

Computer Systems

CSE 410 Autumn 2013

7 – Procedures, parameters, and the call stack

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

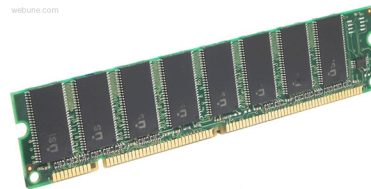
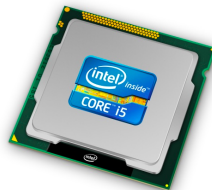
Assembly
language:

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

Machine
code:

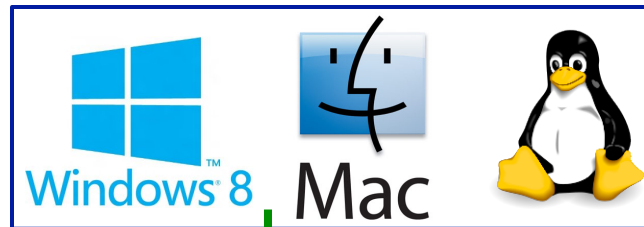
```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer
system:



Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

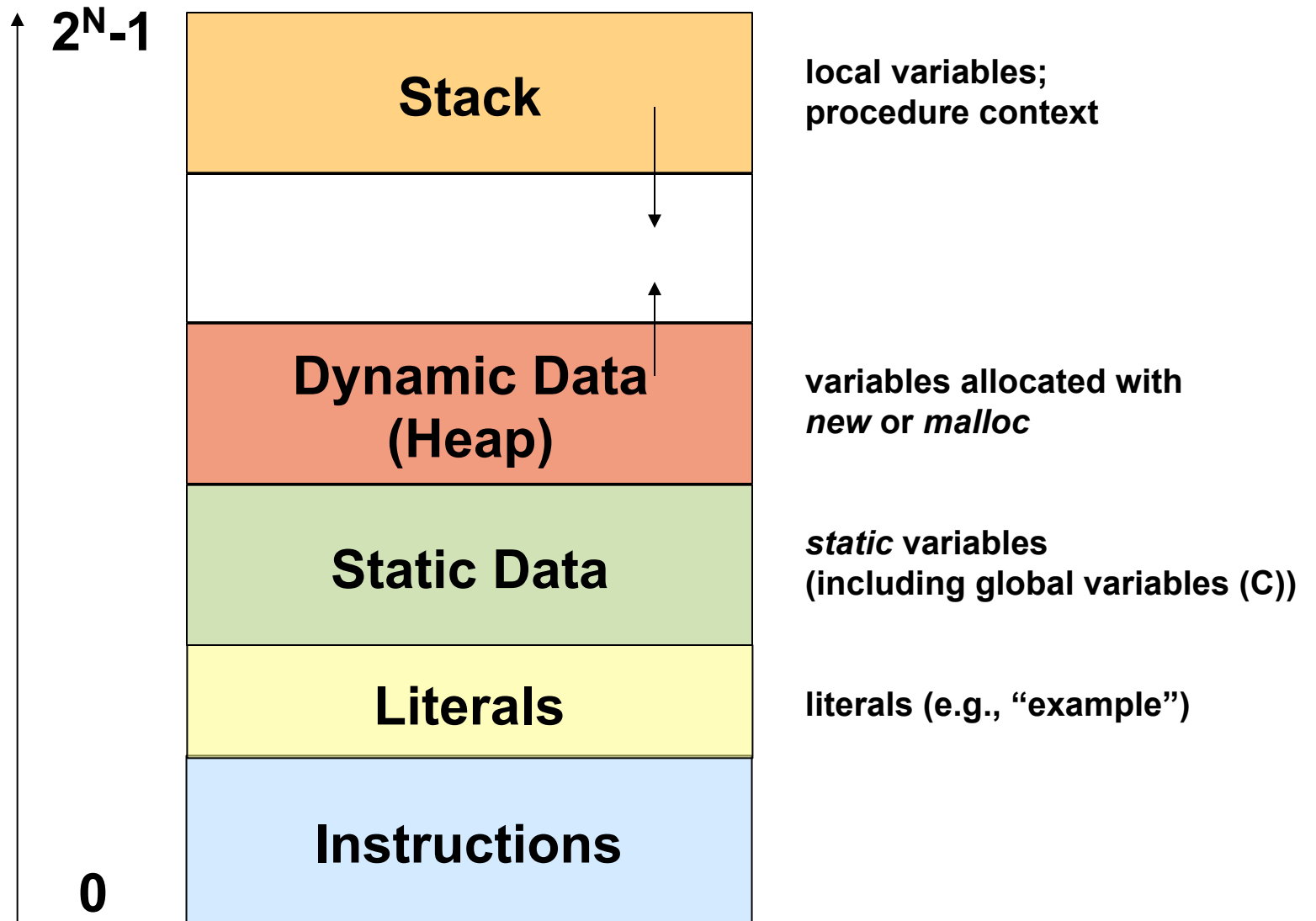
OS:



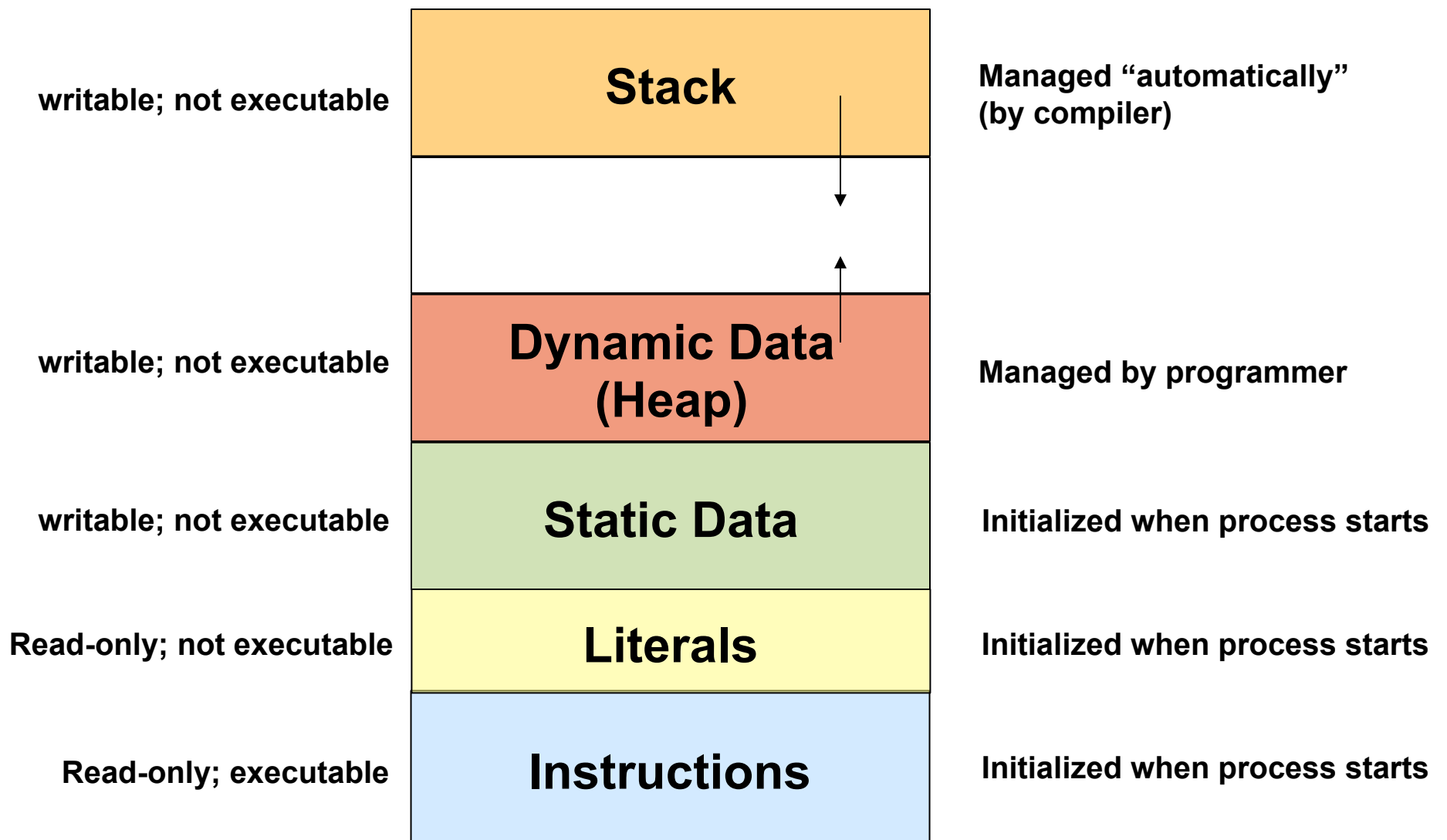
Procedures & Stacks

- Stacks in memory and stack operations
- The stack used to keep track of procedure calls
- Return addresses and return values
- Stack-based languages
- The Linux stack frame
- Passing arguments on the stack
- Allocating local variables on the stack
- Register-saving conventions
- Procedures and stacks on x64 architecture

Memory Layout

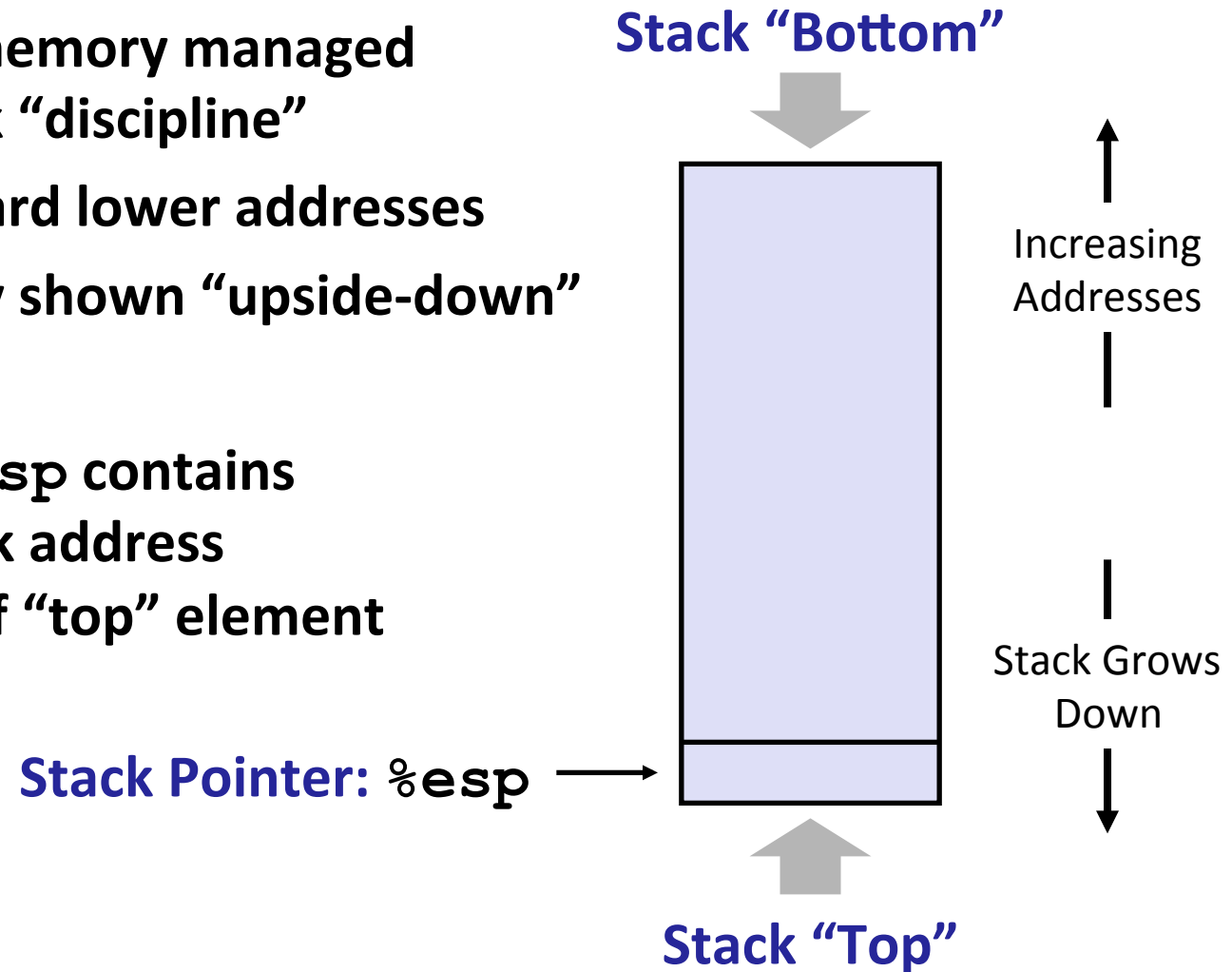


Memory Layout



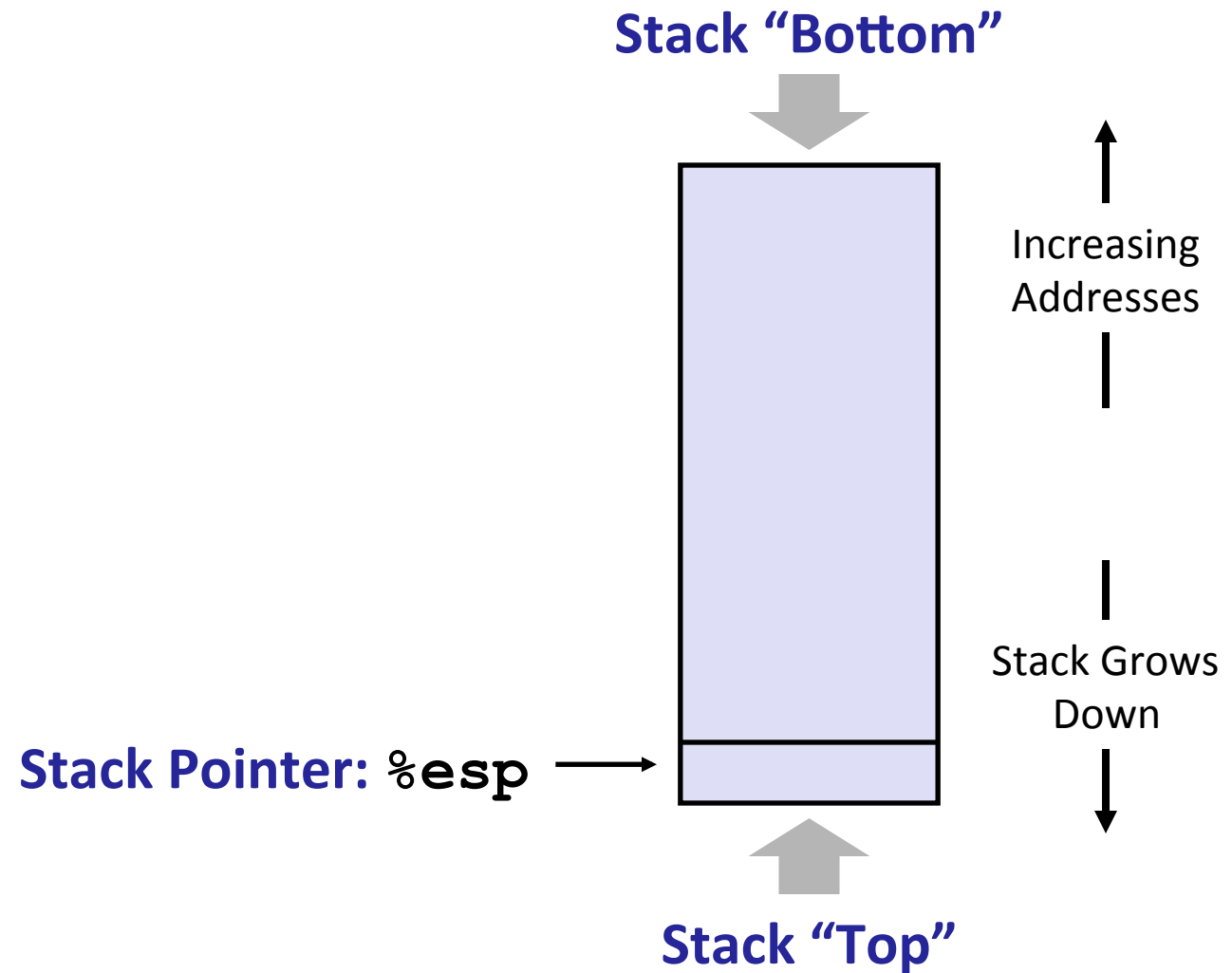
IA32 Call Stack

- Region of memory managed with a stack “discipline”
- Grows toward lower addresses
- Customarily shown “upside-down”
- Register `%esp` contains lowest stack address = address of “top” element



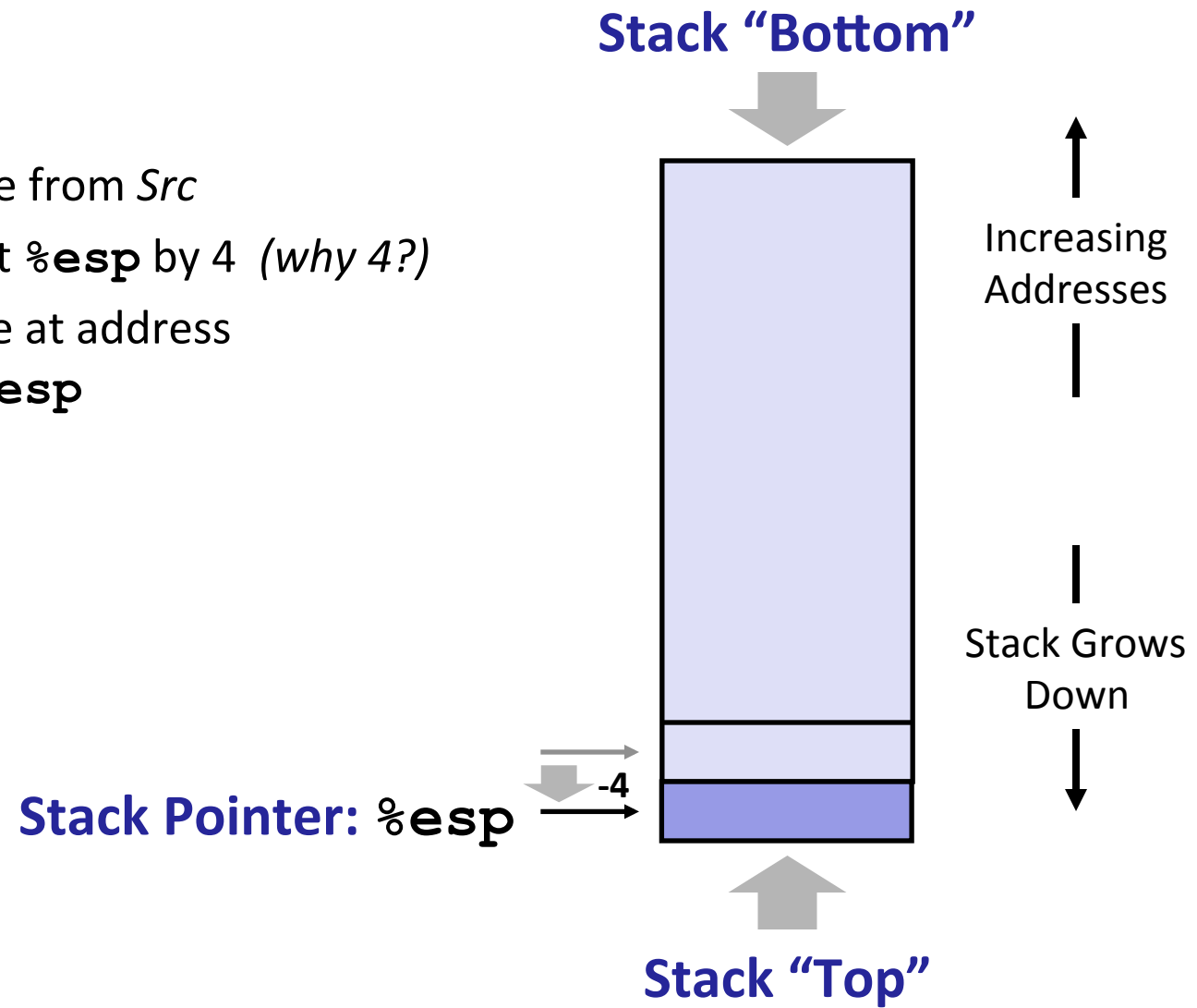
IA32 Call Stack: Push

- `pushl Src`



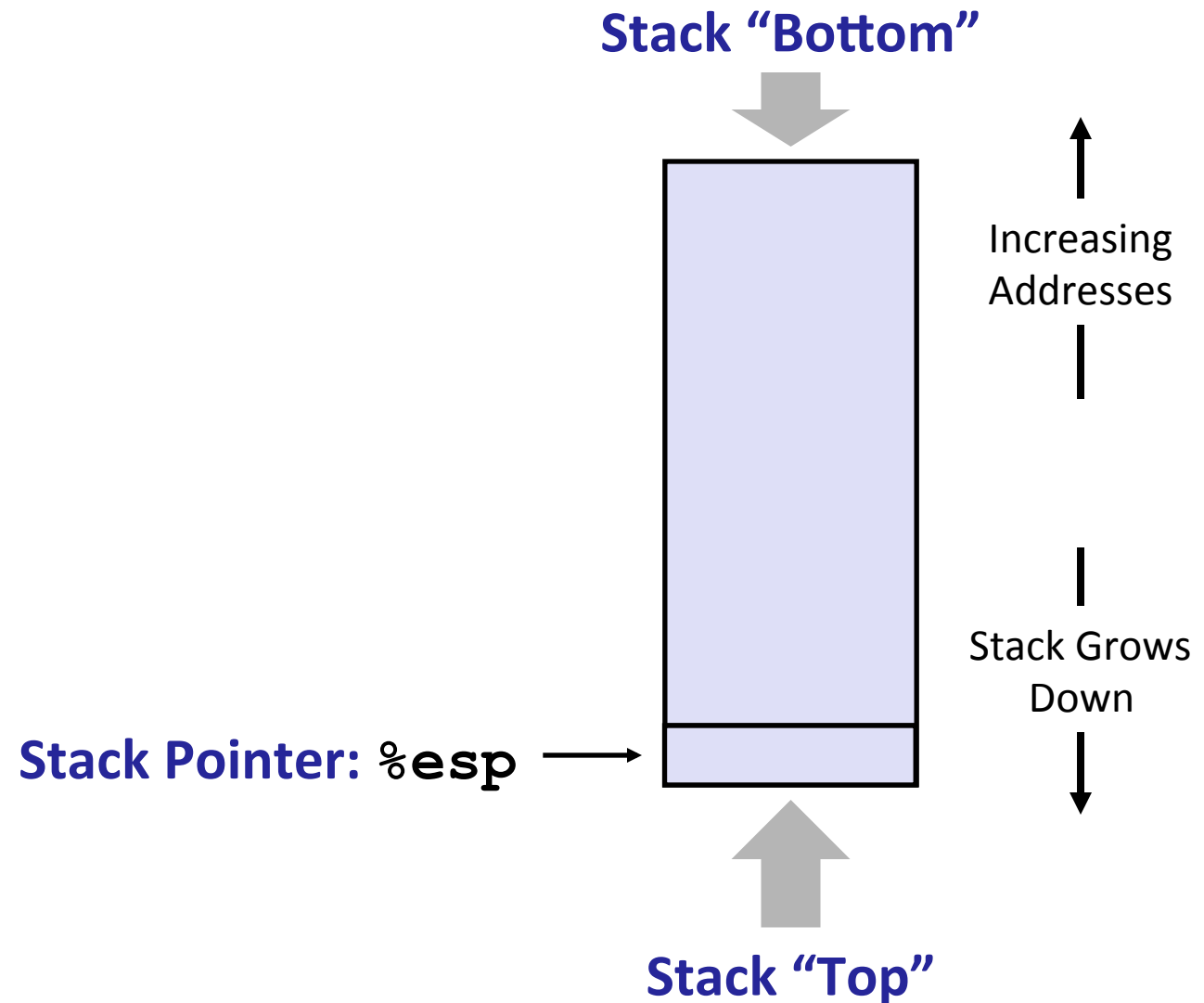
IA32 Call Stack: Push

- **pushl Src**
 - Fetch value from *Src*
 - Decrement `%esp` by 4 (*why 4?*)
 - Store value at address given by `%esp`



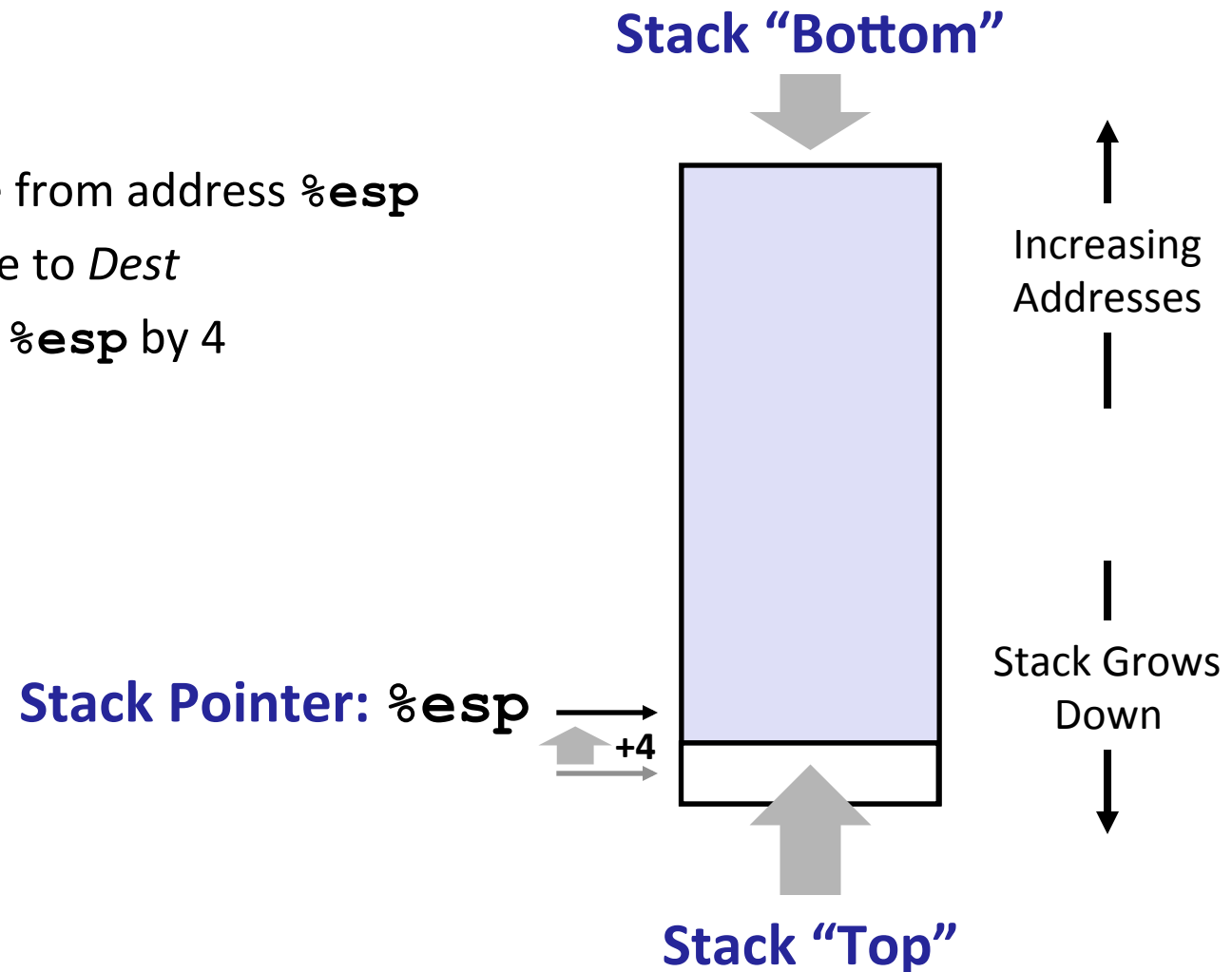
IA32 Call Stack: Pop

- `popl Dest`



IA32 Call Stack: Pop

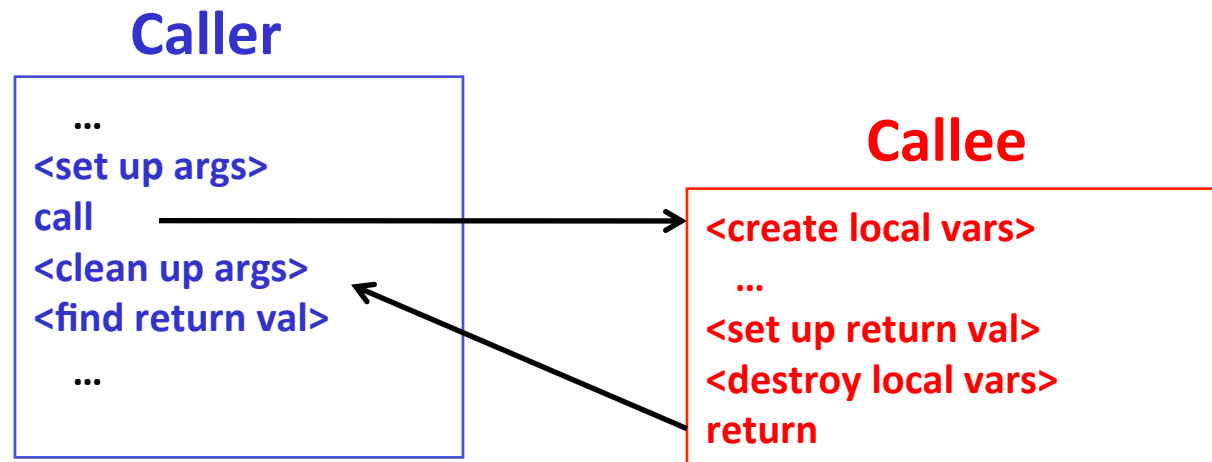
- `popl Dest`
 - Load value from address `%esp`
 - Write value to `Dest`
 - Increment `%esp` by 4



Procedures & Stacks

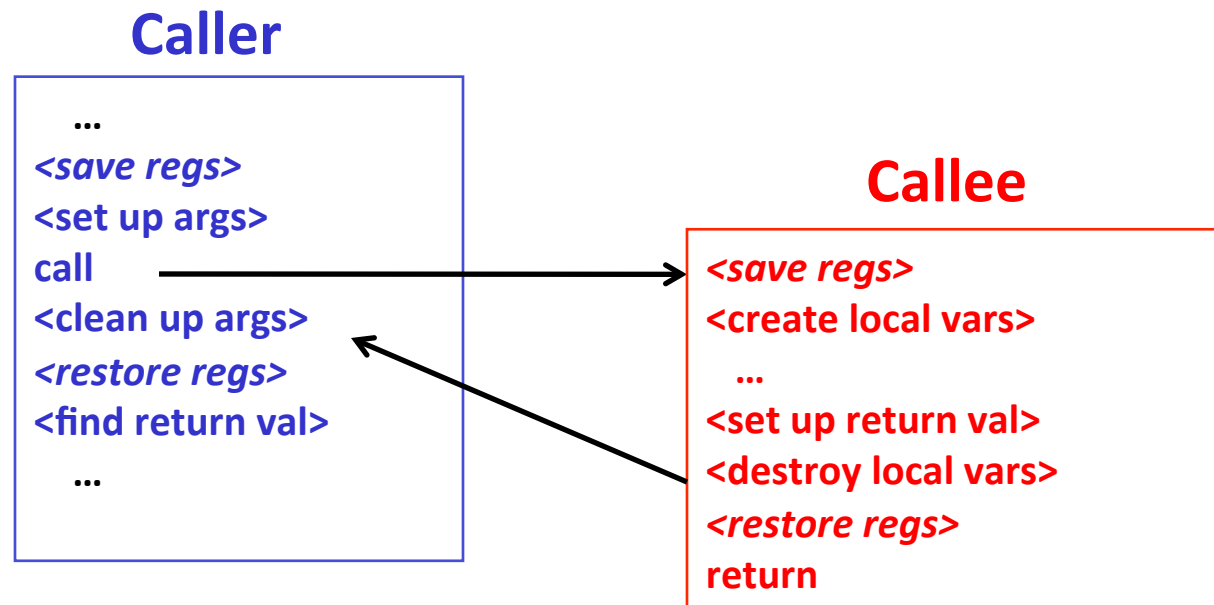
- **Stacks in memory and stack operations**
- **The stack used to keep track of procedure calls**
- **Return addresses and return values**
- **Stack-based languages**
- **The Linux stack frame**
- **Passing arguments on the stack**
- **Allocating local variables on the stack**
- **Register-saving conventions**
- **Procedures and stacks on x64 architecture**

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 val
- **Caller** and **Callee** run on same CPU → use the same registers
 - **Caller** might need to save registers that **Callee** might use
 - **Callee** might need to save registers that **Caller** has used

Procedure Call Overview



- The convention of where to leave/find things is called the procedure call linkage
 - Details vary between systems
 - We will see the convention for IA32/Linux in detail
 - What could happen if our program didn't follow these conventions?

Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** `call label`
 - Push return address on stack
 - Jump to *label*

Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** `call label`
 - Push return address on stack
 - Jump to *label*
- **Return address:**
 - Address of instruction after `call`
 - Example from disassembly:

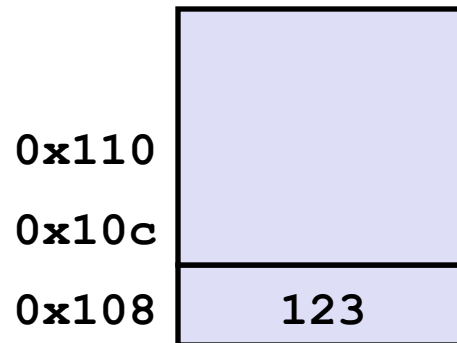
804854e:	e8 3d 06 00 00	call	8048b90 <main>
8048553:	50	pushl	%eax

- Return address = `0x8048553`
- **Procedure return:** `ret`
 - Pop return address from stack
 - Jump to address

Procedure Call Example

804854e:	e8 3d 06 00 00	call	8048b90 <main>
8048553:	50	pushl	%eax

call 8048b90



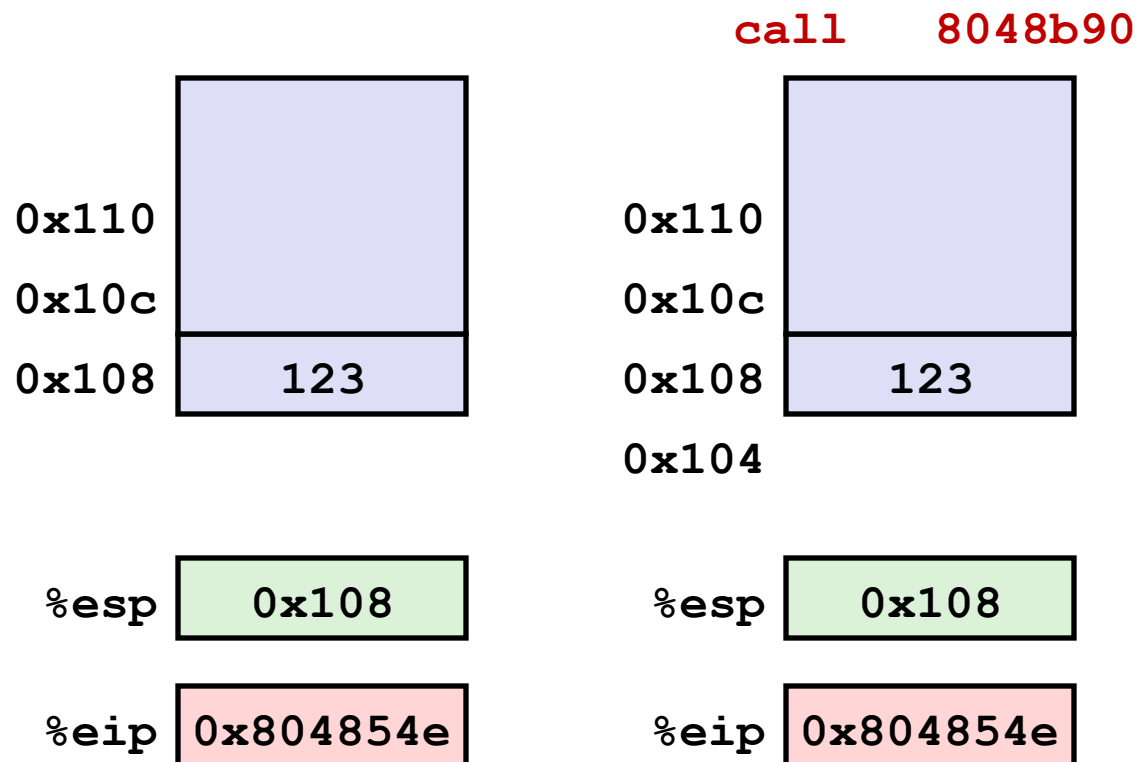
%esp 0x108

%eip 0x804854e

%eip: program counter

Procedure Call Example

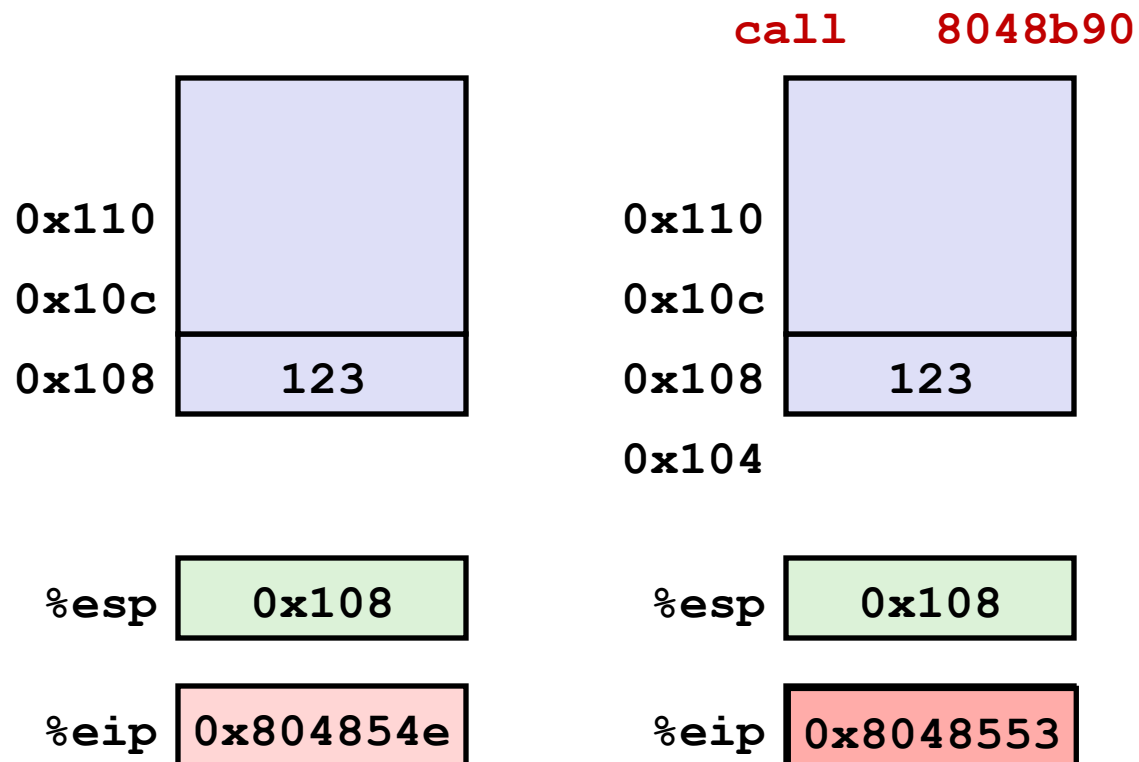
804854e:	e8 3d 06 00 00	call	8048b90 <main>
8048553:	50	pushl	%eax



%eip: program counter

Procedure Call Example

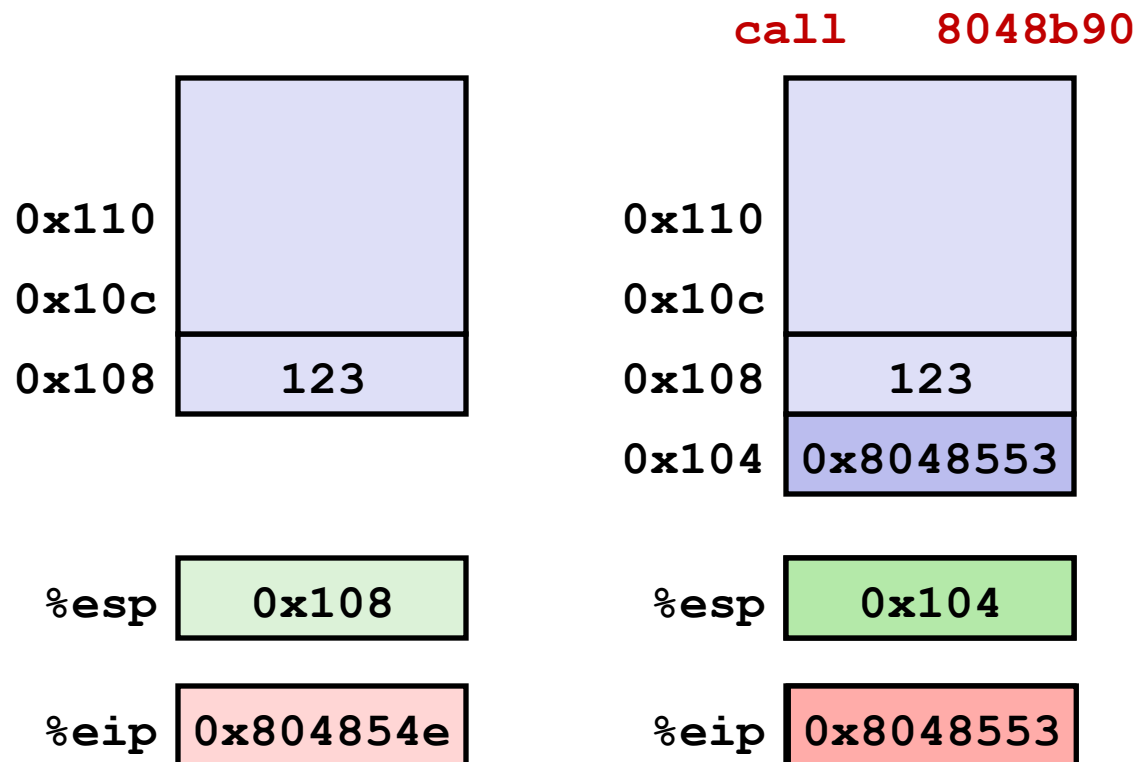
804854e:	e8 3d 06 00 00	call	8048b90 <main>
8048553:	50	pushl	%eax



%eip: program counter

Procedure Call Example

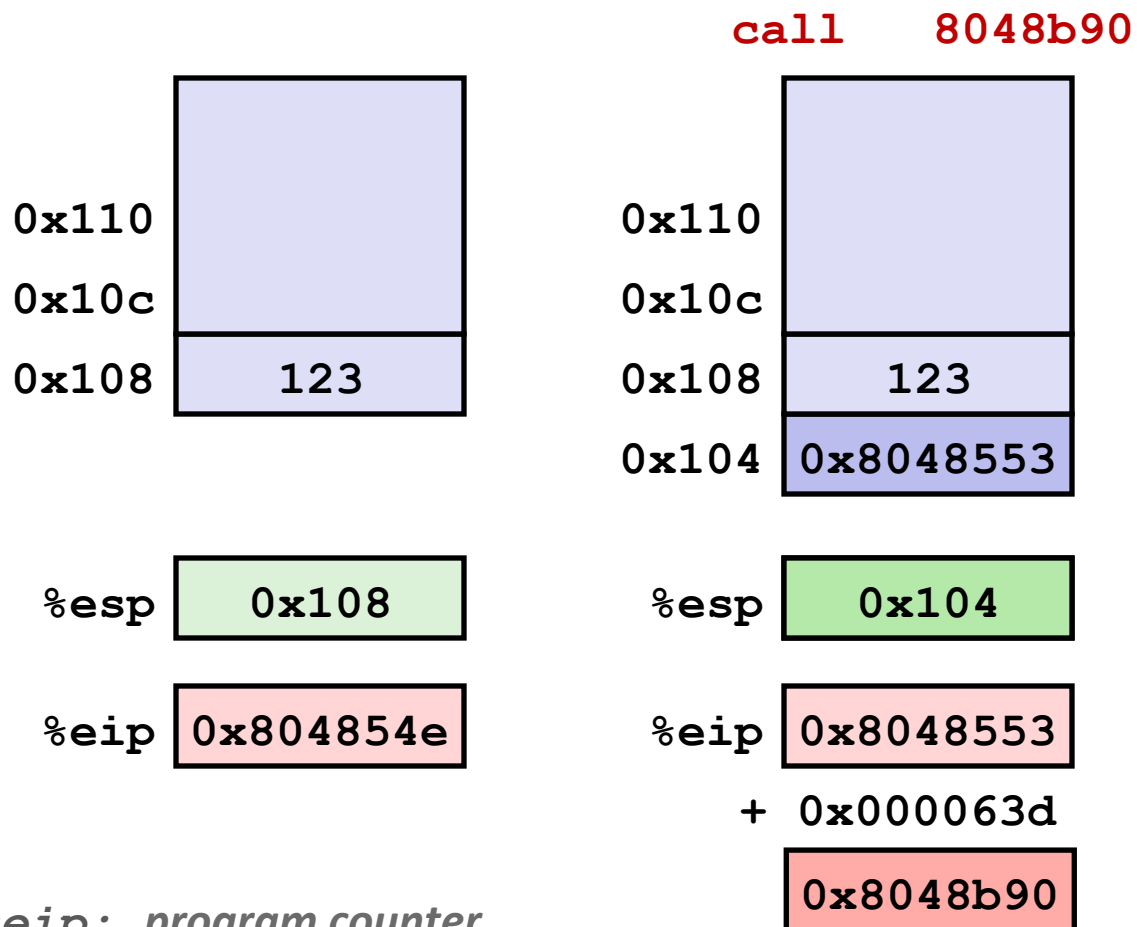
804854e:	e8 3d 06 00 00	call	8048b90 <main>
8048553:	50	pushl	%eax



%eip: program counter

Procedure Call Example

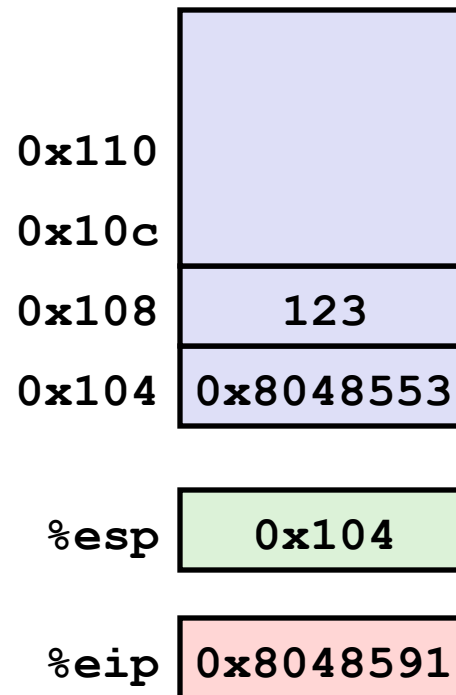
804854e:	e8 3d 06 00 00	call	8048b90 <main>
8048553:	50	pushl	%eax



Procedure Return Example

8048591:	c3	ret
----------	----	-----

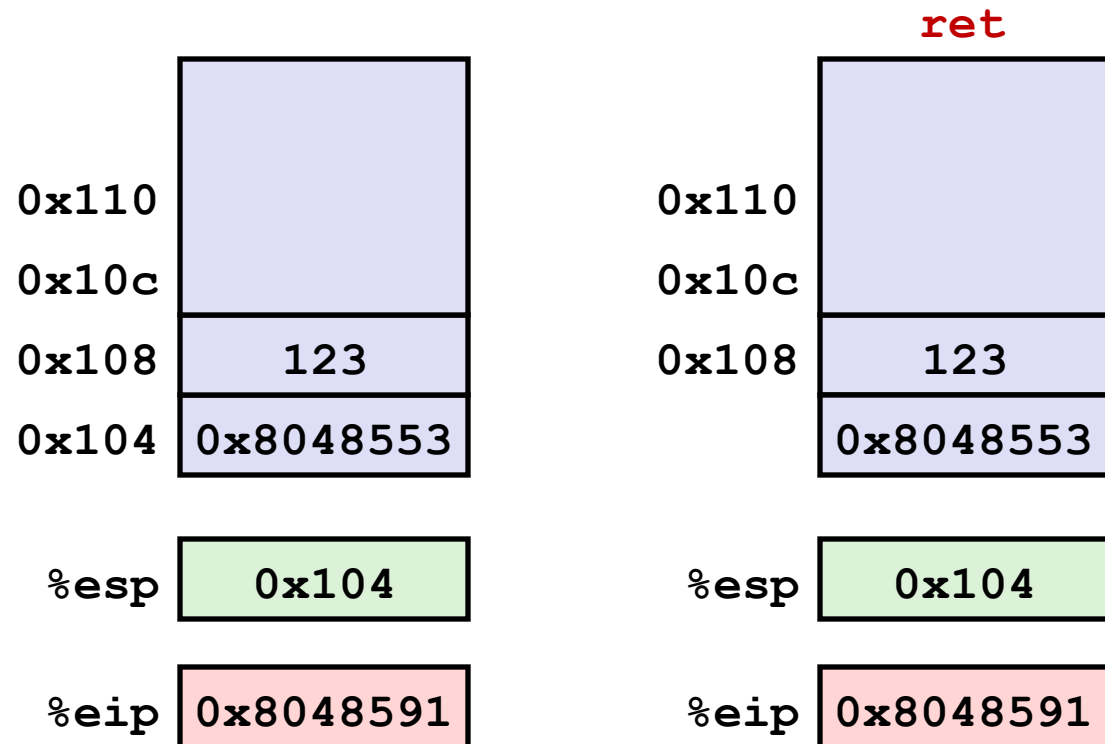
ret



%eip: program counter

Procedure Return Example

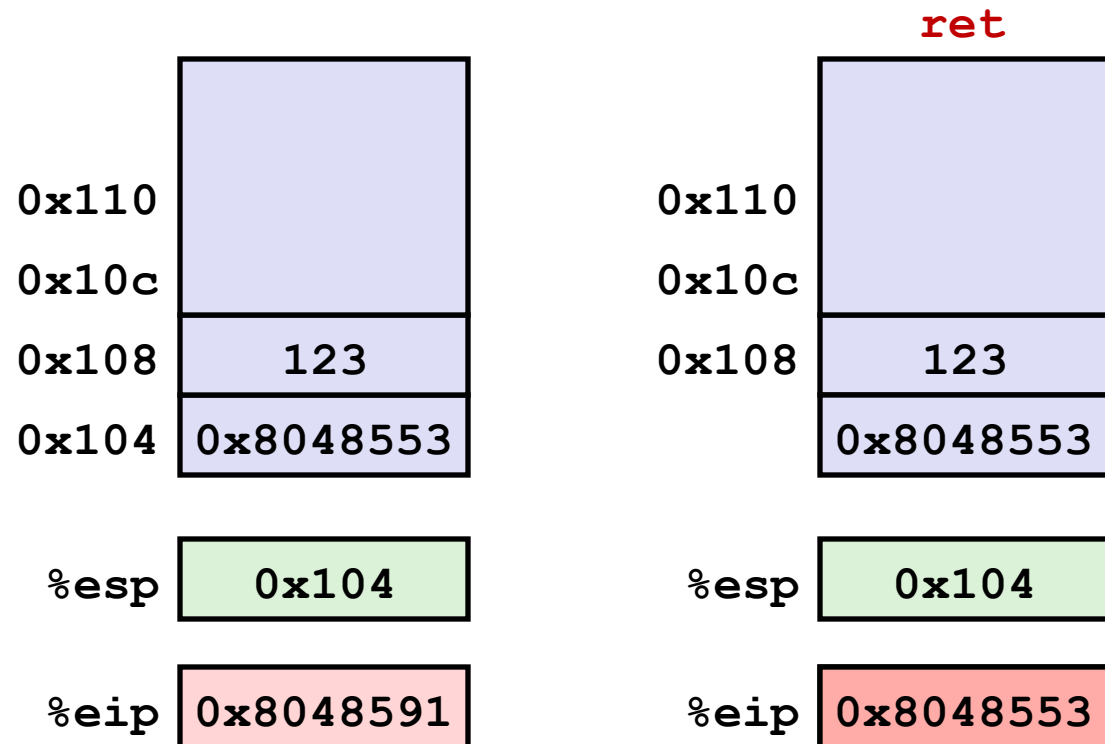
8048591:	c3	ret
----------	----	-----



`%eip`: program counter

Procedure Return Example

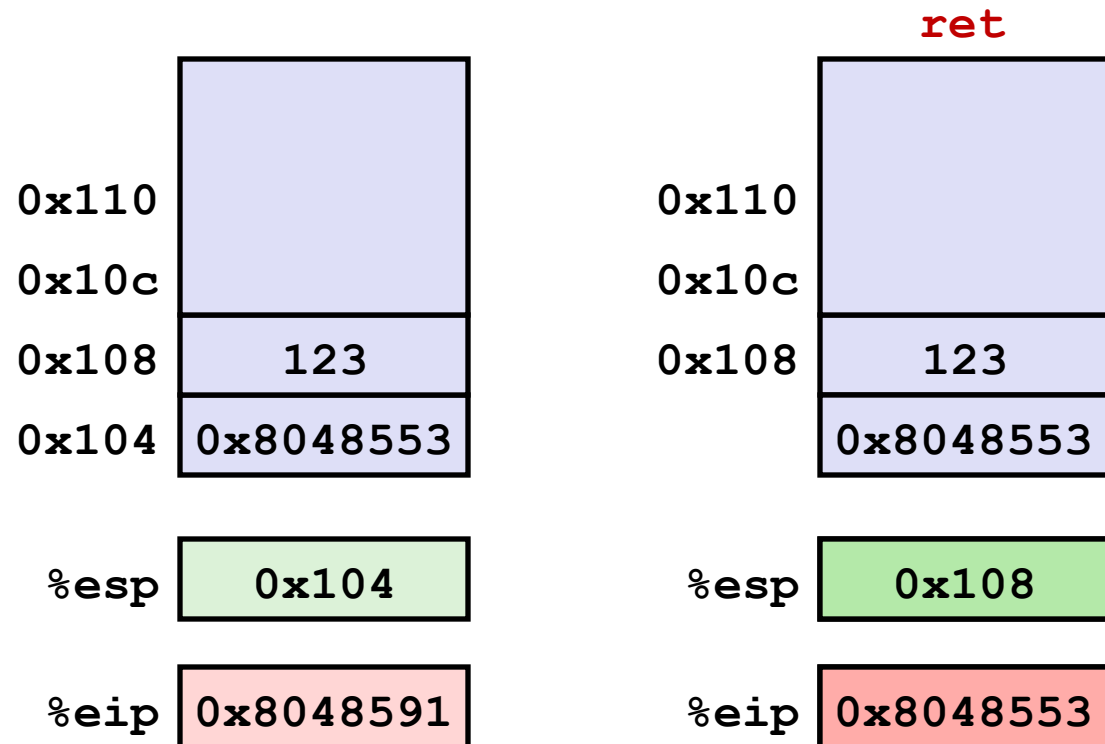
8048591:	c3	ret
----------	----	-----



`%eip`: program counter

Procedure Return Example

8048591:	c3	ret
----------	----	-----



`%eip`: program counter

Return Values

- **By convention, values returned by procedures are placed in the %eax register**
 - Choice of %eax is arbitrary, could have easily been a different register
- **Caller must make sure to save that register before calling a callee that returns a value**
 - Part of register-saving convention we'll see later
- **Callee placed return value (any type that can fit in 4 bytes – integer, float, pointer, etc.) into the %eax register**
 - For return values greater than 4 bytes, best to return a pointer to them
- **Upon return, caller finds the return value in the %eax register**

Procedures & Stacks

- Stacks in memory and stack operations
- The stack used to keep track of procedure calls
- Return addresses and return values
- Stack-based languages
- The Linux stack frame
- Passing arguments on the stack
- Allocating local variables on the stack
- Register-saving conventions
- Procedures and stacks on x64 architecture

Stack-Based Languages

- **Languages that support recursion**
 - e.g., C, Pascal, Java
 - 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 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
- **Stack allocated in *frames***
 - State for a single procedure instantiation

Call Chain Example

```

yoo (...)
{
  .
  .
  who ();
  .
  .
}

```

```

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

```

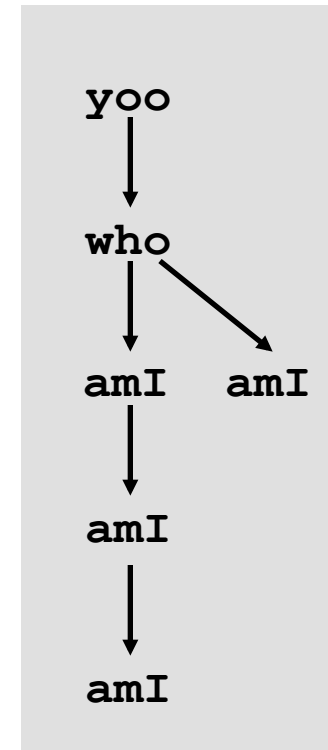
```

amI (...)
{
  .
  .
  amI ();
  .
  .
}

```

Procedure amI is recursive
(calls itself)

Example Call Chain



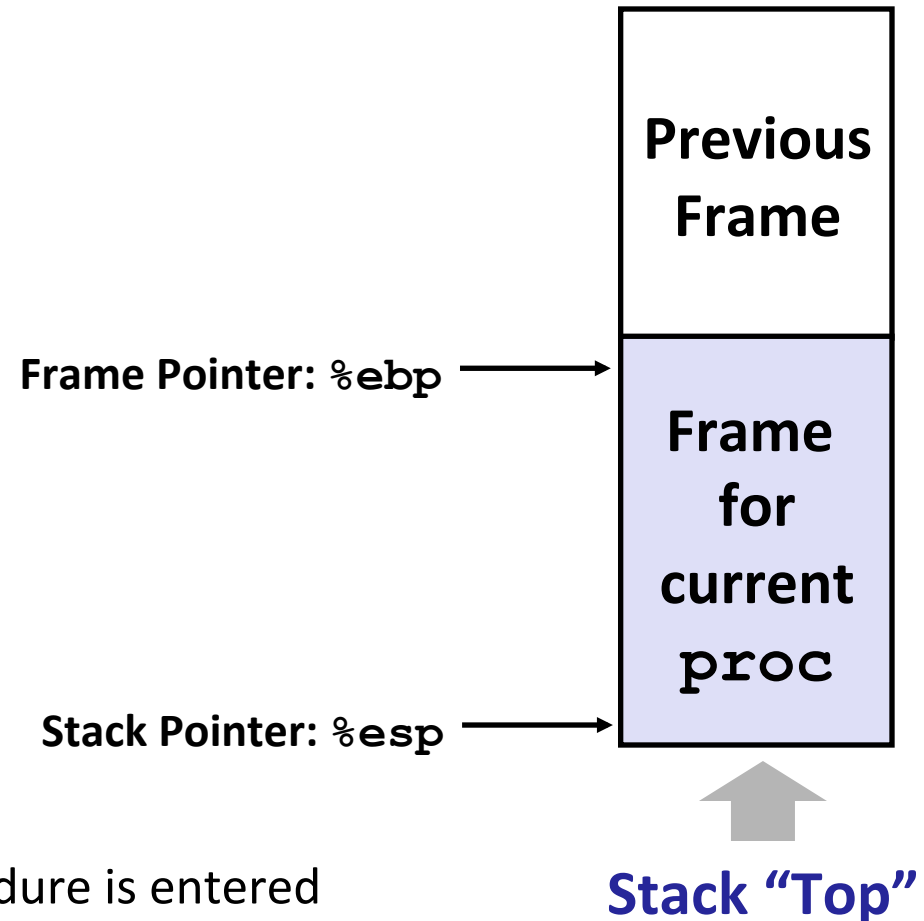
Stack Frames

■ Contents

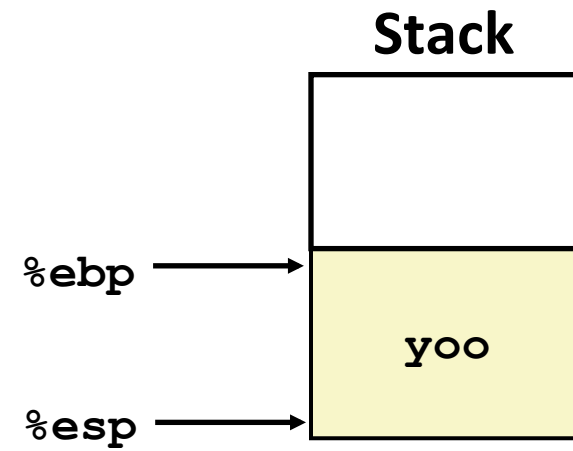
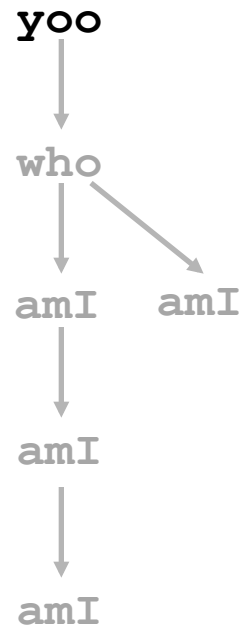
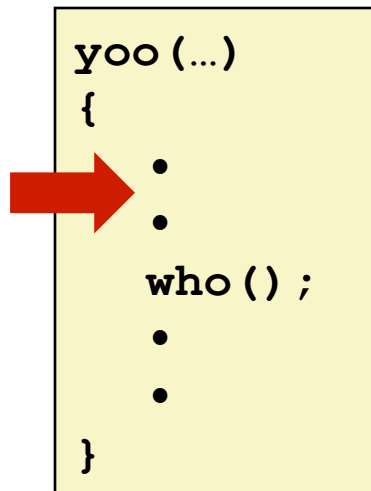
- Local variables
- Function arguments
- Return information
- Temporary space

■ Management

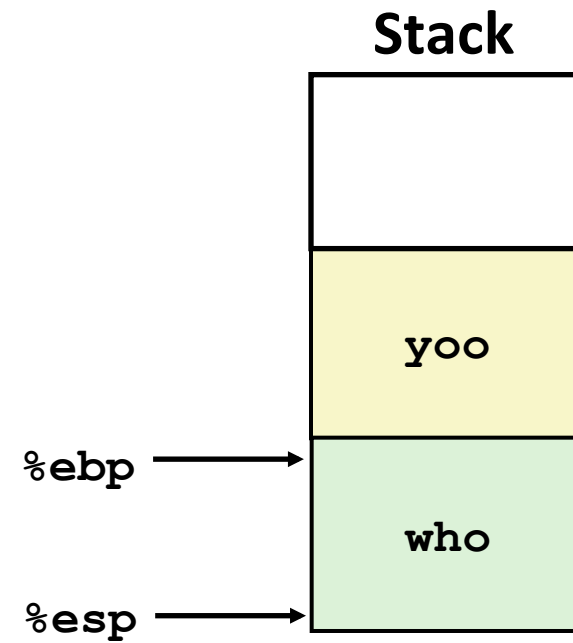
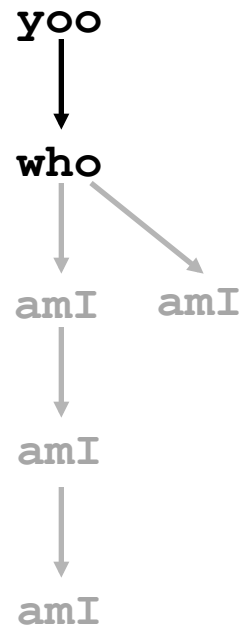
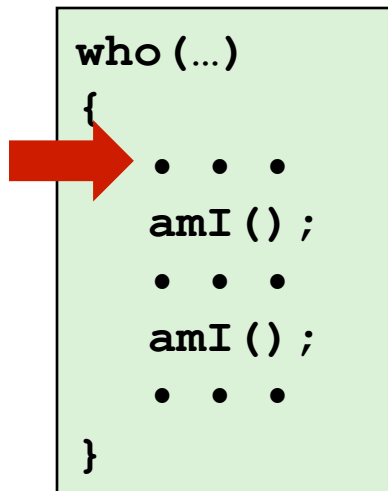
- Space allocated when procedure is entered
 - “Set-up” code
- Space deallocated upon return
 - “Finish” code



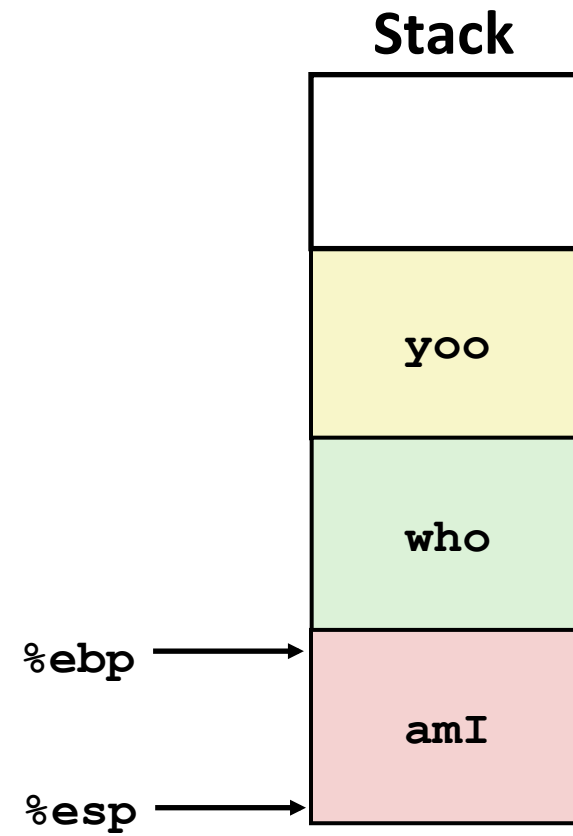
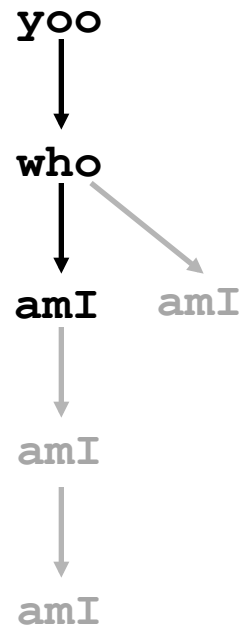
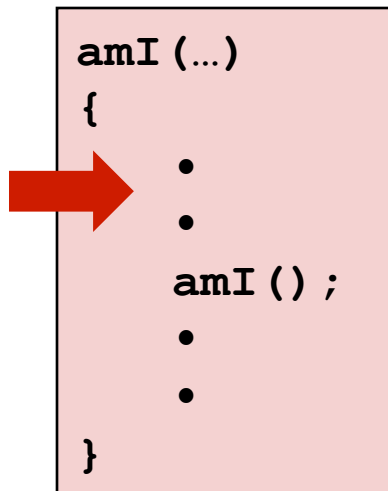
Example



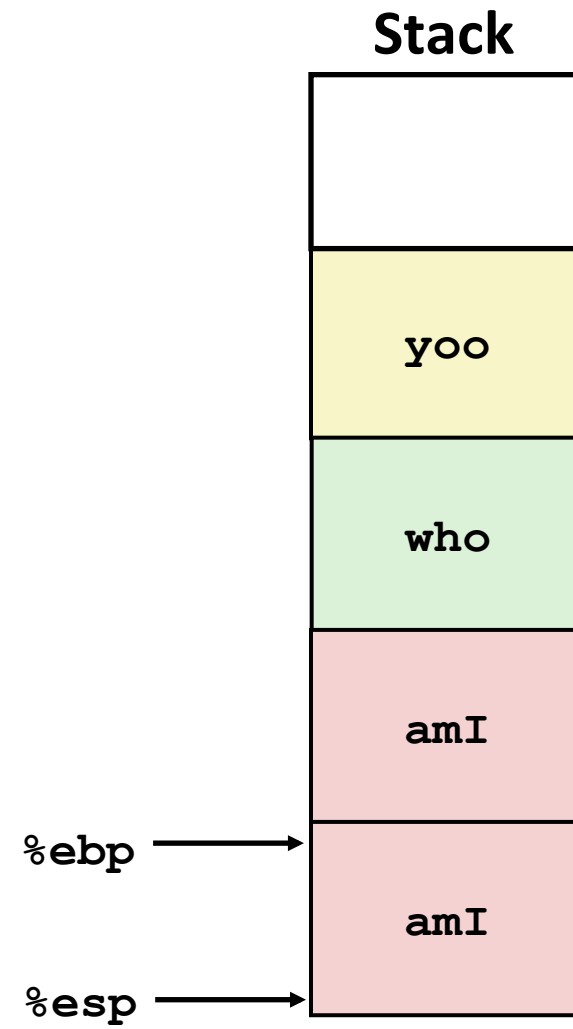
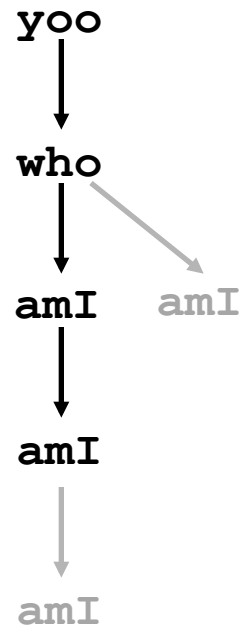
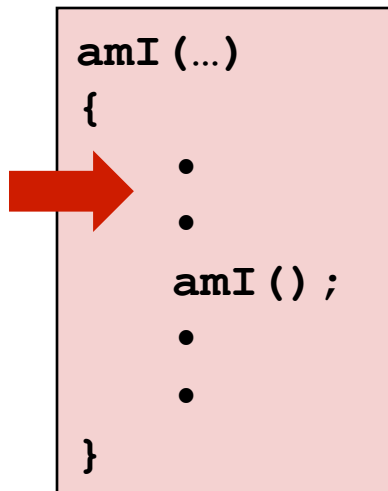
Example



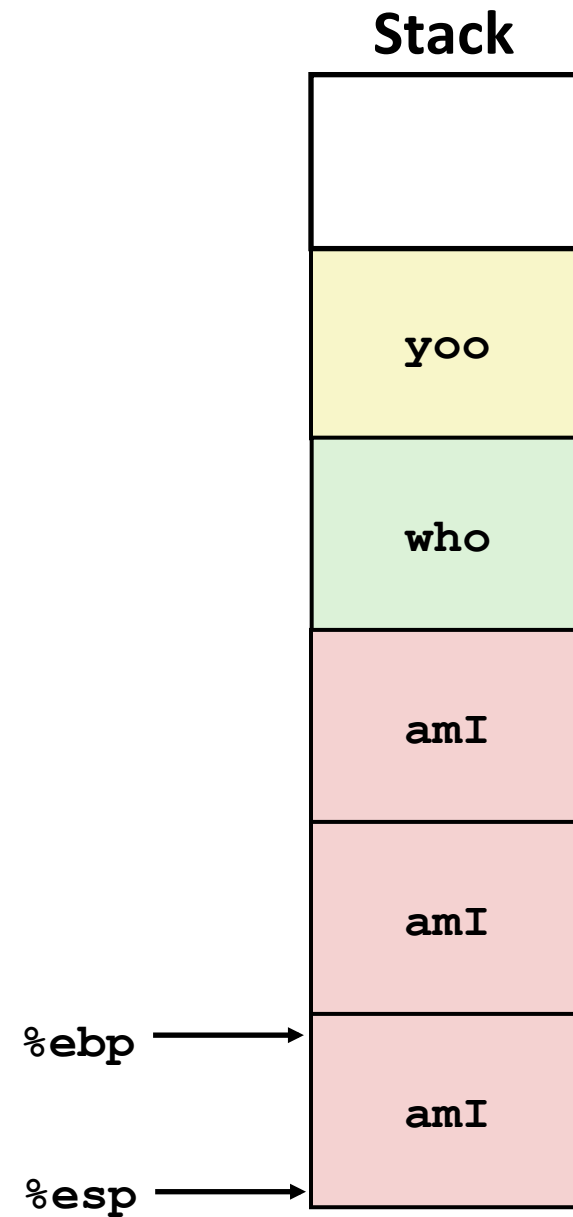
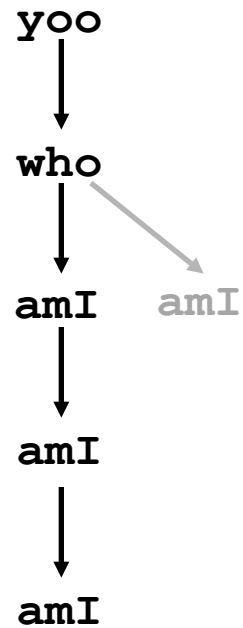
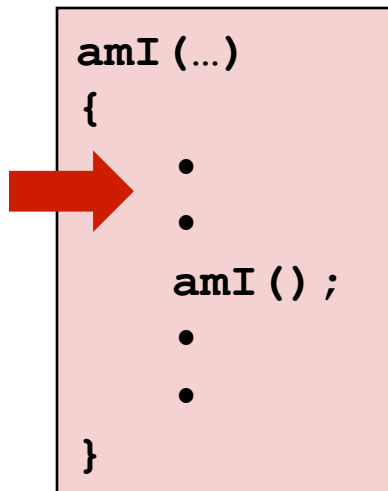
Example



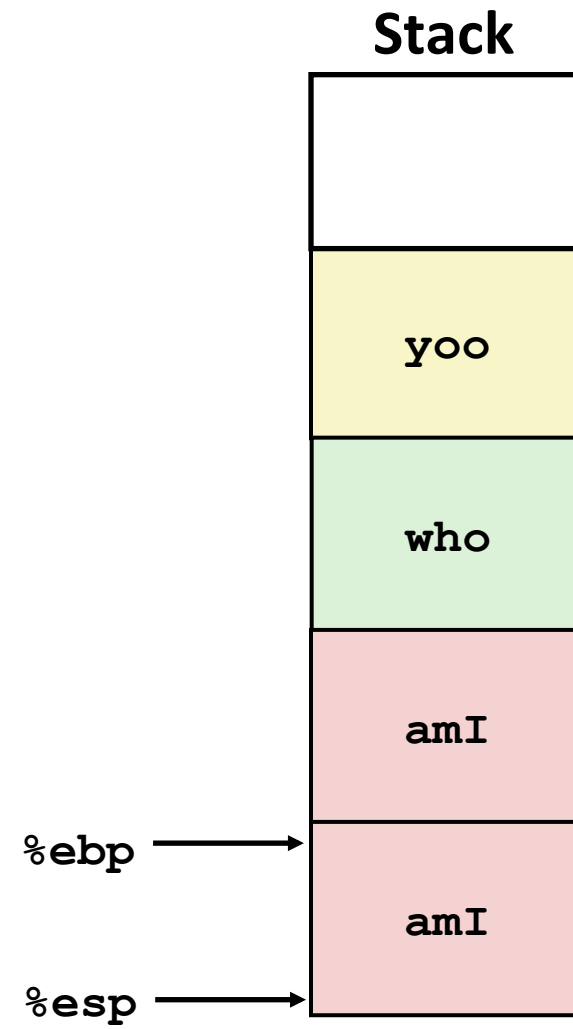
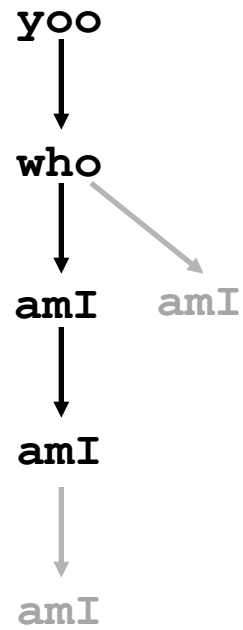
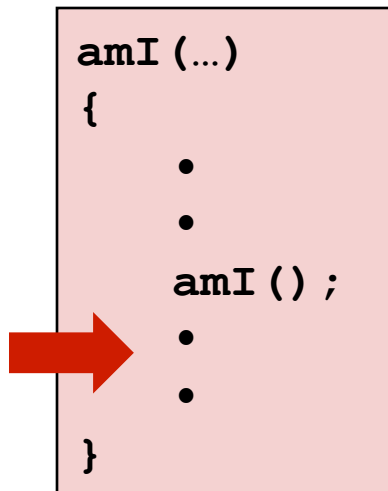
Example



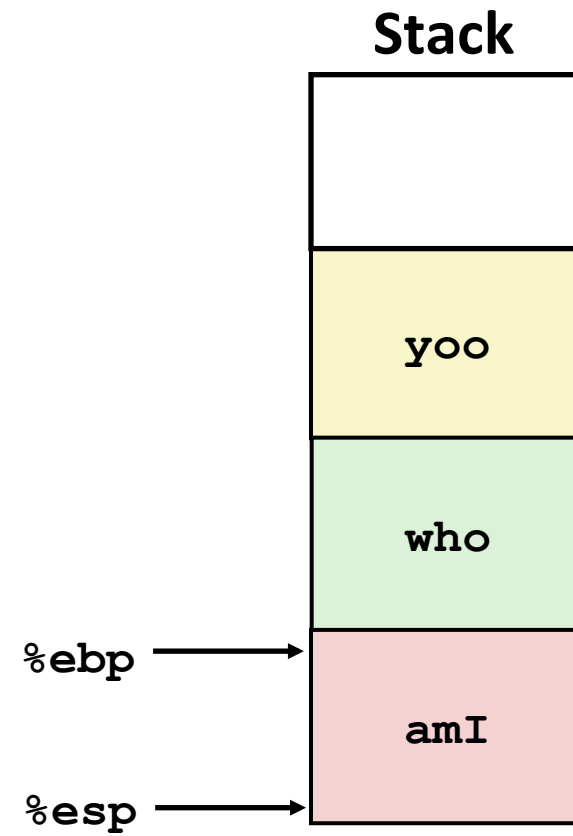
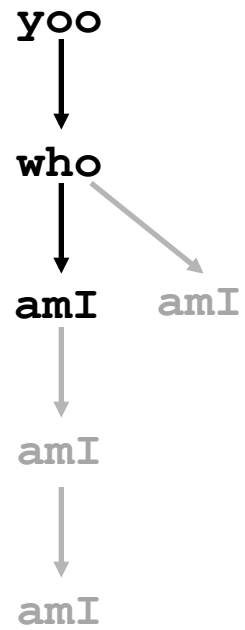
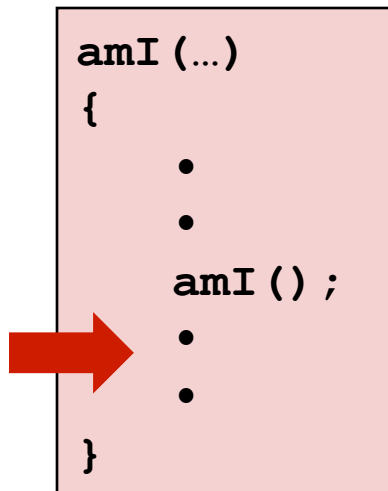
Example



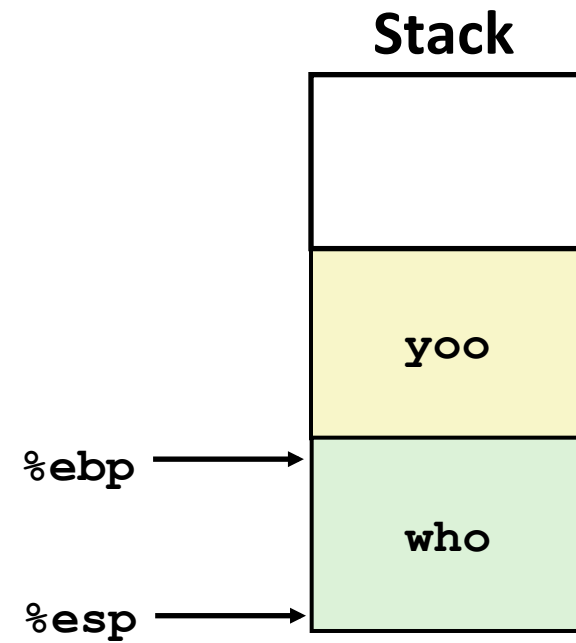
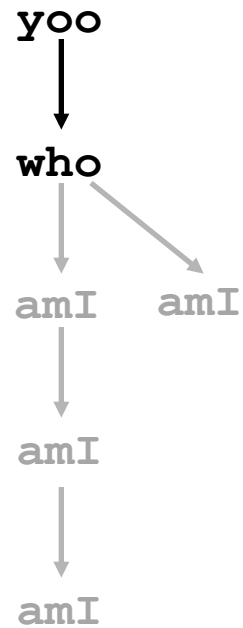
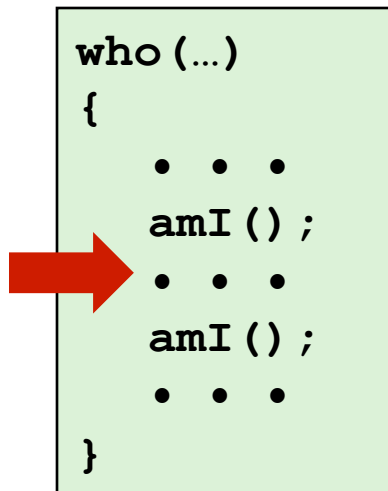
Example



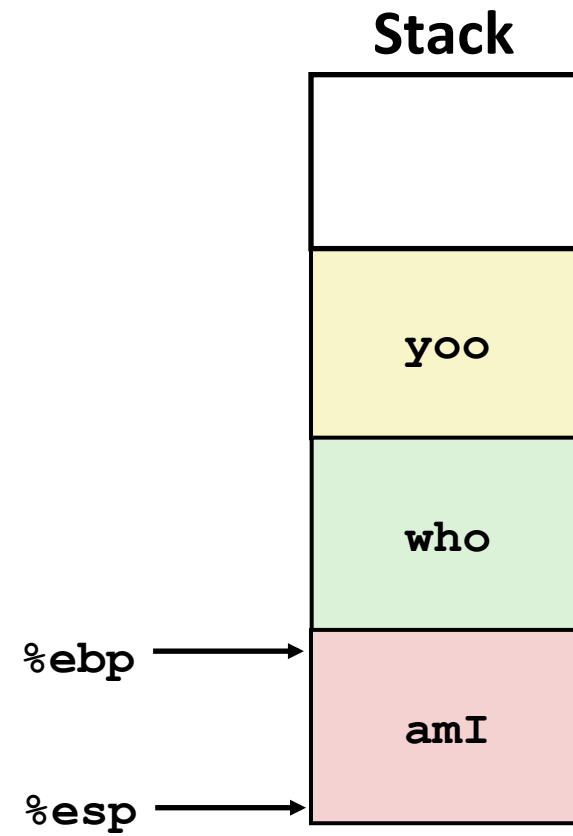
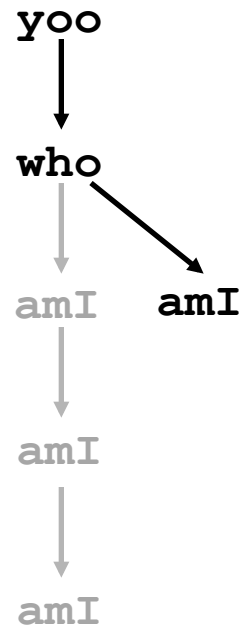
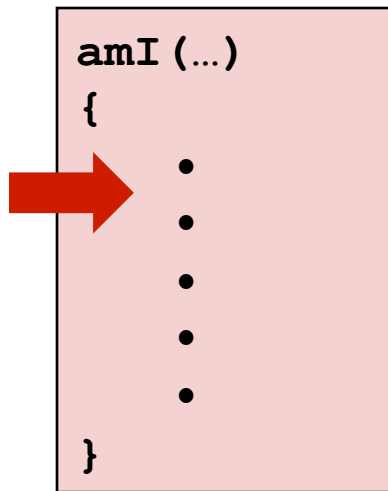
Example



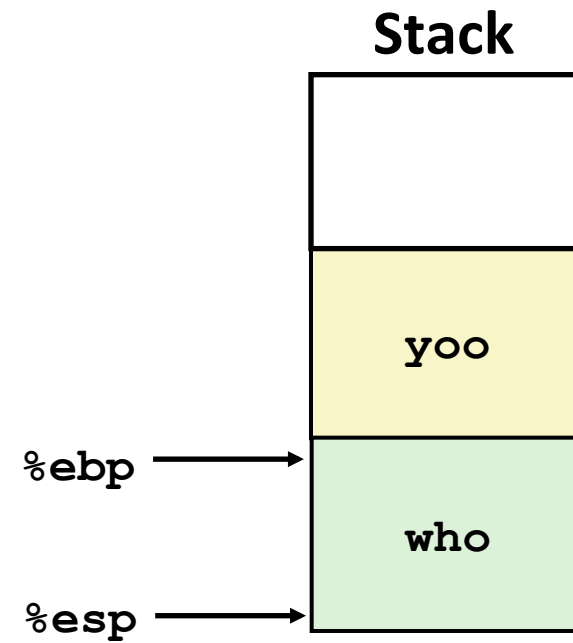
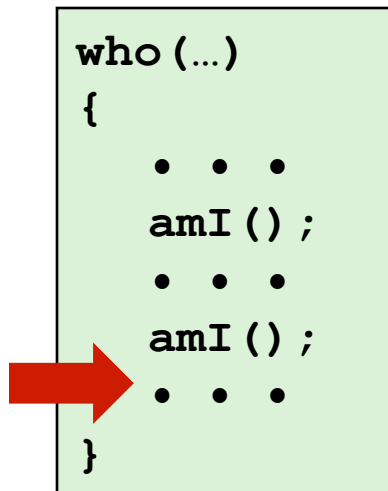
Example



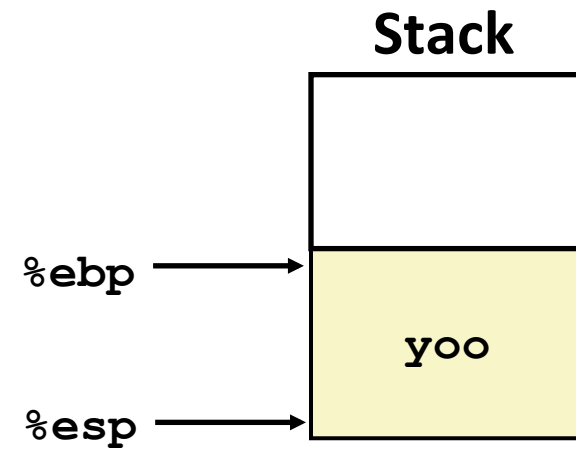
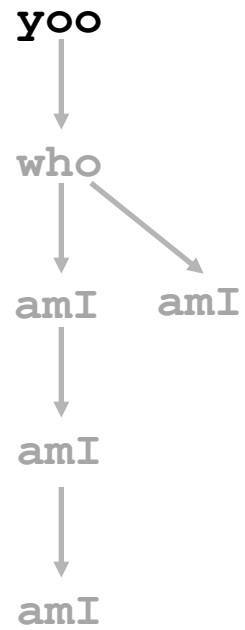
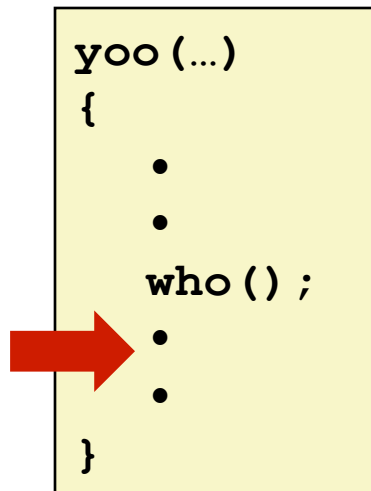
Example



Example



Example



Procedures & Stacks

- Stacks in memory and stack operations
- The stack used to keep track of procedure calls
- Return addresses and return values
- Stack-based languages
- The Linux stack frame
- Passing arguments on the stack
- Allocating local variables on the stack
- Register-saving conventions
- Procedures and stacks on x64 architecture

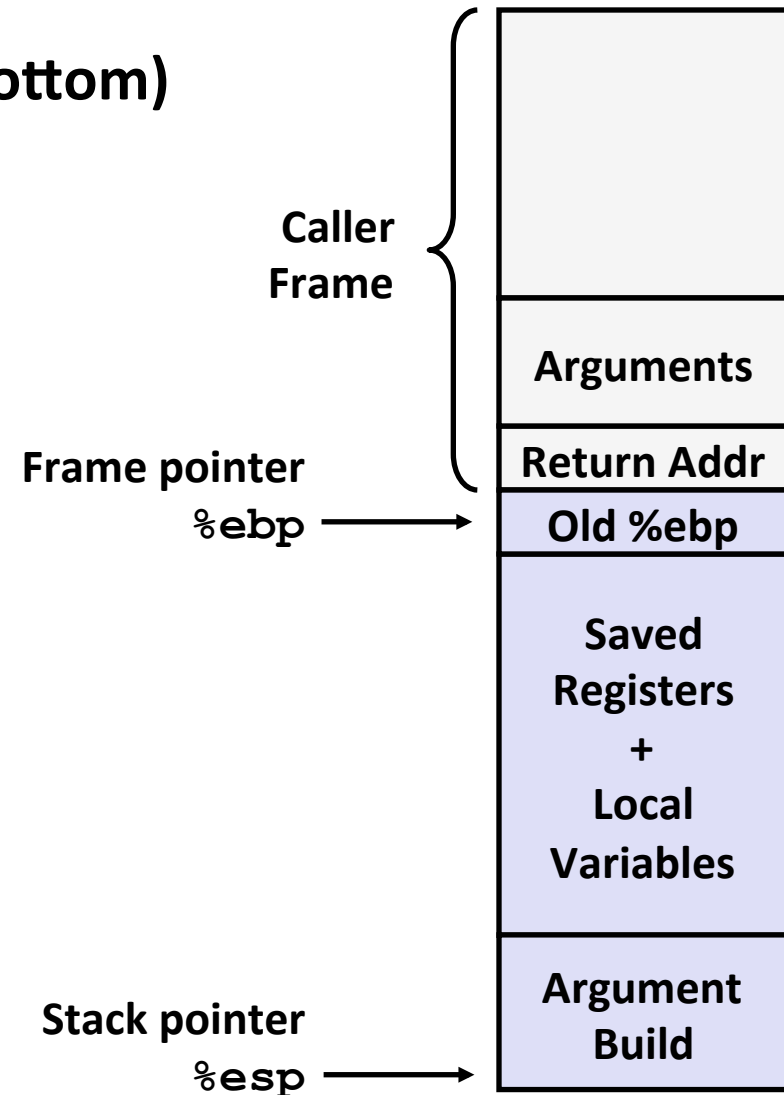
IA32/Linux Stack Frame

■ Current Stack Frame (“Top” to Bottom)

- “Argument build” area
(parameters for function about to be called)
- Local variables
(if can’t be kept in registers)
- Saved register context
(when reusing registers)
- Old frame pointer (for caller)

■ Caller’s Stack Frame

- Return address
 - Pushed by `call` instruction
- Arguments for this call



Revisiting swap

```
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
    swap(&zip1, &zip2);
}
```

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Revisiting swap

```
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
    swap(&zip1, &zip2);
}
```

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Calling swap from call_swap

```
call_swap:
    . . .
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    call swap
    . . .
```

Revisiting swap

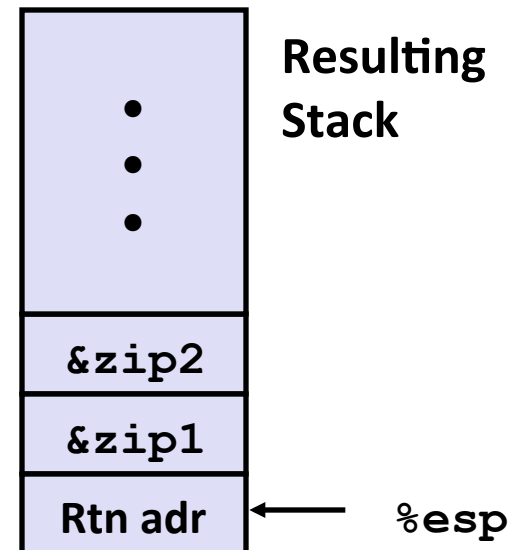
```
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
    swap(&zip1, &zip2);
}
```

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Calling swap from call_swap

```
call_swap:
    . . .
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    call swap
    . . .
```



Revisiting swap

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

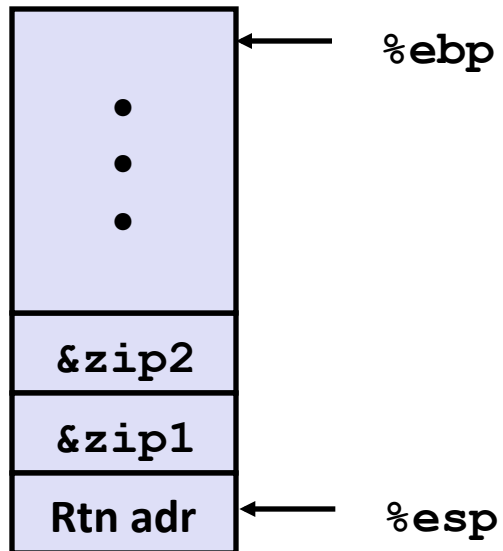
```
swap:
    pushl %ebp                }
    movl  %esp,%ebp          } Set Up
    pushl %ebx

    movl  12(%ebp),%ecx      }
    movl  8(%ebp),%edx       } Body
    movl  (%ecx),%eax
    movl  (%edx),%ebx
    movl  %eax,(%edx)
    movl  %ebx,(%ecx)

    movl  -4(%ebp),%ebx     }
    movl  %ebp,%esp        } Finish
    popl  %ebp
    ret
```

swap Setup #1

Entering Stack

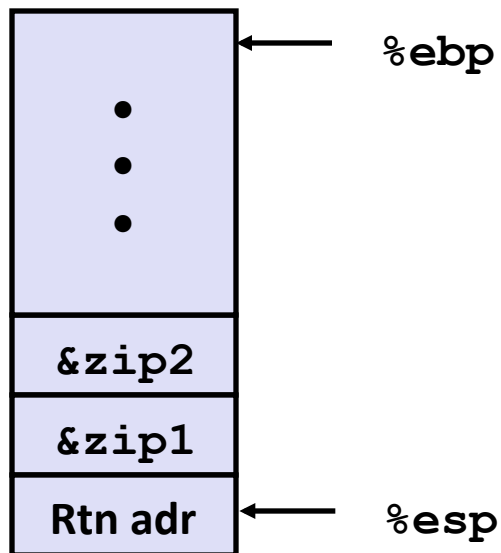


Resulting Stack?

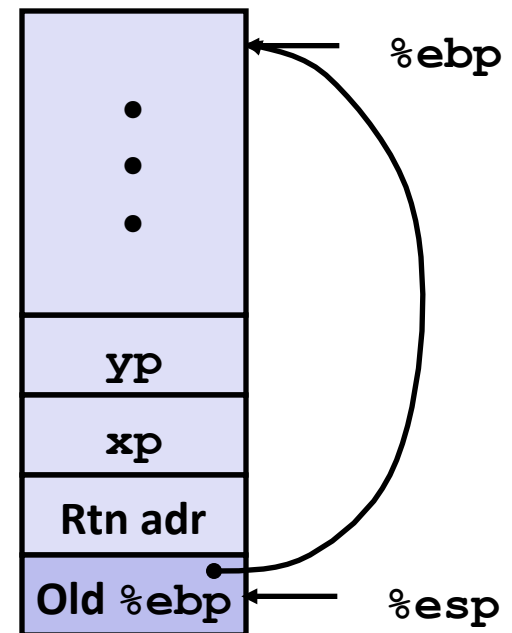
```
swap:  
    pushl %ebp  
    movl %esp,%ebp  
    pushl %ebx
```

swap Setup #1

Entering Stack



Resulting Stack



`swap:`

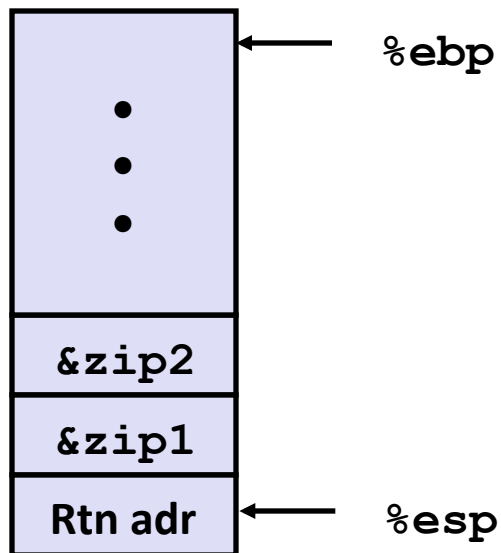
```

pushl %ebp
movl %esp,%ebp
pushl %ebx

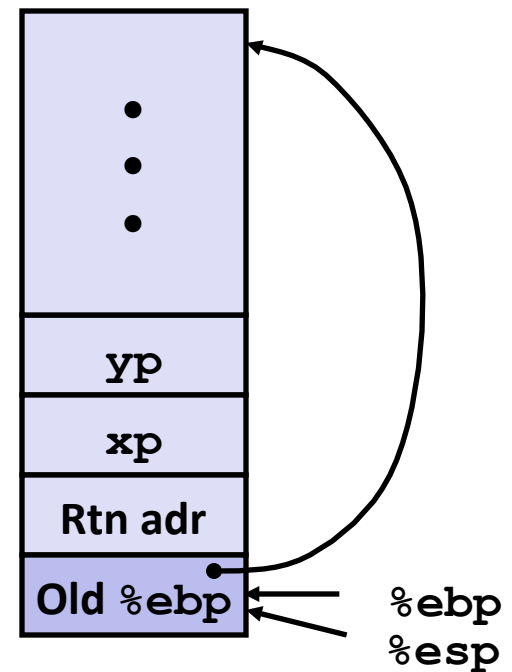
```


swap Setup #2

Entering Stack



Resulting Stack

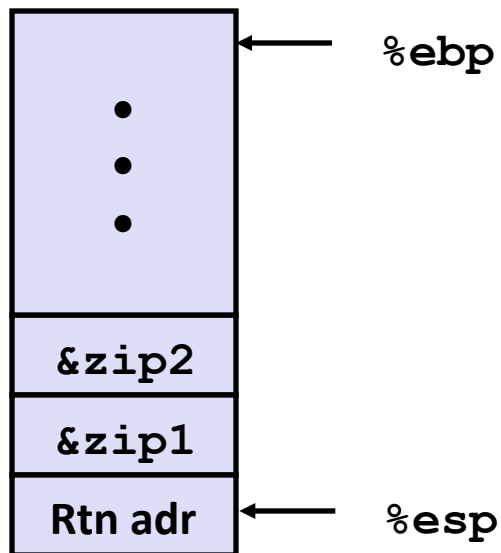


```

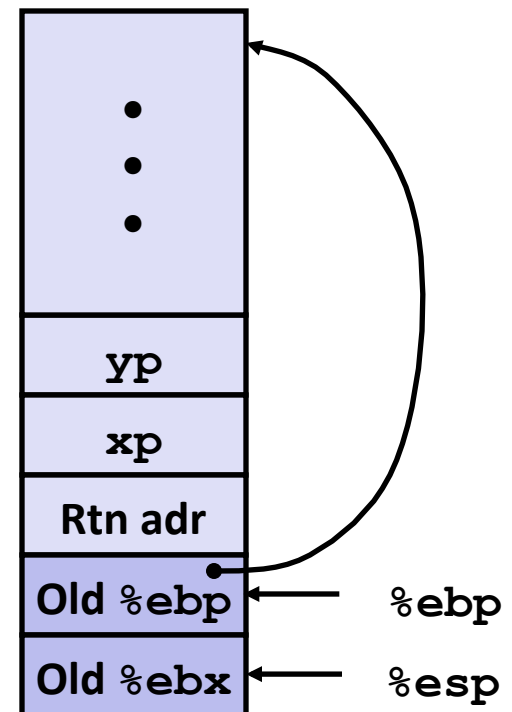
swap:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
  
```

swap Setup #3

Entering Stack



Resulting Stack

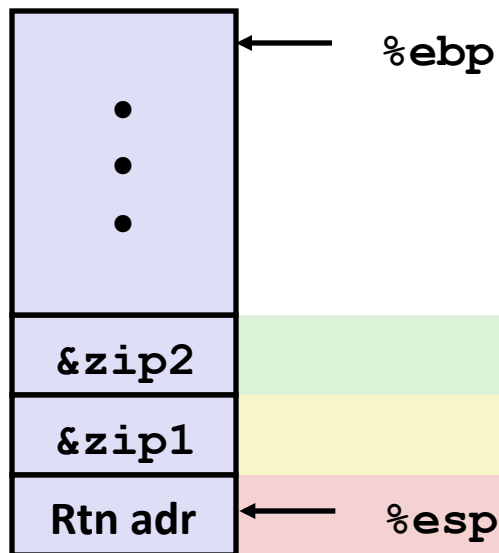


```

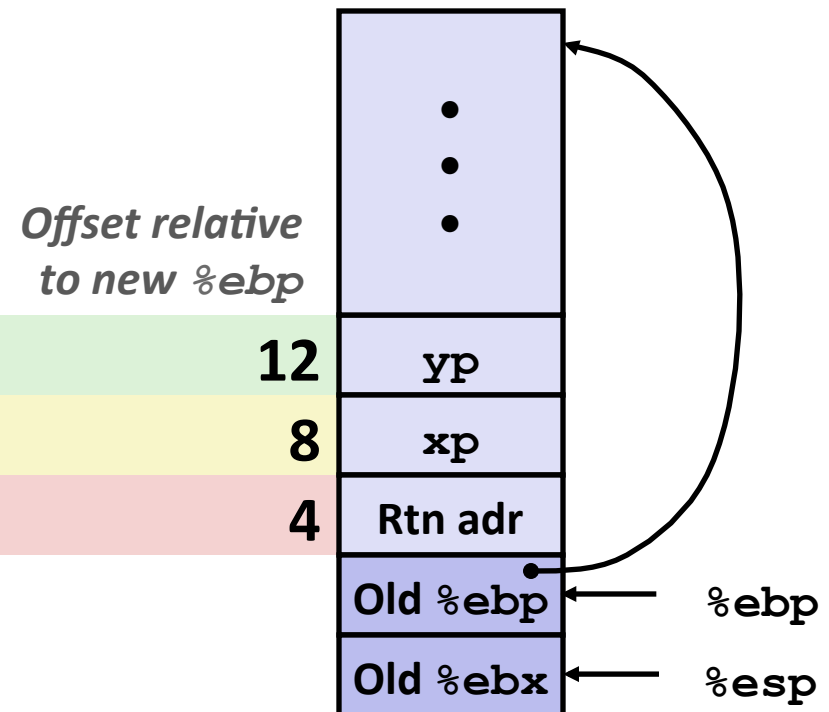
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
  
```

swap Body

Entering Stack



Resulting Stack



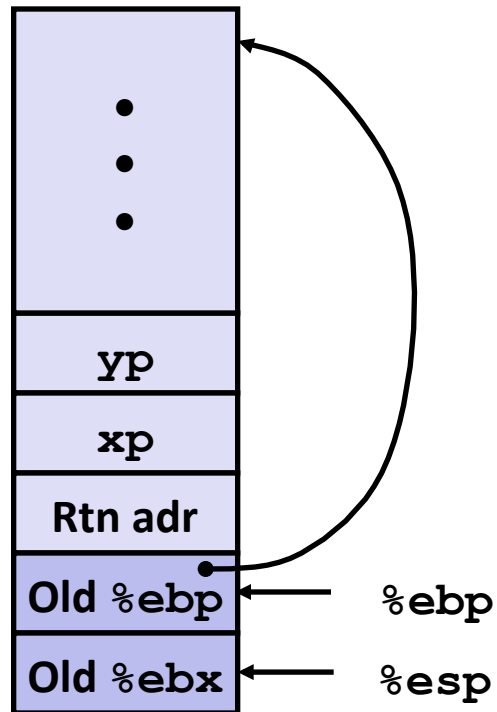
```

movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx # get xp
. . .

```

swap Finish #1

swap' s Stack

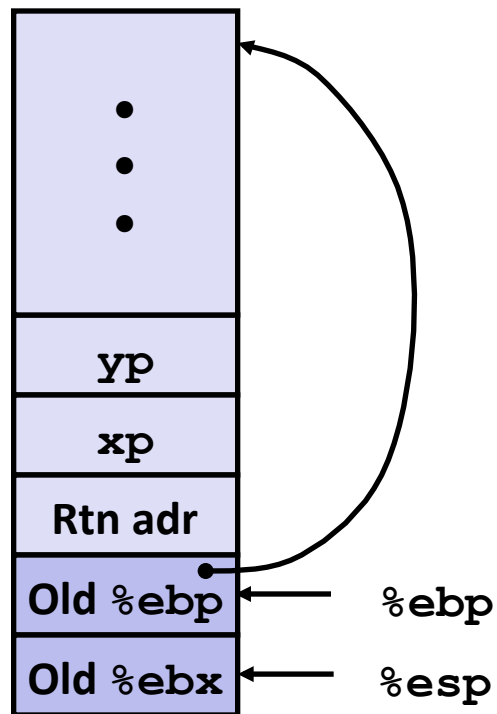


Resulting Stack?

```
movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret
```

swap Finish #1

swap's Stack

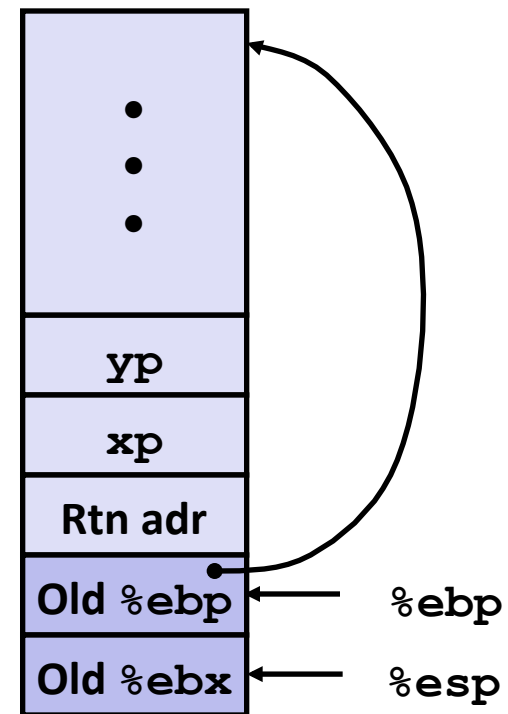


```

movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret

```

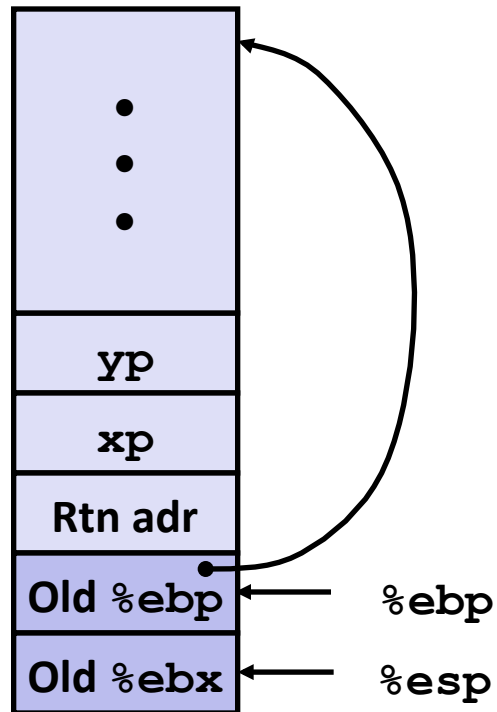
Resulting Stack



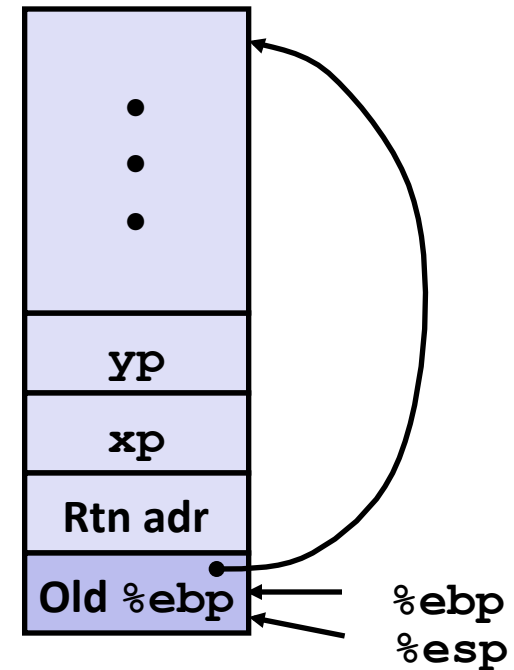
Observation: Saved and restored register %ebx

swap Finish #2

swap's Stack



Resulting Stack



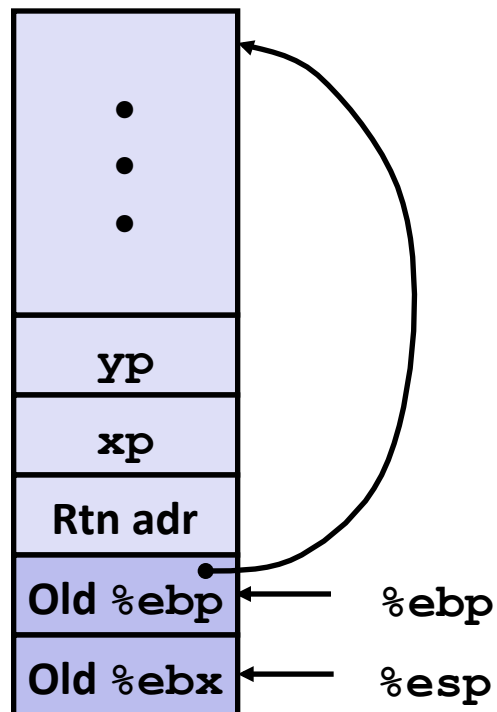
```

movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret

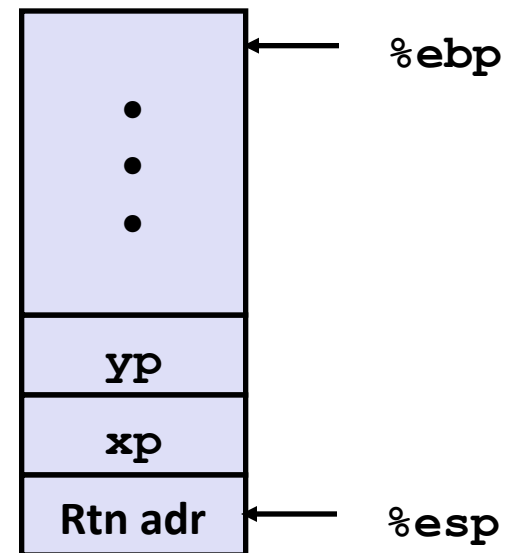
```

swap Finish #3

swap's Stack



Resulting Stack



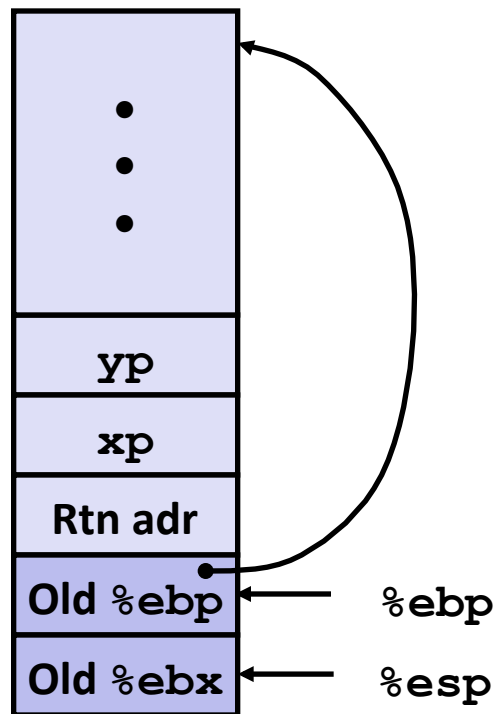
```

movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret

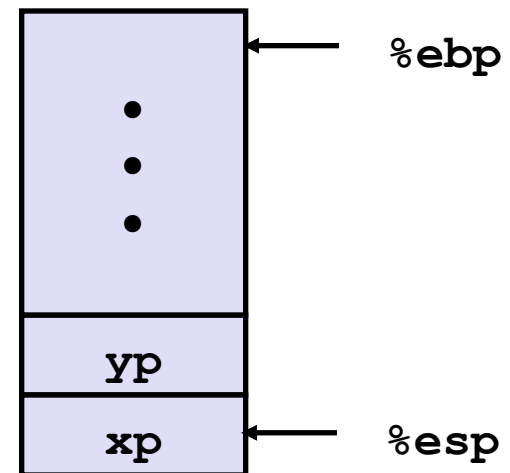
```

swap Finish #4

swap's Stack



Resulting Stack



```

movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret

```


Disassembled swap

080483a4 <swap>:

```

80483a4:  55          push   %ebp
80483a5:  89 e5      mov    %esp, %ebp
80483a7:  53          push   %ebx
80483a8:  8b 55 08   mov    0x8(%ebp), %edx
80483ab:  8b 4d 0c   mov    0xc(%ebp), %ecx
80483ae:  8b 1a      mov    (%edx), %ebx
80483b0:  8b 01      mov    (%ecx), %eax
80483b2:  89 02      mov    %eax, (%edx)
80483b4:  89 19      mov    %ebx, (%ecx)
80483b6:  5b        pop    %ebx
80483b7:  c9        leave
80483b8:  c3        ret

```



```

mov    %ebp, %esp
pop    %ebp

```

Calling Code

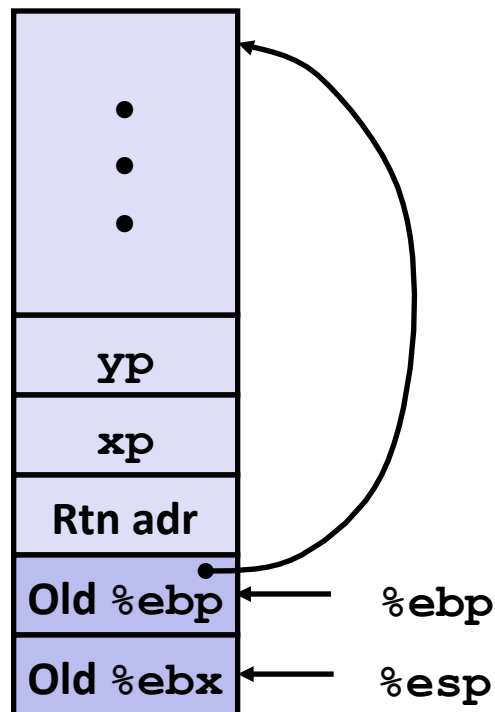
```

8048409:  e8 96 ff ff ff   call 80483a4 <swap>
804840e:  8b 45 f8        mov 0xffffffff8(%ebp), %eax

```

swap Finish #4

swap' s Stack

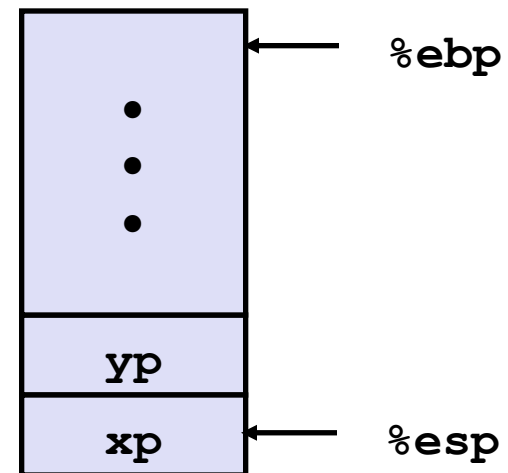


```

movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret

```

Resulting Stack



■ Observation

- Saved & restored register **%ebx**
- Didn't do so for **%eax, %ecx, or %edx**

Procedures & Stacks

- **Stacks in memory and stack operations**
- **The stack used to keep track of procedure calls**
- **Return addresses and return values**
- **Stack-based languages**
- **The Linux stack frame**
- **Passing arguments on the stack**
- **Allocating local variables on the stack**
- **Register-saving conventions**
- **Procedures and stacks on x64 architecture**

Register Saving Conventions

■ When procedure `yoo` calls `who`:

- `yoo` is the *caller*
- `who` is the *callee*

■ Can a register be used for temporary storage?

```
yoo:
    . . .
    movl $12345, %edx
    call who
    addl %edx, %eax
    . . .
    ret
```

```
who:
    . . .
    movl 8(%ebp), %edx
    addl $98195, %edx
    . . .
    ret
```

- Contents of register `%edx` overwritten by `who`

Register Saving Conventions

- When procedure *yoo* calls *who*:
 - *yoo* is the *caller*
 - *who* is the *callee*
- Can a register be used for temporary storage?
- Conventions
 - “*Caller Save*”
 - Caller saves temporary values in its frame before calling
 - “*Callee Save*”
 - Callee saves temporary values in its frame before using

IA32/Linux Register Usage

■ `%eax`, `%edx`, `%ecx`

- Caller saves prior to call if values are used later

■ `%eax`

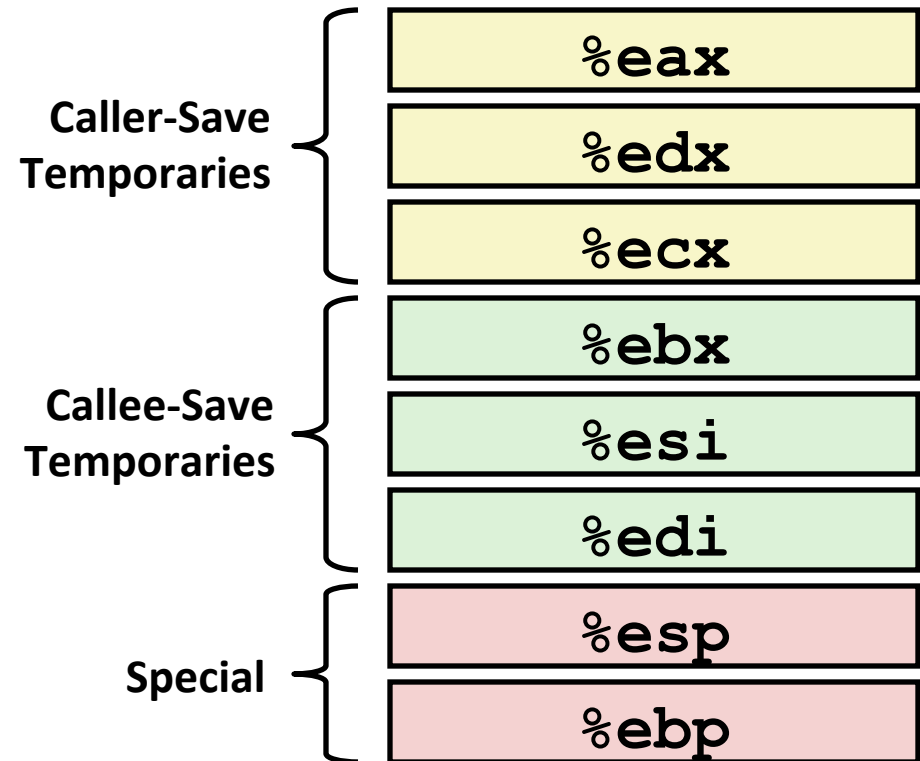
- also used to return integer value

■ `%ebx`, `%esi`, `%edi`

- Callee saves if wants to use them

■ `%esp`, `%ebp`

- special form of callee save – restored to original values upon exit from procedure



Example: Pointers to Local Variables

Recursive Procedure

```
void s_helper
(int x, int *accum)
{
    if (x <= 1)
        return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper (x-1, accum);
    }
}
```

Top-Level Call

```
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- Pass pointer to update location

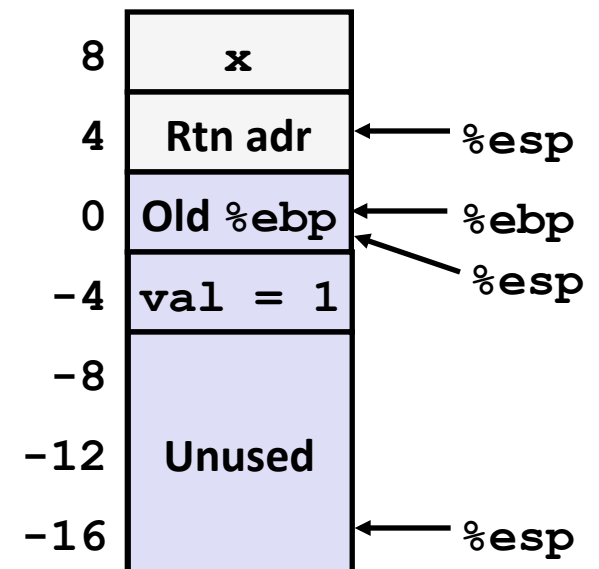
Creating & Initializing Pointer

```
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- Variable `val` must be stored on stack
 - Because: Need to create pointer to it
- Compute pointer as `-4 (%ebp)`
- Push on stack as second argument

Initial part of `sfact`

```
_sfact:
    pushl %ebp          # Save %ebp
    movl  %esp,%ebp    # Set %ebp
    subl  $16,%esp     # Add 16 bytes
    movl  8(%ebp),%edx  # edx = x
    movl  $1,-4(%ebp)  # val = 1
```



Passing Pointer

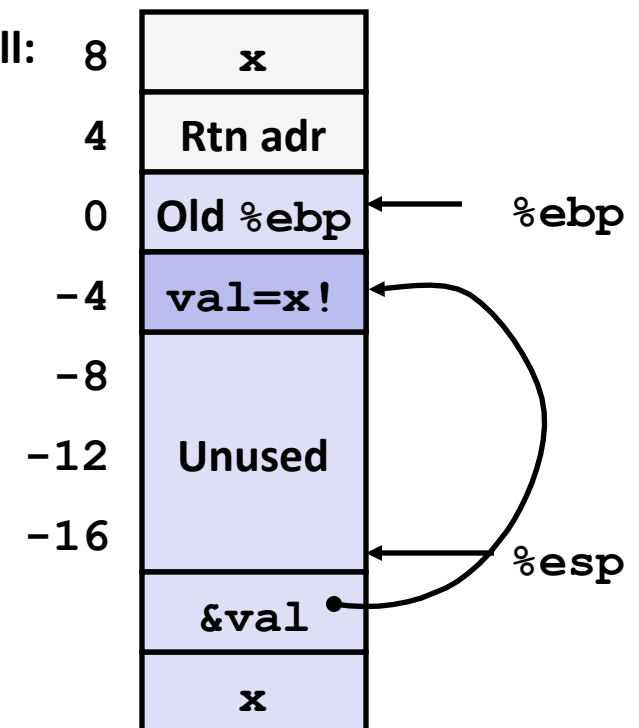
```
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- Variable `val` must be stored on stack
 - Because: Need to create pointer to it
- Compute pointer as `-4 (%ebp)`
- Push on stack as second argument

Calling `s_helper` from `sfact`

```
leal -4(%ebp), %eax # Compute &val
pushl %eax          # Push on stack
pushl %edx          # Push x
call s_helper       # call
movl -4(%ebp), %eax # Return val
. . .               # Finish
```

Stack at time of call:



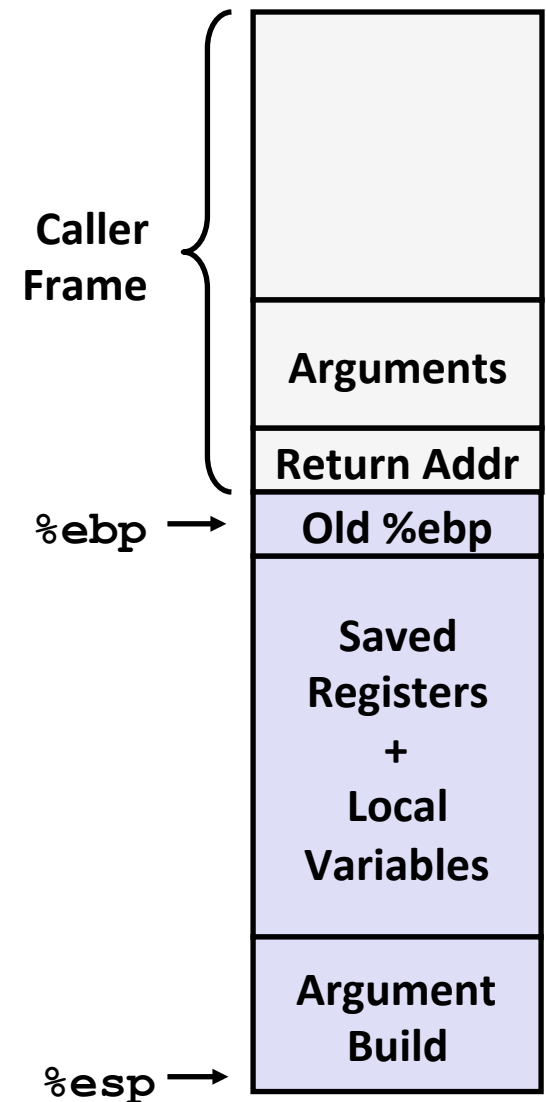
IA 32 Procedure Summary

■ Important points:

- IA32 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

- Can safely store values in local stack frame and in callee-saved registers
- Put function arguments at top of stack
- Result returned in **%eax**



Procedures & Stacks

- Stacks in memory and stack operations
- The stack used to keep track of procedure calls
- Return addresses and return values
- Stack-based languages
- The Linux stack frame
- Passing arguments on the stack
- Allocating local variables on the stack
- Register-saving conventions
- Procedures and stacks on x64 architecture

x86-64 Procedure Calling Convention

- **Doubling of registers makes us less dependent on stack**
 - Store argument in registers
 - Store temporary variables in registers
- **What do we do if we have too many arguments or too many temporary variables?**

x86-64 64-bit Registers: Usage Conventions

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

Revisiting swap, IA32 vs. x86-64 versions

```

swap:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
} Set Up

    movl 12(%ebp), %ecx
    movl 8(%ebp), %edx
    movl (%ecx), %eax
    movl (%edx), %ebx
    movl %eax, (%edx)
    movl %ebx, (%ecx)
} Body

    movl -4(%ebp), %ebx
    movl %ebp, %esp
    popl %ebp
    ret
} Finish

```

```

swap (64-bit long ints):
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    ret

```

- **Arguments passed in registers**
 - First (**xp**) in `%rdi`,
 - second (**yp**) in `%rsi`
 - 64-bit pointers
- **No stack operations required (except `ret`)**
- **Avoiding stack**
 - Can hold all local information in registers

X86-64 procedure call highlights

- **Arguments (up to first 6) in registers**
 - Faster to get these values from registers than from stack in memory
- **Local variables also in registers (if there is room)**
- **callq instruction stores 64-bit return address on stack**
 - Address pushed onto stack, decrementing %rsp by 8
- **No frame pointer**
 - All references to stack frame made relative to %rsp; eliminates need to update %ebp/%rbp, which is now available for general-purpose use
- **Functions can access memory up to 128 bytes beyond %rsp: the “red zone”**
 - Can store some temps on stack without altering %rsp
- **Registers still designated “caller-saved” or “callee-saved”**

x86-64 Stack Frames

- **Often (ideally), x86-64 functions need no stack frame at all**
 - Just a return address is pushed onto the stack when a function call is made
- **A function *does* need a stack frame when it:**
 - Has too many local variables to hold in registers
 - Has local variables that are arrays or structs
 - Uses the address-of operator (&) to compute the address of a local variable
 - Calls another function that takes more than six arguments
 - Needs to save the state of callee-save registers before modifying them

Example

```

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)
    movl   $2,24(%rsp)
    movw   $3,28(%rsp)
    movb   $4,31(%rsp)
    . . .

```

Return address to caller of call_proc

← %rsp

**NB: Details may vary
depending on compiler.**

Example

```

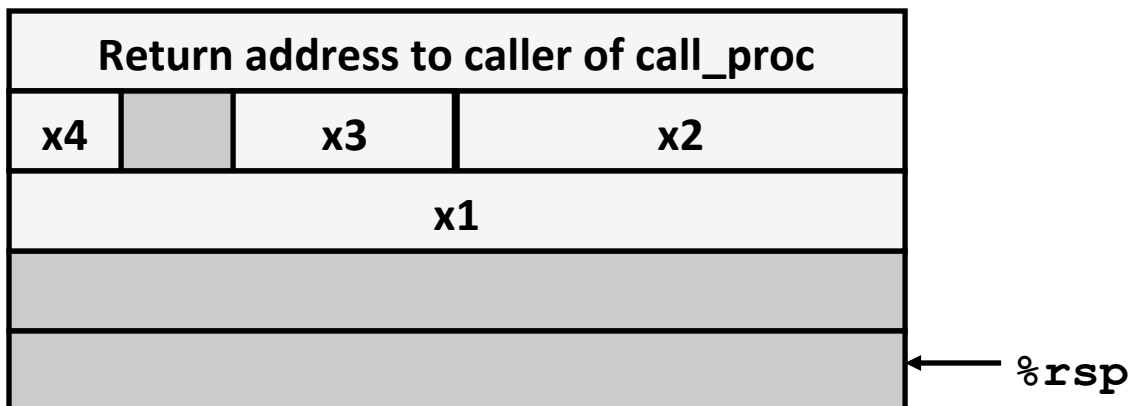
long int call_proc()
{
    long  x1 = 1;
    int   x2 = 2;
    short x3 = 3;
    char  x4 = 4;
    proc(x1, &x1, x2, &x2,
        x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}

```

```

call_proc:
    subq  $32,%rsp
    movq  $1,16(%rsp)
    movl  $2,24(%rsp)
    movw  $3,28(%rsp)
    movb  $4,31(%rsp)
    . . .

```



Example

```
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:
```

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

Return address to caller of call_proc

x4		x3	x2
x1			
Arg 8			
Arg 7			

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

← %rsp

Example

```

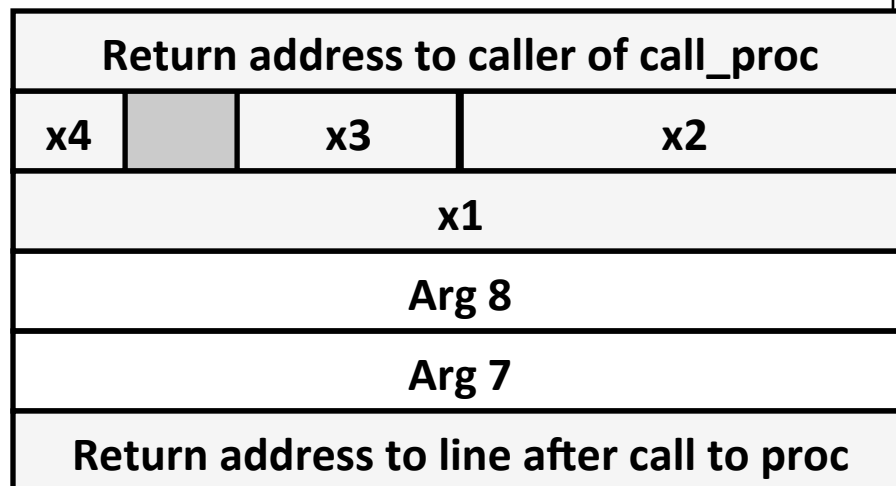
long int call_proc()
{
    long  x1 = 1;
    int   x2 = 2;
    short x3 = 3;
    char  x4 = 4;
    proc(x1, &x1, x2, &x2,
        x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}

```

```

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

```



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

← %rsp

Example

```

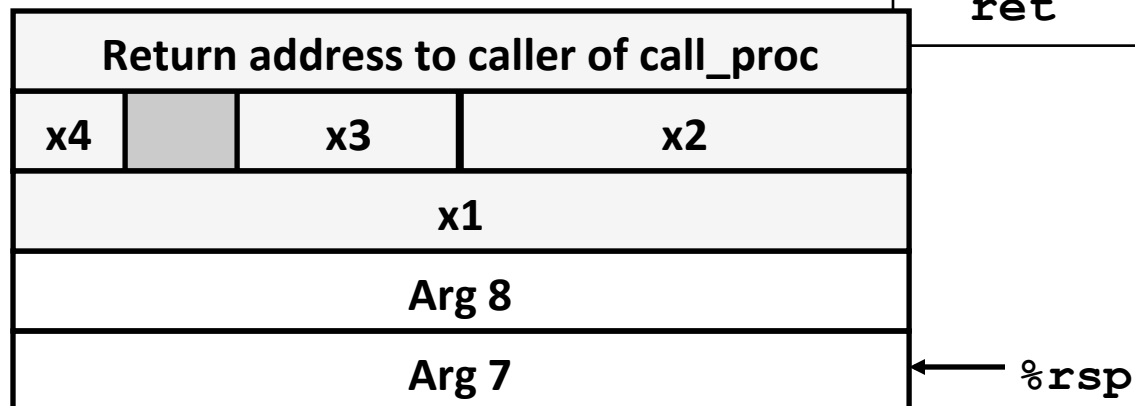
long int call_proc()
{
    long  x1 = 1;
    int   x2 = 2;
    short x3 = 3;
    char  x4 = 4;
    proc(x1, &x1, x2, &x2,
        x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}

```

```

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

```



Example

```

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

x86-64 Procedure Summary

- **Heavy use of registers (faster than using stack in memory)**
 - Parameter passing
 - More temporaries since more registers

- **Minimal use of stack**
 - Sometimes none
 - When needed, allocate/deallocate entire frame at once
 - No more frame pointer: address relative to stack pointer

- **More room for compiler optimizations**
 - Prefer to store data in registers rather than memory
 - Minimize modifications to stack pointer