Producer-Consumer Using queue.Queue (threads)

Concept Overview: Producer-Consumer with Queue

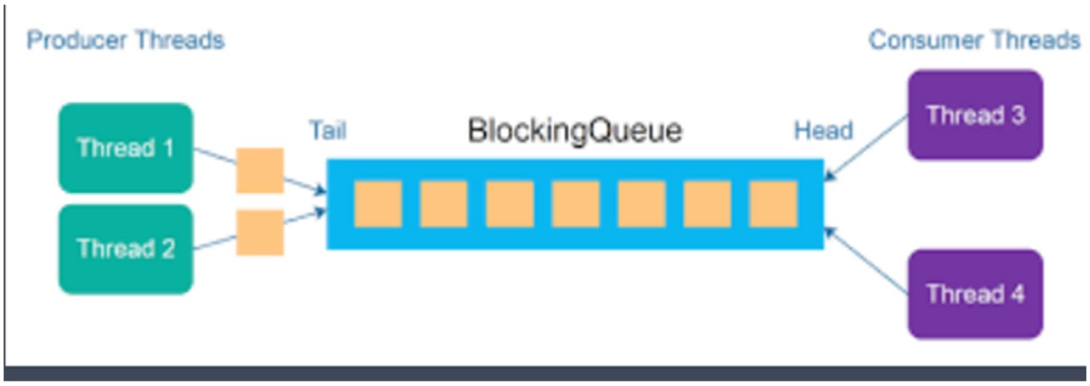
- **Producer**: generates data and places it into the queue.
- **Consumer**: takes from the queue and processes it.
- **Queue**: a thread-safe buffer holding items until the consumer is ready.

Step-by-Step Implementation

1. **Import Required Modules**
   threading, queue, time, random — for concurrency, safe buffer, timing, randomness.
2. **Create the Queue**
   q = queue.Queue() — makes a FIFO, thread-safe queue.
3. **Define Producer Function**
   Loops count times, produces random items, q.put(item), sleeps to simulate delay. After loop, q.put(None) as sentinel.
4. **Define Consumer Function**
   Loops forever, item = q.get() (blocks). If item is None, break (stop). Else process item, q.task_done(). Finally, mark sentinel done.
5. **Start Threads**
   Create threads for producer and consumer, start them.
   Use join() on producer, then q.join() to wait all tasks done, then join() consumer.

Advantages & Limitations

- **Pros**: simple, built-in, thread-safe, good for I/O tasks.
- **Cons**: limited to same process (threads), Python's GIL may hinder CPU-bound work.




```
import threading
import queue
import time
import random

def producer(q, count=10):
    """
    Producer: generates `count` items and puts them into queue q,
    then signals termination by putting a sentinel (None).
    """
    for i in range(count):
        item = random.randint(1, 100)
        print(f"Producer: Produced item {item}")
        q.put(item)
        time.sleep(random.uniform(0.5, 1.5))
    # Signal end
    q.put(None)
    print("Producer: Finished producing.")

def consumer(q):
```

```
    """
    Consumer: takes items from q and processes them.
    Exits when it receives the sentinel (None).
    """
    while True:
        item = q.get()
        if item is None:
            print("Consumer: Received sentinel, stopping.")
            break
        print(f"Consumer: Consumed item {item}")
        time.sleep(random.uniform(1, 2))
        q.task_done()
    # mark sentinel done
    q.task_done()
    print("Consumer: Exiting.")

def main():
    q = queue.Queue()

    t_consumer = threading.Thread(target=consumer, args=(q,))
    t_producer = threading.Thread(target=producer, args=(q,))

    t_consumer.start()
    t_producer.start()

    t_producer.join()
    q.join()
    t_consumer.join()
```

```
    print("Main: All done.")


if __name__ == "__main__":
main()
```

Concept Overview: Producer-Consumer with multiprocessing.Queue

- **Producer (process)**: runs in its own process, generates data, and puts into a shared queue.
- **Consumer (process)**: runs separately, gets data from queue, processes it, stops when sentinel is received.
- **multiprocessing.Queue**: a queue safe for inter-process communication (IPC).

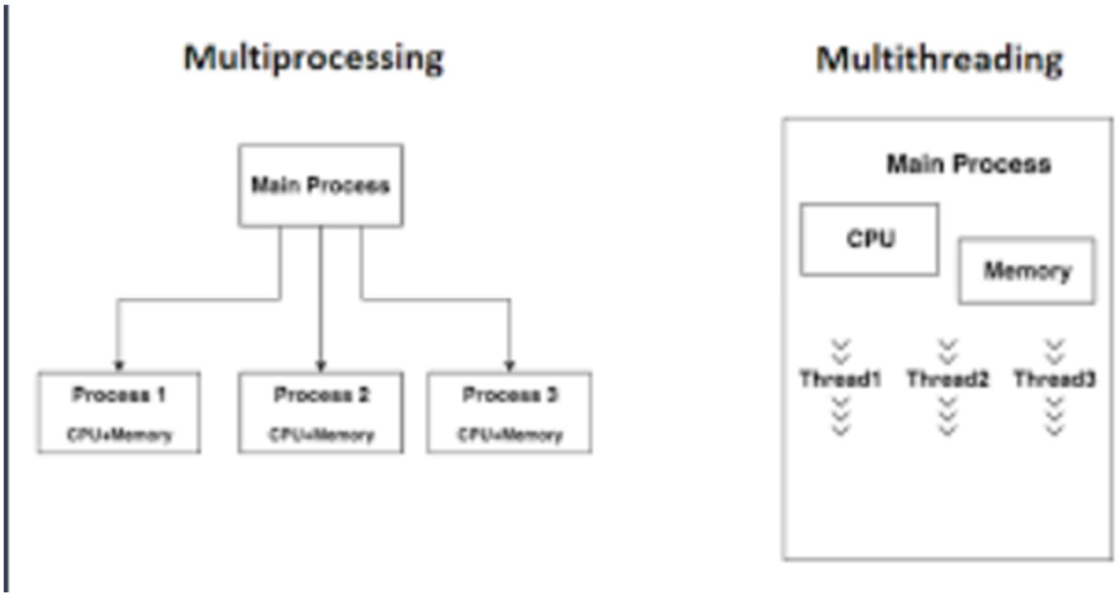
Step-by-Step Implementation

1. **Import Modules**
   from multiprocessing import Process, Queue plus time, random for delay and randomness.
2. **Define Producer Function (producer_mp)**
   Loop count times, generate a random item, q.put(item), sleep. After the loop, q.put(None) as sentinel to signal end.
3. **Define Consumer Function (consumer_mp)**
   Infinite loop: item = q.get(). If item is None, break. Otherwise, print/consume and sleep.
4. **Main Routine**
   - Create a Queue()
   - Define Process objects for producer & consumer
   - Start consumer first (so it's ready to receive), then producer
   - join() both to wait for completion

- Final print statement

Advantages & Limitations

- **Pros**
  - Works across processes (not just threads)
  - Bypasses Python's GIL for CPU-bound work
  - Good for more heavy tasks or workload isolation
- **Cons**
  - More overhead than threads
  - IPC serialization cost (data passed between processes)
  - Slightly more complex than thread-based queue model



Producer-Consumer Using multiprocessing.Queue (processes)

```
from multiprocessing import Process, Queue
import time
import random

def producer_mp(q, count=10):
```

```
    """
    Producer (process): produces `count` items and puts into queue q,
    then signals termination by putting sentinel (None).
    """
    for i in range(count):
        item = random.randint(1, 100)
        print(f"[Producer] Produced {item}")
        q.put(item)
        time.sleep(random.uniform(0.5, 1.5))
    # Send sentinel to tell consumer to stop
    q.put(None)
    print("[Producer] Done producing.")

def consumer_mp(q):
    """
    Consumer (process): fetches items from queue q and processes them.
    Exits when sentinel (None) is encountered.
    """
    while True:
        item = q.get()
        if item is None:
            print("[Consumer] Got sentinel, exiting")
            break
        print(f"[Consumer] Consumed {item}")
        time.sleep(random.uniform(1, 2))

def main_mp():
    q = Queue()
```

```
    p = Process(target=producer_mp, args=(q,))
    c = Process(target=consumer_mp, args=(q,))

    c.start()
    p.start()

    p.join()
    c.join()

    print("Main (multiprocessing): All done.")

if __name__ == "__main__":
                                                main_mp()
```

Concept Overview: Producer-Consumer via Redis

- **Producer**: sends tasks (strings) into a Redis list (queue) using LPUSH.
- **Consumer**: retrieves tasks using RPOP, processes them.
- **Redis list as queue**: using LPUSH and RPOP gives FIFO behavior (first in, first out).

Step-by-Step Implementation Using Redis

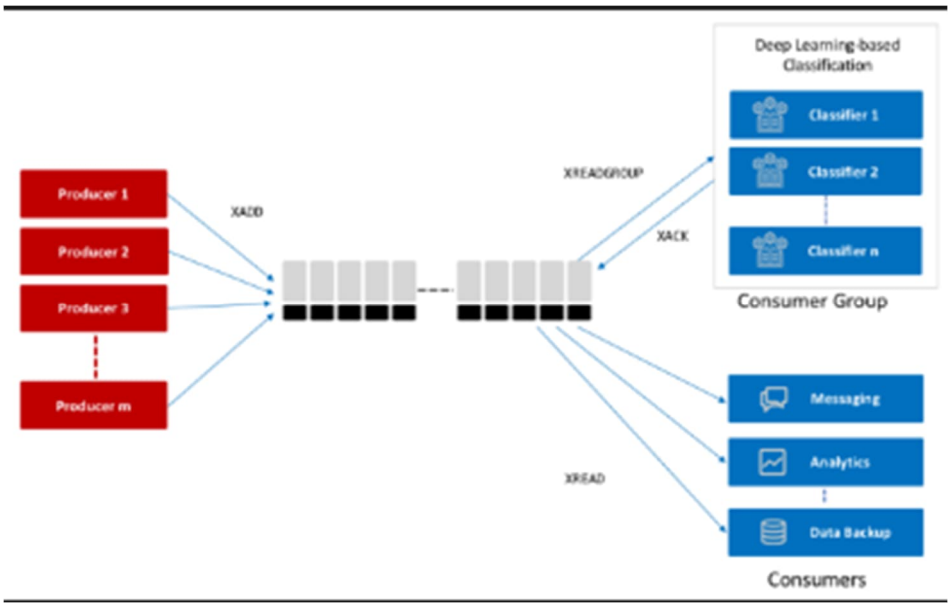
1. **Import Modules**
   redis for Redis client, threading, time, random for concurrency & delays.
2. **Configuration / Constants**
   REDIS_HOST, REDIS_PORT, QUEUE_NAME define where Redis is and queue key.
3. **Define Producer Function**
   - Connect: r = redis.Redis(...)

- Loop count times: create a random task string
- r.lpush(QUEUE_NAME, item) to add to queue
- Sleep to simulate production delay

4. **Define Consumer Function**
   - Connect same as producer
   - Loop forever: item = r.rpop(QUEUE_NAME)
   - If item not None: decode and process (print)
   - Else: queue is empty, wait, then retry
5. **Main Routine**
   - Create Thread for producer and Thread for consumer (daemon so it won't block exit)
   - Start both, join producer
   - Wait a bit, then exit

Advantages & Limitations

- **Pros**
  • Works across threads and across machines (since Redis server is networked)
  • Persistent queue (Redis stores it in memory, can be configured for persistence)
  • Decouples producer and consumer (they just need access to Redis)
- **Cons**
  • No built-in acknowledgement or retry mechanisms (if consumer fails after popping, you lose that item)
  • Need to manage queue empty states, retries, poison messages manually
  • Dependent on Redis server availability



Producer-Consumer Using Redis (LPUSH / RPOP)

```
import redis
import threading
import time
import random

REDIS_HOST = 'localhost'
REDIS_PORT = 6379
QUEUE_NAME = 'task_queue'

def producer_redis(count=10):
    """
    Producer: connects to Redis, pushes `count` items into the list queue.
    Uses LPUSH to push items to the left.
    """
    r = redis.Redis(host=REDIS_HOST, port=REDIS_PORT, db=0)
```

```
    for i in range(count):
        item = f"task-{random.randint(1, 1000)}"
        r.lpush(QUEUE_NAME, item)
        print(f"Producer: Produced {item}")
        time.sleep(random.uniform(0.5, 1.5))
    print("Producer: Done producing.")


def consumer_redis():
    """
    Consumer: connects to Redis, repeatedly pops items (using RPOP),
    processes them, or waits if queue is empty.
    """
    r = redis.Redis(host=REDIS_HOST, port=REDIS_PORT, db=0)
    while True:
        item = r.rpop(QUEUE_NAME)
        if item:
            # Redis returns bytes, decode to str
            s = item.decode('utf-8')
            print(f"Consumer: Consumed {s}")
        else:
            # No items now — wait, then check again
            print("Consumer: Queue empty, waiting...")
            time.sleep(1)


def main():
    t_prod = threading.Thread(target=producer_redis, args=(10,))
    t_cons = threading.Thread(target=consumer_redis, daemon=True)
```

```
    t_cons.start()
    t_prod.start()

    t_prod.join()
    print("Main: Producer finished. Waiting a bit more for consumer...")
    # Let consumer run a bit — or you may add a mechanism to stop it
    time.sleep(5)
    print("Main: Exiting program.")

if __name__ == "__main__":
    main()
```