Instruction			State values (at beginning)					
Label	PC	Instruction	%rdi	%rsi	%rax	%rsp	*%rsp	Description
M1	0x400560	callq	10	—	—	0x7fffffffe820	—	Call first(10)
F1								
F2								
F3								
L1								
L2								
L3								
F4								
M2								

3.7.3 Data Transfer

In addition to passing control to a procedure when called, and then back again when the procedure returns, procedure calls may involve passing data as arguments, and returning from a procedure may also involve returning a value. With x86-64, most of these data passing to and from procedures take place via registers. For example, we have already seen numerous examples of functions where arguments are passed in registers %rdi, %rsi, and others, and where values are returned in register %rax. When procedure P calls procedure Q, the code for P must first copy the arguments into the proper registers. Similarly, when Q returns back to P, the code for P can access the returned value in register %rax. In this section, we explore these conventions in greater detail.

With x86-64, up to six integral (i.e., integer and pointer) arguments can be passed via registers. The registers are used in a specified order, with the name used for a register depending on the size of the data type being passed. These are shown in Figure 3.28. Arguments are allocated to these registers according to their

Operand size (bits)	Argument number					
	1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

Figure 3.28 Registers for passing function arguments. The registers are used in a specified order and named according to the argument sizes.

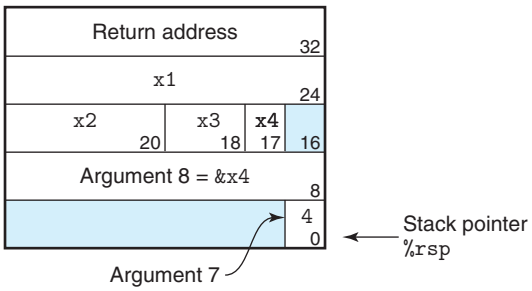

Figure 3.33
Stack frame for function `call_proc`. The stack frame contains local variables, as well as two of the arguments to pass to function `proc`.

When procedure `proc` is called, the program will begin executing the code shown in Figure 3.29(b). As shown in Figure 3.30, arguments 7 and 8 are now at offsets 8 and 16 relative to the stack pointer, because the return address was pushed onto the stack.

When the program returns to `call_proc`, the code retrieves the values of the four local variables (lines 17–20) and performs the final computations. It finishes by incrementing the stack pointer by 32 to deallocate the stack frame.

3.7.5 Local Storage in Registers

The set of program registers acts as a single resource shared by all of the procedures. Although only one procedure can be active at a given time, we must make sure that when one procedure (the *caller*) calls another (the *callee*), the callee does not overwrite some register value that the caller planned to use later. For this reason, x86-64 adopts a uniform set of conventions for register usage that must be respected by all procedures, including those in program libraries.

By convention, registers `%rbx`, `%rbp`, and `%r12`–`%r15` are classified as *callee-saved* registers. When procedure `P` calls procedure `Q`, `Q` must *preserve* the values of these registers, ensuring that they have the same values when `Q` returns to `P` as they did when `Q` was called. Procedure `Q` can preserve a register value by either not changing it at all or by pushing the original value on the stack, altering it, and then popping the old value from the stack before returning. The pushing of register values has the effect of creating the portion of the stack frame labeled "Saved registers" in Figure 3.25. With this convention, the code for `P` can safely store a value in a callee-saved register (after saving the previous value on the stack, of course), call `Q`, and then use the value in the register without risk of it having been corrupted.

All other registers, except for the stack pointer `%rsp`, are classified as *caller-saved* registers. This means that they can be modified by any function. The name "caller saved" can be understood in the context of a procedure `P` having some local data in such a register and calling procedure `Q`. Since `Q` is free to alter this register, it is incumbent upon `P` (the caller) to first save the data before it makes the call.

As an example, consider the function `P` shown in Figure 3.34(a). It calls `Q` twice. During the first call, it must retain the value of `x` for use later. Similarly, during the second call, it must retain the value computed for `Q(y)`. In Figure 3.34(b),