

# Operating Systems

## Lab Manual # 5

### Process Creation and Parent/Child Interaction

#### Objectives:

- Understand the concepts of process creation and parent/child interactions in Unix-like operating systems.
- Learn how to use the `fork` system call to create a new process and child process.
- Observe and trace the interaction between parent and child processes.

#### Pre-requisites:

- Basic understanding of GCC and C language in Ubuntu.
- Completion of Lab 4.
- Ubuntu operating system (either installed on your computer or in a virtual machine).

#### Lab Tasks # 1: Exercise 1: Understanding Process Creation and Interaction

- Below is an example program that demonstrates parent and child process creation with the usage of variables. In this example, the parent and child processes each have their own variables, and they communicate by sharing a variable.

```
c
```

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

int main() {
    // Variable declaration
    int parentVariable = 10;
    int childVariable = 5;

    pid_t childPid = fork();

    if (childPid == 0) {
        // Child process
        childVariable += 10; // Modify child variable
        printf("Child Process: Variable = %d\n", childVariable);
    } else if (childPid > 0) {
        // Parent process

        // Parent process
        parentVariable += 20; // Modify parent variable
        printf("Parent Process: Variable = %d\n", parentVariable);
        wait(NULL); // Wait for the child process to complete
    } else {
        // Fork failed
        perror("Error in fork");
        return 1;
    }

    return 0;
}
```

## Exercise 2: Instructions to Run

- Open a terminal and navigate to the directory where you saved the program.
- Compile the program using the following command:

```
$ gcc process_creation.c -o process_creation
```

- Execute the program:

```
$ ./process_creation
```

## Explanation:

1. Both the parent and child processes have their own `parentVariable` and `childVariable`.
2. The parent process increments its `parentVariable` by 20, while the child process increments its `childVariable` by 10.
3. The output will show the modified values of the variables in both the parent and child processes.
4. The `wait` system call is used in the parent process to wait for the child process to complete before printing the final output.

## Sample Output:

```
Parent Process: Variable = 30
Child Process: Variable = 15
```

### **Exercise 3: Modify and Experiment**

- Modify the program to execute a different command in the child process (e.g., using `float`, `char` and `double` data-types).
- Re-compile the program and observe the changes in the output.
- Experiment with different commands in the child process and analyze the results.

### **Exercise 4: Instructions to Run**

- Open the program in a text editor and make the necessary modifications.
- Save the changes and re-compile the program.
- Execute the modified program:

```
$ ./process_creation
```

#### **Explanations:**

- By modifying the program and changing the command executed in the child process, you can observe how the child process behavior affects the overall output.

### **Exercise 5: Handling Errors**

- To understand error handling in process creation.
- Intentionally introduce an error in the data type explicit casting/conversions in the child process (e.g., provide an incorrect data value conversion).
- Observe how the program handles the error.

### **Exercise 6: Instructions**

- Open the program in a text editor and intentionally introduce an error in the `int`, `float`, `char` data type conversions.
- Save the changes and re-compile the program.
- Execute the modified program
- Observe the error message and understand how the program responds to the error.

#### **Explanations:**

- Introducing an error in the data types simulates a scenario where the child process cannot execute the specified conversion in data types.

## **Lab Tasks # 2: Exercise 1: Calculate the factorial of a number using parent/child.**

- Below is the example, which demonstrate a parent and child process creation where the child process calculates the factorial of a number passed from the parent process.

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

int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    // Variable declaration
    int number = 5; // Number for which factorial will be calculated

    pid_t childPid = fork();

    if (childPid == 0) {
        // Child process
        int result = factorial(number);
        printf("Child Process: Factorial of %d = %d\n", number, result);
    } else if (childPid > 0) {
        // Parent process
        printf("Parent Process: Child process is calculating the factorial...\n");
        wait(NULL); // Wait for the child process to complete
        printf("Parent Process: Child process completed.\n");
    } else {
        // Fork failed
        perror("Error in fork");
        return 1;
    }

    return 0;
}
```

### Exercise 2: Instructions to Run

- Open a terminal and navigate to the directory where you saved the program.
- Compile the program using the following command:

```
$ gcc process_creation.c -o process_creation
```

- Execute the program:

```
$ ./process_creation
```

### Explanation:

1. The parent process sets the value of `number` to 5.
2. The child process calculates the factorial of the given number using a recursive factorial function.
3. The parent process prints a message indicating that the child process is calculating the factorial and then waits for the child process to complete before printing another message.

### Sample Output:

```
Parent Process: Child process is calculating the factorial...
```

```
Child Process: Factorial of 5 = 120
```

```
Parent Process: Child process completed.
```

### Exercise 3: Modify and Experiment

- Modify the program to execute a different command in the child process (e.g., using different factorial values of `integer`).
- Re-compile the program and observe the changes in the output.
- Experiment with different commands in the child process and analyze the results.

### Exercise 4: Instructions to Run

- Open the program in a text editor and make the necessary modifications.

- Save the changes and re-compile the program.
- Execute the modified program:

```
$ ./process_creation
```

#### Explanations:

This example illustrates how the child process can perform a specific computation (in this case, calculating the factorial) while the parent process continues with its own tasks. The `wait` system call ensures that the parent process waits for the child process to finish before proceeding.

#### Exercise 5: Handling Errors

- To understand error handling in process creation.
- Intentionally introduce an error in the factorial calculations in the child process (e.g., provide an incorrect integer data value).
- Observe how the program handles the error.

#### Exercise 6: Instructions

- Open the program in a text editor and intentionally introduce an error in the factorial loop.
- Save the changes and re-compile the program.
- Execute the modified program
- Observe the error message and understand how the program responds to the error.

#### Lab Tasks # 3: Exercise 1: Calculate the average of random numbers using parent/child.

- Below is the example, which demonstrate a parent and child process creation where the child process generates random numbers and the parent process calculates their average.

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

#define NUM_NUMBERS 5

int main() {
    // Variables
    int numbers[NUM_NUMBERS];
    double average = 0;

    // Seed the random number generator
    srand((unsigned int) getpid());

    pid_t childPid = fork();

    if (childPid == 0) {
        // Child process
        printf("Child Process: Generating random numbers...\n");
        for (int i = 0; i < NUM_NUMBERS; ++i) {
            numbers[i] = rand() % 100; // Generate random numbers between 0 and 99
            printf("Child Process: Random Number %d: %d\n", i + 1, numbers[i]);
        }
    } else if (childPid > 0) {
        // Parent process
        wait(NULL); // Wait for the child process to complete

        // Calculate the average in the parent process
        for (int i = 0; i < NUM_NUMBERS; ++i) {
            average += numbers[i];
        }
        average /= NUM_NUMBERS;

        printf("Parent Process: Average of Random Numbers: %.2f\n", average);
    }
}
```

```
    } else {
        // Fork failed
        perror("Error in fork");
        return 1;
    }

    return 0;
}
```

### Exercise 2: Instructions to Run

- Open a terminal and navigate to the directory where you saved the program
- Compile the program using the following command:

```
$ gcc process_creation.c -o process_creation
```

- Execute the program:

```
$ ./process_creation
```

### Explanation:

1. The parent process and the child process each have their own array of random numbers.
2. The child process generates and prints random numbers.
3. The parent process waits for the child process to complete and then calculates the average of the generated numbers.

### Sample Output:

```
Child Process: Generating random numbers...
Child Process: Random Number 1: 23
Child Process: Random Number 2: 67
Child Process: Random Number 3: 45
Child Process: Random Number 4: 12
Child Process: Random Number 5: 89
Parent Process: Average of Random Numbers: 47.20
```

### Exercise 3: Modify and Experiment

- Modify the program to execute a different command in the child process (e.g., using different random seed values of `integer`).
- Re-compile the program and observe the changes in the output.
- Experiment with different commands in the child process and analyze the results.

### Exercise 4: Instructions to Run

- Open the program in a text editor and make the necessary modifications.
- Save the changes and re-compile the program.
- Execute the modified program:

```
$ ./process_creation
```

### Explanations:

This example illustrates how the child process can perform a specific computation (in this case, calculating the factorial) while the parent process continues with its own tasks. The `'wait'` system call ensures that the parent process waits for the child process to finish before proceeding.

### Exercise 5: Handling Errors

- To understand error handling in process creation.
- Intentionally introduce an error in the random number generations and calculations in the child process (e.g., provide an incorrect integer seed (float) value).
- Observe how the program handles the error.

## Exercise 6: Instructions

- Open the program in a text editor and intentionally introduce an error in the random generation code.
- Save the changes and re-compile the program.
- Execute the modified program
- Observe the error message and understand how the program responds to the error.

## Lab Tasks # 4: Exercise 1: Create a shared file using parent/child interactions.

- Below is the example, which demonstrate a parent and child process creation where both processes work together to read and write to a shared file.

```
c

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

#define FILE_PATH "shared_file.txt"

int main() {
    // Variables
    FILE *sharedFile;
    int dataFromParent = 42;
    int dataFromChild = 0;

    // Create or open the shared file
    sharedFile = fopen(FILE_PATH, "w+");
    if (sharedFile == NULL) {
        perror("Error opening the file");
        return 1;
    }
```

```

pid_t childPid = fork();

if (childPid == 0) {
    // Child process
    printf("Child Process: Reading data from the file...\n");
    fseek(sharedFile, 0, SEEK_SET);
    fscanf(sharedFile, "%d", &dataFromChild);
    printf("Child Process: Data read from the file: %d\n", dataFromChild);
} else if (childPid > 0) {
    // Parent process
    printf("Parent Process: Writing data to the file...\n");
    fseek(sharedFile, 0, SEEK_SET);
    fprintf(sharedFile, "%d", dataFromParent);
    wait(NULL); // Wait for the child process to complete

    // Read data from the file to confirm the child's read
    fseek(sharedFile, 0, SEEK_SET);
    fscanf(sharedFile, "%d", &dataFromChild);
    printf("Parent Process: Data read by the child from the file: %d\n", dataFromChild);
} else {
    // Fork failed
    // Fork failed
    perror("Error in fork");
    return 1;
}

// Close the file
fclose(sharedFile);

return 0;
}

```

### Exercise 2: Instructions to Run

- Open a terminal and navigate to the directory where you saved the program.
- Compile the program using the following command:

```
$ gcc process_creation.c -o process_creation
```

- Execute the program:

```
$ ./process_creation
```

### Explanation:

1. The parent process writes data (42 in this case) to a shared file.
2. The child process reads the data from the same file.
3. The parent process waits for the child process to complete before reading and printing the data read by the child.

### Sample Output:

```
Parent Process: Writing data to the file...
Child Process: Reading data from the file...
Child Process: Data read from the file: 42
Parent Process: Data read by the child from the file: 42
```

### Exercise 3: Modify and Experiment

- Modify the program to execute a different command in the child process (e.g., using different values of `integer`).
- Re-compile the program and observe the changes in the output.
- Experiment with different commands in the child process and analyze the results.

### Exercise 4: Instructions to Run

- Open the program in a text editor and make the necessary modifications.
- Save the changes and re-compile the program.
- Execute the modified program:

```
$ ./process_creation
```

### **Explanations:**

This example illustrates how the child process can perform a specific computation (in this case, calculating the factorial) while the parent process continues with its own tasks. The `wait` system call ensures that the parent process waits for the child process to finish before proceeding.

### **Exercise 5: Handling Errors**

- To understand error handling in process creation.
- Intentionally introduce an error in the shared file paths in the child process (e.g., provide an incorrect file path).
- Observe how the program handles the error.

### **Exercise 6: Instructions**

- Open the program in a text editor and intentionally introduce an error in the random generation code.
- Save the changes and re-compile the program.
- Execute the modified program
- Observe the error message and understand how the program responds to the error.

### **Lab Tasks # 5: Exercise 1: Create a shared file using pointer arrays of parent/child.**

- Below is the example, which demonstrate parent and child process creation where both processes use pointers and file handling to read and write arrays to a shared file.

```
c
```

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

#define FILE_PATH "shared_array_file.txt"
#define ARRAY_SIZE 5

void writeArrayToFile(FILE *file, int *array, int size) {
    for (int i = 0; i < size; ++i) {
        fprintf(file, "%d ", array[i]);
    }
}

void readArrayFromFile(FILE *file, int *array, int size) {
    for (int i = 0; i < size; ++i) {
        fscanf(file, "%d", &array[i]);
    }
}

int main() {
    // Variables
    FILE *sharedFile;
    int parentArray[ARRAY_SIZE] = {1, 2, 3, 4, 5};
    int childArray[ARRAY_SIZE];

    // Create or open the shared file
    sharedFile = fopen(FILE_PATH, "w+");
    if (sharedFile == NULL) {
        perror("Error opening the file");
        return 1;
    }

    pid_t childPid = fork();

    if (childPid == 0) {
        // Child process
        printf("Child Process: Reading array from the file...\n");
    }
}
```

```
fseek(sharedFile, 0, SEEK_SET);
readArrayFromFile(sharedFile, childArray, ARRAY_SIZE);
printf("Child Process: Array read from the file: ");
for (int i = 0; i < ARRAY_SIZE; ++i) {
    printf("%d ", childArray[i]);
}
printf("\n");
} else if (childPid > 0) {
    // Parent process
    printf("Parent Process: Writing array to the file...\n");
    fseek(sharedFile, 0, SEEK_SET);
    writeArrayToFile(sharedFile, parentArray, ARRAY_SIZE);
    wait(NULL); // Wait for the child process to complete

    // Read array from the file to confirm the child's read
    fseek(sharedFile, 0, SEEK_SET);
    readArrayFromFile(sharedFile, childArray, ARRAY_SIZE);
    printf("Parent Process: Array read by the child from the file: ");




```

```
for (int i = 0; i < ARRAY_SIZE; ++i) {
    printf("%d ", childArray[i]);
}
printf("\n");
} else {
    // Fork failed
    perror("Error in fork");
    return 1;
}

// Close the file
fclose(sharedFile);

return 0;
}
```

### **Exercise 2: Instructions to Run**

- Open a terminal and navigate to the directory where you saved the program.
- Compile the program using the following command:

```
$ gcc process_creation.c -o process_creation
```

- Execute the program:

```
$ ./process_creation
```

#### **Explanation:**

1. The parent process writes an array of integers to a shared file.
2. The child process reads the array from the same file.
3. Both processes use pointers to handle arrays and file operations.

#### **Sample Output:**

```
Parent Process: Writing array to the file...
Child Process: Reading array from the file...
Child Process: Array read from the file: 1 2 3 4 5
Parent Process: Array read by the child from the file: 1 2 3 4 5
```

### **Exercise 3: Modify and Experiment**

- Modify the program to execute a different command in the child process (e.g., using different values of `integer` arrays).
- Re-compile the program and observe the changes in the output.
- Experiment with different commands in the child process and analyze the results.

### **Exercise 4: Instructions to Run**

- Open the program in a text editor and make the necessary modifications.
- Save the changes and re-compile the program.
- Execute the modified program:

```
$ ./process_creation
```

### Explanations:

This example illustrates how the child process can perform a specific computation (in this case, calculating the factorial) while the parent process continues with its own tasks. The `wait` system call ensures that the parent process waits for the child process to finish before proceeding.

### Exercise 5: Handling Errors

- To understand error handling in process creation.
- Intentionally introduce an error in the shared file paths and array bounds in the child process (e.g., provide an incorrect file path).
- Observe how the program handles the error.

### Exercise 6: Instructions

- Open the program in a text editor and intentionally introduce an error in the array declaration of file handling path code.
- Save the changes and re-compile the program.
- Execute the modified program
- Observe the error message and understand how the program responds to the error.

## Conclusions:

Congratulations! You've successfully completed this lab. In this exploration of parent and child processes in C programming, a foundational understanding of process creation, communication, and collaboration was established. The core `fork` system call was emphasized, illustrating how parent and child processes operate concurrently with separate memory spaces. Various examples demonstrated effective data sharing between processes through mechanisms like files, arrays, and linked lists. These examples showcased the versatility of parallel processing, dividing tasks between parent and child processes, as seen in parallel sorting and searching algorithms. Error handling played a significant role, ensuring the robustness of the code was achieved through shared data structures and synchronization techniques like the `wait` system call. The importance of proper memory management, including allocation and deallocation, was underscored to prevent memory leaks. Overall, these examples provide a practical foundation for comprehending the intricacies of concurrent processes and their collaborative functionality in C programming.