!!First-Come, First-Served (FCFS) Scheduling:

→To open the code interface:

mousepad file-name.c

→ Enter the code and save it.

```c
#include <stdio.h>
int main() {
    int n, i;
    printf("Enter number of processes: ");
    scanf("%d", &n);

    int bt[n], wt[n], tat[n];
    wt[0] = 0;

    printf("Enter burst time for each process:\n");
    for(i = 0; i < n; i++) {
        printf("P%d: ", i+1);
        scanf("%d", &bt[i]);
    }

    for(i = 1; i < n; i++) {
        wt[i] = wt[i-1] + bt[i-1];
    }

    for(i = 0; i < n; i++) {
        tat[i] = wt[i] + bt[i];
    }

    printf("\nProcess\tBT\tWT\tTAT\n");
    for(i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\n", i+1, bt[i], wt[i], tat[i]);
    }

    return 0;
}
```

-> for compiling:

gcc -o name file-name.c

->for execution:

./name

→ for code interface editing mousepad is used,

→ for code interface reading mode nano and vim is used.

2.Shortest-Job-First (SJF) Scheduling →

For non-preemptive:

```
#include <stdio.h>

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

    int pid[n], bt[n], wt[n], tat[n], temp;

    for(i = 0; i < n; i++) {
        pid[i] = i + 1;
        printf("Enter burst time for process P%d: ", pid[i]);
        scanf("%d", &bt[i]);
    }

    // Sort processes by burst time (SJF logic)
    for(i = 0; i < n - 1; i++) {
        for(j = i + 1; j < n; j++) {
            if(bt[i] > bt[j]) {
                // Swap burst time
                temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;
                // Swap process ID to match burst time
                temp = pid[i];
                pid[i] = pid[j];
                pid[j] = temp;
            }
        }
    }

    wt[0] = 0;
    for(i = 1; i < n; i++) {
        wt[i] = wt[i - 1] + bt[i - 1];
    }

    for(i = 0; i < n; i++) {
        tat[i] = wt[i] + bt[i];
    }

    printf("\nProcess\tBT\tWT\tTAT\n");
    for(i = 0; i < n; i++) {
```

```c
        printf("P%d\t%d\t%d\t%d\n", pid[i], bt[i], wt[i], tat[i]);
    }

    return 0;
}
```

For preemptive:

```c
#include <stdio.h>

int main() {
    int n, i, smallest, count = 0, time;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    int at[n], bt[n], rt[n], wt[n], tat[n];
    for(i = 0; i < n; i++) {
        printf("Enter arrival time and burst time for process P%d: ", i + 1);
        scanf("%d %d", &at[i], &bt[i]);
        rt[i] = bt[i]; // remaining time = burst time initially
    }

    int completed = 0;
    int min_time = 9999;
    smallest = -1;

    for(time = 0; completed < n; time++) {
        smallest = -1;
        min_time = 9999;

        for(i = 0; i < n; i++) {
            if(at[i] <= time && rt[i] > 0 && rt[i] < min_time) {
                min_time = rt[i];
                smallest = i;
            }
        }

        if(smallest == -1) continue;

        rt[smallest]--;

        if(rt[smallest] == 0) {
            completed++;
            int finish_time = time + 1;
            wt[smallest] = finish_time - at[smallest] - bt[smallest];
```

```
            if(wt[smallest] < 0) wt[smallest] = 0;
            tat[smallest] = wt[smallest] + bt[smallest];
        }
    }

    printf("\nProcess\tAT\tBT\tWT\tTAT\n");
    for(i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], wt[i], tat[i]);
    }

    return 0;
}
```

3.Priority Scheduling→

```
#include <stdio.h>

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

    int pid[n], bt[n], pr[n], wt[n], tat[n], temp;

    for(i = 0; i < n; i++) {
        pid[i] = i + 1;
        printf("Enter burst time and priority for process P%d: ", pid[i]);
        scanf("%d %d", &bt[i], &pr[i]);
    }

    // Sort by priority (lower number = higher priority)
    for(i = 0; i < n - 1; i++) {
        for(j = i + 1; j < n; j++) {
            if(pr[i] > pr[j]) {
                // Swap burst time
                temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;
                // Swap priority
                temp = pr[i];
                pr[i] = pr[j];
                pr[j] = temp;
                // Swap process ID
```

```
                temp = pid[i];
                pid[i] = pid[j];
                pid[j] = temp;
            }
        }
    }

    wt[0] = 0;
    for(i = 1; i < n; i++) {
        wt[i] = wt[i - 1] + bt[i - 1];
    }

    for(i = 0; i < n; i++) {
        tat[i] = wt[i] + bt[i];
    }

    printf("\nProcess\tBT\tPriority\tWT\tTAT\n");
    for(i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t\t%d\t%d\n", pid[i], bt[i], pr[i], wt[i], tat[i]);
    }

    return 0;
}
```

4.Round Robin (RR) →

```
#include <stdio.h>

int main() {
    int i, n, time = 0, tq, remain, flag = 0;
    printf("Enter number of processes: ");
    scanf("%d", &n);

    int bt[n], rt[n], wt[n], tat[n];
    remain = n;

    for(i = 0; i < n; i++) {
        printf("Enter burst time for P%d: ", i + 1);
        scanf("%d", &bt[i]);
        rt[i] = bt[i]; // Remaining time initialized
    }
```

```
    printf("Enter time quantum: ");
    scanf("%d", &tq);

    int count = 0;
    while(remain != 0) {
        flag = 0;
        for(i = 0; i < n; i++) {
            if(rt[i] > 0) {
                if(rt[i] > tq) {
                    time += tq;
                    rt[i] -= tq;
                } else {
                    time += rt[i];
                    wt[i] = time - bt[i];
                    rt[i] = 0;
                    tat[i] = wt[i] + bt[i];
                    remain--;
                }
                flag = 1;
            }
        }
        if(flag == 0) break;
    }

    printf("\nProcess\tBT\tWT\tTAT\n");
    for(i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\n", i + 1, bt[i], wt[i], tat[i]);
    }

    return 0;
}
```

5.Producer/Consumer Problem→

6.Bounded-Buffer Problem→

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

int *buffer;
int size;
int in = 0, out = 0;
```

```
int count = 0;

void produce() {
    if(count == size) {
        printf("Buffer is full! Cannot produce more.\n");
        return;
    }

    int item;
    printf("Enter item to produce: ");
    scanf("%d", &item);

    buffer[in] = item;
    printf("Produced: %d\n", item);

    in = (in + 1) % size;
    count++;
}

void consume() {
    if(count == 0) {
        printf("Buffer is empty! Nothing to consume.\n");
        return;
    }

    int item = buffer[out];
    printf("Consumed: %d\n", item);

    out = (out + 1) % size;
    count--;
}

int main() {
    int choice;

    printf("Enter buffer size: ");
    scanf("%d", &size);

    buffer = (int*)malloc(size * sizeof(int));
    if(buffer == NULL) {
        perror("Buffer allocation failed");
        exit(1);
    }
```

```
    printf("\n--- Producer-Consumer Menu ---\n");
    printf("1: Produce\n2: Consume\n0: Exit\n");

    while(1) {
        printf("\nEnter choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                produce();
                break;
            case 2:
                consume();
                break;
            case 0:
                printf("Exiting...\n");
                free(buffer);
                exit(0);
            default:
                printf("Invalid choice. Try again.\n");
        }
    }

    return 0;
}
```

7.Readers and Writers Problem→

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

int read_count = 0;
int data = 0;

pthread_mutex_t mutex;     // for read_count
sem_t wrt;                 // for writing access

void* reader(void* arg) {
    int id = *((int*)arg);
    pthread_mutex_lock(&mutex);
```

```
    read_count++;
    if(read_count == 1)
        sem_wait(&wrt);  // First reader locks writer
    pthread_mutex_unlock(&mutex);

    // Reading
    printf("Reader %d is reading data = %d\n", id, data);
    sleep(1);

    pthread_mutex_lock(&mutex);
    read_count--;
    if(read_count == 0)
        sem_post(&wrt);  // Last reader unlocks writer
    pthread_mutex_unlock(&mutex);

    free(arg);
    return NULL;
}

void* writer(void* arg) {
    int id = *((int*)arg);
    sem_wait(&wrt); // Wait for exclusive access

    // Writing
    data++;
    printf("Writer %d wrote data = %d\n", id, data);
    sleep(2);

    sem_post(&wrt); // Release access
    free(arg);
    return NULL;
}

int main() {
    int choice, id = 1;
    pthread_t tid[100]; // Store threads
    int t_index = 0;

    // Initialize synchronization primitives
    pthread_mutex_init(&mutex, NULL);
    sem_init(&wrt, 0, 1);

    printf("\n--- Readers-Writers Problem ---\n");
    printf("1: Create Reader\n2: Create Writer\n0: Exit\n");
```

```
    while(1) {
        printf("\nEnter choice: ");
        scanf("%d", &choice);

        if (choice == 1) {
            int *arg = malloc(sizeof(*arg));
            *arg = id;
            pthread_create(&tid[t_index++], NULL, reader, arg);
            id++;
        }
        else if (choice == 2) {
            int *arg = malloc(sizeof(*arg));
            *arg = id;
            pthread_create(&tid[t_index++], NULL, writer, arg);
            id++;
        }
        else if (choice == 0) {
            printf("Exiting...\n");
            break;
        }
        else {
            printf("Invalid choice.\n");
        }
    }

    // Join all threads before exiting
    for (int i = 0; i < t_index; i++) {
        pthread_join(tid[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    sem_destroy(&wrt);

    return 0;
}
```

8. Dining-Philosophers Problem→

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

```c
#define THINKING 0
#define HUNGRY 1
#define EATING 2

int *state;
pthread_mutex_t *mutex;
pthread_cond_t *cond;
int num_philosophers;

void test(int i) {
    // Check if philosopher can eat
    int left = (i + num_philosophers - 1) % num_philosophers;
    int right = (i + 1) % num_philosophers;
    if (state[i] == HUNGRY && state[left] != EATING && state[right] != EATING) {
        state[i] = EATING;
        printf("Philosopher %d is eating.\n", i + 1);
        pthread_cond_signal(&cond[i]);  // Notify philosopher to eat
    }
}

void take_forks(int i) {
    pthread_mutex_lock(&mutex[i]);

    state[i] = HUNGRY;
    printf("Philosopher %d is hungry.\n", i + 1);
    test(i);  // Try to eat immediately

    while (state[i] != EATING) {
        pthread_cond_wait(&cond[i], &mutex[i]);  // Wait until philosopher can eat
    }

    pthread_mutex_unlock(&mutex[i]);
}

void put_forks(int i) {
    pthread_mutex_lock(&mutex[i]);

    state[i] = THINKING;
    printf("Philosopher %d is thinking.\n", i + 1);

    int left = (i + num_philosophers - 1) % num_philosophers;
    int right = (i + 1) % num_philosophers;
```

```
    // Test neighbors if they can eat
    test(left);
    test(right);

    pthread_mutex_unlock(&mutex[i]);
}

void* philosopher(void* num) {
    int i = *((int*)num);
    while (1) {
        sleep(rand() % 5 + 1);  // Thinking
        take_forks(i);          // Hungry -> Eat
        sleep(rand() % 5 + 1);  // Eating
        put_forks(i);           // Finished eating -> Thinking
    }
    return NULL;
}

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

    state = (int*)malloc(num_philosophers * sizeof(int));
    mutex = (pthread_mutex_t*)malloc(num_philosophers * sizeof(pthread_mutex_t));
    cond = (pthread_cond_t*)malloc(num_philosophers * sizeof(pthread_cond_t));
    pthread_t *threads = (pthread_t*)malloc(num_philosophers * sizeof(pthread_t));

    // Initialize the states, mutexes, and condition variables
    for (int i = 0; i < num_philosophers; i++) {
        state[i] = THINKING;
        pthread_mutex_init(&mutex[i], NULL);
        pthread_cond_init(&cond[i], NULL);
    }

    // Create philosopher threads
    for (int i = 0; i < num_philosophers; i++) {
        int *arg = malloc(sizeof(*arg));
        *arg = i;
        pthread_create(&threads[i], NULL, philosopher, arg);
    }

    // Join all threads
    for (int i = 0; i < num_philosophers; i++) {
        pthread_join(threads[i], NULL);
```

```
    }

    // Clean up resources
    free(state);
    free(mutex);
    free(cond);
    free(threads);

    return 0;
}
```

9.The Sleeping Barber problem→

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

#define MAX_WAITING_CUSTOMERS 5

// Shared resources
int waiting_customers = 0; // Number of waiting customers
pthread_mutex_t mutex;
pthread_cond_t barber_ready, customer_ready;

void* barber(void* arg) {
    while (1) {
        pthread_mutex_lock(&mutex);

        // If there are no customers, barber sleeps
        while (waiting_customers == 0) {
            printf("Barber is sleeping...\n");
            pthread_cond_wait(&customer_ready, &mutex);
        }

        // Cutting hair
        waiting_customers--;
        printf("Barber is cutting hair. %d customers left in the waiting area.\n",
waiting_customers);
        pthread_cond_signal(&barber_ready); // Signal the customer to sit in the chair
        pthread_mutex_unlock(&mutex);

        // Simulate cutting hair time
        sleep(3);  // Barber cuts hair for 3 seconds
    }
```

```
    return NULL;
}

void* customer(void* arg) {
    pthread_mutex_lock(&mutex);

    if (waiting_customers < MAX_WAITING_CUSTOMERS) {
        waiting_customers++;
        printf("Customer arrived and is waiting. %d customers in the waiting area.\n",
waiting_customers);
        pthread_cond_signal(&customer_ready);  // Wake up the barber if necessary
        pthread_cond_wait(&barber_ready, &mutex);  // Wait until the barber is ready
        printf("Customer is getting a haircut.\n");
    } else {
        printf("Customer leaves because there are no available seats.\n");
    }

    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main() {
    pthread_t barber_thread, customer_thread[10];

    // Initialize the mutex and condition variables
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&barber_ready, NULL);
    pthread_cond_init(&customer_ready, NULL);

    // Create the barber thread
    pthread_create(&barber_thread, NULL, barber, NULL);

    // Simulate customers arriving at different times
    for (int i = 0; i < 10; i++) {
        sleep(rand() % 2);  // Random delay between customer arrivals
        pthread_create(&customer_thread[i], NULL, customer, NULL);
    }

    // Wait for customer threads to finish
    for (int i = 0; i < 10; i++) {
        pthread_join(customer_thread[i], NULL);
    }

    // Join the barber thread
```

```c
    pthread_join(barber_thread, NULL);

    // Cleanup
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&barber_ready);
    pthread_cond_destroy(&customer_ready);

    return 0;
}
```