EECS 343 Operating Systems -- Project 1 -- ps on xv6

Project Description | Submission Instructions

PROJECT 1: PS -- PROCESS STATUS -- ON XV6

IMPORTANT DATES

Out: Wednesday, September 21st, 2016.

Due: Friday, September 30th, 2015 (11:59 PM CDT).

Submission: Submission instructions are now posted. Click on the tab above.

RESOURCES

- Some useful GDB and QEMU commands: **https://pdos.csail.mit.edu/6.828/2016/labguide.html**
- xv6 textbook: **https://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf**
  (note that we are using a version of the code from 2012)
- Guide to xv6 codebase: **https://pdos.csail.mit.edu/6.828/2012/xv6/xv6-rev7.pdf**
- Hexadecimal converter: **https://www.mathsisfun.com/binary-decimal-hexadecimal-converter.html**

PROJECT OVERVIEW

This project is composed of three parts, which you must complete in order. Part 0 is the easiest part and simply requires you to get your development environment setup correctly. Part 1, which is slightly more difficult, will require you to trace a `fork` system call ("syscall") using GDB and answer some questions. Part 2, which is the most challenging, will have you implement the `ps` shell command along with a syscall that is required for the command to work. Detailed instructions will be given further below, but here is a brief summary of each part.

Part 0 (15 points)
Must be done individually and should be completed ASAP. This part will require you to carefully follow instructions to get your development environment setup. Since this is an operating systems course, the projects will require you to write operating system code, which requires a fairly complex development environment. So even though this is the easiest part of the project, it isn't *that* easy.

Part 1 (15 points)
Must be done individually as well. This part also involves no coding, but it does require you to have your xv6 kernel up and running. That is, you must complete Part 0 in order to complete Part 1. In this part, you will use GDB to walk through the execution of the `fork` system call and answer a few questions as you do. When you are done, you will submit a text file with your answers.

Part 2 (70 points)
Should be done in groups of 2-3 people. In this part, you will write some kernel code as well as a user program. The user program you must write is the `ps` shell command. In order to write this user program, you will also have to write a new syscall for the kernel called `getprocs`. This will give you a first taste of operating system development.

EDUCATIONAL OBJECTIVES

This project aims to achieve several educational objectives:

- Every individual gets their development environment set up with a functioning xv6 kernel.
- Gain a thorough understanding of how new processes are created via the fork system call.
- Become a master at using GDB to inspect and debug code.
- Understand how information about each process is stored in the process table.
- Understand how a user program can invoke kernel code via system calls.
- Cut your teeth as a budding OS developer.

PART 0: SET UP DEVELOPMENT ENVIRONMENT

15 points

To be completed individually

To get your development environment setup, carefully read and follow the instructions here:
http://www.aqualab.cs.northwestern.edu/class/333-eecs343-xv6

PART 1: HOW PROCESSES ARE BORN

15 points

To be completed individually

In this part, you will use GDB to walk through the xv6 code as it creates a new process. We will provide the GDB commands for you. All you have to do is follow the instructions and answer the questions that we ask along the way.

To get started, open two terminals and make sure you are in the xv6 directory in both. In one terminal, run xv6 with remote debugging enabled by typing:

```
$ make qemu-nox-gdb
```

In the other terminal, run GDB as a remote debugger by typing:

```
$ gdb kernel/kernel

+++++++++++++++++++++++++++++++++
```

This terminal (your GDB terminal) should now be connected to your xv6 terminal. Meanwhile, in the other terminal, xv6 is currently stalling the boot-up process while it waits for you to give commands via GDB. Tell GDB to allow execution to continue by typing:

```
(gdb) c
```

Wait for xv6 to boot up until it reaches the shell prompt. Then in your GDB terminal, interrupt xv6 by hitting:

```
control+c
```

You should see GDB give the following output:

```
Program received signal SIGINT, Interrupt.
The target architecture is assumed to be i386
```

NOTE: Rumor has it that if you are using putty to ssh from a Windows machine into a lab machine, the `control+c` command may not work to interrupt xv6. You can work around this problem by skipping from the "+++++++" line above to the "******" line below.

GDB will also print out an address and a line number of the exact spot in code that is currently being executed. This will vary because the kernel's scheduler is looping through the process table looking for processes to execute. You don't have to worry about that for now. The only process we currently care about is the shell process that is sitting there, waiting for input.

****

We want to set a breakpoint in the shell program, but GDB currently has the symbol table for the kernel loaded, so we have to tell it to load the symbol file for the shell program by typing:

```
(gdb) symbol-file user/sh.o
```

When it asks if you want to load a new symbol table, type 'y'. We can now set a breakpoint in the shell program by typing:

```
(gdb) b sh.c:160
```

This command sets a breakpoint (b) in the file `user/sh.c` at line 160. Go ahead and open a text editor such as gedit or Sublime, open the file `user/sh.c` and look at line 160. Take a quick look at the while loop that this line sits within. Now return control to xv6 by typing 'c' in your GDB terminal.

Switch over to your xv6 terminal. We want to run a program that will trigger the breakpoint we just set. Let's run the echo program. Type:

```
$ echo hello world
```

You should see that your breakpoint has been hit. In GDB you will see this output:

```
Breakpoint 1, main () at user/sh.c:160
```

GDB will also print the code on line 160. In GDB, type 'n' which should bring you to line 168. Make sure you understand why you didn't enter the `if` block on lines 161-167. On line 168, we are about to execute a function called `fork1()`. We want to watch that function execute so set a breakpoint by typing:

```
(gdb) b fork1
```

Then type 'c' to continue. You are now inside the `fork1()` function. Pretty much the first thing this function does is call a function called `fork()`. Don't be fooled by how similar the name is. `fork()` is a system call and will invoke the execution of kernel code.

Try to set a breakpoint in the `fork` syscall by typing:

```
(gdb) b fork
```

GDB will give the following warning:

```
Function "fork" not defined.
Make breakpoint pending on future shared library load? (y or [n])
```

The reason for this warning is that we currently have the symbol table for sh loaded, but the code for the `fork` syscall is in kernel code. Type 'y' then load the symbol table for the kernel by typing:

```
(gdb) symbol-file kernel/kernel
```

Type 'y' again to confirm that you want to load the new symbol table. You should see the following warnings in GDB:

```
Error in re-setting breakpoint 1: No source file named sh.c.
Error in re-setting breakpoint 2: Function "fork1" not defined.
```

This is telling us that our breakpoints that we previously set are no longer valid in the new symbol table. We have to delete them to avoid any trouble:

```
(gdb) d 1
(gdb) d 2
```

These commands tell GDB to delete (d) breakpoints #1 and #2.

Now we should also confirm that our `fork` breakpoint was automatically set after the kernel symbol table loaded. To get info about your current breakpoints, type:

```
(gdb) info b
```

You should should see a list of your breakpoints (just one breakpoint in the list for now). Make sure that it looks like this:

```
3 breakpoint keep y 0x00103e3c in fork at kernel/proc.c:134
```

Type 'c' in GDB and you should hit your `fork` breakpoint. Go back to your text editor, open the file kernel/proc.c, and look at line 134. Indeed, you are inside the `fork` function. To see how we got here, let's inspect the call stack. In GDB, type:

```
(gdb) bt
```

On the top of the stack in spot #0 is the `fork()` function we are currently sitting in. You may have expected that this function was called directly by the `fork1()` function we were looking at above, but instead, in spot #1, we see that this `fork()` function was actually called by `sys_fork()`, which was called by `syscall()`, which was called by `trap()`, which was called by `alltraps()`. These functions are evidence of the work done by the kernel "behind the curtain" whenever a user program makes a syscall.

Now let's look more closely at the `fork()` function we are still sitting in. Read the comments above the function to get an idea of what it does. Then read through the function code itself and try to understand how it works.

Question 1: What is the *type* of variable `np`?

Question 2: What does `np` represent? In other words, what do the letters 'n' and 'p' stand for?

Question 3: What does the variable `proc` represent? Hint: You can find its type by looking at its declaration in `kernel/proc.h`.

If you can't answer these questions definitively right now, that's OK, we're going to step through the code. In GDB, type 'n' to execute the line 134. Then type:

```
(gdb) print np
```

GDB outputs the type of the variable and the value. The type is your answer to question 1 above. The value is actually an address. Since `np` is a pointer, it stores the address of an object that it is pointing to. This leads to our next question...

Question 4: What is the address of the object that `np` is pointing to?

Now have a look at the object itself by typing:

```
(gdb) print *np
```

We'll actually want to keep an eye on this object because its attributes will gradually be initialized throughout the `fork` syscall. We can tell GDB to display this object automatically by typing:

```
(gdb) display *np
```

Then type 'n' to execute the next line of code and notice that GDB automatically displays the `np` object again. Look at the line of code that just executed (line 138), then look at which attributes of the `np` object have changed as a result.

Continue stepping through the `fork` function watching as the values within the np object continue to change. Make sure you stop before the function returns on line 159.

Question 5: What is the `name` of the child process that was created? Note: Although you may not believe your own answer at first, you can trust that GDB is not lying to you. The child process will later be renamed and repurposed.

Now take a look at the value of `pid`, which the `fork` syscall is about to return, by typing:

```
(gdb) print pid
```

Question 6: What is the value of `pid`?

Question 7: In which process, the parent or the child, will this value be returned?

PART 2: IMPLEMENTING THE PS COMMAND

70 points

To be completed in groups of 2-3

Part 2 requires you to implement the `ps` shell command. When you are done with this project, you will be able to get info about the currently running processes on xv6 by typing:

```
$ ps
```

To get an idea of what the `ps` command is supposed to do, open a regular Linux shell (e.g. on a lab machine) and type `ps`. You will see some info about the currently running processes. To see even more info, run `ps` again with the `-u` flag. You can read more about the `ps` command **on wikipedia** or by doing a google search.

Part 2a: Writing the getprocs syscall

Before you can implement the `ps` command, you will need to write a new syscall to get information about the current processes from the kernel. Here is the function signature for the syscall you will write:

```
int getprocs(struct ProcessInfo processInfoTable[]);
```

You'll have to dig through the codebase to find where exactly this function signature should be added.

This syscall, once implemented, will return the number of current processes in the kernel--that is, the number of entries in the kernel's process table that are in any state other than `UNUSED`. If the syscall cannot be completed for any reason, it shall return -1.

In Part 2b down below, we'll worry about populating the `ProcessInfo` table, but for now, we just need to add a struct definition so that your code will compile. Add a new file `include/ProcessInfo.h` with this definition of struct ProcessInfo:

```
struct ProcessInfo {
  int pid; // process id
  int ppid; // parent pid
  int state; // state
  uint sz; // size in bytes
  char name[16]; // name of process
};
```

Now you are ready to begin implementing your new syscall. Again, you'll have to search the codebase to see how and where other syscalls are implemented, then follow suit.

In order to help you search the code, you'll probably want to use a text editor that allows you to search multiple files at once so that you can find every occurence of your search term. This will help you figure out all the files that you have to touch in order to implement a new syscall.

Once you have your syscall implemented, see if your code compiles by simply typing `make` in the xv6 directory. Once your code is compiling, you can try to run xv6 by typing `make qemu-nox`.

Finally you can do a sanity check to see if your syscall is working by writing a simple user program that uses it. Create a new file `user/ps.c` that invokes your new syscall and outputs the result using `printf`. In order to run this new user program within xv6, you'll have to make a small modification to `user/makefile.mk`.

Part 2b: Writing the ps user program

In the final part of this project, you'll expand your `user/ps.c` program to actually print out information about each process similar to the Linux `ps` command. Your command won't print out the header line, but for each process it will print out a line containing the following info:

process id, parent process id, state, size, name

Here are some additional requirements:

- Each process should be on a new line.
- Each piece of info on a line should be separated by two spaces.
- The parent process id of the first process is meaningless. For this value, print -1.
- The state must be printed as human-readable text in all caps (not an integer value).

Here is an example of what your output will approximately look like when you run your `ps` command from the xv6 shell prompt:

```
$ ps
1  -1  SLEEPING  8192  init
2  1  SLEEPING  12288  sh
3  2  RUNNING  8192  ps
```

In order to accomplish this, you'll have to expand your `getprocs` syscall to actually populate the `processInfoTable[]` with information that it gathers from the kernel's `ptable` (process table). You'll be working in pretty much all the same files as in Part 2a, just adding functionality.

One tricky part is figuring out how a syscall gets the arguments that were passed to it (e.g. how your kernel code will get a hold of the pointer to the `processInfoTable`). Fortunately, there are functions that are provided to help with this. You should be able to find them by looking at some of the other syscalls.

Once you are done with your modifications to your `getprocs` syscall and your `ps.c` program, you can test everything by running xv6 and typing `ps` at the prompt. For more interesting outputs you can run a shell within a shell (within a shell...) by typing `sh` at the xv6 prompt. And you can kill processes by typing `kill` followed by the pid.

Attachments: