Operating Systems HW1 Report

Zeynep Gülhan Uslu - 504192524,

30.03.2023

1 Introduction

In the operating systems, the fork system call allows a parent process to create one or more child processes, forming a tree-like structure known as a process tree. This document is an explanation of codes that given in homework.

2 Question 1

The process tree requested in the first question has the following features: The left side of the tree is always the same and the right part of the given process tree is parametric and its depth changes by the N value, which can take any positive integer[Figure 1].

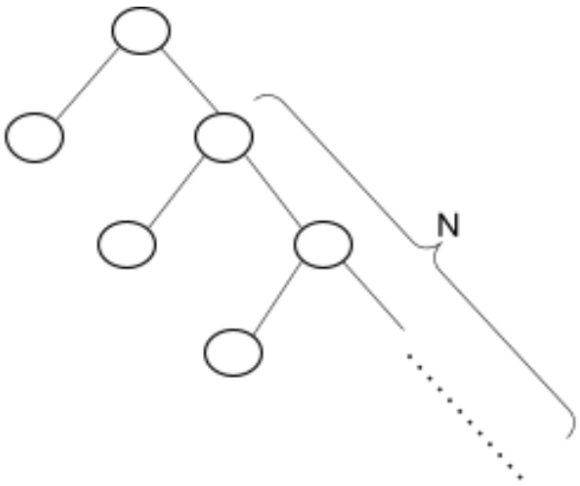

Figure 1: Question 1 Process Tree

2.1 Solution Code

The fork() function is a system call in Unix and Unix-like operating systems that creates a new process by duplicating the calling process. The new process is called the child process, and the original process is called the parent process.

When the fork() function is called, a new process is created that is an exact copy of the calling process. This means that both the parent and the child processes start executing at the same point in the program, and both have their own copies of the program code, data, and stack.

However, the child process has its own unique process ID (PID), and a copy of the parent's file descriptor table. The file descriptor table is a table that keeps track of open files and other I/O resources associated with the process. Each file descriptor in the table is a reference to a system-wide table of open files.

To create a left child and a right child process, we can use the fork() function twice in the parent process. The first call to fork() will create the left child process, and the second call to fork() will create the right child process. Here is an example code for creating right and left child.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    int pid1 = fork();  // create left child process
    if (pid1 == 0) {
        // code for left child process
    } else {
        int pid2 = fork();  // create right child process
        if (pid2 == 0) {
            // code for right child process
        } else {
            // code for parent process
        }
    }
    return 0;

}
```

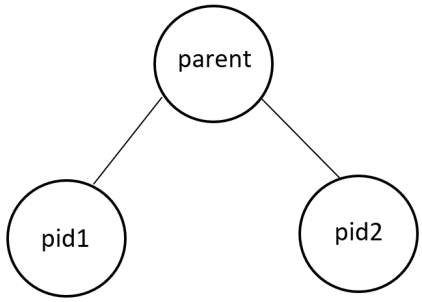

Figure 2: Sample Tree with Left and Right Child

Now we can create for expected process tree like this below:

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

#define ANSI_COLOR_RED     "\x1b[31m"
#define ANSI_COLOR_GREEN   "\x1b[32m"
#define ANSI_COLOR_PURPLE  "\x1b[35m"
#define ANSI_COLOR_RESET   "\x1b[0m"
/**
 * This code creates a necessary process tree with given depth of n
 * */
void create_process_tree(int n, int initial_depth) {
    if (n == 0) {
        return;
    }
    int pid1 = fork();
    if (pid1 < 0) {
        // if pid number below 0 that means fork failed.
        printf("Fork failed\n");
        exit(EXIT_FAILURE);
    } else if (pid1 == 0) {
        // if pid is zero that means it is a child process. It is a left child process.
        printf(ANSI_COLOR_RED "Left   Child   PID: %5d \t Parent PID: %5d \t depth %2d\n" ANSI_COLOR_RESET, getpid(), getppid(),
      initial_depth);
        exit(EXIT_SUCCESS);
    } else {
        // This line creates a new child process using fork.
        int pid2 = fork();
        if (pid2 < 0) {
            printf("Fork failed\n");
            exit(EXIT_FAILURE);
        } else if (pid2 == 0) {
            // right child process created in here
            printf(ANSI_COLOR_GREEN "Right  Child   PID: %5d \t Parent PID: %5d \t depth %2d\n" ANSI_COLOR_RESET, getpid(), getppid(),
       initial_depth);
            if (n > 1) {
                // If n is greater than 1, the function recursively calls itself with n-1 and depth+1 as parameters
                create_process_tree(n - 1, initial_depth + 1);
            } else {
                printf("last depth of a tree:\n");
                // last depth - create only left child.
                int pid3 = fork();
                if (pid3 < 0) {
                    printf("Fork failed\n");
                    exit(EXIT_FAILURE);
                } else if (pid3 == 0) {
                    printf(ANSI_COLOR_RED "Left   Child   PID: %5d \t Parent PID: %5d \t depth %2d  \n" ANSI_COLOR_RESET, getpid(),
      getppid(), initial_depth);
                    exit(EXIT_SUCCESS);
                } else {
                    printf(ANSI_COLOR_PURPLE "Parent Process PID: %5d \t child PID:  %5d \t depth %2d\n" ANSI_COLOR_RESET, getpid(),
      pid3, initial_depth);
                    exit(EXIT_SUCCESS);
                }
            }
        } else {
            // parent
            printf(ANSI_COLOR_PURPLE "Parent Process PID: %5d \t child PIDs: %d,%d \t depth %2d\n" ANSI_COLOR_RESET, getpid(), pid1,
      pid2, initial_depth);
            waitpid(pid1, NULL, 0);
            waitpid(pid2, NULL, 0);
        }
    }
}


int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <n> where n is depth of right sub-tree. n must be a positive integer.\n", argv[0]);
        return EXIT_FAILURE;
    }

    int n = atoi(argv[1]);

    create_process_tree(n, 0);

    return EXIT_SUCCESS;
}

}
```

This if statement checks if n is greater than 1. If it is, then the function *create_process_tree* is called recursively with $n - 1$ and $initial\_depth + 1$ as parameters. This means that a new level of the process tree will be created, with one less process than the previous level, and with a greater depth value.

For example, if n is initially set to 3 and *initial_depth* is initially set to 0, the first call to *create_process_tree* will create two child processes and call itself recursively with $n-1$ (which is 2) and *initial_depth* $+1$ (which is 1). This will create another level of the process tree with two child processes, each with their own child processes, and so on until n is 1, at which point the recursive calls will stop.

If n is not greater than 1, then the else block is executed, which creates only a left child process at the current level of the tree, using the same *initial_depth* value. This means that this is the last level of the process tree.

n parameter is depth of a right sub-tree. *initial_depth* parameter is the starting depth. It is just to print level of children. It will be always 0.

You can see the sample run for $N = 3$ in Figure[3]. Right children are printed in green, left children are printed in red and parents are printed in purple.

```
Parent Process PID:   446        child PIDs: 447,448     depth  0
Left   Child   PID:   447        Parent PID:   446       depth  0
Right  Child   PID:   448        Parent PID:   446       depth  0
Left   Child   PID:   449        Parent PID:   448       depth  1
Parent Process PID:   448        child PIDs: 449,450     depth  1
Right  Child   PID:   450        Parent PID:   448       depth  1
Parent Process PID:   450        child PIDs: 451,452     depth  2
Left   Child   PID:   451        Parent PID:   450       depth  2
Right  Child   PID:   452        Parent PID:   450       depth  2
last depth of a tree:
Parent Process PID:   452        child PID:    453       depth  2
Left   Child   PID:   453        Parent PID:   452       depth  2
```

Figure 3: Sample Run for N=3

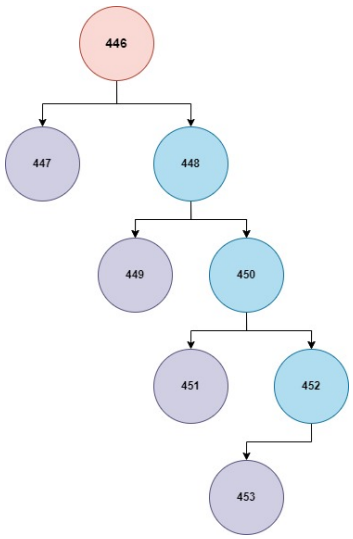

Figure 4: Sample Tree with N=3

2.2 Mathematical Expression of Tree

In the given code, right sub-tree has N processes. Each right child have a left sibling also the last one has one left child. So the left sub-tree has $N + 1$ processes.

Therefore, the total number of processes created can be calculated by the formula:

$$total\_processes = 2 * N + 1$$

Out of these total processes, all but the left processes can be identified as parent processes. Therefore, the expression for the number of created processes that can be identified as parent processes is:

$$parent\_processes = N$$

Note: If we include main parent process, we should add one to parent count.

3 Question 2

In the second question, we extend the tree as left sub-trees' depths parametric too. The depths of each left sub-tree changes with positive integer M.

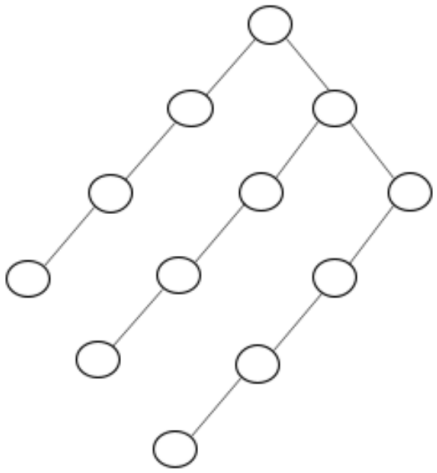

Figure 5: Question 2 Process Tree

3.1 Solution Code

Now we can create for expected process tree like this below:

```c
void create_process_tree(int m, int n) {
    if (m > 0) {
        if (fork() == 0) {
            printf(ANSI_COLOR_RED "Left  Child  PID: %5d \t Parent PID: %5d\n" ANSI_COLOR_RESET, getpid(), getppid());
            // recursively call create_process_tree with m-1 and 0
            create_process_tree(m - 1, 0);
            exit(EXIT_SUCCESS);
        }
    }
    if (n > 0) {
        if (m == 0) {
            if (fork() == 0) {
                printf(ANSI_COLOR_GREEN  "Right Child  PID: %5d \t Parent PID: %5d\n" ANSI_COLOR_RESET, getpid(),
                       getppid());
                // recursively call create_process_tree with 0 and n-1
                create_process_tree(0, n - 1);
                exit(EXIT_SUCCESS);
            }
        } else {
            if (fork() == 0) {
                printf(ANSI_COLOR_GREEN "Right Child  PID: %5d \t Parent PID: %5d\n" ANSI_COLOR_RESET, getpid(),
                       getppid());
                // recursively call create_process_tree with m and n-1
                create_process_tree(m, n - 1);
                exit(EXIT_SUCCESS);
            }
        }
    }
    // wait for all child processes to finish
    while (wait(NULL) > 0);
}
int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <m> <n> where m is depth of left sub-tree, n is depth of right sub-tree. m and n must be a positive integer
      .\n",
               argv[0]);
        return EXIT_FAILURE;
    }
    int m = atoi(argv[1]);
    int n = atoi(argv[2]);
    create_process_tree(m, n);
    return EXIT_SUCCESS;
    }
```

This function takes m and n as an argument. m for left sub-tree depth, n for right sub-tree depth.

In the first recursive case ($if(m > 0)$), the function checks if m is greater than zero. If it is, it creates a new child process with fork(), and then calls *create_process_tree* with $m - 1$ and 0. This means that the child process will create a left child of its own (because $m - 1$ is greater than zero), but no right child (because n is zero). Once the child process is done creating its own child processes, it exits using $exit(EXIT\_SUCCESS)$.

In the second recursive case ($if(n > 0)$), the function checks if n is greater than zero. If it is, it creates a new child process with fork(), and then calls *create_process_tree* with either 0 and $n - 1$ (if m is zero), or m and$n - 1$ (if m is greater than zero). This means that the child process will create a right child of its own (because $n - 1$ is greater than zero), and either a left child or no left child depending on the value of m. Once the child process is done creating its own child processes, it exits using $exit(EXIT\_SUCCESS)$.

You can see the sample run for $M = 3, N = 2$ in Figure[6]. Right children are printed in green, left children are printed in red.

```
Left  Child  PID:   462          Parent PID:   461
Right Child  PID:   463          Parent PID:   461
Left  Child  PID:   464          Parent PID:   462
Left  Child  PID:   465          Parent PID:   463
Right Child  PID:   466          Parent PID:   463
Left  Child  PID:   467          Parent PID:   464
Left  Child  PID:   469          Parent PID:   466
Left  Child  PID:   468          Parent PID:   465
Left  Child  PID:   470          Parent PID:   468
Left  Child  PID:   471          Parent PID:   469
Left  Child  PID:   472          Parent PID:   471
```

Figure 6: Question 2 Sample Run for M = 3, N = 2

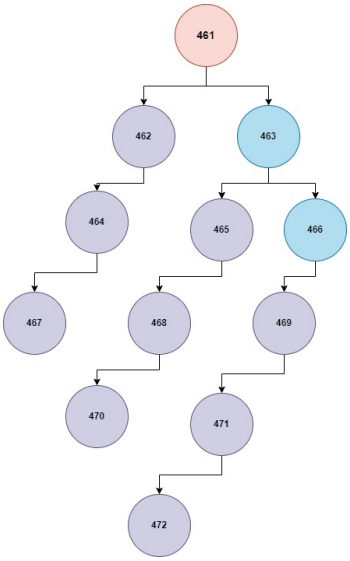

Figure 7: Sample Tree with M=3, N=2

3.2 Mathematical Expression of Tree

In the given code, right sub-tree has N processes. Each right child is also parent and have a M left sibling. Also the last one has M left child. So the left sub-tree has $(N+1)*M$ processes.

Therefore, the total number of processes created can be calculated by the formula:

$$total\_processes = N + ((N+1) * M)$$

Out of these total processes, all but the last processes can be identified as parent processes. Therefore, the expression for the number of created processes that can be identified as parent processes is:

$$parent\_processes = N + ((N+1) * (M-1))$$

Note: If we include main parent process, we should add one to parent count.

4 Running the Code

I upload the code with zip file. The directory contains these files:

- q1.c - Question 1
- q2.c - Question 2
- Makefile
- README.md

I will explain how to run these code in below. Also, you can find the usage in "README.md" file.

You can compile the project like this:

```
cd ./hw1
make
```

4.1 Running Question 1

You can run the first question with this command.

```
Usage: ./q1 <n> where n is depth of right sub-tree.
        n must be a positive integer.

./q1 3
```

4.2 Running Question 2

You can run the second question with this command.

```

Usage: ./q2 <m> <n> where m is depth of left sub-tree,
      n is depth of right sub-tree.
      m and n must be a positive integer.

./q2 3 2
```