Run-time Environment

Roadmap (Where are we?)

Last lecture
- Code generation
  - → Arrays
  - → Boolean and relational values
  - → Control flow

This lecture
- Run-time environment
  - → Procedure abstraction
  - → Activation records
  - → Addressability
  - → Procedure linkages

Procedure Abstraction

- Chapter 6 in EAC
- The compiler must deal with interface between compile time and run time (static versus dynamic)
  - → Most of the tricky issues arise in implementing "procedures"
- Issues
  - → Compile-time versus run-time behavior
  - → Finding storage for EVERYTHING, and mapping names to addresses
  - → Generating code to compute addresses that the compiler cannot know !
  - → Interfaces with other programs, other languages, and the OS
  - → Efficiency of implementation

The Procedure

Procedures allow us to use separate compilation
- Separate compilation allows us to build non-trivial programs
- Keeps compile times reasonable
- Lets multiple programmers collaborate
- Requires independent procedures

Without separate compilation, we *would not* build large systems

The procedure linkage convention
- Ensures that each procedure inherits a valid run-time environment and that the callers environment is restored on return
  - → The compiler must generate code to ensure this happens according to conventions established by the system

The Procedure (More Abstract View)

A procedure is an abstract structure constructed via software

Underlying hardware directly supports little of the abstraction—it understands bits, bytes, integers, reals, and addresses, but not:
- Entries and exits
- Interfaces
- Call and return mechanisms
  - → may be a special instruction to save context at point of call
- Name space
- Nested scopes

All these are established by a carefully-crafted system of mechanisms provided by compiler, run-time system, linkage editor and loader, and OS

Run Time versus Compile Time

These concepts are often confusing to the newcomer
- Procedure linkages execute at run time
- Code for the procedure linkage is emitted at compile time
- The procedure linkage is designed long before either of these

"This issue (compile time versus run time) confuses students more than *any other* issue" —Keith Cooper (Rice University)

The Procedure as a 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

```
...                int p(a,b,c)
s = p(10,t,u);       int a, b, c;          int q(x,y)
...                {                         int x,y;
                     int  d;               {
                     d = q(c,b);             return x + y;
                     ...                   }
                   }
```

- Most languages allow recursion

The Procedure as a Control Abstraction

Implementing procedures with this behavior
- Requires code to save and restore a "return address"
- Must map actual parameters to formal parameters   *(c→x, b→y)*
- Must create storage for local variables (&, maybe, parameters)
  - → *p* needs space for *d* (&, maybe, *a, b, & c*)
  - → where does this space go in recursive invocations?

```
...                int p(a,b,c)
s = p(10,t,u);       int a, b, c;          int q(x,y)
...                {                         int x,y;
                     int  d;               {
                     d = q(c,b);             return x + y;
                     ...                   }
                   }
```

Compiler emits code that causes all this to happen at run time

The Procedure as a Control Abstraction

Implementing procedures with this behavior
- 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

```
...                int p(a,b,c)
s = p(10,t,u);       int a, b, c;          int q(x,y)
...                {                         int x,y;
                     int  d;               {
                     d = q(c,b);             return x + y;
                     ...                   }
                   }
```

Compiler emits code that causes all this to happen at run time

The Procedure as a 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
- Different sets of rules & conventions: "lexical scoping" and "dynamic scoping".

Examples (lexical scoping)
- C has global, static, local, and *block* scopes   *(Fortran-like)*
  - → Blocks can be nested, procedures cannot
- Scheme has global, procedure-wide, and nested scopes *(let)*
  - → Procedure scope (typically) contains formal parameters

The Procedure as a Name Space

Why introduce lexical scoping?
- Provides a compile-time mechanism for binding "free" variables
- Simplifies rules for naming & resolves conflicts

How can the compiler keep track of all those names?

The Problem
- At point *p*, which declaration of *x* is current?
- At run-time, where is *x* found?
- As parser goes in & out of scopes, how does it delete *x*?

The Answer
- Lexically scoped symbol tables   (see § 5.7.3)

The Procedure as an External Interface

OS needs a way to start the program's execution
- Programmer needs a way to indicate where it begins
  - → The "main" procedure in most languages
- When user invokes "grep" at a command line   UNIX/Linux specific discussion
  - → OS finds the executable
  - → OS creates a process and arranges for it to run "grep"
  - → "grep" is code from the compiler, linked with run-time system
    - Starts the run-time environment & calls "main"
    - After main, it shuts down run-time environment & returns
- When "grep" needs system services
  - → It makes a system call, such as fopen()

Where Do All These Variables Go?

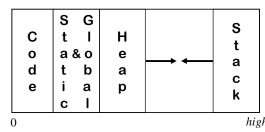
Automatic & Local
- Keep them in the procedure activation record or in a register
- Automatic ⇒ lifetime matches procedure's lifetime

Static
- Procedure scope ⇒ storage area affixed with procedure name
- File scope ⇒ 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



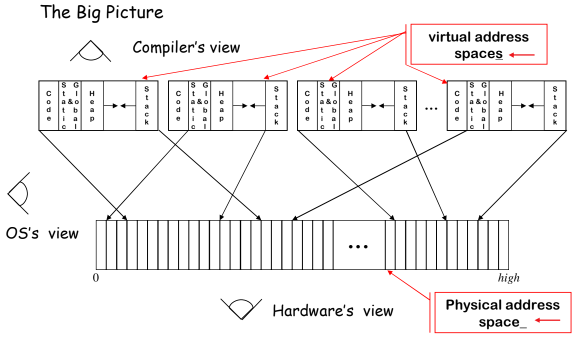
Single Logical Address Space

- Better utilization if stack & heap grow toward each other
- Very old result (Knuth)
- Code & data separate or interleaved

- Code, static, & global data have known size
  - Use symbolic labels in the code
- Heap & stack both grow & shrink over time
- This is a virtual address space

How Does This Really Work?

The Big Picture



Where Do Local Variables Live?

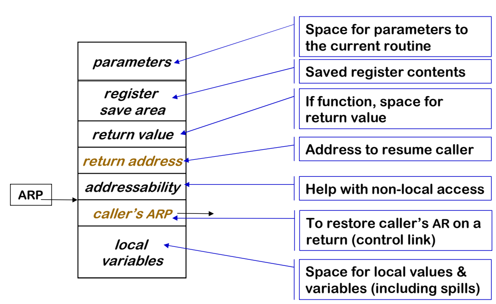
A Simplistic model
- Allocate a data area for each distinct scope
- One data area per nesting-level in scoped table

What about recursion?
- Need a data area per invocation (or activation) of a scope
- We call this the scope's activation record
- The compiler can also store control information there !

Translating Local Names

How does the compiler represent a specific instance of *x* ?
- Name is translated into a *static coordinate*
  - → ‹ *level,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
- *"level"* is a function of the table in which *x* is found
  - → Stored in the entry for each *x*
- *"offset"* must be assigned and stored in the symbol table
  - → Assigned at compile time
  - → Known at compile time
  - → Used to generate code that executes at run-time

Activation Record Basics



One **AR** for each invocation of a procedure

Activation Record Details

How does the compiler find the variables?

- They are at known offsets from the AR pointer
- The static coordinate leads to a "loadAI" operation
  - → Level specifies an ARP, offset is the constant

loadAI r1, c1 ⇒ r2 : MEM(r1 + c1) → r2

Variable-length data

- If AR can be extended, put it below local variables
- Leave a pointer at a known offset from ARP
- Otherwise, put variable-length data on the heap

Initializing local variables

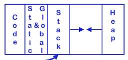
- Must generate explicit code to store the values
- Among the procedure's first actions

Activation Record Details

Where do activation records live?

- If lifetime of AR matches lifetime of invocation, *AND*
- If code normally executes a "return"

⇒ Keep ARs on a stack



- If a procedure can outlive its caller, *OR*
- If it can return an object that can reference its execution state

Yes! This stack.

⇒ ARs must be kept in the heap

- If a procedure makes no calls

⇒ AR can be allocated statically

Efficiency prefers static, stack, then heap

Communicating Between Procedures

Most languages provide a parameter passing mechanism

⇒ Expression used at "call site" becomes variable in callee

Two common binding mechanisms

- Call-by-reference passes a pointer to actual parameter
  - → Requires slot in the AR (for address of parameter)
  - → Multiple names with the same address?

`call fee(x,x,x);`

- Call-by-value passes a copy of its value at time of call
  - → Requires slot in the AR
  - → Each name gets a unique location
  - → Arrays are mostly passed by reference, not value
- Can always use global variables …

Establishing Addressability

Must create base addresses

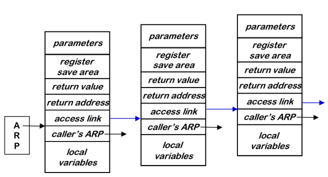
- Global & static variables
  - → Construct a label by mangling names (*i.e.*, &_fee)
- Local variables
  - → Convert to static data coordinate and use **ARP** + offset
- Local variables of other procedures
  - → Convert to static coordinates
  - → Find appropriate **ARP**
  - → Use that **ARP** + offset

$\left\{ \begin{array}{l} \textbf{Must find the right AR} \\ \textbf{Need links to nameable ARs} \end{array} \right.$

Establishing Addressability

Using access links (static links)

- Each AR has a pointer to AR of lexical ancestor
- Lexical ancestor need not be the caller



Some setup cost on each call

- Reference to ‹*p*,16› runs up access link chain to *p*
- Cost of access is proportional to lexical distance

Establishing Addressability

Using access links

SC	**Generated Code**
<2,8>	loadAI r_0, 8 ⇒ r_2
<1,12>	loadAI r_0, -4 ⇒ r_1 loadAI r_1, 12 ⇒ r_2
<0,16>	loadAI r_0, -4 ⇒ r_1 loadAI r_1, -4 ⇒ r_1 loadAI r_1, 16 ⇒ r_2

Assume

- Current lexical level is 2
- Access link is at **ARP** - 4

Maintaining access link

- Calling level *k*+1 (*k* is current level)

→ Use current **ARP** as link in new AR

- Calling level *j* ‹ *k*

→ Find ARP for *j* –1

→ Use that ARP as link in new AR

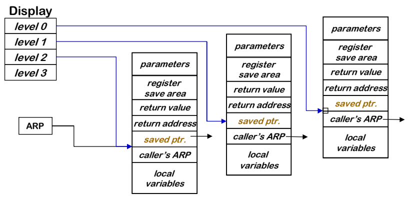
Access & maintenance cost varies with level

All accesses are relative to ARP (r_0)

Establishing Addressability

Using a display
- Global array of pointer to nameable ARs
- Needed ARP is an array access away

Some setup cost on each call



- Reference to <p,16> looks up *p's* ARP in display & adds 16
- Cost of access is constant (ARP + offset)

Establishing Addressability

Using a display

SC	Generated Code
<2,8>	loadAI r_0, 8 ⇒ r_2
<1,12>	loadI _disp ⇒ r_1 loadAI r_1, 4 ⇒ r_1 loadAI r_1, 12 ⇒ r_2
<0,16>	loadI _disp ⇒ r_1 loadAI r_1, 16 ⇒ r_2

Desired AR is at _disp + 4 x *level*

Assume
- Current lexical level is 2
- Display is at label _disp

Maintaining access link
- On entry to level *j*
  - → Save level *j* entry into AR (Saved Ptr field)
  - → Store ARP in level *j* slot
- On exit from level *j*
  - → Restore level *j* entry

Access & maintenance costs are fixed
Address of display may consume a register

Establishing Addressability

Access links versus Display
- Each adds some overhead to each call
- Access links costs vary with level of reference
  - → Overhead only incurred on references & calls
  - → If ARs outlive the procedure, access links still work
- Display costs are fixed for all references
  - → References & calls must load display address
  - → Typically, this requires a register

Your mileage will vary
- Depends on ratio of non-local accesses to calls
- Extra register can make a difference in overall speed

For either scheme to work, the compiler must insert code into each procedure call & return

Procedure Linkages

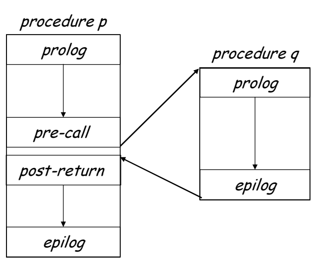
How do procedure calls actually work?
- At compile time, callee may not be available for inspection
  - → Different calls may be in different compilation units
  - → Compiler may not know system code from user code
  - → All calls must use the same protocol

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

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

Procedure Linkages

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

The Details
- Allocate space for the callee's AR
  - → except space for local variables
- Evaluates each parameter & stores value or address
- Saves return address, caller's ARP (control link) into callee's AR
- If access links are used
  - → Find appropriate lexical ancestor & copy into callee's AR
- Save any caller-save registers
  - → Save into space in caller's AR
- Jump to address of callee's prolog code

Procedure Linkages

Post-return Sequence

- Finish restoring caller's environment
- Place any value back where it belongs

The Details

- Copy return value from callee's AR, if necessary
- Free the callee's AR
- Restore any caller-save registers
- Copy back call-by-value/result parameters
- Continue execution after the call

Procedure Linkages

Prolog Code

- Finish setting up callee's environment
- Preserve parts of caller's environment that will be disturbed

The Details

- Preserve any callee-save registers
- If display is being used
  - → Save display entry for current lexical level
  - → Store current ARP into display for current lexical level
- Allocate space for local data
  - → Easiest scenario is to extend the AR
- Handle any local variable initializations

With heap allocated AR, may need to use a separate heap object for local variables

Procedure Linkages

Epilog Code

- Wind up the business of the callee
- Start restoring the caller's environment

The Details

- Store return value?
  - → Some implementations do this on the return statement
  - → Others have return assign it & epilog store it into caller's AR
- Restore callee-save registers
- Free space for local data, if necessary (on the heap)
- Load return address from AR
- Restore caller's ARP
- Jump to the return address

If ARs are stack allocated, this may not be necessary. (Caller can reset stacktop to its pre-call value.)