Run-Time Storage Organization

Leonidas Fegaras
• Memory layout of an executable program:
Run-Time Stack

• At run-time, function calls behave in a stack-like manner
  – when you call, you push the return address onto the run-time stack
  – when you return, you pop the return address from the stack
  – reason: a function may be recursive

• When you call a function, inside the function body, you want to be able to access
  – formal parameters
  – variables local to the function
  – variables belonging to an enclosing function (for nested functions)

```pascal
procedure P ( c: integer )
x: integer;
procedure Q ( a, b: integer )
i, j: integer;
begin
  x := x+a+j;
end;
begin
  Q(x,c);
end;
```
Activation Records (Frames)

- When we call a function, we push an entire frame onto the stack.
- The frame contains:
  - the return address from the function
  - the values of the local variables
  - temporary workspace
  - ...
- The size of a frame is not fixed:
  - need to chain together frames into a list (via dynamic link)
  - need to be able to access the variables of the enclosing functions efficiently.
A Typical Frame Organization

- Argument 1, Argument 2, ..., Argument n
- Dynamic link
- Return address
- Static link
- Local and temporary variables
- Next frame (if not top of stack)
- Lower addresses
- Higher addresses
- Previous frame
- Activation record (frame)
Static Links

- The static link of a function $f$ points to the latest frame in the stack of the function that statically contains $f$
  - If $f$ is not lexically contained in any other function, its static link is null.

```latex
\begin{verbatim}
procedure P ( c: integer )
    x: integer;

    procedure Q ( a, b: integer )
        i, j: integer;
        begin
            x := x+a+j;
        end;

    begin
        Q(x,c);
    end;
\end{verbatim}
```

- If $P$ called $Q$ then the static link of $Q$ will point to the latest frame of $P$ in the stack.

- Note that
  - we may have multiple frames of $P$ in the stack; $Q$ will point to the latest
  - there is no way to call $Q$ if there is no $P$ frame in the stack, since $Q$ is hidden outside $P$ in the program.
The Code for Function Calls

• When a function (the caller) calls another function (the callee), it executes the following code:
  – *pre-call*: do before the function call
    • allocate the callee frame on top of the stack
    • evaluate and store function parameters in registers or in the stack
    • store the return address to the caller in a register or in the stack
  – *post-call*: do after the function call
    • copy the return value
    • deallocate (pop-out) the callee frame
    • restore parameters if they passed by reference
The Code for Function Calls (cont.)

- In addition, each function has the following code:
  - **prologue:** to do at the beginning of the function body
    - store frame pointer in the stack or in a display
    - set the frame pointer to be the top of the stack
    - store static link in the stack or in the display
    - initialize local variables
  - **epilogue:** to do at the end of the function body
    - store the return value in the stack
    - restore frame pointer
    - return to the caller
We can classify the variables in a program into four categories:

1) statically allocated data that reside in the static data part of the program
   – these are the global variables.

1) dynamically allocated data that reside in the heap
   – these are the data created by malloc in C

1) register allocated variables that reside in the CPU registers
   – these can be function arguments, function return values, or local variables

1) frame-resident variables that reside in the run-time stack
   – these can be function arguments, function return values, or local variables
• Every frame-resident variable (ie. a local variable) can be viewed as a pair of (level, offset)
  – the variable level indicates the lexical level in which this variable is defined
  – the offset is the location of the variable value in the run-time stack relative to the frame pointer

```
procedure P ( c: integer )
x: integer;
procedure Q ( a, b: integer )
i, j: integer;
begin
  x := x+a+j;
end;
begin
  Q(x,c);
end;
```

<table>
<thead>
<tr>
<th></th>
<th>level</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
<td>-12</td>
</tr>
<tr>
<td>j</td>
<td>2</td>
<td>-16</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>-12</td>
</tr>
</tbody>
</table>
procedure P ( c: integer )
x: integer;
    procedure Q ( a, b: integer )
    i, j: integer;
    begin
        x := x+a+j;
    end;
begin
    Q(x,c);
end;

The view of the stack inside procedure P

The view of the stack inside procedure Q

Run-time stack at the point of x := x+a+j
Accessing a Variable

- Let $fp$ be the frame pointer
- You are generating code for the body of a function at the level L1
- For a variable with (level, offset) = (L2, O) you generate code:
  1) traverse the static link (at offset -8) L1-L2 times to get the containing frame
  2) access the location at the offset O in the containing frame
- eg, for L1=5, L2=2, and O=-16, we have
  - Mem[Mem[Mem[Mem[Mem[$fp-8]-8]-8]-8]-16]
- eg:

<table>
<thead>
<tr>
<th></th>
<th>level</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
<td>-12</td>
</tr>
<tr>
<td>j</td>
<td>2</td>
<td>-16</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>-12</td>
</tr>
</tbody>
</table>
The Code for the Call Q(x,c)

\[
\begin{align*}
\text{Mem}[\$sp] &= \text{Mem}[\$fp-12] & \text{; push x} \\
\$sp &= \$sp-4 \\
\text{Mem}[\$sp] &= \text{Mem}[\$fp+4] & \text{; push c} \\
\$sp &= \$sp-4 \\
\text{static\_link} &= \$fp \\
call Q \\
\$sp &= \$sp+8 & \text{; pop arguments}
\end{align*}
\]
The Code for a Function Body

• Prologue:
  Mem[$sp] = $fp ; store $fp
  $fp = $sp ; new beginning of frame
  $sp = $sp+frame_size ; create frame
  save return_address
  save static_link

• Epilogue:
  restore return_address
  $sp = $fp ; pop frame
  $fp = Mem[$fp] ; follow dynamic link
  return using the return_address
Finding Static Link

- The caller set the static_link of the callee before the call
  - this is because the caller knows both the caller and callee
  - the callee doesn't know the caller

- Suppose that L1 and L2 are the nesting levels of the caller and the callee procedures
  - When the callee is lexically inside the caller's body, that is, when L2=L1+1, we have:
    
    static_link = $fp
    
    - Otherwise, we follow the static link of the caller L1-L2+1 times

- For L1=L2, that is, when both caller and callee are at the same level, we have
  static_link = Mem[$fp-8]

- For L1=L2+2 we have
  static_link = Mem[Mem[Mem[$fp-8]-8]-8]
Finding Static Link (cont.)

1) R calls Q
   \[ \text{level}(R) = \text{level}(Q) - 1 \]
   the static link of Q is set to the beginning of R
   \[ \text{move} \$v0, \$fp \]

2) R calls Q
   \[ \text{level}(R) = \text{level}(Q) \]
   the static link of Q is set to the static link of R
   \[ \text{lw} \$v0, -8(\$fp) \]

3) R calls Q
   \[ \text{level}(R) = \text{level}(Q) + n \]
   the static link of Q is set by following the static link of R \( n+1 \) times
   \[ \text{lw} \$t0, -8(\$fp) \]
   \[ \text{lw} \$t0, -8(\$t0) \]
   \[ \text{lw} \$v0, -8(\$t0) \]