CS 202 (-002): Operating Systems

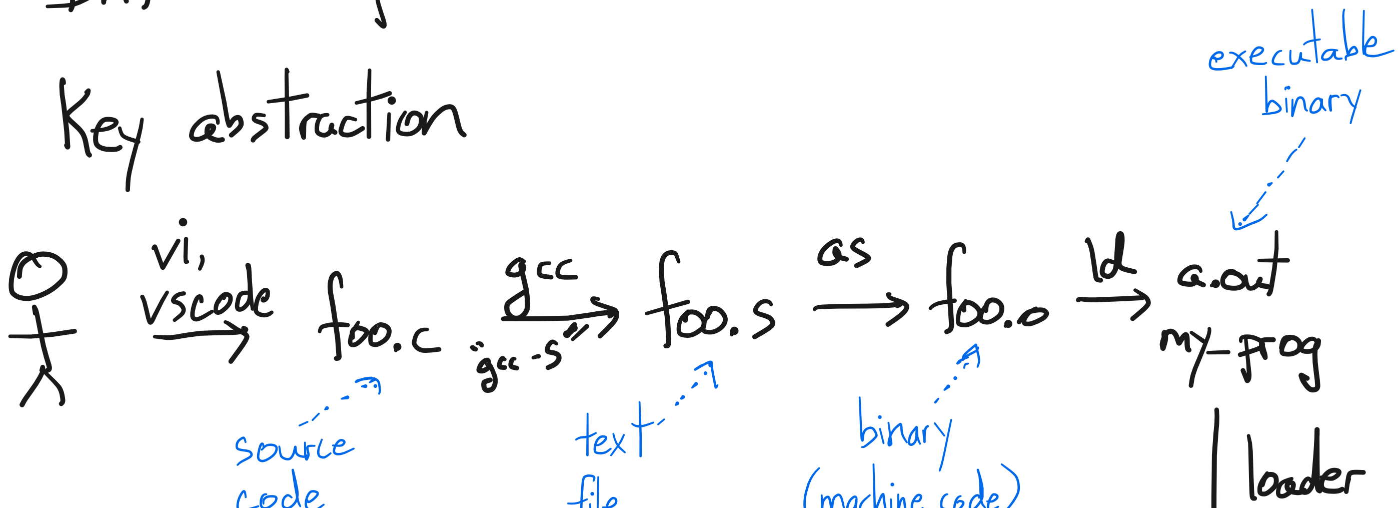
http://cs.nyu.edu/~mwalfish/classes/24fa

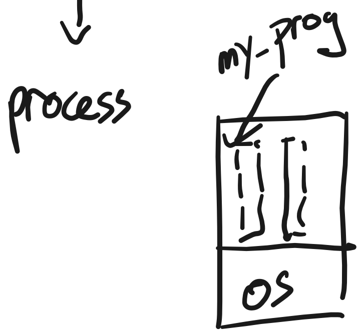
- □ 1. Last time
- □ 2. Intro to processes
- □ 3. Process's view of memory (and registers)
- □ 4. Stack frames
- □ 5. System calls

Today: use the "process's view of the world" to:
- demystify functional scope
- demystify pointers

2. Intro to processes

Key abstraction

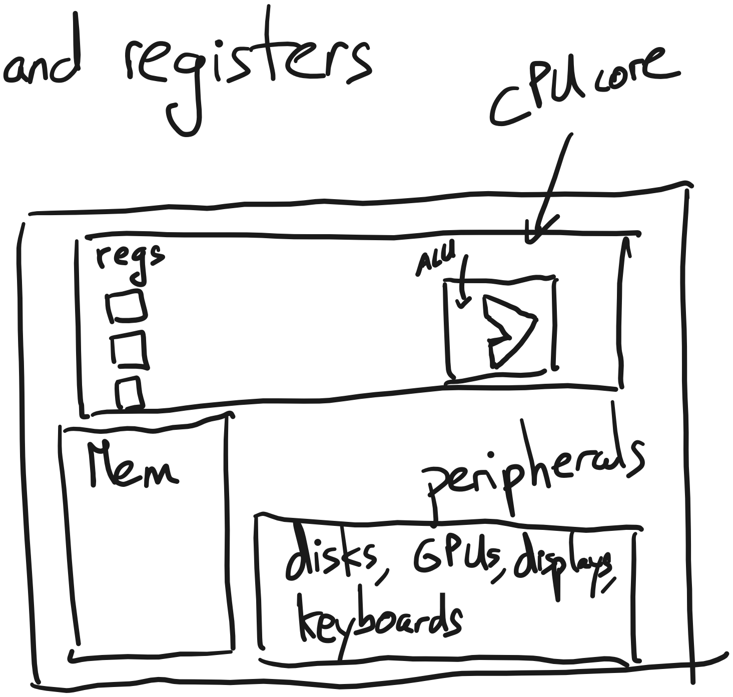




process can be understood in two ways:
- from the process's point of view

'from the OS's point of view

3. Process's view of memory and registers



Background:

registers (x86-64 arch):

general-purpose:

%rax, %rbx, %rcx, %rdx

%rsi, %rdi, %r8 – %r15,

%rsp, %rbp

special-purpose:

%rip

Three special registers:

%rsp: stack pointer

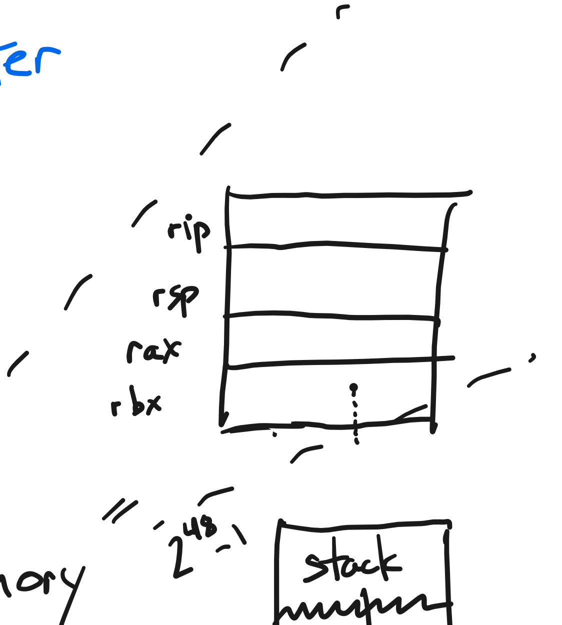
%rbp: base pointer, or frame pointer

%rip: instruction pointer, or program counter

Three aspects to a process:

(i) "each process has its own registers"

(ii) "each process has its own view of memory"

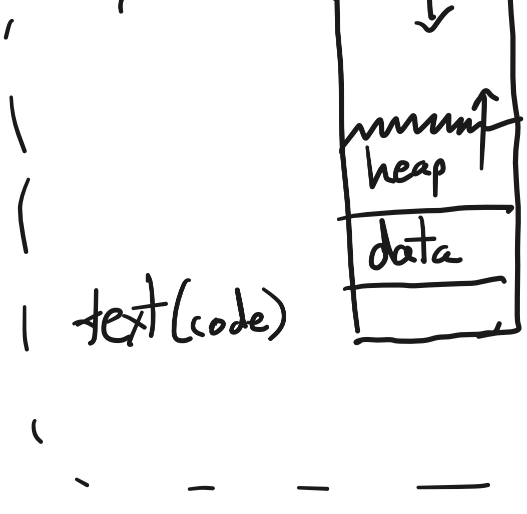


(ii) each process

(iii) very little else needed!

some associated info:

- signal state
- UID, signal mask, whether being debugged, ...



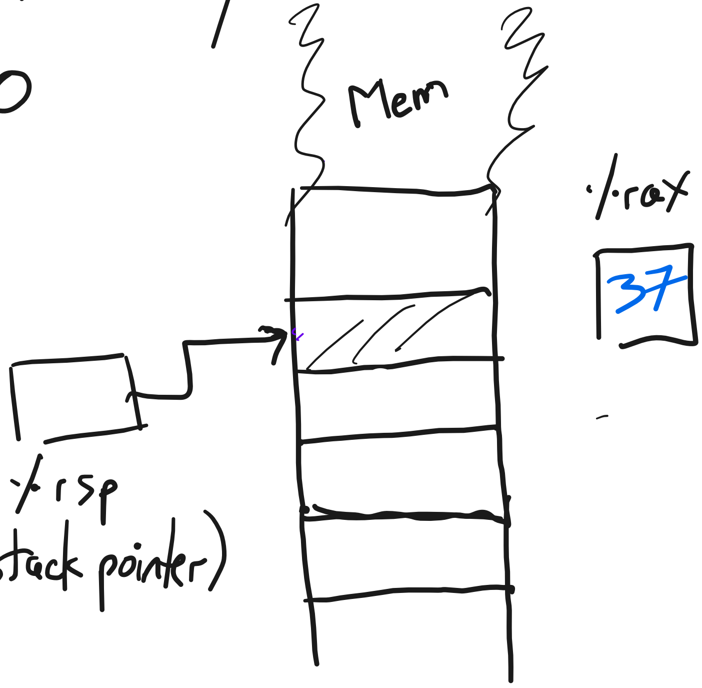
4. stack frames

crash course in X86-64 assembly + stack

movq FROM, TO

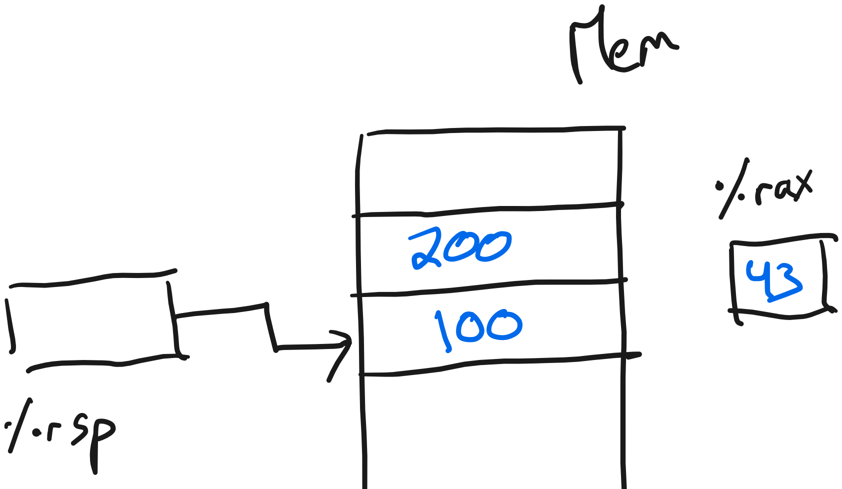
pushq %rax ≡

read it as "push FROM"

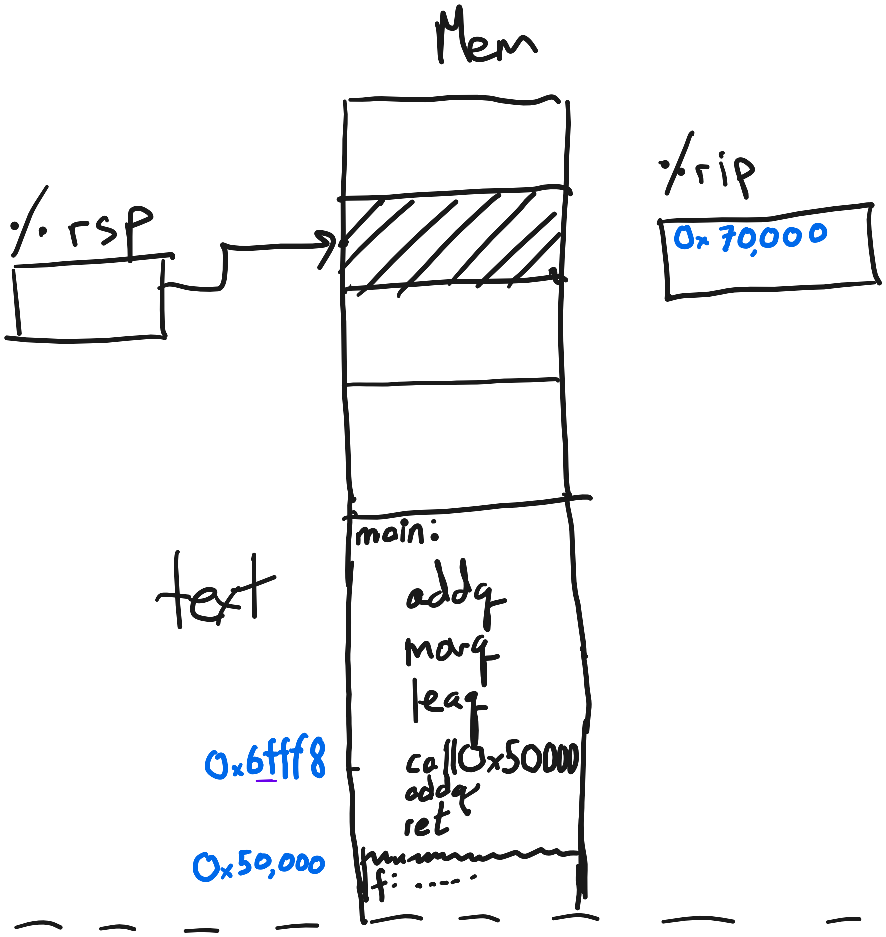


popq %rax ≡

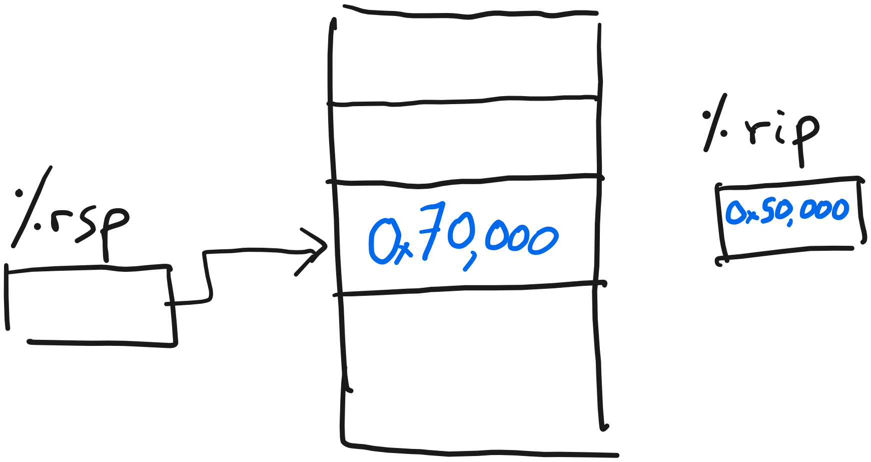
read it as "pop TO"



call 0x50,000 ≡

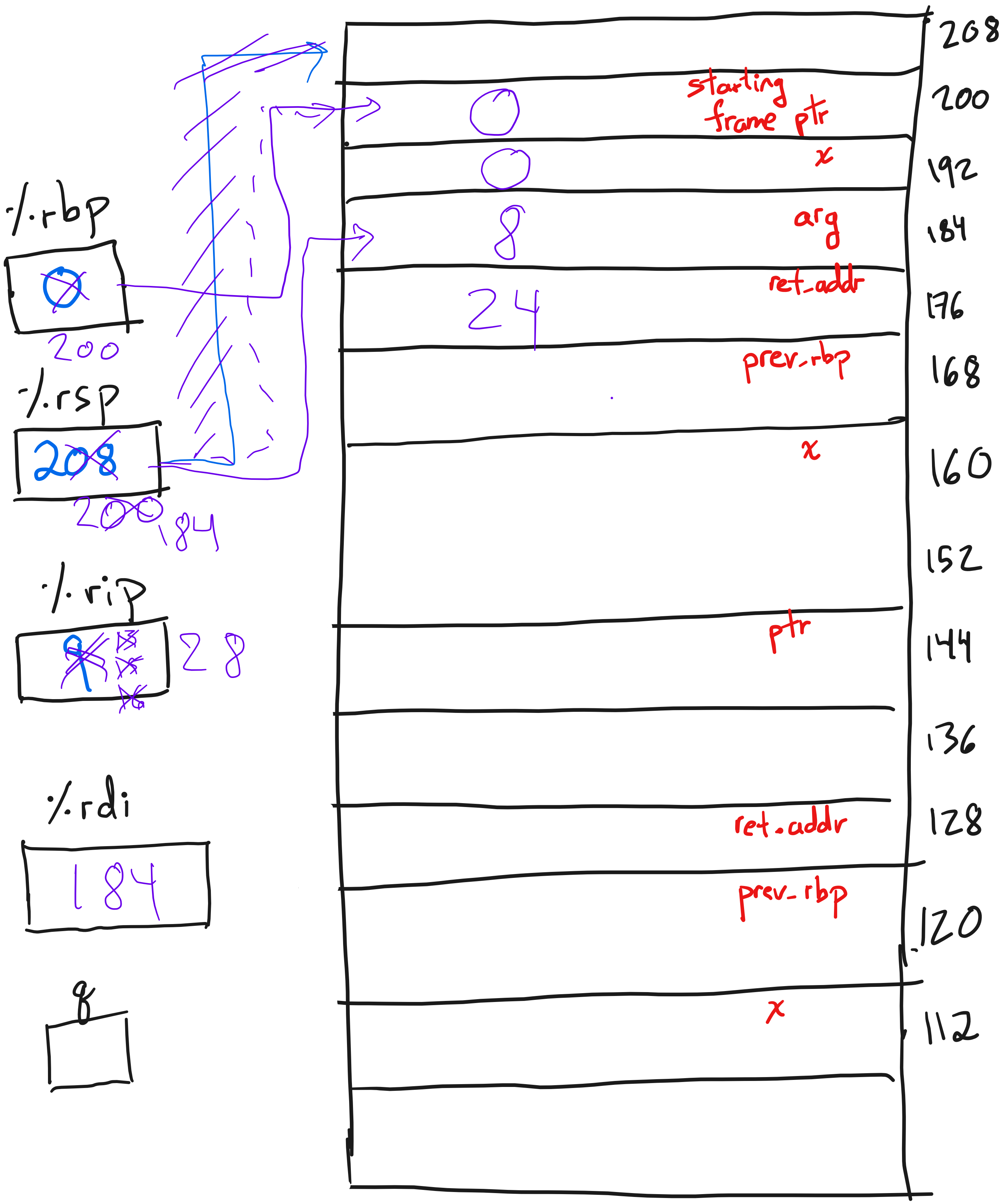


ret ≡

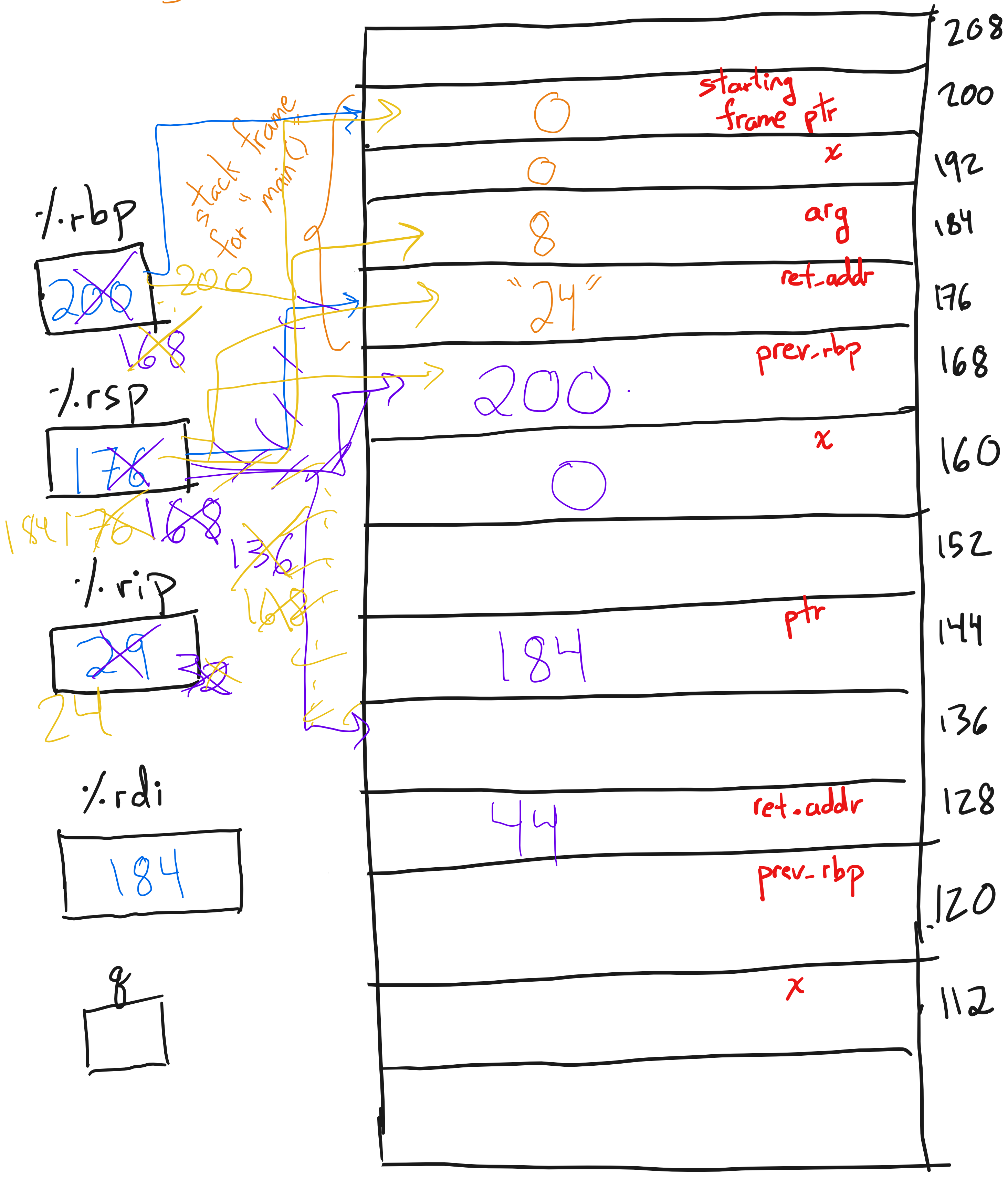


Example (the handout)

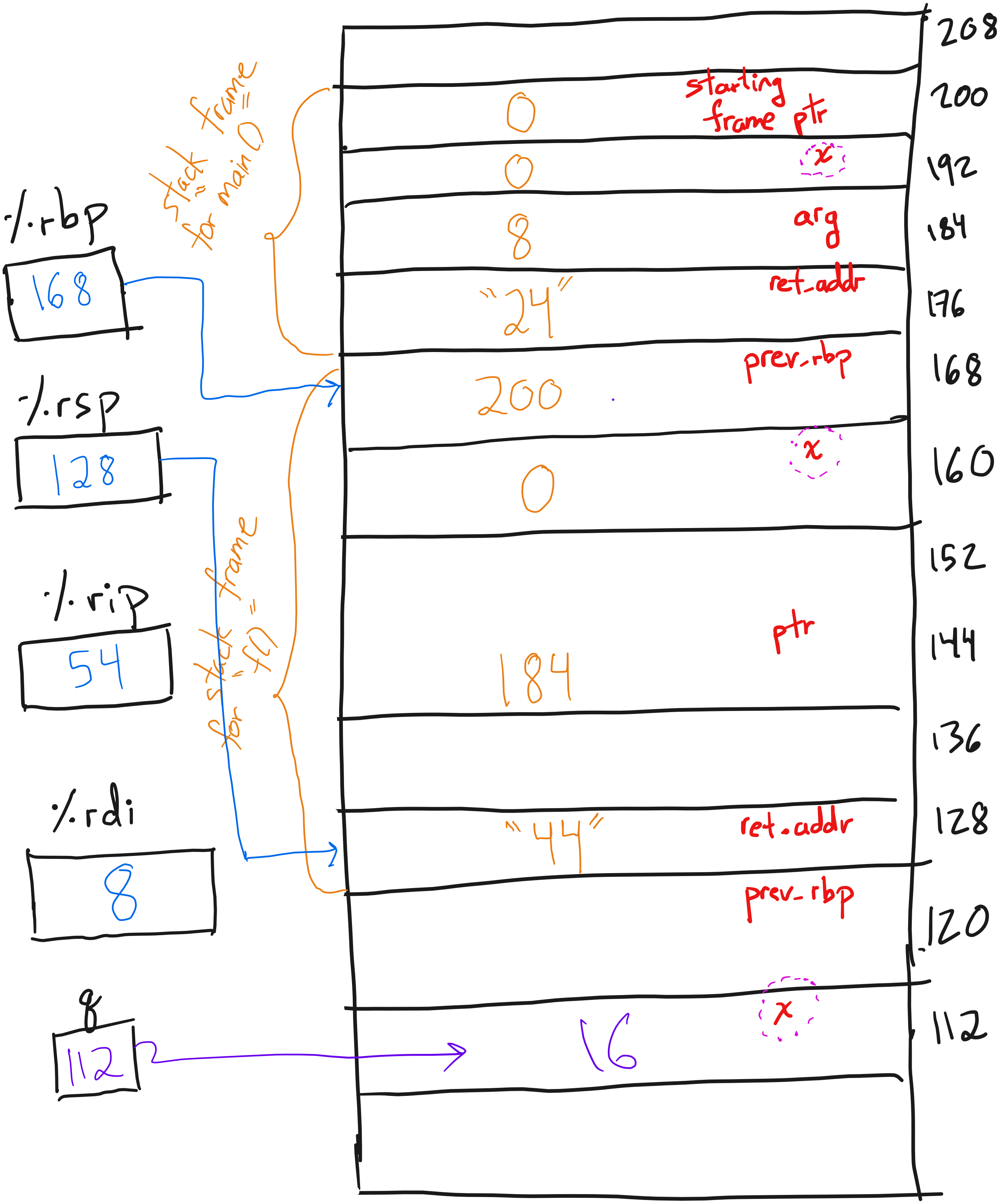
Example (the handout)



Right after f() is called (via "call f")



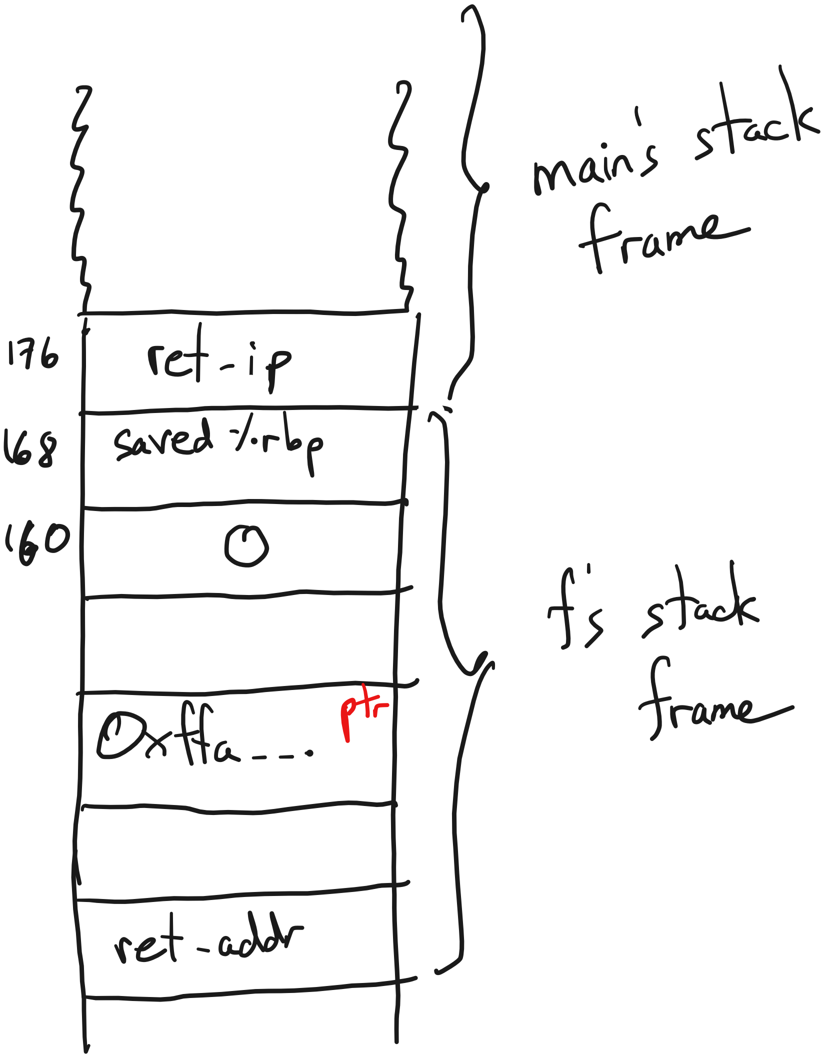
Right after g() is called (via "call g")



NOTE: all those x are different!

- What happens right before g is called?
- Right after g is called?

What does the world look like to function f() right after g() returns? What are %rbp and %rsp?



Calling conventions:

Call-preserved (aka "callee-save"): %rbx, %rbp, %r12 - %r15

Call-clobbered (aka "caller-save"): everything else

```
/* CS202 -- handout 1
 *   compile and run this code with:
 *   $ gcc -g -Wall -o example example.c
 *   $ ./example
 *
 *   examine its assembly with:
 *   $ gcc -O0 -S example.c
 *   $ [editor] example.s
*/

#include <stdio.h>
#include <stdint.h>

uint64_t f(uint64_t* ptr);
uint64_t g(uint64_t a);
uint64_t* q;

int main(void)
{
    uint64_t x = 0;
    uint64_t arg = 8;

    x = f(&arg);

    printf("x: %lu\n", x);
    printf("dereference q: %lu\n", *q);

    return 0;
}

uint64_t f(uint64_t* ptr)
{
    uint64_t x = 0;
    x = g(*ptr);
    return x + 1;
}

uint64_t g(uint64_t a)
{
    uint64_t x = 2*a;
    q = &x;   // <-- THIS IS AN ERROR (AKA BUG)
    return x;
}
```

```
2. A look at the assembly...

    To see the assembly code that the C compiler (gcc) produces:
        $ gcc -O0 -S example.c
    (then look at example.s.)
    NOTE: what we show below is not exactly what gcc produces. We have
    simplified, omitted, and modified certain things.

    main:
        pushq   %rbp               # prologue: store caller’s frame pointer
        movq    %rsp, %rbp         # prologue: set frame pointer for new frame

        subq    $16, %rsp          # prologue: make stack space

        movq    $0, -8(%rbp)       # x = 0 (x lives at address rbp - 8)
        movq    $8, -16(%rbp)      # arg = 8 (arg lives at address rbp - 16)

        leaq   -16(%rbp), %rdi     # load the address of (rbp-16) into %rdi
                                   # this implements "get ready to pass (&arg)
                                   # to f"

        call    f                  # invoke f

        movq    %rax, -8(%rbp)     # x = (return value of f)

        # eliding the rest of main()

    f:
        pushq   %rbp               # prologue: store caller’s frame pointer
        movq    %rsp, %rbp         # prologue: set frame pointer for new frame

        subq   $32, %rsp           # prologue: make stack space
        movq   %rdi, -24(%rbp)     # Move ptr to the stack
                                   # (ptr now lives at rbp - 24)
        movq   $0, -8(%rbp)        # x = 0 (x’s address is rbp - 8)

        movq    -24(%rbp), %r8     # move ’ptr’ to %r8
        movq    (%r8), %r9         # dereference ’ptr’ and save value to %r9
        movq    %r9, %rdi          # Move the value of *ptr to rdi,
                                   # so we can call g

        call    g                  # invoke g

        movq   %rax, -8(%rbp)      # x = (return value of g)
        movq   -8(%rbp), %r10      # compute x + 1, part I
        addq    $1, %r10           # compute x + 1, part II
        movq    %r10, %rax         # Get ready to return x + 1

        movq    %rpb, %rsp         # epilogue: undo stack frame
        popq    %rbp               # epilogue: restore frame pointer from caller
        ret                        # return

    g:
        pushq   %rbp               # prologue: store caller’s frame pointer
        movq    %rsp, %rbp         # prologue: set frame pointer for new frame
        subq    $0x8, %rsp         # prologue: make stack space

        ....

        movq    %rbp, %rsp         # epilogue: undo stack frame
        popq    %rbp               # epilogue: restore frame pointer from caller
        ret                        # return
```