READER WRITER PROBLEM:

```
import threading
import time

# Shared data
hours = 0
minutes = 0
seconds = 0

# Semaphores
sem_reader = threading.Semaphore(0)  # Reader semaphore starts at 0
sem_writer = threading.Semaphore(1)  # Writer semaphore starts at 1

# Writer function (thread)
def writer():
    global hours, minutes, seconds
    while True:
        # Wait for the writer semaphore
        sem_writer.acquire()

        # Simulate time update
        seconds += 1
        if seconds == 60:
            seconds = 0
            minutes += 1
        if minutes == 60:
            minutes = 0
            hours += 1
        if hours == 24:
            hours = 0

        print(f"Writer updated time to {hours:02d}:{minutes:02d}:{seconds:02d}")

        # Signal the reader semaphore
        sem_reader.release()

        # Sleep to simulate delay
        time.sleep(1)

# Reader function (thread)
def reader():
    global hours, minutes, seconds
    while True:
        # Wait for the reader semaphore
```

```
        sem_reader.acquire()

        # Read the shared data
        print(f"Reader read time: {hours:02d}:{minutes:02d}:{seconds:02d}")

        # Signal the writer semaphore
        sem_writer.release()

        # Sleep to simulate delay
        time.sleep(1)

# Main function
if __name__ == "__main__":
    # Create threads
    writer_thread = threading.Thread(target=writer)
    reader_thread = threading.Thread(target=reader)

    # Start threads
    writer_thread.start()
    reader_thread.start()

    # Join threads (to keep the program running)
    writer_thread.join()
    reader_thread.join()
```

PRODUCER CONSUMER PROBLEM:

```
import threading
import time

# Shared buffer
BUFFER_SIZE = 100
buffer = [""] * BUFFER_SIZE  # Simulated buffer for shared data

# Semaphores
bfull = threading.Semaphore(0)  # Indicates if the buffer is full (data available to consume)
bempty = threading.Semaphore(1)  # Indicates if the buffer is empty (space available to produce)

# Define a limit for the number of iterations
ITERATION_LIMIT = 10

# Producer function
def producer():
    for i in range(ITERATION_LIMIT):
        # Wait for the buffer to be empty
```

```
        bempty.acquire()

        # Critical section: produce the data
        produced_data = f"Produced String Data {i + 1}"
        buffer[0] = produced_data  # Simulate storing in the buffer
        print(f"Producer: Produced '{produced_data}'")

        # Signal that the buffer is full (data available for consumption)
        bfull.release()

        # Sleep to simulate production time
        time.sleep(1)

# Consumer function
def consumer():
    for i in range(ITERATION_LIMIT):
        # Wait for the buffer to be full (data available)
        bfull.acquire()

        # Critical section: consume the data
        consumed_data = buffer[0]  # Simulate consuming from the buffer
        print(f"Consumer: Consumed '{consumed_data}'")

        # Clear the buffer after consuming
        buffer[0] = ""

        # Signal that the buffer is empty (ready for production again)
        bempty.release()

        # Sleep to simulate consumption time
        time.sleep(1)

# Main function
if __name__ == "__main__":
    # Create threads for producer and consumer
    producer_thread = threading.Thread(target=producer)
    consumer_thread = threading.Thread(target=consumer)

    # Start threads
    producer_thread.start()
    consumer_thread.start()

    # Wait for both threads to complete
    producer_thread.join()
```

```
    consumer_thread.join()

    print(f"Process completed after {ITERATION_LIMIT} iterations.")
```

SCHEDULING ALGORITHM PROBLEM:

a. **FCFS:**

```
# Class to represent a process
class Process:
    def __init__(self, pid, arrival_time, burst_time):
        self.pid = pid  # Process ID
        self.arrival_time = arrival_time  # Arrival time of the process
        self.burst_time = burst_time  # Burst time of the process
        self.finish_time = 0  # Finish time of the process
        self.waiting_time = 0  # Waiting time of the process
        self.turnaround_time = 0  # Turnaround time of the process

# Function to sort processes by arrival time
def sort_processes_by_arrival_time(processes):
    return sorted(processes, key=lambda x: x.arrival_time)

def main():
    # Input number of processes
    n = int(input("Enter the number of processes: "))
    processes = []

    # Input arrival time and burst time for each process
    for i in range(n):
        arrival_time, burst_time = map(int, input(f"Enter arrival time and burst time for process {i + 1}: 
").split())
        processes.append(Process(pid=i + 1, arrival_time=arrival_time, burst_time=burst_time))

    # Sort processes by arrival time
    processes = sort_processes_by_arrival_time(processes)

    # Calculate finish time, waiting time, and turnaround time
    current_time = 0
    for process in processes:
        if current_time < process.arrival_time:
            current_time = process.arrival_time
        process.finish_time = current_time + process.burst_time
        process.turnaround_time = process.finish_time - process.arrival_time
        process.waiting_time = process.turnaround_time - process.burst_time
        current_time = process.finish_time
```

```
    # Display process information
    print("\nProcess\tArrival Time\tBurst Time\tFinish Time\tTurnaround Time\tWaiting Time")
    for process in processes:

print(f"P{process.pid}\t{process.arrival_time}\t\t{process.burst_time}\t\t{process.finish_time}\t\t{proces
s.turnaround_time}\t\t{process.waiting_time}")

    # Display Gantt chart
    print("\nGantt Chart:")
    print("|", end="")
    for process in processes:
        print(f" P{process.pid} |", end="")
    print()

    print("0", end="")
    current_time = 0
    for process in processes:
        if current_time < process.arrival_time:
            current_time = process.arrival_time
        current_time += process.burst_time
        print(f"\t{current_time}", end="")
    print()

if __name__ == "__main__":
    main()
```

b. SJF:

```
class Process:
    def __init__(self, pid, arrival_time, burst_time):
        self.pid = pid  # Process ID
        self.arrival_time = arrival_time  # Arrival time of the process
        self.burst_time = burst_time  # Burst time of the process
        self.finish_time = 0  # Finish time of the process
        self.waiting_time = 0  # Waiting time of the process
        self.turnaround_time = 0  # Turnaround time of the process
        self.completed = False  # To track if the process is completed


# Function to find the process with the shortest burst time that has arrived
def find_shortest_job(processes, current_time):
    shortest = None
    min_burst_time = float('inf')  # Large value to find the minimum burst time
    for process in processes:
```

```
        if process.arrival_time <= current_time and not process.completed and process.burst_time < 
min_burst_time:
            min_burst_time = process.burst_time
            shortest = process
    return shortest


def main():
    # Input number of processes
    n = int(input("Enter the number of processes: "))
    processes = []

    # Input arrival time and burst time for each process
    for i in range(n):
        arrival_time, burst_time = map(int, input(f"Enter arrival time and burst time for process {i + 1}: 
").split())
        processes.append(Process(pid=i + 1, arrival_time=arrival_time, burst_time=burst_time))

    completed_processes = 0
    current_time = 0
    gantt_chart = []  # To store the Gantt chart information

    # Loop until all processes are completed
    while completed_processes < n:
        # Find the shortest job that has arrived and is not yet completed
        shortest = find_shortest_job(processes, current_time)
        if shortest is None:
            current_time += 1  # If no process has arrived yet, increment time
            continue

        # Execute the selected process
        current_time += shortest.burst_time
        shortest.finish_time = current_time
        shortest.turnaround_time = shortest.finish_time - shortest.arrival_time
        shortest.waiting_time = shortest.turnaround_time - shortest.burst_time
        shortest.completed = True
        completed_processes += 1
        gantt_chart.append(shortest)

    # Display results
    print("\nProcess\tArrival Time\tBurst Time\tFinish Time\tTurnaround Time\tWaiting Time")
    for process in processes:
```

```
print(f"P{process.pid}\t{process.arrival_time}\t\t{process.burst_time}\t\t{process.finish_time}\t\t{proces
s.turnaround_time}\t\t{process.waiting_time}")

    # Display Gantt chart
    print("\nGantt Chart:")
    print("|", end="")
    for process in gantt_chart:
        print(f" P{process.pid} |", end="")
    print()

    print("0", end="")
    current_time = 0
    for process in gantt_chart:
        current_time += process.burst_time
        print(f"\t{current_time}", end="")
    print()


if __name__ == "__main__":
    main()
```

c. PRIORITY SCHEDULING(NON-PREEMPTIVE):

```
class Process:
    def __init__(self, pid, arrival_time, burst_time, priority):
        self.pid = pid  # Process ID
        self.arrival_time = arrival_time  # Arrival time of the process
        self.burst_time = burst_time  # Burst time of the process
        self.priority = priority  # Priority of the process
        self.finish_time = 0  # Finish time of the process
        self.turnaround_time = 0  # Turnaround time of the process
        self.waiting_time = 0  # Waiting time of the process


# Function to find the process with the highest priority that has arrived
def find_highest_priority(processes, current_time):
    highest = None
    max_priority = float('inf')  # Smaller values represent higher priority
    for process in processes:
        if process.arrival_time <= current_time and process.finish_time == 0 and process.priority <
max_priority:
            max_priority = process.priority
            highest = process
    return highest
```

```
def main():
    # Input number of processes
    n = int(input("Enter the number of processes: "))
    processes = []

    # Input arrival time, burst time, and priority for each process
    for i in range(n):
        arrival_time, burst_time, priority = map(int, input(f"Enter arrival time, burst time, and priority for
process {i + 1} (lower number = higher priority): ").split())
        processes.append(Process(pid=i + 1, arrival_time=arrival_time, burst_time=burst_time,
priority=priority))

    current_time = 0
    completed_processes = 0
    gantt_chart = []

    # Loop until all processes are completed
    while completed_processes < n:
        # Find the process with the highest priority that has arrived
        highest = find_highest_priority(processes, current_time)
        if highest is None:
            current_time += 1  # If no process has arrived yet, increment time
            continue

        # Execute the selected process
        current_time += highest.burst_time
        highest.finish_time = current_time
        highest.turnaround_time = highest.finish_time - highest.arrival_time
        highest.waiting_time = highest.turnaround_time - highest.burst_time
        completed_processes += 1
        gantt_chart.append(highest)

    # Display results
    print("\nProcess\tArrival Time\tBurst Time\tPriority\tFinish Time\tTurnaround Time\tWaiting Time")
    for process in processes:

print(f"P{process.pid}\t{process.arrival_time}\t\t{process.burst_time}\t\t{process.priority}\t\t{process.fin
ish_time}\t\t{process.turnaround_time}\t\t{process.waiting_time}")

    # Display Gantt chart
    print("\nGantt Chart:")
    print("|", end="")
```

```
    for process in gantt_chart:
        print(f" P{process.pid} |", end="")
    print()

    print("0", end="")
    current_time = 0
    for process in gantt_chart:
        current_time += process.burst_time
        print(f"\t{current_time}", end="")
    print()


if __name__ == "__main__":
    main()
```

d. PRIORITY SCHEDULING(PREEMPTIVE):

```
def preemptive_priority_scheduling(processes):
    """
    Simulates preemptive priority scheduling.
    """
    # Sort processes by arrival time
    processes.sort(key=lambda x: (x["arrival_time"], x["priority"]))
    time = 0
    completed_processes = 0
    n = len(processes)

    # Initialize tracking variables
    waiting_time = [0] * n
    turnaround_time = [0] * n
    remaining_time = [process["burst_time"] for process in processes]
    current_process = -1

    # Start simulation
    while completed_processes < n:
        # Select the process with the highest priority (lowest value)
        available_processes = [
            i
            for i in range(n)
            if processes[i]["arrival_time"] <= time and remaining_time[i] > 0
        ]
        if available_processes:
            current_process = min(
                available_processes, key=lambda x: processes[x]["priority"]
            )
```

```
            remaining_time[current_process] -= 1
            time += 1

            # If the process finishes execution
            if remaining_time[current_process] == 0:
                completed_processes += 1
                finish_time = time
                turnaround_time[current_process] = (
                    finish_time - processes[current_process]["arrival_time"]
                )
                waiting_time[current_process] = (
                    turnaround_time[current_process]
                    - processes[current_process]["burst_time"]
                )
        else:
            time += 1

    # Output the results
    print("\nProcess\tArrival\tBurst\tPriority\tWaiting\tTurnaround")
    for i, process in enumerate(processes):
        print(
            f"P{i+1}\t{process['arrival_time']}\t{process['burst_time']}\t{process['priority']}\t"
            f"{waiting_time[i]}\t{turnaround_time[i]}"
        )

    # Calculate average waiting and turnaround time
    avg_waiting_time = sum(waiting_time) / n
    avg_turnaround_time = sum(turnaround_time) / n
    print(f"\nAverage Waiting Time: {avg_waiting_time:.2f}")
    print(f"Average Turnaround Time: {avg_turnaround_time:.2f}")


# Input and execution
def main():
    n = int(input("Enter the number of processes: "))
    processes = []
    for i in range(n):
        arrival_time = int(input(f"Enter arrival time for process P{i+1}: "))
        burst_time = int(input(f"Enter burst time for process P{i+1}: "))
        priority = int(input(f"Enter priority for process P{i+1} (lower value = higher priority): "))
        processes.append(
            {"arrival_time": arrival_time, "burst_time": burst_time, "priority": priority}
        )
```

```
    preemptive_priority_scheduling(processes)


if __name__ == "__main__":
    main()
```

BANKER'S DEADLOCK ALGORITHM:

```
# Maximum number of processes and resources
MAX = 10
RESOURCES = 3

# Matrices and arrays
allocation = [[0 for _ in range(RESOURCES)] for _ in range(MAX)]
max_demand = [[0 for _ in range(RESOURCES)] for _ in range(MAX)]
need = [[0 for _ in range(RESOURCES)] for _ in range(MAX)]
available = [0 for _ in range(RESOURCES)]
finished = [False for _ in range(MAX)]


# Function to calculate the need matrix
def calculate_need(n, m):
    for i in range(n):
        for j in range(m):
            need[i][j] = max_demand[i][j] - allocation[i][j]


# Function to check if the system is in a safe state
def is_safe(n, m):
    work = available[:]  # Initialize work with available resources
    finished[:] = [False for _ in range(n)]  # Mark all processes as unfinished
    safe_sequence = []  # To store the safe sequence

    count = 0
    while count < n:
        found = False  # To check if a process can be executed
        for p in range(n):
            if not finished[p]:  # If process p is not finished
                # Check if needs can be satisfied
                can_execute = all(need[p][j] <= work[j] for j in range(m))
                if can_execute:  # If the process can execute
                    # Release allocated resources
                    for j in range(m):
```

```
                work[j] += allocation[p][j]
            safe_sequence.append(p)  # Add process to safe sequence
            finished[p] = True  # Mark process as finished
            found = True
            count += 1

        if not found:
            # If no process could be executed, the system is not in a safe state
            return False, []

    # Display the safe sequence
    print("System is in a safe state.")
    print("Safe Sequence is:", " -> ".join(f"P{p}" for p in safe_sequence))
    return True, safe_sequence


def main():
    # Input the number of processes and resources
    n = int(input("Enter the number of processes: "))
    m = int(input("Enter the number of resources: "))

    # Input the allocation matrix
    print("Enter allocation matrix:")
    for i in range(n):
        allocation[i] = list(map(int, input(f"Process {i} allocation: ").split()))

    # Input the maximum demand matrix
    print("Enter maximum demand matrix:")
    for i in range(n):
        max_demand[i] = list(map(int, input(f"Process {i} max demand: ").split()))

    # Input the available resources
    print("Enter available resources:")
    global available
    available = list(map(int, input("Available resources: ").split()))

    # Calculate the need matrix
    calculate_need(n, m)

    # Check for a safe state
    safe, safe_sequence = is_safe(n, m)
    if not safe:
        print("System is not in a safe state.")
```

```
if __name__ == "__main__":
    main()
```

PAGE REPLACEMENT ALGORITHM:

a. **FIFO**

```
def fifo_page_replacement():
    # Input the number of frames
    num_frames = int(input("Enter the number of frames: "))

    # Input the number of pages
    num_pages = int(input("Enter the number of pages: "))

    # Input the page reference sequence
    pages = list(map(int, input("Enter the page reference sequence: ").split()))

    # Initialize frames and variables
    frames = [-1] * num_frames  # Frames array initialized to -1
    front = 0  # Pointer for the FIFO replacement
    page_faults = 0  # Counter for page faults

    print("\nPage Replacement Process:")

    # Process each page in the reference sequence
    for page in pages:
        # Check if the page is already in the frames
        if page not in frames:
            # Page fault occurs; replace the oldest page
            frames[front] = page
            front = (front + 1) % num_frames  # Move the pointer in a circular manner
            page_faults += 1

            # Print the current state of the frames
            print("Frames:", " ".join(str(f) if f != -1 else "-" for f in frames))

    # Output the total number of page faults
    print(f"\nTotal Page Faults: {page_faults}")

# Run the function
fifo_page_replacement()
```

b. LRU

```
def find_lru(time, n):
    """
    Function to find the position of the least recently used page.
    """
    minimum = time[0]
    pos = 0
    for i in range(1, n):
        if time[i] < minimum:
            minimum = time[i]
            pos = i
    return pos


def lru_page_replacement(pages, num_pages, num_frames):
    """
    Function to implement the LRU Page Replacement Algorithm.
    """
    # Initialize frames and time arrays
    frames = [-1] * num_frames
    time = [0] * num_frames
    count = 0
    page_faults = 0

    print("\nPage Replacement Process:")

    for i in range(num_pages):
        flag1 = flag2 = 0

        # Check if the page is already in one of the frames
        for j in range(num_frames):
            if frames[j] == pages[i]:
                count += 1
                time[j] = count  # Update the time for the accessed page
                flag1 = flag2 = 1
                break

        # If the page is not already in frames
        if flag1 == 0:
            for j in range(num_frames):
                if frames[j] == -1:  # Empty frame found
                    count += 1
                    page_faults += 1
                    frames[j] = pages[i]
```

```
                time[j] = count
                flag2 = 1
                break

        # If no empty frame was found, replace the least recently used page
        if flag2 == 0:
            pos = find_lru(time, num_frames)
            count += 1
            page_faults += 1
            frames[pos] = pages[i]
            time[pos] = count

        # Output the current state of the frames
        print("Frames:", " ".join(str(f) if f != -1 else "-" for f in frames))

    print(f"\nTotal Page Faults: {page_faults}")


# Main function to take user input and execute the algorithm
def main():
    num_frames = int(input("Enter the number of frames: "))
    num_pages = int(input("Enter the number of pages: "))
    pages = list(map(int, input("Enter the page reference sequence: ").split()))

    lru_page_replacement(pages, num_pages, num_frames)


if __name__ == "__main__":
    main()
```

c. OPTIMAL

```
def find_optimal(pages, frames, current_pos, num_pages, num_frames):
    """
    Find the position of the frame that will not be used for the longest time.
    """
    farthest = current_pos
    pos = -1

    for i in range(num_frames):
        found = False
        for j in range(current_pos + 1, num_pages):
            if frames[i] == pages[j]:
                if j > farthest:
                    farthest = j
```

```
                pos = i
            found = True
            break
        # If the page is never referenced again, return its position
        if not found:
            return i

    # If no future reference is found, return the position of the farthest page
    return 0 if pos == -1 else pos


def optimal_page_replacement(pages, num_pages, num_frames):
    """
    Implementation of the Optimal Page Replacement Algorithm.
    """
    frames = [-1] * num_frames
    page_faults = 0

    print("\nPage Replacement Process:")

    for i in range(num_pages):
        flag1 = False

        # Check if the page is already in memory
        for j in range(num_frames):
            if frames[j] == pages[i]:
                flag1 = True
                break

        # If page is not present in memory
        if not flag1:
            if -1 in frames:
                # Fill empty frames initially
                frames[frames.index(-1)] = pages[i]
            else:
                # Find the optimal page to replace
                pos = find_optimal(pages, frames, i, num_pages, num_frames)
                frames[pos] = pages[i]

            page_faults += 1

        # Output the current frame status
        print("Frames:", " ".join(str(f) if f != -1 else "-" for f in frames))
```

```
    print(f"\nTotal Page Faults = {page_faults}")


def main():
    num_frames = int(input("Enter the number of frames: "))
    num_pages = int(input("Enter the number of pages: "))
    pages = list(map(int, input("Enter the page reference sequence: ").split()))

    optimal_page_replacement(pages, num_pages, num_frames)


if __name__ == "__main__":
    main()
```