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

foe must operate in multiple distinct calling contexts.
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)
One of the most difficult aspects of understanding how compilers work is keeping track of what happens at compiler-design time, compiler-build 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, epilog, & 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

To confuse matters further, the compiler may preserve the symbol tables so that the runtime debugging environment can locate variables and values, and locate the various data areas and the ARs
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 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

One AR for each invocation of a procedure
One register dedicated to hold the current ARP
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*
- Jumps to address of callee’s prolog code

**Pre-call sequence is a lot of work**

COMP 412, Fall 2017

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

---

\(^1\) Return value needs storage that survives the call

**Takeaway point:** procedure calls have many cycles of overhead.
Where Do Saved Registers Go?

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 \textit{prolog}), \( p \)'s own callee saves registers (in \textit{pre-call} when \( p \) calls \( q \)), and \( p \)'s own spills in the register allocator run on \( p \). (\textit{prolog patched after allocation})
### Where Do Parameters Go?

<table>
<thead>
<tr>
<th>Parameters from callee go here</th>
<th>Parameters passed to another call go here</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>space for parameters</strong></td>
<td><strong>space for parameters</strong></td>
</tr>
<tr>
<td>register save area</td>
<td></td>
</tr>
<tr>
<td>slot for return value</td>
<td></td>
</tr>
<tr>
<td>slot for return address</td>
<td></td>
</tr>
<tr>
<td>slot for addressability</td>
<td></td>
</tr>
<tr>
<td>slot for caller’s ARP</td>
<td></td>
</tr>
<tr>
<td><strong>space for local variables</strong></td>
<td></td>
</tr>
</tbody>
</table>

- 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

---

Callee’s activation record
Creating and Destroying Activation Records

The activation record embodies the runtime state of the procedure

• How are ARs created and destroyed?
  – Procedure call must allocate & initialize (preserve caller’s world)
  – Return must dismantle environment (and restore caller’s world)

• Caller & callee must collaborate on the problems
  – Caller alone knows some of the necessary state
    → Return address, parameter values, access to other scopes
  – Callee alone knows the rest
    → Size of local data area (with spills), registers it will use

Their collaboration takes the form of a linkage convention

Assume, for the moment, an Algol-60-like environment where the AR is dead after the return.

1. Linkage convention is defined at compiler design time
2. Linkage code is emitted at compile time
3. Linkage code executes at runtime, when a procedure is called.
Storing Activation Records

If activation records are stored on the stack

• Easy to extend — simply bump top of stack pointer
• Caller & callee share responsibility
  – Caller can push parameters, space for registers, return value slot, return address, addressability info, & its own ARP
  – Callee can push space for local variables (fixed & variable size)
Address Space Layout

Most language runtimes layout the address space in a similar way:

- Pieces (stack, heap, code, & globals) may move, but all will be there.
- Stack and heap grow toward each other (if heap grows).

Java Memory Layout:

- LIFO Runtime Stack

New ARs

COMP 412, Fall 2017
Storing Activation Records

If activation records are stored on the stack

• Easy to extend — simply bump top of stack pointer
• Caller & callee share responsibility
  — Caller can push parameters, space for registers, return value slot, return address, addressability info, & its own ARP
  — Callee can push space for local variables (fixed & variable size)

If activation records are stored on the heap

• Hard to extend
• Several options
  — Caller passes everything in registers; callee allocates & fills AR
  — Store parameters, return address, etc., in caller’s AR!
  — Store callee’s AR size in a defined static constant
Most language runtimes layout the address space in a similar way

- Pieces (stack, heap, code, & globals) may move, but all will be there
- Stack and heap grow toward each other (if heap grows)
Storing Activation Records

If activation records are stored on the stack

• Easy to extend — simply bump top of stack pointer
• Caller & callee share responsibility
  – Caller can push parameters, space for registers, return value slot, return address, addressability info, & its own ARP
  – Callee can push space for local variables (fixed & variable size)

If activation records are stored on the heap

• Hard to extend
• Several options
  – Caller passes everything in registers; callee allocates & fills AR
  – Store parameters, return address, etc., in caller’s AR!
  – Store callee’s AR size in a defined static constant

Without recursion, activation records can be static
Address Space Layout

Most language runtimes layout the address space in a similar way

- Pieces (stack, heap, code, & globals) may move, but all will be there
- Stack and heap grow toward each other (if heap grows)

Statically allocated ARs go into the “Global” data area — which we would need to redraw in as a larger area ...
Where Do Variables Live?

We have seen ARs, static data areas, global data areas, ...

How does the compiler decide where to place each variable

• A combination of visibility and lifetime determines that placement

<table>
<thead>
<tr>
<th>Lifetime</th>
<th>Scope</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>automatic</td>
<td>local</td>
<td>declaring procedure’s AR</td>
</tr>
<tr>
<td>static</td>
<td>any</td>
<td>named static data area for its lifetime</td>
</tr>
<tr>
<td>dynamic</td>
<td>any</td>
<td>heap</td>
</tr>
</tbody>
</table>

Variable length items?

• Put a descriptor in the natural place & allocate space in AR or in the heap
  - Requires one level of indirection (a “pointer”)
  - Allows uniform addressability
  - In AR if AR is extendible
  - On heap if AR is not extendible

COMP 412, Fall 2017
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: \(<level, offset>\)
  - *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 \((mangled\ name\ for\ base\ address)\)
  - Generate a load of the base address
  - Add the offset and load the value \((loadA!\ if\ offset\ is\ small\ enough)\)
- Must be recognizable as non-local case \((represent\ differently)\)

**Dynamic variables**
- Programmer manages access through pointers, names, ...
How Do We Address Variables?

**Local variable of current procedure**
- Convert to static coordinate
- Add offset to \( r_{ARP} \) and load

**Local variable in surrounding lexical scope**
- Convert to static coordinate
- Convert \( level \) to \( ARP \) for procedure at that level
- Add offset to that \( ARP \) and load

Local variables of surrounding scopes, static variables at various scopes, and global variables are often called “free variables” in the PL literature.
Finding the Correct 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 dereferences 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

COMP 412, Fall 2017
Using Access Links

Finding the Correct AR

\[ \text{parameters} \]
\[ \text{registers} \]
\[ \text{return value} \]
\[ \text{return address} \]
\[ \text{caller’s ARP} \]
\[ \text{addressability} \]
\[ \text{local variables} \]

\[ \text{parameters} \]
\[ \text{registers} \]
\[ \text{return value} \]
\[ \text{return address} \]
\[ \text{caller’s ARP} \]
\[ \text{addressability} \]
\[ \text{local variables} \]

\[ \text{parameters} \]
\[ \text{registers} \]
\[ \text{return value} \]
\[ \text{return address} \]
\[ \text{caller’s ARP} \]
\[ \text{addressability} \]
\[ \text{local variables} \]

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_{\text{ARP}}, 8 ) ( \rightarrow r_{10} )</td>
</tr>
</tbody>
</table>
| <2,12>       | loadAI \( r_{\text{ARP}}, 4 \) \( \rightarrow r_{10} \)  
|              | loadAI \( r_{\text{ARP}}, 12 \) \( \rightarrow r_{10} \) |
| <1,16>       | loadAI \( r_{\text{ARP}}, 4 \) \( \rightarrow r_{10} \)  
|              | loadAI \( r_{\text{ARP}}, 4 \) \( \rightarrow r_{10} \)  
|              | loadAI \( r_{\text{ARP}}, 12 \) \( \rightarrow r_{10} \) |
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 $k$
  - Call at level $k+1$  
    - Use current ARP as link
  - Call at level $j$, $0 < j \leq k$
    - Walk up access link chain to level $j-1$
    - Use that ARP as link
  - Call 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 \(p \text{ ARP}\) in the display and adds 16 to it
- Cost of access is constant
Finding the Correct AR

Using A Display

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