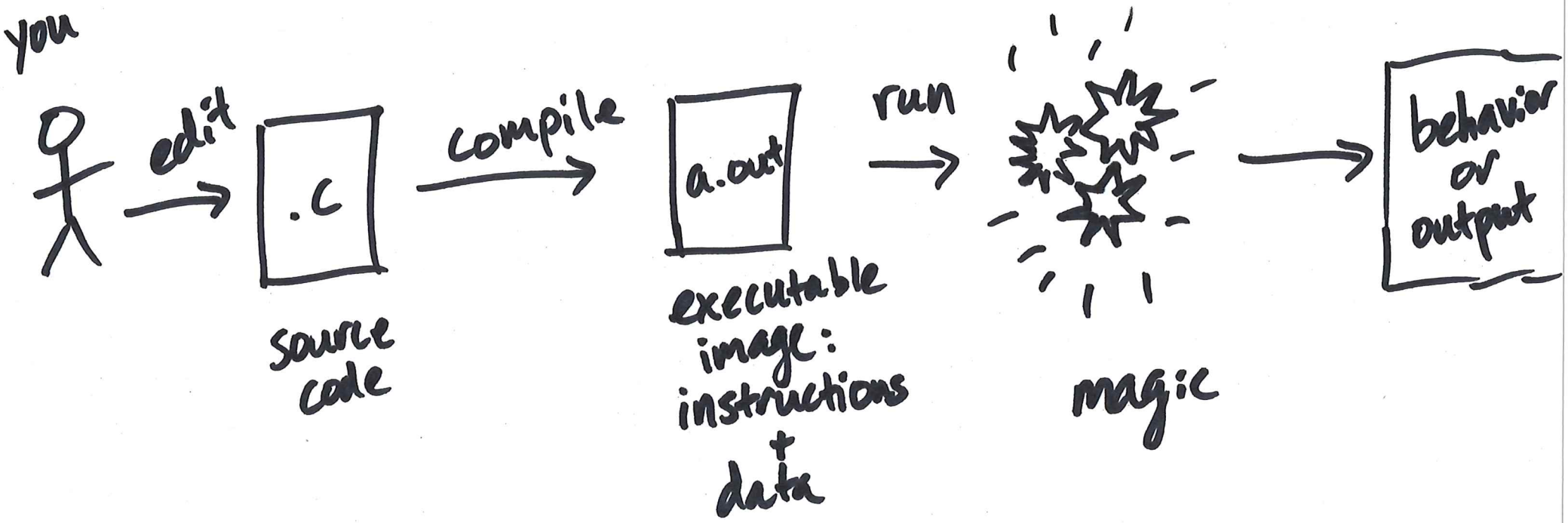
Lec 2

Kernel Mode + the Process Abstraction

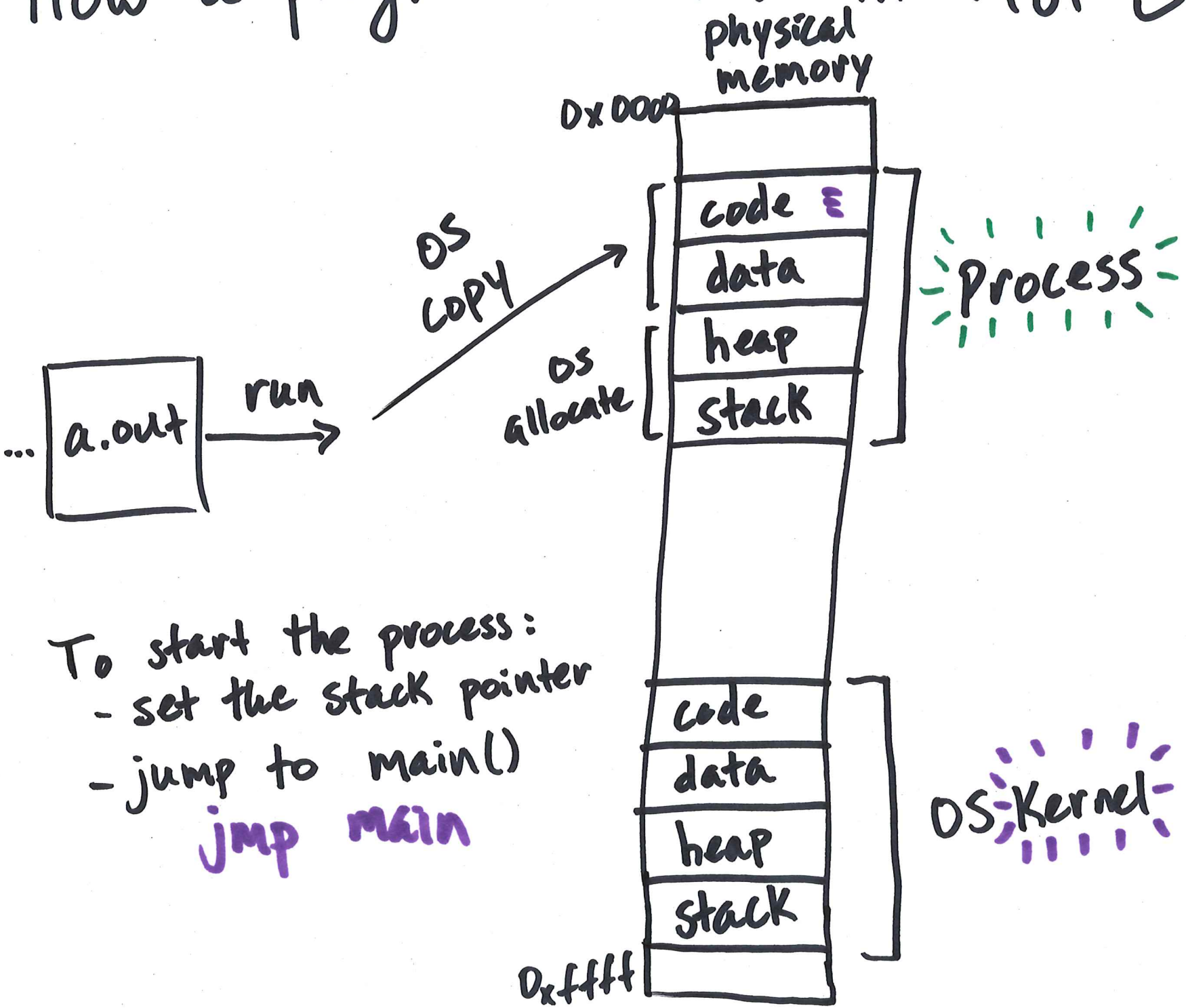
How to program? pre-451



~~gcc -o name~~ prog.c

./a.out

How a program runs? in 1981 :



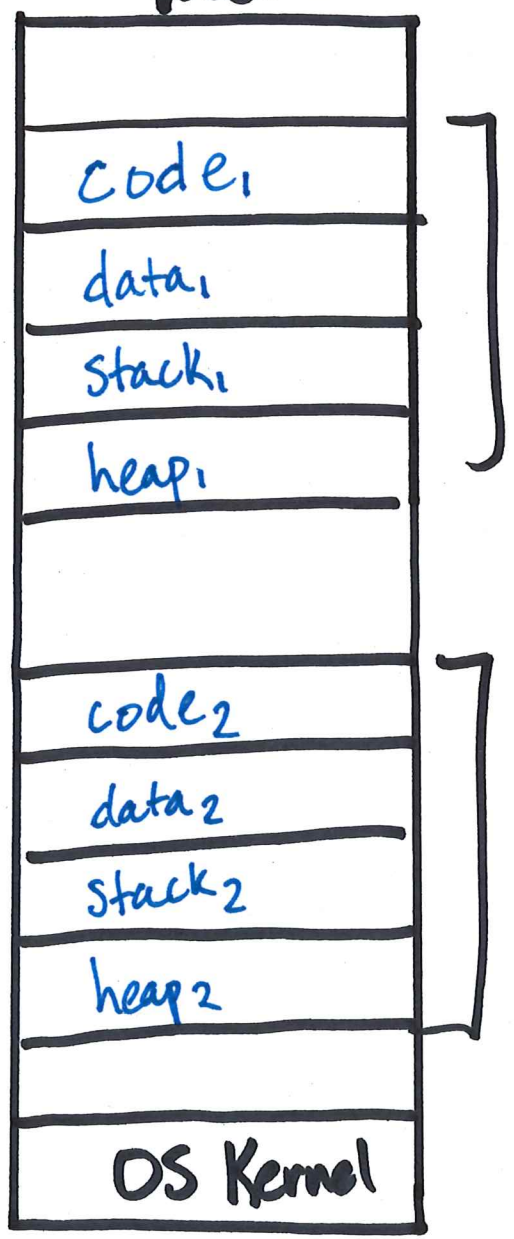
To start the process:
- set the stack pointer
- jump to main()

```
jmp main
```

Running multiple copies of 1 program (still 1981)

each copy needs:

- own stack
- own copy of any mutable data
  - usually all data + heap
  - maybe share code if not self-modify



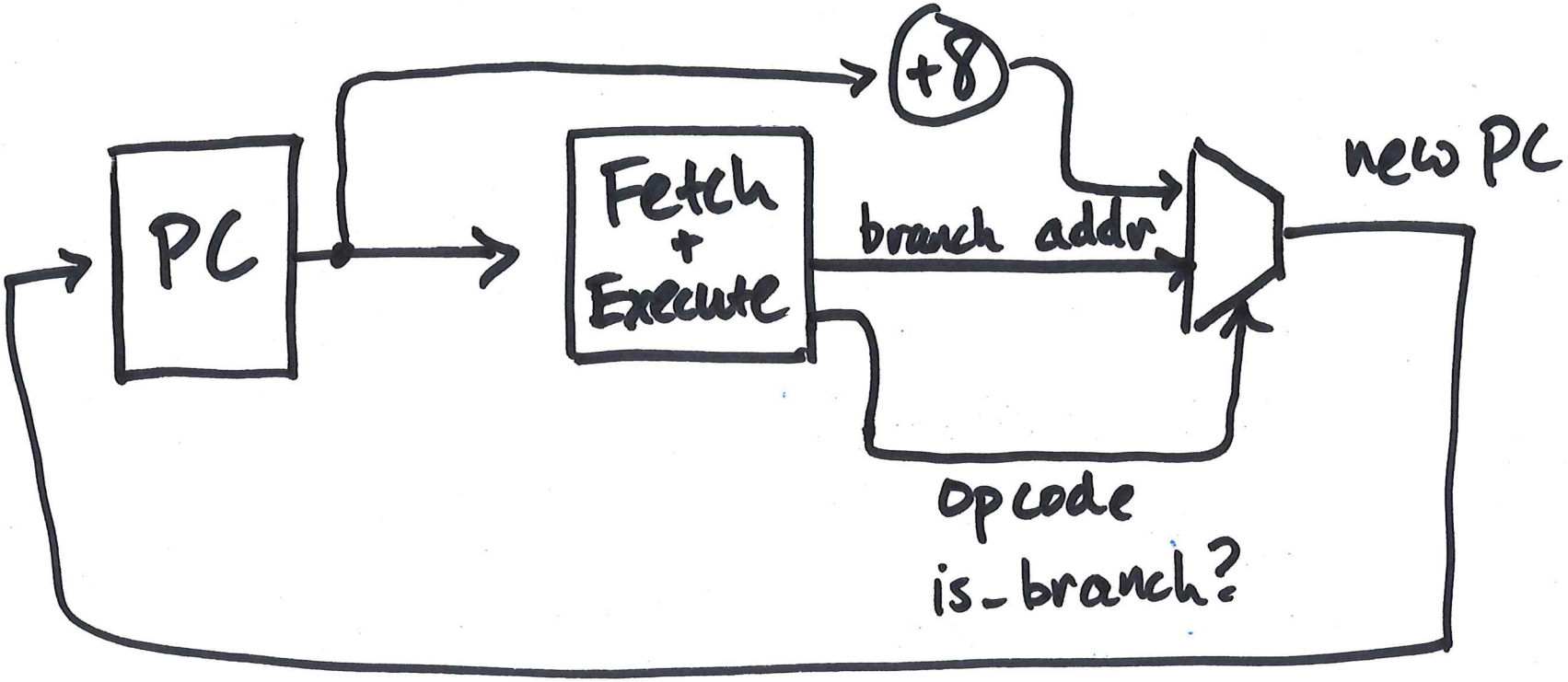
A process is an instance of a program

- each instance has its own copy of the program in memory

(for now we assume each process is single threaded.)

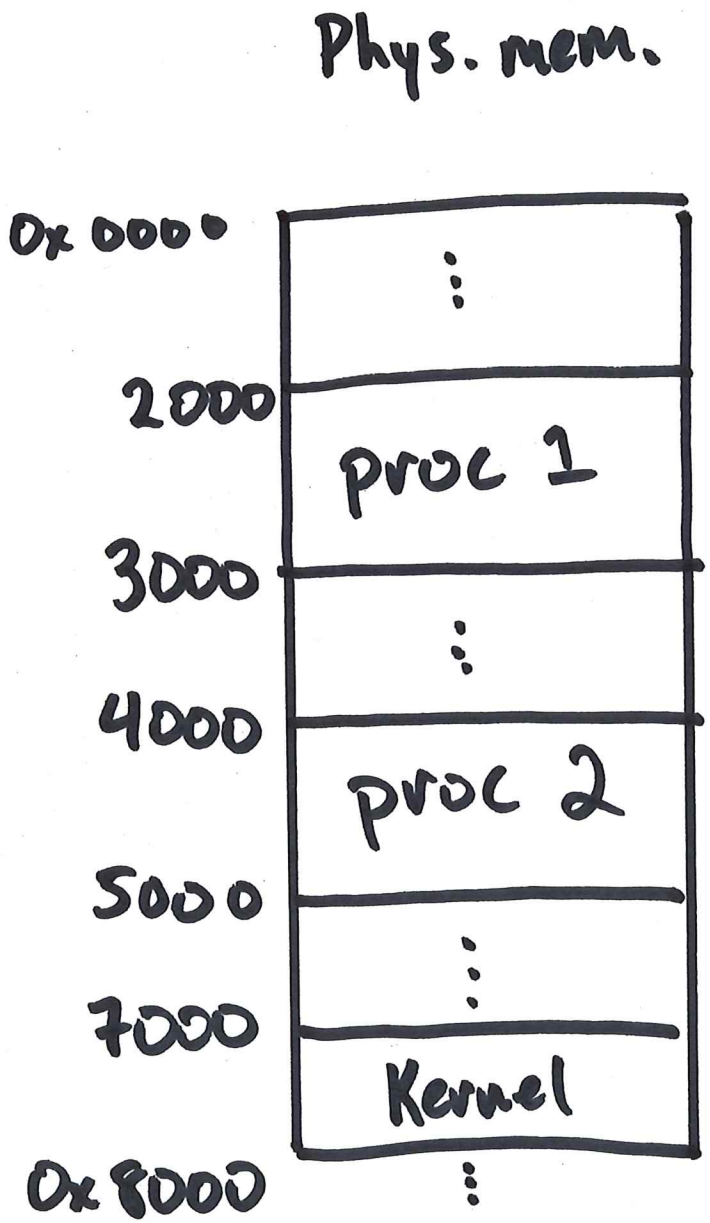
- threads start in ~10 days

After jumping to main() ...




```
while True:
    instr = fetch(pc)
    is_branch, branch_addr = execute(instr)
    if is_branch:
        pc = branch_addr
    else:
        pc += 8
```

What could go wrong?



proc 1 code

```
    ⋮
char  *p = _____
*p = 0;
    ⋮
```

2500
3500
4500
7500

?

Your mental model of a CPU:

"does what we tell it"

Pretty much accurate in 1981!

but

Unreliable + Insecure

Need some way to restrict privileges of running process

- can only access its own memory
- cannot perform "dangerous" IO
- cannot hog resources

Straw Proposal: Simulate

- instead of executing on Hardware, simulate CPU in software
- can insert any checks we want!

Better: implement checks in hardware

dual-mode operation

- CPU keeps track of one extra bit
  - in user mode, CPU checks each instr.
  - in Kernel mode, no checks

What checks?

Whatever it takes to achieve our goal:

malicious or buggy apps
cannot corrupt the OS
or other apps

At least:

- prevent app from executing privileged instructions
- memory protection
- timer interrupts

Privileged Instructions

- can only be executed in Kernel mode

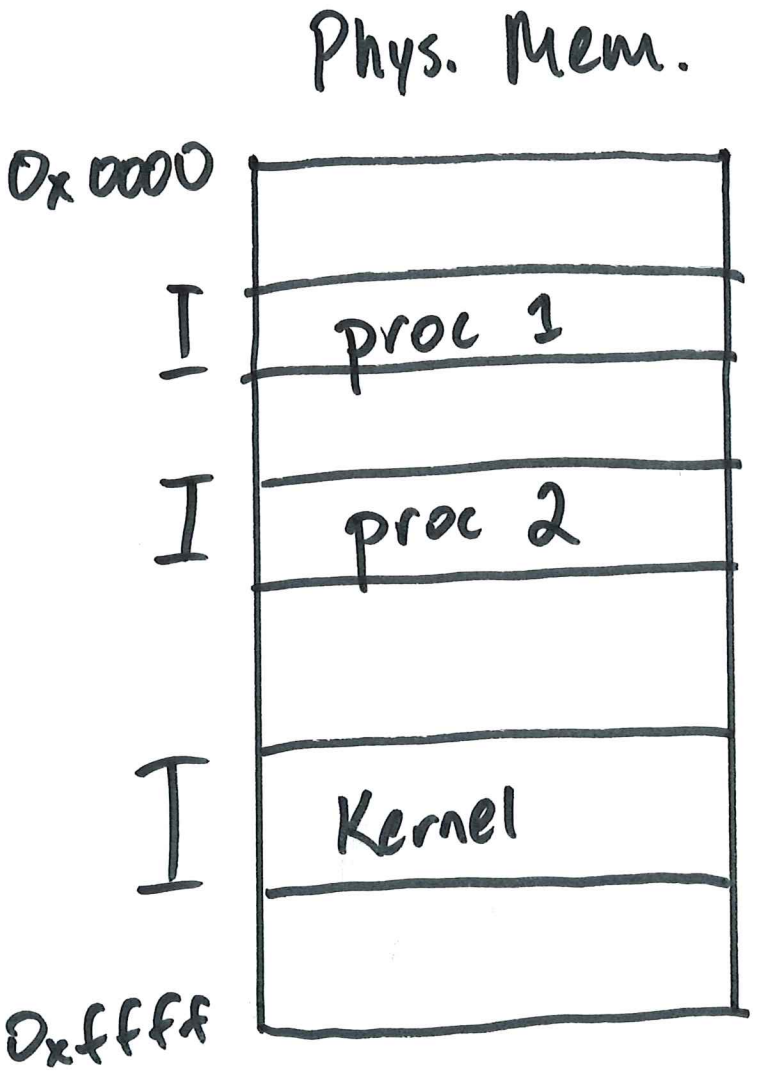
- examples:
  - any instruction that changes the mode!
    (one exception: system calls)
  - instructions that configure what memory the app can access
  - instructions that configure interrupt handling

Again: these can be used by the OS

What if app tries a privileged instr?

- OK: CPU treats as nop
- better: processor exception
  ↳ transfers control to exception handler in kernel

Memory Protection



- multiple procs + Kernel all in memory at same time
- must prevent apps from accessing Kernel memory (and other apps)

first idea: assign each proc a contiguous range of addresses it can access

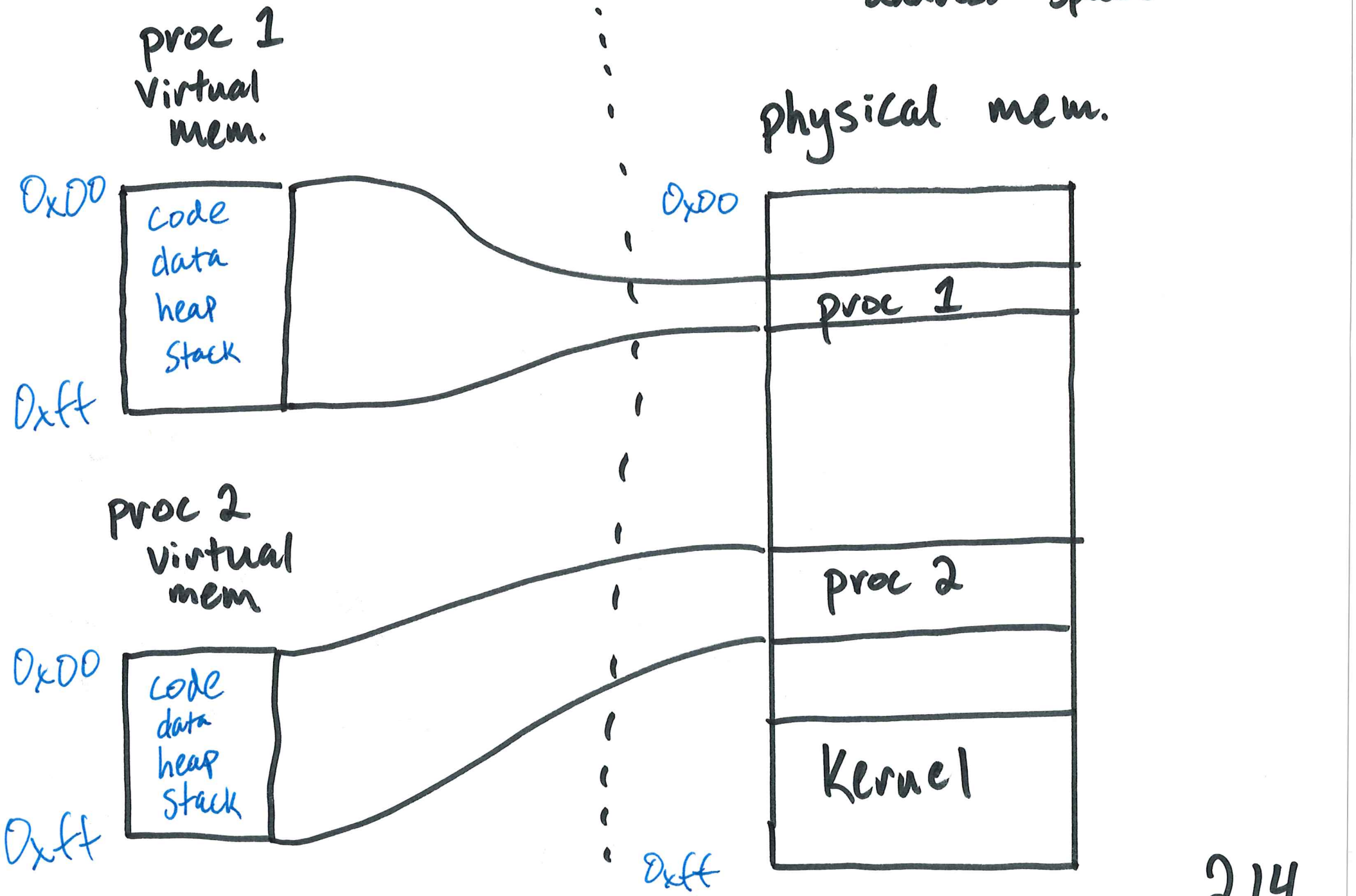
- hardware checks every memory access is in range of current proc
- OS runs w/o restriction

"base + bound"

2.13

Virtual Memory

idea: give each process separate address space



Virtual Memory (continued)

- OS sets up mapping for each process

  Virtual addr → physical addr

  - instructions for modifying this map are privileged!

- every memory access by the process is first translated by the hardware

Timer Interrupts

- OS must be able to "take back" the CPU
  - what if app enters infinite loop?
  - what if user asks us to stop the app?
- When the app is running, the OS is not!

↳ configure hardware timer to interrupt
  - transfers control back to OS
    - change mode to Kernel mode
    - call interrupt handler

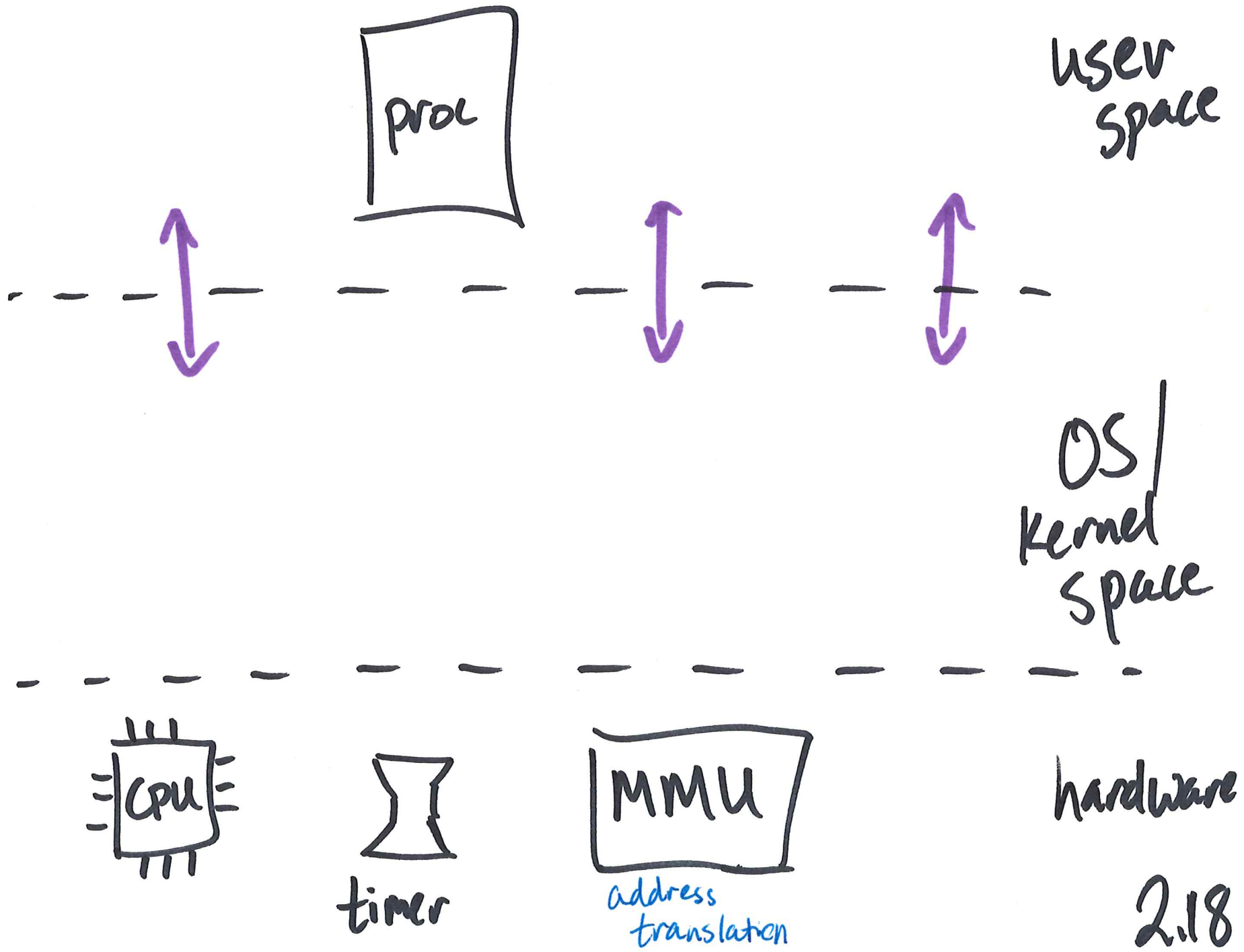
2.16

A process is an instance of a program that runs in a sand box

- cannot execute privileged instructions
- access to memory mediated by OS + HW
- must give up CPU when timer fires

2.17

Mode Transfer: when do we switch b/w user + kernel?



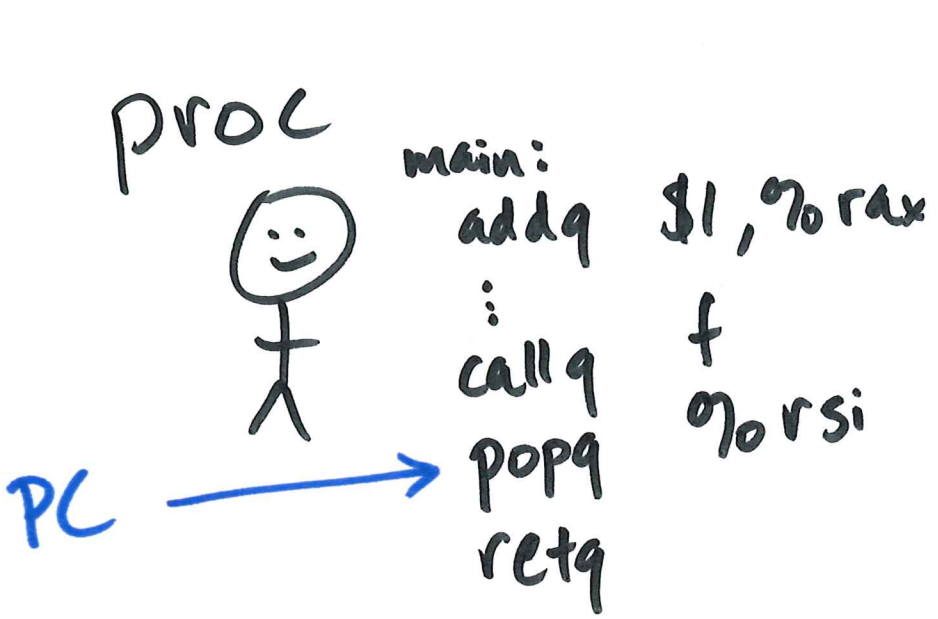
2.18

User → Kernel mode transfer

- interrupts
  asynchronous signal from hardware

- processor exceptions (synchronous)
  process tries to execute privileged instr,
  divide by zero, access unmapped addr...

- system calls (synchronous)
  process requests an operation from OS



thinks it has full control of CPU
- uses all registers
- controls the stack

suddenly, a timer interrupt occurs. now what?

- need to start running Kernel code
  - which needs registers + a stack
- later, want to resume the process