Computing Lab (CS69201)

Project: SyncText - A CRDT-Based Collaborative Text Editor

Maximum Marks: 100 **Deadline: 10th November, 2025 (11:59 PM)**

You've likely used Google Docs—a tool where multiple people can edit the same document simultaneously, and changes appear in real-time for everyone. Even with multiple users online at once, the system maintains consistency without conflicts. But how does it actually work?

This project asks you to build and simulate a real-time collaborative editing system similar to Google Docs. While Google Docs historically relied on Operational Transform (OT), modern collaborative editors increasingly use **CRDT (Conflict-Free Replicated Data Types)** for more efficient distributed editing. In this project, you'll adopt the CRDT-based approach using **lock-free programming** to enable real-time, conflict-free collaboration.

CRDT (Conflict-Free Replicated Data Types)

CRDTs enable conflict-free merging of concurrent operations through mathematical properties:

Properties:

- **Commutativity:** Operations can be applied in any order
- **Associativity:** Grouping of operations doesn't matter
- **Idempotency:** Applying the same operation multiple times has the same effect as applying it once

Last-Writer-Wins (LWW) Strategy

When conflicts occur, the operation with the latest timestamp wins:

V_final = V_user1 if timestamp_user1 > timestamp_user2, otherwise V_user2

Conflict Detection: Two operations conflict if they:

- Affect the same line number, AND
- Have overlapping column ranges

Conflict Resolution Priority:

- **Primary:** Operation with the latest timestamp wins.
- **Tiebreaker:** If timestamps are identical, operation from the user with the smaller user_id takes precedence.

Core Features expected in this project:

1. **Multiple Concurrent Users:** Support 3-5 users editing simultaneously
2. **Local Document Copies:** Each user maintains their own local copy of the document
3. **Automatic Change Detection:** System detects what changed when users edit their files
4. **Real-Time Synchronization:** Changes propagate to all users and merge automatically
5. **Conflict-Free Merging:** Use CRDT principles to resolve conflicts without locks
6. **Inter-Process Communication:** Use shared memory with message queues for communication between users
7. **Message Queues:** Each user should have their own message queue to receive operations from others

NOTE: This is a **lock-free programming** project. The entire system must operate without any locks. Using locks defeats the purpose of the CRDT-based approach and will result in project failure. The correctness must rely entirely on CRDT properties and atomic operations.

Project Implementation (Three Parts)

Part 1: User Creation & Local Editing with Automatic Change Detection (30%)

Objective: Create a system where users can edit their local document, and the system automatically detects, tracks, and displays changes in real-time.

What Must Work:

1. Program Execution:

```
./editor <user_id>
```

- Each user runs the program in a separate terminal with a unique user_id (e.g., user_1, user_2, user_3 etc)

2. User Registration & Discovery:

- When a user starts the program, they register themselves in a shared memory registry
- The registry maintains information about all active users (user_id and their message queue name)
- Users can query the registry to discover other active users
- System should support up to 5 concurrent users

3. Local Document:

- Each user maintains their own local copy: `<user_id>_doc.txt`
- All users start with the same initial document
- Example initial document:

Line 0: Hello World
Line 1: This is a collaborative editor
Line 2: Welcome to SyncText
Line 3: Edit this document and see real-time updates

4. Editing Workflow:

- Users open their local file `<user_id>_doc.txt` in any text editor (vim, nano, gedit, notepad, etc.)
- Users make changes naturally—adding, deleting, or modifying text
- Users save the file in their text editor
- The changes should be automatically detected and displayed

5. Automatic Change Detection - Implementation Approach:

Your program must continuously monitor the local file for modifications. Here's how you can implement this:

File Monitoring:

- Keep track of the file's last modification time (use system calls like `stat()` to get file metadata)
- Periodically check (e.g., every 2 seconds) if the modification time has changed
- When modification time changes, it means the user has saved new changes

Change Identification:

- Store the previous version of the file content in memory
- When a modification is detected, read the new file content
- Compare the old content with the new content line by line
- For each modified line, identify:
    - Line number that changed
    - Starting and ending column positions of the change

  - What content was removed (old_content)
  - What content was added (new_content)
  - Current timestamp
  - User_id who made the change

Creating Update Objects:

- For each detected change, create an update object containing:
  - Type of operation (insert, delete, replace)
  - Line number and column range
  - Old content and new content
  - Timestamp
  - User ID

6. Terminal Display:

- Your program should display the current state of the document in the terminal
- After detecting changes, update the terminal display to show the new content
- You can clear the terminal and redisplay the entire document, or update specific lines
- The display should refresh automatically whenever changes are detected

Example Flow:

```
Terminal running: ./editor user_1

[Terminal Display]
Document: user_1_doc.txt
Last updated: 14:05:30
----------------------------------------
Line 0: Hello World
Line 1: This is a collaborative editor
Line 2: Welcome to ConfluxEdit
Line 3: Edit this document and see real-time updates
----------------------------------------
Active users: user_1, user_2
Monitoring for changes...

[User edits the file in another window and saves]

[Terminal Display - Auto Updates]
Document: user_1_doc.txt
Last updated: 14:05:45
----------------------------------------
Line 0: Hello World
Line 1: This is an amazing collaborative editor  [MODIFIED]
Line 2: Welcome to ConfluxEdit
Line 3: Edit this document and see real-time updates
----------------------------------------
Active users: user_1, user_2
Change detected: Line 1, columns 10-20, "a" → "an amazing"
Monitoring for changes...
```

Deliverable for Part 1:

- Users can start the program with their user_id in separate terminals
- Users register in shared memory and can discover other active users
- Users can edit their local document using any text editor
- System continuously monitors the file for modifications
- System correctly detects and identifies changes (line, column, old/new content)
- Changes are displayed in real-time in the terminal
- Update objects are created for each detected change

Part 2: Broadcasting Local Updates via Message Passing (20%)

Objective: Enable users to broadcast their changes to all other users using message passing, and receive updates from others.

What Must Work:

1. Message Queue Setup:

- Use **message queues implementation** for inter-process communication
- Each user creates their own message queue with a unique name (e.g., `/queue_user_1`)

2. Broadcasting Local Updates:

When a user modifies their local file, the system must broadcast the changes:

Step-by-Step Process:

1. **User Makes Changes:**
    - User edits their local file and saves it
    - System detects the changes (from Part 1)
    - System generates update objects for all detected changes
2. **Accumulate Operations:**

    - Store detected changes locally in a buffer
    - After accumulating **N=5 operations**, prepare to broadcast
3. **Broadcast Update Objects:**
    - For each update object, send it to every other user's message queue
    - Each update object contains:
        - Type of operation performed (insert, delete, replace)
        - Line number and column range
        - Old content and new content
        - Timestamp when the change was made
        - User ID of the person making the change
4. **Real-Time Delivery:**
    - All other users receive the update objects in real-time through their message queues
    - This ensures every participant is immediately informed of the changes

3. Multi-Threading Architecture:

Each user program must run multiple threads:

Main Thread:

- Monitors the local file for changes (from Part 1)
- Detects and identifies changes
- Accumulates operations in a buffer
- After every 5 operations, broadcasts them to all other users
- Sends update objects to other users' message queues

Listener Thread:

- Runs continuously in parallel with the main thread
- Monitors the user's own message queue for incoming updates
- Receives update objects from other users
- Adds received updates to a local buffer for processing (merging happens in Part 3)

4. Message Passing Flow:

User_1 Terminal:
- User_1 edits line 0: "Hello" → "Hi"
- System detects change
- After 5 operations accumulated
- Broadcast to user_2, user_3 message queues
- Listener thread continues receiving updates from others

User_2 Terminal:
- Listener thread receives update from user_1
- Stores in local buffer
- Meanwhile, user_2 edits line 1
- System detects and broadcasts to user_1, user_3

User_3 Terminal:
- Listener thread receives updates from user_1 and user_2
- Stores in local buffer
- User_3 makes their own edits and broadcasts

5. Handling Dynamic Users:

- New user should starts receiving broadcasts from existing users
- All users can communicate with the new user

Deliverable for Part 2:

- Each user has their own message queue created
- Users successfully broadcast update objects to all other active users after every 5 operations
- Main thread and listener thread work concurrently
- Listener thread continuously receives updates from other users
- Received updates are stored in a local buffer
- Multi-threading works correctly without locks

Part 3: Listening, Merging, and Synchronization using CRDT (50%)

Objective: Process received updates, merge them with local changes using CRDT principles, and maintain consistency across all users.

What Must Work:

1. Listening for Incoming Updates:

The listener thread (from Part 2) continuously monitors for incoming updates:

Listener Thread Responsibilities:

- Continuously check the message queue for new update objects
- When a new update arrives:
    - Read the complete update object

    - Extract information: operation type, line, column, old/new content, timestamp, user_id
    - Add the update to a local buffer for processing
- Run concurrently with the main thread without blocking

2. Merging Updates Using CRDT:

After receiving updates or after every N=5 operations (whichever comes first), merge the buffered updates:

Merging Process:

1. **Collect Updates:**
    - Gather all updates from the local buffer (received from other users)
    - Gather all local updates that haven't been merged yet
2. **Detect Conflicts:**
    - Check if any two updates (local or received) affect the same location
    - Two updates conflict if they:
        - Modify the same line number, AND
        - Have overlapping column ranges
3. **Resolve Conflicts using LWW (Last-Writer-Wins):**
    - Compare timestamps of conflicting updates
    - The update with the **latest timestamp** takes precedence
    - If timestamps are identical, the update from the user with **smaller user_id** wins
    - Discard the losing update
4. **Apply Non-Conflicting Updates:**
    - Apply all non-conflicting updates to the local file
    - Updates can be applied in any order (commutativity property of CRDT)
5. **Apply Winning Updates from Conflicts:**
    - Apply the winning updates from conflict resolution
    - This ensures deterministic conflict resolution

3. Updating the Local Document:

After merging, update the local file and display:

Update Process:

- Modify the local file `<user_id>_doc.txt` with the merged content
- Update the terminal display to show the new document state

- All users' documents should eventually show the same content
- The update should happen automatically without user intervention

Example Merge Scenario:

User_1's buffer:
- Local: Line 0, col 0-4, "Hello" → "Hi", timestamp: 14:05:10, user_1
- Received: Line 0, col 0-4, "Hello" → "Hey", timestamp: 14:05:12, user_2

Conflict detected: Both modify Line 0, columns 0-4
Resolution: user_2's update wins (14:05:12 > 14:05:10)
Final: Line 0 becomes "Hey"

User_1's document updates to match user_2's change

4. Synchronization Timing:

- Merge and synchronize after every **N=5 operations** (local or received, whichever reaches 5 first)
- This ensures periodic synchronization while maintaining reasonable performance
- All users perform merging independently using the same CRDT rules
- This guarantees all users converge to identical states

Deliverable for Part 3:

- Listener thread successfully receives and buffers updates from other users
- System correctly detects conflicts between updates
- Conflicts are resolved using LWW based on timestamps
- Non-conflicting updates are applied correctly
- Local document file is updated with merged content
- Terminal display shows the updated document state
- All users' documents converge to the same final state
- Entire system operates lock-free using CRDT principles

Submission Guidelines

Submit a single ZIP archive named `<roll_no>_project_2.zip`.

Required Contents:

1. Source Code

- All source files (.cpp, .h, .c, or other language files)
- Well-organized directory structure
- Clear code comments explaining key logic

2. README File

Must include:

Compilation Instructions: Step-by-step commands to compile your code
Execution Instructions: How to run the program for multiple users
Terminal 1: ./editor user_1Terminal 2: ./editor user_2Terminal 3: ./editor user_3

- **Dependencies:** Required libraries (e.g., pthread, rt for message queues)
- **Platform:** OS and compiler information (e.g., Ubuntu 20.04, g++ 9.4.0)
- **How to Test:** Brief description of how to test the system

3. DESIGNDOC File

Must contain:

a) System Architecture:

- High-level design overview
- Major components and how they interact
- Key data structures used

b) Implementation Details:

- How you implemented change detection (file monitoring approach)
- How you structured message queues and shared memory
- How you implemented the CRDT merge algorithm
- Thread architecture and communication

c) Design Decisions:

- Important implementation choices and rationale
- How you ensured lock-free operation
- Trade-offs you made

d) Challenges and Solutions:

- Major difficulties faced during implementation
- How you debugged and resolved issues

Complete Example Walkthrough

Setup:

Initial Document (provided):

```
Line 0: int x = 10;
Line 1: int y = 20;
Line 2: int z = 30;
```

All users start with this same document.

Execution:

Terminal 1:

```
$ ./editor user_1
Registered as user_1
Message queue created: /queue_user_1
Active users: user_1

Document: user_1_doc.txt
----------------------------------------
Line 0: int x = 10;
Line 1: int y = 20;
Line 2: int z = 30;
----------------------------------------
Monitoring for changes...
```

Terminal 2:

```
$ ./editor user_2
Registered as user_2
Message queue created: /queue_user_2
Active users: user_1, user_2

Document: user_2_doc.txt
----------------------------------------
Line 0: int x = 10;
Line 1: int y = 20;
Line 2: int z = 30;
----------------------------------------
Monitoring for changes...
```

Terminal 3:

```
$ ./editor user_3
Registered as user_3
Message queue created: /queue_user_3
Active users: user_1, user_2, user_3

Document: user_3_doc.txt
----------------------------------------
Line 0: int x = 10;
Line 1: int y = 20;
Line 2: int z = 30;
----------------------------------------
Monitoring for changes...
```

Scenario 1: Non-Conflicting Edits

User_1 edits their file: Opens `user_1_doc.txt` in Notepad, changes line 0, saves:

```
Line 0: int x = 50;  (changed 10 to 50)
```

Terminal 1 updates:

```
Document: user_1_doc.txt
Last updated: 14:05:10
----------------------------------------
Line 0: int x = 50;  [MODIFIED]
Line 1: int y = 20;
Line 2: int z = 30;
----------------------------------------
Change detected: Line 0, col 8-9, "10" → "50", timestamp: 14:05:10
```

User_2 edits their file: Opens `user_2_doc.txt`, changes line 1, saves:

```
Line 1: int y = 100;  (changed 20 to 100)
```

After broadcasting and merging, all terminals show:

```
Document: <user_id>_doc.txt
```

```
---------------------------------------
Line 0: int x = 50;
Line 1: int y = 100;
Line 2: int z = 30;
---------------------------------------
Received update from user_1: Line 0 modified
Received update from user_2: Line 1 modified
All updates merged successfully
```

Scenario 2: Conflicting Edits

User_1 edits line 0:

```
Line 0: int x = 75;  (timestamp: 14:06:10)
```

User_2 edits line 0 (almost simultaneously):

```
Line 0: int x = 99;  (timestamp: 14:06:12)
```

Conflict Resolution:

- Both updates modify Line 0, columns 8-9
- User_2's timestamp (14:06:12) > User_1's timestamp (14:06:10)
- User_2's update wins

All terminals converge to:

```
Document: <user_id>_doc.txt
---------------------------------------
Line 0: int x = 99;
Line 1: int y = 100;
Line 2: int z = 30;
---------------------------------------
Conflict detected and resolved using LWW
Final state synchronized across all users
```