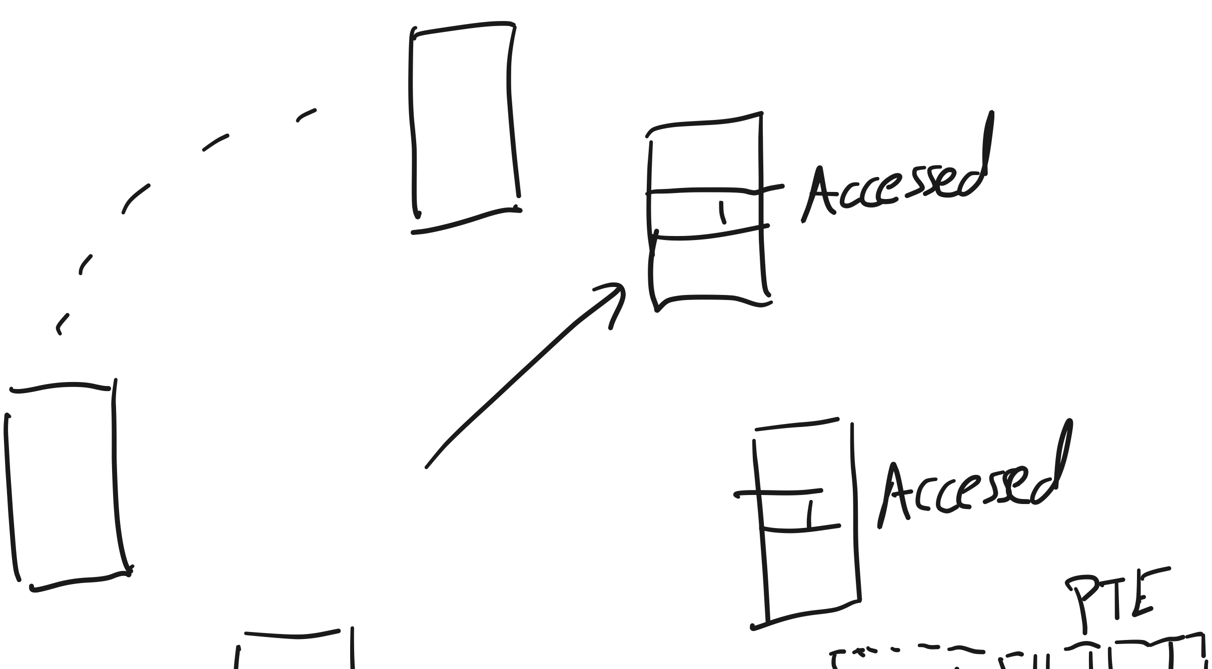
1. Last time
2. Page replacement policies (continued)
3. Thrashing
4. mmap()
5. Where does the OS live?
6. Weensy OS

---

2. Page replacement, contd.

- OPT minimizes misses/swaps/evictions
  - but can't be implemented in general
- LRU: approximates OPT (assuming what?)
- approximate LRU with CLOCK



H/w sets Accessed + Dirty bits

OS consumes these bits and clears them

- Generalization of CLOCK: N^{th} Chance (see notes)

---

3. Thrashing

<u>Ex</u>

- Prog. touches 50 pages, each equally likely
- 40 slots (pages) available in RAM

100ns/mem ref

10ms/page fault that reads from disk

$$\frac{4\left(\frac{100ns}{\cancel{mem}\ ref}\right) + 1\left(\frac{10ms}{\cancel{mem}\ ref}\right)}{5} \approx \frac{2ms}{ref}$$

shed load:

(a) working set

(b) pg fault freq.

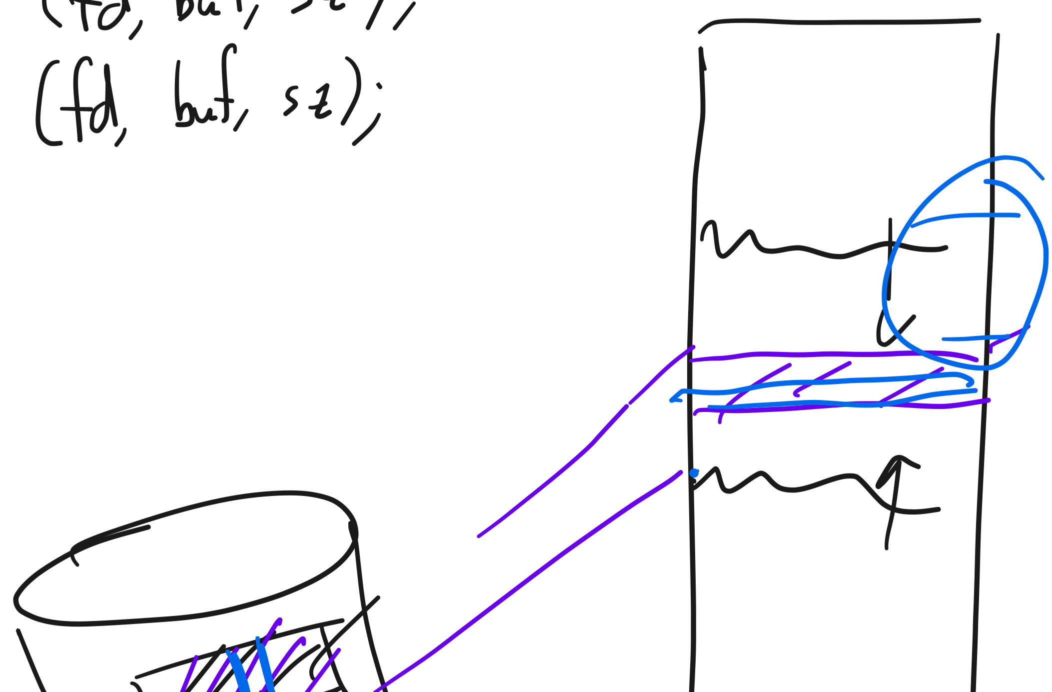
---

4. mmap()

recall:

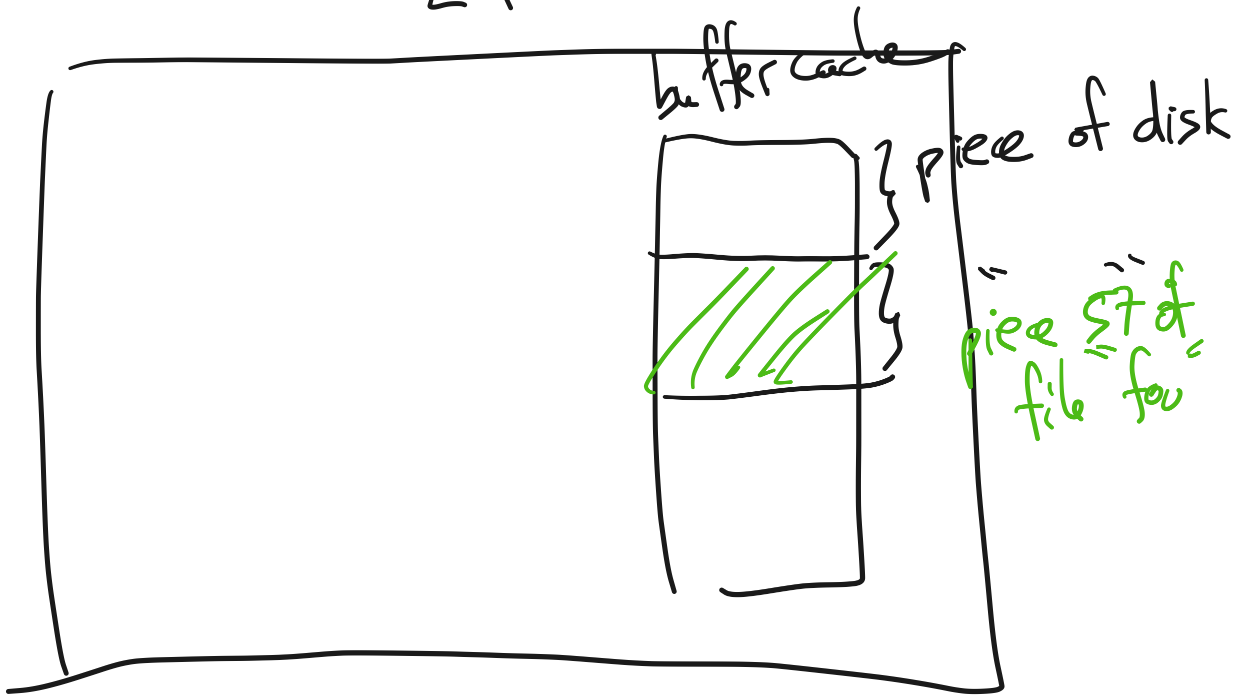
```
fd = open(pathname, mode);
rc = write(fd, buf, sz);
rc = read(fd, buf, sz);
```

mmap()



<<proc, VPN>, entry>

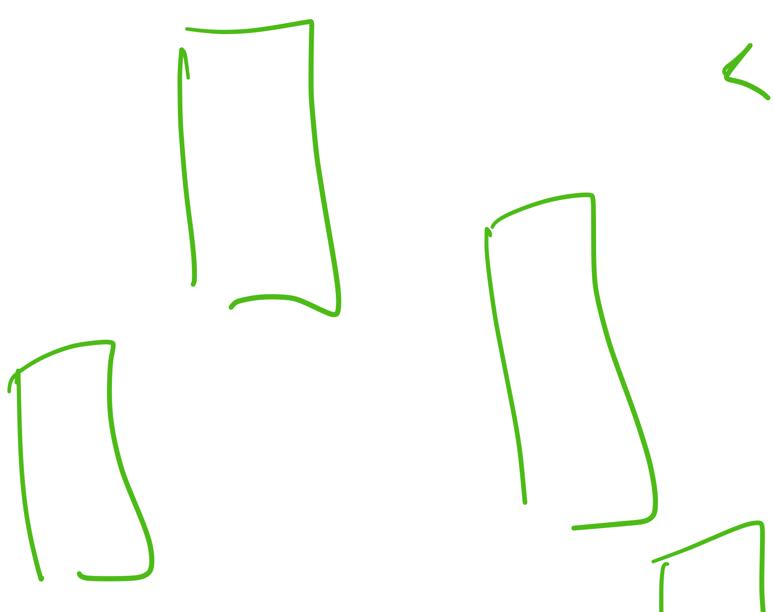
OS



page table for proc that mapped file "foo"

page table gets a new mapping

<mmaped ptr, PPN of the relevant buffer cache entry>



median: 74

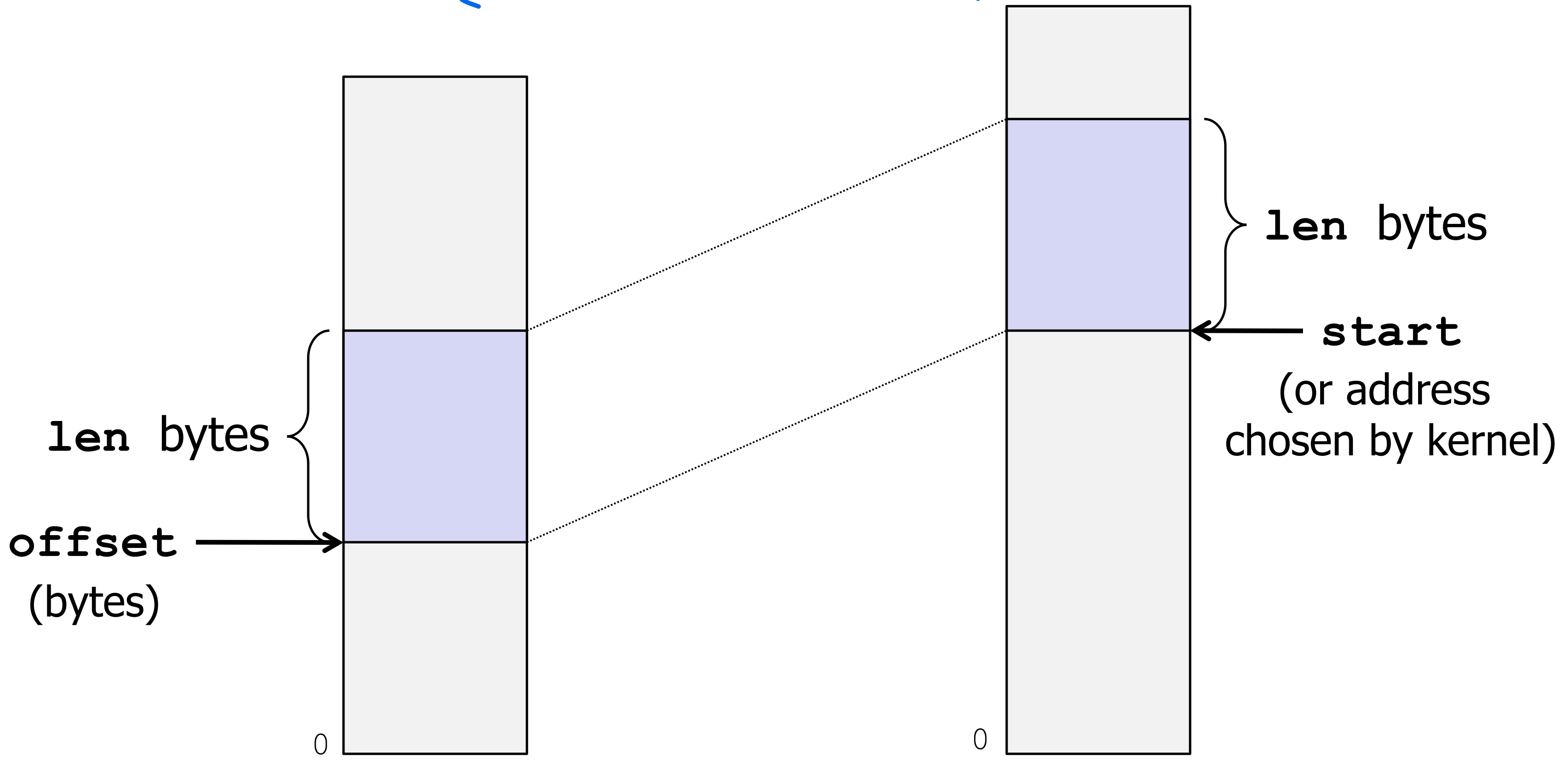
avg: 72

σ : 18.2

high: 98

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



Disk file specified by file descriptor **fd**

Process virtual memory

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

void mmapcopy(int fd, int size);

int main(int argc, char **argv) {
  struct stat stat;
  int fd;

  /* Check for required cmd line arg */
  if (argc != 2) {
    printf("usage: %s <filename>\n", argv[0]);
    exit(0);
  }

  /* Copy input file to stdout */
  if ((fd = open(argv[1], O_RDONLY, 0)) < 0)
    perror("open");

  fstat(fd, &stat);
  mmapcopy(fd, stat.st_size);

  close(fd);

  return 0;
}

void mmapcopy(int fd, int size) {

    /* Ptr to memory mapped area */
    char *bufp;

    bufp = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);

    write(STDOUT_FILENO, bufp, size);

    return;
}
```

$ cat

$ cat bar

assert((fd != -1)

char buf[512];

int rc;

int fd = open(argv[1], RD_ONLY);

while ((rc = read(fd, buf, sizeof(buf)),
!= NULL) {

write(1, buf, rc);

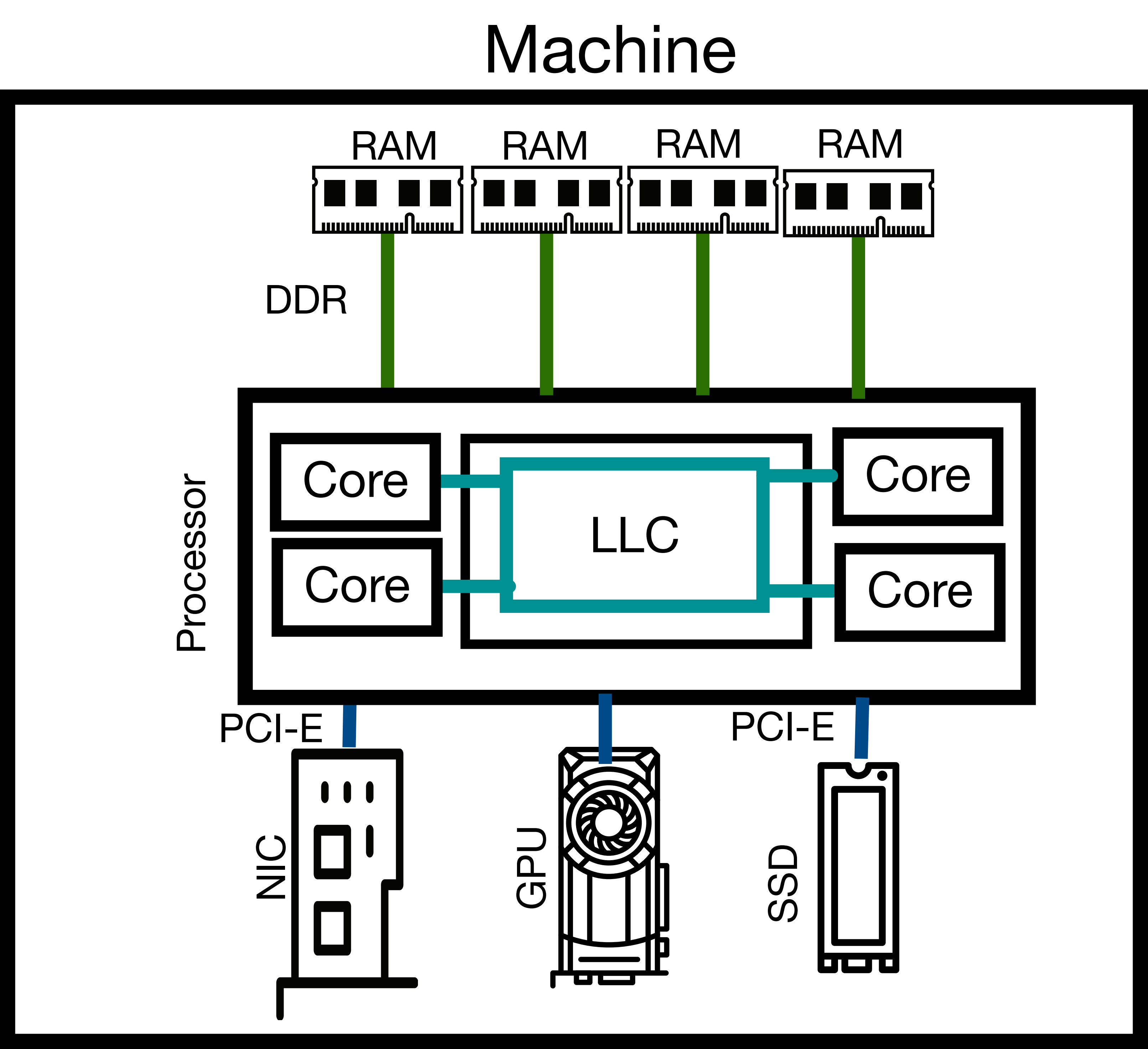
}

write

1 memcopy:
from buffer cache
to buffer in front
of terminal

sendfile();

2 copies
buffer cache → user-buf
user-buf → buffer in front of
terminal


Machine
RAM
RAM
RAM
RAM
DDR
Processor
Core
Core
LLC
Core
Core
PCI-E
PCI-E
NIC
GPU
SSD

CS 202, Fall 2021
Handout 9 (Class 15)

1. Example use of I/O instructions: boot loader

    Below is the WeensyOS boot loader

    It may be helpful to understand the overall picture

    This code demonstrates I/O, specifically with the disk: the bootloader reads in the kernel from the disk.

    See the functions boot_waitdisk() and boot_readsect(). Compare to Figures 36.5 and 36.6 in OSTEP.

```
/* boot.c */
#include "x86-64.h"
#include "elf.h"

// boot.c
//
//   WeensyOS boot loader. Loads the kernel at address 0x40000 from
//   the first IDE hard disk.
//
//   A BOOT LOADER is a tiny program that loads an operating system into
//   memory. It has to be tiny because it can contain no more than 510 bytes
//   of instructions: it is stored in the disk's first 512-byte sector.
//
//   When the CPU boots it loads the BIOS into memory and executes it. The
//   BIOS intializes devices and CPU state, reads the first 512-byte sector of
//   the boot device (hard drive) into memory at address 0x7C00, and jumps to
//   that address.
//
//   The boot loader is contained in bootstart.S and boot.c. Control starts
//   in bootstart.S, which initializes the CPU and sets up a stack, then
//   transfers here. This code reads in the kernel image and calls the
//   kernel.
//
//   The main kernel is stored as an ELF executable image starting in the
//   disk's sector 1.

#define SECTORSIZE      512
#define ELFHDR          ((elf_header*) 0x10000) // scratch space

void boot(void) __attribute__((noreturn));
static void boot_readsect(uintptr_t dst, uint32_t src_sect);
static void boot_readseg(uintptr_t dst, uint32_t src_sect,
                         size_t filesz, size_t memsz);

// boot
//    Load the kernel and jump to it.
void boot(void) {
    // read 1st page off disk (should include programs as well as header)
    // and check validity
    boot_readseg((uintptr_t) ELFHDR, 1, PAGESIZE, PAGESIZE);
    while (ELFHDR->e_magic != ELF_MAGIC) {
        /* do nothing */
    }

    // load each program segment
    elf_program* ph = (elf_program*) ((uint8_t*) ELFHDR + ELFHDR->e_phoff);
    elf_program* eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ++ph) {
        boot_readseg(ph->p_va, ph->p_offset / SECTORSIZE + 1,
                     ph->p_filesz, ph->p_memsz);
    }

    // jump to the kernel
    typedef void (*kernel_entry_t)(void) __attribute__((noreturn));
    kernel_entry_t kernel_entry = (kernel_entry_t) ELFHDR->e_entry;
    kernel_entry();
}
```



```


// boot_readseg(dst, src_sect, filesz, memsz)
//    Load an ELF segment at virtual address `dst` from the IDE disk's sector
//    `src_sect`. Copies `filesz` bytes into memory at `dst` from sectors
//    `src_sect` and up, then clears memory in the range
//    `[dst+filesz, dst+memsz)`.
static void boot_readseg(uintptr_t ptr, uint32_t src_sect,
                         size_t filesz, size_t memsz) {
    uintptr_t end_ptr = ptr + filesz;
    memsz += ptr;

    // round down to sector boundary
    ptr &= ~(SECTORSIZE - 1);

    // read sectors
    for (; ptr < end_ptr; ptr += SECTORSIZE, ++src_sect) {
        boot_readsect(ptr, src_sect);
    }

    // clear bss segment
    for (; end_ptr < memsz; ++end_ptr) {
        *(uint8_t*) end_ptr = 0;
    }
}


// boot_waitdisk
//    Wait for the disk to be ready.
static void boot_waitdisk(void) {
    // Wait until the ATA status register says ready (0x40 is on)
    // & not busy (0x80 is off)
    while ((inb(0x1F7) & 0xC0) != 0x40) {
        /* do nothing */
    }
}


// boot_readsect(dst, src_sect)
//    Read disk sector number `src_sect` into address `dst`.
static void boot_readsect(uintptr_t dst, uint32_t src_sect) {
    // programmed I/O for "read sector"
    boot_waitdisk();
    outb(0x1F2, 1);             // send `count = 1` as an ATA argument
    outb(0x1F3, src_sect);      // send `src_sect`, the sector number
    outb(0x1F4, src_sect >> 8);
    outb(0x1F5, src_sect >> 16);
    outb(0x1F6, (src_sect >> 24) | 0xE0);
    outb(0x1F7, 0x20);          // send the command: 0x20 = read sectors

    // then move the data into memory
    boot_waitdisk();
    insl(0x1F0, (void*) dst, SECTORSIZE/4); // read 128 words from the disk
}
```

2. Two more examples of I/O instructions

(a) Reading keyboard input

The code below is an excerpt from WeensyOS's k-hardware.c

This reads a character typed at the keyboard (which shows up on the "keyboard data port" (kEYBOARD_DATAREG)).

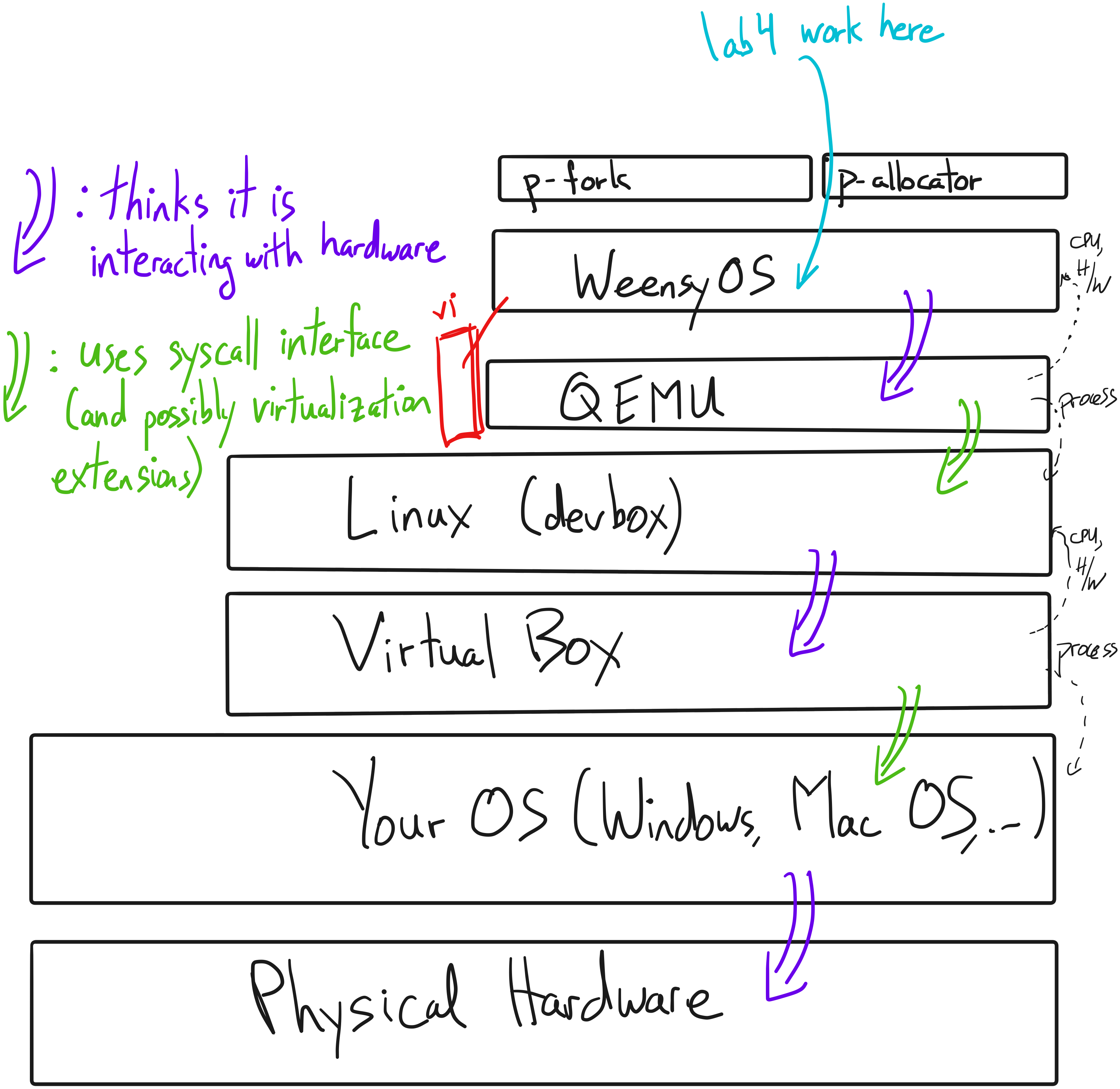
```
/* Excerpt from WeensyOS x86-64.h */
// Keyboard programmed I/O
#define KEYBOARD_STATUSREG      0x64
#define KEYBOARD_STATUS_READY   0x01
#define KEYBOARD_DATAREG        0x60

int keyboard_readc(void) {
    static uint8_t modifiers;
    static uint8_t last_escape;

    if ((inb(KEYBOARD_STATUSREG) & KEYBOARD_STATUS_READY) == 0) {
        return -1;
    }

    uint8_t data = inb(KEYBOARD_DATAREG);
    uint8_t escape = last_escape;
    last_escape = 0;

    if (data == 0xE0) {         // mode shift
        last_escape = 0x80;
        return 0;
    } else if (data & 0x80) {   // key release: matters only for modifier keys
        int ch = keymap[(data & 0x7F) | escape];
        if (ch >= KEY_SHIFT && ch < KEY_CAPSLOCK) {
            modifiers &= ~(1 << (ch - KEY_SHIFT));
        }
        return 0;
    }

    int ch = (unsigned char) keymap[data | escape];

    if (ch >= 'a' && ch <= 'z') {
        if (modifiers & MOD_CONTROL) {
            ch -= 0x60;
        } else if (!(modifiers & MOD_SHIFT) != !(modifiers & MOD_CAPSLOCK)) {
            ch -= 0x20;
        }
    } else if (ch >= KEY_CAPSLOCK) {
        modifiers ^= 1 << (ch - KEY_SHIFT);
        ch = 0;
    } else if (ch >= KEY_SHIFT) {
        modifiers |= 1 << (ch - KEY_SHIFT);
        ch = 0;
    } else if (ch >= CKEY(0) && ch <= CKEY(21)) {
        ch = complex_keymap[ch - CKEY(0)].map[modifiers & 3];
    } else if (ch < 0x80 && (modifiers & MOD_CONTROL)) {
        ch = 0;
    }

    return ch;
}
```


(b) Setting the cursor position

The code below is also excerpted from WeensyOS's k-hardware.c. It uses I/O instructions to set a blinking cursor in the upper left of the screen.

```
// console_show_cursor(cpos)
//    Move the console cursor to position `cpos`, which should be between 0
//    and 80 * 25.

void console_show_cursor(int cpos) {
    if (cpos < 0 || cpos > CONSOLE_ROWS * CONSOLE_COLUMNS) {
        cpos = 0;

    outb(0x3D4, 14);        // Command 14 = upper byte of position
    outb(0x3D5, 0 / 256);   // row 0
    outb(0x3D4, 15);        // Command 15 = lower byte of position
    outb(0x3D5, 0 % 256);   // column 0

}
```

```
3. Memory-mapped I/O

    a. Here is a 32-bit PC's physical memory map:

        +------------------+  <- 0xFFFFFFFF (4GB)
        |      32-bit      |
        |  memory mapped   |
        |     devices      |
        |                  |
        /\/\/\/\/\/\/\/\/\/\

        /\/\/\/\/\/\/\/\/\/\
        |                  |
        |      Unused      |
        |                  |
        +------------------+  <- depends on amount of RAM
        |                  |
        |                  |
        | Extended Memory  |
        |                  |
        |                  |
        +------------------+  <- 0x00100000 (1MB)
        |     BIOS ROM     |
        +------------------+  <- 0x000F0000 (960KB)
        |  16-bit devices, |
        |  expansion ROMs  |
        +------------------+  <- 0x000C0000 (768KB)
        |   VGA Display    |
        +------------------+  <- 0x000A0000 (640KB)
        |                  |
        |    Low Memory    |
        |                  |
        +------------------+  <- 0x00000000

    [Credit to Frans Kaashoek, Robert Morris, and Nickolai Zeldovich for
    this picture]


    b. Loads and stores to the device memory "go to hardware".

    An example is in the console printing code from WeensyOS. Here is an
    excerpt from link/shared.ld:

    /* Compare the address below to the map above. */
    PROVIDE(console = 0xB8000);

    /*
     * prints a character to the console at the specified
     * cursor position in the specified color.
     * Question: what is going on in the check
     *        if (c == '\n')
     * ?
     * Hint: '\n' is "C" for "newline" (the user pressed enter).
     */
    static void console_putc(printer* p, unsigned char c, int color) {
        console_printer* cp = (console_printer*) p;
        if (cp->cursor >= console + CONSOLE_ROWS * CONSOLE_COLUMNS) {
            cp->cursor = console;
        }
        if (c == '\n') {
            int pos = (cp->cursor - console) % 80;
            for (; pos != 80; pos++) {
                *cp->cursor++ = ' ' | color;
            }
        } else {
            *cp->cursor++ = c | color;
        }
    }
```



Hints:

- processes: files matching p-*.c
- kernel code: files matching k-*.c , k-*.s
- system calls and returns /* cousin of mmap() */

sys_page_alloc( );

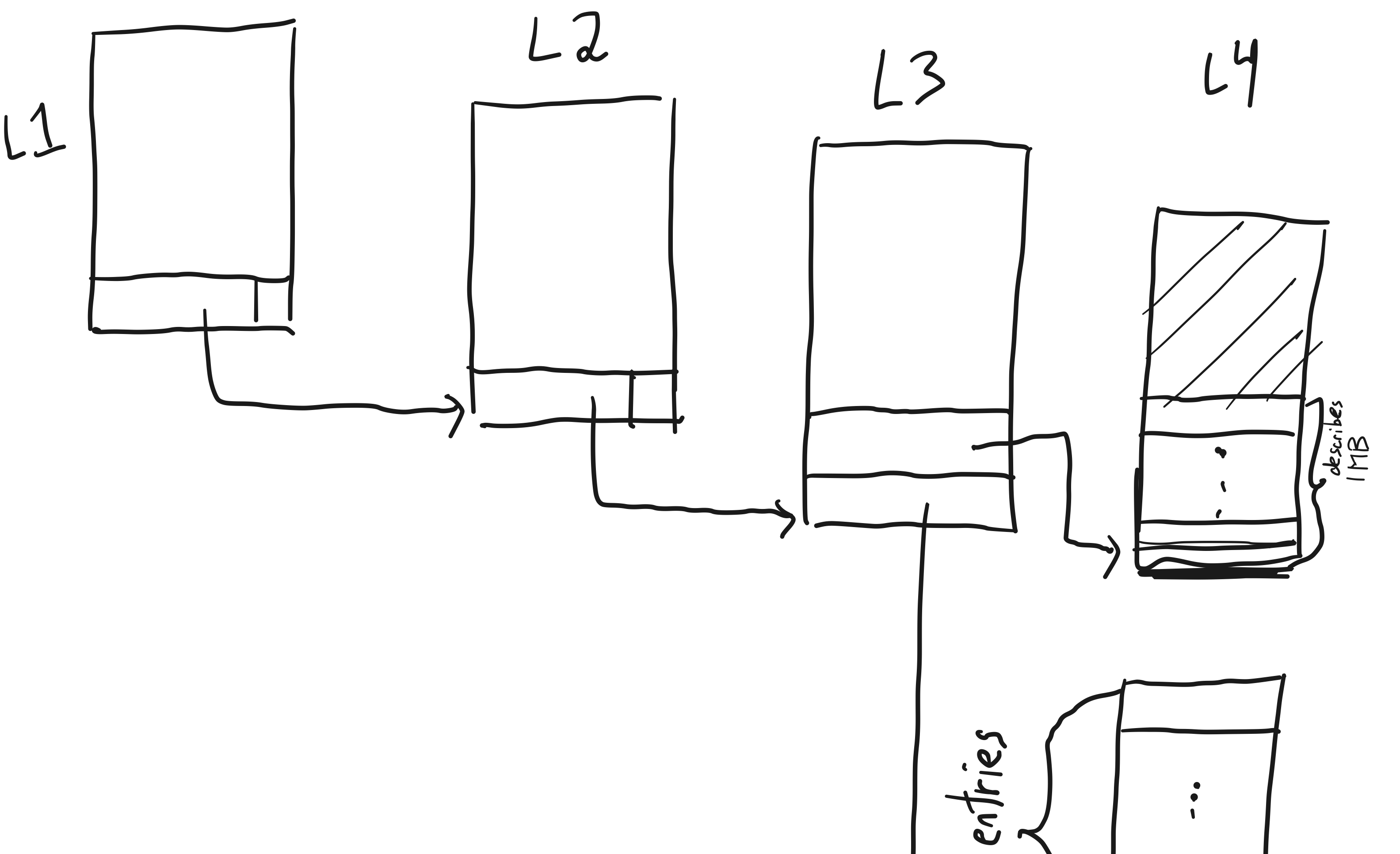
look in process.h

kernel returns: exception_return();
%rax contains return value of system call
<VPN, PPN>

%rdi: arg to syscall

- you'll use virtual_memory_map() (NULL vs. non-NULL)
  - pay attention to the allocator argument
  - make sure your allocator initializes the page table
    memset(addr, 0, len);

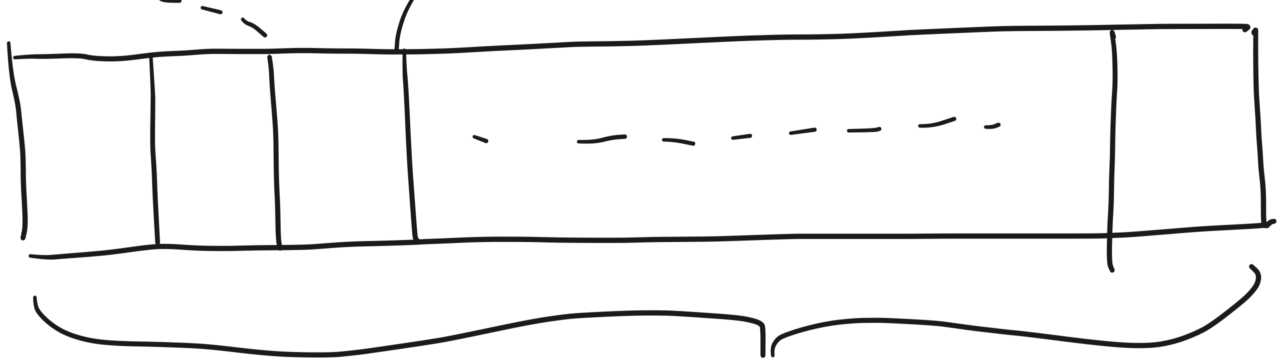
- a process's virtual address space: 3 MB. What's the page table structure?



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

PCB ≡ struct proc (in kernel.h)

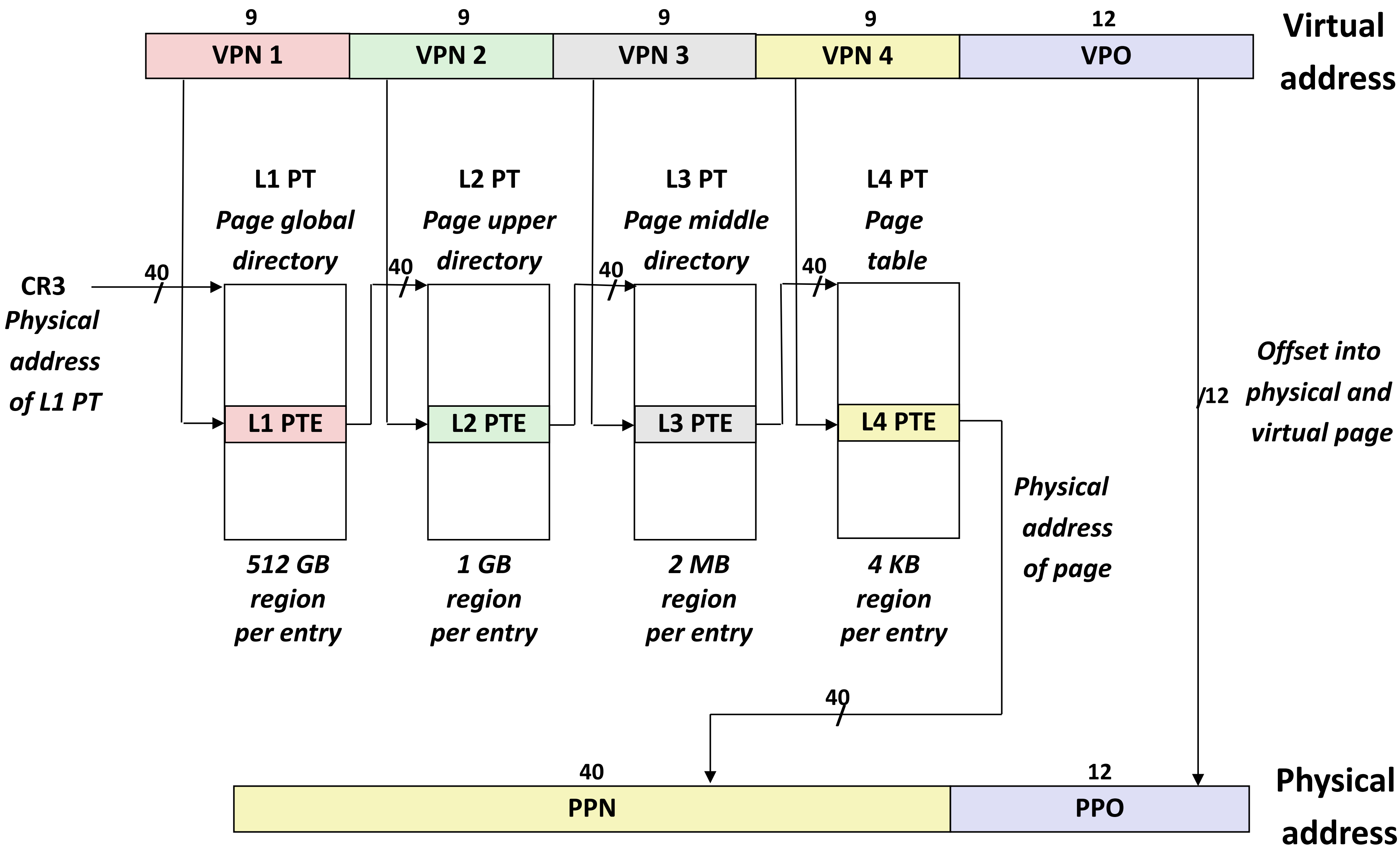
```
struct physical_pageinfo {
    int8_t owner;
    int8_t refcount;
}
```



one per physical page

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

Core i7 Page Table Translation



Review of Symbols

- **Basic Parameters**
  - **$N = 2^n$** : Number of addresses in virtual address space
  - **$M = 2^m$** : Number of addresses in physical address space
  - **$P = 2^p$** : Page size (bytes)
- **Components of the virtual address (VA)**
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
- **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number
  - **CO**: Byte offset within cache line
  - **CI:** Cache index
  - **CT**: Cache tag

Core i7 Level 1-3 Page Table Entries

63	62 52	51 12	11 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base address	Unused	G	PS		A	CD	WT	U/S	R/W	P=1

Available for OS	P=0

Each entry references a 4K child page table. Significant fields:

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size: if bit set, we have 2 MB or 1 GB pages (bit can be set in Level 2 and 3 PTEs only).

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

XD: Disable or enable instruction fetches from all pages reachable from this PTE.

Core i7 Level 4 Page Table Entries

63	62 52	51 12	11 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page physical base address	Unused	G		D	A	CD	WT	U/S	R/W	P=1

Available for OS (for example, if page location on disk)	P=0

Each entry references a 4K child page. Significant fields:

P: Child page is present in memory (1) or not (0)

R/W: Read-only or read-write access permission for this page

U/S: User or supervisor mode access

WT: Write-through or write-back cache policy for this page

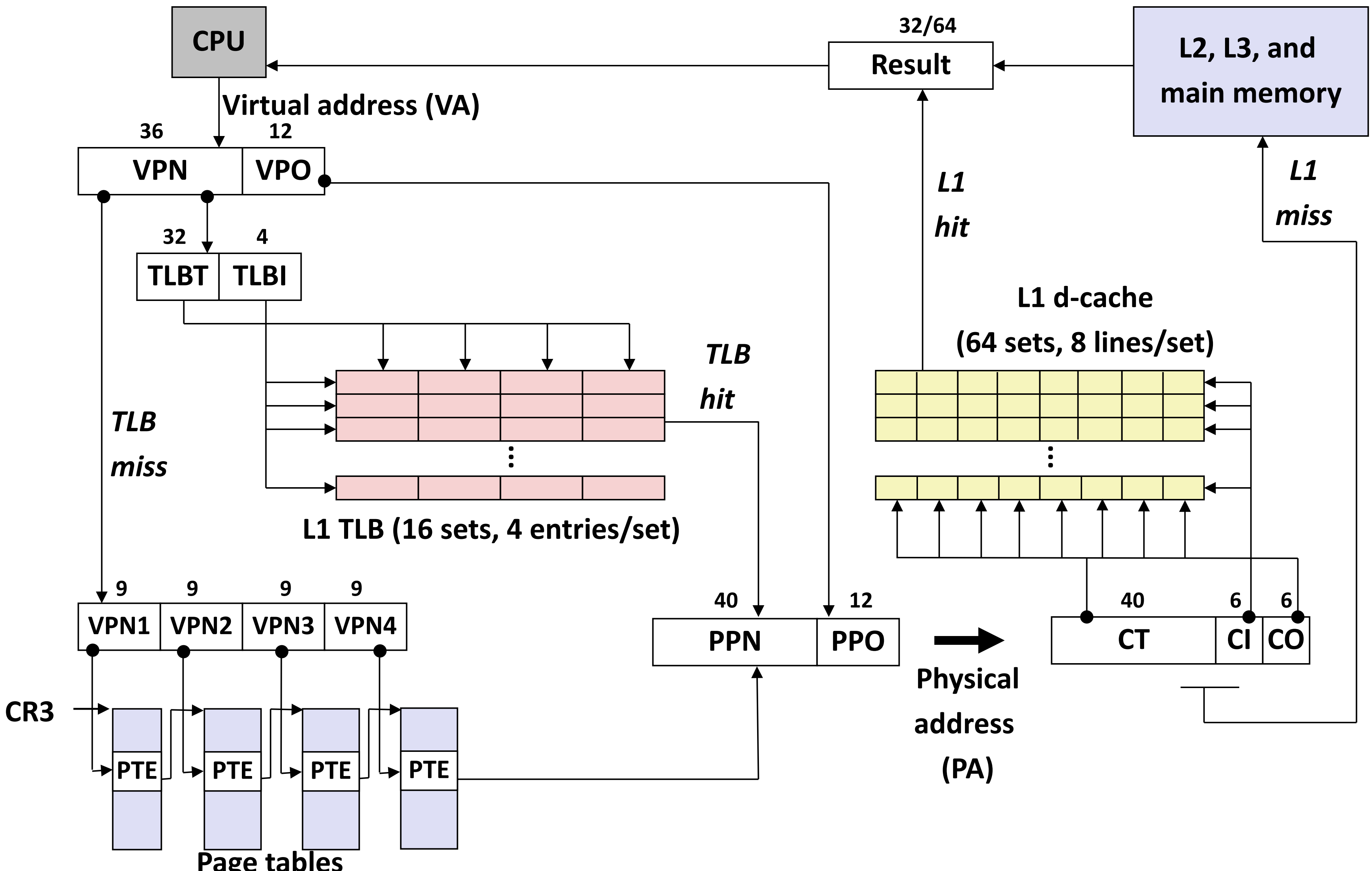
A: Reference bit (set by MMU on reads and writes, cleared by software)

D: Dirty bit (set by MMU on writes, cleared by software)

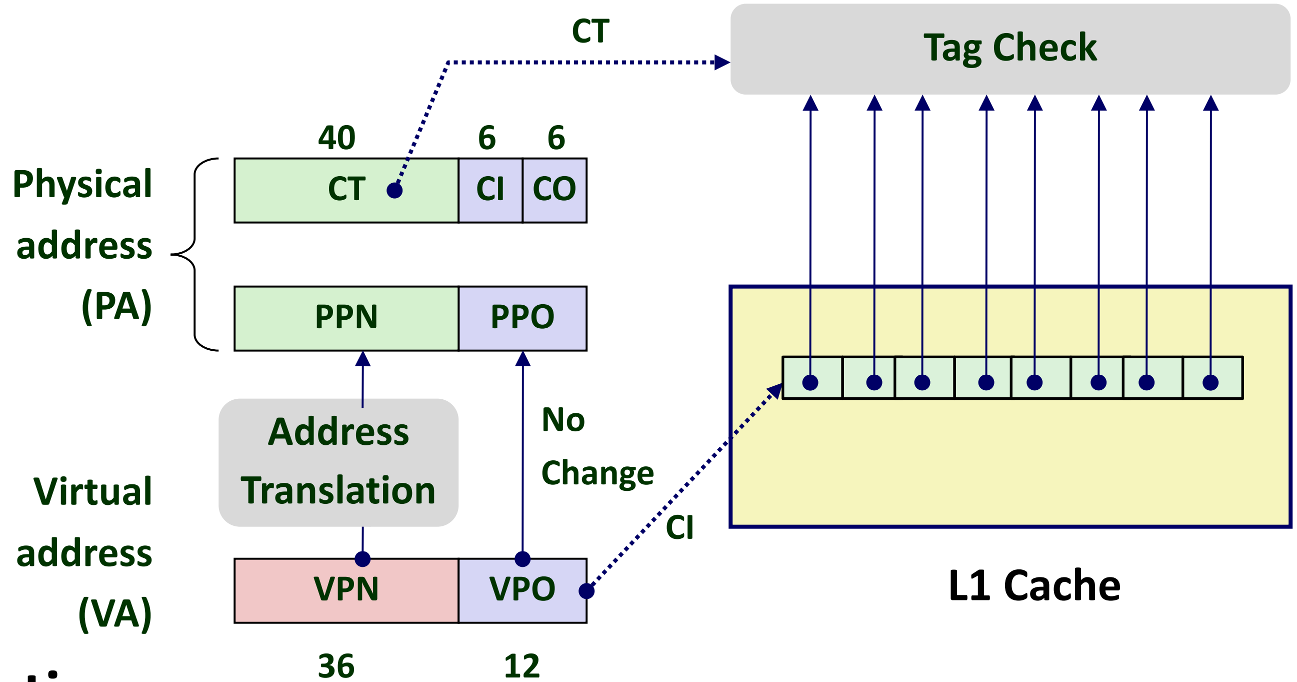
Page physical base address: 40 most significant bits of physical page address
(forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.

End-to-end Core i7 Address Translation

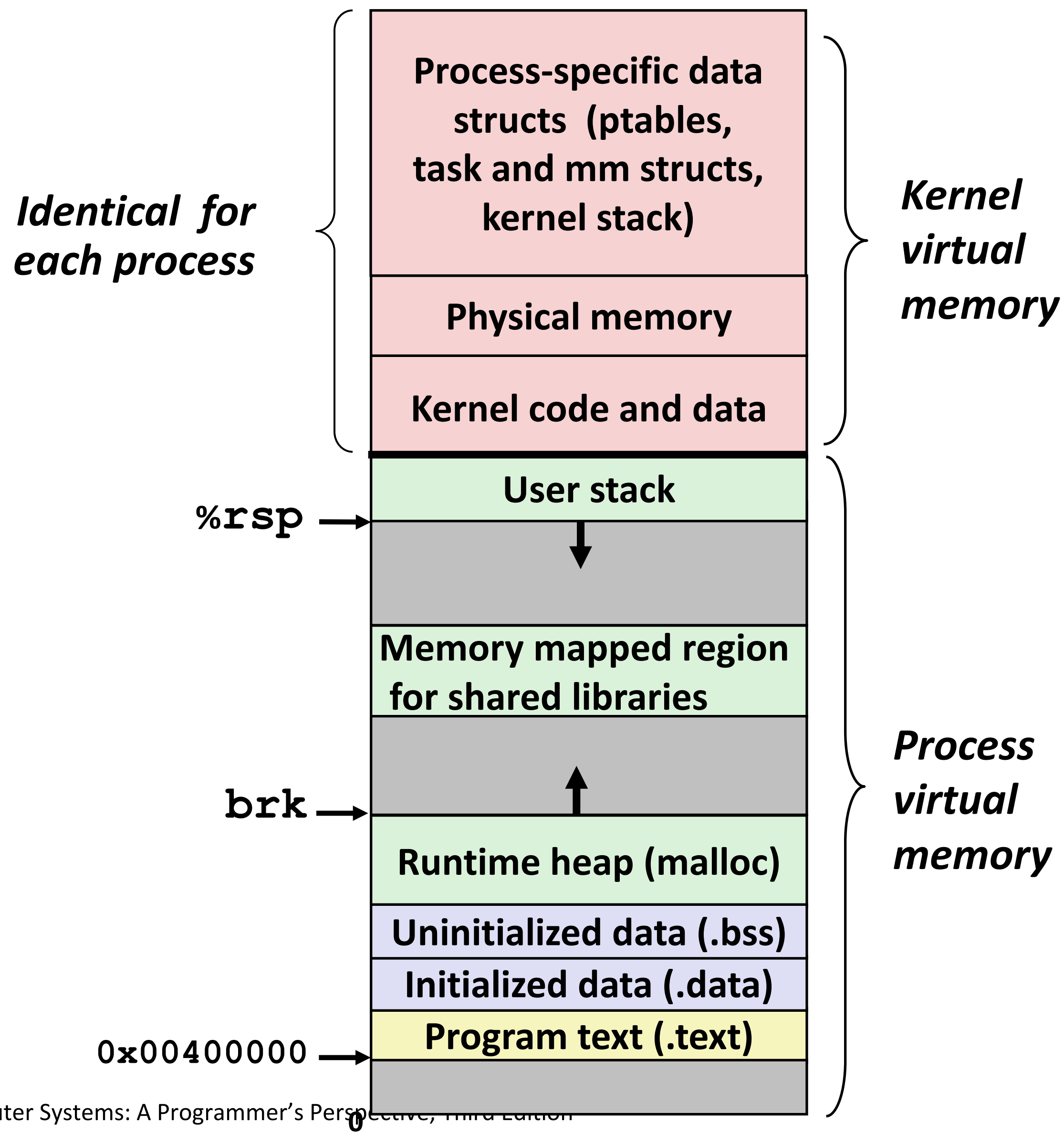


Cute Trick for Speeding Up L1 Access



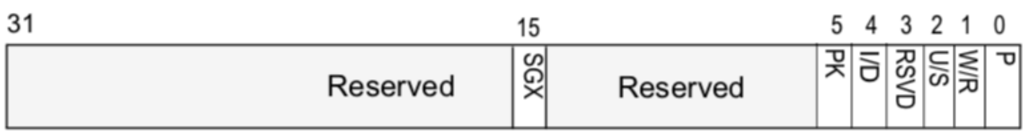
- **Observation**
  - Bits that determine CI identical in virtual and physical address
  - Can index into cache while address translation taking place
  - Cache carefully sized to make this possible: 64 sets, 64-byte cache blocks
  - Means 6 bits for cache index, 6 for *cache* offset
  - That's 12 bits; matches *VPO, PPO* → One reason pages are 2^{12} bits = 4 KB

Virtual Address Space of a Linux Process



meltdown

Spectre



P	0	The fault was caused by a non-present page.
	1	The fault was caused by a page-level protection violation.
W/R	0	The access causing the fault was a read.
	1	The access causing the fault was a write.
U/S	0	A supervisor-mode access caused the fault.
	1	A user-mode access caused the fault.
RSVD	0	The fault was not caused by reserved bit violation.
	1	The fault was caused by a reserved bit set to 1 in some paging-structure entry.
I/D	0	The fault was not caused by an instruction fetch.
	1	The fault was caused by an instruction fetch.
PK	0	The fault was not caused by protection keys.
	1	There was a protection-key violation.
SGX	0	The fault is not related to SGX.
	1	The fault resulted from violation of SGX-specific access-control requirements.

Figure 4-12. Page-Fault Error Code