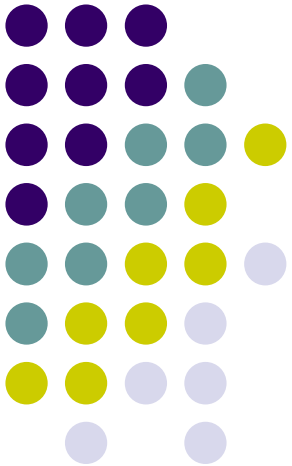
System Programming

08. Machine-Level Programming III: Procedures



2019. Fall

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

datalab

data science laboratory

These slides are based on your text book on the slides prepared by Dan Grossman, Ruth Anderson.

Roadmap

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. 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();
```

Assembly language:

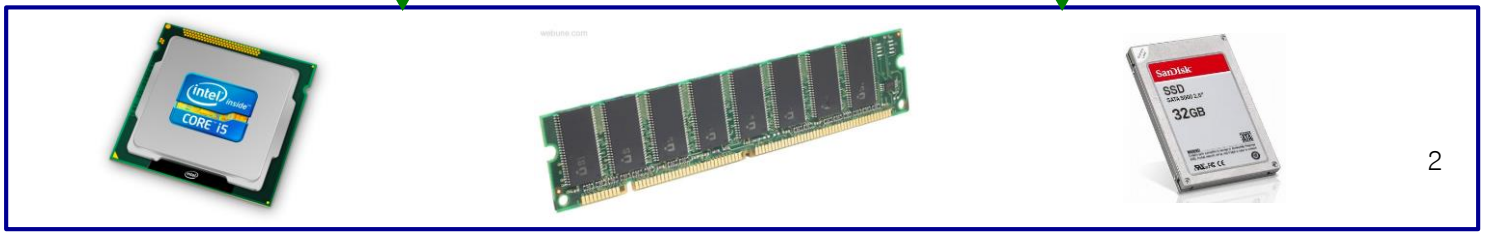
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
01110100000110000
10001101000001000000010
1000100111000010
11000001111110100001111
```

OS:

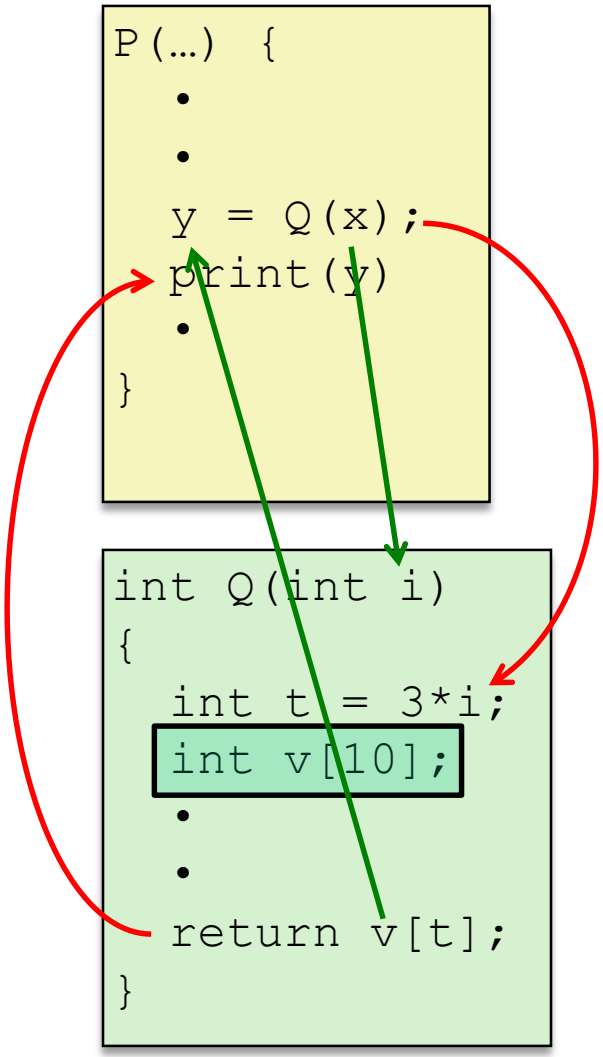
Computer system:



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



Questions to answer about procedures

- How do I pass arguments to a procedure?
- How do I get a return value from a procedure?
- Where do I put local variables?
- When a function returns, how does it know where to return?

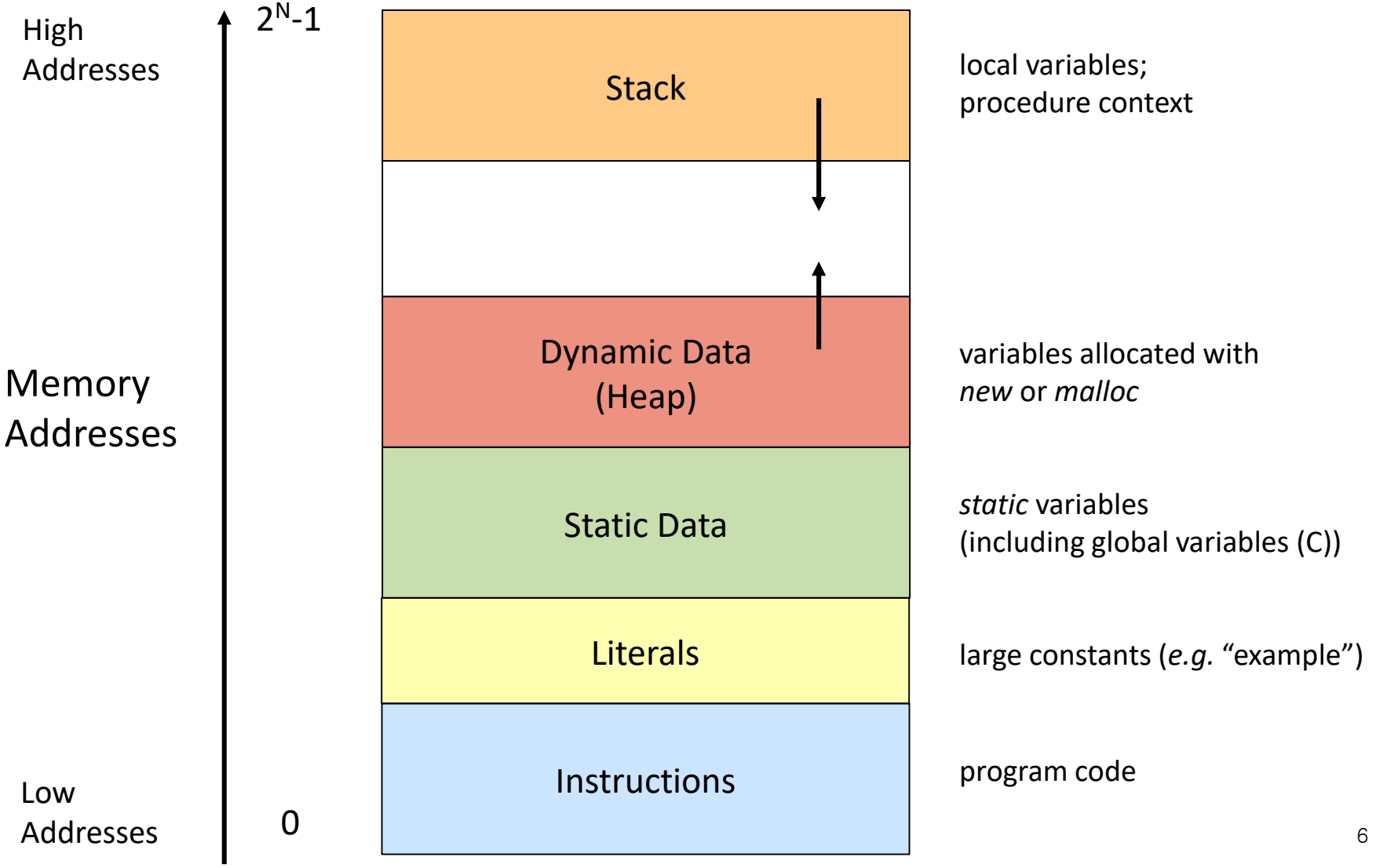
- To answer some of these questions,

  we need a **call stack** …

Outline

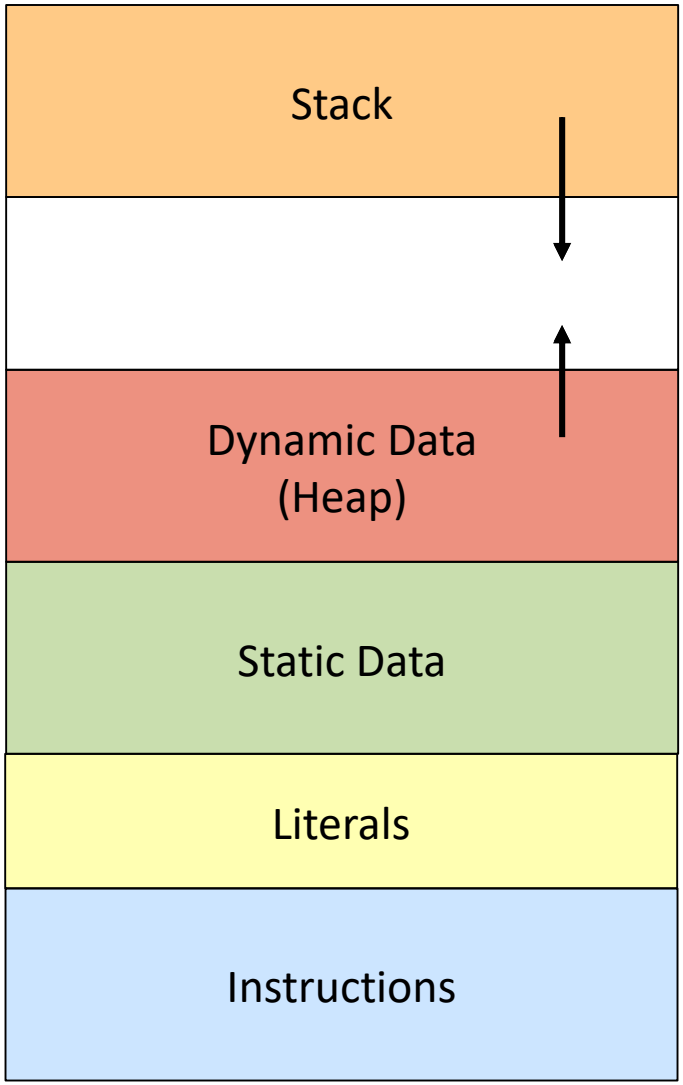
- **Stack Structure**
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Memory Layout



Memory Permissions

segmentation faults?



writable; not executable

writable; not executable

writable; not executable

read-only; not executable

read-only; executable

Managed "automatically"
(by compiler)

Managed by programmer

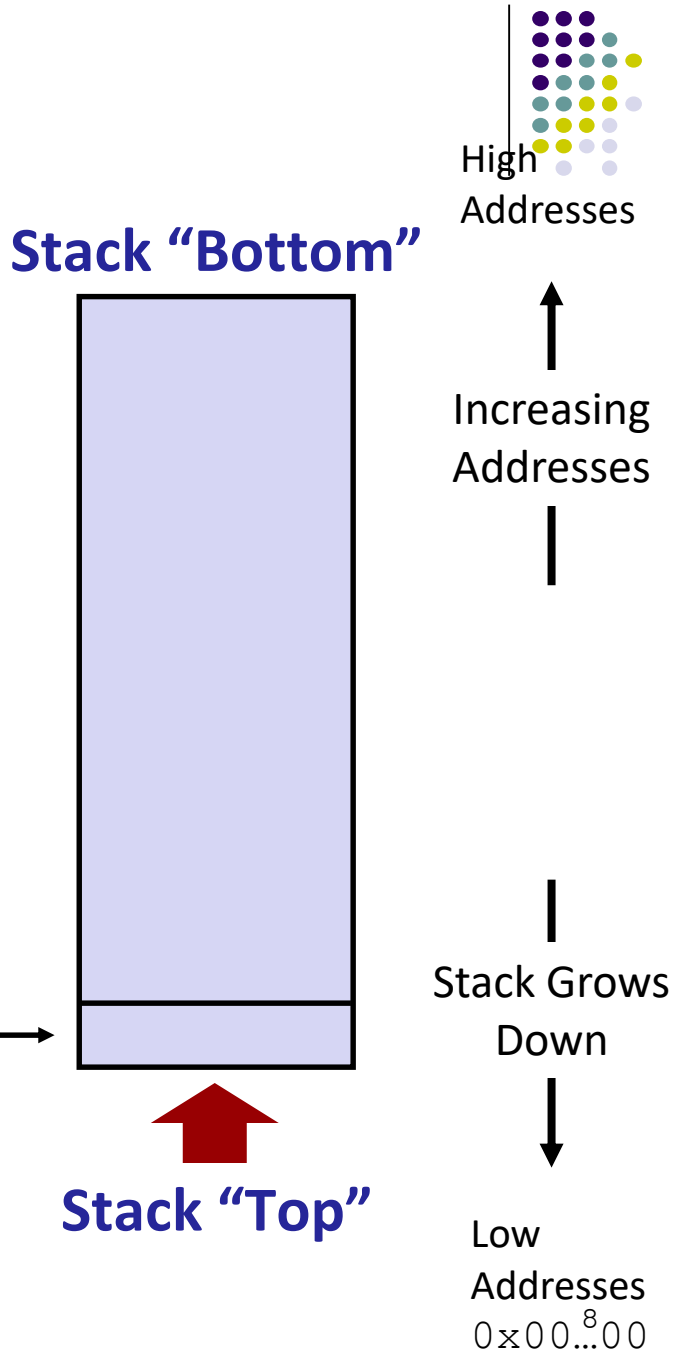
Initialized when process starts

Initialized when process starts

Initialized when process starts

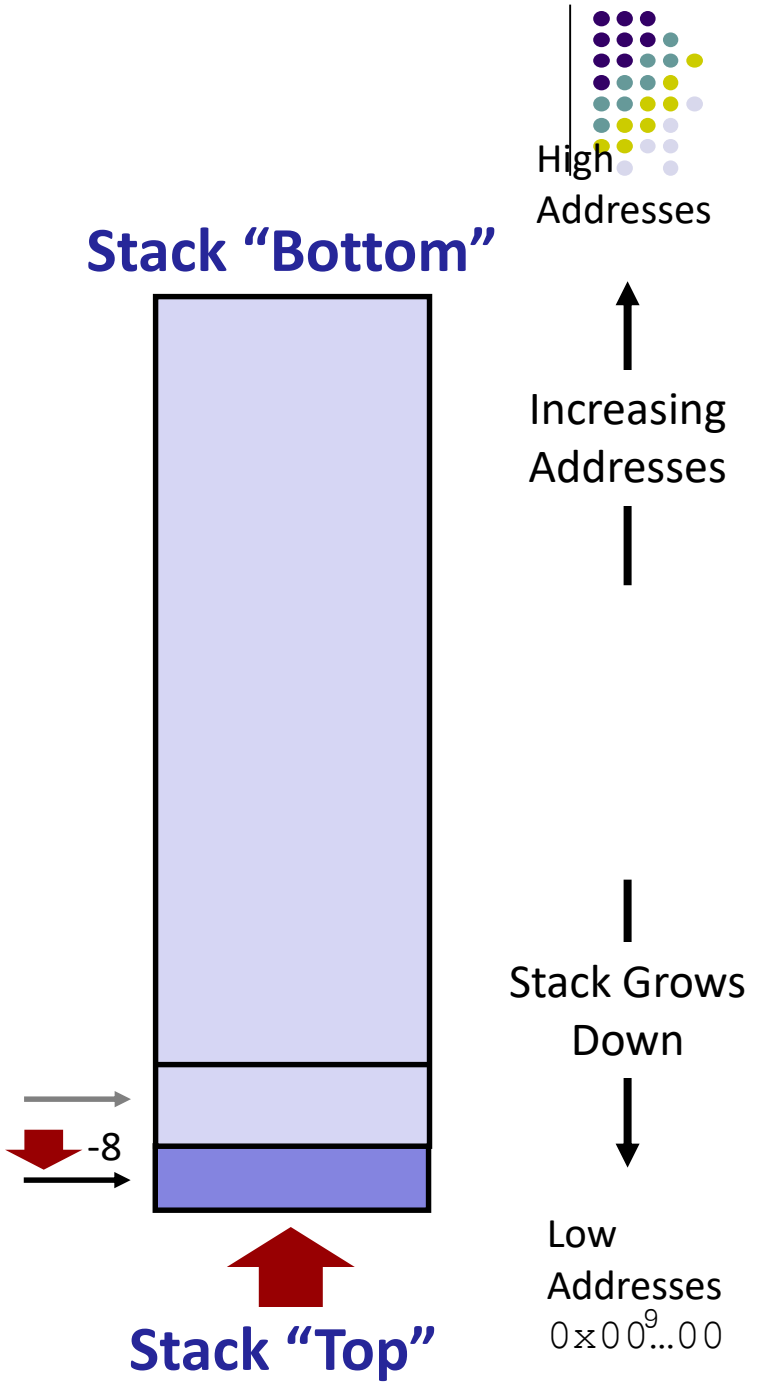
x86-64 Stack

- 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

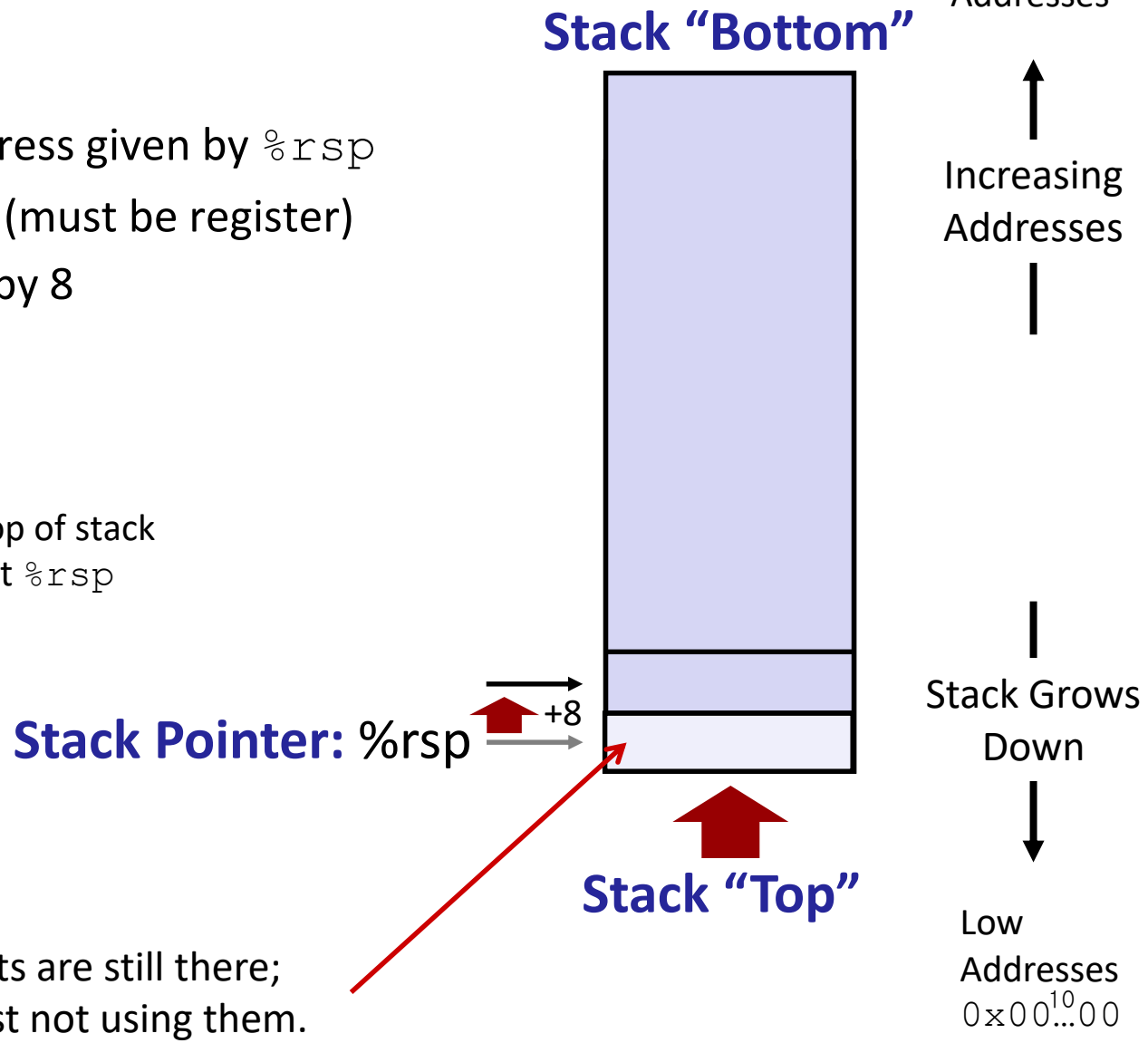
- `pushq` *src*
  - Fetch operand at *src*
    - *Src* can be reg, memory, immediate
  - ***Decrement*** `%rsp` by 8
  - Store value at address given by `%rsp`
- Example:
  - **`pushq %rcx`**
  - Adjust `%rsp` and store contents of `%rcx` on the stack



x86-64 Stack: Pop

- `popq` *dst*
  - Load value at address given by `%rsp`
  - Store value at *dst* (must be register)
  - ***Increment*** `%rsp` by 8

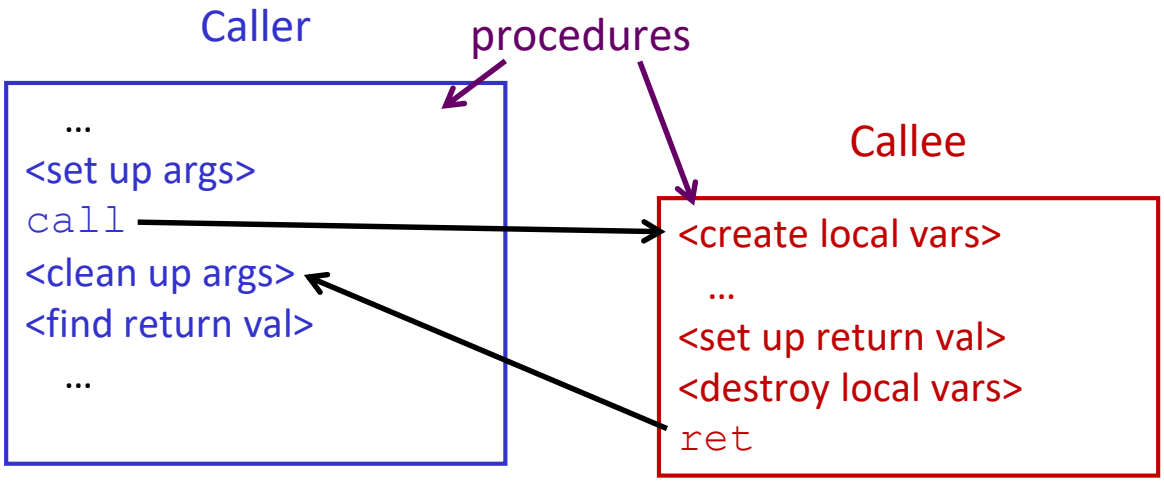
- Example:
  - **`popq %rcx`**
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`



Outline

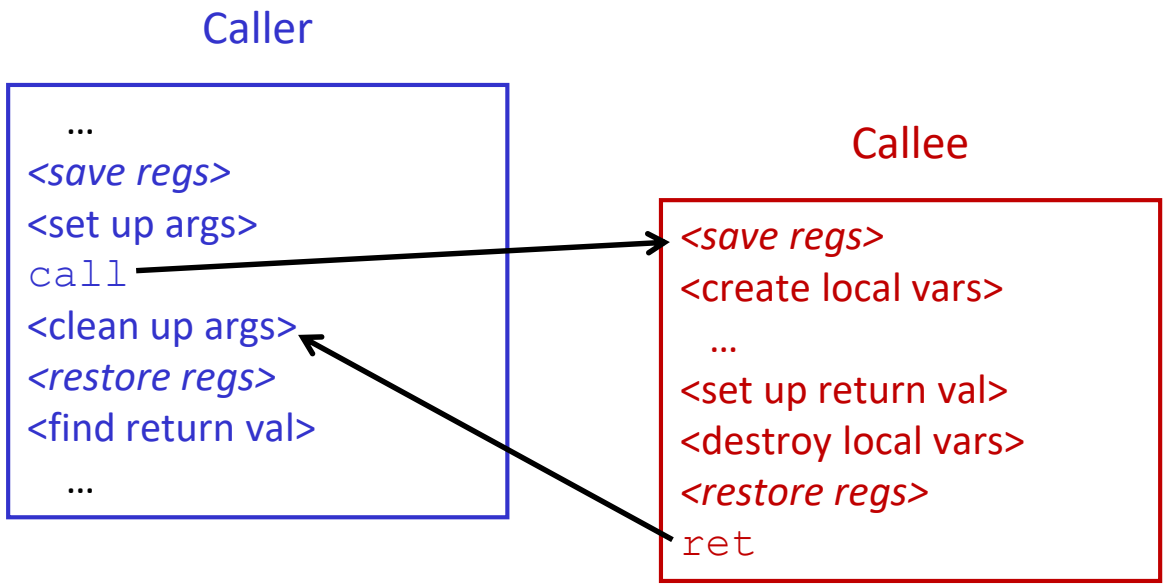
- Stack Structure
- Calling Conventions
  - **Passing control**
  - Passing data
  - Managing local data
- Illustration of Recursion

Procedure Call Overview (1)



- **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 (2)



- 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 Examples

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

Compiler Explorer:
https://godbolt.org/g/52Sqxj

```
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 (1)

- 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 (2)

- 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**

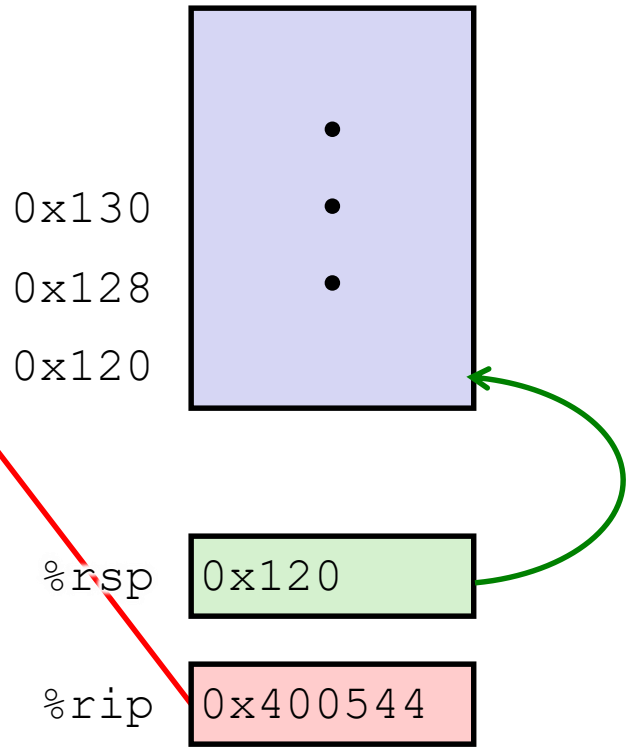
- Procedure return: `ret`
  1) Pop return address from stack
  2) Jump to address

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)
   •
   •
```

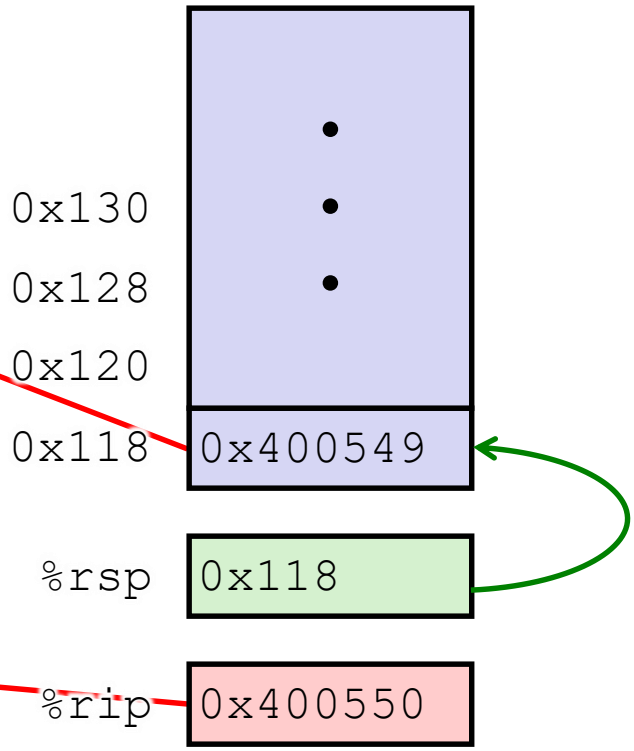
```
0000000000400550 <mult2>:
   400550:  movq    %rdi,%rax
   •
   •
   400557:  ret
```



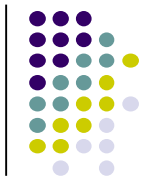
Procedure Call Example (Step2)

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

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

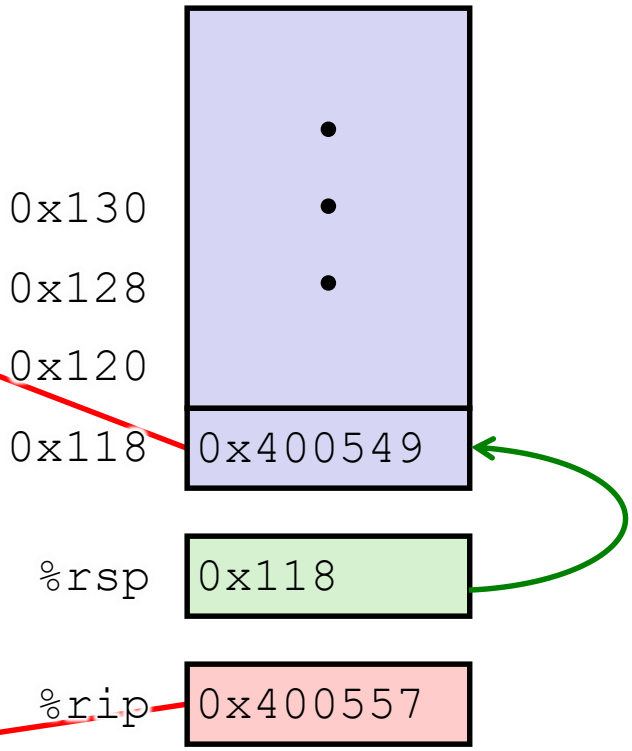


Procedure Return Example (Step1)



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

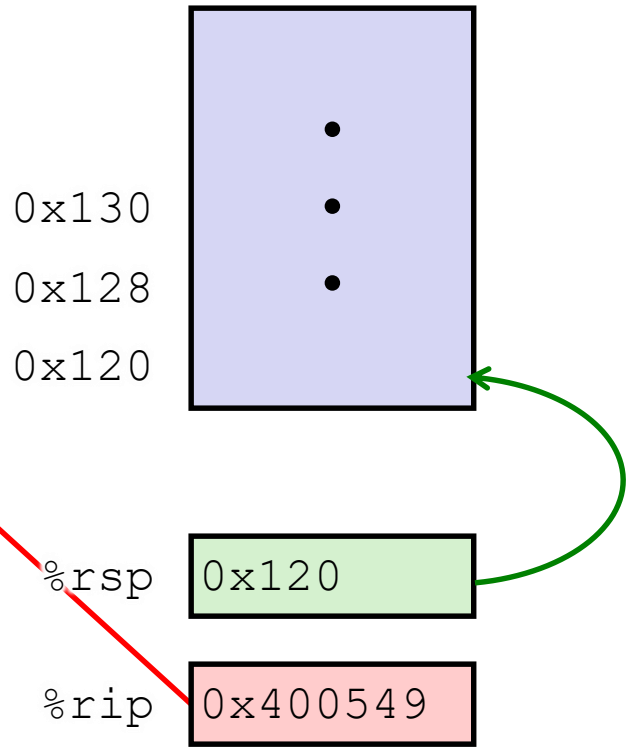
```
0000000000400550 <mult2>:
   400550:  movq    %rdi,%rax
   •
   •
   400557:  ret
```



Procedure Return Example (Step2)

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

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



Outline

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

Procedure Data Flow

- **Registers – NOT in Memory!**
  - **First 6 arguments**

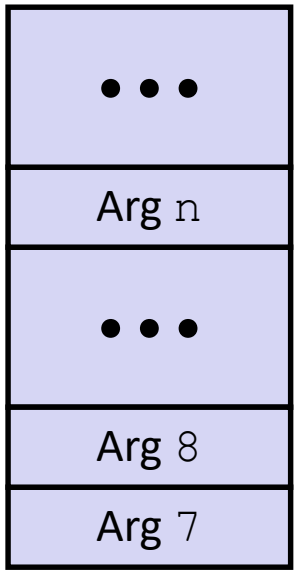
%rdi
%rsi
%rdx
%rcx
%r8
%r9

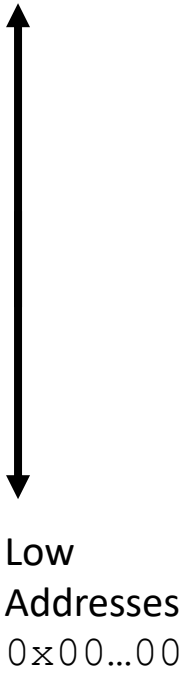
Diane's Silk Dress Costs $8 9

  - **Return value**

%rax

- **Stack – in 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
```

Outline

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - **Managing local data**
- 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

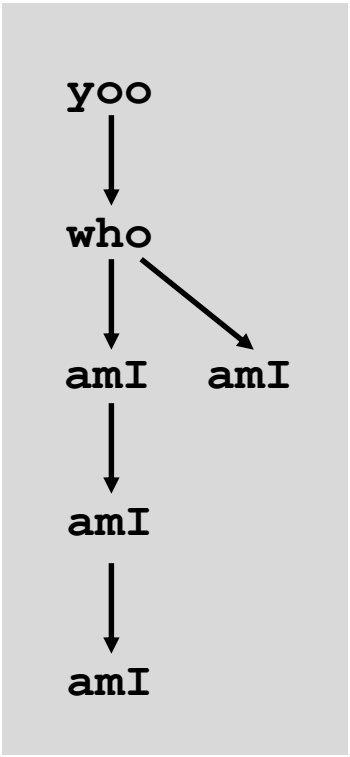
Call Chain Example

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

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

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

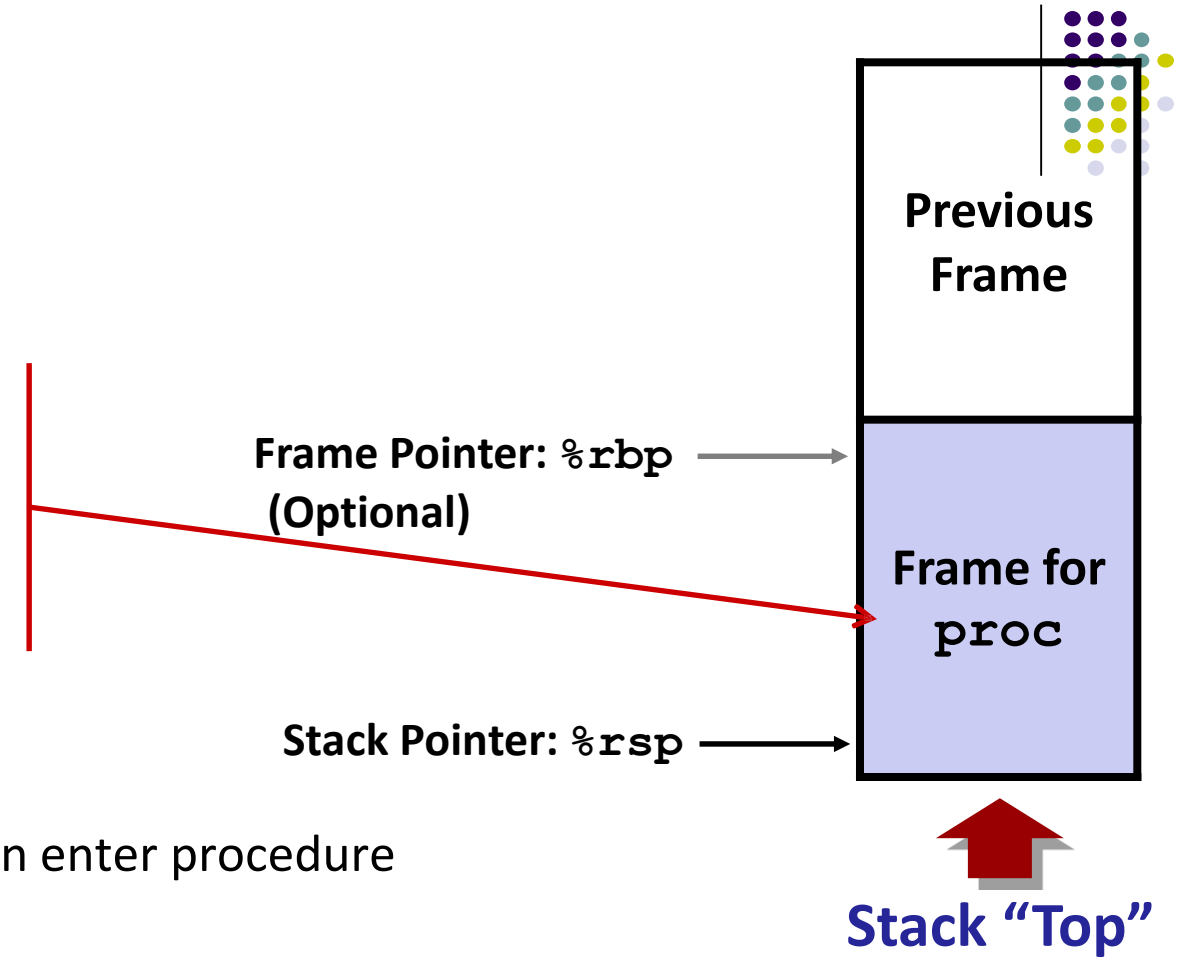
Example
Call Chain



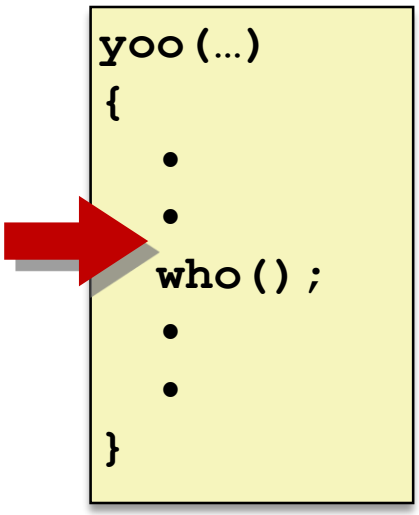
Procedure `amI` is recursive
(calls itself)

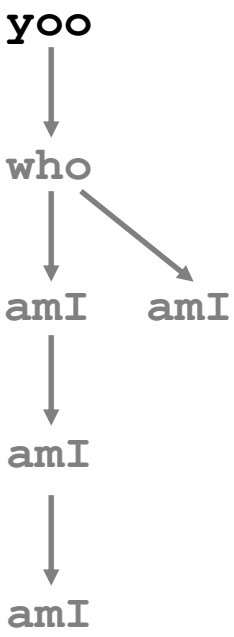
Stack Frames

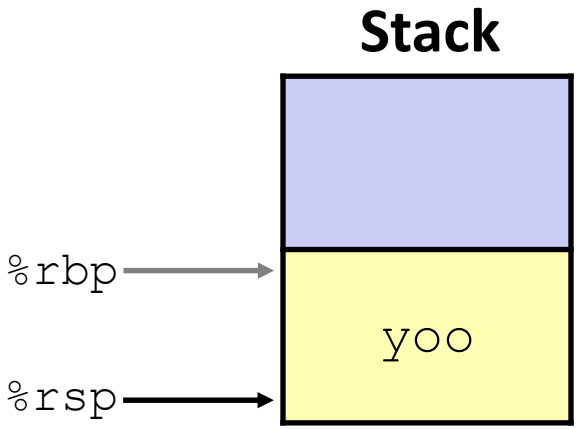
- Contents
  - Return information
  - If needed
    - Local storage
    - Temporary space
- Management
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by **`call`** instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by **`ret`** instruction



1) Call to yoo

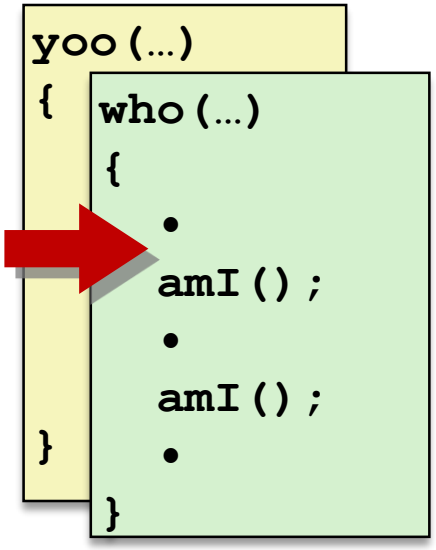


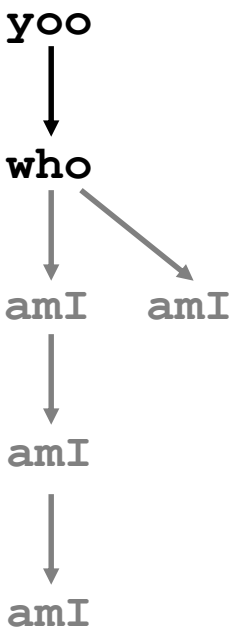


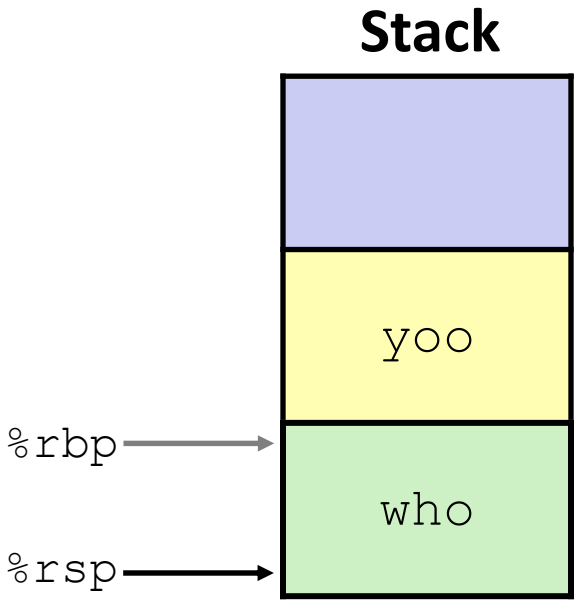


29

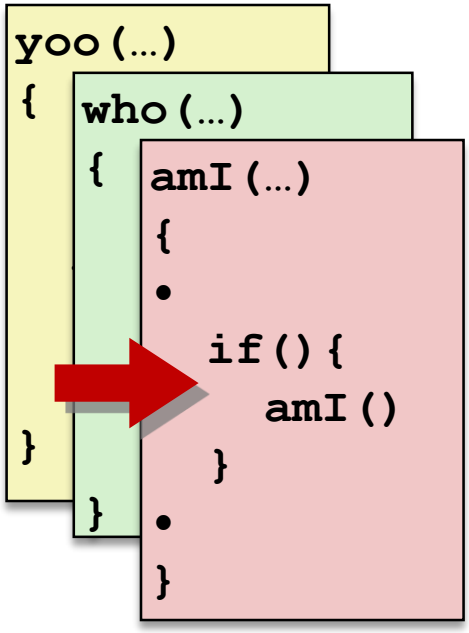
2) Call to who

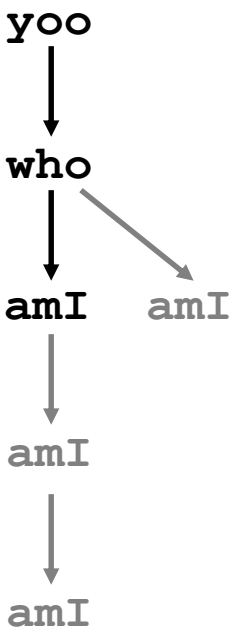


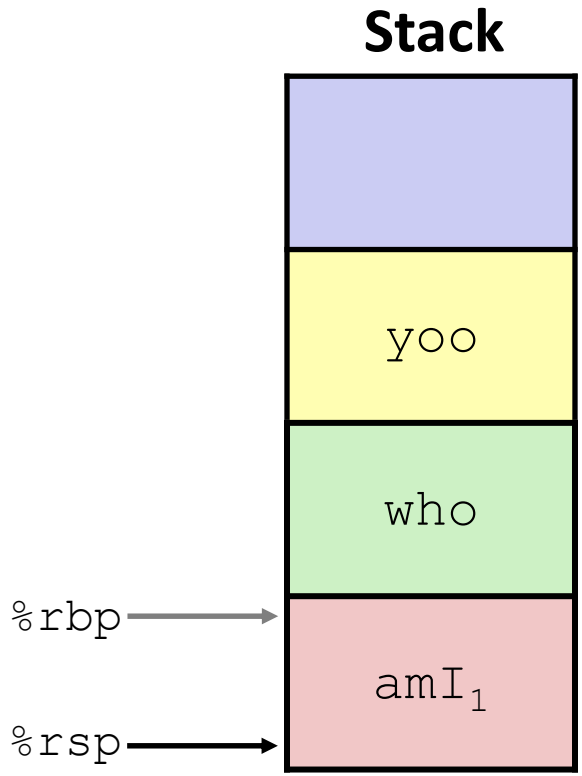




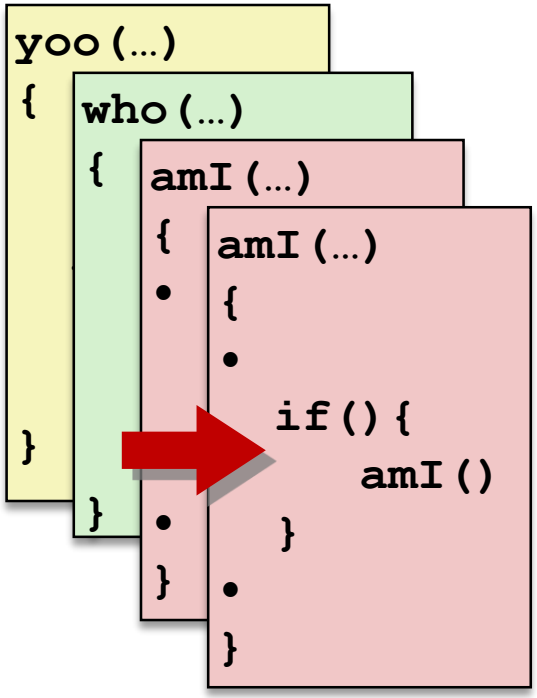
3) Call to amI (1)

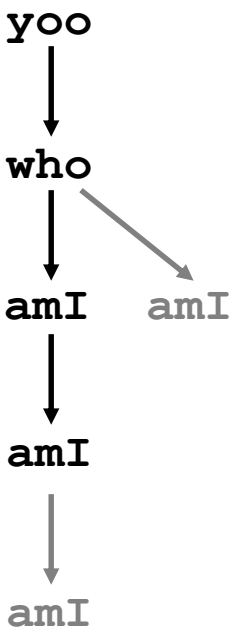


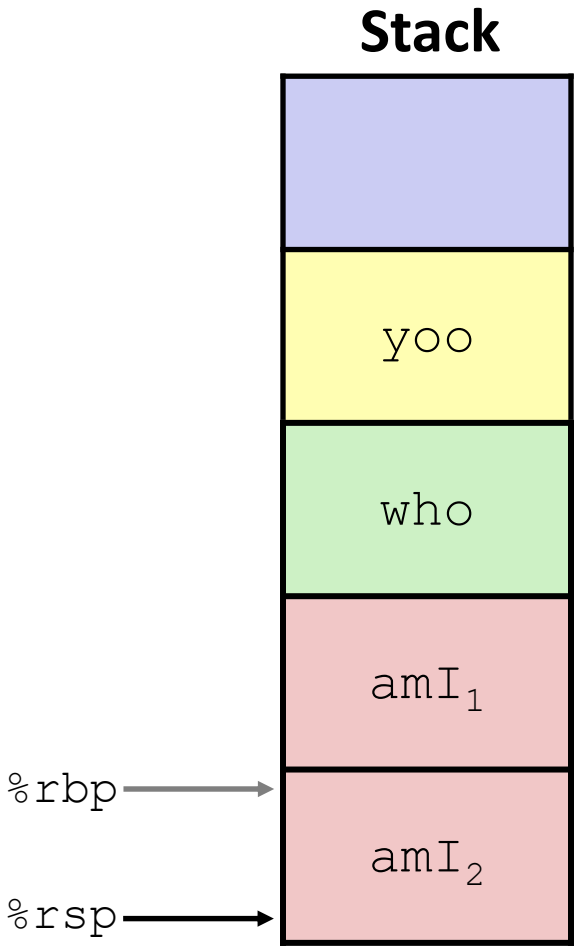




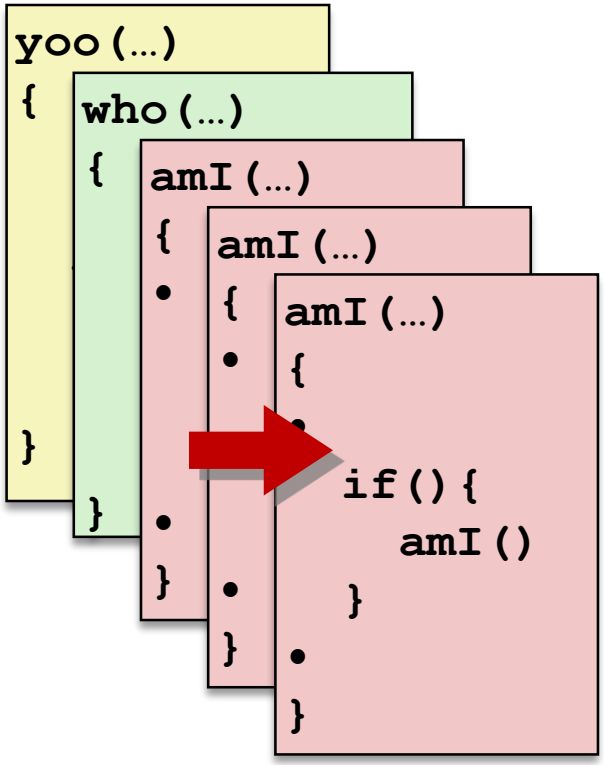
4) Recursive call to amI (2)

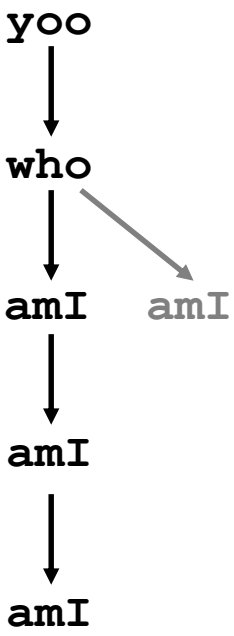


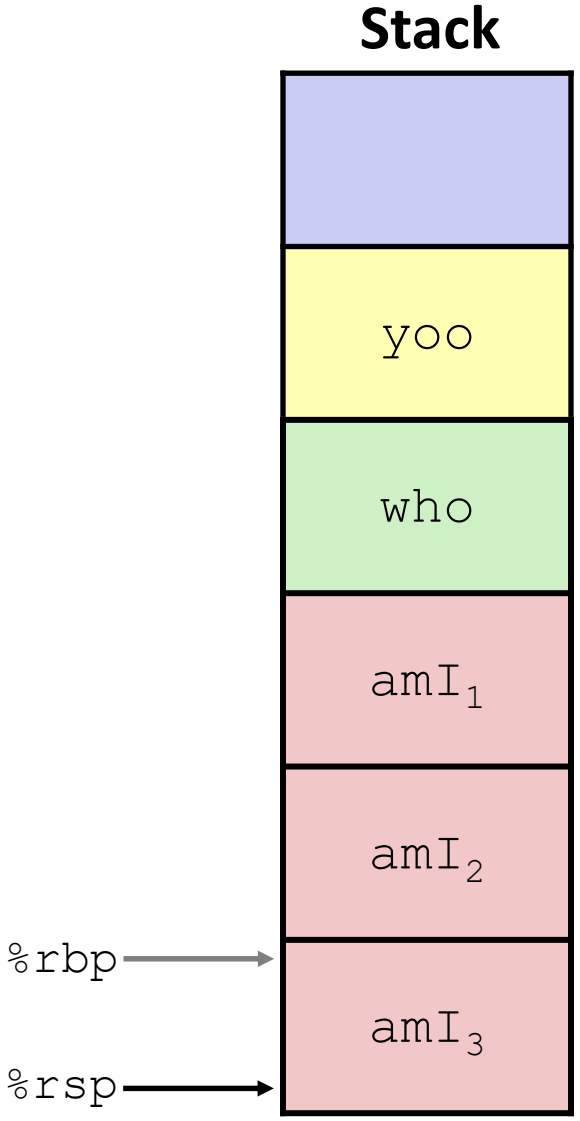




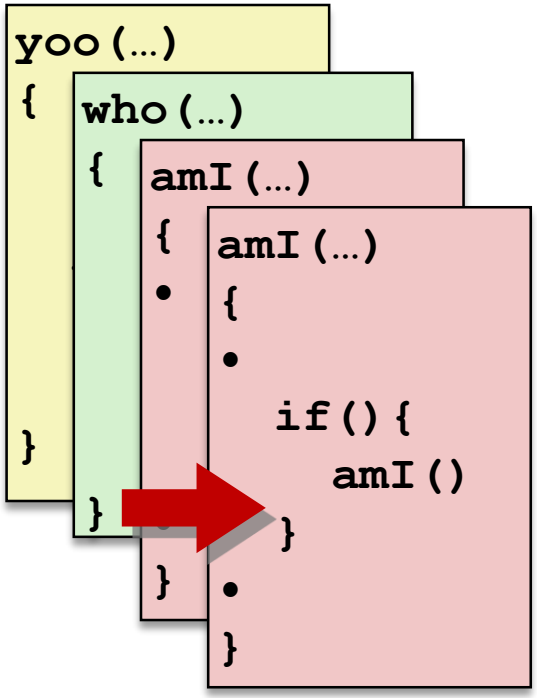
5) (another) Recursive call to amI (3)

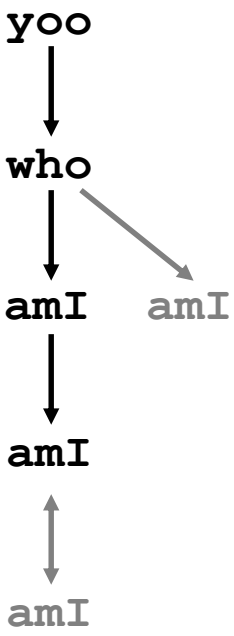


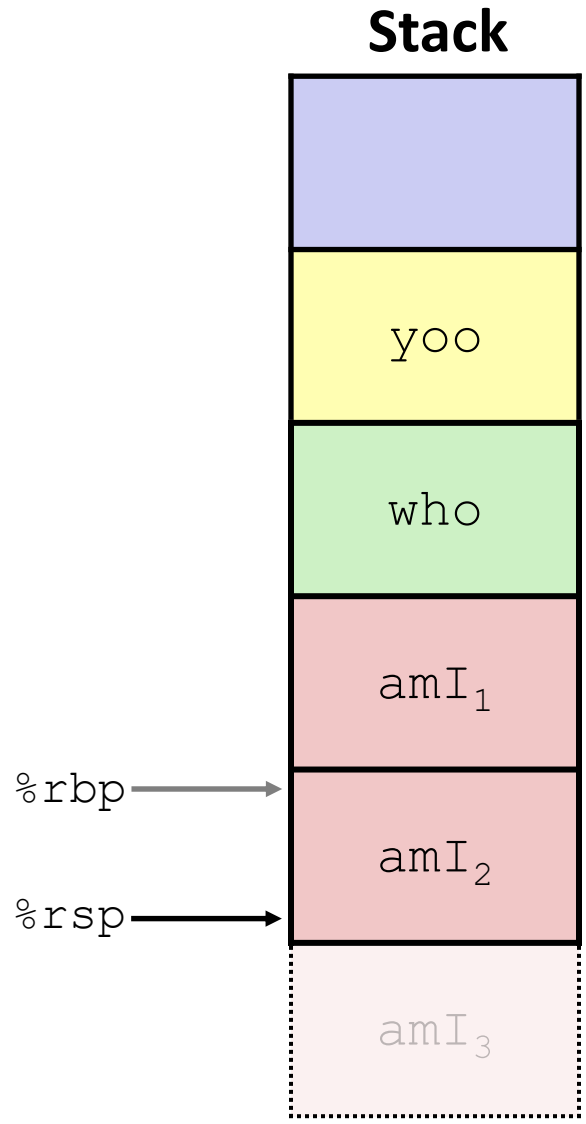




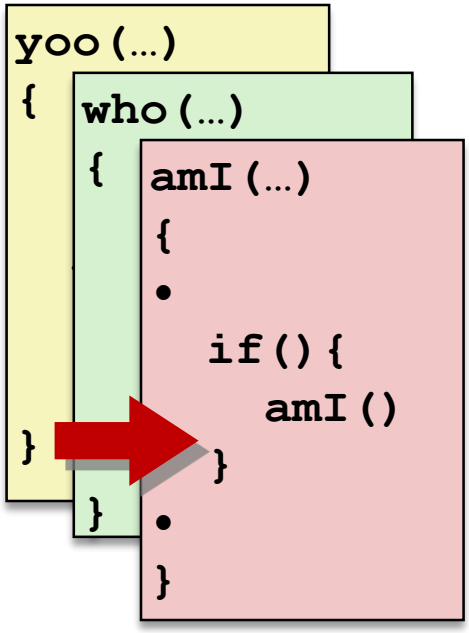
6) Return from (another) recursive call to amI

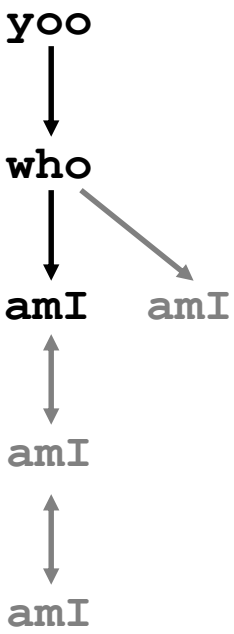


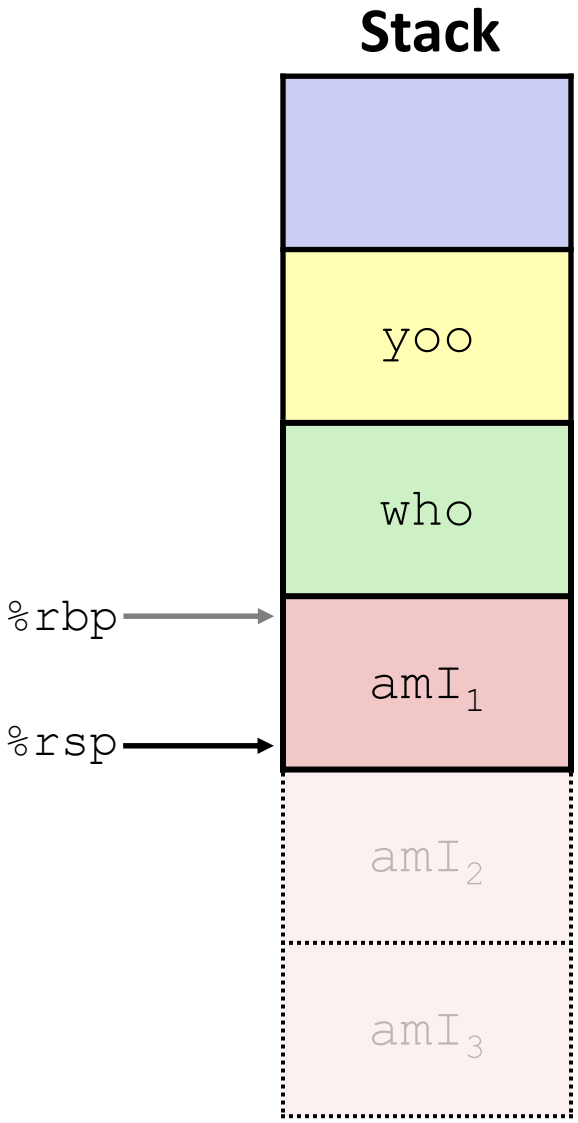




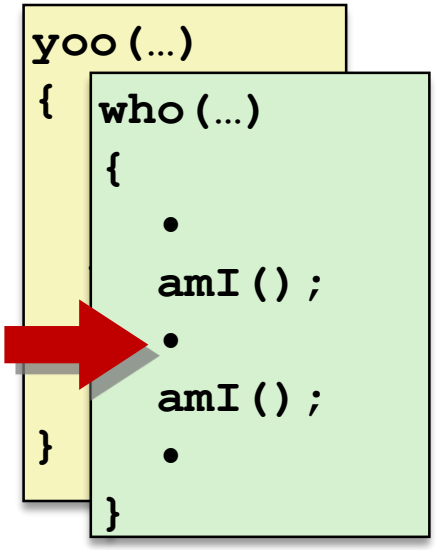
7) Return from recursive call to amI

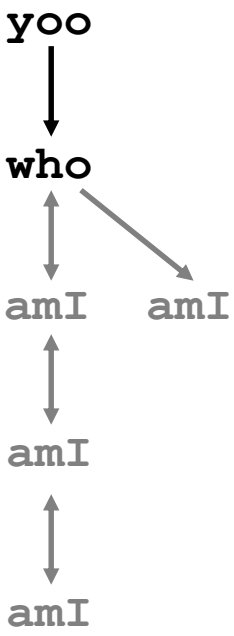


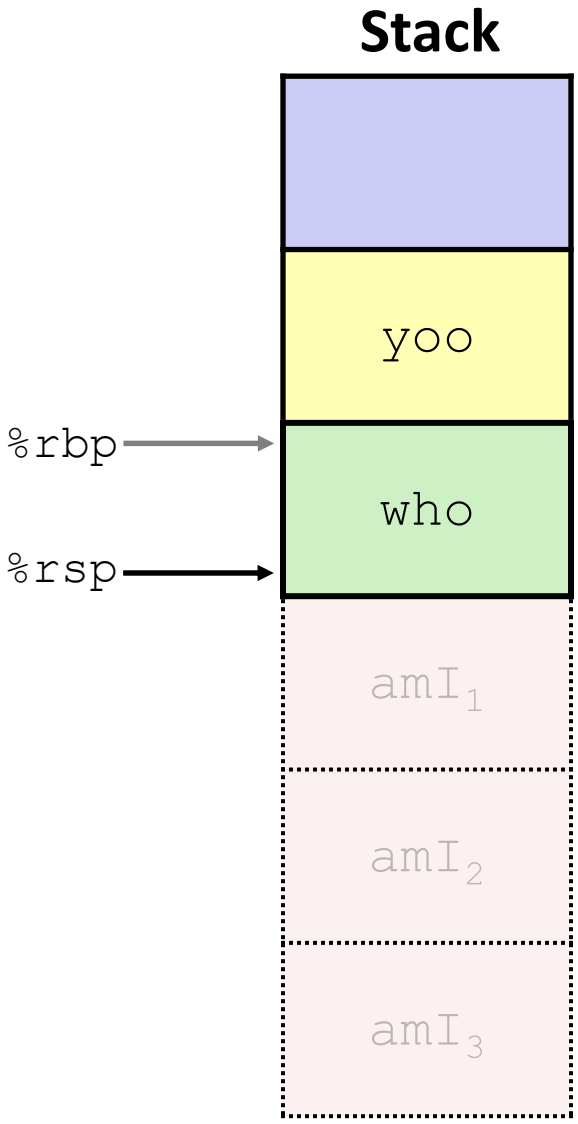




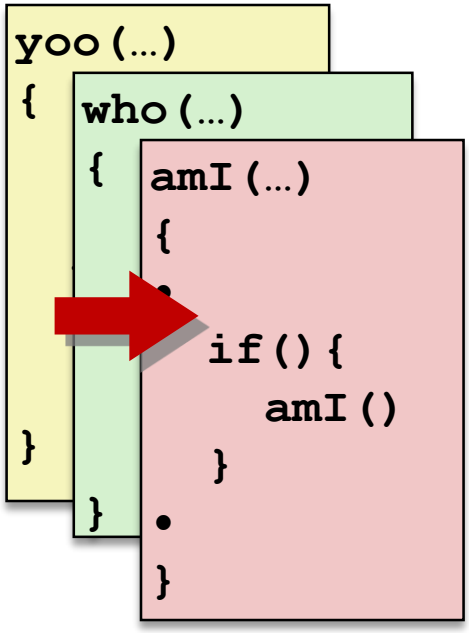
8) Return from call to amI

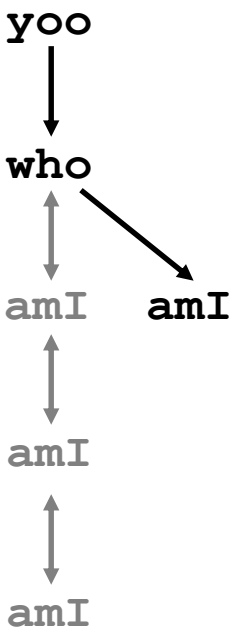


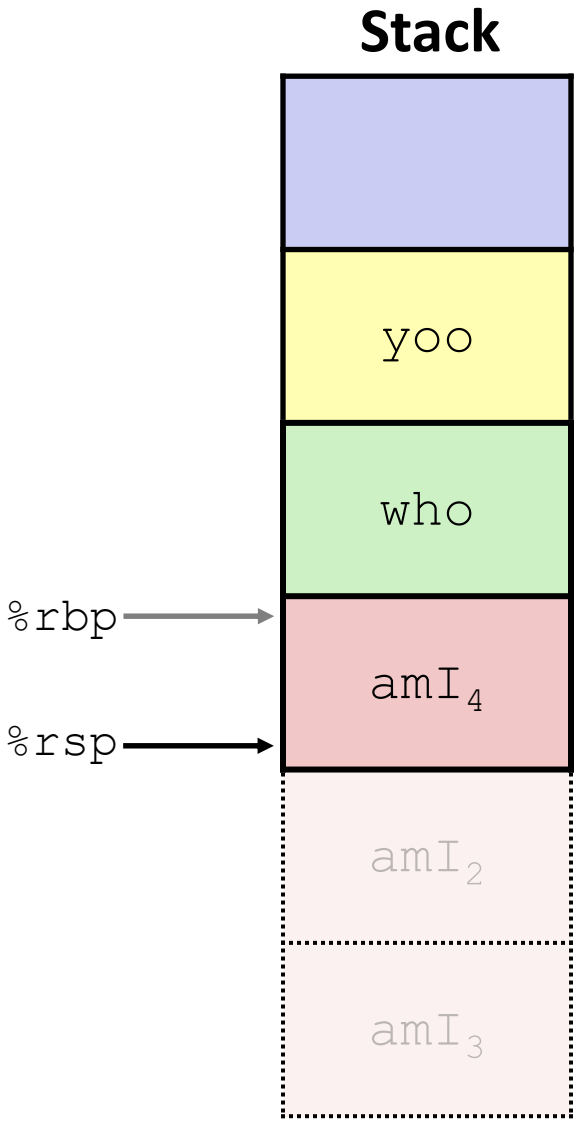




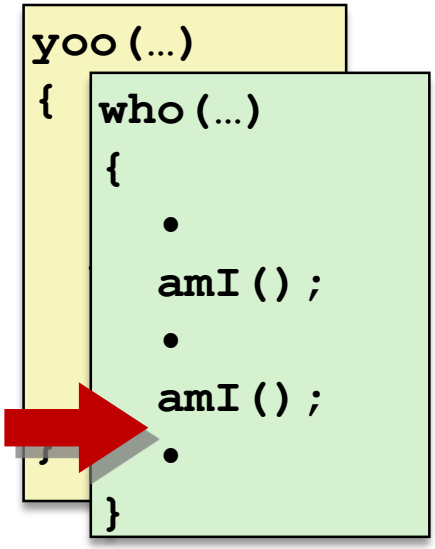
9) (second) Call to amI (4)

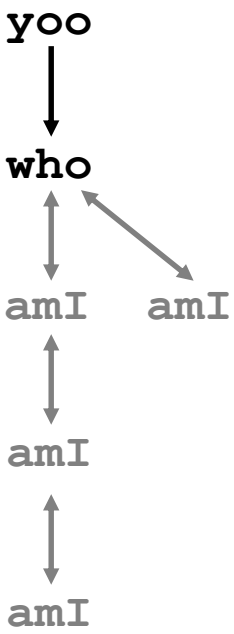


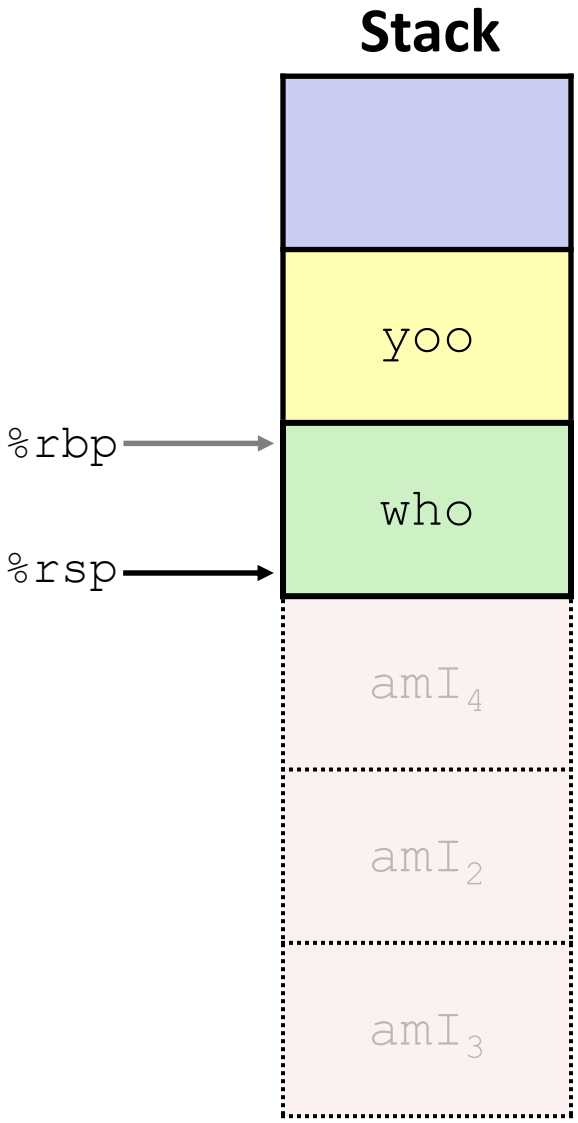




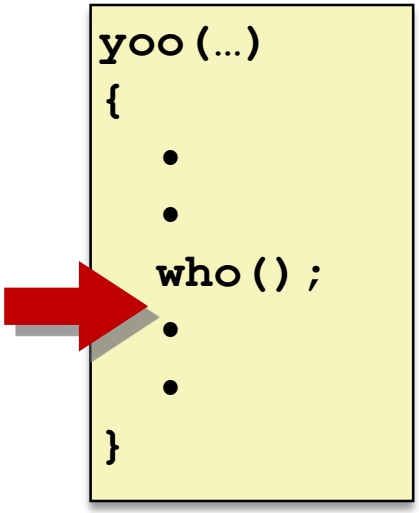
10) Return from (second) call to amI

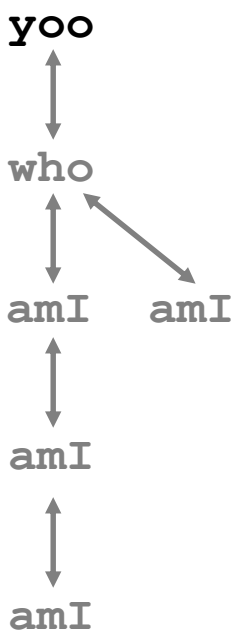


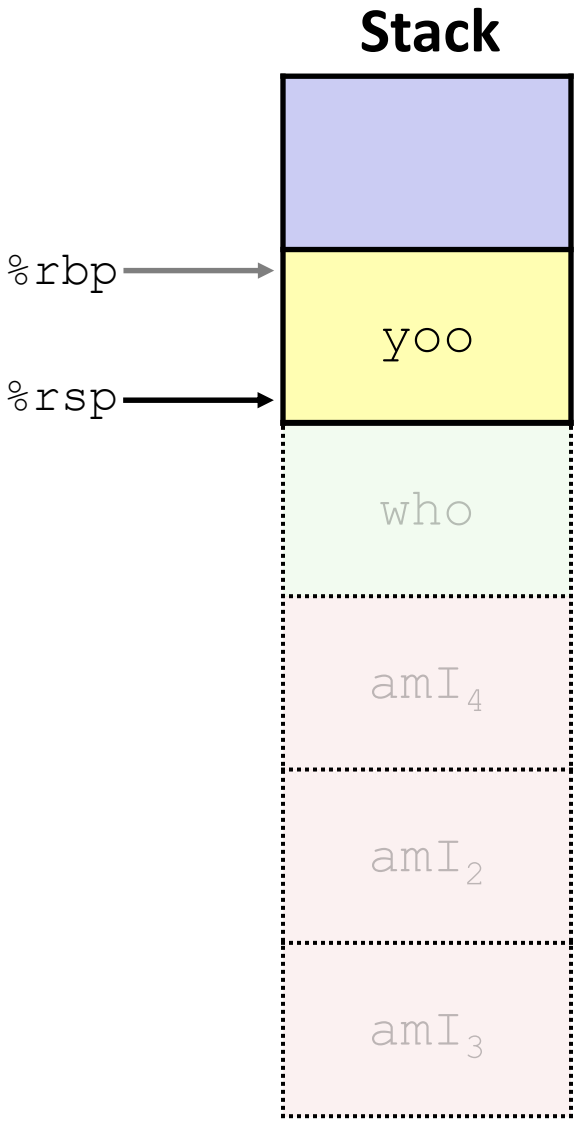




11) Return from call to who

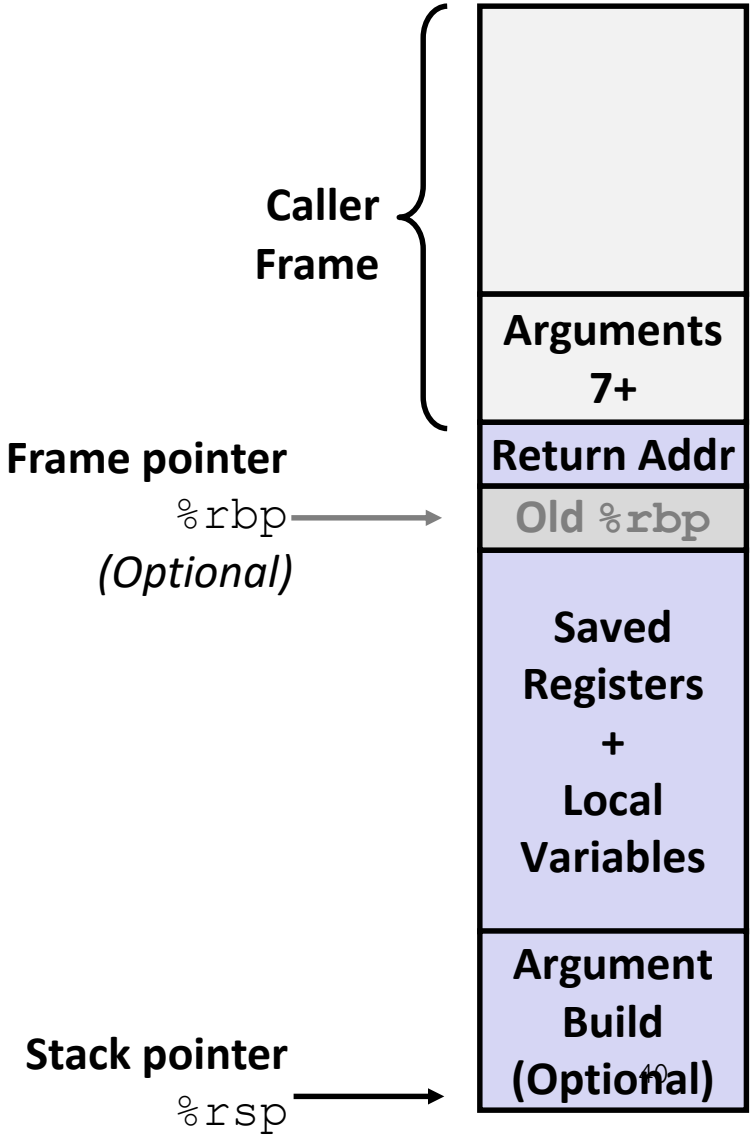






x86-64/Linux Stack Frame

- 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)



Question

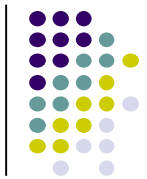
- Answer the following questions about when `main()` is run (assume `x` and `y` stored on the Stack):

```
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?

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
```

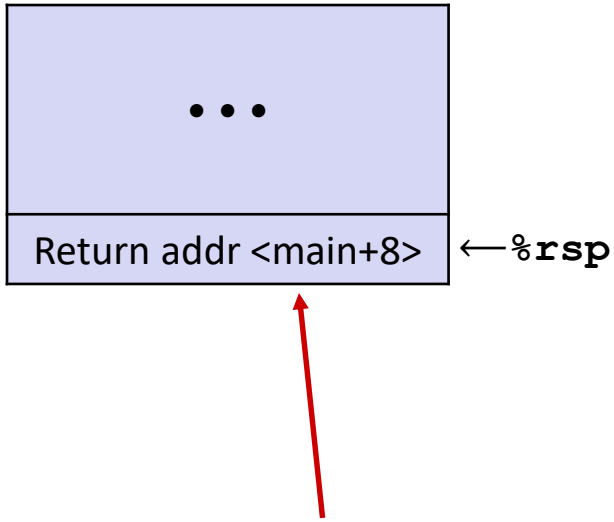
Register	Use(s)
%rdi	1st arg (p)
%rsi	2nd arg (val), y
%rax	x, return value

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)

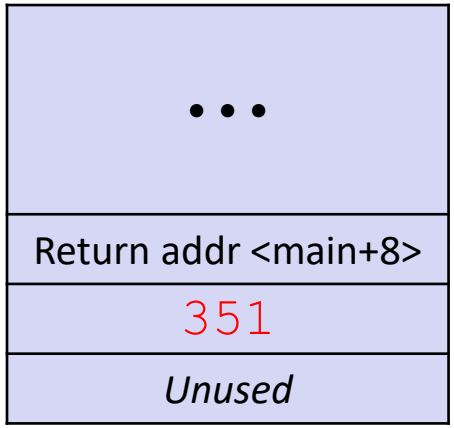
Procedure Call Example (step 1)

```
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
```

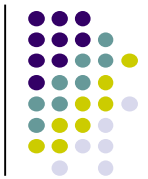
Allocate space for local vars

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)

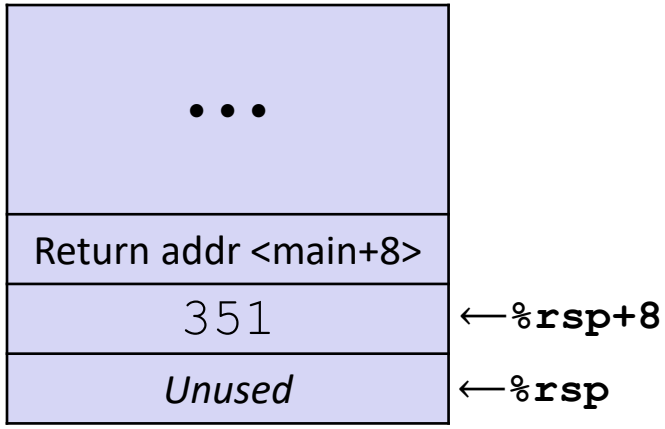


```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
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



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

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`.**

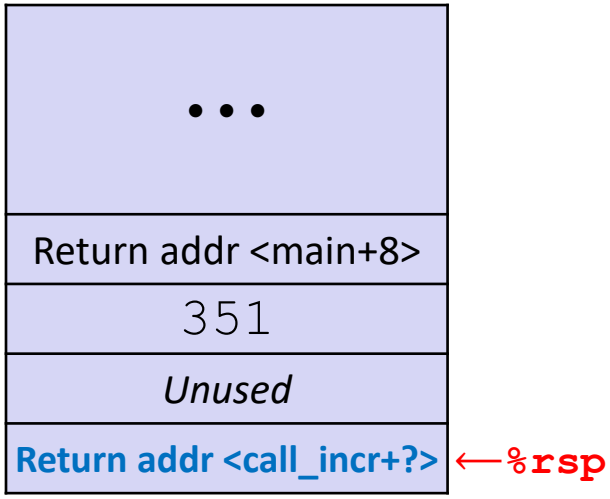
Procedure Call Example (step 3)

```
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 foll owing call to `increment`

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

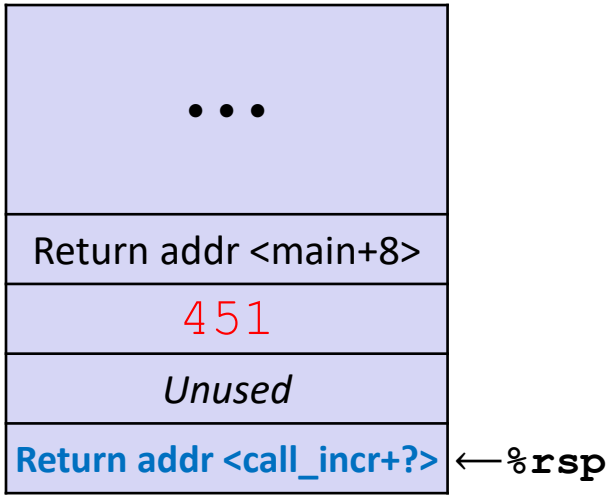
Procedure Call Example (step 4)

```
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

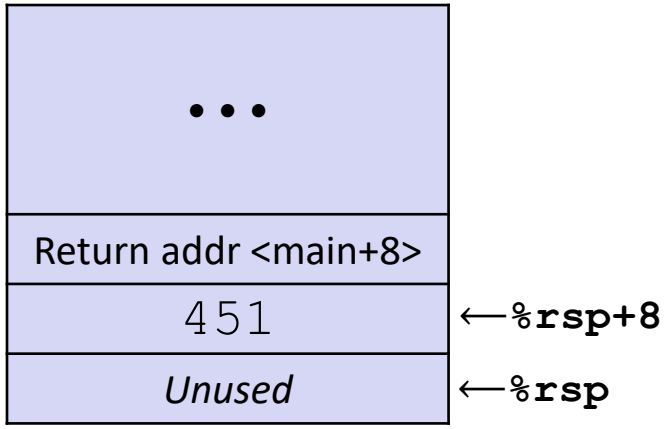
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	351

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 st ack

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

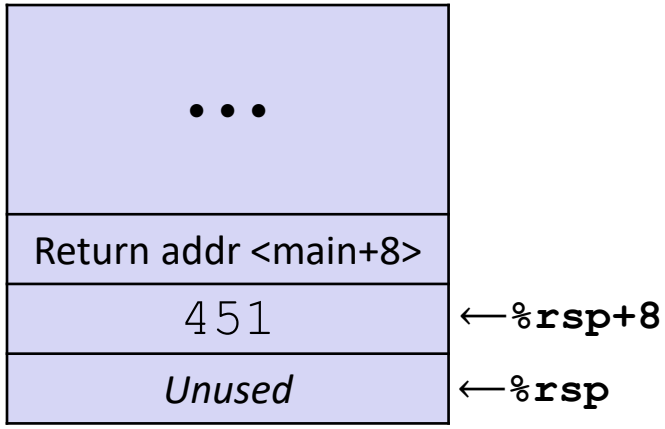
Procedure Call Example (step 6)

```
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

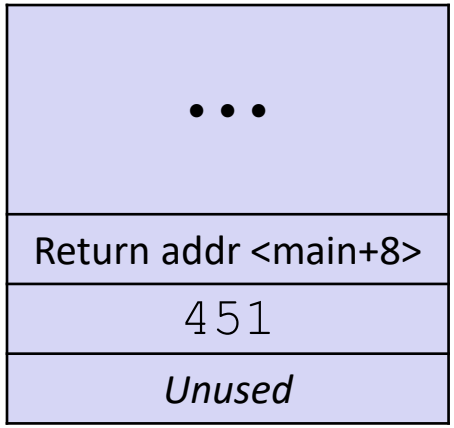
Procedure Call Example (step 7)

```
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



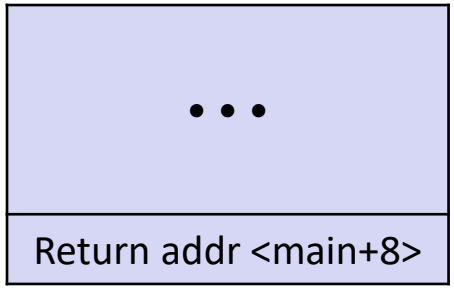
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

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`

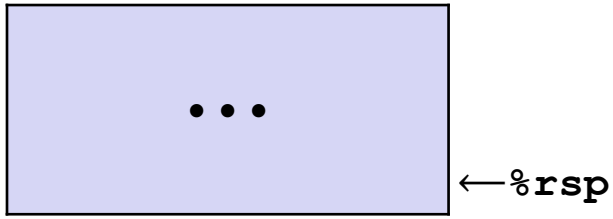
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

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

Outline

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

Register Saving Conventions

- When procedure **yoo** calls **who**:
  - **yoo** is the ***caller***
  - **who** is the ***callee***
- Can register be used for temporary storage?

```
yoo:
    . . .
    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

- ***"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

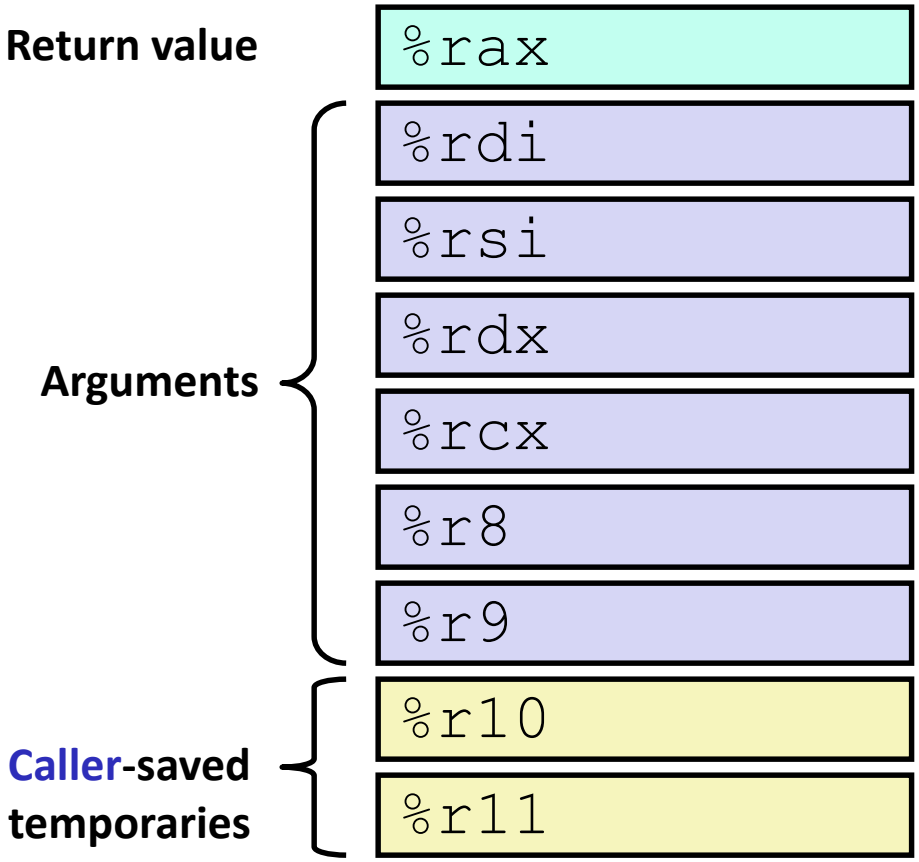
1) Parents (***caller***) leave for the weekend and give the keys to the house to their child (***callee***)
   - Being suspicious, they put away/hid the valuables (***caller****-saved*) before leaving
   - Warn child to leave the bedrooms untouched: "These rooms better look the same when we return!"

2) Child decides to throw a wild party (*computation*), spanning the entire house
   - To avoid being disowned, child moves all of 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 (***caller****-saved*) and continue with their lives

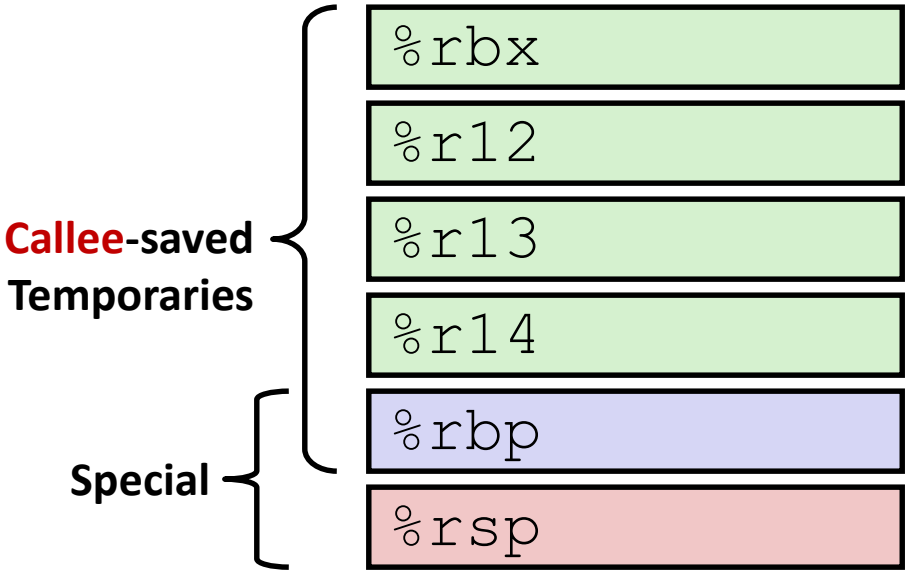
x86-64 Linux Register Usage, part1

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



x86-64 Linux Register Usage, part2

- `%rbx`, `%r12`, `%r13`, `%r14`
  - **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 64-bit Registers: Usage Conventions

%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

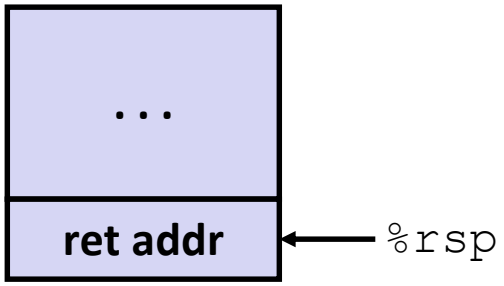
%r8	Argument #5 - Caller saved
%r9	Argument #6 - Caller saved
%r10	Caller saved
%r11	Caller Saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

Callee-Saved Example (step 1)

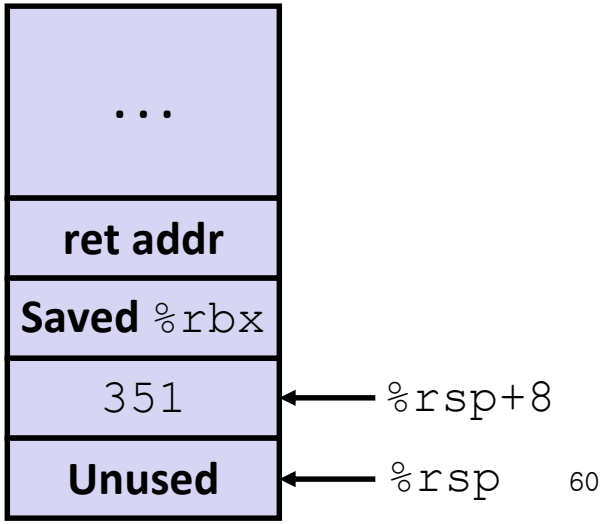
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

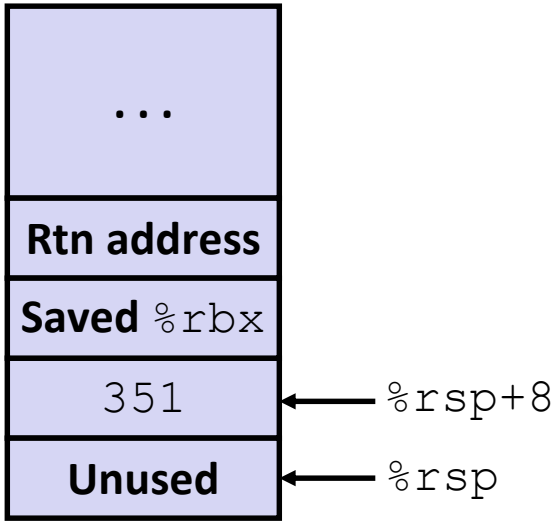


Callee-Saved Example (step 2)

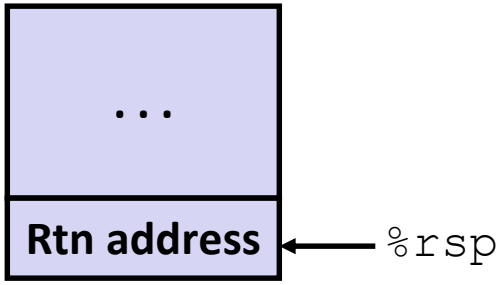
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Stack Structure



Pre-return Stack Structure



Why Caller *and* Callee Saved?

- **We want *one* calling convention to simply separate implementation details between caller and callee**

- **In general, neither caller-save nor callee-save is "best":**
  - If caller isn't using a register, caller-save is better
  - If callee doesn't need a register, callee-save is better
  - If "do need to save", callee-save generally makes smaller programs
    - Functions are called from multiple places

- **So… "some of each" and compiler tries to "pick registers" that minimize amount of saving/restoring**

Register Conventions Summary

- **Caller**-saved register values need to be pushed onto the stack before making a procedure call *only if the Caller needs that value later*
  - **Callee** may change those register values
- **Callee**-saved register values need to be pushed onto the stack *only if the Callee intends to use those registers*
  - **Caller** expects unchanged values in those registers

- Don't forget to restore/pop the values later!

Outline

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

Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Compiler Explorer:

https://godbolt.org/g/4ZJbz1

- Compiled with `-O1` for brevity instead of `-Og`
- Try `-O2` instead!

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

Recursive Function: Base Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

Trick because some AMD hardware doesn't like jumping to `ret`

Recursive Function: Callee Register Save

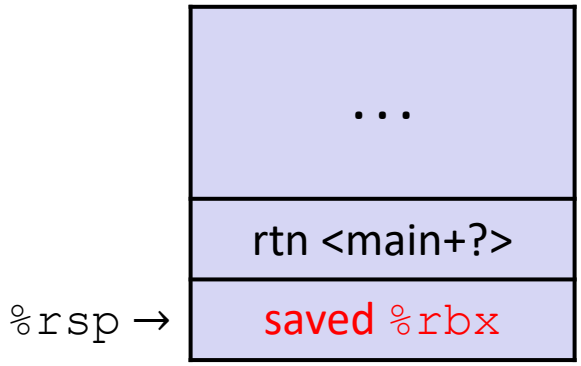
```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rdi	x	Argument

Need original value of `x` *after* recursive call to `pcount_r`.

"Save" by putting in `%rbx` (**callee** saved), but need to save old value of `%rbx` before you change it.

The Stack



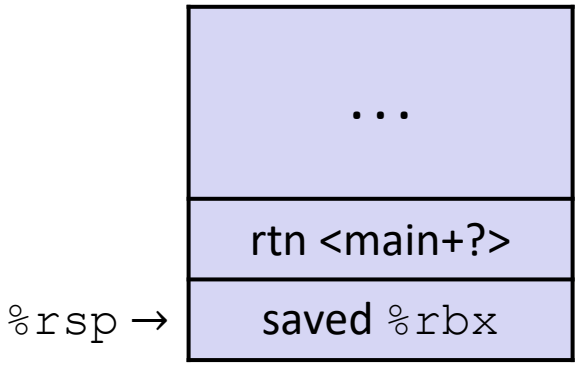

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

Recursive Function: Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rdi	x (new)	Argument
%rbx	x (old)	Callee saved

The Stack



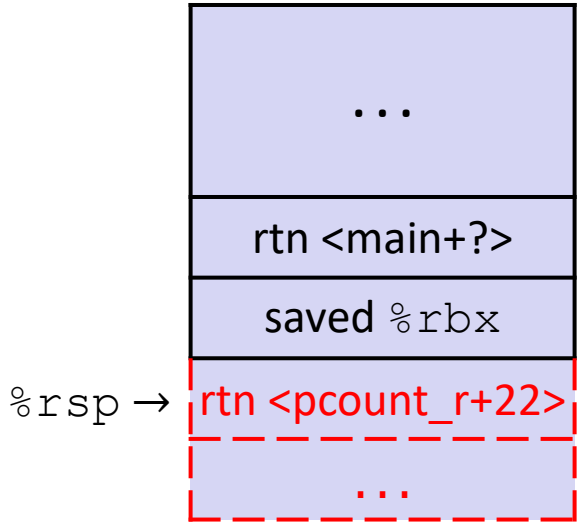

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

Recursive Function: Call (ink walkthrough)

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

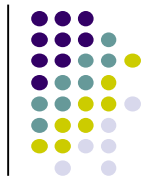
Register	Contents
%rdi	
%rbx	
%rax	

The Stack




```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

Recursive Function: Call

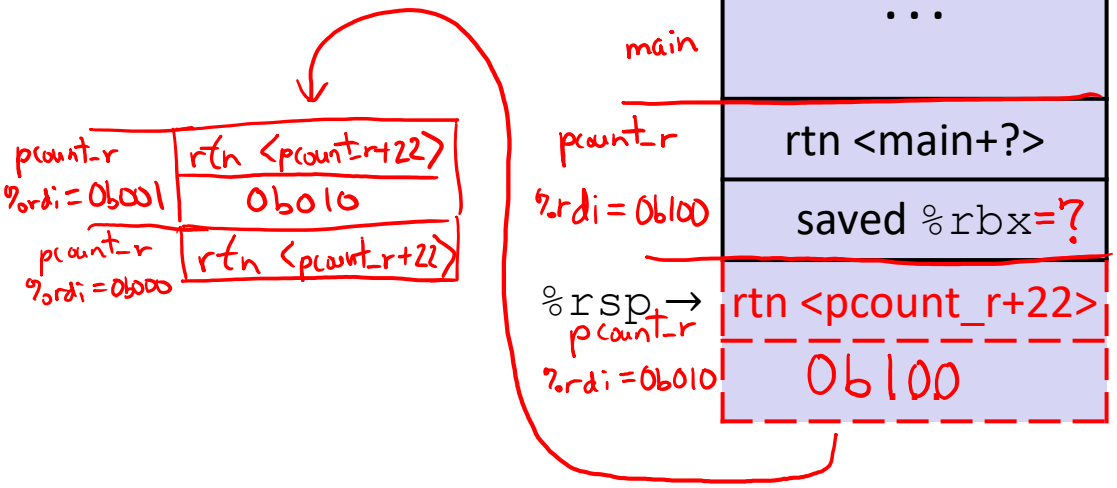


```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rax	Recursive call return value	Return value
%rbx	x (old)	Callee saved

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```


if original x = 0b100:

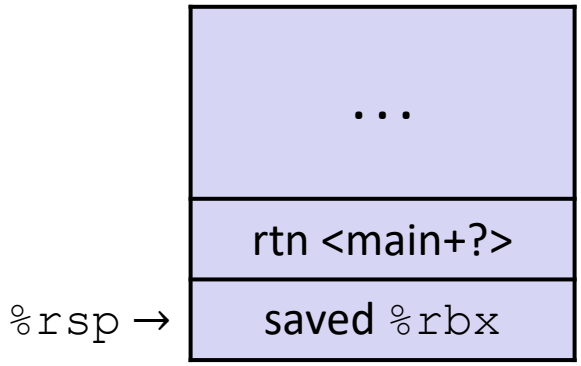


Recursive Function: Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rax	Return value	Return value
%rbx	x&1	Callee saved

The Stack

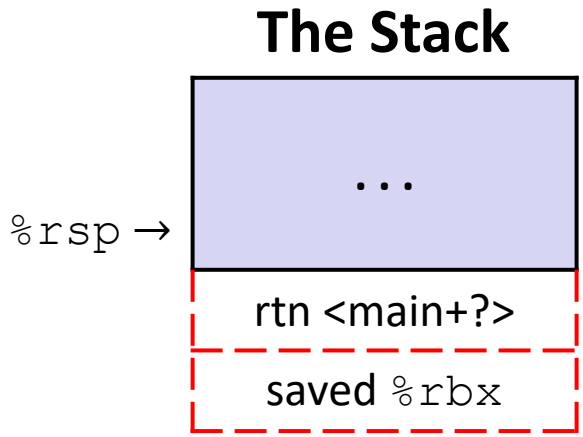



```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

Recursive Function: Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rax	Return value	Return value
%rbx	Previous %rbx value	Callee restored



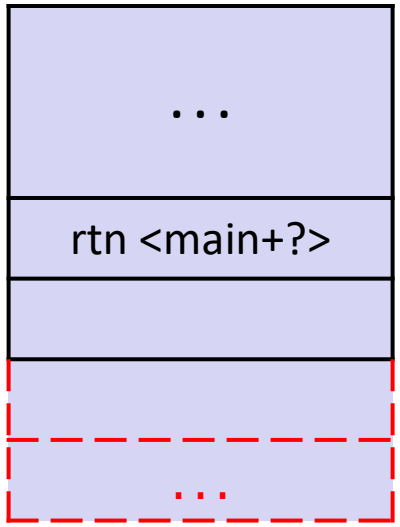

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

Recursive Function: Handout

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Register	Contents
%rdi	
%rbx	
%rax	

The Stack




```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

Observations About Recursion

- Handled Without Special Consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return address
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out

- Also works for mutual recursion
  - P calls Q; Q calls P

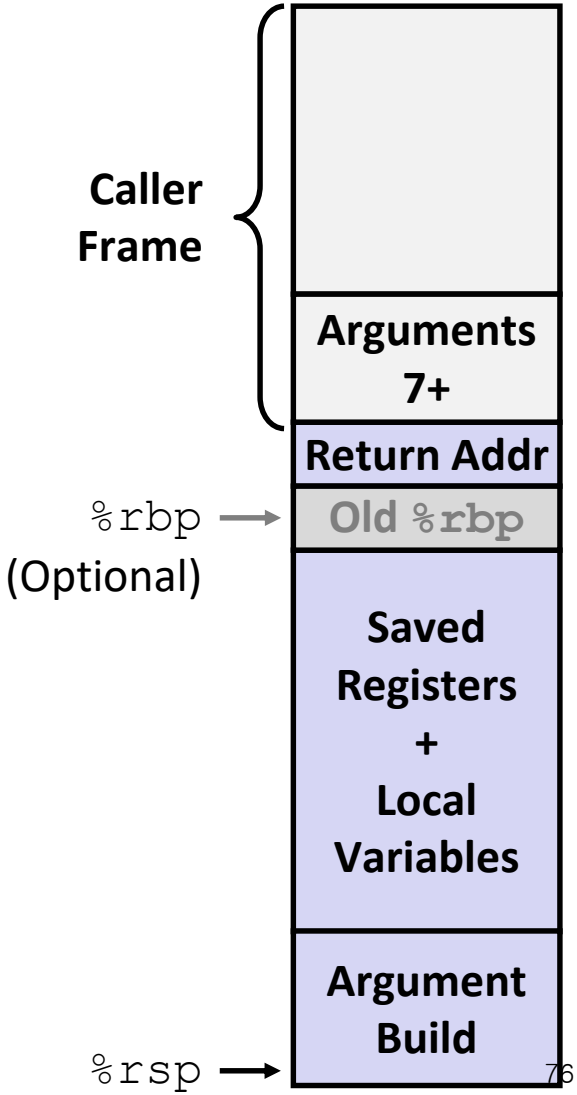
x86-64 Stack Frames

- Many x86-64 procedures have a minimal stack frame
  - Only return address is pushed onto the stack when calling procedure is called

- A procedure *needs* to grow its stack frame when it:
  - Has too many local variables to hold in **caller**-saved registers
  - Has local variables that are arrays or structs
  - Uses `&` to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Is using **caller**-saved registers and then calls a procedure
  - Modifies/uses **callee**-saved registers

x86-64 Procedure Summary

- Important Points
  - Procedures are a combination of *instructions* and *conventions*
    - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call/return
    - If P calls Q, then Q returns before P
  - Recursion handled by normal calling conventions
- Heavy use of registers
  - Faster than using memory
  - Use limited by data size and conventions
- Minimize use of the Stack

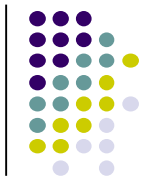


One more x86-64 example

- Example of passing more than 6 parameters and passing addresses of local variables

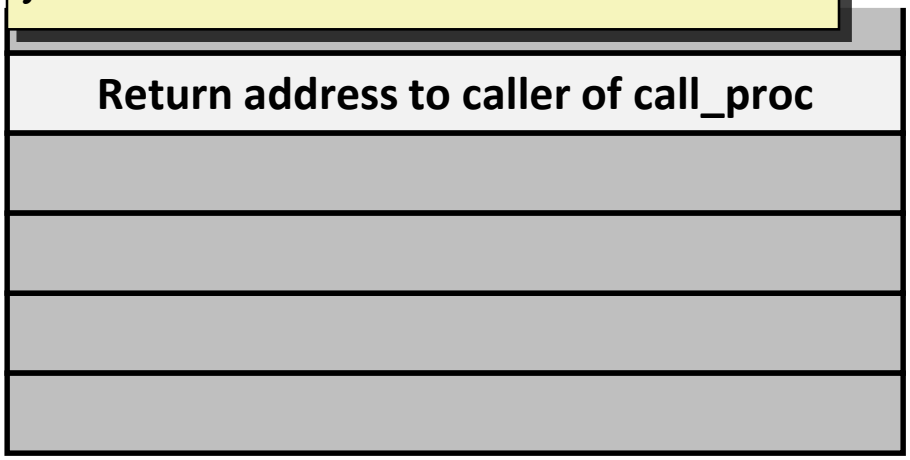
- The following example, along with a brief re-cap of x86-64 calling conventions is in this video:

x86-64 Example (1)



```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

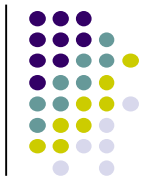
```
call_proc:
  subq   $32,%rsp
  movq   $1,16(%rsp)  # x1
  movl   $2,24(%rsp)  # x2
  movw   $3,28(%rsp)  # x3
  movb   $4,31(%rsp)  # x4
  • • •
```



←%rsp

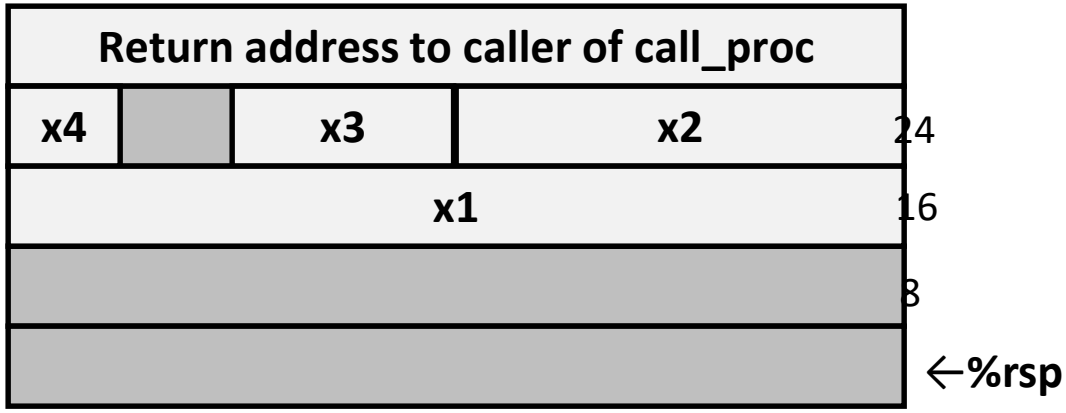
Note: Details may vary depending on compiler.

x86-64 Example (2) – Allocate local vars

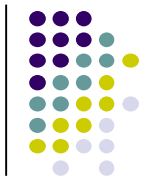


```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  subq  $32,%rsp
  movq  $1,16(%rsp)  # x1
  movl  $2,24(%rsp)  # x2
  movw  $3,28(%rsp)  # x3
  movb  $4,31(%rsp)  # x4
  • • •
```

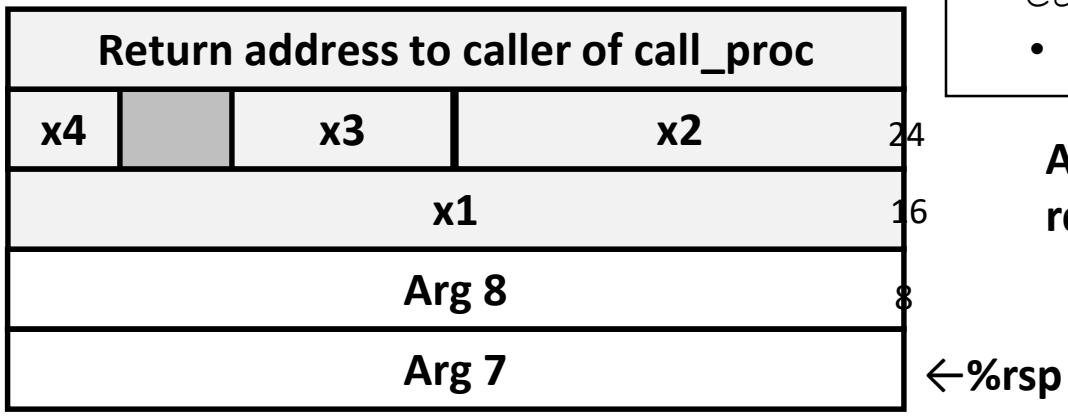


x86-64 Example (3) – setup params to proc



```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

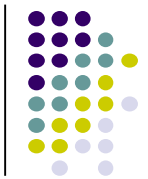
```
call_proc:
  . . .
  leaq  24(%rsp),%rcx # %rcx=&x2
  leaq  16(%rsp),%rsi # %rsi=&x1
  leaq  31(%rsp),%rax # %rax=&x4
  movq  %rax,8(%rsp)  # arg8=&4
  movl  $4,(%rsp)     # arg7=4
  leaq  28(%rsp),%r9  # %r9=&x3
  movl  $3,%r8d   # %r8 = 3
  movl  $2,%edx   # %rdx = 2
  movq  $1,%rdi   # %rdi = 1
  call  proc
  . . .
```



Arguments passed in (in order):
rdi, rsi, rdx, rcx, r8, r9

Same instructions as in video, just a different order.

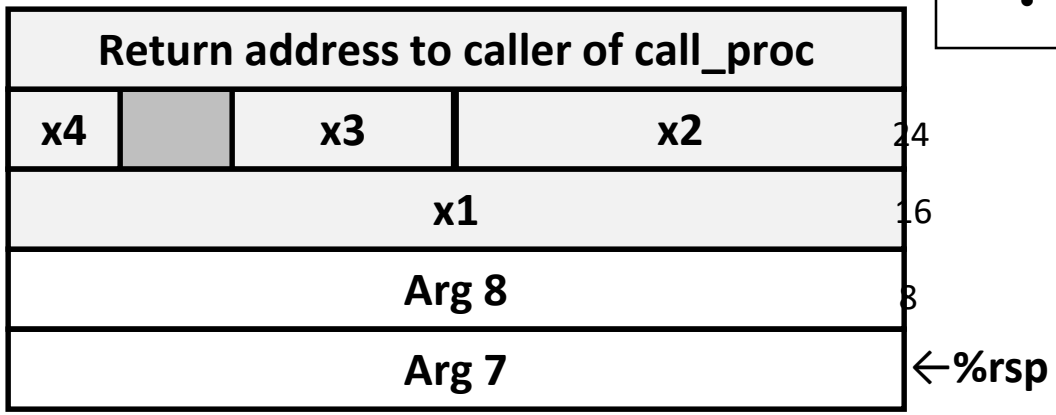
x86-64 Example (4) – setup params to proc



```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  . . .
  leaq  24(%rsp),%rcx
  leaq  16(%rsp),%rsi
  leaq  31(%rsp),%rax
  movq  %rax,8(%rsp)
  movl  $4,(%rsp)
  leaq  28(%rsp),%r9
  movl  $3,%r8d
  movl  $2,%edx
  movq  $1,%rdi
  call  proc
  . . .
```

Note sizes

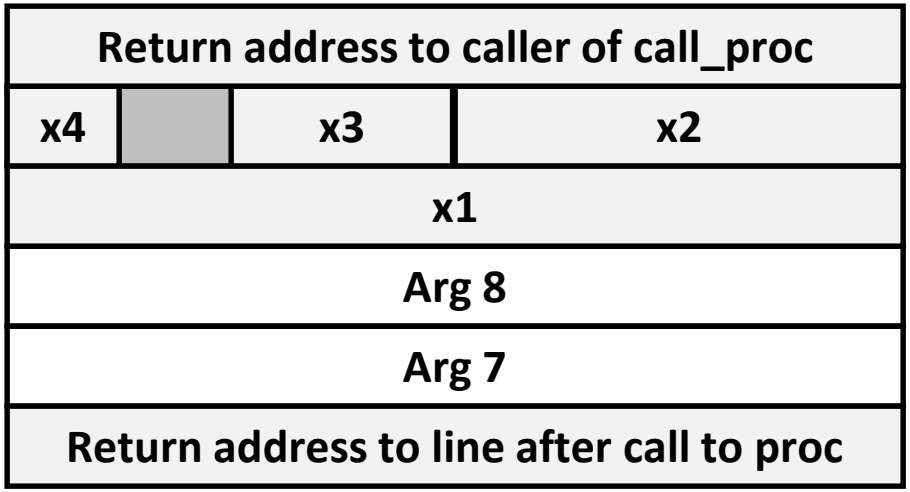


Arguments passed in (in order): rdi, rsi, rdx, rcx, r8, r9

x86-64 Example (5) – call proc

```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

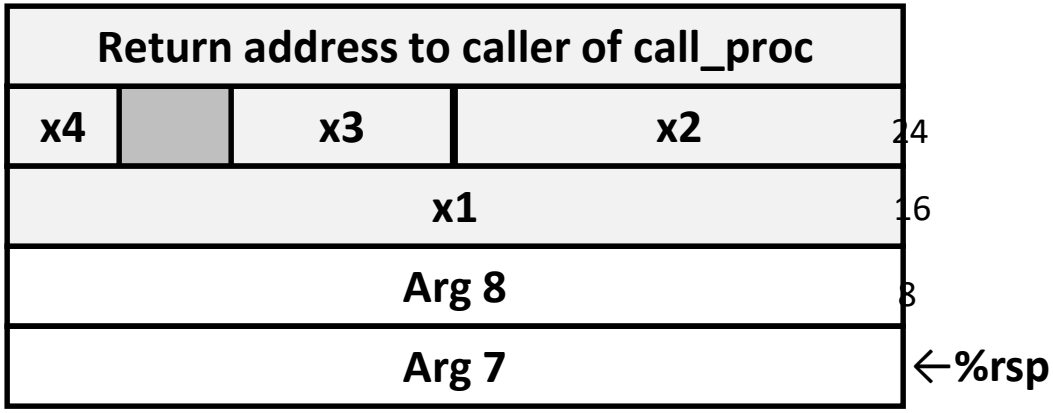
```
call_proc:
  . . .
  leaq  24(%rsp),%rcx
  leaq  16(%rsp),%rsi
  leaq  31(%rsp),%rax
  movq  %rax,8(%rsp)
  movl  $4,(%rsp)
  leaq  28(%rsp),%r9
  movl  $3,%r8d
  movl  $2,%edx
  movq  $1,%rdi
  call  proc
  . . .
```



x86-64 Example (6) – after call to proc

```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

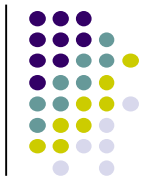
```
call_proc:
  . . .
  movswl 28(%rsp),%eax # %eax=x3
  movsbl 31(%rsp),%edx # %edx=x4
  subl   %edx,%eax # %eax=x3-x4
  cltq
  movslq 24(%rsp),%rdx # %rdx=x2
  addq   16(%rsp),%rdx #%rdx=x1+x2
  imulq  %rdx,%rax # %rax=rax*rdx
  addq   $32,%rsp
  ret
```



movs__:
move and sign extend

cltq:
sign extend %eax into %rax
(special-case to save space)

x86-64 Example (7) – de-allocate local vars



```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  . . .
  movswl 28(%rsp),%eax
  movsbl 31(%rsp),%edx
  subl   %edx,%eax
  cltq
  movslq 24(%rsp),%rdx
  addq   16(%rsp),%rdx
  imulq  %rdx,%rax
  addq   $32,%rsp
  ret
```

Return address to caller of call_proc

←%rsp

Q&A