CSCE 313—Fall 2023
Homework
Due November 5, 2023

Instructions

- For each question submit both the code/program and its corresponding output. Deductions will be made for code that contains errors.
- This assignment should be done individually. If you copy code from a classmate or use unauthorized sources and present it as your own work, it will be considered as a violation of the Honor Code.

1. (20 points) Write a stub in x86-64 assembly for a new function `openr()` that sets up arguments for the Unix `open` system call with the `O_READ` flag and returns the system call result to the caller. See Section 5.4 of the class notes for an example of how to do this. The signature for the function `openr` is the following.

   ```
   int openr(const char *pathname)
   ```

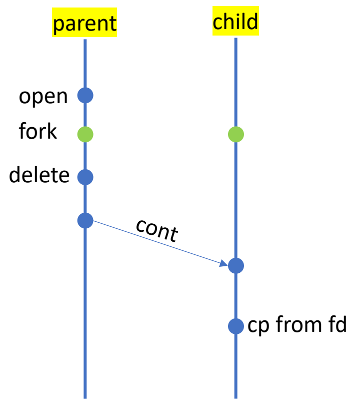
   To demonstrate the correct working of `openr`, write a program that takes a pathname as an argument, uses your `openr` function to open the pathname for reading, and then copies the contents of the file to `stdout`.

2. (20 points) In this problem we'll compare the overhead of a *system call* with that of a *function call*. One area where researchers have tried to reduce system call overhead is in scheduling of threads when a program uses lots (tens of thousands) of threads. The idea is to create *user-space* threads so that scheduling and context switching of threads happens in the application program instead of in the kernel.

   Write a program that measures the performance of a function call and compare it with that of a system call that does very little work in the kernel, for example the `getpid` system call. Section 6.3 of your class notes may help you get started. You may (and should) research other ways of accurately measuring time spend on behalf of a process in Unix. Document your technique of measuring time for this problem.

   Tabulate the relative performance (mainly the ratio of the average time/invocation) of a *function call* versus that of a *system call* on a few different Unix machines. Which one is faster and by how much? Use a large number of invocations in each case and take the average. Briefly explain your results.

3. (20 points) Write a program that demonstrates reference counting on filesystem objects. Your program should be written as a communicating parent-child program illustrated by the following diagram.



The parent does the following. It

- opens an existing file for reading,
- `fork`s so that the just opened `fd` for the file is available in the child,
- closes and deletes the file. Use the `unlink` system call to delete the file. Verify that the file is no longer available using the `ls` command.
- sends a *synchronization* message to the child over a `pipe`, so the child can now continue.

When the child continues, it should still be able to read the file even though the file is (seemingly) deleted and is no longer visible via `ls`. Briefly explain what might be happening?

4. (20 points) What is the importance of inodes in the Unix file system? Write a program that accepts user input for a file path and displays the following metadata for the provided path.

   - whether the provided path corresponds to a file/pipe/symlink.
   - the permissions of the *owner* of the file.

   Hint: Use the *lstat* function and the *struct stat* data structure.

5. (20 points) Write a program to determine the maximum number of open files permitted by a process in your favorite Unix-like operating system. You can repeatedly open a file until the open call results in an `EMFILE` error. How will you test that the open system call failed with this error? Briefly explain. What is the maximum # of open files that a process can open on your GCP Ubuntu VM?