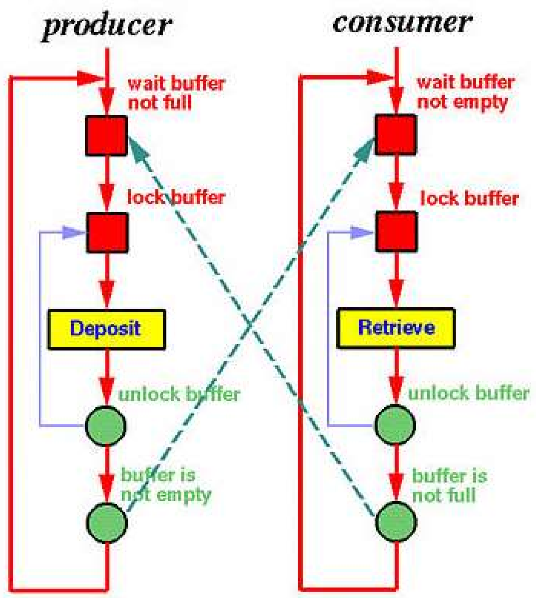
Study

- The Producer-Consumer Problem is also known as the bounded buffer problem
- For example, models an application producing a listing that must be consumed by a printer process; or perhaps a keyboard handler producing a line of data that will be consumed by an application
- Items are places in a buffer when produced, therefore:
- 1.) The consumer should wait if there isn't an item to consume
- 2.) The Producer shouldn't overwrite an item in the buffer:



- Synchronisation is required because:
- 1.) The Producer should not replace values in the buffer if the Consumer has not processed it yet
- 2.) The Consumer should not consume the same value twice

- Example Consumer-Producer code beneath:

```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, 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, 1444)
, (39, 1521), (40, 1600), (41, 1681), (42, 1764), (43, 1849), (44, 1936), (45, 202
5), (46, 2116), (47, 2209), (48, 2304), (49, 2401), (50, 2500), (51, 2601), (52, 2
704), (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, 5184), (
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, 864
9), (94, 8836), (95, 9025), (96, 9216), (97, 9409), (98, 9604), (99, 9801)]
```

- https://docs.python.org/3/library/queue.html - A Synchronised Queue Class
- The queue module implements multi-producer, multi-consumer queues, which is useful in threading a program when information must be exchanged safely between multiple threads -> It implements all the required locking semantics
- The data structure from queue module uses internal locks to temporarily block competing threads
- The Queue(*maxsize* = *0*) method produces a typical FIFO queue data structure as seen above
- Queue.put(item) puts an item into the queue structure
- Queue.get() removes and returns an item from the queue
- Queue.task_done() indicates that a formerly enqueued task is completed -> Used by queue consumer threads -> For each .get() used, a subsequent call to task_done() tells the queue that the processing on the task is complete
- Queue.join() blocks until all items in the queue have been gotten and processed

- https://docs.python.org/3/library/threading.html - Thread-based Parallelism
- This modules constructs higher-level threading interfaces on top of the lower level _thread module

- In Cpython due to the Global Interpreter Lock, only one thread can execute Python code at once -> If one wants to make better use of the computational resources of multicore machines, it is advised to use the multiprocessing or concurrent.futures.ProcessPoolExecutor modules -> Threading module is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously
- The Thread class represents the an activity that is run in a separate thread of control -> To specify an activity: 1.) Pass a callable object to the constructor 2.) Or by overriding the .run() method in a subclass
- Once the Thread's activity is created, it is considered alive
- Other threads can call the thread's join() method, which blocks the calling thread until the thread whose join() method which is called is terminated
- A thread can be flagged as a daemon thread, where the entire Python program exists when only daemon threads are left
- There is a main thread object, which corresponds to the initial thread of control in the Python program, and not a daemon thread
- In multitasking computer operating systems a daemon is a computer program that runs as a background process, rather than under the direct control of an interactive user
- threading.Thread(..target=none...) -> Target is the callable object to be invoked
- .start() method starts the threads activity
- .daemon is a boolean value indicating whether the thread is a daemon (True) or not (False)

<u>Questions</u>

1. How is the queue data structure used to achieve the purpose of the code?
2. What is the purpose of q.put(I)?
3. What is achieved by q.get()?
4. What functionality is provided by q.join()?
5. Extend this producer-consumer code to make the producer-consumer scenario available in a secure way. What technique(s) would be appropriate to apply?

<u>Answers</u>

1. The queue data structure is used as a buffer
2. The q.put(i) method adds an element to the queue
3. q.get() removes the first item in the queue and returns it
4. q.join() blocks the calling thread until the called thread has terminated
5. By making the consumer while loop have a terminating condition