Scenario 1: More Consumers than Producers

```
trenonian@Michael-PC:~/Files/OS/HW/3$ ./buffer 10 4 8 10
Producer manderson_P3 produced 10
Consumer manderson_C1 comsumed 10
Producer manderson_P1 produced 11
Consumer manderson_C2 comsumed 11
Producer manderson_P3 produced 12
Consumer manderson_C5 comsumed 12
Producer manderson_P2 produced 13
Consumer manderson_C3 comsumed 13
Producer manderson_P3 produced 14
Consumer manderson_C7 comsumed 14
Producer manderson_P0 produced 15
Consumer manderson_C6 comsumed 15
Producer manderson_P3 produced 16
Consumer manderson_C0 comsumed 16
Producer manderson_P2 produced 17
Producer manderson_P1 produced 18
Consumer manderson_C2 comsumed 17
Consumer manderson_C5 comsumed 18
Producer manderson_P0 produced 19
Consumer manderson_C4 comsumed 19
Producer manderson_P3 produced 20
Consumer manderson_C3 comsumed 20
Producer manderson_P2 produced 21
Consumer manderson_C1 comsumed 21
Producer manderson_P1 produced 22
Consumer manderson_C0 comsumed 22
DONE
```

Scenario 2: Equal number of Producers and Consumers

```
trenonian@Michael-PC:~/Files/OS/HW/3$ ./buffer 6 8 8 23
Producer manderson_P3 produced 23
Producer manderson_P5 produced 24
Consumer manderson_C4 comsumed 23
Producer manderson_P1 produced 25
Producer manderson_P6 produced 26
Producer manderson_P3 produced 27
Consumer manderson_C1 comsumed 24
Consumer manderson_C7 comsumed 25
Producer manderson_P2 produced 28
Producer manderson_P5 produced 29
Producer manderson_P7 produced 30
Consumer manderson_C2 comsumed 26
Consumer manderson_C3 comsumed 27
Producer manderson_P0 produced 31
Producer manderson_P6 produced 33
Producer manderson_P4 produced 32
Producer manderson_P1 produced 34
Producer manderson_P5 produced 35
Consumer manderson_C4 comsumed 28
Consumer manderson_C6 comsumed 29
Producer manderson_P0 produced 36
Consumer manderson_C0 comsumed 30
Consumer manderson_C5 comsumed 31
Consumer manderson_C1 comsumed 33
Producer manderson_P2 produced 37
Producer manderson_P3 produced 38
Producer manderson_P7 produced 39
DONE
```

Scenario 3: More Producers than Consumers

```
trenonian@Michael-PC:~/Files/OS/HW/3$ ./buffer 8 10 3 15
Producer manderson_P3 produced 15
Producer manderson_P5 produced 16
Consumer manderson_C2 comsumed 15
Producer manderson_P1 produced 17
Producer manderson_P6 produced 18
Producer manderson_P9 produced 19
Producer manderson_P2 produced 20
Producer manderson_P1 produced 21
Producer manderson_P7 produced 22
Consumer manderson_C0 comsumed 16
Consumer manderson_C1 comsumed 17
Consumer manderson_C2 comsumed 18
Producer manderson_P0 produced 23
Producer manderson_P4 produced 24
Producer manderson_P6 produced 25
Producer manderson_P0 produced 26
Consumer manderson_C0 comsumed 19
Producer manderson_P2 produced 27
Consumer manderson_C2 comsumed 20
Producer manderson_P1 produced 28
Consumer manderson_C0 comsumed 21
Producer manderson_P8 produced 29
DONE
```

Source Code

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>

#define TRUE 1
#define FALSE 0
typedef int buffer_item;
#define BUFFER_SIZE 8

char FIRST_INITIAL = 'm';
char *LAST_NAME = "anderson";

buffer_item START_NUMBER;

typedef struct
{
        int index;
} parameter;

buffer_item produce_value;

buffer_item buffer[BUFFER_SIZE];

pthread_mutex_t mutex;
pthread_mutex_t value_mutex;
sem_t empty;
sem_t full;

int insertPointer = 0, removePointer = 0;

void *producer(void *param);
void *consumer(void *param);

int insert_item(buffer_item item, int index);

int remove_item(buffer_item *item, int index);
```

```
int main(int argc, char *argv[])
{
		int sleepTime, producerThreads, consumerThreads;
	int i, j;

	if (argc != 5)
	{
		printf("Useage: <sleep time> <producer threads> <consumer threads> <start number>\n");
		return -1;
	}

	sleepTime = atoi(argv[1]);
	producerThreads = atoi(argv[2]);
	consumerThreads = atoi(argv[3]);
	START_NUMBER = atoi(argv[4]);




	/* Initialize the synchronization tools */
	if (pthread_mutex_init(&mutex, NULL) != 0)
	{
		printf("mutex init has failed\n");
		return 1;
	}
	if (pthread_mutex_init(&value_mutex, NULL) != 0)
	{
		printf("mutex init has failed\n");
		return 1;
	}
	sem_init(&empty, 1, BUFFER_SIZE);
	sem_init(&full, 1, 0);

	produce_value = START_NUMBER;

	/* Create the producer and consumer threads */

	pthread_t producers[producerThreads];
	for (int i = 0; i < producerThreads; i++)
	{
		parameter *data = (parameter *)malloc(sizeof(parameter));
		data->index = i;
		pthread_create(&producers[i], NULL, producer, data);
	}

	pthread_t consumers[consumerThreads];
	for (int i = 0; i < consumerThreads; i++)
	{
		parameter *data = (parameter *)malloc(sizeof(parameter));
		data->index = i;
		pthread_create(&consumers[i], NULL, consumer, data);
	}

	/* Sleep for user specified number of seconds */
	sleep(sleepTime);

	sem_destroy(&empty);
	sem_destroy(&full);

	if (pthread_mutex_lock(&mutex) != 0)
	{
		return -1;
	}

	printf("DONE\n");

	return 0;
}
```

```
void *producer(void *param)
{
        /* Implementation of the producer thread -- refer to Figure 5.26 on page 256 */

        parameter *data = (parameter *)param;
        int index = data->index;
        free(data);
        buffer_item item;
        while (TRUE)
        {
                /* sleep for a random period of time */
                sleep(rand() % 5 + 1);

                /*
                        generate/retrieve item
                        ensures no duplicates
                */
                pthread_mutex_lock(&value_mutex);
                item = produce_value++;
                pthread_mutex_unlock(&value_mutex);

                /* insert item */
                insert_item(item, index);
        }
}

void *consumer(void *param)
{
        /* Implementation of the consumer thread -- refer to Figure 5.26 on page 256 */
        parameter *data = (parameter *)param;
        int index = data->index;
        free(data);
        buffer_item item;
        while (TRUE)
        {
                /* sleep for a random period of time */
                sleep(rand() % 5 + 1);

                remove_item(&item, index);
                /* process item */
        }
}
```

```
int insert_item(buffer_item item, int index)
{
        /*
                insert item into buffer
                return 0 if successful
                return -1 if not successful
        */

        if (sem_wait(&empty) != 0)
        {
                return -1;
        }
        if (pthread_mutex_lock(&mutex) != 0)
        {
                return -1;
        }

        // sem_wait(&empty);
        // pthread_mutex_lock(&mutex);

        buffer[insertPointer] = item;
        insertPointer++;
        insertPointer %= BUFFER_SIZE;
        printf("Producer %c%s_P%d produced %d\n",
                   FIRST_INITIAL, LAST_NAME, index, item);

        if (pthread_mutex_unlock(&mutex) != 0)
        {
                return -1;
        }
        if (sem_post(&full) != 0)
        {
                return -1;
        }

        // pthread_mutex_unlock(&mutex);
        // sem_post(&full);

        return 0;
}
```

```
int remove_item(buffer_item *item, int index)
{
        /*
                remove object from buffer
                place it in item
                return 0 if successful
                return -1 if not successful
        */

        if (sem_wait(&full) != 0)
        {
                return -1;
        }
        if (pthread_mutex_lock(&mutex) != 0)
        {
                return -1;
        }

        // sem_wait(&full);
        // pthread_mutex_lock(&mutex);

        *item = buffer[removePointer];
        removePointer++;
        removePointer %= BUFFER_SIZE;
        printf("Consumer %c%s_C%d comsumed %d\n",
                   FIRST_INITIAL, LAST_NAME, index, *item);

        if (pthread_mutex_unlock(&mutex) != 0)
        {
                return -1;
        }
        if (sem_post(&empty) != 0)
        {
                return -1;
        }

        // pthread_mutex_unlock(&mutex);
        // sem_post(&empty);

        return 0;
}
```