CS 432
Fall 2019

Mike Lam, Professor

Runtime 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
- 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

Subprogram Activation

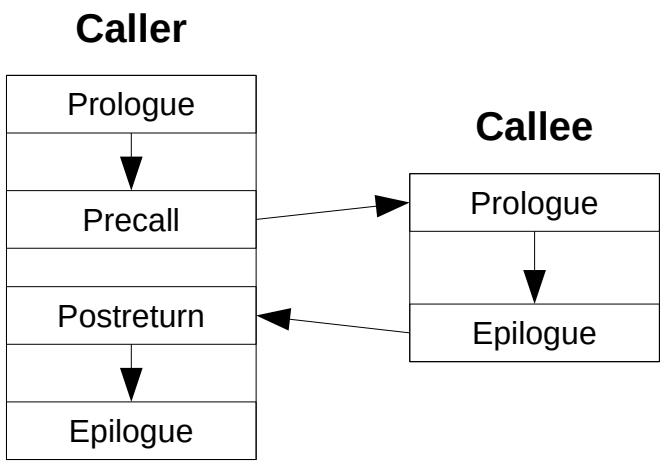
- Call semantics:
  - Save caller status
  - Compute and store parameters
  - Save return address
  - Transfer control to callee
- Return semantics:
  - Save return value(s) and out parameters
  - Restore caller status
  - Transfer control back to the caller

Linkage contract or Calling conventions (caller and callee must agree)

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

Standard Linkages

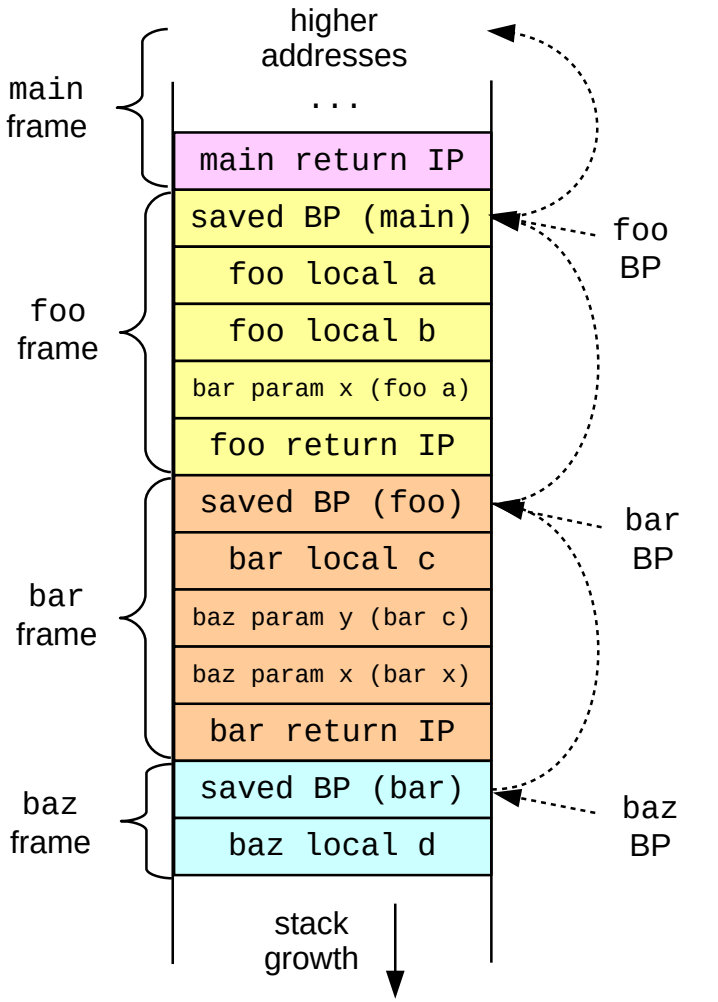
- Caller and callee must agree
- Standard contract:
  - Caller: precall sequence
    - Evaluate and store parameters
    - Save return address
    - Transfer control to callee
  - Callee: prologue sequence
    - Save & initialize base pointer
    - Allocate space for local variables
  - Callee: epilogue 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 (i.e., saved BP)
- "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

```
void foo()        void bar(x)        void baz(x,y)
{                 {                  {
    int a,b;          int c;             int d;
    bar(a)            baz(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>
mov  %ebp, %esp          ; restore old stack pointer
pop  %ebp                ; restore old base pointer
ret                      ; pop stack and jump to popped address
```

Function calling:

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

ILOC Calling Conventions

Prologue:

```
push BP                 ; save old base pointer
i2i  SP => BP           ; save top of stack as base pointer
addI SP, -X => SP       ; reserve X bytes for local vars
```

Within function:

```
[BP+OFFSET]             ; function parameter
[BP-OFFSET]             ; local variable
```

Epilogue:

```
<optional: save return value in RET>
i2i  BP => SP           ; restore old stack pointer
pop  BP                 ; restore old base pointer
return                  ; pop stack and jump to popped address
```

Function calling:

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

Calling Conventions

	Integral parameters	**Base pointer**	**Caller-saved registers**	**Return value**
cdecl (x86)	On stack (RTL)	Always saved	EAX, ECX, EDX	EAX
x86-64 (x64)	RDI, RSI, RDX, RCX, R8, R9, then on stack (RTL)	Saved only if necessary	RAX, RCX, RDX, R8-R11	RAX
ILOC	On stack (RTL)	Always saved	All virtual registers	RET

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?
  - For all parameters or just the last one(s)?
- 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

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
  - Recursively follow pointers through heap structures
- Reference counting (incremental)
  - Memory/time overhead to track the number of active references to each structure
  - Catch the transition to unreachable (count becomes zero)
  - Has trouble with cyclic data structures
- Mark and sweep (batch-oriented)
  - Occasionally pause and detect unreachable structures
  - High time overhead and potentially undesirable "pause the world" semantics
  - Partial collection: collect only a subset of memory on each run
  - Generational collection: collect newer objects more often

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