CS 432
Fall 2015

Mike Lam, Professor

Runtime Environments

(a.k.a. procedure calls and heap management)

Subprograms

- General characteristics
  - Single entry point
  - Caller is suspended while subprogram is executing
  - Control returns to caller when subprogram completes
- Procedure vs. function
  - Functions have return values

Subprograms

- New-ish terms
  - Header: signaling syntax for defining a subprogram
  - Parameter profile: number, types, and order of parameters
  - Signature/protocol: parameter types and return type(s)
  - Prototype: declaration without a full definition
  - Referencing environment: variables visible inside a subprogram
  - Name space / scope: set of visible names
  - Call site: location of a subprogram invocation
  - Return address: destination in caller after call completes

Parameters

- Formal vs. actual parameters
  - Formal: parameter inside subprogram definition
  - Actual: parameter at call site
- Semantic models: *in*, *out*, *in-out*
- Implementations (key differences are *when* values are copied and exactly *what* is being copied)
  - **Pass-by-value (*in, value*)**
  - Pass-by-result (*out, value*)
  - Pass-by-copy (*in-out, value*)
  - **Pass-by-reference (*in-out, reference*)**
  - **Pass-by-name (*in-out, name*)**

Parameters

- Pass-by-value
  - Pro: simple
  - Con: costs of allocation and copying
  - Often the default
- Pass-by-reference
  - Pro: efficient (only copy 32/64 bits)
  - Con: hard to reason about, extra layer of indirection, aliasing issues
  - Often used in object-oriented languages
- Pass-by-name
  - Pro: powerful
  - Con: expensive to implement, very difficult to reason about
  - **Rarely used!**

Other Design Issues

- How are name spaces defined?
  - Lexical vs. dynamic scope
- How are formal/actual parameters associated?
  - Positionally, by name, or both?
- Are parameter default values allowed?
- Are method parameters type-checked?
  - Statically or dynamically?

Other Design Issues

- Are local variables statically or dynamically allocated?
- Can subprograms be passed as parameters?
  - How is this implemented?
- Can subprograms be nested?
- Can subprograms be polymorphic?
  - Ad-hoc/manual, subtype, or parametric/generic?
- Are function side effects allowed?
- Can a function return multiple values?

Misc. Topics

- Macros
  - Call-by-name, “executed” at compile time
- Closures
  - A subprogram and its referencing environment
- Coroutines
  - Co-operating procedures
- Just-in-time (JIT) compilation
  - Defer compilation of each function until it is called

Subprogram Activation

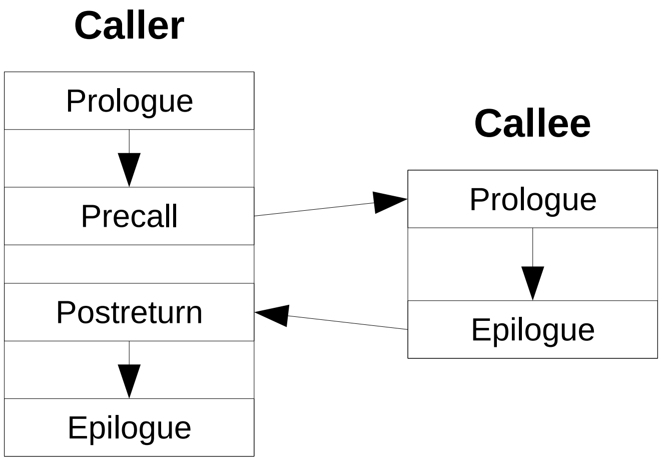
- Call semantics:
  - Save caller status
  - Compute and save parameters
  - Save return address
  - Transfer control to callee
- Return semantics:
  - Save return value(s) and out parameters
  - Restore caller status
  - Transfer control back to the caller

Linkage contract
(caller and callee must
agree)

- *Activation record:* data for a single subprogram execution
  - Local variables
  - Parameters
  - Return address
  - Dynamic link

Standard Linkages

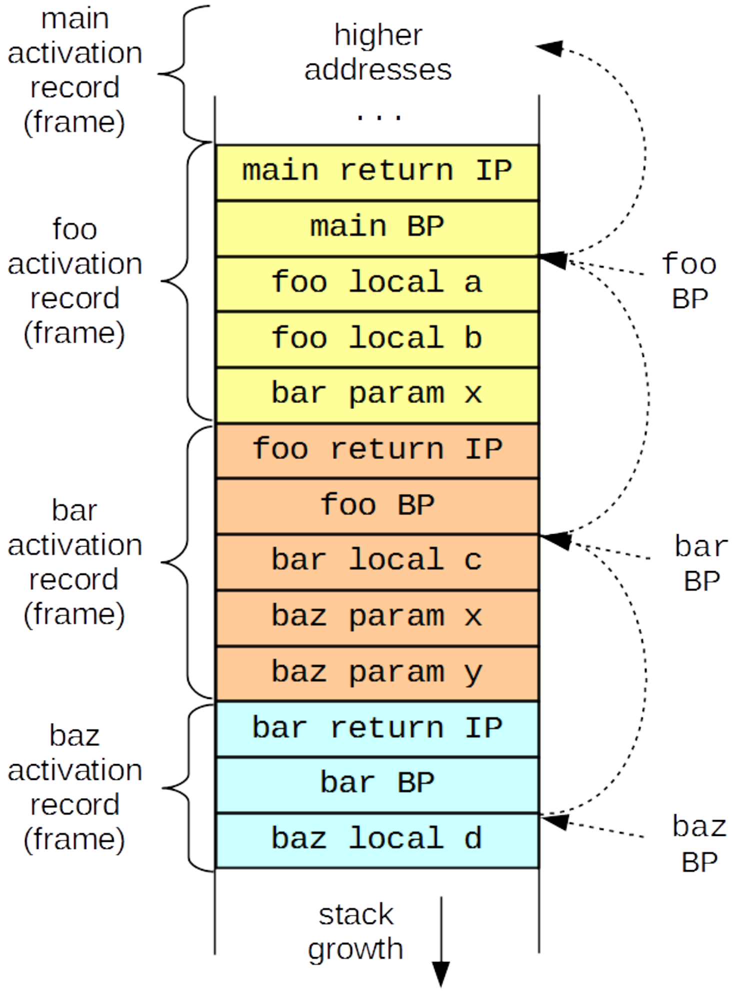
- Caller and callee must agree
- Standard contract:
  - Caller: precall sequence
    - Evaluate and push parameters
    - Save return address
    - Transfer control to callee
  - Callee: prologue sequence
    - Save & initialize base pointer
    - Allocate space for local variables
  - Callee: epilogue 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
- Base pointer (BP)
  - Start of current frame
- "cdecl" calling conventions
  - callee must preserve bx, bp, sp, si (save/restore if used)
  - callee may use ax, cx, dx, flags, st0-7, mm0-7, xmm0-15
  - parameters may be passed in di, si, dx, cx, r8-15
  - return value saved in ax
  - function stack frame starting at base pointer (bp)

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



main activation record (frame)

higher addresses

. . .

main return IP

main BP

foo local a

foo local b

bar param x

foo return IP

foo BP

bar local c

baz param x

baz param y

bar return IP

bar BP

baz local d

foo activation record (frame)
bar activation record (frame)
baz activation record (frame)
foo BP
bar BP
baz BP
stack growth

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
```

x86_64 "red zone" (128 bytes reserved below SP)

- optimization: do not explicitly build frame (no SP manipulation)

Decaf Calling Conventions

- **`param`** instruction to pass all parameters
  - Pushed on system stack
  - Accessible in function using [bp+offset]
  - No need to manually pop after call
- **`call`** instruction to transfer control
  - Save return address on stack
  - Set up stack frame (BP and SP)
  - Set IP to function entry point
- **`return`** instruction to return to caller
  - Tear down stack frame (BP and SP)
  - Set IP to return address
  - Return value saved in "ret" special register

Heap Management

- Desired properties
  - Space efficiency
  - Exploitation of locality (time and space)
  - Low overhead
- Allocation (malloc/new)
  - First-fit vs. best-fit vs. next-fit
  - Coalescing free space (defragmentation)
- Manual deallocation (free/delete)
  - Dangling pointers
  - Memory leaks

Automatic De-allocation

- Criteria: overhead, pause time, space usage, locality impact
- Basic problem: finding reachable structures
  - Root set: static and stack pointers
  - Follow pointers through heap structures
- Reference counting
  - Catch the transition to unreachable
  - Has trouble with cyclic data structures
- Mark and sweep (tracing)
  - Occasionally pause and detect reachable
  - High overhead and undesirable "pause the world" semantics
  - Incremental collection: interleave computation and collection
  - Partial collection: collect only a subset of memory on each run
  - Generational collection: collect newer objects more often
  - Collection can be parallelized to a certain extent

Object-Oriented Languages

- Classes vs. objects
- Inheritance relationships (subclass/superclass)
  - Single vs. multiple inheritance
- Closed vs. open class structure
- Visibility: public vs. private vs. protected
- Static vs. dynamic dispatch
- Object-records and virtual method tables