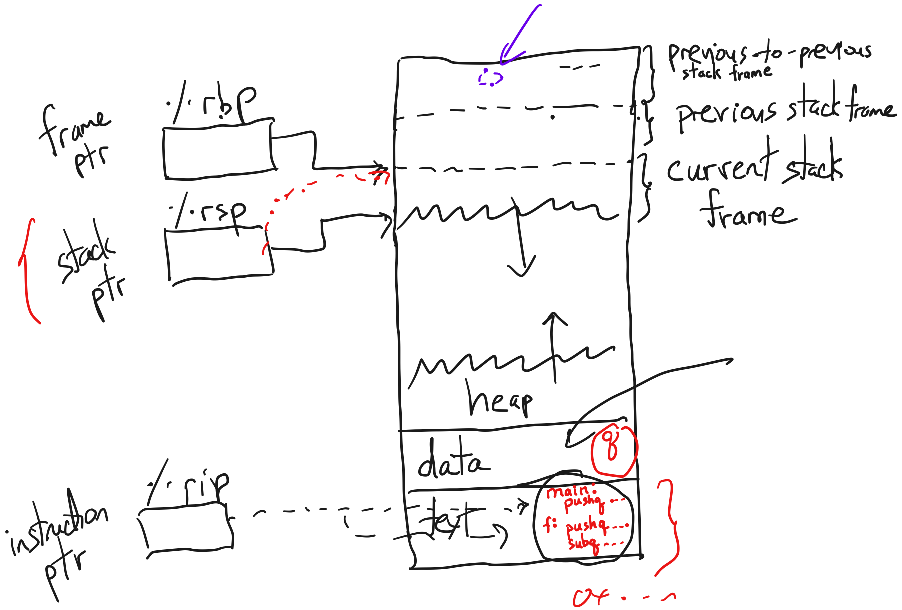
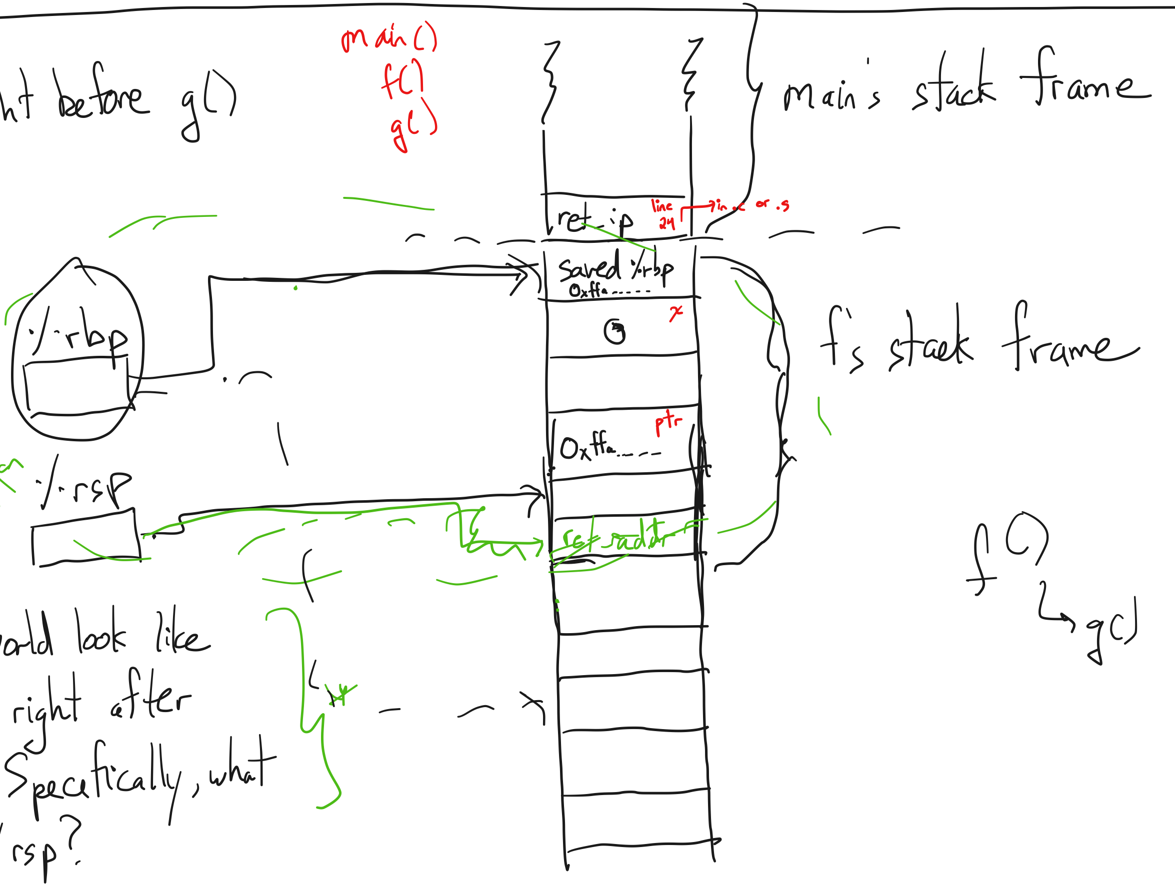
- ☑ 1. Last time
- ☑ 2. Stack frames again
- ☑ 3. System calls
- ☑ 4. Process/OS control transfers
- ☑ 5. Git/lab setup
- ☑ 6. Process birth
- ☐ 7. The shell, part I
- ☐ 8. File descriptors
- ☐ 9. The shell, part II



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

exercise
↳ starts with: ret. addr going on the stack

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



- Calling conventions
  - arguments passed in registers:
    %rdi, %rsi, %rdx, %rcx
  - return value in %rax
  - call-preserved (aka "callee-save"): %rbx, %rbp, %r12-%r15
  - call-clobbered (aka "caller-save"): everything else

pushq %r8
call __
popq %r8

- ... example have not saved any call-clobbered registers.

Our drawn examples have not

For an example that saves call-clobbered, see the notes.

---

3. System calls

Examples:

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

stat()

readdir()

lab3:

mutex...

$ man 2 open

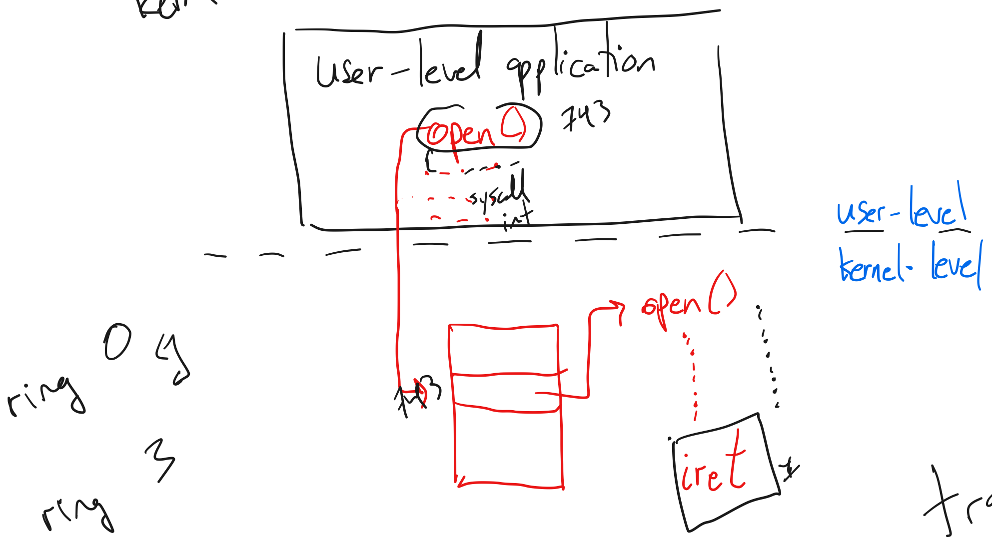
printf();

↳ write()

---

4. Process/OS control transfers

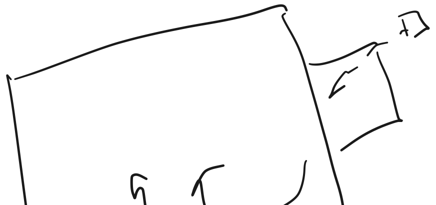
↳ kernel



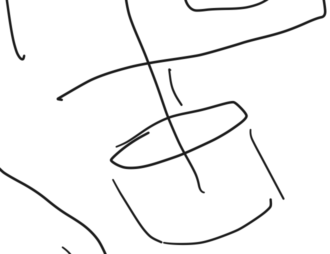
user fn calls	syscalls
call f	syscall 0x23
f	
ret	iret

trapping

A. System calls *



B. Interrupts *



C. Exceptions

↳ cpu → kernel

```
movq $0, %rbx
movq $0x100, (%rbx)
```

div-by-0.

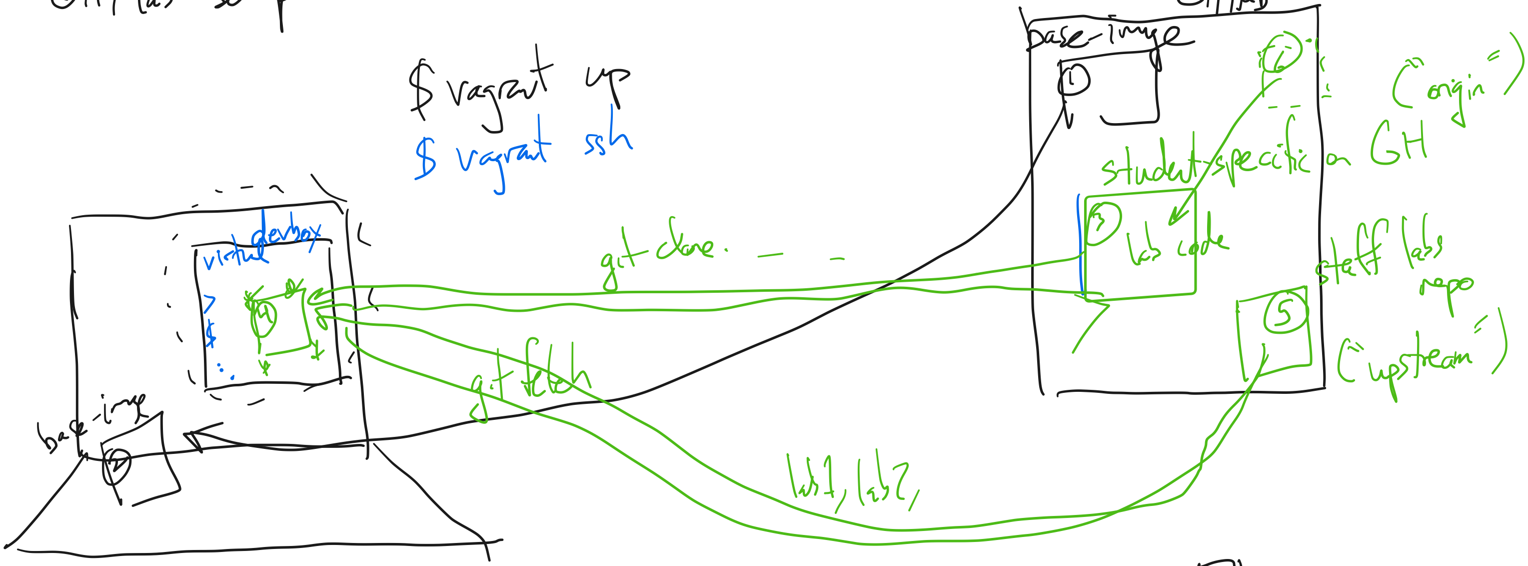
```
int* foo = 0;
*foo = 0x100;
```

---

5. Git/lab setup

$ vagrant up

$ vagrant ssh



returning values through pointers.

%rax (int) func(int, int, int* ret2, float*, llist*...)

node**

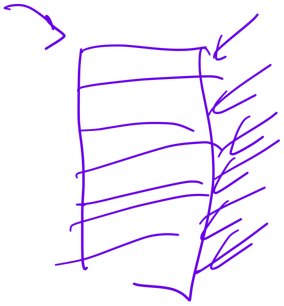
{

main()

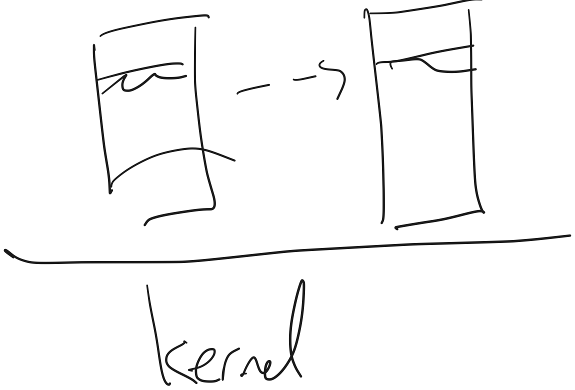
```
int ret2;
func(a, b, &ret2);
```

}

*ret2 = 56;



6. Process birth



fork();

%rax=0

%rax= id of the child

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

i   stack var

%rip

1024

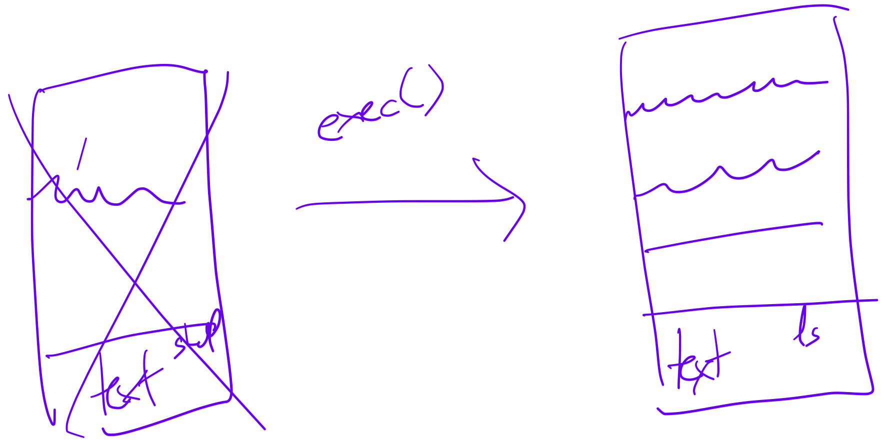
---

7. The shell

fork()

```
$ ls
$ git clone
$ [illegible]
```

exec:



CS 202, Spring 2021
Handout 2 (Class 3)

The handout is meant to:

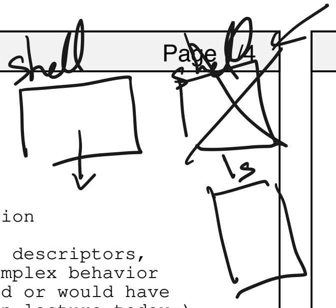
--illustrate how the shell itself uses syscalls

--communicate the power of the fork()/exec() separation

--give an example of how small, modular pieces (file descriptors, pipes, fork(), exec()) can be combined to achieve complex behavior far beyond what any single application designer could or would have specified at design time. (We will not cover pipes in lecture today.)

1. Pseudocode for a very simple 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");
        }
```



$ ls

$ echo

2. Now add two features to this simple shell: output redirection and backgrounding

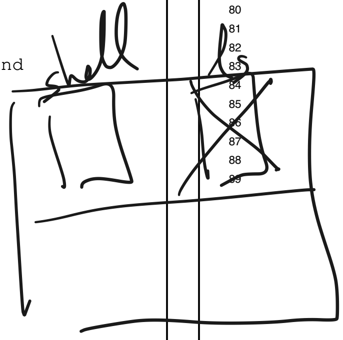
By output redirection, we mean, for example:

```
    $ ls > list.txt
```

By backgrounding, we mean, for example:

```
    $ myprog &
    $
```

```
        while (1) {
            write(1, "$ ", 2);
            readcommand(command, args); // parse input
            if ((pid = fork()) == 0) { // child?
                if (output_redirected) {
                    close(1);
                    open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
                }
                // when command runs, fd 1 will refer to the redirected file
                execve(command, args, 0);
            } else if (pid > 0) { // parent?
                if (foreground_process) {
                    wait(0);  //wait for child
                }
            } else {
                    perror("failed to fork");
            }
        }
```



$ — &

wait(0);

3. Another syscall example: pipe()

The pipe() syscall is used by the shell to implement pipelines, such as

```
    $ ls | sort | head -4
```

We will see this in a moment; for now, here is an example use of pipes.

```
        // C fragment with simple use of pipes

        int fdarray[2];
        char buf[512];
        int n;

        pipe(fdarray);
        write(fdarray[1], "hello", 5);
        n = read(fdarray[0], buf, sizeof(buf));
        // buf[] now contains 'h', 'e', 'l', 'l', 'o'
```

4. File descriptors are inherited across fork

```
        // C fragment showing how two processes can communicate over a pipe

        int fdarray[2];
        char buf[512];
        int n, pid;

        pipe(fdarray);
        pid = fork();
        if(pid > 0){
          write(fdarray[1], "hello", 5);
        } else {
          n = read(fdarray[0], buf, sizeof(buf));
        }
```

5. Putting it all together: implementing shell pipelines using fork(), exec(), and pipe().

```
// Pseudocode for a Unix shell that can run processes in the
// background, redirect the output of commands, and implement
// two element pipelines, such as "ls | sort"

 void main_loop() {

    while (1) {
        write(1, "$ ", 2);
        readcommand(command, args); // parse input
        if ((pid = fork()) == 0) { // child?
            if (pipeline_requested) {
                handle_pipeline(left_command, right_command)
            } else {
                if (output_redirected) {
                    close(1);
                    open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
                }
                exec(command, args, 0);
            }
        } else if (pid > 0) { // parent?
            if (foreground_process) {
                wait(0);  // wait for child
            }
        } else {
                perror("failed to fork");
        }
    }
}

void handle_pipeline(left_command, right_command) {

    int fdarray[2];

    if (pipe(fdarray) < 0) panic ("error");
    if ((pid = fork ()) == 0) {  // child (left end of pipe)

       dup2 (fdarray[1], 1);  // make fd 1 the same as fdarray[1],
                              // which is  the write end of the
                              // pipe. implies close (1).
       close (fdarray[0]);
       close (fdarray[1]);
       parse(command1, args1, left_command);
       exec (command1, args1, 0);

    } else if (pid > 0) {        // parent (right end of pipe)

       dup2 (fdarray[0], 0);  // make fd 0 the same as fdarray[0],
                              // which is the read end of the pipe.
                              // implies close (0).
       close (fdarray[0]);
       close (fdarray[1]);
       parse(command2, args2, right_command);
       exec (command2, args2, 0);

    } else {
       printf ("Unable to fork\n");
    }
}
```

6. Commentary

Why is this interesting? Because pipelines and output redirection are accomplished by manipulating the child's environment, not by asking a program author to implement a complex set of behaviors. That is, the *identical code* for "ls" can result in printing to the screen ("ls -l"), writing to a file ("ls -l > output.txt"), or getting ls's output formatted by a sorting program ("ls -l | sort").

This concept is powerful indeed. Consider what would be needed if it weren't for redirection: the author of ls would have had to anticipate every possible output mode and would have had to build in an interface by which the user could specify exactly how the output is treated.

What makes it work is that the author of ls expressed their code in terms of a file descriptor:

```
    write(1, "some output", byte_count);
```

This author does not, and cannot, know what the file descriptor will represent at runtime. Meanwhile, the shell has the opportunity, *in between fork() and exec()*, to arrange to have that file descriptor represent a pipe, a file to write to, the console, etc.

our_head.c

```c
/*
 * our_head.c -- a C program that prints the first L lines of its input,
 *   where L defaults to 10 but can be specified by the caller of the
 *   program.
 *
 *   (This program is inefficient and does not check its error
 *   conditions. It is meant to illustrate filters aka pipelines.)
 */
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char** argv)
{
    int i = 0;
    int nlines;
    char ch;
    int ret;

    if (argc == 2) {
        nlines = atoi(argv[1]);
    } else if (argc == 1) {
        nlines = 10;
    } else {
        fprintf(stderr, "usage: our_head [nlines]\n");
        exit(1);
    }

    for (i = 0; i < nlines; i++) {

        do {

            /* read in the first character from fd 0 */
            ret = read(0, &ch, 1);

            /* if there are no more characters to read, then exit */
            if (ret == 0) exit(0);

            write(1, &ch, 1);

        } while (ch != '\n');

    }

    exit(0);
}
```

our_yes.c

```c
/*
 * our_yes.c -- a C program that prints its argument to the screen on a
 * new line every second.
 *
 */
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char** argv)
{
    char* repeated;
    int len;

    /* check to make sure the user gave us one argument */
    if (argc != 2) {
        fprintf(stderr, "usage: our_yes string_to_repeat\n");
        exit(1);
    }

    repeated = argv[1];

    len = strlen(repeated);

    /* loop forever */
    while (1) {

        write(1, repeated, len);

        write(1, "\n", 1);

        sleep(1);
    }

}
```

```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;
}
```

%.rax

```
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          # 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           # 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

        ....

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

movq, %.rax