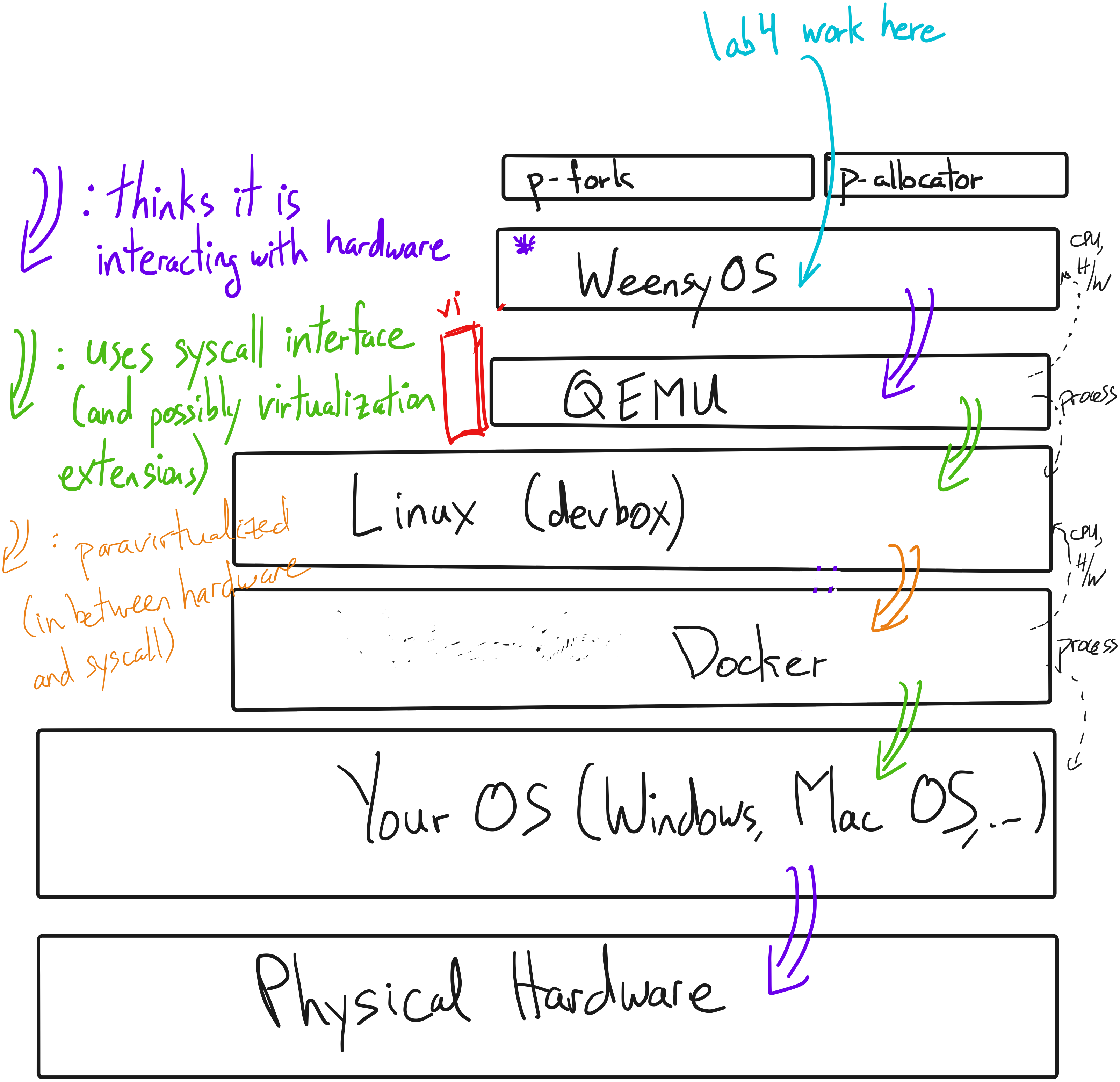
- [x] 1. Last time
- [x] 2. Virtual memory reinforcement
- [x] 3. WeensyOS walkthrough
- [ ] 4. WeensyOS context switches
- [ ] 5. gdb




Hints:

- processes: files matching p-*.c
- kernel code: files matching k-*.c , k-*.s
- system calls and returns

sys_page_alloc();

look in process.h

kernel returns: exception_return();

%rax contains return value of system call

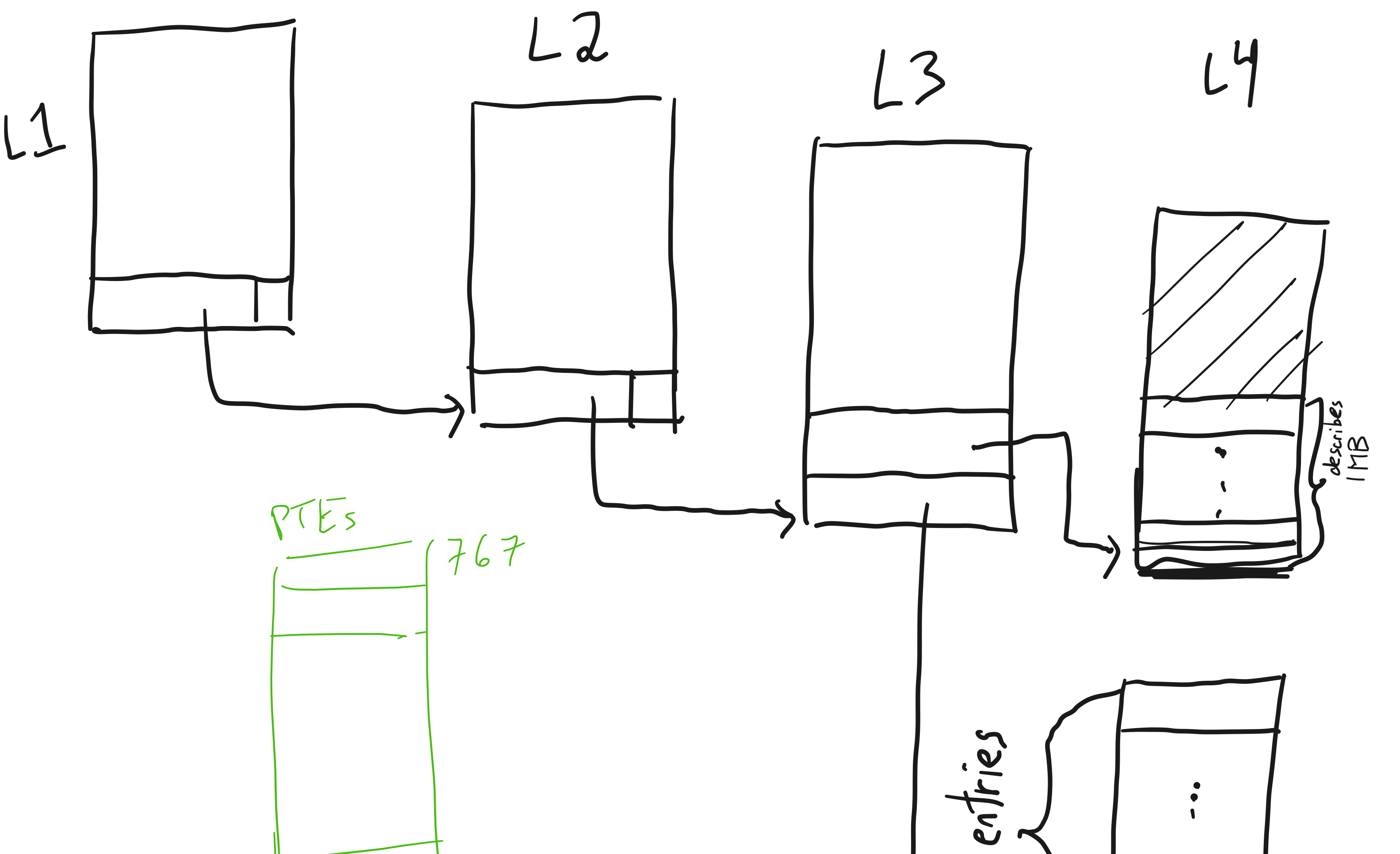
<VPN, PPN>

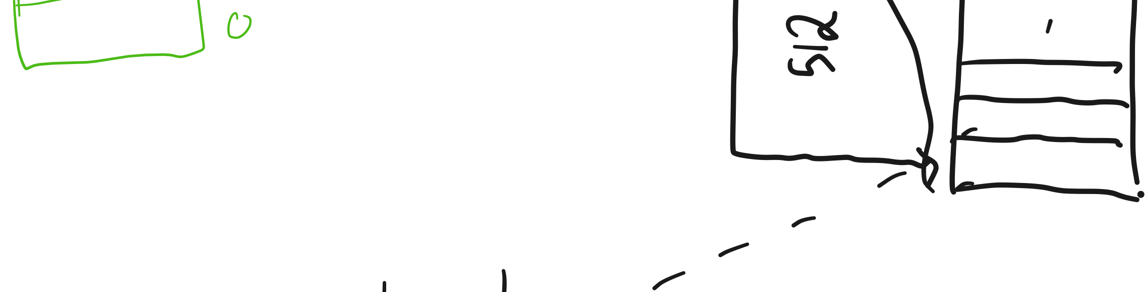
%rdi: arg to syscall

- you'll use virtual_memory_map()
  - pay attention to the allocator argument (NULL vs. non-NULL)
  - make sure your allocator initializes the page table

memset(addr, 0, len);

- a process's virtual address space: 3 MB. What's the page table structure?

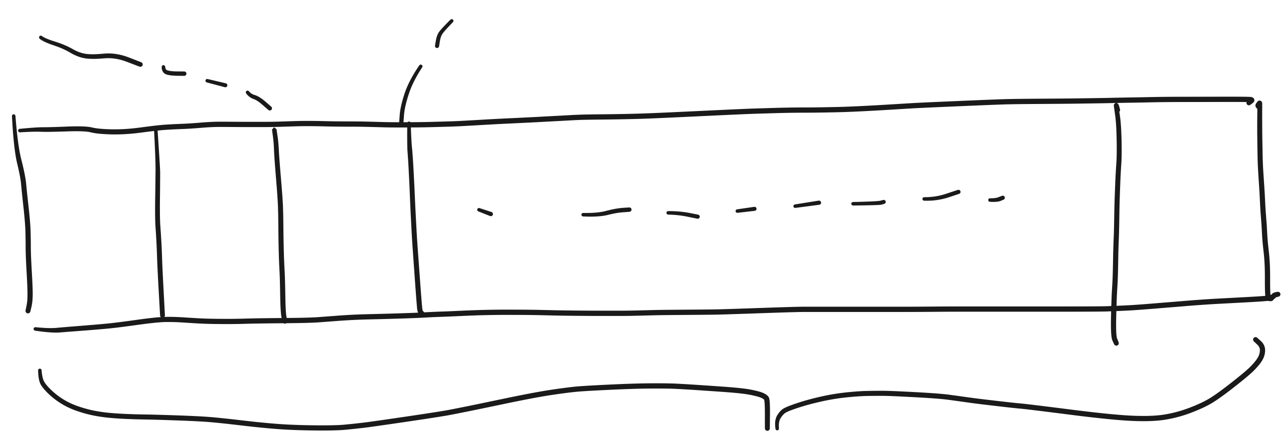





each entry
describes a mapping
for one page
(4KB)

PCB ≡ struct proc (in kernel.h)

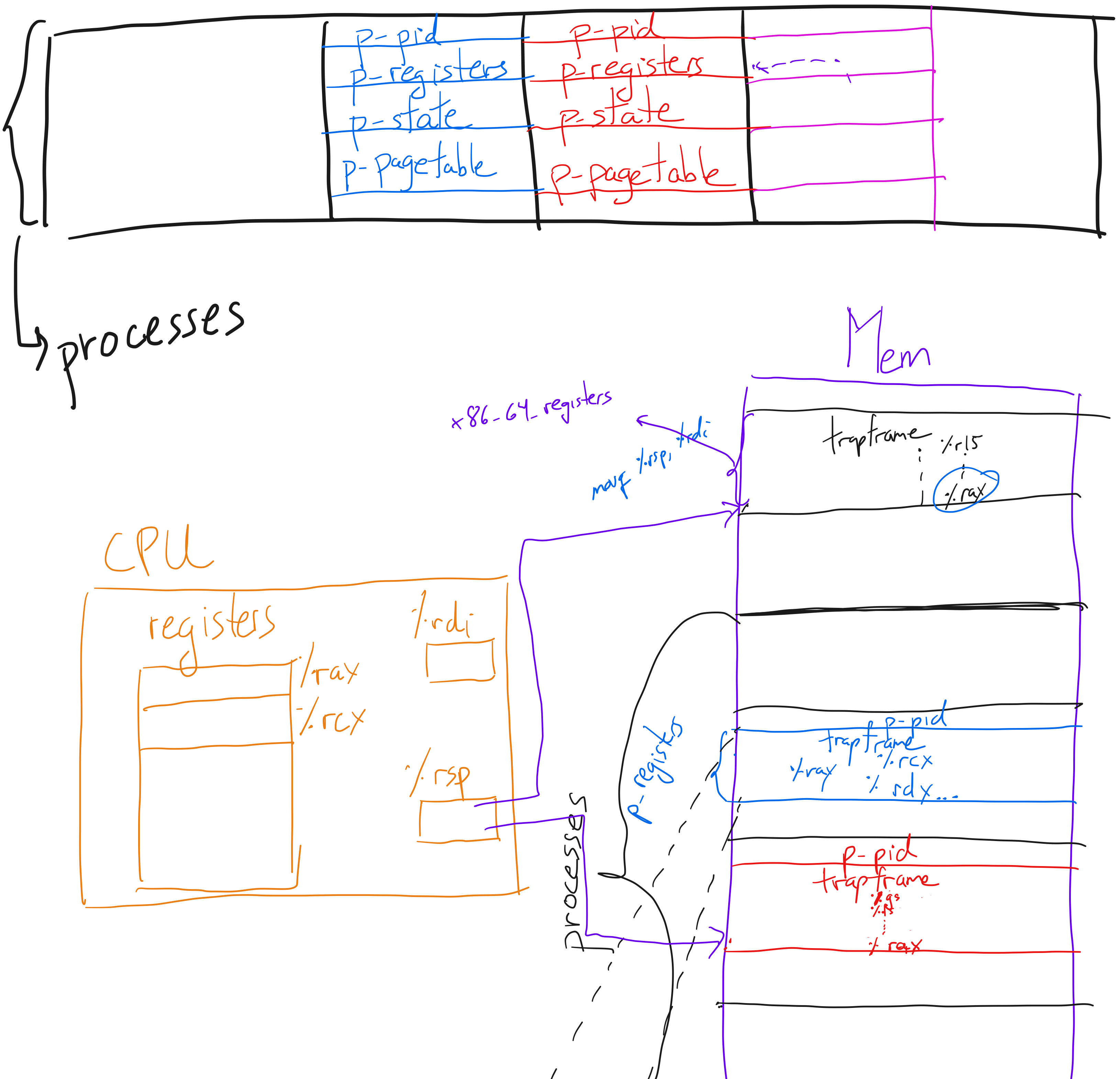
```
struct physical_pageinfo {
    int8_t owner;
    int8_t refcount;
}
```

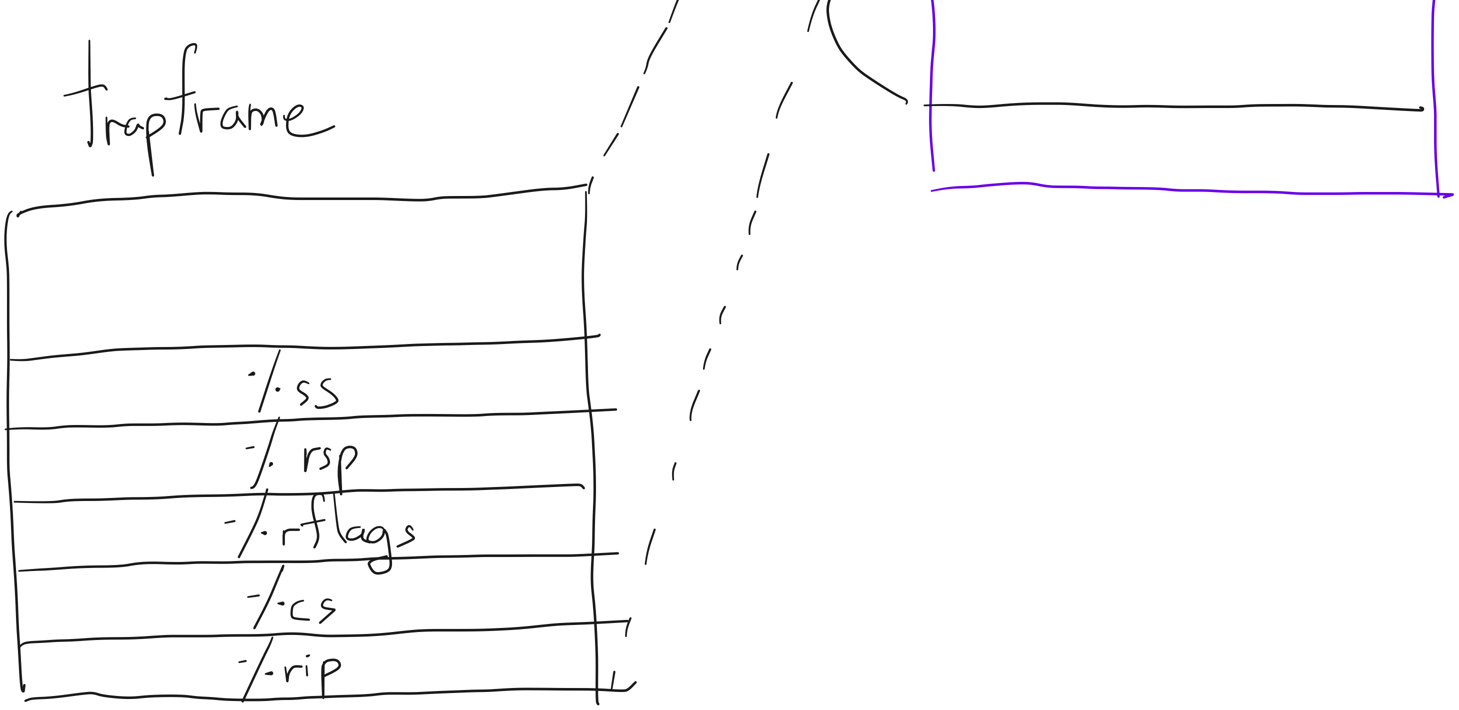


one per physical page

this is <u>not</u> a page table; it is bookkeeping.

# Context switches in Weensy OS




trapframe
%ss
%rsp
%rflags
%cs
%rip

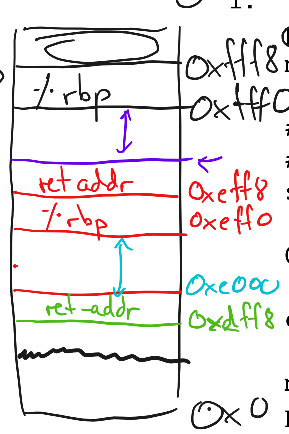
## VII Stack frames and virtual memory (12 points)

**9. [12 points]** Consider the following machine:

- It uses an instruction set and register names like the x86-64. It also has a special `OUT` instruction that delivers its argument to the output terminal. For example, `OUT 'Y'` prints `Y` to the screen.
- It's byte-addressed (like the x86-64). This machine's memory addresses are 16 bits, and there is virtual address translation via paging. 12 bits are used for the offset, so the page size is 4 KB (4096 bytes), as in x86-64. That leaves four bits of the address for the virtual page number, which is used as an index directly into a single *linear* page table; there is no page table walking.
- A page table entry indicates that a virtual memory page is valid if the bottom bit of the entry is set (there is no R/W or U/S memory protection).
- There is no paging to the disk: if a process references a virtual page whose corresponding page table entry is invalid, the OS handles the corresponding page fault by ending the process.
- Every stack slot is 8 bytes.

The assembly program below (on the left) executes, during which the machine uses the page table below, on the right (arranged from highest index to lowest).

Notice that this assembly program is recursive. Assume that the program begins with the instruction pointer, `%rip`, set to the address of `f` while the stack pointer, `%rsp`, starts out equal to `0xfff8` (this is the address of the last 8-byte quantity in the process's virtual memory space). Further assume that the code lives in the virtual address space between addresses `0x1000` and `0x1fff`.



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

    # subtract 4080 (in decimal) from
    # the stack pointer
    subq $4080, %rsp

    OUT 'X'

    call f

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

| PPN | valid | index |
|---|---|---|
| 0x30 | 1 | 15 |
| 0x7C | 1 | |
| 0x12 | 1 | |
| 0x9F | 1 | |
| 0x45 | 0 | |
| 0xB2 | 0 | |
| 0x02 | 1 | |
| 0xF8 | 0 | |
| 0x5A | 1 | |
| 0xAF | 1 | |
| 0x0C | 0 | |
| 0x8E | 0 | |
| 0x6B | 1 | |
| 0xE0 | 0 | |
| 0x3C | 1 | |
| 0xDA | 0 | 0 |

**What does this program output?**

- **A** Nothing; it page faults before producing output
- **B** Nothing; it hangs without page faulting
- **C** Infinite sequence of `X`
- **D** One `X`
- **E** Two `X`
- **F** Three `X`
- **G** Four `X`
- **H** Five `X`
- **I** Nine `X`
- **J** Sixteen `X`