CSU0033 Operating Systems, Homework 2

Department of Computer Science and Information Engineering
National Taiwan Normal University

Mar. 13, 2025

The deadline for Moodle upload of your solution is 9PM, March 24, 2025. Again, you are encouraged to ask and help answer questions on Moodle. We are CSU0033 community :)

1 OS booting sequence (15 points)

When an OS boots, the machine will start in the kernel space first (machine mode and supervisor mode) and then switches into the user space (user mode). Consider such a switch in xv6 and give one concrete, necessary TO-DO item required for a correct switch, in addition to the following two items: (1) update the privilege mode, and (2) update the program counter address[1]. Explain the purpose of the TO-DO item you picked. The video we've uploaded to Moodle can help you get started.

2 Analysis of lottery scheduling (15 points)

In the textbook introduction of the lottery scheduling, a simple implementation works by attaching jobs to a list data structure (Section 9.3). Use your own words to explain

1. Why is it that the ordering in the list does not affect the correctness of the algorithm?
2. In which way(s) is such an ordering-independent property (as stated in sub-question 1) helpful? A way to reason this is to think about what would be needed if the implementation is ordering-dependent.

[1]In computer engineering, the term *concrete* means something specific; for example, in RISC-V, `addi` is a concrete example of arithmetic instructions, and 32 is a concrete example of the number of bits in a register.

3 The `fork()` system call (20 points)

What would be the total number of processes created if a process calls the following C function[2]?

```
forkFun()
{
  int rc = fork();
  if (rc != 0) {
    fork();
    fork();
  }
}
```

You should study Chapter 5 of the textbook first, and think about the answer, and only after that, write a program to validate your answer. For homework submission, in this part you only need to provide the number. But for your own benefits you should go through the mentioned process (no pun intended).

4 Zombie and its killer (20 points)

In the Linux man page of `waitpid()`, the NOTES section describes a ghost story of *zombie* process. Read the story therein before you move on. Now, your job here is to write a short C program that will create 2000 zombie processes in Linux. Show the existence of two thousand zombie processes during your program execution, and then kill them all by killing the parent process. For this part of the assignments, submit two things: (1) your source code for such a zombie creation; (2) a screenshot showing both the existence of that many zombies and the proof that our dear Linux is a qualified zombie killer. Think carefully about the logic of your program, before you run it; otherwise, you could accidentally trigger a *fork bomb*[3].

Some further story: In Linux, `wait()` will return as soon as one of the child processes terminated. Therefore, if we only make a single call of `wait()` in a multi-child program, the program may terminate just fine, but what really happened under the table is that the rest of the children were inherited by the `init` process, which then patiently waited for their termination and re-claimed the resources afterward.

[2]By this statement it implies that the calling process is not part of the count.
[3]See https://en.wikipedia.org/wiki/Fork_bomb.

5 Implementation and code tracing (30 points)

In xv6, to create a file, we may use `echo` and then redirect the data to a file. Then we can use `cat` to show the content of a file and use `rm` to remove the file if desired. The following trace gives you some examples:

```
$ echo Hello World! > hello.txt
$ cat ./hello.txt
Hello World!
$ rm ./hello.txt
$
```

Like what we can do in Linux, we can create a *background process* by appending the command with `&`. In this way, we may run another program before the previous one finishes. That is also how we demonstrated CPU virtualization (aka time sharing) in class. If we do not append a command with `&`, the shell will block until the running process finished. To get yourself familiar with the xv6 environment, try to first play with this shell using the default commands (i.e., those you see from `ls`). Don't be surprised here that some commands in Linux do not work or works differently—they are just not implemented (yet)!

5.1 Writing your own user-space sleep program (15 points)

Write a user-space program *sleep.c* that will pause for some user-defined number of clock ticks. We will use this in future homework. Your program should call the `sleep()` system call, and when it is done, call the `exit()` system call. The `sleep()` system call only takes one argument, which is the mentioned user-defined number of clock ticks. Here is an example execution:

```
$ sleep 20
$
```

In the above example, the program will sleep for 20 ticks and then exit. Before you jump into coding, it is a good idea to take a look at some existing user-space program implementation in xv6.

5.2 Code tracing (15 points)

Now let's study the implementation of xv6's shell `sh`. The shell is implemented in *./xv6-riscv/user/sh.c*. Do some code tracing and use the code to explain the implementation of a background process. To be specific, explain how the shell enables the user to run the next command before the previous one finishes. The `wait(0)` on line 170 in *sh.c* seems to prevent that from working, no?