Runtime Access to Variables
Roadmap

Last Time
- Parameter-passing conventions

This time
- How do we deal with variables and scope?
- How do we organize activation records?
- How do we retrieve values of variables from activation records?
Scope

We mostly worry about 3 flavors

- Local
  - Declared and used in the same function
  - Further divided into “block” scope in b

- Global
  - Declared at the outermost level of the program

- Non-local (i.e., from nested scopes)
  - For static scope: variables declared in an outer scope
  - For dynamic scope: variables declared in the calling context
Local Variables: Examples

What are the local variables here?

```c
int fun(int a, int b) {
    int c;
    c = 1;
    if (a == 0) {
        int d;
        d = 4;
    }
}
```
How Do We Access the Stack?

Need a little MIPS knowledge
– Full tutorial next week
– General anatomy of a MIPS instruction

opcode Operand1 Operand2
How Do We Access the Stack?

Use “load” and “store” instructions

– Recall that every memory cell has an address
– Calculate that memory address, then move data from/to that address
Basic Memory Operations

lw register memoryAddress

register = *memoryAddress;

*memoryAddress = register;

sw register memoryAddress
Load-Word Example

\[ \text{lw} \quad \$t1, \quad -20(\$fp) \]

\[ \$t1 = *(\$fp - 20); \]
Load Word in Action

lw  $t1, -12($fp)
sw $t1, 0($fp)
Relative Access for Locals

Why do we access locals from $fp?
– That’s where the activation record starts

What if we used $sp instead?

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3f0</td>
<td>0x3f0</td>
</tr>
<tr>
<td>0x3f4</td>
<td>0x3f4</td>
</tr>
<tr>
<td>0x3f8</td>
<td>0x3f8</td>
</tr>
<tr>
<td>0x3fc</td>
<td>0x3fc</td>
</tr>
<tr>
<td>0x400</td>
<td>0x400</td>
</tr>
</tbody>
</table>

$sp 0x334
$fp 0x400

4 bytes
A Simple Memory-Allocation Scheme

Reserve a slot for each variable in the function

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

```
<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0x400</td>
</tr>
<tr>
<td>y</td>
<td>0x3fc</td>
</tr>
<tr>
<td>(x)</td>
<td>0x3d4</td>
</tr>
<tr>
<td>(y)</td>
<td>0x3e0</td>
</tr>
<tr>
<td>(v)</td>
<td>0x3e4</td>
</tr>
<tr>
<td>(u)</td>
<td>0x3e8</td>
</tr>
<tr>
<td>(s)</td>
<td>0x3f0</td>
</tr>
<tr>
<td>(b)</td>
<td>0x3f4</td>
</tr>
<tr>
<td>(control link)</td>
<td>0x3f8</td>
</tr>
<tr>
<td>(return addr)</td>
<td>0x400</td>
</tr>
<tr>
<td>a</td>
<td>0x3d4</td>
</tr>
<tr>
<td>b</td>
<td>0x3ec</td>
</tr>
<tr>
<td>s</td>
<td>0x3e0</td>
</tr>
<tr>
<td>t</td>
<td>0x3e8</td>
</tr>
<tr>
<td>u</td>
<td>0x3e4</td>
</tr>
<tr>
<td>v</td>
<td>0x3e0</td>
</tr>
</tbody>
</table>
```
Simple Memory-Allocation Algorithm

For each function
Set offset = +4
for each parameter
    add name to symbol table
    offset += size of parameter
offset = -4
offset -= size of callee saved registers
for each local
    offset -= size of variable
    add name to symbol table
Simple Memory-Allocation Implementation

Add an offset field to each symbol table entry
During name analysis, add the offset along with the name (Wait until Project 6 to do this)
Walk the AST performing decrements at each declaration node
Algorithm Example

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

In a sense, globals easier to handle than locals

- Space allocated directly at compile time instead of indirectly via $fp and $sp registers
- Never needs to be deallocated

Place in static data area

- In MIPS, handling with a special storage directive
- Variables referred to by name, not by address
Memory-Region Example

.data
_x: .word 10
_y: .byte 1
_z: .asciiz "I am a string"
.text
lw $t0, _x  #Load from x into $t0
sw $t0, _x  #Store from $to into x
Accessing Non-Local Variables

Static scope
– Variable declared in one procedure and accessed in a nested one

Dynamic scope
– Any variable $x$ used that is not declared locally resolves to instance of $x$ in the AR closest to the current AR
Example: Static Non-Local Scope

Each function has its own AR
– Inner function accesses the outer AR

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

    function subprog(){
        a = a + 1;
    }
}
```
Memory Access: Static Non-Local Scope

```c
void procA() {   // level 1
    int x, y;
    void procB() {  // level 2

        void procC() {  // level 3
            int z;
            void procD() { // level 4
                int x;
                x = z + y;
                procB();
            }
            x = 4;
            z = 2;
            procB();
            procD();
        }
        x = 3;
        y = 5;
    }
```
Access Links

Add an additional field in the AR

– Points to the locals area of the outer function

– Sometimes called the static link (since it refers to the static nesting)
How Access Links Work

We know how many *levels* to traverse statically

- Example: When current scope is at nesting level 3 and the variable that we want to access is at nesting level 1: go back 2 access links

  \[(3 - 1) \text{ 2 levels}\]
Traversing Stack Using Access Links

Using 1 access link

```
lw  $t0, 0($fp)
lw  $t0, -20($t0)
```

Where $fp is the location of the access link, and the variable in the outer scope is at offset 20 in its AR

Using 2 access links

```
lw  $t0, 0($fp)
lw  $t0, ($t0)
lw  $t0, -20($t0)
```
Thinking About Access Links

We know the variable we want to access statically.

Why don’t we just index into the parent’s AR using a large positive offset from $fp$?

```
lw $t0 380($fp)
```
Displays

High-level idea:
– Keep the transitive effects of multiple access-link traversals
– Uses a side-table with this info

Tradeoffs vs. Access Links?
– Faster to call far up the hierarchy
– Takes extra space
  • At most the maximum nesting depth in the entire program
  • Therefore, the display can be an array (a stack no bigger than a known maximum size)
program Main;
var x: integer;

procedure P;
  write(x);
end;

procedure Q;
var y: integer = x;
end;

procedure R;
x = x + 1;
y = y + x;
if y < 6 call R;
call P
end;

call R;
call P;
if x < 5 call Q;
x = 2;
call P;
call Q;
program Main;
var x: integer;

procedure P;
write(x);
end;

procedure Q;
var y: integer = x;
procedure R;
x = x + 1;
y = y + x;
if y<6 call R;
call P
end;
end;
end;
call P
end;
call R;
call P;
if x < 5 call Q;
x = 2;
call P;
call Q;

Level (1)
Level (2) With access links
Level (2) With a display

Main y:
x:

P
R
Q
program Main;
  var x: integer;

procedure P;
  write(x);
end;

procedure Q;
  var y: integer = x;
end;

procedure R;
  x = x + 1;
  y = y + x;
  if y < 6 call R;
  call P;
end;

x = 2;
call P;
call Q;
Questions about Static Scope?
Example: Dynamic Non-Local Scope

```plaintext
function main(){
    int a = 0;
    fun1();
    fun2();
}

function fun2(){
    int a = 27;
    fun1();
}

function fun1(){
    a = a + 1;
}
```
Dynamic Scope Storage

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

Two ways to set up dynamic access
1. Deep Access – somewhat similar to Access links
2. Shallow Access – somewhat similar to displays
Deep Access

If the variable isn’t local
– Follow the control link to the caller’s AR
– Check to see if it defines the variable
– If not, follow the next control link down the stack

Note that we somehow 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 a function call on entry to function F
  • F saves, in its own AR, the current values of all of the variables that F declares itself
  • F restores those values when it finishes
Roadmap

We learned about variable access

– Local vs. global variables
– Static vs. dynamic scopes

Next time

– We’ll start getting into the details of MIPS
– Code generation