

The Stack & Procedures

CSE 351 Winter 2022

Instructor:

Sam Wolfson

Teaching Assistants:

Angela Xu

Dara Stotland

Kevin Wang

Sanjana Sridhar

Anirudh Kumar

Harrison Bay

Mara Kirdani-Ryan

Catherine Guevara

Ian Hsiao

Nick Durand

↻ You Retweeted



Senior Oops Engineer @ReinH · Feb 28, 2019

I am a full stack engineer which means if you give me one more task my stack will overflow

💬 42

↻ 2.2K

❤️ 6.8K



Relevant Course Information

- ❖ Lab 2 due next Friday (2/4)
 - Can start in earnest after today's lecture!
 - GDB tutorial and phase 1 walkthrough in section tomorrow

- ❖ Midterm (take home, 2/9—2/11)
 - Make notes and use the [midterm reference sheet](#)
 - Form study groups and look at past exams!
 - Socio-technical content **is** fair game

x86 Control Flow

- ❖ Condition codes
- ❖ Conditional and unconditional branches
- ❖ Loops
- ❖ **Switches**

```
long switch_ex
(long x, long y, long z)
{
    long w = 1;
    switch (x) {
        case 1:
            w = y*z; break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z; break;
        case 5:
        case 6:
            w -= z; break;
        case 7:
            w = y%z; break;
        default:
            w = 2;
    }
    return w;
}
```

Switch Statement Example

- ❖ Multiple case labels
 - Here: 5 & 6 *both run same code block*
- ❖ Fall through cases
 - Here: 2 *will execute blocks 2 and 3*
- ❖ Missing cases
 - Here: 4 *will execute default block*
- ❖ Implemented with:
 - *Jump table*
 - *Indirect jump instruction*

Jump Table Structure

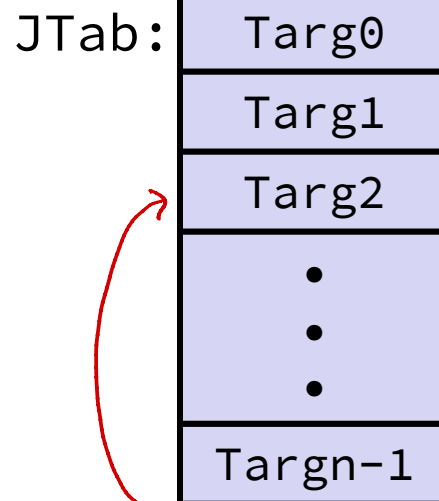
Switch Form

```
switch (x) {  
  case val_0:  
    Block 0  
  case val_1:  
    Block 1  
    . . .  
  case val_n-1:  
    Block n-1  
}
```

Approximate Translation

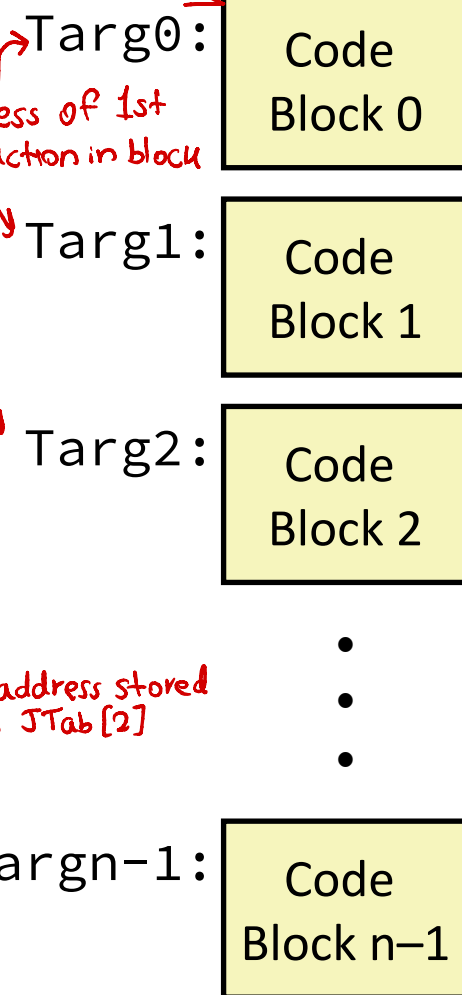
```
target = JTab[x];  
goto target;
```

Jump Table



code blocks
stored in memory

Jump Targets



e.g., $x = 2$

target = JTab[2] // get address stored
at JTab[2]

jump to that address

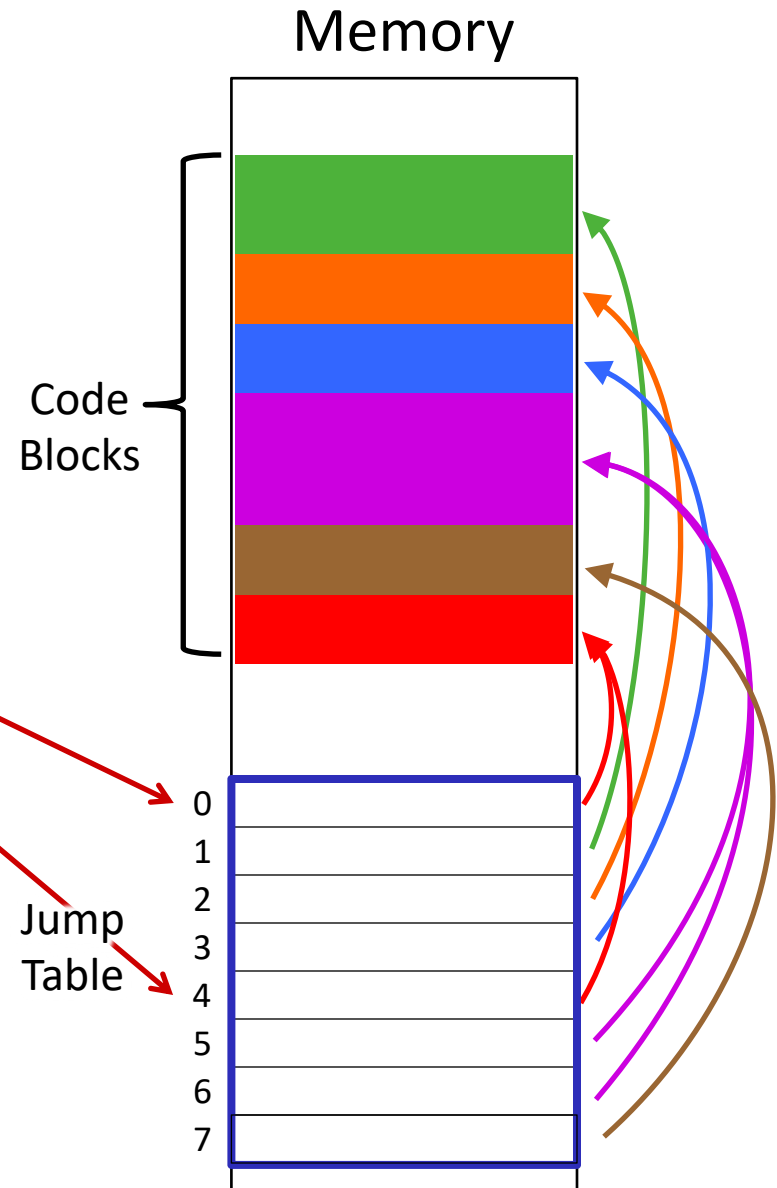
Jump Table Structure

C code:

```
switch (x) {  
  case 1: <code> break;  
  case 2: <code>  
  case 3: <code> break;  
  case 5:  
  case 6: <code> break;  
  case 7: <code> break;  
  default: <code>  
}
```

Use the jump table when $x \leq 7$:

```
if (x <= 7)  
  target = JTab[x];  
  goto target;  
else  
  goto default;
```



Switch Statement Example

```
long switch_ex(long x, long y, long z)
{
    long w = 1;
    switch (x) {
        . . .
    }
    return w;
}
```

Register	Use(s)
%rdi	1 st argument (x)
%rsi	2 nd argument (y)
%rdx	3 rd argument (z)
%rax	return value

Note compiler chose to not initialize w

```
switch_ex:
    movq    %rdx, %rcx
    cmpq    $7, %rdi          # x:7
    ja      .L9               # default
    jmp     *.L4(, %rdi, 8)    # jump table
```

Take a look!

<https://godbolt.org/z/Y9Kerb>

handle $x < 0$ or $x > 7$

jump above – unsigned > catches negative default cases

Switch Statement Example

```
long switch_ex(long x, long y, long z)
{
    long w = 1;
    switch (x) {
        . . .
    }
    return w;
}
```

```
switch_ex:
    movq    %rdx, %rcx
    cmpq    $7, %rdi        # x:7
    ja      .L9              # default
    jmp     *.L4(, %rdi, 8)   # jump table
```

Jump table

```
.section    .rodata
    .align 8
.L4:
    .quad   .L9    # x = 0
    .quad   .L8    # x = 1
    .quad   .L7    # x = 2
    .quad   .L10   # x = 3
    .quad   .L9    # x = 4
    .quad   .L5    # x = 5
    .quad   .L5    # x = 6
    .quad   .L3    # x = 7
```

Indirect
jump

① calculate location of address: $D(, R_i, S) \Rightarrow .L4 + x * 8$ ← size of address
 ↑ desired index
 beginning of jump table

② jump to the address stored there (indirect)

Assembly Setup Explanation

❖ Table Structure

- Each target requires 8 bytes (address)
- Base address at `.L4`

❖ Direct jump: `jmp .L9`

- Jump target is denoted by label `.L9`

❖ Indirect jump: `jmp *.L4(,%rdi,8)`

- Start of jump table: `.L4`
- Must scale by factor of 8 (addresses are 8 bytes)
- Fetch target from effective address `.L4 + x*8`
 - Only for $0 \leq x \leq 7$

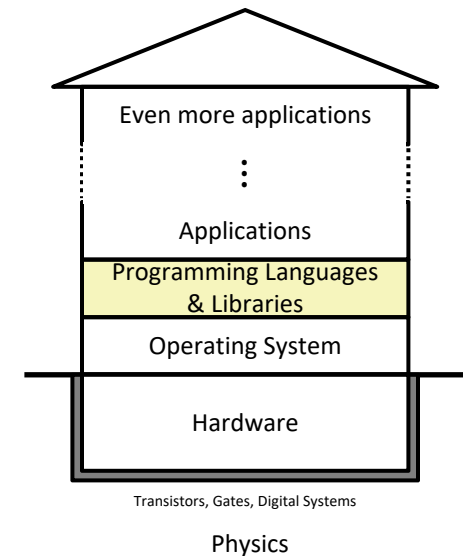
Jump table

```
.section      .rodata
    .align 8
.L4:
    .quad     .L9      # x = 0
    .quad     .L8      # x = 1
    .quad     .L7      # x = 2
    .quad     .L10     # x = 3
    .quad     .L9      # x = 4
    .quad     .L5      # x = 5
    .quad     .L5      # x = 6
    .quad     .L3      # x = 7
```

The Hardware/Software Interface

❖ Topic Group 2: **Programs**

- x86-64 Assembly, **Procedures, Stacks, Executables**



- ❖ How are programs created and executed on a CPU?
 - How does your source code become something that your computer understands?
 - How does the CPU organize and manipulate local data?

Reading Review

❖ Terminology:

- Stack, Heap, Static Data, Literals, Code
- Stack pointer (`%rsp`), `push`, `pop`
- Caller, callee, return address, `call`, `ret`
 - Return value: `%rax`
 - Arguments: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- Stack frames and stack discipline

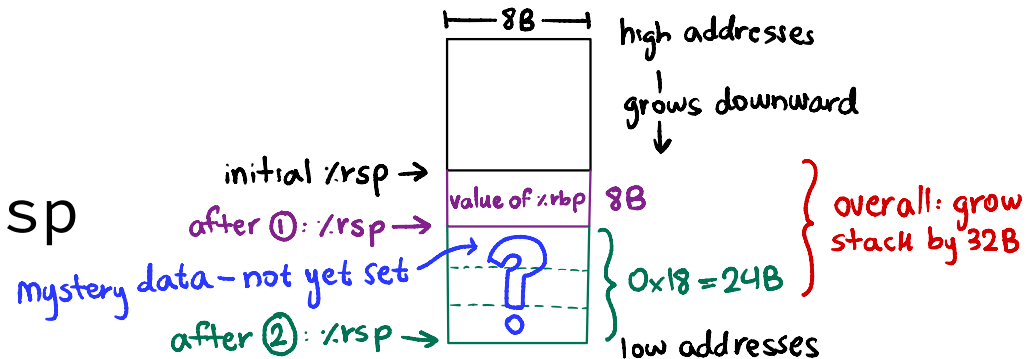
❖ Questions from the Reading?

Review Questions

- ❖ How does the stack change after executing the following instructions?

① `pushq %rbp`

② `subq $0x18, %rsp`



- ❖ For the following function, which registers do we know *must* be used?

```
void* memset(void* ptr, int value, size_t num);
```

`%rax`

↑
return register
(convention)

`%rdi`

`%rsi`

`%rdx`

argument registers (by convention)

`%rsp`: used by `call`/`ret` instructions

`%rip`: changes whenever instruction is executed

Mechanisms required for *procedures*

1) Passing control

- To beginning of procedure code
- Back to return point

2) Passing data

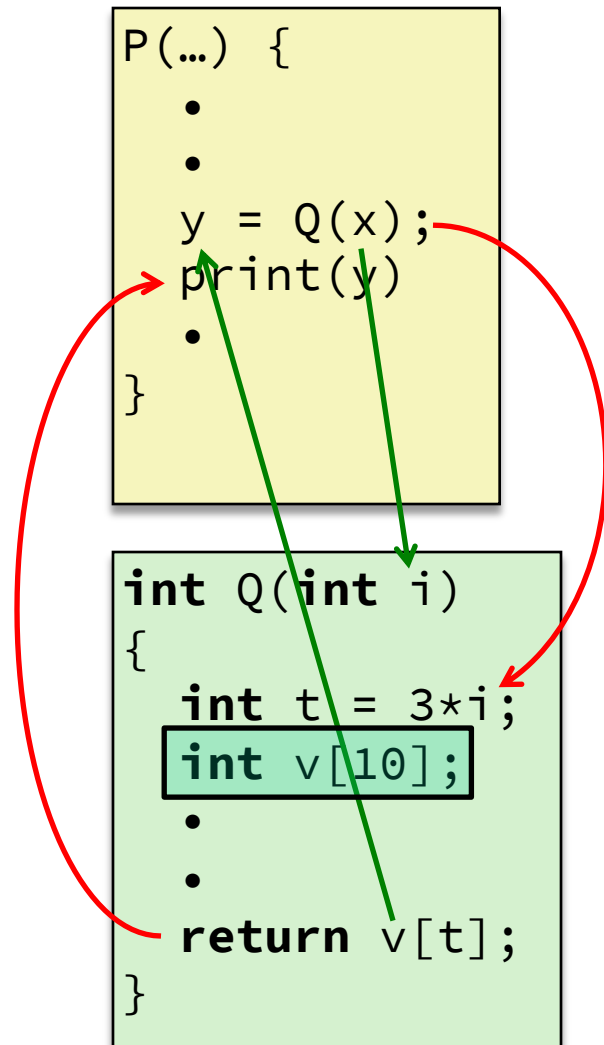
- Procedure arguments
- Return value

3) Memory management

- Allocate during procedure execution
- De-allocate upon return

❖ All implemented with machine instructions!

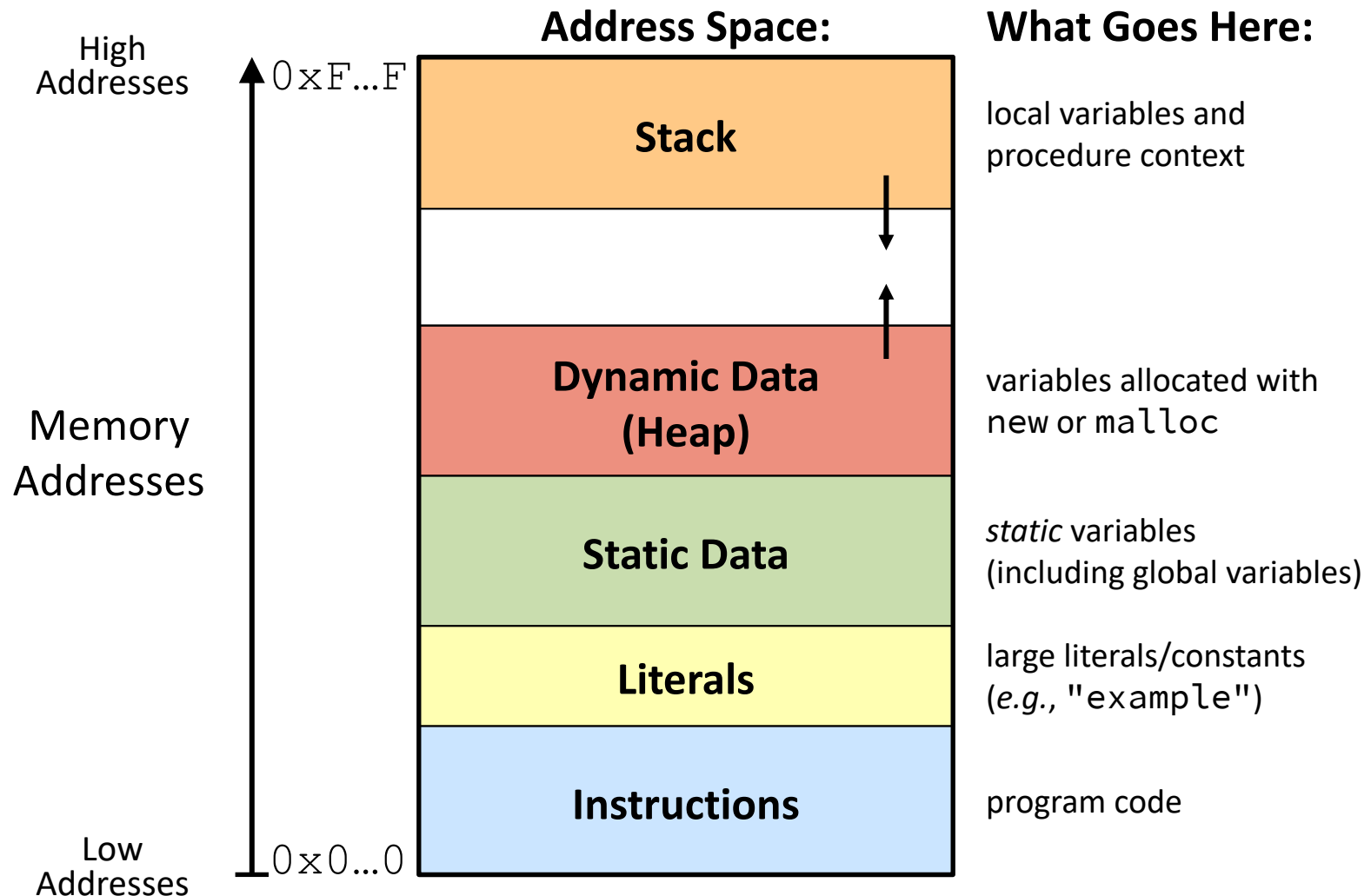
- An x86-64 procedure uses only those mechanisms required for that procedure



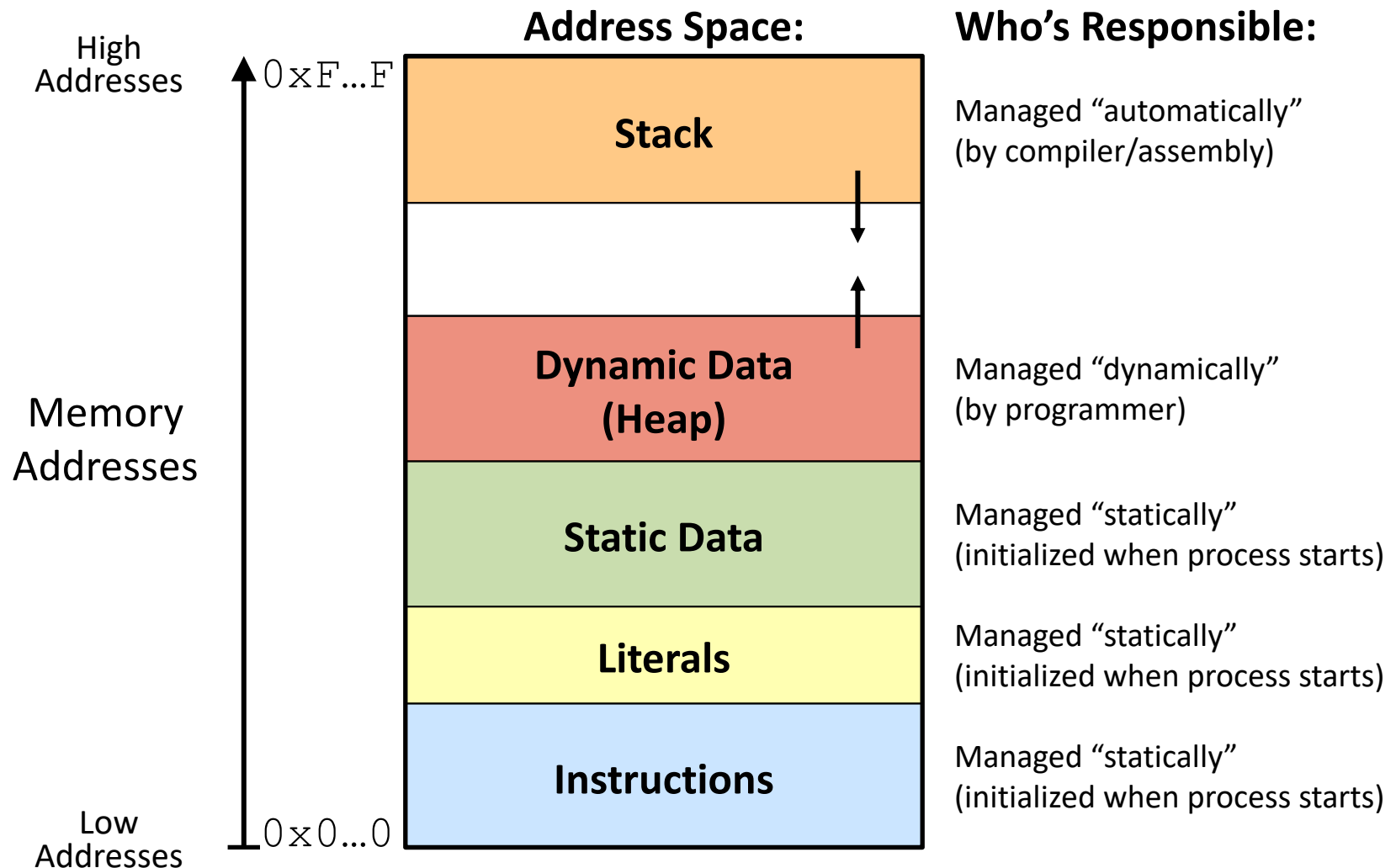
Procedures

- ❖ **Stack Structure**
- ❖ Calling Conventions
 - Passing control
 - Passing data
 - Managing local data
- ❖ Register Saving Conventions
- ❖ Illustration of Recursion

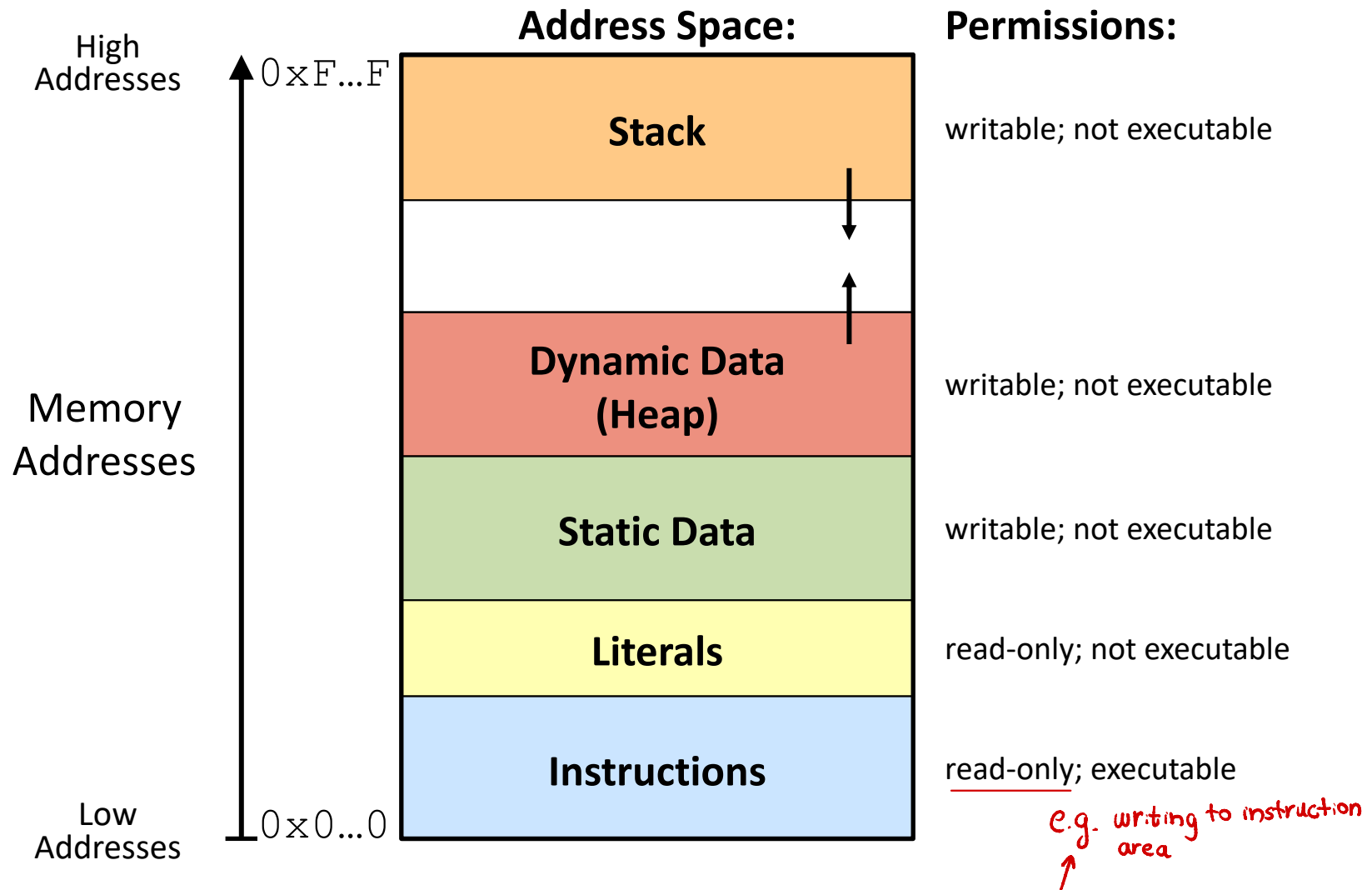
Simplified Memory Layout (Review)



Memory Management



Memory Permissions



- Segmentation fault: impermissible memory access

x86-64 Stack (Review)

- ❖ Region of memory managed with stack “discipline”
 - Grows toward lower addresses
 - Customarily shown “upside-down”
- ❖ Register `%rsp` contains *lowest* stack address
 - `%rsp` = address of *top* element, the most-recently-pushed item that is not-yet-popped

Stack Pointer: `%rsp` →

Stack “Bottom”



Stack “Top”

High
Addresses

↑
Increasing
Addresses
|

|
Stack Grows
Down
↓

Low
Addresses
0x00...00

x86-64 Stack: Push (Review)

❖ `pushq src`

- Fetch operand at `src`
 - Src can be reg, memory, immediate
- Decrement** `%rsp` by 8
- Store value at address given by `%rsp`

❖ Example:

■ `pushq %rcx`

- Adjust `%rsp` and store contents of `%rcx` on the stack

does 2 things:

- ① decrement `%rsp` by 8 (expand downwards) e.g. `subq $8, %rsp`
- ② copy `%rcx` to there e.g. `movq %rcx, (%rsp)`

Stack Pointer: `%rsp`



Stack "Bottom"



Stack "Top"

High
Addresses

↑
Increasing
Addresses

↓
Stack Grows
Down

Low
Addresses
0x00...00

x86-64 Stack: Pop (Review)

❖ `popq dst`

- Load value at address given by `%rsp`
- Store value at `dst`
- **Increment** `%rsp` by 8

❖ Example:

■ `popq %rcx`

- Stores contents of top of stack into `%rcx` and adjust `%rsp`

like doing:

```
movq (%rsp), %rcx
addq $8, %rsp
```

Those bits are still there;
we're just not using them.

Stack Pointer: `%rsp`

+8

Stack "Bottom"

Stack "Top"

High
Addresses

↑
Increasing
Addresses
|

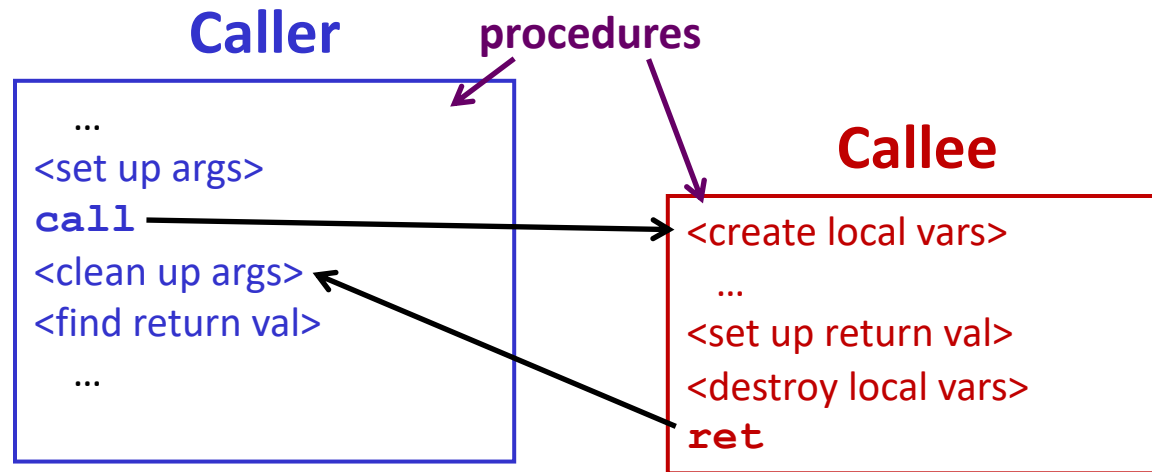
|
Stack Grows
Down
↓

Low
Addresses
0x00...00

Procedures

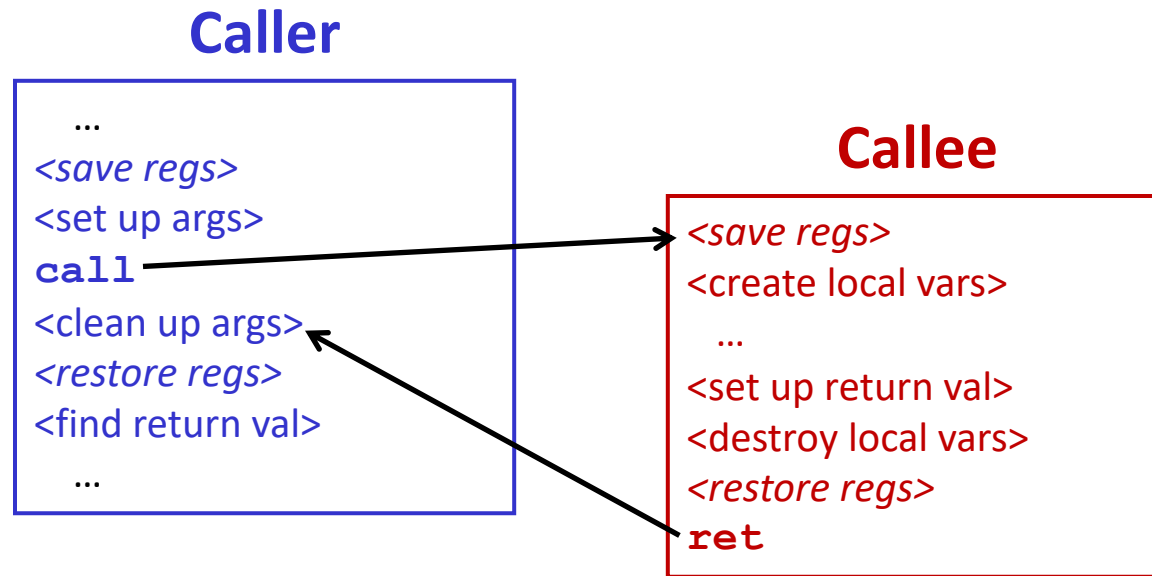
- ❖ Stack Structure
- ❖ **Calling Conventions**
 - **Passing control**
 - Passing data
 - Managing local data
- ❖ Register Saving Conventions
- ❖ Illustration of Recursion

Procedure Call Overview



- ❖ **Callee** must know where to find args
- ❖ **Callee** must know where to find *return address*
- ❖ **Caller** must know where to find *return value*
- ❖ **Caller** and **Callee** run on same CPU, so use the same registers
 - How do we deal with register reuse?
- ❖ Unneeded steps can be skipped (e.g., no arguments)

Procedure Call Overview



- ❖ The *convention* of where to leave/find things is called the calling convention (or procedure call linkage)
 - Details vary between systems
 - We will see the convention for x86-64/Linux in detail
 - What could happen if our program didn't follow these conventions?