CS307 Programming Assignment 1

Akif Işıtan 29354

This programming assignment involves in-order creation of a binary tree of processes. Before going into the details of the implementation, a brief overview of the helper functions is stated below.

printDepth()/printFullDepth(): Print formatted arrows depending on the current depth alongside the lr value. printFullDepth also prints num1 and num2. Printing is done to stderr as stdout is reserved for inter process communication via pipes.

printResult(): Prints depth-formatted arrows alongside result or num1 depending on the useCase variable. Printing is again done to stderr.

isRootNode(): Returns 1 if the current process is the root process and 0 if it is not by checking if the current depth is 0.

isLeafNode(): Returns 1 if the current process is a leaf process and 0 if it is not by comparing current depth to max depth.

The main function can be divided into 5 main parts. Argument checks, handling the leaf nodes, handling left child creation, worker child creation and right child creation. The program first begins by checking if there are exactly 4 command line arguments, the first one being the name of the program **(“./treePipe”)**. If not, the usage info string is printed and the program is terminated. If exactly 4 command line arguments are present, the arguments are parsed, converting **curDepth**, **maxDepth** and **lr** from string to integers.

After the argument check phase, the program continues by printing the **curDepth** and the **lr** value, and displays the “Enter num1” prompt if the current process is the root process. The **num1** variable will either be input by the user in the case of the root process, or read from stdin via pipes. This process will be detailed later on. The program then checks if the process is a leaf node using the helper function. If it is, the process should not create left and right child nodes, but rather only apply computation via a worker child process. To achieve communication between the parent and the worker child, two pipes are created. After creating the pipes and forking a child, the resulting diagram can be seen in figure 1.

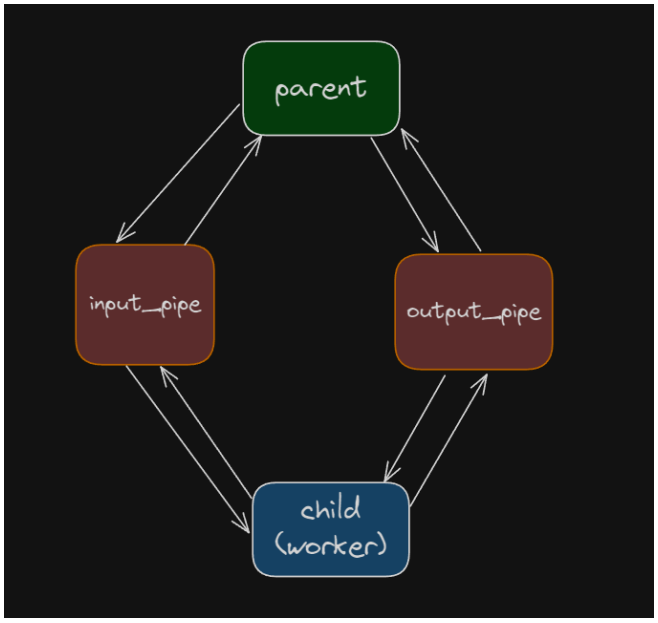

Figure 1. Pipes, parent and children illustration

Next, the parent process closes the read end of the input pipe and write end of the output pipe, writes **num1** and **1** (as the default **num2**) to the write end of the input pipe, and waits for the worker child to finish execution. Simultaneously, the worker child process closes the write end of the input pipe and read end of the output pipe, calls dup2 to redirect stdin to the input pipe's read end and to redirect stdout to the output pipe's write end. The unused closed ends are shown in figure 2. The worker child process then calls execvp with either **"./left"** or **"./right"** depending on the **lr** value to transform itself into one of the operation processes (add, multiply, etc.). After the computation is done, and the program called with execvp prints its result to stdin, which is instead written to the output pipe due to the stdout redirection, the parent process reads this value from the read end of the output pipe, converts it into an integer and prints the result to stderr. If this process is also the root process, the final result is printed instead. Otherwise, the result is printed to stdout, which is redirected to another pipe. This redirection will be mentioned further on. The redirection allows the current parent process' parent process to read the value created by the worker child process.

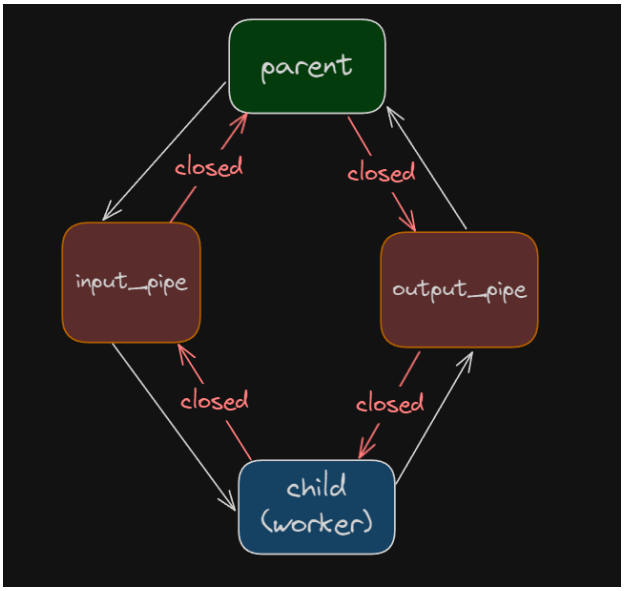

Figure 2. Illustration after the unused ends have been closed

If the process is not a leaf node process, the program continues by creating the left child process. Again, input and output pipes are created for communication, and the parent forks to create the left child. The unused ends are closed, and **num1** is written to the pipe by the parent process. The parent process then waits for the left child process to finish, this wait is what allows for the in-order traversal of the tree. The left child process calls dup2 twice to redirect stdin and stdout to the respective ends of the pipe, and calls execvp, but this time with the program itself as the first argument, that is **"./treePipe"**, which is accessed via **argv[0]**. The **current depth**, **max depth** and **lr=0** values are also passed to execvp as args. The child process converting itself into the program allows for the recursive step to occur, and continues until the base case, the leaf node, is encountered. After the left subtree process is complete, the parent reads the output from the pipe, which the left child wrote to. The output is converted into an integer and is stored as **num2**.

After obtaining **num2**, the next step is for the parent process to apply its own operation using **num1** and **num2**. For this, a worker process is required similar to the previously mentioned leaf node process. Pipes are created, the parent forks again, and creates the worker child process. This sequence of steps is almost identical to the leaf node worker process creation. After the fork, the parent writes **num1** and **num2** to the pipe, then waits for the worker child process to complete. The worker child process calls dup2 twice to redirect stdin and stdout, which allows it to read the written value from the pipe via stdin, then calls execvp with either "**./left"** or **"./right"** depending on the **lr** value. After the worker process finishes, it emits the computation result via stdout to the pipe, then the parent process reads the output of the pipe, converts it into an integer and stores it as **res**.

After the node itself is computed as **res** is obtained, the parent process forks again to create the right node with the exact steps as the left node, just passing **lr=1** to execvp to specify calling **"./right"**. The output of the right worker is read, this is the final result. If the process is the root node, this final result is printed to stderr. Otherwise, the final result is

printed to stdout, which allows for the parent of the current parent process to read the final output from the pipe established between them.

The program continues the steps mentioned recursively until the final result is printed to stderr. This implementation is built upon the details provided in the assignment report, specifically **`2.2 TreePipe Program`**, and matches the output of the sample cases provided.