CSE 312 Operating Systems Spring 2020, Homework 2 Report

Ahmed Semih Özmekik
171044039

April 5, 2020

Abstract

Demonstration of small units and system calls and then the whole system will be shown. A better and extensive tests has been applied on the demo video. This is a quick demonstration report.

Demonstration Link: https://youtu.be/0w6sqyw18oA

1 Introduction

We will show that the requests stated in the homework are provided one by one. A demo video containing the unit test examples described here has been prepared. If you wish, you can watch the video from the given link above. It is almost impossible to demo micro kernels only with screenshots, so the demonstration link for 3 micro kernels that make up the main part of the assignment is included in the shared video. The first part of the demo video includes the unit tests mentioned here, and the second part shows the demonstration of micro kernels. The second part starts from minute 17.30.

2 System Calls

Implementing these POSIX system calls: fork, waitpid, execve, any other POSIX call that you need.

In addition to the three specified system calls, I also added an exit system call so that the processes exit properly and indicate this to the operating system of mine.

2.1 fork

I added a print into the fork system call code, so we will see the processes that has been created after the fork is made.

```
.text
main:   li $v0, 18      # fork
        syscall


exit1:  li $v0, 10
        syscall
```

Here is the output. As can be seen, a new process has been created. And the R[V0] value appropriate as well.

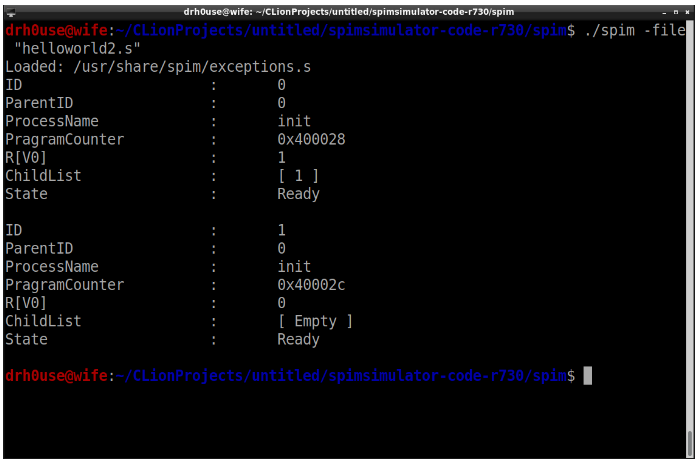

Figure 1: fork output

2.2 childpid

In this test, a new process is started first and child process performs a different print operation. Then the parent process is waiting for it to finish and performs printing.

```
        .data
msg:    .asciiz "Hello World, from parent!\n"
msg2:   .asciiz "Hello World, from child!\n"
msg1:   .asciiz "Waiting for child...\n"
    .extern foobar 4

.text

main:   li $v0, 18       # fork
        syscall
        beqz $v0, child # if fork() == 0, process is child.
        j wait

child:  li $v0, 4        # print child.
        la $a0, msg2
        syscall
        li $v0, 21
        syscall

wait:   li $v0, 4        # print child.
        la $a0, msg1
        syscall
        li $v0, 19       # wait for any child.
        li $a0, 0
        syscall
        beqz $v0, wait  # return == 0, keep waiting.
```

```

parent: li $v0, 4        # print parent
        la $a0, msg
        syscall

exit:   li $v0, 10
        syscall
        syscall
```

The expected output here.

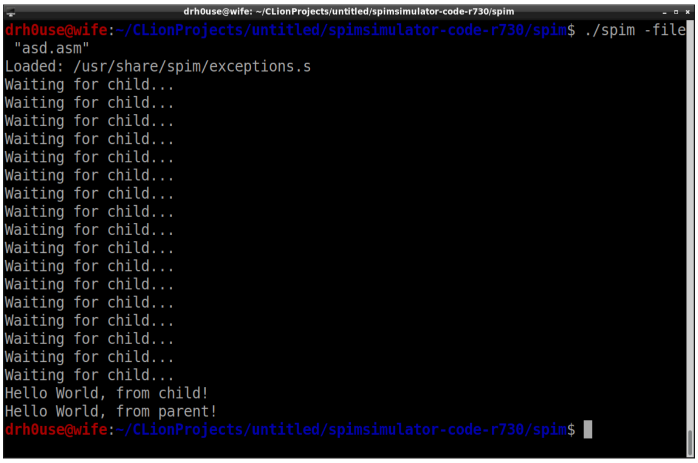

Figure 2: childpid output

2.3 execve

A program that gives a simple output to the screen and our sample test running that program is shown below.

```
         .data
msg:    .asciiz "Hello World\n"
     .extern foobar 4

        .text
        .globl main
main:   li $v0, 4        # syscall 4 (print_str)
        la $a0, msg      # argument: string
        syscall          # print the string

        li $v0, 21
        syscall
```

```
        .data
msg:    .asciiz "helloworld.s"

.text

main:   li $v0, 20      # execve current process.
        la $a0, msg
        syscall

exit:  li $v0, 10
        syscall
```

Switches to the upper program as expected.

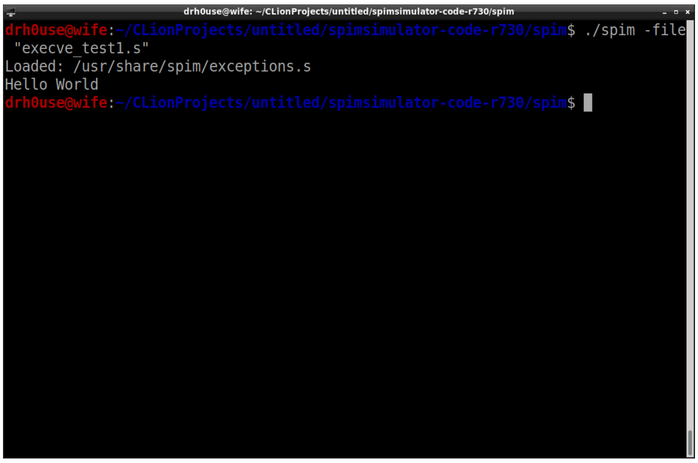

Figure 3: childpid output

3 Mixed Tests

- All 3 of the system calls together tested.

```
        .data
msg:    .asciiz "Hello World\n"
    .extern foobar 4

        .text
        .globl main
main:   li $v0, 4       # syscall 4 (print_str)
        la $a0, msg     # argument: string
        syscall         # print the string

        li $v0, 21
        syscall
```

```
        .data
msg:   .asciiz "helloworld.s"
msg3:  .asciiz "Child has returned!\n"
msg2:  .asciiz "Waiting for child...\n"

.text

main:   li $v0, 18        # fork
        syscall
        beqz $v0, child
        j exit

child:
        li $v0, 20
        la $a1, msg       # execve. helloworld.s
        syscall


exit:   li $v0, 4         # syscall 4 (print_str)
        la $a0, msg2
        syscall

        li $v0, 19        # wait for any child.
        li $a0, 0
        syscall
        beqz $v0, exit  # return == 0, keep waiting.

        li $v0, 4         # syscall 4 (print_str)
        la $a0, msg3
        syscall

exit1:   li $v0, 10
        syscall
```

The expected output came true.

- This test is a very good test. First the process calls the fork, and then both the child process and the parent (itself) call the fork once again. Let's consider how many processes should occur in total. First of all he has himself, so we have 1 process. There was also a child process after fork and that makes 2 processes in total. Because they will call the second fork system call, both child and himself, 2 more processes have been added. One of these two processes became his new child process and the other

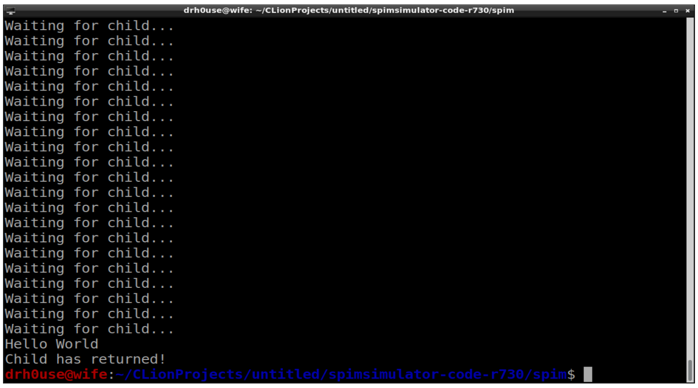

Figure 4: output

became the child of his child. Thus, there were 4 in total. A print has been added to the fork call, so you can examine who's father or child was at the time of creation.

```
.text
main:    li $v0, 18        # fork
         syscall

         li $v0, 18        # fork
         syscall

exit1:   li $v0, 10
         syscall
```

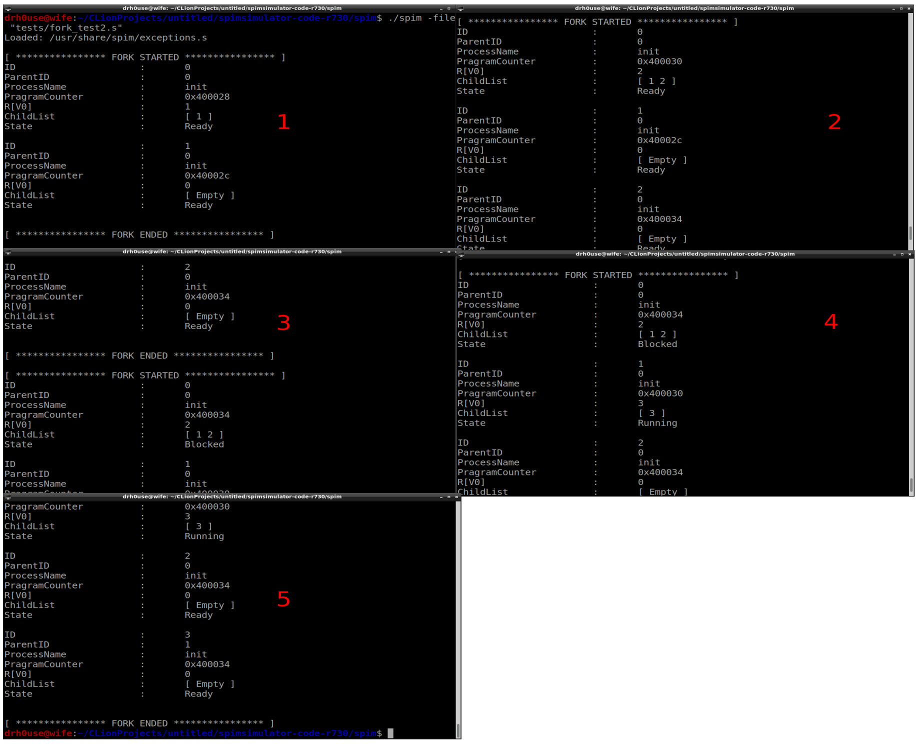

Figure 5: output

4 Micro Kernels (Whole System)

All small unit tests are running. Let's test micro kernels.

- During this test, a parent process creates 3 different forks, 3 different processes from the same ("Hello, World!" printing) program. Then, it waits for 3 child processes to finish and exit at the end. With the success of this test, we are almost confident that it will now work correctly on all micro kernels we will create. Because what they will do is no different from this test. They will either create a process 3 or 10 times and wait for the children to finish. The order of outputs is always different in the results of the test. This means that the scheduler, which makes context switches, is invoked at different non-deterministic intervals and that our system (system calls) works just like today's operating systems.

```
.data
program1:    .asciiz "helloworld.s"
program2:    .asciiz "helloworld.s"
program3:    .asciiz "helloworld.s"
msg4:        .asciiz "waiting for childs..\n"

.text

main:   li $v0, 18
        syscall
```

```
        bnez $v0, cont1 # parent continues to forking.
        la $a0, program1
        li $v0, 20      # fork and execve #1
        syscall
        j exit

cont1:  li $v0, 18
        syscall
        bnez $v0, cont2 # parent continues to forking.
        la $a0, program2
        li $v0, 20     # fork and execve #2
        syscall

cont2:  li $v0, 18
        syscall
        bnez $v0, loop # parent exits
        la $a0, program3
        li $v0, 20      # fork and execve #3
        syscall
        j exit

    # wait for all child process here.
loop:   li $v0, 4       # syscall 4 (print_str)
        la $a0, msg4     # argument: string
        syscall
        li $a0, 0      # wait for any childprocess.
        li $v0, 19
        syscall
        beqz $v0, loop  # $v0 == 0, child is still running.
        bgtz $v0, loop  # $v0> 0 one child has terminated, continue waiting
            .

exit:   li $v0, 21
        syscall
```

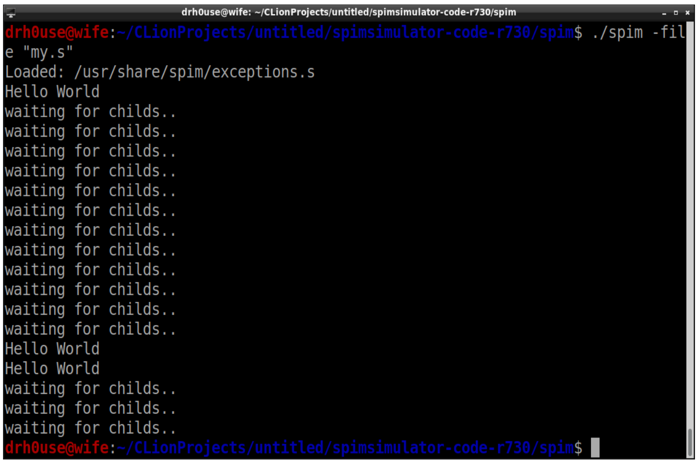

Figure 6

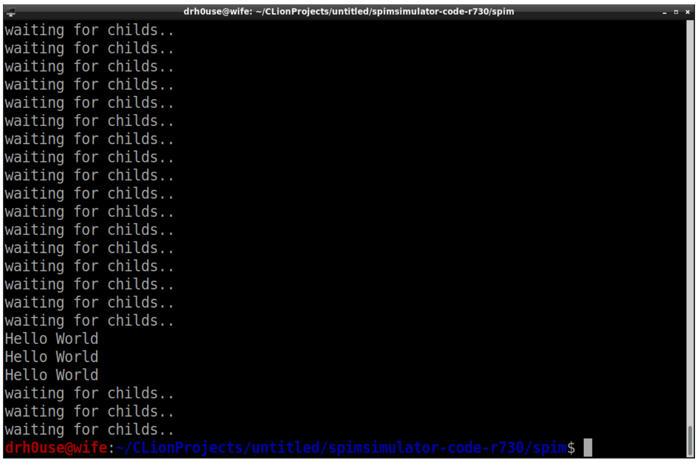

Figure 7

- There are 3 different programs in our inventory, let's show that they are working properly as follows before filling the kernel. They are regular programs (nothing fancier than print "Hello, World!") that are just working perfectly fine. Hence no need the overdo it.

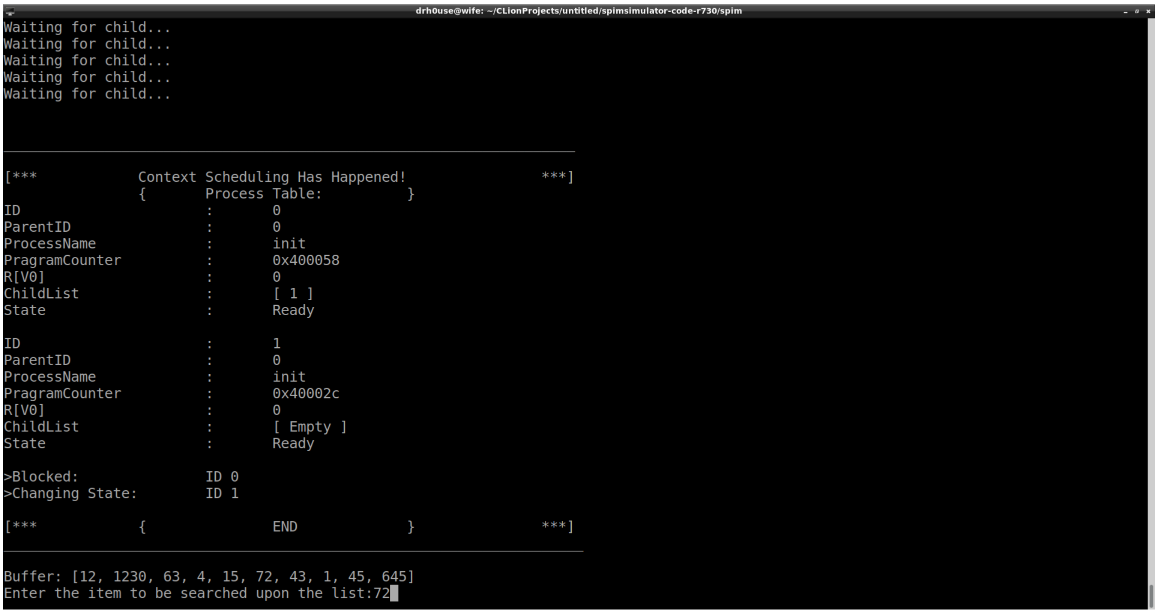

Figure 8

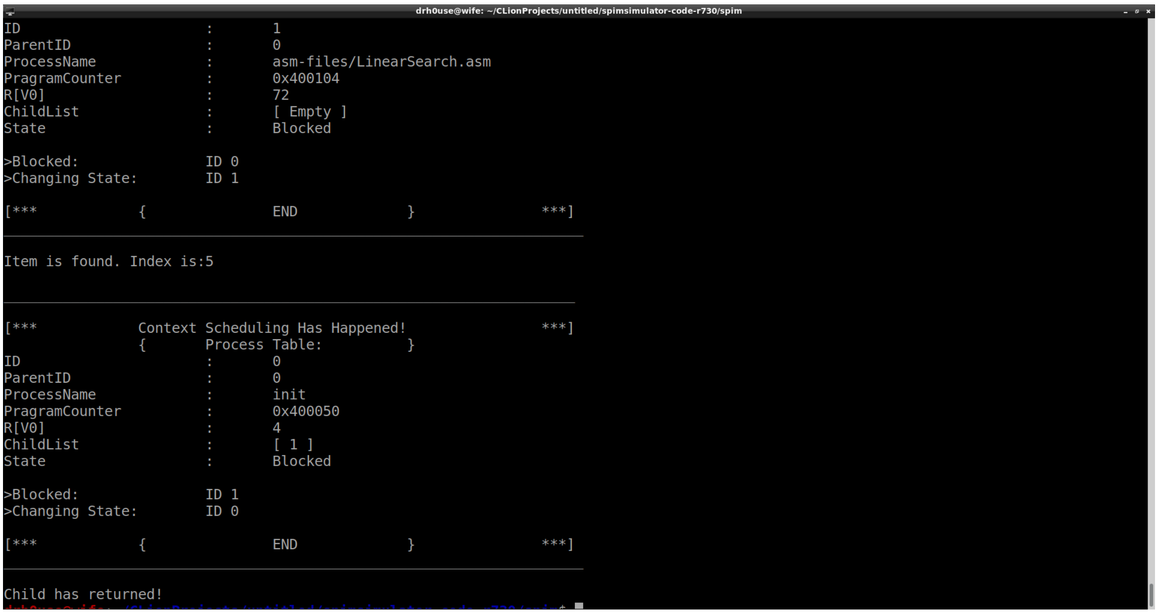

Figure 9

That's all for the unit tests and kernels will be explained in the demo video.