The Stack & Procedures

CSE 351 Summer 2021

Instructor:

Justin Hsia

~~Mara Kirdani-Ryan~~

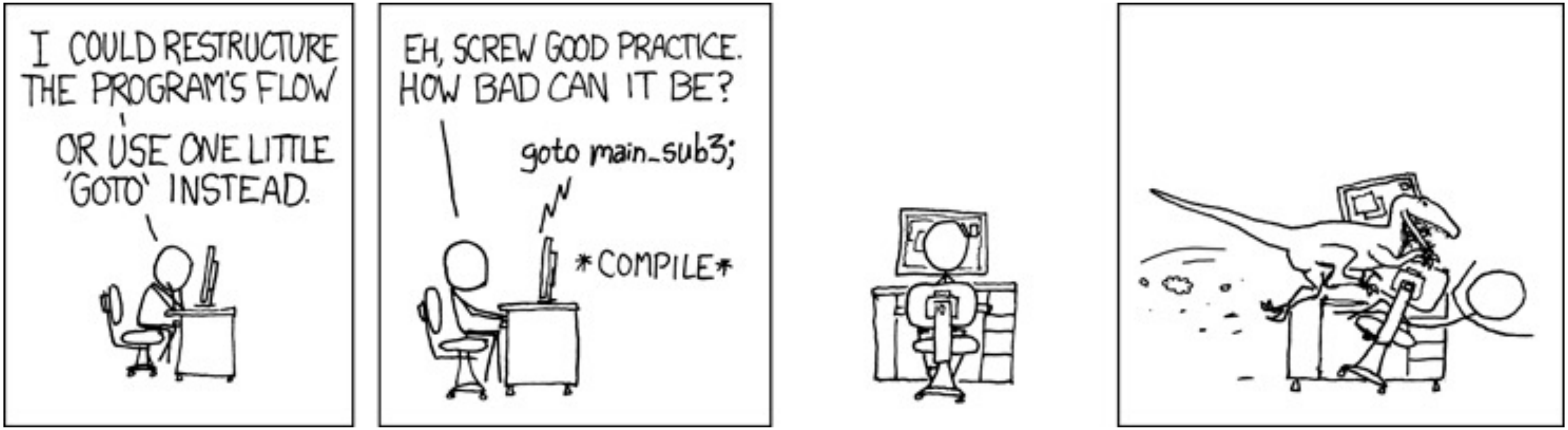
Teaching Assistants:

Kashish Aggarwal

Nick Durand

Colton Jobes

Tim Mandzyuk



http://xkcd.com/571/

Gentle, Loving Reminders

- Mid-quarter Survey due Friday (7/16) – 8 pm
  - Submit via Canvas!
- hw9 due tonight, hw10 due Friday
- Lab 2 due Wednesday (7/21)
  - GDB Tutorial on website walks through first phase
  - *Mara's favorite lab!*

As a note

- No midterm or final! But the midterm and final cheat sheets could be useful for the course
  - Can find them at the exams page of the course website
  - https://courses.cs.washington.edu/courses/cse351/21su/exams/

Learning Objectives

- Understanding this lecture means you can…
  - Draw stack diagrams, and track stack usage throughout function calls
  - Trace stack usage through recursive calls, with stack frames for each invocation
  - Trace stack frames and registers through a procedure call

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

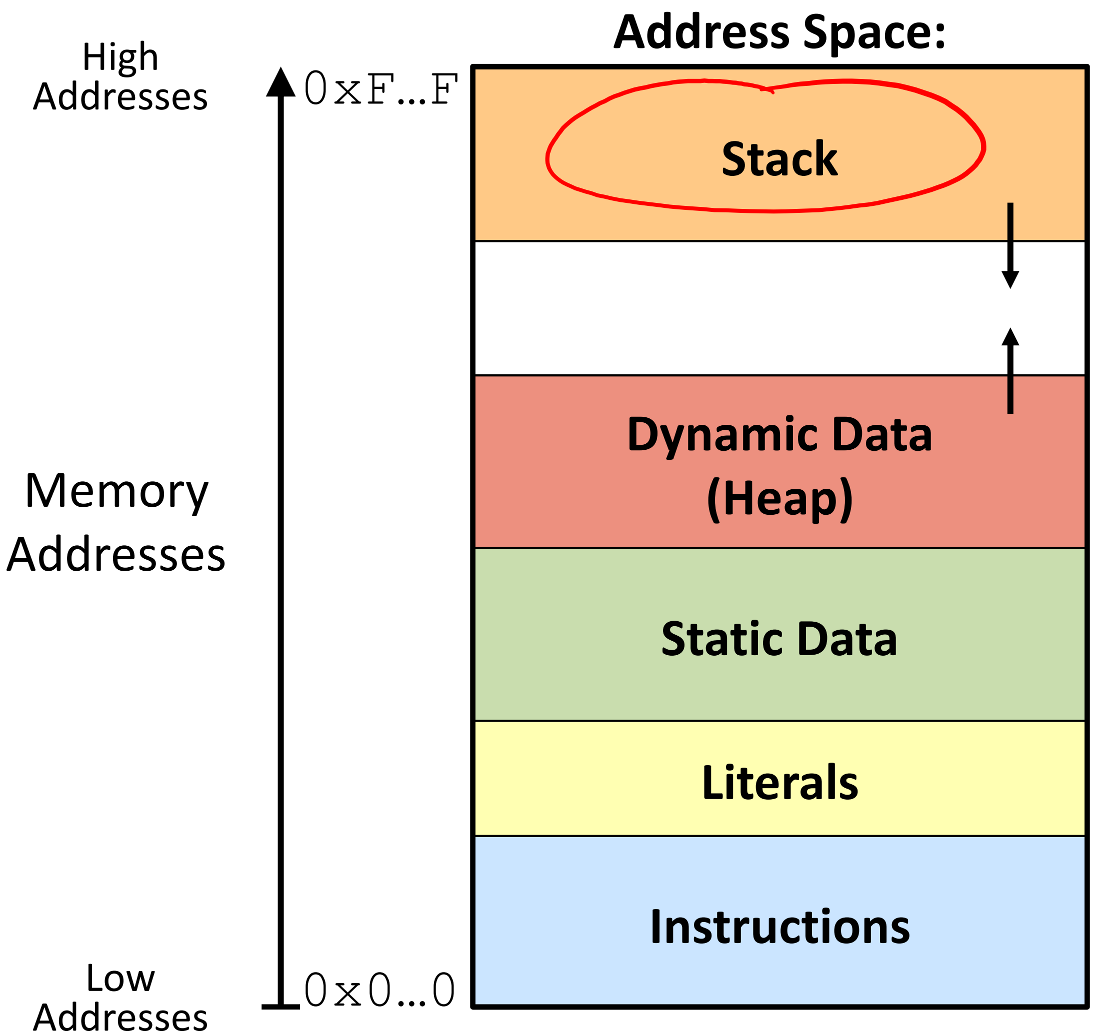
```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

Procedures

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

Simplified Memory Layout



What Goes Here:

local variables and procedure context

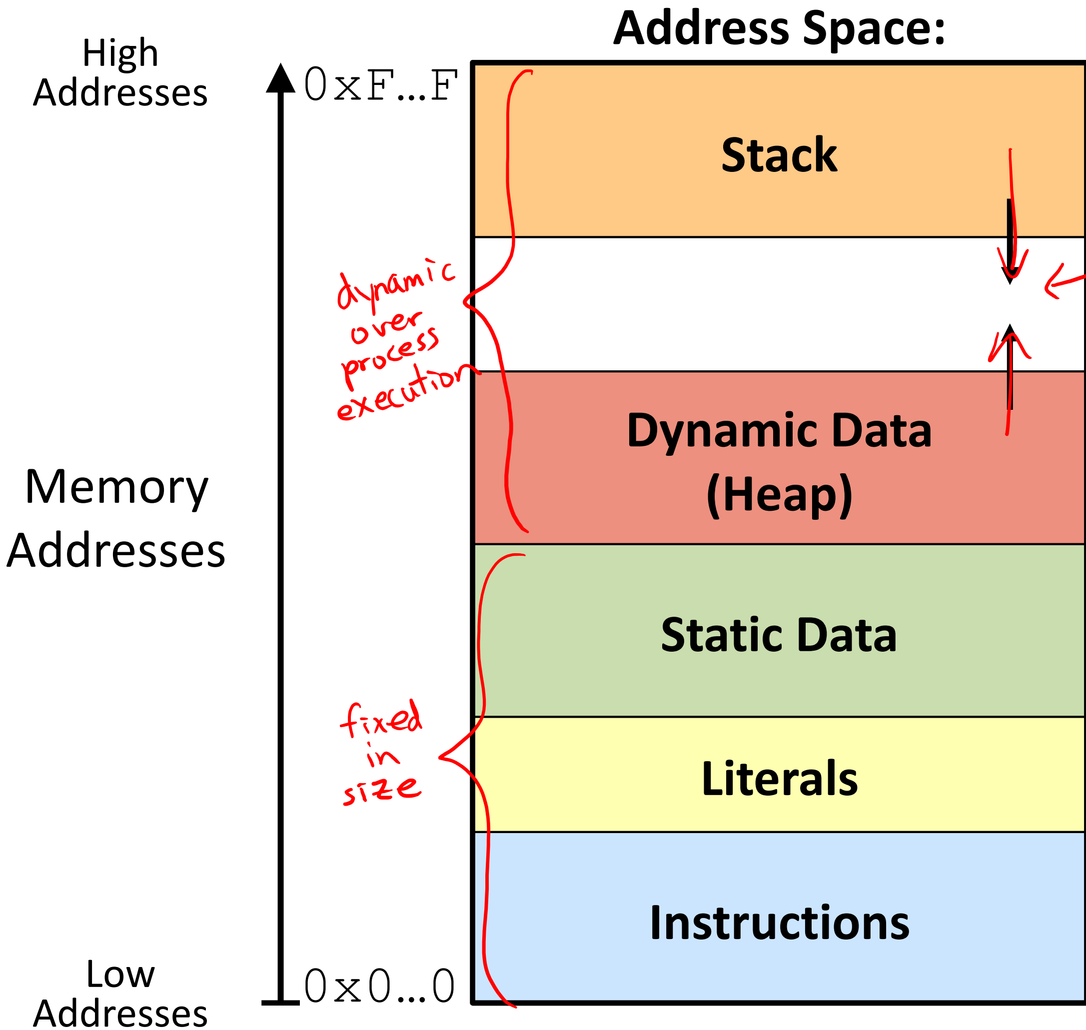
variables allocated with new or `malloc`

static variables (including global variables)

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

program code

Memory Management



Who's Responsible:

Managed "automatically" (by compiler/assembly)

grow towards each other to maximize use of space

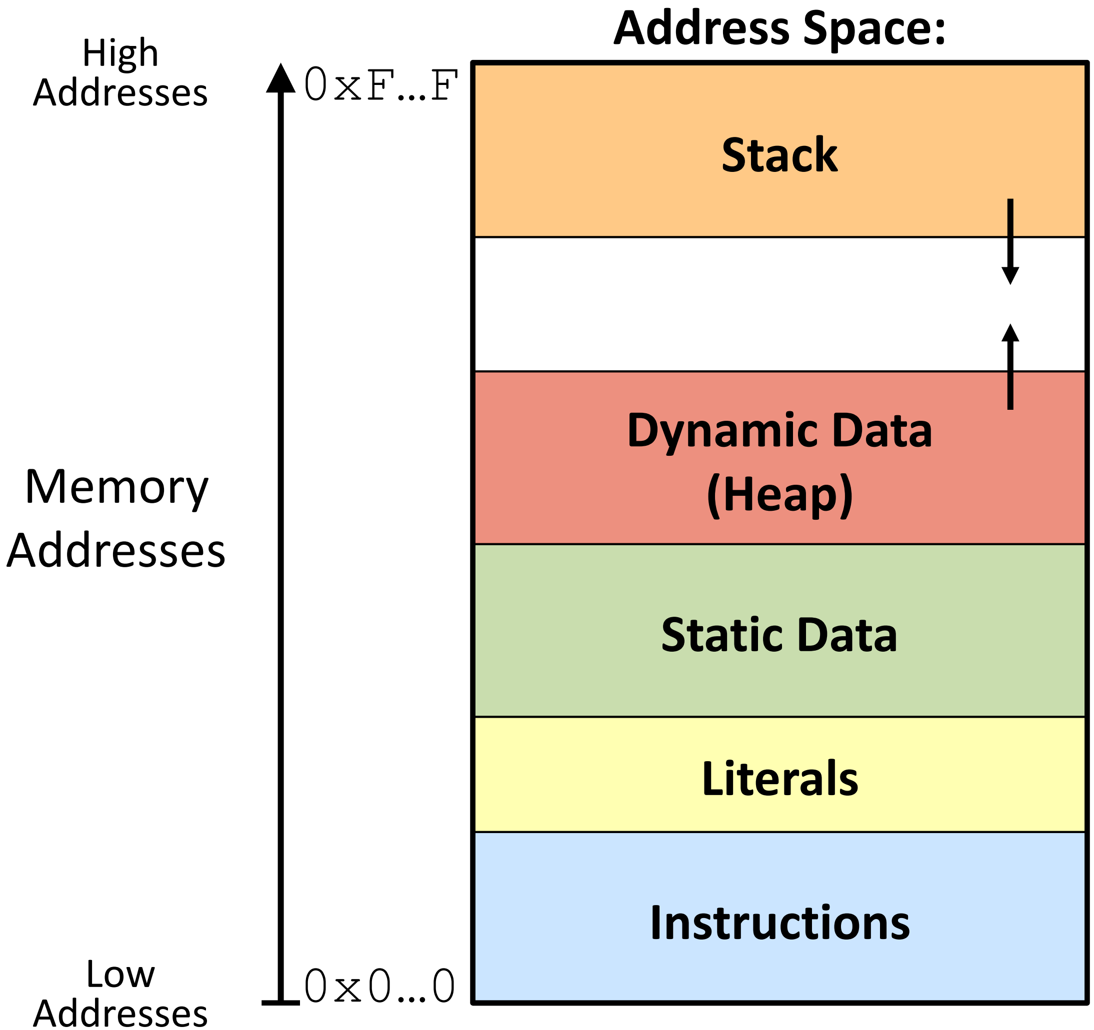
Managed "dynamically" (by programmer)

Managed "statically" (initialized when process starts)

Managed "statically" (initialized when process starts)

Managed "statically" (initialized when process starts)

Memory Permissions



Permissions:

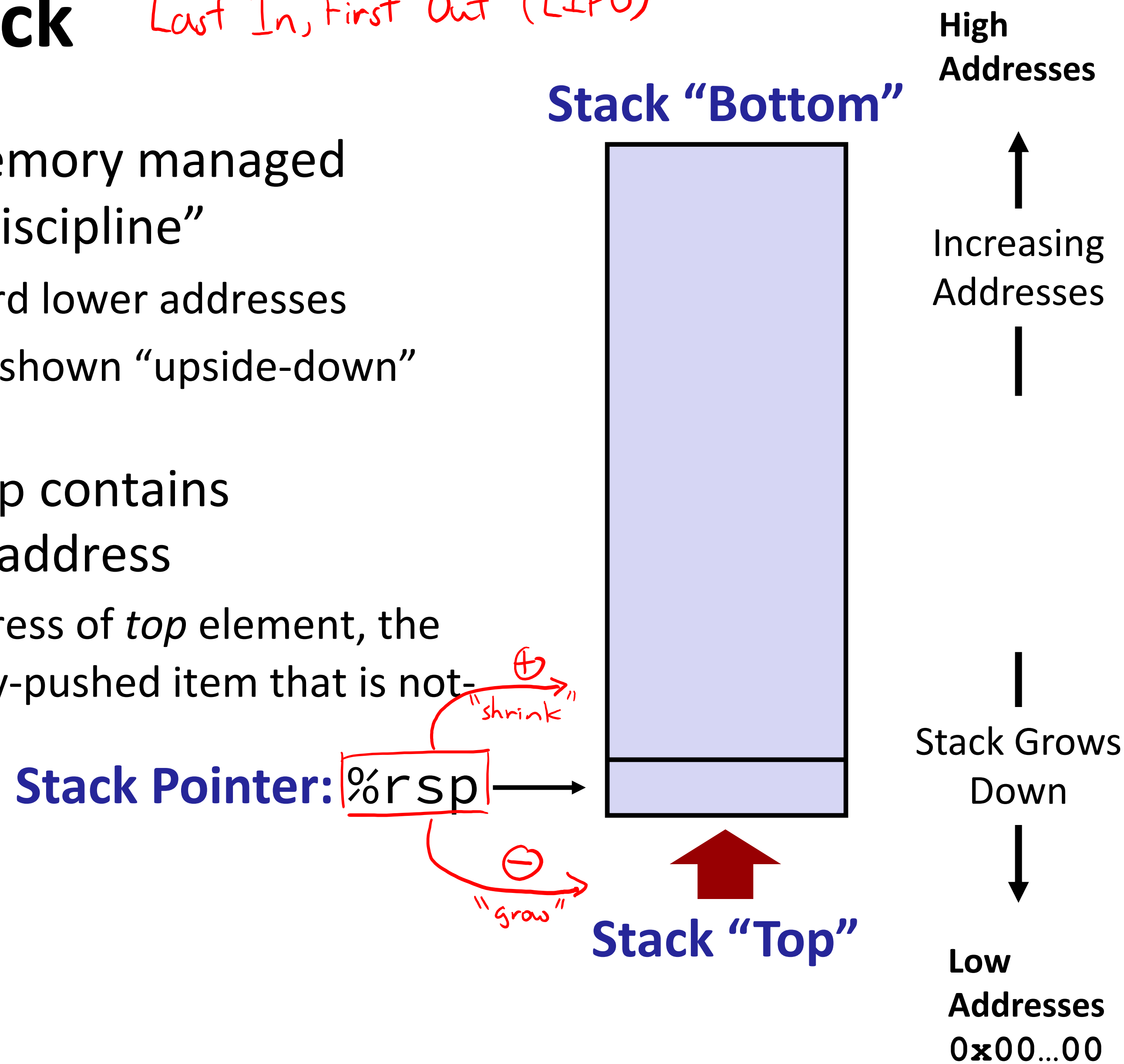
Segment	Permissions
Stack	writable; not executable
Dynamic Data (Heap)	writable; not executable
Static Data	writable; not executable
Literals	read-only; not executable
Instructions	read-only; executable

- Segmentation fault: impermissible memory access

x86-64 Stack

Last In, First Out (LIFO)

- 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

- pushq *src*
  size specifier
  - Fetch operand at src
    - Src can be reg, memory, immediate
  - ***Decrement*** `%rsp` by 8
  - Store value at address given by `%rsp`
- <u>Example</u>:
  - **`pushq %rcx`**
  - Adjust `%rsp` and store contents of `%rcx` on the stack

Registers

%rcx

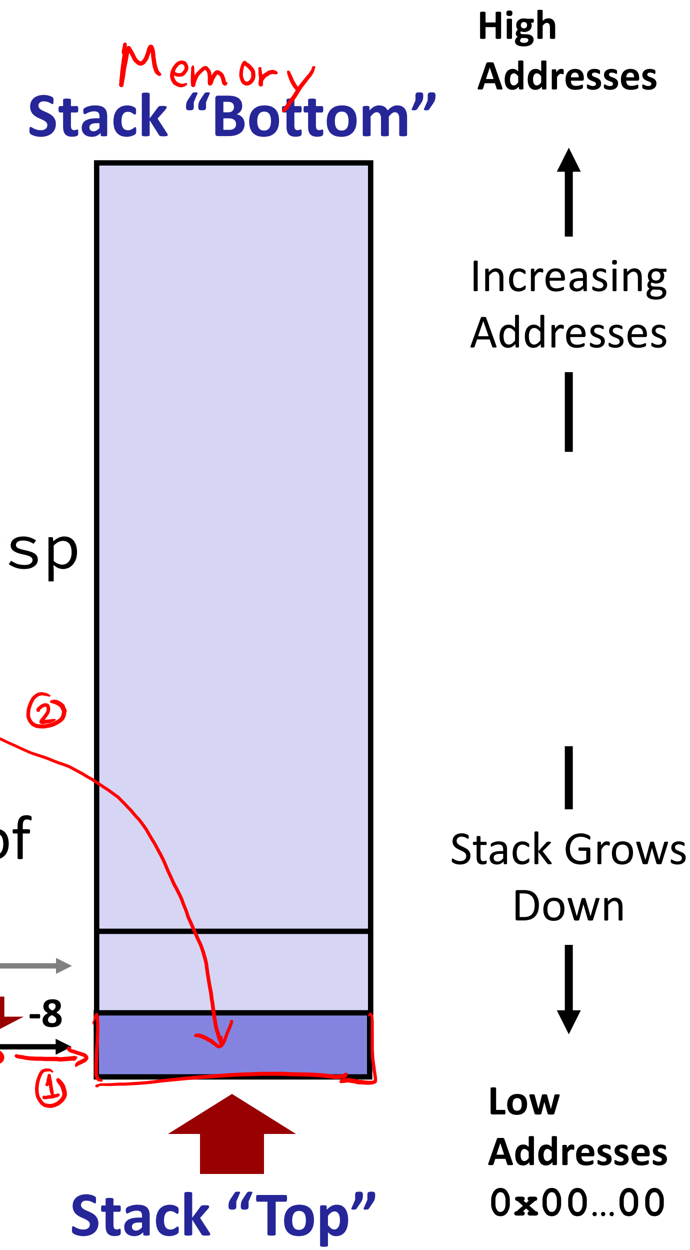
②

Stack Pointer: `%rsp` -8

new %rsp ①

① move %rsp down (subtract)

② store src at %rsp



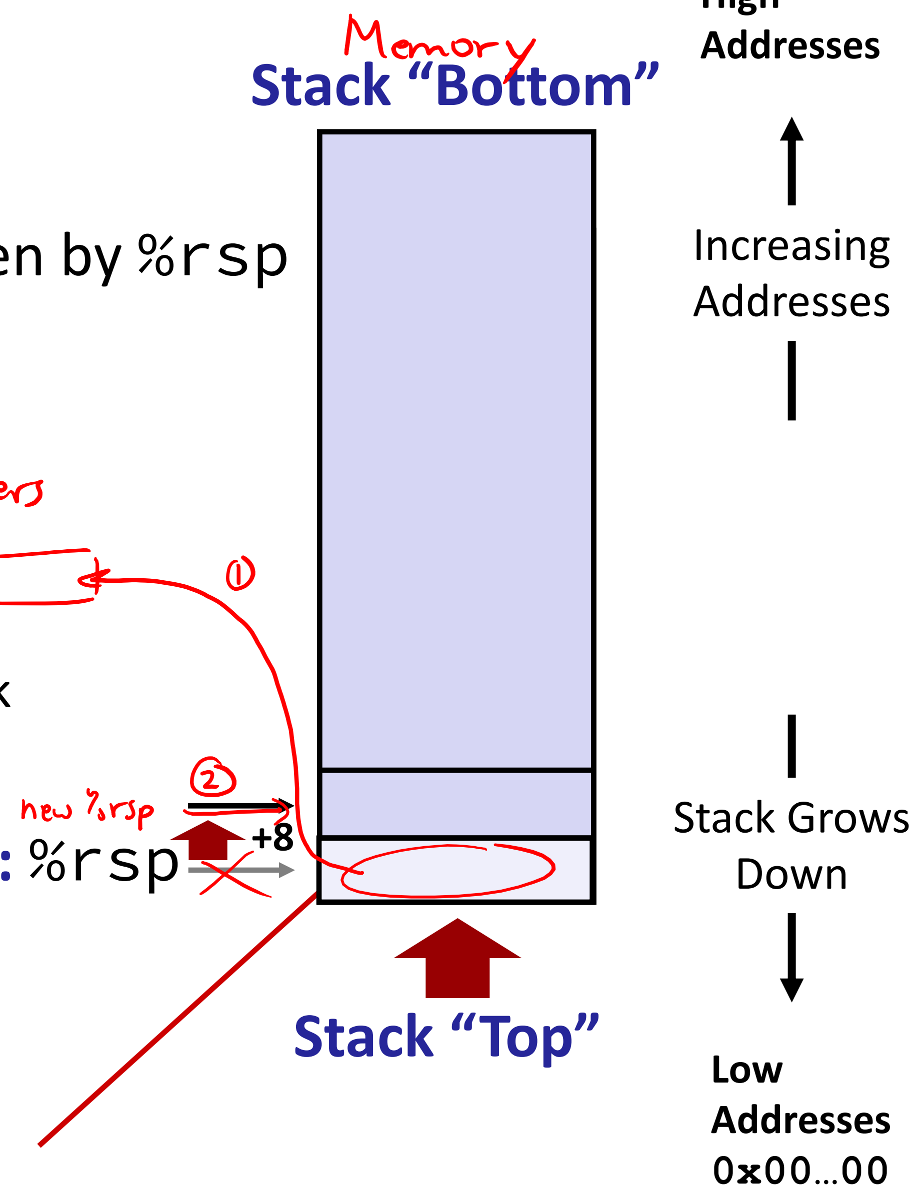
x86-64 Stack: Pop

- popq *dst*
  size specifier
  - 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`

Registers

%rcx

①



① read out data at %rsp
② move %rsp up (addition)

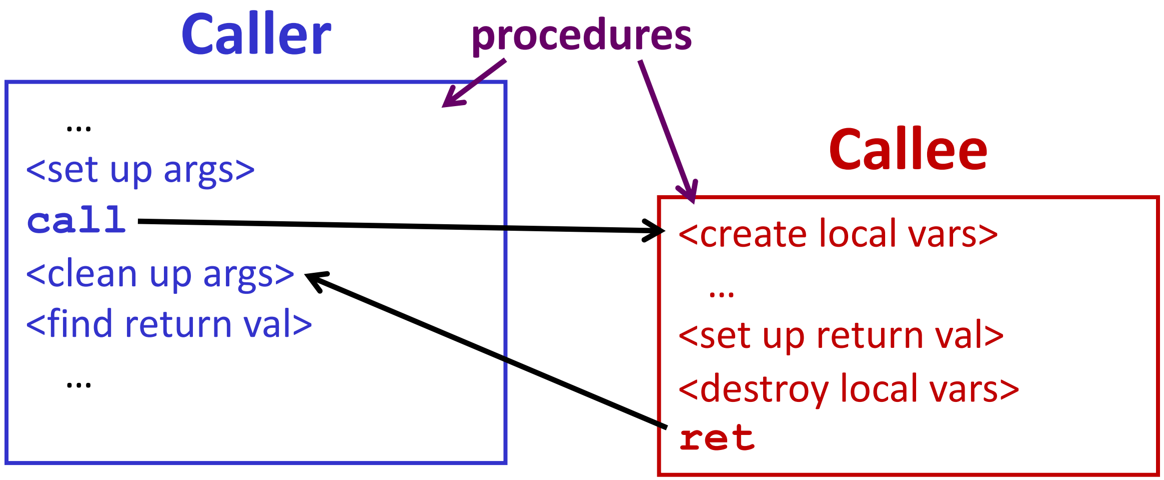
Those bits are still there; we're just not using them.

How are y’all feeling about stack structure, push, pop?

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

Caller

…
<save regs>
<set up args>
`call`
<clean up args>
<restore regs>
<find return val>
…

Callee

<save regs>
<create local vars>
…
<set up return val>
<destroy local vars>
<restore regs>
`ret`

- 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;
}
```

Caller

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
```

these are instruction addresses

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
```

Procedure Control Flow

- 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

- 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**

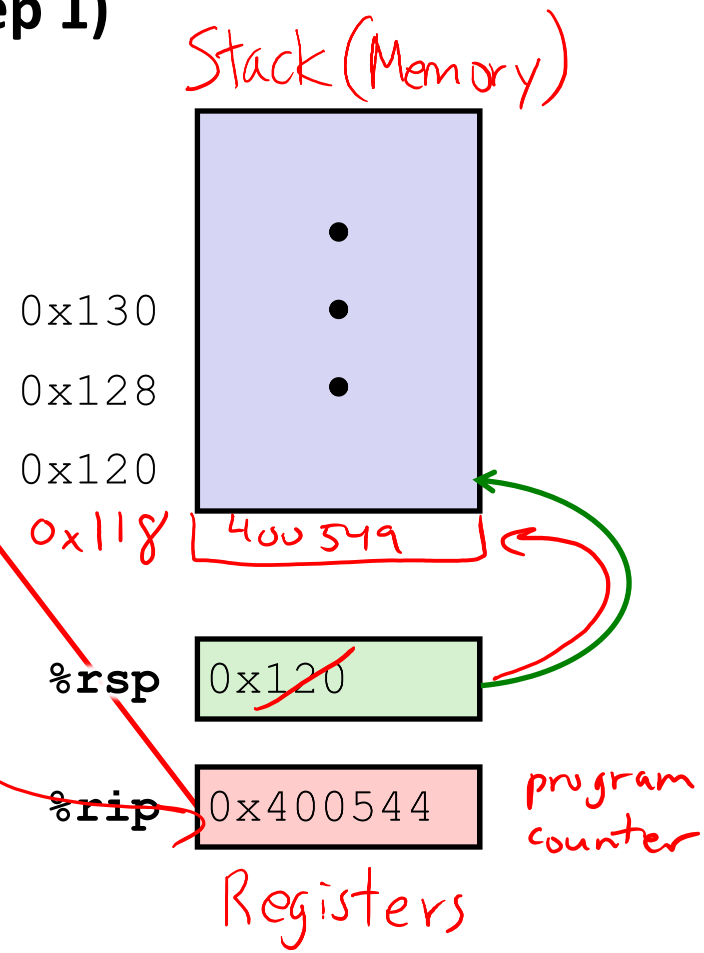
next instruction happens to be a move, but could be anything

- 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

Procedure Call Example (step 1)

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

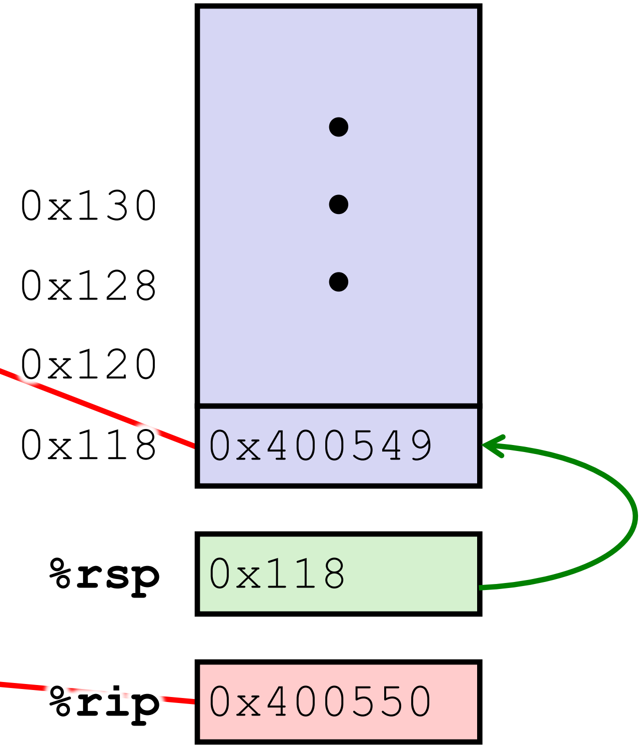
```
0000000000400550 <mult2>:
   400550:   movq    %rdi,%rax
   •
   •
   400557:   ret
```



Procedure Call Example (step 2)

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

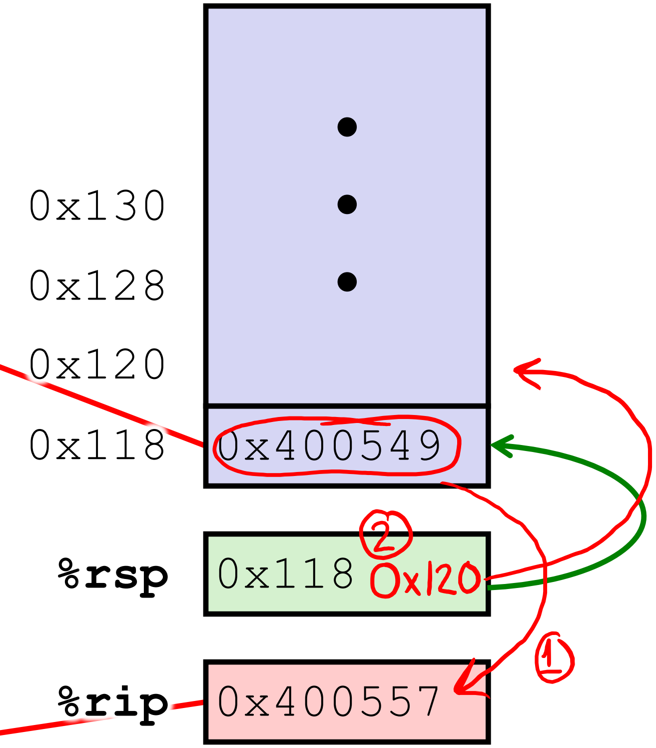
```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```



Procedure Return Example (step 1)

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

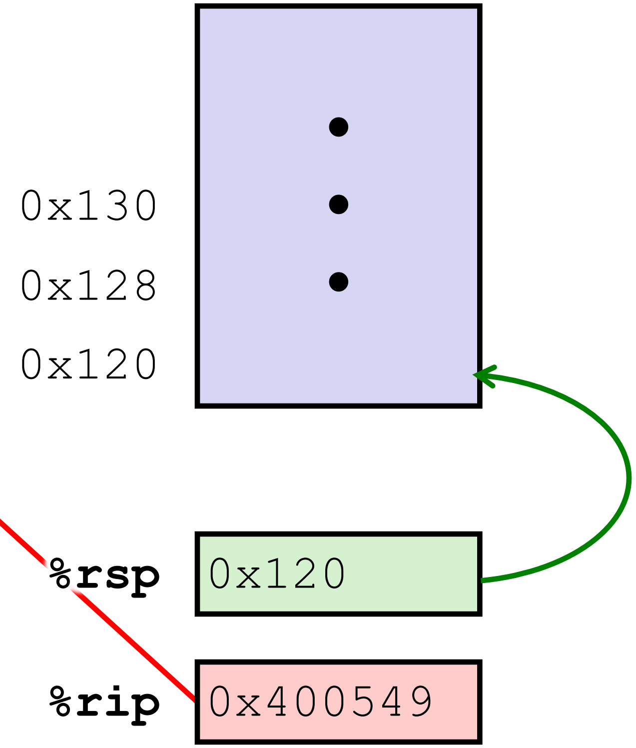
```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```



Procedure Return Example (step 2)

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

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



Feelings check: Procedure call/return?

Procedures

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

Procedure Data Flow

Registers (NOT in Memory)

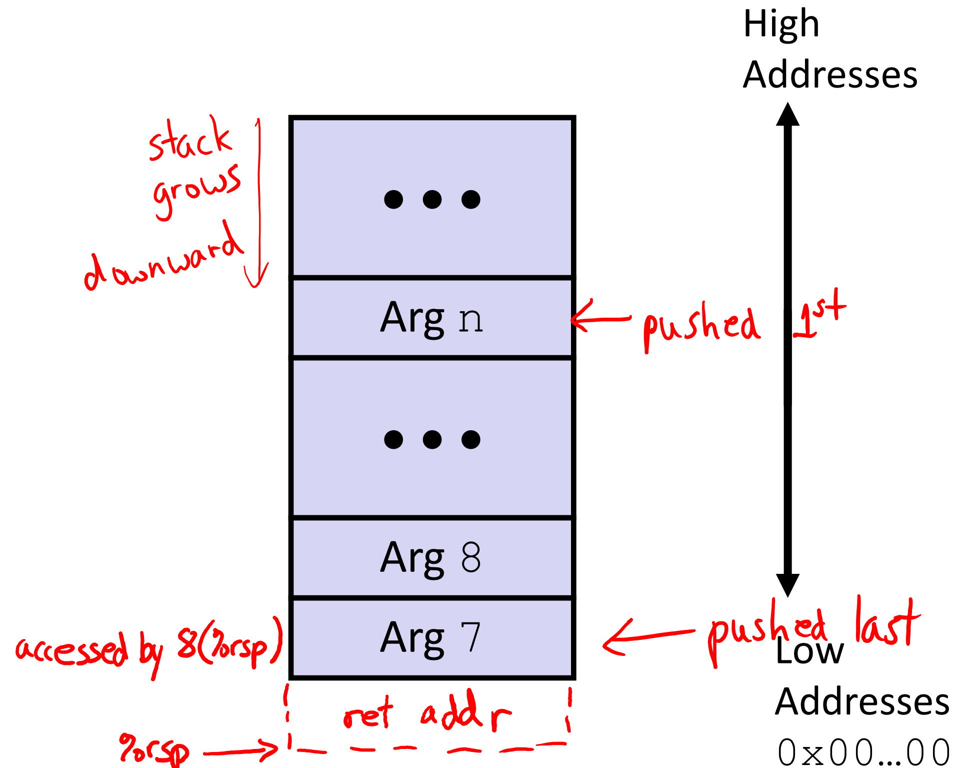
- First 6 arguments

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

- Return value

`%rax`

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;
}
```

rdi rsi rdx

rdi rsi

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

```
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
    • • •
```

(will explain later)

```
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-Based Languages

- 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();
    •
}
```

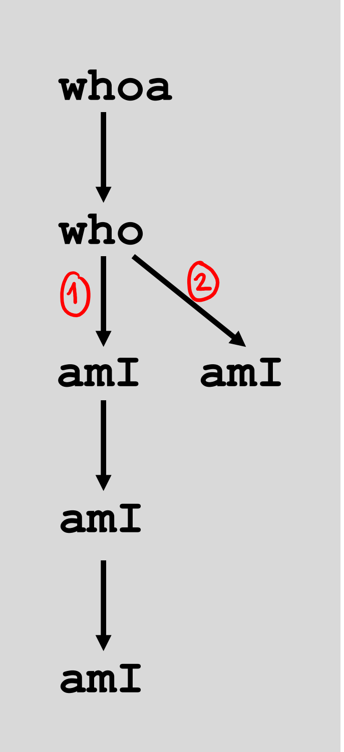
1st call recurses twice

2nd call doesn't recurse

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

based on condition

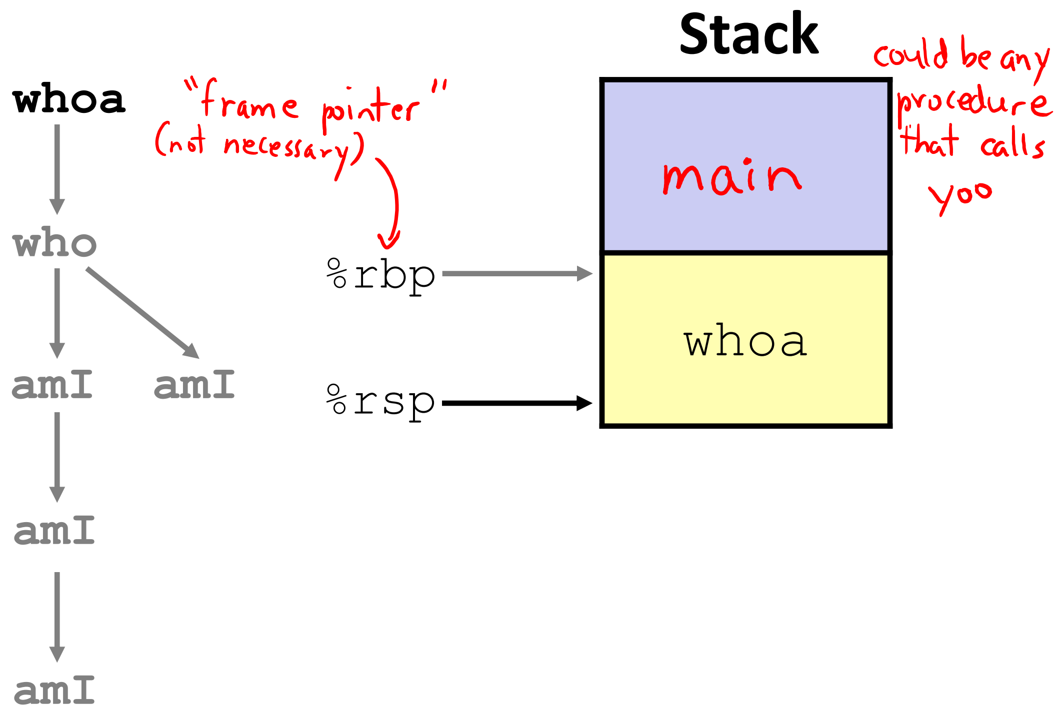
Example
Call Chain



Procedure amI is recursive
(calls itself)

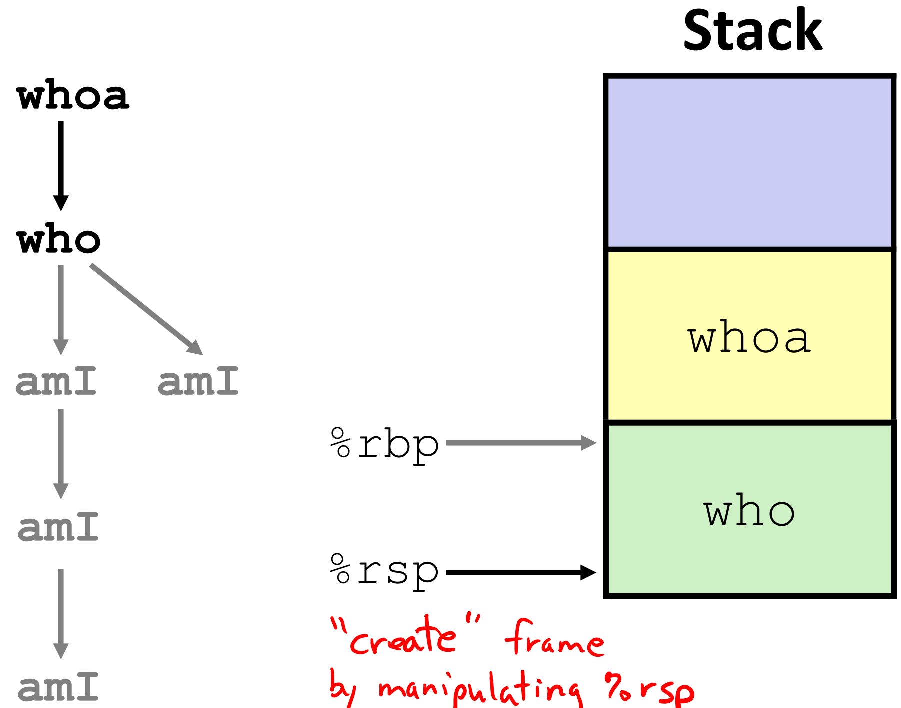
1) Call to whoa

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



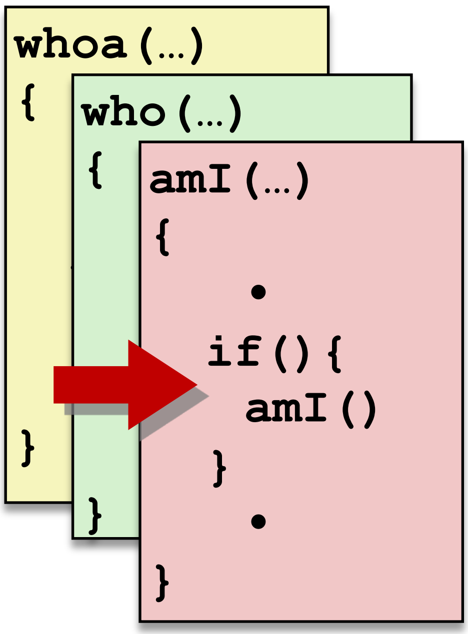
2) Call to who

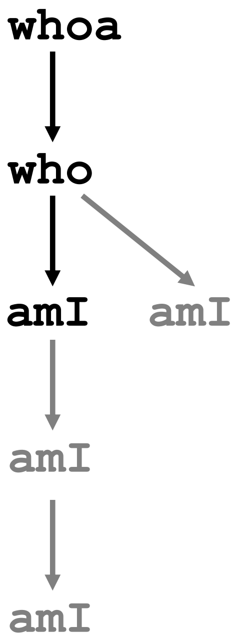
```
whoa(…)
{
    who(…)
    {
      •
      amI();
      •
      amI();
      •
    }
}
```

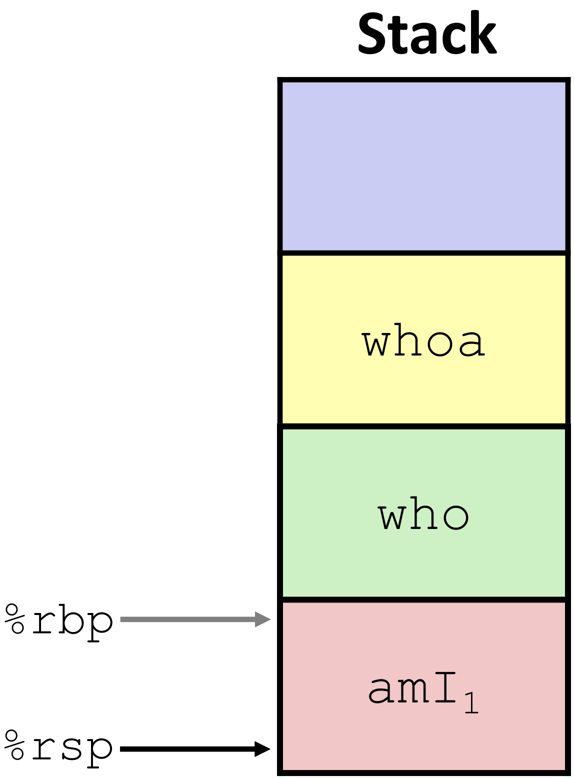


"create" frame by manipulating %rsp

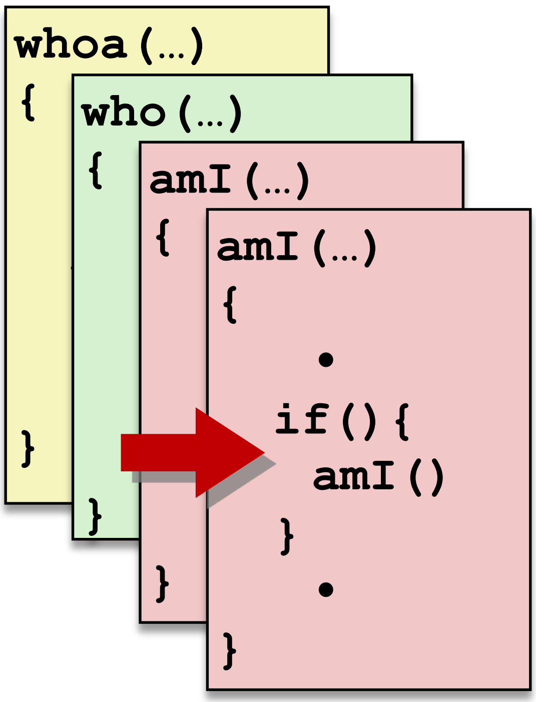
3) Call to amI (1)

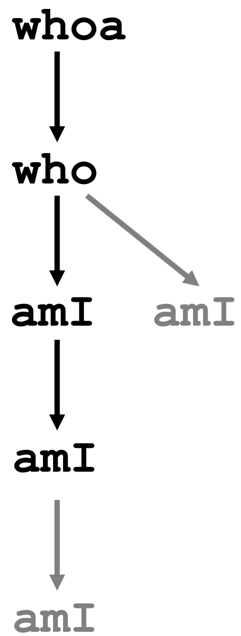


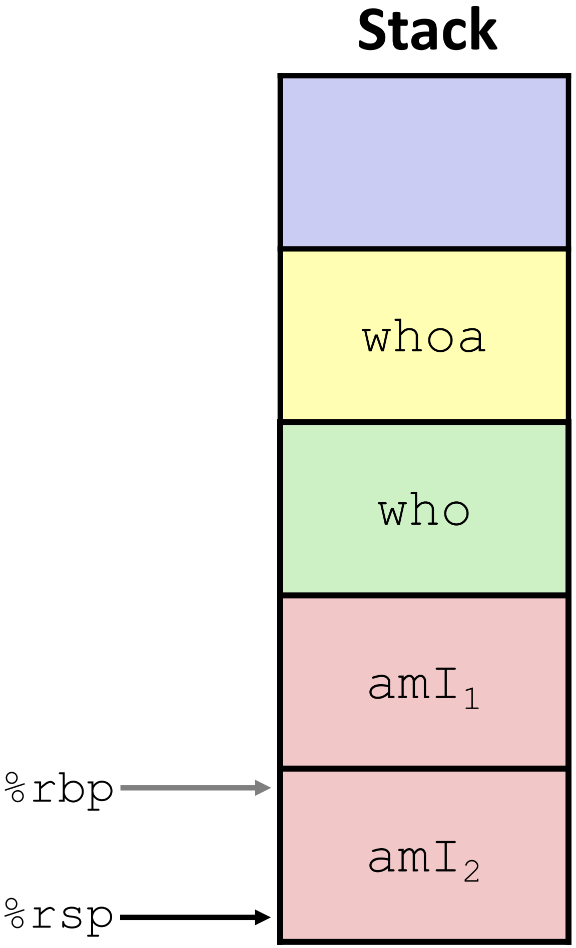




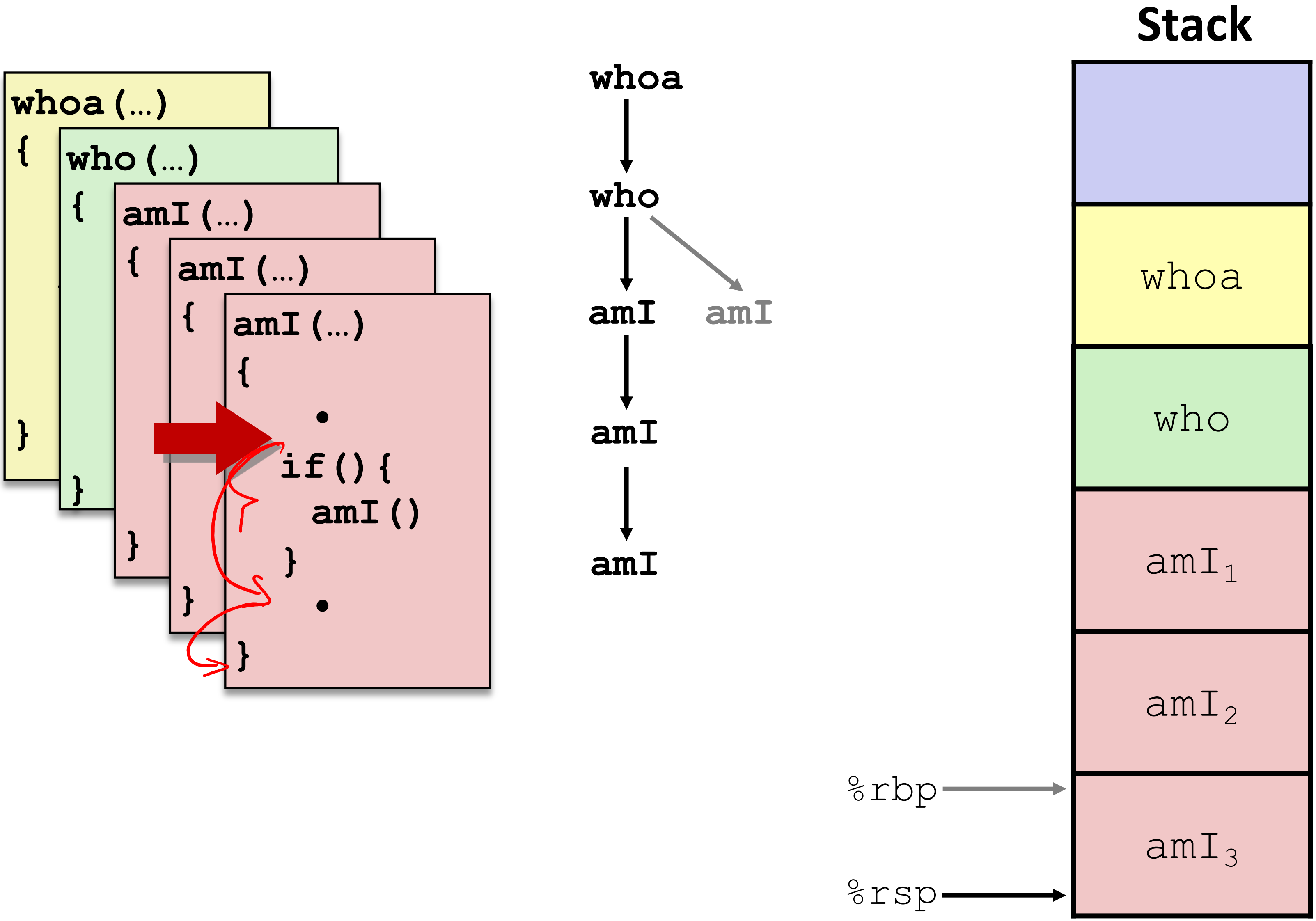
4) Recursive call to `amI` (2)



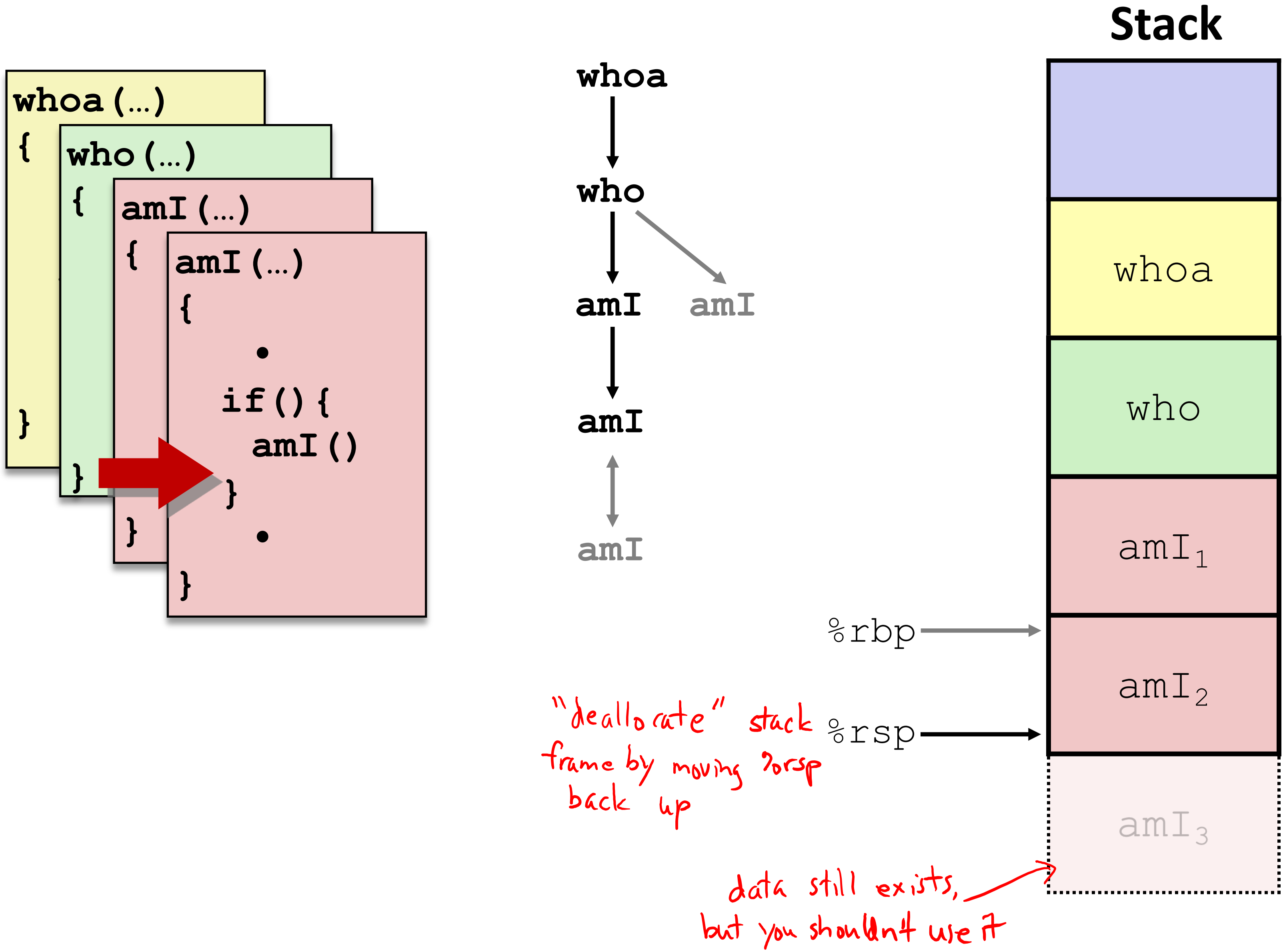




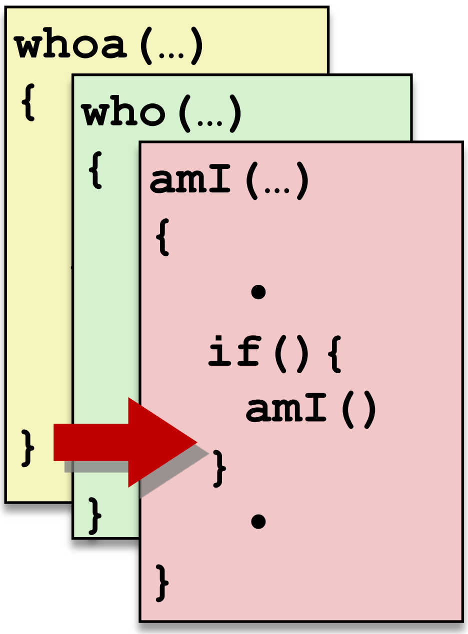
5) (another) Recursive call to `amI` (3)

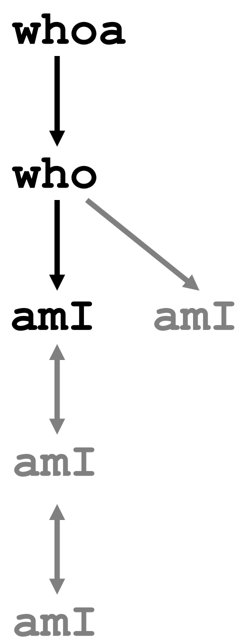


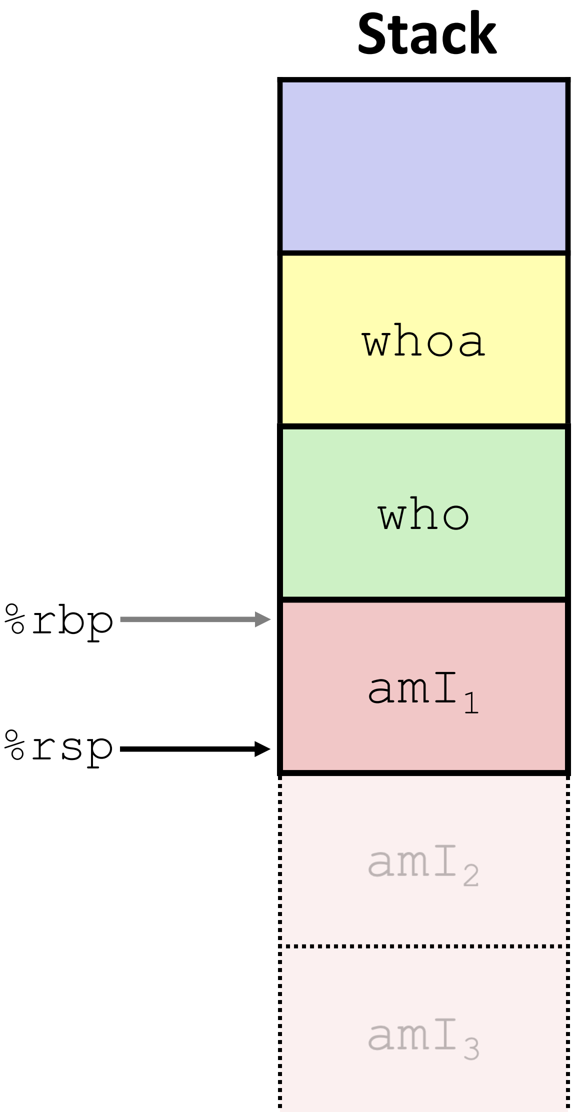
6) Return from (another) recursive call to `amI`



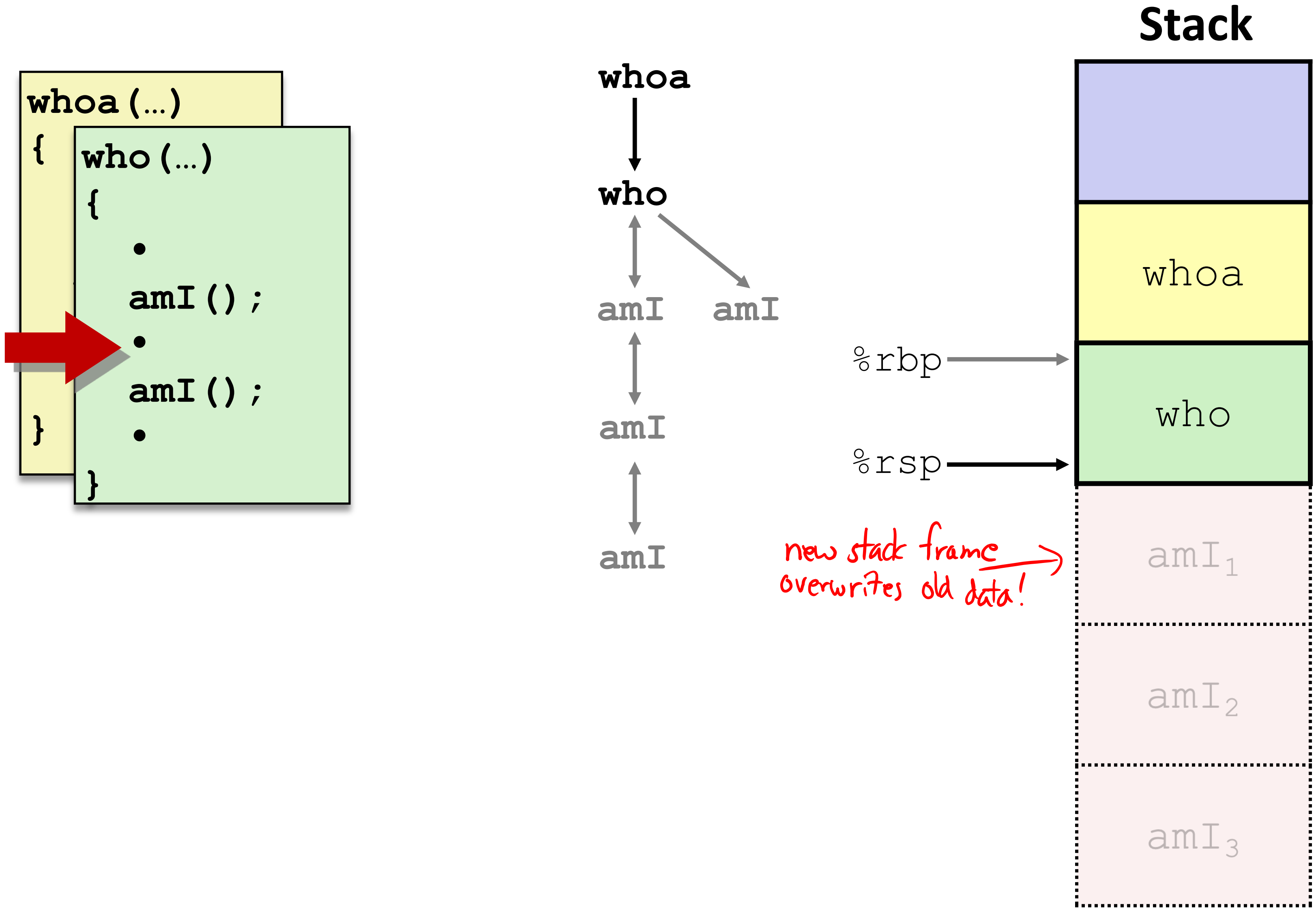
7) Return from recursive call to amI



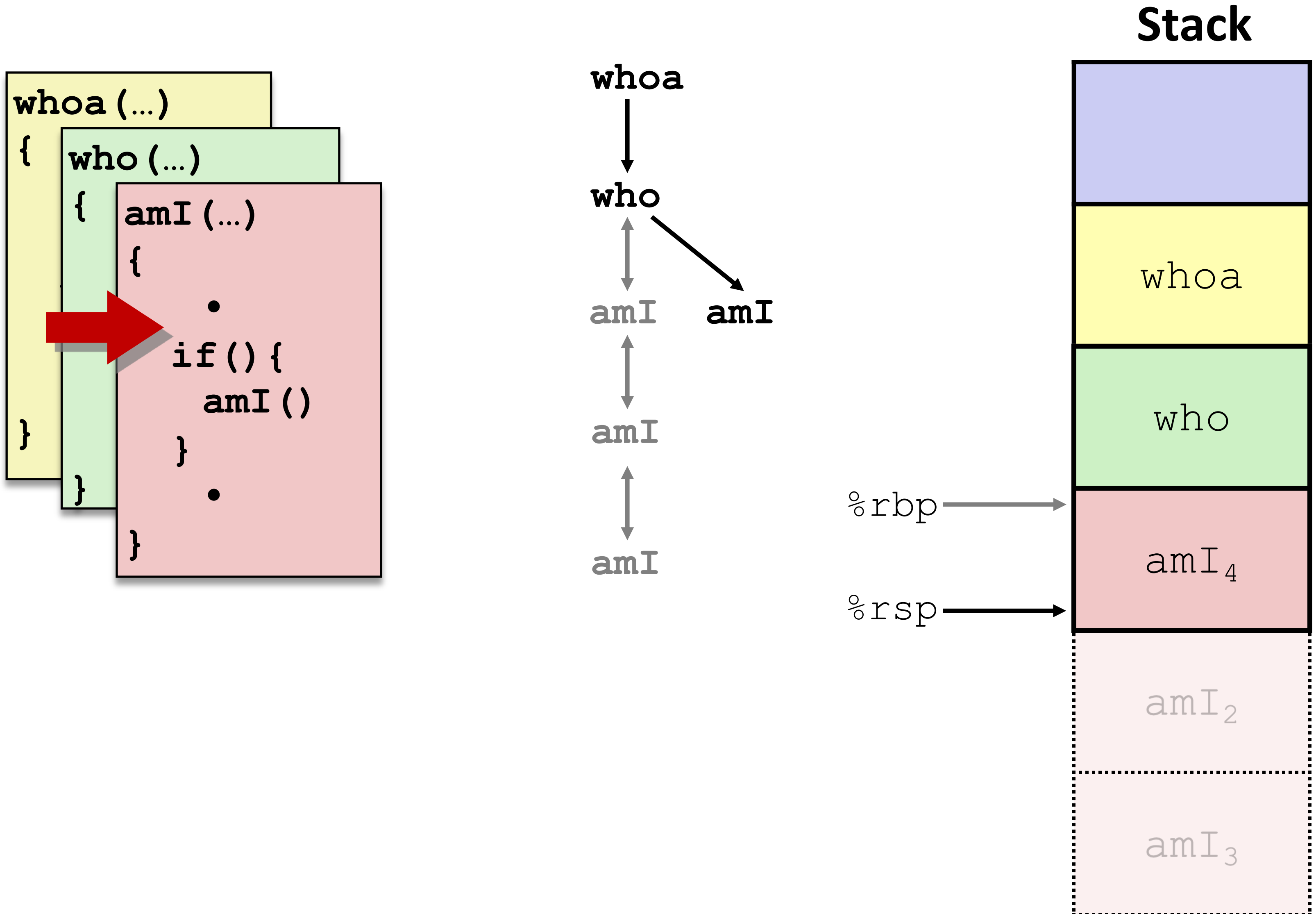




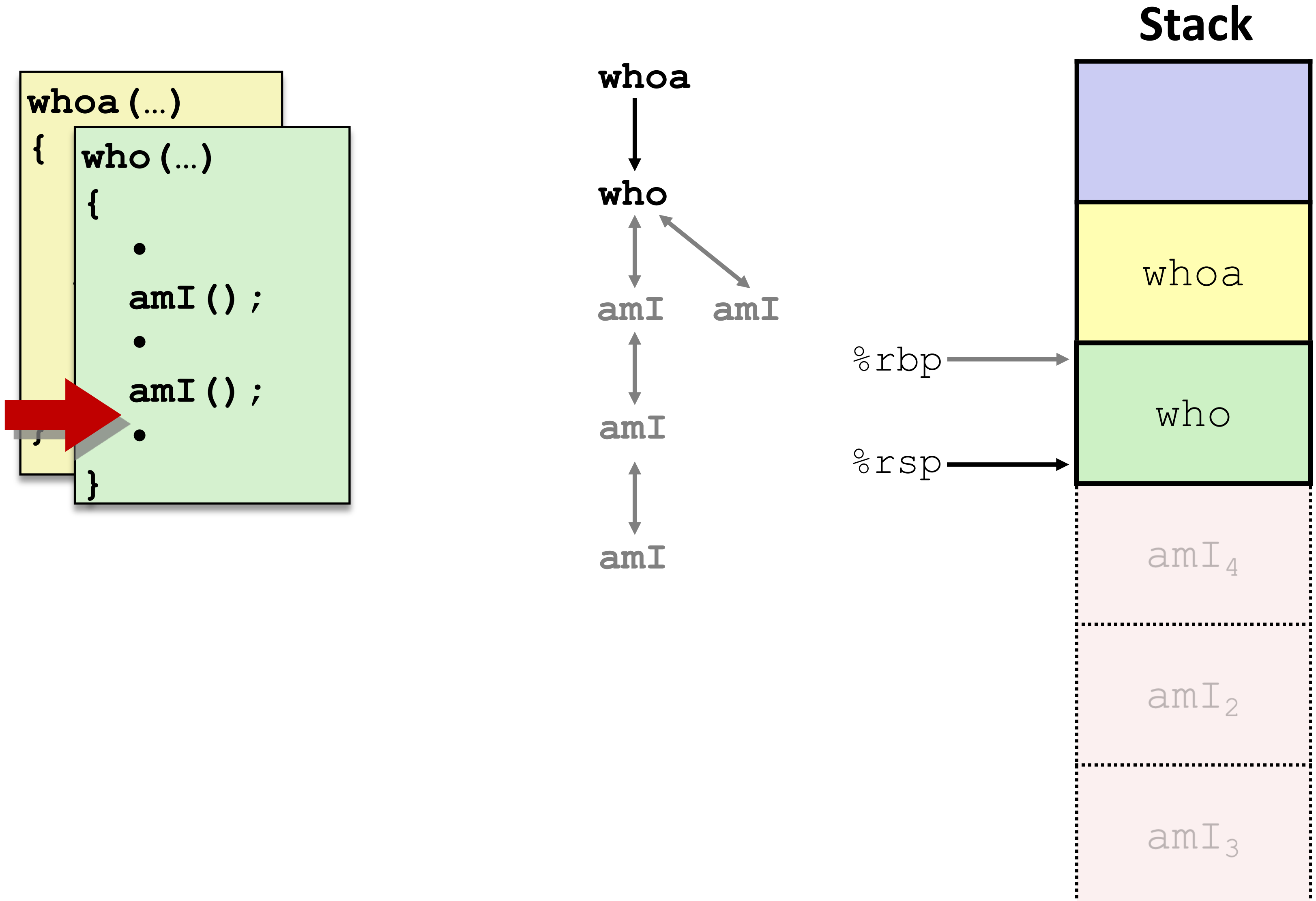
8) Return from call to `amI`



9) (second) Call to amI (4)

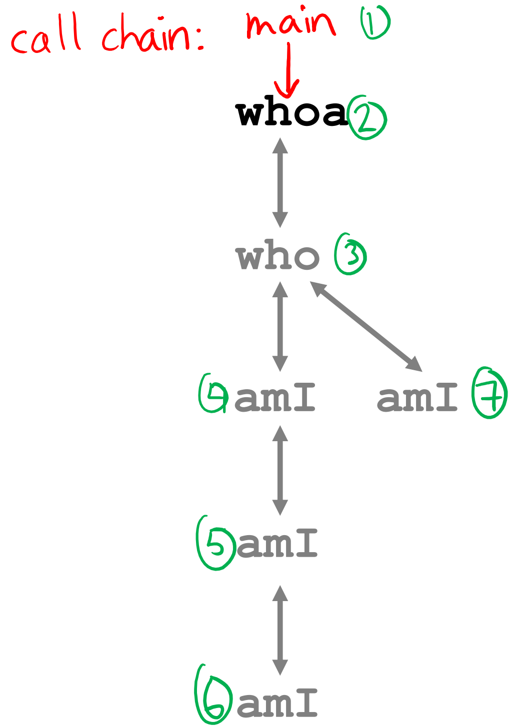


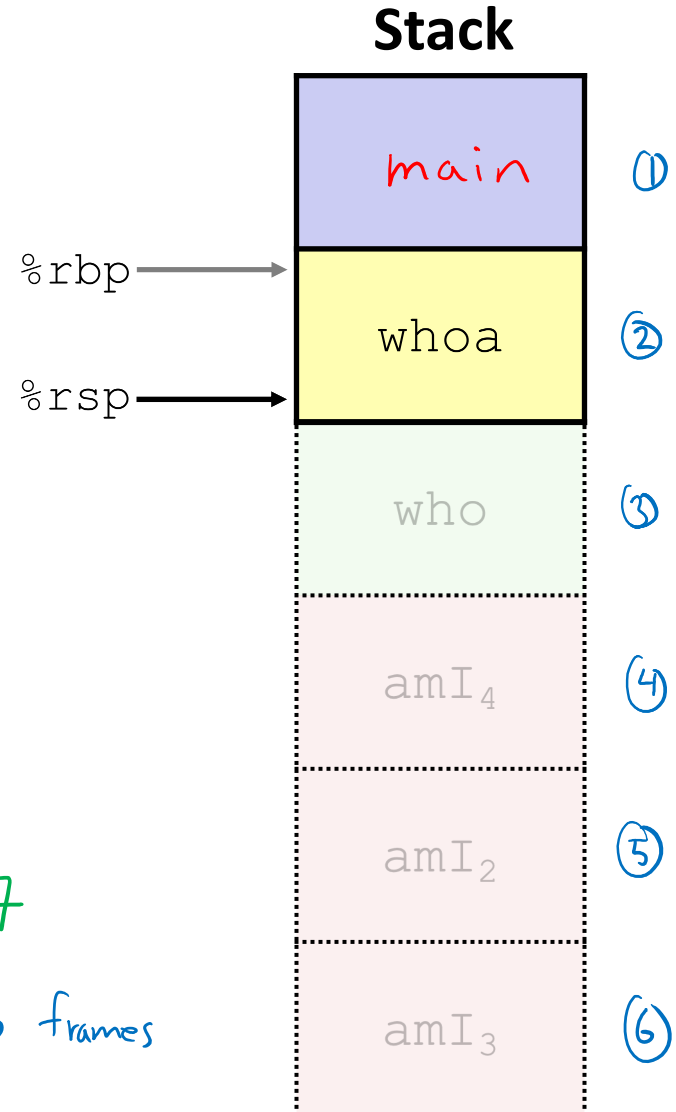
10) Return from (second) call to `amI`



11) Return from call to who

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



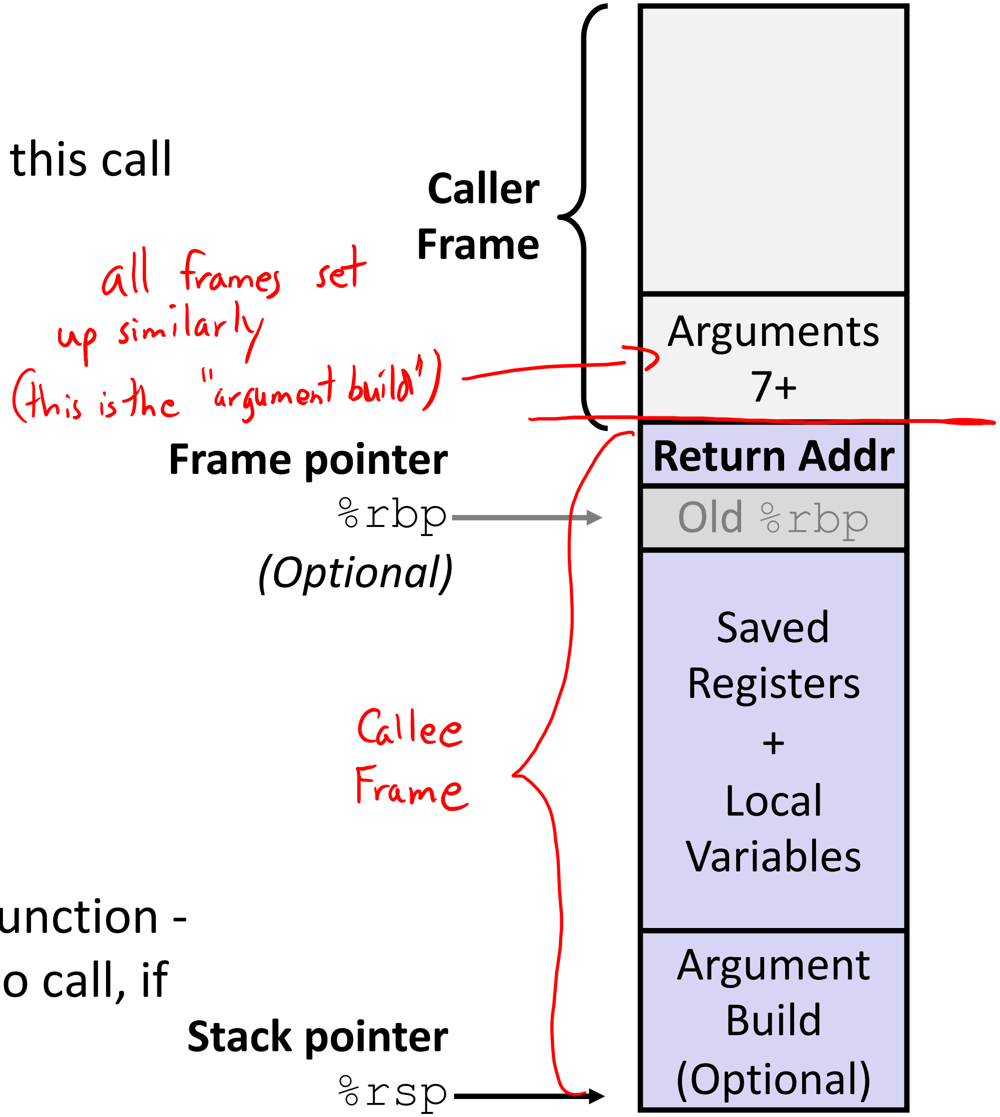


total stack frames created: 7

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)



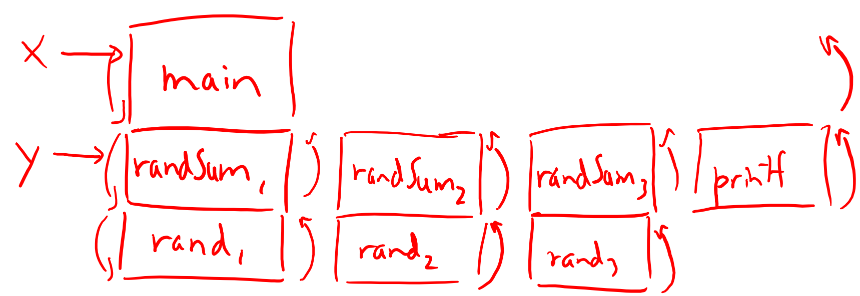
Checking in!

- 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?



1 2 3 4

Help!

Checking in!

- During the setup and execution of the following function, how many registers do we know ***must*** be modified?

```
int foo(int a, int b, int c);
```

🐶 **3**

🐈 **4**

🐑 **5**

🦄 **6**

🥶 **Help!**

Feelings check:
Stack Frames!