Documentation

The bounded buffer problem is when two processes, a producer and a consumer, must modify the same shared buffer object without deadlocking or interfering with the other. As stated in the assignment instructions, the buffer can only hold two items at once. When there are no items, the producer fills up the buffer until it is full. When the buffer is full, the consumer removes the items from the buffer. Both processes need to use semaphores in order to maintain mutual exclusion and not overproduce or over consume the buffer.

Admittedly, I had never worked with using the C language prior to this assignment, thankfully it was not too different to C++ and C#, which I had plenty of experience with. I tested my programs using Windows Subsystem for Linux with Ubuntu and Visual Studio Code, which I had set up the same day I started working after realizing I would have a difficult time using C++ and Windows. Working with something I was unfamiliar with paid off in the long run as I managed to finish the assignment in roughly two days of work.

I feel most of my time was spent learning how to use POSIX semaphores and shared memory. The websites geeksforgeeks.org and delftstack.com provided valuable information to me in finishing the assignment. Geeksforgeeks taught me how to use the POSIX semaphores and shared memory, alongside a basic producer consumer “Hello World” program that used the same commands asked for by the instructions that I used. On the other hand, Delfstack showed me how to use an integer array as a shared memory object and how to use mmap for creating the object. A lot of my self-teaching experience came from manually typing each line in the example programs given in order to gain an understanding of each line’s purpose and modifying the programs to something similar to what I am trying to accomplish.

My solution involved using a shared memory integer array as a buffer that stored numbers that would represent a unique item. The critical sections in the producer and consumer use a for-loop that produces or consumes the items in the array. Two semaphores are used in order to protect the shared memory, one that is called by the producer when the buffer is full and one that is called by the consumer when the buffer is empty. The producer file initializes the semaphores and shared memory and the consumer unlinks them when the program finishes.

My programs execute well except for one issue. Occasionally, the programs halt after the producer finishes filling the buffer for the first time. When using CTRL + C on the terminal, the consumer reports that it had encountered a segmentation fault. It is unknown to me how this can occur but it does not seem to cause any major problems as running the executables again will fix the problem. Running it a third time will close the extra producer thread as well. As far as I know, this is the only error in my programming.

If I had more time to develop the program, I would possibly create an abstract data type for the buffer; allowing the buffer to be used with any data type and not just limited to integers like in my code. Additionally, I would like to find a solution to the aforementioned issue and include additional functionality with the buffer, such as being able to view a specific index of the buffer. I would also like to figure out how to share constants between the two programs’ code without having to retype them, my guess would be making a header file with the declarations in them.

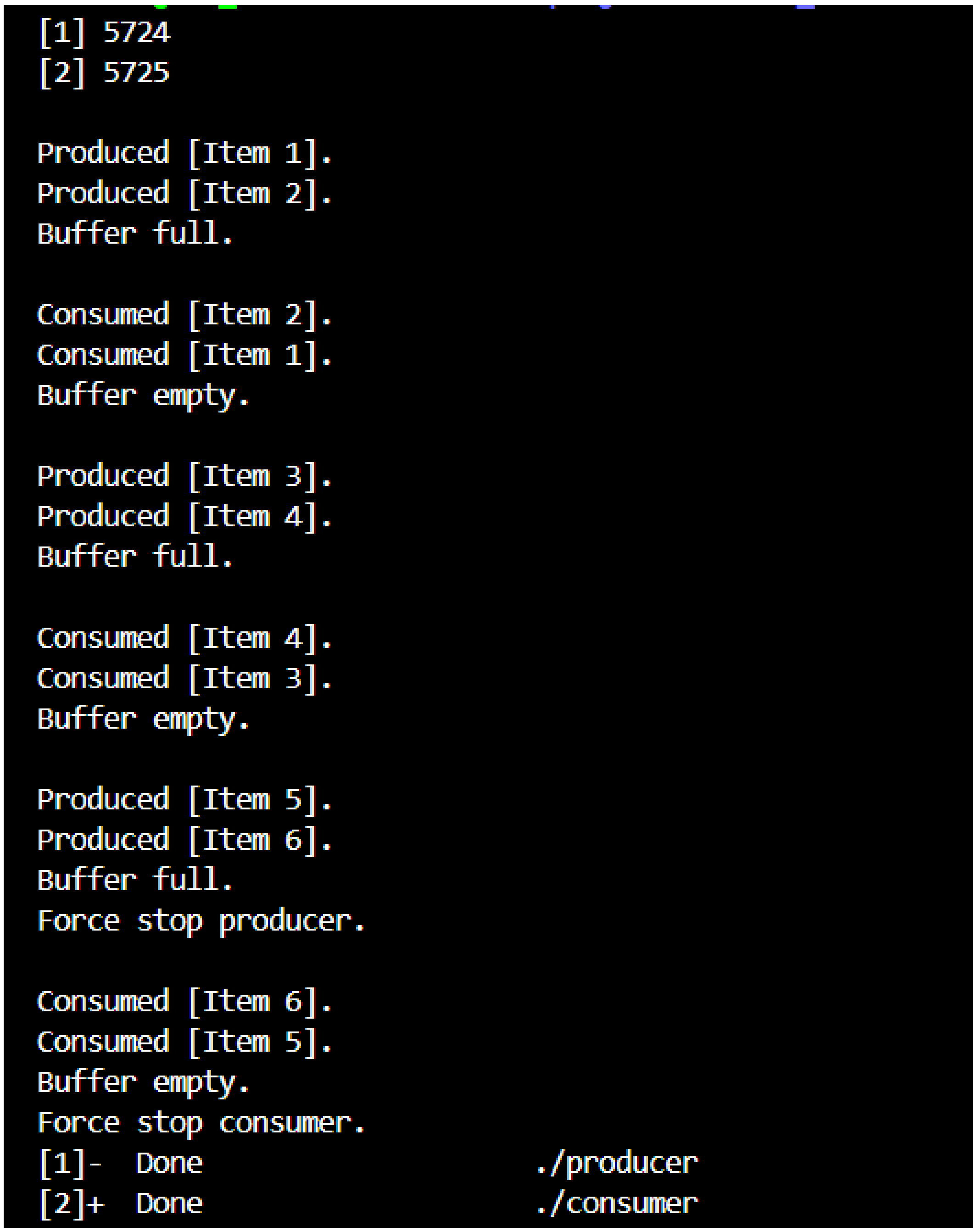
Websites used

Delftstack: How to Use mmap Function

Geeksforgeeks: POSIX Semaphores

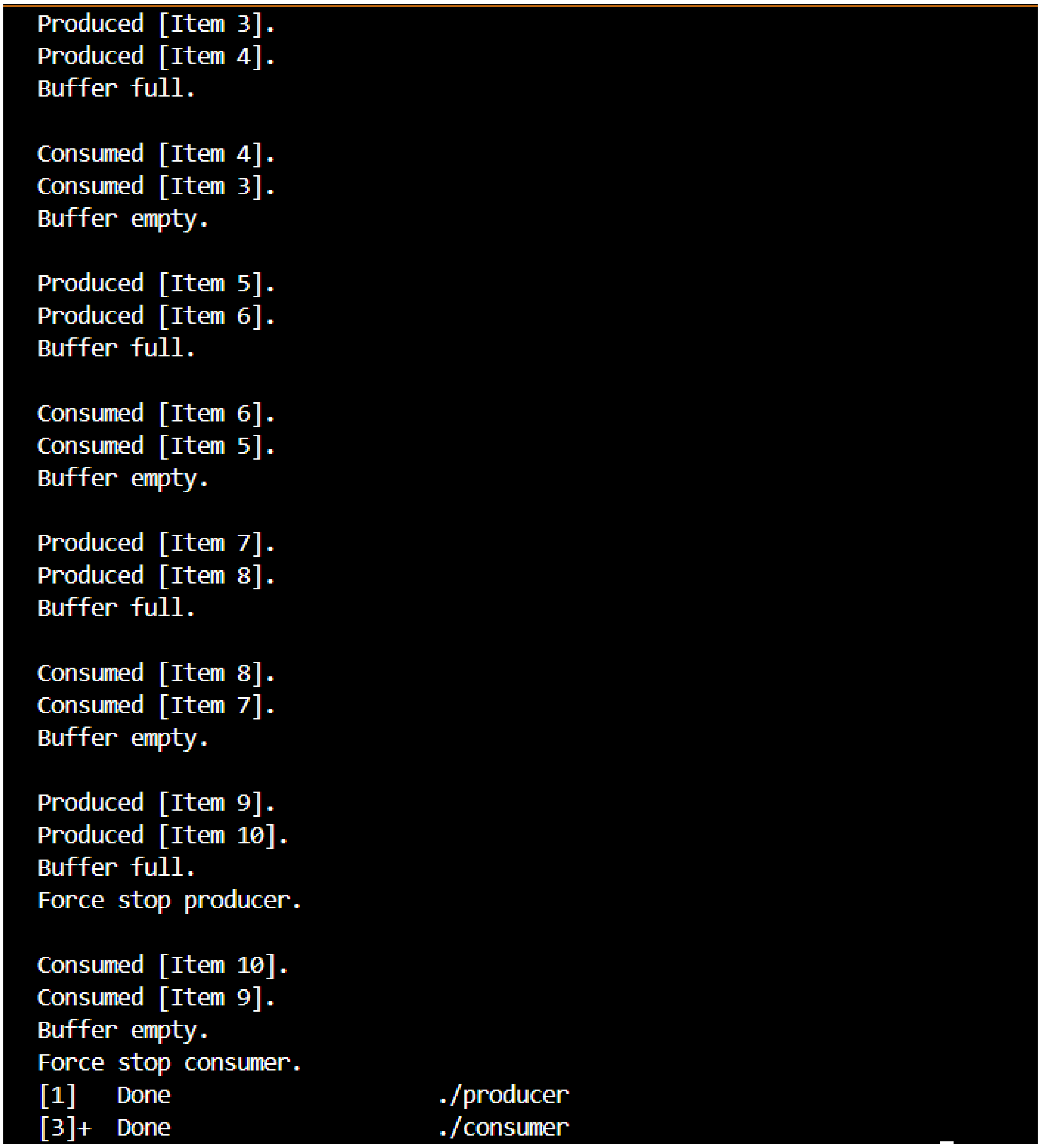
Geeksforgeeks: POSIX Shared Memory

Examples



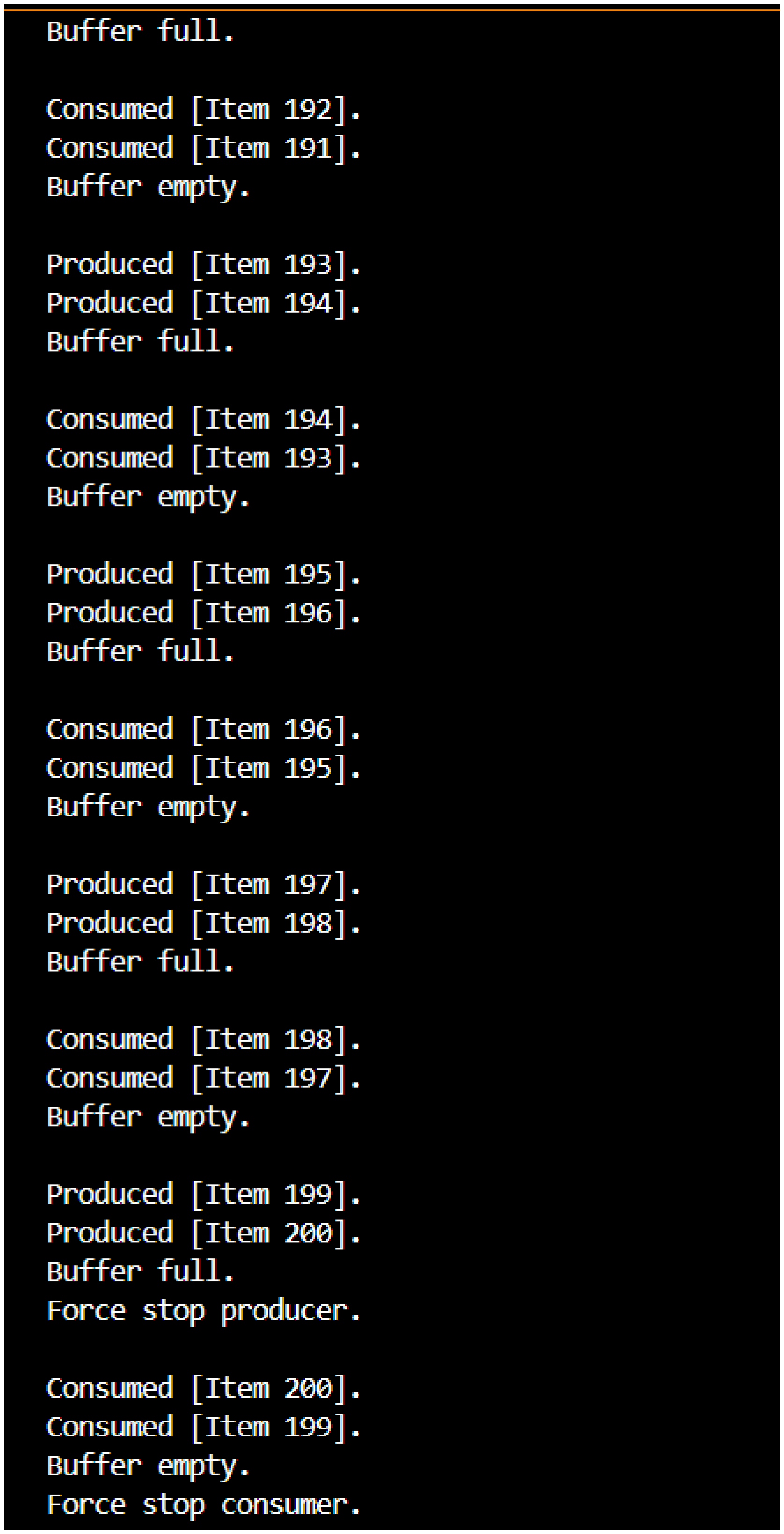
```
[1] 5724
[2] 5725

Produced [Item 1].
Produced [Item 2].
Buffer full.

Consumed [Item 2].
Consumed [Item 1].
Buffer empty.

Produced [Item 3].
Produced [Item 4].
Buffer full.

Consumed [Item 4].
Consumed [Item 3].
Buffer empty.

Produced [Item 5].
Produced [Item 6].
Buffer full.
Force stop producer.

Consumed [Item 6].
Consumed [Item 5].
Buffer empty.
Force stop consumer.
[1]-  Done                    ./producer
[2]+  Done                    ./consumer
```

Program on successful run with the total runs set to 3. The producer and consumer state which items have been produced/consumed and when they have finished their critical sections. They also report when they finish their total runs.



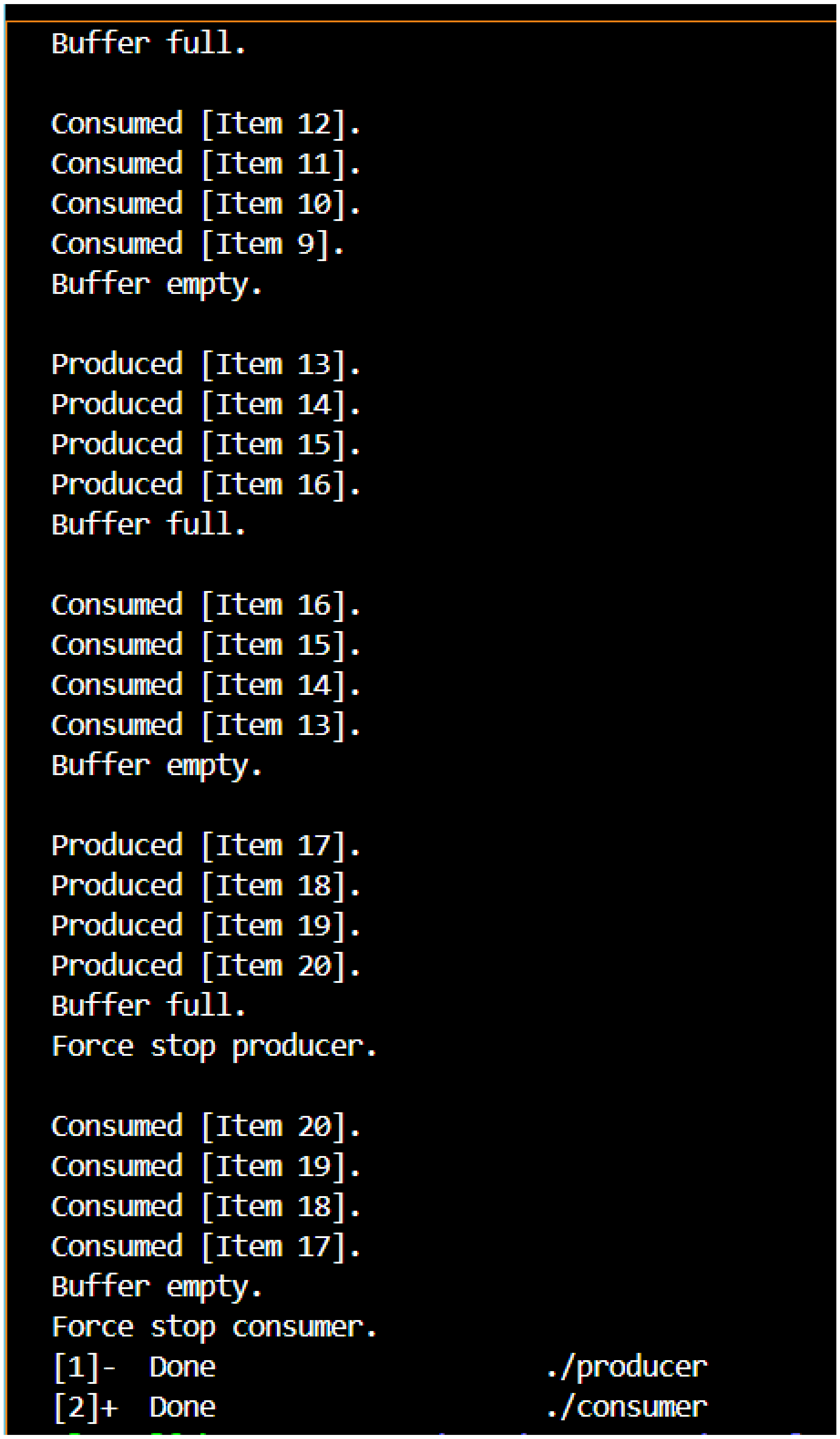
```
Produced [Item 3].
Produced [Item 4].
Buffer full.

Consumed [Item 4].
Consumed [Item 3].
Buffer empty.

Produced [Item 5].
Produced [Item 6].
Buffer full.

Consumed [Item 6].
Consumed [Item 5].
Buffer empty.

Produced [Item 7].
Produced [Item 8].
Buffer full.

Consumed [Item 8].
Consumed [Item 7].
Buffer empty.

Produced [Item 9].
Produced [Item 10].
Buffer full.
Force stop producer.

Consumed [Item 10].
Consumed [Item 9].
Buffer empty.
Force stop consumer.
[1]   Done                    ./producer
[3]+  Done                    ./consumer
```

Successful run with total runs set to 5 to show how the number of runs can be modified.



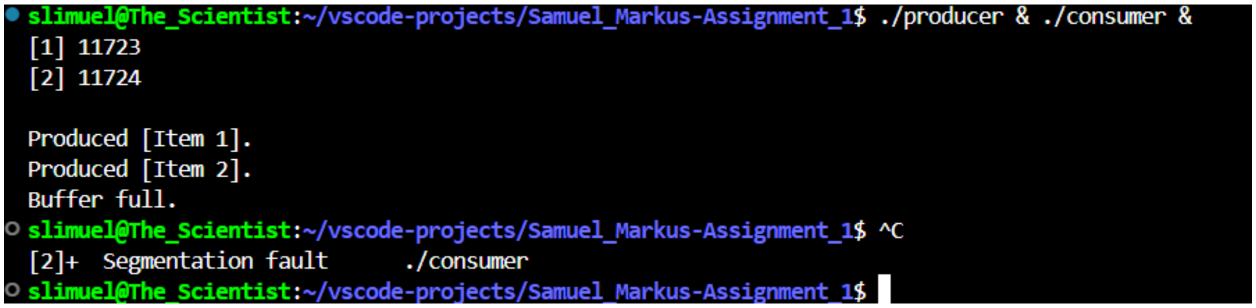
```
Buffer full.

Consumed [Item 192].
Consumed [Item 191].
Buffer empty.

Produced [Item 193].
Produced [Item 194].
Buffer full.

Consumed [Item 194].
Consumed [Item 193].
Buffer empty.

Produced [Item 195].
Produced [Item 196].
Buffer full.

Consumed [Item 196].
Consumed [Item 195].
Buffer empty.

Produced [Item 197].
Produced [Item 198].
Buffer full.

Consumed [Item 198].
Consumed [Item 197].
Buffer empty.

Produced [Item 199].
Produced [Item 200].
Buffer full.
Force stop producer.

Consumed [Item 200].
Consumed [Item 199].
Buffer empty.
Force stop consumer.
```

Successful run with total runs set to 100 to prove there are no memory leaks.



```
Buffer full.

Consumed [Item 12].
Consumed [Item 11].
Consumed [Item 10].
Consumed [Item 9].
Buffer empty.

Produced [Item 13].
Produced [Item 14].
Produced [Item 15].
Produced [Item 16].
Buffer full.

Consumed [Item 16].
Consumed [Item 15].
Consumed [Item 14].
Consumed [Item 13].
Buffer empty.

Produced [Item 17].
Produced [Item 18].
Produced [Item 19].
Produced [Item 20].
Buffer full.
Force stop producer.

Consumed [Item 20].
Consumed [Item 19].
Consumed [Item 18].
Consumed [Item 17].
Buffer empty.
Force stop consumer.
[1]-  Done                    ./producer
[2]+  Done                    ./consumer
```

Successful run with total runs set to 5 and buffer size set to 4 to show how the buffer size is modifiable.



```
slimuel@The_Scientist:~/vscode-projects/Samuel_Markus-Assignment_1$ ./producer & ./consumer &
[1] 11723
[2] 11724

Produced [Item 1].
Produced [Item 2].
Buffer full.
slimuel@The_Scientist:~/vscode-projects/Samuel_Markus-Assignment_1$ ^C
[2]+  Segmentation fault      ./consumer
slimuel@The_Scientist:~/vscode-projects/Samuel_Markus-Assignment_1$
```

Example of the occasional segmentation fault issue.