HW1 - Explanation of the treePipe Program

Utku Çağlar / 32579

1 Introduction

The `treePipe` program is a C implementation that simulates a full binary tree structure using processes in a Unix-like operating system. Each node in the tree represents a process that performs computations based on inputs received from its parent and children, following an in-order traversal pattern. This program demonstrates fundamental operating system concepts, including:

- **Process creation** using `fork()`,
- **Process execution** with `execvp()`,
- **Inter-process communication (IPC)** via pipes,
- **Hierarchical process management**.

The primary objective is to showcase how complex tasks can be decomposed into smaller subtasks, distributed across multiple processes in a tree structure, and coordinated through pipes to compute a final result. This report explains the program's design, execution flow, and the role of its key components, providing a thorough understanding of its functionality.

2 Program Overview

The `treePipe` program constructs a binary tree where each process (node) recursively spawns two child processes—one for the left subtree and one for the right subtree—up to a specified maximum depth. It takes three command-line arguments:

```
./treePipe <current depth> <max depth> <left-right>
```

Where:

- `<current depth>`: An integer representing the depth of the current process in the tree. The root starts at depth 0, its children at depth 1, and so on. This argument helps each process identify its position in the hierarchy.
- `<max depth>`: An integer specifying the maximum depth of the tree. When a process's `curDepth` equals `maxDepth`, it becomes a leaf node and does not create further children.

- `<left-right>`: A flag (0 or 1) indicating whether the process executes the `left` program (0, typically addition) or the `right` program (1, typically multiplication), based on whether it is a left or right child.

The program orchestrates the creation of child processes to build the tree, manages their computations, and ensures data flows correctly between them using pipes, adhering to the in-order traversal logic: left subtree, current node, then right subtree.

3 Program Execution

The execution of `treePipe` follows a structured sequence, with each process performing specific tasks based on its depth and role in the tree. Below is a detailed breakdown of the execution phases.

3.1 Root Process Initialization

- **Root Case** (`curDepth == 0`): The root process directly interacts with the user, prompting for an integer input `num1` via the console using `scanf()`. Example output: `Please enter num1 for the root:` .
- **Non-Root Case** (`curDepth > 0`): Non-root processes read `num1` from their standard input (`stdin`), which is redirected from a pipe connected to their parent process.
- **Purpose**: This step establishes the initial input value that propagates through the tree, with the root serving as the entry point for user interaction.

3.2 Handling the Left Child Process

- **Condition**: If `curDepth < maxDepth`, the process creates a left child to build the left subtree.
- **Steps**:
  1. Two pipes are created:
     - `pipe_parent_to_leftchild`: For sending `num1` from parent to left child.
     - `pipe_leftchild_to_parent`: For receiving the left child's result (`num2`) back to the parent.
  2. A new process is forked using `fork()`.
  3. **Child Process**: Redirects `stdin` to the read end of `pipe_parent_to_leftchild` and `stdout` to the write end of `pipe_leftchild_to_parent` using `dup2()`. Executes a new instance of `treePipe` with arguments: `curDepth + 1`, `maxDepth`, and `0` (indicating a left child).
  4. **Parent Process**: Writes `num1` to the write end of `pipe_parent_to_leftchild`. Waits for the left child to complete using `wait()`. Reads the left child's result from the read end of `pipe_leftchild_to_parent` into `num2`.

- **Purpose**: Ensures the left subtree computes its result first, providing `num2` for the current node's computation, adhering to in-order traversal.

3.3 Computation Process

- **Action**: The process performs its own computation by executing either the `left` or `right` program, depending on the `lr` flag.
- **Steps**:
  1. Two pipes are created:
     - `pipe_parent_to_child`: For sending `num1` and `num2` to the computation process.
     - `pipe_child_to_parent`: For receiving the computed result back from the computation process.
  2. A new process is forked using `fork()`.
  3. **Child Process**: Redirects `stdin` to the read end of `pipe_parent_to_child` and `stdout` to the write end of `pipe_child_to_parent`. Executes `./left` (if `lr == 0`) or `./right` (if `lr == 1`) using `execvp()`.
  4. **Parent Process**: Sends `num1` and `num2` to the write end of `pipe_parent_to_child`. Waits for the child to finish using `wait()`. Reads the result from the read end of `pipe_child_to_parent` into `result`.
- **Leaf Node Exception**: If `curDepth == maxDepth`, the process uses a default `num2 = 1` since it has no left child to provide this value.
- **Purpose**: Executes the node's specific operation (e.g., addition for `left` or multiplication for `right`), producing a result that may be forwarded to the right child or parent.

3.4 Handling the Right Child Process

- **Condition**: If `curDepth < maxDepth`, the process creates a right child to build the right subtree.
- **Steps**:
  1. Two pipes are created:
     - `pipe_parent_to_right_child`: For sending the current node's `result` as `num1` to the right child.
     - `pipe_right_child_to_parent`: For receiving the right subtree's final result back to the parent.
  2. A new process is forked using `fork()`.

3. **Child Process**: Redirects `stdin` and `stdout` to the respective pipe ends. Executes `treePipe` with arguments: `curDepth + 1`, `maxDepth`, and `1` (indicating a right child).
4. **Parent Process**: Writes `result` to the write end of `pipe_parent_to_right_child`. Waits for the right child to complete using `wait()`. Reads the final result from the read end of `pipe_right_child_to_parent` into `right_res`.

- **Purpose**: Completes the in-order traversal by processing the right subtree after the current node's computation, ensuring the hierarchical flow of results.

3.5 Final Output

- **Root Case** (`curDepth == 0`): Prints the final result to the console: `The final result is: <right_res>`.
- **Non-Root Case**: Writes the result (`right_res`) to `stdout`, which is redirected to the parent via a pipe.
- **Purpose**: Ensures that only the root process provides the user-facing output, while other processes communicate results upward through the tree.

4 Inter-Process Communication with Pipes

Pipes are the backbone of communication in `treePipe`, enabling the flow of data between parent and child processes. The program uses three distinct pipe pairs:

1. **Parent-to-Left Child Communication**:
   - **Pipe**: `pipe_parent_to_leftchild[2]`
     - Parent writes `num1` to `[1]` (write end).
     - Left child reads from `[0]` (read end) as its `num1`.
   - **Pipe**: `pipe_leftchild_to_parent[2]`
     - Left child writes its result to `[1]`.
     - Parent reads from `[0]` as `num2`.
   - **Role**: Facilitates the left subtree's computation and returns its result to the parent.
2. **Parent-to-Computation Process Communication**:
   - **Pipe**: `pipe_parent_to_child[2]`
     - Parent writes `num1` and `num2` to `[1]`.
     - Computation process reads from `[0]`.
   - **Pipe**: `pipe_child_to_parent[2]`

- Computation process writes its result to `[1]`.
- Parent reads from `[0]` as `result`.

- **Role**: Enables the execution of the `left` or `right` program with the correct inputs and retrieves the output.

3. **Parent-to-Right Child Communication**:

- **Pipe**: `pipe_parent_to_right_child[2]`
  - Parent writes `result` to `[1]` as the right child's `num1`.
  - Right child reads from `[0]`.
- **Pipe**: `pipe_right_child_to_parent[2]`
  - Right child writes its final result to `[1]`.
  - Parent reads from `[0]` as `right_res`.
- **Role**: Propagates the current node's result to the right subtree and retrieves the subtree's final output.

- **Redirection**: In each child process, `dup2()` redirects `stdin` to the read end of the input pipe and `stdout` to the write end of the output pipe. This allows seamless use of `scanf()` and `printf()` for I/O, abstracting the pipe mechanics from the child's perspective.
- **Significance**: Pipes ensure a structured data flow that respects the in-order traversal, making the program a robust example of IPC in a distributed computation model.

5 Function Descriptions

5.1 `print_message()`

- **Purpose**: A versatile utility function for outputting messages to various destinations.
- **Parameters**:
  - `output_type`: Determines the output method (0: stdout, 1: stderr, 2: file descriptor, 3: string buffer).
  - `output_dest`: The target (e.g., file descriptor or buffer).
  - `format, ...`: Variable arguments for formatted output.
- **Behavior**: Uses `vprintf()`, `vfprintf()`, `vdprintf()`, or `vsnprintf()` based on `output_type`.
- **Role**: Enables flexible debugging and result reporting, such as printing to `stderr` for visibility or pipes for IPC.

5.2 `print_depth()`

- **Purpose**: Visualizes the tree structure by prefixing messages with depth-based indentation.
- **Parameters**:
  - `depth`: The current process's depth.
  - `message`: The message to print.
- **Behavior**: Prints three dashes (`---`) per depth level, followed by `>` and the message, using `print_message()` to `stderr`.
- **Role**: Helps trace execution flow and understand the hierarchy, aligning with the assignment's output formatting requirements.

6 Conclusion

The `treePipe` program is a sophisticated demonstration of process management and IPC in a Unix environment. By constructing a full binary tree of processes, it illustrates:

- **Process Creation and Execution**: Using `fork()` and `execvp()` to build and execute the tree.
- **Hierarchical Coordination**: Following in-order traversal to ensure correct computation order.
- **Inter-Process Communication**: Leveraging pipes to manage data flow between parent and child processes.

Each process contributes to a distributed computation, with pipes ensuring that inputs and results propagate accurately through the tree. The program's design highlights the power of breaking down complex tasks into manageable, parallelizable units, making it an educational tool for understanding operating system concepts and parallel processing paradigms.