



Lab 3 Intro

Memory Management



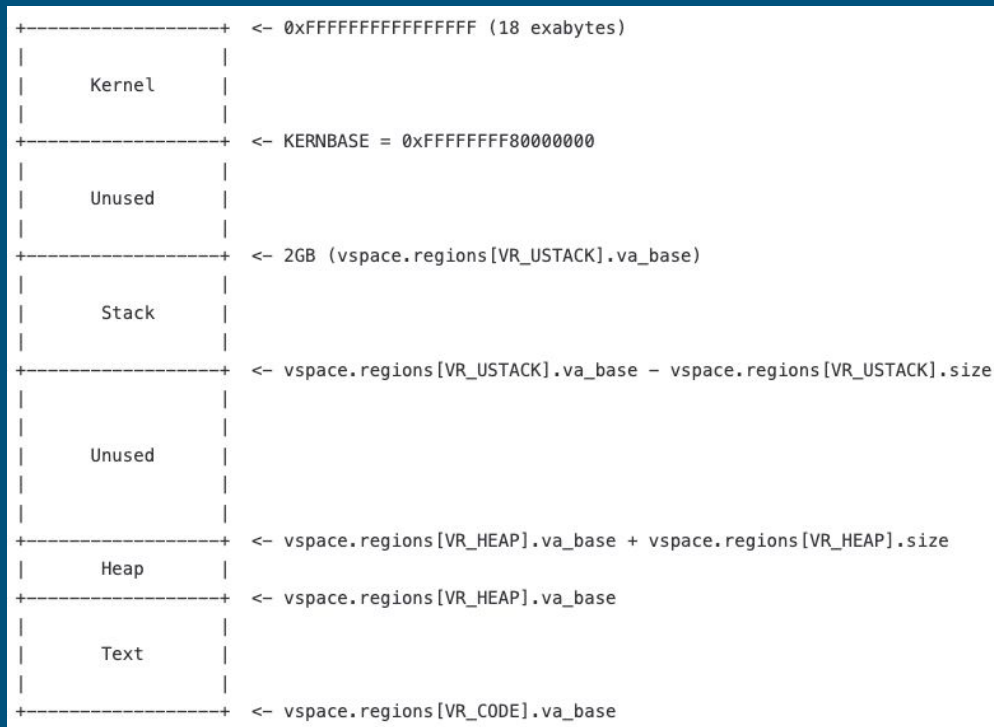
Reminder

- Lab 3 is out
 - Design doc due next Thursday (11/9)
 - Lab due Friday (11/17)
- Midterm feedback
 - closes this Friday, please fill it out!
 - <https://uw.iasystem.org/survey/278603>

Today's Agenda

- xk virtual memory
- 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

xk virtual memory



```
struct vspace {  
    struct vregion regions[NREGIONS];  
    pml4e_t* pgtbl;  
};
```

```
enum {  
    VR_CODE = 0,  
    VR_HEAP = 1,  
    VR_USTACK = 2,  
};
```

- vspace: metadata for an address space
 - tracks a list of vregion (code region, stack region, heap region)
 - a pointer to the actual page table (used by hardware to perform memory translation)

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

sbrk

(“set program break”)
Get more heap space

sbrk(n)

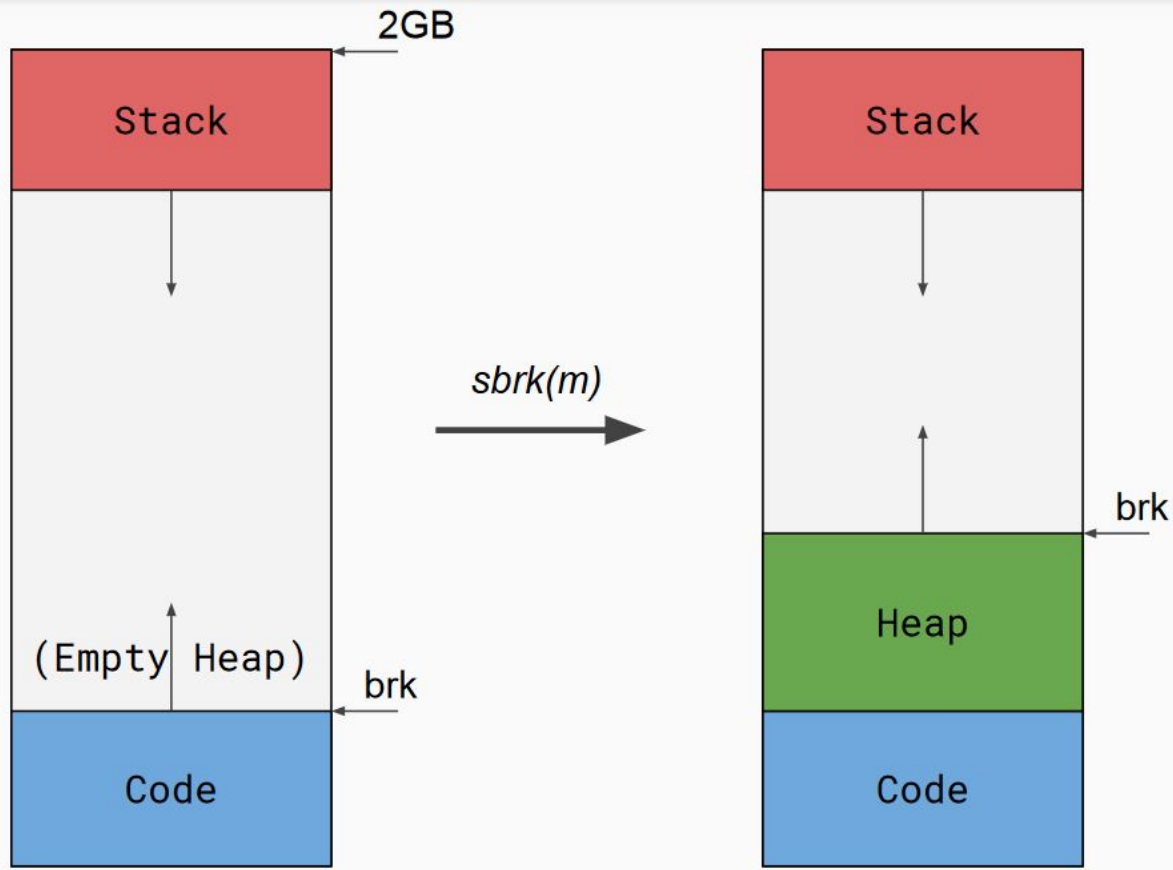
Increase the size of the heap by n bytes, updating the “program break”

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)



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)`)
- extended heap range needs to be updated in the heap vregion, may need to allocate physical memory for newly extended range of virtual address space
 - use `vregionaddmap` to allocate frames for newly expanded pages
 - use `vspaceupdate` to update the page table with new mappings
- NOTE: sbrk is called with byte granularity, but physical memory can only be allocated in pages!
 - if a new process calls `sbrk(1)` followed by a `sbrk(100)`, how many pages are allocated?
- Once implemented, malloc and sbrk 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! (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)
 - `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

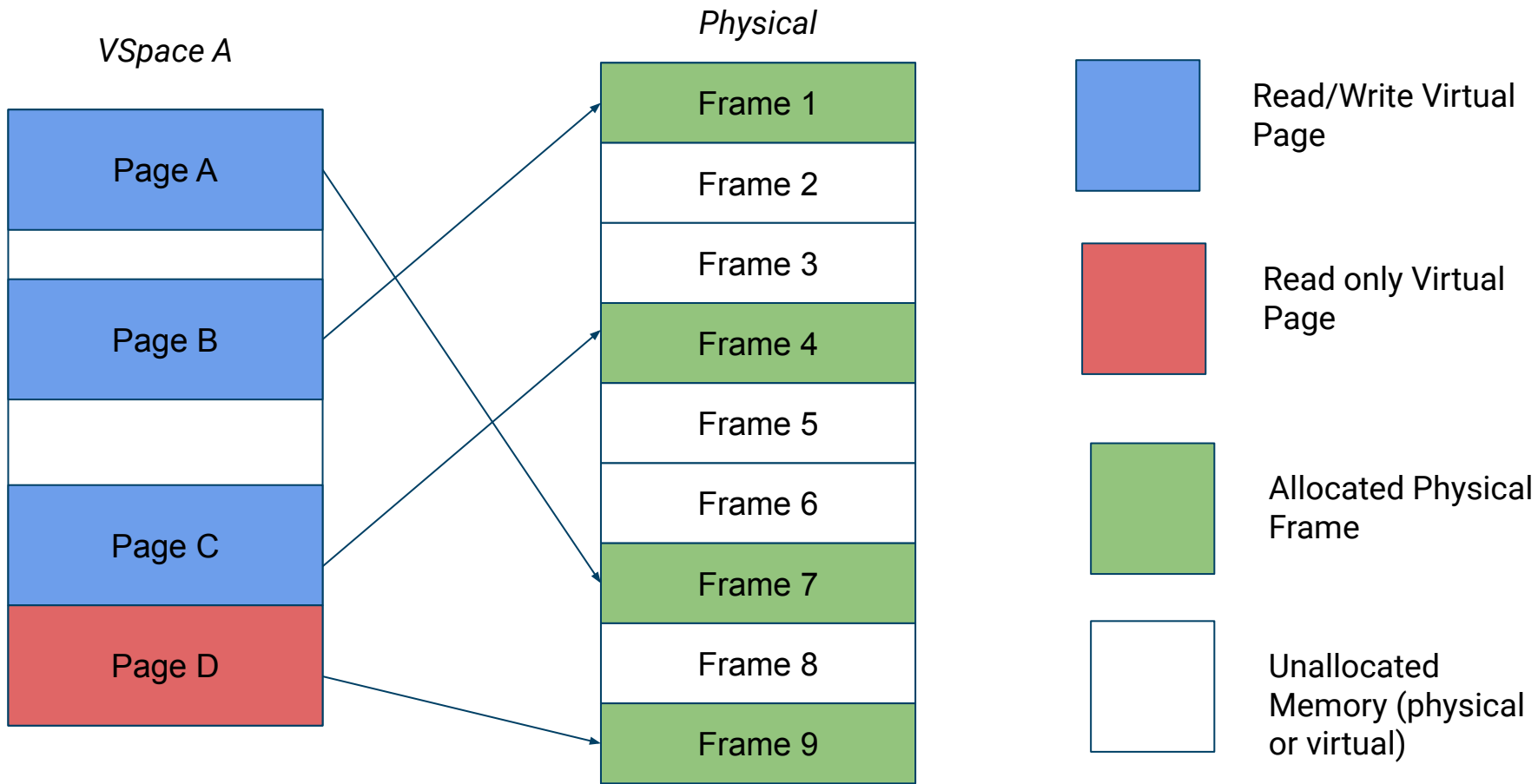
The lab 2 fork implementation is inefficient: `vspacecopy` goes through pages in the parent, if a page is mapped to a frame, allocate a new frame, copies the content to the new frame. It then calls `vspaceupdate` to update the child's table with the new mappings.

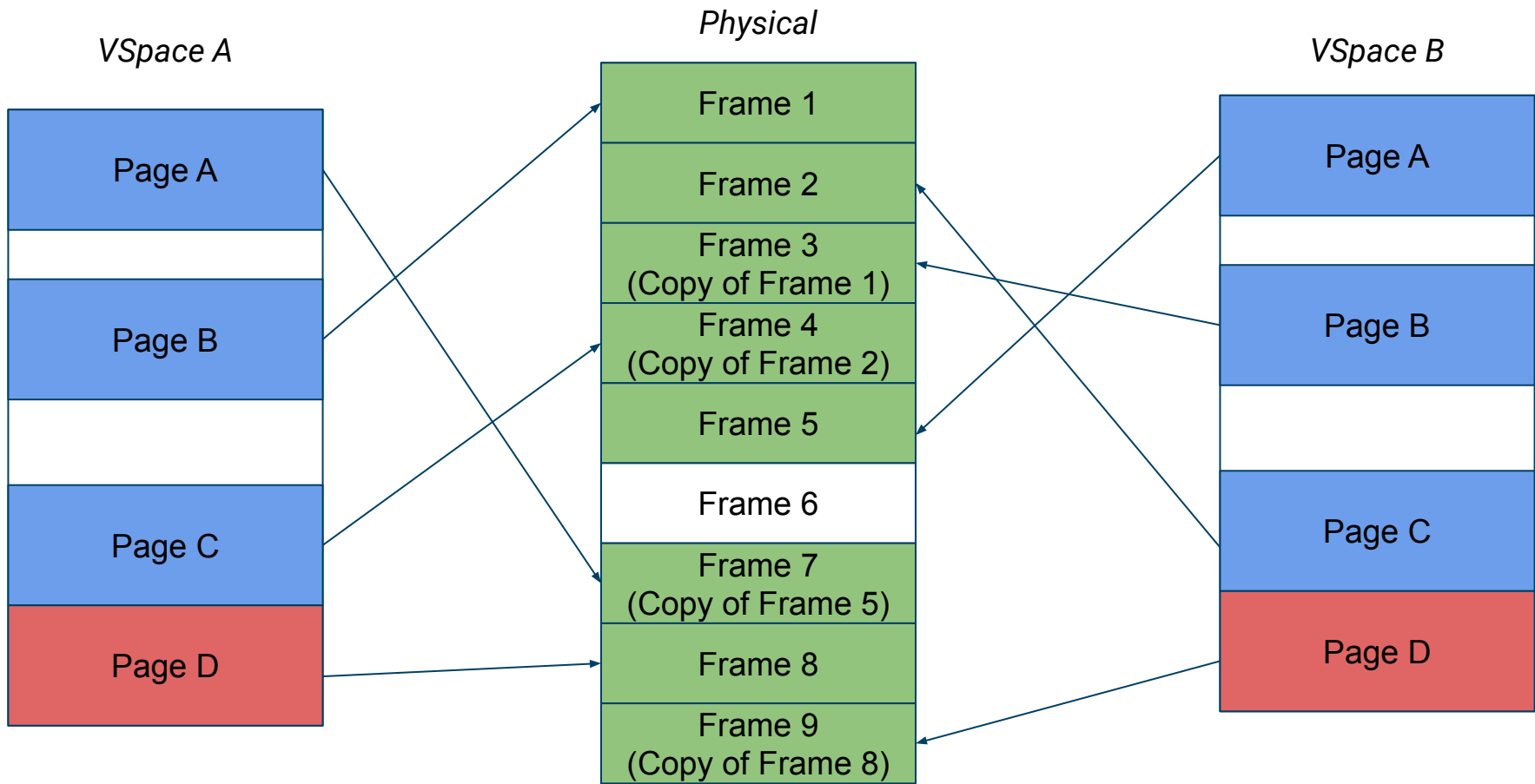
```
// copies the regions and pages of the src vspace to dst
int
vspacecopy(struct vspace *dst, struct vspace *src)
{
    struct vregion *vr;

    memmove(dst->regions, src->regions, sizeof(struct vregion) * NREGIONS);

    for (vr = dst->regions; vr < &dst->regions[NREGIONS]; vr++)
        if (copy_vpi_page(&vr->pages, vr->pages) < 0)
            return -1;

    vspaceupdate(dst);
}
```



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

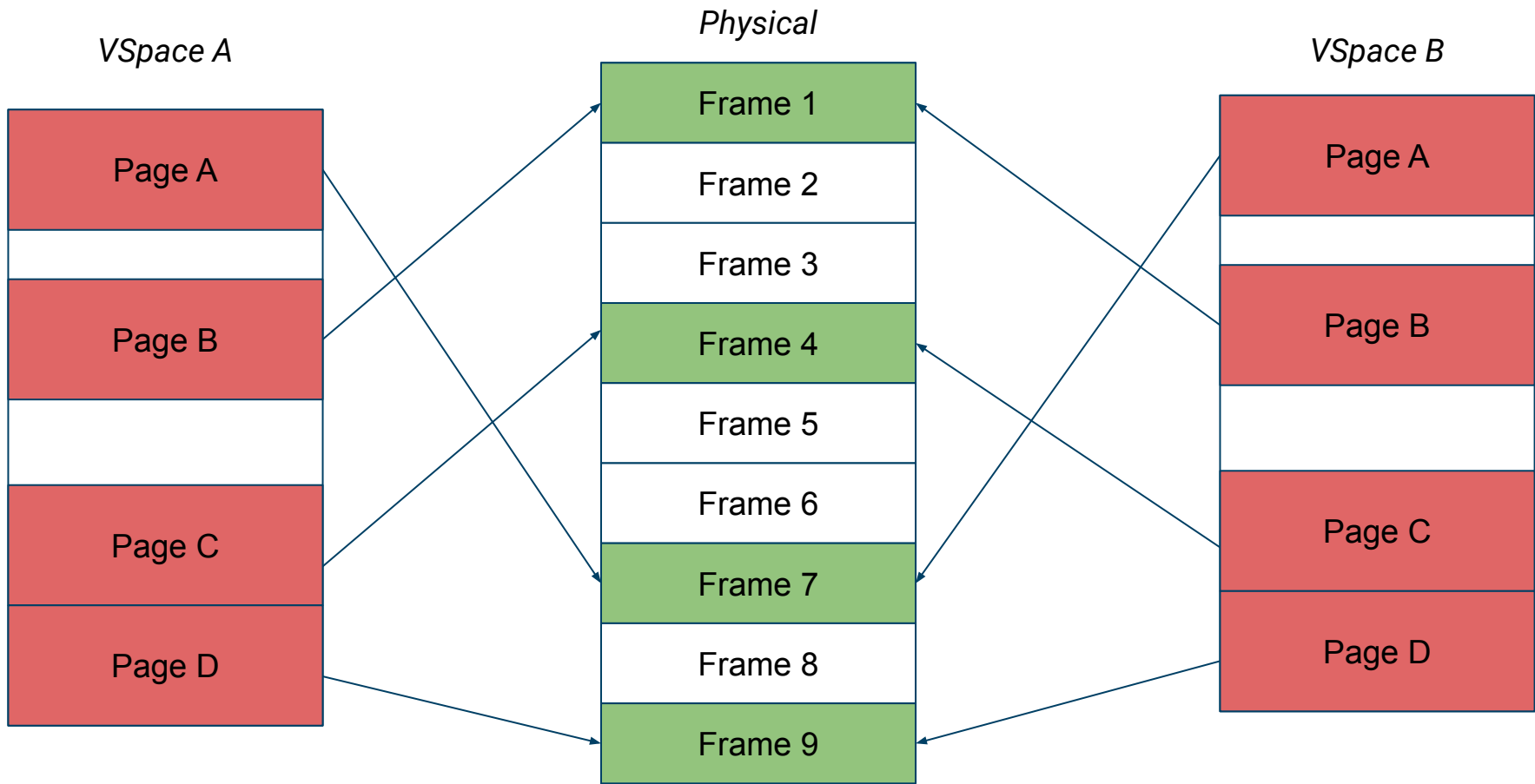
We copied all the pages :(

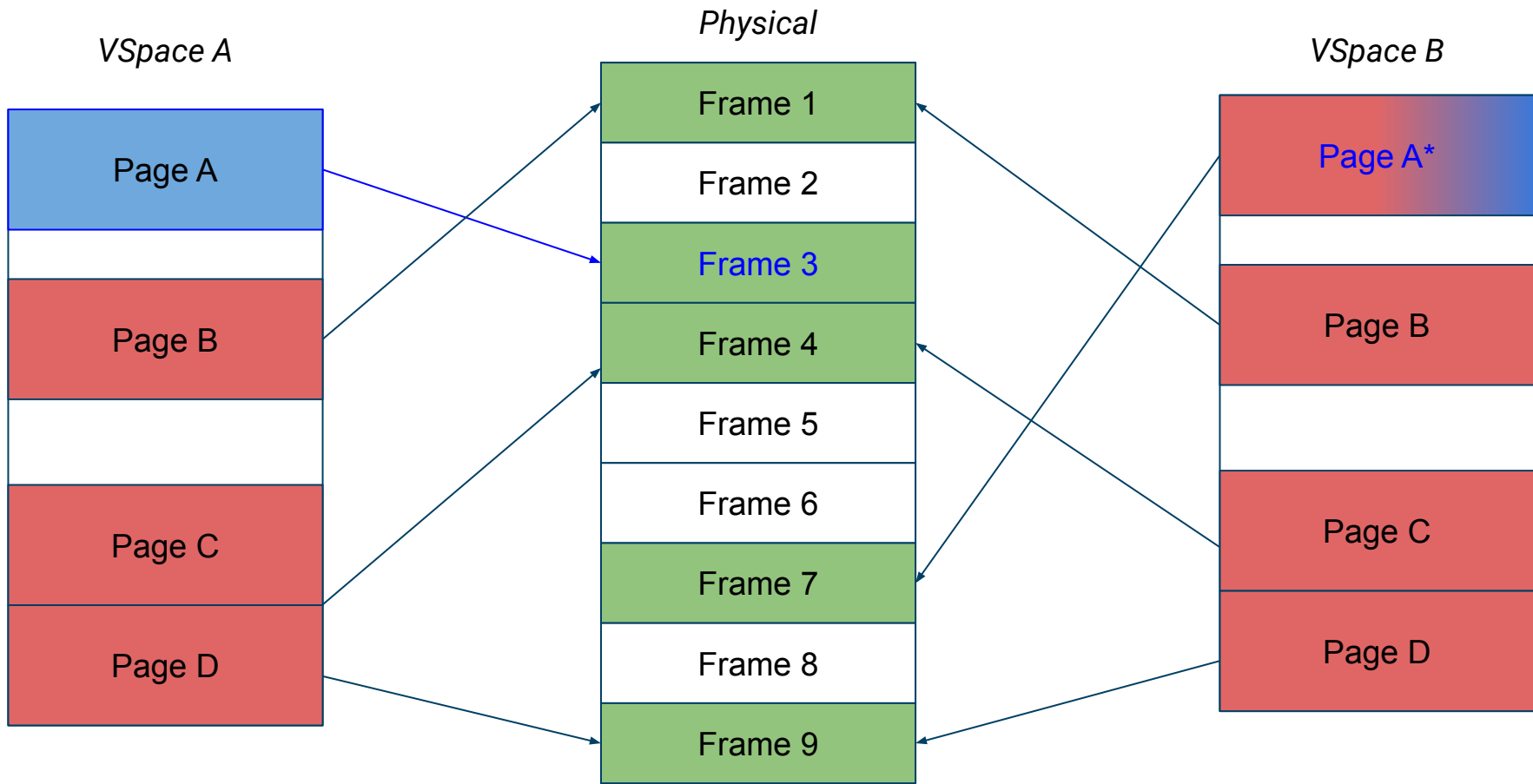
As a consequence:

- 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







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

 Read/Write Virtual Page

 Read only Virtual Page

 Allocated Physical Frame

 Unallocated Memory (physical or virtual)

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

Vregions vs Page Tables

- Both have virtual to physical address mappings
- **vspace.pgtbl**
 - Used by hardware to translate virtual addresses to physical addresses
 - **CR3** register holds the top level page table (i.e. **vspace.pgtbl**)
 - TLB caches virtual -> physical mappings
- **vspace.regions**
 - Portable *architecture independent* software representation of the address space
 - Used by kernel to track/update mappings without affecting hardware page table lookups
 - May be incomplete at times (e.g. mappings in `exec()`)
- How do we update the page table to reflect the vspace regions?

vspaceupdate(vs)

- “Build the architecture dependent page table based on vspace information”
 - I.e. virtual mappings in `vs.regions` are reflected in `vs.pgtbl`
- Call when you’ve changed a mapping in vspace

When should you call `vspaceupdate` in Lab 3?

vspaceinstall(p)

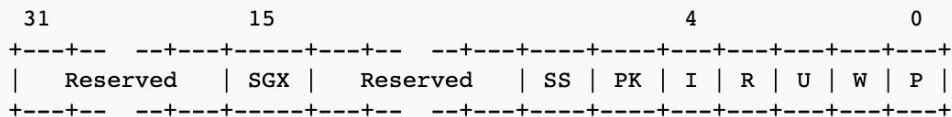
- “Installs the page table into the page table register”
 - I.e. `CR3 = vs.pgtb1`
 - In x86-64, this flushes the TLB!
- Flushes the TLB (remove any cached translation)

When should you call `vspaceinstall` in Lab3?

Handling Page Faults in x86-64

- 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.

More on 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?