# P2: xv6 New System Call - Screenshot

Re-submit Assignment

**Due** Feb 15 by 11:59pm **Points** 10 **Submitting** a file upload
**File Types** tiff, jpg, jpeg, and png **Available** after Feb 2 at 11am

Updates:

- [Feb. 12] Add notes on locking.

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)
- Use the gdb debugger on xv6

**This project must be performed alone (without a project partner). You have a total of THREE slip days that can be used throughout the semester for late projects; you can use some of those days for this project, but we hope you can save them!**

## Code Base

You will be using the current version of xv6. You could find a copy of xv6 source code in `~cs537-1/projects/xv6.tar.gz`. Copy it into your private directory and untar it.

```
prompt> cp ~cs537-1/projects/xv6.tar.gz /path/to/your/private/dir
prompt> tar -xvzf xv6.tar.gz
```

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`. With `ls` inside the emulator, you'll be able to see a few other applications that are available (as well as files that have been created within the xv6 environment).

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 through make (e.g., `qemu-nox` or `qemu-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) plus a **2nd video from a previous discussion section** (https://www.youtube.com/watch?v=vR6z2QGcoo8) describe some of the relevant files. Note that the code and project in the videos do 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).

## Debugging

Your first task is to demonstrate that you can use **gdb** to debug xv6 code. You should show the integer value that `fdalloc()` (in `sysfile.c`) 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** (using make). In another window where you are logged into 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 (Ctrl+C for Windows) gdb and set a `breakpoint` in the `fdalloc()` routine. Continue gdb. Then, run the `stressfs` user application at the xv6 prompt since this will cause `fdalloc()` to be called. 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.

## System Calls and User-Level Application

You will add two related system calls to xv6:

- `int getnumsyscalls(int pid)` returns the number of system calls that the process identified by pid has completed (not just initiated); calls to `getnumsyscalls()`, `getnumsyscallsgood()`, `fork()`, `exec()`, and `sbrk()` should not be counted in that total*. Returns -1 if pid is not a valid pid.
- `int getnumsyscallsgood(int pid)` returns the number of system calls that the process identified by pid has completed successfully (i.e., with a return code that is not -1); calls to `getnumsyscalls()`, `getnumsyscallsgood()`, `fork()`, `exec()`, and `sbrk()` should not be counted in this total. Returns -1 if pid is not a valid pid.

So that you can test your system calls, you should also create a user-level application with the following behavior:

- `syscalls N g`. This program takes two arguments: **N**, which is the total number of system calls that it should make, and **g**, which is the number of those which should be successful. Note that **N** and **g** should >= 1 because you always need a syscall `getpid()` to know the pid of the current running process. You can choose to make any system calls that you know will be successful or unsuccessful; you can perform those system calls in any order you choose. After this work has been done, it should print out two values: the value returned by `getnumsyscalls(mypid)` and the value returned by `getnumsyscallsgood(mypid)`. For example, if you run the program as "syscalls 20 5" the output should be exactly "20 5\n". You can handle errors (e.g., an incorrect number of arguments or the number of good system calls is greater than the total number of calls however you choose) but we won't test it.

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

*Note:

- Typically, `fork()`, `exec()`, and `sbrk()` will be called during the process creation (before the `main()` of the user-level application `syscalls` is invoked); for simplicity, we exclude them here. You should also be careful when calling some library functions as they may also invoke syscalls (e.g. `malloc()` might call `sbrk()`). Try to avoid them before printing the final result.
- A valid pid indicates any pid that is in-use which includes the zombie processes. A process is in the zombie state after it exits and before it is waited for by its parent process.

## Implementation Hints and Details

The primary files you will want to examine in detail include `syscall.c`, `sysproc.c`, `proc.h`, and `proc.c`. You might also want to take a look at `usys.S`, which (unfortunately) is written in assembly.

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 track the two counters for each process? You will want to add both of these new counters to a structure that already exists for each process. Your new system calls will simply return the value of one of the two counters.
- Make sure you increment the appropriate counter after the system call (except for those we have said to exclude) has returned and check its return value to make sure it isn't -1.
- 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.

For this project, you do not need to worry about concurrency or locking. **[updated Feb. 12] You still need to have lock acquire and lock release code (just as what kill() does in proc.c), but you are not required to fully understand them. See this piazza note (https://piazza.com/class/kjn4sz4kq7t2d2?cid=216) for more details.**

To create the user-level application **syscalls** we again suggest copying one of the straight-forward utilities that already exist.

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!

## 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/ontime/src`.

Copy all of your source files (but not .o files 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.

```
prompt> mkdir ~cs537-1/handin/LOGIN/p2/ontime/src
prompt> cd ~cs537-1/handin/LOGIN/p2/ontime/src
prompt> make
prompt> make clean
```

See p1 specification for the late submission/slip days protocol.

All the public tests are provided at *~cs537-1/tests/p2*. To read more detail about how to run the tests, you can execute the command `cat ~cs537-1/tests/p2/README.md` on any CSL machine. Note that there will be a small amount of hidden test cases (20%), and therefore you are encouraged to create your own test cases instead of relying on the public tests.

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