Project 2: Writeup

Joshua Bridges

This project started by causing null pointer dereference to throw a trap 14 error. This means that it's causing a page fault and trying to access a memory page that is not currently mapped. In order to get this to throw the proper error I created the simple test code nulltest.c to try to utilize the null pointer. This code works on the base xv6 machine. With the proper test case set up, I started going through the areas where the memory pages are allocated to change the initial location from address 0 to address 4096 aka the next page; since the page size is dictated in mmu.h. The cases where the change needs to occur is when processes are allocated so the first location was the exec.c file. This file has the declaration that "sz = 0" by default, setting the initial location to address 0. As a result, a simple change to "sz = PGSIZE" sets it to the second page and solves the problem in that file.

Next, in the makefile the initial starting location for user processes is specified in the instruction starting on line 149. The "$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^" line specifies this location in memory as 0, so I changed that to 0x1000 to specify the address to 4096 to start. This solves the problem in the makefile.

The last step was the fork call for when processes are duplicated. Following the fork call shows that it calls the copyuvm() function. Looking in this file there is a for loop in the copyuvm function that creates the address space for the child process. This loop starts at address 0, so a change to PGSIZE for the initial value solves the problem for the fork process. Running the test after these changes results in a trap 14 as expected.

The other aspect of the project was to create two system calls to implement mprotect and munprotect linux commands that change the writeable bits of the passed addr to 0 or 1 respectively. I first went through the same steps as Project 1 to implement new system calls.

I started in the syscall.h file adding numbers for the calls and adding them as indexes into the array in syscall.c along with declaring the external system call references. Then I added them to usys.S and added their signatures to the user.h file. Then I added the signatures to the proc.h file to enable their usage in the sysproc.c file. Then in sysproc.c and vm.c I added the actual implementation.

In sysproc.c I added the two system calls to enable them to grab the arguments passed to the kernel using argint and argptr appropriately. Argptr already checks if the address is in the process space so I just needed to add an additional check to see if it is page aligned. For the len we just need to check if the argument is greater than 0. After these conditions are met the system calls call the actual definitions from vm.c.

In vm.c the implementation of the two functions becomes very similar. They first grab the current process structure to access the page table directory, then declare some local variables. The meat comes in the main for loop over the amount of page tables specified by the len variable. For each of the tables within the for loop, the walkpgdir function is called to get the page table entry for the argument passed. Then once the entry is found, the mprotect calls an

exclusive OR to set the writeable bit to 0. The munprotect calls a normal OR in this situation to set the bit to 1. Then after the loop is finished, the lcr3 function is called on the page table directory to update the CR3 register and flush the TLB cache.

With these functions implemented the test cases were designed around testing the functionality. There are three test processes for the aspect of the project: childfailtest, parentfailtest, and systest. Childfailtest protects the file then forks to test if the protection is inherited properly. This case fails when the child tries to write to the protected memory. Parentfailtest protects the file then unprotects it in the child. This test case fails when the parent function falls through the conditionals and tries to write to the protected memory whereas the child still can write since it was changed earlier. The last test, systest, protects and unprotects the memory in all the processes to allow for the processes to write to the memory when it is unprotected. This process works to completion.

The picture below is the output of the 4 tests being run on the final xv6 with everything implemented.

```
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 51
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ nulltest
pid 3 nulltest: trap 14 err 4 on cpu 1 eip 0x1019 addr 0x0--kill proc
$ childfailtest
pid 5 childfailtest: trap 14 err 7 on cpu 1 eip 0x1063 addr 0xb000--kill proc
$ parentfailtest
pid 6 parentfailtest: trap 14 err 7 on cpu 1 eip 0x1089 addr 0xb000--kill proc
$ systest
$
```