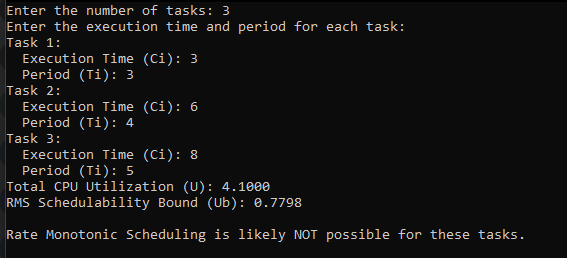
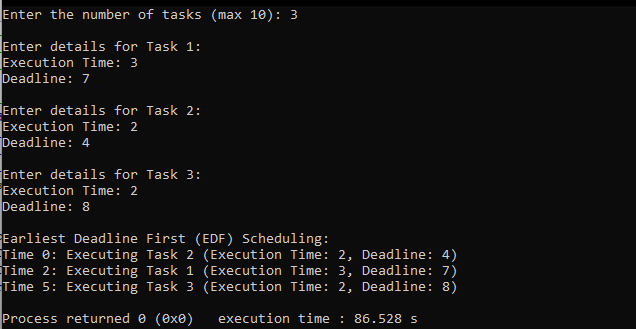
Rate Monotonic

#include <stdio.h>  
#include <stdbool.h>  
#include <math.h>  
#include <stdlib.h>  
  
typedef struct {  
    int id;  
    int execution\_time;  
    int period;  
} Task;  
  
  
bool is\_rms\_schedulable(Task \*tasks, int num\_tasks) {  
    double cpu\_utilization\_bound = num\_tasks \* (pow(2.0, 1.0 / num\_tasks) - 1);  
    double total\_utilization = 0.0;  
  
    for (int i = 0; i < num\_tasks; i++) {  
        if (tasks[i].period <= 0) {  
            printf("Error: Task %d has a non-positive period.\n", tasks[i].id);  
            return false;  
        }  
        total\_utilization += (double)tasks[i].execution\_time / tasks[i].period;  
    }  
  
    printf("Total CPU Utilization (U): %.4f\n", total\_utilization);  
    printf("RMS Schedulability Bound (Ub): %.4f\n", cpu\_utilization\_bound);  
  
    if (total\_utilization <= cpu\_utilization\_bound) {  
        return true;  
    } else {  
        return false;  
    }  
}  
  
int main() {  
    int num\_tasks;  
  
    printf("Enter the number of tasks: ");  
    if (scanf("%d", &num\_tasks) != 1 || num\_tasks <= 0) {  
        printf("Invalid number of tasks.\n");  
        return 1;  
    }  
  
  
    Task \*tasks = (Task \*)malloc(num\_tasks \* sizeof(Task));  
    if (tasks == NULL) {  
        printf("Memory allocation failed.\n");  
        return 1;  
    }



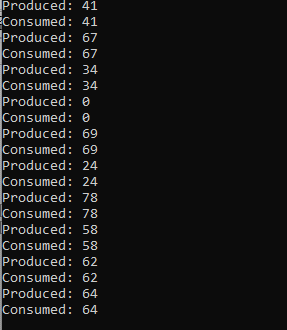
Earliest deadline first

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
#define MAX\_TASKS 10     
  
typedef struct {  
    int id;               
    int execution\_time;   
    int deadline;         
} Task;  
  
int compare\_deadlines(const void \*a, const void \*b) {  
    Task \*taskA = (Task \*)a;  
    Task \*taskB = (Task \*)b;  
    return taskA->deadline - taskB->deadline;  
}  
  
void edf\_schedule(Task tasks[], int n) {  
    int time = 0; // Start time  
  
  
    qsort(tasks, n, sizeof(Task), compare\_deadlines);  
  
    printf("\nEarliest Deadline First (EDF) Scheduling:\n");  
  
     
    for (int i = 0; i < n; i++) {  
  
        if (time < tasks[i].deadline) {  
            printf("Time %d: Executing Task %d (Execution Time: %d, Deadline: %d)\n", time, tasks[i].id, tasks[i].execution\_time, tasks[i].deadline);  
            time += tasks[i].execution\_time; // Increment time by the task's execution time  
        } else {  
            printf("Time %d: Task %d missed its deadline!\n", time, tasks[i].id);  
        }  
    }  
}  
  
int main() {  
    Task tasks[MAX\_TASKS];  
    int num\_tasks;  
  
  
    printf("Enter the number of tasks (max %d): ", MAX\_TASKS);  
    scanf("%d", &num\_tasks);  
  
    if (num\_tasks <= 0 || num\_tasks > MAX\_TASKS) {  
        printf("Invalid number of tasks. Please enter a number between 1 and %d.\n", MAX\_TASKS);  
        return 1;  
    }  
  
  
    for (int i = 0; i < num\_tasks; i++) {  
        printf("\nEnter details for Task %d:\n", i + 1);  
        tasks[i].id = i + 1;  // Assign a unique ID to each task  
        printf("Execution Time: ");  
        scanf("%d", &tasks[i].execution\_time);  
        printf("Deadline: ");  
        scanf("%d", &tasks[i].deadline);  
    }  
  
    edf\_schedule(tasks, num\_tasks);  
  
    return 0;  
}



Producer and consumer

#include <stdio.h>  
#include <pthread.h>  
#include <semaphore.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
#define BUFFER\_SIZE 5  
#define MAX\_ITEMS 10 // Maximum items to be produced and consumed  
  
int buffer[BUFFER\_SIZE];  
int in = 0, out = 0;  
  
sem\_t empty, full;  
pthread\_mutex\_t mutex;  
  
void\* producer(void\* arg) {  
    for (int i = 0; i < MAX\_ITEMS; i++) {  
        int item = rand() % 100;  // Produce an item  
        sem\_wait(&empty);         // Wait for an empty space in the buffer  
        pthread\_mutex\_lock(&mutex); // Critical section  
  
        buffer[in] = item;  
        printf("Produced: %d\n", item);  
        in = (in + 1) % BUFFER\_SIZE;  
  
        pthread\_mutex\_unlock(&mutex); // Exit critical section  
        sem\_post(&full);             // Signal that buffer is full for consumer  
  
        sleep(1); // Sleep to simulate production time  
    }  
    return NULL;  
}  
  
void\* consumer(void\* arg) {  
    for (int i = 0; i < MAX\_ITEMS; i++) {  
        sem\_wait(&full);           // Wait for a filled space in the buffer  
        pthread\_mutex\_lock(&mutex); // Critical section  
  
        int item = buffer[out];  
        printf("Consumed: %d\n", item);  
        out = (out + 1) % BUFFER\_SIZE;  
  
        pthread\_mutex\_unlock(&mutex); // Exit critical section  
        sem\_post(&empty);            // Signal that buffer has empty space for producer  
  
        sleep(1); // Sleep to simulate consumption time  
    }  
    return NULL;  
}  
  
int main() {  
    pthread\_t prod\_thread, cons\_thread;  
  
    // Initialize the semaphores  
    sem\_init(&empty, 0, BUFFER\_SIZE); // Initially, the buffer is empty  
    sem\_init(&full, 0, 0);            // Initially, no items are in the buffer  
    pthread\_mutex\_init(&mutex, NULL);  
  
    // Create the producer and consumer threads  
    pthread\_create(&prod\_thread, NULL, producer, NULL);  
    pthread\_create(&cons\_thread, NULL, consumer, NULL);  
  
    // Wait for the threads to finish  
    pthread\_join(prod\_thread, NULL);  
    pthread\_join(cons\_thread, NULL);  
  
    // Destroy the semaphores and mutex  
    sem\_destroy(&empty);  
    sem\_destroy(&full);  
    pthread\_mutex\_destroy(&mutex);  
  
    return 0;  
}



Dining Philosopher

#include <stdio.h>  
#include <pthread.h>  
#include <semaphore.h>  
#include <unistd.h>  
  
#define NUM\_PHILOSOPHERS 5  
  
sem\_t chopsticks[NUM\_PHILOSOPHERS];  
  
void\* philosopher(void\* num) {  
    int phil = \*(int\*)num;  
  
    printf("Philosopher %d is thinking.\n", phil);  
    sleep(1);  // Simulate thinking time  
  
    printf("Philosopher %d is hungry.\n", phil);  
  
    // Pick up left chopstick (phil)  
    sem\_wait(&chopsticks[phil]);  
  
    // Pick up right chopstick (phil + 1)  
    sem\_wait(&chopsticks[(phil + 1) % NUM\_PHILOSOPHERS]);  
  
    // Eating  
    printf("Philosopher %d is eating.\n", phil);  
    sleep(2);  // Simulate eating time  
  
    // Put down right chopstick (phil + 1)  
    sem\_post(&chopsticks[(phil + 1) % NUM\_PHILOSOPHERS]);  
  
    // Put down left chopstick (phil)  
    sem\_post(&chopsticks[phil]);  
  
    printf("Philosopher %d finished eating.\n", phil);  
  
    return NULL;  
}  
  
int main() {  
    pthread\_t threads[NUM\_PHILOSOPHERS];  
    int phil\_num[NUM\_PHILOSOPHERS];  
  
    // Initialize the chopsticks (semaphores)  
    for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {  
        sem\_init(&chopsticks[i], 0, 1);  // Each chopstick is available (value = 1)  
    }  
  
    // Create philosopher threads, but only run one or two philosophers  
    for (int i = 0; i < 2; i++) {  // Limit to the first 2 philosophers for output  
        phil\_num[i] = i;  
        pthread\_create(&threads[i], NULL, philosopher, &phil\_num[i]);  
    }  
  
    // Wait for the philosophers to finish  
    for (int i = 0; i < 2; i++) {  // Wait only for the first 2 philosophers  
        pthread\_join(threads[i], NULL);  
    }  
  
    // Destroy the semaphores  
    for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {  
        sem\_destroy(&chopsticks[i]);  
    }  
  
    return 0;  
}

