

TreePipe Implementation Report
CS307 - OS - FALL 24-25 - PA1

Barış Pome - 31311

Contents

1	Introduction	3
2	Program Overview	3
3	Detailed Explanation of the Code	3
3.1	Header Files and Macros	3
3.2	Utility Functions	4
3.2.1	Dash Printer	4
3.2.2	Argument Checker	4
3.2.3	Root Input Handler	4
3.2.4	Pipe Redirection Setup	5
3.2.5	Reading from Pipe	5
3.2.6	Value Formatting	6
3.3	Main Function	6
3.3.1	Argument Validation and Initialization	6
3.3.2	Creating Child Processes	7
4	Executing the Computation Program	9
5	Conclusion	10
A	Source Code	11

1 Introduction

This report provides a detailed explanation of the implementation of the `TreePipe` command in `treePipe.c`. The `TreePipe` command simulates a shell command that creates a full binary tree of processes, where each node executes either an addition (`left`) or multiplication (`right`) program, depending on whether it is a left or right child. Each node communicates with its parent or children using Unix pipes, simulating a piped command structure in a command-line environment.

2 Program Overview

The primary goal of `treePipe.c` is to recursively create a binary tree of processes up to a specified maximum depth. Each process represents a node in the tree and performs computations based on its position (left or right child). The communication between processes is managed using Unix pipes, and the processes are created using the `fork()` system call.

Key system calls and functions used include:

- `fork()` to create new child processes.
- `execvp()` to execute either the left (`./left`) or right (`./right`) executable programs.
- `pipe()` and `dup2()` to set up inter-process communication through pipes.
- Utility functions for argument checking, formatting, and I/O handling.

The tree is traversed in a post-order manner, ensuring child computations are completed before the parent integrates the results.

3 Detailed Explanation of the Code

In this section, we delve into the specifics of the implementation, explaining each component of the code in detail.

3.1 Header Files and Macros

The program includes standard header files required for process control, I/O operations, and string manipulation.

```
1 // Barış Pome - 31311
2 // CS307 - OS - 24-25 Fall - PA1
3
4 #include <sys/wait.h>
5 #include <string.h>
6 #include <stdio.h>
7 #include <stdlib.h>
```

```
8 #include <unistd.h>
```

3.2 Utility Functions

Several utility functions are defined to modularize the code and improve readability.

3.2.1 Dash Printer

Function: `dash_printer(int number)`

This function prints dashes to visually represent the depth of the current node in the process hierarchy.

```
1 void dash_printer(int number)
2 {
3     if (number > 0)
4     {
5         for (int i = 0; i < number; i++)
6         {
7             fprintf(stderr, "---");
8         }
9     }
10 }
```

3.2.2 Argument Checker

Function: `argument_checker(int argc)`

This function checks whether the correct number of command-line arguments (`argc`) is provided. The program expects exactly three additional arguments: current depth, max depth, and left-right indicator.

```
1 int argument_checker(int argc)
2 {
3     if (argc != 4)
4     {
5         fprintf(stderr, "Usage: treePipe <current depth>
6             <max depth> <left-right>\n");
7         return 0;
8     }
9     return 1;
}
```

3.2.3 Root Input Handler

Function: `num_for_root(int current_depth)`

This function prompts the user for input if the current node is the root (i.e., `current_depth` is zero). For non-root nodes, it reads the input value using `scanf()`.

```

1 int num_for_root(int current_depth)
2 {
3     int num_root;
4     if (current_depth == 0)
5     {
6         fprintf(stderr, "Please enter num1 for the root: ");
7         scanf("%d", &num_root);
8     }
9     else
10    {
11        // Read num1 for non-root nodes
12        scanf("%d", &num_root);
13    }
14    return num_root;
15 }

```

3.2.4 Pipe Redirection Setup

Function: `setup_pipe_redirection(int pipe_to_child[2], int pipe_from_child[2])`

This function sets up the necessary pipe redirections for inter-process communication. It closes the unused ends of the pipes and uses `dup2()` to redirect STDIN and STDOUT to the appropriate pipe ends.

```

1 void setup_pipe_redirection(int pipe_to_child[2], int
2   pipe_from_child[2])
3 {
4     // Close the unused write end of the pipe to child
5     close(pipe_to_child[1]);
6     // Close the unused read end of the pipe from child
7     close(pipe_from_child[0]);
8
9     // Redirect standard input to the read end of the pipe
10    // to child
11    dup2(pipe_to_child[0], STDIN_FILENO);
12    // Redirect standard output to the write end of the
13    // pipe from child
14    dup2(pipe_from_child[1], STDOUT_FILENO);
15
16    // Close the original pipe file descriptors after
17    // redirection
18    close(pipe_to_child[0]);
19    close(pipe_from_child[1]);
20 }

```

3.2.5 Reading from Pipe

Function: `read_from_pipe(int pipe_fd)`

This function reads data from a given pipe file descriptor and converts it to an integer.

```
1 int read_from_pipe(int pipe_fd)
2 {
3     char buffer[10] = {0};           // Buffer to
4     read(pipe_fd, buffer, sizeof(buffer)); // Read from the
5     close(pipe_fd);                  // Close the
6     return atoi(buffer);              // Convert the
7 }
```

3.2.6 Value Formatting

Function: `format_values(int current_depth, int max_depth, int is_left, char *depth, char *max, char *lr)`

This function formats integer values into strings to be used as command-line arguments for child processes.

```
1 void format_values(int current_depth, int max_depth, int
2 is_left, char *depth, char *max, char *lr)
3 {
4     sprintf(depth, "%d", current_depth + 1); // Format
5     sprintf(max, "%d", max_depth);           // Format
6     sprintf(lr, "%d", is_left);               // Format the
7 }
```

3.3 Main Function

The `main()` function orchestrates the creation of the process tree and manages inter-process communication.

3.3.1 Argument Validation and Initialization

```
1 int main(int argc, char *argv[])
2 {
3     if (!argument_checker(argc))
4     {
5         return 1;
6     }
7
8     // Retrieve program arguments
```

```

9     int l_r = atoi(argv[3]);
10    int max_depth = atoi(argv[2]);
11    int current_depth = atoi(argv[1]);
12
13    dash_printer(current_depth);
14    fprintf(stderr, "> Current depth: %d, lr: %d\n",
15            current_depth, l_r);
16
17    // Initialize variables
18    int num_leaf = 1;
19    int num_root = num_for_root(current_depth);

```

3.3.2 Creating Child Processes

If the current depth is less than the maximum depth, the program creates left and right child processes.

Left Child Process

```

1     if (current_depth < max_depth)
2     {
3         // Left child process
4         int parent_child_l[2], child_parent_l[2];
5         if (pipe(parent_child_l) == -1 ||
6             pipe(child_parent_l) == -1)
7         {
8             perror("Pipe failed");
9             exit(EXIT_FAILURE);
10        }
11
12        pid_t left_child_pid = fork();
13        if (left_child_pid < 0)
14        {
15            perror("Fork failed");
16            exit(EXIT_FAILURE);
17        }
18        else if (left_child_pid == 0)
19        {
20            // Left child process setup for pipe redirection
21            setup_pipe_redirection(parent_child_l,
22                                  child_parent_l);
23
24            char depth[10], max[10], lr[2];
25            format_values(current_depth, max_depth, 0,
26                          depth, max, lr); // Format values for left
27                                           child
28
29            char *args[] = { "./treePipe", depth, max, lr,
30                             NULL };
31            execvp("./treePipe", args);

```

```

27         perror("Exec failed");
28         return 1;
29     }
30
31     close(parent_child_l[0]);
32     close(child_parent_l[1]);
33
34     // Send num1 to left child
35     dprintf(parent_child_l[1], "%d\n", num_root);
36     close(parent_child_l[1]);
37
38     // Get result from left child
39     num_root = read_from_pipe(child_parent_l[0]);
40
41     dash_printer(current_depth);
42     fprintf(stderr, "> My num1 is: %d\n", num_root);

```

Right Child Process

```

1     // Right child process
2     int parent_child_r[2], child_parent_r[2];
3     if (pipe(parent_child_r) == -1 ||
4         pipe(child_parent_r) == -1)
5     {
6         perror("Pipe failed");
7         exit(EXIT_FAILURE);
8     }
9
10    pid_t right_child_pid = fork();
11    if (right_child_pid < 0)
12    {
13        perror("Fork failed");
14        exit(EXIT_FAILURE);
15    }
16    else if (right_child_pid == 0)
17    {
18        // Right child process setup for pipe
19        // redirection
20        setup_pipe_redirection(parent_child_r,
21                                child_parent_r);
22
23        char depth[10], max[10], lr[2];
24        format_values(current_depth, max_depth, 1,
25                        depth, max, lr); // Format values for right
26                                         child
27
28        char *args[] = {"../treePipe", depth, max, lr,
29                        NULL};
30        execvp("./treePipe", args);
31        perror("Exec failed");

```



```

26         return 1;
27     }
28
29     close(parent_child_r[0]);
30     close(child_parent_r[1]);
31
32     // Send num1 to right child
33     dprintf(parent_child_r[1], "%d\n", num_root);
34     close(parent_child_r[1]);
35
36     // Get result from right child
37     num_leaf = read_from_pipe(child_parent_r[0]);
38
39     dash_printer(current_depth);
40     fprintf(stderr, "> Current depth: %d, lr: %d, my
41         num1: %d, my num2: %d\n",
42             current_depth, l_r, num_root, num_leaf);
43
44     // Wait for child processes to finish
45     wait(NULL);
46     wait(NULL);
47 }

```

4 Executing the Computation Program

Each node executes either the addition (left) or multiplication (right) program based on its left-right indicator.

```

1 // Program process (left or right)
2 int program_send[2], program_get[2];
3 if (pipe(program_send) == -1 || pipe(program_get) == -1)
4 {
5     perror("Pipe failed");
6     exit(EXIT_FAILURE);
7 }
8
9 pid_t prog_pid = fork();
10 if (prog_pid == 0)
11 {
12     // Pipe redirection for program process
13     setup_pipe_redirection(program_send, program_get);
14
15     char *prog;
16     if (l_r)
17     {
18         prog = "./right";
19     }
20     else

```

```

21     {
22         prog = "./left";
23     }
24     char *args[] = {prog, NULL};
25     execvp(prog, args);
26     perror("Exec failed");
27     return 1;
28 }
29
30 close(program_send[0]);
31 close(program_get[1]);
32
33 // Send numbers to the computation program
34 dprintf(program_send[1], "%d\n%d\n", num_root,
35         num_leaf);
36 close(program_send[1]);
37
38 // Get result from the computation program
39 int result = read_from_pipe(program_get[0]);
40
41 dash_printer(current_depth);
42 fprintf(stderr, "> My result is: %d\n", result);
43
44 if (current_depth == 0)
45 {
46     printf("The final result is: %d\n", result);
47 }
48 else
49 {
50     printf("%d\n", result);
51 }
52 wait(NULL);
53 return 0;
54 }

```

5 Conclusion

The `treePipe.c` program effectively simulates a binary tree of processes using Unix system calls for process creation and inter-process communication. The implementation demonstrates an understanding of:

- Process control using `fork()` and `execvp()`.
- Inter-process communication using `pipe()` and `dup2()`.
- Recursive process creation and post-order traversal in a process tree.
- Error handling and logging for robust program execution.

By carefully managing pipes and process hierarchy, the program ensures correct data flow and computation across all nodes in the tree.

A Source Code

For completeness, the full source code of `treePipe.c` is included below.

Listing 1: `treePipe.c` Source Code

```
1 // Barış Pome - 31311
2 // CS307 - OS - 24-25 Fall - PA1
3
4 #include <sys/wait.h>
5 #include <string.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <unistd.h>
9
10 // Utility functions
11
12 // The function to print info about node to the console
13 void dash_printer(int number)
14 {
15     if (number > 0)
16     {
17         for (int i = 0; i < number; i++)
18         {
19             fprintf(stderr, "---");
20         }
21     }
22 }
23
24 // The function to check argc returning 1 indicates success
25 // returning 0 indicates failure
26 int argument_checker(int argc)
27 {
28     if (argc != 4)
29     {
30         fprintf(stderr, "Usage: treePipe <current depth>
31             <max depth> <left-right>\n");
32         return 0;
33     }
34     return 1;
35 }
36
37 // The function to get number if the node is root
38 int num_for_root(int current_depth)
39 {
40     int num_root;
41     if (current_depth == 0)
```

```

40     {
41         fprintf(stderr, "Please enter num1 for the root: ");
42         scanf("%d", &num_root);
43     }
44     else
45     {
46         // Read num1 for non-root nodes
47         scanf("%d", &num_root);
48     }
49     return num_root;
50 }
51
52 // The function for pipe redirections
53 void setup_pipe_redirection(int pipe_to_child[2], int
54     pipe_from_child[2])
55 {
56     // Close the unused write end of the pipe to child
57     close(pipe_to_child[1]);
58     // Close the unused read end of the pipe from child
59     close(pipe_from_child[0]);
60     // Redirect standard input to the read end of the pipe
61     // to child
62     dup2(pipe_to_child[0], STDIN_FILENO);
63     // Redirect standard output to the write end of the
64     // pipe from child
65     dup2(pipe_from_child[1], STDOUT_FILENO);
66     // Close the original pipe file descriptors after
67     // redirection
68     close(pipe_to_child[0]);
69     close(pipe_from_child[1]);
70 }
71
72 // Function to read data from a pipe and convert to integer
73 int read_from_pipe(int pipe_fd)
74 {
75     char buffer[10] = {0};           // Buffer to
76     // store the data read from the pipe
77     read(pipe_fd, buffer, sizeof(buffer)); // Read from the
78     // pipe
79     close(pipe_fd);                  // Close the
80     // pipe after reading
81     return atoi(buffer);              // Convert the
82     // string to an integer and return it
83 }
84
85 // Function to format values into strings
86 void format_values(int current_depth, int max_depth, int
87     is_left, char *depth, char *max, char *lr)
88 {
89     sprintf(lr, "%d", is_left);      // Format the

```

```

81     left-right indicator (0 or 1) as a string
    sprintf(depth, "%d", current_depth + 1); // Format
        current_depth + 1 as a string
82     sprintf(max, "%d", max_depth);          // Format
        max_depth as a string
83 }
84
85 int main(int argc, char *argv[])
86 {
87     if (!argument_checker(argc))
88     {
89         return 1;
90     }
91     // get the information of program
92     int l_r = atoi(argv[3]);
93     int max_depth = atoi(argv[2]);
94     int current_depth = atoi(argv[1]);
95
96     dash_printer(current_depth);
97     fprintf(stderr, "> Current depth: %d, lr: %d\n",
        current_depth, l_r);
98
99     // default value that comes from the leaf nodes
100    int num_leaf = 1;
101    int num_root = num_for_root(current_depth);
102
103    if (current_depth < max_depth)
104    {
105        // Left child process
106        int parent_child_l[2];
107        int child_parent_l[2];
108        pipe(parent_child_l);
109        pipe(child_parent_l);
110        if (pipe(parent_child_l) == -1 ||
            pipe(child_parent_l) == -1)
111        {
112            perror("Pipe failed");
113            exit(EXIT_FAILURE);
114        }
115
116        pid_t left_child_pid = fork();
117        if (left_child_pid < 0)
118        {
119            perror("Fork failed");
120            exit(EXIT_FAILURE);
121        }
122        else if (left_child_pid == 0)
123        {
124            // Left child process setup for pipe redirection
125            setup_pipe_redirection(parent_child_l,

```

```

126         child_parent_l);
127
128     char depth[10], max[10], lr[2];
129     format_values(current_depth, max_depth, 0,
130         depth, max, lr); // Format values for left
131                             child
132
133     char *args[] = {"/treePipe", depth, max, lr,
134         NULL};
135     execvp("/treePipe", args);
136     perror("Exec failed");
137     return 1;
138 }
139 close(parent_child_l[0]);
140 close(child_parent_l[1]);
141
142 // Send num1 to left child
143 dprintf(parent_child_l[1], "%d\n", num_root);
144 close(parent_child_l[1]);
145
146 // Get result from left child
147 num_root = read_from_pipe(child_parent_l[0]);
148
149 dash_printer(current_depth);
150 fprintf(stderr, "> My num1 is: %d\n", num_root);
151
152 // Right child process
153 int child_parent_r[2];
154 int parent_child_r[2];
155 pipe(child_parent_r);
156 pipe(parent_child_r);
157 if (pipe(parent_child_r) == -1 ||
158     pipe(child_parent_r) == -1)
159 {
160     perror("Pipe failed");
161     exit(EXIT_FAILURE);
162 }
163
164 pid_t right_child_pid = fork();
165 if (right_child_pid < 0)
166 {
167     perror("Fork failed");
168     exit(EXIT_FAILURE);
169 }
170 else if (right_child_pid == 0)
171 {
172     // Right child process setup for pipe
173     redirection
174     setup_pipe_redirection(child_parent_r,
175         parent_child_r);

```

```

169
170     char lr[2];
171     char depth[10];
172     char max[10];
173     format_values(current_depth, max_depth, 1,
174                   depth, max, lr); // Format values for right
175                                     child
176
177     char *args[] = { "./treePipe", depth, max, lr,
178                     NULL};
179     execvp("./treePipe", args);
180     perror("Exec failed");
181     return 1;
182 }
183
184 close(child_parent_r[0]);
185 close(parent_child_r[1]);
186 dprintf(child_parent_r[1], "%d\n", num_root);
187 close(child_parent_r[1]);
188
189 // Get result from right child
190 num_leaf = read_from_pipe(parent_child_r[0]);
191
192 dash_printer(current_depth);
193 fprintf(stderr, "> Current depth: %d, lr: %d, my
194           num1: %d, my num2: %d\n",
195           current_depth, l_r, num_root, num_leaf);
196
197 wait(NULL);
198 wait(NULL);
199 }
200 else
201 {
202     dash_printer(current_depth);
203     fprintf(stderr, "> My num1 is: %d\n", num_root);
204 }
205
206 // Program process (left or right)
207 int program_send[2];
208 int program_get[2];
209 pipe(program_send);
210 pipe(program_get);
211
212 pid_t program_pid = fork();
213 if (program_pid == 0)
214 {
215     // The pipe redirection for program process
216     setup_pipe_redirection(program_send, program_get);
217
218     char *prog;

```

```

215         if (l_r)
216         {
217             prog = "./right";
218         }
219         else
220         {
221             prog = "./left";
222         }
223         char *args[] = {prog, NULL};
224         execvp(prog, args);
225         perror("Exec failed");
226         return 1;
227     }
228
229     close(program_send[0]);
230     close(program_get[1]);
231
232     dprintf(program_send[1], "%d\n%d\n", num_root,
233             num_leaf);
234     close(program_send[1]);
235
236     // Get result from program process
237     int result = read_from_pipe(program_get[0]);
238
239     dash_printer(current_depth);
240     fprintf(stderr, "> My result is: %d\n", result);
241
242     if (current_depth == 0)
243     {
244         printf("The final result is: %d\n", result);
245     }
246     else
247     {
248         printf("%d\n", result);
249     }
250     wait(NULL);
251     return 0;

```