OS Lab Second Assignment

Saba Madadi

14 Dey 1403

Overview

In this exercise, we focus on producer-consumer problem and use multiprocessing to handle it. Information of producer and consumer is stored on their own sides. Connection between producer and consumer is handled in two ways: through pipe and through queue. Producer and consumer notify each other during production and consumption. This problem is covered in three scenarios: production stops, consumption stops, and concurrent production and consumption.
In second part, we will talk about readers-writers problem using multiprocessing. We consider two scenarios. In first scenario, readers have priority. In second scenario, writers have priority.

I. Producer-Consumer Problem

One of the problems in process synchronization is Producer-Consumer Problem. We want to handle it here using multiprocessing. Information will be stored on both sides. Information between consumer and producer will be transferred using queue or pipe. Both consumer and producer must notify each other when a certain amount has been consumed or a new item has been produced.

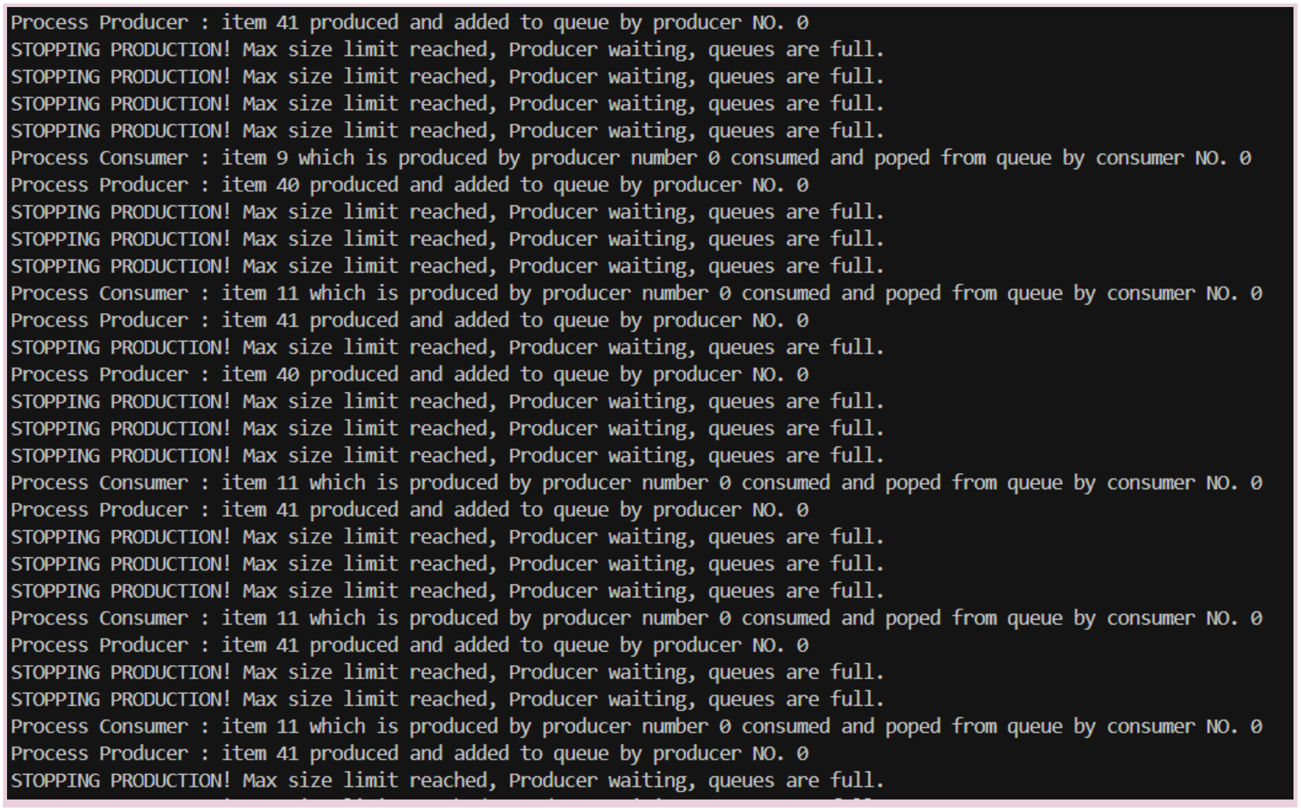
★ 1) Transferring information between consumer and producer using queue:

➔ A. Stopping Production

a: 1 Producer & 1 Consumer

Producer process: (0.1, 0.3,) Consumer process: (1.1, 2.0,)

A part of the output:


```
Process Producer : item 41 produced and added to queue by producer NO. 0
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Process Consumer : item 9 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 40 produced and added to queue by producer NO. 0
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Process Consumer : item 11 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 41 produced and added to queue by producer NO. 0
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Process Producer : item 40 produced and added to queue by producer NO. 0
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Process Consumer : item 11 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 41 produced and added to queue by producer NO. 0
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Process Consumer : item 11 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 41 produced and added to queue by producer NO. 0
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Process Consumer : item 11 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 41 produced and added to queue by producer NO. 0
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
```

We define *MAX_SIZE* and set num of producers (*PRODUCER_NUMBER*) and consumers (*CONSUMER_NUMBER*) to handle different situations.
We create a Message class, which exchanges messages through a shared queue. We stated earlier that we are using multiprocessing to handle issue without a shared memory space; here, we are only exchanging messages. Inside this class, *message_type* can take a value of 0 or 1. We contract if it is 0, a producer has generated an item, and if it is 1, a consumer has consumed an item.

process_number shows which producer or consumer is producing or consuming item, and we also keep track of item number being produced or consumed.

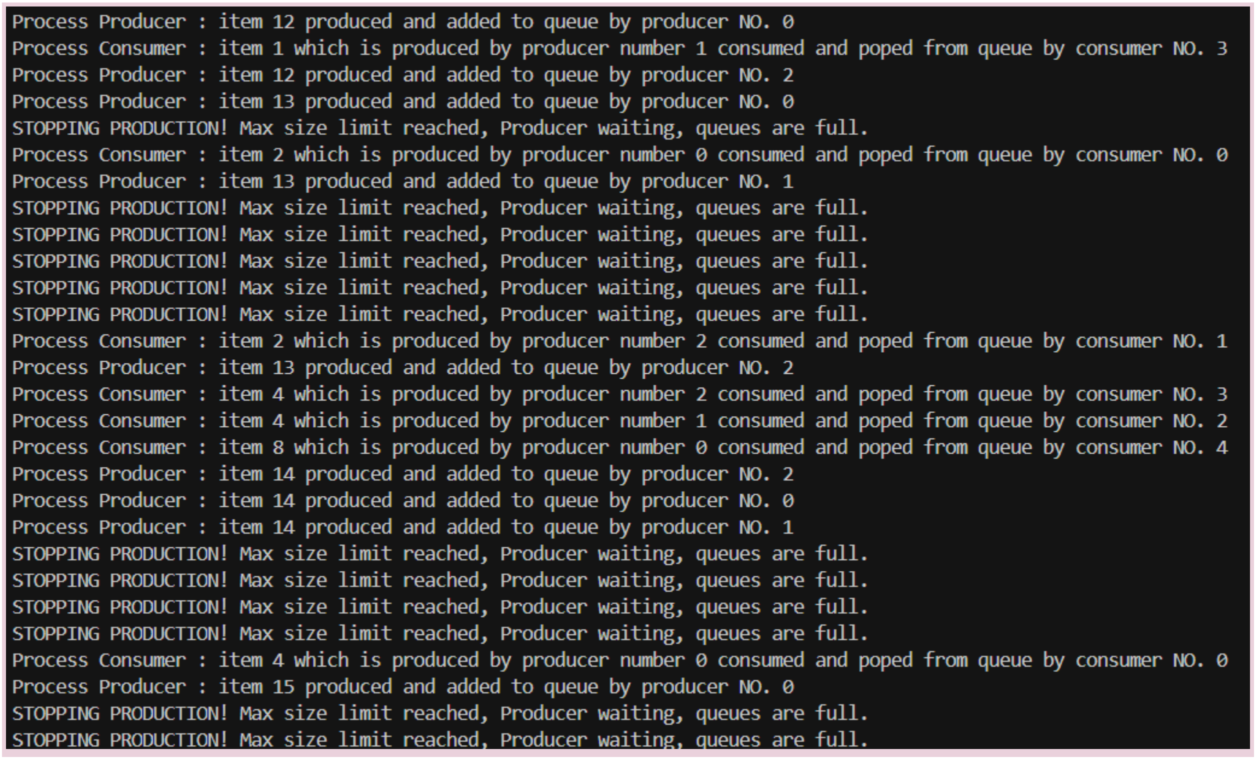
We have *read_list*, a list of booleans. We initialize this list with false for as many entries as total of *PRODUCER_NUMBER* and *CONSUMER_NUMBER*. This will become true when a producer reads it, for example, producer num 5, making fifth entry true. When all have read it, message is removed from queue.
We have *set_read*, shows someone has read message, and *check_read*, checks if it have readen. If it's not, will read it, set it as read, and then return all message information.
We have *check_list* to see if whole list is true.
Producer class has a queue and a *stop_event* for stopping at end. In *run*, *counter* shows which item we are producing. *waiting* tells whether it should wait or not, and *items* holds queue items in local memory.
item_list is used for every time a consumer comes, it reads all messages from queue and puts them back in.
T is message, and we check if we have read it or not. If not, we use its information. If *message_type* is zero, we append it to item list according to our agreement, and if it is one, we remove it.
If *waiting* is less than *MAX_SIZE*, we set *item* equal to *counter*, create a new message, and send it to everyone, notifying them I have produced this item.
We have Consumer class, which is similar to Producer, but we check if *waiting* is greater than zero to consume an item and produce a message to notify everyone; if not, we say it's empty.
We create a list of producers and consumers, start them all, wait for 20 seconds, set *stop_event* to notify everyone to stop, and then join all producers and consumers.

In this case, producer's speed is faster than consumer's. We have stop production because if we produce too much, we will run into problem of not having enough space.

b: 3 Producer & 5 Consumer

Producer process: (i, 0.1, 0.3,) Consumer process: (i, 1.1, 2.0,)

A part of the output:



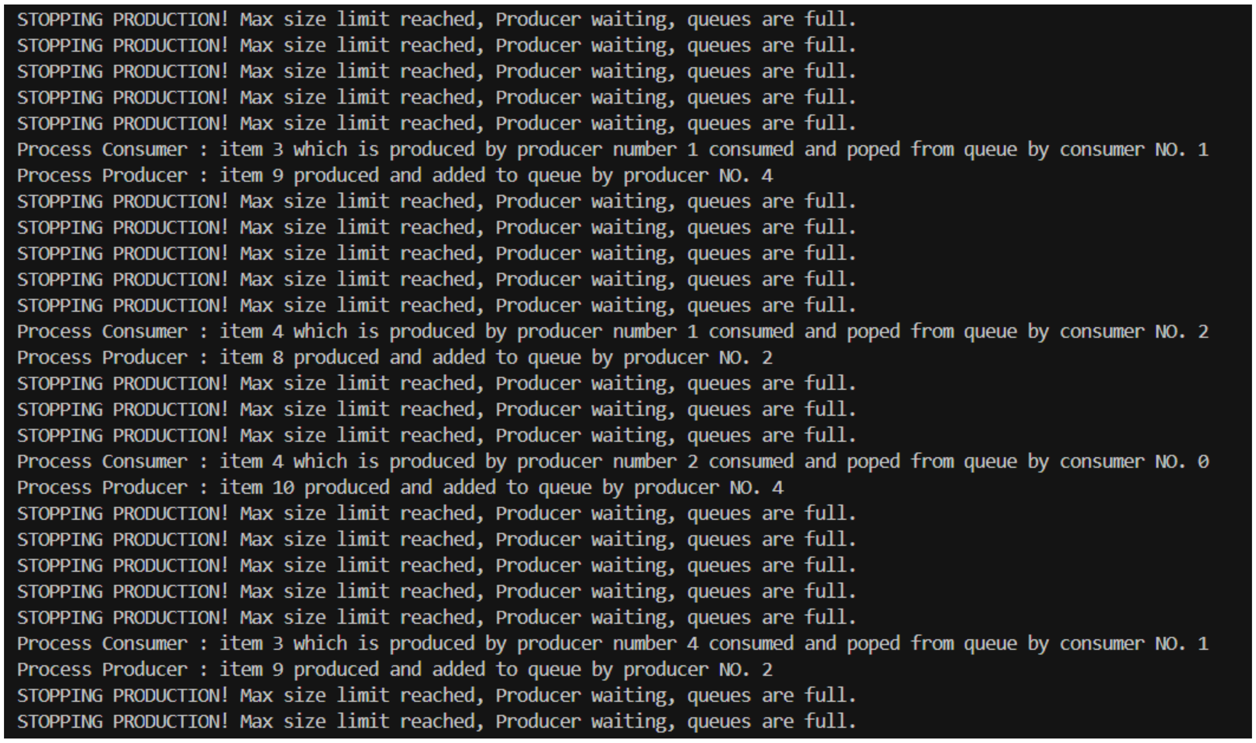
```
Process Producer : item 12 produced and added to queue by producer NO. 0
Process Consumer : item 1 which is produced by producer number 1 consumed and poped from queue by consumer NO. 3
Process Producer : item 12 produced and added to queue by producer NO. 2
Process Producer : item 13 produced and added to queue by producer NO. 0
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Process Consumer : item 2 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 13 produced and added to queue by producer NO. 1
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Process Consumer : item 2 which is produced by producer number 2 consumed and poped from queue by consumer NO. 1
Process Producer : item 13 produced and added to queue by producer NO. 2
Process Consumer : item 4 which is produced by producer number 2 consumed and poped from queue by consumer NO. 3
Process Consumer : item 4 which is produced by producer number 1 consumed and poped from queue by consumer NO. 2
Process Consumer : item 8 which is produced by producer number 0 consumed and poped from queue by consumer NO. 4
Process Producer : item 14 produced and added to queue by producer NO. 2
Process Producer : item 14 produced and added to queue by producer NO. 0
Process Producer : item 14 produced and added to queue by producer NO. 1
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Process Consumer : item 4 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 15 produced and added to queue by producer NO. 0
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
```

In this case we have same situation, only we should handle 3 producers & 5 consumers.

c: 5 Producer & 3 Consumer

Producer process: (i, 0.1, 0.3,)   Consumer process: (i, 1.1, 2.0,)

A part of the output:



```
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Process Consumer : item 3 which is produced by producer number 1 consumed and poped from queue by consumer NO. 1
Process Producer : item 9 produced and added to queue by producer NO. 4
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Process Consumer : item 4 which is produced by producer number 1 consumed and poped from queue by consumer NO. 2
Process Producer : item 8 produced and added to queue by producer NO. 2
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Process Consumer : item 4 which is produced by producer number 2 consumed and poped from queue by consumer NO. 0
Process Producer : item 10 produced and added to queue by producer NO. 4
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Process Consumer : item 3 which is produced by producer number 4 consumed and poped from queue by consumer NO. 1
Process Producer : item 9 produced and added to queue by producer NO. 2
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
```

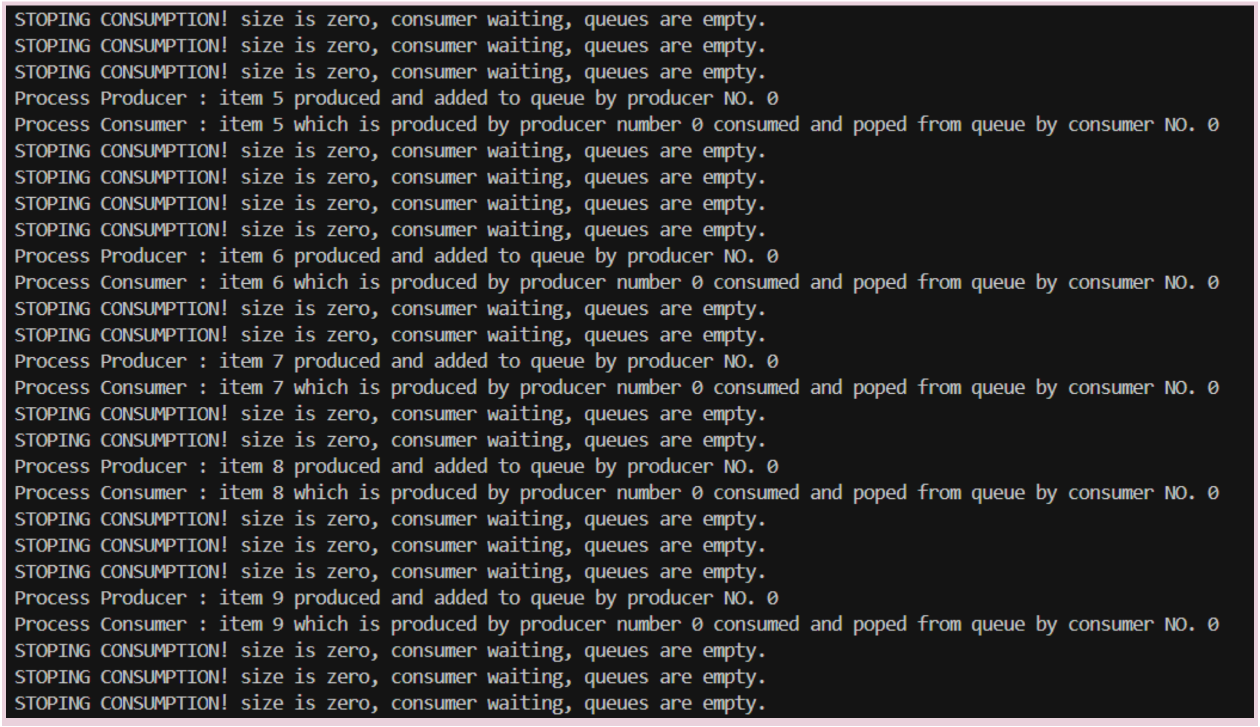
In this case we have same situation, only we should handle 5 producers & 3 consumers.

➔ B. Stopping Consumption

a: 1 Producer & 1 Consumer

Producer process: (1.1, 2.0,) Consumer process: (0.1, 0.3,)

A part of the output:



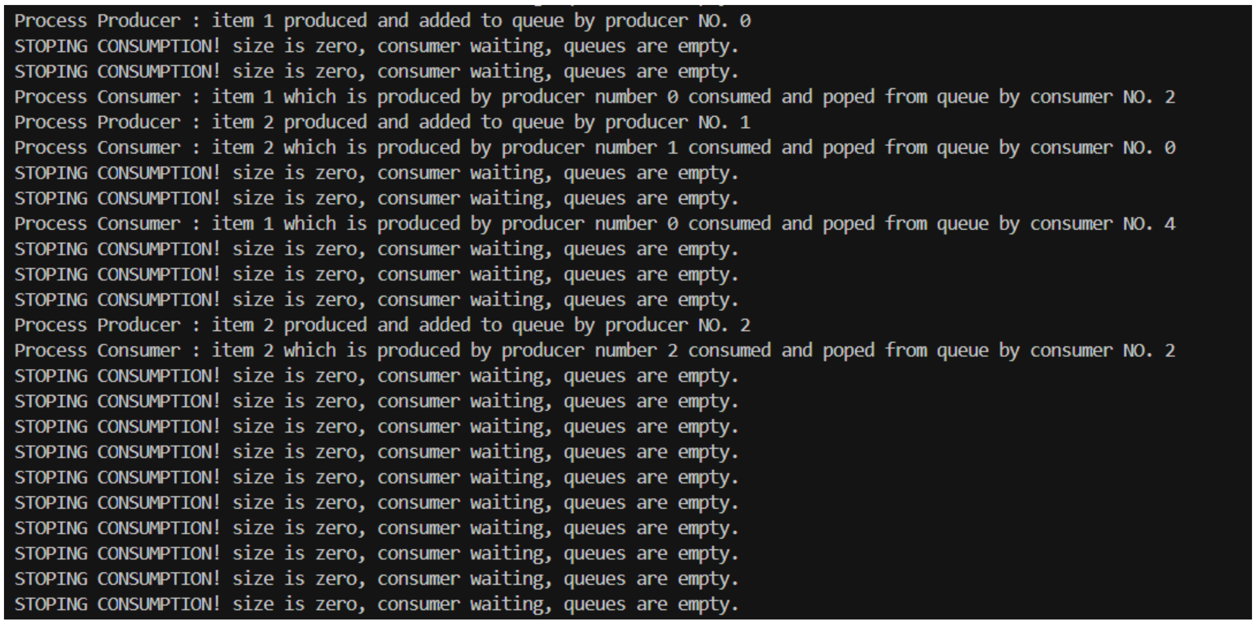
```
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
Process Producer : item 5 produced and added to queue by producer NO. 0
Process Consumer : item 5 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
Process Producer : item 6 produced and added to queue by producer NO. 0
Process Consumer : item 6 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
Process Producer : item 7 produced and added to queue by producer NO. 0
Process Consumer : item 7 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
Process Producer : item 8 produced and added to queue by producer NO. 0
Process Consumer : item 8 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
Process Producer : item 9 produced and added to queue by producer NO. 0
Process Consumer : item 9 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
```

Explanation for this part is similar to A for 1 producer and 1 consumer. Difference here is the consumption speed is faster than production speed. Because of this, we face problem of having nothing to consume.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.

b: 3 Producer & 5 Consumer

Producer process: (i, 1.1, 2.0,) Consumer process: (i, 0.1, 0.3,)

A part of the output:



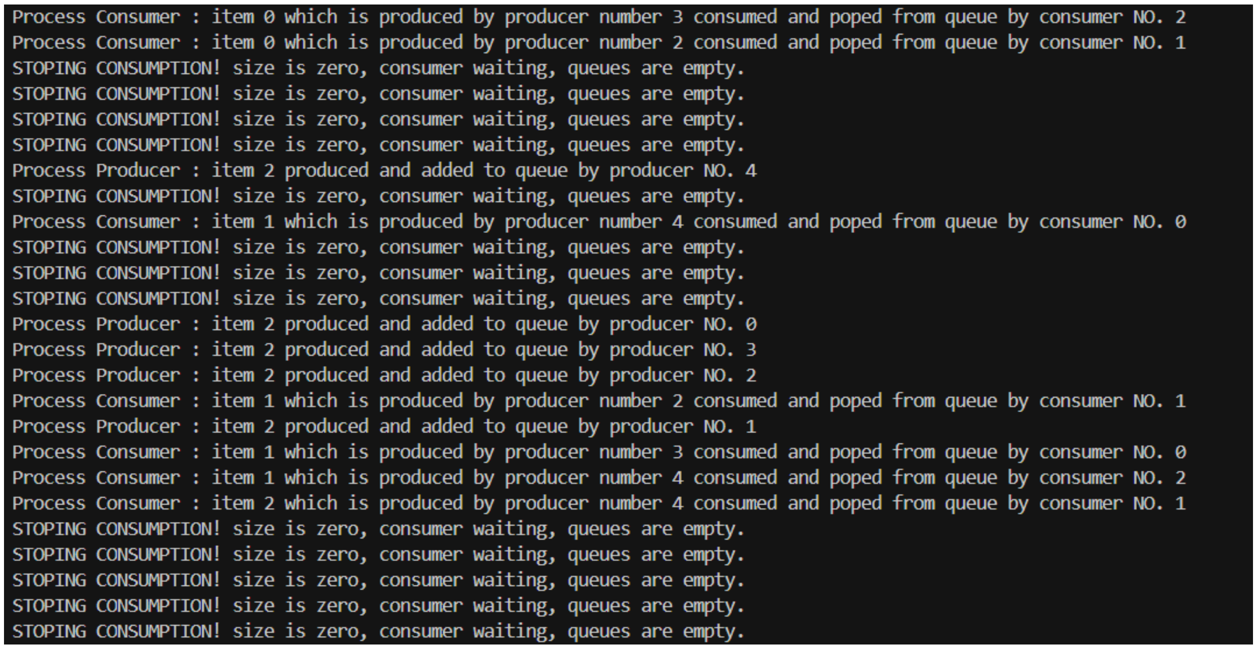
```
Process Producer : item 1 produced and added to queue by producer NO. 0
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
Process Consumer : item 1 which is produced by producer number 0 consumed and poped from queue by consumer NO. 2
Process Producer : item 2 produced and added to queue by producer NO. 1
Process Consumer : item 2 which is produced by producer number 1 consumed and poped from queue by consumer NO. 0
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
Process Consumer : item 1 which is produced by producer number 0 consumed and poped from queue by consumer NO. 4
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
Process Producer : item 2 produced and added to queue by producer NO. 2
Process Consumer : item 2 which is produced by producer number 2 consumed and poped from queue by consumer NO. 2
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
```

In this case we have same situation, only we should handle 3 producers & 5 consumers.

c: 5 Producer & 3 Consumer

Producer process: (i, 1.1, 2.0,)   Consumer process: (i, 0.1, 0.3,)

A part of the output:



```
Process Consumer : item 0 which is produced by producer number 3 consumed and poped from queue by consumer NO. 2
Process Consumer : item 0 which is produced by producer number 2 consumed and poped from queue by consumer NO. 1
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
Process Producer : item 2 produced and added to queue by producer NO. 4
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
Process Consumer : item 1 which is produced by producer number 4 consumed and poped from queue by consumer NO. 0
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
Process Producer : item 2 produced and added to queue by producer NO. 0
Process Producer : item 2 produced and added to queue by producer NO. 3
Process Producer : item 2 produced and added to queue by producer NO. 2
Process Consumer : item 1 which is produced by producer number 2 consumed and poped from queue by consumer NO. 1
Process Producer : item 2 produced and added to queue by producer NO. 1
Process Consumer : item 1 which is produced by producer number 3 consumed and poped from queue by consumer NO. 0
Process Consumer : item 1 which is produced by producer number 4 consumed and poped from queue by consumer NO. 2
Process Consumer : item 2 which is produced by producer number 4 consumed and poped from queue by consumer NO. 1
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.
```

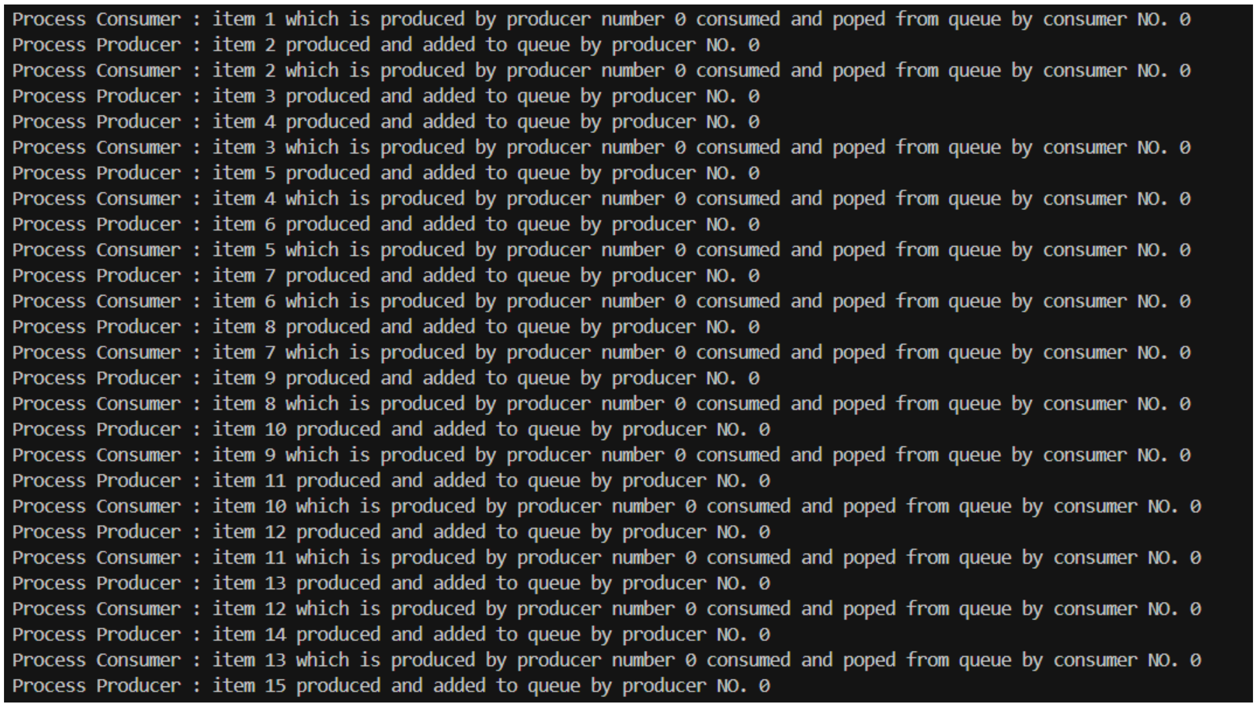
In this case we have same situation, only we should handle 5 producers & 3 consumers.

➔ C. Concurrent Production & Consumption

a: 1 Producer & 1 Consumer

Producer process: (0.9, 1.0,)   Consumer process: (0.9, 1.1,)

A part of the output:



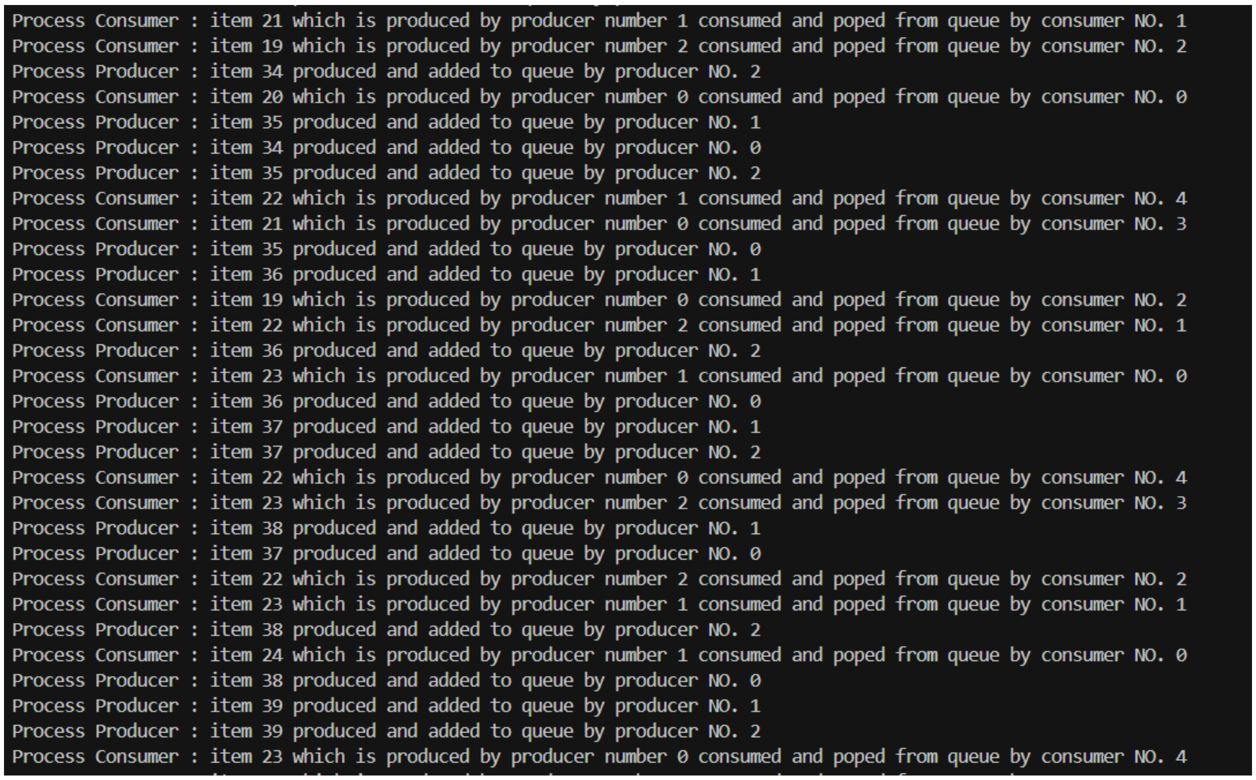
```
Process Consumer : item 1 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 2 produced and added to queue by producer NO. 0
Process Consumer : item 2 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 3 produced and added to queue by producer NO. 0
Process Producer : item 4 produced and added to queue by producer NO. 0
Process Consumer : item 3 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 5 produced and added to queue by producer NO. 0
Process Consumer : item 4 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 6 produced and added to queue by producer NO. 0
Process Consumer : item 5 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 7 produced and added to queue by producer NO. 0
Process Consumer : item 6 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 8 produced and added to queue by producer NO. 0
Process Consumer : item 7 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 9 produced and added to queue by producer NO. 0
Process Consumer : item 8 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 10 produced and added to queue by producer NO. 0
Process Consumer : item 9 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 11 produced and added to queue by producer NO. 0
Process Consumer : item 10 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 12 produced and added to queue by producer NO. 0
Process Consumer : item 11 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 13 produced and added to queue by producer NO. 0
Process Consumer : item 12 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 14 produced and added to queue by producer NO. 0
Process Consumer : item 13 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 15 produced and added to queue by producer NO. 0
```

Explanation for this part is similar to part A for 1 producer and 1 consumer. Here we have managed timing of producer and consumer so they are balanced. So we don't have stopping production or stopping consumption, and they work together in balance.

b: 3 Producer & 5 Consumer

Producer process: (i, 0.4, 0.6,) Consumer process: (i, 0.9, 1.1,)

A part of the output:



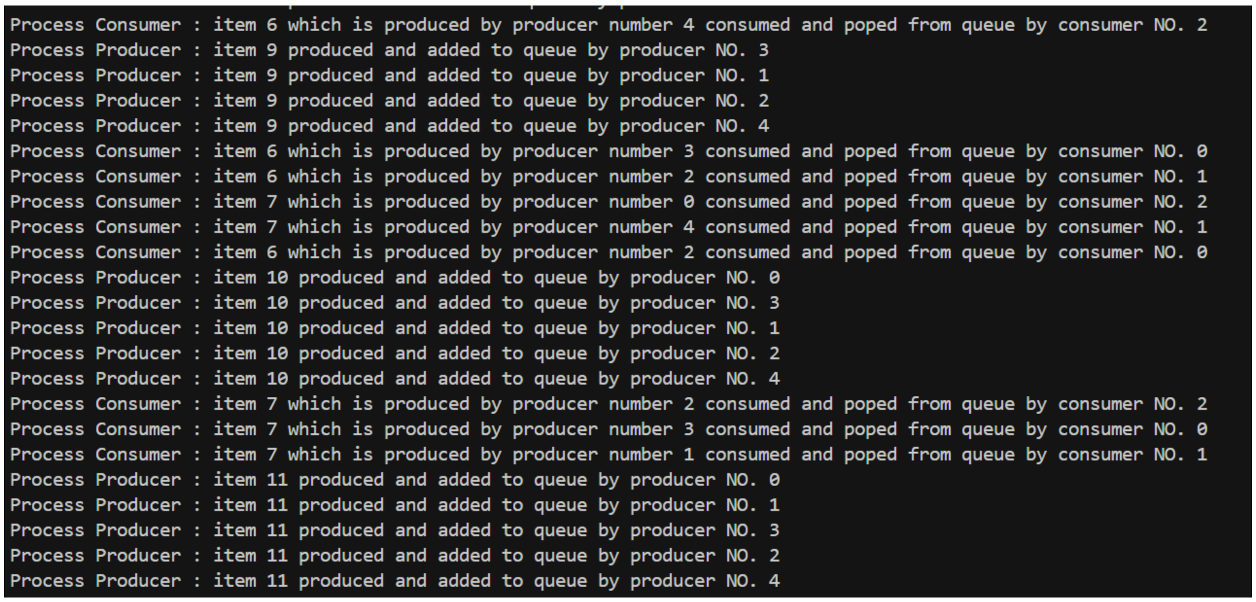
```
Process Consumer : item 21 which is produced by producer number 1 consumed and poped from queue by consumer NO. 1
Process Consumer : item 19 which is produced by producer number 2 consumed and poped from queue by consumer NO. 2
Process Producer : item 34 produced and added to queue by producer NO. 2
Process Consumer : item 20 which is produced by producer number 0 consumed and poped from queue by consumer NO. 0
Process Producer : item 35 produced and added to queue by producer NO. 1
Process Producer : item 34 produced and added to queue by producer NO. 0
Process Producer : item 35 produced and added to queue by producer NO. 2
Process Consumer : item 22 which is produced by producer number 1 consumed and poped from queue by consumer NO. 4
Process Consumer : item 21 which is produced by producer number 0 consumed and poped from queue by consumer NO. 3
Process Producer : item 35 produced and added to queue by producer NO. 0
Process Producer : item 36 produced and added to queue by producer NO. 1
Process Consumer : item 19 which is produced by producer number 0 consumed and poped from queue by consumer NO. 2
Process Consumer : item 22 which is produced by producer number 2 consumed and poped from queue by consumer NO. 1
Process Producer : item 36 produced and added to queue by producer NO. 2
Process Consumer : item 23 which is produced by producer number 1 consumed and poped from queue by consumer NO. 0
Process Producer : item 36 produced and added to queue by producer NO. 0
Process Producer : item 37 produced and added to queue by producer NO. 1
Process Producer : item 37 produced and added to queue by producer NO. 2
Process Consumer : item 22 which is produced by producer number 0 consumed and poped from queue by consumer NO. 4
Process Consumer : item 23 which is produced by producer number 2 consumed and poped from queue by consumer NO. 3
Process Producer : item 38 produced and added to queue by producer NO. 1
Process Producer : item 37 produced and added to queue by producer NO. 0
Process Consumer : item 22 which is produced by producer number 2 consumed and poped from queue by consumer NO. 2
Process Consumer : item 23 which is produced by producer number 1 consumed and poped from queue by consumer NO. 1
Process Producer : item 38 produced and added to queue by producer NO. 2
Process Consumer : item 24 which is produced by producer number 1 consumed and poped from queue by consumer NO. 0
Process Producer : item 38 produced and added to queue by producer NO. 0
Process Producer : item 39 produced and added to queue by producer NO. 1
Process Producer : item 39 produced and added to queue by producer NO. 2
Process Consumer : item 23 which is produced by producer number 0 consumed and poped from queue by consumer NO. 4
```

In this case we have same situation, only we should handle 3 producers & 5 consumers.

c: 5 Producer & 3 Consumer

Producer process: (i, 1.7, 1.9,)   Consumer process: (i, 1.2, 1.5,)

A part of the output:



```
Process Consumer : item 6 which is produced by producer number 4 consumed and poped from queue by consumer NO. 2
Process Producer : item 9 produced and added to queue by producer NO. 3
Process Producer : item 9 produced and added to queue by producer NO. 1
Process Producer : item 9 produced and added to queue by producer NO. 2
Process Producer : item 9 produced and added to queue by producer NO. 4
Process Consumer : item 6 which is produced by producer number 3 consumed and poped from queue by consumer NO. 0
Process Consumer : item 6 which is produced by producer number 2 consumed and poped from queue by consumer NO. 1
Process Consumer : item 7 which is produced by producer number 0 consumed and poped from queue by consumer NO. 2
Process Consumer : item 7 which is produced by producer number 4 consumed and poped from queue by consumer NO. 1
Process Consumer : item 6 which is produced by producer number 2 consumed and poped from queue by consumer NO. 0
Process Producer : item 10 produced and added to queue by producer NO. 0
Process Producer : item 10 produced and added to queue by producer NO. 3
Process Producer : item 10 produced and added to queue by producer NO. 1
Process Producer : item 10 produced and added to queue by producer NO. 2
Process Producer : item 10 produced and added to queue by producer NO. 4
Process Consumer : item 7 which is produced by producer number 2 consumed and poped from queue by consumer NO. 2
Process Consumer : item 7 which is produced by producer number 3 consumed and poped from queue by consumer NO. 0
Process Consumer : item 7 which is produced by producer number 1 consumed and poped from queue by consumer NO. 1
Process Producer : item 11 produced and added to queue by producer NO. 0
Process Producer : item 11 produced and added to queue by producer NO. 1
Process Producer : item 11 produced and added to queue by producer NO. 3
Process Producer : item 11 produced and added to queue by producer NO. 2
Process Producer : item 11 produced and added to queue by producer NO. 4
```

In this case we have same situation, only we should handle 5 producers & 3 consumers.

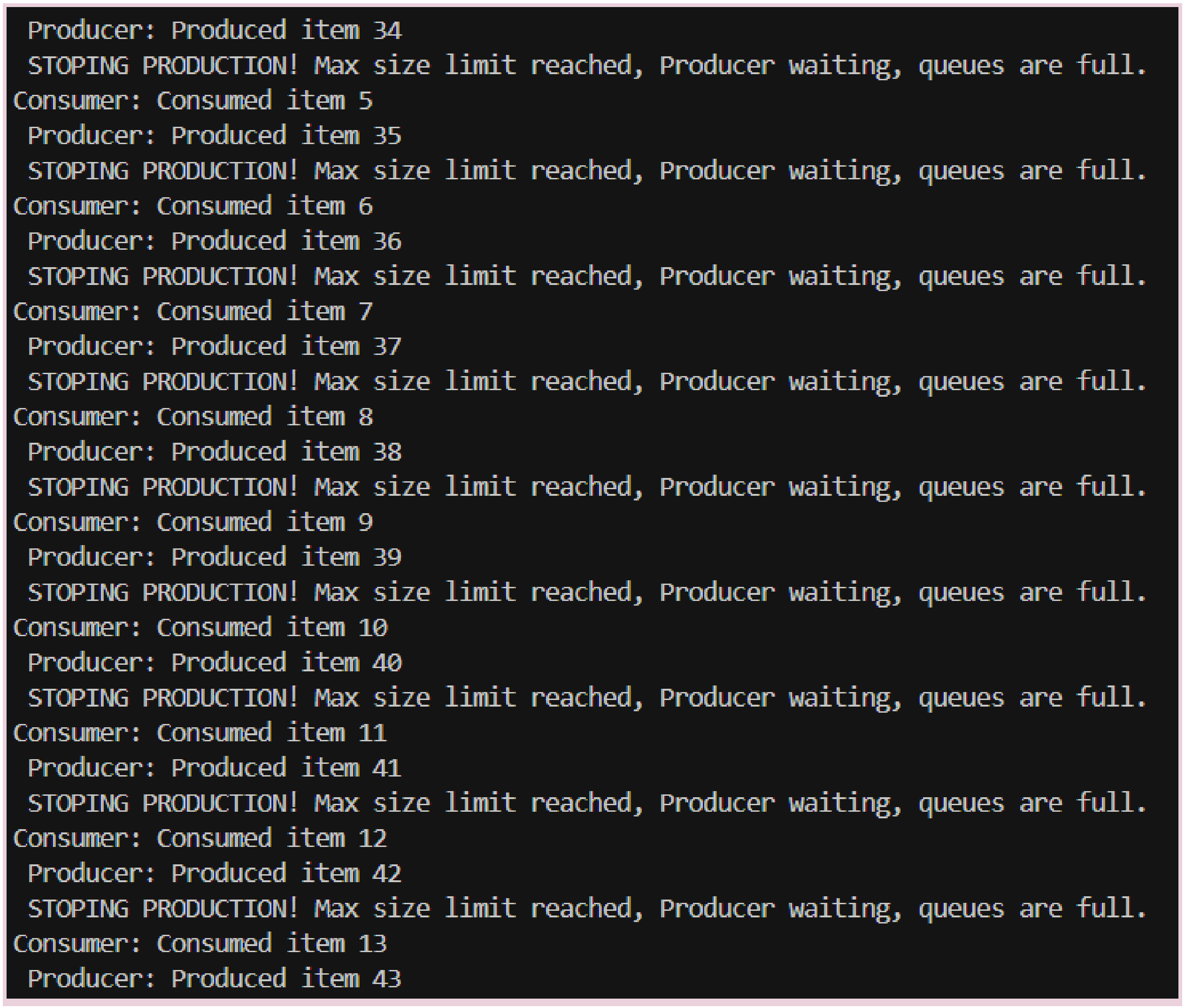
★ 2) Transferring information between consumer and producer using pipe:

➔ A. Stopping Production

1 Producer & 1 Consumer

Producer process: (0.1, 0.3,)   Consumer process: (1.1, 2.0,)

A part of the output:



```
 Producer: Produced item 34
 STOPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Consumer: Consumed item 5
 Producer: Produced item 35
 STOPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Consumer: Consumed item 6
 Producer: Produced item 36
 STOPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Consumer: Consumed item 7
 Producer: Produced item 37
 STOPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Consumer: Consumed item 8
 Producer: Produced item 38
 STOPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Consumer: Consumed item 9
 Producer: Produced item 39
 STOPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Consumer: Consumed item 10
 Producer: Produced item 40
 STOPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Consumer: Consumed item 11
 Producer: Produced item 41
 STOPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Consumer: Consumed item 12
 Producer: Produced item 42
 STOPING PRODUCTION! Max size limit reached, Producer waiting, queues are full.
Consumer: Consumed item 13
 Producer: Produced item 43
```

First, we define *MAX_SIZE* to determine max memory size. In producer func, we have two pipes: pipe_1 & pipe_2. pipe_1 sends data from producer to consumer, while other does opposite.
waiting variable keeps track of num of items currently waiting, and we store these items in a queue.

counter indicates which item is being produced and increase it each time.
stop_event is an instance of multiprocessing.Event(). It is set when working time (20 seconds) is over. At start of each loop in producer and consumer func, we check with *is_set*. If it is set, program ends.
We use *.poll()* for all inputs to make *input_pipes* non-blocking. In a blocking situation, program would wait until a message arrives. With *.poll()*, it checks if a message has been sent before blocking to receive it.
In producer, if a message is received from consumer, we decrease *waiting* by 1 because one item has been consumed. We also pop first item from queue.

In producer, we check if *waiting* equals *MAX_SIZE*. If it does, we print message: *"STOPPING PRODUCTION! Max size limit reached, Producer waiting, queues are full."* It means there is not enough space for producing new items. Producer then waits for a message to be received. If it's not full, we produce *item* equal to *counter*, send it, and increase *waiting* by 1. We also increase the *counter* and add a sleep time for production.
In consumer func, logic is similar, but we don't have a counter since we aren't producing anything. When we use *items.put()*, we need to specify what item is so we know its num when consuming it later.
In producer, we compared *waiting* with *MAX_SIZE*, but in consumer, we check: if waiting != 0 to see if it is empty. If it is 0, we say it is empty, and consumer will wait.
At start of main, we create pipes and the *stop_event*. We then start producer and consumer using multiprocessing. After starting, we define a 20 second sleep. After these 20 seconds, we set *stop_event* to stop *while* in producer and consumer funcs. We close pipes and join processes.

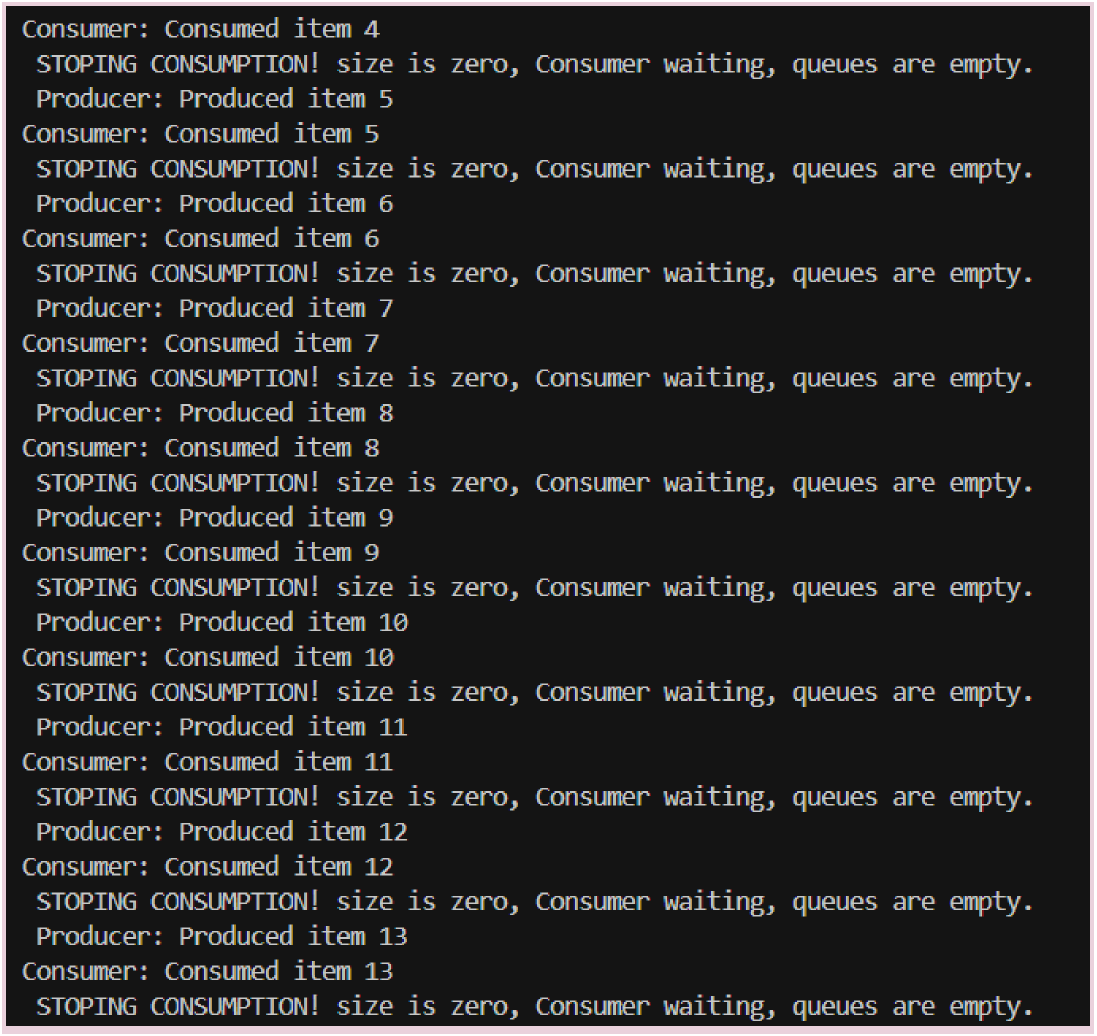
In this case, producer's speed is faster than consumer's. We have stop production because if we produce too much, we will run into problem of not having enough space.

➔ B. Stopping Consumption

1 Producer & 1 Consumer

Producer process: (1.1, 2.0,) Consumer process: (0.1, 0.3,)

A part of the output:



```
Consumer: Consumed item 4
 STOPING CONSUMPTION! size is zero, Consumer waiting, queues are empty.
 Producer: Produced item 5
Consumer: Consumed item 5
 STOPING CONSUMPTION! size is zero, Consumer waiting, queues are empty.
 Producer: Produced item 6
Consumer: Consumed item 6
 STOPING CONSUMPTION! size is zero, Consumer waiting, queues are empty.
 Producer: Produced item 7
Consumer: Consumed item 7
 STOPING CONSUMPTION! size is zero, Consumer waiting, queues are empty.
 Producer: Produced item 8
Consumer: Consumed item 8
 STOPING CONSUMPTION! size is zero, Consumer waiting, queues are empty.
 Producer: Produced item 9
Consumer: Consumed item 9
 STOPING CONSUMPTION! size is zero, Consumer waiting, queues are empty.
 Producer: Produced item 10
Consumer: Consumed item 10
 STOPING CONSUMPTION! size is zero, Consumer waiting, queues are empty.
 Producer: Produced item 11
Consumer: Consumed item 11
 STOPING CONSUMPTION! size is zero, Consumer waiting, queues are empty.
 Producer: Produced item 12
Consumer: Consumed item 12
 STOPING CONSUMPTION! size is zero, Consumer waiting, queues are empty.
 Producer: Produced item 13
Consumer: Consumed item 13
 STOPING CONSUMPTION! size is zero, Consumer waiting, queues are empty.
```

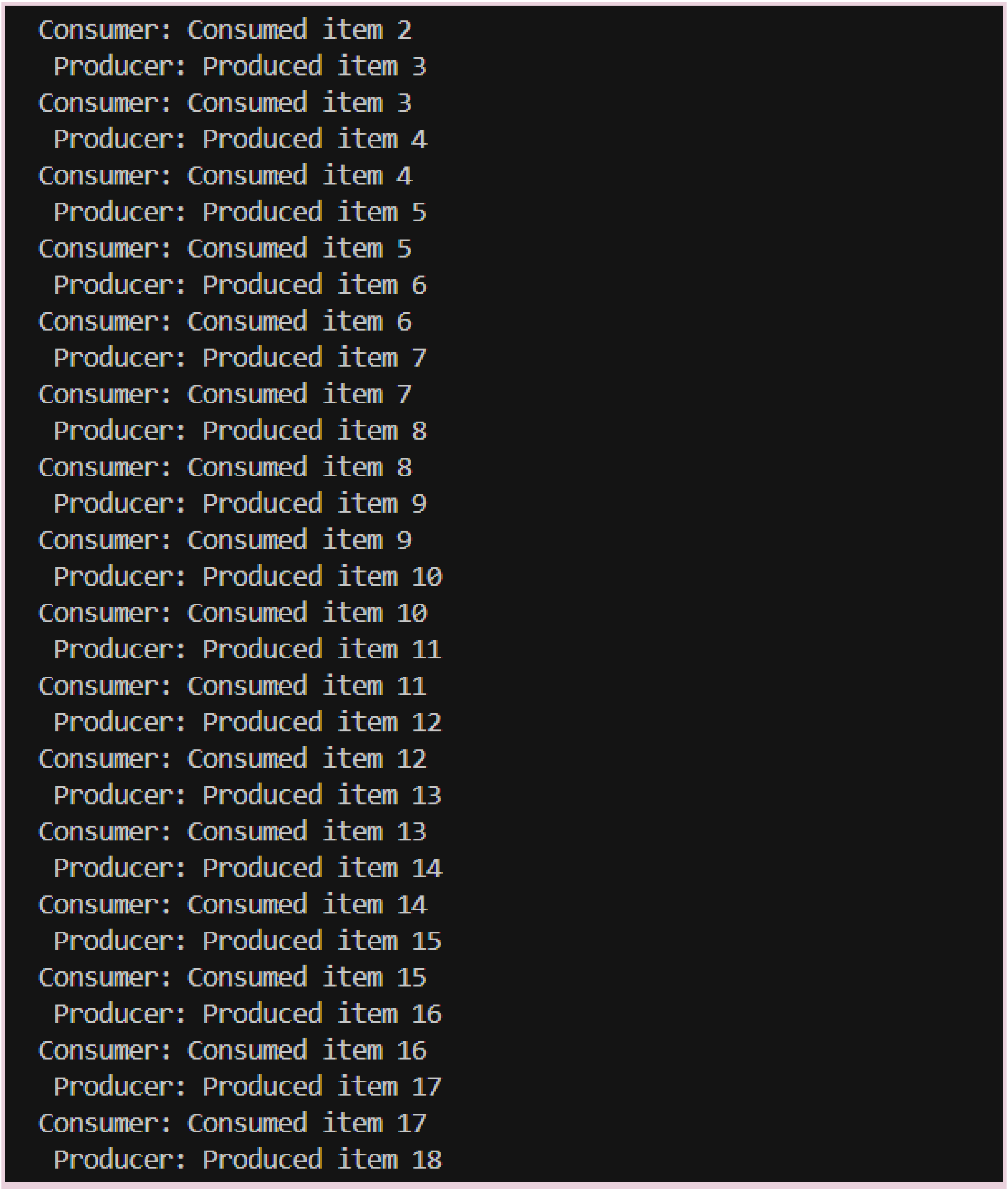
Explanation for this part is similar to A for 1 producer and 1 consumer. Difference here is the consumption speed is faster than production speed. Because of this, we face problem of having nothing to consume.
STOPING CONSUMPTION! size is zero, consumer waiting, queues are empty.

➔ C. Concurrent Production & Consumption

1 Producer & 1 Consumer

Producer process: (0.9, 1.0,) Consumer process: (0.9, 1.1,)

A part of the output:



```
Consumer: Consumed item 2
 Producer: Produced item 3
Consumer: Consumed item 3
 Producer: Produced item 4
Consumer: Consumed item 4
 Producer: Produced item 5
Consumer: Consumed item 5
 Producer: Produced item 6
Consumer: Consumed item 6
 Producer: Produced item 7
Consumer: Consumed item 7
 Producer: Produced item 8
Consumer: Consumed item 8
 Producer: Produced item 9
Consumer: Consumed item 9
 Producer: Produced item 10
Consumer: Consumed item 10
 Producer: Produced item 11
Consumer: Consumed item 11
 Producer: Produced item 12
Consumer: Consumed item 12
 Producer: Produced item 13
Consumer: Consumed item 13
 Producer: Produced item 14
Consumer: Consumed item 14
 Producer: Produced item 15
Consumer: Consumed item 15
 Producer: Produced item 16
Consumer: Consumed item 16
 Producer: Produced item 17
Consumer: Consumed item 17
 Producer: Produced item 18
```

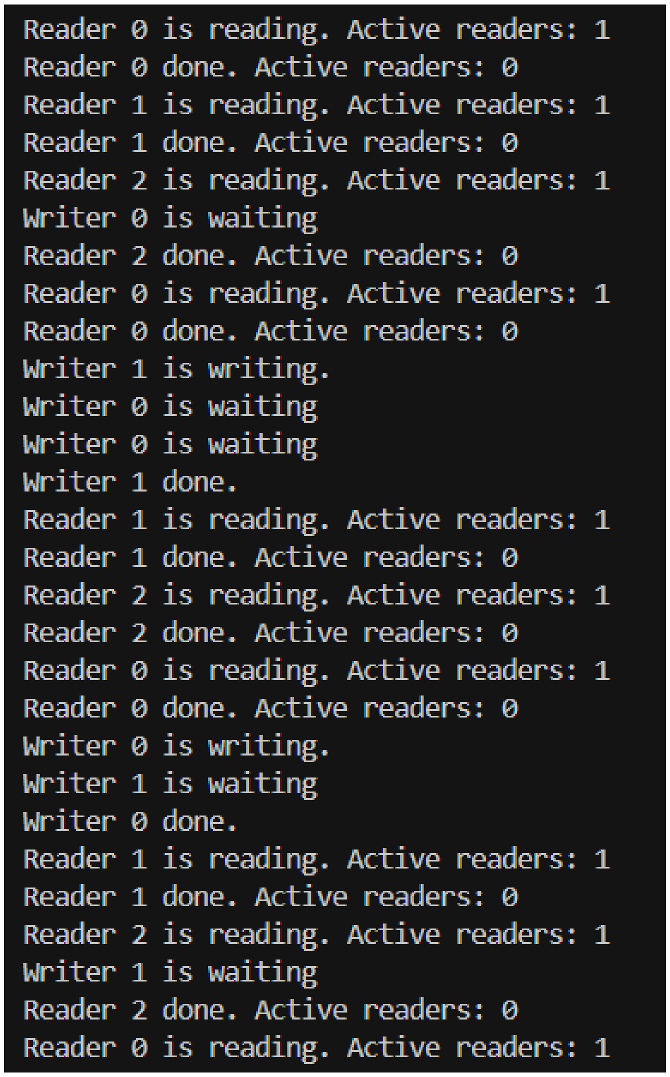
Explanation for this part is similar to part A for 1 producer and 1 consumer. Here we have managed timing of producer and consumer so they are balanced. So we don't have stopping production or stopping consumption, and they work together in balance.

II. Readers–Writers Problem

The other problem in process synchronization is the Reader-Writer Problem. In this section, we will examine two scenarios using multiprocessing: one where readers have priority and another where writers have priority.

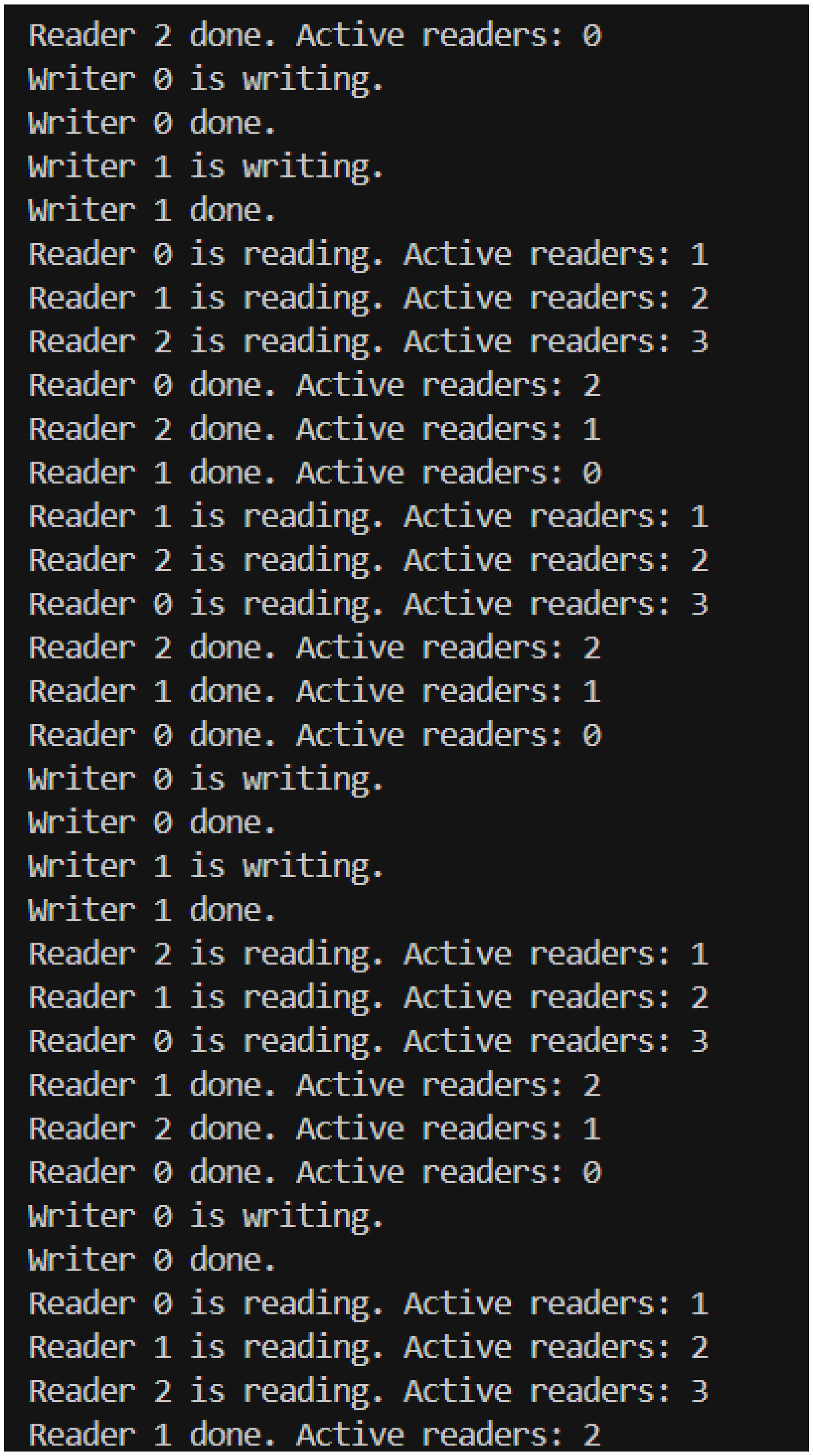
- ★ A. Readers have priority:
  Readers are given preference over writers. Till readers are reading, writers will have to wait. Writers can access resource only when no reader is accessing it.

- *One Reader*



```
Reader 0 is reading. Active readers: 1
Reader 0 done. Active readers: 0
Reader 1 is reading. Active readers: 1
Reader 1 done. Active readers: 0
Reader 2 is reading. Active readers: 1
Writer 0 is waiting
Reader 2 done. Active readers: 0
Reader 0 is reading. Active readers: 1
Reader 0 done. Active readers: 0
Writer 1 is writing.
Writer 0 is waiting
Writer 0 is waiting
Writer 1 done.
Reader 1 is reading. Active readers: 1
Reader 1 done. Active readers: 0
Reader 2 is reading. Active readers: 1
Reader 2 done. Active readers: 0
Reader 0 is reading. Active readers: 1
Reader 0 done. Active readers: 0
Writer 0 is writing.
Writer 1 is waiting
Writer 0 done.
Reader 1 is reading. Active readers: 1
Reader 1 done. Active readers: 0
Reader 2 is reading. Active readers: 1
Writer 1 is waiting
Reader 2 done. Active readers: 0
Reader 0 is reading. Active readers: 1
```

- *Many Readers*



```
Reader 2 done. Active readers: 0
Writer 0 is writing.
Writer 0 done.
Writer 1 is writing.
Writer 1 done.
Reader 0 is reading. Active readers: 1
Reader 1 is reading. Active readers: 2
Reader 2 is reading. Active readers: 3
Reader 0 done. Active readers: 2
Reader 2 done. Active readers: 1
Reader 1 done. Active readers: 0
Reader 1 is reading. Active readers: 1
Reader 2 is reading. Active readers: 2
Reader 0 is reading. Active readers: 3
Reader 2 done. Active readers: 2
Reader 1 done. Active readers: 1
Reader 0 done. Active readers: 0
Writer 0 is writing.
Writer 0 done.
Writer 1 is writing.
Writer 1 done.
Reader 2 is reading. Active readers: 1
Reader 1 is reading. Active readers: 2
Reader 0 is reading. Active readers: 3
Reader 1 done. Active readers: 2
Reader 2 done. Active readers: 1
Reader 0 done. Active readers: 0
Writer 0 is writing.
Writer 0 done.
Reader 0 is reading. Active readers: 1
Reader 1 is reading. Active readers: 2
Reader 2 is reading. Active readers: 3
Reader 1 done. Active readers: 2
```

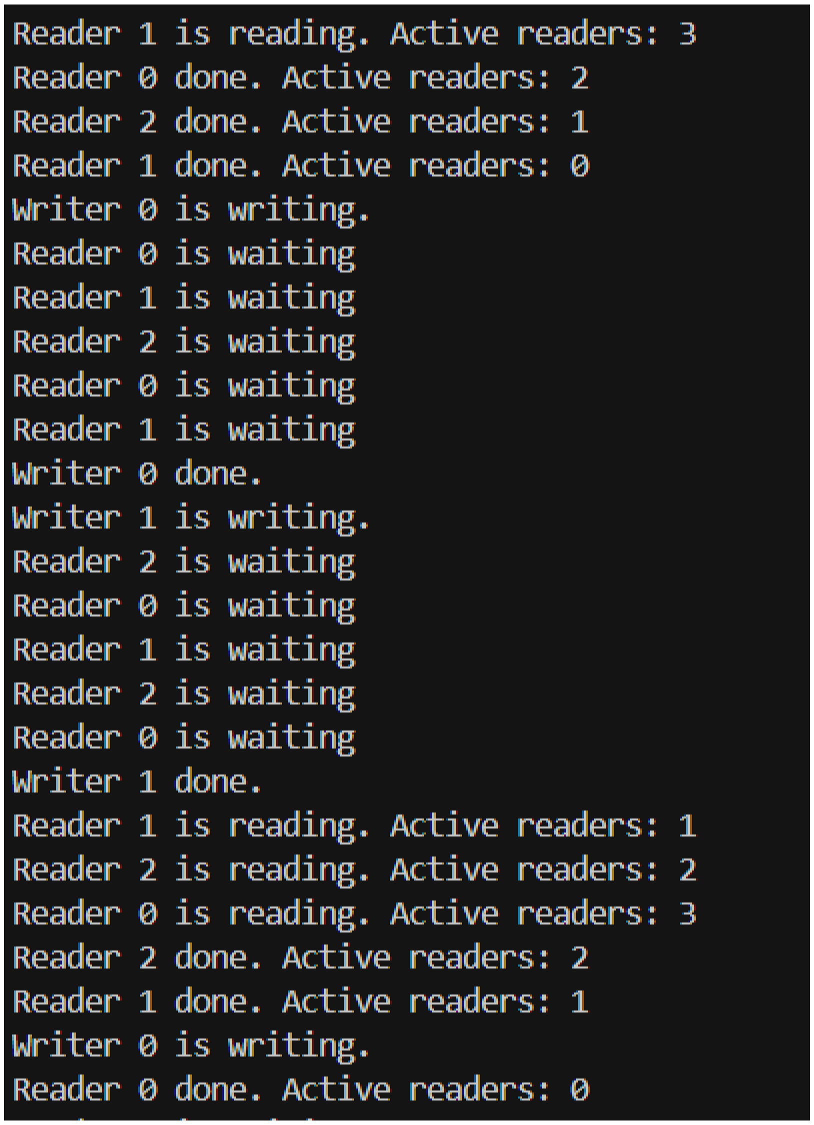
First, we define locks. *read_count* keeps track of how many readers are reading, and *mutex* is for readers. *rw_mutex* is for both readers and writers, and *waiting_readers* is a value that shows how many readers are waiting.

We have a while true loop that acquires mutex and increments *count*. If *count* is one, it means this is first reader. So, it blocks writers and resets waiting count, indicating that it is currently reading.
Difference btw <u>many readers</u> and <u>one reader</u> is that in <u>many </u>(line 17 of its code), we have *mutex.release()*, allowing other readers to enter. So, in <u>many </u>case, multiple readers can work together, while in <u>one reader</u> case, only one reader is allowed.
If count is zero, both reader and writer mutexes are released, allowing writers to proceed if they are waiting. Finally, mutex is released.
In writer function, if *waiting_readers* is zero, it can enter. If not, it waits for 0.005 seconds and then tries again.
Then, *rw_mutex* is acquired, writer performs its task, and finally, it is released.

★ B. Writers have priority:

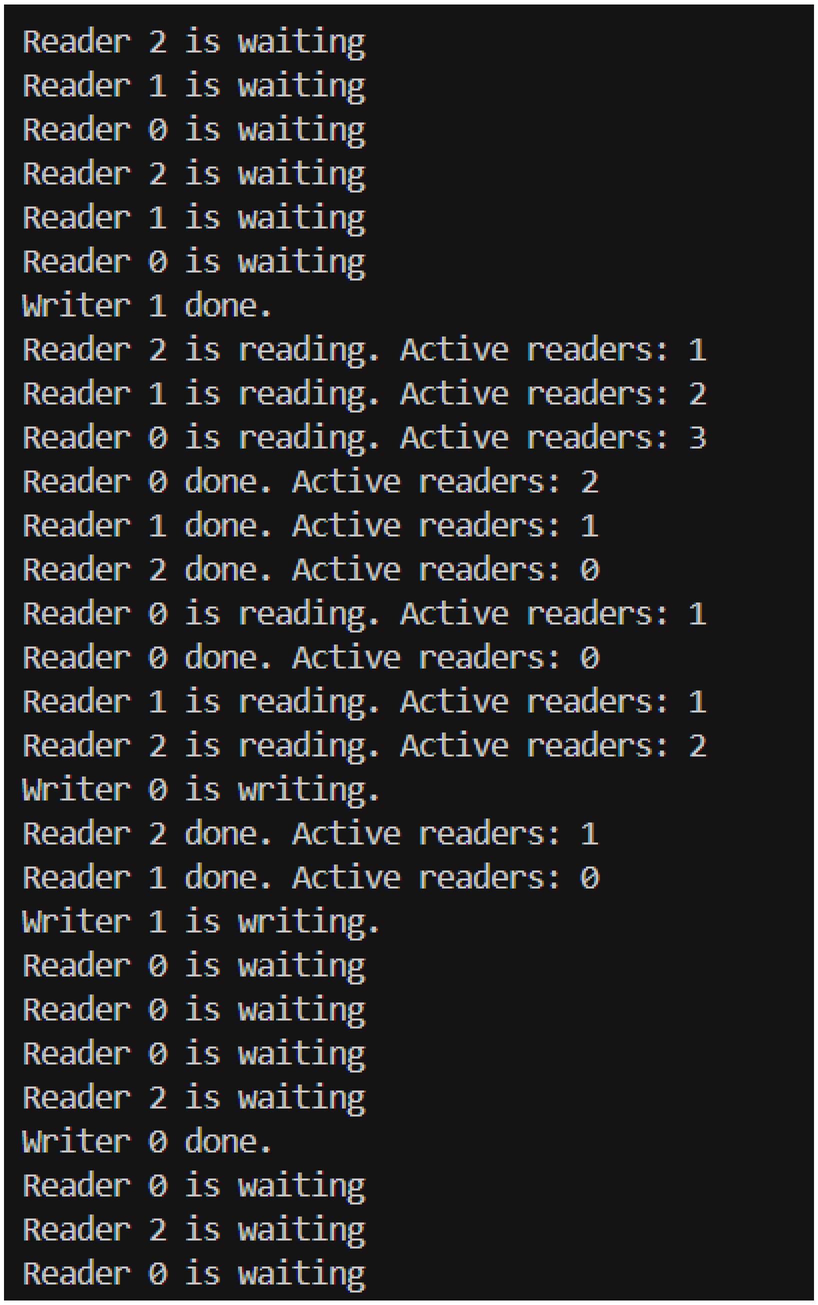
If a writer requests access to its critical section, it should not have to wait for all readers to finish. It should be able to proceed with its work immediately. Preference is given to writers. After arrival, writers can go ahead with their operations; though perhaps there are readers currently accessing resource.

- *One Writer*



```
Reader 1 is reading. Active readers: 3
Reader 0 done. Active readers: 2
Reader 2 done. Active readers: 1
Reader 1 done. Active readers: 0
Writer 0 is writing.
Reader 0 is waiting
Reader 1 is waiting
Reader 2 is waiting
Reader 0 is waiting
Reader 1 is waiting
Writer 0 done.
Writer 1 is writing.
Reader 2 is waiting
Reader 0 is waiting
Reader 1 is waiting
Reader 2 is waiting
Reader 0 is waiting
Writer 1 done.
Reader 1 is reading. Active readers: 1
Reader 2 is reading. Active readers: 2
Reader 0 is reading. Active readers: 3
Reader 2 done. Active readers: 2
Reader 1 done. Active readers: 1
Writer 0 is writing.
Reader 0 done. Active readers: 0
```

- *Many Writers*



```
Reader 2 is waiting
Reader 1 is waiting
Reader 0 is waiting
Reader 2 is waiting
Reader 1 is waiting
Reader 0 is waiting
Writer 1 done.
Reader 2 is reading. Active readers: 1
Reader 1 is reading. Active readers: 2
Reader 0 is reading. Active readers: 3
Reader 0 done. Active readers: 2
Reader 1 done. Active readers: 1
Reader 2 done. Active readers: 0
Reader 0 is reading. Active readers: 1
Reader 0 done. Active readers: 0
Reader 1 is reading. Active readers: 1
Reader 2 is reading. Active readers: 2
Writer 0 is writing.
Reader 2 done. Active readers: 1
Reader 1 done. Active readers: 0
Writer 1 is writing.
Reader 0 is waiting
Reader 0 is waiting
Reader 0 is waiting
Reader 2 is waiting
Writer 0 done.
Reader 0 is waiting
Reader 2 is waiting
Reader 0 is waiting
```

In this part we implemented Readers-Writers problem with writers priority. In this code we defined 2 separated mutexes, 1 in only for readers and the other one only for writers. We also defined a value for writers count and the other for readers count. In readers and writers functions we didn't use shared mutexes and one mutex is only lucking readers and the other one only luck writers.
The only limitation is on readers and when reader are trying to enter to their critical sections no writer must be inside. The writer count value checks this condition. Only if writers count is zero reader are allowed to enter the critical

section. We also handled 1 writer and many writers being in their critical sections using the writer mutex and for the many writers situation after increasing the writer count, writer mutex in released so other writers can enter their critical section. But in the 1 writer situation the mutex is not released until writer is done.