Project 4: Xv6 VM Layout

**In this assignment, you will try to better understand the virtual memory (VM) layout for XV6. Sounds Simple, but it does require careful understanding of the internal workings of XV6. Recall in prior assignments, you have learned how processes, fork, and exec work, all this knowledge will come in handy for this assignment.**

**Part A: Pointer Dereference -- 200 points**

In xv6, the VM system uses a simple two-level page table. As it currently is structured, user code is loaded into the very first part of the address space. Thus, if you dereference a null pointer, you should not see an exception (as you might expect) ; instead, you will see whatever code is the first bit of code in the program that is running. **(In the latest version of Xv6 public you will still get an exception trap error but it is unique compared to expected)** Try it and see! Report what you find in your Readme file.

Thus, the first fundamental thing you might do in this case is to create a program that dereferences a null pointer. It is simple! See if you can do it. Then run it on Linux as well as xv6 to see the difference.

Thus, the first fundamental thing you might do in this case is to create a program that dereferences a null pointer. It is simple! See if you can do it. Then run it on Linux as well as xv6 to see the difference.

Your job here will be to figure out how xv6 sets up a page table and then change it to leave the first three pages (0x0 - 0x3000) unmapped. The code segment should be starting at 0x3000. Thus, once again, this project is mainly about understanding the code and not writing very much. Look at how exec() works to understand better how address spaces get filled with code and, in general, initialized. That will get you most of the way.

You should also look at fork(), in particular the part where the address space of the child is created by copying the address space of the parent. What needs to change in there?

Remember that the first process is a little unique and is constructed by the kernel directly? Take a look at userinit() and do not forget to update it too.

The rest of your task will be completed by looking through the code to figure out where there are checks or assumptions made about the address space. Consider what happens when you pass a parameter into the kernel; for example, if passing a pointer, the kernel needs to be very careful to ensure you haven't passed it a bad pointer. How does it do this now? Does this code need to change in order to work in your new version of xv6?

One last hint: you'll have to look at the xv6 makefile as well. In there, user programs are compiled so as to set their entry point (where the first instruction is) to 0. If you change xv6 to make the first two pages invalid, clearly, the entry point must be the new beginning of the code segment. Thus, something in the makefile will need to change to reflect this as well.

You should be able to demonstrate what happens when the user code tries to access a null pointer. If you do this part correctly, xv6 should trap and kill the process without too much trouble on your part.

**This is the exciting part, check how your older uniq, head, and ps program works in this scenario. Report your findings.**

**Part B: Stack Rearrangement – 300 points**

*The xv6 address space (after modifications from Part A) is set up like this:*

*USERTOP = 640KB*

*(free)*

*heap (grows towards the high-end of the address space)*

*stack (fixed-sized, one page)*

*code*

*(3 unmapped pages)*

*ADDR = 0x0*

In this part of the xv6 project, you'll rearrange the address in the following manner:

*USERTOP = 640KB*

*stack (at end of address space; grows backwards)*

*... (gap >= 5 pages)*

*heap (grows towards the high-end of the address space)*

*code*

*(3 unmapped pages)*

*ADDR = 0x0*

This will take a little work on your part. First, you'll have to figure out where xv6 allocates and initializes the code, heap, and user stack; then, you'll have to figure out how to change the allocation to be in new places. You'll need to place the stack at the high end of the xv6 user address space instead of between the code and heap.

Some tricky parts: one thing you'll have to be very careful with is how xv6 currently tracks the size of a process's address space (currently with the sz field in the proc struct). There are a number of places in the code where this is used (e.g., to check whether an argument passed into the kernel is valid; to copy the address space). We recommend keeping this field to track the size of the code and heap, but doing some other accounting to track the stack, and changing all relevant code (i.e., that used to deal with sz ) to now work with your new accounting.

You should also be wary of growing your heap and overwriting your stack. In this assignment, you should always leave 5 unallocated (invalid) pages between the stack and heap. These are reserved for supporting memory mapping segments.

The high end of the xv6 user address space is 640KB (see the USERTOP value defined in the xv6 code). Thus your first stack page should live at 636KB-640KB.

One final part of this assignment, which is challenging, automatically growing the stack backward when needed. Doing so would require you to see if a fault occurred on the page above the stack and then, instead of killing the offending process, allocate a new page, mapping it into the address space and continuing to run.

**Now you will test your older uniq, head, ps code with this new version of XV6 and report your findings.**

**You will also report how your FCFS and Priority scheduling work with this new version of XV6, do you observe any noticeable difference. Report your findings – 100 points**

A few important testing criteria

1. Move the code segment and leave the first three pages unmapped: null, null2, null3,
2. Checks syscall arguments with new assumptions about the address space: bounds
3. Move the stack to the top: stack, bounds2, bounds\_str
4. Heap does not grow into stack: heap
5. The stack grows on page fault: stack2, bounds3, stack3
6. Stack does not grow into a heap: stack4, stack5

**Additional Info:**

*Part A*

* Null pointer dereference in Linux and xv6 (Explain what those error codes mean)
* Dereferencing 3 pages (During this phase, basic xv6 should work, such as cat, mkdir, simple hello world)
* Running older codes (uniq, head, ps, test, etc.) on new xv6 (report your findings)

*Part B*

Stack Rearrangement

* 3 pages are unmapped, but the stack is fixed.
* Stack and heap grow, but 3 pages are not unmapped
* Running older code with this new xv6 (Part A + Part B) (report your findings)

**Additional Hints:**

**eip** - This will tell the thread where to start executing from when it returns to userspace

**esp** - This points to the top of the stack. This means that if you did this right, 4 bytes stored at the top of the stack should contain your return address

**eax** in this the thread jumps to a new context and not the one where it was created. Otherwise, *eax* will store the return value of the last system call. To better understand this check your fork system call.

**ebp** is not manipulated by user, but rather by the x86 function call conventions and is usually set to the value of *esp* when the function is called. As such you will usually see this sort of thing in the disassembly of most function calls

push ebp ; Preserve current frame pointer

mov ebp, esp ; Create new frame pointer pointing to current stack top

***ebp* is also useful for stack tracing because it stores the top of the previous function's stack before it is then changed to point to the current stack top**

**Submission: You will zip your new xv6 folder with a detailed Readme File (discussing files updated, findings, and all other relevant information) and submit it on Canvas. Be sure to note in your readme the file changes that were made (file name and functional modifications made) as this assignment does not really have many testing criteria (e.g., showing as output in xv6 can be challenging). As such, for Part B showing where the stack and heap are, how you modify them, and editing the correct files with explicit logic is what is being looked for in part of your report.**