CSC 4310 Homework 2 Report

Ulysses Carlos

<2020-10-09 Fri>

Contents

1 Homework

1.1 Source Code

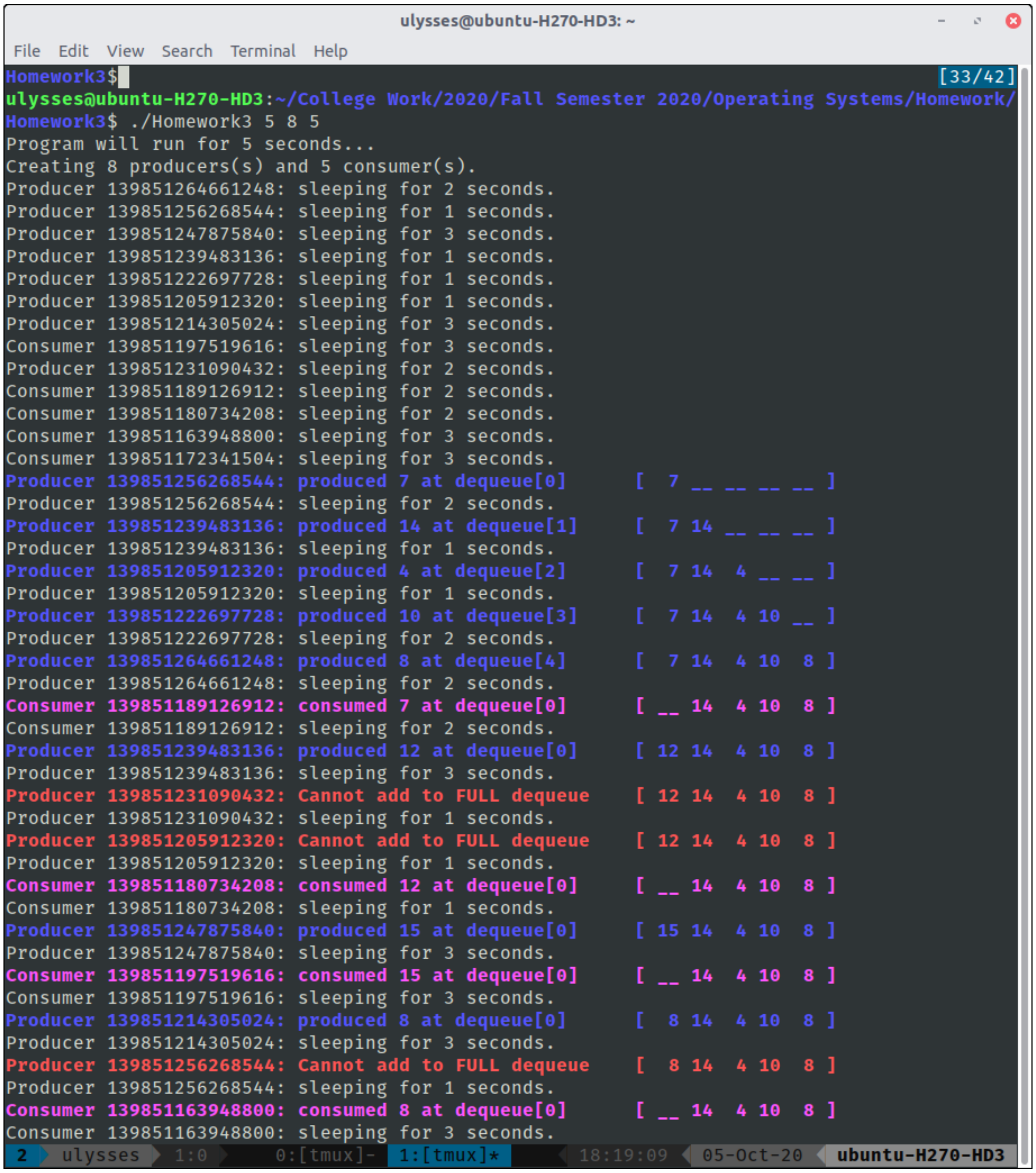
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <ctype.h>
#include <fcntl.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include "Buffer.h"

// Macros for program
#define SUCCESS (0)
#define FAILURE (1)
#define SLEEP_MAX (3)
#define MAX_VAL (20)
#define MIN_VAL (1)

buffer_item dequeue[BUFFER_SIZE];

const buffer_item producer_queue_end = BUFFER_SIZE - 1;
const buffer_item consumer_queue_end = 0;

sem_t *producer_semaphore;
sem_t *consumer_semaphore;

// Sets the maximum block time
struct timespec max_wait_time = {.tv_sec = 5, .tv_nsec = 0};

// Prevent multiple producers from inserting at the same time
pthread_mutex_t buffer_mutex = PTHREAD_MUTEX_INITIALIZER;

// Handles when to kill all producer and consumer threads
bool all_threads_active = true;

//------------------------------------------------------------------------------
// Functions to print producer and consumer messages
//------------------------------------------------------------------------------

void print_producer_message(buffer_item item, int position){
    fprintf(stderr, "\033[1;34m"); // Print blue for Producer
    fprintf(stderr, "Producer %lu: produced %d at dequeue[%d]\t",
            pthread_self(), item, position);
    print_dequeue(dequeue);
    fprintf(stderr, "\033[0m");
    fprintf(stderr, "\n");
}

void print_consumer_message(buffer_item item, int position){
    fprintf(stderr, "\033[1;35m"); // Print Magenta for Consumer
    fprintf(stderr, "Consumer␣%lu:␣consumed␣%d␣at␣dequeue[%d]\t",
            pthread_self(), item, position);
    print_dequeue(dequeue);
    fprintf(stderr, "\033[0m");
    fprintf(stderr, "\n");
}

//------------------------------------------------------------------------
// Dequeue declaration and definitions
//------------------------------------------------------------------------
bool dequeue_is_empty(const buffer_item *dequeue){
    for (int i = 0; i < BUFFER_SIZE; i++)
        if (dequeue[i] != empty_val) return false;
    return true;
}

//------------------------------------------------------------------------
// find_farthest_index(): Search for the farthest index where the producer
// can insert an item into the dequeue.
//------------------------------------------------------------------------
int find_farthest_index(const buffer_item *dequeue){

    for (int i = 0; i <= producer_queue_end; i++)
        if (dequeue[i] == empty_val)
            return i;
    return BUFFER_SIZE; // If the dequeue is full.
}

//------------------------------------------------------------------------
// find_closest_index(): Search for the closest index where the consumer
// can remove an item from the dequeue.
//------------------------------------------------------------------------
int find_closest_index(const buffer_item *dequeue){
    for (int i = consumer_queue_end; i < BUFFER_SIZE; i++) {
        if (dequeue[i] != empty_val)
            return i;
    }
    return BUFFER_SIZE; // Buffer is empty
}

//------------------------------------------------------------------------
// insert_item(): The producer inserts an item into the deque. If successful,
// the function returns 0. Otherwise it returns −1.
```

```
//------------------------------------------------------------------------------
int insert_item(buffer_item item){
    if (dequeue_is_empty(dequeue)){
        // Add a value to the buffer at the beginning
        dequeue[consumer_queue_end] = item;
        print_producer_message(item, consumer_queue_end);
        return SUCCESS;
    }

    // auto dequeue_debug = dequeue; // For debugging
    int p_index = find_farthest_index(dequeue);
    if (p_index == BUFFER_SIZE){
        fprintf(stderr, "\033[1;31m");
        fprintf(stderr, "Producer␣%lu:␣Cannot␣add␣to␣FULL␣dequeue\t",
                pthread_self());
        print_dequeue(dequeue);
        fprintf(stderr, "\033[0m");
        fprintf(stderr, "\n");
        return FAILURE;
    }
    // Now check if both producer and consumer have equal amount of index
    // space:
    int c_index = find_closest_index(dequeue);
    if ((c_index - consumer_queue_end) == (producer_queue_end - p_index)){
        // Add on the producer side and remove on the consumer side
        dequeue[p_index] = item;
        print_producer_message(item, p_index);
        return SUCCESS;
    }
    else {
        // Normal operation
        dequeue[p_index] = item;
        print_producer_message(item, p_index);
        return SUCCESS;
    }

}

//------------------------------------------------------------------------------
// remove_item(): The consumer removes the closet item from the buffer.
//------------------------------------------------------------------------------
int remove_item(buffer_item *item){
    if (dequeue_is_empty(dequeue)){
        fprintf(stderr, "\033[1;31m");
        fprintf(stderr, "Consumer␣%lu:␣EMPTY␣buffer:␣Skipping..\t",
                pthread_self());
        print_dequeue(dequeue);
        fprintf(stderr, "\033[0m");
        fprintf(stderr, "\n");
        return FAILURE;
    }

    // auto dequeue_debug = dequeue; // For debugging
    // Find closet value to consume
    int c_index = find_closest_index(dequeue);
    if (c_index == BUFFER_SIZE){
        fprintf(stderr, "Consumer␣%lu:␣First␣check␣didn't␣work␣it␣seems.␣"
                "␣The␣buffer␣is␣still␣empty.\n", pthread_self());
        return FAILURE;
    }

    // Now check producer_index as well.
    int p_index = find_farthest_index(dequeue);
    if ((c_index - consumer_queue_end) == (producer_queue_end - p_index)){
        // Add on the producer side and REMOVE on the consumer side
        *item = dequeue[c_index];
        dequeue[c_index] = empty_val;

        print_consumer_message(*item, c_index);
        return SUCCESS;
    }
    else {
        // Same as the above case.
        *item = dequeue[c_index];
        dequeue[c_index] = empty_val;
        print_consumer_message(*item, c_index);
        return SUCCESS;
    }
}

//------------------------------------------------------------------------------
// Producer(): Sleep for a random amount of time, and then insert a new item
// in the queue.
//------------------------------------------------------------------------------
void * producer(void *arg){
    buffer_item item;
    int sleep_time, check;
    while (all_threads_active) {
        // Sleep for random period of time

        sleep_time = rand() % SLEEP_MAX + 1;
        pthread_mutex_lock(&buffer_mutex);
        fprintf(stderr, "Producer␣%lu:␣sleeping␣for␣%d␣seconds.\n",
                pthread_self(), sleep_time);

        pthread_mutex_unlock(&buffer_mutex);
        sleep(sleep_time);
        // Generate random number:
        item = rand() % MAX_VAL + MIN_VAL;

        // Acquire the semaphore: Most of the problems occur here
        sem_timedwait(producer_semaphore, &max_wait_time);
        pthread_mutex_lock(&buffer_mutex);
        check = insert_item(item);
        // Release the mutex and then the semaphore
        pthread_mutex_unlock(&buffer_mutex);
        sem_post(consumer_semaphore);
    }

    return NULL;
}

//---------------------------------------------------------------------------
// Consumer(): Handle remove items from the queue
//---------------------------------------------------------------------------
void * consumer(void *arg){
    buffer_item item;
    int sleep_time;
    while (all_threads_active) {
        // Sleep for random period of time
        sleep_time = rand() % SLEEP_MAX + 1;
        pthread_mutex_lock(&buffer_mutex);
        fprintf(stderr, "Consumer␣%lu:␣sleeping␣for␣%d␣seconds.\n",
                pthread_self(), sleep_time);

        pthread_mutex_unlock(&buffer_mutex);
        // sem_post(semaphore);
        sleep(sleep_time);

        // Semaphore gets stuck at this point
        sem_timedwait(consumer_semaphore, &max_wait_time);
        pthread_mutex_lock(&buffer_mutex);
        int return_code = remove_item(&item);

        pthread_mutex_unlock(&buffer_mutex);
        // Release the semaphore
        sem_post(producer_semaphore);
    }

    return NULL;
}

//---------------------------------------------------------------------------
```

```c
// Helper functions:
//------------------------------------------------------------------------

bool str_isdigit(char *str){
    for (char *p = str; *p; p++)
        if (!isdigit(*p)) return false;
    return true;
}

void print_dequeue(const buffer_item *dequeue){
    fprintf(stderr, "[␣");
    for (int i = 0; i < BUFFER_SIZE; i++)
        fprintf(stderr, (dequeue[i] == empty_val) ? "__␣" : "%2d␣",
                dequeue[i]);
    fprintf(stderr, "]");
}


int main(int argc, char *argv[]){
    if (argc != 4){
        fprintf(stderr, "\033[1;31m");
        fprintf(stderr, "Usage:␣./Homework3␣[Sleep␣Time␣(s)]␣"
                "[#␣of␣Producer␣Threads]␣[#␣of␣Consumer␣Threads]\n");
        fprintf(stderr, "\033[0m");
        exit(EXIT_FAILURE);
    }

    // Prevent invalid input for arguments:
    bool check = str_isdigit(argv[1]) && str_isdigit(argv[2])
        && str_isdigit(argv[3]);

    if (!check){
        fprintf(stderr, "Error:␣Argument(s)␣contain␣nonnumerical␣characters.\n");
        exit(EXIT_FAILURE);
    }

    const int sleep_time = atoi(argv[1]);
    int producer_num = atoi(argv[2]);
    int consumer_num = atoi(argv[3]);

    if (sleep_time < 0){
        fprintf(stderr, "\033[1;31m");
        fprintf(stderr, "Error:␣Cannot␣have␣a␣negative␣sleep␣time.\n");
        fprintf(stderr, "\033[0m");
        exit(EXIT_FAILURE);
    }

    if (producer_num < 1 || consumer_num < 1){
```

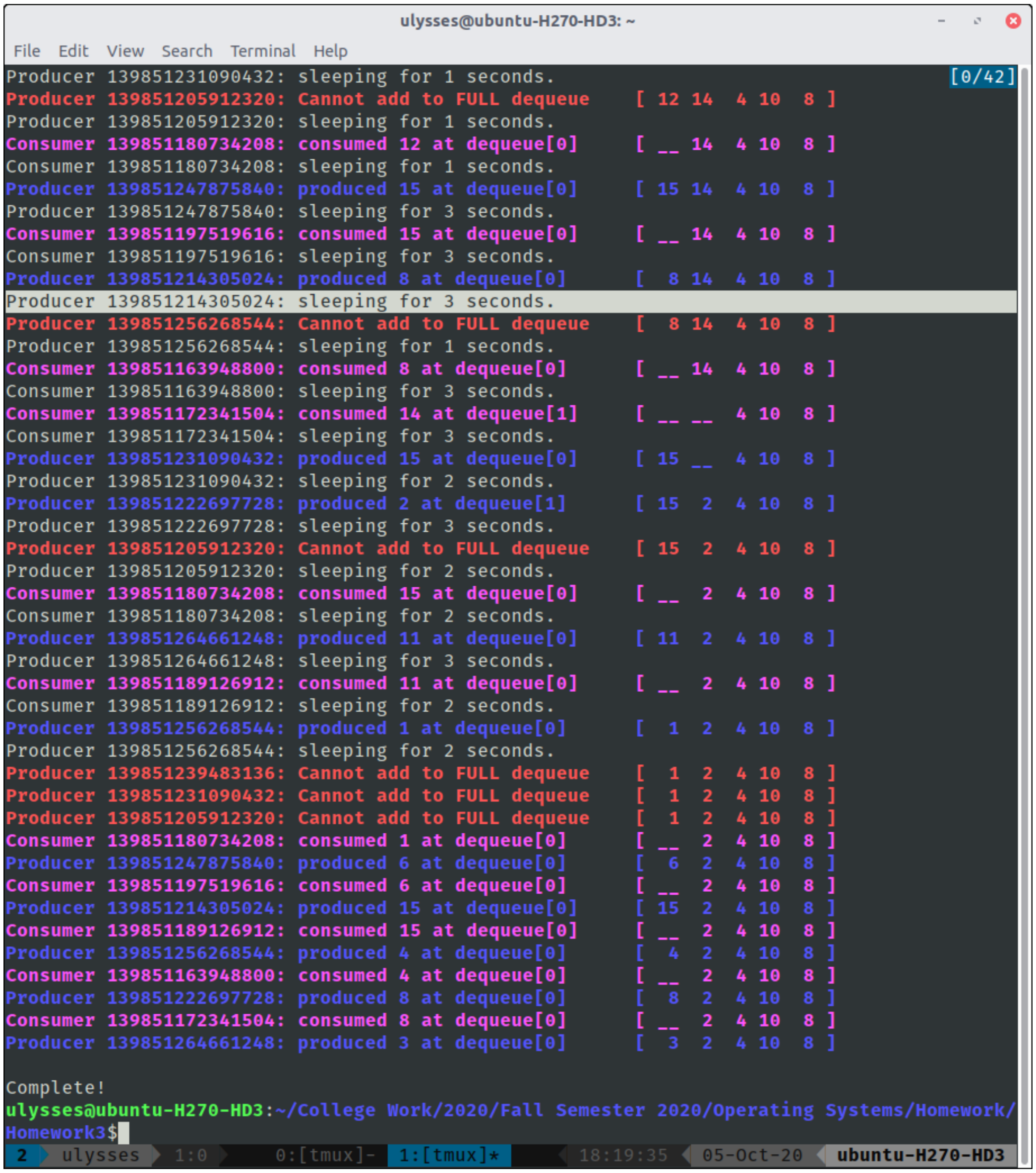
```
    fprintf(stderr, "\033[1;31m");
    fprintf(stderr, "Error:␣There␣must␣be␣at␣least␣one␣producer"
            "and␣consumer␣thread.\n");
    fprintf(stderr, "\033[0m");
    exit(EXIT_FAILURE);
}

// Fill the vector with empty_character:
for (int i = 0; i < BUFFER_SIZE; i++)
    dequeue[i] = empty_val;

//Initialize the semaphore and any mutexes:
producer_semaphore = sem_open("HW3:␣Producer␣Semaphore",
                              O_CREAT, 0666, 1);

consumer_semaphore = sem_open("HW3:␣Consumer␣Semaphore",
                              O_CREAT, 0666, 0);

pthread_mutex_init(&buffer_mutex, NULL);

// Now do the stuff
srand(time(NULL));

fprintf(stderr, "Program␣will␣run␣for␣%d␣seconds...\n", sleep_time);
fprintf(stderr, "Creating␣%d␣producers(s)␣and␣%d␣consumer(s).\n",
        producer_num, consumer_num);

// Create Producer and Consumer arrays:

pthread_t producer_list[producer_num];
pthread_t consumer_list[producer_num];

// Create pthreads for Producer and Consumers
for (int i = 0; i < producer_num; i++)
    pthread_create(&producer_list[i], NULL, producer, NULL);

for (int i = 0; i < consumer_num; i++)
    pthread_create(&consumer_list[i], NULL, consumer, NULL);

sleep(sleep_time);

// Kill all threads by activating the bool
all_threads_active = false;

// Now close each list
for (int i = 0; i < producer_num; i++)
    pthread_join(producer_list[i], NULL);
    for (int i = 0; i < consumer_num; i++)
        pthread_join(consumer_list[i], NULL);

    // Now exit.
    sem_destroy(producer_semaphore);
    sem_destroy(consumer_semaphore);
    fprintf(stdout, "\nComplete!\n");

}
```

1.2 Screenshots



```
Homework3$                                                                           [33/42]
ulysses@ubuntu-H270-HD3:~/College Work/2020/Fall Semester 2020/Operating Systems/Homework/
Homework3$ ./Homework3 5 8 5
Program will run for 5 seconds...
Creating 8 producers(s) and 5 consumer(s).
Producer 139851264661248: sleeping for 2 seconds.
Producer 139851256268544: sleeping for 1 seconds.
Producer 139851247875840: sleeping for 3 seconds.
Producer 139851239483136: sleeping for 1 seconds.
Producer 139851222697728: sleeping for 1 seconds.
Producer 139851205912320: sleeping for 1 seconds.
Producer 139851214305024: sleeping for 3 seconds.
Consumer 139851197519616: sleeping for 3 seconds.
Producer 139851231090432: sleeping for 2 seconds.
Consumer 139851189126912: sleeping for 2 seconds.
Consumer 139851180734208: sleeping for 2 seconds.
Consumer 139851163948800: sleeping for 3 seconds.
Consumer 139851172341504: sleeping for 3 seconds.
Producer 139851256268544: produced 7 at dequeue[0]       [  7 __ __ __ __ ]
Producer 139851256268544: sleeping for 2 seconds.
Producer 139851239483136: produced 14 at dequeue[1]      [  7 14 __ __ __ ]
Producer 139851239483136: sleeping for 1 seconds.
Producer 139851205912320: produced 4 at dequeue[2]       [  7 14  4 __ __ ]
Producer 139851205912320: sleeping for 1 seconds.
Producer 139851222697728: produced 10 at dequeue[3]      [  7 14  4 10 __ ]
Producer 139851222697728: sleeping for 2 seconds.
Producer 139851264661248: produced 8 at dequeue[4]       [  7 14  4 10  8 ]
Producer 139851264661248: sleeping for 2 seconds.
Consumer 139851189126912: consumed 7 at dequeue[0]       [ __ 14  4 10  8 ]
Consumer 139851189126912: sleeping for 2 seconds.
Producer 139851239483136: produced 12 at dequeue[0]      [ 12 14  4 10  8 ]
Producer 139851239483136: sleeping for 3 seconds.
Producer 139851231090432: Cannot add to FULL dequeue     [ 12 14  4 10  8 ]
Producer 139851231090432: sleeping for 1 seconds.
Producer 139851205912320: Cannot add to FULL dequeue     [ 12 14  4 10  8 ]
Producer 139851205912320: sleeping for 1 seconds.
Consumer 139851180734208: consumed 12 at dequeue[0]      [ __ 14  4 10  8 ]
Consumer 139851180734208: sleeping for 1 seconds.
Producer 139851247875840: produced 15 at dequeue[0]      [ 15 14  4 10  8 ]
Producer 139851247875840: sleeping for 3 seconds.
Consumer 139851197519616: consumed 15 at dequeue[0]      [ __ 14  4 10  8 ]
Consumer 139851197519616: sleeping for 3 seconds.
Producer 139851214305024: produced 8 at dequeue[0]       [  8 14  4 10  8 ]
Producer 139851214305024: sleeping for 3 seconds.
Producer 139851256268544: Cannot add to FULL dequeue     [  8 14  4 10  8 ]
Producer 139851256268544: sleeping for 1 seconds.
Consumer 139851163948800: consumed 8 at dequeue[0]       [ __ 14  4 10  8 ]
Consumer 139851163948800: sleeping for 3 seconds.
 2  ulysses  1:0     0:[tmux]- 1:[tmux]*      18:19:09  05-Oct-20  ubuntu-H270-HD3
```

Figure 1: Homework Output



```
Producer 139851231090432: sleeping for 1 seconds.
Producer 139851205912320: Cannot add to FULL dequeue    [ 12 14  4 10  8 ]
Producer 139851205912320: sleeping for 1 seconds.
Consumer 139851180734208: consumed 12 at dequeue[0]     [ __ 14  4 10  8 ]
Consumer 139851180734208: sleeping for 1 seconds.
Producer 139851247875840: produced 15 at dequeue[0]     [ 15 14  4 10  8 ]
Producer 139851247875840: sleeping for 3 seconds.
Consumer 139851197519616: consumed 15 at dequeue[0]     [ __ 14  4 10  8 ]
Consumer 139851197519616: sleeping for 3 seconds.
Producer 139851214305024: produced 8 at dequeue[0]      [  8 14  4 10  8 ]
Producer 139851214305024: sleeping for 3 seconds.
Producer 139851256268544: Cannot add to FULL dequeue    [  8 14  4 10  8 ]
Producer 139851256268544: sleeping for 1 seconds.
Consumer 139851163948800: consumed 8 at dequeue[0]      [ __ 14  4 10  8 ]
Consumer 139851163948800: sleeping for 3 seconds.
Consumer 139851172341504: consumed 14 at dequeue[1]     [ __ __  4 10  8 ]
Consumer 139851172341504: sleeping for 3 seconds.
Producer 139851231090432: produced 15 at dequeue[0]     [ 15 __  4 10  8 ]
Producer 139851231090432: sleeping for 2 seconds.
Producer 139851222697728: produced 2 at dequeue[1]      [ 15  2  4 10  8 ]
Producer 139851222697728: sleeping for 3 seconds.
Producer 139851205912320: Cannot add to FULL dequeue    [ 15  2  4 10  8 ]
Producer 139851205912320: sleeping for 2 seconds.
Consumer 139851180734208: consumed 15 at dequeue[0]     [ __  2  4 10  8 ]
Consumer 139851180734208: sleeping for 2 seconds.
Producer 139851264661248: produced 11 at dequeue[0]     [ 11  2  4 10  8 ]
Producer 139851264661248: sleeping for 3 seconds.
Consumer 139851189126912: consumed 11 at dequeue[0]     [ __  2  4 10  8 ]
Consumer 139851189126912: sleeping for 2 seconds.
Producer 139851256268544: produced 1 at dequeue[0]      [  1  2  4 10  8 ]
Producer 139851256268544: sleeping for 2 seconds.
Producer 139851239483136: Cannot add to FULL dequeue    [  1  2  4 10  8 ]
Producer 139851231090432: Cannot add to FULL dequeue    [  1  2  4 10  8 ]
Producer 139851205912320: Cannot add to FULL dequeue    [  1  2  4 10  8 ]
Consumer 139851180734208: consumed 1 at dequeue[0]      [ __  2  4 10  8 ]
Producer 139851247875840: produced 6 at dequeue[0]      [  6  2  4 10  8 ]
Consumer 139851197519616: consumed 6 at dequeue[0]      [ __  2  4 10  8 ]
Producer 139851214305024: produced 15 at dequeue[0]     [ 15  2  4 10  8 ]
Consumer 139851189126912: consumed 15 at dequeue[0]     [ __  2  4 10  8 ]
Producer 139851256268544: produced 4 at dequeue[0]      [  4  2  4 10  8 ]
Consumer 139851163948800: consumed 4 at dequeue[0]      [ __  2  4 10  8 ]
Producer 139851222697728: produced 8 at dequeue[0]      [  8  2  4 10  8 ]
Consumer 139851172341504: consumed 8 at dequeue[0]      [ __  2  4 10  8 ]
Producer 139851264661248: produced 3 at dequeue[0]      [  3  2  4 10  8 ]

Complete!
ulysses@ubuntu-H270-HD3:~/College Work/2020/Fall Semester 2020/Operating Systems/Homework/
Homework3$
 2  ulysses  1:0      0:[tmux]-  1:[tmux]*      18:19:35  05-Oct-20  ubuntu-H270-HD3
```

Figure 2: Homework Output (Cont.)