The Sleeping Teaching Assistant

A Synchronization Problem using POSIX Threads, Mutexes & Semaphores

---

Team Members

Mohammad Ikhlas - AP23110010087

Praveen Vemulapati - AP23110011455

Venkatesh Manipati - AP23110011626

Problem Statement

A university TA helps students in an office with one desk and three waiting chairs in the hallway.

- If no students are present, the TA naps.
- If a student arrives and the TA is sleeping, the student must awaken the TA.
- If a student arrives and the TA is busy, the student waits in a hallway chair.
- If all hallway chairs are full, the student leaves and will come back later.

Physical Analogy: This is a classic "Producer-Consumer" problem. Students "produce" requests for help, and the TA "consumes" them.

Goals & Constraints

Goal: Model this scenario concurrently, ensuring the TA and Students operate correctly without errors (race conditions or deadlocks).

Constraint: 1 TA (Single Consumer)
Only one "help session" can be active at a time. The TA must process requests one by one.

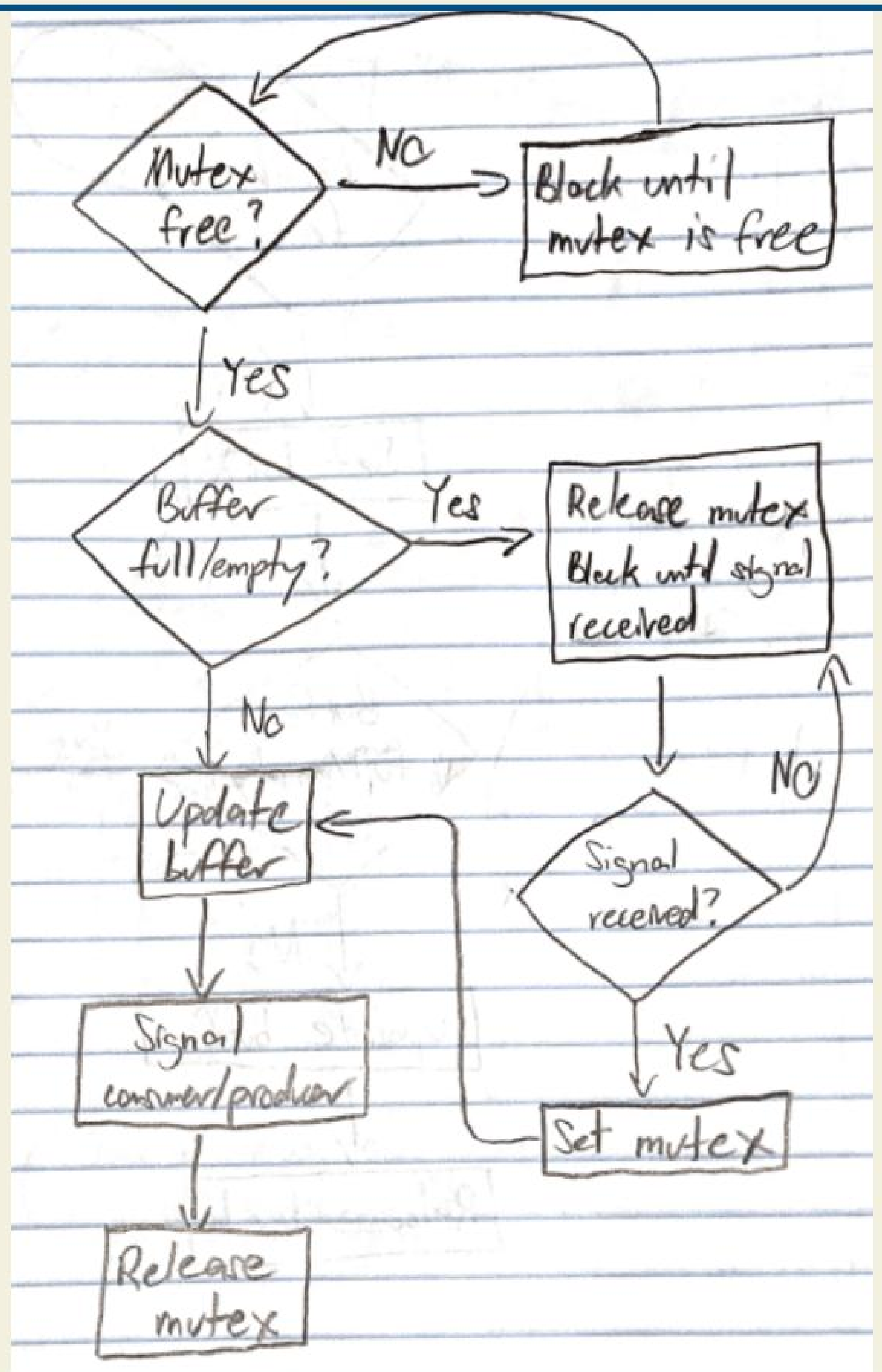
Constraint: 3 Chairs (Bounded Buffer)
The waiting area has a fixed size. We can't have more than 3 students waiting.

Constraint: Leaving Policy ("Balking")
If the buffer (hallway) is full, arriving students (producers) do not wait; they leave immediately.

Design & Flow

Logical Flow (Student)

1. Student thread is created (arrives).
2. Attempt to acquire lock on chairs.
3. **Check `waiting_count`:**
   - **If `count < 3` (Space):** Increment count, signal TA (wake up), wait for TA to be free, then get help.
   - **If `count == 3` (Full):** Release lock and leave.
4. Thread exits.



Synchronization Primitives

pthread_t (Threads)

We use POSIX threads to model concurrency. The TA is one long-running thread, and each student is a new thread that is created and joins.

Mutex (The "Talking Stick")

A `pthread_mutex_t` is used for **Mutual Exclusion**. It protects the shared `waiting_count` variable. Only the thread holding the lock can read or write this count, preventing race conditions.

Semaphores (The "Signals")

`sem_t` is used for **Signaling**. We use two:
`sem_students` (Doorbell): Counts waiting students. TA sleeps on this.
`sem_ta` (Office Door): Signals the TA is ready. Students wait on this.

Algorithm Pseudocode

Student (Loop)

```
Student_Thread() { lock(mutex); if (waiting_count < 3) { waiting_count++; unlock(mutex); // Ring doorbell to wake TA post(sem_stude
```

Implementation: The "Handshake"

Key Code Snippets

```
// --- STUDENT --- // Wake up the TA sem_post(&sem_students); // Wait for TA to be ready sem_wait(&sem_ta); // --- TA --- // Wait for
```

Correctness Reasoning

- **Race Conditions Prevented:** All access to the shared `waiting_count` variable is "mutually exclusive," protected by the `mutex`. Only one thread can read or write it at a time.
- **"Lost Wakeups" Avoided:** Semaphores are counters, not flags. If a student signals (`sem_post`) *before* the TA sleeps (`sem_wait`), the semaphore count becomes 1. When the TA later calls `sem_wait`, it sees the count is 1, decrements it, and **does not sleep**.
- **Deadlock Avoided:** Deadlock ("Circular Wait") is impossible. A student *never* holds the mutex while waiting for a semaphore. They `unlock(mutex)` *before* they `wait(sem_ta)`, breaking the chain.

Students arrive one at a time. Tests the TA's sleep/wake cycle.

- **Test 2: Burst Arrival** 5 students arrive at once. Tests the "balking" logic. The first 3 should wait, the next 2 should leave.
- **Edge Case:** Student arrives

Extensions & Improvements

Multiple TAs

The project could be extended to support multiple TA threads. This would change the logic significantly:

- We would need a semaphore to count *available TAs*.
- Students would `wait()` on this "available TA" semaphore.
- This turns the problem into a more complex "Multi-Producer, Multi-Consumer" model.

Student Priorities

We could implement a system where some students get priority (e.g., final-year vs. first-year).

- This would require replacing the simple `waiting_count` with a **Priority Queue** data structure.
- The *same mutex* would be used to protect the queue, but the TA would "dequeue" the highest-priority student.

Conclusion & Lessons Learned

The Right Tool for the Job

This project highlights the difference between synchronization tools.

Mutex: Use for **Protection** (protecting shared data like a variable).

Semaphore: Use for **Coordination** (signaling, sleeping, and waking threads).

Concurrency is Manageable

Handling concurrency pitfalls like race conditions and deadlocks is complex, but possible with a careful design. By strictly enforcing lock order and using semaphores correctly, we can build a robust, efficient, and correct system.

References

- Silberschatz, Galvin, & Gagne. (2018). *Operating System Concepts (10th Ed.)*.
- Downey, Allen B. (2016). *The Little Book of Semaphores*.
- POSIX Threads Programming Tutorial (GeeksforGeeks, IBM Developer).

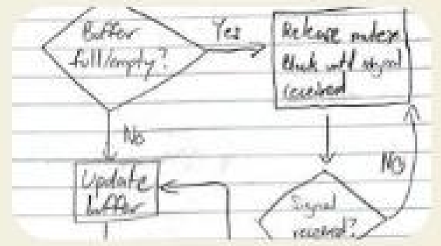
Questions?

Thank you.

Image Sources

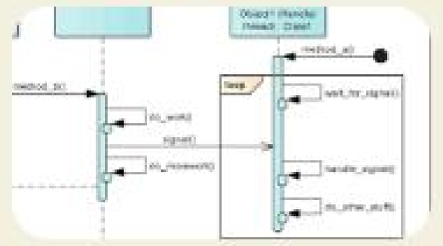
https://media.istockphoto.com/id/1158203930/vector/burnout-concept-illustration-with-exhausted-man-office-worker-sitting-at-the-table.jpg?s=612x612&w=0&k=20&c=jleYMmABeH5fjh7wc--ySZdPmyHmI2Lyuy3GmcNdzhE=

Source: www.istockphoto.com



https://miro.medium.com/v2/resize:fit:846/1*JD93hAvRCrR7uxyDVutRKw.png

Source: levelup.gitconnected.com



https://i.sstatic.net/e92Ju.png

Source: stackoverflow.com