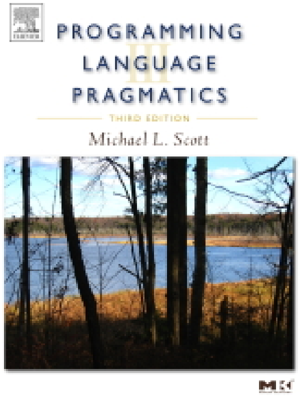
Subroutines and Control Abstraction

UNIVERSIDADE DA CORUÑA

GRAO EN ENXEÑERÍA INFORMÁTICA

DESEÑO DAS LINGUAXES DE PROGRAMACIÓN

Based on Chapter 8 of:
Michael L. Scott, *Programming Language Pragmatics*,
Morgan Kaufmann, 2008.



Review Of Stack Layout

- Allocation strategies
  - Static
    - Code
    - Globals
    - *Own* variables
    - Explicit constants (including strings, sets, other aggregates)
    - Small scalars may be stored in the instructions themselves

Review Of Stack Layout

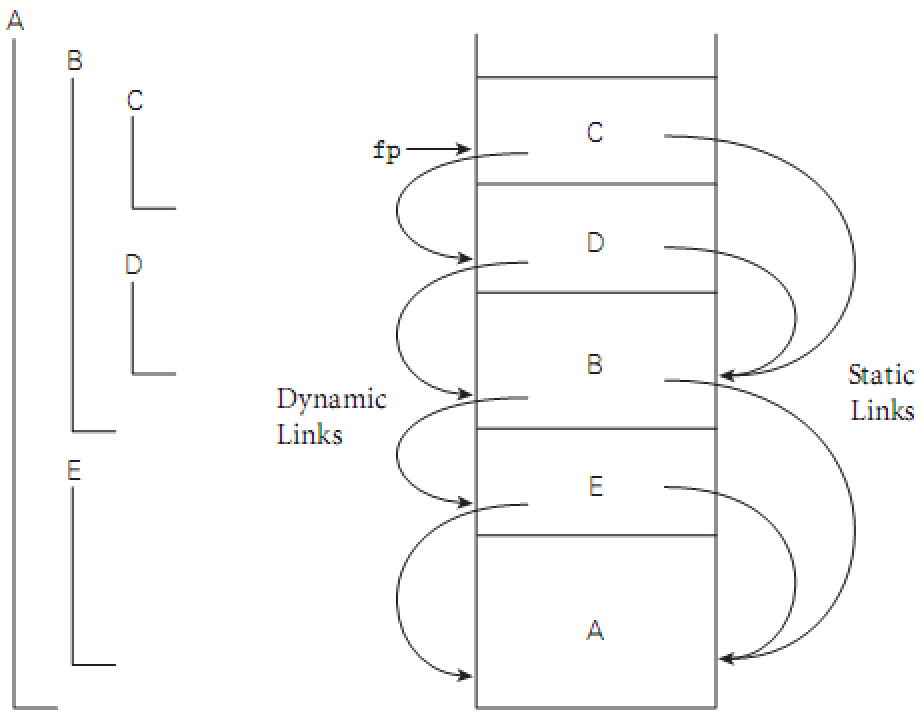


Figure 8.1 Example of subroutine nesting, taken from Figure 3.5. Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right, with the indicated static and dynamic links.

Review Of Stack Layout

- Allocation strategies (2)
  - Stack
    - parameters
    - local variables
    - temporaries
    - bookkeeping information
  - Heap
    - dynamic allocation

Review Of Stack Layout

- Contents of a stack frame
  - bookkeeping
    - return PC (dynamic link)
    - saved registers
    - line number
    - saved display entries
    - static link
  - arguments and returns
  - local variables
  - temporaries

Calling Sequences

- Maintenance of stack is responsibility of *calling sequence* and *subroutine prolog* and *epilog*
  - space is saved by putting as much in the prolog and epilog as possible
  - time *may* be saved by putting stuff in the caller instead, where more information may be known
    - e.g., there may be fewer registers IN USE at the point of call than are used SOMEWHERE in the callee

Calling Sequences

- Common strategy is to divide registers into *caller-saves* and *callee-saves* sets
  - caller uses the "callee-saves" registers first
  - "caller-saves" registers if necessary
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time
  - some storage layouts use a separate arguments pointer
  - the VAX architecture encouraged this

Calling Sequences

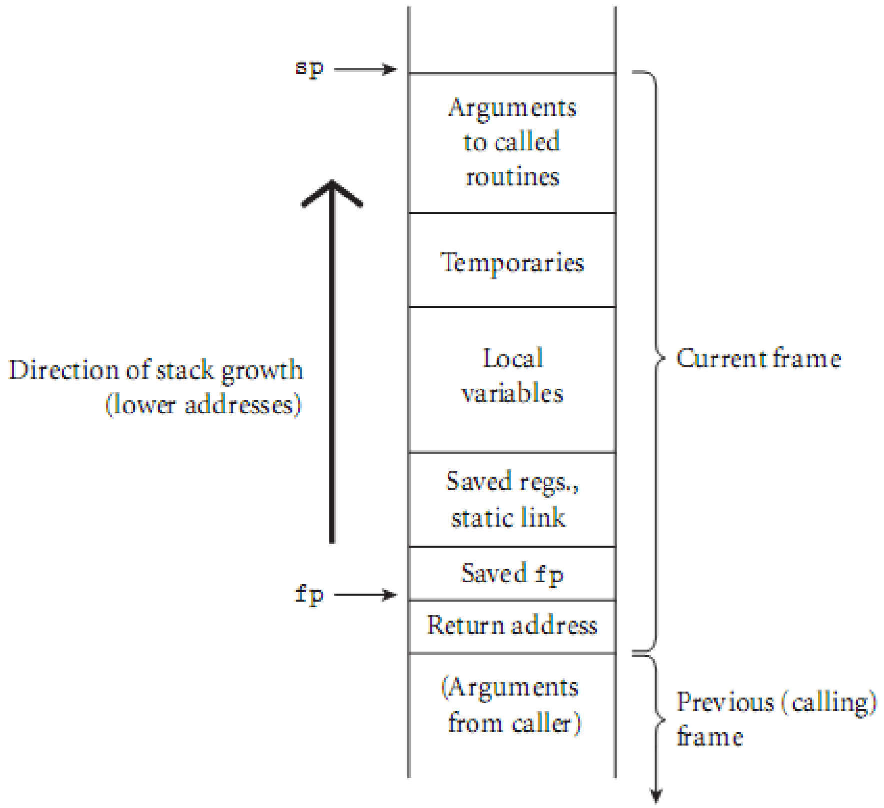


Figure 8.2 A typical stack frame. Though we draw it growing upward on the page, the stack actually grows downward toward lower addresses on most machines. Arguments are accessed at positive offsets from the fp. Local variables and temporaries are accessed at negative offsets from the fp. Arguments to be passed to called routines are assembled at the top of the frame, using positive offsets from the sp.

Calling Sequences (C on MIPS)

- Caller
  - saves into the temporaries and locals area any caller-saves registers whose values will be needed after the call
  - puts up to 4 small arguments into registers $4-$7 (a0-a3)
    - it depends on the types of the parameters and the order in which they appear in the argument list
  - puts the rest of the arguments into the arg build area at the top of the stack frame
  - does jal, which puts return address into register ra and branches
    - note that jal, like all branches, has a delay slot

Calling Sequences (C on MIPS)

- In prolog, Callee
  - subtracts framesize from sp
  - saves callee-saves registers used anywhere inside callee
  - copies sp to fp
- In epilog, Callee
  - puts return value into registers (mem if large)
  - copies fp into sp (see below for rationale)
  - restores saved registers using sp as base
  - adds to sp to deallocate frame
  - does jra

Calling Sequences (C on MIPS)

- After call, Caller
  - moves return value from register to wherever it's needed (if appropriate)
  - restores caller-saves registers lazily over time, as their values are needed
- All arguments have space in the stack, whether passed in registers or not
- The subroutine just begins with some of the arguments already cached in registers, and 'stale' values in memory

Calling Sequences (C on MIPS)

- This is a normal state of affairs; optimizing compilers keep things in registers whenever possible, flushing to memory only when they run out of registers, or when code may attempt to access the data through a pointer or from an inner scope

Calling Sequences (C on MIPS)

- Many parts of the calling sequence, prologue, and/or epilogue can be omitted in common cases
  - particularly LEAF routines (those that don't call other routines)
    - leaving things out saves time
    - simple leaf routines don't use the stack - don't even use memory – and are exceptionally fast

Parameter Passing

- Parameter passing mechanisms have three basic implementations
  - *value*
  - *value/result* (copying)
  - *reference* (aliasing)
  - *closure/name*
- Many languages (e.g., Pascal) provide value and reference directly

Parameter Passing

- C/C++: functions
  - parameters passed by value (C)
  - parameters passed by reference can be simulated with pointers (C)

```
void proc(int* x,int y){*x = *x+y } …
proc(&a,b);
```

  - or directly passed by reference (C++)

```
void proc(int& x, int y) {x = x + y }
proc(a,b);
```

Parameter Passing

- Ada goes for semantics: who can do what
  - *In*: callee reads only
  - *Out*: callee writes and can then read (formal not initialized); actual modified
  - *In out*: callee reads and writes; actual modified
- Ada in/out is always implemented as
  - value/result for scalars, and either
  - value/result or reference for structured objects

Parameter Passing

- In a language with a reference model of variables (Lisp, Clu), pass by reference (*sharing*) is the obvious approach
- It's also the only option in Fortran
  - If you pass a constant, the compiler creates a temporary location to hold it
  - If you modify the temporary, who cares?
- Call-by name is an old Algol technique
  - Think of it as call by textual substitution (procedure with all name parameters works like macro) - what you pass are hidden procedures called THUNKS

Parameter Passing

parameter mode	representative languages	implementation mechanism	permissible operations	change to actual?	alias?
value	C/C++, Pascal, Java/C# (value types)	value	read, write	no	no
`in`, `const`	Ada, C/C++, Modula-3	value or reference	read only	no	maybe
`out`	Ada	value or reference	write only	yes	maybe
value/result	Algol W	value	read, write	yes	no
`var`, `ref`	Fortran, Pascal, C++	reference	read, write	yes	yes
sharing	Lisp/Scheme, ML, Java/C# (reference types)	value or reference	read, write	yes	yes
`in out`	Ada	value or reference	read, write	yes	maybe
name	Algol 60, Simula	closure (thunk)	read, write	yes	yes
need	Haskell, R	closure (thunk) with memoization	read, write*	yes*	yes*

Figure 8.3 Parameter passing modes.

Generic Subroutines and Modules

- Generic modules or classes are particularly valuable for creating *containers*: data abstractions that hold a collection of objects
- Generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right

Exception Handling

- What is an exception?
  - a hardware-detected run-time error or unusual condition detected by software
- Examples
  - arithmetic overflow
  - end-of-file on input
  - wrong type for input data
  - user-defined conditions, not necessarily errors

Exception Handling

- What is an exception handler?
  - code executed when exception occurs
  - may need a different handler for each type of exception
- Why design in exception handling facilities?
  - allow user to explicitly handle errors in a uniform manner
  - allow user to handle errors without having to check these conditions
  - explicitly in the program everywhere they might occur

Coroutines

- Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other explicitly, by name
- Coroutines can be used to implement
  - iterators
  - threads
- Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack

Coroutines

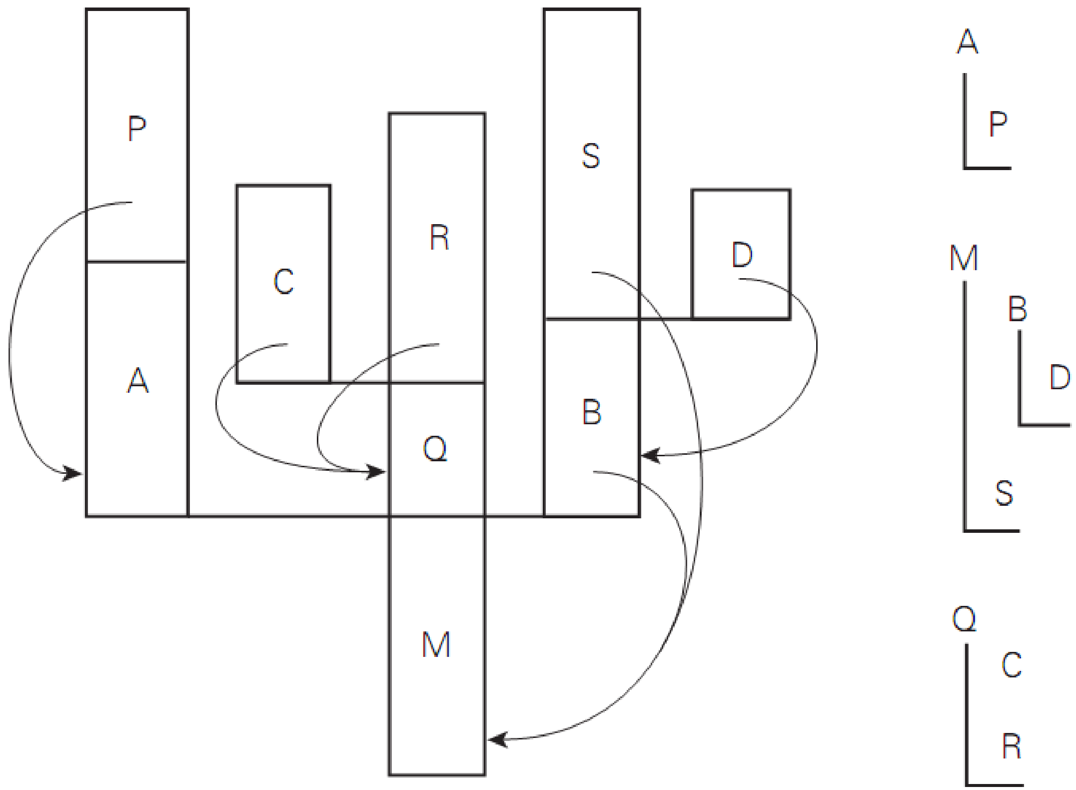


Figure 8.6 A cactus stack. Each branch to the side represents the creation of a coroutine (A, B, C, and D). The static nesting of blocks is shown at right. Static links are shown with arrows. Dynamic links are indicated simply by vertical arrangement: each routine has called the one above it. (Coroutine B, for example, was created by the main program, M. B in turn called subroutine S and created coroutine D.)