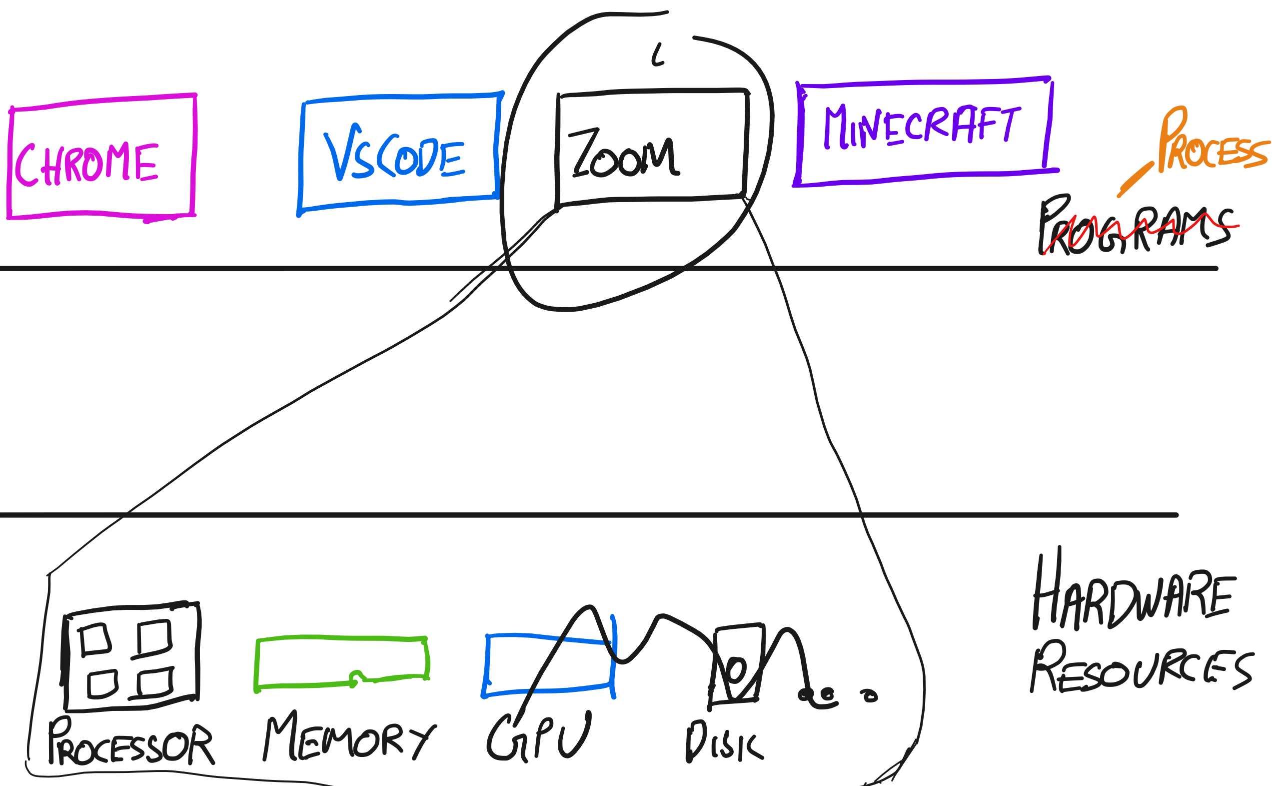
CS202-002: Operating Systems

Two Days Ago... Ended With History

Today

- Processes
- Process Execution
  - ↳ Stack
  - → Function Calls — Stack Frames
  - → System Calls
- Other Traps

Maybe went through some of this in CSO.
- Review it again
- Agree on terminology
- Might add details



Only Considering Processor + Memory

Two Questions

Two Questions

Q1. (a) What Does The Processor Look Like To A Process?

(b) What Does Memory Look Like To A Process?

→ Today!

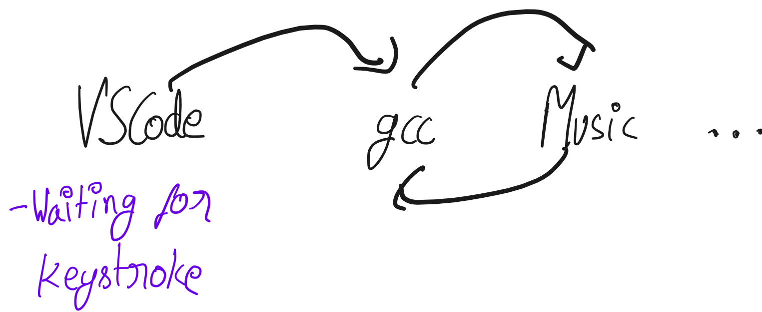
Q2. How Does The Kernel Implement & Manage Processes?

→ Leave This For Another Day

But Why?

- Convenience

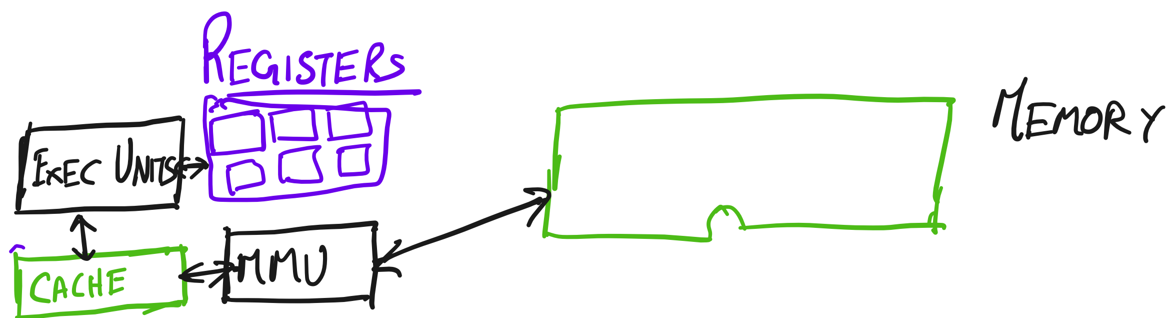
- Efficiency



- Combining functionality

Q1. (a) WHAT DOES THE PROCESSOR LOOK LIKE TO A PROCESS?
(b) WHAT DOES MEMORY LOOK LIKE TO A PROCESS?

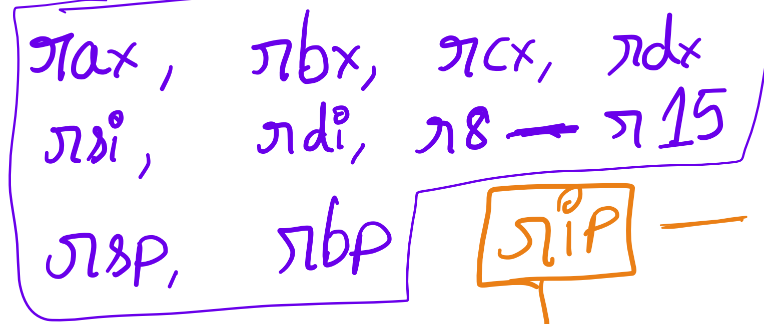
PROCESSOR ("CORE") & MEMORY: AN ABSTRACT VIEW



Exec units: Have no state.
Must read both instruction & data from elsewhere

Registers: A set (of known size, determined by processor designer) of fixed-size holders of state (in this class 64 bits).

x86-64



general purpose

special purpose

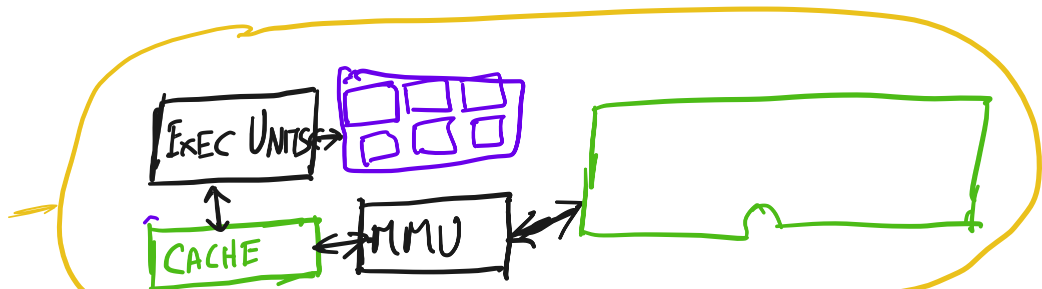
This one is **special**

CONSISTENT + FAST ACCESS

MEMORY : MORE PLENTIFUL : AMOUNT DECIDED WHEN BUILDING A COMPUTER

BUT SLOWER ACCESS (O(ns — 100ns)) with variance.

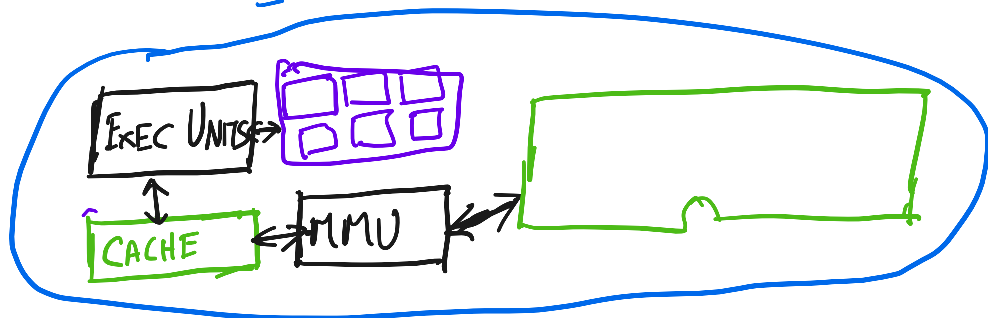
```
int p= fork();
 if (p!= 0) {
   printf("I'm yellow");
```



```
} else {
  printf("I'm blue");
}
```

I am yellow

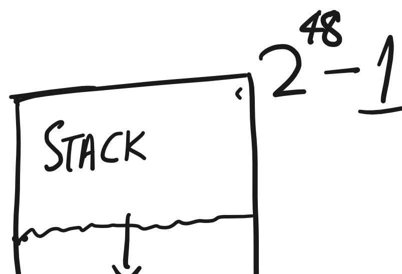
I'm blue

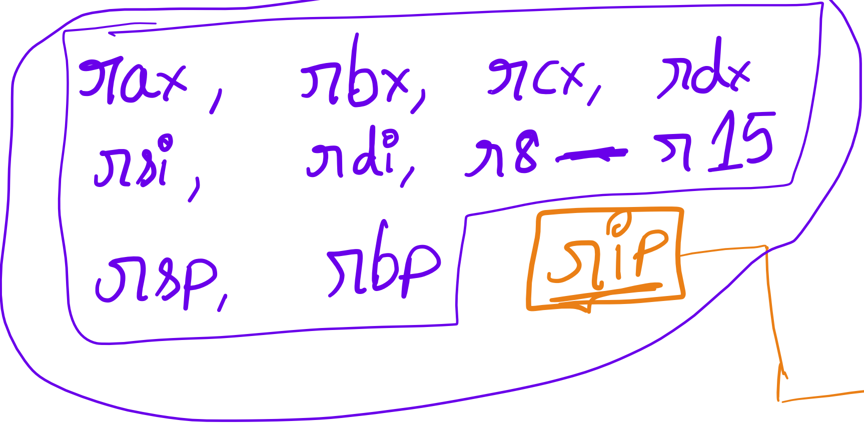


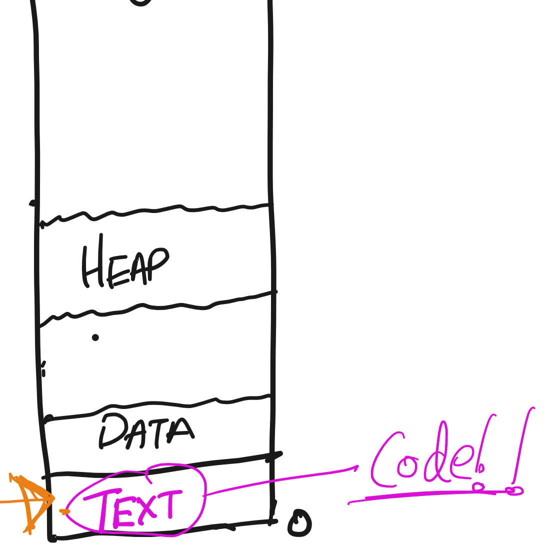
Each process gets

- Its own set of registers
- Its own view of memory
- Some other fiddly metadata.

PROCESS MEMORY

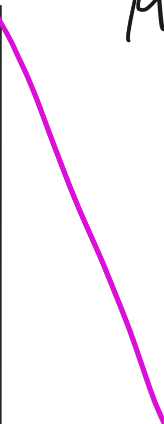






IMPORTANT: ALL INFORMATION USED BY THE PROCESSOR IS EITHER IN REGISTERS OR MEMORY

```
1   #include <stdio.h>
2   #include <stdint.h>
3
4   uint64_t f(uint64_t* ptr);
5   uint64_t* q;
6
7   int main(void)
8   {
9       uint64_t x = 0;
10      uint64_t arg = 8;
11
12      x = f(&arg);
13
14      printf("x: %lu\n", x);
15      printf("dereference q: %lu\n", *q);
16
17      return 0;
```



```
18  }
19
20  uint64_t f(uint64_t* ptr)
21  {
22      uint64_t x = 0;
23      x = g(*ptr);
24      return x + 1;
25  }
26
27  uint64_t g(uint64_t a)
28  {
29      uint64_t x = 2*a;
30      q = &x;    // <-- THIS IS AN ERROR (AKA BUG)
31      return x;
32  }
```

Somewhat More Explicit In Assembly

```
1       main:
2           pushq   %rbp
3           movq    %rsp, %rbp
4
5           subq    $16, %rsp
6
7           movq    $0, -8(%rbp)
8           movq    $8, -16(%rbp)
9
10          leaq    -16(%rbp), %rdi
11
12
13
14          call    f
15
16          movq    %rax, -8(%rbp)
17
18          # eliding the rest of main()
```

rsp, foo, Old RBP @ Line 2, x = 0, arg = 8, rip@16, Top of Stk, rsp, RBP, rip = 14, rip, rdi = foo, rax, Base Pointer, STACK!!

Simple assembly

%rax ← Registers written with %

$22 ← Numbers (immediates) written with $

(%rbp) ← Pointer arithmetic

$-8(\%rbp)$ ...

$\%rbp - 8$

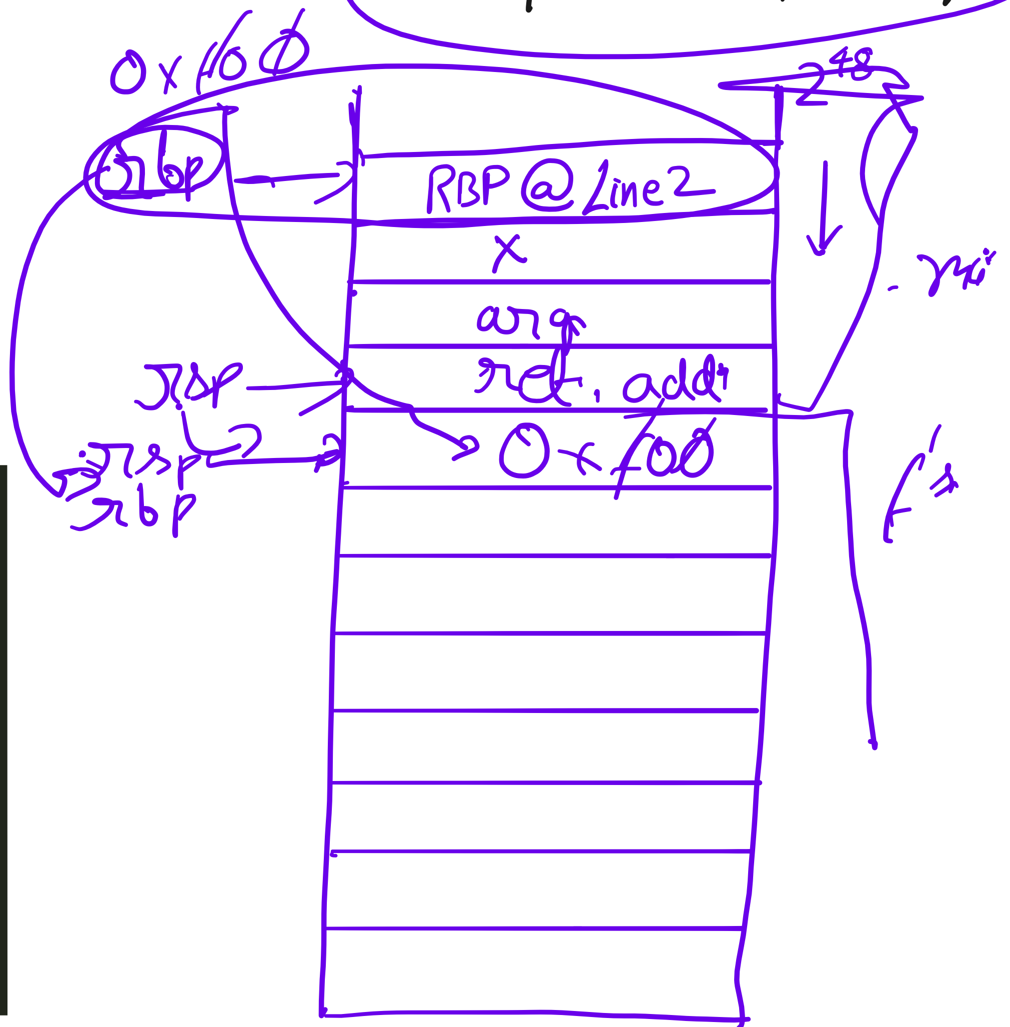
movq source, destination

subq x, y ⇒ y = y − x

pushq %rax ≡ %rsp = %rsp − 8
movq %rax, %rsp

```
1   main:
2       pushq   %rbp
3       movq    %rsp, %rbp
4
5       subq    $16, %rsp
6
7       movq    $0, -8(%rbp)
8       movq    $8, -16(%rbp)
9
10      leaq    -16(%rbp), %rdi
11
12
13
14      call    f
15
16      movq    %rax, -8(%rbp)
17
18      # eliding the rest of main()
```

rax
rdi
rbp
rsp




```
f:
    pushq   %rbp
    movq    %rsp, %rbp

    subq    $32, %rsp
    movq    %rdi, -24(%rbp)

    movq    $0, -8(%rbp)

    movq     -24(%rbp), %r8
    movq     (%r8), %r9
    movq     %r9, %rdi
```

"PROLOG"

```
movq    %rax, -8(%rbp)
movq    -8(%rbp), %r10
addq     $1, %r10
movq     %r10, %rax

movq     %rpb, %rsp
popq     %rbp
ret
```

"EPILOG"

Ok! Now let us understand the bug in g'

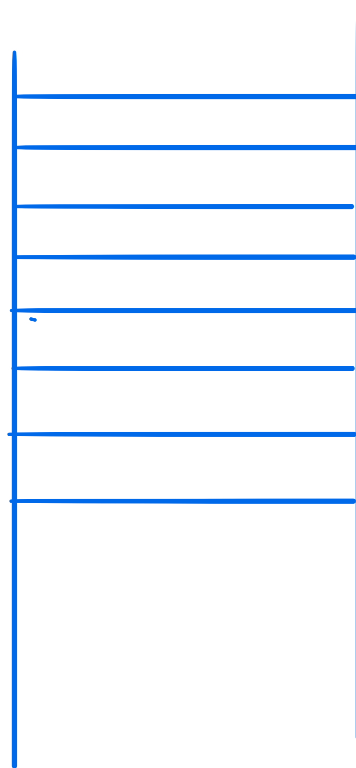
uint64_t* q;

```
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  }
```

...

x = g(*ptr);

...



Calling Convention

① How to pass arguments

② Where does the return value go

③ Who saves what registers?

Caller saved / volatile

%rax, rcx, rdx, r8, r9, r10, r11

Callee saved

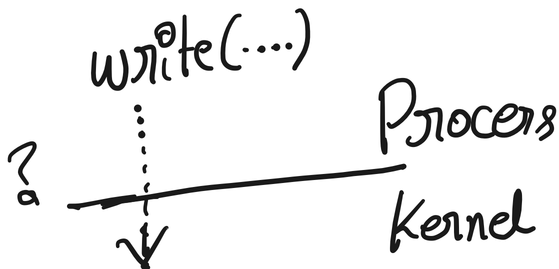
rbx, rbp, rdi, rsi, rsp, r12–15

```
movq    %rax, -8(%rbp)
movq    -8(%rbp), %r10
addq     $1, %r10
movq     %r10, %rax

movq     %rpb, %rsp
popq     %rbp
ret
```

f epilog

Syscalls



```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
```