CS 536 Announcements for Monday, April 8, 2024

Last Time
- parameter passing
- terminology
- different styles
  - what they mean
  - how they look on the stack

Today
- wrap up parameter passing
- compare and contrast
- accessing variables at runtime
  - how do we deal with variables and scope?
  - how do we organize activation records?
  - how do we retrieve values of variables from activation records?

Next Time
- code generation

Code generation and parameter passing

Efficiency considerations (calls, accesses by callee, return)

Pass by value
- copy values into callee’s AR - slow
- callee directly accesses AR locations - fast

Pass by reference
- copy addresses into callee’s AR - fast
- access in callee via indirection - slow

Pass by value-result
- strictly slower than pass by value
- need to know where to copy values back on return

Handling objects
In Java, variables hold the addresses of objects
- no overhead of copying entire objects

In C++, variables are objects in the stack
- use pointers to objects in heap for efficiency
Compare and contrast

Pass by value
- no aliasing
- easier for static analysis
- called function (callee) is faster

Pass by reference
- more efficient when passing large objects
- can modify actuals

Pass by value-result
- more efficient than pass by reference for small objects
- if no aliasing, can be implemented as pass by reference for large objects

but determining if there is aliasing (and what is aliased) is a challenging task (in general)

Accessing variables at runtime

local variables
- declared and used in the same function
- further divided into "block" scope in base

global variables
- declared at the outermost level of the program
- in C/C++/base - globals - integer x.
- in Java - class (static) data members - Java keyword

non-local variables (i.e., from nested scopes)
- for static scope: variables declared in an outer scope
- for dynamic scope: variables declared in the calling context

- compile-time vs run-time
- nested class (Java)
- nested procedures (Pascal)
Accessing local variables at runtime

Local variables
- includes parameters and all local variables in a function
- stored in activation record of function in which they are declared
- accessed using offset from frame pointer

Accessing the stack
- general anatomy of MIPS instruction
  \[ \text{opcode operand1 operand2} \]
- use "load" and "store" instructions
  - every memory cell has an address
  - calculate that memory address, then move data from/to that address

```c
void test(int x, int y) {
    int a, b;
    ...
    if (...) {
        int s;
        ...
    }
    else {
        int t, u, v;
        ...
        u = b + y;
    }
}
```

<table>
<thead>
<tr>
<th>Activation record for test</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
</tr>
<tr>
<td>s</td>
</tr>
<tr>
<td>t</td>
</tr>
<tr>
<td>u</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>v</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>x</td>
</tr>
</tbody>
</table>

MIPS code for \( u = b + y \)

```assembly
lw  $t1, -12($fp)  # load b
lw  $t2,  8($fp)   # load y
add $t3, $t1, $t2  # b+y
sw  $t3, -24($fp)  # store into u
```
Simple memory-allocation scheme

Reserve a slot for each variable in the function

Algorithm (for each function)

```plaintext
set offset = +4

for each parameter
  add name to symbol table
  offset += size of parameter

offset = -4  \( \text{to account for control link} \)

offset -= size of callee saved registers

for each local
  offset -= size of variable
  add name to symbol table
```

Implementation (in P6)

- add an offset field to each symbol table entry
- during name analysis, add the offset along with the name
- walk the AST performing decrements at each declaration node

Example

```plaintext
void test(int x, int y) {
  int a, b;
  if (...) {
    int s;
  } else {
    int t, u, v;
    u = b + y;
  }
}
```

Accessing global variables at runtime

Place in static data area

- in MIPS, handled with a special storage directive
- variables referred to by name, not address

Note: space allocated directly at compile time (never needs to be deallocated)

Example

```plaintext
.data
  _x: .word 10

.text
  lw $t0, _x  # load from x into $t0
  sw $t0, _x  # store from $t0 into x
```
Accessing non-local variables at runtime

Two situations
- static scope
  - variable declared in one procedure and accessed in a nested one
- dynamic scope
  - any variable x that is not declared locally resolves to instance of x in the AR closest to the current AR

Example: static non-local scope

```c
function main() {
    int a = 0;

    function subprog() {
        a = a + 1;
    }
}
```

Example: static non-local scope

```c
void procA() {  // level 1
    int x, y;
    void procB() {  // level 2
        print x;   // x1 (always)
    }
    void procC() {  // level 2
        int z;
        void procD() {  // level 3
            int x;
            x = z + y;  // x3 = z2 + y1
            procB();
        }
        x = 4;  // x1 = 4
        z = 2;  // z2 = 2
        procB();
        procD();
    }
    x = 3;  // x1 = 3
    y = 5;  // y1 = 5
    procC();
}
```
Access links

Add additional field in the AR (called access link, or static link)

points to locals area (or fp) of enclosing procedure

How access links work

• we know how many levels to traverse statically

  • current scope is at nesting level 3
  • variable x is at nesting level 1
  • go back 3 - 1 = 2 access levels

Setting up access links

void procA() {
  int x, y;
  void procB() {
    print x;
  }
  void procC() {
    int z;
    void procD() {
      int x;
      x = z + y;
      procB();
    }
  }
  x = 4;
  z = 2;
  procB();
  procD();
  x = 3;
  y = 5;
  procC();
}

Handling use of non-local variable x (at compile time)

• each variable keeps track of nesting level in which it is declared
• when x is used in procedure P
  • follow predetermined # of links to get to AR for procedure in which x is declared

MIPS (assume $fp is location of access link)

lw $t0, 0($fp) # 1 link followed
lw $t0, ($t0) # 2 links followed

lw $t0, -12($t0) # use x's offset in AR of declaring procedure
Using a display

Idea: avoid run-time overhead of following access links by having a global array (called the display) containing links to the procedures that lexically enclose the current procedure

How it works
- given procedure P at nesting level k is currently executing
- display[0], display[1], ..., display[k-2] hold pointers to ARs of the most recent activations of the k-1 procedures that enclose P
- display[k-1] holds pointer to P's AR
- to access non-local variable x declared in nesting level n
  - use display[n-1] to get to AR that holds x
  - then use regular offset (within AR) to get to x

How to maintain the display in the code
- add new "save-display" field to AR
- when procedure P at nesting level k is called
  - save current value of display[k-1] in save-display field of P's AR
  - set display[k-1] to point to save-display field of P's AR
- when procedure P is ready to return
  - restore display[k-1] using value in save-display field

Example
void procA() {
    int x, y;
    void procB() {
        print x;
    }
    void procC() {
        int z;
        void procD() {
            int x;
            x = z + y;
            procB();
        }
        x = 4;
        z = 2;
        procB();
        procD();
    }
    x = 3;
    y = 5;
    procC();
}
Dynamic non-local scope

Example

```javascript
function main() {
    int a = 0;
    fun1();
    fun2();
}
function fun2() {
    int a = 27;
    fun1();
}
function fun1() {
    a = a + 1;
}
```

**Key point** – we don't know *which* non-local variable we are referring to

**Two ways to set up dynamic access**
- deep access – somewhat similar to access links
- shallow access – somewhat similar to displays

**Deep access**
- if the variable isn't local
  - follow control link to caller's AR
  - check to see if it defines the variable
  - if not, follow the next control link down the stack
- note that we need to know if a variable is defined *with that name* in an AR
  - usually means we'll have to associate a name with a stack slot

**Shallow access**
- keep a table with an entry for each variable declaration
- compile a direct reference to that entry
- at function call on entry to function F
  - F saves (in its AR) the current values of all variables that F declares itself
  - F restores these values when it finishes