City University London

IN2009
Language Processors

Session 6

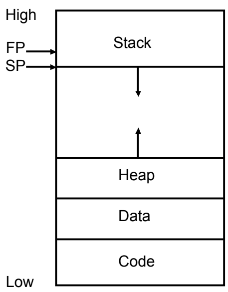
Activation Records

Igor Siveroni

Session Plan

Session 6: Activation Records (Stack Frames)
- Memory model
- Local variables
- Stack frames
  - layout
  - frame pointer and stack pointer
  - parameter passing
  - calling conventions
- Static links
- Frames implementation

Layout in memory



possible format of a code file before it is loaded into memory:

header [magic number,sizes,entry point]
text [the code]
data [global variable space]

symbol table
[variable & method names etc]
string table
[the text of names in symbol table]

'memory model'

Local variables

- Functions/methods may have *local* variables
- Several invocations at same time each with own instantiations of local variables - e.g. recursive calls
- Local variables destroyed on method return
- LIFO behaviour (implemented with stack data structure)

```
int f(int x) {
   int y = x+x;
   if y < 10
     return f(y);
   else
     return y-1;
}
```

- New instantiation of x created & initialised by f's caller each time f called
- Recursive calls - many x's exist simultaneously
- New instantiation of y created each time body of f entered

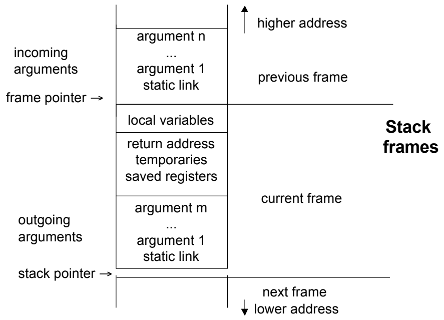
Stack frames

- Frame layout design
  - Takes into account particular features of instruction set architecture and programming language being compiled

Stack frames

- Usually a standard frame layout prescribed by manufacturer
  - not necessarily convenient for compiler writers, but...
  - functions/methods written in one language can call functions/methods written in another, so...
  - gain programming language interoperability
  - can combine modules/classes compiled from different languages in same running program

Stack frames



Stack frame layout

- Set of *incoming arguments* (part of previous frame) passed (stored) by caller code
- Return address (often stored by CALL instruction)
- Local variables (those not in registers)

Stack frame layout

- Area for local variables held in registers but that may need to be saved into frame
- Outgoing argument space (to pass (store) parameters when method calls other methods)
- Temporaries - locations where code temporarily saves register values when necessary

Frame pointer and stack pointer

- Caller *g(...)* calls callee $f(a_1,...,a_n)$
- Calling code in *g* puts arguments to *f* at end of *g* frame
  - referenced through SP, incrementing SP
- On entry to *f*,
  - SP points to first argument *g* passes to *f*
  - old SP becomes current *frame pointer* FP
  - *f* then allocates frame by setting SP=(SP - framesize)

Frame pointer and stack pointer

- Old SP becomes current *frame pointer* FP
- Many implementations have FP as a separate register
  - so method code:
    - has incoming arguments referenced by FP-an offset
    - has local variables referenced by SP+an offset or FP-an offset
    - has saved registers, return address and outgoing arguments referenced by SP+an offset
- On exit from *f* : SP = FP, removing frame

Registers

- Fast execution ⇒ keep local variables, intermediate results of expressions etc in registers, not stack frame (memory)
- Registers are accessed directly by arithmetic instructions
  - (memory access requires load & store instructions; even if arithmetic instructions access memory, registers are always faster)

Registers

- ***caller save*** vs ***callee save*** registers
  – method f uses reg r to hold local variable; then f calls g and g uses r
  – which code saves r contents in stack frame, f or g?
  – often machine conventions defining set of caller- and callee-saves
- Sometimes saves & restores unnecessary
  – if variable not required after call, caller code can put in a caller-save register and compiler leaves out the code to save it before call
  – if local variable i in f needed before & after many method calls - put in callee-save register, save once on entry to f, fetch back before returning from f
- Register allocator in compiler chooses best register set

Parameter passing

- Pre-1960: passed in statically allocated memory blocks - no recursive functions or methods
- 1970s machines: function arguments passed on the stack
- But program analysis shows that very few functions/methods have >4 arguments, and almost none >6.

Parameter passing

- So on most modern machines
  – first k arguments (k=4 or 6) are passed in registers $r_p,\ldots,r_{p+k-1}$ and the rest passed in memory on the stack
- But if function or method call $f(a_1,\ldots a_n)$
  – receives its parameters in registers $r_1\ldots r_n$
  – and then calls h(z), argument z is passed in r_1
  – f must save old contents of r_1 (contents of a_1) in stack frame before calling h
  – this is memory traffic, so has use of registers saved any time?
  – (it of course might be worse: $h(z_1,z_2,z_3,z_4\ldots)$)

Why use registers?

- *Leaf functions* or methods (don't call other methods)
  – no need to write incoming arguments to memory; often no need even to create new stack frame
- *Interprocedural register allocation*
  – analyse all methods in entire program
  – assign different methods different registers to receive parameters & hold variables
  – eg f(x) receives x in r1, calls h(z): z in r7

Why use registers?

- *Dead variables* on method call: overwrite registers
- Register windows
  – architecture has fresh set of registers (a window) for each method invocation
  – but eventually run out; then a window must be saved on stack

Parameter-passing calling convention

- Even if arguments are passed in registers, and do not need to be saved into stack, space is reserved in the stack
  – *Caller* code reserves space for arguments that are passed in registers next to the space for any other arguments
  – But does not save anything into this space
  – *Callee* code saves into this space if necessary

Parameter-passing calling convention

- When is it necessary to save like this?
  - In some languages, the address of a parameter may be taken
    - This must be a memory (ie stack) address, not a register
  - Some languages have call-by-reference parameter passing
  - When a *register window* must be saved

Frame-resident variables

- Code generator produces code to write values from registers to the stack frame only when:
  - Variable will be passed-by-reference, or its address is taken
  - Variable is accessed by a function/method nested inside current
  - Value is too big to fit in a register
  - Variable is an array
  - Register holding the variable is needed for a specific purpose (eg parameter passing)

Frame-resident variables

  - There are so many local variables and temporary values necessary to perform expression computations that they won't all fit in the available registers (spilling)
- A variable ***escapes*** (code from outside its function/method may access it) if
  - It is passed as a parameter by reference
  - Its address is taken
  - It is accessed *from* a nested function/method

Escapes in MiniJava

- Thankfully, there are none!
  - No nesting of classes and methods
  - Not possible to take address of variable
  - Integers and booleans passed by ***value***
  - Objects, including integer arrays, represented by pointers also passed by value

Block structure - static links

```
type tree =  {key: string, left: tree, right: tree}

function prettyprint(tree: tree) : string =
 let
      var output := ""

      function write(s:string) =
               output := concat(output,s)

      function show(n:int, t:tree) =
               let function indent(s:string) =
                        (for i := 1 to n
                           do write(" ");
                         output := concat(output,s); write("\n"))
               in if t=nil then indent(".")
                  else (indent(t.key);
                        show(n+1,t.left);
                        show(n+1,t.right))
               end

     in show(0,tree); output
  end
```


Static links

- Block structure
  - Nested method/function definitions use variables or parameters declared in outer definitions
- Whenever a function f is called, a pointer to the frame of the function statically enclosing f is passed
  - This is the static link
  - It points to the most recent activation of the enclosing function

Static links

- when a function f at nesting depth f_d calls (caller) a function g at depth g_d (callee), the static link set up is
  – to caller, if g is declared within f
  – computed by following f_d - g_d static links, if g is declared outside f
- a variable or parameter declared in a function g at depth g_d is accessed from function f at depth f_d
  – by code that follows f_d - g_d static links to get to the appropriate frame

Static links examples

21 `prettyprint` calls `show`, passes `prettyprint`'s own frame pointer as `show` 's static link

10 `show` stores its static link (address of `prettyprint`'s frame) into its own frame

15 `show` calls `indent`, passing its own frame pointer as `indent`'s static link

17 `show` calls `show`, passing its own static link (not frame pointer) as static link

12 `indent` uses value n from show's frame - fetches appropriate offset from `indent` 's static link

13 `indent` calls `write`. Passes frame pointer of `prettyprint` as static link. Fetches an offset from its own static link (from `show`'s frame) - the static link passed to `show`

14 `indent` uses var output from `prettyprint` 's frame; starts with own static link, then fetches `show`'s then fetches output

General Frame package

- **`abstract class Frame.Access`**
  – Describes formals and local variables that may be in frame or registers
  ```
  class inFrame extends Frame.Access { int offset; … }
  class inReg extends Frame.Access { Temp temp; … }
  ```
- **`abstract class Frame.Frame`**
  – A list of formals (an AccessList) denoting locations where formals will be accessed by method/function (callee) code
  – Method **`Frame newFrame(Label `*`name`*`, Util.BoolList formals)`**
    - for k parameters, list of k booleans, true for each parameter that escapes

General Frame package

  – Method **`Access allocLocal(boolean escape)`**
    - allocates space in frame for a local which may be an InFrame or an InReg
  – Hides the machine architecture; for particular architecture eg MIPS
    - will have class MIPS.Frame extends Frame.Frame, and
    - classes MIPS.InFrame, MIPS.InReg extends Frame.Access
- An abstract syntax tree traversal can calculate escapes
  – none in MiniJava, as we saw earlier

What you should do now…

- Read and digest chapter 6
  – but you don't need *higher order functions*
- think about writing a Frame package for MiniJava
  – remember no nested methods in MiniJava
  – so no static links