Ipek Uzun
30837

CS307 PA1 Report

1. **What Code Does Generally?**

The general logic of the program is to simulate a binary tree where each node is actually a process. Each process is responsible for handling its subtree by interacting with left or right processes using pipes. The program begins at the root node, which receives an initial input from the user. This input is then passed to the left child, and the process waits for the left subtree to compute and return a result. Once the result is received from the left subtree, it moves to the computation part using either left.c (addition) or right.c (multiplication) which depends on the current position. When executed this result now is sent to the right subtree and the process waits for this result. Then if this is the root node the final result is printed on the console or if not the result is passed to the parent process. This is a recursive process which follows in-order traversal.

2. **Detailed Implementation Explanation**

At the beginning, the program expects three arguments from the command line: current depth, maximum depth, and left-right which indicates whether this node should execute left or right. Then if we are at the root node, it takes num1 as an input from the user. If it is not a root node it takes num1 from a parent by a pipe.

We move to create a left subtree. If the current depth is smaller than the maximum depth, the node forks a left child. To facilitate communication between the parent and the left child, two pipes are established: pipe_to_child for sending num1 from the parent to the left child, and pipe_from_child for retrieving num2 from the left child to the parent. In the child process, the STDIN is redirected to read from pipe_to_child[0] and the STDOUT is redirected to write to pipe_from_child[1]. After closing unnecessary file descriptors, the child executes another instance of treePipe using execvp(), with updated arguments: incremented curDepth, the same maxDepth, and lr = 0. Meanwhile, the parent process writes num1 into pipe_to_child[1], which the left child reads as input. Once the left child completes execution, it writes num2 to pipe_from_child[1], and the parent retrieves the computed num2 by reading from pipe_from_child[0]. The parent then waits for the left child to finish using wait(NULL), ensuring synchronization. If the node is a leaf node, it does not create a left child and instead assigns num2 = 1, as a default value.

After obtaining both num1 and num2, the TreePipe program must execute a computation program, which is either left or right, based on the lr flag. To facilitate the communication, two pipes, pipe_to_target, and pipe_from_target, are created. The first pipe (pipe_to_target) is used to send num1 and num2 from the parent process to the child process, while the second pipe (pipe_from_target) allows the child process to return the computed result (target_result) to the parent. The program then calls fork(), creating a new child process dedicated to executing the computation. Inside the child process, dup2() is used to redirect STDIN to pipe_to_target[0], ensuring that the program reads input from the parent rather than from the console. Similarly, STDOUT is redirected to pipe_from_target[1], so the computation result is written into the pipe instead of being printed to the screen. The child then executes either the left or right program using execvp(), replacing itself with the target program. Meanwhile, the parent process writes num1 and num2 to pipe_to_target[1], then reads the computation result (target_result) from pipe_from_target[0]. Finally, after ensuring that the child process has been completed by wait(NULL), the program prints the computed result, allowing it to be used next subsequent step. This execution ensures that each node processes its assigned computation before proceeding to the right subtree or returning the final result up the tree.

After computing its intermediate result using either the left or right program, the TreePipe program constructs the right subtree if curDepth < maxDepth. To facilitate communication between the parent and the right child process, two pipes, pipe_to_right and pipe_from_right, are created. The first pipe (pipe_to_right) is used for transmitting the computed result (target_result) from the parent to the right child, allowing it to be used as num1. The second pipe (pipe_from_right) enables the right child to send back its computed final result (final_result) to the parent once the right subtree has completed execution. The program then calls fork() to create a new right child process, which follows a similar approach to the left child. Inside the right child, dup2() redirects standard input (STDIN) to pipe_to_right[0], ensuring it reads input from the parent instead of the console. Similarly, STDOUT is redirected to pipe_from_right[1], so that the result of its computation is written into the pipe instead of being displayed. The child then executes a new instance of treePipe, with incremented curDepth, the same maxDepth, and lr = 1. Meanwhile, the parent process writes target_result into pipe_to_right[1], which the right child reads as its num1. Once the right child finishes execution, it writes its computed result (final_result) to pipe_from_right[1], and the parent retrieves it using read(). The parent then waits for the right child to complete execution using wait(NULL), ensuring proper synchronization. Finally, if the node is the root, it prints the final result to the console. Otherwise, it forwards final_result to its own parent, allowing the computed values to propagate up the tree structure.

3. Functions

I only have 2 functions as a helper. All other computations are done in the main part.

2.1 error_exit(const char *msg)

It handles errors by printing a message and terminating the program.

Used when to check If scanf fails to read input properly, fork fails to create a new process, or execvp() fails to execute.

2.2 print_indent(int depth)

Prints indentation for debugging output to visualize the tree structure. Used every time before printing to console. It should be 3 times the depth.

4. Pipes

I used 6 pipes in total. Each pipe is crucial for transmitting data between the parent and child processes, for the left subtree, right subtree, and executing the left or right program I used 2 pipes for each of them. One is for sending to parents and the other one is for getting from parents. The reason why I chose 2 pipes is to allow data to flow in both directions and work independently when sending and receiving.

3.1 Left Subtree

Pipe1: **Pipe_to_child [2]:**

- Purpose: Sends num1 from the parent process to the left child process.
- Write end (pipe_to_child[1]): The parent writes num1 to this pipe.
- Read end (pipe_to_child[0]): The left child reads num1 from this pipe using scanf().

Pipe 2: pipe_from_child[2]

- Purpose: Transmits num2 from the left child process back to the parent process.
- Write end (pipe_from_child[1]): The left child writes its computed result (num2) to this pipe.
- Read end (pipe_from_child[0]): The parent reads num2 from this pipe using read().

3.2 For Executing left or right programs

Pipe 3: pipe_to_target[2]

- Purpose: Sends num1 and num2 from the parent to the target program (left or right).
- Write end (pipe_to_target[1]): The parent writes both num1 and num2 to this pipe.
- Read end (pipe_to_target[0]): The target program reads them via scanf().

Pipe 4: pipe_from_target[2]

- Purpose: Retrieves the computed result from the target program.
- Write end (pipe_from_target[1]): The target program writes its computed result (res) to this pipe.
- Read end (pipe_from_target[0]): The parent reads this result.

3.3 For Right Subtree

Pipe 5: pipe_to_right[2]

- Purpose: Sends target_result from the parent to the right child as num1.
- Write end (pipe_to_right[1]): Parent writes target_result to this pipe.
- Read end (pipe_to_right[0]): Right child reads num1 from this pipe using scanf().

Pipe 6: pipe_from_right[2]

- Purpose: Retrieves the computed result from the right child after it finishes execution.
- Write end (pipe_from_right[1]): The right child writes its computed result (final_result) to this pipe.
- Read end (pipe_from_right[0]): Parent reads the result.

Ipek Uzun
30837

4. Some Challenges that I encountered

One of the challenges that I encountered was reading inputs from pipes. I got confused about where to use scanf(), read(). But I have learned that scanf() is typically used for reading from stdin, and read() works directly with raw data like from reading from buffer.

Additionally, I struggled with deciding which pipe end to connect to STDIN or STDOUT in each process while using dup2() function. I had to carefully track which end was being used for sending data and which was for receiving. This misunderstanding caused incorrect flow between parent and child processes.