Chapter 8 Subroutines and Control Abstraction

June 25, 2015

Stack layout

- Common to have subroutine activation record allocated on a stack
- Typical fare for a frame: arguments, return value, saved registers, local variables, temporaries...
- Maybe the best part of using a stack: it's easy to deallocate (also, decent hardware support usually)

Stack layout

- Processors almost always support at least one stack; generally there is a register devoted to that stack called the "stack pointer" (usually abbreviated something like SP.) Multiple stack computer designs aren't as common, but allow for clean separation of return addresses, expression evaluation, argument lists, and so forth.

Stack layout

- While a stack pointer points to the active end of the stack, a frame pointer generally is used to point somewhere in the stack (usually at the "current" activation record). In the x86/x86_64 family, this register is usually BP/EBP/RBP.

Static and dynamic links

Static and dynamic links

- Dynamic links allow one to walk back the frame pointer linearly down the call stack.
- Static links allow one to walk back the frame pointer from a lexical viewpoint.

Calling sequence

- The maintenance of stacks (or, indeed, anything else, particularly registers) prior to and at the end of a subroutine invocation is called the "calling sequence". While in assembly language programming this is often ad hoc, in higher languages, it is generally quite rigid. In most languages today, a single united stack is the center of the action.

A prologue

- The housekeeping which is done before entry into a subroutine is called the prologue; typically involve any needed set up of parameters, saving the return address (though usually this taken care by the CALL instruction), modifying the program counter (again, usually the bailiwick of the CALL instruction), moving the stack pointer for space allocation, saving registers, moving the frame pointer, and perhaps some initialization code.

An epilogue

- The housekeeping which must be done after a routine has finished is called the epilogue; return values have to adjudicated, stacks allocated must be deallocated, registers restored, and of course, moving the program counter (that is usually done by a RETURN instruction).

A matter of decisions

- As the text so correctly points out, figuring out who does what with the registers is critical to any calling sequence; indeed, with assembly language programming, it's usually *the only significant issue* in the calling sequence. Architectures offering large amounts of registers mean that this bounty can simply be split among the caller and callee.

Architecture (the actuality of the machine) plays a part with most implementations

- Ignoring the hardware is generally the wrong answer.
- Compilers on CISC machines tend to pass arguments on the stack; on RISC machines, they tend to use registers
- Compilers on CISC machines tend to dedicate a register for the frame; less likely to see this on RISC architecture
- As you might surmise, compilers on CISC architectures attempt usually to make use of that complexity.

In-line expansion

- Instead of actually calling a very simple non-generally-recursive routine with all of the calling sequences costs inherent, it often makes sense to simply do the code in place. Some languages offer the ability to hint to a compiler that such in-lining makes sense, such as some members of the C family and even Ada.

Parameter passing

- Generally we distinguish the formal parameters (those specified in the subroutine's definition) from the actual parameters passed (though such distinctions certainly don't exist in all languages, and don't exist in assembly language programming.)
- Call-by-value versus call-by-reference: in languages that have a value model, you have a bit of a dilemma when passing parameters: should you just pass the value, or should you pass a pointer to the value? If you pass the latter, it certainly makes it simple for the called subroutine to modify the underlying data. However, you quickly get quagmires associated with aliasing – though you could remedy *that* by then not allowing *any* modifications of the state of any variable.

Parameter passing

- Closures as parameters: languages that allowing nesting and allow subroutines to be passed as parameters need closures to pass both the subroutine and its referencing environment.

Position parameters versus named parameters

- Instead of merely matching actual and formal parameters by their position, languages like Ada allow one to name parameters – which is quite useful when you have the ability to give parameters default values; you just name values for the parameters that either do not have a default value, or ones that you wish to use a non-default value.

Variable numbers of arguments

- Recall the code for our RecursiveDescent parser: parser.c. At the end, we have an emit() function that allows a variable number of argments.
- This turns out to be fairly useful, though languages with a native list type already have a powerful mechanism for expressing a similar idea.

Returns

- In functional languages, generally the value of the body of the function specifies what is returned.
- In imperative languages, it's more common to have a explicit "return()"; some languages allow the function to specify its return value by either allowing an assignment to the function's name, or having some syntax to specify a special name to refer to the value of a function.

Generics

- Very useful for creating containers, generics allow a programmer to specify a set of routines that can be defined over arbitrary types, and are quite analogous to macros in assembly language. Indeed, the most common implementation for generics is literally macro expansion, just as in assembly language.
- Your text distinguishes the two by the level of the rewriter: pure macro expansion is done outside the language as text-rewriting (for example, m4 could be used to do this), but generic expansion is done by the compilation environment, giving it an ability to make syntactic and semantic distinctions that m4 could not.

Exception handling and unwinding the stack

- Exceptions are unexpected/unusual situations that are not easily handled locally. Run-time errors, particularly those related to I/O, are often awkward at the point of contact, and often are more cleanly handled elsewhere. If the elsewhere is up the stack in a parent activation record, then the stack needs to be *unwound* to that point.
- Unwinding means not only popping off all those activation frames, but also restoring the state at the point of recovery in the propagation process.

Note on page 424

- "Exception-handling mechanisms are among the most complex aspects of modern language design, from both a semantic and a pragmatic point of view. Programmers have used subroutines since before there were compilers (they appear, among other places, in the 19th-century notes of Countess Ada Augusta Byron). Structured exceptions, by contrast, were not invented until the 1970s and did not become commonplace until the 1980s."

setjmp and longjmp

- Between the ad hoc methods often employed in languages like Pascal, which do not have explicit exception-handling, and structured exceptions lies the C solution of setjmp() and longjmp(); as the manual page says: "setjmp() and sigsetjmp() make programs hard to understand and maintain. If possible an alternative should be used."

Co-routines

- These are pretty rare; I don't remember ever seeing these actually used anywhere (well, that is outside of assembly language; co-routines are trivial in assembly language), though apparently some languages do like to use these to implement iterators.
- As your text notes, threads are quite similar in many ways, and offer more functionality at a very modest price.