ECE 391, Computer Systems Engineering
MP3 Checkpoint 3 Hints

General Guidelines

This document is intended to clarify our expectations for the demo procedure. If you have any feedback to make the document more clear and concise, please let us know on Piazza.

1. It's time to execute user programs! We will not run your tests. Instead, we will run a few of the user programs given to you. On startup, you must be running the `shell` program.
2. You must the `execute` system call running. If you don't, but have other system calls running, you **must have tests to demonstrate that they work** to earn any functionality points.
3. **Read the Appendix very carefully!**
4. We suggest that you get comfortable with function pointers and writing assembly code in separate .S files, as inline assembly can be tricky and lead to subtle bugs in your code.
5. Remember to take advantage of the functionalities C and x86 Assembly offer you, **code smart!**. While it may be amusing to write all of the system calls in assembly, it is hard to do so and to debug. Instead, write most of the code in C and use either asm volatile or call assembly functions from your C functions.
6. Remember to maintain your **bug log**! While we know that you are capable programmers, we know that everyone has bugs. If you tell us that you had no bugs and hence have none in your bug log, we won't believe you.
7. As always try to use your best style and document code as you write it.
8. Although the doc for this checkpoint splits the description up into five sections, but parts 6.3.2, 6.3.3, 6.3.4 and, 6.4.5 are all portions that you must implement while implementing the system calls as a whole, mostly in `execute` and `halt`.
9. Remember to validate your input parameters!

System Call Handler

1. Make sure that you have proper assembly linkage for the IDT entry for system calls!
2. You should be verifying that the system call number passed in through `eax` is valid.
3. You **must** use a jump table to execute the right system call here.

Execute

1. If you are having a tough time debugging here, which you most likely will, its always a good idea to take a step back and see what every block of your code is doing at a high level. Making helper functions here is highly encouraged as that helps you test individual portions of what the system call should be doing as a whole.
2. Though you don't have to use the arguments until the next checkpoint, it might be useful to store them somewhere(!?) that you can use later.
3. When reading in the start address of the program make sure you read the bytes in the right order!
4. Usually each user program is given its own page directory, for this MP you do not need to do this. You can use the same page directory and just modify the few entries in it to adapt it for the user program you're running.
5. Make sure that the address of the first instruction for the user program is an address in the user page that you allocated! RTDC to find where this page should be.
6. Though we will be primarily testing using `shell` and `testprint` in this checkpoint, its highly advised that you run as many of the user programs as possible to test if all your system calls work! `shell` uses the `write`, `execute`, and `read` system calls. `testprint` uses the `write` system call. All user programs call the `halt` system call on termination. Other user programs will be using other system calls. Ex: `cat` and `grep` use the `open` and `close` system calls but also require you to have finished the `getargs` system call which isn't required until next week.
7. You must be squashing erroneous user programs. This means that if a user program exits unexpectedly you have to tell `shell`(the parent process) that they did so, so that `shell` can handle the failure properly: in our case, display the correct prompt. You can use the `sigtest` user program to test if you're squashing programs correctly (you will need `getargs` working for this, try "`sigtest 0`").
8. Don't forget to flush the TLB!

Halt

1. Remember that `halt` must return a value to the parent `execute` system call so that we know how the program ended.
2. Halting the base shell should either not let the user halt at all, or after halting must re-spawn a new base shell. This way you must ensure that there is always one program running.

Open, Read, Write & Close

1. Remember to validate the input variables!
2. These system calls, except `open`, will be very small, typically not more than 10 instructions long.
3. The `open` system call must initialize the file descriptor array entry for the file descriptor(the index into the file descriptor array) that it is going return appropriately based on the file type.
4. While setting up the fops table in the fda, you **MUST** use static tables with the function pointers already defined and assign the correct one to the fops table for that particular file descriptor.
5. It is important that you always call the corresponding driver `open`, `close`, `read`, and `write` functions, even if they do not do anything! For the open and close functions for the terminal, you can just call a "`bad_call`" function that always returns -1.
6. You might wonder where you should store the fda. Remember that each process has its own fda. That should point you in the right direction ;)
7. On backspacing while `shell` is accepting input, you should not backspace over the shell prompt that shows up.