HW2 - Exploring System Calls

Note, you can find the man pages (documentation) for most linux system calls here:
http://man7.org/linux/man-pages/man2/syscalls.2.html
The calls used by the mac OS are usually very similar if not identical to these.

You can find some more specific Mac/Unix detail on the system calls here:
https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/
if you click the "Section 2" link. I often find it helpful to read both.

First, how to use DTrace on my research machines:

- log in to `sgoings48570` on mathcs.carleton.edu.
- log in as user `OS-student`. You can do this from any terminal on campus by typing
  `ssh OS-student@sgoings48570.mathcs.carleton.edu`
  the password is `goingsCS332`
- if you are off campus you will have to first connect to Carleton's network through VPN, instructions on how to do so can be found here:
  https://wiki.carleton.edu/display/itskb/Off-campus+Access+to+Network+Drives
  note you want the instructions for VPN on the right side, the WebDAV won't work for this purpose.

- THIS NEXT STEP IS VERY IMPORTANT, DO IT FIRST!
  Create a directory for yourself named with your username and COPY (don't move) the files from sgoings to your own directory. You can do this using

  ```
  mkdir mydir
  cp sgoings/* mydir/
  ```

  Now everything you do should be in your own directory. Please do not use this for anything but running the necessary dtrace commands. Don't upload any files here, or even create a file here to answer the homework questions, keep that on your own computer. Be warned that files on my machines may be erased at any time!

- The DTrace commands you can use:
  - `errinfo` – shows all errors encountered by the OS
  - `dtruss` ... – allows you to trace the system calls for a process
  - note that all of these commands require root permissions to run, so you will have to add the word `sudo` before each.

- To trace the calls for a single command
  - `sudo dtruss <cmd>`, e.g.
    - `sudo dtruss ls`
    - `sudo dtruss python <aprogram.py>`

- To trace the calls for a process by name
  - `sudo dtruss -n process_name`, e.g.
  - `sudo dtruss -n mds`
  - `top` will show you the most active processes currently running. The name of the process under the `COMMAND` column is the name you can use with `dtruss -n`.

- To trace the calls for a process by pid

- `sudo dtruss -p pid`, e.g.
- `sudo dtruss -p 7569`
- again you can use `top` to get the pid of currently running processes.

- **To trace the Python copyfile programs (questions 1-3)** - you cannot use the above trick directly to see the system calls made by the Python examples you will run because the Python program is forked off into a new process and you are only tracing the calls made by the bash shell itself. You can do something similar though, explained here:
  - the copyfile programs all ask for an input of the filename to copy before they do the actual copying, so in your first terminal you can start the program normally (e.g. `python copyfile.py`) and it will ask you for an input file.
  - before typing in the filename, switch to your 2nd terminal and run the `top` command, note the pid of the `Python` process.
  - still in the 2nd terminal, run `dtruss` on the pid you just noted.
  - now return to the 1st terminal and input the filename to continue the program, the 2nd terminal will show you all system calls made by the Python program from that point on.

- **To trace a basic shell command (question 4)** – Unfortunately I haven't discovered any trick like the above to see only the system calls associated with a shell command, but the system calls for the trace stuff itself are all before the calls for the command, so just scroll past those.
  - note that the output from the command (if any) appears first, then the system calls. So for example if you run `sudo dtruss ls` the very first few lines will be the actual list of files and the rest will be the system calls.

Questions to Answer:

1. Run a dtruss trace **as described above** on the Python program copyfile.py for both test.txt and longtest.txt. For *test.txt* you should get similar results to the calls I passed out in class. Specifically there are 2 **read** syscalls on the file *test.txt*, the first reads all 29 bytes and the 2nd 0 bytes. Then there is 1 **write** syscall that writes all 29 bytes to *out.txt*.
   a. How many **read** and **write** syscalls occur when you run the program on *longtest.txt*, and how many bytes does each read/write?
   b. Explain why the behavior is different for the longer file.

2. Open copyoneline.py to see that it reads the first line from the input file and writes that line to *out.txt*. Run a trace on this program using *longtest.txt*.
   a. How many **read** and **write** syscalls occur when you run the program on *longtest.txt*, and how many bytes does each read/write?
   b. Explain why the number of bytes read may be different than the number written.

3. Open cpflinebyline.py to see how it reads one line at a time from the input file and immediately writes that line to the output file. Run a trace on cpflinebyline.py using *longtest.txt*.
   a. How many **read** and **write** syscalls occur when you run the program on *longtest.txt*, and how many bytes does each read/write?
   b. Is this program actually copying the file line by line? Explain how/why it behaves the way it does.

4. Run dtruss on any **2** of the unix commands you explored in HW1. There will be lots of output and system calls you won't understand, but for each one (**don't forget to do 2)** pick a few lines that you can explain pertinent to that command and describe what they do. Generally the output pertaining to the actual unix command will come toward the end, skip past anything to do with opening urandom and dtrace files, and remember to **read** the man pages for the system calls. The general format for a system call is `name(args) = returns`

   For example, my answer for "ls" might be:

   1. `open_nocancel(".\0", 0x100004, 0x0)  = 5 0`
      - this opens the first argument "." (which is the current directory, the "\0" simply marks the end of a string) as a file and places it at index 5 in the table of open files (5 is the first return value, in this case the index into the file table).

   2. `fstatfs64(0x5, 0x7FFF5FBFEAA0, 0x0)  = 0 0`
      `getdirentries64(0x5, 0x100800600, 0x1000) = 764 0`
      `getdirentries64(0x5, 0x100800600, 0x10s00)= 0 0`
      `close_nocancel(0x5)  = 0 0`
      - the first line gets info about the file opened above, and the second reads the directory entries for '.' to the buffer specified by the given address, returning the number of bytes successfully read. The second call to getdirentries gets nothing and so writes nothing, and the 0 return means all entries have been read. The final line then closes the file.

   3. `write_nocancel(0x1,`
      `".CFUserTextEncoding\tDesktop\t\t\tMusic\t\t\tcpflinebyline.py\n\0",`
      `0x37)  = 55 0`
      `write_nocancel(0x1,`
      `".bash_history\t\tDocuments\t\tPictures\t\tlongtest.txt\n\0", 0x31)`
      `= 49 0`
      `write_nocancel(0x1,`
      `".bash_profile\t\tDownloads\t\tPublic\t\t\tout.txt\n\0", 0x2B)`
      `= 43 0`
      `write_nocancel(0x1,`
      `".emacs.d\t\tLibrary\t\t\tSites\t\t\ttest.txt\n\0", 0x25)  =`
      `37 0`
      `write_nocancel(0x1,`
      `".ssh\t\t\tMovies\t\t\tcopyfile.py\t\tunixcmds.s\n\0", 0x28)`
      `= 40 0`
      - These lines write the data from the buffer the file entries were read into in step 2 above to the standard output (specified by file id 0x1). \t writes a tab and \n an endline, which is how these are printed for the user nicely formatted.

      Just #3 above would have been a sufficient answer. Or any 2 of the lines from #1 and #2, or any other few lines.