9 Subroutines and Control Abstraction

9.2.1 Displays

EXAMPLE 9.67
Nonlocal access using a display

As noted in the main text, a display is an embedding of the static chain into an array. The jth element of the display contains a reference to the frame of the most recently active subroutine at lexical nesting level j. The first element of the display is thus a reference to the frame of some subroutine S nested directly inside the main program; the second element is a reference to the frame of a routine that is nested inside of S, and so forth, until we reach the currently active routine. Figure C-9.7 contains an example.

If the display is stored in memory, then a nonlocal object can be loaded into a register with two memory accesses: one to load the display element into a register, the second to load the object. On a machine with a large number of registers, one might be tempted to reduce the overhead to only one memory access by keeping the entire display in registers, but that would probably be a bad idea: display elements tend to be accessed much less frequently than other things (e.g., local variables) that might be kept in the registers instead.

Maintaining the Display

Maintenance of a display is slightly more complicated than maintenance of a static chain, but not by much. Perhaps the most obvious approach would be to maintain the static chain as usual, and simply fill the display at procedure entry and exit, by walking down the chain. In most cases, however, the following (much faster) scheme suffices: when calling a subroutine at lexical nesting level j, the callee saves the current value of the jth display element into the stack, and then replaces that element with a copy of its own (newly created) frame pointer. Before returning, it restores the old element. Why does this mechanism work? As with static chains, there are two cases to consider:

1. The callee is nested (directly) inside the caller. In this case the caller and the callee share all display elements up to the current level. Putting the callee's frame pointer into the display simply extends the current level by one. It is conceivable that the old value needn't be saved, but in general there is no way

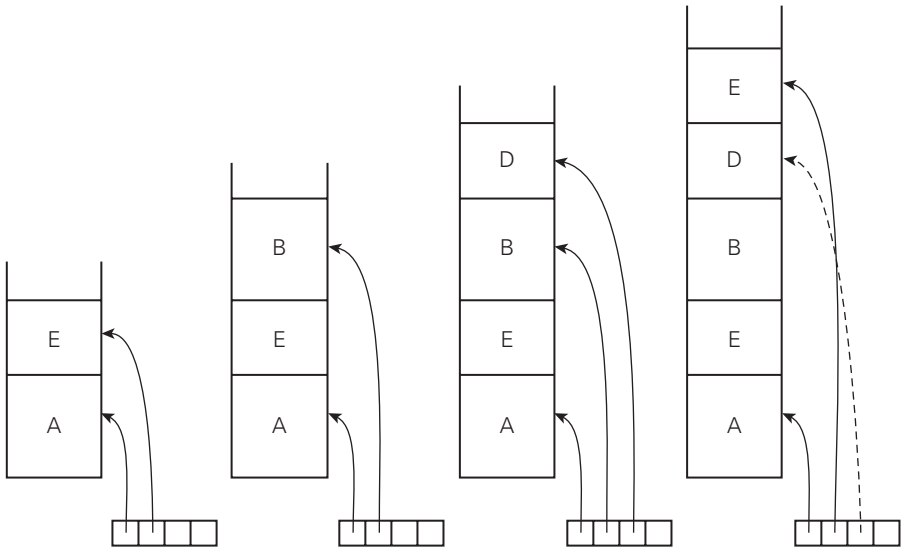

Figure 9.7 **Nonlocal access using a display.** The stack configurations, from left to right, illustrate the contents of the display (at bottom) for a sequence of subroutine calls, assuming the lexical nesting of Figure 9.1. Display elements beyond that of the currently executing subroutine are not used.

to tell. The caller itself might have been called by code that is very deeply nested, and that is counting on the integrity of a very deep display, in which case the old display element *will* be needed. A smart compiler may be able to avoid the save in certain circumstances.

2. The callee is at lexical nesting level j, $k \geq 0$ levels out from the caller. In this case the caller and callee share all display elements up through $j - 1$. The caller's

DESIGN & IMPLEMENTATION

9.9 Lexical nesting and displays

Because the display is a fixed-size array, compilers that use a display to implement access to nonlocal objects generally impose a limit (the size of the display) on the maximum depth to which subroutines may be nested. If this limit is larger than, say, five or six, it is unlikely that any programmer will ever wish for more. Note that the display does not eliminate the need for a frame pointer. Because local variables are accessed so often, it is important to have the address of the current frame in a register, where it can be used for displacement-mode addressing. Similarly, on a RISC processor, where a 32-bit address will not fit in one instruction, it is important to maintain a base register for the most commonly accessed global variables as well.

entry at level j is different from the callee's, so the callee must save it before storing its own frame pointer. If the callee in turn calls a routine at level $j + 1$, that routine will change another element of the display, but all old elements will be restored before they are needed again.

If the callee is a leaf routine then the display can be left intact; no one will use the element corresponding to the callee's nesting level before control returns to the caller.

Closures

A subroutine that is passed as a parameter, stored in a variable, or returned from a function must be called through some sort of *closure* (Section 3.6) that captures the referencing environment. In a language implementation based on static chains, a closure can be represented as a ⟨code address, static link⟩ pair. Displays are not as simple. A standard technique is to create two "entry points"—starting addresses—for every subroutine. One of these is for "normal" calls, the other for calls through closures. When a closure is created, it contains the address of the alternative entry point. The code at that entry point saves elements 1 through j of the display into the stack (it will have to create a larger-than-normal stack frame in order to do this), and then replaces those elements with values taken from (or calculated from) the closure. The alternative entry then makes a nested call to the main body of the subroutine (it skips the code immediately following the normal entry—the code that creates the normal stack frame and updates the display). When the subroutine returns, it comes back to the code of the alternative entry, which restores the old value of the display before returning to the actual caller.

More space-conserving implementations of display-based closures are possible (see Exercise C-9.29), but with higher run-time overhead.

Comparison to Static Chains

In general, maintaining a display is slightly more expensive than maintaining a static chain, though the comparison is not absolute. In the usual case, passing a static link to a called routine requires $k \geq 0$ load instructions in the caller, followed by one store instruction in the callee (to place the static link at the appropriate offset in the stack frame). The store may be skipped in leaf routines, assuming that a register is available to hold the link as long as it is needed. No overhead is required to maintain the static chain when returning from a subroutine. With a display, a nonleaf callee requires two loads and three stores (1 + 2 in the prologue and 1 + 1 epilogue) to save and restore display elements. Because the callee does all the work, displays may save a little bit on code size, compared to static chains. As noted above, displays significantly complicate the creation and use of closures.

The original advantage of displays—reduced cost for access to objects in outer scopes—seems less clear today than once it did. In fact, while displays were popular in the CISC compilers of the 1970s and 1980s, they are less common in recent compilers. Most programs don't nest subroutines more than two or three levels deep, so static chains are seldom very long, and variables in surrounding scopes tend

not to be accessed very often. If they *are* accessed often, common subexpression optimizations (to be discussed in Chapter 17) are likely to ensure that a pointer to the appropriate frame remains in a register.

Some language designers have argued that the development of object-oriented programming (the subject of Chapter 10) has eliminated the need for nested subroutines [Han81]. Others might even say that the success of C has shown such routines to be unneeded. Without nested subroutines, of course, the choice between static chains and displays is moot.

CHECK YOUR UNDERSTANDING

46. Describe how we access an object at lexical nesting level k in a language implementation based on displays.
47. Why isn't the display typically kept in registers?
48. Explain how to maintain the display during subroutine calls.
49. What special concerns arise when creating closures in a language implementation that uses displays?
50. Summarize the tradeoffs between displays and static chains. Describe a program for which displays will result in faster code. Describe another for which static chains will be faster.

9 Subroutines and Control Abstraction

9.2.2 Stack Case Studies: LLVM on Arm; `gcc` on x86

To make stack management a bit more concrete, we present a pair of case studies: Apple's LLVM-based C compiler for the iPhone (Arm) and the GNU compiler suite for 32- and 64-bit x86. Both examples follow the general scheme outlined in Section 3.2.2, with differences in details that reflect the history of the respective compilers and the architecture of the target machines.

LLVM on Arm

An overview of the Arm instruction set architecture (ISA) can be found in Section C-5.4.5. For the sake of interoperability, Arm Ltd. publishes a standard for subroutine calling sequences that allows code from different vendors and compilers to link and run together. The standard has several variants, reflecting hardware features (Thumb mode, floating-point or vector instructions and registers, dynamic linking) that may or may not be present on a given processor or in its software environment. We focus here on the conventions adopted by Apple's C compiler for the iPhone and iPad (version 4.2), at optimization level `-O2`. The Apple compiler uses the 32-bit Arm back end (version 3.2svn) of the LLVM compiler suite. Given the level of detail in Arm's standard, code produced by other compilers is likely to be quite similar. Note, however, that the conventions for 64-bit code are very different; they are not documented here.

As noted in Section C-5.4.5, register `r14` (also known as `lr`) is special-cased by the ISA to receive the return address in subroutine call (`bl`—branch-and-link) instructions. Register `r13` (also known as `sp`) is similarly reserved for use as the stack pointer. It is not modified by `bl` instructions, but several variants of `push` and `pop`, which do update `sp`, are commonly part of the subroutine calling sequence. Some compilers for Arm, though not all, dedicate a third register by convention for use as a frame pointer; LLVM uses `r7` for this purpose.

EXAMPLE 9.68
LLVM/Arm stack layout

A typical LLVM/Arm stack frame appears in Figure C-9.8. The `sp` register points to the *last used* location in the stack (note that some compilers aim the pointer at

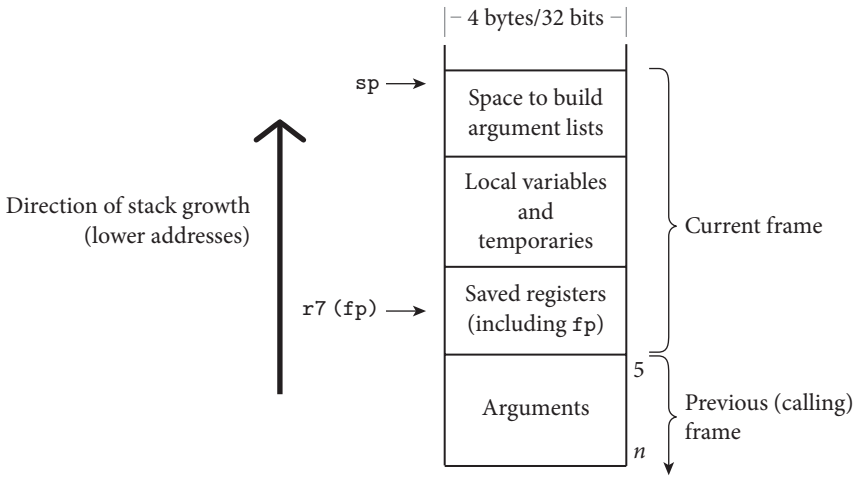

Figure 9.8 Layout of the subroutine call stack for Apple's LLVM-based C compiler for Arm, running in 32-bit mode. As in Figure 9.2, lower addresses are toward the top of the page.

the *first unused* location). Arm's subroutine calling standard requires that the stack always be word-aligned (`sp` $\bmod 4 = 0$). At an external call (to a subroutine in a different compilation unit) it must be double-word aligned (`sp` $\bmod 8 = 0$).

The first four arguments to a subroutine are always passed in registers. Additional arguments may be passed on the stack, with the last argument in the deepest location. Space for stack-based arguments is considered a part of the calling routine. If the current routine is not a leaf, space for any stack-based arguments it needs to pass to additional routines is preallocated, at the top (lowest-addressed-end) of the frame, as part of the subroutine prologue.

Space for local variables and for any temporary values that will not fit in registers is allocated in the middle of the frame. If the subroutine ever applies an address-of operator (`&` in C) to a low-numbered argument (one that will have been passed in a register), or if it passes such an argument to another routine by reference, the compiler creates a local variable to hold the argument, and initializes it with the value passed in the register.

Any callee-saves registers that may be overwritten by the current routine are saved at the bottom of the frame, directly beyond any stack-based arguments. The frame pointer (`r7`) is typically among these. LLVM arranges for the current `fp` to point to the location of the saved `fp`.

Argument Passing Conventions Arguments and locals of the current subroutine are accessed via offsets from the `fp`. Arguments in the process of being passed to the next routine are assembled at the top of the frame, and are accessed via offsets from the `sp`. The first four arguments are passed in registers `r0`–`r3`. A double-precision floating-point number is divided into two 32-bit halves, and passed as if it were two integers. (Some other Arm compilers pass floating-point

arguments in the floating-point registers.) Records (`structs`) that appear early in the argument list may also be split into 32-bit pieces, and passed in multiple registers. An argument may be split between registers and the stack, if part but not all of it will fit in registers.

The argument build area at the top of the frame is designed to be large enough to hold the largest argument list that may be passed to any called routine. This convention may waste a bit of space in certain cases, but it ensures that arguments never need to be "pushed" in the usual sense of the word: the `sp` does not change when they are placed into the stack.

Return values up to 4 bytes in length occupy register `r0`. Double-word scalar return values occupy register pair `r0–r1`; quad-word scalar return values occupy registers `r0–r3`. Record-type return values of more than four bytes are placed in memory, at a location chosen by the caller and passed as an extra, hidden argument. If the return value is to be assigned immediately into a variable (e.g., `x = foo()`), the caller can simply pass the address of the variable. If the value is to be passed in turn to another subroutine, the caller can pass the appropriate address within its own argument build area. (Writing the return value into this space will probably destroy the returning function's own arguments, but that's fine in the absence of call-by-value/result: at this point the arguments are no longer needed.) Finally, though one doesn't see this idiom often (and most languages don't support it), C allows the caller to extract a field directly from the return value of a function (e.g., `x = foo().a + y;`); in this case the caller must pass the address of a temporary location within the "local variables and temporaries" part of its stack frame.

DESIGN & IMPLEMENTATION

9.10 Leveraging `pc = r15`

Because Arm assigns a register number to the program counter, that counter can be read and written (almost) like any other register. Writes to the `pc` cause a branch in control. This convention, together with the choice of `lr` = `r14` and `pc` = `r15`, enables an interesting optimization. If a subroutine is not a leaf (i.e., it calls another routine), `lr` will be among the registers saved at the bottom of the frame. If we suppose, for concreteness, that the subroutine plans to overwrite callee-saves registers `r4` and `r5`, and we know that we need to update the frame pointer (`r7`), then the subroutine prologue is likely to contain a `push {r4, r5, r7, lr}` instruction. This instruction stores the registers in sorted order, with the highest-numbered register (in this case, `lr`) at the highest address—deepest in the stack. One might naturally expect the epilogue to contain a symmetric `pop {r4, r5, r7, lr}` instruction, followed immediately by `bx lr` (branch to location in `lr`). But since the `pc` and `lr` have adjacent register numbers, the compiler can—and typically does—achieve the same result with a single `pop {r4, r5, r7, pc}` instruction.

Arm and Thumb Mode Switching One of the more unusual features of the 32-bit Arm ISA (as described in Section C-5.4.5) is the presence of two separate instruction encodings. As on most RISC machines, the A32 encoding represents each instruction with 32 bits. The alternative T32 encoding, also known as "Thumb," represents the most common instructions in only 16 bits; the resulting improvement in code density can be important in embedded applications. While the two encodings are quite different (and in particular, T32 is not a subset of A32), program fragments that use different encodings can be linked into a single program.

To switch from one format to another, the program uses special `bx` (branch and exchange instruction set) and `blx` (branch with link and exchange instruction set) instructions. When the target address is statically known, the assumption is that the programmer knows that the source and target encodings are different, so the processor needs to change modes in the course of performing the branch. When the target address is in a register (as it will be when returning from a subroutine, or when calling through a pointer, a virtual method table, or a closure), Arm exploits the fact that instructions never appear at an odd address (T32 instructions are always word aligned; A32 instructions are always longword aligned). Because the least significant bit of the target address must always be 0, this bit can be used in the register to specify the target instruction set: 0 means A32; 1 means T32.

EXAMPLE 9.69
LLVM/Arm calling sequence

Calling Sequence Details The calling sequence to maintain the LLVM/Arm stack is as follows. The caller

1. saves (into the "local variables and temporaries" part of its frame) any caller-saves registers whose values are still needed
2. puts up to four small arguments (or "chunks" of larger arguments) into registers `r0–r3`
3. stores the remaining arguments into the argument build area at the top of the current frame
4. performs a `bl` or `blx` instruction, which puts the return address in register `lr`, jumps to the target address, and optionally changes instruction set encoding

On 32-bit Arm, the caller-saves registers are just the ones that are used for arguments—namely, `r0–r3`. In a language with nested subroutines (not supported by Apple's compiler), the caller would need to place the static link into another register immediately before performing the `bl` or `blx`.

In its prologue, the callee

1. pushes any necessary registers onto the stack
2. initializes the frame pointer by adding an appropriate small constant to the `sp`, placing the result in `r7`
3. subtracts enough from the `sp` to make space for local variables, temporaries, and the argument build area at the top of the stack, rounding down to a lower address if necessary to ensure that these objects have appropriate alignment

Saved registers include (a) the frame pointer, `r7` (assuming the current routine needs a frame pointer of its own); (b) any callee-saves registers (`r4`–`r6` and `r8`–`r11`) whose values may be changed before returning; and (c) the link register, `lr`, if the current routine is not a leaf, or if it uses `lr` as an additional temporary.

In its epilogue, immediately before returning, the callee

1. places the function return value (if any) into `r0`–`r3` or memory, as appropriate
2. subtracts a small constant from `r7`, placing the result in `sp`; this effectively deallocates the bulk of the frame
3. pops saved registers from the stack, with the `pc` taking the place held by `lr` in the corresponding save in the prologue; this has the side effect of branching back to the caller (see Sidebar C-9.10)

Finally, if appropriate, the caller moves the return value to wherever it is needed. Caller-saves registers are restored lazily over time, as their values are needed.

To support the use of symbolic debuggers, the compiler generates a wealth of symbol table information, in the open-source DWARF format [DWA17]. It embeds this information into the object file. The information is most accurate when the program is compiled without any code improvement (`-O0`). For each subroutine, the information includes the starting and ending addresses of the routine; the name, type, and location (register name or frame pointer offset) of every formal parameter and local variable; the set of instructions corresponding to each line of source code; the size and layout of the stack frame; and a list of which registers were saved.

gcc on x86

To illustrate the differences among compilers and architectures, our second case study considers the GNU compiler collection (`gcc`, version 4.8.1) on the x86. We begin with 32-bit code and then explain the differences that obtain on 64-bit machines. Our example again focuses mostly on C, which acts as sort of a "lowest common denominator" among high-level languages. We also consider nested subroutines and closures, however, since these appear in some of the collection's supported languages.

An overview of the x86-32 ISA appears in Section C-5.4.5. Given the machine's CISC heritage and the comparatively small number of registers (only six are available for general-purpose use), all arguments are passed on the stack when running in 32-bit mode. To give the compiler the freedom to evaluate arguments out of order when desired, recent versions of `gcc` employ an argument build area similar to that of the LLVM case study. Unlike LLVM, recent versions of `gcc` omit the use of a separate frame pointer by default, making register `ebp` (`rbp` in 64-bit mode) available for other purposes; exceptions occur when specified by the programmer (using the `-no-omit-frame-pointer` command-line switch), when compiling at optimization levels `-O0` and `-O1`, when a subroutine has a local variable whose size is not known at compile time (Figure 8.7), or when a subroutine calls `alloca` (a legacy mechanism to create temporary space within the current stack frame).

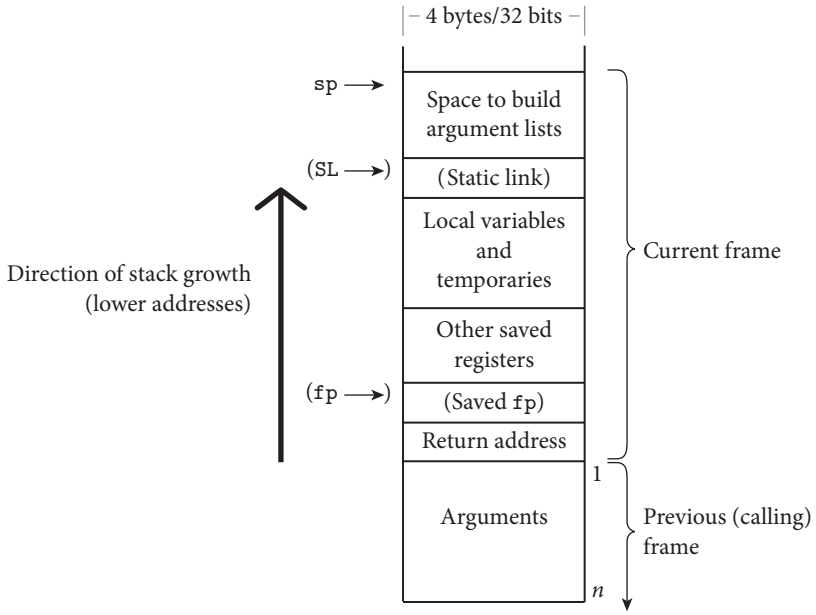

Figure 9.9 **Layout of the subroutine call stack for the GNU Compiler Collection (`gcc`) on 32-bit x86.** The return address is present in all frames. All other parts of the frame are optional; they are present only if required by the current subroutine. In x86 terminology, the `sp` is named `esp`; the `fp` is `ebp` (extended base pointer). The static link, in languages with nested subroutines, is passed in register `ecx`. `SL` marks the location that will be referenced by the static link (if any) of any subroutine nested immediately inside this one. A routine that is neither innermost nor outermost will save its own static link at the location referenced by the static link of its children.

Historically, omission of the frame pointer made it difficult or even impossible for symbolic debuggers to perform a "backtrace" operation (identifying the frames of calling routines), but this limitation has been removed with modern debugging standards like DWARF.

Calling sequences for the x86 vary from vendor to vendor, and have evolved considerably over time, as changes in microarchitecture changed performance tradeoffs. Most modern sequences use the `call` and `ret` instructions. The former pushes the return address onto the stack, updating the `sp`, and branches to the called routine. The latter pops the return address off the stack, again updating the `sp`, and branches back to the caller. Several additional, more complex instructions, retained for backward compatibility, are typically not generated by modern compilers, because they were designed for calling sequences with an explicit display and without an argument build area, or because they don't pipeline as well as equivalent sequences of simpler instructions.

EXAMPLE 9.70
`gcc`/x86-32 stack layout

Argument Passing Conventions Figure C-9.9 shows a stack frame for the x86-32. As in the LLVM case study, the `sp` points to the last used location on the stack.

Arguments in the process of being passed to another routine are accessed via offsets from the sp; everything else is accessed via offsets from the fp, if present—otherwise the sp. All arguments are passed in the stack. In languages (Ada, in particular) that permit nested subroutines, register ecx is used to pass the static link. If the current routine has at least one lexically nested child and is itself lexically nested in some parent, then a copy of the static link will be saved into the stack just above (at a lower address than) the area used for local variables and temporaries. When a nested routine is running, its own static link will point to the saved link in this current routine, or to the local variables and temporaries, if this current routine is outermost.

Functions return integer or pointer values in register eax. Floating-point values are returned in the first of the "x87" floating-point registers, st(0). Composite values (records, arrays, etc.) of 8 bytes or less are returned in the register pair eax–edx, as are "long long" (64-bit) integers. For larger return values (records, arrays, etc.), the compiler passes a hidden first argument (on the stack) whose value is the address at which the return value should be written. ■

EXAMPLE 9.71
gcc/x86-32 calling sequence

Calling Sequence Details The calling sequence to maintain the gcc/x86-32 stack is as follows. The caller

1. saves (into the "local variables and temporaries" part of its frame) any caller-saves registers whose values are still needed
2. puts arguments into the build area at the top of the current frame
3. places the static link (if any) in register ecx
4. executes a call instruction

The caller-saves registers consist of eax, edx, and ecx. Step 1 is skipped if none of these contain a value that will be needed later. Step 2 is skipped if the subroutine has no parameters. Step 3 is skipped if the language has no nested subroutines, or if the called routine is declared at the outermost nesting level. The call instruction pushes the return address and jumps to the subroutine.

In its prologue, the callee

1. pushes the fp onto the stack (if the current routine uses the fp), implicitly decrementing the sp by 4 (one word).
2. copies the sp into the fp if necessary, thereby establishing a frame pointer for the current routine
3. pushes any callee-saves registers whose values may be overwritten by the current routine
4. pushes the static link (ecx) if the language has nested subroutines and this is not a leaf
5. subtracts the remainder of the frame size from the sp

The callee-saves registers are ebx, esi, edi, and, for routines that don't need a frame pointer, ebp. For routines that do need a frame pointer, registers esp and

ebp (the sp and fp, respectively) are saved by Steps 1 and 2. The instructions for some of these steps may be replaced with equivalent sequences by the compiler's code improver, and may be mixed into the rest of the subroutine by the instruction scheduler. In particular, if the value subtracted from the sp in Step 5 is made large enough to accommodate the callee-saves registers, then the pushes in Steps 3 and 4 may be moved after Step 5 and replaced with fp- or sp-relative stores.

In its epilogue, the callee

1. sets the return value
2. restores any callee-saved registers
3. copies the fp into the sp, or subtracts a constant from the sp, as appropriate, thereby deallocating the frame
4. pops the fp, if any, off the stack
5. returns

Steps 3 and 4 may be effected on the x86 by a single leave instruction. As in the previous case study, the caller moves the return value, if it is in a register, to wherever it is needed. It restores any caller-saves registers lazily over time. ■

EXAMPLE 9.72
Subroutine closure trampoline

Because Ada allows subroutines to nest (and Ada 2005 allows arbitrary subroutines to be passed as parameters), a subroutine *S* that is passed as a parameter from *P* to *Q* must be represented by a closure, as described in Section 3.6.1. In many compilers the closure is a data structure containing the address of *S* and the static link that should be used when *S* is called. In gcc, however, the closure contains an *x86 code sequence* known as a *trampoline*—typically a pair of instructions to load ecx with the appropriate static link and then jump to the beginning of *S*. The trampoline resides in the "local variables and temporaries" section of *P*'s activation record. Its address is passed to *Q*. Rather than "interpret" the closure at run time, *Q* actually calls it. One advantage of this mechanism is its interoperability across programming languages: C functions passed as parameters are simply code addresses. In fact, if *S* is declared at the outermost level of lexical nesting, then gcc can pass an ordinary code address even when compiling Ada source; in this case no trampoline is required. ■

x86-64 As noted in Section C-5.4.5, the x86-64 has 16 integer registers instead of only 8. AMD, which developed the ISA for the wider architecture, suggests a calling sequence that makes more use of registers (and less of the stack), in a manner reminiscent of Arm (Example C-9.69) and other RISC machines. The GNU compiler generally conforms to AMD's suggestions.

Figure C-9.10 shows a stack frame for the x86-64. The first six integer arguments are passed in registers rdi, rsi, rdx, rcx, r8, and r9, in that order. The static link, when needed, is passed in r10 (not rcx). Registers rbx and r12–r15 are callee saves; rax, r10, r11, and the argument registers are caller-saves. Integer function values are returned in rax and (if needed) rdx. The first eight floating-point arguments are passed in XMM/SSE registers xmm0–xmm7 (the legacy x87 registers are for the most part ignored). Additional floating-point arguments are

passed on the stack. Floating-point function values are returned in `xmm0` and (if needed) `xmm1`. The stack is always 16-byte aligned at the time of a call.

EXAMPLE 9.73
The x86-64 red zone

Perhaps the most interesting difference between the x86-32 and x86-64 conventions is AMD's specification of a "red zone" beyond the `sp`. Where the last used word on the stack is guaranteed on x86-32 to be at an address no lower than the `sp`, on x86-64 it can be up to 128 bytes beyond this point—in effect, the `sp` protects not only the data at higher addresses (below it in the stack), but up to 128 bytes of additional data as well. Signal handlers and other system software are required to respect this convention. As a result, leaf routines that need a stack frame smaller than 128 bytes need not update the `sp`. For frequent calls to very small routines, the two-instruction savings in per-call bookkeeping can be significant. ■

CHECK YOUR UNDERSTANDING

51. For each of our three case studies, explain which aspects of the calling sequence and stack layout are dictated by the hardware, and which are a matter of software convention.

52. Why don't LLVM and `gcc` restore caller-saves registers immediately after a call?

53. What is a subroutine closure *trampoline*? How does it differ from the usual implementation of a closure described in Section 3.6.1? What are the comparative advantages of the two alternatives?

DESIGN & IMPLEMENTATION

9.11 Executing code in the stack

A disadvantage of trampoline-based closures is the need to execute code in the stack. Many machines and operating systems disallow such execution, for at least two important reasons. First, as noted in Section C-5.1, modern microprocessors typically have separate instruction and data caches, for fast concurrent access. Allowing a process to write and execute the same region of memory means that these caches must be kept mutually consistent (coherent), a task that introduces significant hardware complexity (on some machines it requires execution of a special hardware instruction). Second, many computer security breaches involve a *code injection* attack, in which an intruder exploits software vulnerabilities (e.g., the lack of array bounds checking in C) to write instructions into the stack, and to overwrite the saved return address so that execution will jump into that code when the current subroutine returns. Such an attack is possible only on machines in which writable data are also executable. When compiling code for use on modern systems, `gcc` embeds a call to a library routine that reverses the system default and re-enables stack execution prior to using a trampoline.

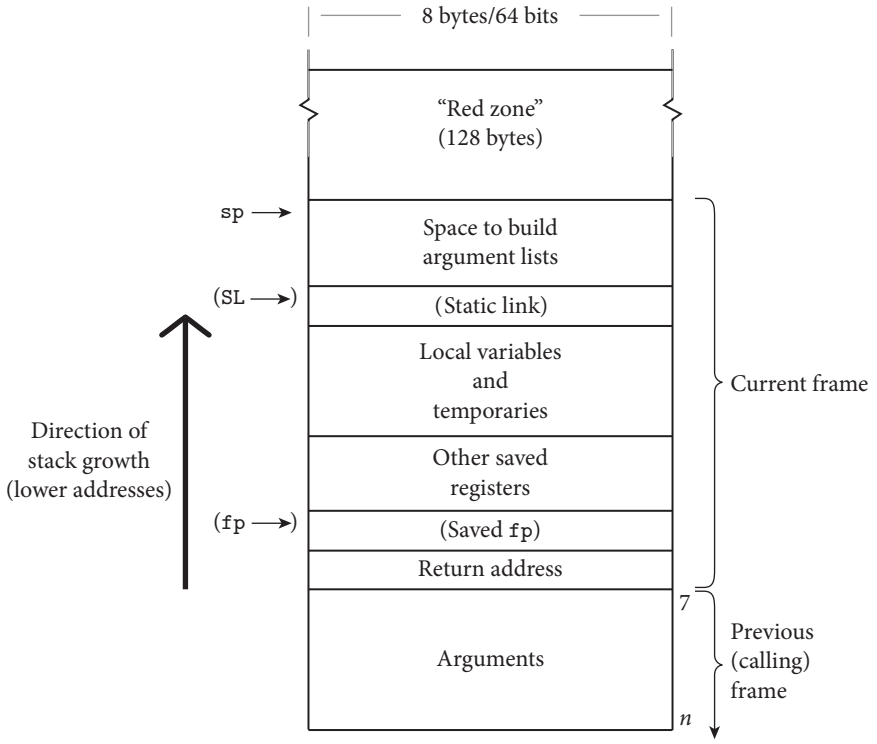

Figure 9.10 **Layout of the subroutine call stack for the GNU Compiler Collection (gcc) on 64-bit x86.** Conventions differ from those of Figure C-9.9 in three principal ways: (1) most data are 64 bits wide; (2) the first 6 integer arguments are passed in registers rather than on the stack; (3) leaf routines are permitted to use up to 128 bytes of space beyond the top of the stack, without updating the sp.

54. Explain the circumstances under which a subroutine needs a frame pointer (i.e., under which access via displacement addressing from the stack pointer will not suffice).

55. Under what circumstances must an argument that was passed in a register also be saved into the stack?

56. What is the purpose of the "red zone" on x86-64?

9 Subroutines and Control Abstraction

9.2.3 Register Windows

EXAMPLE 9.74
Register windows on the SPARC

As an alternative to saving and restoring registers on subroutine calls and returns, the original Berkeley RISC machines [PD80, Pat85] incorporated a hardware mechanism known as *register windows*. The basic idea is to provide a very large set of physical registers, most of which are organized as a collection of overlapping windows (Figure C-9.11). A few register names (`r0–r7` in the figure) always refer to the same locations, but the rest (`r8–r31` in the figure) are interpreted relative to the currently active window. On a subroutine call, the hardware moves to a different window. To facilitate the passing of parameters, the old and new windows overlap: the top few registers in the caller's window (`r24–r31` in the figure) are the same as the bottom few registers in the callee's window (`r8–r15` in the figure). On a machine with register windows, the compiler places values of use only within the current subroutine in the middle part of the window. It copies values to the upper part of the window to pass them to a called routine, within which they are read from the lower part of the window.

Since the number of physical windows is fixed, a long chain of subroutine calls can cause the hardware to run off the end of the register set, resulting in a "window overflow" interrupt that drops the processor into the operating system. The interrupt handler then treats the set of available windows as a circular buffer. It copies the contents of one or more windows to memory and then resumes execution. Later, a "window underflow" interrupt will occur when control attempts to return into a window whose contents have been written to memory. Again the operating system recovers, by restoring the saved registers and resuming execution. In practice, eight windows appear to suffice to make overflow and underflow relatively rare in typical programs.

Register windows have been used in several RISC processors, but only one of these, the SPARC, is commercially significant today. The Intel IA-64 (Itanium), introduced shortly after the turn of the century, also uses register windows, though it is not a RISC machine. The advantage of windows, of course, is that they reduce

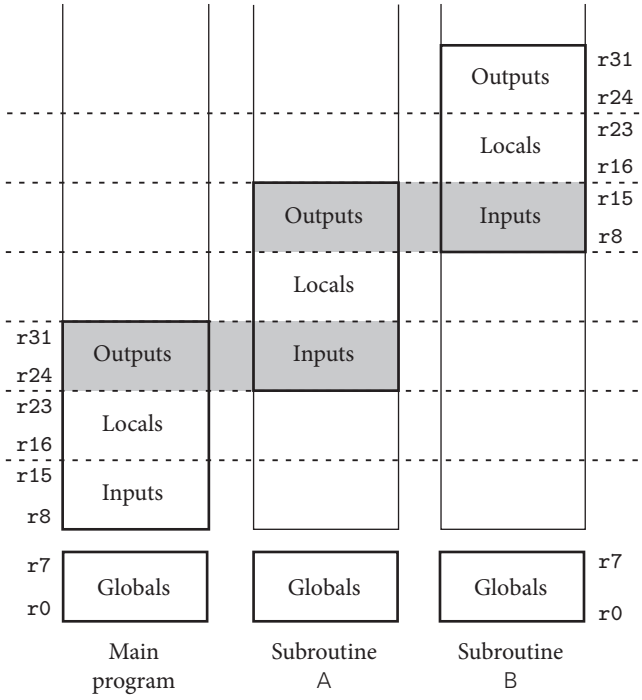

Figure 9.11 **Register windows.** When the main program calls subroutine A, and again when A calls B, register names `r0`–`r7` continue to refer to the same locations, but register names `r8`–`r31` are changed to refer to a new, overlapping window. High-numbered registers in the caller share locations with low-numbered registers in the callee.

the number of loads and stores required for the typical subroutine call. At the same time, register windows significantly increase the amount of state associated with the currently running program. When the operating system decides to give the processor to a different application for a while (something that most systems do many times per second), it must save all this state to memory, or arrange for the processor to trap back into the OS if the new process attempts to access an unsaved window. Worse, while register windows nicely capture the referencing environment of a single thread of control, they do not work well for languages that need more than one referencing environment (execution context). Several language features, including continuations (Section 6.2.2), iterators (Section 6.5.3), and coroutines (Section 9.5), are difficult to implement on a machine with register windows, because they require that we save and restore not only the visible registers, but those in other windows as well, when switching between contexts. It is unclear whether the reduction in subroutine call overhead outweighs the extra cost of context switches for typical application workloads, particularly given that loads and stores for parameters are almost always cache hits.

CHECK YOUR UNDERSTANDING

57. What are *register windows*? What purpose do they serve?
58. Which commercial instruction sets include register windows?
59. Explain the concepts of register window *overflow* and *underflow*.
60. Why are register windows a potential problem for multithreaded programs?

9 Subroutines and Control Abstraction

9.3.2 Call by Name

Call by name implements the normal-order argument evaluation described in Section 6.6.2. A call-by-name parameter is reevaluated in the caller's referencing environment every time it is used. The effect is as if the called routine had been textually expanded at the point of call, with the actual parameter (which may be a complicated expression) replacing every occurrence of the formal parameter. To avoid the usual problems with macro parameters, the "expansion" is defined to include parentheses around the replaced parameter wherever syntactically valid, and to make "suitable systematic changes" to the names of any formal parameters or local identifiers that share the same name, so that their meanings never conflict [NBB+63, p. 12]. Call by name was the default in Algol 60; call by value was available as an alternative. In Simula call by value was the default; call by name was the alternative.

To implement call by name, Algol 60 implementations passed a hidden subroutine that evaluated the actual parameter in the caller's referencing environment. Such a hidden routine is usually called a *thunk*.[1] In most cases thunks are trivial. If an actual parameter is a variable name, for example, the thunk simply reads the variable from memory. In some cases, however, a thunk can be elaborate. Perhaps the most famous occurs in what is known as *Jensen's device*, named after Jørn Jensen [Rut67]. The idea is to pass to a subroutine both a built-up expression and one or more of the variables used in the expression. Then by changing the values of the individual variable(s), the called routine can deliberately and systematically change the value of the built-up expression. This device can be used, for example, to write a summation routine:

EXAMPLE 9.75
Jensen's device

1 In general, a thunk is a procedure of zero arguments used to delay evaluation of an expression. Other examples of thunks can be seen in the `delay` mechanism of Example 6.88 and the `promise` constructor of Exercise 11.18.

```
real procedure sum(expr, i, low, high);
    value low, high;
        comment low and high are passed by value;
        comment expr and i are passed by name;
    real expr;
    integer i, low, high;
begin
    real rtn;
    rtn := 0;
    for i := low step 1 until high do
        rtn := rtn + expr;
        comment the value of expr depends on the value of i;
    sum := rtn
end sum
```

Now to evaluate the sum

$$y = \sum_{1 \le x \le 10} 3x^2 - 5x + 2$$

we can simply say

```
y := sum(3*x*x - 5*x + 2, x, 1, 10);
```

Label Parameters

Both Algol 60 and Algol 68 allowed a label to be passed as a parameter. If a called routine performed a `goto` to such a label, control would usually need to escape the local context, unwinding the subroutine call stack as it did so. Details of the unwinding operation would depend on the location of the label. For each intervening scope, the `goto` would have to restore saved registers, deallocate the

DESIGN & IMPLEMENTATION

9.12 Call by name

In practice, most uses of call by name in Algol 60 and Simula programs served to allow a subroutine to change the value of an actual parameter; neither language offered call by reference. Unfortunately, call by name is significantly more expensive than call by reference: it requires the invocation of a thunk (as opposed to a simple indirection) on every use of a formal parameter. Call by name is also prone to subtle program bugs when a change to a variable in a surrounding scope unintentionally alters the value of a formal parameter. (Call by reference suffers from a milder form of this problem, as discussed in Example 3.20.) Such deliberate subtleties as Jensen's device are comparatively rare, and can be imitated in other languages through the use of formal subroutines. Call by name was dropped in Algol 68, in favor of call by reference.

stack frame, and perform any other operations normally handled by epilogue code. To implement label parameters, Algol implementations typically passed a thunk that performed the appropriate operations for the given label. Note that the target label would generally need to lie in some surrounding scope, where it was visible to the caller under static scoping rules.

Label parameters were usually used to handle *exceptional conditions*—conditions that prevent a subroutine from performing its usual operation, and that cannot be handled in the local context. Instead of returning, an Algol routine that encountered a problem (e.g., invalid input) could perform a `goto` to a label parameter, on the assumption that the label referred to code that would perform some remedial operation, or print an appropriate error message. In more recent languages, label parameters have been replaced by more structured exception handling mechanisms, as discussed in Section 9.4.

✓ CHECK YOUR UNDERSTANDING

61. What is *call by name*? What language first provided it? Why isn't it used by the language's descendants?
62. What is *call by need*? How does it differ from call by name? What modern languages use it?
63. How does a subroutine with call-by-name parameters differ from a macro?
64. What is a *thunk*? What is it used for?
65. What is *Jensen's device*?

DESIGN & IMPLEMENTATION

9.13 Call by need

Functional languages like Miranda and Haskell typically pass parameters using a *memoizing* implementation of normal-order evaluation, as described in Section 6.6.2. This *lazy* implementation is sometimes called *call by need*. Memoization calculates and records the value of a parameter the first time it is needed, and uses the recorded value thereafter. In the absence of side effects, call by need is indistinguishable from call by name. It avoids the expense of repeated evaluation, but precludes the use of techniques like Jensen's device in languages that *do* have side effects. Among imperative languages, call by need appears in the scripting language R, where it serves to avoid the expense of evaluating (even once) any complex arguments that are not actually needed.

9 Subroutines and Control Abstraction

9.5.3 Implementation of Iterators

EXAMPLE 9.76
Coroutine-based iterator invocation

Consider the following `for` loop from Example 6.66:

```
for i in range(first, last, step):
    ...
```

Using coroutines, a compiler might translate this as

```
iter := new from_to_by(first, last, step, i, done, current_coroutine)
while not done do
    ...
    transfer(iter)
destroy(iter)
```

After the loop completes, the implementation can reclaim the space consumed by iter.

EXAMPLE 9.77
Coroutine-based iterator implementation

The definition of from_to_by itself is quite straightforward:

```
coroutine from_to_by(from_val, to_val, by_amt : int;
                     ref i : int; ref done : bool; caller : coroutine)
    i := from_val
    if by_amt > 0 then
        done := from_val ≥ to_val
        detach
        loop
            i +:= by_amt
            done := i ≥ to_val
            transfer(caller)      -- yield i
```

```
else
    done := from_val ≤ to_val
    detach
    loop
        i +:= by_amt
        done := i ≤ to_val
        transfer(caller)     -- yield i
```

Parameters i and done are passed by reference so that the iterator can modify them in the caller's context. The caller's identity is passed as a final argument so that the iterator can tell which coroutine to resume when it has computed the next loop index. Because the caller is named explicitly, it is easy for iterators to nest, as in Figure 6.5.

Single-Stack Implementation

While coroutines *suffice* for the implementation of iterators, they are not *necessary*. A simpler, single-stack implementation is also possible. Because a given iterator (e.g., an instance of `from_to_by`) is always resumed at the same place in the code (between iterations of a given `for` loop), we can be sure that the subroutine call stack will always contain the same frames whenever the iterator runs. Moreover, since `yield` statements can appear only in the main body of the iterator (never in nested routines), we can be sure that the stack will always contain the same frames whenever the iterator transfers back to its caller. These two facts imply that we can place the frame of the iterator directly on top of the frame of its caller in a single central stack.

When an iterator is created, its frame is pushed on the stack. When it yields a value, control returns to the `for` loop, but the iterator's frame is left on the stack. If the body of the loop makes any subroutine calls, the frames for those calls will be allocated beyond the frame of the iterator. Since control must return to the loop before the iterator resumes, we know that such frames will be gone again before the iterator has a chance to see them: if it needs to call subroutines itself, the stack above it will be clear. Likewise, if the iterator calls any subroutines, they will return (popping their frames from the stack) before the `for` loop runs again. Nested iterators present no special problems (see Exercise C-9.37).

Data Structure Implementation

EXAMPLE 9.78
Iterator usage in C#

Compilers for C# 2.0 employ yet another implementation of iterators. Like Java, C# 1.1 provided iterator objects. Each such object implements the `IEnumerator` interface, which provides `MoveNext` and `Current` methods. Typically an iterator is obtained by calling the `GetEnumerator` method of an object (a container) that implements the `IEnumerable` interface:

```
for (IEnumerator i = myTree.GetEnumerator(); i.MoveNext();) {
    object o = i.Current;
    Console.WriteLine(o.ToString());
}
```

EXAMPLE 9.79
Implementation of C# iterators

C# 2.0 provides true iterators as an extension of iterator objects. The programmer simply declares a method that contains one or more `yield return` statements, and whose return type is `IEnumerator` or `IEnumerable`. Here is an example of the latter:

```
static IEnumerable FromToBy (int fromVal, int toVal, int byAmt)
{
    if (byAmt >= 0) {
        for (int i = fromVal; i <= toVal; i += byAmt) {
            yield return i;
        }
    } else {
        for (int i = fromVal; i >= toVal; i += byAmt) {
            yield return i;
        }
    }
}
```

The compiler automatically transforms this code into a hidden class with a `GetEnumerator` method, along the lines of Figure C-9.12. Within this code, an explicit state variable keeps track of the "program counter" of the last `yield` statement. In addition, local variable `i` of the true iterator becomes a data member of the `FromToByImpl` class, leaving the iterator with no need for a stack frame across iterations of the loop. In a quite literal sense, the compiler transforms each true iterator into an iterator object.

Recursive iterators present no particular difficulties: a nested iterator is allocated on demand when the outer iterator enters a `foreach` loop, and is referred to by a reference in that outer iterator. The details are deferred to Exercise C-9.38. Because iterator objects are allocated from the heap, the C# implementation of true iterators may be somewhat slower than the stack-based implementation of the previous subsection.

CHECK YOUR UNDERSTANDING

66. Describe the "obvious" implementation of iterators using coroutines.
67. Explain how the state of multiple active iterators can be maintained in a single stack.
68. Describe the transformation used by C# compilers to turn a true iterator into an iterator object.

```
static IEnumerable FromToBy(int fromVal, int toVal, int byAmt) {
    return new FromToByImpl(fromVal, toVal, byAmt);
}
class FromToByImpl : IEnumerator, IEnumerable {
    enum State {starting, goingUp, goingDown, done}
    int i, tv, ba;
    State s;

    public FromToByImpl(int fromVal, int toVal, int byAmt) {
        i = fromVal; tv = toVal; ba = byAmt; s = State.starting;
    }
    public IEnumerator GetEnumerator() {
        return this;
    }
    public object Current {
        get { return i; }
    }
    public bool MoveNext() {
        switch (s) {
            case State.starting :
                if (ba >= 0) {
                    if (i <= tv) { s = State.goingUp; return true; }
                    else { s = State.done; return false; }
                } else {
                    if (i >= tv) { s = State.goingDown; return true; }
                    else { s = State.done; return false; }
                }
            case State.goingUp :
                i += ba;
                if (i <= tv) return true;
                else { s = State.done; return false; }
            case State.goingDown :
                i += ba;
                if (i >= tv) return true;
                else { s = State.done; return false; }
            default: // for completeness
            case State.done : return false;
        }
    }
    public void Reset() {
        s = State.starting;
    }
}
```

Figure 9.12 **Iterator object equivalent of a true iterator in C#.** This handwritten code corresponds to Example C-9.79. It represents, at the source level, what the compiler creates at the level of intermediate code: a state machine that tracks the program counter of the original iterator, with a starting state, an ending state, and one state for each `yield return` statement. The arms of the `switch` statement capture the code paths in the original iterator that move from one state to the next.

9 Subroutines and Control Abstraction

9.5.4 Discrete Event Simulation

EXAMPLE 9.80
Sequential simulation of a complex physical system

Suppose that we wish to experiment with the flow of traffic in a city. A computerized traffic model, if it captures the real world with sufficient accuracy, will allow us to predict the effects of construction projects, accidents, increased traffic due to new development, or changes to the layout of streets. It is difficult (though certainly not impossible) to write such a simulation in a conventional sequential language. We would probably represent each interesting object (automobile, intersection, street segment, etc.) with a data structure. Our main program would then look something like this:

```
while current_time < end_of_simulation
    calculate next time t at which an interesting interaction will occur
    current_time := t
    update state of objects to reflect the interaction
    record desired statistics
print collected statistics
```

The problem with this approach lies in determining which objects will interact next, and in remembering their state from one interaction to the next. It is in some sense unnatural to represent active objects such as cars with passive data structures, and to make time the active entity in the program. An arguably more attractive approach is to represent each active object with a coroutine, and to let each object keep track of its own state.

EXAMPLE 9.81
Initialization of a coroutine-based traffic simulation

If each active object can tell when it will next do something interesting, then we can determine which objects will interact next by keeping the currently inactive coroutines in a priority queue, ordered by the time of their next event. We might begin a one-day traffic simulation by creating a coroutine for each trip to be taken by a car that day, and inserting each coroutine into the priority queue with a "wakeup" time indicating when the trip is to begin:

```
coroutine trip(. . . )
. . .
for each trip t
    p := new trip(. . . )
    schedule(p, t.start_time)
```

EXAMPLE 9.82

Traversing a street segment in the traffic simulation

Let us assume that we think of street segments as passive, and represent them with data structures. At any given moment, we can model a segment by the number of cars that it is carrying in each direction. This number in turn will affect the speed at which the cars can safely travel. Whenever it awakens, the coroutine representing a trip examines the next street segment over which it needs to travel. Based on the current load on that segment, it calculates how much time it will take to traverse it, and schedules itself to awaken again at an appropriate point in the future:

```
coroutine trip(origin, destination : location)
    plan a route from origin to destination
    detach
    for each segment of the route
        calculate time i to reach the end of the segment
        schedule(current_coroutine, current_time + i)
```

EXAMPLE 9.83

Scheduling a coroutine for future execution

The `schedule` operation is easily built on top of transfer:

```
schedule(p : coroutine; t : time)
    -- p may be self or other
    insert (p, t) in priority queue
    if p = current_coroutine    -- self
        extract earliest pair (q, s) from priority queue
        current_time := s
        transfer(q)
```

EXAMPLE 9.84

Queueing cars at a traffic light

In some cases, it may be difficult to determine when to reschedule a given object. Suppose, for example, that we wish to more accurately model the effects of traffic signals at intersections. We might represent each traffic signal with a data structure that records the waiting cars in each direction, and a coroutine that lets cars through as the signal changes color:

```
record controlled_intersection =
    EW_cars, NS_cars : queue of trip
    const per_car_lag_time : time
        -- how long it takes a car to start after its predecessor does
    coroutine signal(EW_duration, NS_duration : time)
        detach
        loop
            change_time := current_time + EW_duration
            while current_time < change_time
                if EW_cars not empty
                    schedule(dequeue(EW_cars), current_time)
                schedule(current_coroutine, current_time + per_car_lag_time)
```

```
        change_time := current_time + NS_duration
        while current_time < change_time
            if NS_cars not empty
                schedule(NS_cars.dequeue(), current_time)
            schedule(current_coroutine, current_time + per_car_lag_time)
```

EXAMPLE 9.85
Waiting at a light

When it reaches the end of a street segment that is controlled by a traffic signal, a trip need not calculate how long it will take to get through the intersection. Rather, it enters itself into the appropriate queue of waiting cars and "goes to sleep," knowing that the `signal` coroutine will awaken it at some point in the future:

```
coroutine trip(origin, destination : location)
    plan a route from origin to destination
    detach
    for each segment of the route
        calculate time i to reach the end of the segment
        schedule(current_coroutine, current_time + i)
        if end of segment has a traffic light
            identify appropriate queue Q
            Q.enqueue(current_coroutine)
            sleep()
```

EXAMPLE 9.86
Sleeping in anticipation of future execution

Like `schedule`, `sleep` is easily built on top of transfer:

```
sleep()
    extract earliest pair (q, s) from priority queue
    current_time := s
    transfer(q)
```

The `schedule` operation, in fact, is simply

```
schedule(p : coroutine; t : time)
    insert (p, t) in priority queue
    if p = current_coroutine
        sleep()
```

Obviously this traffic simulation is too simplistic to capture the behavior of cars in a real city, but it illustrates the basic concepts of discrete event simulation. More sophisticated simulations are used in a wide range of application domains, including all branches of engineering, computational biology, physics and cosmology, and even computer design. Multiprocessor simulations (see reference [VF94], for example) are typically divided into a "front end" that simulates the processors and a "back end" that simulates the memory subsystem. Each coroutine in the front end consists of a machine-language interpreter that captures the behavior of one of the system's processing cores. Each coroutine in the back end represents a load or a store instruction. Every time a processor performs a load or store, the front end creates a new coroutine in the back end. Data structures in the

back end represent various hardware resources, including caches, buses, network links, message routers, and memory modules. The coroutine for a given load or store checks to see if its location is in the local cache. If not, it must traverse the interconnection network between the processor and memory, competing with other coroutines for access to hardware resources, much as cars in our simple example compete for access to street segments and intersections. The behavior of the back-end system in turn affects the front end, since a processor must wait for a load to complete before it can use the data, and since the rate at which stores can be injected into the back end is limited by the rate at which they propagate to memory.

✓ CHECK YOUR UNDERSTANDING

69. Summarize the computational model of discrete event simulation. Explain the significance of the time-based priority queue.

70. When building a discrete event simulation, how does one decide which things to model with coroutines, and which to model with data structures?

71. Are all inactive coroutines guaranteed to be in the priority queue? Explain.

9 Subroutines and Control Abstraction

9.9 Exercises

9.29 Suppose you wish to minimize the size of closures in a language implementation that uses a display to access nonlocal objects. Assuming a language like Pascal or Ada, in which subroutines have limited extent, explain how an appropriate display for a formal subroutine can be calculated when that routine is finally called, starting with only (1) the value of the frame pointer, saved in the closure at the time that the closure was created, (2) the subroutine return addresses found in the stack at the time the formal subroutine is finally called, and (3) static tables created by the compiler. How costly is your scheme?

9.30 Elaborate on the reasons why even parameters passed in registers may sometimes need to have locations in the stack. Consider all the cases in which it may not suffice to keep a parameter in a register throughout its lifetime.

9.31 Most versions of the C library include a function, `alloca`, that dynamically allocates space within the current stack frame.[2] It has two advantages over the usual `malloc`, which allocates space in the heap: it's usually very fast, and the space it allocates is reclaimed automatically when the current subroutine returns. Assuming the programmer *wants* deallocation to happen then, it's convenient to be able to skip the explicit `free` operations. How might you implement `alloca` in conjunction with the calling conventions of our various case studies?

9.32 Explain how to extend the conventions of Figure C-9.9 and Section C-9.2.2 to accommodate arrays whose bounds are not known until elaboration time (as discussed in Section 8.2.2). What ramifications does this have for the use of separate stack and frame pointers?

---

2 Unfortunately, `alloca` is not POSIX compliant, and implementations vary greatly in their semantics and even in details of the interface. Portable programs are wise to avoid this routine.

9.33 In all three of our case studies, stack-based arguments were placed into the argument build area in "reverse" order, with the lowest-numbered argument at the top. Explain why this is important. (Hint: Consider subroutines with variable numbers of arguments, as discussed in Section 9.3.3.)

9.34 How would you implement nested subroutines as parameters on a machine that doesn't let you execute code in the stack? Can you pass a simple code address, or do you require that closures be interpreted at run time?

9.35 If you have read the rest of Chapter 9, you may have noticed that the term "trampoline" is also used in conjunction with the implementation of signal handlers (Section 9.6.1). What is the connection (if any) between these uses of the term?

9.36 Explain how you might implement `setjmp` and `longjmp` on a SPARC.

9.37 Following the code in Figure 6.5, and assuming a single-stack implementation of iterators, trace the contents of the stack during the execution of a `for` loop that iterates over all nodes of a complete, 3-level (6-node) binary tree.

9.38 Build a preorder iterator for binary trees in Java, C#, or Python. Do not use a true iterator or an explicit stack of tree nodes. Rather, create nested iterator objects on demand, linking them together as a C# compiler might if it were building the iterator object equivalent of a true preorder iterator.

9.39 One source of inaccuracy in the traffic simulation of Section C-9.5.4 has to do with the timing at traffic signals. If a signal is currently green in the `EW` direction, but the queue of waiting cars is empty, the `signal` coroutine will go to sleep until `current_time + EW_duration`. If a car arrives before the coroutine wakes up again, it will needlessly wait. Discuss how you might remedy this problem.

9 Subroutines and Control Abstraction

9.10 Explorations

9.53 Read the Arm calling sequence standard for 64-bit (v8) code. Compare and contrast to the conventions of Section C-9.2.2. Pay particular attention to the lists of caller- and callee-saves registers, and to the registers used to pass arguments. Speculate as to reasons for the differences.

9.54 Research the full range of hardware support for subroutines on the x86, including all variants of `call`. Note that the `leave` instruction is sometimes generated by modern compilers, but others, including `enter`, `pushad`, `popad`, `pushfd`, and `popfd`, usually are not. In addition, the optional argument of `ret` is almost never used, and `push` and `pop` are used sparingly. Discuss the technological trends that have made this machinery obsolete.

9.55 As an example of hard-core CISC design, research the subroutine calling conventions of the Digital VAX. Be sure to describe the behavior of the `calls` instruction in detail.

9.56 Study the implementation of a user-level thread management package written for the SPARC. How does it manage register windows?

9.57 Learn how parameter passing is implemented in the Glasgow Haskell compiler. How expensive is its call-by-need–based lazy evaluation?

9.58 Learn about the Time Warp system for discrete event simulation, developed by David Jefferson and colleagues [JBW+87]. Discuss its relationship to both the classic discrete event simulation of Section C-9.5.4 and the speculative parallelism of mechanisms like transactional memory (to be discussed in Section 13.4.5).