An x64 processor is screaming along at billions of cycles per second to run the XNU kernel, which is frantically working through all the POSIX-specified abstraction to create the Darwin system underlying OS X, which in turn is straining itself to run Firefox and its Gecko renderer, which creates a Flash object which renders dozens of video frames every second because I wanted to see a cat jump into a box and fall over.

I am a god.

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 vs. method**
  - Functions have return values
  - Methods have an associated object (the receiver)
• 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
  - **Aliases**: different names for the same location
  - **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

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

Linkage contract or Calling conventions (caller and callee must agree)
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)
  - r_{arp} in EAC, EP in Sebesta
- "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
x86 Calling Conventions

Prologue:

\textbf{push} \%ebp \quad ; \text{save old base pointer} \\
\textbf{mov} \%esp, \%ebp \quad ; \text{save top of stack as base pointer} \\
\textbf{sub} \quad X, \%esp \quad ; \text{reserve X bytes for local vars} \\

Within function:

\text{+OFFSET(\%ebp)} \quad ; \text{function parameter} \\
\text{-OFFSET(\%ebp)} \quad ; \text{local variable} \\

Epilogue:

\text{<optional: save return value in %eax>} \\
\textbf{mov} \%ebp, \%esp \quad ; \text{restore old stack pointer} \\
\textbf{pop} \%ebp \quad ; \text{restore old base pointer} \\
\textbf{ret} \quad ; \text{pop stack and jump to popped address} \\

Function calling:

\text{<push parameters>} \quad ; \text{precall} \\
\text{<push return address>} \\
\text{<jump to fname>} \quad ; \text{call} \\
\text{<dealloc parameters>} \quad ; \text{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

<table>
<thead>
<tr>
<th></th>
<th>Integral parameters</th>
<th>Base pointer</th>
<th>Caller-saved registers</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdecl (x86)</td>
<td>On stack (RTL)</td>
<td>Always saved</td>
<td>EAX, ECX, EDX</td>
<td>EAX</td>
</tr>
<tr>
<td>x86-64 (x64)</td>
<td>RDI, RSI, RDX, RCX, R8, R9, then on stack (RTL)</td>
<td>Saved only if necessary</td>
<td>RAX, RCX, RDX, R8-R11</td>
<td>RAX</td>
</tr>
<tr>
<td>ILOC</td>
<td>On stack (RTL)</td>
<td>Always saved</td>
<td>All virtual registers</td>
<td>RET</td>
</tr>
</tbody>
</table>
Other Design Issues

- How are name spaces defined?
  - Lexical (static) vs. dynamic scope
  - Parent scope determined by code vs. call order
- 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

• Can subprograms be passed as parameters?
  - How is this implemented?
• Can subprograms be nested?
  - Lexical or dynamic scoping?
• 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
  - Often provide call-by-name semantics

- **Closures**
  - A subprogram and its referencing environment
  - Requires a more general structure than the stack

- **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