CS/SE 4348 F18 Operating Systems
Project 2: Introduction to Kernel Programming
Due date 23rd September, 2018

Welcome to Kernel programming. Read the project description below carefully.

You can do this project individually or with a partner. You cannot share your work with anyone other than your project partner.

This project can be done only in **csjaws.**

Add a system call to xv6

In this project, you will write a new system call getreadcount() for xv6 OS. The return type and parameters are:

```
int getreadcount(void)
```

<- name of function you will put in the /user folder of xv6
<- this function is the "wrapper" for a new function you will add to the /kernel folder of xv6

The call returns the total number of times the `read()` system call has been invoked since the time the kernel was booted. Note that the kernel does not keep a counter. You have to add it.

https://www.youtube.com/watch?v=vR6z2QGcoo8
44:07 - gives the solution to this assignment for counting syscalls.
You'll have to use a if statment to only count read() calls.
46:16 - shows you how to check the syscall name from the trap frame
47:20 - some humor lol

Background

To understand how to build and run xv6, and about system call implementation watch the discussion video, by Prof. Remzi. ~~You should also read (several times) this background section. Chapter 3 of the xv6 book contains details on traps and system calls. However, you may not need to understand most of the low level stuff (on interrupt tables and trap frames) to implement this project.~~

Hints

You need to modify a number of files to implement this project. Carefully look at these files: syscall.h, syscall.c, sysfile.c, user.h, and usys.S. Study how other system calls such getpid(), uptime() are implemented. You have to do something similar to those for this project.

xv6 Source code

The xv6 source code for this project is /cs4348-xv6/src/xv6.tar.gz. Copy this file to your local working directory for this project and extract the source code tree using the command

```
tar –zxvf xv6.tar.gz
```

Building and Running xv6

In the xv6 directory, you will find:

```
FILES .gitignore include kernel Makefile Makefile~ README
tools user version
```

Look at FILES using vim for a description of each directory.

Compile the xv6 kernel:

In the xv6 directory, type make. When the compilation is complete, you should now find two additional files in the xv6 directory:

```
fs.img xv6.img
```

Run xv6 on QEMU:

Xv6 is developed for x86 architecture. To run on csjaws, you need an emulator. You will use QEMU. The command to run xv6 on QEMU is:

```
$ make qemu
```

You may see a couple of warnings, then you should see the following output lines as xv6 boots up:

```
xv6...
lapicinit: 1 0xfee00000
cpu1: starting
cpu0: starting
init: starting sh
$
```

Take a tour of xv6:

You are now in a xv6 shell. Type **ls** to see the list of programs that you can run. You should see a few familiar commands such as 'cat' and 'mkdir'. Try,

```
$ cat README
```

Now try running the rest of the programs to get a feel for what they do. If a program doesn't exit on its own, you can kill it by typing:

control+d

Exit xv6 and QEMU:

You can exit xv6 by typing:

```
control+a, then release both keys and type x
```

Run xv6 again, this time with GDB remote debugging enabled:

The GNU Debugger (GDB) will be one of your best friends this semester. xv6 allows you to "look inside" while it is running using GDB. This requires two terminals: one to run xv6 and another to run GDB with remote debugging.

First, run xv6 with remote debugging enabled:

```
$ make qemu-gdb
```

You will notice that this time xv6 hangs during the boot-up process. It is waiting for you to make a remote connection using GDB. In another terminal window, in the xv6 directory, type:

```
$ gdb kernel/kernel
```

The first time you will probably get a warning that says:

...auto-loading has been declined by your `auto-load safe-path'...

Fortunately, it also tells you how to fix the problem:

```
To enable execution of this file add
    add-auto-load-safe-path /<path-to-xv6>/xv6/.gdbinit
line to your configuration file "/home/<username>/.gdbinit".
```

```
To completely disable this security protection add
    set auto-load safe-path /
line to your configuration file "/home/<username>/.gdbinit".
```

Go ahead and quit out of GDB so that you can follow the message instructions. You'll probably have to create a .gdbinit file in your home directory (there is already a separate .gdbinit in your xv6 directory, which you should NOT touch--leave it be). You should create (or, if it already exists, modify) /home/<username>/.gdbinit as suggested in the message. We recommend using the add-auto-load-safe-path method, which we have highlighted in red above.

Once you have fixed your .gdbinit file, you can run GDB again:

```
$ gdb kernel/kernel
```

This time, when GDB starts up, you should see the following lines in the output:

```
Connecting to QEMU
(gdb) target remote localhost:25273    (<--port number may differ)
```

Now let's insert a breakpoint in the exec function. This kernel-level function is invoked whenever a new process is executed. In GDB, you can set the breakpoint by typing:

```
(gdb) b exec
```

We are now ready to let xv6 finish booting. In GDB, let xv6 continue by typing:

```
(gdb) c
```

In your xv6 terminal, you should see the system begin to boot:

```
xv6...
lapicinit: 1 0xfee00000
cpu1: starting
cpu0: starting
```

Then it hits the exec function for the first time, triggering the breakpoint. In GDB, you should see:

```
Breakpoint 1, exec (path=0x1c "/init", argv=0xff0e98) at
kernel/exec.c:11
11 {
```

Notice the part that says: path=0x1c "/init"
This means that exec is about to execute the init process. We have just discovered the first process that xv6 executes during boot-up!

Keep typing 'c' or "continue" in GBD until xv6 reaches the shell prompt, noting the process(es) executed along the way. Once at the shell prompt, you can type an xv6 command to run a program, which should trigger your breakpoint again. You can then type 'c' in GDB to allow the program to run to completion. NOTE: Some programs may behave differently when remote debugging is in use.

You can now repeat your previous tour of xv6, but this time you can set breakpoints and use other GDB features to do some more thorough snooping around.

Exit xv6, QEMU, and GDB:

You can exit xv6 and QEMU as before by typing control+a then x. You can exit GDB by typing q and then enter.

Testing

Sample programs to test your implementation is available at `/cs4348-xv6/src/testscripts/p2/`. Copy these programs to your local .../xv6/user/ directory. Now to run these test programs you need to compile them and rebuild the xv6 kernel with them. To compile them, make appropriate changes to the makefile.mk in that directory, and rebuild xv6. (Refer to makefile.mk in /cs4348-xv6/src/testscripts/p2/ to find out what changes to make. See in the definition of USER_PROGS.) Now start xv6 to run the test programs readcount1, readcount2, and readcount3.

Submission

Copy your entire source code tree under xv6 to the directory /cs4348-xv6/xxxyyyyyy/p2. Then change to this directory and ensure that your test programs still work. Finally, run 'make clean' to remove the *.o files and the kernel image files.

If you have worked with a partner, only one of you need to submit the files. But, both of you should create a text file named PARTNER in /cs4348-xv6/xxxyyyyy/p2 and save your partner's name and netid in the file.