CS558

ASSIGNMENT 2

ADITYA - 234101004
JINTU - 234101022
MUNESH - 244101028

Q1. Bus Station Synchronization

The bus station problem is a synchronization challenge that demonstrates the producer-consumer pattern with a twist. It requires coordination between multiple passenger threads (producers) and a bus thread (consumer) with capacity constraints.

The solution employs two key synchronization primitives:

1. Mutex (`mtx`): Ensures mutual exclusion when accessing shared variables

2. Condition Variables:

- `cv_gate_open`: Signals when passengers can board

- `cv_bus_depart`: Signals when the bus can depart

The passenger function demonstrates proper use of condition variables with predicates to avoid spurious wakeups. The predicate `[] { return boarded_passengers < N; }` ensures that passengers only board when there's space available.

Deadlock Prevention

The solution prevents deadlock by:

1. Using condition variables with predicates to ensure correct wakeup conditions

2. Properly signaling between threads using `notify_one()` and `notify_all()`

3. Ensuring the bus thread is always ready to depart when signaled

Testing and Validation

The program was tested with N=5 and TOTAL_PASSENGERS=20, resulting in 4 bus cycles. The output confirms that:

- Passengers wait at the gate until they can board

- The bus departs when full

- The gate reopens after the bus departs

- All passengers are eventually transported

```
manish@MacBook-Pro SysLab % g++ Q1.cpp -o q1 -std=c++11 -pthread
manish@MacBook-Pro SysLab % ./q1
Passenger 1 is waiting at the gate.
Passenger 1 enters the bus.
Passenger 2 is waiting at the gate.
Passenger 2 enters the bus.
Passenger 3 is waiting at the gate.
Passenger 3 enters the bus.
Passenger 4 is waiting at the gate.
Passenger 4 enters the bus.
Passenger 5 is waiting at the gate.
Passenger 5 enters the bus.
Bus is departing with 5 passengers.
Bus departed. Gate reopens for new passengers.
Passenger 6 is waiting at the gate.
Passenger 6 enters the bus.
Passenger 8 is waiting at the gate.
Passenger 8 enters the bus.
Passenger 7 is waiting at the gate.
Passenger 7 enters the bus.
Passenger 9 is waiting at the gate.
Passenger 9 enters the bus.
Passenger 10 is waiting at the gate.
Passenger 10 enters the bus.
Bus is departing with 5 passengers.
Bus departed. Gate reopens for new passengers.
Passenger 11 is waiting at the gate.
Passenger 11 enters the bus.
Passenger 13 is waiting at the gate.
Passenger 13 enters the bus.
Passenger 12 is waiting at the gate.
Passenger 12 enters the bus.
Passenger 14 is waiting at the gate.
Passenger 14 enters the bus.
Passenger 15 is waiting at the gate.
Passenger 15 enters the bus.
Bus is departing with 5 passengers.
Bus departed. Gate reopens for new passengers.
Passenger 16 is waiting at the gate.
Passenger 16 enters the bus.
Passenger 18 is waiting at the gate.
Passenger 18 enters the bus.
Passenger 17 is waiting at the gate.
Passenger 17 enters the bus.
Passenger 19 is waiting at the gate.
Passenger 19 enters the bus.
Passenger 20 is waiting at the gate.
Passenger 20 enters the bus.
Bus is departing with 5 passengers.
manish@MacBook-Pro SysLab %
```

Problem 2: One-Lane Bridge Problem

The one-lane bridge problem illustrates resource allocation with directional constraints. It's similar to the readers-writers problem but with the added complexity of direction management and fairness considerations.

The solution implements two distinct scheduling policies:

1. Strict Directional Priority (Case 1):

- Once vehicles start crossing in one direction, they continue until no more vehicles are waiting in that direction

- Direction only changes when the bridge is empty

- Simple but can lead to starvation if traffic is heavy in one direction

2. Fair Direction Switching (Case 2):

- Limits consecutive crossings in one direction to k vehicles

- Forces direction change after k vehicles have crossed

- Prevents starvation but may be less efficient if traffic is unbalanced

The `enter_bridge` function demonstrates complex condition checking based on the selected mode. The fair mode implementation is particularly interesting as it handles both the direction constraint and the consecutive crossing limit.

Starvation Prevention

In fair mode, starvation is prevented by:

1. Limiting consecutive crossings to k vehicles

2. Forcing direction change when the limit is reached

3. Notifying waiting vehicles in the new direction

Race Condition Handling

The solution handles potential race conditions by:

1. Using a mutex to protect all shared state

2. Checking conditions in while loops to handle spurious wakeups

3. Carefully managing the notification order to ensure fairness

Testing and Validation

The program was tested with k=5 and 10 vehicles (5 in each direction). The output confirms that:

- In strict mode, vehicles in one direction cross until no more are waiting

- In fair mode, direction switches after k consecutive crossings

- All vehicles eventually cross the bridge

- No deadlocks or starvation occur

```
manish@MacBook-Pro SysLab % g++ Q2.cpp -o q2 -std=c++11 -pthread
manish@MacBook-Pro SysLab % ./q2
Select mode:
1. Strict Directional Priority (Case 1)
2. Fair Direction Switching (Case 2)
Enter choice: 1
Vehicle crossing bridge (Dir: NORTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: NORTH)
Vehicle crossing bridge (Dir: NORTH)
Vehicle crossing bridge (Dir: NORTH)
Vehicle crossing bridge (Dir: NORTH)
Vehicle crossing bridge (Dir: NORTH)
manish@MacBook-Pro SysLab % ./q2
Select mode:
1. Strict Directional Priority (Case 1)
2. Fair Direction Switching (Case 2)
Enter choice: 2
Vehicle crossing bridge (Dir: NORTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: NORTH)
Vehicle crossing bridge (Dir: NORTH)
Vehicle crossing bridge (Dir: NORTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: NORTH)
Vehicle crossing bridge (Dir: NORTH)
Vehicle crossing bridge (Dir: SOUTH)
Vehicle crossing bridge (Dir: SOUTH)
manish@MacBook-Pro SysLab %
```

Problem 3: E-commerce Request Scheduling System

The e-commerce scheduling system demonstrates advanced resource allocation in a multi-service environment. It combines concepts from priority scheduling, resource allocation, and deadlock detection in a complex real-world scenario.

The solution employs a well-structured object-oriented design with four main classes:

1. WorkerThread: Encapsulates thread properties (priority, resources)

2. Request: Represents transaction requests with resource requirements

3. Service: Groups worker threads that handle specific transaction types

4. SchedulingSystem: Orchestrates the entire scheduling process

Scheduling Algorithm

The core scheduling algorithm in `scheduleRequests()` follows these steps:

1. Update thread availability based on completed requests

2. Check for high traffic conditions

3. Sort requests by arrival time

4. For each unprocessed request:

   - Find a suitable service based on transaction type

   - Find a suitable thread based on priority and resources

   - Process the request if a thread is available

   - Handle high traffic scenarios with retry mechanisms

5. Check for deadlock conditions

6. Increment time and repeat until all requests are processed

The thread assignment algorithm prioritizes:

1. Threads with higher priority values

2. When priorities are equal, threads with the smallest resource difference (best fit)

Deadlock Detection and Recovery

The system implements a simple but effective deadlock detection mechanism

Deadlock is detected when:

1. No requests are processed in a time step

2. The current time exceeds a threshold (requests.size() * 10)

Recovery is achieved by forcefully terminating unprocessed requests, which is a simple but effective approach for this simulation.

The system calculates comprehensive performance metrics:

1. Waiting Time: Time between arrival and start of processing

2. Turnaround Time: Time between arrival and completion

3. Resource Utilization: Percentage of available resources used over time

4. Request Statistics: Counts of processed, rejected, blocked, and waiting requests

Testing and Validation

The program was tested with various configurations, including:

- 2 services ("pay" and "order") with different thread configurations

- 4 requests with varying resource requirements

- Different arrival patterns and resource demands

```
manish@MacBook-Pro SysLab % g++ Q3.cpp -o q3 -std=c++11 -pthread
manish@MacBook-Pro SysLab % ./q3
Enter the number of services: 2

Enter the service type for service 0: pay

Enter the number of worker threads for service 0: 2

Enter priority level and resources for thread 0 of service 0: 1 5

Enter priority level and resources for thread 1 of service 0: 2 9

Enter the service type for service 1: cart

Enter the number of worker threads for service 1: 3

Enter priority level and resources for thread 0 of service 1: 1 11

Enter priority level and resources for thread 1 of service 1: 2 7

Enter priority level and resources for thread 2 of service 1: 3 19


Enter the number of requests: 5

Enter transaction type and resources required for request 0: pay 4

Enter transaction type and resources required for request 1: pay 7

Enter transaction type and resources required for request 2: pay 11

Enter transaction type and resources required for request 3: cart 13

Enter transaction type and resources required for request 4: cart 24
```

```
Time: 0, Processed: 0, Total: 5

Request 3 will complete at time 3 (Processing time: 2)

Request 3 assigned to thread 2 of service 1 (Priority: 3)
Time: 1, Processed: 0, Total: 5

Time: 2, Processed: 0, Total: 5

Request 3 completed at time 3

Request 4 blocked (High Traffic) at time 3

Request 0 will complete at time 4 (Processing time: 1)

Request 0 assigned to thread 1 of service 0 (Priority: 2)
Request 1 blocked (High Traffic) at time 3

Time: 3, Processed: 1, Total: 5

Request 0 completed at time 4

Request 4 blocked (High Traffic) at time 4

Request 1 will complete at time 6 (Processing time: 2)

Request 1 assigned to thread 1 of service 0 (Priority: 2)
Request 2 blocked (High Traffic) at time 4

Time: 4, Processed: 2, Total: 5

Request 4 blocked (High Traffic) at time 5

Request 2 rejected (High Traffic) at time 5

Time: 5, Processed: 3, Total: 5

Request 1 completed at time 6

Time: 6, Processed: 4, Total: 5

Time: 7, Processed: 4, Total: 5

Time: 8, Processed: 4, Total: 5

Time: 9, Processed: 4, Total: 5

Time: 10, Processed: 4, Total: 5

Time: 11, Processed: 4, Total: 5

Time: 12, Processed: 4, Total: 5

Time: 13, Processed: 4, Total: 5

Time: 14, Processed: 4, Total: 5

Time: 15, Processed: 4, Total: 5

Time: 16, Processed: 4, Total: 5

Time: 17, Processed: 4, Total: 5

Time: 18, Processed: 4, Total: 5

Time: 19, Processed: 4, Total: 5

Time: 20, Processed: 4, Total: 5

Time: 21, Processed: 4, Total: 5

Time: 22, Processed: 4, Total: 5

Time: 23, Processed: 4, Total: 5

Time: 24, Processed: 4, Total: 5
```

```
Time: 24, Processed: 4, Total: 5
Time: 25, Processed: 4, Total: 5
Time: 26, Processed: 4, Total: 5
Time: 27, Processed: 4, Total: 5
Time: 28, Processed: 4, Total: 5
Time: 29, Processed: 4, Total: 5
Time: 30, Processed: 4, Total: 5
Time: 31, Processed: 4, Total: 5
Time: 32, Processed: 4, Total: 5
Time: 33, Processed: 4, Total: 5
Time: 34, Processed: 4, Total: 5
Time: 35, Processed: 4, Total: 5
Time: 36, Processed: 4, Total: 5
Time: 37, Processed: 4, Total: 5
Time: 38, Processed: 4, Total: 5
Time: 39, Processed: 4, Total: 5
Time: 40, Processed: 4, Total: 5
Time: 41, Processed: 4, Total: 5
Time: 42, Processed: 4, Total: 5
Time: 43, Processed: 4, Total: 5
Time: 44, Processed: 4, Total: 5
Time: 45, Processed: 4, Total: 5
Time: 46, Processed: 4, Total: 5
Time: 47, Processed: 4, Total: 5
Time: 48, Processed: 4, Total: 5
Time: 49, Processed: 4, Total: 5
Time: 50, Processed: 4, Total: 5

Deadlock detected! Forcing completion of remaining requests...
Request 4 forcefully terminated (Deadlock)

==== PROCESSING STATISTICS ====
Order of processed requests:
Request 3 (Type: cart, Resources: 13)
  Waiting Time: 0, Turnaround Time: 2
Request 0 (Type: pay, Resources: 4)
  Waiting Time: 0, Turnaround Time: 1
Request 1 (Type: pay, Resources: 7)
  Waiting Time: 1, Turnaround Time: 3
```

```
==== PROCESSING STATISTICS ====

Order of processed requests:

Request 3 (Type: cart, Resources: 13)

  Waiting Time: 0, Turnaround Time: 2

Request 0 (Type: pay, Resources: 4)

  Waiting Time: 0, Turnaround Time: 1

Request 1 (Type: pay, Resources: 7)

  Waiting Time: 1, Turnaround Time: 3


Successfully processed requests: 3

Average waiting time: 0.333333 time units

Average turnaround time: 2 time units

Number of requests rejected: 2

Number of requests blocked: 6

Number of requests waiting: 6


==== RESOURCE UTILIZATION ====

Service 0 (pay):
  Total Resources: 14
  Average Resource Utilization: 2.52101%
  Successfully Processed Requests: 2
  Resource Time Used: 18
  Total Available Time: 714

Service 1 (cart):
  Total Resources: 37
  Average Resource Utilization: 1.37785%
  Successfully Processed Requests: 1
  Resource Time Used: 26
  Total Available Time: 1887
```