Procedure and Function Calls,
— Part II —

Comp 412
Linkage Convention: High-Level View

The linkage must:

• Preserve & protect the caller’s environment from callee (& its callees)
  – For example, it must preserve values in registers that are live across the call
• Create a new environment for the callee (new name space)
  – At runtime, it creates a local data area for the callee & ties it to the context
• Map names & resources from caller into the callee, as needed

To accomplish these goals:

• Convention divides responsibility between caller & callee
• Caller & callee must use the same set of conventions
  – Implies limited opportunities for customization & optimization

We expect compiled code to work even when created with different compilers, perhaps in different languages.
Linkage Convention

A procedure or function has two distinct representations

**Code to implement the procedure**
- Code for all the statements
- **Code to implement the linkage convention**
- Labels on **entry points** and **return points**
  - Use **name mangling** to create consistent labels

**Data to represent an invocation**
- Activation record (**AR**)
- Local data + saved context of caller
- Control information
- Storage to preserve caller’s env’t
  - Chain **ARs** together to create context
- Static data area (**name mangling**)
The Meta Issue

Design time, compile time, and run time

• At compile time, the compiler emits code for the application program, including all of its procedures (and their prologs, epilogs, & sequences)
  – The compiler uses symbol tables to keep track of names
  – Those tables are *compile-time* data structures

• At runtime, those sequences (prolog, epilog, pre-call, & post-return) create, populate, use, and destroy ARs for invocations of procedures
  – The running code stores data (both program data and control data) in the ARs
  – Those ARs are *run-time* data structures

• In an OOL, the running code may need information stored in various object records (ORs) to locate the code pointer for the callee
  – The ORs are laid out at compile time; built, populated, & used at runtime

All of these interactions are planned at design time.
The code for all of them is emitted at runtime.
Each procedure has

• A standard **prolog**
• A standard **epilog**

Each call uses

• A **pre-call** sequence
• A **post-return** sequence

The code for the pre-call and post-return sequences must be completely predictable from the call site. It depends on the number & the types of the actual parameters.

**Note:** The gap between pre-call & post-return code is artistic license.
Activation Record Basics

In most systems, Activation Records have a similar layout:

- **space for parameters**: Space for parameters to the current procedure.
- **register save area**: Contents of saved registers.
- **slot for return value**: If function, space for return value.
- **slot for return address**: Address to resume caller.
- **slot for addressability**: Help with non-local access.
- **slot for caller’s ARP**: To restore caller’s ARP on return.
- **space for local variables**: Space for local variables, temporaries, & spills.

One AR for each invocation of a procedure
One register dedicated to hold the current ARP

ARP ≅ Activation Record Pointer
Procedure Linkages

**Pre-call Sequence**
- Sets up callee’s basic AR
- Helps preserve its own environment

**The Details**
- Allocates space for the callee’s AR
  - *except space for local variables*
- Evaluates each parameter & stores value or address
- Saves return address & caller’s ARP into callee’s AR
- Addressability maintenance, if necessary
- Saves any caller-save registers that are live across the call
  - *Save into space in caller’s AR*
- Copies callee AR into ARP & jumps to callee’s prolog code

Where do parameter values reside?
Conventional wisdom says:
- In registers (First 3 or 4)
- In callee’s AR (the rest)

Pre-call sequence is a lot of work
Procedure Linkages

Post-return Sequence

• Finishes restoring caller’s environment
• Places any value back where it belongs
  – e.g., any reference parameter or global that is kept in a register in the callee

The Details

• Copies return value from AR into a register, if needed
• Frees the callee’s AR
  – If access links are used, this action discards callee’s link
• Restores any caller-save registers
• Restores any call-by-reference parameters to registers, if needed
  – Also copy back call-by-value/result parameters
• Continues execution after the call
Procedure Linkages

Prolog Code

- Finishes setting up callee’s environment
- Preserves parts of caller’s environment that will be disturbed by the callee

The Details

- Preserves any callee-save registers
- Addressability maintenance, if necessary
- Allocates space for local data
  - *Easiest scenario is to extend the AR*
- Finds any static data areas referenced in the callee
- Handles any local variable initializations
  - *Not any cheaper to say “int x = 0;”*

With heap allocated AR, may need a separate heap object for local variables. (Must know size of local data area)
Procedure Linkages

Epilog Code
• Winds up the business of the callee
• Starts restoring the caller’s environment

The Details
• Stores return value?
  – Place it, or a pointer, in the return value slot of the caller’s AR\(^1\)
  – Other schemes are equally feasible
• Restore callee-save registers, as needed
• Addressability maintenance, if necessary
• Free space for local data, if necessary (on the heap)
• Restore return address & caller’s ARP from AR
• Jump to the return address

Takeaway point: procedure call overhead is many cycles.

\(^1\) Return value needs storage that survives the call
Where Do Saved Registers Go?

- Space for parameters to the current procedure
- Contents of saved registers
- If function, space for return value
- Address to resume caller
- Help with non-local access
- To restore caller’s AR on return
- Space for local variables, temporaries, & spills

Review

If \( p \) always saves registers into its own AR, then the compiler knows how much space to reserve in the AR for those purposes — \( p \)’s own caller saves registers (in prolog), \( p \)’s own callee saves registers (in pre-call when \( p \) calls \( q \)), and \( p \)’s own spills in the register allocator run on \( p \). (prolog patched after allocation)
Where Do Parameters Go?

- **space for parameters**
  - register save area
  - slot for return value
  - slot for return address
  - slot for addressability
  - slot for caller’s ARP

- **space for local variables**

- **Space for parameters to the current procedure**
  - Contents of saved registers
  - If function, space for return value
  - Address to resume caller
  - Help with non-local access
  - To restore caller’s AR on return

- **space for parameters**

- **Callee’s activation record**

Parameters from caller go here

Parameters passed to another call go here
How Do We Address Variables?

Local variables
• Need a mechanism to locate the local data area in appropriate AR
• Represent the variable as a static coordinate: $<\text{level}, \text{offset}>$
  – Level is the lexical nesting level of the procedure that declares the variable
  – Offset is the offset within the AR’s local data area for that procedure
• Mechanism takes static coordinate to run-time address

Static variables (including global)
• Need a base-address, offset pair
  – Emit a load of the base address
  – Emit an add of offset and a load (loadA if offset is small enough)
• Must be recognizable as non-local case (represent differently)

Dynamic variables
• Programmer manages access through pointers, names, ...
Addressability for Variables Stored in an AR

**Local variables of some procedure**
- Convert to static coordinate ($<\text{lexical level}, \text{offset}>$ pair)

**Local variable of current procedure**  
$(\text{level} = \text{current level})$
- Add $\text{offset}$ to $r_{\text{ARP}}$ and load

**Local variable in a surrounding lexical scope**  
$(\text{level} < \text{current level})$
- Convert $\text{level}$ to $\text{ARP}$ for procedure at that level
- Add $\text{offset}$ to that $\text{ARP}$ and load

**What if $\text{level} > \text{current level}$?**
- Should never happen

Local variables of surrounding scopes, static variables at various scopes, and global variables are often called “free variables” in the PL literature.
Addressability for Variables Stored in an AR

Each AR has a field for addressability

One common scheme uses access links

- Each AR has a link to the AR of its lexical ancestor (surrounding scope)
- Lexical ancestor need not be the procedure that called it
- Compiler can use static coordinate to generate code that chases the chain of access links to find the correct AR
  - Reference to \(<p, 16>\) runs up chain to find level \(p\)’s ARP, then adds 16
  - Number of derefences is \((c - p)\), where \(c\) is the current lexical level
- Cost of access is proportional to lexical distance between \(c\) and \(p\)

\(r_{\text{ARP}}\) is a physical register, dedicated to holding the ARP
Using Access Links

Finding the Correct AR

```
procedure fee {
    procedure fie {
        procedure foe {
            call foe()
        }
        call fie()
    }
    call fie()
}
```

<table>
<thead>
<tr>
<th>Static Coord</th>
<th>Generated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;3, 8&gt;</td>
<td>loadAI r_{ARP}, 8 =&gt; r_{10}</td>
</tr>
<tr>
<td>&lt;2,12&gt;</td>
<td>loadAI r_{ARP}, 4 =&gt; r_{10}</td>
</tr>
<tr>
<td></td>
<td>loadAI r_{ARP}, 12 =&gt; r_{10}</td>
</tr>
<tr>
<td>&lt;1,16&gt;</td>
<td>loadAI r_{ARP}, 4 =&gt; r_{10}</td>
</tr>
<tr>
<td></td>
<td>loadAI r_{ARP}, 12 =&gt; r_{10}</td>
</tr>
</tbody>
</table>
Finding the Correct AR

Maintaining Access Links

At each call:

• In pre-call sequence, caller must find the appropriate ARP

• Assume caller is at level $k$
  – Callee at level $k+1$
    → Use current ARP as link
  – Callee at level $j$, $0 < j \leq k$
    → Walk up access link chain to level $j-1$
    → Use that ARP as link
  – Callee is global (e.g., at level 0)
    → Use NULL as link
    → Caller & callee share no lexical context

• As long as the code is correct
  – Caller’s chain will be long enough
  – Callee cannot walk off end of chain
Finding the Correct AR

Each AR has a field for addressability

The Linkage Can Also Use A Display

- Global vector of pointers to the ARs of surrounding procedures
- Any nameable AR can be reached in constant time
  - Current AR is reached through ARP
  - Surrounding scopes are reached by treating display as a vector
- Reference to \( <p, 16> \), for \( p < k \), looks up the level \( p \) ARP in the display and adds 16 to it
- Cost of access is constant
Finding the Correct AR

```
procedure fee {
    procedure fie {
        procedure foe {
            call foe()
        }
        call fie()
    }
}
```

<table>
<thead>
<tr>
<th>Static Coord</th>
<th>Generated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;3, 8&gt;</td>
<td>loadAI r_{ARP}, 8 =&gt; r_{10}</td>
</tr>
</tbody>
</table>
| <2,12>       | loadAI _display_ => r_{10}  
                loadAI r_{10}, 4  => r_{10}  
                loadAI r_{10}, 12 => r_{10}  |
| <1,16>       | loadI _display_ => r_{10}  
                load r_{10}    => r_{10}  
                loadAI r_{ARP}, 12 => r_{10} |

_COMP 412, Fall 2019_
Finding the Correct AR

Maintaining A Display

• Display maintenance is simple, as long as the compiler writer is willing to “waste” one display slot

• In prolog of each level $j$ procedure
  – Save the level $j$ display entry into the procedure’s own AR (addressability slot)
  – Store current ARP into level $j$ display slot

• In epilog of each level $j$ procedure
  – Restore old level $j$ display entry from the current AR’s addressability slot

Simple, clean, and constant time
Access Links versus Display

**Access Links**

- Maintenance adds overhead to each call
  - Overhead depends on nesting
- Adds overhead on each reference to a local in surrounding scope
  - Overhead depends on nesting
- If ARs outlive procedure, access links still work
  - Access links between ARs ensures that they are live (& uncollected)

**Display**

- Maintenance adds overhead to each call & return
  - Overhead is constant
- Adds overhead on each reference to a local in surrounding scope
  - Overhead is constant
- If ARs outlive procedure, display links are not preserved
  - Need to preserve them in a way that ensures ARs are live (collector issue)

**Your mileage will vary**

- Depends on ratio of non-local accesses to calls
- Depends on demand for registers

*(available register to hold _display_ ?)*
Access Links and Displays

The Meta Issue

- Decision to use one or the other is made when the linkage is designed
  - May be beyond the control of the compiler writer

- Lexical levels for procedures and variables are determined in parsing
  - Requires the use of a lexically scoped symbol table

- Code to maintain access link or display is emitted at compile time
  - When compiler generates a procedure or a call, it emits the code
    - **Access links:** Lexical levels are known at compile time, so compiler can generate the code to find a level $j$ display (*knows how far to run up the chain*)
    - **Display:** Lexical levels are known at compile time, so compiler can generate the save (in prolog) and restore (in epilog)

- Either an access link or a display is a runtime object
  - Exists at runtime,
  - Points to ARs, which are runtime objects
Access to Parameter Arrays

Array-valued parameters pose a problem
Called procedure can receive different sized arrays from different calls
• Called procedure must work with any array it is passed

The Solution: Pass the array as a descriptor — a dope vector
• Encapsulate dimension information in a vector
• Pass the vector’s address to callee

Same approach works for dynamically-allocated arrays
• Create a dope vector at allocation time
• Use dimensions from dope vector in the address polynomial

Accesses to parameter arrays and variable-sized arrays will be slightly slower than accesses to a fixed-size, automatic or static array. (violates principle of least astonishment)
Access to Parameter Arrays

Details of the Dope Vector Mechanism

• Compiler allocates space for dope vector in local data area
• Pre-call builds the dope vector & passes its address
• Callee generates code assuming dope vector
• Compiler can store parameters for naïve polynomial, or for the optimized polynomial
  – Must decide, up front, the code shape issue for all calls & call sites

The dope vector presents opportunity for optimizer

• Common subexpressions in address polynomials
• Contents of dope vector are fixed between allocations
  – In many situations, they are provably fixed across the entire invocation
• Good optimizer can make these accesses efficient