CS 432
Fall 2024

Mike Lam, Professor

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 procedure calls)
  - Heap (used for dynamic memory allocation)

Procedures

- A procedure is a portion of code packaged for re-use
  - Key abstraction in software development
  - Provides modularity, encapsulation, and information hiding
- Common characteristics
  - Single entry point, (potentially) multiple exit points
  - Caller is suspended while procedure is executing
  - Control returns to caller when procedure completes
  - Caller and callee info stored on stack
- Procedure vs. function vs. method
  - We’ll use “subprogram” as a synonym for “procedure”
  - Functions generally have return values
  - Methods have an associated object (the receiver)

Procedures

- New-ish terms
  - Header: signaling syntax for defining a procedure
  - 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 procedure
  - Name space / scope: set of visible names
  - Aliases: different names for the same location
  - Caller: procedure that calls another procedure
  - Callee: procedure called by another procedure
  - Call site: location of a procedure invocation
  - Return address: destination in caller after call completes

Parameters

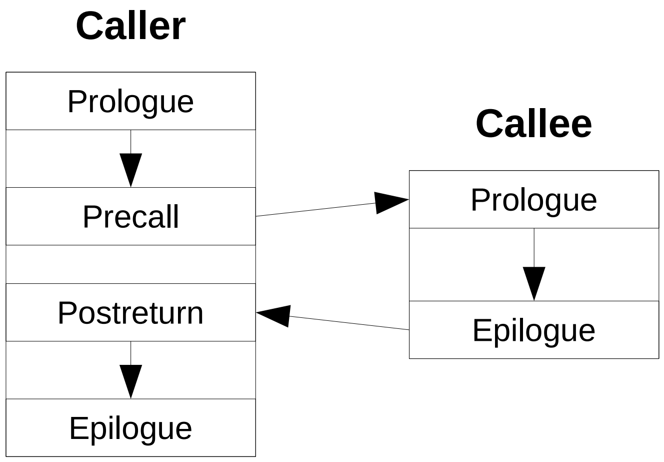
- Formal vs. actual parameters
  - Formal: parameter inside procedure definition
  - Actual: parameter at call site
- Semantic models: *in*, *out*, *in-out*
- Implementations (key differences are *when* values are copied and exactly *what* is being copied)
  - **Pass-by-value (*in, value*)**
  - Pass-by-result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

Procedure Activation

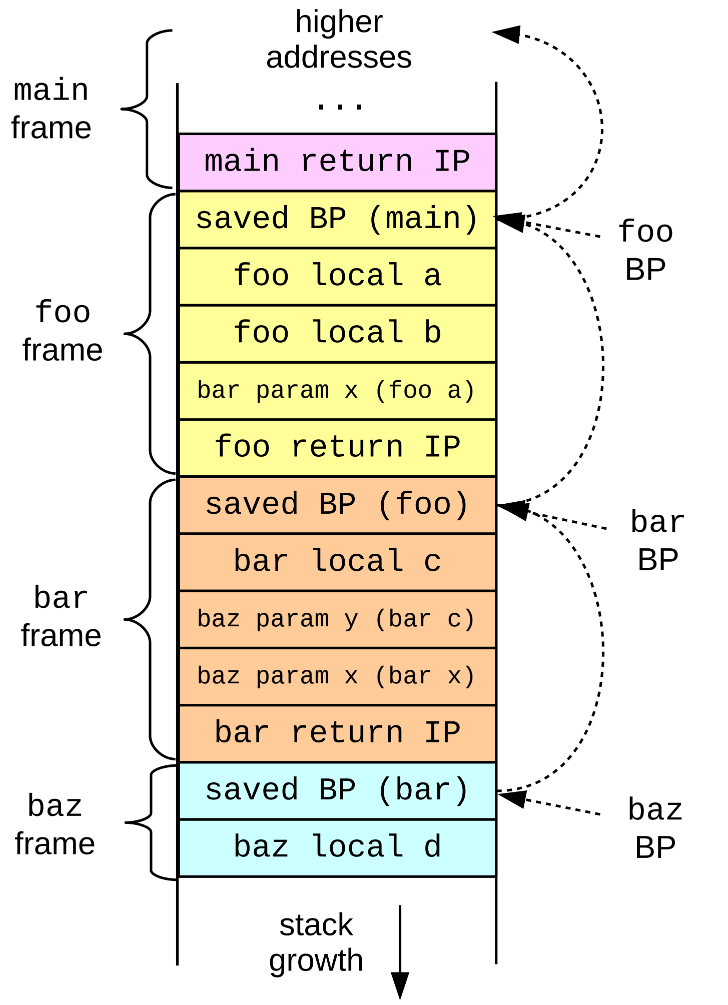
- Caller and callee must agree on calling conventions
- Standard calling contract:
  - Caller: precall sequence
    - Evaluate and store parameters
    - Save return address
    - Transfer control to callee
  - Callee: prologue sequence
    - Save & initialize base pointer
    - Allocate space for local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



General Stack Layout

- Stack pointer (SP)
  - Top of stack (lowest address)
- Base pointer (BP)
  - Start of current frame (i.e., saved BP)
  - r_{arp} in EAC (CS 432)
  - EP in Sebesta (CS 430)
- Stack frame / activation record per call
  - Parameters
    - Positive offset from BP
  - **Saved return address (required)**
  - Saved BP (dynamic link)
  - Local variables
    - Negative offset from BP
    - Allocated by decrementing SP

```
void foo()          void bar(x)          void baz(x,y)
{                   {                    {
    int a,b;            int c;               int d;
    bar(a)              baz(x,c);            return;
    return;             return;          }
}                   }
```



higher addresses

main frame

...

main return IP

foo frame

saved BP (main)

foo local a

foo local b

bar param x (foo a)

foo return IP

foo BP

bar frame

saved BP (foo)

bar local c

baz param y (bar c)

baz param x (bar x)

bar return IP

bar BP

baz frame

saved BP (bar)

baz local d

baz BP

stack growth

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

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
call <fname>          ; save return address and jump
<dealloc parameters>  ; postreturn
```

Much of prologue & epilogue is optional in x86-64

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
```

(required even if there are no local vars – X may need to be adjusted during register allocation for spilled registers)

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
call <fname>          ; save return address and jump
<dealloc parameters>  ; postreturn
```

Described in detail in section 8 of Decaf reference

Example

```
def void foo()
{
}

def int main()
{
    foo();
    return 0;
}
```

```
foo:
  push BP                 ; prologue
  i2i SP => BP            ;
  addI SP, 0 => SP        ;
l0:
  i2i BP => SP            ; epilogue
  pop BP                  ;
  return                  ;

main:
  push BP                 ; prologue
  i2i SP => BP            ;
  addI SP, 0 => SP        ;
  call foo
  addI SP, 0 => SP
  loadI 0 => r0
  i2i r0 => RET
  jump l1
l1:
  i2i BP => SP            ; epilogue
  pop BP                  ;
  return                  ;
```

Example

```
def int add(int x, int y)
{
    int sum;
    sum = x + y;
    return sum;
}

def int main()
{
    return add(3, 7);
}
```

```
add:
  push BP                        ; prologue
  i2i SP => BP                   ;
  addI SP, -8 => SP              ;
  loadAI [BP+16] => r0           ; load param x
  loadAI [BP+24] => r1           ; load param y
  add r0, r1 => r2
  storeAI r2 => [BP-8]           ; store local sum
  loadAI [BP-8] => r3            ; load local sum
  i2i r3 => RET
  jump l0
l0:
  i2i BP => SP                   ; epilogue
  pop BP                         ;
  return                         ;

main:
  push BP                        ; prologue
  i2i SP => BP                   ;
  addI SP, 0 => SP               ;
  loadI 3 => r4
  loadI 7 => r5
  push r5                        ; precall
  push r4                        ;
  call add
  addI SP, 16 => SP              ; postreturn
  i2i RET => r6
  i2i r6 => RET
  jump l1
l1:
  i2i BP => SP                   ; epilogue
  pop BP                         ;
  return                         ;
```

PL Design Issues

- There are many procedure-related design questions; decisions are made by language designers w/ impacts on compiler writers
- 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?

PL Design Issues

- Can procedures be passed as parameters?
  - How is this implemented? (e.g., with *closures*)
- Can procedures be *nested*?
  - *Lexical* or *dynamic* scoping?
- Can procedures be *polymorphic*?
  - Ad-hoc/manual, subtype, or parametric/generic?
- Can method calls be resolved at compile time?
  - *Static* vs. *dynamic* dispatch (and *single* vs. *multiple* dispatch)
- Are function side effects allowed?
- Can a function return multiple values?

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 (and single vs. multiple)
- Object-records and virtual method tables

Miscellaneous Topics

- Macros
  - “Executed” at compile time
  - Often provide call-by-name semantics
- Closures
  - A procedure 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
  - New chapter in EAC3e!

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-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