Multithreaded Data Processing Application Documentation

Overview

This C++ application demonstrates a multithreaded approach for efficiently processing data concurrently. The program simulates data input from multiple sources, processes the data in parallel using multiple threads, and employs synchronisation mechanisms to ensure data integrity. The design includes proper error handling and considerations for performance.

Design Choices

Data Structure (`Data`):

- A simple structure named `Data` is defined to represent the data being processed. For simplicity, it contains an integer field (`sensorData`), which can be replaced with actual data fields.

Synchronisation Mechanisms:

- The application uses a combination of a mutex (`dataMutex`) and a condition variable (`dataCV`) for synchronisation.
- The mutex ensures exclusive access to shared data, preventing race conditions.
- The condition variable is employed to notify threads when data is available for processing, eliminating the need for busy-waiting.

Simulating Data Input:

- The `simulateDataInput` function simulates data input by generating instances of `Data` and adding them to a shared data queue (`dataQueue`).
- The function introduces a delay to simulate the data arrival rate.

Processing Data Threads:

- Multiple threads are created using `std::thread`, each executing the `processData` function.
- The `processData` function processes data from the shared queue. Threads wait for data to be available using a condition variable.

Data Input Completion Flag:

- A boolean flag (`inputDataComplete`) is used to signal the end of data input. Threads check this flag to determine if there is more data to process.

Synchronisation Mechanisms

- Mutex (`dataMutex`):
  - Acquired before modifying the shared data queue or processing data.
  - Released after the critical section to allow other threads to access shared data.
- Condition Variable (`dataCV`):
  - Used in conjunction with the mutex for signalling and waiting.
  - Threads waiting for data use `dataCV.wait()` to efficiently wait without consuming CPU resources.
  - Threads producing data use `dataCV.notify_one()` or `dataCV.notify_all()` to wake up waiting threads when new data is available.

Error Handling

- Error Handling Mechanisms:
  - The code focuses on the core multithreading concepts, and error handling is kept minimal for clarity.
  - In a production environment, robust error handling should be implemented, especially for thread creation, synchronisation operations, and data processing logic.

Performance Considerations

- Condition Variable Usage:
  - The application uses a condition variable to efficiently notify threads when data is available, avoiding busy-waiting.
- Thread Sleep Intervals:
  - Intervals for thread sleeps are introduced to simulate data input rates and prevent excessive CPU usage during idle periods.

Building and Running

- Compilation:
  - Compile the code using the following command:
  - bash

- Copy code

- Execution:
  - Run the compiled executable:
  - bash
  - Copy code

Output Explanation

- Data Input Messages:
  - Messages indicating when data is added to the queue are printed during the `simulateDataInput` function.
- Processing Messages:
  - Messages indicating when threads are processing data are printed during the `processData` function.
- Data Input Completion Message:
  - A message indicating data input is complete is printed after the `simulateDataInput` function.

Conclusion

This multithreaded data processing application demonstrates a structured approach to concurrent programming in C++. The design choices, synchronisation mechanisms, and performance considerations aim to achieve efficient and error-resilient multithreading. Adjustments and enhancements can be made based on specific application requirements.