Loops, Switches, Intro to Stack & Procedures

CSE 351 Winter 2019

Instructors:

Max Willsey

Luis Ceze

Teaching Assistants:

Britt Henderson

Lukas Joswiak

Josie Lee

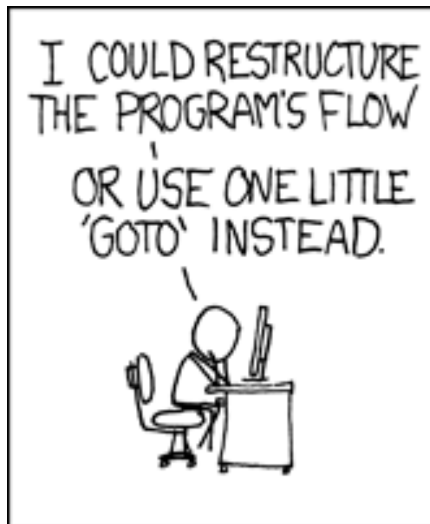
Wei Lin

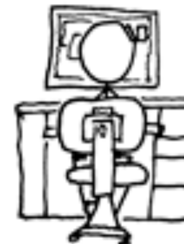
Daniel Snitkovsky

Luis Vega

Kory Watson

Ivy Yu





http://xkcd.com/571/

Administrivia

- Homework 2 due tonight
- Lab 2 due next Friday (Feb 8)
  - Ideally want to finish well before the midterm
- Homework 3 released next week
  - On midterm material, but due after the midterm

- **Midterm** (Feb 13, 8:30 am, KNE130)
  - Reference sheet + 1 *handwritten* cheat sheet
  - Find a study group! Look at past exams!
  - Average is typically around 75%
  - **Review session sections next week** (Feb 07)

Compiling Loops

While Loop:

C:

```
while ( sum != 0 ) {
    <loop body>
}
```

x86-64:

```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```

Do-while Loop:

C:

```
do {
    <loop body>
} while ( sum != 0 )
```

x86-64:

```
loopTop:
            <loop body code>
            testq  %rax, %rax
            jne    loopTop
loopDone:
```

While Loop (ver. 2):

C:

```
while ( sum != 0 ) {
    <loop body>
}
```

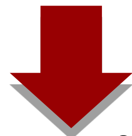
x86-64:

```
            testq %rax,
%rax
            je
loopDone
loopTop:
            <loop body
```

For-Loop → While-Loop

For-Loop:

```
for (Init; Test; Update) {
      Body
}
```



While-Loop Version:

```
Init;
while (Test) {
      Body
      Update;
}
```

Caveat: C and Java have `break` and `continue`

- Conversion works fine for break
  - **Jump to same label as loop exit condition**
- But not continue: would skip doing *Update*, which it should do with for-loops
  - **Introduce new label at *Update***

x86 Control Flow

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

Switch Statement Example

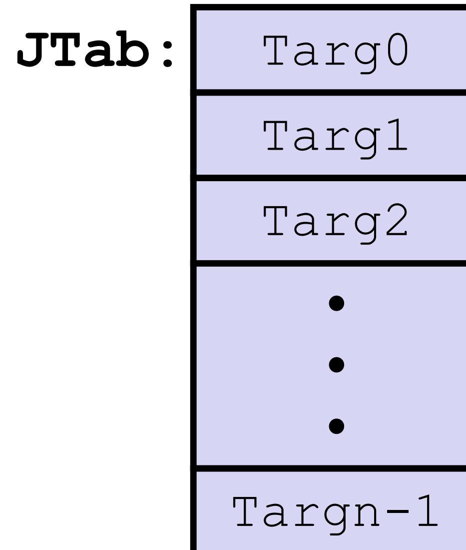
```
long switch_ex
    (long x, long y, long z)
{
    long w = 1;
    switch (x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
        /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
          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

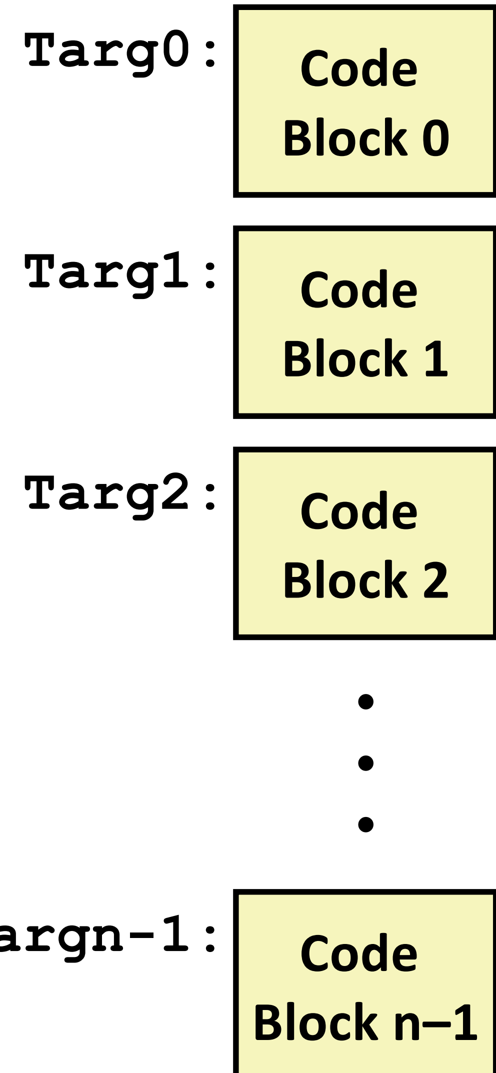
- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4
- Implemented with:
  - *Jump table*
  - *Indirect jump instruction*

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;
```

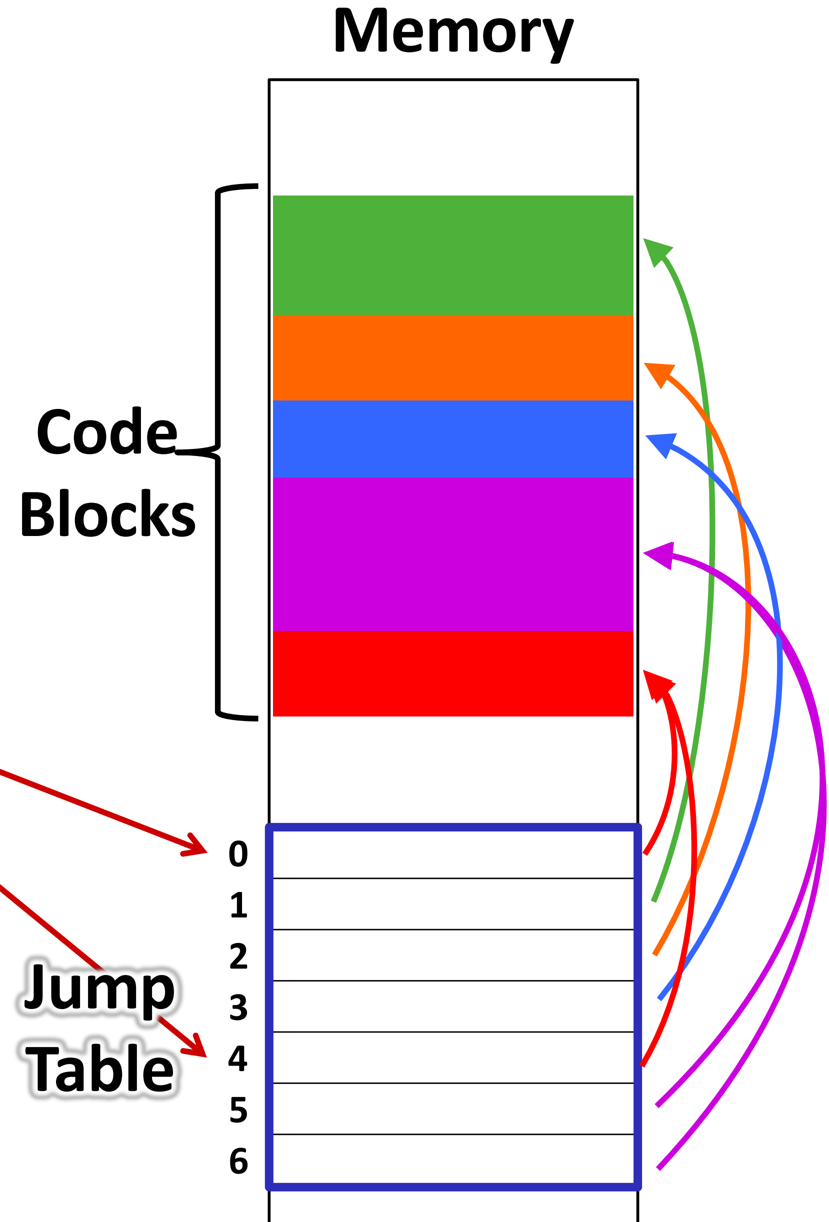
Jump Table Structure

C code:

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

Use the jump table when x ≤ 6:

```
if (x <= 6)
  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 compiler chose to not initialize w

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi          # x:6
    ja      .L8              # default
    jmp     *.L4(,%rdi,8)    # jump table
```

Take a look!

https://godbolt.org/z/dOWSFR

jump above – unsigned > catches negative default cases

Switch Statement Example

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

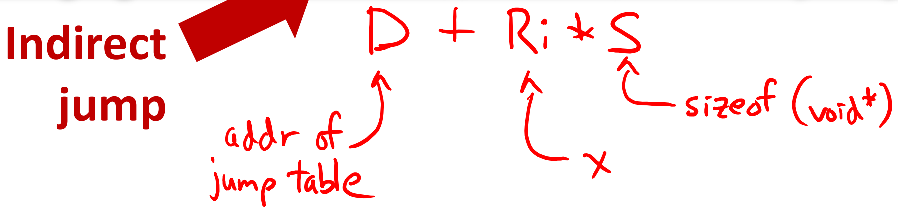
Jump table

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

address

following data is a "quad word" = 8 bytes

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi            # x:6
    ja      .L8                # default
    jmp     *.L4(,%rdi,8)      # jump table
```



Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes (address)
  - Base address at `.L4`
- Direct jump: `jmp .L8`
  - Jump target is denoted by label `.L8`
- 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 $0 \le x \le 6$

Jump table

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

Jump Table

declaring data, not instructions

8-byte memory alignment

Jump table

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

this data is 64-bits wide

```
switch(x) {
case 1:       // .L3
    w = y*z;
    break;
case 2:       // .L5
    w = y/z;
    /* Fall Through */
case 3:       // .L9
    w += z;
    break;
case 5:
case 6:       // .L7
    w -= z;
    break;
default:      // .L8
    w = 2;
}
```

Code Blocks (x == 1)

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

```
switch(x) {
  case 1:     // .L3
    w = y*z;
    break;
   . . .
}
```

```
.L3:
   movq    %rsi, %rax   # y
   imulq   %rdx, %rax   # y*z
   ret
```

Handling Fall-Through

```
long w = 1;
   . . .
switch (x) {
   . . .
  case 2:   // .L5
    w = y/z;
  /* Fall Through */
  case 3:   // .L9
    w += z;
    break;
   . . .
}
```

```
case 2:
      w = y/z;
      goto merge;
```

```
case 3:
      w = 1;
```

```
merge:
      w += z;
```

More complicated choice than "just fall-through" forced by "migration" of `w = 1;`

- *Example compilation trade-off*

Code Blocks (x == 2, x == 3)

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

```
long w = 1;
     . . .
switch (x) {
   . . .
  case 2:   // .L5
    w = y/z;
  /* Fall Through */
  case 3:   // .L9
    w += z;
    break;
   . . .
}
```

```
.L5:                          # Case 2:
    movq    %rsi, %rax        #   y in rax
    cqto                      #   Div prep
    idivq   %rcx              #   y/z
    jmp     .L6               #   goto merge
.L9:                          # Case 3:
    movl    $1, %eax          #   w = 1
.L6:                          # merge:
    addq    %rcx, %rax        #   w += z
    ret
```

Code Blocks (rest)

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

```
switch (x) {
    . . .
   case 5:  // .L7
   case 6:  // .L7
     w -= z;
     break;
   default: // .L8
     w = 2;
}
```

```
.L7:                  # Case 5,6:
  movl   $1, %eax     #   w = 1
  subq   %rdx, %rax   #   w -= z
  ret
.L8:                  # Default:
  movl   $2, %eax     #   2
  ret
```

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

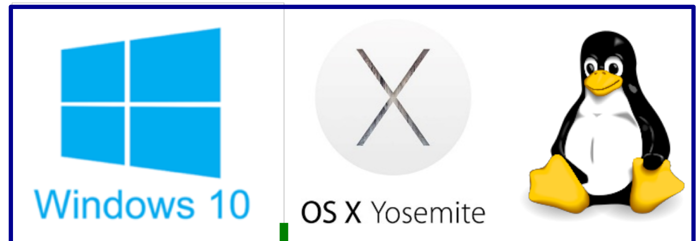
Assembly language:

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

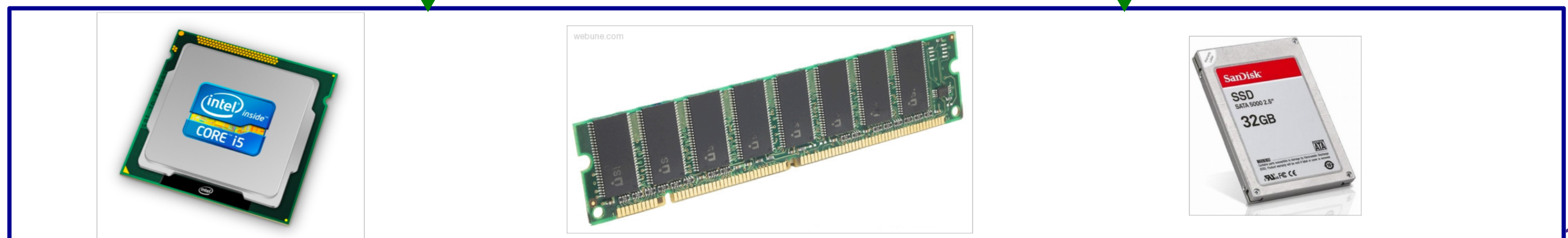
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:



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

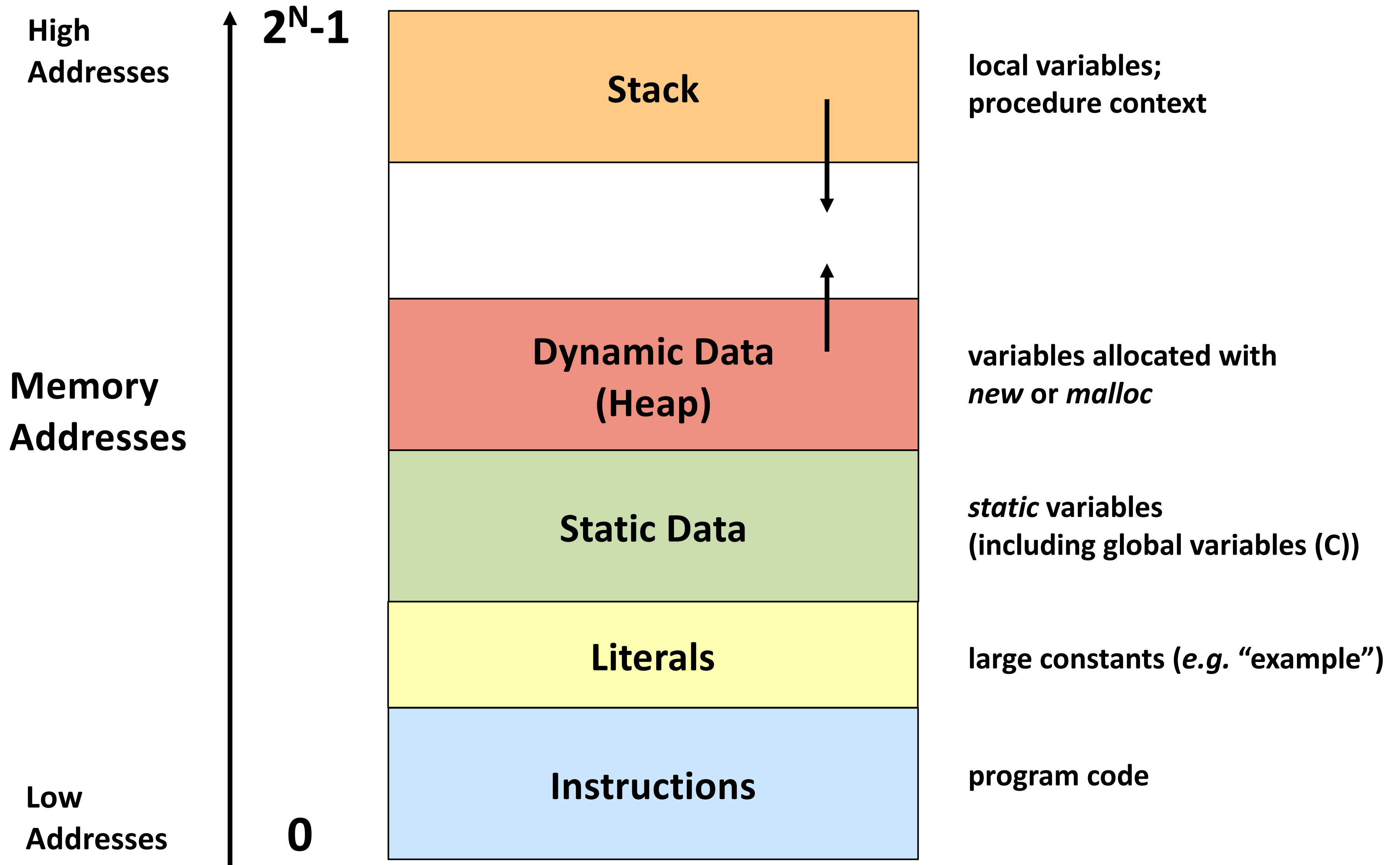
```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

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

Procedures

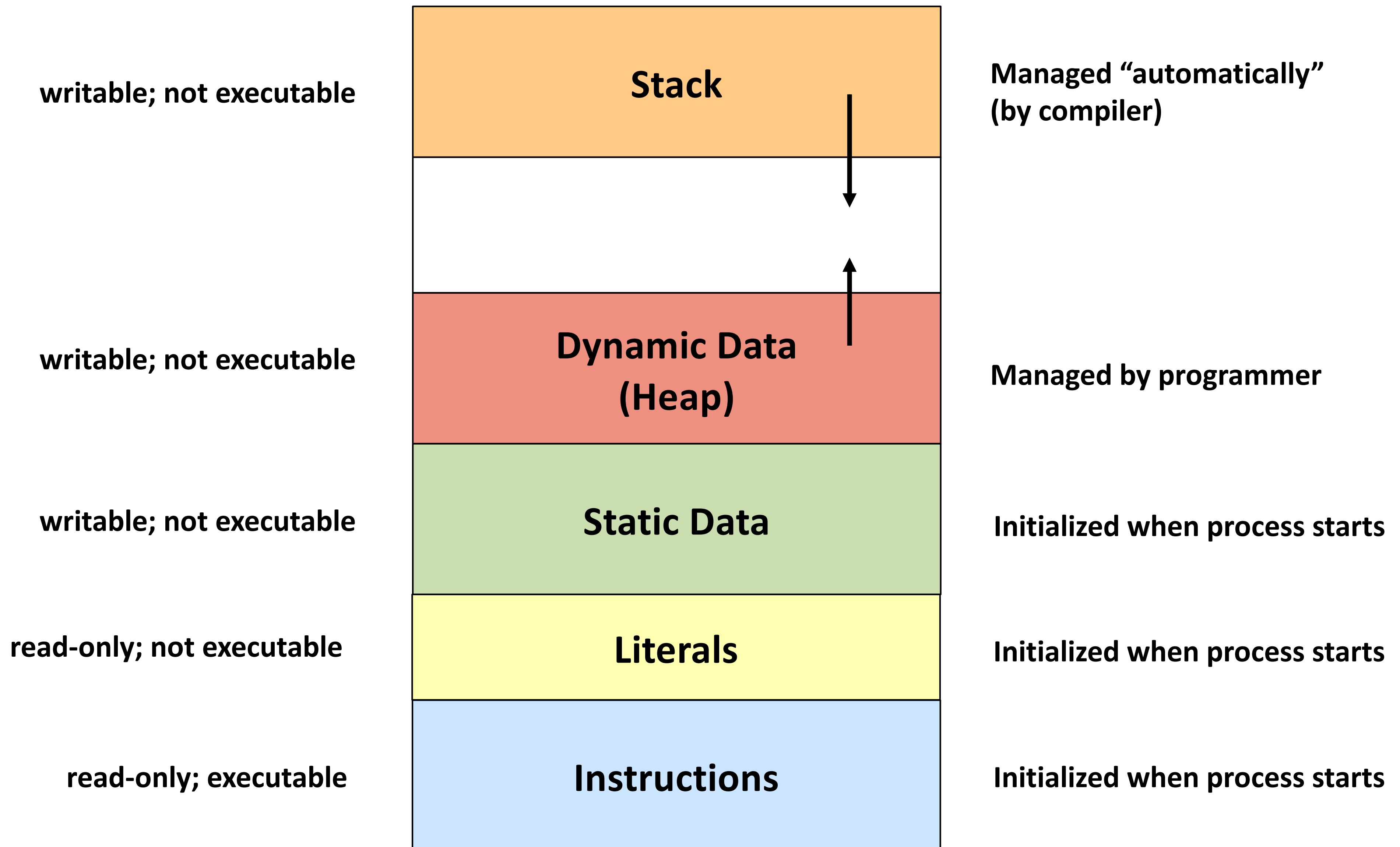
- **Stack Structure**
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Simplified Memory Layout



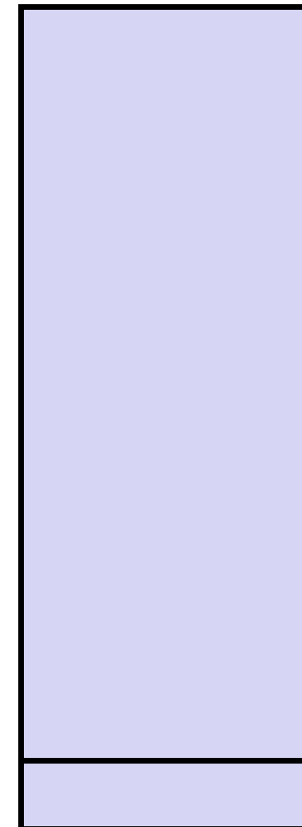
segmentation faults?

Memory Permissions



x86-64 Stack

- 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



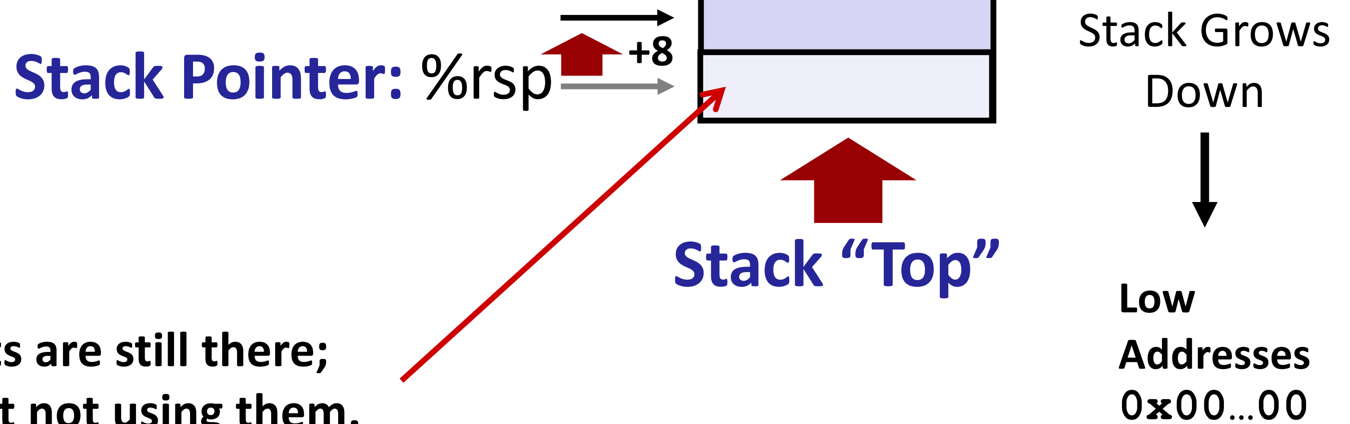
x86-64 Stack: Push

- `pushq` *src*
  - 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

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

x86-64 Stack: Pop

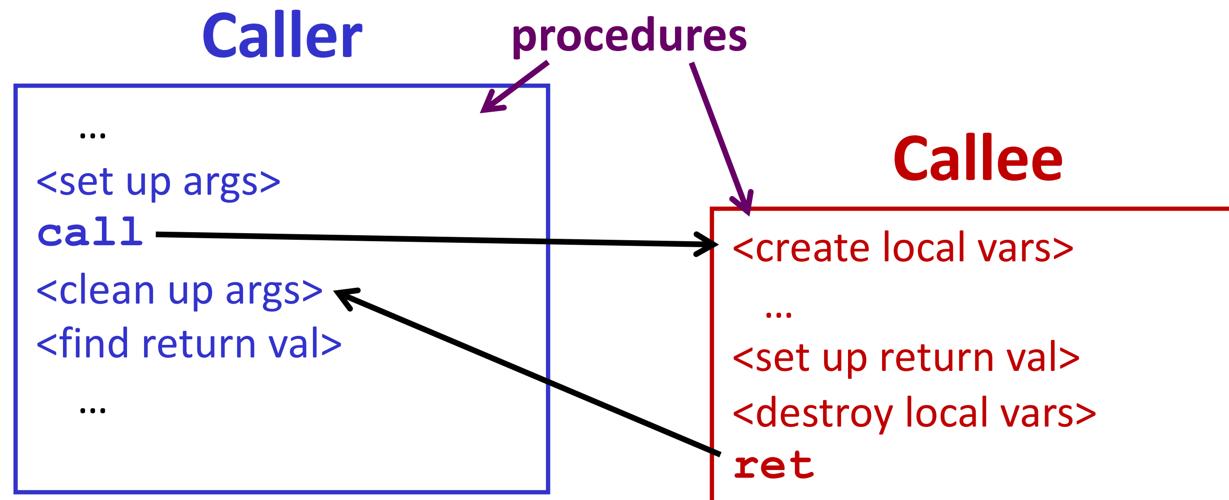
- `popq` *dst*
  - Load value at address given by `%rsp`
  - Store value at *dst*
  - ***Increment*** `%rsp` by 8
- Example:
  - **`popq %rcx`**
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`



Procedures

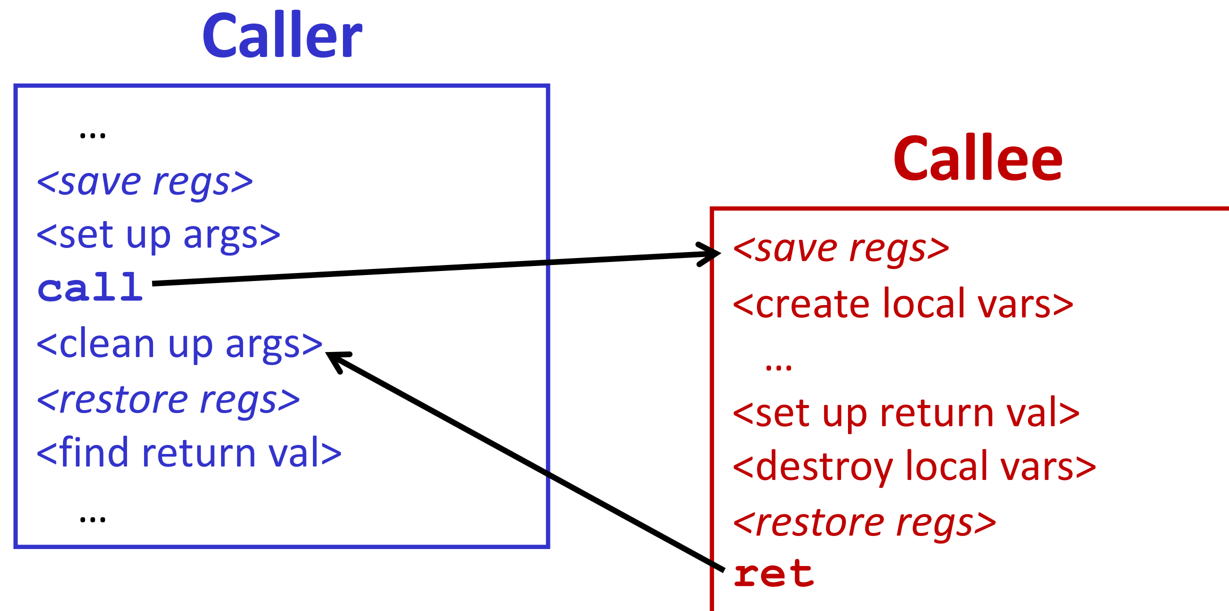
- Stack Structure
- **Calling Conventions**
  - **Passing control**
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

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;
}
```

```
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
```

Compiler Explorer:
https://godbolt.org/g/cKKDZn

```
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`
  1) Push return address on stack (*why? which address?*)
  2) Jump to ***`label`***

Procedure Control Flow

- 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`**

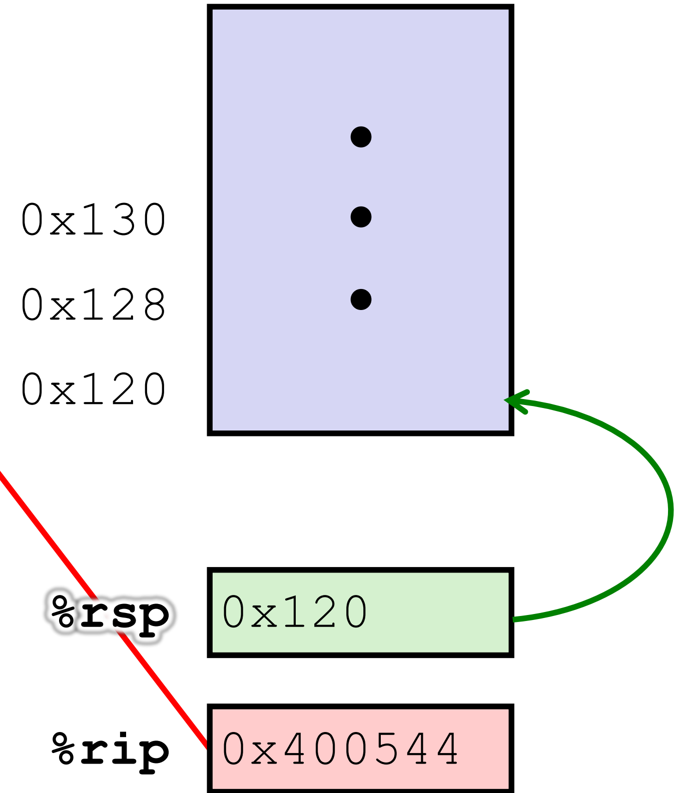
next instruction happens to be a move, but could be anything

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

Procedure Call Example (step 1)

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

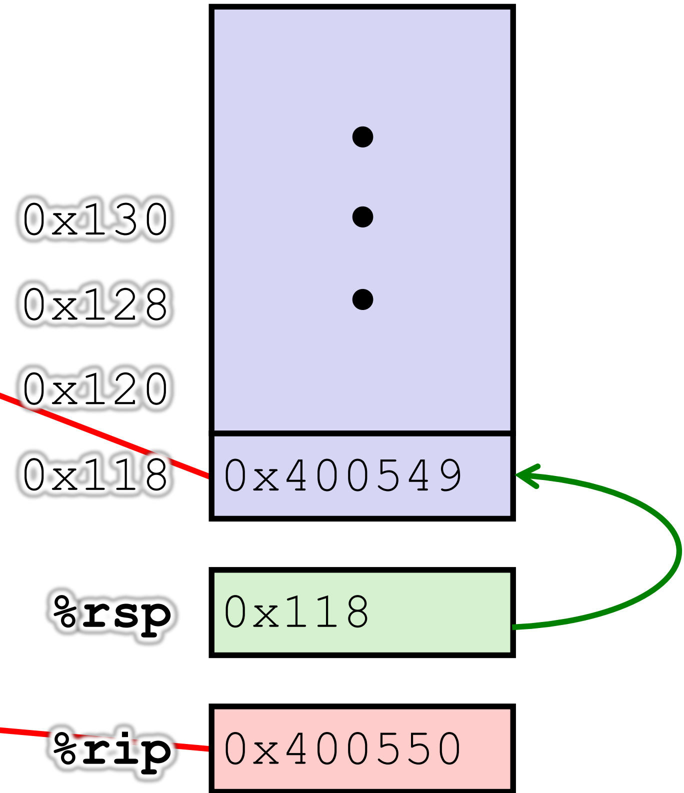
```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```



Procedure Call Example (step 2)

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

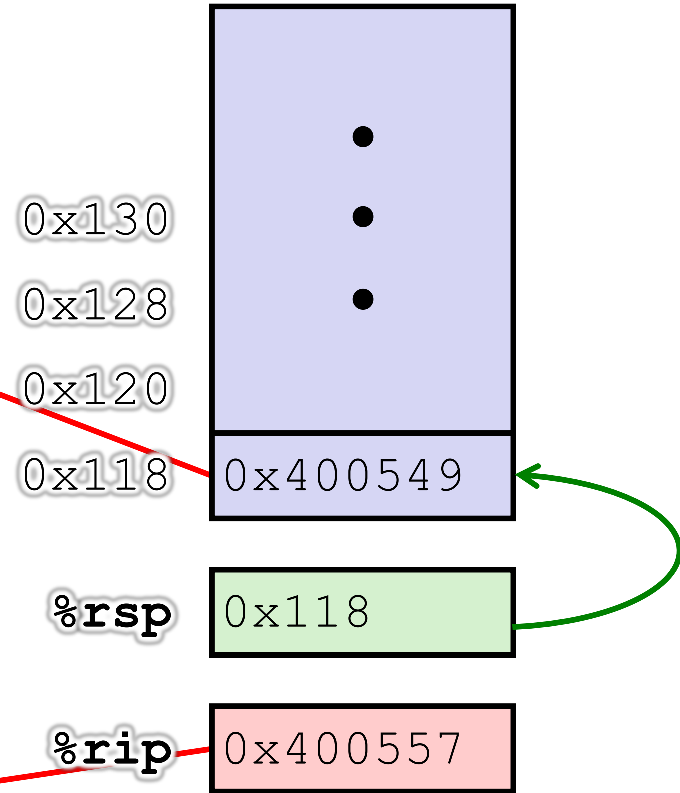
```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```



Procedure Return Example (step 1)

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

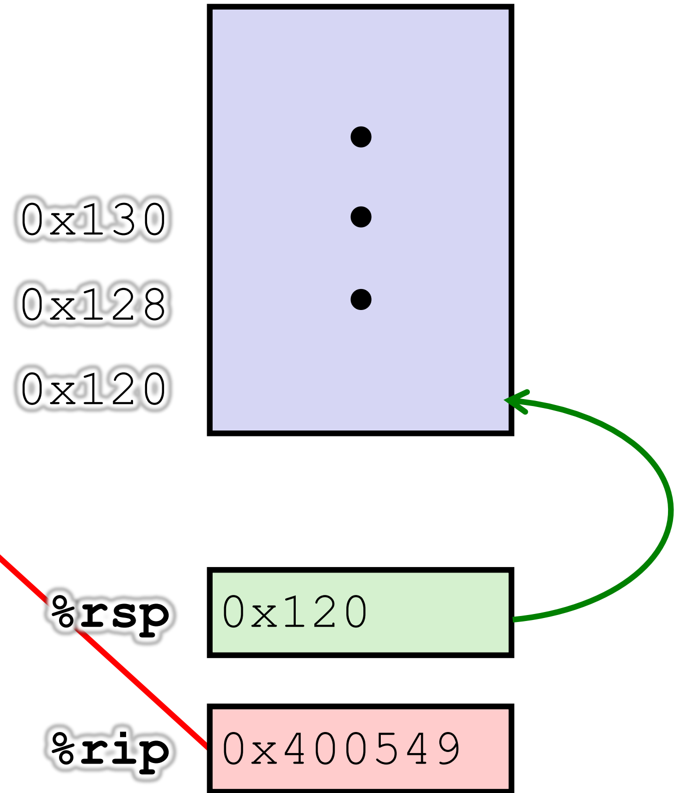
```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```



Procedure Return Example (step 2)

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

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



Procedures

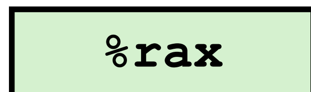
- Stack Structure
- **Calling Conventions**
  - Passing control
  - **Passing data**
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Procedure Data Flow

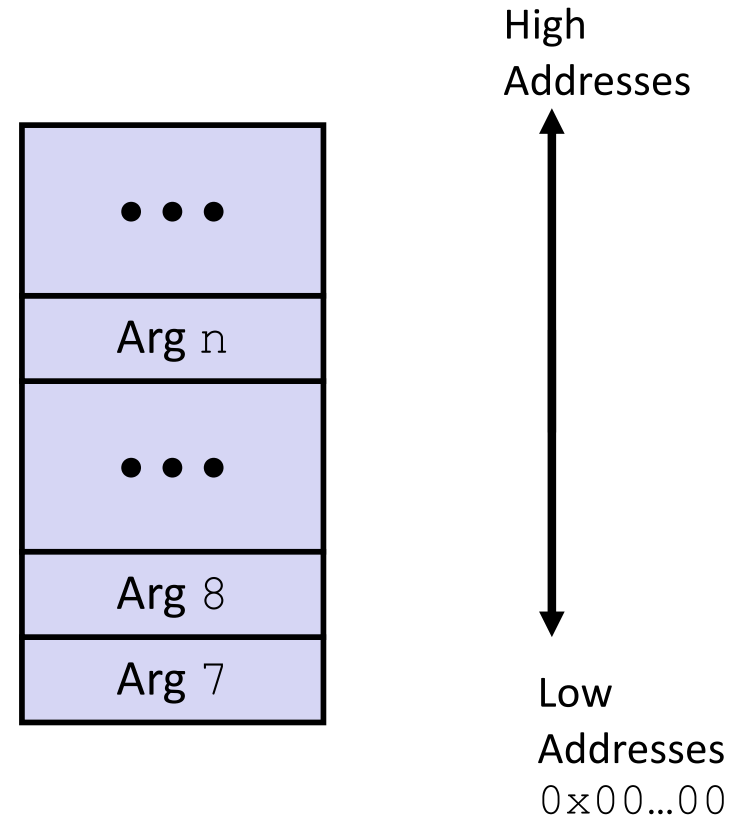
Registers (NOT in Memory)

- First 6 arguments

- Return value



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 pointer
- 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

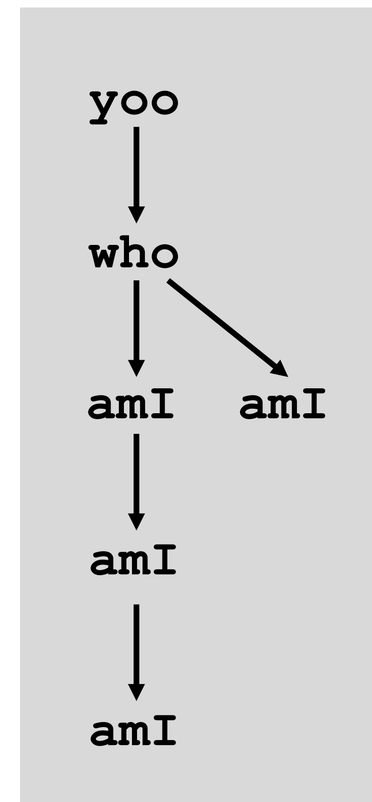
Call Chain Example

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

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

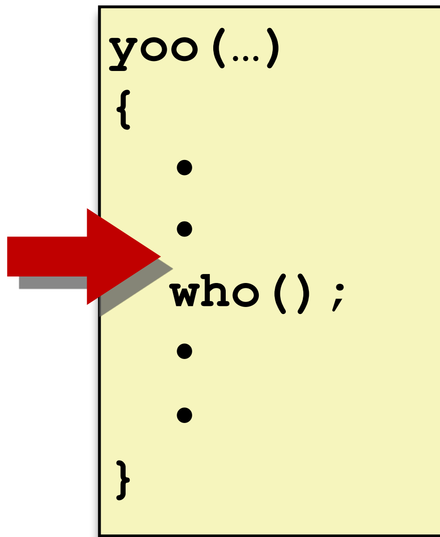
```
amI(…)
{
    •
    if(…){
      amI()
    }
    •
}
```

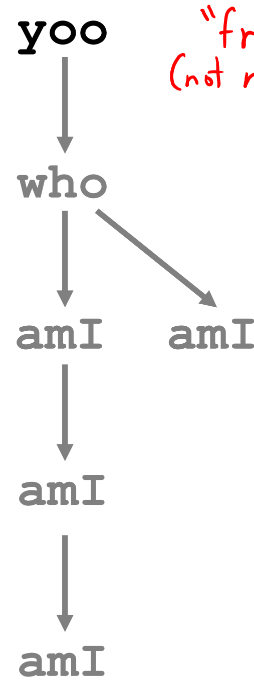
Example
Call Chain

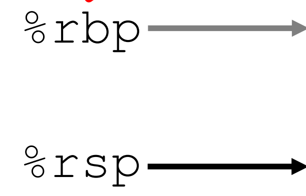


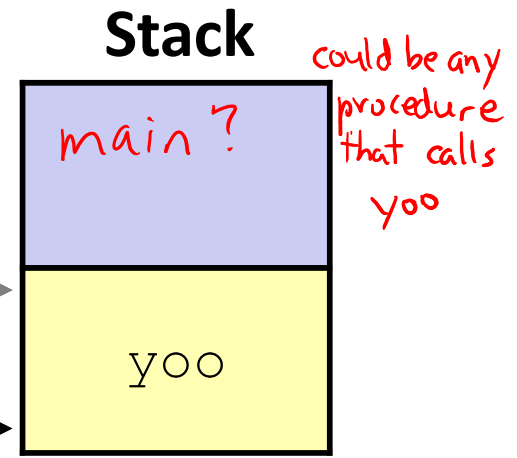
Procedure amI is recursive
(calls itself)

1) Call to yoo

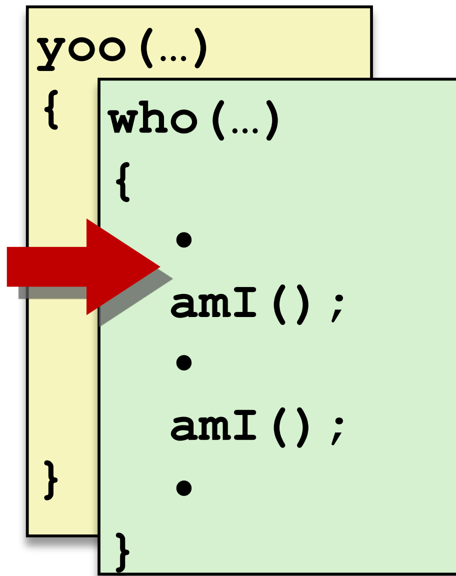


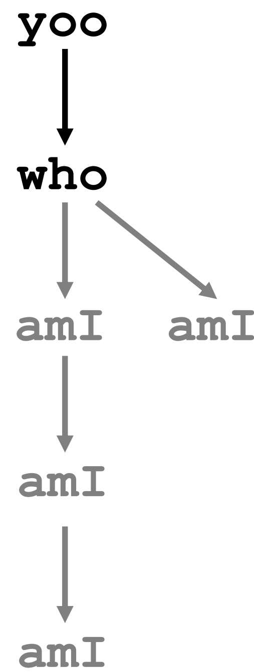


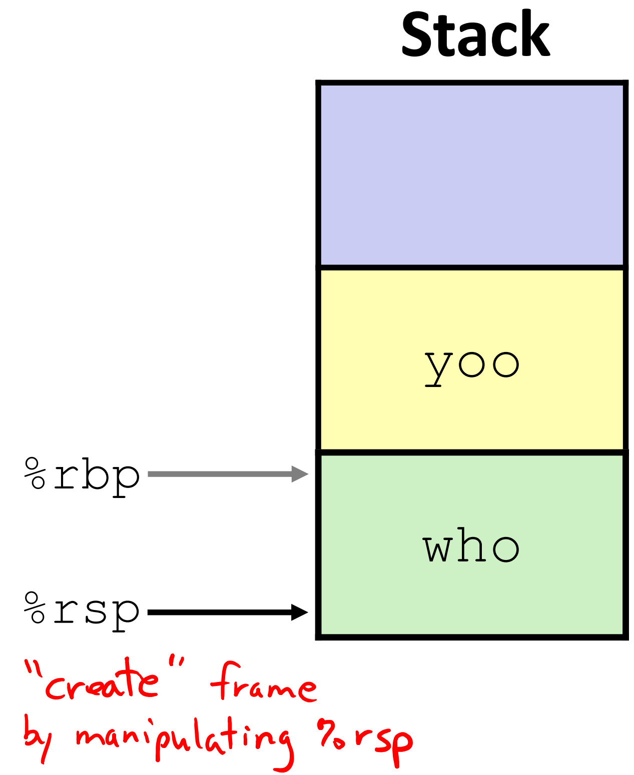




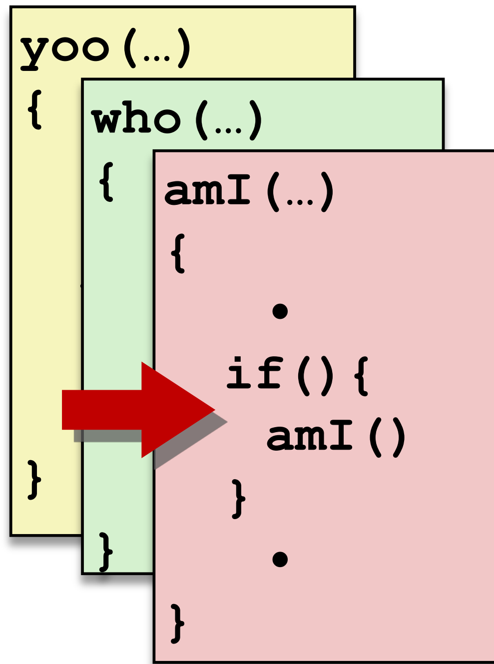
2) Call to who

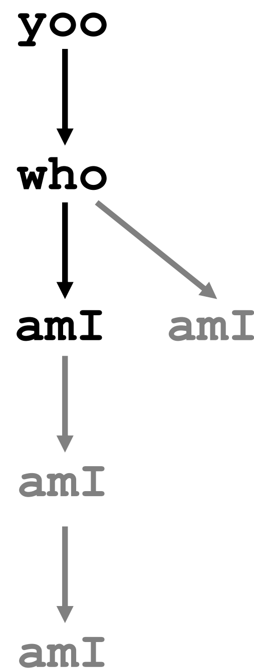


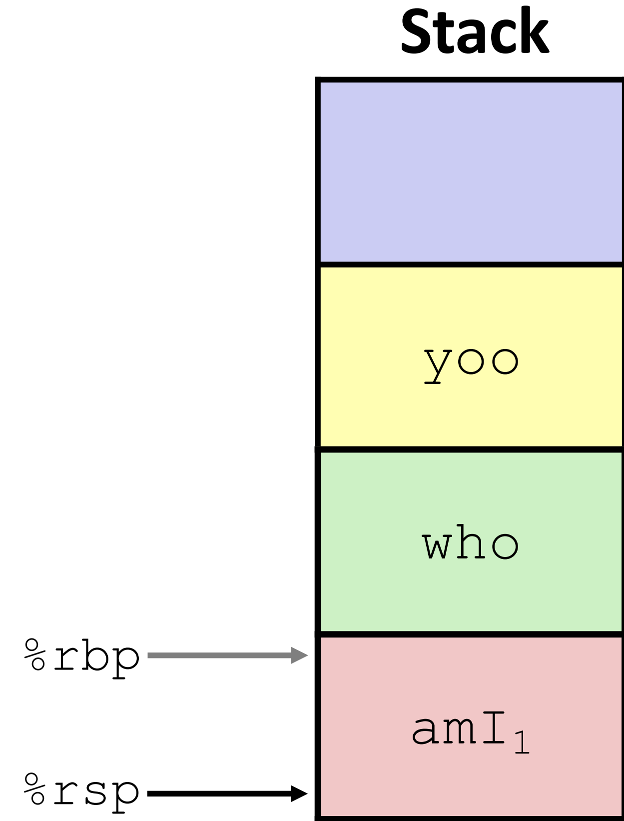




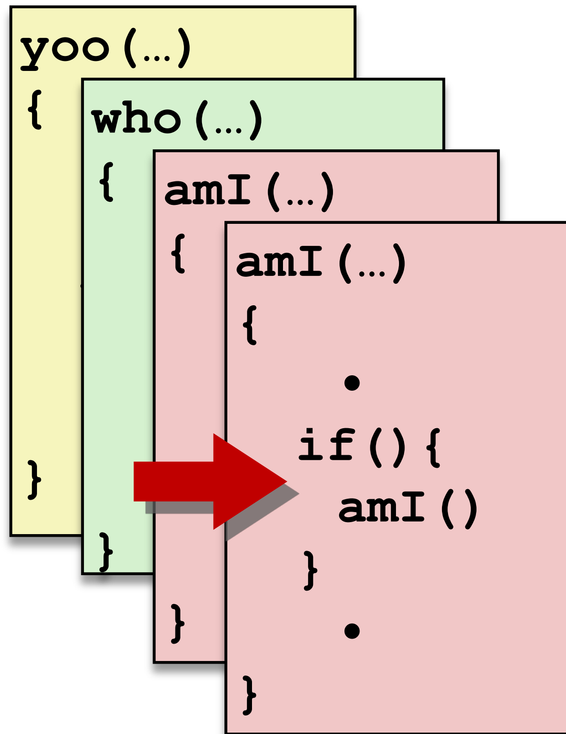
3) Call to amI (1)

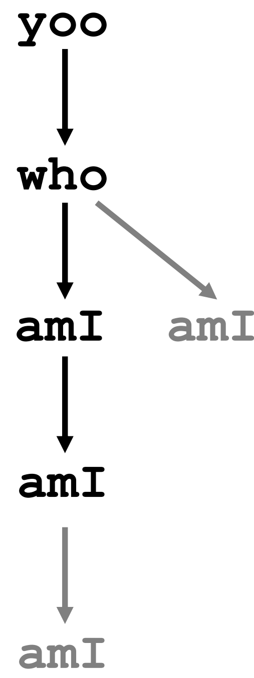


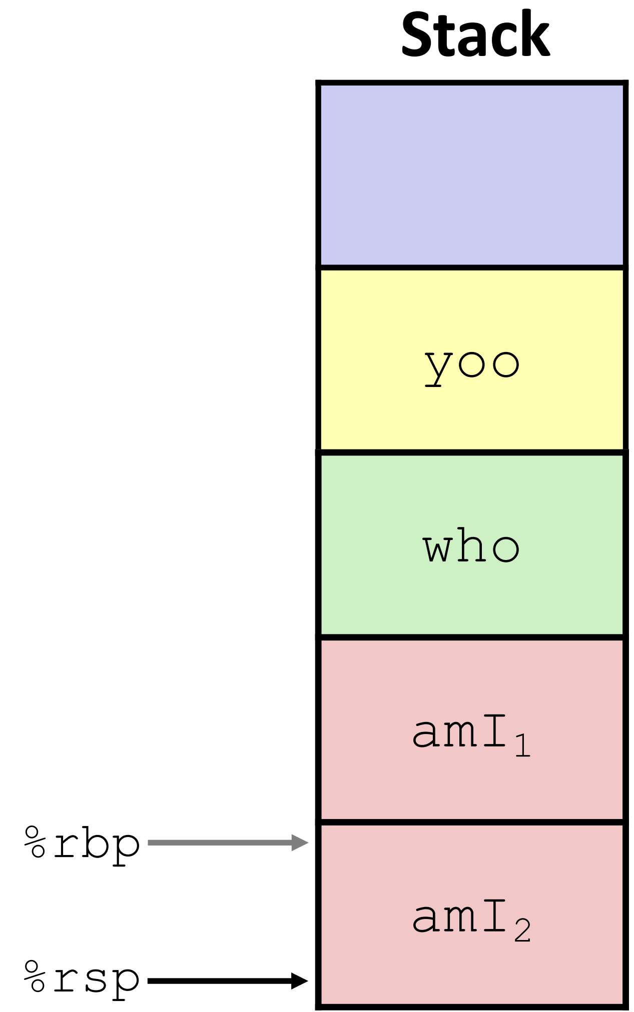




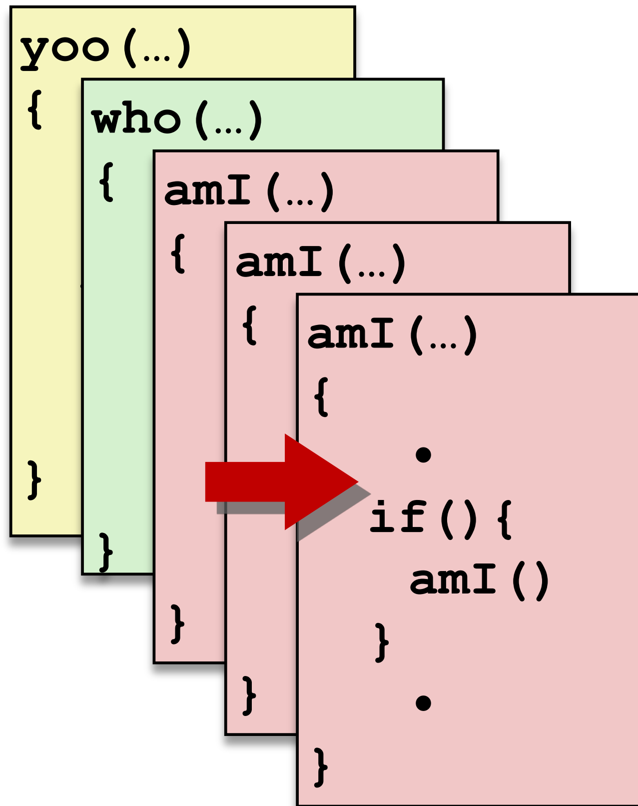
4) Recursive call to amI (2)

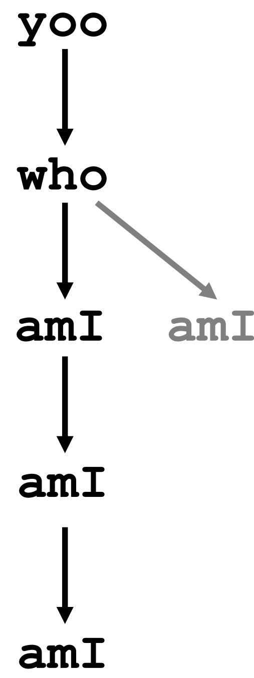


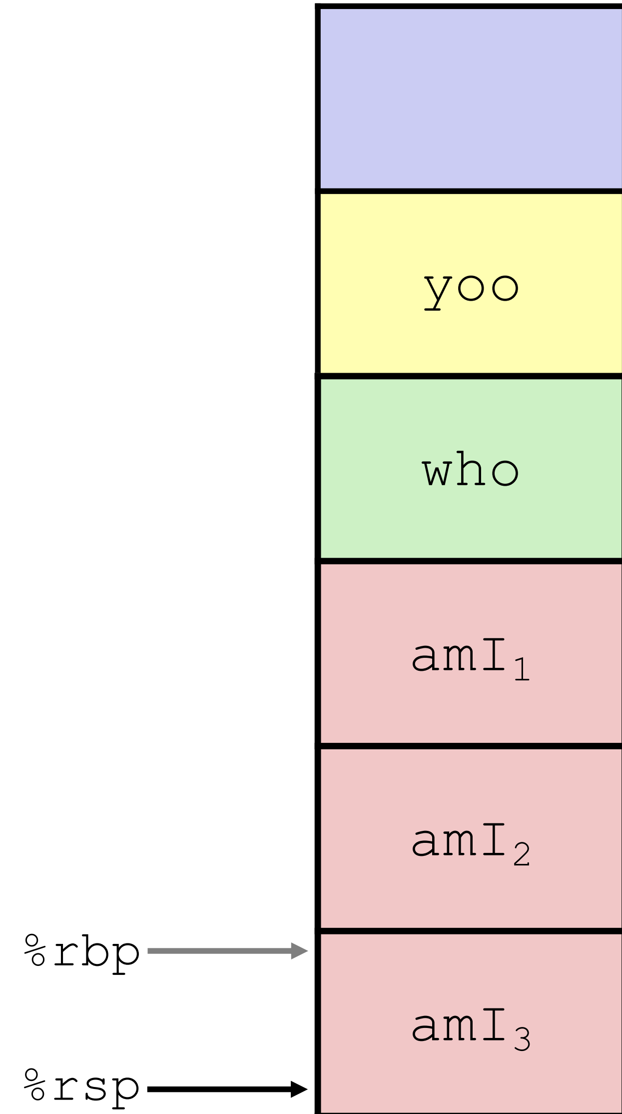




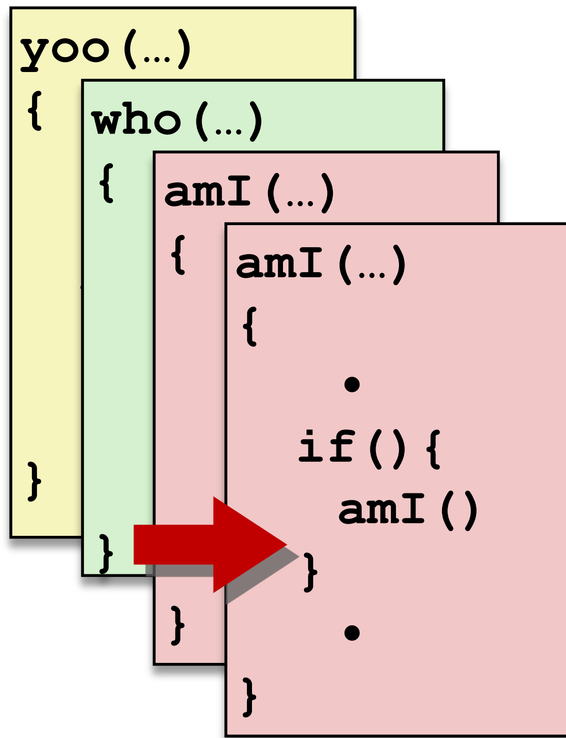
5) (another) Recursive call to `amI` (3)

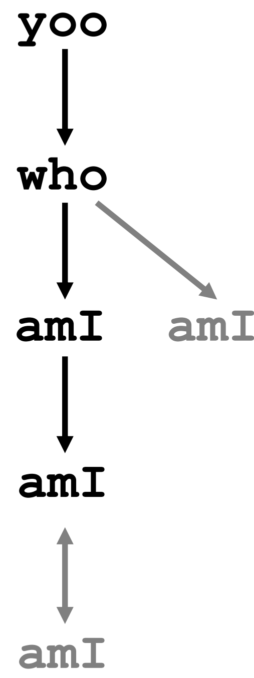


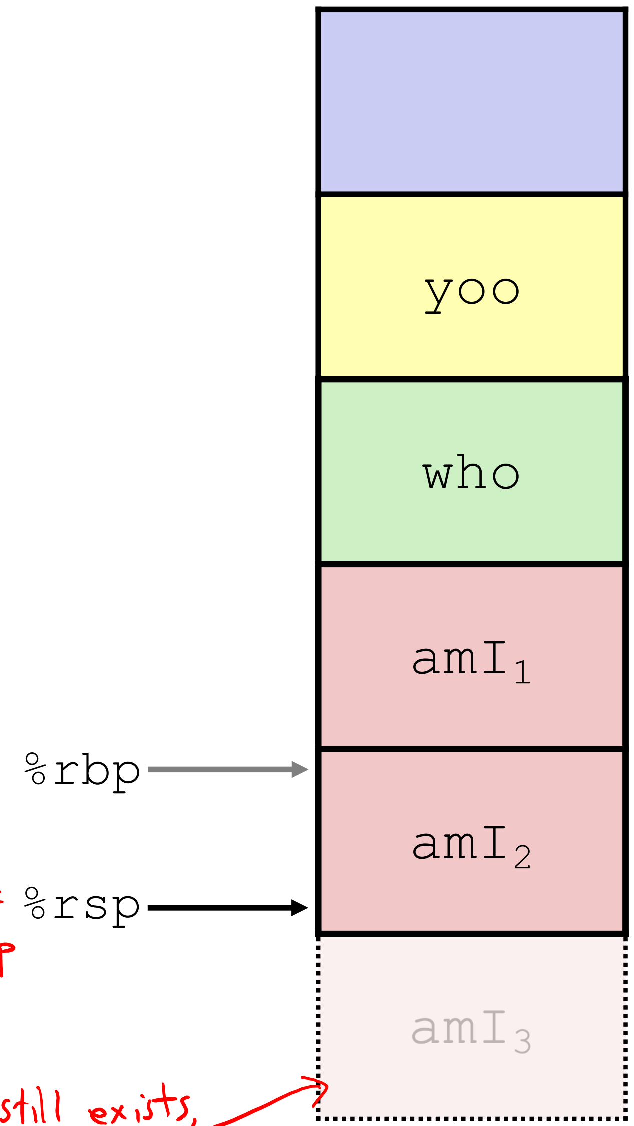




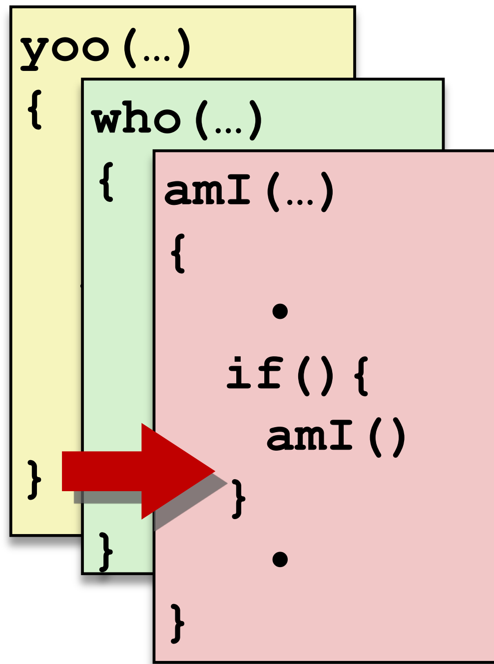
6) Return from (another) recursive call to `amI`

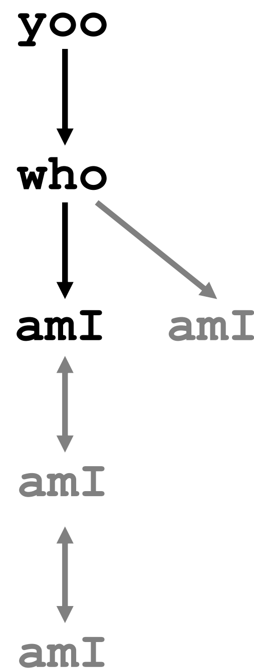


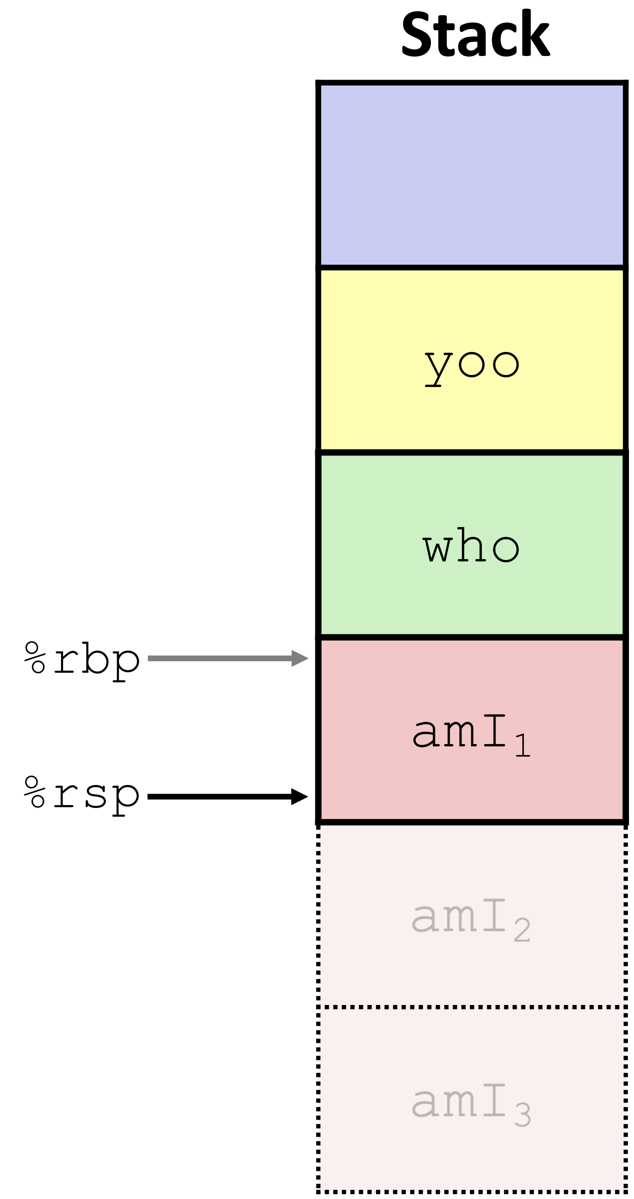




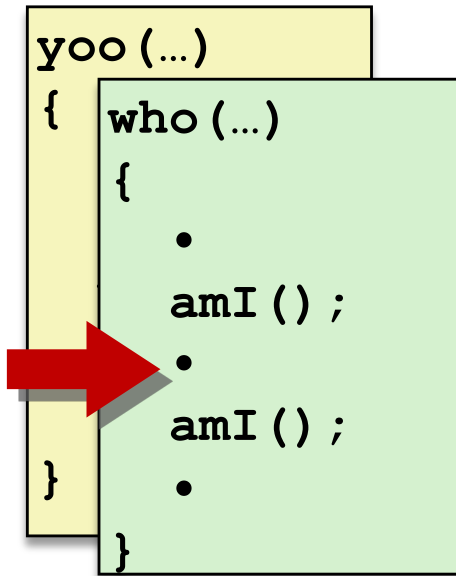
7) Return from recursive call to amI

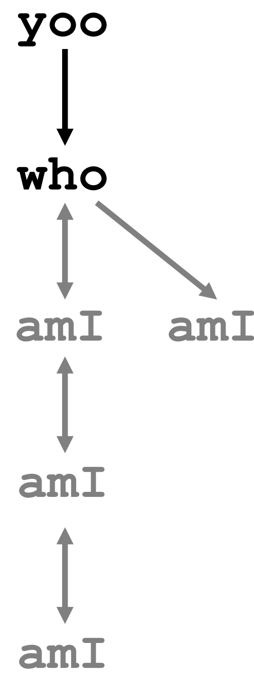


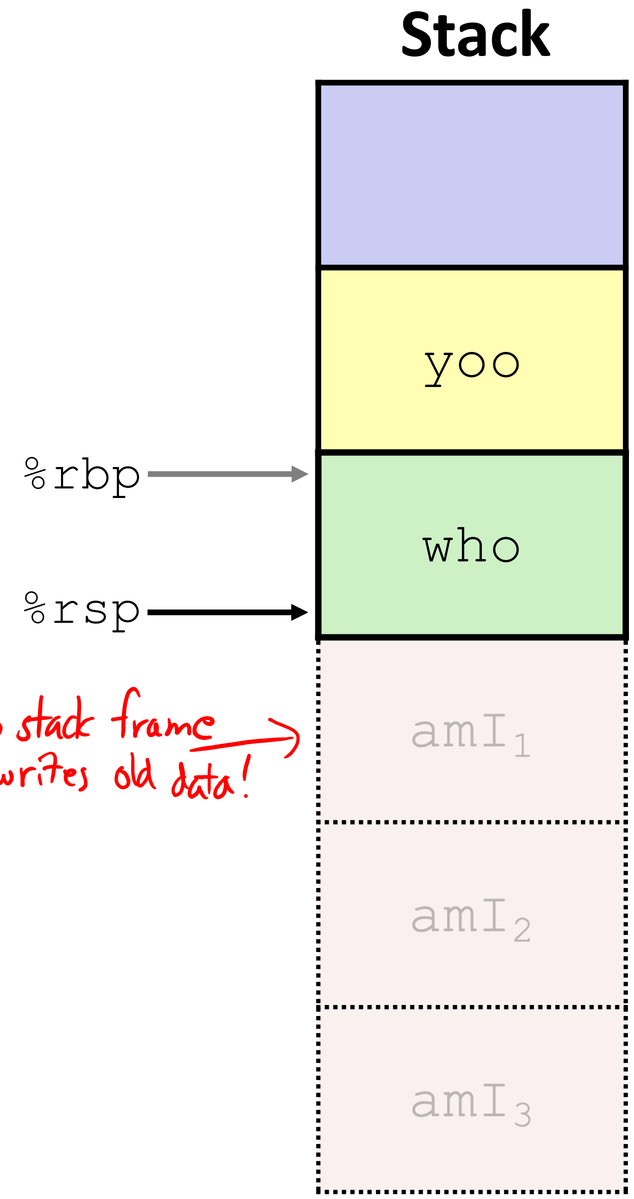




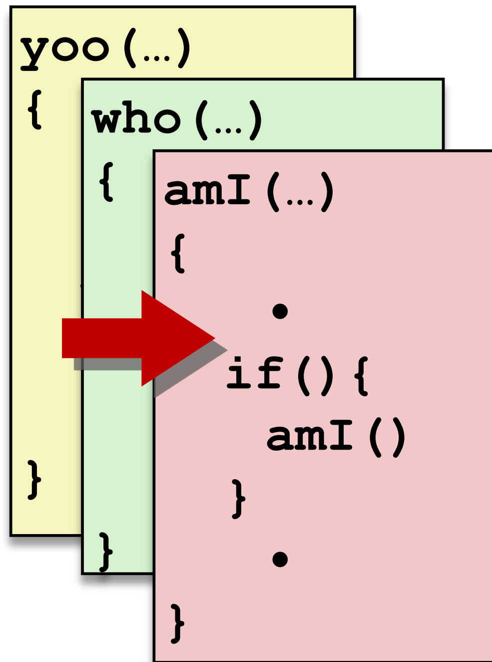
8) Return from call to amI

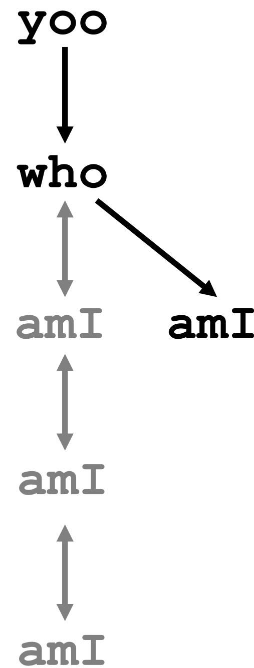


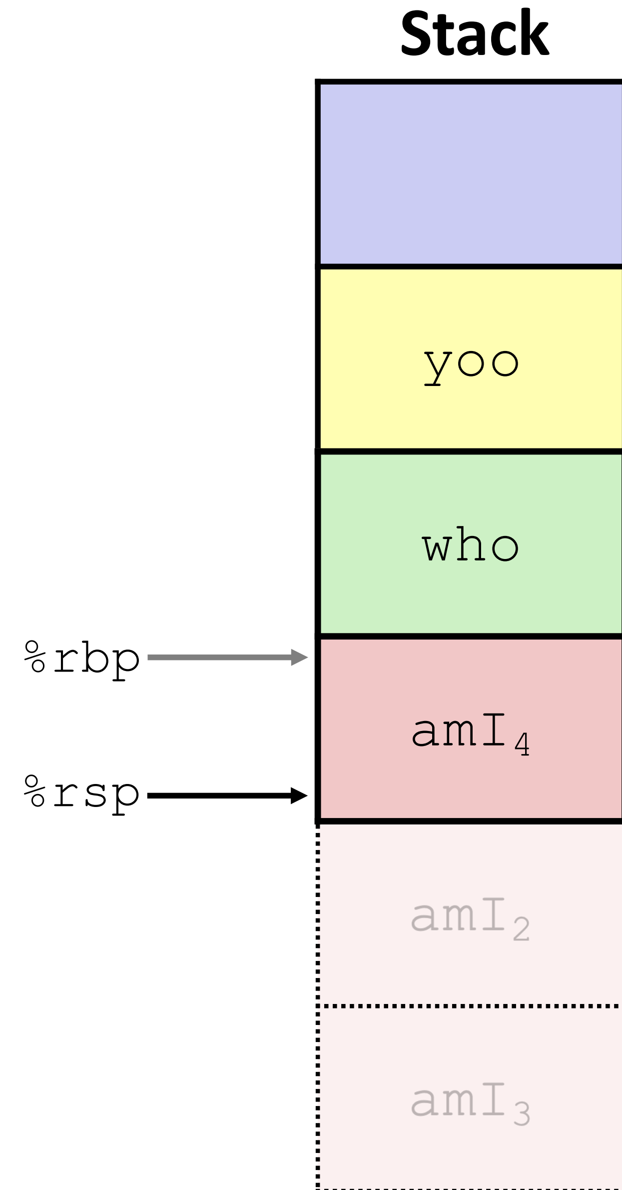




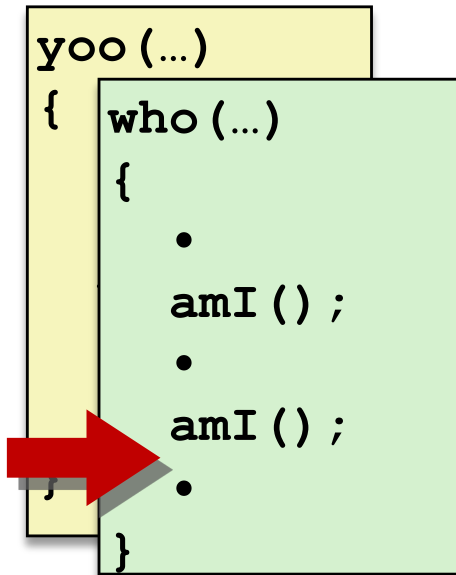
9) (second) Call to amI (4)

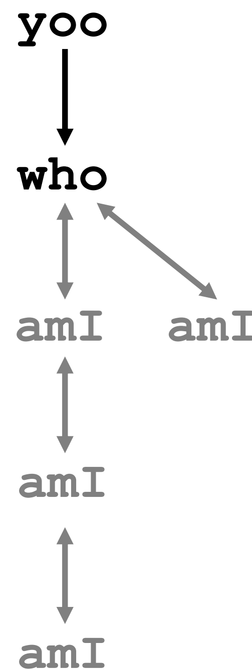


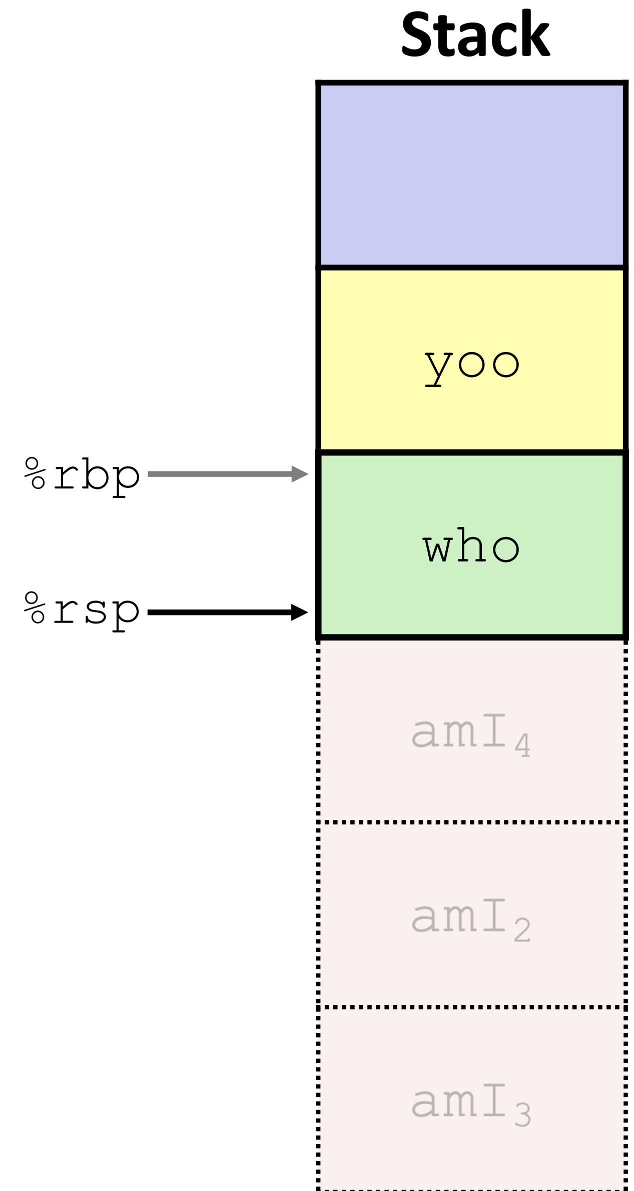




10) Return from (second) call to `amI`

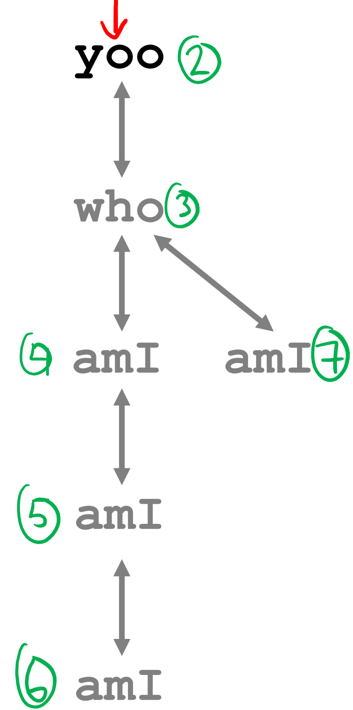


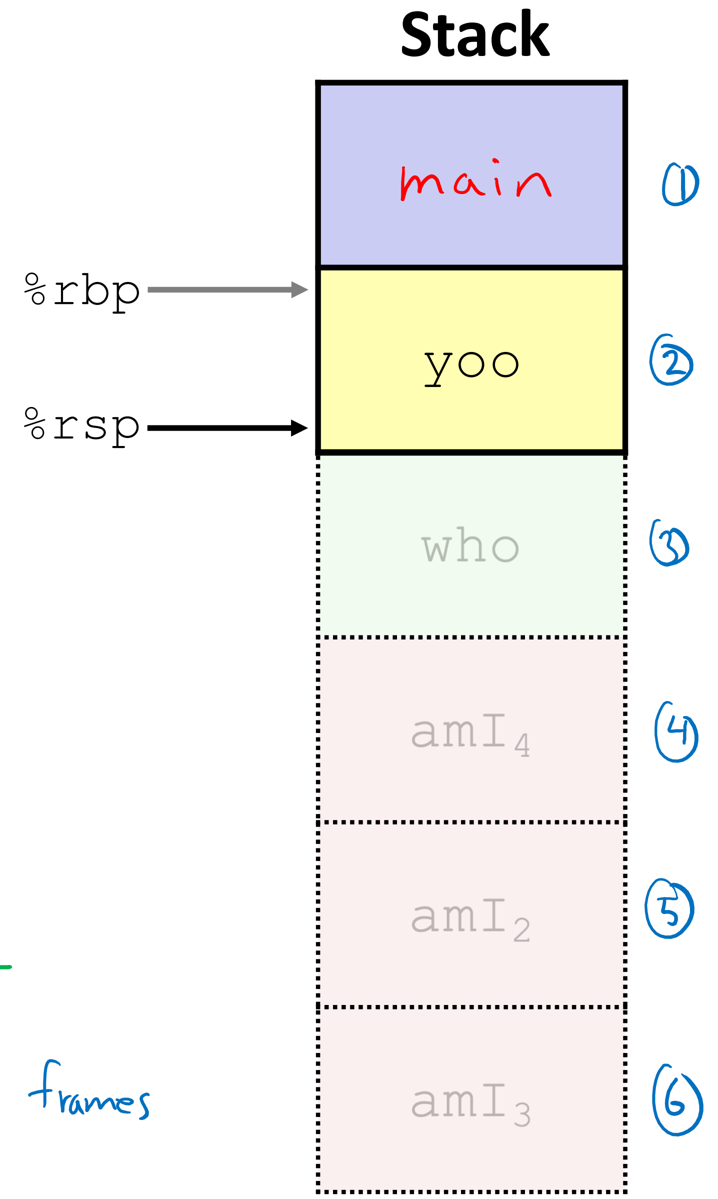




11) Return from call to who

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



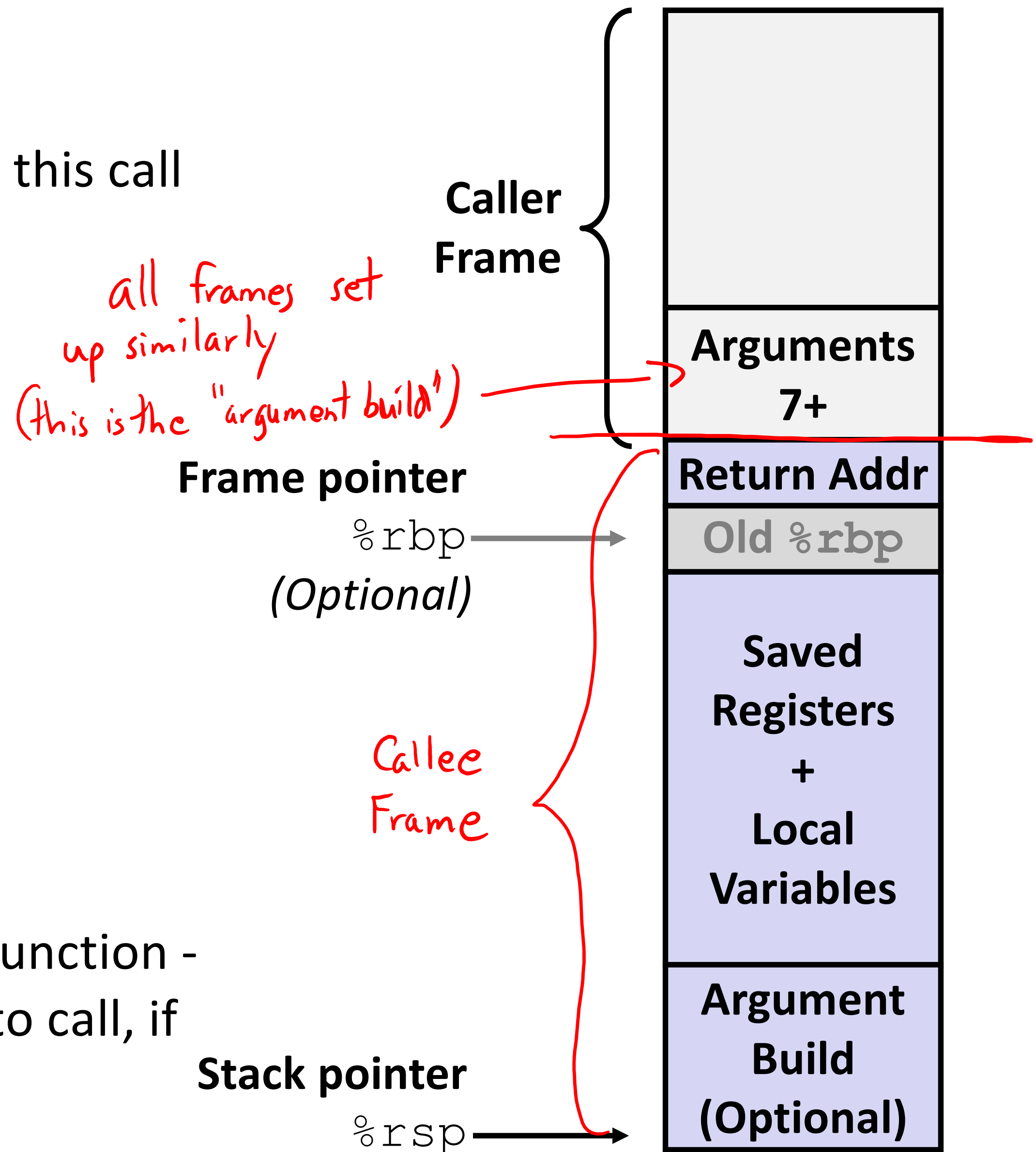


total stack frames created: 7

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)



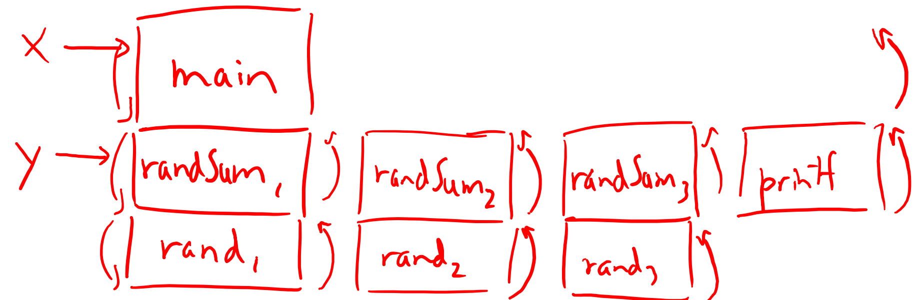
Peer Instruction Question

Vote only on 3rd question at http://PollEv.com/justinh

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

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

```
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