Description of my system call, and the program that uses it:

My system call is called SYS_getreadcount. All system calls in xv6 are called with a generic function that uses an ID number to pass the proper instructions to the kernel. Getreadcount uses ID number 22. The system call itself is simple, it gets two integer arguments. The first one is the “reset” argument. Accepted values are 0 or 1. 1 Indicates that the system call counter should be reset to 0. The second argument is the ID of the system call the user wants to track. Accepted values are between 1 and 22. When the tracked system call gets changed, the counter gets also reset to 0. You can only track one system call at a time. SYS_getreadcount returns a single integer value, the sysCallCounter. sysCallCounter is an external integer variable, and it gets incremented in syscall.c, the system call itself only either returns it’s value or resets it, and then returns its value.

How to run:

1. Download and extract the xv6-public-master folder.
2. Make sure you have qemu installed. I used the command:
   “sudo apt install -y qemu qemu-kvm libvirt-daemon libvirt-clients bridge-utils virt-manager”
3. Open a new shell in the extracted folder.
4. Build and run xv6 with command “make qemu” in the shell
5. Qemu should now be running, you can run the test program in either Qemu or the shell you built it from.
6. Run the test program with the command “getreadcount” to display how many times the read-systemcall has been called.
    a. The test program can also take one argument. Try using “getreadcount 0” to reset the counter without changing the tracked system call.
    b. You can also give an integer between 1-22 to change which system call is tracked. Changing the tracked system call naturally also resets the counter. The number represents the integer id of the tracked system call. Tracking system call number 22 tracks how many times you’ve called the getreadcount system call. By default, the tracked system call is 5 (sys_read)
    c. Try different commands in xv6, such as “ls”, and then use command “getreadcount” to see how many times these a specific system call has been called.

Description of my work process:

After downloading the files to build xv6 from github (https://github.com/mit-pdos/xv6-public) and getting the emulator (qemu-kvm) to run it via the shell, I could start browsing the source code after building xv6 using the “make” command. Running it was achieved by running “make qemu”, which starts up the system. Whenever changes were made, you need to remake both.

I needed a way call some system call myself, and verify with some prints that it was indeed called. For this I needed to create a custom user program. I created a new file in the source code directory, and added it to the makefile in two different places to get the user program to compile. At first I just made a hello world, which I edited into the final one.

Next I needed to learn how the system calls work. Grep was invaluable here, since you can quickly search for function names, structures or whatever else and find out in which files they are hiding.

With the help of grep, I first found the generic system call function in syscall.c, and later where different system calls are defined, sysproc.c. Once I found the syscall() function, I tried to add some prints to see where system calls are made. There were a lot of system calls. Then I wanted to figure out how to print only when a read system call gets called. The syscall function itself had a clue how to do this, since it had the “unknown system call” print, which prints out the number of the unknown system call. So the number of the system call is in the current process’ trap frame struct, in the eax variable. If I find out which number is read, I can get a print every time read gets called with a simple if statement.

I tried opening syscall.c in Visual Studio Code, and I found out the integer for the read system call by hovering over [SYS_read], the integer is 5. For the duration of the project, I mistakenly used the integer of SYS_open (15) to test my system call, which is the one used in the screenshots. Thankfully the system call works just as well with sys_read as it does with sys_open, but the counts are different. SYS_Read seems to get called for every character you type in the shell, and once after every execution, while SYS_open gets called when files are opened.

I created a new function in sysproc.c, where all the other system calls are defined, and got some errors in make. There were a lot of definitions that needed to be added in different files before I even got it to compile. I needed to add a definition into usys.S, syscall.h, syscall.c to get it to compile.

Now I could test if my new system call was called and if I could detect it. I edited my hello world to call my new system call to test it. Then I edited the print I made earlier to print out when a system call with the integer 22 (my new system call) was called. And I happily noticed that a print did indeed appear when I ran my test program. Now all I needed to do was figure out how to count all reads in a system call, instead of adding a print with an IF-clause to syscall(). I ended up using a global variable, which my system call would simply return when called. Syscall() would increment it by 1 every time it got called by any process where the eax field was 5.

Now I had a working counter that would increment by 1 every time any process calls syscall() with the integer belonging to read()

I wanted to implement the reset function, so I had to research how the parameters get passed to system calls. I had to edit the declarations to include a “reset integer”, which would determine if the counter should get reset or not. You need to use argint() to get the arguments. Once that worked, I could just make another if-clause into the system call, where I can set the counter to it’s original value. I then tested the system call with prints to make sure that the resetting and the number of reads is correct. With the prints, I figured out that read seems to get called for each character you type into the shell, as well as once after each execution. So if I reset the counter, I will get one read immediately after the execution is complete, and 12 additional calls if I type out the command “getreadcount”, so it always gives a minimum of 13 reads.

Screenshots:

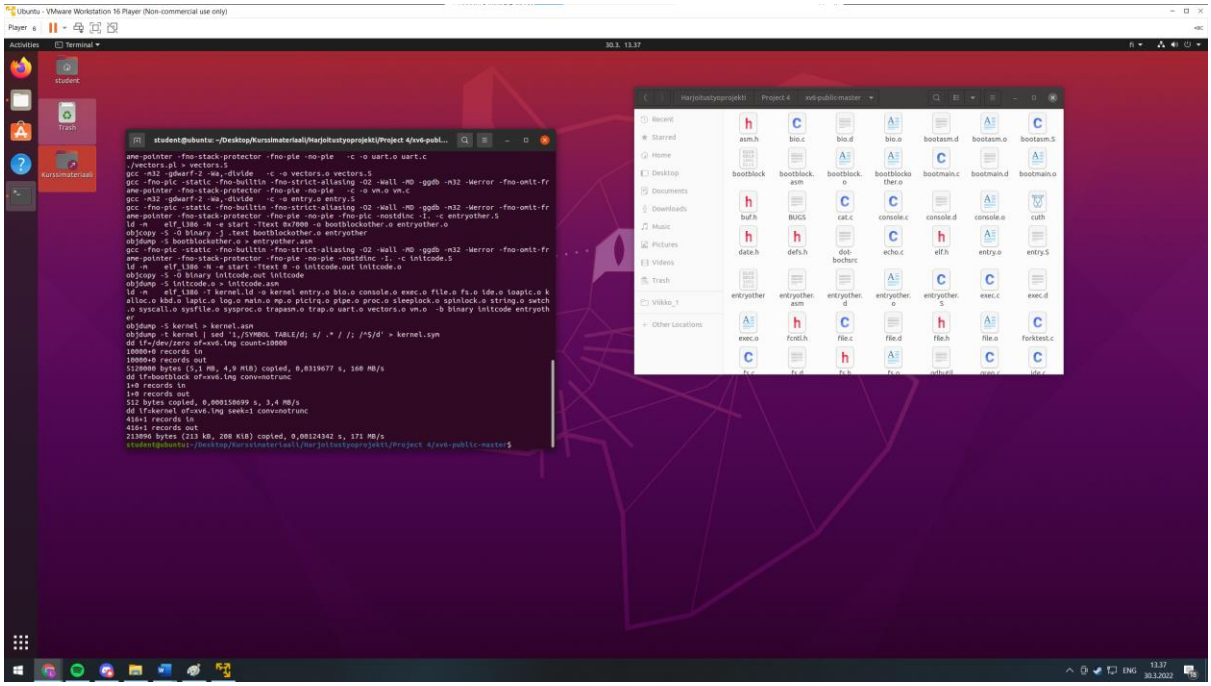

Figure 1. Building xv6 with makefile

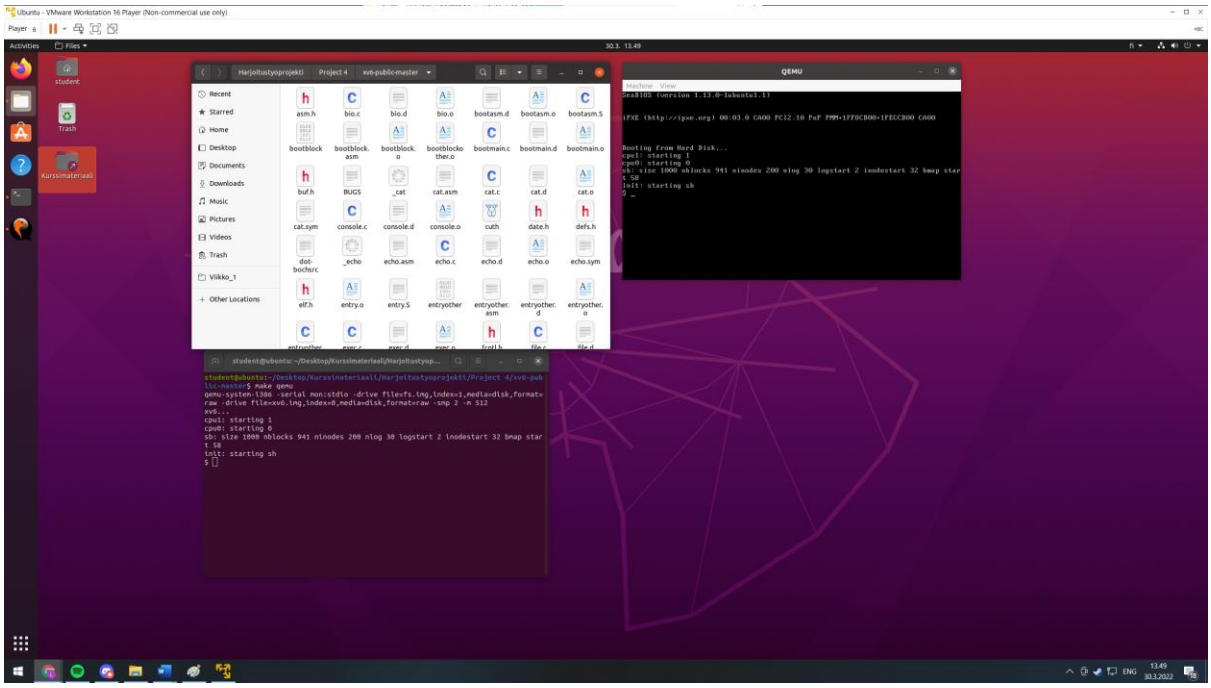

Figure 2. xv6 running on QEMU running in Ubuntu running on VMware Workstation running in Windows 10.

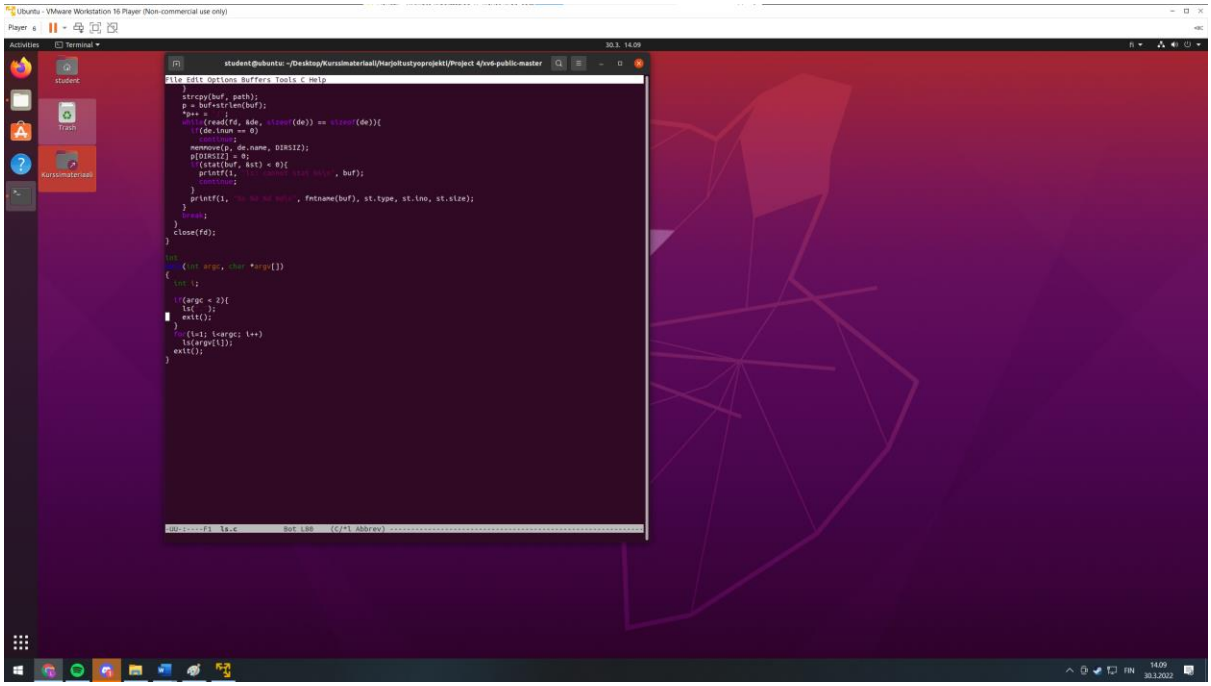

Figure 3 browsing the source code of ls with emacs

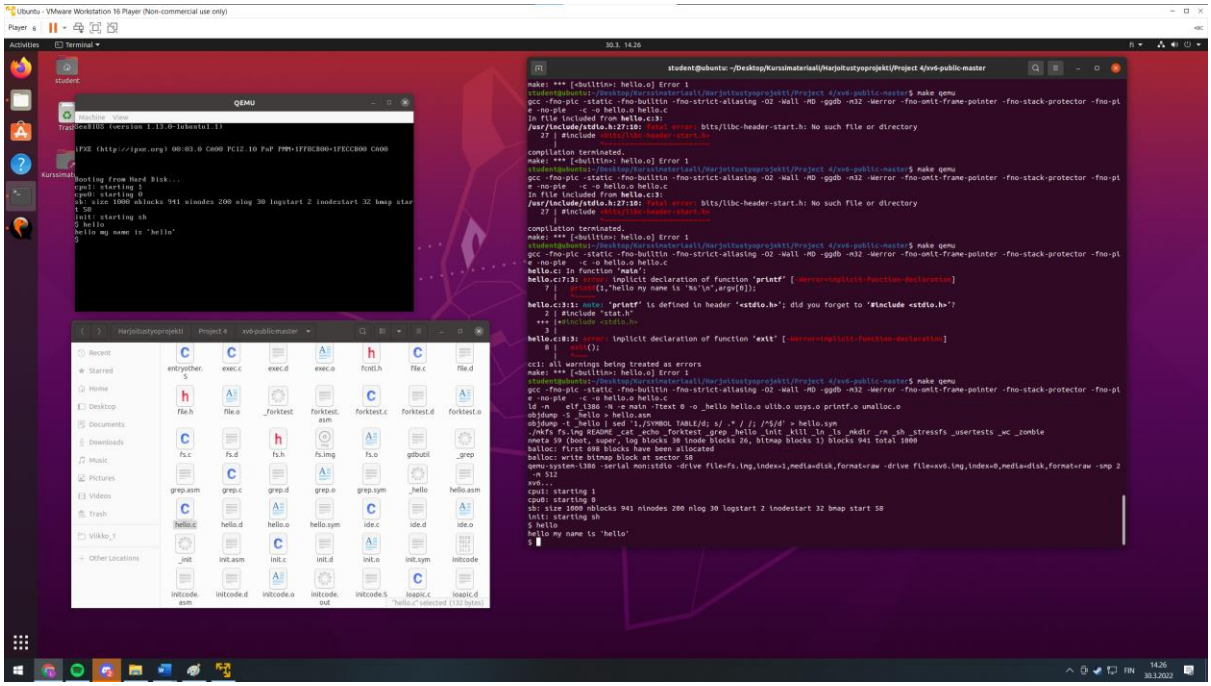

Figure 4 Custom hello world program successfully ran

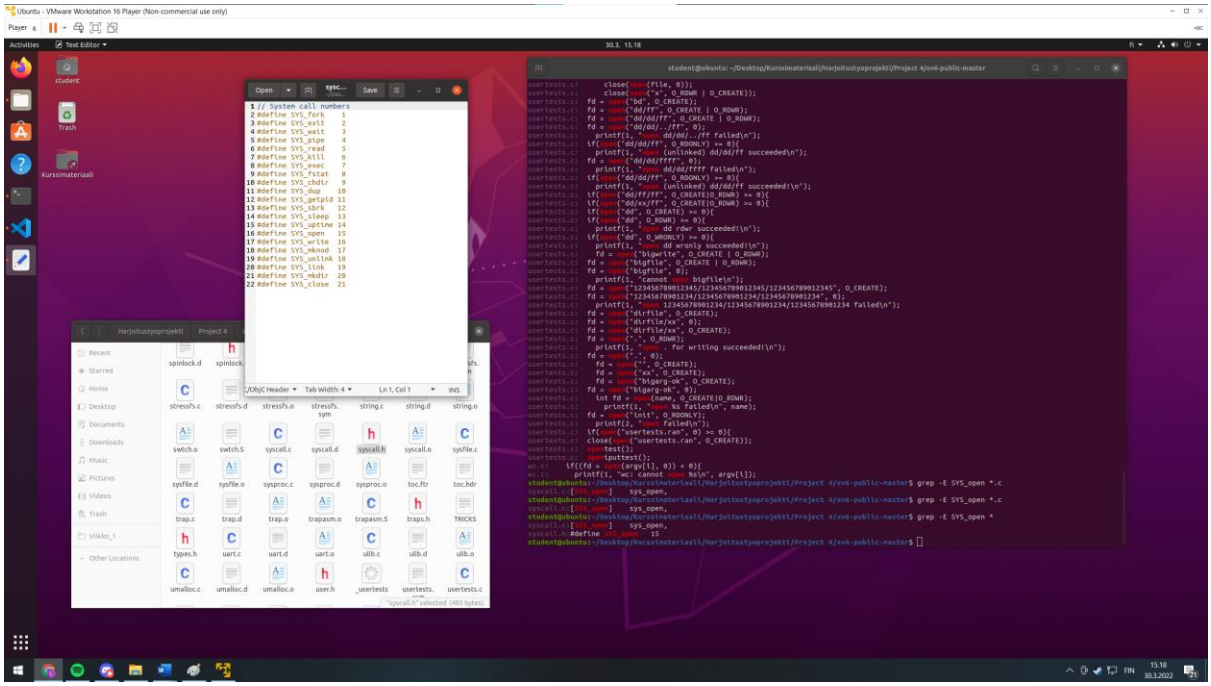

Figure 5 Finding out which syscall number is SYS_OPEN

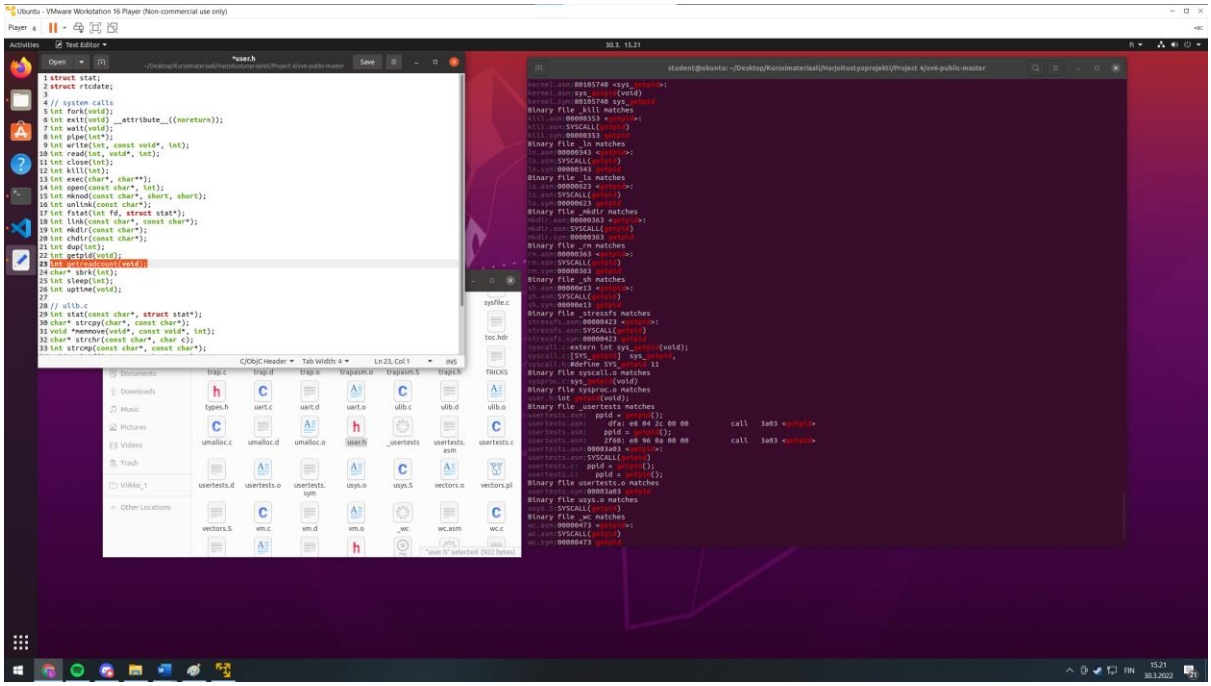

Figure 6 Adding definition of getreadcount to user.h

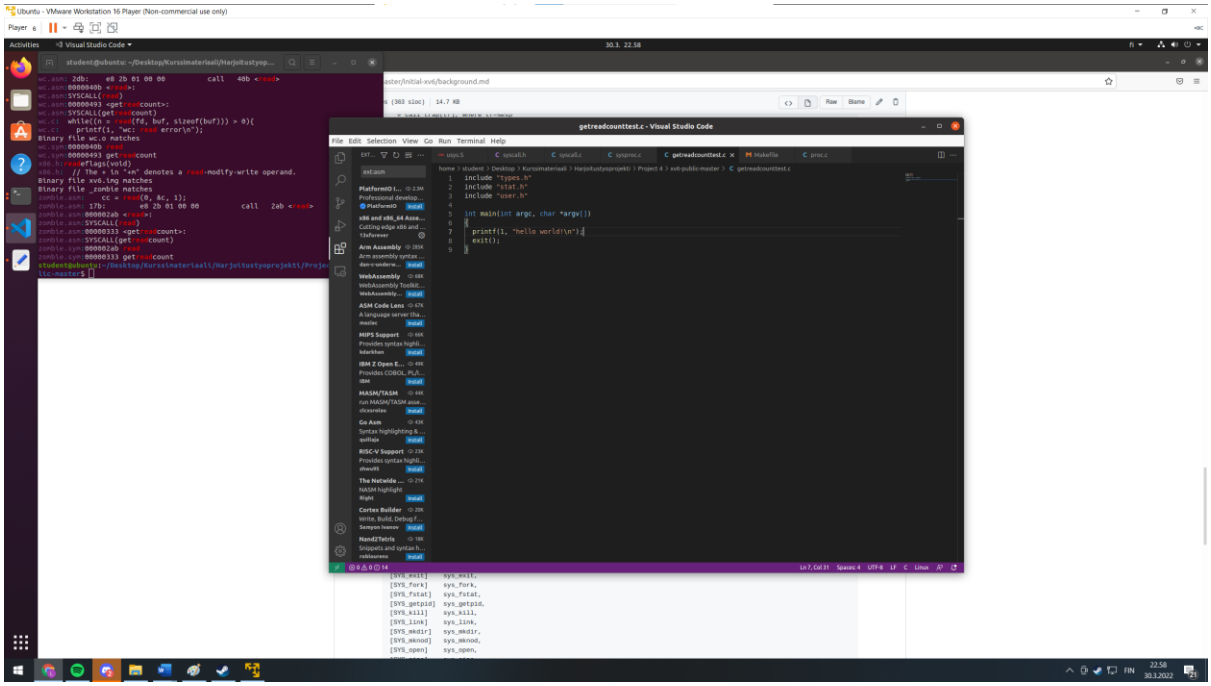

Figure 7 Adding a test program, starting with hello world

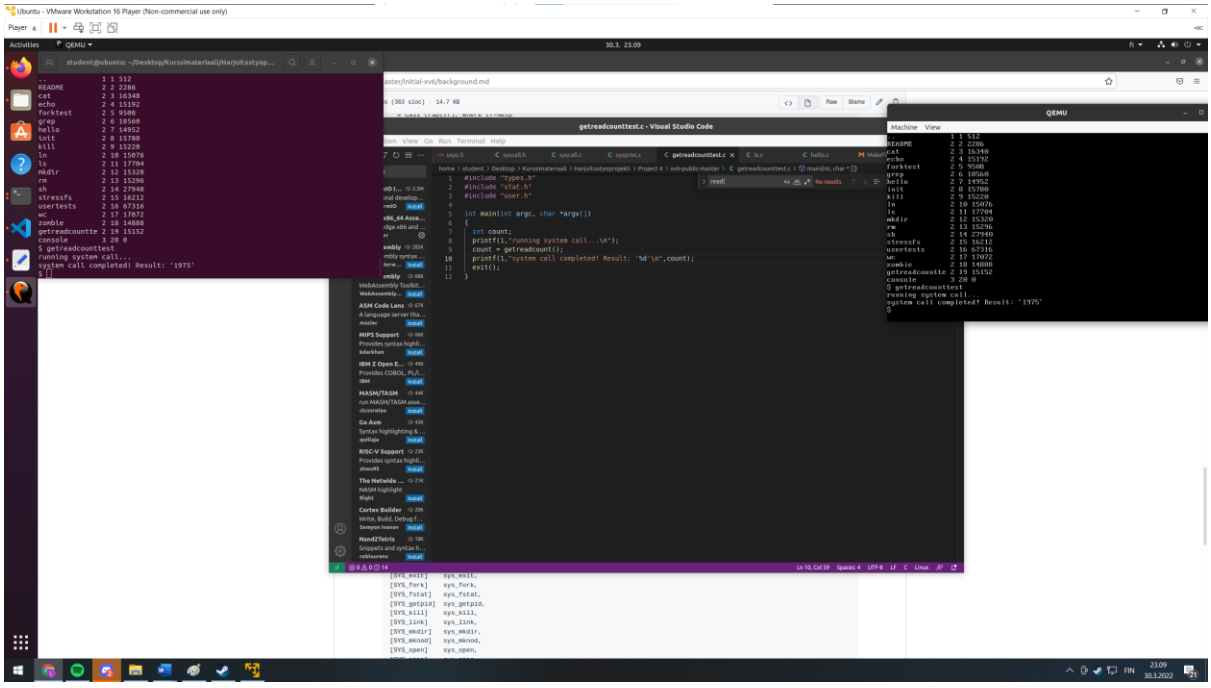

Figure 8 Getting a return value from a custom system call, at this stage its a hardcoded 1975.

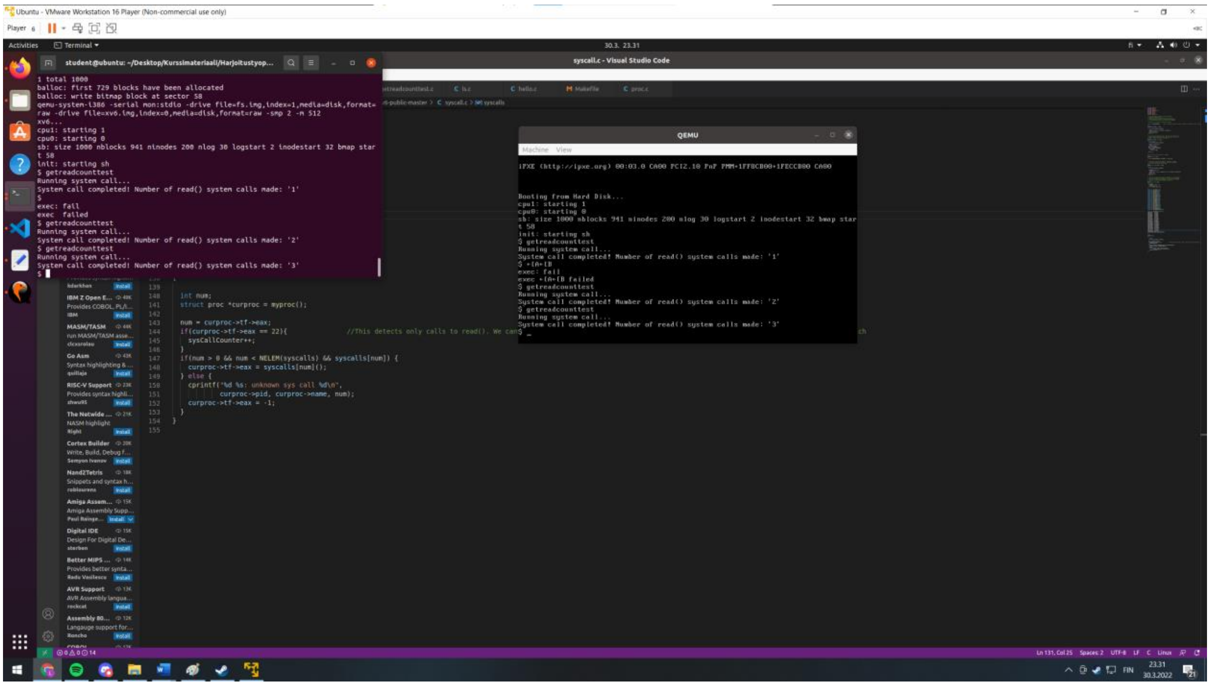

Figure 9 Testing the system call. Changed the integer from 15(open) to 22(getreadcount) to confirm that it indeed counts only when system calls are made

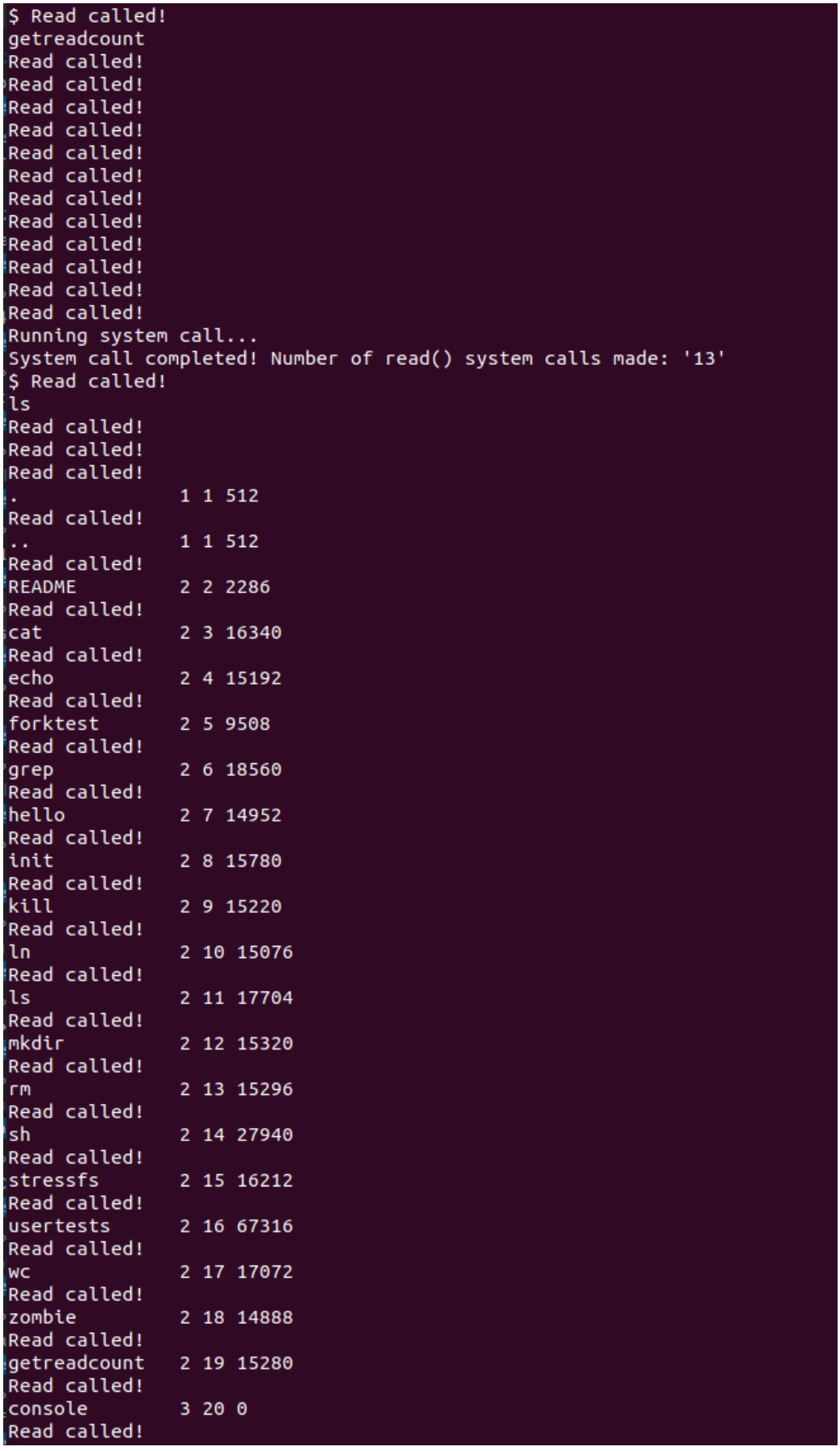

Figure 10 Testing the system call with prints for each call of read()

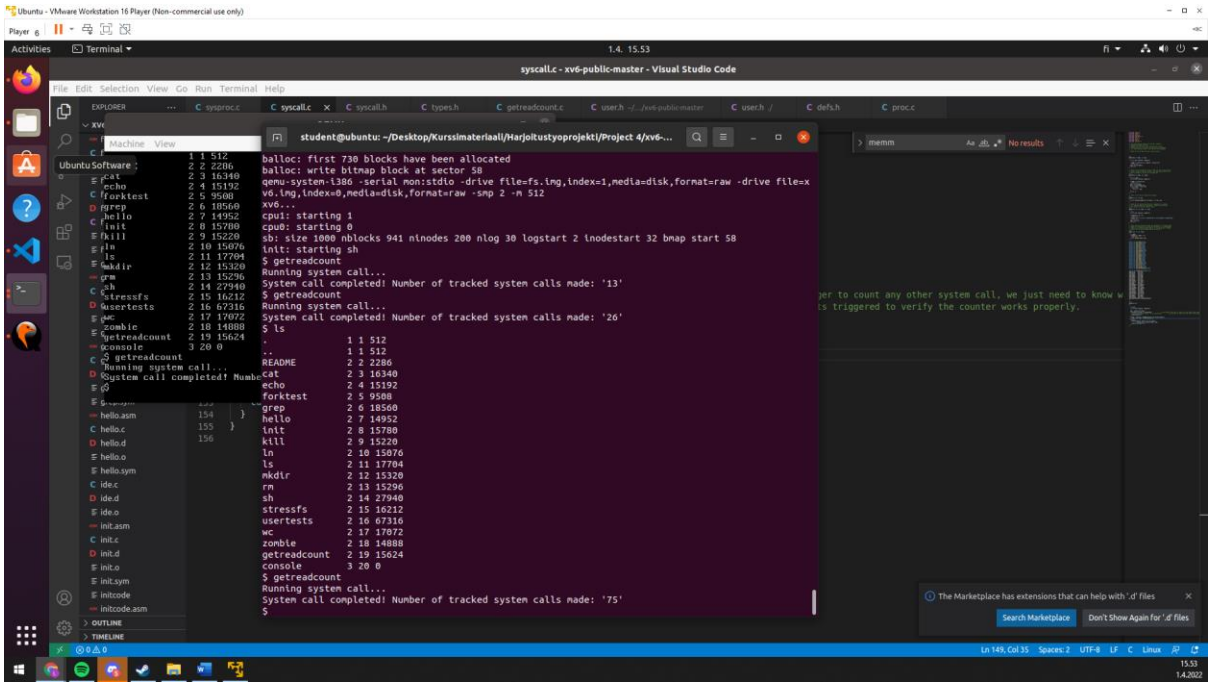

Figure 11 Final output of the system call without changing the tracked syscall.

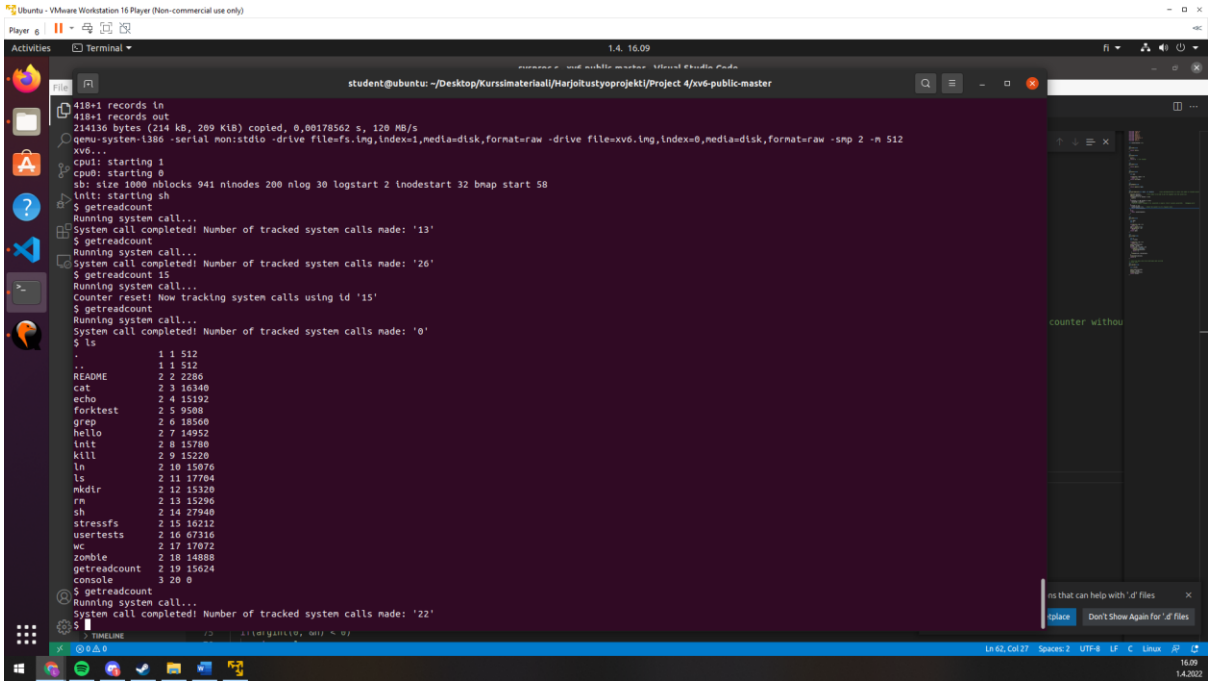

Figure 12 Example output after changing the tracked system call to be number 15 (sys_open)