Solving Producers-Consumers Problem with Pthread

Nicholas Johnson & **Cy Dixon**
Professor Qi Li

November 30, 2023

1 Problem Analysis

1.1 Relationship Between Producer-Consumer Problem and Critical Section Problem

Similarities:

- *Shared Access to Resources* - Both problems involve multiple threads/processes sharing resources. In the producer-consumer problem, producers generate data while consumers consume these items. The critical section problem deals with synchronized access to critical sections of code or memory.
- *Controlling Access to Resources* - Both problems require synchronization mechanisms to ensure controlled access to shared resources, preventing conflicts.

Differences:

- *Resources* - The producer-consumer problem manages a buffer containing objects for production and consumption. Meanwhile, the critical section problem handles access to critical code segments or memory to avoid race conditions.
- *Purpose/Objective* - The producer-consumer problem focuses on synchronizing data production and consumption to maintain balance. On the other hand, the critical section problem concentrates on regulating access to specific code sections or shared resources to minimize simultaneous execution.

1.2 Difference Between Producer-Consumer Problem and Producers-Consumers Problem

The primary distinction lies in their scope:

- *Producer-Consumer Problem* deals with a single shared buffer, involving one set of producers and consumers.
- *Producers-Consumers Problem* involves multiple sets of producers and consumers, each with their own shared buffers, significantly increasing complexity.

2 Design

2.1 Pseudocode

```
1
2  Int buffer_size
3  Int buffer[buffer_size]
4  Int nextProduced = 0
5  Int nextConsumed = 0
6  sem_t mutex, full, empty
7
8  Struct ThreadData {
9      Int thread_id
10     Int upper_limit
11 }
12
13 Int main(args) {
14     // Take in command line input and set vars: buffersize, numProd, numCon,
15      upper_limit
16     // Initialize semaphores with 0 so they are only shared by threads
17     sem_init(&mutex, 0, 1)
18     sem_init(&full, 0, 0)
19     sem_init(&empty, 0, buffer_size)
20
21     // Set up threads
22     Pthread_t tid_producer[numProd]
23     Pthread_t tid_consumer[numCon]
24     // Make thread data structs: Struct ThreadData producer_data[numProd]
25     and similar for consumer
26
27     // For loop for making the threads and passing the data; one for
28     both producer and consumer
29
30     // For loops for joining the threads
31
32     // Finally, use sem_destroy for all semaphores
33 }
34
35 Void *consumer(void *param) {
36     // Set a struct equal to the param passes data
37     // While loop
38         // Call sem_wait on full to check the buffer and then mutex to get
39         the lock
40         // If statement to check if above the upper limit
41             // If so break
42
43         // Set item to consume by getting the buffer value
44         Next_consume_item++
45
46         Call sem_post for &mutex and &empty
```

```
47      // Outside of the while loop, use pthread_exit(NULL)
48  }
49
50  Void *producer(void *param) {
51      // Set the struct data
52      // While loop(1)
53          // Call the semaphores
54
55          // If for upper limit check
56              // If so break
57
58          // Set buffer index to the next produced item
59          // next_item_produced++
60
61          // Call mutex locks again using sem_post
62      // pthread_exit(NULL)
63  }
```

Listing 1: Pseudo Code for Producer-Consumer Problem

2.2 Code Segment 4.1

Producer-Consumer using Semaphores

2.3 Code Segment 4.2

Producer-Consumer using Spinlock

3 Presentation and Analysis of Elapsed Time

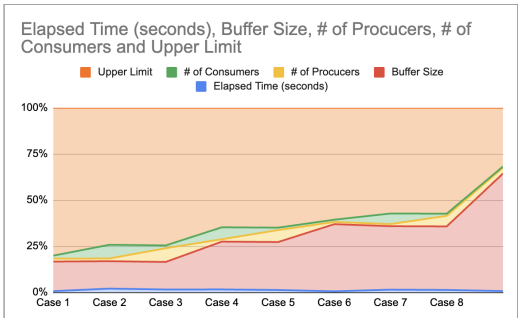
3.1 Data and Graphs

	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8	Case 9
Elapsed Time (seconds)	0.38	1.34	1.01	1.19	0.95	0.43	1.24	1.1	1.04
Buffer Size	10	10	10	20	20	30	30	30	100
# of Producers	1	1	5	1	5	1	1	5	5
# of Consumers	1	5	1	5	1	1	5	1	1
Upper Limit	50	50	50	50	50	50	50	50	50

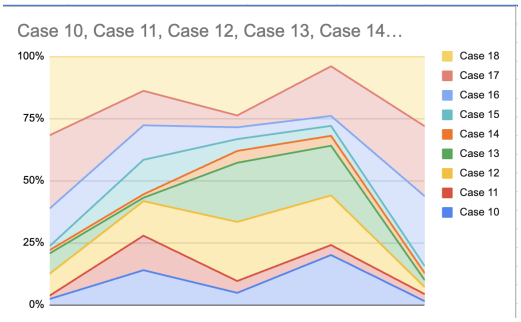
	Case 10	Case 11	Case 12	Case 13	Case 14	Case 15	Case 16	Case 17	Case 18
Elapsed Time (seconds)	1.19	0.7	4.55	4.23	0.69	0.83	7.82	15.24	16.41
Buffer Size	100	100	100	10	10	100	100	100	100
# of Producers	1	1	5	5	1	1	1	1	5
# of Consumers	5	1	5	5	1	1	1	5	1
Upper Limit	50	100	100	100	100	100	1000	1000	1000

3.2 Anaylsis

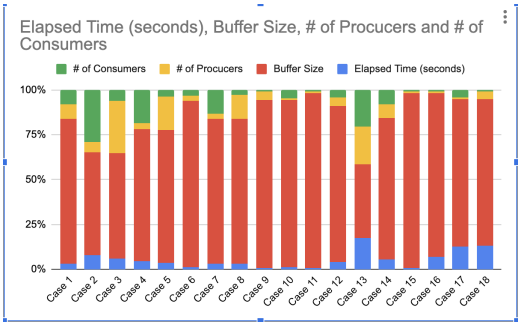
This data in Table 1, and shown via graphs in Figure 1, was collected using UTM virtual machine running Ubuntu Linux. We used a version of the Semaphore Code modified with a data collecting function to store the elapsed time, buffer size, producers, consumers, and upper limit. Each case was collected into a text file and then put onto these charts. We can see from the data that when we increase the upper limit, the elapsed time tends to fall. We can also see that the higher buffer size on the usual tends to increase the elapsed time as well. We can see from this data that with case 1-3 the data is rather inconclusive with various of elapsed time as it spans from 0.38 seconds to 1.34 seconds.One other factor that has a high impact on the run time of the program is the producer to consumer ratio. In respect to the producer consumer ration, when they are balanced, the execution time tends to be lower. Overall we see that the producer consumer balance plays a big role.



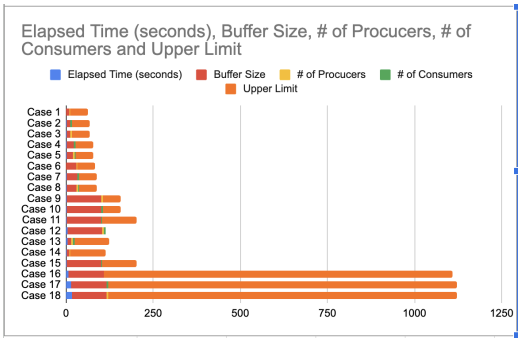
(a) Image 1



(b) Image 2



(c) Image 3



(d) Image 4

Figure 1: Graphs of Data

Semaphore Code

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

#define BUFFER_SIZE 10

struct ThreadData {
    int thread_id;
    int upper_limit;
};

int buffer[BUFFER_SIZE];
int next_produced_item = 0;
int next_consumed_item = 0;
sem_t mutex, full, empty;

void *producer(void *param) {
    struct ThreadData *data = (struct ThreadData *)param;
```

```
20      int id = data->thread_id;
21
22      while (1) {
23          sem_wait(&empty);
24          sem_wait(&mutex);
25
26          if (next_produced_item >= data->upper_limit) {
27              sem_post(&mutex);
28              sem_post(&full);
29              break;
30          }
31
32          buffer[next_produced_item % BUFFER_SIZE] = next_produced_item;
33          next_produced_item++;
34
35          sem_post(&mutex);
36          sem_post(&full);
37      }
38      pthread_exit(NULL);
39  }
40
41  void *consumer(void *param) {
42      struct ThreadData *data = (struct ThreadData *)param;
43      int id = data->thread_id;
44
45      while (1) {
46          sem_wait(&full);
47          sem_wait(&mutex);
48
49          if (next_consumed_item >= data->upper_limit) {
50              sem_post(&mutex);
51              sem_post(&empty);
52              break;
53          }
54
55          int item_to_consume = buffer[next_consumed_item % BUFFER_SIZE];
56          printf("%d, %d\n", item_to_consume, id);
57          next_consumed_item++;
58
59          sem_post(&mutex);
60          sem_post(&empty);
61      }
62      pthread_exit(NULL);
63  }
64
65  int main(int argc, char *argv[]) {
66      if (argc != 5) {
67          printf("Usage: %s <buffer_size> <num_producers> <num_consumers>
68  ␣␣␣␣␣␣␣␣␣␣<upper_limit>\n", argv[0]);
69          return 1;
```

```
70      }
71
72      int buffer_size = atoi(argv[1]);
73      int num_producers = atoi(argv[2]);
74      int num_consumers = atoi(argv[3]);
75      int upper_limit = atoi(argv[4]);
76
77      sem_init(&mutex, 0, 1);
78      sem_init(&full, 0, 0);
79      sem_init(&empty, 0, buffer_size);
80
81      pthread_t tid_producer[num_producers];
82      pthread_t tid_consumer[num_consumers];
83
84      struct ThreadData producer_data[num_producers];
85      struct ThreadData consumer_data[num_consumers];
86
87      for (int i = 0; i < num_producers; ++i) {
88          producer_data[i].thread_id = i + 1; // Adjusting thread ID starting
89          from 1
90          producer_data[i].upper_limit = upper_limit;
91          pthread_create(&tid_producer[i], NULL, producer, &producer_data[i]);
92      }
93
94      for (int i = 0; i < num_consumers; ++i) {
95          consumer_data[i].thread_id = i + 1; // Adjusting thread ID starting
96          from 1
97          consumer_data[i].upper_limit = upper_limit;
98          pthread_create(&tid_consumer[i], NULL, consumer, &consumer_data[i]);
99      }
100
101     for (int i = 0; i < num_producers; ++i) {
102         pthread_join(tid_producer[i], NULL);
103     }
104
105     for (int i = 0; i < num_consumers; ++i) {
106         pthread_join(tid_consumer[i], NULL);
107     }
108
109     sem_destroy(&mutex);
110     sem_destroy(&full);
111     sem_destroy(&empty);
112
113     return 0;
114 }
```

Listing 2: Producer-Consumer using Semaphores

Spinlock Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define BUFFER_SIZE 10

struct ThreadData {
    int thread_id;
    int upper_limit;
};

int buffer[BUFFER_SIZE];
int next_produced_item = 0;
int next_consumed_item = 0;
pthread_spinlock_t lock;

void *producer(void *param) {
    struct ThreadData *data = (struct ThreadData *)param;
    int id = data->thread_id;

    while (1) {
        pthread_spin_lock(&lock);

        if (next_produced_item >= data->upper_limit) {
            pthread_spin_unlock(&lock);
            break;
        }

        buffer[next_produced_item % BUFFER_SIZE] = next_produced_item;
        printf("%d, %d\n", next_produced_item, id);
        next_produced_item++;

        pthread_spin_unlock(&lock);
    }
    pthread_exit(NULL);
}

void *consumer(void *param) {
    struct ThreadData *data = (struct ThreadData *)param;
    int id = data->thread_id;

    while (1) {
        pthread_spin_lock(&lock);

        if (next_consumed_item >= data->upper_limit) {
            pthread_spin_unlock(&lock);
            break;
```

```
        }

        printf("%d, %d\n", buffer[next_consumed_item % BUFFER_SIZE], id);
        next_consumed_item++;

        pthread_spin_unlock(&lock);
    }
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    if (argc != 5) {
        printf("Usage: %s <buffer_size> <num_producers>
        <num_consumers> <upper_limit>\n", argv[0]);
        return 1;
    }

    int buffer_size = atoi(argv[1]);
    int num_producers = atoi(argv[2]);
    int num_consumers = atoi(argv[3]);
    int upper_limit = atoi(argv[4]);

    pthread_spin_init(&lock, PTHREAD_PROCESS_PRIVATE);

    pthread_t tid_producer[num_producers];
    pthread_t tid_consumer[num_consumers];

    struct ThreadData producer_data[num_producers];
    struct ThreadData consumer_data[num_consumers];

    for (int i = 0; i < num_producers; ++i) {
        producer_data[i].thread_id = i + 1;
        producer_data[i].upper_limit = upper_limit;
        pthread_create(&tid_producer[i], NULL, producer, &producer_data[i]);
    }

    for (int i = 0; i < num_consumers; ++i) {
        consumer_data[i].thread_id = i + 1;
        consumer_data[i].upper_limit = upper_limit;
        pthread_create(&tid_consumer[i], NULL, consumer, &consumer_data[i]);
    }

    for (int i = 0; i < num_producers; ++i) {
        pthread_join(tid_producer[i], NULL);
    }

    for (int i = 0; i < num_consumers; ++i) {
        pthread_join(tid_consumer[i], NULL);
    }

```

```
99      pthread_spin_destroy(&lock);
100
101     return 0;
102 }
```

Listing 3: Producer-Consumer using Spinlock