TDDB68/TDDD47 - Lab 5

Felipe Boeira

February 2020

1 Goal

In this assignment the `wait()` system call will be implemented. This allows parents to wait until a given child has exited and then receive its return value (`exit()` status). You will use (or improve) the data structure you have previously developed in lab 3 to perform this task.

2 Overview

This assignment covers:

- Understanding how `wait()` works
- Implementing `wait()` using your parent and child relation data structure
- Handling different cases where a child or a parent may exit before one another
- Performing input validation

3 Assignment in Detail

You should know the pintos process creation and termination flow very well by now, therefore handling `wait()` may be clear if you keep in mind a few points:

- A parent may only call wait on one of its children
- The parent is put to sleep (possibly using one of the synchronisation primitives) and awake when the child returns (calls `exit()`)
- The child may `exit()` before the parent calls `wait()` (see Figure 1)
- The data structure that represents the parent and child relationship must be kept alive until both processes exit (recall Figure 1, if the structure had been freed when the child exited, then the parent would be unable to access its return code when it calls `wait()`)

- You may use an alive count to determine when the data structure should be freed (e.g. initialise it with value 2 and decrement when the parent or child terminates, when it reaches 0 it means both have terminated and the structure may be freed)

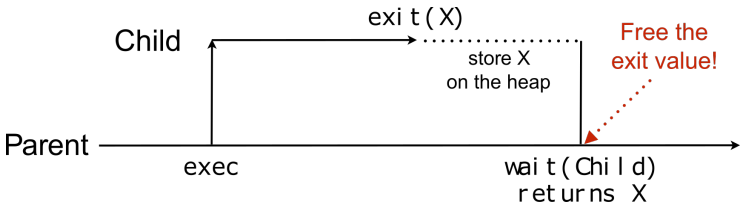

Figure 1: Parent and child wait/exit

4 Input Validation

In this part of the assignment you must make the operating system robust by validating the arguments to the system calls. If validation is not implemented a user program may execute `create((char*)0,1)`, for instance, causing the kernel to crash when trying to read the name of the file to be created. Your task is to validate all arguments passed by user programs to the interrupt handler via system calls.

Another example: `read(STDIN_FILENO, 0xc0000000, 666)` writes data to kernel space (will likely overwrite important data). Hence, ALL pointers from the user processes to the kernel must be checked! Including the stack pointer, strings, and buffers.

A valid pointer from a user process is:

- Below PHYS_SPACE in virtual memory (not kernel memory)
- Associated with a page in the page table for the process (use pagedir_get_page)
- If some pointer is not valid, then terminate the process! The exit status is then -1

Make sure to check if the whole buffer is valid when a user supplies an array, for example.

5 Testing

Once you have finished this assignment you can run the tests by issuing `make check`. Please note that you must also finish the previous assignments before you can run the tests; otherwise all tests may fail. For the test script to work you need to add:

```
printf("%s: exit(%d)\n", thread->name, thread->exit_value);
```

to the code that is executed when a user process exits or is killed.

When you have finished this lab all 50 tests should pass. As always when testing: passing all tests is not a guarantee that the code is correct.

6 Helpful Information

Code directory: `src/userprog`, `src/threads`, `src/lib`, `src/lib/kernel`
Textbook chapters: Chapter 2.3: System Calls
Chapter 2.4: Types of System Calls
Chapter 4.4: Threading Issues
Chapter 6.2: The Critical-Section Problem
Chapter 8.4: Paging
Documentation: Pintos documentation related to Project 2
(Always remember that the TDDB68 lab instructions always have higher precedence)

7 Acknowledgement

This document is based on previous content from the TDDB68 website.