Relazione Tecnica: Frogger Clone (Versione Multithread)

1. Obiettivi e Architettura

Il progetto consiste nella riproduzione del videogioco Frogger in linguaggio **C** su ambiente Linux. A differenza delle implementazioni sequenziali classiche, questa versione utilizza un'architettura **Fortemente Concorrente** basata su **Thread** (libreria `pthread`). L'intero stato del gioco risiede in memoria condivisa, richiedendo una rigorosa gestione della sincronizzazione per evitare *Race Conditions*.

2. Modello di Concorrenza (Shared Memory)

Il sistema è strutturato come un'applicazione multi-thread in cui diverse entità operano in parallelo nello stesso spazio di indirizzamento:

- **Main Thread (Rendering & Logic):** Esegue la funzione `gameArea`. Funge da "osservatore": legge ciclicamente lo stato degli oggetti condivisi, gestisce le collisioni, l'input utente e aggiorna la grafica a video.
- **Worker Threads (Entità di Gioco):**
    - **Veicoli:** Ogni veicolo è gestito da un thread indipendente (`funcVehicle`) che ne aggiorna la posizione.
    - **Tronchi:** Thread dedicati (`funcLog`) gestiscono il flusso del fiume.
    - **Proiettili:** Gestione dinamica dei thread (`funcBullet`). Quando si spara, viene creato un thread "on-demand" che termina autonomamente una volta uscito dallo schermo o colpito un bersaglio.

3. Sincronizzazione e Mutex

Poiché tutti i thread scrivono e leggono le stesse variabili globali (`vehicles`, `logs_data`, `bullets_data`), l'accesso è regolato tramite **Mutex** (`pthread_mutex_t`) per garantire la coerenza dei dati.

Strategie di Locking adottate:

1. **Locking Granulare (Fine-grained Locking):** Invece di un unico "lock gigante" che bloccherebbe tutto il gioco, è stato allocato un array di mutex, uno per ogni entità (es. `vec_mutex[i]`).
    - *Vantaggio:* Il thread del veicolo A non blocca il thread del veicolo B. Il Main Thread acquisisce il lock solo sul veicolo specifico che sta disegnando in quel momento, massimizzando il parallelismo.

2. **Protezione della Grafica (Ncurses):** La libreria *ncurses* non è thread-safe. È stato implementato un mutex dedicato (`mutexcurses`) che avvolge tutte le chiamate di disegno (`wprintw`, `wrefresh`). Questo impedisce che un thread provi a scrivere a video mentre il main thread sta aggiornando la schermata, prevenendo glitch grafici.

4. Gestione Dinamica della Memoria

Il codice fa ampio uso di allocazione dinamica (`malloc`/`free`) per gestire le strutture dati condivise e i relativi mutex, permettendo al gioco di scalare in base alla difficoltà o alla dimensione della finestra senza ricompilare (es. `vec_mutex = malloc(...)`).

5. Multimedia e Librerie Esterne

- **Interfaccia (TUI):** Utilizzo avanzato di **ncurses** per la gestione di finestre multiple, colori personalizzati e input non bloccante.
- **Audio:** Integrazione della libreria **SDL2_mixer** per gestire effetti sonori e musica di sottofondo in parallelo alla logica di gioco, arricchendo l'esperienza utente senza bloccare il flusso di esecuzione.

6. Gestione delle Collisioni

Il rilevamento delle collisioni (`collisions.c`) avviene nel thread principale. Grazie alla memoria condivisa, il sistema ha accesso immediato alle coordinate aggiornate di tutti gli oggetti (protette da mutex), permettendo interazioni complesse come:

- Collisioni Rana-Veicolo.
- Collisioni Proiettile-Nemico (con gestione distruzione thread proiettile tramite `pthread_cancel`).
- Logica di trascinamento Rana-Tronco.