Assignment 4

1. Design a C program named complexProcessComm.c that showcases advanced process management and inter-process communication (IPC) using both pipes and shared memory. In this program, the parent process will fork three child processes to perform sequential tasks. The first child process generates a list of 10 random integers and sends this list through a pipe to the second child process. The second child receives the list, calculates its average, and sends the average value through another pipe to the third child process. The third child reads the average, updates a global counter in shared memory, and prints both the average value and the updated counter. The parent process should print its own process ID (PID) and the PIDs of the three child processes, then terminate, allowing the child processes to complete their tasks. Ensure that each child process performs its task in the correct sequence, demonstrating effective use of both pipes for communication and shared memory for synchronization and data sharing. The final output of the third child process should show the calculated average and the incremented global counter value.

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <time.h>

#define NUM_INTEGERS 10
#define SHM_KEY 1234
#define SEM_KEY 5678

void generate_numbers(int pipe_fd[]);
void calculate_average(int input_pipe[], int output_pipe[]);
void update_counter(int input_pipe[], int shm_id, int sem_id);

int main() {
    int pipe1[2], pipe2[2];
    int shm_id, sem_id;
    pid_t pid1, pid2, pid3;
```

```
    if (pipe(pipe1) == -1 || pipe(pipe2) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    shm_id = shmget(SHM_KEY, sizeof(int), IPC_CREAT | 0666);
    if (shm_id == -1) {
        perror("shmget");
        exit(EXIT_FAILURE);
    }

    sem_id = semget(SEM_KEY, 1, IPC_CREAT | 0666);
    if (sem_id == -1) {
        perror("semget");
        exit(EXIT_FAILURE);
    }
    semctl(sem_id, 0, SETVAL, 1);

    if ((pid1 = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid1 == 0) {
        close(pipe1[0]);
        generate_numbers(pipe1);
        close(pipe1[1]);
        exit(EXIT_SUCCESS);
    }

    if ((pid2 = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid2 == 0) {
        close(pipe1[1]);
        close(pipe2[0]);
        calculate_average(pipe1, pipe2);
        close(pipe1[0]);
        close(pipe2[1]);
        exit(EXIT_SUCCESS);
    }

    if ((pid3 = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
```

```
    }

    if (pid3 == 0) {
        close(pipe1[0]);
        close(pipe1[1]);
        close(pipe2[1]);
        update_counter(pipe2, shm_id, sem_id);
        close(pipe2[0]);
        exit(EXIT_SUCCESS);
    }

    close(pipe1[0]);
    close(pipe1[1]);
    close(pipe2[0]);
    close(pipe2[1]);

    printf("Parent PID: %d\n", getpid());
    printf("Child 1 PID: %d\n", pid1);
    printf("Child 2 PID: %d\n", pid2);
    printf("Child 3 PID: %d\n", pid3);

    wait(NULL);
    wait(NULL);
    wait(NULL);

    shmctl(shm_id, IPC_RMID, NULL);
    semctl(sem_id, 0, IPC_RMID);

    return 0;
}

void generate_numbers(int pipe_fd[]) {
    int numbers[NUM_INTEGERS];
    srand(time(NULL));

    for (int i = 0; i < NUM_INTEGERS; i++) {
        numbers[i] = rand() % 100;
    }

    write(pipe_fd[1], numbers, sizeof(numbers));
}

void calculate_average(int input_pipe[], int output_pipe[]) {
    int numbers[NUM_INTEGERS];
    int sum = 0;
    double average;

    read(input_pipe[0], numbers, sizeof(numbers));
```

```
    for (int i = 0; i < NUM_INTEGERS; i++) {
        sum += numbers[i];
    }

    average = (double)sum / NUM_INTEGERS;
    write(output_pipe[1], &average, sizeof(average));
}

void update_counter(int input_pipe[], int shm_id, int sem_id) {
    double average;
    int *counter;
    struct sembuf sem_op;

    read(input_pipe[0], &average, sizeof(average));

    counter = (int *)shmat(shm_id, NULL, 0);
    if (counter == (int *)-1) {
        perror("shmat");
        exit(EXIT_FAILURE);
    }

    sem_op.sem_num = 0;
    sem_op.sem_op = -1;
    sem_op.sem_flg = 0;
    semop(sem_id, &sem_op, 1);

    printf("Average: %.2f\n", average);

    (*counter)++;
    printf("Updated Counter: %d\n", *counter);

    sem_op.sem_op = 1;
    semop(sem_id, &sem_op, 1);

    shmdt(counter);
}
```

Input

The program does not take explicit user input. Instead, it generates a list of random integers internally. These integers are generated in the first child process and passed to the second child process. The specific values of these random integers will vary with each execution.

Output

The output of the program consists of:

1. **Process IDs (PIDs)**: Printed by the parent process. These are the PIDs of the parent and its child processes.
2. **Average and Counter Values**: Printed by the third child process. This includes:
    - The average value calculated from the random integers.
    - The updated global counter value maintained in shared memory.

```
3. Parent PID: 1234
4. Child 1 PID: 1235
5. Child 2 PID: 1236
6. Child 3 PID: 1237
7. Average: 45.50
8. Updated Counter: 1
```

2. Write a C program named complex_zombie.c that demonstrates process management, inter-process communication (IPC), and data structures. The program should fork two child processes: the first child generates a list of 10 random integers, sends this list through a pipe to the second child, which sorts the list using an efficient algorithm (like quicksort), and measures the sorting time. The parent process prints its PID and the PIDs of the two child processes, then sleeps for 1 minute, allowing the second child to become a zombie. After sleeping, the parent should use the ps command to check and display the zombie state of the second child process. Finally, the parent should wait for the second child to exit and clean up the zombie. Concepts involved include process creation (fork()), IPC with pipes, sorting algorithms, zombie processes, and process state checking using ps.

Code:

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

#define NUM_INTEGERS 10

void quicksort(int *arr, int low, int high);
int partition(int *arr, int low, int high);
void print_array(int *arr, int size);
```

```c
int main() {
    int pipefd[2];
    pid_t pid1, pid2;
    int numbers[NUM_INTEGERS];
    struct timeval start, end;
    double elapsed;

    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    if ((pid1 = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (pid1 == 0) {
        close(pipefd[0]);
        srand(getpid());
        for (int i = 0; i < NUM_INTEGERS; i++) {
            numbers[i] = rand() % 100;
        }
        write(pipefd[1], numbers, sizeof(numbers));
        close(pipefd[1]);
        exit(EXIT_SUCCESS);
    }
    if ((pid2 = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid2 == 0) {
        close(pipefd[1]);
        read(pipefd[0], numbers, sizeof(numbers));
        close(pipefd[0]);
        gettimeofday(&start, NULL);
        quicksort(numbers, 0, NUM_INTEGERS - 1);
        gettimeofday(&end, NULL);
        elapsed = (end.tv_sec - start.tv_sec) * 1.0 + (end.tv_usec -
start.tv_usec) / 1000000.0;
        printf("Second child (PID %d) sorted the array in %.6f seconds\n",
getpid(), elapsed);
        printf("Sorted array: ");
        print_array(numbers, NUM_INTEGERS);
        exit(EXIT_SUCCESS);
    }

    close(pipefd[0]);
```

```
    close(pipefd[1]);

    printf("Parent PID: %d\n", getpid());
    printf("First child PID: %d\n", pid1);
    printf("Second child PID: %d\n", pid2);

    sleep(60);

    printf("Checking the state of the second child process...\n");
    printf("You can run `ps -l | grep %d` to check if the second child is a
zombie.\n", pid2);

    waitpid(pid2, NULL, 0);
    waitpid(pid1, NULL, 0);

    return 0;
}

void quicksort(int *arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}

int partition(int *arr, int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}

void print_array(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
```

```
}
```

Input

The program does not take explicit user input. Instead, it generates random data internally:

- **Random Integers**: The first child process generates a list of 10 random integers between 0 and 99.

Output

```
Parent PID: 1234
First child PID: 1235
Second child PID: 1236
Second child (PID 1236) sorted the array in 0.000123 seconds
Sorted array: 2 5 7 15 23 34 45 56 78 89
Checking the state of the second child process...
You can run `ps -l | grep 1236` to check if the second child is a zombie.
```