The Producer–Consumer Problem

> “Two threads walk into a buffer. One adds, one removes.
> Chaos ensues — unless you synchronize.”

💡 Scenario

- One or more **producers** generate data (items, tasks, messages).
- One or more **consumers** process that data.
- They share a **finite buffer**.

We must ensure:

1. Producers don’t overflow the buffer (no overfilling).
2. Consumers don’t underflow it (no empty reads).
3. Access is synchronized.

🧩 Imports and Setup

```
import threading, time, random

BUFFER_SIZE = 5
buffer = []
lock = threading.Lock()
```

⚠️ Attempt #1 – The Naïve Version

```
def producer():
    global buffer
    while True:
        item = random.randint(1, 100)
        buffer.append(item)
        print(f"Produced {item} (size={len(buffer)})")
        time.sleep(random.uniform(0.2, 0.6))

def consumer():
    global buffer
    while True:
        if buffer:
            item = buffer.pop(0)
            print(f"Consumed {item} (size={len(buffer)})")
        time.sleep(random.uniform(0.3, 0.7))
```

💥 **Race conditions** everywhere!

🧱 Attempt #2 – Add Locks (But Poorly)

```
def producer():
    while True:
        with lock:
            item = random.randint(1, 100)
            buffer.append(item)
            print(f"Produced {item} (size={len(buffer)})")
        time.sleep(random.uniform(0.2, 0.6))

def consumer():
    while True:
        with lock:
            if buffer:
                item = buffer.pop(0)
                print(f"Consumed {item} (size={len(buffer)})")
        time.sleep(random.uniform(0.3, 0.7))
```

✅ Prevents simultaneous access

❌ Still **no coordination**:

🚦 Attempt #3 – Use Semaphores (Bounded Buffer)

```
empty = threading.Semaphore(BUFFER_SIZE)
full = threading.Semaphore(0)

def producer():
    while True:
        item = random.randint(1, 100)
        empty.acquire()         # wait for an empty slot
        with lock:
            buffer.append(item)
            print(f"Produced {item} (size={len(buffer)})")
        full.release()          # signal that buffer has an item
        time.sleep(random.uniform(0.2, 0.6))

def consumer():
    while True:
        full.acquire()          # wait for an item
        with lock:
            item = buffer.pop(0)
            print(f"Consumed {item} (size={len(buffer)})")
        empty.release()         # signal that slot is free
        time.sleep(random.uniform(0.3, 0.7))
```

⚙️ Main Function (for all working versions)

```
def main():
    threads = []
    t1 = threading.Thread(target=producer, daemon=True)
    t2 = threading.Thread(target=consumer, daemon=True)
    threads.extend([t1, t2])

    for t in threads: t.start()
    time.sleep(10)
    print("Simulation complete.")
```

Run it once and watch alternating “Produced” and “Consumed” messages.

Attempt #4 – Pythonic Queue (Modern Approach)

```
from queue import Queue

q = Queue(maxsize=BUFFER_SIZE)

def producer():
    while True:
        item = random.randint(1, 100)
        q.put(item)  # blocks if full
        print(f"Produced {item} (size={q.qsize()})")
        time.sleep(random.uniform(0.2, 0.6))

def consumer():
    while True:
        item = q.get()  # blocks if empty
        print(f"Consumed {item} (size={q.qsize()})")
        q.task_done()
        time.sleep(random.uniform(0.3, 0.7))
```

🧠 Summary Table

Approach	Race Safety	Busy Waiting	Bounded	Ease	Notes
Naïve	❌	😵	❌	Easy	Everything wrong
Lock Only	✅	😬	❌	Moderate	No coordination
Semaphores	✅	✅	✅	Medium	Textbook OS
Queue	✅	✅	✅	🤓	Idiomatic Python

🧩 OS Concepts Mapped

OS Concept	Analogy
Producer	I/O device writing data
Consumer	CPU reading/processing
Buffer	Shared memory or disk queue
Semaphore	OS kernel synchronization primitive
Queue	Abstracted thread-safe resource manager

⚖️ Discussion Prompts

1. What would happen if we reversed `full` and `empty` order?
2. Why is it important that the semaphores are initialized correctly?
3. How would this differ if there were **multiple producers** or **multiple consumers**?
4. What happens if the consumer processes much slower than producer?

🏁 Takeaway

> “The Producer–Consumer problem isn’t about milk and cookies.
> It’s about **respecting shared space** and taking turns like civilized threads.”