OPERATING SYSTEMS LAB
DIGITAL ASSIGNMENT-3

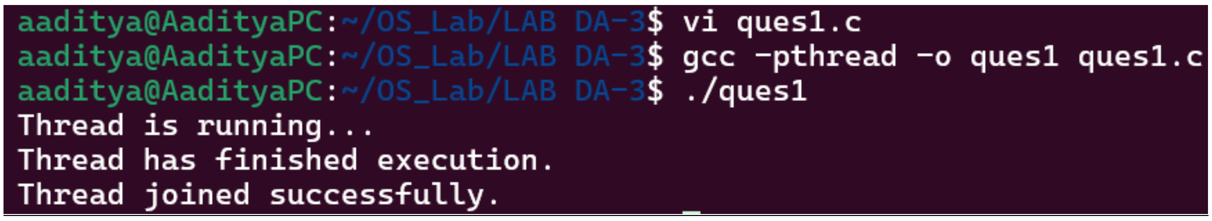
AADITYA ROSHAN
22BIT0250

1. **Write a program to create a thread and perform the following :-**
    - **Create a thread runner function**
    - **Set thread attributes**
    - **Join the parent and thread**
    - **Wait for thread to complete.**

CODE:-

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

void* threadRunner(void* arg) {
    printf("Thread is running...\n");
    sleep(1);
    printf("Thread has finished execution.\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_attr_t attr;

    pthread_attr_init(&attr);

    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    if (pthread_create(&thread, &attr, threadRunner, NULL) != 0) {
        perror("Failed to create thread");
        return 1;
    }

    pthread_attr_destroy(&attr);

    if (pthread_join(thread, NULL) != 0) {
        perror("Failed to join thread");
        return 1;
    }

    printf("Thread joined successfully.\n");

    return 0;
}
```

OUTPUT:-



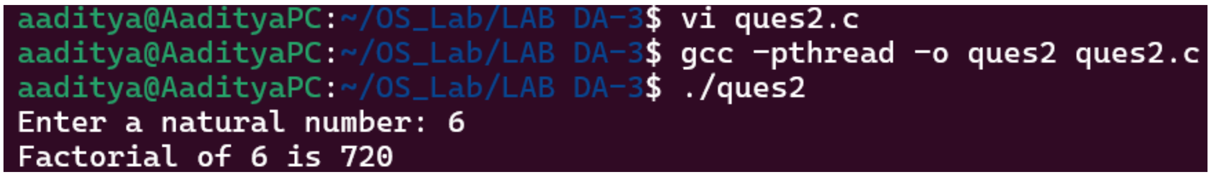
```
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ vi ques1.c
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ gcc -pthread -o ques1 ques1.c
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ ./ques1
Thread is running...
Thread has finished execution.
Thread joined successfully.
```

2. **Write a program to create a thread to find the factorial of a natural number 'n'.**

   CODE:-

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


typedef struct {
    int number;
    unsigned long long result;
} FactorialData;

void* factorial(void* arg) {
    FactorialData* data = (FactorialData*)arg;
    int n = data->number;
    data->result = 1;

    for (int i = 1; i <= n; i++) {
        data->result *= i;
    }

    return NULL;
}

int main() {
    pthread_t thread;
    FactorialData data;

    printf("Enter a natural number: ");
    scanf("%d", &data.number);

    if (data.number < 0) {
        printf("Factorial is not defined for negative numbers.\n");
        return 1;
    }

    if (pthread_create(&thread, NULL, factorial, &data) != 0) {
        perror("Failed to create thread");
        return 1;
    }
```

OUTPUT:-



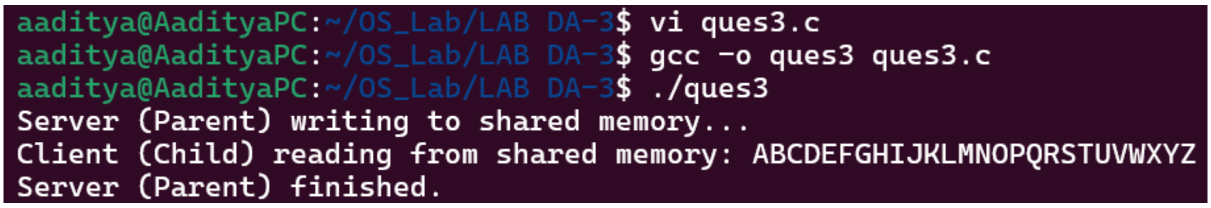
```
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ vi ques2.c
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ gcc -pthread -o ques2 ques2.c
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ ./ques2
Enter a natural number: 6
Factorial of 6 is 720
```

3. **Assume that two processes named client and server running in the system. It is required that these two processes should communicate with each other using shared memory concept. The server writes alphabets from a..z to the shared memory .the client should read the alphabets from the shared memory and convert it to A…Z. Write a program to demonstrate the above mentioned scenario.**

   CODE:-

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

#define SHM_NAME "/shm_example"
#define SHM_SIZE 27

int main() {
    int shm_fd;
    char *shared_mem;
    shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd, SHM_SIZE);
    shared_mem = (char *)mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
shm_fd, 0);

    pid_t pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    } else if (pid == 0) {
        sleep(1);
        printf("Client (Child) reading from shared memory: ");
        for (int i = 0; i < 26; i++) {
            printf("%c", shared_mem[i] - 32);
        }
        printf("\n");
        exit(0);
    } else {
        printf("Server (Parent) writing to shared memory...\n");
        for (char ch = 'a'; ch <= 'z'; ch++) {
             shared_mem[ch - 'a'] = ch;
        }
        shared_mem[26] = '\0';

        wait(NULL);

        munmap(shared_mem, SHM_SIZE);
        close(shm_fd);
        shm_unlink(SHM_NAME);

        printf("Server (Parent) finished.\n");
    }
    return 0;
}
```

OUTPUT:-



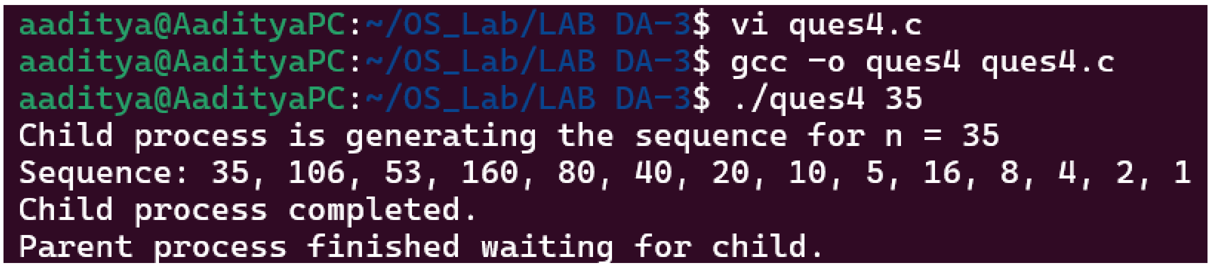
```
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ vi ques3.c
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ gcc -o ques3 ques3.c
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ ./ques3
Server (Parent) writing to shared memory...
Client (Child) reading from shared memory: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Server (Parent) finished.
```

4. **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.**

CODE:-

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
void generate_sequence(int n) {
    printf("Sequence: ");
    while (n != 1) {
        printf("%d, ", n);
        if (n % 2 == 0) {
            n /= 2;
        } else {
            n = 3 * n + 1;
        }
    }
    printf("1\n");
}
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <starting number>\n", argv[0]);
        return 1;
    }
    int n = atoi(argv[1]);
    if (n <= 0) {
        fprintf(stderr, "Please enter a positive integer.\n");
        return 1;}
    pid_t pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        return 1;
    } else if (pid == 0) {
       printf("Child process is generating the sequence for n=%d\n",n);
        generate_sequence(n);
        printf("Child process completed.\n");
        exit(0);
    } else {
        wait(NULL);
        printf("Parent process finished waiting for child.\n");}
    return 0;
}
```

OUTPUT:-



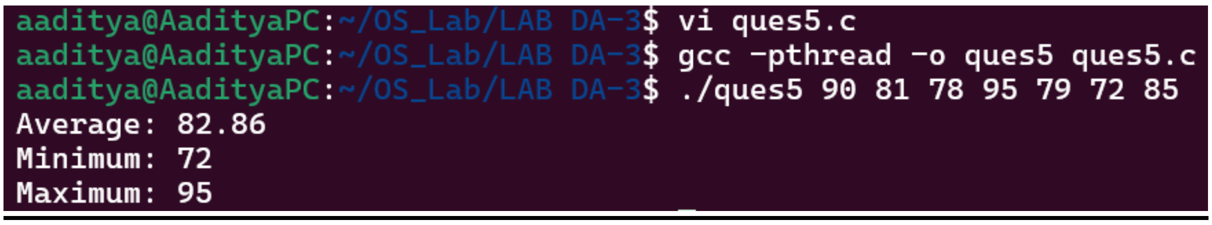
```
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ vi ques4.c
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ gcc -o ques4 ques4.c
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ ./ques4 35
Child process is generating the sequence for n = 35
Sequence: 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1
Child process completed.
Parent process finished waiting for child.
```

5. **Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers 90 81 78 95 79 72 85 , the program will report the average value as 82. The minimum value as 72. The maximum value as 95. The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited.**

   <u>CODE:-</u>

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
double average = 0;
int minimum = 0;
int maximum = 0;
void* calculate_average(void* arg) {
    int* numbers = (int*)arg;
    int sum = 0;
    int count = numbers[0];

    for (int i = 1; i <= count; i++) {
        sum += numbers[i];
    }
    average = (double)sum / count;
    pthread_exit(0);
}
void* calculate_minimum(void* arg) {
    int* numbers = (int*)arg;
    minimum = numbers[1];
    int count = numbers[0];
    for (int i = 2; i <= count; i++) {
        if (numbers[i] < minimum) {
            minimum = numbers[i];
        }
    }
    pthread_exit(0);
}
void* calculate_maximum(void* arg) {
    int* numbers = (int*)arg;
    maximum = numbers[1];
    int count = numbers[0];
for (int i = 2; i <= count; i++) {
        if (numbers[i] > maximum) {
            maximum = numbers[i];
        }
    }
    pthread_exit(0);
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <list of numbers>\n", argv[0]);
        return 1;
    }
    int count = argc - 1;
    int numbers[count + 1];
    numbers[0] = count;
    for (int i = 1; i <= count; i++) {
        numbers[i] = atoi(argv[i]);
    }
    pthread_t thread1, thread2, thread3;
    pthread_create(&thread1, NULL, calculate_average, (void*)numbers);
    pthread_create(&thread2, NULL, calculate_minimum, (void*)numbers);
    pthread_create(&thread3, NULL, calculate_maximum, (void*)numbers);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
    printf("Average: %.2f\n", average);
    printf("Minimum: %d\n", minimum);
    printf("Maximum: %d\n", maximum);

    return 0;
}
```

OUTPUT:-



```
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ vi ques5.c
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ gcc -pthread -o ques5 ques5.c
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ ./ques5 90 81 78 95 79 72 85
Average: 82.86
Minimum: 72
Maximum: 95
```

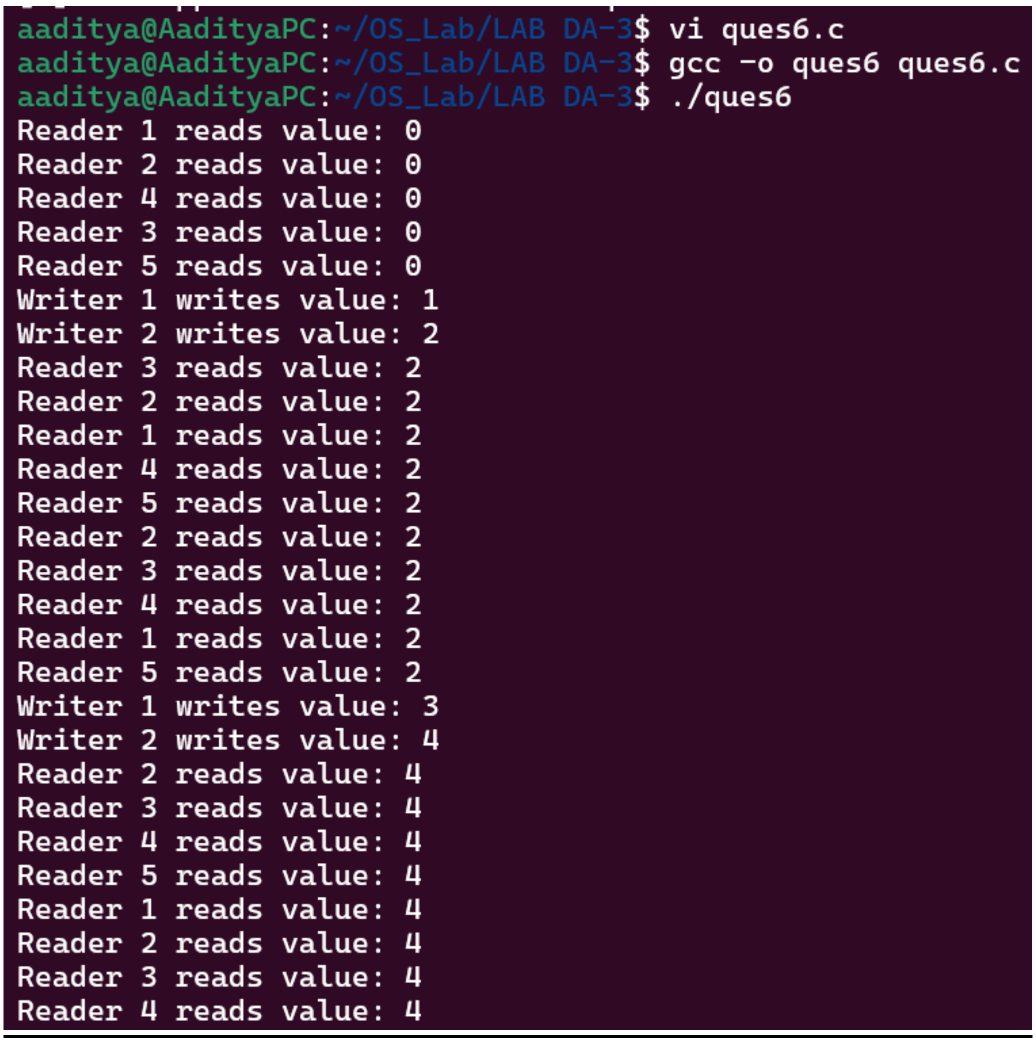
Process Synchronization

6. **Implement the solution for reader - writer's problem.**

   CODE:-

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdbool.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t writelock = PTHREAD_MUTEX_INITIALIZER;
int readers = 0;
int shared_data = 0;
bool running = true;
void* reader(void* arg) {
    int reader_id = *(int*)arg;
    while (running) {
        pthread_mutex_lock(&mutex);
        readers++;
        if (readers == 1) {
            pthread_mutex_lock(&writelock);
        }
        pthread_mutex_unlock(&mutex);
        printf("Reader %d reads value: %d\n", reader_id, shared_data);
        pthread_mutex_lock(&mutex);
        readers--;
        if (readers == 0) {
            pthread_mutex_unlock(&writelock);}
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
    return NULL;
}
void* writer(void* arg) {
    int writer_id = *(int*)arg;
while (running) {
        pthread_mutex_lock(&writelock);
        shared_data++;
        printf("Writer %d writes value: %d\n", writer_id, shared_data);
        pthread_mutex_unlock(&writelock);
        sleep(2);
    }
    return NULL;
}

int main() {
    pthread_t readers_thread[5], writers_thread[2];
    int reader_ids[5], writer_ids[2];
    for (int i = 0; i < 5; i++) {
        reader_ids[i] = i + 1;
        pthread_create(&readers_thread[i], NULL, reader,
&reader_ids[i]);
    }
    for (int i = 0; i < 2; i++) {
        writer_ids[i] = i + 1;
        pthread_create(&writers_thread[i], NULL, writer,
&writer_ids[i]);
    }
    sleep(10);
    running = false;
    for (int i = 0; i < 5; i++) {
        pthread_join(readers_thread[i], NULL);}
    for (int i = 0; i < 2; i++) {
        pthread_join(writers_thread[i], NULL);}
    printf("All threads have finished execution.\n");
    return 0;
}
```

OUTPUT:-

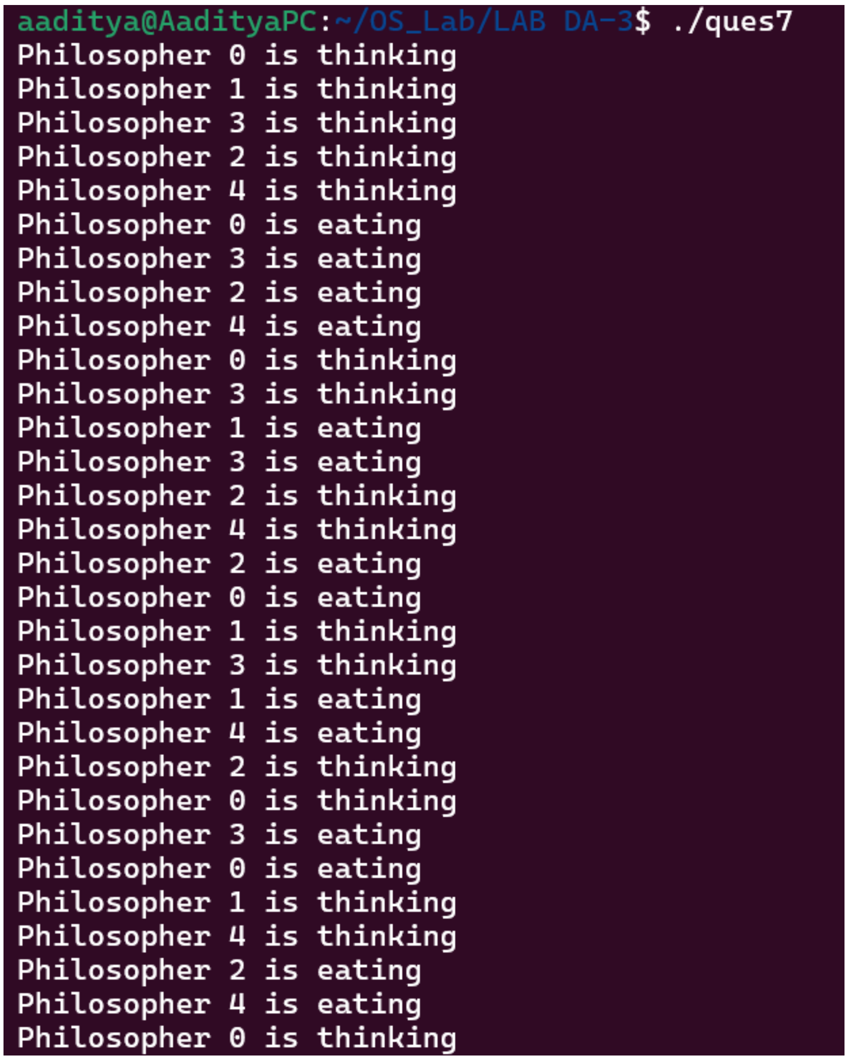


```
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ vi ques6.c
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ gcc -o ques6 ques6.c
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ ./ques6
Reader 1 reads value: 0
Reader 2 reads value: 0
Reader 4 reads value: 0
Reader 3 reads value: 0
Reader 5 reads value: 0
Writer 1 writes value: 1
Writer 2 writes value: 2
Reader 3 reads value: 2
Reader 2 reads value: 2
Reader 1 reads value: 2
Reader 4 reads value: 2
Reader 5 reads value: 2
Reader 2 reads value: 2
Reader 3 reads value: 2
Reader 4 reads value: 2
Reader 1 reads value: 2
Reader 5 reads value: 2
Writer 1 writes value: 3
Writer 2 writes value: 4
Reader 2 reads value: 4
Reader 3 reads value: 4
Reader 4 reads value: 4
Reader 5 reads value: 4
Reader 1 reads value: 4
Reader 2 reads value: 4
Reader 3 reads value: 4
Reader 4 reads value: 4
```

7. **Implement the solution for dining philosopher's problem.**

CODE:-
```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t forks[5];
void* philosopher(void* arg) {
    int phil_id = *((int*)arg);
    while (1) {
        printf("Philosopher %d is thinking\n", phil_id);
        sleep(1);
sem_wait(&forks[phil_id]);
        sem_wait(&forks[(phil_id + 1) % 5]);
        printf("Philosopher %d is eating\n", phil_id);
        sleep(2);
sem_post(&forks[phil_id]);
        sem_post(&forks[(phil_id + 1) % 5]);
sleep(1);
    }
}
int main() {
pthread_t phil[5];
int phil_ids[5] = {0, 1, 2, 3, 4};
for (int i = 0; i < 5; i++)
        sem_init(&forks[i], 0, 1);
for (int i = 0; i < 5; i++)
        pthread_create(&phil[i], NULL, philosopher, &phil_ids[i]);
for (int i = 0; i < 5; i++)
pthread_join(phil[i], NULL);
for (int i = 0; i < 5; i++)
sem_destroy(&forks[i]);
    return 0;
}
```

OUTPUT:-



```
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ ./ques7
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 3 is thinking
Philosopher 2 is thinking
Philosopher 4 is thinking
Philosopher 0 is eating
Philosopher 3 is eating
Philosopher 2 is eating
Philosopher 4 is eating
Philosopher 0 is thinking
Philosopher 3 is thinking
Philosopher 1 is eating
Philosopher 3 is eating
Philosopher 2 is thinking
Philosopher 4 is thinking
Philosopher 2 is eating
Philosopher 0 is eating
Philosopher 1 is thinking
Philosopher 3 is thinking
Philosopher 1 is eating
Philosopher 4 is eating
Philosopher 2 is thinking
Philosopher 0 is thinking
Philosopher 3 is eating
Philosopher 0 is eating
Philosopher 1 is thinking
Philosopher 4 is thinking
Philosopher 2 is eating
Philosopher 4 is eating
Philosopher 0 is thinking
```

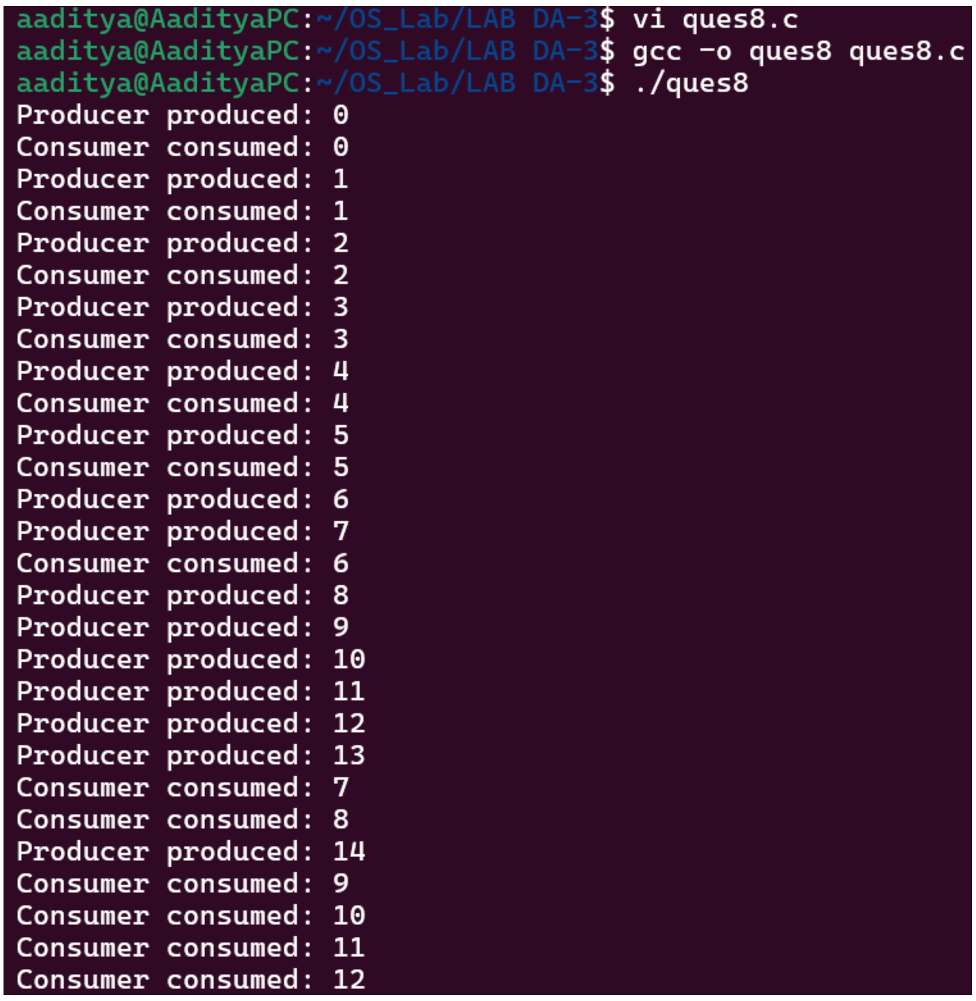
8. **A pair of processes involved in exchanging a sequence of integers. The number of integers that can be produced and consumed at a time is limited to 100. Write a Program to implement the producer and consumer problem using POSIX semaphore for the above scenario.**

CODE:-

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

#define BUFFER_SIZE 100
int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
sem_t empty;
sem_t full;
pthread_mutex_t mutex;


void* producer(void* arg) {
    for (int i = 0; i < 100; i++) {
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        buffer[in] = i;
        printf("Producer produced: %d\n", buffer[in]);
        in = (in + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
sleep(rand() % 2);
    }
return NULL;
}


void* consumer(void* arg) {
for (int i = 0; i < 100; i++) {
        sem_wait(&full);
pthread_mutex_lock(&mutex);
        int value = buffer[out];
        printf("Consumer consumed: %d\n", value);
out = (out + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
sleep(rand() % 2);
    }
    return NULL;
}
```

```c
int main() {
    pthread_t producer_thread, consumer_thread;
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);
    printf("Program finished successfully.\n");
    return 0;
}
```

OUTPUT:-



```
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ vi ques8.c
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ gcc -o ques8 ques8.c
aaditya@AadityaPC:~/OS_Lab/LAB DA-3$ ./ques8
Producer produced: 0
Consumer consumed: 0
Producer produced: 1
Consumer consumed: 1
Producer produced: 2
Consumer consumed: 2
Producer produced: 3
Consumer consumed: 3
Producer produced: 4
Consumer consumed: 4
Producer produced: 5
Consumer consumed: 5
Producer produced: 6
Producer produced: 7
Consumer consumed: 6
Producer produced: 8
Producer produced: 9
Producer produced: 10
Producer produced: 11
Producer produced: 12
Producer produced: 13
Consumer consumed: 7
Consumer consumed: 8
Producer produced: 14
Consumer consumed: 9
Consumer consumed: 10
Consumer consumed: 11
Consumer consumed: 12
```