Laboratory Assignments 4
Subject: Design Principles of Operating Systems
Subject code: CSE 3249

Name: A.Ritikh
Registration Number: 2241018124
Section:51

Assignment 4: Familiarization with Process Management in Linux environment.

Objective of this Assignment:

- **To trace the different states of a process during its execution.**
- **To learn the use of different system calls such as (fork(),vfork(),wait(),execl()) for process handling in Unix/Linux environment.**

1. Write a C program to create a child process using fork() system call. The child process will print the message "Child" with its process identifier and then continue in an indefinite loop. The parent process will print the message "Parent" with its process identifier and then continue in an indefinite loop.

a) Run the program and trace the state of both processes.

b) Terminate the child process. Then trace the state of processes.

c) Run the program and trace the state of both processes. Terminate the parent process. Then trace the state of processes.

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

int main() {
    pid_t pid = fork();  // Create a child process

    if (pid == -1) {
        // Error handling if fork() fails
        perror("fork failed");
        return 1;
    }
    else if (pid == 0) {
        // Child process
        while (1) {
            printf("Child Process: PID = %d\n", getpid());
            sleep(1);  // Sleep to simulate indefinite execution
        }
    }
    else {
        // Parent process
        while (1) {
            printf("Parent Process: PID = %d\n", getpid());
            sleep(1);  // Sleep to simulate indefinite execution
        }
    }

    return 0;
}
```

```
Parent Process: PID = 9066
Child Process: PID = 9067
Parent Process: PID = 9066
Child Process: PID = 9067
Child Process: PID = 9067
Parent Process: PID = 9066
Child Process: PID = 9067
Parent Process: PID = 9066
Child Process: PID = 9067
Parent Process: PID = 9066
Child Process: PID = 9067
Parent Process: PID = 9066
Child Process: PID = 9067
Parent Process: PID = 9066
Parent Process: PID = 9066
Child Process: PID = 9067
Child Process: PID = 9067
Parent Process: PID = 9066
Child Process: PID = 9067
Parent Process: PID = 9066
Child Process: PID = 9067
Parent Process: PID = 9066
Child Process: PID = 9067
Parent Process: PID = 9066
Killed
student@iteradmin-vostro-3268:~/Desktop/                    $ Child Process: PID = 9067
Child Process: PID = 9067
Child Process: PID = 9067
Child Process: PID = 9067
Child Process: PID = 9067
Child Process: PID = 9067
```

```
student@iteradmin-vostro-3268:~/Desktop/DOS_2241016204/Assignment4$ ps -all
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    2109    2099  0  80   0 - 76719 do_pol tty2     00:00:00 gnome-sess
0 S  1000    9066    9027  0  80   0 -   670 hrtime pts/1    00:00:00 a.out
1 S  1000    9067    9066  0  80   0 -   670 hrtime pts/1    00:00:00 a.out
0 R  1000    9068    9020 99  80   0 -  5612 -      pts/0    00:00:00 ps
student@iteradmin-vostro-3268:~/Desktop/                          $ kill -9 9066
student@iteradmin-vostro-3268:~/Desktop/                          $ kill -9 9067
student@iteradmin-vostro-3268:~/Desktop/                          $
```

d) Modify the program so that the parent process after displaying the message will wait for child process to complete its task. Again run the program and trace the state of both processes.

e) Terminate the child process. Then trace the state of processes.

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

int main() {
    pid_t pid = fork();  // Create a child process

    if (pid == -1) {
        // Error handling if fork() fails
        perror("fork failed");
        return 1;
    }
    else if (pid == 0) {
        // Child process
        while (1) {
            printf("Child Process: PID = %d\n", getpid());
            sleep(1);  // Sleep to simulate indefinite execution
        }
    }
    else {
        // Parent process
        printf("Parent Process: PID = %d\n", getpid());
        wait(NULL);  // Wait for the child process to complete
        printf("Parent: Child process has completed.\n");
    }

    return 0;
}
```

```
Parent Process: PID = 9129
Child Process: PID = 9130
Child Process: PID = 9130
Child Process: PID = 9130
Child Process: PID = 9130
Child Process: PID = 9130
Child Process: PID = 9130
Child Process: PID = 9130
Child Process: PID = 9130
Child Process: PID = 9130
Child Process: PID = 9130
Child Process: PID = 9130
Parent: Child process has completed
```

```
student@iteradmin-vostro-3268:~/Desktop/                          $ ps -all
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    2109    2099  0  80   0 - 76719 do_pol tty2     00:00:00 gnome-sess
0 S  1000    9129    9027  0  80   0 -   670 do_wai pts/1    00:00:00 a.out
1 S  1000    9130    9129  0  80   0 -   670 hrtime pts/1    00:00:00 a.out
0 R  1000    9131    9020  0  80   0 -  5612 -      pts/0    00:00:00 ps
student@iteradmin-vostro-3268:~/Desktop/                          $ kill -9 9130
student@iteradmin-vostro-3268:~/Desktop/                          $
```

3. Write a C program that will create three child process to perform the following operations respectively:

- First child will copy the content of file1 to file2

- Second child will display the content of file2

- Third child will display the sorted content of file2 in reverse order.

- Each child process being created will display its id and its parent process id with appropriate message.

- The parent process will be delayed for 1 second after creation of each child process. It will display appropriate message with its id after completion of all the child processes.

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

#define MAX_SIZE 1024

// Function to copy the content of one file to another
void copy_file(const char *src, const char *dest) {
    FILE *src_file = fopen(src, "r");
    FILE *dest_file = fopen(dest, "w");
    if (!src_file || !dest_file) {
        perror("Error opening file");
        exit(1);
    }

    char buffer[MAX_SIZE];
    while (fgets(buffer, MAX_SIZE, src_file) != NULL) {
        fputs(buffer, dest_file);
    }

    fclose(src_file);
    fclose(dest_file);
}

// Function to display the content of a file
void display_file(const char *file_name) {
    FILE *file = fopen(file_name, "r");
    if (!file) {
        perror("Error opening file");
        exit(1);
```

```
    char buffer[MAX_SIZE];
    while (fgets(buffer, MAX_SIZE, src_file) != NULL) {
        fputs(buffer, dest_file);
    }

    fclose(src_file);
    fclose(dest_file);
}

// Function to display the content of a file
void display_file(const char *file_name) {
    FILE *file = fopen(file_name, "r");
    if (!file) {
        perror("Error opening file");
        exit(1);
    }

    char buffer[MAX_SIZE];
    while (fgets(buffer, MAX_SIZE, file) != NULL) {
        printf("%s", buffer);
    }

    fclose(file);
}

// Function to display the sorted content of a file in reverse order
void sort_reverse_file(const char *file_name) {
    FILE *file = fopen(file_name, "r");
    if (!file) {
        perror("Error opening file");
        exit(1);
    }

    char lines[MAX_SIZE][MAX_SIZE];
    int line_count = 0;
```

```
    // Read lines from the file
    while (fgets(lines[line_count], MAX_SIZE, file) != NULL) {
        line_count++;
    }

    fclose(file);

    // Sort the lines in reverse order
    for (int i = 0; i < line_count - 1; i++) {
        for (int j = i + 1; j < line_count; j++) {
            if (strcmp(lines[i], lines[j]) < 0) {
                // Swap the lines
                char temp[MAX_SIZE];
                strcpy(temp, lines[i]);
                strcpy(lines[i], lines[j]);
                strcpy(lines[j], temp);
            }
        }
    }

    // Print the sorted lines in reverse order
    for (int i = 0; i < line_count; i++) {
        printf("%s", lines[i]);
    }
}

int main() {
    pid_t pid1, pid2, pid3;

    // Create the first child process to copy file1 to file2
    pid1 = fork();
    if (pid1 == -1) {
        perror("Error creating first child");
        exit(1);
    }
```

```
if (pid1 == 0) {
    // First child process
    printf("First Child Process: PID = %d, Parent PID = %d\n", getpid(), getppid());
    copy_file("file1.txt", "file2.txt");
    exit(0);  // Exit after task is done
} else {
    sleep(1);  // Parent waits for 1 second
}

// Create the second child process to display the content of file2
pid2 = fork();
if (pid2 == -1) {
    perror("Error creating second child");
    exit(1);
}

if (pid2 == 0) {
    // Second child process
    printf("Second Child Process: PID = %d, Parent PID = %d\n", getpid(), getppid());
    display_file("file2.txt");
    exit(0);  // Exit after task is done
} else {
    sleep(1);  // Parent waits for 1 second
}

// Create the third child process to display sorted content of file2 in reverse order
pid3 = fork();
if (pid3 == -1) {
    perror("Error creating third child");
    exit(1);
}

if (pid3 == 0) {
    // Third child process
    printf("Third Child Process: PID = %d, Parent PID = %d\n", getpid(), getppid());
    sort_reverse_file("file2.txt");
    exit(0);  // Exit after task is done
} else {
    sleep(1);  // Parent waits for 1 second
}

// Parent process waits for all child processes to finish
waitpid(pid1, NULL, 0);
waitpid(pid2, NULL, 0);
waitpid(pid3, NULL, 0);

// Parent process displays its message after all children are done
printf("Parent Process: PID = %d, All children have completed their tasks.\n", getpid())

return 0;
```

```
student@iteradmin-vostro-3268:~/Desktop                          $ cat file1.txt
324
764
555
student@iteradmin-vostro-3268:~/Desktop/                           cat file2.txt
324
764
555
student@iteradmin-vostro-3268:~/Desktop                          $ gcc Question_3.c
student@iteradmin-vostro-3268:~/Desktop                          $ ./a.out
First Child Process: PID = 9396, Parent PID = 9395
Second Child Process: PID = 9401, Parent PID = 9395
324
764
555
Third Child Process: PID = 9402, Parent PID = 9395
764
555
324
Parent Process: PID = 9395, All children have completed their tasks._
```

4. Write a C program that will create a child process to generate a Fibonacci series of specified length and store it in an array. The parent process will wait for the child to complete its task and then display the Fibonacci series and then display the prime Fibonacci number in the series along with its position with appropriate message.

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

// Function to check if a number is prime
bool is_prime(int num) {
    if (num <= 1) return false;
    for (int i = 2; i * i <= num; i++) {
        if (num % i == 0) return false;
    }
    return true;
}

// Function to generate Fibonacci series
void generate_fibonacci(int *fib, int length) {
    if (length >= 1) fib[0] = 0;
    if (length >= 2) fib[1] = 1;
    for (int i = 2; i < length; i++) {
        fib[i] = fib[i - 1] + fib[i - 2];
    }
```

```c
int main() {
    pid_t pid;
    int length;

    // Get the length of Fibonacci series from user
    printf("Enter the length of Fibonacci series: ");
    scanf("%d", &length);

    if (length <= 0) {
        printf("Invalid length. Must be a positive integer.\n");
        exit(1);
    }

    int fib[length]; // Array to store the Fibonacci series
    int pipefd[2];   // Pipe for communication between parent and child

    if (pipe(pipefd) == -1) {
        perror("Pipe failed");
        exit(1);
    }

    pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    } else if (pid == 0) {
        // Child process: Generate Fibonacci series
        close(pipefd[0]); // Close reading end of the pipe
        generate_fibonacci(fib, length);
        write(pipefd[1], fib, sizeof(fib)); // Write Fibonacci series to pipe
        close(pipefd[1]); // Close writing end of the pipe
        printf("Child process (PID = %d) completed.\n", getpid());
        exit(0);
    } else {
        // Parent process: Wait for the child and process the data
        close(pipefd[1]); // Close writing end of the pipe
        wait(NULL);       // Wait for child process to complete

        read(pipefd[0], fib, sizeof(fib)); // Read Fibonacci series from pipe
        close(pipefd[0]); // Close reading end of the pipe

        // Display the Fibonacci series
        printf("Parent process (PID = %d): Fibonacci series:\n", getpid());
        for (int i = 0; i < length; i++) {
            printf("%d ", fib[i]);
        }
        printf("\n");

        // Display prime Fibonacci numbers with their positions
        printf("Prime Fibonacci numbers in the series:\n");
        for (int i = 0; i < length; i++) {
            if (is_prime(fib[i])) {
                printf("Position %d: %d\n", i + 1, fib[i]);
            }
        }

        printf("Parent process completed.\n");
    }

    return 0;
}
```

```
Enter the length of Fibonacci series: 10
Child process (PID = 9517) completed.
Parent process (PID = 9516): Fibonacci series:
0 1 1 2 3 5 8 13 21 34
Prime Fibonacci numbers in the series:
Position 4: 2
Position 5: 3
Position 6: 5
Position 8: 13
Parent process completed.
```