1. Write a program in C to implement producer-consumer problem using semaphore.

- The producer-consumer problem is a classic synchronization problem where:
  - Producers generate data and put it into a buffer
  - Consumers take data from the buffer
  - We need to ensure producers don't add data to a full buffer and consumers don't remove data from an empty buffer

```c
#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 mutex, empty, full;

void *producer(void *arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        item = rand() % 100;  // Produce random item

        sem_wait(&empty);     // Wait if buffer is full
        sem_wait(&mutex);     // Enter critical section

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

        sem_post(&mutex);     // Leave critical section
        sem_post(&full);      // Signal that buffer has one more item

        sleep(rand() % 2);   // Simulate random production time
    }
    return NULL;
}

void *consumer(void *arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        sem_wait(&full);      // Wait if buffer is empty
        sem_wait(&mutex);     // Enter critical section

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

        sem_post(&mutex);     // Leave critical section
```

```
        sem_post(&empty);     // Signal that buffer has one more empty slot

        sleep(rand() % 2);    // Simulate random consumption time
    }
    return NULL;
}

int main() {
    pthread_t prod_thread, cons_thread;

    // Initialize semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    // Create producer and consumer threads
    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);

    // Wait for threads to finish
    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    // Destroy semaphores
    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

Step-by-Step Explanation

1. Header Files and Preprocessor Directives

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define BUFFER_SIZE 5
```

- **Purpose**: These lines include necessary libraries and define a constant.
- **Details**:
  - `<stdio.h>`: For input/output operations (e.g., `printf`).
  - `<stdlib.h>`: For general utilities (e.g., `rand`).
  - `<pthread.h>`: For POSIX threads (creating and managing threads).
  - `<semaphore.h>`: For semaphore operations (synchronization primitives).
  - `<unistd.h>`: For POSIX functions (e.g., `sleep`).
  - `#define BUFFER_SIZE 5`: Defines a constant for the size of the shared buffer (5 slots).

2. Global Variables

```
int buffer[BUFFER_SIZE];
int in = 0, out = 0;
sem_t mutex, empty, full;
```

- **Purpose**: Define the shared buffer and synchronization variables.
- **Details**:
  - `buffer[BUFFER_SIZE]` : An array of size 5 to store produced items (integers).
  - `in` : Index where the producer will insert the next item (points to the next empty slot).
  - `out` : Index where the consumer will retrieve the next item (points to the next filled slot).
  - `sem_t mutex` : A binary semaphore (mutex) initialized to 1 for mutual exclusion, ensuring only one thread accesses the buffer at a time.
  - `sem_t empty` : A counting semaphore initialized to `BUFFER_SIZE` (5), tracking the number of empty slots in the buffer.
  - `sem_t full` : A counting semaphore initialized to 0, tracking the number of filled slots in the buffer.

3. Producer Function

```
void *producer(void *arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        item = rand() % 100;  // Produce random item
        sem_wait(&empty);     // Wait if buffer is full
        sem_wait(&mutex);     // Enter critical section
        buffer[in] = item;
        printf("Producer produced %d\n", item);
        in = (in + 1) % BUFFER_SIZE;
        sem_post(&mutex);     // Leave critical section
        sem_post(&full);      // Signal that buffer has one more item
        sleep(rand() % 2);    // Simulate random production time
    }
    return NULL;
}
```

- **Purpose**: Defines the producer thread’s behavior, which generates items and places them in the buffer.
- **Step-by-Step**:
  - **Loop**: Runs 10 times to produce 10 items.
  - `item = rand() % 100` : Generates a random integer (0–99) to simulate an item.
  - `sem_wait(&empty)` : Decrements the `empty` semaphore. If the buffer is full (`empty == 0`), the producer waits until a slot becomes available.
  - `sem_wait(&mutex)` : Locks the mutex (decrements to 0) to ensure exclusive access to the buffer, preventing race conditions.
  - `buffer[in] = item` : Places the item in the buffer at index `in` .
  - `printf("Producer produced %d\n", item)` : Prints the produced item for tracking.

- `in = (in + 1) % BUFFER_SIZE`: Advances the `in` index circularly (modulo `BUFFER_SIZE`) to point to the next empty slot.
- `sem_post(&mutex)`: Unlocks the mutex (increments to 1), allowing other threads to access the buffer.
- `sem_post(&full)`: Increments the `full` semaphore, signaling that a new item is available for the consumer.
- `sleep(rand() % 2)`: Pauses for 0 or 1 second to simulate variable production time.
- `return NULL`: Indicates thread completion.

4. Consumer Function

```
void *consumer(void *arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        sem_wait(&full);      // Wait if buffer is empty
        sem_wait(&mutex);     // Enter critical section
        item = buffer[out];
        printf("Consumer consumed %d\n", item);
        out = (out + 1) % BUFFER_SIZE;
        sem_post(&mutex);     // Leave critical section
        sem_post(&empty);     // Signal that buffer has one more empty slot
        sleep(rand() % 2);    // Simulate random consumption time
    }
    return NULL;
}
```

- **Purpose**: Defines the consumer thread's behavior, which retrieves and processes items from the buffer.
- **Step-by-Step**:
  - **Loop**: Runs 10 times to consume 10 items.
  - `sem_wait(&full)`: Decrements the `full` semaphore. If the buffer is empty (`full == 0`), the consumer waits until an item is available.
  - `sem_wait(&mutex)`: Locks the mutex to ensure exclusive access to the buffer.
  - `item = buffer[out]`: Retrieves the item from the buffer at index `out`.
  - `printf("Consumer consumed %d\n", item)`: Prints the consumed item for tracking.
  - `out = (out + 1) % BUFFER_SIZE`: Advances the `out` index circularly to point to the next filled slot.
  - `sem_post(&mutex)`: Unlocks the mutex, allowing other threads to access the buffer.
  - `sem_post(&empty)`: Increments the `empty` semaphore, signaling that a slot is now free for the producer.
  - `sleep(rand() % 2)`: Pauses for 0 or 1 second to simulate variable consumption time.
  - `return NULL`: Indicates thread completion.

5. Main Function

```
int main() {
    pthread_t prod_thread, cons_thread;
```

```
    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);
    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);
    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);
    return 0;
}
```

- **Purpose**: Sets up the program, initializes synchronization primitives, creates threads, and ensures proper cleanup.
- **Step-by-Step**:
  - `pthread_t prod_thread, cons_thread` : Declares thread identifiers for one producer and one consumer thread.
  - `sem_init(&mutex, 0, 1)` : Initializes the mutex semaphore to 1 (unlocked, allowing one thread to access the critical section).
  - `sem_init(&empty, 0, BUFFER_SIZE)` : Initializes the `empty` semaphore to 5 (all buffer slots are initially empty).
  - `sem_init(&full, 0, 0)` : Initializes the `full` semaphore to 0 (no items in the buffer initially).
  - `pthread_create(&prod_thread, NULL, producer, NULL)` : Creates the producer thread, passing the `producer` function and no arguments (`NULL`).
  - `pthread_create(&cons_thread, NULL, consumer, NULL)` : Creates the consumer thread, passing the `consumer` function and no arguments.
  - `pthread_join(prod_thread, NULL)` : Waits for the producer thread to complete.
  - `pthread_join(cons_thread, NULL)` : Waits for the consumer thread to complete.
  - `sem_destroy(&mutex)` : Frees the mutex semaphore.
  - `sem_destroy(&empty)` : Frees the `empty` semaphore.
  - `sem_destroy(&full)` : Frees the `full` semaphore.
  - `return 0` : Indicates successful program termination.

6. Synchronization Mechanism

- **Semaphores**:
  - **Mutex**: Ensures mutual exclusion so that only one thread (producer or consumer) modifies the buffer at a time.
  - **Empty**: Tracks available buffer slots. The producer waits if the buffer is full.
  - **Full**: Tracks filled buffer slots. The consumer waits if the buffer is empty.
- **Circular Buffer**: The `in` and `out` indices use modulo (`% BUFFER_SIZE`) to wrap around, simulating a circular buffer where the producer and consumer cycle through the 5 slots.

7. Program Flow

- The program starts by initializing semaphores and creating one producer and one consumer thread.

- The producer generates 10 random items, placing them in the buffer when slots are available (`empty > 0`).
- The consumer retrieves 10 items from the buffer when items are available (`full > 0`).
- The mutex ensures that buffer operations (insertions and removals) are atomic.
- Random sleep times simulate varying production and consumption speeds, which may cause the producer to wait (if the buffer is full) or the consumer to wait (if the buffer is empty).
- After both threads complete (each processing 10 items), the program cleans up semaphores and exits.

8. Sample Output

```
Producer produced 83
Consumer consumed 83
Producer produced 47
Consumer consumed 47
Producer produced 12
Producer produced 91
Consumer consumed 12
Consumer consumed 91
...
```

- The output varies due to random item values and sleep times, but it will show interleaved producer and consumer actions, with no race conditions or buffer overflows/underflows.

9. Key Points

- **Correctness**: The semaphores ensure:
  - The buffer doesn’t overflow (producer waits when full).
  - The buffer doesn’t underflow (consumer waits when empty).
  - No race conditions occur (mutex protects buffer access).
- **Bounded Buffer**: The buffer size is fixed at 5, enforced by the `empty` and `full` semaphores.
- **Thread Safety**: The mutex prevents simultaneous access to the shared buffer.
- **Scalability**: This version uses one producer and one consumer, but the logic can be extended to multiple threads (as in the original code you provided earlier).

---

2. Sleeping Barber Problem

The sleeping barber problem is a classic inter-process communication and synchronization problem that illustrates the challenges of managing shared resources among multiple processes.

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

#define NUM_CHAIRS 5  // Number of waiting chairs in the barber shop

sem_t customers;      // Counts waiting customers (0 initially)
```

```c
sem_t barber;         // Signals when barber is ready (0 initially)
sem_t access_seats;   // Mutex for accessing shared variables
int free_seats = NUM_CHAIRS; // Number of available waiting chairs

void *barber_thread(void *arg) {
    while(1) {
        // Sleep when no customers (sem_wait blocks if customers == 0)
        sem_wait(&customers);

        // Get access to modify seat count
        sem_wait(&access_seats);

        // One chair becomes free
        free_seats++;
        printf("Barber is cutting hair. Seats available: %d\n", free_seats);

        // Signal that barber is ready
        sem_post(&barber);

        // Release seat access
        sem_post(&access_seats);

        // Time taken to cut hair
        sleep(rand() % 3 + 1);
    }
    return NULL;
}

void *customer_thread(void *arg) {
    int id = *(int *)arg;

    // Try to get access to seats
    sem_wait(&access_seats);

    if(free_seats > 0) {
        // There's a free chair
        free_seats--;
        printf("Customer %d is waiting. Seats left: %d\n", id, free_seats);

        // Notify barber there's a customer
        sem_post(&customers);

        // Release seat access
        sem_post(&access_seats);

        // Wait for barber to be ready
        sem_wait(&barber);

        printf("Customer %d is getting a haircut\n", id);
    }
    else {
        // No chairs available, customer leaves
```

```
        printf("Customer %d left - no chairs available\n", id);
        sem_post(&access_seats);
    }

    return NULL;
}

int main() {
    pthread_t barber_tid;
    pthread_t customer_tids[15]; // Let's simulate 15 customers
    int customer_ids[15];

    // Initialize semaphores
    sem_init(&customers, 0, 0);
    sem_init(&barber, 0, 0);
    sem_init(&access_seats, 0, 1);

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

    // Create customer threads
    for(int i = 0; i < 15; i++) {
        customer_ids[i] = i + 1;
        pthread_create(&customer_tids[i], NULL, customer_thread, &customer_ids[i]);
        sleep(rand() % 2); // Random arrival time between customers
    }

    // Wait for all customers to finish
    for(int i = 0; i < 15; i++) {
        pthread_join(customer_tids[i], NULL);
    }

    // Clean up (though in practice the barber thread runs indefinitely)
    sem_destroy(&customers);
    sem_destroy(&barber);
    sem_destroy(&access_seats);

    return 0;
}
```

Step-by-Step Explanation

1. Header Files and Preprocessor Directives

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define NUM_CHAIRS 5
```

- **Purpose**: Include necessary libraries and define a constant for the number of waiting chairs.
- **Details**:
  - `<stdio.h>`: For input/output (`printf`).
  - `<stdlib.h>`: For utilities like `rand`.
  - `<pthread.h>`: For POSIX threads.
  - `<semaphore.h>`: For semaphore operations.
  - `<unistd.h>`: For `sleep`.
  - `#define NUM_CHAIRS 5`: Sets the number of waiting chairs to 5.

2. Global Variables

```
sem_t customers;      // Counts waiting customers (0 initially)
sem_t barber;         // Signals when barber is ready (0 initially)
sem_t access_seats;   // Mutex for accessing shared variables
int free_seats = NUM_CHAIRS; // Number of available waiting chairs
```

- **Purpose**: Define synchronization primitives and shared state.
- **Details**:
  - `customers`: A counting semaphore initialized to 0, tracking the number of waiting customers.
  - `barber`: A semaphore initialized to 0, signaling when the barber is ready to cut hair.
  - `access_seats`: A binary semaphore (mutex) initialized to 1, ensuring exclusive access to `free_seats`.
  - `free_seats`: Tracks available waiting chairs, initially set to 5.

3. Barber Thread Function

```
void *barber_thread(void *arg) {
    while(1) {
        sem_wait(&customers);        // Sleep when no customers
        sem_wait(&access_seats);     // Get access to modify seat count
        free_seats++;                // One chair becomes free
        printf("Barber is cutting hair. Seats available: %d\n", free_seats);
        sem_post(&barber);           // Signal that barber is ready
        sem_post(&access_seats);     // Release seat access
        sleep(rand() % 3 + 1);       // Time taken to cut hair
    }
    return NULL;
}
```

- **Purpose**: Defines the barber's behavior, which involves waiting for customers, cutting hair, and freeing chairs.
- **Step-by-Step**:
  - **Infinite Loop**: The barber runs indefinitely, simulating continuous operation.
  - `sem_wait(&customers)`: Blocks if no customers are waiting (`customers == 0`), simulating the barber sleeping.
  - `sem_wait(&access_seats)`: Locks the mutex to safely modify `free_seats`.
  - `free_seats++`: Increments available chairs as the customer getting a haircut leaves the waiting area.

- `printf("Barber is cutting hair. Seats available: %d\n", free_seats)` : Logs the barber's action and current chair availability.
- `sem_post(&barber)` : Signals that the barber is ready to cut hair for the customer.
- `sem_post(&access_seats)` : Unlocks the mutex, allowing other threads to access `free_seats` .
- `sleep(rand() % 3 + 1)` : Simulates haircutting time (1–3 seconds).
- `return NULL` : Included for completeness, though the loop is infinite.

4. Customer Thread Function

```c
void *customer_thread(void *arg) {
    int id = *(int *)arg;
    sem_wait(&access_seats);     // Try to get access to seats
    if(free_seats > 0) {
        free_seats--;            // There's a free chair
        printf("Customer %d is waiting. Seats left: %d\n", id, free_seats);
        sem_post(&customers);    // Notify barber there's a customer
        sem_post(&access_seats); // Release seat access
        sem_wait(&barber);       // Wait for barber to be ready
        printf("Customer %d is getting a haircut\n", id);
    }
    else {
        printf("Customer %d left - no chairs available\n", id);
        sem_post(&access_seats); // Release seat access
    }
    return NULL;
}
```

- **Purpose**: Defines the behavior of a customer who either waits for a haircut or leaves if no chairs are available.
- **Step-by-Step**:
  - `int id = *(int *)arg` : Extracts the customer's ID from the thread argument.
  - `sem_wait(&access_seats)` : Locks the mutex to check/modify `free_seats` .
  - **If `free_seats > 0` :**
    - `free_seats--` : Reserves a chair by decrementing available seats.
    - `printf("Customer %d is waiting. Seats left: %d\n", id, free_seats)` : Logs that the customer is waiting.
    - `sem_post(&customers)` : Increments the `customers` semaphore, waking the barber if sleeping.
    - `sem_post(&access_seats)` : Unlocks the mutex.
    - `sem_wait(&barber)` : Waits for the barber to signal readiness.
    - `printf("Customer %d is getting a haircut\n", id)` : Logs that the haircut is happening.
  - **Else (no chairs)**:
    - `printf("Customer %d left - no chairs available\n", id)` : Logs that the customer leaves due to no available chairs.
    - `sem_post(&access_seats)` : Unlocks the mutex.
  - `return NULL` : Indicates thread completion.

5. Main Function

```c
int main() {
    pthread_t barber_tid;
    pthread_t customer_tids[15];
    int customer_ids[15];
    sem_init(&customers, 0, 0);
    sem_init(&barber, 0, 0);
    sem_init(&access_seats, 0, 1);
    pthread_create(&barber_tid, NULL, barber_thread, NULL);
    for(int i = 0; i < 15; i++) {
        customer_ids[i] = i + 1;
        pthread_create(&customer_tids[i], NULL, customer_thread, &customer_ids[i]);
        sleep(rand() % 2);
    }
    for(int i = 0; i < 15; i++) {
        pthread_join(customer_tids[i], NULL);
    }
    sem_destroy(&customers);
    sem_destroy(&barber);
    sem_destroy(&access_seats);
    return 0;
}
```

- **Purpose**: Initializes the system, creates threads, and manages cleanup.
- **Step-by-Step**:
  - `pthread_t barber_tid, customer_tids[15]` : Declares thread IDs for one barber and 15 customers.
  - `int customer_ids[15]` : Array to store customer IDs (1–15).
  - `sem_init(&customers, 0, 0)` : Initializes `customers` to 0 (no waiting customers initially).
  - `sem_init(&barber, 0, 0)` : Initializes `barber` to 0 (barber not ready initially).
  - `sem_init(&access_seats, 0, 1)` : Initializes `access_seats` mutex to 1 (unlocked).
  - `pthread_create(&barber_tid, NULL, barber_thread, NULL)` : Starts the barber thread.
  - **Customer Loop**:
    - Sets `customer_ids[i] = i + 1` for unique IDs (1–15).
    - `pthread_create(&customer_tids[i], NULL, customer_thread, &customer_ids[i])` : Creates a customer thread, passing its ID.
    - `sleep(rand() % 2)` : Introduces random delays (0–1 seconds) between customer arrivals.
  - `pthread_join(customer_tids[i], NULL)` : Waits for all customer threads to complete.
  - `sem_destroy` : Frees the semaphores.
  - `return 0` : Indicates successful termination.
  - **Note**: The barber thread runs indefinitely (infinite loop), so it’s not joined. In practice, you’d need a mechanism to terminate it gracefully.

6. Synchronization Mechanism

- **Semaphores**:
  - `customers` : Tracks waiting customers. The barber waits (`sem_wait`) when no customers are present.

    - `barber`: Signals when the barber is ready to cut hair. Customers wait until the barber is available.
    - `access_seats`: A mutex ensuring exclusive access to `free_seats` to prevent race conditions.
- **Shared Resource**: `free_seats` tracks available chairs (initially 5). It's decremented when a customer sits and incremented when the barber serves a customer.
- **Logic**:
    - Customers check for free chairs and wait if available, otherwise leave.
    - The barber serves one customer at a time, freeing a chair after each haircut.
    - The semaphores ensure the barber sleeps when no customers are waiting and customers wait or leave when appropriate.

7. Program Flow

- **Initialization**: Semaphores are set up, and the barber thread starts (barber sleeps initially, as `customers == 0`).
- **Customer Arrivals**: 15 customers arrive with random delays (0–1 seconds). Each:
    - Checks for a free chair.
    - If available, sits, notifies the barber, and waits for the haircut.
    - If no chairs, leaves immediately.
- **Barber Operation**: The barber wakes when a customer arrives, cuts hair (1–3 seconds), and signals readiness, repeating indefinitely.
- **Termination**: The program waits for all customer threads to finish, then cleans up semaphores. The barber thread is not terminated (infinite loop).

8. Sample Output

```
Customer 1 is waiting. Seats left: 4
Barber is cutting hair. Seats available: 5
Customer 1 is getting a haircut
Customer 2 is waiting. Seats left: 4
Customer 3 is waiting. Seats left: 3
Barber is cutting hair. Seats available: 4
Customer 2 is getting a haircut
Customer 4 left - no chairs available
...
```

- Output varies due to random sleep times and thread scheduling, but it shows synchronized interactions:
    - Customers wait if chairs are available, or leave if not.
    - The barber cuts hair for one customer at a time, updating seat availability.

9. Key Points

- **Problem Solved**: The Sleeping Barber Problem models a limited-resource system (chairs) with a single server (barber) and multiple clients (customers).
- **Synchronization**:
    - `access_seats` prevents race conditions on `free_seats`.
    - `customers` ensures the barber sleeps when no customers are waiting.
    - `barber` ensures customers wait for the barber to be ready.
- **Correctness**: The program avoids:

- Overfilling the waiting area (max 5 customers).
- Race conditions on shared resources.
- Deadlocks, as semaphores are used correctly.

---

2. Write a program to implement the concept of the dining philosopher problem.

 What is the Dining Philosophers Problem?

- Imagine 5 philosophers sitting around a table.
- Each has a plate of food and needs 2 chopsticks to eat (left and right).
- There are only 5 chopsticks shared between them.
- Philosophers alternate between thinking and eating.
- Problem: Avoid deadlock (where everyone is waiting forever) and starvation (some never get to eat).

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

#define NUM_PHILOSOPHERS 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2

sem_t mutex;
sem_t chopsticks[NUM_PHILOSOPHERS];
int state[NUM_PHILOSOPHERS];

void *philosopher(void *num);
void take_chopsticks(int);
void put_chopsticks(int);
void test(int);

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_numbers[NUM_PHILOSOPHERS];

    // Initialize semaphores
    sem_init(&mutex, 0, 1);
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_init(&chopsticks[i], 0, 1);
        philosopher_numbers[i] = i;
    }

    // Create philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_create(&philosophers[i], NULL, philosopher, &philosopher_numbers[i]);
    }
```

```c
    // Wait for threads to finish (though they run indefinitely)
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Clean up semaphores
    sem_destroy(&mutex);
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_destroy(&chopsticks[i]);
    }

    return 0;
}

void *philosopher(void *num) {
    int *philosopher_num = (int *)num;
    int i = *philosopher_num;

    while (1) {
        printf("Philosopher %d is thinking\n", i);
        sleep(1); // Thinking time

        take_chopsticks(i);

        printf("Philosopher %d is eating\n", i);
        sleep(1); // Eating time

        put_chopsticks(i);
    }
}

void take_chopsticks(int i) {
    sem_wait(&mutex);
    state[i] = HUNGRY;
    printf("Philosopher %d is hungry\n", i);
    test(i);
    sem_post(&mutex);
    sem_wait(&chopsticks[i]);
}

void test(int i) {
    if (state[i] == HUNGRY &&
        state[(i + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS] != EATING &&
        state[(i + 1) % NUM_PHILOSOPHERS] != EATING) {

        state[i] = EATING;
        sem_post(&chopsticks[i]);
    }
}

void put_chopsticks(int i) {
```

```
    sem_wait(&mutex);
    state[i] = THINKING;
    printf("Philosopher %d has finished eating\n", i);

    // Notify neighbors
    test((i + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS);
    test((i + 1) % NUM_PHILOSOPHERS);

    sem_post(&mutex);
}
```

Step-by-Step Explanation

1. Header Files and Preprocessor Directives

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define NUM_PHILOSOPHERS 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
```

- **Purpose**: Include necessary libraries and define constants.
- **Details**:
  - `<stdio.h>`: For input/output operations (`printf`).
  - `<pthread.h>`: For POSIX threads to create philosopher threads.
  - `<semaphore.h>`: For semaphore operations to manage synchronization.
  - `<unistd.h>`: For the `sleep` function to simulate thinking/eating time.
  - `#define NUM_PHILOSOPHERS 5`: Sets the number of philosophers (and chopsticks) to 5.
  - `#define THINKING 0`, `HUNGRY 1`, `EATING 2`: Constants representing philosopher states.

2. Global Variables

```
sem_t mutex;
sem_t chopsticks[NUM_PHILOSOPHERS];
int state[NUM_PHILOSOPHERS];
```

- **Purpose**: Define synchronization primitives and shared state.
- **Details**:
  - `mutex`: A binary semaphore initialized to 1, used for mutual exclusion when accessing the `state` array.
  - `chopsticks[NUM_PHILOSOPHERS]`: An array of semaphores, one per chopstick, each initialized to 1, representing the availability of chopsticks.
  - `state[NUM_PHILOSOPHERS]`: An array tracking each philosopher's state (`THINKING`, `HUNGRY`, or `EATING`).

3. Main Function

```c
int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_numbers[NUM_PHILOSOPHERS];
    sem_init(&mutex, 0, 1);
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_init(&chopsticks[i], 0, 1);
        philosopher_numbers[i] = i;
    }
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_create(&philosophers[i], NULL, philosopher, &philosopher_numbers[i]);
    }
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }
    sem_destroy(&mutex);
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_destroy(&chopsticks[i]);
    }
    return 0;
}
```

- **Purpose**: Initializes the system, creates philosopher threads, and handles cleanup.
- **Step-by-Step**:
  - `pthread_t philosophers[NUM_PHILOSOPHERS]` : Declares an array to store thread IDs for 5 philosophers.
  - `int philosopher_numbers[NUM_PHILOSOPHERS]` : Stores philosopher IDs (0 to 4).
  - `sem_init(&mutex, 0, 1)` : Initializes the mutex semaphore to 1 (unlocked).
  - **Loop**: Initializes each `chopsticks[i]` semaphore to 1 (each chopstick is initially available) and sets `philosopher_numbers[i] = i` .
  - **Loop**: Creates 5 philosopher threads, passing the `philosopher` function and the address of each philosopher's ID.
  - **Loop**: Joins all philosopher threads, waiting for them to finish (though they run indefinitely due to the infinite loop in `philosopher`).
  - **Cleanup**: Destroys the `mutex` and `chopsticks` semaphores.
  - `return 0` : Indicates successful termination (though not reached in practice due to infinite loops).
  - **Note**: The threads run indefinitely, so `pthread_join` will block unless a termination mechanism is added.

4. Philosopher Function

```c
void *philosopher(void *num) {
    int *philosopher_num = (int *)num;
    int i = *philosopher_num;
    while (1) {
        printf("Philosopher %d is thinking\n", i);
        sleep(1); // Thinking time
        take_chopsticks(i);
        printf("Philosopher %d is eating\n", i);
        sleep(1); // Eating time
```

```
        put_chopsticks(i);
    }
}
```

- **Purpose**: Defines the behavior of a philosopher, cycling between thinking, attempting to eat, and eating.
- **Step-by-Step**:
  - `int *philosopher_num = (int *)num; int i = *philosopher_num` : Extracts the philosopher's ID (0 to 4) from the thread argument.
  - **Infinite Loop**:
    - `printf("Philosopher %d is thinking\n", i)` : Logs that the philosopher is thinking.
    - `sleep(1)` : Simulates thinking for 1 second.
    - `take_chopsticks(i)` : Attempts to acquire chopsticks to eat.
    - `printf("Philosopher %d is eating\n", i)` : Logs that the philosopher is eating.
    - `sleep(1)` : Simulates eating for 1 second.
    - `put_chopsticks(i)` : Releases chopsticks and checks if neighbors can eat.

5. Take Chopsticks Function

```
void take_chopsticks(int i) {
    sem_wait(&mutex);
    state[i] = HUNGRY;
    printf("Philosopher %d is hungry\n", i);
    test(i);
    sem_post(&mutex);
    sem_wait(&chopsticks[i]);
}
```

- **Purpose**: Handles a philosopher's attempt to acquire chopsticks to eat.
- **Step-by-Step**:
  - `sem_wait(&mutex)` : Locks the mutex to safely modify the `state` array.
  - `state[i] = HUNGRY` : Marks the philosopher as hungry.
  - `printf("Philosopher %d is hungry\n", i)` : Logs the hungry state.
  - `test(i)` : Checks if the philosopher can eat (i.e., both chopsticks are available and neighbors are not eating).
  - `sem_post(&mutex)` : Unlocks the mutex.
  - `sem_wait(&chopsticks[i])` : Waits for the philosopher's semaphore to be signaled (set by `test` when eating is possible).

6. Test Function

```
void test(int i) {
    if (state[i] == HUNGRY &&
        state[(i + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS] != EATING &&
        state[(i + 1) % NUM_PHILOSOPHERS] != EATING) {
        state[i] = EATING;
        sem_post(&chopsticks[i]);
    }
}
```

- **Purpose**: Determines if a philosopher can eat by checking the availability of chopsticks.
- **Step-by-Step**:
  - **Condition**: Checks if:
    - The philosopher is `HUNGRY`.
    - The left neighbor (`(i + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS`) is not `EATING`.
    - The right neighbor (`(i + 1) % NUM_PHILOSOPHERS`) is not `EATING`.
  - If true:
    - `state[i] = EATING`: Marks the philosopher as eating.
    - `sem_post(&chopsticks[i])`: Signals the philosopher's semaphore, allowing them to proceed from `sem_wait(&chopsticks[i])` in `take_chopsticks`.

7. Put Chopsticks Function

```
void put_chopsticks(int i) {
    sem_wait(&mutex);
    state[i] = THINKING;
    printf("Philosopher %d has finished eating\n", i);
    test((i + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS);
    test((i + 1) % NUM_PHILOSOPHERS);
    sem_post(&mutex);
}
```

- **Purpose**: Releases chopsticks after eating and checks if neighbors can eat.
- **Step-by-Step**:
  - `sem_wait(&mutex)`: Locks the mutex to modify `state`.
  - `state[i] = THINKING`: Marks the philosopher as thinking (done eating).
  - `printf("Philosopher %d has finished eating\n", i)`: Logs the completion of eating.
  - `test((i + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS)`: Checks if the left neighbor can eat.
  - `test((i + 1) % NUM_PHILOSOPHERS)`: Checks if the right neighbor can eat.
  - `sem_post(&mutex)`: Unlocks the mutex.

8. Synchronization Mechanism

- **Semaphores**:
  - `mutex`: Ensures exclusive access to the `state` array, preventing race conditions.
  - `chopsticks[NUM_PHILOSOPHERS]`: Each semaphore represents a philosopher's ability to eat (not the chopsticks directly). A philosopher waits on their own semaphore, which is signaled when they can eat.
- **State Array**: Tracks whether each philosopher is `THINKING`, `HUNGRY`, or `EATING`.
- **Deadlock Avoidance**: The `test` function ensures that a philosopher only eats when both neighbors are not eating, preventing all philosophers from picking up one chopstick and causing a deadlock.
- **Resource Management**: Each philosopher requires two chopsticks (implicitly modeled by checking neighbor states). The `chopsticks` semaphore for a philosopher is signaled only when both chopsticks are available.

9. Program Flow

- **Initialization**: The `mutex` is set to 1, and each `chopsticks[i]` semaphore is set to 1. The `state` array is implicitly initialized to 0 (`THINKING`).
- **Thread Creation**: Five philosopher threads are created, each running the `philosopher` function with a unique ID (0–4).
- **Philosopher Behavior**:
  - Each philosopher cycles through thinking (1 second), becoming hungry, attempting to eat, eating (1 second), and releasing chopsticks.
  - The `test` function ensures that no two adjacent philosophers eat simultaneously.
- **Execution**: The threads run indefinitely, with philosophers thinking, getting hungry, eating, and notifying neighbors in a synchronized manner.
- **Termination**: The program doesn't terminate naturally due to the infinite loops. In practice, a signal (e.g., Ctrl+C) or a fixed iteration count would be needed.
- **Cleanup**: Semaphores are destroyed (though not reached due to infinite loops).

10. Sample Output

```
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 0 is thirsty
Philosopher 0 is eating
Philosopher 1 is thirsty
Philosopher 2 is thirsty
Philosopher 0 has finished eating
Philosopher 2 is eating
...
```

- Output varies due to thread scheduling and sleep times, but it shows philosophers transitioning between states, with no adjacent philosophers eating simultaneously.

11. Key Points

- **Problem Solved**: The Dining Philosophers Problem demonstrates resource contention (chopsticks) and synchronization, avoiding deadlock and starvation.
- **Synchronization**:
  - The `mutex` protects the `state` array.
  - The `chopsticks` semaphores ensure a philosopher waits until they can eat safely.
  - The `test` function prevents deadlock by allowing a philosopher to eat only when both chopsticks are available (neighbors not eating).
- **Correctness**: The program ensures:
  - No two adjacent philosophers eat at the same time.
  - Deadlock is avoided by conditional eating based on neighbor states.
  - Mutual exclusion for state updates.

---

3. Write a program to simulate the following scheduling algorithm.

- **FCFS**

- **SJF**
- **Round Robin**

For each of the algorithm finds out turnaround time and waiting time.

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

#define MAX_PROCESSES 10

typedef struct {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
} Process;

void fcfs(Process processes[], int n);
void sjf(Process processes[], int n);
void priority(Process processes[], int n);
void round_robin(Process processes[], int n, int quantum);
void print_results(Process processes[], int n);

int main() {
    Process processes[MAX_PROCESSES];
    int n, choice, quantum;

    printf("Enter number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", &n);

    for(int i = 0; i < n; i++) {
        printf("\nProcess %d:\n", i+1);
        processes[i].pid = i+1;
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
        printf("Priority: ");
        scanf("%d", &processes[i].priority);
        processes[i].remaining_time = processes[i].burst_time;
    }

    printf("\nScheduling Algorithms:\n");
    printf("1. FCFS\n2. SJF\n3. Priority\n4. Round Robin\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch(choice) {
        case 1:
```

```
            fcfs(processes, n);
            break;
        case 2:
            sjf(processes, n);
            break;
        case 3:
            priority(processes, n);
            break;
        case 4:
            printf("Enter time quantum: ");
            scanf("%d", &quantum);
            round_robin(processes, n, quantum);
            break;
        default:
            printf("Invalid choice\n");
            return 1;
    }

    print_results(processes, n);
    return 0;
}

void fcfs(Process processes[], int n) {
    // Sort by arrival time
    for(int i = 0; i < n-1; i++) {
        for(int j = 0; j < n-i-1; j++) {
            if(processes[j].arrival_time > processes[j+1].arrival_time) {
                Process temp = processes[j];
                processes[j] = processes[j+1];
                processes[j+1] = temp;
            }
        }
    }

    int current_time = 0;
    for(int i = 0; i < n; i++) {
        if(current_time < processes[i].arrival_time)
            current_time = processes[i].arrival_time;

        processes[i].waiting_time = current_time - processes[i].arrival_time;
        current_time += processes[i].burst_time;
        processes[i].turnaround_time = processes[i].waiting_time +
processes[i].burst_time;
    }
}

void sjf(Process processes[], int n) {
    int completed = 0, current_time = 0;

    while(completed < n) {
        int shortest = -1;
        int min_burst = 9999;
```

```
        for(int i = 0; i < n; i++) {
            if(processes[i].arrival_time <= current_time &&
               processes[i].remaining_time > 0 &&
               processes[i].remaining_time < min_burst) {
                min_burst = processes[i].remaining_time;
                shortest = i;
            }
        }

        if(shortest == -1) {
            current_time++;
            continue;
        }

        processes[shortest].remaining_time = 0;
        processes[shortest].turnaround_time = current_time +
processes[shortest].burst_time - processes[shortest].arrival_time;
        processes[shortest].waiting_time = processes[shortest].turnaround_time -
processes[shortest].burst_time;
        current_time += processes[shortest].burst_time;
        completed++;
    }
}

void priority(Process processes[], int n) {
    int completed = 0, current_time = 0;

    while(completed < n) {
        int highest_priority = -1;
        int min_priority = 9999;

        for(int i = 0; i < n; i++) {
            if(processes[i].arrival_time <= current_time &&
               processes[i].remaining_time > 0 &&
               processes[i].priority < min_priority) {
                min_priority = processes[i].priority;
                highest_priority = i;
            }
        }

        if(highest_priority == -1) {
            current_time++;
            continue;
        }

        processes[highest_priority].remaining_time = 0;
        processes[highest_priority].turnaround_time = current_time +
processes[highest_priority].burst_time - processes[highest_priority].arrival_time;
        processes[highest_priority].waiting_time =
processes[highest_priority].turnaround_time - processes[highest_priority].burst_time;
        current_time += processes[highest_priority].burst_time;
```

```
        completed++;
    }
}

void round_robin(Process processes[], int n, int quantum) {
    int remaining = n;
    int current_time = 0;

    while(remaining > 0) {
        for(int i = 0; i < n; i++) {
            if(processes[i].arrival_time <= current_time &&
processes[i].remaining_time > 0) {
                int exec_time = (processes[i].remaining_time > quantum) ? quantum :
processes[i].remaining_time;

                processes[i].remaining_time -= exec_time;
                current_time += exec_time;

                if(processes[i].remaining_time == 0) {
                    remaining--;
                    processes[i].turnaround_time = current_time -
processes[i].arrival_time;
                    processes[i].waiting_time = processes[i].turnaround_time -
processes[i].burst_time;
                }
            }
        }
    }
}

void print_results(Process processes[], int n) {
    float avg_wait = 0, avg_turnaround = 0;

    printf("\nPID\tArrival\tBurst\tPriority\tWaiting\tTurnaround\n");
    for(int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\n",
               processes[i].pid,
               processes[i].arrival_time,
               processes[i].burst_time,
               processes[i].priority,
               processes[i].waiting_time,
               processes[i].turnaround_time);

        avg_wait += processes[i].waiting_time;
        avg_turnaround += processes[i].turnaround_time;
    }

    printf("\nAverage Waiting Time: %.2f\n", avg_wait/n);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround/n);
}
```

Step-by-Step Explanation

1. Header Files and Preprocessor Directives

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_PROCESSES 10
```

- **Purpose**: Include necessary libraries and define the maximum number of processes.
- **Details**:
  - `<stdio.h>`: For input/output (`printf`, `scanf`).
  - `<stdlib.h>`: For general utilities (though not explicitly used here).
  - `<string.h>`: Included but unused in this program.
  - `#define MAX_PROCESSES 10`: Limits the number of processes to 10.

2. Process Structure

```
typedef struct {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
} Process;
```

- **Purpose**: Defines a structure to represent a process.
- **Details**:
  - `pid`: Process ID.
  - `arrival_time`: Time when the process arrives.
  - `burst_time`: Total CPU time required by the process.
  - `priority`: Priority value (lower value = higher priority).
  - `remaining_time`: Tracks remaining burst time (used in SJF and RR).
  - `waiting_time`: Time the process waits before execution completes.
  - `turnaround_time`: Total time from arrival to completion.

3. Main Function

```
int main() {
    Process processes[MAX_PROCESSES];
    int n, choice, quantum;
    printf("Enter number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", &n);
    for(int i = 0; i < n; i++) {
        printf("\nProcess %d:\n", i+1);
        processes[i].pid = i+1;
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
```

```
        printf("Priority: ");
        scanf("%d", &processes[i].priority);
        processes[i].remaining_time = processes[i].burst_time;
    }
    printf("\nScheduling Algorithms:\n");
    printf("1. FCFS\n2. SJF\n3. Priority\n4. Round Robin\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    switch(choice) {
        case 1: fcfs(processes, n); break;
        case 2: sjf(processes, n); break;
        case 3: priority(processes, n); break;
        case 4:
            printf("Enter time quantum: ");
            scanf("%d", &quantum);
            round_robin(processes, n, quantum);
            break;
        default: printf("Invalid choice\n"); return 1;
    }
    print_results(processes, n);
    return 0;
}
```

- **Purpose**: Handles user input, selects the scheduling algorithm, and displays results.
- **Step-by-Step**:
  - Declares an array `processes[MAX_PROCESSES]` to store process information.
  - Prompts user for the number of processes (`n`) and ensures it's within `MAX_PROCESSES`.
  - **Input Loop**: For each process:
    - Sets `pid` to `i+1` (1-based indexing).
    - Reads `arrival_time`, `burst_time`, and `priority` from user input.
    - Initializes `remaining_time` to `burst_time`.
  - Displays a menu of scheduling algorithms (1: FCFS, 2: SJF, 3: Priority, 4: Round Robin).
  - Reads user's choice and, for Round Robin, prompts for the time quantum.
  - Uses a `switch` statement to call the appropriate scheduling algorithm.
  - Calls `print_results` to display the results.
  - Returns 0 for successful execution.

4. FCFS (First-Come-First-Serve) Function

```
void fcfs(Process processes[], int n) {
    for(int i = 0; i < n-1; i++) {
        for(int j = 0; j < n-i-1; j++) {
            if(processes[j].arrival_time > processes[j+1].arrival_time) {
                Process temp = processes[j];
                processes[j] = processes[j+1];
                processes[j+1] = temp;
            }
        }
    }
```

```
    int current_time = 0;
    for(int i = 0; i < n; i++) {
        if(current_time < processes[i].arrival_time)
            current_time = processes[i].arrival_time;
        processes[i].waiting_time = current_time - processes[i].arrival_time;
        current_time += processes[i].burst_time;
        processes[i].turnaround_time = processes[i].waiting_time +
processes[i].burst_time;
    }
}
```

- **Purpose**: Implements non-preemptive FCFS scheduling (processes run in order of arrival).
- **Step-by-Step**:
  - **Sorting**: Uses bubble sort to order processes by `arrival_time`.
  - Initializes `current_time` to 0 (system clock).
  - **Loop**: For each process:
    - If `current_time` is less than `arrival_time`, advances `current_time` to `arrival_time` (CPU idles until the process arrives).
    - `waiting_time = current_time - arrival_time`: Time the process waits after arriving.
    - `current_time += burst_time`: Advances clock by the process's execution time.
    - `turnaround_time = waiting_time + burst_time`: Total time from arrival to completion.
  - **Note**: Modifies the original `processes` array by sorting, which affects output order.

5. SJF (Shortest Job First) Function

```
void sjf(Process processes[], int n) {
    int completed = 0, current_time = 0;
    while(completed < n) {
        int shortest = -1;
        int min_burst = 9999;
        for(int i = 0; i < n; i++) {
            if(processes[i].arrival_time <= current_time &&
               processes[i].remaining_time > 0 &&
               processes[i].remaining_time < min_burst) {
                min_burst = processes[i].remaining_time;
                shortest = i;
            }
        }
        if(shortest == -1) {
            current_time++;
            continue;
        }
        processes[shortest].remaining_time = 0;
        processes[shortest].turnaround_time = current_time +
processes[shortest].burst_time - processes[shortest].arrival_time;
        processes[shortest].waiting_time = processes[shortest].turnaround_time -
```

```
processes[shortest].burst_time;
        current_time += processes[shortest].burst_time;
        completed++;
    }
}
```

- **Purpose**: Implements non-preemptive SJF scheduling (shortest burst time among available processes runs next).
- **Step-by-Step**:
  - Initializes `completed` (number of finished processes) and `current_time` to 0.
  - **While Loop**: Continues until all processes are completed (`completed < n`).
    - Searches for the process with the shortest `remaining_time` among those that have arrived (`arrival_time <= current_time`) and are not completed (`remaining_time > 0`).
    - If no process is found (`shortest == -1`), increments `current_time` (CPU idles).
    - For the selected process:
      - Sets `remaining_time` to 0 (process completes).
      - Calculates `turnaround_time = current_time + burst_time - arrival_time` .
      - Calculates `waiting_time = turnaround_time - burst_time` .
      - Advances `current_time` by `burst_time` .
      - Increments `completed` .

6. Priority Scheduling Function

```
void priority(Process processes[], int n) {
    int completed = 0, current_time = 0;
    while(completed < n) {
        int highest_priority = -1;
        int min_priority = 9999;
        for(int i = 0; i < n; i++) {
            if(processes[i].arrival_time <= current_time &&
               processes[i].remaining_time > 0 &&
               processes[i].priority < min_priority) {
                min_priority = processes[i].priority;
                highest_priority = i;
            }
        }
        if(highest_priority == -1) {
            current_time++;
            continue;
        }
        processes[highest_priority].remaining_time = 0;
        processes[highest_priority].turnaround_time = current_time +
processes[highest_priority].burst_time - processes[highest_priority].arrival_time;
        processes[highest_priority].waiting_time =
processes[highest_priority].turnaround_time - processes[highest_priority].burst_time;
        current_time += processes[highest_priority].burst_time;
        completed++;
```

```
    }
}
```

- **Purpose**: Implements non-preemptive priority scheduling (lowest priority value = highest priority).
- **Step-by-Step**:
  - Similar to SJF but selects the process with the lowest `priority` value among those that have arrived and are not completed.
  - Initializes `completed` and `current_time` to 0.
  - **While Loop**:
    - Finds the process with the minimum `priority` (highest priority) among available processes.
    - If none found, increments `current_time`.
    - For the selected process:
      - Sets `remaining_time` to 0.
      - Calculates `turnaround_time` and `waiting_time`.
      - Advances `current_time` by `burst_time`.
      - Increments `completed`.

7. Round Robin Function

```
void round_robin(Process processes[], int n, int quantum) {
    int remaining = n;
    int current_time = 0;
    while(remaining > 0) {
        for(int i = 0; i < n; i++) {
            if(processes[i].arrival_time <= current_time &&
processes[i].remaining_time > 0) {
                int exec_time = (processes[i].remaining_time > quantum) ? quantum :
processes[i].remaining_time;
                processes[i].remaining_time -= exec_time;
                current_time += exec_time;
                if(processes[i].remaining_time == 0) {
                    remaining--;
                    processes[i].turnaround_time = current_time -
processes[i].arrival_time;
                    processes[i].waiting_time = processes[i].turnaround_time -
processes[i].burst_time;
                }
            }
        }
    }
}
```

- **Purpose**: Implements Round Robin scheduling with a user-specified time quantum.
- **Step-by-Step**:
  - Initializes `remaining` (number of unfinished processes) to `n` and `current_time` to 0.
  - **While Loop**: Continues until all processes are completed (`remaining > 0`).
    - **Loop**: Iterates through processes in order.

        - For each process that has arrived and has `remaining_time > 0`:
            - Executes for `exec_time = min(quantum, remaining_time)`.
            - Reduces `remaining_time` by `exec_time`.
            - Advances `current_time` by `exec_time`.
            - If `remaining_time == 0`, calculates `turnaround_time` and `waiting_time`, and decrements `remaining`.
    - **Note**: Processes are not sorted, so they are checked in input order, simulating a circular queue.

8. Print Results Function

```
void print_results(Process processes[], int n) {
    float avg_wait = 0, avg_turnaround = 0;
    printf("\nPID\tArrival\tBurst\tPriority\tWaiting\tTurnaround\n");
    for(int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\n",
               processes[i].pid,
               processes[i].arrival_time,
               processes[i].burst_time,
               processes[i].priority,
               processes[i].waiting_time,
               processes[i].turnaround_time);
        avg_wait += processes[i].waiting_time;
        avg_turnaround += processes[i].turnaround_time;
    }
    printf("\nAverage Waiting Time: %.2f\n", avg_wait/n);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround/n);
}
```

- **Purpose**: Displays the results for each process and computes averages.
- **Step-by-Step**:
    - Initializes `avg_wait` and `avg_turnaround` to 0.
    - Prints a table with columns: PID, Arrival Time, Burst Time, Priority, Waiting Time, Turnaround Time.
    - Accumulates `waiting_time` and `turnaround_time` for averaging.
    - Prints average waiting time (`avg_wait/n`) and average turnaround time (`avg_turnaround/n`).

9. Program Flow

- **Input**: User specifies the number of processes and their attributes (arrival time, burst time, priority).
- **Algorithm Selection**: User chooses an algorithm (1–4). For Round Robin, a time quantum is also input.
- **Execution**: The chosen algorithm updates `waiting_time` and `turnaround_time` for each process.
- **Output**: Results are printed in a table, followed by average waiting and turnaround times.

- **Formulas**:
  - `turnaround_time = completion_time - arrival_time`
  - `waiting_time = turnaround_time - burst_time`
- Example input:

```
Enter number of processes (max 10): 4
Process 1:
Arrival Time: 0
Burst Time: 5
Priority: 3
Process 2:
Arrival Time: 1
Burst Time: 3
Priority: 1
Process 3:
Arrival Time: 2
Burst Time: 8
Priority: 4
Process 4:
Arrival Time: 3
Burst Time: 6
Priority: 2
Scheduling Algorithms:
1. FCFS
2. SJF
3. Priority
4. Round Robin
Enter your choice: 1
```

- Output for FCFS:

```
PID     Arrival Burst   Priority        Waiting Turnaround
1       0       5       3               0       5
2       1       3       1               4       7
3       2       8       4               6       14
4       3       6       2               13      19
Average Waiting Time: 5.75
Average Turnaround Time: 11.25
```

10. Key Points

- **Algorithms**:
  - **FCFS**: Executes processes in arrival order, simple but may lead to high waiting times.
  - **SJF**: Non-preemptive, prioritizes shortest burst time, minimizing average waiting time but may cause starvation for long jobs.
  - **Priority**: Non-preemptive, prioritizes lowest priority value, may also cause starvation for low-priority processes.
  - **Round Robin**: Preemptive, ensures fairness with a time quantum, suitable for time-sharing systems.

- **Correctness**: The program correctly calculates waiting and turnaround times, handling arrival times and process completion.
