Runtime Environments I

Basilio B. Fraguela
Runtime System Responsibilities

• Allocation of storage for program data
• Sometimes also deallocation
  – Garbage collection
• Management of data structures the compiled program uses to access data
Data Storage Allocation

• Mostly dynamic
  – Some things can be done statically

• Main dynamic allocation possibilities:
  – Stack: data that do not outlive the procedure where it is declared
  – Heap: remaining data
From Names to Values

• *Environment*: mapping from (variable) names to store addresses
• *State*: mapping from addresses to their values
• Both mappings can be static or dynamic
• Declaration vs Definition
  – *Declarations* specify types, interfaces
  – *Definitions* provide values, implementations
Scope of a Declaration

• The *scope of a declaration* of $x$ is the context in which uses of $x$ refer to this declaration
  – *Static or lexical* scope: identifiable from program source code
    • Usually relies on code block structure
  – *Dynamic* scope: otherwise
Static Scope based on Block Structure
Explicit Access Control

• Specify who can access fields in a record/class

• Accessible members are usable in any subclass unless it redefines the member

• Encapsulation through explicit control using keywords
  – E.g: C++’s public, private and protected
Dynamic Scope

- Name resolution depends on most recently called function, without there being a redeclaration
  - Macro expansion
  - Polymorphic methods
Stack Management

• Used for data whose usage is restricted to the procedure where it is declared or procedures it calls
• Each time a procedure is invoked, it reserves its space at the top of the stack
• When the procedure returns, its space is popped off the stack

The stack structure is adequate because procedure calls (activations) nest in time
  – If $p$ calls $q$, $q$ must finish before $p$ can call another procedure or finish
Activation Tree

• Represents the activation of procedures during the execution of the program
• Activations happen top-to-bottom and left-to-right (preorder traversal)
• Returns happen bottom-to-top and left-to-right (postorder traversal)
• The activations open when control reaches a node of the tree are its ancestors
Activation Records (or Frames)

- Stack space allocated for an activation of a procedure in the (control) stack
- Traditional representation:

```
Root Activation Record
Proc1 Activation Record
Proc2 Activation Record
```

Bottom of the stack

Top of the stack
What’s Inside an Activation Record

- Actual parameters
- Returned values
- Control link
- Access link
- Saved machine status
- Local data
- Temporaries
Procedure Call/Return Implementation

• The *calling sequence* fills in the activation record of the procedure called
• The *return sequence* restore the state upon return using the activation record
• Both can be a shared responsibility of the calling procedure (*caller*) and the called procedure (*callee*)
  – In general it is better the callee does most work
Usual Disposition of Data in the Activation Record

- Values communicated between caller and callee are usually put at the beginning
- Fields with a fixed width come in the middle
- Variable length data usually go at the end (top) of the record
  - Dynamically sized arrays
- Top-of-stack pointer usually points to the end of the fixed-length fields
Example
Parameter Passing Mechanisms

• Association of formal and actual parameters policies:
  – Call-by-value
  – Call-by-reference
  – Call-by-name (obsolete)
Access to non-local data

• Simple for languages that do not allow nested procedure declarations:
  – Variables are either local to the procedure
  – Or globally/statically declared

• In languages with nested procedures
  – the declaration for a non-local name can be found statically, but
  – dynamic mechanisms are needed to find the relevant activation record of the caller that contains the data
Access Link

• Points to most recent activation record for immediately enclosing function.
• Access links form a chain from the current (highest) nesting level activation record to the lowest one
• All accessible activation records are in the chain
  – $N$ hops to reach activation record with nesting depth current-$N$
Access Link Calculation

• When $q$ calls $p$:
  – $\text{depth } p > \text{depth } q \Rightarrow q$ immediately encloses $p$
    • Access link for $p$ points to $q$’s activation record
  – $\text{depth } p = \text{depth } q \Rightarrow$ (mutually) recursive call
    • Access link for $p = \text{Access link for } q$
  – $\text{depth } p > \text{depth } q \Rightarrow$ both are nested inside a common procedure $r$
    • $\text{depth}(p) – \text{depth}(q)$ hops to find access link for $p$
Access Link Illustration

(a) \[ s \]
\[ \text{access link} \]
\[ a \]
\[ q(1,9) \]
\[ \text{access link} \]
\[ v \]

(b) \[ s \]
\[ \text{access link} \]
\[ a \]
\[ q(1,9) \]
\[ \text{access link} \]
\[ v \]
\[ q(1,3) \]
\[ \text{access link} \]
\[ v \]

(c) \[ s \]
\[ \text{access link} \]
\[ a \]
\[ q(1,9) \]
\[ \text{access link} \]
\[ v \]
\[ q(1,3) \]
\[ \text{access link} \]
\[ v \]
\[ p(1,3) \]
\[ \text{access link} \]

(d) \[ s \]
\[ \text{access link} \]
\[ a \]
\[ q(1,9) \]
\[ \text{access link} \]
\[ v \]
\[ q(1,3) \]
\[ \text{access link} \]
\[ v \]
\[ p(1,3) \]
\[ \text{access link} \]
\[ e(1,3) \]
\[ \text{access link} \]
Display

• Array $d$ with one pointer per nesting depth
  – $d[i]$ points to highest activation record for any
    procedure at nesting level $i$
• Any variable defined in a procedure at
  nesting level $i$, can be found through $d[i]$
  – No need to follow a chain of access links
• When a procedure overwrites $d[i]$, it must
  first save it, then restore it when it returns
Example of Display Evolution

Function s (depth 1)

Function e (depth 2)

Function q (depth 2)

Function p (depth 3)