



Lab 3 Intro

Memory Management



Administrivia

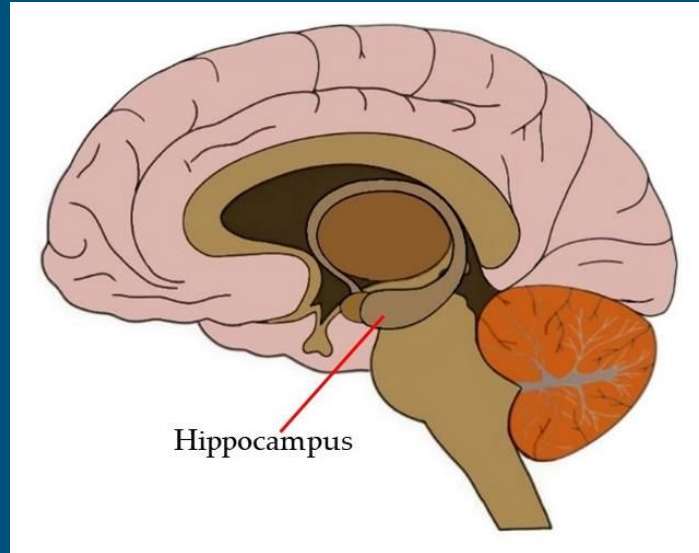
- Lab 2 Part 2 due yesterday 5/01
- Lab 3 is out!
 - Design doc due on Monday (5/06)
 - Lab 3 due 5/13

- Pset 4 due tomorrow 5/03
- Pset 5 out tomorrow 5/03

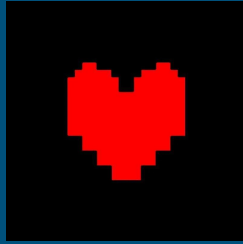
Today's Agenda

- `xk` virtual memory
- Go over the parts of Lab 3:
 - Part 1: `sbrk`
 - Part 2: The User Shell
 - Part 3: Stack Growth
 - Part 4: COW Fork
- Next week - deeper dive into `vspace` structs and functions

Memory in xk



Motivation

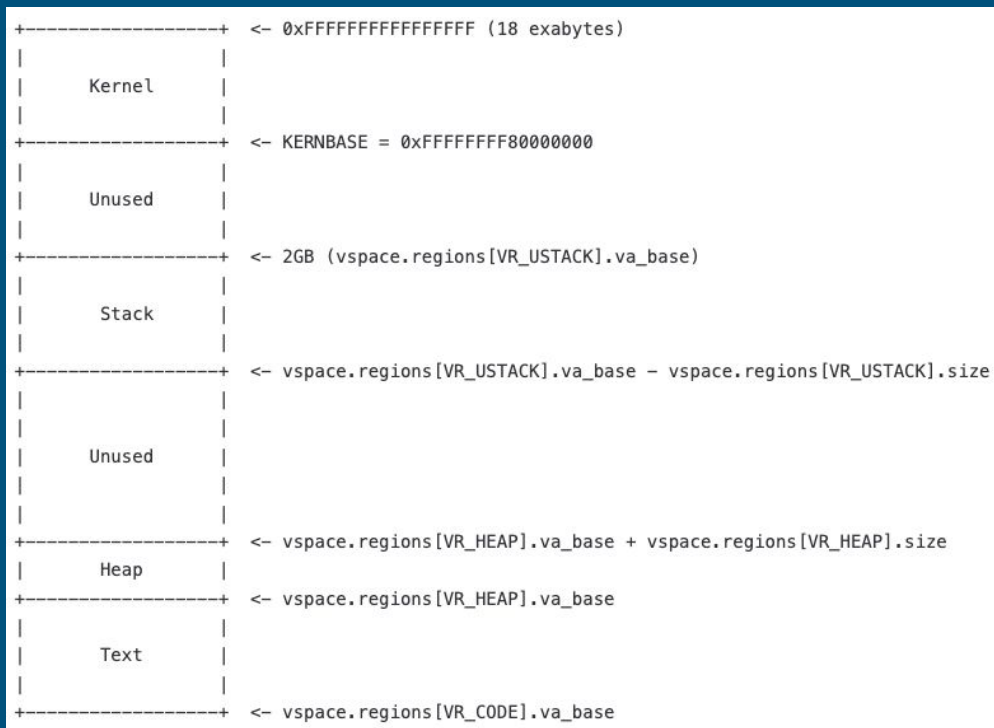


In Lab 3 you are required to implement multiple features which deal with virtual memory.

Therefore, understanding the existing virtual memory implementation in `xk` is essential! *(Unless you want to rebuild it all from scratch... please don't)*

So: How does virtual
memory do in **xk**?

Refresher: Xk's Virtual Memory Layout



This diagram from [memory.md](#) shows the layout of virtual addresses for a process in xk. (The addresses shown on left are virtual).

Key Takeaway: “virtual memory” is just a mapping from virtual addresses to physical addresses.

^From “*memory.md*” in the repo.

x86 Virtual Memory Datastructures

...but where do those mappings come from?

- (As you learned in lecture) paging in x86-64 is handled directly by the hardware using a *page table* data structure which lives in memory.
 - The `pml4e_t* pgtbl;` field of `struct vspace`
- The `%CR3` register tells the CPU the physical address of the page table.
 - In `xk lcr3()` updates `%CR3`

Thus a “virtual address space” is defined by a single page table, and “switching virtual address spaces” means to point `%CR3` to a new page table.

So... simple right?



To implement lab 3 you just need to modify the appropriate page tables, switching between them at the appropriate times.

... almost...

xk's Platform-Agnostic Virtual Memory

xk implements virtual memory data structures which:

- Are platform agnostic.
- Present a simpler interface.
- Can be translated to x86-specific page tables.

Key point: you should manipulate these (rather than the page table directly).

Note that xk provides functions in `vspace.c` for manipulating these.

xk's Platform-Agnostic Virtual Memory

You've already seen some of these platform-agnostic data structures, but the main ones are:

- Struct `vspace`
- Struct `vregion`
- Struct `vpi_page`
- Struct `vpage_info`

Throughout this lab you will become masters of wielding these structs.

Virtual Memory Summary

And that's that! Just as a quick recap:

- Actual virtual memory is implemented in hardware
- The hardware uses platform-specific data structures like the page table
- `xk` uses additional platform-agnostic virtual memory data structures
- You should primarily interact with the platform-agnostic data structures.

Now let's talk about the lab!

Part 1: `sbrk`

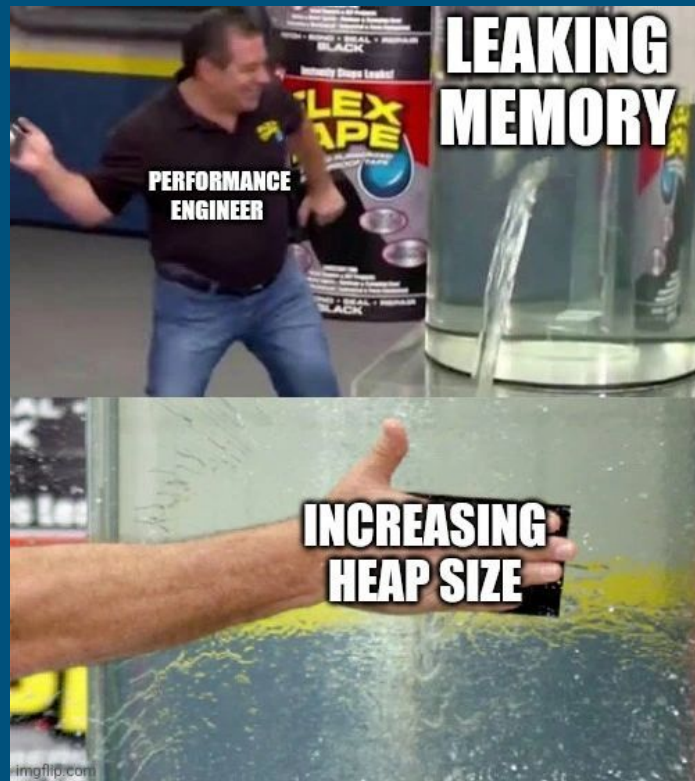
(“set program break”)
Get more heap space

Refresher: User Level Heap Management

- Recall that 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**

I've run out of heap space...

But modifying the page table is a privileged operation... how does `malloc` get more free memory?



I've run out of heap space...

But modifying the page table is a privileged operation... how does `malloc` get more free memory?

- 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, allocate physical memory and map the extended heap region to it

What is the name of the system call that gets more heap space?



Enter `sbrk(n)`

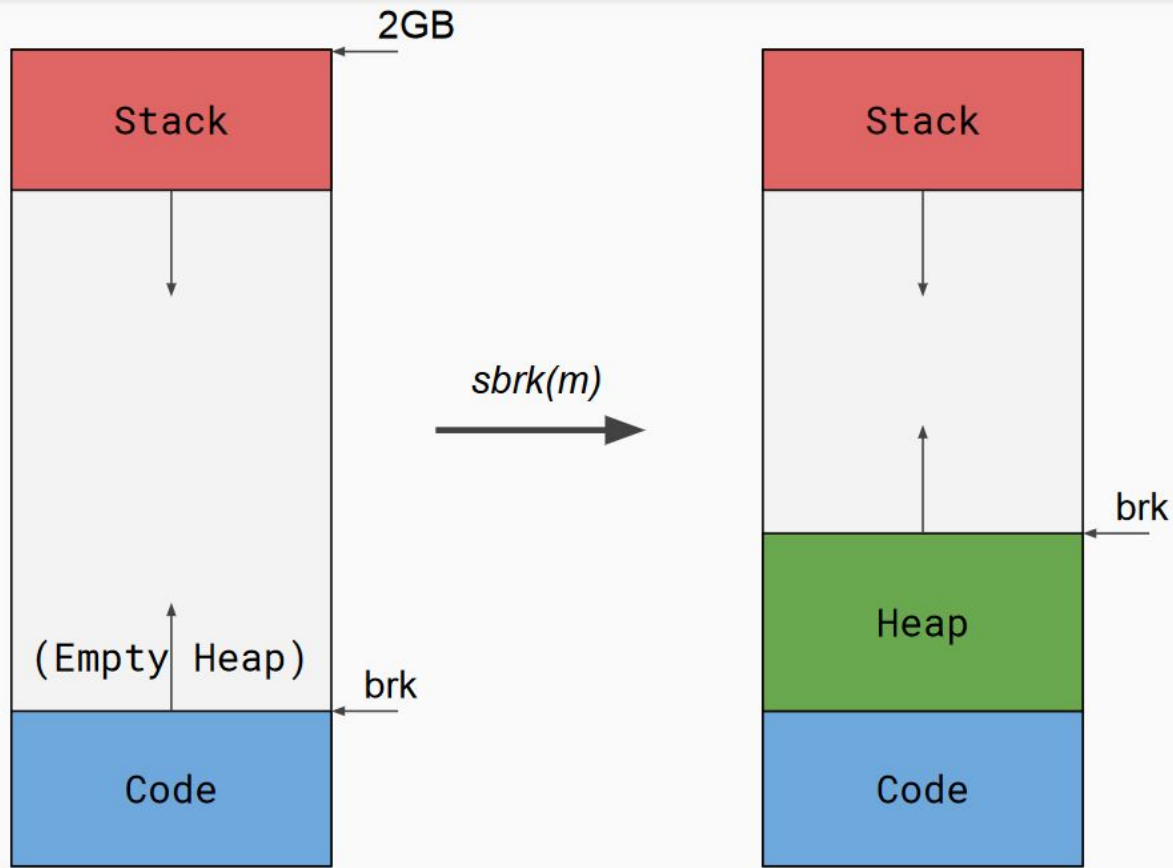
What does `sbrk()` do?

- 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 is not required for xk
- Returns -1 if it can't allocate enough space
- Otherwise, return the *previous* heap limit (the old top of the heap)

You need to implement this syscall for Lab 3 Part 1

before

after



sbrk details

- sbrk is called with **byte** granularity
 - You can expand by a couple bytes: `sbrk(10)`
 - You can expand by multiple pages: `sbrk(8K)`
- Extended heap range needs to be updated in the heap **vregion**
 - You may need to allocate physical memory for newly extended range of virtual address space
 - Where is the heap **vregion**? (Hint: take a look at memory.md)
- Heap can grow as long as it doesn't collide with any other **vregion**.

sbrk's Byte Granularity

Wait a second... Physical memory can only be allocated in pages, and `sbrk()` operates on byte granularity!

If a new process calls `sbrk(1)` followed by a `sbrk(100)`, how many pages should we allocate?

What to do for `sbrk`

What is provided for you:

- `vspace.c` contains a function which adds new virtual address mappings to a `vregion` (and allocates physical frames as necessary).

What you need to do in `sbrk()`:

- Calculate previous top of heap
- Add/update mappings in the page table by calling provided functions
- Return previous top of heap (or -1 on error)

If you find yourself writing > dozen lines of code, you're probably reinventing the wheel...

Part 2: The Shell

finally some interactions

The Shell

How does the shell work?

- The initial process (user/init.c) will fork to create a shell
- Shell uses malloc and sbrk!
- The shell will take user input and run commands, spawning other programs
 - Just like bash, cmd, or whatever else

- You can now use the shell by typing different commands!
 - 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)

How to Test the Shell

Remember when we said to make your lab 1 and lab 2 system calls as comprehensive as possible? Well, the shell will put them to the test!

What you need to do:

- Boot the shell when you “make qemu” (the spec outlines how to do this)
- Test your shell with the following commands:
 - `echo "hello world" | wc`
 - `cat small.txt | wc`
 - `ls`
- If there are bugs, fix them now!
- Often the pipe command reveals issues that were not covered by Lab 2 Part 2 tests

Part 3: Let it Grow

more stack

Grow Stack on Demand

- The initial version of `exec()` is pretty simple
 - one page of stack from `SZ_2G` down
- One page of stack probably isn't enough for larger programs.
- But giving more pages is wasteful if the program doesn't need it.
- In lab 3, you need to ***dynamically add more pages of stack as needed.***

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!** (Hint: in trap.c, trap 14 is 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)

- 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)

Stack Growth vs sbrk

Stack growth and sbrk call similar vspace functions to grow the respective vregions.

However, the growth trigger for stack growth vs heap growth is different:

- Page fault → Stack growth
- System call → Heap growth

What to do for Lab 3

What is provided:

- Basic trap handler is provided for you as `trap()` in `trap.c`
- Once again, there are provided `vspace` functions
 - Similar to what you use in `sbrk()`!

What you need to do:

- Alter `trap()` function to handle stack growth
 - Extract information about faulting address using error codes



Part 4: COW Fork

(no harm was done to these cows)



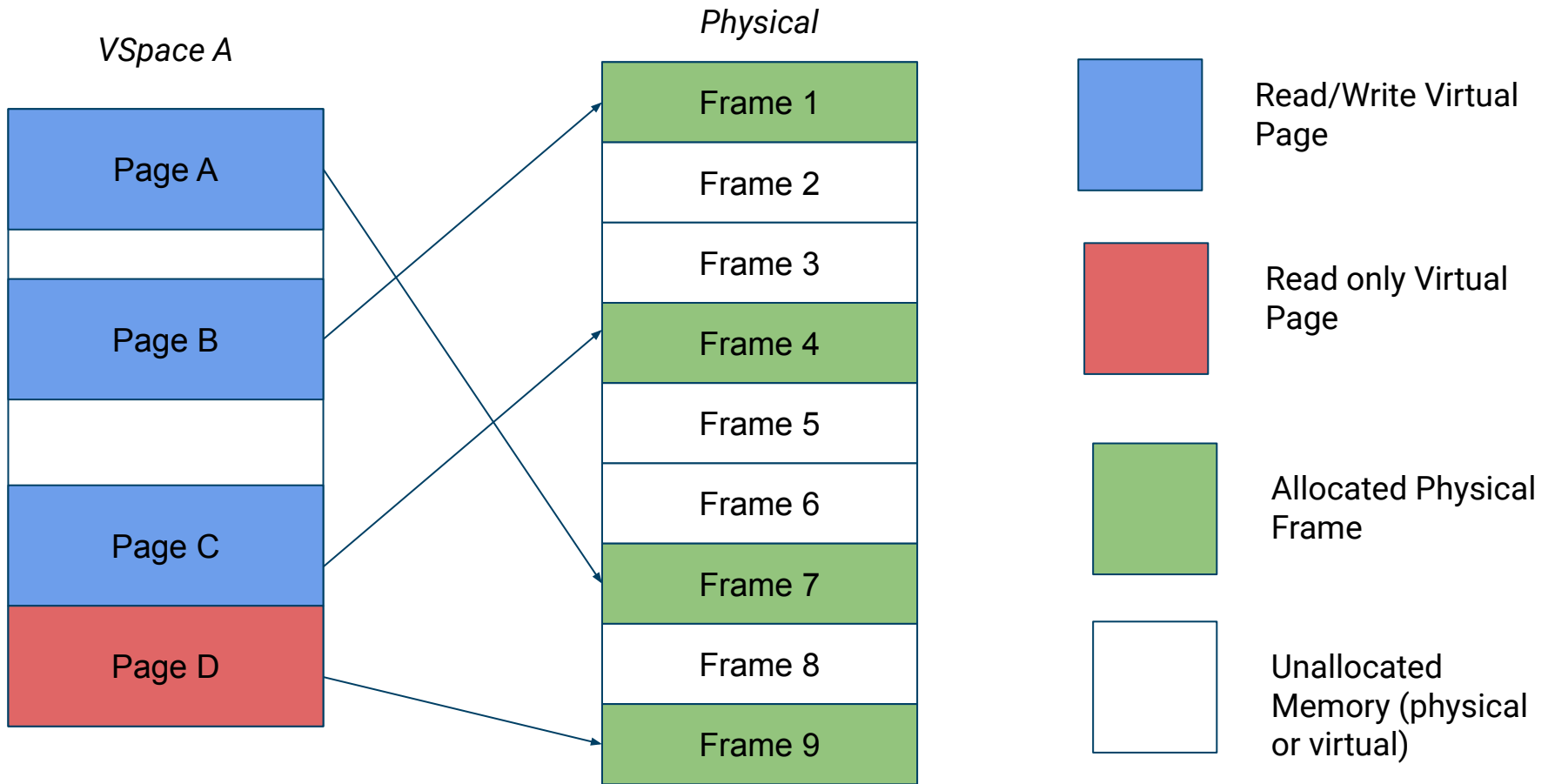
Copy on Write Fork

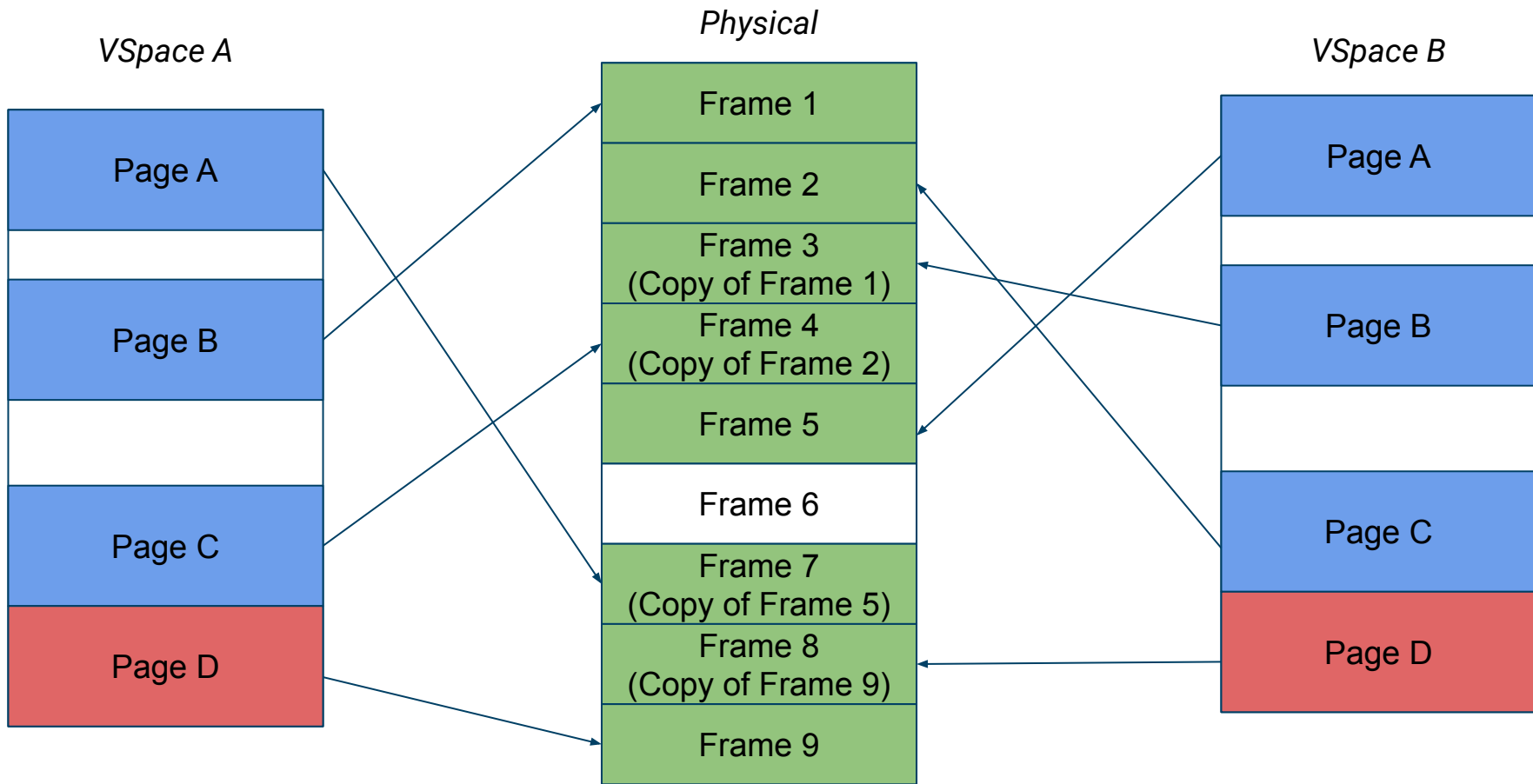
Remember when you called `vspacecopy` to copy the `vspace` of the parent to the child?

Well, your lab2 fork implementation is inefficient because of how this function currently works :(

`vspacecopy` at a high-level:

- Go through pages in the parent
- if a page is mapped to a frame, allocate a new frame
 - Copy the content to the new frame
 - Call `vspaceupdate` to update the child's table with the new mappings.





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

We copied all the pages :(

As a consequence:

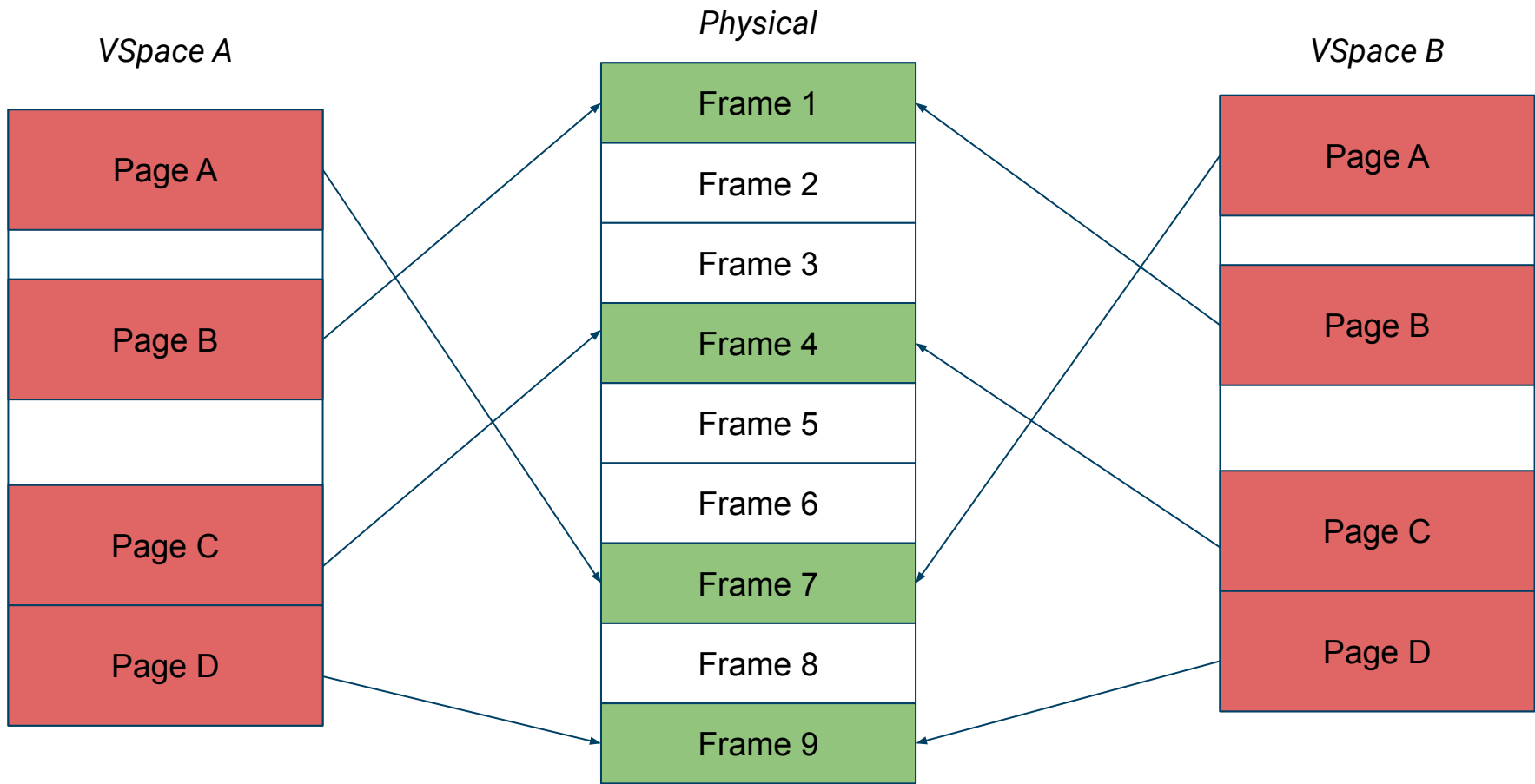
- Child and parent duplicate the same unchanging pages of code and static data
- If we fork and exec, we waste time copying all pages before immediately discarding the vspace

Your job in Lab 3 Part 4 is to fix this!

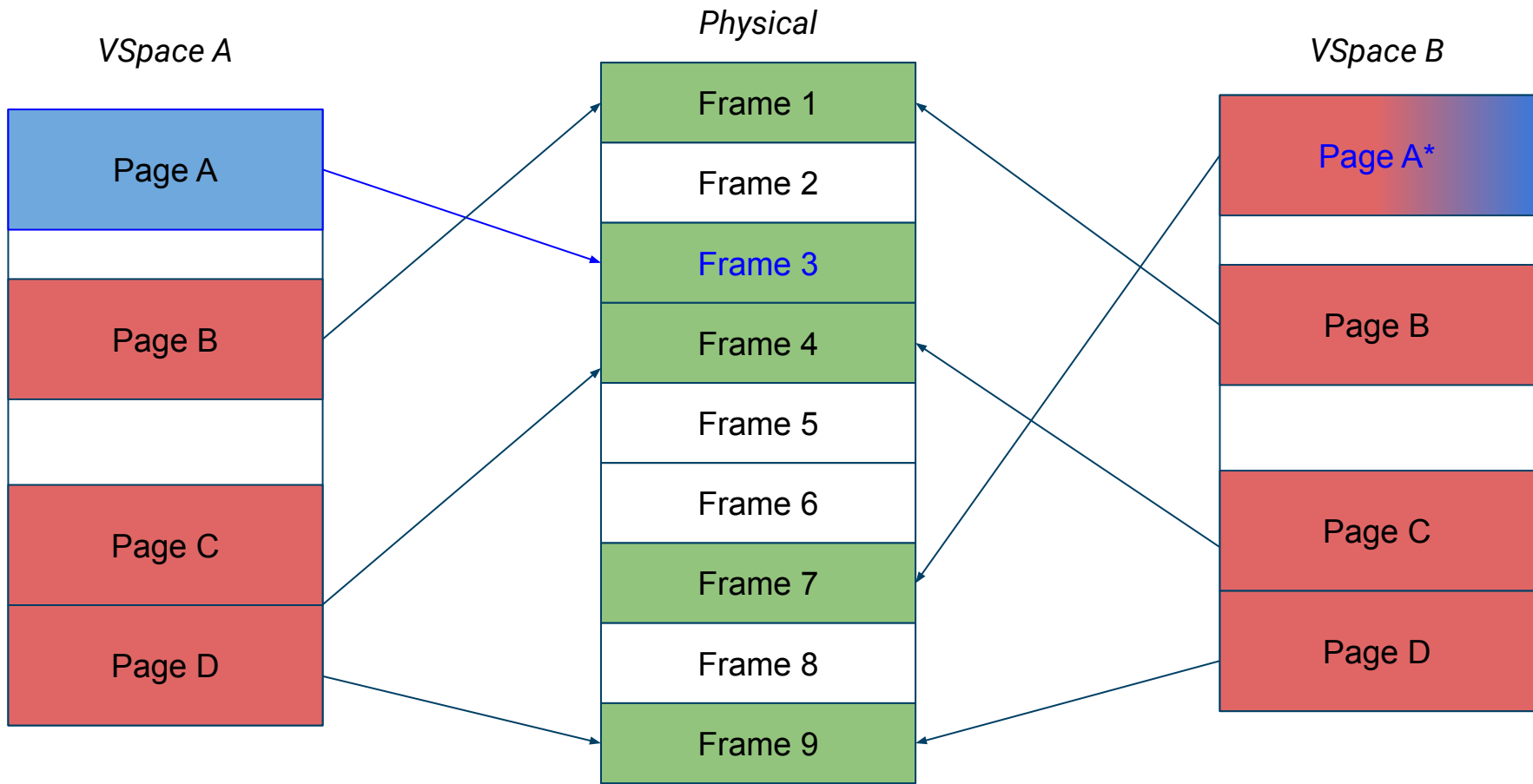
Optimizing Fork Memory

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
- 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)




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

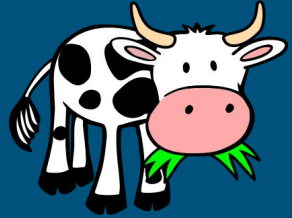
 Read/Write Virtual Page

 Read only Virtual Page

 Allocated Physical Frame

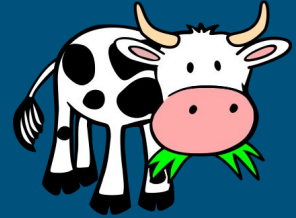
 Unallocated Memory (physical or virtual)

COW Food For Thought



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. How to track the lifetime of a physical frame now that it can be shared by many processes?
 - a. Hint: there is a per frame metadata called `core_map_entry`
3. What happens (specifically) when a process tries to write to a COW page?
 - a. Hint: a page fault? what error code?

COW Food For Thought cont.



1. What happens if both parent and child writes to the same cow page?
 - a. Hint: is there any race condition? how do you synchronize this?
2. What happens when a child with COW pages calls fork()?
 - a. Hint: The parent, child, grandchild, etc. will all reference the same physical pages, so what do you need to update?

What to do for Lab 3

What is provided:

- Trap handler
- Relevant structs/functions in `vspace.c` and `vspace.h`

What you need to do:

- Alter `trap()` function to handle COW cases
- Create a `vspacecowcopy()` function that *shares physical memory*
- Update `fork()` implementation if necessary
- Update relevant structs to track extra information related to COW pages

Ok cool! I'm starting to understand Lab 3...

But wait! How do I distinguish stack growth vs COW page faults?

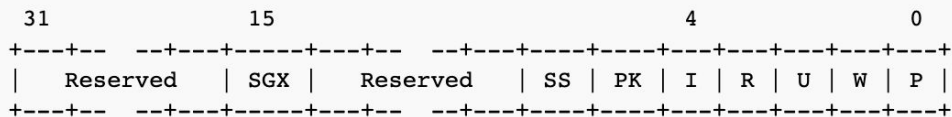
- You, a trap-handling expert in the making



Handling Page Faults in x86-64

- The **%CR2** register holds the faulting virtual address
 - How do you read or load a control register? (look in trap.c in the default case)
- `tf->err` holds the exception error code
 - You can use this to determine the type of fault!

The Page Fault sets an error code:



	Length	Name	Description
P	1 bit	Present	When set, the page fault was caused by a page-protection violation. When not set, it was caused by a non-present page.
W	1 bit	Write	When set, the page fault was caused by a write access. When not set, it was caused by a read access.
U	1 bit	User	When set, the page fault was caused while CPL = 3. This does not necessarily mean that the page fault was a privilege violation.

Error Codes

- Last 3 bits of `tf->err`
 - B2 is set if fault occurred in user mode
 - B1 is set if fault occurred on a write
 - B0 is set if the faulting page is mapped to a physical frame
 - if we page fault on a page that's mapped, then it's caused by permission issues

Error Codes

- Last 3 bits of `tf->err`
 - B2 is set if fault occurred in user mode
 - B1 is set if fault occurred on a write
 - B0 is set if the faulting page is mapped to a physical frame
 - if we page fault on a page that's mapped, then it's caused by permission issues

What will the error code be if the page fault was from touching the stack region of memory?

Error Codes

- Last 3 bits of `tf->err`
 - B2 is set if fault occurred in user mode
 - B1 is set if fault occurred on a write
 - B0 is set if the faulting page is mapped to a physical frame
 - if we page fault on a page that's mapped, then it's caused by permission issues

What will the error code be if the page fault was from touching the stack region of memory?

What about writing to a copy-on-write page?