知乎 首发于 linux内核技术

x86栈帧原理


小乐叔叔
资深打工人
关注他

前言

上一篇《x86通用寄存器》我们了解了IA32平台和x86_64平台的通用寄存器，以及在GCC中的过程调用惯例。本文继续探讨这两个平台的栈帧结构，深入了解栈、寄存器在函数调用中的作用，为后面《linux栈回溯》系列文章奠定基础。

什么是栈帧

C语言中，每个栈帧（stack frame）对应着一个未运行完的函数，栈帧中保存了该函数的返回地址和局部变量。比如：入口函数是main，然后调用各个子函数。在对应机器语言中，GCC把**过程转化成栈帧**（frame），简单的说，每个栈帧对应一个过程（函数）。

i386典型栈帧结构中，由%ebp指向栈帧开始，%esp指向栈顶。

栈帧结构（32位）：

图一

过程调用分析

c函数的进出主要靠两个汇编指令：call和ret指令。前者跳转到函数入口执行，后者处理函数返回，可以配合栈实现多层函数调用。下面列出一个简单得例子：

```
#include <stdio.h>
int main()
{
    int a = 2, b = 3;
    int func(int a, int b);
    int c = func(a, b);
    printf("%d\n%d\n%d\n",a, b, c);
}
int func(int a, int b)
{
    int c = 20;
    return a + b + c;
}
```

```
_main:
LFB13:
 .cfi_startproc
 pushl %ebp
 movl %esp, %ebp
 andl $-16, %esp
 subl $32, %esp
... ...
 call _func
 movl %eax, 20(%esp)
 movl 20(%esp), %eax
 movl %eax, 12(%esp)
 movl 24(%esp), %eax
... ...
_func:
LFB14:
 .cfi_startproc
 pushl %ebp
 movl %esp, %ebp
 subl $16, %esp
 movl $20, -4(%ebp)
 movl 8(%ebp), %edx
 movl 12(%ebp), %eax
 addl %eax, %edx
 movl -4(%ebp), %eax
 addl %edx, %eax
 leave
 .cfi_restore 5
 .cfi_def_cfa 4, 4
 ret
```

（1）指令“**call _func**”其实干了两件事情：

```
[1] Pushl %eip //保存下一条指令（ movl %eax, 20(%esp) ）的地址，用于函数返回继续执行；
[2] Jmp _func //跳转到函数_func
```

（2）_func函数结束

_func函数中的指令“ret ”相当于：

栈帧中，最重要的是**帧指针**%ebp和**栈指针**%esp，有了这两个指针，我们就可以刻画一个完整的栈帧。

当从main函数，进入_func函数时：

```
//保存上一个栈帧的帧指针，并设置当前的指针
pushl %ebp
 movl %esp, %ebp
```

上面的汇编表示，将上一个栈帧指针（存在ebp中）压栈保存，并将此时栈顶地址作为下一个栈帧指针（实际esp指向的就是保存上一个栈帧指针的内存地址），如图一所示，就是一次完整的栈帧建立过程。

当从_func函数返回main时：

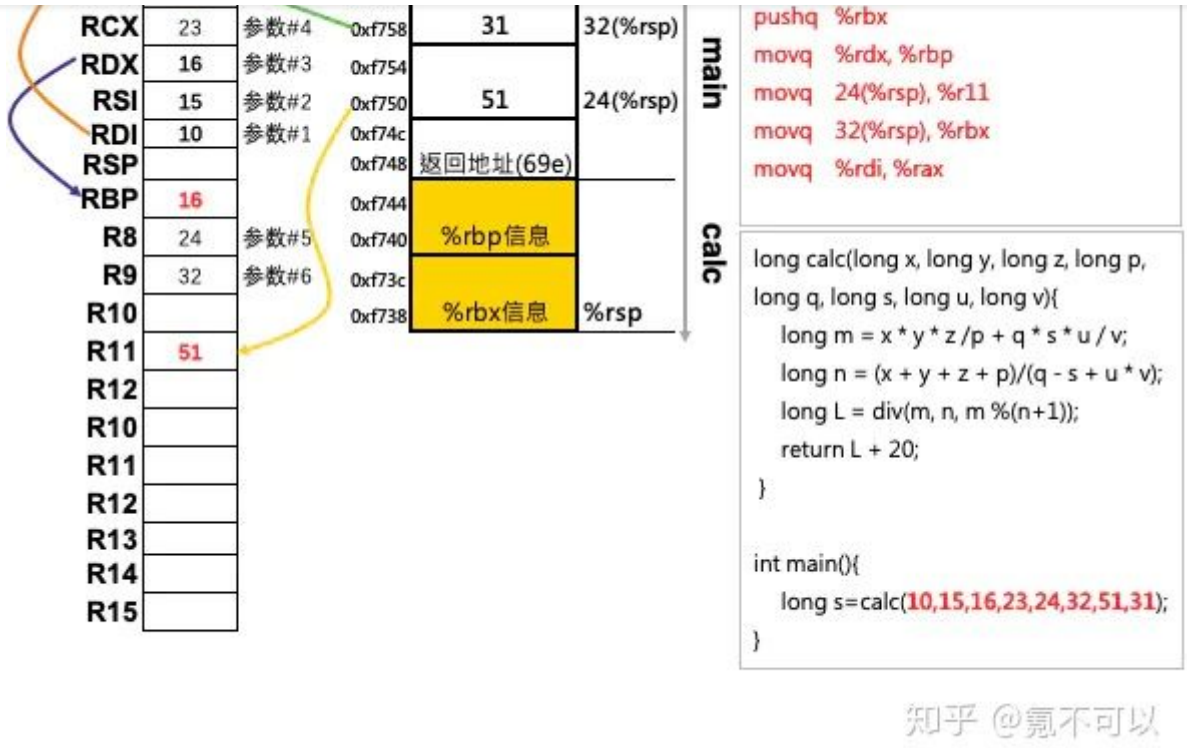
```
//撤销当前栈帧，恢复上一个栈帧的%rbp
leave
```

leave指令相当于下面两条：

```
Movq %rbp %rsp //撤销栈空间，回滚%rsp。
Popq %rbp      //恢复上一个栈帧的%rbp。
```

x86_64栈帧

> 通常情况下（简单函数，参数和局部变量很少），x86-64函数不在需要栈帧，唯一写入栈的操作就是执行到callq指令时候的“返回地址”8个字节
> 不过以下情况需要使用栈帧：
> [1] 局部变量太多,64位寄存器处理不过来。
> [2] 局部变量中存在数组或者结构体(或者叫做类类型)的变量
> [3] 使用取址操作符&就计算局部变量的的内存地址。
> [4] 调用另外一个超过6个参数的函数
> [5] 需要在修改它们之前保存被调用者保存寄存器的状态
> 复杂情况下的x86_64栈帧：
> 引自博客：第6篇 栈帧程序栈 x86-64过程调用



图二

上图右边是C源码和部分汇编。因为calc有8个参数，栈里面分别压入了参数7、参数8、函数返回地址、上一个栈帧rbp地址、存放main函数局部变量的rbx，因此在读取参数7和8的时候，栈指针rsp需要位移8*3 和8*4：

```
movq 24(%rsp), %r11
movq 32(%rsp), %rbx
```

1）x86_64过程调用,大量地使用来寄存器来传递参数，甚至保；存函数的局部变量（使用空闲的寄存器）,因为寄存器的存取速度远远高于内存访问和内存写入；

2）x86_64过程调用的每个函数的栈帧尺寸非常少，甚至不需要。实际上gcc 在x86_64平台默认不使用rbp保存栈帧指针，使用%rsp索引栈帧。gcc坚持推荐这一技术并将它设置为默认选项，同时提供-**fno-omit-frame-pointer** 选项，只要设置了这个标记，编译器就不会再省略栈帧了。32位的x86也可以通过-fomit-frame-pointer优化选项使ebp不再用来保存栈帧地址。

访问栈顶之外（red zone）

x86-64遵循ABI规则，它定义了一些规范，遵循ABI的具体实现应该满足这些规范，其中，他就规定了程序可以使用栈顶之外128字节的地址。下面在x86-64平台上，加上编译优化选项-O2后的源

```
int foo(int x)
{
        int array[] = {1,3,5};
        return array[x];
}
int main(int argc, char *argv[])
{
        int i = 1;
        int j = foo(i);
        printf("i=%d,j=%d\n", i, j);
        return 0;
}
```

gcc -O1 -S 编译后（节选）：

```
foo:
.LFB13:
        movl    $1, -24(%rsp)
        movl    $3, -20(%rsp)
        movl    $5, -16(%rsp)
        movslq  %edi,%rdi
        movl    -24(%rsp,%rdi,4), %eax
        ret
... ...
main:
.LFB14:
        subq    $8, %rsp
.LCFI0:
        movl    $1, %edi
        call    foo
        movl    %eax, %edx
        movl    $1, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        addq    $8, %rsp
        ret
```

sub就是减法指令，栈向下生长，减8个字节，就是栈生长了8个字节预留空间。这部分空间是用来保存“call foo”指令的返回地址，上面32位栈帧里介绍过，call指令包含了“Pushl %eip”。

2）main函数返回前，回收分配的空间

```
addq    $8, %rsp
```

add就是加法指令，即栈顶恢复原来位置。

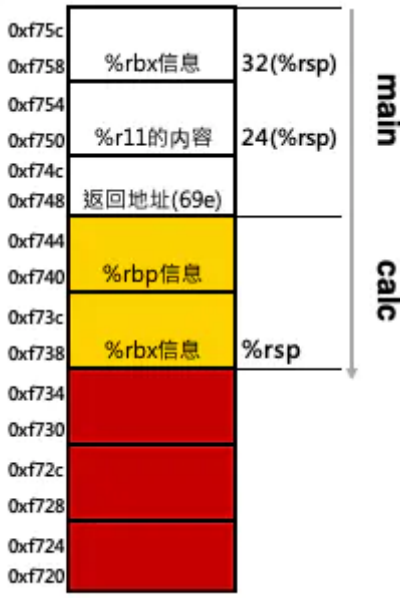
3）访问预先分配的栈帧空间

在foo函数里有对栈顶rsp的引用访问，比如：

```
movl    -24(%rsp,%rdi,4), %eax
```

为何foo函数没有分配栈帧呢？而是直接访问main的栈帧且越界了，这个就是ABI规则规定的，栈外的128字节“red zone”区域。

%rsp所指向的位置之外的**128字节区域**被视为已保留，并且不得由信号或中断处理程序修改。 因此，**函数可以使用此区域存储跨函数调用不需要的临时数据**。也就是说当使用“红灯区”时，编译器优化掉了%rsp指针的递减和重新填充的两条指令。 尤其是，叶子函数(就是位于整个函数调用链的末端的函数)可以在整个堆栈框架中使用此区域。



第6篇-戏说程序栈 x86_64过程调用

《深入理解计算机体系结构》

《x86系列汇编语言程序设计》

blog.csdn.net/zhbt1234/...

编辑于 2020-11-17

Linux 内核 x86-64

文章被以下专栏收录

linux内核技术
分享、探讨linux kernel技术

推荐阅读

x86_64架构下的函数调用及栈帧原理

本文为看雪论坛优秀文章 看雪论坛 作者ID：有毒 x86_64架构下的函数调用及栈帧原理1. x86_64寄存器在分析函数调用时，必须要对CPU的寄存器熟悉。在所有的体系架构中，每个寄存器都有建议的使...

看雪学院

linux 栈回溯(x86_64)

前序前面几个章节我们了解了ELF文件格式》、《ELF文件加载程》、《x86通用寄存器》、《栈帧原理》和《linux 进程内核栈》，对x86平台上程序运行和机制有了一定认识。接下来我们

小乐叔叔 发表于li