Wissam Razouki

CSE 4300

Programming Assignment 2

**Part1**

**Source code (with inline comments):**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define NUM\_THREADS 5 // total number of threads

int CurrentID = 1; // shared variable between threads

pthread\_mutex\_t mutex\_lock; // mutex lock

int myturn\_counter[NUM\_THREADS]; // counter for "my turn" statements

int notmyturn\_counter[NUM\_THREADS]; // counter for "not my turn" statements

void \*increment\_id(void \**threadid*)

{

    long tid;

    tid = (long)threadid + 1; // add one to thread id

    pthread\_mutex\_lock(&mutex\_lock); // lock mutex

    if (CurrentID == tid) { // if id's match

        myturn\_counter[tid-1]++; // increment "my turn" counter for this thread

        CurrentID++;

        if (CurrentID == 6)

            CurrentID = 1;

        printf("Thread #%ld: My turn (#%d)! CurrentID is now %d\n", tid, myturn\_counter[tid-1], CurrentID);

        /\* termination condition \*/

        if (myturn\_counter[tid-1] >= 10 && tid == 5) {

            pthread\_mutex\_destroy(&mutex\_lock); // destroy mutex

            exit(0); // terminate program

        }

    }

    else {

        notmyturn\_counter[tid-1]++; // increment "not my turn" counter for this thread

        printf("Thread #%ld: Not My Turn (#%d)! CurrentID is %d\n", tid, notmyturn\_counter[tid-1], CurrentID);

    }

    pthread\_mutex\_unlock(&mutex\_lock); // release mutex

    pthread\_exit(NULL); // exit thread

}

int main(void)

{

    pthread\_t threads[NUM\_THREADS]; // initialize threads

    pthread\_mutex\_init(&mutex\_lock, NULL); // initialize mutex

    long t;

    while (1) { // create threads until termination condition is reached

        for (t = 0; t < NUM\_THREADS; t++) {

            pthread\_create(&threads[t], NULL, increment\_id, (void\*)t);

        }

    }

    pthread\_mutex\_destroy(&mutex\_lock);

    pthread\_exit(NULL);

}

**Explanation:**

The code begins by initializing the “CurrentID” variable, along with the mutex, and two counter arrays for keeping count of the number of “My turn” and “Not my turn” statements for each thread ID. In “main()”, the thread array and mutex are initialized, and a nested for loop creates a thread with the method “increment\_id” for each thread ID (0 to 4) sequentially inside an infinite while loop.

The “increment\_id” method adds 1 to the thread ID (1 to 5), locks the mutex, and checks whether CurrentID is equal to the thread ID.

* If so, “my turn” counter is incremented for that thread ID, CurrentID is incremented (if it’s equal to 6, it goes back to 1), “my turn” is printed, and the termination condition is checked. Since the code needs to stop executing when all threads have printed “my turn” 10 times, an easy way to check this is by checking if the “my turn” counter for thread #5 is greater than or equal to 10. This works because thread #5 is the last thread that increments CurrentID. If the condition is true, the mutex is destroyed and a call to “exit()” is initiated.
* If not, the “not my turn” counter is incremented and “not my turn” is printed.

At the end, the mutex is unlocked and the thread is exited.

**Sample output:**

Thread #2: Not My Turn (#1)! CurrentID is 1

Thread #3: Not My Turn (#1)! CurrentID is 1

Thread #4: Not My Turn (#1)! CurrentID is 1

Thread #5: Not My Turn (#1)! CurrentID is 1

Thread #1: My turn (#1)! CurrentID is now 2

Thread #1: Not My Turn (#1)! CurrentID is 2

Thread #2: My turn (#1)! CurrentID is now 3

Thread #3: My turn (#1)! CurrentID is now 4

Thread #4: My turn (#1)! CurrentID is now 5

Thread #5: My turn (#1)! CurrentID is now 1

Thread #5: Not My Turn (#2)! CurrentID is 1

Thread #1: My turn (#2)! CurrentID is now 2

Thread #2: My turn (#2)! CurrentID is now 3

Thread #3: My turn (#2)! CurrentID is now 4

Thread #4: My turn (#2)! CurrentID is now 5

Thread #5: My turn (#2)! CurrentID is now 1

Thread #1: My turn (#3)! CurrentID is now 2

Thread #4: Not My Turn (#2)! CurrentID is 2

Thread #2: My turn (#3)! CurrentID is now 3

Thread #3: My turn (#3)! CurrentID is now 4

Thread #4: My turn (#3)! CurrentID is now 5

Thread #5: My turn (#3)! CurrentID is now 1

Thread #1: My turn (#4)! CurrentID is now 2

Thread #2: My turn (#4)! CurrentID is now 3

Thread #3: My turn (#4)! CurrentID is now 4

Thread #4: My turn (#4)! CurrentID is now 5

Thread #3: Not My Turn (#2)! CurrentID is 5

Thread #5: My turn (#4)! CurrentID is now 1

Thread #1: My turn (#5)! CurrentID is now 2

Thread #2: My turn (#5)! CurrentID is now 3

Thread #3: My turn (#5)! CurrentID is now 4

Thread #2: Not My Turn (#2)! CurrentID is 4

Thread #3: Not My Turn (#3)! CurrentID is 4

Thread #4: My turn (#5)! CurrentID is now 5

Thread #1: Not My Turn (#2)! CurrentID is 5

Thread #1: Not My Turn (#3)! CurrentID is 5

Thread #2: Not My Turn (#3)! CurrentID is 5

Thread #2: Not My Turn (#4)! CurrentID is 5

Thread #4: Not My Turn (#3)! CurrentID is 5

Thread #5: My turn (#5)! CurrentID is now 1

Thread #1: My turn (#6)! CurrentID is now 2

Thread #2: My turn (#6)! CurrentID is now 3

Thread #3: My turn (#6)! CurrentID is now 4

Thread #4: My turn (#6)! CurrentID is now 5

Thread #1: Not My Turn (#4)! CurrentID is 5

Thread #5: My turn (#6)! CurrentID is now 1

Thread #2: Not My Turn (#5)! CurrentID is 1

Thread #5: Not My Turn (#3)! CurrentID is 1

Thread #3: Not My Turn (#4)! CurrentID is 1

Thread #5: Not My Turn (#4)! CurrentID is 1

Thread #1: My turn (#7)! CurrentID is now 2

Thread #2: My turn (#7)! CurrentID is now 3

Thread #5: Not My Turn (#5)! CurrentID is 3

Thread #3: My turn (#7)! CurrentID is now 4

Thread #3: Not My Turn (#5)! CurrentID is 4

Thread #4: My turn (#7)! CurrentID is now 5

Thread #4: Not My Turn (#4)! CurrentID is 5

Thread #4: Not My Turn (#5)! CurrentID is 5

Thread #5: My turn (#7)! CurrentID is now 1

Thread #1: My turn (#8)! CurrentID is now 2

Thread #3: Not My Turn (#6)! CurrentID is 2

Thread #4: Not My Turn (#6)! CurrentID is 2

Thread #5: Not My Turn (#6)! CurrentID is 2

Thread #1: Not My Turn (#5)! CurrentID is 2

Thread #2: My turn (#8)! CurrentID is now 3

Thread #3: My turn (#8)! CurrentID is now 4

Thread #4: My turn (#8)! CurrentID is now 5

Thread #5: My turn (#8)! CurrentID is now 1

Thread #1: My turn (#9)! CurrentID is now 2

Thread #2: My turn (#9)! CurrentID is now 3

Thread #3: My turn (#9)! CurrentID is now 4

Thread #4: My turn (#9)! CurrentID is now 5

Thread #5: My turn (#9)! CurrentID is now 1

Thread #1: My turn (#10)! CurrentID is now 2

Thread #2: My turn (#10)! CurrentID is now 3

Thread #2: Not My Turn (#6)! CurrentID is 3

Thread #3: My turn (#10)! CurrentID is now 4

Thread #4: My turn (#10)! CurrentID is now 5

Thread #5: My turn (#10)! CurrentID is now 1

**Output data:**

|  |  |
| --- | --- |
| Thread ID# | Number of times it prints “Not My Turn!” |
| 1 | 5 |
| 2 | 6 |
| 3 | 6 |
| 4 | 6 |
| 5 | 6 |

**Part 2**

**Source code (with inline comments):**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define NUM\_THREADS 5 // total number of threads

#define BUFFER\_SIZE 5 // size of buffer (CHANGE THIS TO 5, 10, OR 25)

pthread\_mutex\_t mutex\_lock; // mutex lock

int buffer[BUFFER\_SIZE]; // shared buffer between producers and consumers

int next\_produced = 0, next\_consumed = 0; // variables for storing produced/consumed items

int in = 0, out = 0; // indexes of produced/consumed item

int num\_produced = 0, num\_consumed = 0; // counters for number of produced/consumed items

int full\_count = 0, empty\_count = 0; // counters for number of times producers/consumers go to sleep

void \*producer(void \**threadid*)

{

    long tid;

    tid = (long)threadid;

    while (1) {

        pthread\_mutex\_lock(&mutex\_lock); // lock mutex

        // check if the queue is full or not

        int i;

        int is\_not\_full = 0;

        for (i = 0; i < BUFFER\_SIZE; i++) {

            if (buffer[i] == -1) {

                is\_not\_full = 1;

            }

        }

        // if it is, release mutex and iterate

        if (is\_not\_full == 0) {

            pthread\_mutex\_unlock(&mutex\_lock);

            full\_count++;

        }

        // if not, put new integer in queue

        else {

            next\_produced++;

            buffer[in] = next\_produced; // put integer in slot

            num\_produced++;

            //printf("Producer #%ld: Put %d in slot %d\n", tid, next\_produced, in);

            in = (in + 1) % BUFFER\_SIZE;

            pthread\_mutex\_unlock(&mutex\_lock); // release mutex

        }

    }

    pthread\_exit(NULL);

}

void \*consumer(void \**threadid*)

{

    long tid;

    tid = (long)threadid;

    while (1) {

        pthread\_mutex\_lock(&mutex\_lock); // lock mutex

        // check if the queue is empty or not

        int i;

        int is\_not\_empty = 0;

        for (i = 0; i < BUFFER\_SIZE; i++) {

            if (buffer[i] != -1) {

                is\_not\_empty = 1;

            }

        }

        // if it is, release mutex and iterate

        if (is\_not\_empty == 0) {

            pthread\_mutex\_unlock(&mutex\_lock);

            empty\_count++;

        }

        // if not, take an integer from queue

        else {

            next\_consumed = buffer[out]; // take integer from slot

            //printf("Consumer #%ld: Took %d from slot %d\n", tid, next\_consumed, out);

            num\_consumed++;

            /\* termination condition \*/

            if (num\_consumed >= 200) {

                pthread\_mutex\_destroy(&mutex\_lock); // destroy mutex

                printf("\nThe producer went to sleep %d times due to full queue\n", full\_count);

                printf("The consumer went to sleep %d times due to empty queue\n", empty\_count);

                exit(0); // terminate program

            }

            buffer[out] = -1; // set slot to empty once integer is taken out

            out = (out + 1) % BUFFER\_SIZE;

            pthread\_mutex\_unlock(&mutex\_lock); // release mutex

        }

    }

    pthread\_exit(NULL);

}

int main(void)

{

    pthread\_t threads[NUM\_THREADS]; // initialize threads

    pthread\_mutex\_init(&mutex\_lock, NULL); // initialize mutex

    // set all elements in queue to -1 initially (-1 = an empty slot)

    int i;

    for (i = 0; i < BUFFER\_SIZE; i++) {

        buffer[i] = -1;

    }

    // create producer and consumer threads

    printf("Creating producer, ID = #%d\n", 0);

    pthread\_create(&threads[0], NULL, producer, (void\*)0);

    printf("Creating consumer, ID = #%d\n", 1);

    pthread\_create(&threads[1], NULL, consumer, (void\*)1);

    printf("Creating producer, ID = #%d\n", 2);

    pthread\_create(&threads[2], NULL, producer, (void\*)2);

    printf("Creating consumer, ID = #%d\n", 3);

    pthread\_create(&threads[3], NULL, consumer, (void\*)3);

    printf("Creating consumer, ID = #%d\n", 4);

    pthread\_create(&threads[4], NULL, consumer, (void\*)4);

    // join threads

    for (i = 0; i < NUM\_THREADS; i++) {

        pthread\_join(threads[i], NULL);

    }

    pthread\_mutex\_destroy(&mutex\_lock);

    pthread\_exit(NULL);

}

**Explanation:**

The code begins by initializing some variables. “BUFFER\_SIZE” is used to change the size of the buffer. Several variables are initialized to keep track of important data (see inline comments for details). In “main()”, the thread array and mutex are initialized. All elements of the buffer are initially set to “-1” to represent an empty slot in the queue. 2 producer threads and 3 consumer threads are then created, and their thread IDs are printed. The threads are joined at the end.

The “producer” method is made of an infinite while loop. It begins by locking the mutex. Then, it checks if the buffer is full or not by iterating through it and seeing if there are any empty slots (== -1). If there are no empty slots (i.e. queue is full), the mutex is unlocked, the sleep counter “full\_count” is incremented, and the loop iterates. If there is at least one empty slot, a new integer is produced and placed in the buffer at index “in”, counter “num\_produced” is incremented, the index “in” is advanced, and the mutex is unlocked.

The “consumer” method works in a similar fashion, except that it checks if the queue is empty by iterating through it and seeing if there are any non-empty slots (!= -1). If there are no integers to consume (i.e. queue is empty), the mutex is unlocked, the sleep counter “empty\_count” is incremented, and the loop iterates. If there is at least one item in the queue, the integer at the “out” index is consumed, and counter “num\_consumed” is incremented. If “num\_consumed” is greater than or equal to 200, the mutex is destroyed, the sleep counters are printed, and the program terminates. At the end, the buffer is set to “-1” at index “out” (to represent that the item has been consumed, and the slot is now empty), the index “out” is advanced, and the mutex is unlocked.

**Sample output (BUFFER\_SIZE = 5):**

Creating producer, ID = #0

Creating consumer, ID = #1

Creating producer, ID = #2

Creating consumer, ID = #3

Creating consumer, ID = #4

The producer went to sleep 3673477 times due to full queue

The consumer went to sleep 5193692 times due to empty queue

**Sample output (BUFFER\_SIZE = 10):**

Creating producer, ID = #0

Creating consumer, ID = #1

Creating producer, ID = #2

Creating consumer, ID = #3

Creating consumer, ID = #4

The producer went to sleep 1316470 times due to full queue

The consumer went to sleep 2184912 times due to empty queue

**Sample output (BUFFER\_SIZE = 25):**

Creating producer, ID = #0

Creating consumer, ID = #1

Creating producer, ID = #2

Creating consumer, ID = #3

Creating consumer, ID = #4

The producer went to sleep 385826 times due to full queue

The consumer went to sleep 525136 times due to empty queue

**Output analysis:**

Looking at these outputs, we see that the number of times the producers and consumers go to sleep decreases as the buffer size increases. This is to be expected as there is more space to accommodate more items in the queue, meaning that the threads don’t have to wait as long.

Also, the ratio is almost equal to for all outputs. Again, this is to be expected as there are 2 producers and 3 consumers, which means items are being consumed faster than they are being produced.