COURSE:	CS307
NAME:	UTKU
SURNAME:	GENC
SUNAME:	utku.genc
SUID:	30611
ASSIGNMENT:	PA1

CONTENTS

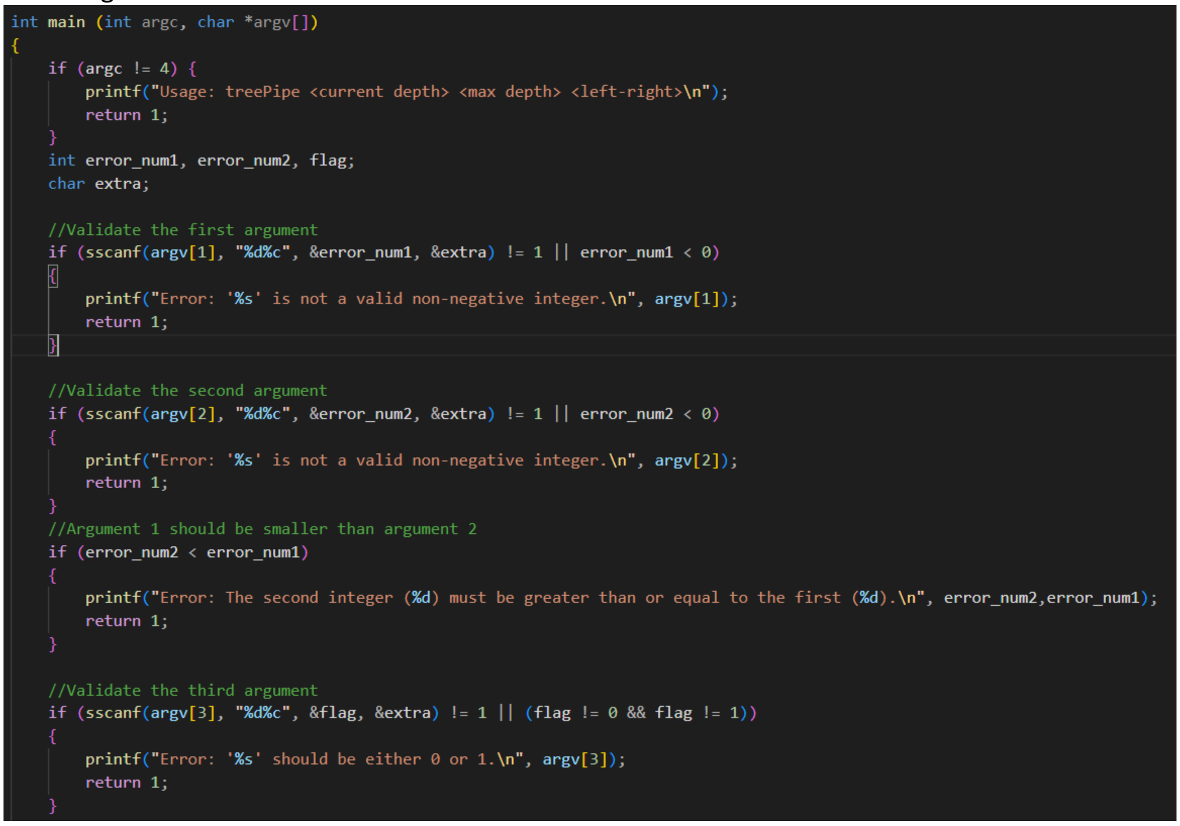
OUR GOAL IN THIS ASSIGNMENT

Our goal in this assignment is to gain knowledge about how and why Linux Pipes are used and to implement the command called ‘treePipe’ using the C programming language.

PROGRAM DESCRIPTION

The rules we need to follow when implementing this program are written in the CS_307_PA1_FL24.pdf document. Therefore, I will not repeat these rules here. I will show how the program works with the treePipe implementation code I wrote in stages.

First of all, our program checks whether the arguments it receives are complete and whether these arguments are correct.



```c
int main (int argc, char *argv[])
{
    if (argc != 4) {
        printf("Usage: treePipe <current depth> <max depth> <left-right>\n");
        return 1;
    }
    int error_num1, error_num2, flag;
    char extra;

    //Validate the first argument
    if (sscanf(argv[1], "%d%c", &error_num1, &extra) != 1 || error_num1 < 0)
    {
        printf("Error: '%s' is not a valid non-negative integer.\n", argv[1]);
        return 1;
    }

    //Validate the second argument
    if (sscanf(argv[2], "%d%c", &error_num2, &extra) != 1 || error_num2 < 0)
    {
        printf("Error: '%s' is not a valid non-negative integer.\n", argv[2]);
        return 1;
    }
    //Argument 1 should be smaller than argument 2
    if (error_num2 < error_num1)
    {
        printf("Error: The second integer (%d) must be greater than or equal to the first (%d).\n", error_num2,error_num1);
        return 1;
    }

    //Validate the third argument
    if (sscanf(argv[3], "%d%c", &flag, &extra) != 1 || (flag != 0 && flag != 1))
    {
        printf("Error: '%s' should be either 0 or 1.\n", argv[3]);
        return 1;
    }
```

The program should take a total of 4 arguments. The first of these is the executable name of our program, './treePipe', the second is the current depth, the third is the maximum depth and the fourth is our 0 or 1 input that determines the right or left program. Here are the rules:

1- Current depth and max depth should not be negative

2- Current depth should be less than max depth

3- The third argument specifying the right or left program should be 1 or 0.

Here, 1 corresponds to the right program and 0 corresponds to the left program. In this example, the right program multiplies and the left program adds.

When we continue with the implementation, we see the required numbers. These are num1, num2 and default value. Since the default value is given to us as 1, we set it equal to 1.

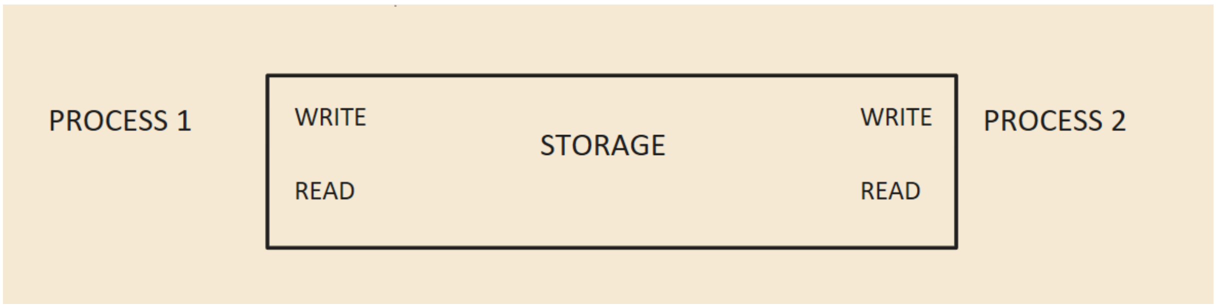
Afterwards, we need to convert the arguments we receive to integers. Because arguments are received as strings (Or something like that, I don't know the C language very well, but they are not integers and we need integers. The 'atoi' function does this conversion.)

Each process first reflects its current depth, maximum depth and the right or left number to the screen. Here, this reflection is done using standard error, because the standard inputs and outputs of the current process will have changed if the current process is not the root. Therefore, we use standard error to see these texts on the screen. I will touch on this topic again in a moment.

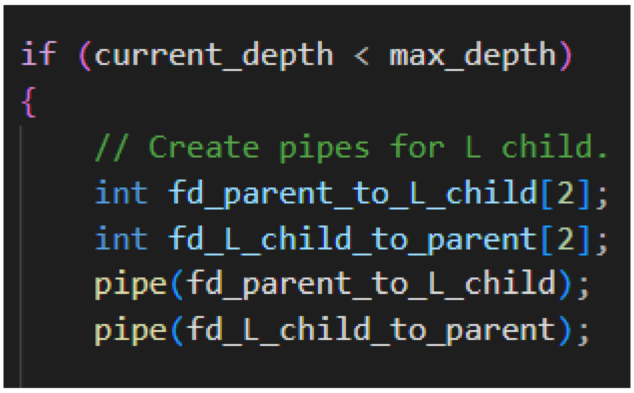
```c
int num1;
int num2 = 1;


int default_value = 1;
int current_depth = atoi(argv[1]);
int max_depth = atoi(argv[2]);
int calculation_program = atoi(argv[3]);

for (int i = 0; i < current_depth; i++)
{
    fprintf(stderr, "---");
}
fprintf(stderr, "> Current depth: %d, lr: %d\n", current_depth, calculation_program);


if (current_depth == 0)
{
    printf("Please enter num1 for the root: ");
    scanf("%d", &num1);
}
else
{
    // Input will be scan from the parent.
    scanf("%d", &num1);
}
```

If we are in the root process, this means our current depth is 0. Since root process is our first process, we should provide num1 to it.

If current process is not root, we should get out input from the PARENT, so we should use pipes.



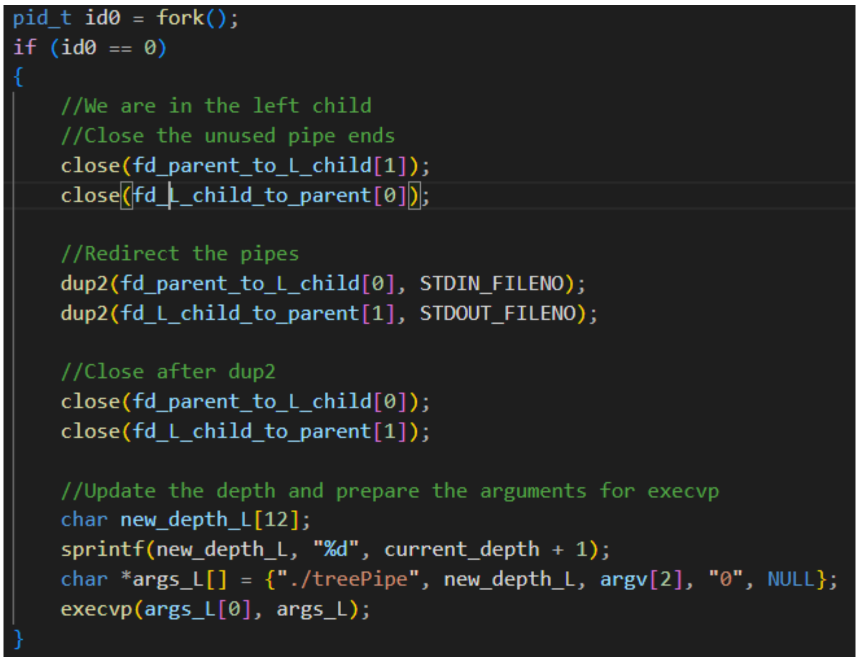
What we call a pipe is the structure I drew above. Here we see two processes. These processes can send data to each other via a pipe. Both processes have write and read sections. The write sections allow writing data to the storage section, and the read section allows reading this data by pulling it from storage. In this way, we have succeeded in sending data between processes.



```
if (current_depth < max_depth)
{
    // Create pipes for L child.
    int fd_parent_to_L_child[2];
    int fd_L_child_to_parent[2];
    pipe(fd_parent_to_L_child);
    pipe(fd_L_child_to_parent);
```

Here we must first test, is our current depth smaller than the maximum depth? If not, this means we are in a leaf process. But for now, let's assume it is small and continue on our way.

In our program the left child always comes to life first. The left child actually means the process will run left program at the end of its life. We create two pipes in order to achieve communicate between child and PARENT. There are two pipes and the reason is simple: we want to send and receive data concurently.



```
pid_t id0 = fork();
if (id0 == 0)
{
    //We are in the left child
    //Close the unused pipe ends
    close(fd_parent_to_L_child[1]);
    close(fd_L_child_to_parent[0]);

    //Redirect the pipes
    dup2(fd_parent_to_L_child[0], STDIN_FILENO);
    dup2(fd_L_child_to_parent[1], STDOUT_FILENO);

    //Close after dup2
    close(fd_parent_to_L_child[0]);
    close(fd_L_child_to_parent[1]);

    //Update the depth and prepare the arguments for execvp
    char new_depth_L[12];
    sprintf(new_depth_L, "%d", current_depth + 1);
    char *args_L[] = {"./treePipe", new_depth_L, argv[2], "0", NULL};
    execvp(args_L[0], args_L);
}
```

After creation of pipe, child must born. And the god is fork() comment. Before continue, i should add: Every process have file descriptor. We can visualize like this:

0	STDIN_FILENO
1	STDOUT_FILENO
2	STDERR_FILENO

When we create pipes:

0	STDIN_FILENO
1	STDOUT_FILENO
2	STDERR_FILENO

3	Fd_parent_to_L_child[0]
4	Fd_parent_to_L_child[1]
5	Fd_L_child_to_parent[0]
6	Fd_L_child_to_parent[1]

In here 0 means read, 1 means write. When we use fork() this table copies into the child process.

By the way, we use fork return value to understand where we are. If it returns 0, this means we are in child. Else we are in parent. We close the unused pipe ends in the child process. And than we use dup2 to change standart input and output.

Dup2() basically do this:

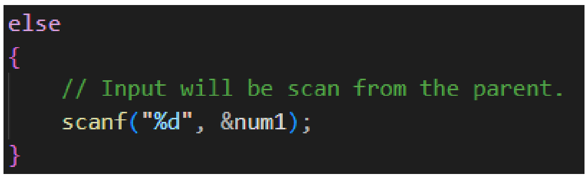
0	Fd_parent_to_L_child[0]
1	Fd_L_child_to_parent[1]
2	STDERR_FILENO
3	Fd_parent_to_L_child[0]
4	Fd_parent_to_L_child[1]
5	Fd_L_child_to_parent[0]
6	Fd_L_child_to_parent[1]

When we do that, standart input and output works with the pipe, not the terminal. The reason we use this actually execvp command. When we use execvp, everything except this table is forgotten and the data of the new program comes. Among these forgotten items is the variable that we normally use to access the pipe. In order to be able to use the pipe without knowing this variable, we redirect the standard output and input to the pipe.

After performing the Dup2 operation, we close the unused pipe ends. Now the child process will access the pipe with standard commands.

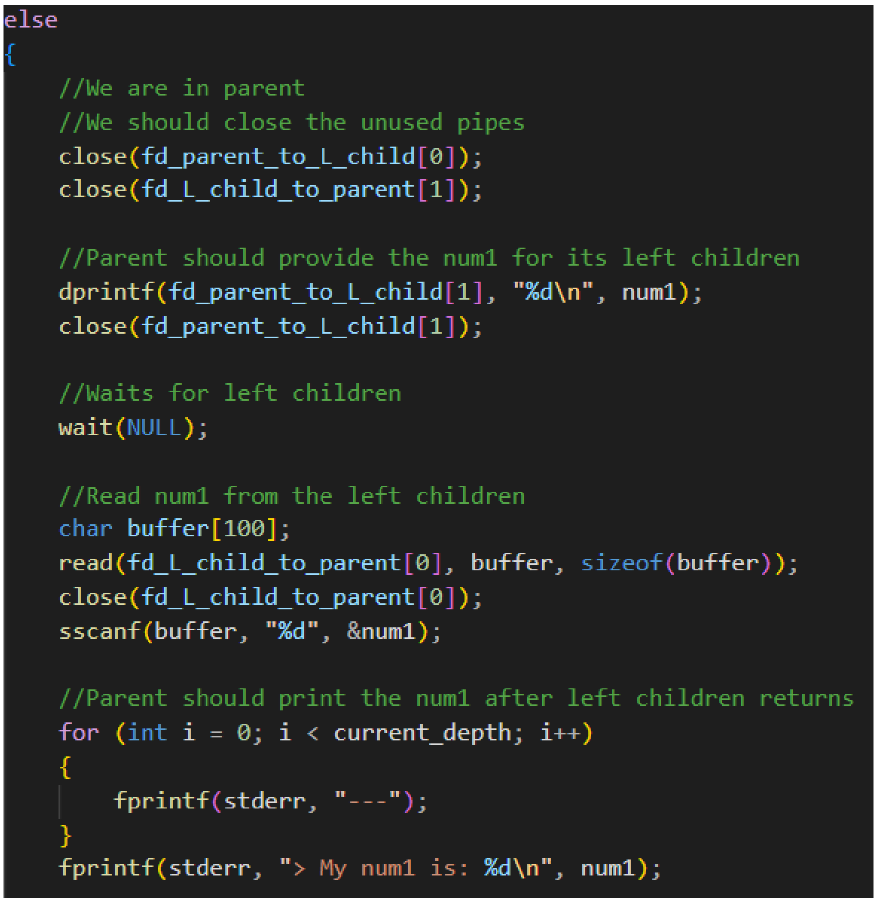
Then we update our depth and prepare our arguments for the execvp command. The ‘execvp’ command causes the child process to become a different process. Here, this program is itself again,

If id0 does not return 0, this shows that we are in the PARENT process. PARENT also closes the pipe parts that it does not use in the same way. And then, using the pipe it created, it sends its own num1 value to the child process. The child process that executes the execvp will also receive this num1 with its redirected standard input. I am sharing the first part of the treePipe program again as a reminder:

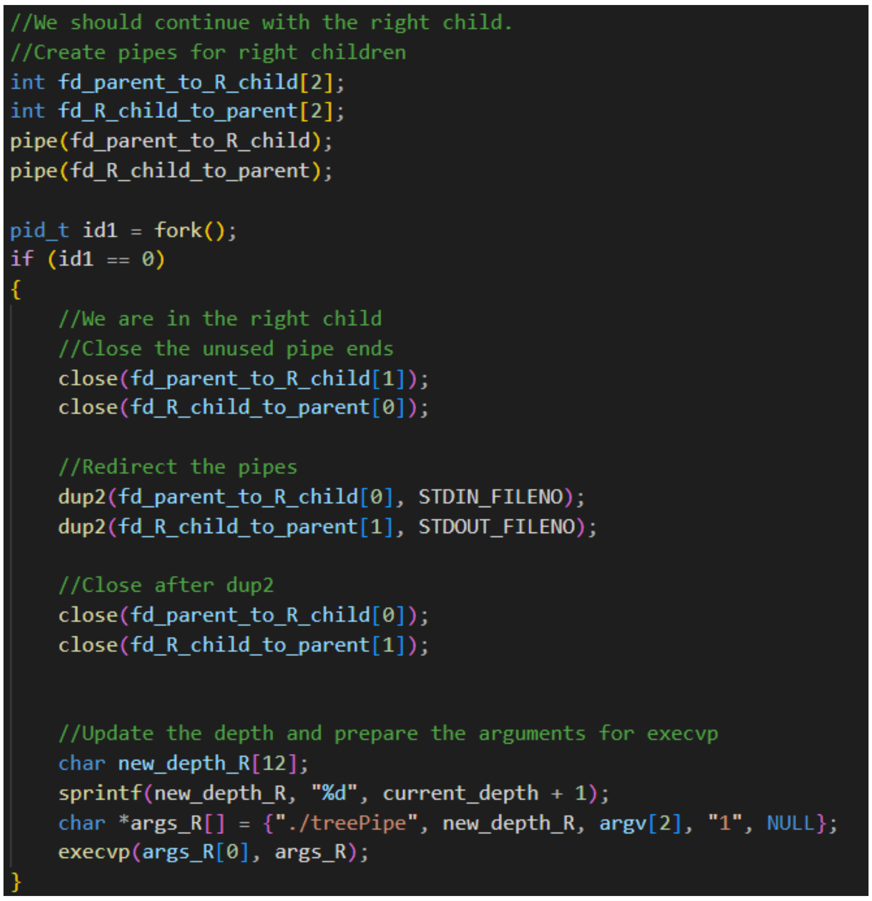


```
else
{
    // Input will be scan from the parent.
    scanf("%d", &num1);
}
```

After sending num1, the parent must wait for the child to calculate. That's why we use wait. When the child process finishes, it will pass its result back to the pipe. Thus, the PARENT process can get its new num1 from the pipe. After receiving it, of course, it should inform us using the standard error.

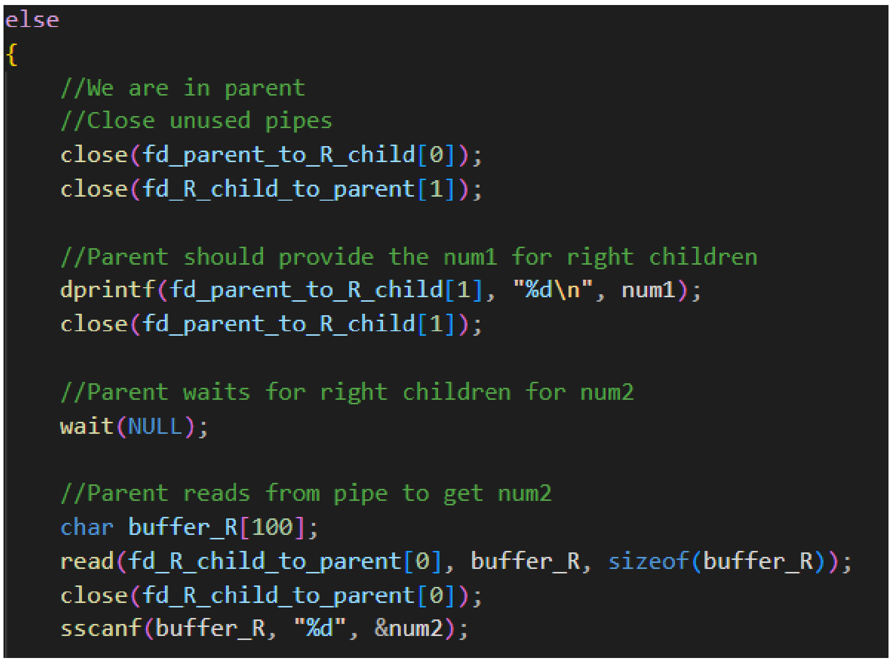


```c
else
{
    //We are in parent
    //We should close the unused pipes
    close(fd_parent_to_L_child[0]);
    close(fd_L_child_to_parent[1]);

    //Parent should provide the num1 for its left children
    dprintf(fd_parent_to_L_child[1], "%d\n", num1);
    close(fd_parent_to_L_child[1]);

    //Waits for left children
    wait(NULL);

    //Read num1 from the left children
    char buffer[100];
    read(fd_L_child_to_parent[0], buffer, sizeof(buffer));
    close(fd_L_child_to_parent[0]);
    sscanf(buffer, "%d", &num1);

    //Parent should print the num1 after left children returns
    for (int i = 0; i < current_depth; i++)
    {
        fprintf(stderr, "---");
    }
    fprintf(stderr, "> My num1 is: %d\n", num1);
```



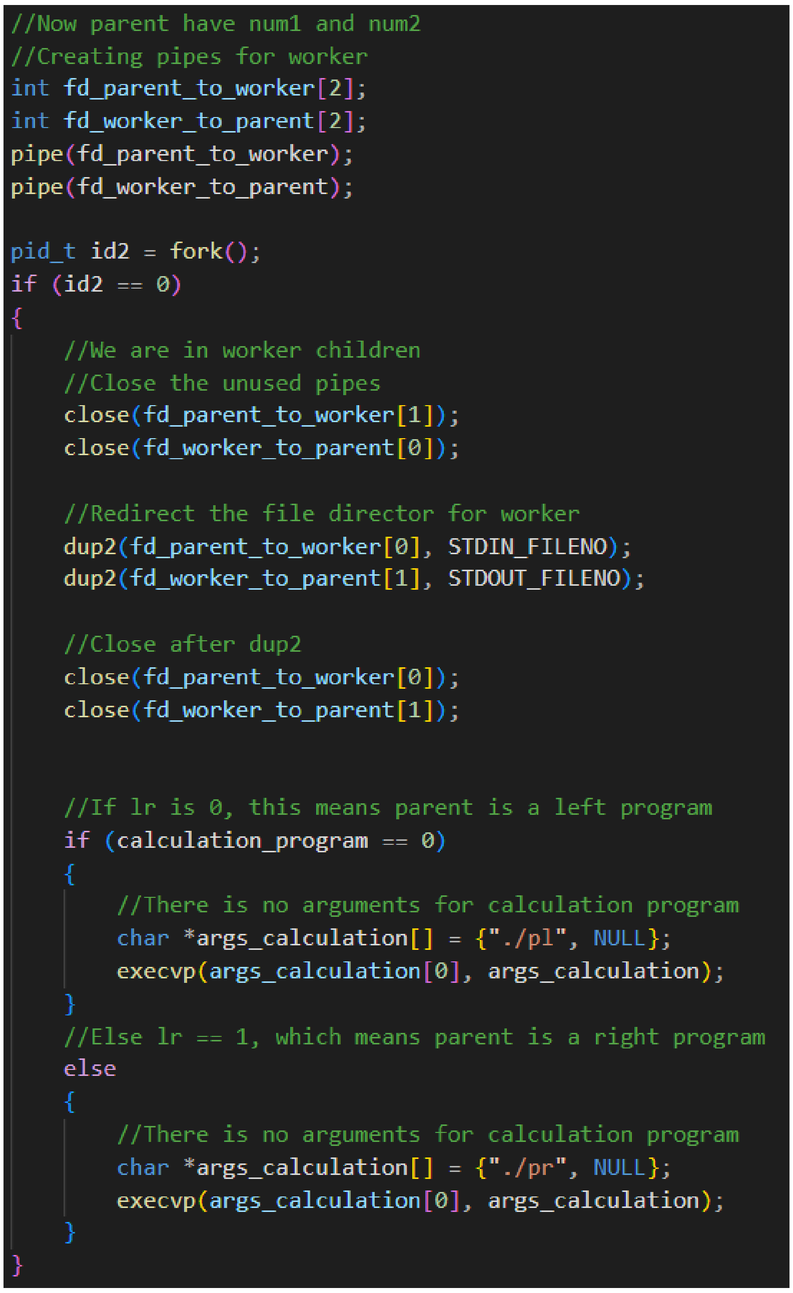
```c
//We should continue with the right child.
//Create pipes for right children
int fd_parent_to_R_child[2];
int fd_R_child_to_parent[2];
pipe(fd_parent_to_R_child);
pipe(fd_R_child_to_parent);

pid_t id1 = fork();
if (id1 == 0)
{
    //We are in the right child
    //Close the unused pipe ends
    close(fd_parent_to_R_child[1]);
    close(fd_R_child_to_parent[0]);

    //Redirect the pipes
    dup2(fd_parent_to_R_child[0], STDIN_FILENO);
    dup2(fd_R_child_to_parent[1], STDOUT_FILENO);

    //Close after dup2
    close(fd_parent_to_R_child[0]);
    close(fd_R_child_to_parent[1]);


    //Update the depth and prepare the arguments for execvp
    char new_depth_R[12];
    sprintf(new_depth_R, "%d", current_depth + 1);
    char *args_R[] = {"./treePipe", new_depth_R, argv[2], "1", NULL};
    execvp(args_R[0], args_R);
}
```

The PARENT process should implement the right child after the left child. Therefore, it should create a different pipe to communicate with the new child. Then, we create this right child with fork(). The part here is the same as the left child. We close the unused pipes, direct the standard input and outputs to the pipe, prepare the arguments and run the execvp command.



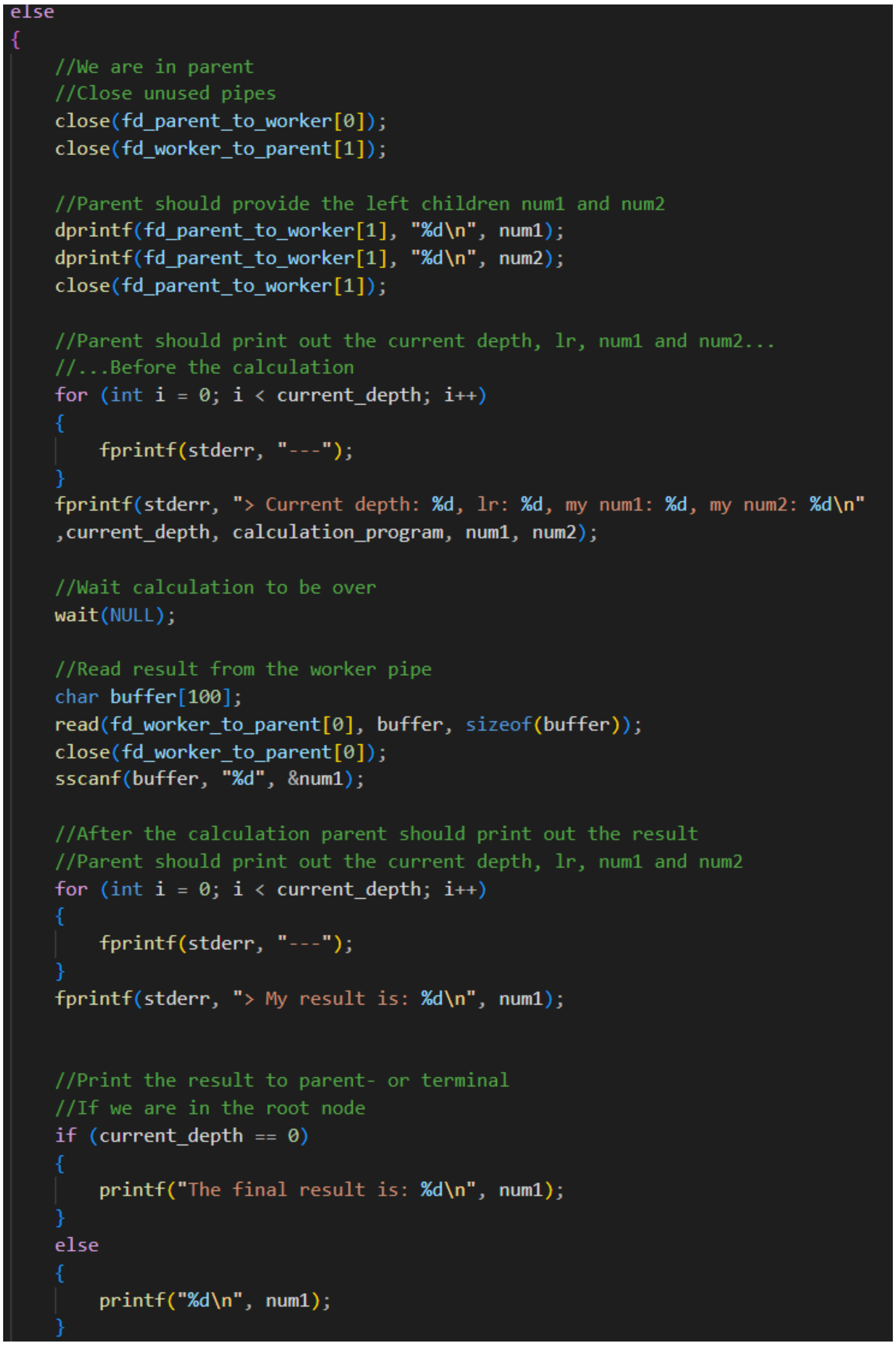
```
else
{
    //We are in parent
    //Close unused pipes
    close(fd_parent_to_R_child[0]);
    close(fd_R_child_to_parent[1]);

    //Parent should provide the num1 for right children
    dprintf(fd_parent_to_R_child[1], "%d\n", num1);
    close(fd_parent_to_R_child[1]);

    //Parent waits for right children for num2
    wait(NULL);

    //Parent reads from pipe to get num2
    char buffer_R[100];
    read(fd_R_child_to_parent[0], buffer_R, sizeof(buffer_R));
    close(fd_R_child_to_parent[0]);
    sscanf(buffer_R, "%d", &num2);
```

Now we are in the PARENT process again. After giving the value of num1 to the right child via the pipe, we wait for the right child to complete its operations. Then we get the result of the right child as num2. Right now, the PARENT process has the value of num1 and num2. It is ready to run the program it was born to run.



```c
//Now parent have num1 and num2
//Creating pipes for worker
int fd_parent_to_worker[2];
int fd_worker_to_parent[2];
pipe(fd_parent_to_worker);
pipe(fd_worker_to_parent);

pid_t id2 = fork();
if (id2 == 0)
{
    //We are in worker children
    //Close the unused pipes
    close(fd_parent_to_worker[1]);
    close(fd_worker_to_parent[0]);

    //Redirect the file director for worker
    dup2(fd_parent_to_worker[0], STDIN_FILENO);
    dup2(fd_worker_to_parent[1], STDOUT_FILENO);

    //Close after dup2
    close(fd_parent_to_worker[0]);
    close(fd_worker_to_parent[1]);


    //If lr is 0, this means parent is a left program
    if (calculation_program == 0)
    {
        //There is no arguments for calculation program
        char *args_calculation[] = {"./pl", NULL};
        execvp(args_calculation[0], args_calculation);
    }
    //Else lr == 1, which means parent is a right program
    else
    {
        //There is no arguments for calculation program
        char *args_calculation[] = {"./pr", NULL};
        execvp(args_calculation[0], args_calculation);
    }
}
```

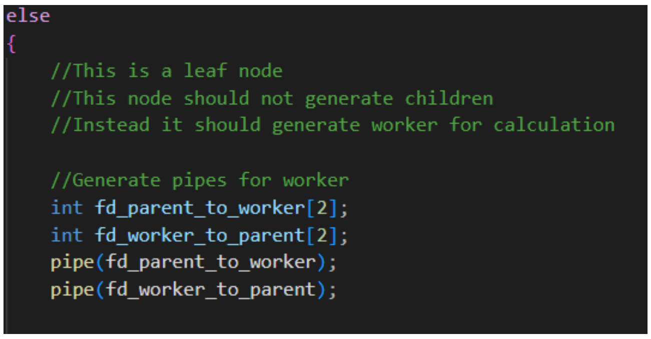
We also create a pipe to communicate with the worker program that it will run. Then the fork() command is run again. After ensuring that the worker can access the pipe with standard input and output with the dup2 command, we convert the child process to the worker program with the execvp command. Here, if the input we received at the beginning is 1, we run the right program, and if it is zero, we run the left program.



```c
else
{
    //We are in parent
    //Close unused pipes
    close(fd_parent_to_worker[0]);
    close(fd_worker_to_parent[1]);

    //Parent should provide the left children num1 and num2
    dprintf(fd_parent_to_worker[1], "%d\n", num1);
    dprintf(fd_parent_to_worker[1], "%d\n", num2);
    close(fd_parent_to_worker[1]);

    //Parent should print out the current depth, lr, num1 and num2...
    //...Before the calculation
    for (int i = 0; i < current_depth; i++)
    {
        fprintf(stderr, "---");
    }
    fprintf(stderr, "> Current depth: %d, lr: %d, my num1: %d, my num2: %d\n"
    ,current_depth, calculation_program, num1, num2);

    //Wait calculation to be over
    wait(NULL);

    //Read result from the worker pipe
    char buffer[100];
    read(fd_worker_to_parent[0], buffer, sizeof(buffer));
    close(fd_worker_to_parent[0]);
    sscanf(buffer, "%d", &num1);

    //After the calculation parent should print out the result
    //Parent should print out the current depth, lr, num1 and num2
    for (int i = 0; i < current_depth; i++)
    {
        fprintf(stderr, "---");
    }
    fprintf(stderr, "> My result is: %d\n", num1);


    //Print the result to parent- or terminal
    //If we are in the root node
    if (current_depth == 0)
    {
        printf("The final result is: %d\n", num1);
    }
    else
    {
        printf("%d\n", num1);
    }
```

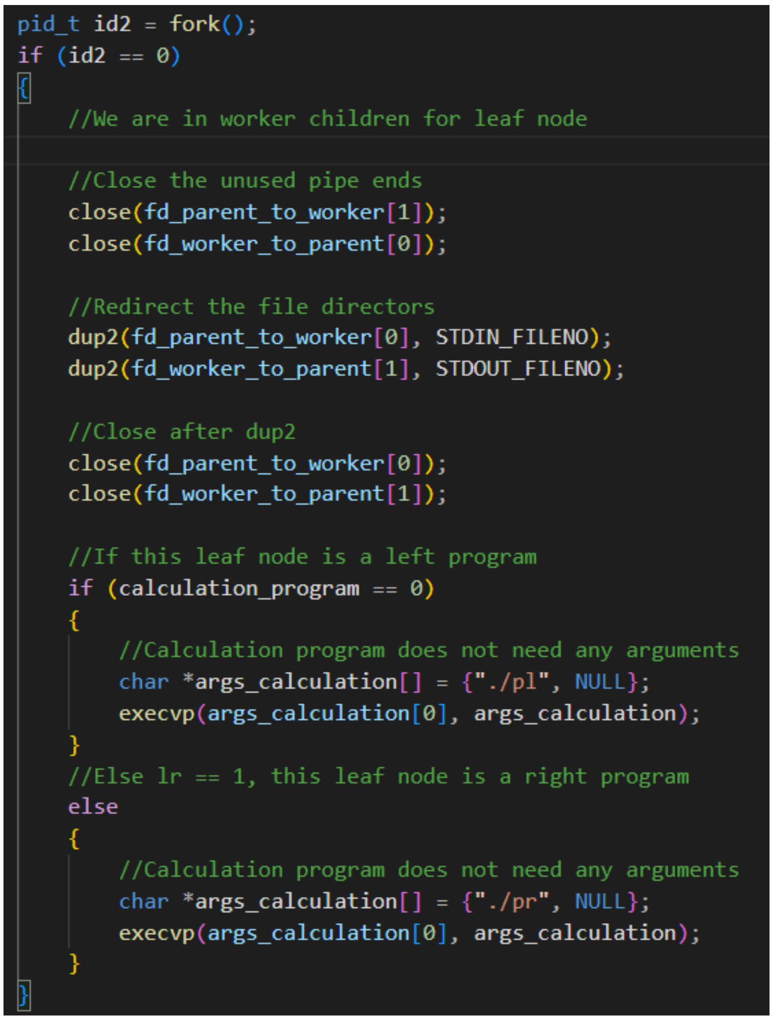
We are in the parent program again. We provide num1 and num2 to the worker program we created. Then we wait for the result of the program. After the result of the program comes, we write this value to num1. Then the PARENT process informs us using the standard error. Finally, if the PARENT process == root, it sends the result to the console, otherwise it sends it to its own PARENT. As you can guess, if it is not root, its standard output and inputs have already changed. Therefore, when it uses standard output, it actually sends the num1 value it has to the pipe created by its own PARENT.

Here we come to the last part. After all, there is a limit to PARENT processes. If a process is at maximum depth, this process is called a leaf. This means that the process will not have any children. Thus, we come to the end of recursive processes.

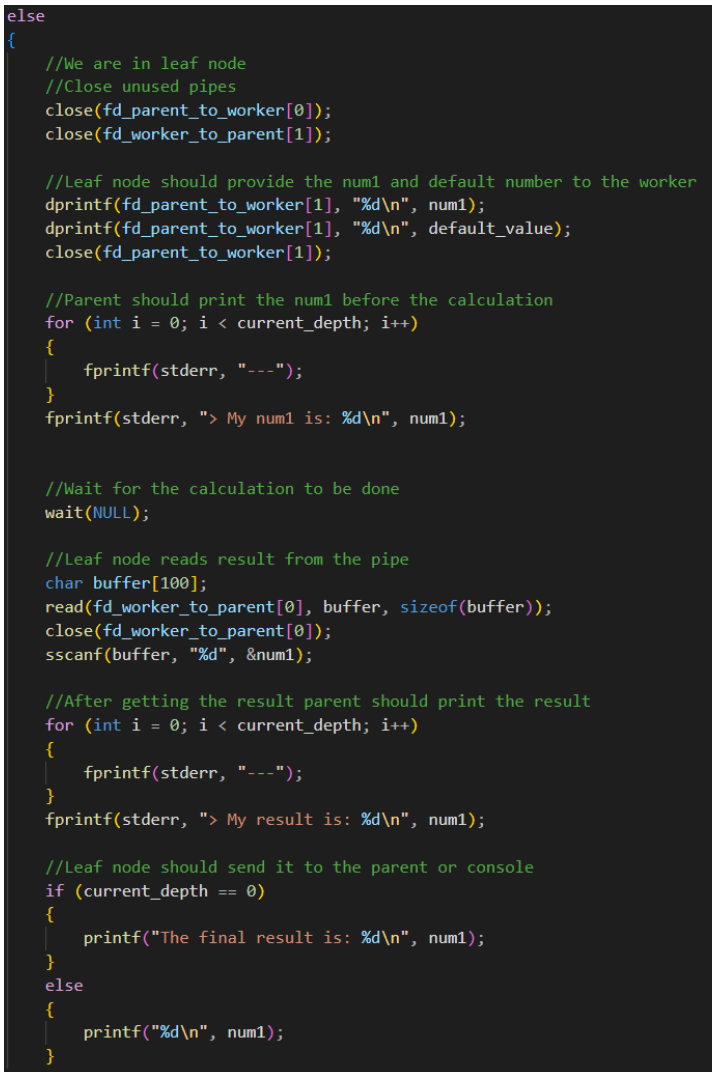


```
else
{
    //This is a leaf node
    //This node should not generate children
    //Instead it should generate worker for calculation

    //Generate pipes for worker
    int fd_parent_to_worker[2];
    int fd_worker_to_parent[2];
    pipe(fd_parent_to_worker);
    pipe(fd_worker_to_parent);
```

The leaf process first establishes a pipe to communicate with the worker it will run. Then, as I mentioned above, if we are in a child process, we close unused pipes and direct the standard input and output to the pipe with dup2. Finally, depending on the argument of the leaf process, the child process transforms itself into a left or right program.



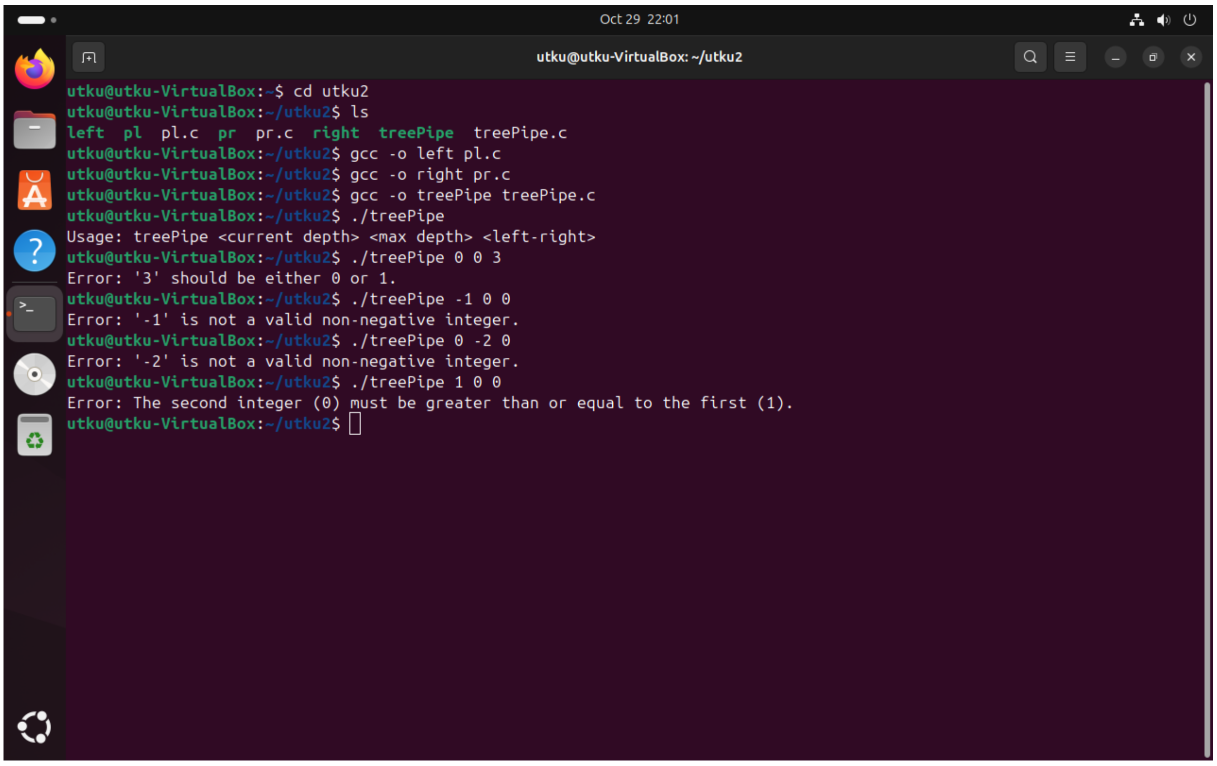
```
pid_t id2 = fork();
if (id2 == 0)
{
    //We are in worker children for leaf node

    //Close the unused pipe ends
    close(fd_parent_to_worker[1]);
    close(fd_worker_to_parent[0]);

    //Redirect the file directors
    dup2(fd_parent_to_worker[0], STDIN_FILENO);
    dup2(fd_worker_to_parent[1], STDOUT_FILENO);

    //Close after dup2
    close(fd_parent_to_worker[0]);
    close(fd_worker_to_parent[1]);

    //If this leaf node is a left program
    if (calculation_program == 0)
    {
        //Calculation program does not need any arguments
        char *args_calculation[] = {"./pl", NULL};
        execvp(args_calculation[0], args_calculation);
    }
    //Else lr == 1, this leaf node is a right program
    else
    {
        //Calculation program does not need any arguments
        char *args_calculation[] = {"./pr", NULL};
        execvp(args_calculation[0], args_calculation);
    }
}
```



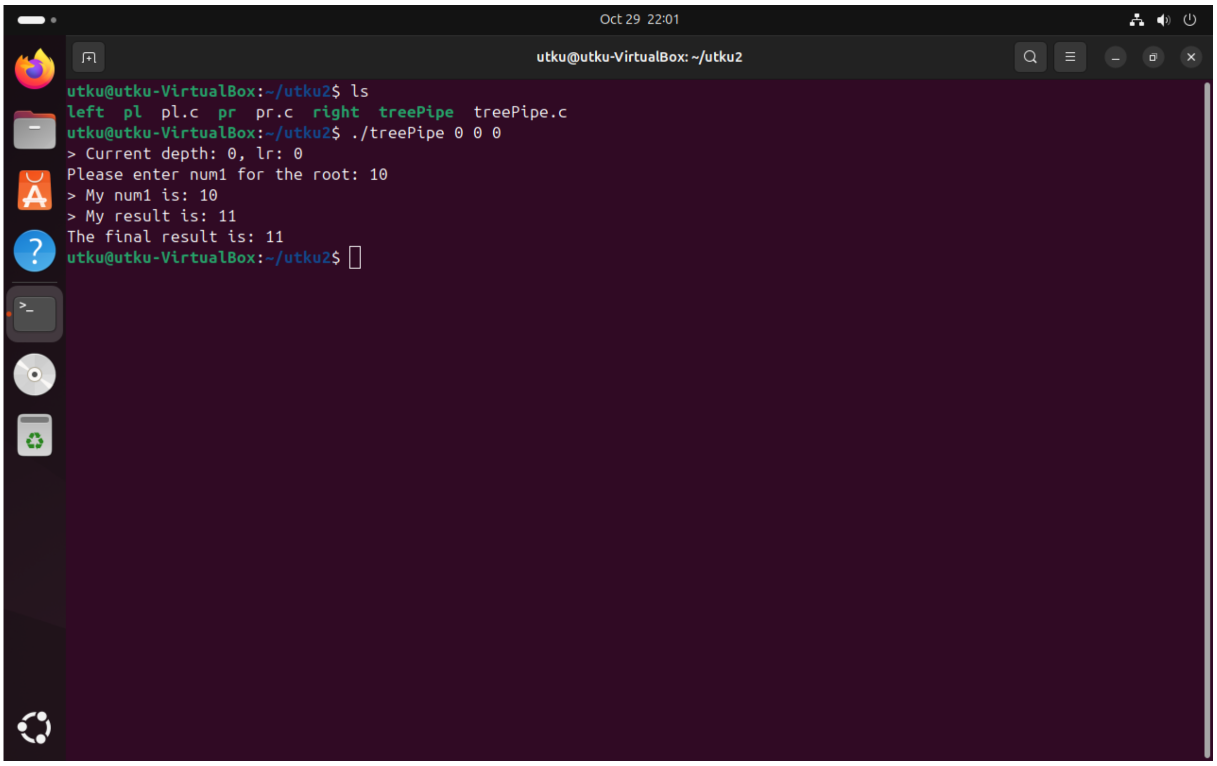
```c
else
{
    //We are in leaf node
    //Close unused pipes
    close(fd_parent_to_worker[0]);
    close(fd_worker_to_parent[1]);

    //Leaf node should provide the num1 and default number to the worker
    dprintf(fd_parent_to_worker[1], "%d\n", num1);
    dprintf(fd_parent_to_worker[1], "%d\n", default_value);
    close(fd_parent_to_worker[1]);

    //Parent should print the num1 before the calculation
    for (int i = 0; i < current_depth; i++)
    {
        fprintf(stderr, "---");
    }
    fprintf(stderr, "> My num1 is: %d\n", num1);


    //Wait for the calculation to be done
    wait(NULL);

    //Leaf node reads result from the pipe
    char buffer[100];
    read(fd_worker_to_parent[0], buffer, sizeof(buffer));
    close(fd_worker_to_parent[0]);
    sscanf(buffer, "%d", &num1);

    //After getting the result parent should print the result
    for (int i = 0; i < current_depth; i++)
    {
        fprintf(stderr, "---");
    }
    fprintf(stderr, "> My result is: %d\n", num1);

    //Leaf node should send it to the parent or console
    if (current_depth == 0)
    {
        printf("The final result is: %d\n", num1);
    }
    else
    {
        printf("%d\n", num1);
    }
```

The Leaf process first sends the numbers num1 and default value to the worker program it created via the pipe. Then it waits for the response. When the response comes, it pulls the response via the pipe. Now there are two options. If it is also a root process, it prints the result it received directly to the console. If not, since it was already created by another PARENT process, its standard output is directed to a pipe. In other words, it sends this result to its own PARENT.

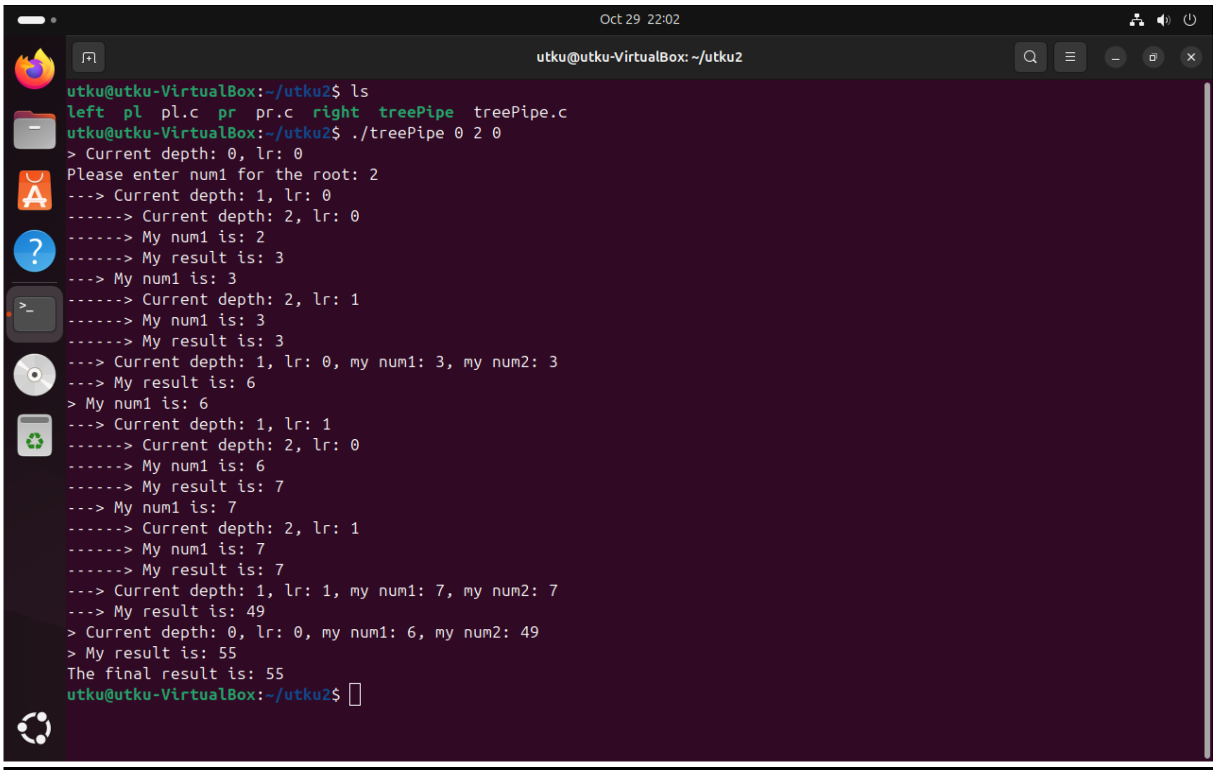
<u>OUTPUT</u>



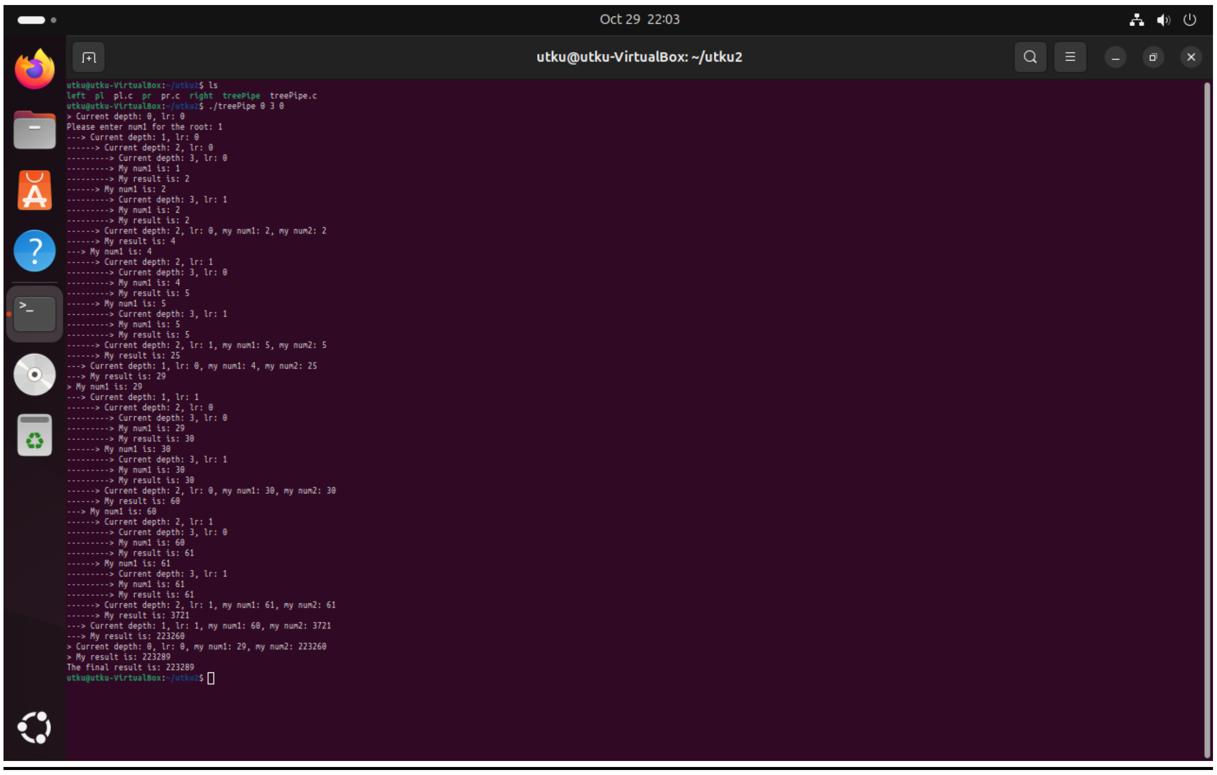
```
utku@utku-VirtualBox:~$ cd utku2
utku@utku-VirtualBox:~/utku2$ ls
left  pl  pl.c  pr  pr.c  right  treePipe  treePipe.c
utku@utku-VirtualBox:~/utku2$ gcc -o left pl.c
utku@utku-VirtualBox:~/utku2$ gcc -o right pr.c
utku@utku-VirtualBox:~/utku2$ gcc -o treePipe treePipe.c
utku@utku-VirtualBox:~/utku2$ ./treePipe
Usage: treePipe <current depth> <max depth> <left-right>
utku@utku-VirtualBox:~/utku2$ ./treePipe 0 0 3
Error: '3' should be either 0 or 1.
utku@utku-VirtualBox:~/utku2$ ./treePipe -1 0 0
Error: '-1' is not a valid non-negative integer.
utku@utku-VirtualBox:~/utku2$ ./treePipe 0 -2 0
Error: '-2' is not a valid non-negative integer.
utku@utku-VirtualBox:~/utku2$ ./treePipe 1 0 0
Error: The second integer (0) must be greater than or equal to the first (1).
utku@utku-VirtualBox:~/utku2$ 
```



```
utku@utku-VirtualBox:~/utku2$ ls
left  pl  pl.c  pr  pr.c  right  treePipe  treePipe.c
utku@utku-VirtualBox:~/utku2$ ./treePipe 0 0 0
> Current depth: 0, lr: 0
Please enter num1 for the root: 10
> My num1 is: 10
> My result is: 11
The final result is: 11
utku@utku-VirtualBox:~/utku2$ 
```



```
utku@utku-VirtualBox:~/utku2$ ls
left  pl  pl.c  pr  pr.c  right  treePipe  treePipe.c
utku@utku-VirtualBox:~/utku2$ ./treePipe 0 2 0
> Current depth: 0, lr: 0
Please enter num1 for the root: 2
---> Current depth: 1, lr: 0
------> Current depth: 2, lr: 0
------> My num1 is: 2
------> My result is: 3
---> My num1 is: 3
------> Current depth: 2, lr: 1
------> My num1 is: 3
------> My result is: 3
---> Current depth: 1, lr: 0, my num1: 3, my num2: 3
---> My result is: 6
> My num1 is: 6
---> Current depth: 1, lr: 1
------> Current depth: 2, lr: 0
------> My num1 is: 6
------> My result is: 7
---> My num1 is: 7
------> Current depth: 2, lr: 1
------> My num1 is: 7
------> My result is: 7
---> Current depth: 1, lr: 1, my num1: 7, my num2: 7
---> My result is: 49
> Current depth: 0, lr: 0, my num1: 6, my num2: 49
> My result is: 55
The final result is: 55
utku@utku-VirtualBox:~/utku2$ 
```



```
utku@utku-VirtualBox:~/utku2$ ls
left  pl  pl.c  pr  pr.c  right  treePipe  treePipe.c
utku@utku-VirtualBox:~/utku2$ ./treePipe 0 3 0
> Current depth: 0, lr: 0
Please enter num1 for the root: 1
---> Current depth: 1, lr: 0
------> Current depth: 2, lr: 0
---------> Current depth: 3, lr: 0
---------> My num1 is: 1
---------> My result is: 2
------> My num1 is: 2
---------> Current depth: 3, lr: 1
---------> My num1 is: 2
---------> My result is: 2
------> Current depth: 2, lr: 0, my num1: 2, my num2: 2
------> My result is: 4
---> My num1 is: 4
------> Current depth: 2, lr: 1
---------> Current depth: 3, lr: 0
---------> My num1 is: 4
---------> My result is: 5
------> My num1 is: 5
---------> Current depth: 3, lr: 1
---------> My num1 is: 5
---------> My result is: 5
------> Current depth: 2, lr: 1, my num1: 5, my num2: 5
------> My result is: 25
---> Current depth: 1, lr: 0, my num1: 4, my num2: 25
---> My result is: 29
> My num1 is: 29
---> Current depth: 1, lr: 1
------> Current depth: 2, lr: 0
---------> Current depth: 3, lr: 0
---------> My num1 is: 29
---------> My result is: 30
------> My num1 is: 30
---------> Current depth: 3, lr: 1
---------> My num1 is: 30
---------> My result is: 30
------> Current depth: 2, lr: 0, my num1: 30, my num2: 30
------> My result is: 60
---> My num1 is: 60
------> Current depth: 2, lr: 1
---------> Current depth: 3, lr: 0
---------> My num1 is: 60
---------> My result is: 61
------> My num1 is: 61
---------> Current depth: 3, lr: 1
---------> My num1 is: 61
---------> My result is: 61
------> Current depth: 2, lr: 1, my num1: 61, my num2: 61
------> My result is: 3721
---> Current depth: 1, lr: 1, my num1: 60, my num2: 3721
---> My result is: 223260
> Current depth: 0, lr: 0, my num1: 29, my num2: 223260
> My result is: 223289
The final result is: 223289
utku@utku-VirtualBox:~/utku2$ 
```

SUMMARY

In short, the implementation works recursively like this. A process creates first left and then right children as long as there is depth. Similarly, these children create first left and then right children as long as there is depth. Creating children continues until the maximum depth is reached. The process with the maximum depth sends the result it has found to the next higher depth. The PARENT process, which sends the result it received from the left child to its own right child, runs its own worker program when the result comes from its right child and sends the numbers it received from the left and right children to its program. It also sends the result of the program to its own PARENT.

NOTE: I changed the output of errors, they will be same with the first error ‘Usage…’.