Systems Bootcamp

OS Internals Track
Week-2

May 9, 2022

Background material for the lab

- You must cover these - Lectures 1, 2, 22, 3, 23, 4 from Lectures on Operating Systems, in the order specified here.
- (Optional) Chapters 0, 1 from this book
- Link to the PDF formatted xv6 source code if you want the whole xv6 kernel code in one place. This doesn't include the user programs like `ls, echo` etc.

Before you begin the assignment

1. Download, install, and run the original xv6 OS code. You can use your regular desktop/laptop to run xv6; it runs on an x86 emulator called QEMU that emulates x86 hardware on your local machine. In the xv6 folder, run `make`, followed by `make qemu` or `make-qemu-nox`, to boot xv6 and open a shell. Detailed instructions are available here.
2. We have modified some xv6 files for this lab, and these patched files are provided as part of this lab's code. Before you begin the lab, copy the patched files into the main xv6 code directory.
3. For this lab, you will need to understand and modify following files: `proc.c`, `proc.h`, `syscall.c`, `syscall.h`, `sysproc.c`, `defs.h`, `user.h`, and `usys.S`. Below are some details on these files. You may want to revisit this for Part B.
   - `user.h` contains the system call definitions in xv6.
   - `usys.S` contains a list of system calls exported by the kernel, and the corresponding invocation of the trap instruction.
   - `syscall.h` contains a mapping from system call name to system call number. Every system call must have a number assigned here.
   - `syscall.c` contains helper functions to parse system call arguments, and pointers to the actual system call implementations.
   - `sysproc.c` contains the implementations of process related system calls. All the new system calls that you add will be implemented here.
   - `defs.h` is a header file with function definitions in the kernel.
   - `proc.h` contains the struct proc structure.
   - `proc.c` contains implementations of various process related system calls, and the scheduler function. This file also contains the definition of ptable, so any code that must traverse the process list in xv6 must be written here.

Part A: User Programs in xv6

1. **Debugging user programs on xv6 with gdb**

   Look at 'Remote Debugging xv6 under QEMU' section of this page for details on how to start debugging xv6 using gdb.
   Execute the user program `debug` and observe that the process traps because of some program fault. You have to find the bug and make this program execute without any traps.

   (a) In xv6 folder, run `make` followed by `make qemu-nox-gdb` to run the QEMU emulator in debug-mode.
   (b) In other terminal run `gdb kernel`
   (c) From the gdb interface run `(gdb) target remote localhost:26000` where *26000* is the TCP port that step 1 reported at the end (this might change).
   (d) Load the user executable with `(gdb) file user_program`. Note that *user_program* is name of executable as in host machine. In xv6 it will be `_debug`.
   (e) Place a breakpoint `(gdb) break main` and continue with `(gdb) continue`
   (f) Now you can use the gdb commands as learnt in Week 1 assignment to debug further.

2. **Adding user programs in xv6**

   A simple example is given here.

   (a) 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. (You can search for `wc` in the file to see what additions are made.)

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

   Hint : Just search where the existing shell prompt `$` is and modify it.

   (c) Complete the program `cmd.c` so that it 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 executed by the child process can be one of the simple xv6 programs and will be specified at the command line.
   Example usage:
   `./cmd ls`
   `./cmd echo hello`

   We have already defined the arguments to be passed to `exec` for you in `cmd.c`. You have to use them and implement the core logic in the provided block

```
...
\\ Implement your code here

\\
...
```

   (d) Modify the `makefile` and test your implementations.

Part B: New system calls in xv6

We will add a couple of new system calls to the xv6 kernel.
Ready? Start by understanding the system call path followed when a system call is invoked. A simple tutorial is linked here. You don't need to understand everything, just get an overview of the flow through kernel code on invocation of a system call and which files need to be modified to add a new system call.

Now look at an existing system call to understand how system calls are to be added. For example, look at how `sysproc.c` implements system calls like `sys_fork` (which calls functions in other files).

You will implement the following new system calls in xv6.

1. **Hello xv6 World!**

   (a) Implement a system call, called `hello()`, which prints `Hi! Welcome to the world of xv6!` to the console. You can use `cprintf` for printing in kernel mode.

   (b) Next, we will implement a system call called `helloYou(name)` that takes an argument `char* name`. It prints *name* to the console. You can use `cprintf` for printing in kernel mode.

   Check out the `open` system call in `sysfile.c` that takes in multiple arguments like strings and integers, and return a simple integer value.
   Also check `argint, argptr, argstr` functions and their descriptions in `syscall.c`.

   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 `sysproc.c`, `syscall.h`, `syscall.c` `user.h`, and `usys.S` to implement your new system calls.

   You can test your implementation with provided testcases: `hello.c` and `helloyou.c`

2. **Exploring OS State**

   We will be using the `struct ptable` in the remaining exercises. Whenever you access the `ptable` you must hold a *lock* - a synchronization primitive, before you start using it. Don't worry if you don't understand locks, we will learn about them in the upcoming weeks.
   For now just make sure that if you need to use the `ptable` you follow the pattern :

   ```
   ...
   acquire(&ptable.lock);
   // Make use of the ptable structure
   release(&ptable.lock); // DO NOT FORGET THIS!!
   ...
   ```

   (a) **got siblings?**

   We will implement a system call `get_siblings_info()` which prints the details of siblings (processes forked from the same parent process) of the calling process to the console.

   `get_siblings_info()` should print process ID and process state of it's sibings in the following format.
   `<pid> <procstate>`
   `<pid> <procstate>`
   ...

Main Idea:

i. Find process ID of the calling process.
ii. Find process ID of the parent process of the calling process.
iii. Traverse the list of PCBs and compare their parent PID with parent of calling process.

Since this requires you to access the `ptable`, you will have to implement the main logic of your system call in `proc.c` file and then invoke this function from `sysproc.h`.

To implement this system call, you will need to understand `struct ptable` from `proc.c` and `struct proc, enum procstate` from `proc.h`.

Note: Details of calling process shouldn't be printed. Only sibling details should be printed.

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 on the input argument (0/1/2 specified for each of the *n* child processes). The last process executes your system call `get_siblings_info()` and displays the output.

Add `my_siblings.c` 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
```

(b) **parameterizing system calls (Optional - for those who want more challenge!)**

Implement a system call `get_ancestors()`, which takes a positive number `n` and a pointer to an integer array as arguments.
`get_ancestors(n, array)`

This system call writes the process IDs of *n* parents of the calling process to the given array. Parent of calling process is at level 1. You can assume that the size of array 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` that we learnt in 1(b) of Part B.

Note: Make sure you handle the case when `n` is less than number of ancestors and you reach the `init` process.

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 `n` 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

1. Name your submission as: `<id>_week2.tar.gz`
2. The tar should contain the following files in the following directory structure:

```
<id>_week2>/
||____debug.c
||____sh.c
||____head.c
||____cmd.c
||____<all modified files in xv6>
||____Makefile
```

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.
4. **Deadline:**