CS307 PA1 REPORT

Batuhan Güzelyurt 31003

Overview

The program TreePipe has simulated a binary tree of processes in which each node performs some computation depending on its position. Left children perform addition, right ones multiplication, and the result travelled up the tree through the use of inter-process communication through pipes, until the final result was computed at the root.

Program Components and Execution Flow

1. Input Handling and Argument Parsing

The program accepts three command-line arguments:

- `curDepth`: The depth of the current node.
- `maxDepth`: The maximum depth of the tree.
- `lr`: Indicates whether the node is a left (`0` for addition) or right (`1` for multiplication) child.

Here's the code handling the input parsing and validation:

```
if (argc != 4) {
  fprintf(stderr, "Usage: treePipe <current depth> <max depth> <left-right>\n");
  exit(1);
}

curDepth = atoi(argv[1]);
maxDepth = atoi(argv[2]);
lr = atoi(argv[3]);
```

2. Tree Structure and Post-Order Traversal

The program uses a post-order traversal structure:

- Leaf Nodes: At `curDepth == maxDepth`, these nodes use by default `num2=1` and do the computation needed right away.
- Non-Leaf Nodes: First, the left child is created, then the right child, and after collecting the results of both children, non-leaf nodes will perform their computation.

3. Inter-Process Communication

The program utilizes pipes to enable communication between parent and child processes:

- Parent to Child Pipe (`p_to_c`): Sends `num1` and `num2` values from the parent to the child.
- Child to Parent Pipe (`c_to_p`): Returns the result of the child's computation back to the parent.

For every transaction between parent and child, there is a pair of pipes for explicit communication. The pipes are created with the following code snippet:

```
int p_to_c[2], c_to_p[2];
if (pipe(p_to_c) == -1 || pipe(c_to_p) == -1) {
  perror("pipe failed");
  exit(1);
}
```

4. Process Creation and Execution

Each node forks a child process to either recurse or perform an addition/multiplication operation. The child then uses `execvp` to transform into either the `left` or `right` program based on the `lr` argument. Below is the code that handles this transformation:

```
pid_t pid = fork();
if (pid == -1) {
    perror("fork failed");
    exit(1);
}

if (pid == 0) { // Child process
    char *args[] = { (lr == 0) ? "./left" : "./right", NULL };
    execvp(args[0], args);
    perror("execvp failed");
    exit(1);
}
```

5. Root Node Handling

If `curDepth == 0`, the node is identified as the root. It requests an initial value for `num1` from the user and initiates the traversal. Here’s how the root handles input:

```
if (curDepth == 0) {
    fprintf(stderr, "> Current depth : %d, lr : %d\n", curDepth, lr);
    fprintf(stderr, "Please enter num1 for the root : ");
    scanf("%d", &num1); // User input
}
```

6. Leaf Node Execution

For leaf nodes, `num2` is set to `1`, and the node immediately performs a computation. The result is sent back to the parent using pipes. Below is the relevant code for setting up `num2` and sending the result:

```
if (curDepth == maxDepth) { // Leaf node
  num2 = 1;
  dprintf(p_to_c[1], "%d\n%d\n", num1, num2); // Sending num1 and num2
  close(p_to_c[1]);

  // Read result from child
  char buffer[11];
  int n = read(c_to_p[0], buffer, 10);
  buffer[n] = '\0';
  result = atoi(buffer);
}
```

7. Non-Leaf Node Execution

For non-leaf nodes, the program first creates a left child, then a right child, and finally performs the assigned operation. Here's how each child process is created and how the result is managed:

```
pid_t pid_left = fork();
if (pid_left == 0) { // Left child process
  char *args[] = {"./treePipe", curDepth_str, maxDepth_str, "0", NULL};
  execvp(args[0], args);
  perror("execvp failed");
  exit(1);
} else { // Parent after left child
  char buffer[11];
  int n = read(c_to_p_left[0], buffer, 10);
  buffer[n] = '\0';
  num1 = atoi(buffer); // Result from left subtree
}
```

After the left child completes, a similar process is repeated for the right child, receiving `num1` from the left subtree and passing it along.

8. Final Computation for Non-Leaf Nodes

After gathering results from both left and right subtrees (`num1` and `num2`), the non-leaf node performs the final computation based on the `lr` parameter:

```
pid_t pid = fork();
if (pid == 0) { // Child process for computation
   char *args[] = { (lr == 0) ? "./left" : "./right", NULL };
   execvp(args[0], args);
   perror("execvp failed");
   exit(1);
} else { // Parent process receives the result
   char buffer[11];
   int n = read(c_to_p[0], buffer, 10);
   buffer[n] = '\0';
   result = atoi(buffer);
   fprintf(stderr, "%sMy result is : %d\n", indent, result);
}
```

9. Error Handling

The program includes error handling for critical operations, such as `pipe` and `fork` failures:

```
if (pipe(p_to_c) == -1 || pipe(c_to_p) == -1) {
   perror("pipe failed");
```

```
    exit(1);
}
```

If any system call fails, a descriptive error message is printed to `stderr`, and the program exits.

System Calls and Their Usage

Fork and Execvp

- fork: To create child processes for every tree node. Each node's child performs either an addition or multiplication operation or recurses to further child nodes.
- execvp: This will transform the child process into either `left` or `right`, depending upon the `lr` parameter, to execute the required computation.

Pipe and Dup2

- pipe: Provides for both the parent and child processes to pass data between one another. It has two pipes in every node, `p_to_c` and `c_to_p`, to pass data in one direction.
- dup2: Redirects `STDIN` and `STDOUT` to appropriate pipe ends so that nodes read input from their parent and send results back via pipes, rather than the console.

Example Execution Flow (Post-Order Traversal)

In a scenario where `maxDepth = 2`:

1. The root node prompts for `num1`, then creates a left child to compute the left subtree.

2. The left child creates its own children and recursively gathers results, finally performing an addition operation before returning the result to the root.

3. The root then creates the right child, which performs a similar computation and returns a result.

4. Finally, the root combines both results and prints the final output.

Conclusion

The `TreePipe` program efficiently implements a binary tree of processes with post-order traversal and inter-process communication. By leveraging system calls, the program dynamically manages pipes and processes to achieve recursive computations, handling both addition and multiplication operations.