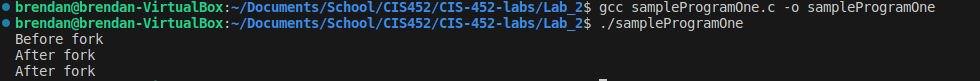
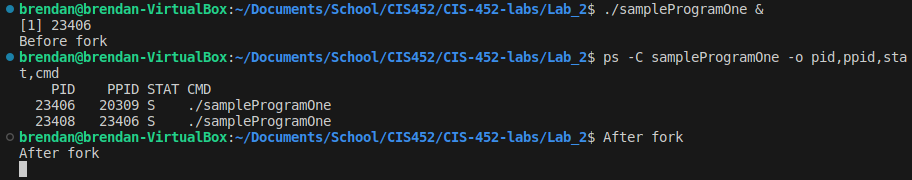
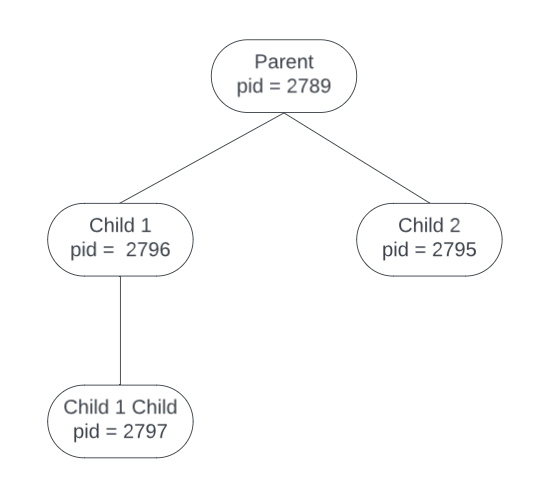
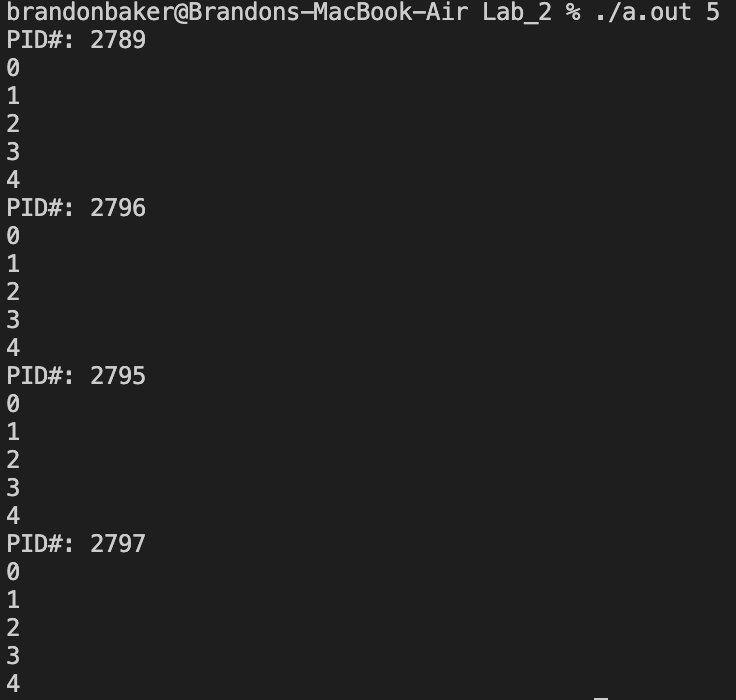
1. 3 lines are printed before fork is printed once and after fork is printed twice 
2. In the sampleProgramOne file, the program executes line by line starting by printing “Before fork” because it is the first line in the main method. In the next line, fork() is called which creates a duplicate process (child) of the original one (parent) that is the exact same. The rest of the execution of both processes continues to run which prints two different “After fork”.
3. There is two separate PID which indicates that there is multiple different processes operating. Since a fork() command is executed in the code a second process that is identical to the first is created. Since both will be from the ./sampleProgramOne cmd but will have two separate PID. Note\* The -o command only shows the specific fields within the ps command so -l isn’t needed.
4. The following is the output of sampleProgramTwo.c being run on a macOS. The diagram process hierarchy diagram is shown beneath it.
5. It can be seen from the output and the children the main process is creating two children from the same state. After the first child is created the parent process creates another child. Alongside the child process creates a child, making a total of 4 processes. The main process will then attempt to finish the program and print up to the variable “limit’s” value which is assigned to the first argument as an integer. It does appear that children in the same level of the hierarchy compete for scheduling with each other as seen in the diagram from question 4. Child 1 and 2 appear to run near the same time while the parent (before the children) and Child 1’s child (after child 1 and 2) execute in the hierarchical order due to not having competition. This gives a chance of multiple outputs for each time the program runs. This is dependent on how the OS’s scheduler chooses to execute the programs.
6. child = wait(&pid);
7. In the finished code, the child prints first, although the parent starts running first. This is because the child program is created with fork() and the pid value is assigned to the child’s pid #. Since the parent process has the pid variable as the child’s pid, the else statement is entered and the wait() system call is executed. This system call then waits for the child process to exit() before the parent continues execution. Once the child process is terminated the parent process finishes.
8. The second print line, “After the exec” is never printed. This is because if there is an unsuccessful attempt to run the program (bad inputs), the program will catch the error before this line and exit using the exit() system call. In the case that the execvp() system call is successful, the process running the main is replaced by the command that was given in the command line. Since it is replaced, the rest of the program doesn’t get the chance to execute and “After the exec” is never printed.
9. Execvp takes two arguments, the first points to the file being executed and the second should be an array of pointers followed by a null pointer (end of argument list). By passing in the address to &argv[1] we give the system call execvp() an array of pointers that will be terminated with a null pointer (as required in the manpage). execvp() will continue to run until the null pointer is reached.

Lab Programming assignment output:

