Multithreaded RPC-Based Distributed Task Processing System

CSE 303 - Assignment 2

Fundamentals of Operating Systems

Student: Burak Yalçın

ID: 20220808069

Date: December 2025

The Challenge

What We Need to Build

A distributed task processing system that combines two fundamental OS concepts:

- **RPC (Remote Procedure Call):** Allows clients to submit tasks to a remote server as if calling local functions
- **Multithreading:** Server uses multiple worker threads to process tasks concurrently

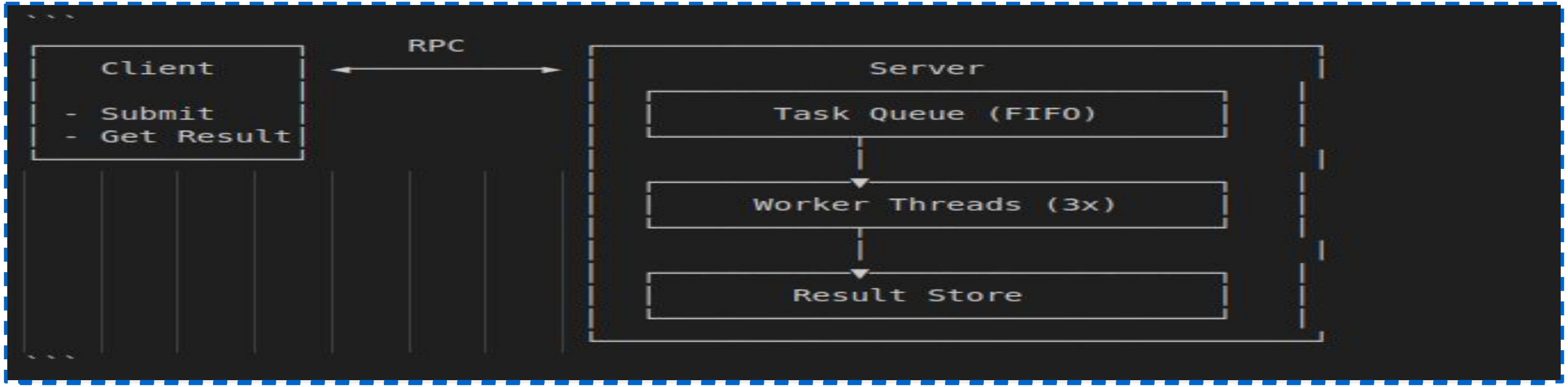
Why This Is Difficult

- RPC requires understanding network communication, serialization, and the rpcgen code generation tool
- Multithreading introduces concurrency challenges: race conditions, deadlocks, and synchronization issues
- Combining both means managing shared data structures (task queue, result store) accessed by multiple threads simultaneously
- Must ensure asynchronous operation: clients can submit tasks and retrieve results independently

> **The Core Problem:** How do we safely coordinate multiple threads accessing shared resources while maintaining efficient, asynchronous communication between client and server?

System Architecture

The system follows a client-server model where clients submit tasks via RPC, the server manages a task queue, and worker threads process them concurrently.



Client

Submits tasks and retrieves results via RPC calls

RPC Listener

Receives client requests and acts as producer

Task Queue

Thread-safe FIFO queue for pending tasks

Worker Threads (3x)

Consumer threads that process tasks from queue

Result Store

Array-based storage for completed results

Synchronization

Mutex and condition variables for thread safety

Phase 1: RPC Framework Setup

1 RPC Interface Definition (task.x)

Defined the communication contract between client and server

Created two data structures: **task** (client request) and **result** (server response)

Defined two remote procedures: **SUBMIT_TASK** and **GET_RESULT**

```
struct task { int id; int type; string payload<256>; };
program TASKPROG { version TASKVERS { int SUBMIT_TASK(task) = 1; result GET_RESULT(int) = 2; } = 1; } = 0x23451111;
```

2 Code Generation with rpcgen

Ran **rpcgen task.x** to automatically generate client and server stubs

Generated files: **task.h** (data structures), **task_clnt.c** (client stubs), **task_svc.c** (server stubs)

These stubs handle all RPC protocol details (serialization, network communication, TCP/UDP)

3 Initial Testing

Created a minimal client to test basic RPC connection before adding multithreading

Verified that SUBMIT_TASK calls successfully reached the server

Confirmed that task IDs were returned correctly

This baseline ensured the RPC framework was working before Phase 2 complexity

Phase 2: Core Data Structures

1. Task Queue

Singly Linked List (FIFO)

Implemented as a linked list to maintain FIFO order. The RPC listener pushes tasks to the back, worker threads pop from the front.

> **Why linked list?** Dynamic size without pre-allocation. Efficient O(1) push and pop operations at both ends. FIFO ensures fairness: first submitted tasks are processed first.

2. Result Store

Array-Based Storage

An array with a maximum capacity of 1000 results. Task IDs are sequential integers, making array indexing straightforward.

> **Why array?** O(1) lookup by task ID. When a client calls GET_RESULT, we can immediately check if the result exists without searching. Much faster than a linked list for retrieval.

3. Task Processor

Isolated Processing Logic

Separate functions for each task type (reverse_string, sum_integers, fibonacci). Keeps server logic clean and modular.

> **Why separate?** Isolating task processing logic from server/threading logic makes debugging easier and allows independent testing of each task type before integration.

Phase 3: Multithreading Integration

1. Worker Thread Pool Creation

Created a fixed pool of 3 worker threads using pthread_create():

```
for (int i = 0; i < 3; i++) {
 pthread_create(&worker_threads[i], NULL,
 worker_thread_func, NULL);
 }
```

Each worker runs an infinite loop continuously popping tasks from the queue

Threads remain alive for the server's entire lifetime

2. Mutex Protection

A pthread_mutex_t protects all access to shared data structures:

Task queue (push/pop operations)

Result store (put/get operations)

Prevents race conditions where multiple threads access data simultaneously

3. Condition Variables for Efficiency

Used pthread_cond_t to avoid busy-waiting:

```
// Worker thread waits when queue is empty
 while (queue_is_empty()) {
 pthread_cond_wait(&cond_var, &mutex);
 }
 // RPC listener signals after adding task
 pthread_cond_signal(&cond_var);
```

Workers sleep when no tasks available (no CPU waste)

Listener wakes one worker when task arrives

Producer-Consumer Flow

Producer (RPC Listener): Receives task ⟶ acquires mutex ⟶ pushes to queue ⟶ signals condition variable ⟶ releases mutex

Consumer (Worker): Acquires mutex ⟶ waits if queue empty ⟶ pops task ⟶ releases mutex ⟶ processes task ⟶ stores result

Synchronization Mechanisms

1. pthread_mutex_t (Mutual Exclusion)

A mutex ensures only one thread enters a critical section, preventing race conditions.

```
pthread_mutex_lock(&mutex);
/* critical section */
pthread_mutex_unlock(&mutex);
```

Protects the task queue from concurrent modifications

3. Producer-Consumer Flow

```
/* Producer */
pthread_mutex_lock(&mutex);
queue.push(task);
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);

/* Consumer */
pthread_mutex_lock(&mutex);
while (queue_is_empty) pthread_cond_wait(&cond, &mutex);
task = queue.pop();
pthread_mutex_unlock(&mutex);
```

2. pthread_cond_t (Condition Variables)

Condition variables let threads wait efficiently without busy-waiting; used with mutexes.

```
pthread_mutex_lock(&mutex);
while (queue_is_empty) pthread_cond_wait(&cond, &mutex);
task = queue.pop();
pthread_mutex_unlock(&mutex);
```

Worker threads sleep when queue is empty (no CPU waste)

Challenges and Solutions

1. ### Race Conditions in Task Queue

Problem

Worker threads occasionally crashed or returned incorrect results due to concurrent access to the linked list pointers in the task queue. Multiple threads modifying the queue simultaneously caused data corruption.

Solution

Ensured that pthread_mutex_t was acquired **before** checking the queue status and **only** released after the entire pop operation was complete. This guaranteed atomic access to the queue structure.

2. ### Deadlock Risk with Condition Variables

Problem

Potential for deadlock if the mutex was held while waiting on the condition variable, causing threads to block indefinitely.

Solution

Used pthread_cond_wait() which atomically releases the mutex and blocks the thread, then reacquires the mutex upon waking. This prevents deadlock while maintaining thread safety.

3. ### RPC Integration with Thread Pool

Problem

Integrating the rpcgen generated server stubs (submit_task_1_svc) with the custom thread pool logic was unclear. The generated code needed to act as the producer.

Solution

Modified the submit_task_1_svc function to acquire the mutex, push the task to the queue, and signal the condition variable. This made the RPC handler the producer in the producer-consumer model.

Key Design Decisions

Decision	Choice	Rationale
Result Store Structure	Array (O(1) lookup)	Task IDs are sequential integers. Array indexing provides instant O(1) retrieval when clients call GET_RESULT. Much faster than searching a linked list.
Worker Waiting Mechanism	Condition Variables	Avoids busy-waiting (polling the queue repeatedly). Workers sleep when queue is empty, consuming zero CPU. Woken by signal when task arrives.
Thread Pool Size	Fixed Pool (3 threads)	Meets assignment requirement. Simplifies resource management compared to dynamic pool. 3 threads provide good concurrency without excessive context switching.
Asynchronous Operation	PENDING Status	GET_RESULT returns "PENDING" if task not yet complete. Allows clients to submit multiple tasks and check results independently without blocking.
Task Queue Structure	Linked List (FIFO)	Dynamic size without pre-allocation. O(1) push/pop at both ends. FIFO ordering ensures fairness: first submitted tasks processed first.
RPC Communication	TCP Protocol	Reliable, ordered delivery. Ensures no task submissions are lost. Suitable for a task processing system where reliability is critical.

Testing and Verification

Comprehensive testing was performed to verify correctness of all task types, thread-safe operations, and concurrent request handling. Each test scenario was executed multiple times to ensure consistency.

Test Scenario	Input	Expected Output	Actual Output	Status
Reverse String	"hello world"	"dlrow olleh"	"dlrow olleh"	✓ PASS
Sum Integers	"5 7 1 12 4"	"29"	"29"	✓ PASS
Fibonacci(20)	"20"	"6765"	"6765"	✓ PASS
PENDING Status	Long-running task	"PENDING"	"PENDING"	✓ PASS
Concurrent Submission (5 clients)	Multiple simultaneous tasks	All tasks processed correctly	All tasks processed correctly	✓ PASS

```
=== Task Processing Client ===
1. Submit Task (Type 1: Reverse String)
2. Submit Task (Type 2: Sum Integers)
3. Submit Task (Type 3: Fibonacci)
4. Get Result
5. Exit
Choose an option: 1
Enter string to reverse: hello world
Task submitted successfully! Task ID: 4

=== Task Processing Client ===
1. Submit Task (Type 1: Reverse String)
2. Submit Task (Type 2: Sum Integers)
3. Submit Task (Type 3: Fibonacci)
4. Get Result
5. Exit
Choose an option: 4
Enter task ID: 4
Task 4 result: dlrow olleh
```

```
=== Task Processing Client ===
1. Submit Task (Type 1: Reverse String)
2. Submit Task (Type 2: Sum Integers)
3. Submit Task (Type 3: Fibonacci)
4. Get Result
5. Exit
Choose an option: 2
Enter space-separated integers: 5 7 1 12 4
Task submitted successfully! Task ID: 5

=== Task Processing Client ===
1. Submit Task (Type 1: Reverse String)
2. Submit Task (Type 2: Sum Integers)
3. Submit Task (Type 3: Fibonacci)
4. Get Result
5. Exit
Choose an option: 4
Enter task ID: 5
Task 5 result: 29
```

```
=== Task Processing Client ===
1. Submit Task (Type 1: Reverse String)
2. Submit Task (Type 2: Sum Integers)
3. Submit Task (Type 3: Fibonacci)
4. Get Result
5. Exit
Choose an option: 3
Enter n for Fibonacci(n): 20
Task submitted successfully! Task ID: 6

=== Task Processing Client ===
1. Submit Task (Type 1: Reverse String)
2. Submit Task (Type 2: Sum Integers)
3. Submit Task (Type 3: Fibonacci)
4. Get Result
5. Exit
Choose an option: 4
Enter task ID: 6
Task 6 result: 6765
```

```
=== Task Processing Client ===
1. Submit Task (Type 1: Reverse String)
2. Submit Task (Type 2: Sum Integers)
3. Submit Task (Type 3: Fibonacci)
4. Get Result
5. Exit
Choose an option: 4
Enter task ID: 7
Task 7 result: PENDING
```

Conclusion

Key Achievements

Successfully implemented RPC communication using rpcgen with proper task.x interface definition and TCP protocol

Designed and implemented thread-safe data structures (task queue and result store) using pthread_mutex_t and pthread_cond_t

Created a worker thread pool (3 threads) that processes tasks concurrently and correctly handles all three task types

Verified system reliability under concurrent load from multiple clients with proper PENDING status handling

Lessons Learned

Race conditions are subtle and difficult to debug in multithreaded programs; proper synchronization primitives are essential

Condition variables are far more efficient than busy-waiting; understanding pthread_cond_wait atomicity is critical

Separating concerns (RPC layer, data structures, task processing) makes integration and debugging much easier

Testing concurrent systems requires running tests multiple times; race conditions may not appear every run

THANK YOU SIR.

BURAK YALÇIN

20220808069