SYSC 4001 A – Operating Systems | Fall 2025
Carleton University

Assignment 3 – Part 2: Concurrent Processes in Unix

Comparative Analysis of Concurrent TA Processes

Author: Ozan Kaya
Student Number: 101322055

Introduction

This report provides a comparative analysis of two implementations for the concurrent TA marking system. The first implementation (**Part 2.a**) deliberately excludes synchronization mechanisms, demonstrating the problems of race conditions and data corruption. The second implementation (**Part 2.b**) resolves these issues by utilizing **System V Semaphores** to enforce mutual exclusion over all shared resources, ensuring data integrity and correct program logic.

---

1. Analysis of Race Conditions in Part 2.a (Unsynchronized)

The output from Part 2.a shows two distinct race conditions arising from concurrent, unsynchronized access to shared resources by the TA child processes.

1.1 Race Condition on Question Marking Status

Shared Resource: The `question_marked[5]` boolean array in shared memory.

Problem: This occurred due to a classic **Read-Modify-Write** race. Multiple TAs would concurrently read the status of a question (e.g., Question 1) as `false` and independently decide to mark it. Before one TA could set the flag to `true`, another TA would have already checked the old value.

Observed Evidence (Terminal Output):

```
TA 2 started marking Question 1 for Student 1
TA 1 started marking Question 1 for Student 1
…
TA 1 FINISHED marking Question 1 for Student 1
…
TA 2 FINISHED marking Question 1 for Student 1
```

Impact: This resulted in wasted work (two TAs performing the same task) and violated the logic requiring each question to be marked only once.

1.2 Race Condition on Rubric Data and File I/O

Shared Resources: The `rubric[5][3]` character array in shared memory and the persistent `rubric.txt` file.

Problem: This was a **Lost Update** race condition within the critical section of modifying and saving the rubric. Multiple TAs would read the existing file content into shared memory, attempt a modification (e.g., ' ' → '!'), and then simultaneously call `save_rubric()`. Since file writes are not atomic, the TA who finishes the I/O operation last dictates the final state, overwriting any intermediate saves with potentially stale data.

Observed Evidence (Final `rubric.txt` State and Output):

- The terminal showed three separate `Rubric file saved` messages immediately following three separate `MODIFIED` logs.
- The final state of line 1 in the rubric was corrupted (e.g., `1,#`). This unexpected character (an ASCII value resulting from rapid, unsynchronized increments and simultaneous file overwrites) confirmed the loss of data integrity.

---

2. Synchronization and Resolution in Part 2.b

In the Part 2.b implementation, three binary semaphores (mutexes) were introduced to protect all shared resources, successfully eliminating the observed races.

2.1 Resolution of Rubric File Race (Protected by `semid_rubric`)

Mechanism: The entire operation, reading the shared memory, deciding to modify a character, executing the modification, and calling `save_rubric()`, was wrapped in a critical section protected by `semid_rubric`.

Impact on Output: The chaotic, simultaneous saves were replaced by a sequential flow.

```
TA 2 entered Rubric critical section.
TA 2 MODIFIED Rubric line 1: ' ' -> 'A'
--- Rubric file saved —
TA 2 left Rubric critical section.
TA 1 entered Rubric critical section. // Can only enter after TA 2
leaves
TA 1 MODIFIED Rubric line 1: 'A' -> 'B'
--- Rubric file saved —
```

Result (Data Integrity): The final `rubric.txt` reflects a clean, sequential update chain (e.g., `1,E`), proving that no update was lost or corrupted.

2.2 Resolution of Question Marking Race (Protected by `semid_marks`)

Mechanism: The **Read-Modify-Write** operation on the `question_marked` array was placed inside a critical section protected by `semid_marks`. This forced TAs to wait their turn to check the array, claim an unmarked question (by setting the flag to `true`), and then immediately release the lock.

Impact on Output: The output confirms clean division of labor. For Student 1, TAs marked questions uniquely:

```
TA 1 started marking Question 1 for Student 1
TA 2 started marking Question 2 for Student 1
TA 3 started marking Question 3 for Student 1
// No concurrent marking of the same question observed.
```

Result: Wasted work is eliminated, and the system operates efficiently as all TAs contribute to marking unique questions.

2.3 Resolution of Next Exam Load Signal (Protected by `semid_parent_signal`)

Mechanism: This semaphore synchronized the logic for detecting completion and signaling the parent (TA process) with the logic for checking the signal and loading the next exam (Parent process).

Impact on Output: Only one TA signaled completion (`TA 3 detected all questions marked. SIGNALING parent.`), followed by a clean load operation by the parent (`--- NEXT EXAM LOADED --- Student ID: 2`), ensuring a smooth, synchronized transition between exams.

---

3. Conclusion

The transition from Part 2.a to Part 2.b definitively demonstrates the necessity of synchronization in concurrent systems.

Feature	Part 2.a (Unsynchronized)	Part 2.b (Synchronized with Semaphores)
Question Marking	Inefficient. Multiple TAs marked the same question (wasted CPU).	Efficient. TAs marked unique questions only.
Rubric File	Data corruption and lost updates due to concurrent file writes. Final state used corrupted ASCII symbols (#).	Data integrity maintained. Sequential writes ensured the final state was a clean, expected ASCII progression (`E`, `B`, `D`, etc.).
Process Control	Chaotic; multiple TAs could try to signal next exam or overwrite shared state simultaneously.	Ordered; clean sequential entry/exit into critical sections and synchronized handoff to the parent process for exam loading.

The Part 2.b implementation successfully achieved **mutual exclusion** using System V Semaphores, guaranteeing data integrity and ensuring the system's logic (one mark per question, one successful rubric update at a time) is maintained despite the concurrent execution of multiple TA processes.