UNIX Concurrent IO Simulator Exercises

This document was derived from simulation software created by Steve Robbins which was supported by NSF DUE-9752165

Instructions: Download the UNIX Concurrent I/O simulator and extract the zip file. This produces a folder named **io**. The user manual for this simulator, **io_doc.html**, is included in the folder. It is strongly suggested that you carefully read through this documentation describing how the simulator operates prior to beginning this section.

Steps: Perform the following steps using the concurrent I/O simulator:

1. In the **io** directory you will find the file **ioconfig**. This is the configuration file for the concurrent I/O simulator. Add a line at the top of this file that begins with **user** followed by your name. For example, if your name is Avi Galvin, you would enter the following:

   user Avi Galvin

   Save and close the configuration file. This will ensure any log files you generate will have your name at the top.

2. In the **io** directory, execute `runio`(UNIX, Linux, Mac OS X) or `runio.bat` (Windows.) This will start the concurrent I/O simulator.

3. Click the button labeled **Open Log** in the left-hand column of the UNIX Concurrent IO Simulation window. After you have pushed this button, its name will change to **Close Log**.

4. Click on the button labeled **Run** in the right-hand column of the main window. This will run the first program of the simulator which forks a child process.

5. After the simulator runs this program, click the button **Log Image** in the left-hand column of the main window.

6. Click the pink button labeled **Reset** in the right-hand column of the main window.

7. Next, click the button labeled **Step** at the top of the right-hand column of the main window. The **Step** button executes the line of code pointed to by the red arrow in the program. Click this button several times so that both the parent and child processes complete execution.

8. Reset the simulator once again.

9. At the top of the middle column of the main window is a button which begins with the label **After Fork:** The default value of this button should be **parent**. Click this button so that it changes to **After Fork: child**. The value of this button determines whether the parent or child process runs first after the call to `fork()`. Run the program again.

10. Click the pink button labeled **Choose Program** in the lower right-hand corner of the main window, opening a pop-up window that allows you to choose from a list of programs the simulator may execute. Choose **Program2** from this list.

11. Run the program again, and then log the image.

12. Click the button labeled **After Fork:** to reset its value to **parent**. Reset the program, and then run it again. After this program has run, log the image.

13. Choose **Program3**.

14. Run the program again, and then log the image.

15. Reset, then run, the program. After this program has completed, log the image.

16. Choose **Program4**.

17. Run the program, and then log the image.

18. Resent the program and experiment with this program by altering whether the parent or child process runs after the fork.

19. Close the log file by clicking on the **Close Log** button. You have created a file called **logfile01.html** in the **logs** directory within the io directory.

20. Click the pink **Quit** button in the upper left-hand corner of the main window to terminate the simulator.

21. Submit your log file -- **logfile01.html** -- to your instructor per his or her instructions.

Questions:

Answer the following questions after completing Steps 1-21 above.

1. After completing Step 4, which process ran first?

   a) Parent process
   b) Child process

2. After completing Step 4, what is the value of `buf` for both the parent and child processes?

   a) ab
   b) cd
   c) abcd
   d) cdab

3. After completing Step 7, which of the following statements best explains the final value of `buf`?

a) Since the parent and child are the same program, we can expect the same output.
b) Since the `open()` system call is done after `fork()`, the parent and child each open separate copies of the same file.
c) Since the `open()` system call is done after `fork()`, the parent and child each open separate copies of a different file.
d) None of the above

4. After completing Step 9, does the process which runs first affect the final value of `buf` for both the parent and child processes?

a) Yes
b) No

5. After completing Step 11, notice the final value of `buf` for the parent and child processes is different? Which of the following statements best explains why it is different?

a) The parent and child programs are the same -- the different output makes no sense.
b) Because the `open()` system call is done before `fork()`, the parent and child processes are sharing the same file descriptor and are sequentially reading from the same file.
c) Because the `open()` system call is done before `fork()`, the parent and child processes each open separate copies of the same file.
d) Because the `open()` system call is done before `fork()`, the parent and child processes each open separate copies of a different file.

6. After completing Step 12, does the process which runs first affect the final value of `buf` for both the parent and child processes?

a) Yes
b) No

7. After completing Step 14, what is the final value of `outfile`? (It appears on the right-hand side of the main window under **Disk Space**.)

a) abcd
b) ABcd
c) abCD

d) ABCD

8. After completing Step 14, which of the following statements best explains the final value of the file `outfile`?

a) Both the parent and child processes open the same file with the same inode, although each process writes different values to the file.
b) Both the parent and child processes open different files with the same inode; the process that finishes last determines the final value of `outfile`.
c) Both the parent and child processes open the same file with the same name but different inodes; the process that finishes last determines the final value of `outfile`.
d) Since the open() system call is done after fork(), the parent and child processes each open a copy of the same file.

9. After completing Step 15, does the process which runs last affect the final value of the file `outfile`?

a) Yes
b) No

10. After completing Step 18, does the order of which process runs first affect the final value of the file `outfile`?

a) Yes
b) No

11. After completing Step 18, which of the following statements best explains the final value of the file `outfile`?

a) The parent and child programs are the same -- the different output makes no sense.
b) Since the `open()` system call is done before `fork()`, the parent and child processes share the same file descriptor and are sequentially writing to the same file.
c) Since the `open()` system call is done before `fork()`, the parent and child processes each open separate copies of the same file.
d) Since the `open()` system call is done before `fork()`, the parent and child processes each open separate copies of a file with the same name, but different inode values.

12. The button with the label **Active:** in the right-hand column can be used to determine which process is running. Using this button, run **Program4** such that the value of outfile is `abABcdCD`. What is the necessary order of execution (indicated by process number) to generate this output? (You will have to use the **Step** button which will allow you to step through the program line-by-line.)

a) 1001, 1002, 1001, 1002
b) 1002, 1001, 1002, 1001
c) 1001, 1001. 1002, 1002
d) 1002, 1002, 1001, 1001