Project 1b: xv6 Intro

We'll be doing kernel hacking projects in **xv6**, a port of a classic version of unix to a modern processor, Intel's x86. It is a clean and beautiful little kernel.

This first project is just a warmup, and thus relatively light.

The goal of the project is simple: to add one system call to xv6 and create one user-level application that calls it.

The system call is:

- **int getnumsyscallp(void)** returns the total number of system calls that have been issued by the calling process, not including calls to `getnumsyscallp()` itself. The count should be incremented **before** a system call is issued, not after. The system call will simply return the value of a counter that is associated with the calling process.

The user-level application should behave as follows:

- **syscallptest N**. This program takes one argument, **N**, which is the number of system calls (excluding getnumsyscallp()) it makes between calls to `getnumsyscallp()`. Before it calls `exit()`, it should print out two values: the value returned by `getnumsyscallp()` when it is called first within `main()` and the value returned by `getnumsyscallp()` after the **N** system calls have been made.

You must use the names of the system call and the application exactly as specified!

The Code

The source code for xv6 (and associated README) can be found in **~cs537-1/ta/xv6/** . Everything you need to build, run, and even debug the kernel is in there; start by reading the README.

After you have un-tarred the `xv6.tar.gz` file, you can run `make qemu-nox` to compile all the code and run it using the QEMU emulator. Test out the unmodified code by running a few of the existing user-level applications, like `ls` and `forktest`. To quit the emulator, type `Ctl-a x`.

Using gdb (the debugger) may be helpful in understanding code. Look at the Makefile to see how to start up the debugger. Get familiar with this fine tool!

You will not write many lines of code for this project. Instead, a lot of your time will be spent learning where different routines are located in the existing source code. You will end up modifying files that are mostly in the **kernel** subdirectory. The primary files you will want to examine in detail include `syscall.c`, `sysproc.c`, `proc.h`, and `proc.c`.

You may also find the following book about xv6 useful, written by the same team that ported xv6 to x86: book . **Particularly useful for this project: Chapters 0 and 3 (and maybe 4).** Note that our version of xv6 is slightly older than the book's, so you may encounter a difference here and there.

Tips

To add a system call, find some other very simple system call, like `getpid()`, copy it in all the ways you think are needed, and modify it to havethe name `getnumsyscallp()`. Compile the code to see if you found everything you need to copy and change.

Then think about the changes that you will need to make so `getnumsyscallp()` acts like itself instead of `getpid()`.

- You need a counter **per process**. What is the data structure that is associated with each process? Try adding a new field to this structure.
- You need to **initialize** the counter when the process is first created. Where is a good place to initialize a per-process counter? In xv6, a process is created using the `fork()` routine.
- You need to **increment** the counter in the right place. As the video from discussion section described in detail, the `syscall()` procedure is where you want to look. Be sure you don't increment the counter if the system call number corresponds to `getnumsyscallp()`!

For this project, you do not need to worry about concurrency or locking.

You also need to create a user-level application `syscallptest` that calls `getnumsyscallp()` exactly two times. Again, we suggest copying one of the straight-forward utilities that exist in the `user` subdirectory.

Some things to watch out for:

- Calling variants of printf() in your application involves making system calls! Therefore, make sure you save the values returned by `getnumsyscallp()` before your program prints out any values!
- You will see that the initial value returned by `getnumsyscallp()` is not zero (big hint: it should be TWO). Creating a new process from the shell involves making system calls. If you are curious, you can determine what these system calls are by looking through `sh.c`. You will see that one is `exec()` and one is `sbrk()`, which allocates memory to this new process.
- For invoking a number of system calls equal to the argument **N** passed to this application, we recommend invoking a simple system call like `getpid()`.

Good luck! While the xv6 code base might seem intimidating at first, you only need to understand very small portions of it for this project. This project is very doable!