

指令			状态值 (指令执行前)				描述
标号	PC	指令	%rdi	%rax	%rsp	*%rsp	
M1	0x40055b	callq	100	—	0x7fffffff820	—	调用top(100)
T1	0x400555	sub	100	—	0x7fffffff818	0x400560	进入top
T2	0x400559	callq	95	—	0x7fffffff818	0x400560	调用leaf(95)
L1	0x400540	lea	95	—	0x7fffffff810	0x40054e	进入leaf
L2	0x400544	retq	—	97	0x7fffffff810	0x40054e	从leaf返回97
T3	0x40054e	add	—	97	0x7fffffff818	0x400560	继续top
T4	0x400551	retq	—	194	0x7fffffff818	0x400560	从top返回194
M2	0x400560	mov	—	194	0x7fffffff820	—	继续main

b) 示例代码的执行过程


图 3-27 (续)

行的详细过程, main 调用 top(100), 然后 top 调用 leaf(95)。函数 leaf 向 top 返回 97, 然后 top 向 main 返回 194。前面三列描述了被执行的指令, 包括指令标号、地址和指令类型。后面四列给出了在该指令执行前程序的状态, 包括寄存器 %rdi、%rax 和 %rsp 的内容, 以及位于栈顶的值。仔细研究这张表的内容, 它们说明了运行时栈在管理支持过程调用和返回所需的存储空间中的重要作用。

leaf 的指令 L1 将 %rax 设置为 97, 也就是要返回的值。然后指令 L2 返回, 它从栈中弹出 0x400054e。通过将 PC 设置为这个弹出的值, 控制转移回 top 的 T3 指令。程序成功完成对 leaf 的调用, 返回到 top。

指令 T3 将 %rax 设置为 194, 也就是要从 top 返回的值。然后指令 T4 返回, 它从栈中弹出 0x4000560, 因此将 PC 设置为 main 的 M2 指令。程序成功完成对 top 的调用, 返回到 main。可以看到, 此时栈指针也恢复成了 0x7fffffff820, 即调用 top 之前的值。

可以看到, 这种把返回地址压入栈的简单的机制能够让函数在稍后返回到程序中正确的点。C 语言(以及大多数程序语言)标准的调用/返回机制刚好与栈提供的后进先出的内存管理方法吻合。

 **练习题 3.32** 下面列出的是两个函数 first 和 last 的反汇编代码, 以及 main 函数调用 first 的代码:

Disassembly of last(long u, long v)

u in %rdi, v in %rsi

1 000000000400540 <last>:

2	400540: 48 89 f8	mov	%rdi,%rax	L1: u
3	400543: 48 0f af c6	imul	%rsi,%rax	L2: u*v
4	400547: c3	retq		L3: Return

Disassembly of first(long x)

x in %rdi

5 000000000400548 <first>:

6	400548: 48 8d 77 01	lea	0x1(%rdi),%rsi	F1: x+1
7	40054c: 48 83 ef 01	sub	\$0x1,%rdi	F2: x-1
8	400550: e8 eb ff ff ff	callq	400540 <last>	F3: Call last(x-1,x+1)