

The Stack & Procedures

CSE 351 Summer 2022

Instructor:

Kyrie Dowling

Teaching Assistants:

Aakash Srazali

Allie Pflieger

Ellis Haker

↳ You Retweeted



Senior Oops Engineer @ReinH · Feb 28, 2019

I am a full stack engineer which means if you give me one more task my stack will overflow

42

↳ 2.2K

6.8K



Relevant Course Information

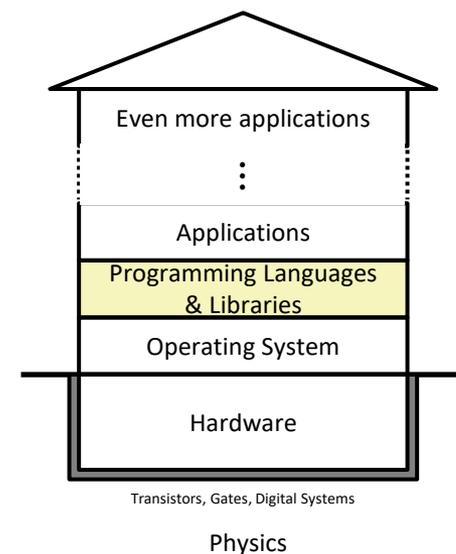
- ❖ hw8 due tonight, hw9 due Monday
- ❖ Lab 2 due next Friday (7/22)
 - See GDB Tutorial and Phase 1 walkthrough in Section 4 Lesson on Ed
- ❖ Unit Portfolio 1 due tonight at 11:59 pm
 - No late days!

x86 Control Flow

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

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

- ❖ Questions from the Reading?

Review Questions

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

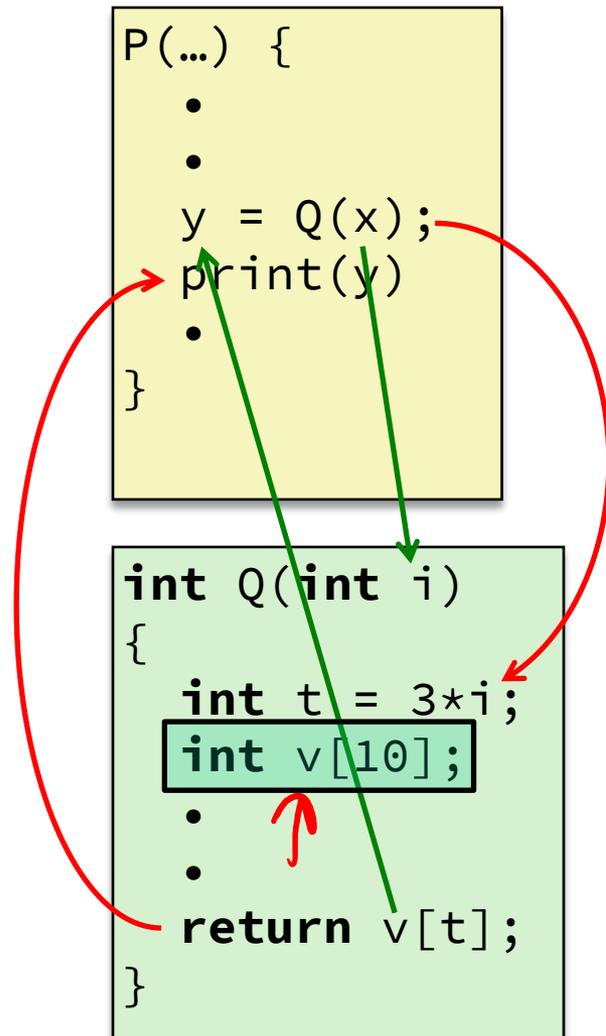
```
pushq %rbp  
subq $0x18, %rsp
```

- ❖ 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*

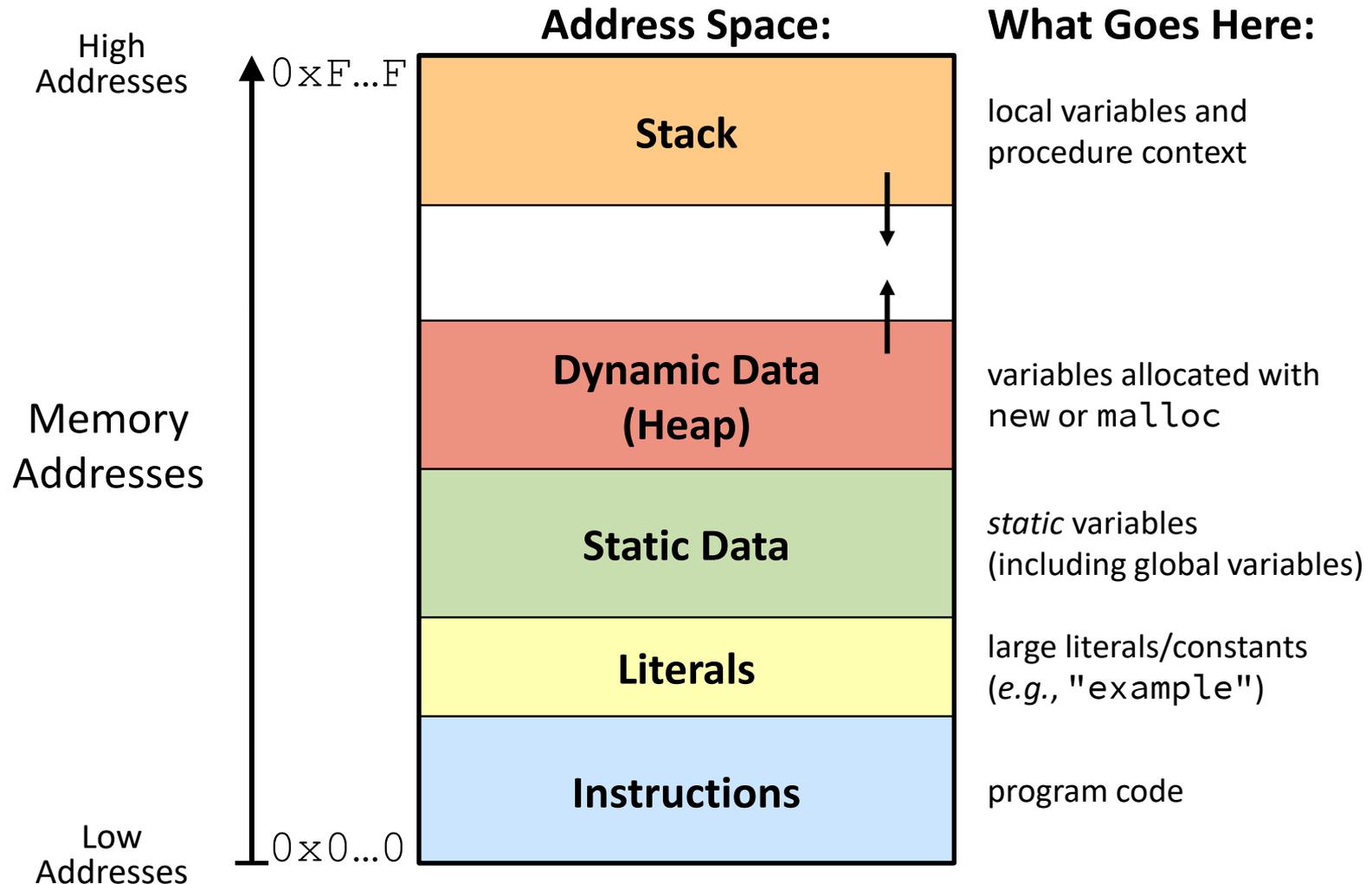
- 1) Passing control
 - To beginning of procedure code
 - Back to return point
 - 2) Passing data
 - Procedure arguments
 - Return value
 - 3) Memory management
 - Allocate during procedure execution
 - De-allocate upon return
- ❖ All implemented with machine instructions!
- An x86-64 procedure uses only those mechanisms required for that procedure



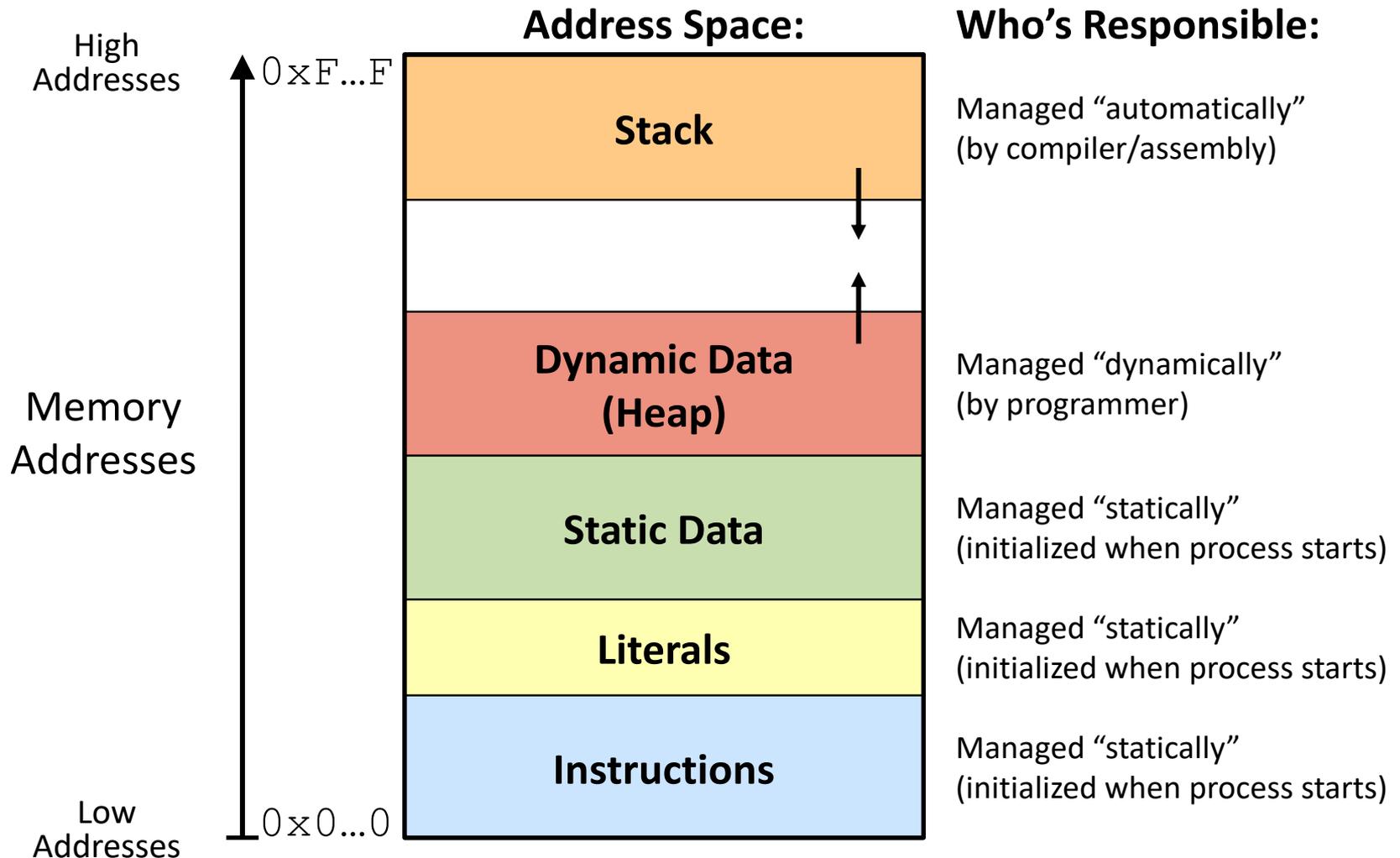
Procedures

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

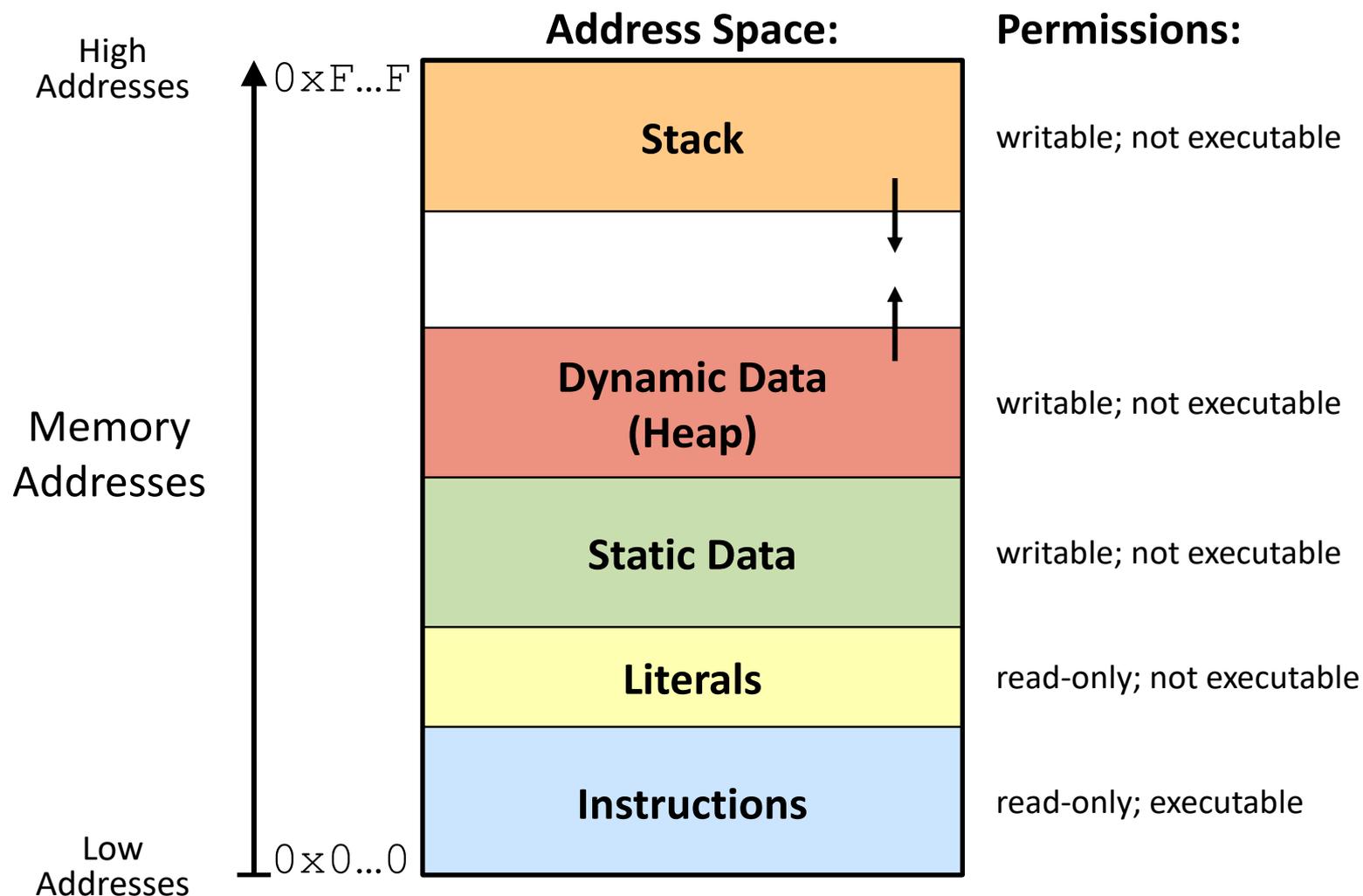
Simplified Memory Layout (Review)



Memory Management



Memory Permissions



- Segmentation fault: impermissible memory access

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

Stack Pointer: `%rsp` →

Stack “Bottom”



High
Addresses

↑
Increasing
Addresses
|

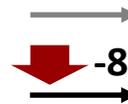
|
Stack Grows
Down
↓

Low
Addresses
0x00...00

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`**
 - Adjust `%rsp` and store contents of `%rcx` on the stack

Stack Pointer: `%rsp`

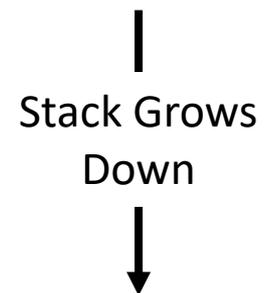
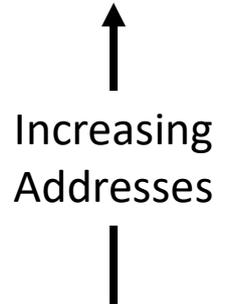


Stack "Bottom"



Stack "Top"

High
Addresses



Low
Addresses
0x00...00

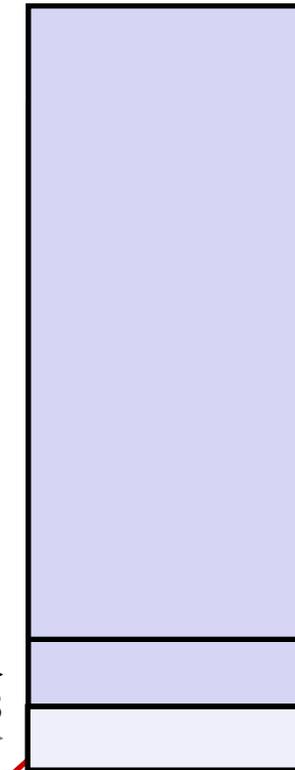
x86-64 Stack: Pop (Review)

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

Stack Pointer: `%rsp`



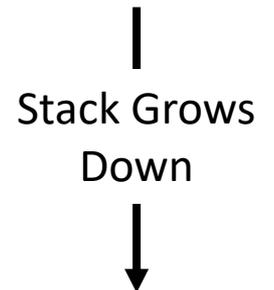
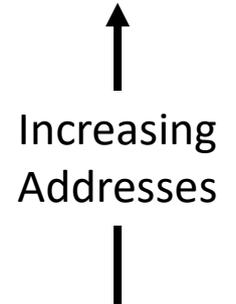
Stack "Bottom"



Stack "Top"

Those bits are still there; we're just not using them.

High Addresses

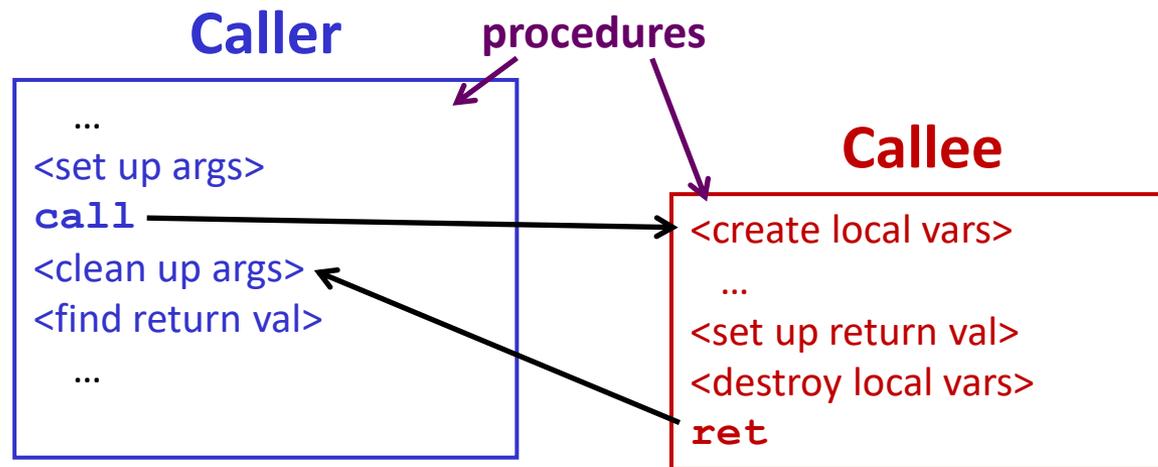


Low Addresses
0x00...00

Procedures

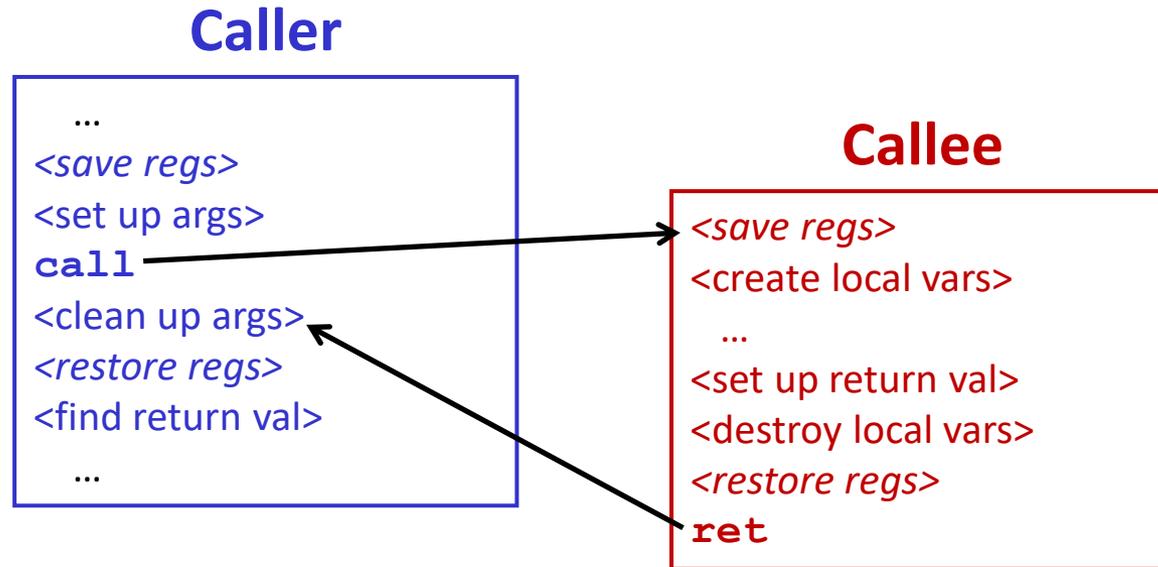
- ❖ Stack Structure
- ❖ **Calling Conventions**
 - **Passing control**
 - Passing data
 - Managing local data
- ❖ Register Saving Conventions
- ❖ Illustration of Recursion

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 systems
 - We will see the convention for x86-64/Linux in detail
 - What could happen if our program didn't follow these conventions?

Code Example (Preview)

```
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

Compiler Explorer:

<https://godbolt.org/z/ndro9E>

```
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
```

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

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

Procedure Control Flow (Review)

- ❖ Use stack to support procedure call and return
- ❖ **Procedure call:** `call label`
 - 1) Push return address on stack (*why? which address?*)
 - 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**

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

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

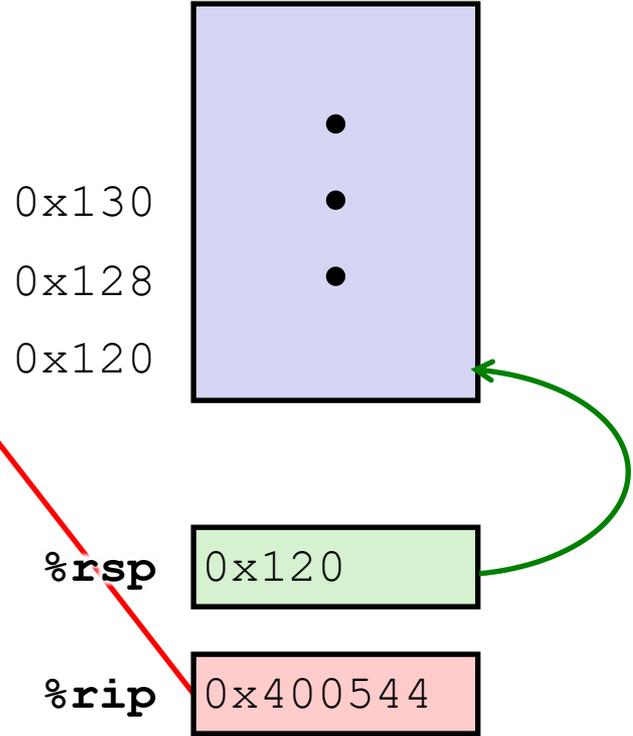
Procedure Call Example (step 1)

```

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

```

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



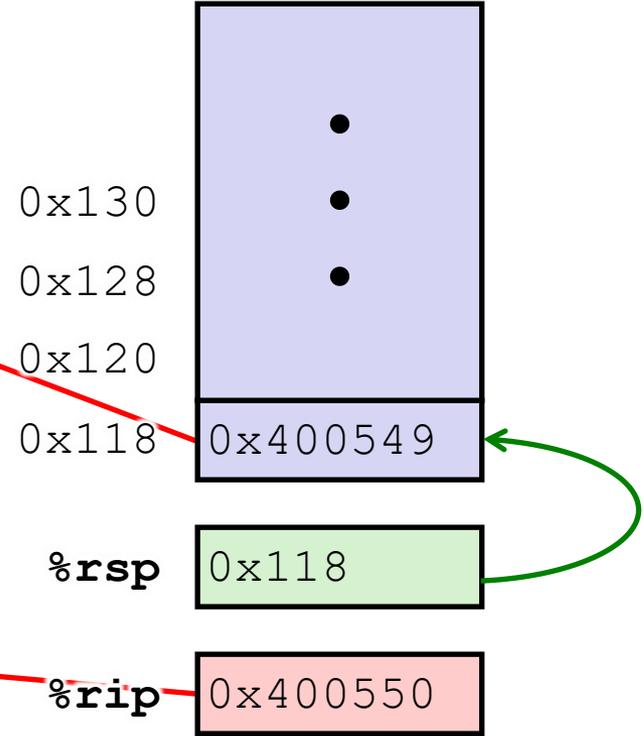
Procedure Call Example (step 2)

```

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

```

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



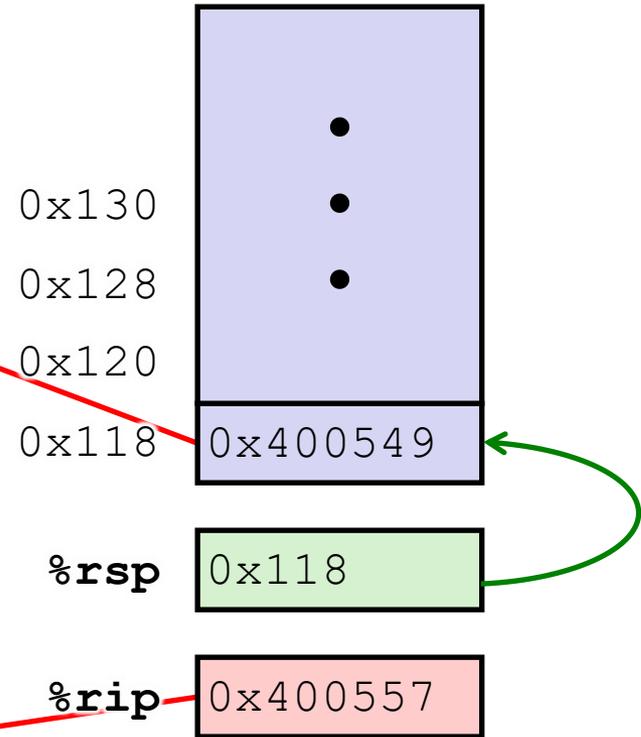
Procedure Return Example (step 1)

```

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

```

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



Procedure Return Example (step 2)

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

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

0x130

0x128

0x120

%rsp

0x120

%rip

0x400549

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)

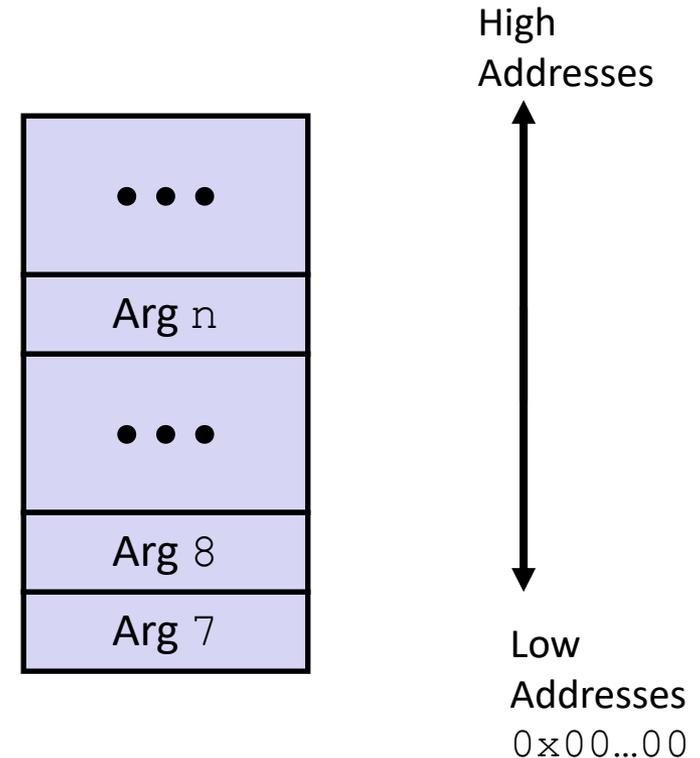
❖ First 6 arguments

<code>%rdi</code>	<u><i>Diane's</i></u>
<code>%rsi</code>	<u><i>Silk</i></u>
<code>%rdx</code>	<u><i>Dress</i></u>
<code>%rcx</code>	<u><i>Costs</i></u>
<code>%r8</code>	<u><i>\$89</i></u>
<code>%r9</code>	

❖ Return value

<code>%rax</code>

Stack (**Memory**)



- 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
- 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

```
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
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
    ...
```

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

```
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
```

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 () ;
    •
    •
}
    
```

```

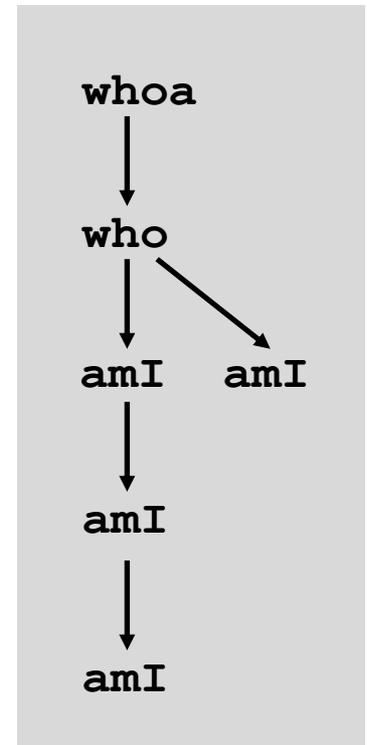
who (...)
{
    •
    amI () ;
    •
    amI () ;
    •
}
    
```

```

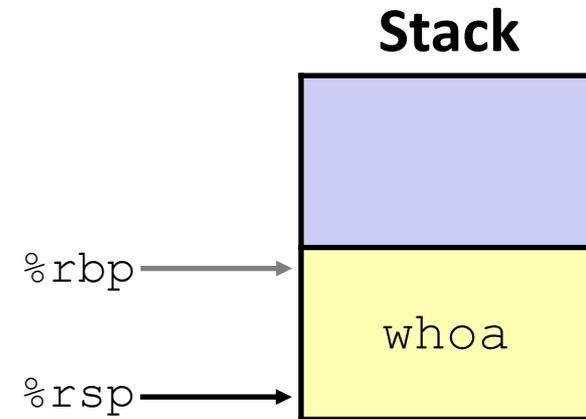
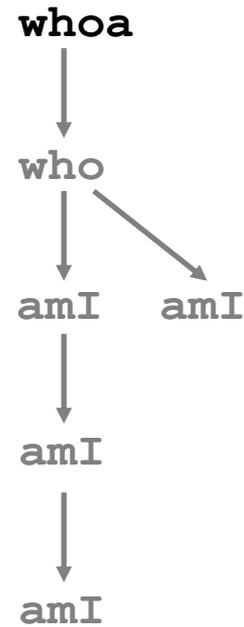
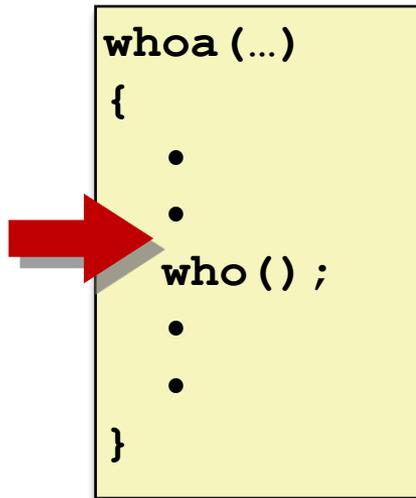
amI (...)
{
    •
    if (...) {
        amI ()
    }
    •
}
    
```

Procedure `amI` is recursive
(calls itself)

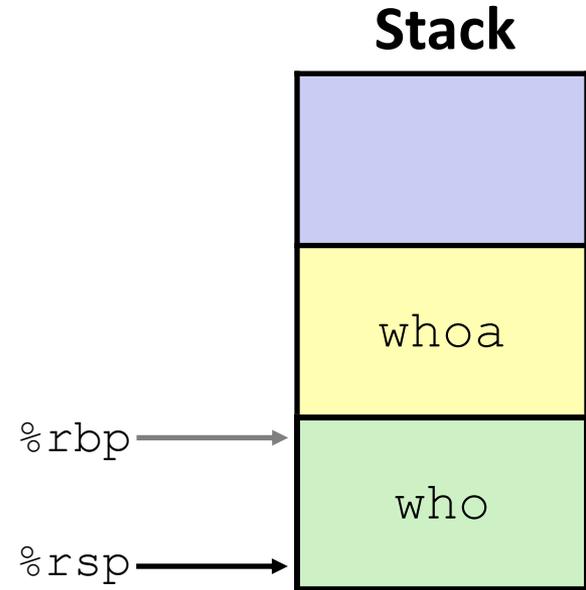
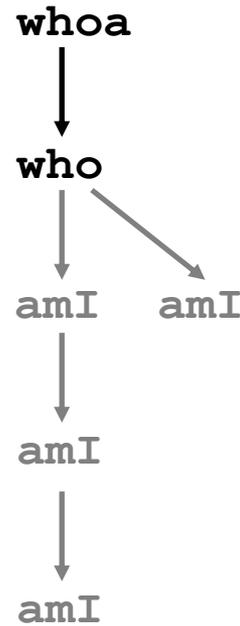
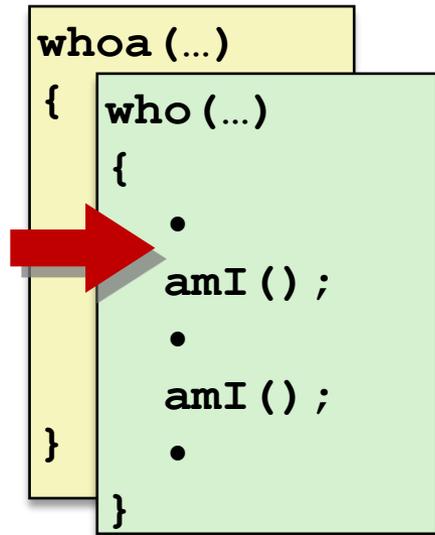
Example
Call Chain



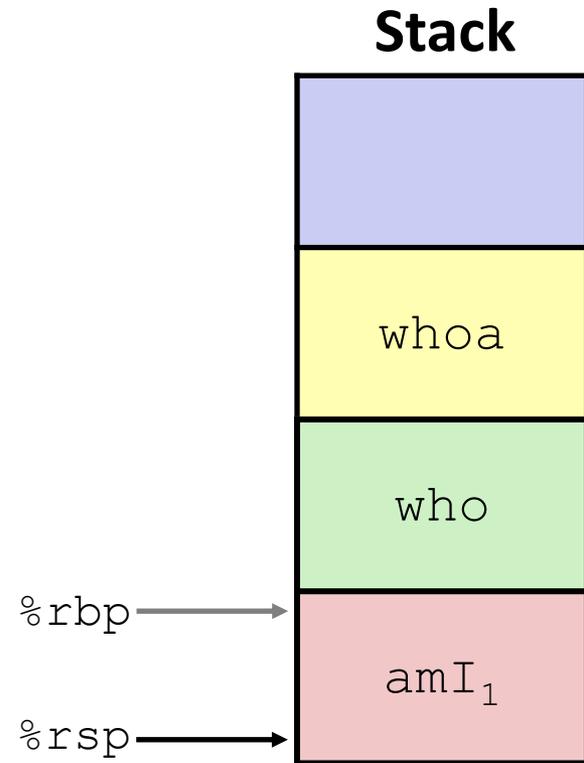
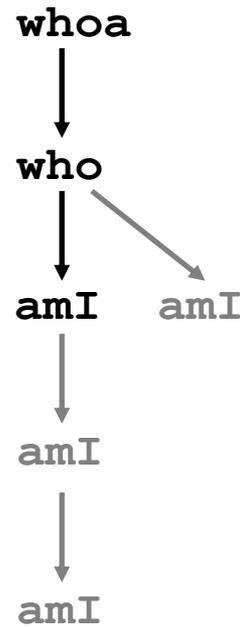
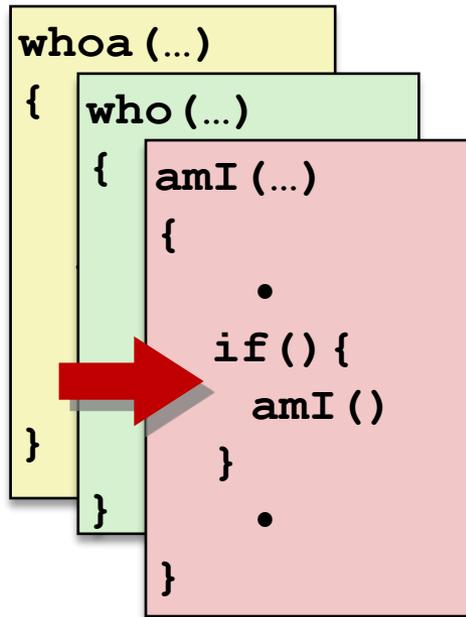
1) Call to whoa



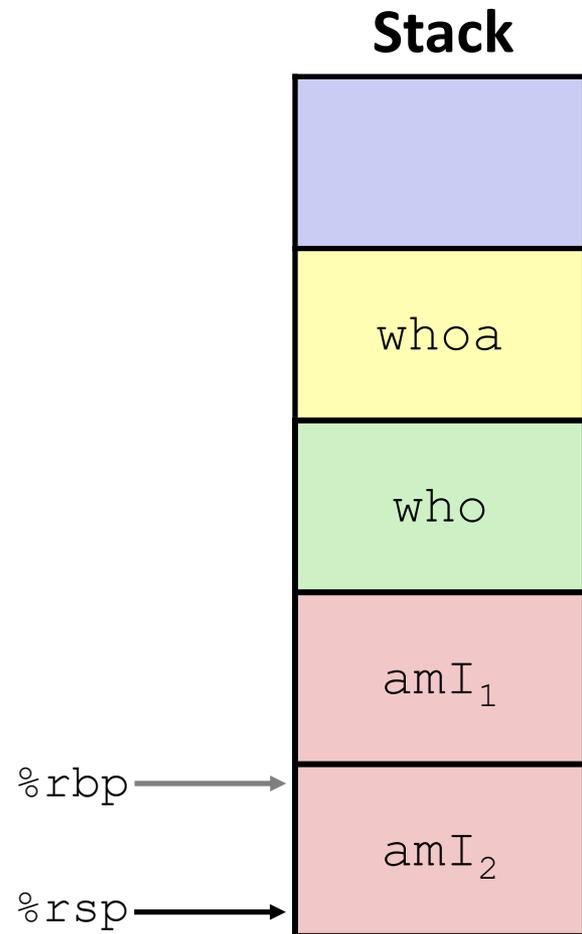
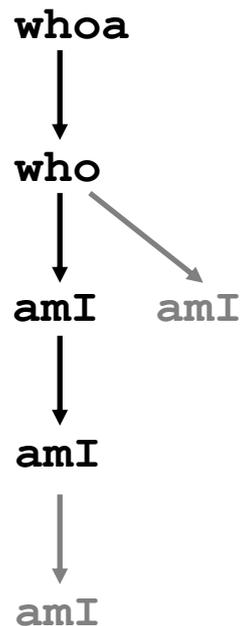
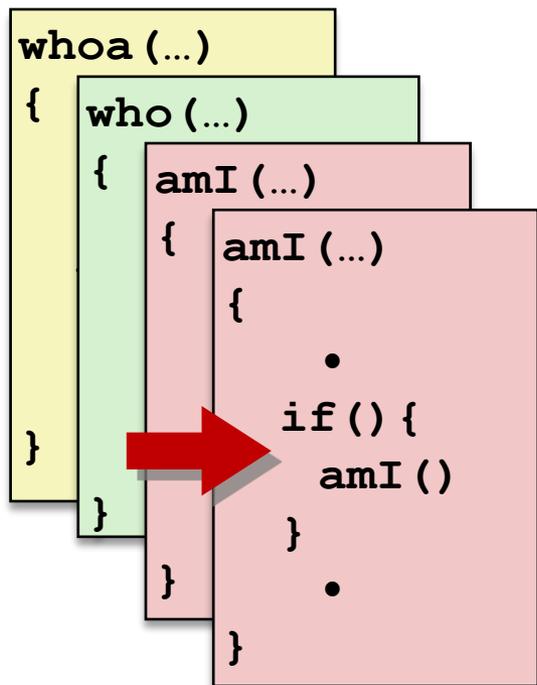
2) Call to who



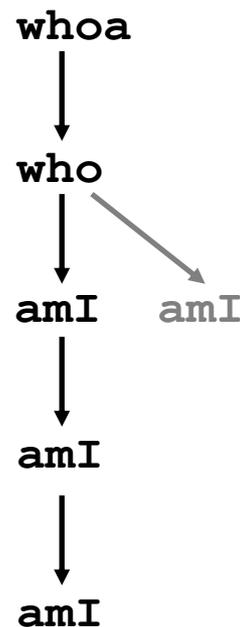
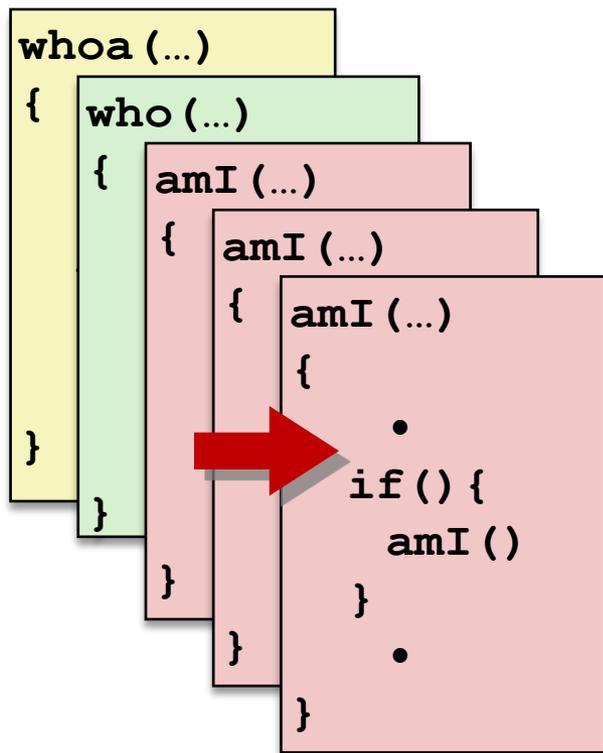
3) Call to amI (1)



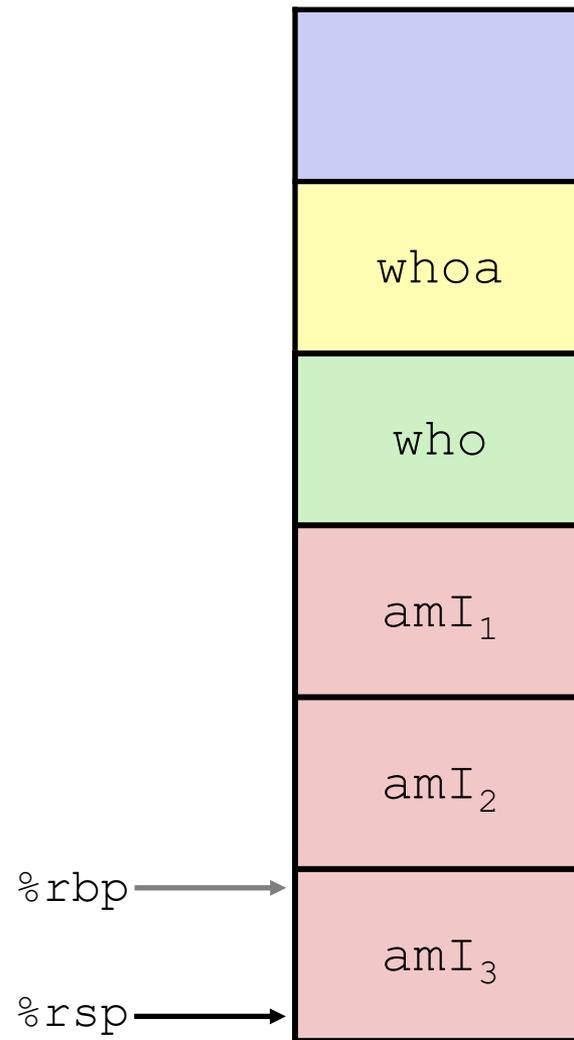
4) Recursive call to amI (2)



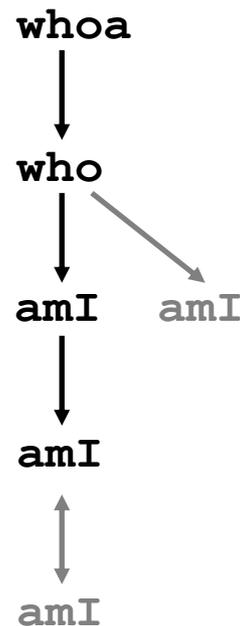
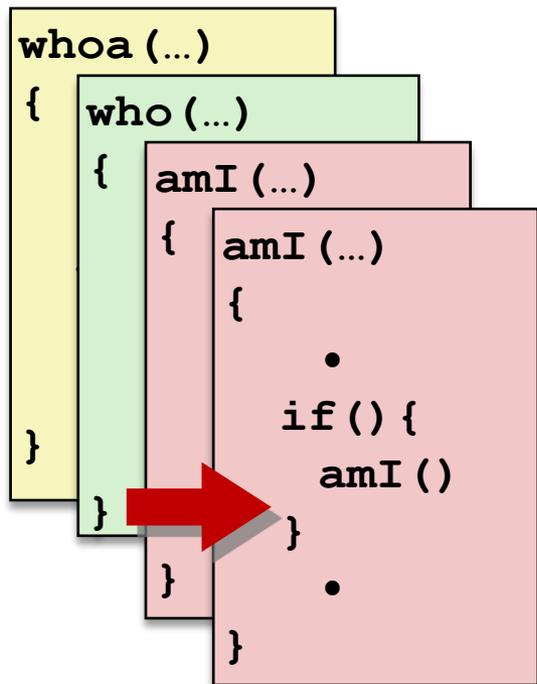
5) (another) Recursive call to amI (3)



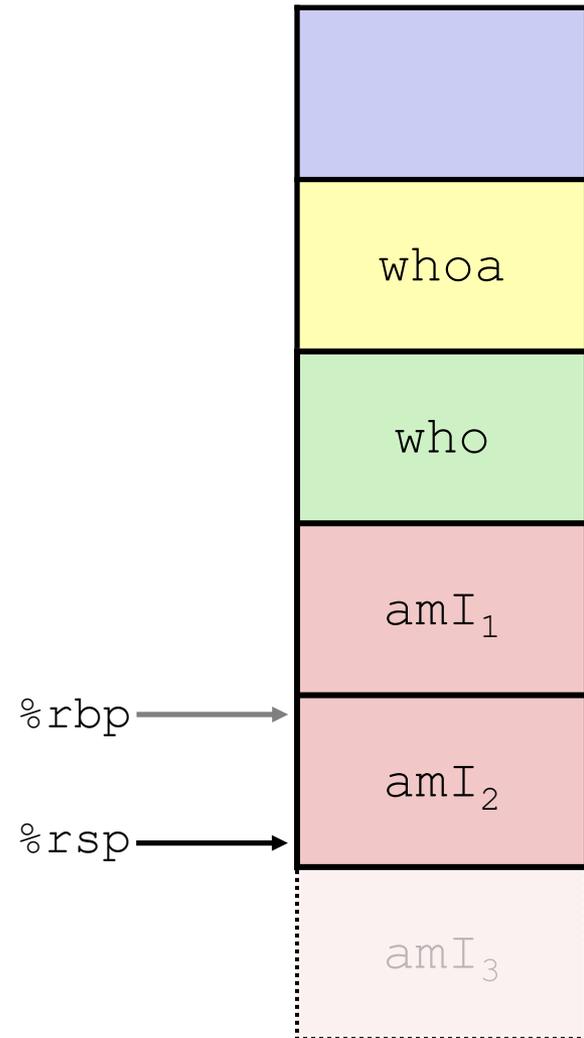
Stack



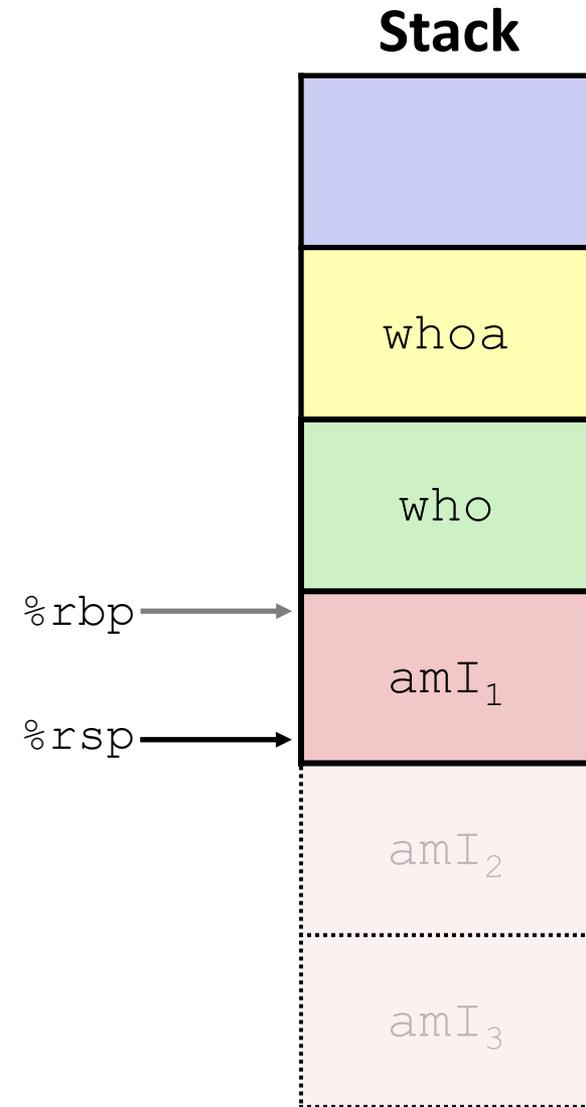
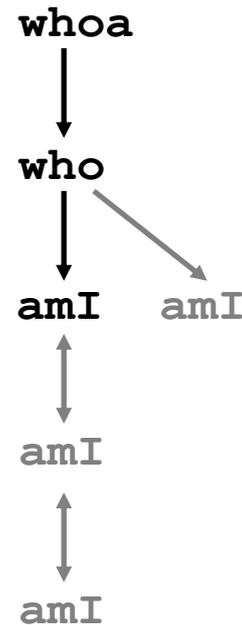
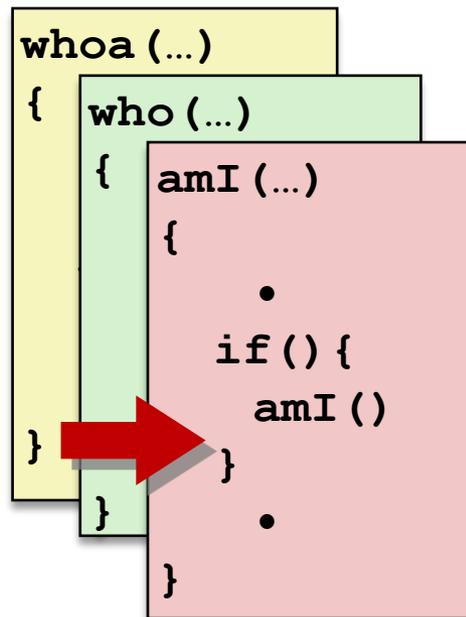
6) Return from (another) recursive call to amI



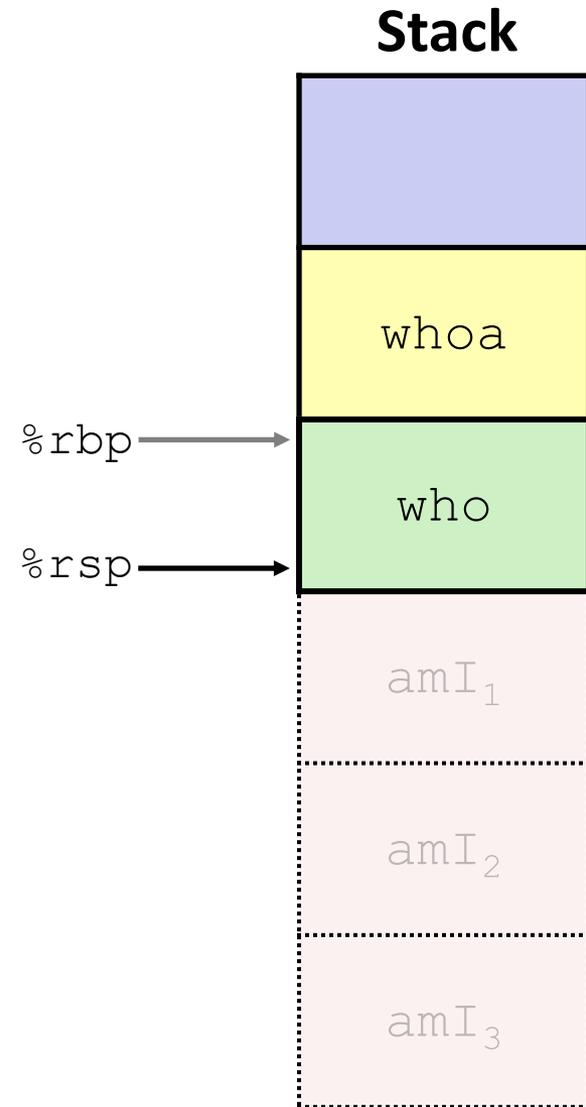
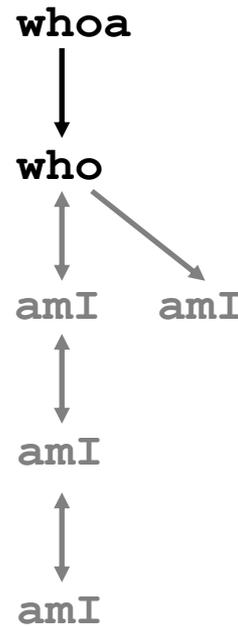
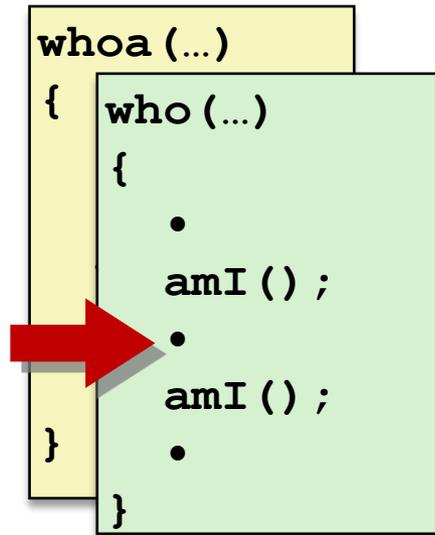
Stack



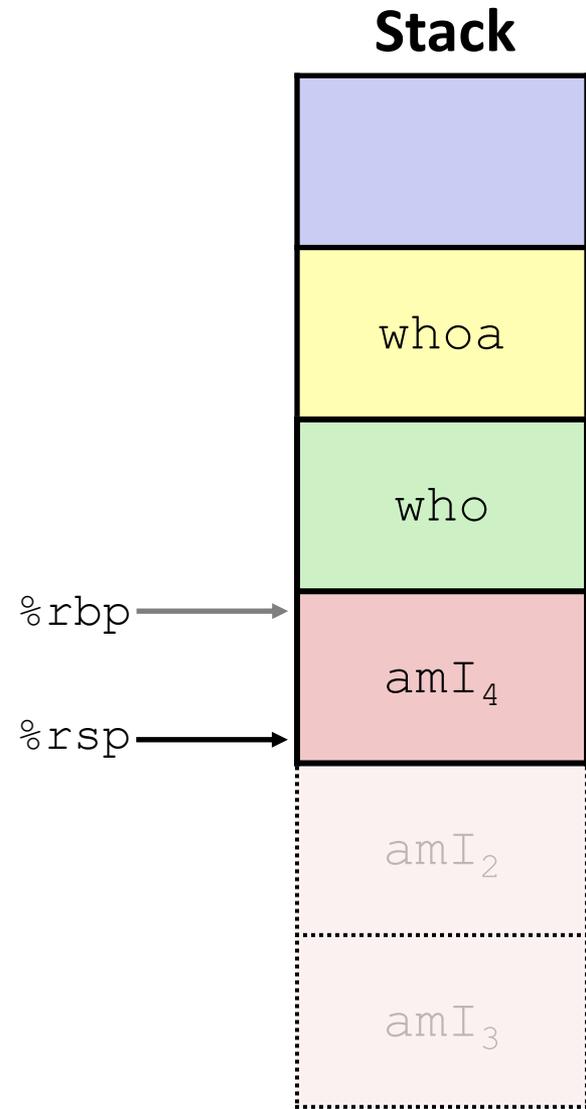
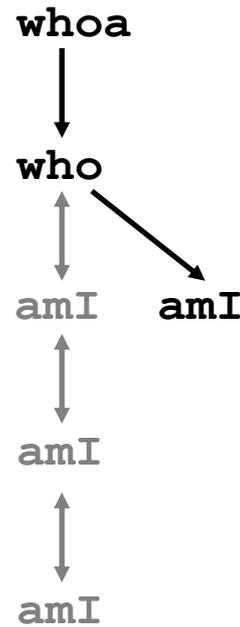
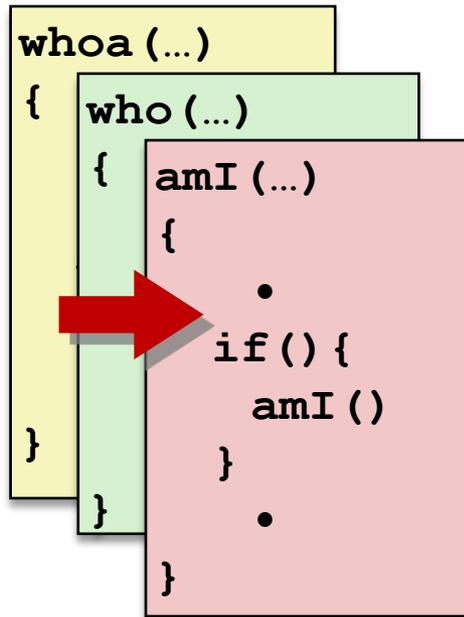
7) Return from recursive call to amI



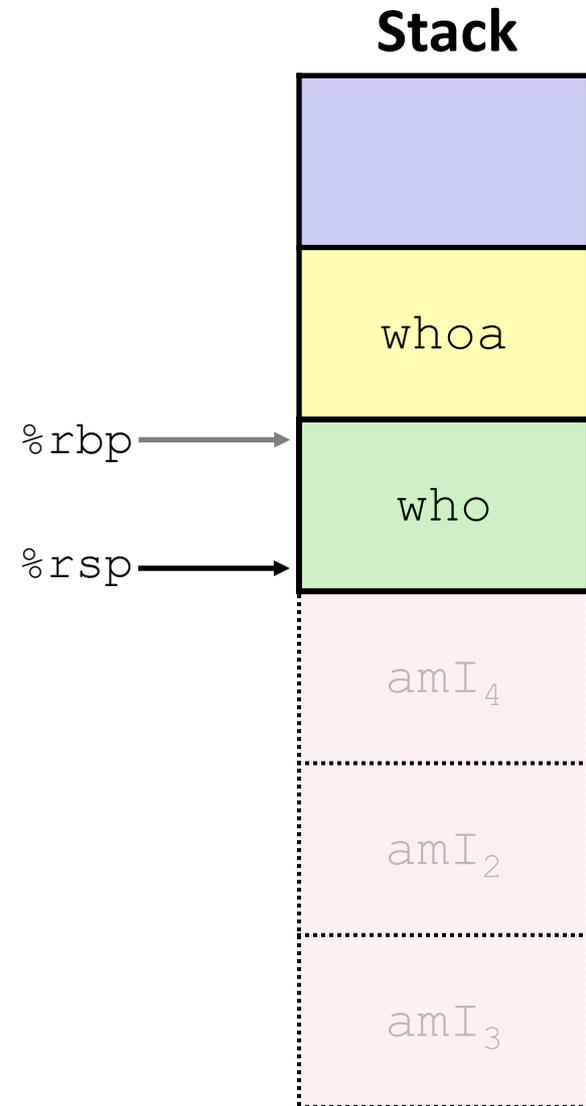
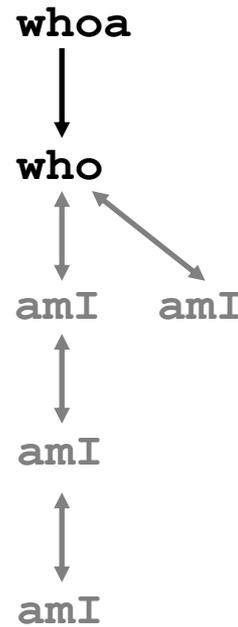
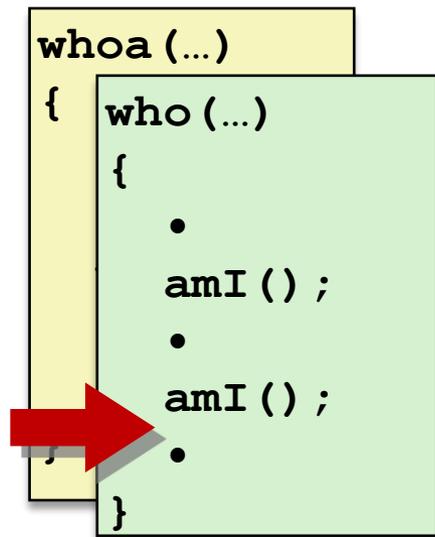
8) Return from call to amI



9) (second) Call to amI (4)



10) Return from (second) call to amI



11) Return from call to who

```

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

