TDDE47/TDDE68
Lab 1: Command line

Dag Jönsson

January 24, 2024

1 Goal

Now that you have gotten Pintos running, you need to solve the problem that we worked around in the first lab with the changes to `userprog/process.c`.

When we run programs, we can send them instructions through the use of command line arguments, for example: `cat userprog/process.c`, where the `userprog/process.c` will end up as an argument to the program `cat`. Excatly how those are made available to the program will be covered in this assignment. The main goal is to understand how the arguments are organized in the x86 architecture, and implement the parsing layout of arguments when spawning new processes in Pintos.

2 Overview

This assignment covers:

- Understanding the memory layout of arguments in the x86 architecture
- Parse arguments received in a `char` array
- Pushing arguments to the stack according to the calling convention

3 Preparatory Questions

Take some time to go through these questions before running into the code:

- What are `argc` and `argv` that `main` in a **C** program takes as parameters?
- Where are they stored when the program begins executing?
- How will you calculate the space needed for the parameters?

- Take a look at the `strtok_r()` function in `lib/string.c`. Give a short explantion of how it works.

4 Assignment in Detail

A user program may be called with arguments in the command line. Implement arguments passing, so the arguments of a user program can be accessible within it (**details**).

First, you should open `userprog/process.c`, find the function `setup_stack()` and change back the following line:

```
*esp = PHYS_BASE - 12;
```
into
```
*esp = PHYS_BASE;
```

Now your program will always fail until you implement argument passing. Try to run any user program.

Stop! Before continuing, think about why you have KERNEL PANIC after you have removed "-12". What is wrong and why did the program work before?

The user program with arguments should be called with apostrophes (') from the Pintos command line. Consider we have a program called "binary", then we run `pintos -- run 'binary -s 17'`, where binary is called with arguments `-s` and `17`. Figure 1 depicts an example of the stack layout when executing such command, your task is to build such layout when spawning a new process according to *char* cmd_line*.

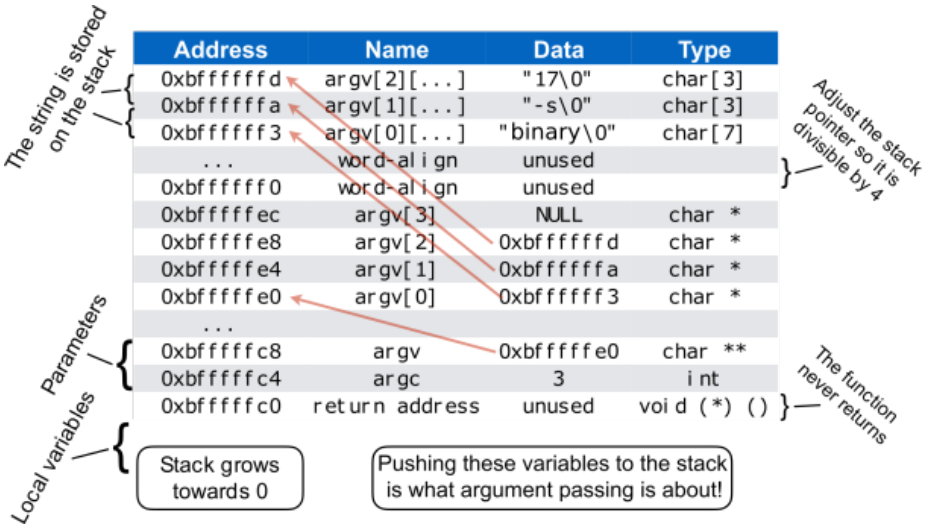

Figure 1: Argument passing details

Note: The figure might suggest that the adresses `0xbffffffc9` through `0xbffffffdf` should be left empty, this is not the case, it's just a bug in the image.

Although you can parse the string from the command line in the way you prefer, we recommend you use the function `strtok_r()`, declared in `lib/string.h` and implemented with thorough comments in `lib/string.c`. You can find out more about it by looking at the man page (run `man strtok_r` in a terminal). We suggest that you limit the number of arguments to, for example, 32, which will simplify your implementation because you can use a static array of a fixed size to store the parsed arguments.

Necessary details about setting up the stack for this task you can find in **3.5.1 Program Startup Details** section of the Pintos documentation.

There is one more thing you need to solve as part of this assignment. When Pintos need to load the binary into memory it calls on `load()` in `userprog/process.c`, which requires the argument **char *file_name** to only contain the actual file name and nothing more. As it is implemented now it will be sent the entire command line which will cause it to fail if any arguments are given.

5 Testing

To test your solution and also confirm that your solution works as expected, you can use the debug function `dump_stack()` that is defined in `userprog/`

`process.c`. Add a call to the function after you have created the stack according to the above specification, passing the stack pointer as a parameter.

In the `examples` directory you will find some example programs that can be run in Pintos. A program you can use to test your solution is `examples/noop.c`. Since you haven't implemented the system call SYS_WRITE yet, the `examples/lab1test.c` file will not work as expected.

`cd` into the `examples` directory, run `make -j` to compile all the programs, `cd` into the `userprog` directory, run `make -j` here to build Pintos, run the program as documented in the `examples/noop.c` file.

The output should be similar to Figure 1. Make sure that you test your solution thoroughly.

6 Helpful Information

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

- Chapter 2.3.2: Application Programming Interface
- Chapter 3.1: Process Concept

Documentation: Pintos documentation
(Always remember that the TDDE47/TDDE68 lab instructions have higher precedence)

7 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.