Chapter 3.7: Call Stacks and Procedures

Topics

- IA32 stack discipline
- Register saving conventions

Announcements

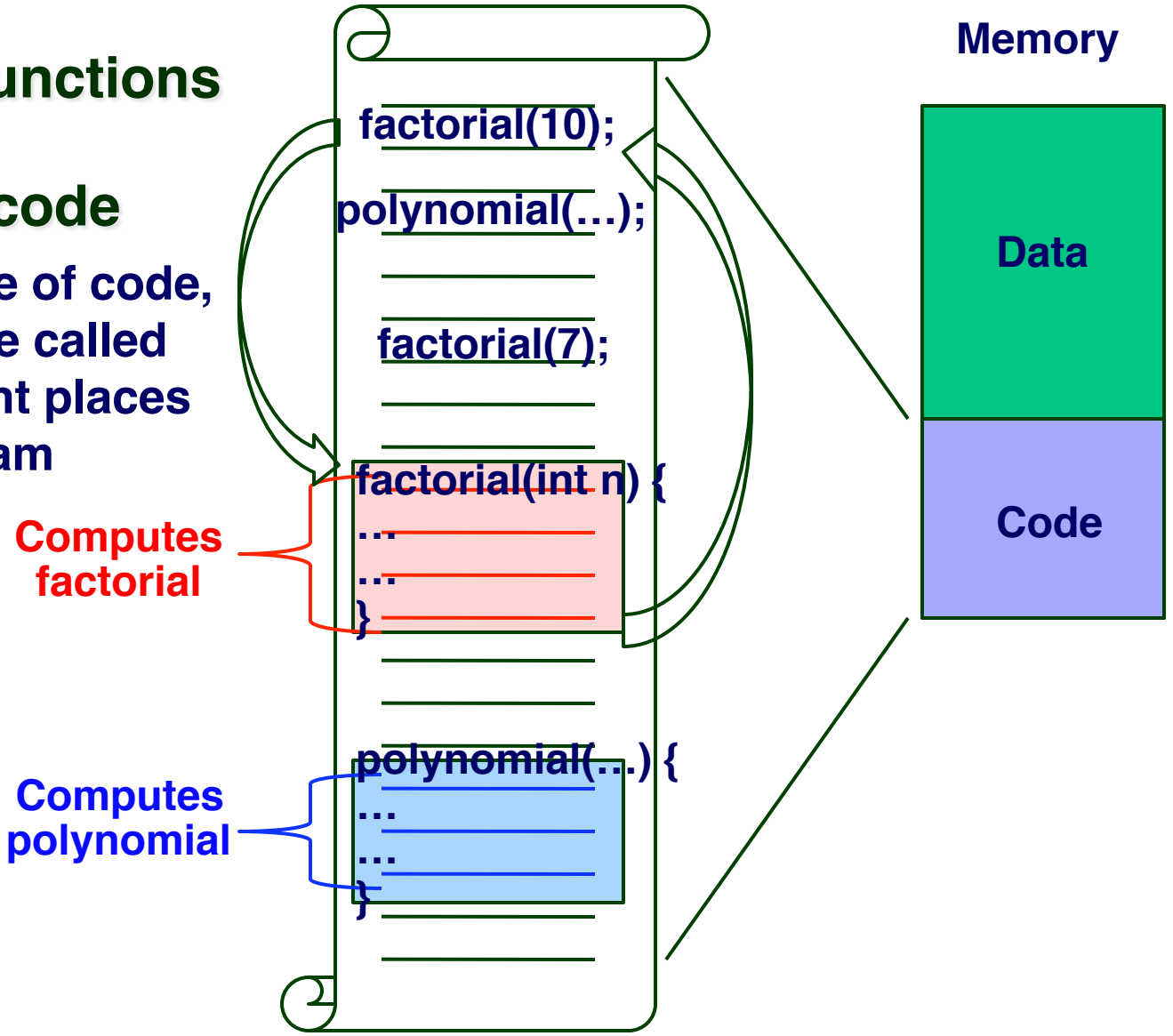
- Bomb Lab is due Friday Oct 3 by 11:55 pm
  - Extra credit secret bomb: add 7 points to your final lab grade
- First midterm is probably Tuesday Oct 7
- Recitation Exercises #2 released, due Monday Sept 29
  - Hand in at your recitation
- Essential that you read the textbook in detail & do the practice problems
  - Read Chapter 3.1-3.14, skip 3.12 for now

Recap…

- How loops are implemented in assembly
  - Do-while implemented with cmpl and conditional jumps
- Other loops are rewritten as Do-while
  - While loops
    - More efficient jump-to-middle implementations
  - For loops
- switch statement
  - Can be implemented as a jump table or a series of if-then-else conditional jumps
  - Jump table is faster to jump but may be a memory hog for sparse switch ranges

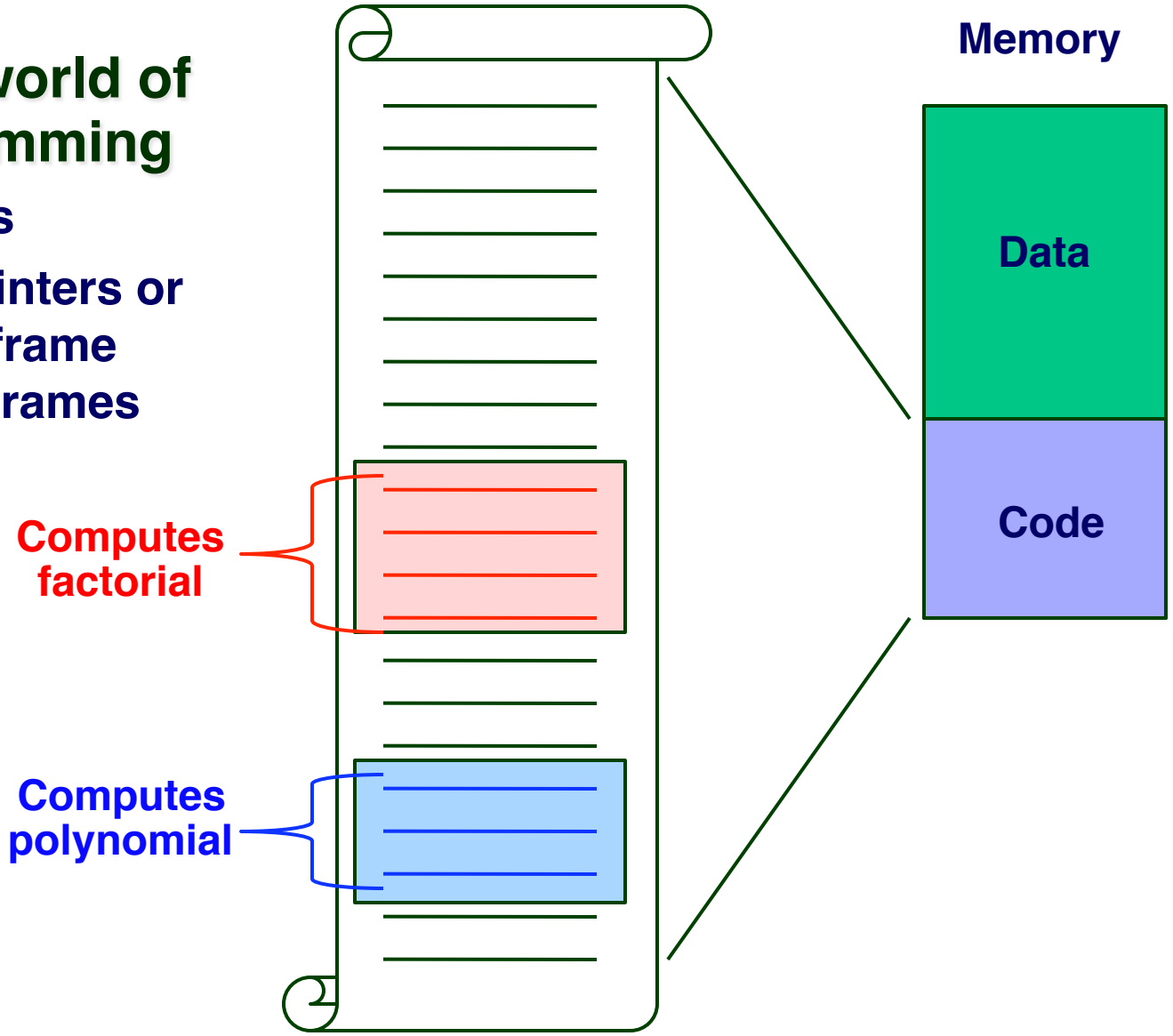
Procedures/Function Calls

- **Procedures/functions allow logical grouping of code**
  - Enable reuse of code, which can be called from different places in the program



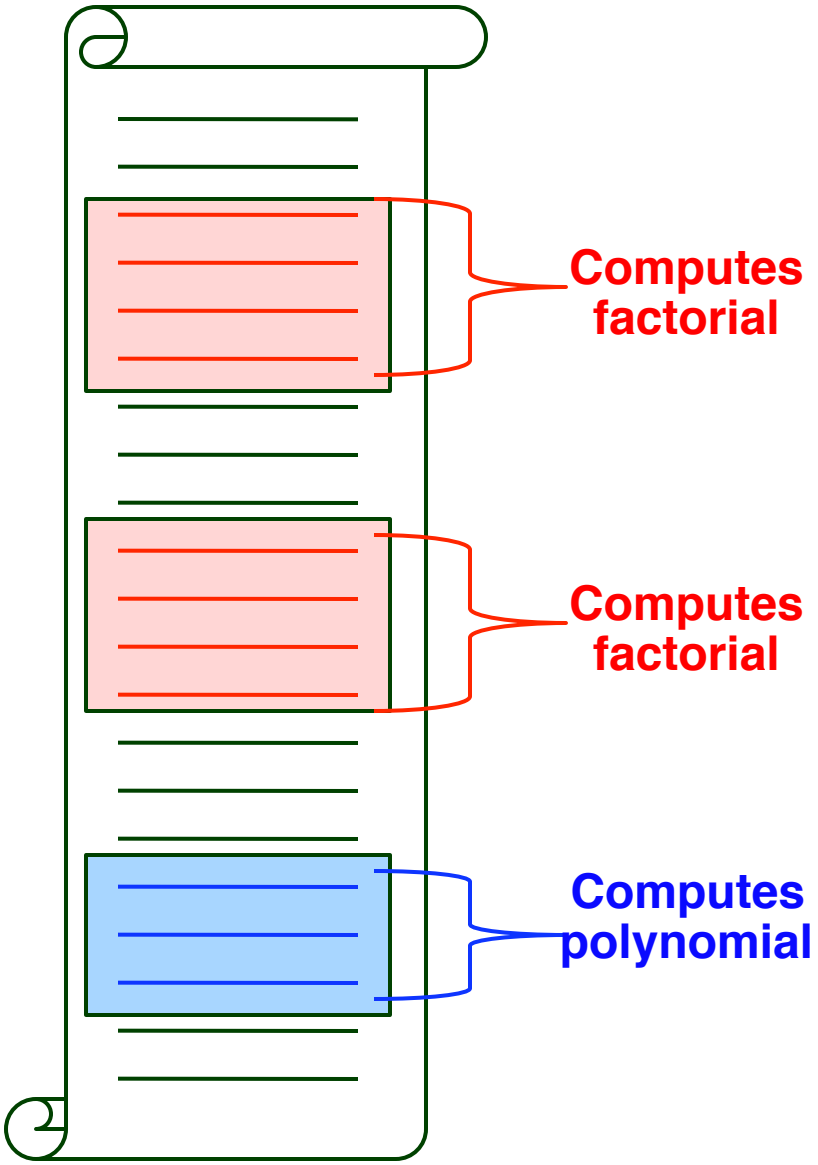
Motivation for a Call Stack (1)

- Imagine the world of early programming
  - No functions
  - No stack pointers or stacks. No frame pointers or frames



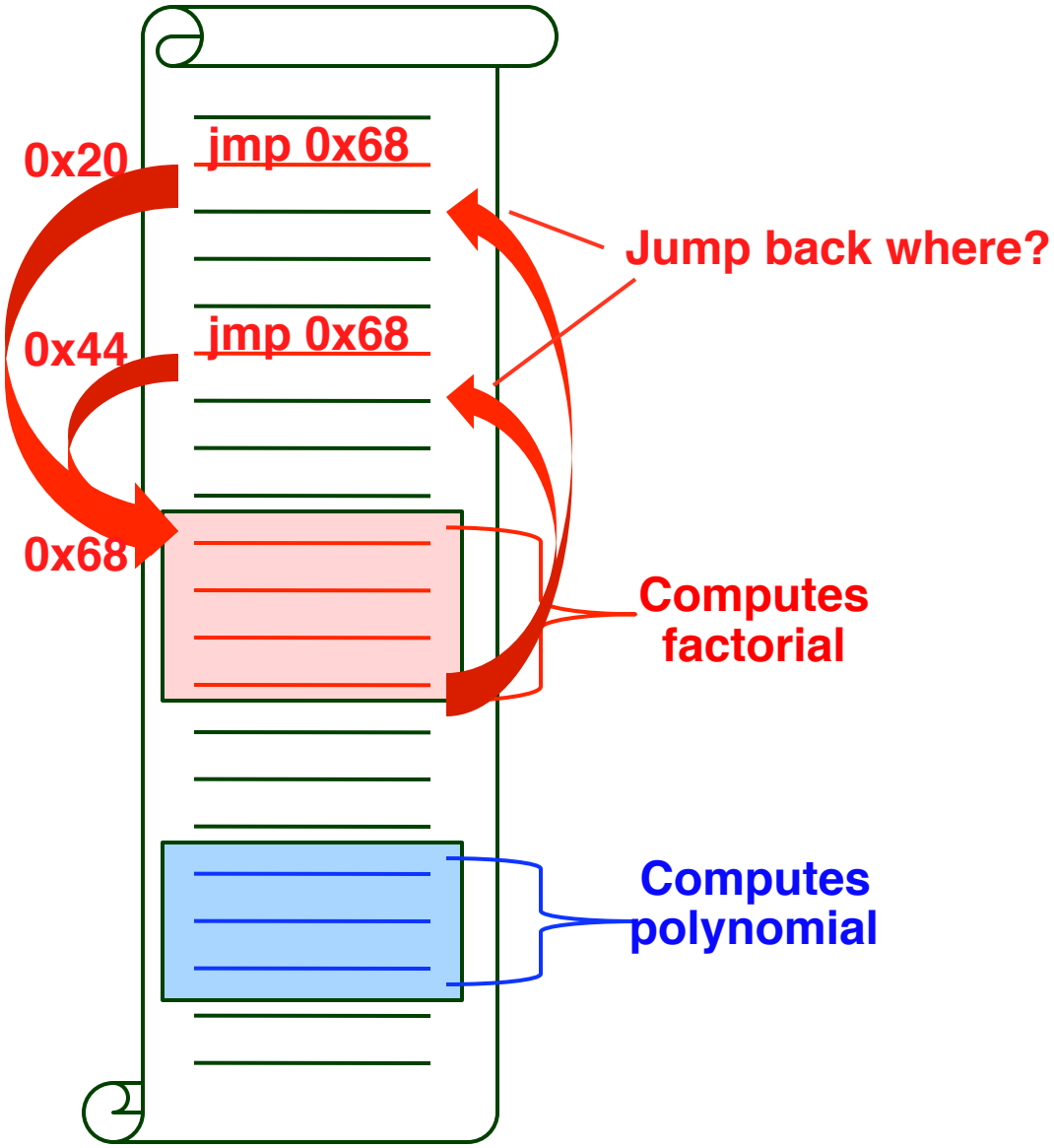
Motivation for a Call Stack (2)

- Suppose 2 parts of code want to execute the same computation
  - Repeating the computational code is inefficient
  - Maintaining multiple copies of same computation is inconvenient



Motivation for a Call Stack (3)

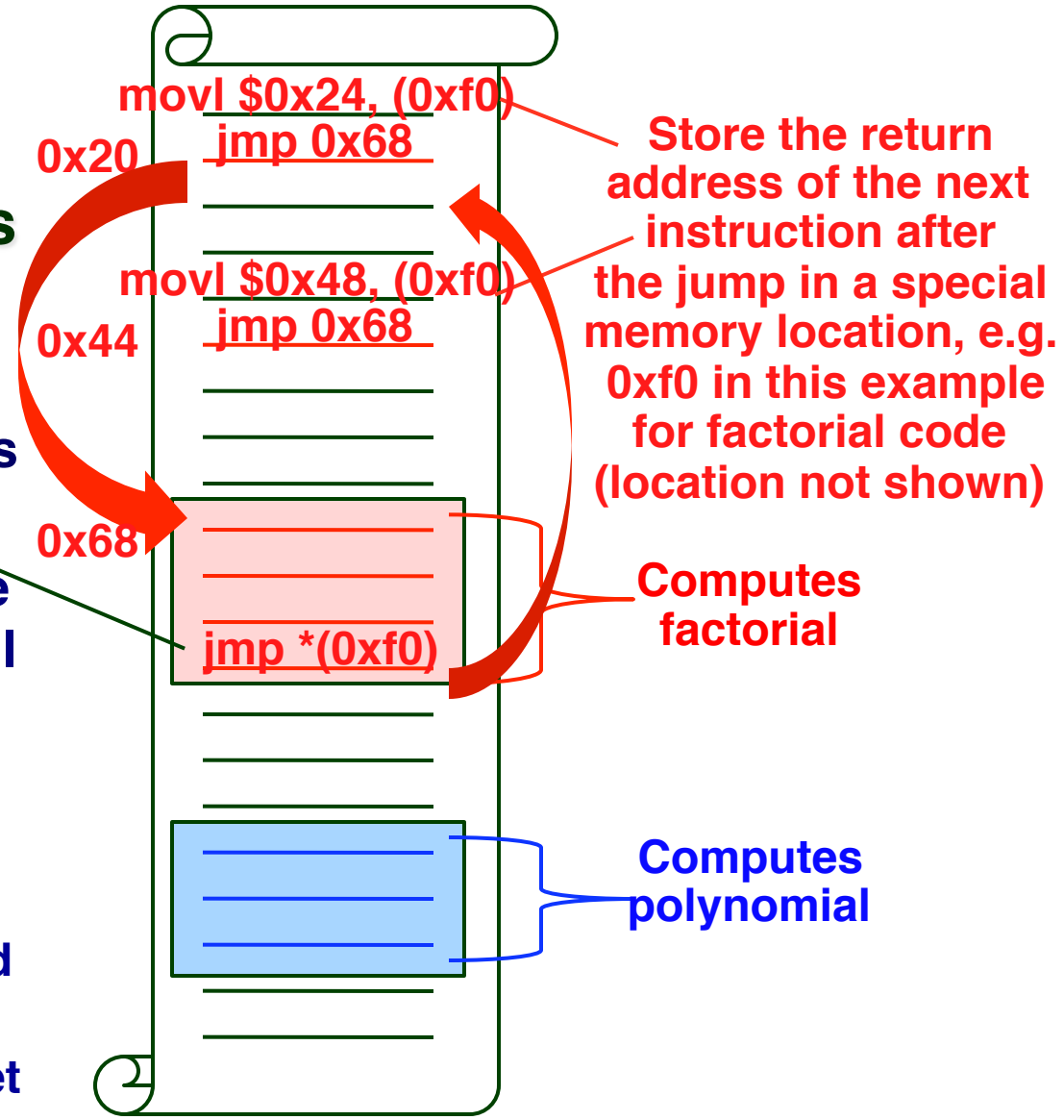
- Instead, write the common code once and jump to it from different locations
  - Problem #1: How do you jump back?
  - Must remember the address where you were calling the common code from
  - Could allocate some memory to store the return address for this factorial code



Motivation for a Call Stack (4)

- Before each jump to the common code, store return address in special memory location
  - Jump back using this special location
  - Problem: can't share 1 mem location for all return addresses
    - 1 set of common code may call another set
    - Instead, would need 1 return address location for each set of common code

Problem: Recursion?



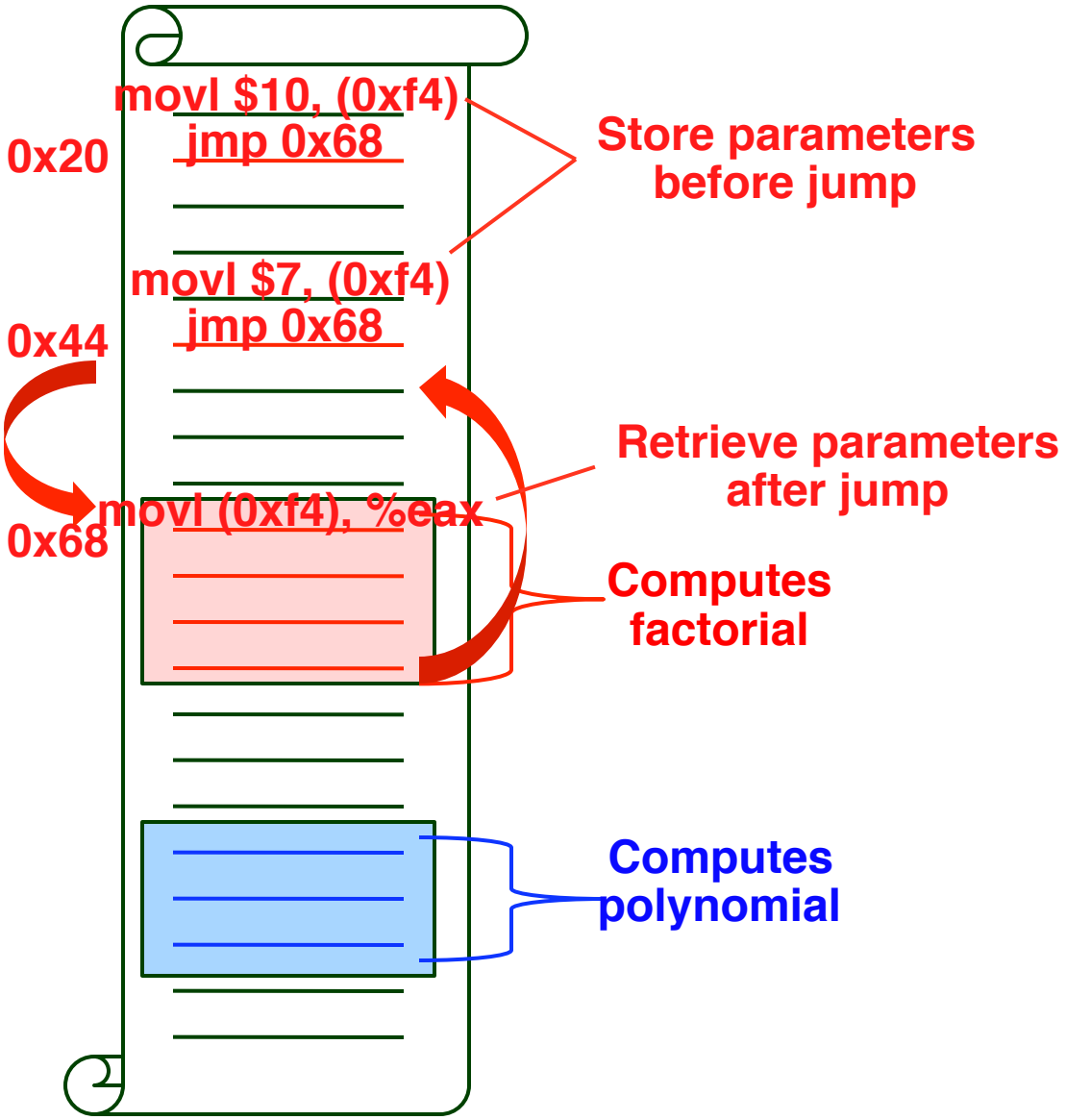
Motivation for a Call Stack (5)

- **Problem #2: How to pass parameters to common code?**
  - Example: from the 1st jump point, may want to calculate 10!, while 2nd jump point may want to calculate 7!
  - Again, could allocate special memory to store parameters before jumping

Problem: Waste Memory

  - The called common code is programmed to look in this special memory location to retrieve parameters

Problem: Recur-sion?



Motivation for a Call Stack (6)

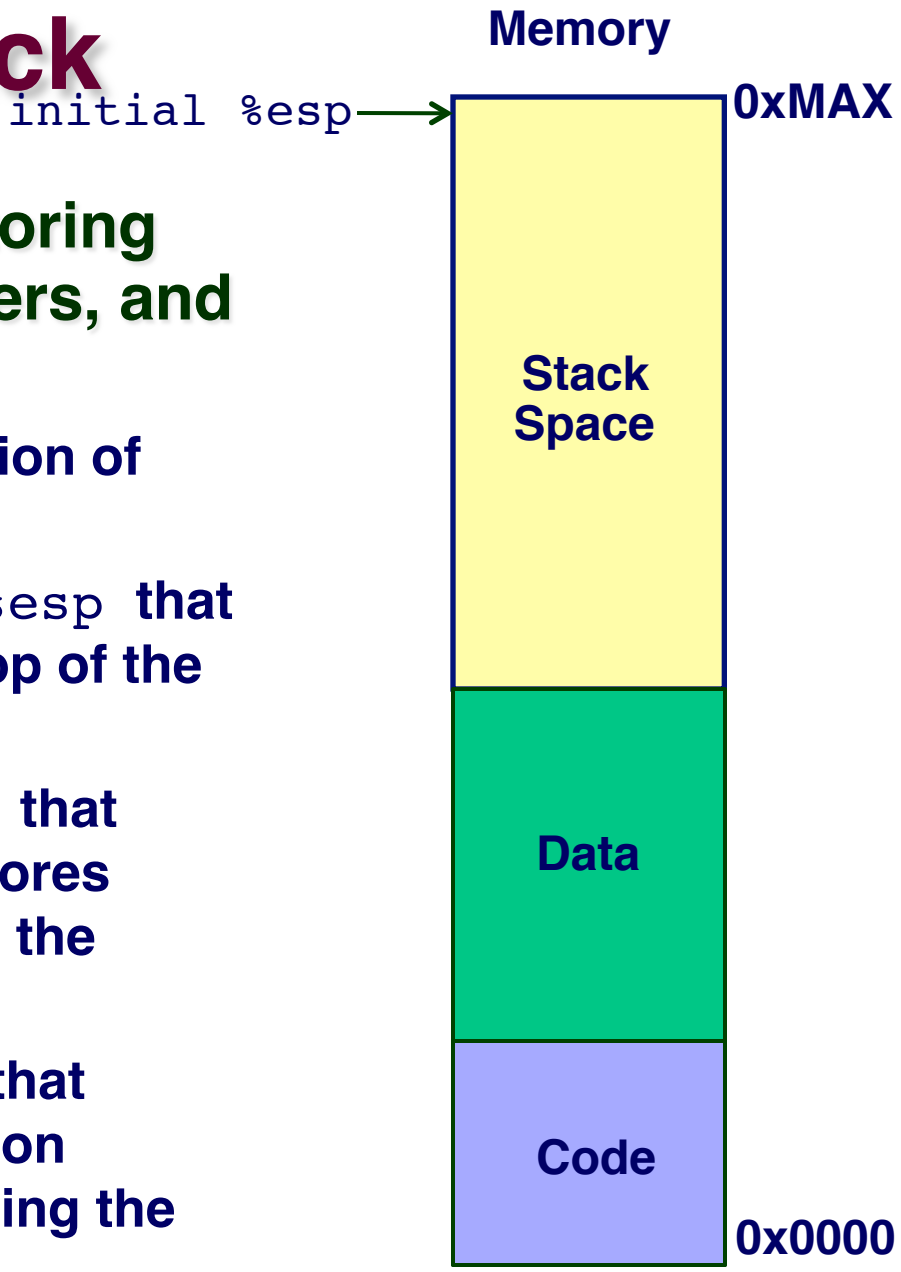
- **Problem #3: Wasting memory**
  - the common code may never be called, wasting allocated memory for parameters and return addresses
- **Problem #4: Wasting memory II**
  - the common code may allocate local variables only used within the common code, yet the common code may never be called, wasting memory

Motivation for a Call Stack (7)

- **Design goals: want a solution to support functions that:**
  1. **Automatically stores the return address before a function is called, and retrieves that return address when exiting a function**
  2. **Allows storage of parameters before a function is called, and parameter retrieval within the function**
  3. **Is memory-efficient for return addresses, parameters, and local variables**
  4. **Each instance of a function should have its own unique set of {return address, parameters, and local variables}, to support recursion**

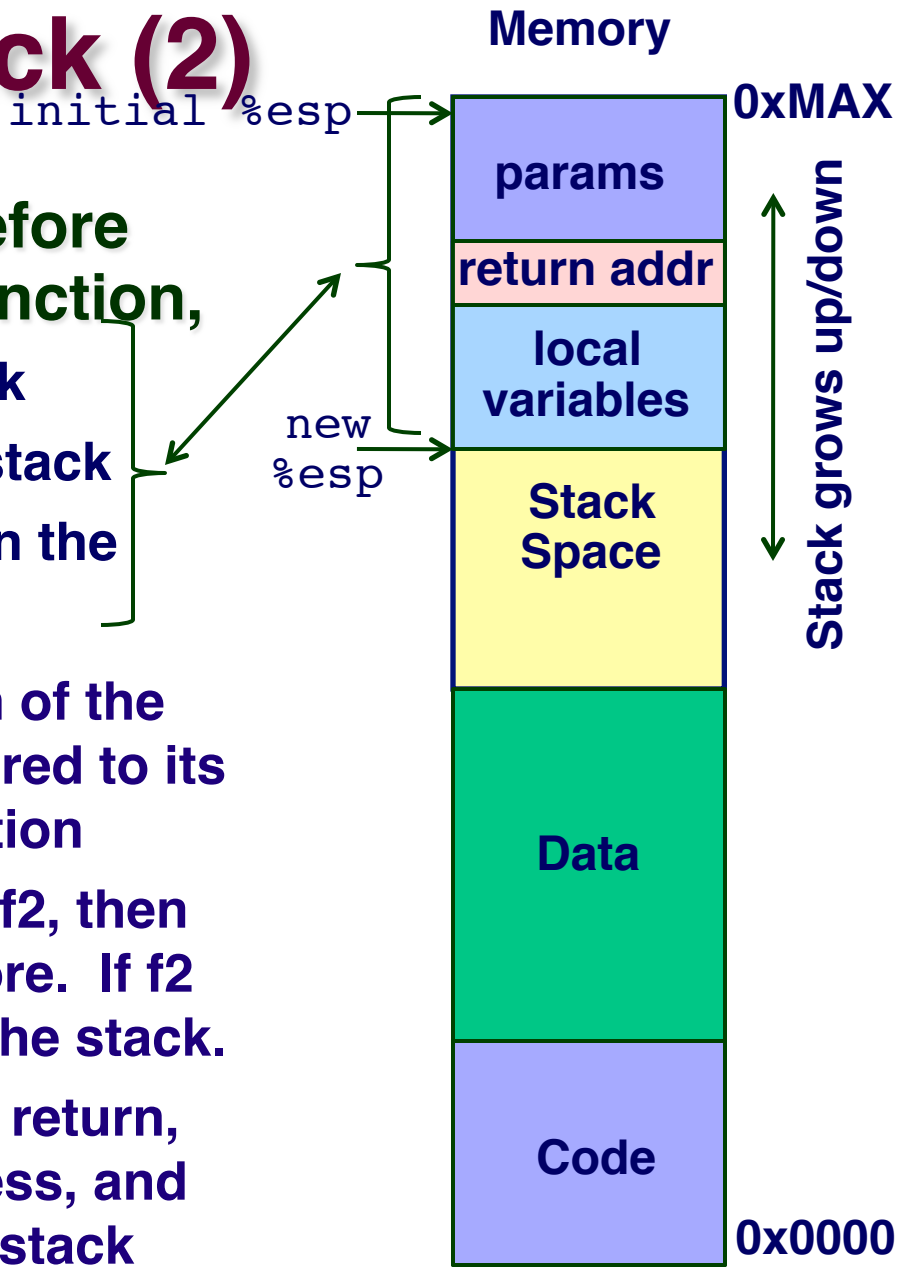
Solution: a Call Stack

- **Supports function calls by storing the return address, parameters, and local variables on a *stack***
  - **Allocate a special reusable region of memory for the stack**
  - **Create a special CPU register `%esp` that the stores the location of the top of the stack, i.e. the *stack pointer***
  - **Create a CPU instruction `call` that before jumping to a function stores (pushes) the return address on the stack**
  - **Create a CPU instruction `ret` that jumps to the return address upon exiting from the function, popping the address from the stack**



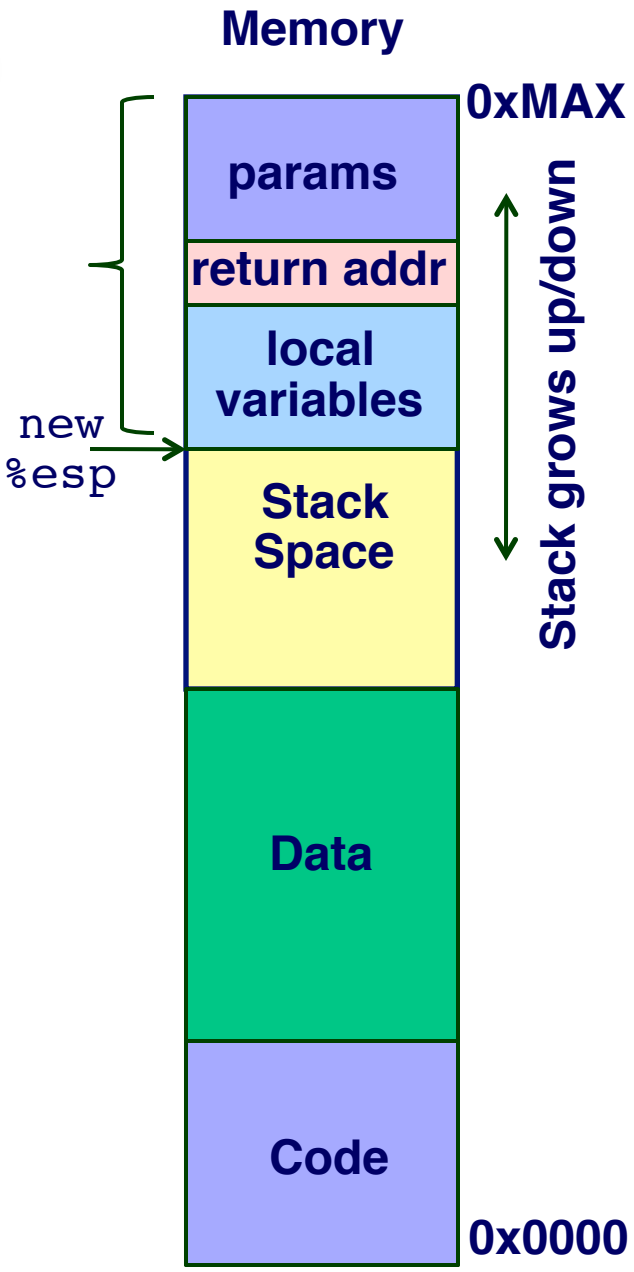
Solution: a Call Stack (2)

- On each function call, and before executing the body of the function,
  - Push parameters onto the stack
  - Push return address onto the stack
  - Push/allocate local variables on the stack
  - Must also save the old location of the stack pointer so it can be restored to its old value after exiting the function
  - If a function f1 calls a function f2, then the stack grows down even more. If f2 calls a function f3, that grows the stack.
  - As each function finishes/calls return, its local variables, return address, and parameters are popped off the stack (deallocated), shrinking the stack



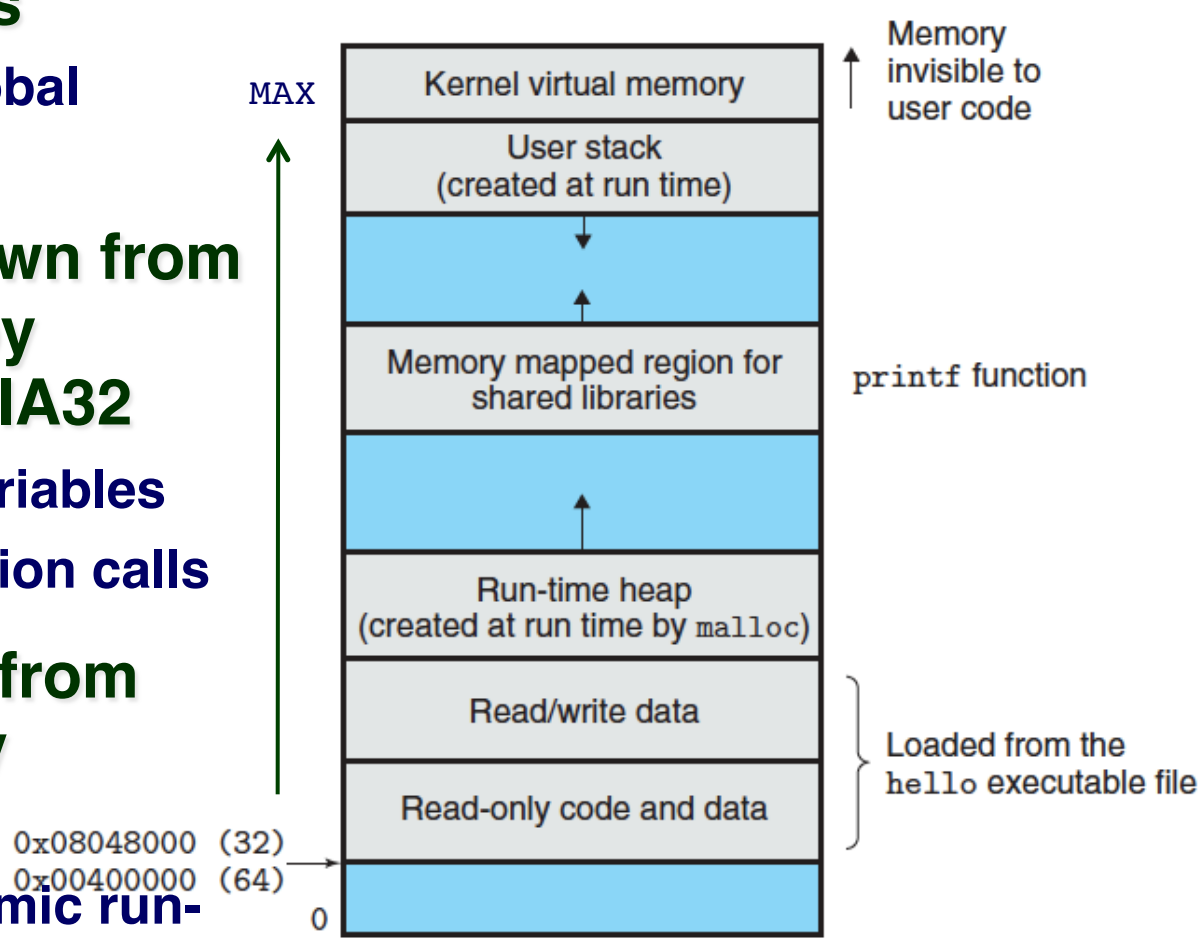
Solution: a Call Stack (3)

- In this way, we satisfy the original design goals:
  1. Enables passing parameters to functions
  2. Automated support for storing the return address
  3. Memory-efficient storage that reuses memory and only allocates space for parameters, return addresses, and local variables if a function is called
  4. Supports recursion – each instance/ call of a function has its own stack frames to store its own return address, parameters, and local variables



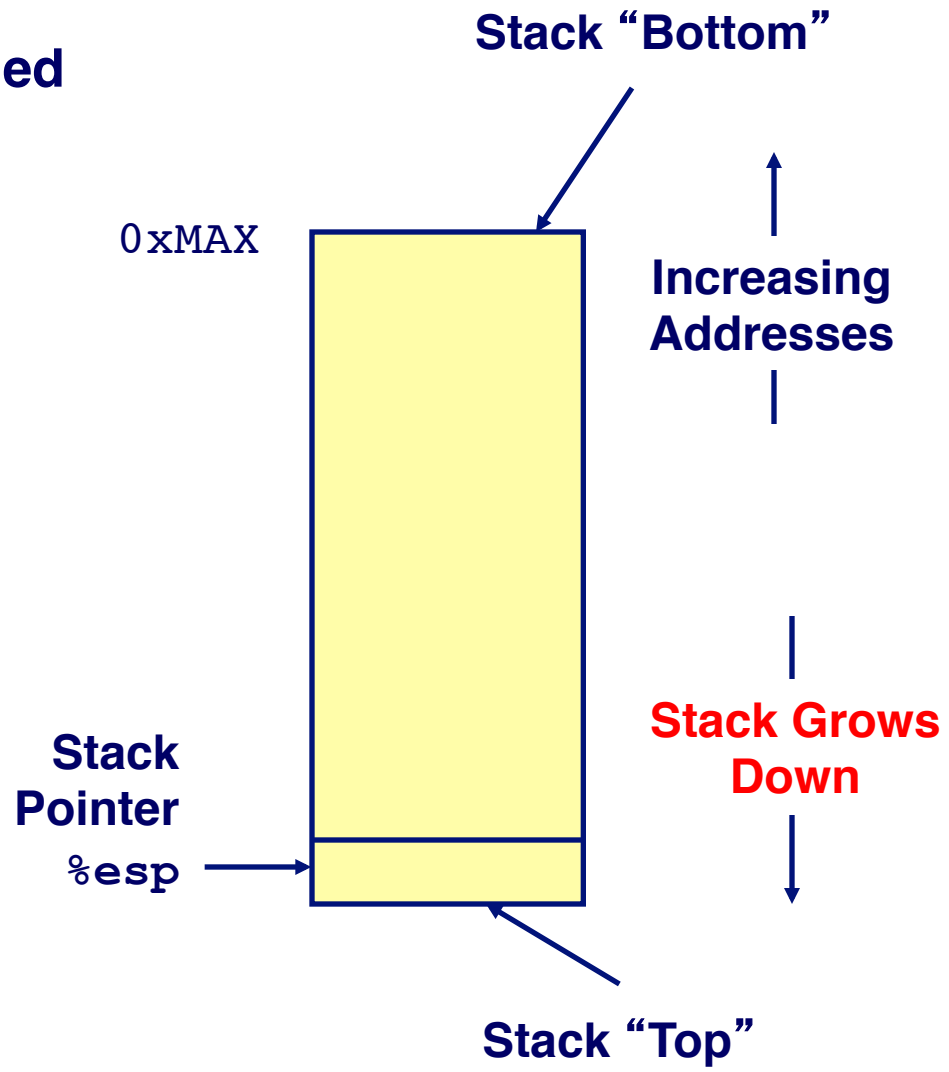
Layout of a Program in Memory

- Code, data, heap and stack segments
  - Data stores global variables
- Stack grows down from high memory by convention on IA32
  - Stores local variables
  - Supports function calls
- Heap grows up from low memory by convention
  - Supports dynamic run-time memory allocation like malloc()



IA32 Stack

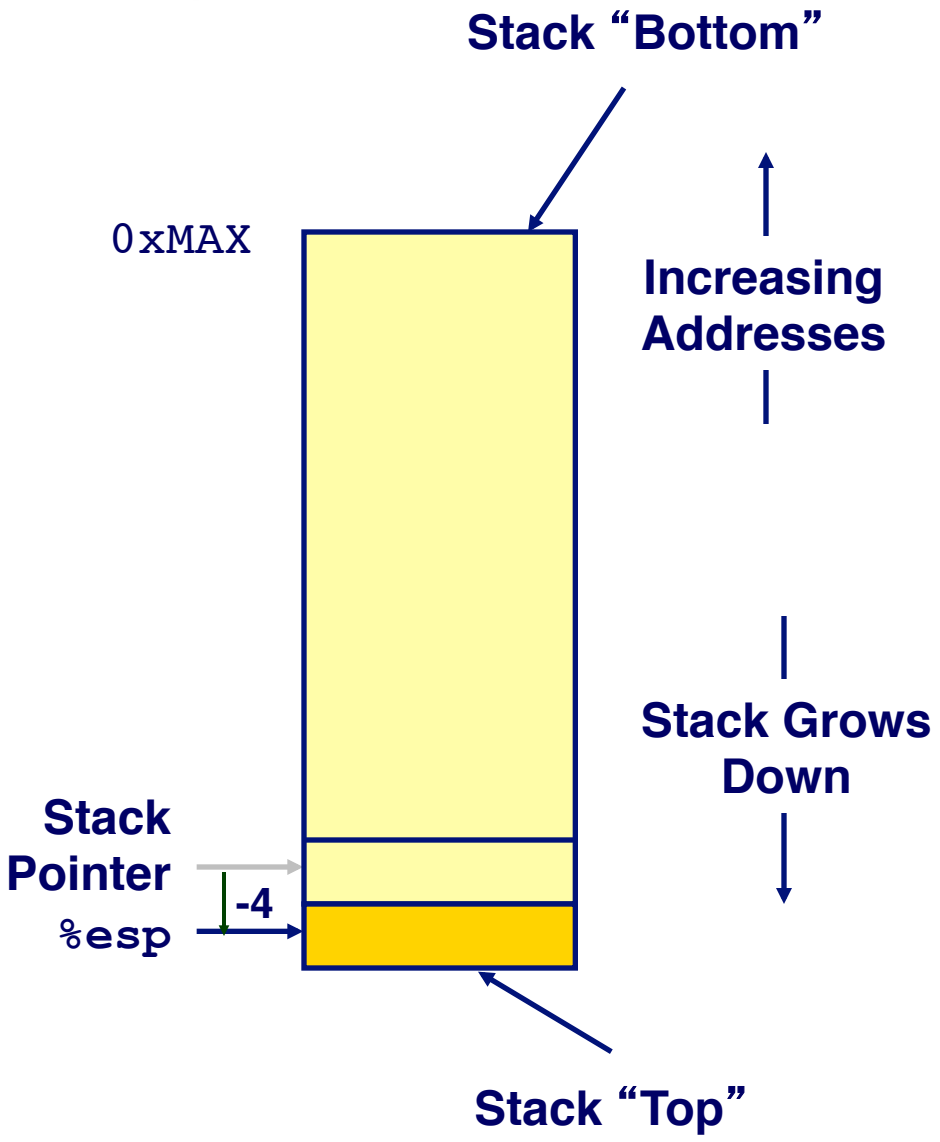
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
  - address of top element



IA32 Stack Pushing

Pushing

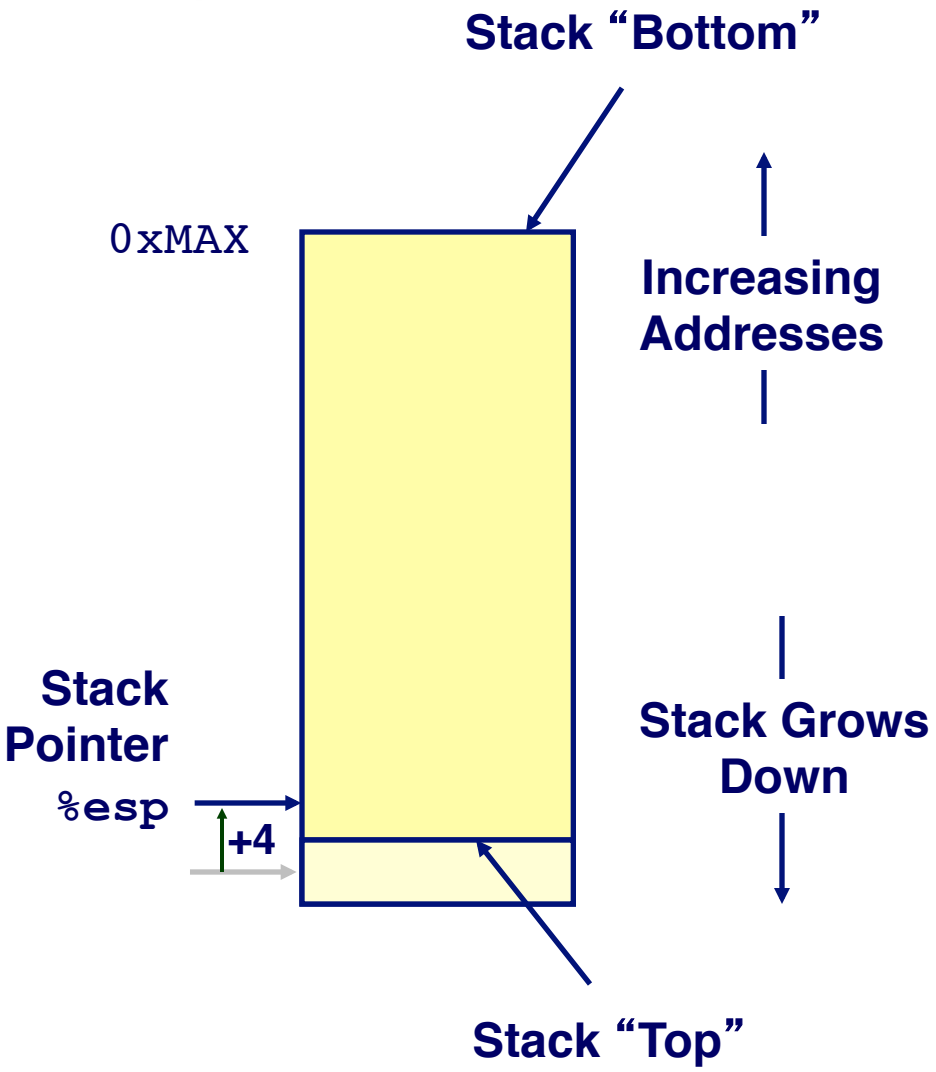
- `pushl` *Src*
- Fetch operand at *Src*
- Decrement `%esp` by 4
- Write operand at address given by `%esp`



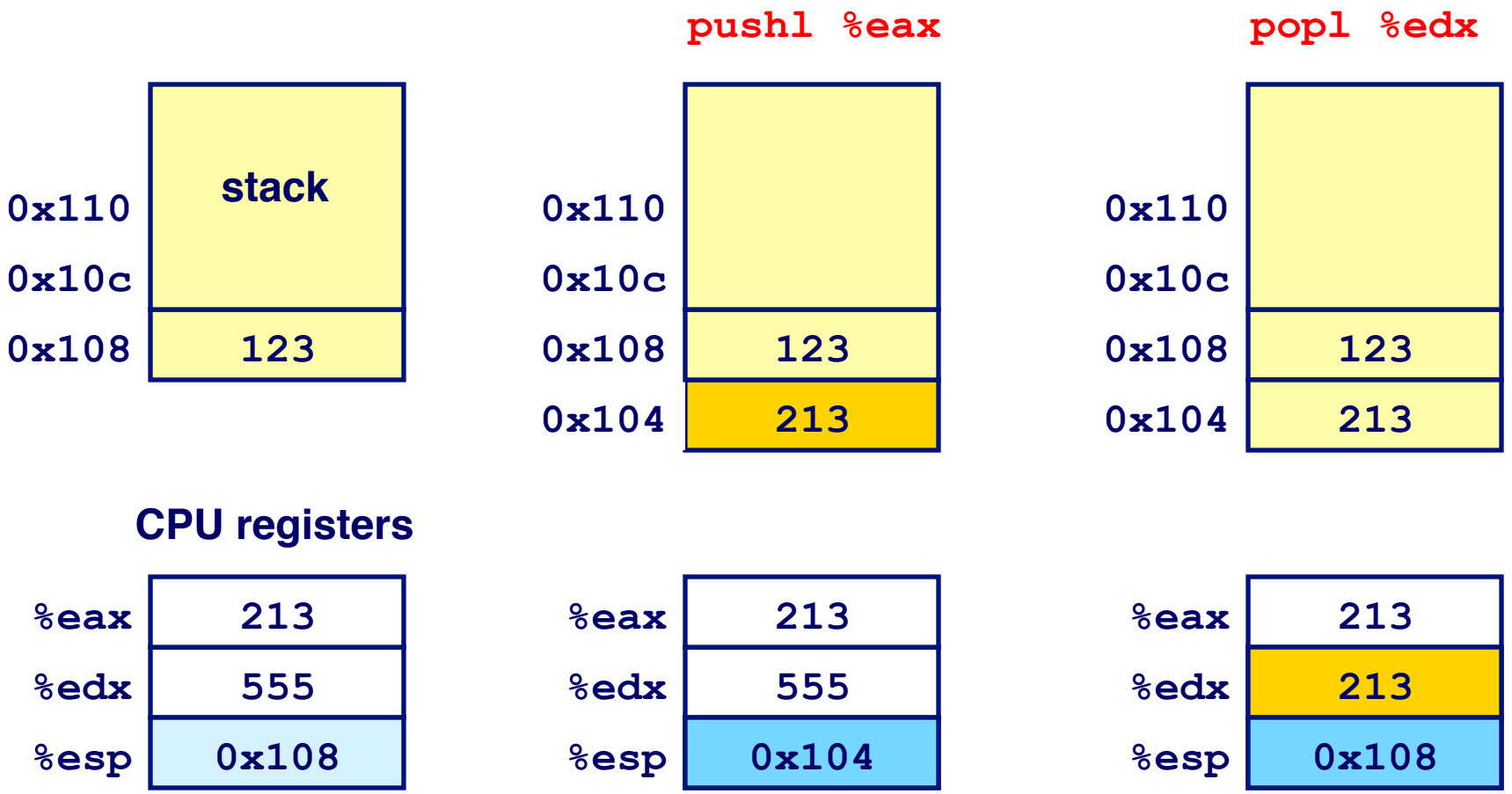
IA32 Stack Popping

Popping

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



Stack Operation Examples



Procedure Control Flow

- Use stack to support procedure call and return

Procedure call:

`call` *`label`* Push return address on stack; Jump to *`label`*

- Example: if my function name is `swap_numbers`, then
  - `call swap_numbers`

Return address value

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

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

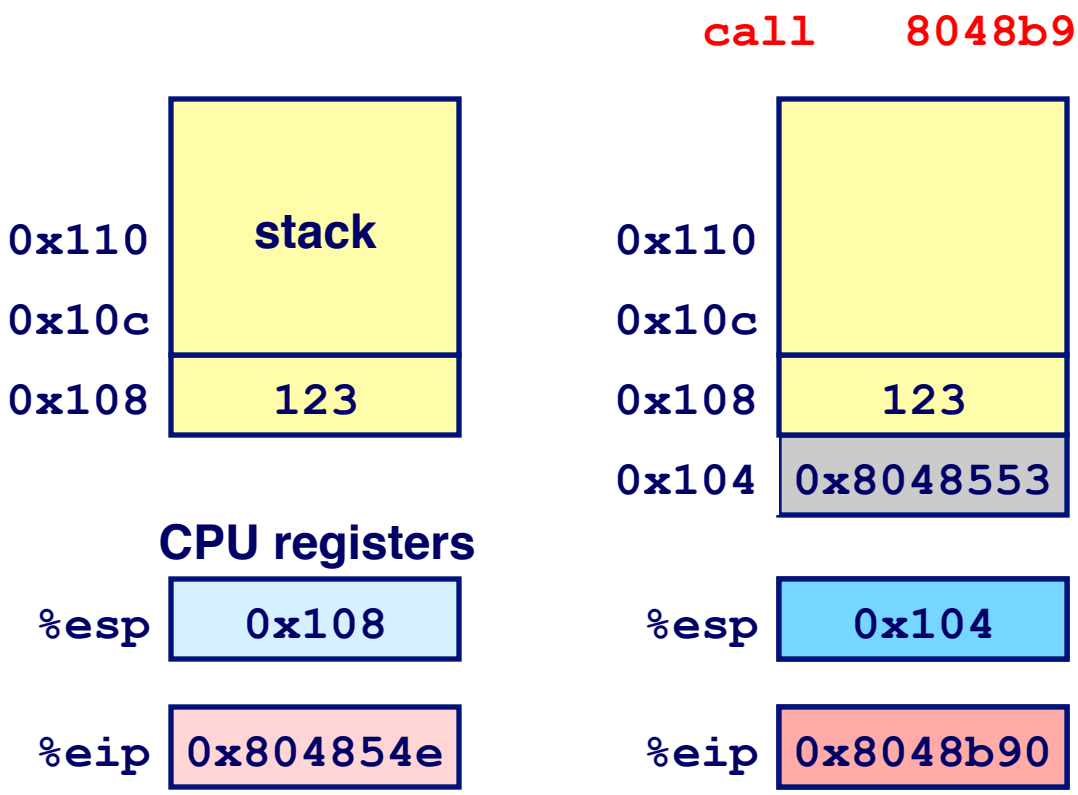
  - Return address = `0x8048553`

Procedure return:

- `ret` Pop address from stack; Jump to address

Procedure Call Example

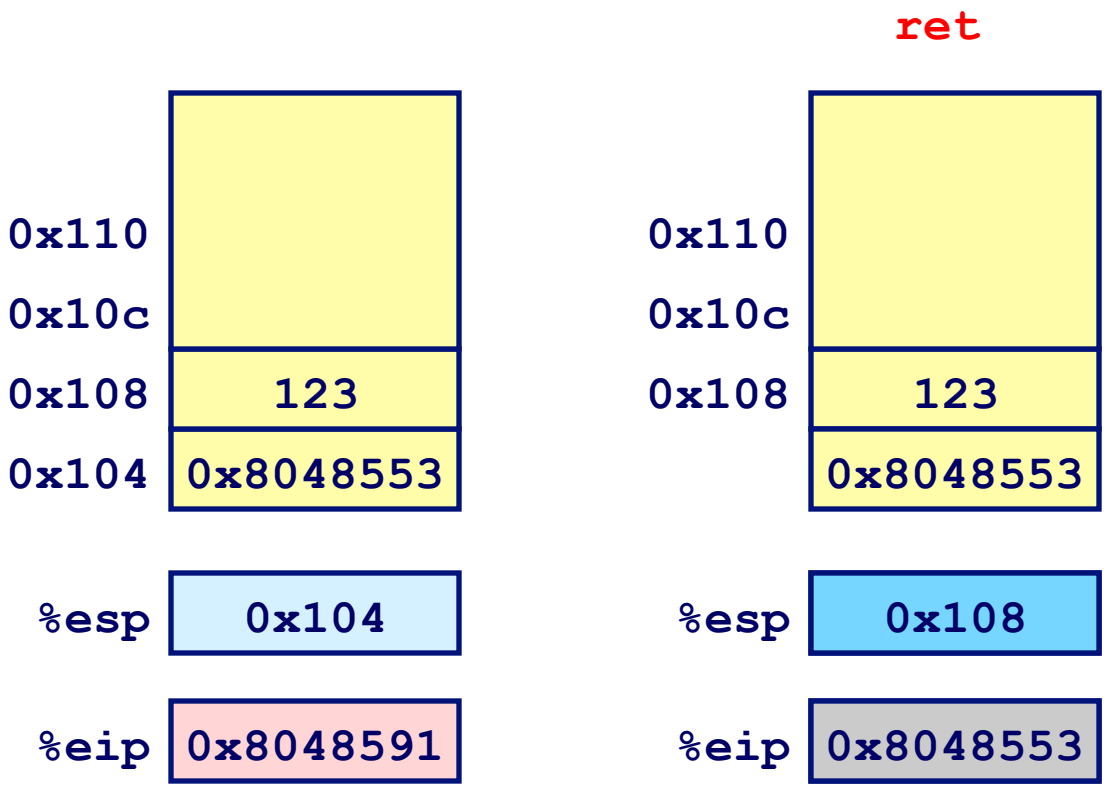
```
804854e:  e8 3d 06 00 00        call   8048b90 <main>
8048553:  50                    pushl  %eax
```



`%eip` is program counter

Procedure Return Example

```
8048591:   c3                    ret
```



`%eip` is program counter

Call Chain Example

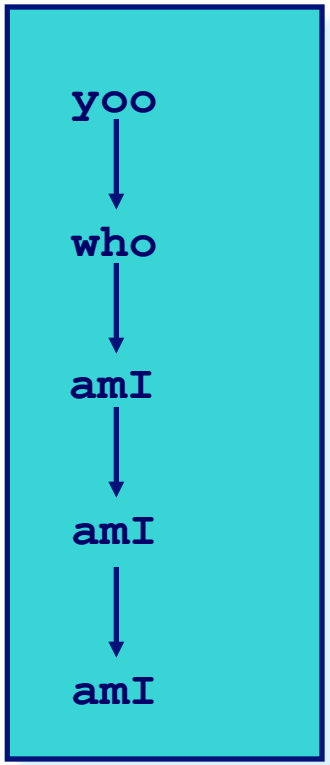
Code Structure

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

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

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

Call Chain

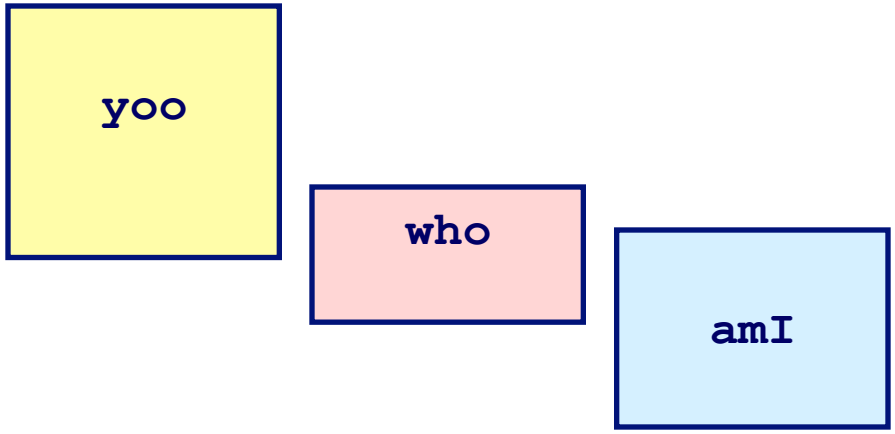


- Procedure `amI` recursive

Stack Frames

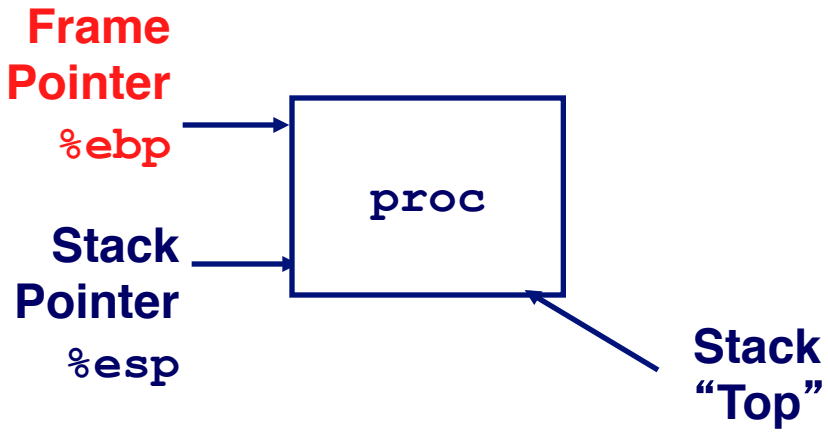
Contents

- Local variables
- Return information
- Temporary space



Management

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



Pointers

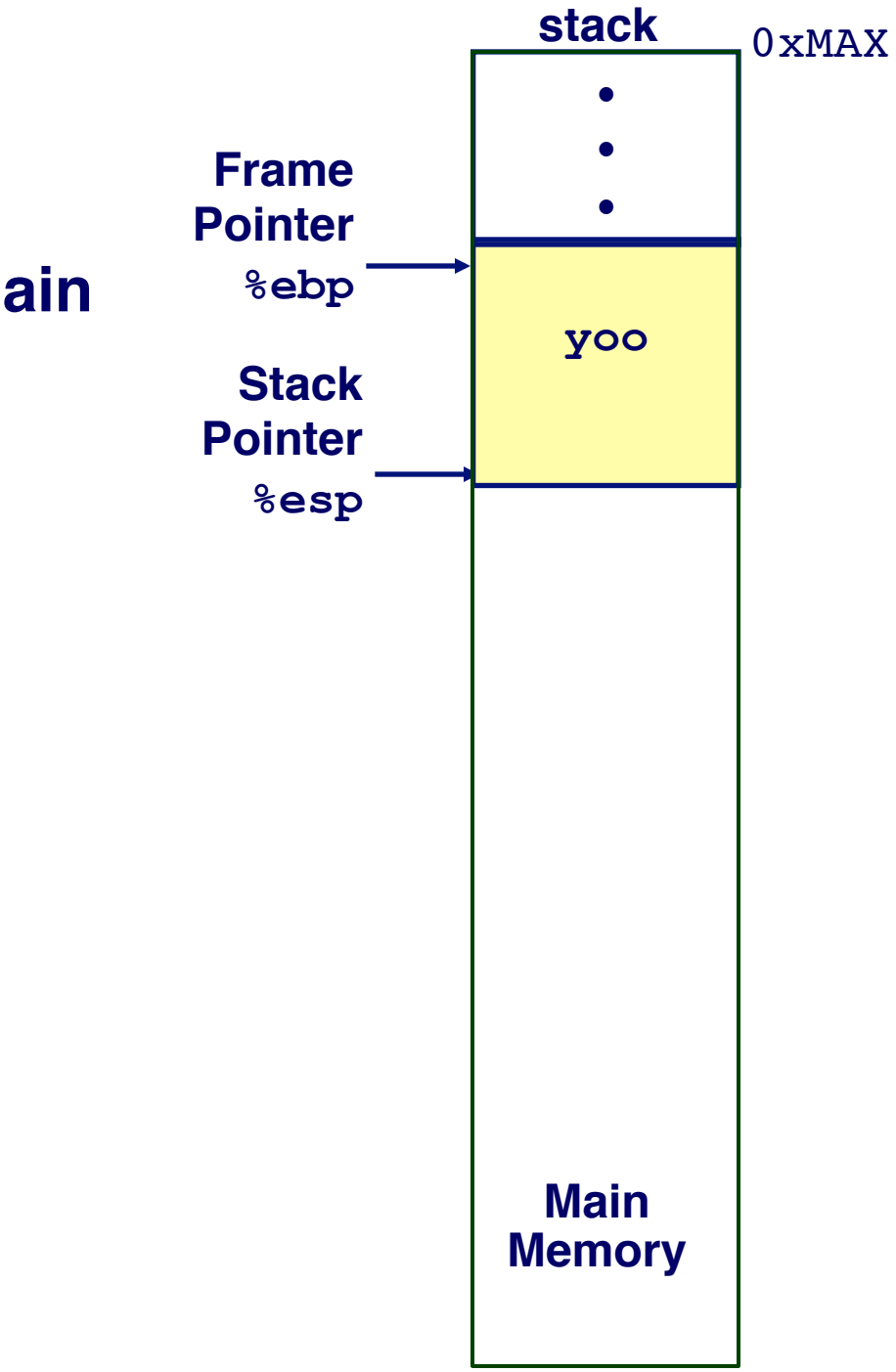
- Stack pointer `%esp` indicates stack top
- Frame pointer `%ebp` indicates start of current frame

Stack Operation

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

Call Chain

yoo



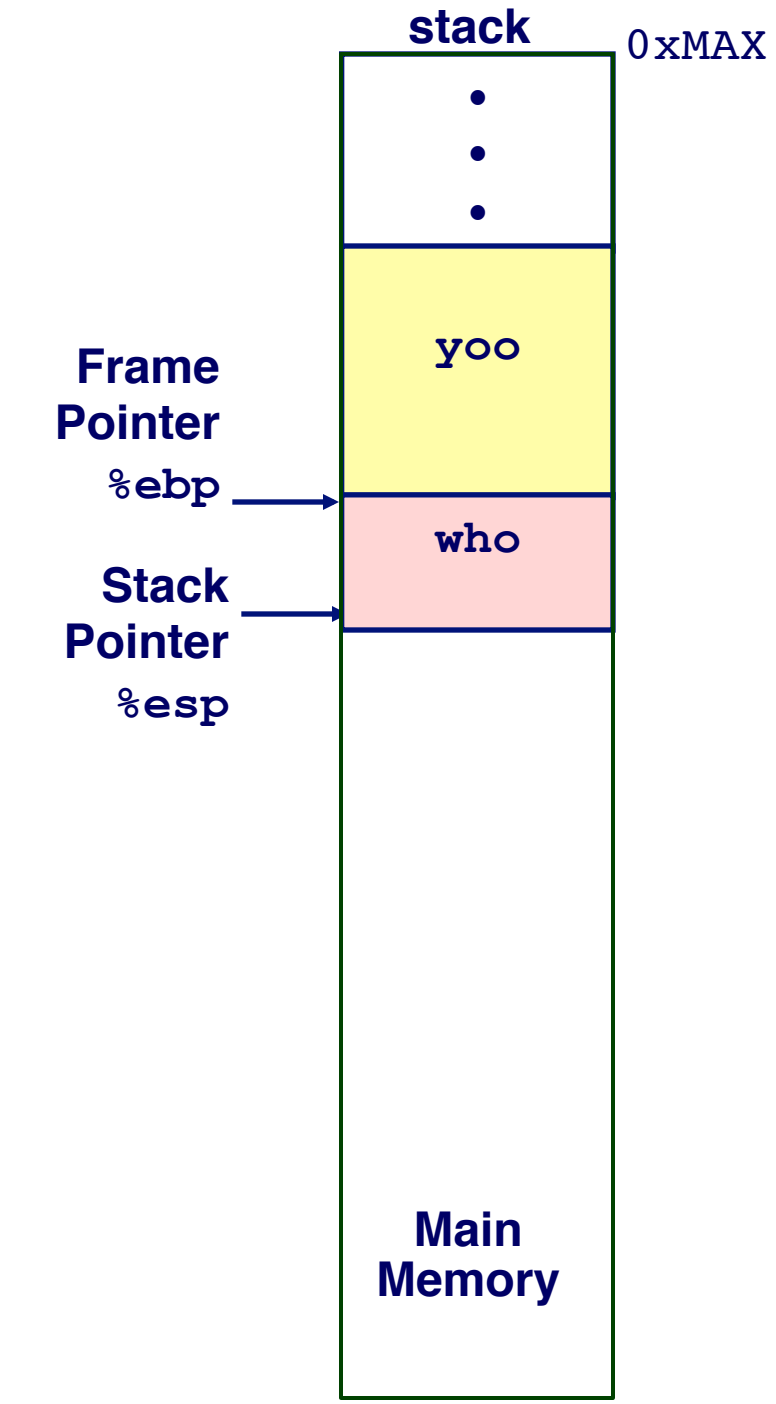
Stack Operation

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

Call Chain

yoo

who

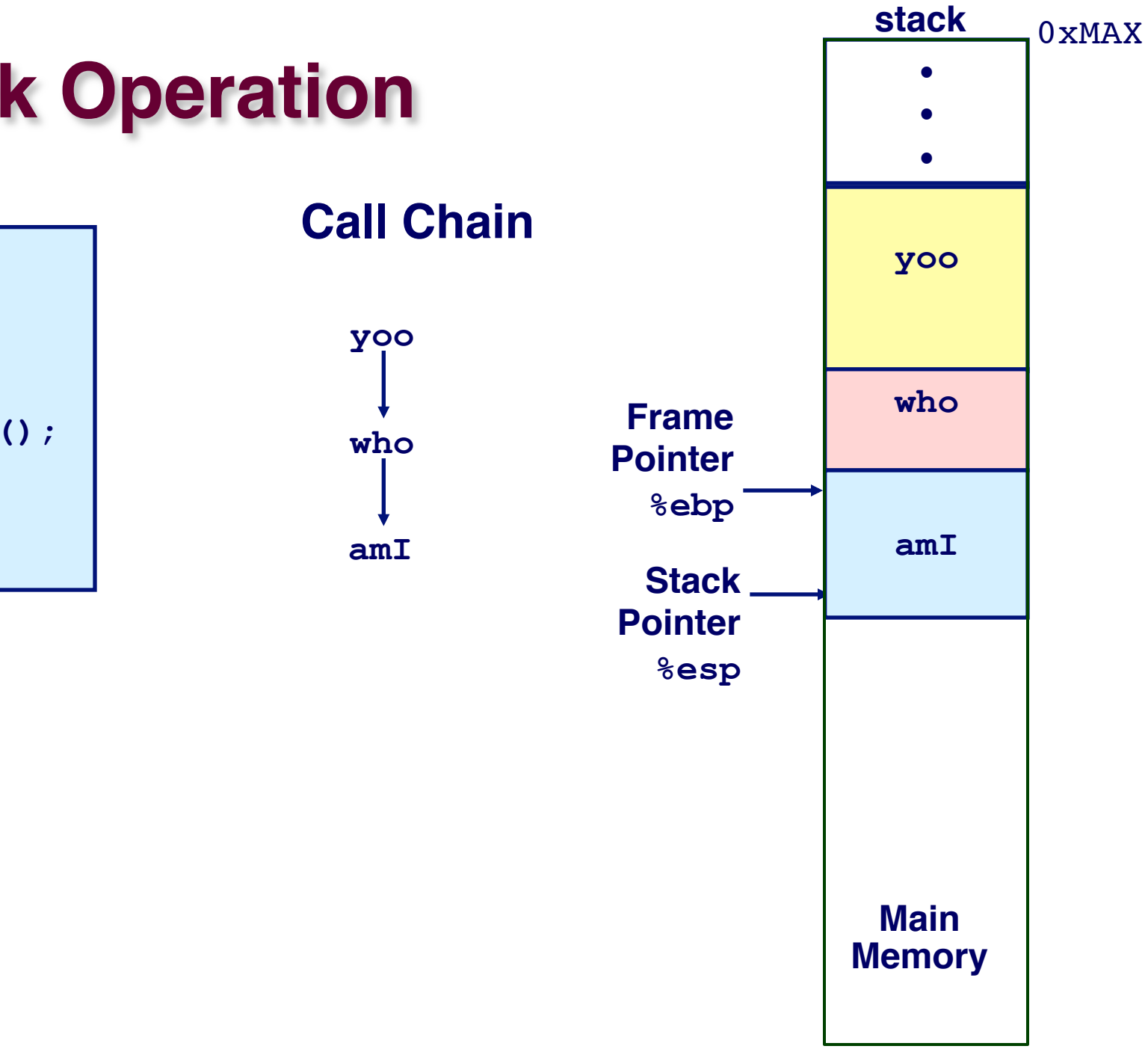


Stack Operation

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

Call Chain

yoo
↓
who
↓
amI

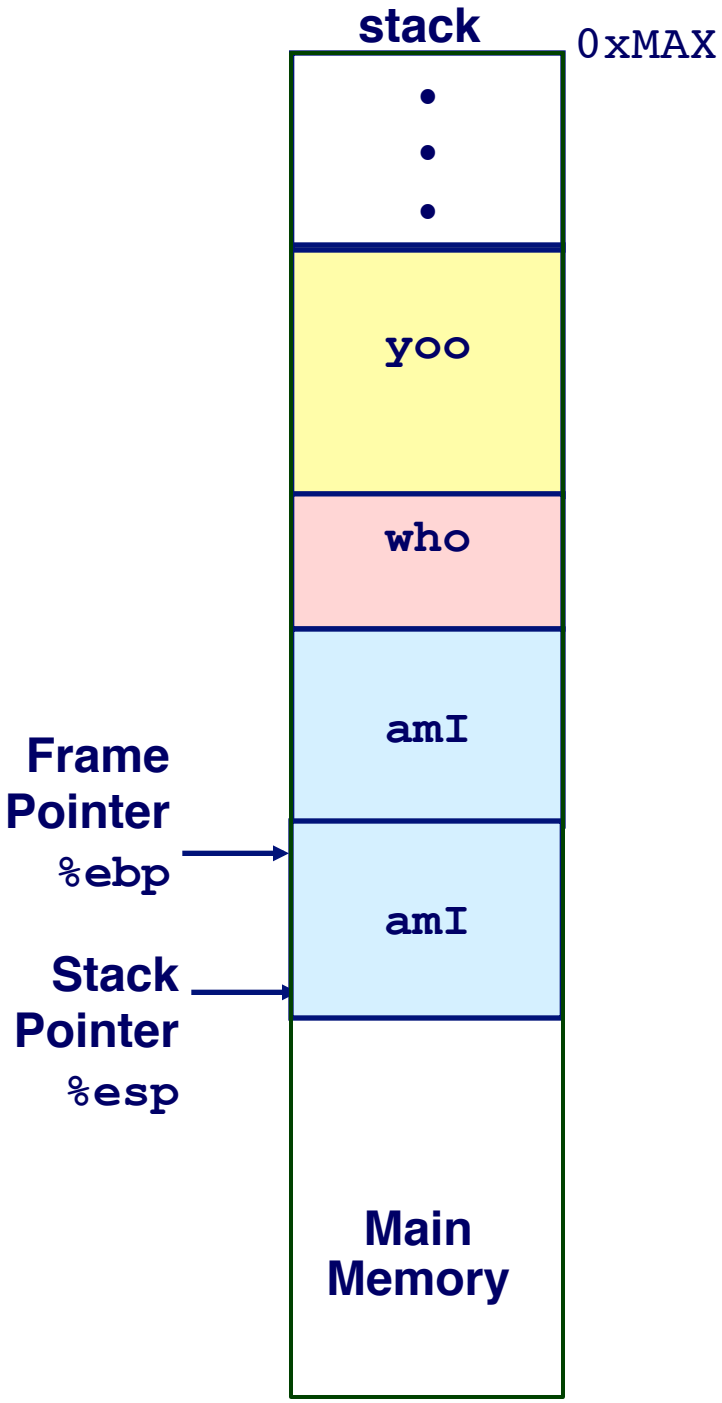


Stack Operation

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

Call Chain

yoo
↓
who
↓
amI
↓
amI

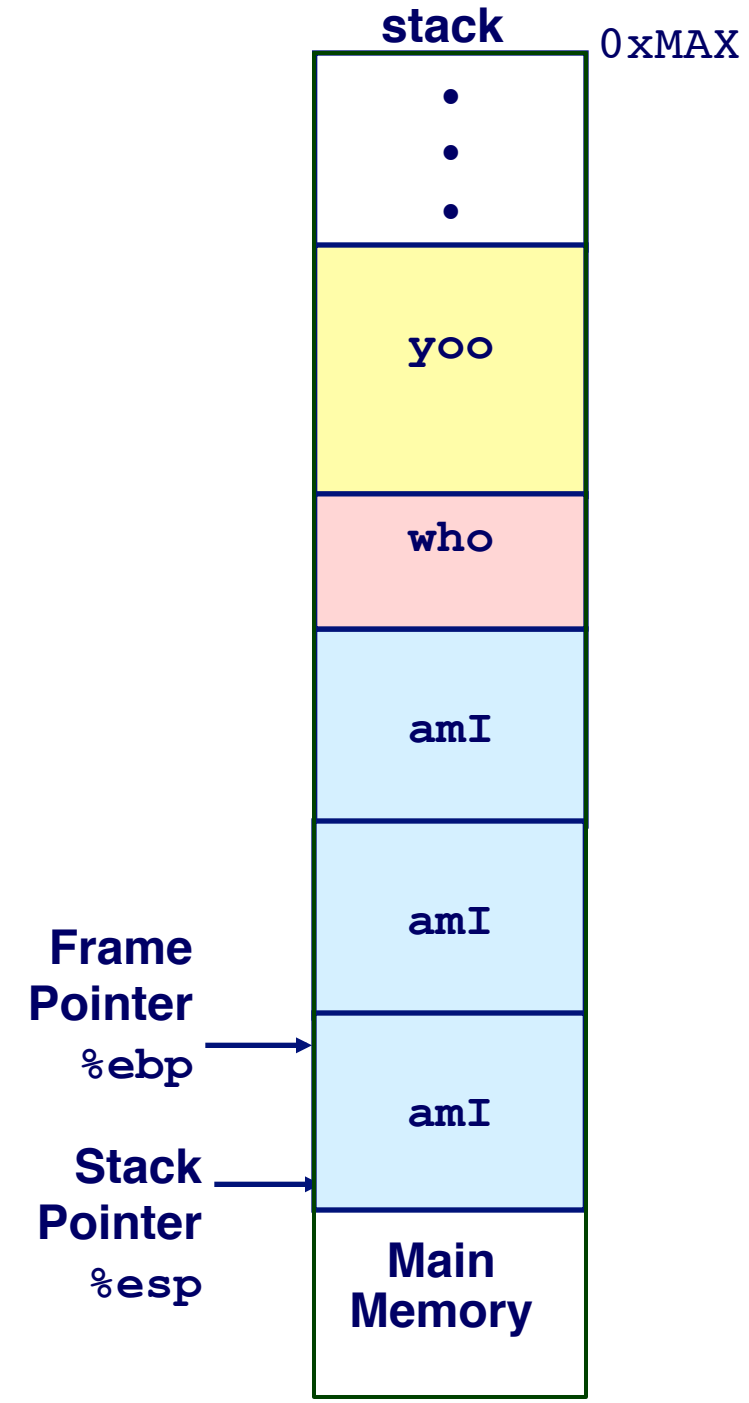


Stack Operation

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

Call Chain

yoo
↓
who
↓
amI
↓
amI
↓
amI

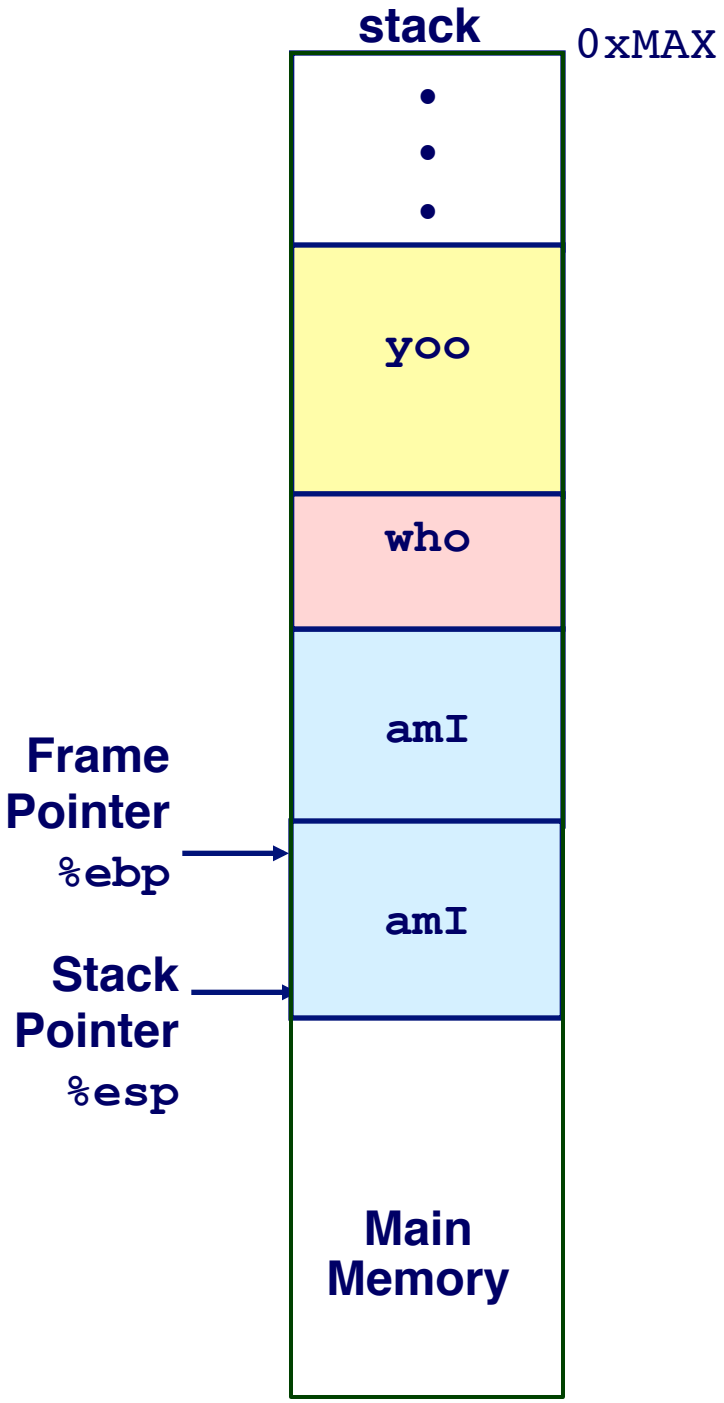


Stack Operation

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

Call Chain

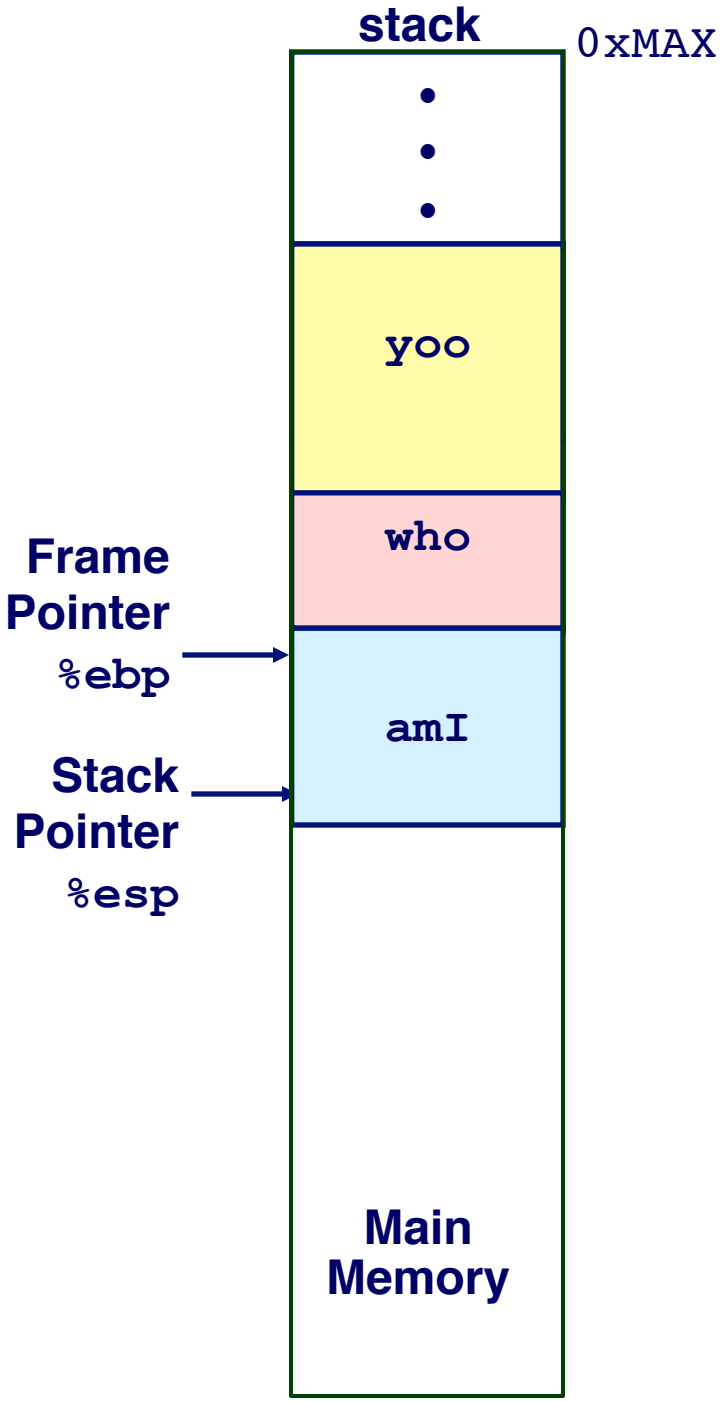
yoo
↓
who
↓
amI
↓
amI
↓
amI



Stack Operation

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

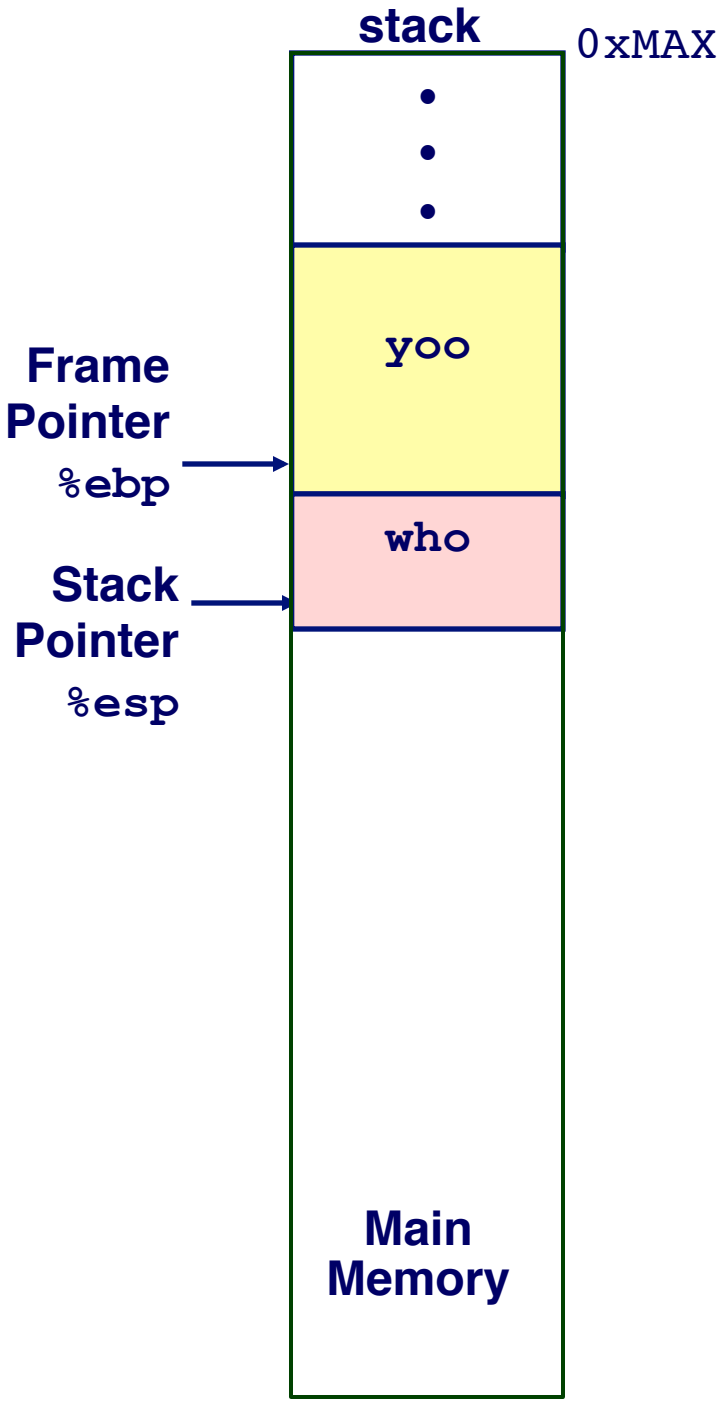
Call Chain



Stack Operation

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

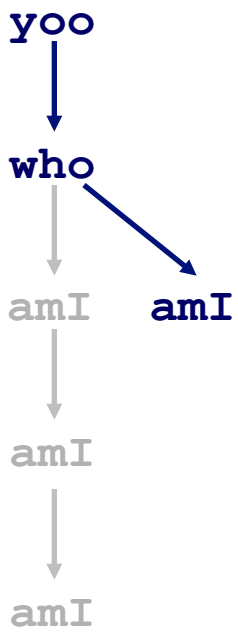
Call Chain

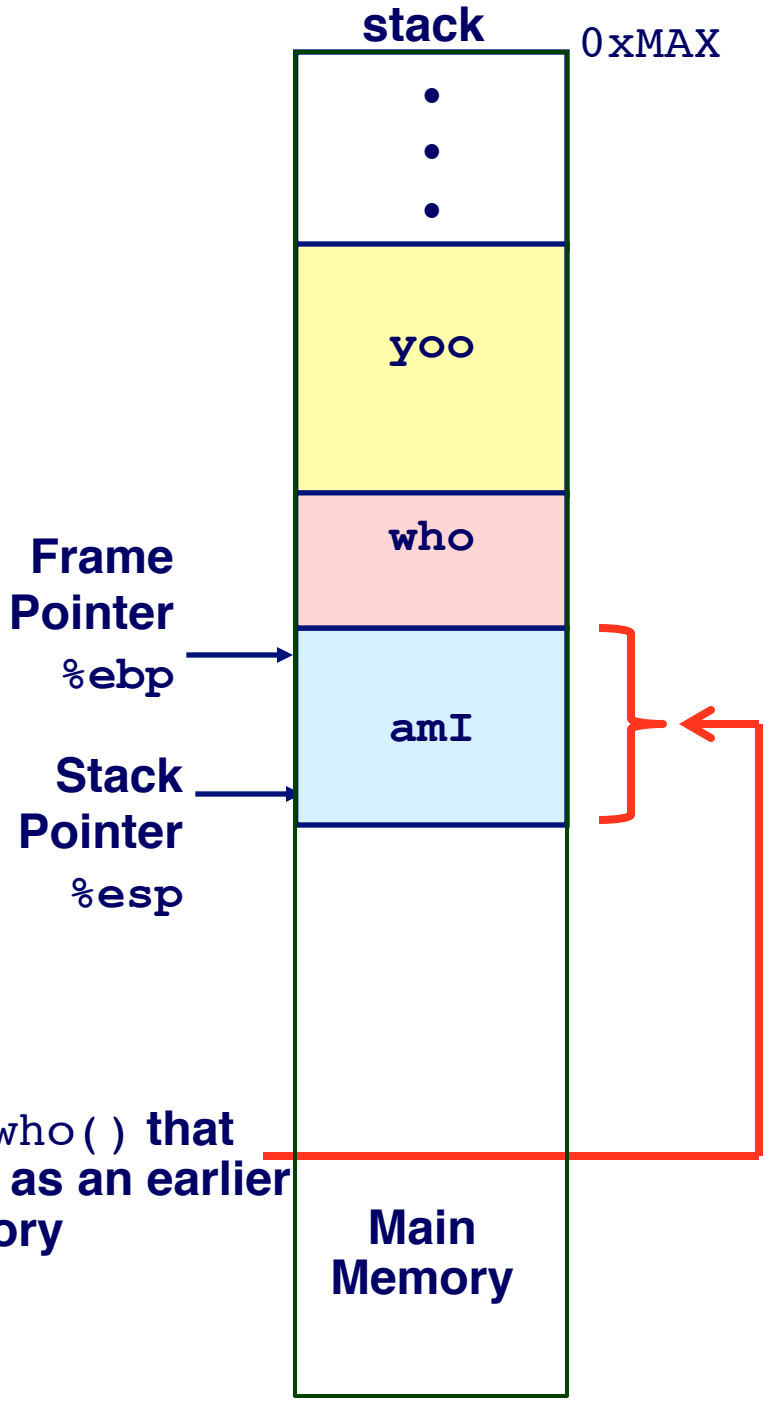


Stack Operation

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

Call Chain



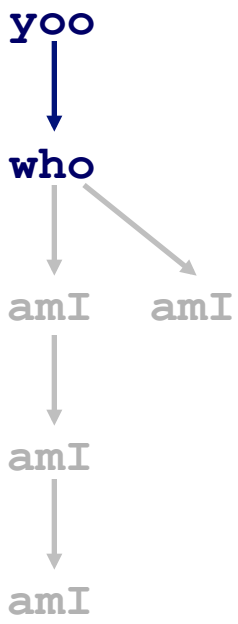


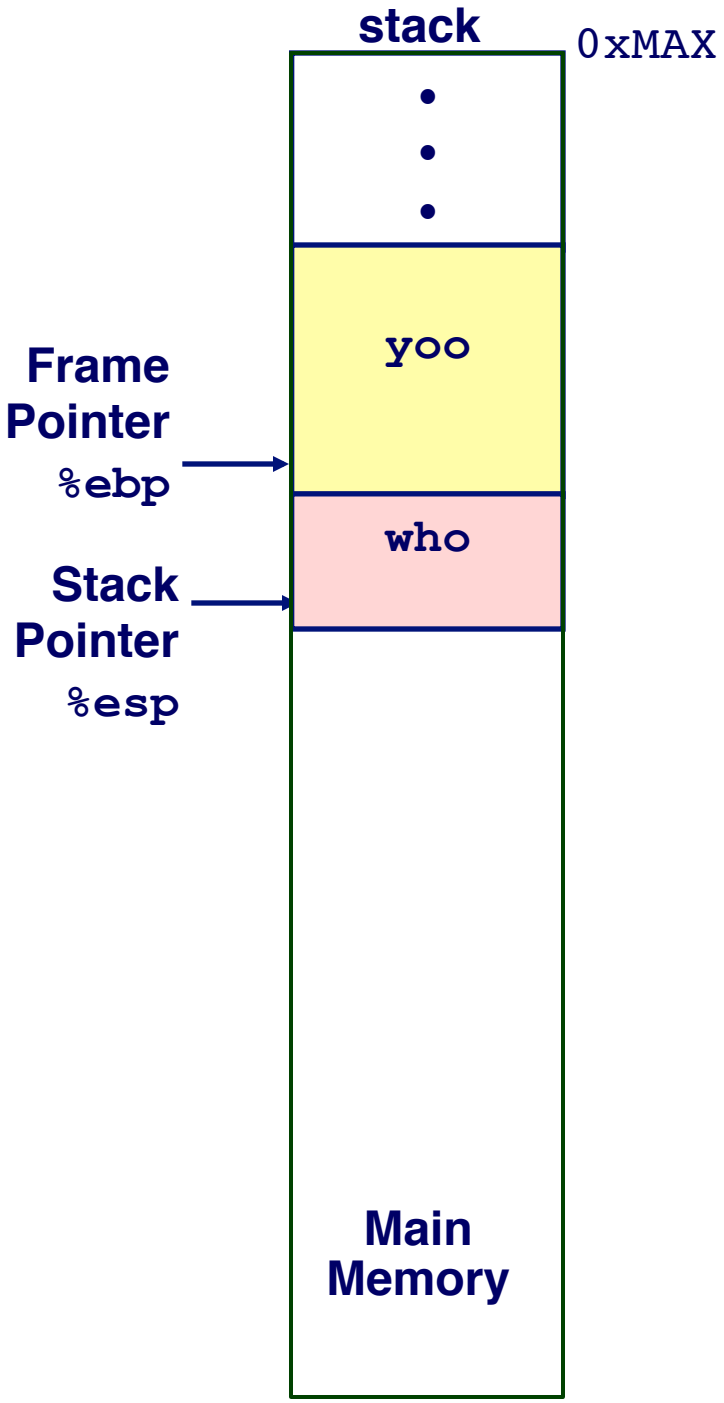
Note how the second time `amI()` is called by `who()` that this instance of `amI()` reuses the same memory as an earlier stack frame, i.e. it overwrites the memory

Stack Operation

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

Call Chain

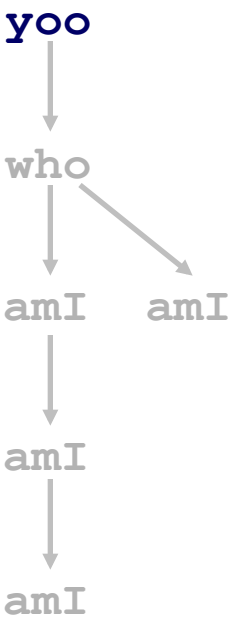


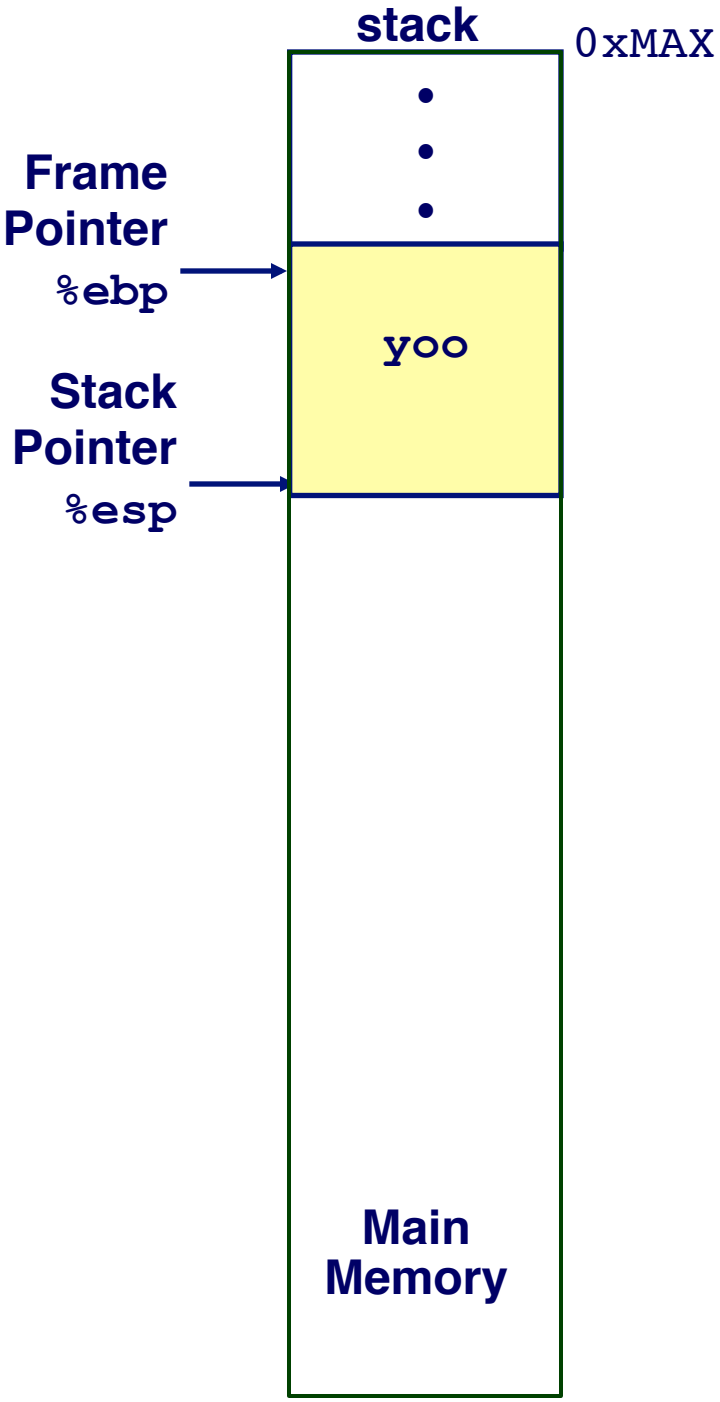


Stack Operation

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

Call Chain





Stack-Based Languages

Stack Allocated in *Frames*

- state for single procedure instantiation

Stack Discipline

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

Languages that Support Recursion

- e.g., C, Pascal, Java
- Code must be "*Reentrant*"
  - Multiple simultaneous instantiations of single procedure
- Store state of each instantiation on the stack
  - Arguments
  - Local variables
  - Return pointer

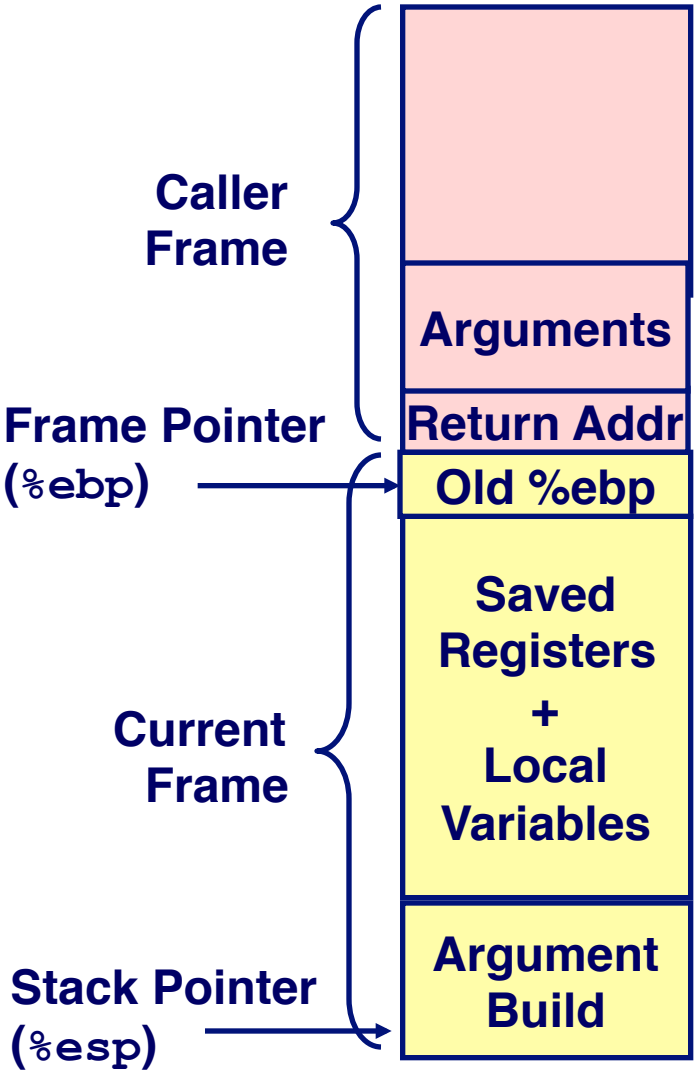
IA32/Linux Stack Frame

Caller Stack Frame

- Push parameters for function about to call
  - "Argument build"
  - Last argument pushed first, … first argument is pushed last – see p.220 of textbook
- Push return address
  - Pushed by `call` instruction

Current Stack Frame

- Push (save) old frame pointer
- Saved register context
- Allocate space for local variables
- More argument building as needed



Revisiting swap

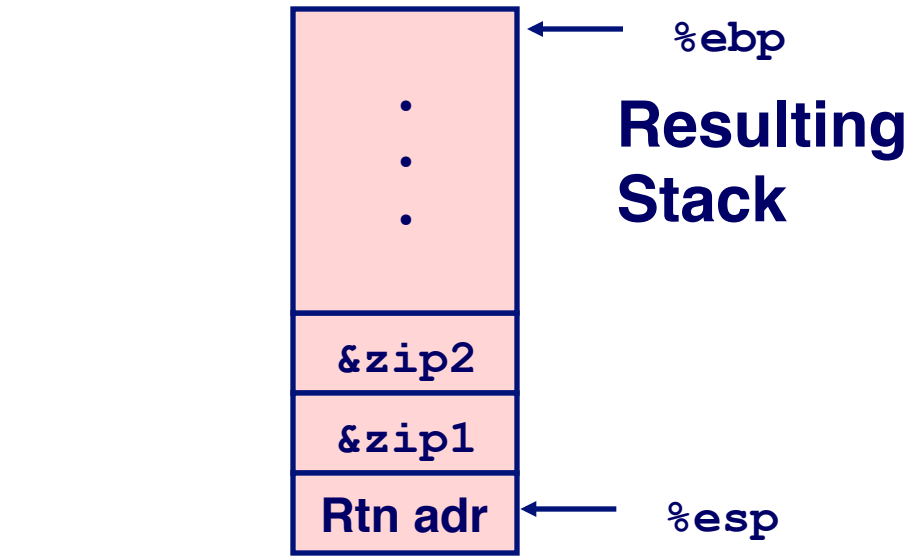
```
int zip1 = 15213;
int zip2 = 91125;

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
    pushl %ebx

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

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

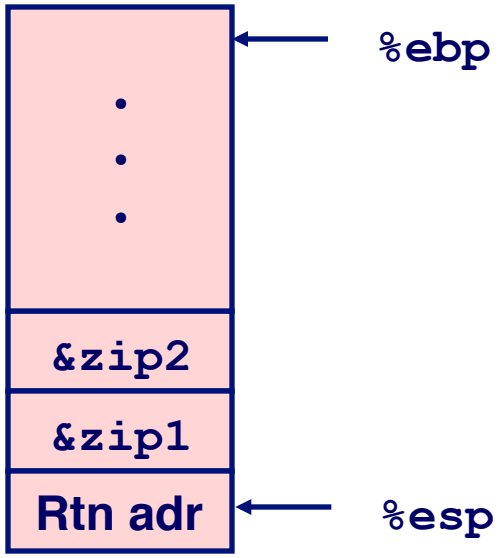
Set Up

Body

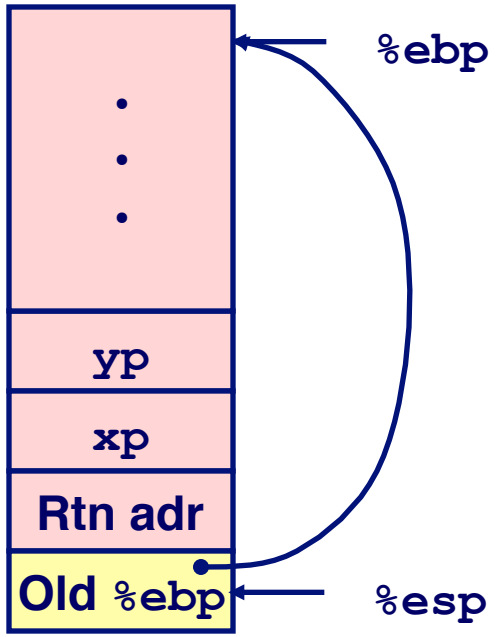
Finish

swap Setup #1

Entering Stack



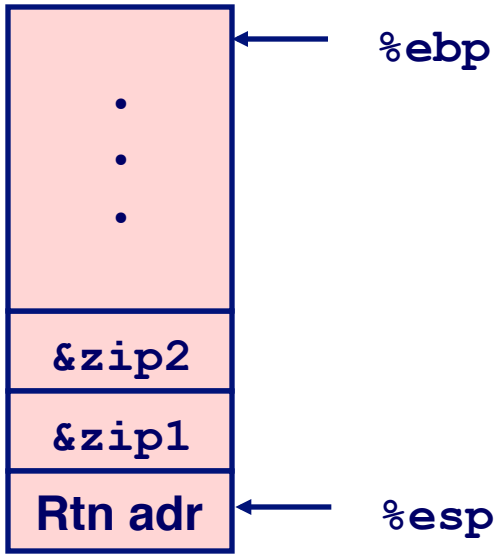
Resulting Stack



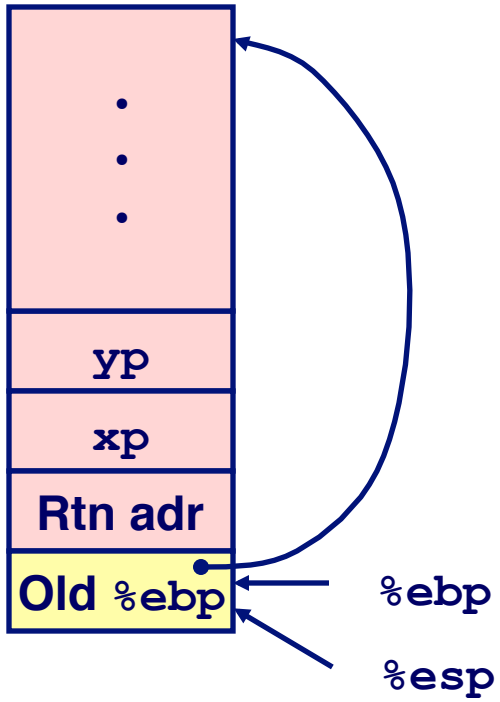

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #2

Entering Stack



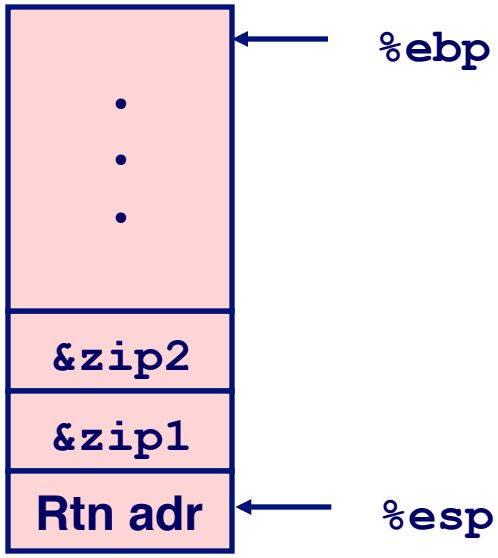
Resulting Stack



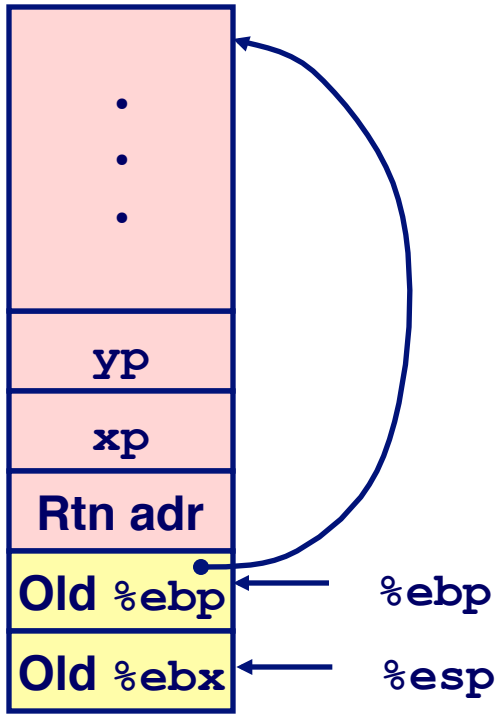

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #3

Entering Stack

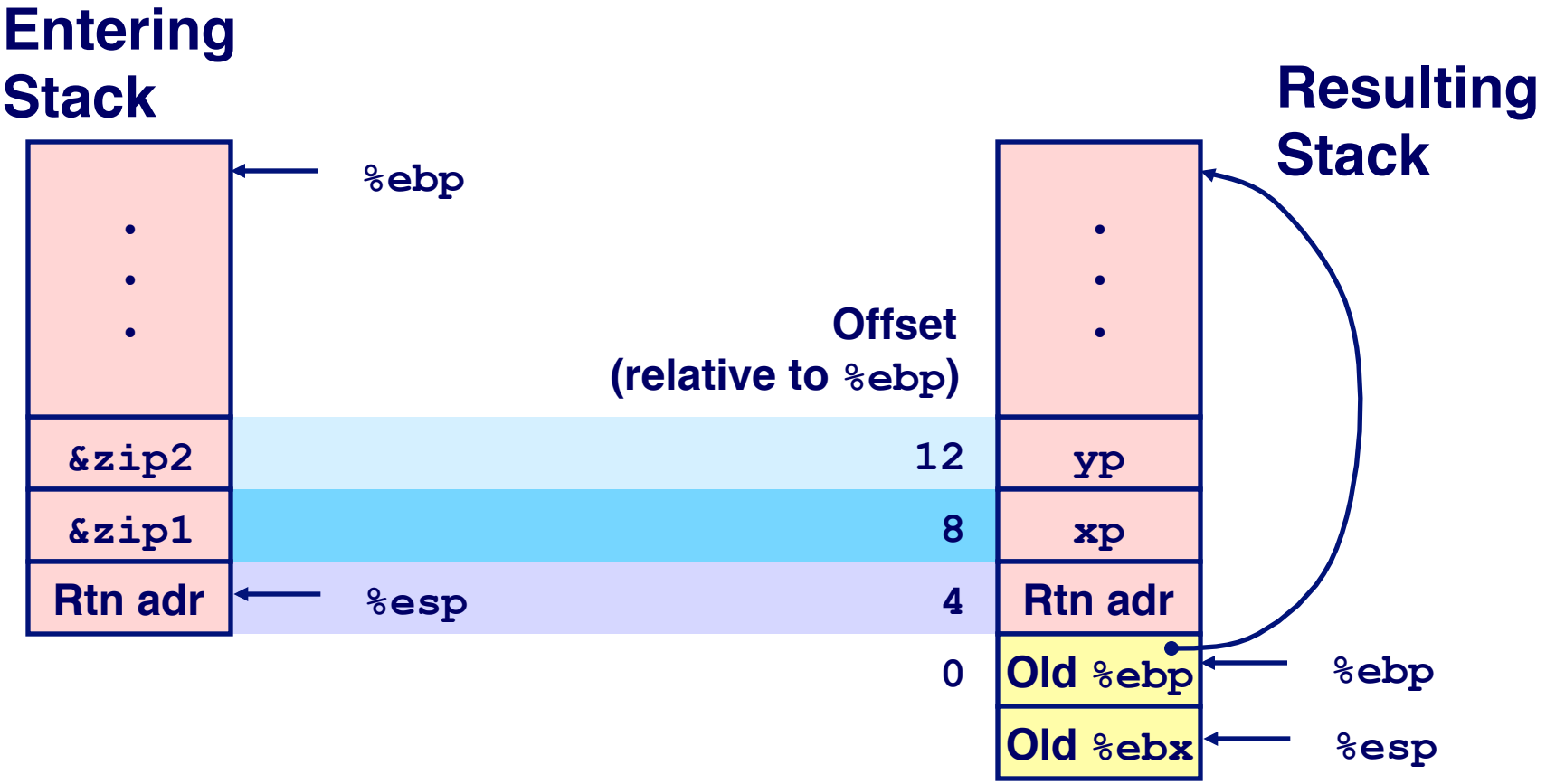


Resulting Stack




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

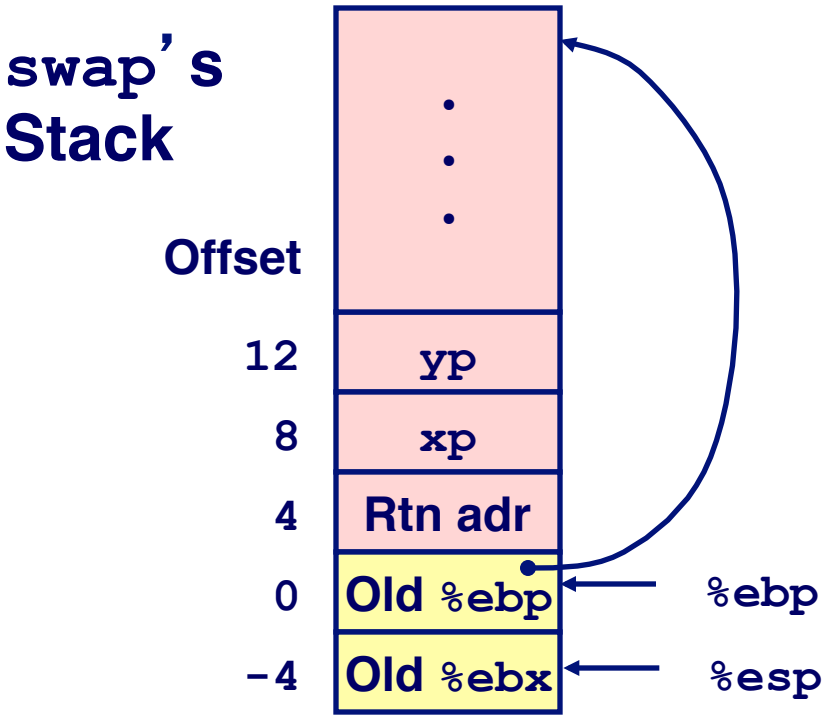
Effect of swap Setup

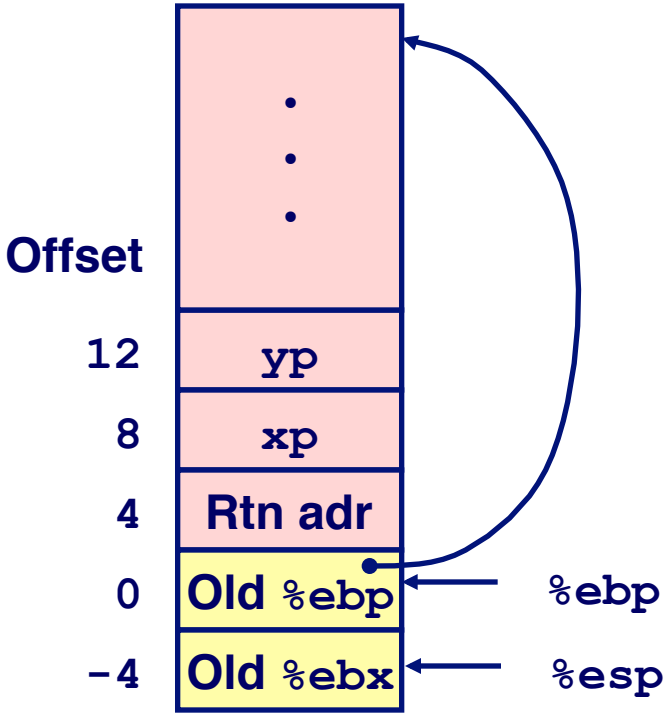



```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

Body

swap Finish #1



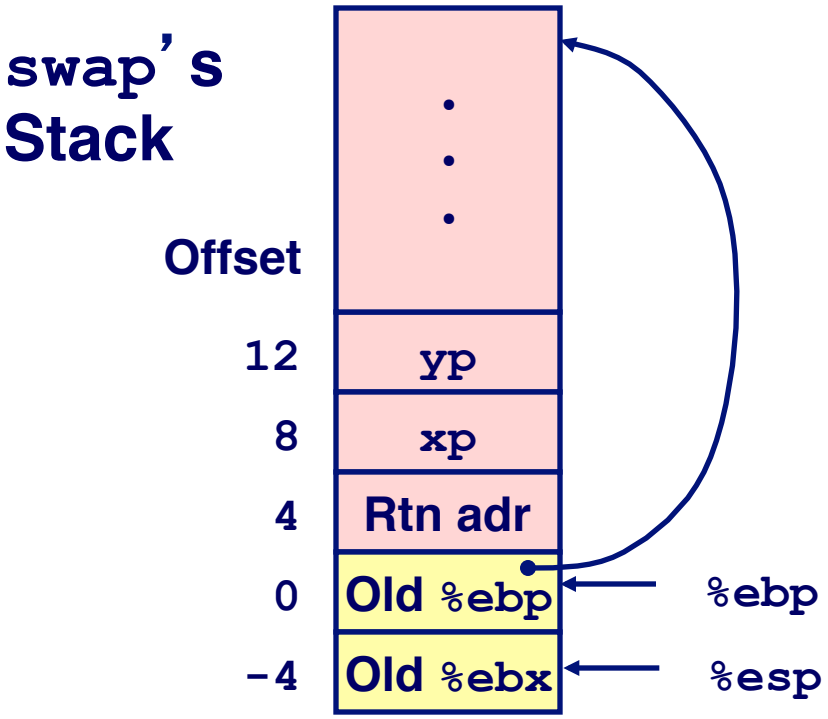


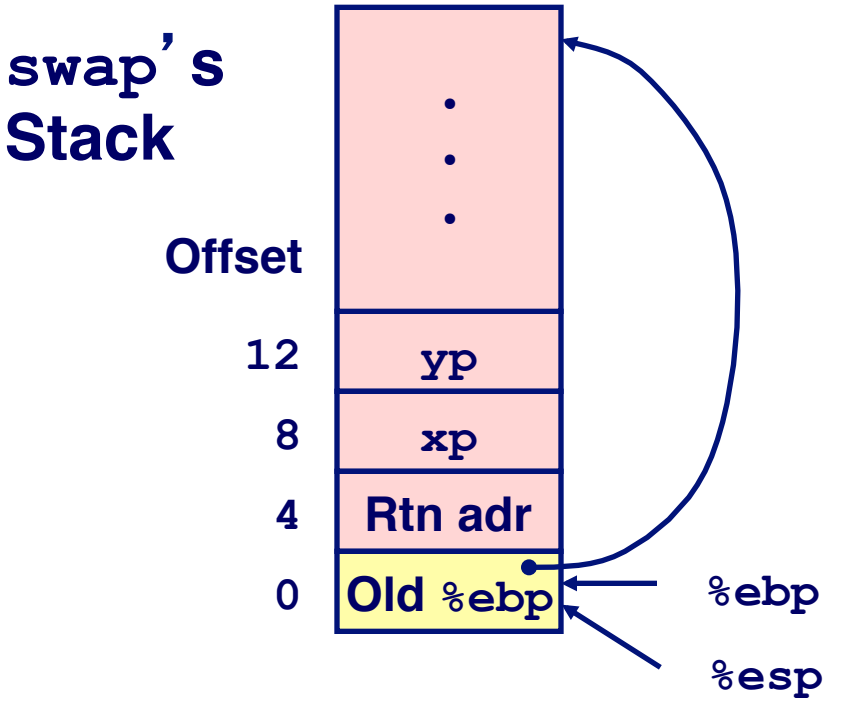

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

- Saved & restored register `%ebx`

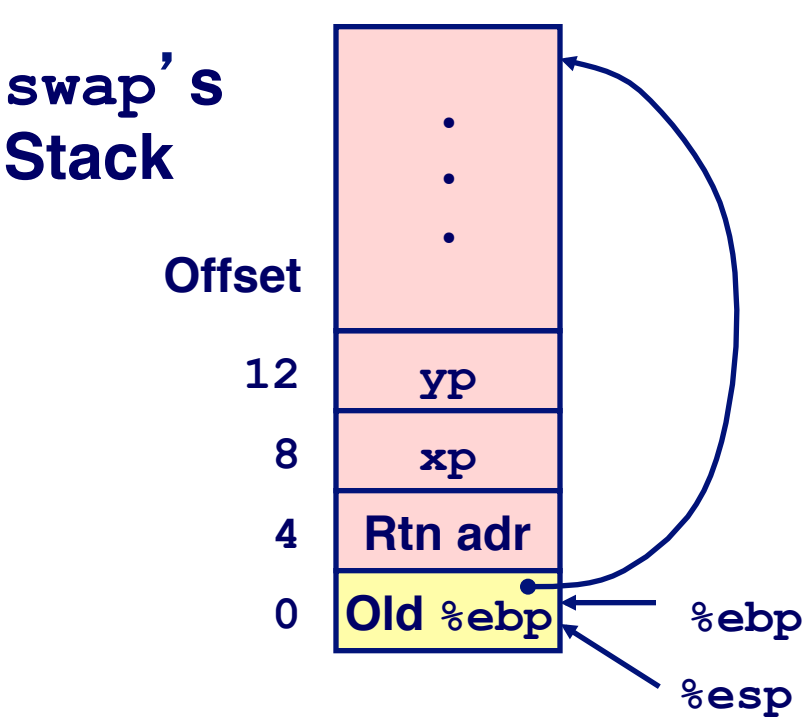
swap Finish #2

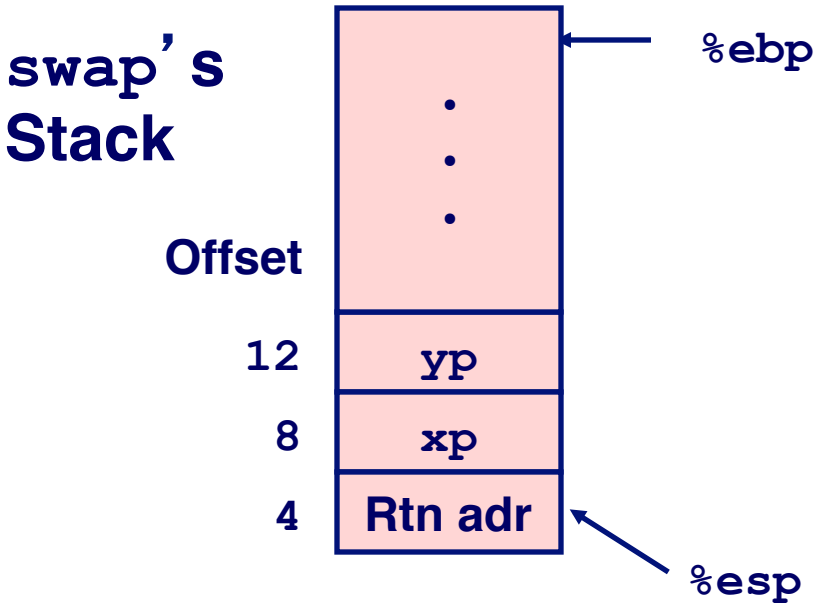





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

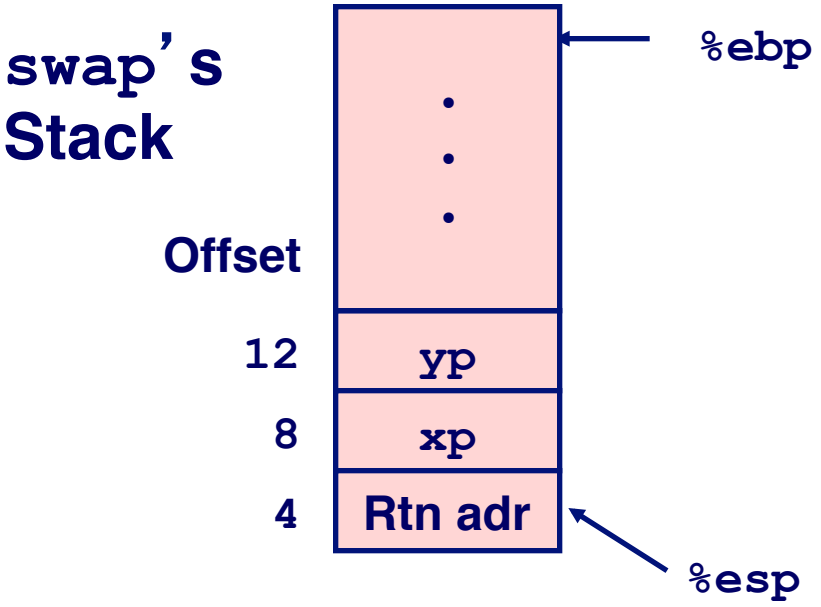
swap Finish #3

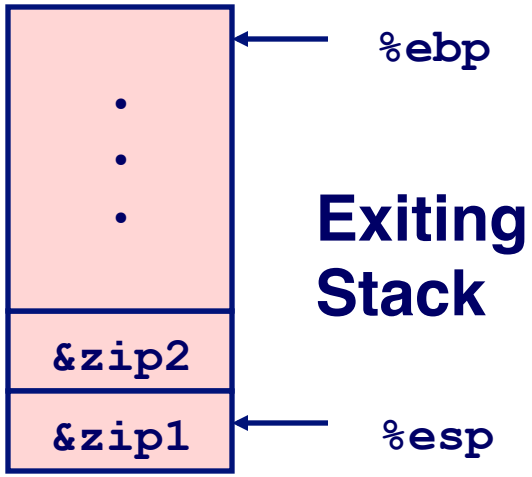





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #4






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

Observation

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

Register Saving Conventions

When procedure `yoo` calls `who`:

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

Can Register be Used for Temporary Storage?

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

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $91125, %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 Register be Used for Temporary Storage?

Conventions

- "Caller Save"
  - Caller saves temporary in its frame before calling
- "Callee Save"
  - Callee saves temporary in its frame before using

IA32/Linux Register Usage

Integer Registers

- Two have special uses
  - `%ebp`, `%esp`
- Three managed as callee-save
  - `%ebx`, `%esi`, `%edi`
    - Old values saved on stack prior to using
- Three managed as caller-save
  - `%eax`, `%edx`, `%ecx`
    - Do what you please, but expect any callee to do so, as well
- Register `%eax` also stores returned value