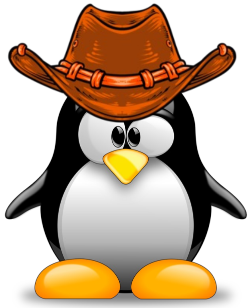
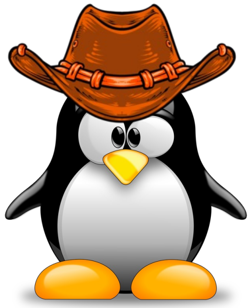


CSE 330: Operating Systems

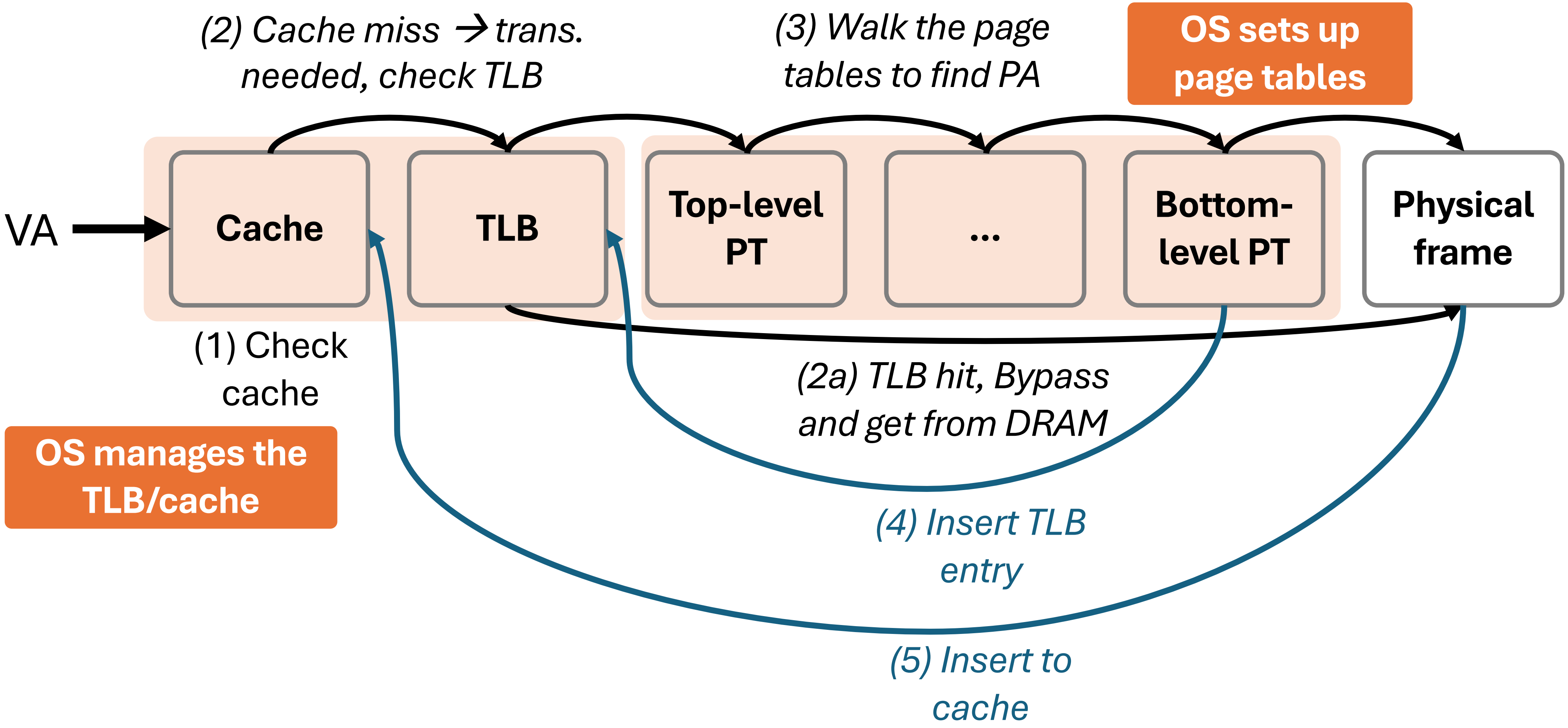
Adil Ahmad

Lecture #16: SSDs versus HDDs, higher-level abstractions



Project #4: brief introduction

An end-to-end memory access graph



The `mmap` system call in Linux

- **What is `mmap`?**
  - Used by processes to request the OS to allocate new memory
- `mmap` (address, …, permissions, size, …)
- Let's take a look at an example:
  - `mmap` (0x1000, ..., READ | WRITE, 4096, ...)
  - **What does this mean?**
    - Allocate 4096 bytes (1 page) of memory at virtual address 0x1000

Let's design the functionality of mmap together!

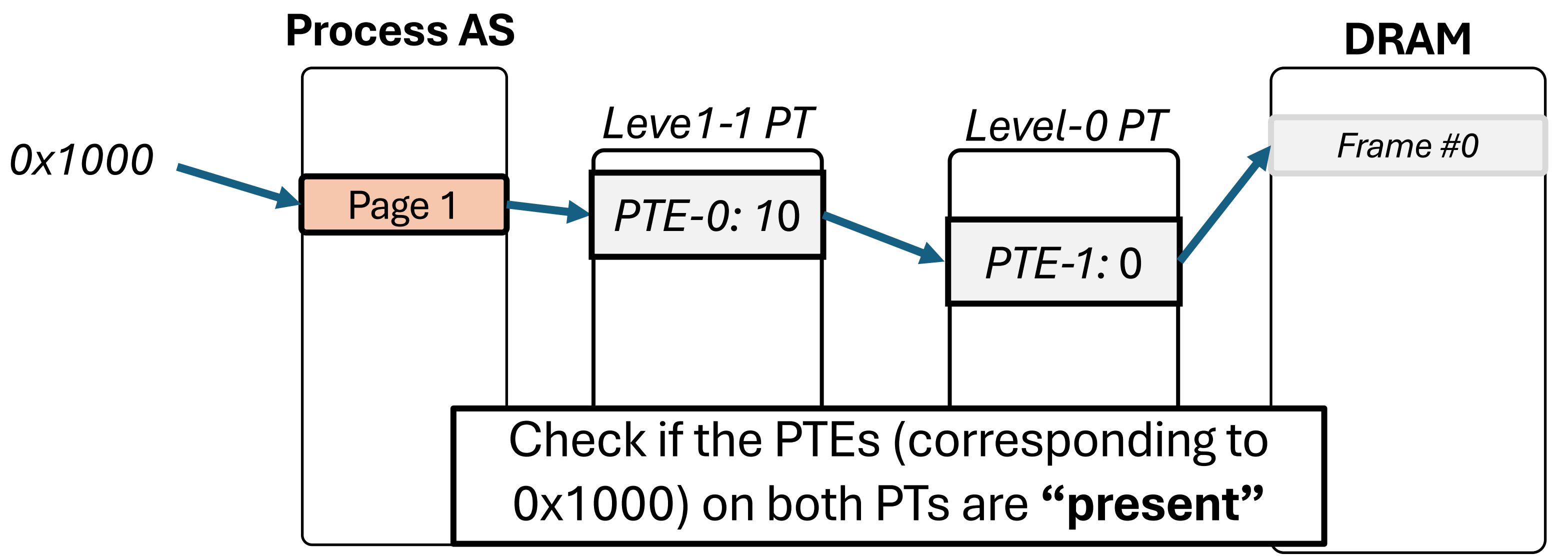
What steps would you follow to build `mmap` implementation?

```
mmap (0x1000, …, READ | WRITE, 4096, …)
```

- Step #0: check if the requested “addr” is already mapped
- Step #1: find a free physical page (or *frame*) to allocate
- Step #2: allocate (and map) the page tables
- Step #3: map the page frame

Step #0: Check if memory is already mapped at 0x1000

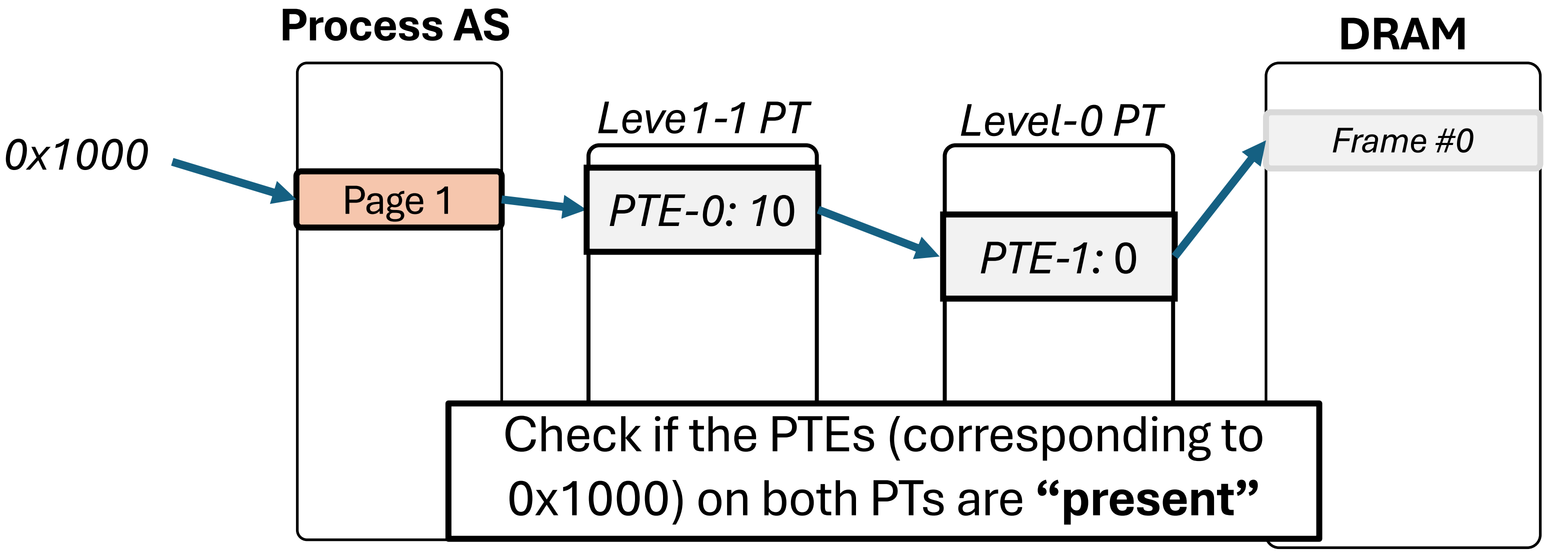
- **How do we know if memory is mapped at the address?**
  - One simple way is to traverse *or walk* the process' page tables!



How would you check that a PTE is "present"?

Step #0: Check if memory is already mapped at 0x1000

- **How do we know if memory is mapped at the address?**
  - One simple way is to traverse *or walk* the process' page tables!

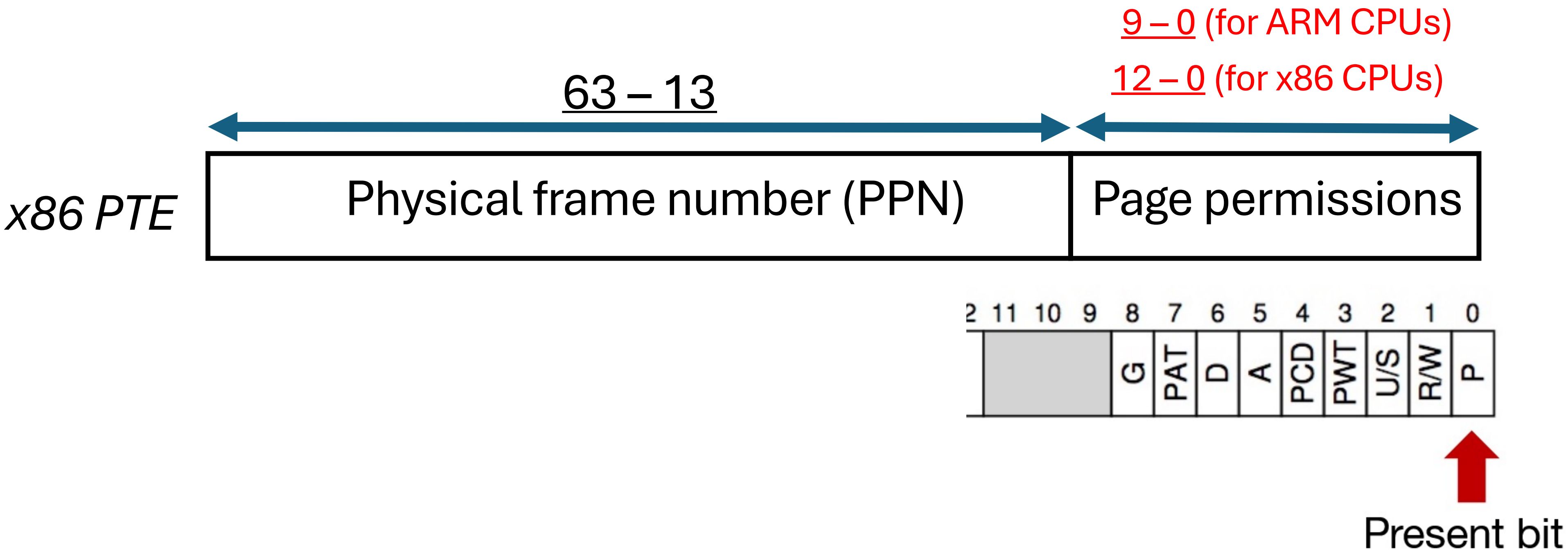


Check if the PTEs (corresponding to 0x1000) on both PTs are **"present"**

How would you check that a PTE is "present"?

Step #0a: Checking if a page table entry is present

- Recall the layout of page table entries (PTEs):



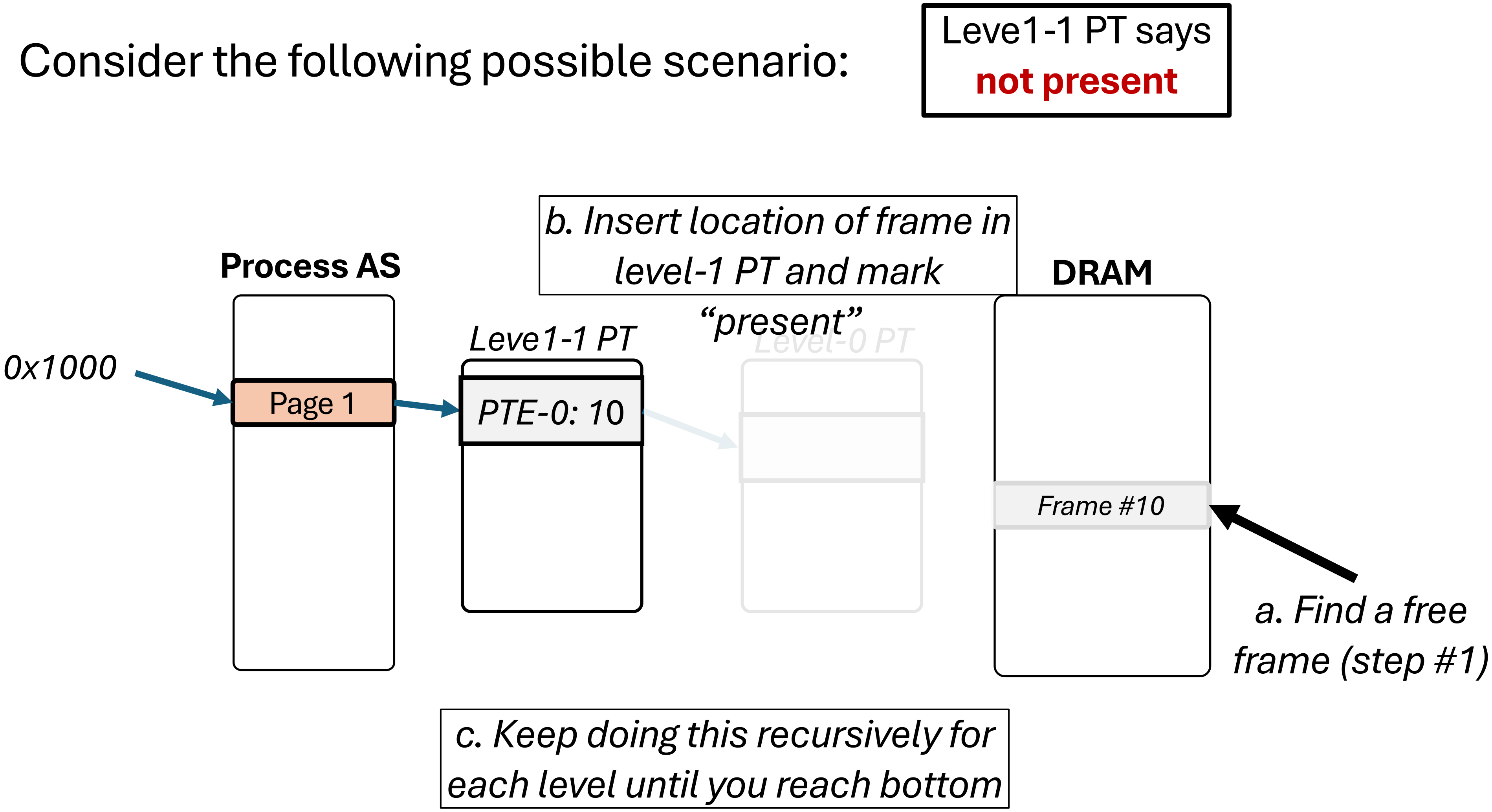
If the PTE is present and mapped, we say page is **resident** in memory

Step #1: find free frames

- When a computer boots up, all frames not used by the OS are **free**
  - OS initializes an internal allocator that **keeps track of the location** (i.e., physical address) of all free frames
- In Linux, if you want to get a free frame from the internal allocator:
  - `va_frame = get_zeroed_page(..)`
    - ➢ this will return the virtual address of a frame
- If you need the physical address, you can execute:
  - `__pa(va_frame)`

Step #2: Allocate a page table (if not allocated)

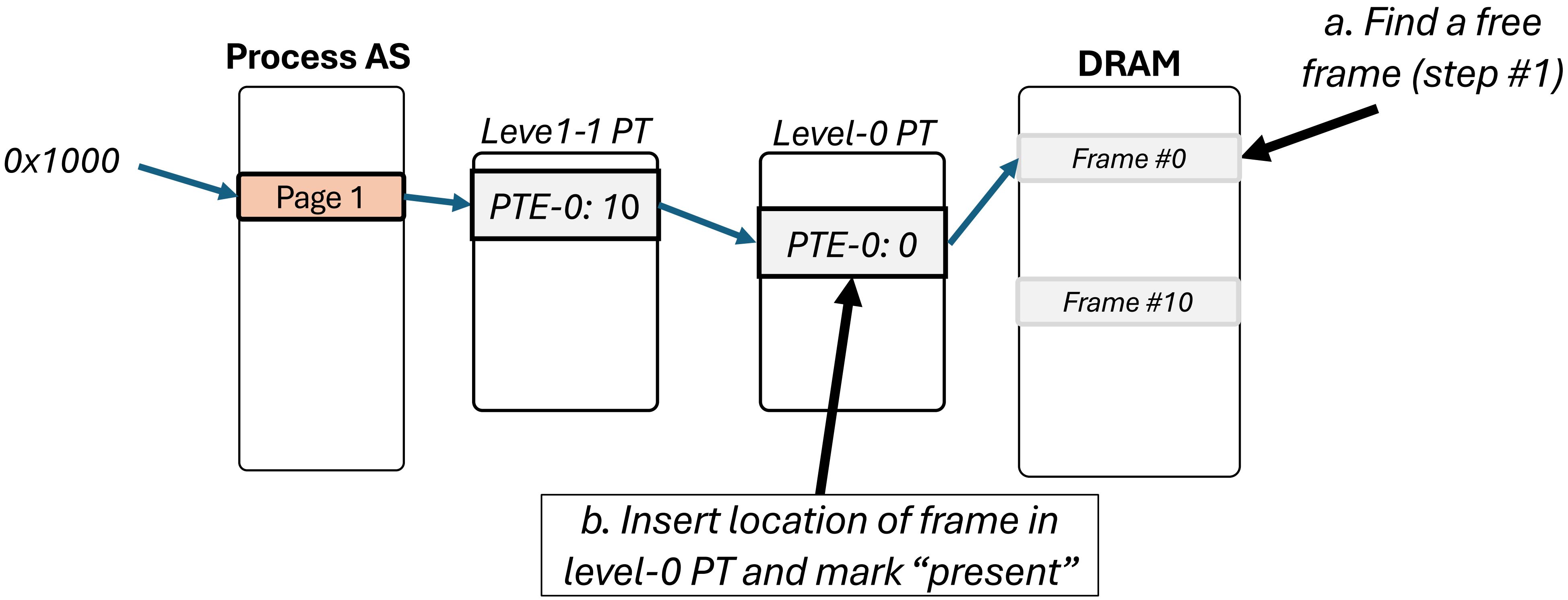
- Consider the following possible scenario:



Step #3: Allocate a page frame (if not allocated)

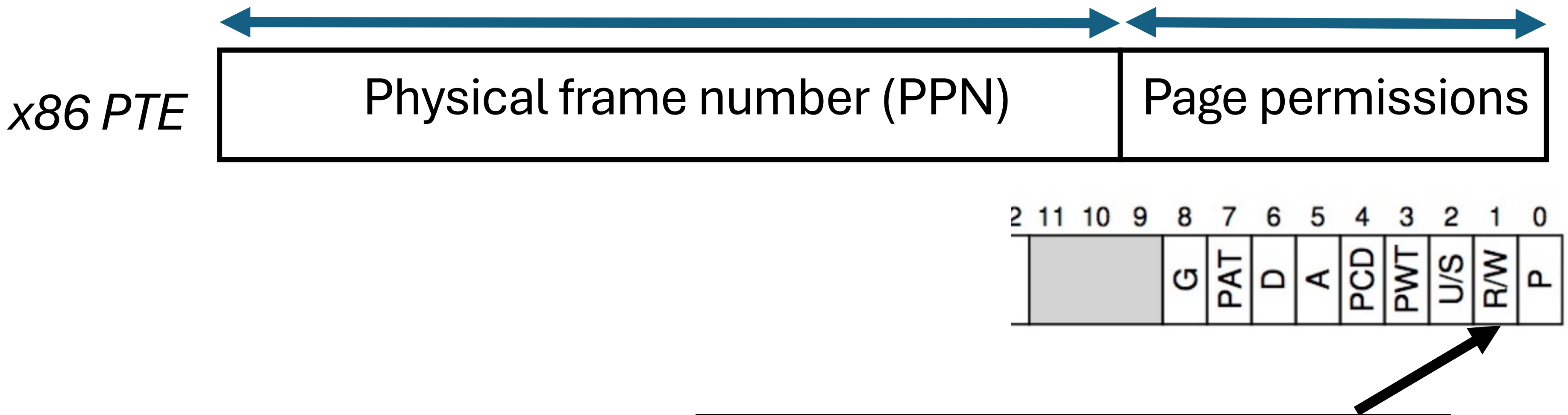
- Consider the following possible scenario:

Leve1-0 PT says **not present**



Step #3a: Implement permissions in the bottom PTE

- Recall mmap (0x1000, …, READ | WRITE, 4096, …)
- In the bottom-level page table, you can set these permissions



If bit is set to 0, only read allowed.
if bit is set to 1, both read/write allowed.

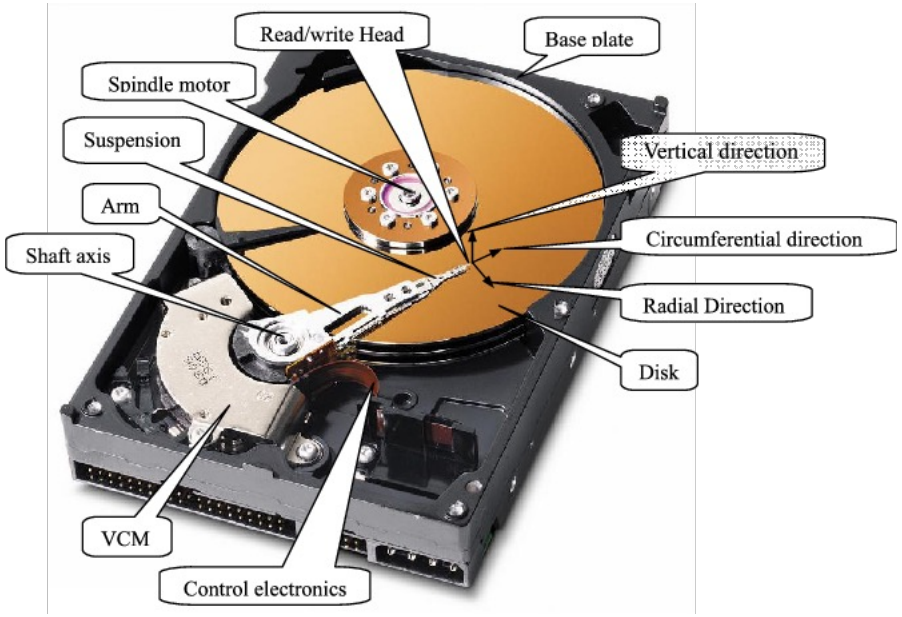
Step #4: Keep track of pages allocated to process

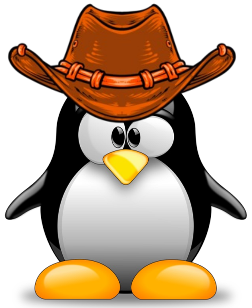
- Required for later housekeeping (e.g., reclaiming/freeing certain pages from the process' address space)
    - `munmap` system call

- Future allocation requests:
    - ➢ Recall step #1 had us walking the page tables
    - ➢ If we keep track of what pages are allocated, no need to walk anymore!

Project #4: *Build a custom `mmap` implementation!*

- At a high-level, follow steps #0 – step #4 inside a kernel module
  - **Please follow the ordering of steps mentioned in the document!!!!**
- Use the helper functions/code provided to you:
  - Traversing (or walking) page tables
  - Checking if a page table entry is "present"
  - Allocating new frames using internal allocators
  - Setting different read/write/execute bits within PTEs
- **Note:** this is a hard assignment. Please start early.





Quick recap of **HDDs** and their scheduling

Storage performance overview

- Disk performance depends significantly on the working mechanism
- The disk latency can be calculated by considering the three steps that the HDD must perform:
  - HDD I/O latency = Latency_{seek} + Latency_{rotate} + $\text{Latency}_{transfer}$
- Let's see how many milliseconds are taken by these *(seek, rotate, and transfer)* operations!

Examining **workload suitability** for HDDs

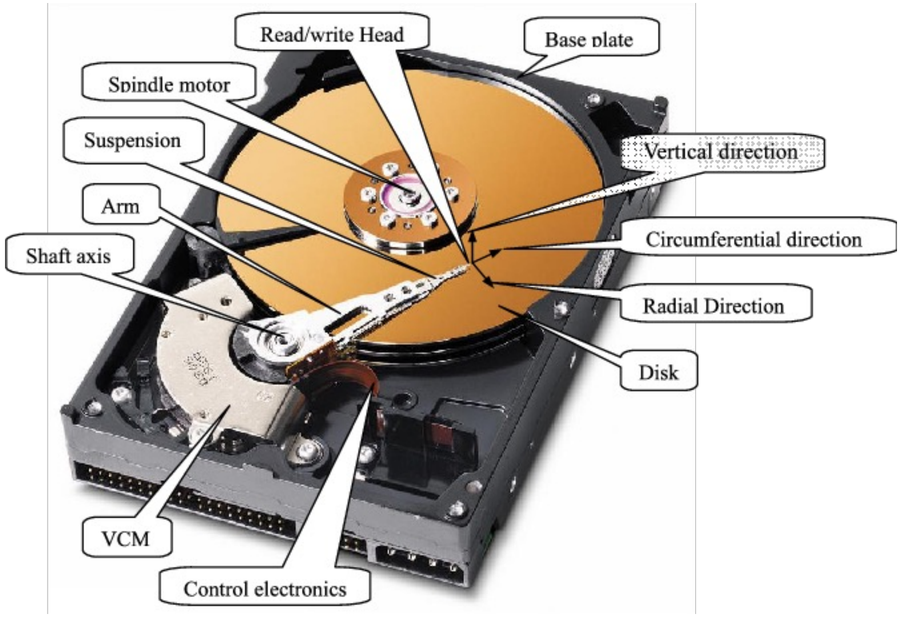
- Seeks and rotations are slow while transfer is relatively fast
  - seek = ~4 – 10 ms, rotate = ~4 – 5 ms, transfer = ~5 us
- **What kind of workload is best suited for disks?**
  - **Sequential I/O**: access sectors in order (transfer dominated)
- **Random** workloads access sectors in a random order (seek+rotation dominated); thus, slow on disks

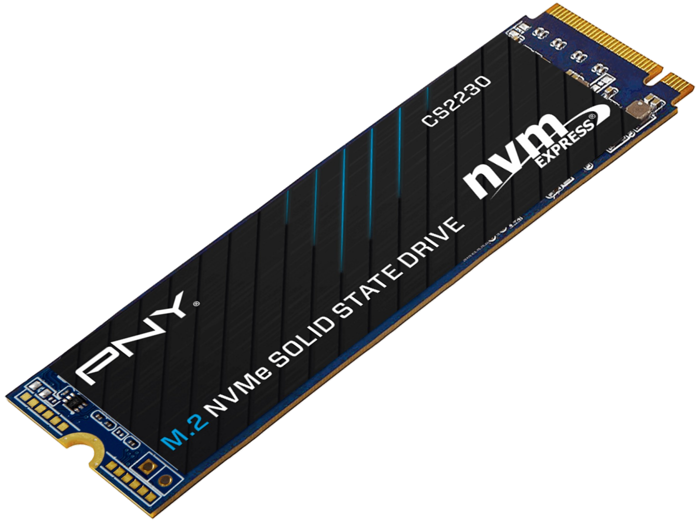
Disk scheduling introduction and requirements

- The OS must answer: “Given a stream of I/O requests, in what order should they be served?”
- Strategy: **reorder** requests to meet certain goals
  - Performance (e.g., by making I/O sequential)
  - Fairness
  - Consistent latency
- **Performance objective:** minimize seek + rotation time
  - Minimize the distance the head needs to go

Summary of HDD scheduling

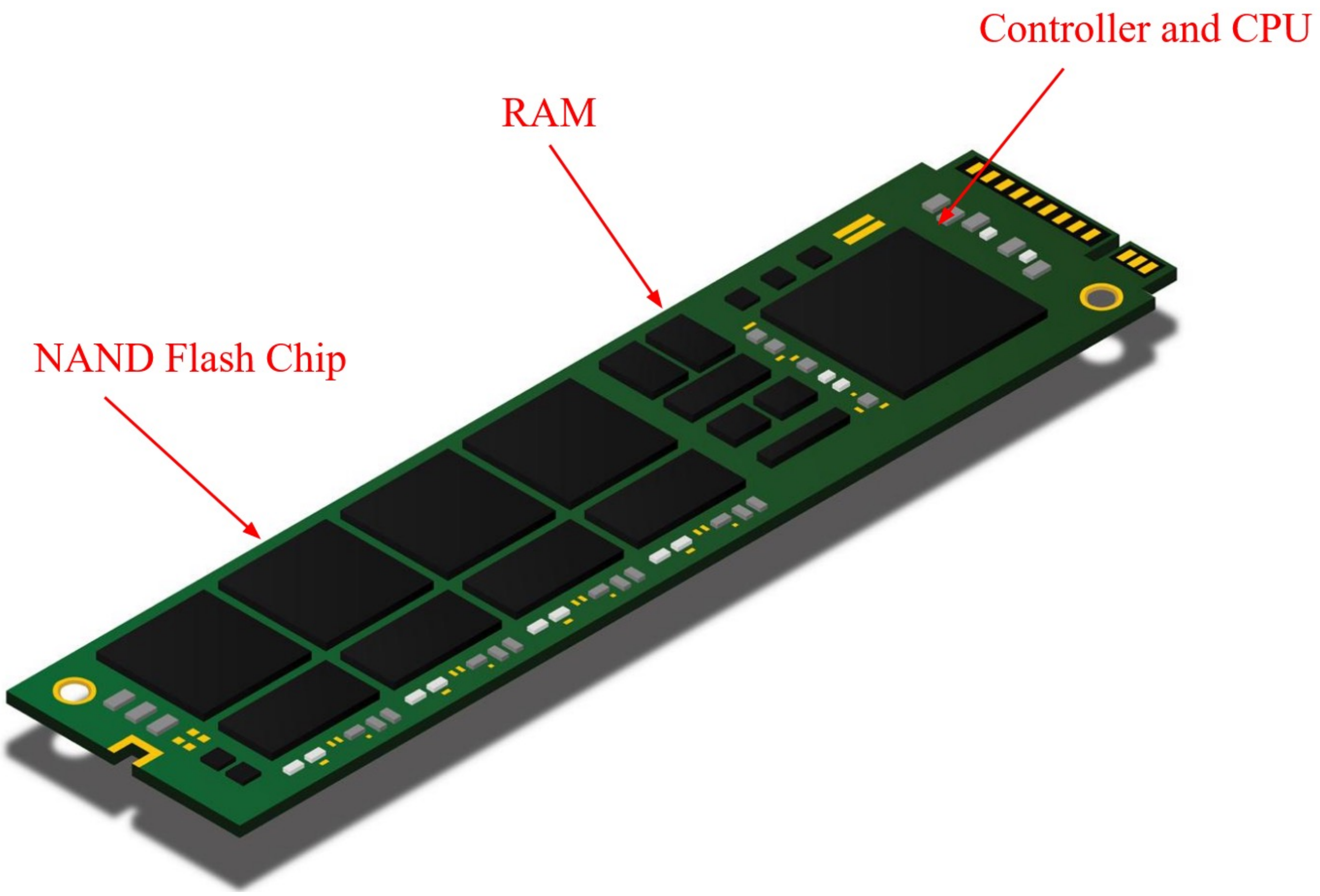
- Shortest Positioning Time First (SPTF) reduces seek and rotation
  - But, it leads to starvation
- SCAN, C-SCAN, and C-LOOK perform better for systems that place a heavy load on the disk
  - Also, provide better fairness
- Performance depends on the workload (i.e., number and types of requests)
- Linux leverages Completely Fair Queueing (CFQ)



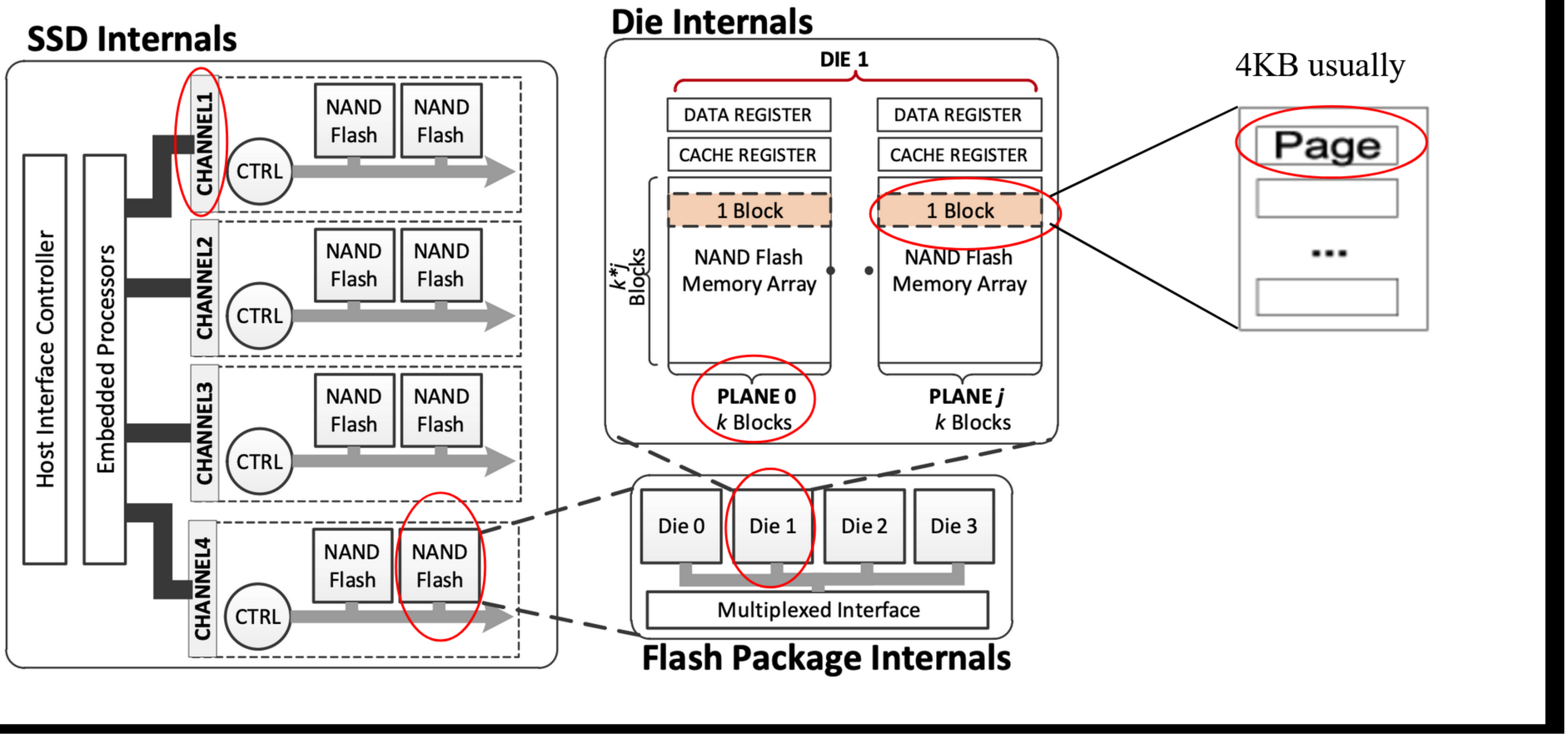


Comparing HDDs and SSDs

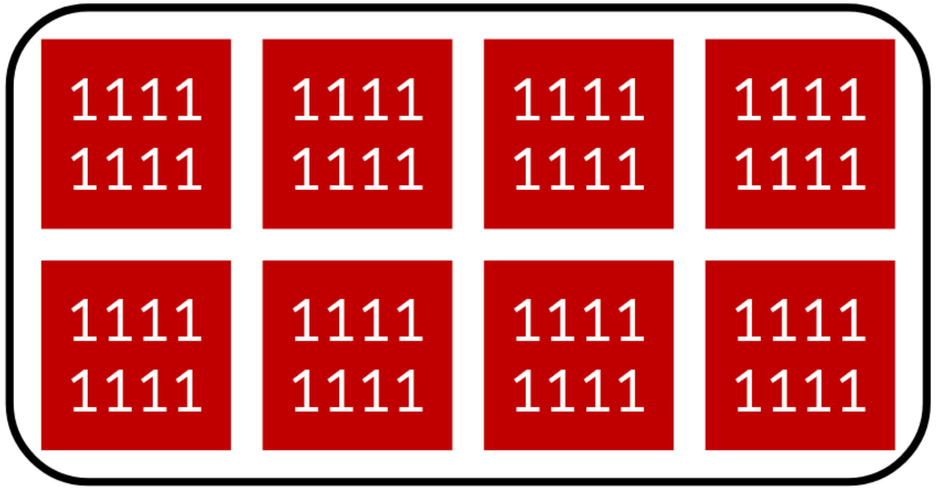
Solid State Drive (SSD) overview



Solid State Drive (SSD) Internals



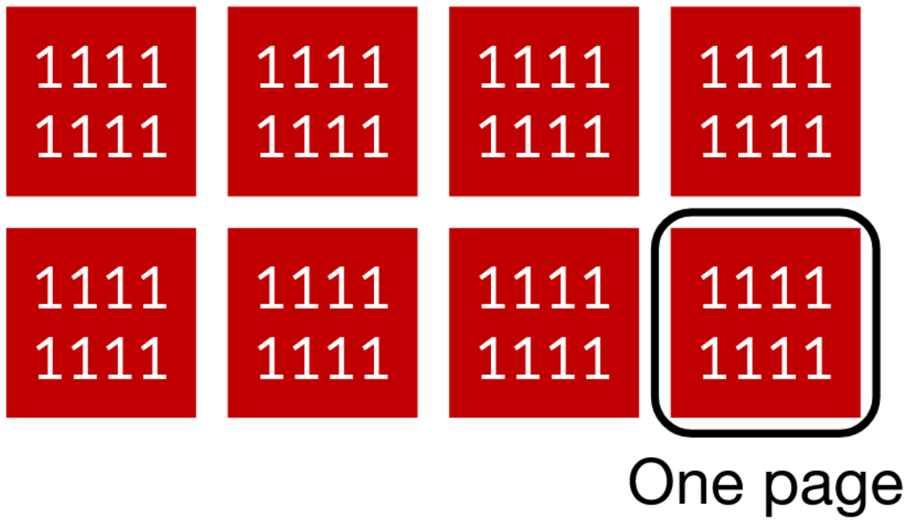
Blocks and pages



One block

- Each flash array is made up of many blocks and this one block

Blocks and pages



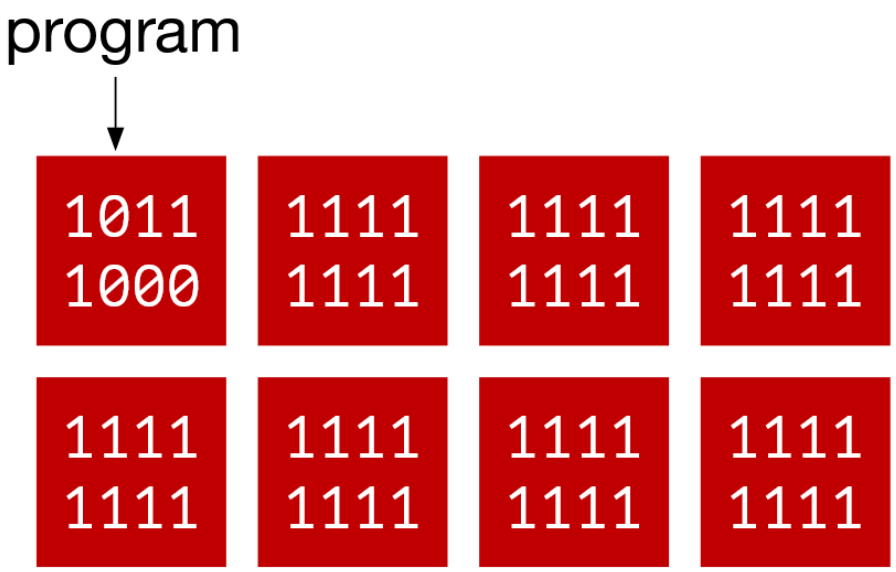
Writing to a flash region

- All "1" in a block → block is **clean** and ready to be used
- Data is **written** by changing the "1" into a "0" (called **program**)
  - Operation at "page-level"
- To remove data, you must write back "1" (called **erase**)
  - Operation at "block-level" (block → a set of pages)
- Hence, program is much faster than erase

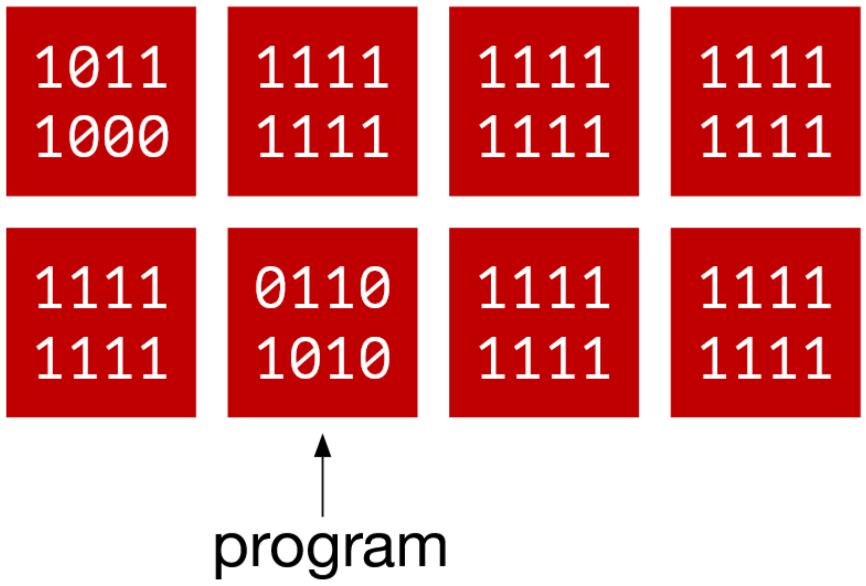
Writing to a flash region (illustration)

All pages are clean
(“programmable”)

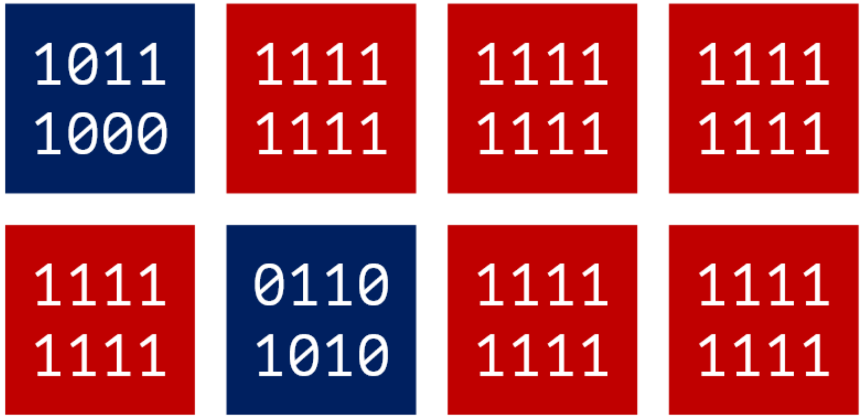
Writing to a flash region (illustration)



Writing to a flash region (illustration)



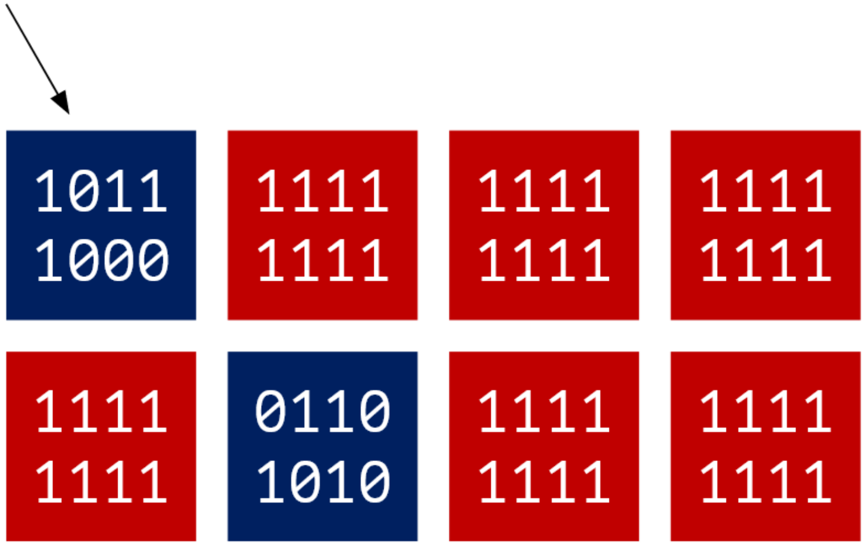
Writing to a flash region (illustration)



Two pages hold data
(**cannot be overwritten**)

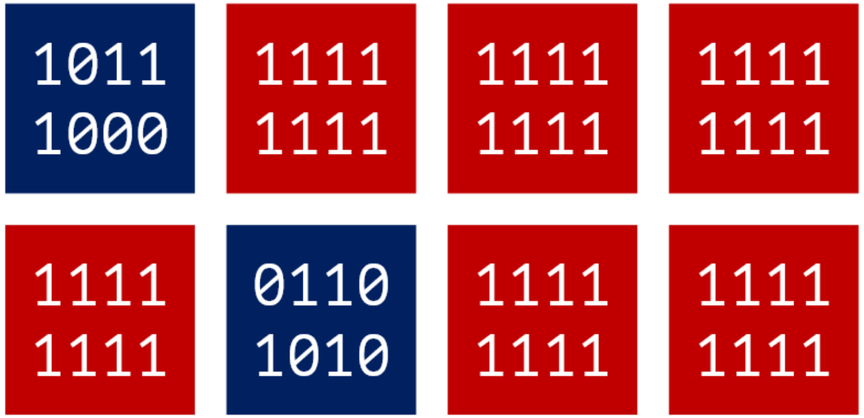
Writing to a flash region (illustration)

still want to write data into this page???



Two pages hold data
(**cannot be overwritten**)

Writing to a flash region (illustration)



erase

Writing to a flash region (illustration)

erase
(the whole block)

Writing to a flash region (illustration)

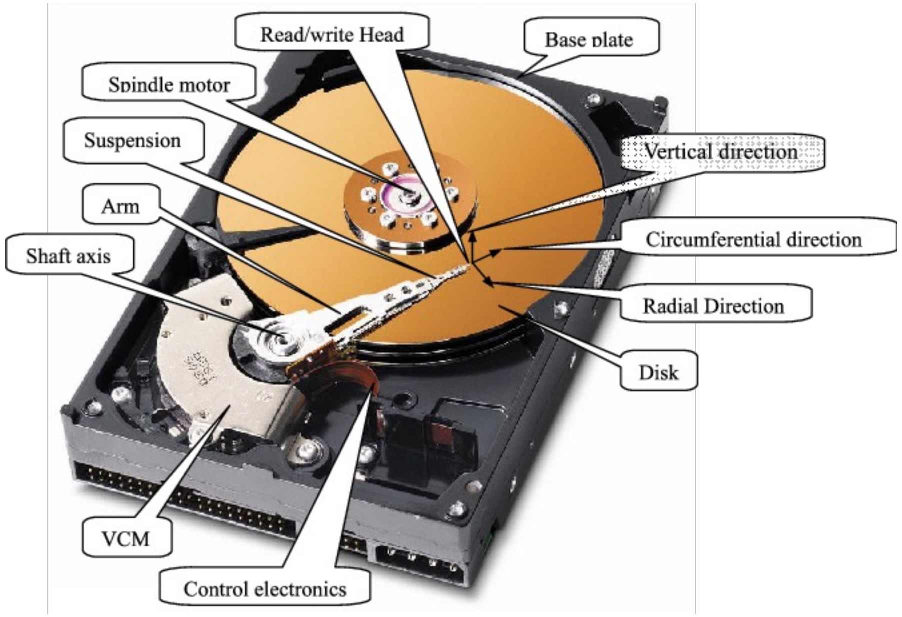
After erase, again, **free state**
(can write new data in any page)

Writing to a flash region (illustration)

This blue page holds data

Comparison of APIs OS must support for HDDs and SSDs

	disk	flash
read	read sector	read page
write	write sector	**program** page (0’s) **erase** block (1’s)



Device drivers

Device driver introduction

- Components of the kernel that support communication with different devices (e.g., HDDs, SSDs, Bluetooth, etc.)
- Typically, these can be installed on-demand during execution
- In Windows, device drivers are given a lower privilege (i.e., *ring-1/2*) than the kernel
  - Can access kernel memory but cannot execute privileged instructions

	disk	flash
read	read sector	read page
write	write sector	**program** page (0's) **erase** block (1's)

Device driver functionality broken into **two halves**

- *Top half:* executed when the device is called by the process (e.g., open)
  - Process executes system call (e.g., open)
  - OS starts I/O request and waits for response from the device; sleeps
  - Executed during the context of I/O requesting process

- *Bottom half:* executed when the device signals an interrupt
  - Device has completed request
  - Could happen during the context of any process
  - Device interrupt handler is called, and it wakes-up the I/O requesting process

- Let's check **virtio-disk.c** for in the **xv6 OS** as an example

The xv6 operating system

- We’ve seen some code from the xv6 OS before, and we will see some more today related to it’s block layer

- Let’s briefly introduce the OS first:
  - xv6 is a “toy” operating system designed for teaching OSs
  - designed by professors at MIT in early 2005

- Great OS if you’re interested in tinkering/hacking kernels

xv6 code: virtio_disk_rw (in kernel/virtio-disk.c)

Top-half execution of the virtio device driver during process' context

```
void
virtio_disk_rw(struct buf *b, int write)
{
  uint64 sector = b->blockno * (BSIZE / 512);

  acquire(&disk.vdisk_lock);
```

…

```
  __sync_synchronize();

  *R(VIRTIO_MMIO_QUEUE_NOTIFY) = 0; // value is queue number

  // Wait for virtio_disk_intr() to say request has finished.
  while(b->disk == 1) {
    sleep(b, &disk.vdisk_lock);
  }
```

Executed on FS read/write call

Notify the device using MMIO register

Go to sleep while waiting for response

xv6 code: virtio_disk_intr (in kernel/virtio-disk.c)

Bottom-half execution of the virtio device driver during any context

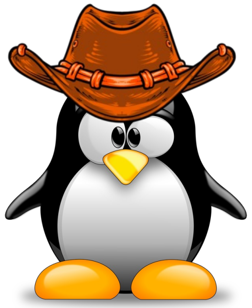
```
void
virtio_disk_intr()
{
  acquire(&disk.vdisk_lock);

  // the device won't raise another interrupt until we tell it
  // we've seen this interrupt, which the following line does.
  // this may race with the device writing new entries to
  // the "used" ring, in which case we may process the new
  // completion entries in this interrupt, and have nothing to do
  // in the next interrupt, which is harmless.
  *R(VIRTIO_MMIO_INTERRUPT_ACK) = *R(VIRTIO_MMIO_INTERRUPT_STATUS) & 0x3;
```

Executed in response to interrupt

Acknowledge interrupt but nothing else

```
    struct buf *b = disk.info[id].b;
    b->disk = 0;   // disk is done with buf
    wakeup(b);
```

Wake-up requesting process and ask it to handle response



Moving to **higher level** abstractions

Can anyone tell me a problem with having different interfaces?

	disk	flash
read	read sector	read page
write	write sector	**program** page (0's) **erase** block (1's)

- Each new storage disk could have its own R/W mechanism
- OS developers would have to cater individually to the specific APIs of a device

This adds **complexity** to using disks for any general task (e.g., writing a file system)

What's the solution to this "many interface" problem?

	disk	flash
read	read sector	read page
write	write sector	**program** page (0's) **erase** block (1's)

Create a **unified interface** and translate all operations from different interfaces to the unified one

Visualizing a **unified interface** for storage access

	disk	flash
read	read sector	read page
write	write sector	**program** page (0’s) **erase** block (1’s)

block_read(*device*, *location*, *data*)

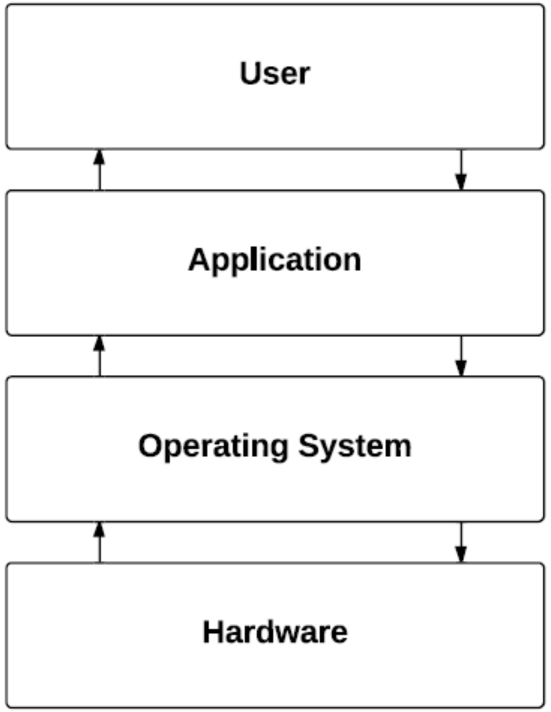
```
hdd_write:
        seek();
        rotate();
        transfer();
```

```
ssd_write:
        If (erase) erase_block();
        program(page, data);
```

block_write(*device*, *block/sector*, *data*)

Once again, OS does the plumbing in computers

Provides an abstraction for the messy hardware



- Manipulating hardware requires understanding of the hardware internals

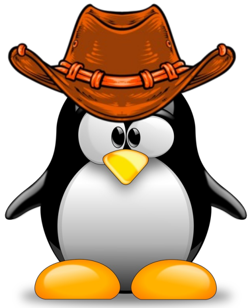
- Abstractions simplify using the hardware and hide away all these messy details

The block abstraction layer

- We want storage devices to allow **random** access of sectors or *blocks* (typically 512 or 4096 bytes)

- Simplify accessing all these devices with a block-level abstraction
  - Higher-level OS code can directly interact with blocks, and the lower-level device driver can support that interaction

- Generally, two operations:
  - bread(device, x, buffer) → read from device at sector X into buffer
  - bwrite(device, y, buffer) → write buffer into sector Y of device



Questions? Otherwise, see you next class!