(a) Code

```
void switcher(long a, long b, long c, long *dest)
a in %rsi, b in %rdi, c in %rdx, d in %rcx
switcher:
  cmpq    $7, %rdi
  ja      .L2
  jmp     *.L4(,%rdi,8)
  .section        .rodata
.L7:
  xorq    $15, %rsi
  movq    %rsi, %rdx
.L3:
  leaq    112(%rdx), %rdi
  jmp     .L6
.L5:
  leaq    (%rdx,%rsi), %rdi
  salq    $2, %rdi
  jmp     .L6
.L2:
  movq    %rsi, %rdi
.L6:
  movq    %rdi, (%rcx)
  ret
```

(b) Jump table

```
.L4:
  .quad   .L3
  .quad   .L2
  .quad   .L5
  .quad   .L2
  .quad   .L6
  .quad   .L7
  .quad   .L2
  .quad   .L5
```

Figure 3.24 Assembly code and jump table for Problem 3.31.

3.7 Procedures

Procedures are a key abstraction in software. They provide a way to package code that implements some functionality with a designated set of arguments and an optional return value. This function can then be invoked from different points in a program. Well-designed software uses procedures as an abstraction mechanism, hiding the detailed implementation of some action while providing a clear and concise interface definition of what values will be computed and what effects the procedure will have on the program state. Procedures come in many guises 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

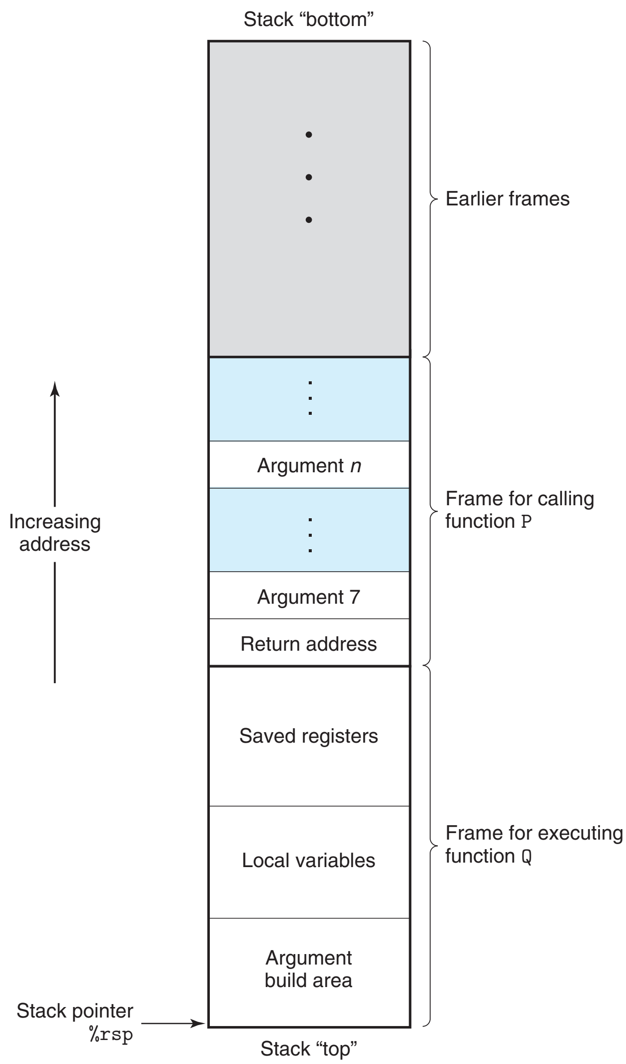


Figure 3.25
General stack frame structure. The stack can be used for passing arguments, for storing return information, for saving registers, and for local storage. Portions may be omitted when not needed.

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

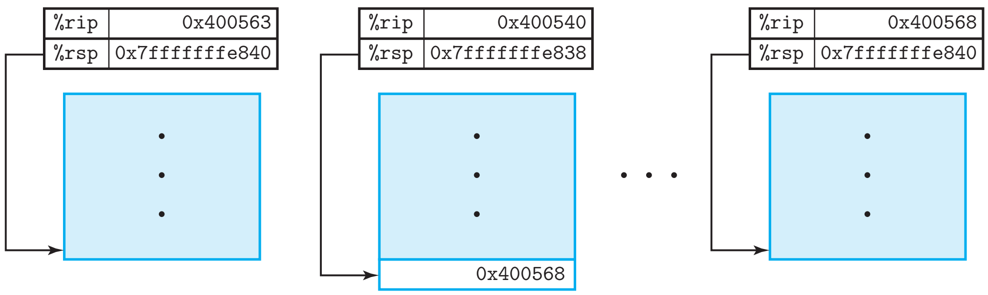
Passing control from function `P` to function `Q` involves simply setting the program counter (PC) to the starting address of the code for `Q`. However, when it later comes time for `Q` to return, the processor must have some record of the code location where it should resume the execution of `P`. This 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.



(a) Executing `call` (b) After `call` (c) After `ret`

Figure 3.26 Illustration of `call` and `ret` functions. The `call` instruction transfers control to the start of a function, while the `ret` instruction returns back to the instruction following the call.

Figure 3.26 illustrates the execution of the `call` and `ret` instructions for the `multstore` and `main` functions introduced in Section 3.2.2. The following are excerpts of the disassembled code for the two functions:

```
   Beginning of function multstore
  0000000000400540 <multstore>:
    400540:  53                   push   %rbx
    400541:  48 89 d3             mov    %rdx,%rbx
   . . .
   Return from function multstore
    40054d:  c3                   retq
   . . .
   Call to multstore from main
    400563:  e8 d8 ff ff ff       callq  400540 <multstore>
    400568:  48 8b 54 24 08       mov    0x8(%rsp),%rdx
```

In this code, we can see that the `call` instruction with address `0x400563` in `main` calls function `multstore`. This status is shown in Figure 3.26(a), with the indicated values for the stack pointer `%rsp` and the program counter `%rip`. The effect of the `call` is to push the return address `0x400568` onto the stack and to jump to the first instruction in function `multstore`, at address `0x0400540` (3.26(b)). The execution of function `multstore` continues until it hits the `ret` instruction at address `0x40054d`. This instruction pops the value `0x400568` from the stack and jumps to this address, resuming the execution of `main` just after the `call` instruction (3.26(c)).

As a more detailed example of passing control to and from procedures, Figure 3.27(a) shows the disassembled code for two functions, `top` and `leaf`, as well as the portion of code in function `main` where `top` gets called. Each instruction is identified by labels `L1`–`L2` (in `leaf`), `T1`–`T4` (in `top`), and `M1`–`M2` in `main`. Part (b) of the figure shows a detailed trace of the code execution, in which `main` calls `top(100)`, causing `top` to call `leaf(95)`. Function `leaf` returns 97 to `top`, which

(a) Disassembled code for demonstrating procedure calls and returns

```
  Disassembly of leaf(long y)
  y in %rdi
0000000000400540 <leaf>:
  400540:  48 8d 47 02          lea    0x2(%rdi),%rax   L1: z+2
  400544:  c3                   retq                    L2: Return

0000000000400545 <top>:
  Disassembly of top(long x)
  x in %rdi
  400545:  48 83 ef 05          sub    $0x5,%rdi        T1: x-5
  400549:  e8 f2 ff ff ff       callq  400540 <leaf>    T2: Call leaf(x-5)
  40054e:  48 01 c0             add    %rax,%rax        T3: Double result
  400551:  c3                   retq                    T4: Return

  . . .
  Call to top from function main
  40055b:  e8 e5 ff ff ff       callq  400545 <top>     M1: Call top(100)
  400560:  48 89 c2             mov    %rax,%rdx        M2: Resume
```

(b) Execution trace of example code

Instruction			State values (at beginning)				
Label	PC	Instruction	`%rdi`	`%rax`	`%rsp`	`*%rsp`	Description
M1	`0x40055b`	`callq`	100	—	`0x7fffffffe820`	—	Call `top(100)`
T1	`0x400545`	`sub`	100	—	`0x7fffffffe818`	`0x400560`	Entry of `top`
T2	`0x400549`	`callq`	95	—	`0x7fffffffe818`	`0x400560`	Call `leaf(95)`
L1	`0x400540`	`lea`	95	—	`0x7fffffffe810`	`0x40054e`	Entry of `leaf`
L2	`0x400544`	`retq`	—	97	`0x7fffffffe810`	`0x40054e`	Return 97 from `leaf`
T3	`0x40054e`	`add`	—	97	`0x7fffffffe818`	`0x400560`	Resume `top`
T4	`0x400551`	`retq`	—	194	`0x7fffffffe818`	`0x400560`	Return 194 from `top`
M2	`0x400560`	`mov`	—	194	`0x7fffffffe820`	—	Resume `main`

Figure 3.27 Detailed execution of program involving procedure calls and returns. Using the stack to store return addresses makes it possible to return to the right point in the procedures.

then returns 194 to `main`. The first three columns describe the instruction being executed, including the instruction label, the address, and the instruction type. The next four columns show the state of the program *before* the instruction is executed, including the contents of registers `%rdi`, `%rax`, and `%rsp`, as well as the value at the top of the stack. The contents of this table should be studied carefully, as they demonstrate the important role of the run-time stack in managing the storage needed to support procedure calls and returns.

Instruction `L1` of `leaf` sets `%rax` to 97, the value to be returned. Instruction `L2` then returns. It pops `0x400054e` from the stack. In setting the PC to this popped value, control transfers back to instruction `T3` of `top`. The program has successfully completed the call to `leaf` and returned to `top`.

Instruction `T3` sets `%rax` to 194, the value to be returned from `top`. Instruction `T4` then returns. It pops `0x4000560` from the stack, thereby setting the PC to instruction `M2` of `main`. The program has successfully completed the call to `top` and returned to `main`. We see that the stack pointer has also been restored to `0x7fffffffe820`, the value it had before the call to `top`.

We can see that this simple mechanism of pushing the return address onto the stack makes it possible for the function to later return to the proper point in the program. The standard call/return mechanism of C (and of most programming languages) conveniently matches the last-in, first-out memory management discipline provided by a stack.

Practice Problem 3.32 (solution page 375)

The disassembled code for two functions `first` and `last` is shown below, along with the code for a call of `first` by function `main`:

```
    Disassembly of last(long u, long v)
    u in %rdi, v in %rsi
0000000000400540 <last>:
  400540:  48 89 f8             mov    %rdi,%rax        L1: u
  400543:  48 0f af c6          imul   %rsi,%rax        L2: u*v
  400547:  c3                   retq                    L3: Return

    Disassembly of last(long x)
    x in %rdi
0000000000400548 <first>:
  400548:  48 8d 77 01          lea    0x1(%rdi),%rsi   F1: x+1
  40054c:  48 83 ef 01          sub    $0x1,%rdi        F2: x-1
  400550:  e8 eb ff ff ff       callq  400540 <last>    F3: Call last(x-1,x+1)
  400555:  f3 c3                repz retq               F4: Return
  .
  .
  .
  400560:  e8 e3 ff ff ff       callq  400548 <first>   M1: Call first(10)
  400565:  48 89 c2             mov    %rax,%rdx        M2: Resume
```

Each of these instructions is given a label, similar to those in Figure 3.27(a). Starting with the calling of `first(10)` by `main`, fill in the following table to trace instruction execution through to the point where the program returns back to `main`.

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.

ordering in the argument list. Arguments smaller than 64 bits can be accessed using the appropriate subsection of the 64-bit register. For example, if the first argument is 32 bits, it can be accessed as `%edi`.

When a function has more than six integral arguments, the other ones are passed on the stack. Assume that procedure `P` calls procedure `Q` with n integral arguments, such that $n > 6$. Then the code for `P` must allocate a stack frame with enough storage for arguments 7 through n, as illustrated in Figure 3.25. It copies arguments 1–6 into the appropriate registers, and it puts arguments 7 through n onto the stack, with argument 7 at the top of the stack. When passing parameters on the stack, all data sizes are rounded up to be multiples of eight. With the arguments in place, the program can then execute a `call` instruction to transfer control to procedure `Q`. Procedure `Q` can access its arguments via registers and possibly from the stack. If `Q`, in turn, calls some function that has more than six arguments, it can allocate space within its stack frame for these, as is illustrated by the area labeled "Argument build area" in Figure 3.25.

As an example of argument passing, consider the C function `proc` shown in Figure 3.29(a). This function has eight arguments, including integers with different numbers of bytes (8, 4, 2, and 1), as well as different types of pointers, each of which is 8 bytes.

The assembly code generated for `proc` is shown in Figure 3.29(b). The first six arguments are passed in registers. The last two are passed on the stack, as documented by the diagram of Figure 3.30. This diagram shows the state of the stack during the execution of `proc`. We can see that the return address was pushed onto the stack as part of the procedure call. The two arguments, therefore, are at positions 8 and 16 relative to the stack pointer. Within the code, we can see that different versions of the ADD instruction are used according to the sizes of the operands: `addq` for `a1` (`long`), `addl` for `a2` (`int`), `addw` for `a3` (`short`), and `addb` for `a4` (`char`). Observe that the `movl` instruction of line 6 reads 4 bytes from memory; the following `addb` instruction only makes use of the low-order byte.

Practice Problem 3.33 (solution page 375)

A C function `procprob` has four arguments `u`, `a`, `v`, and `b`. Each is either a signed number or a pointer to a signed number, where the numbers have different sizes. The function has the following body:

```
*u += a;
*v += b;
return sizeof(a) + sizeof(b);
```

It compiles to the following x86-64 code:

```
procprob:
  movslq  %edi, %rdi
  addq    %rdi, (%rdx)
  addb    %sil, (%rcx)
```

(a) C code

```
void proc(long  a1, long  *a1p,
          int   a2, int   *a2p,
          short a3, short *a3p,
          char  a4, char  *a4p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```

(b) Generated assembly code

```
     void proc(a1, a1p, a2, a2p, a3, a3p, a4, a4p)
     Arguments passed as follows:
       a1  in %rdi          (64 bits)
       a1p in %rsi          (64 bits)
       a2  in %edx          (32 bits)
       a2p in %rcx          (64 bits)
       a3  in %r8w          (16 bits)
       a3p in %r9           (64 bits)
       a4  at %rsp+8        ( 8 bits)
       a4p at %rsp+16       (64 bits)
proc:
  movq    16(%rsp), %rax    Fetch a4p    (64 bits)
  addq    %rdi, (%rsi)      *a1p += a1   (64 bits)
  addl    %edx, (%rcx)      *a2p += a2   (32 bits)
  addw    %r8w, (%r9)       *a3p += a3   (16 bits)
  movl    8(%rsp), %edx     Fetch a4     ( 8 bits)
  addb    %dl, (%rax)       *a4p += a4   ( 8 bits)
  ret                       Return
```

Figure 3.29 Example of function with multiple arguments of different types. Arguments 1–6 are passed in registers, while arguments 7–8 are passed on the stack.

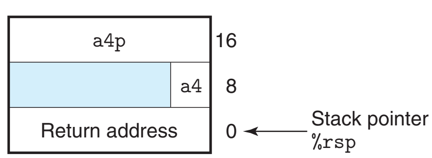


Figure 3.30
Stack frame structure for function proc. Arguments a4 and a4p are passed on the stack.

```
    movl    $6, %eax
    ret
```

Determine a valid ordering and types of the four parameters. There are two correct answers.

3.7.4 Local Storage on the Stack

Most of the procedure examples we have seen so far did not require any local storage beyond what could be held in registers. At times, however, local data must be stored in memory. Common cases of this include these:

- There are not enough registers to hold all of the local data.
- The address operator '`&`' is applied to a local variable, and hence we must be able to generate an address for it.
- Some of the local variables are arrays or structures and hence must be accessed by array or structure references. We will discuss this possibility when we describe how arrays and structures are allocated.

Typically, a procedure allocates space on the stack frame by decrementing the stack pointer. This results in the portion of the stack frame labeled "Local variables" in Figure 3.25.

As an example of the handling of the address operator, consider the two functions shown in Figure 3.31(a). The function `swap_add` swaps the two values designated by pointers `xp` and `yp` and also returns the sum of the two values. The function `caller` creates pointers to local variables `arg1` and `arg2` and passes these to `swap_add`. Figure 3.31(b) shows how `caller` uses a stack frame to implement these local variables. The code for `caller` starts by decrementing the stack pointer by 16; this effectively allocates 16 bytes on the stack. Letting S denote the value of the stack pointer, we can see that the code computes `&arg2` as $S + 8$ (line 5), `&arg1` as S (line 6). We can therefore infer that local variables `arg1` and `arg2` are stored within the stack frame at offsets 0 and 8 relative to the stack pointer. When the call to `swap_add` completes, the code for `caller` then retrieves the two values from the stack (lines 8–9), computes their difference, and multiplies this by the value returned by `swap_add` in register `%rax` (line 10). Finally, the function deallocates its stack frame by incrementing the stack pointer by 16 (line 11.) We can see with this example that the run-time stack provides a simple mechanism for allocating local storage when it is required and deallocating it when the function completes.

As a more complex example, the function `call_proc`, shown in Figure 3.32, illustrates many aspects of the x86-64 stack discipline. Despite the length of this example, it is worth studying carefully. It shows a function that must allocate storage on the stack for local variables, as well as to pass values to the 8-argument function `proc` (Figure 3.29). The function creates a stack frame, diagrammed in Figure 3.33.

Looking at the assembly code for `call_proc` (Figure 3.32(b)), we can see that a large portion of the code (lines 2–15) involves preparing to call function

(a) Code for `swap_add` and calling function

```
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

(b) Generated assembly code for calling function

```
   long caller()
   caller:
     subq    $16, %rsp          Allocate 16 bytes for stack frame
     movq    $534, (%rsp)       Store 534 in arg1
     movq    $1057, 8(%rsp)     Store 1057 in arg2
     leaq    8(%rsp), %rsi      Compute &arg2 as second argument
     movq    %rsp, %rdi         Compute &arg1 as first argument
     call    swap_add           Call swap_add(&arg1, &arg2)
     movq    (%rsp), %rdx       Get arg1
     subq    8(%rsp), %rdx      Compute diff = arg1 - arg2
     imulq   %rdx, %rax         Compute sum * diff
     addq    $16, %rsp          Deallocate stack frame
     ret                        Return
```

Figure 3.31 Example of procedure definition and call. The calling code must allocate a stack frame due to the presence of address operators.

`proc`. This includes setting up the stack frame for the local variables and function parameters, and for loading function arguments into registers. As Figure 3.33 shows, local variables `x1`–`x4` are allocated on the stack and have different sizes. Expressing their locations as offsets relative to the stack pointer, they occupy bytes 24–31 (`x1`), 20–23 (`x2`), 18–19 (`x3`), and 17 (`s3`). Pointers to these locations are generated by `leaq` instructions (lines 7, 10, 12, and 14). Arguments 7 (with value 4) and 8 (a pointer to the location of `x4`) are stored on the stack at offsets 0 and 8 relative to the stack pointer.

(a) C code for calling function

```
long call_proc()
{
    long  x1 = 1; int  x2 = 2;
    short x3 = 3; char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

(b) Generated assembly code

```
    long call_proc()
    call_proc:
      Set up arguments to proc
      subq    $32, %rsp           Allocate 32-byte stack frame
      movq    $1, 24(%rsp)        Store 1 in &x1
      movl    $2, 20(%rsp)        Store 2 in &x2
      movw    $3, 18(%rsp)        Store 3 in &x3
      movb    $4, 17(%rsp)        Store 4 in &x4
      leaq    17(%rsp), %rax      Create &x4
      movq    %rax, 8(%rsp)       Store &x4 as argument 8
      movl    $4, (%rsp)          Store 4 as argument 7
      leaq    18(%rsp), %r9       Pass &x3 as argument 6
      movl    $3, %r8d            Pass 3 as argument 5
      leaq    20(%rsp), %rcx      Pass &x2 as argument 4
      movl    $2, %edx            Pass 2 as argument 3
      leaq    24(%rsp), %rsi      Pass &x1 as argument 2
      movl    $1, %edi            Pass 1 as argument 1
      Call proc
      call    proc
      Retrieve changes to memory
      movslq  20(%rsp), %rdx      Get x2 and convert to long
      addq    24(%rsp), %rdx      Compute x1+x2
      movswl  18(%rsp), %eax      Get x3 and convert to int
      movsbl  17(%rsp), %ecx      Get x4 and convert to int
      subl    %ecx, %eax          Compute x3-x4
      cltq                        Convert to long
      imulq   %rdx, %rax          Compute (x1+x2) * (x3-x4)
      addq    $32, %rsp           Deallocate stack frame
      ret                         Return
```

Figure 3.32 Example of code to call function `proc`, defined in Figure 3.29. This code creates a stack frame.

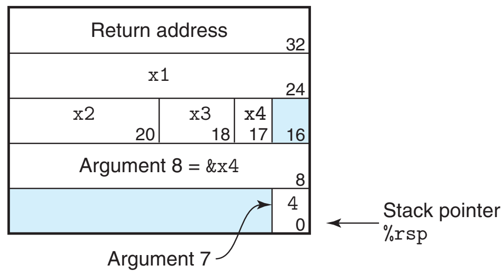


Figure 3.33
Stack frame for function `call_proc`. The stack frame contains local variables, as well as two of the arguments to pass to function `proc`.

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),

(a) Calling function

```
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

(b) Generated assembly code for the calling function

```
     long P(long x, long y)
     x in %rdi, y in %rsi
1    P:
2      pushq   %rbp              Save %rbp
3      pushq   %rbx              Save %rbx
4      subq    $8, %rsp          Align stack frame
5      movq    %rdi, %rbp        Save x
6      movq    %rsi, %rdi        Move y to first argument
7      call    Q                 Call Q(y)
8      movq    %rax, %rbx        Save result
9      movq    %rbp, %rdi        Move x to first argument
10     call    Q                 Call Q(x)
11     addq    %rbx, %rax        Add saved Q(y) to Q(x)
12     addq    $8, %rsp          Deallocate last part of stack
13     popq    %rbx              Restore %rbx
14     popq    %rbp              Restore %rbp
15     ret
```

Figure 3.34 **Code demonstrating use of callee-saved registers.** Value `x` must be preserved during the first call, and value `Q(y)` must be preserved during the second.

we can see that the code generated by GCC uses two callee-saved registers: `%rbp` to hold `x`, and `%rbx` to hold the computed value of `Q(y)`. At the beginning of the function, it saves the values of these two registers on the stack (lines 2–3). It copies argument `x` to `%rbp` before the first call to `Q` (line 5). It copies the result of this call to `%rbx` before the second call to `Q` (line 8). At the end of the function (lines 13–14), it restores the values of the two callee-saved registers by popping them off the stack. Note how they are popped in the reverse order from how they were pushed, to account for the last-in, first-out discipline of a stack.

Practice Problem 3.34 (solution page 376)

Consider a function `P`, which generates local values, named `a0`–`a8`. It then calls function `Q` using these generated values as arguments. GCC produces the following code for the first part of `P`:

```
long P(long x)
x in %rdi
P:
  pushq   %r15
  pushq   %r14
  pushq   %r13
  pushq   %r12
  pushq   %rbp
  pushq   %rbx
  subq    $24, %rsp
  movq    %rdi, %rbx
  leaq    1(%rdi), %r15
  leaq    2(%rdi), %r14
  leaq    3(%rdi), %r13
  leaq    4(%rdi), %r12
  leaq    5(%rdi), %rbp
  leaq    6(%rdi), %rax
  movq    %rax, (%rsp)
  leaq    7(%rdi), %rdx
  movq    %rdx, 8(%rsp)
  movl    $0, %eax
  call    Q
      . . .
```

A. Identify which local values get stored in callee-saved registers.

B. Identify which local values get stored on the stack.

C. Explain why the program could not store all of the local values in callee-saved registers.

3.7.6 Recursive Procedures

The conventions we have described for using the registers and the stack allow x86-64 procedures to call themselves recursively. Each procedure call has its own private space on the stack, and so the local variables of the multiple outstanding calls do not interfere with one another. Furthermore, the stack discipline naturally provides the proper policy for allocating local storage when the procedure is called and deallocating it before returning.

Figure 3.35 shows both the C code and the generated assembly code for a recursive factorial function. We can see that the assembly code uses register `%rbx` to hold the parameter `n`, after first saving the existing value on the stack (line 2) and later restoring the value before returning (line 11). Due to the stack discipline, and the register-saving conventions, we can be assured that when the recursive call to `rfact(n-1)` returns (line 9) that (1) the result of the call will be held in register

(a) C code

```
long rfact(long n)
{
    long result;
    if (n <= 1)
        result = 1;
    else
        result = n * rfact(n-1);
    return result;
}
```

(b) Generated assembly code

```
  long rfact(long n)
  n in %rdi
rfact:
  pushq   %rbx              Save %rbx
  movq    %rdi, %rbx        Store n in callee-saved register
  movl    $1, %eax          Set return value = 1
  cmpq    $1, %rdi          Compare n:1
  jle     .L35              If <=, goto done
  leaq    -1(%rdi), %rdi    Compute n-1
  call    rfact             Call rfact(n-1)
  imulq   %rbx, %rax        Multiply result by n
.L35:                     done:
  popq    %rbx              Restore %rbx
  ret                       Return
```

Figure 3.35 **Code for recursive factorial program.** The standard procedure handling mechanisms suffice for implementing recursive functions.

`%rax`, and (2) the value of argument `n` will held in register `%rbx`. Multiplying these two values then computes the desired result.

We can see from this example that calling a function recursively proceeds just like any other function call. Our stack discipline provides a mechanism where each invocation of a function has its own private storage for state information (saved values of the return location and callee-saved registers). If need be, it can also provide storage for local variables. The stack discipline of allocation and deallocation naturally matches the call-return ordering of functions. This method of implementing function calls and returns even works for more complex patterns, including mutual recursion (e.g., when procedure `P` calls `Q`, which in turn calls `P`).

Practice Problem 3.35 (solution page 376)

For a C function having the general structure

```
long rfun(unsigned long x) {
    if ( __________ )
        return __________;
    unsigned long nx = __________;
    long rv = rfun(nx);
    return __________;
}
```

GCC generates the following assembly code:

```
     long rfun(unsigned long x)
     x in %rdi
rfun:
  pushq   %rbx
  movq    %rdi, %rbx
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L2
  shrq    $2, %rdi
  call    rfun
  addq    %rbx, %rax
.L2:
  popq    %rbx
  ret
```

A. What value does `rfun` store in the callee-saved register `%rbx`?

B. Fill in the missing expressions in the C code shown above.

3.8 Array Allocation and Access

Arrays in C are one means of aggregating scalar data into larger data types. C uses a particularly simple implementation of arrays, and hence the translation into machine code is fairly straightforward. One unusual feature of C is that we can generate pointers to elements within arrays and perform arithmetic with these pointers. These are translated into address computations in machine code.

Optimizing compilers are particularly good at simplifying the address computations used by array indexing. This can make the correspondence between the C code and its translation into machine code somewhat difficult to decipher.

3.8.1 Basic Principles

For data type T and integer constant N, consider a declaration of the form

T `A[`N`];`