Lab 3 Intro

Memory Management

Administrivia

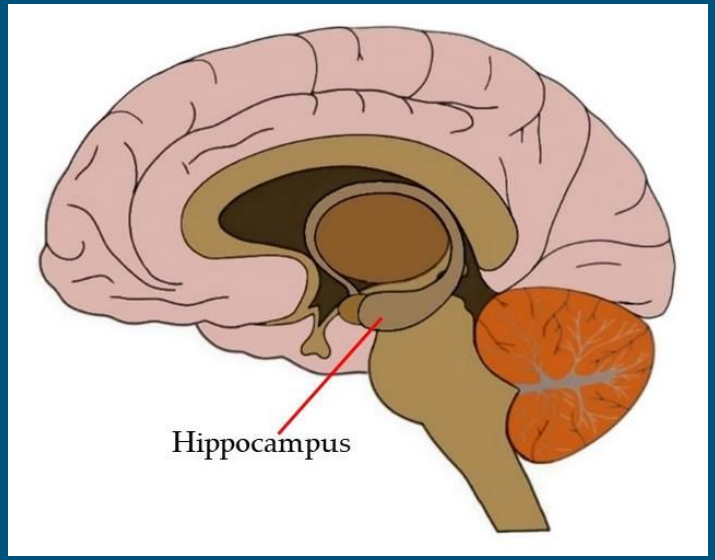
- Psets 4 and 5 due Monday (11/4)
- Lab 3 is out!
  - Design doc due 11/12
  - Lab 3 due 11/18
- Lab 2 design doc revisions (for W credit) due next (11/12)

Today’s Agenda

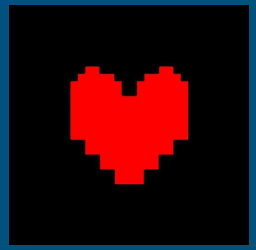
- xk virtual memory
- Go over the parts of Lab 3:
  - Part 1: sbrk
  - Part 2: The User Shell
  - Part 3: Stack Growth

- Tomorrow: Dive deeper into COW fork

Memory in xk



Motivation

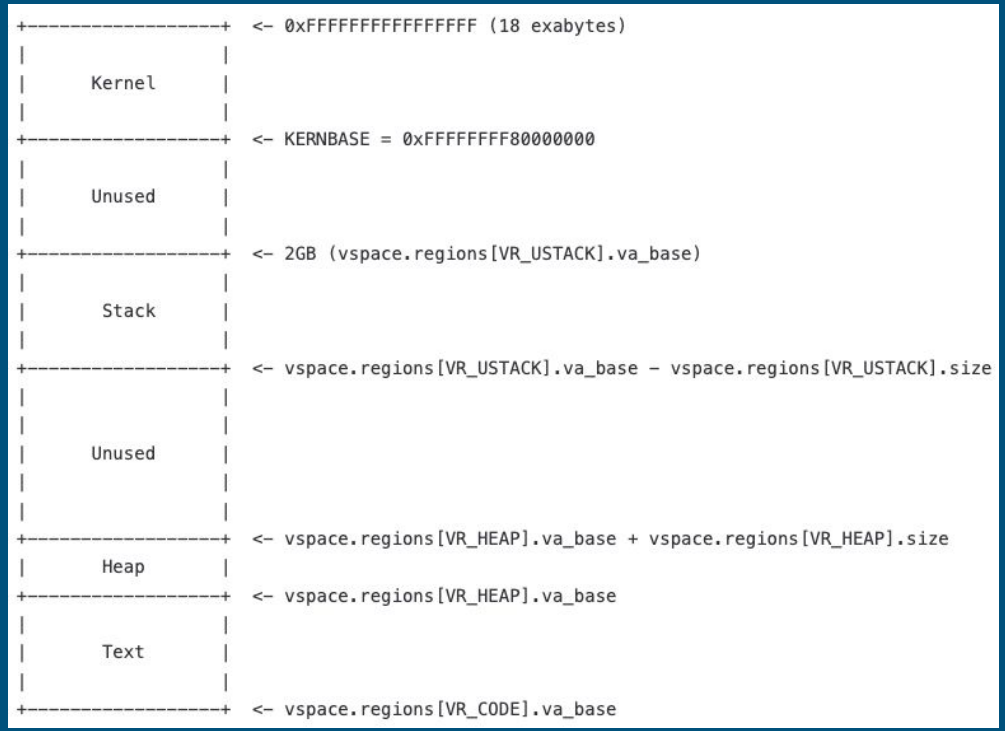


In Lab 3 you are required to implement multiple features which deal with virtual memory.

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



^From *"memory.md"* in the repo.

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.

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.

vpage_info

```
struct vpage_info {
  short used;     // whether the page is in use
  uint64_t ppn;   // physical page number
  short present;  // whether the page is in physical memory
  short writable; // does the page have write permissions
  // user defined fields

};
```

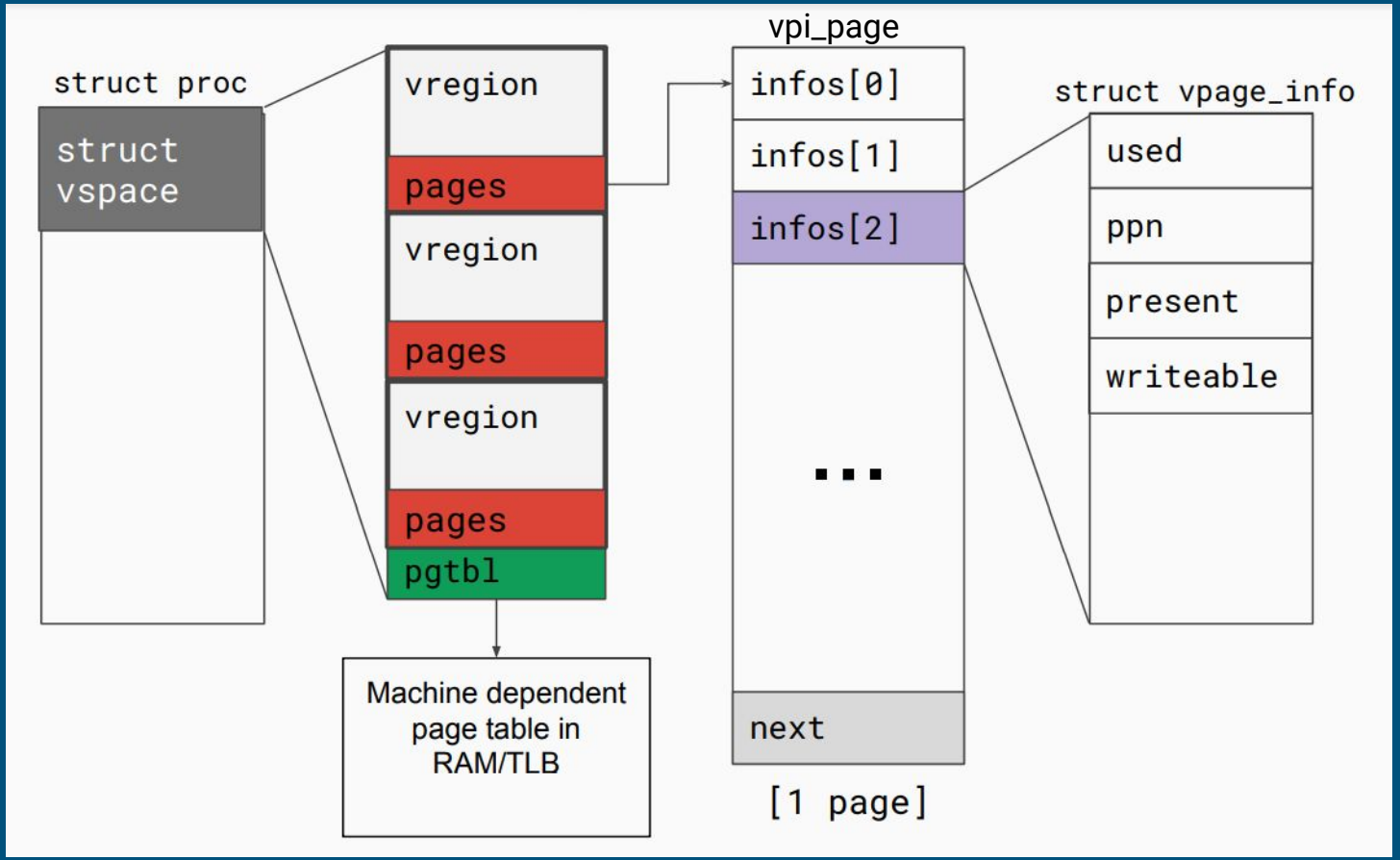
A struct vpage_info describes characteristics of the virtual page that we are pointing to, e.g used, physical page number, present, writable

vpi_page

```
struct vpi_page {
  struct vpage_info infos[VPIPPAGE];  // info struct for the given page
  struct vpi_page *next;              // the next page
};
```

- A vpi_page is a container of vpage_info's
  - (vpi_page = “virtual page info page”).
- A vregion is made up of a linked list of vpi_pages.
  - (vregion can grow dynamically as needed)
- It stores an array of infos plus enough space for a pointer to a "next" vpi_page struct.

vspace Visual Diagram



vregions vs Page Tables

Ok so the vspace is made up of regions and the page table…

- What's the difference between xk's vregions and the page table?

vregions vs Page Tables

- Can you make modifications to struct vpage_info?
- What happens if you make changes to vregions/vpage_info? Is it automatically reflected on the page table?

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.

Vspace Functions

For each question, there is a corresponding function in vspace.c

- Given a virtual address, how do you find which vregion it belongs to?
  - va2vregion
- Given a virtual address, how do you find its metadata (vpage_info)?
  - va2vpage_info
- How do you add a new virtual to physical mapping?
  - vregionaddmap
- How do you update the page table to reflect changes in vregion/vpage_info?
  - vspaceupdate
- How do you flush the TLB?
  - vspaceinstall

Now let’s talk about the lab!

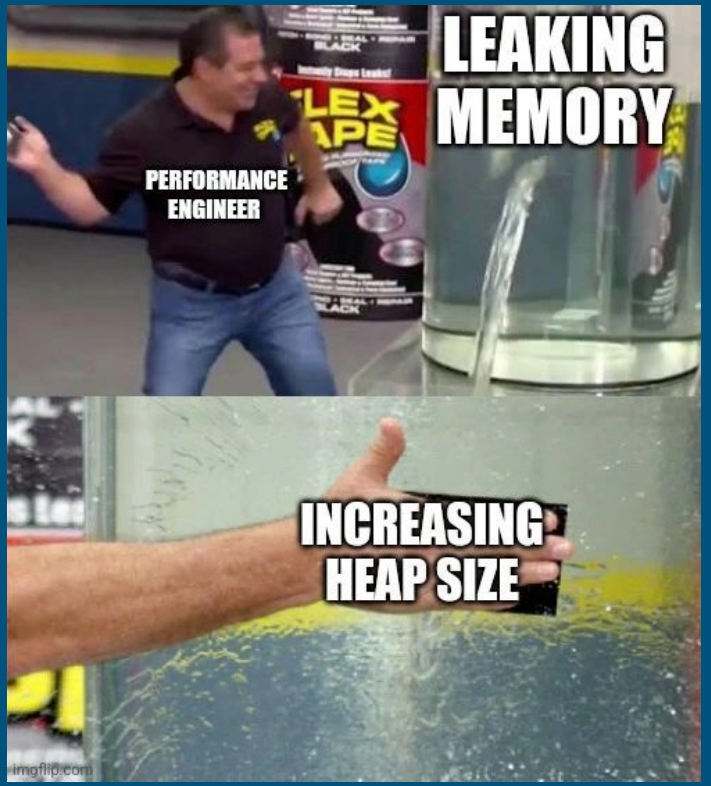
Part 1: sbrk
(“set program break”)
Get more heap space

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

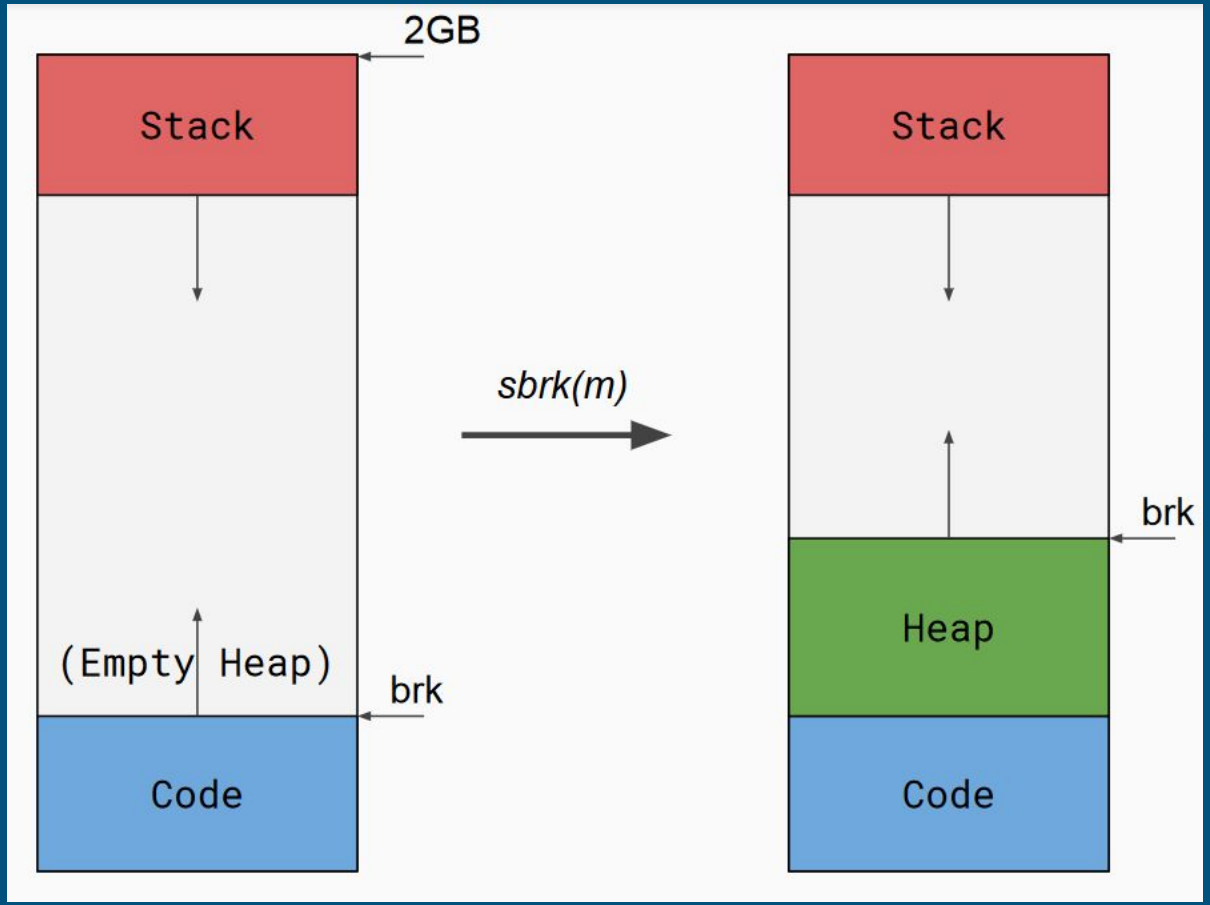
What is the name of the system call that gets more heap space?

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
2GB
Stack
(Empty Heap)
brk
Code
sbrk(m)
Stack
brk
Heap
Code

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

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

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