Implementing Subprograms

Semantics of Call and Return

- The call and return operations of a language are together called its subprogram linkage
- Actions involved in implementing call
  - Implement parameter passing method
  - Allocate local storage
  - Bind variables
  - Save execution status of calling program
    - Includes register values, processor flag/status bits, frame pointer
- Actions involved implementing return
  - Deallocate local storage
  - Communicate function return results if any
  - Restore execution environment and program counter

Flat and Nested Execution Scopes

- Subprogram call is further complicated by nested execution scopes
- Called functions need access to non-local variables
  - These can be static variables but more often are stack-dynamic local variables

“Simple” Subroutines

- Simple (like early Fortran):
  - Static locals
  - Not nested
- Call Semantics:
  - Save the execution status of the caller
  - Pass the parameters
  - Pass the return address to the callee
  - Transfer control to the callee

Simple Subroutine Return

- Return Semantics:
  - If pass-by-value-result or out mode parameters are used, copy the current values of those parameters to their corresponding actual parameters
  - If it is a function, copy the return value to a place the caller can get it
  - Restore the execution status of the caller
  - Transfer control back to the caller
- Required storage:
  - Status information, parameters, return address, return value for functions, local variables

Distribution of Responsibility

- Generally call semantics are executed by the caller and return semantics by callee
- One possible exception is saving the state of the caller
  - This can be done by either
    - Caller knows what needs to be saved
    - Callee knows what global resources (registers, etc) need to be saved
- Linkage actions of callee are called the “prologue” and the “epilogue”
  - For simple subroutines with static locals and without nested scope there is no need for prologue code
Subroutine Organization

- Subroutines have two separate parts: the code and the non-code part consisting of:
  - Data (local and temporary variables)
  - Linkage information (saved frame pointers, return addresses, etc.)
- The format, or layout, of the non-code part of an executing subprogram is called an activation record.
- An activation record instance is a concrete example of an activation record (the collection of data for a particular subprogram activation).

Activation Record for Simple Subs

- Simple subroutines can only have one activation instance per routine (no support for recursion)
  - Activation record is fixed in size and allocated statically
  - Could be actually attached to the code segment
  - Below is a possible layout

Linkage

- A program can be composed of multiple parts separately compiled
- The linker is responsible for assembling the overall program layout

Stack-Dynamic Local Variables

- Most languages (whether compiled or interpreted) use stack dynamic local variables
  - Provides support for recursive because each invocation of a subprogram can have a new activation record instance on the stack
- Activation records are more complex
  - The compiler must generate code to cause implicit allocation and deallocation of local variables
  - Activation record instances are created dynamically
  - Usually the caller’s frame pointer to its own activation instance has been saved

Frame Pointer / Dynamic Link

- Each subprogram activation needs a pointer to its own activation record
  - Usually called the “frame pointer” from “stack frame” (another name for activation record)
- Some machines may have only one register available for this purpose
  - So the caller’s frame pointer has be to saved on the stack
  - But the frame pointer also provides other valuable information
  - For dynamically scoped languages it is a dynamic link to non-local variables
  - For statically scoped languages it can be used to provide stack traceback information

Activation Record

- An activation record is dynamically created when a subprogram is called
  - Activation record instances reside on the run-time stack
  - The activation record format is static, but its size may be dynamic
- The Environment Pointer (EP) or Frame Pointer (FP) must be maintained by the run-time system.
  - It always points at a known location in the activation record instance of the currently executing program unit
  - Conceptually we think of the Frame Pointer as pointing to the base of the activation record but in practice it may point at a known location that is not the base
  - Example: x86 architecture the frame pointer points to the dynamic link and parameters are located at negative offsets
Typical Activation Record Structure

- Diagram from text
- More common is this order

Dynamic Chain and Local Offset

- The collection of dynamic links in the stack at a given time is called the dynamic chain, or call chain
- Local variables can be accessed by their offset from the beginning of the activation record, whose address is in the EP. This offset is called the local_offset
- The local_offset of a local variable can be determined by the compiler at compile time

An Example: C Function

```c
void sub(float total, int part)
{
    int list[5];
    float sum;
    ...
}
```

Prologue and Epilogue Code

```c
int main()
{
    int i, j;
    clock_t start, finish;
    _i$ = -4
    _j$ = -8
    _start$ = -16
    _finish$ = -12
    $T2208 = -24
    $T2209 = -32
    $T2210 = -40
    $T2211 = -48
    ... push ebp
    mov ebp, esp
    sub esp, 64
    ; 00000040H
}
```

Prologue and Epilogue Code

```c
char *strcopy1 (char *dst, char *src)
{
    char *cp = dst;
    while( *cp++ = *src++ )
    return( dst );
}
```

Prologue and Epilogue Code

```c
    mov eax, DWORD PTR _dst$[ebp]
    mov esp, ebp
    pop ebp
    ret
```
An Example Without Recursion

void A(int x) {
    int y;
    ...
    C(y);
    ...
}
void B(float r) {
    int s, t;
    ...
    A(r);
    ...
}
void C(int q) {
    ...
}
void main() {
    float p;
    ...
    B(p);
    ...
}

An Example With Recursion

• The activation record used in the previous example supports recursion, e.g.

```
int factorial (int n) {
    if (n <= 1) return 1;
    else return (n * factorial(n - 1));
}
```

void main() {
    int value;
    value = factorial(3);
}

Real-world stack frames

• Can be substantially more complex and vary with architecture
• Ex: ARM Processor
Win32 C++ With Exception Handlers

- Static scoped languages such as Fortran 95, Ada, Python, JavaScript, Ruby, and Lua use stack-dynamic local variables and allow subprograms to be nested.
- All variables that can be non-locally accessed reside in some activation record instance in the stack.
  - By definition a nested routine can only be called when its ancestors are active.
  - Therefore all non-local references are either to stack dynamic local variables of an ancestor or are global references.
- To locate a non-local reference:
  1. Find the correct activation record instance
  2. Determine the correct offset within that activation record instance

Locating a Non-local Reference

- Finding the offset is easy
- Finding the correct activation record instance is a bit more tricky
- Two major techniques:
  1. Static chain: add a static link to the activation record that points to the static link of the direct ancestor
  2. Display: maintain static links in a separate stack
- We will discuss static chains only

MIPS and Power PC

- Win32 cdecl
- Nested Subprograms
  - Static scoped languages such as Fortran 95, Ada, Python, JavaScript, Ruby, and Lua use stack-dynamic local variables and allow subprograms to be nested.
  - All variables that can be non-locally accessed reside in some activation record instance in the stack.
    - By definition a nested routine can only be called when its ancestors are active.
    - Therefore all non-local references are either to stack dynamic local variables of an ancestor or are global references.
  - To locate a non-local reference:
    1. Find the correct activation record instance
    2. Determine the correct offset within that activation record instance

Static Scoping

- A static chain is a chain of static links that connects activation records:
  - The static link in an activation record for subprogram A points to one of the activation record instances of A’s static parent.
  - The static chain from an activation record instance connects it to all of its static ancestors.
  - Static depth is an integer associated with a static scope whose value is the depth of nesting of that scope.
Static Scoping

• The chain offset or nesting depth of a nonlocal reference is the difference between the static depth of the reference and that of the scope where it is declared.

• A reference to a variable can be represented by the pair:
  \((\text{chain_offset}, \text{local_offset})\)
  where local_offset is the offset in the activation record of the variable being referenced.

• Chain offset is the number of static links that we follow to find the referencing environment.

Example Ada Program

```
procedure Main_2 is
  X : Integer;
begin
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin
        A := B + C;
      end;
    end;
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin
          Sub1;
          E := B + A;
        end;
      end;
      begin
        Sub3;
        A := D + E;
      end;
    end;
  end;
  begin
    Sub2(7);
  end;
end; of Main_2
```

Static Depth

• Static depth for each procedure:
  - `procedure Main_2 is` \(\rightarrow 0\)
  - `procedure Bigsub is` \(\rightarrow 1\)
  - `procedure Sub1 is` \(\rightarrow 2\)
  - `procedure Sub2 is` \(\rightarrow 2\)
  - `procedure Sub3 is` \(\rightarrow 3\)

• Sub3 has \(E(s3) := B(s2) + A(bigsun)\):
  - So Sub3 reference to B has chain offset 1 and the reference to B has chain offset 2.

Static Chain Maintenance

• When a call is made, the activation record instance must be built:
  - The dynamic link is the old frame pointer.
  - The static link will point to the most recent activation record of the static parent.
  - Two methods to construct the static link:
    1. Search the dynamic chain.
    2. Treat subprogram calls and definitions like variable references and definitions.

Issues with Static Chains

• Problems:
  1. A nonlocal reference is slow if the nesting depth is large.
  2. Time-critical code is difficult:
     a. Costs of nonlocal references are difficult to determine.
     b. Code changes can change the nesting depth, and therefore the cost.
Displays
- An alternative to static chains that solves the problems with that approach
- Static links are stored in a single array called a display
- The contents of the display at any given time is a list of addresses of the accessible activation record instances

Blocks
- Blocks are user-specified local scopes for variables
- Example in C:
  ```c
  int temp;
  list [upper] = list [lower];
  list [lower] = temp
  ```
- The lifetime of `temp` begins when control enters the block
- An advantage of using a local variable like `temp` is that assignments cannot alter any other variable with the same name

Implementing Blocks
- Two Methods:
  1. Treat blocks as parameter-less subprograms that are always called from the same location
     - Every block has an activation record; an instance is created every time the block is executed
  2. Since the maximum storage required for a block can be statically determined, this amount of space can be allocated after the local variables in the activation record

Implementing Dynamic Scoping
- Two possible ways to implement dynamic scoping are called deep access and shallow access
- These are different from deep and shallow binding (different semantics) in that the semantics of dynamic scoping are unaltered by the access method
Deep Access Example

```c
void sub3() {
    int x, z;
    x = u + v;
}

void sub2() {
    int w, x;
}

void sub1() {
    int v, w;
}

void main() {
    int v, u;
}
```

Shallow Access for Dynamic Scoping

```c
void sub3() {
    int x, z;
    x = u + v;
}

void sub2() {
    int w, x;
}

void sub1() {
    int v, w;
}

void main() {
    int v, u;
}
```

Where:
- main calls sub1
- sub1 calls sub1
- sub1 calls sub2
- sub2 calls sub3

(Shallow Access for Dynamic Scoping)

```plaintext
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

(The names in the stack cells indicate the program units of the variable declaration.)