Intro To xv6 Virtual Memory

In this project, you'll be changing xv6 to support a feature virtually every modern OS does: causing an exception to occur when your program dereferences a null pointer, and adding the ability to change the protection levels of some pages in a process's address space.

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.

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.

Read-only Code

In most operating systems, code is marked read-only instead of read-write. However, in xv6 this is not the case, so a buggy program could accidentally overwrite its own text. Try it and see!

In this portion of the xv6 project, you'll change the protection bits of parts of the page table to be read-only, thus preventing such over-writes, and also be able to change them back.

To do this, you'll be adding two system calls: `int mprotect(void *addr, int len)` and `int munprotect(void *addr, int len)`.

Calling `mprotect()` should change the protection bits of the page range starting at `addr` and of `len` pages to be read only. Thus, the program could still read the pages in this range after `mprotect()` finishes, but a write to this region should cause a trap (and thus kill the process). The `munprotect()` call does the opposite: sets the region back to both readable and writeable.

Also required: the page protections should be inherited on fork(). Thus, if a process has mprotected some of its pages, when the process calls fork, the OS should copy those protections to the child process.

There are some failure cases to consider: if `addr` is not page aligned, or `addr` points to a region that is not currently a part of the address space, or `len` is less than or equal to zero, return -1 and do not change anything. Otherwise, return 0 upon success.

Hint: after changing a page-table entry, you need to make sure the hardware knows of the change. On 32-bit x86, this is readily accomplished by updating the `CR3` register (what we generically call the *page-table base register* in class). When the hardware sees that you overwrote `CR3` (even with the same value), it guarantees that your PTE updates will be used upon subsequent accesses. The `lcr3()` function will help you in this pursuit.

Handling Illegal Accesses

In both the cases above, you should be able to demonstrate what happens when user code tries to (a) access a null pointer or (b) overwrite an mprotected region of memory. In both cases, xv6 should trap and kill the process (this will happen without too much trouble on your part, if you do the project in a sensible way).