The Producer-Consumer Mechanism

Task

Run producer-consumer.py in the provided Codio workspace (**Producer-Consumer Mechanism**), where the queue data structure is used.

A copy of the code is below.

```
# code source: https://techmonger.github.io/55/producer-consumer-python/

from threading import Thread
from queue import Queue

q = Queue()
final_results = []

def producer():
    for i in range(100):
        q.put(i)


def consumer():
    while True:
        number = q.get()
        result = (number, number**2)
        final_results.append(result)
        q.task_done()


for i in range(5):
    t = Thread(target=consumer)
    t.daemon = True
    t.start()

producer()

q.join()

print (final_results)
```

Output

```
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9
, 81), (10, 100), (11, 121), (12, 144), (13, 169), (14, 196), (15, 225), (16, 25
6), (17, 289), (18, 324), (19, 361), (20, 400), (21, 441), (22, 484), (23, 529),
 (24, 576), (25, 625), (26, 676), (27, 729), (28, 784), (29, 841), (30, 900), (3
1, 961), (32, 1024), (33, 1089), (34, 1156), (35, 1225), (36, 1296), (37, 1369),
 (38, 1444), (39, 1521), (40, 1600), (41, 1681), (42, 1764), (43, 1849), (44, 19
36), (45, 2025), (46, 2116), (47, 2209), (48, 2304), (49, 2401), (50, 2500), (51
, 2601), (52, 2704), (53, 2809), (54, 2916), (55, 3025), (56, 3136), (57, 3249),
 (58, 3364), (59, 3481), (60, 3600), (61, 3721), (62, 3844), (63, 3969), (64, 40
96), (65, 4225), (66, 4356), (67, 4489), (68, 4624), (69, 4761), (70, 4900), (71
, 5041), (72, 5184), (73, 5329), (74, 5476), (75, 5625), (76, 5776), (77, 5929),
 (78, 6084), (79, 6241), (80, 6400), (81, 6561), (82, 6724), (83, 6889), (84, 70
56), (85, 7225), (86, 7396), (87, 7569), (88, 7744), (89, 7921), (90, 8100), (91
, 8281), (92, 8464), (93, 8649), (94, 8836), (95, 9025), (96, 9216), (97, 9409),
 (98, 9604), (99, 9801)]
```

J. Irvine – Secure Software Development

Answer the following questions:

1. How is the queue data structure used to achieve the purpose of the code?

This purpose of this code is to demonstrate how multiple threads can be used to process items from a queue at the same time. The producer thread adds items to the queue, while the consumer threads remove items from the queue and process them.

The queue data structure stores items in a First In First Out (FIFO) manner as opposed to Last In First Out (LIFO). In the code, the queue is created using the Queue() function from the queue module. The producer() function adds items to the queue using the put() method. The consumer() function removes items from the queue using the get() method. The final_results list is used to store the results of each item processed by the consumer function.

2. What is the purpose of q.put(I)?

The q.put(i) is used to add an item to the queue. In this code, the producer thread adds items to the queue using the put() method. The i variable is a loop variable that takes on values from 0 to 99. Each value of i is added to the queue using the put() method.

3. What is achieved by q.get()?

The q.get() method is used to remove an item from the queue. In this code, the consumer threads remove items from the queue using the get() method. The number variable is assigned the value of the next item in the queue. The result variable is a tuple that contains two values: the original number and its square. The final_results list is used to store the results of each item processed by the consumer function.

4. What functionality is provided by q.join()?

The q.join() method is used to block until all items in the queue have been processed. In this code, the producer thread adds items to the queue using the put() method. The consumer threads remove items from the queue using the get() method and process them. The q.join() method is used to block until all items in the queue have been processed by the consumer threads. This ensures that all items have been processed before the program exits.

5. Extend this producer-consumer code to make the producer-consumer scenario available in a secure way.

Extended Code

```
# Lock function added to so only 1 thread is accesses at a time.
from threading import Thread, Lock
from queue import Queue

q = Queue()
final_results = []

lock = Lock()

# The producer funtion needs to get the lock before addingg an item.
def producer():
    for i in range(100):
        lock.acquire()
        q.put(i)
        lock.release()

# The producer funtion needs to get the lock before addingg an item.
# It will chek to ee if the ques is empty, items wiil be emptied using q.get()
def consumer():
    while True:
        lock.acquire()
        if not q.empty():
            number = q.get()
            result = (number, number**2)
            final_results.append(result)
            q.task_done()
        lock.release()

for i in range(5):
    t = Thread(target=consumer)
    t.daemon = True
    t.start()

producer()

q.join()

print(final_results)
```

Output

```
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9,
81), (10, 100), (11, 121), (12, 144), (13, 169), (14, 196), (15, 225), (16, 256),
(17, 289), (18, 324), (19, 361), (20, 400), (21, 441), (22, 484), (23, 529), (24,
576), (25, 625), (26, 676), (27, 729), (28, 784), (29, 841), (30, 900), (31, 961)
, (32, 1024), (33, 1089), (34, 1156), (35, 1225), (36, 1296), (37, 1369), (38, 14
44), (39, 1521), (40, 1600), (41, 1681), (42, 1764), (43, 1849), (44, 1936), (45,
2025), (46, 2116), (47, 2209), (48, 2304), (49, 2401), (50, 2500), (51, 2601), (5
2, 2704), (53, 2809), (54, 2916), (55, 3025), (56, 3136), (57, 3249), (58, 3364),
(59, 3481), (60, 3600), (61, 3721), (62, 3844), (63, 3969), (64, 4096), (65, 4225
), (66, 4356), (67, 4489), (68, 4624), (69, 4761), (70, 4900), (71, 5041), (72, 5
184), (73, 5329), (74, 5476), (75, 5625), (76, 5776), (77, 5929), (78, 6084), (79
, 6241), (80, 6400), (81, 6561), (82, 6724), (83, 6889), (84, 7056), (85, 7225),
(86, 7396), (87, 7569), (88, 7744), (89, 7921), (90, 8100), (91, 8281), (92, 8464
), (93, 8649), (94, 8836), (95, 9025), (96, 9216), (97, 9409), (98, 9604), (99, 9
801)]
```

6. What technique(s) would be appropriate to apply?

To make the producer-consumer scenario available in a secure way, I have added the locks function, so only one thread accesses the queue at a time. The lock object can be used to synchronise access to the queue between the producer and consumer threads.

Remember to record your thoughts and answers in your e-portfolio.

Learning Outcomes

- Identify and critically analyse operating system risks and issues, and identify appropriate methodologies, tools and techniques to solve them.
- Critically analyse and evaluate solutions produced.