Procedures and the **Call Stack**

Topics

- Procedures
- Call stack
- Procedure/stack instructions
- Calling conventions
- Register-saving conventions

Why Procedures?

Why functions? Why methods?

```
int contains_char(char* haystack, char needle) {
  while (*haystack != '\0') {
    if (*haystack == needle) return 1;
    haystack++;
  }
  return 0;
}
```

Procedural Abstraction

Implementing Procedures

How does a caller pass **arguments** to a procedure?

How does a caller get a **return value** from a procedure?

Where does a procedure store **local variables**?

How does a procedure know **where to return**
(what code to execute next when done)?

How do procedures **share limited registers** and **memory**?

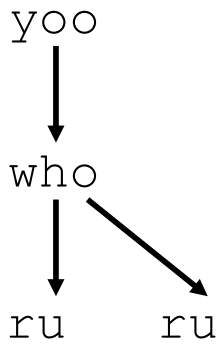
Call Chain

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

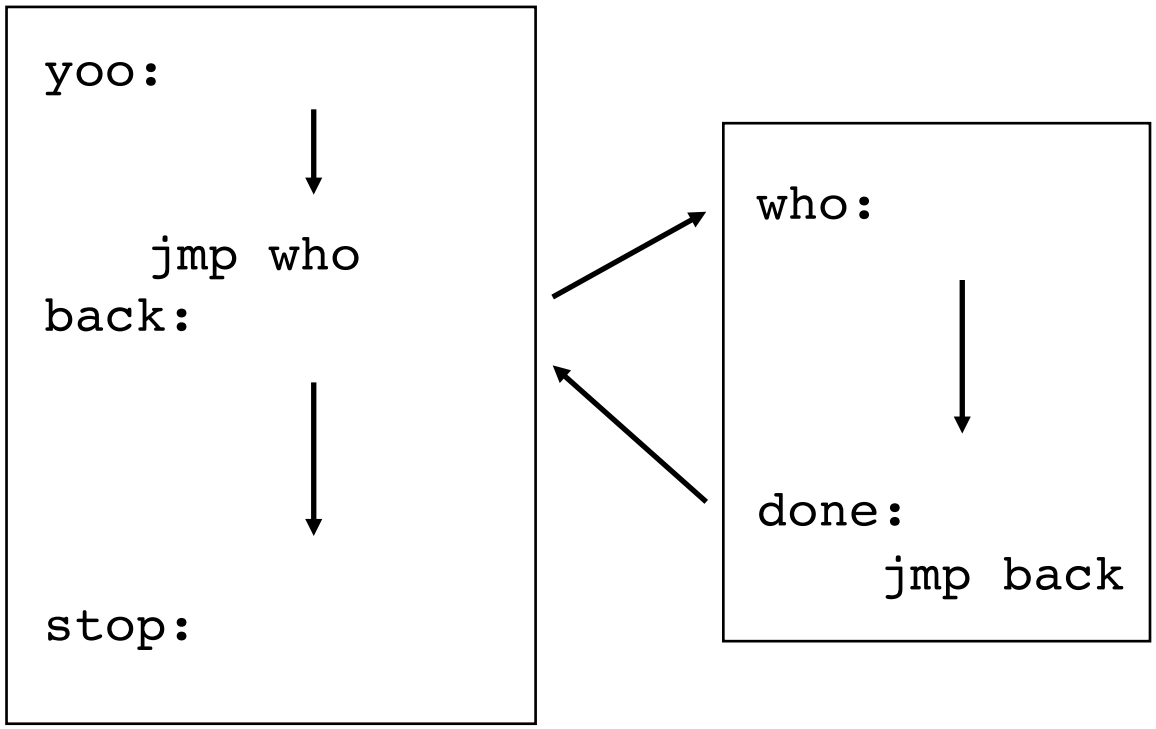
```
who(…)
{
    • • •
    ru();
    • • •
    ru();
    • • •
}
```

```
ru(…)
{
    •
    •
    •
}
```

Example
Call Chain



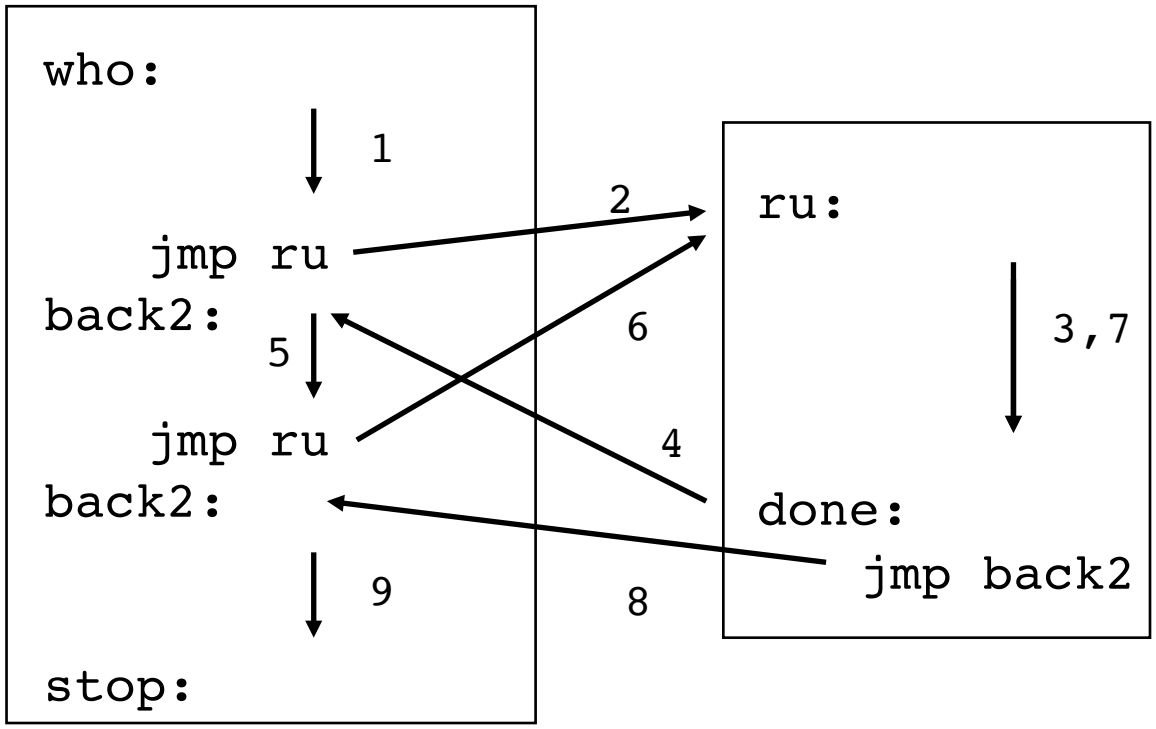
First Try (broken)



What if we want to call a function from multiple places in the code?

First Try (broken)

What if we want to call a function from multiple places in the code?



Implementing Procedures

How does a caller pass **arguments** to a procedure?

How does a caller get a **return value** from a procedure?

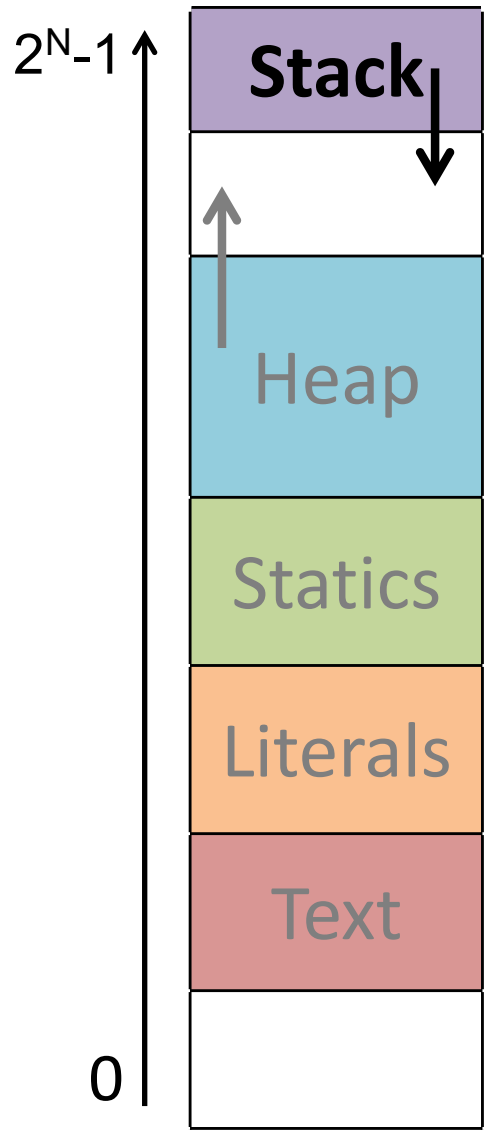
Where does a procedure store **local variables**?

How does a procedure know **where to return**
(what code to execute next when done)?

How do procedures **share limited registers** and **memory**?

All these need **separate storage *per call!***
(not just per procedure)

Memory Layout

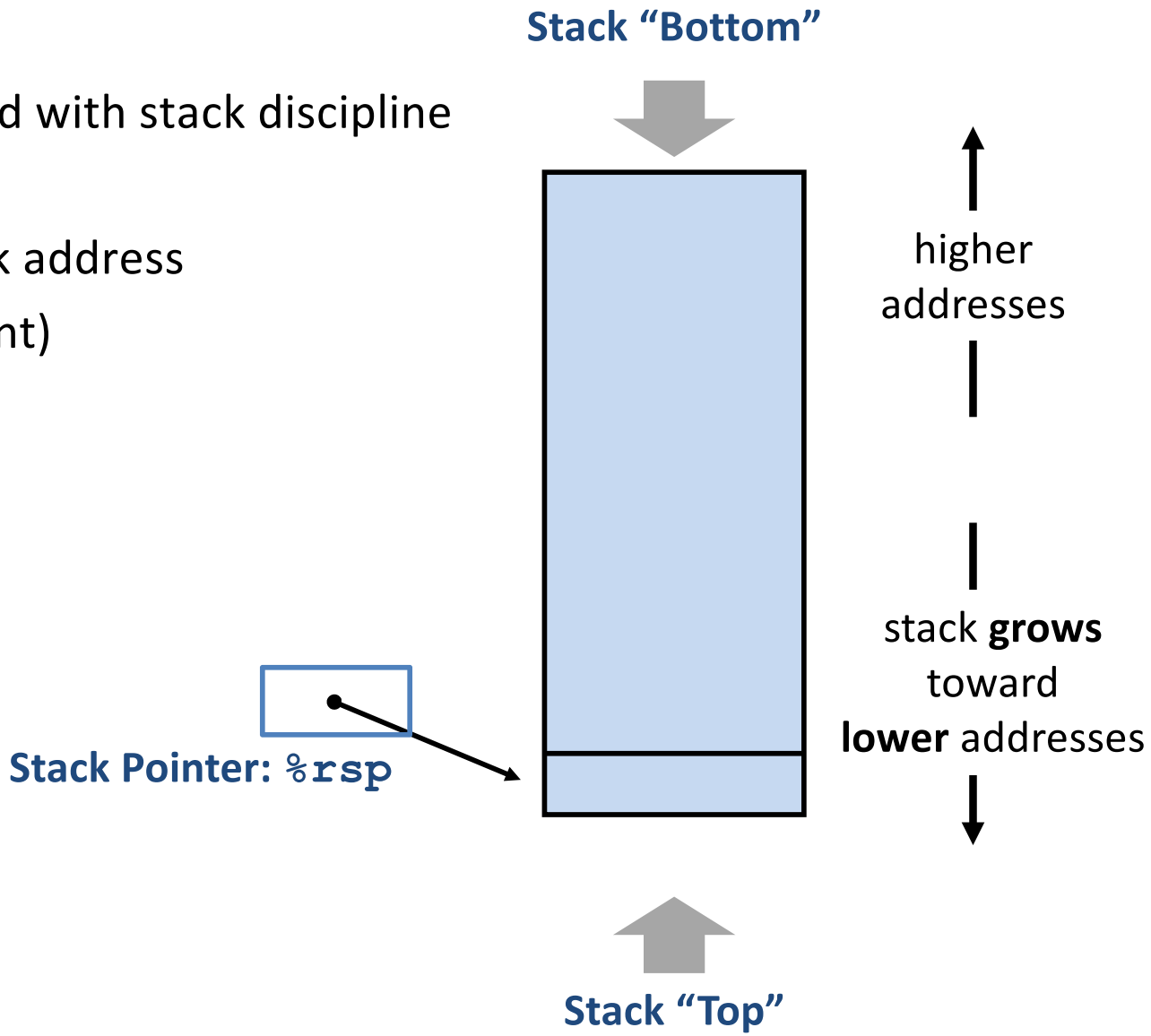


Perm	Contents	Managed by	Initialized
RW	**Procedure context**	**Compiler**	**Run-time**
RW	Dynamic data structures	Programmer, malloc/free, new/GC	Run-time
RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
R	String literals	Compiler/ Assembler/Linker	Startup
X	Instructions	Compiler/ Assembler/Linker	Startup

Call Stack

Memory region managed with stack discipline

`%rsp` holds lowest stack address
(address of "top" element)



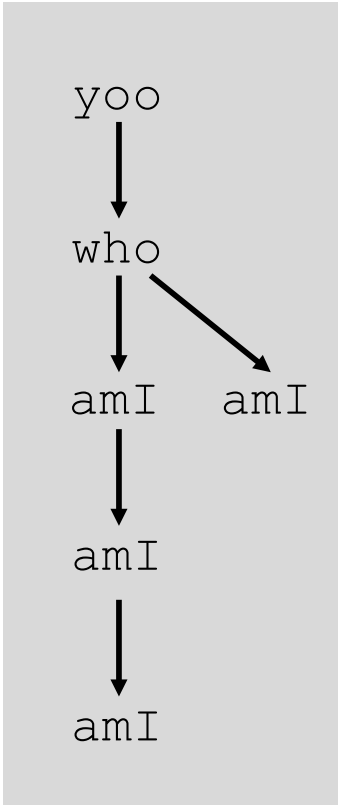
Call Chain Example

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

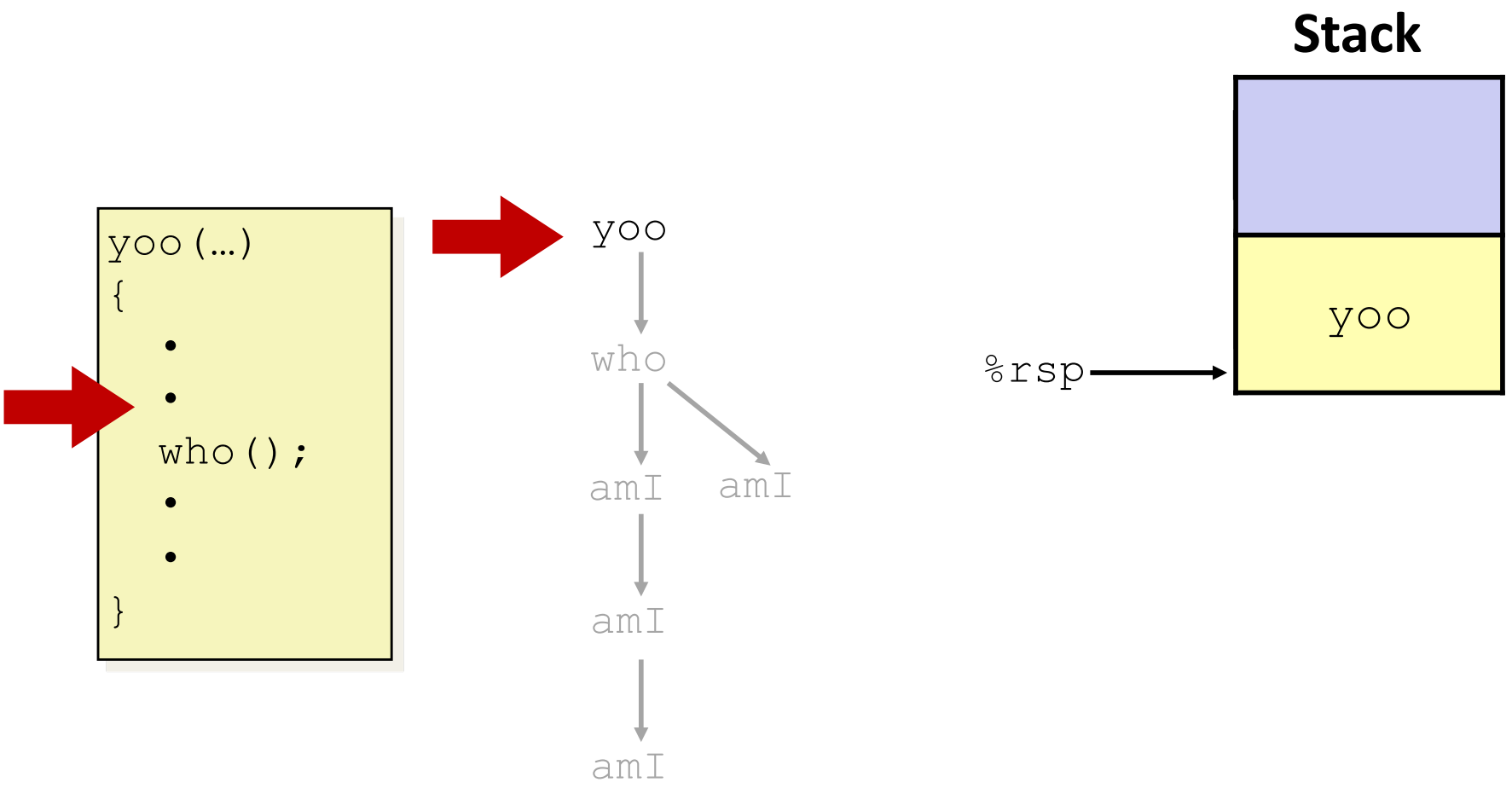
```
who(…)
{
    •
    amI();
    •
    amI();
    •
}
```

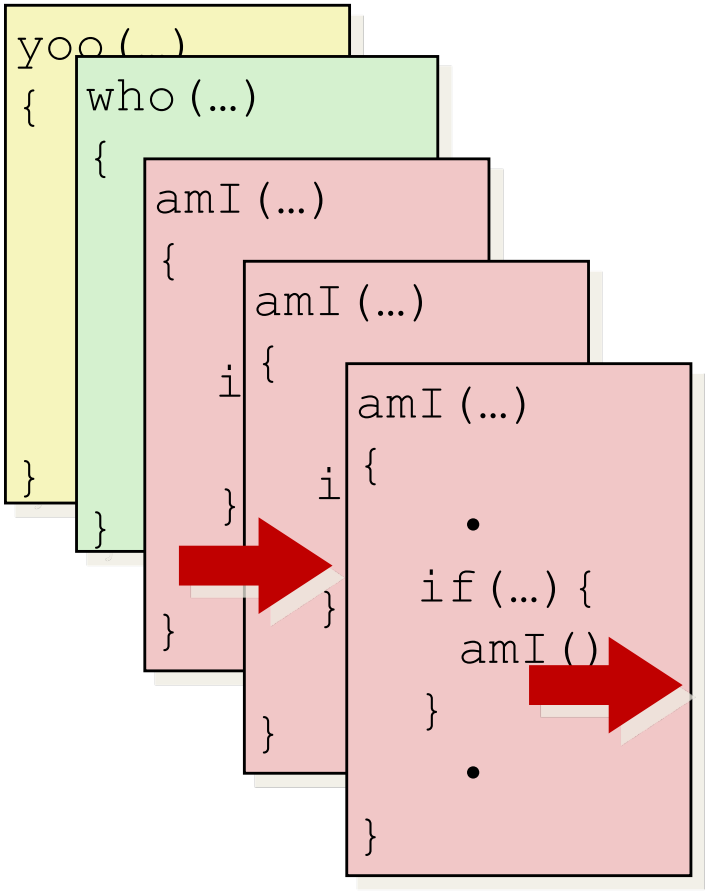
```
amI(…)
{
    •
   if(…){
     amI()
   }
    •
}
```

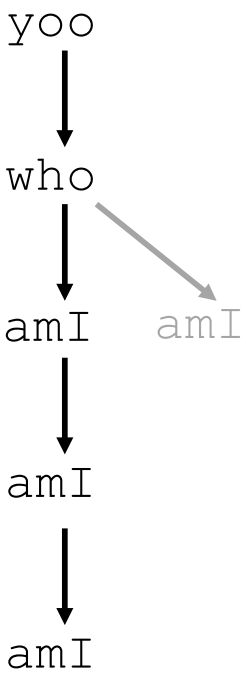
Example
Call Chain

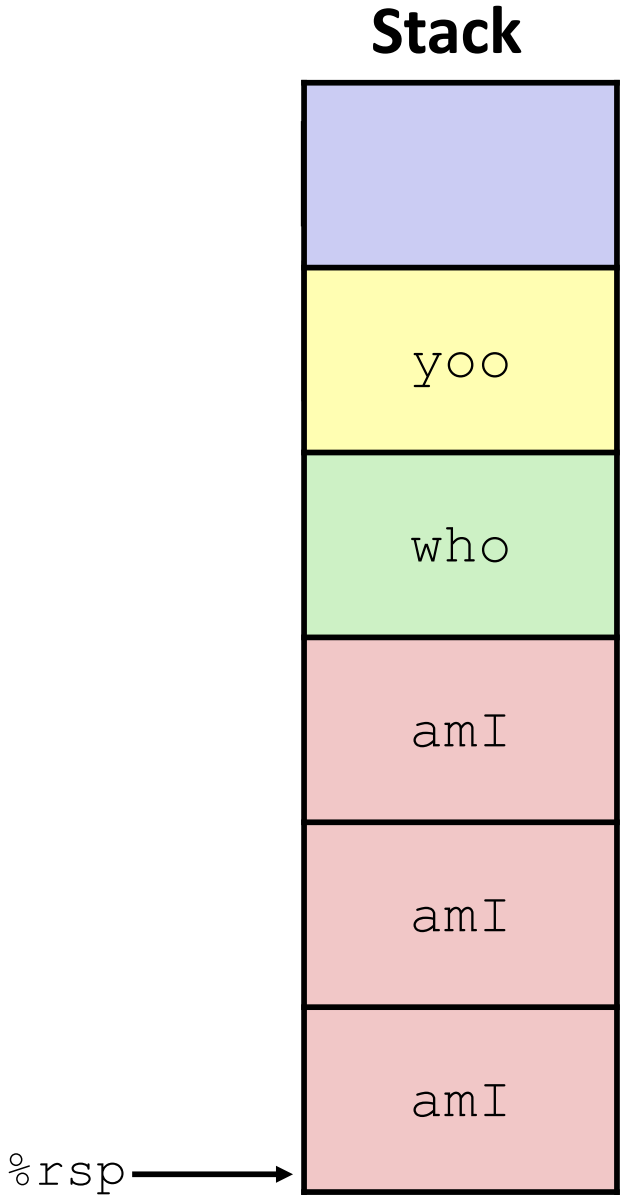


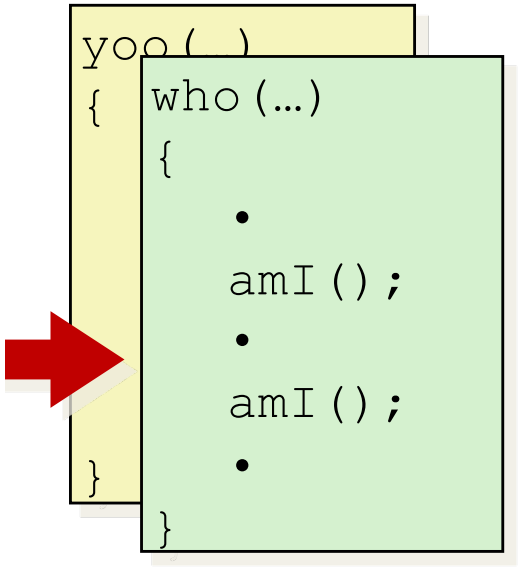
Procedure `amI` is recursive
(calls itself)

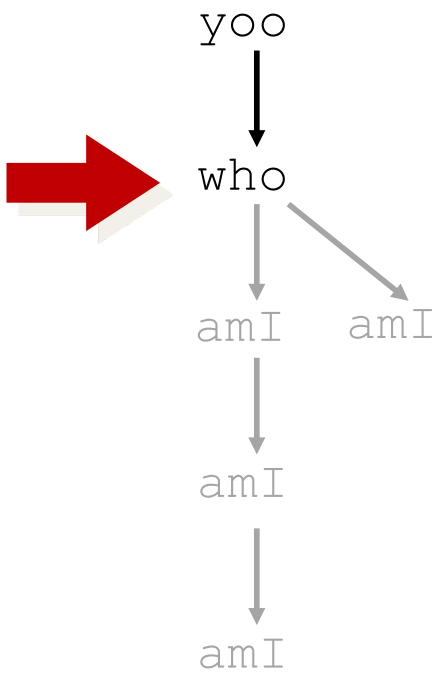

yoo(…)
{
•
•
who();
•
•
}
yoo
who
amI
amI
amI
amI
Stack
yoo
%rsp

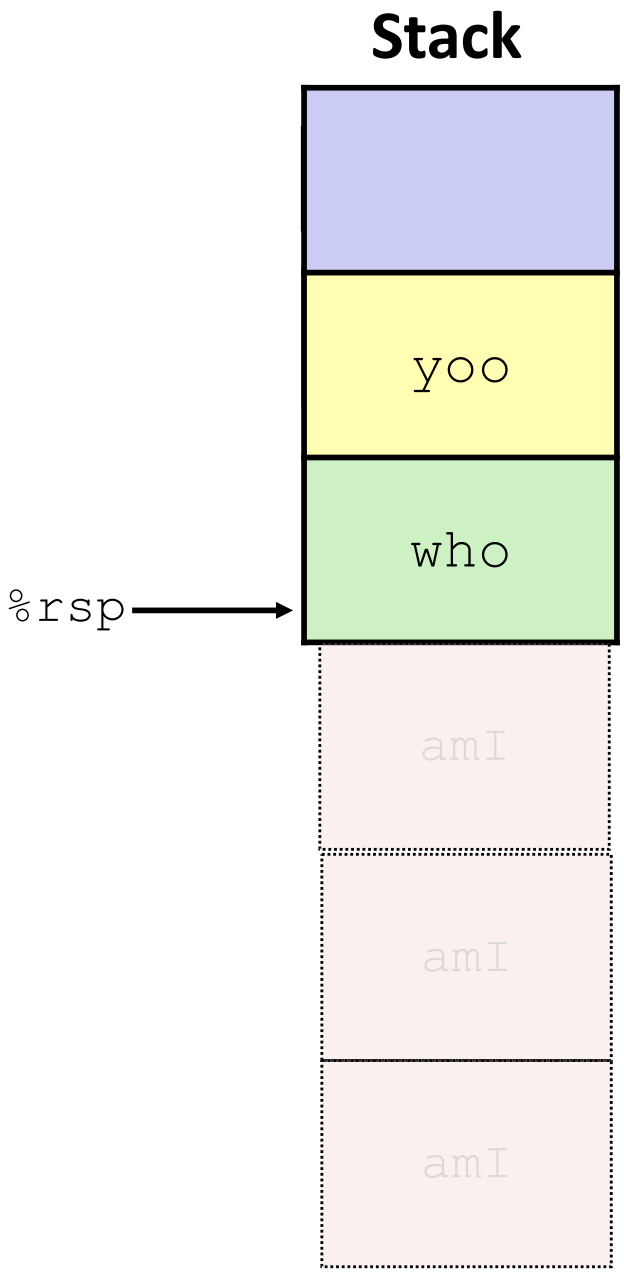

yoo ( )
{
}
who (…)
{
}
amI (…)
{
}
amI (…)
{
}
amI (…)
{
if (…) {
amI ()
}
}

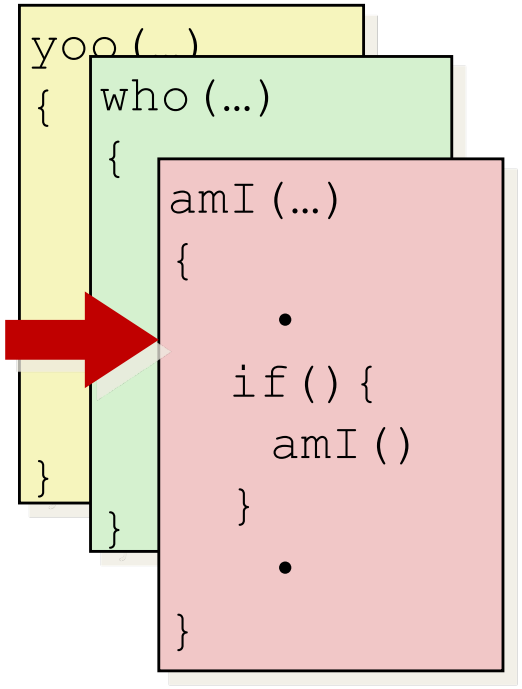

yoo
who
amI
amI
amI
amI

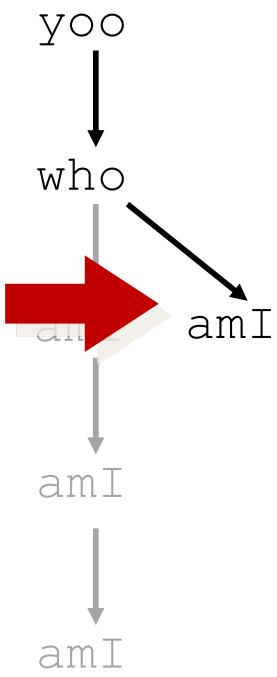

Stack
yoo
who
amI
amI
amI
%rsp

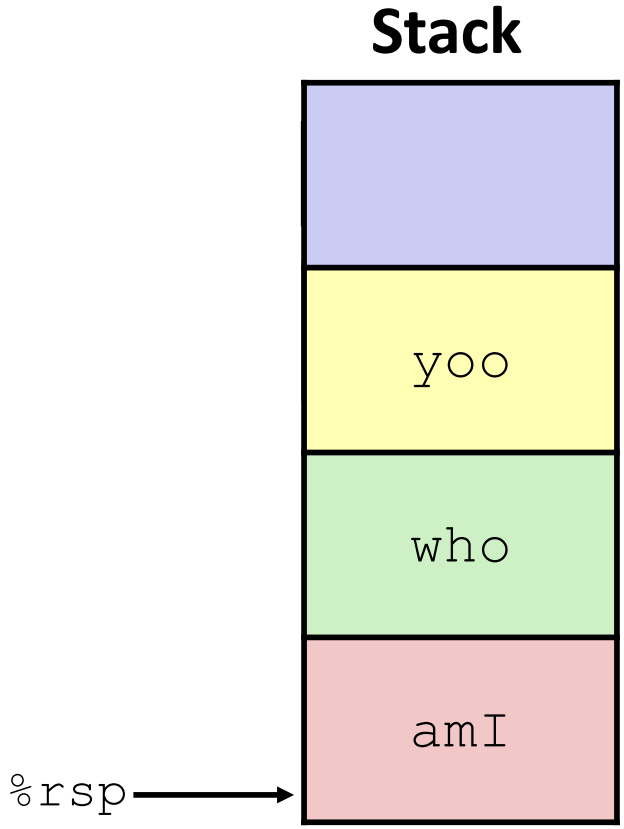

yoo( )
{
}
who(…)
{
•
amI();
•
amI();
•
}

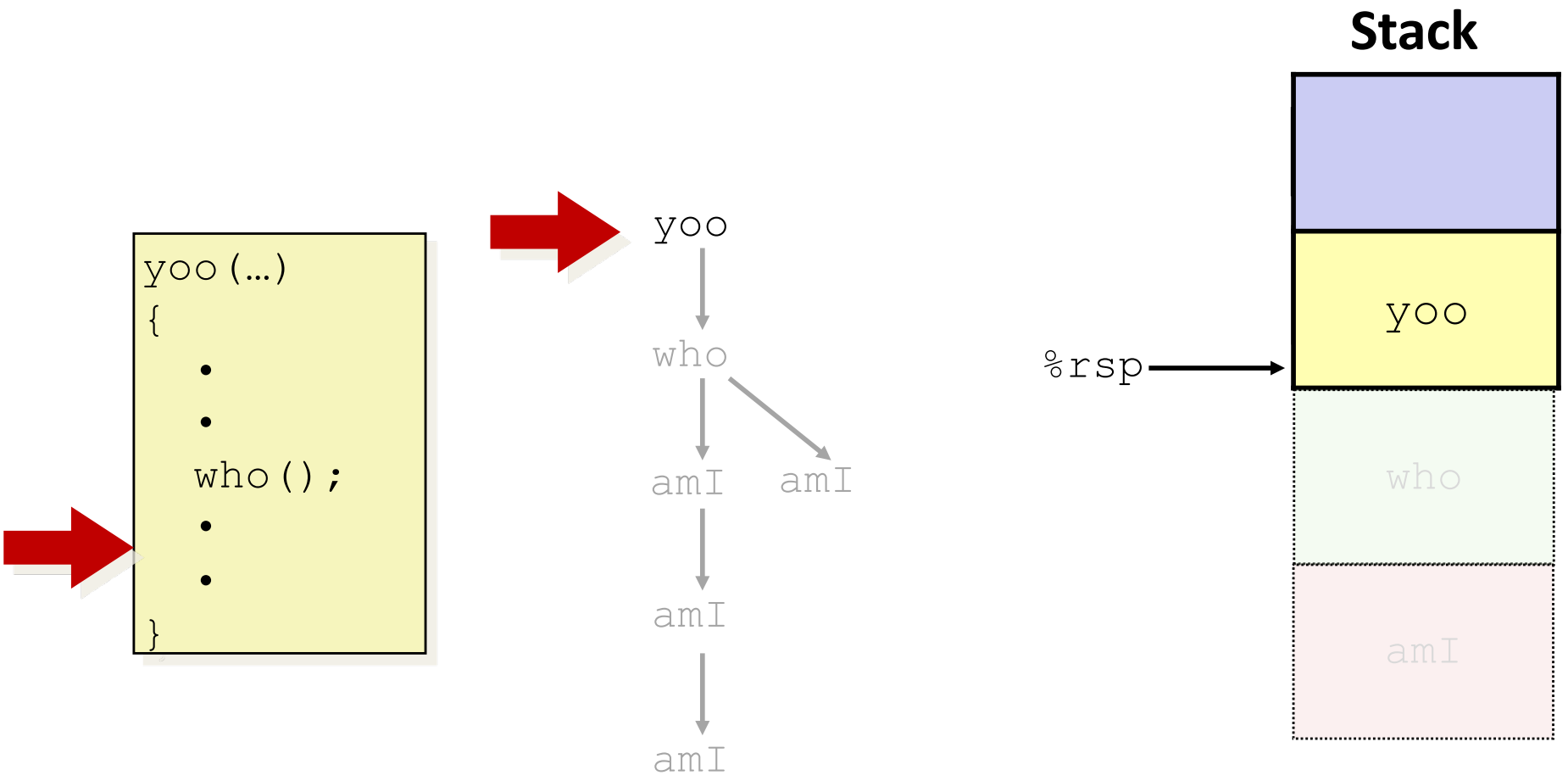

yoo
who
amI
amI
amI
amI


Stack
yoo
who
%rsp
amI
amI
amI

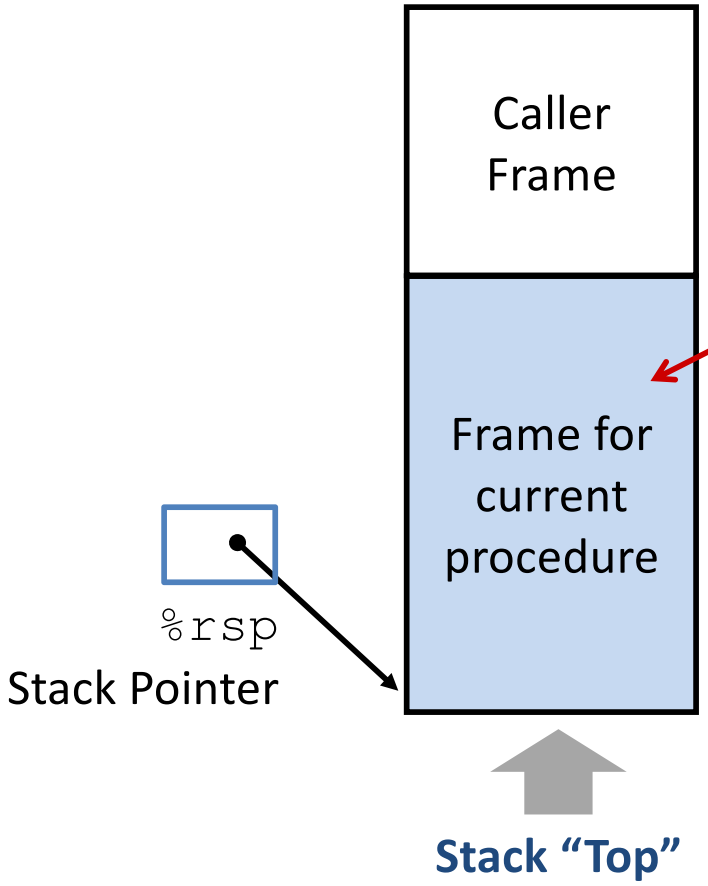

yoo( )
{
who(…)
{
amI(…)
{
•
if(){
amI()
}
•
}
}
}


yoo
who
amI
amI
amI


Stack
yoo
who
amI
%rsp


yoo(…)
{
•
•
who();
•
•
}
yoo
who
amI
amI
amI
amI
Stack
%rsp
yoo
who
amI

Stack frames support procedure calls.



Contents

- Local variables
- Function arguments (after first 6)
- Return information
- Temporary space

Management

- Space allocated when procedure is entered
  - "Setup" code
- Space deallocated before return
  - "Finish" code

Why not just give every *procedure* a permanent chunk of memory to hold its local variables, etc?

Code Examples

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

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
```

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

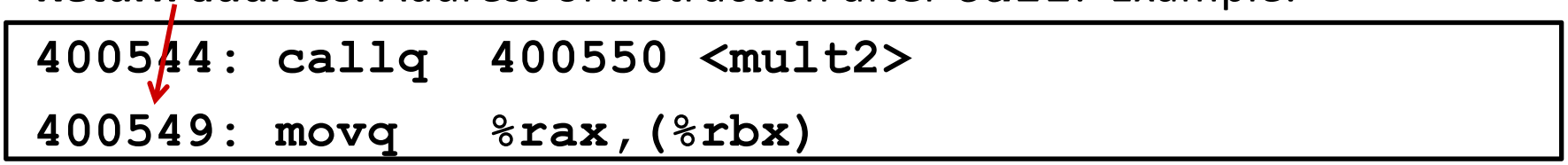
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax    # a
  400553:  imul   %rsi,%rax    # a * b
  400557:  retq                # Return
```

Procedure Control Flow Instructions

Procedure call: `callq` *`label`*

1. Push return address on stack
2. Jump to *`label`*

Return address: Address of instruction after `call`. Example:



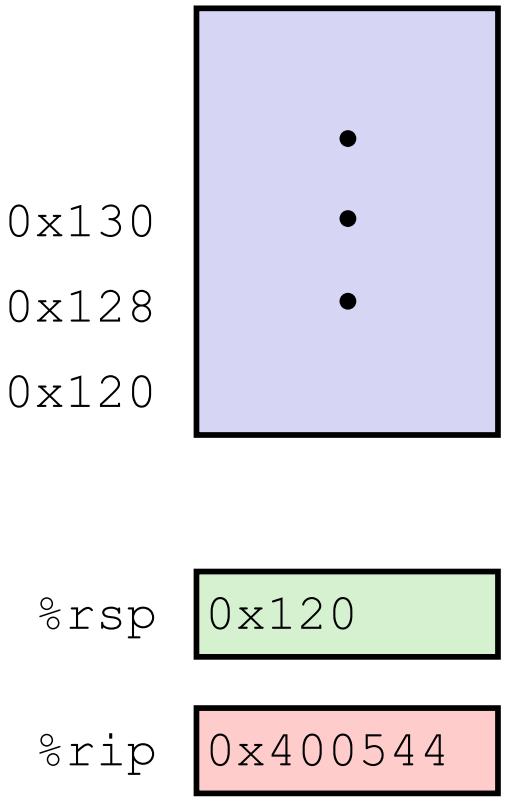
Procedure return: `retq`

1. Pop return address from stack
2. Jump to address

Call Example (step 1)

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

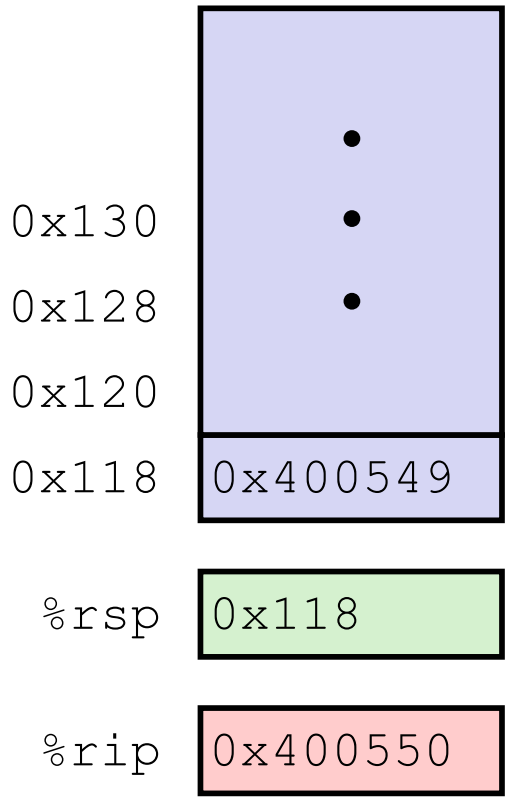
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Call Example (step 2)

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

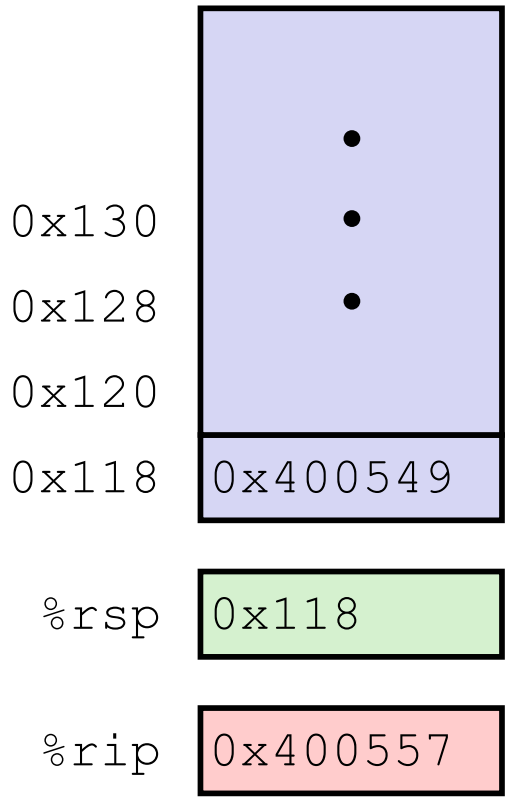
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Return Example (step 1)

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

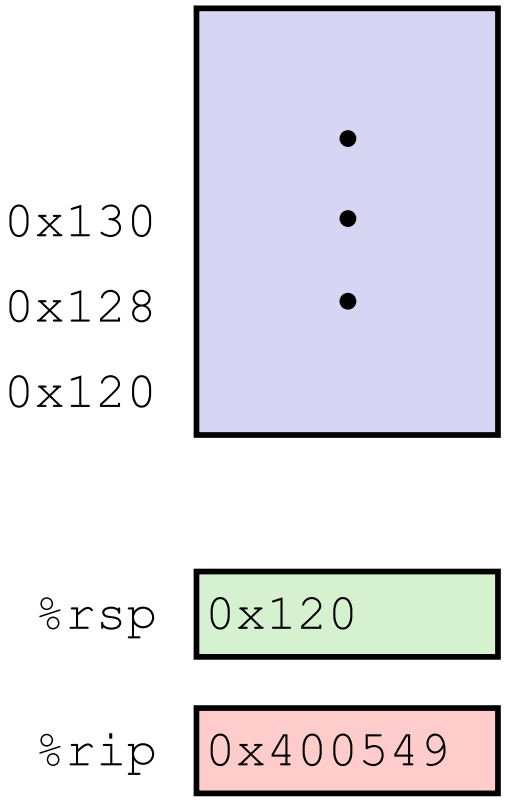
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Return Example (step 2)

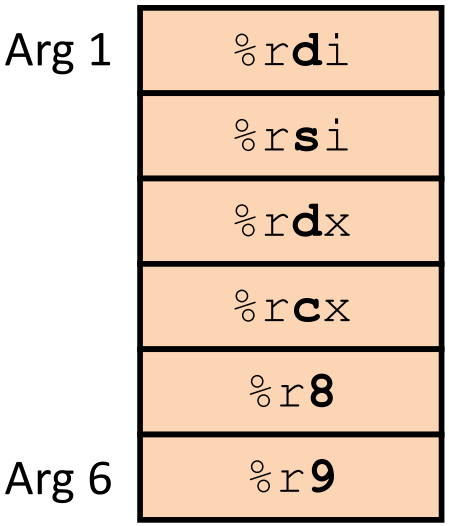
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



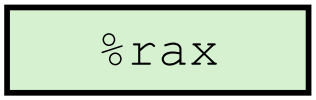
Procedure Data Flow

First 6 arguments passed in registers

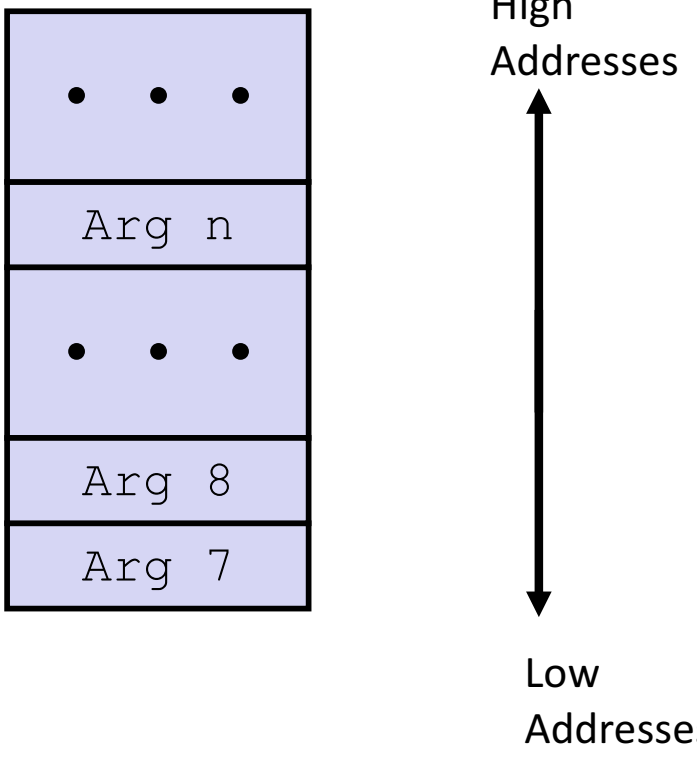


Diane's Silk Dress Costs $8 9

Return value

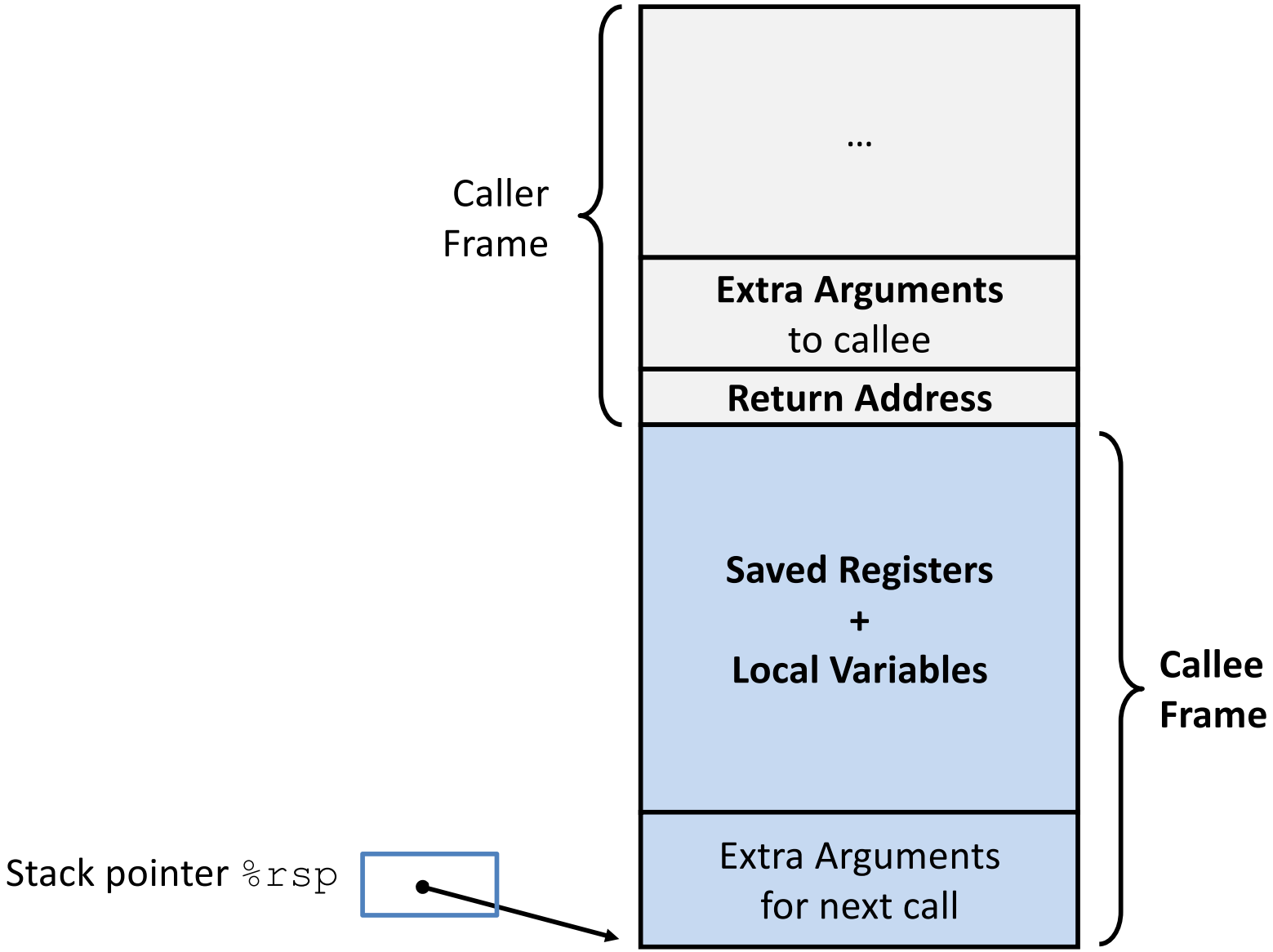


Remaining arguments passed on stack (in memory)

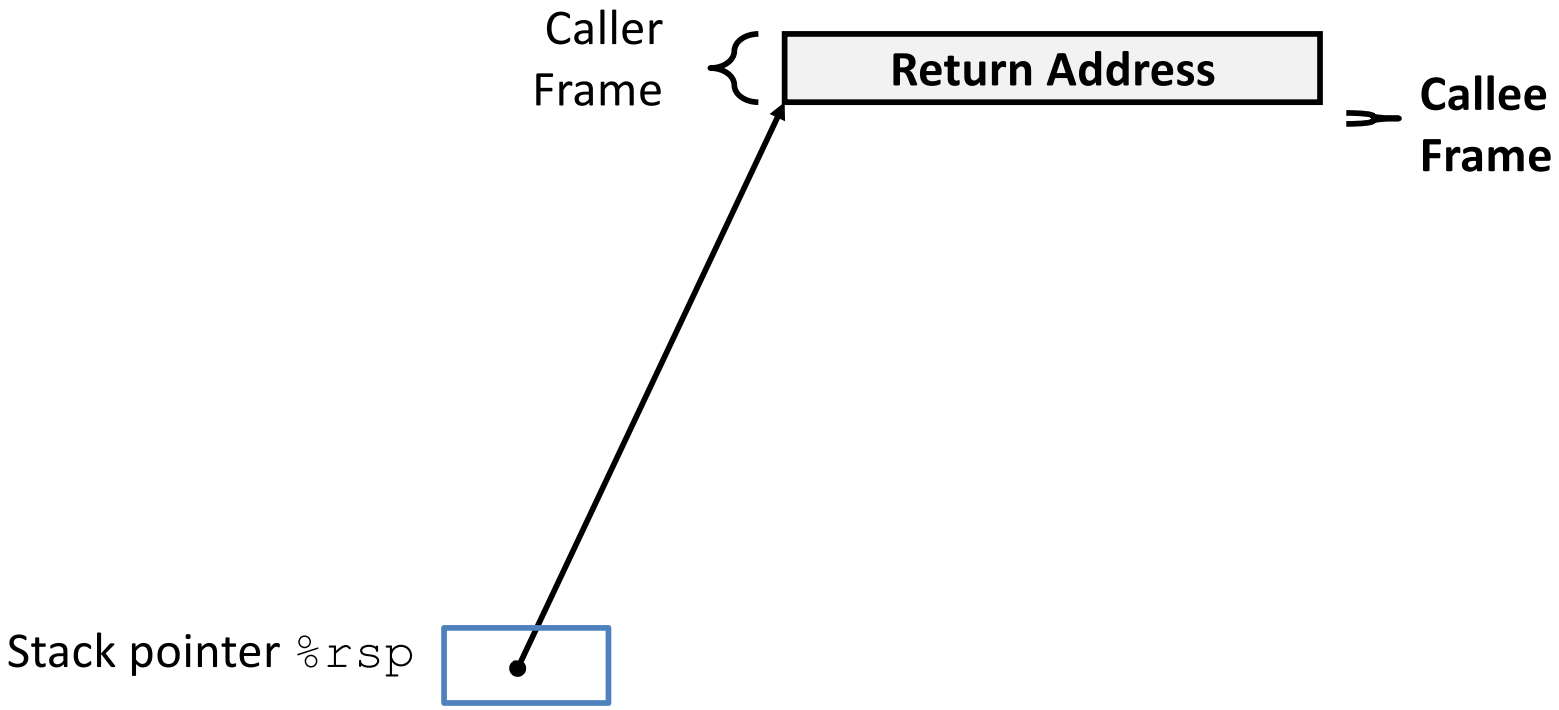


Only allocate stack space when needed

Stack Frame


...
Caller
Frame
Extra Arguments
to callee
Return Address
Saved Registers
+
Local Variables
Callee
Frame
Extra Arguments
for next call
Stack pointer %rsp

Common Stack Frame



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: callq  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:  imul   %rsi,%rax    # a * b
  # s in %rax
  400557:  retq      # Return
```

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
```

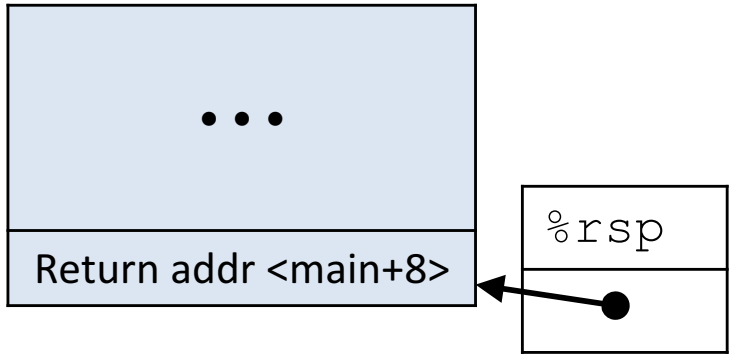
Register	Use(s)
%rdi	Argument **p**
%rsi	Argument **val, y**
%rax	**x**, Return value

Procedure Call Example (initial state)

```
long call_incr() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
call_incr:
   subq     $16, %rsp
   movq     $240, 8(%rsp)
   movl     $61, %esi
   leaq     8(%rsp), %rdi
   call     increment
   addq     8(%rsp), %rax
   addq     $16, %rsp
   ret
```

Initial Stack Structure



Procedure Call Example (step 1)

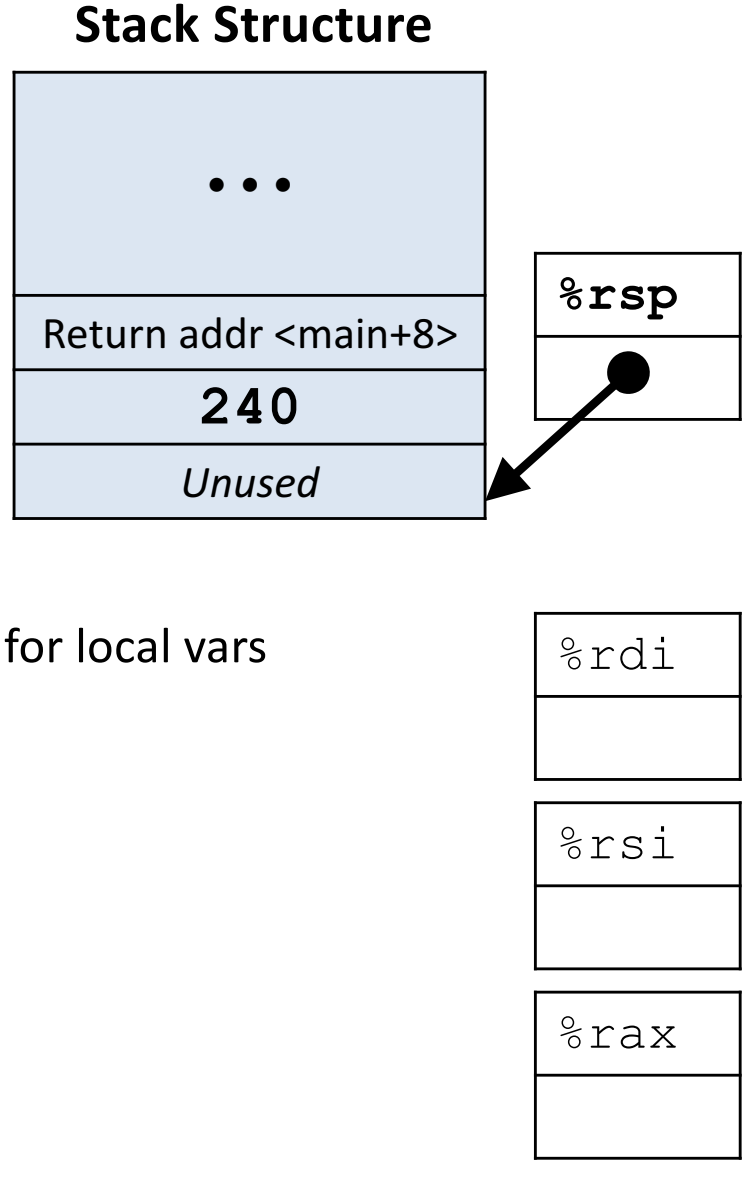
```
long call_incr() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
call_incr:
   subq     $16, %rsp
   movq     $240, 8(%rsp)
   movl     $61, %esi
   leaq     8(%rsp), %rdi
   call     increment
   addq     8(%rsp), %rax
   addq     $16, %rsp
   ret
```

Allocate space for local vars

Compiler allocated extra space for alignment →

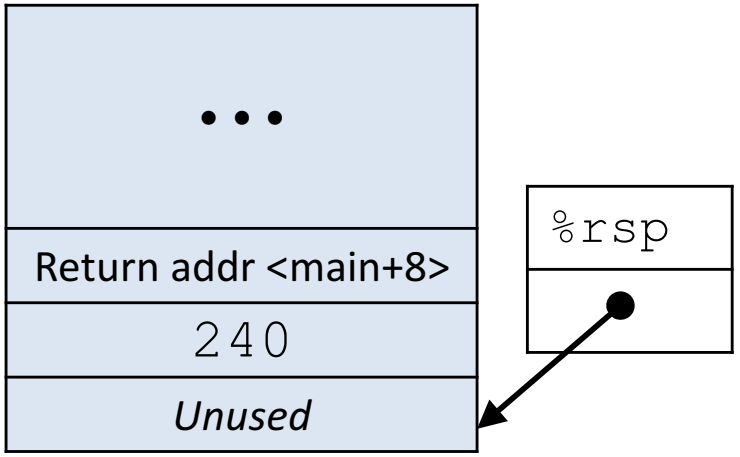
Space for v1 →



Procedure Call Example (step 2)

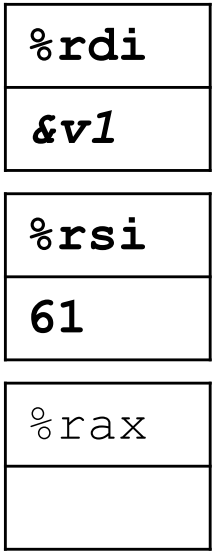
```
long call_incr() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

Stack Structure




```
call_incr:
   subq     $16, %rsp
   movq     $240, 8(%rsp)
   movl     $61, %esi
   leaq     8(%rsp), %rdi
   call     increment
   addq     8(%rsp), %rax
   addq     $16, %rsp
   ret
```

Set up args for call to `increment`



Aside: `movl` is used because 61 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`.

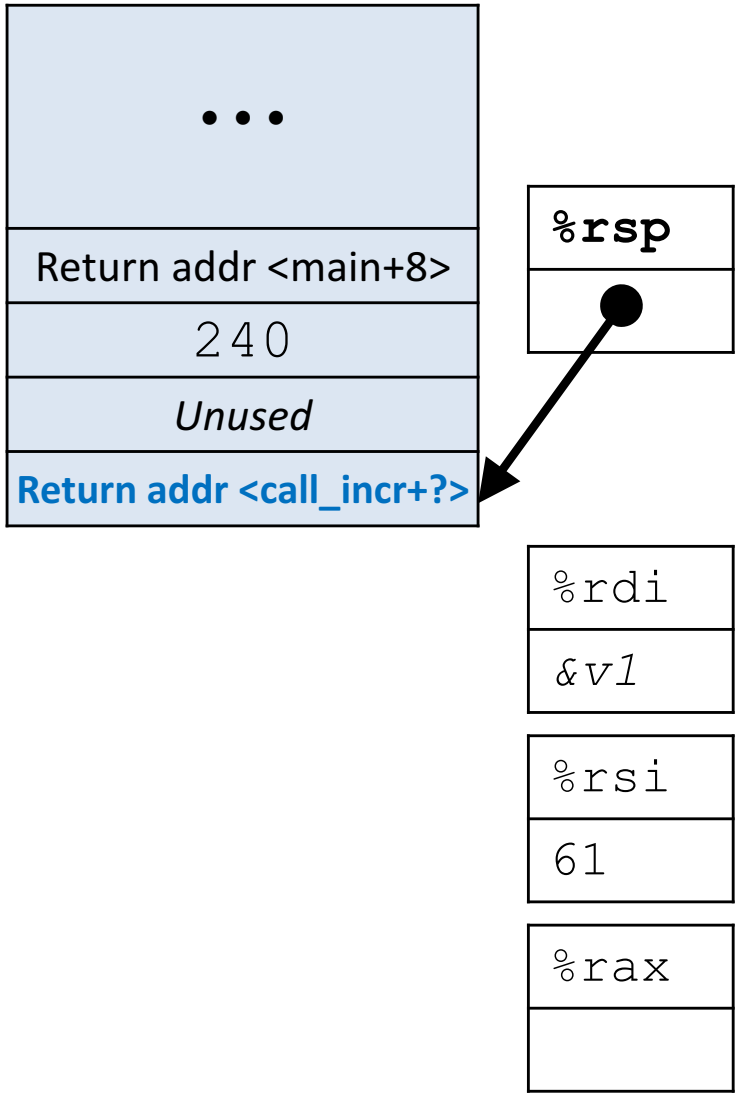
Procedure Call Example (step 3)

```
long call_incr() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $240, 8(%rsp)
  movl    $61, %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



Procedure Call Example (step 4)

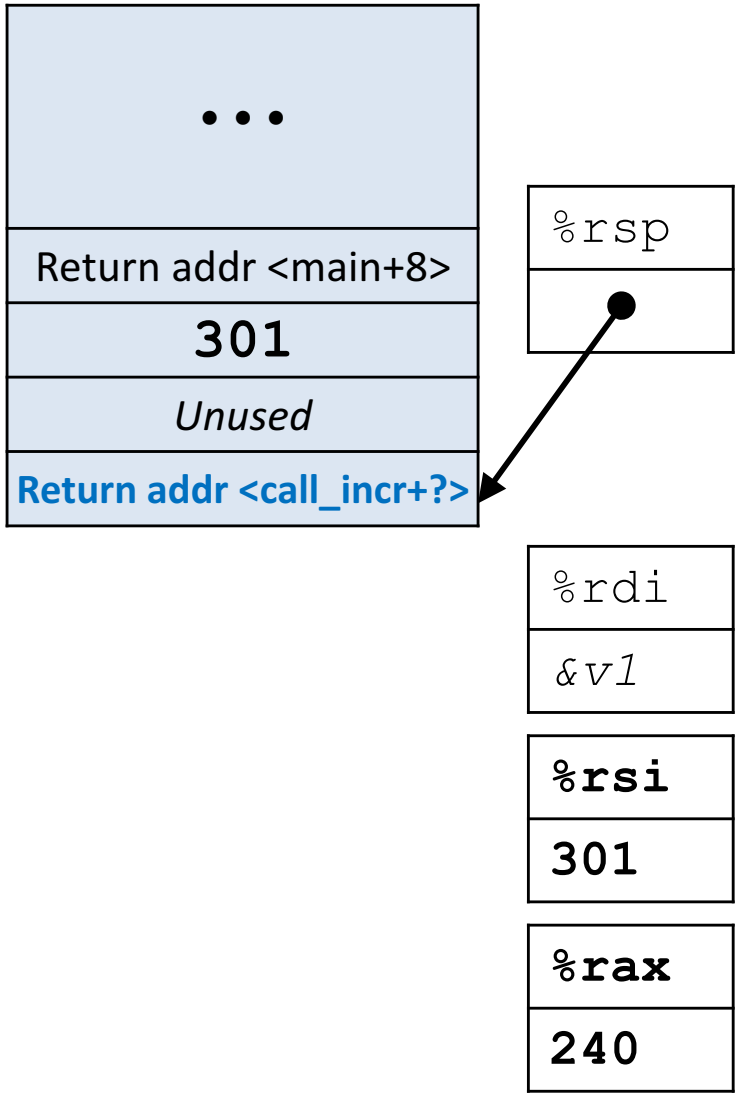
```
long call_incr() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    re
}
```

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

```
call_i
  subq
  movq
  movl    $61, %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+61
  movq    %rsi, (%rdi) # *p = y
  ret
```

Stack Structure



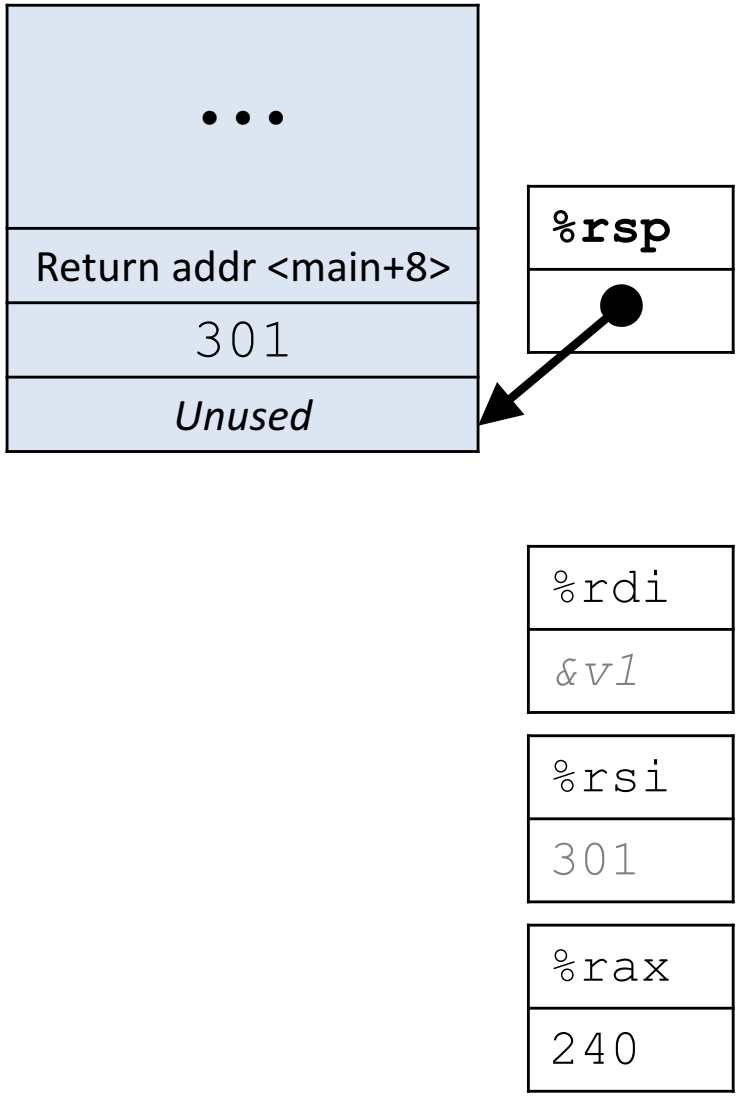
Procedure Call Example (step 5)

```
long call_incr() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $240, 8(%rsp)
  movl    $61, %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+61
  movq    %rsi, (%rdi) # *p = y
  ret
```

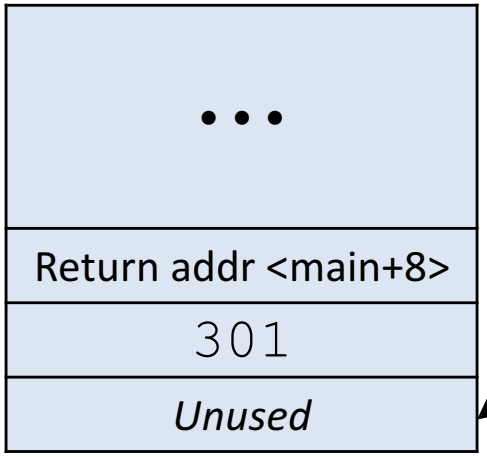
Stack Structure

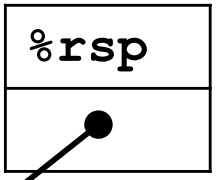


Procedure Call Example (step 6)

```
long call_incr() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

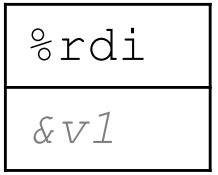
Stack Structure

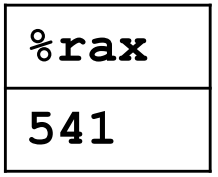





```
call_incr:
   subq     $16, %rsp
   movq     $240, 8(%rsp)
   movl     $61, %esi
   leaq     8(%rsp), %rdi
   call     increment
   addq     8(%rsp), %rax
   addq     $16, %rsp
   ret
```

Update %rax: v1+v2

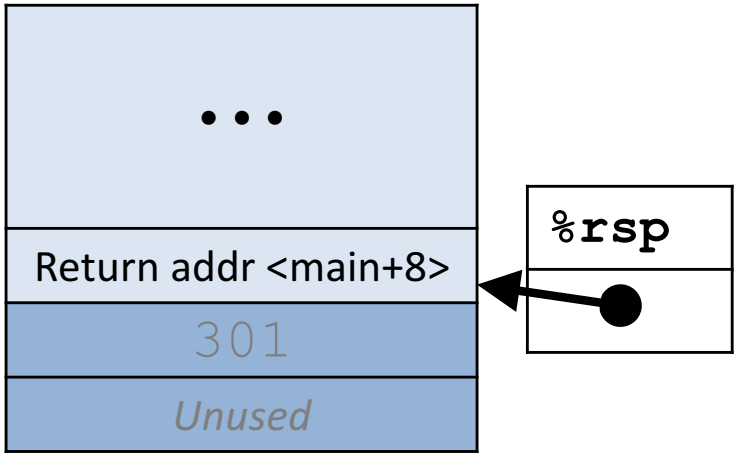




Procedure Call Example (step 7)

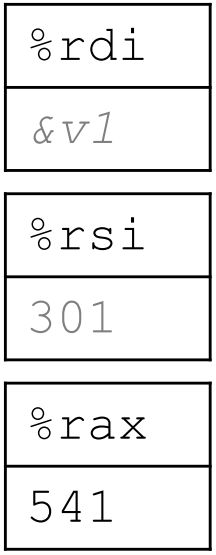
```
long call_incr() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

Stack Structure




```
call_incr:
   subq     $16, %rsp
   movq     $240, 8(%rsp)
   movl     $61, %esi
   leaq     8(%rsp), %rdi
   call     increment
   addq     8(%rsp), %rax
   addq     $16, %rsp
   ret
```

De-allocate space for local vars

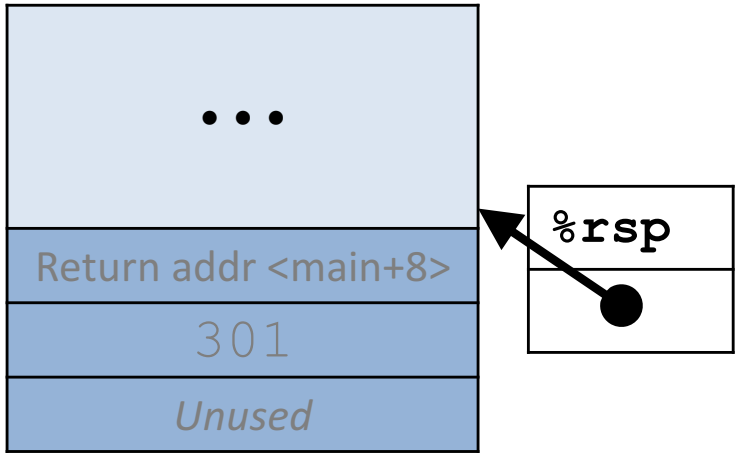


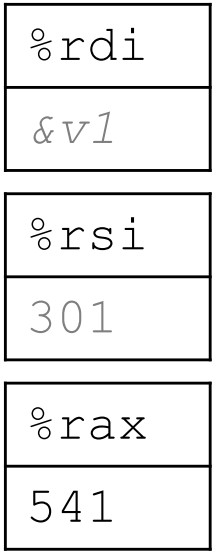
Procedure Call Example (return from call_incr)

```
long call_incr() {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return v1+v2;
}
```

```
main:
   ...
   call     call_incr
   ...
```

Stack Structure





Register Saving Conventions

yoo calls who:

Caller ***Callee***

Will register contents still be there after a procedure call?

```
yoo:
    • • •
    movq $12345, %rbx
    call who
    addq %rbx, %rax
    • • •
    ret
```

?

```
who:
    • • •
    addq %rdi, %rbx
    • • •
    ret
```

Conventions:

Caller Save

Caller saves temporary values in its frame **before calling**

Callee Save

Callee saves temporary values in its frame **before using**

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

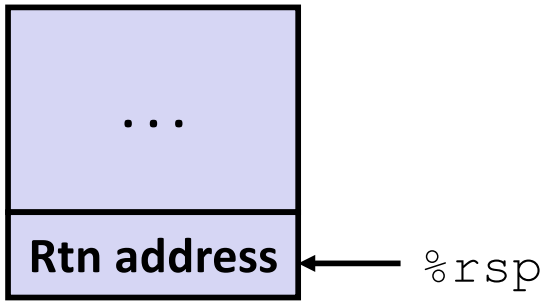
%r8	Argument #5 – Caller saved
%r9	Argument #6 – Caller saved
%r10	Caller saved
%r11	Caller Saved
%r12	**Callee** saved
%r13	**Callee** saved
%r14	**Callee** saved
%r15	**Callee** saved

Callee-Saved Example (step 1)

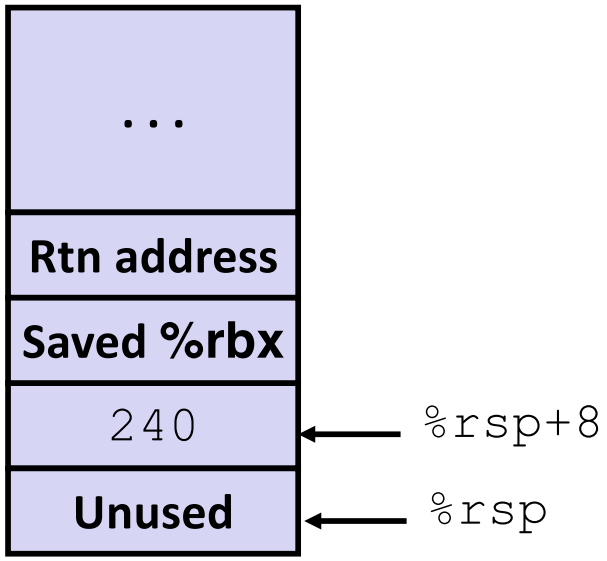
```
long call_incr2(long x) {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $240, 8(%rsp)
  movl    $61, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

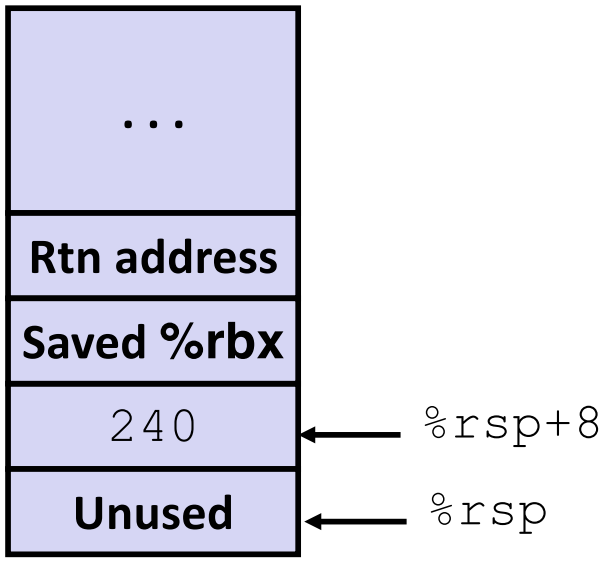


Callee-Saved Example (step 2)

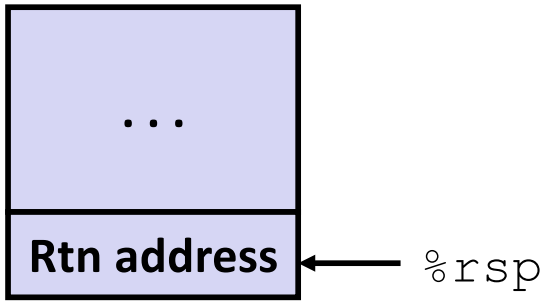
```
long call_incr2(long x) {
    long v1 = 240;
    long v2 = increment(&v1, 61);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $240, 8(%rsp)
  movl    $61, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Stack Structure



Pre-return Stack Structure



A Puzzle

C function body:

```
*p = d;
return x - c;
```

assembly:

```
movsbl %dl,%edx
movl   %edx,(%rsi)
movswl %di,%edi
subl   %edi,%ecx
movl   %ecx,%eax
```

Write the C function header, types, and order of parameters.

movsbl = move sign-extending a byte to a long (4-byte)
movswl = move sign-extending a word (2-byte) to a long (4-byte)

Recursive Function

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

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  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);
  }
}
```

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

Trick because some HW doesn't like jumping to ret

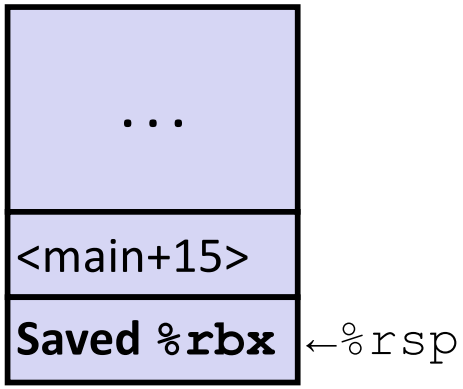
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

Recursive Function: Register Save

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

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

Register	Use(s)	Type
%rdi	**x**	Argument

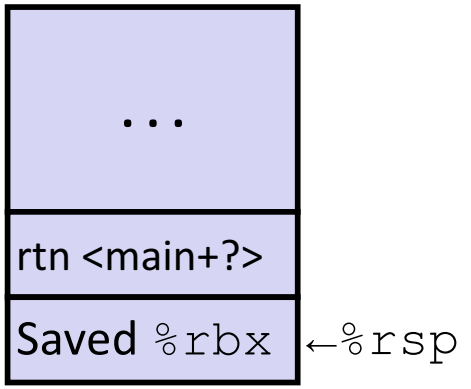


Recursive Function: Call Setup

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

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

Register	Use(s)	Type
%rdi	x >> 1	Recursive arg
%rbx	x & 1	Callee-saved

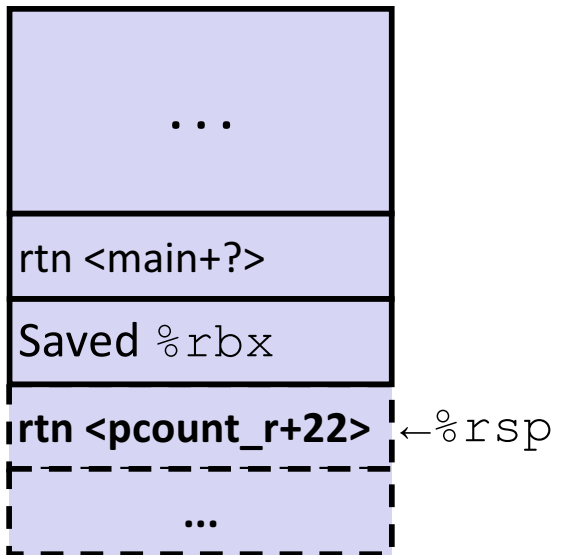


Recursive Function: Call

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

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

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	–

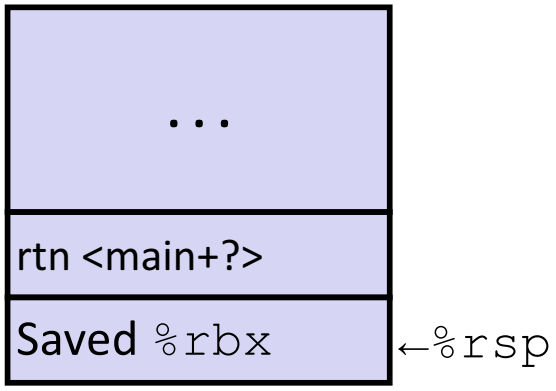


Recursive Function: Result

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

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

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

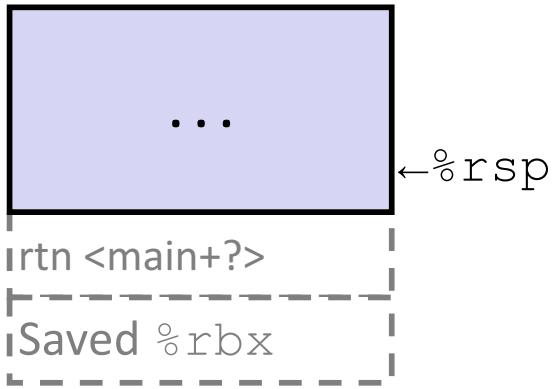


Recursive Function: Completion

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

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

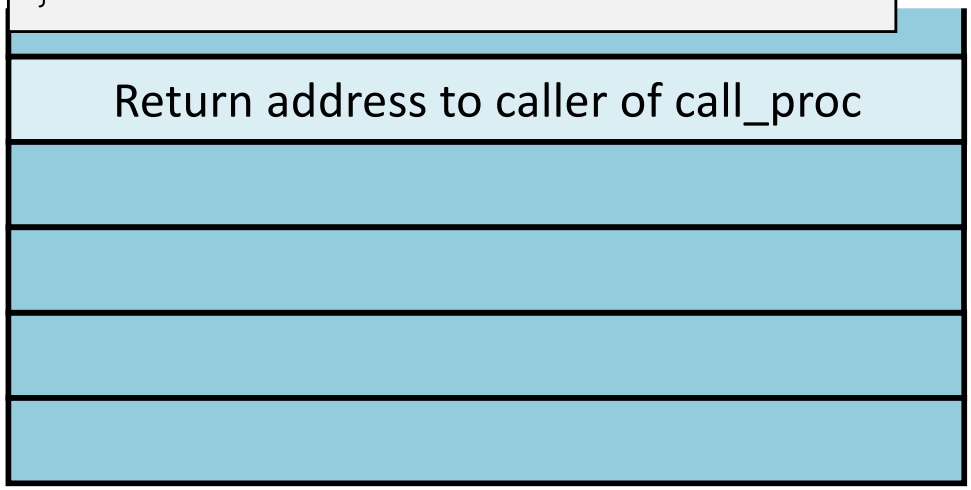
Register	Use(s)	Type
%rax	Return value	



x86-64 stack storage 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
  • • •
```



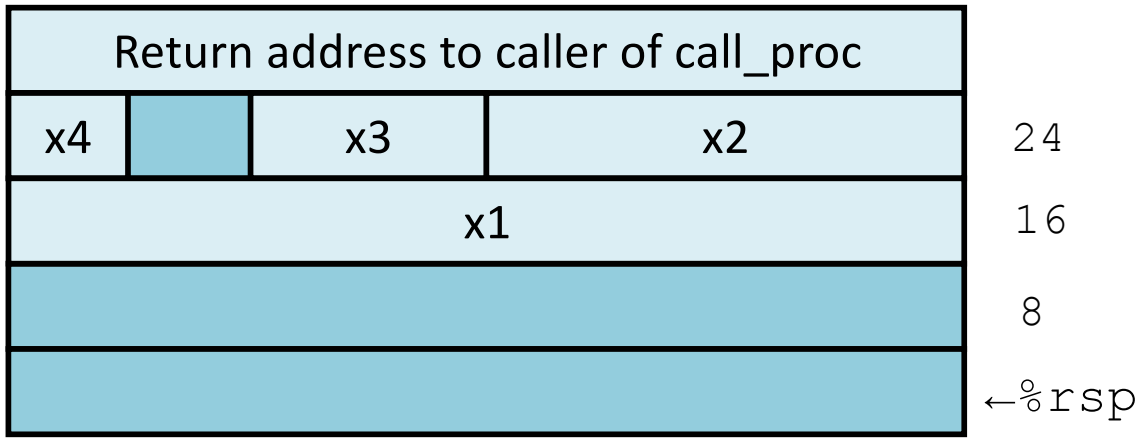
←%rsp

x86-64 stack storage 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);
}
```

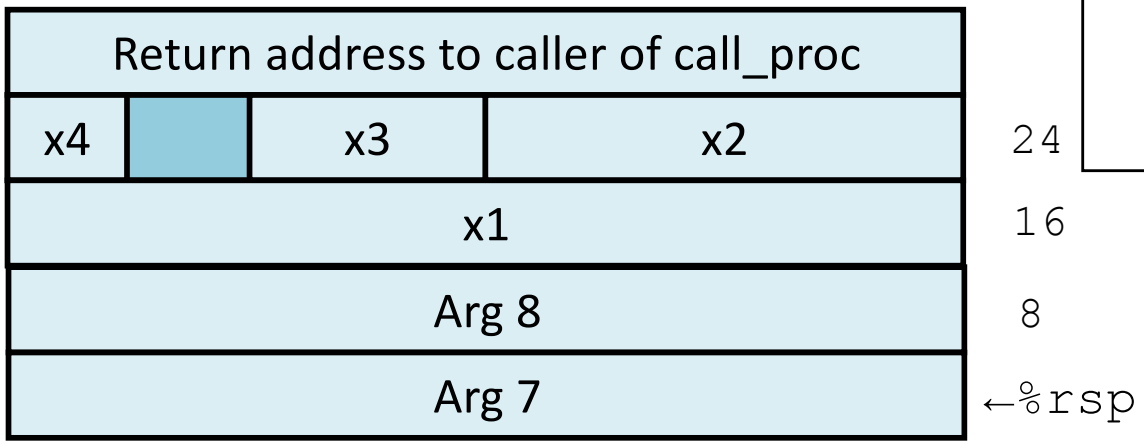
```
call_proc:
  subq  $32,%rsp
  movq  $1,16(%rsp)  # x1
  movl  $2,24(%rsp)  # x2
  movw  $3,28(%rsp)  # x3
  movb  $4,31(%rsp)  # x4
  • • •
```



x86-64 stack storage example

(3) setup args 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 # &x2
   leaq  16(%rsp),%rsi # &x1
   leaq  31(%rsp),%rax # &x4
   movq  %rax,8(%rsp)  # ...
   movl  $4,(%rsp)     # 4
   leaq  28(%rsp),%r9  # &x3
   movl  $3,%r8d       # 3
   movl  $2,%edx       # 2
   movq  $1,%rdi       # 1
   call  proc
   . . .
```

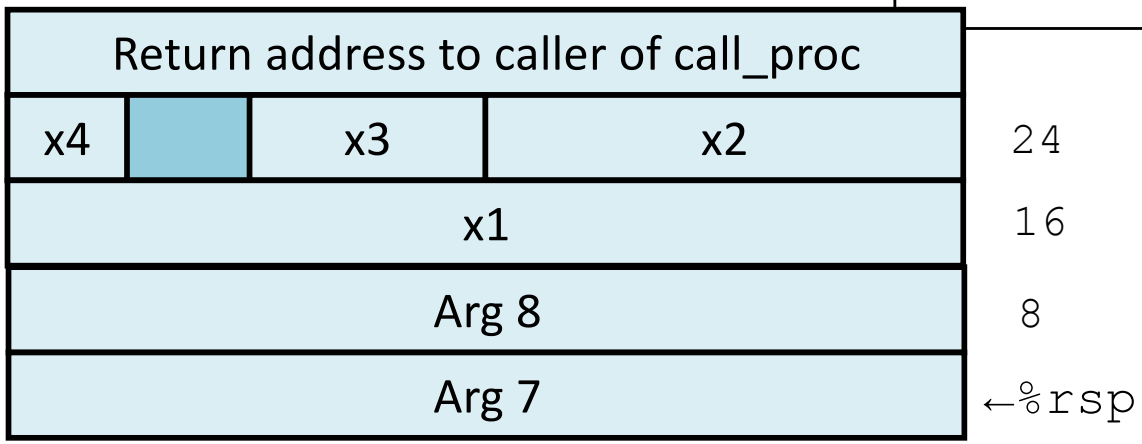
Arguments passed in (in order): rdi, rsi, rdx, rcx, r8, r9

x86-64 stack storage example

(4) 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 # x3
  movsbl 31(%rsp),%edx # x4
  subl   %edx,%eax  # x3-x4
  cltq   # sign-extend %eax->rax
  movslq 24(%rsp),%rdx # x2
  addq   16(%rsp),%rdx # x1+x2
  imulq  %rdx,%rax     # *
  addq   $32,%rsp
  ret
```

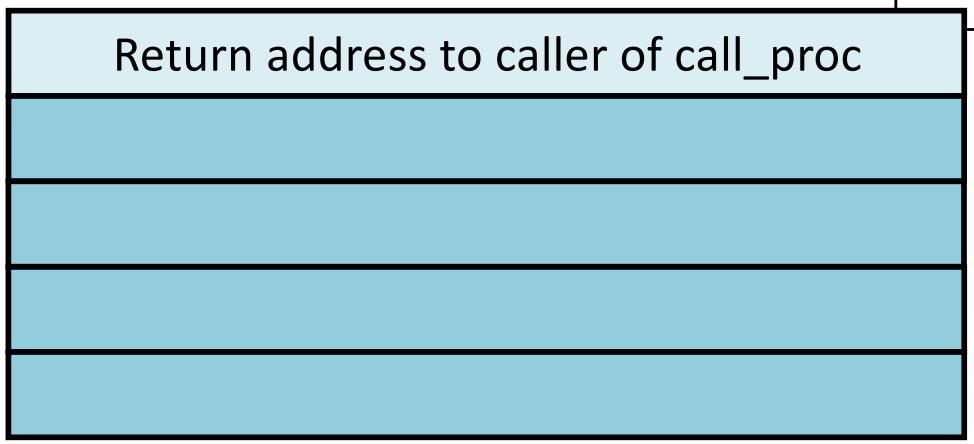


x86-64 stack storage example

(5) deallocate 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
```



←%rsp

Procedure Summary

call, ret, push, pop

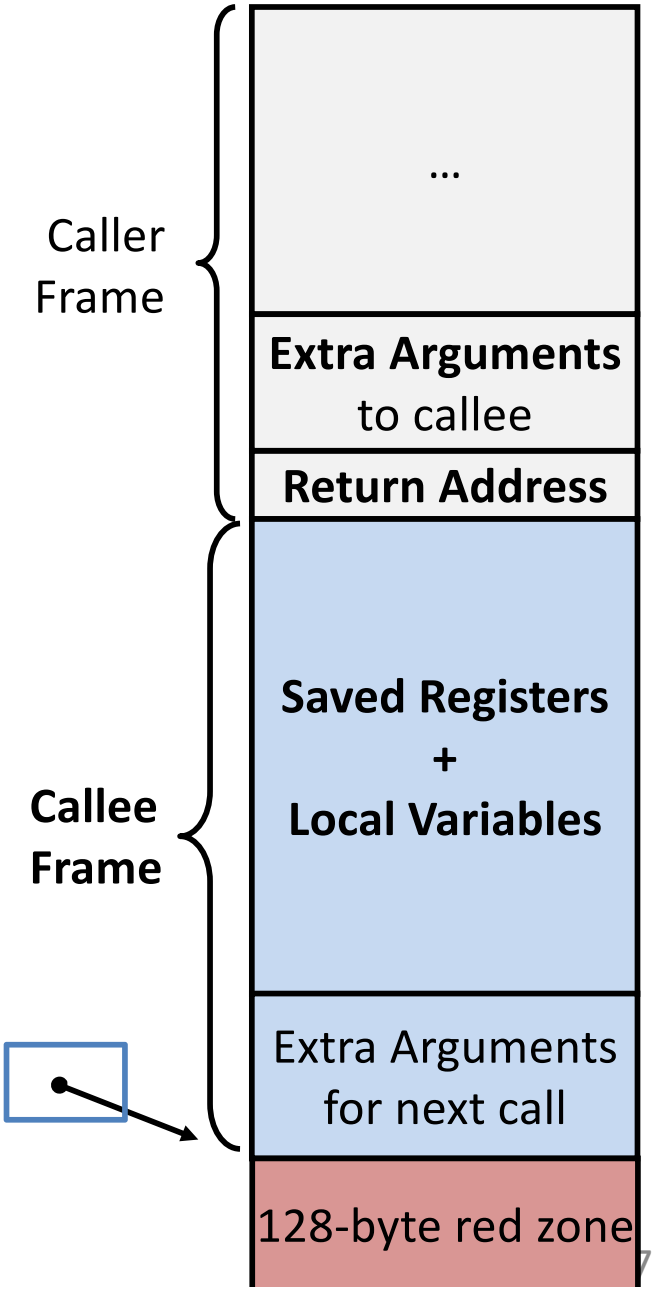
Stack discipline fits procedure call / return.*

If P calls Q: Q (and calls by Q) returns before P

Conventions support arbitrary function calls.

Register-save conventions.
Stack frame saves extra args or local variables.
Result returned in **`%rax`**

`%rax` Return value – Caller saved	`%r8` Argument #5 – Caller saved
`%rbx` Callee saved	`%r9` Argument #6 – Caller saved
`%rcx` Argument #4 – Caller saved	`%r10` Caller saved
`%rdx` Argument #3 – Caller saved	`%r11` Caller Saved
`%rsi` Argument #2 – Caller saved	`%r12` Callee saved
`%rdi` Argument #1 – Caller saved	`%r13` Callee saved
`%rsp` Stack pointer	`%r14` Callee saved
`%rbp` Callee saved	`%r15` Callee saved



*Take 251 to learn about languages where it doesn't.