Lab 3 Intro

Memory Management

Reminder

- Lab 3 is out
  - Design doc due next Thursday (11/10)
  - Lab due Friday (11/18)

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
  - 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 sends a request to OS to expand the heap region
    - OS needs to see if the request can be granted (when might it not?), if so, it can allocate physical memory for the newly expanded heap

sbrk

(“set program break”)
Get more heap space

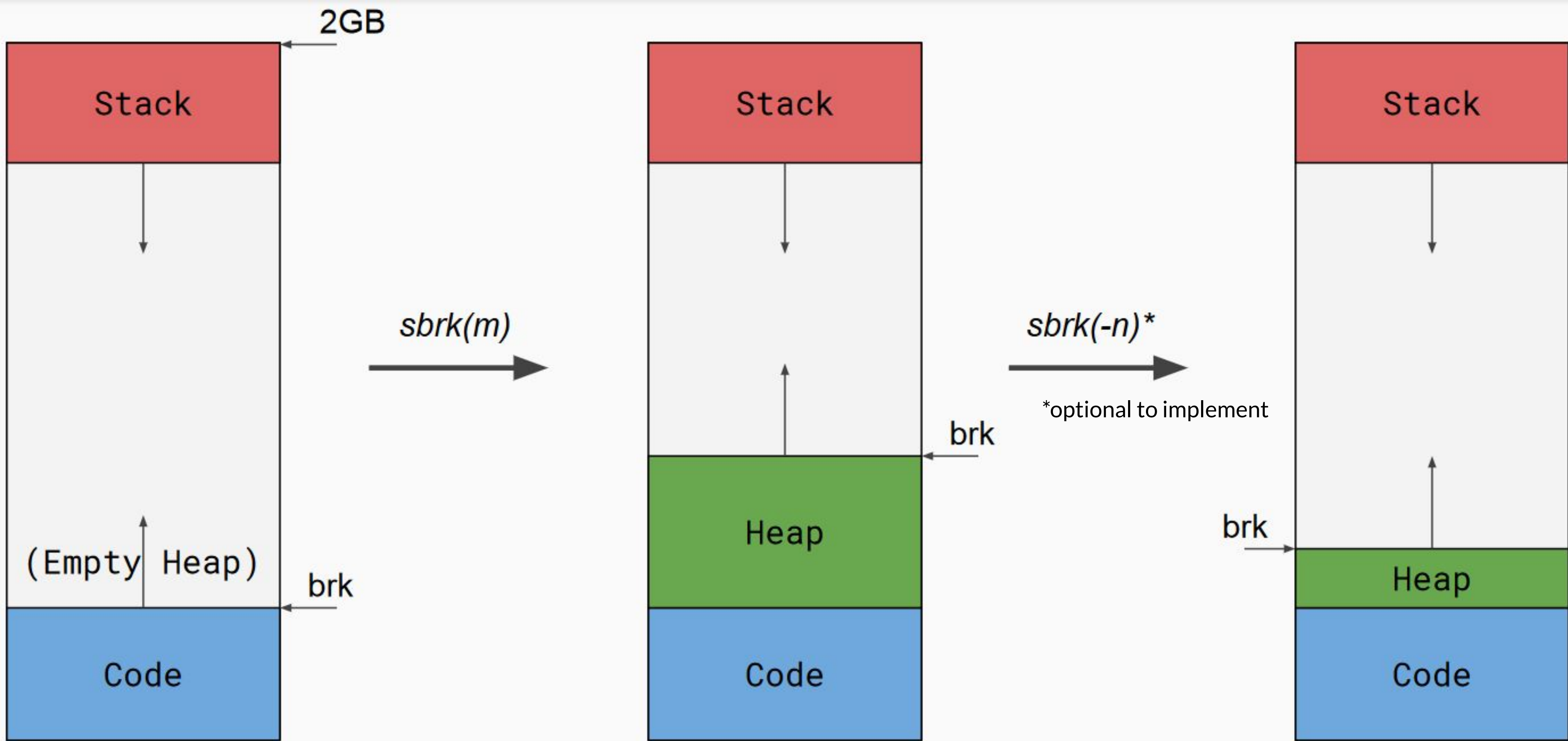
sbrk(n)

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

- Program break = max space allocated to the heap 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)


2GB
Stack
(Empty Heap)
brk
Code
sbrk(m)
Stack
brk
Heap
Code
sbrk(-n)*
*optional to implement
Stack
brk
Heap
Code

sbrk details

- sbrk is called with **byte** granularity
    - applications can expand the heap by a couple bytes (sbrk(1), sbrk(10)), or by multiple pages (sbrk(8K))
- New heap range needs to be updated in the bookkeeping data structures (vregion) and OS can allocate physical memory to it
    - use `vregionaddmap` to allocate frames for newly expanded pages
    - use `vspaceinvalidate` to update the page table with new mappings
- Once implemented, malloc/free tests should pass

The Shell

finally some interactions

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

Grow Stack on Demand

- The initial version of exec is pretty simple
  - one page of stack from SG_2G down
- 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? What happens when you access beyond the current stack page?

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
    - On page fault, you should grow the stack up to the faulting page (usually one page at a time)
        - char buf[12K]; access buf[10K]
- For simplicity, xk has a stack limit of 10 pages
    - If page fault and address > stack_base - 10 pages: grow stack;
    - Else: “normal” page fault (exit)
- Expand the stack similarly to sbrk
    - Adding pages to particular region, but trigger is different (page fault vs syscall)

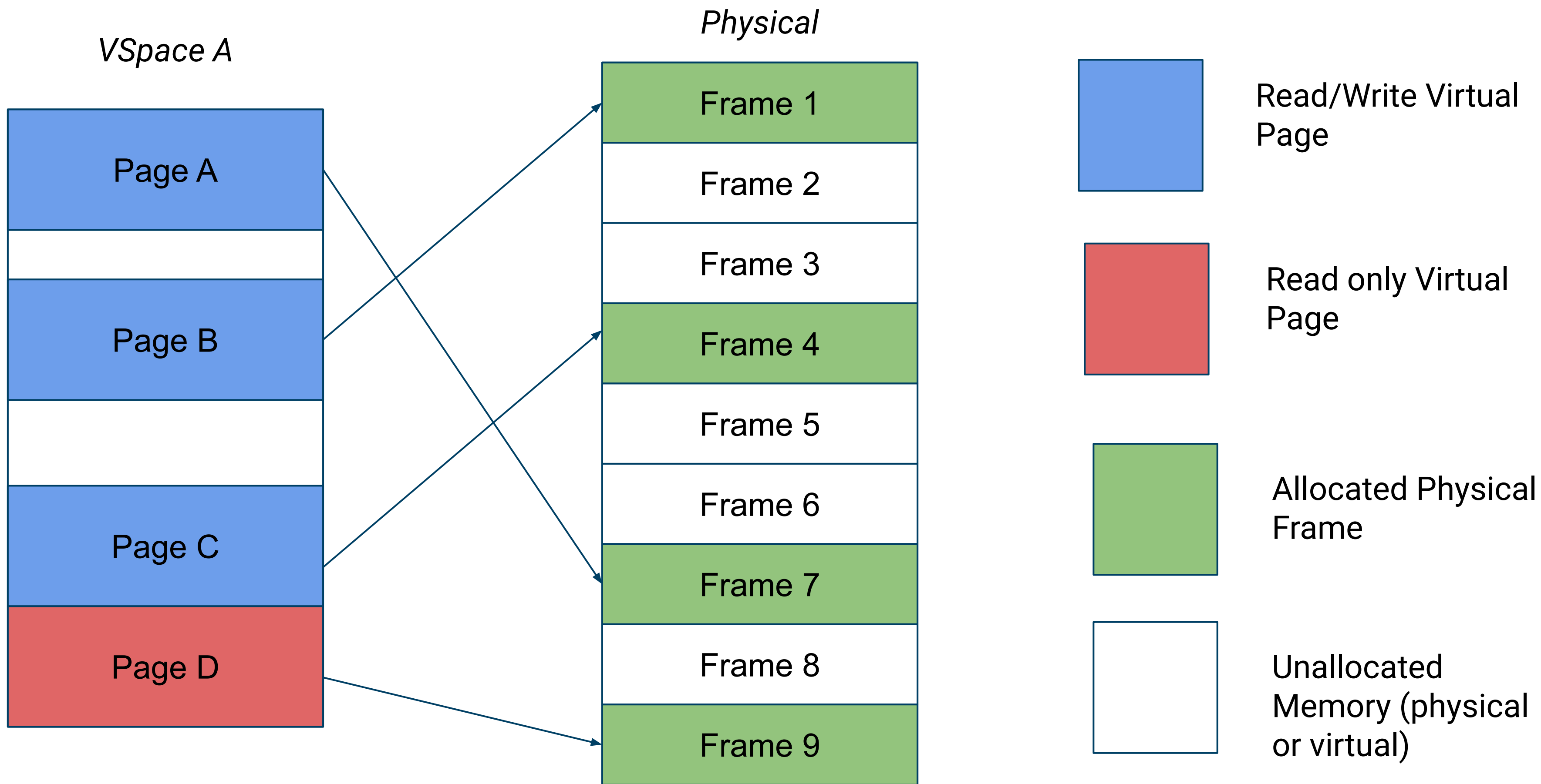
COW Fork

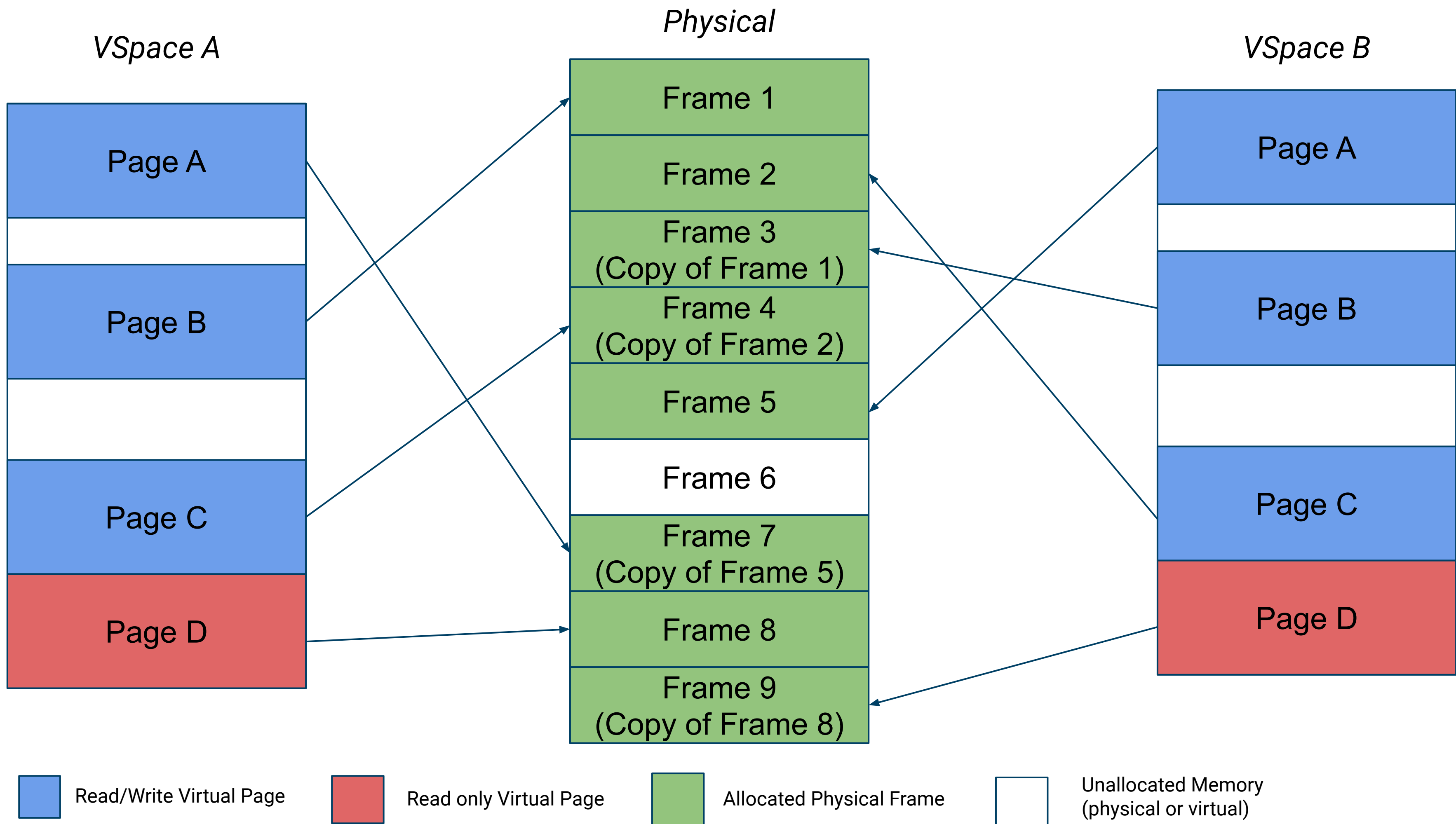
(no harm was done to these cows)

Copy on Write Fork

The lab 2 fork implementation is inefficient: `Vspacecopy` goes pages in the parent, if a page is mapped to a frame, allocate a new frame, copies the content to the new frame. It then calls `vspaceinvalidate` to update the child's table with the new mappings.

```
489 // copies the regions and pagesof the src vspace to dst
490 int
491 vspacecopy(struct vspace *dst, struct vspace *src)
492 {
493   struct vregion *vr;
494
495   memmove(dst->regions, src->regions, sizeof(struct vregion) * NREGIONS);
496
497   for (vr = dst->regions; vr < &dst->regions[NREGIONS]; vr++)
498     if (copy_vpi_page(&vr->pages, vr->pages) < 0)
499       return -1;
500
501   vspaceinvalidate(dst);
```


VSpace A
Page A
Page B
Page C
Page D
Physical
Frame 1
Frame 2
Frame 3
Frame 4
Frame 5
Frame 6
Frame 7
Frame 8
Frame 9
Read/Write Virtual Page
Read only Virtual Page
Allocated Physical Frame
Unallocated Memory (physical or virtual)


Physical
VSpace A
VSpace B
Page A
Page B
Page C
Page D
Frame 1
Frame 2
Frame 3
(Copy of Frame 1)
Frame 4
(Copy of Frame 2)
Frame 5
Frame 6
Frame 7
(Copy of Frame 5)
Frame 8
Frame 9
(Copy of Frame 8)
Page A
Page B
Page C
Page D
Read/Write Virtual Page
Read only Virtual Page
Allocated Physical Frame
Unallocated Memory
(physical or virtual)

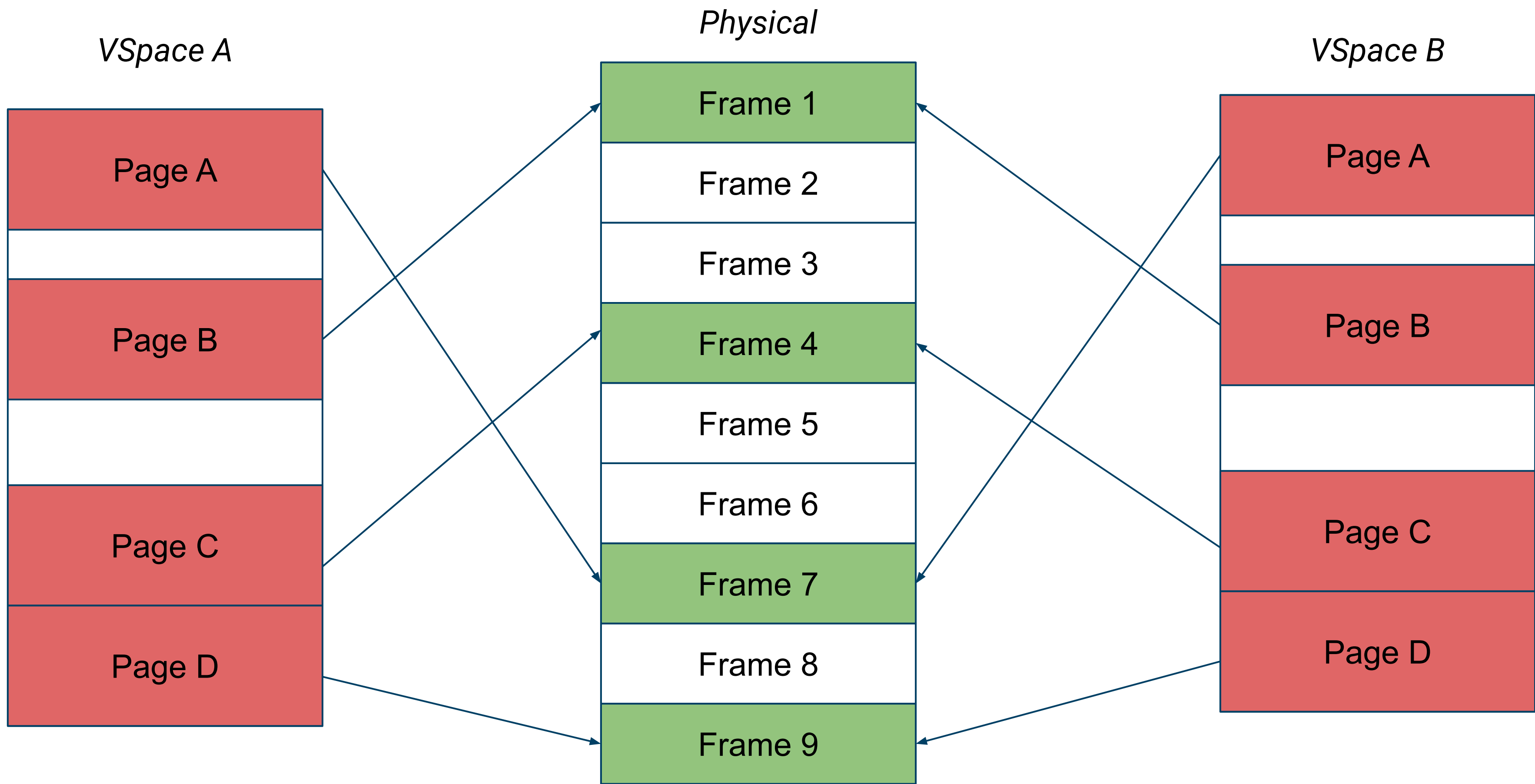
We copied all the pages :(

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? What can we do better?

- Don’t actually copy pages
- Copy 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



Read/Write Virtual Page
Read only Virtual Page
Allocated Physical Frame
Unallocated Memory
(physical or virtual)

VSpace A

Physical

VSpace B

Page A

Page B

Page C

Page D

Frame 1

Frame 2

Frame 3

Frame 4

Frame 5

Frame 6

Frame 7

Frame 8

Frame 9

Page A*

Page B

Page C

Page D

Page A in Vspace B could be safely set to writable (if there are no other vpsace pointers pointing at Frame 7).

Things to Consider

1. How do you distinguish a copy-on-write read-only page from a normal read-only page?
    a. hint: each page has a vpage_info struct that can be retrieved w/ va2vpage_info
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?