

CS 202C-001: Operating Systems

<http://cs.nyu.edu/~mwaldfish/classes/25sp>

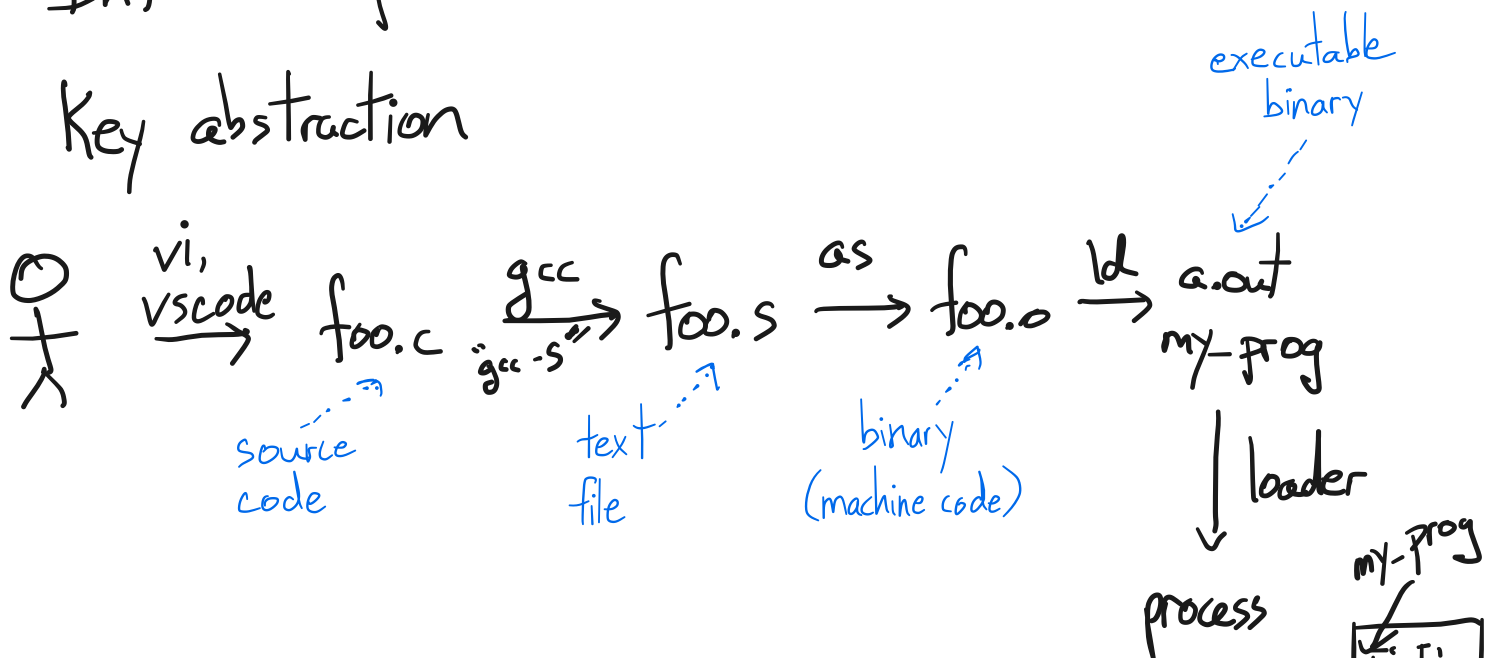
- 1. Last time
- 2. Intro to processes
- 3. Process's view of memory (and registers)
- 4. Stack frames
- 5. System calls

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

CPU core

Background:

registers (x86-64 arch):

general-purpose:

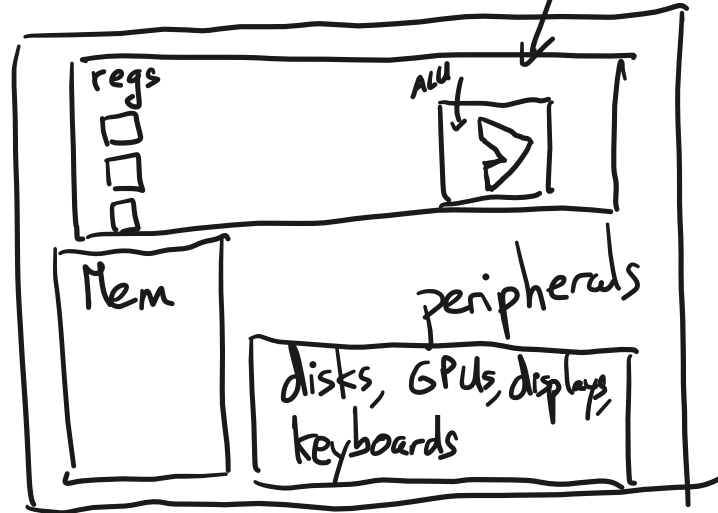
`%rax`, `%rbx`, `%rcx`, `%rdx`

`%rsi`, `%rdi`, `%r8` - `%r15`,

`%rsp`, `%rbp`

special-purpose:

`%rip`



Three special registers:

`%rsp`: stack pointer

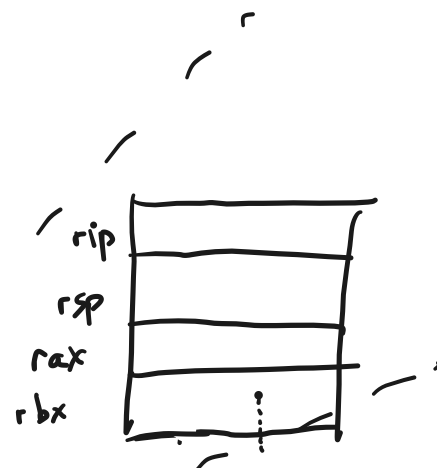
`%rbp`: base pointer, or frame pointer

`%rip`: instruction pointer, or program counter

Three aspects to a process:

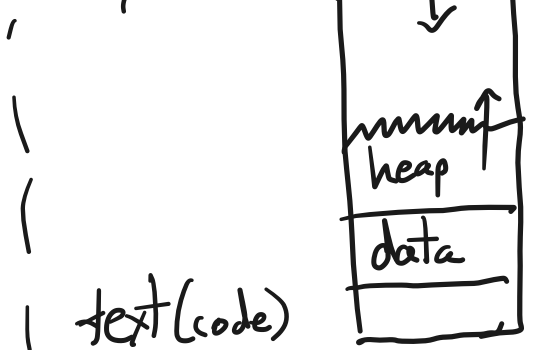
(i) "each process has its own registers"

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



248

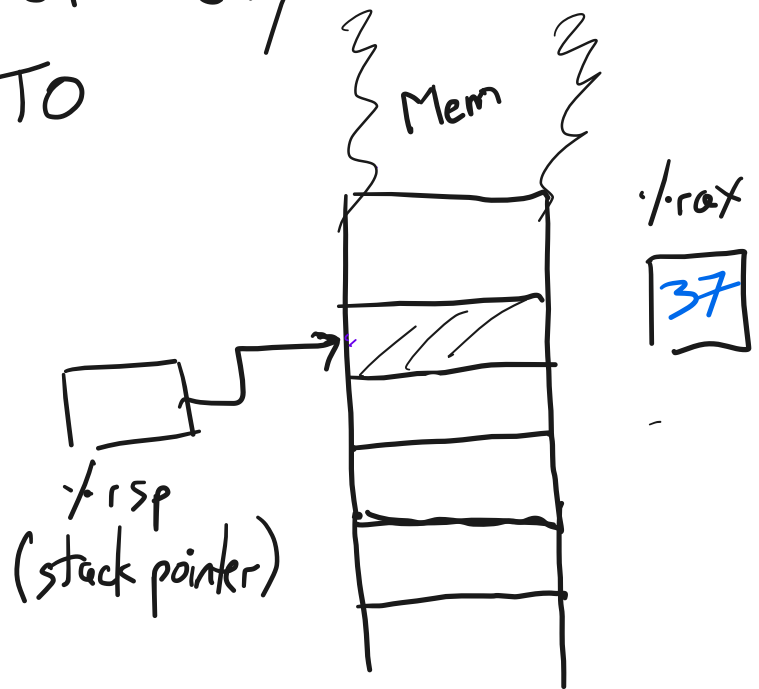
- (ii) each process has its own...
- (iii) very little else needed!
 some associated info:
- signal state
 - UID, signal mask, whether being debugged, ...



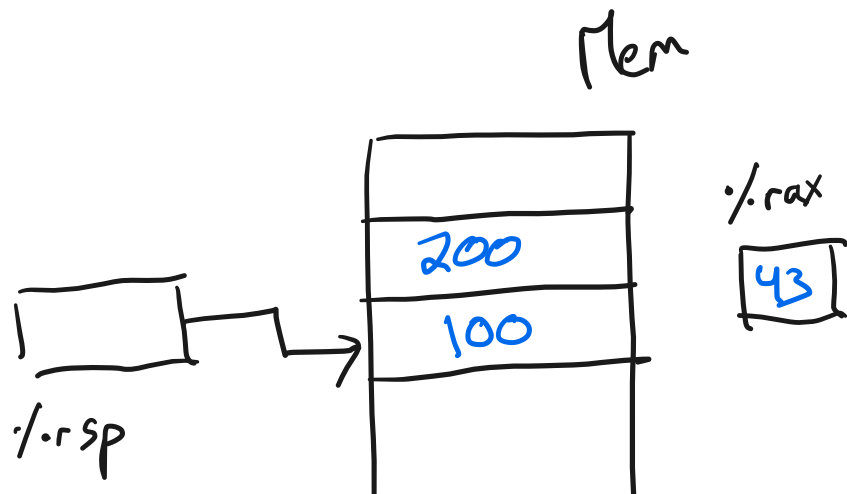
4. stack frames

crash course in X86-64 assembly + stack
 moving FROM, TO

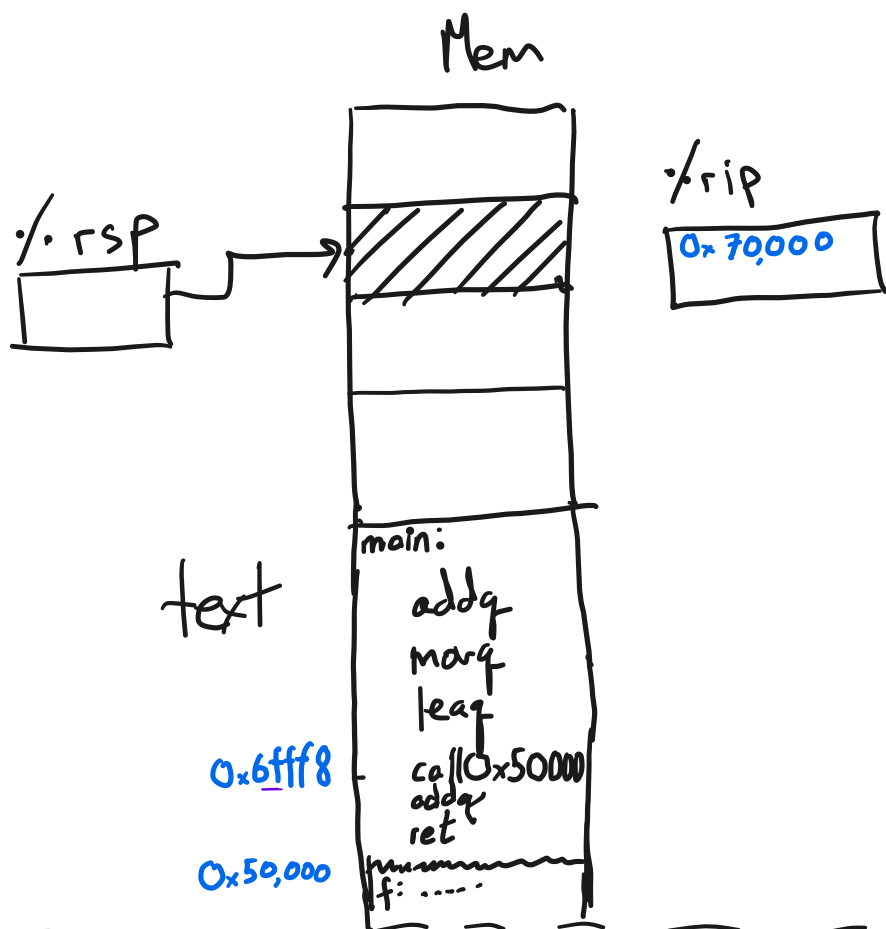
pushq %rax $\equiv$
 read it as "push FROM"



popq %rax $\equiv$
 read it as "pop TO"



call 0x50,000 ≡



ret ≡

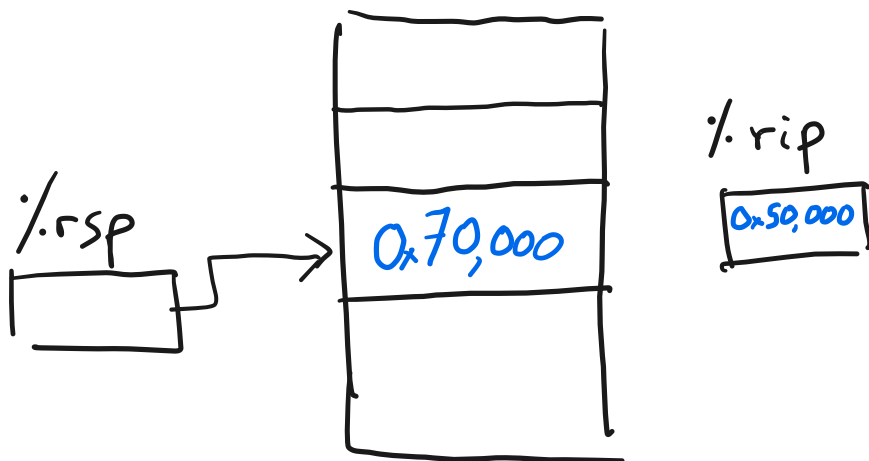
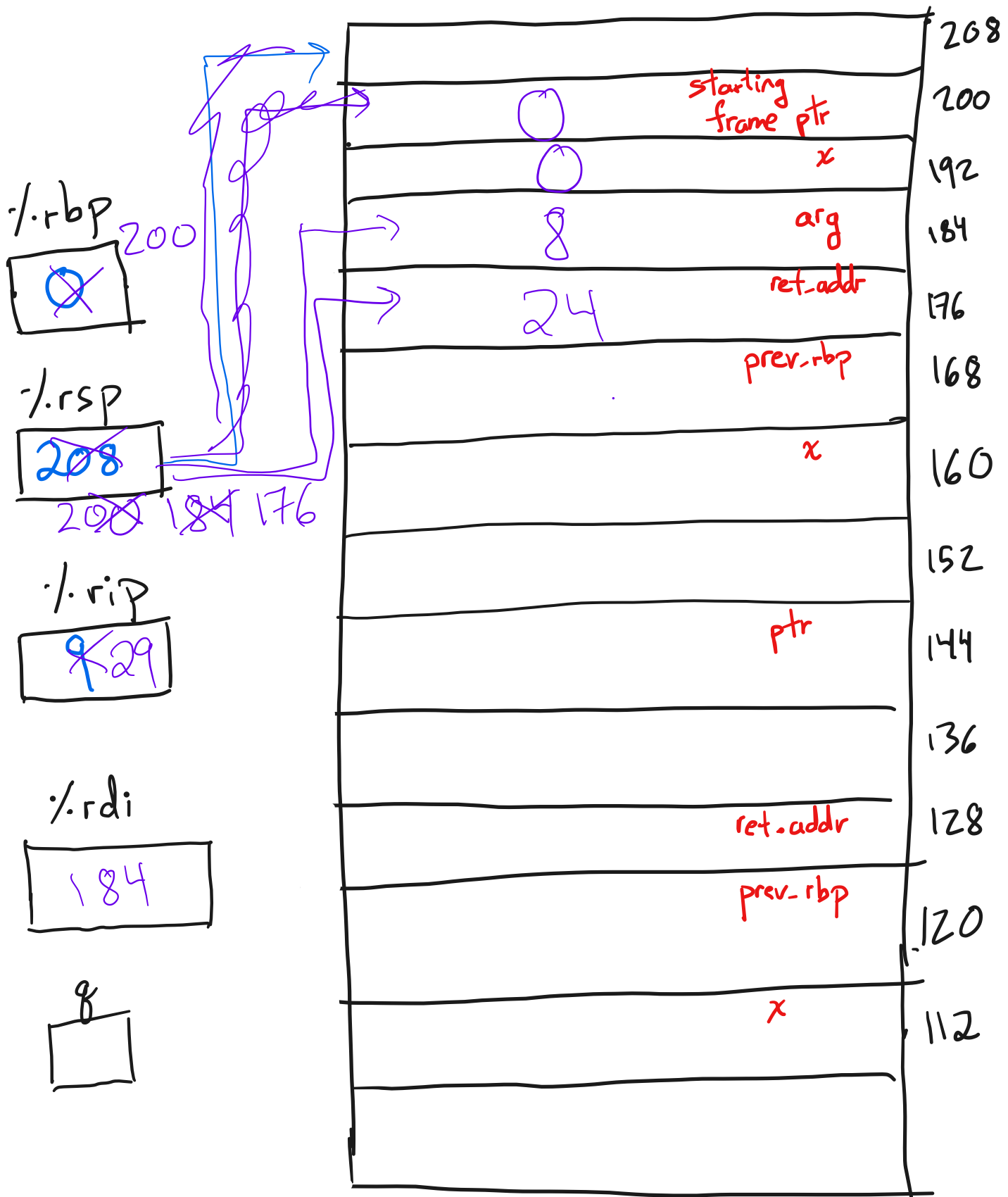
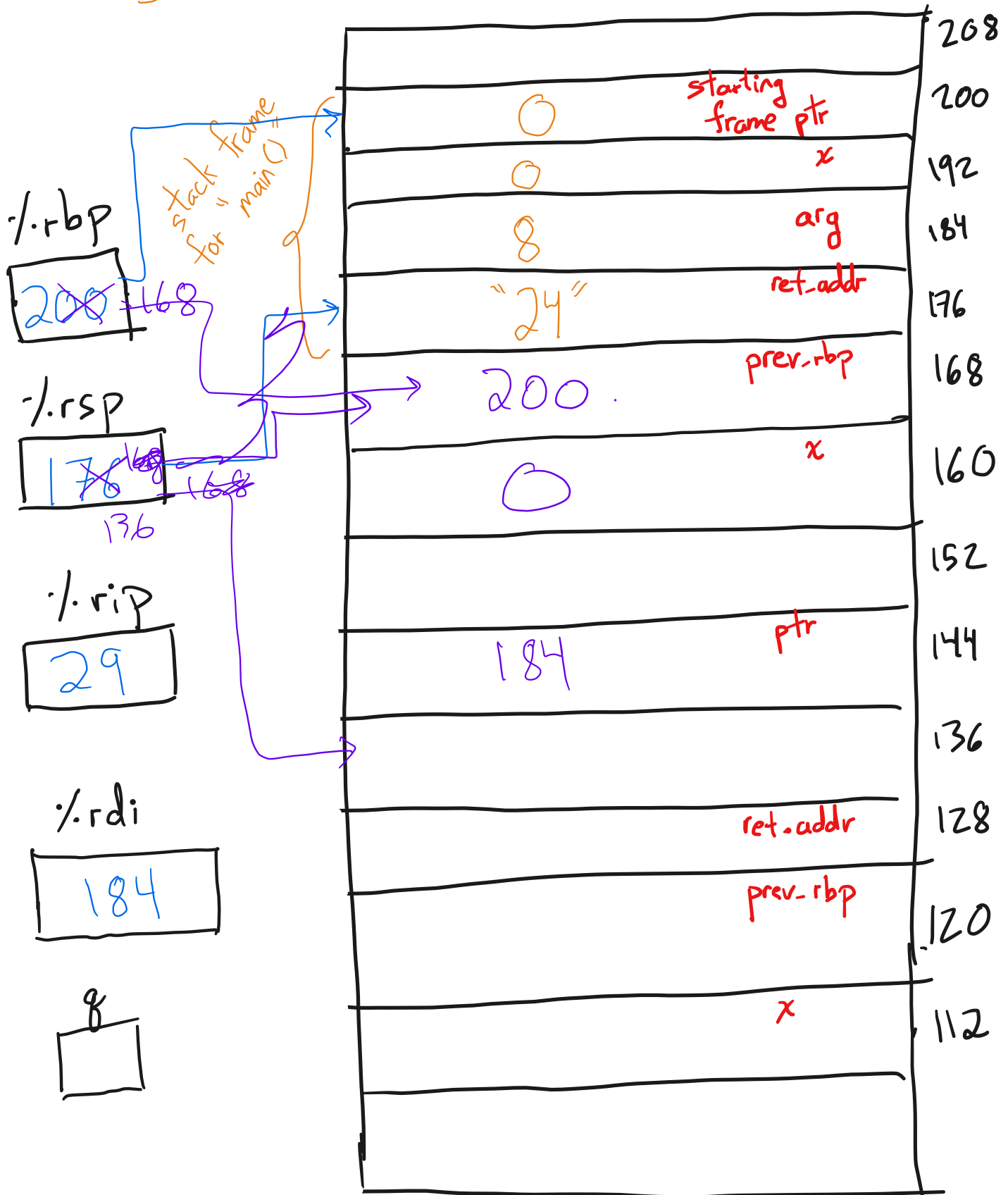


Fig. 1. (the handout)

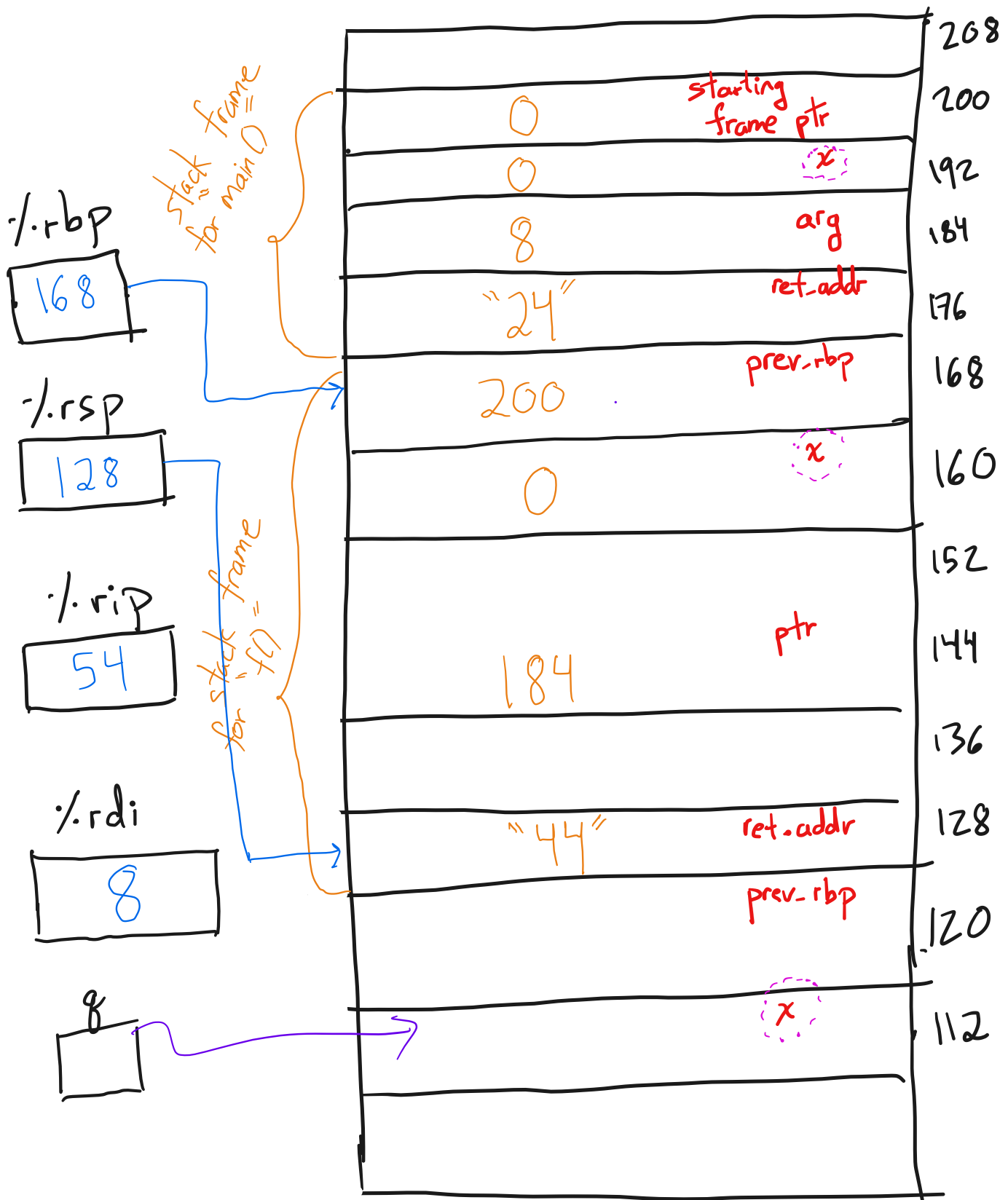
Example time



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



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

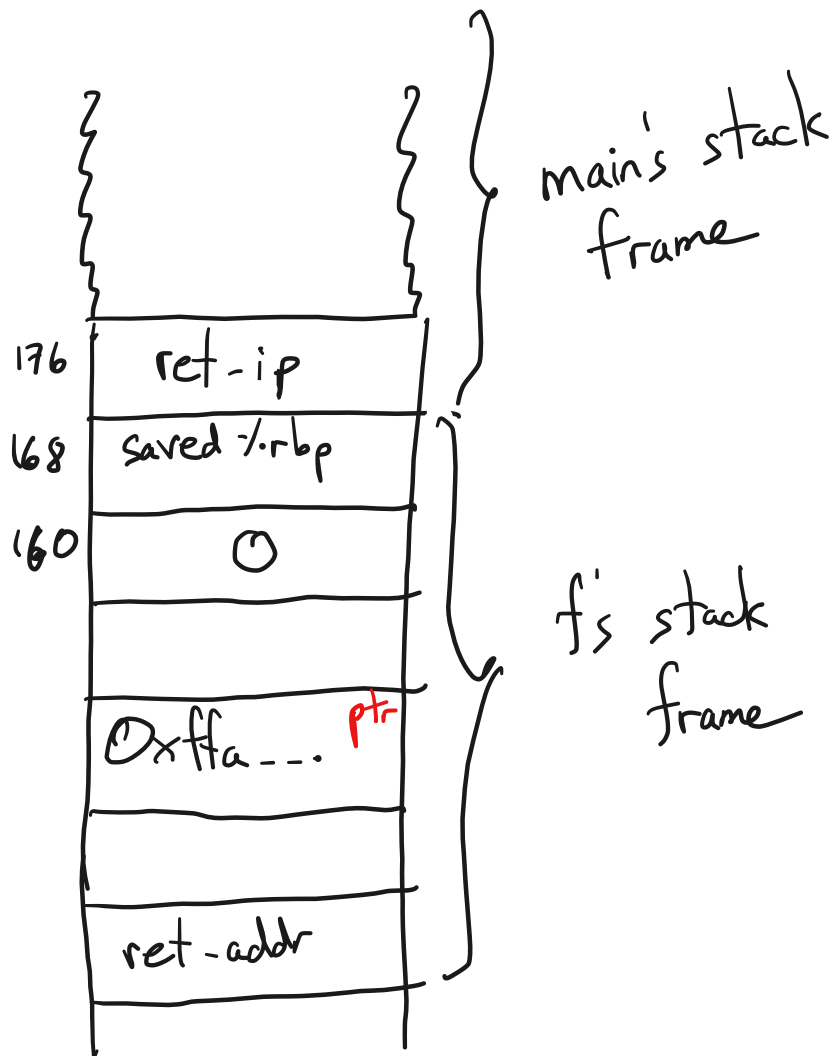


NOTE: all those (x) are different!

• 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

Sep 08, 24 22:52

example.c

Page 1/1

```

1  /* CS202 -- handout 1
2  *   compile and run this code with:
3  *   $ gcc -g -Wall -o example example.c
4  *   $ ./example
5  *
6  *   examine its assembly with:
7  *   $ gcc -O0 -S example.c
8  *   $ [editor] example.s
9  */
10
11 #include <stdio.h>
12 #include <stdint.h>
13
14 uint64_t f(uint64_t* ptr);
15 uint64_t g(uint64_t a);
16 uint64_t* q;
17
18 int main(void)
19 {
20     uint64_t x = 0;
21     uint64_t arg = 8;
22
23     x = f(&arg);
24
25     printf("x: %lu\n", x);
26     printf("dereference q: %lu\n", *q);
27
28     return 0;
29 }
30
31 uint64_t f(uint64_t* ptr)
32 {
33     uint64_t x = 0;
34     x = g(*ptr);
35     return x + 1;
36 }
37
38 uint64_t g(uint64_t a)
39 {
40     uint64_t x = 2*a;
41     q = &x;  // <-- THIS IS AN ERROR (AKA BUG)
42     return x;
43 }

```

Sep 08, 24 22:52

as.txt

Page 1/1

```

1  2. A look at the assembly...
2
3  To see the assembly code that the C compiler (gcc) produces:
4      $ gcc -O0 -S example.c
5  (then look at example.s.)
6  NOTE: what we show below is not exactly what gcc produces. We have
7  simplified, omitted, and modified certain things.
8
9  main:
10     pushq   %rbp                # prologue: store caller's frame pointer
11     movq    %rsp, %rbp          # prologue: set frame pointer for new frame
12
13     subq    $16, %rsp           # prologue: make stack space
14
15     movq    $0, -8(%rbp)        # x = 0 (x lives at address rbp - 8)
16     movq    $8, -16(%rbp)       # arg = 8 (arg lives at address rbp - 16)
17
18     leaq    -16(%rbp), %rdi     # load the address of (rbp-16) into %rdi
19                                 # this implements "get ready to pass (&arg)
20                                 # to f"
21
22     call    f                   # invoke f
23
24     movq    %rax, -8(%rbp)      # x = (return value of f)
25
26     # eliding the rest of main()
27
28  f:
29     pushq   %rbp                # prologue: store caller's frame pointer
30     movq    %rsp, %rbp          # prologue: set frame pointer for new frame
31
32     subq    $32, %rsp           # prologue: make stack space
33                                 # Move ptr to the stack
34     movq    %rdi, -24(%rbp)     # (ptr now lives at rbp - 24)
35                                 # x = 0 (x's address is rbp - 8)
36
37     movq    -24(%rbp), %r8      # move 'ptr' to %r8
38     movq    (%r8), %r9          # dereference 'ptr' and save value to %r9
39     movq    %r9, %rdi           # Move the value of *ptr to rdi,
40                                 # so we can call g
41
42     call    g                   # invoke g
43
44     movq    %rax, -8(%rbp)      # x = (return value of g)
45     movq    -8(%rbp), %r10      # compute x + 1, part I
46     addq    $1, %r10            # compute x + 1, part II
47     movq    %r10, %rax         # Get ready to return x + 1
48
49     movq    %rbp, %rsp         # epilogue: undo stack frame
50     popq    %rbp               # epilogue: restore frame pointer from caller
51     ret                                # return
52
53  g:
54     pushq   %rbp                # prologue: store caller's frame pointer
55     movq    %rsp, %rbp          # prologue: set frame pointer for new frame
56     subq    $0x8, %rsp         # prologue: make stack space
57
58     ....
59
60     movq    %rbp, %rsp         # epilogue: undo stack frame
61     popq    %rbp               # epilogue: restore frame pointer from caller
62     ret                                # return

```