Section 6: Intro to Lab 3

section 6: 2/13/2020
Please pick up section handout as you come in :)

Announcements

- Lab3 design doc due tomorrow
  - Thoughts on design docs?
- Lab 3 due next Friday
- Lab2 needs to work, submit your lab2 if you haven't done so

Page faults

- A trap number 14 means a page fault
- this means that the memory address accessed is
    - not mapped
    - or the access protection is violated (write to read-only page).

Data structures

- memregion
    - Keeps track of information for a continuous range of virtual addresses
    - Not a part of page table: just for bookkeeping inside the OS
- vpmap
    - Contains the actual page table

Stack On Demand

(dynamic stack growth)

User: `sub $0x30, %rsp`

Kernel: Stack Attack Alert! Stack Attack Alert!

Part 1: Grow user stack on-demand

- **`setup_stack()`** fixed the stack size but we want to support stack growth
- Step 1: update valid range for stack memregion (10 pages from USTACK_UPPERBOUND)
- Step 2: change the page fault handler to deal with valid page faults
    - as_find_memregion() to identify which memory region owns this page
    - pmem_alloc() to allocate a physical page
    - vpmap_map() to map the fault address with the allocated physical page
    - vm.h: helper functions to check permission bits

Part 1: Grow user stack on-demand

Questions for thought:

- Can the kernel cause a page fault that was meant for stack growth?
- Write some C user level code that causes a page fault for stack growth.

sbrk (set program break)

Hey Kernel, give me more heap space!

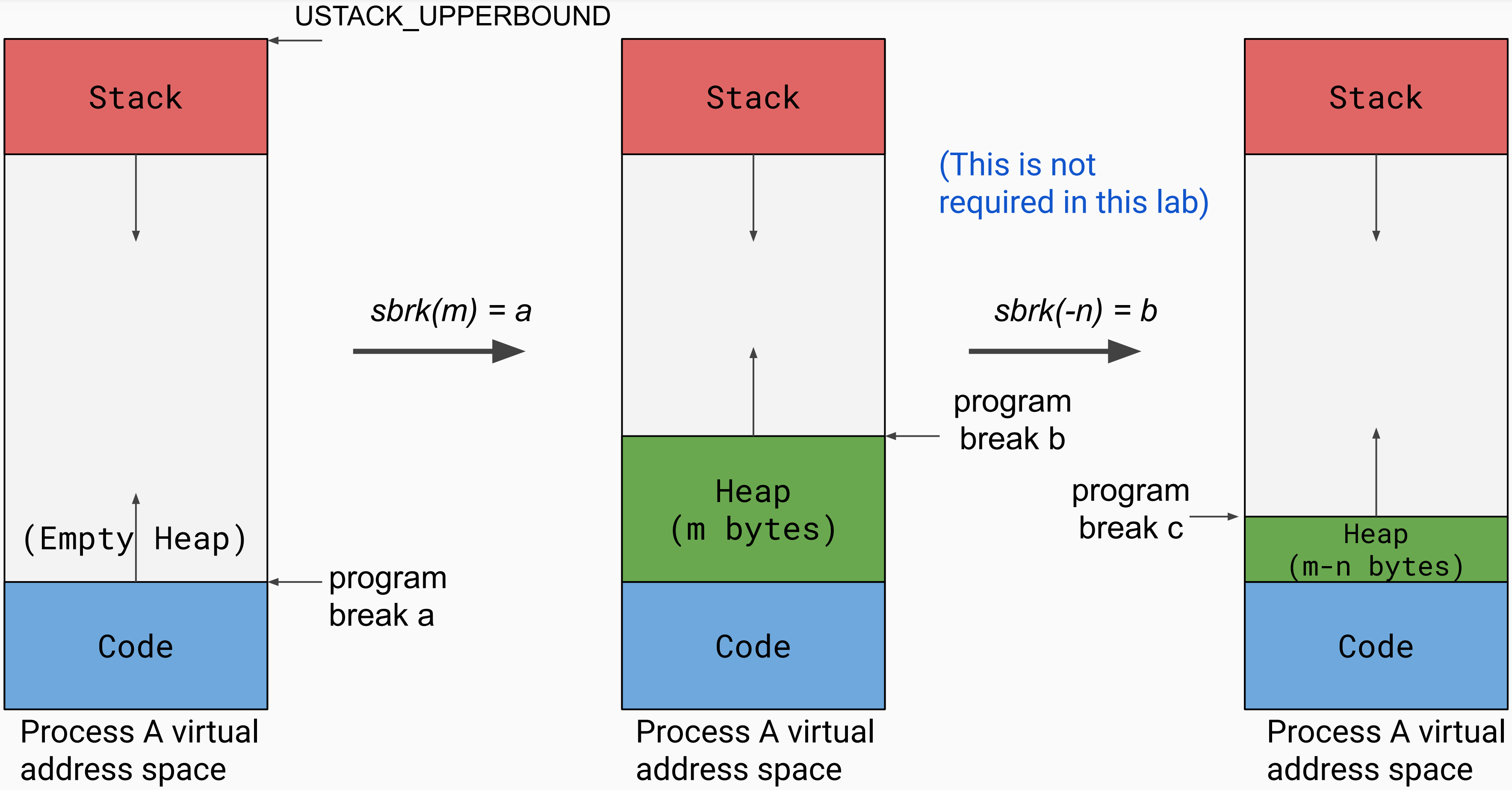
Part 2: Create a User-Level Heap

- 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
    - We have provided malloc and free for you in *lib/malloc.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(n)

- Increment/decrement 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 ERR_NOMEM if there is not enough space
- Otherwise, returns the previous heap limit (i.e. the *old* top of the heap)

sbrk(n) Visual Diagram



sbrk(n)

- Implement memregion_extend:
    - Extend the memory region, but don't allocate pages for now. We use on-demand allocation, similar to stack
- Hint: each address space has a pointer to heap memregion
- Once you implement memregion_extend, on demand allocation of heap pages is similar to on demand stack allocation
    - In fact, you can reuse your code
- page fault => validate if fault address is in a valid memregion => if so allocate, else terminates the process

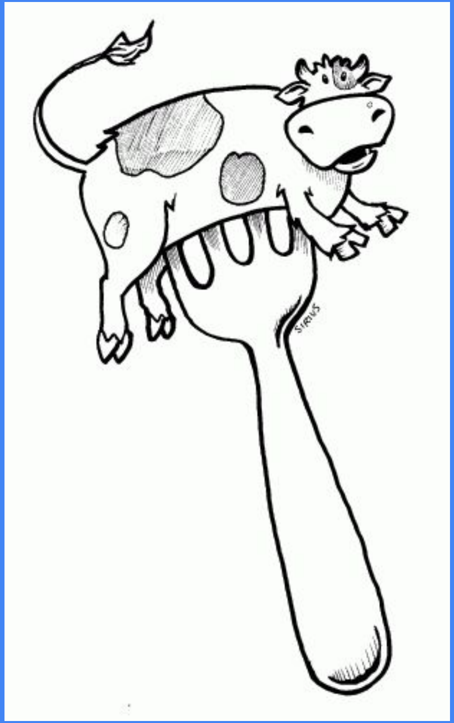
sbrk(n)

- Section handout: heap
- sbrk byte granularity allocation vs virtual memory page granularity mapping
    - Note that as_find_memregion will round the end address (see source code)

COW Fork

(copy-on-write)

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



Part 3: Copy-on-write Fork

- What is the most expensive operation in our lab 2 fork implementation?

Discuss amongst yourselves.

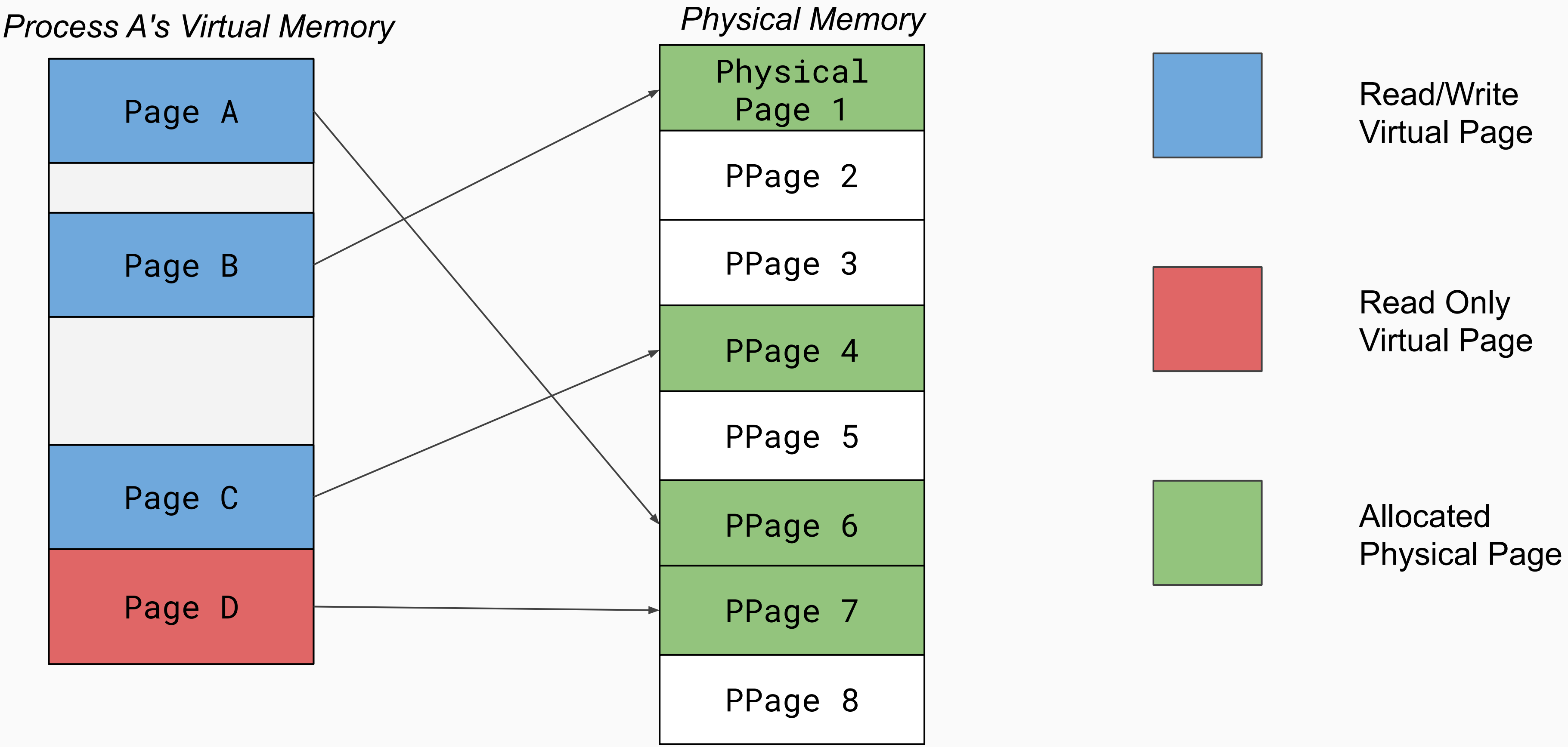
Part 3: Copy-on-write Fork

In lab2's fork, child gets a deep copy of parent's address space:

- Child and parent have different physical pages for the **same code**!
- If we implement exec(), we would throw away all copied pages 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()



Process A's Virtual Memory

Page A

Page B

Page C

Page D

Physical Memory

Physical Page 1

PPage 2

PPage 3

PPage 4

PPage 5

PPage 6

PPage 7

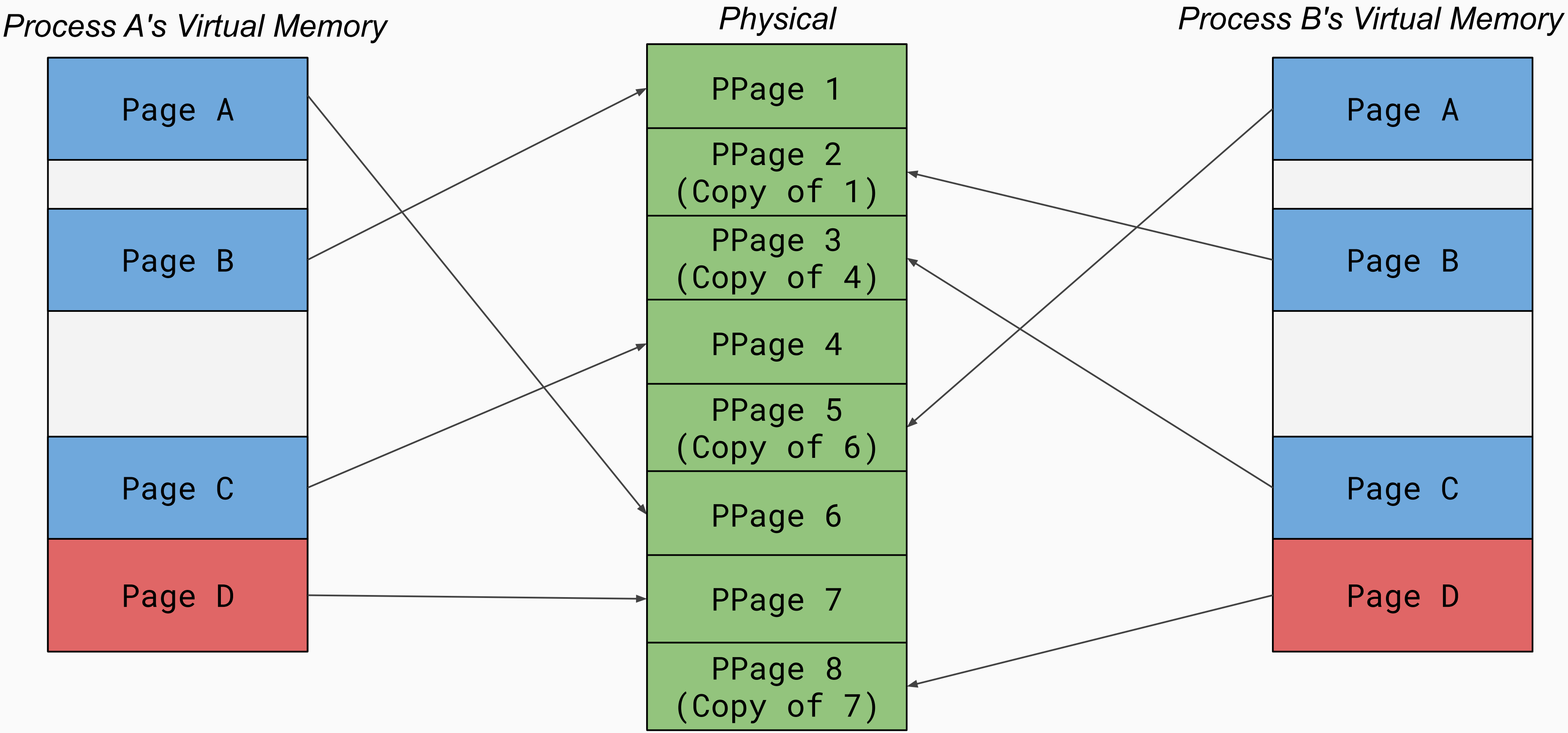
PPage 8

🟦 Read/Write Virtual Page

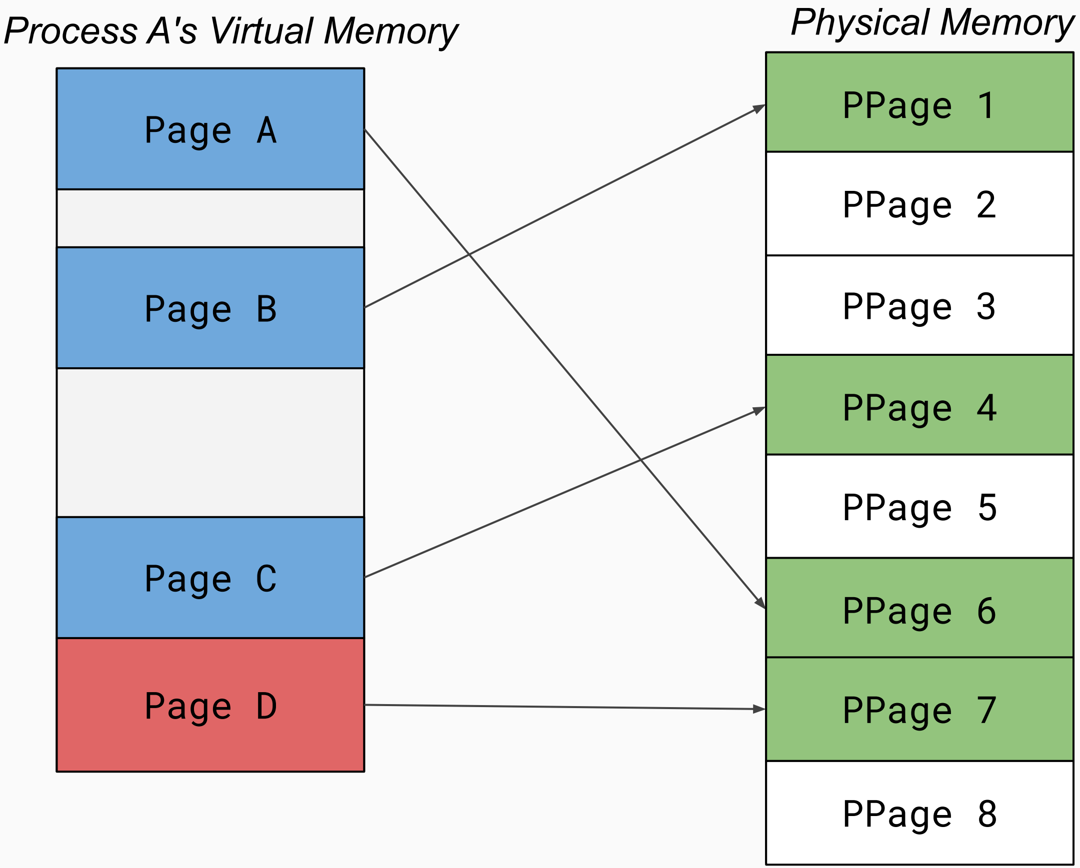
🟥 Read Only Virtual Page

🟩 Allocated Physical Page

Lab 2 Fork Visual Diagram after fork()



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



Process A's Virtual Memory

Page A

Page B

Page C

Page D

Physical Memory

PPage 1

PPage 2

PPage 3

PPage 4

PPage 5

PPage 6

PPage 7

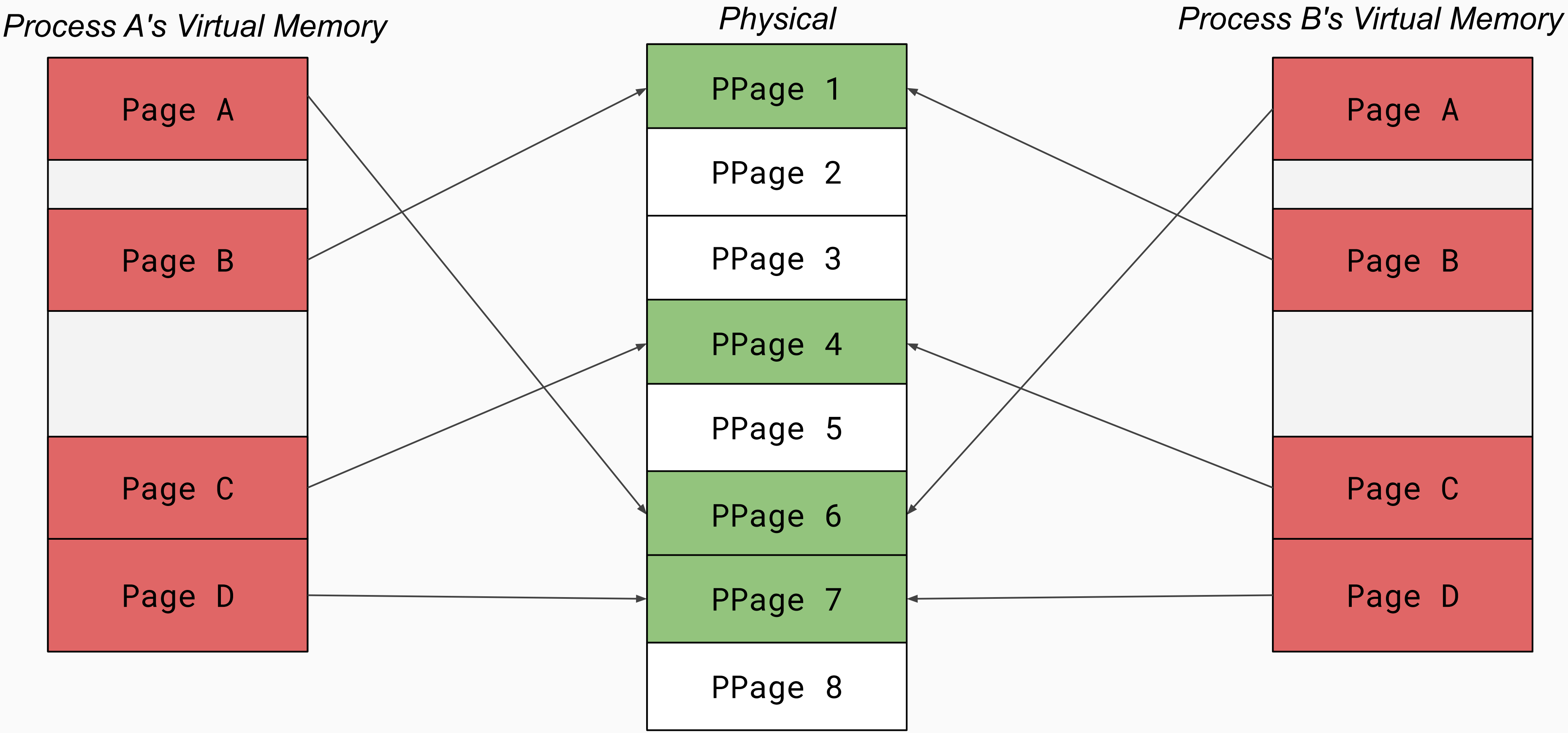
PPage 8

🟦 Read/Write Virtual Page

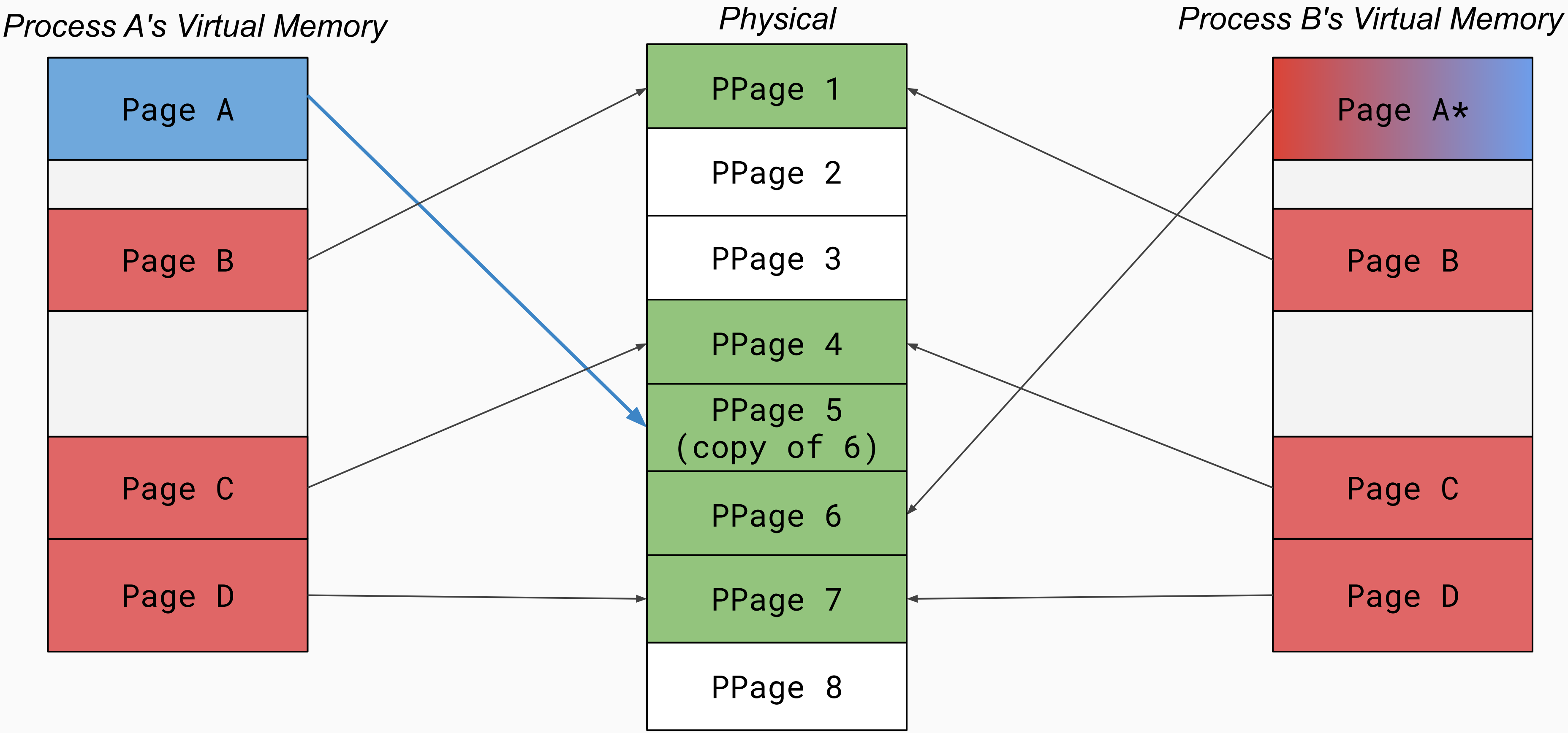
🟥 Read Only Virtual Page

🟩 Allocated Physical Page

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



COW Fork Visual Diagram once Process A writes to Page A



* Note: If Process B is the last reference of ppage 6, you can make it writable when it tries to write to it (instead of making a copy of 6)

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 the last process writes to a COW page?
- When should we use vpmap_flush_tlb() to flush TLB cache?