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

Barış Pome - 31311

Contents

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>
```

```c
#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.

```c
void dash_printer(int number)
{
    if (number > 0)
    {
        for (int i = 0; i < number; i++)
        {
            fprintf(stderr, "---");
        }
    }
}
```

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`.

```c
int argument_checker(int argc)
{
    if (argc != 4)
    {
        fprintf(stderr, "Usage: treePipe <current depth>
            <max depth> <left-right>\n");
        return 0;
    }
    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()`.

```c
int num_for_root(int current_depth)
{
    int num_root;
    if (current_depth == 0)
    {
        fprintf(stderr, "Please enter num1 for the root: ");
        scanf("%d", &num_root);
    }
    else
    {
        // Read num1 for non-root nodes
        scanf("%d", &num_root);
    }
    return num_root;
}
```

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.

```c
void setup_pipe_redirection(int pipe_to_child[2], int
    pipe_from_child[2])
{
    // Close the unused write end of the pipe to child
    close(pipe_to_child[1]);
    // Close the unused read end of the pipe from child
    close(pipe_from_child[0]);

    // Redirect standard input to the read end of the pipe
        to child
    dup2(pipe_to_child[0], STDIN_FILENO);
    // Redirect standard output to the write end of the
        pipe from child
    dup2(pipe_from_child[1], STDOUT_FILENO);

    // Close the original pipe file descriptors after
        redirection
    close(pipe_to_child[0]);
    close(pipe_from_child[1]);
}
```

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.

```c
int read_from_pipe(int pipe_fd)
{
    char buffer[10] = {0};                  // Buffer to store the data read from the pipe
    read(pipe_fd, buffer, sizeof(buffer)); // Read from the pipe
    close(pipe_fd);                         // Close the pipe after reading
    return atoi(buffer);                    // Convert the string to an integer and return it
}
```

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.

```c
void format_values(int current_depth, int max_depth, int is_left, char *depth, char *max, char *lr)
{
    sprintf(depth, "%d", current_depth + 1); // Format current_depth + 1 as a string
    sprintf(max, "%d", max_depth);           // Format max_depth as a string
    sprintf(lr, "%d", is_left);              // Format the left-right indicator (0 or 1) as a string
}
```

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

```c
int main(int argc, char *argv[])
{
    if (!argument_checker(argc))
    {
        return 1;
    }

    // Retrieve program arguments
```

```
    int l_r = atoi(argv[3]);
    int max_depth = atoi(argv[2]);
    int current_depth = atoi(argv[1]);

    dash_printer(current_depth);
    fprintf(stderr, "> Current depth: %d, lr: %d\n",
        current_depth, l_r);

    // Initialize variables
    int num_leaf = 1;
    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

```
    if (current_depth < max_depth)
    {
        // Left child process
        int parent_child_l[2], child_parent_l[2];
        if (pipe(parent_child_l) == -1 ||
            pipe(child_parent_l) == -1)
        {
            perror("Pipe failed");
            exit(EXIT_FAILURE);
        }

        pid_t left_child_pid = fork();
        if (left_child_pid < 0)
        {
            perror("Fork failed");
            exit(EXIT_FAILURE);
        }
        else if (left_child_pid == 0)
        {
            // Left child process setup for pipe redirection
            setup_pipe_redirection(parent_child_l,
                child_parent_l);

            char depth[10], max[10], lr[2];
            format_values(current_depth, max_depth, 0,
                depth, max, lr); // Format values for left
                child

            char *args[] = {"./treePipe", depth, max, lr,
                NULL};
            execvp("./treePipe", args);
```

```
            perror("Exec failed");
            return 1;
        }

        close(parent_child_l[0]);
        close(child_parent_l[1]);

        // Send num1 to left child
        dprintf(parent_child_l[1], "%d\n", num_root);
        close(parent_child_l[1]);

        // Get result from left child
        num_root = read_from_pipe(child_parent_l[0]);

        dash_printer(current_depth);
        fprintf(stderr, "> My num1 is: %d\n", num_root);
```

Right Child Process

```
        // Right child process
        int parent_child_r[2], child_parent_r[2];
        if (pipe(parent_child_r) == -1 ||
            pipe(child_parent_r) == -1)
        {
            perror("Pipe failed");
            exit(EXIT_FAILURE);
        }

        pid_t right_child_pid = fork();
        if (right_child_pid < 0)
        {
            perror("Fork failed");
            exit(EXIT_FAILURE);
        }
        else if (right_child_pid == 0)
        {
            // Right child process setup for pipe
                redirection
            setup_pipe_redirection(parent_child_r,
                child_parent_r);

            char depth[10], max[10], lr[2];
            format_values(current_depth, max_depth, 1,
                depth, max, lr); // Format values for right
                child

            char *args[] = {"./treePipe", depth, max, lr,
                NULL};
            execvp("./treePipe", args);
            perror("Exec failed");
```

```
            return 1;
        }

        close(parent_child_r[0]);
        close(child_parent_r[1]);

        // Send num1 to right child
        dprintf(parent_child_r[1], "%d\n", num_root);
        close(parent_child_r[1]);

        // Get result from right child
        num_leaf = read_from_pipe(child_parent_r[0]);

        dash_printer(current_depth);
        fprintf(stderr, "> Current depth: %d, lr: %d, my
            num1: %d, my num2: %d\n",
                current_depth, l_r, num_root, num_leaf);

        // Wait for child processes to finish
        wait(NULL);
        wait(NULL);
    }
```

4 Executing the Computation Program

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

```
    // Program process (left or right)
    int program_send[2], program_get[2];
    if (pipe(program_send) == -1 || pipe(program_get) == -1)
    {
        perror("Pipe failed");
        exit(EXIT_FAILURE);
    }

    pid_t prog_pid = fork();
    if (prog_pid == 0)
    {
        // Pipe redirection for program process
        setup_pipe_redirection(program_send, program_get);

        char *prog;
        if (l_r)
        {
            prog = "./right";
        }
        else
```

```
        {
            prog = "./left";
        }
        char *args[] = {prog, NULL};
        execvp(prog, args);
        perror("Exec failed");
        return 1;
    }

    close(program_send[0]);
    close(program_get[1]);

    // Send numbers to the computation program
    dprintf(program_send[1], "%d\n%d\n", num_root,
        num_leaf);
    close(program_send[1]);

    // Get result from the computation program
    int result = read_from_pipe(program_get[0]);

    dash_printer(current_depth);
    fprintf(stderr, "> My result is: %d\n", result);

    if (current_depth == 0)
    {
        printf("The final result is: %d\n", result);
    }
    else
    {
        printf("%d\n", result);
    }
    wait(NULL);
    return 0;
}
```

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

```c
// Barış Pome - 31311
// CS307 - OS - 24-25 Fall - PA1

#include <sys/wait.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// Utility functions

// The function to print info about node to the console
void dash_printer(int number)
{
    if (number > 0)
    {
        for (int i = 0; i < number; i++)
        {
            fprintf(stderr, "---");
        }
    }
}

// The function to check argc returning 1 indicates success
    returning 0 indicates failure
int argument_checker(int argc)
{
    if (argc != 4)
    {
        fprintf(stderr, "Usage: treePipe <current depth>
            <max depth> <left-right>\n");
        return 0;
    }
    return 1;
}

// The function to get number if the node is root
int num_for_root(int current_depth)
{
    int num_root;
    if (current_depth == 0)
```

```
    {
        fprintf(stderr, "Please enter num1 for the root: ");
        scanf("%d", &num_root);
    }
    else
    {
        // Read num1 for non-root nodes
        scanf("%d", &num_root);
    }
    return num_root;
}

// The function for pipe redirections
void setup_pipe_redirection(int pipe_to_child[2], int
    pipe_from_child[2])
{
    // Close the unused write end of the pipe to child
    close(pipe_to_child[1]);
    // Close the unused read end of the pipe from child
    close(pipe_from_child[0]);
    // Redirect standard input to the read end of the pipe
        to child
    dup2(pipe_to_child[0], STDIN_FILENO);
    // Redirect standard output to the write end of the
        pipe from child
    dup2(pipe_from_child[1], STDOUT_FILENO);
    // Close the original pipe file descriptors after
        redirection
    close(pipe_to_child[0]);
    close(pipe_from_child[1]);
}

// Function to read data from a pipe and convert to integer
int read_from_pipe(int pipe_fd)
{
    char buffer[10] = {0};                  // Buffer to
        store the data read from the pipe
    read(pipe_fd, buffer, sizeof(buffer)); // Read from the
        pipe
    close(pipe_fd);                         // Close the
        pipe after reading
    return atoi(buffer);                    // Convert the
        string to an integer and return it
}

// Function to format values into strings
void format_values(int current_depth, int max_depth, int
    is_left, char *depth, char *max, char *lr)
{
    sprintf(lr, "%d", is_left);              // Format the
```

```
        left-right indicator (0 or 1) as a string
    sprintf(depth, "%d", current_depth + 1); // Format
        current_depth + 1 as a string
    sprintf(max, "%d", max_depth);           // Format
        max_depth as a string
}

int main(int argc, char *argv[])
{
    if (!argument_checker(argc))
    {
        return 1;
    }
    // get the information of program
    int l_r = atoi(argv[3]);
    int max_depth = atoi(argv[2]);
    int current_depth = atoi(argv[1]);

    dash_printer(current_depth);
    fprintf(stderr, "> Current depth: %d, lr: %d\n",
        current_depth, l_r);

    // default value that comes from the leaf nodes
    int num_leaf = 1;
    int num_root = num_for_root(current_depth);

    if (current_depth < max_depth)
    {
        // Left child process
        int parent_child_l[2];
        int child_parent_l[2];
        pipe(parent_child_l);
        pipe(child_parent_l);
        if (pipe(parent_child_l) == -1 ||
            pipe(child_parent_l) == -1)
        {
            perror("Pipe failed");
            exit(EXIT_FAILURE);
        }

        pid_t left_child_pid = fork();
        if (left_child_pid < 0)
        {
            perror("Fork failed");
            exit(EXIT_FAILURE);
        }
        else if (left_child_pid == 0)
        {
            // Left child process setup for pipe redirection
            setup_pipe_redirection(parent_child_l,
```

```
            child_parent_l);

        char depth[10], max[10], lr[2];
        format_values(current_depth, max_depth, 0,
            depth, max, lr); // Format values for left
            child

        char *args[] = {"./treePipe", depth, max, lr,
            NULL};
        execvp("./treePipe", args);
        perror("Exec failed");
        return 1;
    }
    close(parent_child_l[0]);
    close(child_parent_l[1]);

    // Send num1 to left child
    dprintf(parent_child_l[1], "%d\n", num_root);
    close(parent_child_l[1]);

    // Get result from left child
    num_root = read_from_pipe(child_parent_l[0]);

    dash_printer(current_depth);
    fprintf(stderr, "> My num1 is: %d\n", num_root);

    // Right child process
    int child_parent_r[2];
    int parent_child_r[2];
    pipe(child_parent_r);
    pipe(parent_child_r);
    if (pipe(parent_child_r) == -1 ||
        pipe(child_parent_r) == -1)
    {
        perror("Pipe failed");
        exit(EXIT_FAILURE);
    }

    pid_t right_child_pid = fork();
    if (right_child_pid < 0)
    {
        perror("Fork failed");
        exit(EXIT_FAILURE);
    }
    else if (right_child_pid == 0)
    {
        // Right child process setup for pipe
            redirection
        setup_pipe_redirection(child_parent_r,
            parent_child_r);
```

```
169
170             char lr[2];
171             char depth[10];
172             char max[10];
173             format_values(current_depth, max_depth, 1,
                    depth, max, lr); // Format values for right
                    child
174
175             char *args[] = {"./treePipe", depth, max, lr,
                    NULL};
176             execvp("./treePipe", args);
177             perror("Exec failed");
178             return 1;
179         }
180
181         close(child_parent_r[0]);
182         close(parent_child_r[1]);
183         dprintf(child_parent_r[1], "%d\n", num_root);
184         close(child_parent_r[1]);
185
186         // Get result from right child
187         num_leaf = read_from_pipe(parent_child_r[0]);
188
189         dash_printer(current_depth);
190         fprintf(stderr, "> Current depth: %d, lr: %d, my
                num1: %d, my num2: %d\n",
191                 current_depth, l_r, num_root, num_leaf);
192
193         wait(NULL);
194         wait(NULL);
195     }
196     else
197     {
198         dash_printer(current_depth);
199         fprintf(stderr, "> My num1 is: %d\n", num_root);
200     }
201
202     // Program process (left or right)
203     int program_send[2];
204     int program_get[2];
205     pipe(program_send);
206     pipe(program_get);
207
208     pid_t program_pid = fork();
209     if (program_pid == 0)
210     {
211         // The pipe redirection for program process
212         setup_pipe_redirection(program_send, program_get);
213
214         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,
        num_leaf);
233     close(program_send[1]);
234 
235     // Get result from program process
236     int result = read_from_pipe(program_get[0]);
237 
238     dash_printer(current_depth);
239     fprintf(stderr, "> My result is: %d\n", result);
240 
241     if (current_depth == 0)
242     {
243         printf("The final result is: %d\n", result);
244     }
245     else
246     {
247         printf("%d\n", result);
248     }
249     wait(NULL);
250     return 0;
251 }
```