The Hardware/Software Interface

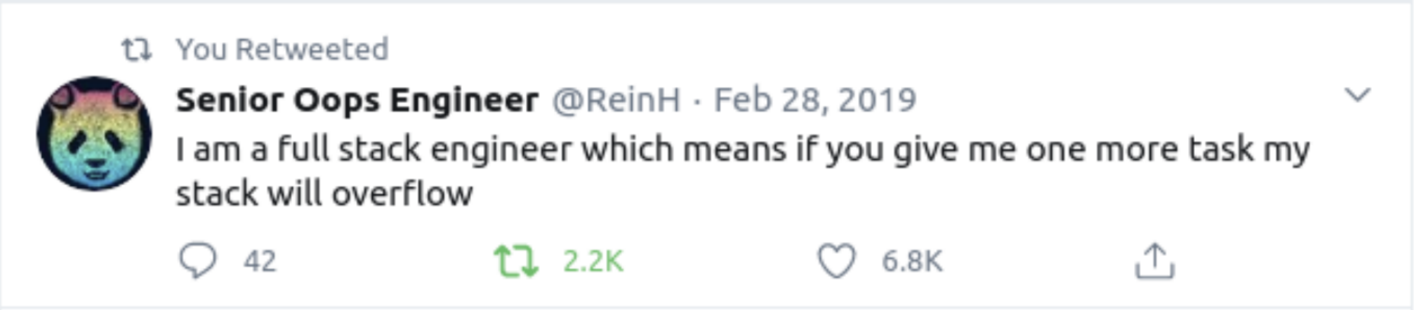
Procedures I

Instructors:

Justin Hsia, Amber Hu

Teaching Assistants:

Anthony Mangus, Divya Ramu, Grace Zhou, Jessie Sun, Jiuyang Lyu, Kanishka Singh, Kurt Gu, Liander Rainbolt, Mendel Carroll, Ming Yan, Naama Amiel, Pollux Chen, Rose Maresh, Soham Bhosale, Violet Monserate



Relevant Course Information

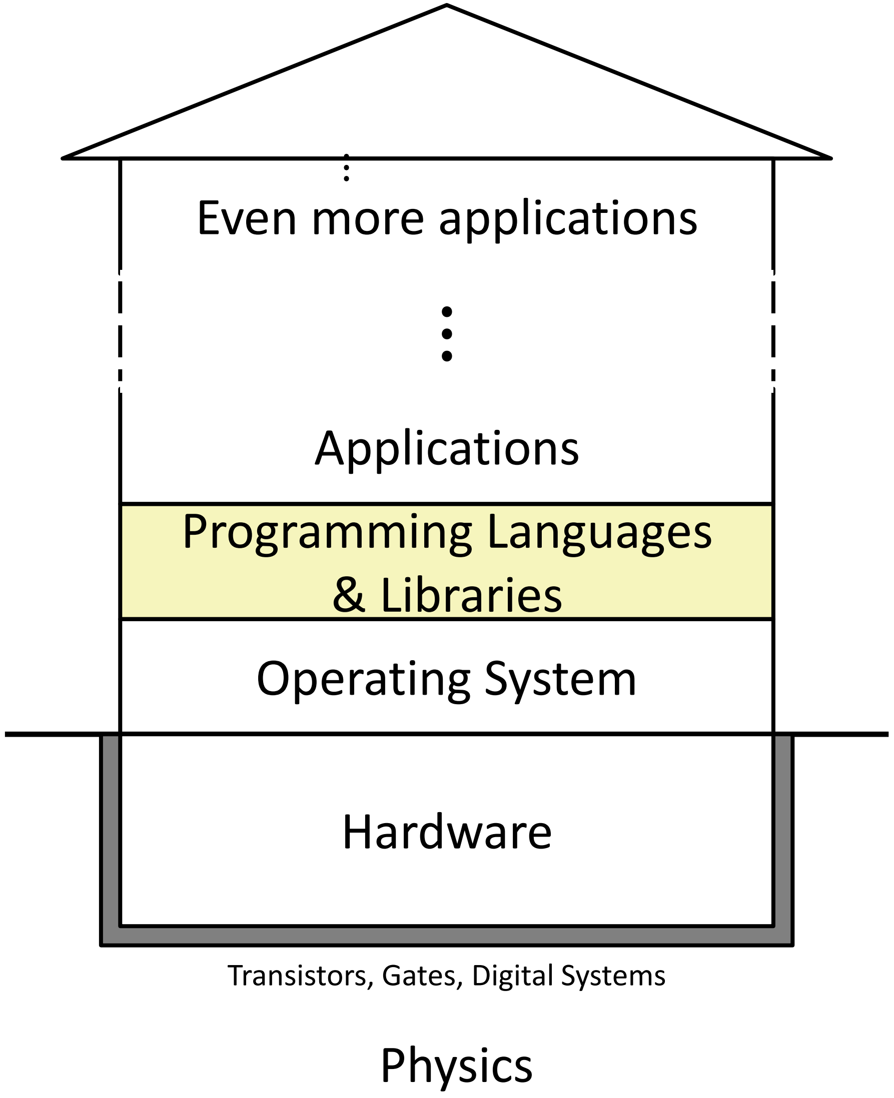
- Lab 2 due next Friday (10/24)
  - Can start in earnest after today's lecture!
  - See GDB Tutorial Lesson and and Phase 1 walkthrough in Section 4 Lesson

- Midterm: 10/27, 5:30 pm in ARC 147, CSE2 G20, SIG 134
  - You will be provided a fresh midterm reference sheet
  - You get 1 *handwritten*, double-sided cheat sheet (letter-size)
  - Form study groups and look at past exams!

House of Computing Check-In

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

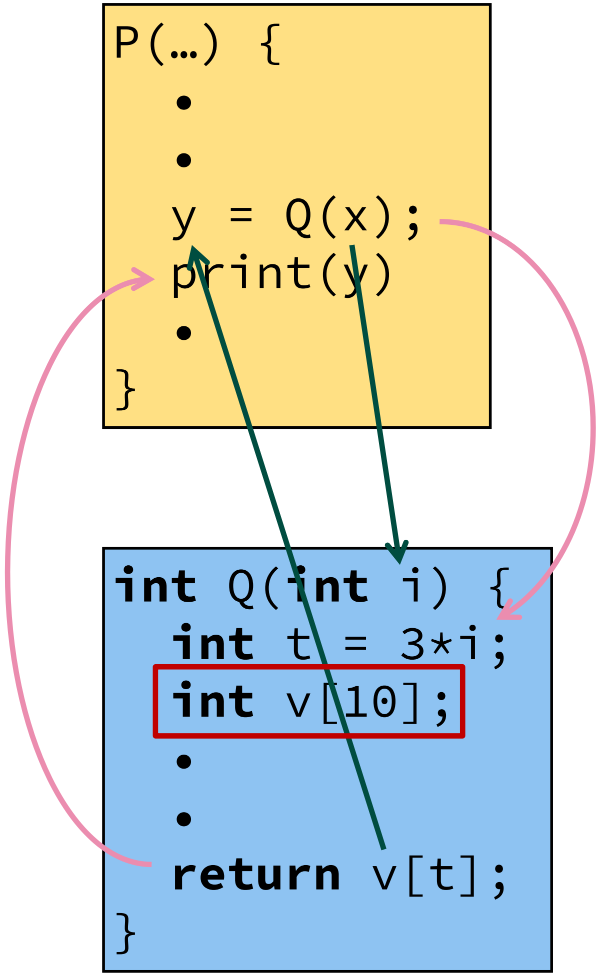
- How are programs created and executed on a CPU?
  - How does your source code become something that your computer understands?
  - How does the CPU organize and manipulate local data?



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

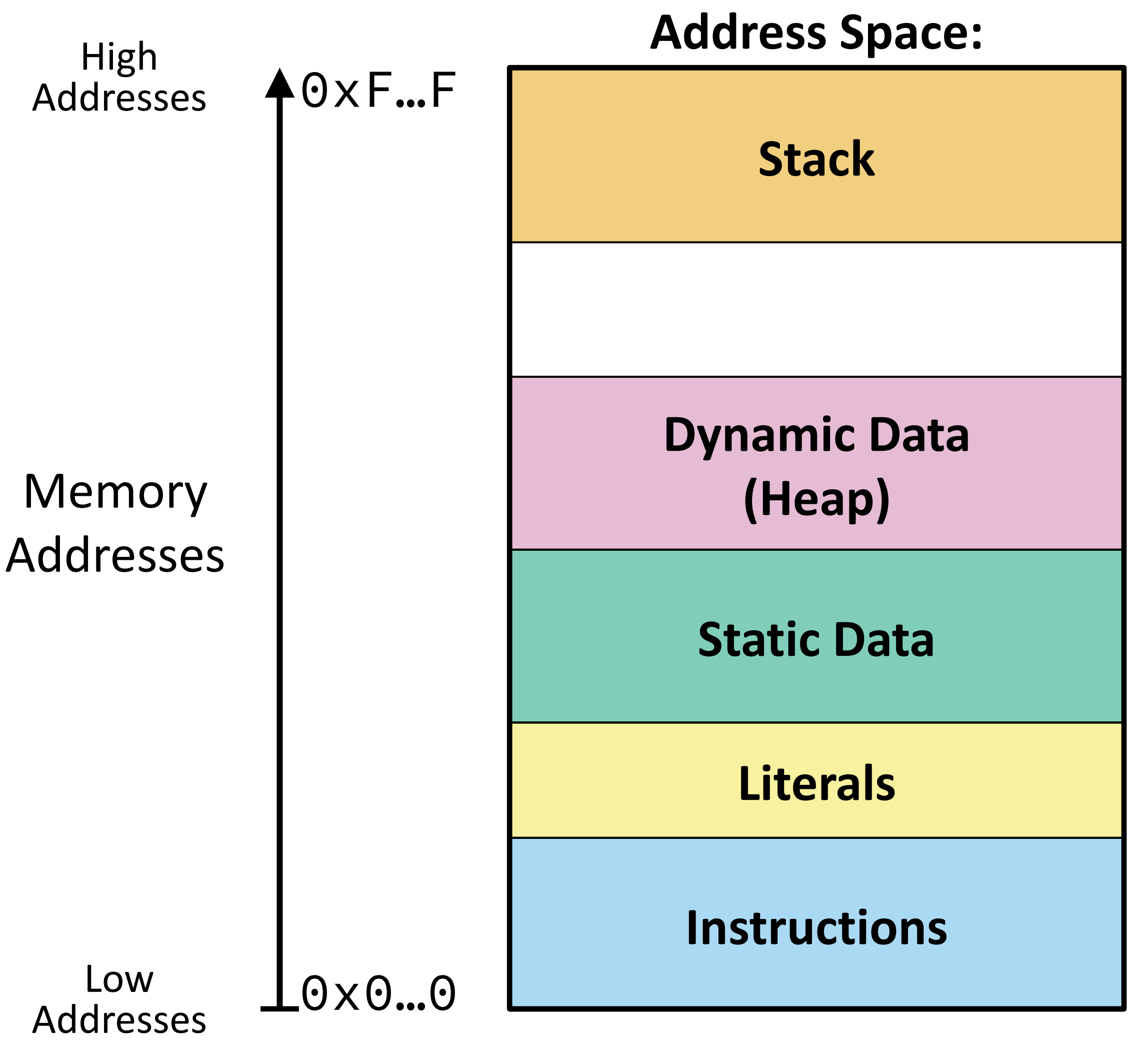
- All implemented with machine instructions!
  - An x86-64 procedure uses only those mechanisms required for that procedure



Lecture Outline (1/4)

- **Memory Layout**
- Stack Manipulation
- x86-64 Procedure Calling Conventions
- Stack Frames

Simplified Memory Layout (Review)



What Goes Here:

Local variables and procedure context

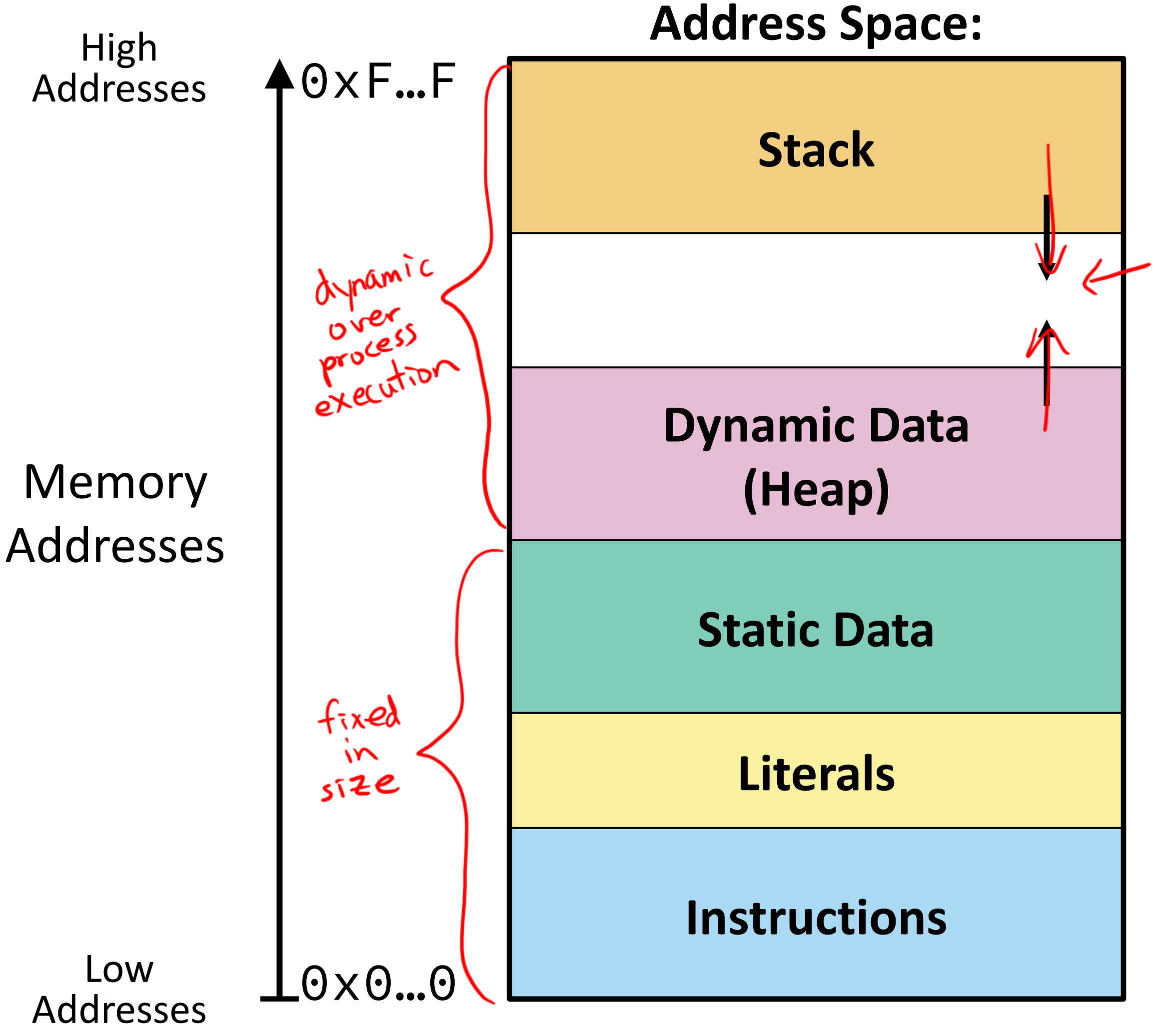
Variables allocated with new or `malloc`

Static variables (including global variables)

Immutable literals/constants (*e.g.*, `"example"`)

Program code

Memory Management



grow towards each other to maximize use of space

How Managed:

"Automatically" by compiler/assembly

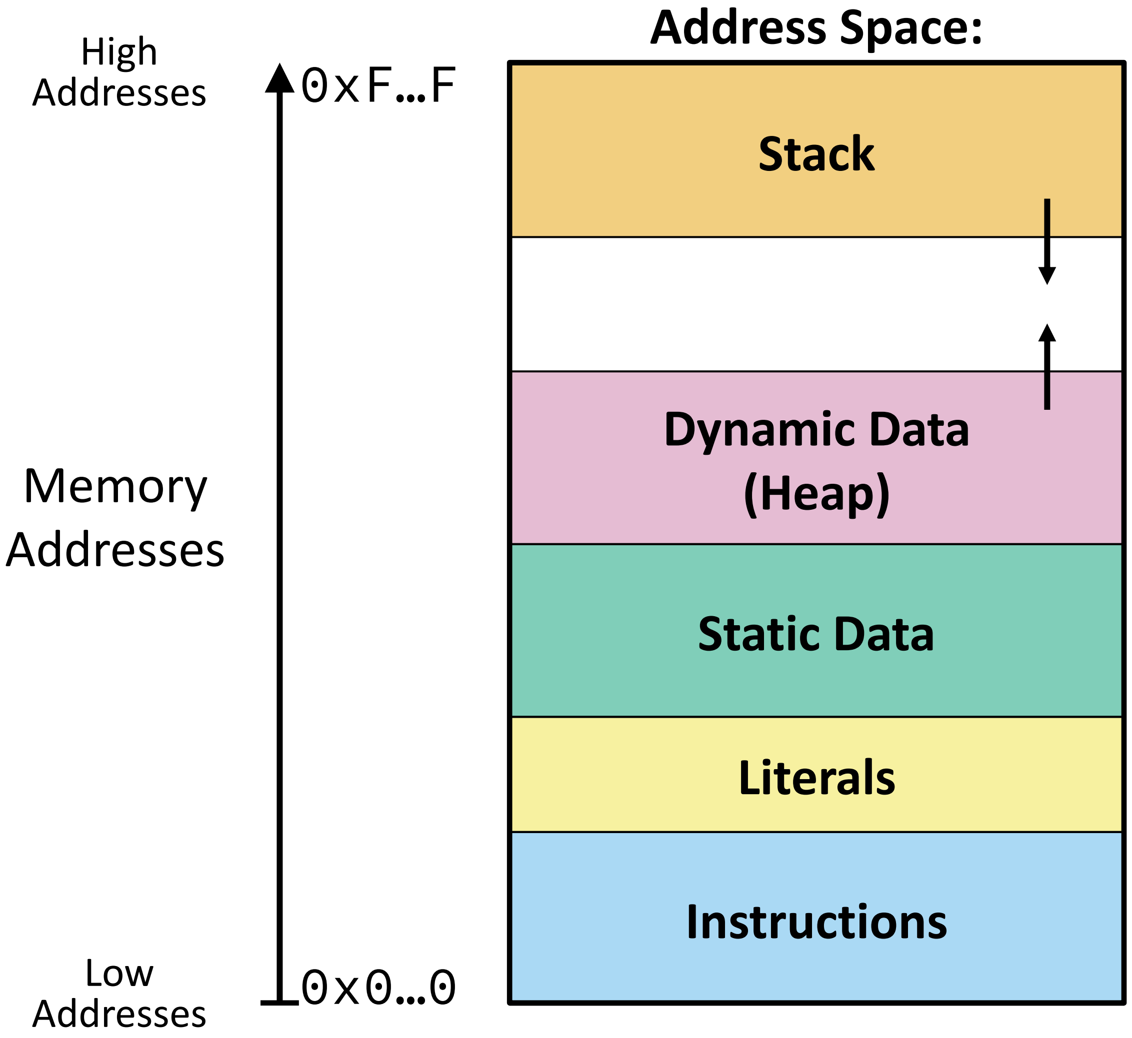
"Dynamically" by programmer

"Statically" at process start

"Statically" at process start

"Statically" at process start

Memory Permissions



Permissions:

Writable; not executable

Writable; not executable

Writable; not executable

Read-only; not executable

Read-only; executable

Segmentation fault: impermissible memory access!

Lecture Outline (2/4)

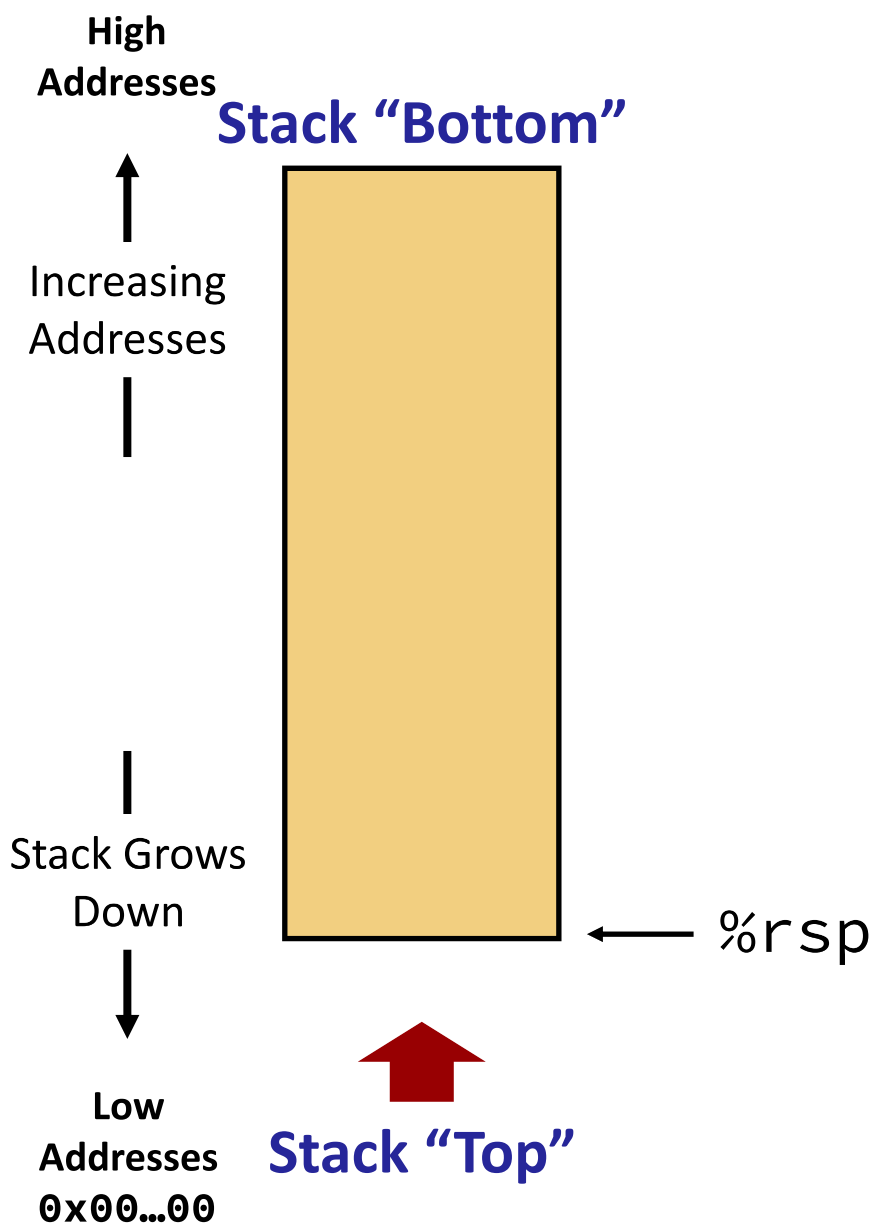
- Memory Layout
- **Stack Manipulation**
- x86-64 Procedure Calling Conventions
- Stack Frames

x86-64 Stack (Review)

Last In, First Out (LIFO)

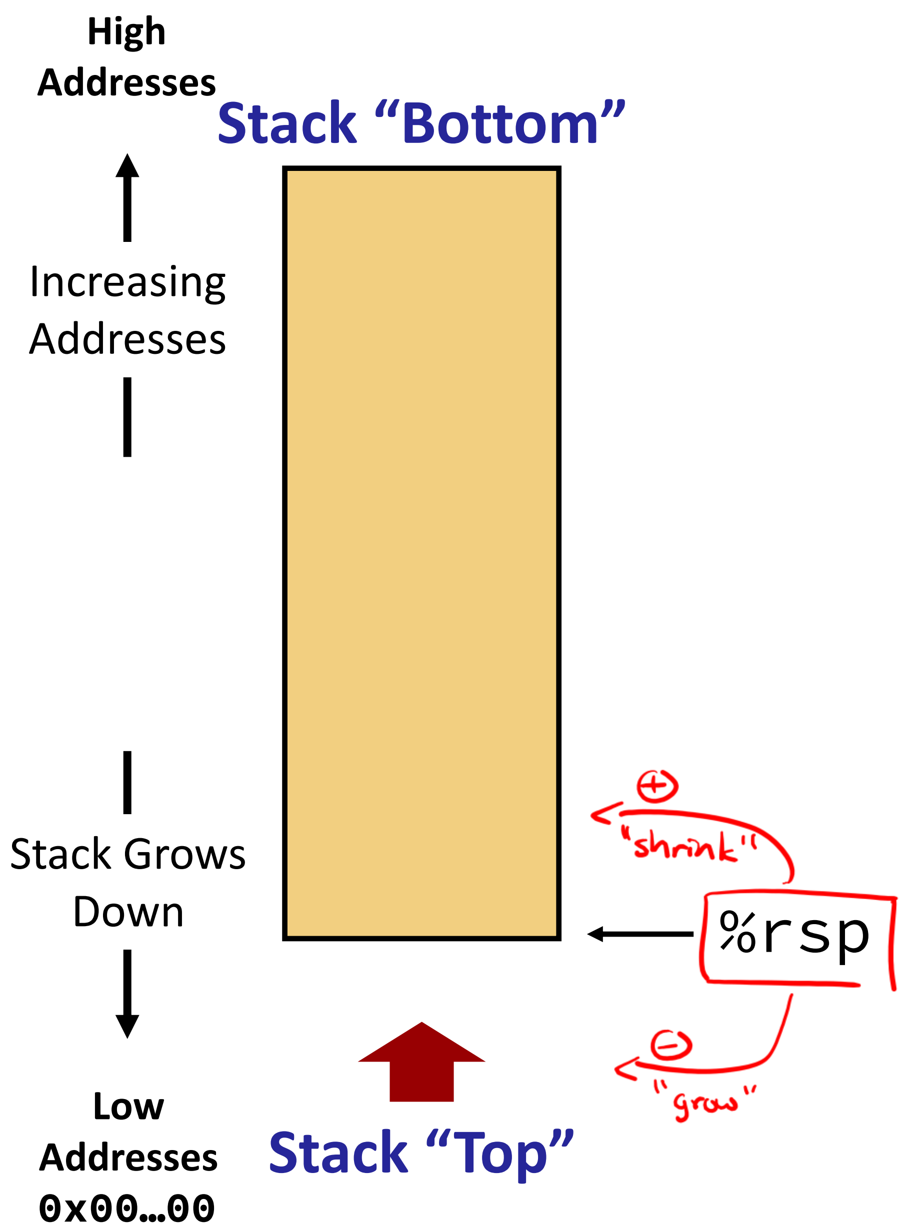
- Region of memory managed automatically via assembly instructions
  - Grows toward lower addresses
  - Customarily shown "upside-down"

- Register `%rsp` is the ***stack pointer***
  - Contains the *lowest* stack address (*top* element)
  - Useful reference point – stack data accessed via nonnegative offsets
    - Example: `8(%rsp)` is 8 bytes from the top/end of the stack



x86-64 Stack: Add/Sub (Review)

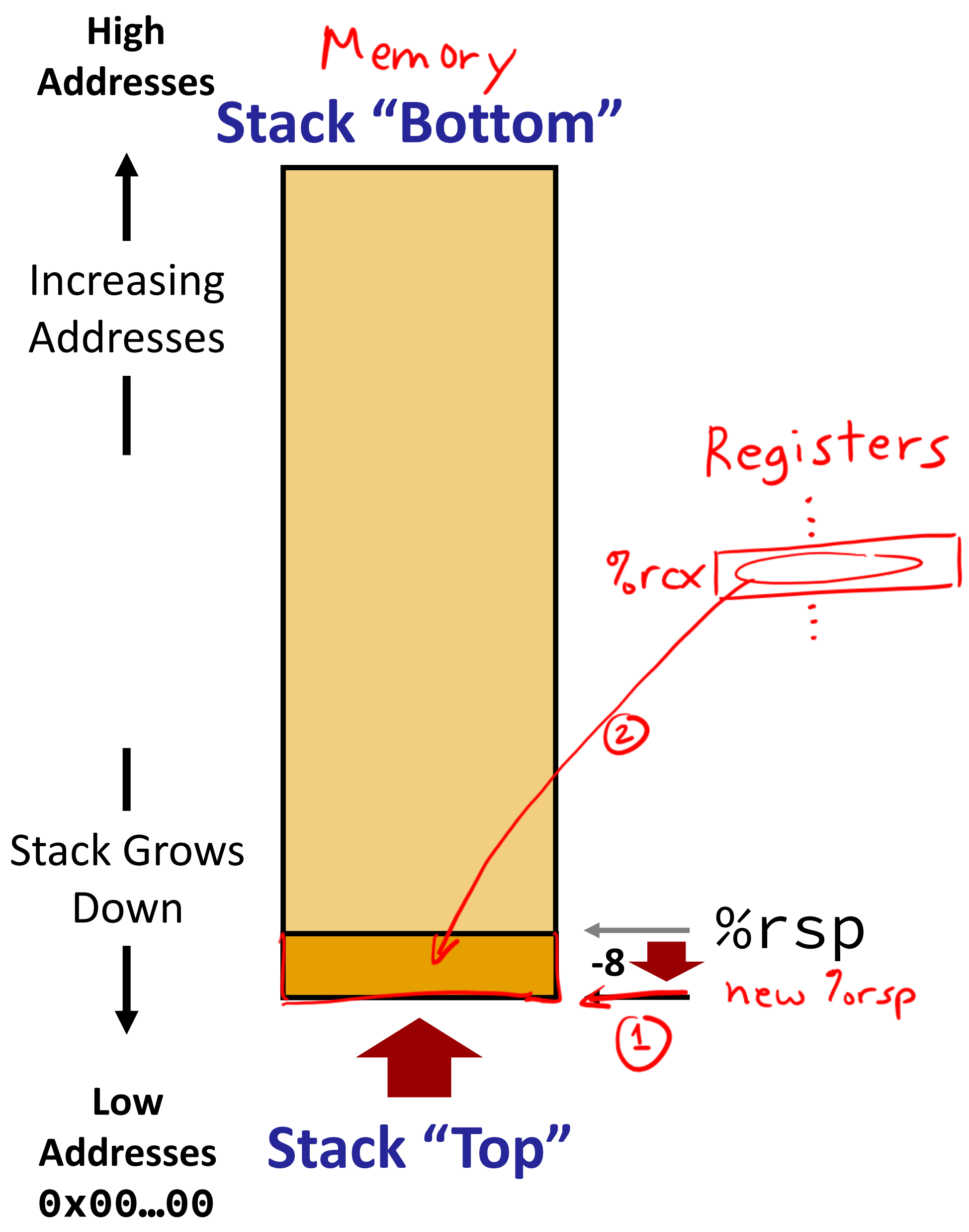
- The size of the stack can be directly manipulated
  - Remember which direction is grow/shrink
  - Example: **subq** `$8, %rsp`
- Doesn't change the data on the stack
  - Need specialized instructions to avoid separate instructions for (1) changing size and (2) manipulating data



x86-64 Stack: Push (Review)

- **push_** *src*
  size specifier
  1) *Decrement* `%rsp` by specified size
  2) Store *src* operand value at `%rsp`'s address
     - Operand can be Reg, Mem, or Imm

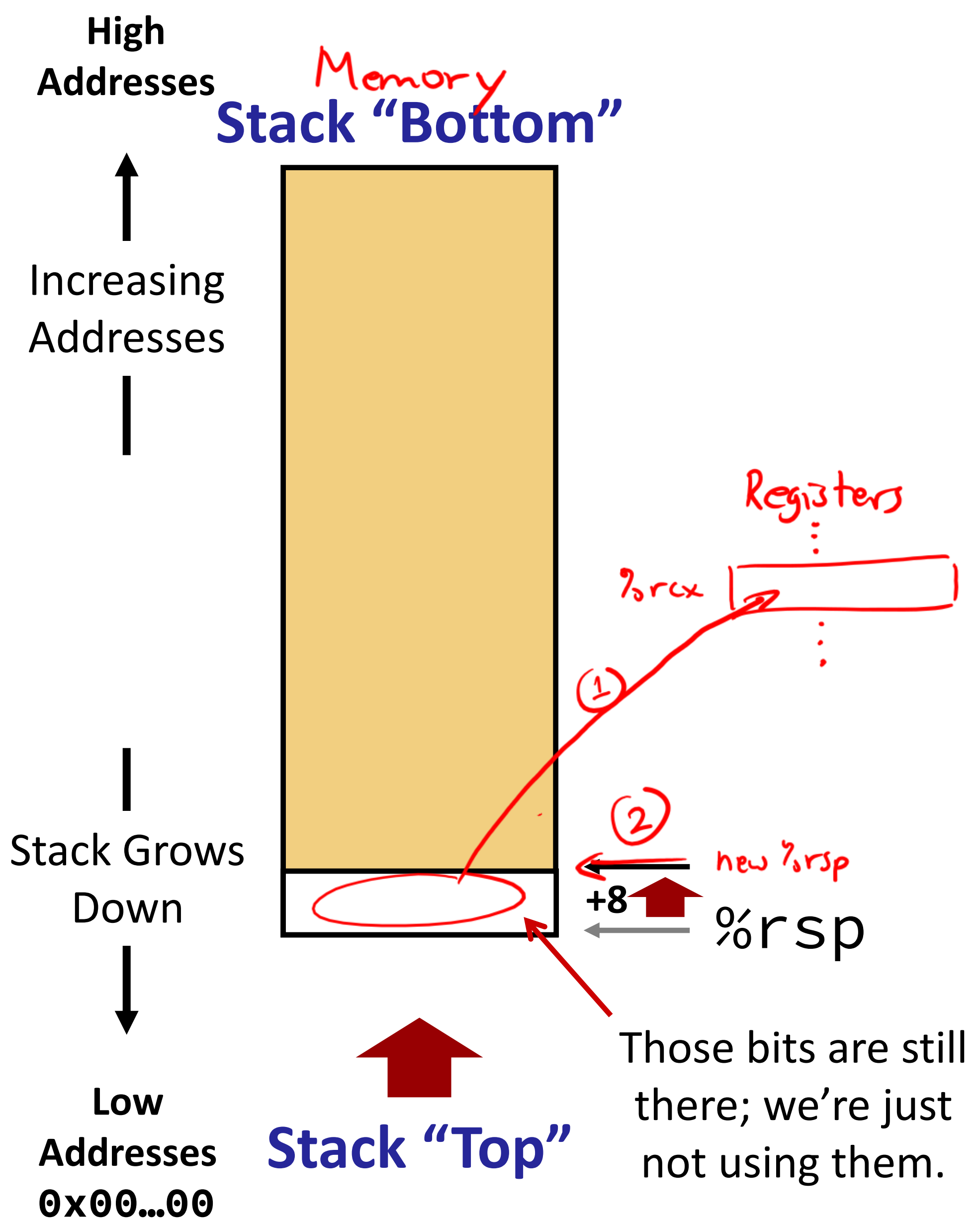
- Example: **pushq** `%rcx`
  - Decrement `%rsp` by 8 bytes
  - Store copy of `%rcx` on the stack



x86-64 Stack: Pop (Review)

- **pop_** *dst*
  size specifier
  1) Store value at address given by `%rsp` into *dst*
     - Operand can be Reg or Mem
  2) *Increment* `%rsp` by specified size

- Example: **popq** `%rcx`
  - Store copy of top 8 bytes of the stack into `%rcx`
  - Increment `%rsp` by 8 bytes

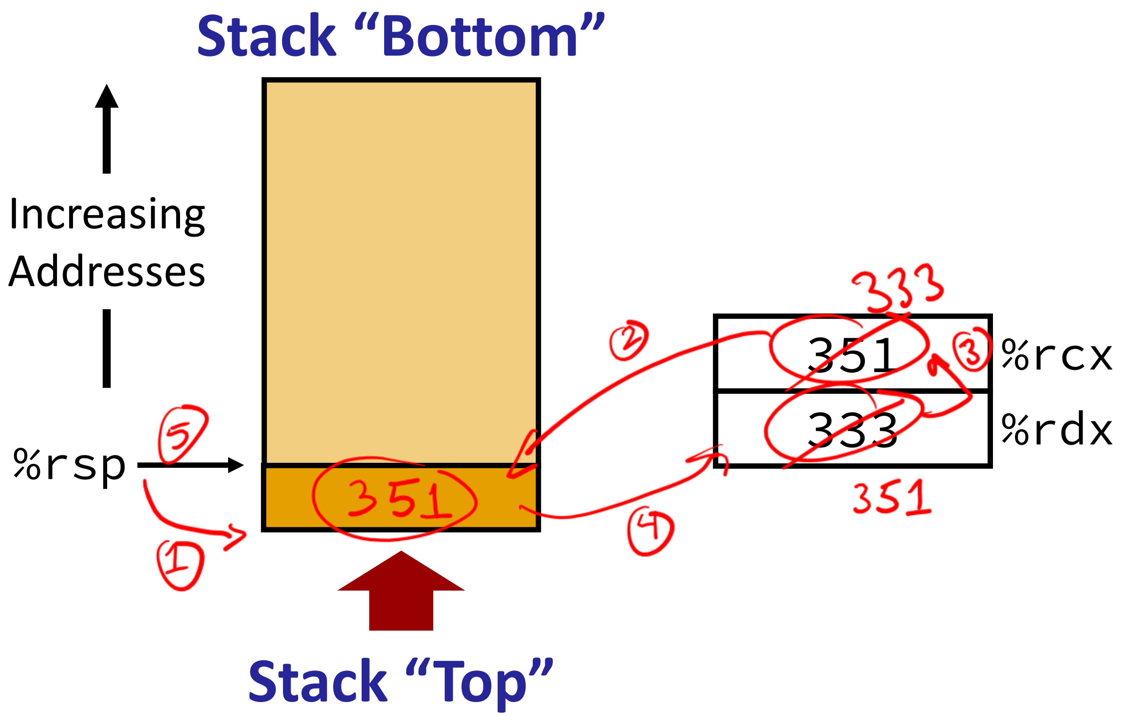


Stack Manipulation Example

- Using the stack for temporary space to swap `%rcx` and `%rdx`
  - Note: You wouldn't want to do this because memory is slower than registers

```
swap_stack1:
   1 subq $8, %rsp
   2 movq %rcx, (%rsp)
   3 movq %rdx, %rcx
   4 movq (%rsp), %rdx
   5 addq $8, %rsp
     ret
```

```
swap_stack2:
     pushq %rcx
     movq %rdx, %rcx
     popq %rdx
     ret
```



Polling Questions (1/3)

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

add to stack

8B

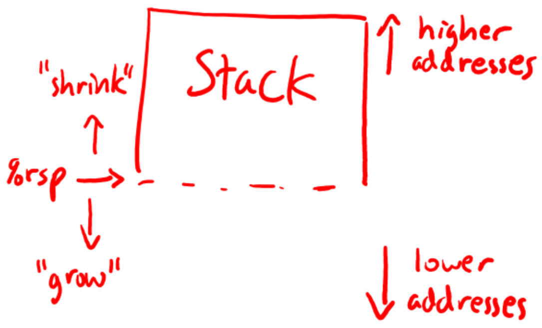
```
pushq  %rbp            # grow 8 B
```

lower address

```
subq   $0x18, %rsp     # grow 24 B
```

0x18 = 24

grow by 32 B

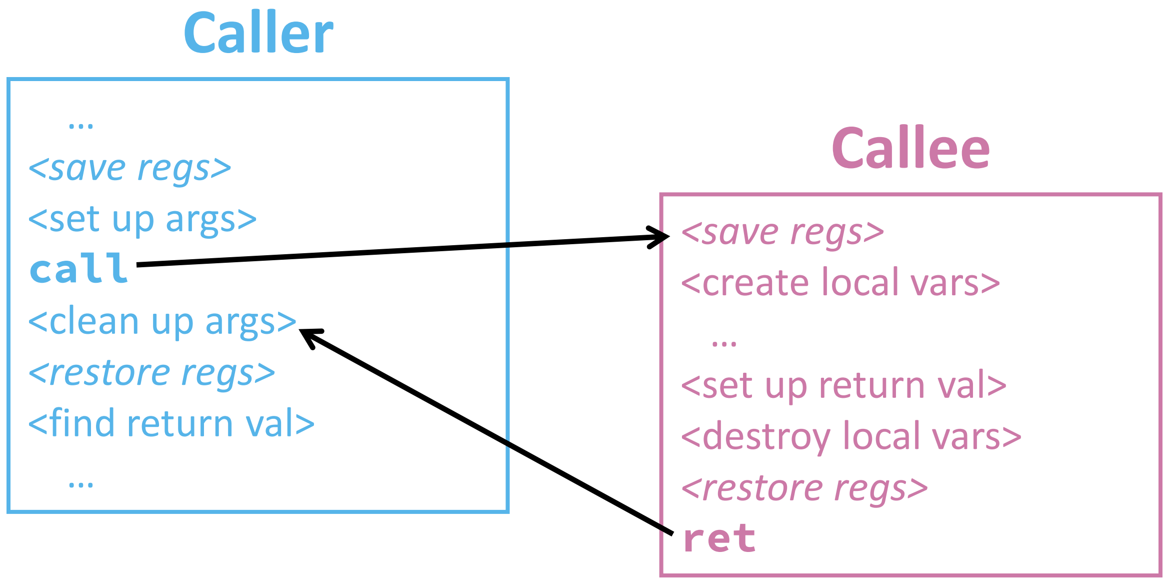


Lecture Outline (3/4)

- Memory Layout
- Stack Manipulation
- **x86-64 Procedure Calling Conventions**
- Stack Frames

Calling Conventions (Review)

- Set of rules to guarantee that procedures can pass *control* and *data* to one another (*i.e.*, where to leave and find things)
  - Goal is to allow register and memory reuse without losing/corrupting data
  - The procedure doing the calling is the ***caller***
  - The procedure being called is the ***callee***



Code Example: Preview

Compiler Explorer: https://godbolt.org/z/TYG6x44Gs

executable disassembly

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

Caller

```
000000000040110e <multstore>:
  40110e: push   %rbx              # save %rbx
  40110f: movq   %rdx,%rbx         # save dst
  401112: call   401106 <mult2>    # mult2(x,y)
  401117: movq   %rax,(%rbx)       # save at dst
  40111a: pop    %rbx              # restore %rbx
  40111b: ret                      # return
```

these are instruction addresses

Callee

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

```
0000000000401106 <mult2>:
  401106:  movq   %rdi,%rax   # a
  401109:  imulq  %rsi,%rax   # a * b
  40110d:  ret                # return
```

Procedures: Passing Control (Review)

- *Return address* indicates how to return to the Caller
  - Callee can be invoked from multiple places in code
  - Which address?

```
401112: call   401106 <mult2>
401117: movq   %rax,(%rbx)
```

next instruction could be anything

- Address of instruction immediately *after* the `call` instruction (**`0x401117`**)

- **`call`** `label` – pass control to Callee (special push)
  1) Automatically push the return address onto the stack
  2) Update the program counter (`%rip`) to the address of the specified label

① move %rsp down
② store ret addr at %rsp

- **`ret`** – return control to Caller (special pop)
  1) Automatically pop the return address off of the stack and then
  2) Update the program counter (`%rip`) to the popped address

① read ret addr at %rsp into %rip
② move %rsp up

Procedure Call Example (Step 1)

```
0000000000040110e <multstore>:
  •
  •
  401112: call    401106 <mult2>
  401117: movq    %rax,(%rbx)
  •
  •
```

```
0000000000401106 <mult2>:
  401106:   movq   %rdi,%rax
  •
  •
  40110d:   ret
```

2

3

Stack
(Memory)

0x130	•
0x128	•
0x120	•
	0x401117

1

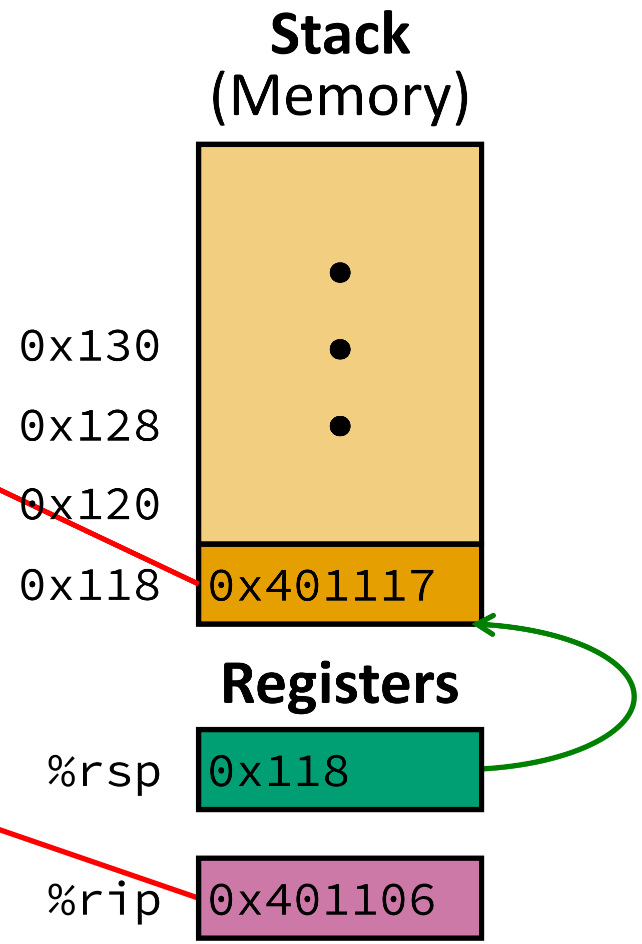
Registers

%rsp	~~0x120~~ 118
%rip	~~0x401112~~ 0x401106

Procedure Call Example (Step 2)

```
0000000000040110e <multstore>:
  •
  •
  401112: call   401106 <mult2>
  401117: movq   %rax,(%rbx)
  •
  •
```

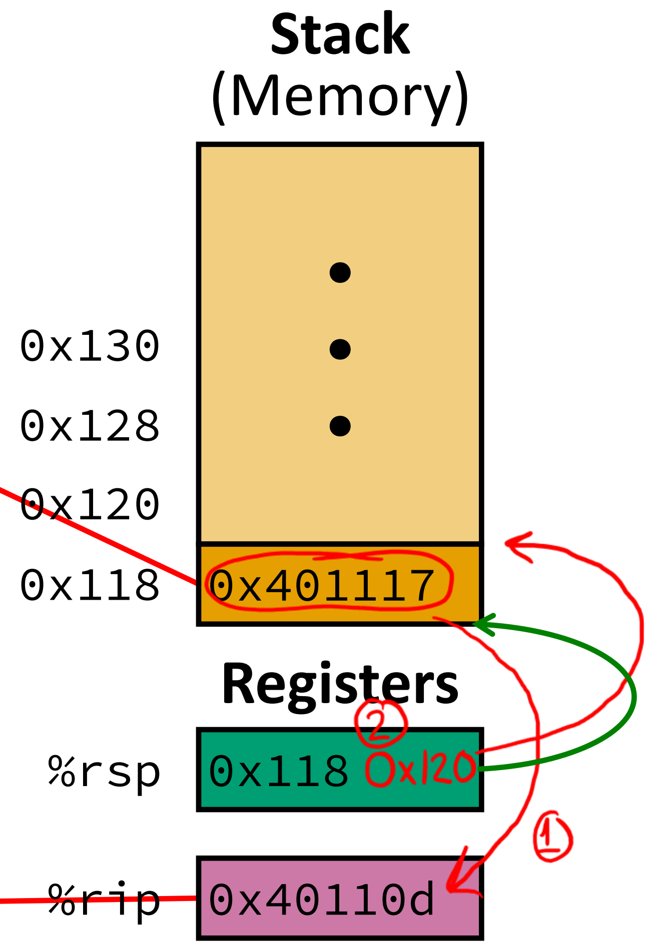
```
0000000000401106 <mult2>:
  401106:  movq   %rdi,%rax
  •
  •
  40110d:  ret
```



Procedure Return Example (Step 1)

```
0000000000040110e <multstore>:
  •
  •
  401112: call   401106 <mult2>
  401117: movq   %rax,(%rbx)
  •
  •
```

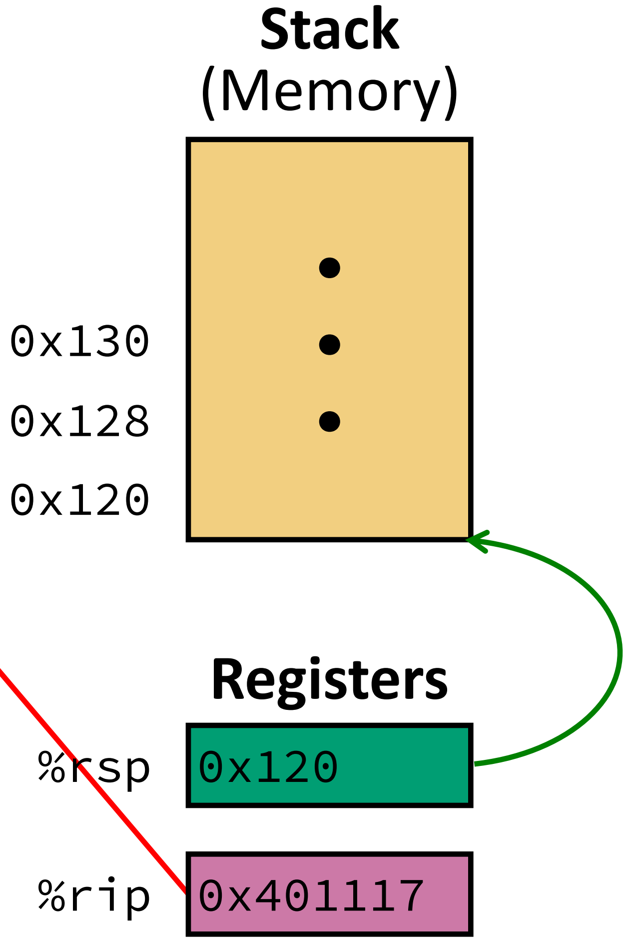
```
0000000000401106 <mult2>:
  401106:  movq   %rdi,%rax
  •
  •
  40110d:  ret
```



Procedure Return Example (Step 2)

```
000000000040110e <multstore>:
  •
  •
  401112: call   401106 <mult2>
  401117: movq   %rax,(%rbx)
  •
  •
```

```
0000000000401106 <mult2>:
  401106:  movq   %rdi,%rax
  •
  •
  40110d:  ret
```



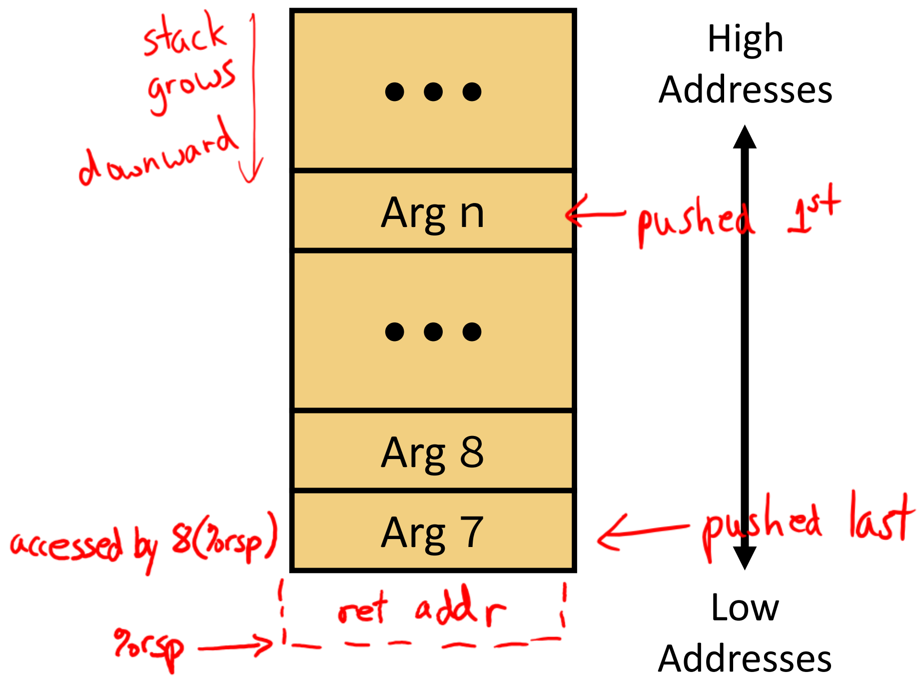
Procedures: Passing Data (Review)

- Registers (NOT in Memory)
  - First 6 arguments:

1	%rdi	Diane's
2	%rsi	Silk
3	%rdx	Dress
4	%rcx	Costs
5	%r8	$8
6	%r9	9

  - Return value: %rax
    - Return pointer for anything >8 bytes wide
    - Caller can directly use `%rax` after return

- Stack (Memory)
  - Only allocate memory if needed



Code Example: Passing Data

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

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

rdi rsi rdx

rdi rsi

```
0000000000040110e <multstore>:
  # x in %rdi, y in %rsi, dst in %rdx
    • • •
  40110f: movq   %rdx,%rbx        # "save" dst
  401112: call   401106 <mult2> # mult2(x,y)
  # t in %rax
  401117: movq   %rax,(%rbx)     # save at dst
    • • •
```

will explain later

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

```
0000000000401106 <mult2>:
  # a in %rdi, b in %rsi
  401106:  movq   %rdi,%rax   # a
  401109:  imulq  %rsi,%rax   # a * b
  # s in %rax
  40110d:  ret                # return
```

Polling Questions (2/3)

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

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

return value in $\boxed{\%rax}$

arguments in $\boxed{\%rdi, \%rsi, \text{and } \%rdx}$

$\boxed{\%rsp}$ changed by call & ret

$\boxed{\%rip}$ changed while executing instructions

Lecture Outline (4/4)

- Memory Layout
- Stack Manipulation
- x86-64 Procedure Calling Conventions
- **Stack Frames**

Stack Organization (Review)

- Enables multiple instantiations of a procedure to be "running" simultaneously
  - Need some place to store *state* of each instantiation
  - This makes recursion possible!

- ***Stack discipline*** – how do we prevent bad things from happening?
  - Observation #1: State for a given procedure only needed for a limited time
  - Observation #2: Callee always returns before Caller does

- Stack organized in conceptual units called *stack frames*
  - Each stack frame contains the state for a single procedure instantiation

Stack Frames Example

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

```
who(…)
{
    •
  ①amI();
    •
  ②amI();
    •
}
```

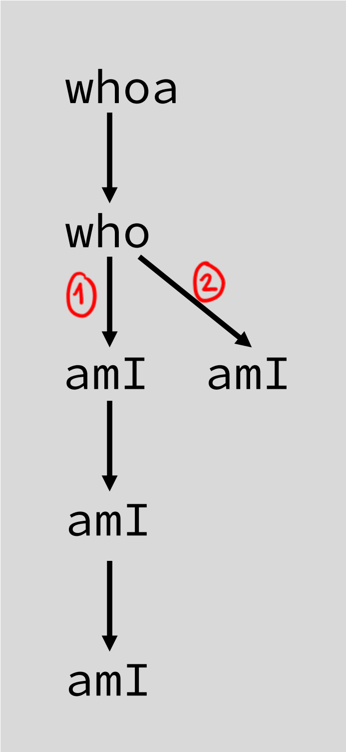
1st call recurses twice

2nd call doesn't recurse

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

based on condition

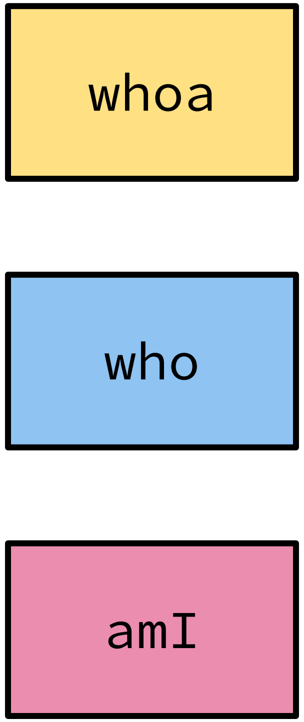
Example
Call Chain



Procedure amI is conditionally recursive (calls itself)

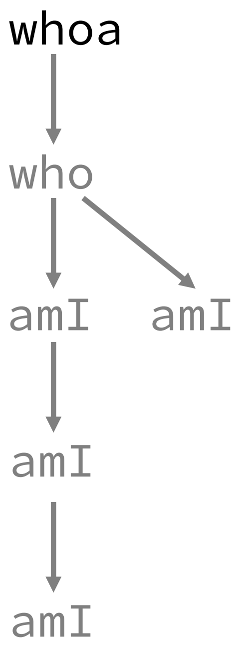
Stack Frame Management

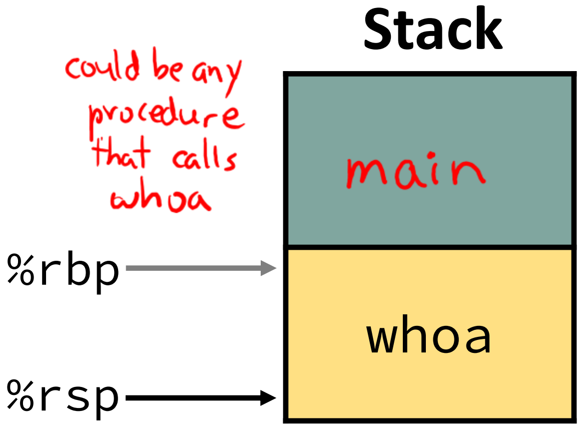
- Contents
  - All necessary local context – we'll see details later
  - Size will vary based on procedure specifics
    - Showing same size in this example, but not usually the case
- Management
  - Space gets allocated as procedure executes
  - Space must get deallocated by the time it returns
- Reference points
  - Stack pointer (`%rsp`) indicates the top of the stack and current frame
  - *Frame pointer* (`%rbp`) may indicate the "start"/"bottom" of current frame
    - Less commonly used this way in x86-64



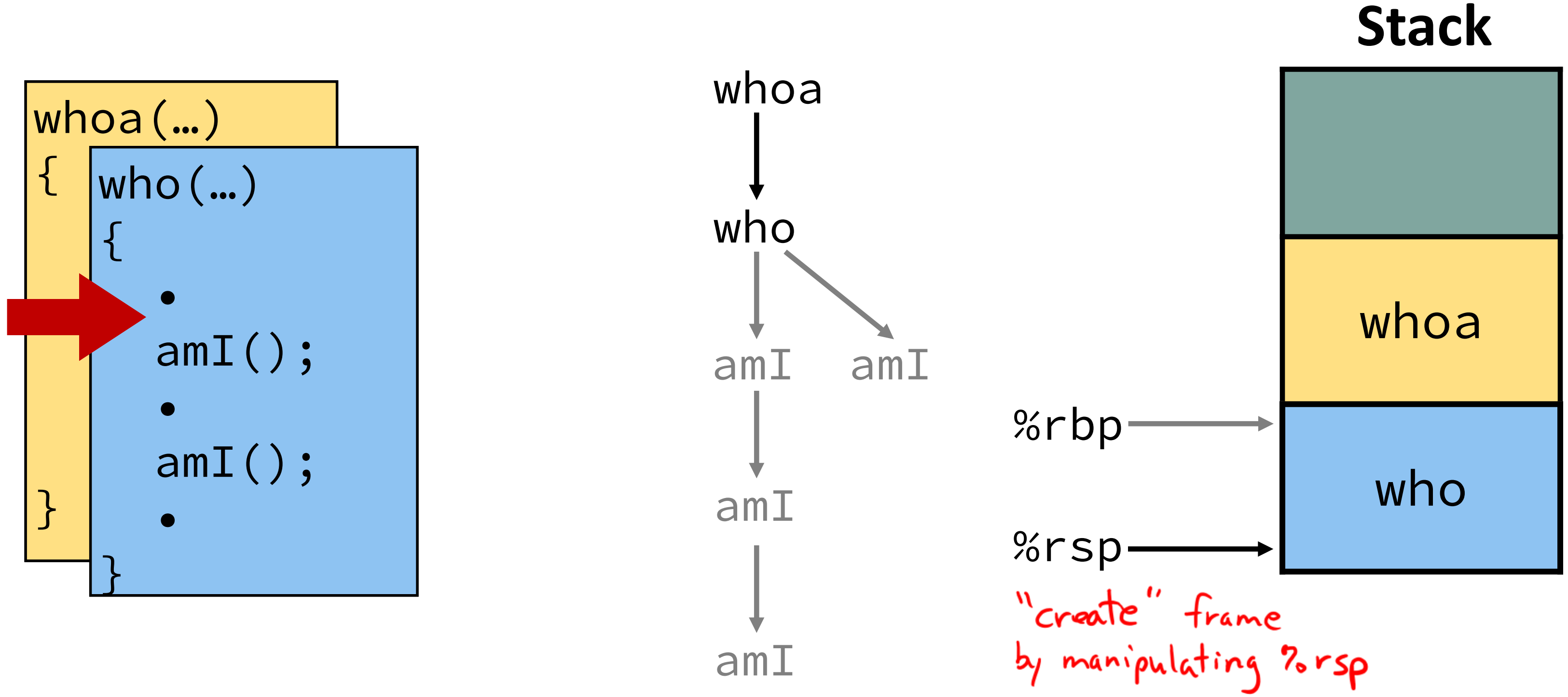
1) Call to whoa

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

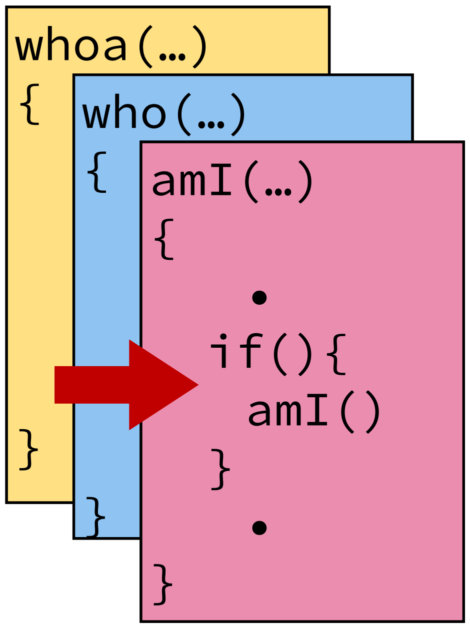


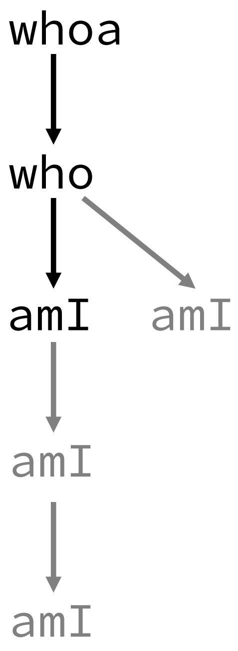


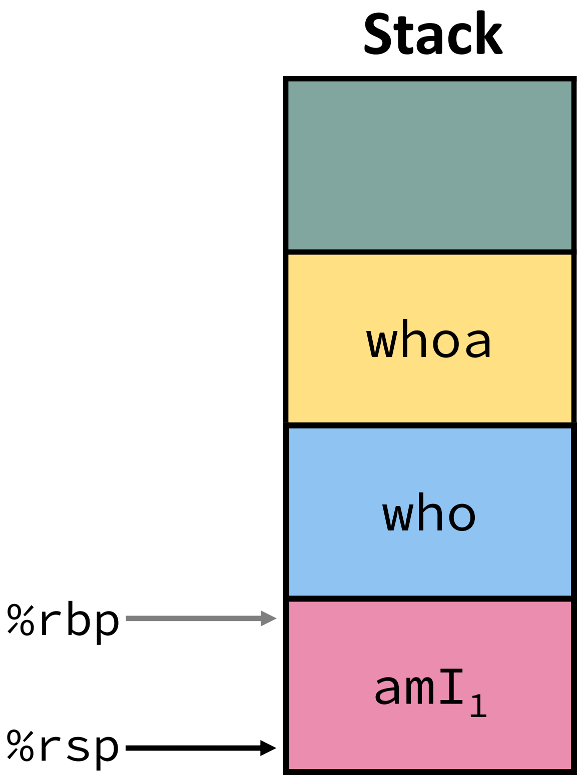
2) Call to who


whoa(…)
{
}
who(…)
{
•
amI();
•
amI();
•
}
whoa
who
amI
amI
amI
amI
Stack
whoa
who
%rbp
%rsp
"create" frame
by manipulating %rsp

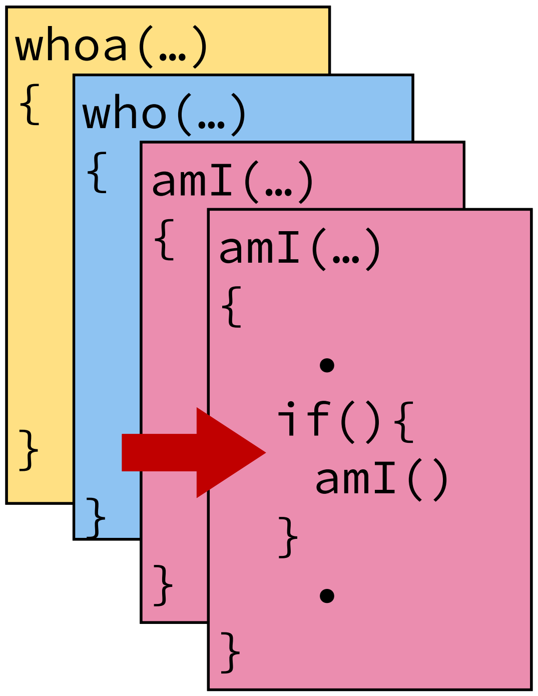
3) Call to `amI` (1st)

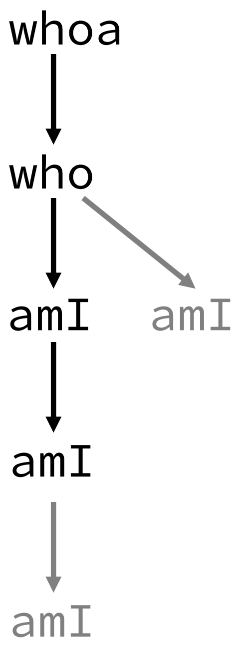


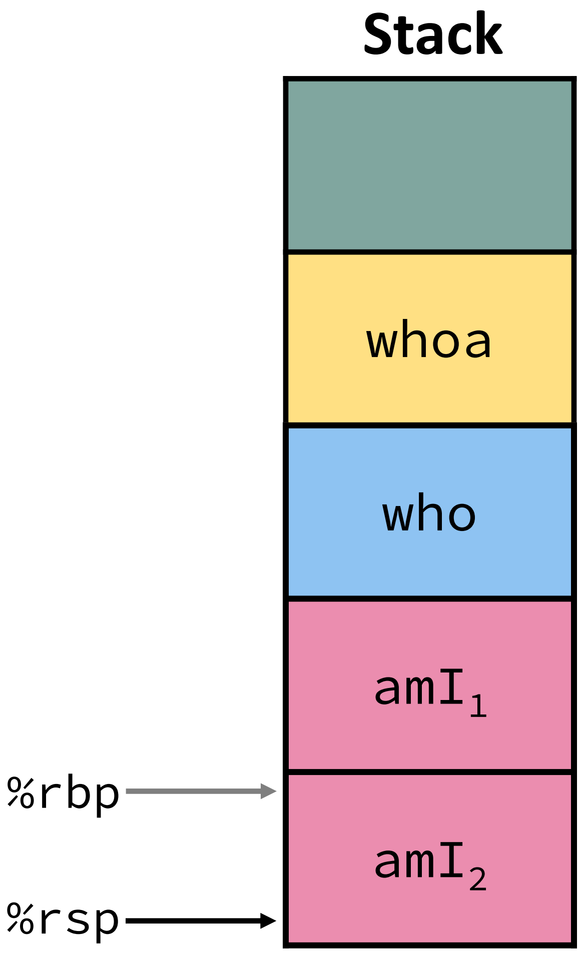




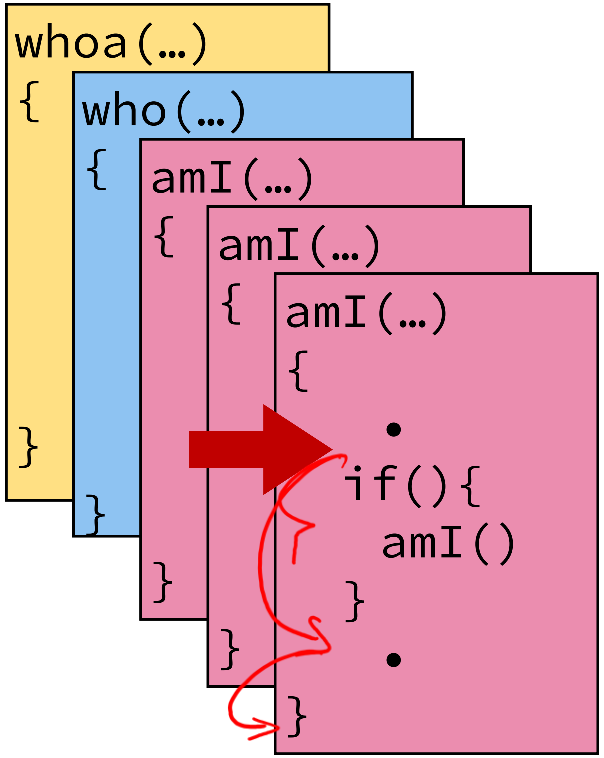
4) Recursive call to `amI` (2nd)

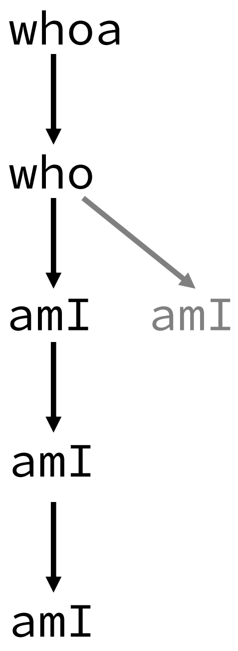


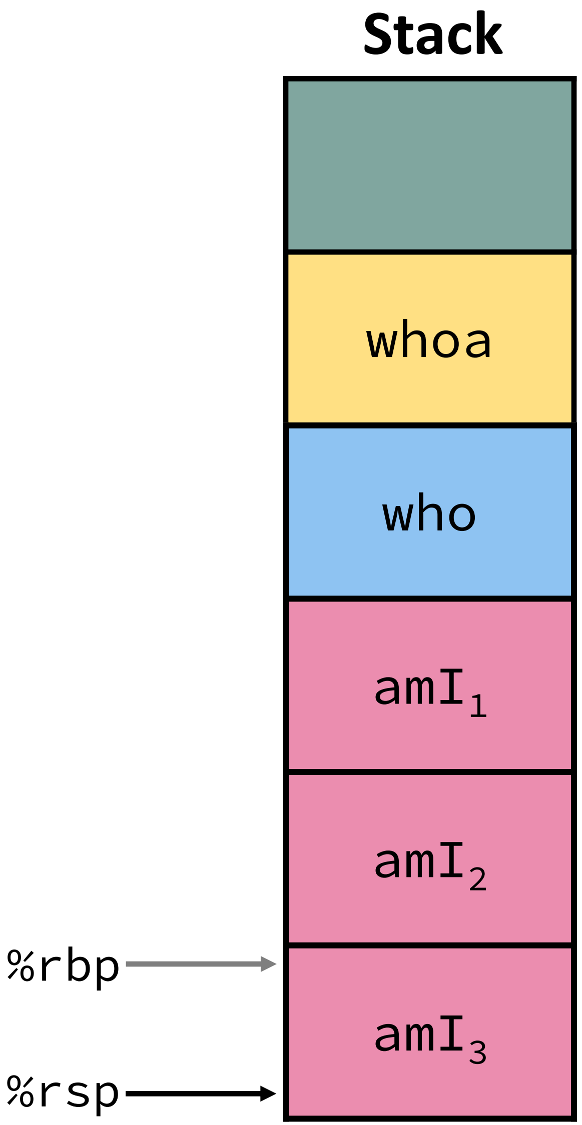




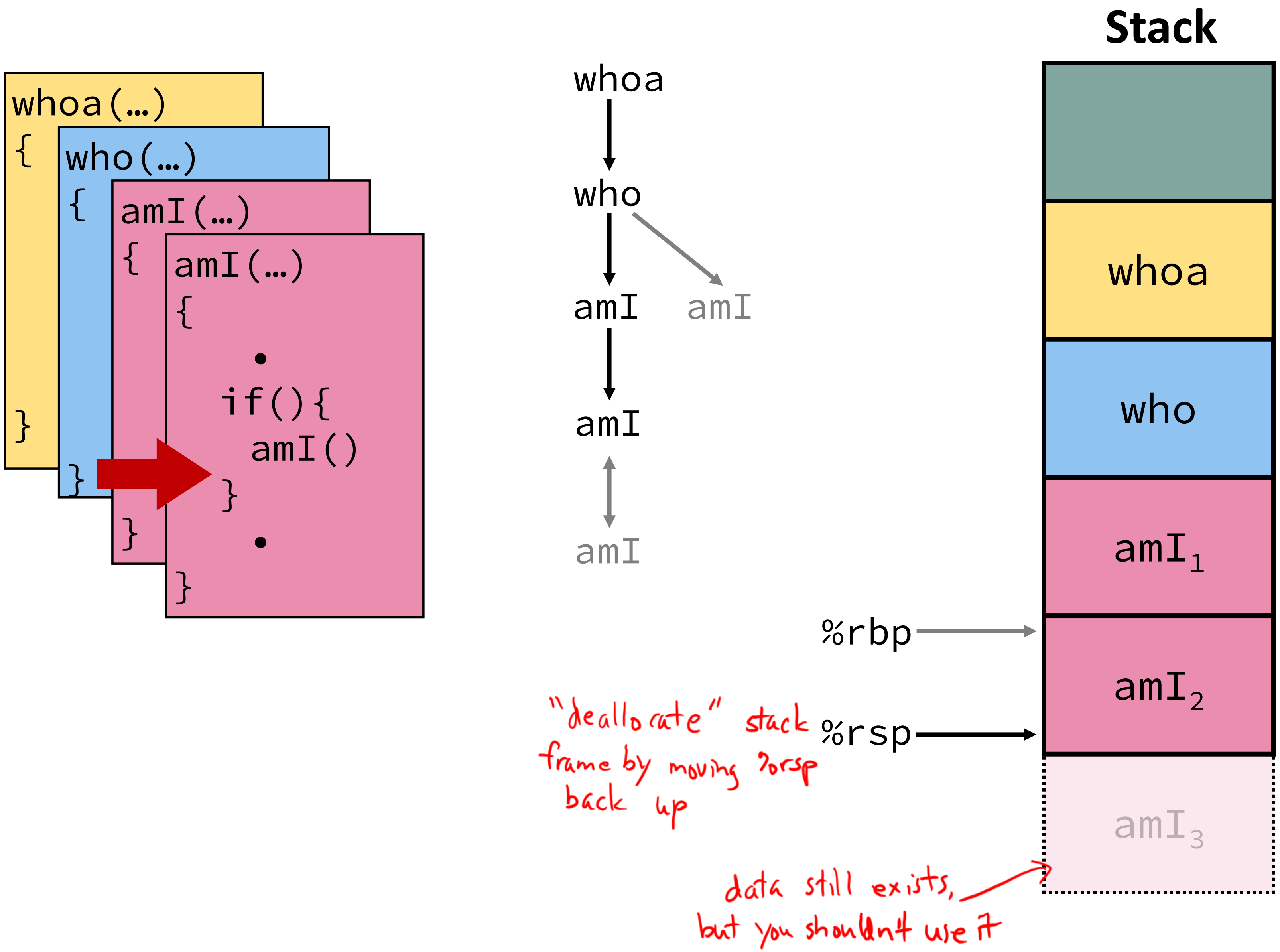
5) Another recursive call to `amI` (3^{rd})



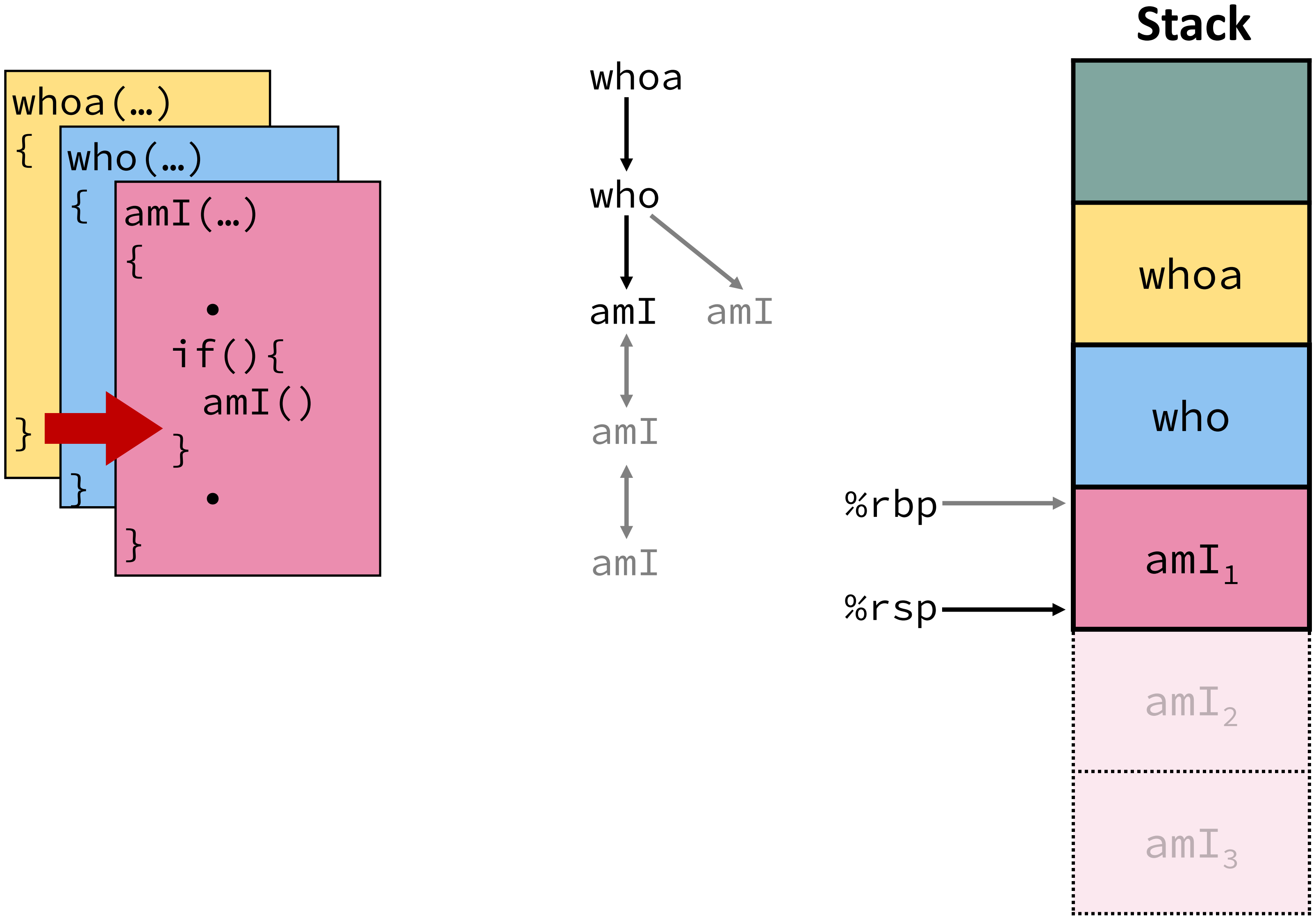




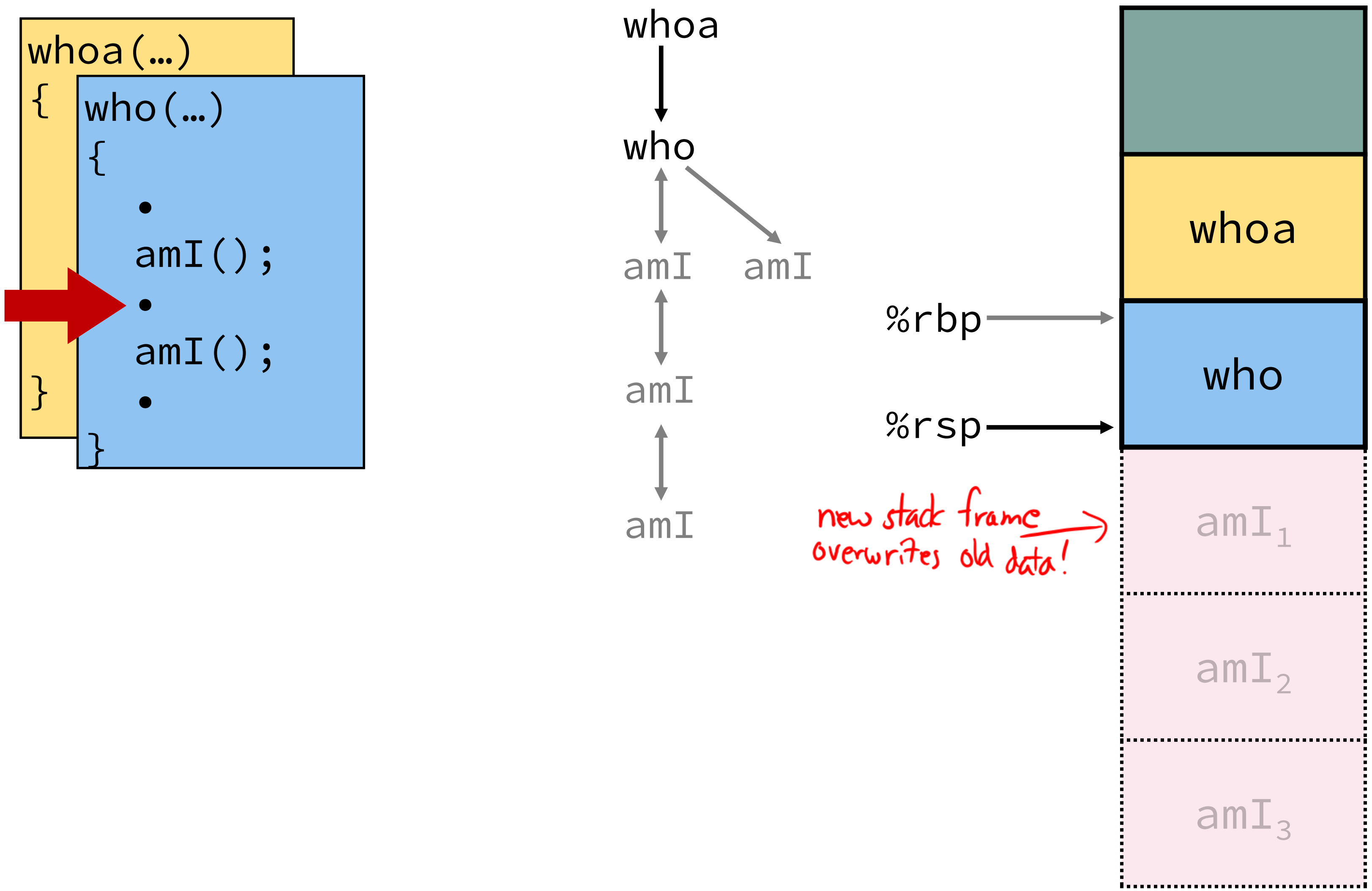
6) Return from another recursive call to `amI` (3rd)



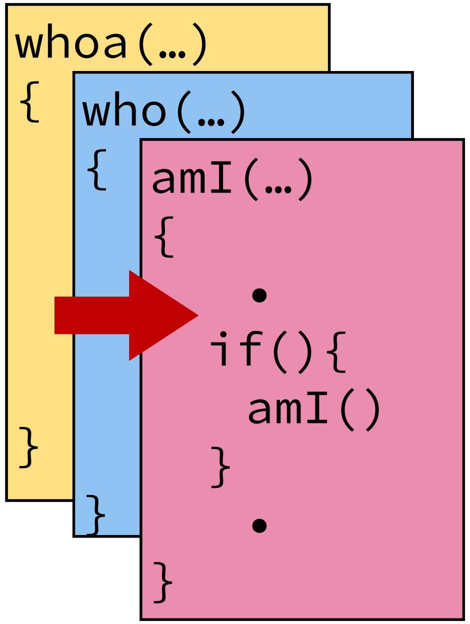
7) Return from recursive call to `amI` (2^{nd})

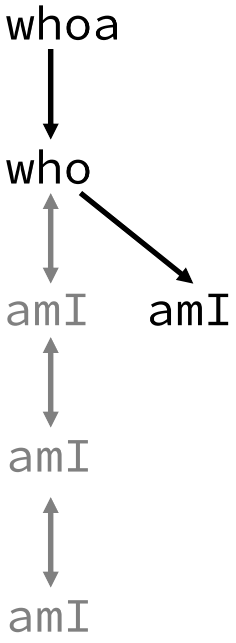


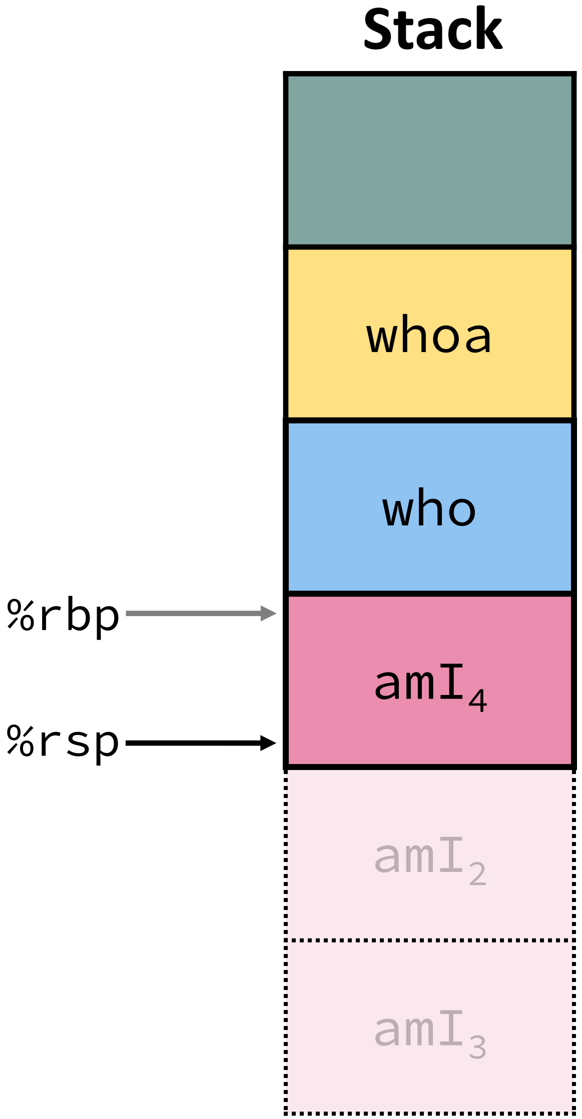
8) Return from call to `amI` (1^{st})



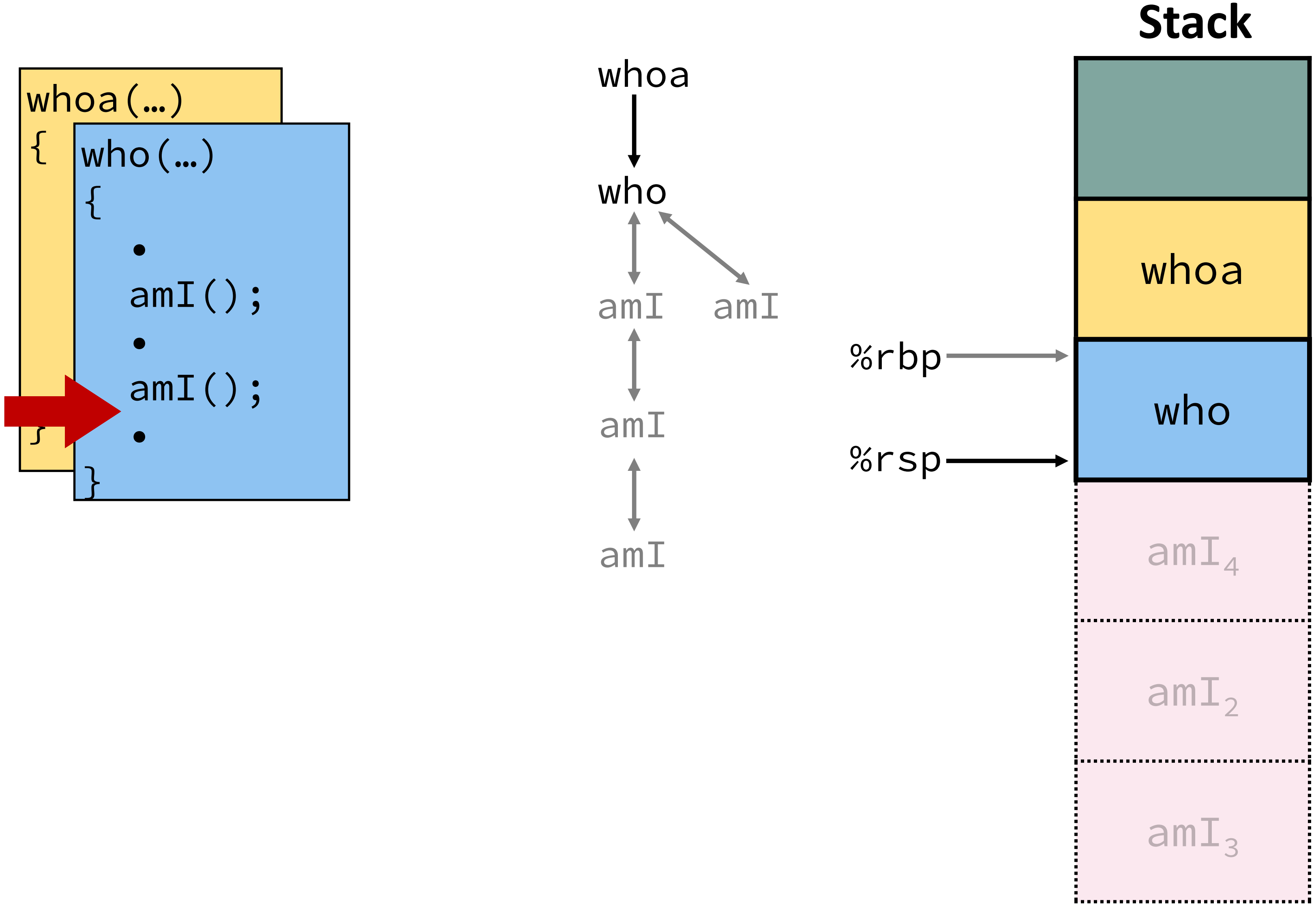
9) Second call to `amI` (4th)





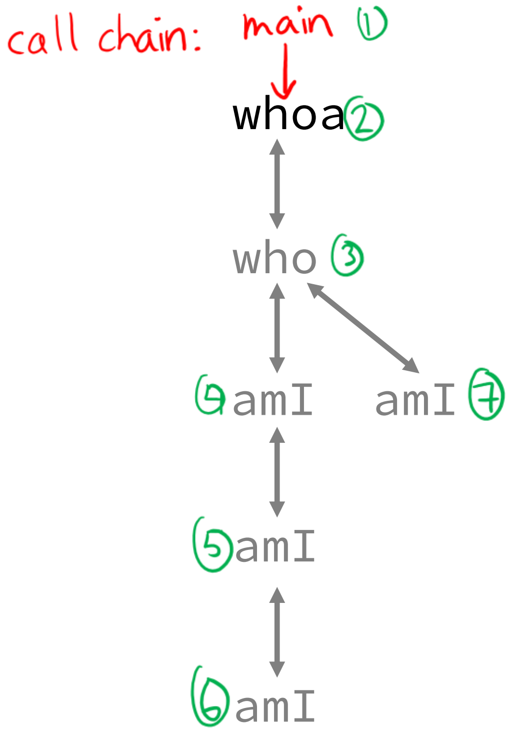


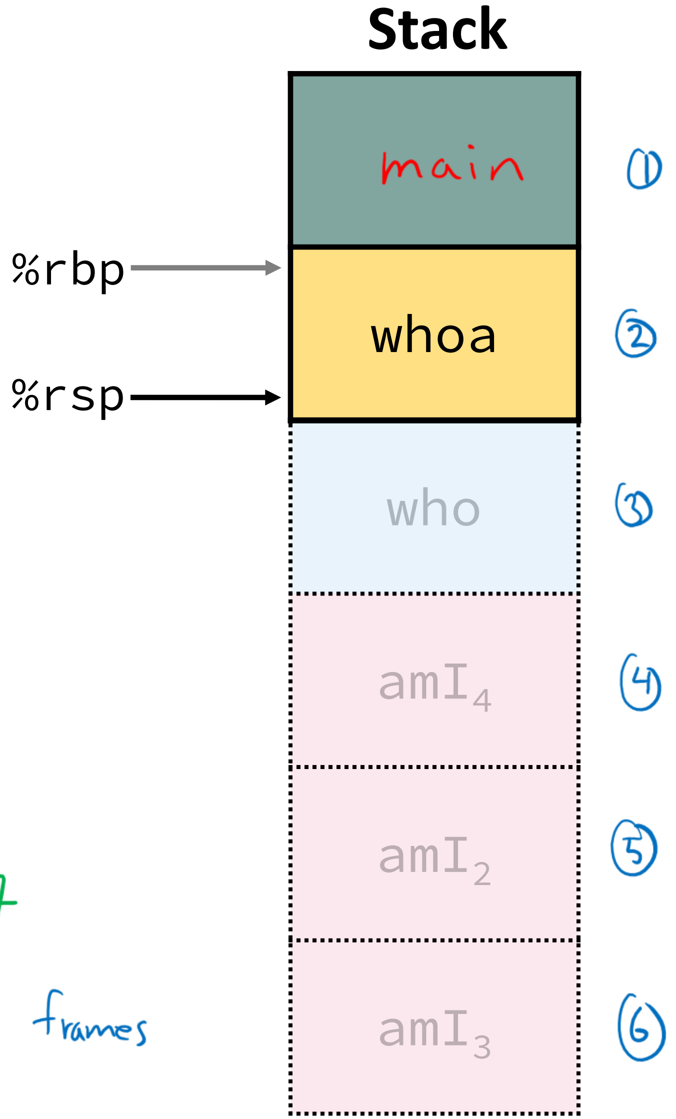
10) Return from second call to `amI` (4^{th})



11) Return from call to who

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





total stack frames created: 7

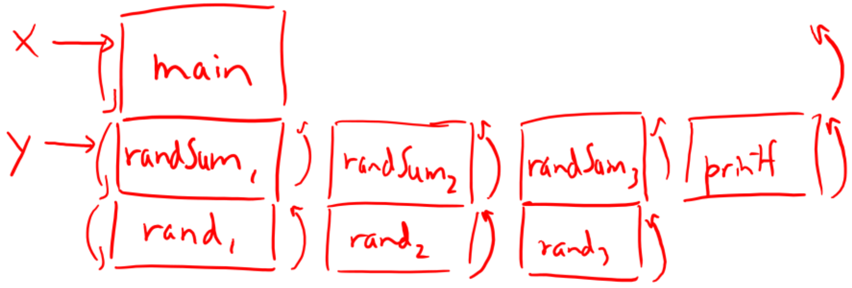
maximum stack depth: 6 frames

Polling Questions (3/3)

- Answer the following questions about when `main()` is run (assume x and y stored on the Stack):

```c
int main() {
    int i, x = 0;
    for(i=0;i<3;i++)
        x = randSum(x);
    printf("x = %d\n",x);
    return 0;
}
```

```c
int randSum(int n) {
    int y = rand()%20;
    return n+y;
}
```



  - *Higher/larger address*: x or y?
  - How many total stack frames are *created*? 8
  - What is the maximum *depth* (# of frames) of the Stack?

A. 1 B. 2 C. 3 D. 4

Aside: Stack Overflow

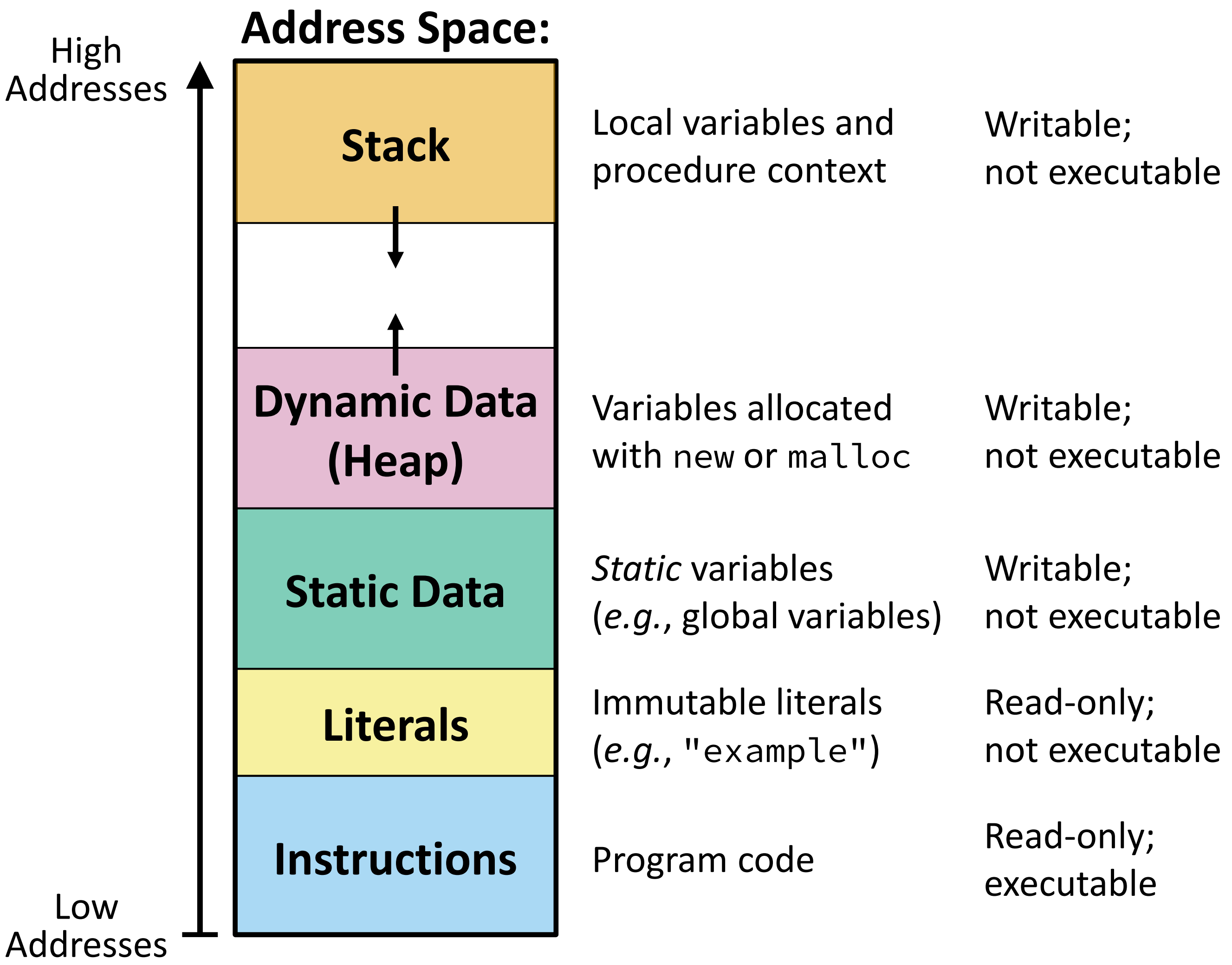
- When the stack pointer exceeds the stack bounds (segmentation fault)
  - In theory: when it collides with the Heap
  - In x86-64 Linux, when it exceeds 8 MiB limit

- Causes?
  - Infinite/deep recursion
  - Very large local variables

- Fixes?
  - Use iterative solution, compiler tail-call optimization
  - Allocate large variables elsewhere (more on the Heap later this quarter)

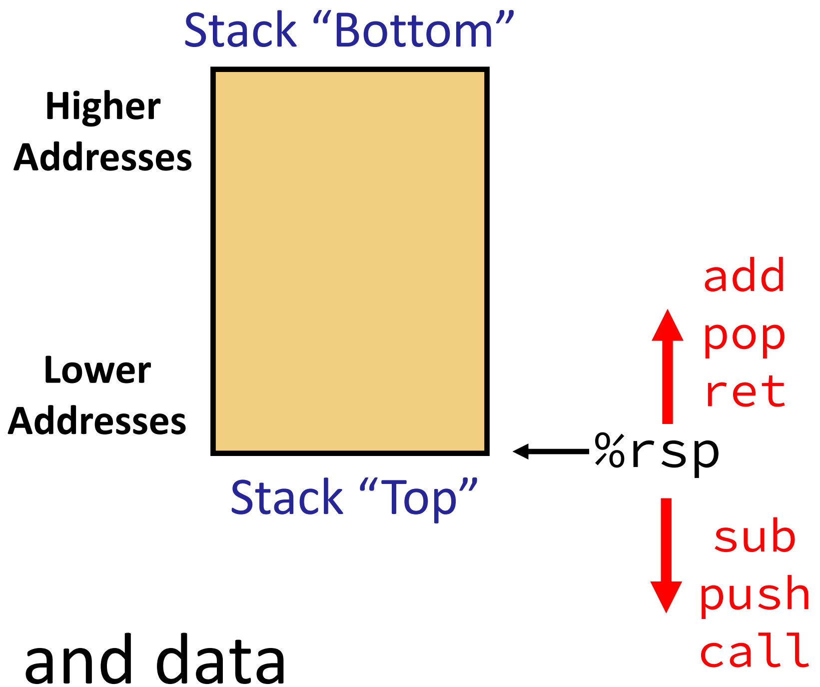
Summary (1/3)

- Memory is organized into 5 segments based on data declaration and lifetime
  - Goals: maximize use of space, manage data differently, apply separate permissions
- A **segmentation fault** is caused by an impermissible memory access



Summary (2/3)

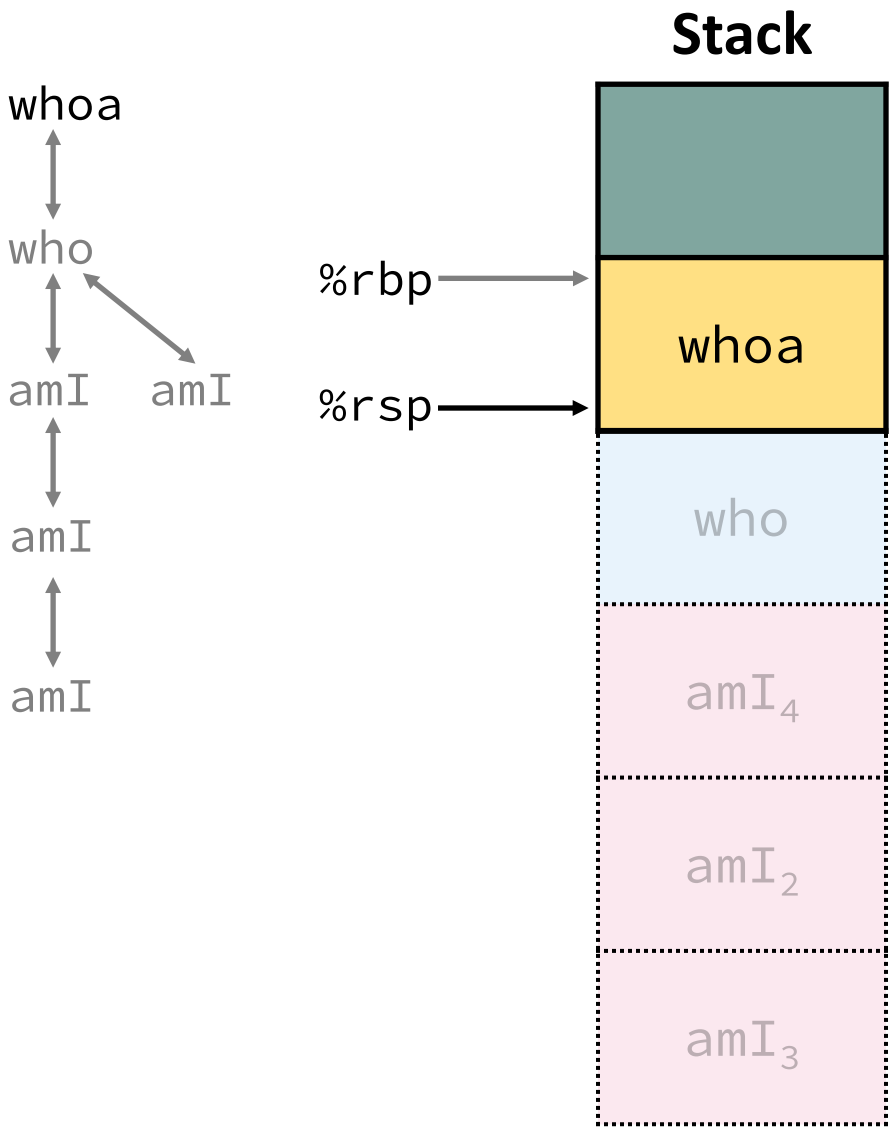
- **The Stack** is the memory segment with the highest addresses and grows downward
  - Stack "top" (lowest address) is defined by the value of the stack pointer (`%rsp`)
  - Can manipulate using `add`, `sub`, `push`, and `pop`



- Procedure calling conventions for passing control and data
  - `call` and `ret` pass control using `%rip` and a return address on the stack
  - Return value: `%rax`, Arguments: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, Stack

Summary (3/3)

- Stack organized into **stack frames** that hold a procedure instance's data
  - Size will vary based on procedure specifics
  - Space gets allocated as procedure executes, deallocated by the time it returns



Bonus: x86-64 Linux Memory Layout

This is extra (non-testable) material

- Stack
  - Runtime stack has 8 MiB limit
- Heap
  - Dynamically allocated as needed
  - `malloc()`, `calloc()`, `new`, …
- Statically allocated data (Data)
  - Read-only: string literals
  - Read/write: global arrays and variables
- Code / Shared Libraries
  - Executable machine instructions
  - Read-only