Assignment: Multithreaded RPC-Based Distributed Task Processing System

Course: CSE 303 Fundamentals of Operating Systems

Due: December 17th, 2025

1. Objective

In this assignment you will combine **Remote Procedure Call (RPC)** with **multithreading** to build a distributed task-processing system. A client will submit tasks to a server via RPC. The server places incoming tasks into a shared queue and processes tasks using multiple worker threads concurrently.

2. System Overview

You will implement a **Task Server** and a **Task Client**:

- The *client* submits tasks to the server and later retrieves results.
- The *server* accepts tasks via RPC and processes them using a pool of worker threads.
- Communication between client and server must use RPC generated with `rpcgen`.
- The server's task queue and result storage must be thread-safe.

3. Supported Task Types

Each submitted task contains a `type` and a `payload`. The server will use the type to determine the operation and the payload as input.

1. **Reverse String**
2. **Sum of Integer List**
3. **Fibonacci Number**

For grading and testing, the following concrete examples must be supported by your implementation:

- **Task Type 1: Reverse String**

  ```
  payload: "hello world"
  expected result: "dlrow olleh"
  ```

- **Task Type 2: Sum of Integer List**

```
payload: "5 7 1 12 4"
expected result: "29"
```

- **Task Type 3: Fibonacci Number**

  ```
  payload: "20"
  expected result: "6765"
  ```

4. RPC Interface (`task.x`)

Create an RPC definition file named `task.x`. The following specification is required:

```
struct task {
    int id;
    int type;           // 1, 2, or 3
    string payload<256>;
};

struct result {
    int id;
    string output<256>;
};

program TASKPROG {
    version TASKVERS {
        int SUBMIT_TASK(task) = 1;
        result GET_RESULT(int) = 2;
    } = 1;
} = 0x23451111;
```

Use `rpcgen` to generate the client and server stubs from `task.x`. Keep payload size limits in mind (the rpcgen string limit in the specification is 256 bytes).

5. Server Requirements

- Start with at least **3 worker threads**.
- Maintain a thread-safe FIFO task queue (use `pthread_mutex_t` or `pthread_cond_t`).
- Each worker thread must: wait for tasks, pop a task from the queue, process it, and store the result in a thread-safe result store.
- Generate unique incremental task IDs for each submitted task.
- Support concurrent submissions from multiple clients.
- If a client requests a result that is not yet ready, return a `"PENDING"` indicator in the result output.

6. Client Requirements

The client program should:

1. Prompt the user to choose a task type (1–3).
2. Prompt for the required payload (string or space-separated integers).
3. Call `SUBMIT_TASK()` via RPC and print the returned task ID.
4. Allow the user to call `GET_RESULT(task_id)` to fetch the result.
5. Support submitting multiple tasks in one client session.

7. Implementation Details and Constraints

- Language: **C**
- RPC: `rpcgen`
- Threads: `pthreads`
- Synchronization primitives: `pthread_mutex_t`, `pthread_cond_t`
- Platform: Linux (Ubuntu or similar) or WSL
- Provide a `Makefile` that builds both server and client binaries. (just use the name Makefile and not Makefile.yourAppName)

8. Deliverables

Submit a compressed archive containing:

- `task.x`
- Server source files (e.g., `server.c`, `server_impl.c`)
- Client source files (e.g., `client.c`)
- Source files for the task queue and result storage
- `Makefile`
- `README.md` with compilation and execution instructions and a short design description

9. Grading Rubric (100 points)

RPC interface and rpcgen usage	20
Thread-safe task queue	20
Worker thread correctness	20
Correct task processing	15
Reliable client–server interaction	15
Code quality + README and Documentation	10

10. Testing Recommendations

Provide a short test plan in your README that includes:

- Submitting each of the four example tasks and verifying correct results.
- Submitting multiple tasks concurrently from multiple client instances.
- Verifying that `GET_RESULT` returns `"PENDING"` for incomplete tasks and the computed result when ready.

Good luck.