

# 15-410

*“An Experience Like No Other”*

Stack Discipline  
Aug. 28, 2024

**Dave Eckhardt**

**Slides originally stolen from 15-213**

# Synchronization

**Reminder: the syllabus has been released!**

- Please read it, carefully, right away
  - Please do *not* wait until halfway through P0



# Synchronization

## Registration

- 15-410
  - Still some advisor discussions in the air
  - Seems very unlikely everybody qualified will fit
- 15-605
  - I think enrollment is nearly over
- If you're here but not on *any* wait list, see me *right away*
- If you are an M.S. or or Ph.D. student and have not discussed this class with your advisor, do so *today*
  - We will not be registering graduate students without hearing from their advisors

## If you haven't taken 15-213 (A/B, malloc lab ok)

- Contact me no later than *today*

# Synchronization

## Office hours today?

- I believe so – see the Staff web page
- Don't forget about the syllabus!



# Outline

## Topics

- **Process memory model**
- **IA32 stack organization**
- **Register saving conventions**
- **Before & after `main()`**
- **Project 0**

# Why Only 32?

**You may have learned x86-64 aka EMT64 aka AMD64**

- **x86-64 is simpler than x86(-32) for user program code**
  - **Lots of registers, registers more orthogonal**

**Why will 410 be x86 / IA32?**

# Why Only 32?

## You may have learned x86-64 aka EMT64 aka AMD64

- x86-64 is simpler than x86(-32) for user program code
  - Lots of registers, registers more orthogonal

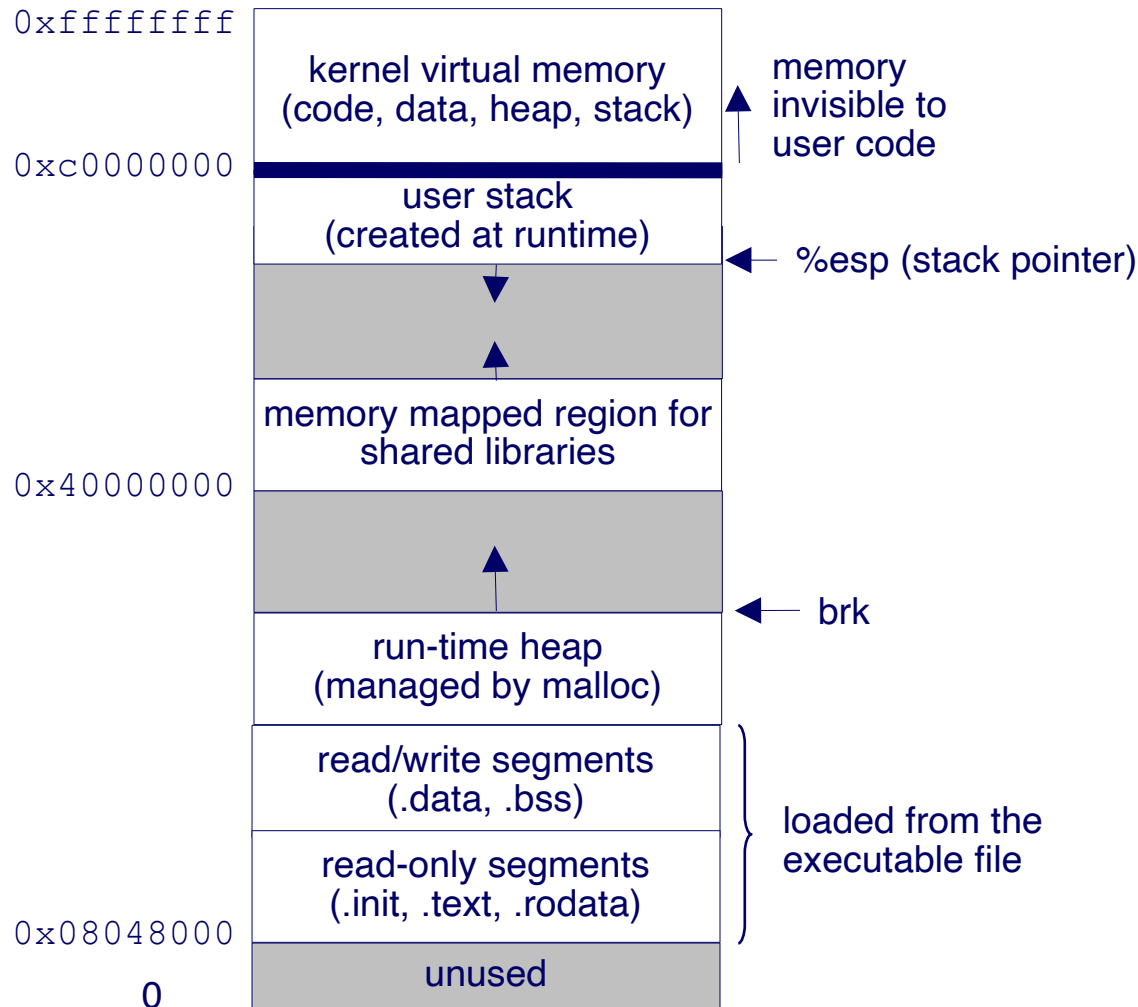
## Why will 410 be x86 / IA32?

- x86-64 is *not* simpler for kernel code
  - Machine begins in 16-bit mode, then 32, finally 64
    - » You don't have time to write 32⇒64 transition code
    - » If we gave it to you, it would be a *big* black box
  - Interrupts are more complicated
- x86-64 is *not* simpler during debugging
  - More registers means more registers to have wrong values
- x86-64 virtual memory is a bit of a drag
  - More steps than x86-32, but not more intellectually stimulating
- There are still a lot of 32-bit machines in the world

## CS:APP 32-bit guide

# Private Address Spaces

**Each process has its own private address space.**



**Warning:**  
numbers  
specific to  
Linux 2.x  
on IA32!!

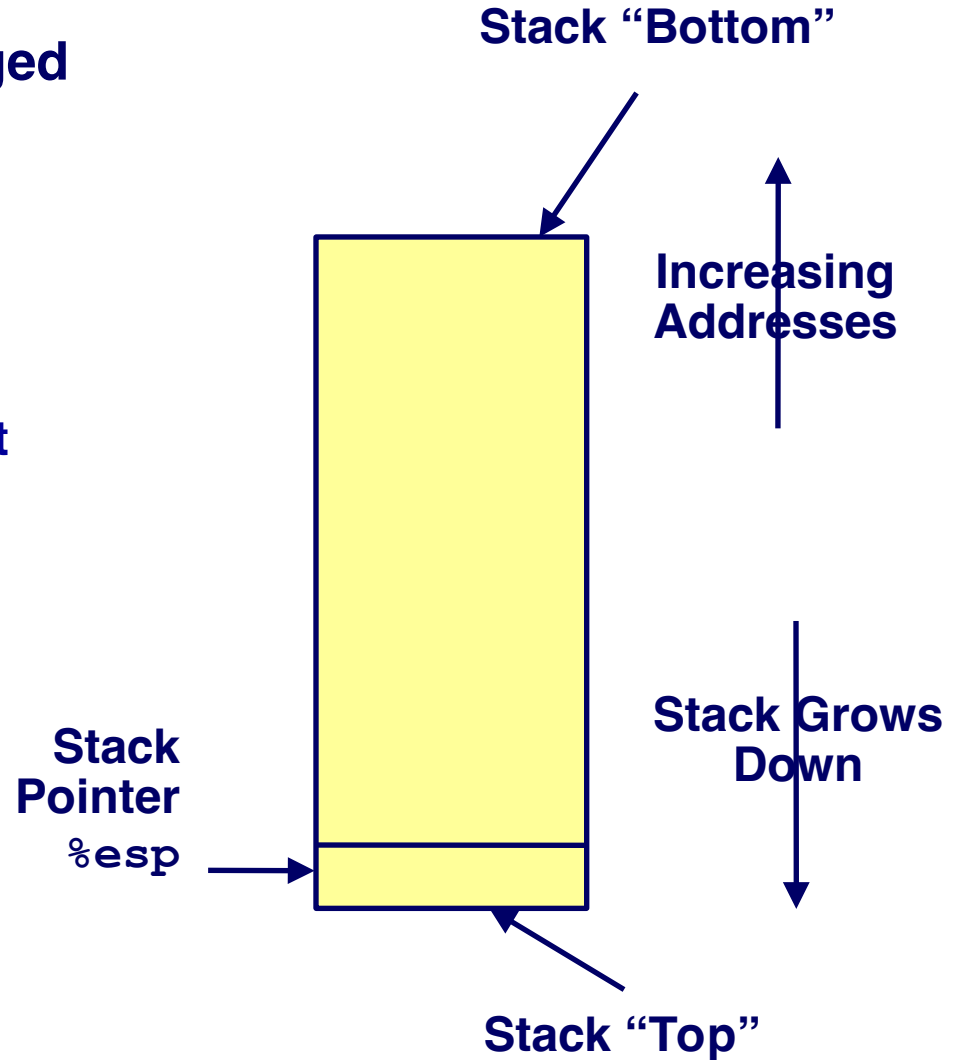


**Warning:**  
details vary  
by OS and  
kernel  
version!



# IA32 Stack

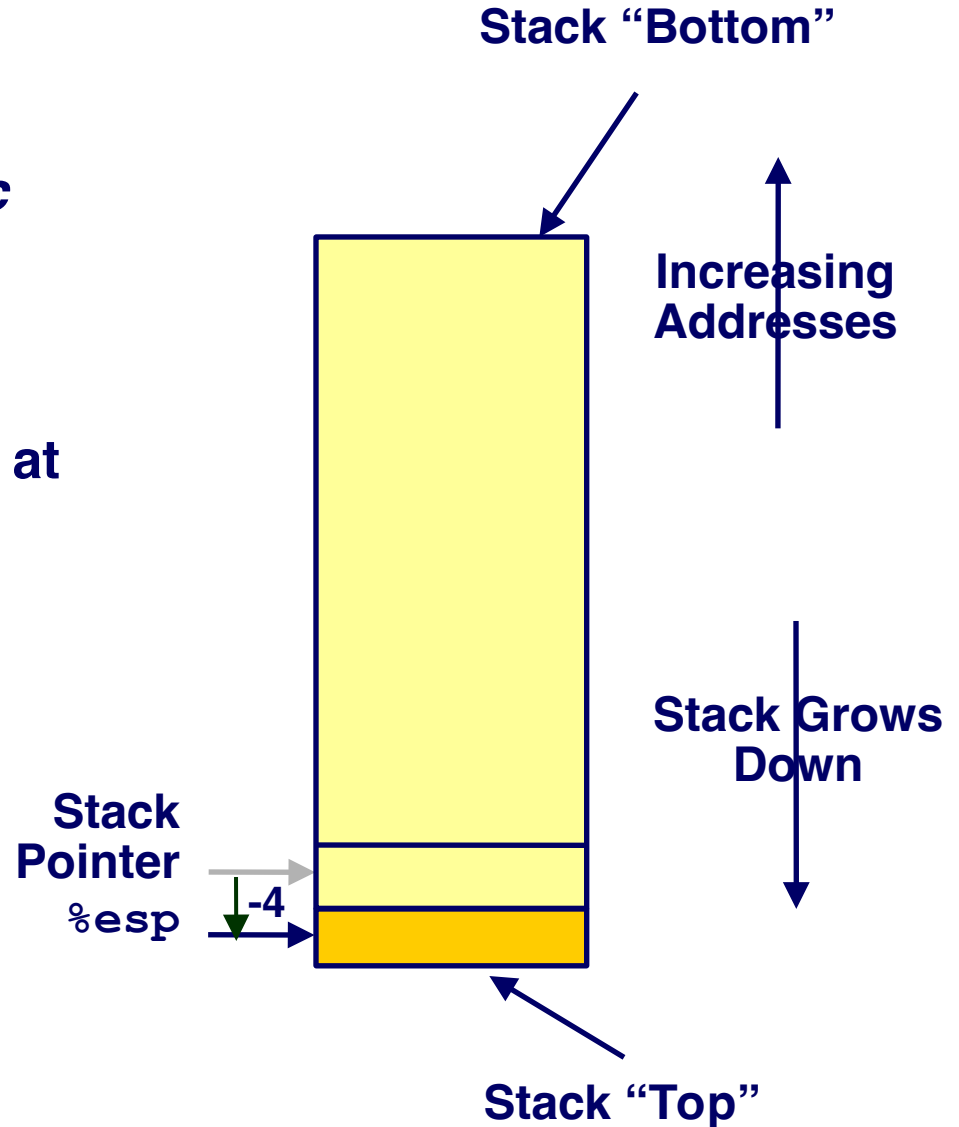
- Region of memory managed with stack discipline
- “Grows” toward lower addresses
- Register `%esp` indicates lowest stack address
  - address of “top” element
  - stack *pointer*



# IA32 Stack Pushing

## Pushing

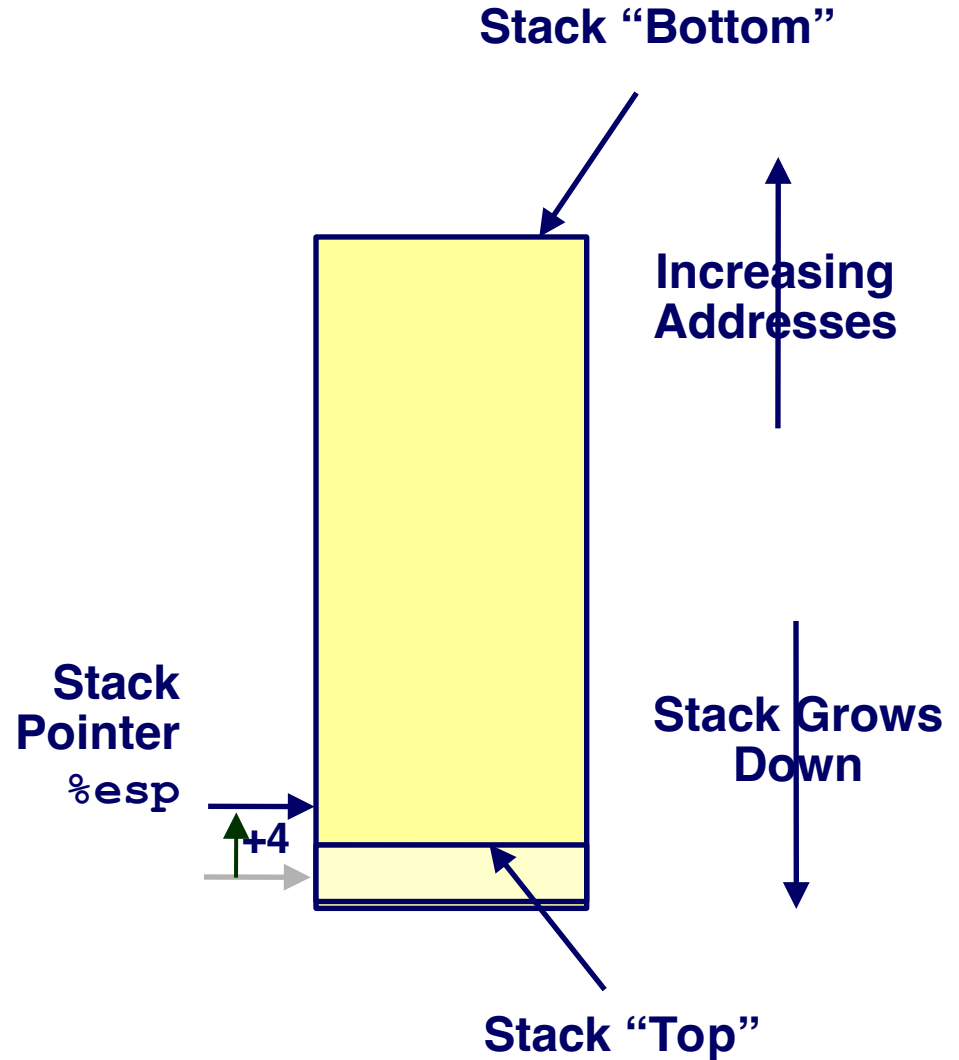
- `pushl Src`
- Fetch “operand” from *Src*
  - Maybe a register: `%ebp`
  - Maybe memory: `8(%ebp)`
- Decrement `%esp` by 4
- Store operand in memory at address given by `%esp`



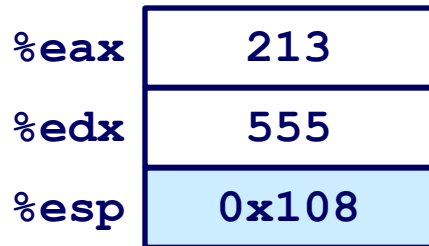
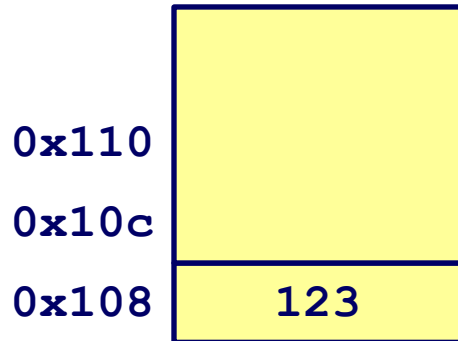
# IA32 Stack Popping

## Popping

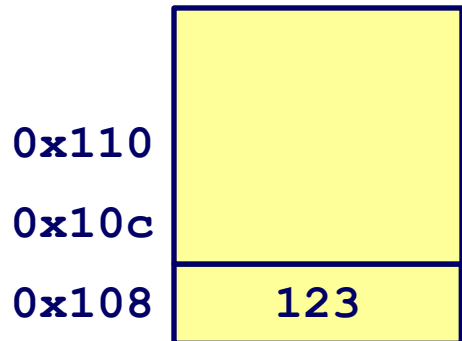
- `popl Dest`
- Read memory at address given by `%esp`
- Increment `%esp` by 4
- Store into *Dest* operand



# Stack Operation Examples



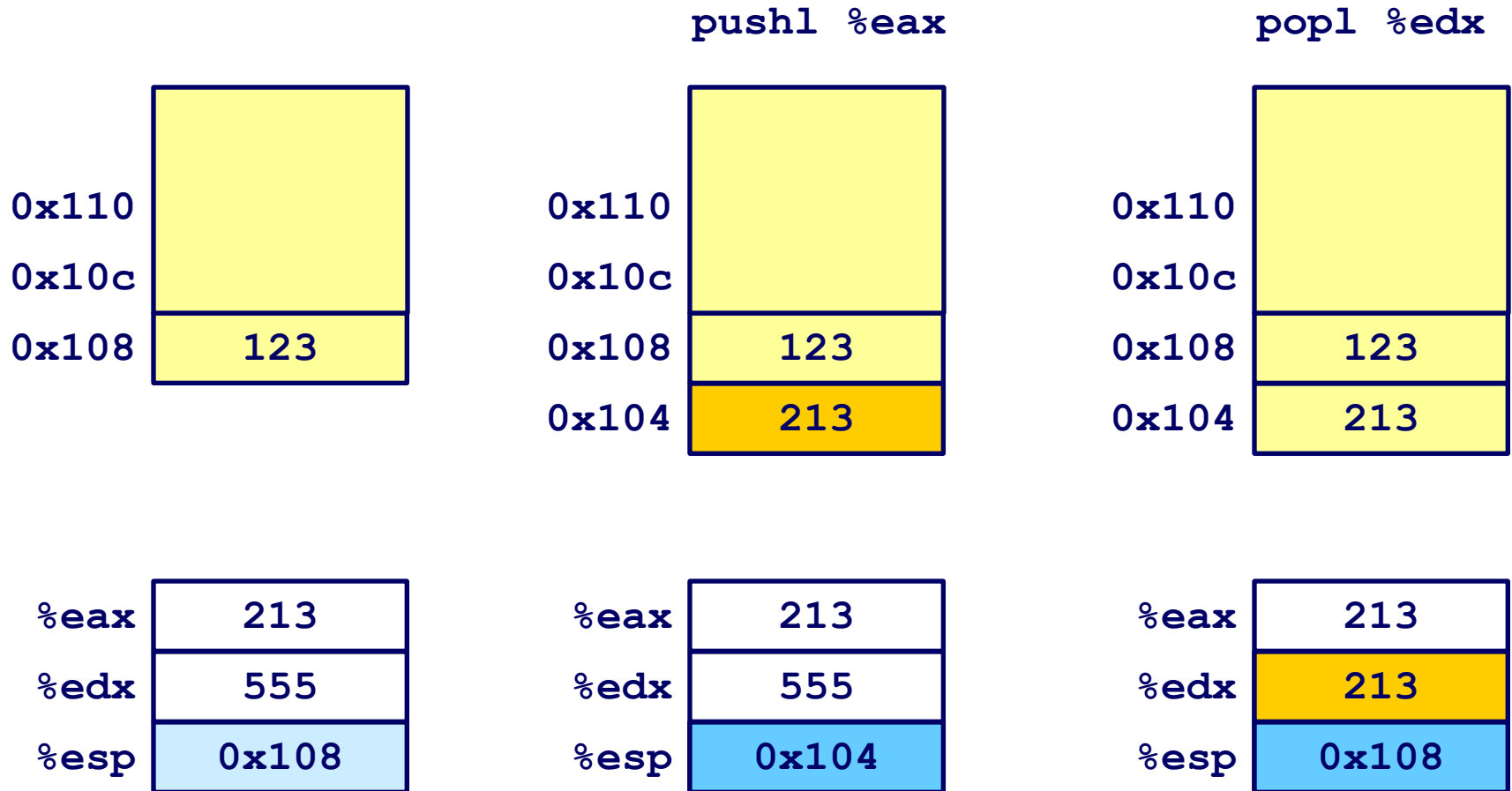
# Stack Operation Examples



<code>%eax</code>	213
<code>%edx</code>	555
<code>%esp</code>	0x108

“You are having a bad problem and you will not go to space today.”

# Stack Operation Examples



# Procedure Control Flow

- Use stack to support procedure call and return

## Procedure call:

- `call label`      Push return address;  
                         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`

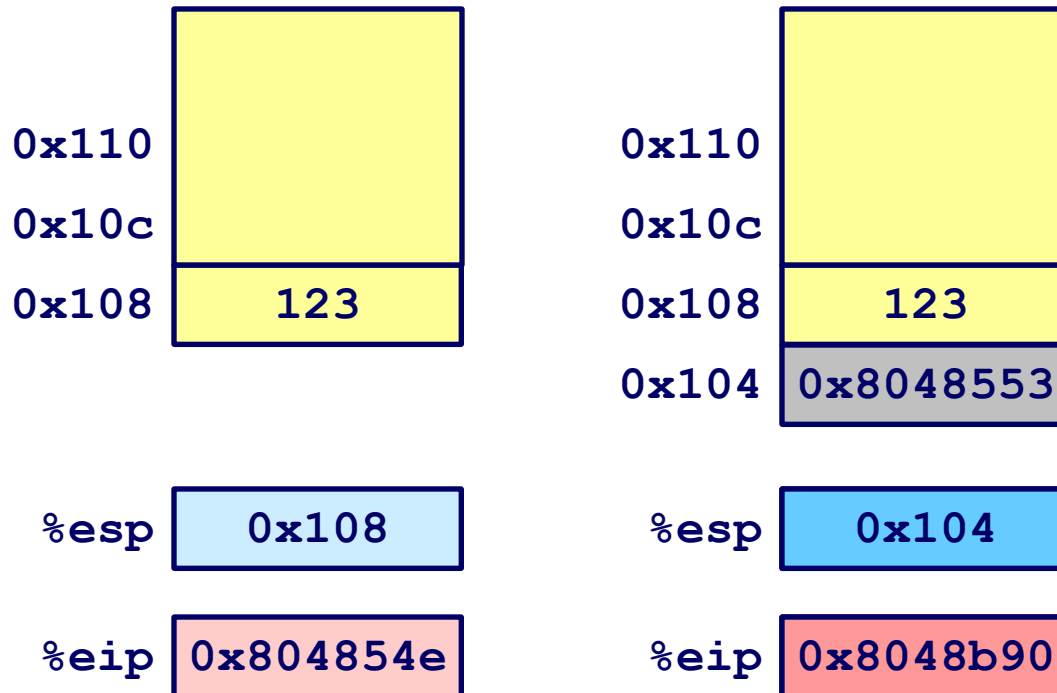
## Procedure return:

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

# Procedure Call Example

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

```
call 8048b90
```



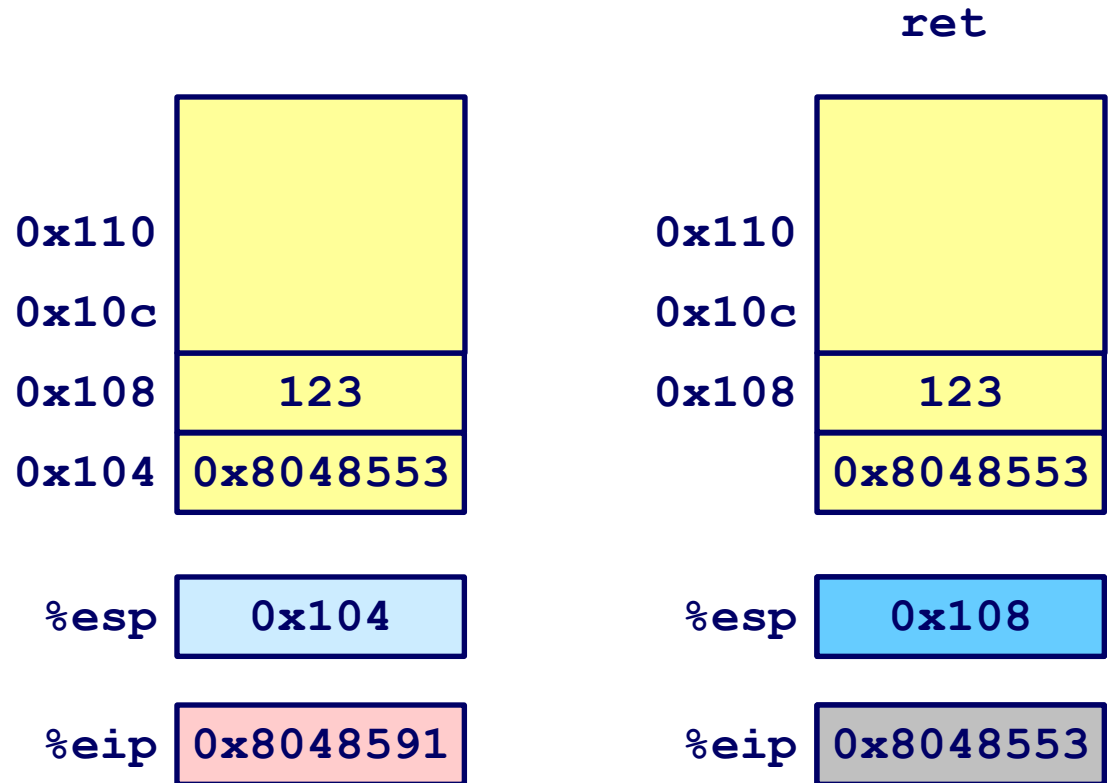
`%eip` is program counter



# Procedure Return Example

8048591: c3

ret



**%eip** is program counter

# Stack-Based Languages

## Languages that support recursion

- e.g., C, Pascal, Java
- Code must be “*reentrant*”
  - Multiple instantiations of a single procedure “live” at same time
- Need some place to store state of each instantiation
  - Arguments
  - Local variables
  - Return pointer (maybe)
  - Weird things (static links, exception handling, ...)

## Stack discipline – key observation

- State for given procedure needed for limited time
  - From time of call to time of return
- Note: callee returns before caller does

## Therefore stack allocated in nested *frames*

- State for single procedure instantiation

# Call Chain Example

## Code Structure

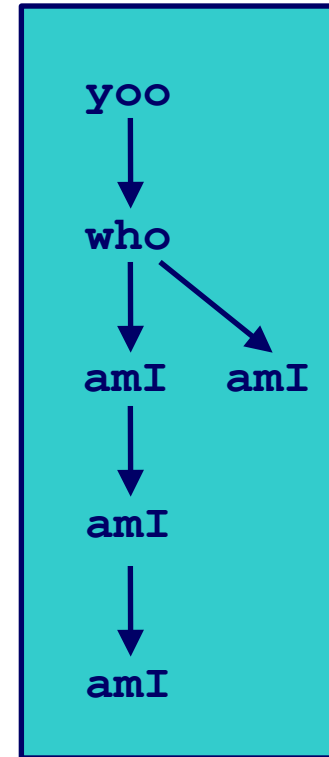
```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

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

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

- Procedure `amI ()` recursive

## Call Chain



# Stack Frames

## Contents

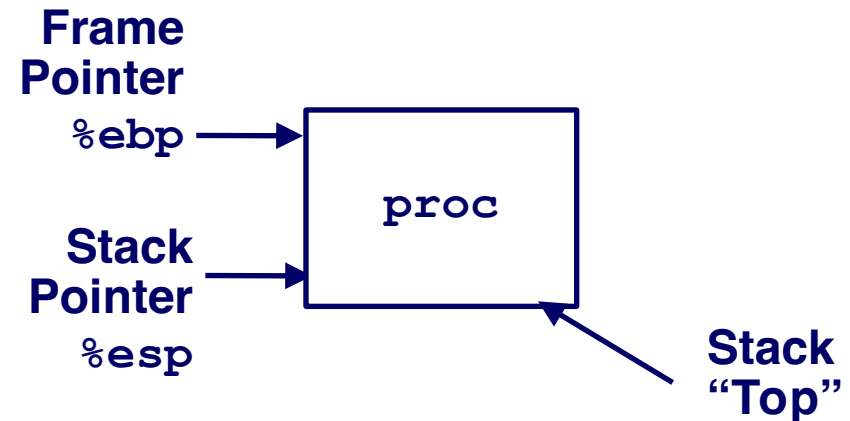
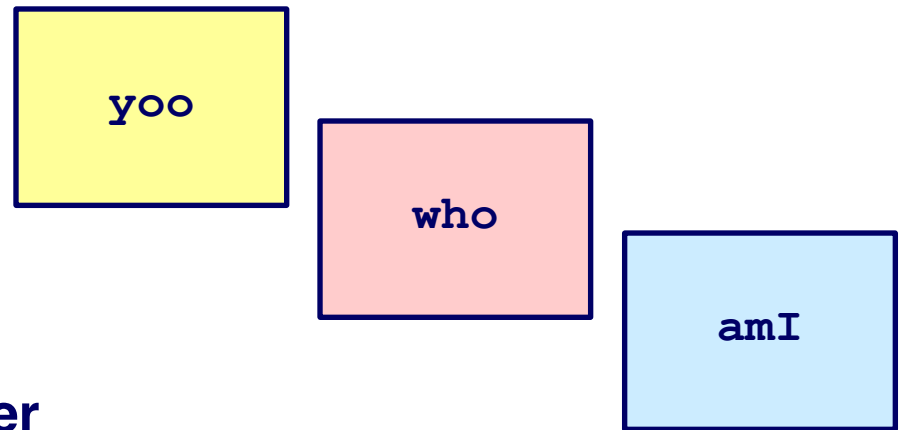
- Local variables
- Return information
- Temporary space

## 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



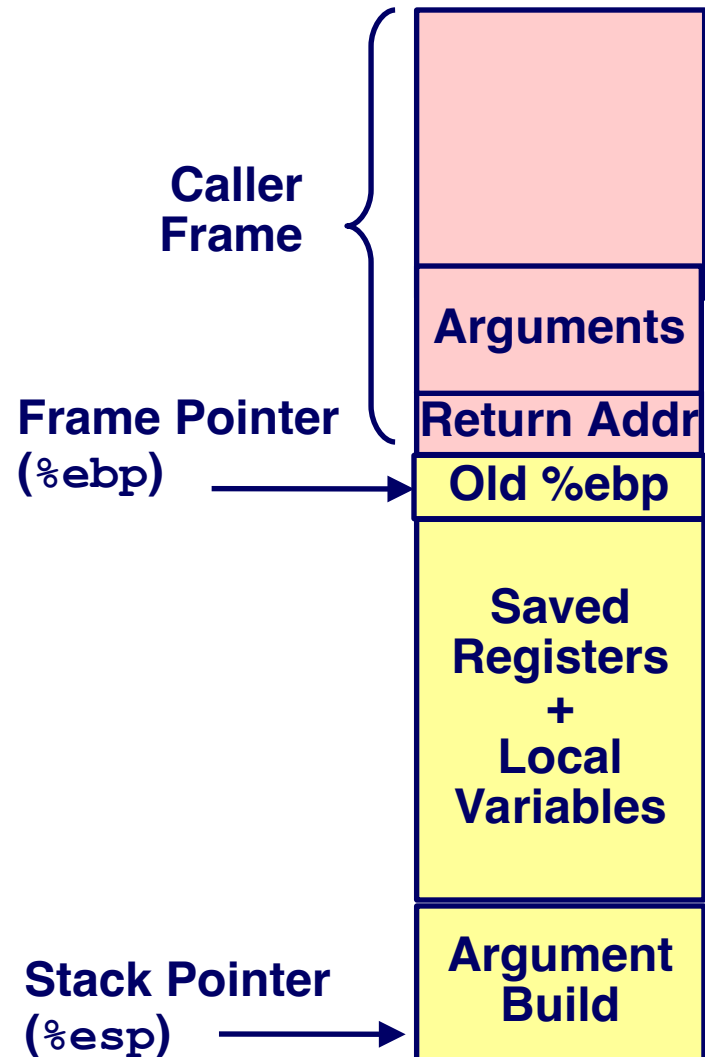
# IA32/Linux Stack Frame

## Current Stack Frame (“Top” to “Bottom”)

- Parameters for function we're about to call
  - “Argument build”
- Local variables
  - If don't all fit in registers
- Caller's saved registers
- Caller's saved frame pointer

## Caller's Stack Frame

- Return address
  - Pushed by `call` instruction
- Arguments for us!



# swap ()

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

# 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;
}
```

# 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
    . . .
```



# swap ()

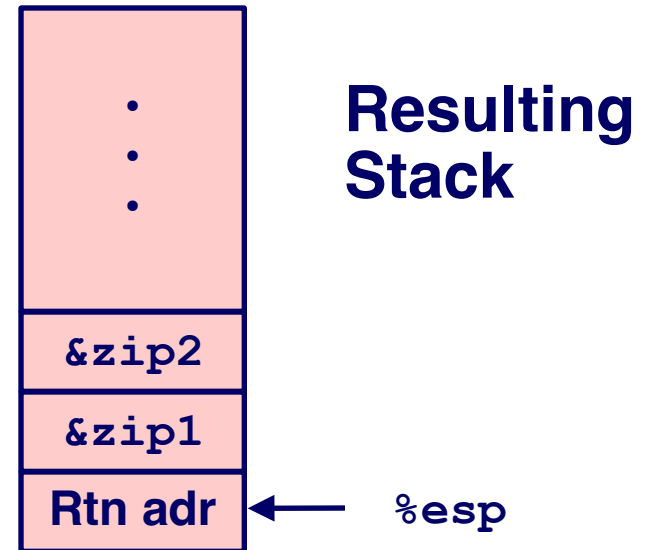
```
int zip1 = 15213;
int zip2 = 91125;

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
    . . .
```



# 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
```

} Set Up

Core {

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

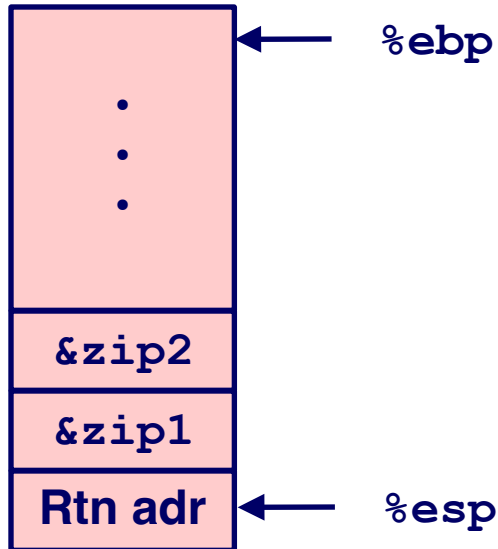
} Body

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

} Finish

# swap () Setup

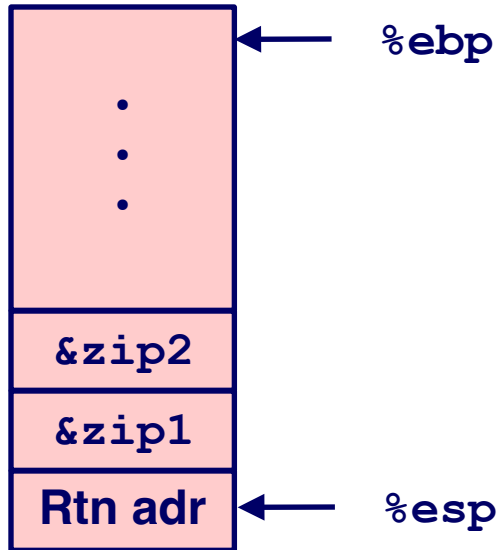
## Entering Stack



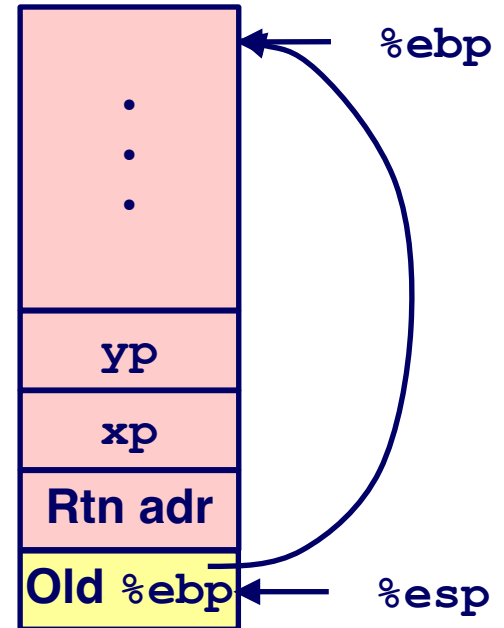
```
swap:  
    pushl %ebp  
    movl %esp,%ebp  
    pushl %ebx
```

# swap () Setup #1

## Entering Stack



## Resulting Stack

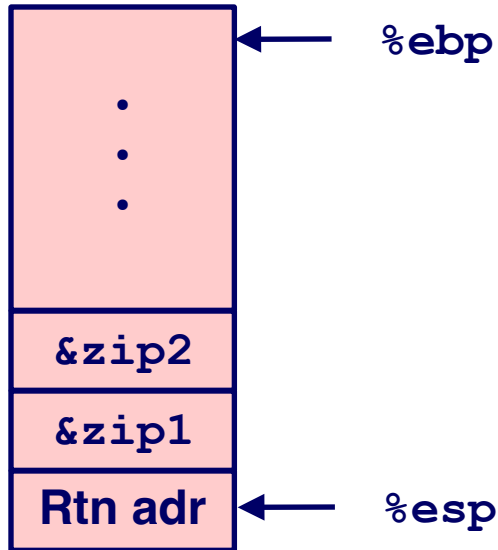


`swap:`

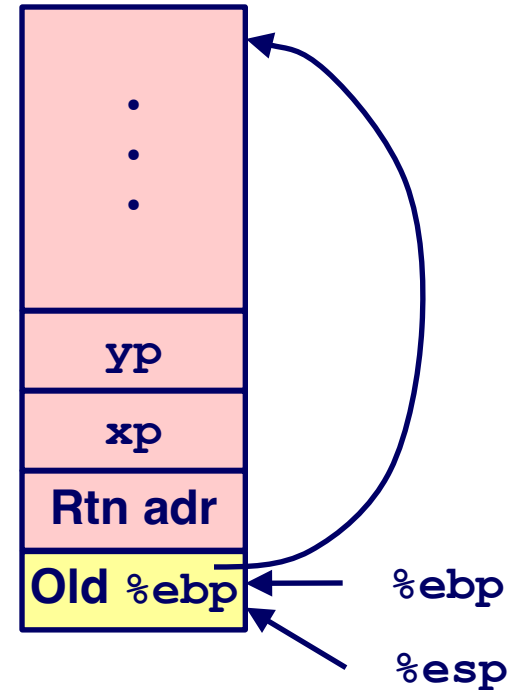
```
pushl %ebp  
movl %esp,%ebp  
pushl %ebx
```

# swap () Setup #2

## Entering Stack



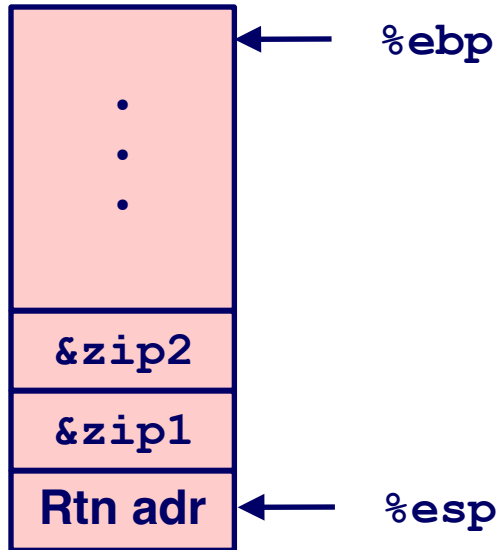
## Resulting Stack



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

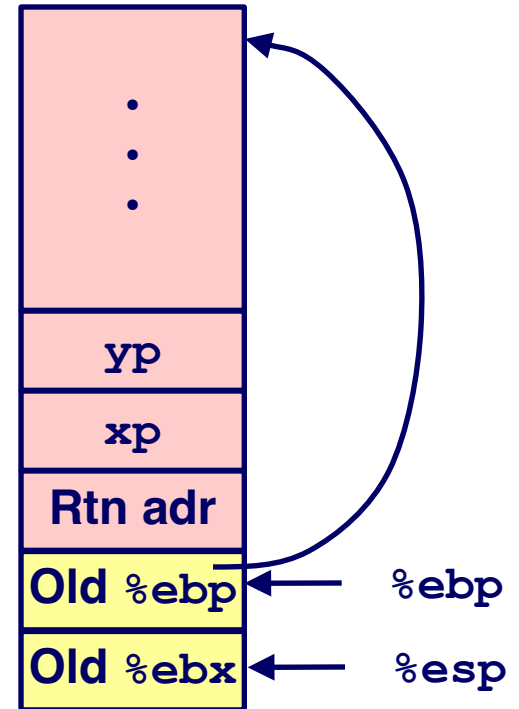
# swap () Setup #3

## Entering Stack



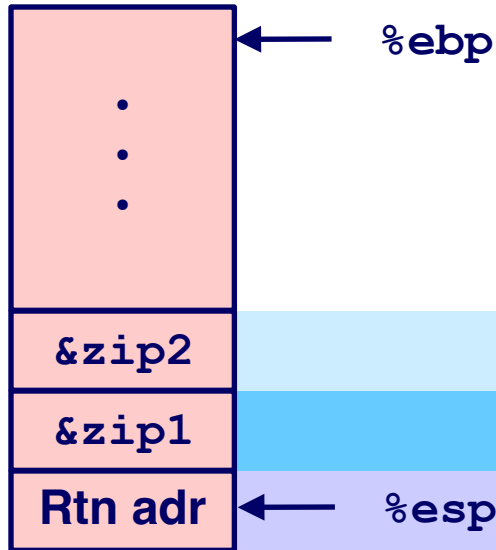
```
swap:  
    pushl %ebp  
    movl %esp,%ebp  
    pushl %ebx
```

## Resulting Stack



# Effect of swap () Setup

## Entering Stack



Offset  
(relative to  $\%ebp$ )

12

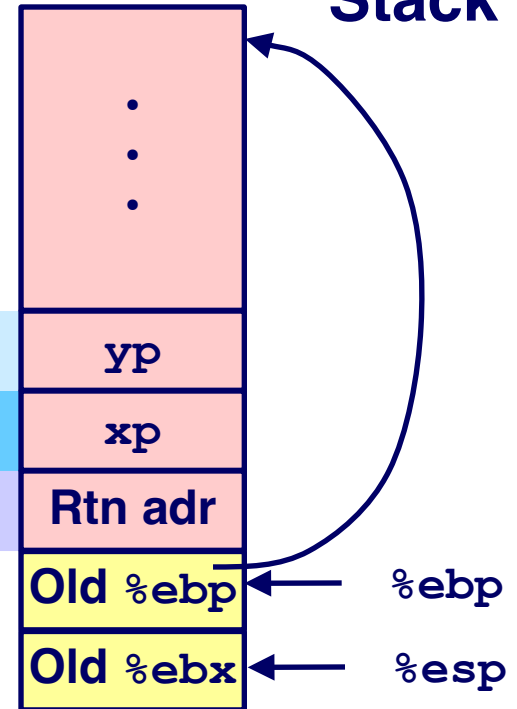
8

4

0

-4

## Resulting Stack



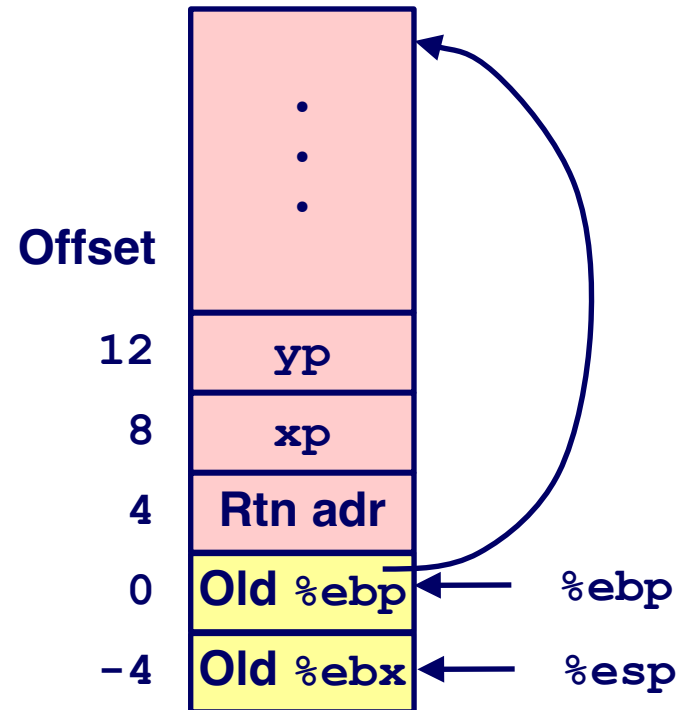
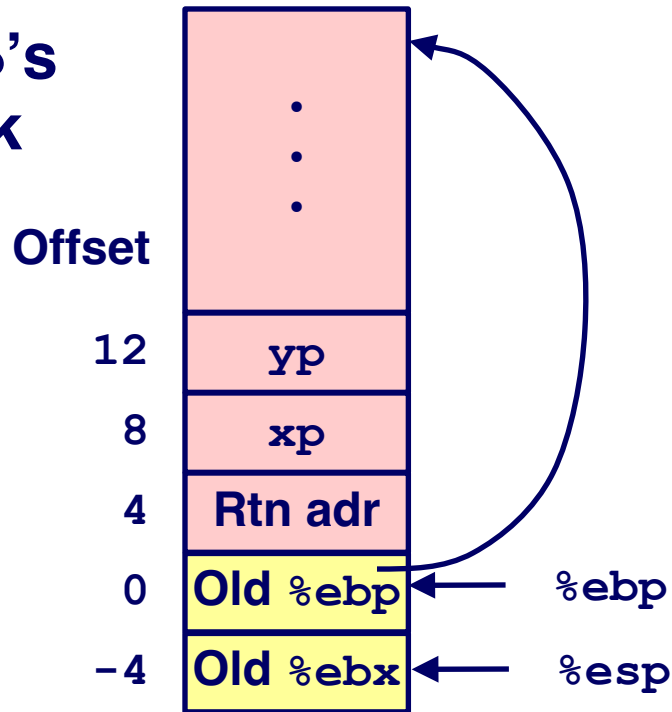
```

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

} Body

# swap () Finish #1

swap's  
Stack



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

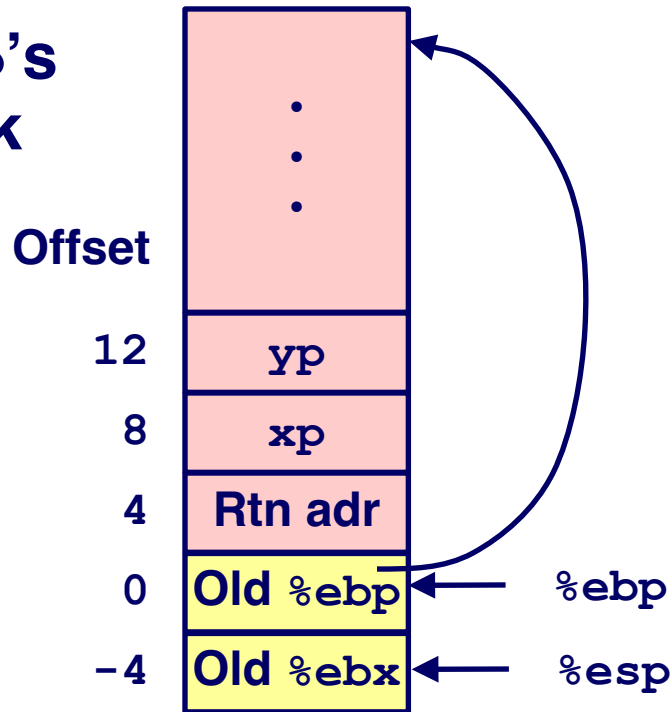
## Observation

- Restoring saved register %ebx
  - “Hold that thought”

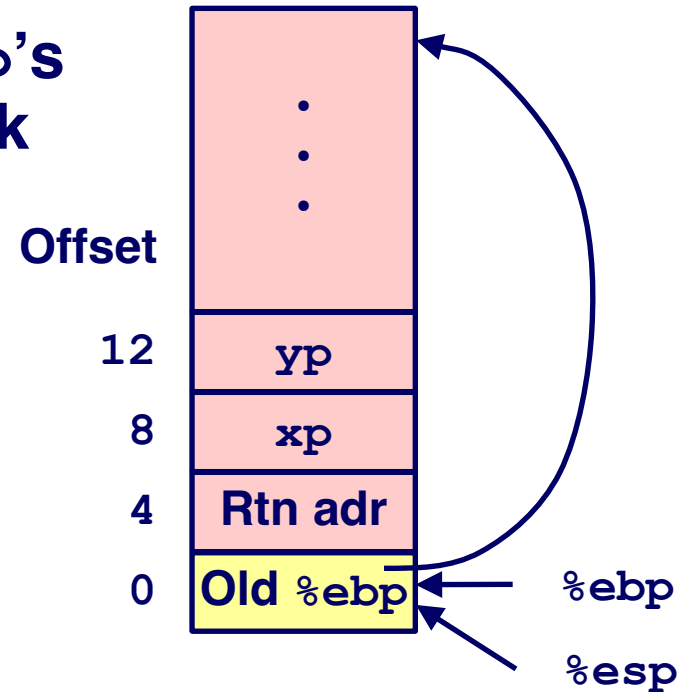


# swap () Finish #2

swap's  
Stack



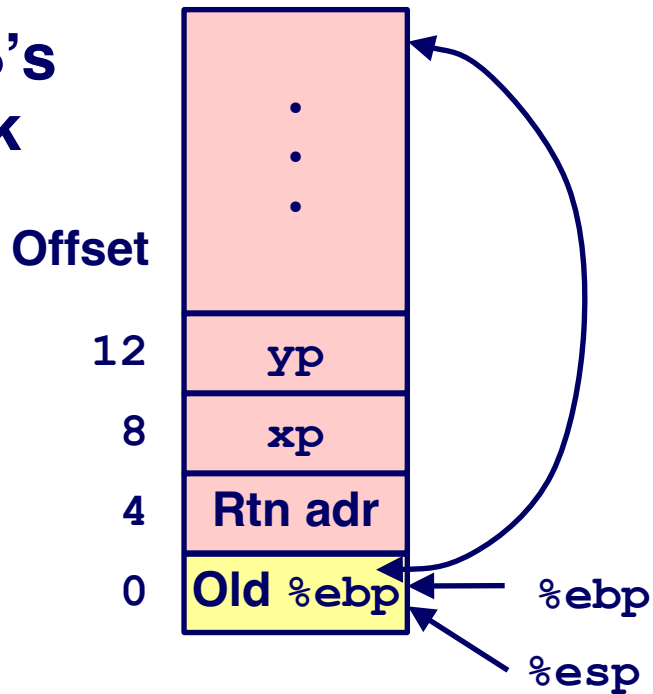
swap's  
Stack



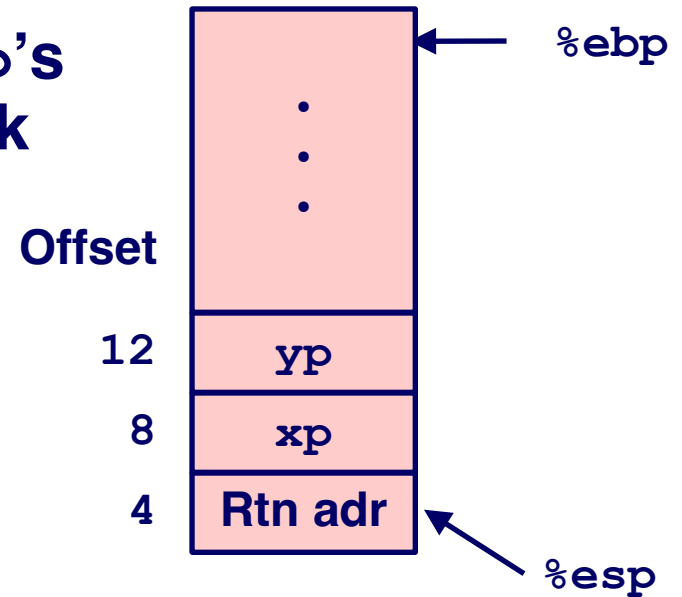
```
movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret
```

# swap () Finish #3

swap's Stack

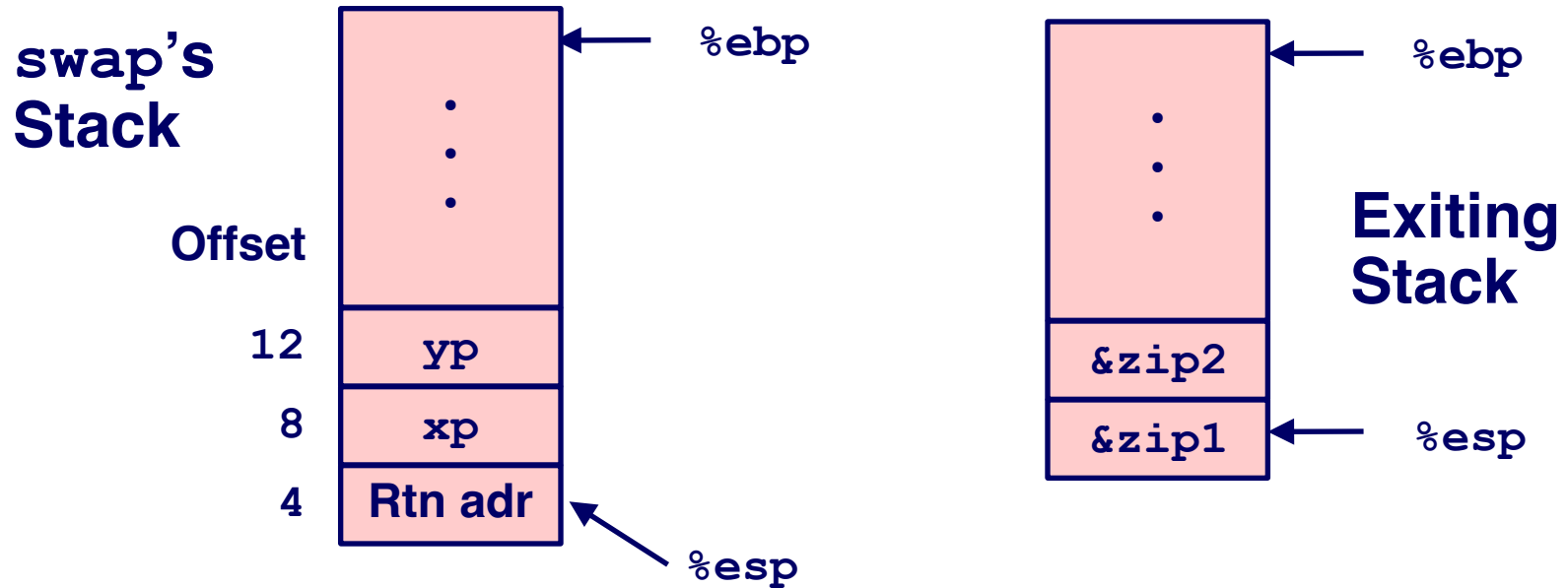


swap's Stack



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

# swap () Finish #4



## Observation/query

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

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

# 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 $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 a register be used for temporary storage?**

## Definitions

- “Caller Save” register
  - Caller saves temporary in its frame before calling
- “Callee Save” register
  - Callee saves temporary in its frame before using

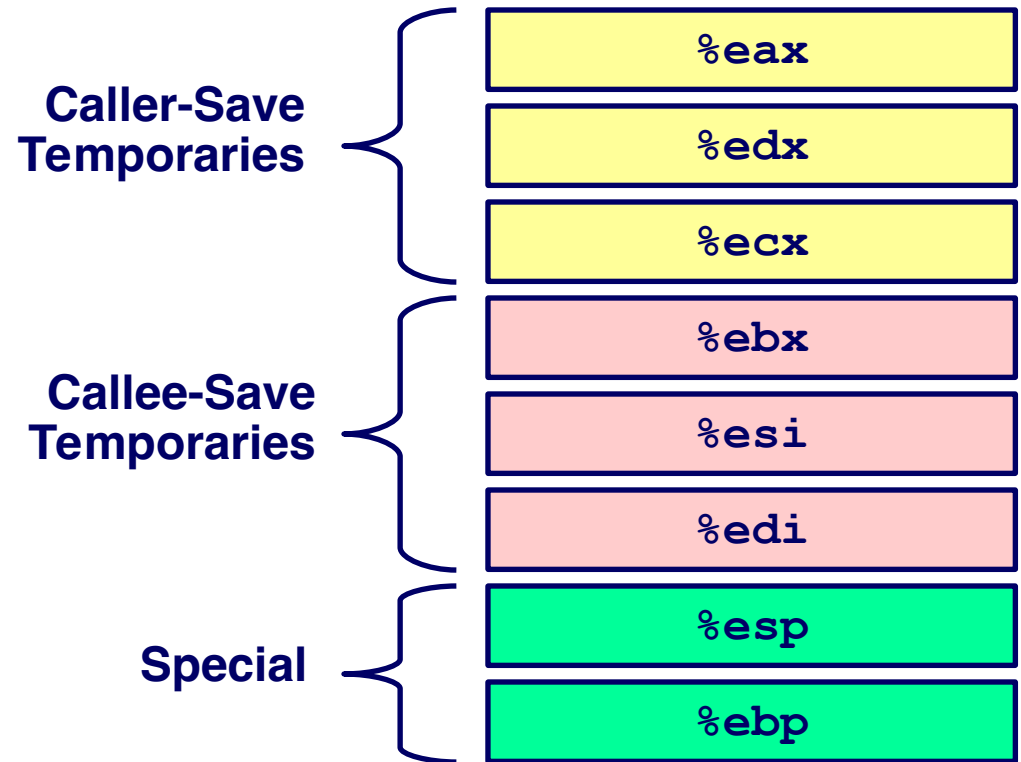
## Conventions

- Which registers are caller-save, callee-save?

# 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 holds return value



# Stack Summary

## Stack makes recursion work

- Private storage for each *instance* of procedure call
  - Instantiations don't clobber each other
  - Addressing of locals + arguments can be relative to stack positions
- Can be managed by stack discipline
  - Procedures return in inverse order of calls

## IA32 procedures: instructions + conventions

- `call` / `ret` instructions mix `%eip`, `%esp` in a fixed way
- Register usage conventions
  - Caller / Callee save
  - `%ebp` and `%esp`
- Stack frame organization conventions
  - Which argument is pushed first

# Before & After main()

```
int main(int argc, char *argv[]) {
    if (argc > 1) {
        printf("%s\n", argv[1]);
    } else {
        char *av[3] = { 0, 0, 0 };
        av[0] = argv[0];  av[1] = "Fred";
        execvp(av[0], av);
    }
    return (0);
}
```



# The Mysterious Parts

## **argc, argv**

- Strings from one program
- Available while another program is running
- Which part of the memory map are they in?
- How did they get there?

## **What happens when `main()` does “`return(0)`”???**

- There's no more program to run...right?
- Where does the 0 go?
- How does it get there?

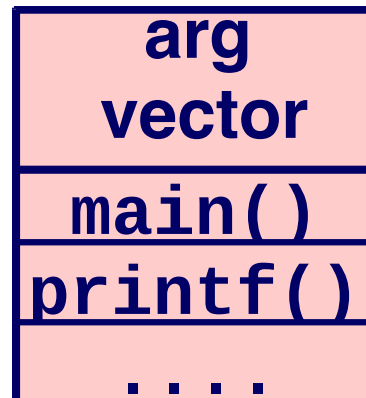
## **410 students should seek to abolish mystery**

- So we will (un)cover each mysterious thing

# The Mysterious Parts

## **argc, argv**

- **Strings from one program**
- **Available while another program is running**
- **Inter-process sharing/information transfer is OS's job**
  - **OS copies strings from old address space to new in exec()**
  - **Traditionally placed “below bottom of stack”**
  - **Other weird things (environment, auxiliary vector) (above argv)**



# The Mysterious Parts

## What happens when `main()` does “`return(0)`”?

- Defined by C standard to have same effect as “`exit(0)`”
- But how??

# The Mysterious Parts

## What happens when `main()` does “`return(0)`”?

- Defined by C standard to have same effect as “`exit(0)`”
- But how??

## The “`main()` wrapper”

- Receives `argc`, `argv` from OS
- Calls `main()`, then calls `exit()`
- Provided by C library, traditionally in “`crt0.s`”
- Often has a “strange” name (not a legal C function name)

```
/* not actual code */
```

```
void ~~main(int argc, char *argv[]) {  
    exit(main(argc, argv));  
}
```

# Project 0 - “Stack Crawler”

## **C/Assembly function**

- Can be called by any C function
- Prints stack frames in a symbolic way

**---Stack Trace Follows---**

**Function fun3(c='c', d=2.090000), in**

**Function fun2(f=35.000000), in**

**Function fun1(count=0), in**

**Function fun1(count=1), in**

**Function fun1(count=2), in**

**...**

# Project 0 - “Stack Crawler”

## Conceptually easy

- Calling convention specifies layout of stack
- Stack is “just memory” - C happily lets you read & write

## Key questions

- How do I know `0x80334720` is “fun1”?
- How do I know `fun3()`'s second parameter is called “d”?

# Project 0 “Data Flow”

fun.c

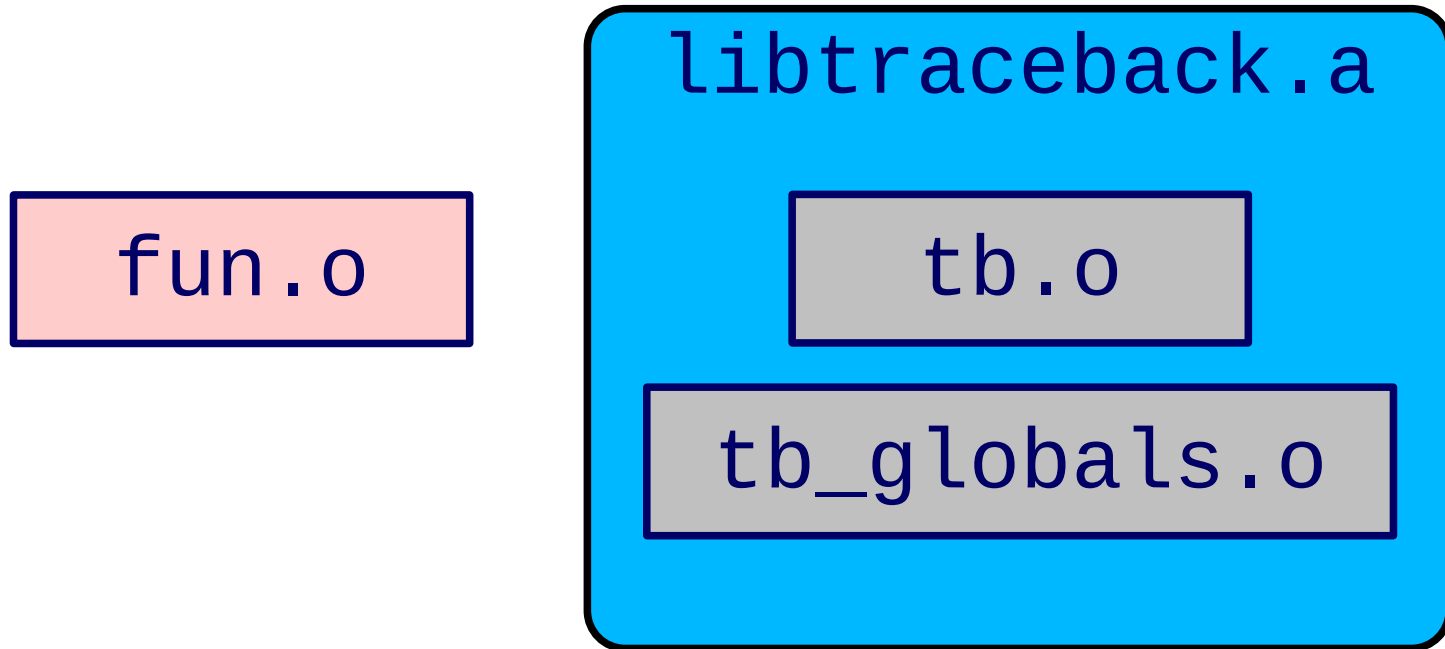
tb.c

tb\_globals.c

symbol-table array  
many slots (blank)

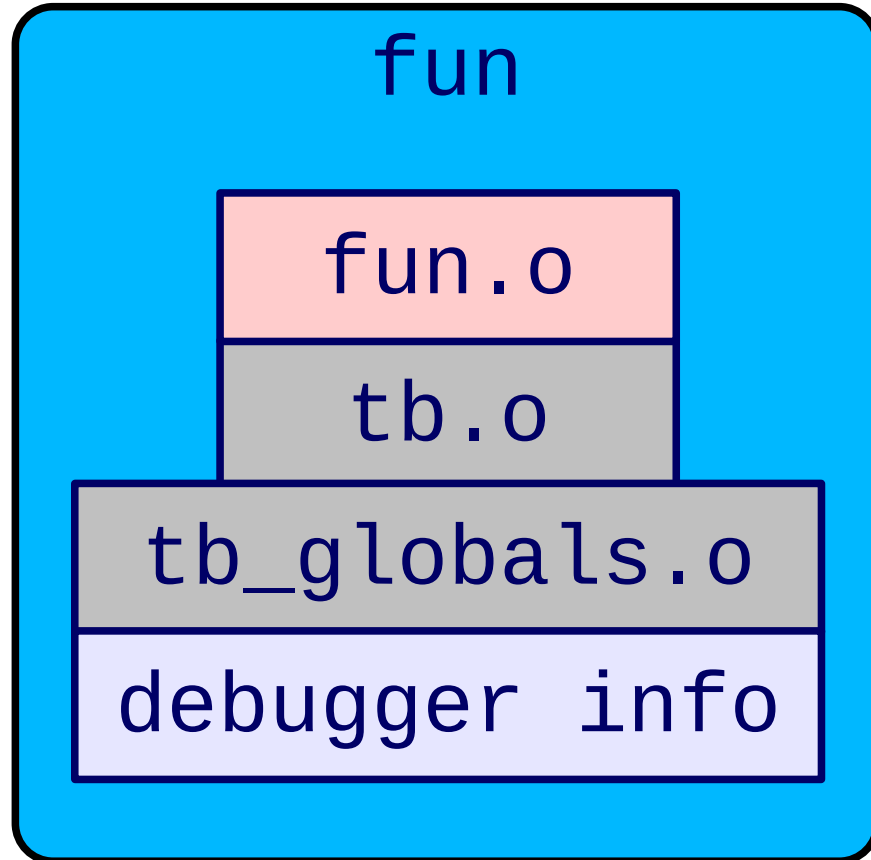


# Project 0 “Data Flow” - Compilation

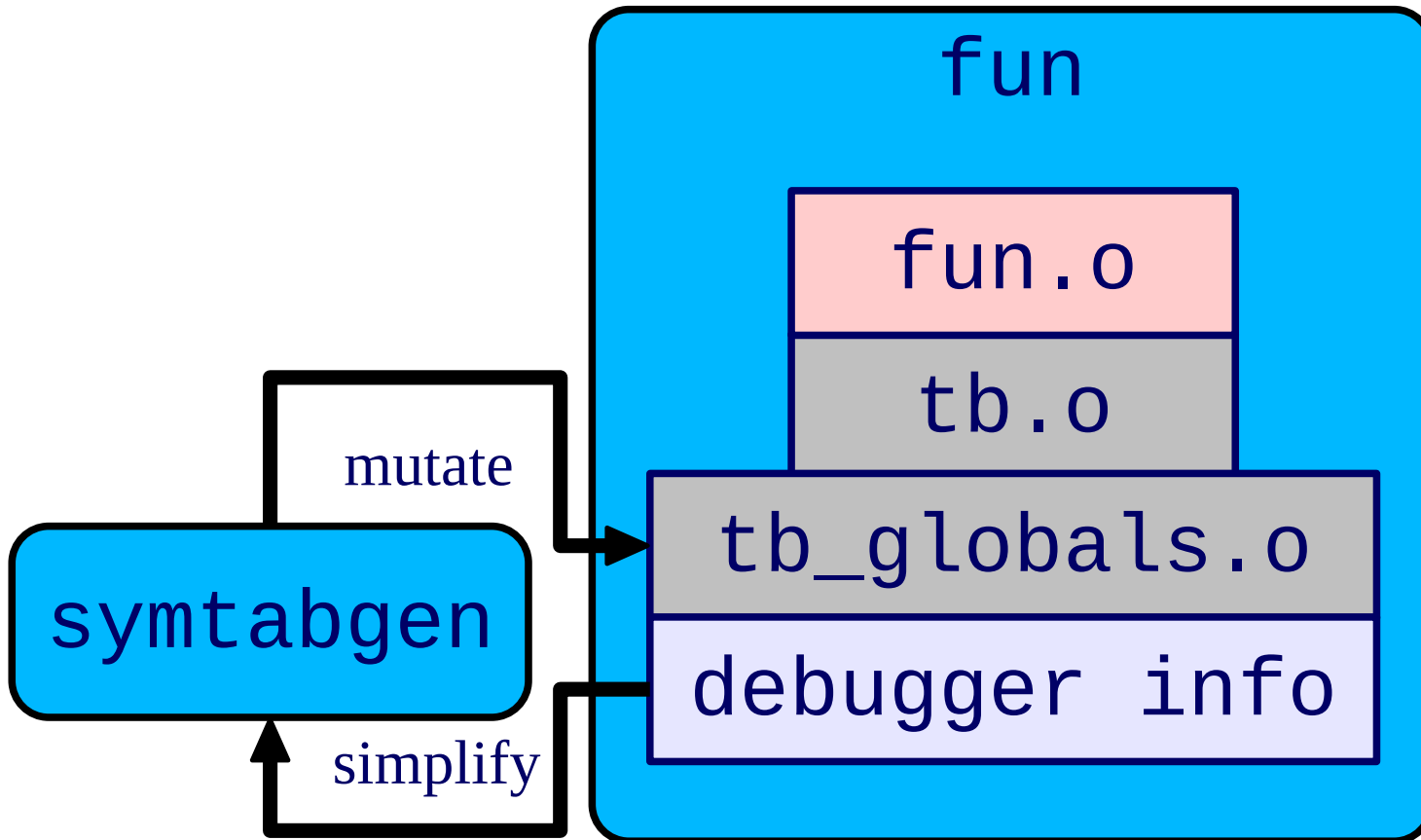




# Project 0 “Data Flow” - Linking



# Project 0 “Data Flow” - P0 “Post-Linking”



# Summary

**Review of stack knowledge**

**What makes `main()` special**

**Project 0 overview**

Look for handout this afternoon/evening

Please read *all of the handout* before asking questions!

Meanwhile, *reading the syllabus* is timely!

**Start interviewing Project 2/3/4 partners!**