CS202-002: Lecture 2

Processes 1

- Admin:
  - Quiz timing: Last 10 minutes of class!
    First one on Tuesday
- PROCESSES
  - WHAT
  - WHY
  - ABSTRACT MACHINE – EXEC ENVIRONMENT
    - ↳ REGISTERS
    - → MEMORY LAYOUT
- EXECUTION
  - STACK
  - FUNC. CALL & RETURN
  - SYSTEM CALLS
  - OTHER WAYS TO TRANSFER CONTROL

Some of this might be familiar from 201!

PROCESS

- Instance of a running program

BROWSER VSCODE ZOOM ← PROCESSES

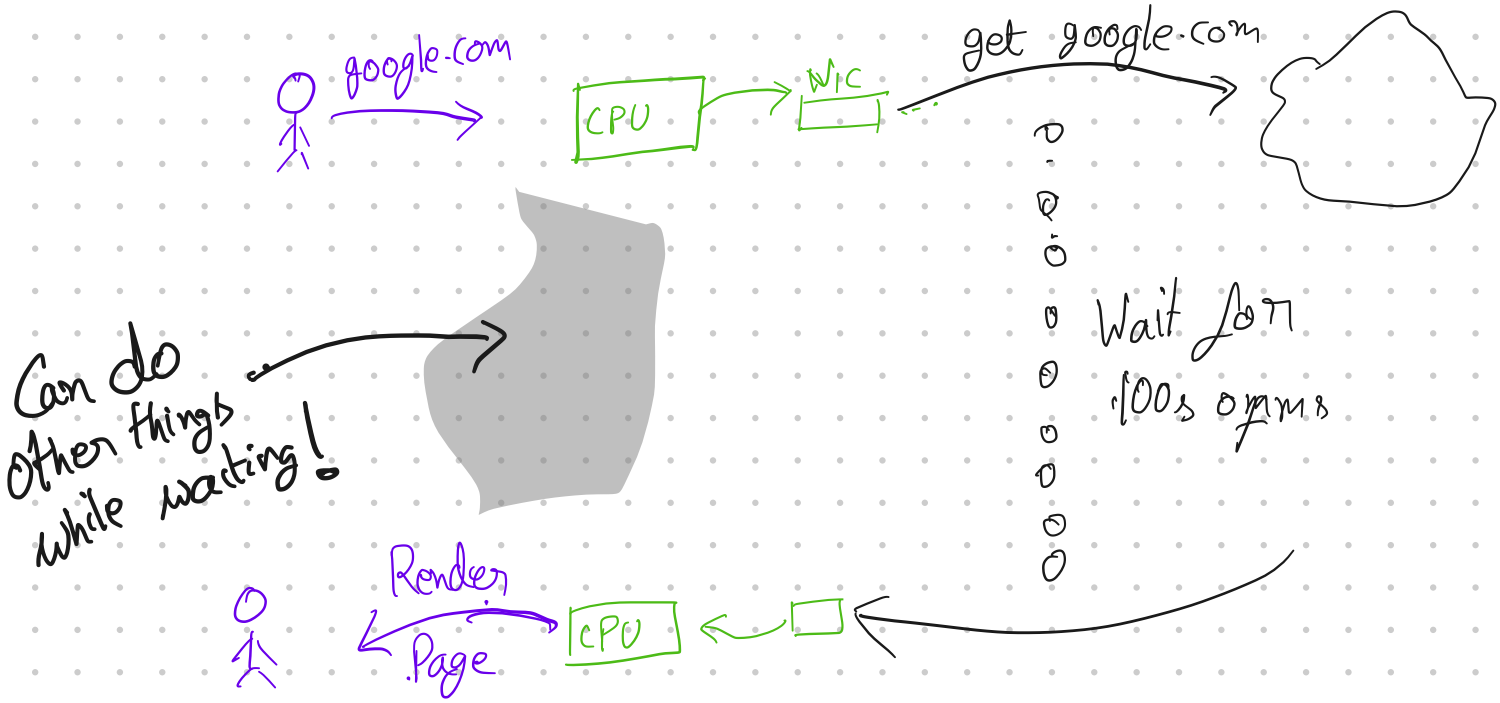
GOAL: RUN MULTIPLE PROCESSES ON ONE COMPUTER

WHY: Discussed this last time, but

(a) Users want to

- Listen to music while working on HW
- Watch Youtube while compiling
- Monitor e-mail while working

⑥ Efficiency (people who manage computers want this)

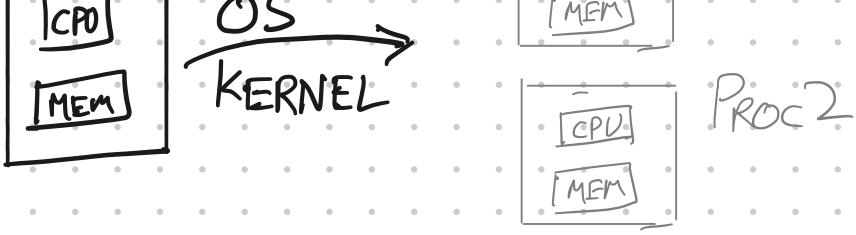


Wait for user to decide what to search for, etc.

What is a process provided

Book (Ch 4)

The OS



virtualizes

physical resources to create an abstract machine.

Two things we could look at

PROCESS	KERNEL

- ABSTRACT MACHINE
  - Memory
  - CPU
  - ...
- Running exactly one program
- Interface to manipulate abstract machine & shared resources
	- How to virtualize?

What does the abstract machine look like & how does it execute programs.

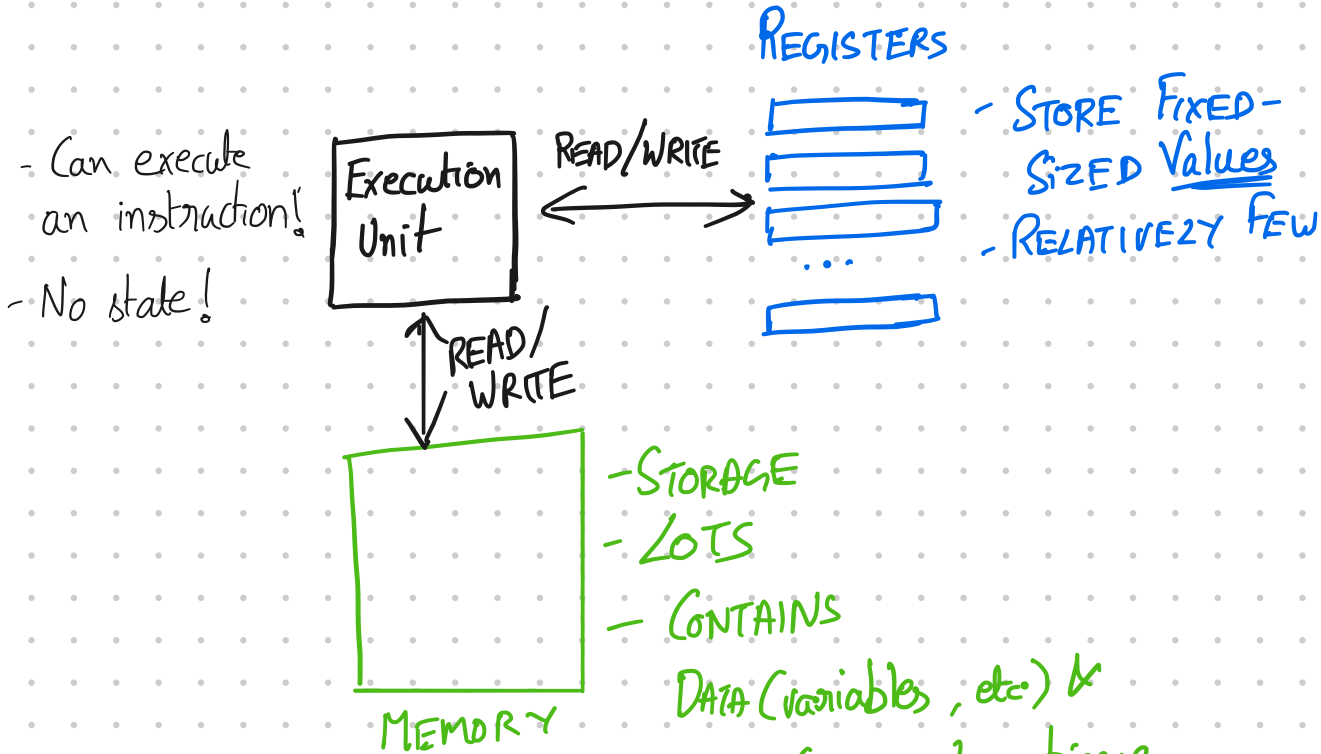
Later in Semester.

Today, Next class

CORE IDEA: EACH PROCESS APPEARS TO EXECUTE ON ITS OWN ABSTRACT MACHINE.

That is, it is isolated from other processes.

What does the abstract machine have



IMPORTANT: All state is in registers or memory.

```
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;
```

- The location of the next instruction.
- The next instruction

36 }

→ The value of x

...

REGISTERS:

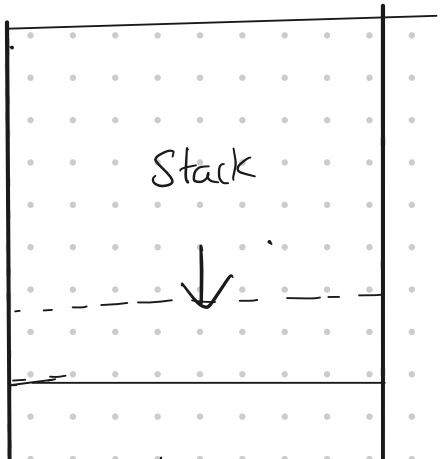
- THIS CLASS x86-64/AMD64.
- Registers in the processes abstract machine

Special significance.

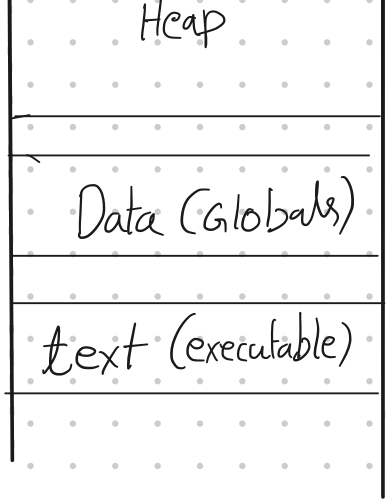
RAX, RBX, RCX, RDX, RDI, R8 – R15, RSP, RBP } PROCESS CAN READ & UPDATE THESE REGISTERS | GENERAL PURPOSE

RIP } POINTS TO NEXT INSTRUCTON TO EXECUTE. READ/MODIFIED BY SPECIAL INSTRUCTIONS CALL/RET/JMP!.. | SPECIAL PURPOSE

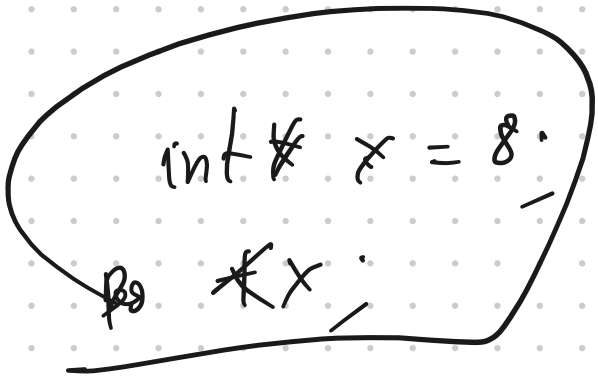
MEMORY



- Layout (what goes where) determined by CONVENTION.
- THE ABSTRACT MACHINE



MAKES IT SO EACH PROCESS ONLY SEES ITS OWN MEMORY.



- PROGRAM EXECUTION

ⓐ Assembly crash course

Registers	%rax, %rbx, ... %r8	
Constants (Immediates)	$42	$0x8
Pointer deref	(%r8)	(%rbp)
Pointer arith	-8(%rbp)	

address

*(r8) =

→ *(rbp - 8)

ⓑ Break for stack

- STACK FRAME
  - For each function
- Contains
  - Where to return.
  - Local variables
  - ...

%rbp: Base pointer
Beginning of running function's stack frame

%rsp: Stack pointer.
Last thing in the stack

© Back to assembly

movq src, dst : 'move' 64-bits from src to dst

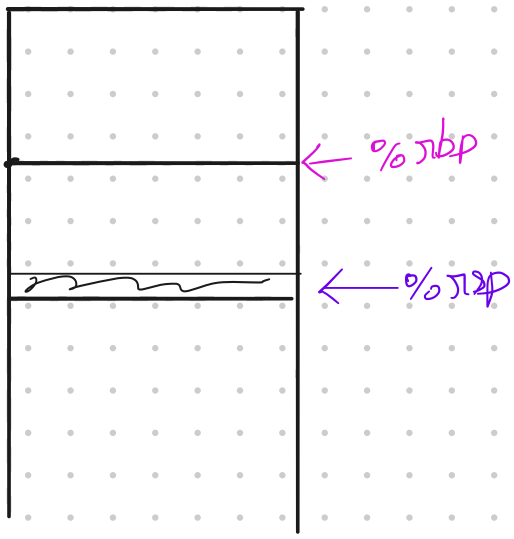
src can be register, memory or immediate

dst can be register or memory.

addq/subq op1, op2 op2 = op2 ± op1

pushq %rax : Push rax to stack

```
subq $8, %rsp
movq %rax, (%rsp)
```



popq %rax : Pop rax from stack

↳ Do the opposite of above

```
movq (%rsp), %rax
addq $8, %rsp
```

⑦ Calling & returning

call f

Note: In reality f ≡ f's address.

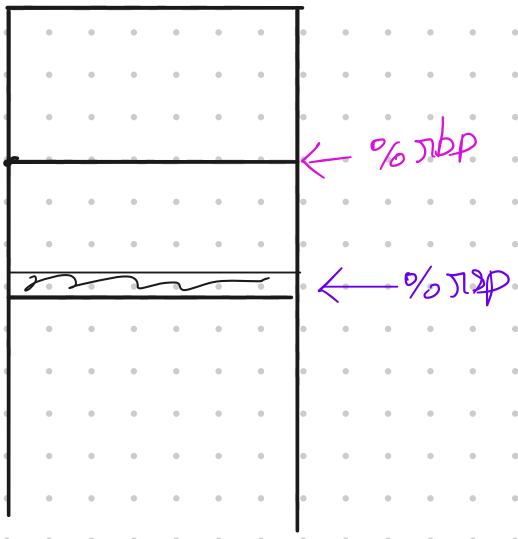
↳ Transfer control to f

≡

```
pushq %rip
movq  f, %rip
```

IMPORTANT: THIS IS PSEUDOCODE! YOU CANNOT RUN THESE INSTRUCTONS.

f():



ret ← Return to the function that called us

≡ popq %rip

OBSERVE: PROGRAM'S RESPONSIBILITY TO

@ ENSURE %RSP POINTS TO THE RIGHT ADDRESS WHEN RET IS CALLED

- CREATE STACK FRAME ON ENTRY INTO FUNC (PROLOG)
- DESTROY STACK FRAME BEFORE RET WHEN EXITING FUNC (EPILOG)

Calling convention

b) Figure out how to pass arguments

c) Manage registers across function calls.

Let us see an example

```
15          movq    $0, -8(%rbp)
16          movq    $8, -16(%rbp)
17
18          leaq    -16(%rbp), %rdi
19
20
21
22          call    f
23
24          movq    %rax, -8(%rbp)
```

0x4400

%rdi, %rsp

0x5000 | rsp

%rbp

STACK

MEM

main: x

main: arg

0x4400

old rbp

%rip

= 0x5000

text

f

0x5000

f Prolog

```
28      f:
29          pushq   %rbp
30          movq    %rsp, %rbp
31
32          subq    $32, %rsp
33          movq    %rdi, -24(%rbp)
34
35          movq    $0, -8(%rbp)
```

f Epilog

Return Value

```
47          movq    %r10, %rax
48
49          movq    %rpb, %rsp
50          popq    %rbp
51          ret
```

The actual work in f

```
37          movq    -24(%rbp), %r8
38          movq    (%r8), %r9
39          movq    %r9, %rdi
40
41
```

```
42          call    g
43
```

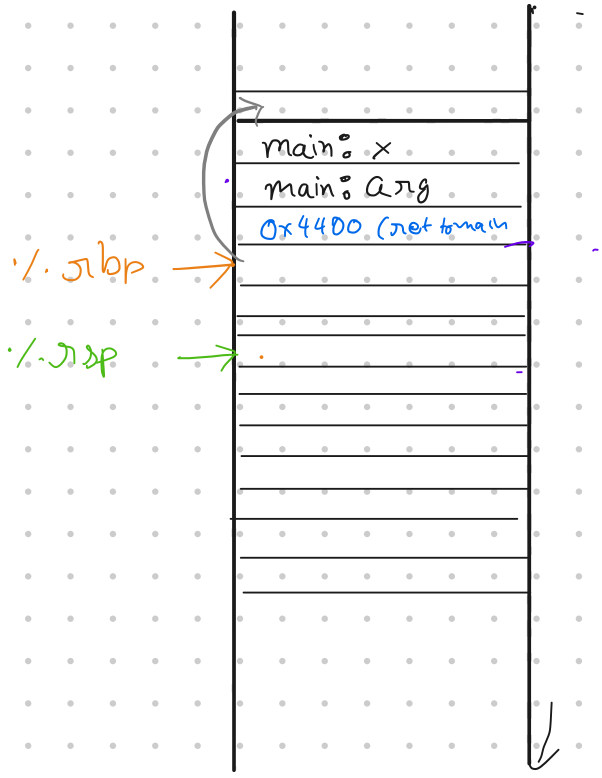
g has its own prolog & epilog

f PROLOG

```
28      f:
29          pushq   %rbp
30          movq    %rsp, %rbp
31
32          subq    $32, %rsp
33          movq    %rdi, -24(%rbp)
34
35          movq    $0, -8(%rbp)
```

g Prolog

```
53      g:
54          pushq   %rbp
55          movq    %rsp, %rbp
56          subq    $0x8, %rsp
57
```



g Epilog

```
59
60          movq    %rbp, %rsp
61          popq    %rbp
62          ret
63
```

f EPILOG

```
47          movq    %r10, %rax
48
49          movq    %rpb, %rsp
50          popq    %rbp
51          ret
```

Calling Convention

- Arguments (in order)

  %rdi, %rsi, %rdx, %rcx, %r8, %r9

- Return

  %rax

- Call preserved (callee save)

%rbx, %rbp, %r12 - %r15

Function must restore value before returning

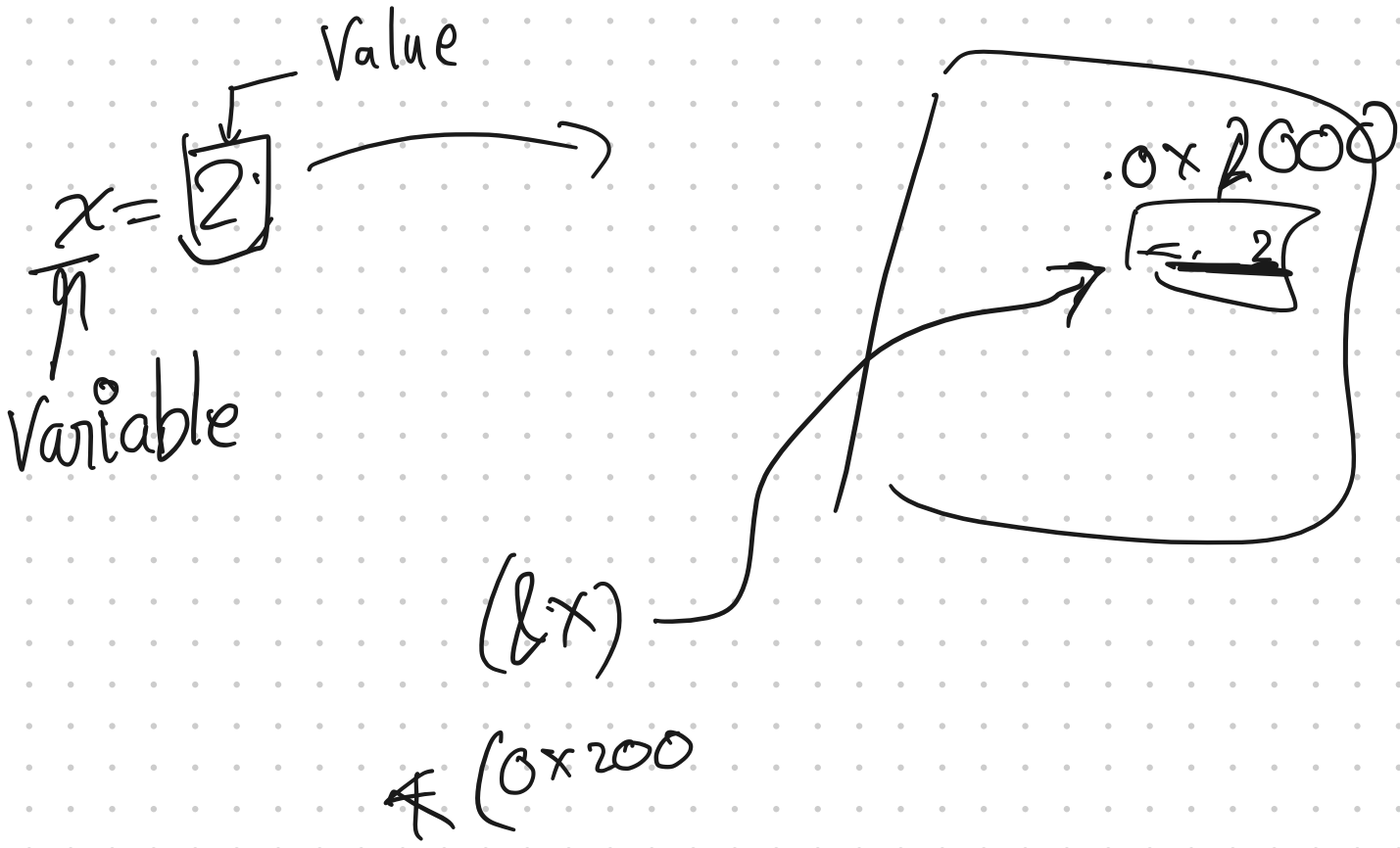
- Call clobbered (Caller save)

Everything else.

```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;
}
```



```
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
```


Value
x = 2
Variable
0x2000
2
(&x)
0x200