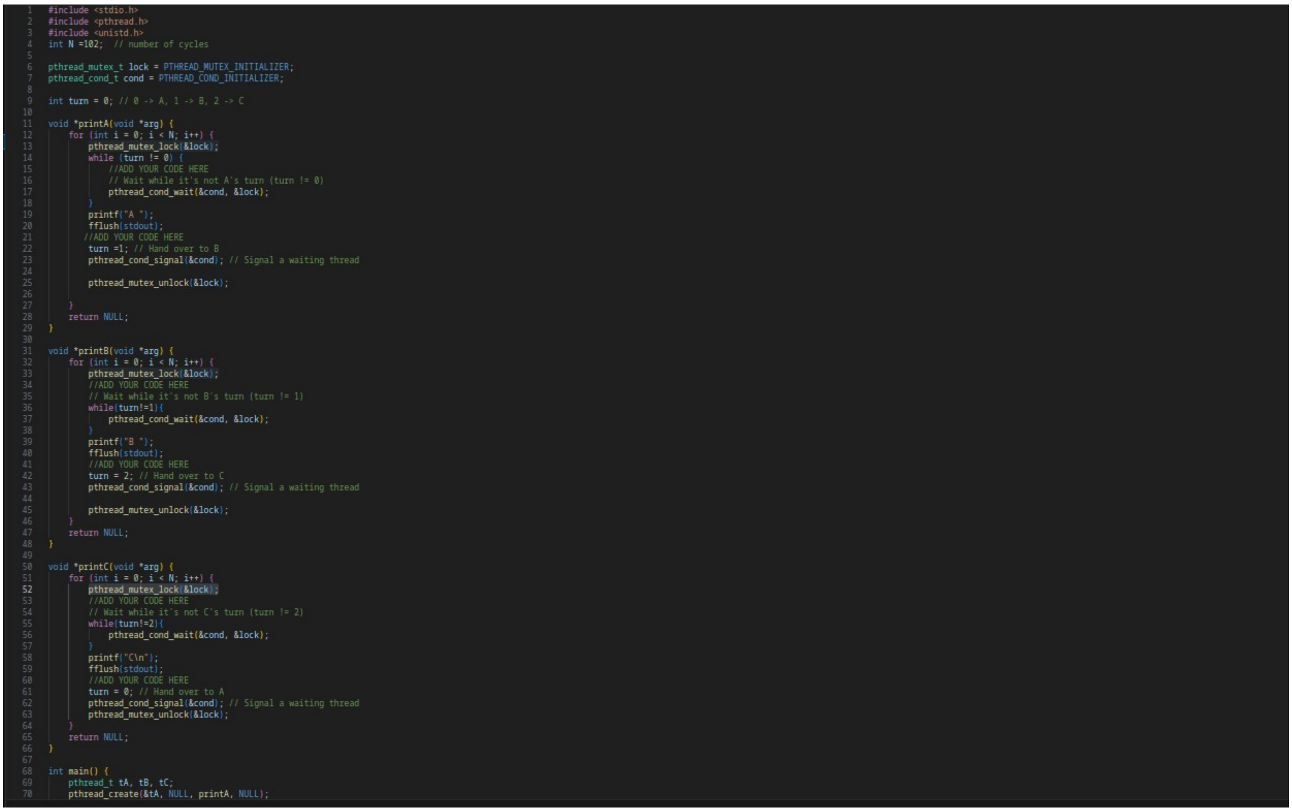
CLASS ASSIGNMENT 10

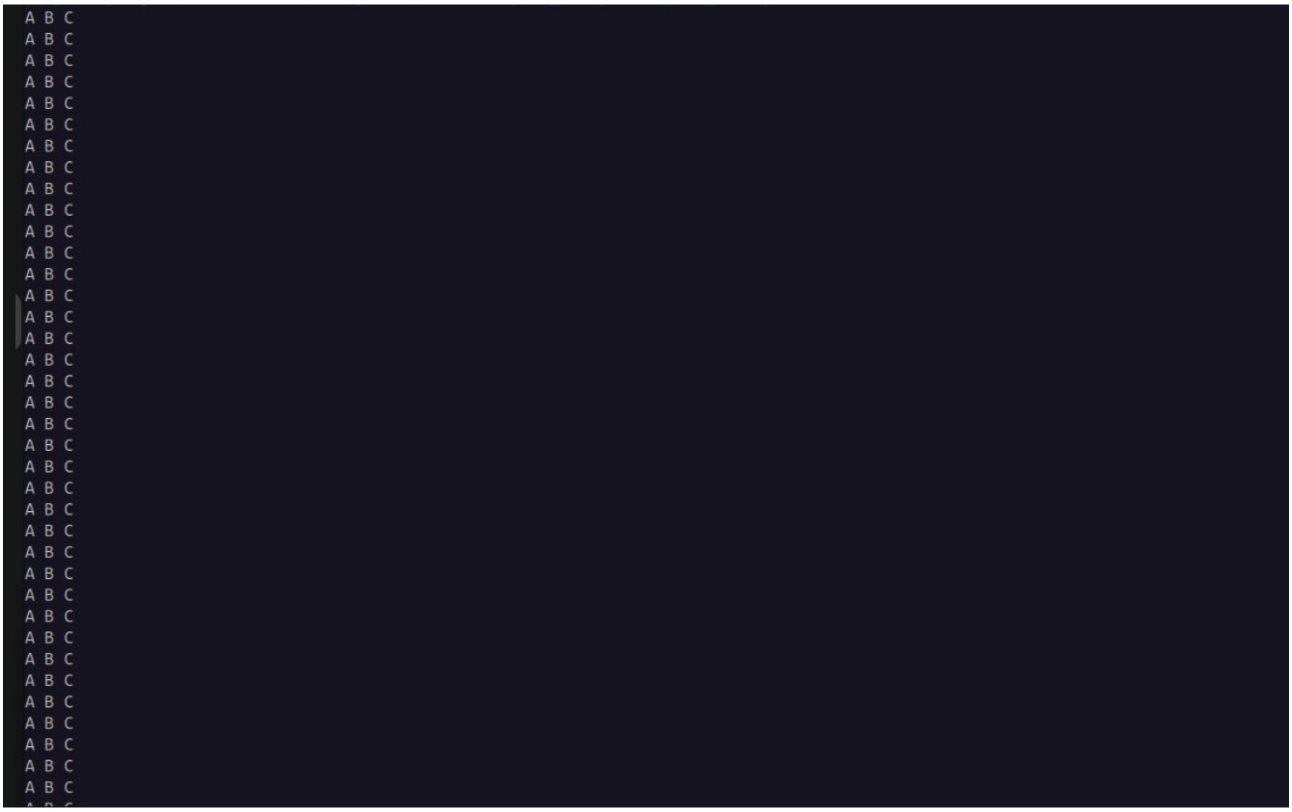
Date – 4-11-25
Admission no-12340740

Question 1)
The code used is as follows:-



```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
int N =102;  // number of cycles

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int turn = 0; // 0 -> A, 1 -> B, 2 -> C

void *printA(void *arg) {
    for (int i = 0; i < N; i++) {
        pthread_mutex_lock(&lock);
        while (turn != 0) {
            //ADD YOUR CODE HERE
            // Wait while it's not A's turn (turn != 0)
            pthread_cond_wait(&cond, &lock);
        }
        printf("A ");
        fflush(stdout);
       //ADD YOUR CODE HERE
        turn =1; // Hand over to B
        pthread_cond_signal(&cond); // Signal a waiting thread

        pthread_mutex_unlock(&lock);

    }
    return NULL;
}

void *printB(void *arg) {
    for (int i = 0; i < N; i++) {
        pthread_mutex_lock(&lock);
        //ADD YOUR CODE HERE
        // Wait while it's not B's turn (turn != 1)
        while(turn!=1){
            pthread_cond_wait(&cond, &lock);
        }
        printf("B ");
        fflush(stdout);
        //ADD YOUR CODE HERE
        turn = 2; // Hand over to C
        pthread_cond_signal(&cond); // Signal a waiting thread

        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

void *printC(void *arg) {
    for (int i = 0; i < N; i++) {
        pthread_mutex_lock(&lock);
        //ADD YOUR CODE HERE
        // Wait while it's not C's turn (turn != 2)
        while(turn!=2){
            pthread_cond_wait(&cond, &lock);
        }
        printf("C\n");
        fflush(stdout);
        //ADD YOUR CODE HERE
        turn = 0; // Hand over to A
        pthread_cond_signal(&cond); // Signal a waiting thread
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main() {
    pthread_t tA, tB, tC;
    pthread_create(&tA, NULL, printA, NULL);
```

The output is as follows:-

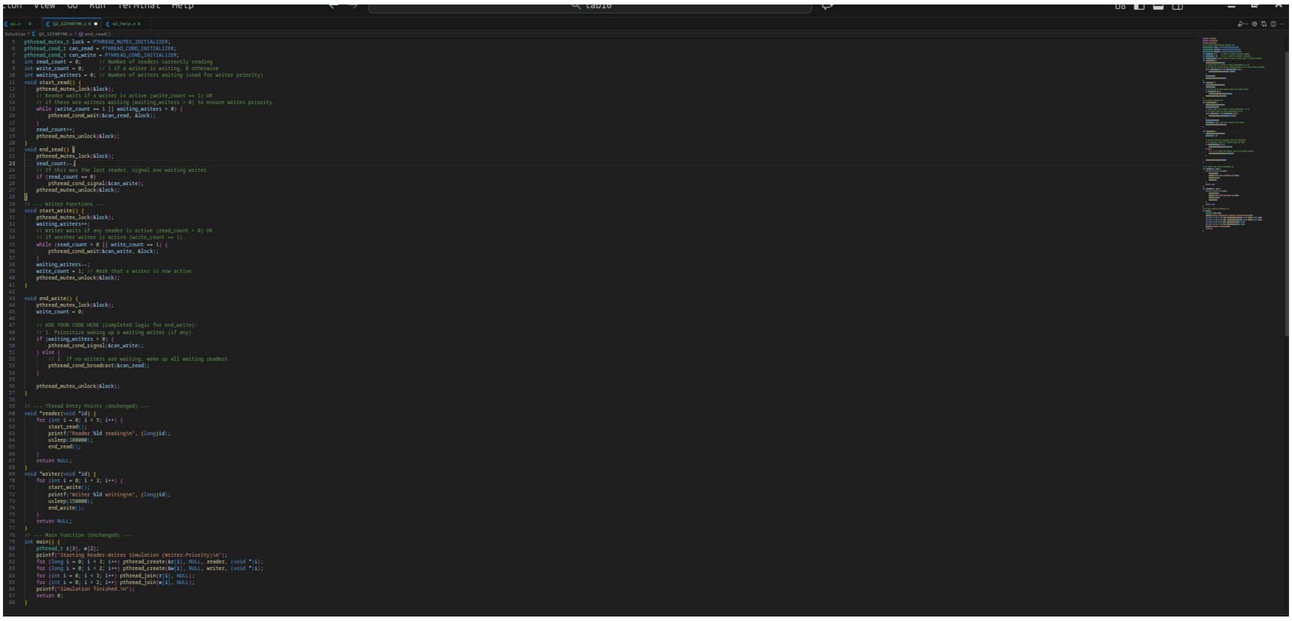


```
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
A B C
```

The same strategy fom q1_help.c was implemented only c being additional vairable thus the value 2 was given.

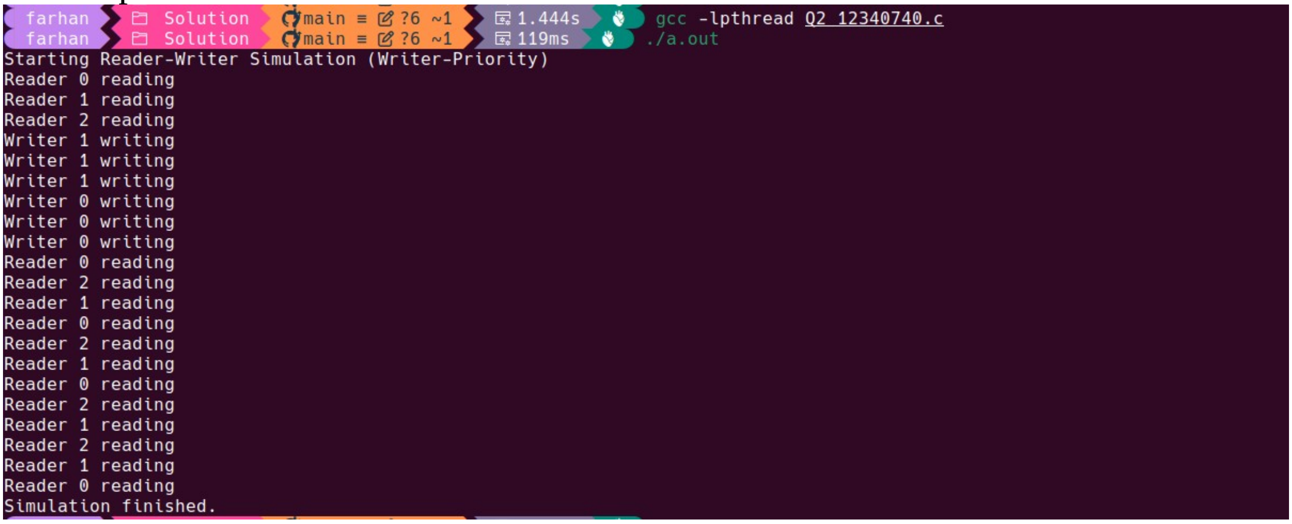
Question 2)

For the second question the code used is as follows:-



```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t can_read = PTHREAD_COND_INITIALIZER;
pthread_cond_t can_write = PTHREAD_COND_INITIALIZER;
int read_count = 0;      // Number of readers currently reading
int write_count = 0;     // 1 if a writer is writing, 0 otherwise
int waiting_writers = 0; // Number of writers waiting (used for writer priority)
void start_read() {
    pthread_mutex_lock(&lock);
    // Reader waits if a writer is active (write_count == 1) OR
    // if there are writers waiting (waiting_writers > 0) to ensure writer priority.
    while (write_count == 1 || waiting_writers > 0) {
        pthread_cond_wait(&can_read, &lock);
    }
    read_count++;
    pthread_mutex_unlock(&lock);
}
void end_read() {
    pthread_mutex_lock(&lock);
    read_count--;
    // If this was the last reader, signal one waiting writer.
    if (read_count == 0)
        pthread_cond_signal(&can_write);
    pthread_mutex_unlock(&lock);
}
// --- Writer Functions ---
void start_write() {
    pthread_mutex_lock(&lock);
    waiting_writers++;
    // Writer waits if any reader is active (read_count > 0) OR
    // if another writer is active (write_count == 1).
    while (read_count > 0 || write_count == 1) {
        pthread_cond_wait(&can_write, &lock);
    }
    waiting_writers--;
    write_count = 1; // Mark that a writer is now active
    pthread_mutex_unlock(&lock);
}

void end_write() {
    pthread_mutex_lock(&lock);
    write_count = 0;

    // ADD YOUR CODE HERE (Completed logic for end_write):
    // 1. Prioritize waking up a waiting writer (if any).
    if (waiting_writers > 0) {
        pthread_cond_signal(&can_write);
    } else {
        // 2. If no writers are waiting, wake up all waiting readers.
        pthread_cond_broadcast(&can_read);
    }

    pthread_mutex_unlock(&lock);
}

// --- Thread Entry Points (Unchanged) ---
void *reader(void *id) {
    for (int i = 0; i < 5; i++) {
        start_read();
        printf("Reader %ld reading\n", (long)id);
        usleep(100000);
        end_read();
    }
    return NULL;
}
void *writer(void *id) {
    for (int i = 0; i < 3; i++) {
        start_write();
        printf("Writer %ld writing\n", (long)id);
        usleep(150000);
        end_write();
    }
    return NULL;
}
// --- Main Function (Unchanged) ---
int main() {
    pthread_t r[3], w[2];
    printf("Starting Reader-Writer Simulation (Writer-Priority)\n");
    for (long i = 0; i < 3; i++) pthread_create(&r[i], NULL, reader, (void *)i);
    for (long i = 0; i < 2; i++) pthread_create(&w[i], NULL, writer, (void *)i);
    for (int i = 0; i < 3; i++) pthread_join(r[i], NULL);
    for (int i = 0; i < 2; i++) pthread_join(w[i], NULL);
    printf("Simulation finished.\n");
    return 0;
}
```

The output is as follows:-



```
 farhan   Solution   main ≡  ?6 ~1   1.444s   gcc -lpthread Q2_12340740.c
 farhan   Solution   main ≡  ?6 ~1   119ms   ./a.out
Starting Reader-Writer Simulation (Writer-Priority)
Reader 0 reading
Reader 1 reading
Reader 2 reading
Writer 1 writing
Writer 1 writing
Writer 1 writing
Writer 0 writing
Writer 0 writing
Writer 0 writing
Reader 0 reading
Reader 2 reading
Reader 1 reading
Reader 0 reading
Reader 2 reading
Reader 1 reading
Reader 0 reading
Reader 2 reading
Reader 1 reading
Reader 2 reading
Reader 1 reading
Reader 0 reading
Simulation finished.
```

1. `start_read()`

- **Wait Condition:** Implemented the condition that blocks readers if a writer is active or if there are writers waiting. This enforces **Writer Priority**.
    - `while (write_count == 1 || waiting_writers > 0)`

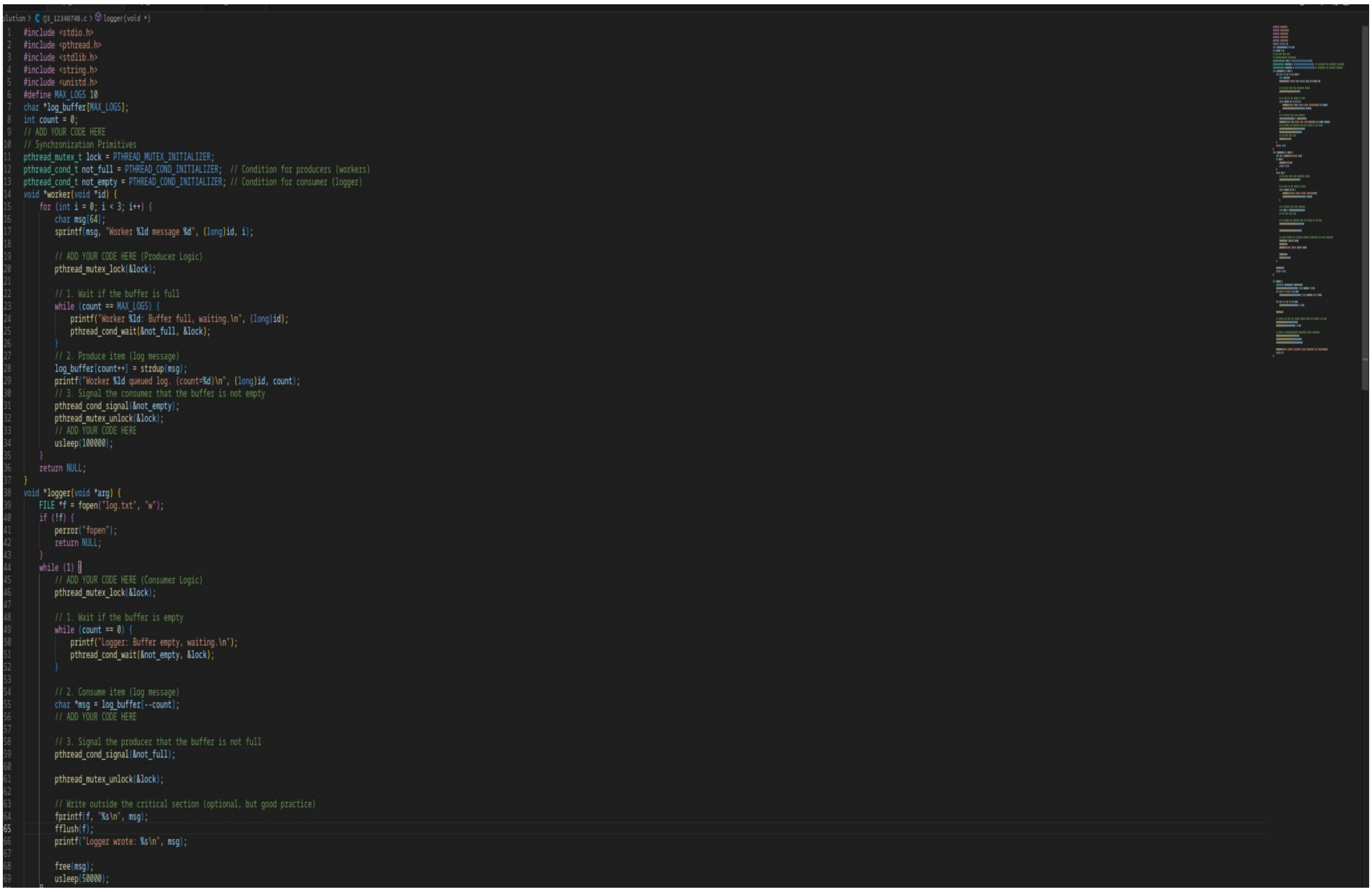
2. `start_write()`

- **Pre-wait Action:** Incremented `waiting_writers` before the `while` loop to accurately count the number of writers waiting to enter the critical section.
    - `waiting_writers++;`
- **Wait Condition:** Implemented the condition that blocks writers if any reader is active or if another writer is active.
    - `while (read_count > 0 || write_count == 1)`
- **Post-wait Action:** Decremented `waiting_writers` and set `write_count` to 1 upon successfully entering the critical section.
    - `waiting_writers--;`
    - `write_count = 1;`

3. `end_write()`

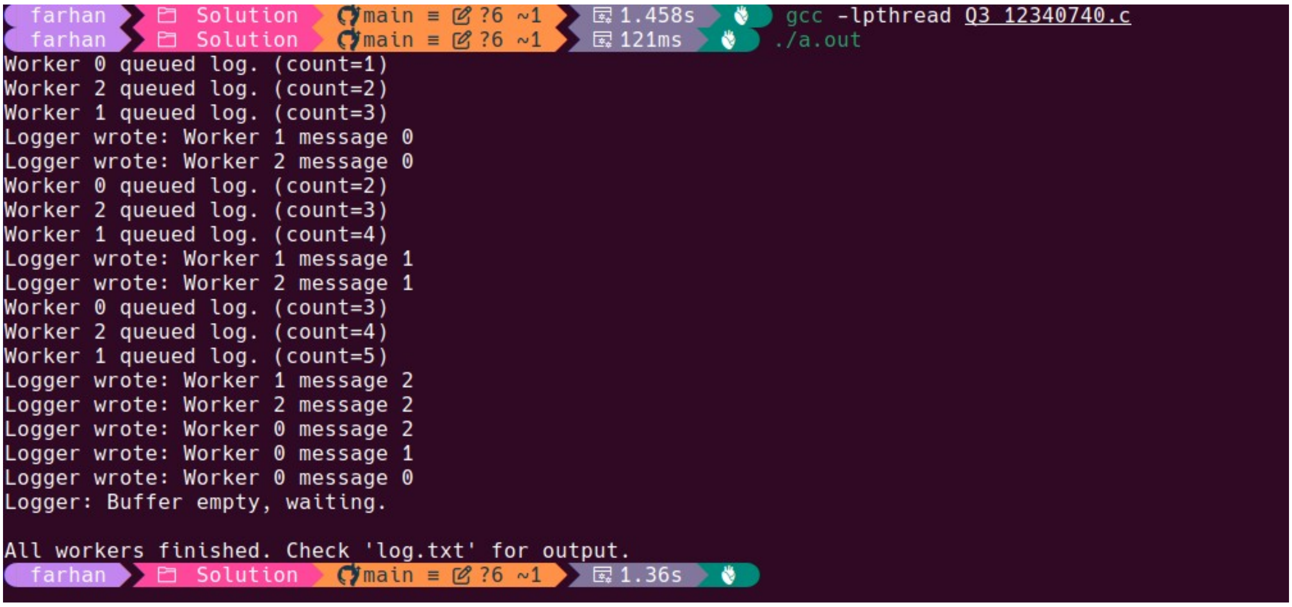
- **Post-write Action:** Implemented the logic to decide whether to wake up a waiting writer or waiting readers.
    - Set `write_count = 0;` to signal the writer is finished.
    - **Writer Priority Signaling:** Used an `if/else` block:
        - `if (waiting_writers > 0)`: Signal **one** waiting writer using `pthread_cond_signal(&can_write)`.
        - `else`: Signal **all** waiting readers using `pthread_cond_broadcast(&can_read)`.

Question 3)
The code used is as follows:-



```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define MAX_LOGS 10
char *log_buffer[MAX_LOGS];
int count = 0;
// ADD YOUR CODE HERE
// Synchronization Primitives
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;  // Condition for producers (workers)
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER; // Condition for consumer (logger)
void *worker(void *id) {
    for (int i = 0; i < 3; i++) {
        char msg[64];
        sprintf(msg, "Worker %ld message %d", (long)id, i);

        // ADD YOUR CODE HERE (Producer Logic)
        pthread_mutex_lock(&lock);

        // 1. Wait if the buffer is full
        while (count == MAX_LOGS) {
            printf("Worker %ld: Buffer full, waiting.\n", (long)id);
            pthread_cond_wait(&not_full, &lock);
        }
        // 2. Produce item (log message)
        log_buffer[count++] = strdup(msg);
        printf("Worker %ld queued log. (count=%d)\n", (long)id, count);
        // 3. Signal the consumer that the buffer is not empty
        pthread_cond_signal(&not_empty);
        pthread_mutex_unlock(&lock);
        // ADD YOUR CODE HERE
        usleep(100000);
    }
    return NULL;
}
void *logger(void *arg) {
    FILE *f = fopen("log.txt", "w");
    if (!f) {
        perror("fopen");
        return NULL;
    }
    while (1) {
        // ADD YOUR CODE HERE (Consumer Logic)
        pthread_mutex_lock(&lock);

        // 1. Wait if the buffer is empty
        while (count == 0) {
            printf("Logger: Buffer empty, waiting.\n");
            pthread_cond_wait(&not_empty, &lock);
        }

        // 2. Consume item (log message)
        char *msg = log_buffer[--count];
        // ADD YOUR CODE HERE

        // 3. Signal the producer that the buffer is not full
        pthread_cond_signal(&not_full);

        pthread_mutex_unlock(&lock);

        // Write outside the critical section (optional, but good practice)
        fprintf(f, "%s\n", msg);
        fflush(f);
        printf("Logger wrote: %s\n", msg);

        free(msg);
        usleep(50000);
```

The output is as follows:-



```
 farhan   Solution  main ≡ ?6 ~1   1.458s   gcc -lpthread Q3_12340740.c
 farhan   Solution  main ≡ ?6 ~1   121ms   ./a.out
Worker 0 queued log. (count=1)
Worker 2 queued log. (count=2)
Worker 1 queued log. (count=3)
Logger wrote: Worker 1 message 0
Logger wrote: Worker 2 message 0
Worker 0 queued log. (count=2)
Worker 2 queued log. (count=3)
Worker 1 queued log. (count=4)
Logger wrote: Worker 1 message 1
Logger wrote: Worker 2 message 1
Worker 0 queued log. (count=3)
Worker 2 queued log. (count=4)
Worker 1 queued log. (count=5)
Logger wrote: Worker 1 message 2
Logger wrote: Worker 2 message 2
Logger wrote: Worker 0 message 2
Logger wrote: Worker 0 message 1
Logger wrote: Worker 0 message 0
Logger: Buffer empty, waiting.

All workers finished. Check 'log.txt' for output.
 farhan   Solution  main ≡ ?6 ~1   1.36s 
```

1. Global Declarations

- **Added Synchronization Primitives:** Defined and initialized the mutex and condition variables needed to control access and signal status:
  - `pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;`
  - `pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;` (For when the buffer has space)
  - `pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;` (For when the buffer has items)

---

2. `worker` Function (Producer)

- **Mutual Exclusion:** Added `pthread_mutex_lock(&lock)` before accessing shared variables and `pthread_mutex_unlock(&lock)` after.
- **Buffer Full Check (Wait):** Implemented the logic to wait if the buffer is full:

  C

  ```
  while (count == MAX_LOGS) {
      // ... (optional print statement)
      pthread_cond_wait(&not_full, &lock);
  }
  ```

- **Signal Consumer:** Added a signal after successfully adding an item to the buffer:
  - `pthread_cond_signal(&not_empty);`

---

3. `logger` Function (Consumer)

- **Mutual Exclusion:** Added `pthread_mutex_lock(&lock)` before accessing shared variables and `pthread_mutex_unlock(&lock)` after.
- **Buffer Empty Check (Wait):** Implemented the logic to wait if the buffer is empty:

  C

  ```
  while (count == 0) {
      // ... (optional print statement)
      pthread_cond_wait(&not_empty, &lock);
  }
  ```

- **Signal Producer:** Added a signal after successfully removing an item from the buffer:
  - `pthread_cond_signal(&not_full);`
- **Moved I/O:** The file I/O operations (`fprintf`, `fflush`, `printf`, and `free`) were kept **outside** the critical section (outside the mutex lock) for better performance, as they do not modify the shared state.

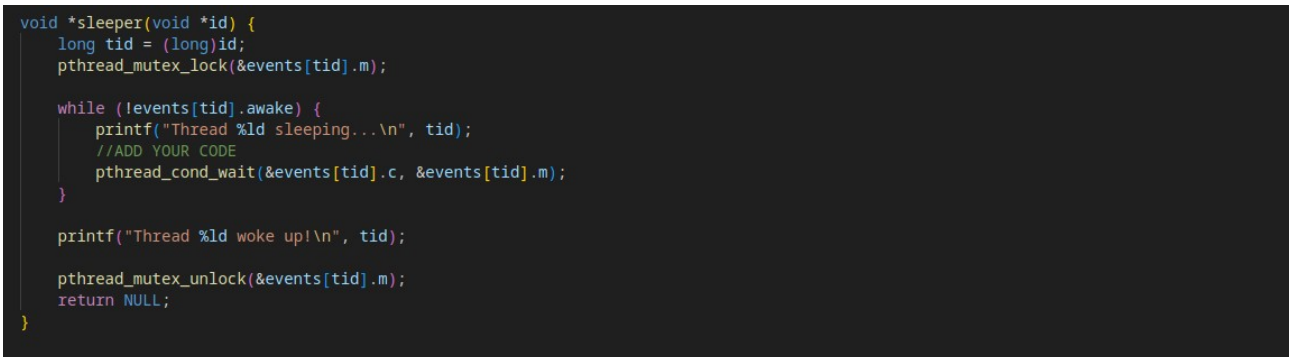
---

4. `main` Function

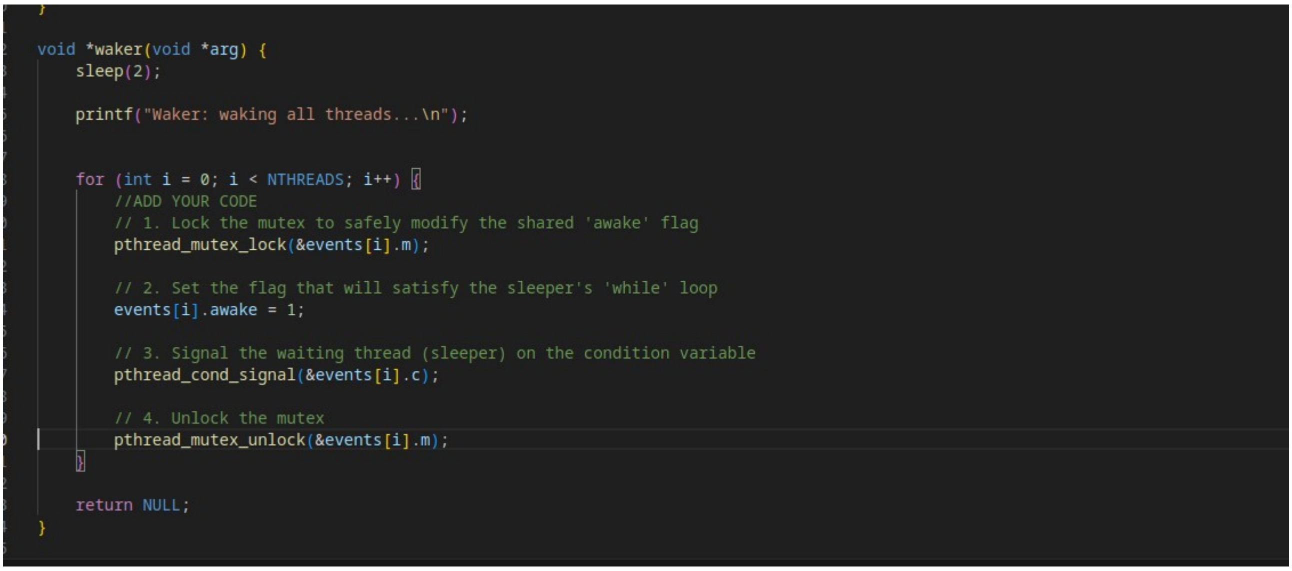
- **Clean Up:** Added destruction calls for the synchronization primitives at the end of `main` (good practice):
  - `pthread_mutex_destroy(&lock);`
  - `pthread_cond_destroy(&not_full);`
  - `pthread_cond_destroy(&not_empty);`

Question 4)

The code added is as follows:-

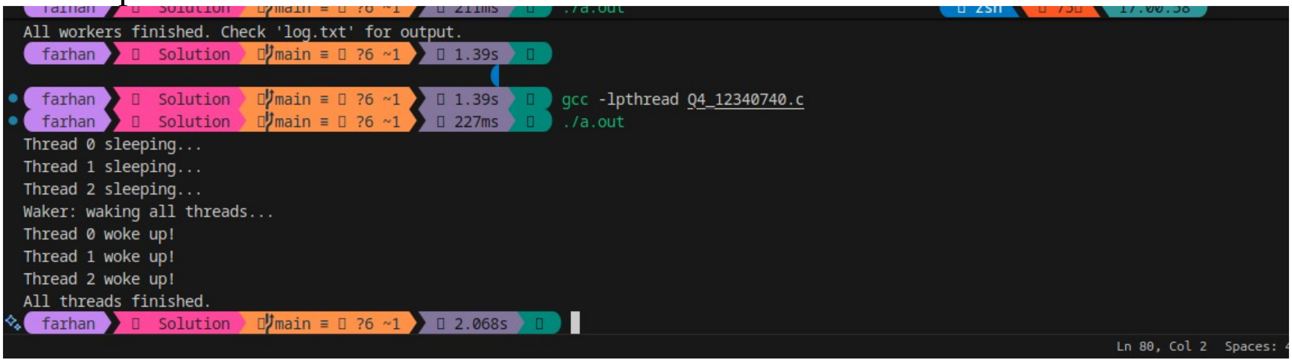


```
void *sleeper(void *id) {
    long tid = (long)id;
    pthread_mutex_lock(&events[tid].m);

    while (!events[tid].awake) {
        printf("Thread %ld sleeping...\n", tid);
        //ADD YOUR CODE
        pthread_cond_wait(&events[tid].c, &events[tid].m);
    }

    printf("Thread %ld woke up!\n", tid);

    pthread_mutex_unlock(&events[tid].m);
    return NULL;
}
```



```
}

void *waker(void *arg) {
    sleep(2);

    printf("Waker: waking all threads...\n");


    for (int i = 0; i < NTHREADS; i++) {
        //ADD YOUR CODE
        // 1. Lock the mutex to safely modify the shared 'awake' flag
        pthread_mutex_lock(&events[i].m);

        // 2. Set the flag that will satisfy the sleeper's 'while' loop
        events[i].awake = 1;

        // 3. Signal the waiting thread (sleeper) on the condition variable
        pthread_cond_signal(&events[i].c);

        // 4. Unlock the mutex
        pthread_mutex_unlock(&events[i].m);
    }

    return NULL;
}
```

The output is as follows:-



```
All workers finished. Check 'log.txt' for output.
 farhan   Solution  main ≡  ?6 ~1   1.39s 

 farhan   Solution  main ≡  ?6 ~1   1.39s   gcc -lpthread Q4_12340740.c
 farhan   Solution  main ≡  ?6 ~1   227ms   ./a.out
Thread 0 sleeping...
Thread 1 sleeping...
Thread 2 sleeping...
Waker: waking all threads...
Thread 0 woke up!
Thread 1 woke up!
Thread 2 woke up!
All threads finished.
 farhan   Solution  main ≡  ?6 ~1   2.068s 
                                                              Ln 80, Col 2   Spaces: 4
```

This program is designed to demonstrate how a dedicated **waker** thread can wake up multiple **sleeper** threads using a custom event structure built with **mutexes** and **condition variables**. Here is the completed code with the synchronization logic filled in:

C

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h> // Needed for pthread_mutex_destroy/pthread_cond_destroy
(good practice)

#define NTHREADS 3
```

```
struct thread_event {
    pthread_mutex_t m;
    pthread_cond_t c;
    int awake;
};

struct thread_event events[NTHREADS];

void *sleeper(void *id) {
    long tid = (long)id;
    pthread_mutex_lock(&events[tid].m);

    while (!events[tid].awake) {
        printf("Thread %ld sleeping...\n", tid);
        //ADD YOUR CODE
        // Atomically releases the mutex and waits on the condition variable.
        // When signaled, it re-acquires the mutex and checks the 'while'
condition.
        pthread_cond_wait(&events[tid].c, &events[tid].m);
    }

    printf("Thread %ld woke up!\n", tid);

    pthread_mutex_unlock(&events[tid].m);
    return NULL;
}

void *waker(void *arg) {
    sleep(2);

    printf("Waker: waking all threads...\n");


    for (int i = 0; i < NTHREADS; i++) {
        //ADD YOUR CODE
        // 1. Lock the mutex to safely modify the shared 'awake' flag
        pthread_mutex_lock(&events[i].m);

        // 2. Set the flag that will satisfy the sleeper's 'while' loop
        events[i].awake = 1;

        // 3. Signal the waiting thread (sleeper) on the condition variable
        pthread_cond_signal(&events[i].c);

        // 4. Unlock the mutex
        pthread_mutex_unlock(&events[i].m);
    }

    return NULL;
}

int main() {
    pthread_t t[NTHREADS], w;

    for (int i = 0; i < NTHREADS; i++) {
        events[i].awake = 0;
        pthread_mutex_init(&events[i].m, NULL);
        pthread_cond_init(&events[i].c, NULL);
        pthread_create(&t[i], NULL, sleeper, (void *)(long)i);
    }

    pthread_create(&w, NULL, waker, NULL);
    pthread_join(w, NULL);
    for (int i = 0; i < NTHREADS; i++)
        pthread_join(t[i], NULL);

    // Clean up resources (good practice)
    for (int i = 0; i < NTHREADS; i++) {
        pthread_mutex_destroy(&events[i].m);
        pthread_cond_destroy(&events[i].c);
    }

    printf("All threads finished.\n");
    return 0;
}
```

The solution uses a separate **event structure** (`thread_event`) for each thread, containing its own mutex, condition variable, and state flag (`awake`).

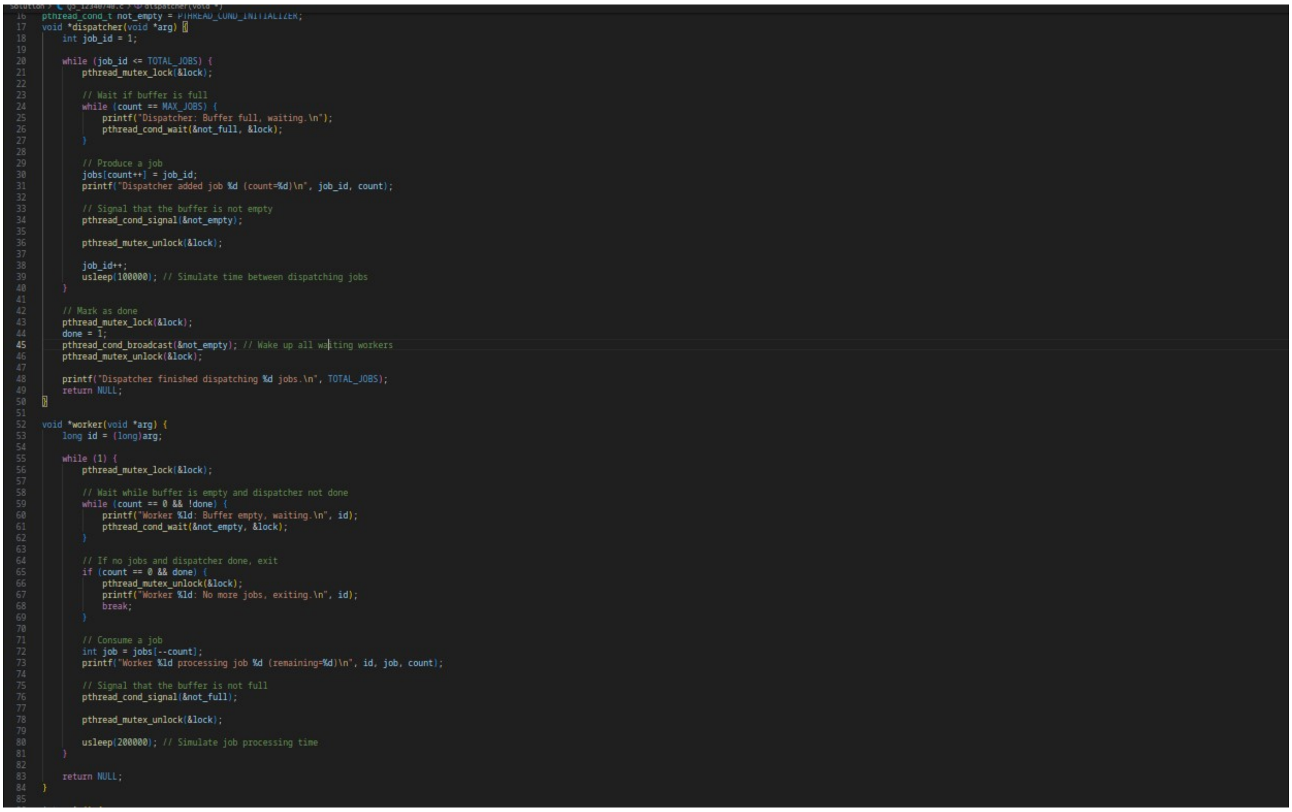
`sleeper` function (The Waiting Thread)

1. **Wait Loop:** It enters a `while (!events[tid].awake)` loop. This is the **standard pattern** for condition variables to guard against spurious wakeups (waking up without a signal).
2. **Waiting:** `pthread_cond_wait(&events[tid].c, &events[tid].m)` is used inside the loop. This function atomically:
    - **Releases** the mutex.
    - **Blocks** the thread, waiting for a signal.
    - When signaled, it **re-acquires** the mutex and returns, allowing the `while` condition to be re-checked.

`waker` function (The Signaling Thread)

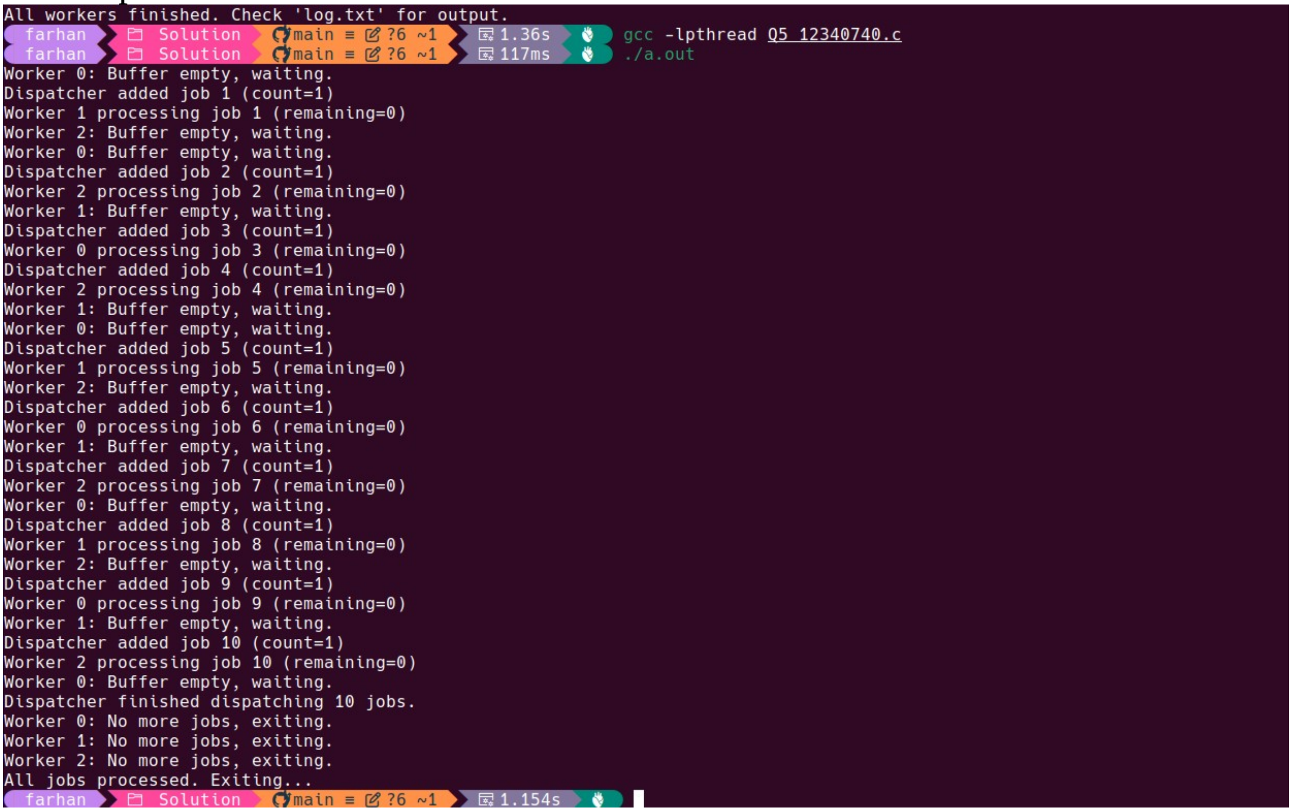
1. **Iteration:** It loops through all `NTHREADS`.
2. **State Change:** For each thread's event structure, it performs the following:
    - **Locks** the mutex (`pthread_mutex_lock`).
    - **Changes the state** (`events[i].awake = 1`) to satisfy the sleeper's `while` loop condition.
    - **Signals** the waiting thread (`pthread_cond_signal(&events[i].c)`).
    - **Unlocks** the mutex (`pthread_mutex_unlock`).

Question 5
The code used is as follows:-



```
Solution > C Q5_12340740.c > dispatcher(void *)
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;
void *dispatcher(void *arg) {
    int job_id = 1;

    while (job_id <= TOTAL_JOBS) {
        pthread_mutex_lock(&lock);

        // Wait if buffer is full
        while (count == MAX_JOBS) {
            printf("Dispatcher: Buffer full, waiting.\n");
            pthread_cond_wait(&not_full, &lock);
        }

        // Produce a job
        jobs[count++] = job_id;
        printf("Dispatcher added job %d (count=%d)\n", job_id, count);

        // Signal that the buffer is not empty
        pthread_cond_signal(&not_empty);

        pthread_mutex_unlock(&lock);

        job_id++;
        usleep(100000); // Simulate time between dispatching jobs
    }

    // Mark as done
    pthread_mutex_lock(&lock);
    done = 1;
    pthread_cond_broadcast(&not_empty); // Wake up all waiting workers
    pthread_mutex_unlock(&lock);

    printf("Dispatcher finished dispatching %d jobs.\n", TOTAL_JOBS);
    return NULL;
}

void *worker(void *arg) {
    long id = (long)arg;

    while (1) {
        pthread_mutex_lock(&lock);

        // Wait while buffer is empty and dispatcher not done
        while (count == 0 && !done) {
            printf("Worker %ld: Buffer empty, waiting.\n", id);
            pthread_cond_wait(&not_empty, &lock);
        }

        // If no jobs and dispatcher done, exit
        if (count == 0 && done) {
            pthread_mutex_unlock(&lock);
            printf("Worker %ld: No more jobs, exiting.\n", id);
            break;
        }

        // Consume a job
        int job = jobs[--count];
        printf("Worker %ld processing job %d (remaining=%d)\n", id, job, count);

        // Signal that the buffer is not full
        pthread_cond_signal(&not_full);

        pthread_mutex_unlock(&lock);

        usleep(200000); // Simulate job processing time
    }

    return NULL;
}
```

The output is as follows:-



```
All workers finished. Check 'log.txt' for output.
 farhan  Solution  main ≡ ?6 ~1  1.36s  gcc -lpthread Q5_12340740.c
 farhan  Solution  main ≡ ?6 ~1  117ms  ./a.out
Worker 0: Buffer empty, waiting.
Dispatcher added job 1 (count=1)
Worker 1 processing job 1 (remaining=0)
Worker 2: Buffer empty, waiting.
Worker 0: Buffer empty, waiting.
Dispatcher added job 2 (count=1)
Worker 2 processing job 2 (remaining=0)
Worker 1: Buffer empty, waiting.
Dispatcher added job 3 (count=1)
Worker 0 processing job 3 (remaining=0)
Dispatcher added job 4 (count=1)
Worker 2 processing job 4 (remaining=0)
Worker 1: Buffer empty, waiting.
Worker 0: Buffer empty, waiting.
Dispatcher added job 5 (count=1)
Worker 1 processing job 5 (remaining=0)
Worker 2: Buffer empty, waiting.
Dispatcher added job 6 (count=1)
Worker 0 processing job 6 (remaining=0)
Worker 1: Buffer empty, waiting.
Dispatcher added job 7 (count=1)
Worker 2 processing job 7 (remaining=0)
Worker 0: Buffer empty, waiting.
Dispatcher added job 8 (count=1)
Worker 1 processing job 8 (remaining=0)
Worker 2: Buffer empty, waiting.
Dispatcher added job 9 (count=1)
Worker 0 processing job 9 (remaining=0)
Worker 1: Buffer empty, waiting.
Dispatcher added job 10 (count=1)
Worker 2 processing job 10 (remaining=0)
Worker 0: Buffer empty, waiting.
Dispatcher finished dispatching 10 jobs.
Worker 0: No more jobs, exiting.
Worker 1: No more jobs, exiting.
Worker 2: No more jobs, exiting.
All jobs processed. Exiting...
 farhan  Solution  main ≡ ?6 ~1  1.154s 
```