CSN6214
Operating Systems
Assignment

Concurrent Networked Board Game Interpreter

Course: Operating Systems

Due Date: **February 8th, before 5:00 PM**

- **Team Size:** 3 to 4 members, must be from the same tutorial/lab section

1 Project Overview and Goal

You will design and implement a **multiplayer, text-based board game** for **3 to 5 players** using a **hybrid concurrency model** that **combines multiprocessing and multithreading**. The system must also support the playing of **multiple, successive games** without requiring a full server restart. This assignment demonstrates advanced understanding of:

- **Multiprocessing** via `fork()` for client isolation
- **Multithreading** for internal server tasks (e.g., logging, scheduling)
- **Inter-Process Communication (IPC)**
- **Synchronization** across threads **and** processes (using mutexes, semaphores)
- **Round Robin (RR) scheduling** for turn management
- **Concurrent, safe logging** of all game events

You may **choose your own turn-based, text-based game**, provided it meets the constraints in Section 2. The system must support **one** of the following deployment modes:

- **Single-machine mode**: All components on one host; client-server communication via **IPC** (e.g., named pipes, message queues).
- **Multi-machine mode**: Clients and server on different machines; communication via **TCP sockets (IPv4)**.

Hybrid concurrency is mandatory: Your server must use **both `fork()` (for clients) AND POSIX threads (`pthreads`) for internal coordination**.

2 Game Requirements (Student-Selected)

Select any **turn-based, text-based game** that satisfies:

- Supports **exactly 3 to 5 players**
- **Server-enforced rules** (no client-side validation)
- **Text-only CLI interface**
- **Clear win/loss/draw condition**
- **Moderate complexity** (e.g., variants of Tic-Tac-Toe, simplified card games, race games, word games)

Important: All randomness (dice, cards) must be generated **by the server only**.

3 Mandatory Concurrency Model: Hybrid (`fork` + Threads)

Your server **must combine**:

3.1 Multiprocessing (via `fork()`)

- For each client that connects or joins, the server **forks a child process** to handle that player's session.
- The parent process **must reap zombies** (e.g., using `SIGCHLD` + `waitpid()`).

3.2 Multithreading (via `pthreads`)

- The **main server process** (parent) **must create at least two internal threads** to handle:
  - **Round Robin turn scheduler** (manages whose turn it is and advances turns)
  - **Concurrent logger** (writes all game events to `game.log`)
- These threads must run **concurrently** with the main accept loop and with each other.
- Threads must **coordinate safely** with `fork()`ed children via **shared memory and synchronization primitives**.

4 Logger Requirements (Thread-Safe & Concurrent)

- Log all events to `game.log`: connections, moves, turn changes, game end.
- The **logger must run in its own thread**.
- All log messages must be:

  - **Complete** (no interleaving)
  - **Ordered** (chronologically consistent)
  - **Non-blocking** to gameplay

- Use **synchronization** (e.g., a mutex or semaphore) to protect the log queue or file access **across threads and processes**.

5 Round Robin Scheduler (Thread-Based)

- A **dedicated scheduler thread** in the parent process must:
  - Maintain the **cyclic player order**
  - Determine the **current player's turn**
  - Signal when a player may act
- Turn state must reside in **shared memory** so `fork()`ed children can read it.
- All updates to turn state must be **synchronized** (mutex/semaphore) and visible across processes.
- The scheduler must **skip disconnected/inactive players**.

6 Architectural & Synchronization Requirements

6.1 IPC & Communication

- **Shared game state** (board, positions, turn, etc.) must reside in **POSIX shared memory**.
- In **single-machine mode**, client-server communication uses **IPC** (e.g., named pipes).
- In **multi-machine mode**, client-server uses **TCP**, but server internals still use shared memory + threads.

6.2 Synchronization Across Domains

- You must use **both**:
  - **Process-shared mutexes/semaphores** (for `fork()`ed children $\leftrightarrow$ parent threads)
  - **Thread mutexes** (for internal thread coordination, if needed)
- All access to shared memory (by threads **or** child processes) must be **mutually exclusive**.
- Initialize shared mutexes with `PTHREAD_PROCESS_SHARED`.

7 Persistent Scoring and History

The server must implement a persistent scoring mechanism to track player statistics across sessions.

7.1 Scores File Requirements

- **Persistent Storage:** The server must maintain player win statistics in a file named `scores.txt`.
- **Loading:** The server must **load the `scores.txt` file into shared memory** upon server startup. If the file does not exist, it should be created.
- **Updating:** At the conclusion of every game, the winning player's score must be **atomically updated** in the shared memory structure.
- **Saving:** The server must write the updated scores from memory back to `scores.txt` upon server shutdown (using signal handling, e.g., `SIGINT`) and potentially after every completed game.

7.2 Concurrency and Synchronization

- The in-memory score structure is a critical shared resource.
- All read and write operations on the score structure must be protected using the **Process-Shared Mutexes/Semaphores** to prevent race conditions, especially when game-end events are triggered by child processes.

8 Deliverables

8.1 A. Source Code (C/C++ only)

- `server.c`, `client.c`, `Makefile`
- Must use `fork()` **and** `pthread_create()`
- Compile on Linux with `gcc -pthread`

8.2 B. Design Report (PDF, 10-15 pages)

Include:

1. **Game description** and rules
2. **Deployment mode** (IPC or TCP)
3. **Hybrid architecture**: diagram showing processes, threads, and data flow
4. **IPC mechanism** and shared memory layout
5. **Synchronization strategy**: how mutexes/semaphores coordinate threads **and** processes

6. **Logger design**: thread structure, queue, safety
7. **RR scheduler**: how the thread manages turns across processes
8. **Persistence Strategy**: Describe the file format, the loading/saving mechanism for `scores.txt`, and the synchronization used to protect the in-memory score data.
9. **Multi-Game Handling**: Explain how the server resets and re-initializes for the start of a new game.
10. **Testing evidence**: gameplay screenshots + sample `game.log`
11. **Screenshots**: screenshots of each client view, part of the logger, and persistent storage content.

8.3 C. README.txt

- How to compile (`make`) and run
- Example commands
- Game rules summary
- Mode supported

8.4 D. Video Demonstration

- A video recording (max **5 minutes**) demonstrating the following:
  1. Compilation using `make`.
  2. Running the server and connecting the minimum number of clients (3 players).
  3. Demonstrating a few full rounds of gameplay.
  4. Showing the concurrent logging (`game.log`) occurring in real-time.
  5. Showing the updated `scores.txt` file after a game ends.
  6. Each student must present his part of the project as provided in the table of responsibilities (8.5 below).

8.5 E. Team Responsibilities

The team must consist of 3 to 4 members from the same tutorial/lab section. Include the following table detailing the division of labor. Ensure all critical components are assigned. If the team has 4 members, add an extra row to the table.

Team Member Responsibilities

Name/ID	**Primary Role**	**Key Components Developed/Implemented**
[Member 1 Name/ID]	[e.g., Server Core, IPC]	[List specific components: `fork()`, shared memory setup, logger thread]
[Member 2 Name/ID]	[e.g., Client/Game Logic]	[List specific components: Client handler, game state rules, scheduler thread]
[Member 3 Name/ID]	[e.g., Persistence/Networking]	[List specific components: `scores.txt` management, TCP/IPC connection logic]

9 Critical Constraints

- **Both `fork()` AND `pthreads` are required**. Omitting either results in significant point loss.
- **No pure-thread or pure-process solutions**.
- **Shared mutexes must be initialized with** `PTHREAD_PROCESS_SHARED` to work across `fork()`.
- **The logger and scheduler must be threads in the parent process**—not separate processes.
- **All players must be handled in child processes**—not threads.

10 Evaluation Criteria (Total: 100 pts) - Detailed Rubric

The assignment will be graded based on the technical implementation of concurrency and synchronization, adherence to the hybrid model, and the quality of the deliverables.

Table 1: Detailed Operating Systems Assignment Rubric

Component	**Fail (0 Pts)**	**Poor (Partial Pts)**	**Good (Intermediate Pts)**	**Excellent (Max Pts)**	**Max Pts**
A. Group Mark (Technical Core - 75 pts)					
Multiprocessing (`fork`)	Sequential processing or no `waitpid` used.	**2 Pts:** Forking exists but zombie reaping is unstable/blocking.	**3 Pts:** Stable forking per client; functional, non-blocking zombie reaping.	**4-5 Pts:** Stable forking; `SIGCHLD`-based zombie reaping; proper error handling.	5

Table 1: Detailed Operating Systems Assignment Rubric

Component	**Fail (0 Pts)**	**Poor (Partial Pts)**	**Good (Intermediate Pts)**	**Excellent (Max Pts)**	**Max Pts**
Multithreading (`pthreads`)	Server uses only processes or only one internal thread.	**3 Pts:** Two threads exist but are unstable, frequently blocking, or fail often.	**6 Pts:** Two threads exist, run concurrently, but exhibit minor coordination issues.	**8-10 Pts:** Scheduler and Logger threads correctly created, stable, concurrent, and fulfill their roles perfectly.	10
IPC & Shared Memory	No shared memory or critical data stored locally per process.	**3 Pts:** Shared memory set up, but only partial critical data stored, or IPC is unreliable.	**6 Pts:** Shared memory holds all critical state; communication mechanism works.	**8-10 Pts:** Shared memory correctly initialized and efficiently used for all critical state; IPC/TCP is robust and reliable.	10
Cross-Domain Synchronization	Shared memory used without any synchronization (guaranteed race conditions).	**5 Pts:** Synchronization used, but incorrect type (e.g., non-shared mutex) or not protecting all critical sections.	**10 Pts:** Synchronization used correctly on all critical sections.	**12-15 Pts:** All accesses to shared memory are fully protected using the correct primitives; system is provably safe and deadlock-free.	15
Round Robin Scheduler (Thread)	Turn logic is handled sequentially by client processes, or fails to cycle.	**3 Pts:** Scheduler thread exists but updates turn state without synchronization or fails to cycle.	**6 Pts:** Scheduler correctly manages turn order and uses synchronization, but player skipping is buggy.	**8-10 Pts:** Dedicated thread fully implements RR, safely updates shared state, and reliably skips inactive players.	10

Table 1: Detailed Operating Systems Assignment Rubric

Component	**Fail (0 Pts)**	**Poor (Partial Pts)**	**Good (Intermediate Pts)**	**Excellent (Max Pts)**	**Max Pts**
Concurrent Logger (Thread)	Logging is performed in-line by client processes, or file synchronization is absent.	**3 Pts:** Logger thread exists but exhibits file corruption (interleaved logs) or blocking behavior.	**6 Pts:** Logger thread is safe and functional, but may occasionally cause brief delays or logs are not fully ordered.	**8-10 Pts:** Dedicated thread ensures complete, ordered, non-blocking logging to `game.log` using a highly efficient synchronized queue/mechanism.	10
Persistent Scoring & Multi-Game	Scoring is volatile or server crashes after one game.	**3 Pts:** Scores load/save correctly, but updates are not protected (race condition risk), or game reset is buggy.	**6 Pts:** Scores are saved/loaded; updates are protected; system supports multiple games.	**8-10 Pts:** `scores.txt` correctly handled; score updates are **atomically protected**; server reliably resets and handles successive games flawlessly.	10
Student-Chosen Game	Game is unplayable or violates mandatory player constraints.	**2 Pts:** Game is playable but basic (e.g., simple tic-tac-toe) or rules are unclear.	**3 Pts:** Game is functional, meets all constraints, and rules are clear.	**4-5 Pts:** Game is well-designed, functional, meets all constraints, and demonstrates moderate complexity.	5
B. Individual/Deliverable Mark (25 pts)					

Table 1: Detailed Operating Systems Assignment Rubric

Component	**Fail (0 Pts)**	**Poor (Partial Pts)**	**Good (Intermediate Pts)**	**Excellent (Max Pts)**	**Max Pts**
Code Quality & Build	Student did not contribute to both the coding and the report (must contribute equally to both)	**3 Pts:** Student's part of the code is present but does not perform all the expected tasks.	**6 Pts:** Student's code performs the majority of the tasks as expected	**8-10 Pts:** Excellent, professional-quality code with consistent style, comprehensive comments, and a fully functional Makefile.	10
Deliverables Quality	Critical deliverables (Report or Video) are missing.	**5 Pts:** Deliverables present but lack detail (Report $\leq$ 2 pages, video $\leq$ 3 required points shown).	**10 Pts:** All deliverables present; Report is 10 to 15 pages, video shows most required points, roles defined.	**12-15 Pts:** All deliverables are professional and complete; Report is 4-5 pages, video clearly demonstrates all functional points, and roles are clearly defined and balanced.	15
Total Points: 100					

11 Why This Design?

Real-world systems (e.g., web servers, databases) often combine processes (for fault isolation) and threads (for efficiency). This assignment gives you hands-on experience with **complex concurrency orchestration**—a hallmark of robust OS-level programming.

Policies of AI Tool Usage

12 Preamble

These policies govern the responsible, ethical, and academically honest use of organizational internet resources and Artificial Intelligence (AI) tools, specifically for the CSN6214 Operating Systems Assignment. These rules are mandatory and failure to comply will result in disciplinary action.

13 Artificial Intelligence (AI) Tools Usage Policy

I. Academic Integrity and Attribution (Students)

13.1 Disclosure and Citation (Permitted Use):

- AI tools (e.g., ChatGPT, Bard, Copilot) may be used *only* for general concept understanding (e.g., "Explain synchronization primitives") or to generate draft documentation text.
- Any text, diagrams, or utility code generated by an AI tool and included in the **Design Report** or submitted code **must be clearly and accurately cited**. Failure to disclose is plagiarism.

13.2 Prohibition on Core Logic Generation (Strictly Forbidden):

- AI tools **must not** be used to generate the core, architectural, and graded components of this assignment.
- Prohibited Core Components include: Hybrid concurrency logic (`fork()` and `pthreads`), shared memory setup, cross-domain synchronization logic, Round Robin Scheduler logic, and Concurrent Logger logic.

II. Data Privacy and Confidentiality

13.3 No Sensitive Data Input: Users must **never input, upload, or paste confidential, proprietary, or personally identifiable information (PII)** into any third-party AI tool. Data submitted is often used to train the model and may not remain private.

13.4 Verification of Output: Users must **independently verify** all facts, data, and sources generated by AI tools. AI output may contain errors or "hallucinations" (false information).

---

14 Academic Honesty and Code Originality Policy

I. Prohibition on Plagiarism and Code Copying

14.1 Code Originality: The submitted source code (`server.c`, `client.c`, `Makefile`) **must be original work** created solely by the members of the registered group.

14.2 Unauthorized Duplication: Copying code or solutions, even partially, from another student, another team, or external online sources (e.g., GitHub, Stack Overflow) that address the core assignment components is strictly forbidden.

II. Strict Secrecy and Zero Inter-Group Communication

14.3 Total Secrecy: Teams must maintain **absolute secrecy** regarding their project design, implementation, and code.

14.4 No External Discussion: You must **not** discuss the assignment problem, potential solutions, architectural decisions, synchronization strategies, or implementation details with any student outside of your registered group.

14.5 Internal Collaboration Only: All collaboration must be strictly confined to the 3 to 4 registered members of your team.

15 Enforcement and Consequences

Violations: Failure to comply with any part of these policies (including undisclosed AI use, plagiarism, or inter-group communication) will result in disciplinary action, including but not limited to:

- A grade of **zero (0)** for the entire assignment for all involved students.
- Further disciplinary action in line with academic integrity policies.

Note: Students are responsible for reading and understanding all sections of this policy prior to beginning work on the assignment.