15-410

"Way easier than when we were students"

Stack Discipline
Jan. 14, 2004

Dave Eckhardt

Bruce Maggs

Review slides from 15-213 originally developed by Randy Bryant and Dave O'Halloran.

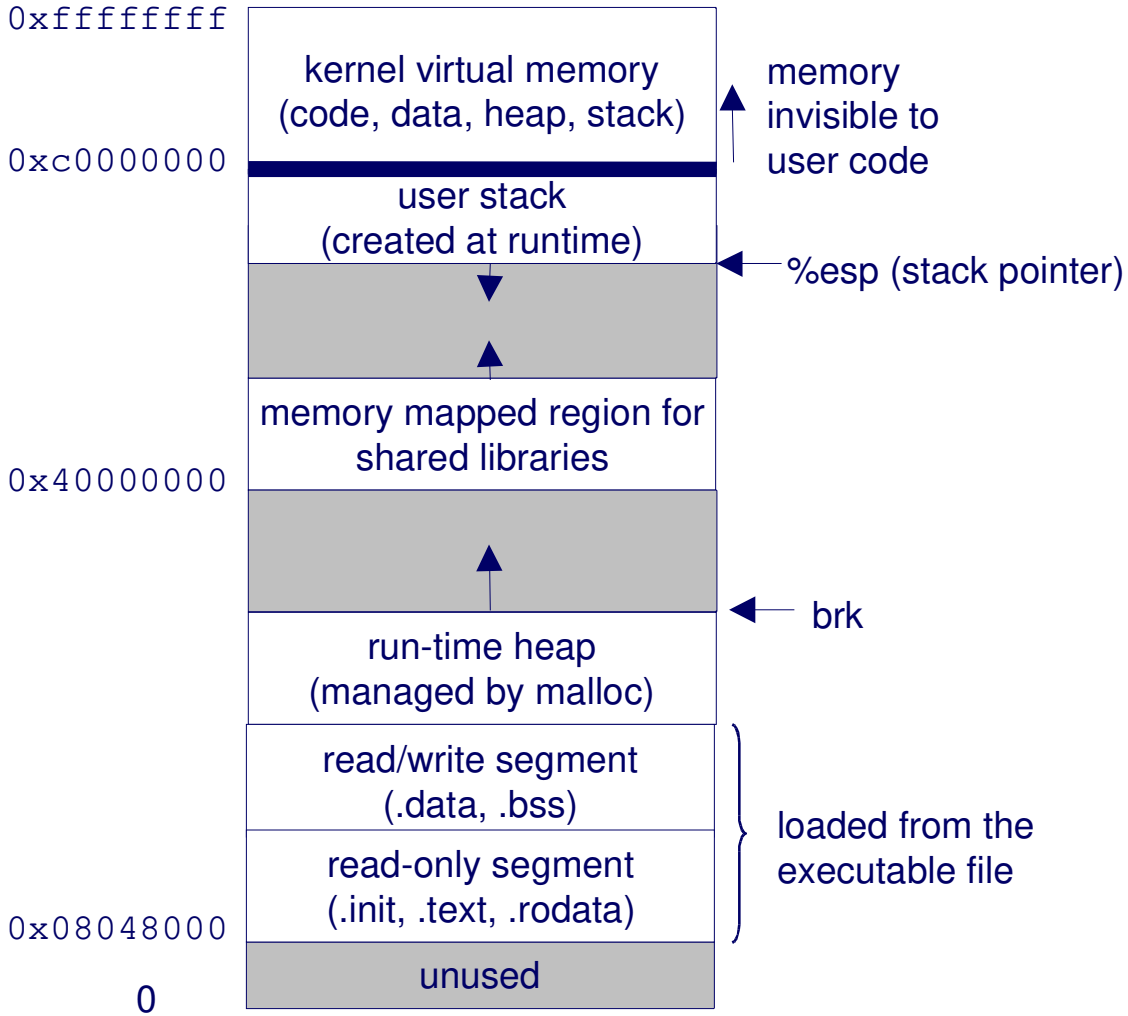
Outline

Topics

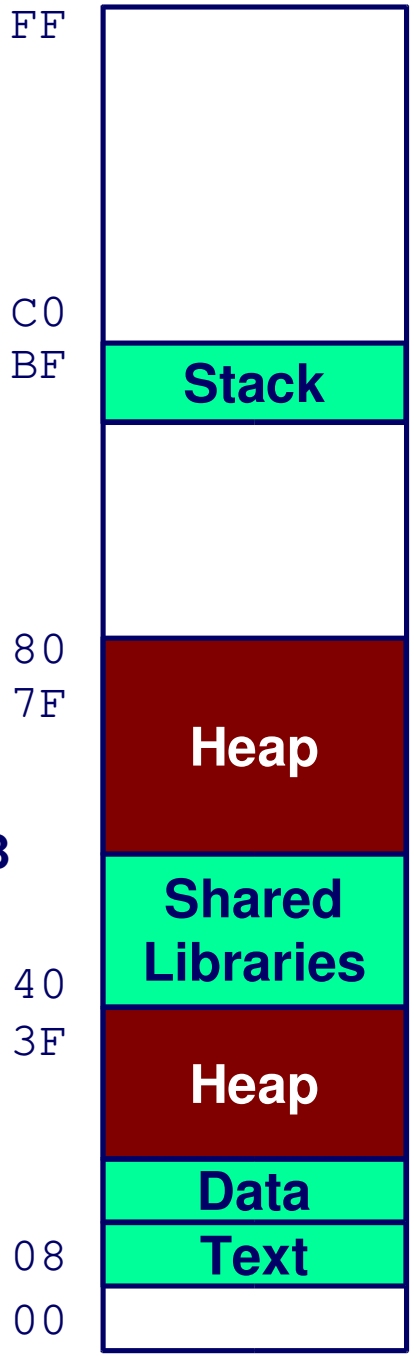
- Process memory model
- IA32 stack organization
- Register saving conventions
- Before & after `main()`
- Project 0

Private Address Spaces

Each process has its own private address space.



Linux Memory Layout



Upper 2 hex digits of address

Red Hat v. 6.2 ~1920MB memory limit

Stack

- Runtime stack (8MB limit)

Heap

- Dynamically allocated storage
- When call `malloc`, `calloc`, `new`

Shared/Dynamic Libraries aka Shared Objects

- Library routines (e.g., `printf`, `malloc`)
- Linked into object code when first executed
- Windows has "DLLs" (semantic differences)

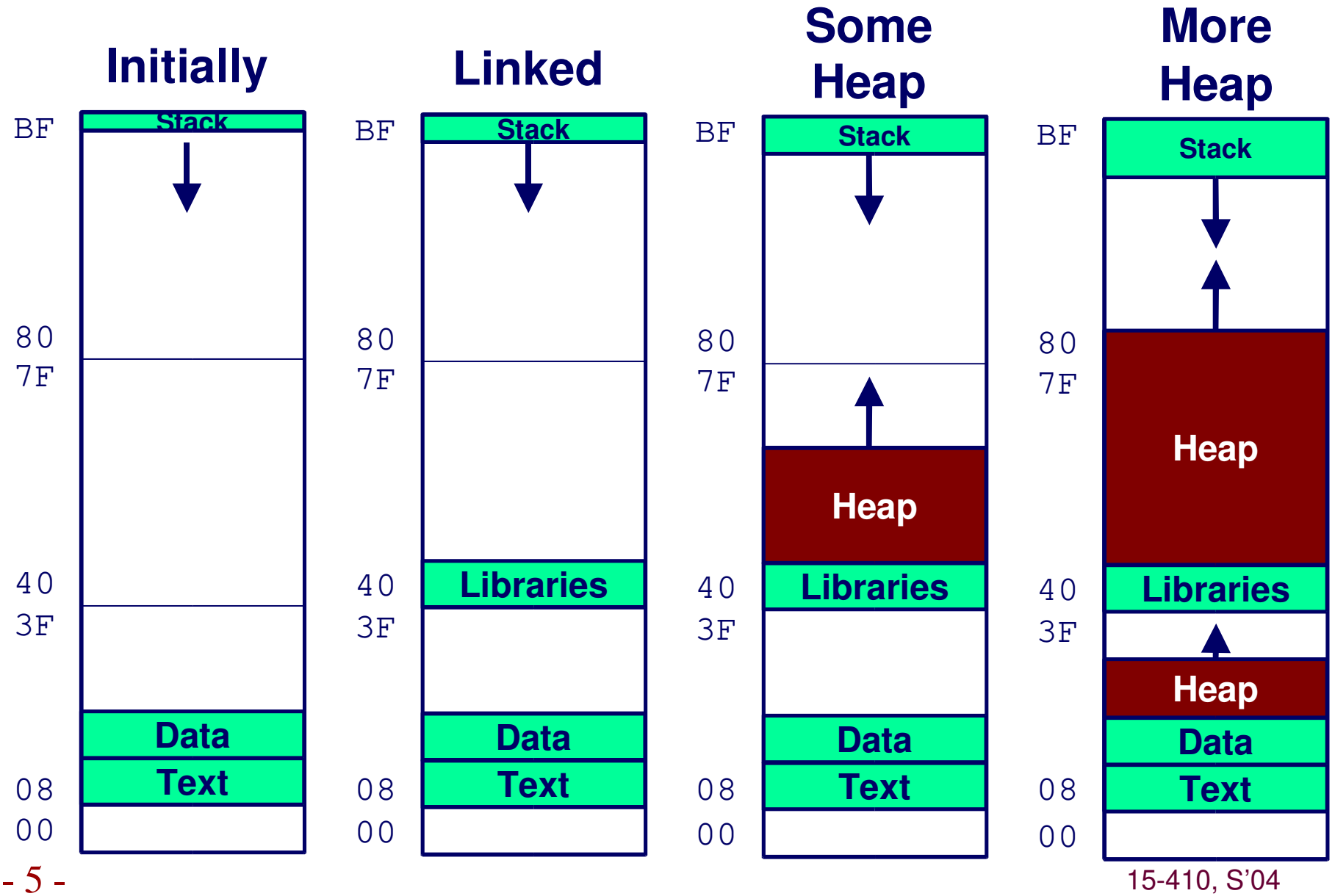
Data, BSS

- Statically allocated data
- E.g., arrays & strings declared in code

Text, RODATA

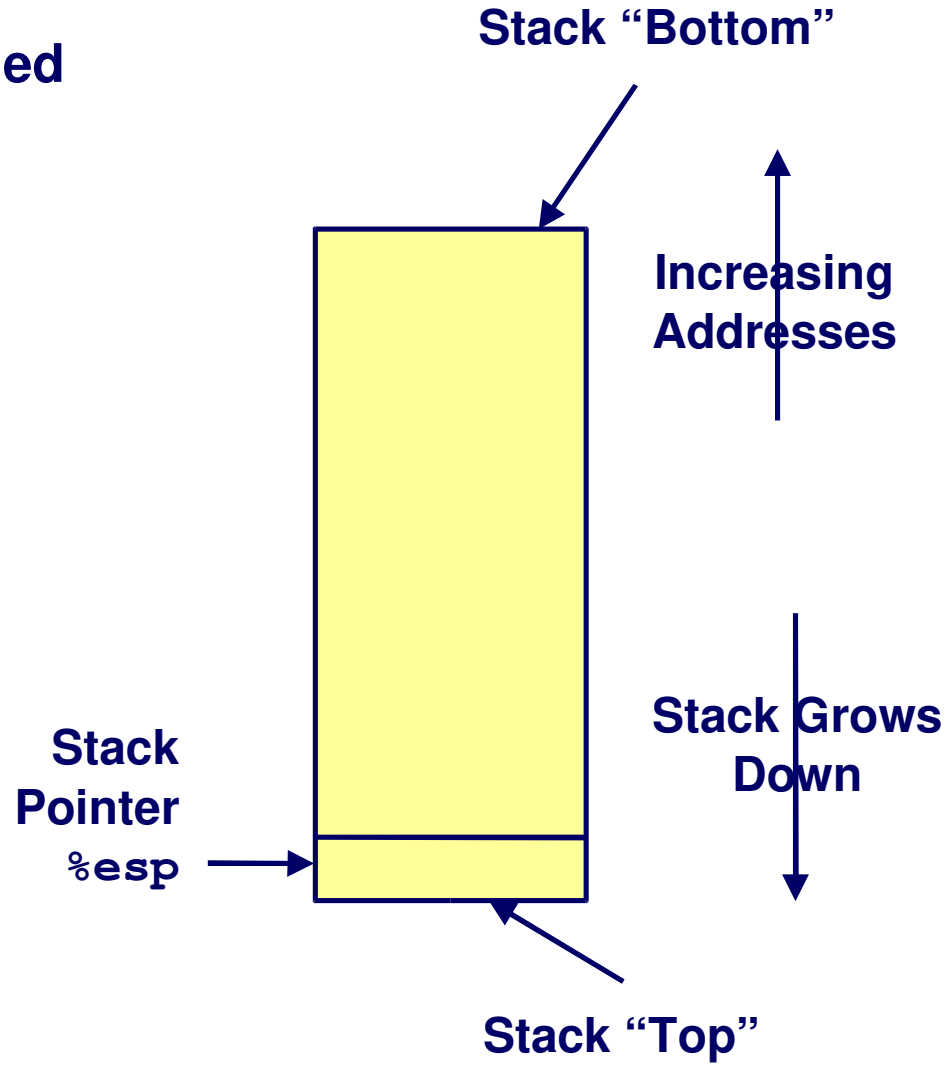
- Text - Executable machine instructions
- RODATA – Read-only (e.g., "const")

Linux Memory Allocation



IA32 Stack

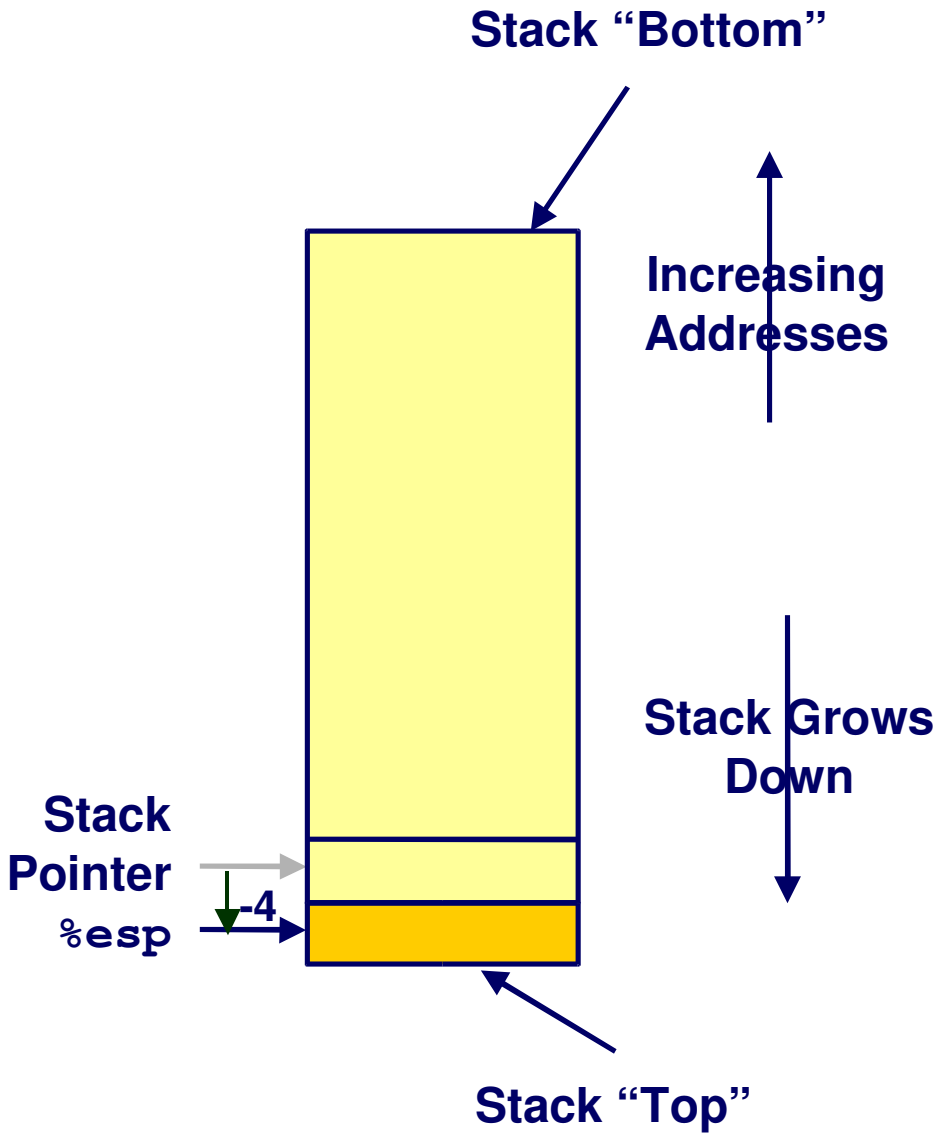
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
  - address of top element



IA32 Stack Pushing

Pushing

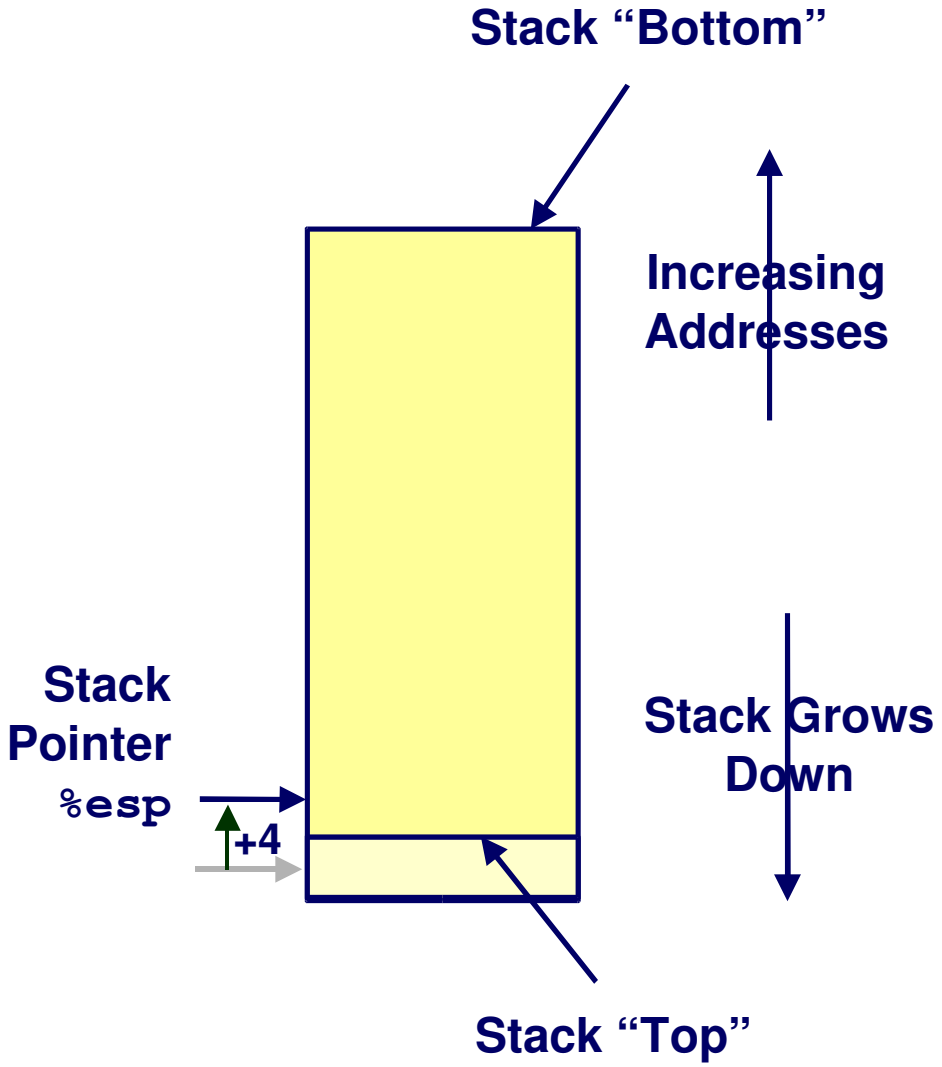
- `pushl` *Src*
- Fetch operand at *Src*
- Decrement `%esp` by 4
- Write operand at address given by `%esp`



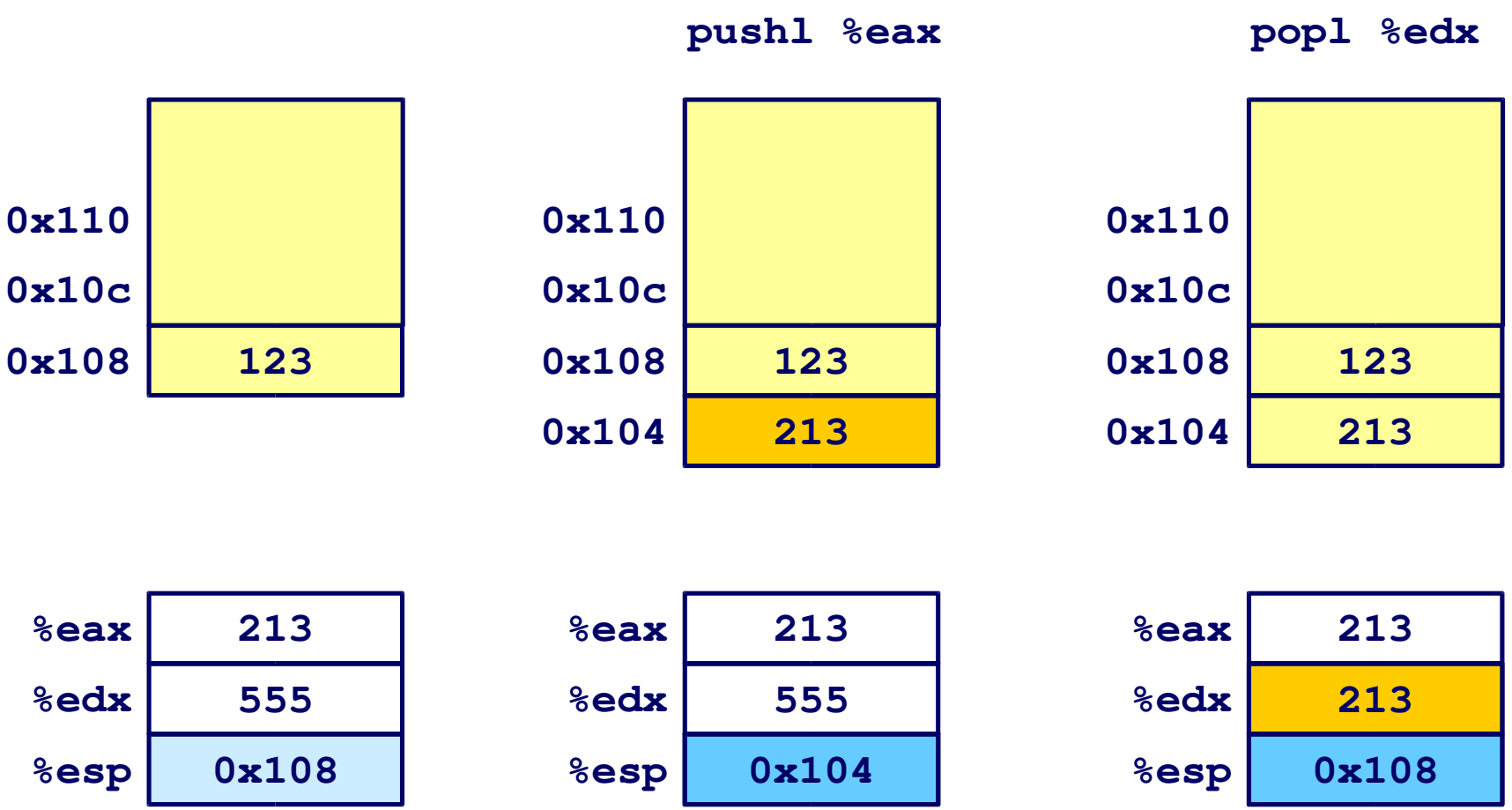
IA32 Stack Popping

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`*

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`

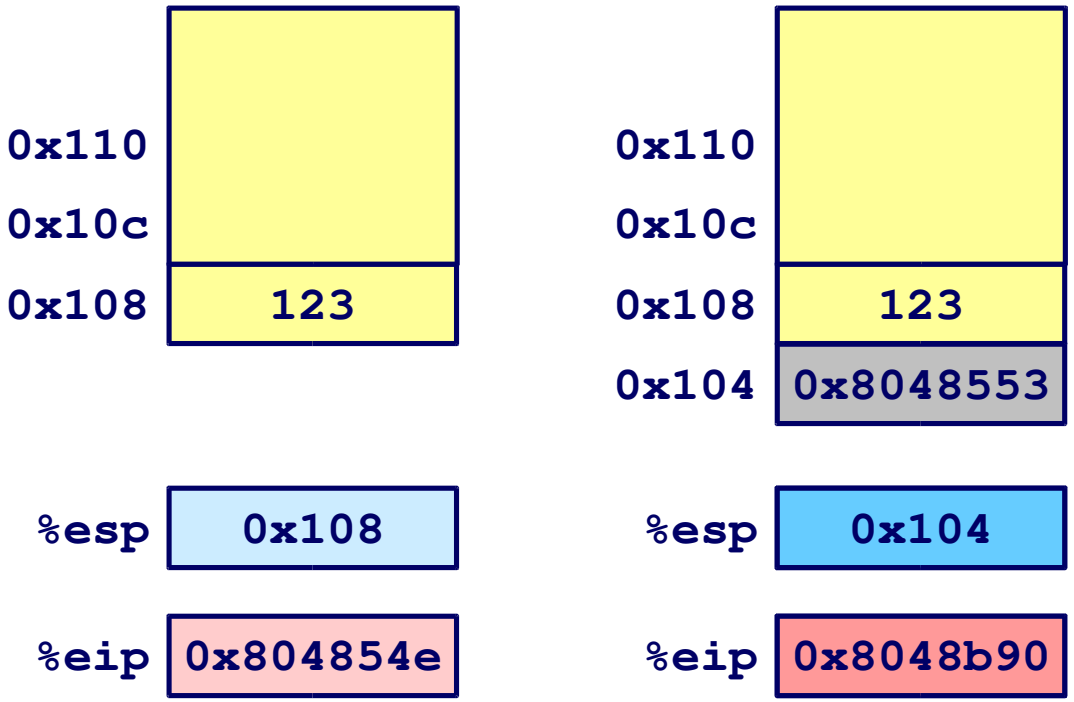
Procedure return:

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

Procedure Call Example

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

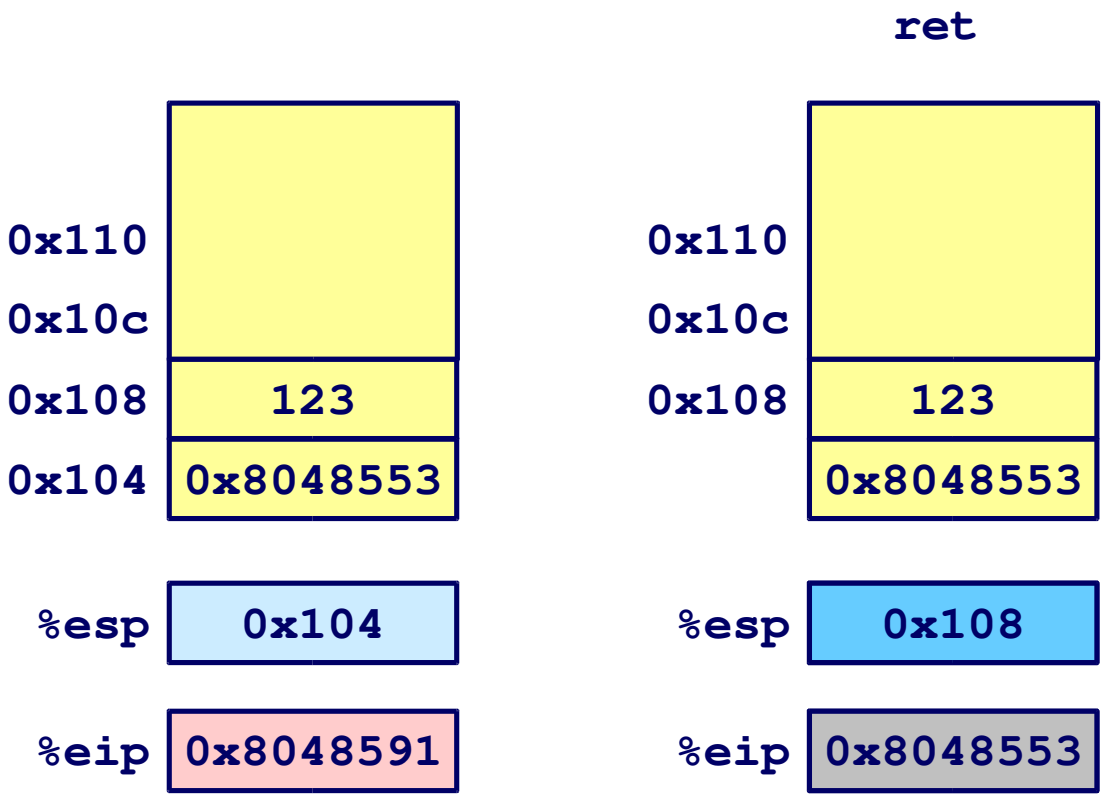
`call 8048b90`



`%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 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

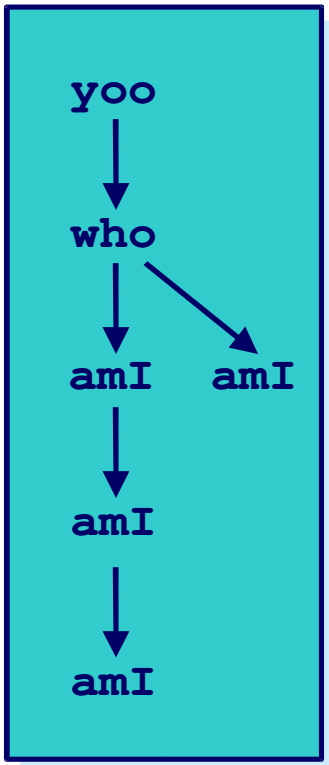
Code Structure

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

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

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

Call Chain

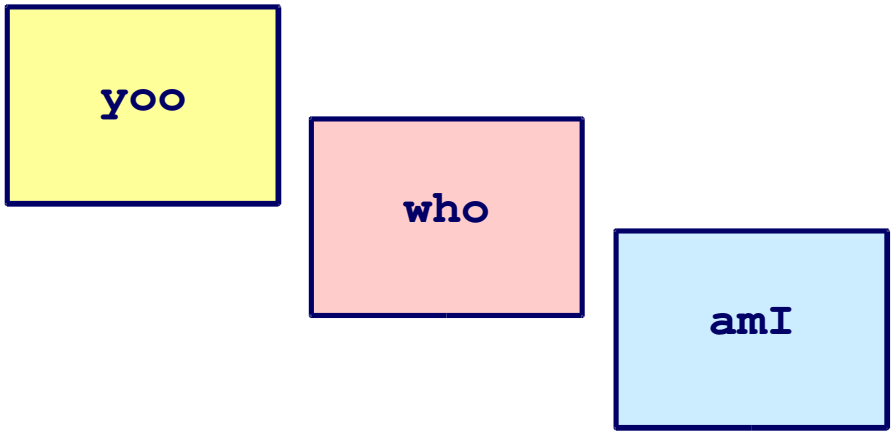


- Procedure `amI` recursive

Stack Frames

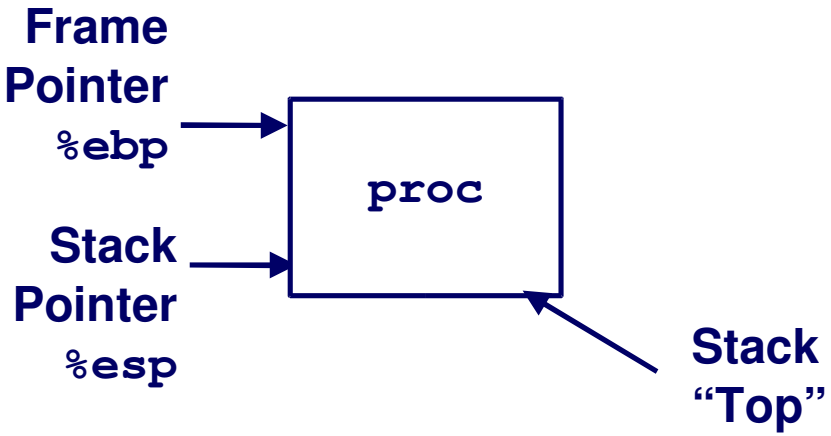
Contents

- 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

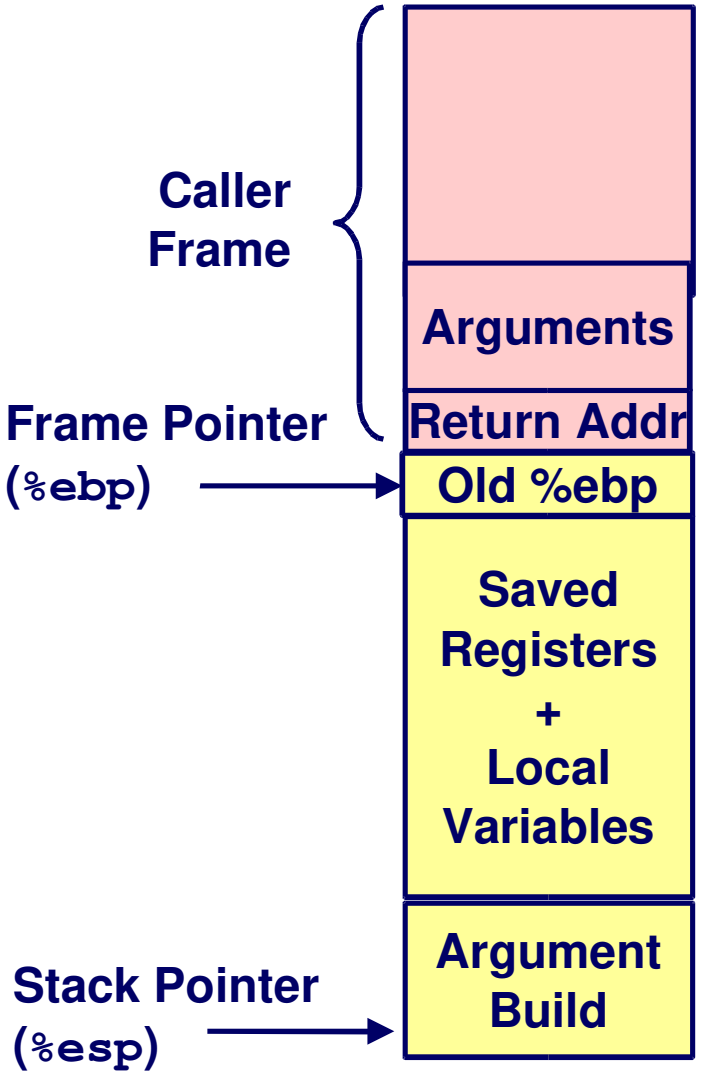
IA32/Linux Stack Frame

Current Stack Frame ("Top" to Bottom)

- Parameters for function about to call
  - "Argument build"
- 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



swap

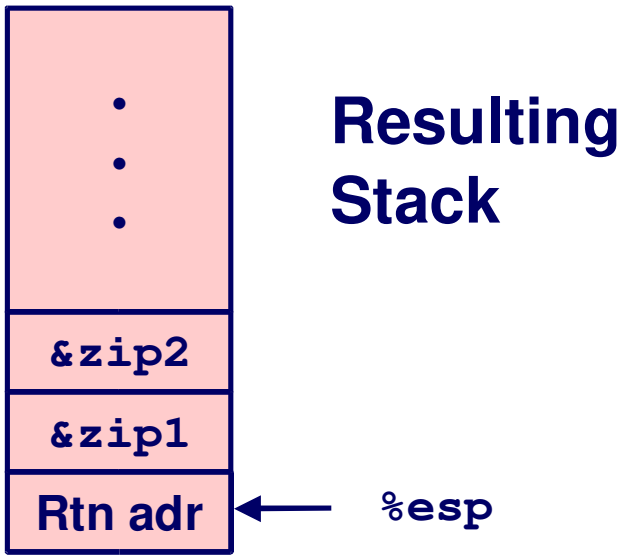
```
int zip1 = 15213;
int zip2 = 91125;

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
    • • •
```



Resulting Stack

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
```

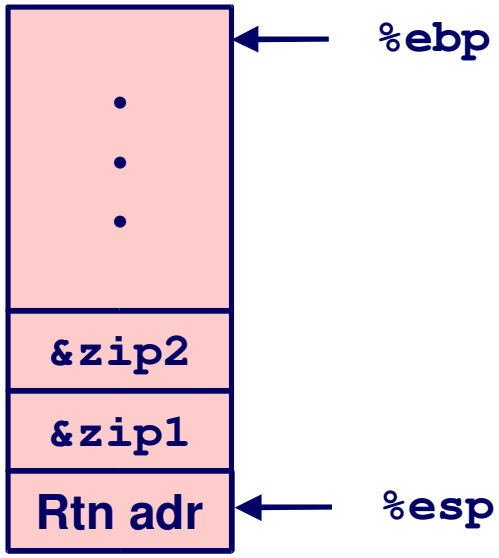
Set Up

Body

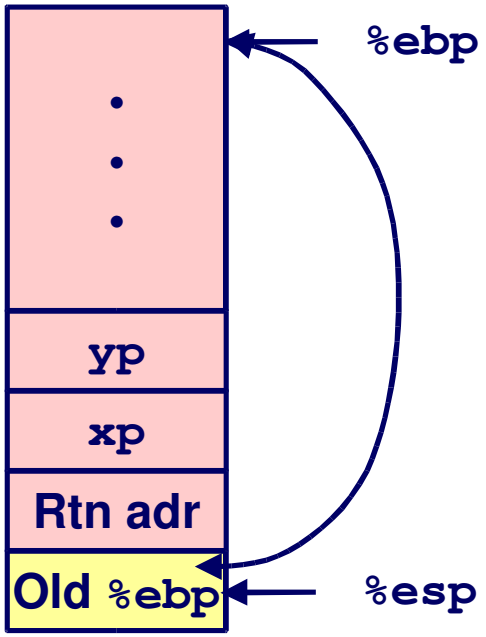
Finish

swap Setup #1

Entering Stack



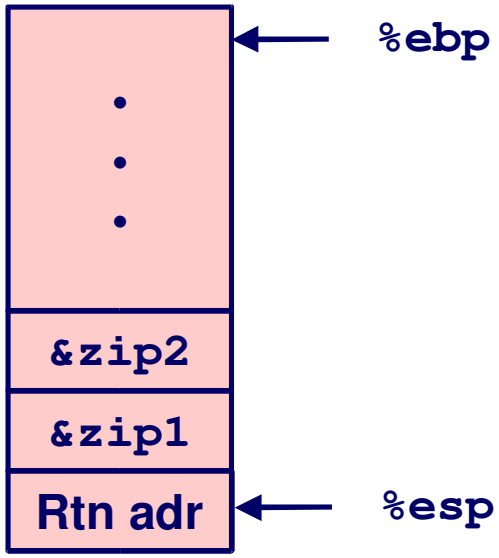
Resulting Stack



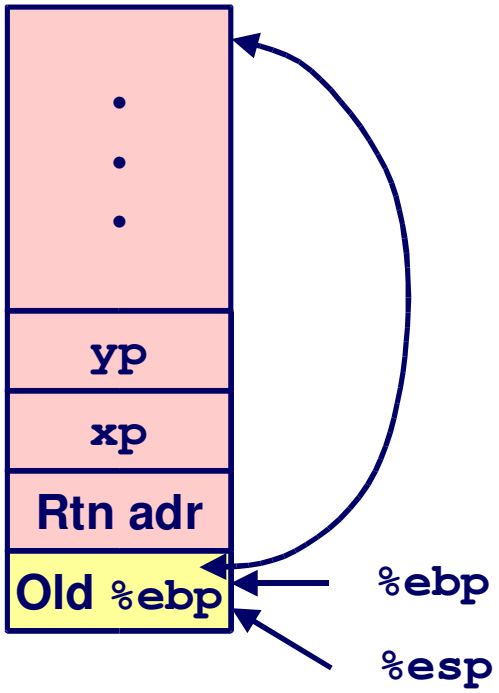

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #2

Entering Stack



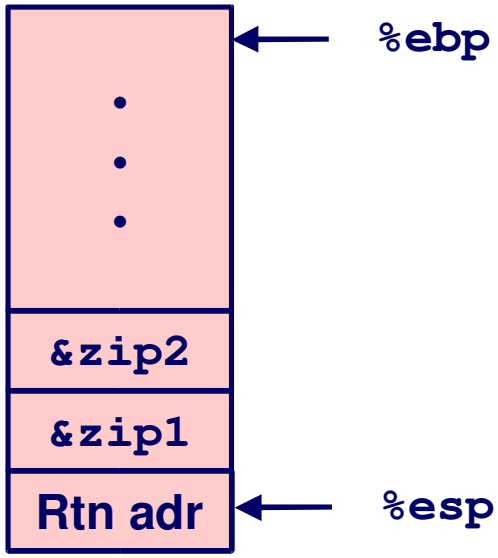
Resulting Stack



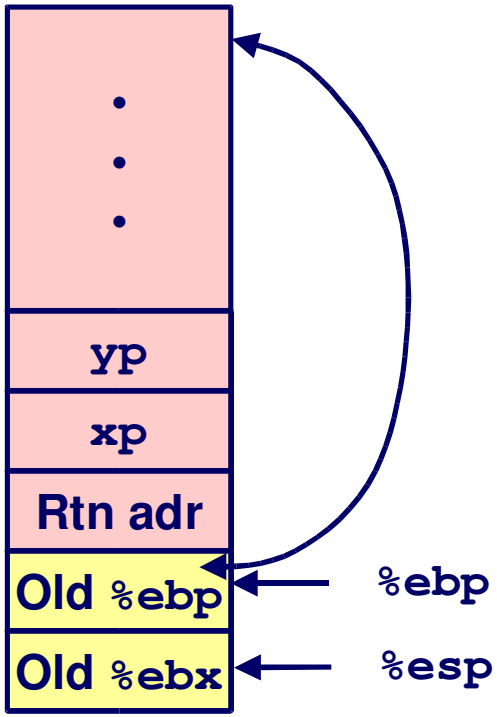

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #3

Entering Stack

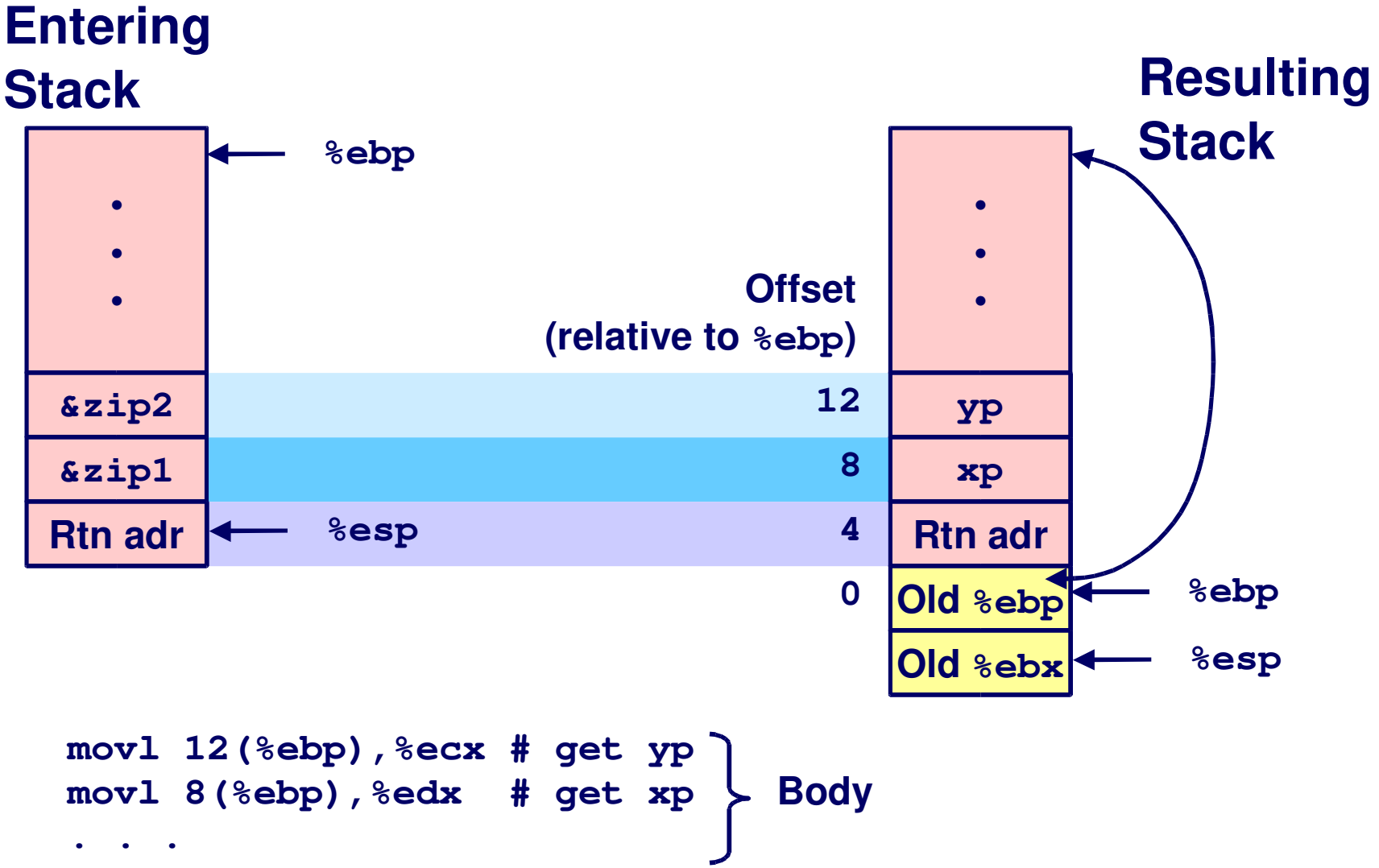


Resulting Stack




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

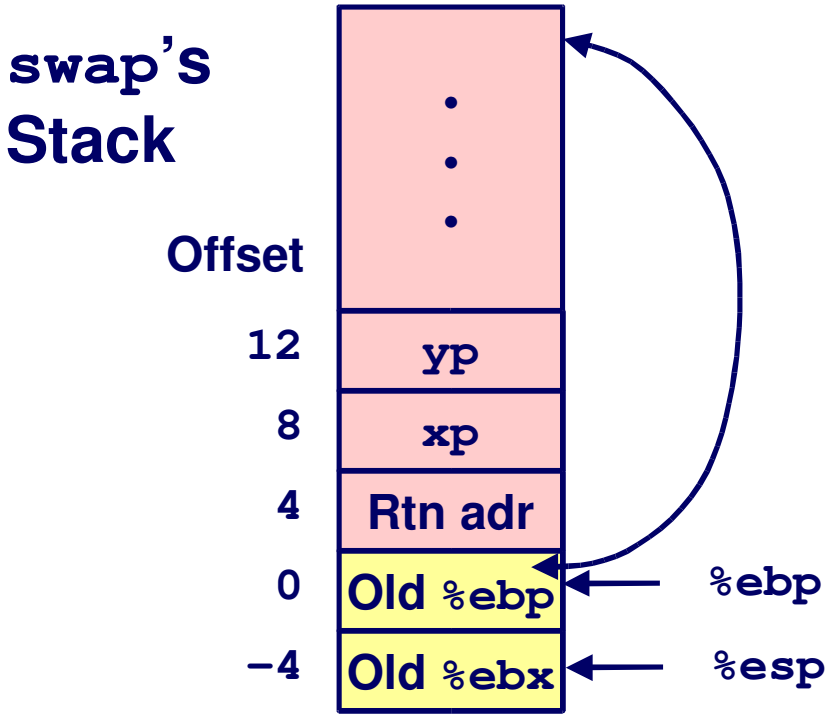
Effect of `swap` Setup

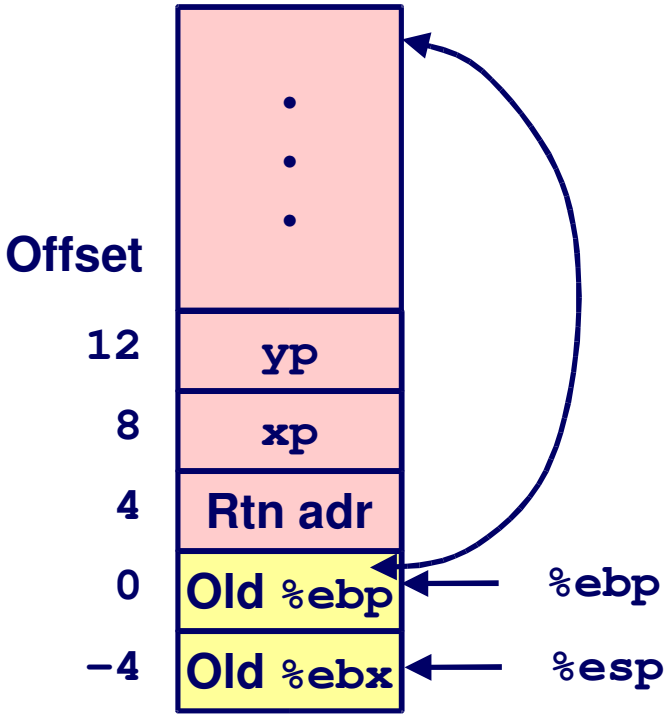



```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

} Body

swap Finish #1



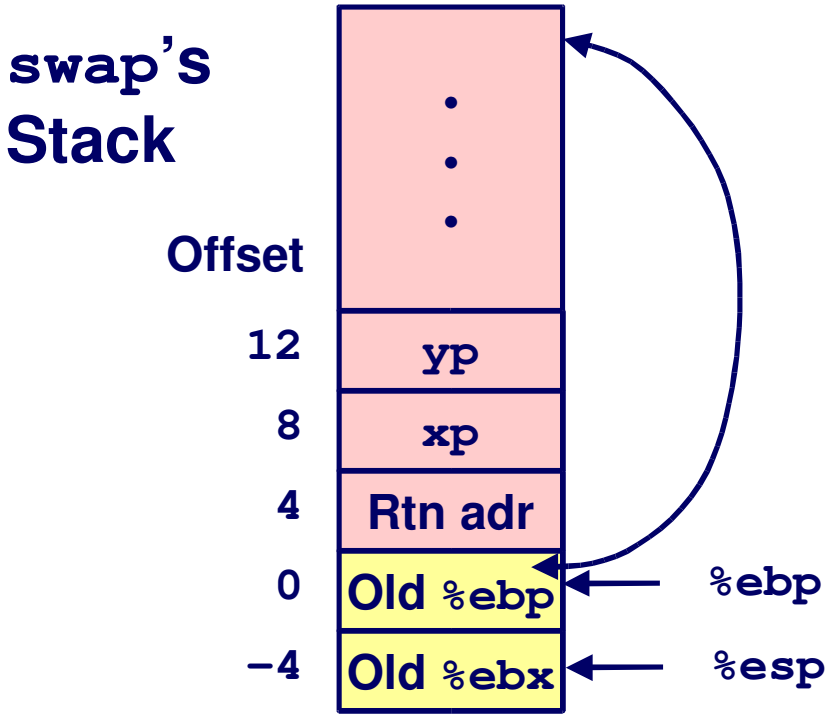


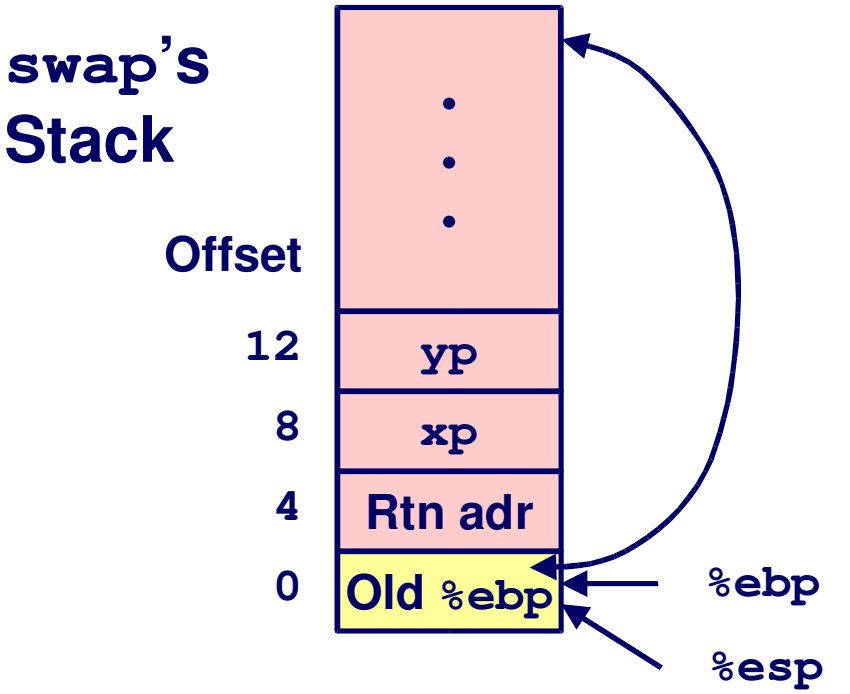

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

- Saved & restored register `%ebx`

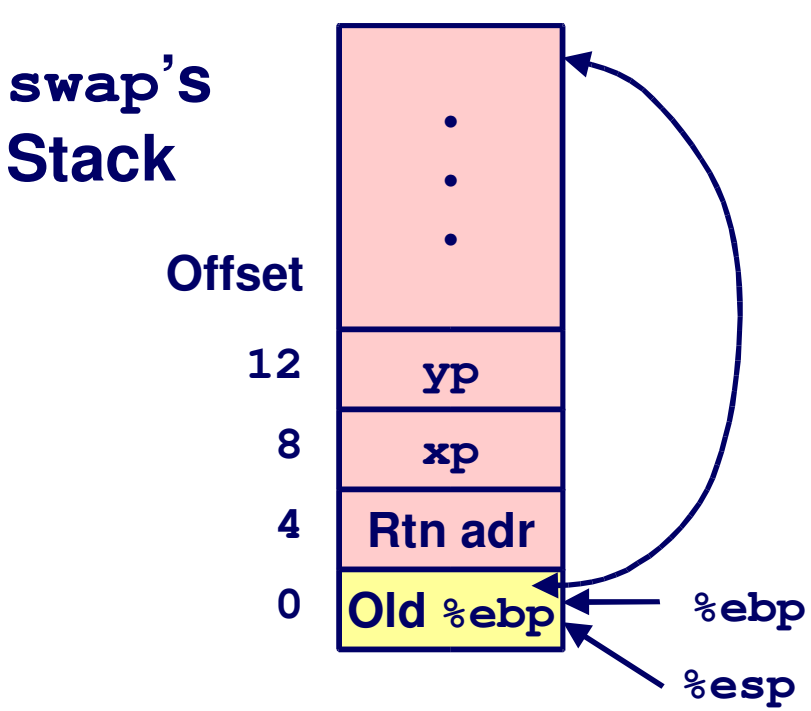
swap Finish #2

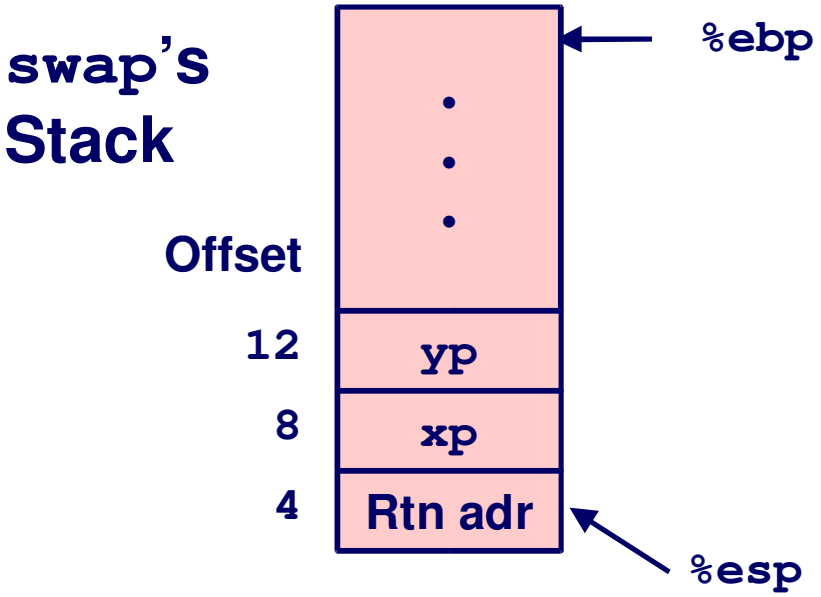





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

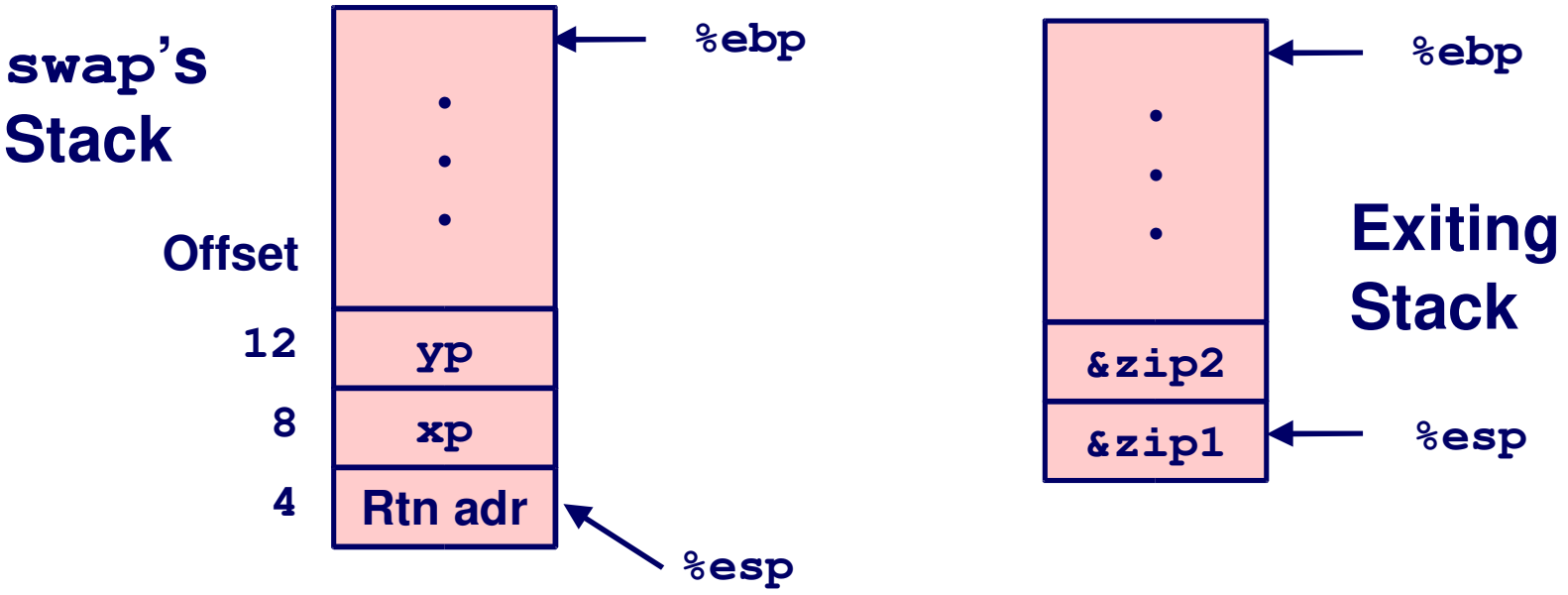
swap Finish #3






```
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?

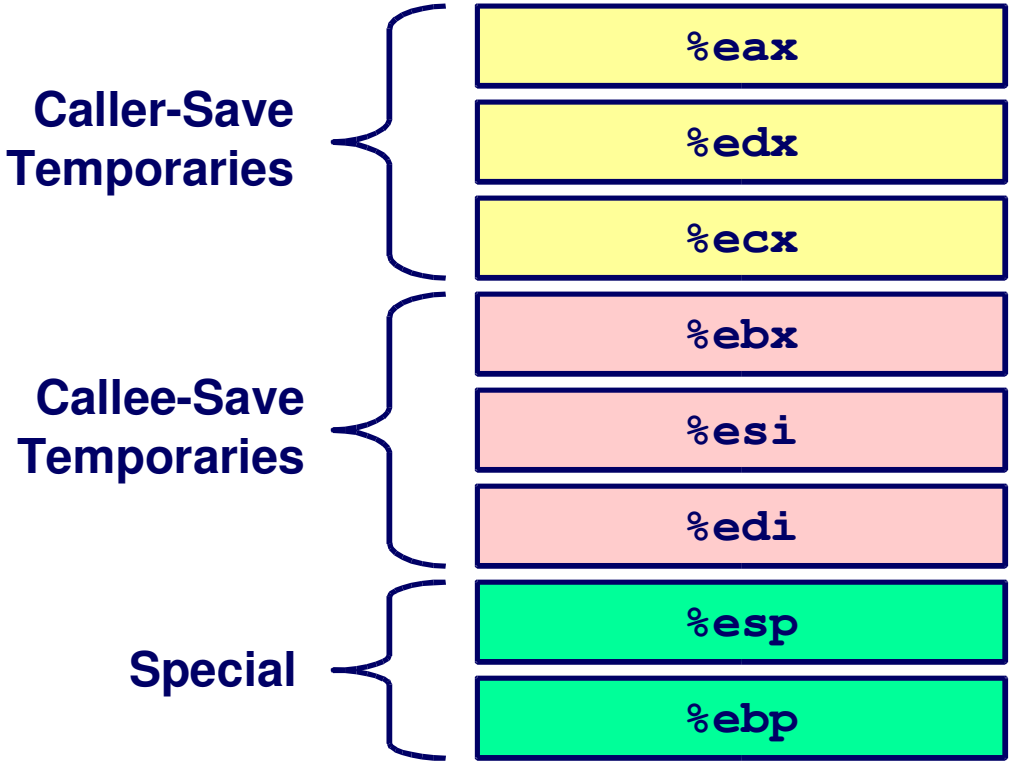
Conventions

- "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



Stack Summary

The 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 Combination of Instructions + Conventions

- Call / Ret instructions
- Register usage conventions
  - Caller / Callee save
  - `%ebp` and `%esp`
- Stack frame organization conventions

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 (1);
}
```

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(1)`" ?

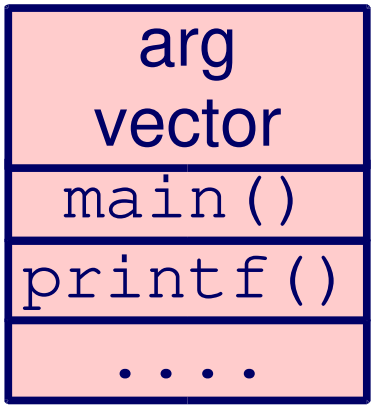
- There's no more program to run...right?
- Where does the 1 go?
- How does it get there?

410 students should seek to abolish mystery

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”



The Mysterious Parts

What happens when `main()` does "`return(1)`"???

- Defined to have same effect as "`exit(1)`"

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 actual code */

void ~~main(int argc, char *argv[]) {

  exit(main(argc, argv));

}
```

Project 0 - “Stack Crawler”

C/Assembly function

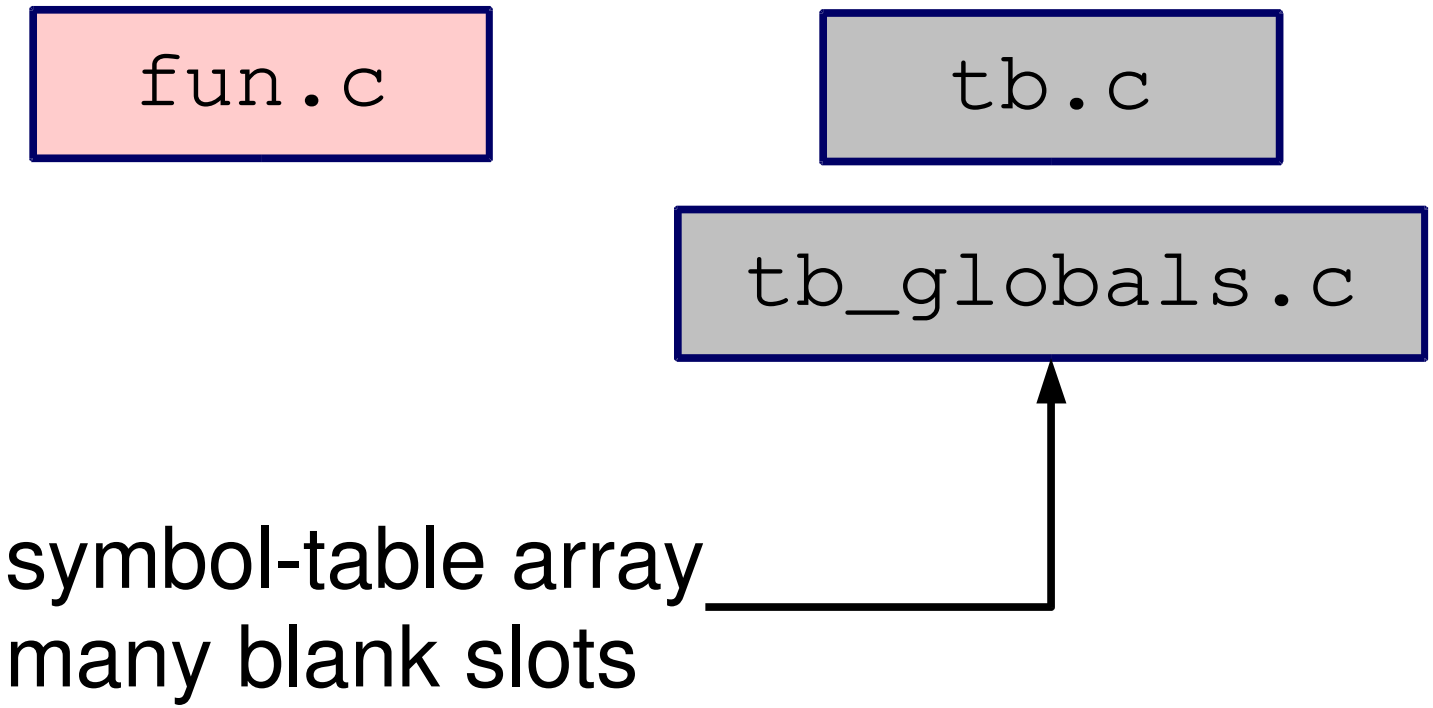
- Can be called by any C function
- Prints stack frames in a symbolic way

```
---Stack Trace Follows---
Function fun3(c='c', d=2.090000d), in
Function fun2(f=35.000000f), in
Function fun1(count=0), in
Function fun1(count=1), in
Function fun1(count=2), in
```

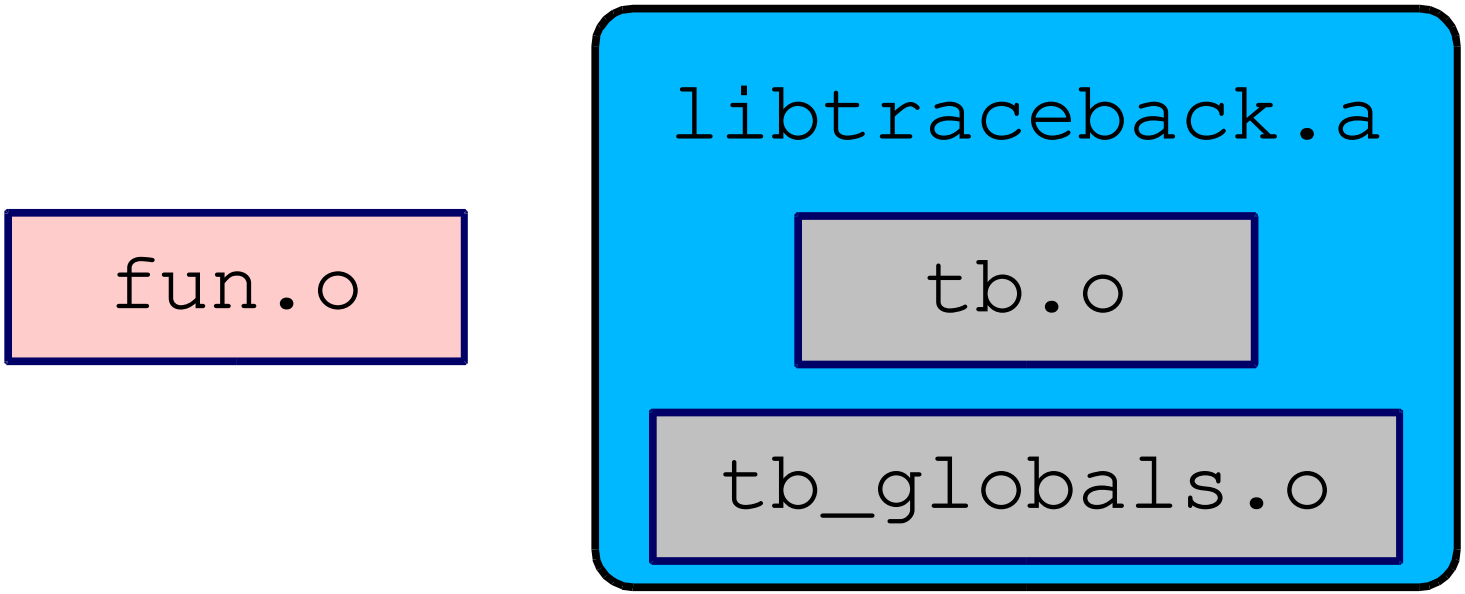
Key question

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

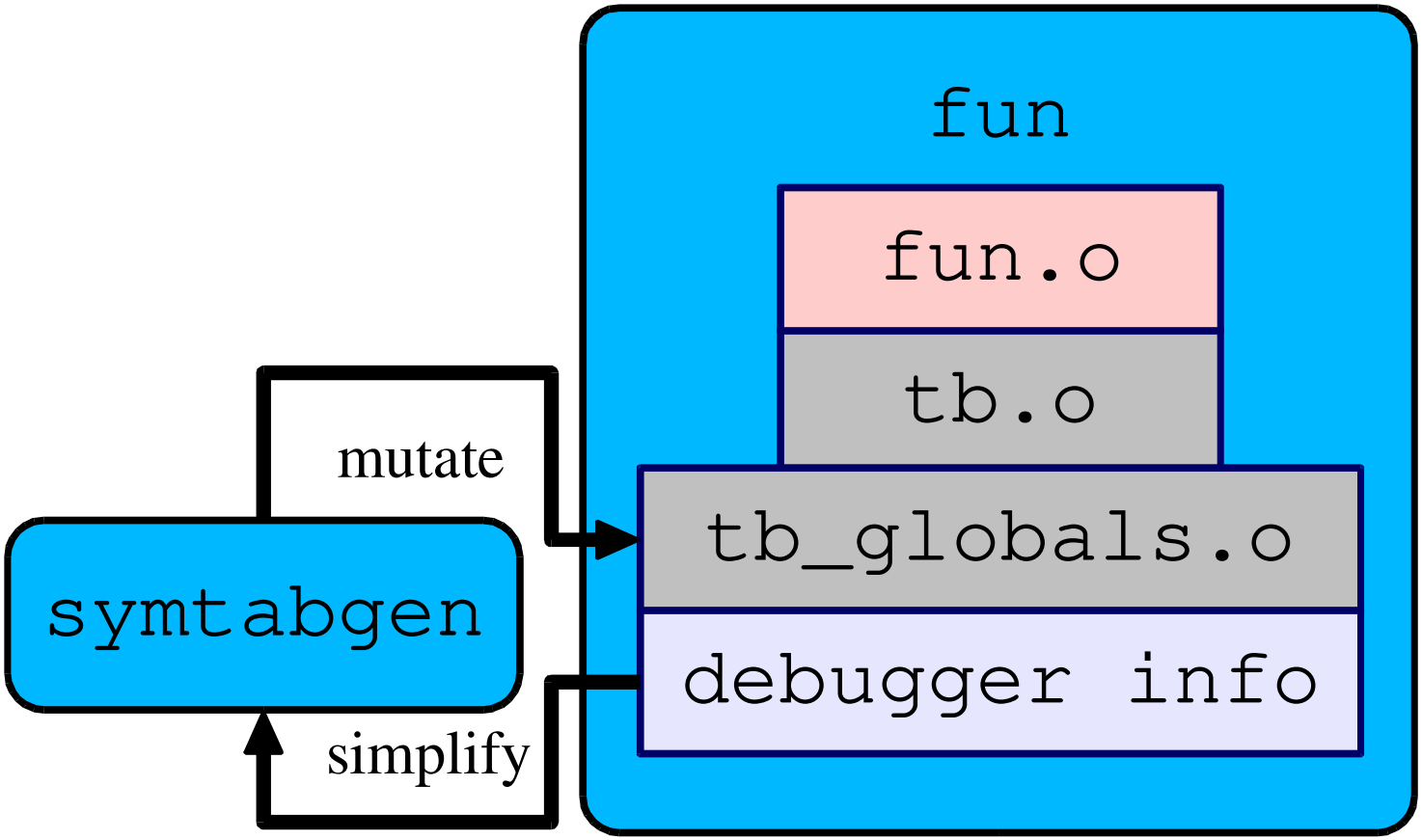
Project 0 “Data Flow”



Project 0 “Data Flow”



Project 0 “Data Flow”



Summary

Review of stack knowledge

What makes `main()` special

Project 0 overview

Start interviewing Project 2/3/4 partners!