



下载APP



17 | 生成本地代码第2关：变量存储、函数调用和栈帧维护

2021-09-15 宫文学

《手把手带你写一门编程语言》

课程介绍 >

**讲述：宫文学**

时长 19:55 大小 18.26M



你好，我是宫文学。

在上一节课里，我们已经初步生成了汇编代码和可执行文件。不过，很多技术细节我还没有来得及给你介绍，而且我们支持的语言特性也比较简单。

那么，这一节课，我就来给你补上这些技术细节。比如，我们要如何把逻辑寄存器映射到物理寄存器或内存地址、如何管理栈帧，以及如何让程序符合调用约定等等。

好了，我们开始吧。先让我们解决逻辑寄存器的映射问题，这其中涉及一个简单的寄存器分配算法。



给变量分配物理寄存器或内存


在上一节课，我们在生成汇编代码的时候，给参数、本地变量和临时变量使用的都是逻辑寄存器，也就是只保存了变量的下标。那么我们要怎么把这些逻辑寄存器对应到物理的存储方式上来呢？

我们还是先来梳理一下实现思路吧。

其实，我们接下来要实现的寄存器分配算法，是一个比较初级的算法。你如果用 clang 或 gcc 把一个 C 语言的文件编译成汇编代码，并且不带 -O1、-O2 这样的优化选项，生成出来的汇编代码就是采用了类似的寄存器分配算法。现在我们就来看看这种汇编代码在实际存储变量上的特点。


首先，程序的参数都被保存到了内存里。具体是怎么来保存的呢？你可以先看看示例程序

[@param.c](#)：

 复制代码

```
1 void println(int a);
2
3 int foo(int p1, int p2, int p3, int p4, int p5, int p6, int p7, int p8){
4     int x1 = p1*p2;
5     int x2 = p3*p4;
6     return x1 + x2 + p5*p6 + p7*p8;
7 }
8
9 int main(){
10     int a = 10;
11     int b = 12;
12     int c = a*b + foo(a,b,1,2,3,4,5,6) + foo(b,a,7,8,9,10,11,12);
13     println(c);
14     return 0;
15 }
```

这个示例程序所对应的汇编代码是 [@param.s](#)，我摘取了其中的一部分，这段代码展示了参数是如何被保存到内存的：

 复制代码

```
1 ## 把参数值保存到内存
2 movl    %edi, -4(%rbp)
3 movl    %esi, -8(%rbp)
4 movl    %edx, -12(%rbp)
5 movl    %ecx, -16(%rbp)
```

```

6  movl    %r8d, -20(%rbp)
7  movl    %r9d, -24(%rbp)

```

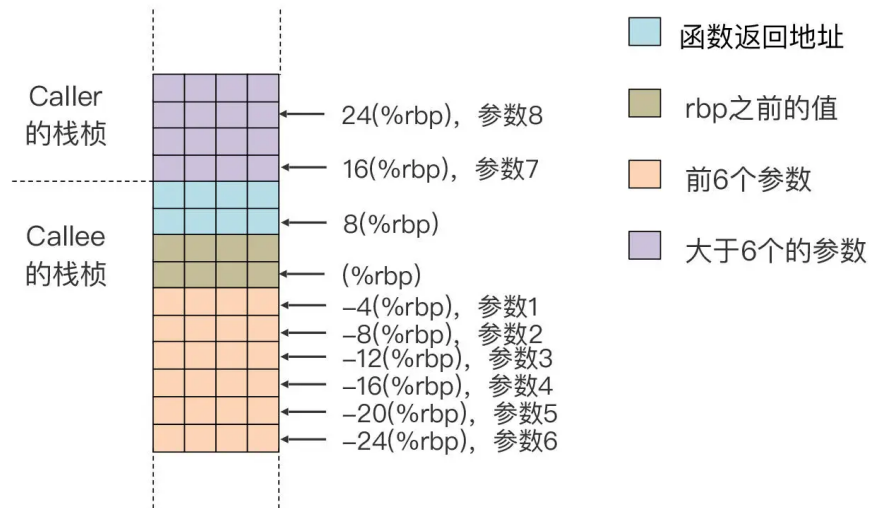
在这个示例程序中，foo 函数有 8 个参数。根据 [System V AMD64 的 ABI](#) 和 C 语言的调用约定，其中前 6 个参数是通过寄存器传递的，其他两个参数是通过栈帧传递的。

在汇编代码中，我们会发现这 6 个通过寄存器传递的参数，都先被保存到了栈帧中。这 6 个寄存器和参数的对应关系是这样的：

	参数1	参数2	参数3	参数4	参数5	参数6
32位	edi	esi	edx	ecx	r8d	r9d
64位	rdi	rsi	rdx	rcx	r8	r9



我也把它们保存在栈帧里的位置画成了一张图，你可以看一下：



你可以看到，这里面的前 6 个参数的位置，都是从 rbp 的位置依次向下 4 个字节，也就是一个整数的位置。参数 1 是 -4(%rbp)，参数 2 是 -8(%rbp)，依此类推。

而大于 6 个的参数，是保存在调用者的栈帧里的。其中参数 7 的地址是 $16(\%rbp)$ ，也就是 `rbp` 指针往上 16 个字节。这里为什么要加上 16 个字节呢？这是因为，这 16 个字节中，有 8 个字节是返回地址，是由 `callq _foo` 指令压到栈里的，还有 8 个字节是 `rbp` 之前的值，是由 `pushq rbp` 压到栈里的。

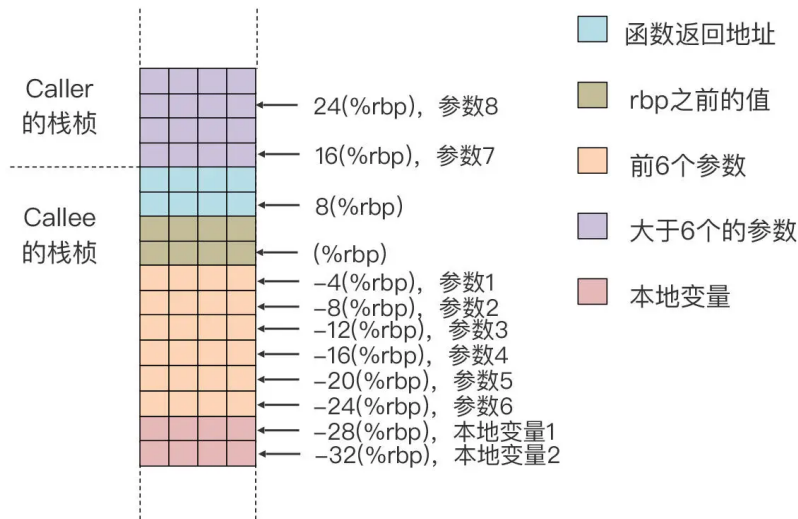
好，到目前为止，我们就知道如何在汇编代码里访问每个参数了。你可以查看代码库里的 [@lowerVars](#) 代码，它把每个参数转变成了一个内存地址类型的 `Oprand`。

[复制代码](#)

```
1 //处理参数
2 for (let varIndex:number = 0; varIndex<this.numTotalVars; varIndex++){
3     let newOprand:Oprand;
4     if (varIndex < this.numParams){
5         if (varIndex < 6){
6             //从自己的栈帧里访问。在程序的序曲里，就把这些变量拷贝到栈里了。
7             let offset = -(varIndex + 1) * 4;
8             newOprand = new MemAddress(Register.rbp,offset);
9         }
10        else{
11            //从Caller的栈里访问参数
12            //+16是因为有一个callq压入的返回地址，一个pushq rbp又加了8位
13            let offset = (varIndex - 6)*8 + 16;
14            newOprand = new MemAddress(Register.rbp,offset);
15        }
16    }
17    ...
18 }
```

接下来，我们再看看本地变量。在示例程序 [@param.c](#) 中，有 `x1` 和 `x2` 两个本地变量，你可以阅读 [@param.s](#)，看看它们都是存在哪的。

我相信，经过这么多节课的训练，现在你阅读汇编代码应该越来越熟练了。在这个代码里，你可以看到 `x1` 实际上是存在 $-28(\%rbp)$ 的，而 `x2` 是存在 $-32(\%rbp)$ 的。也就是说，在栈帧里，它们是紧挨着参数的区域继续往下延伸的，你可以看看下面这张图。



好了，现在对于参数和本地变量的保存，我们都搞清楚了，它们都是保存在内存的栈帧里的。那么临时变量呢？临时变量也保存在内存里吗？

这是不行的。为什么呢？这是因为，X86 指令集中加减乘除等指令和 mov 指令对内存地址的使用是有限制的。

怎么理解呢？我们分析一下 $x1 = p1 * p2$ 这条语句来看看。在上节课中，我们已经知道这条语句在生成汇编代码时，需要用到一个临时变量 t1，所以整条语句在逻辑上相当于下面这三条汇编代码：

```
1 movl p1, t1  #把p1拷贝到t1
2 imull p2, t1 #把p2乘到t1上
3 movl t1, x1  #把t1赋给x1
```

复制代码

在这里，p1 和 p2 都是内存地址。如果 t1 也是内存地址，我们假设它是 $-36(\%rbp)$ ，那么这三条汇编代码就会变成：

```
1 movl -4(%rbp), -36(%rbp) #把p1拷贝到t1
2 imull -8(%rbp), -36(%rbp) #把p2乘到t1上
3 movl -36(%rbp), -4(%rbp) #把t1赋给x1
```


复制代码

但是如果你用汇编器编译这三条代码，汇编器会报错。

这是为什么呢？因为 x86 中汇编代码的规则虽然比较宽松，在加减乘除等很多运算性指令里都支持使用内存地址作为操作数，但它只允许源操作数是内存地址，不允许目标操作数是内存地址。而 mov 指令支持目标操作数是内存地址，但这个时候源操作数必须是立即数或寄存器，不支持把数据从一个内存地址拷贝到另一个内存地址。

那么我们要怎么来修改这三条代码，让它变成合法的 X86 汇编代码呢？

很简单，我们只要给临时变量 t1 分配一个物理寄存器就可以了。你可以参考 [@param.s](#) 的代码，它给 t1 分配了一个 %ecx 寄存器，完美地解决了这个问题：

 复制代码

```
1 movl    -4(%rbp), %ecx    #把p1拷贝到t1
2 mull    -8(%rbp), %ecx    #把p2乘到t1上
3 movl    %ecx, -28(%rbp)  #把t1赋给x1
```

所以，现在我们就清楚了：**对于临时变量，我们都统一给它们分配寄存器就行了。**

那进一步的问题又来了：我们给这些临时变量分配哪些寄存器呢？是先分配那些由 Caller 保护的寄存器，还是 Callee 保护的寄存器？用哪类寄存器的代价更低呢？这几个问题你可以先记着，自己想一会，后面你看看示例代码是如何实现的。

好了，现在我们已经给参数、本地变量和临时变量都分配了物理的存储方式。这个过程，我们叫做把变量做 Lower 处理的过程，你可以看一下 [@lowerVars](#) 方法。

并且，基于我们上面这个分配方法，所有的二元计算都能生成正确的汇编代码。你可以在 [@example.ts](#) 中写一些表达式计算的代码，然后再用 make example 命令做构建。这会生成汇编代码文件 example.s 和可执行文件 example，你再看看它们能否正确地运行。

接下来，我们再研究一下如何实现函数的调用。

实现函数调用

我们仍然用示例代码 [param.c](#) 来做研究。在示例代码中，我们用 main 函数调用了 foo 函数。由于 foo 函数的参数比较多，所以我建议你好好看一下 [param.s](#) 中传参的具体过程。

[复制代码](#)

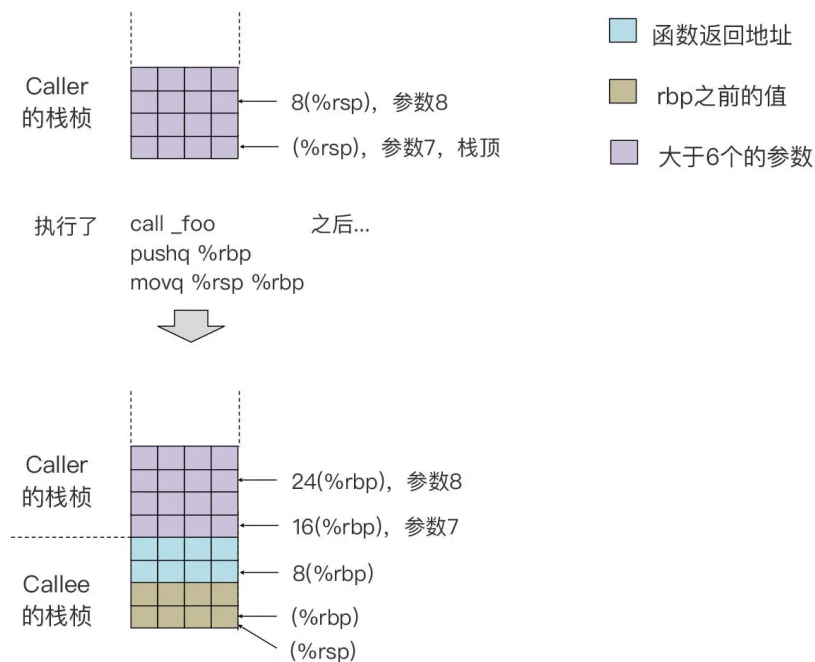
```

1  movl    -8(%rbp), %edi    ##参数1, 变量a
2  movl    -12(%rbp), %esi   ##参数2, 变量b
3  movl    $1, %edx         ##参数3
4  movl    $2, %ecx         ##参数4
5  movl    $3, %r8d         ##参数5
6  movl    $4, %r9d         ##参数6
7  movl    $5, (%rsp)       ##参数7
8  movl    $6, 8(%rsp)      ##参数8
9  ...
10 callq   _foo

```

根据我们目前掌握的参数传递的知识，这里的前 6 个参数赋给了 6 个寄存器，而第 7 和第 8 个参数，则是基于 rsp，赋给 (%rsp) 和 8(%rsp) 这两个内存地址。而在被调用者中，访问这两个参数则是使用 16(%rbp) 和 24(%rbp)。

我也画了一张图，给你展示了如何在 Caller 和 Callee 的栈帧中访问参数 7 和参数 8：



你要注意，这里每个参数都占用了 8 个字节，而不是像前面在 Caller 的栈帧里保存参数和本地变量那样，占用 4 个字节。你能不能想一想这是为什么呢？

我估计你已经猜到了。这是因为在制定 ABI 的时候，要兼容尽量多的场景。这个规定，使得我们能够用同样的方式传递整型和长整型的参数，而长整型就需要占据 8 个字节。

你可以看看 [param_long.c](#) 及其对应的 [param_long.s](#) 汇编代码文件。你会发现，在这个示例文件中，我们把整型的参数改成了长整型，但是传递第 7 个和第 8 个参数的方法方式没有任何改变。在代码里访问这两个参数值的内存地址的表达式，也是一样的。

好了，我们已经了解了函数调用中传参的过程了，具体实现你可以参见代码库中的 [lowerFunctionCall](#) 方法。那么现在我们是不是就可以调用 `callq` 指令来做函数调用了呢？还不行。为什么呢？因为我们还有一项重要的工作要做，就是**保护好某些寄存器的值**。

还记得吗？在函数调用的过程中，有些寄存器的值是由 Caller 负责保护的，而另一些寄存器的值是由 Callee 负责保护的，这是为了避免可能由于多方使用同一个寄存器，导致寄存器的值被破坏，从而导致计算错误的问题。

我们还是通过示例代码来分析一下寄存器的使用可能产生的冲突。你会看到，在 [param.c](#) 中有下面的一行代码：

```
1 int c = a*b + foo(a,b,1,2,3,4,5,6) + foo(b,a,7,8,9,10,11,12);
```

[复制代码](#)

这行代码是我故意设计的，目的就是制造出寄存器使用上的冲突。整个计算过程大致可以分成下面这几步，其中涉及到 t1、t2 和 t3 三个临时变量。

```
1 t1 = a*b
2 t2 = foo(a,b,1,2,3,4,5,6)
3 t1 = t1 + t2
4 t3 = foo(b,a,7,8,9,10,11,12)
5 t1 = t1 + t3
6 c = t1
```

[复制代码](#)

首先，我们需要计算表达式 $a*b$ ，这个结果我们用 $t1$ 表示，并把它映射到寄存器。在 [param.s](#) 里，这个寄存器是 `eax`。

接着，我们需要调用 `foo()` 函数，假设返回值保存在 $t2$ 中。这里要注意，根据 ABI，`foo()` 函数的返回是要保存到 `eax` 中的，所以 $t2$ 和 $t1$ 都要使用 `eax` 寄存器。为了防止 $t1$ 的值被破坏，我们就必须先把它写到栈里去。

[复制代码](#)

```
1 movl    %eax, -20(%rbp)    ## 4-byte Spill
2 callq   _foo
3 movl    -20(%rbp), %ecx    ## 4-byte Reload
4 addl    %eax, %ecx        ## t1 = t1 + t2, 把t2的值加到t1上
```

接下来，在调用 `foo` 函数以后，我们要把 $t1$ 和 $t2$ 相加。在此之前，我们又需要把 $t1$ 从内存中恢复到寄存器里来。但这个寄存器不再是 `eax`，因为现在 $t2$ 已经占用了 `eax`。所以， $t1$ 被装载到了 `ecx` 寄存器中。

再接着，我们还要再一次调用 `foo` 函数。在这次调用中，寄存器的使用再次产生了冲突，因为这次我们要用 `ecx` 来传递 `foo` 函数的第四个参数。那这又要怎么办呢？我们还是要把 $t1$ 的值从 `ecx` 寄存器里保存到内存中，在调用完 `foo` 之后，再从内存恢复到寄存器。

[复制代码](#)

```
1 movl    %ecx, -24(%rbp)    ## 4-byte Spill
2 ...
3 callq   _foo
4 movl    -24(%rbp), %ecx    ## 4-byte Reload
5 addl    %eax, %ecx        ## t1 = t1 + t3, 把t3的值再添加到t1上。
6 movl    %ecx, -16(%rbp)    ## c = t1
```

好了，通过上面这个例子的分析，我相信你已经对寄存器的冲突有了比较直观的了解。这里，我再带你总结一下可能发生冲突的场景：

场景 1，某个临时变量使用了 `eax`，但接下来的函数返回值也要使用 `eax`，也就是这个 `eax` 在 Callee 中去做了设置；

场景 2，某个临时变量使用了 `edi`、`esi`、`edx`、`ecx`、`r8d` 和 `r9d` 这 6 个寄存器其中的一个，接下来的函数调用还要这个寄存器来传递参数；

场景 3，在 Caller 中使用了不由 Callee 保护的寄存器，比如 r10d、r11d 等等。在 ABI 中，规定只有 ebx、r12d~r15d 这几个寄存器是由 Callee 保护的，所以其他的寄存器，都需要由 Caller 保护，避免被 Callee 破坏。

那我们要如何保护这些寄存器呢？我们需要把它们值写到栈帧里，之后再从栈帧里恢复。

注意，把一个临时变量 t 保存到内存的过程叫做 Spill，也叫做溢出，我们在后面专门学习寄存器分配算法的时候会再次见到这个词汇。把 t 从内存再重新装载到寄存器的过程，叫做 Reload。不过这里你也要注意，虽然新的寄存器不一定是原来那个寄存器，但编译器会知道，这还是原来那个临时变量 t。

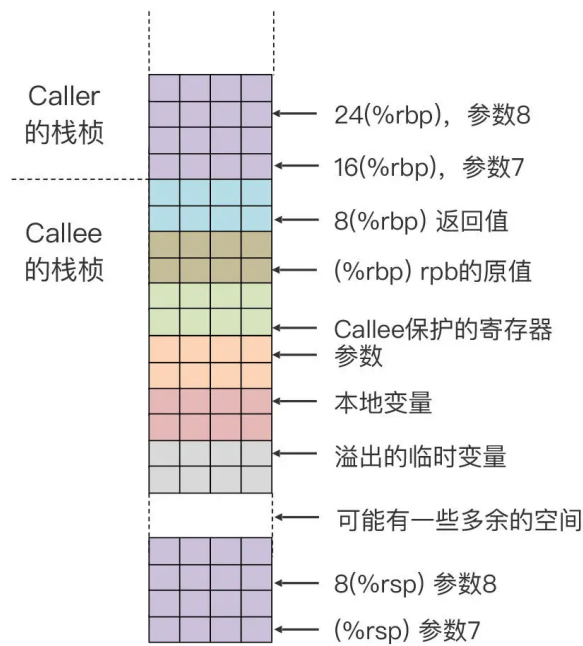
Spill 和 Reload 的实现，你仍然可以参见 [lowerFunctionCall](#) 方法。

那么，到目前为止，我们已经往栈帧里放了很多数据，包括返回值地址、rbp 的原值、参数，还有本地变量。而且，把临时变量溢出（Spill）到内存里也需要预留一些空间，调用函数时给函数传参也需要使用栈的空间。

那我们现在就再进一步，梳理一下栈帧里的内容，也来分析一下我们要怎么做好栈帧的维护。

栈帧的维护

你可以先看一下，采用我们目前的算法形成的栈帧结构是这样的：



一个栈帧从上到下，依次是返回值、rbp 原值、Callee 保护的寄存器（目前我们的示例程序中还没有用到这些寄存器）、参数、本地变量、溢出到内存的临时变量。

然后你再从栈顶，也就是栈帧的最底下往上看，这里是我们在调用函数时，为超过 6 个的参数所保存的空间。

然后你还会注意到，在栈帧中间可能还有一些多余的空间。为什么会这样呢？

第一个原因，是这个函数可能会调用多个函数，而被调用的多个函数所需的参数数量是不同的，因此我们需要为参数最多的那个函数预留出足够的用于保存参数的空间。

第二个原因，是内存对齐。原来，在 ABI 中规定，如果这个函数要调用另一个函数，那么参数区的尾部应该是 16 字节对齐的。所以说，参数区的尾部也是两个栈帧之间的分界线。换句话说，在控制转移到函数入口的那一刻， $(\%rsp+8)$ 的值是能够被 16 整除的。这里的 8，是因为 `callq` 指令把 8 位的一个返回地址压到了栈里。

我建议你看看 [ABI 文档](#) 中对栈帧的描述，我截取了一小段，你可以看看：

3.2.2 The Stack Frame

In addition to registers, each function has a frame on the run-time stack. This stack grows downwards from high addresses. Figure 3.3 shows the stack organization.

The end of the input argument area shall be aligned on a 16 (32, if `__m256` is passed on stack) byte boundary. In other words, the value $(\text{\%rsp} + 8)$ is always a multiple of 16 (32) when control is transferred to the function entry point. The stack pointer, `\%rsp`, always points to the end of the latest allocated stack frame.⁷

来源：System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99.6

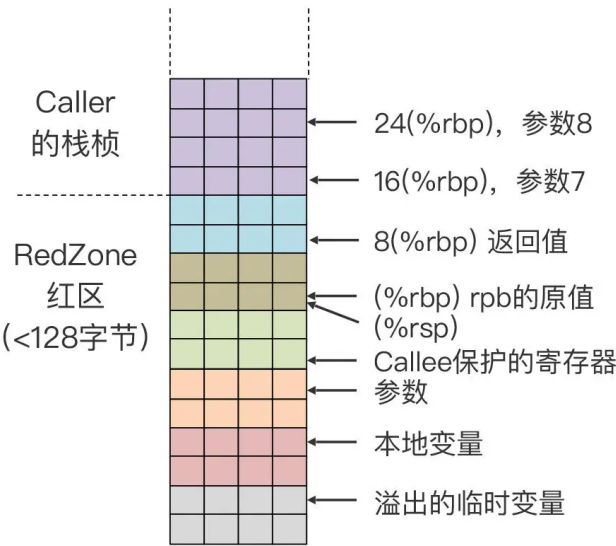
基于上面的原则，我们在为一个函数建立栈帧的时候，需要计算出 `rsp` 移动的量，以便确定正确的栈帧尺寸。在 [@param.s](#) 的 `_main` 函数中，序曲和尾声部分各有一行指令，就是用来移动 `rsp` 指针的。

 复制代码

```
1  ##序曲
2  pushq  %rbp
3  movq   %rsp, %rbp
4  subq   $48, %rsp    ##把栈帧扩展48个字节。整个栈帧的大小是48+16字节
5
6  ##函数体
7  ...
8
9  ##尾声
10 addq   $48, %rsp    ##缩回栈帧
11 popq   %rbp
12 retq
```

不过，刚才我们看的是 `main` 函数的栈帧的情况。但对于 `foo` 函数来说，它的栈帧会有所不同。

有什么区别呢？区别就在于，`foo` 函数是叶子函数，也就是说它没有调用其他函数。在这个情况下，我们根本没有必要移动 `rsp` 指针来为栈帧申请内存。只要 `foo` 函数占用的内存不超过 128 个字节，我们直接使用红区（RedZone）就行了。它的栈帧结构如下图所示，这时 `rsp` 的值和 `rbp` 的值是相同的。



关于红区的知识点，我仍然建议你阅读 [ABI 文档](#)，我在这里也贴了一小段，供你参考。

The 128-byte area beyond the location pointed to by `%rsp` is considered to be reserved and shall not be modified by signal or interrupt handlers.⁸ Therefore, functions may use this area for temporary data that is not needed across function calls. In particular, leaf functions may use this area for their entire stack frame, rather than adjusting the stack pointer in the prologue and epilogue. This area is known as the red zone.

来源：System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99.6

现在，栈帧维护的原理，我们已经讲清楚了，至于具体的计算栈帧大小、维护栈帧的代码，你可以在代码库中的 [addPrologue](#)和 [addEpilogue](#)两个函数中看到。在这里，你要特别关注一下其中计算栈帧大小的相关代码，以此为线索你就能搞清楚整个栈帧的内存布局了。我相信，在明白原理之后，你再阅读这些代码，可以保持一条比较清晰的思路。

课程小结

好了，今天的内容就是这些，到今天这节课为止，我们的语言已经能够为表达式计算和函数调用等功能生成汇编代码和可执行文件了。那么今天这节课，我希望你记住下面几个关键的知识点：

首先，我们在采用简单的寄存器分配算法时，通常是把参数、本地变量都保存到栈帧里，但临时变量则要用使用寄存器，在这个过程中，我们加深了对 x86 指令的理解。像加减乘除等运算，目标操作数应该是寄存器，而对源操作数就没有这个要求了。另外，mov 指令虽然在源操作数和目标操作数中都可以使用内存地址，但不能两个操作数都是内存地址。

第二，我们再次熟悉了函数调用的约定，也就是前 6 个参数用寄存器，超过 6 个的参数使用 Caller 的栈帧。不过这次，我们更细致地了解了从 Caller 和 Callee 中应该如何访问这些额外的参数。具体来说，在 Caller 中我们要基于 rsp 寄存器来寻址，但在 Callee 中则要基于 rbp 寄存器来寻址。

第三，在调用函数时，会发生寄存器使用冲突的情况，这个时候我们需要把保存在寄存器中的临时变量溢出到内存中，需要时再装载回来，而这两个操作使用的寄存器可能并不是同一个。

最后，我们总结了栈帧的内存布局设计，并指出了如何正确地移动栈顶指针 rsp 来申请内存。在这个过程中，栈帧要保证 16 字节内存对齐。而对于叶子函数，我们还可以直接使用 RedZone，不用移动栈顶指针 rsp。不过，这里要指出的是，对于这个内存布局，只有超过 6 个的额外参数的位置、返回地址的位置，还有内存对齐等少量内容是由 ABI 规定的，其他大部分栈帧空间都是语言的设计者自己决定如何使用的。

最后再补充一句，到目前为止，我在这节课提到的所有寄存器，都是与整数计算有关的寄存器。如果你要进行浮点数计算，使用的寄存器会是另外一组，我们在后面的课程里再详细介绍。

思考题

在这节课的示例代码中，我们看到临时变量首先使用的是 Caller 保护的寄存变量，比如 eax，而不是 Callee 保护的。这又是什么道理呢？请你分析一下。

感谢你和我一起学习，也欢迎你把这节课分享给更多对生成本地代码感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

🔗 [这节课的示例代码在这里！](#)

分享给需要的人，Ta订阅后你可得 **20** 元现金奖励

👍 赞 0

💡 提建议

© 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。

上一篇 16 | 生成本地代码第1关：先把基础搭好

下一篇 18 | 生成本地代码第3关：实现完整的功能

精选留言 (4)

💬 写留言



leaf

2021-09-15

使用callee保护的寄存器则方法在prolog之后需要对它做保留, epilog中做恢复; 使用caller-save寄存器, 则由当前函数自己决定是否需要在调用前后对其做保留恢复, 如果寄存器的值不跨调用活跃, 就不需要保留恢复了, 这样性能更好

展开 ▾



qinsi

2021-09-15

个人理解：Caller保护的寄存器在函数返回之后会由Caller恢复，所以在函数执行时可以随便改；Callee保护的寄存器需要确保进入函数时和函数返回时寄存器中的值是不变的，所以要么函数执行时不要去动它们，要动的话Callee就要负责保存和恢复。相比之下用Caller保护的寄存器成本更低，优先使用，不够用的时候再用Callee保护的寄存器。

展开 ▾



奋斗的蜗牛

2021-09-15

优先使用caller保护的寄存器，可能可以在函数内联优化时，去掉callee的寄存器保护代码



罗乾林

2021-09-15

使用Callee保护的寄存器作为临时变量必定涉及寄存器的保存和恢复，存在内存访问开销
优先使用Caller保护的寄存器，如果Caller中未使用该寄存器将不用保存和恢复

