Systems I

Machine-Level Programming V: Procedures

Topics

- Stack abstraction and implementation
- IA32 stack discipline

Procedural Memory Usage

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

Where is the memory that holds t0 and t1 (or local variables in general)?

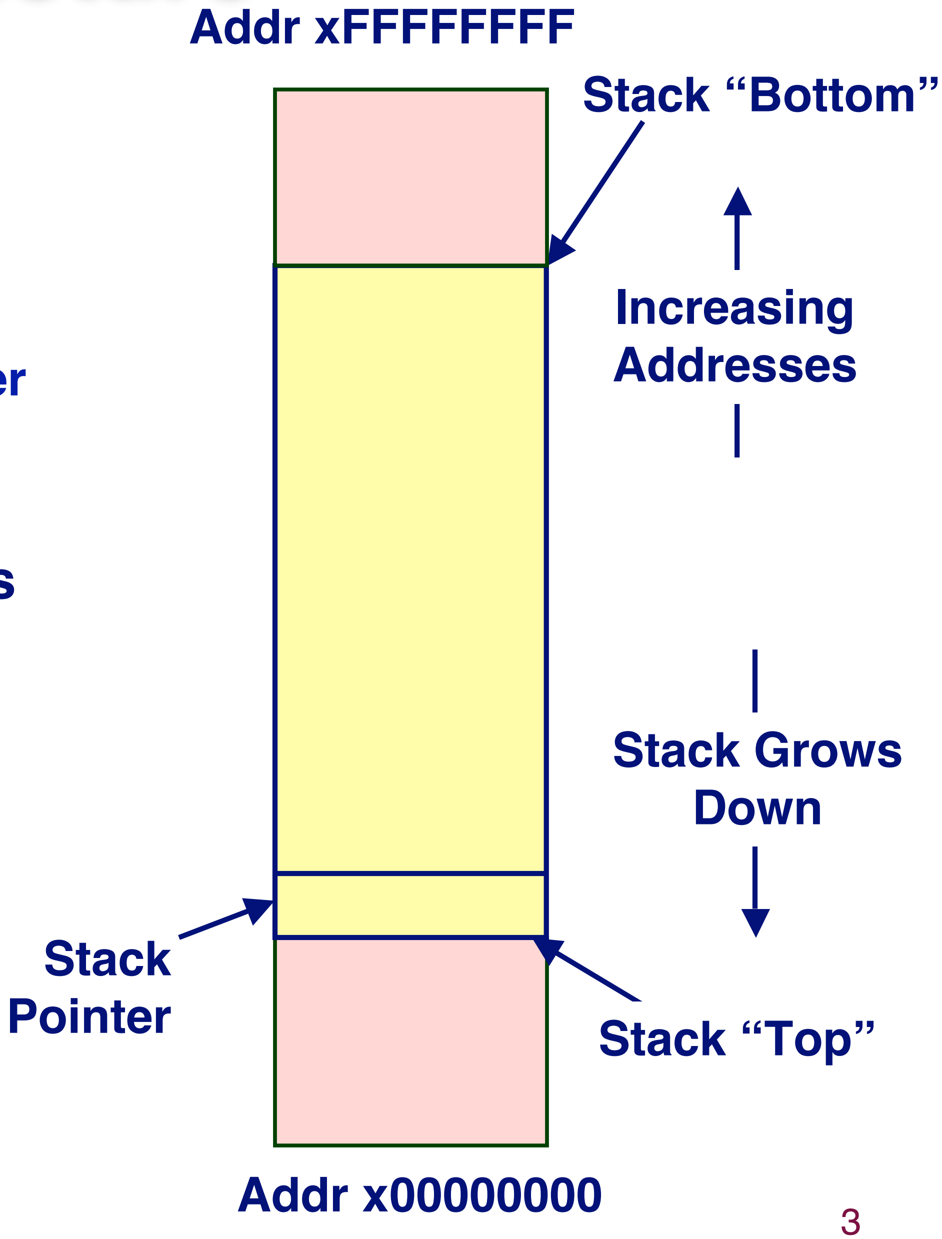
What happens if we run out of registers (x86 only has 8!)?

Where are parameters passed from callers to callee?

- Registers? Memory? What memory?

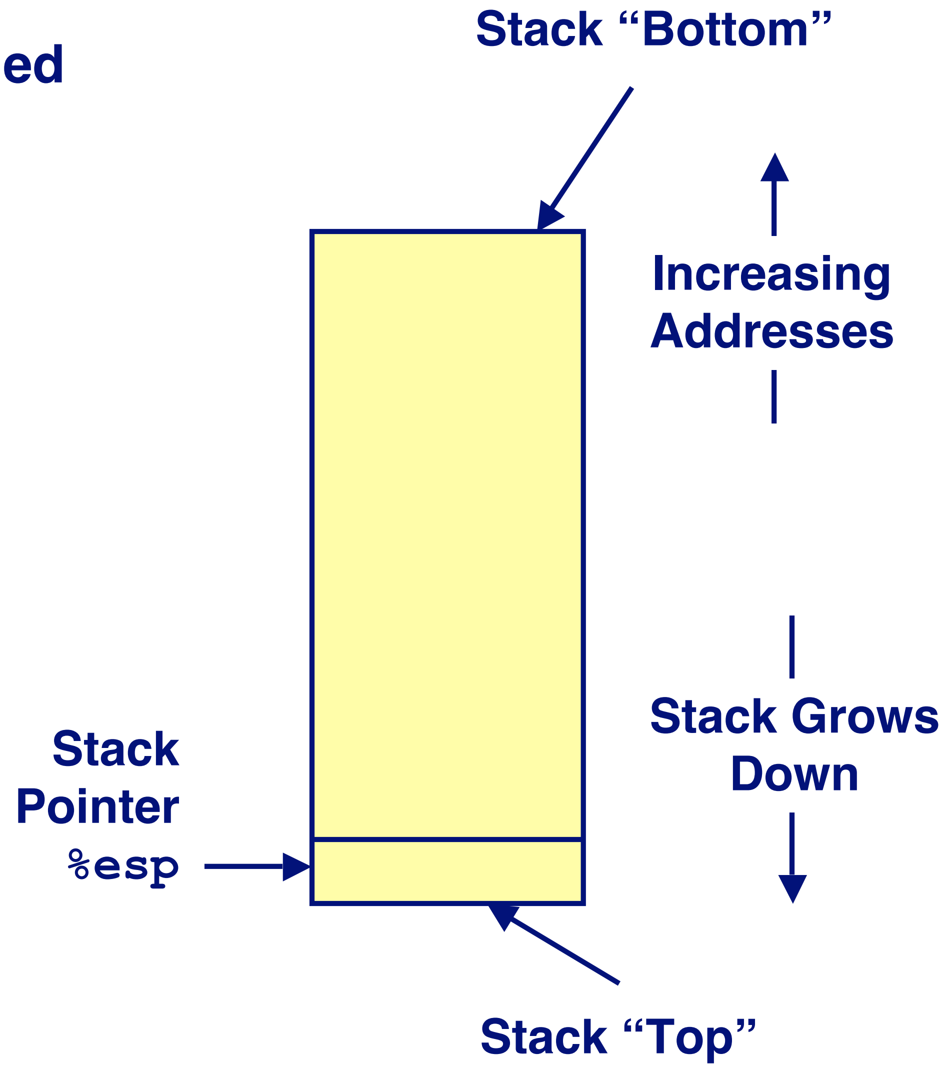
Stack Data Structure

- LIFO data structure
  - Last In, First Out
- Allocated somewhere in memory
  - Where doesn't really matter as long as we store the stack pointer
- By convention, stack grows toward smaller addresses
  - Could do it either way
- Values within the stack are referenced relative to the stack pointer



IA32 Stack

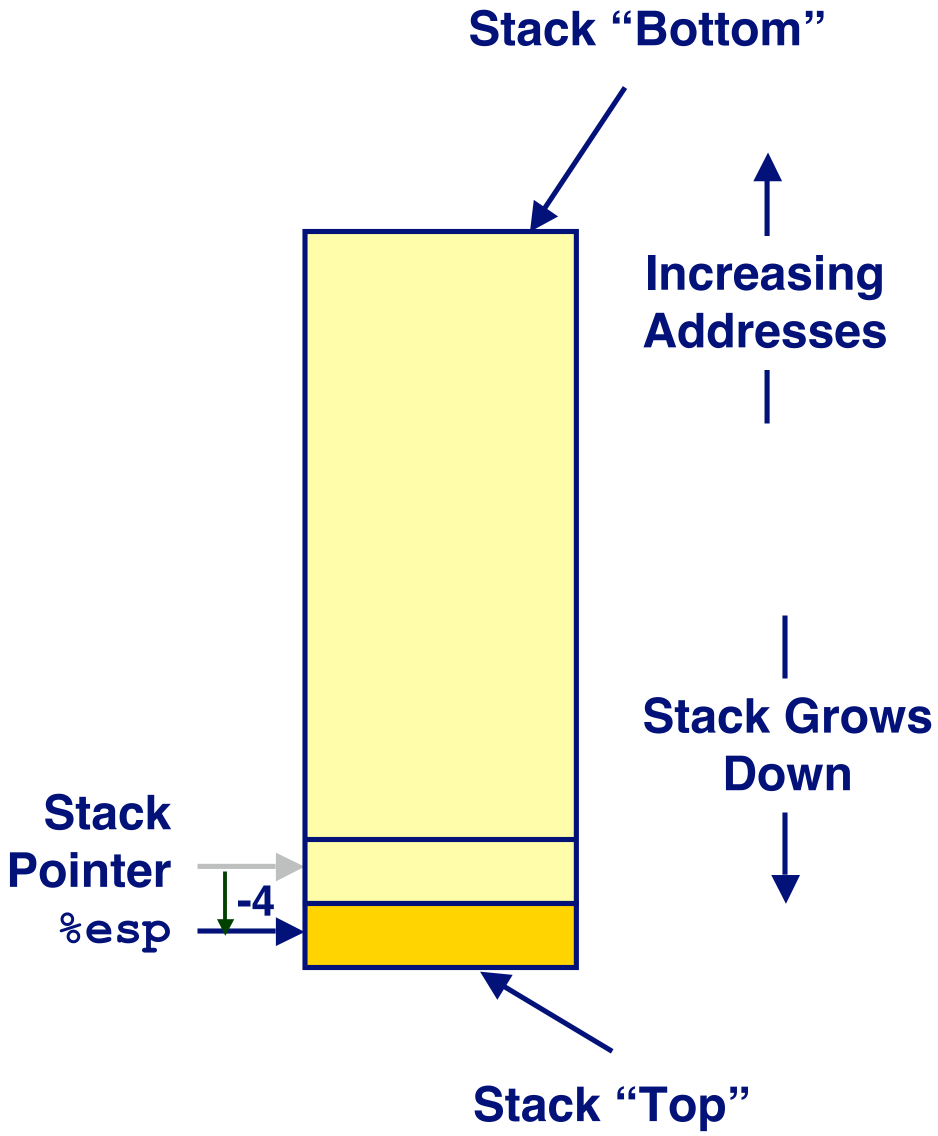
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
  - address of top element



IA32 Stack Pushing

Pushing

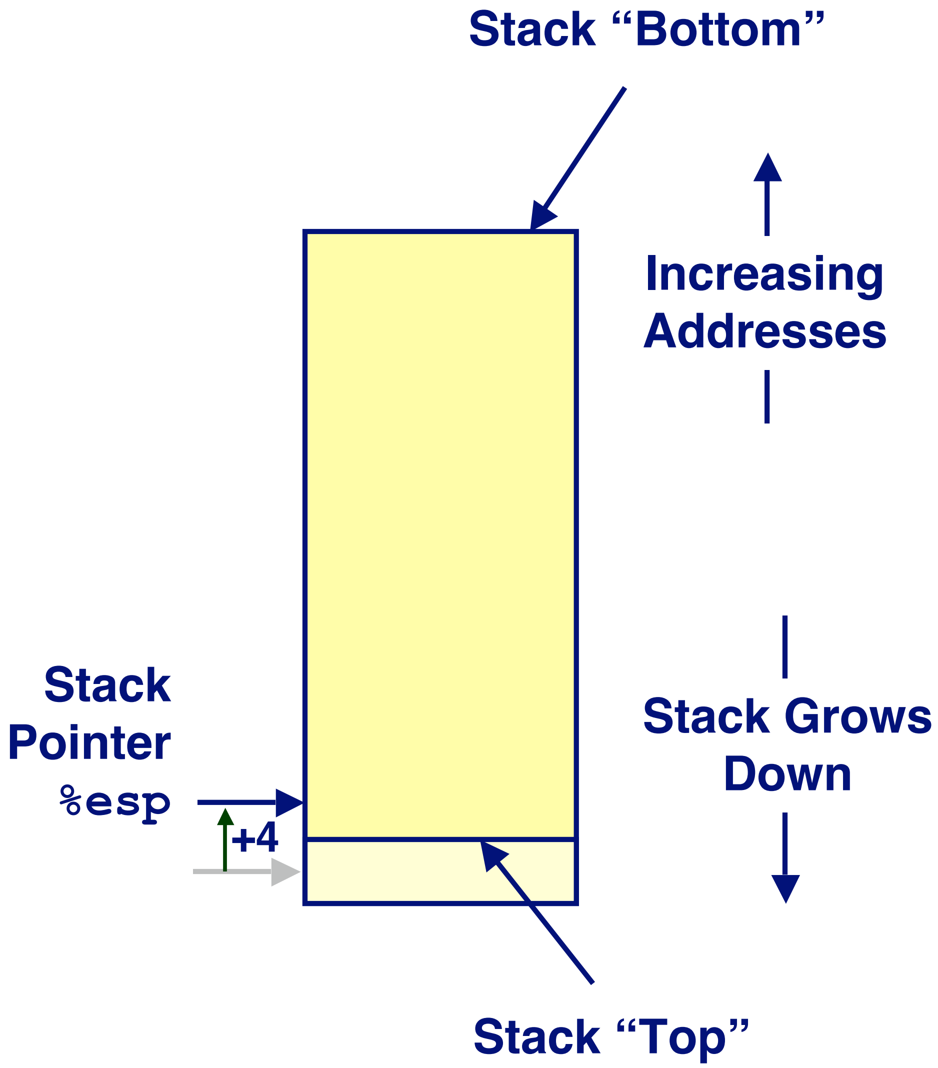
- `pushl` *Src*
- Fetch operand at *Src*
- Decrement `%esp` by 4
- Write operand at address given by `%esp`



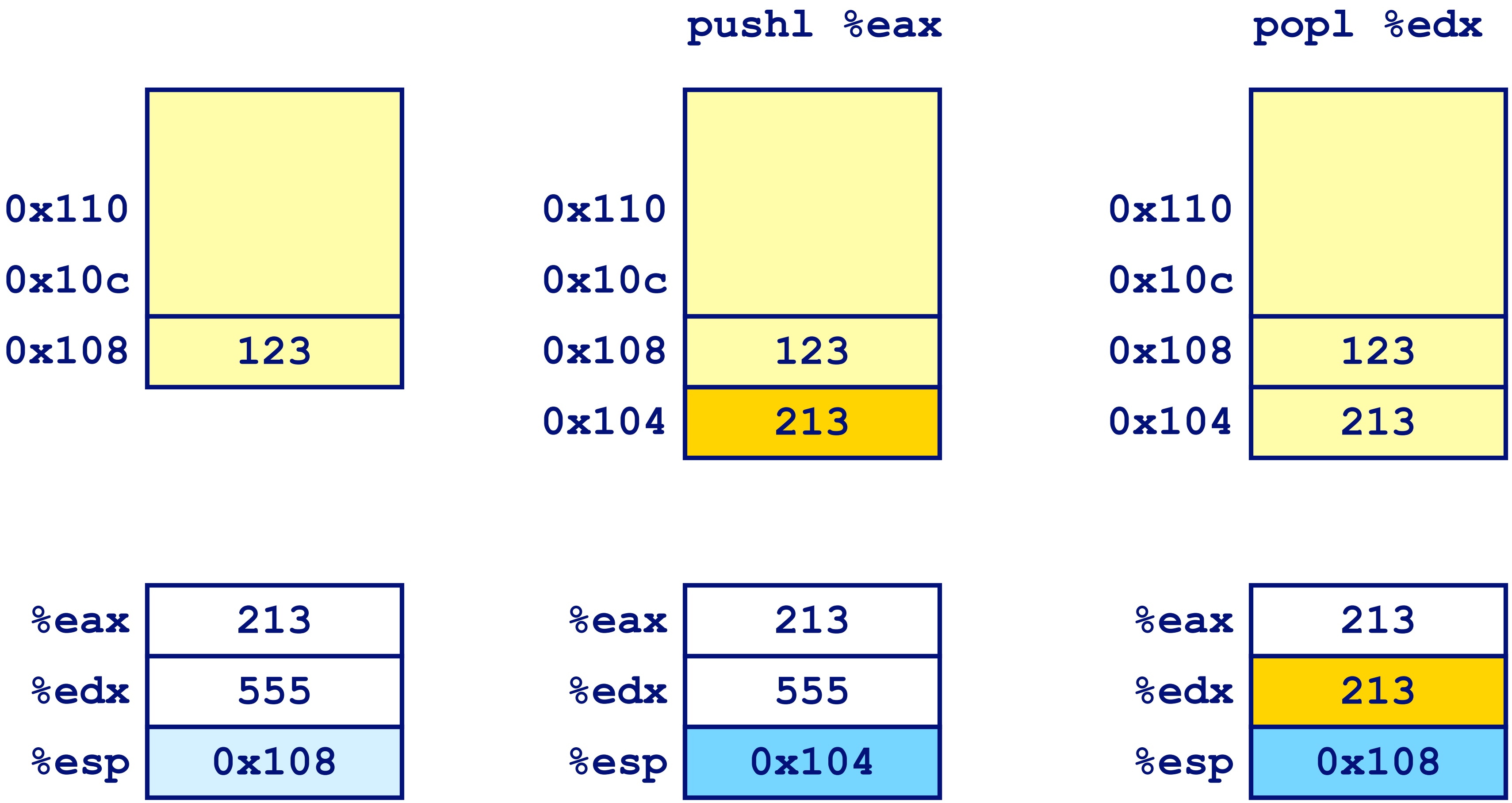
IA32 Stack Popping

Popping

- `popl` *Dest*
- Read operand at address given by `%esp`
- Increment `%esp` by 4
- Write to *Dest*



Stack Operation Examples



0x110	
0x10c	
0x108	123

%eax	213
%edx	555
%esp	0x108

`pushl %eax`

0x110	
0x10c	
0x108	123
0x104	213

%eax	213
%edx	555
%esp	0x104

`popl %edx`

0x110	
0x10c	
0x108	123
0x104	213

%eax	213
%edx	213
%esp	0x108

What Elements for Procedures?

Method of computing address of first instruction of called procedure.

Place to store passed parameters.

- Call by value or by reference

Method of computer return address

- Need to come back to first instruction after point of procedure call

Method of passing return value(s) back

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 value

- Address of instruction beyond `call`
- Example from disassembly

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

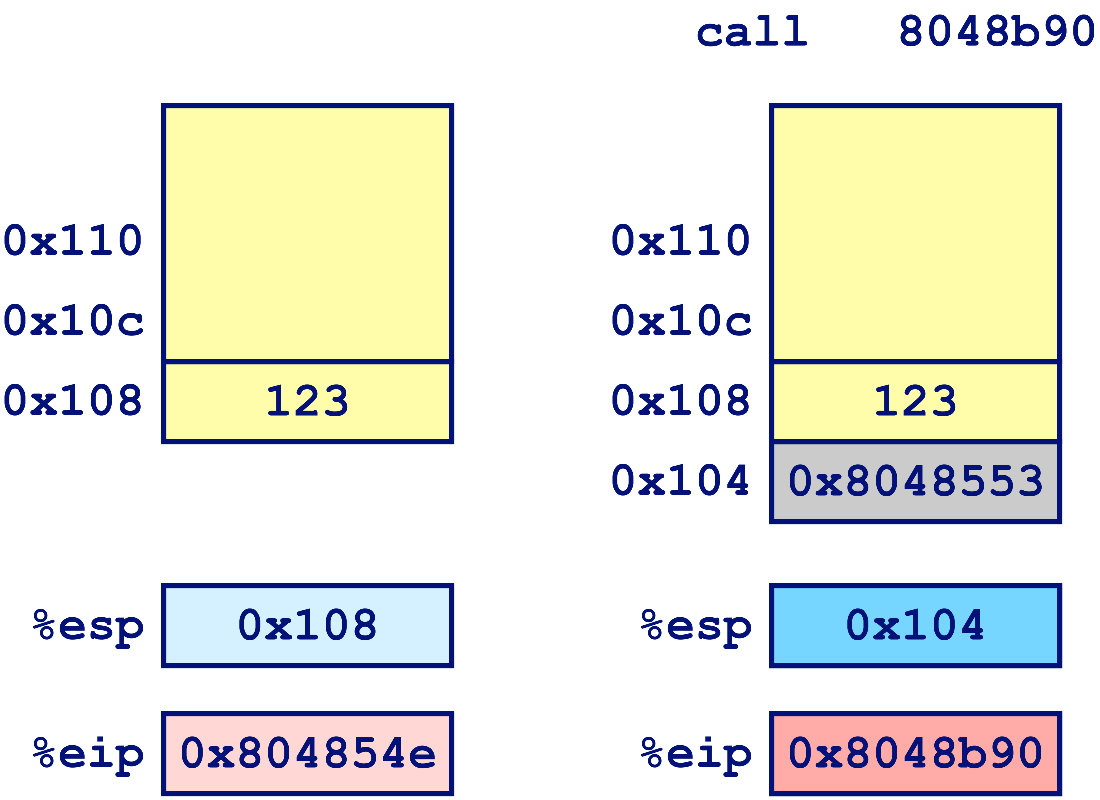
  - Return address = `0x8048553`

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
```



call 8048b90

0x110

0x10c

0x108 123

%esp 0x108

%eip 0x804854e

0x110

0x10c

0x108 123

0x104 0x8048553

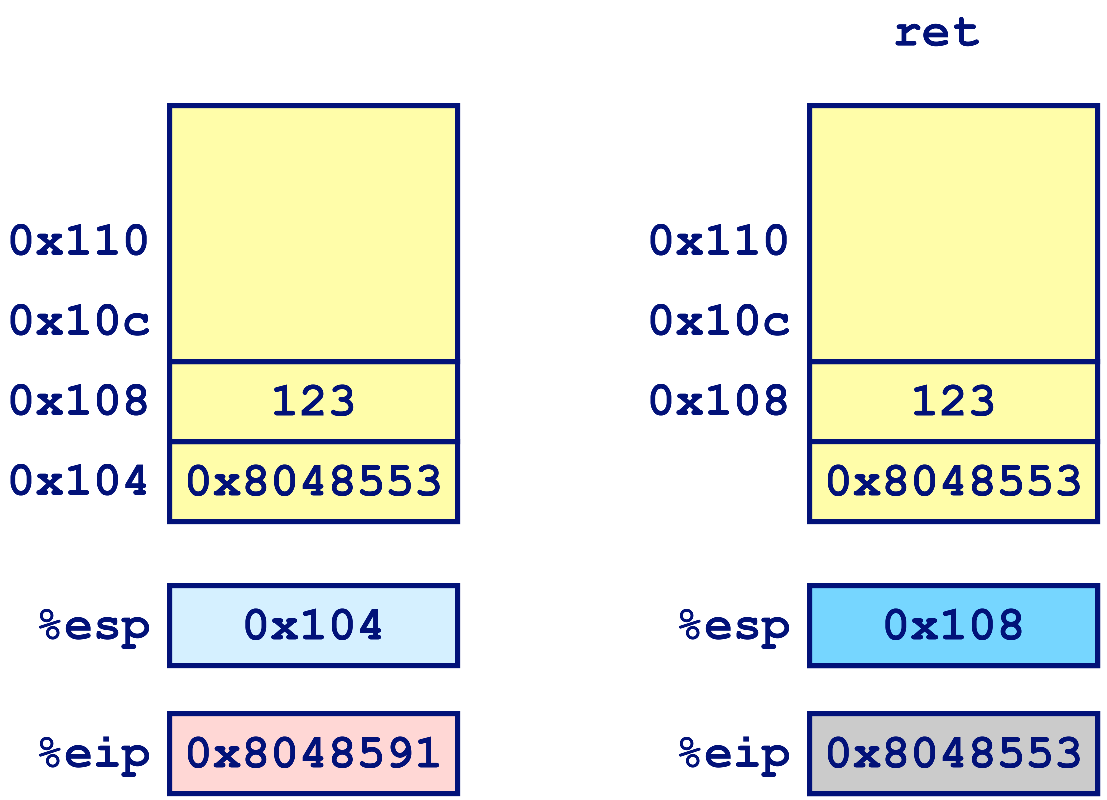
%esp 0x104

%eip 0x8048b90

`%eip` is program counter

Procedure Return Example

```
 8048591:   c3                  ret
```



ret

0x110

0x10c

0x108 123

0x104 0x8048553

%esp 0x104

%eip 0x8048591

0x110

0x10c

0x108 123

0x8048553

%esp 0x108

%eip 0x8048553

`%eip` **is 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

Code Structure

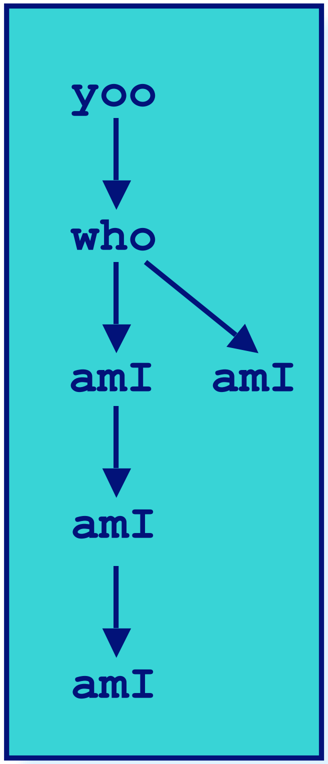
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

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

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

- Procedure `amI` recursive

Call Chain



Stack Frames

Contents

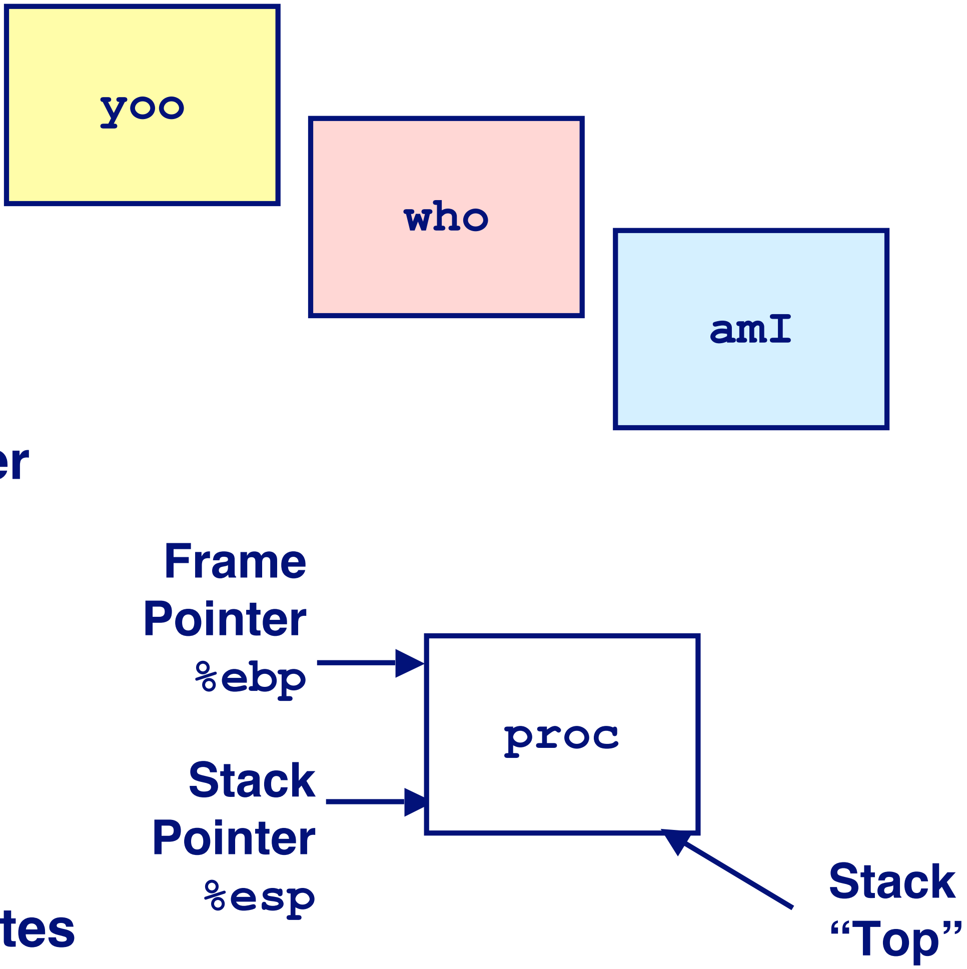
- Local variables
- Return information
- Temporary space

Management

- Space allocated when enter procedure
  - “Set-up” code
- Deallocated when return
  - “Finish” code

Pointers

- Stack pointer `%esp` indicates stack top
- Frame pointer `%ebp` indicates start of current frame

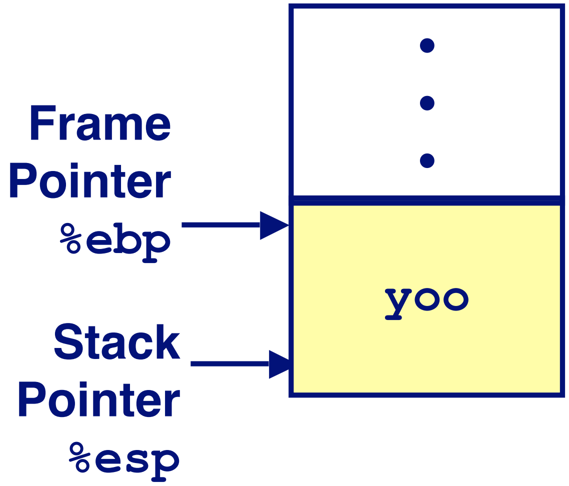


Stack Operation

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

Call Chain

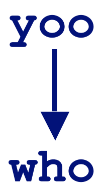
yoo

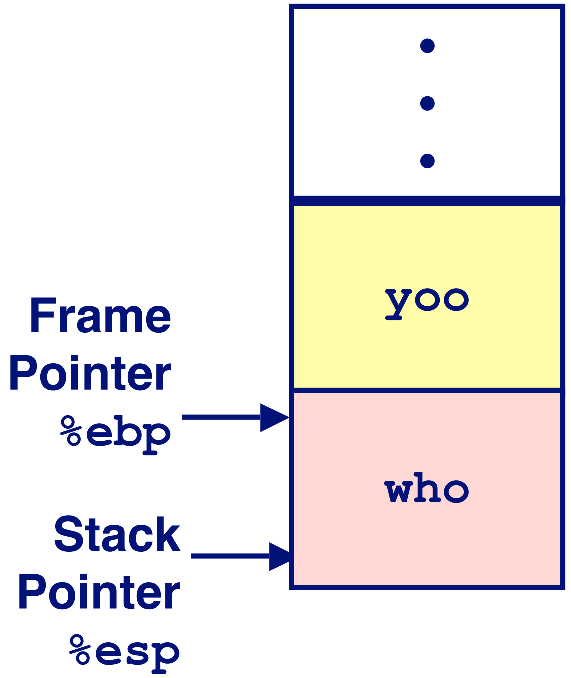


Stack Operation

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

Call Chain

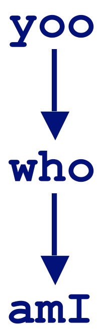


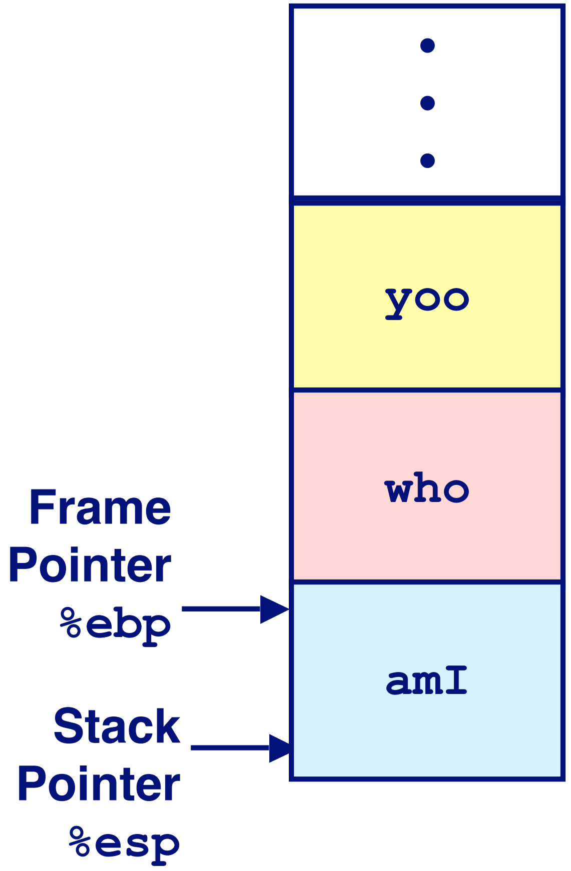


Stack Operation

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

Call Chain

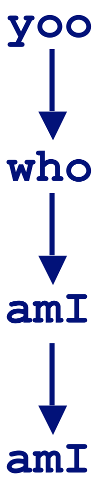


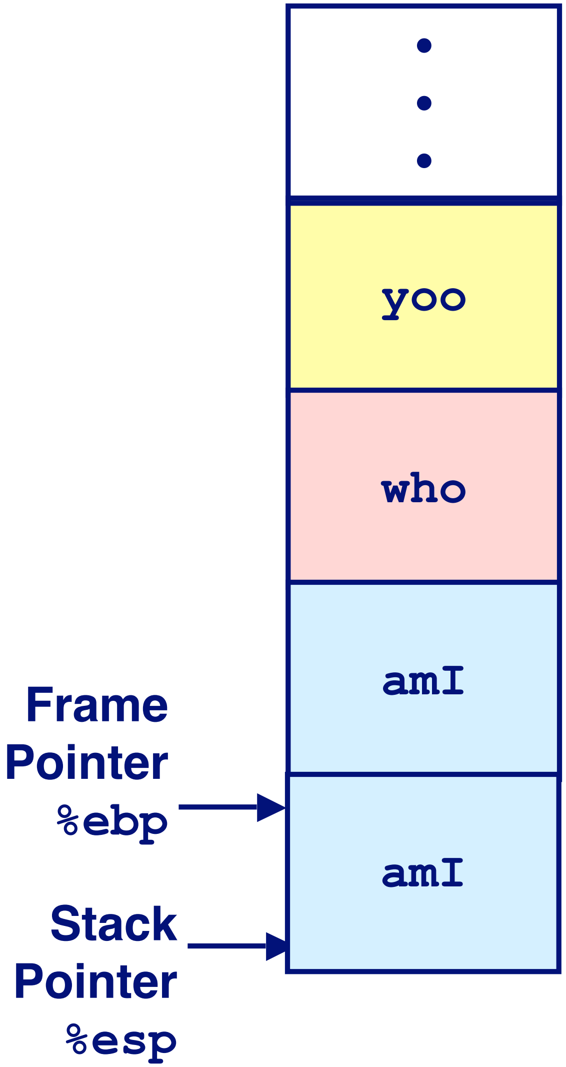


Stack Operation

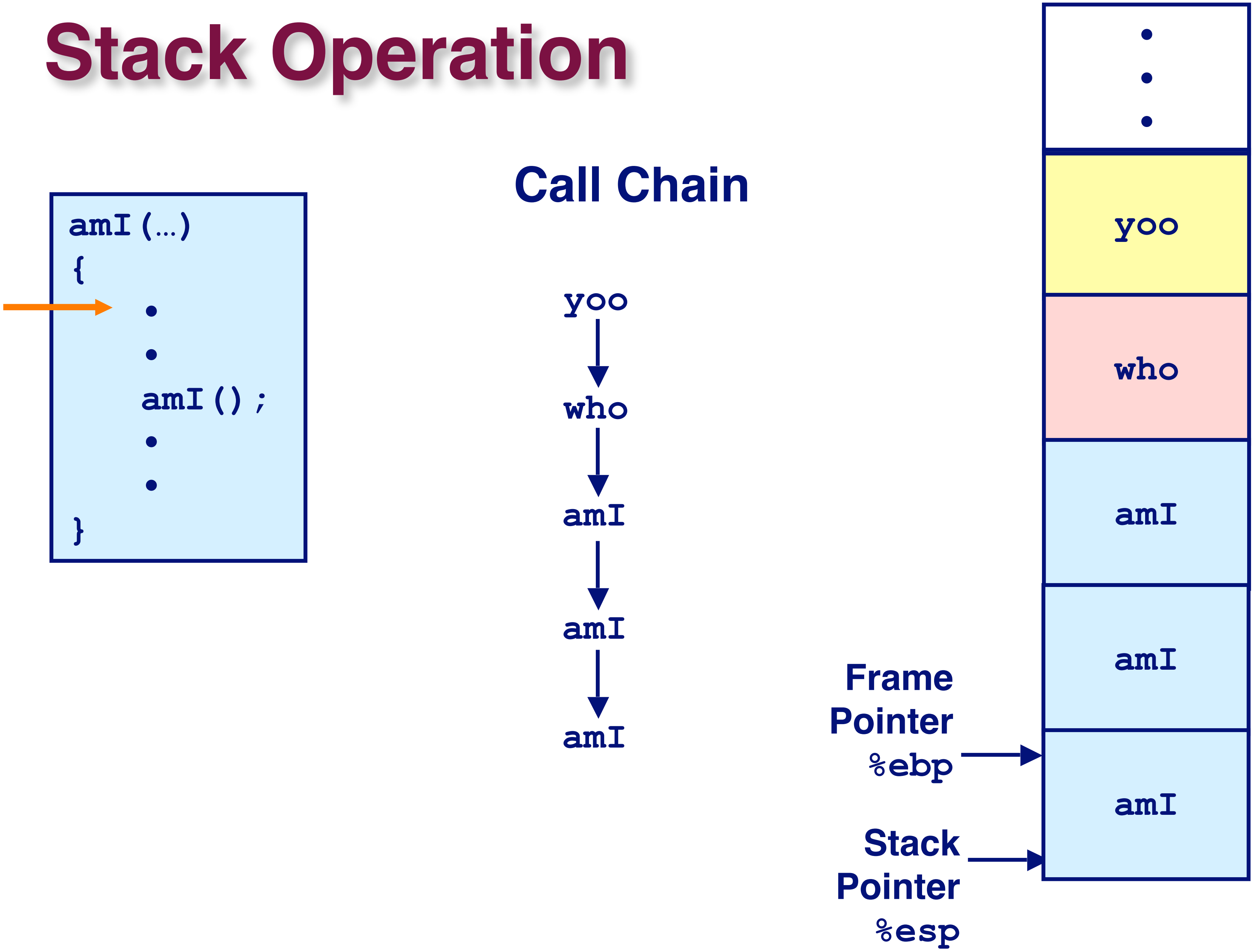
```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain





Stack Operation



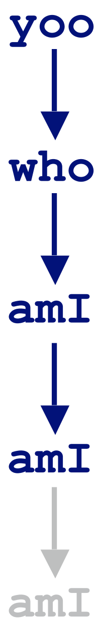
```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

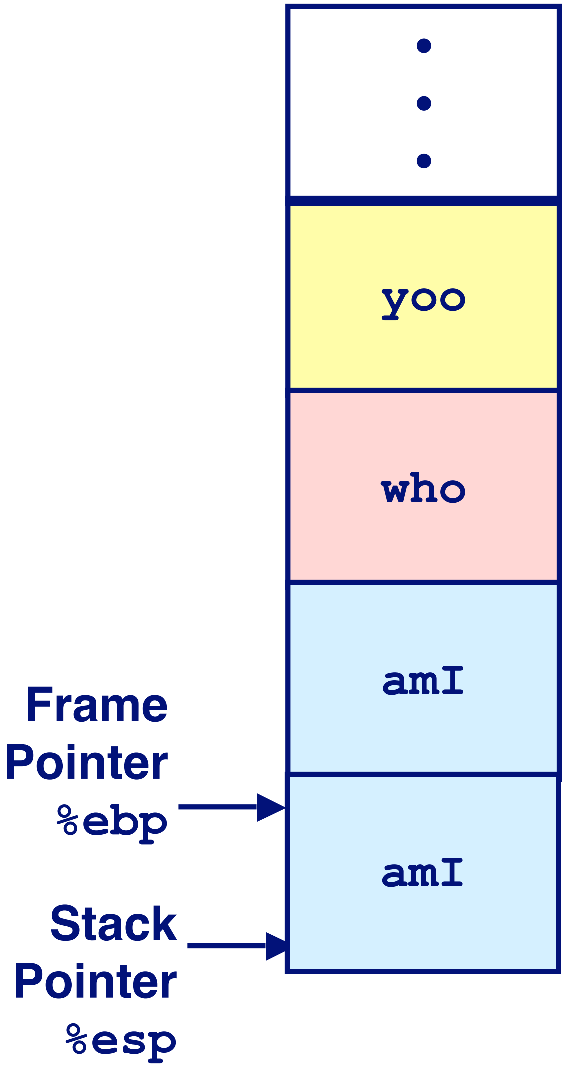
Call Chain

Stack Operation

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

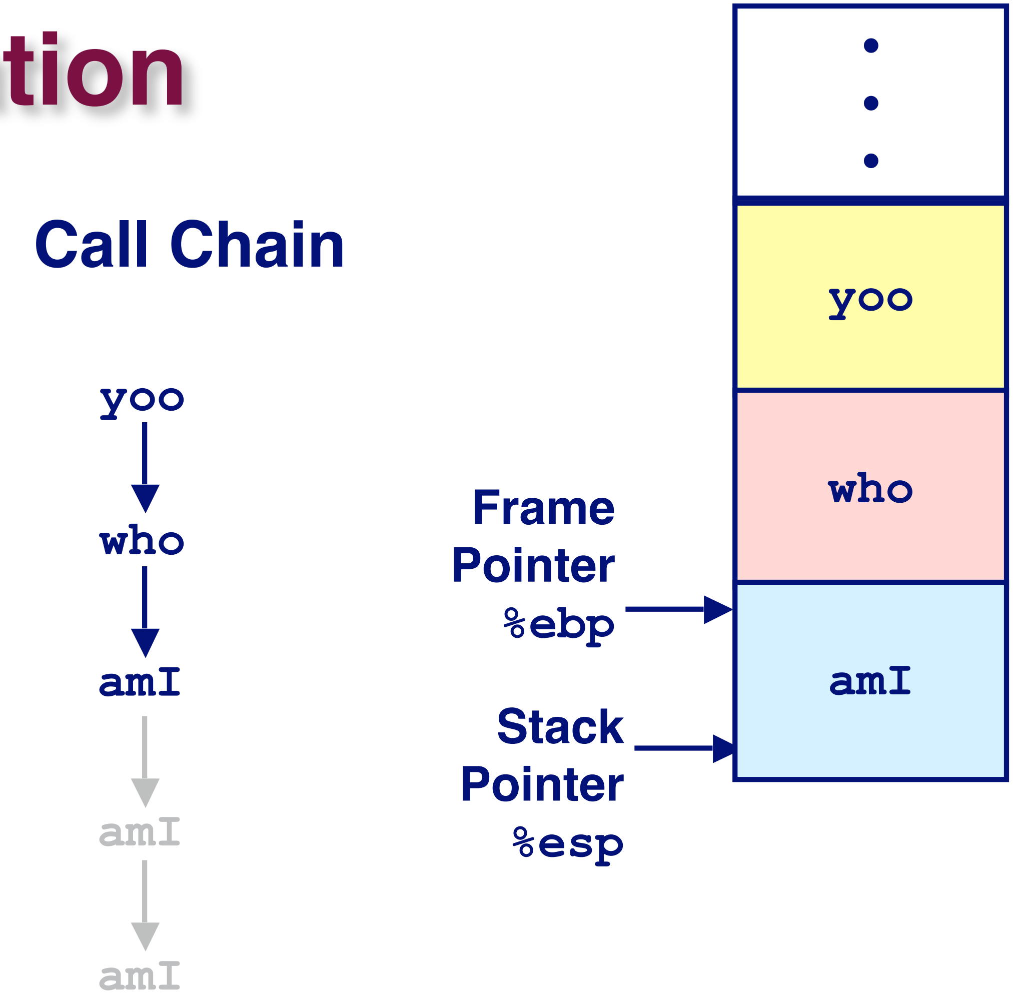
Call Chain





Stack Operation

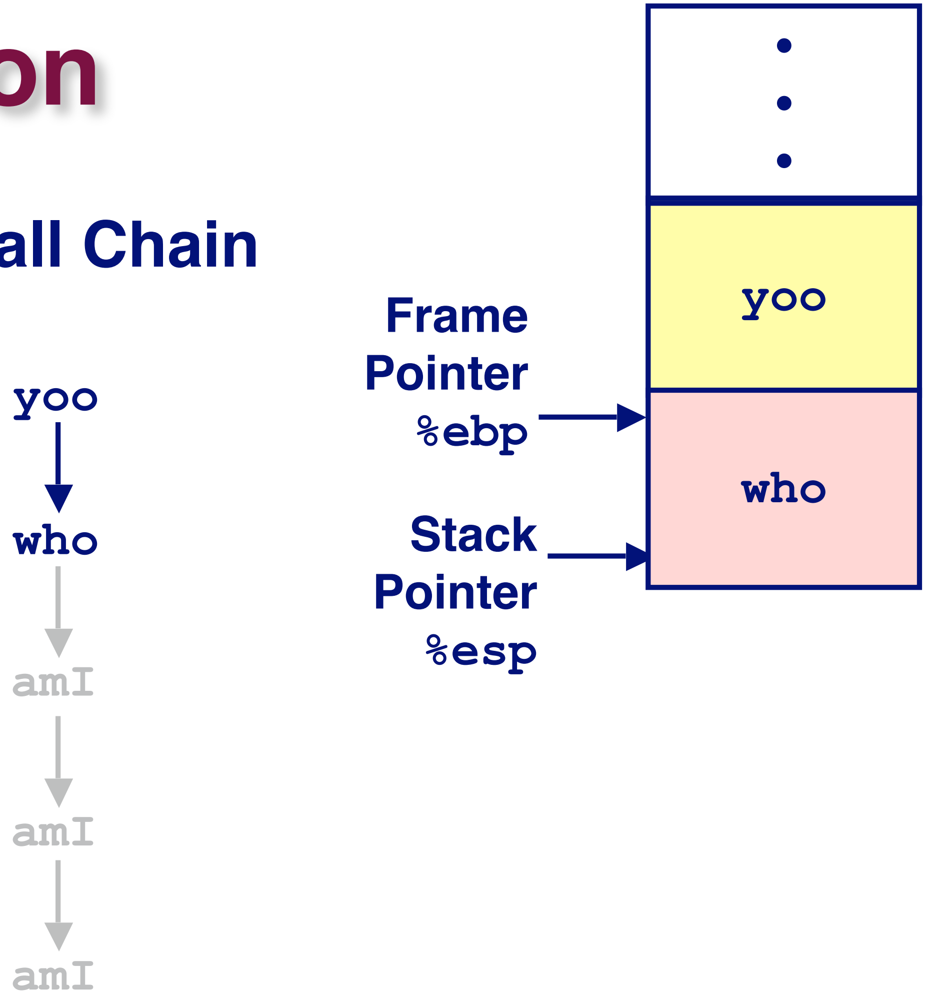
```
amI(…)
{
      •
      •
      amI();
      •
      •
}
```



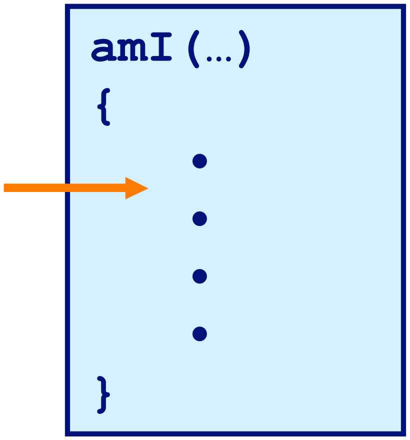
Stack Operation

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

Call Chain

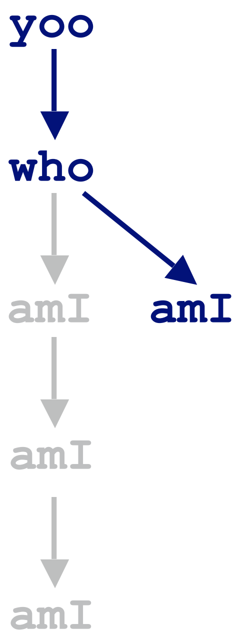


Stack Operation



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

Call Chain



yoo

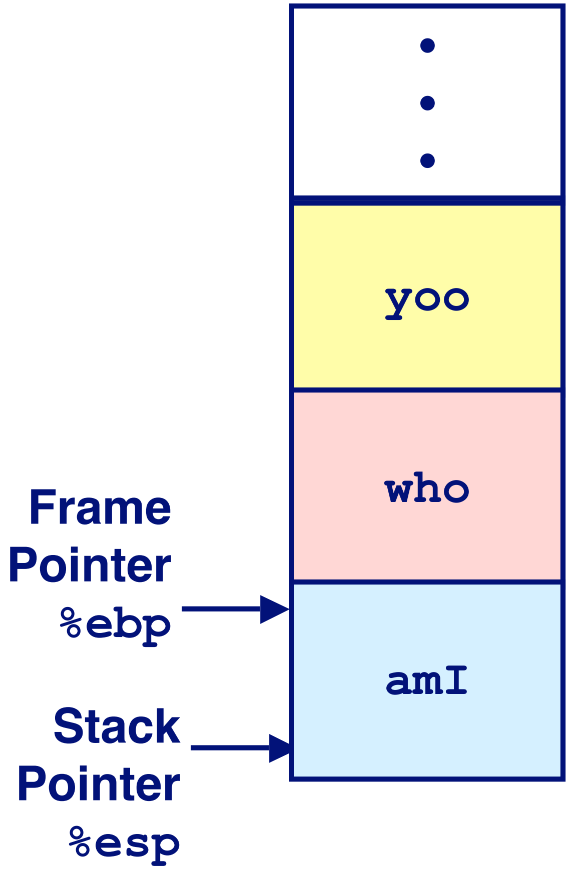
who

amI

amI

amI

amI



•
•
•

yoo

who

amI

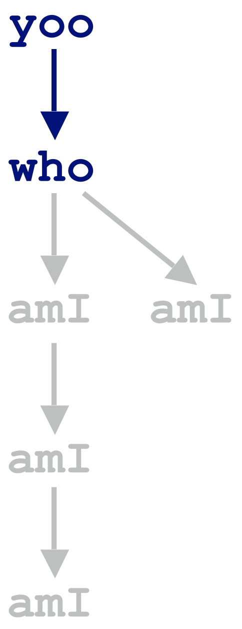
Frame Pointer %ebp

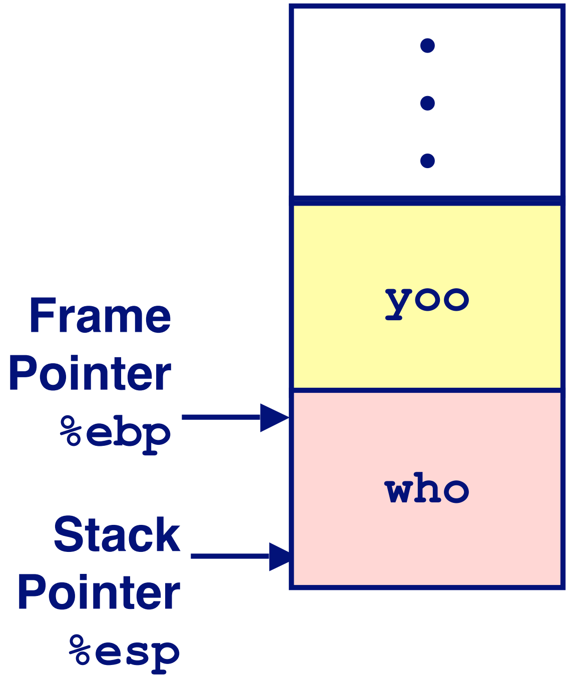
Stack Pointer %esp

Stack Operation

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

Call Chain

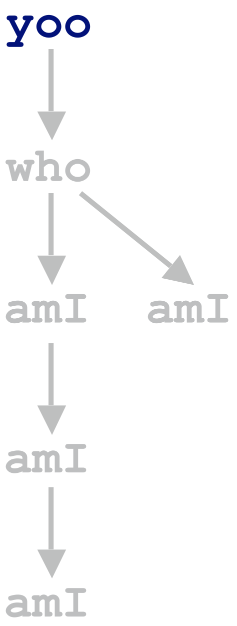


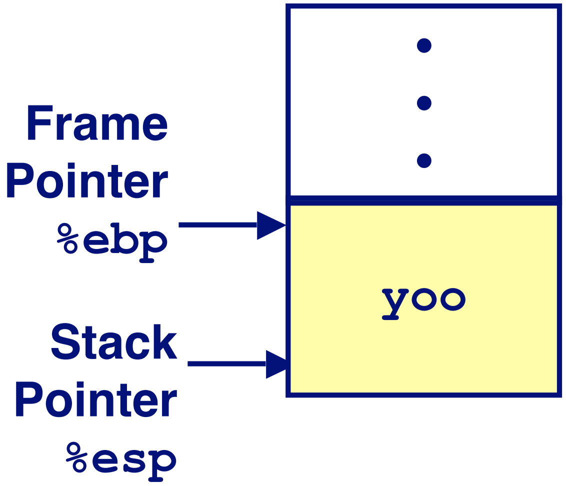


Stack Operation

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

Call Chain





Summary

Today

- Basic stack organization and access
- Activation records (stack frames)
- Call chains

Next time

- Detailed example of calls and stack state
- Register saving conventions
- Recursion