**LAB-5**

**Real-Time CPU Scheduling algorithms**

1. Write a C program to simulate Real-Time CPU Scheduling algorithms:

(a) Rate- Monotonic

(b) Earliest-deadline First.

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <stdbool.h>

#define MAX\_PROCESS 10

typedef struct {

    int id;

    int burst\_time;

    float priority;

} Task;

int num\_of\_process;

int execution\_time[MAX\_PROCESS], period[MAX\_PROCESS], remain\_time[MAX\_PROCESS], deadline[MAX\_PROCESS], remain\_deadline[MAX\_PROCESS];

void get\_process\_info(int selected\_algo) {

    printf("Enter total number of processes (maximum %d):", MAX\_PROCESS);

    scanf("%d", &num\_of\_process);

    if (num\_of\_process < 1) {

        exit(0);

    }

    for (int i = 0; i < num\_of\_process; i++) {

        printf("\nProcess %d:\n", i + 1);

        printf("==> Execution time: ");

        scanf("%d", &execution\_time[i]);

        remain\_time[i] = execution\_time[i];

        if (selected\_algo == 2) {

            printf("==> Deadline: ");

            scanf("%d", &deadline[i]);

        } else {

            printf("==> Period: ");

            scanf("%d", &period[i]);

        }

    }

}

int max(int a, int b, int c) {

    int max;

    if (a >= b && a >= c)

        max = a;

    else if (b >= a && b >= c)

        max = b;

    else

        max = c;

    return max;

}

int get\_observation\_time(int selected\_algo)

{

    if (selected\_algo == 1) {

        return max(period[0], period[1], period[2]);

    } else if (selected\_algo == 2) {

        return max(deadline[0], deadline[1], deadline[2]);

    }

    return 0;

}

void print\_schedule(int process\_list[], int cycles) {

    printf("\nScheduling:\n\n");

    printf("Time: ");

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

        if (i < 10)

            printf("| 0%d ", i);

        else

            printf("| %d ", i);

    }

    printf("|\n");

    for (int i = 0; i < num\_of\_process; i++) {

        printf("P[%d]: ", i + 1);

        for (int j = 0; j < cycles; j++) {

            if (process\_list[j] == i + 1)

                printf("|####");

            else

                printf("|    ");

        }

        printf("|\n");

    }

}

void rate\_monotonic(int time) {

    int process\_list[100] = {0}, min = 999, next\_process = 0;

    float utilization = 0;

    for (int i = 0; i < num\_of\_process; i++) {

        utilization += (1.0 \* execution\_time[i]) / period[i];

    }

    int n = num\_of\_process;

    int m = (float)(n \* (pow(2, 1.0 / n) - 1));

    if (utilization > m) {

        printf("\nGiven problem is not schedulable under the said scheduling algorithm.\n");

        return;

    }

    for (int i = 0; i < time; i++) {

        min = 1000;

        for (int j = 0; j < num\_of\_process; j++) {

            if (remain\_time[j] > 0) {

                if (min > period[j]) {

                    min = period[j];

                    next\_process = j;

                }

            }

        }

        if (remain\_time[next\_process] > 0) {

            process\_list[i] = next\_process + 1;

            remain\_time[next\_process] -= 1;

        }

        for (int k = 0; k < num\_of\_process; k++) {

            if ((i + 1) % period[k] == 0) {

                remain\_time[k] = execution\_time[k];

            }

        }

    }

    print\_schedule(process\_list, time);

}

void earliest\_deadline\_first(int time) {

    float utilization = 0;

    for (int i = 0; i < num\_of\_process; i++) {

        utilization += (1.0 \* execution\_time[i]) / deadline[i];

    }

    int n = num\_of\_process;

    int process[num\_of\_process];

    int max\_deadline, current\_process = 0, min\_deadline, process\_list[time];

    bool is\_ready[num\_of\_process];

    for (int i = 0; i < num\_of\_process; i++) {

        is\_ready[i] = true;

        process[i] = i + 1;

    }

    max\_deadline = deadline[0];

    for (int i = 1; i < num\_of\_process; i++) {

        if (deadline[i] > max\_deadline)

            max\_deadline = deadline[i];

    }

    for (int i = 0; i < num\_of\_process; i++) {

        for (int j = i + 1; j < num\_of\_process; j++) {

            if (deadline[j] < deadline[i]) {

                int temp = execution\_time[j];

                execution\_time[j] = execution\_time[i];

                execution\_time[i] = temp;

                temp = deadline[j];

                deadline[j] = deadline[i];

                deadline[i] = temp;

                temp = process[j];

                process[j] = process[i];

                process[i] = temp;

            }

        }

    }

    for (int i = 0; i < num\_of\_process; i++) {

        remain\_time[i] = execution\_time[i];

        remain\_deadline[i] = deadline[i];

    }

    for (int t = 0; t < time; t++) {

        if (current\_process != -1) {

            --execution\_time[current\_process];

            process\_list[t] = process[current\_process];

        } else {

            process\_list[t] = 0;

        }

        for (int i = 0; i < num\_of\_process; i++) {

            --deadline[i];

            if ((execution\_time[i] == 0) && is\_ready[i]) {

                deadline[i] += remain\_deadline[i];

                is\_ready[i] = false;

            }

            if ((deadline[i] <= remain\_deadline[i]) && !is\_ready[i]) {

                execution\_time[i] = remain\_time[i];

                is\_ready[i] = true;

            }

        }

        min\_deadline = max\_deadline;

        current\_process = -1;

        for (int i = 0; i < num\_of\_process; i++) {

            if ((deadline[i] <= min\_deadline) && (execution\_time[i] > 0)) {

                current\_process = i;

                min\_deadline = deadline[i];

            }

        }

    }

    print\_schedule(process\_list, time);

}

int main()

{

    int option;

    int observation\_time;

    while (1) {

        printf("\n1. Rate Monotonic\n2. Earliest Deadline first\n3. Proportional Scheduling\n\nEnter your choice: ");

        scanf("%d", &option);

        switch (option) {

            case 1:

                get\_process\_info(option);

                observation\_time = get\_observation\_time(option);

                rate\_monotonic(observation\_time);

                break;

            case 2:

                get\_process\_info(option);

                observation\_time = get\_observation\_time(option);

                earliest\_deadline\_first(observation\_time);

                break;

            case 3:

                exit(0);

            default:

                printf("\nInvalid Statement");

        }

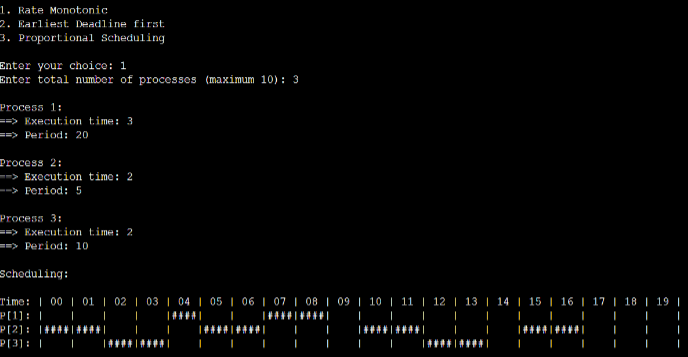
    }

    return 0;

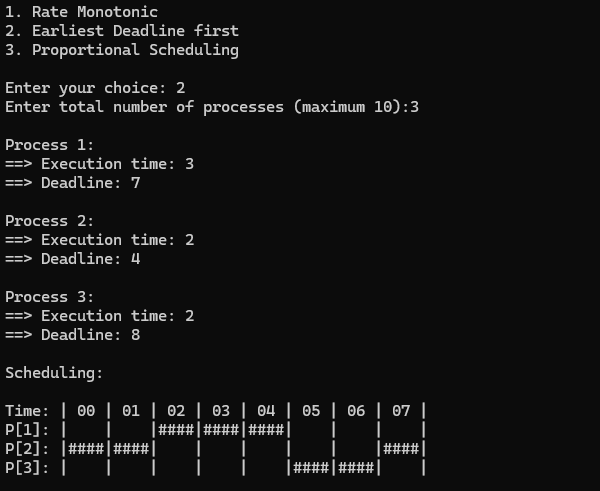
}

OUTPUT:

(a)Rate- Monotonic



(b)Earliest-deadline First



**Producer-Consumer problem**

1. Write a C program to simulate producer-consumer problem using semaphores.

#include <stdio.h>

#include <stdlib.h>

int mutex = 1, full = 0, empty = 3, x = 0;

void producer();

void consumer();

int wait(int);

int signal(int);

int main() {

    int n;

    printf("\n1. Producer\n2. Consumer\n3. Exit");

    while (1) {

        printf("\nEnter your choice: ");

        scanf("%d", &n);

        switch (n) {

            case 1:

                if ((mutex == 1) && (empty != 0)) {

                    producer();

                } else {

                    printf("Buffer is full!!");

                }

                break;

            case 2:

                if ((mutex == 1) && (full != 0)) {

                    consumer();

                } else {

                    printf("Buffer is empty!!");

                }

                break;

            case 3:

                exit(0);

                break;

            default:

                printf("Invalid choice! Please enter 1, 2, or 3.");

                break;

        }

    }

    return 0;

}

int wait(int s) {

    return --s;

}

int signal(int s) {

    return ++s;

}

void producer() {

    mutex = wait(mutex);

    full = signal(full);

    empty = wait(empty);

    x++;

    printf("\nProducer produces the item %d", x);

    mutex = signal(mutex);

}

void consumer() {

    mutex = wait(mutex);

    full = wait(full);

    empty = signal(empty);

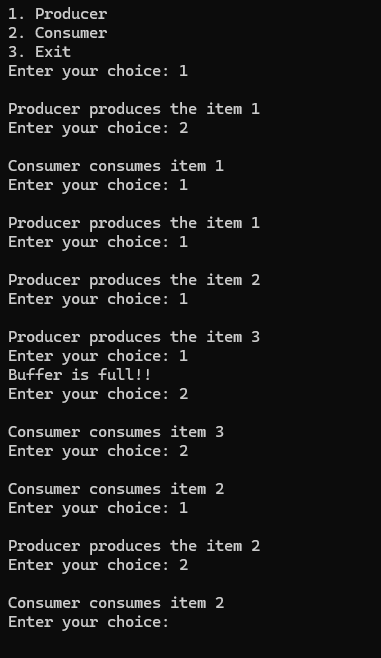
    printf("\nConsumer consumes item %d", x);

    x--;

    mutex = signal(mutex);

}

OUTPUT:



**Dining-Philosophers problem**

1. Write a C program to simulate the concept of Dining-Philosophers

problem.

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h> // Include for sleep function

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (i + 4) % N

#define RIGHT (i + 1) % N

int state[N];

int phil[N] = {0, 1, 2, 3, 4};

sem\_t mutex;

sem\_t S[N];

void test(int i) {

    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {

        state[i] = EATING;

        sleep(2);

        printf("Philosopher %d takes fork %d and %d\n", i + 1, LEFT + 1, i + 1);

        printf("Philosopher %d is Eating\n", i + 1);

        sem\_post(&S[i]);

    }

}

void take\_fork(int i) {

    sem\_wait(&mutex);

    state[i] = HUNGRY;

    printf("Philosopher %d is Hungry\n", i + 1);

    test(i);

    sem\_post(&mutex);

    sem\_wait(&S[i]);

    sleep(1);

}

void put\_fork(int i) {

    sem\_wait(&mutex);

    state[i] = THINKING;

    printf("Philosopher %d putting fork %d and %d down\n", i + 1, LEFT + 1, i + 1);

    printf("Philosopher %d is thinking\n", i + 1);

    test(LEFT);

    test(RIGHT);

    sem\_post(&mutex);

}

void\* philosopher(void\* num) {

    while (1) {

        int\* i = num;

        sleep(1);

        take\_fork(\*i);

        sleep(0);

        put\_fork(\*i);

    }

}

int main() {

    int i;

    pthread\_t thread\_id[N];

    sem\_init(&mutex, 0, 1);

    for (i = 0; i < N; i++)

        sem\_init(&S[i], 0, 0);

    for (i = 0; i < N; i++) {

        pthread\_create(&thread\_id[i], NULL, philosopher, &phil[i]);

        printf("Philosopher %d is thinking\n", i + 1);

    }

    for (i = 0; i < N; i++) {

        pthread\_join(thread\_id[i], NULL);

    }

}

OUTPUT:

