Assignment 8: xv6 Hacking

Due: Tuesday, April 4, 10pm

Starter code: See Assignment 8 on Canvas for the starter code link. Yes, it is the entire xv6 source code.

Submission: This is a **pair assignment**, but you can work alone, if you so choose.

Submit the contents of your repository via Gradescope. See Deliverables below for what to submit. If you are working with a partner, do not forget to include their name with the submission.

There will be no autograder for this assignment ahead of the deadline. Read the requirements and run tests locally.

The purpose of this assignment is for you to explore the xv6 source code and find the appropriate things to modify to implement a system call collecting stats about open files.

After this assignment, you should be familiar with:

- The execution flow of a system call.
- Where data about open files is stored in the kernel data structures.
- How process-related data is stored and handled in the kernel in general.

Before you start, make sure you are able to run xv6 on your machine. Follow the directions given in Lab 10.

Since familiarizing yourself with an unknown source code base might be a little daunting, we have generated an interlinked browsable version of xv6 for your convenience.

Note: The goal is for you to search around and find stuff. It would be best if you avoid posting exact function names or the line numbers that need to be changed to complete this assignment on Piazza or anywhere else.

Task

For this assignment you'll be adding a system call which returns a struct containing the following information:

```
struct iostats {
    uint read_bytes;     // the total number of bytes read
    uint write_bytes;    // the total number of bytes written
};
```

The system call should have the following signature:

```
int getiostats(int fd, struct iostats* stats);
```

Given a file descriptor, `getiostats` gets the total number of bytes read and written on that file descriptor since it was opened. This information is returned in the user-supplied `iostats` structure. Returns 0 on success, or `-1` on failure (e.g. bad file descriptor).

Plan

For this task, you will need to do the following:

1. Modify the structure xv6 uses for open files to track the number of bytes read and written.
2. Modify the implementations of the appropriate system calls to initialize and update these counters.
3. Add the new `getiostats` system call to read these fields.
4. Add the `struct iostats` definition to `stat.h`

Hint: Look at how the other system calls that use a file descriptor work.

Setting up for the tests

The three test programs included with the starter code—`test1.c`, `test2.c` and `test3.c`—are not built into the xv6 image by default, since they rely on a system call that doesn't exist yet.

Make sure to add them to the appropriate list in the xv6 `Makefile`.

Manual Testing

- Run xv6 with `make qemu-nox`. This will start the QEMU emulator with xv6 in your current terminal.
- Run `test1`, `test2`, `test3`, or any additional test program you choose to write for debugging.
- Press `Ctrl-a x` (while holding `Ctrl`, press `a`, then press `x` without any modifiers) to exit the QEMU/xv6 session.

Debugging

- Plan A: Use `cprintf` from kernel code to print debug messages.
- Plan B: Use the `make qemu-gdb` command to run the whole system in GDB.

Deliverables

All Tasks Implement your modification to the OS sources in the `xv6` directory.

Commit the code to your repository. Do not include any executables, `.o` files, `.img` files, or other binary, temporary, or hidden files (unless they were part of the starter code).

Once you are done, remember to submit your solution to Gradescope and do not forget to include your partner.

Hints & Tips

- Part of this homework is to understand enough of the xv6 source code to be able to perform a few modifications. You need to find your way around. Parts of the code look different from a typical C program.
- Understanding how system calls in xv6 work and how they are written is essential. There are some specifics that make them different from just writing a C function. Look at the existing system calls. The best is to find a syscall that takes similar arguments to the one you are asked to write.
- The overall amount of code you need to add is very small. Still, make sure it reads well and follows good coding practices.
- We have provided some tests. Take a good look at what they test and what is the expected output.
- Exploring an unknown source code base might be a little daunting. You can use the `grep` command for basic text searching. `ctags` might be useful for quickly jumping between function uses and definitions in editors like `vim` or `emacs`. You can also use the linked browsable source code. Some IDEs provide their own code-browsing functionality.