



# Intro to Lab 3

05/06/21



# Misc

- Lab 3 is out
  - Due the 17th
  - Design doc due 10th



# Today's Agenda

- High level intro for Lab 3 - Address Management
  - Take a look into ``vspace.c`` functions you'll need on your own
- Next week - deeper dive into `vspace` structs and functions



# Part 1: User Level Heap

- User-level programs use **malloc** and **free** to manage heap memory
  - Track list of the free blocks in memory
  - **Malloc**: Return a free block of memory somewhere in the heap
  - **Free**: Free a block of memory somewhere on the heap, so it can be reused
  - **Calloc**: Malloc, but zeros out the block before returning it
  - We've given you malloc/free in **user/umalloc.c**
    - Or you could paste in your implementation from 351 (actually, please don't)
- But that's not everything
  - What happens if we run out of memory on the heap for malloc to use?
  - Malloc deals with virtual memory, but we need to map it to physical data

---

# sbrk

(“set program break”)

Get more heap space



## sbrk(n)

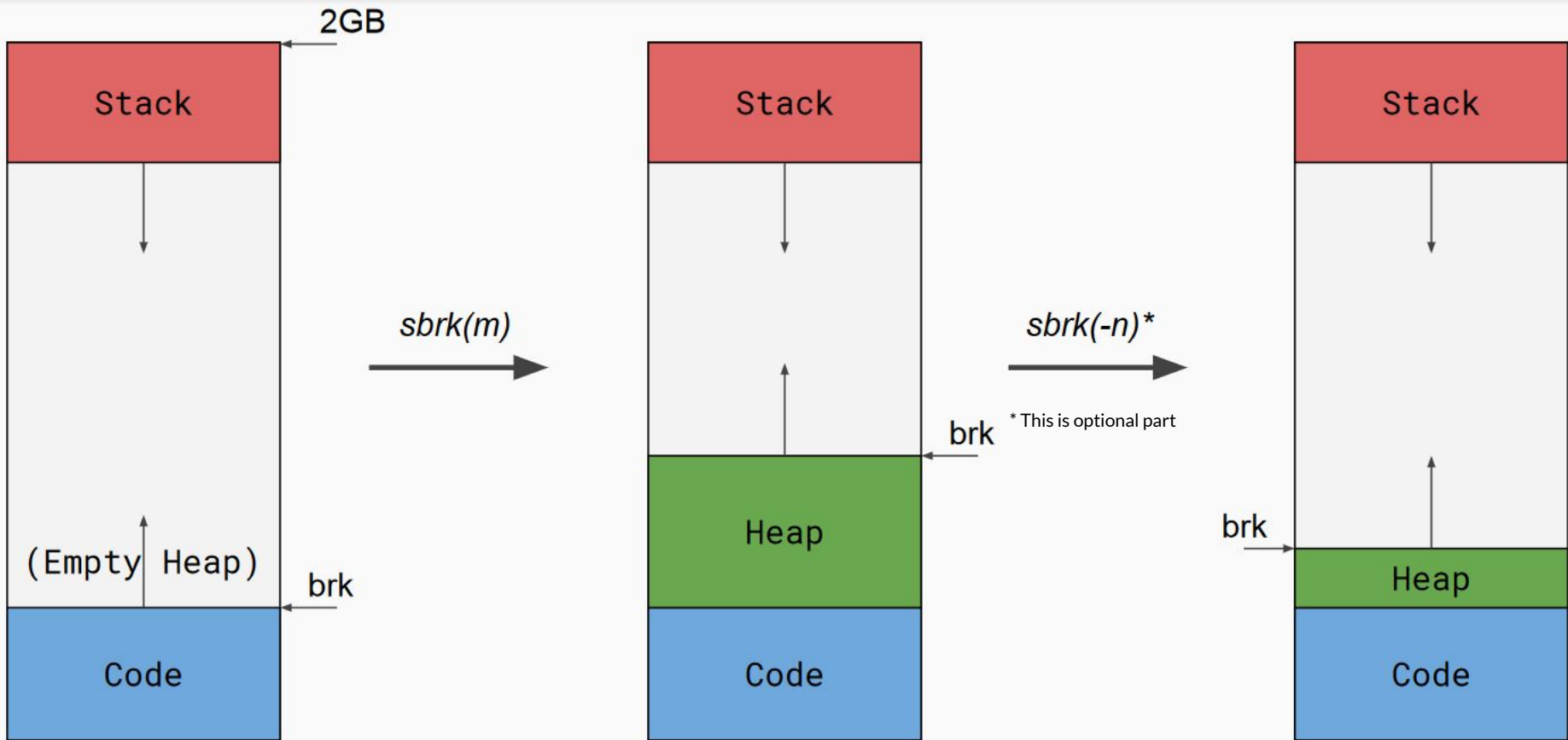
Increase the size of the heap by  $n$  bytes, updating the “program break”

Program break = max space allocated to the heap data segment

$N$  can be negative in a real system, but that doesn't matter for xk

Returns -1 if it can't allocate enough space

Otherwise, return the *previous* heap limit (the old top of the heap)





## Sbrk details

- `Sbrk` should allocate at **byte** granularity (may need to allocate 0, 1, or multiple new pages)
- New pages need to be properly mapped into the process' address space
- Use `vspaceaddmap` to get new pages

Once implemented, malloc/free tests should pass



---

# The Shell

Running multiple programs



# Shell

- The initial process (user/init.c) will fork to create a shell
- The shell will take user input and run commands, spawning other programs
  - Just like bash, cmd, or whatever else
- Shell will spawn other programs
- You can use the shell to pipe things
  - e.g. `ls | wc` will pipe the output of `ls` into the input of `wc`
  - Since fd 0/1/2 are always in/out/err, we can change a process file table entry to be a pipe (when forking, before exec)



**Let it Grow**

more stack



# The Stack

- The initial version of exec is pretty simple
  - It fixed the stack size at one page
- One page of stack probably isn't enough for larger programs
- We want to be able to add more pages of stack.

How do we tell when more stack needs to be added?

- Once we've written off the end of what's currently allocated
  - PAGE FAULT!
  - Add more pages and resume user-level execution
- For simplicity, you can assume the stack will never need more than 10 pages
  - If page fault and address > stack\_base - 10: grow stack; else: "normal" page fault (can't handle)



# Grow Stack on Demand

- The initial version of exec is pretty simple
  - It fixed the stack size at one page
- One page of stack probably isn't enough for larger programs
- We want to be able to add more pages of stack.

How do we tell when more stack needs to be added?



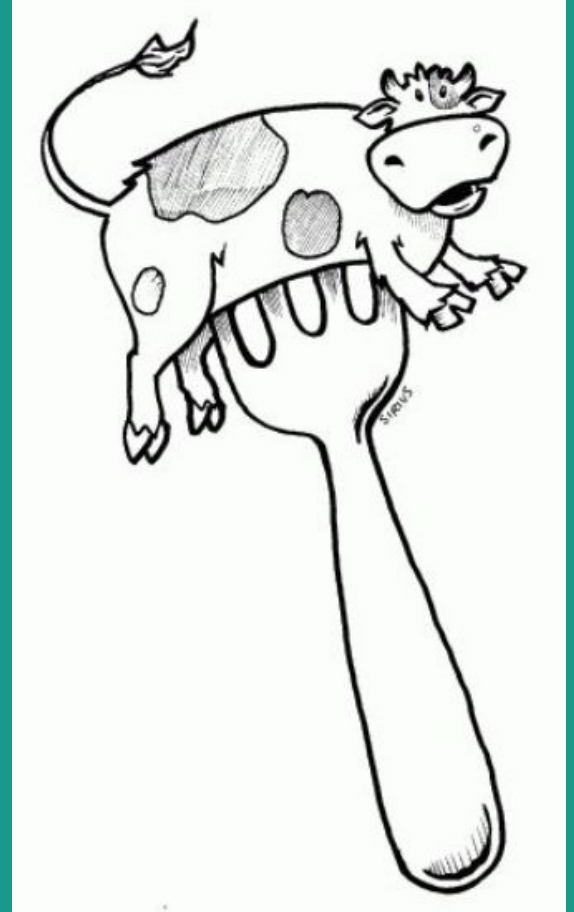
# Grow Stack on Demand

- Once we've written off the end of what's currently allocated
  - PAGE FAULT!
  - Add more pages and resume user-level execution
- For simplicity, you can assume the stack will never need more than 10 pages
  - If page fault and address > stack\_base - 10: grow stack; else: "normal" page fault (can't handle)
- Functions similarly to sbrk
  - Adding pages to particular region, but signal is different

---

# COW Fork

(beef utensils)





# Copy on Write Fork

The lab 2 fork implementation is inefficient.

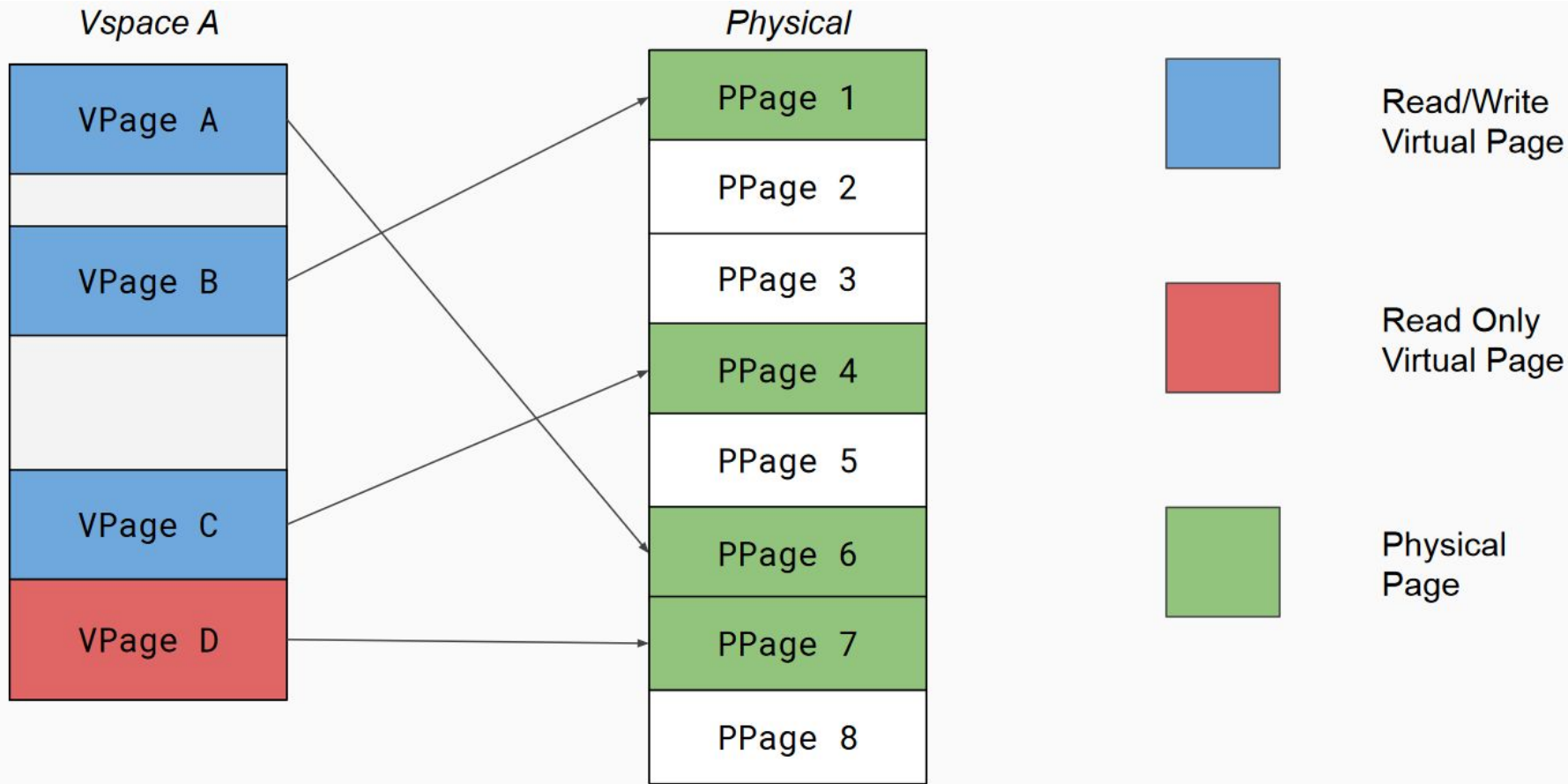
**Vspacecopy** completely memcpys all pages, so the parent and child have disjoint page tables

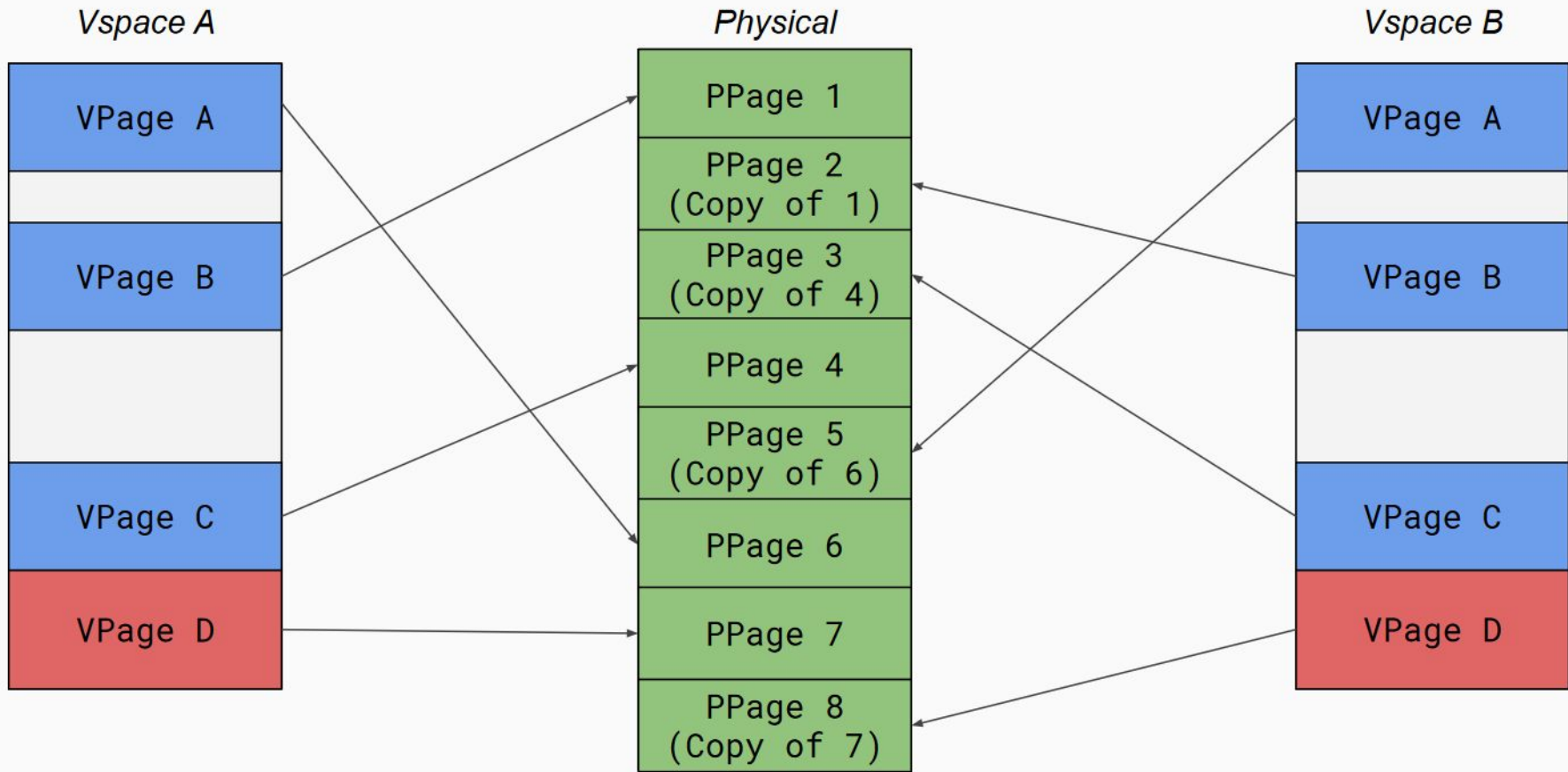
As a consequence:

- Child and parent duplicate the same unchanging pages of code and static data
- If we fork and exec, we waste time copying all pages before immediately discarding the vspace

How would we optimize this to reduce the memory footprint of processes?





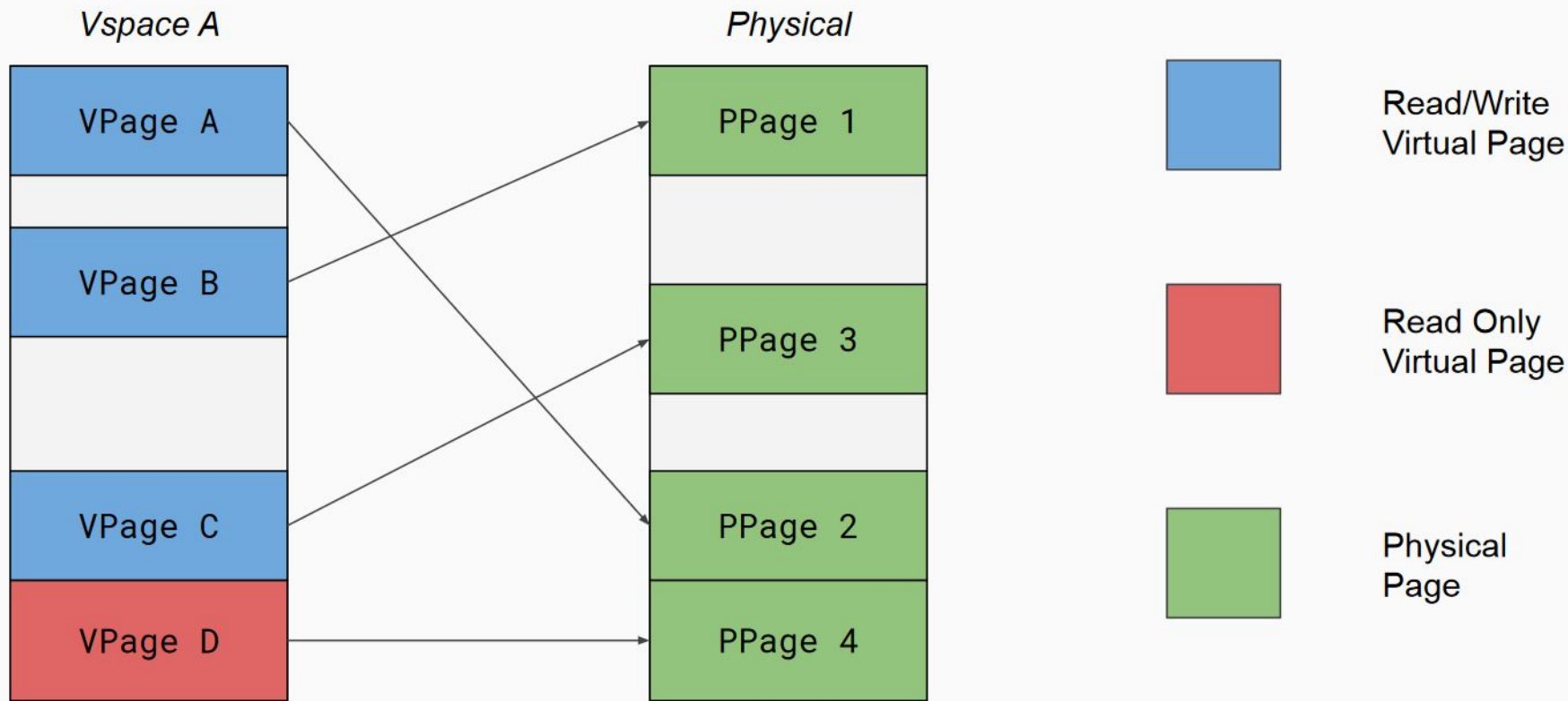


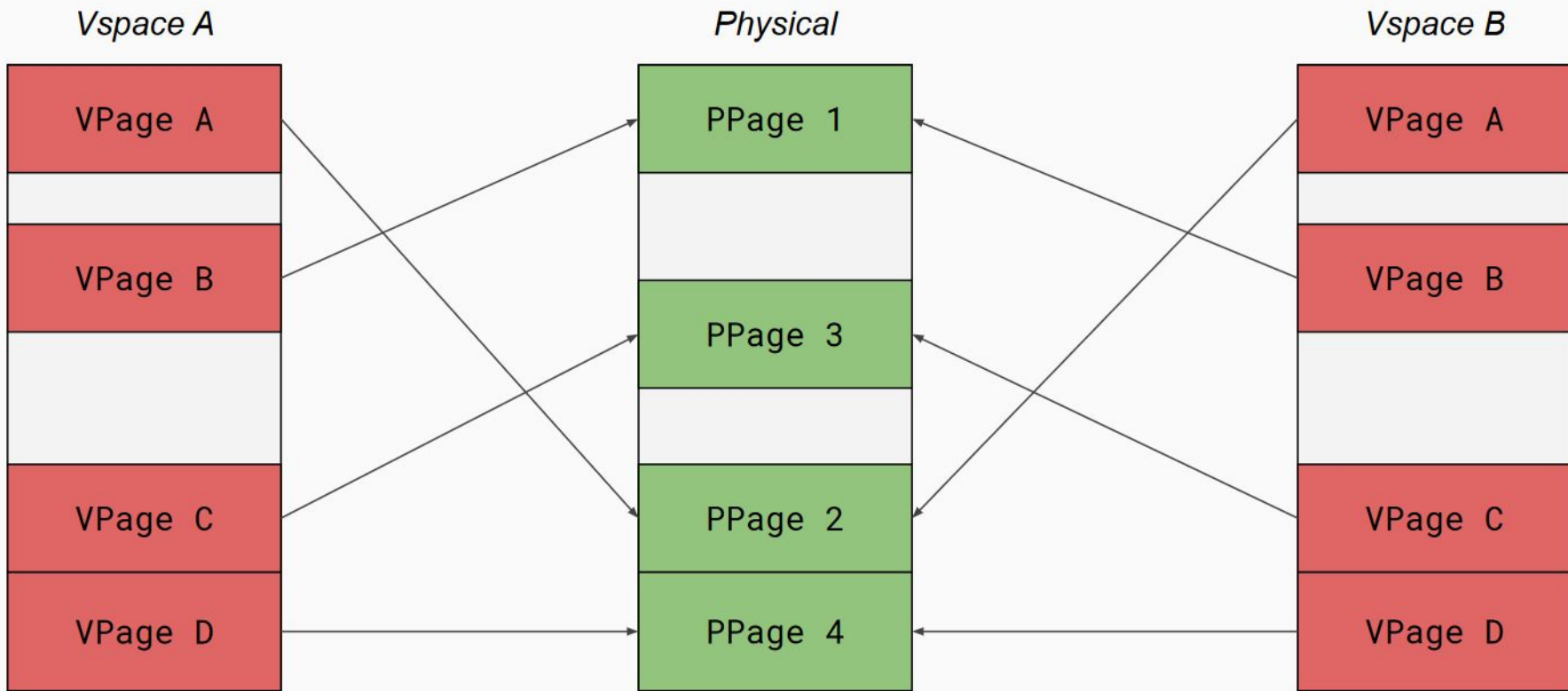


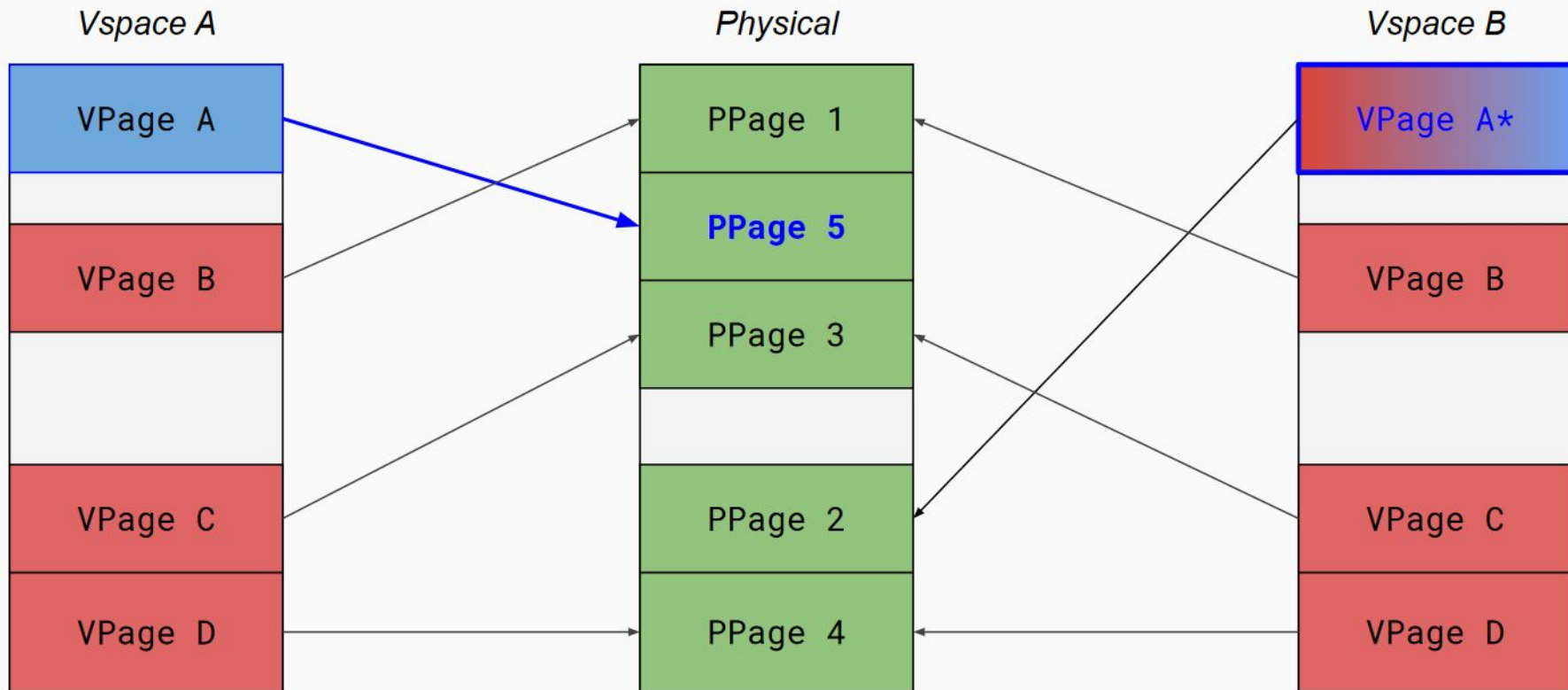
# We copied all the pages :(

What can we do better?

- Don't actually copy pages
- Clone the page table and set everything to read-only
  - Both processes can reference the same data
  - Don't actually copy pages
- Only when we need to write to a page should we duplicate it
  - Most of the time, we won't write to a page again, so no copying needs to be done







VPage A in B could be safely set to writable (if there are no other vpsace pointers to PP2).  
When VSpaceB::A gets set back to writable is an implementation detail



# Things to Consider

1. How do you distinguish a copy-on-write read-only page from a normal read-only page?
2. Make sure that no memory is leaked
  - a. If the reference count of a physical COW page is 1, the process with that reference can reclaim it as non-COW
3. What happens (specifically) when a process tries to write to a COW page?
  - a. For starters, a tried-to-write-to-read-only page fault
4. Synchronization is necessary
  - a. Parent and child could try to write at the same time
5. A child with COW pages can fork to create another child
  - a. Parent, child, grandchild, etc. all referencing the same physical pages



# Design Doc

- You got your feedback for lab 2 design.
- How did your design end up different from your implementation?
- How's that going to change how you do lab3 design?

Thoughts, feedback?