

Section 5: Intro to Lab 3

CSE 451 19sp



Part 1: Create a User-Level Heap

- User level programs call **malloc** and **free** to manage heap memory
 - Free list keeps track of free blocks in heap
 - **malloc** - Returns a free block of memory in the heap
 - **free**- Frees a block of memory in the heap
 - **calloc**- Like malloc, but zeros out memory first
 - We have provided malloc and free for you in *user/umalloc.c*
 - Or you can copy your implementation from 351 (just kidding, please don't)
- But what happens when there is no space left in the heap for **malloc** to return???

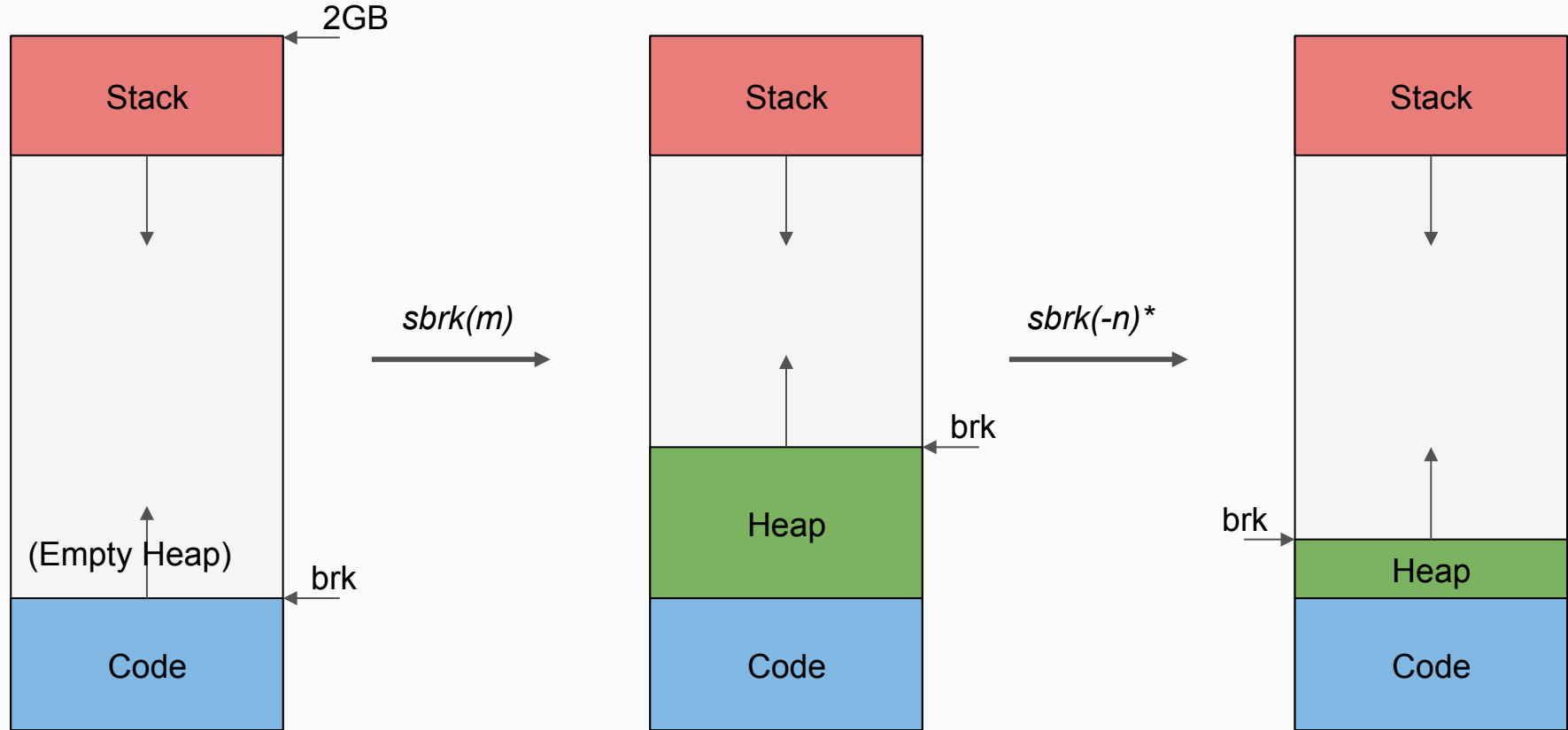
sbrk (set program break)

Hey Kernel, give me more heap space!

sbrk(n)

- Increments the Heap by n bytes, resetting the *program break*
 - Program break determines the max space that can be allocated to the data segment, where the heap lies
- Returns -1 if there is not enough space
- Otherwise, returns the previous heap limit (i.e. the *old* top of the heap)

sbrk(n) Visual Diagram



* Note that you don't need to support negative increments for Lab 3!

shell

All I do is fork fork fork no matter what!

Part 2: Starting Shell

- You'll be adding init (**user/init.c**) process that forks off a shell
- Shell will spawn other programs
- Try piping in the shell
 - E.g. `ls | wc`

Stack On Demand

(dynamic stack growth)

User: sub \$0x30, %rsp

Kernel: Stack Attack Alert! Stack Attack Alert!

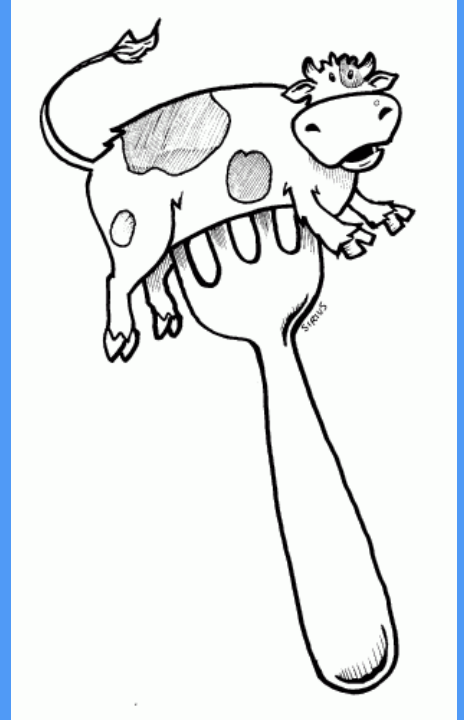
Part 3: On-Demand Stack Growth

- **exec()** fixed the stack size but we want to support stack growth
- What exception occurs when a user reads/writes to an unallocated part of the stack?
- What limits are there?

COW Fork

(copy-on-write)

Stop! Wait a minute! I might not even write there!



Part 4: Copy-on-write Fork

- What are some inefficiencies with our lab 2 fork implementation?

Discuss amongst yourselves.

Hint: Look at the comment for `vspacecopy`.

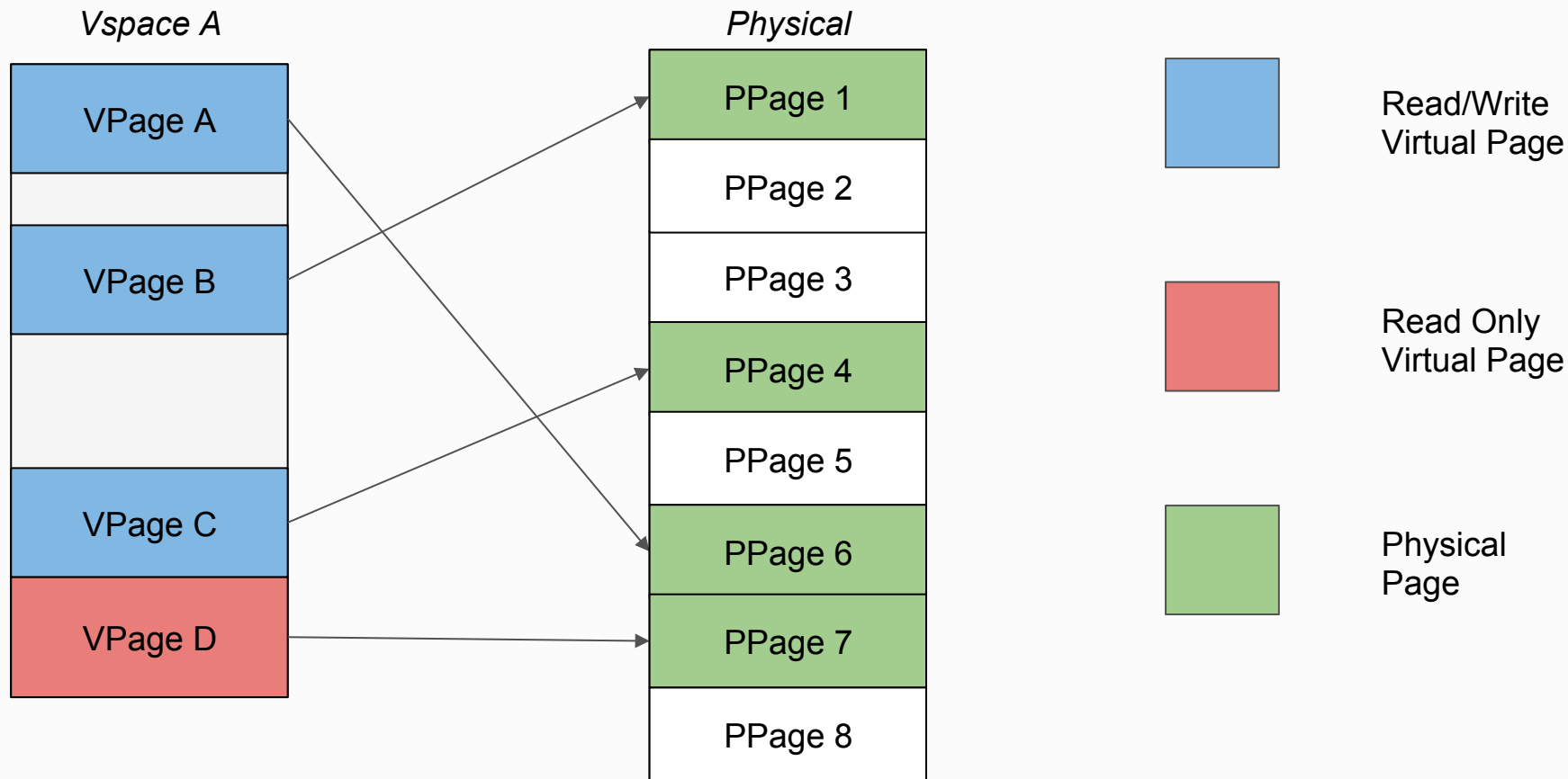
Part 4: Copy-on-write Fork

In lab2's fork, the mapped pages for the same data are **disjoint!** As a consequence:

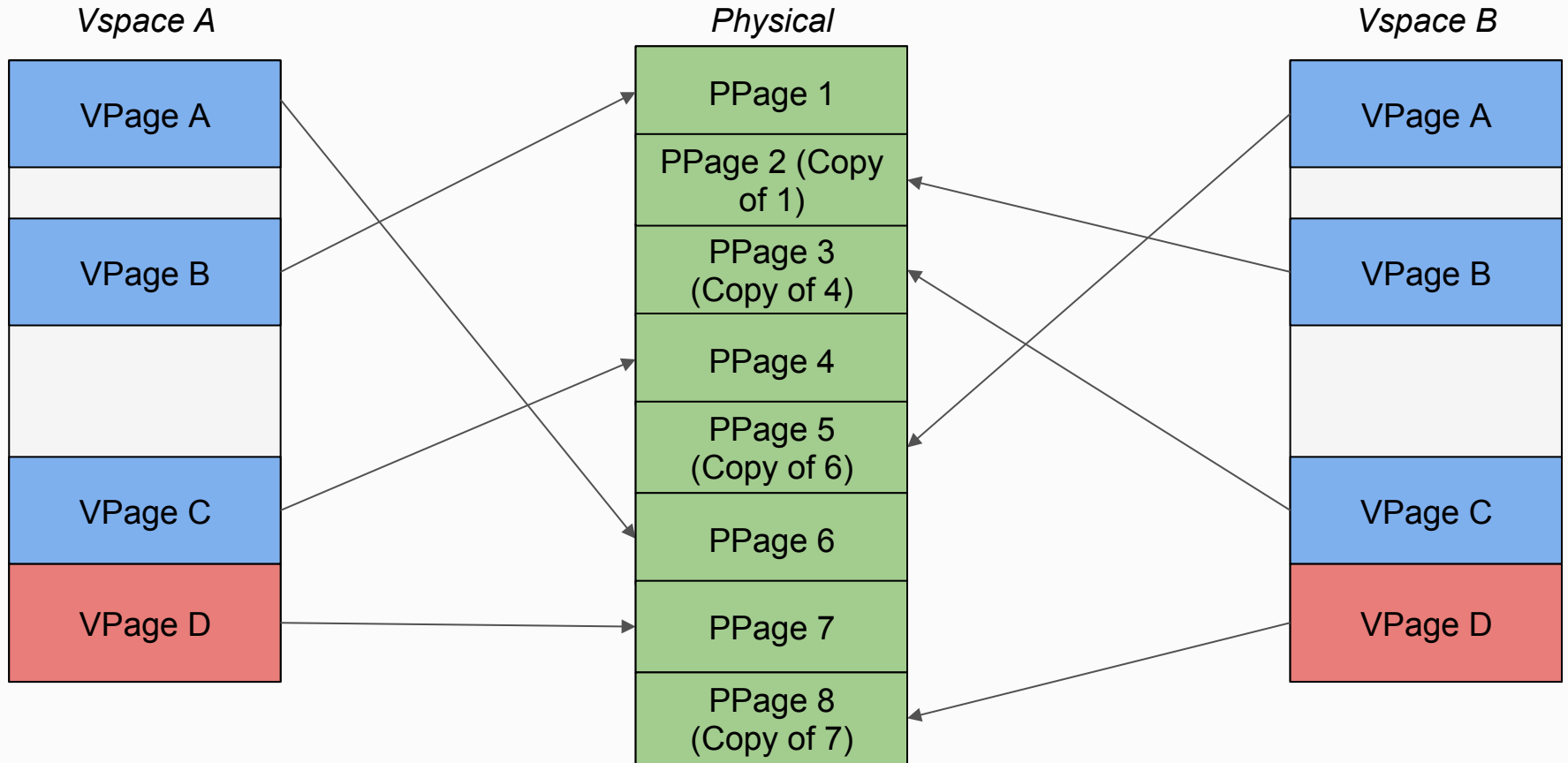
- Child and Parent use multiple physical pages for the **same unchanging code!**
- If child does **exec()**, we throw away the vspace copy created in **fork()**!

How might we address these issues? What are some cases we'll have to design for?

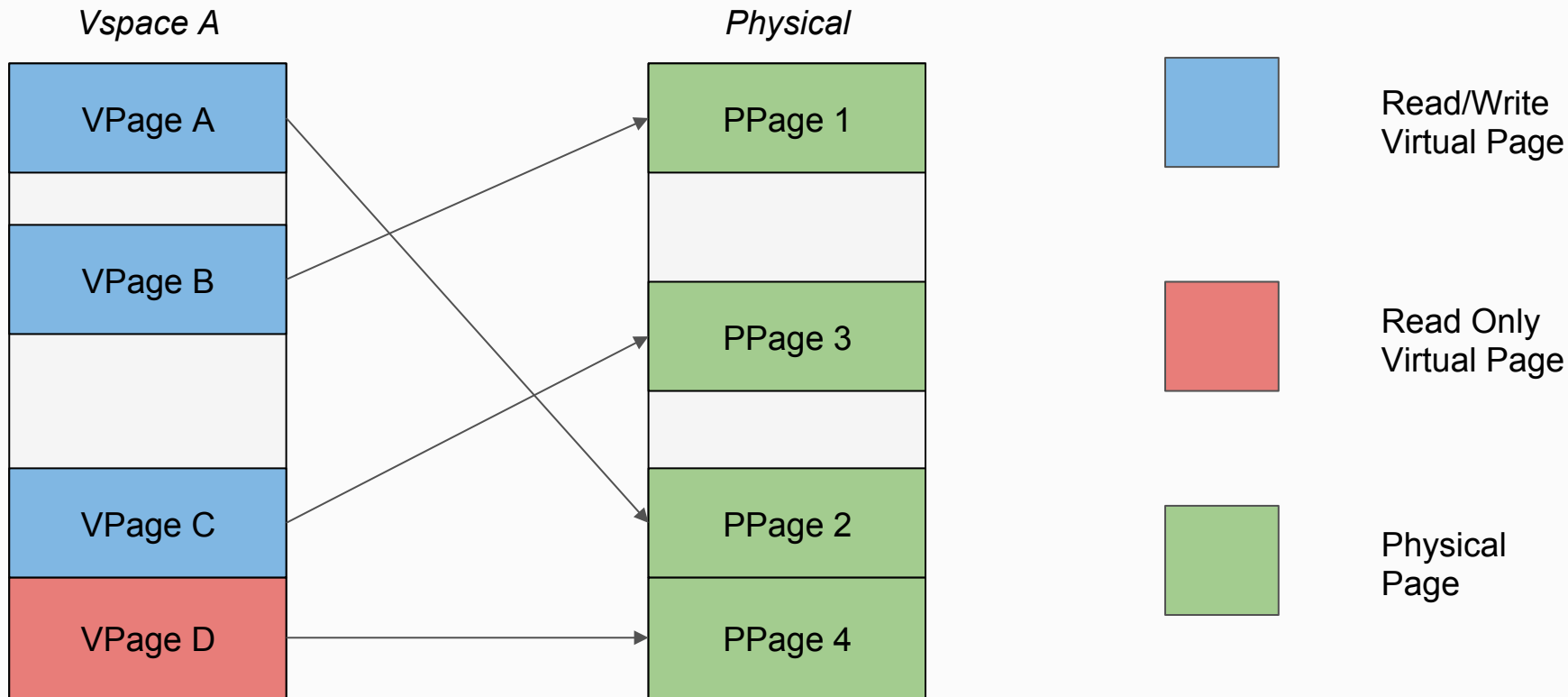
Lab 2 Fork Visual Diagram before fork()



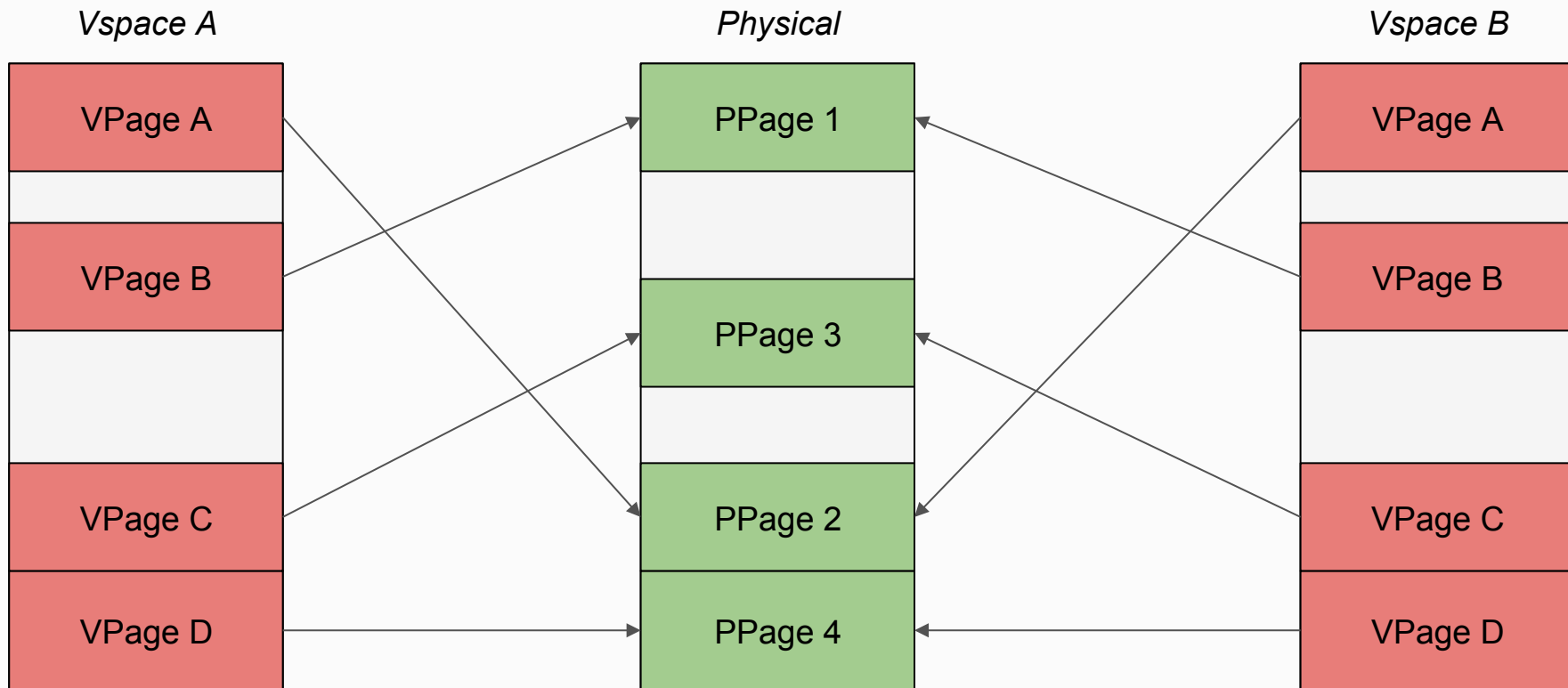
Lab 2 Fork Visual Diagram after fork()



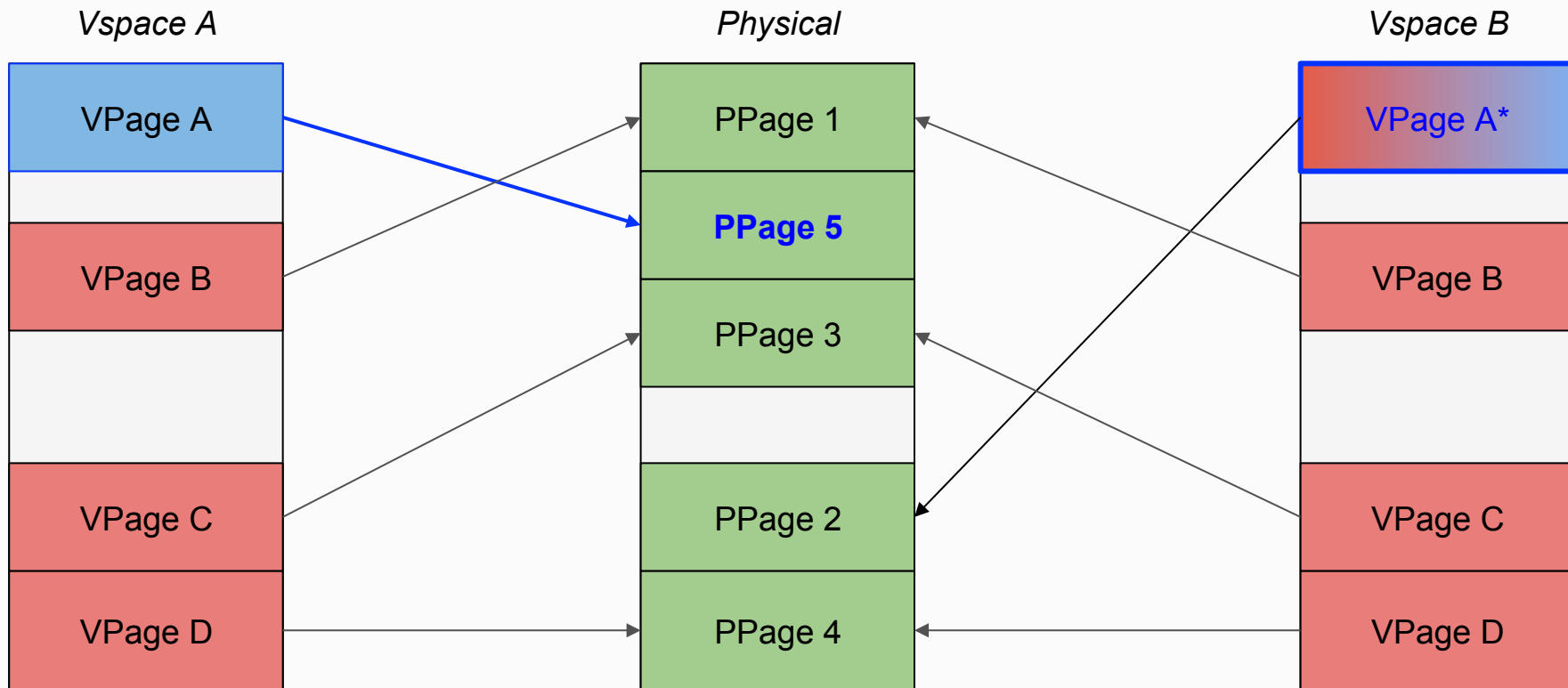
COW Fork Visual Diagram before a copy-on-write fork()



COW Fork Visual Diagram after a copy-on-write fork()



COW Fork Visual Diagram once Process A writes to VPage A



* Note: If Vspace B is the last reference, it makes sense to make its mapping writable too, but you might not want to do that if there are multiple read-only mappings from other vspace.

Part 4: Copy-on-write Fork

- Food For Thought
 - How to distinguish a copy-on-write page from a normal read-only page?
 - What happens when parent and child try to concurrently write to the same page?
 - Could the same physical page be mapped in more than two address spaces?
 - How to resolve the case when one process writes to a COW page?

Design Doc Feedback

- How did your implementation differ from your design?
- Thoughts and feedback?