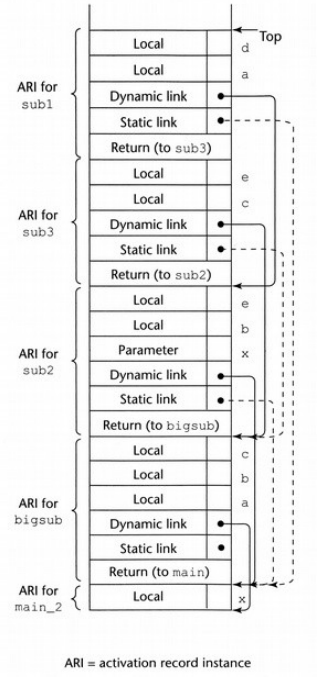
CS 430 Spring 2019

Mike Lam, Professor



Activations and Environments

Runtime Environment

- Programs run in the context of a system
  - Instructions, registers, memory, I/O ports, etc.
- Compilers must emit code that uses this system
  - Must obey the rules of the hardware and OS
  - Must be interoperable with shared libraries compiled by a different compiler
- Memory conventions:
  - Stack (used for subprogram calls)
  - Heap (used for dynamic memory allocation)

Subprograms

- Subprogram general characteristics
  - Single entry point
  - Caller is suspended while subprogram is executing
  - Control returns to caller when subprogram completes
  - Caller/callee info stored on stack

- Activation record*:* data for a single subprogram execution
  - Local variables
  - Parameters
  - Saved registers
  - Dynamic link (base pointer) and/or static link
  - Return address

Subprogram Activation

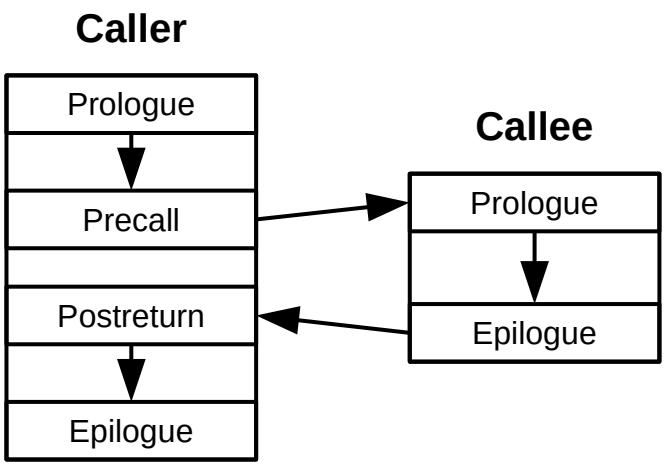
- Call semantics
  - Save caller status
  - Compute and store parameters
  - Save return address
  - Transfer control to callee
- Return semantics
  - Save return value(s) and out parameters
  - Restore caller status
  - Transfer control back to the caller

Linkage contract or calling convention

(caller and callee must agree)

Standard Linkages

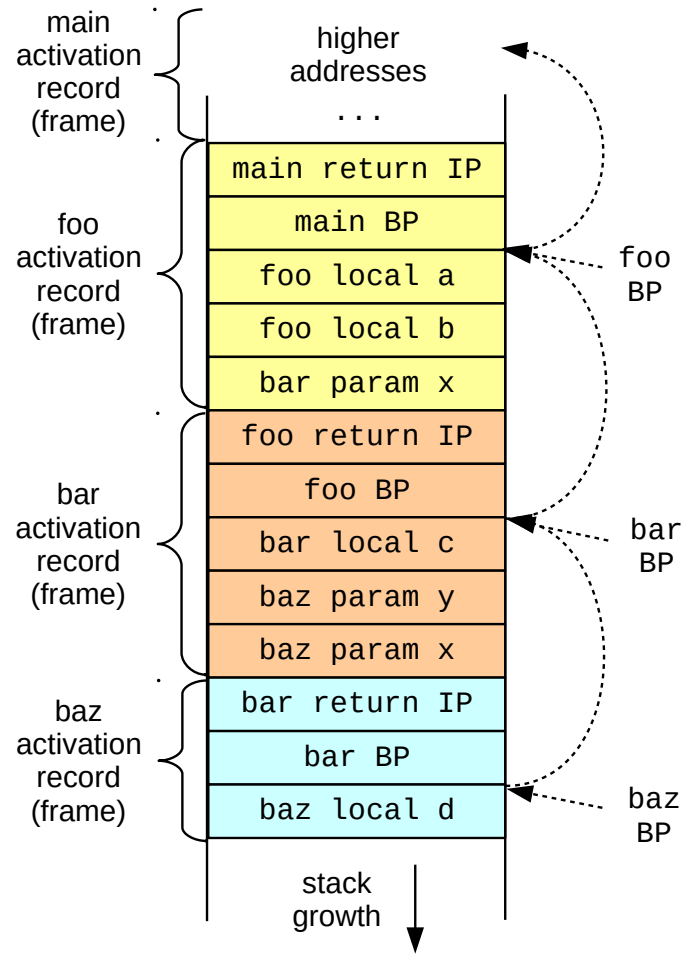
- Standard contract:
  - Caller: precall sequence
    - Evaluate and push parameters
    - Save return address
    - Transfer control to callee
  - Callee: prologue sequence
    - Save & initialize base pointer
    - Allocate space for local variables
  - Callee: epilogue sequence
    - De-allocate activation record
    - Transfer control back to caller
  - Caller: postreturn sequence
    - Clean up parameters



x86 Stack Layout

- Address space
  - Code, static, stack, heap
- Instruction Pointer (IP)
  - Current instruction
- Stack pointer (SP)
  - Top of stack (lowest address)
- Base pointer (BP)
  - Start of current frame (after saving old IP and BP)
  - In CPL: environment pointer or dynamic link
- "cdecl" calling conventions
  - callee may use AX, CX, DX
  - callee must preserve all other registers
  - parameters pushed in reverse order (RTL)
  - return value saved in AX

```
foo():          bar(x):         baz(x, y):
    int a,b         int c           int d
    bar(a)          bax(x,c)        return
    return          return
```



x86 Calling Conventions

Prologue:

```
push %ebp                ; save old base pointer
mov  %esp, %ebp          ; save top of stack as base pointer
sub  X, %esp             ; reserve X bytes for local vars
```

Within function:

```
+OFFSET(%ebp)            ; function parameter
-OFFSET(%ebp)            ; local variable
```

Epilogue:

```
<optional: save return value in %eax>
leave                    ; mov %ebp, %esp
                         ; pop %ebp
ret                      ; pop stack and jump to popped address
```

Function calling:

```
<push parameters>        ; precall
<push return address>
<jump to fname>          ; call
<pop parameters>         ; postreturn
```

Calling Conventions

	Integral parameters	**Base pointer**	**Caller-saved registers**	**Return value**
cdecl (x86)	On stack (RTL)	Always saved	EAX, ECX, EDX	EAX
AMD64 (x64)	RDI, RSI, RDX, RCX, R8, R9, then on stack (RTL)	Saved only if necessary	RAX, RCX, RDX, R8-R11	RAX
Decaf	On stack (RTL)	Always saved	All	RET

Static scoping

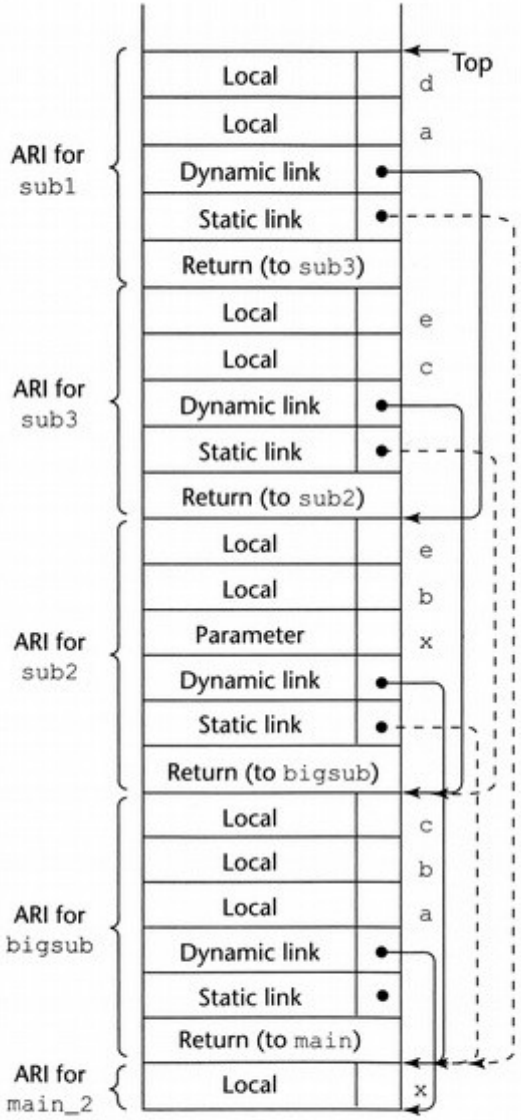
- Static scoping is harder with nested subprograms
  - Must be able to look up variables by **lexical** scope
- Primary method: static chains
  - Introduce a new static link
    - Similar to dynamic link, but tracks lexical scopes
  - Associate (chain-offset, local-offset) pairs with each variable
    - Follow *chain-offset* # of static links
    - Then use *local-offset* to find variable in its activation record

Dynamic scoping

- Dynamic scoping
  - Must be able to look up variables by **dynamic** scope
- One approach: deep access
  - Search all activation records, one at a time
  - Slow access but fast linkage
- Another approach: shallow access
  - Maintain a stack for each variable
  - Push/pop alongside regular activation record
  - Active copy is always on top of the stack
  - Faster access but slower linkage

Example

```
function main(){
  var x;
  function bigsub() {
    var a, b, c;
    function sub1 {
      var a, d;
      ...
      a = b + c; <--------------------------------1
      ...
    }  // end of sub1
    function sub2(x) {
      var b, e;
      function sub3() {
        var c, e;
        ...
        sub1();
        ...
        e = b + a; <--------------------------------2
      } // end of sub3
      ...
      sub3();
      ...
      a = d + e; <---------------------------------3
    } // end of sub2
    ...
    sub2(7);
    ...
  } // end of bigsub
  ...
  bigsub();
  ...
} // end of main
```



ARI = activation record instance

Exercise

```
def P() {

    var x = ‘p’

    def A() {
        println(x)
    }

    def B() {
        var x = ‘b’
        def C() {
            var x = ‘c’
            println(x)
            D()
        }
        def D() {
            println(x)
            A()
        }
        C()
    }
    B()
}
```

Trace this program using the activation record layout below.

Local Variables
Static Link
Dynamic Link
Return Address

Exercise

output

sta	dyn
c	c
b	c
p	c