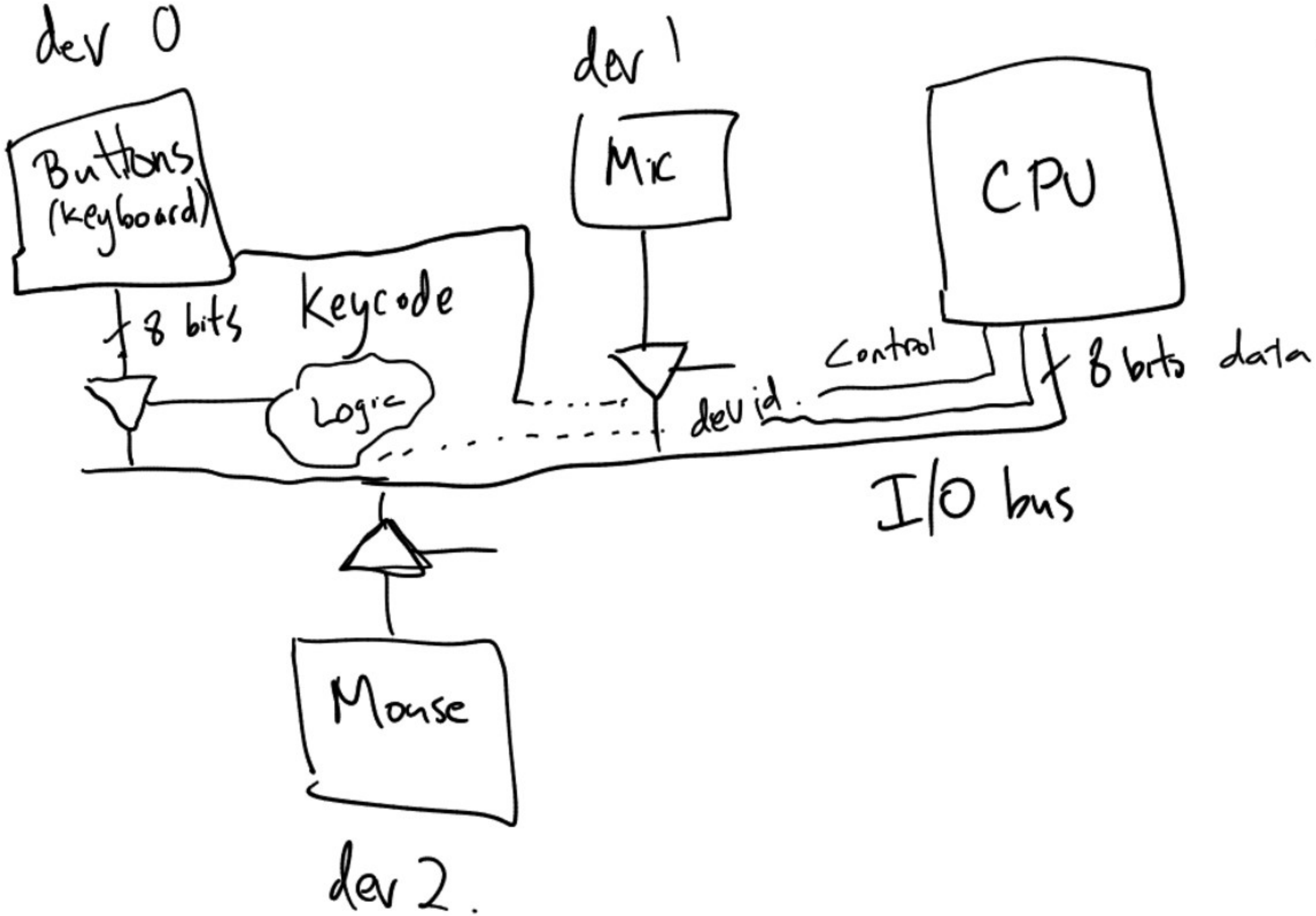
CS4410 Lecture 2: HW/SW interface. Devices

- Design of I/O bus
- Programming with I/O
- Multi-programming & Some history

- MMIO vs. Programmed I/O
- Polling vs. Interrupts
- DMA
- Device controllers
- Processes vs. Programs
- Process state.



to Communicate with devices (version 1)

- add new instructions

eg. • get data from kb (or device #) GET IO

• send data to device PUT IO

Hello World v1 (Programmed I/O, polling)

Message: "HELLO WORLD"

Main: PUTIO message screen_dev_id

```
r1 ← 0
while not r1:
    GETIO keyboard_dev_id → r1
```

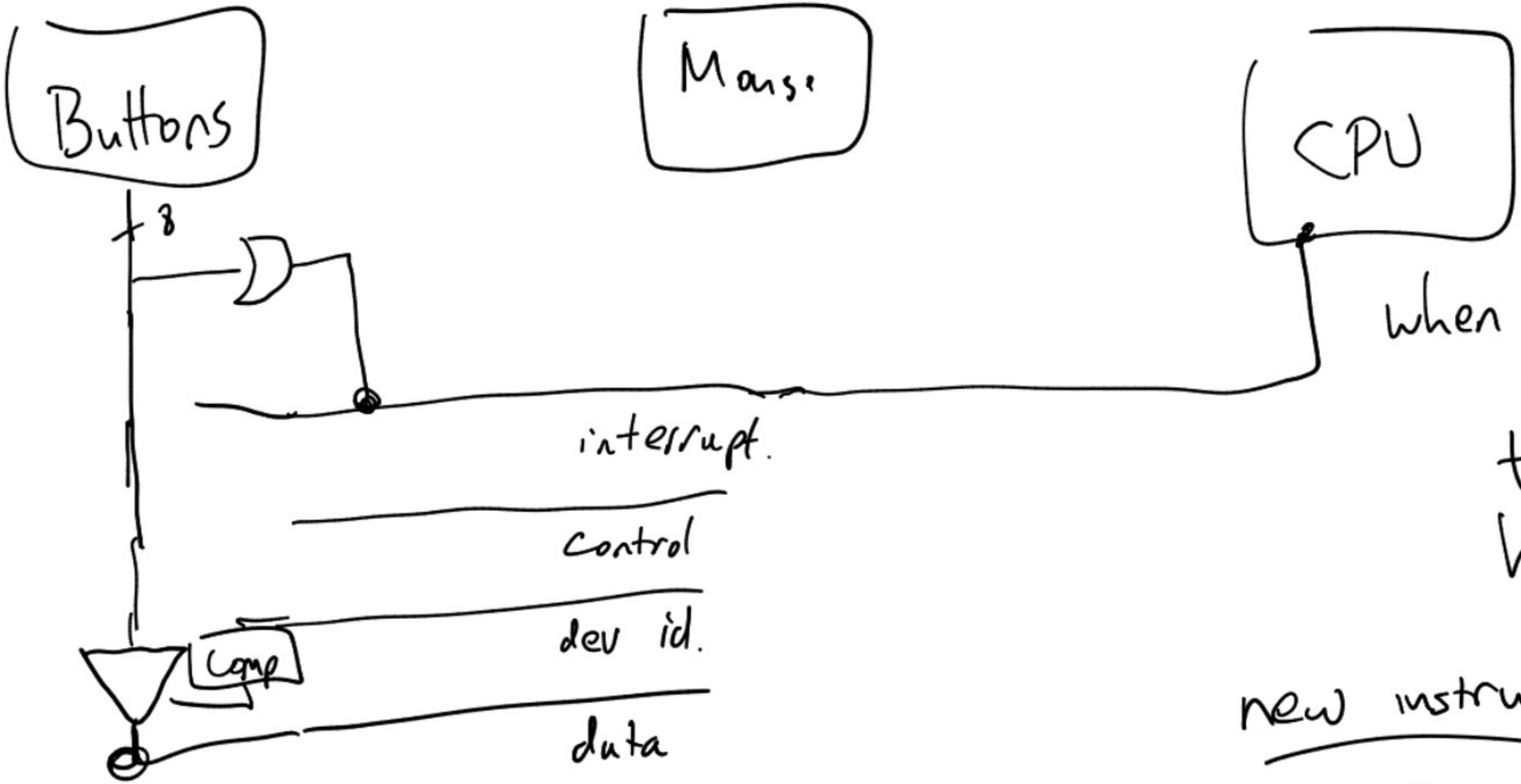
Polling:

- bad, because CPU can't do useful work while waiting
- might miss input

// key press in r1

⋮

Interrupts



when interrupt occurs, jump to code to handle interrupt.

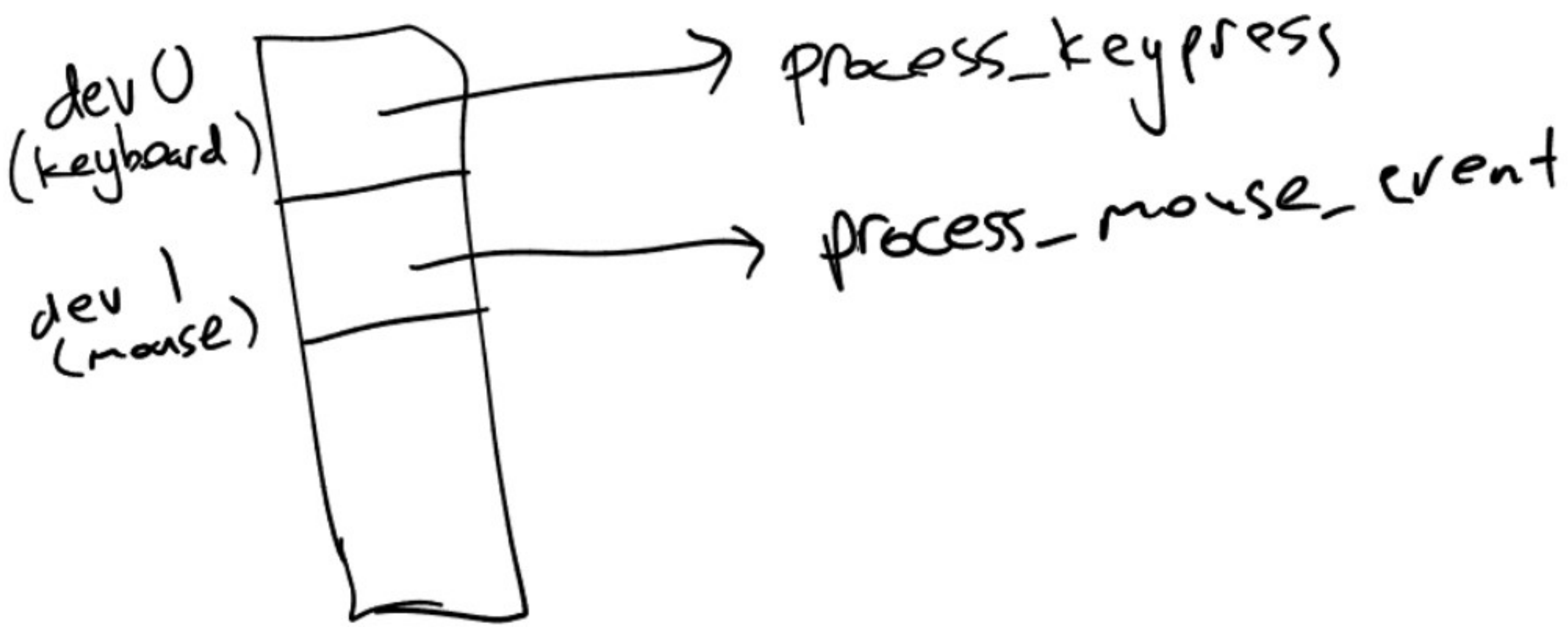
new instruction:

CONFIGURE: stores address of inter. handler code.

Interrupt vector:

- mapping of devices to addresses of I.H. code.

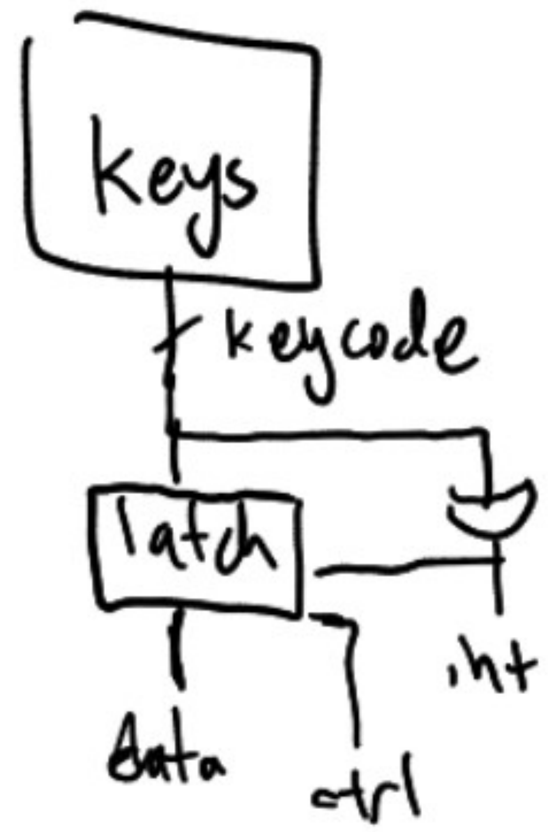
- "CONFIGURE <device id> <address of code>".
  of int. routine



- interrupt controller jumps to appropriate handler on interrupt.

Device Controller

- Keyboard needs internal logic
  - store keypress until int. hdlr. can fetch
  - buffer keystrokes, remove "bounce"
  - convert coordinates to ASCII ...



- Hardware / logic inside device: device controller
  - manage dev. state
  - communicate w/ CPU, interpret commands
  - control device operation
  - can be arbitrarily complex! Modern GPUs, for example, are entire processors.

Hello World v2 (Programmed I/O, interrupts)

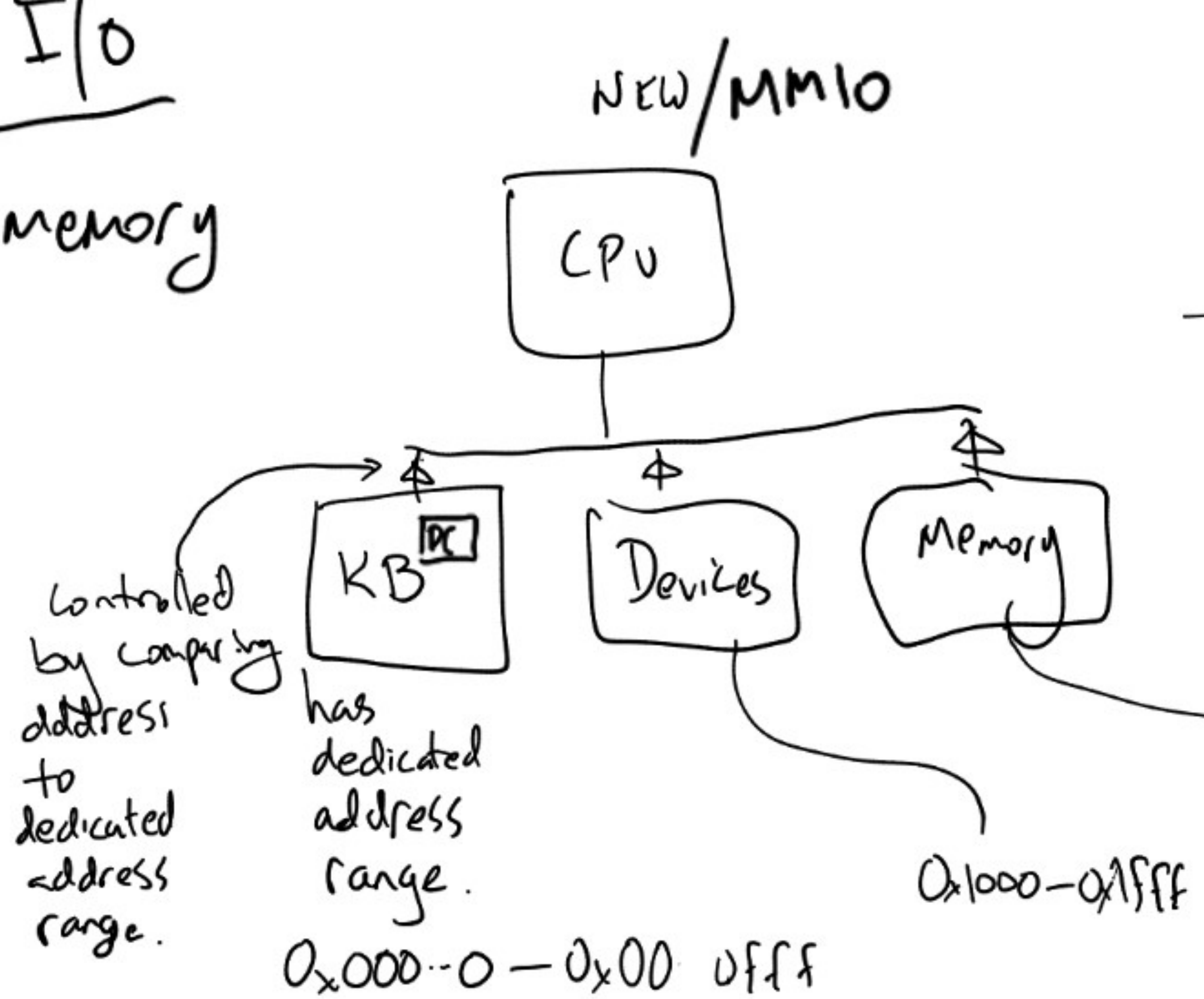
```
message: "HELLO WORLD"

main:    PUTIO message screen_dev_id
         SET_INT_HANDLER keyboard_id handler
         while true:
             pass          (or do other useful work)

handler: GETIO keyboard_id → r1
         // keycode in r1.
           ⋮
```

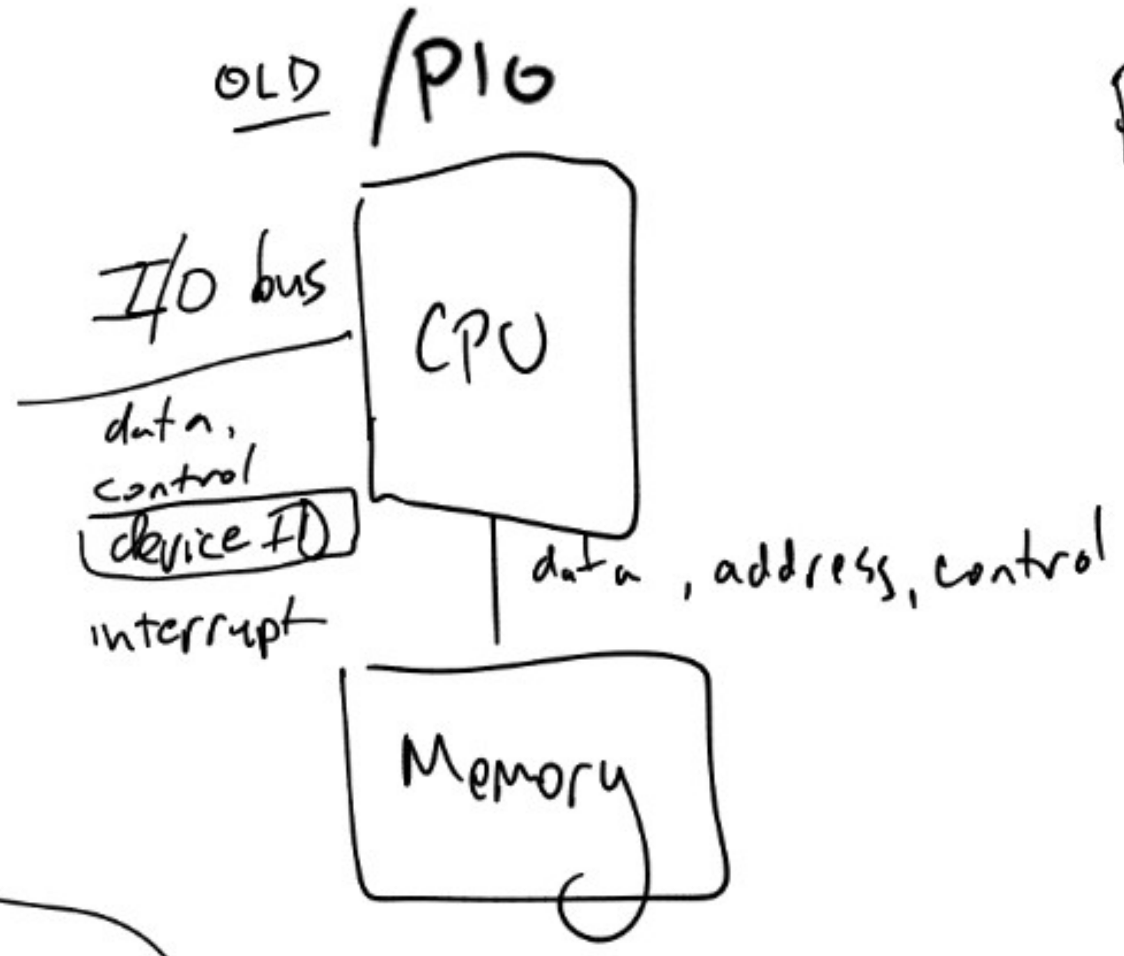
Dedicated I/O bus & instructions add complication

Memory-mapped I/O

- one bus for memory & I/O
- use normal load / store
- each device has addr. range.

+ Devices don't communicate w/ memory.

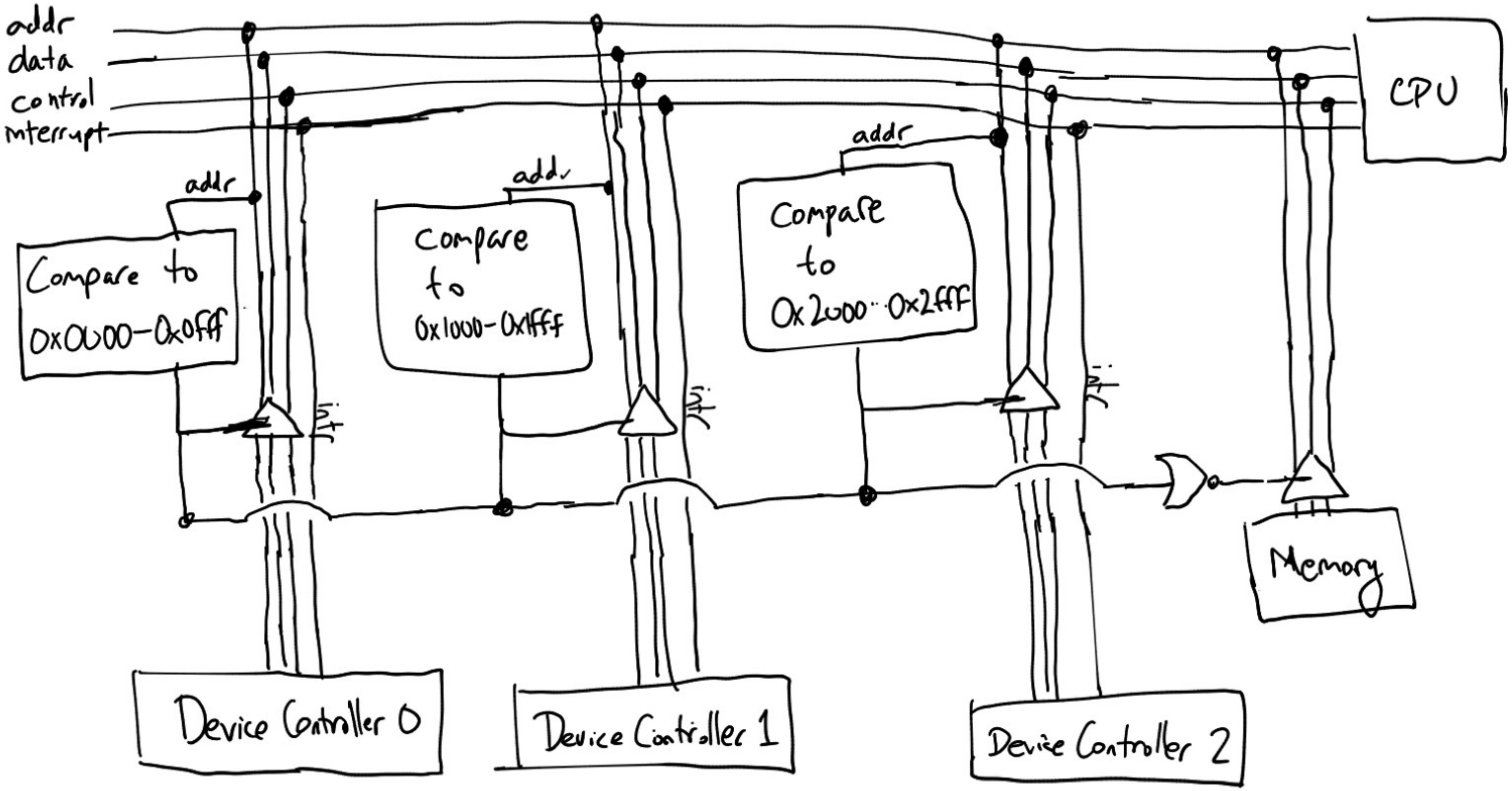




Programmed I/O:

- dedicated bus
- special instr.
- more complicated

MMIO



addr
data
control
interrupt
CPU
addr
Compare to
0x0000–0x0fff
addr
compare
to
0x1000–0x1fff
addr
Compare
to
0x2000‥0x2fff
intr
intr
intr
Memory
Device Controller 0
Device Controller 1
Device Controller 2

Hello World v3 (MMIO, interrupts)

message: "HELLO WORLD\0"

main:

print "HELLO WORLD"

```
r1 ← message
r3 ← screen_data_addr
do
    r2 ← *r1
    STORE r3 ← r2
    r1++
    r3++
until r2=0
STORE screen_control_addr ← screen_print_command
GOTO read
```

read:

```
SET_INT_HANDLER kb_int_id
                handler

STORE kb_control_addr ← kb_input_cmd
while true:
    pass
```

handler:

```
LOAD r1 ← kb_data_addr
// r1 has keystroke
:
```

blue underlined values are HW-specific constants

DMA (Direct Memory Access)

- Devices communicate with main memory
- Ex: Microphone or webcam or graphics card (lots of data)
- Use dedicated address ranges (as with MMIO) to control device
- Device sends / receives data from memory
- Raises interrupt to signal completion

control / data split common (eg networking)

Hello World v4 (MMIO, DMA, interrupts)

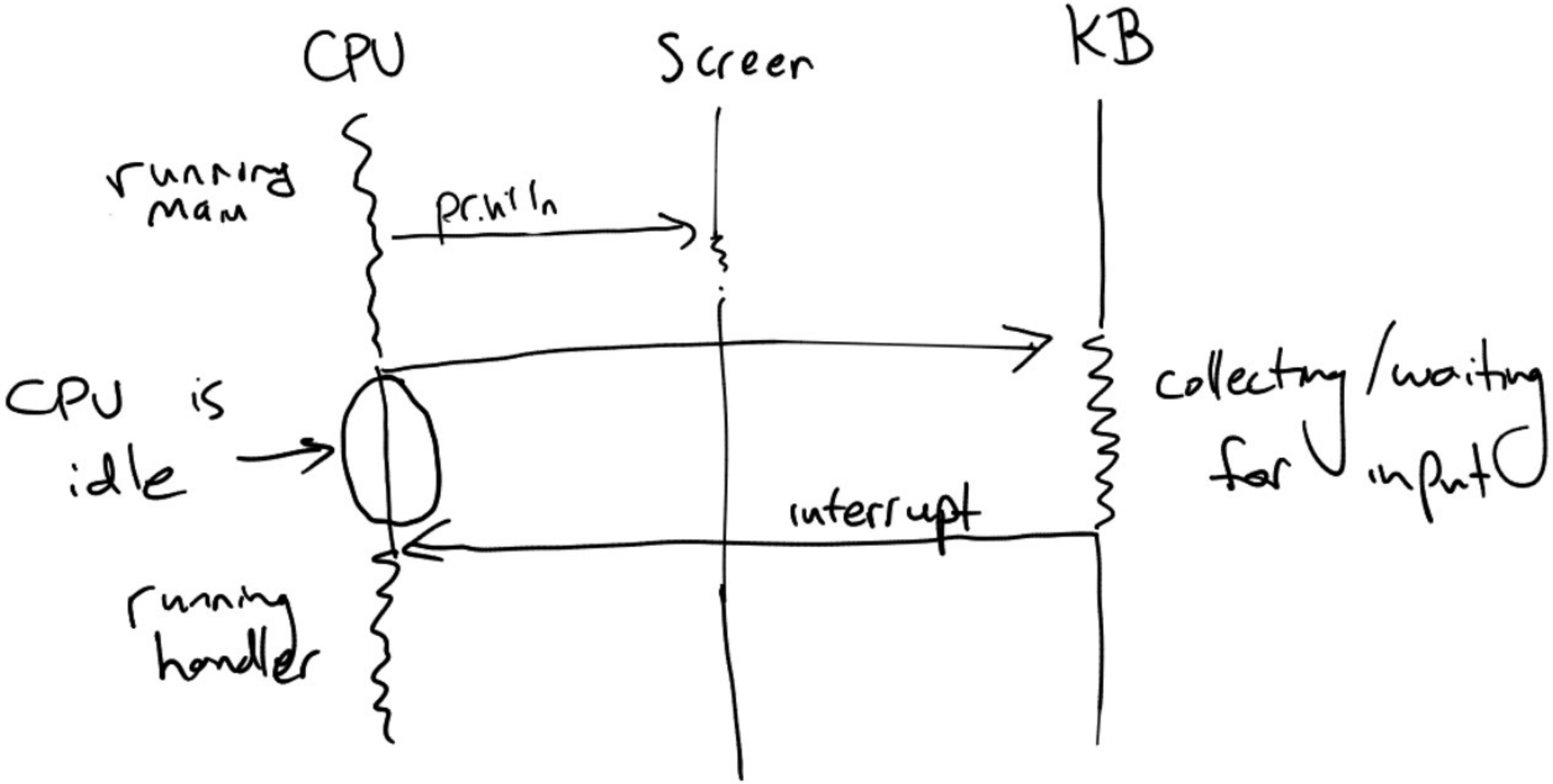
```
message: "Enter a line & press ENTER"
buffer: "\0\0 ....     \0"

main: STORE screen_data_addr ← message
      STORE screen_ctrl_addr ← screen_println_cmd
      STORE kb_output_ptr ← buffer
      SET_INTR_HANDLER kb_dev_id handler
      STORE kb_ctrl_addr ← kb_readline_cmd
      while true:
          pass
handler:
      LOAD r1 ← kb_status_addr
      // input line is in buffer
```

Screen only needs address of message, can read it itself.

KB can transfer a whole string into a buffer

Interrupt when done

Running Hello World v4



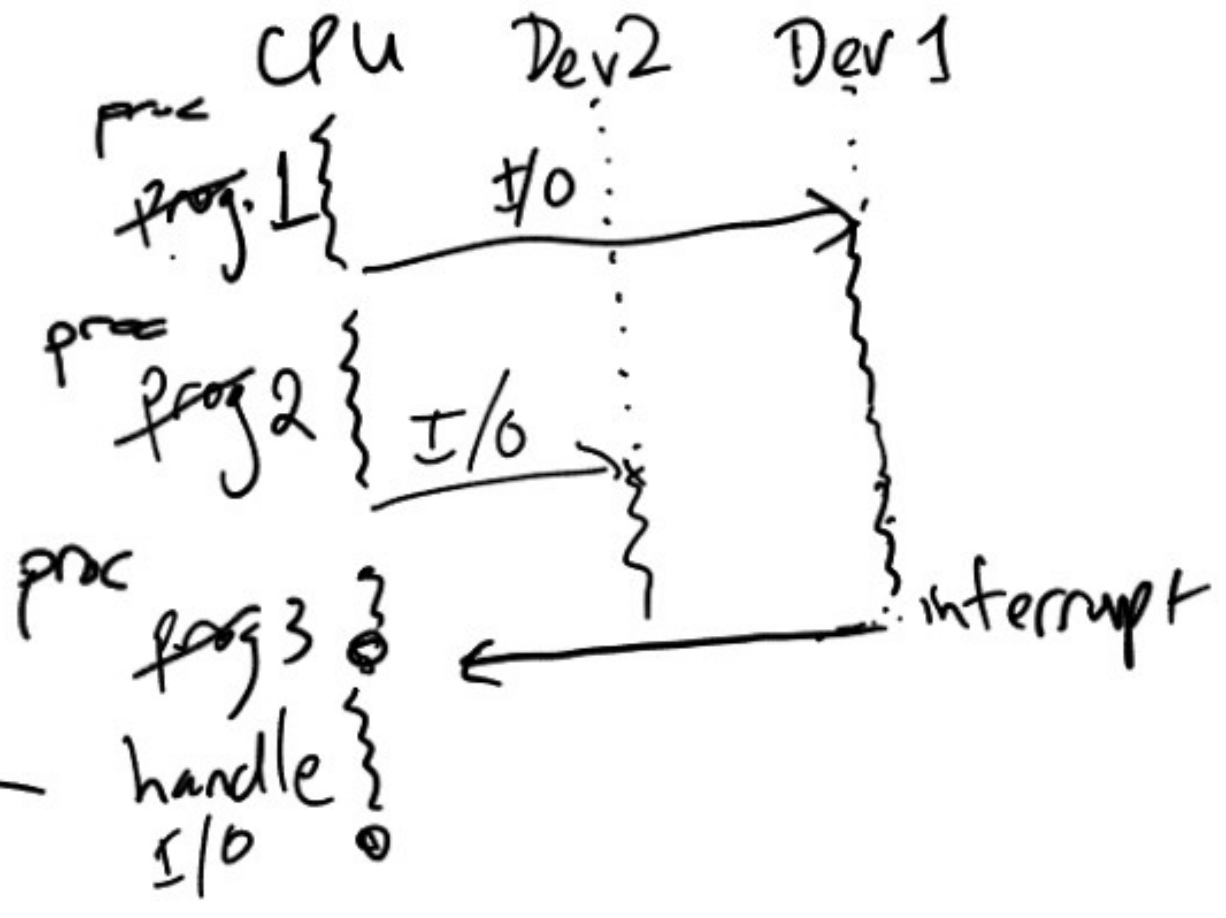
Multiprogramming

Store multiple ready to run programs.

process ~~prog.~~ 1

process ~~prog.~~ 2

process ~~prog.~~ 3



either
1. run proc 1 IH routine
or 2. store data to deliver to P1 later.

go back to prog 3 or to prog 1.

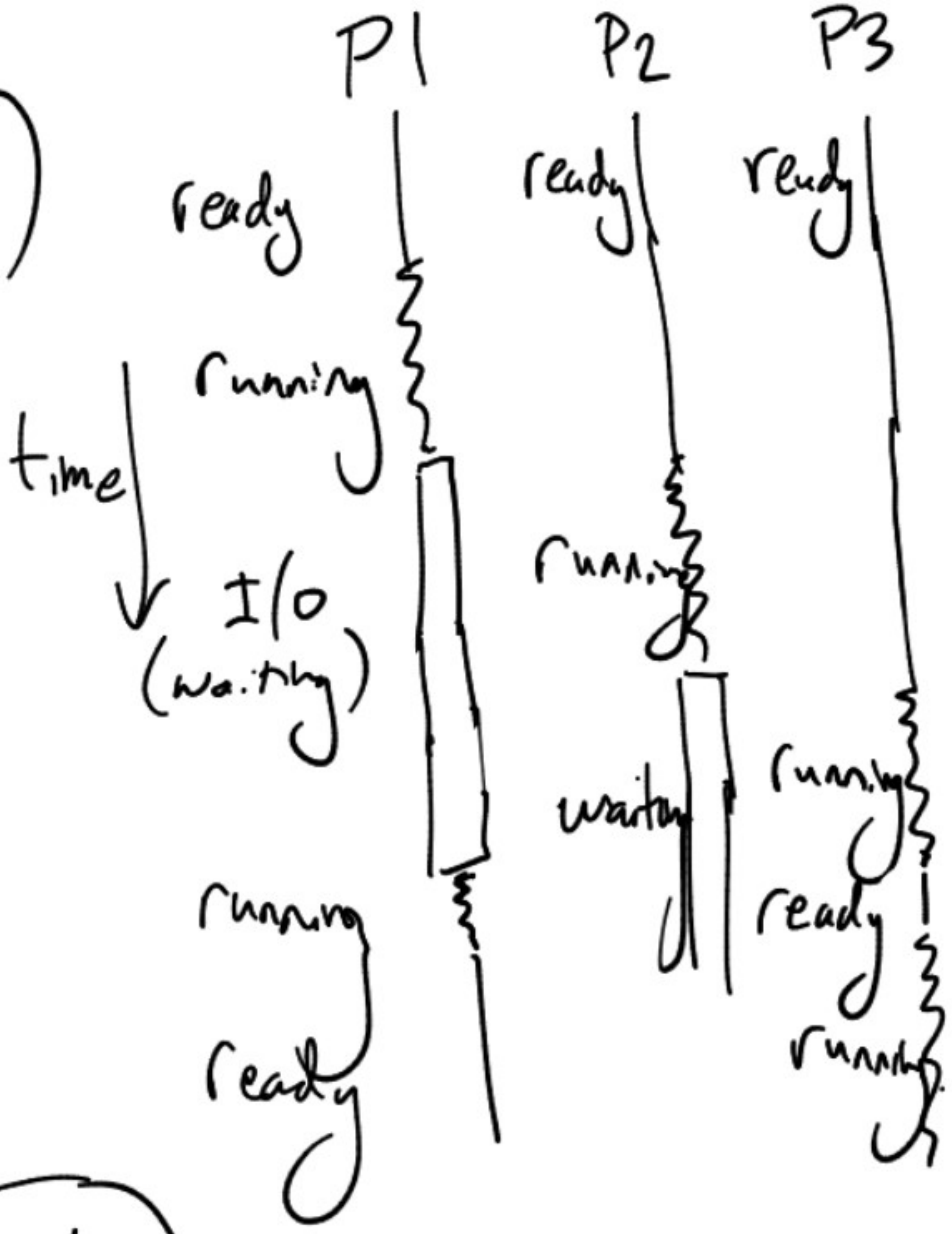
Program: Code, global constants.
i.e. stuff in executable file.

Process: program in motion
- program and
- local/global variables, registers, stack, heap ...

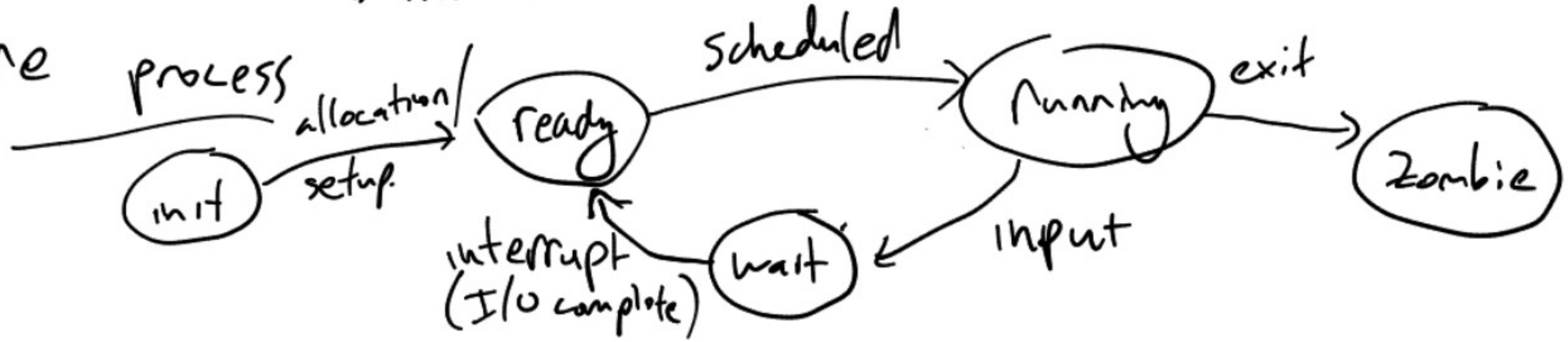
running a program creates a process

Each process has a state

- ready (can make progress)
- running
- waiting (for I/O to complete)
- Zombie (done, needs cleanup)
- initial



One process



Enter the Operating System

- make scheduling decisions
- changing state of CPU / memory
  to load / unload different processes.
- manage access to devices.
- isolating processes from one another.
  - some instructions are disallowed
    e.g. "update interrupt vector"