x86-64 Programming III

CSE 351 Summer 2018

Instructor:

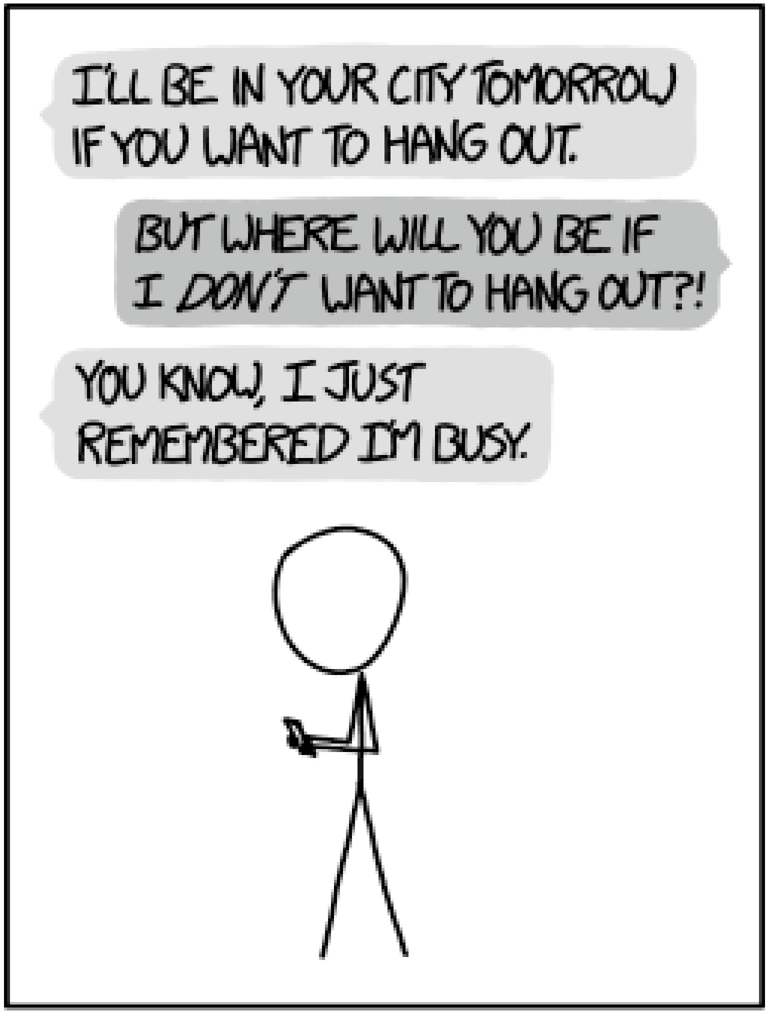
Justin Hsia

Teaching Assistants:

Josie Lee

Natalie Andreeva

Teagan Horkan



WHY I TRY NOT TO BE PEDANTIC ABOUT CONDITIONALS.

http://xkcd.com/1652/

Administrivia

- Homework 2 due Wednesday (7/11)

- Lab 2 (x86-64) due next Monday (7/16)
  - Learn to read x86-64 assembly and use GDB

- Midterm is next Wednesdays (7/18 in lecture)
  - You will be provided a fresh reference sheet
    - Study and use this NOW so you are comfortable with it when the exam comes around
  - You get 1 *handwritten*, double-sided cheat sheet (letter)
  - Find a study group! Look at past exams!

GDB Demo

- Examine the **`movz`** and **`movs`** examples from last lecture on a real machine!
  - `movzbq %al, %rbx`
  - `movsbl (%rax), %ebx`

- You will need to use GDB to get through Lab 2
  - Useful debugger in this class and beyond!

- Pay attention to:
  - Setting breakpoints (`break`)
  - Stepping through code (`step`/`next` and `stepi`/`nexti`)
  - Printing out expressions (`print` – works with regs & vars)
  - Examining memory (`x`)

Choosing instructions for conditionals

- All arithmetic instructions set condition flags based on result of operation (**op**)
  - Conditionals are comparisons against 0
- Come in instruction *pairs*

```
    ①addq 5, (p)
  je:    *p+5 == 0
②jne:    *p+5 != 0
  jg:    *p+5 >  0
  jl:    *p+5 <  0
```

```
     ①orq a, b
  je:    b|a == 0
  jne:   b|a != 0
②jg:     b|a >  0
  jl:    b|a <  0
```

		① **(op) s, d**
je	"Equal"	d (op) s == 0
jne	"Not equal"	d (op) s != 0
js	"Sign" (negative)	d (op) s < 0
jns	(non-negative)	d (op) s >= 0
jg	"Greater"	d (op) s > 0
jge	"Greater or equal"	d (op) s >= 0
② **jl**	"Less"	d (op) s < 0
jle	"Less or equal"	d (op) s <= 0
ja	"Above" (unsigned >)	d (op) s > 0U
jb	"Below" (unsigned <)	d (op) s < 0U

Choosing instructions for conditionals

- Reminder: `cmp` is like `sub`, `test` is like `and`
  - Result is not stored anywhere

		cmp a,b	test a,b
je	"Equal"	b == a	b&a == 0
jne	"Not equal"	b != a	b&a != 0
js	"Sign" (negative)	b-a < 0	b&a < 0
jns	(non-negative)	b-a >=0	b&a >= 0
jg	"Greater"	b > a	b&a > 0
jge	"Greater or equal"	b >= a	b&a >= 0
jl	"Less"	b < a	b&a < 0
jle	"Less or equal"	b <= a	b&a <= 0
ja	"Above" (unsigned >)	b > a	b&a > 0U
jb	"Below" (unsigned <)	b < a	b&a < 0U

```
cmpq 5, (p)
je:   *p == 5
jne:  *p != 5
jg:   *p >  5
jl:   *p <  5
```

```
testq a, a
je:    a == 0
jne:   a != 0
jg:    a >  0
jl:    a <  0
```

```
testb a, 0x1
je:    a_LSB == 0
jne:   a_LSB == 1
```

Choosing instructions for conditionals

		cmp a,b	test a,b
je	"Equal"	b == a	b&a == 0
jne	"Not equal"	b != a	b&a != 0
js	"Sign" (negative)	b-a < 0	b&a < 0
jns	(non-negative)	b-a >=0	b&a >= 0
jg	"Greater"	b > a	b&a > 0
jge	"Greater or equal"	b >= a	b&a >= 0
jl	"Less"	b < a	b&a < 0
jle	"Less or equal"	b <= a	b&a <= 0
ja	"Above" (unsigned >)	b > a	b&a > 0U
jb	"Below" (unsigned <)	b < a	b&a < 0U

Register	Use(s)
%rdi	argument x
%rsi	argument y
%rax	return value

```
if (x < 3) {
  return 1;
}
return 2;
```

do this if x ≥ 3 → x-3 >= 0

```
  cmpq $3, %rdi
  jge T2
T1: # x < 3:        (if)
  movq $1, %rax
  ret
T2: # !(x < 3):     (else)
  movq $2, %rax
  ret
```

labels

Question

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

```
long absdiff(long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

A. `cmpq %rsi, %rdi` / `jle .L4` x-y

B. `cmpq %rsi, %rdi` / `jg .L4` x-y

C. `testq %rsi, %rdi` / `jle .L4` x&y

D. `testq %rsi, %rdi` / `jg .L4` x&y

E. We're lost…

```
absdiff:
    ____________________
    ____________________
                          # x > y:
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                      # x <= y:
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

x-y <= 0

less than or equal to (le)

Labels

```
swap:
    movq  (%rdi), %rax
    movq  (%rsi), %rdx
    movq  %rdx, (%rdi)
    movq  %rax, (%rsi)
    ret
```

```
max:
    movq %rdi, %rax
    cmpq %rsi, %rdi
    jg   done
    movq %rsi, %rax
done:
    ret
```

- A jump changes the program counter (`%rip`)
  - `%rip` tells the CPU the *address* of the next instr to execute
- **Labels** give us a way to refer to a specific instruction in our assembly/machine code
  - Associated with the *next* instruction found in the assembly code (ignores whitespace)
  - Each ***use*** of the label will eventually be replaced with something that indicates the final address of the instruction that it is associated with

x86 Control Flow

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

Expressing with Goto Code

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = (x <= y);
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

conditional jump

cmp

jle

unconditional jump →

jmp

labels (addresses)

- C allows `goto` as means of transferring control (`jump`)
  - Closer to assembly programming style
  - Generally considered bad coding style

Compiling Loops

C/Java code:

```
while ( sum != 0 ) {
    <loop body>
}
```

Assembly code:

```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```

- Other loops compiled similarly
  - Will show variations and complications in coming slides, but may skip a few examples in the interest of time
- Most important to consider:
  - When should conditionals be evaluated? (*while* vs. *do-while*)
  - How much jumping is involved?

Compiling Loops

C/Java code:

```
while ( Test ) {
    Body
}
```

Goto version:

```
Loop: if ( !Test )  goto Exit;
      Body
      goto Loop;
Exit:
```

- What are the Goto versions of the following?
  - Do…while: *Test* and *Body*
  - For-loop: *Init*, *Test*, *Update*, and *Body*

```
do{
  Body
}while(Test);
```

"i=0" "i<n" "i++"

```
for(Init; Test; Update) {
  Body
}
```

Do…while

```
Loop: Body
      if (Test) goto Loop;
```

For loop

```
      Init
Loop: if (!Test) goto Exit;
      Body
      Update
      goto Loop;
Exit:
```

Compiling Loops

all jump instructions update the program counter (%rip)

While Loop:

C:
```
while ( sum != 0 ) {
    <loop body>
}
```
(Test)

x86-64:
```
loopTop:    testq %rax, %rax
            je    loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```
(sum == 0; ~Test)

Do-while Loop:

C:
```
do {
    <loop body>
} while ( sum != 0 )
```
(Test)

x86-64:
```
loopTop:
            <loop body code>
            testq  %rax, %rax
            jne    loopTop
loopDone:
```
(Test)

While Loop (ver. 2):

C:
```
while ( sum != 0 ) {
    <loop body>
}
```
(Test)

x86-64:
```
            testq %rax, %rax
            je    loopDone
loopTop:
            <loop body code>
            testq %rax, %rax
            jne   loopTop
loopDone:
```
(~Test; do-while loop; Test)

For-Loop → While-Loop

For-Loop:

```
for (Init; Test; Update) {
      Body
}
```

While-Loop Version:

```
Init;
while (Test) {
      Body
      Update;
}
```

Caveat: C and Java have `break` and `continue`

- Conversion works fine for `break`
  - Jump to same label as loop exit condition
- But not continue: would skip doing *Update*, which it should do with for-loops
  - Introduce new label at *Update*

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;
        default:
            w = 2;
    }
    return w;
}
```

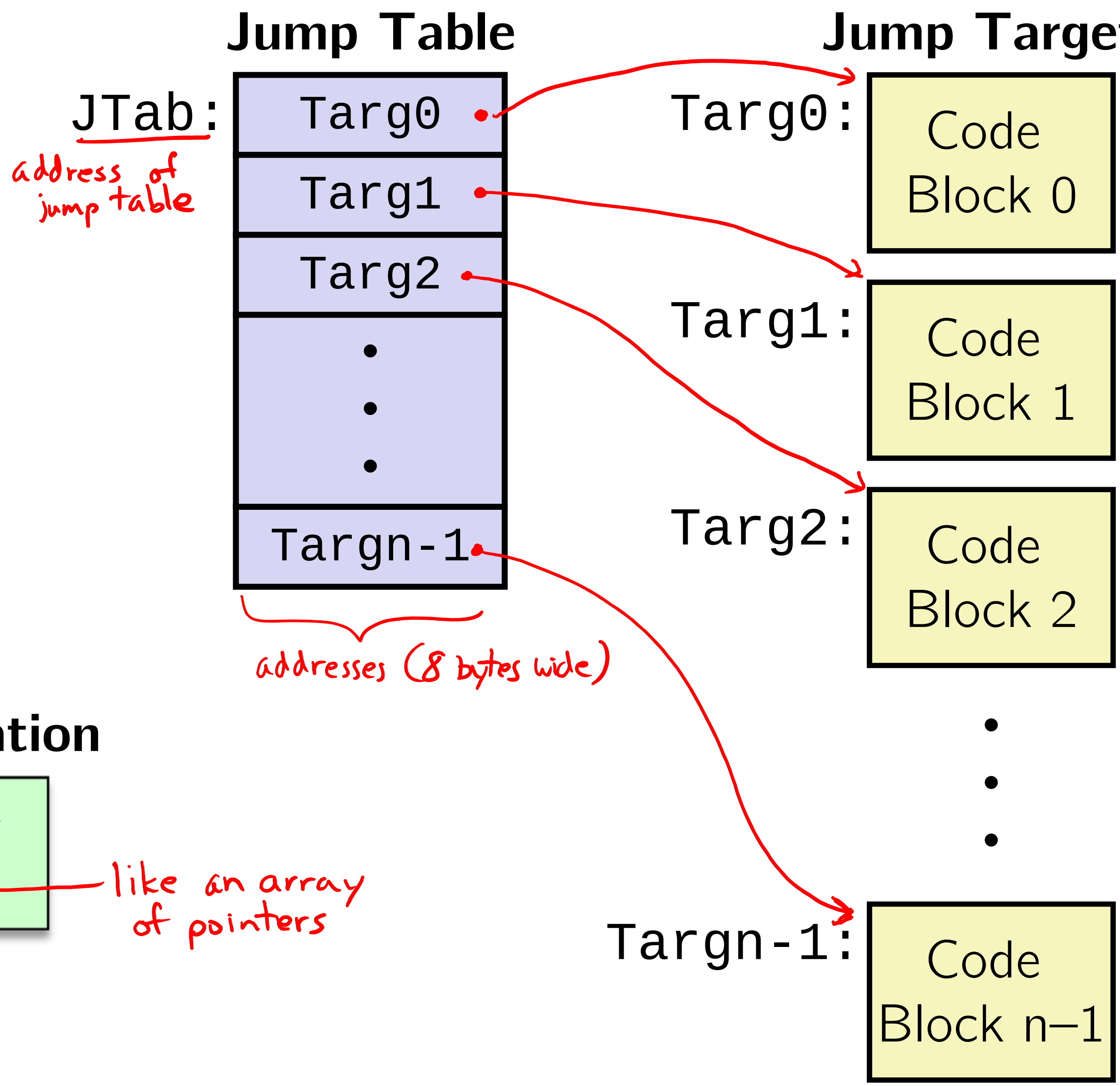
Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4
- 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;
```

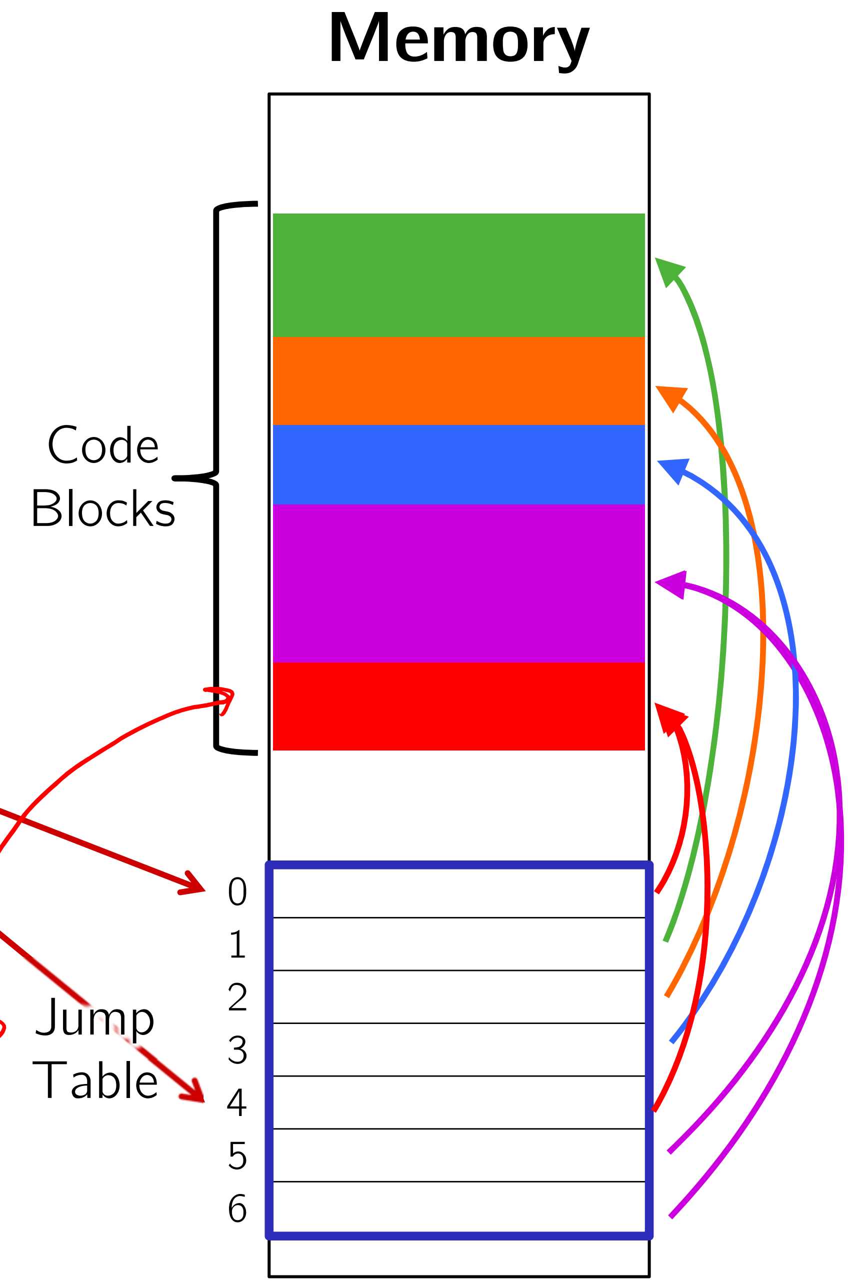
Jump Table Structure

C code:

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

Use the jump table when x ≤ 6:

```
if (x <= 6)
  target = JTab[x];
  goto target;
else
  goto default;
```



Switch Example

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

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

where?

Note compiler chose to not initialize w

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # default
    jmp     *.L4(,%rdi,8)  # jump table
```

jump to default case if x>6 (unsigned)

Take a look!

https://godbolt.org/g/DnOmXb

jump above – unsigned > catches negative default cases

-1 > 6U → jump to default case

Switch Example

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

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

Jump table

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

address

following data is a "quad word" = 8 bytes

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi           # x:6
    ja      .L8                # default
    jmp     *.L4(,%rdi,8) # jump table
```

Indirect jump

D + Ri * S

addr of jump table

x

sizeof (void*)

Assembly Setup Explanation

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

Jump table

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

- **Direct jump:** `jmp .L8`
  - Jump target is denoted by label `.L8`
- **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 $0 \le x \le 6$

Jump Table

declaring data, not instructions

8-byte memory alignment

Jump table

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

this data is 64-bits wide

```
switch(x) {
  case 1:         // .L3
      w = y*z;
      break;
  case 2:         // .L5
      w = y/z;
      /* Fall Through */
  case 3:         // .L9
      w += z;
      break;
  case 5:
  case 6:         // .L7
      w -= z;
      break;
  default:        // .L8
      w = 2;
}
```

Code Blocks (x == 1)

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

```
switch(x) {
  case 1:     // .L3
    w = y*z;
    break;
  . . .
}
```

```
.L3:
   movq    %rsi, %rax   # y
   imulq   %rdx, %rax   # y*z
   ret
```

Handling Fall-Through

```
long w = 1;
   . . .
switch (x) {
   . . .
  case 2:   // .L5
    w = y/z;
  /* Fall Through */
  case 3:   // .L9
    w += z;
    break;
   . . .
}
```

```
case 2:
       w = y/z;
       goto merge;
```

```
case 3:
       w = 1;
```

```
merge:
       w += z;
```

More complicated choice than "just fall-through" forced by "migration" of `w = 1;`

- Example compilation trade-off

Code Blocks (x == 2, 3)

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

```
long w = 1;
   . . .
switch (x) {
   . . .
  case 2:   // .L5
    w = y/z;
  /* Fall Through */
  case 3:   // .L9
    w += z;
    break;
   . . .
}
```

```
.L5:                    # Case 2:
   movq    %rsi, %rax   #  y in rax
   cqto                 #  Div prep
   idivq   %rcx         #  y/z
   jmp     .L6          #  goto merge
.L9:                    # Case 3:
   movl    $1, %eax     #  w = 1
.L6:                    # merge:
   addq    %rcx, %rax   #  w += z
   ret
```

Code Blocks (rest)

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

```
switch (x) {
   . . .
  case 5:  // .L7
  case 6:  // .L7
    w -= z;
    break;
  default: // .L8
    w = 2;
}
```

```
.L7:                      # Case 5,6:
  movl   $1, %eax         #   w = 1
  subq   %rdx, %rax       #   w -= z
  ret
.L8:                      # Default:
  movl   $2, %eax         #   2
  ret
```

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 10

OS X Yosemite

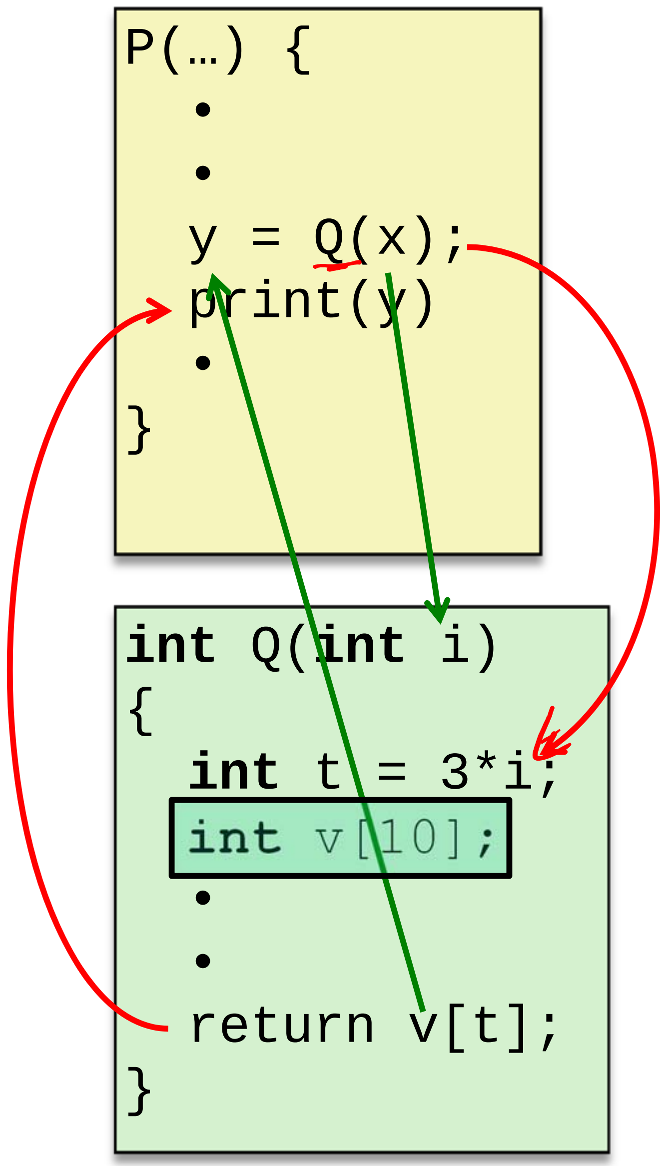
Computer system:

- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

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
   - Deallocate upon return

- All implemented with machine instructions!
   - An x86-64 procedure uses only those mechanisms required for that procedure



Procedures

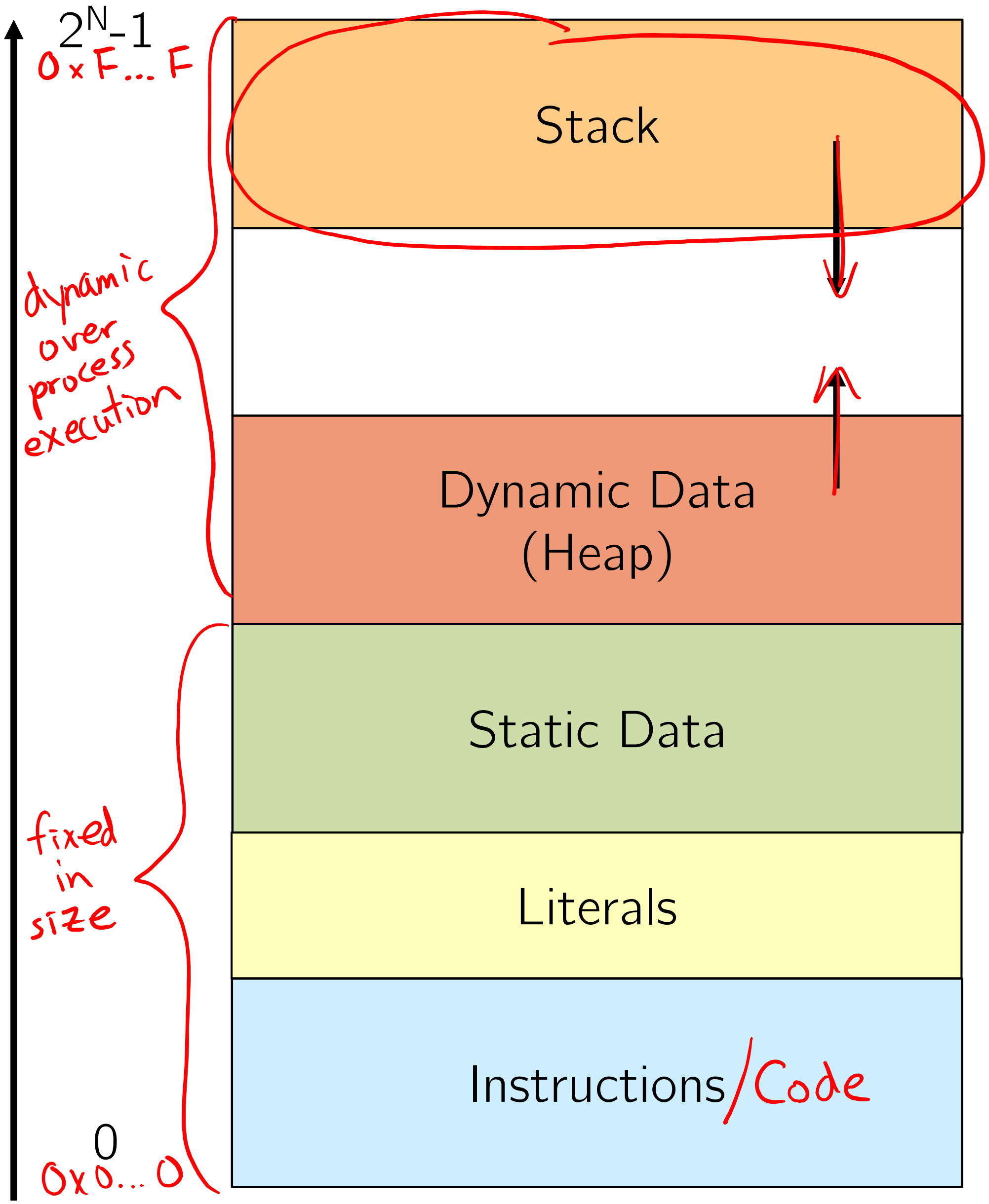
- **Stack Structure**
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Simplified Memory Layout

High Addresses

Memory Addresses

Low Addresses



local variables; procedure context

variables allocated with *new* or *malloc*

static variables (including global variables (C))

large constants (*e.g.* "example")

program code

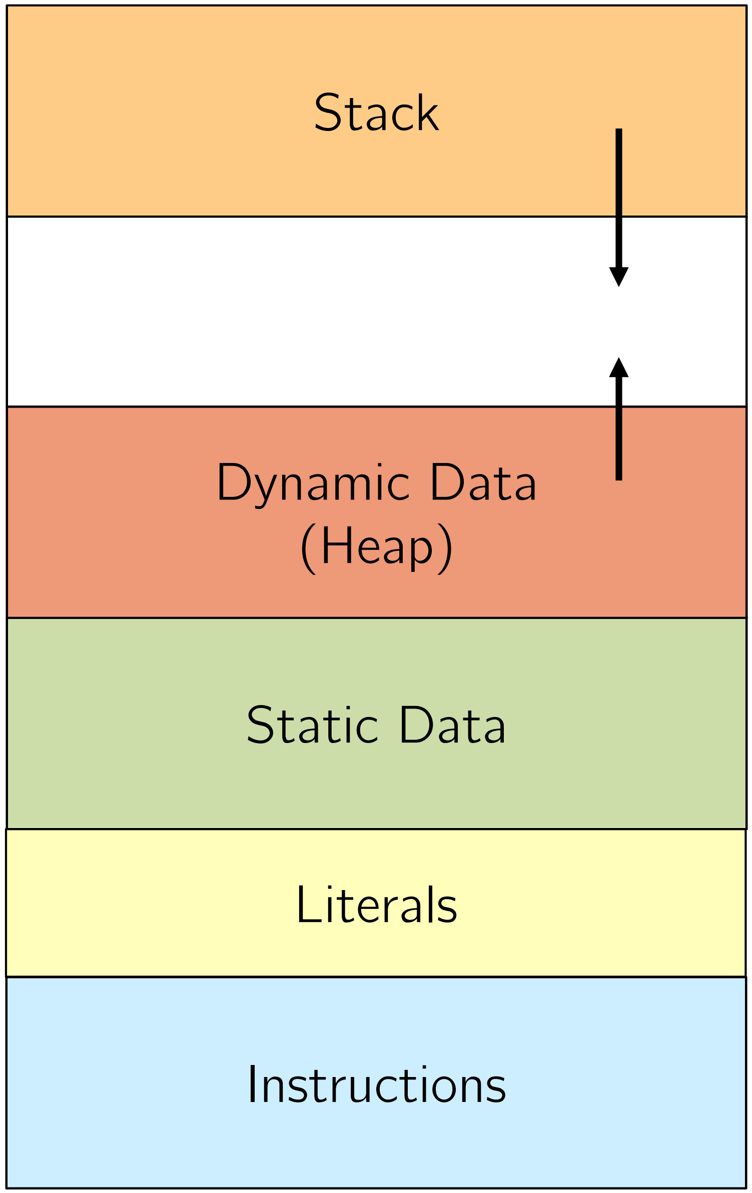
Memory Permissions

segmentation faults?

accessing memory in a way that you are not allowed to

writable; not executable

Managed "automatically" (by compiler)



grow towards each other to maximize use of space

writable; not executable

Managed by programmer

writable; not executable

Initialized when process starts

read-only; not executable

Initialized when process starts

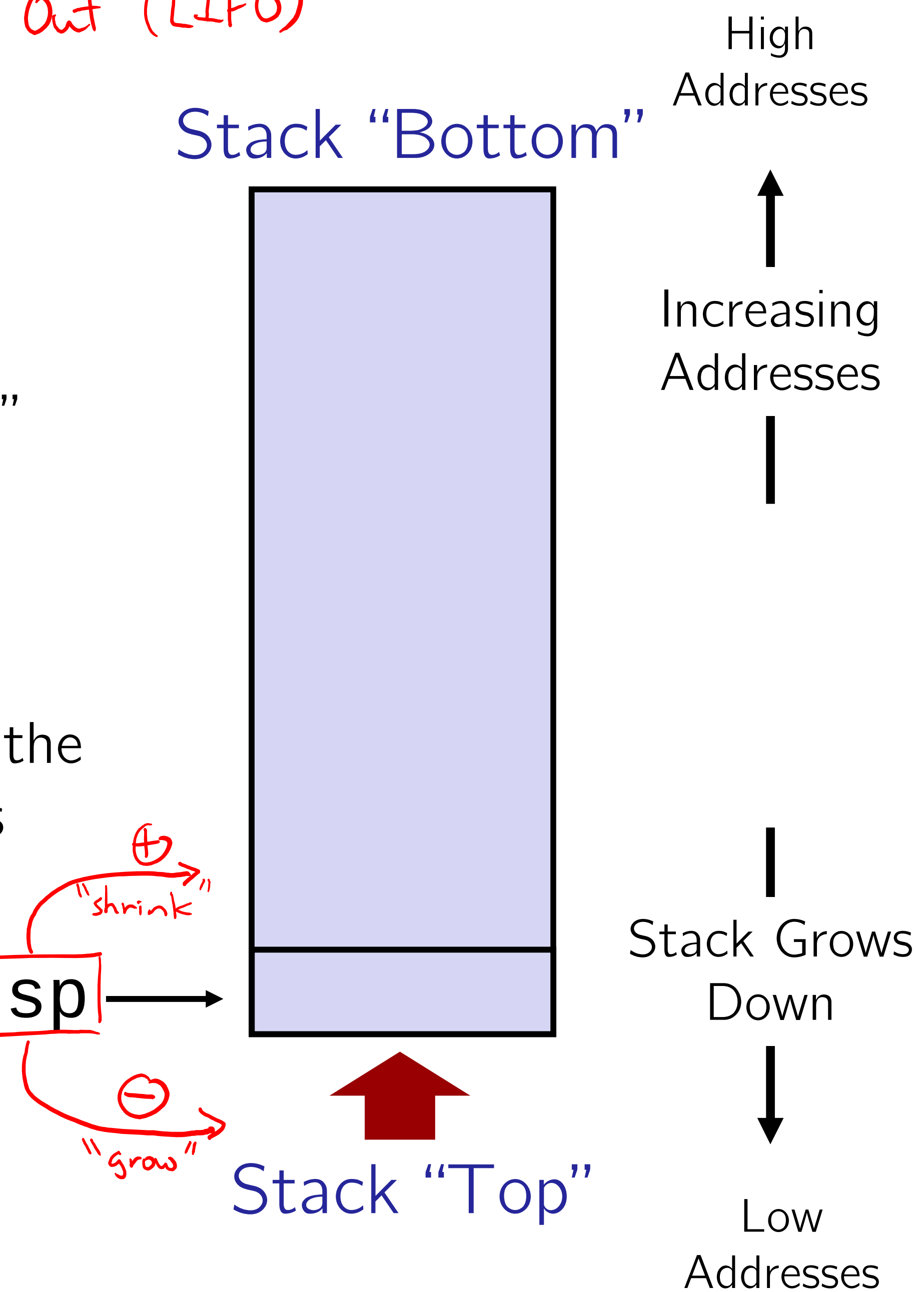
read-only; executable

Initialized when process starts

x86-64 Stack

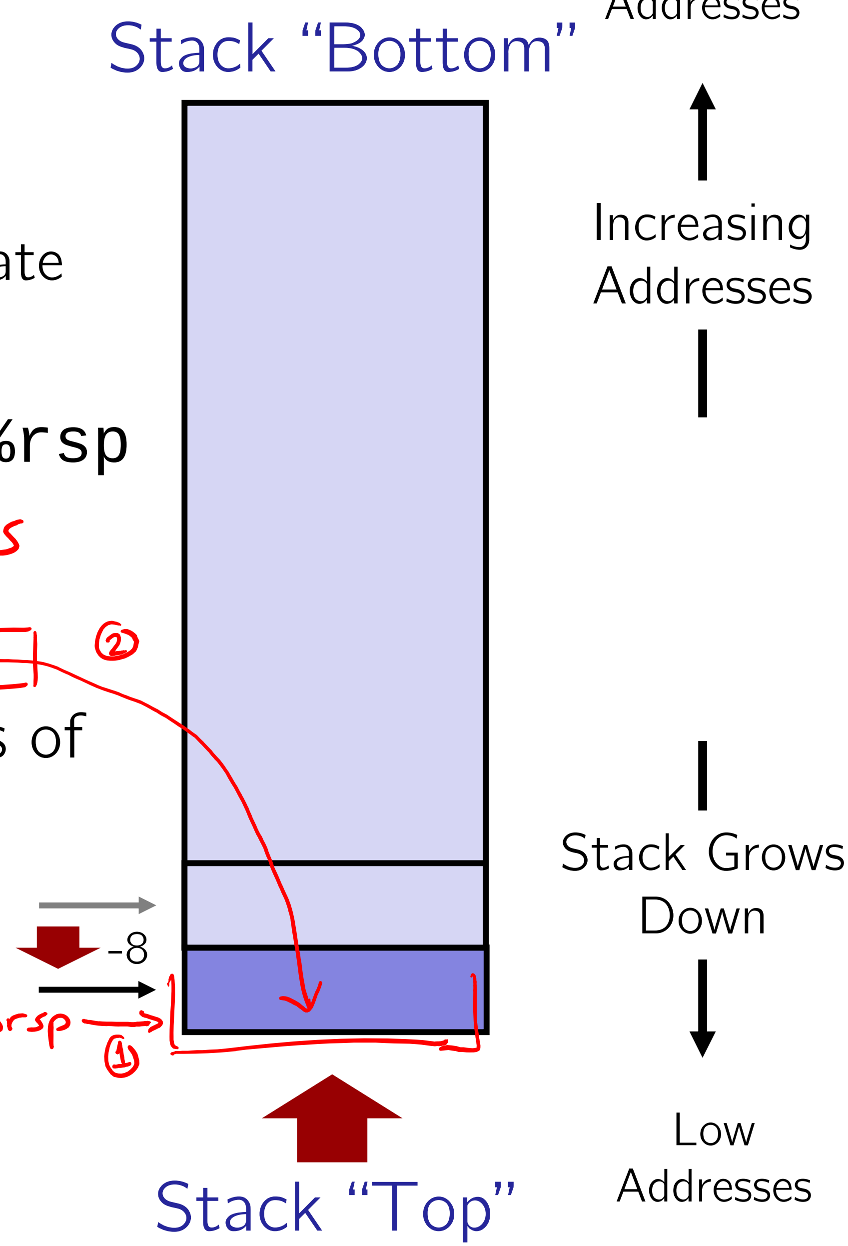
Last In, First Out (LIFO)

- 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



x86-64 Stack: Push

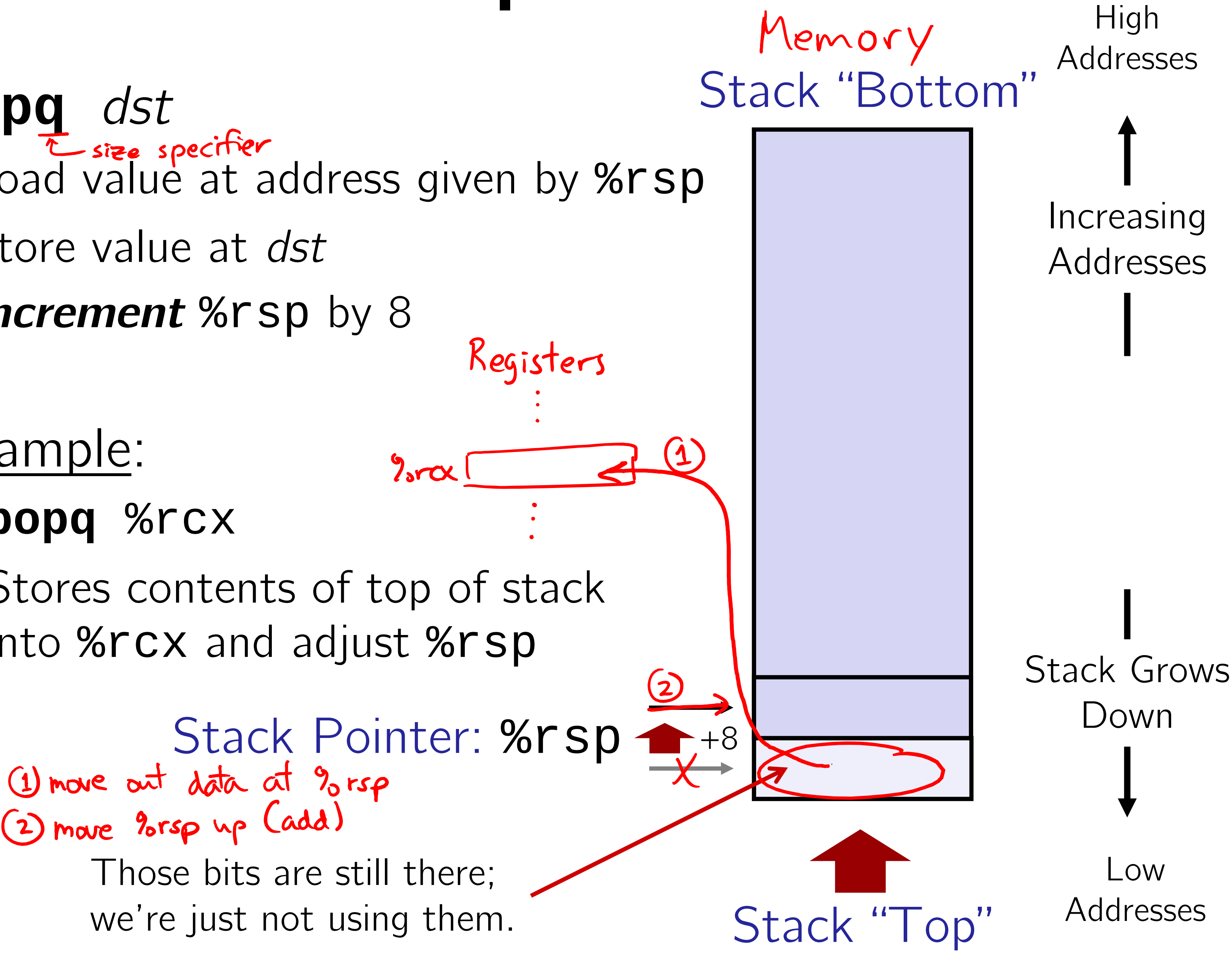
- **pushq** *src*
  - Fetch operand at *src*
    - *src* can be reg, memory, immediate
  - ***Decrement*** `%rsp` by 8
  - Store value at address given by `%rsp`
- Example:
  - **pushq** `%rcx`
  - Adjust `%rsp` and store contents of `%rcx` on the stack



x86-64 Stack: Pop

- **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`

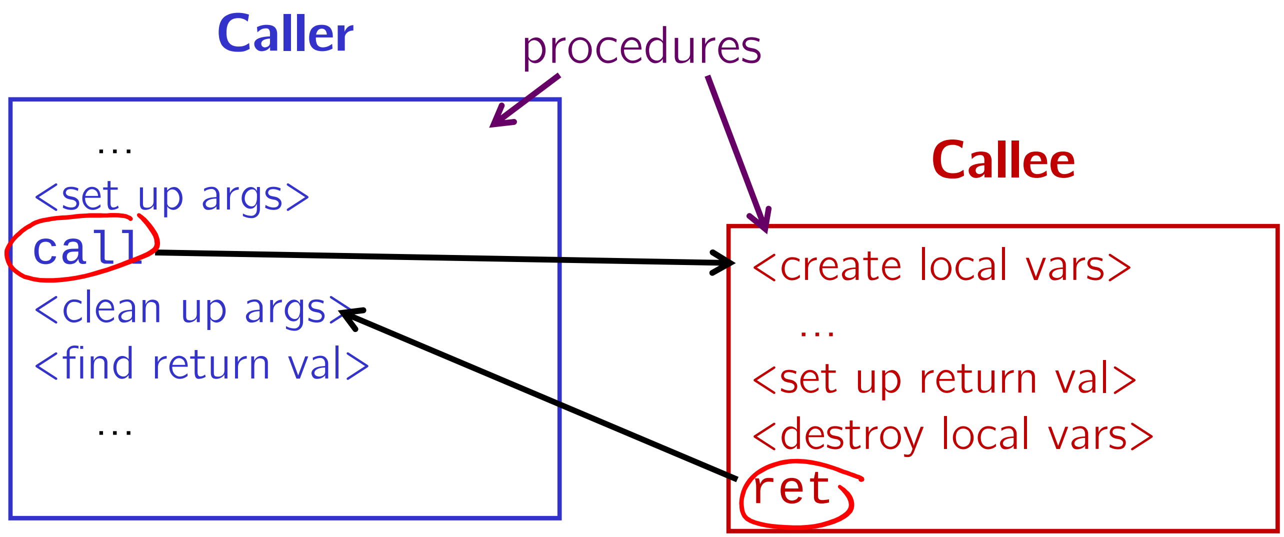


Those bits are still there; we're just not using them.

Procedures

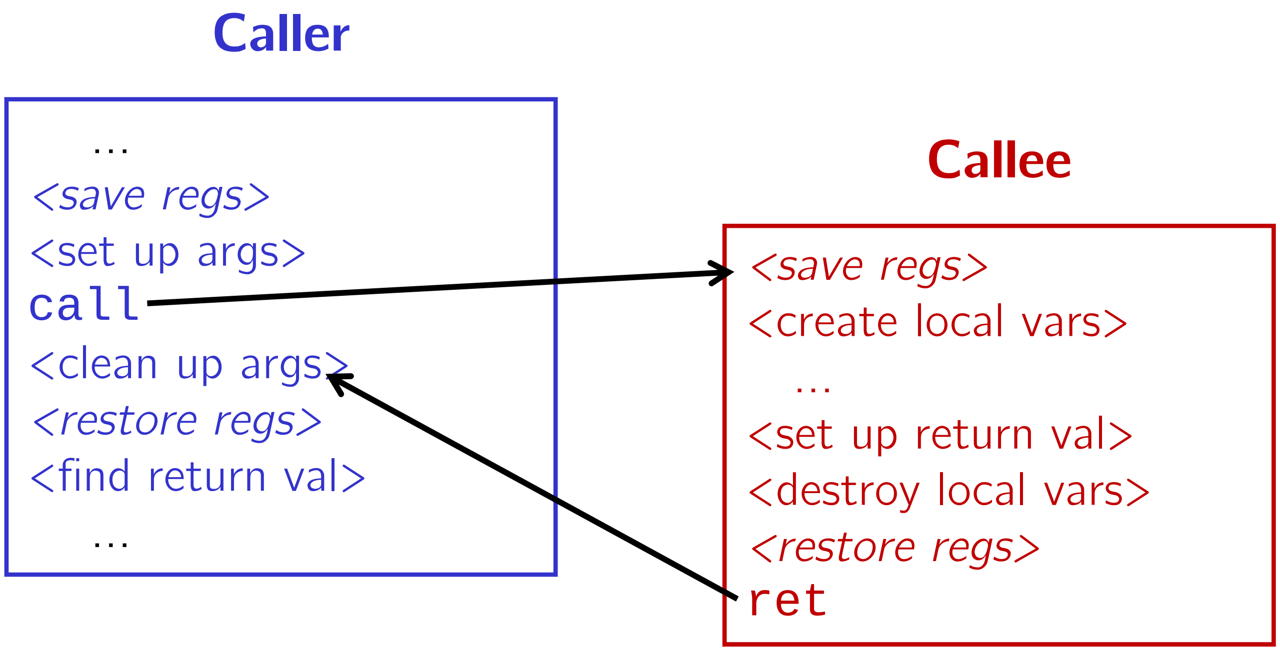
- Stack Structure
- **Calling Conventions**
  - **Passing control**
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

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?

Code Example (Preview)

Compiler Explorer:
https://godbolt.org/g/cKKDZn

Caller

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

executable disassembly

```
0000000000400540 <multstore>:
  400540: push    %rbx              # Save %rbx
  400541: movq    %rdx,%rbx         # Save dest
  400544: call    400550 <mult2>    # mult2(x,y)
  400549: movq    %rax,(%rbx)       # Save at dest
  40054c: pop     %rbx              # Restore %rbx
  40054d: ret                       # Return
```

these are instruction addresses

Callee

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax  # a
  400553:  imulq  %rsi,%rax  # a * b
  400557:  ret               # Return
```

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: **`call`** ***`label`*** (special push)
  1) Push return address on stack (*why? which address?*)
     - (1) move %rsp down
     - (2) store ret addr at %rsp
  2) Jump to *`label`*
     - (3) label → %rip

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: **`call label`** (special push)
  1) Push return address on stack (*why? which address?*)
     - ① move %rsp down
     - ② store ret addr at %rsp
     - ③ label → %rip
  2) Jump to *`label`*
- Return address:
  - Address of instruction immediately after **`call`** instruction
  - Example from disassembly:

```
400544: call   400550 <mult2>
400549: movq   %rax,(%rbx)
```

Return address = **`0x400549`**

next instruction happens to be a move, but could be anything

- Procedure return: **`ret`** (special pop)
  1) Pop return address from stack ① read ret addr at %rsp into %rip
  2) Jump to address ② move %rsp up