Assignment 4

Submitted By: Roshni Pal- 02137180

Part 1:

a) Do newly created child processes, C1 and C2, have their own copies of the different variables declared in the parent process?

Child processes created with fork() will have separate copies of the parent's memory. Variables declared in the parent process are copied to the child process, but changes to variables in the child will not affect the parent.

b) Is there a fixed order of execution between child processes C1 and C2? And how do you
explain that?

No, the execution order of child processes C1 and C2 is not fixed. The scheduler determines when each process runs, leading to different orders across executions.

c) Modify the code so C1 will completes printing its output before C2 starts printing its
own output.

To ensure that child c1 completes its execution before child c2 starts, we can adjust the order of fork() calls and use waitpid() to explicitly wait for C1 to finish before creating C2.

modified code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
```

```
#include <string.h>
#include <sys/wait.h>

#define COUNT 1000
#define PERM 0644

int main(int argc, char *argv[])
{
    int index = 0;
    time_t now;
    int value;
    pid_t C1, C2;
    char outBuffer[128];

    // Seed the random number with different values at different
    srand((unsigned)time(&now));
    value = rand() % 100;

    // Print parent process information
    snprintf(outBuffer, sizeof(outBuffer), "Parent process: my v
    printf("%s\n", outBuffer);
    fflush(stdout);

    // Fork the first child process C1
    C1 = fork();
    if (C1 < 0)
    {
        printf("Error: fork 1 failed\n");
        return -2;
    }

    if (C1 == 0)
    {
        // Child 1 executes this block
        for (index = 1; index <= COUNT; index++)
        {
```

```
            snprintf(outBuffer, sizeof(outBuffer), "Child 1: my
            printf("%s", outBuffer);
            fflush(stdout);
            value++;
        }
        exit(0); // Child 1 terminates after its work is done
    }
    else
    {
        // Parent waits for Child 1 (C1) to complete
        waitpid(C1, NULL, 0);

        // Fork the second child process C2 after C1 has finishe
        C2 = fork();
        if (C2 < 0)
        {
            printf("Error: fork 2 failed\n");
            return -2;
        }

        if (C2 == 0)
        {
            // Child 2 executes this block
            for (index = 1; index <= COUNT; index++)
            {
                snprintf(outBuffer, sizeof(outBuffer), "\tChild
                printf("%s", outBuffer);
                fflush(stdout);
                value++;
            }
            exit(0); // Child 2 terminates after its work is don
        }
        else
        {
            // Parent waits for Child 2 (C2) to complete
            waitpid(C2, NULL, 0);
```

```
        }
    }

    // Parent process terminates
    printf("process %d terminated\n", getpid());
    return 0;
}
```

Part 2: Return a value from child process to parent process (lastnameChildCP.c):

The program creates a child process using fork() and a pipe for communication between the parent and child. The child computes the sum of command-line arguments (numbers) and writes the result to the pipe. The parent waits for the child to finish, reads the sum from the pipe, and prints it. Both processes close their respective unused ends of the pipe to ensure proper communication.

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

int main(int argc, char *argv[]) {
    int pipefd[2];  // Pipe file descriptors: pipefd[0] for read
    pid_t pid;
    int sum = 0;    // Variable to hold the sum

    // Check if the number of arguments is valid (at least 1 nu
    if (argc < 2) {
        printf("Usage: %s <number1> <number2> ... <numberN>\n",
        return -1;
    }

    // Create the pipe
    if (pipe(pipefd) == -1) {
        perror("pipe");
```

```
        exit(EXIT_FAILURE);
    }

    // Fork a new process
    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {  // Child process
        close(pipefd[0]);  // Close unused read end of the pipe

        // Compute the sum of the command-line arguments
        for (int i = 1; i < argc; i++) {
            sum += atoi(argv[i]);  // Convert the string argumer
        }

        // Write the sum to the pipe
        write(pipefd[1], &sum, sizeof(sum));
        close(pipefd[1]);  // Close the write end after writing

        exit(EXIT_SUCCESS);  // Exit child process
    } else {  // Parent process
        close(pipefd[1]);  // Close unused write end of the pipe

        // Wait for the child process to finish
        wait(NULL);

        // Read the sum from the pipe
        read(pipefd[0], &sum, sizeof(sum));
        close(pipefd[0]);  // Close the read end after reading

        // Print the sum
        printf("The sum is: %d\n", sum);
    }
```

```
    return 0;
}
```

```
roshni@Roshnis-MacBook-Air Assignment_4 % ./part2 1
The sum is: 1
roshni@Roshnis-MacBook-Air Assignment_4 % ./part2 1 2
The sum is: 3
roshni@Roshnis-MacBook-Air Assignment_4 % ./part2 1 2 3
The sum is: 6
roshni@Roshnis-MacBook-Air Assignment_4 %
roshni@Roshnis-MacBook-Air Assignment_4 % ./part2 1 2 3 100
The sum is: 106
roshni@Roshnis-MacBook-Air Assignment_4 %
```

Part 3: What PCB information is retained on Exec:

a)

***Test PID and PPID*:**

testMyExec.c prints the **PID** and **PPID** before executing exec. After the exec call, the newly loaded program **myExec.c** prints its own **PID** and **PPID**. Since exec replaces the process image without creating a new process, both **PID** and **PPID** should remain unchanged.

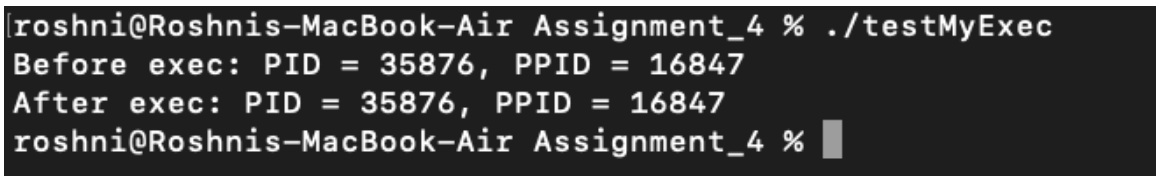
testMyExec.c

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

int main() {
    // Print the PID and PPID before exec
    printf("Before exec: PID = %d, PPID = %d\n", getpid(), getp

    // Call execlp to replace the process with myExec
    execlp("./myExec", "myExec", NULL);
```

```
    // If exec fails
    perror("execlp failed");
    return 1;
}
```

myExec.c:

```
#include <stdio.h>

extern char **environ;  // Declare the environment variables

int main() {
    // Print the environment variable after exec
    printf("After exec: USER=%s\n", environ[1]);
    return 0;
}
```



b) *Test open file descriptors*:

testMyExec.c opens a file and writes the **PID** and **PPID** to it, then calls exec. After exec, myExec.c writes its own **PID** and **PPID** to the same file. The purpose is to check whether the file descriptor remains open after the exec call.

testMyExec.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
```

```
    // Open a file to write
    int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0
    if (fd < 0) {
        perror("Failed to open file");
        return 1;
    }

    // Write the PID and PPID before exec to the file
    dprintf(fd, "Before exec: PID = %d, PPID = %d\n", getpid(),

    // Call execlp to replace the process with myExec
    execlp("./myExec", "myExec", NULL);

    // If exec fails
    perror("execlp failed");
    return 1;
}
```

myExec.c:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    // Open the file in append mode
    int fd = open("output.txt", O_WRONLY | O_APPEND);
    if (fd < 0) {
        perror("Failed to open file");
        return 1;
    }

    // Write the PID and PPID after exec to the file
    dprintf(fd, "After exec: PID = %d, PPID = %d\n", getpid
(), getppid());
```

```
    // Close the file
    close(fd);
    return 0;
}
```

output:

***c) Test environment variables*:**

testMyExec.c modifies an environment variable, which can be set to your actual first and last name, and prints all available environment variables before calling execle. **myExec.c** then prints all environment variables after the exec call to confirm they remain intact. The expected output will show all environment variables both before and after executing **myExec**, reflecting your system's actual environment details. This verifies that the environment variables persist across the exec call.

testMyExec.c

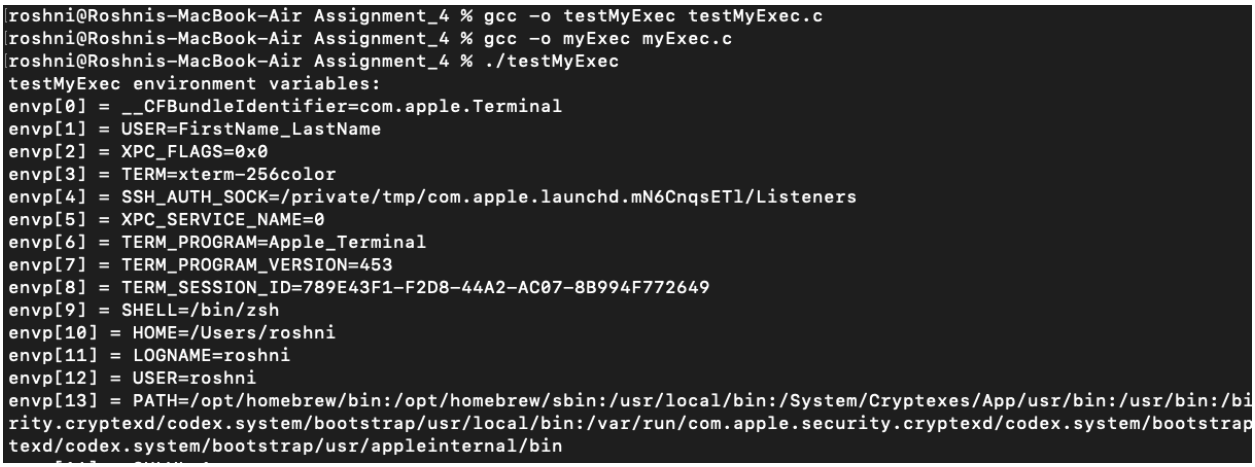
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[], char *envp[]) {
    // Modify an environment variable
    envp[1] = "USER=FirstName_LastName";  // Replace with your a

    // Print environment variables
    printf("testMyExec environment variables:\n");
    for (int i = 0; envp[i] != NULL; i++) {
        printf("envp[%d] = %s\n", i, envp[i]);
```

```
    }

    // Call execle to replace the process with myExec
    execle("./myExec", "myExec", NULL, envp);
    perror("execle failed");
    return 1;
}
```

myExec.c:

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

int main(int argc, char *argv[], char *envp[]) {
    // Print environment variables
    printf("myExec environment variables:\n");
    for (int i = 0; envp[i] != NULL; i++) {
        printf("envp[%d] = %s\n", i, envp[i]);
    }
    return 0;
}
```

output:



4. Combined Fork and Exec:

The **ForkExec** function uses fork() to create a new process, with the child executing a specified command using execvp(). If execvp fails, the child prints an error message and exits. The parent waits for the child to finish using waitpid() and returns the child's PID. The **main** function checks for command-line arguments, calls **ForkExec** with the provided commands, and prints the child's PID upon completion.

main.c:

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

int ForkExec(char *temp[]) {
    pid_t pid = fork();  // Create a new process

    if (pid < 0) {  // Check for fork failure
        perror("fork failed");
        return -1;  // Return -1 on failure
    } else if (pid == 0) {  // Child process
        // Execute the command using execvp
        execvp(temp[0], temp);
        // If execvp fails
        perror("execvp failed");
        exit(EXIT_FAILURE);  // Exit child process if exec fails
    } else {  // Parent process
        int status;
        waitpid(pid, &status, 0);  // Wait for the child to finish
        return pid;  // Return the child's PID
    }
}
```

```
int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: ForkExec [cmd] ... (at least 1 command)\n
        return -1;  // Exit if no command is provided
    }

    int workProcessID = ForkExec(argv + 1);  // Call ForkExec wi
    printf("Child process (PID=%d), performed the task!\n", worl
    return 0;  // Exit successfully
}
```

output:

```
roshni@Roshnis-MacBook-Air Assignment_4 % gcc -o main main.c
roshni@Roshnis-MacBook-Air Assignment_4 % ./main
Usage: ForkExec [cmd] ... (at least 1 command)
roshni@Roshnis-MacBook-Air Assignment_4 % ./main ls
Part1.c         main.c          myExec.c        part1           part2.c         testMyExec.c
main            myExec          output.txt      part2           testMyExec
Child process (PID=45995), performed the task!
roshni@Roshnis-MacBook-Air Assignment_4 % ./main ps
  PID TTY           TIME CMD
16847 ttys000    0:00.79 -zsh
45999 ttys000    0:00.00 ./main ps
22657 ttys001    0:00.11 -zsh
Child process (PID=46000), performed the task!
roshni@Roshnis-MacBook-Air Assignment_4 %
```