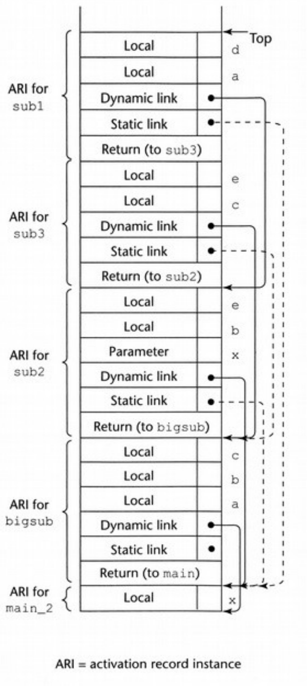
CS 430
Spring 2022

Mike Lam, Professor



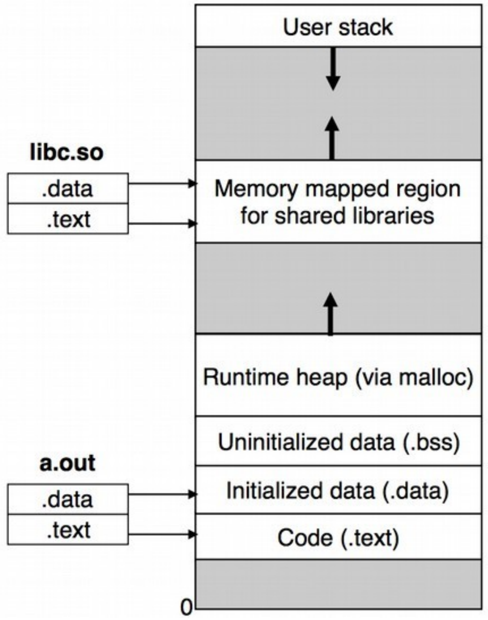
Activations and Environments

Course Outline

- Syntax (modules 2-3)
- Semantics (modules 5-8, 10-11, and 13-14)
  - Variables and scoping
  - Types and type checking
  - Expressions and control structures
  - Parameters and subprograms
- **Implementation** (modules 16 and 18-19)
  - Activation and environments
  - Abstraction and OOP
  - Concurrency and error handling
- History (module 20)

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)

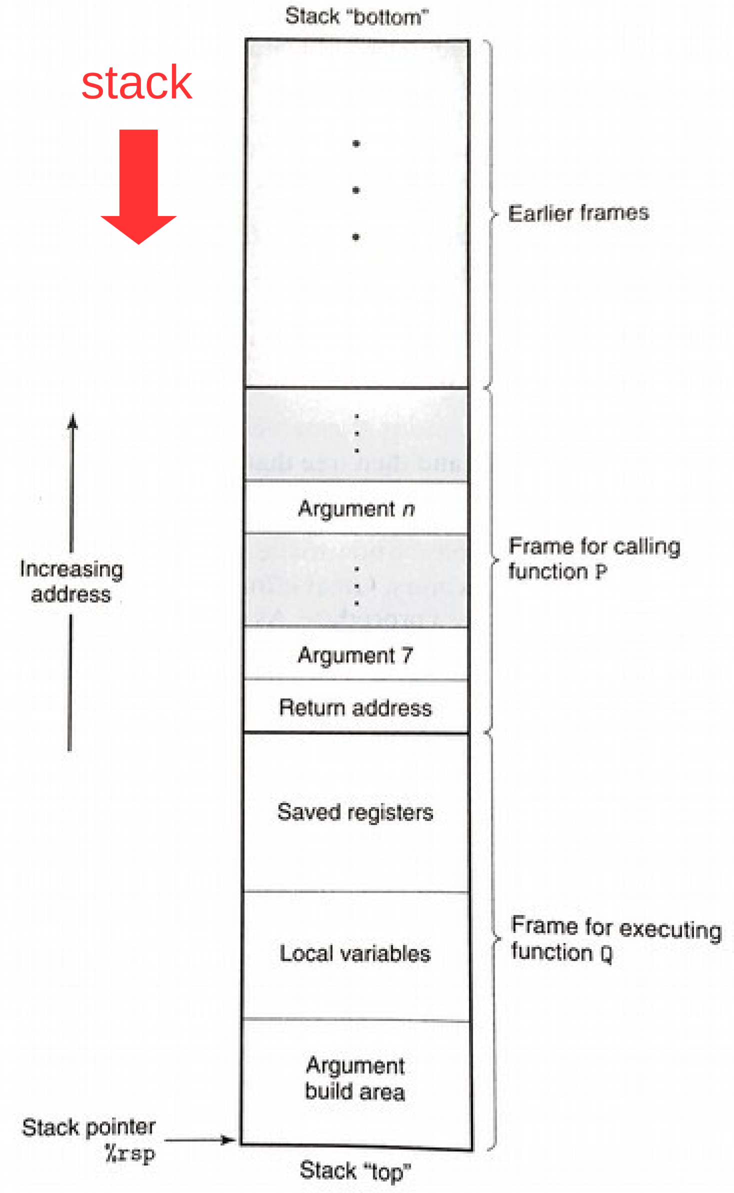


Runtime Environment

- In this module we'll focus on calling conventions
  - How the system stack (w/ top tracked using the stack pointer) is used for subprogram invocation/activation
- But first: a CS 261 review
  - You've seen calling conventions already!
  - Remember these slides?

Runtime stack

- Basic idea: maintain a system stack frame for each procedure call
  - All active procedure have a frame
  - Each frame stores information about a single active call
    - Arguments, local variables, return address
  - GDB's "backtrace" command follows the chain up
  - Recursion just works!



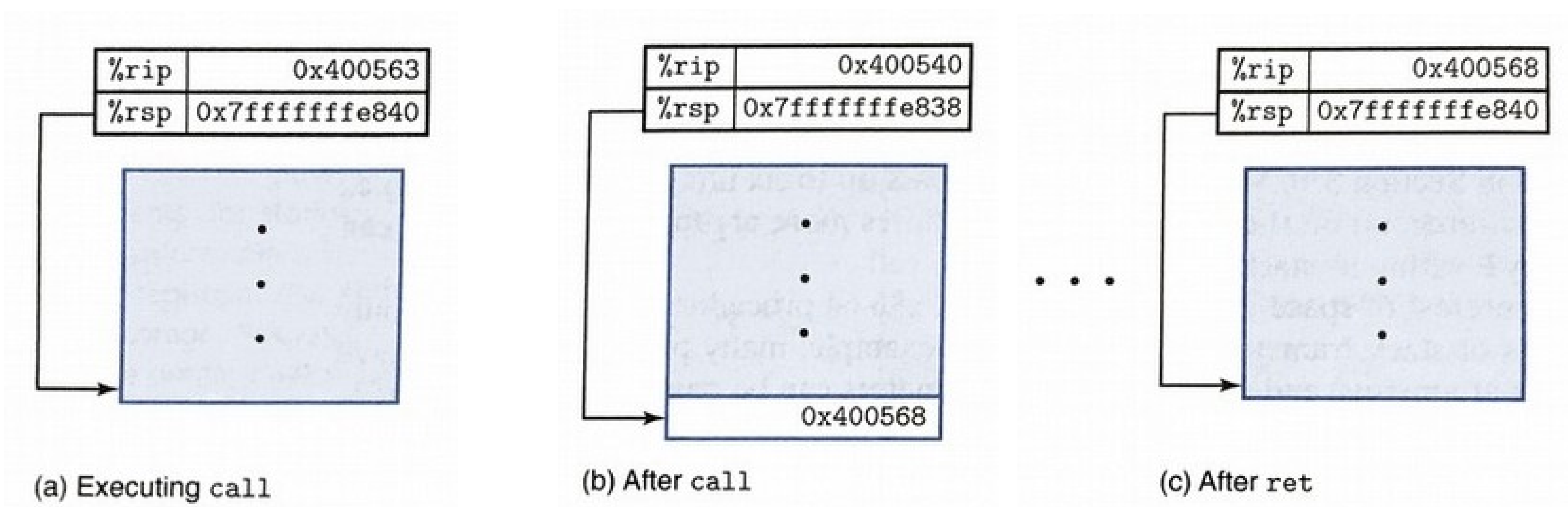
Here function P has called function Q

Control transfer

- Use stack to store return addresses
  - Return address: the instruction AFTER the `call`
  - `call` / `callq` pushes 64-bit return address onto stack
  - `ret` / `retq` pops the return address and sets `%rip`

```
400550 <main>:
  ...
  400563   callq 400540 <foo>
  400568   movq 0x8(%rsp), %rdx
  ...
```

```
400540 <foo>:
  400540   xorq %rax, %rax
  ...
  40054d   retq
```



Data transfer

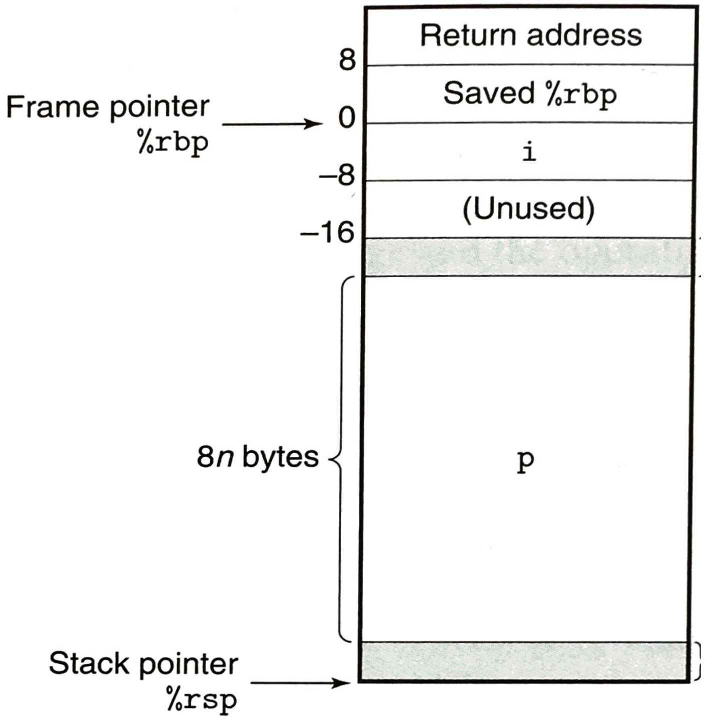
- In x86-64, up to six integral (integer or pointer) arguments are passed to a procedure via registers:
  - `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
  - Other arguments are passed on the stack (and pushed in reverse order)
- A single return value is passed back via `%rax`
  - Large structs often “returned” using a pointer

Local storage (registers)

- Some registers are designated callee-saved
  - In x86-64: `%rbx, %rbp, %r12, %r13, %r14, %r15`
  - A procedure must save/restore these registers (often using push/pop) if they are used during the procedure
  - When possible, avoid using these registers inside procedures (lower overhead)
- Other registers (except `%rsp`) are caller-saved
  - Caller must save them if they need to be preserved
  - The stack pointer is a special case (used for communication)

Local storage (memory)

- Procedures can allocate space on the stack for local variables
  - Subtract # of bytes needed from `%rsp`
  - Deallocate by restoring old `%rsp` value
- Variable-sized allocations require special handling
  - Use base / frame pointer (`%rbp`) to track “anchor” for current frame
  - Save previous base pointer on stack at beginning of function
  - Section 3.10.5 in CS 261 textbook



- Back to CS 430 …
  - Let’s generalize these concepts now

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/environment 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

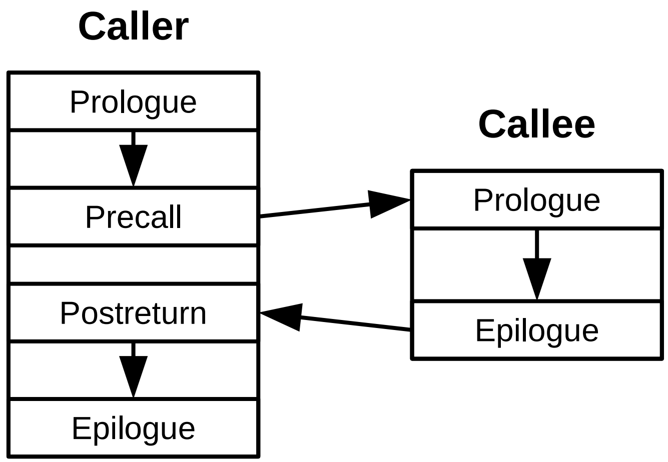
Linkage contract or calling convention

(caller and callee must agree)

Typical Conventions

- Caller: precall sequence
  - Evaluate and save parameters
  - Save return address
  - Transfer control to callee
- Callee: prologue sequence
  - Save & re-initialize base pointer
  - Allocate space for local variables
- Callee: epilogue sequence
  - De-allocate local variables
  - Restore saved base pointer
  - Transfer control back to caller
- Caller: postreturn sequence
  - De-allocate parameters

Note: The caller and/or callee may also need to save and restore other state (e.g., register values), depending on the specific system conventions and the needs of the caller/callee.



Non-local variables

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

Non-local variables

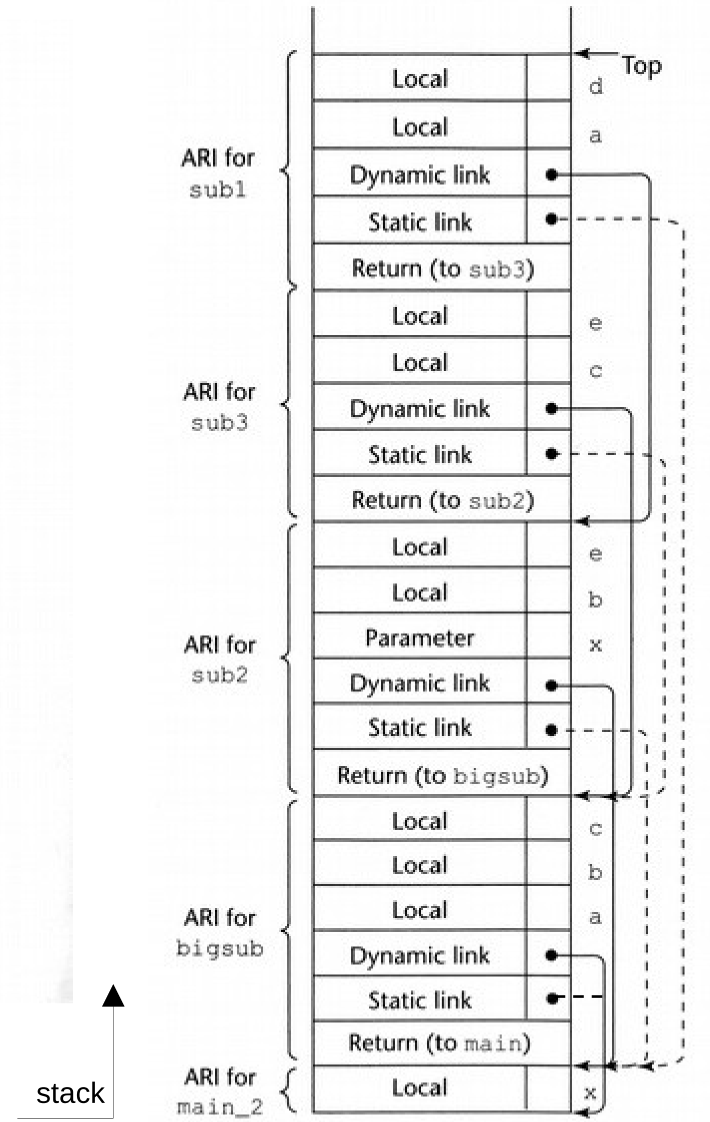
- Static scoping is simple without nested subprograms
  - Local variables are on the stack (track base pointer offsets)
  - Global variables are in static data (track addresses)
- Name resolution is harder with nested subprograms
  - Must be able to look up variables by **lexical** scope
- Primary method: static chains
  - Introduce a new static link
    - Similar to dynamic link, but points to most recent lexical parent
    - Created at runtime using nesting depth calculated at compile time
  - 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 and static links

- Dynamic link points to caller
  - Set link to previous EP at subprogram activation
  - Then move EP to base of new activation record
- Static link points to lexical parent
  - Set link based on subprogram location in code
  - If multiple instances of parent, use the most recent
- Name resolution
  - Use dynamic links for dynamic scoping and static links for static scoping

Example (from CPL)

```
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

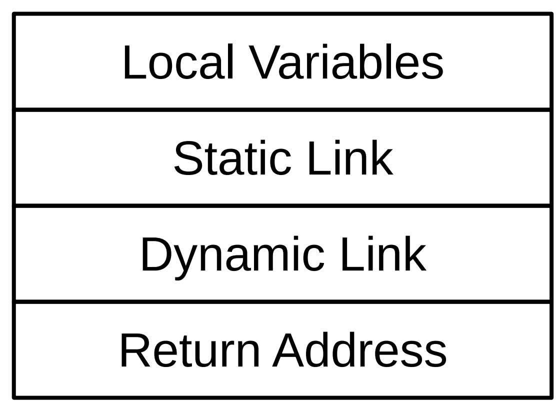
```
def P() {

    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.



Exercise

```
01  def P() {
02
03      var x = 'p'
04
05      def A() {
06          println(x)
07      }
08
09      def B() {
10          var x = 'b'
11          def C() {
12              var x = 'c'
13              println(x)
14              D()
15          }
16          def D() {
17              println(x)
18              A()
19          }
20          C()
21      }
22      B()
23  }
```

Trace this program using the activation record layout below.

Local Variables
Static Link
Dynamic Link
Return Address

OUTPUT

STATIC	DYNAMIC
c	c
b	c
p	c