Compiler Design

Procedure Abstraction

Hwansoo Han
Control Abstraction

✦ Procedures have well-defined control-flow

✦ The Algol-60 procedure call
  - Invoked at a call site, with some set of *actual parameters*
  - Control returns to call site, immediately after invocation

```c
int p(a,b,c)
  int a, b, c;
  {
    int  d;
    d = q(c,b);
    ...
  }
int q(x,y)
  int x,y;
  {
    return x + y;
    }

... s = p(10,t,u);
...
...
...
```

✦ Most languages allow recursion
Requirement for Control Abstraction

- **Implementing procedures with this behavior**
  - Requires code to save and restore a “return address”
  - Must map actual parameters to formal parameters
  - Must create storage for local variables (maybe, for parameters)
    - $p$ needs space for $d$ (maybe, for $a$, $b$, & $c$)
    - where does this space go in recursive invocations?
  - Must preserve $p$'s state while $q$ executes
    - recursion causes the real problem here
  - **Strategy**: Create unique location for each procedure activation
    - Can use a “stack” of memory blocks to hold local storage and return addresses
Name Space

- **Each procedure creates its own name space**
  - Any name (almost) can be declared locally
  - Local names obscure identical non-local names
  - Local names cannot be seen outside the procedure
    - Nested procedures are “inside” by definition
  - We call this set of rules & conventions “lexical scoping”

- **Examples**
  - C has global, static, local, and block scopes
    - Blocks can be nested, procedures cannot
  - Java has private, protected, public, package scopes
    - Blocks can be nested, methods can be nested
Lexical Scopes

- **Lexical scoping**
  - Provides a compile-time mechanism to use *static coordinate* for each variable name
  - Simplifies rules for naming & resolves conflicts
    - *Each name refers to its lexically closest declaration*

- **Keeping track of all the names**
  - At a program point, need to determine which declaration of $x$ is current
  - At run-time, where can $x$ be found? (addressability)
  - Lexically scoped symbol tables allow lexical scoping
Example

procedure p {
    int a, b, c
    procedure q {
        int v, b, x, w
        procedure r {
            int y, z
            ... = x
        }
        procedure s {
            int x, a, v
            ... r()
        }
        ... r()
        s()
    }
    ... r()
    s()
}
... q() ...

B0: {
    int a, b, c
    B1: {
        int v, b, x, w
        B2: {
            int y, z
            ... = x
        }
        B3: {
            int x, a, v
            ... } ... = x
    }
    ... {
        ... ...
    }
    ...
    } ...
Lexically-Scoped Symbol Tables

- **High-level idea**
  - Create a new table for each scope (hash table for each scope)
  - Chain them together for lookup

```
\begin{align*}
\text{insert}(v) & \text{ may need to create table} \\
& \text{it always inserts } v \text{ at current level} \\
\text{lookup}(v) & \text{ walks chain of tables & returns first occurrence of name } v \\
\text{delete()} & \text{ throws away table for current level, if it is top table in the chain} \\
\text{If the compiler must preserve the table, table can be stored in file} \\
\text{Individual tables can be hash tables}
\end{align*}
```
Where Do All Variables Go?

- **Automatic & Local**
  - Keep them in the procedure activation record or in a register
  - Automatic $\Rightarrow$ lifetime matches procedure’s lifetime

- **Static**
  - Procedure scope $\Rightarrow$ storage area affixed with procedure name
    - &_p.x
  - File scope $\Rightarrow$ storage area affixed with file name
  - Lifetime is entire execution

- **Global**
  - One or more named global data areas
  - Lifetime is entire execution
Placing Run-time Data Structures

- **Classic Organization**

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack &amp; Global</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
</table>

- **Better utilization if stack & heap grow toward each other**
- **Uses address space, not allocated memory**

- **Code, static, & global data have known size**
  - Use symbolic labels, not numeric address, in the code

- **Heap & stack both grow & shrink over time**

- **This is a virtual address space**
How Does This Really Work?

The Big Picture

Compiler’s view

OS’s view

Hardware’s view

virtual address spaces

Physical address space

Hardware's view

Compiler’s view

OS’s view
Translating Local Names

- **How does the compiler represent a specific instance of \( x \)?**
  - Name is translated into a static coordinate
    - \( < \text{level}, \text{offset} > \) pair
    - "level" is lexical nesting level of the procedure
    - "offset" is unique within that scope
  - Subsequent code will use the static coordinate to generate addresses and references
**Activation Record**

- **Activation record**
  - One AR per procedure
  - ARP (activation record pointer) points current AR

- **How does the compiler find the local variables?**
  - They are at known offsets from the AR pointer
  - The static coordinate leads to a “loadAI (base + immediate offset)”
    - Level specifies an ARP, offset is the constant

![Diagram showing activation record components](image)
Addressability

- **Nested lexical scoping**
  - Lexically closest declaration

- **Access links**
  - If not found in current AR, walk the chain of access links to search next closest one

- **Global display**
  - "Display" contains ARP of the most recent procedure at each lexical level
  - Constant cost to access non-locals

```c
program main {
    int x;            // def1
    procedure r {
        ... = x;      // which x?
    }
    procedure s {
        int x;        // def2
        r();
        ... = x;      // which x?
    }
    r();
    s();
}
```
IA32/Linux Stack Frame

- **Current Stack Frame**
  - Parameters for function about to call
    - “Argument build”
  - Local variables
    - If can’t keep in registers
  - Saved register context
  - Old frame pointer

- **Caller Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call
**IA32/Linux Register Usage**

- **Integer Registers**
  - Two have special uses:
    - %ebp, %esp
  - Three managed as callee-save:
    - %ebx, %esi, %edi
    - Old values saved on stack prior to using them
  - Three managed as caller-save:
    - %eax, %edx, %ecx
    - Do what you please, but expect any callee to do so, as well
  - Register %eax also stores returned value
Communicating Between Procedures

- **Most languages provide a parameter passing mechanism**
  - Expression used at “call site” becomes a variable in callee

- **Common binding mechanisms for parameter passing**
  - **Call-by-value** passes a copy of its value at time of call
    - Requires slot in the AR (or a register)
    - One name (formal parameter) refers to that value
  - **Call-by-reference** passes a pointer to actual parameter
    - Requires slot in the AR (or a register)
    - Expression in actual parameter is evaluated and stored in AR to pass its address
    - Aliases may exist – other names may refer to the same location
  - **Call-by-result** passes the value of a formal parameter to the corresponding actual parameter at time of return
    - Arguments are used for results, not input parameters
Procedure Linkages - calling convention

- **Calling convention**
  - Parameter passing
    - Call-by-reference, -value, -result, -name
  - Stack clean-up
    - Caller vs. callee (cdecl vs. stdcall)

- **How do procedure calls actually work?**
  - At compile time, callee may not be available for inspection
    - Different calls may be in different compilation units
    - Compiler may not know system code from user code
    - All calls must use the *same protocol*

- **Compiler must use a standard sequence of operations**
  - Enforces control & data abstractions
  - Divides responsibility between caller & callee

- **Usually a system-wide agreement** *(for interoperability)*
Procedure Linkages Overview

- **Standard procedure linkage**

  Procedure has
  - standard **prolog**
  - standard **epilog**

  Each call involves a
  - **pre-call** sequence
  - **post-return** sequence

  These are completely predictable from the call site ⇒ depend on the number & type of the actual parameters
Summary

- **Views for procedure**
  - Control abstraction, name space, interface

- **Single address space for each program**
  - MMU and OS provide virtual address space
  - Compiler sees only single address space

- **Procedure linkage (calling convention)**
  - Provides system-wide interoperability
  - One activation record for each procedure
  - Caller and callee share responsibility to preserve/build/restore execution environment