

加餐 | 汇编代码编程与栈帧管理

2019-10-14 宫文学 来自北京

《编译原理之美》



在🔗22讲中，我们侧重讲解了汇编语言的基础知识，包括构成元素、汇编指令和汇编语言中常用的寄存器。学习完基础知识之后，你要做的就是多加练习，和汇编语言“混熟”。小窍门是查看编译器所生成的汇编代码，跟着学习体会。

不过，可能你是初次使用汇编语言，对很多知识点还会存在疑问，比如：

在汇编语言里调用函数（过程）时，传参和返回值是怎么实现的呢？

🔗21讲中运行期机制所讲的栈帧，如何通过汇编语言实现？

条件语句和循环语句如何实现？


.....

为此，我策划了一期加餐，针对性地讲解这样几个实际场景，希望帮你加深对汇编语言的理解。

示例 1：过程调用和栈帧


这个例子涉及了一个过程调用（相当于 C 语言的函数调用）。过程调用是汇编程序中的基础结构，它涉及到**栈帧的管理**、**参数的传递**这两个很重要的知识点。

假设我们要写一个汇编程序，实现下面 C 语言的功能：

 复制代码

```
1 /*function-call1.c */
2 #include <stdio.h>
3 int fun1(int a, int b){
4     int c = 10;
5     return a+b+c;
6 }
7
8 int main(int argc, char *argv[]){
9     printf("fun1: %d\n", fun1(1,2));
10    return 0;
11 }
```

fun1 函数接受两个整型的参数：a 和 b，来看看这两个参数是怎样被传递过去的，手写的汇编代码如下：

 复制代码

```
1 # function-call1-craft.s 函数调用和参数传递
2 # 文本段,纯代码
3 .section    __TEXT,__text,regular,pure_instructions
4
5 _fun1:
6 # 函数调用的序曲,设置栈指针
7 pushq    %rbp          # 把调用者的栈帧底部地址保存起来
8 movq     %rsp, %rbp    # 把调用者的栈帧顶部地址,设置为本栈帧的底部
9
10 subq     $4, %rsp      # 扩展栈
11
12 movl     $10, -4(%rbp) # 变量c赋值为10, 也可以写成 movl $10, (%rsp)
13
14 # 做加法
15 movl     %edi, %eax     # 第一个参数放进%eax
16 addl     %esi, %eax     # 把第二个参数加到%eax,%eax同时也是存放返回值的寄存器
17 addl     -4(%rbp), %eax # 加上c的值
```

```

18     addq    $4, %rsp        # 缩小栈
19
20     # 函数调用的尾声,恢复栈指针为原来的值
21     popq    %rbp            # 恢复调用者栈帧的底部数值
22     retq                                # 返回
23
24     .globl  _main            # .global伪指令让_main函数外部可见
25 _main:                                ## @main
26
27     # 函数调用的序曲,设置栈指针
28     pushq   %rbp            # 把调用者的栈帧底部地址保存起来
29     movq    %rsp, %rbp      # 把调用者的栈帧顶部地址,设置为本栈帧的底部
30
31     # 设置第一个和第二个参数,分别为1和2
32     movl    $1, %edi
33     movl    $2, %esi
34
35     callq   _fun1            # 调用函数
36
37     # 为printf设置参数
38     leaq    L_.str(%rip), %rdi # 第一个参数是字符串的地址
39     movl    %eax, %esi       # 第二个参数是前一个参数的返回值
40
41     callq   _printf          # 调用函数
42
43     # 设置返回值.这句也常用 xorl %esi, %esi 这样的指令,都是置为零
44     movl    $0, %eax
45
46     # 函数调用的尾声,恢复栈指针为原来的值
47     popq    %rbp            # 恢复调用者栈帧的底部数值
48     retq                                # 返回
49
50     # 文本段,保存字符串字面量
51     .section  __TEXT,__cstring,cstring_literals
52 L_.str:                                ## @.str
53     .asciz   "Hello World! :%d \n"
54

```

需要注意，手写的代码跟编译器生成的可能有所不同，但功能是等价的，代码里有详细的注释，你肯定能看明白。

借用这个例子，我们讲一下栈的管理。在示例代码的两个函数里，有这样的固定结构：

```

1  # 函数调用的序曲,设置栈指针
2
3      pushq  %rbp          # 把调用者的栈帧底部地址保存起来
4      movq  %rsp, %rbp     # 把调用者的栈帧顶部地址, 设置为本栈帧的底部
5
6      ...
7
8      # 函数调用的尾声,恢复栈指针为原来的值
9      popq  %rbp          # 恢复调用者栈帧的底部数值

```


在 C 语言生成的代码中,一般用 `%rbp` 寄存器指向栈帧的底部,而 `%rsp` 则指向栈帧的顶部。**栈主要是通过 `push` 和 `pop` 这对指令来管理的**: `push` 把操作数压到栈里,并让 `%rsp` 指向新的栈顶, `pop` 把栈顶数据取出来,同时调整 `%rsp` 指向新的栈顶。

在进入函数的时候,用 `pushq %rbp` 指令把调用者的栈帧地址存起来(根据调用约定保护起来),而把调用者的栈顶地址设置成自己的栈底地址,它等价于下面两条指令,你可以不用 `push` 指令,而是运行下面两条指令:

```

1  subq  $8, %rsp          #把%rsp的值减8,也就是栈增长8个字节,从高地址向低地址增长
2  movq  %rbp, (%rsp)      #把%rbp的值写到当前栈顶指示的内存位置

```


 复制代码

而在退出函数前,调用了 `popq %rbp` 指令。它恢复了之前保存的栈指针的地址,等价于下面两条指令:

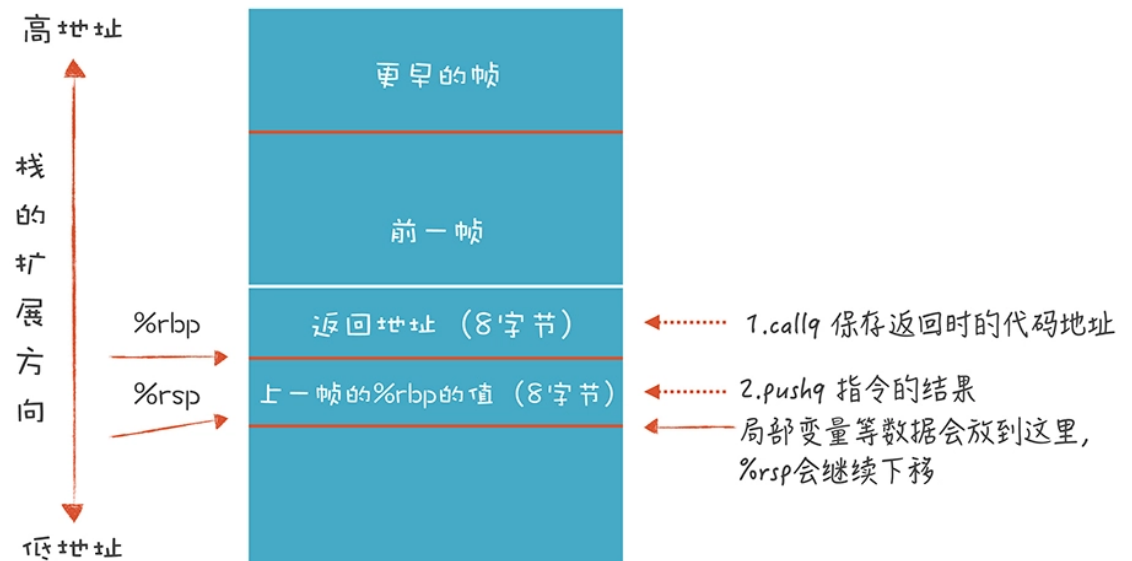
```

1  movq  (%rsp), %rbp      #把栈顶位置的值恢复回%rbp,这是之前保存在栈里的值。
2  addq  $8, %rsp          #把%rsp的值加8,也就是栈减少8个字节

```

 复制代码

上述过程画成一张直观的图,表示如下:



pushq %rbp调用完毕后的情况

上面每句指令执行以后，我们看看 %rbp 和 %rsp 值的变化：

指令	%rbp的值	%rsp的变化
callq _fun1	前一帧的值	原值减8字节，保存了一个返回地址
pushq %rbp	前一帧的值	再减8字节，保存了%rbp的值
movq %rsp, %rbp		
popq %rbp	前一帧的值	加8字节
ret	前一帧的值	回到原值

再看看使用局部变量的时候会发生什么：

复制代码

```

1    subq    $4, %rsp      # 扩展栈
2
3    movl    $10, -4(%rbp) # 变量c赋值为10，也可以写成 movl $10, (%rsp)

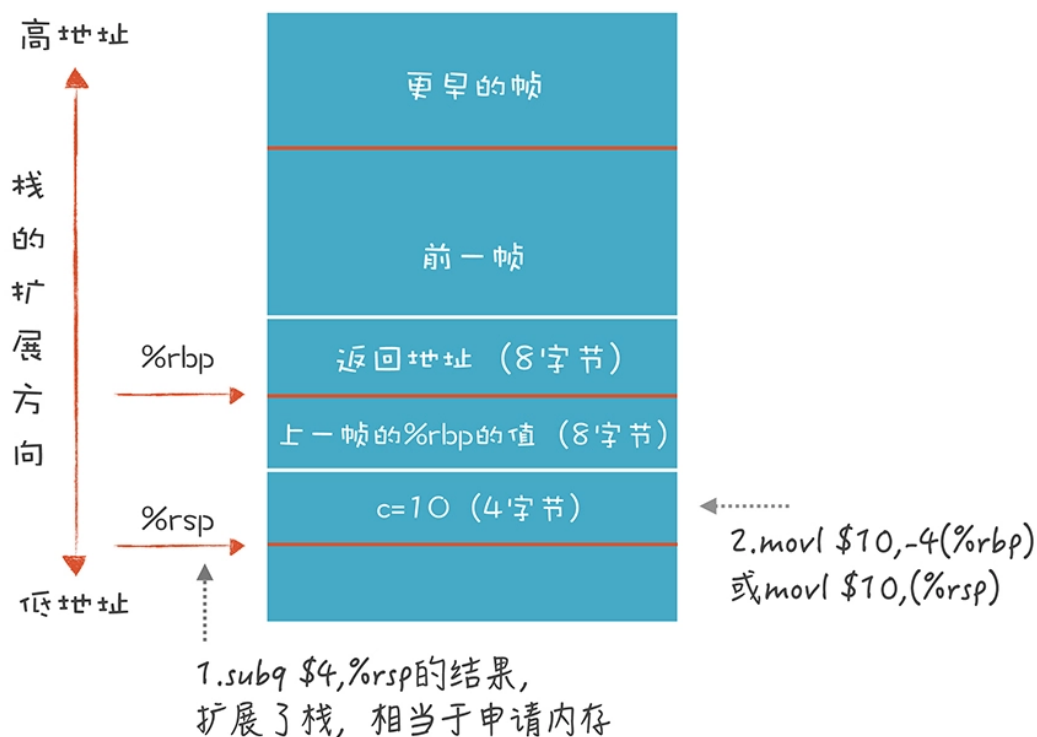
```

```

4      ...
5
6      addq    $4, %rsp      # 缩小栈
7

```

我们通过减少 `%rsp` 的值，来扩展栈，然后在扩展出来的 4 个字节的位置上写入整数，这就是变量 `c` 的值。在返回函数前，我们通过 `addq $4, %rsp` 再把栈缩小。这个过程如下图所示：



在这个例子中，我们通过移动 `%rsp` 指针来改变帧的大小。`%rbp` 和 `%rsp` 之间的空间就是当前栈帧。而过程调用和退出过程，分别使用 `call` 指令和 `ret` 指令。“`callq _fun1`”是调用 `_fun1` 过程，这个指令相当于下面两句代码，它用到了栈来保存返回地址：

```

1 pushq %rip # 保存下一条指令的地址，用于函数返回继续执行
2 jmp _fun1  # 跳转到函数_fun1

```

复制代码

`_fun1` 函数用 `ret` 指令返回，它相当于：

复制代码

```
1 popq %rip    #恢复指令指针寄存器
2 jmp %rip
```

上一讲，我提到，在 X86-64 架构下，新的规范让程序可以访问栈顶之外 128 字节的内存，所以，我们甚至不需要通过改变 %rsp 来分配栈空间，而是直接用栈顶之外的空间。

上面的示例程序，你可以用 as 命令生成可执行程序，运行一下看看，然后试着做一下修改，逐步熟悉汇编程序的编写思路。

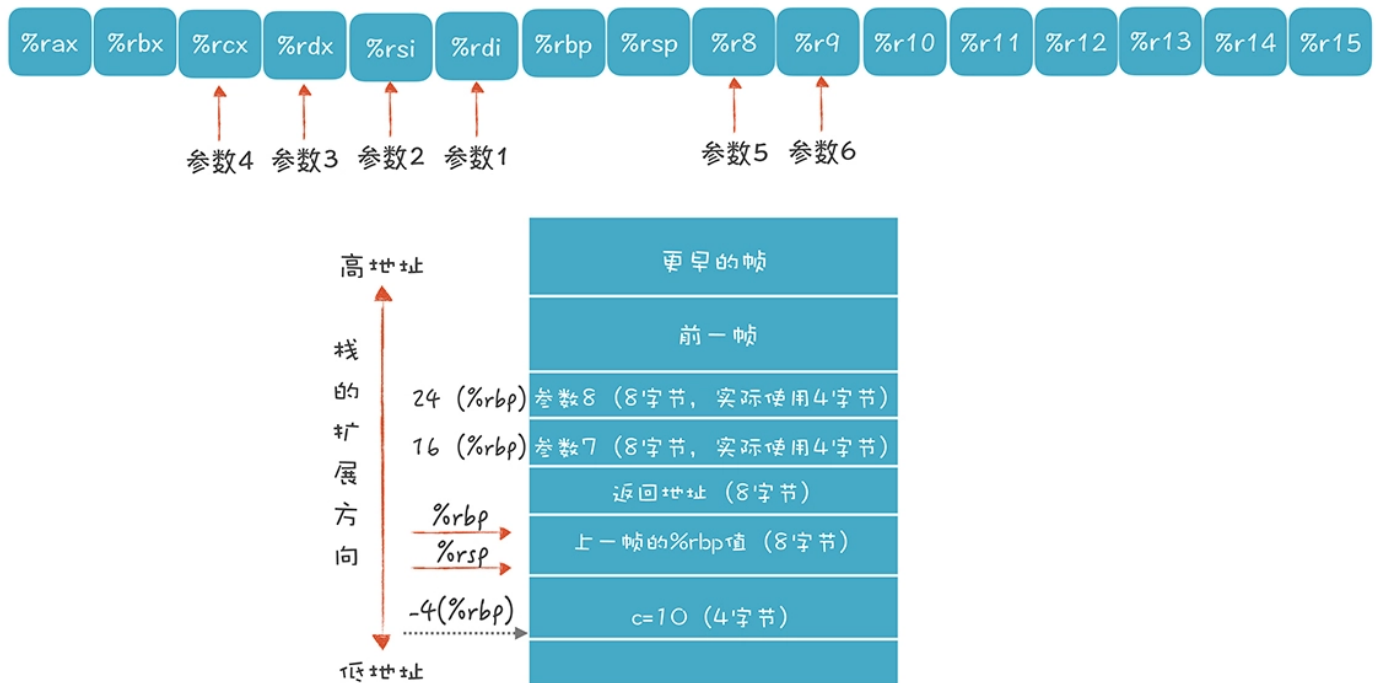
示例 2：同时使用寄存器和栈来传参

上一个示例中，函数传参只使用了两个参数，这时是通过两个寄存器传递参数的。这次，我们使用 8 个参数，来看看通过寄存器和栈传参这两种不同的机制。

在 X86-64 架构下，有很多的寄存器，所以程序调用约定中规定尽量通过寄存器来传递参数，而且，只要参数不超过 6 个，都可以通过寄存器来传参，使用的寄存器如下：

32位名称	64位名称	所传递的参数
%edi	%rdi	参数1
%esi	%rsi	参数2
%edx	%rdx	参数3
%ecx	%rcx	参数4
%r8d	%r8	参数5
%r9d	%r9	参数6

超过 6 个的参数的话，我们要再加上栈来传参：



根据程序调用约定的规定，参数 1 ~ 6 是放在寄存器里的，参数 7 和 8 是放到栈里的，先放参数 8，再放参数 7。

在 23 讲，我会带你为下面的一段 playscript 程序生成汇编代码：

复制代码

```

1 //asm.play
2 int fun1(int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8){
3     int c = 10;
4     return x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + c;
5 }
6
7 println("fun1:" + fun1(1,2,3,4,5,6,7,8));

```

现在，我们可以按照调用约定，先手工编写一段实现相同功能的汇编代码：

复制代码

```

1 # function-call2-craft.s 函数调用和参数传递
2 # 文本段,纯代码
3 .section __TEXT,__text,regular,pure_instructions
4
5 _fun1:

```



```

6      # 函数调用的序曲,设置栈指针
7      pushq    %rbp                # 把调用者的栈帧底部地址保存起来
8      movq     %rsp, %rbp          # 把调用者的栈帧顶部地址,设置为本栈帧的底部
9
10     movl     $10, -4(%rbp)        # 变量c赋值为10,也可以写成 movl $10, (%rsp)
11
12     # 做加法
13     movl     %edi, %eax           # 第一个参数放进%eax
14     addl     %esi, %eax           # 加参数2
15     addl     %edx, %eax           # 加参数3
16     addl     %ecx, %eax           # 加参数4
17     addl     %r8d, %eax           # 加参数5
18     addl     %r9d, %eax           # 加参数6
19     addl     16(%rbp), %eax       # 加参数7
20     addl     24(%rbp), %eax       # 加参数8
21
22     addl     -4(%rbp), %eax       # 加上c的值
23
24     # 函数调用的尾声,恢复栈指针为原来的值
25     popq     %rbp                # 恢复调用者栈帧的底部数值
26     retq                                # 返回
27
28     .globl   _main                # .global伪指令让_main函数外部可见
29 _main:                                ## @main
30
31     # 函数调用的序曲,设置栈指针
32     pushq    %rbp                # 把调用者的栈帧底部地址保存起来
33     movq     %rsp, %rbp          # 把调用者的栈帧顶部地址,设置为本栈帧的底部
34
35     subq     $16, %rsp            # 这里是为了让栈帧16字节对齐,实际使用可以更少
36
37     # 设置参数
38     movl     $1, %edi             # 参数1
39     movl     $2, %esi             # 参数2
40     movl     $3, %edx             # 参数3
41     movl     $4, %ecx             # 参数4
42     movl     $5, %r8d             # 参数5
43     movl     $6, %r9d             # 参数6
44     movl     $7, (%rsp)           # 参数7
45     movl     $8, 8(%rsp)          # 参数8
46
47     callq    _fun1                # 调用函数
48
49     # 为printf设置参数
50     leaq     L_.str(%rip), %rdi    # 第一个参数是字符串的地址
51     movl     %eax, %esi            # 第二个参数是前一个参数的返回值
52
53     callq    _printf              # 调用函数
54

```

```

55     # 设置返回值。这句也常用 xorl %esi, %esi 这样的指令,都是置为零
56     movl    $0, %eax
57
58     addq    $16, %rsp    # 缩小栈
59
60     # 函数调用的尾声,恢复栈指针为原来的值
61     popq    %rbp        # 恢复调用者栈帧的底部数值
62     retq
63
64     # 文本段,保存字符串字面量
65     .section    __TEXT,__cstring,cstring_literals
66 L_.str:
67     .asciz    "fun1 :%d \n"

```

用 `as` 命令,把这段汇编代码生成可执行文件,运行后会输出结果: “fun1: 46”。

 复制代码

```

1 as functio-call2-craft.s -o function-call2
2 ./function-call2

```


这段程序虽然有点儿长,但思路很清晰,比如,每个函数(过程)都有固定的结构。7~10 行,我叫做序曲,是设置栈帧的指针;25~26 行,我叫做尾声,是恢复栈底指针并返回;13~22 行是做一些计算,还要为本地变量在栈里分配一些空间。

我建议你读代码的时候,对照着每行代码的注释,弄清楚这条代码所做的操作,以及相关的寄存器和内存中值的变化,脑海里有栈帧和寄存器的直观的结构,就很容易理解清楚这段代码了。

除了函数调用以外,我们在编程时经常使用循环语句和 `if` 语句,它们转换成汇编是什么样子呢?我们来研究一下,首先看看 `while` 循环语句。

示例 3：循环语句的汇编码解析

看看下面这个 C 语言的语句：

 复制代码


```

1 void fun1(int a){

```

```
2     while (a < 10){
3         a++;
4     }
5 }
```

我们要使用"gcc -S ifstmt.c -o ifstmt.s"命令，把它转换成汇编语句（注意不要带优化参数）：

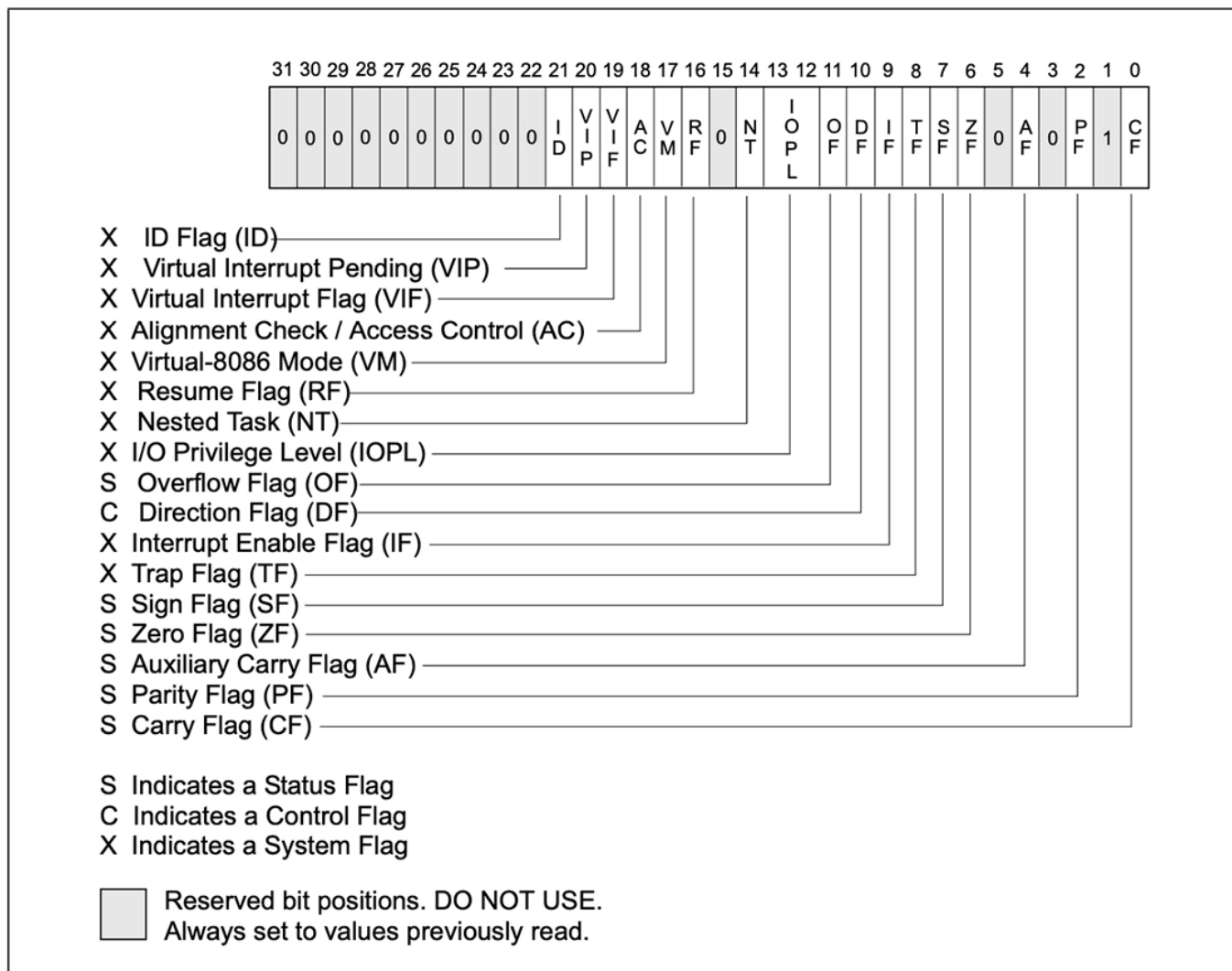
 复制代码

```
1  .section      __TEXT,__text,regular,pure_instructions
2      .macosx_version_min 10, 15
3      .globl    _fun1                      ## -- Begin function fun1
4      .p2align   4, 0x90
5  _fun1:                      ## @fun1
6      .cfi_startproc
7  ## %bb.0:
8      pushq    %rbp
9      .cfi_def_cfa_offset 16
10     .cfi_offset %rbp, -16
11     movq     %rsp, %rbp
12     .cfi_def_cfa_register %rbp
13     movl     %edi, -4(%rbp)    #把参数a放到栈里
14 LBB0_1:                      ## =>This Inner Loop Header: Depth=1
15     cmpl     $10, -4(%rbp)    #比较参数1和立即数10,设置eflags寄存器
16     jge     LBB0_3            #如果大于等于,则跳转到LBB0_3基本块
17  ## %bb.2:                      ## in Loop: Header=BB0_1 Depth=1
18     movl     -4(%rbp), %eax    #这2行,是给a加1
19     addl     $1, %eax
20     movl     %eax, -4(%rbp)
21     jmp     LBB0_1
22 LBB0_3:
23     popq     %rbp
24     retq
25     .cfi_endproc
26
27                                     ## -- End function
28 .subsections_via_symbols
```

这段代码的 15、16、21 行是关键，我解释一下：

第 15 行，用 `cmpl` 指令，将 `%edi` 寄存器中的参数 1（即 C 代码中的参数 a）和立即数 10 做比较，比较的结果会设置 `EFLAGS` 寄存器中的相关位。

`EFLAGS` 中有很多位，下图是 [Intel 公司手册](#) 中对各个位的解释，有的指令会影响这些位的设置，比如 `cmp` 指令，有的指令会从中读取信息，比如 16 行的 `jge` 指令：



第 16 行，`jge` 指令。`jge` 是 “jump if greater or equal” 的缩写，也就是当大于或等于的时候就跳转。大于等于是从哪知道的呢？就是根据 `EFLAGS` 中的某些位计算出来的。

第 21 行，跳转到循环的开始。

在这个示例中，我们看到了 `jmp`（无条件跳转指令）和 `jge`（条件跳转指令）两个跳转指令。条件跳转指令很多，它们分别是基于 `EFLAGS` 的状态位做不同的计算，判断是否满足跳转条件，看看下面这张表格：

指令	含义	状态码
je 或 jz	跳转，如果相等（等于零）	ZF
jne 或 jnz	跳转，如果不相等（不等于零）	~ZF，对ZF取反
js	跳转，如果为负值	SF
jns	跳转，如果不为负值	~SF
jg 或 jnle	跳转，如果大于，有符号数	~(SF^OF) & ~ZF
jge 或 jnl	跳转，如果大于等于，有符号数	~(SF^OF)
jl 或 jnge	跳转，如果小于，有符号数	SF^OF
jle 或 jng	跳转，如果小于等于，有符号数	SF^OF ZF
...		

表格中的跳转指令，是基于有符号的整数进行判断的，对于无符号整数、浮点数，还有很多其他的跳转指令。现在你应该体会到，汇编指令为什么这么多了。**好在其助记符都是有规律的，可以看做英文缩写，所以还比较容易理解其含义。**

另外我再强调一下，刚刚我让你生成汇编时，不要带优化参数，那是因为优化算法很“聪明”，它知道这个循环语句对函数最终的计算结果没什么用，就优化掉了。后面学优化算法时，你会理解这种优化机制。


不过这样做，也会有一个不好的影响，就是代码不够优化。比如这段代码把参数 1 拷贝到了栈里，在栈里做运算，而不是直接基于寄存器做运算，这样性能会低很多，这是没有做寄存器优化的结果。

示例 4：if 语句的汇编码解析

循环语句看过了，if 语句如何用汇编代码实现呢？


看看下面这段代码：

```
1 int fun1(int a){
2     if (a > 10){
```

 复制代码

```
3         return 4;
4     }
5     else{
6         return 8;
7     }
8 }
```

把上面的 C 语言代码转换成汇编代码如下：

 复制代码


```
1     .section      __TEXT,__text,regular,pure_instructions
2     .macosx_version_min 10, 15
3     .globl  _fun1          ## -- Begin function fun1
4     .p2align    4, 0x90
5     _fun1:                ## @fun1
6     .cfi_startproc
7     ## %bb.0:
8     pushq    %rbp
9     .cfi_def_cfa_offset 16
10    .cfi_offset %rbp, -16
11    movq     %rsp, %rbp
12    .cfi_def_cfa_register %rbp
13    movl     %edi, -8(%rbp)
14    cmpl     $10, -8(%rbp)  #将参数a与10做比较
15    jle     LBB0_2         #小于等于的话就调转到LBB0_2基本块
16    ## %bb.1:
17    movl     $4, -4(%rbp)   #否则就给a赋值为4
18    jmp     LBB0_3
19 LBB0_2:
20    movl     $8, -4(%rbp)   #给a赋值为8
21 LBB0_3:
22    movl     -4(%rbp), %eax #设置返回值
23    popq     %rbp
24    retq
25    .cfi_endproc
26
27                                ## -- End function
28    .subsections_via_symbols
```

了解了条件跳转指令以后，再理解上面的代码容易了很多。还是先做比较，设置 EFLAGS 中的位，然后做跳转。

示例 5：浮点数的使用


之前我们用的例子都是采用整数，现在使用浮点数来做运算，看看会有什么不同。

看看下面这段代码：

 复制代码

```
1 float fun1(float a, float b){
2     float c = 2.0;
3     return a + b + c;
4 }
```

使用 -O2 参数，把 C 语言的程序编译成汇编代码如下：

 复制代码

```
1  .section      __TEXT,__text,regular,pure_instructions
2      .macosx_version_min 10, 15
3      .section      __TEXT,__literal4,4byte_literals
4      .p2align      2                      ## -- Begin function fun1
5  LCPI0_0:
6      .long      1073741824                ## float 2 常量
7      .section      __TEXT,__text,regular,pure_instructions
8      .globl      _fun1
9      .p2align      4, 0x90
10     _fun1:                                ## @fun1
11         .cfi_startproc
12     ## %bb.0:
13         pushq      %rbp
14         .cfi_def_cfa_offset 16
15         .cfi_offset %rbp, -16
16         movq      %rsp, %rbp
17         .cfi_def_cfa_register %rbp
18         addss      %xmm1, %xmm0          #浮点数传参用XMM寄存器，加法用addss指令
19         addss      LCPI0_0(%rip), %xmm0  #把常量2.0加到xmm0上，xmm0保存返回值
20         popq      %rbp
21         retq
22         .cfi_endproc
23
24                                     ## -- End function
25     .subsections_via_symbols
```


这个代码的结构你应该熟悉了，栈帧的管理方式都是一样的，都要维护 `%rbp` 和 `%rsp`。不一样的地方，有几个地方：

传参。给函数传递浮点型参数，是要使用 XMM 寄存器。

指令。浮点数的加法运算，使用的是 `addss` 指令，它用于对单精度的标量浮点数做加法计算，这是一个 SSE1 指令。SSE1 是一组指令，主要是对单精度浮点数（比如 C 或 Java 语言中的 `float`）进行运算的，而 SSE2 则包含了一些双精度浮点数（比如 C 或 Java 语言中的 `double`）的运算指令。

返回值。整型返回值是放在 `%eax` 寄存器中，而浮点数返回值是放在 `xmm0` 寄存器中的。调用者可以从这里取出来使用。

课程小结

利用本节课的加餐，我带你把编程中常见的一些场景，所对应的汇编代码做了一些分析。你需要记住的要点如下：

函数调用时，会使用寄存器传参，超过 6 个参数时，还要再加上栈，这都是遵守了调用约定。

通过 `push`、`pop` 指令来使用栈，栈与 `%rbp` 和 `%rsp` 这两个指针有关。你可以图形化地记住栈的增长和回缩的过程。需要注意的是，是从高地址向低地址走，所以访问栈里的变量，都是基于 `%rbp` 来减某个值。使用 `%rbp` 前，要先保护起来，别破坏了调用者放在里面的值。

循环语句和 `if` 语句的秘密在于比较指令和有条件跳转指令，它们都用到了 `EFLAGS` 寄存器。

浮点数的计算要用到 MMX 寄存器，指令也有所不同。

通过这次加餐，你会更加直观地了解汇编语言，接下来的课程中，我会带你尝试通过翻译 AST 自动生成这些汇编代码，让你直观理解编译器生成汇编码的过程。

一课一思

你了解到哪些地方会使用汇编语言编程？有没有一些比较有意思的场景？是否实现了一些普通高级语言难以实现的结果？欢迎在留言区分享你的经验。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的朋友。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (19)



骨汤鸡蛋面

2020-02-22

老师，通过今天学习有以下总结：

1. pushq 和 popq 虽然是单“参数”指令，但一个隐藏的“参数”就是 %rsp。
2. 通过移动 %rsp 指针来改变栈的大小。%rbp 和 %rsp 之间的空间就是当前栈帧。
3. 栈帧先进后出（一个函数的相关信息占用一帧）。或者栈帧二字重点在帧上。%rbp 在函数调用时一次移动一个栈帧的大小，**%rbp在整个函数执行期间是不变的**。
4. 函数内部访问栈帧可以使用 `-4(%rbp)` 表示地址，表示 %rbp 寄存器存储的地址减去4的地址。说白了，**栈帧内可以基于 (%rbp) 随机访问**，`+4(%rsp)` 效果类似。
5. **%rsp并不总是指向真实的栈顶**：在 X86-64 架构下，新的规范让程序可以访问栈顶之外 128 字节的内存，所以，我们甚至不需要通过改变 %rsp 来分配栈空间，而是直接用栈顶之外的空间。比如栈帧大小是16，即 `(%rbp)-(%rsp) = 16`，可以在代码中直接使用内存地址 `-32(%rbp)`。但如果函数内还会调用其它函数，为了pushq/popq 指令的正确性，编译器会为%rsp 设置正确的值使其指向栈顶。
6. 除了callq/pushq/popq/retq 指令操作%rsp外，函数执行期间，可以mov (%rsp)使其指向栈顶一步到位，(%rsp)也可以和(%rbp)挨着一步不动，也可以随着变量的分配慢慢移动。

作者回复：你总结得很细，很清晰。都可以画出一张脑图了！

这些技术细节，可以找到相应的技术规格文档去阅读，并获得更多你感兴趣的内容。课程里讲的，主要是Unix家族的系统约定。Windows系统可能会不同。但你总能找到相应的技术文档来找到这些约定。

进一步，你可以再寻找下面几个问题的答案：

1. 为什么只要移动栈指针就能分配内存。应用程序的内存都是操作系统虚拟出来的，操作系统必须在你使用内存的时候，给你准备好真实的物理内存。对这个问题的回答就涉及一点操作系统的知识，我相信你能查到。
2. 在栈里申请内存，相比在堆里相比内存，哪个更快？哪个容易导致内存碎片？哪个更有利于局部

性？这两种内存申请方式各自的使用场景是什么？像Python这样的动态语言有可能使用栈吗？像Java这样的语言，在new一个对象的时候，有可能使用栈吗？
希望你能在寻找这些问题答案的时候，获得更多的收获。

或者，可能我会在第二季多少涉及这些内容。

共 3 条评论 >

👍 6



zKerry

2019-11-04

栈的扩大和缩小有点反直觉啊，为啥扩大是减，缩小是加

作者回复：因为栈是从高地址向低地址延伸的。所以地址减的话，才是栈的增长。



👍 6



初心、可曾記

2019-10-18

图中的%rbp应该是指向【上一帧的%rbp的值】的下方红线部位，不应该是【返回地址】的下方红线

作者回复：回头我更新一版图，让图中的箭头指向格子而不是线，这样更加没有歧义。

共 2 条评论 >

👍 5



if...else...

2021-10-20

不错，有收获

作者回复：昵称很赞！

共 2 条评论 >

👍 1



骨汤鸡蛋面

2020-02-20

有个疑惑点：函数调用返回时，一个函数的栈帧是作为一整个单位被丢弃掉嘛？

作者回复: 对的。栈顶指针重新赋值了, 栈顶外的内存就抛弃了。这就是一种很自动的内存管理机制, 比在堆里申请和释放内存要简单。

所以说在栈里声明的本地变量, 它的生存期跟作用域是一致的(闭包除外)。



👍 1



风

2019-11-05

局部变量的访问, 既可以用rbp-的方式, 也可以用rsp+的方式, 文中实例里, 都是rbp-的方式, 所以需要管理好rbp这个寄存器。

如果采用rsp+的方式, 是不是根本就不需要rbp这个寄存器了, 这样效率不就更高了?

我看到的一些ARM核, 里面只有rsp寄存器, 没有rbp寄存器, 这样是不是更好呢?

作者回复: 没错。用两个寄存器来标记栈帧, 确实有点浪费。实际上是可以优化掉的。

如果你用gcc编译的话, 可以使用-fomit-frame-pointer参数来生成汇编, 会把下面三行代码都去掉。

```
pushq %rbp
```

```
movq %rsp, %rbp
```

```
popq %rbp
```

我在34讲的一个例子中, 手工去掉了这三行代码, 生成的机器码可以少5个字节, 还少两次内存访问, 其中有一次是写操作, 高速缓存都帮不上忙。对于追求极致性能的程序来说, 这个优化是必要的。



👍 1



阿鼎

2019-10-18

协程的切换, 用户态代码要复制堆栈寄存器信息。也想请教老师, 协程调度是否只能在io线程呢? 非io线程能否用协程呢?

作者回复: 非io当然可以用协程。比如迭代器、状态机用协程来写就很优雅。

共 3 条评论 >

👍 1



沉淀的梦想

2019-10-14

老师在课中讲了不少“栈”的操作, 那编程语言对于“堆”又是用什么指令操作的呢?

作者回复: 鼓励你用c语言, 使用malloc和free来申请和释放内存, 看看生成的汇编是怎样的。

共 3 条评论 >

👍 1



pebble

2019-10-14

例一的俩栈帧图里, rbp跟rsp, 是否应该都指向再下一个位置呢, rsp指向的, 应该是下次要保存数据的位置吧

作者回复: 不是。

rbp, 指向栈底。这个值在整合函数执行期间是不变的。

rsp, 指向栈顶。这个值会在某些情况下改变:

(1)push和pop命令可以改变rsp。

(2)call指令, 因为要把返回地址压栈, 实际也改变了rsp。

(3)在使用本地变量时, 手工改变rsp的值。

rsp如果指向下次要保存数据的位置, 相当于栈里总有一个空单元。

共 2 条评论 >

👍 1



伟

2022-11-22 来自上海

arm架构的汇编指令不一样?



♀楠生♀

2022-11-01 来自广东

as functio-call2-craft.s -o function-call2样得到的是.o文件, 而不是可执行程序.执行./function-call2 会报

zsh: exec format error: ./function-call2



程序员班吉

2022-04-08

扩展栈和缩小栈, 使用的是一个立即数, 这个立即数表示的是将地址向上或者向下移动多少吗?





Geek_656245

2021-10-16

视屏流，音频流在汇编和机器码中又是什么样子的呢？



Geek_656245

2021-10-16

汇编转化成机器码又是什么样子的呢？



minghu6

2021-05-29

看一些汇编代码的例子总是觉得腰背酸痛，原理性不多，就是些具体规定而且高度平台特定，纯搬砖，给平台打工，看了真不带劲儿

作者回复：是，与汇编有关的工作，工程性更强一些。



favoorr

2020-12-09

这个第一次学的时候还真很难一次明白，最好是用 GDB 来单步，观察寄存器的值，一边单步，一边拿自己小本本记，来加深理解

作者回复：你讲的很对。最好善用GDB、LLDB这些调试工具，这样很多抽象的知识就变得可视化了！



__Unsafe

2020-10-22

_fun1: # 函数调用的序曲,设置栈指针

pushq %rbp # 把调用者的栈帧底部地址保存起来

movq %rsp, %rbp # 把调用者的栈帧顶部地址,设置为本栈帧的底部

movl \$10, -4(%rbp) # 变量c赋值为10,也可以写成 movl \$10, (%rsp)

这里不能写成movl \$10, (%rsp)吧，上一步movq %rsp %rbp已经把%rbp的值设为%rsp一样了

作者回复: 只要是等价的, 就可以。

只不过, 这种代码通常都是由编译器机械的生成的, 它会按照一个固定的套路来生成代码。



骨汤鸡蛋面

2020-02-22

在“同时使用寄存器和栈来传参”那一些小节, 且`%rsp`并未指向栈真实的最顶部, 那么当在`fun`里再调用其它方法时, 其它方法执行`pushq %rbp`, 会不会覆盖掉`-4(%rbp)`的值, 进而出错?

作者回复: 必须是叶子函数才可以。

下面是https://en.wikipedia.org/wiki/X86_calling_conventions

中摘的一段, 介绍System V AMD64 ABI的内容, 你可以看看:

For leaf-node functions (functions which do not call any other function(s)), a 128-byte space is stored just beneath the stack pointer of the function. The space is called red-zone. This zone will not be clobbered by any signal or interrupt handlers. Compilers can thus utilize this zone to save local variables. Compilers may omit some instructions at the starting of the function (adjustment of RSP, RBP) by utilizing this zone. However, other functions may clobber this zone. Therefore, this zone should only be used for leaf-node functions. gcc and clang offer the `-mno-red-zone` flag to disable red-zone optimizations.



沉淀的梦想

2019-10-14

老师的as用的什么版本, 为什么我用as汇编文稿中的代码会出错 (注释删了也一样会出错):

function-call2-craft.s: Assembler messages:

function-call2-craft.s:2: Error: no such instruction: `n-call2-craft.s 函数调用和参数传递'

function-call2-craft.s:4: Error: character following name is not '#'

function-call2-craft.s:66: Error: character following name is not '#'

作者回复: macOS操作系统, gnu汇编器。

Linux上生成的汇编码应该也差不多, 有些系统调用是不同的。

windows上, 你就要安装一下相关的环境了和工具了, 比如MinGW。或者装一个Linux虚拟机。

共 3 条评论>

