P2: xv6 New System Call - Screenshot

Due Sep 23, 2019 by 11:59pm **Points** 10 **Submitting** a file upload
File Types tiff, jpg, and jpeg **Available** Sep 12, 2019 at 11am - Sep 25, 2019 at 11am 13 days

This assignment was locked Sep 25, 2019 at 11am.

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 small kernel.

This first project is intended to be a warmup, and thus relatively light. 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.

The objectives of this project are:

- Gain comfort looking through more substantial code bases written by others in which you do not need to understand every line
- Obtain familiarity with the xv6 code base in particular
- Learn how to add a system call to xv6
- Add a user-level application that can be used within xv6
- Become familiar with a few of the data structures in xv6 (e.g., process table and file)
- Use the gdb debugger on xv6

This project must be performed alone (without a project partner). Projects in CS 537 cannot be turned in late.

Code Base

You will be using the current version of xv6. Begin by copying this file **~cs537-1/projects/xv6-rev10.tar.gz** and gunzip and untar it.

If, for development and testing, you would like to run xv6 in an environment other than the CSL instructional Linux cluster, you may need to set up additional software. You can read these **instructions for the MacOS build environment. (https://github.com/remzi-arpacidusseau/ostep-projects/blob/master/INSTALL-xv6.md)** Note that we will run all of our tests and do our grading on the instructional Linux cluster so you should always ensure that the final code you handin works on those machines.

After you have obtained the source files, 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`.

You will want to be familiar with the **Makefile** and comfortable modifying it. In particular, see the list of existing UPROGS. See the different ways of running the environment (e.g., qemu-nox or qemo-nox-gdb).

Find where the number of CPUS is set and change this to be 1.

For additional information about xv6, we strongly encourage you to look through the code while reading this **book (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)** by the xv6 authors.

Or, if you prefer to watch videos, the last ten minutes of the
first video (http://www.youtube.com/watch?v=5H5esXbVkC8)

(http://www.youtube.com/watch?v=5H5esXbVkC8)
plus a
2nd video from a previous discussion section (http://youtu.be/vR6z2QGcoo8)

(http://youtu.be/vR6z2QGcoo8)

describe some of the relevant files. Note that the code and project in the videos does not exactly match what you are doing. We always recommend that you look at the actual code yourself while either reading or watching (perhaps pausing the video as needed).

System Calls and User-Level Application

The two system calls you will add are:

- **int getofilecnt(pid)** returns the number of files that the process identified by pid currently has open.
- **int getofilenext(pid)** returns the next file descriptor that will be returned when the process identified by pid calls create(). Note that xv6 allocates file descriptors in the same order that Unix does: it returns the first free descriptor for that process; this is incredibly useful for redirecting stdin, stdout, and stderr in shells and such. Returns -1 if the next call to open by this process will fail (i.e., all file descriptors have already been allocated).

The user-level application should behave as follows:

- **ofiletest N <list of file nums to close and delete>.** This program takes an argument, **N**, which is the number of (empty) files to open (creating if necessary) , followed by a list of files to close and delete. After this work has been done, it should print out two values: the value returned by `getofilecnt(getpid())` and the value returned by `getofilenext(getpid())`. The N files that are opened should be named ofile0...ofileN-1. For example,

```
ofiletest 5 2 3
```

will result in the files ofile0, ofile1, ofile2, ofile3, and ofile4 being opened and then ofile2 and ofile3 being closed. Calling **getofilecnt(getpid())** will return 6 and **getofilenext(getpid())** will return 5. So, the output will be exactly "6 5\n"

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

Implementation Details

The primary files you will want to examine in detail include `syscall.c`, `sysproc.c`, `proc.h`, and `proc.c`.

To add a system call, find some other very simple system call that also takes an integer parameter, like sys_kill, copy it in all the ways you think are needed, and modify it so it doesn't do anything and has the new name . Compile the code to see if you found everything you need to copy and change. You probably won't find everything the first time you try.

Then think about the changes that you will need to make so your system calls act like themselves.

- How will you find out the pid that has been passed to your two system calls? This is the same as what **sys_kill()** must do.
- How will you find the data structures for the specified process? You'll need to look through the **ptable.proc** data structure to find the process with the matching pid.
- How will you find the next index that will be allocated for this process? You'll probably do something similar to what **fdalloc()** in **sysfile.c** does.

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

You also need to create a user-level application **ofiletest.** Again, we suggest copying one of the straight-forward utilities that already exist. Your application will call **open()** and **close()**. You can find the details of how **open()** and **close()** work by looking in **sysfile.c;** note the importance of the second argument to **open()** and that you will want to combine flags defined in **fctnl.h.**

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!

Debugging

Your final task is to demonstrate that you can use gdb to debug xv6 code. You should show the integer value that fdalloc returns the first time it is called after a process has been completely initialized.

To do this, you can follow these steps:

In one window, start up **qemu-nox-gdb**. In another window on the same machine, start up **gdb** (it will attach to the qemu process) and **continue** it until xv6 finishes its bootup process and gives you a prompt. Now, interrupt gdb and set a **breakpoint** in the **fdalloc()** routine. Continue gdb. Then, run the **stressfs** user application at the xv6 prompt. Your gdb process should now have stopped in fdalloc. Now, **step** (or, probably **next**) through the C code until gdb reaches the point just before fdalloc() returns and **print** the value that will be returned (i.e., the value of fd). Now immediately quit gdb and run **whoami** to display your login name.

To sanity check your results, you should think about the value you expect fdalloc() to return in these circumstances to make sure you are looking at the right information. What is the first fd number returned after stdin, stdout, and stderr have been set up?

If gdb gives you the error message that fd has been optimized out and cannot be displayed, make sure that your Makefile uses the flag "-Og" instead of "-O2". Debugging is also a lot easier with a single CPU, so if you didn't do this already: in your **Makefile** find where the number of CPUS is set and change this to be 1.

Take a screenshot showing your gdb session with the returned value of fd printed and your login name displayed. Submit this screenshot to Canvas.

Handing It In

Remember there are two handin steps.

For your xv6 code, your handin directory is `~cs537-1/handin/LOGIN/p2` where `LOGIN` is your CS login. Please create a subdirectory `~cs537-1/handin/LOGIN/p2/src.`

Copy all of your source files (but not .o files, please, or binaries!) into this directory. A simple way to do this is to copy everything into the destination directory, then type `make` to make sure it builds, and then type `make clean` to remove unneeded files.

```
shell% mkdir ~cs537-1/handin/LOGIN/p2/src
shell% cd ~cs537-1/handin/LOGIN/p2/src
shell% make
shell% make clean
```

Test scripts are now available.

For your screenshot of your debugging session, upload your image here in Canvas.