CSE 351: Week 4

Tom Bergan, TA

Does this code look okay?

```
int binarySearch(int a[], int length, int key) {
    int low = 0;
    int high = length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid;  // key found
    }
    return -1;  // key not found
}
```

Does this code look okay?

```
int binarySearch(int a[], int length, int key) {
    int low = 0;
    int high = length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low =
        else if (midVal > key)
            high = mid - 1;
        else
            return mid;   // key found
    }
    return -1;   // key not found
}
```

What if length > 2^{30}?

Does this code look ok?

```
int mid = (low + high) / 2;
```

What if length > 2^{30}?

... then we could have: **low = 2^{30} = 0x40000000**
high = 2^{30}+1 = 0x40000001
low + high = 2^{31}+1 = 0x80000001

Oops, in two's complement, this is a negative number!

(low + high) / 2 = 0xC0000000
= -3221225472

```
int midVal = a[mid];
```

Crashes because mid < 0

How can we fix the bug?

```
int mid = (low + high) / 2;
```

↓

```
int mid = low + ((high - low) / 2);
```

(There are other ways, but I think this is the simplest to understand)

This was an actual bug in Java

`java.util.Arrays.binarySearch`

This bug went unnoticed for years.

See: http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html

Understanding binary number representations is important!

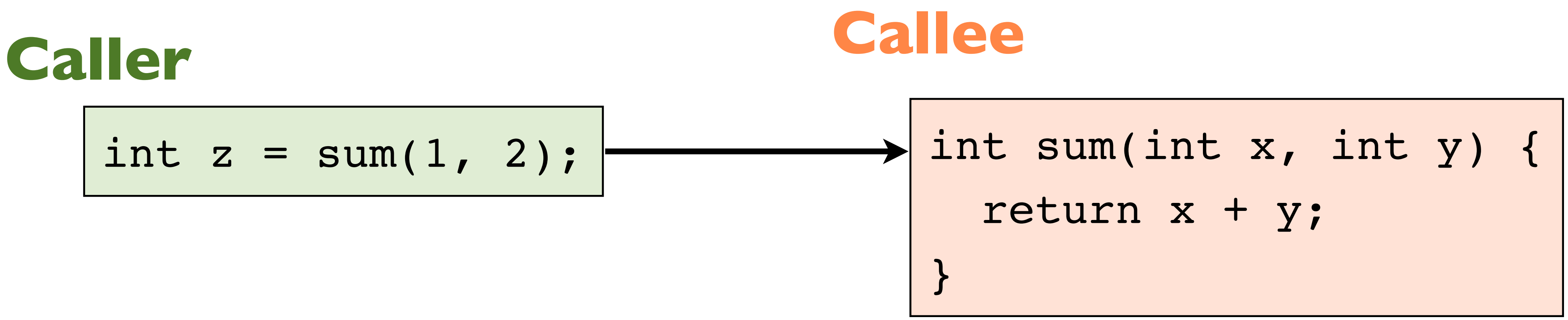
Check your textbook:

Don’t use the international edition!
The homework problems are different.

Today

- Questions on Hw 2 or Lab 2?
- Procedure calls

Procedure Call Example



Caller

```
int z = sum(1, 2);
```

Callee

```
int sum(int x, int y) {
  return x + y;
}
```

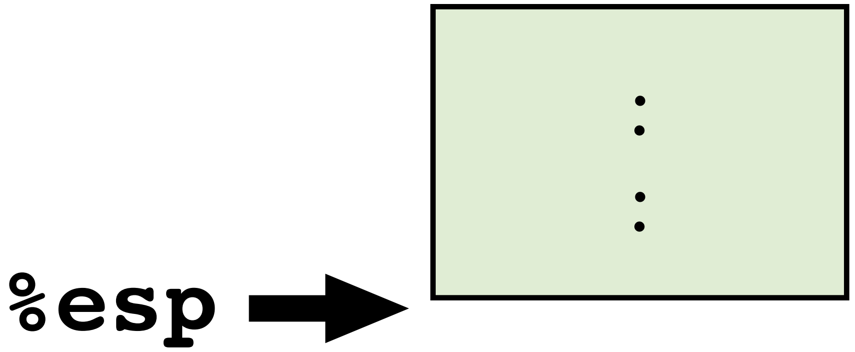
Procedure Call Example

(IA32/Linux)

Caller

```
int z = sum(1, 2);
```

The Stack



Caller in assembly

```
0x8001  pushl $2
0x8005  pushl $1
0x8009  call sum
0x8013  addl $8, %esp
```

*note: these instruction addresses are completely made up for this example

Procedure Call Example

(IA32/Linux)

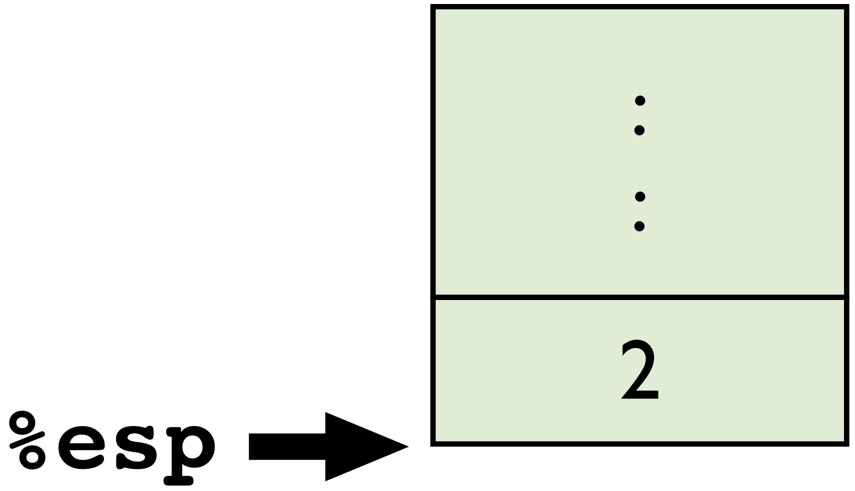
Caller

```
int z = sum(1, 2);
```

Caller in assembly

```
0x8001   pushl $2
0x8005   pushl $1
0x8009   call sum
0x8013   addl $8, %esp
```

The Stack



:

:

2

%esp

*note: these instruction addresses are completely made up for this example

Procedure Call Example

(IA32/Linux)

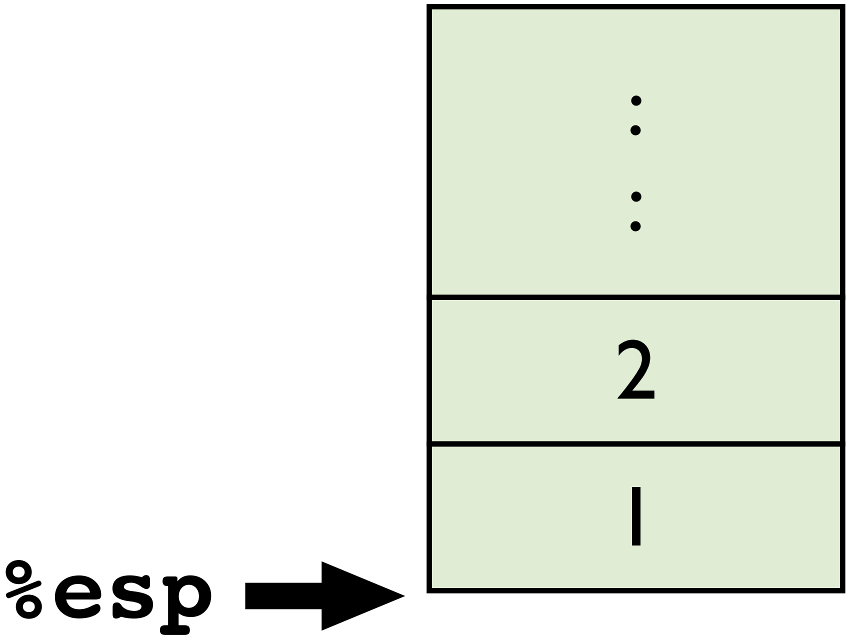
Caller

```
int z = sum(1, 2);
```

Caller in assembly

```
   0x8001   pushl $2
➡  0x8005   pushl $1
   0x8009   call sum
   0x8013   addl $8, %esp
```

The Stack



*note: these instruction addresses are completely made up for this example

Procedure Call Example

(IA32/Linux)

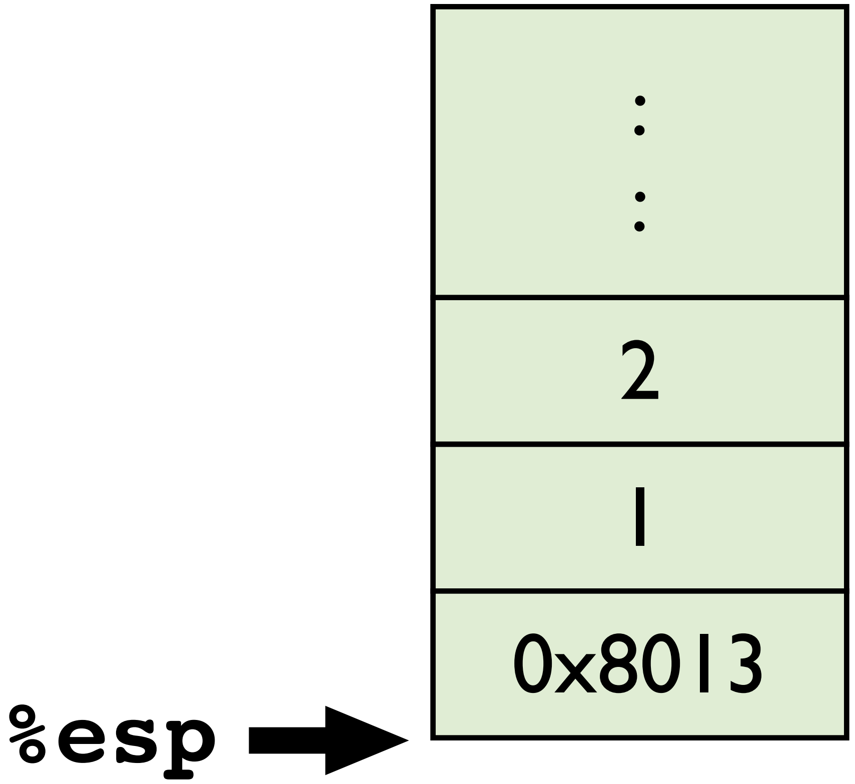
Caller

```
int z = sum(1, 2);
```

Caller in assembly

```
   0x8001  pushl $2
   0x8005  pushl $1
➡ 0x8009  call sum
   0x8013  addl $8, %esp
```

The Stack



:

:

2

1

0x8013

%esp

*note: these instruction addresses are
completely made up for this example

Procedure Call Example

(IA32/Linux)

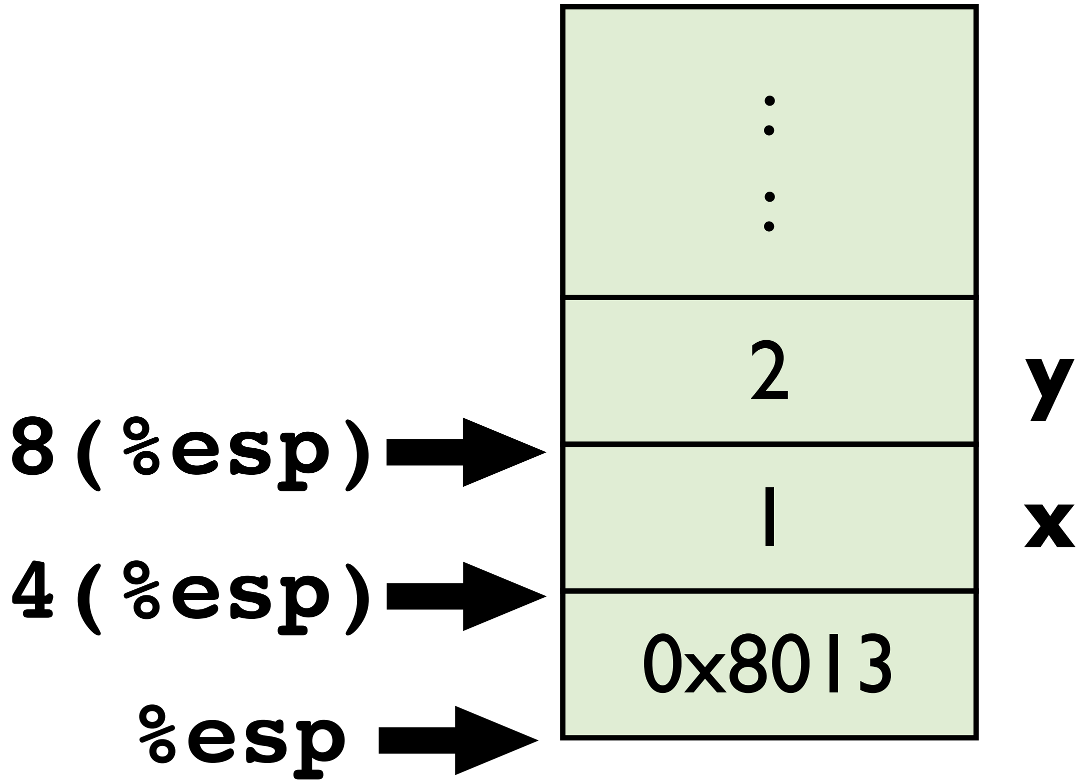
Callee

```
int sum(int x, int y) {
  return x + y;
}
```

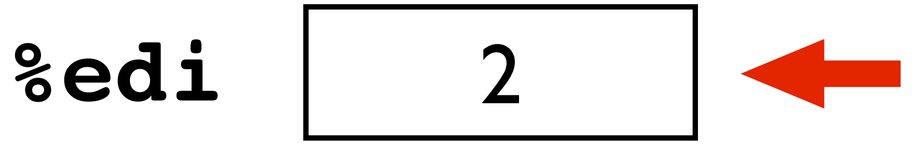
Callee in assembly (simple version)

```
movl 8(%esp), %edi
movl 4(%esp), %eax
addl %edi, %eax
ret
```

The Stack



Registers



Procedure Call Example

(IA32/Linux)

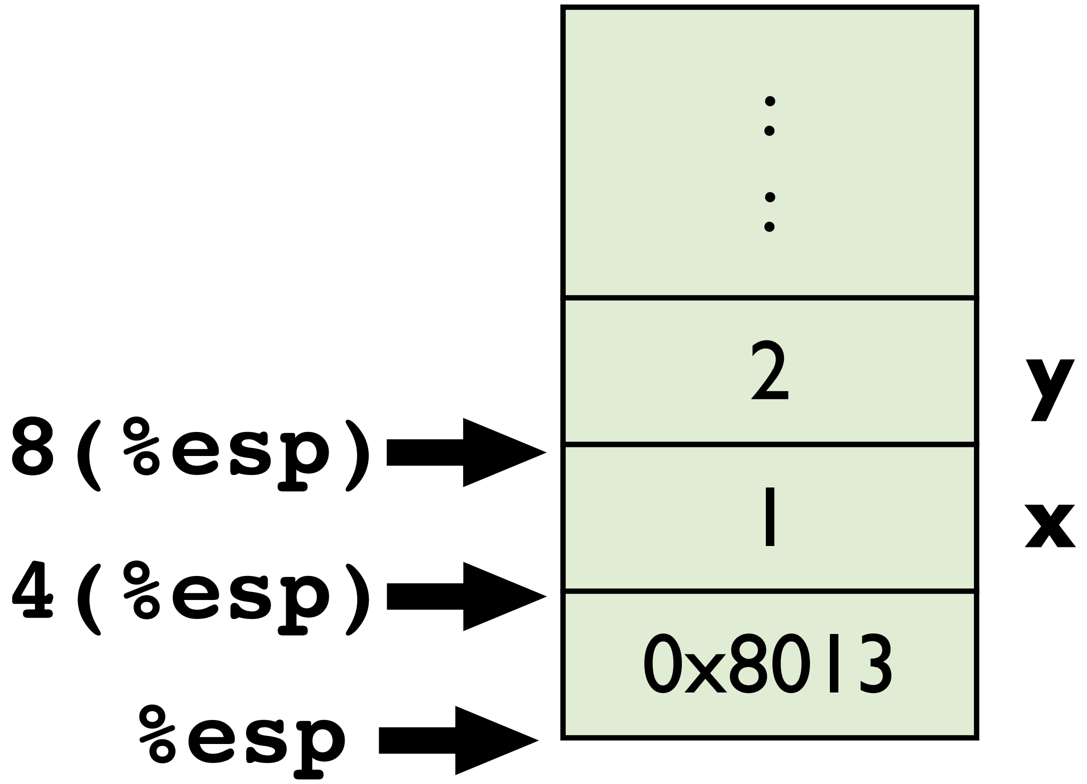
Callee

```
int sum(int x, int y) {
  return x + y;
}
```

Callee in assembly (simple version)

```
movl 8(%esp), %edi
movl 4(%esp), %eax
addl %edi, %eax
ret
```

The Stack



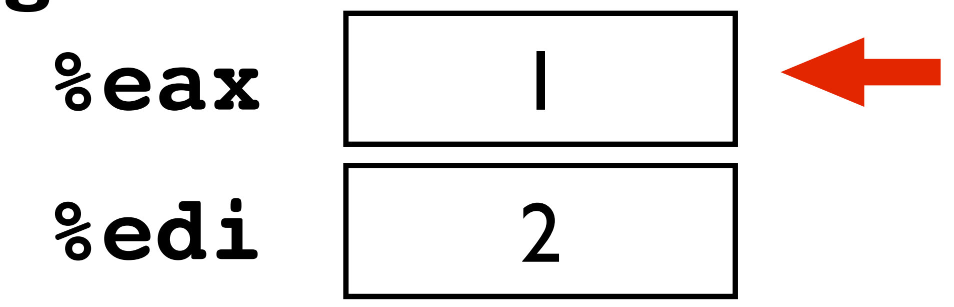
:
:

8(%esp) 2 y

4(%esp) 1 x

%esp 0x8013

Registers



%eax 1

%edi 2

Procedure Call Example

(IA32/Linux)

Callee

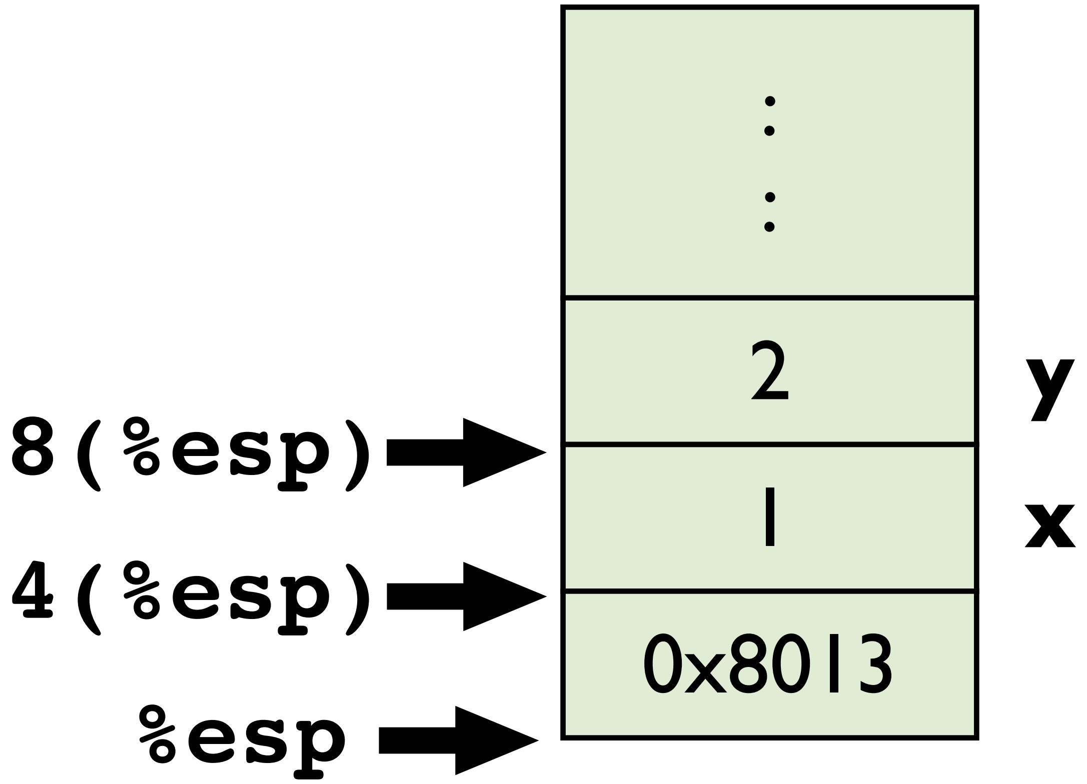
```
int sum(int x, int y) {
  return x + y;
}
```

Callee in assembly (simple version)

```
movl 8(%esp), %edi
movl 4(%esp), %eax
addl %edi, %eax
ret
```

%eax has the return value!

The Stack



Registers

%eax	3
%edi	2

Procedure Call Example

(IA32/Linux)

Callee

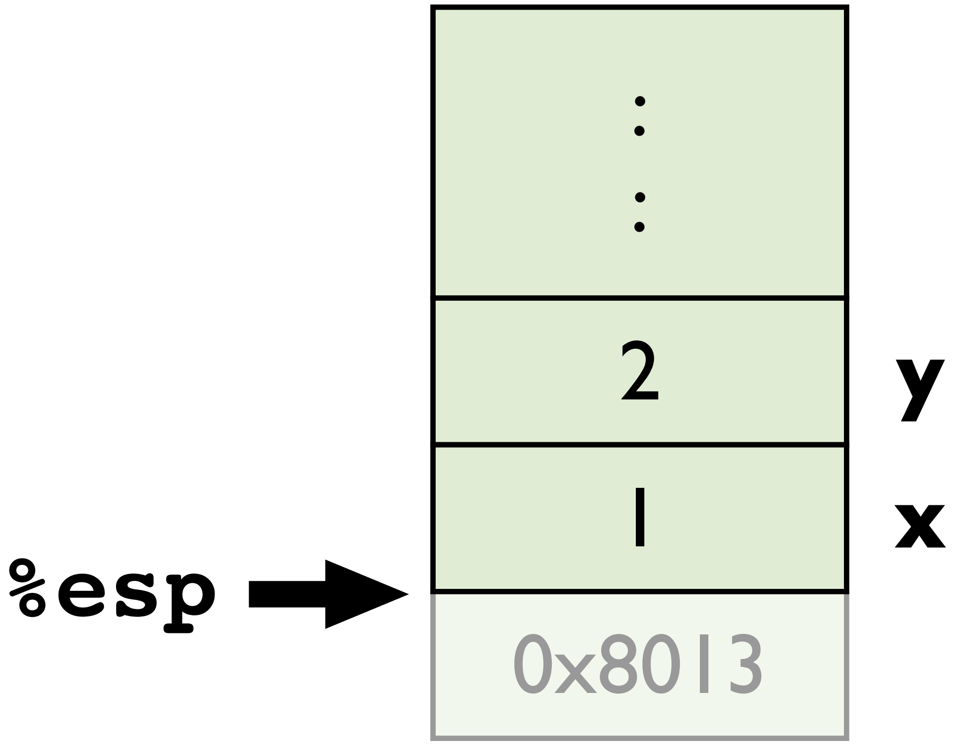
```
int sum(int x, int y) {
  return x + y;
}
```

Callee in assembly (simple version)

```
  movl 8(%esp), %edi
  movl 4(%esp), %eax
  addl %edi, %eax
→ ret
```

%eax has the return value!

The Stack



:
:

2 y

1 x

%esp → 0x8013

Registers

%eax	3
%edi	2
%eip	0x8013

Procedure Call Example

(IA32/Linux)

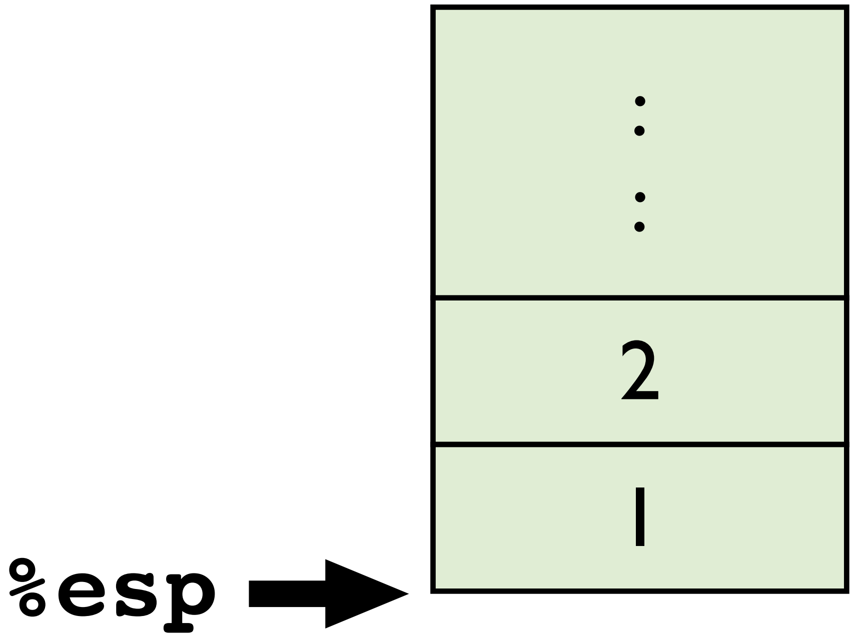
Caller

```
int z = sum(1, 2);
```

Caller in assembly

```
  0x8001   pushl $2
  0x8005   pushl $1
  0x8009   call sum
→ 0x8013   addl $8, %esp
```

The Stack



Registers

%eax	3
%edi	2
%eip	0x8013

*note: these instruction addresses are completely made up for this example

Procedure Call Example

(IA32/Linux)

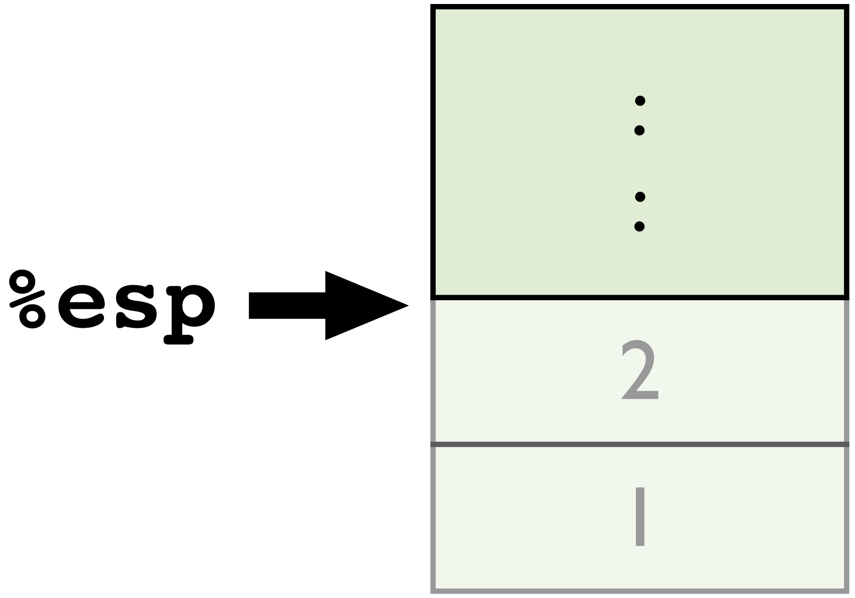
Caller

```
int z = sum(1, 2);
```

Caller in assembly

```
  0x8001  pushl $2
  0x8005  pushl $1
  0x8009  call sum
→ 0x8013  addl $8, %esp
```

The Stack



Registers

%eax	3
%edi	2
%eip	0x8013

*note: these instruction addresses are completely made up for this example

Procedure Call Example

(IA32/Linux)

Caller

```
int z = sum(1, 2);
```

Problem:

- **What if Caller used %edi before making the call?**

Caller in assembly

```
  0x8001  pushl $2
  0x8005  pushl $1
  0x8009  call sum
➡ 0x8013  addl $8, %esp
```

Registers

%eax	3
%edi	2
%eip	0x8013

*note: these instruction addresses are completely made up for this example

Procedure Call Example

(IA32/Linux)

Caller

```
int d = 5;
int z = sum(1, 2);
```

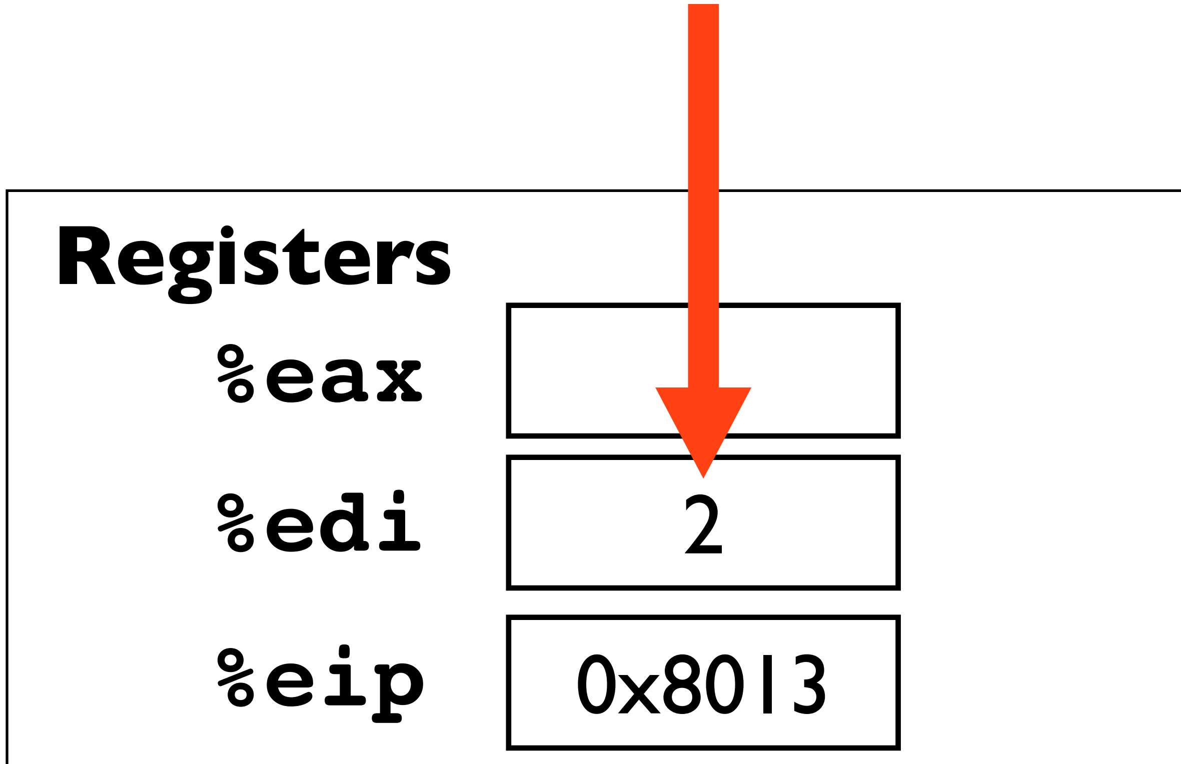
Problem:

- **What if Caller used %edi before making the call?**

Caller in assembly

```
  0x7fff   movl $5, %edi
  0x8001   pushl $2
  0x8005   pushl $1
  0x8009   call sum
→ 0x8013   addl $8, %esp
```

sum() overwrote %edi!
Need to save ...



Registers

%eax	
%edi	2
%eip	0x8013

*note: these instruction addresses are completely made up for this example

Saving Registers

- Some are ***caller save***
  - IA32: `%eax, %edx, %ecx`
  - These are very commonly used
    (caller should expect they will be clobbered)

- Some are ***callee save***
  - IA32: `%ebx, %edi, %esi`
  - These are less commonly used

from prior example

Procedure Call Example

(IA32/Linux)

Callee

```
int sum(int x, int y) {
  return x + y;
}
```

Callee in assembly (better version)

setup

```
pushl %ebp
movl  %esp, %ebp
pushl %edi
```

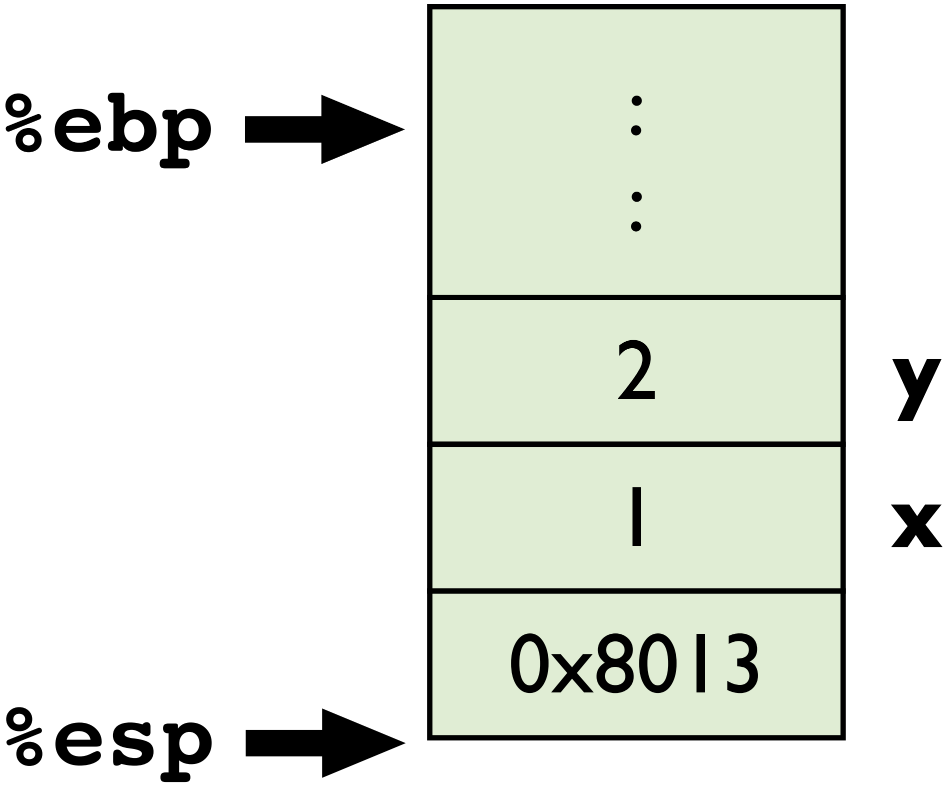
body

```
movl 12(%ebp), %edi
movl 8(%ebp), %eax
addl %edi, %eax
```

cleanup

```
movl (%esp), %edi
movl %ebp, %esp
popl %ebp
ret
```

The Stack



Procedure Call Example

(IA32/Linux)

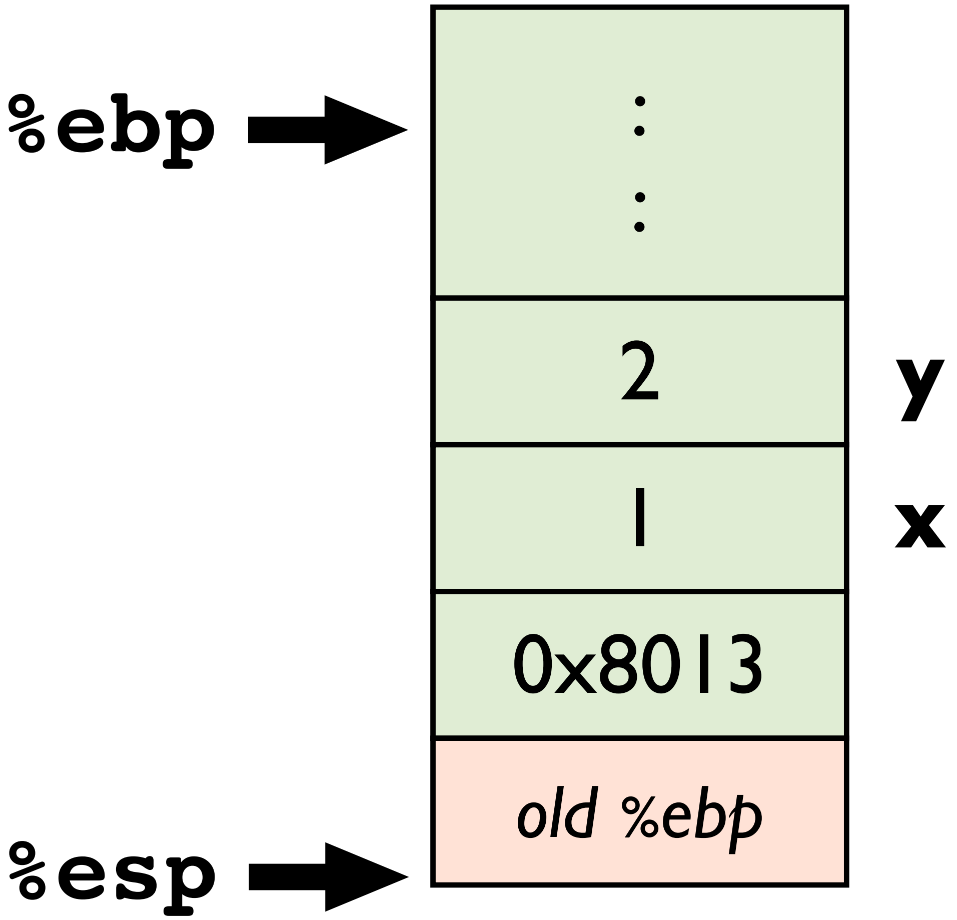
Callee

```
int sum(int x, int y) {
  return x + y;
}
```

Callee in assembly (better version)

```
          pushl %ebp
setup     movl  %esp, %ebp
          pushl %edi
-----------------------------
          movl 12(%ebp), %edi
body      movl 8(%ebp), %eax
          addl %edi, %eax
-----------------------------
          movl (%esp), %edi
cleanup   movl %ebp, %esp
          popl %ebp
          ret
```

The Stack



%ebp

:

:

2 y

1 x

0x8013

old %ebp

%esp

Procedure Call Example

(IA32/Linux)

Callee

```
int sum(int x, int y) {
  return x + y;
}
```

Callee in assembly (better version)

setup

```
  pushl %ebp
→ movl  %esp, %ebp
  pushl %edi
```

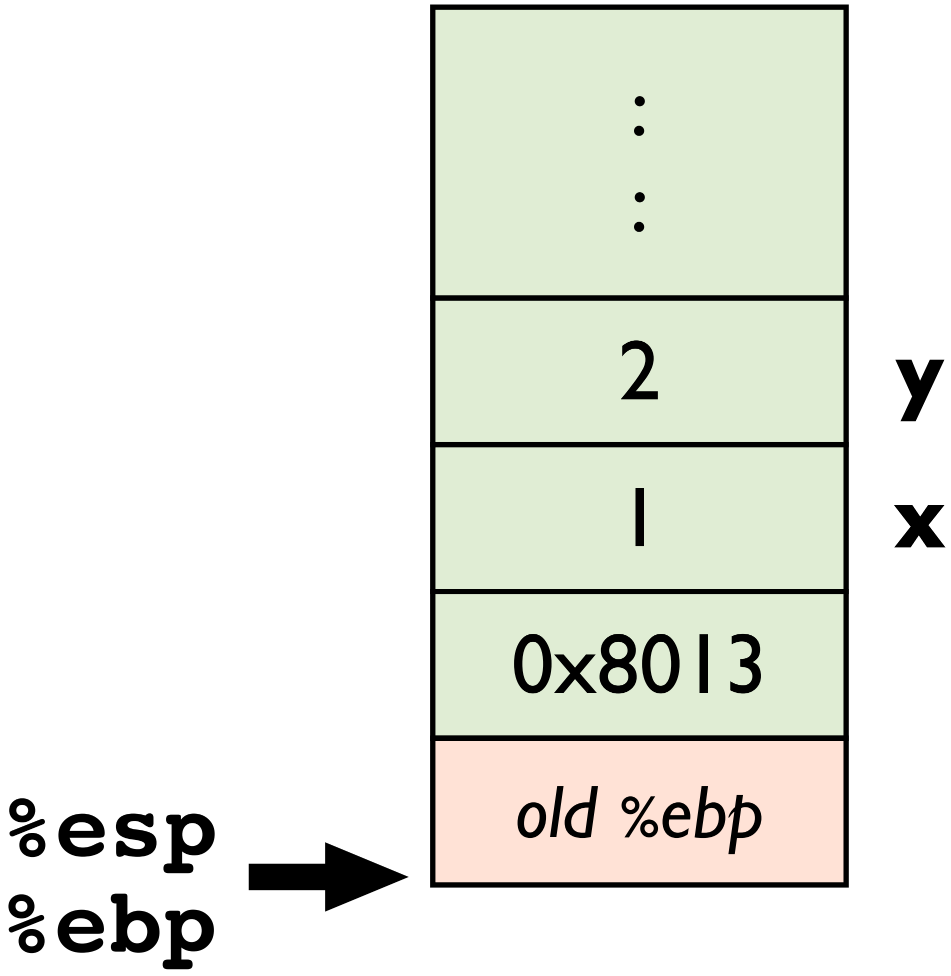
body

```
movl 12(%ebp), %edi
movl 8(%ebp), %eax
addl %edi, %eax
```

cleanup

```
movl (%esp), %edi
movl %ebp, %esp
popl %ebp
ret
```

The Stack



Procedure Call Example

(IA32/Linux)

Callee

```
int sum(int x, int y) {
  return x + y;
}
```

Callee in assembly (better version)

setup

```
    pushl %ebp
    movl  %esp, %ebp
→   pushl %edi
```

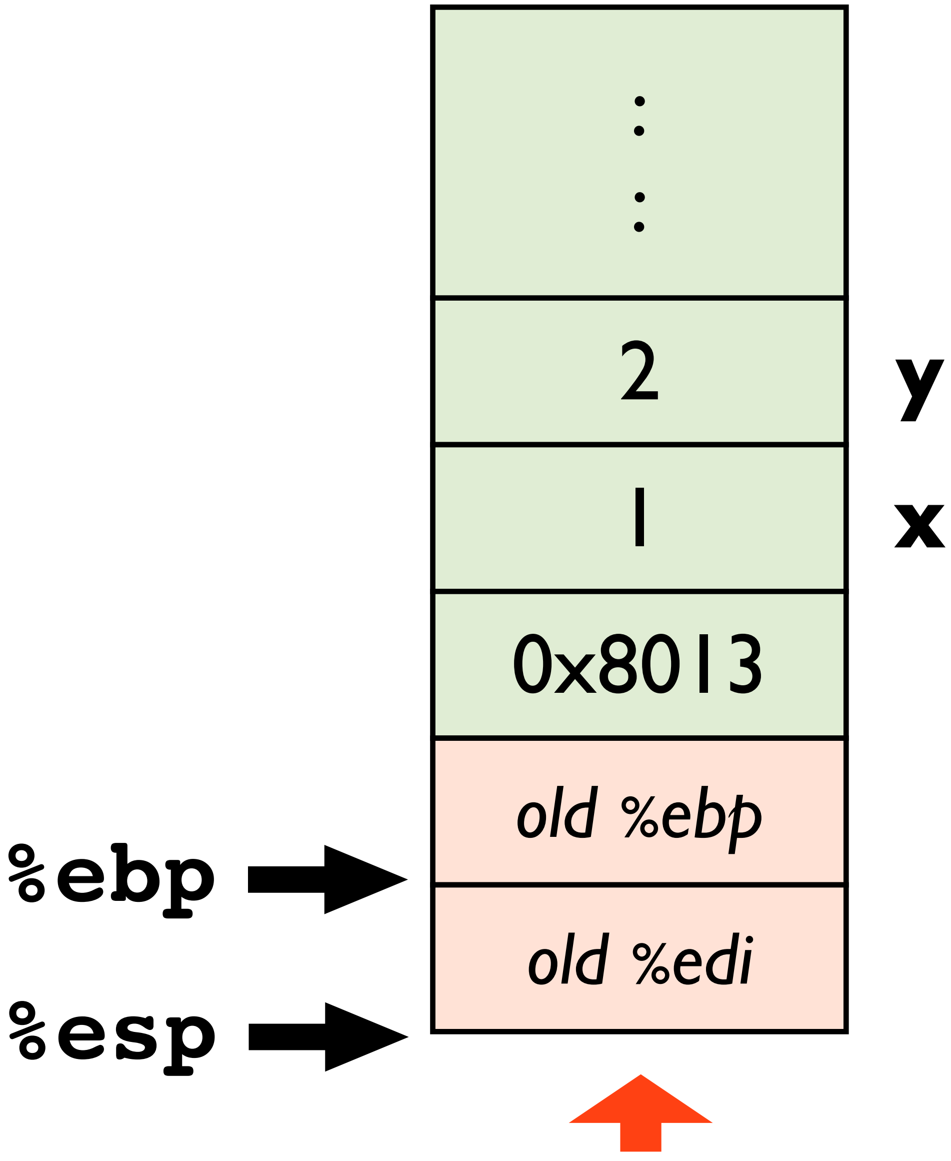
body

```
    movl 12(%ebp), %edi
    movl 8(%ebp), %eax
    addl %edi, %eax
```

cleanup

```
    movl (%esp), %edi
    movl %ebp, %esp
    popl %ebp
    ret
```

The Stack



saved %edi

Procedure Call Example

(IA32/Linux)

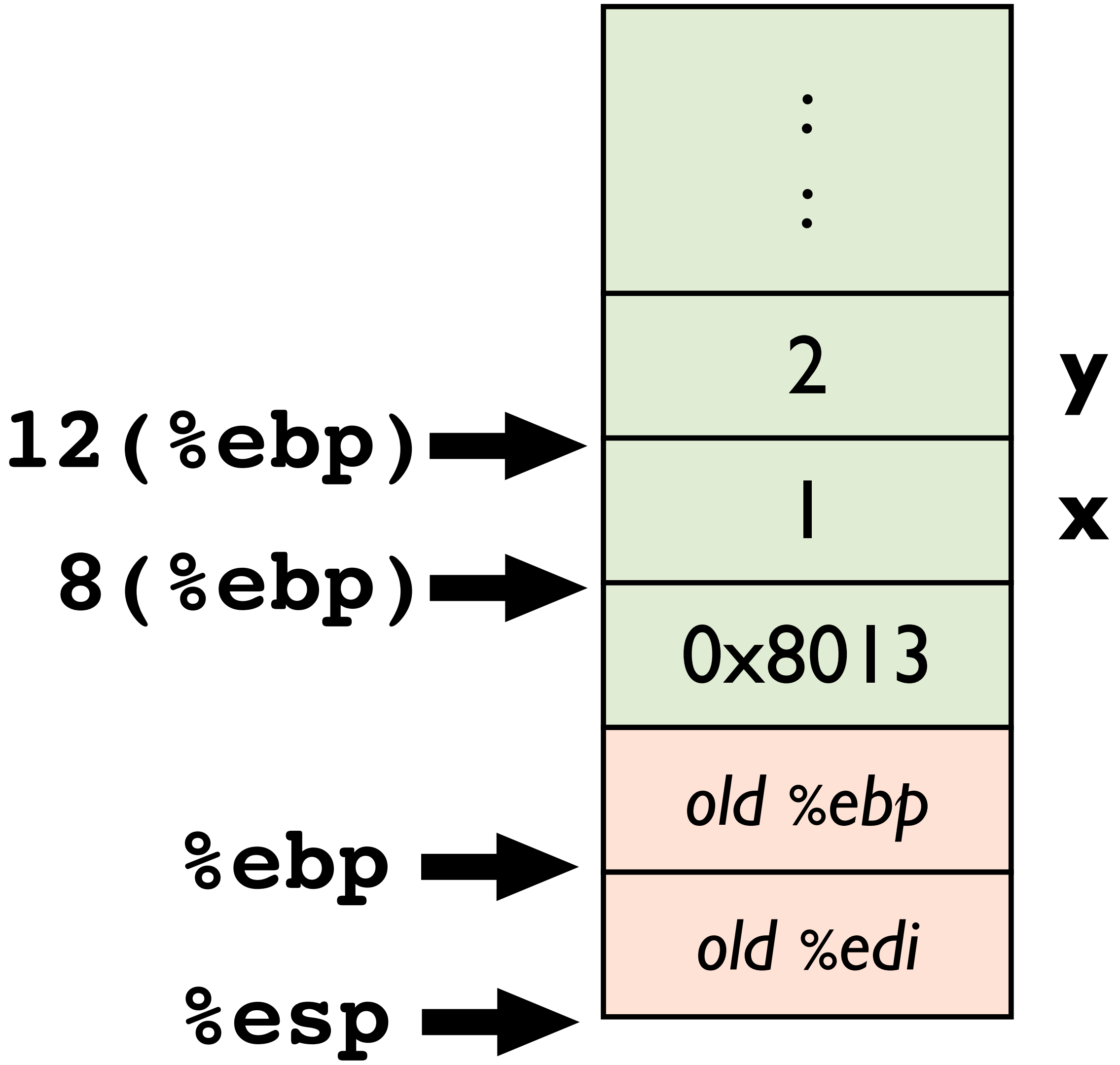
Callee

```
int sum(int x, int y) {
  return x + y;
}
```

Callee in assembly (better version)

```
setup      pushl %ebp
           movl  %esp, %ebp
           pushl %edi
---------------------------------
           movl 12(%ebp), %edi
body       movl 8(%ebp), %eax
           addl %edi, %eax
---------------------------------
           movl (%esp), %edi
cleanup    movl %ebp, %esp
           popl %ebp
           ret
```

The Stack



Key: %ebp is fixed for the entire function

Procedure Call Example

(IA32/Linux)

Callee

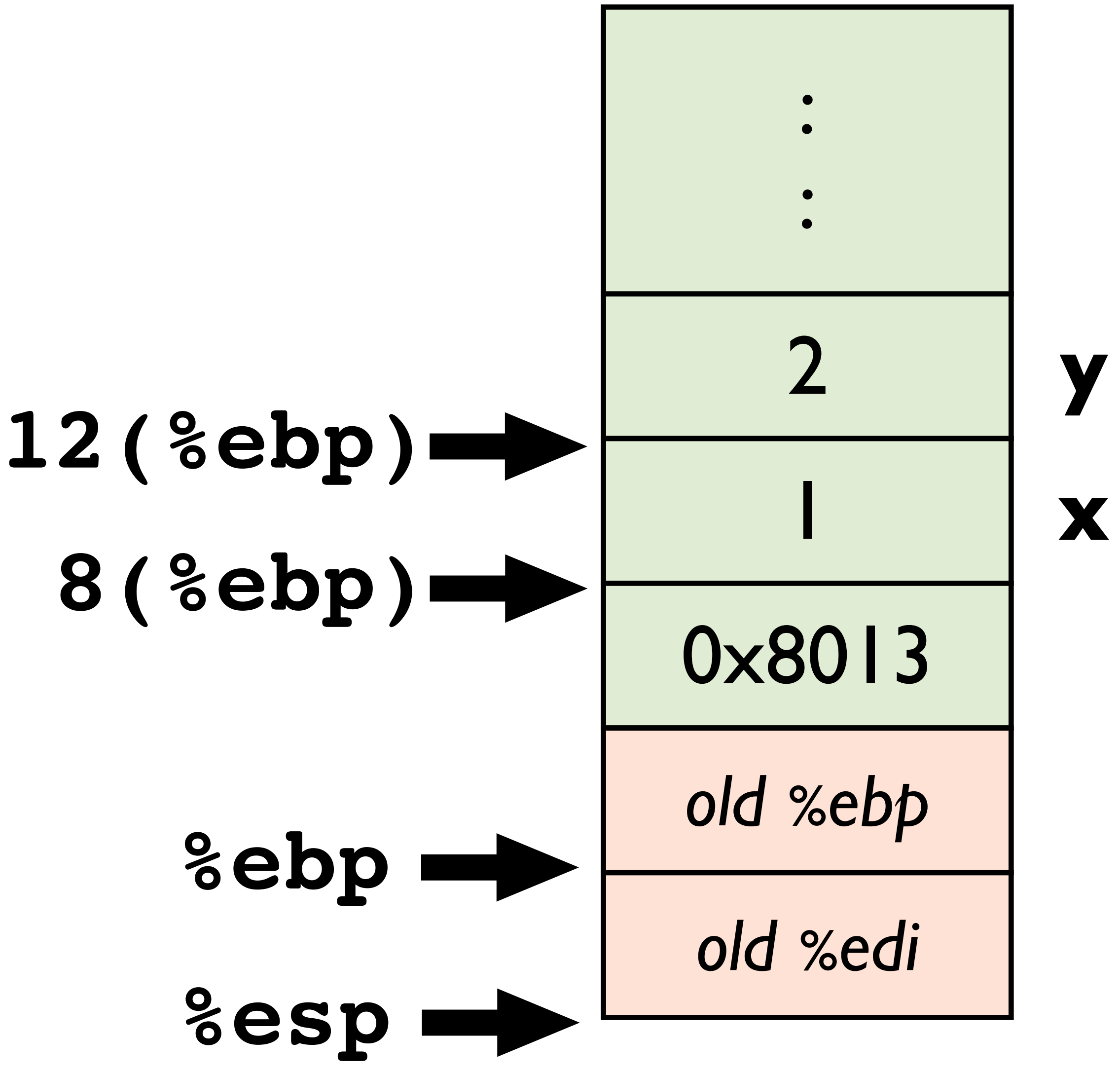
```
int sum(int x, int y) {
  return x + y;
}
```

Callee in assembly (better version)

```
             pushl %ebp
setup        movl  %esp, %ebp
             pushl %edi
---------------------------------
             movl 12(%ebp), %edi
body         movl 8(%ebp), %eax
             addl %edi, %eax
---------------------------------
         ➡   movl (%esp), %edi
cleanup      movl %ebp, %esp
             popl %ebp
             ret
```

restoring %edi

The Stack



Procedure Call Example

(IA32/Linux)

Callee

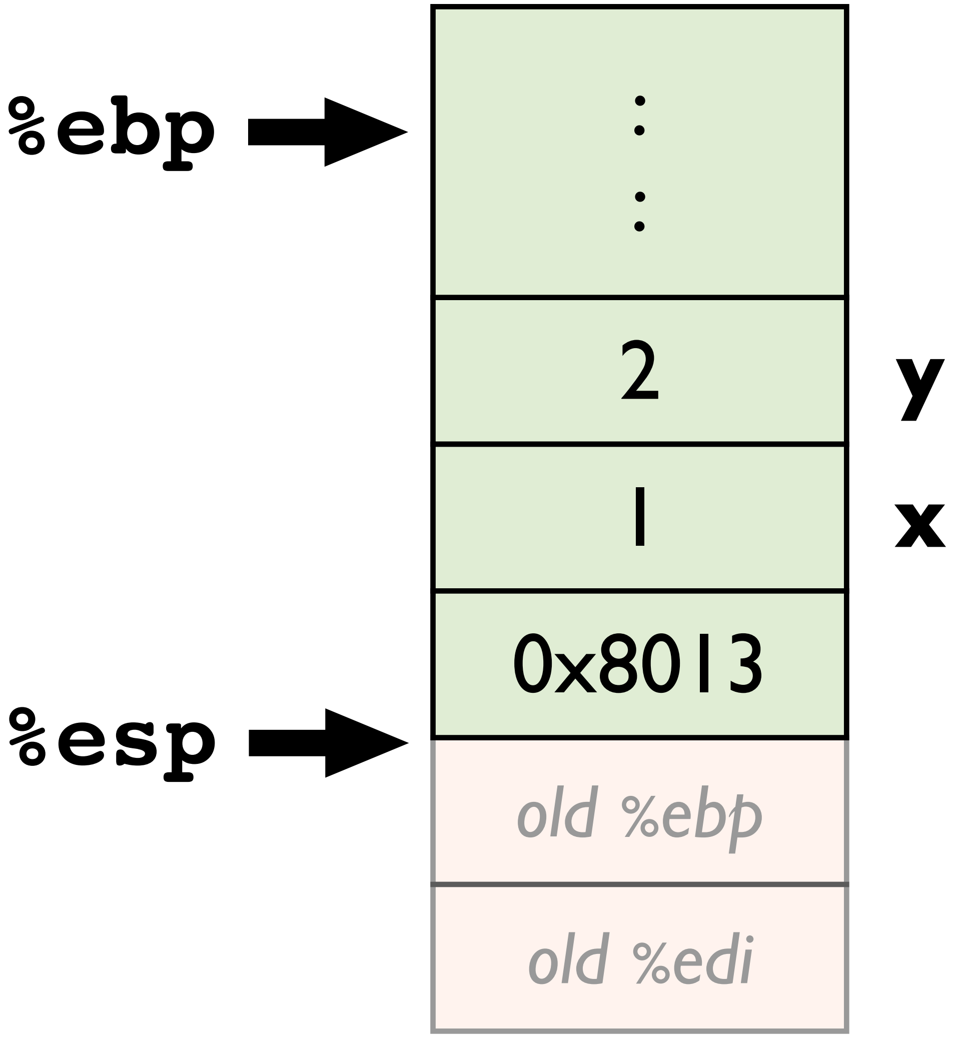
```
int sum(int x, int y) {
  return x + y;
}
```

Callee in assembly (better version)

```
setup      pushl %ebp
           movl  %esp, %ebp
           pushl %edi
--------------------------------
           movl 12(%ebp), %edi
body       movl 8(%ebp), %eax
           addl %edi, %eax
--------------------------------
           movl (%esp), %edi
cleanup    movl %ebp, %esp
       →   popl %ebp
           ret
```

restoring %ebp

The Stack



Why use a frame pointer?

(%ebp)

Callee

```
int sum(int x, int y) {
  return x + y;
}
```

To make debugging easier

- %esp may move
- %ebp is fixed

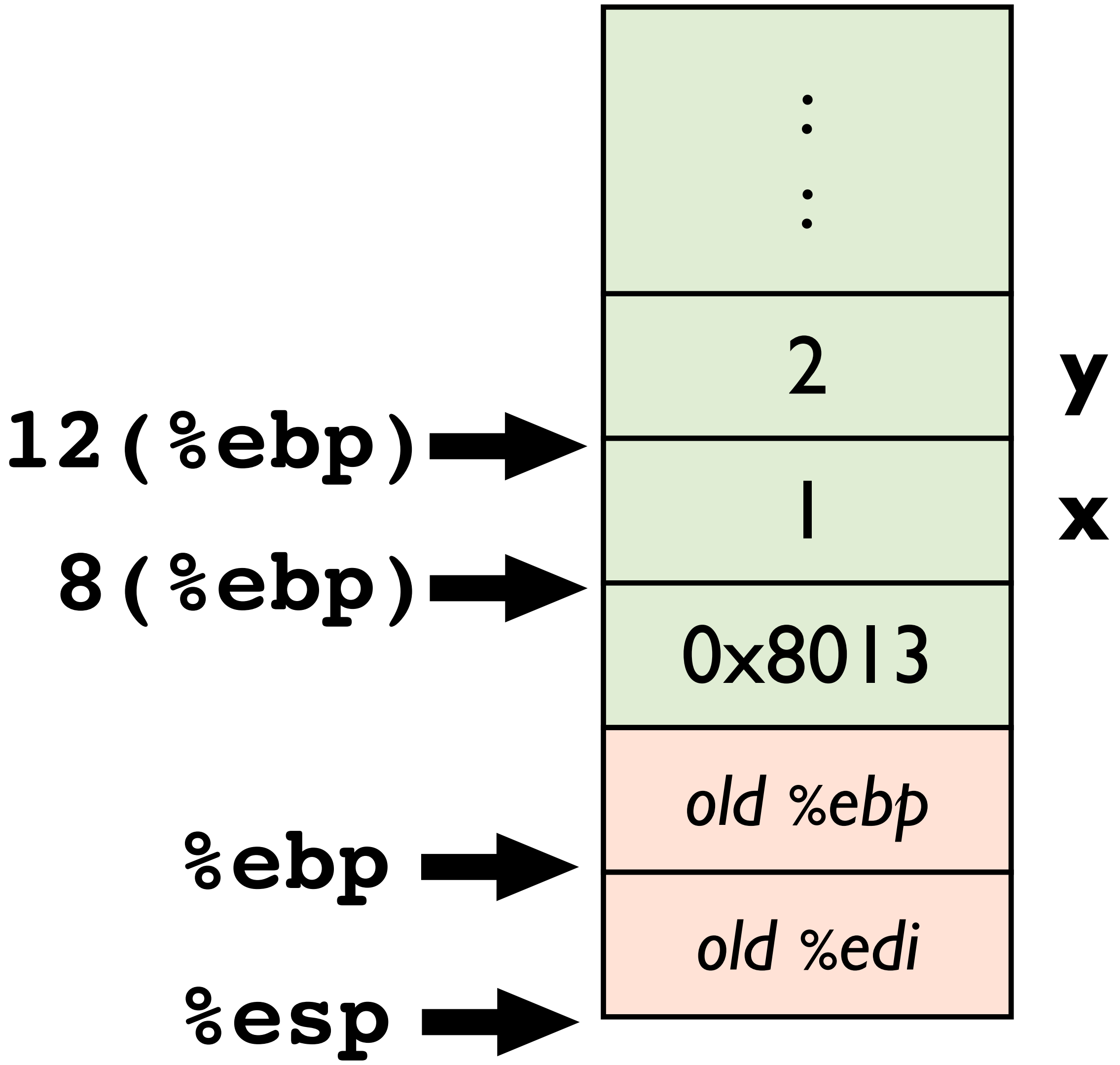
Your compiler emits a symbol map

```
y → 12(%ebp)
x → 8(%ebp)
```

gdb uses this map when you write

```
print x
```

The Stack



:
:
2
1
0x8013
old %ebp
old %edi

y
x

12(%ebp)
8(%ebp)
%ebp
%esp

Aside: how does gdb's "backtrace" work?

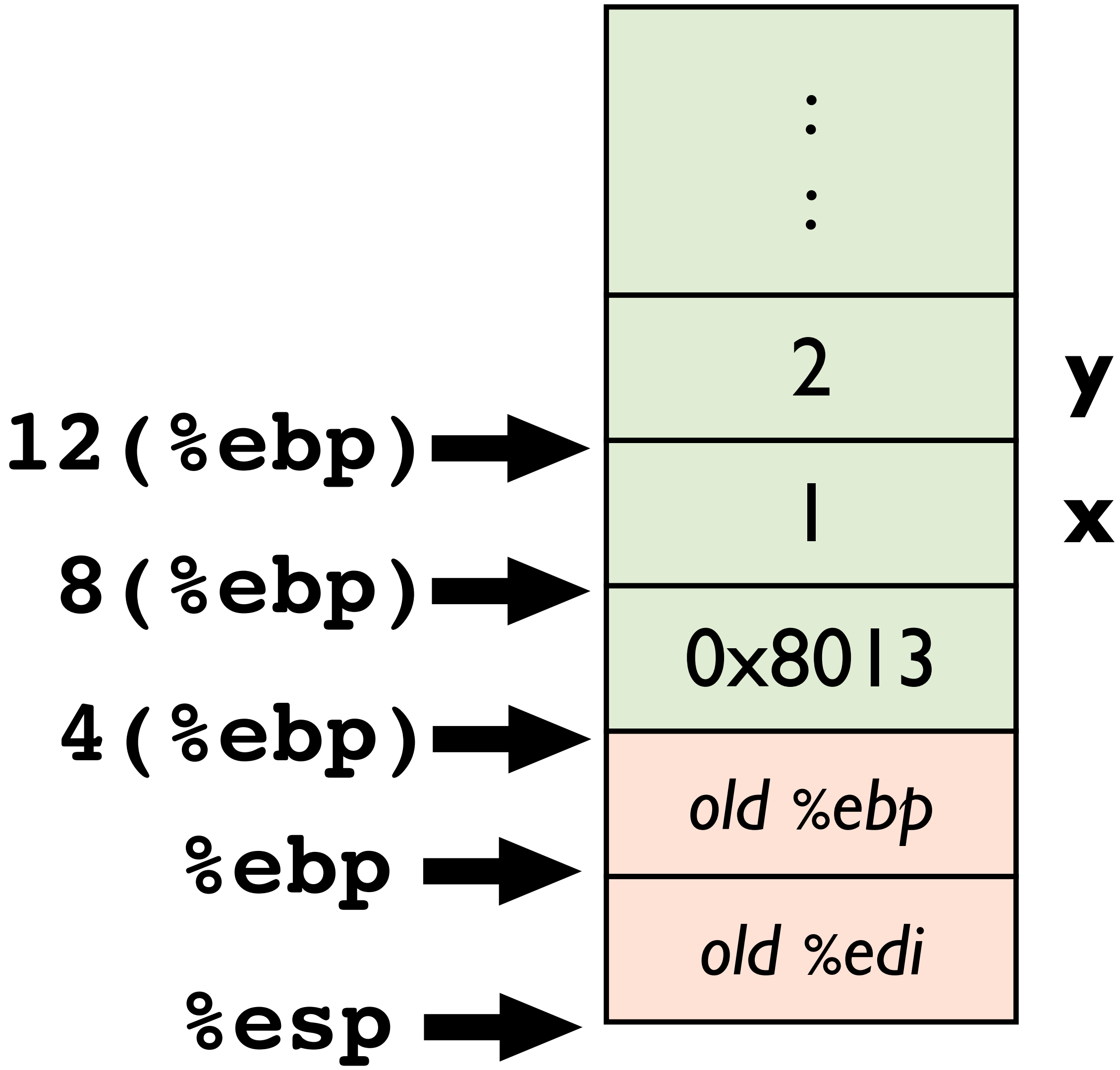
Follow return addresses!

- use *old %ebp* to find prior frame

Pseudocode:

```
while (pc is not in "main") {
    pc = 4(%ebp)
    %ebp = (%ebp)
}
```

The Stack



How is x86-64 different?

- Pass the first six arguments *in registers*
  - In this order: `%rdi,%rsi,%rdx,%rcx,%r8,%r9`
- New register save convention
  - *Callee save*: `%rbx,%rbp,%r12,%r13,%r14,%r15`
  - Others are *caller save*
- By default, gcc omits the frame pointer
  - It has to emit more complex debug info
    (e.g., the location of argument x relative to %esp can change)

Procedure Call Example

(x86-64/Linux)

Caller

```
int z = sum(1, 2);
```

Caller in assembly

```
movl $1, %edi
movl $2, %esi
call sum
```

edi not rdi because int is 32-bits

Callee

```
int sum(int x, int y) {
  return x + y;
}
```

Callee in assembly

```
addl %esi, %edi
movl %edi, %eax
ret
```

x86-64 with gcc does not use a frame pointer

Tip: you can force gcc to emit code with a frame pointer using `gcc -fno-omit-frame-pointer`