in different programming languages—functions, methods, subroutines, handlers, and so on—but they all share a general set of features.

There are many different attributes that must be handled when providing machine-level support for procedures. For discussion purposes, suppose procedure P calls procedure Q, and Q then executes and returns back to P. These actions involve one or more of the following mechanisms:

Passing control. The program counter must be set to the starting address of the code for Q upon entry and then set to the instruction in P following the call to Q upon return.

Passing data. P must be able to provide one or more parameters to Q, and Q must be able to return a value back to P.

Allocating and deallocating memory. Q may need to allocate space for local variables when it begins and then free that storage before it returns.

The x86-64 implementation of procedures involves a combination of special instructions and a set of conventions on how to use the machine resources, such as the registers and the program memory. Great effort has been made to minimize the overhead involved in invoking a procedure. As a consequence, it follows what can be seen as a minimalist strategy, implementing only as much of the above set of mechanisms as is required for each particular procedure. In our presentation, we build up the different mechanisms step by step, first describing control, then data passing, and, finally, memory management.

3.7.1 The Run-Time Stack

A key feature of the procedure-calling mechanism of C, and of most other languages, is that it can make use of the last-in, first-out memory management discipline provided by a stack data structure. Using our example of procedure P calling procedure Q, we can see that while Q is executing, P, along with any of the procedures in the chain of calls up to P, is temporarily suspended. While Q is running, only it will need the ability to allocate new storage for its local variables or to set up a call to another procedure. On the other hand, when Q returns, any local storage it has allocated can be freed. Therefore, a program can manage the storage required by its procedures using a stack, where the stack and the program registers store the information required for passing control and data, and for allocating memory. As P calls Q, control and data information are added to the end of the stack. This information gets deallocated when P returns.

As described in Section 3.4.4, the x86-64 stack grows toward lower addresses and the stack pointer `%rsp` points to the top element of the stack. Data can be stored on and retrieved from the stack using the `pushq` and `popq` instructions. Space for data with no specified initial value can be allocated on the stack by simply decrementing the stack pointer by an appropriate amount. Similarly, space can be deallocated by incrementing the stack pointer.

When an x86-64 procedure requires storage beyond what it can hold in registers, it allocates space on the stack. This region is referred to as the procedure's

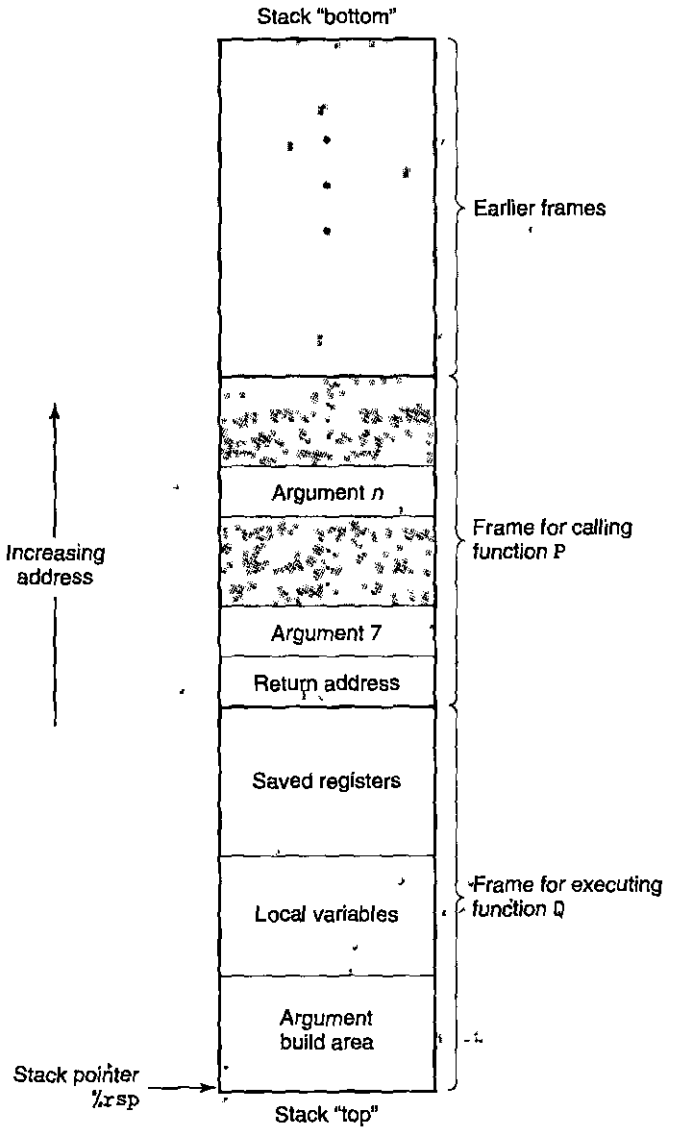

Figure 3.25 General stack frame structure. The stack can be used for passing arguments, for storing return information, for saving registers, and for local storage. Portions may be omitted when not needed.

stack frame. Figure 3.25 shows the overall structure of the run-time stack, including its partitioning into stack frames, in its most general form. The frame for the currently executing procedure is always at the top of the stack. When procedure P calls procedure Q, it will push the *return address* onto the stack, indicating where within P the program should resume execution once Q returns. We consider the return address to be part of P's stack frame, since it holds state relevant to P. The code for Q allocates the space required for its stack frame by extending the current stack boundary. Within that space, it can save the values of registers, allocate

space for local variables, and set up arguments for the procedures it calls. The stack frames for most procedures are of fixed size, allocated at the beginning of the procedure. Some procedures, however, require variable-size frames. This issue is discussed in Section 3.10.5. Procedure P can pass up to six integral values (i.e., pointers and integers) on the stack, but if Q requires more arguments, these can be stored by P within its stack frame prior to the call.

In the interest of space and time efficiency, x86-64 procedures allocate only the portions of stack frames they require. For example, many procedures have six or fewer arguments, and so all of their parameters can be passed in registers. Thus, parts of the stack frame diagrammed in Figure 3.25 may be omitted. Indeed, many functions do not even require a stack frame. This occurs when all of the local variables can be held in registers and the function does not call any other functions (sometimes referred to as a *leaf procedure*, in reference to the tree structure of procedure calls). For example, none of the functions we have examined thus far required stack frames.

3.7.2 Control Transfer

Passing control from function P to function Q involves simply setting the program counter (PC) to the starting address of the code for Q. However, when it later comes time for Q to return, the processor must have some record of the code location where it should resume the execution of P. This information is recorded in x86-64 machines by invoking procedure Q with the instruction `call Q`. This instruction pushes an address *A* onto the stack and sets the PC to the beginning of Q. The pushed address *A* is referred to as the *return address* and is computed as the address of the instruction immediately following the `call` instruction. The counterpart instruction `ret` pops an address *A* off the stack and sets the PC to *A*.

The general forms of the `call` and `ret` instructions are described as follows:

Instruction		Description
`call`	*Label*	Procedure call
`call`	**Operand*	Procedure call
`ret`		Return from call

(These instructions are referred to as `callq` and `retq` in the disassembly outputs generated by the program OBJDUMP. The added suffix 'q' simply emphasizes that these are x86-64 versions of call and return instructions, not IA32. In x86-64 assembly code, both versions can be used interchangeably.)

The `call` instruction has a target indicating the address of the instruction where the called procedure starts. Like jumps, a call can be either direct or indirect. In assembly code, the target of a direct call is given as a label, while the target of an indirect call is given by '*' followed by an operand specifier using one of the formats described in Figure 3.3.