Operating System
CS 370
Spring 2019

Project 1

Due Date: 11:59 PM February 4th, Tuesday 2019
Must be submitted by the deadline

Total Points: 50

Graduate Assistant: Pradip Maharjan
Office Hours: MoWe 12PM-2PM
Teaching Assistant: Luis Maya-Aranda
Office Hours: TuTh: 10AM-12PM

Course Web Page: http://osserver.cs.unlv.edu/moodle/

Objective: Become familiar with the **xv6** Teaching Operating System, shell organization, and system calls

Introduction:

The **xv6** OS is an educational OS designed by MIT for use as a teaching tool to give students hands-on experience with topics such as shell organization, virtual memory, CPU scheduling, and file systems. Although **xv6** lacks the full functionality of a modern OS, it is based on Sixth Edition Unix (also called V6) and is very similar in design and structure.

Resources:

The following resources provide more in-depth information regarding **xv6**. They include the **xv6** reference book, source code (pdf), and a tutorial on running and debugging.

1. **xv6** Reference Book: https://pdos.csail.mit.edu/6.828/2016/xv6/book-rev9.pdf
2. **xv6** Source Code PDF: https://pdos.csail.mit.edu/6.828/2016/xv6/xv6-rev9.pdf
3. Running and Debugging Tutorial: http://zoo.cs.yale.edu/classes/cs422/2010/lec/l2-hw

Project:

Complete the following steps.

- Install **xv6** Teaching Operating System
  - See installation instructions
- Become familiar with **xv6** structure
  - Focus on shell organization
- Create process to execute external process
- Create a new user program to execute basic shell scripts

Submission

When complete, submit:

- Zip the entire **xv6** folder and submit (not qemu).
- A copy of the ***zipped xv6 folder*** via the class web page (assignment submission link) by 11:59 PM 4th February, 2019.

We will check for code similarities using MOSS. Please do not download code from the internet or distribute solutions anywhere including GitHub, Bitbucket, and any other public code publishing platforms.

Submissions received after the due date/time will not be accepted.

Part 1: 20 Points

Installing xv6

The following instructions (provided by Professor Jorgensen and edited by former UNLV students) walk you through the process of setting up xv6 within Ubuntu 14.04.

1. Download and Install VirtualBox
    a. Use default/recommended RAM during installation.
    b. **IMPORTANT**: Allocate at least 20GB of disk space to the machine during set-up. Set up Ubuntu in VirtualBox. If you use the default 8GB setting, you are more likely to run of space.
    c. **IMPORTANT**: Allocate as many CPUs as possible, before installing Ubuntu in VirtualBox the amount of CPUs can be changed.
    d. Use all other recommendations during installation.
2. Download **Ubuntu 14.04** 32-bit or 64-bit **(Do not install higher version of Ubuntu)**
3. Open a terminal and execute the following commands:
    a. sudo apt-get update
    b. OPTIONAL: sudo apt-get install emacs24
    c. sudo apt-get install git
    d. cd <directory>
        i. Move into the directory where you would like to work.
    e. git clone **https://github.com/geofft/qemu.git** -b 6.828-1.7.0
    f. cd qemu
    g. sudo apt-get install libsdl1.2-dev
    h. git submodule update --init pixman
    i. git submodule update --init dtc
    j. ./configure --disable-kvm
    k. sudo apt-get install autoconf
    l. sudo apt-get install libtool
    m. make
        i. **WARNING:** This step is going to take a while (depending on the user's hardware this may take approximately 1-2 hours).
    n. sudo make install
    o. cd ..
    p. git clone **https://github.com/mit-pdos/xv6-public.git**
        i. This link may change, if the clone does not succeed, check the most current repository from MIT's official xv6 website.
    q. cd xv6
    r. make

You are now ready to run QEMU to work in your operating System. To run xv6 on QEMU execute the

following commands in the terminal:

1. cd xv6
2. make qemu
    a. ALTERNATIVE: make qemu-nox (if you do not have a VGA window).

To run xv6 on QEMU with GDB:

Open two terminals

In terminal 1, execute the following commands:

1. cd xv6
2. make qemu-gdb (OR make qemu-nox-gdb)

In terminal 2, execute the following commands:

1. cd xv6
2. gdb –iex "set auto-load safe-path <path-to-xv6-folder>" kernel

Helpful Information:

If your mouse and/or keyboard becomes trapped in QEMU, Ctrl-Alt will exit the mouse grab.

When attempting to exit QEMU, simply close the VGA window (or terminal).

If you are debugging and you make a change to the code, make sure to close both QEMU and GDB. Otherwise they will get out of sync and cause problems.

To quit GDB: type q.

To terminate an action being executed in GDB: Ctrl-C

To find the exact path required for the gdb command you can run gdb kernel in the second terminal. The path will be listed in the error provided.

Inserting a User Program Within xv6

Overview

One of the first steps in navigating xv6 is learning to insert a user level C program into the system. This will eventually allow you to test and implement more advanced features. These features include additional system calls such as time. The following tutorial demonstrates how to add a basic user program and display it at the shell prompt.

Tutorial

1. Create a new C program (sample_program.c) in your chosen text editor. The contents of the program can be as simple as that shown in Figure 1.

```
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int main(void)
6  {
7      printf(1,"This is a sample xv6 user program\n");
8      exit();
9  }
```

Figure 1: Sample xv6 User Program (sample_program.c)

2. Save the program within the xv6 folder.
3. Open the xv6 Makefile from this same directory. The Makefile should be edited to ensure that sample_program.c is properly added to the existing set of user programs to be compiled. This is done as shown in Figures 2 and 3.

```
162  UPROGS=\
163      _cat\
164      _echo\
165      _forktest\
166      _grep\
167      _init\
168      _kill\
169      _ln\
170      _ls\
171      _mkdir\
172      _rm\
173      _sh\
174      _stressfs\
175      _usertests\
176      _wc\
177      _zombie\
178      _sample_program\
```

Figure 2: Add sample_program to the List of User Programs (Line 178)

```
245  EXTRA=\
246      mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
247      ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
248      printf.c umalloc.c _sample_program.c\
249      README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
250      .gdbinit.tmpl gdbutil\
```

Figure 3: Add sample_program.c to the List of User Programs (Line 248)

4. Open the terminal to compile the system and start up xv6 on QEMU. The commands should be the following:

   cd Documents/xv6 (*note – your xv6 directory might be different*)
   make clean
   make
   make qemu *or* make qemu-nox

   If there were no compilation errors xv6 should have been properly launched as shown in **Figure 4.**

```
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.0133889 s, 382 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 9.7148e-05 s, 5.3 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
354+1 records in
354+1 records out
181484 bytes (181 kB) copied, 0.000585624 s, 310 MB/s
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -dr
ive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$
```

Figure 4: Compilation and Start-Up of xv6

5. Use the **ls** command to verify that sample_program.c was added to the existing list of available user programs. You can then type **sample_program** to view the output of the newly created test program. **Figure 5** demonstrates this procedure.

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2517
cat            2 3 14420
echo           2 4 13285
forktest       2 5 8127
grep           2 6 15972
init           2 7 14166
kill           2 8 13325
ln             2 9 13251
ls             2 10 16103
mkdir          2 11 13346
rm             2 12 13323
sh             2 13 24783
stressfs       2 14 14253
usertests      2 15 67185
wc             2 16 15102
zombie         2 17 12995
sample_program 2 18 13063
console        3 19 0
$ sample_program
This is a sample xv6 user program
$
```

Figure 5: Successful Sample Program Output

Fork

Fork is a system call to create a new process within a process. When fork is called, a new process called the child process is created, which is a separate process with a copy of the memory contents of the calling (parent) process. Fork returns in both the parent and child processes. In the parent process, fork returns the child's process id and in the child process, fork returns zero. It returns a negative number if it fails.

```
pid = fork()
if pid > 0
     parent process execution
else if pid =0
     child process execution
else
     error
```

Now, you are required to modify sample_program in such a way that when it is run, following output is displayed.

```
$ sample_program
Inside Parent Process: child id = 8
Hello xv6 world
Child 8 is done executing
```

You need to execute a command "**echo hello xv6 world**" inside child process using exec() system call. A parent process should wait for the child process to execute the command and terminate before proceeding further.

For further details, refer to chapter 0 in xv6 reference book.

Part 2: 30 Points

Introduction:
Part 2 of the assignment is to create a new user program that can read and execute a given script file. A script file contains commands and/or logic for a Command Line Interface (CLI) for the Operating System to execute. You may already be familiar with the BASH (Bourne-Again Shell) which is a popular CLI for Linux and macOS systems. BASH has many features such as conditional logic, static and dynamic variables, pattern recognition, and the ability to run script files. It is possible to extend the xv6 console to include running script files, but for simplicity, we will just create a new and separate user program to do the work for us, and we will only implement basic functionality into our program.

Create a New User Program
Create a new C file called ***script.c*** and a user program called ***script*** by following the steps above for sample_program. Test that your user program was created successfully before moving on.

Input Specification
Your task is to create a user program that will read a script file with the correct filename extension and execute the commands inside. Your program must accept two arguments from the command line (by using argv and argc) in the following format:

script [FILENAME.sh] [Optional Flag: 0/1]

The flag argument is either a 0 or a 1. It is optional, so if the flag is omitted, then your program should default to using 0 as the flag. The flag lets the user choose whether or not the command is printed before it's output. If the flag is 1, then print the command before running it. If the flag is 0, or omitted, then do not print the command, just run the command.

The script file will contain commands for your program to execute. Each command will be on its own line. The script files will **not** contain conditional logic, variables, globbing patterns, or strings in "" or ''. For simplicity, you may assume that each command does not contain more than 10 arguments. An example of a script file, test.sh, is given below:

Example:

```
echo Hello World!
echo Running Test Script!
sample_program
echo Making directory: myDir
mkdir myDir
ls
rm myDir
echo Exiting Test Script!
```

test.sh

We can run this script with the any of the following commands:
script test.sh will run the commands without printing the commands themselves,
script test.sh 0 will run the commands without printing the commands themselves,
script test.sh 1 will print the commands and then run them.

The file name must have the extension "***.sh***". If it does not, then your program must print an error message and exit. If the file has the correct extension, then your program will try to open the file. If there is an error opening the file, then your program must print an error message and exit. If too few or too many arguments are passed to the script program, display a usage message and exit. You do not need to perform any error checking on the contents of the script file.

Adding New Files to xv6

In xv6, it is not enough to just create a file in xv6's directory to add it to the xv6 file system. We must also make changes in the makefile so that when we compile xv6, our new file is added to the system. Create a new file called test.sh in your xv6 folder. In test.sh, write the commands from the above example. Now open the xv6 Makefile. Add the name of the file, **test.sh**, under the label fs.img (about line 175, after the UPROGS list) and on the following line, just like the file README.

Now, run xv6 and check that your new file, test.sh, has been added by running the ls command. If you make changes to your test.sh file, you must run **make clean** before running **make qemu** to force xv6 to rebuild its file system if you want your changes to be visible in xv6.

Open, Read, and Close

The open and read system calls will be used to open and then read a given script file. They have very similar syntax to the open and read system calls in the POSIX library. After you check the file name, your program will open the file by using the open system call. The function prototype is given below:

int open(char* path, int mode);

where **path** is a string containing the filename (or the full path to the file including the filename), and **mode** is the read/write permissions with which to open the file. You should include the file **fcntl.h** in your script.c file as it contains the modes: O_RDONLY, O_WRONLY, O_RDWR, and O_CREATE. The open system call will return the positive integer **file descriptor**, **fd**, on success and a negative integer on failure.

The read system call is used to read characters from a file. The function prototype is given below:

int read(int fd, char* buf, int count);

where **fd** is a valid file descriptor, **buf**, is the starting address to a character buffer array of at least **count** elements, and **count** is the maximum number of characters to be read from the given file, **fd**. The read system call will read at most **count** characters from **fd** and place them in the character array given by **buf**. The read system call returns a positive integer of the number of characters read from the file on success (max is **count**), returns 0 if end of file is reached, and returns a negative integer on failure.

Once you have read all of the input, close the file by using the close system call:

int close(int fd);

Script User Program

To use exec, you will need to pass an argument vector, an array of strings, that contain the command to be run. The script user program will read a script file, and execute each line by populating an argument vector, and then calling fork() and exec(). For example, to run the command: "echo Hello World!" you must create an argument vector that has three elements:

char** myArgv[3] = {"echo", "Hello", "World!"};

Then, pass myArgv to the exec system call. You will need to create the algorithm that will read the script file, put each word (separated by spaces) into it's own string, and then when a new line character is read, it will put each string into a myArgv array, then fork, and then exec passing myArgv to exec.