SYSC4001

Assignment 3:

Seydi Cheikh Wade (101323727)

Sean Baldaia (101315064)

Part 2

Design discussion in context of critical section problem

In this assignment, multiple TA processes share a rubric and per-question state via a System V shared memory segment. In Part 2.b, we introduce POSIX semaphores to coordinate concurrent access to this shared state. Below, we explain how the final design relates to the three classic requirements of the critical-section problem: **mutual exclusion**, **progress**, and **bounded waiting**.

1. Mutual Exclusion:

Our solution uses **process-shared semaphores** embedded directly in the shared memory structure (SharedArea) to protect all shared data:

- mutex_rubric protects:
    - rubric[] (the marking scheme),
    - rubric_dirty (flag that tells the parent to write the rubric back to disk),
    - and also ensures consistent updates of the **terminate flag** when needed.
- mutex_questions protects:
    - question_state[] (0 = not started, 1 = marking, 2 = done),
    - exam_done (whether an exam appears fully marked).
- mutex_log protects:
    - log_counter,
    - and ensures log lines are printed atomically and in a consistent global order.
- exam_ready is used as a synchronization primitive to wake up TAs when a new exam is loaded.

Every time a TA or the parent needs to read or modify shared variables in these groups, it must first call sem_wait(...), perform a short critical section, and then sem_post(...). As a result:

- Two processes can never update the same rubric entry or question state at the same time.
- A TA cannot see half-updated data, because all modifications are done while holding the corresponding mutex semaphore.
- Log messages are serialized as well, so we can observe the real interleaving without garbled lines.

This satisfies **mutual exclusion**: at most one process occupies any protected critical section at once.

2. Progress:

The **progress requirement** says that if no process is in its critical section and one or more processes want to enter, the decision of which process enters next must not be indefinitely postponed.

In our design:

- Critical sections protected by mutex_rubric, mutex_questions, and mutex_log are **short and bounded**:
    - For rubric: we only read or write a single character and flip a dirty flag.
    - For question state: we scan the small question_state[] array, claim a question, or detect that an exam is done.
    - For logging: we just increment log_counter and print one log line.
- All **long operations** (simulated marking time and rubric-checking delays using sleep_random_ms) happen **outside** the critical sections. A TA:

1. Briefly locks the semaphore,
2. Reads or updates shared state,
3. Unlocks as soon as possible,
4. Then performs its long "work" while not holding any mutex.

This design ensures that no process "holds the lock and goes to sleep" for an extended period. Therefore:

- If several TAs are ready to mark, they will take turns acquiring the mutexes; no TA is blocked by another TA being inside a long delay while still holding a lock.
- The parent process also makes progress: it periodically checks exam_done, loads the next exam when appropriate, and posts exam_ready to wake all TAs without having to spin on shared variables.

Assuming the underlying semaphore implementation itself respects reasonable fairness (no permanent starvation inside sem_wait), **progress is preserved**: whenever some process wants to enter a critical section and no one is using it, some process will be allowed to enter in finite time.

3. Bounded Waiting:

The **bounded waiting** requirement says that there must be a bound on the number of times other processes are allowed to enter their critical section after a process has requested to enter and before that request is granted.

In our system:

- Access to each shared structure is controlled by a **single binary semaphore** (for rubric, questions, and logging respectively).
  No process can "jump ahead" an unbounded number of times without releasing the semaphore, because:
  - Each holder performs a small, fixed piece of work inside the critical section.
  - The process then releases the semaphore and only re-enters after some delay (e.g., after checking or marking a question).
- Because critical sections are small and deterministic, the effective "queue" behind each sem_wait(...) has an implicit bound: each waiting TA will eventually get the semaphore once the current holder exits and any processes already blocked before it are served.
- We also avoid typical starvation patterns:
  - There is no complex hierarchy of locks or nested waiting on different semaphores in opposite orders; each critical section uses a **single mutex**.
  - We do not keep a lock while waiting on another synchronization primitive, which reduces the risk of one process being perpetually delayed by others.

In practice, this design provides **bounded waiting (no starvation)** for the TAs and the parent, up to the fairness guarantees of the OS's semaphore implementation. Every TA that repeatedly wants to access the shared rubric or question state will eventually get a turn, and no TA can monopolize a lock due to the tiny critical sections.