TDDE47/TDDE68
Lab 5: wait & exit

Dag Jönsson

February 18, 2024

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 extend `exec` so that a parent can keep track of it's children.

2 Overview

This assignment covers:

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

3 Preparatory Questions

Before you begin doing your lab assignment, you have to answer the following set of questions to ensure that you are ready to continue:

- How are you going to track all children of a given parent and the parent of every children?
- What is the information that you will need to include in the data structures to be shared between parent and children?
- Which synchronization mechanisms can be applied?

4 Assignment in Detail

Implement/extend the following system calls:

- `int wait (int pid)`
  Let a process wait for one of it's child processes, one the child has exited, return the exit status of the child.

- `void exit (int status)`
  Terminates a program and deallocates resources occupied by the program, for example, closes all files opened by the program. You inherit this system call from earlier labs and extend it.

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 at a time.
- 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()`)
- The data structure that represents the parent and child relationship should **not** be the same structure implemented in the `exec()` lab.
- Neither the parent or child processes should have a pointer to each other's thread struct.
- 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)

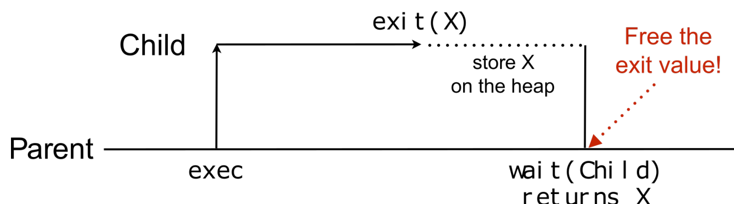

Figure 1: Parent and child wait/exit

5 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.

6 Testing

Once you have finished this assignment you can run the tests by issuing `make -j check`, which will run a fairly extensive test suite. It is very likely that you will not pass all of these tests at first, which means you will have to go back and fix any potential errors in your solutions from earlier. Please note that you must also finish the previous assignments before you can run the tests; otherwise all tests may fail.

Some things to check if you are failing all the tests:

- Remove any debug `printf` that you might have added.
- Make sure that the thread name is set properly in the thread struct. It should only be the binary name.
  You will also have to add the following

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

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

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

7 Helpful Information

Code directory: `userprog`, `threads`, `lib`, `lib/kernel`
Textbook chapters:

- Chapter 2.3: System Calls
- Chapter 4.6: Threading Issues
- Chapter 6.2: The Critical-Section Problem
- Chapter 9.3: Paging

Documentation: Documentation related to Project 2
(Always remember that the TDDE47/TDDE68 lab instructions have higher precedence)

8 Acknowledgement

Parts of this document contains material from the TDIU16 course at LiU, previous lab instructions found on the course web page, or from previous lab instructions written by Felipe Boeira.