C语言程序在计算机是怎样工作的了？

想必很多的童鞋都学习过C语言吧，应该也编写过C语言，那么对于C语言程序在计算机里面是怎么工作的了，一定很好奇吧，今天我抛砖引玉，编写一个简单的C语言小程序，跟大家一起分析下，C语言程序在计算机是怎样工作的。（主要的方式是通过汇编 C 语言程序代码，并分析汇编代码来理解程序的执行过程）。

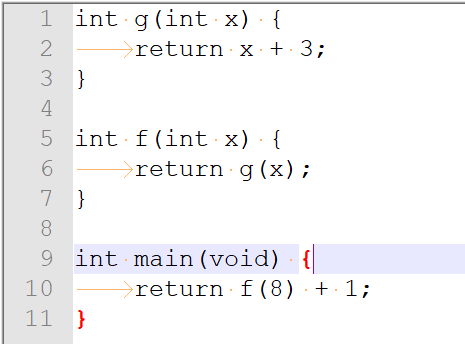


图1-12 简单的C语言程序

这时我们就可以编译 main.c 这个代码文件了。直接编译可以使用如下命令：

gcc main.c

会生成一个目标文件 a.out，它是可以执行的，但执行效果没有任何输出信息。可以通 过如下命令查看一下这个程序的返回值，如图 1-12 所示。

echo $？

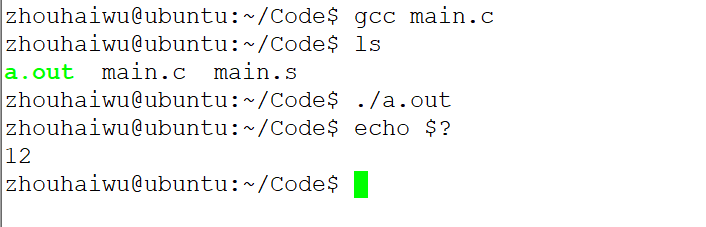


图1-13 简单的C语言程序

我们通过如下命令：

gcc –S –o main.s main.c –m32

产生一个以“.s”作为扩展名的汇编代码文件 main.s，在 vim 中，通 过“g/\.s\*/d”命令即可删除所有以“.”打头的字符串，就获得了“干净”的汇编代码:

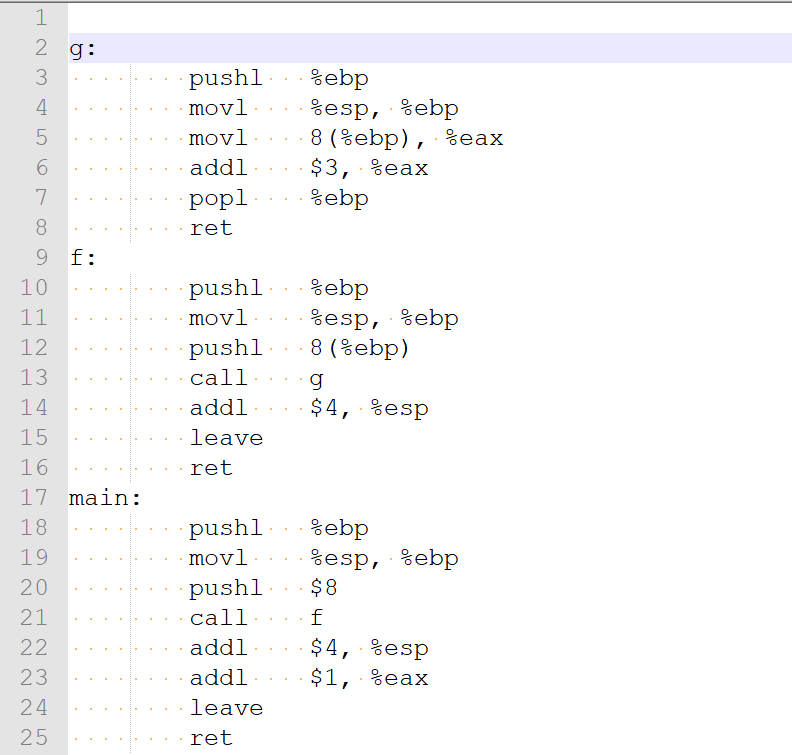


图1-14 C语言对应的汇编语言程序

首先我们夯实一下基础知识：

1、leave指令用来撤销函数堆栈，等价于下面两条指令：

movl %ebp,%esp

popl %ebp

2、另外，enter 指令用来建立函数堆栈，等价于下面两条指令：

pushl %ebp

movl %esp， %ebp

enter 指令的作用就是再堆起一个空栈，后面介绍函数调用堆栈时会进行详细介绍。而 leave 指令就是撤销这个堆栈，和 enter 指令的作用正好相反。

3、EIP 寄存器是指向代码段中的一条条指令，即 main.s 中的汇编指令，从“main:”开始，它 会自加一，调用 call 指令时它会修改 EIP 寄存器。EBP 寄存器和 ESP 寄存器也特别重要， 这两个寄存器总是指向一个堆栈，EBP 指向栈底，而 ESP 指向栈顶。注意，栈底是一个相 对的栈底，每个函数都有自己的函数堆栈和基地址。另外，EAX 寄存器用于暂存一些数值， 函数的返回值默认使用 EAX 寄存器存储并返回给上一级调用函数。

代码在执行过程中，堆栈空间和相应的 EBP/ESP 寄存器会不断变化。首先假定堆栈为空栈的情况下EBP和ESP寄存器都指向栈底，为了简化起见，我们为栈空间的存储单元进行标号，压栈时标号加1，出栈时标号减1，这样更清晰一点。需要注意的是，x86体系结构栈地址是向下增长的（地址减小），但这里只是为了便于知道堆栈存储单元的个数大小，

栈空间的存储单元标号是逐渐增大的。如图 1-15 所示，右侧的数字表示内存地址，EBP 和 ESP 寄存器都指向栈底，即指向一个 4 字节存储单元的下边缘 2000 的位置，指 2000～2003 这 4 个字节，也就是标号为 0 的存储单元，依此类推，标号 1 的存储单元为 1996～1999 这 4 个字节。

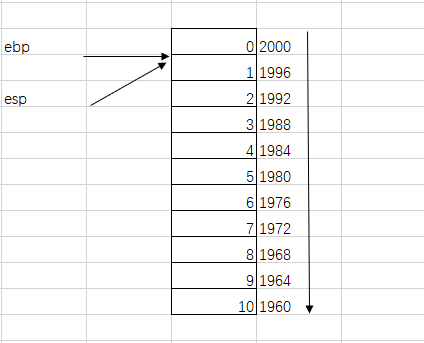


图1-15 堆栈空间示意图

程序从 main 函数开始执行，即上述代码的第 18 行，也就是“main:”下面的第一条汇 编指令“pushl %ebp”，这是开始执行的第一条指令，这条指令的作用实际上就是把 EBP 寄 存器的值（可以理解为标号 0，实际上是图 1-15 中的地址 2000）压栈，pushl 指令的功能 是先把 ESP 寄存器指向标号 1 的位置，即标号加 1 或地址减 4（向下移动 4 个字节），然后 将 EBP 寄存器的值标号 0（地址 2000）放到堆栈标号 1 的位置。

开始执行上一条指令时，EIP 寄存器已经自动加 1 指向了上述代码第 19 行语句“movl %esp,%ebp”，是将 EBP 寄存器也指向标号 1 的位置，这条语句只修改了 EBP 寄存器，栈 空间的内容并没有变化。第 18 行和第 19 行语句是建立 main 函数自己的函数调用堆栈空间。

开始执行上一条指令时，EIP寄存器已经自动加1指向了上述代码的第20行”pushl $8”， 将立即数 8 压栈（即先把 ESP 寄存器的值减 4，然后把立即数 8 放入当

前堆栈栈顶位置）。该指令等价于：

subl $4, %esp

movl $8, (%esp)

把 ESP 寄存器减 4，实际上是 ESP 寄存器向下移动一个标号，指向标号 2 的位置。把立即数 8 放入 ESP 寄存器指向的标号 2 位置。这条语句修改了ESP 寄存器，栈空间也发生了变化。第 20语句是在为接下来调用 f 函数做准备，即压栈 f 函数所需的参数。

开始执行上一条指令时；EIP 寄存器已经自动加 1 指向了上述代码的第 22 行指令“call f”，call 指令我们仔细分析过，第 22 行指令相当于如下两条伪指令：

pushl %eip(\*)

movl f %eip(\*)

第 21 行语句“call f”开始执行时，EIP 寄存器已经自加 1 指向了下一条指令，即上述 代码的第 22 行语句，实际上把 EIP 寄存器的值（行号为 22 的指令地址，我们用行号 22 表 示）放到了栈空间标号 3 的位置。因为压栈前 ESP 寄存器的值是标号 2，压栈时 ESP 寄存 器先减 4 个字节，即指向下一个位置标号 3，然后将 EIP 寄存器的行号 22 入栈到栈空间标 号 3 的位置。接着将 f 函数的第一条指令的行号 10 放入 EIP 寄存器，这样 EIP 寄存器指向 了 f 函数。这条语句既改变了栈空间，又改变了 ESP 寄存器，更重要的是它改变了 EIP 寄 存器。读者会发现原来 EIP 寄存器自加 1 指令是按顺序执行的，现在 EIP 寄存器跳转到了 f 函数的位置。

接着开始执行 f 函数。首先执行第 10 行语句“pushl %ebp”，把 ESP 寄存器的值向下移 一位到标号 4，然后把 EBP 寄存器的值标号 1 放到栈空间标号 4 的位置。

第 11 行语句“movl %esp, %ebp”是让 EBP 寄存器也和 ESP 寄存器一样指向栈空间标 号 4 的位置。

读者可能会发现，第 10 行和第 11 行语句与第 18 行和第 19 行语句完全相同，而且 g 函 数的开头两行也是这两条语句。总结一下：所有函数的头两条指令用于初始化函数自己的 函数调用堆栈空间。

第 12 行语句”pushl 8(%ebp)”等价于下面三条语句：

subl $4, %esp

movl 8(%ebp), %eax

movl %eax, (%esp)

第一条语句把 ESP 寄存器减 4，即指向下一个位置栈空间的标号 5。第二条语句通过 EBP 寄存器变址寻址：EBP 寄存器的值加 8，当前 EBP 寄存器指向 标号 4 的位置，加 8 即再向上移动两个存储单元加两个标号的位置，实际所指向的位置就 是堆栈空间中标号 2 的位置。如上所述，标号 2 的位置存储的是立即数 8，也就是把立即数 8 放到了 EAX 寄存器中。第三条语句把 EAX 寄存器中存储的立即数 8 放到 ESP 寄存器现在所指的位置， 即第一条语句预留出来的栈空间标号 5 的位置。第12行语句的作用，实际上是将函数 f 的参数取出来，主要目的是为调用函数 g 做好参数入 栈的准备。

第 13 行语句是“call g”，与上文中调用函数 f 类似，将 ESP 寄存器指向堆栈空间标号6 的位置，把 EIP 寄存器的内容行号 15 放到堆栈空间标号 6 的位置，然后把 EIP 寄存器指 向函数 g 的第一条指令，即上述代码的第 3 行。

第 14 行语句“addl$4, %esp”是把 ESP 寄存器（esp = (int32\_t \*)(esp + 4)）加立即数 4，也就是把ESP寄存器移动到标号5的位置。

接下来执行函数 g，与执行函数 f 或函数 main 的开头完全相同。第 3 行语句就是先把 EBP 寄存器存储的标号 4 压栈，存到堆栈空间标号 7 的位置，此时 ESP 寄存器为堆栈空间 标号 7。

接下来的第 4 行语句让 EBP 寄存器也和 ESP 寄存器一样指向当前堆栈栈顶，即堆栈 空间标号 7 的位置，这样就为函数 g 建立了一个逻辑上独立的函数调用堆栈空间。

第 5 行语句“movl 8(%ebp), %eax”通过使用 EBP 寄存器变址寻址，EBP 寄存器加 8， 也就是在当前 EBP 寄存器指向的栈空间标号 7 的位置基础上向上移动两个存储单元指向标 号 5，然后把标号 5 的内容（也就是立即数 8）放到 EAX 寄存器中。实际上，这一步是将 函数 g 的参数取出来。

第 6 行语句是把立即数 3 加到 EAX 寄存器里，就是 8+3，EAX 寄存器为 11。

这时 EBP 和 ESP 寄存器都指向标号 7，EAX 寄存器为 11，EIP 寄存器为代码行号 6， 函数调用堆栈空间如图 1-16 所示。EBP 或 ESP＋栈空间的标号表示存储的是某个时刻的 EBP 或 ESP 寄存器的值，EIP＋代码行号表示存储的是某个时刻的 EIP 寄存器的值。

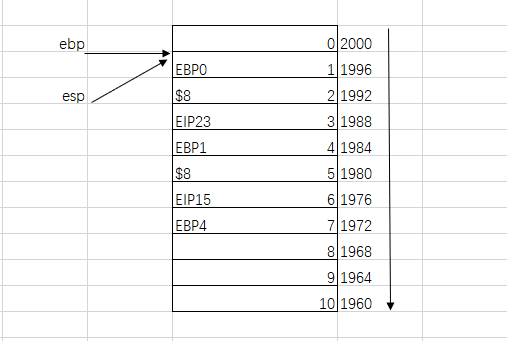


图1-16 执行到第6行代码时函数调用堆栈空间示意图

第 7 行和第 8 行语句的作用是拆除 g 函数调用堆栈，并返回到调用函数 g 的位置。第 6 行语句“popl %ebp”实际上就是把标号 7 的内容（也就是标号 4）放回 EBP 寄存器，也 就是恢复函数 f 的函数调用堆栈基址 EBP 寄存器，效果是 EBP 寄存器又指向原来标号 4 的位置，同时 ESP 寄存器也要加 4 个字节指向标号 6 的位置。

第 8 行语句“ret”实际上就是“popl %eip”，把 ESP 寄存器所指向的栈空间存储单元 标号 6 的内容（行号 15 即代码第 15 行的地址）放到 EIP 寄存器中，同时 ESP 寄存器加 4 个字节指向标号 5 的位置，也就是现在 EIP 寄存器指向代码第 15 行的位置。

这时开始执行第 15 行语句“leave”，如上所述，leave 指令用来撤销函数堆栈，等价 于下面两条指令：

movl %ebp,%esp

popl %ebp

结果是把 EBP 寄存器的内容标号 4 放到了 ESP 寄存器中，也就是 ESP 寄存器也指向 标号 4。然后，“popl %ebp”语句把标号 4 的内容（也就是标号 1）放回 EBP 寄存器，实际 上是把 EBP 寄存器指向标号 1 的位置，同时 ESP 寄存器加 4 个字节指向标号 3 的位置。

第 16 行语句“ret”是把 ESP 寄存器所指向的标号 3 的位置的内容（行号 23 即代码第 23 行指令的地址）放到 EIP 寄存器中，同时 ESP 寄存器加 4 个字节指向标号 2 的位置，也 就是现在 EIP 指向第 23 行的位置。

第 22 行语句“addl$4, %esp”是把 ESP 寄存器(esp = (int32\_t \*)(esp + 4))加立即数 4，也就是把ESP寄存器移动到标号3的位置。

第 23 行语句“addl$1, %eax”是把 EAX 寄存器加立即数 1，也就是 11+1，此时 EAX 寄存器的值为 12。EAX 寄存器是默认存储函数返回值的寄存器。

第 24 行语句“leave”撤销函数 main 的堆栈，把 EBP 和 ESP 寄存器都指向栈空间标 号 1 的位置，同时把栈空间标号 1 存储的内容标号 0 放到 EBP 寄存器，EBP 寄存器就指向 了标号 0 的位置，同时 esp 加 4 个字节，也指向标号 0 的位置。

这时堆栈空间回到了 main 函数开始执行之初的状态，EBP 和 ESP 寄存器也都恢复到开 始执行之初的状态指向标号 0。这样通过函数调用堆栈框架暂存函数的上下文状态信息，整 个程序的执行过程变成了一个指令流，从 CPU 中“流”了一遍，最终栈空间又恢复到空栈 状态。

上面就是C语言在机器里面的执行过程，大家清楚了吗？