CSL301

12340220

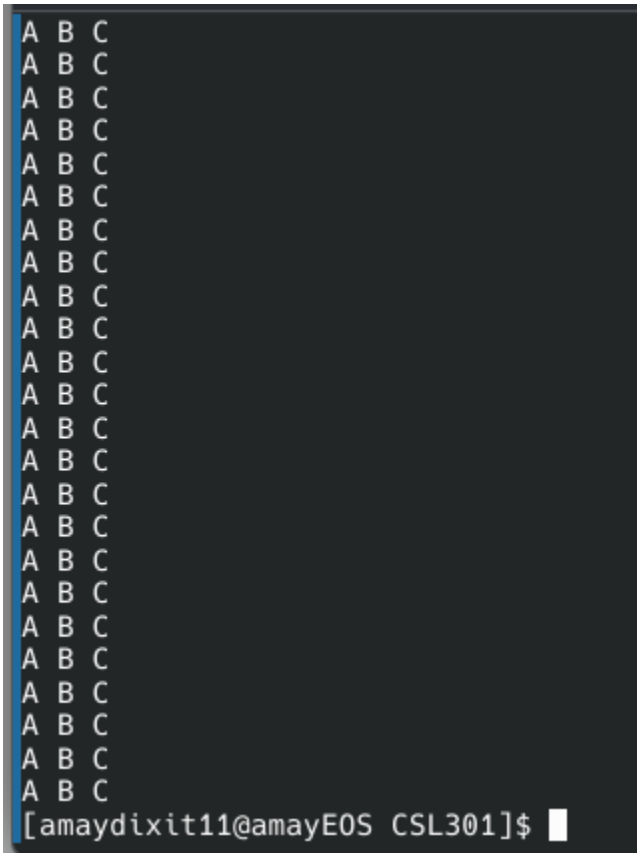
Assignment: CA10

Question 1. Identify & implement ordered thread executing

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#define 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)
            pthread_cond_wait(&cond, &lock);

        printf("A ");
        fflush(stdout);

        turn = 1;
        pthread_cond_broadcast(&cond);

        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

void *printB(void *arg) {
    for (int i = 0; i < N; i++) {
        pthread_mutex_lock(&lock);

        while (turn != 1)
            pthread_cond_wait(&cond, &lock);

        printf("B ");
        fflush(stdout);
```

```
        turn = 2;
        pthread_cond_broadcast(&cond);

        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

void *printC(void *arg) {
    for (int i = 0; i < N; i++) {
        pthread_mutex_lock(&lock);

        while (turn != 2)
            pthread_cond_wait(&cond, &lock);

        printf("C\n");
        fflush(stdout);

        turn = 0;
        pthread_cond_broadcast(&cond);

        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main() {
    pthread_t tA, tB, tC;

    pthread_create(&tA, NULL, printA, NULL);
    pthread_create(&tB, NULL, printB, NULL);
    pthread_create(&tC, NULL, printC, NULL);

    pthread_join(tA, NULL);
    pthread_join(tB, NULL);
    pthread_join(tC, NULL);
    return 0;
}
```

Output:



Explanation:

This program uses three threads to print characters in the strict order A → B → C repeatedly. A mutex is used to ensure that only one thread accesses the shared variable `turn` at a time, while a condition variable is used to make threads wait until it becomes their turn. Each thread checks the `turn` value inside a while loop and prints only when it matches its assigned number. After printing, the thread updates the `turn` for the next thread and calls `pthread_cond_broadcast()` to wake all waiting threads. This coordination ensures the correct sequence is maintained throughout all cycles.

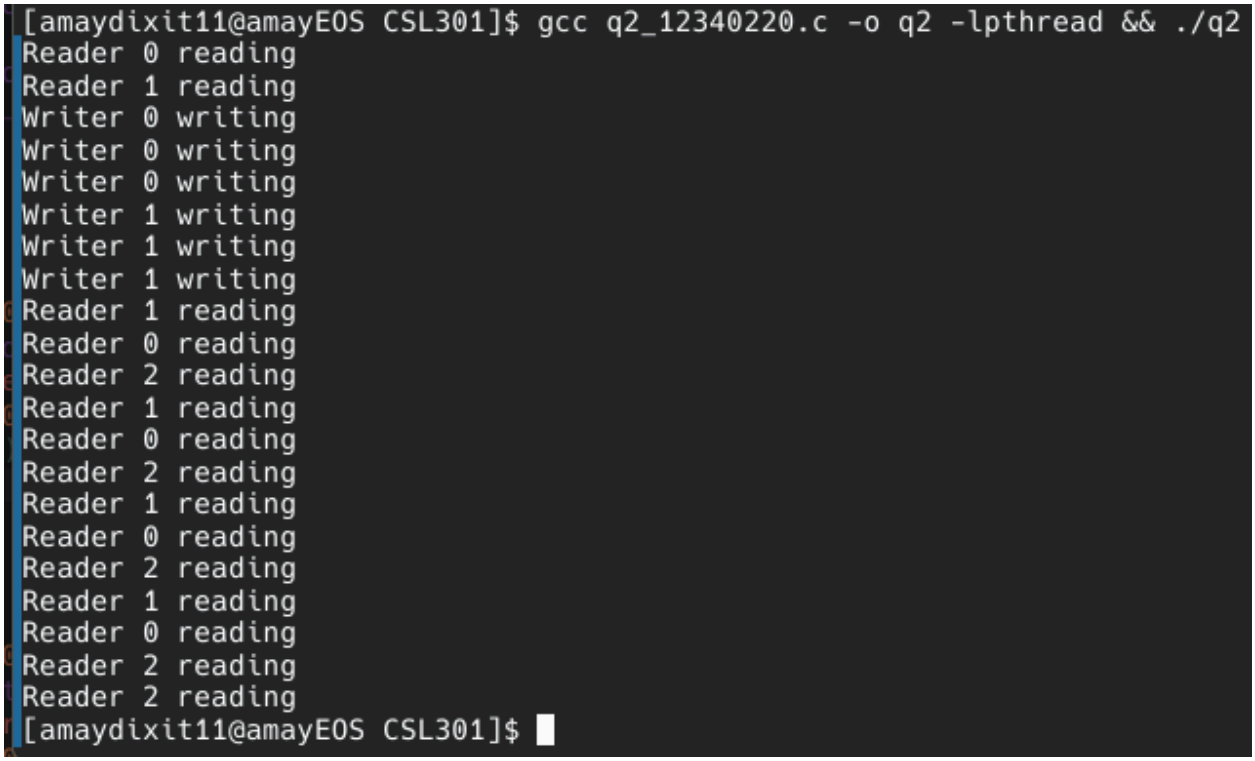
Question 2. Identity Starvation

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

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;
int write_count = 0;
int waiting_writers = 0;

void start_read() {
```

```
    pthread_mutex_lock(&lock);
    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 (read_count == 0)
        pthread_cond_signal(&can_write);
    pthread_mutex_unlock(&lock);
}

void start_write() {
    pthread_mutex_lock(&lock);
    waiting_writers++;
    while (read_count > 0 || write_count == 1) {
        pthread_cond_wait(&can_write, &lock);
    }
    waiting_writers--;
    write_count = 1;
    pthread_mutex_unlock(&lock);
}

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

    if (waiting_writers > 0)
        pthread_cond_signal(&can_write);  // give priority to writers
    else
        pthread_cond_broadcast(&can_read); // allow all readers if no writer waiting
    pthread_mutex_unlock(&lock);
}



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;
}

int main() {
    pthread_t r[3], w[2];
    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);
    return 0;
}
```

Output:



Explanation:

This implementation prevents writer starvation by giving priority to writers whenever any are waiting. Readers are allowed to read only when no writer is writing and no writer is queued. Writers wait until all current readers finish, ensuring exclusive access. When writing ends, if more writers are waiting, one writer is signaled next; otherwise, all

waiting readers are allowed to proceed. This fairness mechanism ensures that continuous incoming readers do not block writers indefinitely.

Question 3. Identify &fix missing synchronization in thread-safe logger

```
#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;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t can_log = PTHREAD_COND_INITIALIZER;
pthread_cond_t can_add = PTHREAD_COND_INITIALIZER;

void *worker(void *id) {
    for (int i = 0; i < 3; i++) {
        char msg[64];
        sprintf(msg, "Worker %ld message %d", (long)id, i);

        pthread_mutex_lock(&lock);
        while (count == MAX_LOGS)  // buffer full
            pthread_cond_wait(&can_add, &lock);

        log_buffer[count++] = strdup(msg);
        printf("Worker %ld queued log. (count=%d)\n", (long)id, count);

        pthread_cond_signal(&can_log);  // notify logger
        pthread_mutex_unlock(&lock);

        usleep(100000);
    }
    return NULL;
}

void *logger(void *arg) {
    FILE *f = fopen("log.txt", "w");
    if (!f) {
        perror("fopen");
        return NULL;
```

```
        }

        while (1) {
            pthread_mutex_lock(&lock);
            while (count == 0)  // nothing to log
                pthread_cond_wait(&can_log, &lock);

            char *msg = log_buffer[--count];
            pthread_cond_signal(&can_add);  // space available
            pthread_mutex_unlock(&lock);

            fprintf(f, "%s\n", msg);
            fflush(f);
            printf("Logger wrote: %s\n", msg);

            free(msg);
            usleep(50000);
        }

        fclose(f);
        return NULL;
    }


    int main() {
        pthread_t log_thread, workers[3];
        pthread_create(&log_thread, NULL, logger, NULL);
        for (long i = 0; i < 3; i++)
            pthread_create(&workers[i], NULL, worker, (void *)i);

        for (int i = 0; i < 3; i++)
            pthread_join(workers[i], NULL);

        sleep(1);

        pthread_cancel(log_thread);
        pthread_join(log_thread, NULL);

        printf("\nAll workers finished. Check 'log.txt' for output.\n");
        return 0;
    }
```

Output:

```
[amaydixit11@amayEOS CSL301]$ gcc q3_12340220.c -o q3 -lpthread && ./q3
Worker 0 queued log. (count=1)
Worker 1 queued log. (count=2)
Worker 2 queued log. (count=3)
Logger wrote: Worker 2 message 0
Logger wrote: Worker 1 message 0
Worker 0 queued log. (count=2)
Worker 1 queued log. (count=3)
Worker 2 queued log. (count=4)
Logger wrote: Worker 2 message 1
Logger wrote: Worker 1 message 1
Worker 0 queued log. (count=3)
Worker 1 queued log. (count=4)
Worker 2 queued log. (count=5)
Logger wrote: Worker 2 message 2
Logger wrote: Worker 1 message 2
Logger wrote: Worker 0 message 2
Logger wrote: Worker 0 message 1
Logger wrote: Worker 0 message 0

All workers finished. Check 'log.txt' for output.
[amaydixit11@amayEOS CSL301]$
```

Explanation:

This solution adds proper synchronization between worker and logger threads using a mutex and condition variables. Workers wait if the buffer is full, preventing overflow, and signal the logger whenever a new message is available. The logger waits whenever the buffer is empty, eliminating busy-waiting and CPU wastage. After consuming a message, the logger signals workers that space is free again. This ensures that all log messages are processed without loss and the CPU is used efficiently.

Question 4. Identify and implement Sleep/Wakeup Mechanism

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#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);
       pthread_cond_wait(&events[tid].c, &events[tid].m);
```

```c
    }

    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++) {
        pthread_mutex_lock(&events[i].m);
        events[i].awake = 1;
        pthread_cond_signal(&events[i].c);  // wake each thread
        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);

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

Output:

```
┌──(~/Desktop/CSL301)
└─(17:05:24)──> gcc q4_12340220.c -o q4 -lpthread && ./q4

Thread 0 sleeping...
Thread 2 sleeping...
Thread 1 sleeping...
Waker: waking all threads...
Thread 1 woke up!
Thread 0 woke up!
Thread 2 woke up!
All threads finished.
```

Explanation:

Each sleeper thread waits on its own condition variable inside a `while` loop until its `awake` flag becomes true. This makes the thread sleep without consuming CPU. The waker thread sets each thread's `awake` flag to 1 and signals its condition variable, which wakes the thread up correctly. Using mutexes ensures shared data is updated safely, and the condition wait prevents busy-waiting, achieving correct sleep–wake synchronization.

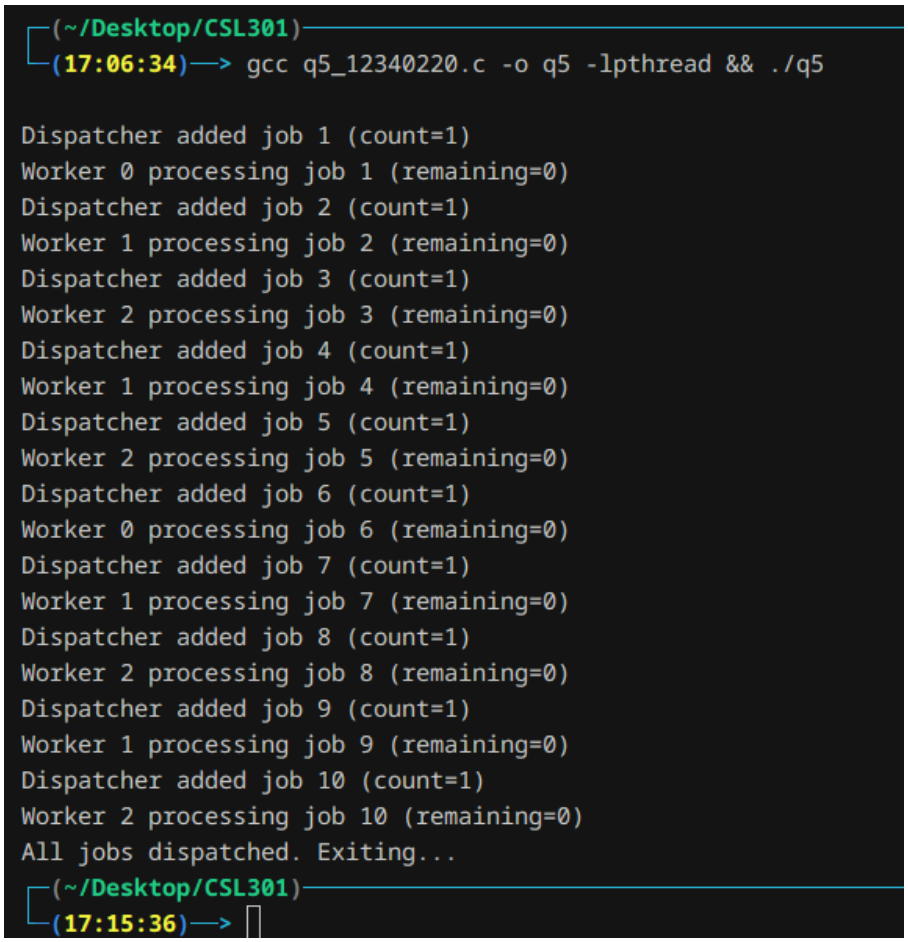
Question 5. Dining Philosophers

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#define MAX_JOBS 5
#define NUM_WORKERS 3

int jobs[MAX_JOBS];
int count = 0;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;

void *dispatcher(void *arg) {
    int job_id = 1;
    while (job_id <= 10) {
        pthread_mutex_lock(&lock);

        // Wait if job queue is full
        while (count == MAX_JOBS)
```

```
            pthread_cond_wait(&not_full, &lock);

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

        // Signal workers that a job is available
        pthread_cond_signal(&not_empty);
        pthread_mutex_unlock(&lock);

        job_id++;
        usleep(100000);
    }
    return NULL;
}

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

    while (1) {
        pthread_mutex_lock(&lock);

        // Wait if no job available
        while (count == 0)
            pthread_cond_wait(&not_empty, &lock);

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

        // Signal dispatcher that space is now free
        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&lock);

        usleep(200000);
    }
    return NULL;
}

int main() {
    pthread_t disp, workers[NUM_WORKERS];

    pthread_create(&disp, NULL, dispatcher, NULL);
    for (long i = 0; i < NUM_WORKERS; i++)
        pthread_create(&workers[i], NULL, worker, (void *)i);

    pthread_join(disp, NULL);
    sleep(2);
```

```
    printf("All jobs dispatched. Exiting...\n");
    return 0;
}
```

Output:



Explanation:

Each sleeper thread waits on its own condition variable inside a `while` loop until its `awake` flag becomes true. This makes the thread sleep without consuming CPU. The waker thread sets each thread's `awake` flag to 1 and signals its condition variable, which wakes the thread up correctly. Using mutexes ensures shared data is updated safely, and the condition wait prevents busy-waiting, achieving correct sleep–wake synchronization.