Lab 2 – Junaid Javed

3.2:

When running int r = 5 / 0; and routing interrupt 0 to go through divzero2, we observe that the print statement inside of divzero2 runs 10 times and then hangs. The reason that divzero2 keep getting called is because of the contents of EIP. When a synchronous interrupt occurs, the return address pushed onto the stack by x86, is the address of the next instruction of the interrupted process to execute upon return. This means, when we run int r = 5 / 0, the address to this instruction is pushed onto the stack. This is useful for instructions that need to be re-executed after an interrupt, like page fault. In our case, the address to the instruction int r = 5 / 0; is pushed onto the stack. When we return from divzero2(), the address is popped off the stack and the system will try to re-execute the instruction thinking that it will pass now. Dividing by zero will always trigger an interrupt so the function will keep going back into divzero2 until we call kill() inside the if statement.

3.3:

When running asm("int $0"); as a way to trigger interrupt 0, we can see that this time the print statements don't repeat. This is because calling an interrupt directly using in line assembly does not pass the EIP of the current instruction but rather that of the next instruction. Say we had

(1) asm("int $0);
(2) asm("int $0);

When (1) runs, the pointer to instruction (2) is pushed onto the stack. This means that when int $0 returns (when divzero2() returns), the program returns to and runs instruction (2) instead of (1) again.

4.4:

To verify my trapped system call implementation, I wrote multiple test cases for each supported system call.

I test xchildrennum() by creating multiple processes resuming them, and checking the return value of xchildrennum(). For xchprio(), I keep track of the old priority and check to see if the output of chprio() matches that. For xsleepms(), I test by putting the process to sleep multiple times and making sure it does not hang.

For testing the isolation/protection, I test by trying to manipulate the EIP value AFTER we switch from user to kernel mode. This seems like the best way to test since a malicious user would want to alter where the program returns to after finishing childrenum(), chprio(), sleepms() since it is technically still in kernel mode. To test it, I try altering the value where EIP would have been stored on the user stack. I set it to an arbitrary number like $0.

If there was no isolation/protection, our expected behavior would be that the function would return to address 0x0 after calculating the pr_children. With isolation/protection, our expected behavior should be the same as if we hadn't altered the return address on the user stack. To

assure that isolation/protection is fully working, I test altering the EIP both with my isolation/protection implementation and without. When I test without, I can see that the program just hangs. This is because there is nothing to run at the address 0x0. When I test with my isolation/protection, we see that the program behaves normally.