Project 3b: The Null Pointer and Other Tricks

DUE 03/31 at 11:59 PM

Objectives

There are three objectives to this assignment:

- To familiarize you with the xv6 virtual memory system.
- To add a few new VM features to xv6 that are common in modern OSes.

Overview

In this project, you'll be changing xv6 to support a few features virtually every modern OS does. The first is causing an exception to occur when your program dereferences a null pointer; the second is rearranging the address space so as to place the stack at the high end. Sound simple? Well, it mostly is. But there are a few details.

Details

Part A: Null-pointer Dereference

In xv6, the VM system uses a simple two-level page table as discussed in class. 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 will not see an exception (as you might expect); rather, you will see whatever code is the first bit of code in the program that is running. Try it and see!

Thus, the first thing you might do is 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. Thus, once again, this project is mostly about understanding the code, and not writing very much. Look at how `exec()` works to better understand 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?

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. Think about what happens when you pass a parameter into the kernel, for example; if passing a pointer, the kernel needs to be very careful with it, 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 page invalid, clearly the entry point will have to be somewhere else (e.g., the next page, or 0x1000). Thus, something in the makefile will need to change to reflect this as well.

You should be able to demonstrate what happens when 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.

Part B: Stack Rearrangement

The xv6 address space is currently set up like this:

```
code
stack (fixed-sized, one page)
heap (grows towards the high-end of the address space)
```

In this part of the xv6 project, you'll rearrange the address space to look more like what we've discussed in class:

```
code
heap (grows towards the high-end of the address space)
... (gap -- ensure atleast one page)
stack (at end of address space; grows backwards)
```

This will take a little work on your part. First, you'll have to figure out where xv6 allocates and initializes the user stack; then, you'll have to figure out how to change that to use a page at the high-end of the xv6 user address space, instead of one 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 fact, you should always leave an unallocated (invalid) page between the stack and heap.

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

One final part of this project, which is challenging: automatically growing the stack backwards 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, allocating a new page, mapping it into the address space, and continuing to run. Getting this to work will make you into a kernel boss, and also get you those last few precious points.

The Code

The source code for xv6 (and associated README) can be found in **~cs537-2/public/html/xv6.tar.gz** . Everything you need to build and run and even debug the kernel is in there. Start with a fresh kernel, instead of using your old scheduler.

You may also find the following readings about xv6 useful: xv6 book.

Particularly useful for this project: Anything about fork() and exec(), as well as virtual memory.