

The Stack & Procedures

CSE 351 Spring 2022

Instructor:

Ruth Anderson

Teaching Assistants:

Melissa Birchfield

Kyrie Dowling

Diya Joy

Armin Magness

Jeffery Tian

Angela Xu

Jacob Christy

Ellis Haker

Anirudh Kumar

Hamsa Shankar

Assaf Vayner

Effie Zheng

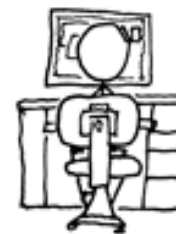
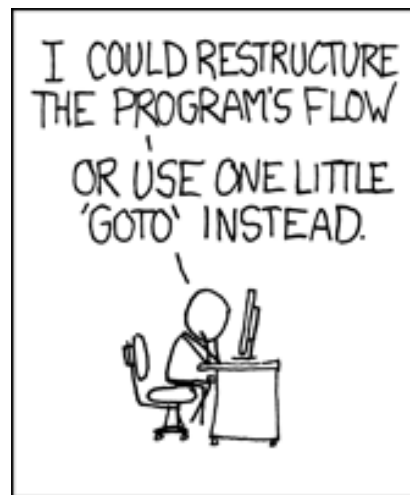
Alena Dickmann

Maggie Jiang

Jim Limprasert

Dara Stotland

Tom Wu



Relevant Course Information

- ❖ Lab 2 (x86-64) due next Friday (4/29)
 - Learn to read x86-64 assembly and use GDB
 - Optional GDB Tutorial on Ed Lessons
 - See GDB Tutorial and Phase 1 walkthrough in Section 4 Lesson
- ❖ Lab 1b – was due on Monday, closes tonight

x86 Control Flow

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

```
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
 - Here: 5 & 6
- ❖ Fall through cases
 - Here: 2
- ❖ Missing cases
 - Here: 4 ???
- ❖ Implemented with:
 - Jump table
 - Indirect jump instruction

Jump Table Structure

Switch Form

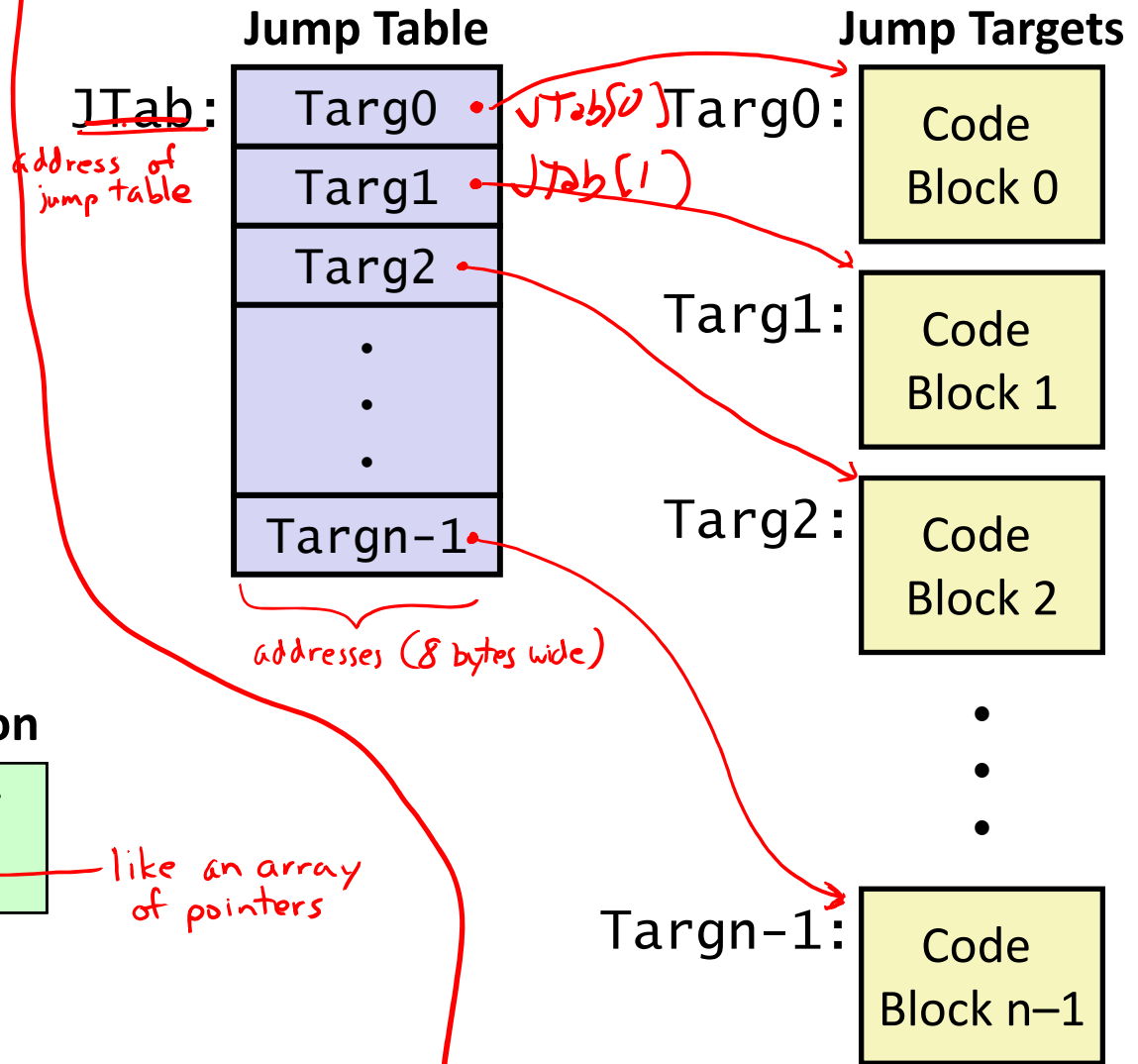
```

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 Structure

C code:

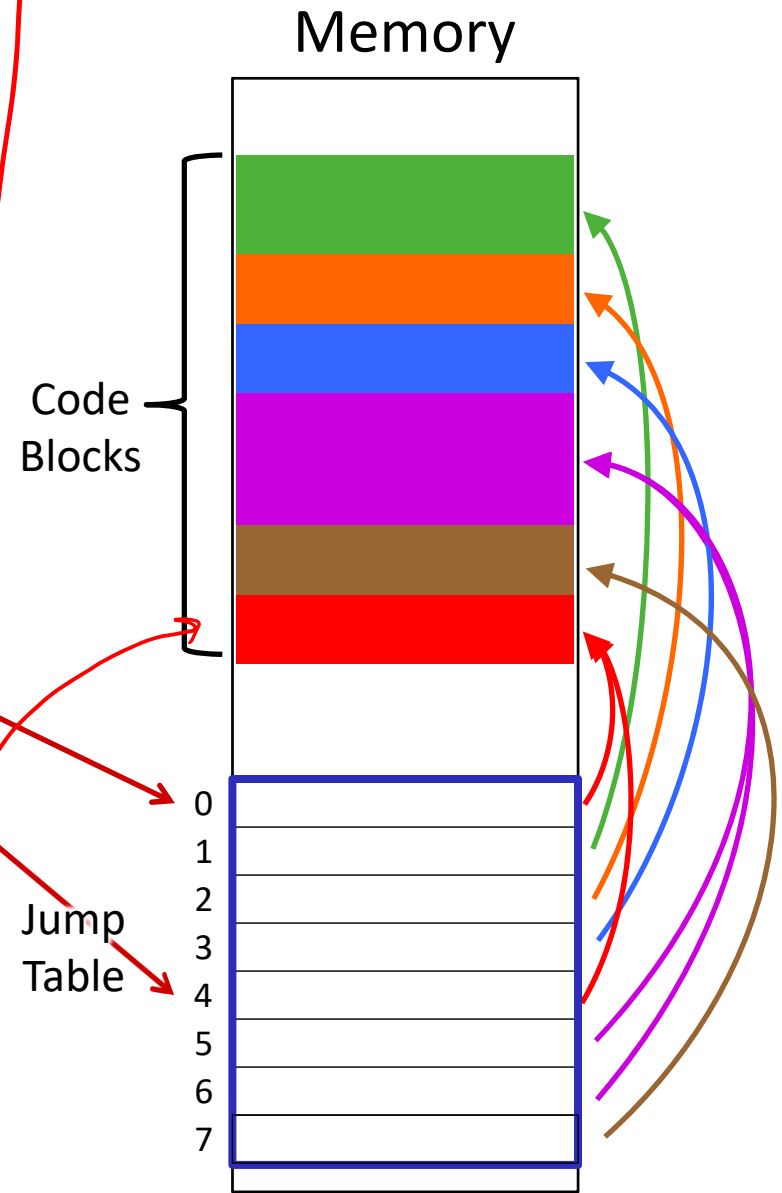
```

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>
}
    
```

Use the jump table when $x \leq 7$:

```

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 | 1 st argument (x) |
| %rsi | 2 nd argument (y) |
| %rdx | 3 rd argument (z) |
| %rax | return value |

Note 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
```

jump to default case if x > 7 (unsigned)

jump above – unsigned > catches negative default cases
 -1 > 7U → jump to default case
 add r

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

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
```

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 |

following data is a "quad word" = 8 bytes

Indirect jump

$$D + R_i * S$$

↑ addr of jump table
 ↑ x
 ↑ sizeof(void*)

Assembly Setup Explanation

❖ Table Structure

- Each target requires 8 bytes (address)
- Base address at .L4

❖ Direct jump: `jmp .L9`

- Jump target is denoted by label .L9

?rip

❖ 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 $0 \leq x \leq 7$

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
```

*Mem[D + Reg[Ri] * 5]*

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks**
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

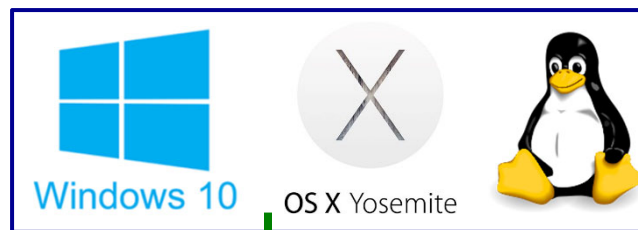
Assembly language:

```
get_mpg:
    pushq    %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

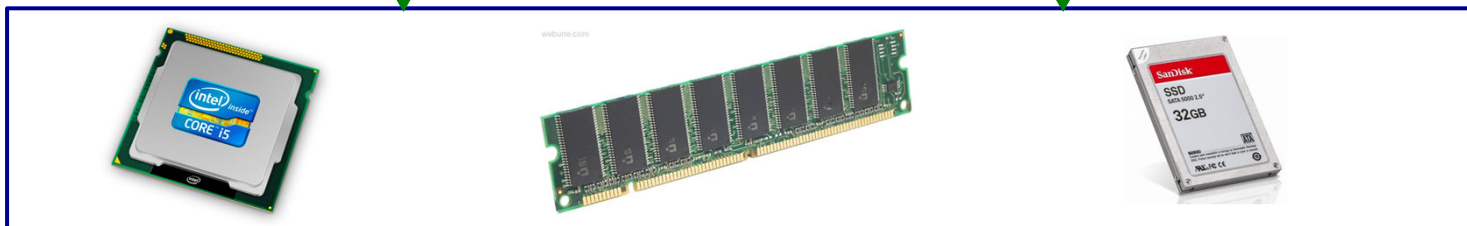
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:



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?

^{add to stack} → `pushq %rbp` ^{8B} # grow 8 B
^{lower address} → `subq $0x18, %rsp` ^{0x18 = 24} # grow 24 B
grow by 32 B

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

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

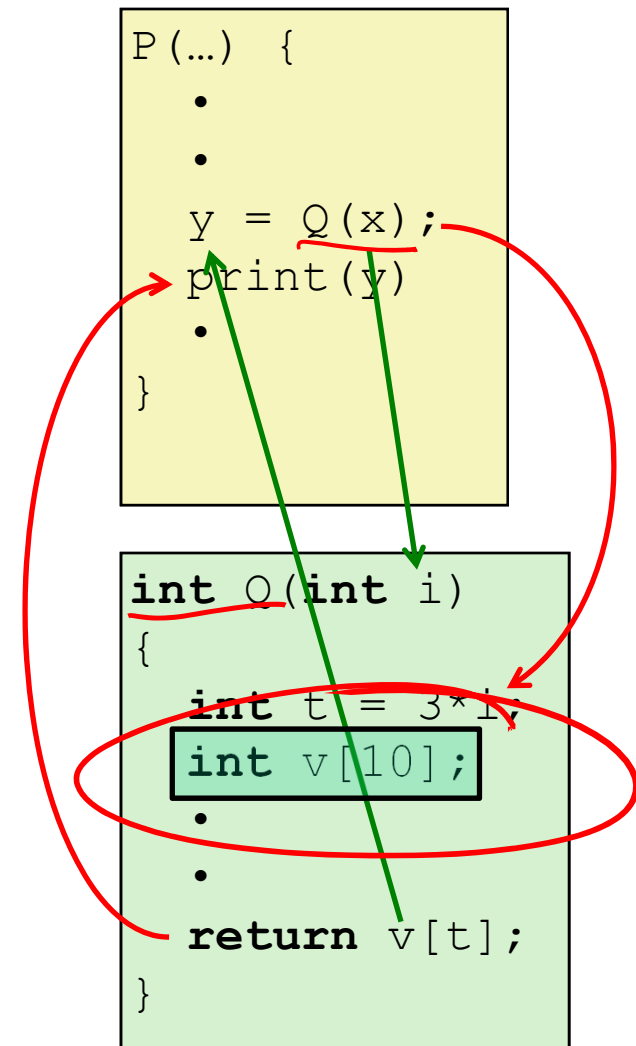
return value in %rax arguments in %rdi, %rsi, and %rdx

%rsp changed by call & ret

%rip changed while executing instructions

Mechanisms required for *procedures*

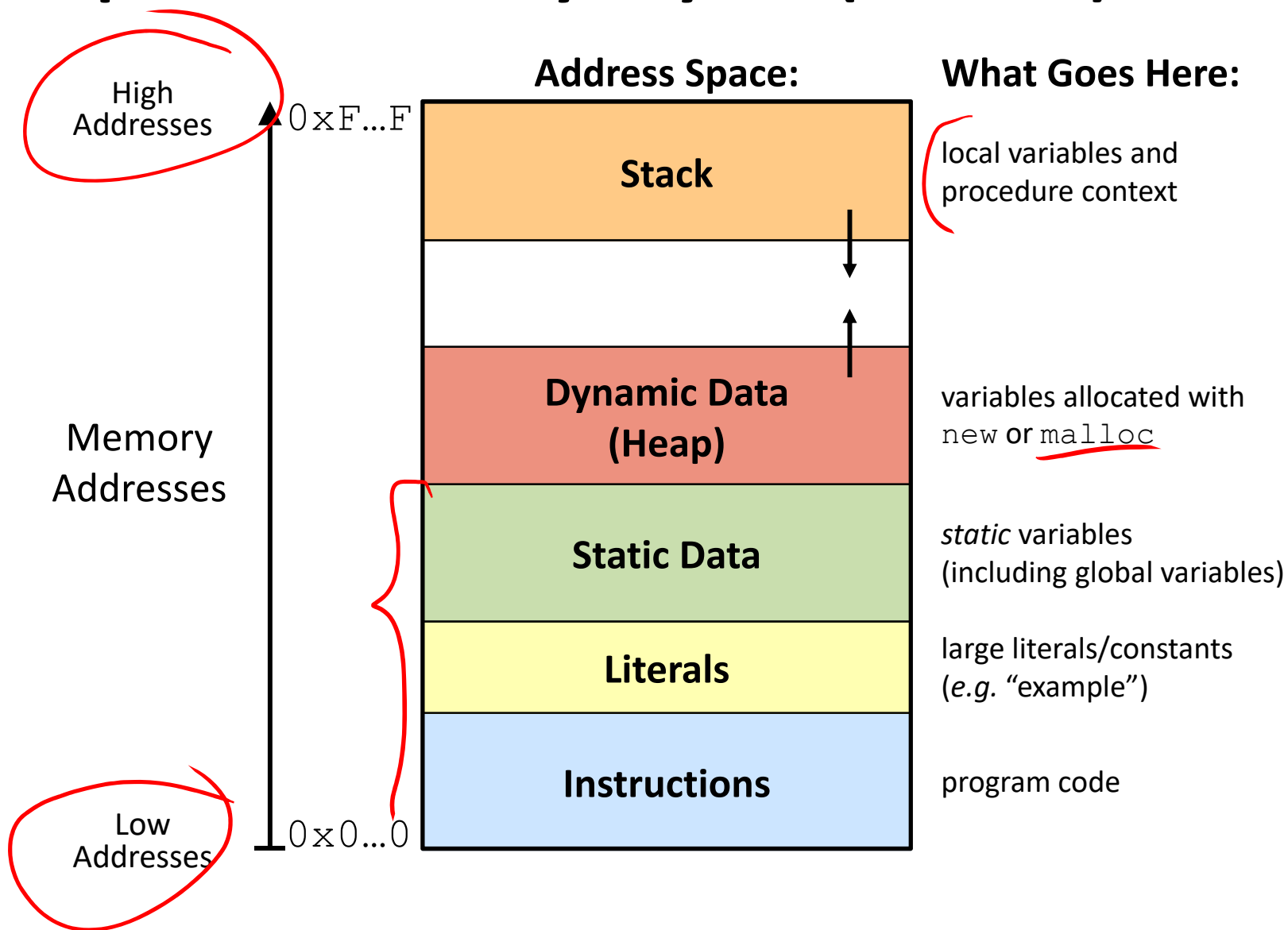
- 1) Passing control
 - To beginning of procedure code
 - Back to return point
 - 2) Passing data
 - Procedure arguments
 - Return value
 - 3) Memory management
 - Allocate during procedure execution
 - Deallocate upon return
- ❖ All implemented with machine instructions!
- An x86-64 procedure uses only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



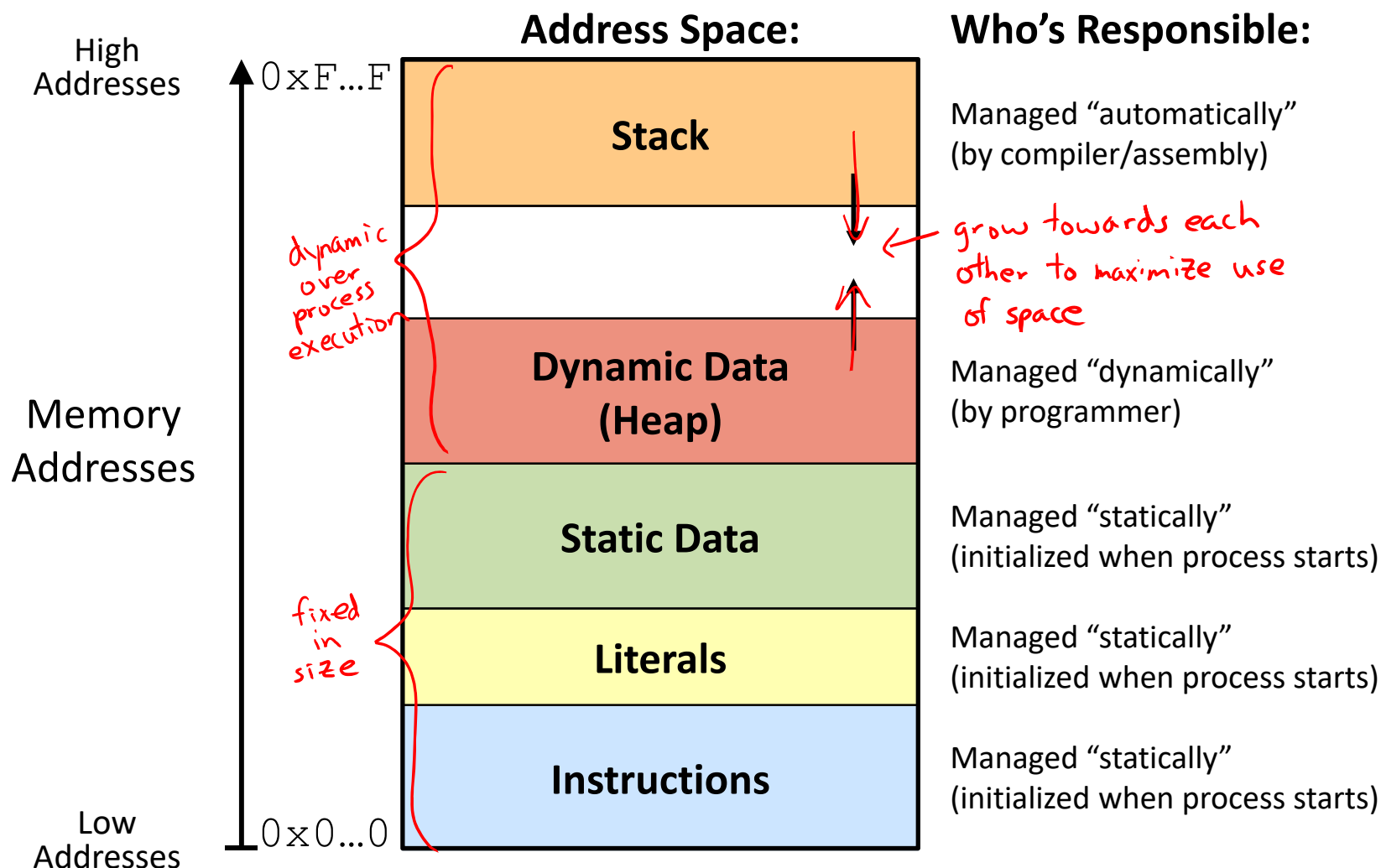
Procedures

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

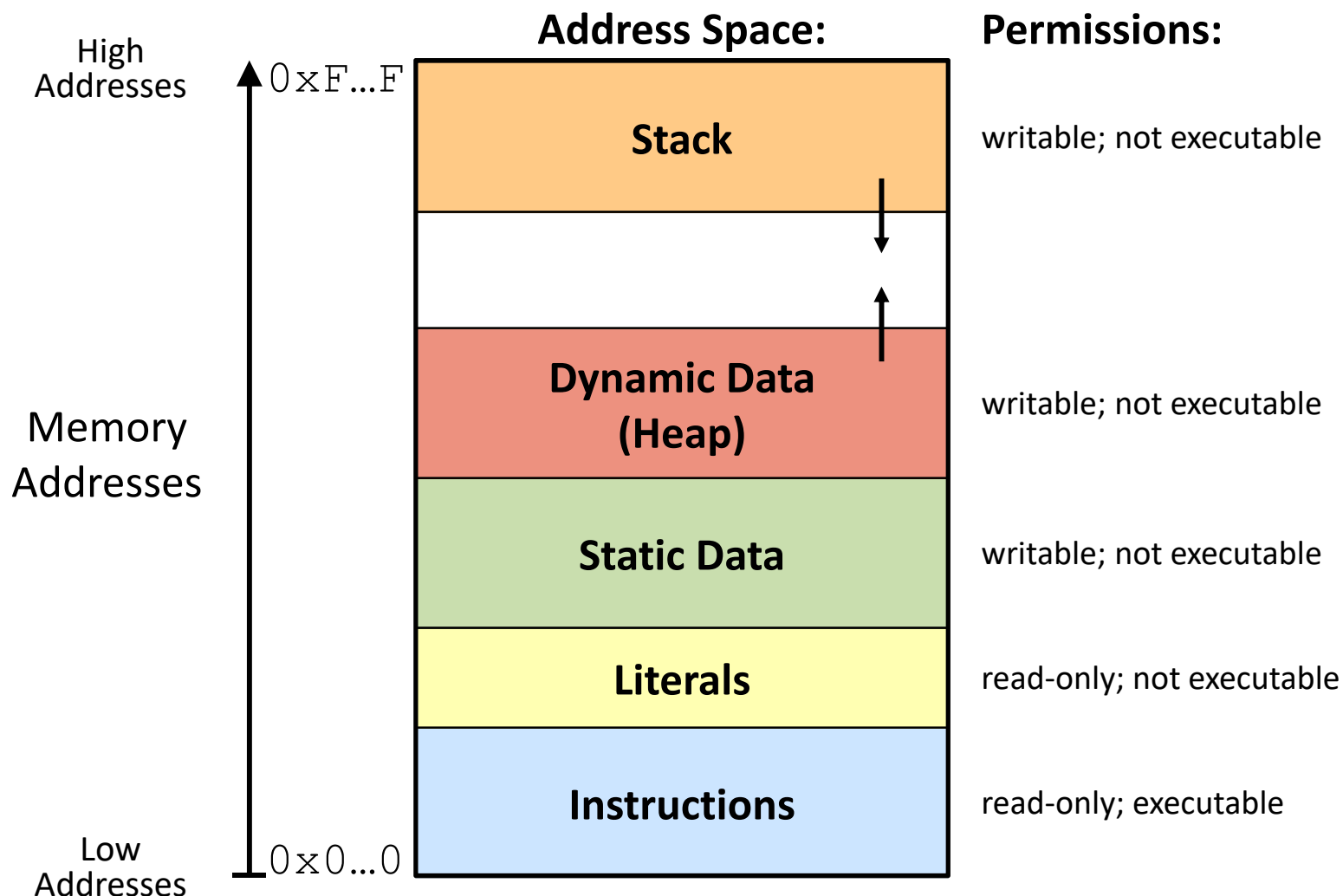
Simplified Memory Layout (Review)



Memory Management



Memory Permissions



- Segmentation faults: impermissible memory access

Last In, First Out (LIFO)

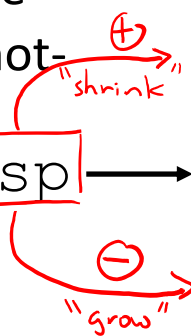
x86-64 Stack (Review)

- ❖ 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”

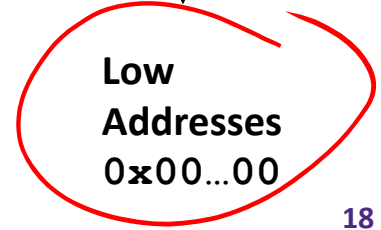


Stack “Top”



Increasing Addresses

Stack Grows Down



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` → -8
 new `%rsp` →

- ① move `%rsp` down (subtract)
- ② store `src` at `%rsp`

Memory
Stack "Bottom"

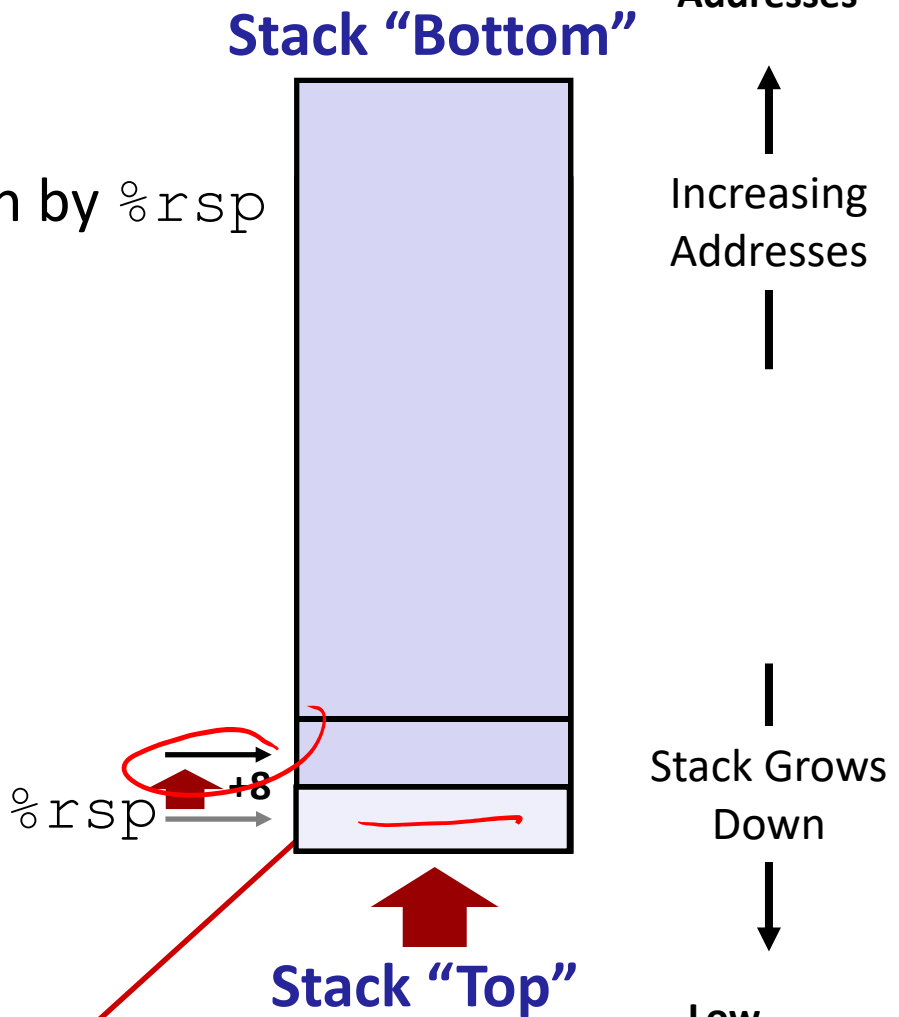


Stack "Top"

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`

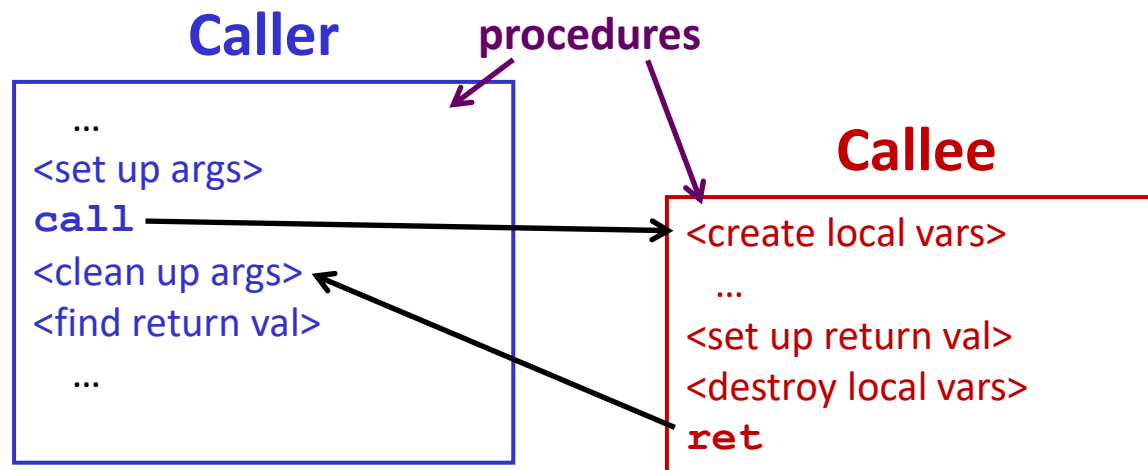


Those bits are still there; we're just not using them.

Procedures

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

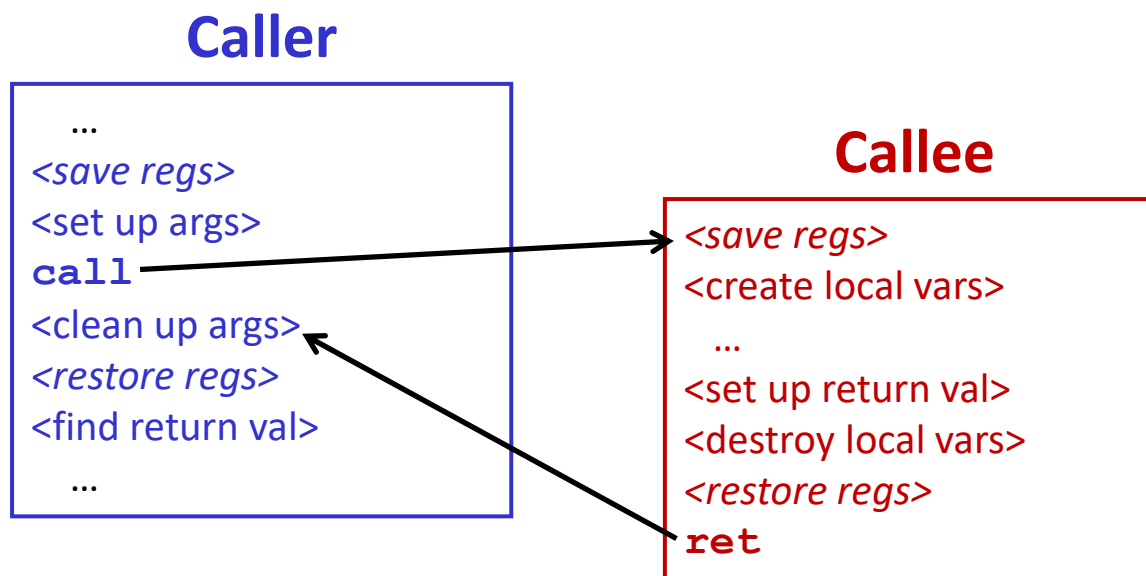
Procedure Call Overview



main
↓
foo
↓
bar

- ❖ **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>

executable 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
```

Caller

Callee

```
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
```

these are instruction addresses

Procedure Control Flow (Review)

- ❖ Use stack to support procedure call and return
- ❖ Procedure call: `call label` (special push)
 - 1) Push return address on stack (*why? which address?*)
 - ① move `%rsp` down
 - ② store ret addr at `%rsp`
 - 2) Jump to `label` → ③ `label` → `%rip`

Procedure Control Flow (Review)

❖ Use stack to support procedure call and return

❖ Procedure call: `call label` (special push)

- 1) Push return address on stack (why? which address?) → ① move %rsp down → ② store ret addr at %rsp
- 2) Jump to `label` → ③ label → %rip

❖ 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` (special pop)

- 1) Pop return address from stack ① read ret addr at %rsp into %rip
- 2) Jump to address ② move %rsp up

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

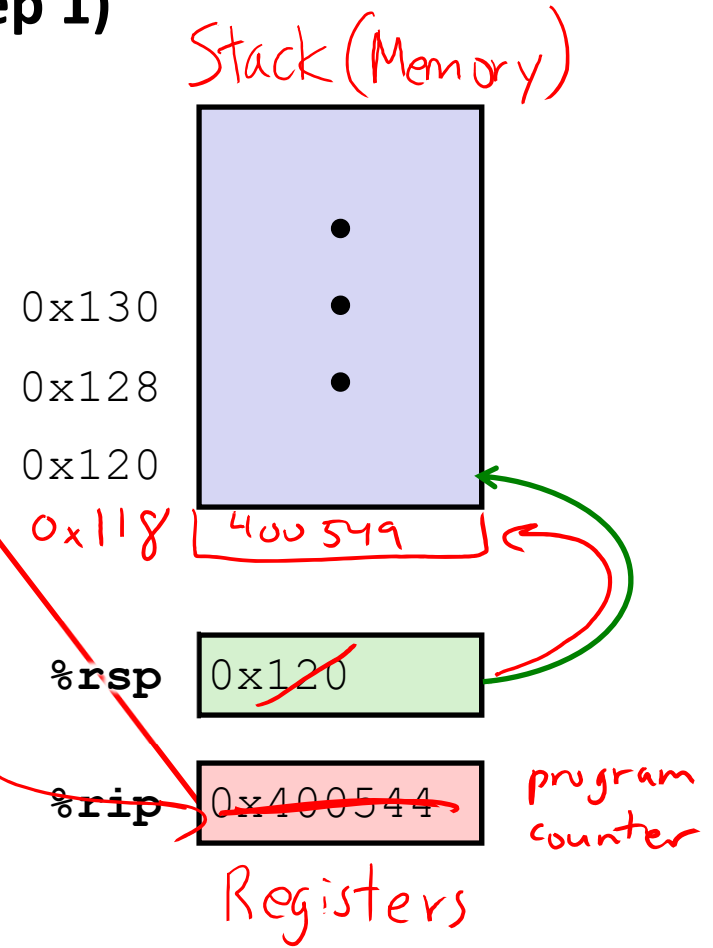
Procedure Call Example (step 1)

```

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

```

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



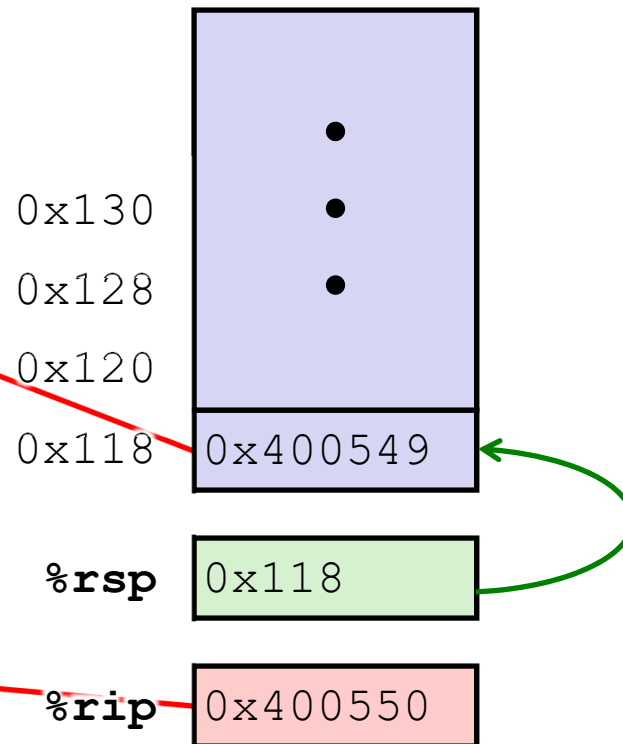
Procedure Call Example (step 2)

```

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

```

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



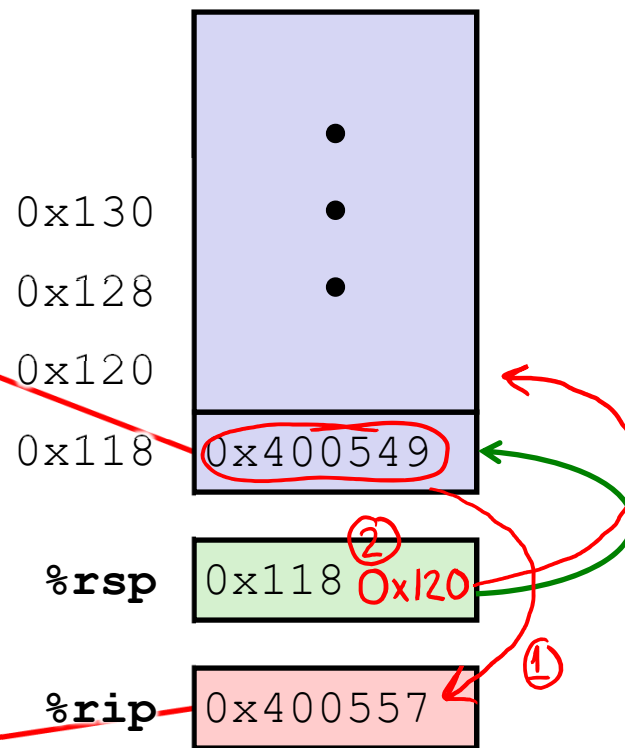
Procedure Return Example (step 1)

```

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

```

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



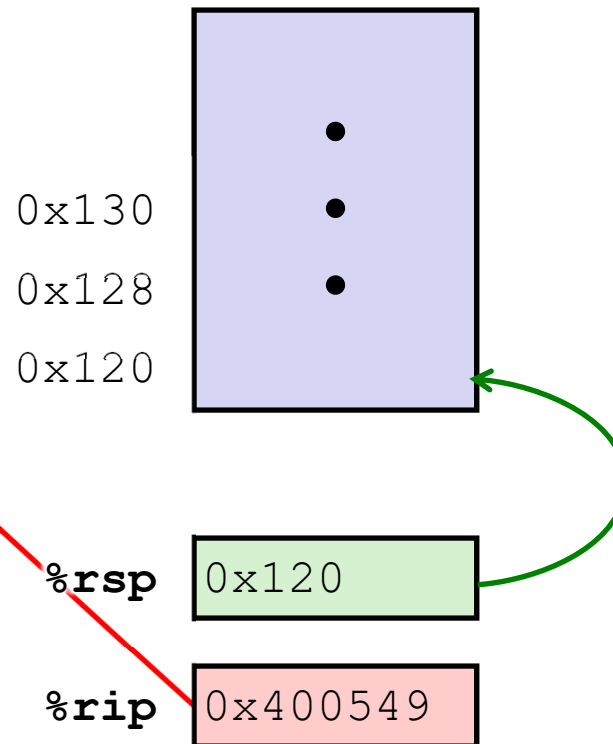
Procedure Return Example (step 2)

```

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

```

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



Procedures

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

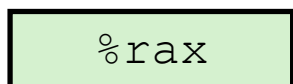
Procedure Data Flow (Review)

Registers (**NOT** in Memory)

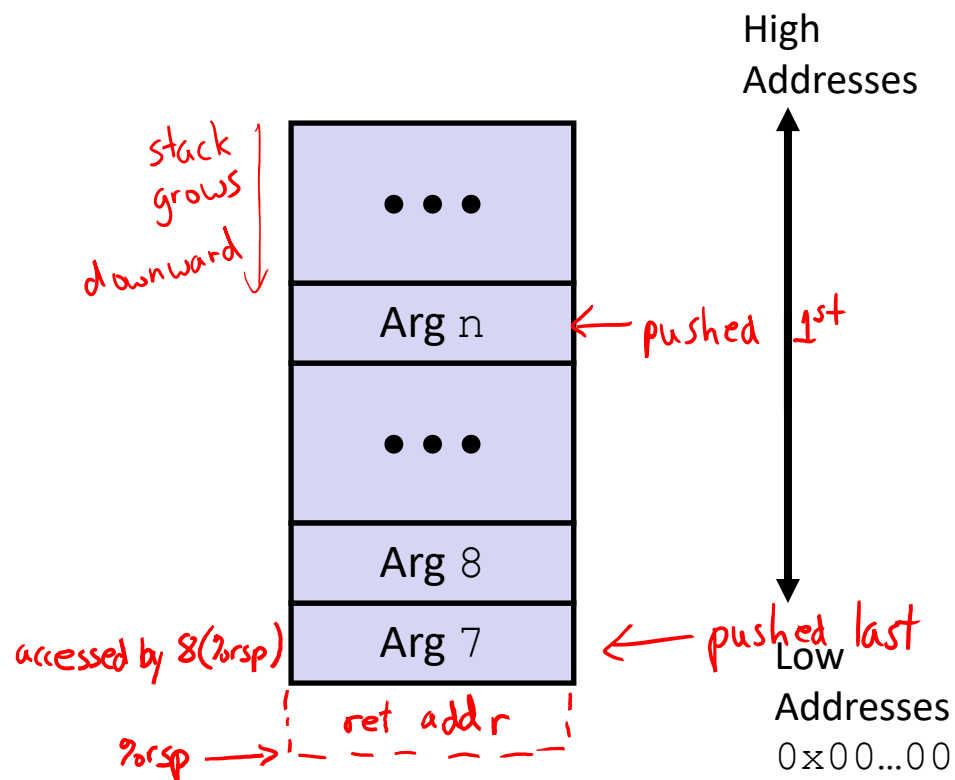
❖ First 6 arguments



❖ Return value



Stack (**M**emory)



- 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;
}
```

lined up nicely so we didn't have to manipulate arguments

```
00000000000400540 <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
    ...
```

(will explain later)

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

```
00000000000400550 <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 (...)
{
  .
  ① amI ();
  .
  ② amI ();
  .
}
    
```

```

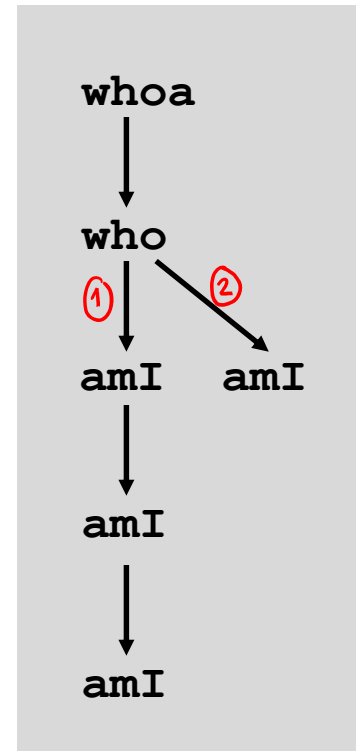
amI (...)
{
  .
  if (...) {
    amI ()
  }
  .
}
    
```

1st call recurses twice

2nd call doesn't recurse

based on condition

Example Call Chain

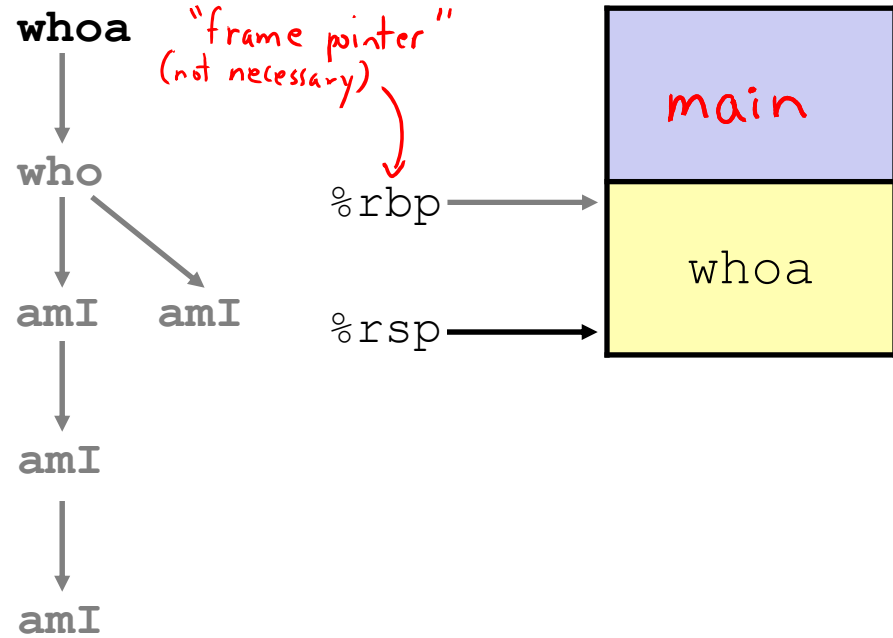


Procedure amI is recursive
(calls itself)

1) Call to whoa

```

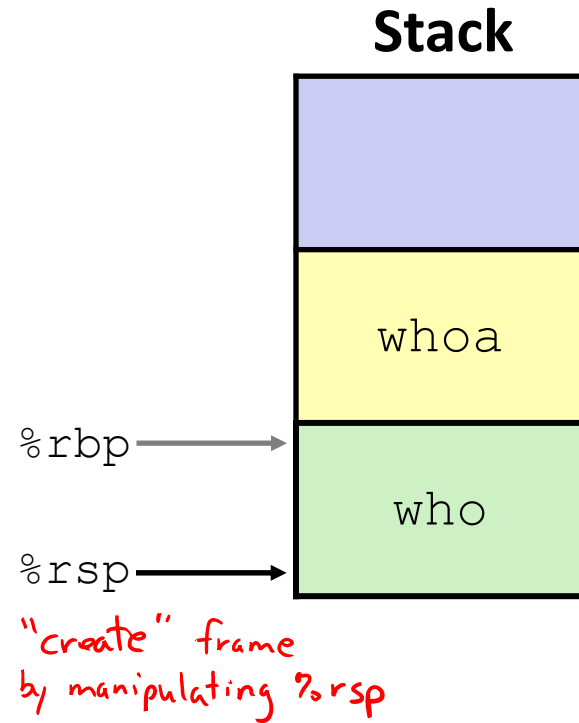
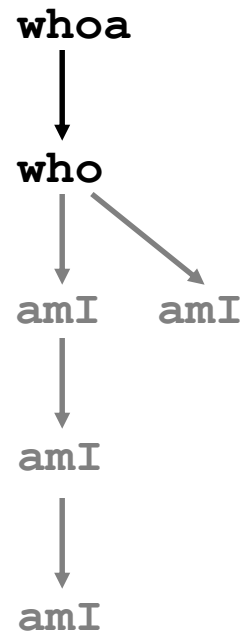
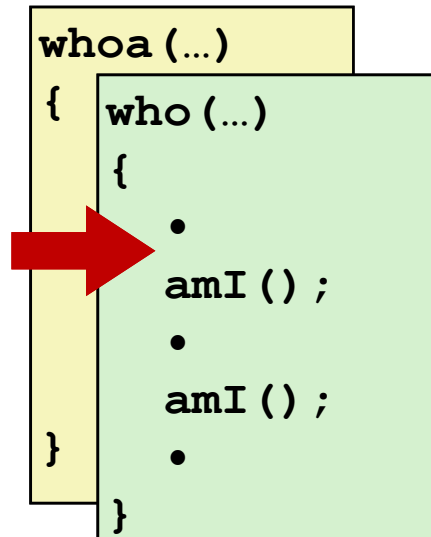
whoa (...)
{
    •
    •
    who ();
    •
    •
}
    
```



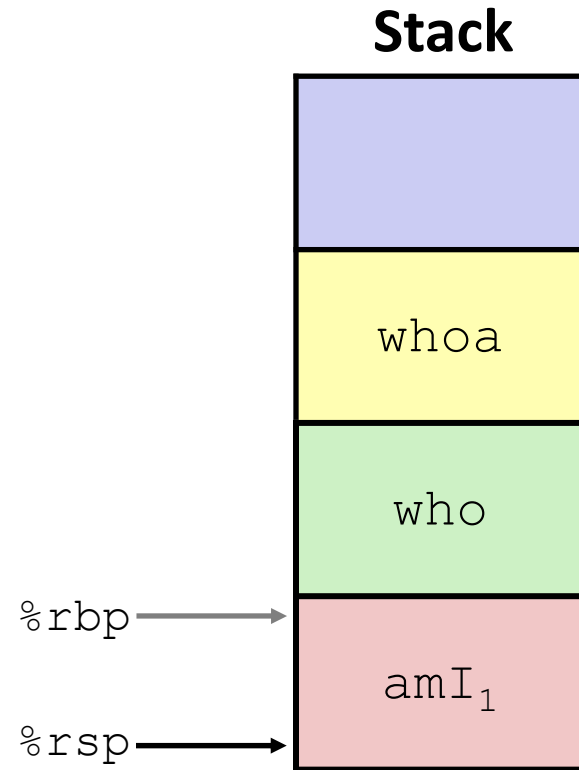
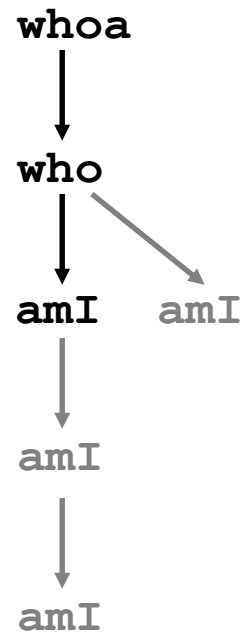
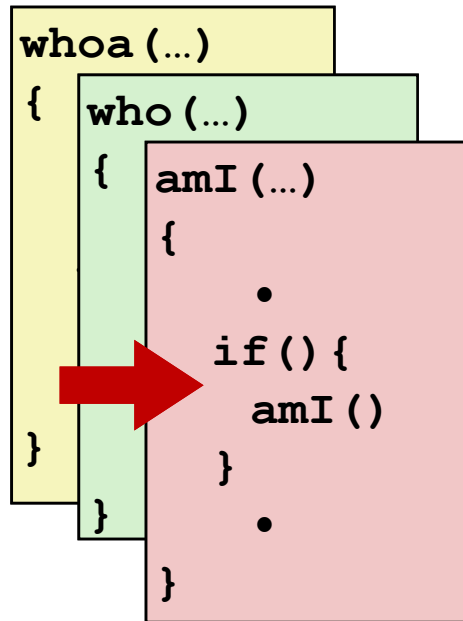
high
could be any procedure that calls whoa

low

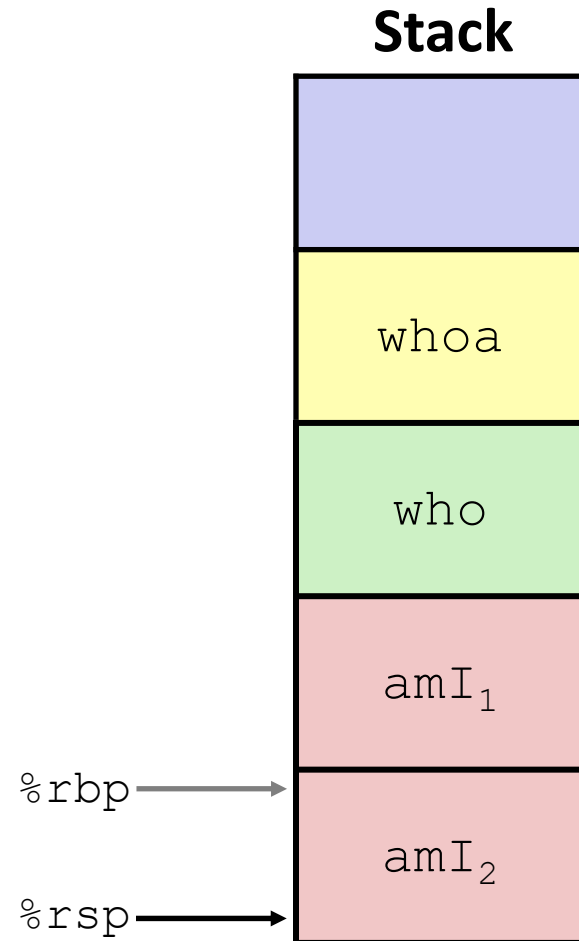
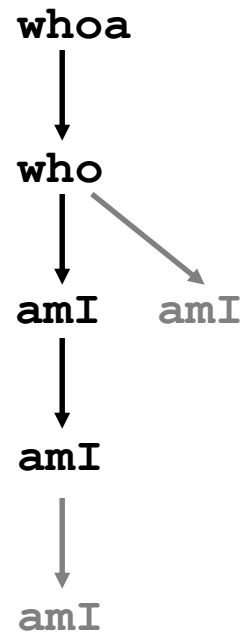
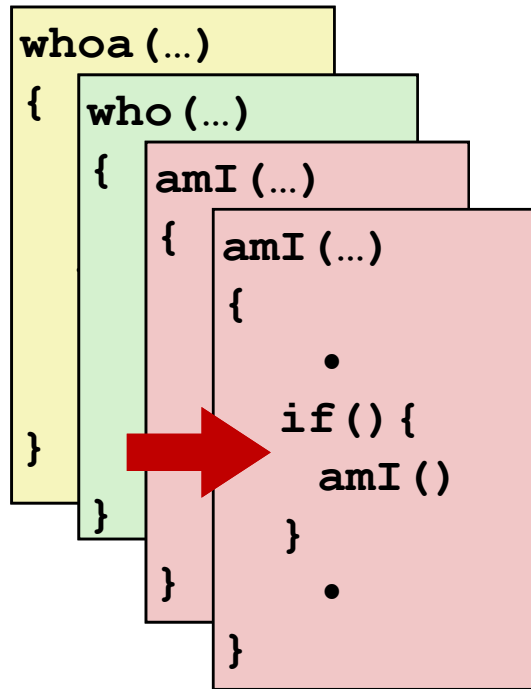
2) Call to who



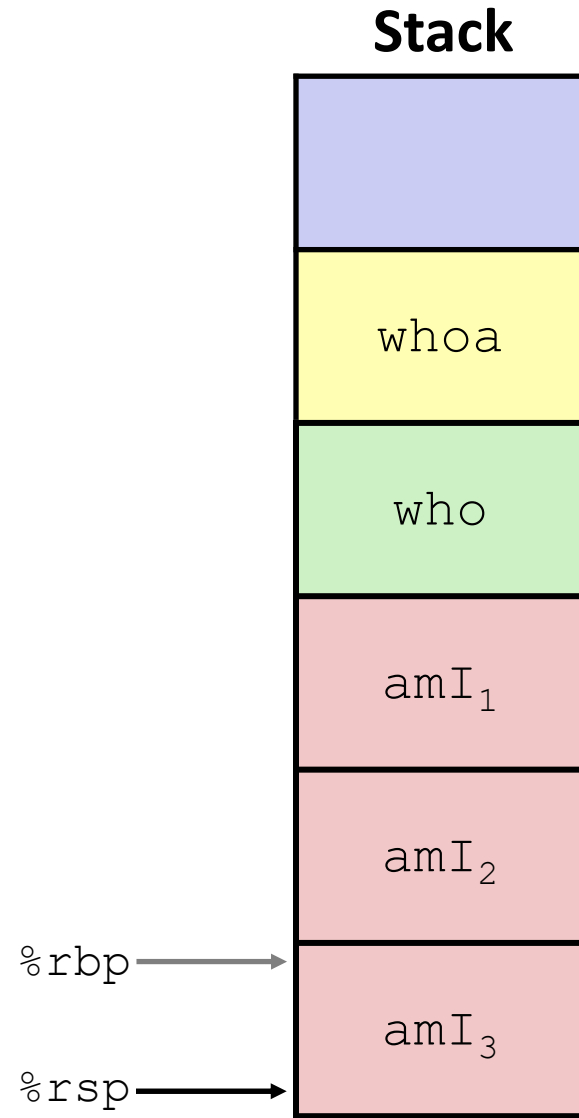
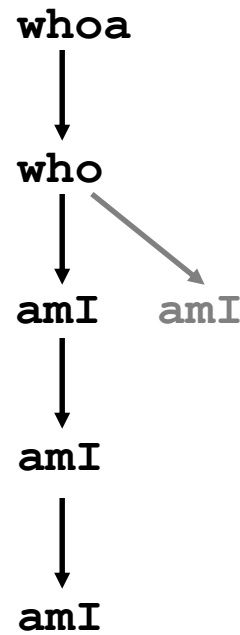
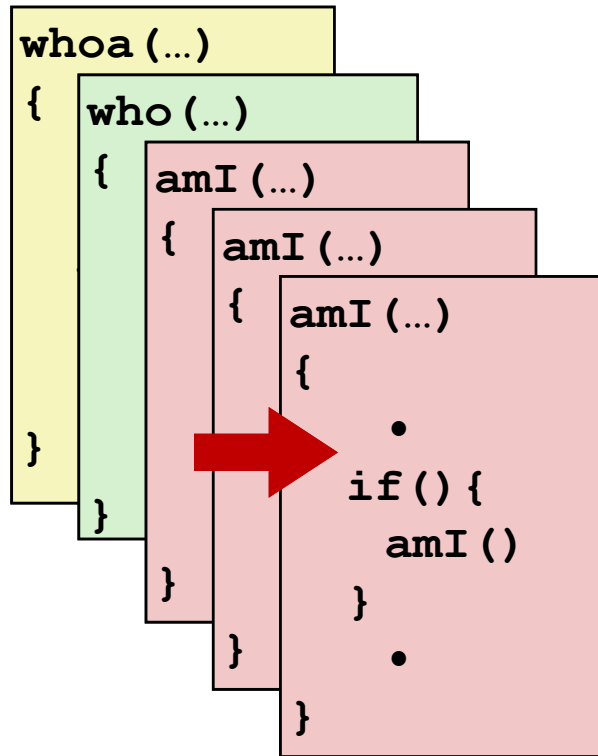
3) Call to amI (1)



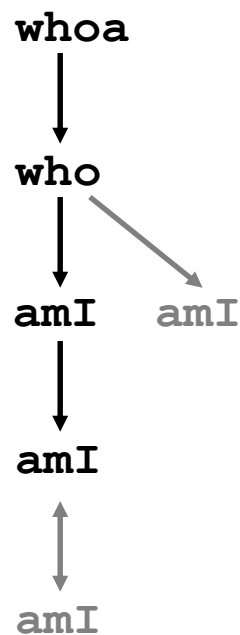
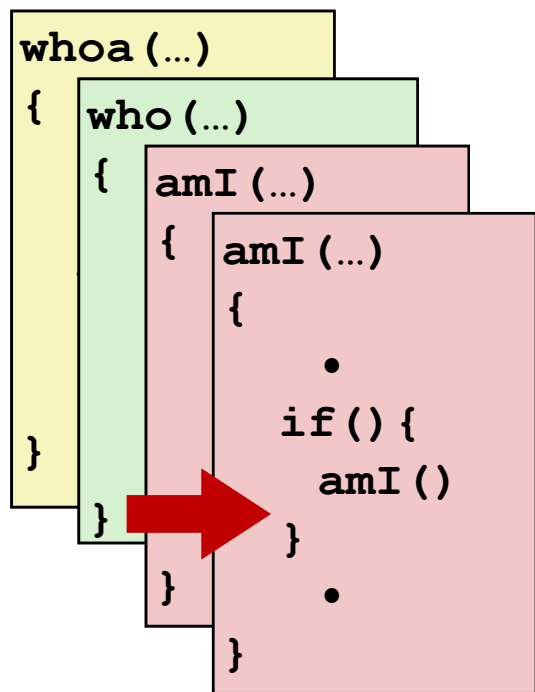
4) Recursive call to amI (2)



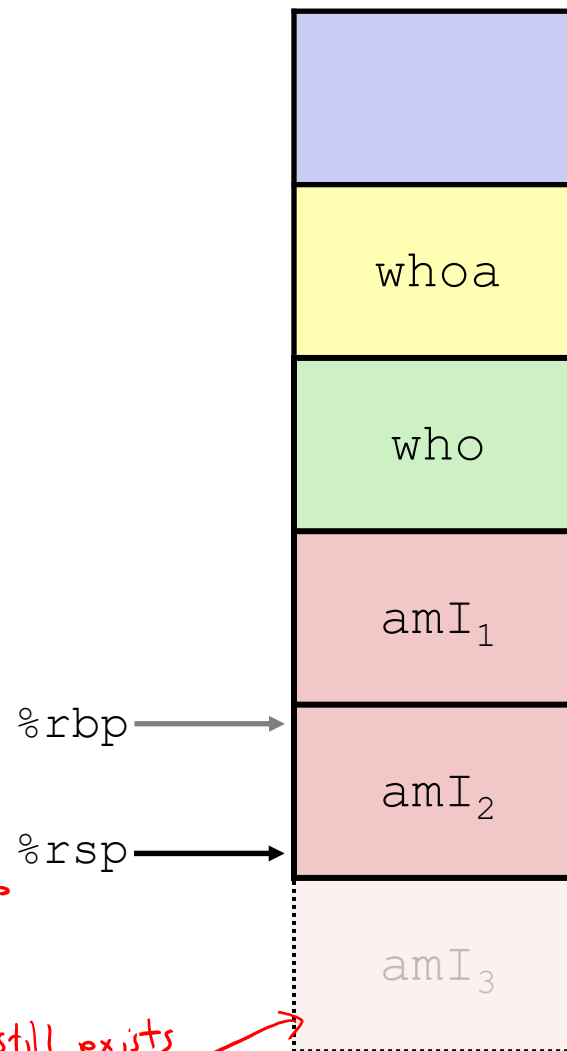
5) (another) Recursive call to amI (3)



6) Return from (another) recursive call to amI



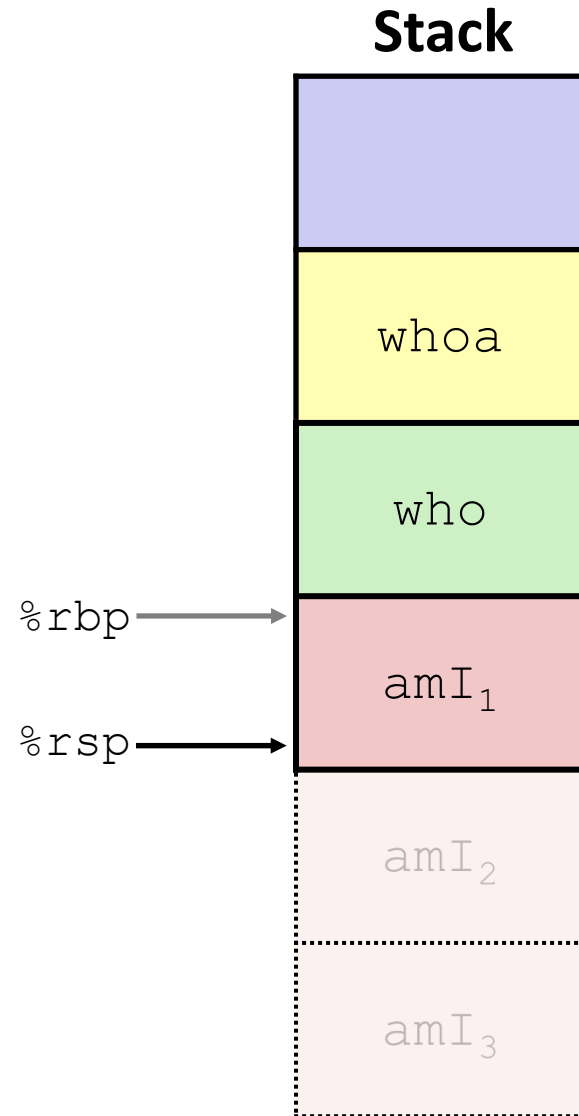
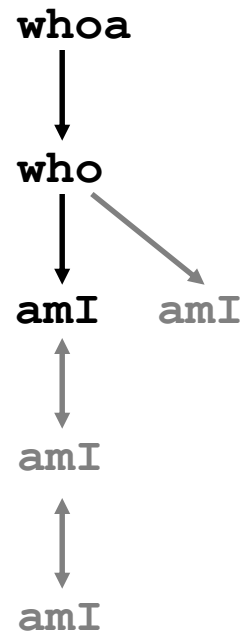
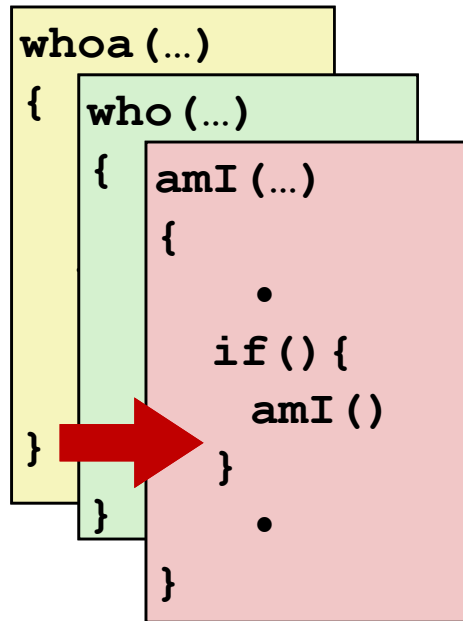
Stack



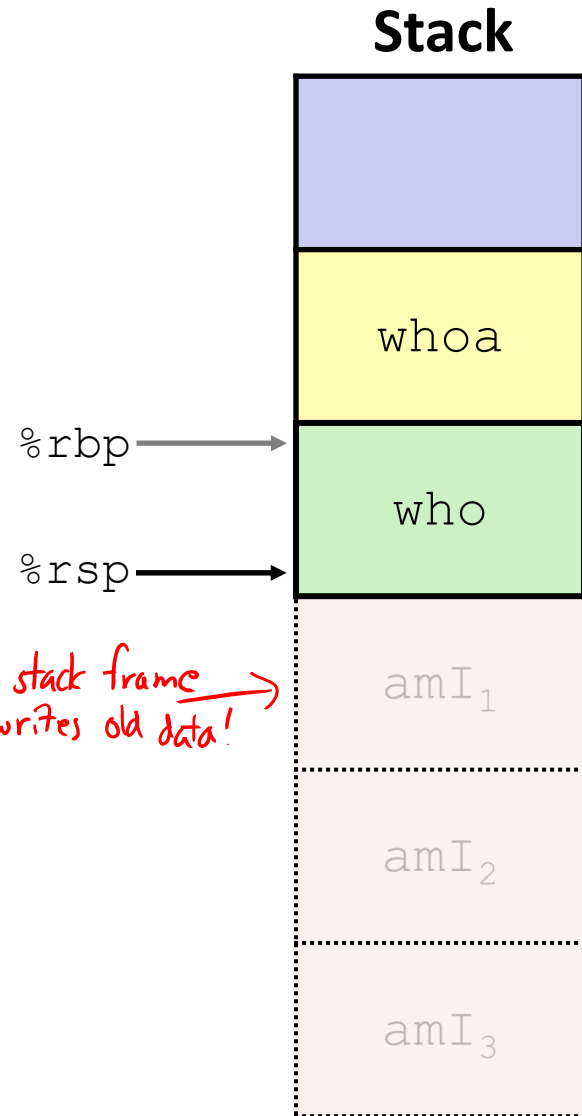
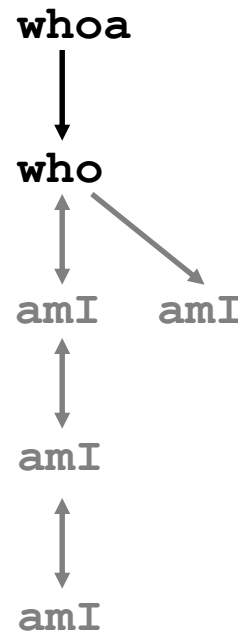
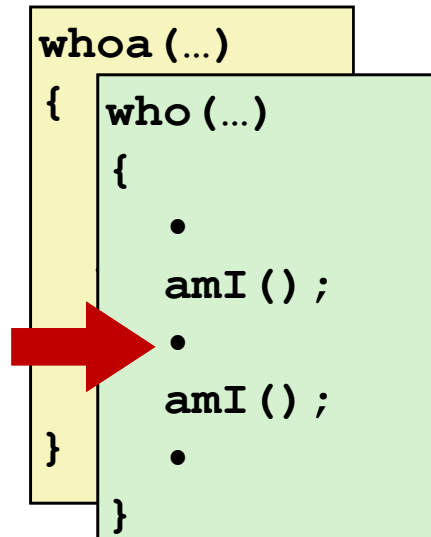
"deallocate" stack frame by moving %rsp back up

data still exists, but you shouldn't use it

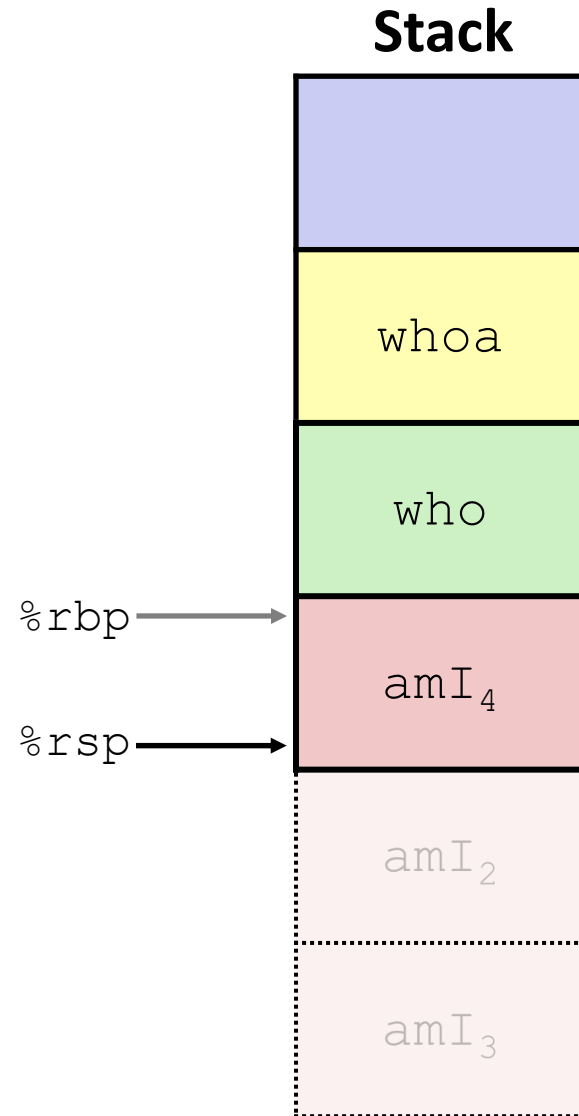
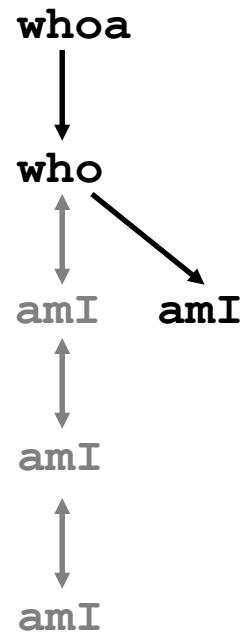
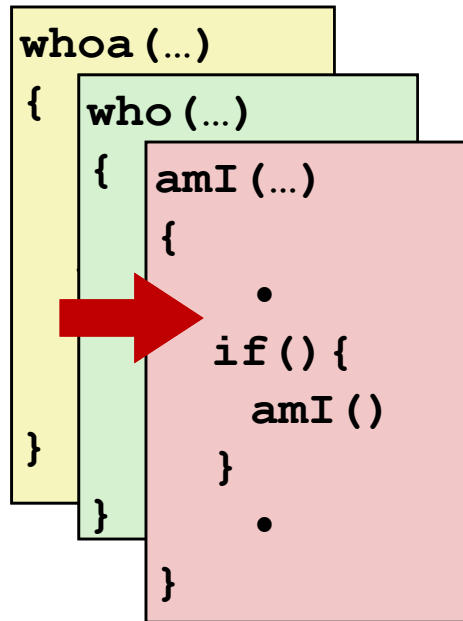
7) Return from recursive call to amI



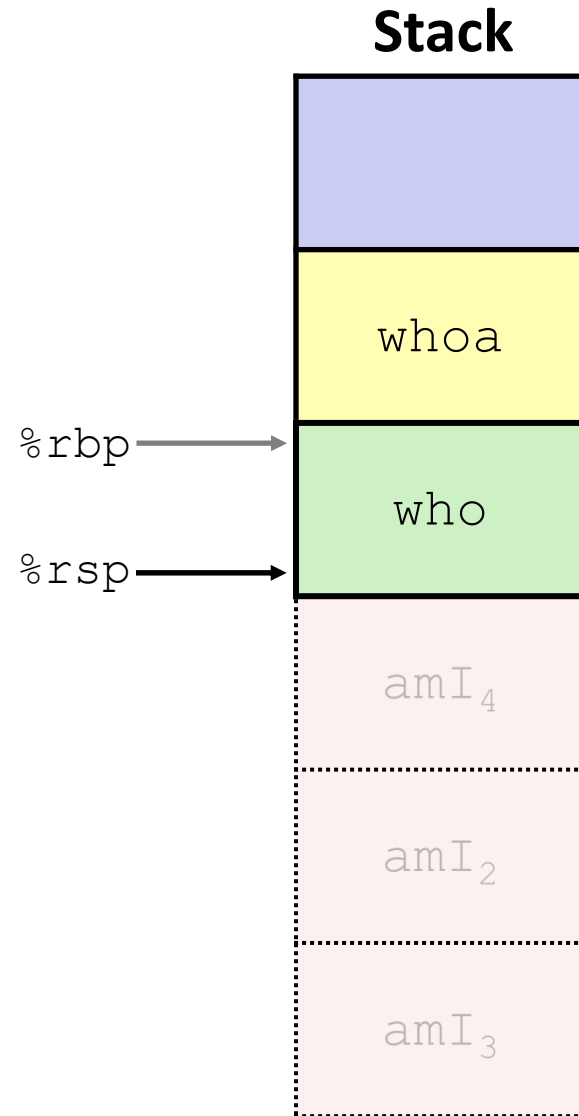
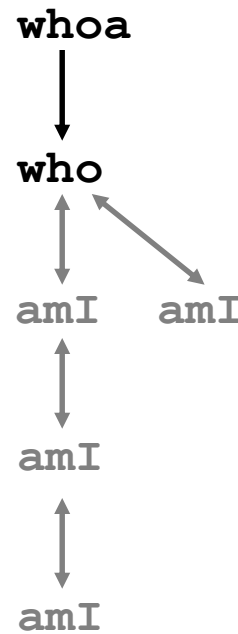
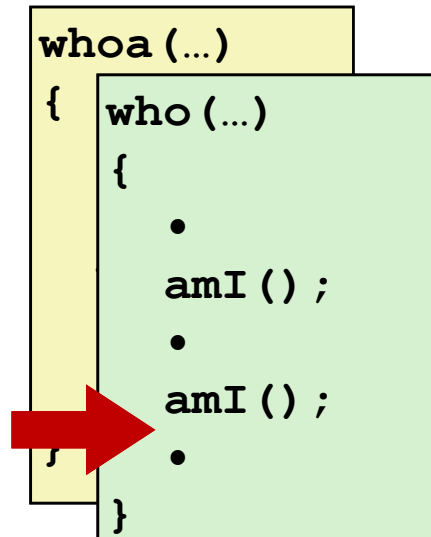
8) Return from call to amI



9) (second) Call to amI (4)



10) Return from (second) call to amI

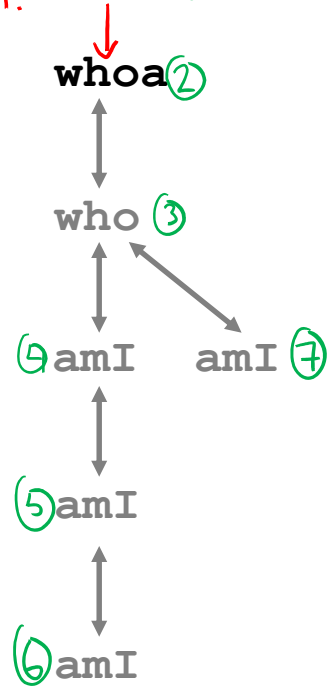


11) Return from call to who

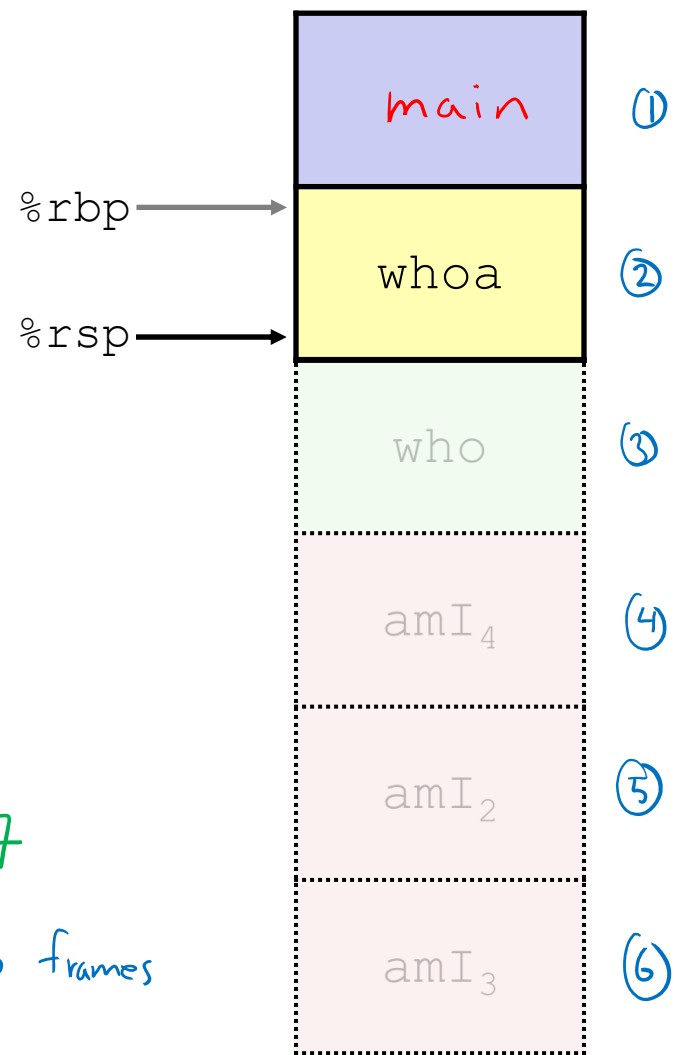
```

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

call chain: main ①



Stack



total stack frames created: 7
 maximum stack depth: 6 frames

x86-64/Linux Stack Frame

- ❖ **Caller's Stack Frame**
 - Extra arguments (if > 6 args) for this call
- ❖ **Current/Callee Stack Frame**
 - Return address
 - Pushed by `call` instruction
 - Old frame pointer (optional)
 - Saved register context (when reusing registers)
 - Local variables (If can't be kept in registers)
 - "Argument build" area (If callee needs to call another function - parameters for function about to call, if needed)

