John L. Peterson
jlp5729
Prof. Chidambaram
July 11, 2022

Project 3 - Null Space & Threads

Part 3A - Null Space

This section, as referenced in the specification, was mainly figuring out how xv6 lays out the page table. I referenced the xv6 book, primarily Chapters 0-2, to determine how a process gets called via fork() and exec(), as well as where all memory locations are referenced in relation to programs.

My first approach was to make a user program, as suggested, to dereference a null pointer. This can be found in the code as hello.c. After a while of perusing the code and the textbook, my revelation was that xv6 allocates two pages to a process and marks the first as protected. From there, it was just finding the locations that marked where a program's page table entry is referenced and modifying it to point to that second page table. Luckily, xv6 already had a #define PGSIZE so it was easy to make the changes in the code, both in vm.c and exec.c, to start counters and start indices at PGSIZE as opposed to zero. I ran into issues also in the Makefile, only updating the entry point for the ULIB line and not updating forktest; once I updated the line for forktest the program passed all tests.

For the mprotect and munprotect syscalls, the process of adding of system calls was the same as the previous project, including updating all the various files involved in the trap handling for sys calls. Validating inputs to mprotect and munprotect required figuring out the argint call, as well as deciding where to locate the system routine. I chose vm.c as the location for my mprotection routine (one routine works both for protect and unprotect with a control bit) since it was similar in nature to other code dealing with memory allocation. Another thing to figure out was bitwise operations; the xv6 book about PTE flags came in handy for determining which bytes to mask/flip. I ran into errors with accessing the page table; learning that I could check page table alignment by modular division with PGSIZE came in handy (I initially attempted to use the builtin PGROUNDDOWN function but this was not in accordance with specification that addresses already had to be page-aligned).

For testing, I preferred to use existing test cases and step through the program(s) with gdb to determine exactly where errors were going wrong. Setting ample breakpoints allowed me to see values passed into functions and where entries were being invalidated (or improperly accepted); pointers vs. indexing in the C language was still an issue until I figured that out more thoroughly.

I think moving forward, I could do a better job writing test cases instead of relying on gdb and the provided tests.

Part 3B - Threads

My first step was to implement the page fault protections from part A; this, I believed, would help me from making bad references while writing code dealing specifically with page tables and stacks/trapframes. My next step was to set up the shell of system calls and user library, in order to be able properly compile and run the provided tests. Dealing with function pointers was something new to me, so some reading and trial-and-error were required before getting everything specified in the header and code files. I then made another user test, also called hello.c, in order to test passing pointers to variables along with a function pointer and running a passed-in function. Once these were all in place I was able to start digging in more deeply to the structure of fork(), exec(), and wait() and the user stack in xv6.

My goal was to adapt as much code as possible from existing xv6 routines, matching the style that the system uses. This meant that my first attempt for clone() was to go line-by-line through fork() and then exec(), trying to understand what each line did and why it was necessary. The xv6 book helped, specifically chapters 2 and 3 about address space and traps/interrupts.

I set up code for the system call, and chose to implement the functions for both sys_join and sys_clone in sysproc.c, as they are closely related to other process system calls. From there, proc.c was a natural fit for the functions clone() and join() in the kernel space.

I attempted to utilize code from exec.c to set up my new user stack, but was running into kernel panic/crashes. I read more about the user stack, and realized that I did not need to copy argc and argv entries into my user stack for the new thread; it was only necessary to copy in the fake PC, arg1, and arg2.

Originally, I was planning to add an entry to the proc struct with an array of threads running on the process. However, this got complicated due to the need to initialize the array on allocating a process, as well as design decisions. These decisions included whether or not to add the processes to running threads' arrays, and how to remove them without having to rewrite an entire array, and made things too complicated for the specification. This array in proc structure is still in the code as submitted, but not utilized; it could be implemented in the future. I then concluded that a thread counter for each process would be more useful; this attribute would allow the system to know how many threads a user has spawned (but in fact, I never needed to use this attribute, either, because the specification requires the user to call join() on threads the program creates). A future revision could include a method such as join_all or a parameter to the syscall join specifying how many threads they wanted to join, or if they wanted to wait for all threads to finish executing repeating until the thread counter reached zero. A naive approach with the current architecture to join all threads could just be looping in a user program until the user receives a "-1" return value for thread_join().

To address concurrency in the kernel for process threads, I utilized the included spinlock implementation for the process table lock on both join() and clone() in similar code locations to

the wait() and fork() functions. The Ticket Lock structure in the user library was pulled directly from the textbook; the inline assembly for xaddl from the Wikipedia page.

It did not at first appear to be necessary to add more locks to the kernel section, as the file duplication filedup and inode duplication idup as used also use their own lock on the file table and the inode cache, respectively. Unfortunately, the methods used in my implementation of xv6-threads was not sufficient to prevent all race cases. After receiving my first grader result of 32/40, I went back in to attempt to implement a lock table for each parent process. This structure was an array of NPROC length in ptable struct, but unfortunately it did not solve the test. I expanded test 8 to count to 1 million, in case that would make it clear, but either the test would complete or it would freeze. I was unable to pinpoint where in GDB it froze, and since this was the issue, adding more locks would not have helped it. More judicious application of the locks over shared data structures would certainly help, but I was not able to figure out to where I should move them; so I did not submit another version of my code with the revised lock structure.

For testing, I wrote my own test "hello.c" test to test passing in pointers/functions and check basic functionality with print statements. I utilized gdb extensively setting many breakpoints, either stepping through the kernel space, or changing the file and stepping through the user space. This allowed me to track variables and pointers and understand what was being passed when, and why certain conditions were not properly checking the things I needed. Learning the difference between the different argument methods xv6 uses for syscalls was critical, as certain addresses needed to be handled with argint instead of argptr, as they were outside the user space, but one of the tests would not pass until I handled that argument with argptr (making sure the stack pointer was accessible to the user program, otherwise a panic was initiated).

One difficulty I had was wrapping my head around the user vs. kernel structure. I was not sure how to implement the ticket lock in my system calls, but reading more on the xv6 book and Piazza helped me understand better the difference between the user space and the kernel space, and how it applied to xv6 and this assignment. The biggest difficulty in implementation was fully understanding the user stack and how it is laid out, since the book goes over the registers and the kernel stack, but seemed less straightforward about the user stack. I will also need to dig further into the chapters and info about locks and concurrency. I was not able to fully figure out where locks were necessary for the threads.

Going forward, keeping straight the separation between the kernel and the user space will be very necessary and save a lot of headaches. This project gave me a lot of insight into the memory, both kernel and user space, and how processes are executed on xv6. I have learned to be much more careful with validating inputs and understanding which addresses are pointing to where; which I imagine will come in handy implementing mmap in the following assignments.