WRITE A C PROGRAM TO SIMULATE REAL-TIME CPU SCHEDULING ALGORITHMS:

1. **RATE- MONOTONIC**
2. **EARLIEST-DEADLINE FIRST**

Rate monotomic

```
#include <stdio.h>

#include <math.h>


struct Process {

   int pid;

   int burst;

   int period;

};


// Function to calculate the least common multiple

int lcm(int a, int b) {

   int temp_a = a, temp_b = b;

   while (temp_b != 0) {

      int temp = temp_b;

      temp_b = temp_a % temp_b;

      temp_a = temp;

   }

   return (a * b) / temp_a;

}


// Function to calculate LCM of all periods

int calculateLCM(int periods[], int n) {

   int res = periods[0];
```

```
    for (int i = 1; i < n; i++) {
        res = lcm(res, periods[i]);
    }
    return res;
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process p[n];
    int periods[n];

    printf("Enter the CPU burst times:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &p[i].burst);
        p[i].pid = i + 1;
    }

    printf("Enter the time periods:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &p[i].period);
        periods[i] = p[i].period;
    }

    int lcm_val = calculateLCM(periods, n);
    printf("LCM=%d\n\n", lcm_val);
```

```
    printf("Rate Monotone Scheduling:\n");

    printf("PID\tBurst\tPeriod\n");

    for (int i = 0; i < n; i++) {

        printf("%d\t%d\t%d\n", p[i].pid, p[i].burst, p[i].period);

    }


    // Calculate CPU utilization

    double utilization = 0;

    for (int i = 0; i < n; i++) {

        utilization += (double)p[i].burst / p[i].period;

    }


    double bound = n * (pow(2, 1.0 / n) - 1);

    printf("\n%.6lf <= %.6lf => %s\n",

           utilization, bound,

           (utilization <= bound) ? "true" : "false");


    return 0;

}
```

```
Enter the number of processes:3
Enter the CPU burst times:
3 6 8
Enter the time periods:
3 4 5
LCM=60

Rate Monotone Scheduling:
PID     Burst   Period
1               3               3
2               6               4
3               8               5

4.100000 <= 0.779763 =>false

Process returned 0 (0x0)   execution time : 17.091 s
Press any key to continue.
```

EARLIST DEADLINE

```
#include <stdio.h>

#include <limits.h>


#define MAX 10


struct Process {

    int pid;

    int burst;

    int deadline;

    int period;

    int remaining_time;

    int next_release;

    int abs_deadline;

};


int main() {

    int n;

    printf("Enter the number of processes: ");

    scanf("%d", &n);


    struct Process p[MAX];


    printf("Enter the CPU burst times:\n");

    for (int i = 0; i < n; i++) {

        scanf("%d", &p[i].burst);

        p[i].pid = i + 1;

        p[i].remaining_time = 0;

        p[i].next_release = 0;
```

```c
    p[i].abs_deadline = 0;
}

printf("Enter the deadlines:\n");
for (int i = 0; i < n; i++) {
    scanf("%d", &p[i].deadline);
}

printf("Enter the time periods:\n");
for (int i = 0; i < n; i++) {
    scanf("%d", &p[i].period);
}

printf("\nEarliest Deadline Scheduling:\n");
printf("PID\tBurst\tDeadline\tPeriod\n");
for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t\t%d\n", p[i].pid, p[i].burst, p[i].deadline, p[i].period);
}

int total_time = 6; // match the screenshot's time
printf("\nScheduling occurs for %d ms\n", total_time);

for (int time = 0; time < total_time; time++) {
    // Release new instances at their release times
    for (int i = 0; i < n; i++) {
        if (time == p[i].next_release) {
            p[i].remaining_time = p[i].burst;
            p[i].abs_deadline = time + p[i].deadline;
            p[i].next_release += p[i].period;
```

```
        }
      }

      // Pick the task with the earliest absolute deadline
      int min_deadline = INT_MAX;
      int selected = -1;

      for (int i = 0; i < n; i++) {
        if (p[i].remaining_time > 0 && p[i].abs_deadline < min_deadline) {
          min_deadline = p[i].abs_deadline;
          selected = i;
        }
      }

      if (selected != -1) {
        printf("%dms: Task %d is running.\n", time, p[selected].pid);
        p[selected].remaining_time--;
      } else {
        printf("%dms: CPU is idle.\n", time);
      }
    }

    return 0;
}
```

```
Enter the number of processes:3
Enter the CPU burst times:
2 3 4
Enter the deadlines:
1 2 3
Enter the time periods:
1 2 3

Earliest Deadline Scheduling:
PID              Burst          Deadline        Period
1               2              1               1
2               3              2               2
3               4              3               3
Scheduling occurs for 6 ms

0ms : Task 1 is running.
1ms : Task 1 is running.
2ms : Task 1 is running.
3ms : Task 1 is running.
4ms : Task 1 is running.
5ms : Task 1 is running.

Process returned 6 (0x6)   execution time : 17.130 s
Press any key to continue.
```

WRITE A C PROGRAM TO IMPLEMENT

1. **PRODUCER CONSUMER PROBLEM**
2. **DINING PHILOSOPHER**

PRODUCER- CONSUMER

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>
```

```
#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0, out = 0;

sem_t empty, full;
pthread_mutex_t mutex;

void* producer(void* arg) {
  int item;
  for (int i = 0; i < 10; i++) {
    item = rand() % 100;
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);

    buffer[in] = item;
    printf("Producer produced: %d\n", item);
    in = (in + 1) % BUFFER_SIZE;

    pthread_mutex_unlock(&mutex);
    sem_post(&full);
    sleep(1);
  }
  return NULL;
}

void* consumer(void* arg) {
  int item;
```

```c
    for (int i = 0; i < 10; i++) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);

        item = buffer[out];
        printf("Consumer consumed: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t prod, cons;

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    sem_destroy(&empty);
```

```
    sem_destroy(&full);

    pthread_mutex_destroy(&mutex);


    return 0;
}
```

```
Producer produced: 41
Consumer consumed: 41
Producer produced: 67
Consumer consumed: 67
Producer produced: 34
Consumer consumed: 34
Producer produced: 0
Consumer consumed: 0
Producer produced: 69
Consumer consumed: 69
Producer produced: 24
Consumer consumed: 24
Producer produced: 78
Consumer consumed: 78
Producer produced: 58
Consumer consumed: 58
Producer produced: 62
Consumer consumed: 62
Producer produced: 64
Consumer consumed: 64

Process returned 0 (0x0)   execution time : 10.110 s
Press any key to continue.
```

DINIG PHILOSOPHER

```
#include <windows.h>

#include <stdio.h>


#define NUM_PHILOSOPHERS 5


HANDLE forks[NUM_PHILOSOPHERS];  // One mutex per fork


DWORD WINAPI philosopher(LPVOID param) {
```

```
int id = *(int*)param;
int left = id;                    // Left fork
int right = (id + 1) % NUM_PHILOSOPHERS; // Right fork

for (int i = 0; i < 3; i++) {  // Let each philosopher eat 3 times
  printf("Philosopher %d is thinking...\n", id);
  Sleep(1000);  // Simulate thinking

  // Deadlock-free strategy: Pick lower-numbered fork first
  if (id % 2 == 0) {
    WaitForSingleObject(forks[left], INFINITE);
    WaitForSingleObject(forks[right], INFINITE);
  } else {
    WaitForSingleObject(forks[right], INFINITE);
    WaitForSingleObject(forks[left], INFINITE);
  }

  // Eating
  printf("Philosopher %d is eating...\n", id);
  Sleep(1500); // Simulate eating

  // Release forks
  ReleaseMutex(forks[left]);
  ReleaseMutex(forks[right]);

  printf("Philosopher %d finished eating and put down forks.\n", id);
  Sleep(1000);  // Back to thinking
}
```

```
    return 0;
}

int main() {
    HANDLE threads[NUM_PHILOSOPHERS];
    int ids[NUM_PHILOSOPHERS];

    // Initialize mutexes for forks
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        forks[i] = CreateMutex(NULL, FALSE, NULL);
    }

    // Create philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        ids[i] = i;
        threads[i] = CreateThread(NULL, 0, philosopher, &ids[i], 0, NULL);
        Sleep(10); // Small delay to avoid all threads sharing the same address
    }

    // Wait for all threads to finish
    WaitForMultipleObjects(NUM_PHILOSOPHERS, threads, TRUE, INFINITE);

    // Cleanup
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        CloseHandle(forks[i]);
        CloseHandle(threads[i]);
    }

    return 0;
```

}

```
Philosopher 0 is thinking...
Philosopher 1 is thinking...
Philosopher 2 is thinking...
Philosopher 3 is thinking...
Philosopher 4 is thinking...
Philosopher 0 is eating...
Philosopher 3 is eating...
Philosopher 0 finished eating and put down forks.
Philosopher 1 is eating...
Philosopher 3 finished eating and put down forks.
Philosopher 4 is eating...
Philosopher 0 is thinking...
Philosopher 3 is thinking...
Philosopher 2 is eating...
Philosopher 1 finished eating and put down forks.
Philosopher 4 finished eating and put down forks.
Philosopher 0 is eating...
Philosopher 1 is thinking...
Philosopher 4 is thinking...
Philosopher 3 is eating...
Philosopher 2 finished eating and put down forks.
Philosopher 0 finished eating and put down forks.
Philosopher 1 is eating...
Philosopher 2 is thinking...
Philosopher 3 finished eating and put down forks.
Philosopher 4 is eating...
Philosopher 0 is thinking...
Philosopher 1 finished eating and put down forks.
Philosopher 2 is eating...
Philosopher 3 is thinking...
Philosopher 4 finished eating and put down forks.
Philosopher 0 is eating...
Philosopher 1 is thinking...
Philosopher 2 finished eating and put down forks.
Philosopher 3 is eating...
Philosopher 4 is thinking...
Philosopher 0 finished eating and put down forks.
Philosopher 1 is eating...
Philosopher 2 is thinking...
Philosopher 3 finished eating and put down forks.
Philosopher 4 is eating...
Philosopher 2 is eating...
Philosopher 1 finished eating and put down forks.
Philosopher 4 finished eating and put down forks.
Philosopher 2 finished eating and put down forks.

Process returned 0 (0x0)   execution time : 14.076 s
Press any key to continue.
```