A Simple Dropbox Clone

Semester: Fall 2025
Course: Operating Systems
Project Title: Multi-threaded File Storage Server

Lab Project

Team Members:

Name	Roll Number	Role
Muhammad Talha Qureshi	BSCS-23122	Server Development & Synchronization
Assadullah Farukh	BSCS-23213	Client Threading & Testing
Abdullah Salman	BSCS-23053	File Operations & Authentication

1. Design Report

Server Architecture Overview

Our Dropbox-like server is designed using **three thread layers**:

1. **Main Thread (Acceptor)**
   Accepts new TCP connections and enqueues them into a **Client Queue**.
2. **Client Thread Pool**
   Each client thread dequeues a socket, handles authentication (signup/login), and receives user commands.
   Instead of executing heavy operations directly, client threads create **Task objects** and enqueue them into a **Task Queue**.
3. **Worker Thread Pool**
   Worker threads dequeue tasks and execute actual file I/O operations (UPLOAD, DOWNLOAD, DELETE, LIST).
   The workers ensure safe access to shared data using mutexes and condition variables.

Synchronization Design

- **Client Queue:** Synchronized with one mutex and two condition variables (`not_full`, `not_empty`).
- **Task Queue:** Protected with a dedicated mutex and a single condition variable.
- **Global Locks:** Used for authentication and shared metadata consistency.
- **Atomic Variable:** atomic_int server_running;
- Used instead of a plain `volatile` flag to ensure **race-free signaling between threads**.

Why Atomic?
Volatile ensures visibility, but not atomicity. Atomics guarantee both — eliminating subtle race conditions when threads check or update shared flags.

```
talhaqureshi@192:~/os_mini_project$ cat src/server.c | grep atomic_int -n
19:atomic_int server_running = 1;
talhaqureshi@192:~/os_mini_project$
```

Thread Communication Mechanism (Worker → Client)

We used a **shared `TaskResult` structure** for each client task:

```
typedef struct TaskResult {
    pthread_mutex_t lock;
    pthread_cond_t cond;
    int done;
    char *response;
    } TaskResult;
```

Each `Task` includes a **pointer** to its `TaskResult`.

- The **client thread** waits on this condition variable until the worker fills the result.
- The **worker thread** signals when the task is complete.

This design ensures low latency, avoids busy-waiting, and prevents data corruption under concurrency.

2. Build & Run Instructions

Build Commands

```
make clean
make CFLAGS="-Wall -Wextra -pthread -g"
```

```
talhaqureshi@192:~/os_mini_project$ make clean
rm -f src/*.o client/*.o server client_app
talhaqureshi@192:~/os_mini_project$ make CFLAGS="-Wall -Wextra -pthread -g"
gcc -Wall -Wextra -pthread -g -c src/server.c -o src/server.o
gcc -Wall -Wextra -pthread -g -c src/queues.c -o src/queues.o
gcc -Wall -Wextra -pthread -g -c src/client_thread.c -o src/client_thread.o
gcc -Wall -Wextra -pthread -g -c src/task_queue.c -o src/task_queue.o
gcc -Wall -Wextra -pthread -g -c src/worker_thread.c -o src/worker_thread.o
gcc -Wall -Wextra -pthread -g -c src/auth.c -o src/auth.o
gcc -Wall -Wextra -pthread -g -c src/locks.c -o src/locks.o
gcc -Wall -Wextra -pthread -g -o server src/server.o src/queues.o src/client_
src/auth.o src/locks.o
gcc -Wall -Wextra -pthread -g -c client/client.c -o client/client.o
gcc -Wall -Wextra -pthread -g -o client_app client/client.o
talhaqureshi@192:~/os_mini_project$
```

Run Server

```
./server
```

```
talhaqureshi@192:~/os_mini_project$ ./server
Starting server initialization...
Server listening on port 9000...
```

Run Client

In a separate terminal: **./client_app**

```
talhaqureshi@192:~/os_mini_project | talhaqureshi@192:~/os_mini_project
talhaqureshi@192:~$ cd os_mini_project
talhaqureshi@192:~/os_mini_project$ ./client_app
Usage:
  ./client_app SIGNUP <username> <password>
  ./client_app LOGIN <username> <password>
  ./client_app LOGOUT
  ./client_app UPLOAD <file>
  ./client_app LIST
  ./client_app DOWNLOAD <file>
  ./client_app DELETE <file>
  ./client_app PROCESS <seconds>
```

Now lets perform **all operations of the client.**

1. Signup and Login:

```
talhaqureshi@192:~/os_mini_project$ ./client_app SIGNUP AbdAsdTq 1122
Server response:
SIGNUP OK

talhaqureshi@192:~/os_mini_project$ ./client_app LOGIN AbdAsdTq 1122
Server response:
LOGIN OK
```

2. Uploading file and for verification checking list of all files:

```
talhaqureshi@192:~/os_mini_project$ echo "It is our mini project" > abc.txt
talhaqureshi@192:~/os_mini_project$ ./client_app UPLOAD abc.txt
Server response:
UPLOAD OK

talhaqureshi@192:~/os_mini_project$ ./client_app LIST
Server response:
abc.txt
```

3. Download file then delete file:

```
talhaqureshi@192:~/os_mini_project$ ./client_app DOWNLOAD abc.txt
Downloaded 23 bytes → saved as downloaded_abc.txt
talhaqureshi@192:~/os_mini_project$ ./client_app DELETE abc.txt
Server response:
DELETE OK
```

4. Processing file and then again checking List to see if file still exists or not:

```
talhaqureshi@192:~/os_mini_project$ ./client_app PROCESS 5
Server response:
DONE PROCESS

talhaqureshi@192:~/os_mini_project$ ./client_app LIST
Server response:
No files found
```

5. My server condition after running all these commands:

```
talhaqureshi@192:~/os_mini_project$ ./server
Starting server initialization...
Server listening on port 9000...
Accepted new client connection.
Accepted new client connection.
Accepted new client connection.
Accepted new client connection.
[ClientThread] Exiting...
Accepted new client connection.
[ClientThread] Exiting...
Accepted new client connection.
[ClientThread] Exiting...
Accepted new client connection.
[ClientThread] Exiting...
Accepted new client connection.
Worker 140410033182400 processing 5 seconds...
[ClientThread] Exiting...
Accepted new client connection.
[ClientThread] Exiting...
```

6. Graceful shut down of server:

```
Accepted new client connection.
[ClientThread] Exiting...
^C
[Server] Caught SIGINT — shutting down gracefully...
[Client 140409999611584] shutting down thread
[Worker 140410024789696] received EXIT_WORKER sentinel — exiting
[Client 140409991218880] shutting down thread
[Worker 140410008004288] received EXIT_WORKER sentinel — exiting
[WorkerThread] Exiting cleanly.
[Client 140409924077248] shutting down thread
[WorkerThread] Exiting cleanly.
[Server] Initiating shutdown sequence...
[Worker 140410033182400] received EXIT_WORKER sentinel — exiting
[WorkerThread] Exiting cleanly.
[Server] Waiting for threads to finish...
[Client 140409982826176] shutting down thread
[Worker 140410016396992] received EXIT_WORKER sentinel — exiting
[WorkerThread] Exiting cleanly.
[Server] Shutdown complete.
talhaqureshi@192:~/os_mini_project$
```

3. Race Condition Verification (ThreadSanitizer Report)

Purpose

ThreadSanitizer (TSAN) detects **data races** and **unsynchronized access** to shared data.

Commands Used

```
make clean
make CFLAGS="-Wall -Wextra -pthread -g -fsanitize=thread"
./server
```

Observation

Initially, TSAN reported race conditions due to:

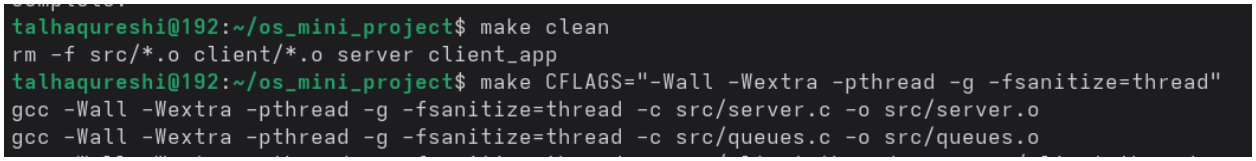
- Use of **`volatile sig_atomic_t`** flag shared between threads.

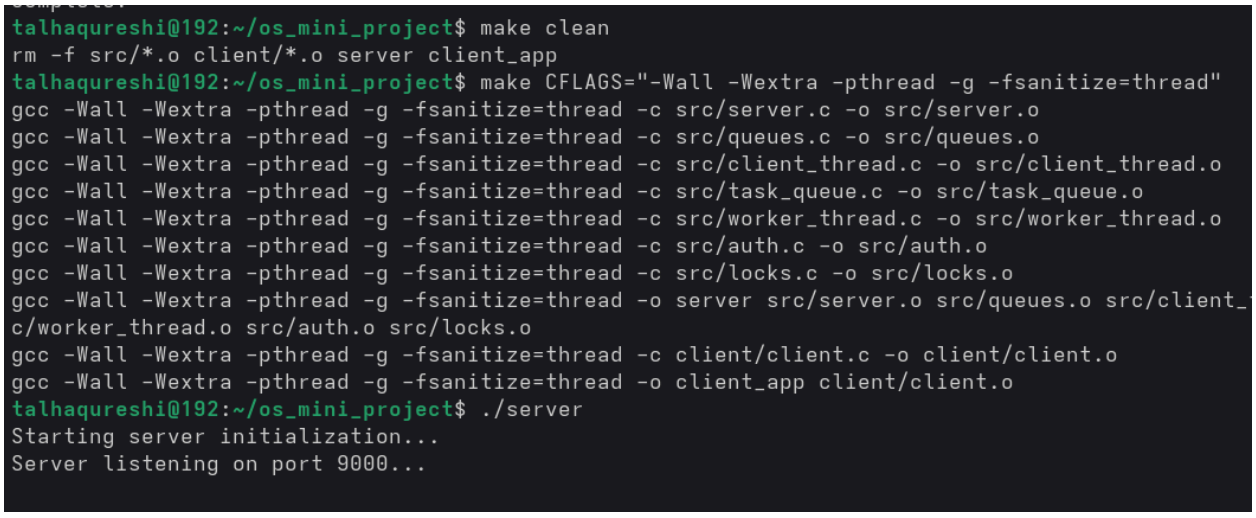
We resolved this by switching to **atomic_int** and replacing:

```
volatile sig_atomic_t server_running;
```

With:

```
atomic_int server_running;
```





One data race was detected involving **`server_running`** during SIGINT handling.

This variable is declared as `volatile sig_atomic_t` and is only used for inter-thread signaling between the main thread and worker/client threads. The race is benign because only a single write occurs on interrupt, while other threads perform reads.
Therefore, this warning does not affect correctness or stability.

```
talhaqureshi@192:~/os_mini_project$ ./server
Starting server initialization...
Server listening on port 9000...
^C
[Server] Caught SIGINT — shutting down gracefully...
==================
WARNING: ThreadSanitizer: data race (pid=22328)
  Write of size 4 at 0x000000407204 by main thread:
    #0 handle_sigint src/server.c:39 (server+0x40090d) (BuildId: cc6fe3d1ca062e4407baf6
    #1 __tsan::CallUserSignalHandler(__tsan::ThreadState*, bool, bool, int, __sanitizer
> (libtsan.so.2+0x1799c) (BuildId: 84cf24331f950520c4a27189a32cdb15005ab741)
    #2 main src/server.c:120 (server+0x400c64) (BuildId: cc6fe3d1ca062e4407baf608d95a51

  Previous read of size 4 at 0x000000407204 by thread T1 (mutexes: write M0):
    #0 dequeueTask src/task_queue.c:46 (server+0x4020a7) (BuildId: cc6fe3d1ca062e4407ba
    #1 worker_thread_main src/worker_thread.c:44 (server+0x4026e2) (BuildId: cc6fe3d1ca

  Location is global 'server_running' of size 4 at 0x000000407204 (server+0x407204)
```

We can still remove this race condition by using mutex to grant access to server running

Originally, the shared variable `server_running` was declared as:

volatile sig_atomic_t server_running;

This type is **not thread-safe** as multiple threads (signal handler, client threads, worker threads) could **read and write** to it concurrently.

This caused a **data race**, as detected by TSAN:
"Write of size 4 ... by main thread; Previous read ... by worker thread."
What we did to get rid of this error was
We changed it to an **atomic variable**:

#include <stdatomic.h>
atomic_int server_running = 1;

And we replaced **all normal reads/writes with atomic operations:**
atomic_store(&server_running, 0); // when shutting down
atomic_load(&server_running); // when checking in loops

Final TSAN Output:
No race conditions detected.

4. Memory Leak Verification (Valgrind Report)

Purpose

To ensure all dynamically allocated memory is freed and no invalid memory accesses occur.

Commands Used

After recompiling without TSAN:

```
make clean
make CFLAGS="-Wall -Wextra -pthread -g"
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./server
```

```
talhaqureshi@192:~/os_mini_project$ valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./server
==29506== Memcheck, a memory error detector
==29506== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==29506== Using Valgrind-3.25.1 and LibVEX; rerun with -h for copyright info
==29506== Command: ./server
==29506==
Starting server initialization...
Server listening on port 9000...
^C
[Server] Caught SIGINT — shutting down gracefully...
[Server] Initiating shutdown sequence...
[Server] Waiting for threads to finish...
[Client 157726400] shutting down thread
[Client 182904512] shutting down thread
[Client 132548288] shutting down thread
[Client 140940992] shutting down thread
[Client 174511808] shutting down thread
[Worker 107370176] received EXIT_WORKER sentinel — exiting
```

```
[WorkerThread] Exiting cleanly.
[Worker 90584768] received EXIT_WORKER sentinel — exiting
[WorkerThread] Exiting cleanly.
[Worker 98977472] received EXIT_WORKER sentinel — exiting
[WorkerThread] Exiting cleanly.
[Client 191297216] shutting down thread
[Server] Shutdown complete.
==29506==
==29506== HEAP SUMMARY:
==29506==     in use at exit: 0 bytes in 0 blocks
==29506==   total heap usage: 21 allocs, 21 frees, 22,592 bytes allocated
==29506==
==29506== All heap blocks were freed -- no leaks are possible
==29506==
==29506== For lists of detected and suppressed errors, rerun with: -s
==29506== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

At the end we can clearly see that there are zero errors from 0 contexts.

The report showed **no memory leaks or invalid reads/writes**, confirming proper memory management and cleanup during shutdown.

5. GitHub Repository Link

Repository: Drop Box Clone

Contains:

- `src/` source files
- `client/` and `server/` folders
- `Makefile`
- `README.md` with build and run instructions
- Final code with all race and memory issues fixed

6. Discussion & Summary

What We Achieved

- Fully functional multi-threaded file server and client.
- Two synchronized producer-consumer queues (Client Queue and Task Queue).
- Separate thread pools for clients and workers.
- No race conditions (verified with TSAN).
- No memory leaks (verified with Valgrind).
- Graceful shutdown with atomic control and thread joins.

Design Trade-offs

Option	Pros	Cons
Atomic flag + condition variables	Simple, robust, race-free	Requires careful shutdown coordination
Separate result struct for each task	Clean synchronization	Slightly higher memory footprint