#include <stdio.h>        // Standard Input and Output library for file and console I/O

#include <stdlib.h>       // Standard Library for miscellaneous functions like exit()

#include <time.h>         // Time library for accessing system time

#include <semaphore.h>    // Semaphore library for synchronization

#include <pthread.h>      // POSIX Threads library for handling threads

#define NUM\_READERS 5     // Macro defining the number of reader threads

#define NUM\_WRITERS 2     // Macro defining the number of writer threads

#define STRING\_LENGTH 60  // Macro defining the length of strings that writers will produce

pthread\_t readers[NUM\_READERS], writers[NUM\_WRITERS];  // Array of thread identifiers for readers and writers

sem\_t mutex, rw\_mutex;                                // Semaphore identifiers for mutual exclusion and read-write lock

int readers\_count = 0;                                // Counter to track the number of active readers

FILE \*file;                                           // File pointer to the shared file

char generateRandomChar() {

    return (char)('a' + rand() % 26); // Generating a random lowercase character from 'a' to 'z'

}

void \*reader(void \*arg) {

    while (1) {                      // Infinite loop for continuous reading

        sem\_wait(&mutex);            // Lock mutex to modify the readers\_count safely

        readers\_count++;             // Increment the count of active readers

        if (readers\_count == 1) {

            sem\_wait(&rw\_mutex);     // If first reader, also lock the rw\_mutex to block writers

        }

        sem\_post(&mutex);            // Release mutex after modifying count

        fseek(file, 0, SEEK\_SET);    // Set file pointer to the beginning

        char buffer[256];            // Buffer to store file content

        while (fgets(buffer, sizeof(buffer), file) != NULL) {

            fprintf(stdout,"Reader %ld: %s", (long)arg, buffer); // Print each line read from the file

        }

        sem\_wait(&mutex);            // Lock mutex again to safely decrement readers\_count

        readers\_count--;             // Decrement the count of active readers

        if (readers\_count == 0) {

            sem\_post(&rw\_mutex);     // If last reader, release rw\_mutex to allow writers

        }

        sem\_post(&mutex);            // Release mutex

        usleep(1000);                // Sleep for a short duration

    }

}

void \*writer(void \*arg) {

    while (1) {                           // Infinite loop for continuous writing

        sem\_wait(&rw\_mutex);              // Lock rw\_mutex to ensure exclusive access to the file

        srand(time(NULL));                // Initialize random seed

        char randomString[STRING\_LENGTH + 1]; // Buffer for the random string

        for (int i = 0; i < STRING\_LENGTH; i++) {

            randomString[i] = generateRandomChar(); // Generate random string

        }

        randomString[STRING\_LENGTH] = '\0'; // Null-terminate the string

        fseek(file, 0, SEEK\_END);         // Move file pointer to the end for appending

        fprintf(file, "%s\n", randomString); // Write the random string to the file

        fprintf(stdout, "Writer %ld: %s\n", (long)arg, randomString); // Log output to console

        fflush(file);                     // Flush the file buffer to ensure writing is complete

        sem\_post(&rw\_mutex);              // Release rw\_mutex to allow other writers or readers

        usleep(1000);                     // Sleep for a short duration

    }

}

int main() {

    file = fopen("shared\_file.txt", "a+"); // Open or create file for reading and appending

    if (file == NULL) {

        perror("Error opening file");      // Error handling if file cannot be opened

        exit(EXIT\_FAILURE);

    }

    sem\_init(&mutex, 0, 1); sem\_init(&rw\_mutex, 0, 1); // Initialize semaphores

    int i;

    for (i = 0; i < NUM\_WRITERS; i++) pthread\_create(&writers[i], NULL, writer, (void \*)(long)i);

    for (i = 0; i < NUM\_READERS; i++) pthread\_create(&readers[i], NULL, reader, (void \*)(long)i);

    for (i = 0; i < NUM\_READERS; i++) pthread\_join(readers[i], NULL);

    for (i = 0; i < NUM\_WRITERS; i++) pthread\_join(writers[i], NULL);

    fprintf(stdout, "reader pthread join completed\n");

    sem\_destroy(&mutex); sem\_destroy(&rw\_mutex); // Destroy semaphores

    fclose(file);                                // Close the file

    return 0;

}

#include <stdio.h>       // Standard input and output functions

#include <stdlib.h>      // Standard library for functions like rand()

#include <pthread.h>     // POSIX threading library

#include <semaphore.h>   // Semaphore library for synchronization

#define BUFFER\_SIZE 5    // Define the size of the buffer to be used

sem\_t mutex, empty, full;  // Semaphores for managing access to the buffer

int buffer[BUFFER\_SIZE];   // Array to serve as the circular buffer

int in = 0, out = 0;       // Indices for insertion and removal in the buffer

void \*producer(void \*arg) {

    int item;

    while (1) {                              // Infinite loop for continuous production

        item = rand() % 100;                 // Generate a random item (0-99)

        sem\_wait(&empty);                    // Decrement the empty semaphore or block if buffer is full

        sem\_wait(&mutex);                    // Enter critical section by acquiring the mutex

        buffer[in] = item;                   // Place the item in the buffer at the 'in' position

        printf("Produced: %d\n", item);      // Print the produced item

        in = (in + 1) % BUFFER\_SIZE;         // Update 'in' to the next position, wrapping around using modulo

        sem\_post(&mutex);                    // Release the mutex, exiting critical section

        sem\_post(&full);                     // Increment the full semaphore, signaling that buffer has an item

        sleep(rand() % 3);                   // Random sleep to simulate processing time

    }

}

void \*consumer(void \*arg) {

    int item;

    while (1) {                              // Infinite loop for continuous consumption

        sem\_wait(&full);                     // Decrement the full semaphore or block if buffer is empty

        sem\_wait(&mutex);                    // Enter critical section by acquiring the mutex

        item = buffer[out];                  // Remove the item from the buffer at the 'out' position

        printf("Consumed: %d\n", item);      // Print the consumed item

        out = (out + 1) % BUFFER\_SIZE;       // Update 'out' to the next position, wrapping around using modulo

        sem\_post(&mutex);                    // Release the mutex, exiting critical section

        sem\_post(&empty);                    // Increment the empty semaphore, signaling space in buffer

        sleep(rand() % 3);                   // Random sleep to simulate processing time

    }

}

int main() {

    pthread\_t producer\_thread, consumer\_thread;  // Thread identifiers for producer and consumer

    // Initialize semaphores

    sem\_init(&mutex, 0, 1);         // Binary semaphore for mutual exclusion

    sem\_init(&empty, 0, BUFFER\_SIZE); // Counting semaphore, initialized to the size of the buffer

    sem\_init(&full, 0, 0);          // Counting semaphore, initialized to 0 as buffer starts empty

    // Create producer and consumer threads

    pthread\_create(&producer\_thread, NULL, producer, NULL); // Create the producer thread

    pthread\_create(&consumer\_thread, NULL, consumer, NULL); // Create the consumer thread

    // Join threads

    pthread\_join(producer\_thread, NULL); // Wait for the producer thread to terminate

    pthread\_join(consumer\_thread, NULL); // Wait for the consumer thread to terminate

    // Destroy semaphores

    sem\_destroy(&mutex);           // Destroy the mutex semaphore

    sem\_destroy(&empty);           // Destroy the empty semaphore

    sem\_destroy(&full);            // Destroy the full semaphore

    return 0;                      // Exit the program

}