Assigment No 1

```
#!/bin/bash
address_book="address_book.txt"

create() {
   if [ ! -f "$address_book" ]; then
      touch "$address_book"
      echo "Address book created."
   else
      echo "Address book already exists."
   fi
}

display() {
   if [ -s "$address_book" ]; then
      cat "$address_book"
   else
      echo "Address book is empty."
   fi
}

insert() {
   echo -n "Enter name: "
   read name
   echo -n "Enter phone: "
   read phone
   echo -n "Enter email: "
   read email
   echo "$name,$phone,$email" >> "$address_book"
```

```
    echo "Record added."
}

delete() {
    echo -n "Enter name to delete: "
    read name
    if grep -q "^$name," "$address_book"; then
        grep -v "^$name," "$address_book" > temp && mv temp "$address_book"
        echo "Record deleted."
    else
        echo "No record found for '$name'."
    fi
}

modify() {
    echo -n "Enter name of the record to modify: "
    read name
    if grep -q "^$name," "$address_book"; then
        old_record=$(grep "^$name," "$address_book")
        grep -v "^$name," "$address_book" > temp && mv temp "$address_book"
        echo "Enter updated details for $name:"
        echo -n "Enter new name: "
        read new_name
        echo -n "Enter phone: "
        read phone
        echo -n "Enter email: "
        read email
        echo "$new_name,$phone,$email" >> "$address_book"
        echo "Record modified."
```

```
    else
        echo "No record found for '$name'."
    fi
}

while true; do
    echo "1. Create Address Book"
    echo "2. Display Address Book"
    echo "3. Insert Record"
    echo "4. Delete Record"
    echo "5. Modify Record"
    echo "6. Exit"
    echo -n "Choose an option: "
    read option
    case $option in
        1) create ;;
        2) display ;;
        3) insert ;;
        4) delete ;;
        5) modify ;;
        6) echo "Goodbye!"; exit ;;
        *) echo "Invalid option!" ;;
    esac
done
```

Assigment No 2

Binary_search.c

```
// binary_search.c
#include <stdio.h>
#include <stdlib.h>
```

```
int binary_search(int arr[], int size, int target) {
    int low = 0, high = size - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage: %s <sorted_array> <target>\n", argv[0]);
        return 1;
    }

    int size = argc - 2;  // Exclude program name and target
    int arr[size];
    for (int i = 1; i <= size; i++) {
        arr[i - 1] = atoi(argv[i]);
    }
    int target = atoi(argv[argc - 1]);

    int result = binary_search(arr, size, target);
    if (result != -1) {
        printf("Target %d found at index %d.\n", target, result);
    } else {
```

```
        printf("Target %d not found.\n", target);
    }

    return 0;
}
```

Main.c

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

// Function to sort an array
void sort_array(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = i + 1; j < size; j++) {
            if (arr[i] > arr[j]) {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}

int main() {
    int arr[] = {34, 7, 23, 32, 5, 62};
    int size = sizeof(arr) / sizeof(arr[0]);
```

```
int target = 23;

// Sort the array
sort_array(arr, size);
printf("Sorted array: ");
for (int i = 0; i < size; i++) {
    printf("%d ", arr[i]);
}
printf("\n");

pid_t pid = fork();

if (pid < 0) {
    perror("fork failed");
    exit(1);
} else if (pid == 0) {
    // Child process
    char *args[size + 3];
    args[0] = "./binary_search";  // Program to execute

    // Convert integers to strings for command-line arguments
    for (int i = 0; i < size; i++) {
        args[i + 1] = malloc(10);
        snprintf(args[i + 1], 10, "%d", arr[i]);
    }

    // Pass the target as the last argument
    args[size + 1] = malloc(10);
    snprintf(args[size + 1], 10, "%d", target);
```

```
        args[size + 2] = NULL;

        execve("./binary_search", args, NULL);

        // Free memory if execve fails
        for (int i = 1; i <= size + 1; i++) {
            free(args[i]);
        }

        // If execve fails
        perror("execve failed");
        exit(1);
    } else {
        // Parent process
        printf("Parent waiting for child to complete...\n");

        // Wait for child process to avoid a zombie state
        wait(NULL);

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

    return 0;
}
```

Assigment No 3

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

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

#define SIZE 10

int A[SIZE][SIZE], B[SIZE][SIZE];
long int C[SIZE][SIZE];

void *mul_thread(void *arg) {
   int *data = (int *)arg;
   int row = data[0];
   int col = data[1];
   int k = data[2];
   long int *result = (long int *)malloc(sizeof(long int));  // Allocate memory for the result
   *result = A[row][k] * B[k][col];  // Calculate the product of the corresponding elements
   pthread_exit((void *)result);  // Return the result
}

void accept_matrix(int M[SIZE][SIZE], int rows, int cols) {
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         printf("Value at [%d][%d]: ", i + 1, j + 1);
         scanf("%d", &M[i][j]);
      }
   }
}

void display_matrix(int M[SIZE][SIZE], int rows, int cols) {
   for (int i = 0; i < rows; i++) {
```

```
        for (int j = 0; j < cols; j++) {
            printf("%2d  ", M[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int rows_A, cols_A, rows_B, cols_B;

    printf("\nEnter no. of rows in matrix A: ");
    scanf("%d", &rows_A);
    printf("Enter no. of columns in matrix A: ");
    scanf("%d", &cols_A);
    accept_matrix(A, rows_A, cols_A);

    printf("\nEnter no. of rows in matrix B: ");
    scanf("%d", &rows_B);
    printf("Enter no. of columns in matrix B: ");
    scanf("%d", &cols_B);
    accept_matrix(B, rows_B, cols_B);

    if (cols_A != rows_B) {
        printf("\nMatrix multiplication not possible.\n");
        exit(1);
    }

    pthread_t threads[SIZE][SIZE][SIZE];
    int thread_args[3];
```

```
    long int *result;

    // Creating threads
    for (int i = 0; i < rows_A; i++) {
        for (int j = 0; j < cols_B; j++) {
            C[i][j] = 0;  // Initialize the result matrix
            for (int k = 0; k < cols_A; k++) {
                thread_args[0] = i;
                thread_args[1] = j;
                thread_args[2] = k;
                pthread_create(&threads[i][j][k], NULL, mul_thread, thread_args);
                pthread_join(threads[i][j][k], (void **)&result);  // Collect the result from the thread
                C[i][j] += *result;  // Sum the result in the main thread
                free(result);  // Free the memory allocated for the result
            }
        }
    }

    printf("\nMatrix A:");
    display_matrix(A, rows_A, cols_A);
    printf("\nMatrix B:");
    display_matrix(B, rows_B, cols_B);
    printf("\nResultant Matrix:");
    display_matrix(C, rows_A, cols_B);

    return 0;
}
```

Assigment No 4

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

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

#define BUFFER_SIZE 5
#define NUM_PRODUCERS 2
#define NUM_CONSUMERS 2
#define NUM_ITEMS 10

int buffer[BUFFER_SIZE];
int in = 0; // Index for the next item to be produced
int out = 0; // Index for the next item to be consumed

sem_t empty; // Counting semaphore to count empty slots in the buffer
sem_t full; // Counting semaphore to count full slots in the buffer
pthread_mutex_t mutex; // Mutex for mutual exclusion

void* producer(void* arg) {
    for (int i = 0; i < NUM_ITEMS; i++) {
        // Produce an item
        int item = rand() % 100;

        // Wait for an empty slot
        sem_wait(&empty);

        // Lock the buffer for exclusive access
        pthread_mutex_lock(&mutex);
```

```
        // Add the item to the buffer
        buffer[in] = item;
        printf("Producer %d produced: %d\n", *((int*)arg), item);
        in = (in + 1) % BUFFER_SIZE;

        // Unlock the buffer
        pthread_mutex_unlock(&mutex);

        // Signal that there is a new item
        sem_post(&full);

        sleep(1); // Simulate time taken to produce an item
    }
    return NULL;
}

void* consumer(void* arg) {
    for (int i = 0; i < NUM_ITEMS; i++) {
        // Wait for a full slot
        sem_wait(&full);

        // Lock the buffer for exclusive access
        pthread_mutex_lock(&mutex);

        // Remove an item from the buffer
        int item = buffer[out];
        printf("Consumer %d consumed: %d\n", *((int*)arg), item);
        out = (out + 1) % BUFFER_SIZE;
```

```
        // Unlock the buffer
        pthread_mutex_unlock(&mutex);

        // Signal that there is an empty slot
        sem_post(&empty);

        sleep(1); // Simulate time taken to consume an item
    }
    return NULL;
}

int main() {
    pthread_t producers[NUM_PRODUCERS], consumers[NUM_CONSUMERS];
    int producer_ids[NUM_PRODUCERS], consumer_ids[NUM_CONSUMERS];

    // Initialize semaphores and mutex
    sem_init(&empty, 0, BUFFER_SIZE); // All slots are empty initially
    sem_init(&full, 0, 0); // No items are produced initially
    pthread_mutex_init(&mutex, NULL);

    // Create producer threads
    for (int i = 0; i < NUM_PRODUCERS; i++) {
        producer_ids[i] = i + 1;
        pthread_create(&producers[i], NULL, producer, &producer_ids[i]);
    }

    // Create consumer threads
    for (int i = 0; i < NUM_CONSUMERS; i++) {
        consumer_ids[i] = i + 1;
```

```
        pthread_create(&consumers[i], NULL, consumer, &consumer_ids[i]);
    }

    // Wait for all producer threads to finish
    for (int i = 0; i < NUM_PRODUCERS; i++) {
        pthread_join(producers[i], NULL);
    }

    // Wait for all consumer threads to finish
    for (int i = 0; i < NUM_CONSUMERS; i++) {
        pthread_join(consumers[i], NULL);
    }

    // Clean up
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

Assigment No 5

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

#define NUM_PHILOSOPHERS 5
```

```
sem_t forks[NUM_PHILOSOPHERS]; // Semaphores for each fork
pthread_mutex_t lock;          // Mutex for printing

void* philosopher(void* num) {
   int id = *(int*)num;

   while (1) {
      // Thinking
      printf("Philosopher %d is thinking.\n", id);
      sleep(rand() % 3); // Random thinking time

      // Try to pick up forks
      if (id % 2 == 0) {
         // Pick up left fork first for even indexed philosophers
         sem_wait(&forks[id]);
         sem_wait(&forks[(id + 1) % NUM_PHILOSOPHERS]);
      } else {
         // Pick up right fork first for odd indexed philosophers
         sem_wait(&forks[(id + 1) % NUM_PHILOSOPHERS]);
         sem_wait(&forks[id]);
      }

      // Eating
      pthread_mutex_lock(&lock);
      printf("Philosopher %d is eating.\n", id);
      pthread_mutex_unlock(&lock);
      sleep(rand() % 3); // Random eating time

      // Put down forks
```

```
        sem_post(&forks[id]);
        sem_post(&forks[(id + 1) % NUM_PHILOSOPHERS]);
    }
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_ids[NUM_PHILOSOPHERS];

    // Initialize semaphores
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_init(&forks[i], 0, 1); // Each fork is initially available
    }

    // Initialize mutex for printing
    pthread_mutex_init(&lock, NULL);

    // Create philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopher_ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &philosopher_ids[i]);
    }

    // Join philosopher threads (this will not happen in this example)
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Cleanup (not reached)
```

```
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_destroy(&forks[i]);
    }
    pthread_mutex_destroy(&lock);

    return 0;
}
```

Assigment No 6

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

void main()
{
        FILE *fp=0,*fp1=0;
        int fd1[2],fd2[2],len=0;
        pid_t pid;
      char path[]="/home/dbl/gsm/test.txt";
        if(pipe(fd1) == -1)
        {
                perror("pipe1");
                exit(0);
        }
        if(pipe(fd2) == -1)
        {
                perror("pipe2");
                exit(0);
        }
```

```
pid = fork();
if(pid<0)
{
        perror("Error in spawning fork");
        exit(0);
}
else if(pid>0)
{
        char buffer[200];
        fp = fopen(path,"a+");
        close(fd1[0]);
        len = write(fd1[1],fp,sizeof(fp));
        wait();
        memset(buffer,0,200);
        close(fd2[1]);
        read(fd2[0],fp,sizeof(fp));
        fp = fopen(path,"r");
        fread(buffer,200,1,fp);
    printf("\n*----------------------*/ PARENT /*----------------------*");
        printf("\n In Parent\n\nReading File...\n\n%s",buffer);
        printf("\n");
}
else
{
        char buffer[200];
        close(fd1[1]);
        len = read(fd1[0],buffer,200);
        fp1 = fopen(path,"a+");
```

```
                memset(buffer,0,200);
               fread(buffer,200,1,fp1);
              printf("\n*-----------------------*/ CHILD /*----------------------*");
                printf("\n In Child\n\nRetriving File...\n\n%s",buffer);
                fseek(fp1,0,SEEK_END);
                fprintf(fp1,"Child wrote this....!!!!");
               printf("\n");
                fclose(fp1);
                close(fd2[0]);
                write(fd2[1],fp1,sizeof(fp1));

        }
}
```

Assigment No 7

Client.c

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <string.h>
#define SHM_SIZE 1024  // Size of shared memory segment

int main() {
   // Generate the same key used by the server
   key_t key = ftok("shmfile", 65);

   // Locate the shared memory segment created by the server
   int shmid = shmget(key, SHM_SIZE, 0666);
   if (shmid == -1) {
```

```
        perror("shmget failed");
        exit(1);
    }

    // Attach the shared memory segment to client's address space
    char *shared_memory = (char *)shmat(shmid, NULL, 0);
    if (shared_memory == (char *)-1) {
        perror("shmat failed");
        exit(1);
    }

    // Read the message from shared memory
    printf("Client read from shared memory: %s\n", shared_memory);

    // Set the flag to indicate that the message has been read
    shared_memory[strlen(shared_memory) + 1] = 1;

    // Detach the shared memory segment
    if (shmdt(shared_memory) == -1) {
        perror("shmdt failed");
        exit(1);
    }

    return 0;
}
```

Server.c

```
#include <stdio.h>
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>  // For sleep()

#define SHM_SIZE 1024  // Shared memory size

int main() {
   // Generate a unique key for shared memory
   key_t key = ftok("shmfile", 65);

   // Create shared memory segment
   int shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);
   if (shmid == -1) {
      perror("Failed to create shared memory");
      exit(1);
   }

   // Attach to the shared memory
   char *shared_memory = (char *)shmat(shmid, NULL, 0);
   if (shared_memory == (char *)-1) {
      perror("Failed to attach shared memory");
      exit(1);
   }

   // Write a message into the shared memory
   char message[] = "Hello from server!";
   strcpy(shared_memory, message);
```

```
    // Set a flag after the message
    shared_memory[strlen(message) + 1] = 0;  // Set the flag to 0 (message not yet read)

    printf("Server wrote to shared memory: %s\n", message);

    // Wait until the client reads the message (flag changes to 1)
    while (shared_memory[strlen(message) + 1] == 0) {
        printf("Waiting for client to read the message...\n");
        sleep(1);  // Sleep for 1 second before checking again
    }

    printf("Client has read the message. Server exiting...\n");

    // Detach from shared memory
    if (shmdt(shared_memory) == -1) {
        perror("Failed to detach shared memory");
        exit(1);
    }

    // Optionally, destroy the shared memory segment
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```