The Stack & Procedures

CSE 351 Spring 2024

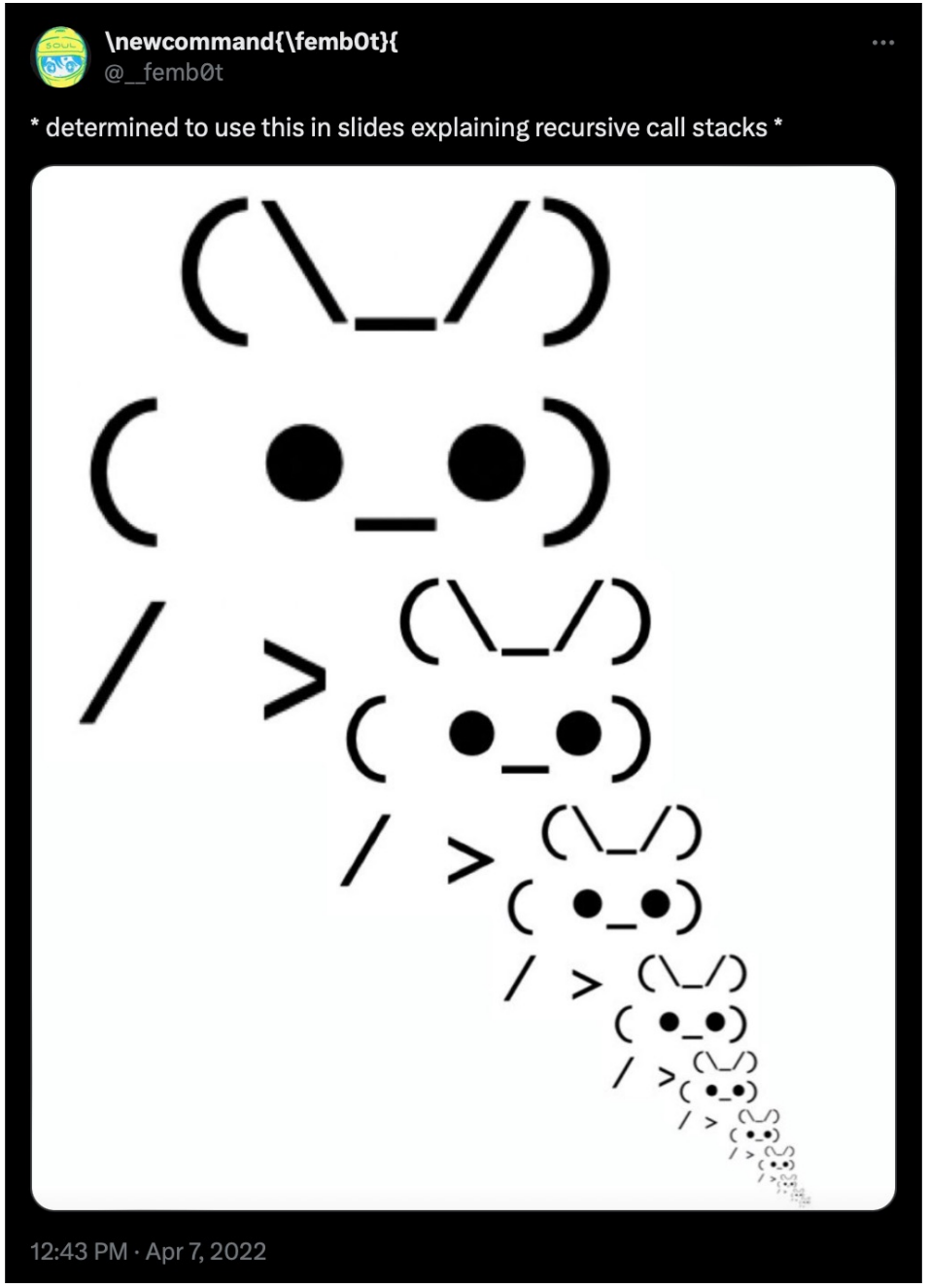
Instructor:

Elba Garza

Teaching Assistants:

Ellis Haker
Adithi Raghavan
Aman Mohammed
Brenden Page
Celestine Buendia
Chloe Fong
Claire Wang
Hamsa Shankar
Maggie Jiang
Malak Zaki
Naama Amiel
Nikolas McNamee
Shananda Dokka
Stephen Ying
Will Robertson

Playlist: CSE 351 24Sp Lecture Tunes!



Announcements, Reminders

- HW8 due tonight, HW9/10 due Friday
- Lab 2 due 26 Apr by 11:59 PM
  - GDB demo of phase 0 in section tomorrow!
  - Start like… **now**! Seriously, now.

x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- **Switches**

https://godbolt.org/z/Y9Kerb

```
long switch_ex
   (long x, long y, long z)
{
    long w = 1;
    switch (x) {
        case 1:
            w = y*z; break;
        case 2:
            w = y/z;
        /* Fall Through */
        case 3:
            w += z;  break;
        case 5:
        case 6:
            w -= z;  break;
        case 7:
            w = y%z; break;
        default:
            w = 2;
    }
    return w;
}
```

Switch Statement Example

- Multiple case labels
  - `case 5 & 6`, what's happening?
- Fall-through cases
  - `case 2`… no `break`?
- Missing cases
  - what about `case 4`?

- Implemented with:
  - Jump table, and
  - Indirect jump instruction

Jump Table Structure

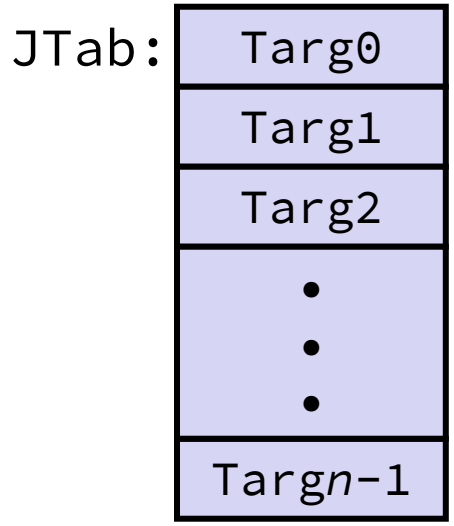
Switch Form

```
switch (x) {
  case val_0:
    <Block 0>
  case val_1:
    <Block 1>
    • • •
  case val_n-1:
    <Block n-1>
}
```

Approximate Translation

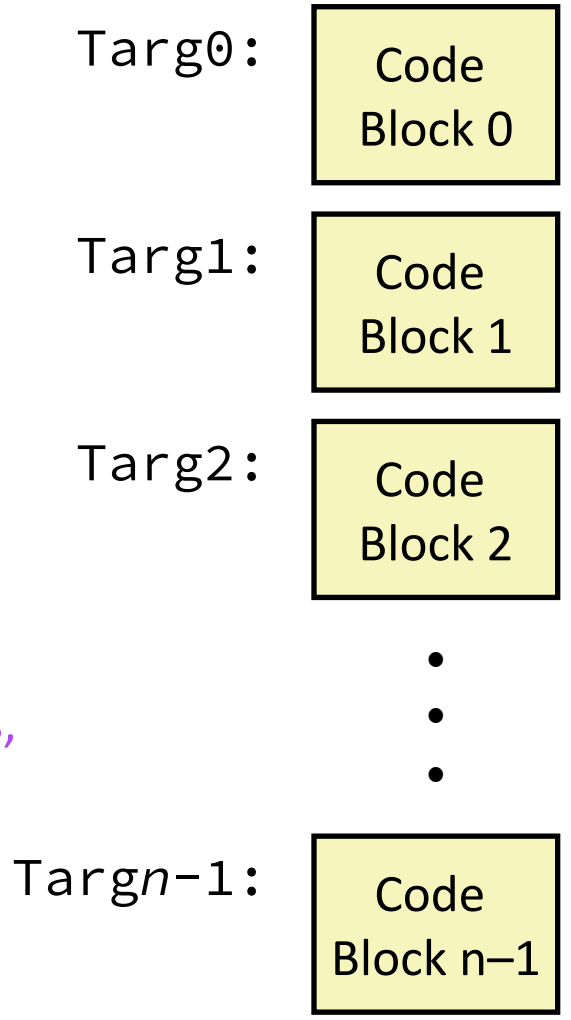
```
target = JTab[x];
goto target;
```

Jump Table



Essentially an array of pointers, stored in memory; instructors are data too, y'all!

Jump Targets



Jump Table Structure

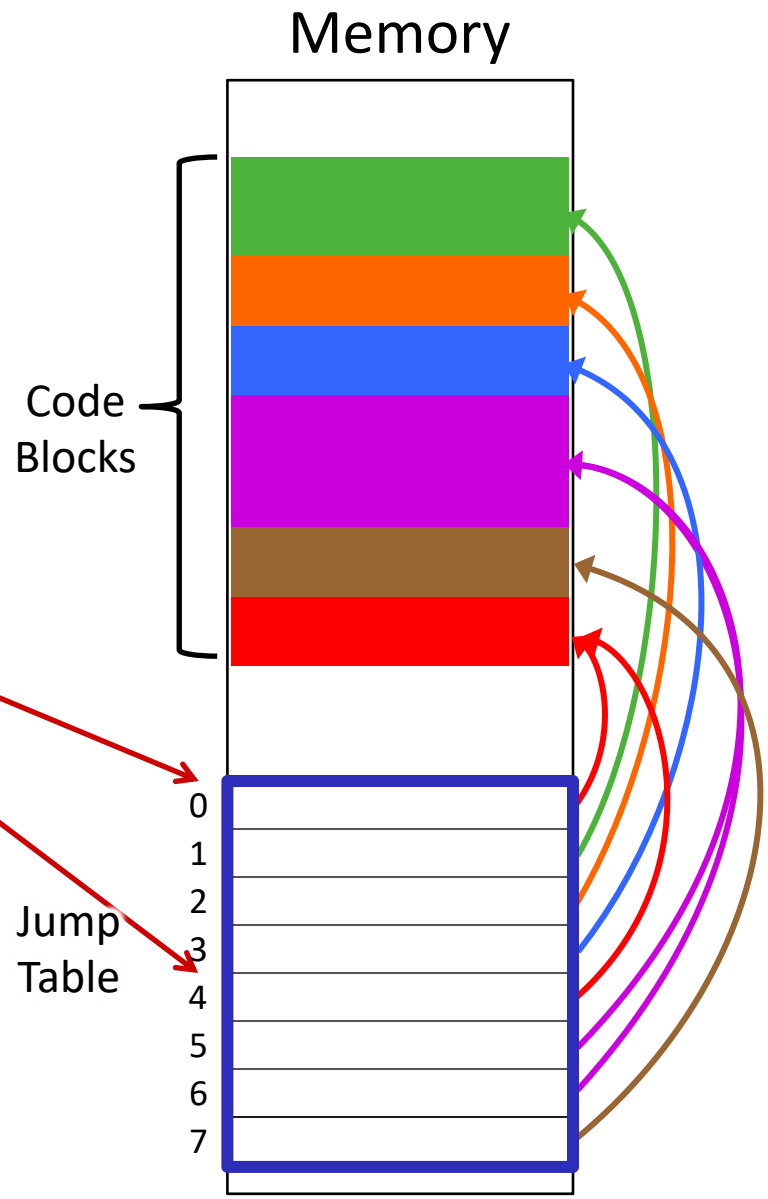
C code:

```
switch (x) {
  case 1: <code> break;
  case 2: <code>
  case 3: <code> break;
  case 5:
  case 6: <code> break;
  case 7: <code> break;
  default: <code>
}
```

Index the jump table array using the switch case number!

Use the jump table when x ≤ 7:

```
if (x <= 7)
  target = JTab[x];
  goto target;
else
  goto default;
```



Switch Statement Example

```
long switch_ex(long x, long y, long z)
{
    long w = 1;
    switch (x) {
      . . .
    }
    return w;
}
```

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	return value

Note that compiler chose to not initialize w

```
switch_ex:
    movq    %rdx, %rcx
    cmpq    $7, %rdi        # x:7
    ja      .L9             # default
    jmp     *.L4(,%rdi,8)   # jump table
```

Take a look!
https://godbolt.org/z/Y9Kerb

jump above – unsigned greater-than catches negative default cases too
e.g. "If x above 7 or less than 0, go to .L9"
why does this work? Any negative number, when evaluated as unsigned, looks big!

Switch Statement Example

```
long switch_ex(long x, long y, long z)
{
    long w = 1;
    switch (x) {
      . . .
    }
    return w;
}
```

```
switch_ex:
    movq    %rdx, %rcx
    cmpq    $7, %rdi        # x:7
    ja      .L9             # default
    jmp     *.L4(,%rdi,8)   # jump table
```

Indirect jump

[My research on this!]

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)
%rax	return value

Jump table

```
.section  .rodata
  .align 8
.L4:
  .quad    .L9    # x = 0
  .quad    .L8    # x = 1
  .quad    .L7    # x = 2
  .quad    .L10   # x = 3
  .quad    .L9    # x = 4
  .quad    .L5    # x = 5
  .quad    .L5    # x = 6
  .quad    .L3    # x = 7
```

.quad? Assembly code directive essentially saying "set aside 8 bytes". Will become addresses during compilation. Becomes array of pointers!

Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes (they're addresses!)
  - Base address at `.L4`

- **Direct jump**: `jmp .L9`
  - Jump target is denoted by label `.L9`

- **Indirect jump**: `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes!)
  - Fetch target from effective address `.L4 + x*8`
    - Only for $0 \le x \le 7$

Jump table

```
.section  .rodata
  .align 8
.L4:
  .quad   .L9    # x = 0
  .quad   .L8    # x = 1
  .quad   .L7    # x = 2
  .quad   .L10   # x = 3
  .quad   .L9    # x = 4
  .quad   .L5    # x = 5
  .quad   .L5    # x = 6
  .quad   .L3    # x = 7
```

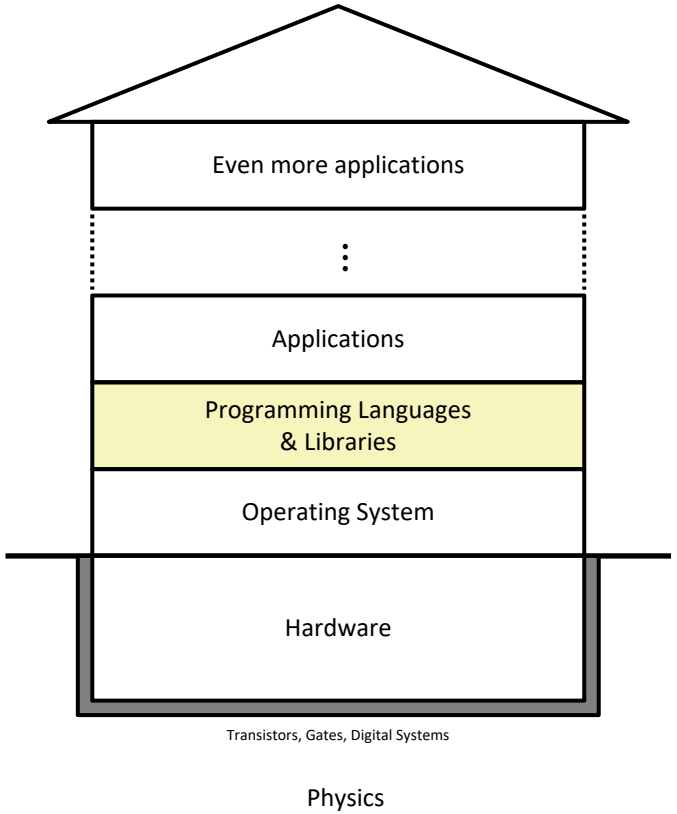
Directives: "set aside 8 bytes of space, please!"

Summary

- Control flow in x86 determined by Condition Codes
  - Showed **C**arry, **Z**ero, **S**ign, and **O**verflow, though others exist
  - Set flags with arithmetic instructions (implicit) or Compare and Test (explicit)
  - `Set` instructions read out flag values
  - Jump instructions use flag values to determine next instruction to execute
  - Most control flow constructs (*e.g.*, if-else, for-loop, while-loop, switch) can be implemented in assembly using combinations of conditional and unconditional jumps

The Hardware/Software Interface

- Topic Group 2: **Programs**
  - x86-64 Assembly, **Procedures**, **Stacks**, Executables



- How are programs created and executed on a CPU?
  - How does your source code become something that your computer understands?
  - How does the CPU organize and manipulate local data?

Reading Review

- Terminology:
  - Stack, Heap, Static Data, Literals, Code
  - Stack pointer (`%rsp`), `push`, `pop`
  - Caller, callee, return address, `call`, `ret`
    - Return value: `%rax`
    - Arguments: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
  - Stack frames and stack discipline

Review Questions

- How does the stack change after executing the following instructions?

```
pushq %rbp
subq  $0x18, %rsp
```

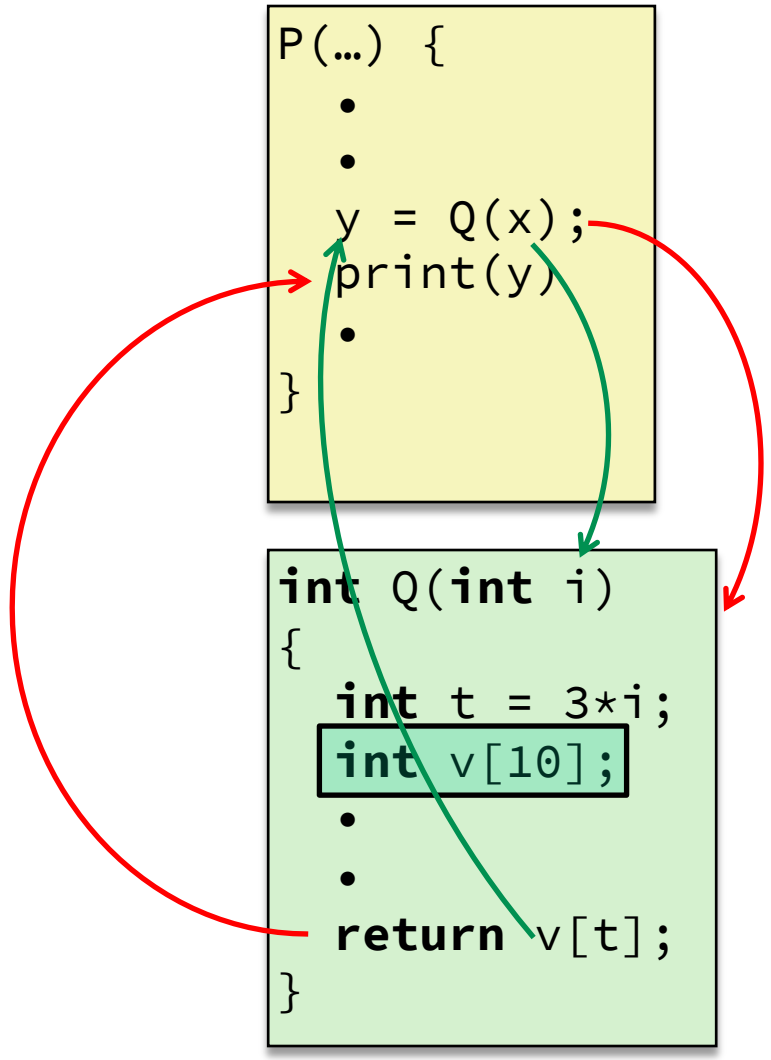
- For the following function, which registers do we know must be used?

```
void* memset(void* ptr, int value, size_t num);
```

Mechanisms required for *procedures*

1) Passing control
   - To beginning of procedure code
   - Back to return point
2) Passing data
   - Procedure arguments
   - Return value
3) Memory management
   - Allocate during procedure execution
   - De-allocate upon return

❖ All implemented with machine instructions!
   - An x86-64 procedure uses only those mechanisms required for that procedure



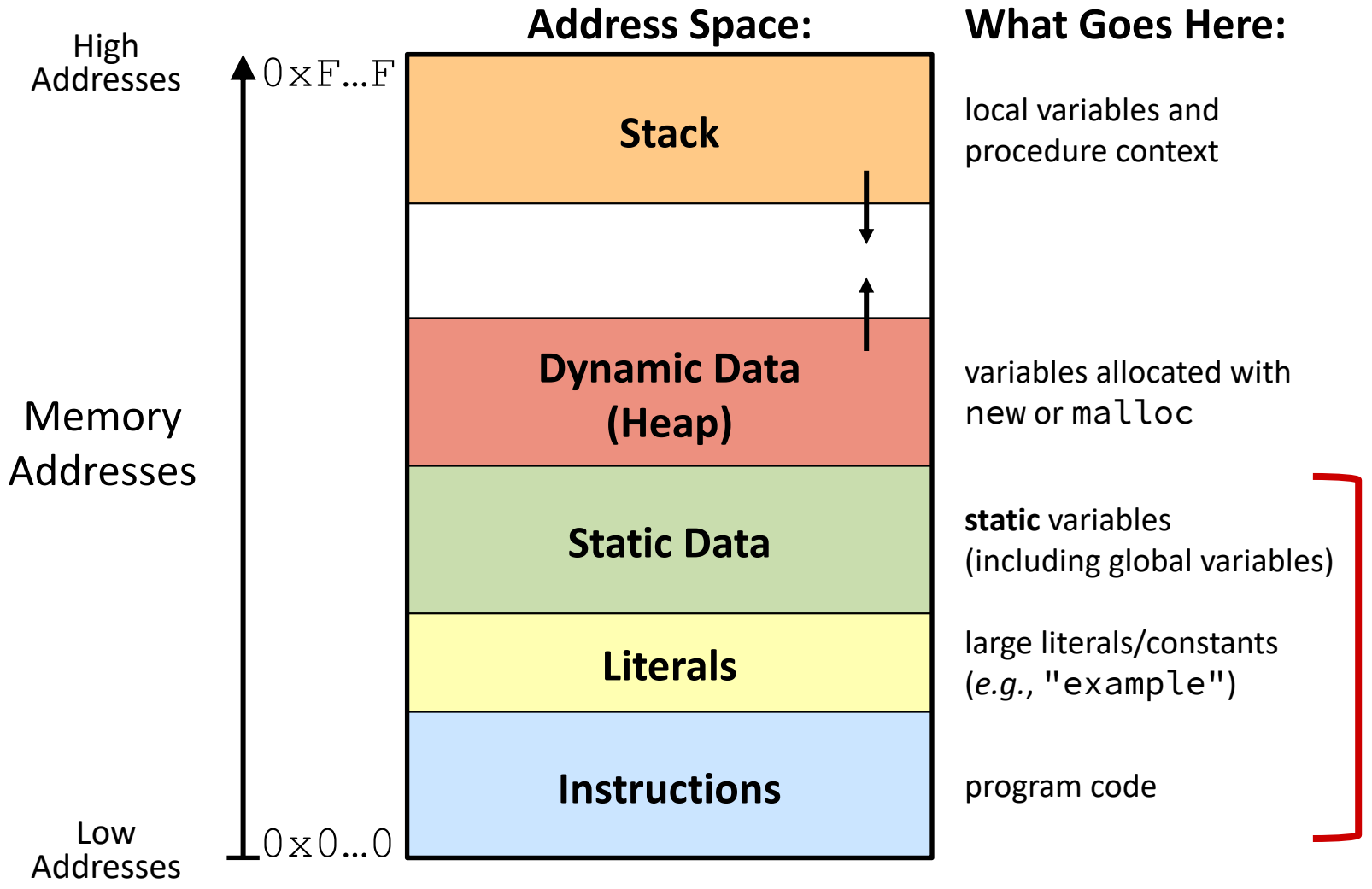
How? Manipulate stack, manipulate registers, etc.

Procedures

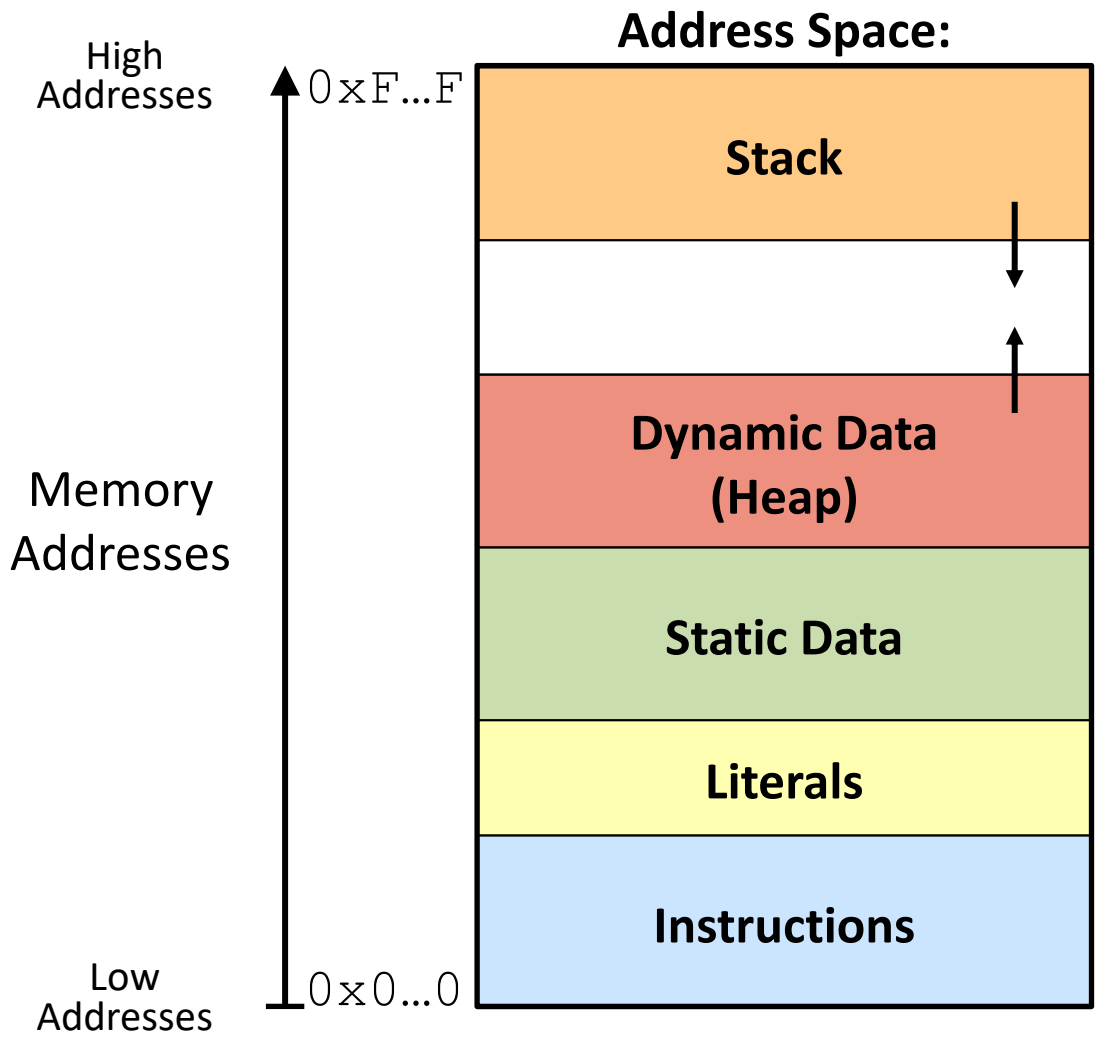
- **Stack Structure**
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Simplified Memory Layout (Review)

So kinda been lying to you: Up till now, been saying "you'll find that value somewhere in memory", but it's actually more organized than that…



Memory Management



Who's Responsible:

Managed "automatically" (by compiler/assembly)

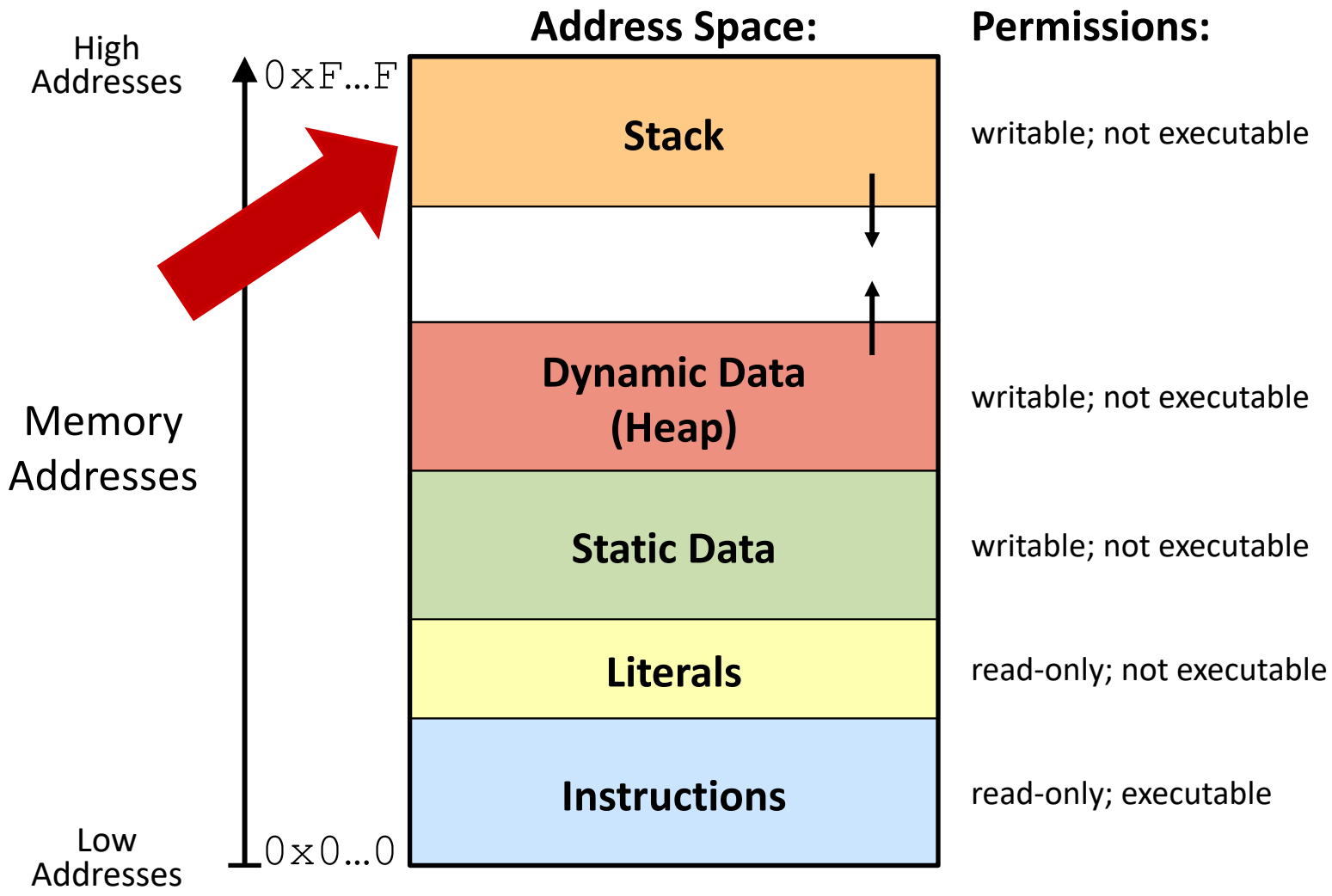
Managed "dynamically" (by programmer)

Managed "statically" (initialized when process starts)

Managed "statically" (initialized when process starts)

Managed "statically" (initialized when process starts)

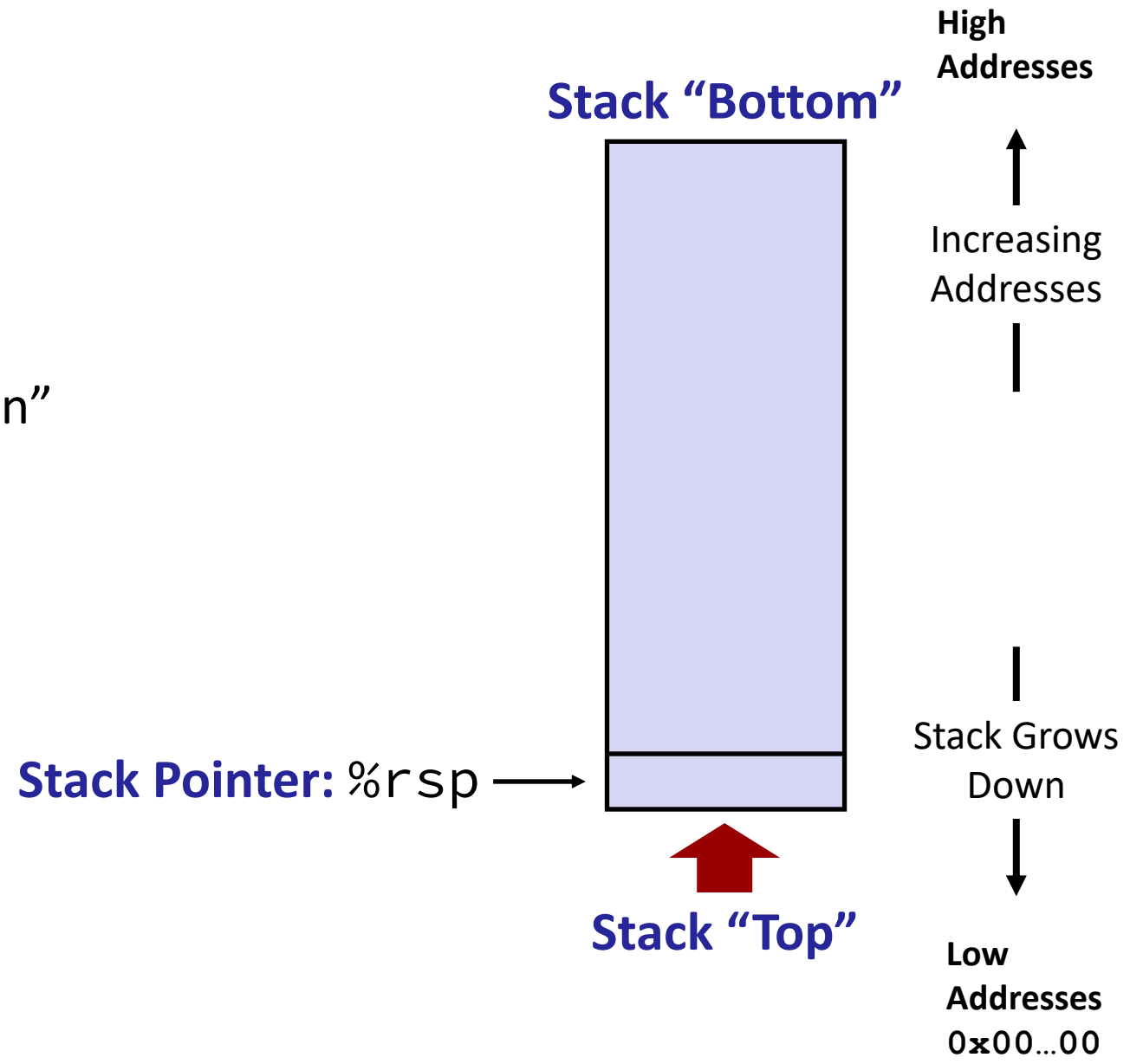
Memory Permissions



- Segmentation fault: **impermissible** memory access; not just "oops, bad address"!

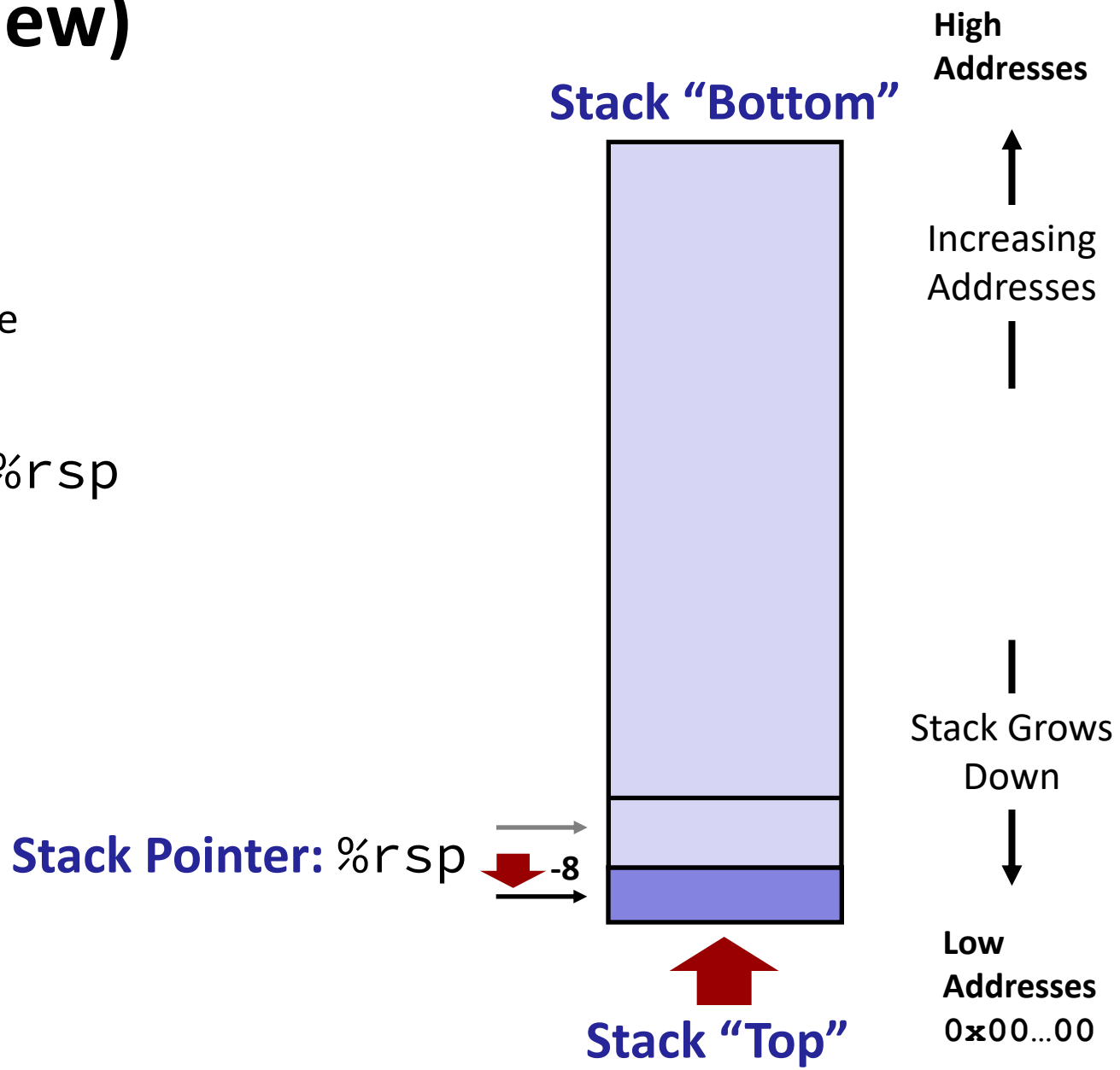
x86-64 Stack (Review)

- Region of memory managed with stack "discipline"
  - Grows toward lower addresses
  - Customarily shown "upside-down"
- Register `%rsp` contains *lowest* stack address
  - `%rsp` = address of *top* element, the most-recently-pushed item that is not-yet-popped



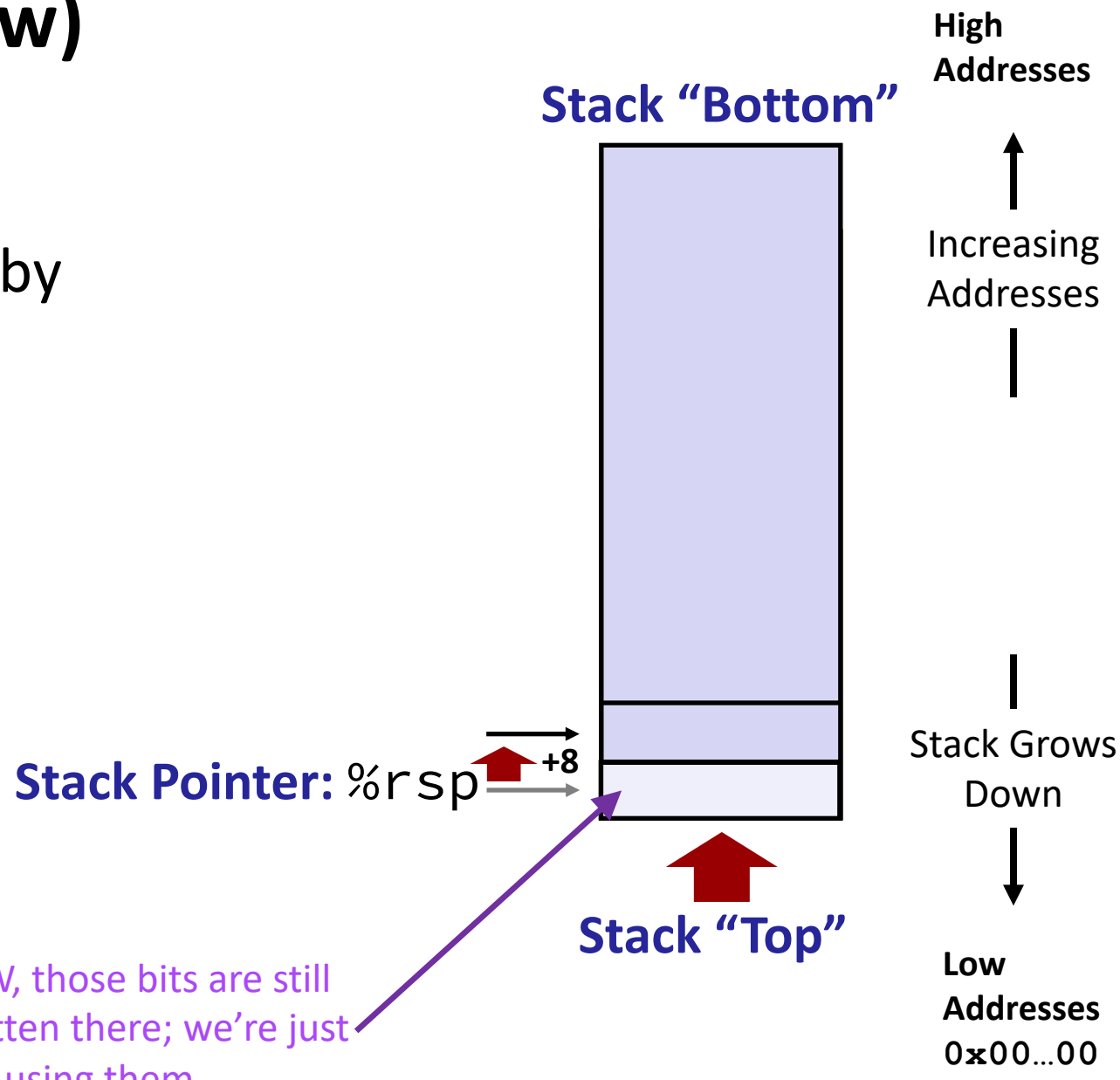
x86-64 Stack: Push (Review)

- `pushq` *src*
  - Fetch operand at src
    - Src can be reg, memory, immediate
  - ***Decrement*** `%rsp` by 8
  - Store value at address given by `%rsp`
- Example:
  - **`pushq %rcx`**
  - 1) Adjust `%rsp`, and
    2) store contents of
    `%rcx` on the stack



x86-64 Stack: Pop (Review)

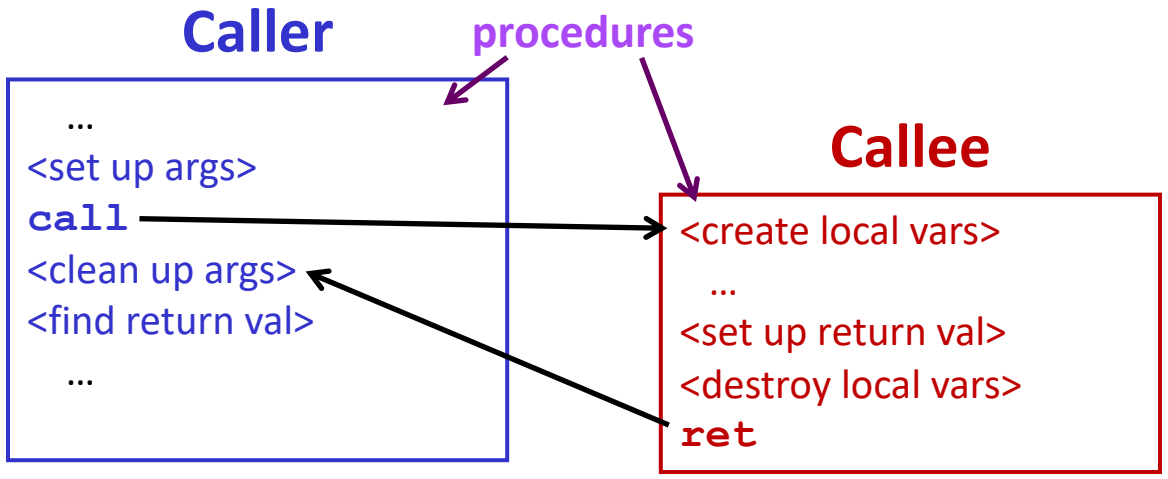
- `popq` *dst*
  - Load value at address given by `%rsp`
  - Store value at *dst*
  - ***Increment*** `%rsp` by 8
- Example:
  - **`popq %rcx`**
  - 1) Store contents of top of stack into `%rcx`, and 2) adjust `%rsp`



Procedures

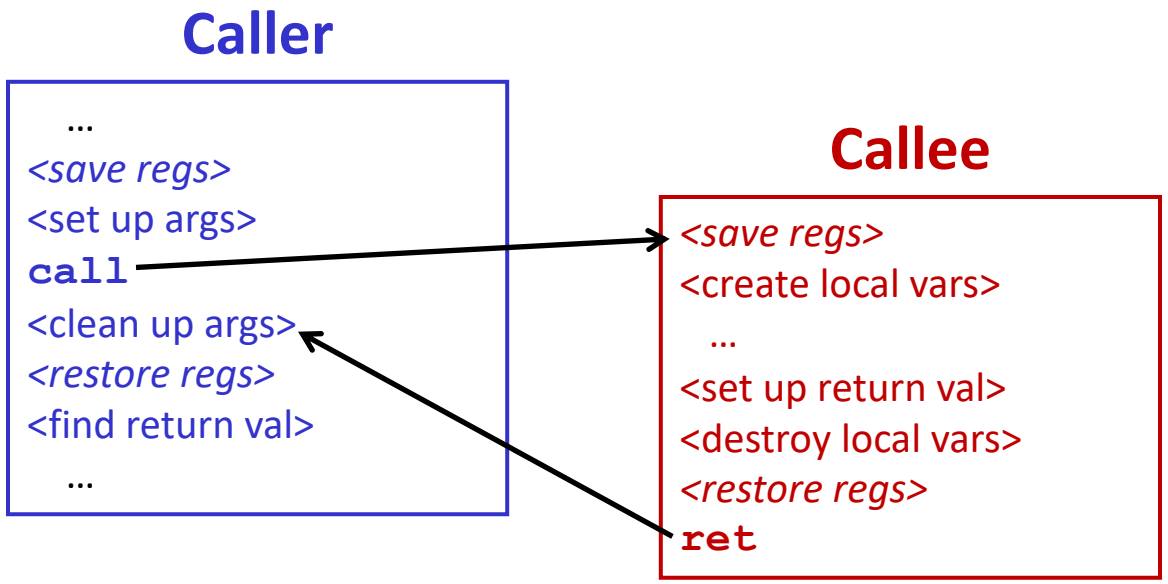
- Stack Structure
- **Calling Conventions**
  - **Passing control**
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Procedure Call Overview



- **Callee** must know where to find args
- **Callee** must know where to find *return address*
- **Caller** must know where to find *return value*
- **Caller** and **Callee** run on same CPU, so use the same registers
  - How do we deal with register reuse?
- Unneeded steps can be skipped (*e.g.*, no arguments)

Procedure Call Overview



- The *convention* of where to leave/find things is called the calling convention (or **procedure call linkage**)
  - Details vary between operating systems
  - We will see the convention for x86-64 on Linux in detail
  - What could happen if our program didn't follow these conventions?

Code Example (Preview)

Compiler Explorer:
https://godbolt.org/z/ndro9E

Multstore Code

```
// Mult two numbers & store result in dest ptr
void multstore (long x, long y, long *dest) {
    long t = mult2(x, y); // call mult2
    *dest = t;
}
```

Mult2 Code

```
long mult2(long a, long b) {
  long s = a * b;
  return s;
}
```

Multstore Disassembly

```
0000000000400540 <multstore>:
  400540: push   %rbx                  # Save %rbx
  400541: movq   %rdx,%rbx             # Save dest
  400544: call   400550 <mult2>        # mult2(x,y)
  400549: movq   %rax,(%rbx)           # Save at dest
  40054c: pop    %rbx                  # Restore %rbx
  40054d: ret                          # Return
```

Mult2 Disassembly

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax   # a
  400553:  imulq  %rsi,%rax   # a * b
  400557:  ret                # Return
```

Instruction addresses!

Procedure Control Flow (Review)

- Use stack to support procedure call and return
- Procedure call: `call label`
  1) Push "return address" on stack (Which address is it in example?)
  2) Jump to ***label***
- Return address:
  - Address of instruction immediately after **`call`** instruction
  - Example from disassembly:

```
400544: call   400550 <mult2>
400549: movq   %rax,(%rbx)
```

Return address = **0x400549**

next instruction happens to be a move, but could be anything!

- Procedure return: `ret`
  1) Pop return address from stack
  2) Jump to address

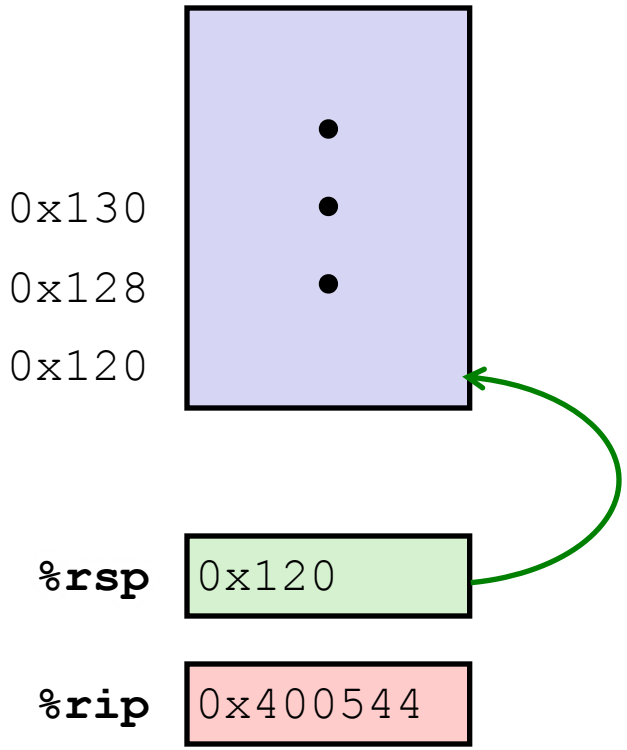
Procedure Call Example (step 1)

Multstore Disassembly

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

Mult2 Disassembly

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```



1) Push the return address onto the stack, and
2) Jump to new call!

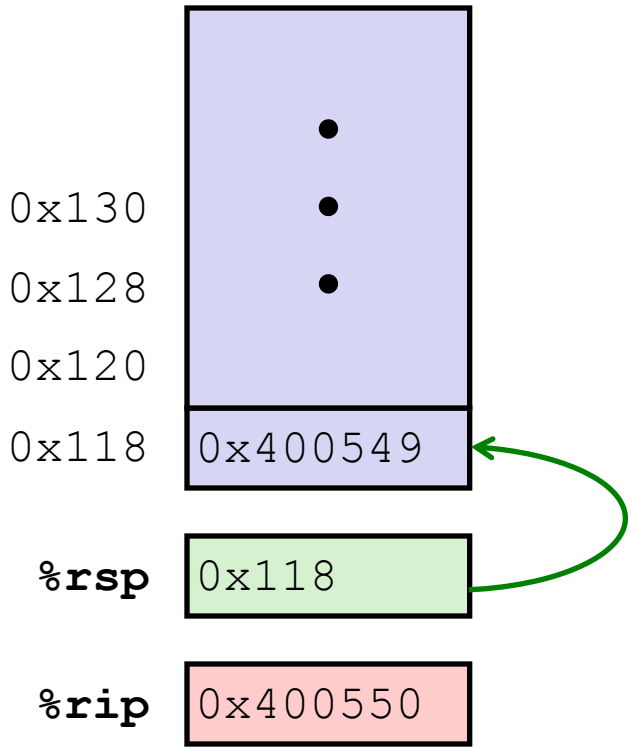
Procedure Call Example (step 2)

Multstore Disassembly

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

Mult2 Disassembly

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```



Can now let mult2 do its thing…

Procedure Return Example (step 1)

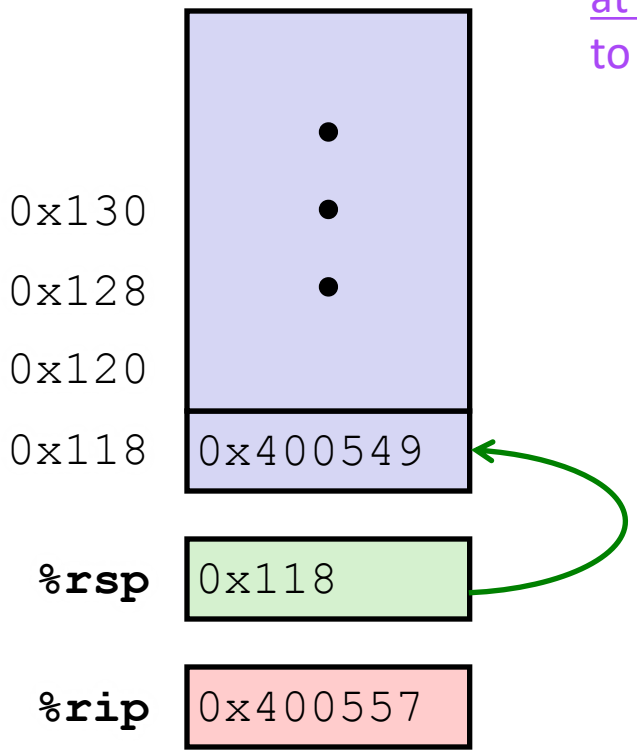
Note: we expect mult2 to "clean up after itself" i.e. stack is left as it was at the beginning of call to mult2

Multstore Disassembly

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

Mult2 Disassembly

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```



Once mult2 is done…

1) Pop the return address from the stack, and

2) Jump to it, i.e. update %rip to return address.

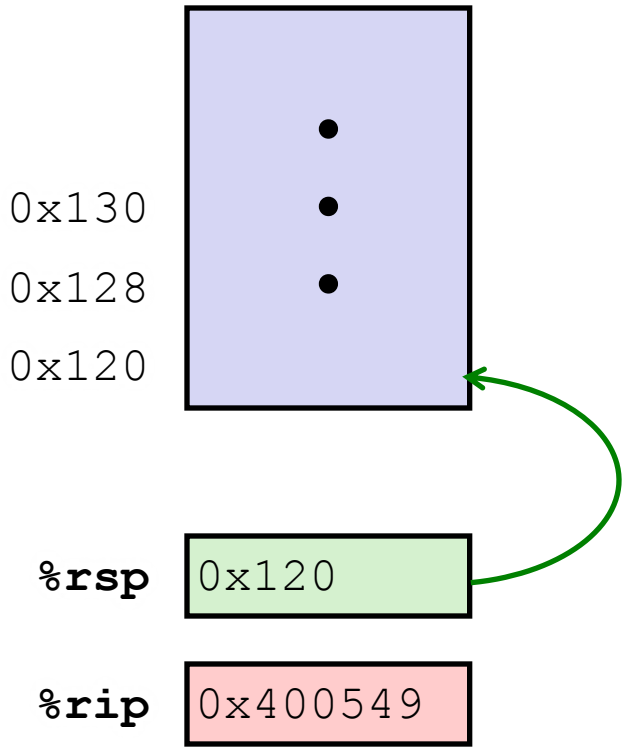
Procedure Return Example (step 2)

Multstore Disassembly

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

Mult2 Disassembly

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```



Done! Called and returned. Note that %rip now holds value of instruction after the call, and stack looks as before the call.

Procedures

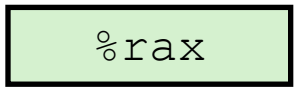
- Stack Structure
- **Calling Conventions**
  - Passing control
  - **Passing data**
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Procedure Data Flow (Review)

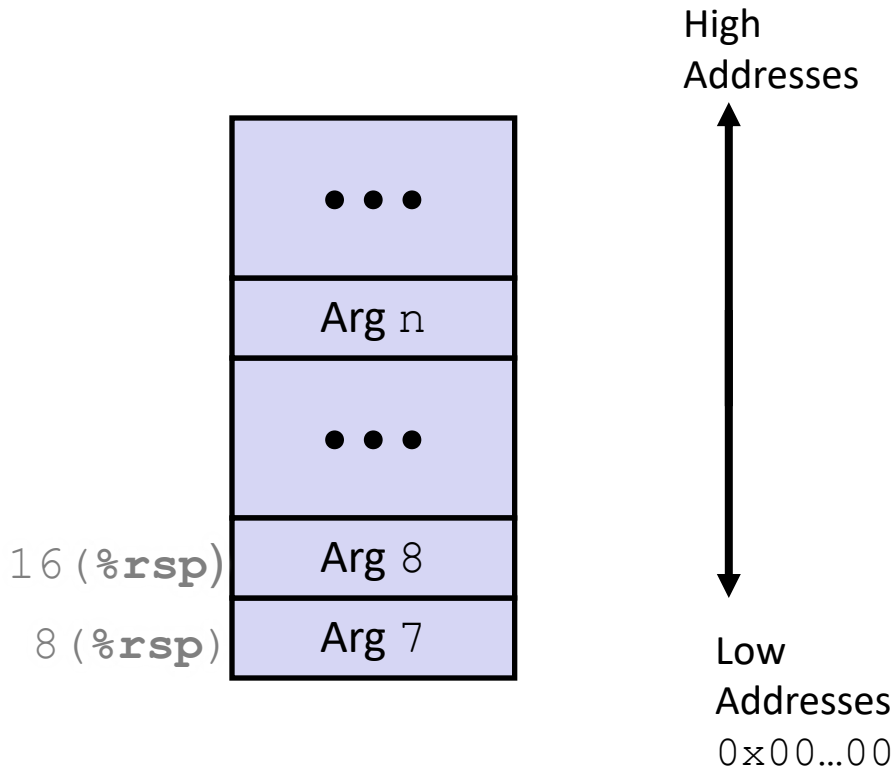
Registers (NOT in Memory)

- First 6 arguments

- Return value



Stack (Memory)



What if more than 6 arguments? Use the stack! Note the reverse order. Only allocate stack space when needed…

x86-64 Return Values

- By convention, values returned by procedures are placed in `%rax`
  - Choice of `%rax` is arbitrary
  - What if both caller and callee return values, though?

1) Caller must make sure to save the contents of `%rax` before calling a callee that returns a value
   - Part of register-saving convention
2) Callee places return value into `%rax`
   - Any type that can fit in 8 bytes – integer, float, pointer, etc.
   - For return values greater than 8 bytes, best to return a *pointer* to them
3) Upon return, caller finds the return value in `%rax`

Data Flow Examples

Multstore Code

```
// Multiply two numbers & store in dest ptr
void multstore (long x, long y, long *dest) {
    long t = mult2(x, y); // call mult2
    *dest = t;
}
```

Multstore Disassembly

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  400541: movq    %rdx,%rbx       # Save dest
  400544: call    400550 <mult2>  # mult2(x,y)
  # t in %rax
  400549: movq    %rax,(%rbx)     # Save at dest
```

Mult2 Code

```
long mult2(long a, long b) {
  long s = a * b;
  return s;
}
```

Mult2 Disassembly

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  movq    %rdi,%rax  # a
  400553:  imulq   %rsi,%rax  # a * b
 # s in %rax
  400557:  ret                # Return
```

x ↔ a and y ↔ b, but what about *dest?
Need to save it for after mult2 returning.

Procedures

- Stack Structure
- **Calling Conventions**
  - Passing control
  - Passing data
  - **Managing local data**
- Register Saving Conventions
- Illustration of Recursion

Stack-Based Languages

- Languages that support recursion
  - *e.g.*, C, Java, most modern languages
  - Code must be *re-entrant*
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store *state* of each instantiation
    - Arguments, local variables, return address
- Stack allocated in *frames*
  - State for a single procedure instantiation
- Stack discipline
  - State for a given procedure needed for a limited time
    - Starting from when it is called to when it returns
  - Callee always returns before caller does

Call Chain Example

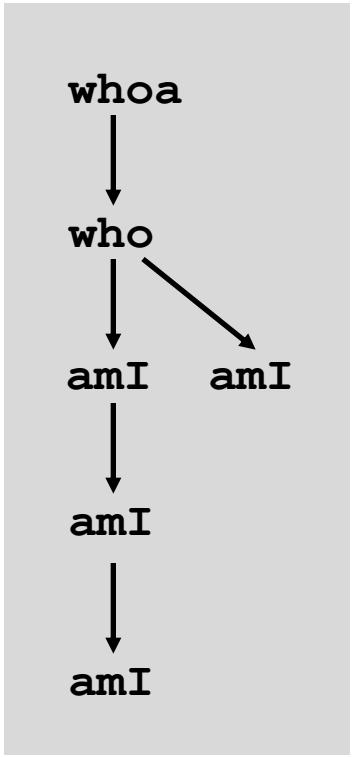
```
whoa(…)
{
    •
    •
    who();
    •
    •
}
```

```
who(…)
{
    •
    amI();
    •
    amI();
    •
}
```

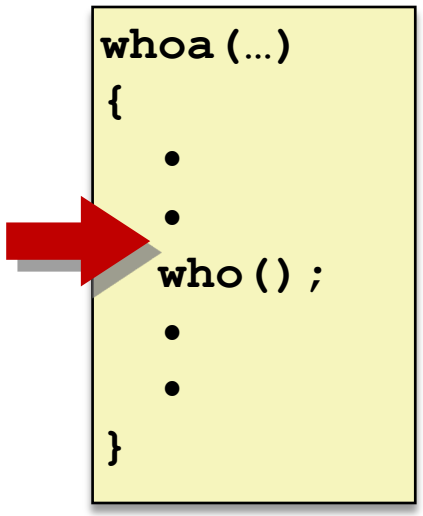
```
amI(…)
{
    •
  if(…){
    amI()
  }
    •
}
```

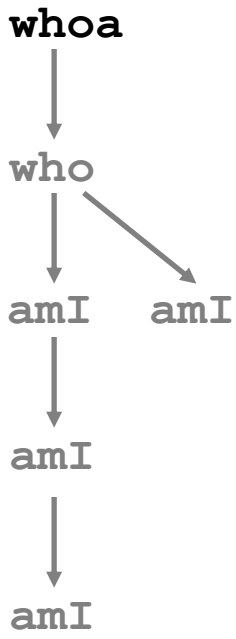
Procedure `amI` is recursive (calls itself)

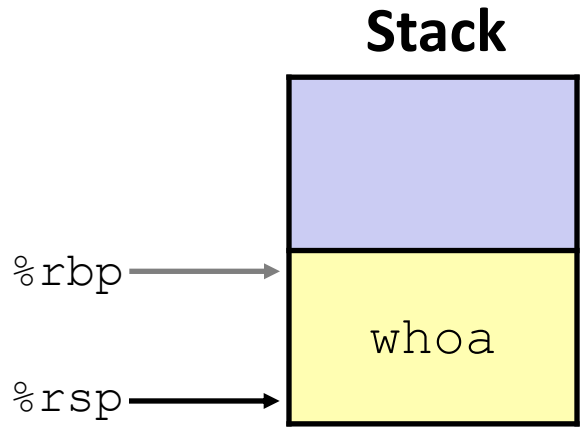
Example Call Chain



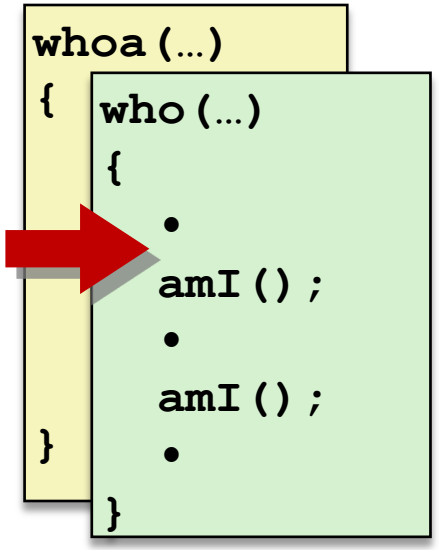
1) Call to whoa

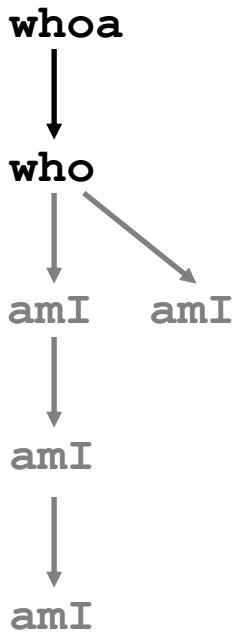


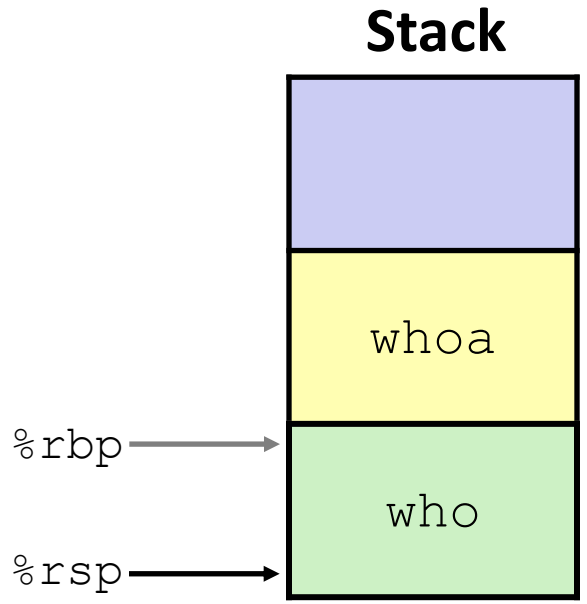




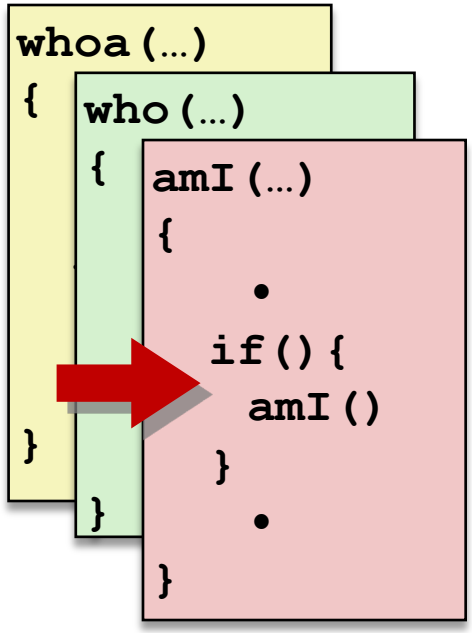
2) Call to who

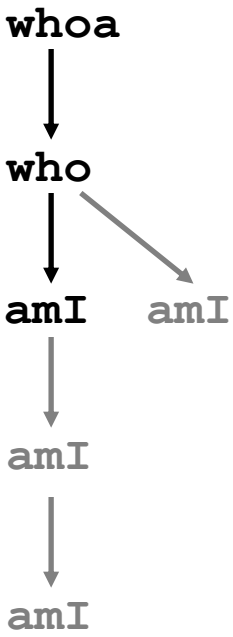

whoa(...)
{
who(...)
{
•
amI();
•
amI();
•
}
}

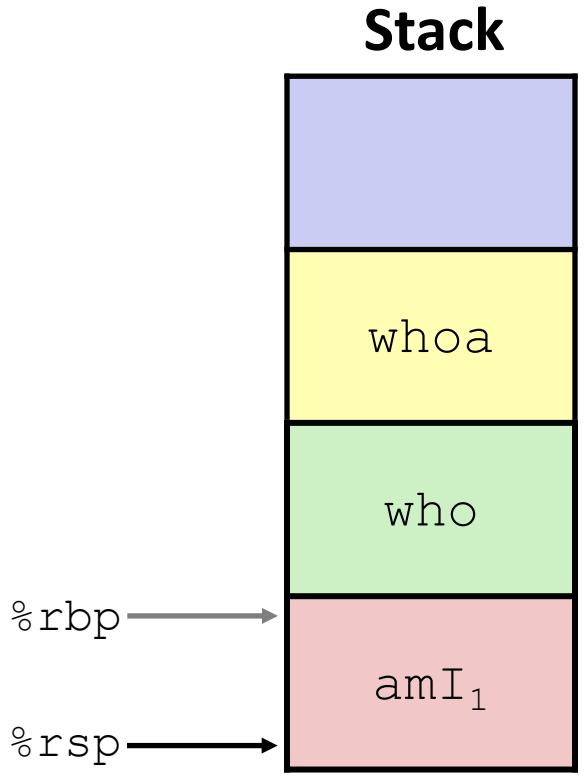

whoa
who
amI
amI
amI
amI


Stack
whoa
%rbp
who
%rsp

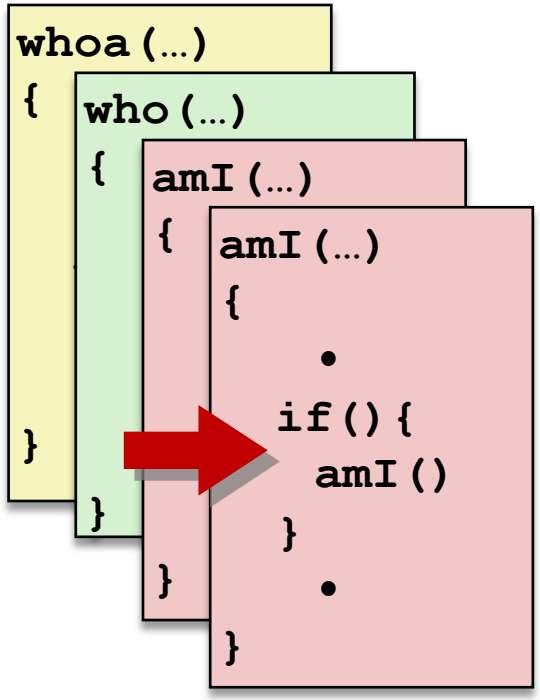
3) Call to amI (1)

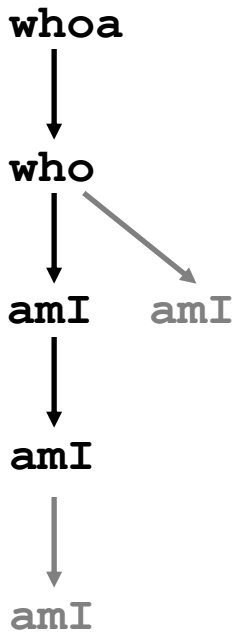


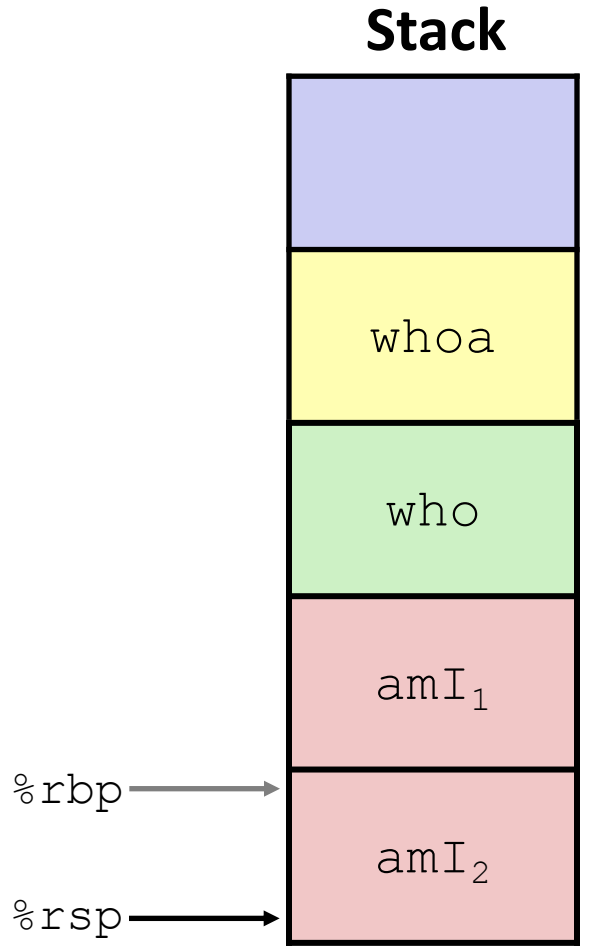




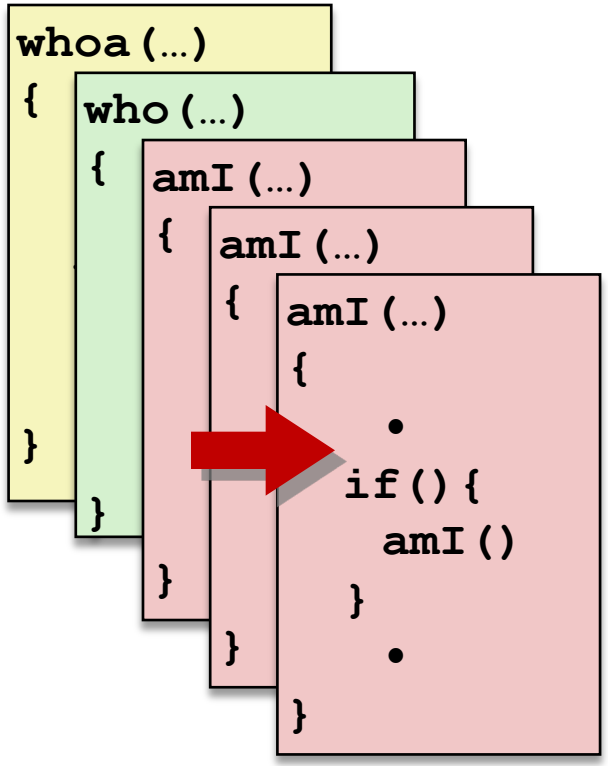
4) Recursive call to amI (2)

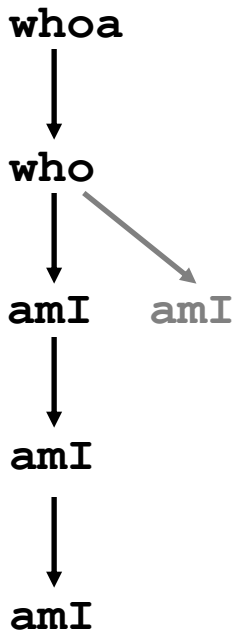


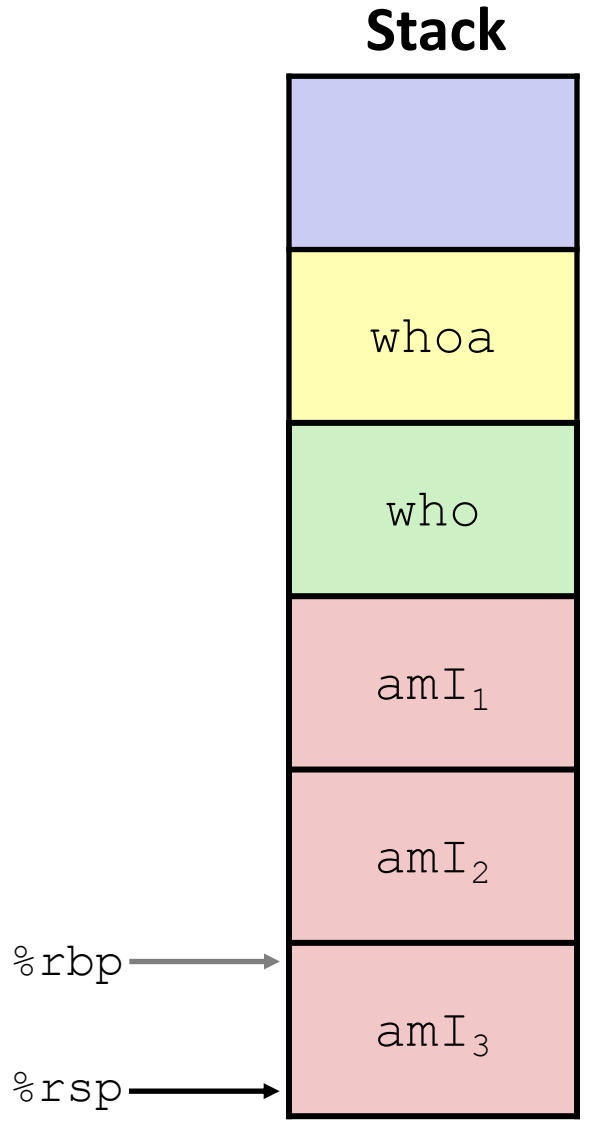




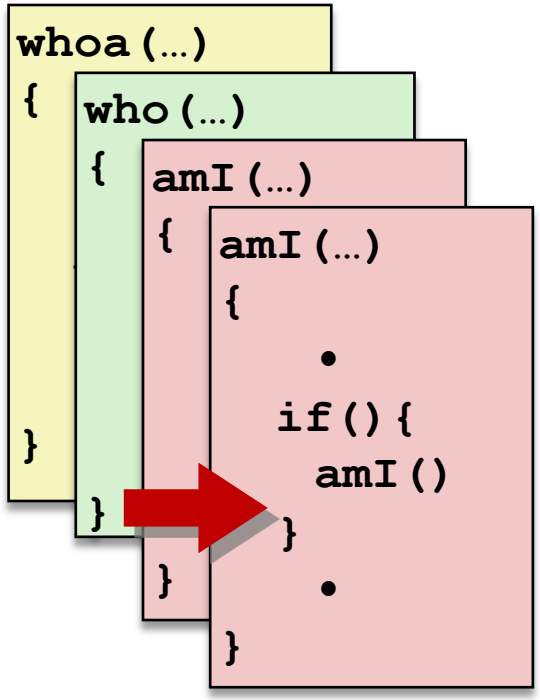
5) (another) Recursive call to amI (3)

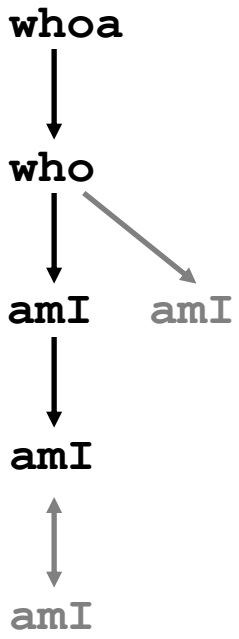


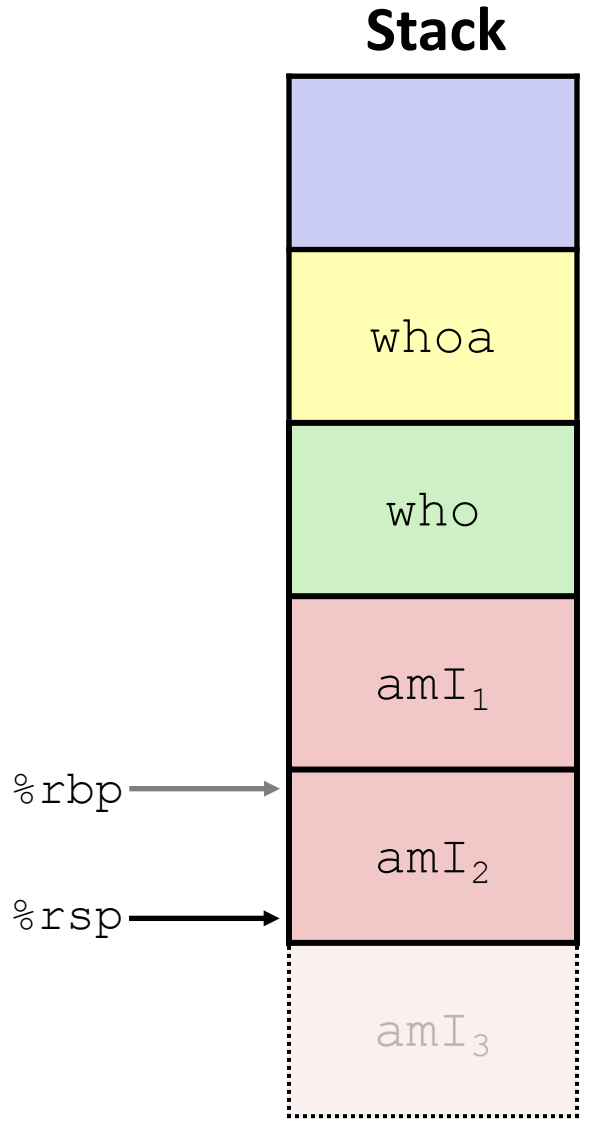




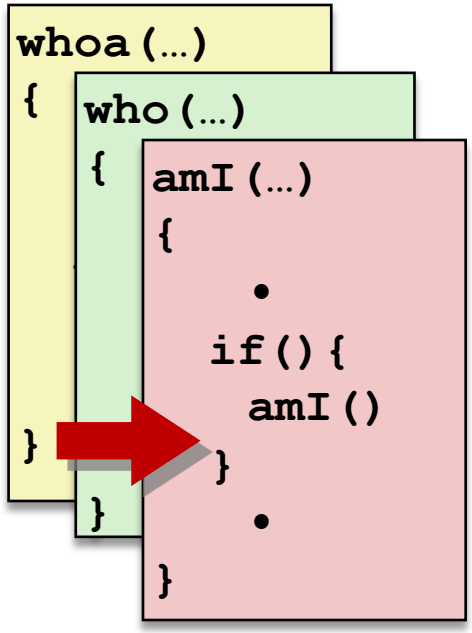
6) Return from (another) recursive call to amI

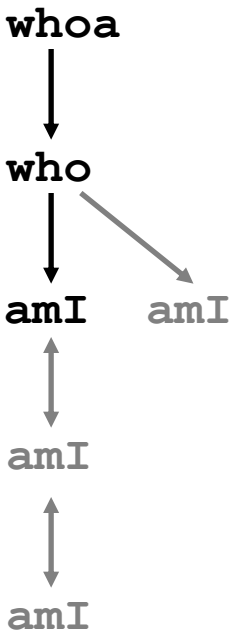


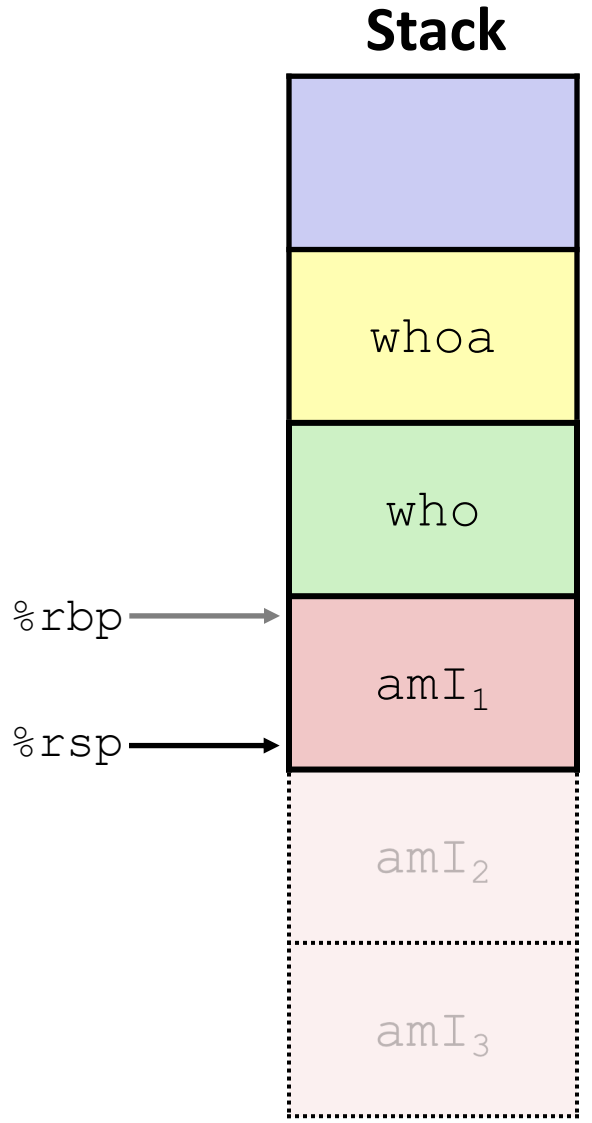




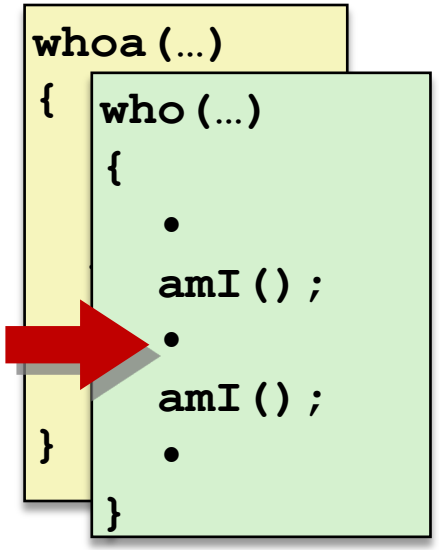
7) Return from recursive call to amI

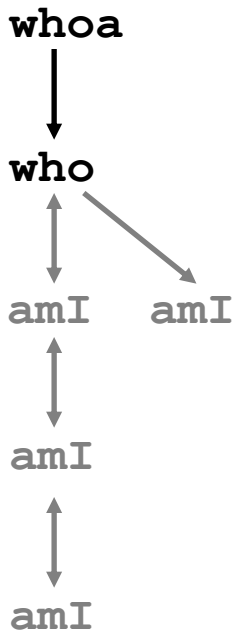


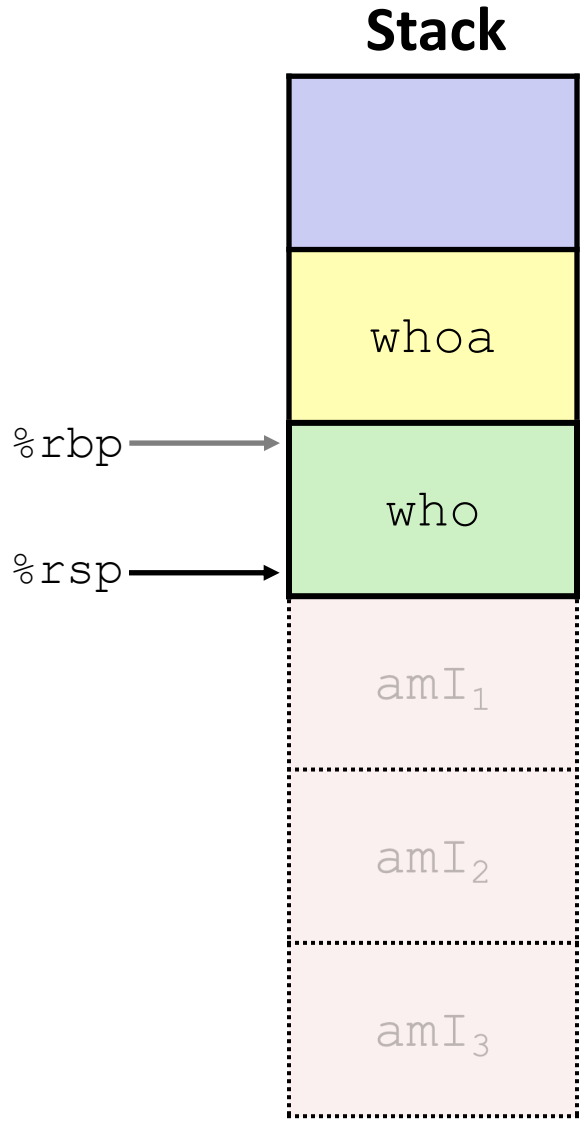




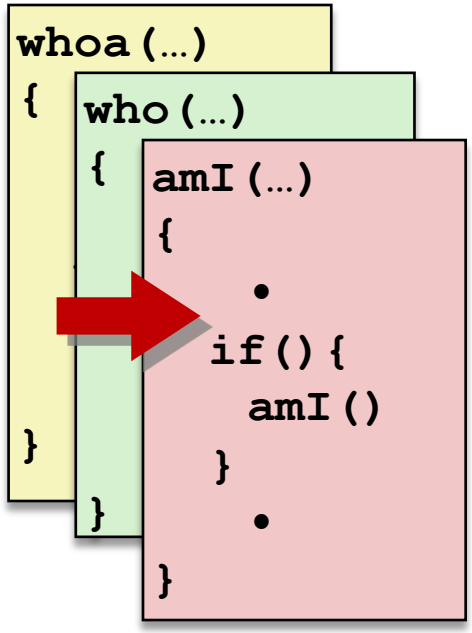
8) Return from call to amI

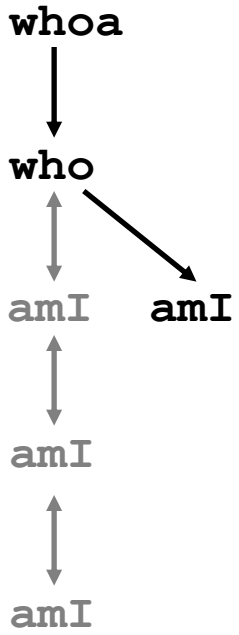


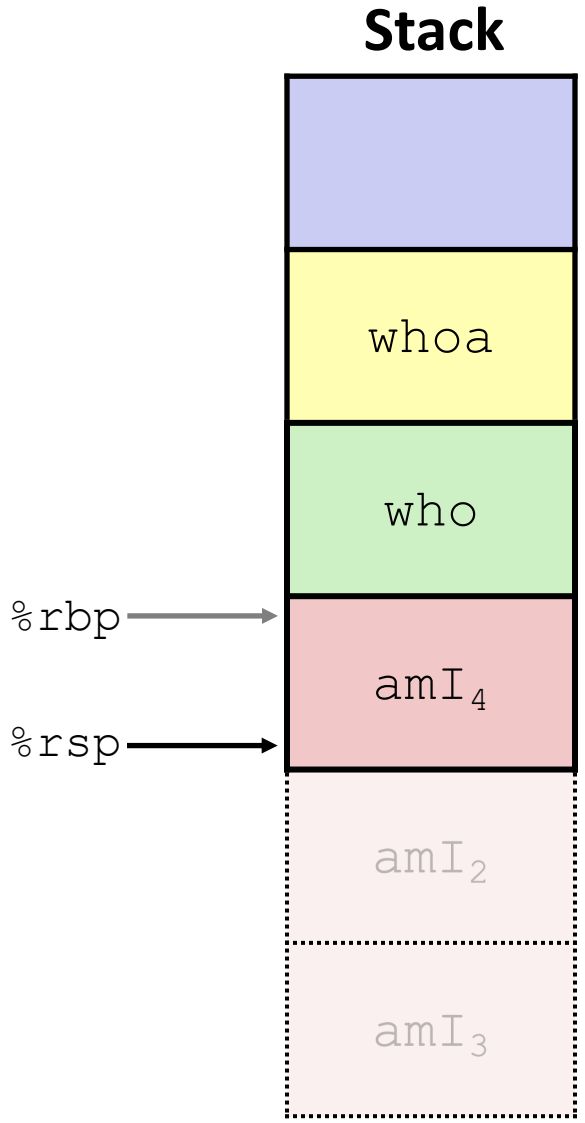




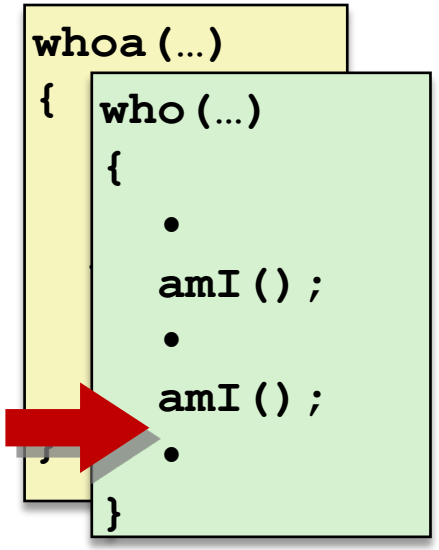
9) (second) Call to amI (4)

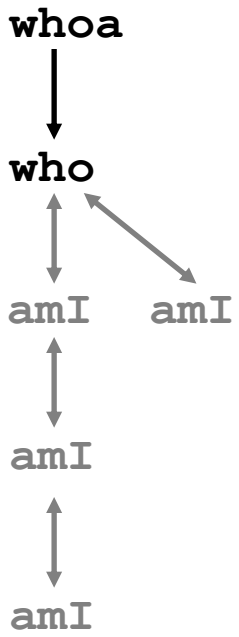


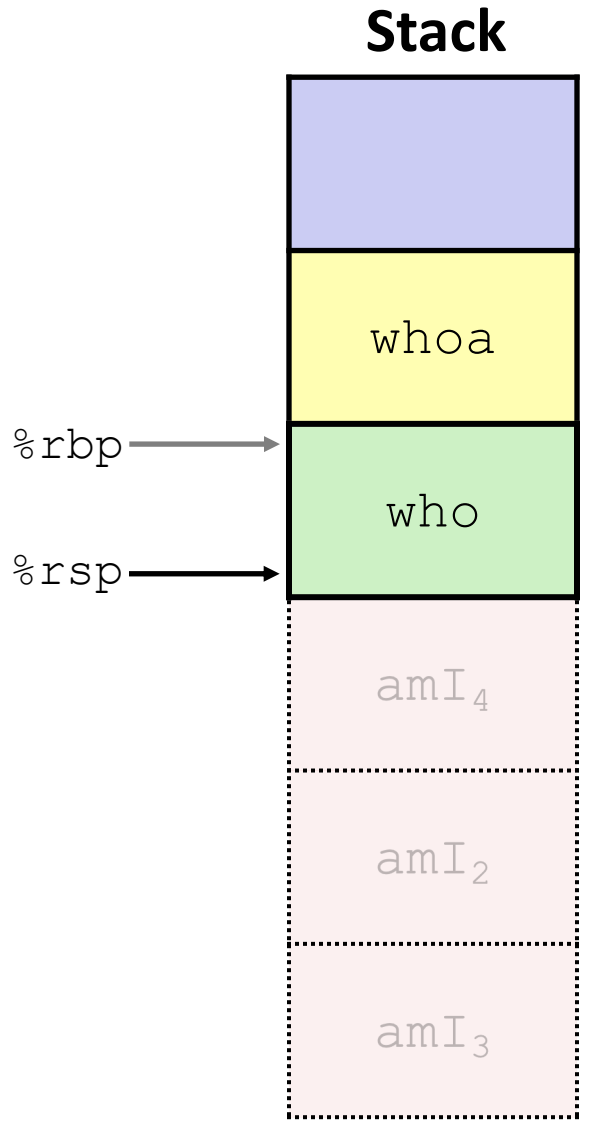




10) Return from (second) call to amI







11) Return from call to who

```
whoa(...)
{
    •
    •
    who();
    •
    •
}
```