个程序：

```
main () {
    unsigned char c;
    unsigned int i = 251;
    c = i;
}
```

请再一次注意这里的整型数值需要将范围限制在 0 到 255 之间，这是因为一个 unsigned char 类型不能处理更到的数字。这段代码生成的二进制代码如下：

```
00000000    55              push ebp
00000001    89E5            mov ebp,esp
00000003    83EC08          sub esp,byte +0x8
00000006    C745F8FB000000  mov dword [ebp-0x8],0xfb
0000000D    8A45F8          mov al,[ebp-0x8]
00000010    8845FF          mov [ebp-0x1],al
00000013    C9              leave
00000014    C3              ret
```

解剖：

实际中的转换指令，是位于 0xD 的 mov 指令，这同将 signed int 转换为 signed char 是一样的。

11.7 singed int 到 unsigned int 的转换

C 代码如下：

```
main () {
    int i = -5;
    unsigned int u;
```

```
    u = i;
}
```

对应的二进制代码如下：

```
00000000    55              push ebp
00000001    89E5            mov ebp,esp
00000003    83EC08          sub esp,byte +0x8
00000006    C745FCFBFFFFFF  mov dword [ebp-0x4],0xfffffffb
0000000D    8B45FC          mov eax,[ebp-0x4]
00000010    8945F8          mov [ebp-0x8],eax
00000013    C9              leave
00000014    C3              ret
```

解剖：

在 signed int 和 unsigned int 之间转换时没有特定的规则（这句话说的比较随意了，原文是 no specific conversion）。唯一的不同就是当你对有符号的数进行乘除操作时，要使用 idiv 和 imul 指令，对于无符号数，使用 div 和 mul。

12 GCC 编译的代码的基本环境

由于我找不到任何关于这个主题的官方文档，所以我试着自己去解释它，下面是我的结论：

- 32 位模式，所以是需要在 GDT 和 LDT 表中使能 32 位标志的保护模式（这句话可得验证验证）。
- 段寄存器 CS，DS，ES，FS，GS 和 SS 不得不指向同样的内存区域。
- 由于未初始化的全局变量被保存在紧跟着代码的地方，因此你必须保存一小段空闲的区域。这段区域被称为 BSS 段。注意在二进制文件中被初始化的全局变量保存在代码段后面的数据段（DATA section）。被定义为常量的全局变量被保存在只读数据段（RODATA），这也是二进制文件的一部分。
- 要确保栈不会覆盖了代码段和全局变量。

在 Intel 的文档中他们将其成为基本的扁平模式（Basic Flat Model）。不要误解。我们不一定非要使用 Basic Flat Model。由于 C 编译器将二进制代码的 CS，DS 和 SS 指向同样的内存区域（使用别名），没有任何问题。所以我们可以使用完全的多段保护页模式（full multisegment protected paging mod），鉴于每一个 C 编译的二进制代码拥有他们自己的局部 basic flat memory 模式。

13 对全局变量的扩展访问

这里我们将看看如何不通过 C 程序来访问全局 C 变量。这对于你使用一个程序（用汇编写的）

去加载一个 C 程序时，尤其是需要初始化一些 C 程序中的全局变量的时候，是非常有用的。当然我们也可以通过 C 程序的栈来传递变量，但保存这些变量的栈并不是我们要讨论的目的。我们可以在内存的一个固定的地方生成一个全局变量表——这样 C 程序将这个表赋给固定的地址——但是那样我们就不得不使用愚蠢的指针去访问这个表。下面就是我们将要做的事情，

test.c：

```
int myVar = 5;
int main () {
}
```

我们使用如下命令编译：

```
gcc -c test.c
ld -Map memmap.txt -Ttext 0x0 -e main -oformat binary -N \
-o test.bin test.o
ndisasm -b 32 test
```

得到如下二进制代码：

```
00000000    55      push ebp
00000001    89E5    mov ebp,esp
00000003    C9      leave
00000004    C3      ret
00000005    0000    add [eax],al
00000007    00      db 0x00
00000008    05      db 0x05
00000009    0000    add [eax],al
0000000B    00      db 0x00
```

如你所见变量 myVar 保存在位置 0x8。现在我们不得不使用它自己的内存映射文件 memmap.txt 来从 ld 中获取获取这个地址。这个映射文件 memmap.txt 使我们使用-Map 参数生成的。这里我们使用如下命令：

```
cat memmap.txt | grep myVar | grep -v ' \.o' | \
sed ' s/ *//' | cut -d' ' -f1
```

这样我们得到 test.o 中的变量 myVar 的地址：

```
0x00000008
```

当我们将这个数值放入一个环境变量（UNIX）MYVAR，我们可以使用这个去告诉 nasm 去哪里查找全局 C 变量 myVar，例如：

```
nasm -f bin -d MYVAR_ADDR=$MYVAR -o init.bin init.asm
```

在 init.asm 中直接使用这个信息的代码看起来如下所示：

```
...
mov ax,CProgramSelector
mov es,ax
mov eax,[TheValueThatMyVarShouldContain]
mov [es:MYVAR_ADDR],eax
```

...

13.1 BSS 段的大小

当 C 程序是一个 kernel 的时候，为了进行内存管理，他必须知道自己的 BSS 段到底有多大。这个大小同样可以从文件 memmap.txt 中提取出来。这里我们使用：

```
cat memmap.txt | grep ' \.bss ' | grep -v ' \.o' | sed ' s/.*0x/0x/'
```

在我们的例子 test.c 中，我们可以得到：

```
0x0
```

我们可以按照上面访问全局变量的方法传递这个数值。

13.2 全局静态变量

在 C 中没有办法直接访问静态变量。这是因为他们被声明为静态的。对于被描述为 extern 的标量，也是这样。当一个全局变量被声明为静态时，在通过连接器 ld 生成的内存映射文件中就不会有静态变量的地址。所以我们没有办法判定这个变量的地址。关键字 static 提供给给了我们一个很好的保护机制。

14 IA-32 上的 ANSI C stdarg.h 的实现

这个头文件给编程者提供了可以移植的定义信息，例如编写一个带有大量参数的 printf 函数时。这个头文件包含了一个 typedef 和三个宏。具体实现方式是同系统相关的，在 IA-32 下面可能的实现如下：

```
#ifndef STDARG_H
#define STDARG_H
/*字符串*/
typedef char* va_list;
/*进行 32 位的对齐截位*/
#define va_rounded_size(type) \
(((sizeof (type) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))
/*ap 指向 valist 的最后，也就是第一个参数？ */
#define va_start(ap, v) \
((void) (ap = (va_list) &v + va_rounded_size (v)))
/*从栈中按照给定的 type 取出一个数来？ */
#define va_arg(ap, type) \
```

```
(ap += va_rounded_size (type), *((type *)(ap - va_rounded_size (type))))
#define va_end(ap) ((void) (ap = 0))
#endif
```

在宏 va_start 中，变量 v 是针对你的函数定义中的型参定义的最后一个参数，也就是参数们的头。这个变量不属于存储级寄存器（storage class register？），也不是任何的数组类型，或者类似于 char 这样自动转换以扩展的类型。宏 va_start 初始化了参数指针 ap。宏 va_arg 访问列表中的下一个型参。宏 va_end 进行清理工作，如果在函数退出之前需要的话。在给定的实现中我们使用了宏 va_rounded_size。这个宏是因为 IA-32 的栈对齐才引入的——参数通过栈传递给一个函数——在 32 位的分界线上（即访问栈的时候，必须是 32 字节对齐的方式访问），即被声明为 sizeof(int)。

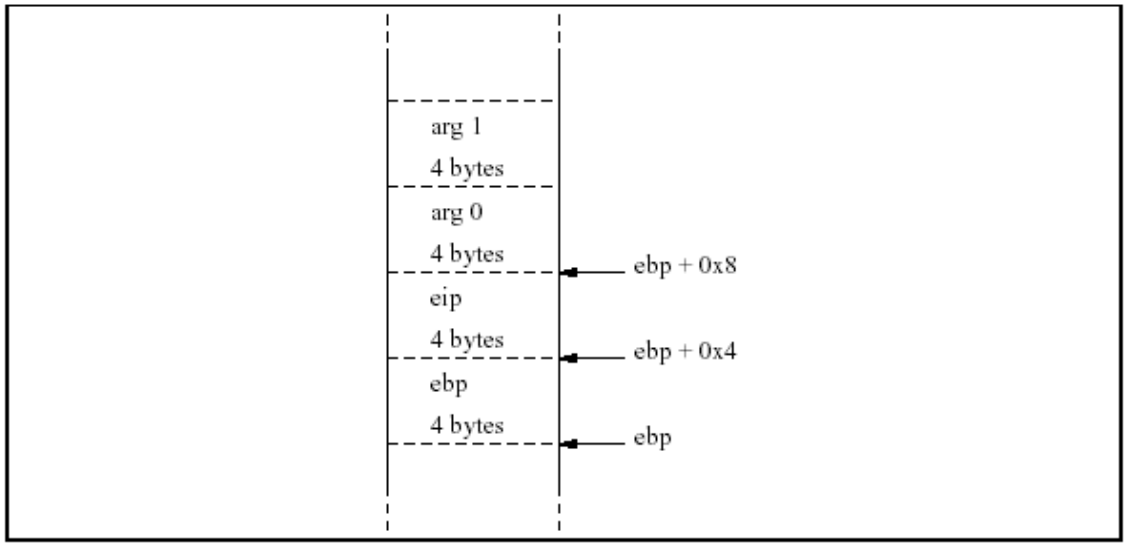

Figure 2: The arguments on the IA-32 stack

宏 va_start 将会让型参指针 ap 指向给定的第一个变量 v 的后面。这个宏不会返回任何东西（因为以 void 开头）。宏 va_arg 首先根据指定的类型 type 增加型参指针 ap ，之后它以类型 type 返回堆栈上的下一个型参（从物理位置上来说，是前一个型参，因为栈是向下增长的）。第一眼看上去这种处理方法非常的怪异，但是由于我们不得不将需要返回的变量放在一个宏定义的末尾（在最后一个逗号之后？什么意思），所以这是我们唯一的办法。最后宏 va_end 将会将型参指针 ap 复位，而不返回任何东西。

参考文献：

[1] A Book on C
Programming in C, fourth edition

Addison-Wesley— ISBN 0-201-18399-4
[2] Intel Architecture Software Developer’s Manual
Volume 1: Basic Architecture
Order Number: 243190
Volume 2: Instruction Set Reference Manual
Order Number: 243191
Volume 3: System Programming Guide
Order Number: 243192
[3] NASM documentation
http://www.cryogen.com/Nasm
[4] Manual Pages
gcc, ld, objcopy, objdump