Procedures II

CSE 351 Winter 2022

Instructor:
Sam Wolfson

Teaching Assistants:
Angela Xu
Anirudh Kumar
Catherine Guevara
Dara Stotland
Harrison Bay
Ian Hsiao
Kevin Wang
Mara Kirdani-Ryan
Nick Durand
Sanjana Sridhar

http://xkcd.com/1270/

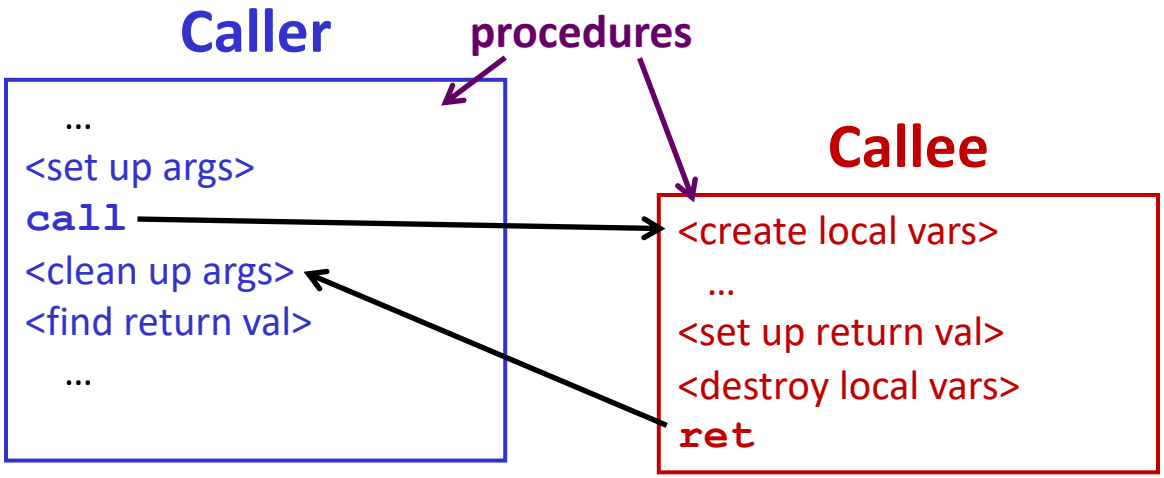
Relevant Course Information

- Return to in-person instruction on Monday!
  - Masks are required **at all times** during lecture, section, and office hours – brief sips of drinks are OK.
  - If you are unwell, please **stay home** – we have designed the course so you will not be penalized for helping to keep our community safe. See our COVID Safety page.
  - If we need to cancel or change the modality of a lecture, section, or office hour, we'll let you know as early as possible.
- Midterm (take home, 2/8—2/11)
  - Make notes and use the midterm reference sheet
  - Form study groups and look at past exams!

Procedures

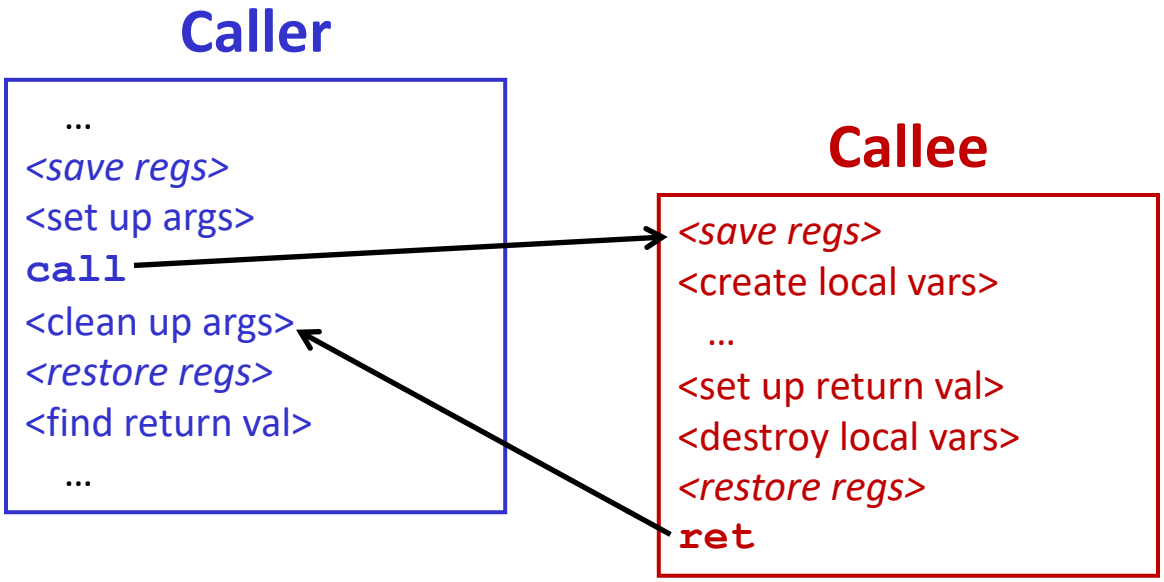
- Stack Structure
- **Calling Conventions**
  - **Passing control**
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

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;
}
```

Compiler Explorer:
https://godbolt.org/z/ndro9E

```
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
```

```
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
```

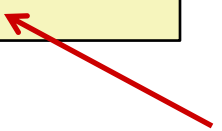
Procedure Control Flow (Review)

- 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**

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

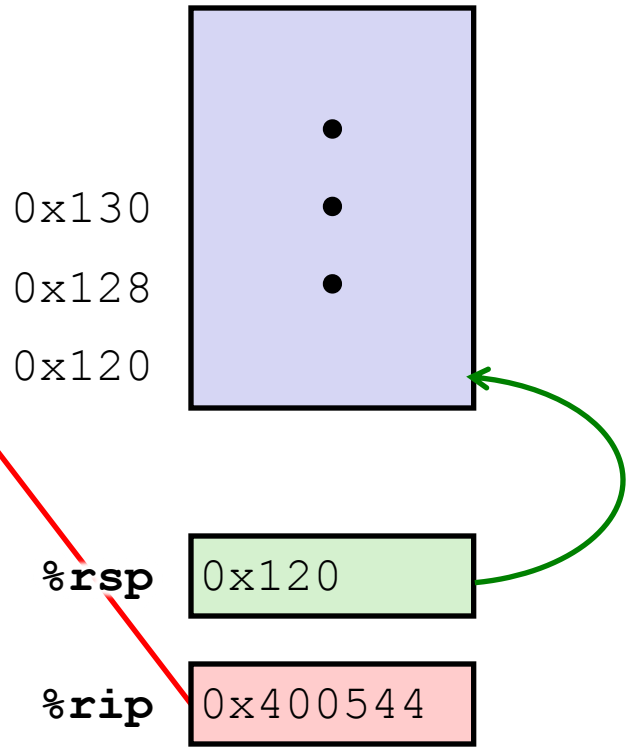


next instruction happens to be a move, but could be anything

Procedure Call Example (step 1)

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

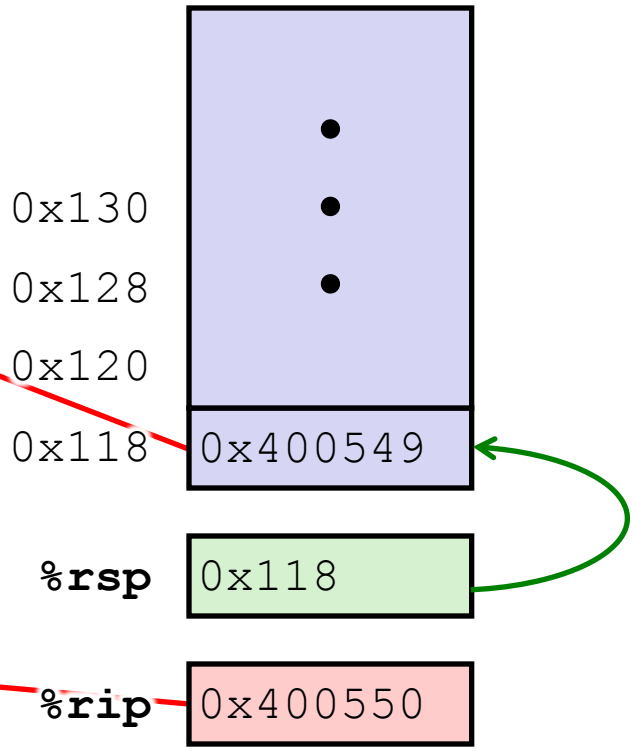
```
0000000000400550 <mult2>:
   400550:  movq   %rdi,%rax
   •
   •
   400557:  ret
```



Procedure Call Example (step 2)

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

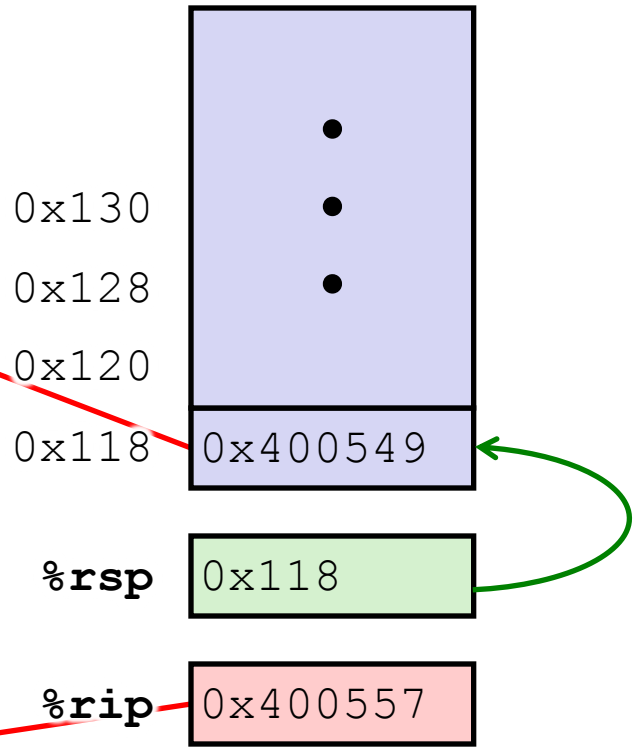
```
0000000000400550 <mult2>:
   400550:  movq   %rdi,%rax
   •
   •
   400557:  ret
```



Procedure Return Example (step 1)

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

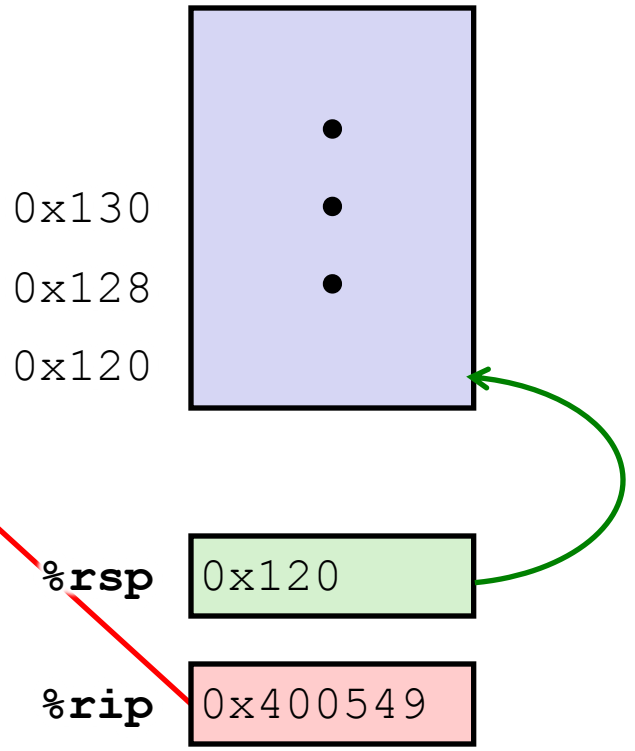
```
0000000000400550 <mult2>:
   400550:  movq   %rdi,%rax
   •
   •
   400557:  ret
```



Procedure Return Example (step 2)

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

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



Procedures

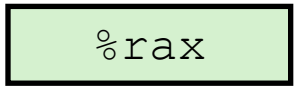
- Stack Structure
- **Calling Conventions**
  - Passing control
  - **Passing data**
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Procedure Data Flow (Review)

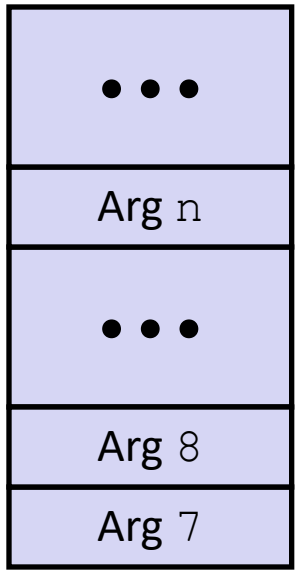
Registers (NOT in Memory)

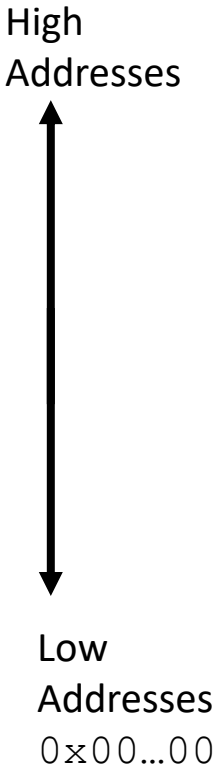
- First 6 arguments

- Return value



Stack (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 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

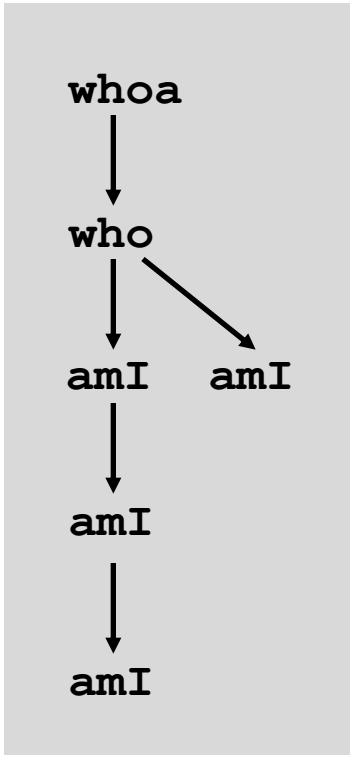
Call Chain Example

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

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

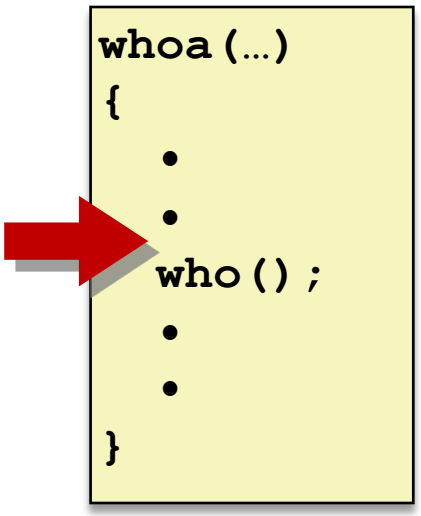
```
amI(…)
{
    •
  if(…){
    amI()
  }
    •
}
```

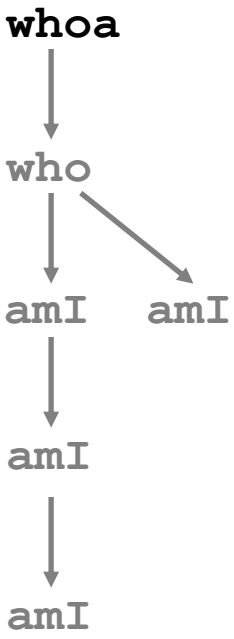
Example
Call Chain

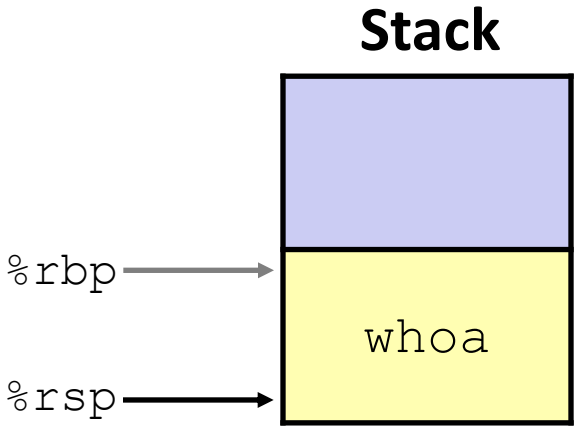


Procedure `amI` is recursive
(calls itself)

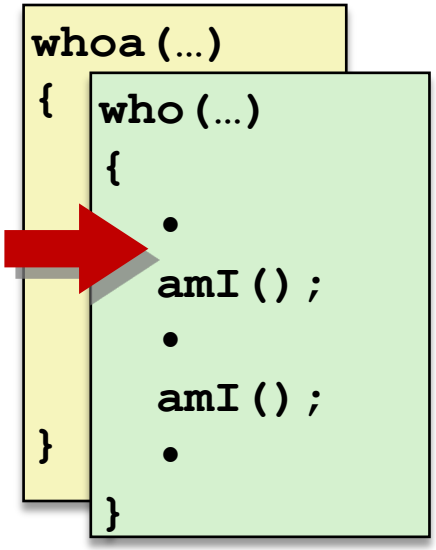
1) Call to whoa

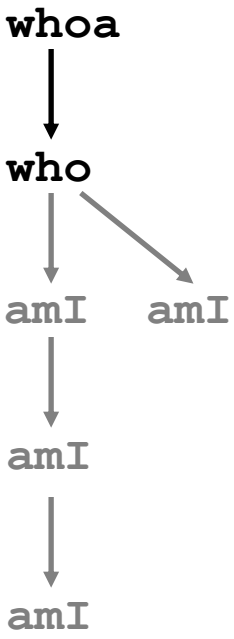


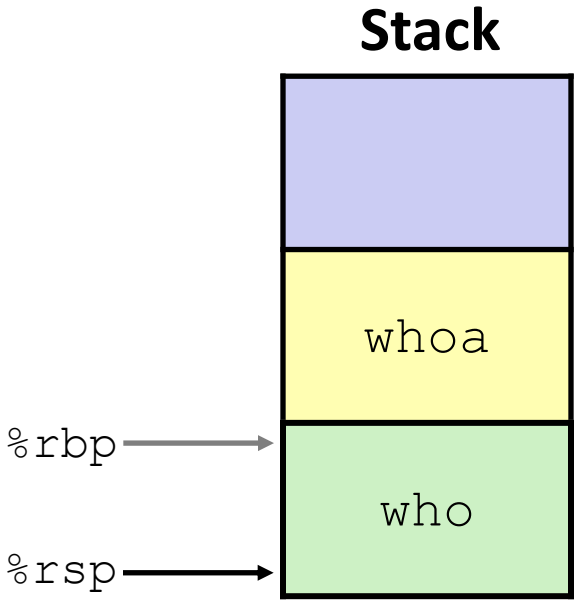




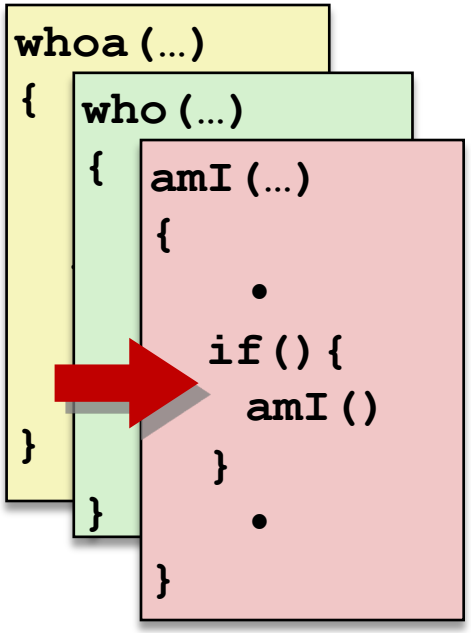
2) Call to who

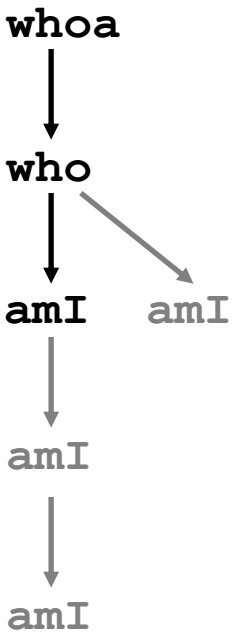

whoa(...)
{
who(...)
{
•
amI();
•
amI();
•
}
}

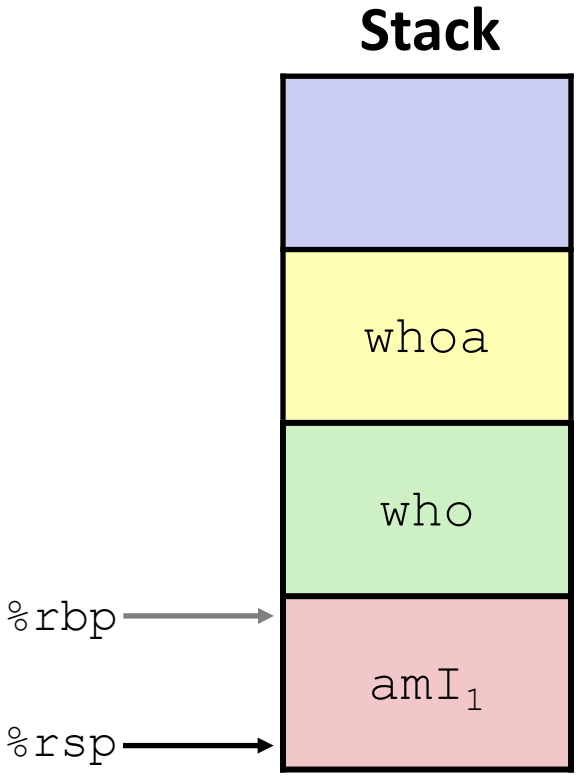

whoa
who
amI
amI
amI
amI


Stack
whoa
who
%rbp
%rsp

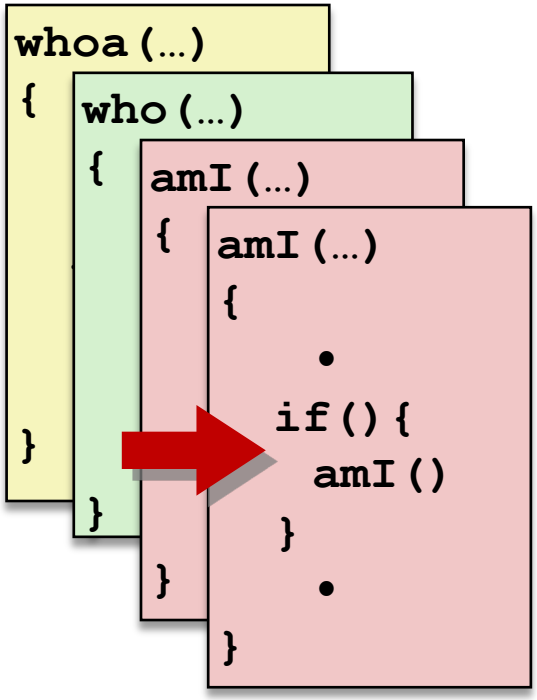
3) Call to amI (1)

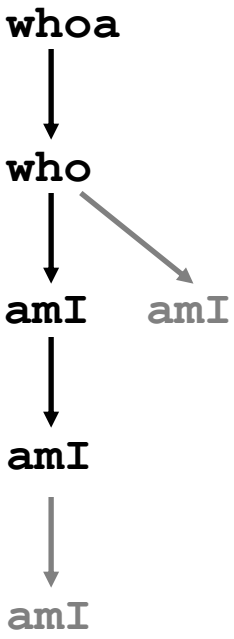


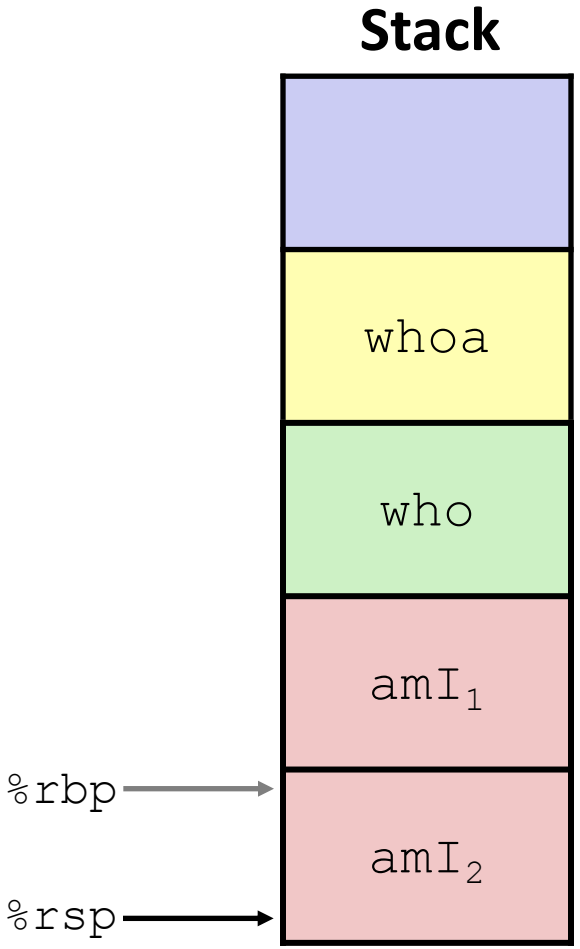




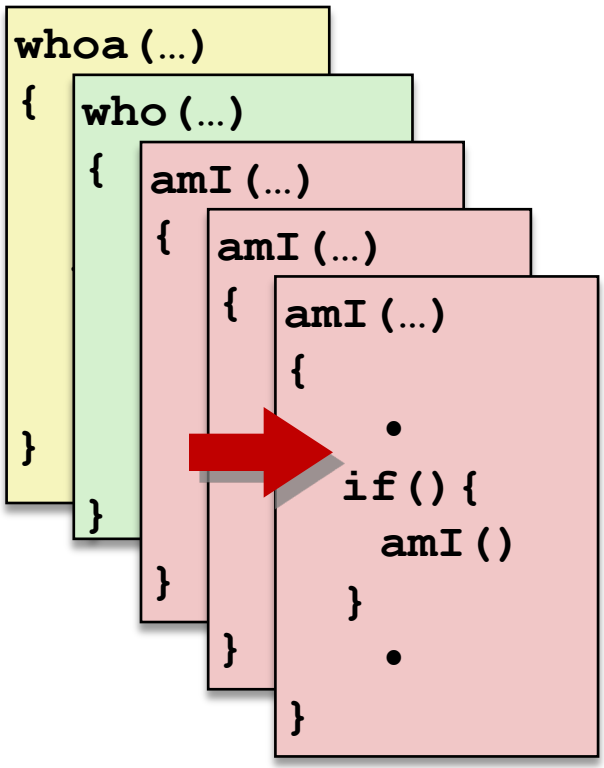
4) Recursive call to amI (2)

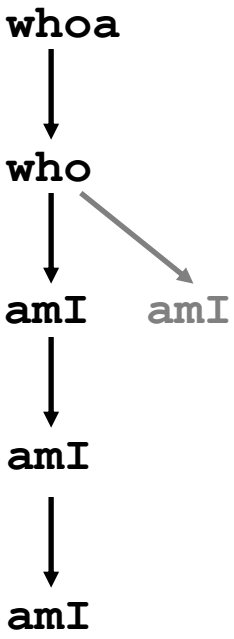


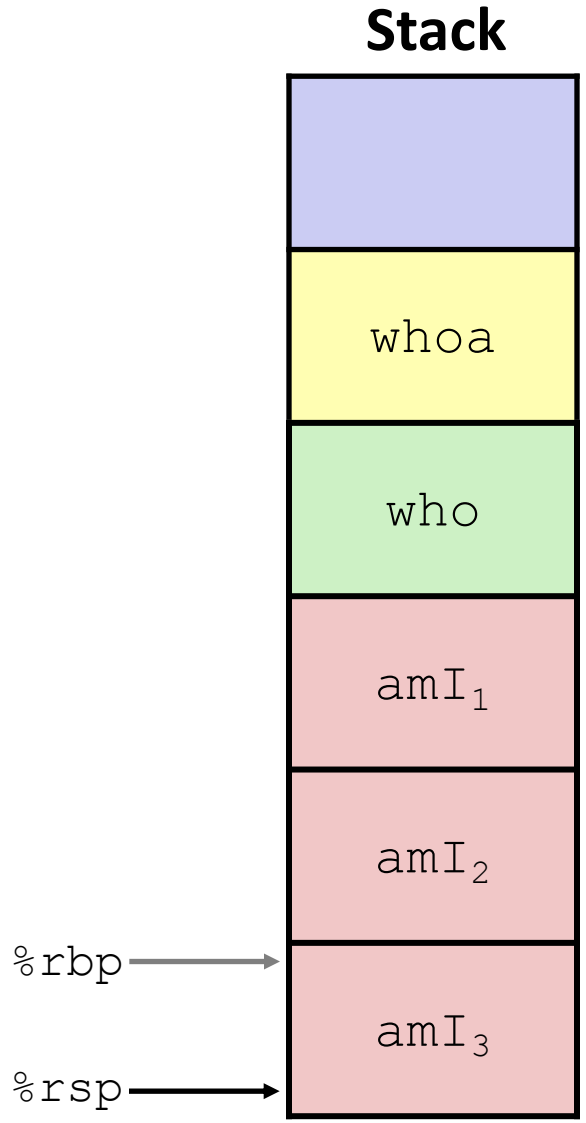




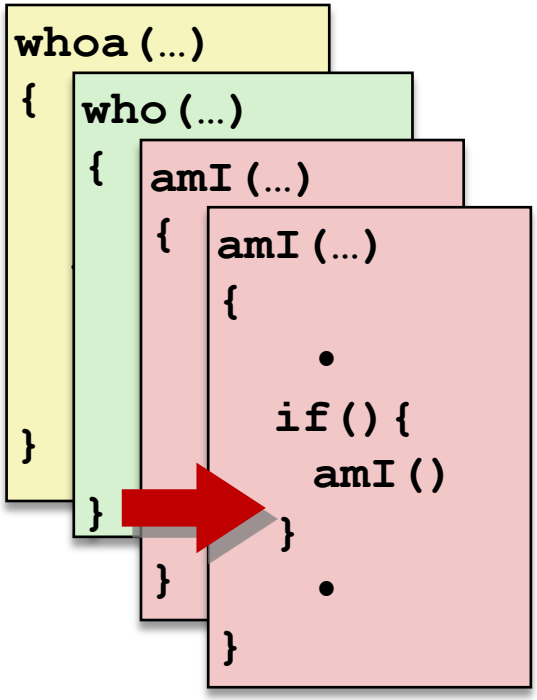
5) (another) Recursive call to amI (3)

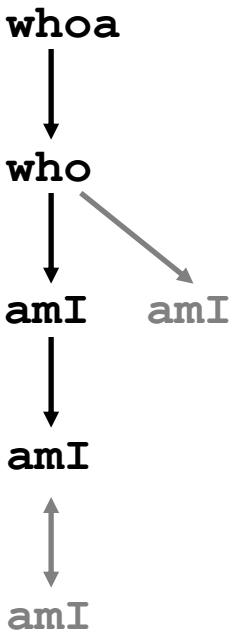


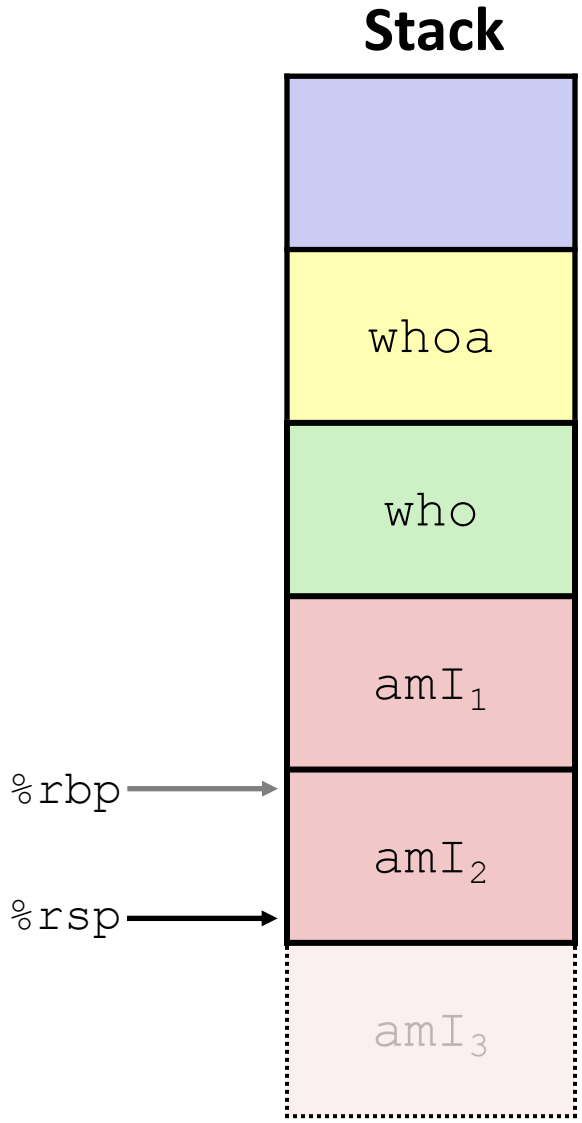




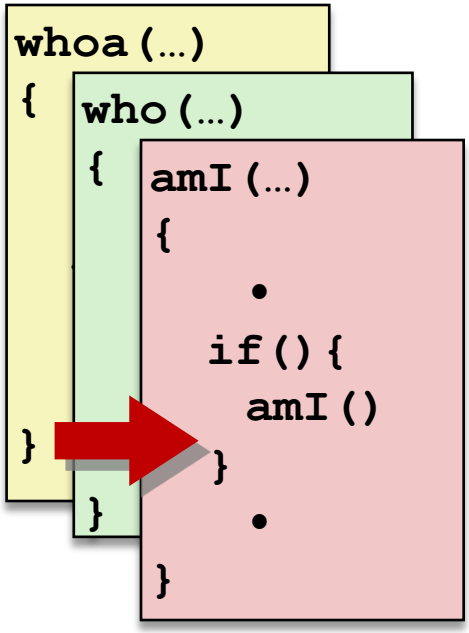
6) Return from (another) recursive call to amI

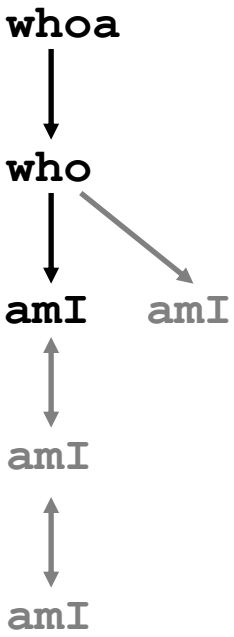


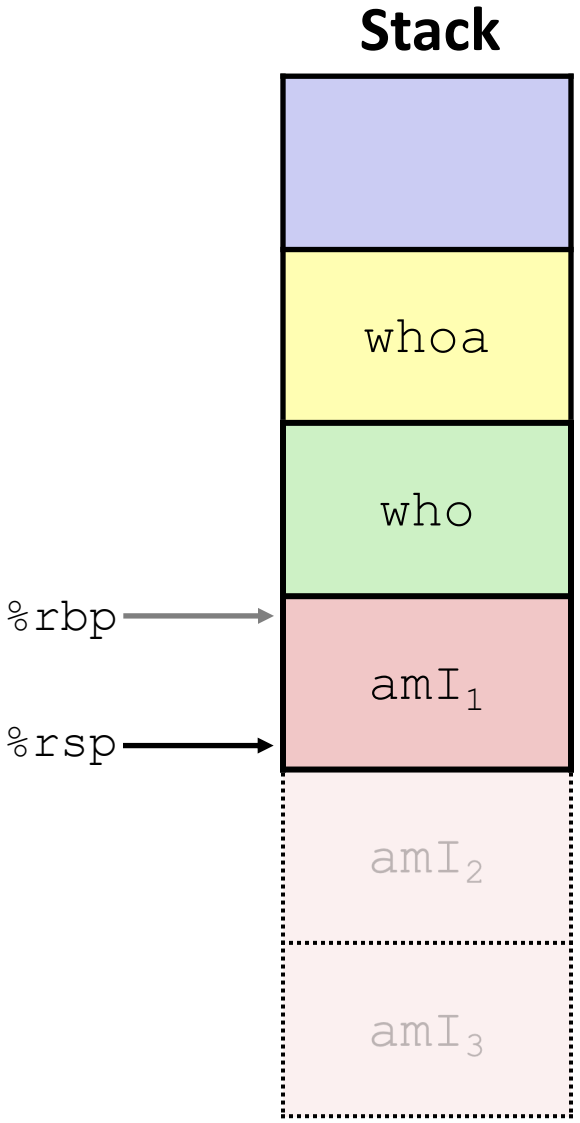




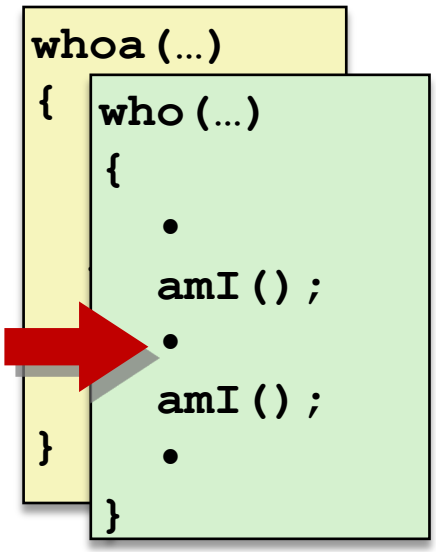
7) Return from recursive call to amI

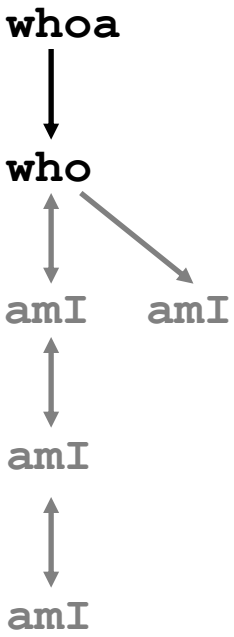


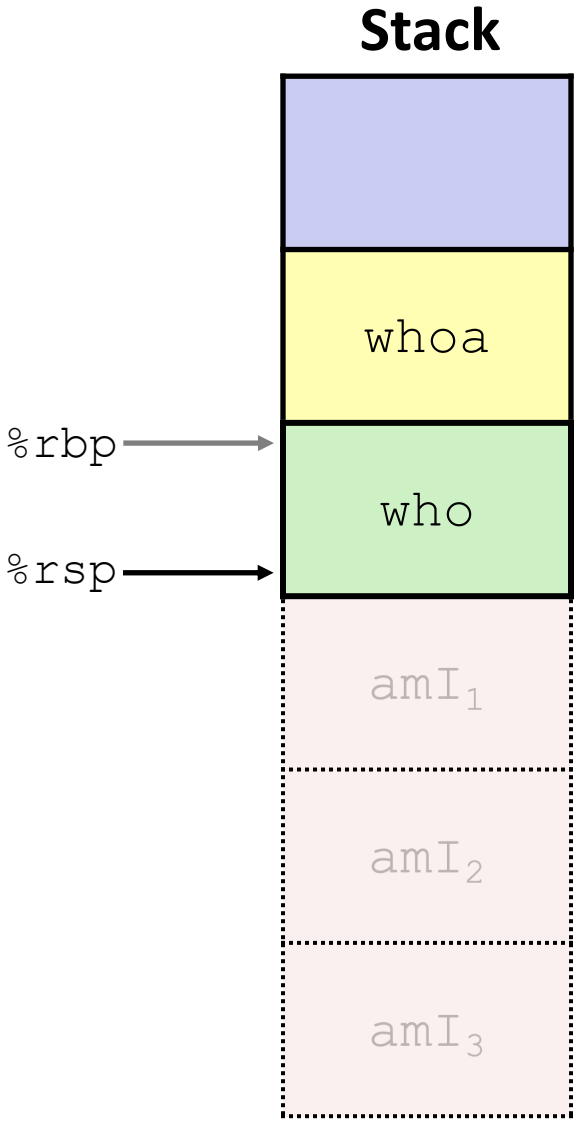




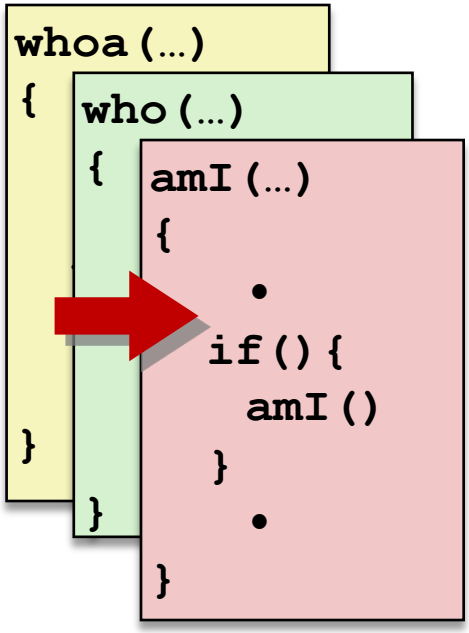
8) Return from call to amI

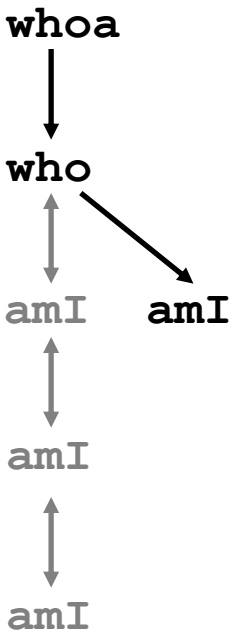


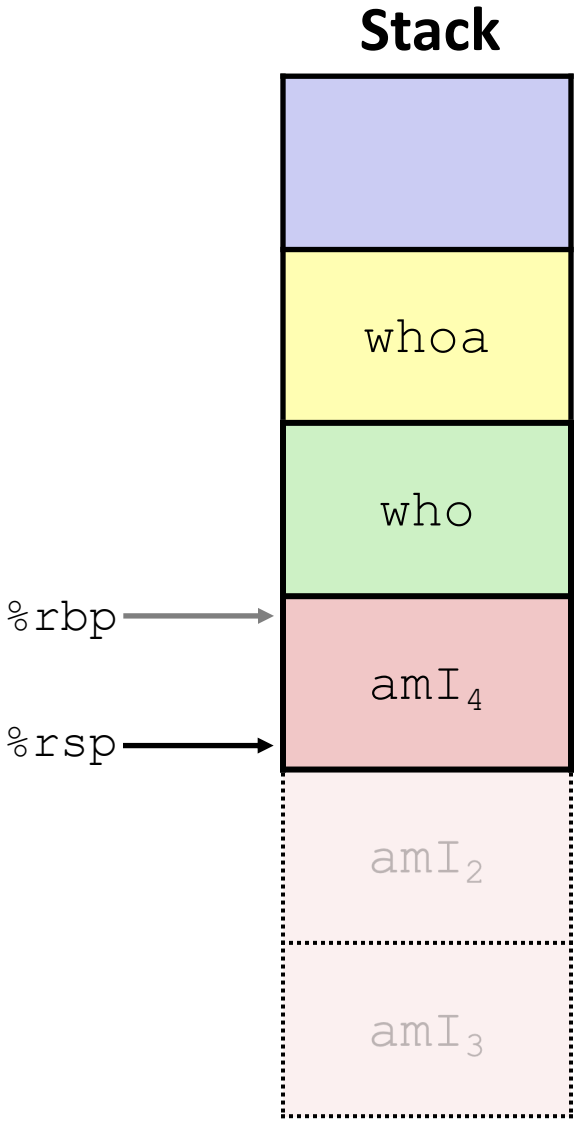




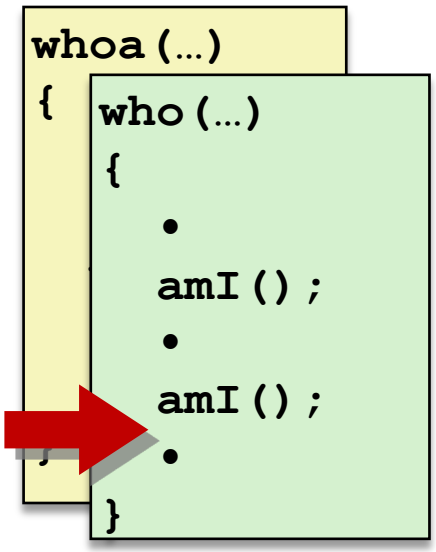
9) (second) Call to amI (4)

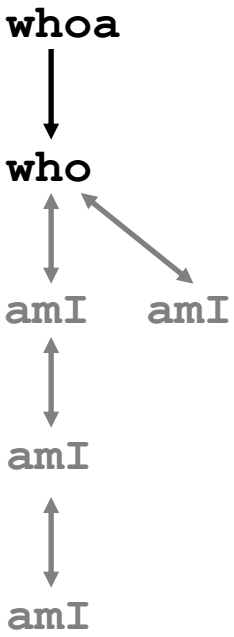


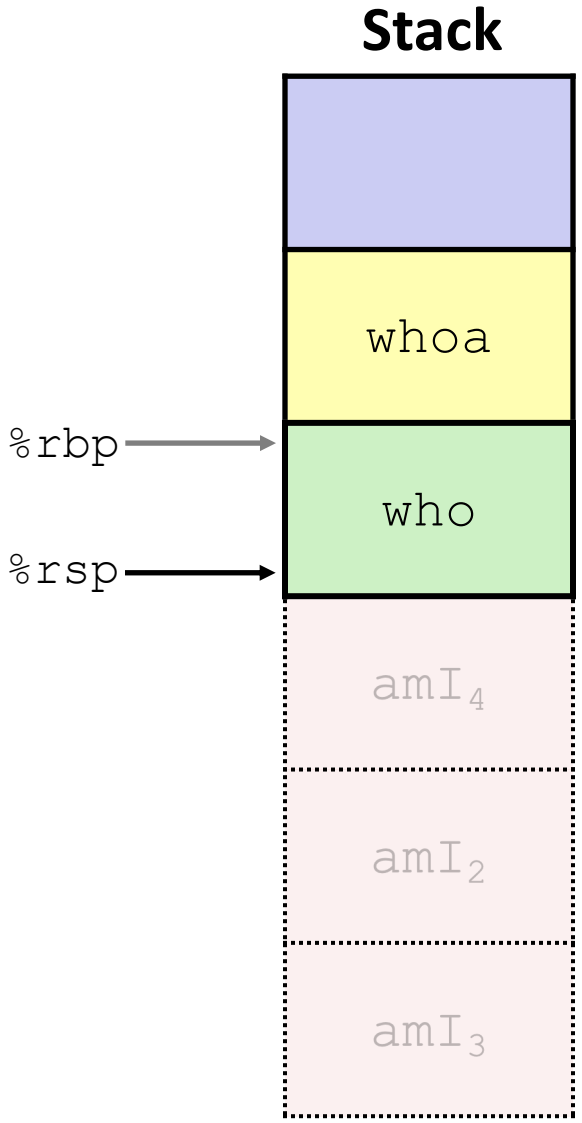




10) Return from (second) call to amI

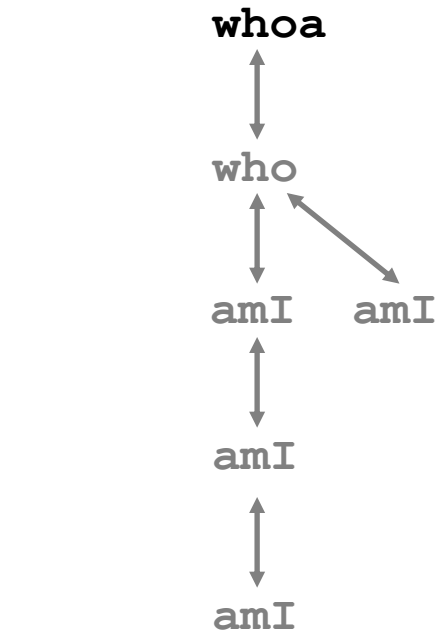


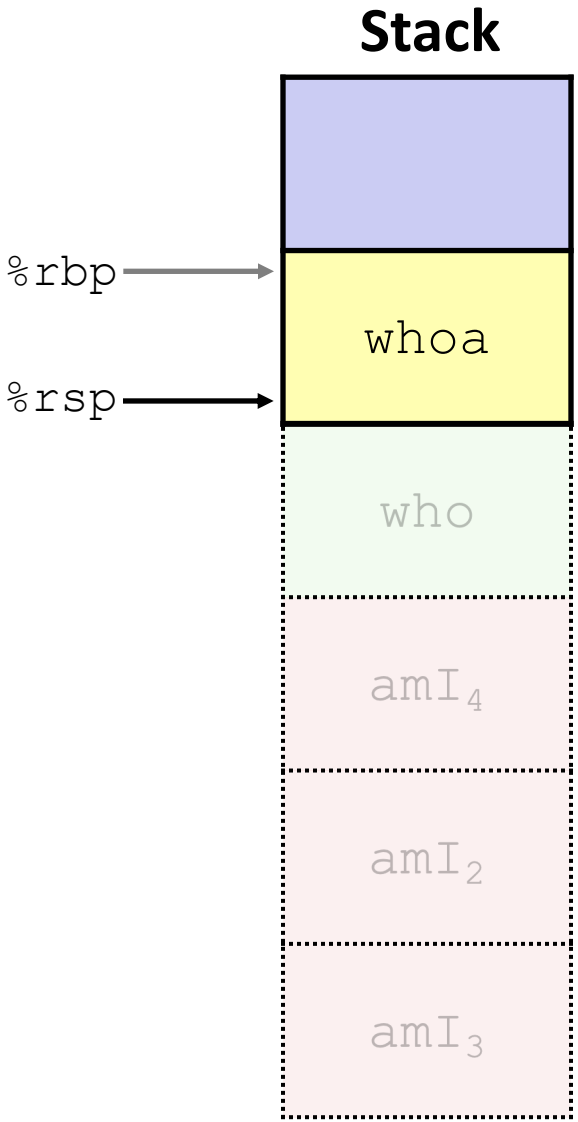




11) Return from call to who

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





Polling Question

- 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*?
  - What is the maximum *depth* (# of frames) of the Stack?

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

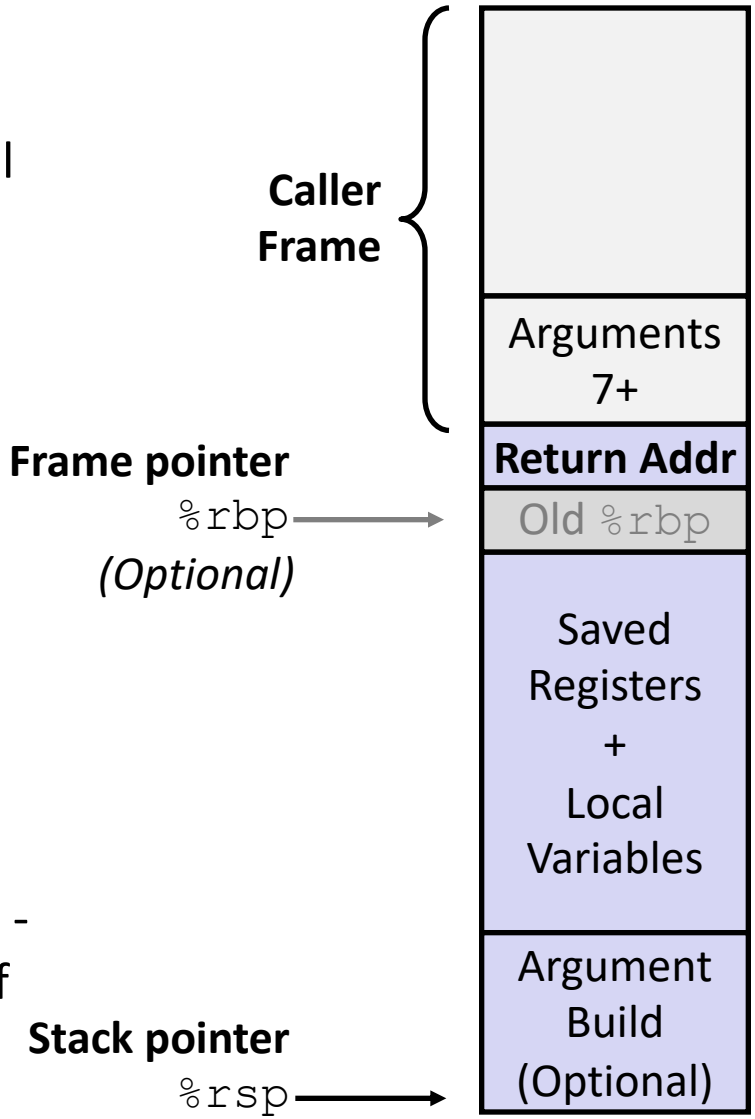
Reading Review

- Terminology:
  - Stack frame: return address, saved registers, local variables, argument build
  - Register saving conventions: callee-saved and caller-saved

- Questions from the Reading?

x86-64/Linux Stack Frame (Review)

- 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)



Review Question

- In the following function, which instruction(s) pertain to the **local variables** and **saved registers** portions of its stack frame?

```
  call_incr2:
 1  pushq   %rbx
 2  subq    $16, %rsp
 3  movq    %rdi, %rbx
 4  movq    $351, 8(%rsp)
 5  movl    $100, %esi
 6  leaq    8(%rsp), %rdi
 7  call    increment
 8  addq    %rbx, %rax
 9  addq    $16, %rsp
10  popq    %rbx
11  ret
```

Example: increment

```
long increment(long* p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
increment:
   movq     (%rdi), %rax
   addq     %rax, %rsi
   movq     %rsi, (%rdi)
   ret
```

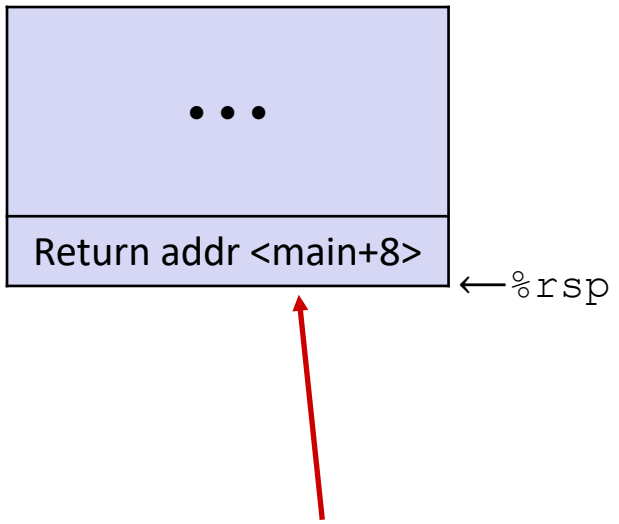
Register	Use(s)
%rdi	1st arg (p)
%rsi	2nd arg (val), y
%rax	x, return value

Procedure Call Example (initial state)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack Structure



- Return address on stack is the address of instruction immediately *following* the call to "`call_incr`"
  - Shown here as `main`, but could be anything)
  - Pushed onto stack by `call call_incr`

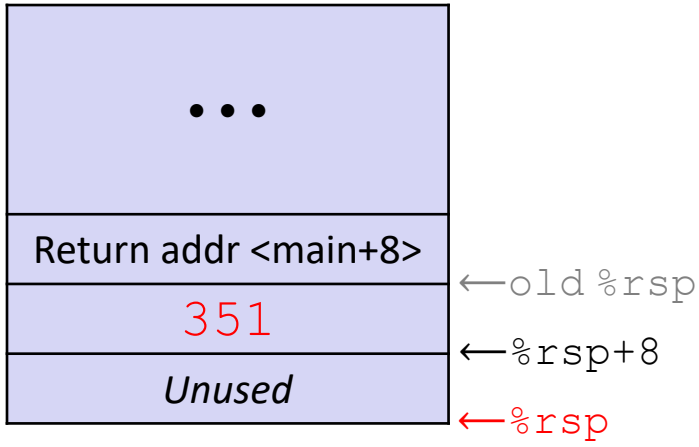
Procedure Call Example (step 1)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Allocate space for local vars

Stack Structure



- Setup space for local variables
  - Only `v1` needs space on the stack
- Compiler allocated extra space
  - Often does this for a variety of reasons, including alignment

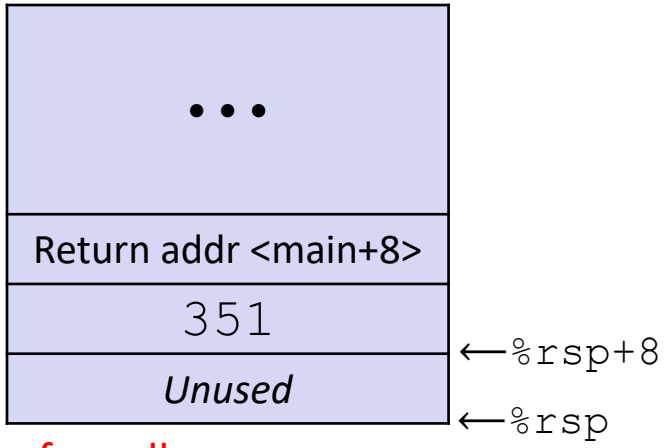
Procedure Call Example (step 2)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Set up parameters for call to `increment`

Stack Structure



Aside: `movl` is used because 100 is a small positive value that fits in 32 bits. High order bits of `rsi` get set to zero automatically. It takes *one less byte* to encode a `movl` than a `movq`.

Register	Use(s)
%rdi	&v1
%rsi	100

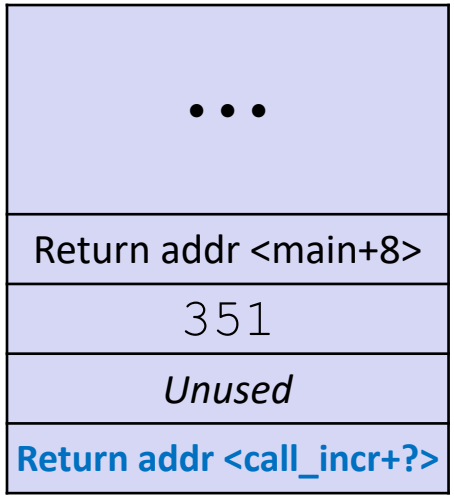
Procedure Call Example (step 3)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

```
increment:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

Stack Structure



- State while inside `increment`
  - **Return address** on top of stack is address of the `addq` instruction immediately following call to `increment`

Register	Use(s)
%rdi	&v1
%rsi	100
%rax	

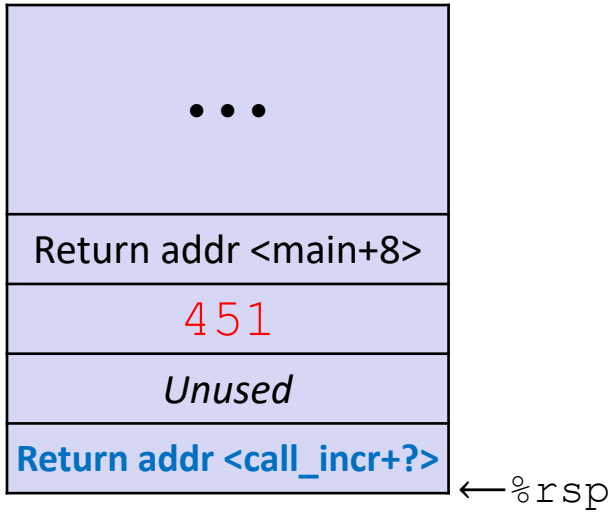
Procedure Call Example (step 4)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

```
increment:
  movq    (%rdi), %rax  # x = *p
  addq    %rax, %rsi    # y = x + 100
  movq    %rsi, (%rdi)  # *p = y
  ret
```

Stack Structure



- State while inside `increment`
  - *After* code in body has been executed

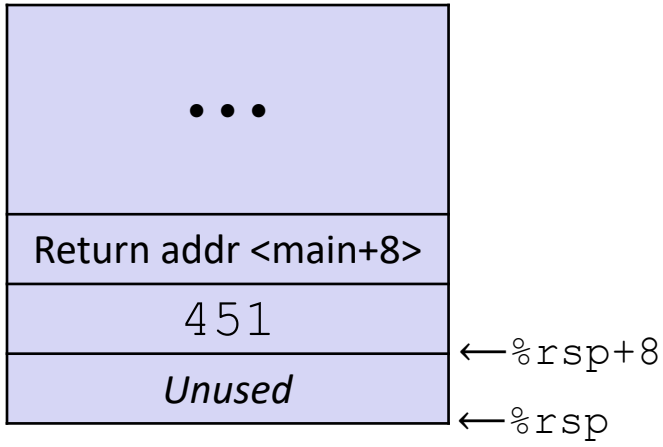
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	351

Procedure Call Example (step 5)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



- After returning from call to `increment`
  - Registers and memory have been modified and return address has been popped off stack

Register	Use(s)
%rdi	&v1
%rsi	451
%rax	351

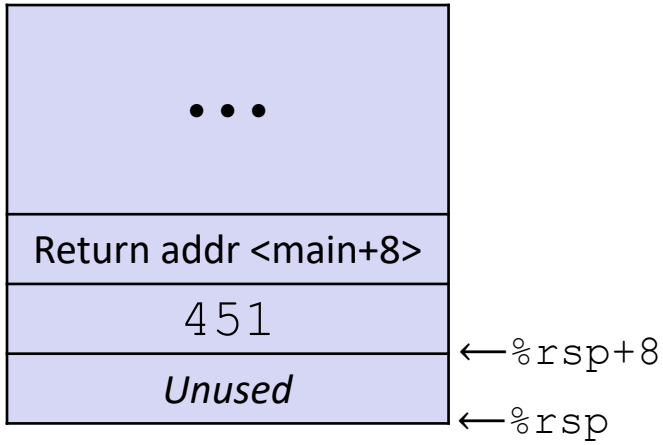
Procedure Call Example (step 6)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

← Update %rax to contain v1+v2

Stack Structure



Register	Use(s)
%rdi	&v1
%rsi	451
%rax	451+351

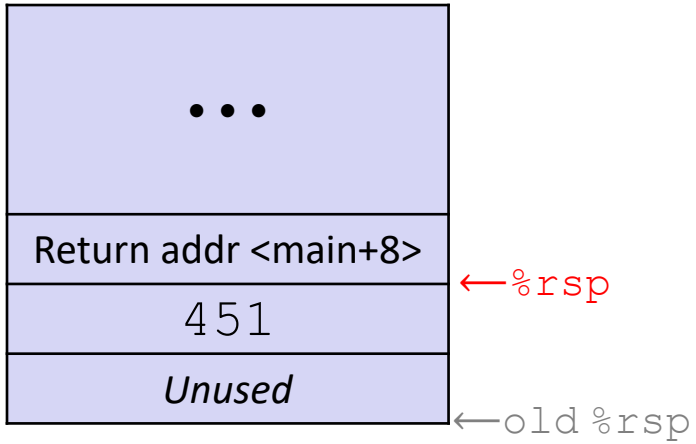
Procedure Call Example (step 7)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

← De-allocate space for local vars

Stack Structure



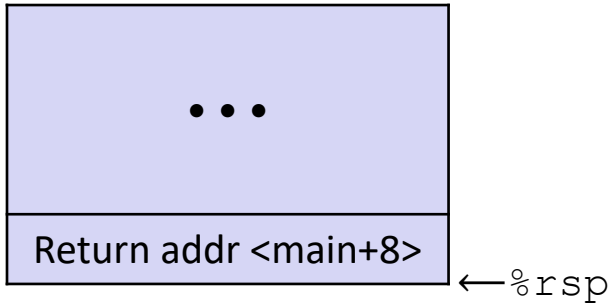
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

Procedure Call Example (step 8)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



- State *just before* returning from call to `call_incr`

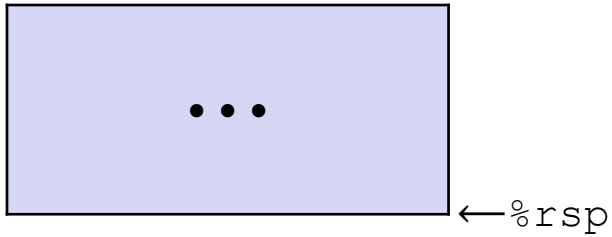
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

Procedure Call Example (step 9)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Final Stack Structure



- State immediately *after* returning from call to `call_incr`
  - Return addr has been popped off stack
  - Control has returned to the instruction immediately following the call to `call_incr` (not shown here)

Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

Procedures

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

Register Saving Conventions (Review)

- When procedure `whoa` calls `who`:
  - `whoa` is the caller
  - `who` is the callee
- Can registers be used for temporary storage?

```
whoa:
    . . .
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    . . .
    ret
```

```
who:
    . . .
    subq $18213, %rdx
    . . .
    ret
```

  - No! Contents of register `%rdx` overwritten by `who`!
  - This could be trouble – something should be done. Either:
    - Caller should save `%rdx` before the call (and restore it after the call)
    - Callee should save `%rdx` before using it (and restore it before returning)

Register Saving Conventions (Review)

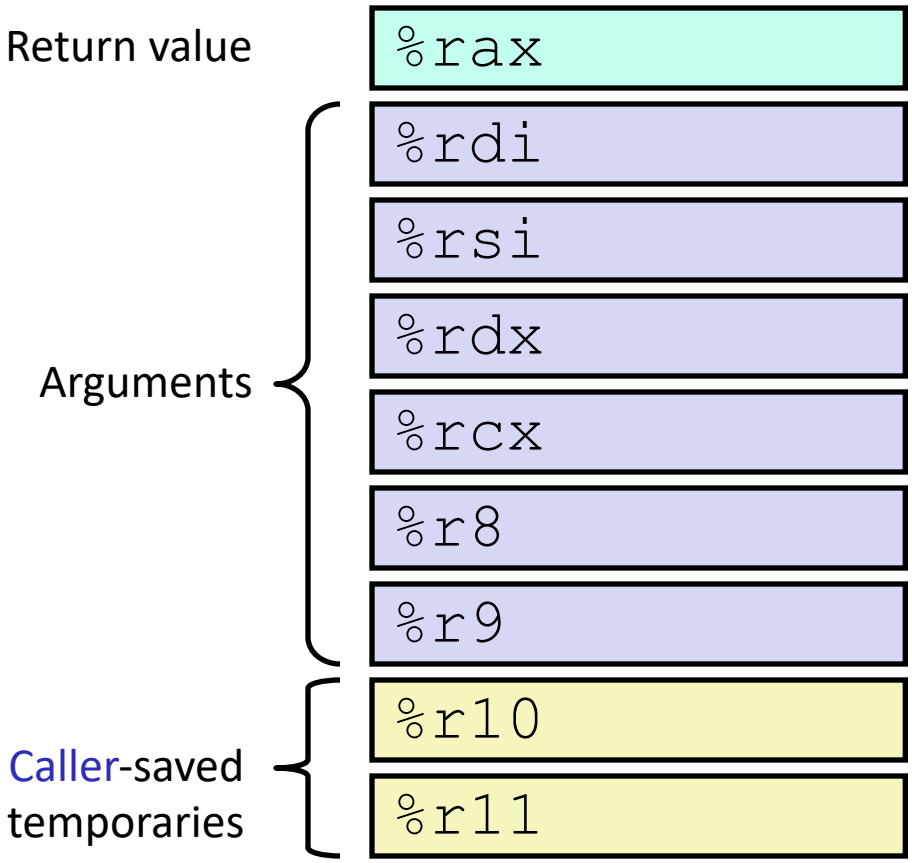
- ***"Caller-saved" registers***
  - It is the **caller**'s responsibility to save any important data in these registers before calling another procedure (*i.e.*, the **callee** can freely change data in these registers)
  - **Caller** saves values in its stack frame before calling **Callee**, then restores values after the call
- ***"Callee-saved" registers***
  - It is the callee's responsibility to save any data in these registers before using the registers (*i.e.*, the **caller** assumes the data will be the same across the **callee** procedure call)
  - **Callee** saves values in its stack frame before using, then restores them before returning to **caller**

Silly Register Convention Analogy

1) Parents (***caller***) leave for the weekend and give the keys to the house to their child (***callee***)
   - Being suspicious, they put away/hid the valuables (***caller***-*saved*) before leaving
   - Warn child to leave the bedrooms untouched: "These rooms better look the same when we return!"
2) Child decides to throw a wild party (*computation*), spanning the entire house
   - To avoid being disowned, child moves all the stuff from the bedrooms to the backyard shed (***callee***-*saved*) before the guests trash the house
   - Child cleans up house after the party and moves stuff back to bedrooms
3) Parents return home and are satisfied with the state of the house
   - Move valuables back and continue with their lives

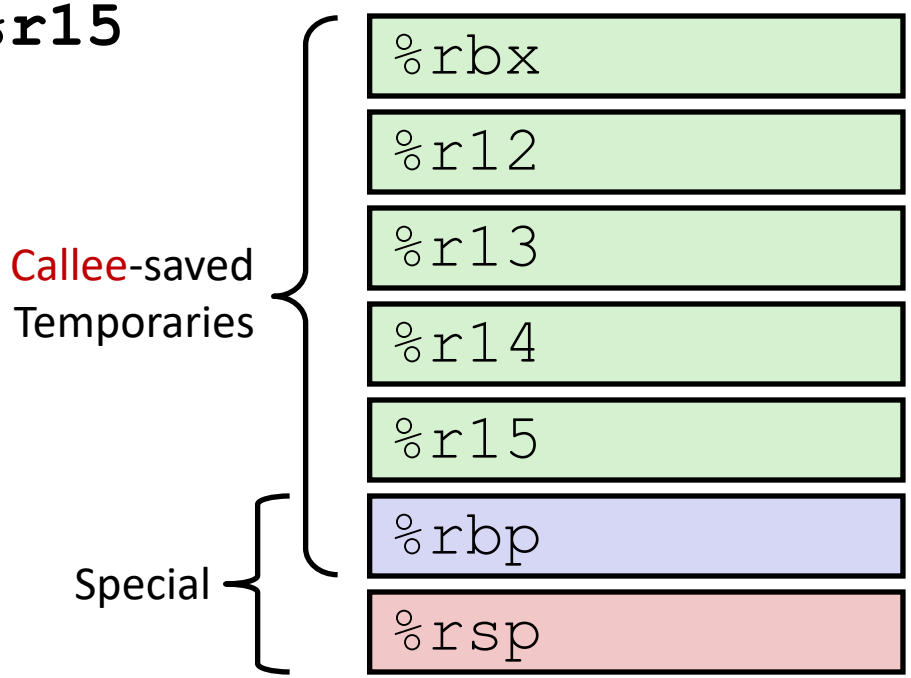
x86-64 Linux Register Usage (Review)

- **`%rax`**
  - Return value
  - Also **caller**-saved & restored
  - Can be modified by procedure
- **`%rdi, ..., %r9`**
  - Arguments
  - Also **caller**-saved & restored
  - Can be modified by procedure
- **`%r10, %r11`**
  - **Caller**-saved & restored
  - Can be modified by procedure



x86-64 Linux Register Usage (Review)

- **`%rbx`, `%r12`, `%r13`, `%r14`, `%r15`**
  - **Callee**-saved
  - **Callee** must save & restore
- **`%rbp`**
  - **Callee**-saved
  - **Callee** must save & restore
  - May be used as frame pointer
  - Can mix & match
- **`%rsp`**
  - Special form of **callee** save
  - Restored to original value upon exit from procedure



x86-64 Linux Register Usage (Review)

%rax	Return value - Caller saved
%rbx	Callee saved
%rcx	Argument #4 - Caller saved
%rdx	Argument #3 - Caller saved
%rsi	Argument #2 - Caller saved
%rdi	Argument #1 - Caller saved
%rsp	Stack pointer
%rbp	Callee saved

%r8	Argument #5 - Caller saved
%r9	Argument #6 - Caller saved
%r10	Caller saved
%r11	Caller Saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved