☑ 1. Last time

☑ 2. Stack frames, continued

☑ 3. System calls

☑ 4. Process/OS control transfers ←

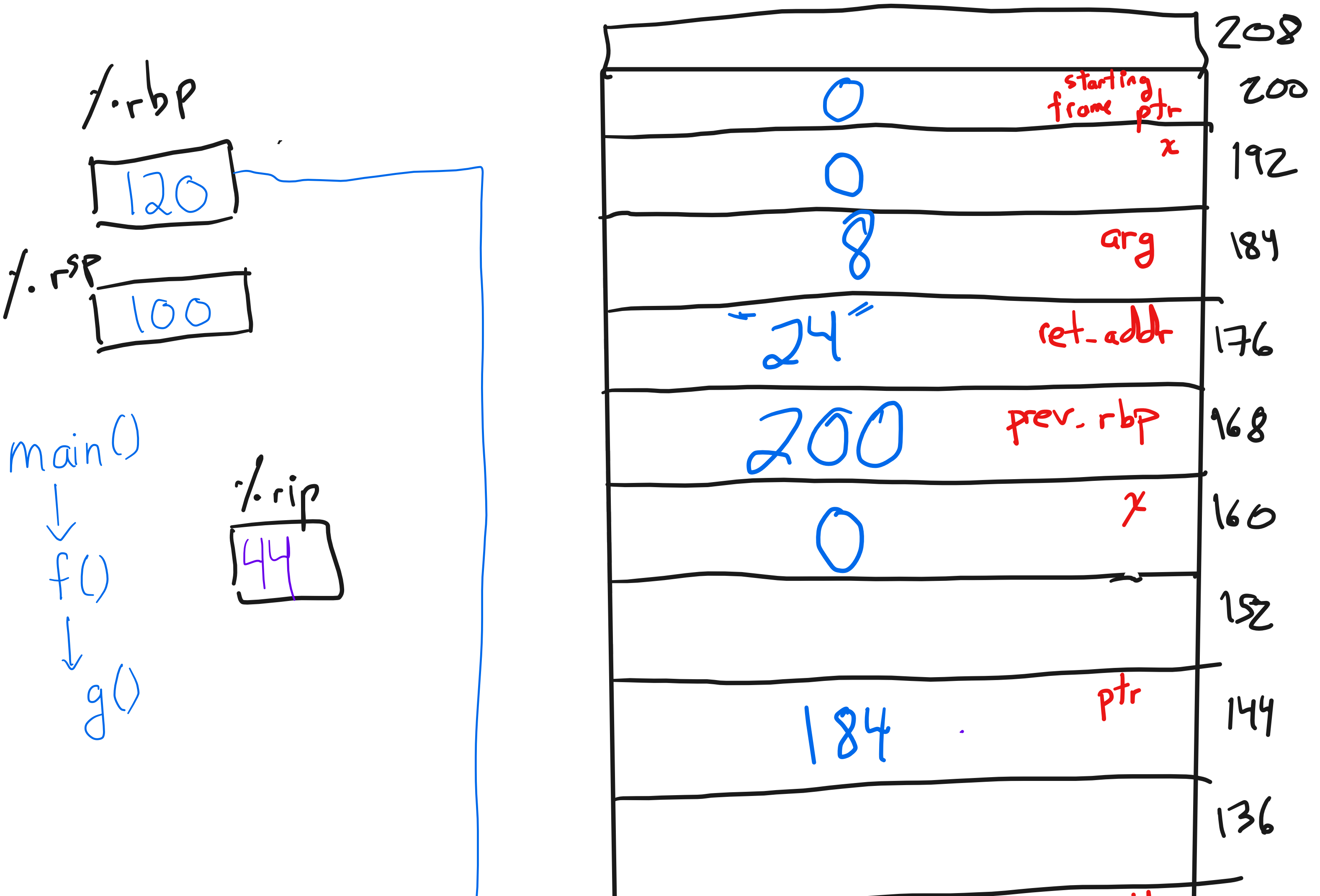
☑ 5. Git/lab setup

☑ 6. Process birth

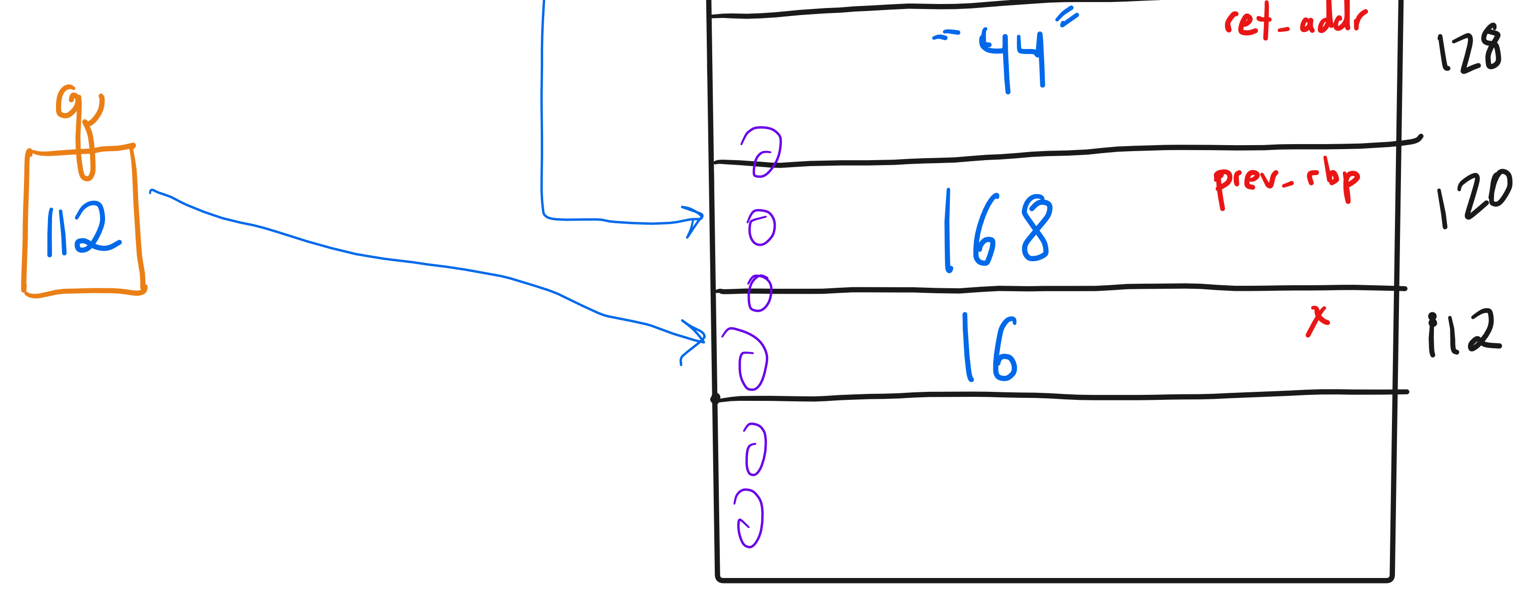
□ 7. Shell, part I

□ 8. File descriptors

□ 9. Shell, part II



		208
0	starting frame ptr	200
0	x	192
8	arg	184
"24"	ret_addr	176
200	prev_rbp	168
0	x	160
		152
184	ptr	144
		136



3. System calls

Examples:

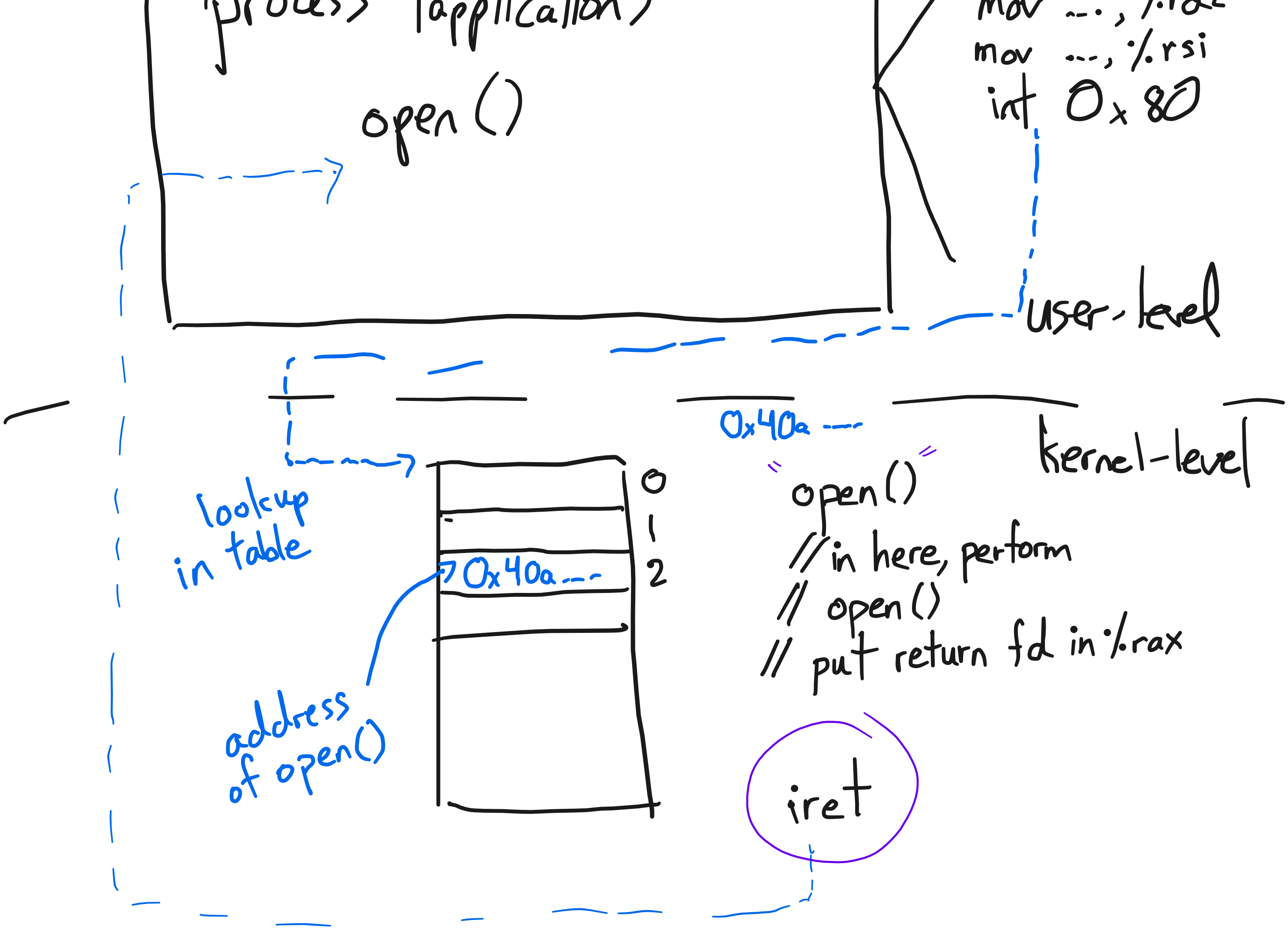
```
int fd = open(const char* path, int flags);
int rc = write(int fd, const void*, size_t s);
int rc = read(int fd, void*, size_t s);
```

```
int fd;
fd = open("/tmp/foo", O_RDWR | O_CREATE);
write(fd, "abc...z", 26);
```

```
$ man 2 open
$ man 2 readdir
```

4. Process/OS control transfers

```
mov $2, %rax
```

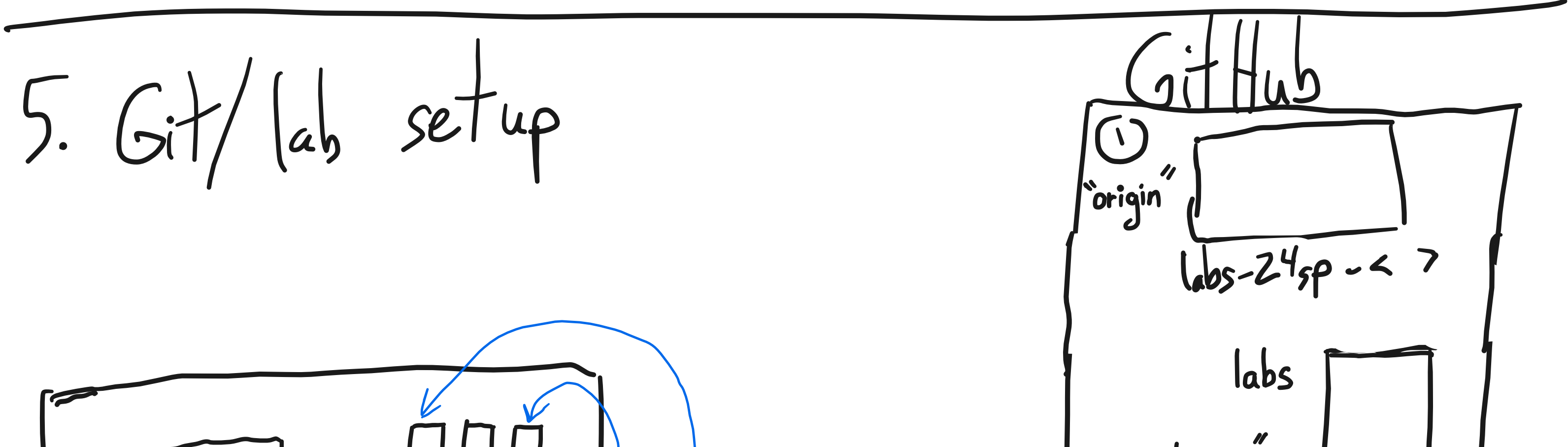


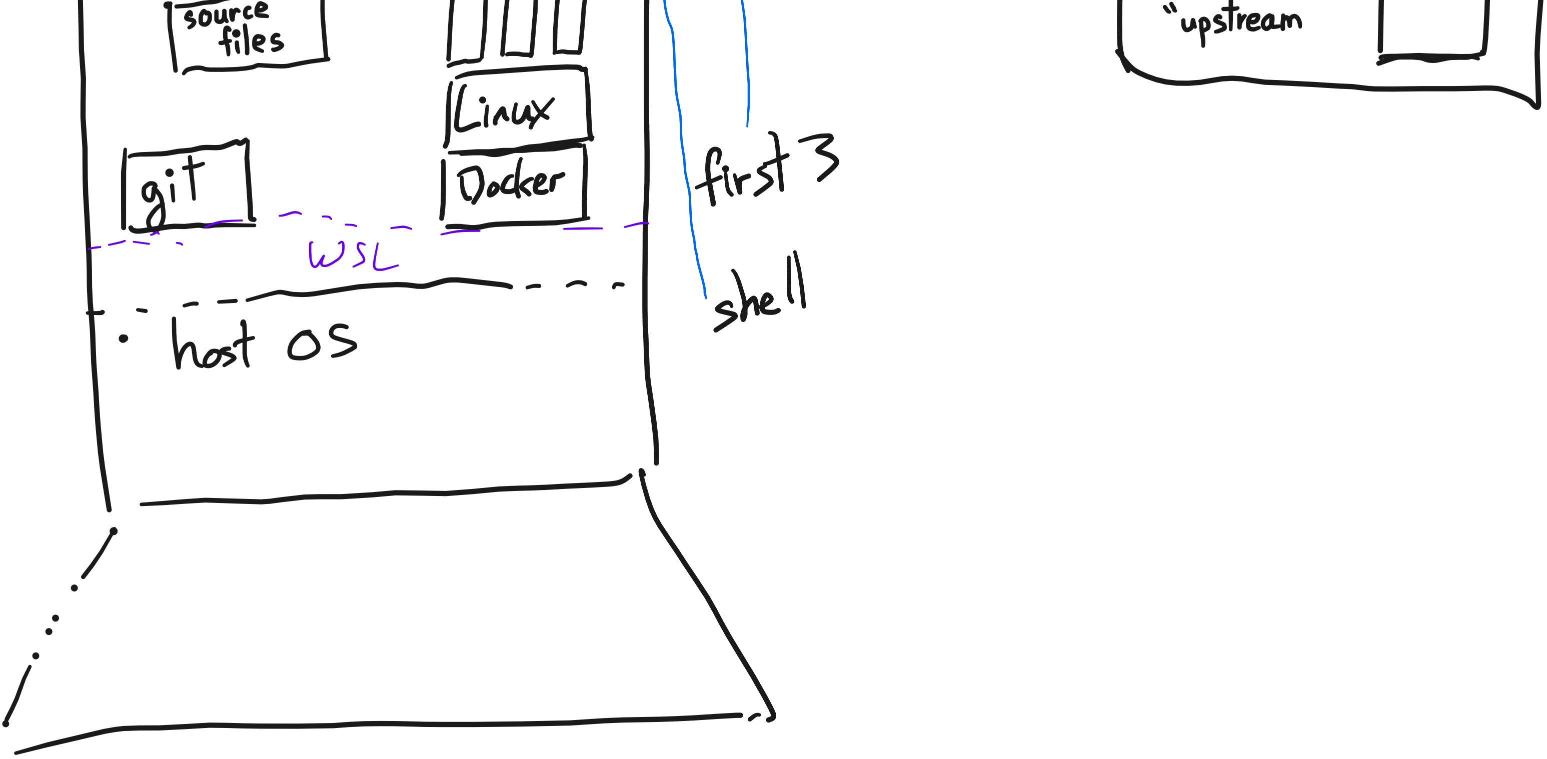
A. System calls

B. Interrupts

C. Exceptions

5. Git/lab setup





⑥ Process birth

```
fork();

switch ( fork() ):

    case 0:

    default:
```

Default

```
        wait();



for (i=0; i< 10; i++ ) {
        fork();

}

  while (1) { }
```

---

(7) The shell, part I

- program that creates processes
- the human's interface to the computer
- GUIs in OSes are another kind of shell

core loop in a text shell:

```
while (1) {
   write(1, "$ ", 2);
```

```
    readcommand(command, args);  // parse input
    if ((pid = fork()) == 0)     // child?
        execve(command, args, 0);
    else if (pid > 0)   // parent?
        wait(0);   // wait for child
    else
        perror("failed to fork()");
}
```

example.c

```c
/* 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;
}
```

as.txt

```
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
```