DESIGN DOCUMENT — SyncText Collaborative Editor

File: own.cpp
Language: C++17
Compile:

```
g++ -std=c++17 -O2 own.cpp -o editor -lrt -pthread
```

---

a) System Architecture

High-Level Overview

The SyncText editor enables multiple users (up to 5) to collaboratively edit independent text files that synchronize automatically. Each user instance runs in a separate process, communicating via POSIX **message queues** and sharing global state through **shared memory**.

The synchronization model follows a **Last-Writer-Wins (LWW)** policy — ensuring eventual consistency across users even under concurrent edits. The system is designed to be **lock-free**, meaning no explicit mutexes or semaphores are used, enabling high concurrency and minimal blocking between threads or processes.

Major Components & Interaction

1. Shared Memory Registry

- Used to maintain a global list of active users.
- Each entry in shared memory (`RegistryPage`) stores:
    - A user ID (`client_name`)
    - Its message queue name (`/queue_<user_id>`)
    - PID (for liveness checks)
    - Active state flag (0 = free, 1 = claiming, 2 = active)

2. POSIX Message Queues

- Each user creates a unique queue (`/queue_<user_id>`).
- Local edits are serialized into `LineUpdate` messages and broadcast to every other user's queue.
- The listener thread receives incoming messages asynchronously.

3. Listener Thread

- Each process spawns one listener thread (`mq_listener`) which constantly receives messages from its queue.
- Received updates are stored into a **lock-free stack**, ensuring that updates can be retrieved safely by the main thread without locks.

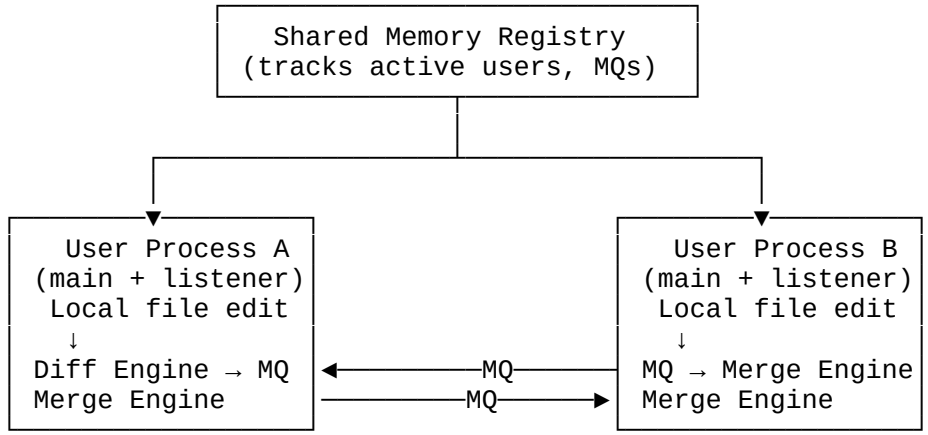
4. File Monitor + Diff Engine

- The main thread continuously monitors the user's text file using its **modification timestamp** (`stat()` system call).
- When a change is detected, it computes differences between the old and new versions line-by-line.
- Each detected change is packaged into a `LineUpdate` structure and broadcast to other users.

5. Merge Engine (LWW Merge)

- When the number of pending updates (local + remote) reaches the global threshold (`SYNC_THRESHOLD = 5`), a merge is triggered.
- All updates are merged line-wise using timestamps and user IDs (Last-Writer-Wins conflict resolution).
- The merged file is rewritten to disk, ensuring all users converge to the same state.

Component Interaction Flow



Each process both sends and receives update messages asynchronously, resulting in consistent state propagation across all participants.

Key Data Structures

Structure	Purpose
`RegistryPage`	Global shared memory containing user slots.

Structure	Purpose
`UserSlot`	Stores one user's ID, PID, MQ name, and state.
`MsgNode`	Lock-free node for incoming messages (stack-based).
`LineUpdate`	Represents a single line change: old/new diff, timestamp, user ID.
`vector<string>`	Stores in-memory representation of the text file.

---

b) Implementation Details

1. Change Detection

- The system uses periodic polling (`usleep(POLL_MS * 1000)`) and compares the file's `st_mtime` (modification time).
- Upon detecting a change, it loads the new file content and computes **line-by-line diffs** using the `compute_line_updates()` function.
- The function identifies the first and last differing indices to extract the minimal substring difference per line.

Functions involved:

- `compute_line_updates()`
- `read_file_lines()`
- `write_file_lines()`

This design avoids file system events (like `inotify`) to ensure portability and simplicity.

---

2. Message Queues and Shared Memory Structure

- **Shared Memory:** Created using `shm_open()` and mapped via `mmap()`.
  Each user registers itself by performing a compare-and-swap (CAS) on its slot's `state` flag.
- **Message Queues:** Each user owns a queue named `/queue_<user_id>`.
  The queues are opened with:

  ```
  mq_open(qname.c_str(), O_CREAT | O_RDONLY | O_NONBLOCK, 0666, &attr);
  ```

- **Attributes:**
  - `mq_maxmsg = 10`
  - `mq_msgsize = 1024`
  - Non-blocking flag (`O_NONBLOCK`) to prevent stalls.

Message broadcasting uses `mq_send()` to every active user's queue except self.
Cleanup is ensured via `mq_unlink()` when a process terminates.

---

3. CRDT Merge (LWW Merge Algorithm)

The merge algorithm ensures all users converge to a consistent final document state.

Logic:

1. Collect both local and remote updates once total count ≥ `SYNC_THRESHOLD`.
2. For each line number, retain only the most recent update.
3. If two updates have the same timestamp, the smaller `user_id` wins (tie-breaker).
4. The merged lines replace the in-memory document, and the file is rewritten.

Key function: `unordered_map<int, LineUpdate> winners`
ensures that only one "winner" per line is applied.

This simple yet robust **LWW-CRDT** guarantees eventual consistency without locks.

---

4. Thread Architecture and Communication

Threads:

- **Main Thread:**
  - Polls the file system.
  - Detects local edits.
  - Merges updates and writes file.
- **Listener Thread:**
  - Runs continuously using `mq_receive()`.
  - Pushes received messages into a **lock-free stack** (`incoming_stack`).
  - The main thread periodically drains this stack (`collect_incoming_updates()`).

Synchronization Model:

- No locks are used between threads.
- Instead, **atomic operations** and **memory-ordering guarantees** ensure thread safety:
  - `atomic<MsgNode*> incoming_stack`
  - `compare_exchange_weak()` for stack push/pop.
- Communication between processes is asynchronous via POSIX message queues.

---

c) Design Decisions

1. Lock-Free Operation

- No `pthread_mutex` or `semaphore` objects are used anywhere.
- Communication between threads is done via atomic CAS (compare-and-swap) on linked nodes.
- This ensures:
    - No deadlocks.
    - Non-blocking message handling.
    - High concurrency for multiple users.

2. Use of Shared Memory + MQ

- Shared memory was chosen over pipes or sockets for simplicity and persistence.
- POSIX MQ provides:
    - Named queues for cross-process communication.
    - Built-in kernel buffering (reducing race conditions).
    - Non-blocking I/O with small message sizes (<1 KB).

3. Line-Level Synchronization

- File synchronization is done at the **line level** instead of character level for clarity and efficiency.
- It strikes a balance between granularity and complexity.

4. Simplicity and Robustness

- Avoided event-based file watchers (e.g., `inotify`) for maximum portability (works on any POSIX system).
- Merge threshold (`SYNC_THRESHOLD = 5`) prevents excessive disk writes.

5. Design Trade-Offs

Trade-Off	Choice	Reason
Fine-grained vs line-based diff	Line-based	Easier conflict resolution
Mutex-based concurrency	Lock-free	Faster, no blocking
Event-driven file monitor	Timestamp polling	Simpler, portable
Real-time update push	Periodic batch merge	Prevents thrashing and keeps global sync predictable

---

d) Challenges and Solutions

1. Challenge: Message Queue Errors (`EINVAL` or `EAGAIN`)

Cause: Kernel limits on message size (`msgsize_max`).
Solution: Dynamically read limits from `/proc/sys/fs/mqueue/` and cap message size accordingly.

---

2. Challenge: Registry not resetting on crash

Cause: Shared memory retained stale user slots.
Solution: Added process liveness check (`kill(pid, 0)`) in `purge_dead_slots()` to auto-clean crashed entries.

---

3. Challenge: Detecting local changes efficiently

Cause: Continuous I/O polling caused multiple false detections.
Solution: Compared stored `st_mtime` and used `usleep()` polling intervals to reduce unnecessary reads.

---

4. Challenge: Race conditions between threads

Cause: The listener thread could modify shared buffers while main thread accessed them.
Solution: Replaced locks with atomic lock-free stack using `compare_exchange_weak` to safely store incoming updates.

---

5. Challenge: Conflicting edits between users

Cause: Two users editing the same line simultaneously.
Solution: Implemented deterministic **LWW resolution** based on timestamps and user IDs — ensuring all nodes converge to the same state.

---

6. Challenge: Debugging Inter-Process Communication

Cause: Multiple queues and shared memory segments made it hard to trace messages.
Solution: Added descriptive debug logs:

```
[Broadcast] Sent N updates to other users.
[Merge] Applied line X from <user>.
[Listener] Buffered N remote updates.
```

These logs helped visualize synchronization and verify correctness.

---

7. Challenge: Ensuring Safe Termination

Cause: Processes sometimes left queues unlinked after termination.
Solution: Added cleanup logic (`do_cleanup()`, `unlink_own_queue_if_exists()`) and signal handling for `SIGINT` and `SIGTERM`.

Conclusion

The **SyncText Collaborative Editor** achieves:

- Lock-free concurrency
- Deterministic conflict resolution via LWW
- Robust inter-process synchronization using shared memory and message queues
- Portable file monitoring and automatic recovery from crashes