This brings us to the last question listed earlier: Does the relative size of the two memories satisfy the cost requirement? The answer is clearly yes. If we need only a relatively small upper-level memory to achieve good performance, then the average cost per bit of the two levels of memory will approach that of the cheaper lower-level memory.

APPENDIX 1B PROCEDURE CONTROL

A common technique for controlling the execution of procedure calls and returns makes use of a stack. This appendix summarizes the basic properties of stacks and looks at their use in procedure control.

Stack Implementation

A stack is an ordered set of elements, only one of which (the most recently added) can be accessed at a time. The point of access is called the *top* of the stack. The number of elements in the stack, or length of the stack, is variable. Items may only be added to or deleted from the top of the stack. For this reason, a stack is also known as a *pushdown list* or a *last-in-first-out (LIFO) list*.

The implementation of a stack requires that there be some set of locations used to store the stack elements. A typical approach is illustrated in Figure 1.25. A contiguous block of locations is reserved in main memory (or virtual memory) for the stack. Most of the time, the block is partially filled with stack elements and the

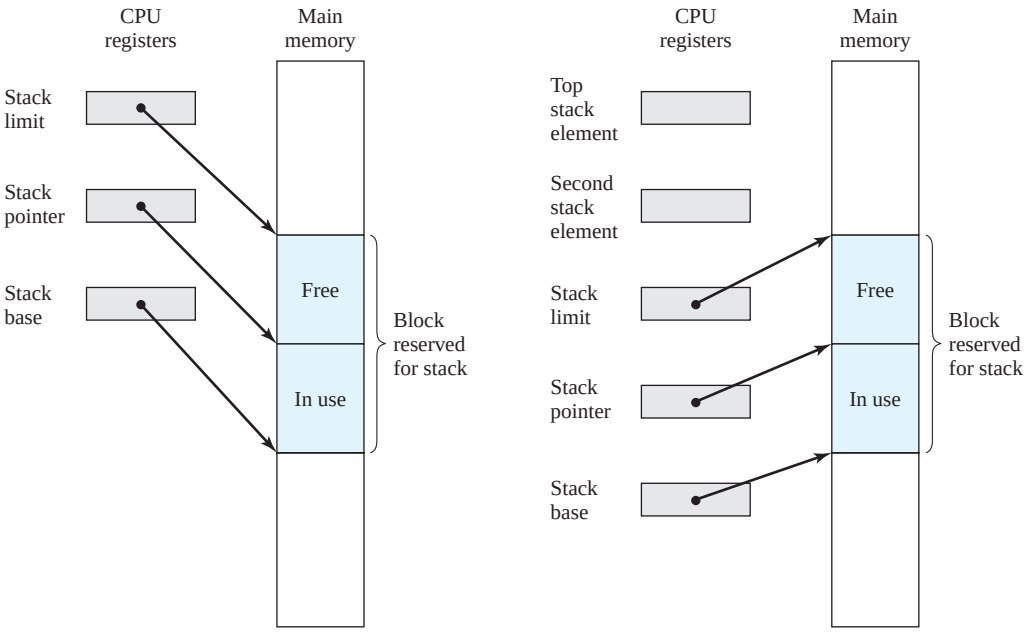

Figure 1.25 **Typical Stack Organization**

remainder is available for stack growth. Three addresses are needed for proper operation, and these are often stored in processor registers:

- **Stack pointer:** Contains the address of the current top of the stack. If an item is appended to (PUSH) or deleted from (POP) the stack, the pointer is decremented or incremented to contain the address of the new top of the stack.
- **Stack base:** Contains the address of the bottom location in the reserved block. This is the first location to be used when an item is added to an empty stack. If an attempt is made to POP an element when the stack is empty, an error is reported.
- **Stack limit:** Contains the address of the other end, or top, of the reserved block. If an attempt is made to PUSH an element when the stack is full, an error is reported.

Traditionally, and on most processors today, the base of the stack is at the high-address end of the reserved stack block, and the limit is at the low-address end. Thus, the stack grows from higher addresses to lower addresses.

Procedure Calls and Returns

A common technique for managing procedure calls and returns makes use of a stack. When the processor executes a call, it places (pushes) the return address on the stack. When it executes a return, it uses the address on top of the stack and removes (pops) that address from the stack. For the nested procedures of Figure 1.26, Figure 1.27 illustrates the use of a stack.

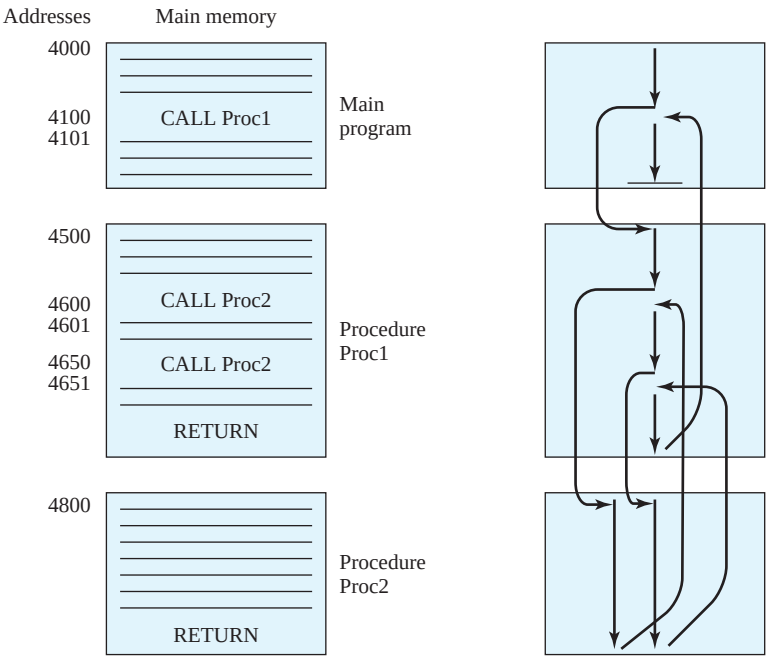

Figure 1.26 **Nested Procedures**

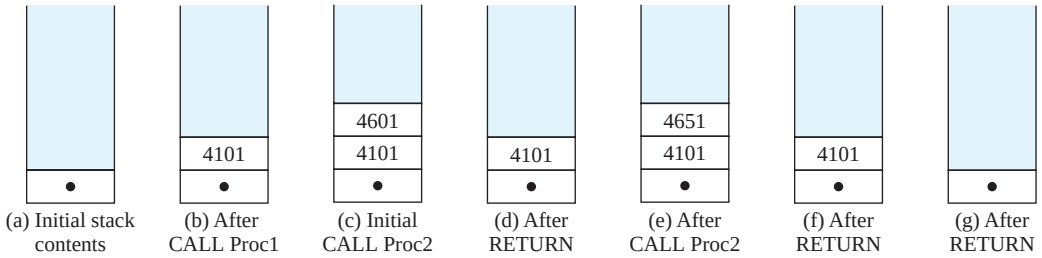

Figure 1.27 **Use of Stack to Implement Nested Procedures of figure 1.26**

It is also often necessary to pass parameters with a procedure call. These could be passed in registers. Another possibility is to store the parameters in memory just after the Call instruction. In this case, the return must be to the location following the parameters. Both of these approaches have drawbacks. If registers are used, the called program and the calling program must be written to assure that the registers are used properly. The storing of parameters in memory makes it difficult to exchange a variable number of parameters.

A more flexible approach to parameter passing is the stack. When the processor executes a call, it not only stacks the return address, it stacks parameters to be passed to the called procedure. The called procedure can access the parameters from the stack. Upon return, return parameters can also be placed on the stack, *under* the return address. The entire set of parameters, including return address, that is stored for a procedure invocation is referred to as a **stack frame**.

An example is provided in Figure 1.28. The example refers to procedure P in which the local variables $x1$ and $x2$ are declared, and procedure Q, which can be

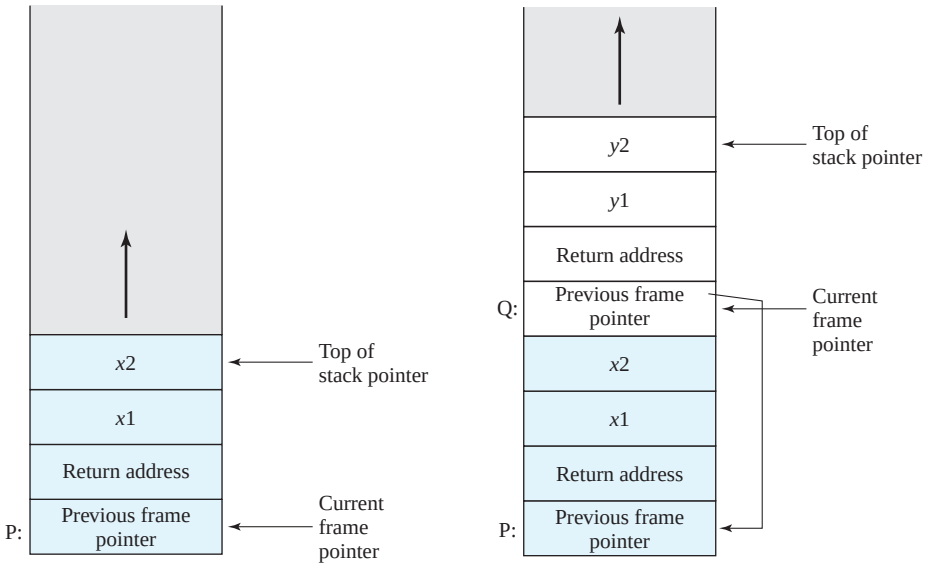

Figure 1.28 **Stack Frame Growth Using Sample Procedures P and Q**

called by P and in which the local variables $y1$ and $y2$ are declared. The first item stored in each stack frame is a pointer to the beginning of the previous frame. This is needed if the number or length of parameters to be stacked is variable. Next is stored the return point for the procedure that corresponds to this stack frame. Finally, space is allocated at the top of the stack frame for local variables. These local variables can be used for parameter passing. For example, suppose that when P calls Q, it passes one parameter value. This value could be stored in variable $y1$. Thus, in a high-level language, there would be an instruction in the P routine that looks like this:

CALL Q($y1$)

When this call is executed, a new stack frame is created for Q (Figure 1.28b), which includes a pointer to the stack frame for P, the return address to P, and two local variables for Q, one of which is initialized to the passed parameter value from P. The other local variable, $y2$, is simply a local variable used by Q in its calculations. The need to include such local variables in the stack frame is discussed in the next subsection.

Reentrant Procedures

A useful concept, particularly in a system that supports multiple users at the same time, is that of the reentrant procedure. A reentrant procedure is one in which a single copy of the program code can be shared by multiple users during the same period of time. Reentrancy has two key aspects: The program code cannot modify itself and the local data for each user must be stored separately. A reentrant procedure can be interrupted and called by an interrupting program and still execute correctly upon return to the procedure. In a shared system, reentrancy allows more efficient use of main memory: One copy of the program code is kept in main memory, but more than one application can call the procedure.

Thus, a reentrant procedure must have a permanent part (the instructions that make up the procedure) and a temporary part (a pointer back to the calling program as well as memory for local variables used by the program). Each execution instance, called activation, of a procedure will execute the code in the permanent part but must have its own copy of local variables and parameters. The temporary part associated with a particular activation is referred to as an *activation record*.

The most convenient way to support reentrant procedures is by means of a stack. When a reentrant procedure is called, the activation record of the procedure can be stored on the stack. Thus, the activation record becomes part of the stack frame that is created on procedure call.