Programming Assignment 3
Reader-Writer Log

Objective

By completing this assignment, you will be able to:

- Build a synchronization monitor using `pthread_mutex_t` and `pthread_cond_t`.
- Enforce a writer-preference reader–writer policy so neither readers nor writers starve.
- Maintain a shared log safely under concurrency with consistent snapshots.
- Collect average latency for both writers and readers.
- Demonstrate no busy-waiting**:** all blocking must use condition variables.

Assignment: Functional Requirements

Implement the following API in `rwlog.c`/`rwlog.h`.

```
#define RWLOG_MSG_MAX 64

typedef struct {
    uint64_t seq;           // global sequence assigned by monitor
    pthread_t tid;          // writing thread id
    struct timespec ts;     // timestamp (CLOCK_REALTIME)
    char msg[RWLOG_MSG_MAX];// writer-provided message
} rwlog_entry_t;

/* Lifecycle */
int     rwlog_create(size_t capacity); //the max entries in log
int     rwlog_destroy(void);

/* Reader ops */
int       rwlog_begin_read(void);

//  readers copy the most recent ≤ max_entries entries

ssize_t rwlog_snapshot(rwlog_entry_t *buf, size_t max_entries);
int     rwlog_end_read(void);

/* Writer ops */
int     rwlog_begin_write(void);
int     rwlog_append(const rwlog_entry_t *e);
int     rwlog_end_write(void);
```

Synchronization Policy

No busy waiting — all blocking must use `pthread_cond_wait` in a `while` loop.
Writer preference / no starvation**:**

- Readers can proceed concurrently only when no writer is active, and no writers are waiting**.**
- When a writer arrives**,** block new readers until the writer has run.
- On writer exit: if writers are waiting → signal one writer; otherwise broadcast to all waiting readers.

Log Behavior

Writers append to an in-memory circular log**.**
Each appended entry gets:

- Global `seq` assigned by the monitor.
- `tid` (writer thread id).
- Timestamp (`clock_gettime(CLOCK_REALTIME)`).
- A free-form message:
  "`writer<id>-msg<local_count>`"

When the buffer is full you may overwrite oldest entries (default).

Program Behavior

Parse the command line arguments:

```
--capacity <N>
--readers <R>
--writers <W>
--writer-batch <B>      # entries each writer appends per write section
--seconds <T>           # total run time
--rd-us <microseconds>  # reader sleep between operations
--wr-us <microseconds>  # writer sleep between operations
```

- Create the log (`rwlog_create`).
- Spawn `R` reader threads and `W` writer threads.
- Run until time `T` expires or the program is interrupted (`Ctrl+C`).
- Use a global `stop_flag` that the main thread sets when the timer expires or `SIGINT` is received.
- After threads finish:
  - Print metrics (see below)
- Destroy the monitor (`rwlog_destroy`).

Example:

```
./rw_log --capacity 1024 --readers 6 --writers 4 --writer-batch 2 --seconds
10 --rd-us 2000 --wr-us 3000
```

Writer Thread

Each writer thread runs in a loop until the program tells it to stop (`stop_flag` is set).
In each loop iteration it:

1. Records the start time, then calls `rwlog_begin_write()`.
   - This may block if readers are active or other writers are waiting.
   - When it returns, the thread has exclusive write access**.**
2. Measures how long it waited to get the write lock and records that time for later average calculation.
3. Appends a fixed number of log entries (the `writer_batch` count).
   - For each entry, it fills the msg field with a string like "writer3-msg17" (writer id and local counter).
   - The monitor itself will fill in the global sequence number, thread id, and timestamp.
4. Call `rwlog_end_write()` to release the write lock.
5. Sleep for `wr-us` microseconds to simulate doing other work and let other threads run.

Reader Thread

Each reader thread also loops until `stop_flag` is set.
In each loop iteration it:

1. Records the start time, then calls `rwlog_begin_read()`.
   - This may block if a writer is active or writers are waiting (give writers preference).
2. Once inside the read section, calls `rwlog_snapshot()` to copy the current contents of the log into a local buffer (e.g., up to 128 entries).
   - The snapshot must be consistent — no writes can occur while the reader is copying.
3. Checks that the sequence numbers in the snapshot continue from the last one it saw (monotonicity check).
4. Calls `rwlog_end_read()` to release the read lock.
5. Measures how long it was inside the read section and records that time for later average calculation.
6. Sleeps for `rd-us` microseconds to simulate processing the data and to allow interleaving with other threads.

Main Process

The main program is responsible for setting up**,** starting**,** and stopping the entire system and for collecting results at the end.

1. Parse command-line arguments
   - Read in the user-specified parameters (`--capacity, --readers, --writers, --writer-batch, --seconds, --rd-us, --wr-us`).
   - Use these to configure the log and how many threads to start.

2. Initialize the monitor
   - Call `rwlog_create(capacity)` to allocate and initialize the shared log and its synchronization primitives (mutex and condition variables).
3. Set up program stop mechanism
   - Create a global `stop_flag` variable and a timer.
   - Start a timer thread that will sleep for the requested run time `(--seconds)` and then set `stop_flag` to true.
   - Also install a Ctrl+C (`SIGINT`) signal handler that sets `stop_flag` and wakes up any threads blocked in the monitor so the program can shut down cleanly.
4. Launch worker threads
   - Create `W` writer threads, passing each its writer id, batch size, and sleep delay.
   - Create `R` reader threads, passing each its reader id and sleep delay.
5. Wait for run time to finish
   - The main thread does not do work itself; it simply waits until the timer expires or the user interrupts with `Ctrl+C`.
   - The workers loop and operate on the log until `stop_flag` is set.
6. Join all threads
   - After the timer thread sets `stop_flag`, the main thread waits (`pthread_join`) for all writer and reader threads to finish their loops and exit.
7. Optional: dump the log
   - Once all workers are done, take one last `rwlog_snapshot()` under a read lock and write the entries to a CSV file.
   - This gives you a permanent record for debugging or grading.
8. Compute and print metrics
   - Combine all the writer wait times and compute the average writer wait in milliseconds.
   - Combine all the reader critical-section times and compute the average reader time in milliseconds.
   - Count the total number of log entries written and compute throughput (entries per second).
   - Print these results to the console.
9. Clean up
   - Call `rwlog_destroy()` to free resources and destroy the monitor.

Important Tips

- No `pthread_rwlock_*`. Build the monitor manually with a mutex and two condition variables.
- No busy-wait loops. Use `pthread_cond_wait` inside while loops.
- Use `CLOCK_MONOTONIC` for timing measurements; `CLOCK_REALTIME` for timestamps in log entries.
- Keep critical sections small — copy indices quickly, copy data outside locks when possible.

Error Handling

Your program should check the return values of all system and library calls (e.g., `pthread_*`, `clock_gettime`, `malloc`, `rwlog_*` functions). If any call fails, print a clear error message to `stderr` and exit gracefully rather than continuing with invalid state.

- Do not let the program crash or deadlock if an error occurs.
- If the monitor cannot be created (`rwlog_create` fails), print an error and exit immediately.
- Threads that detect an unrecoverable error should exit their loop and return.
- Before exiting, release any held locks and free any allocated memory.

Grading

The program will be graded on the basic functionality, error handling and how well the implementation description was followed. Be sure to name your program **rw_log.c** (no extra characters, capitals) Note that documentation and style are worth 10% of the assignment's grade!

Submission

The source code for program should be available on Canvas along with a README/Output file that provides any documentation and sample output.