Project #3A - Adding NULL pages to xv6 address spaces

1. 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.

2. 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 want to do 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. 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.

3. 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)
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 had overwritten **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.

4. Handling Illegal Accesses

In both the cases above, you should be able to demonstrate what happens when the 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).

5. Running Tests

- Use the following script file for running the tests:
```
prompt> ./test-null-pages.sh
```

- If you implemented things correctly, you should get some notification that the tests passed. If not …

- The tests assume that xv6 source code is found in the *src/* subdirectory. If it's not there, the script will complain.

- The test script does a one-time clean build of your xv6 source code using a newly generated makefile called *Makefile.test*. You can use this when debugging (assuming you ever make mistakes, that is), e.g.:

```
prompt> cd src/
prompt> make -f Makefile.test qemu-nox
```

- You can suppress the repeated building of xv6 in the tests with the '-s' flag. This should make repeated testing faster:
```
prompt> ./test-null-pages.sh -s
```

- You can specifically run a single test using the following command
```
./test-mmap.sh -c -t 7 -v
```
The specifically runs the test_7.c alone.

- The other usual testing flags are also available. See the testing appendix for more details.

- This project will have few hidden test cases apart from the provided test cases.

6. Adding test files inside xv6

- Inorder to run the test files inside xv6, manually copy the test files(*test_1.c, test_2.c* etc…) inside xv6/src directory.(preferably in a different name like *xv6test_1.c*, etc…)
- Make the necessary changes to the Makefile

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _xv6test_1\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
```

```
EXTRA=\
    mkfs.c xv6test_1.c ulib.c user.h cat.c echo.c forktest.c
grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
    printf.c umalloc.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

- Now compile the xv6 using the command "*make clean && make qemu-nox*".

```
prompt> make clean && make qemu-nox
```

- Once it has compiled successfully and you are inside xv6 prompt, you can run the test.

```
$
$ xv6test
```

- You can also add your own test cases to test your solution extensively.
- Once you are inside xv6 qemu prompt in your terminal, if you wish to shutdown xv6 and exit qemu use the following key combinations:
- press Ctrl-A, then release your keys, then press X. (Not Ctrl-X)

Appendix – Test options

The run-tests.sh script is called by various testers to do the work of testing. Each test is actually fairly simple: it is a comparison of standard output and standard error, as per the program specification.

In any given program specification directory, there exists a specific tests/ directory which holds the expected return code, standard output, and standard error in files called n.rc, n.out, and n.err (respectively) for each test n. The testing framework just starts at 1 and keeps incrementing tests until it can't find any more or encounters a failure. Thus, adding new tests is easy; just add the relevant files to the tests directory at the lowest available number.

The files needed to describe a test number n are:

- n.rc: The return code the program should return (usually 0 or 1)
- n.out: The standard output expected from the test
- n.err: The standard error expected from the test
- n.run: How to run the test (which arguments it needs, etc.)
- n.desc: A short text description of the test
- n.pre (optional): Code to run before the test, to set something up
- n.post (optional): Code to run after the test, to clean something up

There is also a single file called pre which gets run once at the beginning of testing; this is often used to do a more complex build of a code base, for example. To prevent repeated time-wasting pre-test activity, suppress this with the -s flag (as described below).

In most cases, a wrapper script is used to call ***run-tests.sh*** to do the necessary work.

The options for run-tests.sh include:

- -h (the help message)
- -v (verbose: print what each test is doing)
- -t n (run only test n)
- -c (continue even after a test fails)
- -d (run tests not from tests/ directory but from this directory instead)
- -s (suppress running the one-time set of commands in pre file)