CS 333: Operating Systems Lab
Autumn 2018

Lab 4: xv6 inside-out

Goal

In this lab you will learn how to use the xv6 operating system, implement system calls and explore examples of OS state.

1. Setting up xv6

- Download the source code for xv6 from the following url:
  `http://www.cse.iitb.ac.in/~puru/courses/autumn18/labs/xv6-public.tar.gz`
  Make sure you read the README to understand how to boot into a xv6-based system.

  make clean, `make` and `make qemu` are of interest.
  What is qemu?

- Download, read and use as reference the xv6 source code companion book.
  url: `http://www.cse.iitb.ac.in/~puru/courses/autumn18/labs/xv6-rev10.pdf`

2. User programs in xv6

- After executing `make qemu` you will see a prompt. The prompt is the xv6 command line interface to execute user level programs. Begin with `ls` and find which programs exist and try executing them.

  The source code for all programs is included as part of the xv6 distribution. Look up the implementation of these programs. For example, `cat.c` has the source code for the `cat` program. Execute and lookup the following: `ls`, `cat`, `wc`, `echo`, `grep` etc. Understand how the syntax in some places is different than normal C syntax.

  Check the `makefile` to see how the program `wc` is set up for compilation.

- Modify the existing shell program `sh.c` in xv6 to change the shell prompt.
  E.g. : turtle$

- Copy the existing `cat.c` program to a file `head.c` and modify it to print the first <n> lines of a file to the terminal.
  Sample run:

```
$ head 5 abc.txt hello.txt
-- abc.txt --
abc
def
ghi
jkl
mno
-- hello.txt --
Hello
World
```

- Write a program `cmd.c` that creates a child process, child process executes a program, and parent waits till completion of the child process before terminating. This program should use the `fork` and exec system call of xv6. The program to be execute by the parent process can be and of the simple xv6 programs and should be specified at the command line. Example usage:

```
./cmd ls
./cmd echo hello
```

3. hello xv6!

Ready to dive in?
There is not turning back from here. The exiting and adventurous journey inside xv6 and many more operating systems to follow starts from here. Buckle-up, see you on the other side!

(a) **the `hello` system call**
Implement a xv6 system call, called `hello()`, which writes
`Hi! Welcome to the world of xv6!`
to the console. You can use `cprintf` inside the system call for this.
Also, write a simple C program `helloxv6.c` which calls the newly implemented `hello()` system call, and verify the output.

Hints: You will need to modify a number of different files for this exercise, though the total number of lines of code you will be adding is quite small.
At a minimum, you will need to alter `syscall.h, syscall.c, user.h`, and `usys.S` to implement your new system call.

(b) **got siblings?**
Implement a system call `get_sibling_info()` which prints the details of siblings of the calling process to the console.
`get_sibling_info()` should print `pid` and process state of it's siblings in the following format.
<pid> <procstate>
<pid> <procstate>
...

You are given a sample `my_siblings.c` program that takes an integer n, followed by a combination of 0, 1 and 2 of length n, as command line arguments. This program creates $n+1$ child processes, the first n child processes perform some task based based on the input argument (0/1/2 specified for each of the n child processes.) The last process executes the your system call `get_sibling_info()` and displays the output.
Add this as a user level program to test your implementation.

Sample run:
```
$ my_siblings 6 1 2 1 0 2 0
4 RUNNABLE
5 ZOMBIE
6 RUNNABLE
7 SLEEPING
8 ZOMBIE
9 SLEEPING
```

Hints: You need to find process ID of calling process, and process ID of its parent and traverse all the PCBs and compare their parent PID with parent of the calling process.
Implement your system call in `proc.c` file. To implement this system call, you will need to understand `struct ptable, struct proc`.
The calling process details should not be printed. Only sibling details should be printed.

(c) **parameterizing system calls (no submission required, but will be required for lab quiz.)**
Implement a system call `get_ancestors()`, which takes a positive number `n` and a pointer to a character buffer as arguments.
`get_ancestors(n, buf)`

This system call writes the process IDs of n parents of the calling process to the given character buffer in a space separated manner as given below. Parent of calling process is at level 1. You can assume that the size of buffer is sufficient enough to hold the required details. If the number of ancestors is less than n, then the system call should collect PIDs till `init` process and return 0, otherwise, the system call should return 1.

To pass parameters to system call, understand how it is done for other system calls. You will have to use `argint` and `argptr` in `syscall.c`—check the xv6 system calls that need arguments and how they are handled.

You are also given a sample user-level program `my_ancestors.c` which takes two numbers as command line arguments and uses the system call. The first argument is depth of forking and the second is the parameter passed to the system call. Add this as a user level program to xv6 and test your implementation.

Sample runs:

```
$ my_ancestors 6 10
Process:  97
Ancestors:  95 89 83 77 71 65 2 1
Return value:  0
```

the `my_ancestors` program creates 6 child process in a recursive manner and the `get_ancestors(n, buf)` system call is invoked from the 6th child process. Here, pid of the parent process is 65 and parent pid of 6th child process is 95.

```
$ my_ancestors 5 3
Process:  59
Ancestors:  58 52 46
Return value:  1
```

.

Submission Guidelines

- All submissions via moodle. Name your submission as: **<rollno_lab4>.tar.gz**
- The tar should contain the following files in the following directory structure:

  <roll_number_lab4>/
  |____sh.c
  |____head.c
  |____cmd.c
  |____helloxv6.c
  |____<all modified files in xv6>
  |____Makefile
  |____outputs of sample runs (exercise 2, & 3)

- We will evaluate your submission by reading through your code, executing it with different test cases. Make sure all files updated and required for compilation are part of the submission.
- **Deadline: 13th August 2018, 5 p.m.**