Name-Abhi Mehta
Reg No.-221080001

Experiment-04

Aim-Implement the C program for child process and thread process creation to perform the given tasks.

Q.1 Using either a UNIX or a Linux system, write a C program that forks a child process that ultimately becomes a zombie process. This zombie process must remain in the system for at least 10 seconds. Process states can be obtained from the command ps -l.

Q.2 The Collatz conjecture concerns what happens when we take any positive integer n and apply the
following algorithm:
n = n/2 if n is even
=3×n + 1, if n is odd
The conjecture states that when this algorithm is continually applied, all positive integers will eventually reach 1. For example, if n = 35, the sequence is
35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1
Write a C program using the fork() system call that generates this sequence in the child process. The starting number will be provided from the command line. For example, if 8 is passed as a parameter on the command line, the child process will output 8, 4, 2, 1. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence. Have the parent invoke the wait() call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a positive integer is passed on the command line.

Q.3 Q. 2 The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, ...
Formally, it can be
expressed as:
fib0 = 0
fib1 = 1
fibn = fibn-1 + fibn-2
Write a multithreaded program that generates the Fibonacci sequence. This program should work as follows: On the command line, the user will enter the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the

sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, the parent thread will have to wait for the child thread to finish. Use the techniques described in Section 4.4 (Galvin, 9th edition) to meet this
requirement.

Theory-

1. Child Process: A child process is a new process created by an existing process, referred to as the parent process. The child process is a separate instance that inherits certain attributes from its parent. Child processes allows parallel execution of tasks.

2. Parent Process: The parent process is an existing process that creates and manages child processes. It may communicate with its children, coordinate their actions, and wait for their termination. Parent processes facilitate the creation of new processes, manage their execution, and handle their termination to achieve specific tasks Efficiently.

3. Zombie Process: A zombie process is a terminated child process that still has an entry in the process table. It remains in a "zombie" state until the parent process acknowledges its termination. Zombie processes represent terminated child processes awaiting confirmation from the parent.

4.Forking: In operating systems, "forking" refers to the process creation mechanism where a new process is created by duplicating the existing process. The new process, known as the child process, is an exact copy of the parent process. After forking, both the parent and the child processes continue to execute from the point of the fork system call, but they have separate memory spaces.

5. Threading: Threading involves dividing a program into multiple concurrent threads of execution. Threads share the same memory space and resources within a process. Threading allows for parallel execution of tasks, improving program responsiveness and enabling efficient use of system resources.

Q1.Program-

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

int main() {
    pid_t child_pid;

    // Fork a child process
    child_pid = fork();

    if (child_pid < 0) {
        // Error occurred
        perror("Fork failed");
        return 1;
    } else if (child_pid == 0) {
        // Child process
        printf("Child process created, pid: %d\n", getpid());
        // Child process exits immediately without waiting
        exit(0);
    } else {
        // Parent process
        printf("Parent process, child pid: %d\n", child_pid);
        sleep(100); // Parent process waits for 100 seconds
        // Parent process exits
        return 0;
    }
}
```

In terminal-

```
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$ cat zombie_program.
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t child_pid;

    // Fork a child process
    child_pid = fork();

    if (child_pid < 0) {
        // Error occurred
        perror("Fork failed");
        return 1;
    } else if (child_pid == 0) {
        // Child process
        printf("Child process created, pid: %d\n", getpid());
        // Child process exits immediately without waiting
        exit(0);
    } else {
        // Parent process
        printf("Parent process, child pid: %d\n", child_pid);
        sleep(100); // Parent process waits for 100 seconds
        // Parent process exits
        return 0;
    }
}
```

Output-

```
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$ gcc -o zombie_program zombie_program.c
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$ ./zombie_program &
[1] 5284
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$ Parent process, child pid: 5285
Child process created, pid: 5285
ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    5246    5228  0  80   0 -  2851 do_wai pts/0    00:00:00 bash
0 S  1000    5284    5246  0  80   0 -   694 hrtime pts/0    00:00:00 zombie_program
1 Z  1000    5285    5284  0  80   0 -     0 -      pts/0    00:00:00 zombie_program <defunct>
4 R  1000    5289    5246  0  80   0 -  3168 -      pts/0    00:00:00 ps
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$ kill 5284
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    5246    5228  0  80   0 -  2851 do_wai pts/0    00:00:00 bash
4 R  1000    5290    5246  0  80   0 -  3168 -      pts/0    00:00:00 ps
[1]+  Terminated              ./zombie_program
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$
```

Q2.Program-

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

void collatz_sequence(int n) {
    while (n != 1) {
        printf("%d ", n);
        if (n % 2 == 0)
            n = n / 2;
        else
            n = 3 * n + 1;
    }
    printf("1\n");
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <positive_integer>\n", argv[0]);
        return 1;
    }

    int start_num = atoi(argv[1]);
    if (start_num <= 0) {
        fprintf(stderr, "Error: Please provide a positive integer.\n");
        return 1;
    }

    pid_t child_pid = fork();

    if (child_pid < 0) {
        // Fork failed
        perror("Fork failed");
        return 1;
    } else if (child_pid == 0) {
        // Child process
        collatz_sequence(start_num);
    } else {
        // Parent process
        wait(NULL); // Wait for child to complete
    }

    return 0;
}
```

In terminal-

```
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$ cat  collatz.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

void collatz_sequence(int n) {
    while (n != 1) {
        printf("%d ", n);
        if (n % 2 == 0)
            n = n / 2;
        else
            n = 3 * n + 1;
    }
    printf("1\n");
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <positive_integer>\n", argv[0]);
        return 1;
    }

    int start_num = atoi(argv[1]);
    if (start_num <= 0) {
        fprintf(stderr, "Error: Please provide a positive integer.\n");
        return 1;
    }

    pid_t child_pid = fork();

    if (child_pid < 0) {
        // Fork failed
        perror("Fork failed");
        return 1;
    } else if (child_pid == 0) {
        // Child process
        collatz_sequence(start_num);
    } else {
        // Parent process
        wait(NULL); // Wait for child to complete
    }

    return 0;
}
```

Output-

```
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$ gcc -o collatz collatz.c
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$ ./collatz 35
35 106 53 160 80 40 20 10 5 16 8 4 2 1
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$
```

Q3.Program-

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

#define MAX_SEQUENCE 1000 // Maximum number of Fibonacci numbers to generate

int sequence[MAX_SEQUENCE];

void *generate_fibonacci(void *arg) {
    int n = *((int *)arg);

    sequence[0] = 0;
    sequence[1] = 1;

    for (int i = 2; i < n; i++) {
        sequence[i] = sequence[i - 1] + sequence[i - 2];
    }

    pthread_exit(0);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <number_of_fibonacci_numbers>\n", argv[0]);
        return 1;
    }

    int num_fibonacci = atoi(argv[1]);
    if (num_fibonacci <= 0 || num_fibonacci > MAX_SEQUENCE) {
        fprintf(stderr, "Error: Please enter a positive integer less than or equal to %d\n", MAX_SEQUENCE);
        return 1;
    }

    pthread_t thread;
    int rc;

    rc = pthread_create(&thread, NULL, generate_fibonacci, (void *)&num_fibonacci);
    if (rc) {
        fprintf(stderr, "Error: Unable to create thread\n");
        return 1;
    }

    pthread_join(thread, NULL); // Wait for the thread to finish
```

```c
    printf("Fibonacci sequence:\n");
    for (int i = 0; i < num_fibonacci; i++) {
        printf("%d ", sequence[i]);
    }
    printf("\n");

    return 0;
}
```

In Terminal-

```
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$ cat >  fibonacci.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define MAX_SEQUENCE 1000 // Maximum number of Fibonacci numbers to generate

int sequence[MAX_SEQUENCE]; // Shared data structure to store the Fibonacci sequence

void *generate_fibonacci(void *arg) {
    int n = *((int *)arg);

    sequence[0] = 0;
    sequence[1] = 1;

    for (int i = 2; i < n; i++) {
        sequence[i] = sequence[i - 1] + sequence[i - 2];
    }

    pthread_exit(0);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <number_of_fibonacci_numbers>\n", argv[0]);
        return 1;
    }

    int num_fibonacci = atoi(argv[1]);
    if (num_fibonacci <= 0 || num_fibonacci > MAX_SEQUENCE) {
        fprintf(stderr, "Error: Please enter a positive integer less than or equal to %d\n", MAX_SEQUENCE);
        return 1;
    }

    pthread_t thread;
    int rc;

    rc = pthread_create(&thread, NULL, generate_fibonacci, (void *)&num_fibonacci);
    if (rc) {
        fprintf(stderr, "Error: Unable to create thread\n");
        return 1;
```

```
    pthread_t thread;
    int rc;

    rc = pthread_create(&thread, NULL, generate_fibonacci, (void *)&num_fibonacci);
    if (rc) {
        fprintf(stderr, "Error: Unable to create thread\n");
        return 1;
    }

    pthread_join(thread, NULL); // Wait for the thread to finish

    printf("Fibonacci sequence:\n");
    for (int i = 0; i < num_fibonacci; i++) {
        printf("%d ", sequence[i]);
    }
    printf("\n");

    return 0;
}
```

Output-

```
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$ gcc -o fibonacci fibonacci.c
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$ ./fibonacci 7
Fibonacci sequence:
0 1 1 2 3 5 8
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$
```

Conclusion-

The experiment effectively demonstrated the management of parent-child processes, including handling of zombie processes, alongside threading for parallel computation tasks like generating the Fibonacci sequence. Through forking, child processes were created to perform independent tasks such as generating the Collatz sequence, with careful handling of zombie processes to ensure efficient resource management. Threading was utilized to parallelize the computation of the Fibonacci sequence within the same process, showcasing the lightweight and scalable nature of threading for concurrent tasks.This approach provided comprehensive insights into managing concurrency, including the advantages of leveraging parallelism for improved performance and resource utilization, alongside challenges such as contention for shared resources. Overall, the experiment offered valuable hands-on experience and analysis, shedding light on optimizing concurrency for real-world applications.