COEN 383 Project 6

Group 3

(Dylan Hoover, Andrew Knaus, Venkata Sai Srikar Jilla, Sai Preeti Kakuru, Stuti Jani)

For this project a simulation was created for parent and child processes to establish communication using a virtual file called a pipe. These pipes have both read and write file descriptors and exist solely in memory, without being written to the file system. The simulation followed a straightforward procedure: five pipes were generated within the primary process, and then five child processes were forked, with each child being assigned one pipe. In order to fulfill the project requirements, each child process utilized its assigned pipe to send messages to the main process. The main process read from all the pipes and immediately wrote the output to a file named "Output.txt".

The encountered problems were as follows:

1. Difficulty detecting when the child process closed the write file descriptor for its assigned pipe.
   Challenge: The child process needed to close its assigned pipe after the simulation using a "close" system call. The parent process could detect this closure by utilizing a combination of "select" and "read" system calls. When the child process closed its pipe, the "select" system call indicated data at the end of the pipe. However, when the parent read this data, the number of bytes read was "0," indicating the closure of the pipe.
   Issue: An EOF (End-of-File) byte is added to a pipe only when there are no open write file descriptors for that pipe. However, when a pipe is created and a child process is forked, both the parent and child processes have write file descriptors for that pipe, which violates the condition mentioned earlier. As a result, when the child process closes its assigned pipe, it does not add an EOF character, making it challenging for the parent process to detect the closure.
   Resolution: It is necessary for both the parent and child processes to close any unused file descriptors. The parent process should close the write descriptor for the pipe, and the child process should close the read descriptor for the pipe.
2. Sharing of all five pipes between the five children after forking.
   Context: When the main process forks a child process, the entire memory and stack of the main process are duplicated and passed on to the child process.
   Challenge: In our solution, we needed to create five pipes and assign one pipe to each child process. However, this raised the issue mentioned earlier, where both the parent and child processes had a write file descriptor for the pipe, making it difficult for the parent to detect the closure of the pipe by the child. Furthermore, pipes not assigned to a specific child process were also shared with it, complicating the detection of pipe closures by the parent process.

Resolution: Each child process should close both the read and write file descriptors for any unused pipes.

3. Terminating the fifth child after the simulation's end.
   Context: The fifth child process in this project had the special requirement of reading input from the terminal and communicating it back to the parent process. Additionally, the simulation needed to be terminated after 30 seconds by closing the dedicated pipe.
   Challenge: Initially, the "scanf" function was used to read input from stdin. However, "scanf" is a blocking call that waits for user input, halting the program's execution until input is provided. This created a problem when attempting to terminate the child processes at the end of the simulation, as the fifth child process was waiting for user input and unable to continuously check the simulation time due to the dependency on "scanf".
   Resolution: The challenge was overcome by using "select" and "read" functions on "STDIN" instead. This allowed the fifth child process to read from standard input only when data was available, enabling continuous monitoring of the simulation time in a non-blocking manner without relying on "scanf".