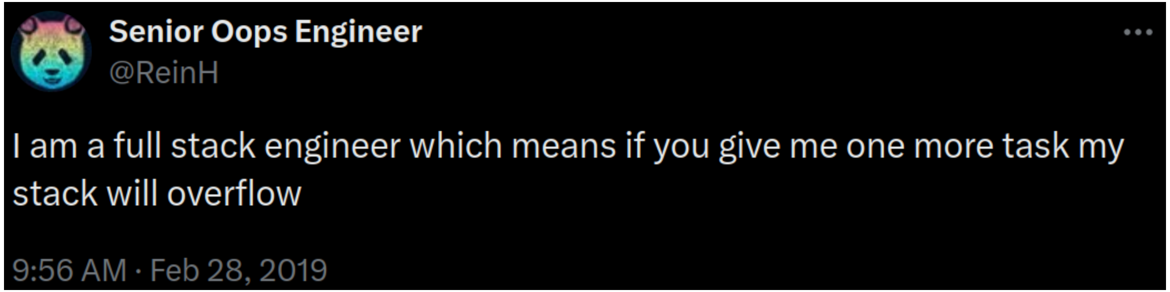
Stack & Procedures I

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang



Administrivia

- Today
  - HW8 due (11:59pm)
  - **Lab1b due** (11:59pm), late due date Friday
- Friday, 7/12
  - RD11 due (1pm)
  - HW9 due (11:59pm)
  - **Quiz 1 due** (11:59pm)
    - *No late submissions!*

Also:
Lab2 demo
in section
tomorrow!

Lab1b Reminders

- If you submit multiple times
  - Resubmit *all* necessary lab files each time
  - <u>Tag your lab partner each time</u>
- Double-check style guidelines
  - No magic numbers
  - Call previous functions instead of rewriting code
- Wait for the autograder to finish
  - You won’t see your score, but it will tell you about missing files, compilation errors, or style violations

Review Questions

1. How does the stack change after executing the following instructions?

8B

```
pushq %rbp
```
- pushes 8B of data onto the stack

```
subq $0x18, %rsp
```
— stack grows down, so subtracting from rsp grows stack

$\underbrace{0x18}_{=24}$ ↑stack ptr

8+24=32

$\boxed{\text{stack grows by 32B}}$

2. For the following function, which registers do we know must be used?

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

Review Questions (pt 2)

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

void* = pointer w/ no type

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

not a void function! has a return value!

return value %rax	%rbx	%rcx	arg3 = num %rdx	arg2 = value %rsi
%rdi arg1 = ptr	%rsp stack ptr, return address pushed during call	%r8	%r9	%rip instruction ptr, modified to point to memset code

Mechanisms Required for *Procedures*

1. Passing control
   - To beginning of procedure code
   - Back to return point
2. Passing data
   - Procedure arguments
3. Memory management
   - Allocate local variables during procedure execution
   - Deallocate on return

- All implemented with machine instructions!

```
P(...) {
  ...
  y = Q(x);
  printf(y);
  ...
}
```

local variables

control

data

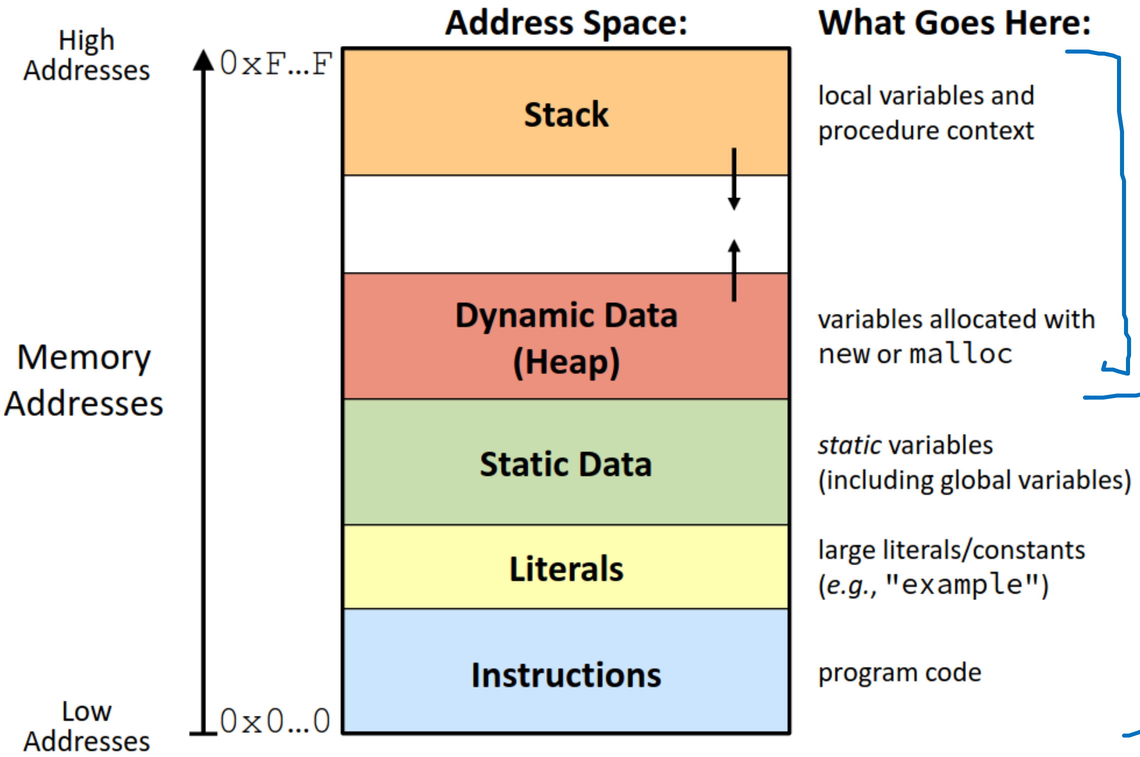
```
int Q(int i) {
  int t = 3*i;
  int v[10];
  ...
  return v[t];
}
```

local variables

Lecture Topics

- **Stack structure**
- Calling conventions
  - Passing control
  - Passing data
  - Managing local data
  - Saved registers
- Stack Frame Layout
- Register saving convention
- Illustration of Recursion

Simplified Memory Layout



writable

read-only

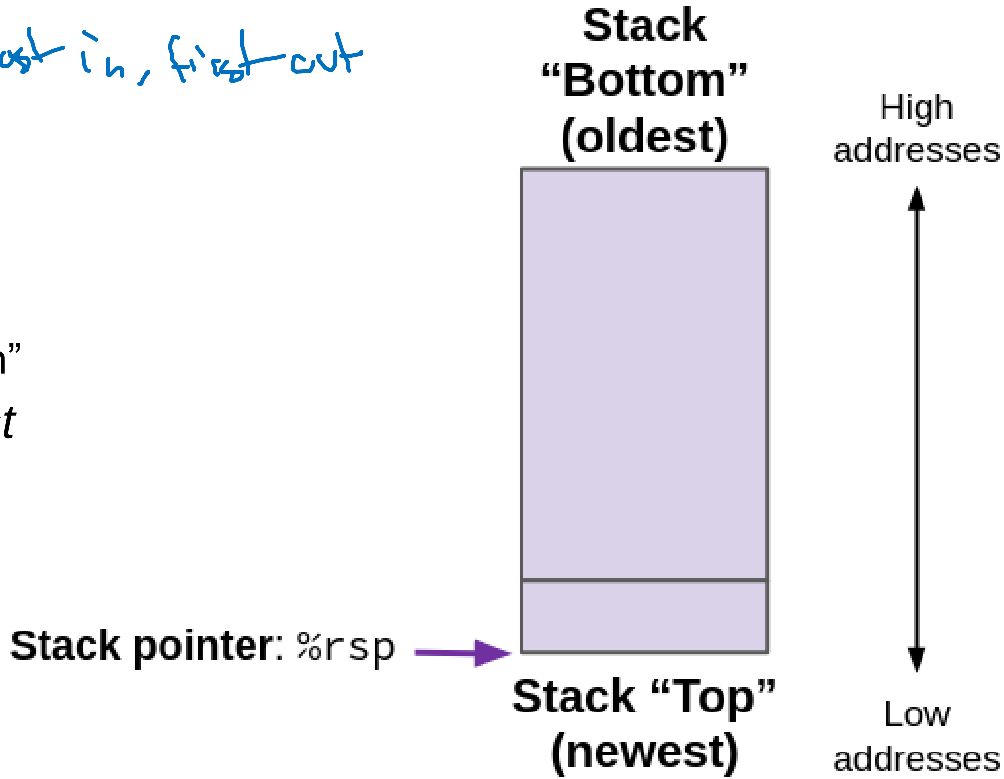
dynamically allocated

statically allocated

x86-64 Stack

stack = last in, first out

- Region of memory for managing procedures
  - Grows towards lower addresses
  - Customarily shown “upside down”
- Register `%rsp` contains the *lowest* address on the stack
  - Lowest address = most recent thing on the stack



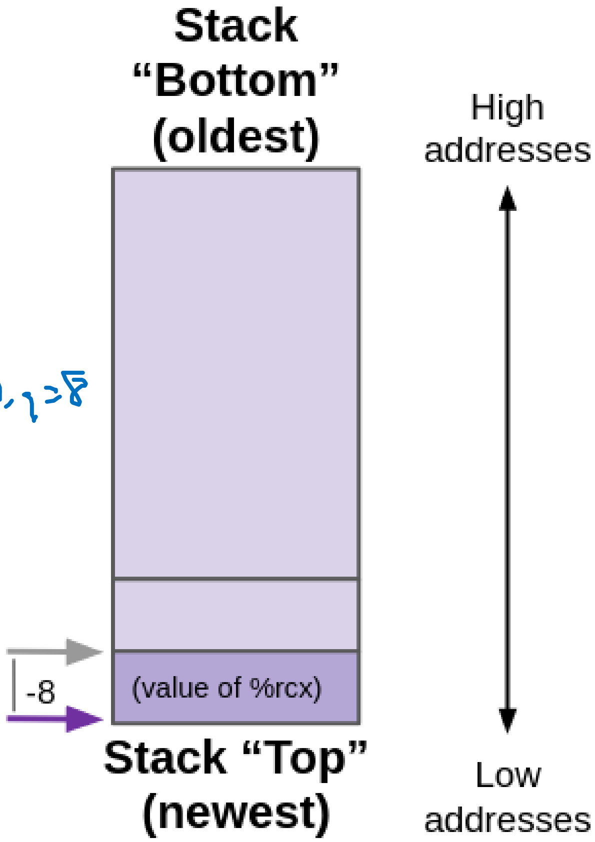
x86-64 Stack: Push

- push_ src
  - Fetch operand at src
    - Can be register, memory, or immediate
  - ***Decrement*** %rsp by size of data (depending on width specifier)

    i.e. b=1, w=2, l=4, q=8
  - Store value at address given by %rsp

Example: pushq %rcx

- Decrement %rsp by 8
- Store contents of %rcx atthat address

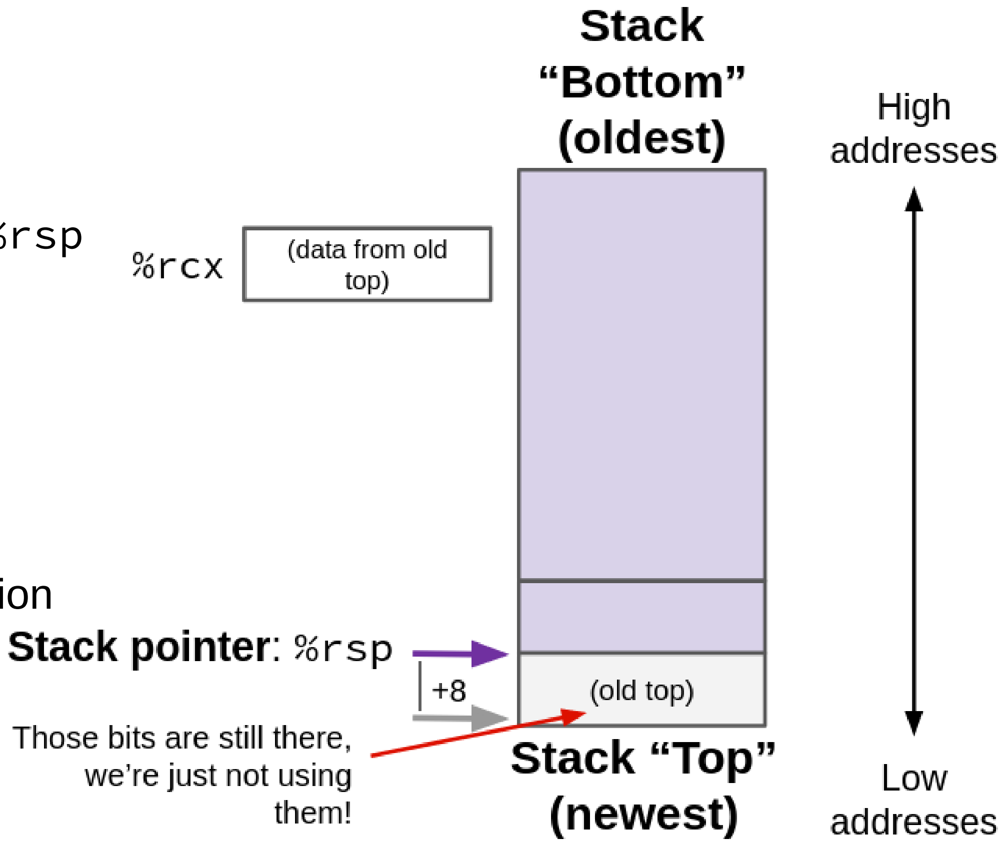


x86-64 Stack: Pop

- `pop_ dst`
  - Load value at address given by `%rsp`
  - ***Increment*** `%rsp` by size of data
  - Store value at `dst`
    - Must be a register

Example: `popq %rcx`

  - Load 8 bytes starting at the location `%rsp` points to
  - Stores into `%rcx`
  - Increment `%rsp` by 8



Lecture Topics

- Stack structure
- **Calling conventions**
  - **Passing control**
  - Passing data
  - Managing local data
  - Saved registers
- Stack Frame Layout
- Register saving convention
- Illustration of Recursion

Procedure Call Overview

Note: caller/callee relationship is for a single call

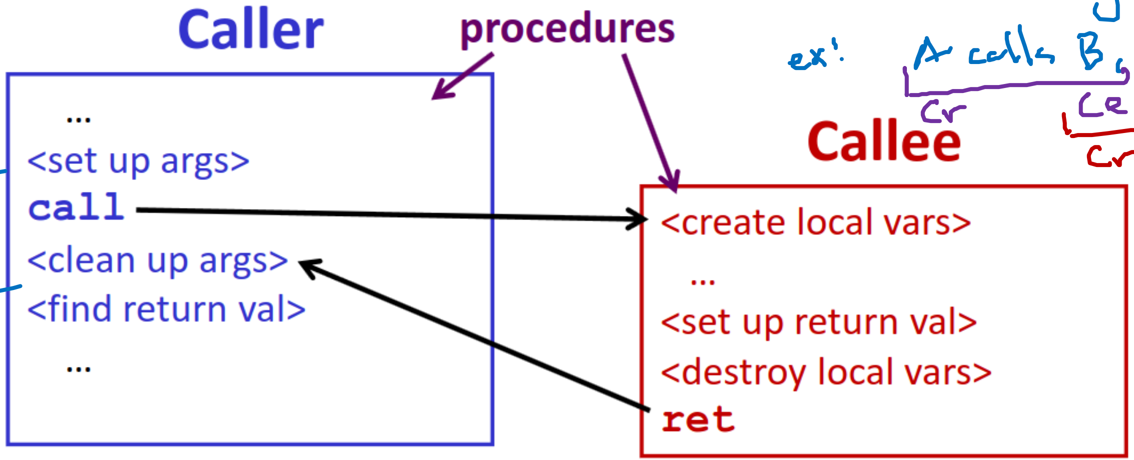
ex: A calls B, B calls C

Cr Ce

Cr Ce

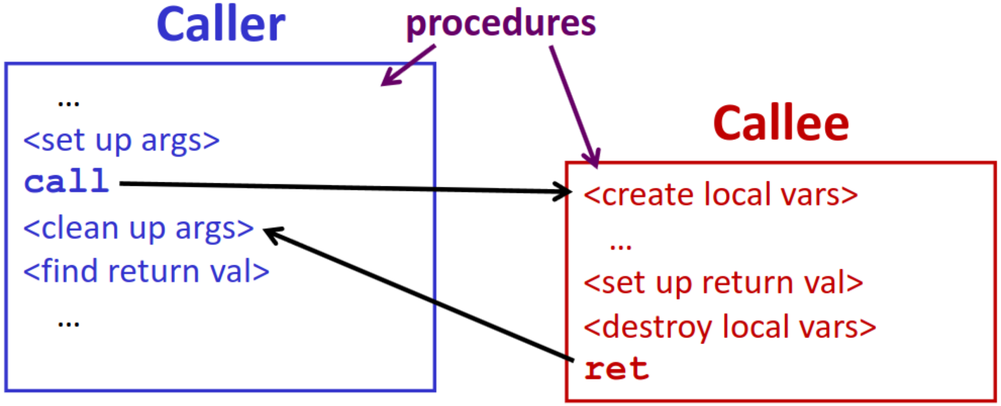
method – called on an object

function – no object



- **Caller** must know where to find **return value**
- **Callee** must know where to find arguments and **return address**
- Both run on the same CPU, so they will use the same registers
  - How do we prevent them from overwriting each other's data?

Procedure Call Overview (pt 2)



- We use the **calling convention** (also called “procedure call linkage”) to handle these problems
  - Details vary between systems
  - We will look at the convention for Linux in x86-64

Procedure Control Flow

- Use the stack to support procedure call and return
- Procedure call: `call label`
  - Push **return address** onto the stack
  - Jump to `label`
- **Return address** – tells CPU where to resume when callee finishes
  - Address of the instruction immediately *after* a procedure call

Memory addresses

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

return address = 0x400549

- Procedure return: `ret`
  - Pop return address off the stack and jump to that address

Code Example

```c
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
```

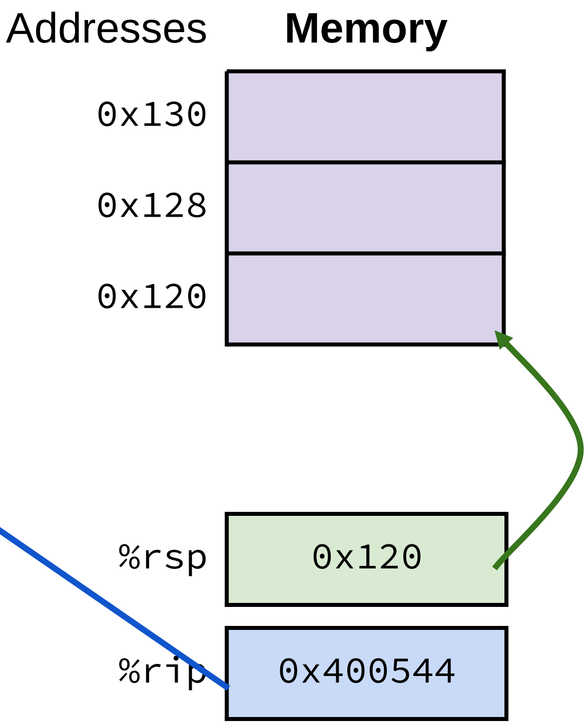
```c
long mult2 (long a, long b)
{
    long s = a * b;
    return s;
}
```

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

Procedure Call Example (before)

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

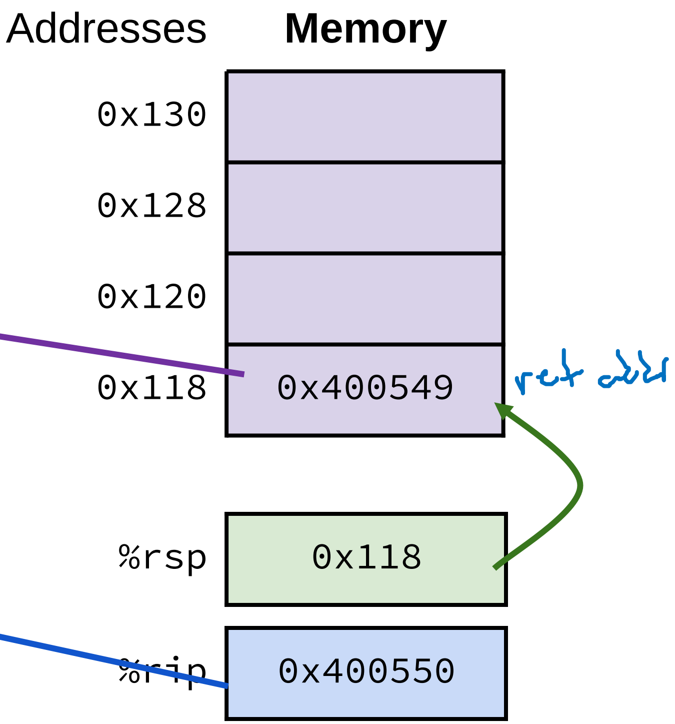
```
0000000000400550 <mult2>:
  400550: movq %rdi,%rax
  ...
  400557: ret
```



Procedure Call Example (after)

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

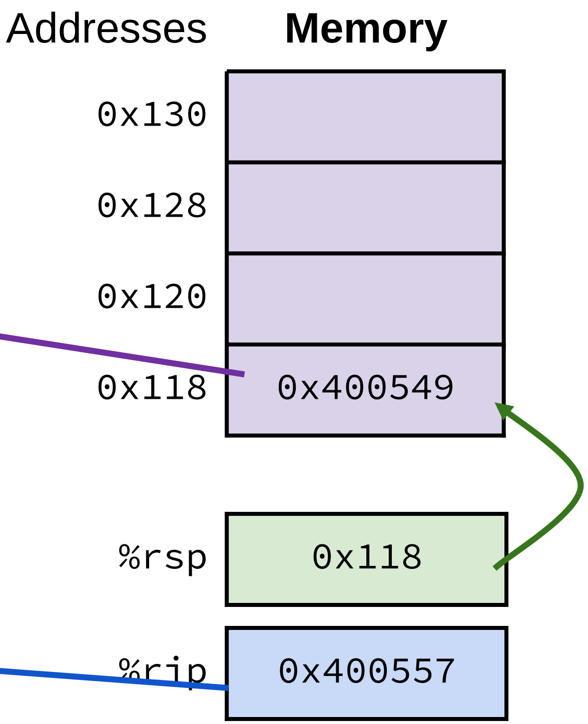
```
0000000000400550 <mult2>:
   400550: movq %rdi,%rax
   ...
   400557: ret
```



Procedure Return Example (before)

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

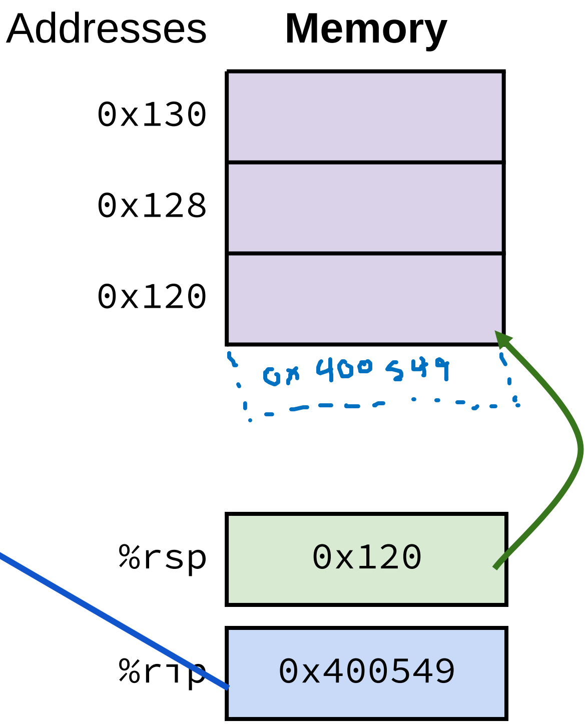
```
0000000000400550 <mult2>:
  400550: movq %rdi,%rax
  ...
  400557: ret
```



Procedure Return Example (after)

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

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



Lecture Topics

- Stack structure
- **Calling conventions**
  - Passing control
  - **Passing data**
  - **Managing local data**
  - **Saved registers**
- Stack Frame Layout
- Register saving convention
- Illustration of Recursion

Passing Arguments

caller pushes these before calling!

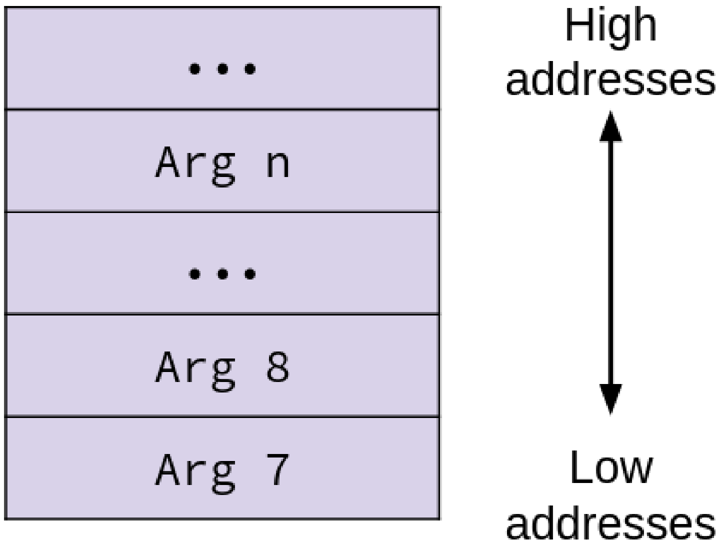
First 6 args: Registers (NOT in memory)

Register	
%rdi	Diane's
%rsi	silk
%rdx	dress
%rcx	costs
%r8	$89
%r9	

Extra args: Stack (Memory)

- Only allocate when needed

pushed in reverse order, arg7 last



Return Values

By convention, stored in `%rax`

1. **Caller** must make sure to save old contents of %rax before calling a function
    - Clears out space so callee can put the return value there
    - Part of the register saving conventions (next lecture)
2. **Callee** places return value into %rax before return
    - Any type <= 8B (pointer, integer, etc.)
    - For larger values (ex: array), returns a *pointer* to the data
3. Upon return, **caller** finds the value in %rax

Data Flow Example

doesn't need to do anything to set args bc the 1st two args to multstore are the args to mult2!

```
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
  ...
  400544: call 400550 <mult2>
  400549: movq %rax,(%rbx)
  # t in %rax
  ...
```

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

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550: movq %rdi,%rax   # %rax=a
  400553: imul %rsi,%rax   # %rax=a*b=s
  # s in %rax
  400557: ret
```

Local Data Storage

- Compiler will usually try to store local variables in **registers**
  - Faster to access than memory
- Otherwise, local data goes on the **stack**
  - Common reasons why the compiler may choose to put data in the stack:
    - No registers available
    - Data is too large (ex: arrays)
    - Variable needs to have an address (ex: C code uses the & operator)
    - Other reasons (sometimes compilers do things we don't understand!)

  registers don't have addresses!

- Programmer can't accurately predict where their data will be stored ¯_(ツ)_/¯

Register Saving

- A function can't predict which other registers other functions may use
  - What if it overwrites a register that its caller was using?
  - What if it calls a function that overwrites its register values?

<u>Example</u>: what's wrong with this code?

rbx = ~~351~~ 123

```
foo:
    movq $351, %rbx   # rbx = 351
    ...
    call bar
    ...
    addq %rax, %rbx   # rbx = rax + 351
    ret
```

wrong value!! will be 123

```
bar:
    ...
    movq $123, %rbx   # rbx = 123
    ...
    ret
```

Register Saving (pt 2)

- If we want to write to a register, save its old value onto the stack to avoid losing data
  - Use **push** instruction to save register
  - Use **pop** to restore the register back to its old value
- **Register saving convention** dictates when registers are saved
  - Next lecture!

Register Saving Example

Now bar is free to use %rbx without messing with foo's data :)

```
foo:
 movq $351, %rbx   # rbx = 351
    ...
    call bar
    ...
    addq %rax, %rbx  # rbx = rax + 351
    ret
```

```
bar:
    ...
    pushq %rbx          # save old rbx
    movq $123, %rbx    # rbx = 123
    ...
    popq %rbx           # restore rbx
    ret
```

Lecture Topics

- Stack structure
- Calling conventions
  - Passing control
  - Passing data
  - Managing local data
  - Saved registers
- **Stack Frame Layout**
- Register saving convention
- Illustration of Recursion

Stack-Based Languages

- *e.g.*, C, Java, most modern languages
- Support recursion
  - Code must be *re-entrant*
    - Allow multiple simultaneous instances of the same procedure
- Stack allocated in **frames**
  - State for a single instance of a procedure
- Stack “discipline”
  - Maintained by the compiler
  - State for a given procedure is only needed for a limited time
    - Starting from when it is called to when it returns
  - Callee always returns before caller does

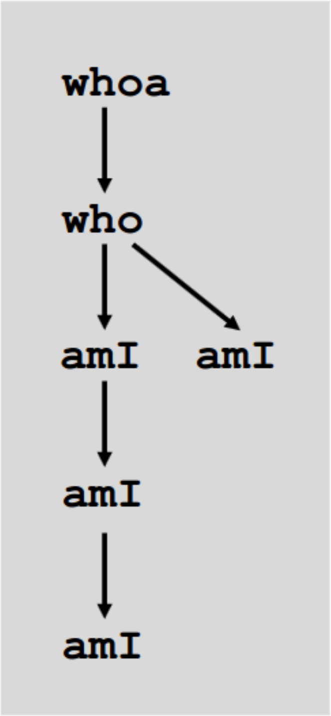
Call Chain Example

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

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

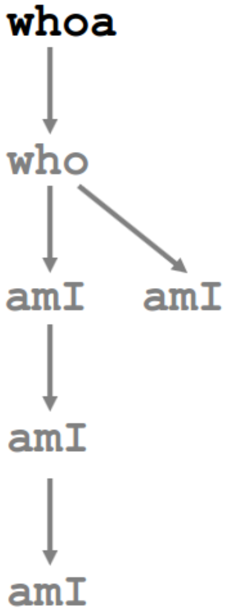
```
amI(...) {
  •
  •
  if(...)
    amI(...);
  •
  •
}
```

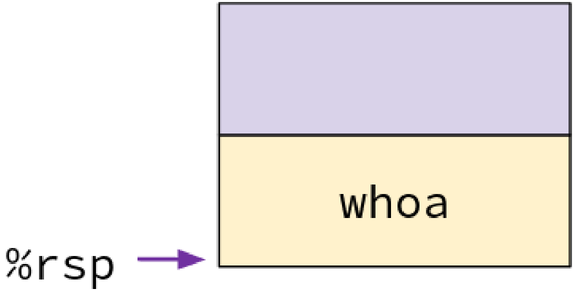
Example
Call Chain



1. Call to `whoa`

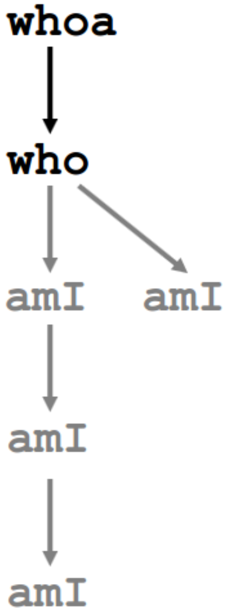
```
whoa(...) {
   •
   •
  who(...);
   •
   •
}
```

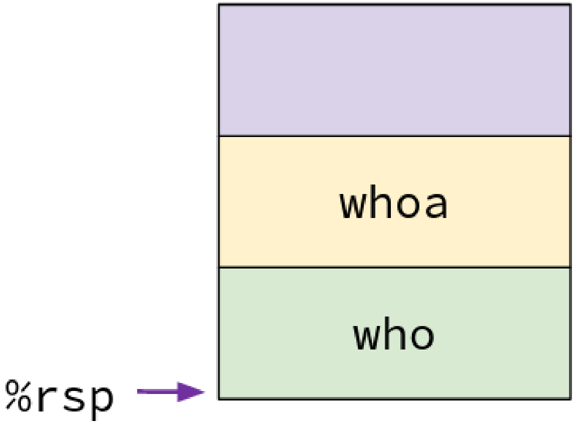




2. Call to who

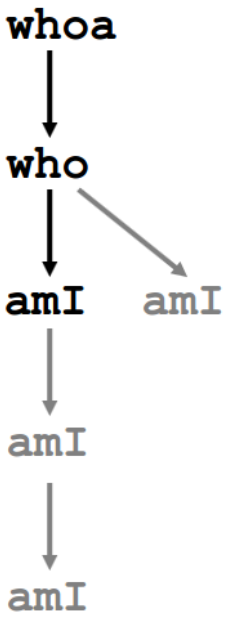
```
who(...) {
   •
   amI(...);
   •
   amI(...);
   •
}
```

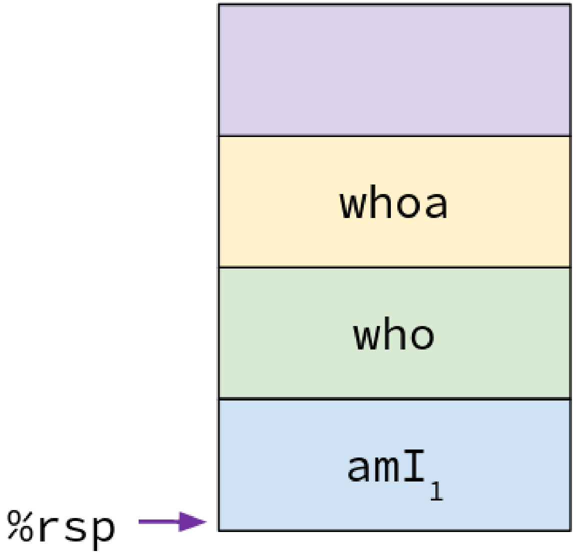




3. Call to `ami`

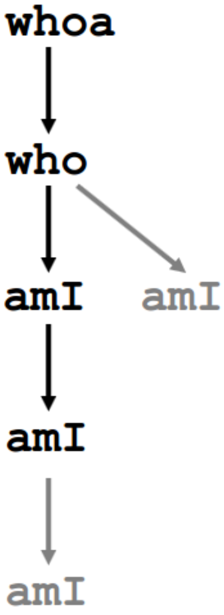
```
amI(...) {
  •
  •
  if(...)
    amI(...);
  •
  •
}
```

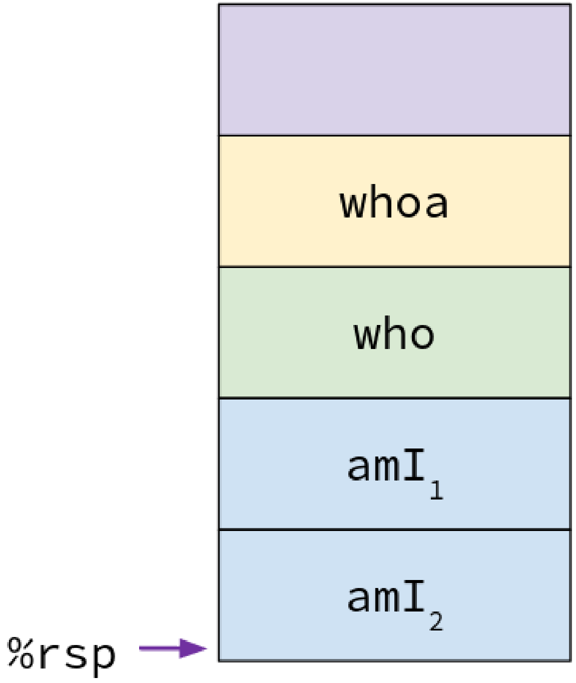




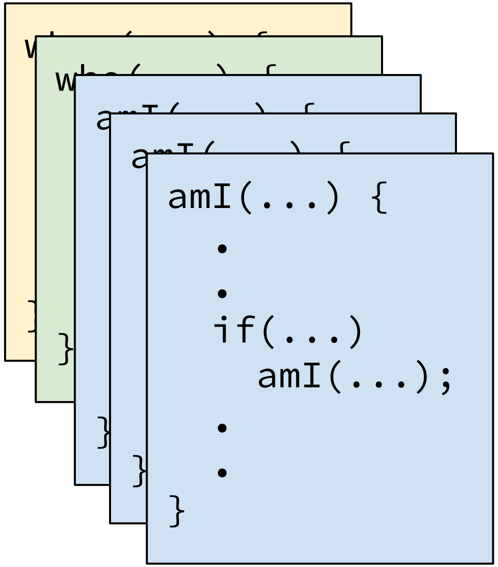
4. Recursive Call to `ami`

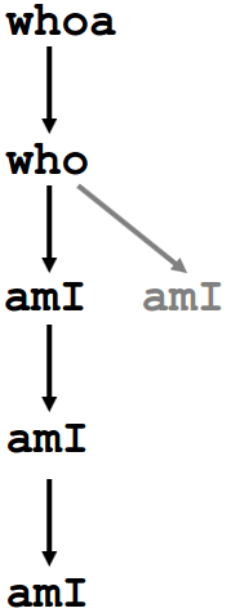
```
amI(...) {
  •
  •
  if(...)
    amI(...);
  •
  •
}
```

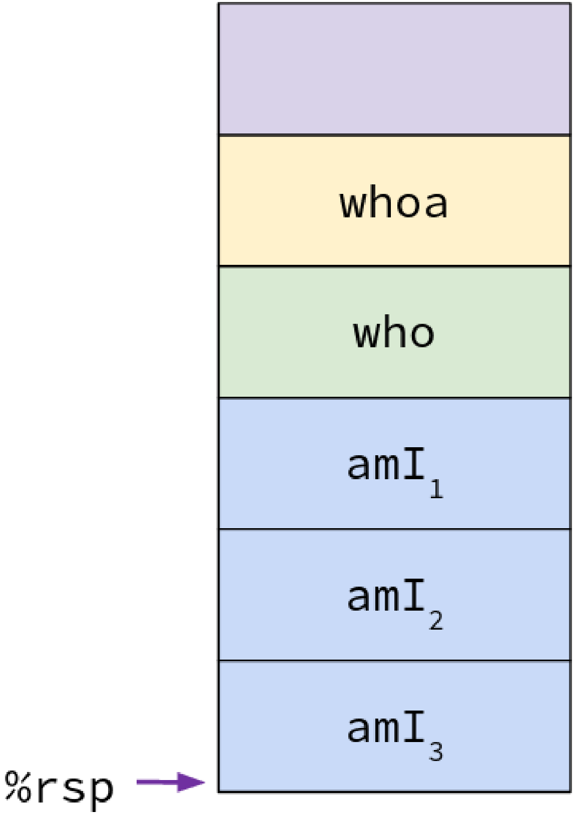




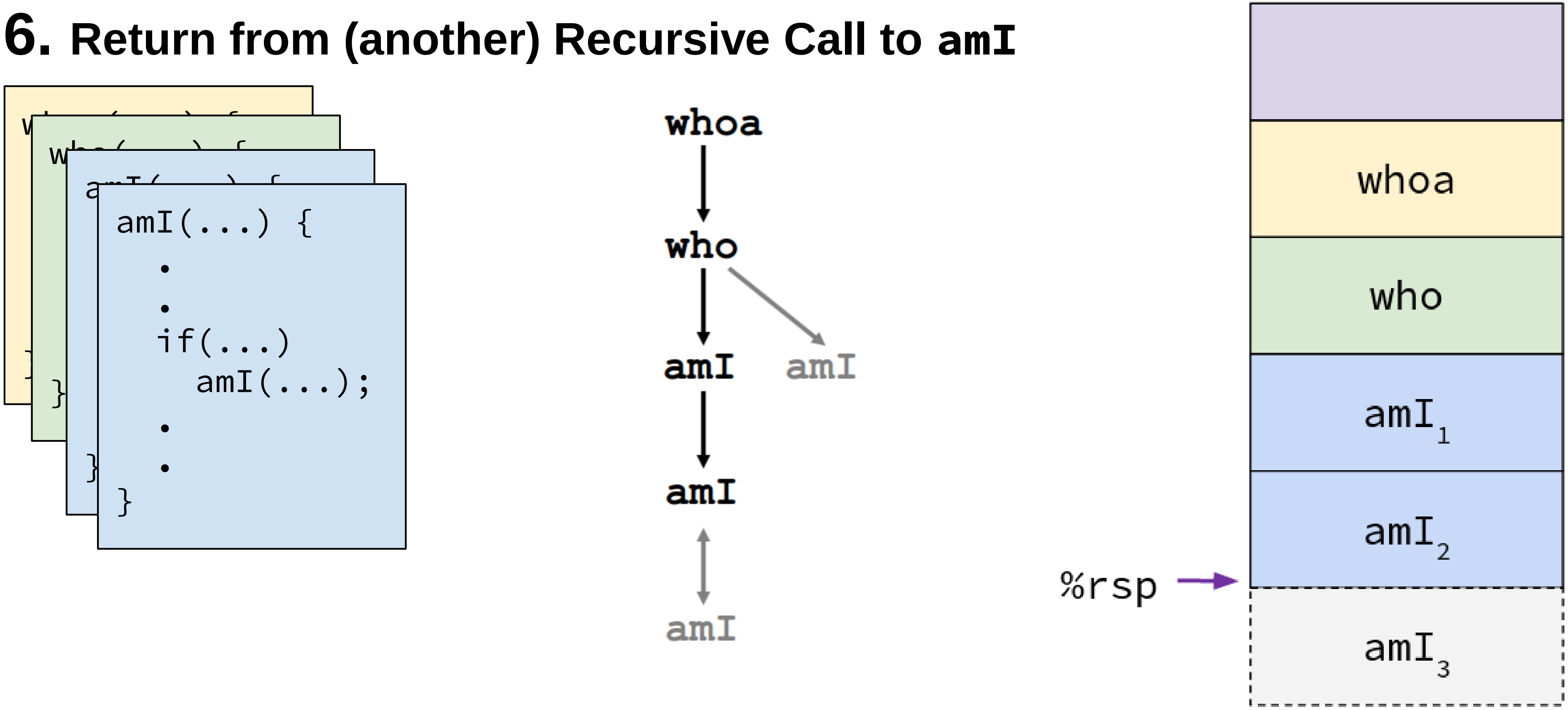
5. (another) Recursive Call to amI





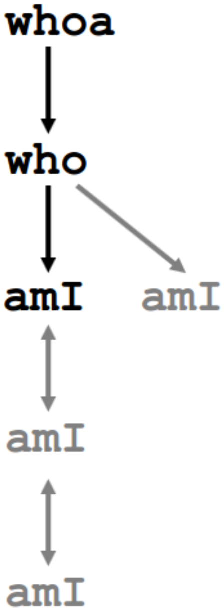


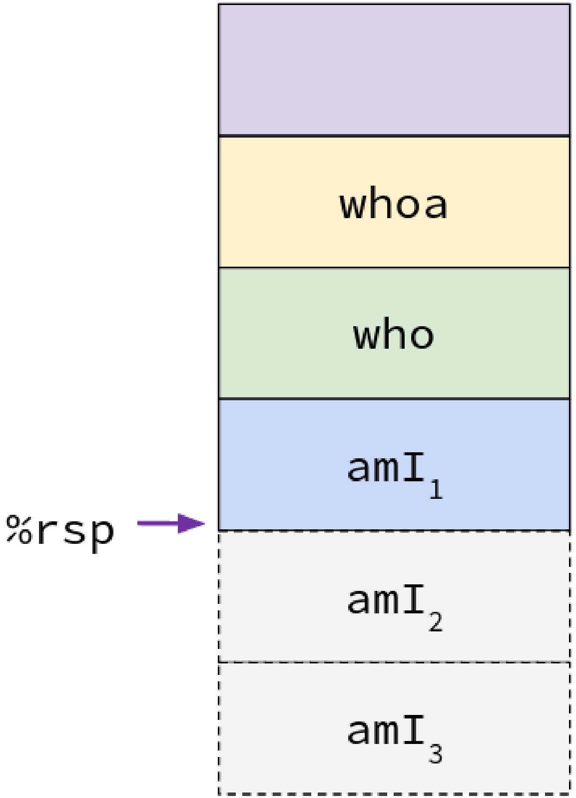
6. Return from (another) Recursive Call to `amI`



7. Return from Recursive Call to `amI`

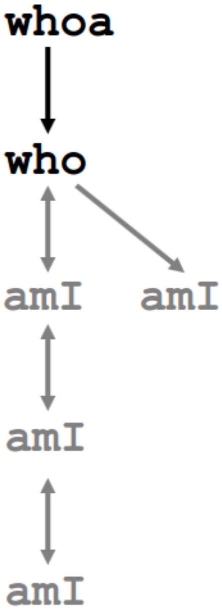
```
amI(...) {
   •
   •
   if(...)
     amI(...);
   •
   •
}
```

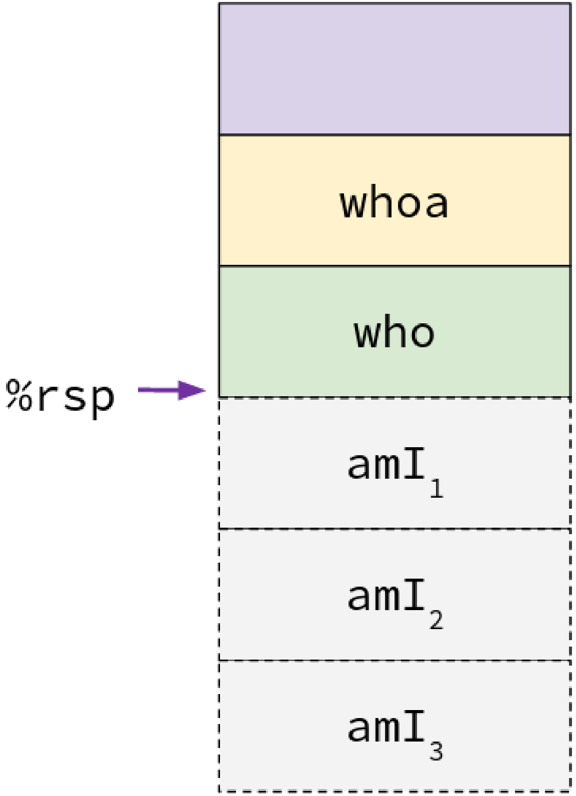




8. Return from Call to amI

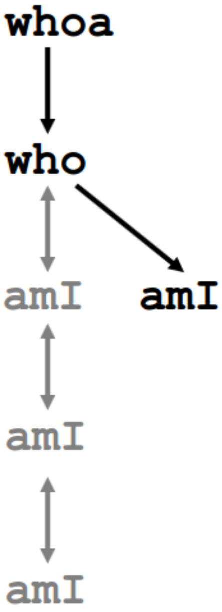
```
who(...) {
   •
   amI(...);
   •
   amI(...);
   •
}
```

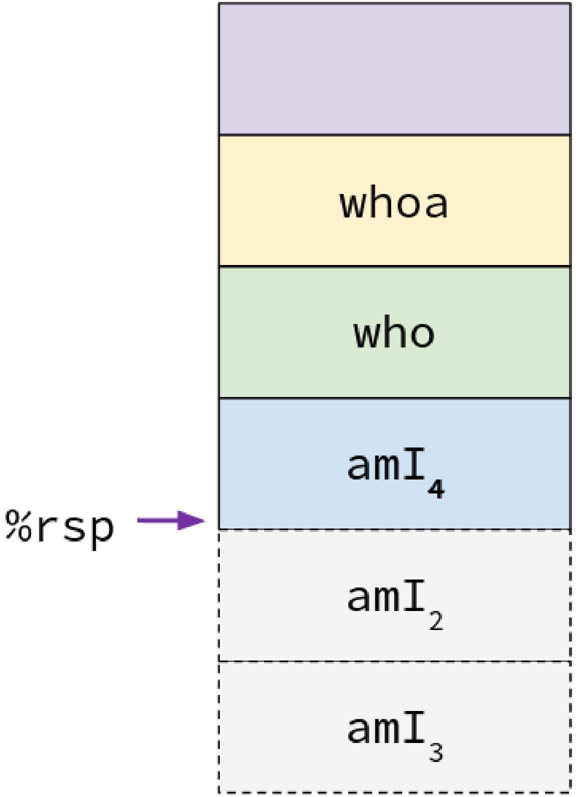




9. (yet another) Call to amI

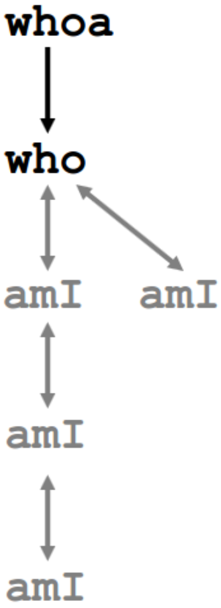
```
amI(...) {
  •
  •
  if(...)
    amI(...);
  •
  •
}
```

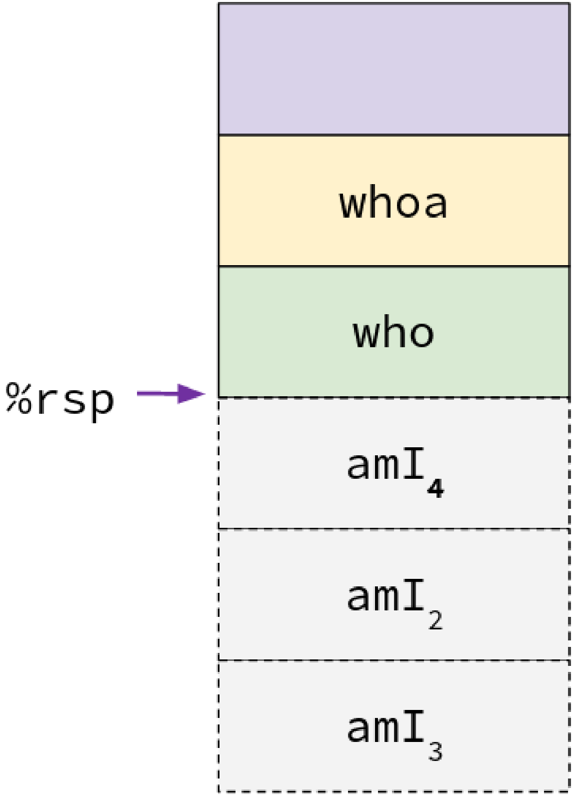




10. Return from (yet another) Call to amI

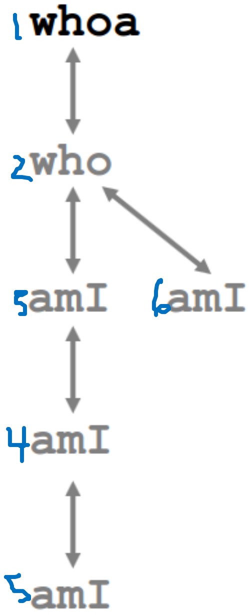
```
who(...) {
   •
   amI(...);
   •
   amI(...);
   •
}
```

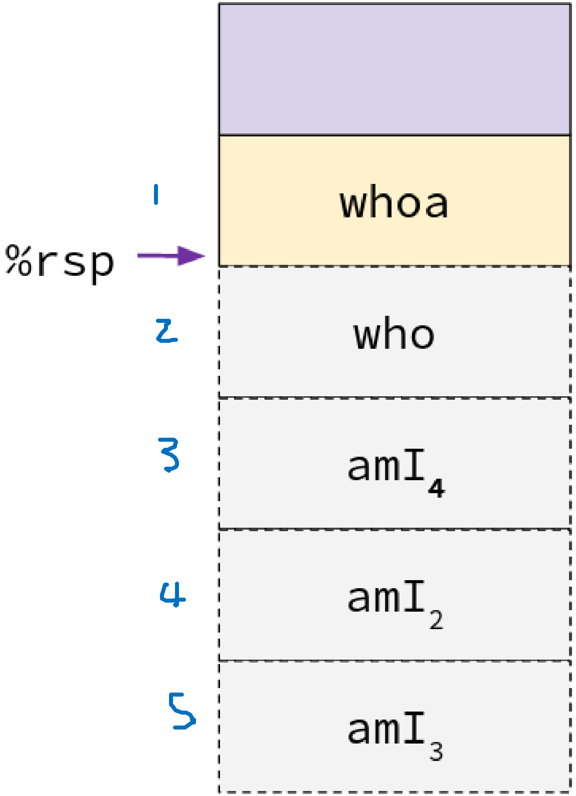




11. Return from Call to who

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





How many stack frames? 6

Max stack depth? 5

Stack Frame Layout

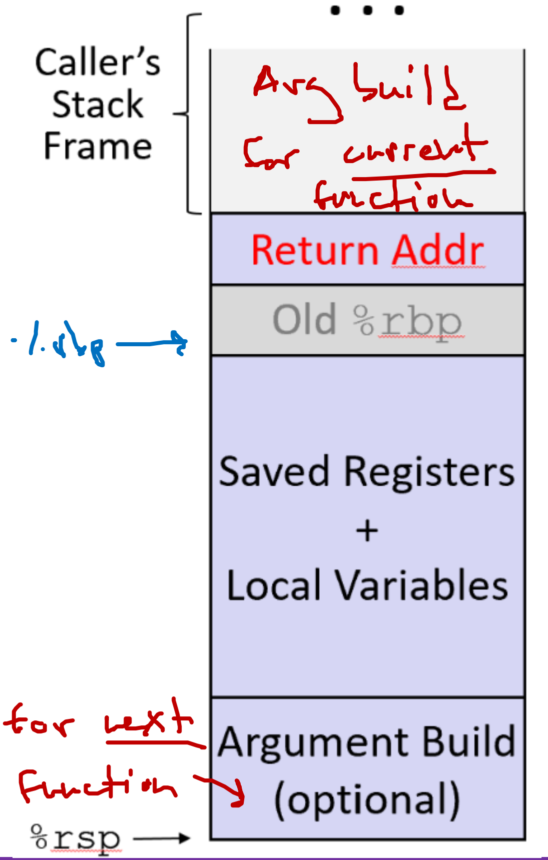
- Return address to marks the beginning of a stack frame
    - This is an arbitrary choice!
- "Old `%rbp`" no longer used in modern compilers
- Argument build = args 7+ for the *next* procedure
    - Pushed onto the stack right before `call`

rbp = base pointer, points to beginning of frame

pushq %/.rbp   save old %/. rbp

movq %/.rsp, rbp   %/.rbp points to old value

outdated, but used in lab 2+3



Stack Overflow

- When the the size of the stack grows too large
  - `%rsp` points to something it’s not supposed to, segmentation fault
  - In theory, happens when stack collides with heap
  - In practice, Linux limits stack to 8 MiB
- Aside: Stack Overflow website was named by popular vote from users. Some of the non-winning options:
  - bitoriented
  - dereferenced
  - privatevoid
  - shiftleft1
  - understandrecursion

Discussion: Stack Conventions

- The stack is maintained between function calls by *conventions*, meaning you could (in theory) write code that doesn't follow them.
  - **Argument passing** convention: 1st 6 args go in specific registers, then the stack
  - **Return value** convention: `%rax`
  - **Register saving** convention: next lecture
- What could happen if your program didn't follow these conventions?

your program would work on its own, but would break when it interacts with other code!

Summary

- The **stack** is a region of memory that stores local data for **procedures**
  - Allocated in **frames**
  - Grows *down*. **Stack Pointer** (`%rsp`) points to the end of the stack
- When a procedure is called, **return address** is pushed onto the stack
  - Popped off again on return
- We use **procedure call convention** to pass data between procedures
  - 1st 6 args in registers (remember with **Di**ane's **si**lk **d**ress **c**osts $**89**)
  - Remaining args on the stack
  - Return value in %rax
- When writing to a register, save its old value on the stack to prevent data loss