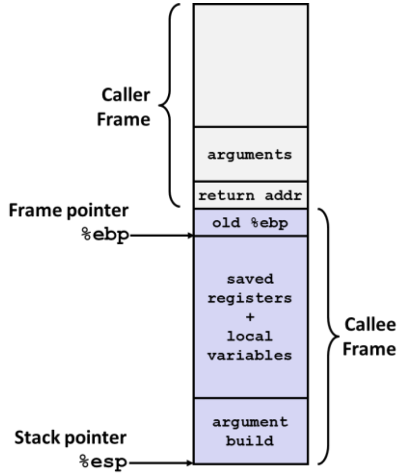
The HW/SW Interface

The x86 ISA: Procedures



Recap: Loops

- Do-While

```
do
  body;
while (test);
```

```
loop:
  body;
  if (test) goto loop;
```

- While

```
while (test)
  body;
```

```
  if (!test) goto done;
  do
    body;
  while (test);
done:
```

```
  if (!test) goto done;
loop:
  body;
  if (test) goto loop;
done:
```

- For

```
for (init; test; update)
  body;
```

```
init;
while (test) {
  body;
  update;
}
```

```
  init;
  if (!test) goto done;
  do {
    body;
    update;
  } while (test);
done:
```

```
  init;
  if (!test) goto done;
loop:
  body;
  update;
  if (test) goto loop;
done:
```

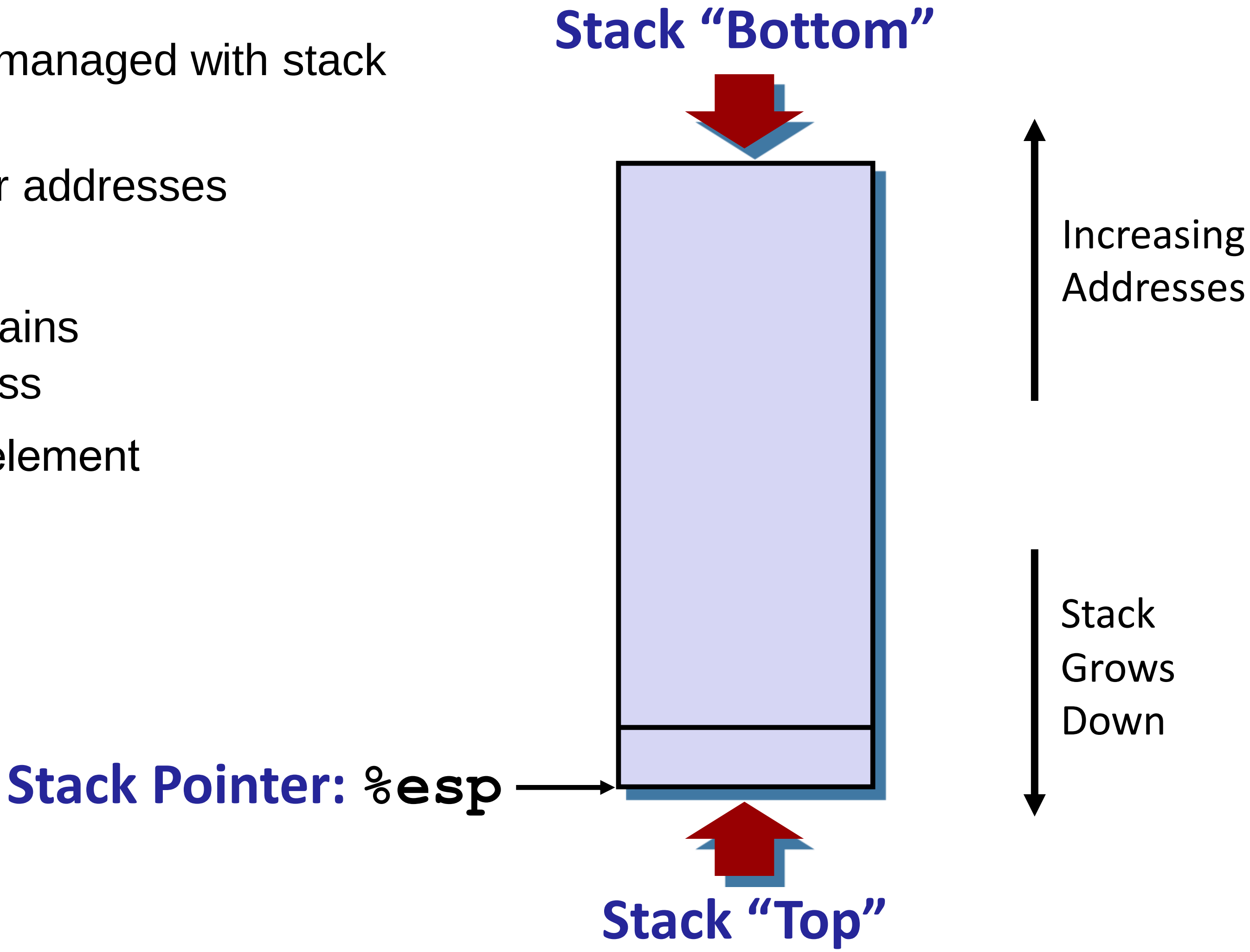
Procedures

- **IA 32 Procedures**
  - **Stack Structure**
  - Calling Conventions
  - Illustrations of Recursion & Pointers
- x86-64 Procedures

Acknowledgement: slides based on the cs:app2e material

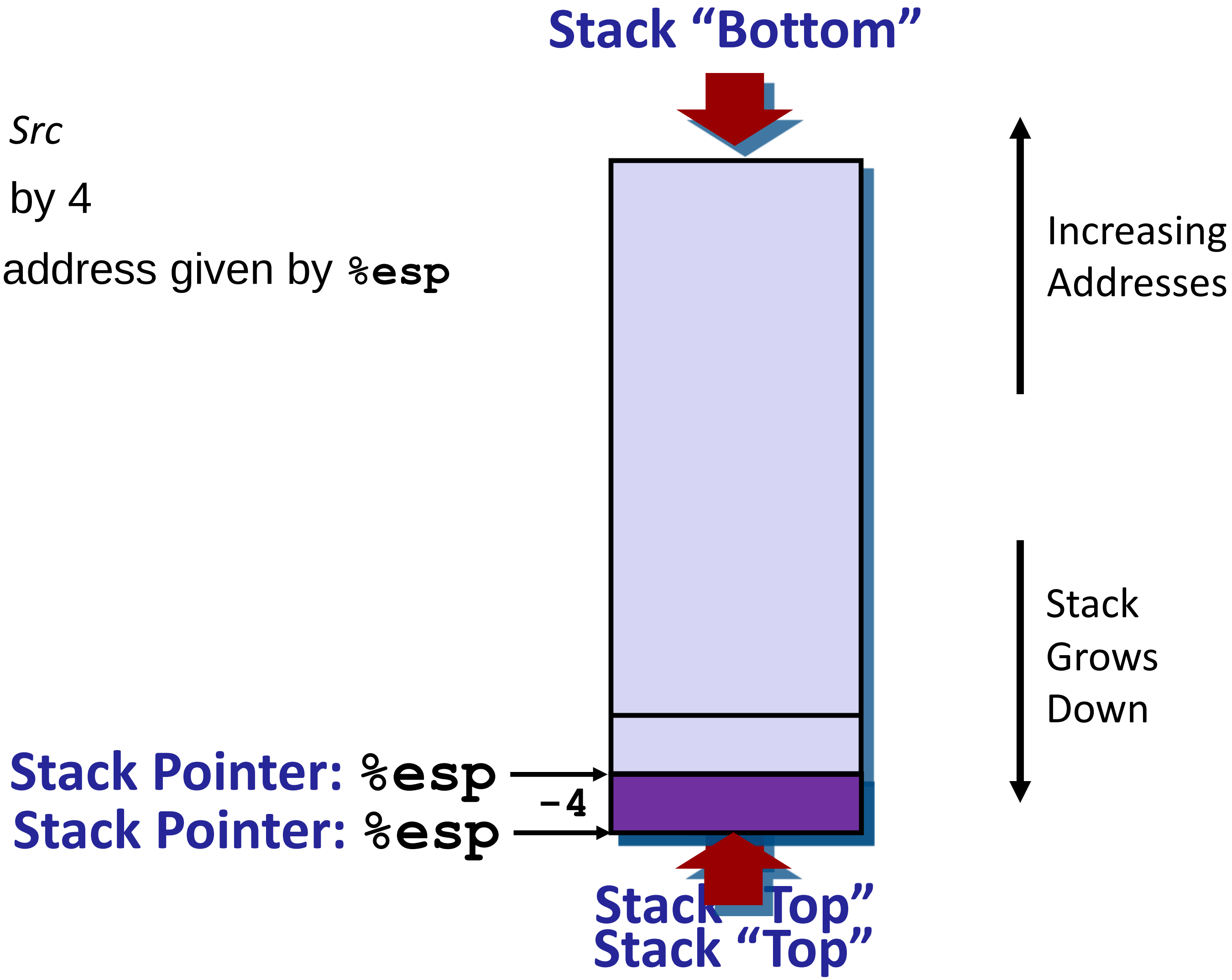
IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` contains lowest stack address
  - address of "top" element



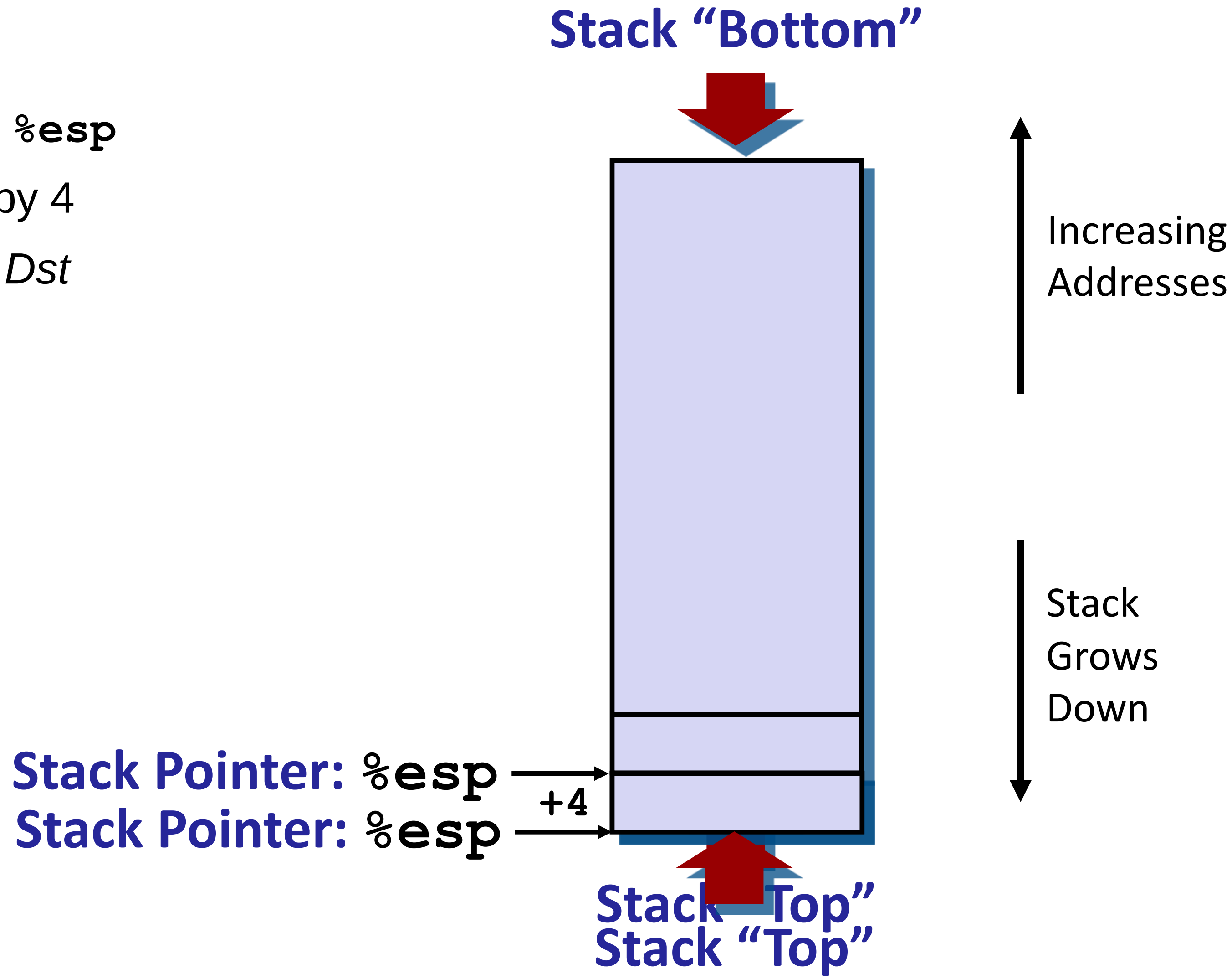
IA32 Stack: Push

- **`pushl`** *Src*
  - Fetch operand at *Src*
  - Decrement **`%esp`** by 4
  - Write operand at address given by **`%esp`**



IA32 Stack: Pop

- `popl` ***Dst***
  - Fetch operand at **`%esp`**
  - Increment **`%esp`** by 4
  - Write operand to *Dst*



Procedure Control Flow

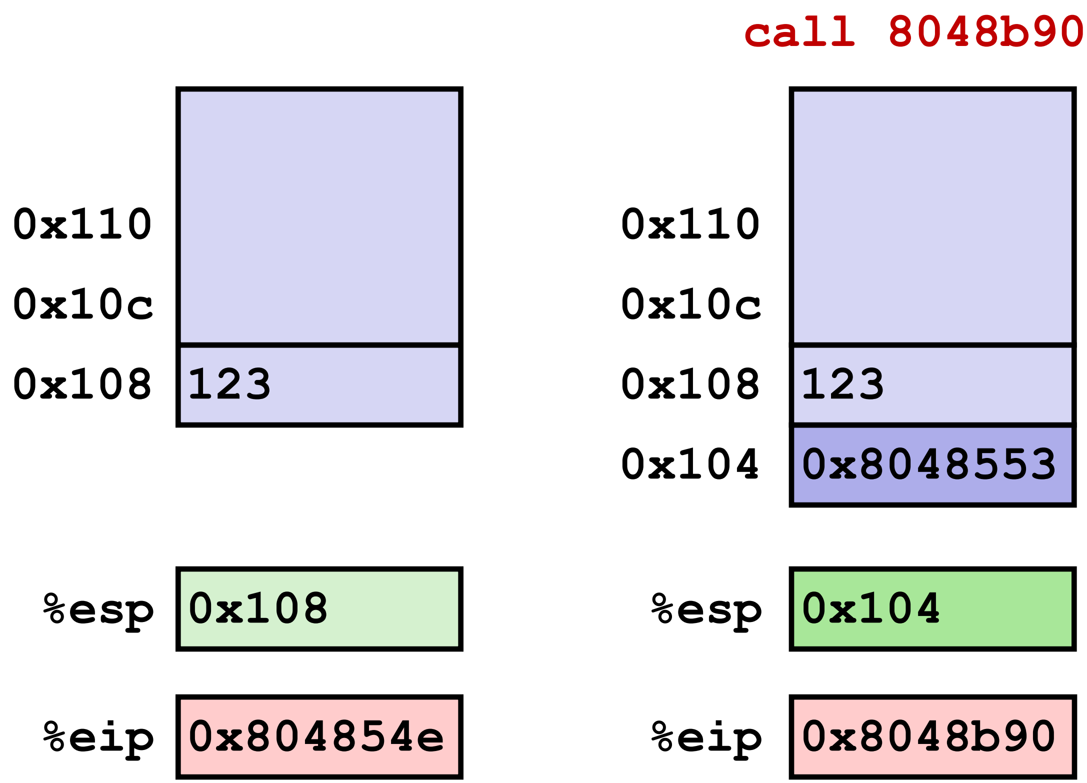
- Use stack to support procedure call and return
- Procedure call: **`call label`**
  - Push return address on stack
  - Jump to ***label***
- Return address:
  - Address of the next instruction right after call
  - Example from disassembly

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

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

Procedure Call Example

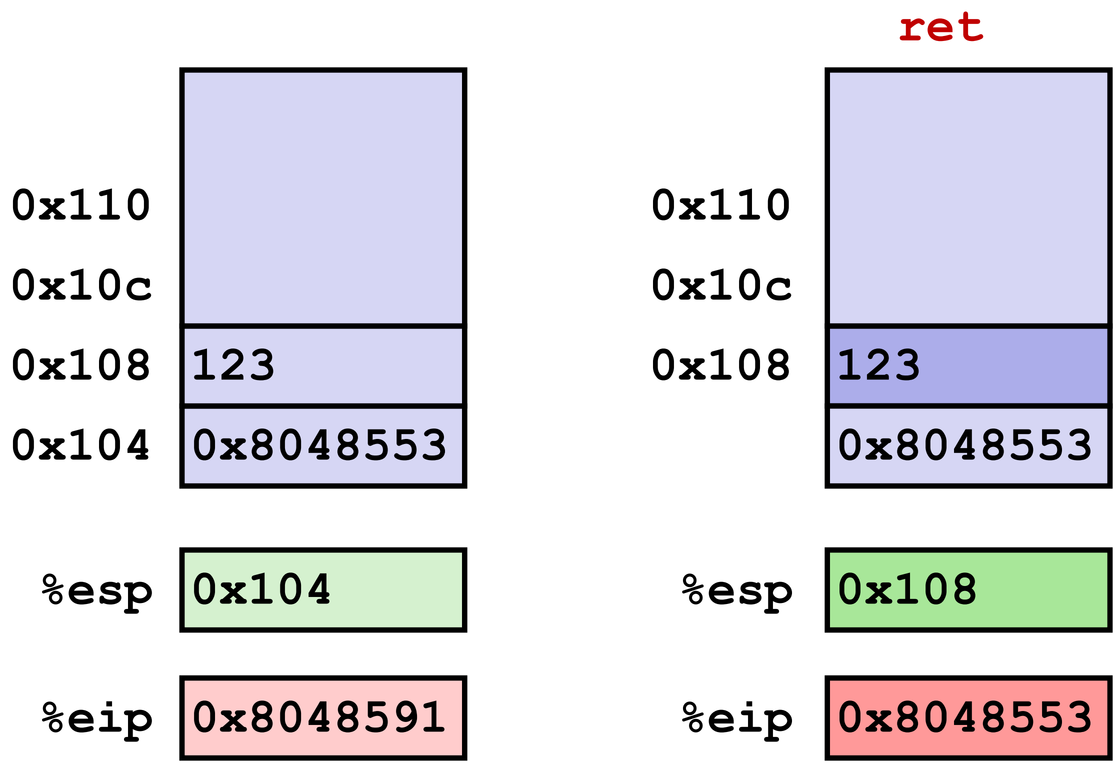
```
804854e:      e8 3d 06 00 00        call    8048b90 <main>
8048553:      50                    pushl   %eax
```



%eip: program counter

Procedure Return Example

```
8048591:     c3                    ret
```



`%eip:` ***program counter***

Stack-Based Languages

- Languages that support recursion
  - e.g., C, Pascal, Java
  - Code must be *"reentrant"*
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer

- Stack discipline
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does

- Stack allocated in ***frames***
  - state for single procedure instantiation

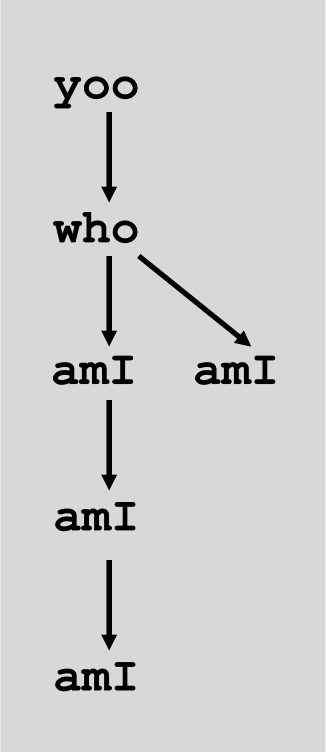
Call Chain Example

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

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

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

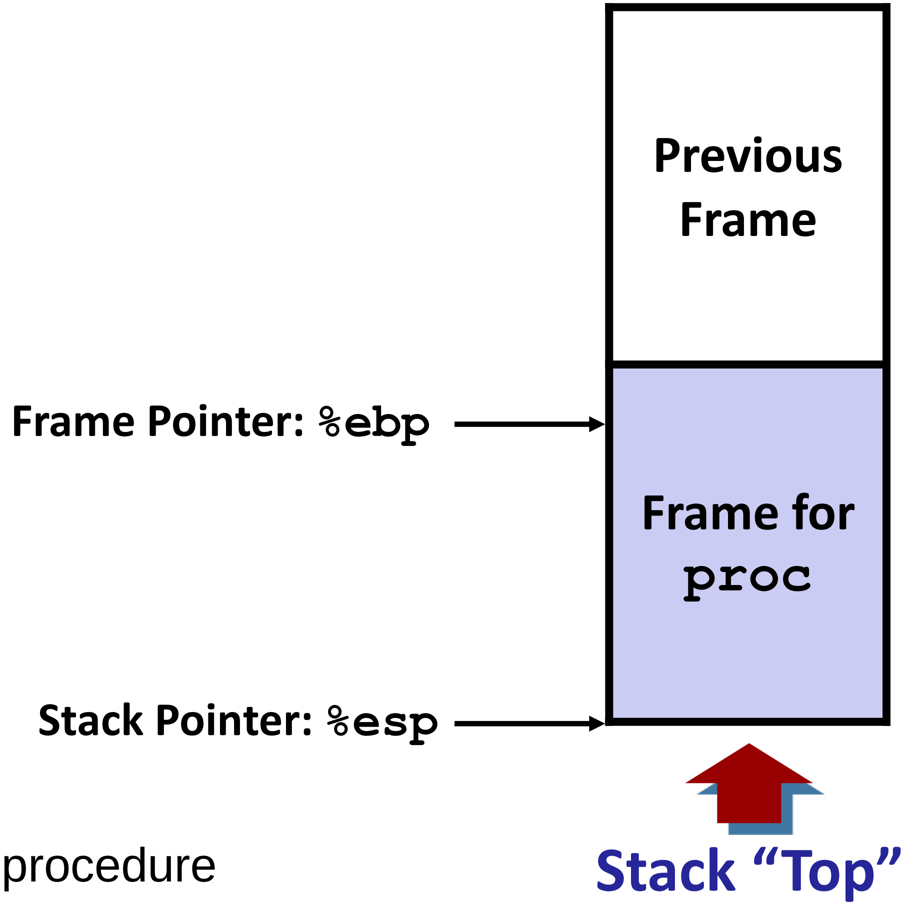
Example Call Chain



Procedure **`amI()`** is recursive

Stack Frames

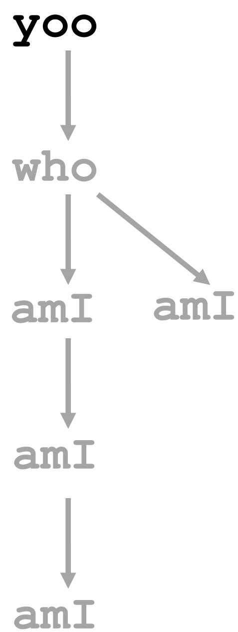
- Contents
  - Local variables
  - Return information
  - Temporary space

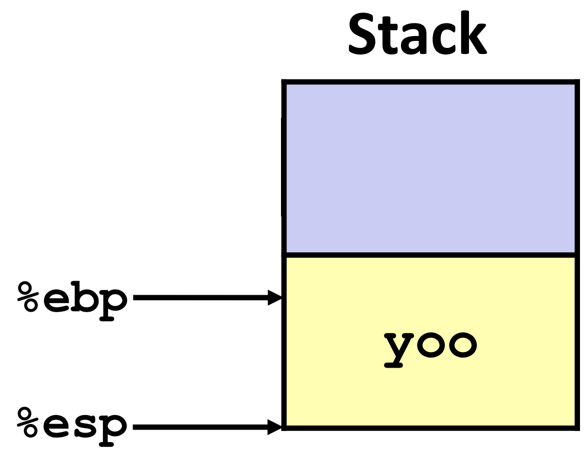


- Management
  - Space allocated when entering a procedure
    - "Set-up" code
  - Deallocated when returning to the caller
    - "Cleanup" code

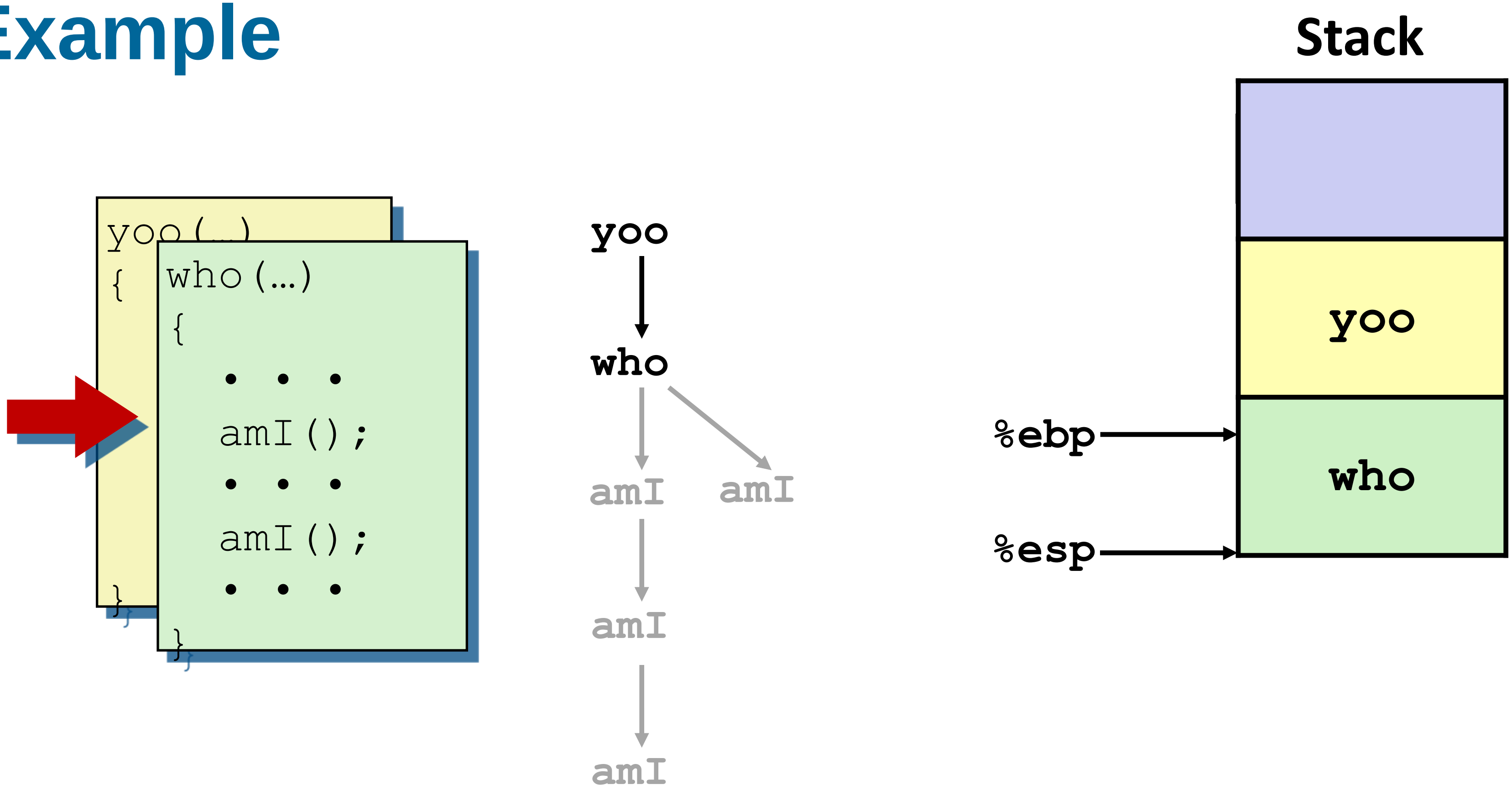
Example

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

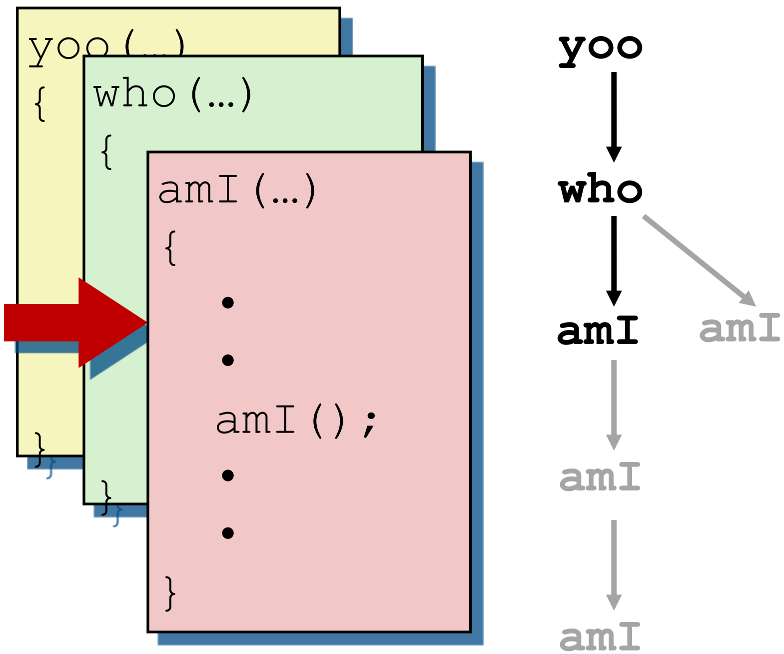




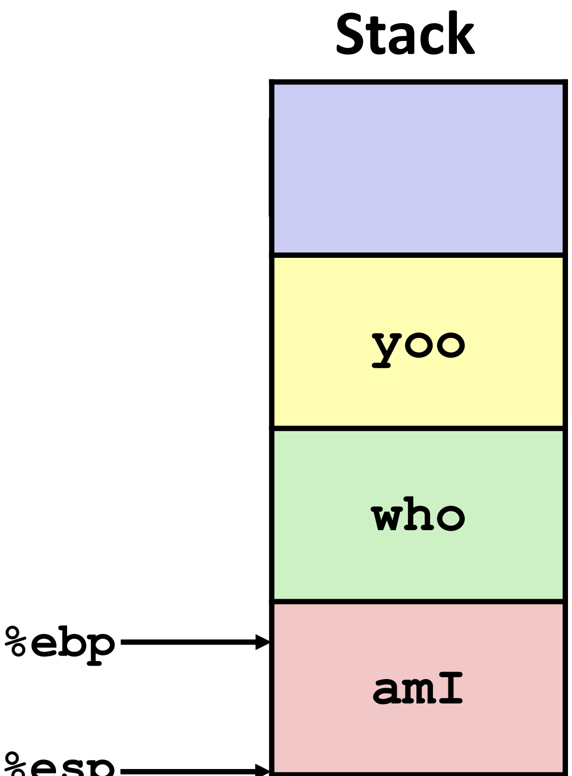
Example



Example

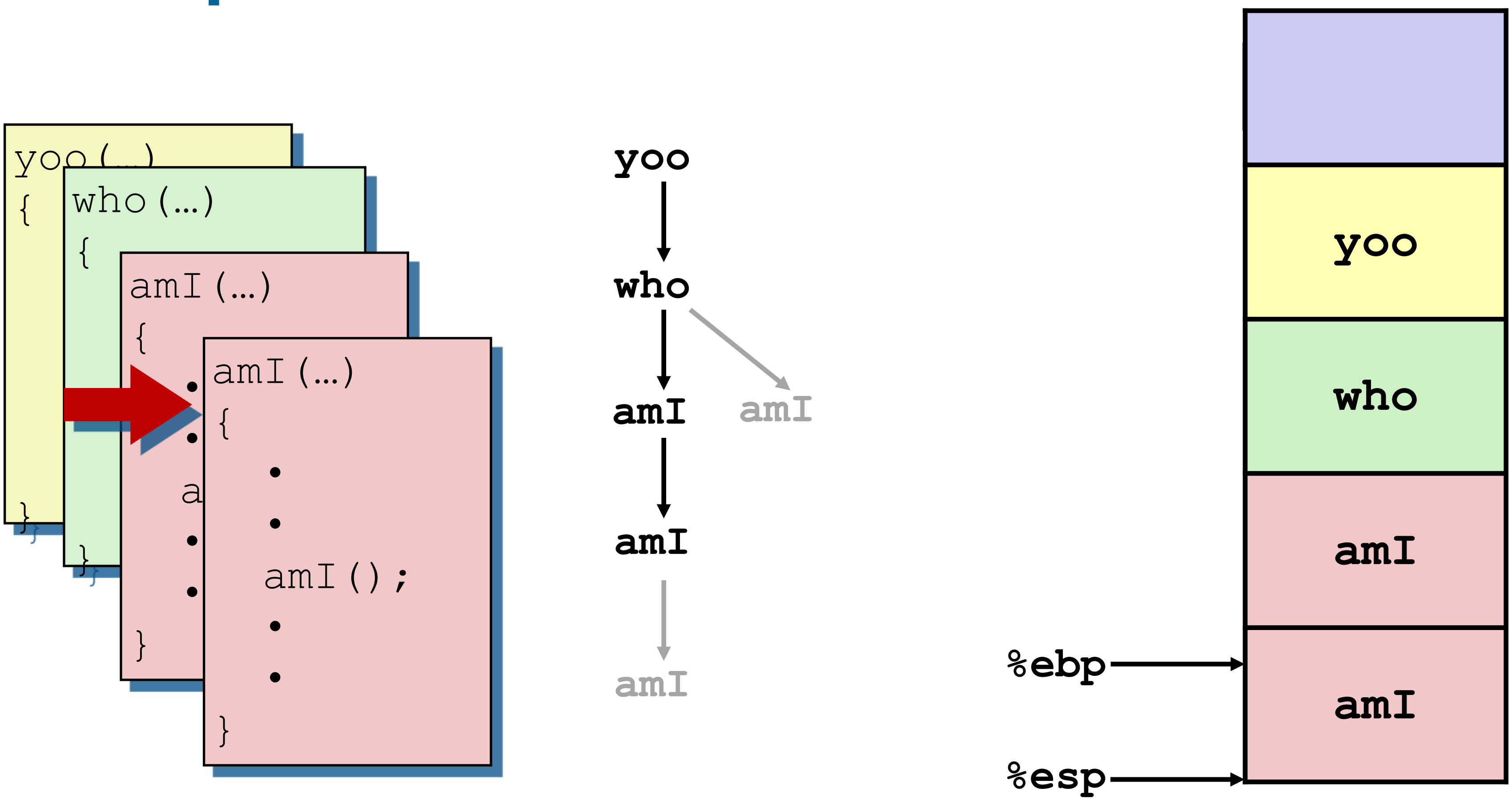


yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}
yoo
who
amI
amI
amI
amI



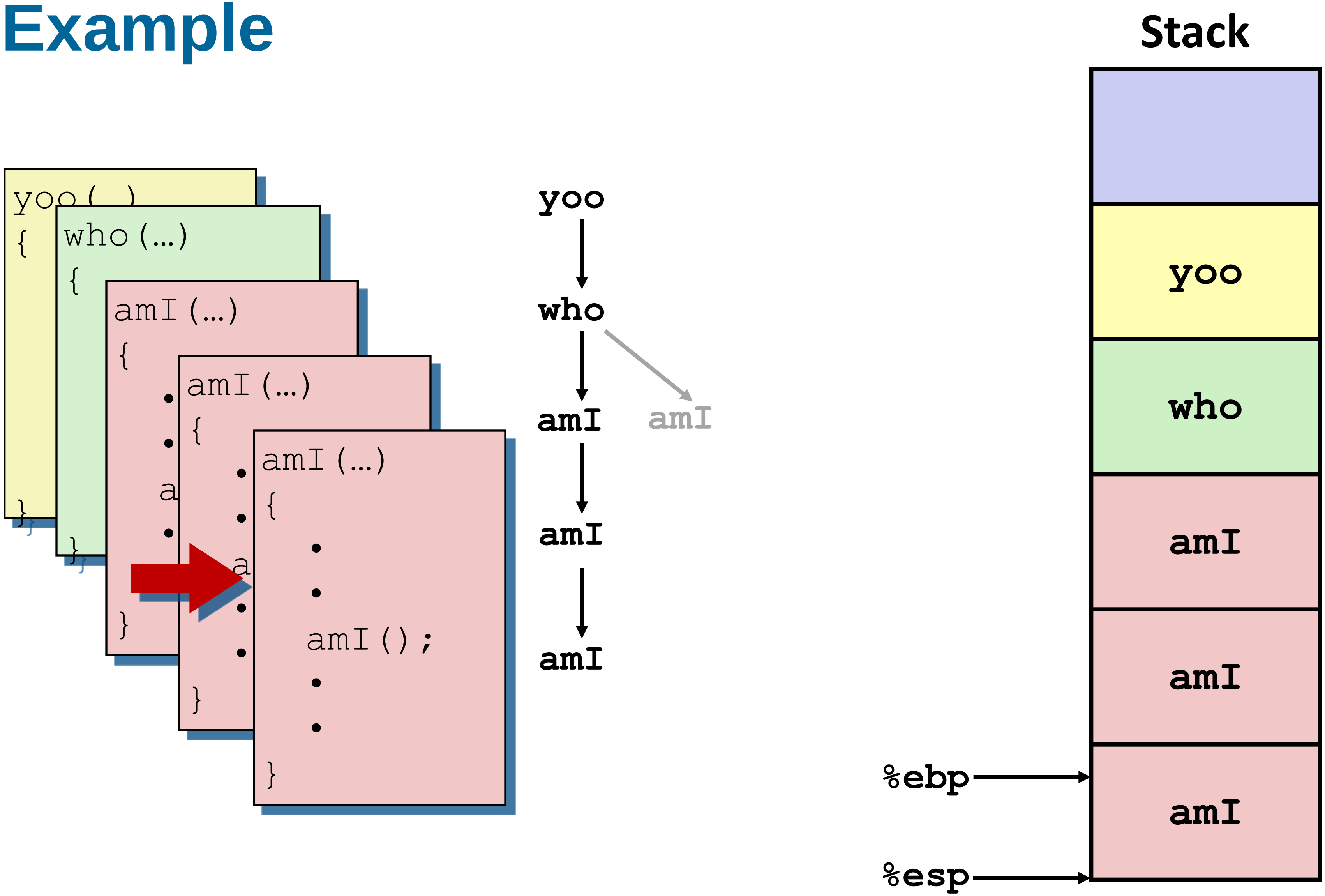
Stack
yoo
who
amI
%ebp
%esp

Example



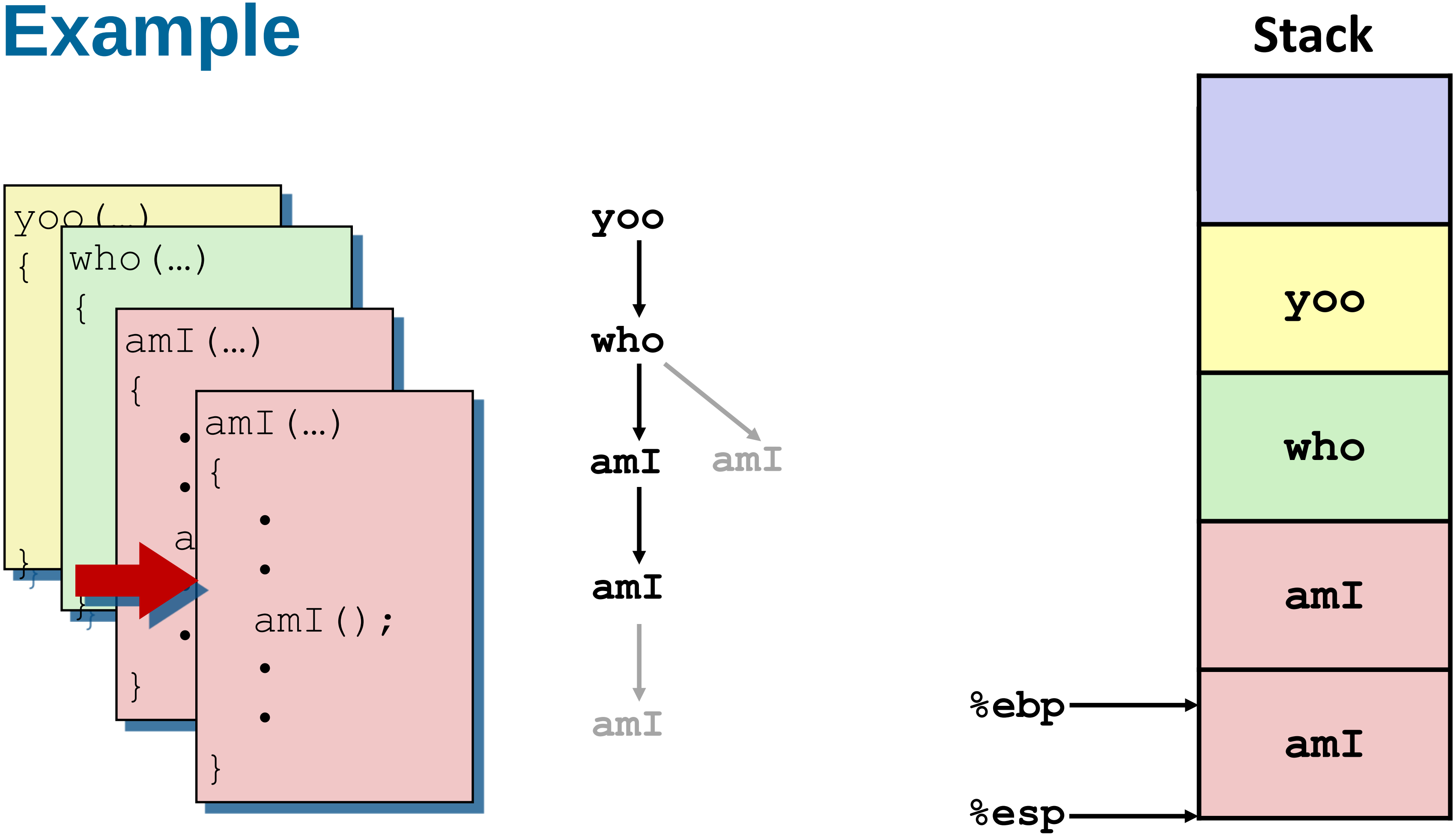
yoo (…)
{
}
who (…)
{
}
amI (…)
{
a
}
amI (…)
{
amI();
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
amI
%ebp
%esp

Example

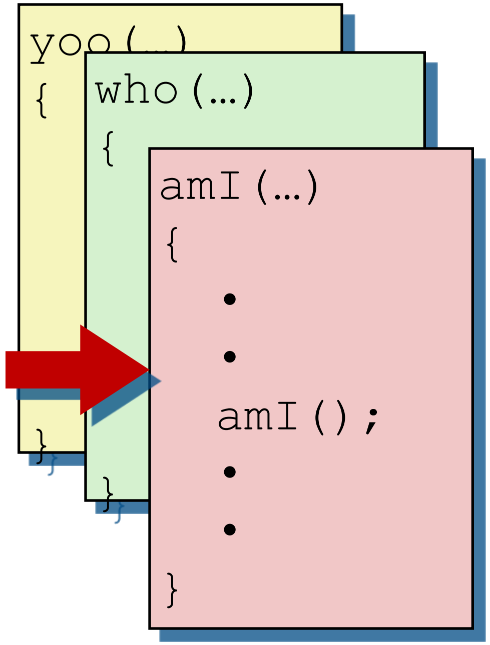


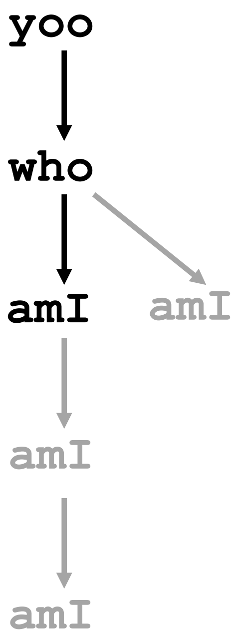
Stack
yoo (…)
{
}
who (…)
{
}
amI (…)
{
}
amI (…)
{
}
amI (…)
{
amI();
}
yoo
who
amI
amI
amI
amI
yoo
who
amI
amI
amI
%ebp
%esp

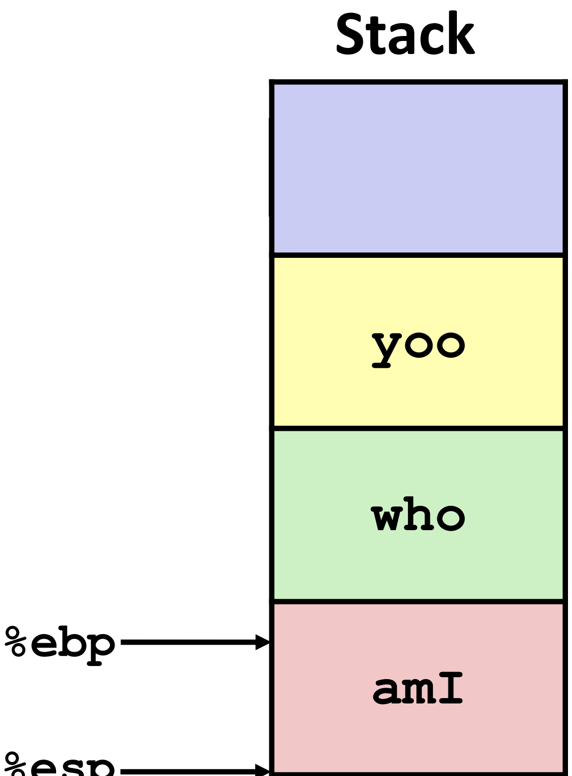
Example


yoo(…)
{
}
who(…)
{
}
amI(…)
{
}
amI(…)
{
amI();
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
amI
%ebp
%esp

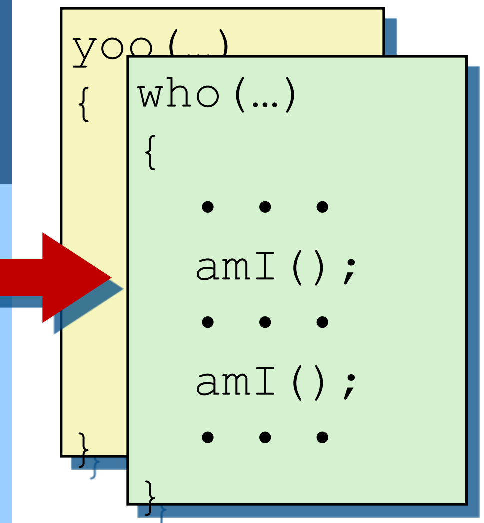
Example

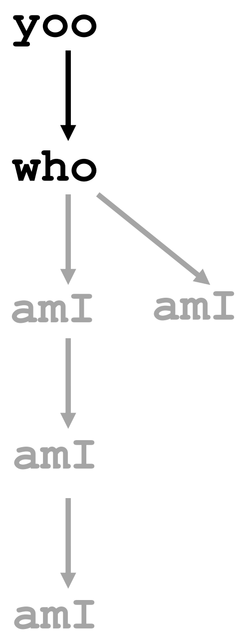

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

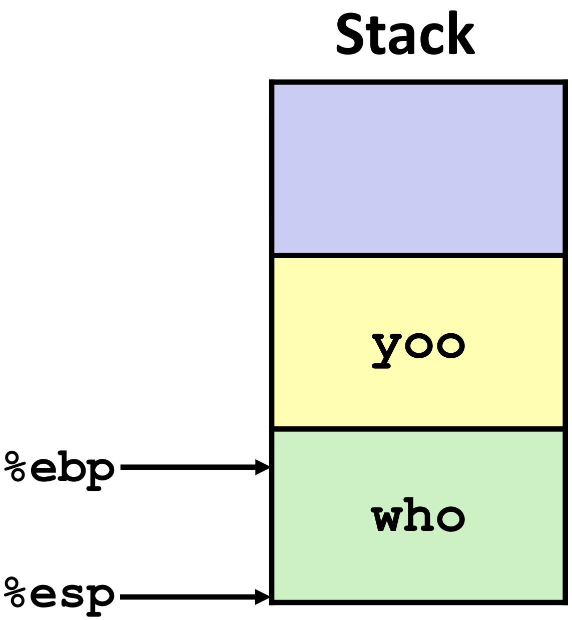

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%ebp
%esp

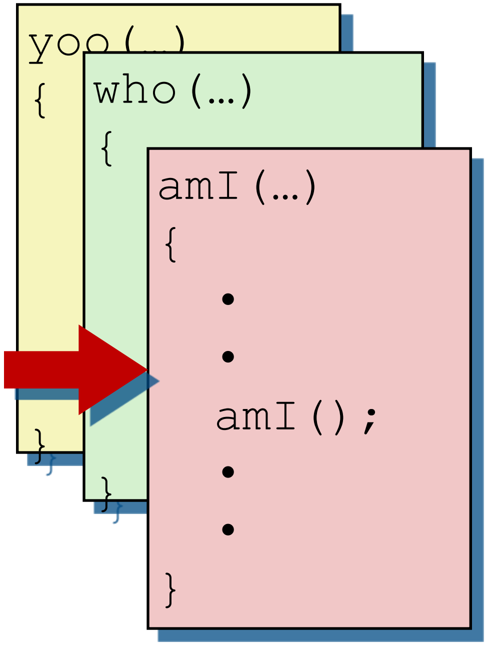
Example



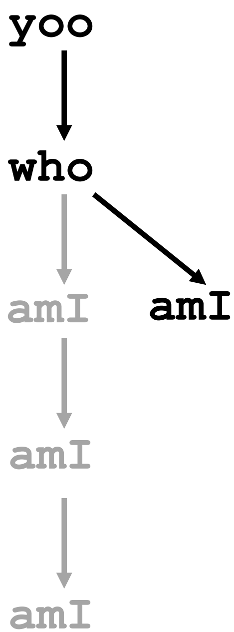




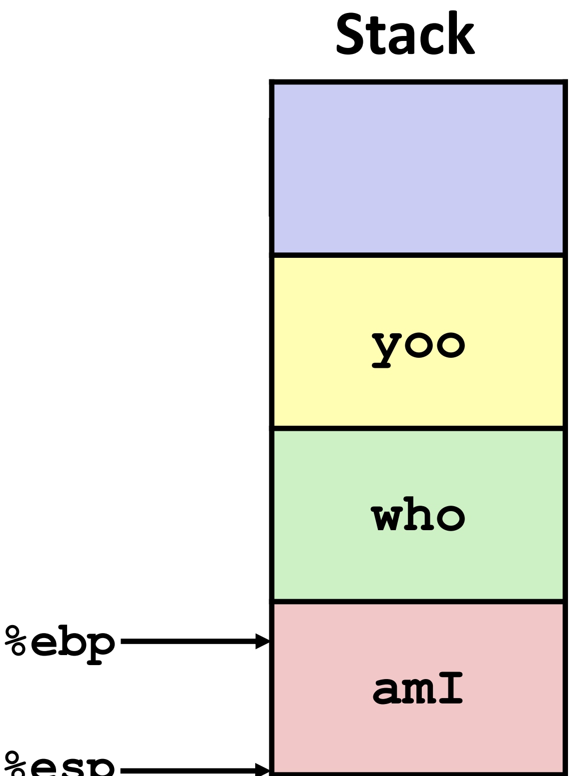
Example



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

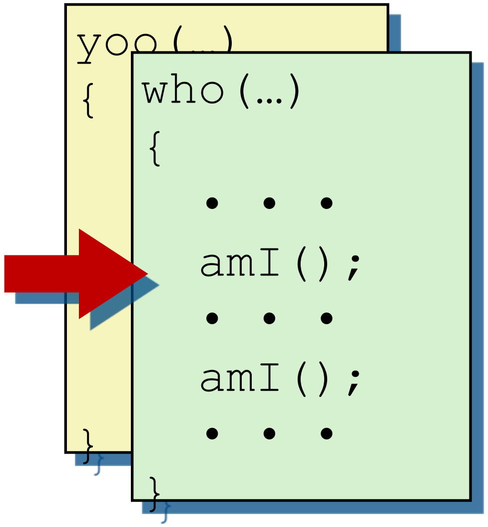


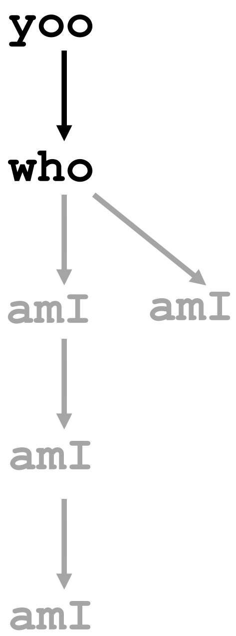
yoo
who
amI
amI
amI
amI

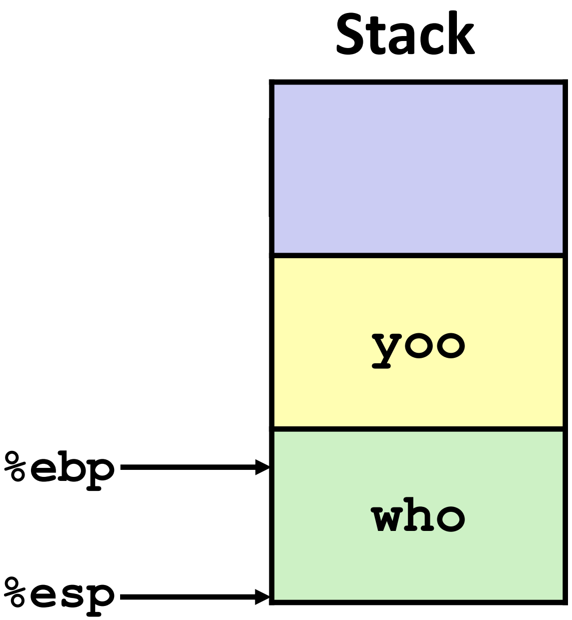


Stack
yoo
who
amI
%ebp
%esp

Example

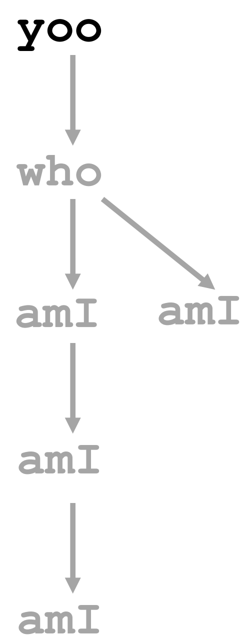


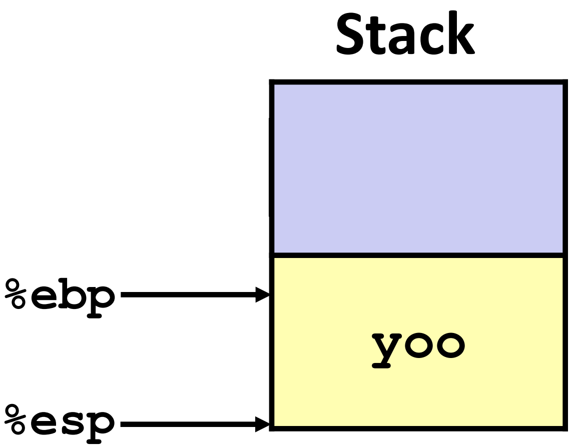




Example

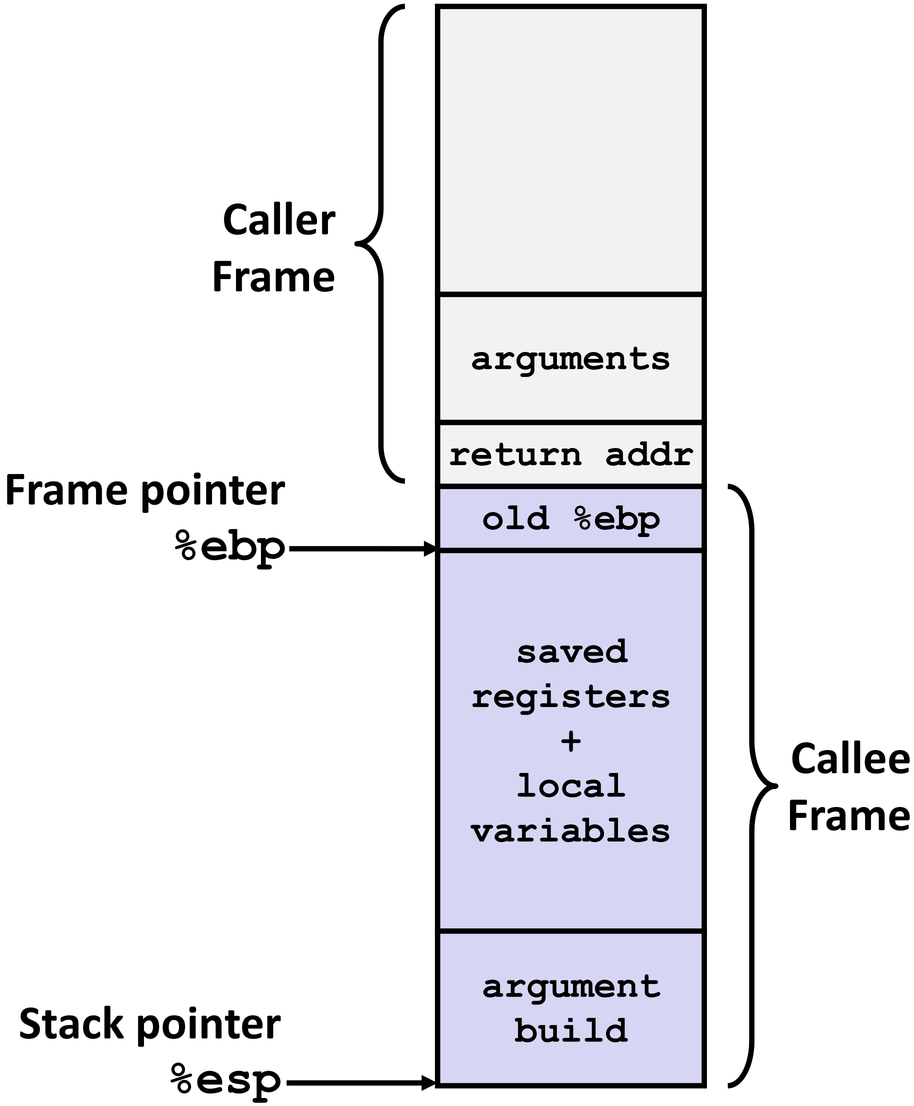
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```





IA32/Linux Stack Frame

- Current Stack Frame (Top to Bottom)
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    If can't keep in registers
  - Saved register context
  - Old frame pointer
- Caller Stack Frame
  - Return address
    - Pushed by **`call`** instruction
  - Arguments for this call



Revisiting `swap`

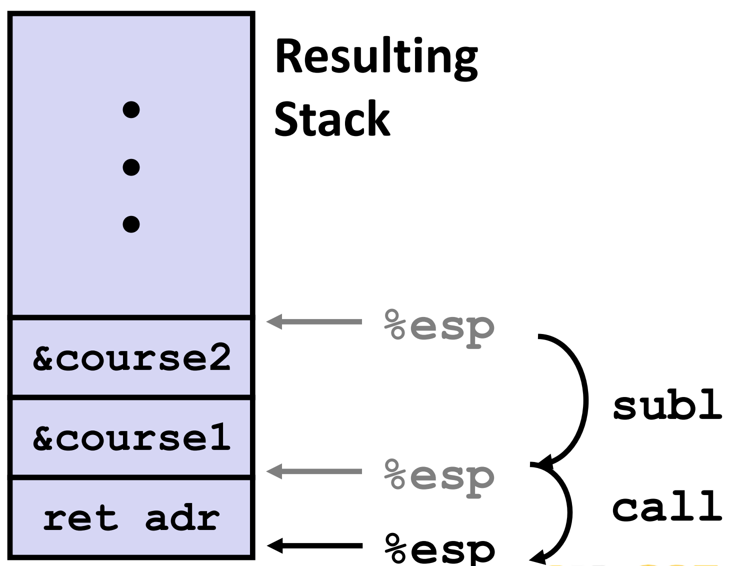
```
int course1 = 15213;
int course2 = 18243;

void call_swap() {
  swap(&course1, &course2);
}
```

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

Calling swap from `call_swap`

```
call_swap:
    • • •
    subl    $8, %esp
    movl    $course2, 4(%esp)
    movl    $course1, (%esp)
    call    swap
    • • •
```

Resulting Stack



Revisiting `swap`

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

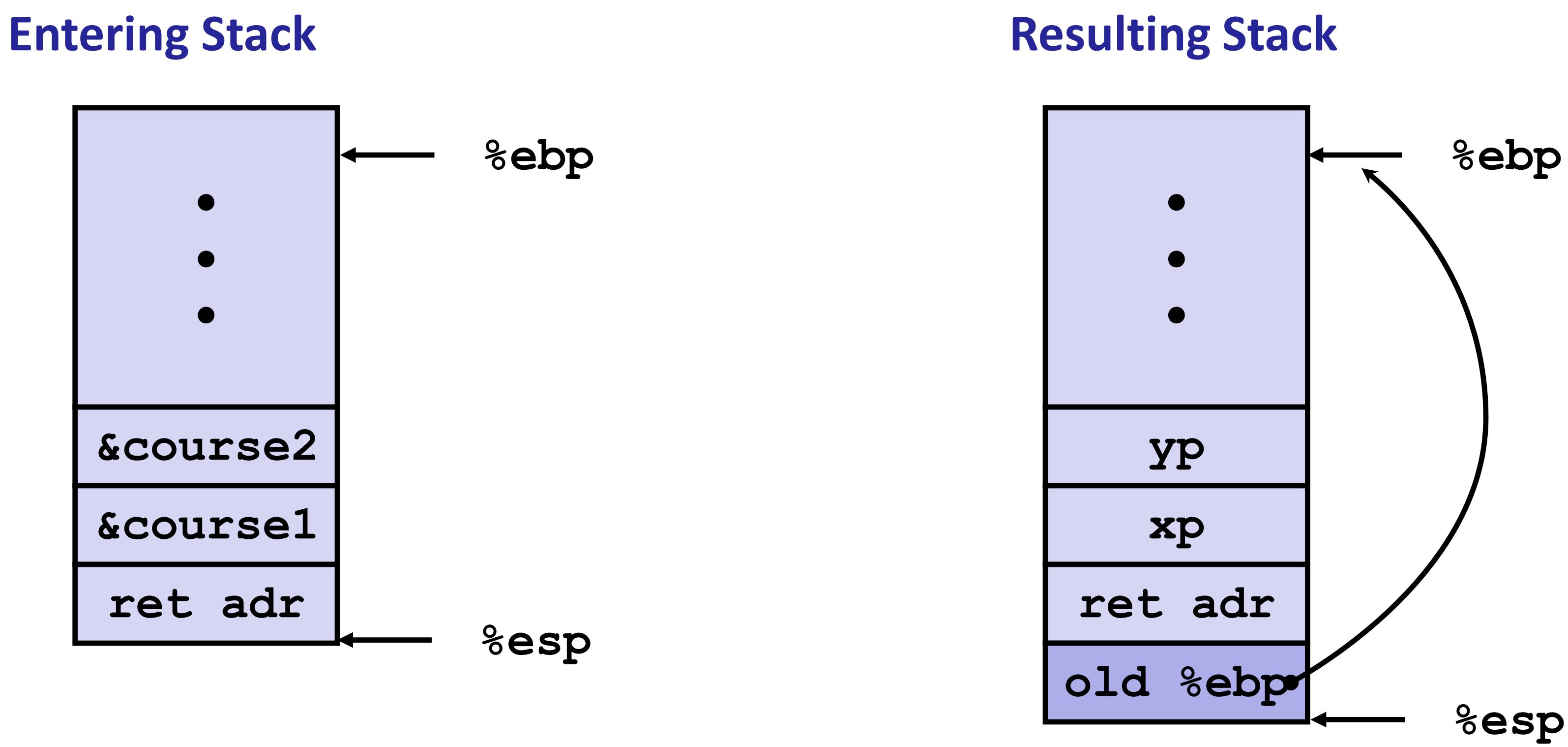
```
swap:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx

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

    popl  %ebx
    popl  %ebp
    ret
```

Set Up

Body

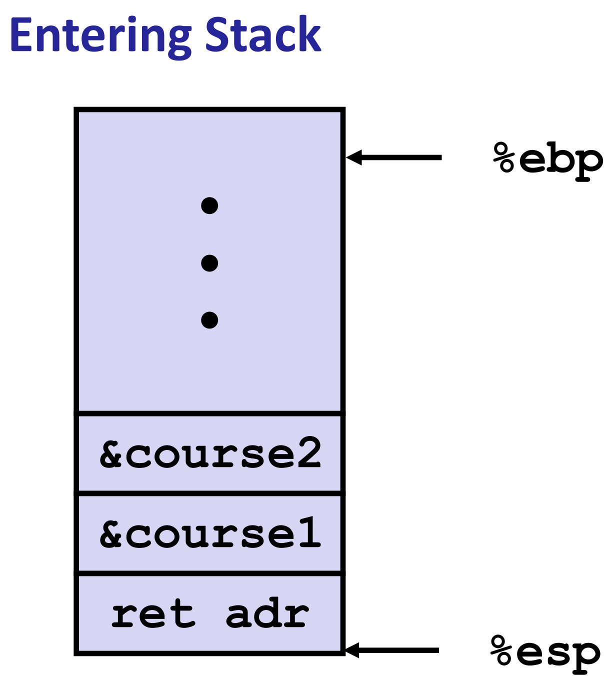
Finish

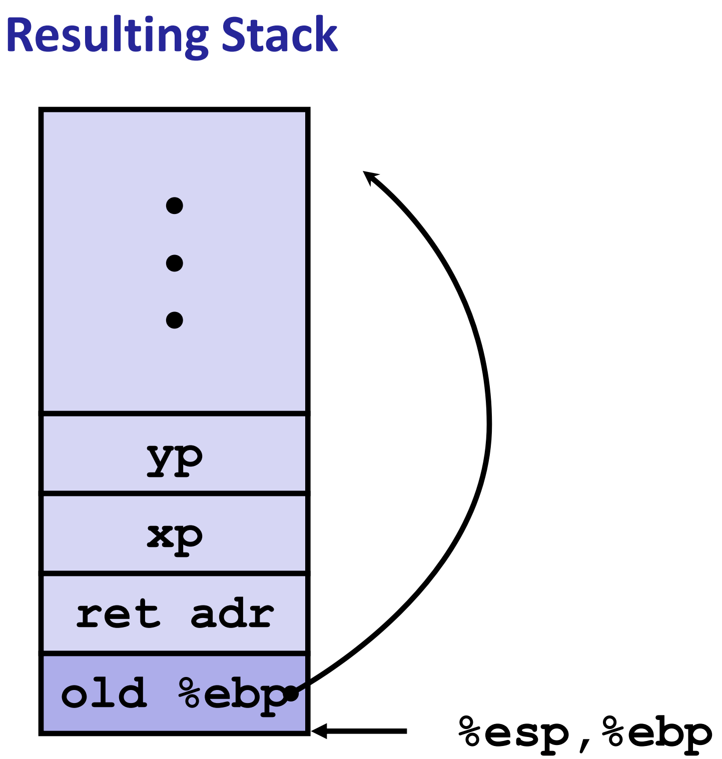
swap Setup #1




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

swap Setup #2



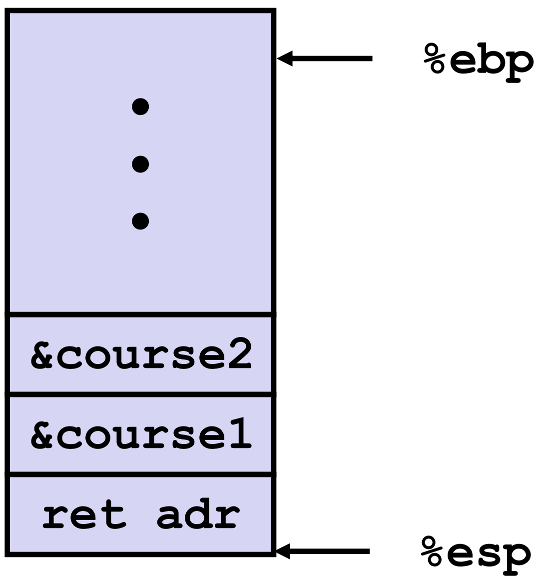



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

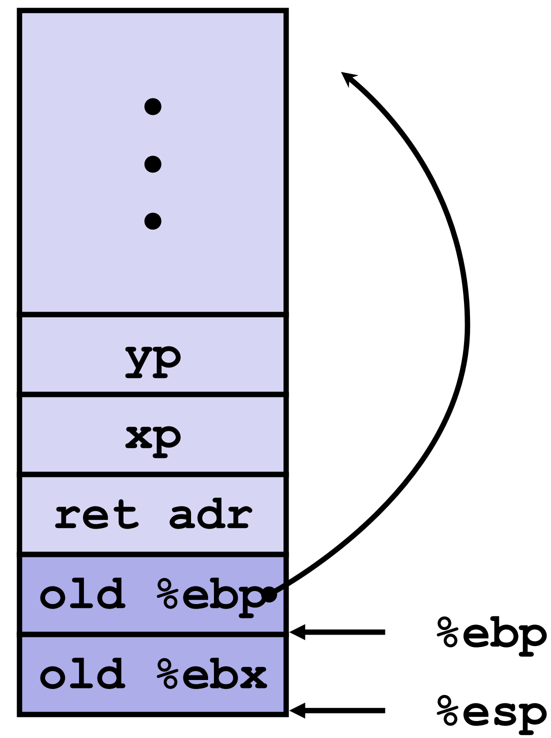

CSE 컴퓨터공학부 Department of Computer Science & Engineering

swap Setup #3

Entering Stack

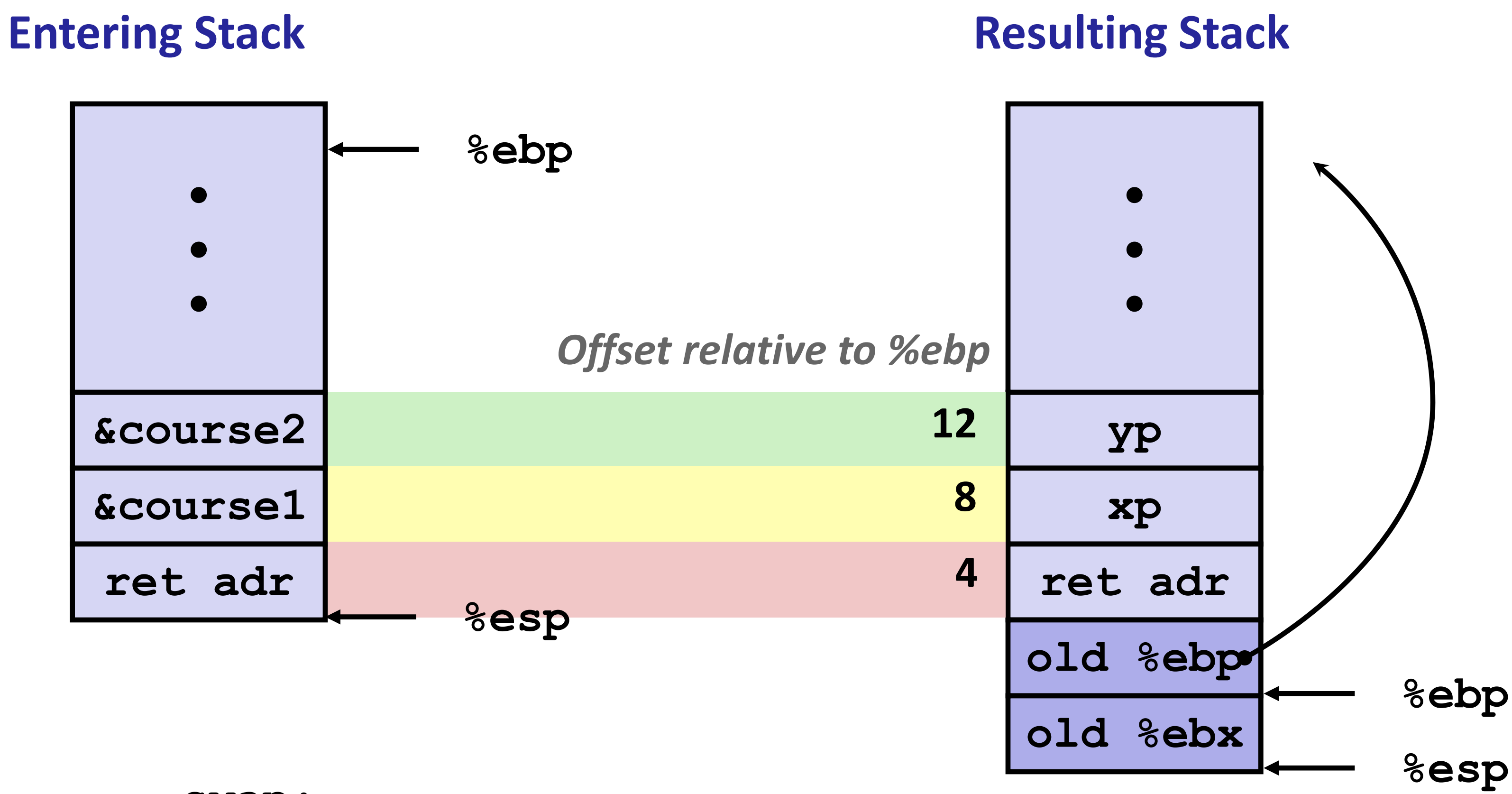


Resulting Stack




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

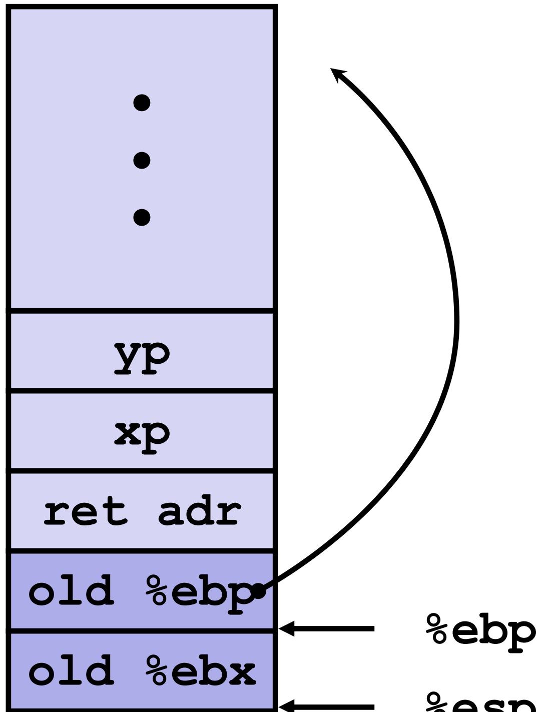
swap Setup #3




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

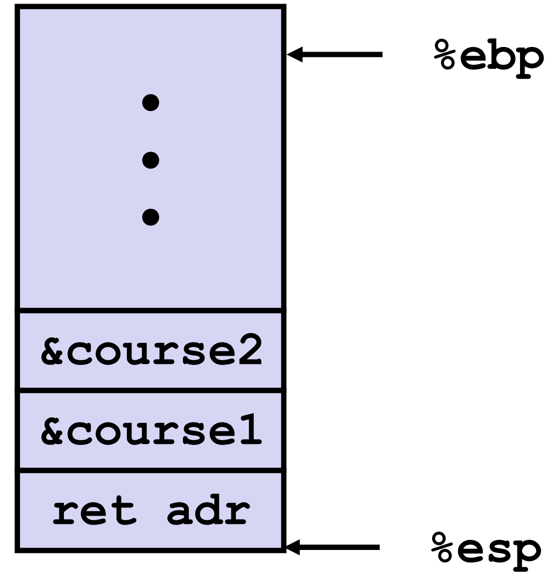
`swap` Cleanup

Stack before Cleanup




```
popl    %ebx
popl    %ebp
```

Resulting Stack



- Observations
  - Saved and restored register **`%ebx, %ebp`**
  - Not so for **`%eax, %ecx, %edx`**
  - Modified **`%esp`**, but value after the call is the same as before the call

Disassembled `swap`

```
08048384 <swap>:
 8048384:   55                          push    %ebp
 8048385:   89 e5                       mov     %esp,%ebp
 8048387:   53                          push    %ebx
 8048388:   8b 55 08                    mov     0x8(%ebp),%edx
 804838b:   8b 4d 0c                    mov     0xc(%ebp),%ecx
 804838e:   8b 1a                       mov     (%edx),%ebx
 8048390:   8b 01                       mov     (%ecx),%eax
 8048392:   89 02                       mov     %eax,(%edx)
 8048394:   89 19                       mov     %ebx,(%ecx)
 8048396:   5b                          pop     %ebx
 8048397:   5d                          pop     %ebp
 8048398:   c3                          ret
```

```
leave :=
movl %ebp, %esp
popl %ebp
```

Calling Code

```
 80483b4:   movl    $0x8049658,0x4(%esp) # Copy &course2
 80483bc:   movl    $0x8049654,(%esp)    # Copy &course1
 80483c3:   call    8048384 <swap>       # Call swap
 80483c8:   leave                        # Prepare to return
 80483c9:   ret                          # Return
```

Procedures

- **IA 32 Procedures**
  - Stack Structure
  - **Calling Conventions**
  - Illustrations of Recursion & Pointers
- x86-64 Procedures

Register Saving Conventions

- When procedure **`yoo`** calls **`who`**:
  - **`yoo`** is the ***caller***
  - **`who`** is the ***callee***

- Can registers be used for temporary storage?

```
yoo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $18243, %edx
    • • •
    ret
```

  - Contents of register **`%edx`** overwritten by **`who`**
  - This could be trouble → something should be done!
    - Need some coordination

Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the ***caller***
  - `who` is the ***callee***

- Can registers be used for temporary storage?

- **Calling Convention**
  - ***“Caller Save”***
    - registers that the callee can overwrite
      (the caller assumes their value is not preserved across procedure calls)
    - Caller saves temporary values in its frame before the call
  - ***“Callee Save”***
    - registers that the callee must preserve before overwriting with a new value
      (the caller can reuse the value across procedure calles)
    - Callee saves temporary values in its frame before using

IA32/Linux+Windows Register Usage

- **%eax**, **%ecx**, **%edx**
  - caller saved prior to call
    (if values are used later)
  - %eax used to return integer value
- **%ebx**, **%esi**, **%edi**
  - callee saved (if used)
- **%esp**, **%ebp**
  - used to manage the stack frames
  - must restore original values upon exit
    from procedure
    (= special form of callee saved)

%eax	Caller saved / Return value
%ecx	Caller saved
%edx	Caller saved
%ebx	Callee saved
%esi	Callee saved
%edi	Callee saved
%esp	Stack pointer
%ebp	Frame pointer

Procedures

- **IA 32 Procedures**
  - Stack Structure
  - Calling Conventions
  - **Illustrations of Recursion & Pointers**
- x86-64 Procedures

Recursive Function

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

- Registers
  - `%eax`, `%edx` used without first saving
  - `%ebx` used, but saved at beginning & restored at end

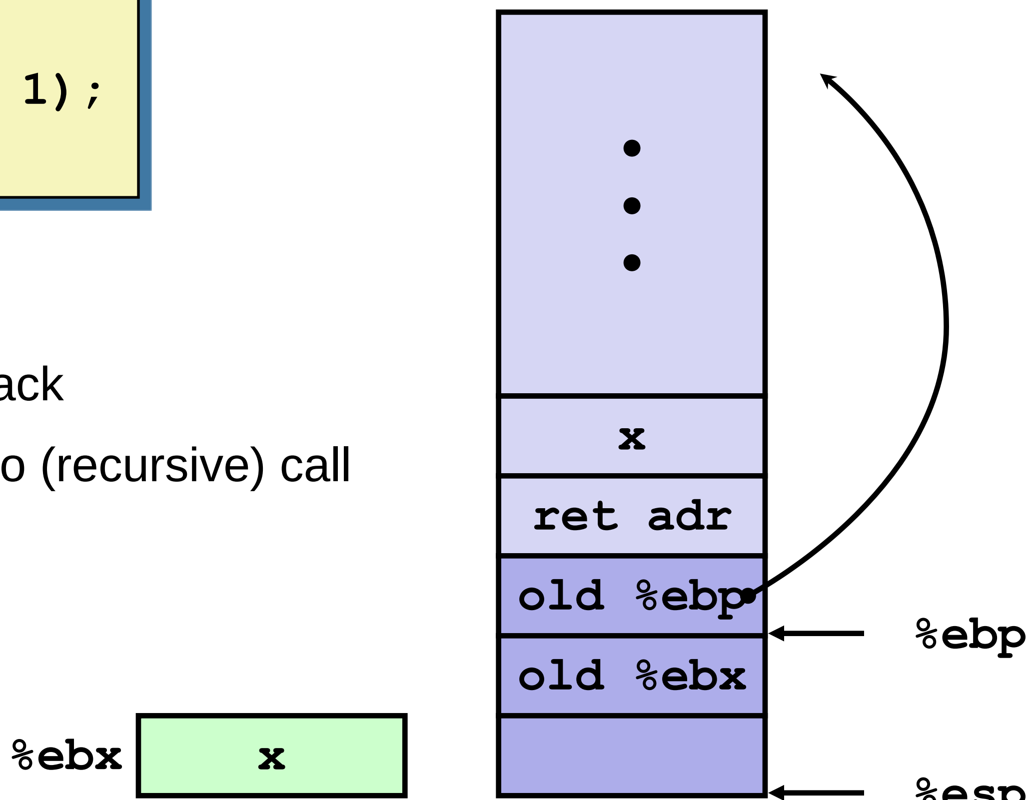
```
pcount_r:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $4, %esp
    movl  8(%ebp), %ebx
    movl  $0, %eax
    testl %ebx, %ebx
    je .L3
    movl  %ebx, %eax
    shrl  %eax
    movl  %eax, (%esp)
    call  pcount_r
    movl  %ebx, %edx
    andl  $1, %edx
    leal  (%edx,%eax), %eax
.L3:
    addl  $4, %esp
    popl  %ebx
    popl  %ebp
    ret
```

Recursive Call #1

```
pcount_r:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $4, %esp
    movl  8(%ebp), %ebx
     • • •
```

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

- Actions
  - Save old value of `%ebx` on stack
  - Allocate space for argument to (recursive) call
  - Store x in `%ebx`



Recursive Call #2

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

```
    • • •
   movl  $0, %eax
   testl %ebx, %ebx
   je .L3
    • • •
.L3:
    • • •
   ret
```

- Actions
  - If x == 0, return
    - with **`%eax`** set to 0

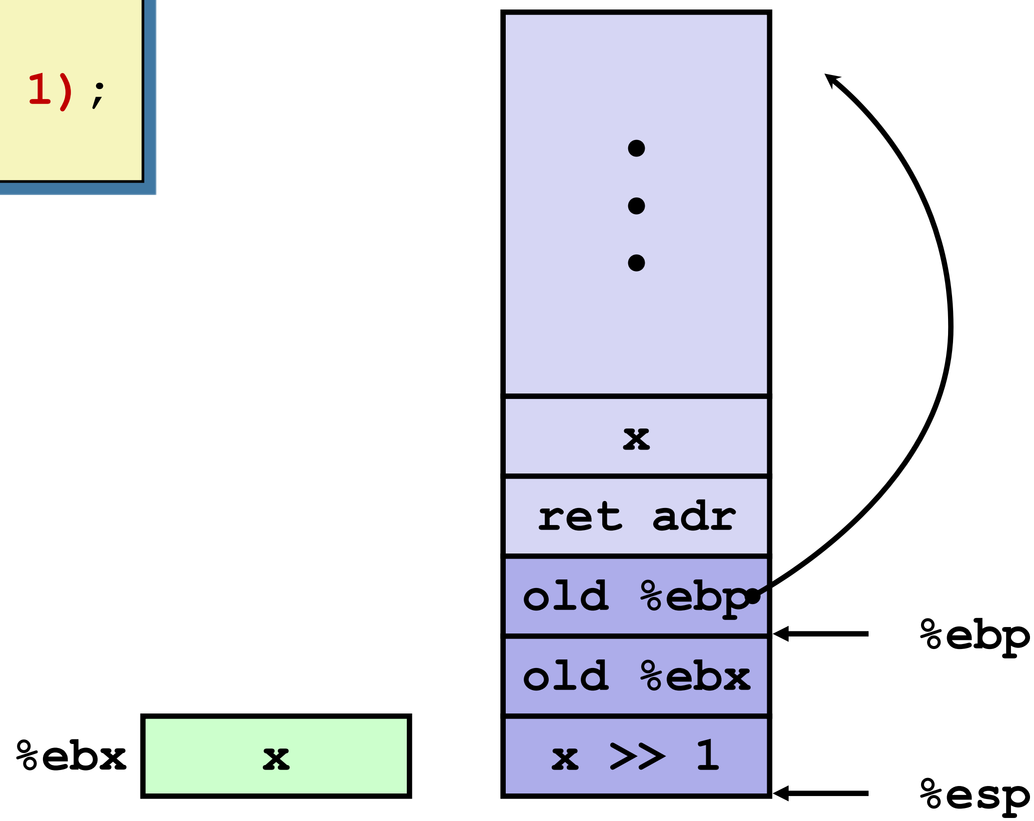
%ebx	x

Recursive Call #3

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

```
 • • •
movl   %ebx, %eax
shrl   %eax
movl   %eax, (%esp)
call   pcount_r
 • • •
```

- Actions
  - Store x >> 1 on stack
  - Make recursive call
- Effect
  - **%eax** set to function result
  - **%ebx** still has value of x



Recursive Call #4

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

```
  • • •
movl   %ebx, %edx
andl   $1, %edx
leal   (%edx,%eax), %eax
  • • •
```

- Assume
  - **%eax** holds value from recursive call
  - **%ebx** holds x
- Actions
  - Compute (x & 1) + computed value
- Effect
  - **%eax** set to function result

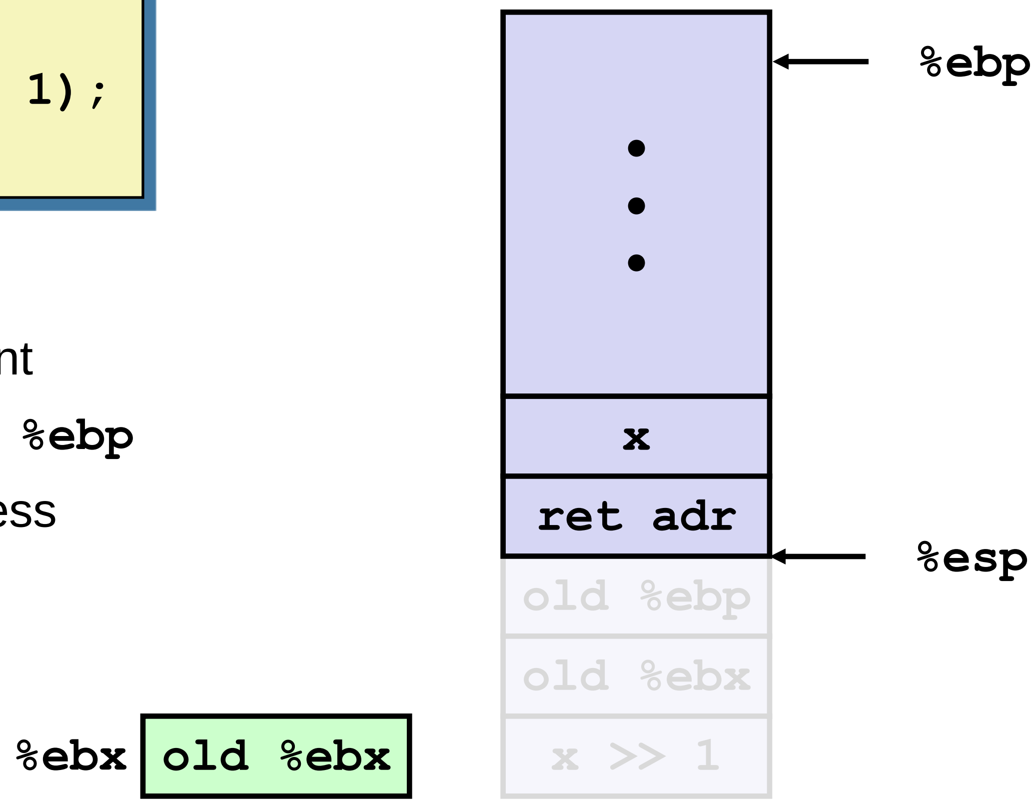
%ebx	x

Recursive Call #5

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

```
      . . .
L3:
      addl   $4, %esp
      popl   %ebx
      popl   %ebp
      ret
```

- Actions
  - Deallocate space for argument
  - Restore values of `%ebx` and `%ebp`
  - `ret` will pop the return address into `%eip`



Observations About Recursion

- Handled Without Special Consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
  - 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

Pointer Code

- **`add3`** creates pointer and passes it to **`incrk`**

Generating a Pointer

```
/* Compute x + 3 */
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

Referencing a Pointer

```
/* Increment value by k */
void incrk(int *ip, int k) {
  *ip += k;
}
```

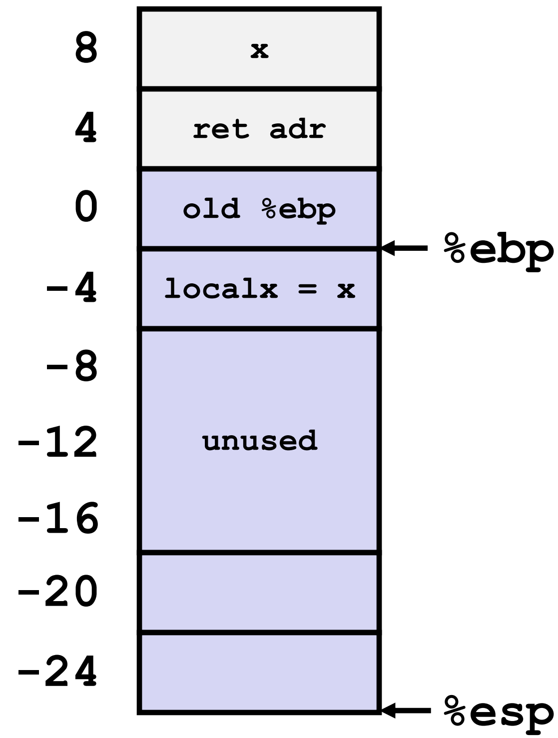
Creating and Initializing Local Variables

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- variable localx must be stored on the stack
  - the compiler needs to create a pointer to it
- compute pointer as -4(%ebp)

First part of add3

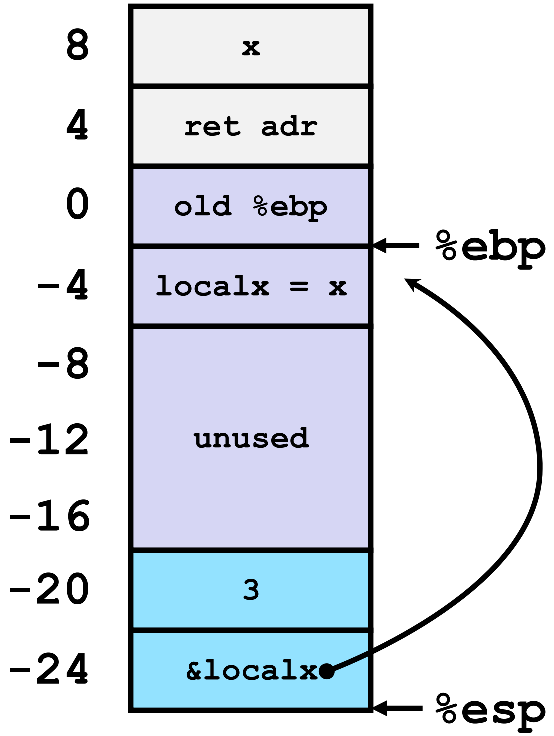
```
add3:
  pushl%ebp
  movl %esp, %ebp
  subl $24, %esp     # Alloc. 24 bytes
  movl 8(%ebp), %eax
  movl %eax, -4(%ebp)# Set localx to x
```



Creating Pointer as Argument

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- Use **`leal`** to compute the address of **`localx`**



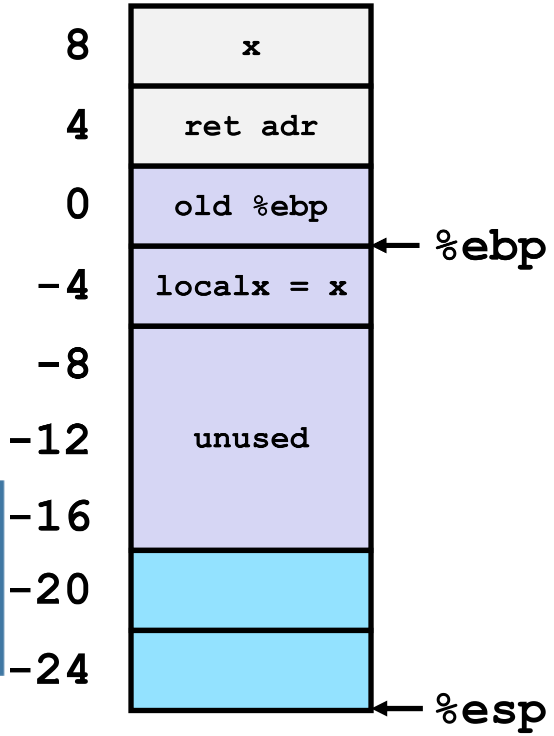
Middle part of add3

```
movl $3, 4(%esp)   # 2nd arg = 3
leal -4(%ebp), %eax# &localx
movl %eax, (%esp)  # 1st arg = &localx
call incrk
```

Retrieving local variable

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- Retrieve **`localx`** from stack as return value

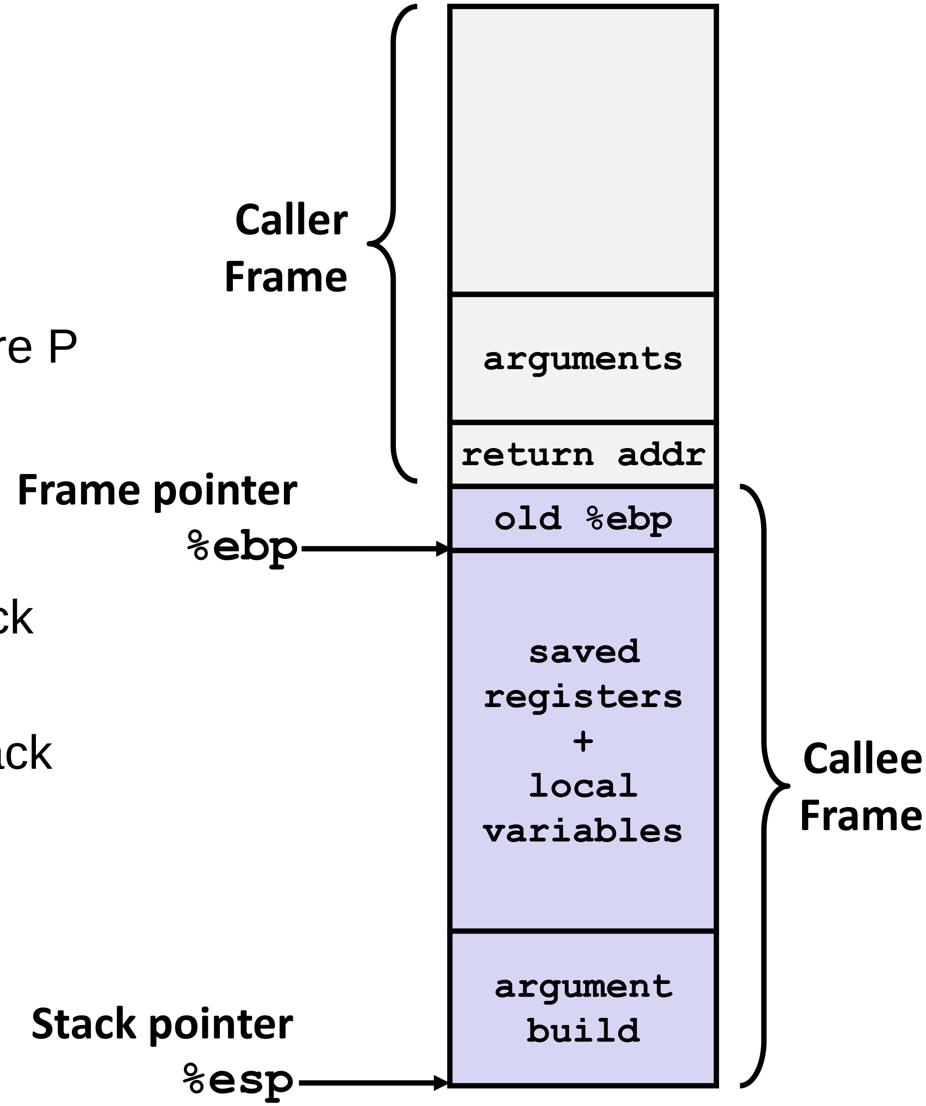


Final part of add3

```
movl -4(%ebp), %eax # Return val= localx
leave
ret
```

IA 32 Procedure Summary

- Important Points
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- Recursion (& mutual 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 return in **`%eax`**
- Pointers are addresses of values
  - On stack or global



Procedures

- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers
- **x86-64 Procedures**

x86-64 Integer Registers: Usage Conventions

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

x86-64 Registers

- Arguments passed to functions via registers
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well

- All references to stack frame via stack pointer
  - Eliminates need to update %ebp/%rbp

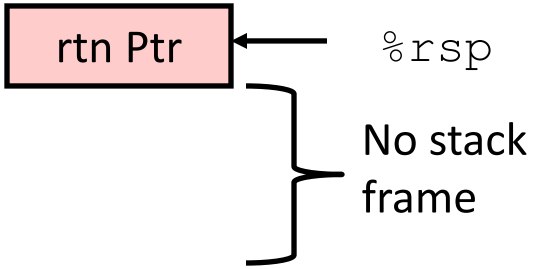
- Other Registers
  - 6 callee saved
  - 2 caller saved
  - 1 return value (also usable as caller saved)
  - 1 special (stack pointer)

x86-64 Long Swap

```
void swap_l(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq    (%rdi), %rdx
   movq    (%rsi), %rax
   movq    %rax, (%rdi)
   movq    %rdx, (%rsi)
   ret
```

- Operands 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

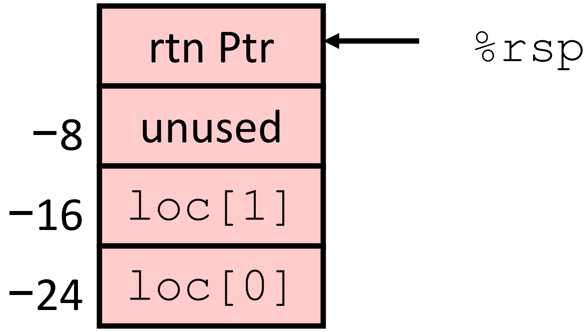


x86-64 Locals in the Red Zone

```
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
   movq   (%rdi), %rax
   movq   %rax, -24(%rsp)
   movq   (%rsi), %rax
   movq   %rax, -16(%rsp)
   movq   -16(%rsp), %rax
   movq   %rax, (%rdi)
   movq   -24(%rsp), %rax
   movq   %rax, (%rsi)
   ret
```

- Avoiding Stack Pointer Change
  - Can hold all information within small window beyond stack pointer



x86-64 NonLeaf without a Stack Frame

```
/* Swap a[i] & a[i+1] */
void swap_ele(long a[], int i)
{
    swap(&a[i], &a[i+1]);
}
```

- No values held while swap being invoked
- No callee save registers needed
- rep instruction inserted as no-op (recommendation from AMD)

```
swap_ele:
   movslq %esi,%rsi               # Sign extend i
   leaq   8(%rdi,%rsi,8), %rax    # &a[i+1]
   leaq   (%rdi,%rsi,8), %rdi     # &a[i] (1st arg)
   movq   %rax, %rsi              # (2nd arg)
   call   swap
   rep                            # No-op
   ret
```

x86-64 Stack Frame Example

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
      (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

- Keeps values of &a[i] and &a[i+1] in callee save registers
- Must set up stack frame to save these registers

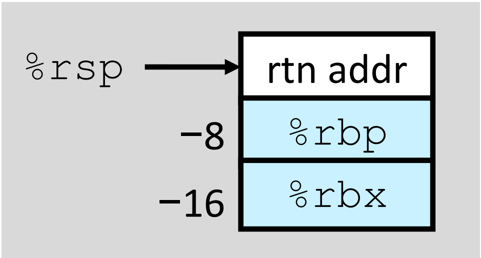
```
swap_ele_su:
   movq    %rbx, -16(%rsp)
   movq    %rbp, -8(%rsp)
   subq    $16, %rsp
   movslq  %esi,%rax
   leaq    8(%rdi,%rax,8), %rbx
   leaq    (%rdi,%rax,8), %rbp
   movq    %rbx, %rsi
   movq    %rbp, %rdi
   call    swap
   movq    (%rbx), %rax
   imulq   (%rbp), %rax
   addq    %rax, sum(%rip)
   movq    (%rsp), %rbx
   movq    8(%rsp), %rbp
   addq    $16, %rsp
   ret
```

Understanding x86-64 Stack Frame

```
swap_ele_su:
    movq    %rbx, -16(%rsp)          # Save %rbx
    movq    %rbp, -8(%rsp)           # Save %rbp
    subq    $16, %rsp                # Allocate stack frame
    movslq  %esi,%rax                # Extend i
    leaq    8(%rdi,%rax,8), %rbx     # &a[i+1] (callee save)
    leaq    (%rdi,%rax,8), %rbp      # &a[i]   (callee save)
    movq    %rbx, %rsi               # 2nd argument
    movq    %rbp, %rdi               # 1st argument
    call    swap
    movq    (%rbx), %rax             # Get a[i+1]
    imulq   (%rbp), %rax             # Multiply by a[i]
    addq    %rax, sum(%rip)          # Add to sum
    movq    (%rsp), %rbx             # Restore %rbx
    movq    8(%rsp), %rbp            # Restore %rbp
    addq    $16, %rsp                # Deallocate frame
    ret
```

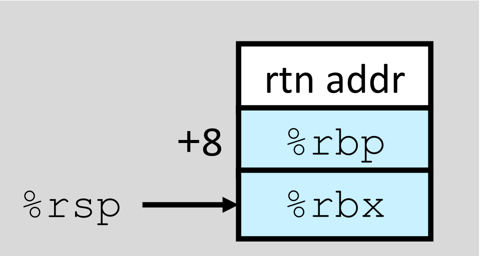
Understanding x86-64 Stack Frame

```
movq    %rbx, -16(%rsp)           # Save %rbx
movq    %rbp, -8(%rsp)            # Save %rbp
```




```
subq    $16, %rsp                 # Allocate stack frame
```

• • •




```
movq    (%rsp), %rbx              # Restore %rbx
movq    8(%rsp), %rbp             # Restore %rbp

addq    $16, %rsp                 # Deallocate frame
```

Interesting Features of Stack Frames

- Allocate entire frame at once
  - All stack accesses can be relative to %rsp
  - Do by decrementing stack pointer
  - Can delay allocation, since safe to temporarily use red zone

- Simple deallocation
  - Increment stack pointer
  - No base/frame pointer needed

x86-64 Procedure Summary

- Heavy use of registers
  - Parameter passing
  - More temporaries since more registers
- Minimal use of stack
  - Sometimes none
  - Allocate/deallocate entire block
- Many tricky optimizations
  - What kind of stack frame to use
  - Various allocation techniques