Activation Records

Activation records (or stack frames)
More on stack frames
Stack frames for MiniJava
Variable Instantiations (1)

- The local variables in a function $f$ are created upon entry to $f$.

- Multiple invocations of $f$ may exist at the same time. How?

- Each invocation has its own instantiations of the local variables.
Variable Instantiations (2)

- New instantiation of x created each time f is called.
  ```
  int f(int x) {
    int y = x+x;
    if (y<10)
      return f(y);
    else
      return y-1;
  }
  ```
- How is x initialized
- Many x's may exist simultaneously.
- What about y?
- In languages such at C and Java, local variables are destroyed when function returns (and are created on function entry).
- We can use a LIFO data structure to hold local variables.
Higher-Order Functions

- Higher-order functions:
  nested functions + functions as returnable values
- Languages (such as ML) support higher-order functions.
- In this case, it is necessary to keep local variables after a function has returned.
- We cannot use a stack to hold all local variables for such languages.
Higher-Order Function Example

fun f(x) = 
    let fun g(y) = x+y
        in g
    end

val h = f(3)
// creates a new x(=3), returns g(y)=x+y
val j = f(4)
// creates a new x(=4), returns g(y)=x+y
val z = h(5)
// computes g(5)=3+5
val w = j(7)
// computes g(7)=4+7
Stack (1)

- We use a *stack* as the LIFO data structure for holding local variable instantiations.

- A “real” stack supports only *push* and *pop* operations.

- However, local variables are pushed (upon function entry) and popped (upon function exit) in large batches.

- Also, after pushing on many variables, we may want to continue accessing variables deep in the stack.
Instead, we treat the stack as a large array.

The stack pointer is a special register that points to some location in the stack.

All locations beyond stack pointer are considered garbage, all locations before are considered allocated.

Upon function entry, stack grows by enough to hold all local variables.
Stack Frame

- A function's *stack frame* (or activation record) is the area on the stack devoted to its local variables, parameters, return address, and other temporaries.

- Usually, stacks start at high memory addresses and grow to low memory addresses.

- Often, each computer architecture has a standard stack frame layout, making it possible for functions written in one language to call functions written in another.
Incoming arguments are technically part of the previous frame, passed by the caller.

However, they are located at a known offset from the frame pointer.
## Stack Frame Layout (3)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>local variables</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>temporaries</td>
</tr>
<tr>
<td>saved registers</td>
</tr>
</tbody>
</table>

- Return address is created by the CALL instruction, indicating where to return control upon completion of the current frame.
- Some local variables are in the frame, others in registers.
- Contents of registers may be saved to the frame, to make room for other uses.
If and when the current function calls other functions, it can use the space for outgoing arguments to pass parameters.
Frame Pointer

- On entry, \( f \) updates the stack pointer (SP) by subtracting \( f \)'s frame size from the old SP.

- The old SP becomes the current frame pointer (FP).

- Depending on whether frame size is fixed, the FP may be a separate register (or known offset of SP).

- Why have an FP at all? \( f \)'s frame size is not known until late, when the number of temporaries and saved registers have been determined.
Registers

- Modern machines typically have 32 registers.

- To make programs fast, it is useful to keep local variables, intermediate expression results, and other values in registers rather than on the stack.

- Many different functions need to use the registers.

- Suppose function g is using register r and then calls f(), which also uses register r.

- r must be saved before f uses it and restored after f.
Saving Registers

- Is it g or f's responsibility to save/restore r?

- r is a **caller-save** register if g (the caller) must save/restore it.

- r is a **callee-save** register if f (the callee) must save/restore it.

- Whether a register is caller- or callee-save is merely a *convention*.

- For MIPS, registers 16-23 are callee-save.
Parameter Passing (1)

- Our stack frame layout indicates that all function arguments are passed on the stack.

- Studies of actual programs show that very few functions have $> 4$ arguments, almost none have $> 6$.

- For so few arguments, passing on the stack causes needless memory traffic.

- Modern convention is to pass the first $k$ arguments in registers and pass any remaining arguments on stack.
Parameter Passing (2)

- Suppose function $f(a_1,\ldots,a_n)$ receives its parameters in registers $r_1,\ldots,r_n$.

- If $f$ calls $h(z)$, it must pass $z$ in $r_1$.

- Thus, $f$ must save the old contents of $r_1$ in its stack frame before calling $h$.

- Aren't we introducing the memory traffic that was supposed to be avoided by passing the first $k$ arguments in registers?
## Stack Frame Layout

<table>
<thead>
<tr>
<th>incoming arguments</th>
<th>( \text{argument } n)</th>
<th>( \text{argument } k+2)</th>
<th>( \text{argument } k+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame pointer</td>
<td>local variables</td>
<td>return address</td>
<td>temporaries</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>saved registers</td>
</tr>
<tr>
<td>outgoing arguments</td>
<td>( \text{argument } m)</td>
<td>( \text{argument } k+2)</td>
<td>( \text{argument } k+1)</td>
</tr>
</tbody>
</table>

\[\text{higher addresses}\]

\[\text{previous frame}\]

\[\text{next frame}\]

\[\text{lower addresses}\]
Return Address (1)

● When g calls f, f must know where to go back to when it is finished.

● If the instruction within g that calls f is at address $a$, then f should go back to address $a+1$, the return address.

● In older machines (70s), the *call* instruction pushed the return address onto the stack.

● Now, *call* puts the return address in a register.
Return Address (2)

- Putting the return address in a designated register (ra) is intended to avoid memory traffic.

- What happens if f (the callee) calls yet another function?

- f will have to write the contents of ra to the stack before calling the new callee.

- If f does not call any functions, the contents of ra need not be written to the stack.
Frame-Resident Variables (1)

- We pass function parameters (at least most of them) in registers.

- We pass the return address in a register.

- We return the function result in a register.

- We use registers whenever possible to hold local variables and intermediate results.

- When are values written to the stack?
Frame-Resident Variables (2)

Values are written to the stack only when necessary, i.e., for one of these reasons.

- variable passed by reference (need a memory address)
- variable accessed by procedure nested inside current one
- value to big for a single register
- variable is an array (address arithmetic needed)
- register holding value is needed for another purpose
- too many local variables and temps, “spill” onto stack
Storing Variables

● When a formal parameter or local variable is declared, we would like to assign it a location (a register or in the stack).

● Then at each use of the variable, we could generate machine code that refers to the right location.

● However at the declaration, we do not know enough about how the variable will be used or if there will be spillage.
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
Nested Functions (2)

- Function `indent` must have access to its own stack frame (for `i` and `s`) and to the frames of `show` (for `n`) and `prettyprint` (for `output`).

- Whenever a function is called, it is passed a pointer to the stack frame of the function immediately enclosing it, a `static link`.

- `indent` uses its static link (passed from `show`) to reach `n` and to fetch `show`'s static link (passed from `prettyprint`) to then reach `output`. 
Stack Frame Layout

- **Incoming Arguments**
  - Frame pointer
- **Outgoing Arguments**
  - Stack pointer

- **Arguments**
  - Argument \(n\)
  - Argument \(k+2\)
  - Argument \(k+1\)
- **Static Link**
- **Local Variables**
- **Temporaries**
- **Saved Registers**
  - Argument \(m\)
  - Argument \(k+2\)
  - Argument \(k+1\)
- **Higher Addresses**
- **Lower Addresses**
- **Previous Frame**
- **Next Frame**
Stack Frame Layout

- Recall that each target-machine architecture may have a different standard stack frame layout.

- What sort of layout should MiniJava use?
  - A standard stack frame layout?
  - If so, the standard layout of which target-machine architecture?

- We could use an abstraction of the stack frame layout, so that machine-independent parts of the compiler need not know of the target machine.
package Frame;
import Temp.Temp;  //names for local vars
import Temp.Label;  //names for static mem addresses
import Util.BoolList;
//describes formals, locals (may be in frame or regs)
public abstract class Access {...}
//list of Accesses
public abstract class AccessList {...}
//an abstract stack frame representation
public abstract class Frame {
    public Label name;
    public AccessList formals;
    public abstract class Frame newFrame(Label name,
                              BoolList formals);
    public abstract Access allocLocal(boolean escape);
    ...
}
newFrame()

- To create a new frame for function $f$ with $k$ formal parameters, call `newFrame(f_label, list)`.

- `f_label` is a `Label` representing the memory address where $f$’s machine code is located.

- `list` contains $k$ booleans, indicating whether each formal parameter “escapes”.

- What does it mean if a variable escapes?
Variables That “Escape”

- We say that a variable escapes if it
  - is passed by reference,
  - has it's address taken (using C's & operator),
  - or is accessed from a nested function.

- What do such variables have in common?

- How does this impact `newFrame()`?

- In MiniJava, can any variables escape?
Target-Specific Stack Frame

- A module specific to the target machine extends the abstract classes Frame and Access.

- Implementation of Access, which describes formals and locals that may be in the frame or in registers:

```java
package Mips;
class InFrame extends Access {
    int offset; ...
}
class InReg extends Access {Temp temp;...}
```
Location of Formal Parameters

- The `formals` field is a list of `k Accesses`, which denote locations where the formal parameters will be kept at run time as seen from the callee.

- Is this different from how it's seen from the caller?

- If parameter on stack, caller may put it at offset 4 from SP. Callee sees it at offset 4 from FP.

- If caller puts parameter in register r6, callee may want to move it out of the way to r13.
“Shift of View”

- “Shift of view” depends on the calling conventions of the target machine.

- The Frame module handles the view shift in `newFrame()`.

- For each formal parameter, `newFrame()` must decide
  - how the parameter is seen from the caller (in register or on stack frame)
  - what instructions are necessary to view shift
Recap: Frame

● We have an abstract representation of a stack frame accessible to the machine-independent parts of the compiler, and a target-machine specific module that implements the abstract classes.

● A `Frame` object holds:
  ● the locations of all formal parameters,
  ● the `Label` for memory address of machine code,
  ● a method for allocating memory for local vars,
  ● and “shift of view” instructions.
● Suppose that function $g$ has three formal parameters, the first of which escapes.
● $g$'s caller passes the parameter values in registers $r4$, $r5$, and $r6$.

| View Shift | sp $\leftarrow$ sp $- K$
|            | $M[sp+K+0] \leftarrow r4$
|            | $t_{157} \leftarrow r5$
|            | $t_{158} \leftarrow r6$

| Formals   | 1 InFrame(0)
|           | 2 InReg($t_{157}$)
|           | 3 InReg($t_{158}$)

update stack pointer
move $r4$ contents to stack
move $r5$ contents to temp
move $r6$ contents to temp
Example (2)

● Why move the contents of registers r5 and r6 to temporaries $t_{157}$ and $t_{158}$?

● Suppose that r5 is the standard register for parameter 2.

● If $g$ calls a function with two or more arguments, the value in r5 will be overwritten.

● The register allocator eventually chooses which registers should hold $t_{157}$ and $t_{158}$. 
allocLocal() (1)

- To allocate a new local variable in a frame \( f \), the translation phase calls \( f\.allocLocal(false) \).

- The \texttt{false} argument indicates that the local variable does not escape, and \texttt{InReg(\( t_{481} \))} may be returned.

- If the argument is \texttt{true}, \texttt{InFrame(-8)} may be returned where -8 is the offset from the FP.
allocLocal() (2)

- Will all calls to allocLocal() come immediately after the frame is created?

```c
void f() {  
  int v=6;
  print(v);
  {
    int v=7;
    print(v);
  }
  print(v);
  {
    int v=8;
    print(v);
  }
  print(v);
}
```

- Variable-declaration blocks may be nested inside the body of a function.

- There is a distinct temporary or frame slot for every variable declared.

- The register allocator will determine that the second and third v variables can be held in the same register.
Temporaries and Labels

- During translation, we want to choose registers to hold variables and machine-code addresses for the procedure bodies.

- However, it is too early to determine which registers are available or where a procedure will be located.

- Temporaries (Temp) denote registers to be determined later.

- Labels (Label) denote a location to be determined.